Contents

Chapter 0

Intro

Positions and rotations in the Unity3D game engine are about the same as they are anywhere else. One of the nice things about Unity is it steals the standard stuff everyone else uses, and doesn't need to make up many custom shortcuts. Learning Unity is close to learning the "real way" and vice-versa.

But this is about specific commands and how the numbers work in Unity, with sample programs as Unity `C#` scripts. The coding will be pretty basic, and limited to rotation stuff. For example, I'll show you how to make a simple y-spin, with an example where y increases. If you want y to bounce between 90 and 270 – that's just regular programming, so not mentioned here.

0.1 Review of basics

Before we start moving and rotating, it's nice to have a review of 3D basics and the rules Unity uses for them.

xyz axes: There are a few ways these can work. In Unity, y is up/down, which means the ground is x&z. X is left/right, Z is forwards/backwards. Or, think of x as east and west, and z as north and south. A "front view" in Unity (where the ground is on the bottom) is an x/y screen (sideways and up,) with +z going away from you along the ground.

Positive/negative goes the way you'd expect: +y is up, +x is right/east and +z is forward/north.

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're using a game board, 1 unit = 1 square might be nice, or some other round number. Like making each square is 9x9 might make it easier to size small, medium and large pieces.
- If the game is real, like in a house or outside, 1 unit = 1 meter (or 1 yard or 1 foot) might be good.
- If you get 3D models from other places, the unit scales are usually all different – a 3 unit tall cow with a 350 unit tall barn. You'll have to resize them, no matter what units you use.
- It's possible to not even know the scale. You might position things in a play area, make the game, and it just so happens the area is 27.3 units wide. You don't need to know what they stand for to know half of that is the middle.
- Officially, 1 unit is 1 meter. But not really. That only matters if you use the preset value for realistic gravity, and no one does.

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 when the code does it, which involves setting numbers, you need to know this stuff.

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 down between the feet. That makes it so you can compute a spot on the ground, place the cow there, and the body will automatically go above the ground. If the cow's origin was centered, placing it on the ground would put half of it below-ground, like that Cube.

Each object has its own origin, which is decided when it was created. You can't change it later (but you can use the parent trick to fake-change it.) Imagine you're creating the cow. You'd find (0,0,0) in that program, then build the cow either around it, or up from there. Where-ever (0,0,0) is in that cow you made, which could be anywhere, that's the origin when someone uses it.

We don't really care about origins yet. We'll be using just a small Cube or Sphere that represents a point. If our code makes us go out to 10, our center should hit there, half will hang past, but we'll know it worked.

It's just something to be aware of. If you try the later stuff with a model you imported, the origin will track your numbers exactly, with the rest just being where-ever.

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 in all directions, 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 for us. Just if you use an imported model and it appears to rotate funny, it's simply a non-centered origin.

For turning (y-rotations,) Unity thinks 0 degrees is facing forward, along +z, and positive degrees spin clockwise. 90 degrees is facing right/east.

In general, Unity thinks (0,0,0) rotation should be facing north, head up (like a cow standing on the ground.)

The rule for direction, for all 3 axis, is that Unity uses 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.

The biggest surprise is z. Grabbing z with your thumb pointing forward is awkward - maybe pretend you're grabbing a railing next to you. Whether you grab it from the top or bottom, your left-hand fingers will be rotating counter-clockwise. If you spin on z to tip the cow, you'll rotate to the *left*. Tipping to the right requires a negative rotation.

If you know real trig, everything about Unity's system seems wrong. Unity does have built-in sin, cosin and so on, and they use 100% correct trig. But the

rest of the Unity rotation system is degrees, clockwise, 0=north. If you decide to mix trig and Unity-rotations, you'll have to convert back and forth.

Problems with real 3D models

Again, using a small Cube or Sphere for testing is fine, but bringing in a real model is fun, and you eventually need to do it for a real game. There can be problems when you do.

Most 3D modeling programs flip the axes. They use z=up and x&y as the ground. No special reason - they just do. If you import a cow, it may be backwards and facing up.

You can manually spin it the right way. But suppose a programs sets the cow to spin from 0 to 360. The first thing it does is snap to 0, destroying your fix and giving a weirdly spinning cow.

That's the real problem with an x-facing cow. It seems perfectly fine - it's facing where it's facing, and it can easily spin to face anywhere else. But Unity thinks +z is forward, and some program will eventually attempt to face the left-side of your cow (which is +z for a +x facing cow) to the target.

Standard sizes of modeling programs are also funny. They are naturally in the hundreds, just because. An imported cow will often be so large that you're entire game is inside it. Scaling it down to size 0.01 can be a pain, and falls off when a program tries to shift it between size 1 and 2.

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. The advantage is it looks like you're not doing anything. cow1 can have no rotation and be size 1. The changes are safely hidden away in the child.

You can even do this again with the same cow. Parent cow2 can have the same cow with a different origin.

Chapter 1

Vectors and offsets

This first part is pure math involving xyz points. It will be very useful later, but for now i'm just showing the math rules.

Unity's xyz points are officially called `Vector3`'s. Adding them is done *pairwise*, which means x to x, y to y and z to z. This adds A and B to get C (the columns add together):

```
Vector3 A = new Vector3(2,  5, 20);
Vector3 B = new Vector3(2,  6,  7);
Vector3 C = A + B; //  (4, 11, 27);
```

Of course, it's just 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;`. But it's a nice shortcut, and we'll be doing it a lot.

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

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

You can multiply a Vector3 by a single `float`. It multiplies each part by that number. We call that a **scalar** since it scales the vector by that amount. Examples:

```
Vector3 A = new Vector3(2,  5, 20);
Vector3 B=A*2; //      (4, 10, 40)

B=3*A; //              (6, 15, 60);
```

The usual extras are allowed. `A+=B;` is the same as `A=A+B;`. You can use `A*=2;` to double every part of `A`. Dividing by a number is also a scalar – `B=A/2;` cuts every part of `A` in half.

Also as usual, the operators can be combined and mixed. Multiply goes before plus/minus. `A+C*3` triples everything in C, then adds pairwise to A. That will be useful, later.

Even though it seems like it would make sense, there's no pairwise multiplication: `A*B` isn't allowed. You rarely need it, and can do it the long way.

Unity provides several 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)
```

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.forward // (0,0,1)
```

It's common to use these shortcuts with math to make a vector. For example, `Vector3.one*7` makes (7,7,7). This might make your code a little easier to read:

```
A = Vector3.right*5; // (5,0,0)
A = Vector3.back*4 + Vector3.up*9; (0, 9, -4)
```

In the last one, you can think of it as saying "4 units back, then 9 units up." But just so you know, it's 100% the same as writing `new Vector3(0,9,-4);`

A common error is confusing Vector3's and single floats. Here are legal ways to add 1 to `x`, then some errors trying to do it:

```
A += Vector3.right;
A += new Vector3(1,0,0); // long version of Vector3.right
A.x += 1; // A.x is a simple float, so use simple addition

A += 1; // error - add 1 to which?
A.x += Vector3.right; // error
```

The last one is a common mistake. `A.x` picks out `x`, which is just a simple float. You can't cram `(1,0,0)` into it.

1.1 Points and Offsets

The first trick to using Vector3's for positioning is to think of them as either points or as offsets. `transform.position` is a point – an actual spot on the

map where you are. Offsets are more like arrows. They're designed to be added to points, and *don't* stand for spots on the map.

For example, suppose we're building a game board. We'll start by setting the lower-left corner position with an Inspector variable (I just picked some numbers.) In or minds, this is an actual point on the map:

```
public Vector3 cornerLL = new Vector3(3,0.5f,7);
```

The board squares will run across and up from there. To help place them I'll make offset vectors, which measure right and forward for 1 square:

```
public float sqSz=1; // width/length of all squares
Vector3 sqOver=new Vector3(sqSz,0,0); // arrow to move 1 square over
Vector3 sqFrd = new Vector3(0,0,sqSz); // arrow to move 1 square back
```

These clearly aren't points. We don't care about the actual position `(sqSz,0,0)`. In our mind these are arrows we can use to move from the lower-left point to the corners of other squares.

This code places a four sample squares (each `sqXX` is one little square):

```
sq00.position = cornerLL;
sq10.position = cornerLL + sqOver;
sq30.position = cornerLL + sqOver*3;
sq32.position = cornerLL + sqOver*3 + sqFwd*2;
```

In our minds, `cornerLL + sqOver` means to start at the `cornerLL` position and follow the sideways arrow `sqOver`. The next line uses vector math to walk 3 squares over from the corner. The last line is the same, but also goes an extra 2 up.

This is why we made those rules about pairwise and scalars. I think the code above looks like starting points and stretched arrows, which it is; but the math also works out properly.

Our squares probably have the origin in the middle. If we want `cornerLL` to actually be the corner we'll have to shift the squares by 1/2 a square sideways and forward. We can do that painlessly with another offset vector:

```
Vector3 cornerToCenter = new Vector3(sqSz/2, 0, sqSz/2);
```

Again, this is clearly not a point on the map we care about – it's an arrow that will take us from the LL corner of any square to the middle of it.

Placing our squares now adds that extra offset:

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

In our minds the last one starts at the board's corner, follows the arrow to the center of the lower-left square, then marches center-to-center 3 over and 2 up.

You might notice that we didn't need point-plus-arrows math for this particular problem. It would be just as easy to hand-add to `x` and up to `z` to get where the squares are.

But later, we'll see problems where point-plus-arrows is clearly the best.

1.1.1 More calculating offsets

A board lined-up on x and z isn't really showing off how nice offsets can be. Suppose we tilt the board and hand-enter three corners. Then we can use all arrow-math without even knowing what the numbers are.

Suppose someone enters these corners:

```
public Vector3 cornerLL, cornerUL, cornerLR; // lower-left, upper-left, lower-right
// assume someone enters reasonable #s for these

UL
 \    diagonal
     game board

     \       ---LR
      LL--
```

If the board is 8 across, each square is 1/8th of the way from left to right. We can compute that:

```
Vector3 sqOver = (cornerLR-cornerLL)/8;
```

That's standard arrow/offset thinking. When you subtract one position from another, you get an arrow going between them. `cornerLR-cornerLL` is an arrow going from the left side of the board, to the right side. `sqOver` is 1/8th of that, so it perfectly lines up with the bottom of one square.

How long is it? What are the exact x and z numbers? Since the board is diagonal, what's the slope? The cool thing about using arrow math is we don't need to worry about that stuff.

Placing a square is the same as before. We use `sqOver` and `sqrFwd` to walk from the LL corner to the right position:

```
Vector3 sqOver = (cornerLR-cornerLL)/8; // repeat from above
Vector3 sqFwd = (cornerUL-cornerLL)/8; // similar idea

// this is still half a square over and forward:
Vector3 cornerToCenter = sqOver/2 + sqFwd/2;

sq32.position = cornerLL + cornerToCenter + sqOver*3 + sqFwd*2;
```

This is finally computing `cornerToCenter` the correct way. In our heads it's 1/2 a square each way. The new code, `sqOver/2 + sqFwd/2` now says that. If you remember, I cheated before by just writing `(sqrSz/2,0,sqrSz/2)`. That won't work for a diagonal board and isn't as easy to read.

`sqOver/2 + sqFwd/2` is another typical vector math case where we don't know the exact `x` and `z` values, and don't need to know. Divide by 2 cuts the arrows in half, plus combines them, and that's good enough.

One more neat arrow trick: I assumed we were told just 3 corners, since that we all we needed for now. We can use those to compute the last corner, the upper-right. The plan is: get the arrow going up along the left side; then add that arrow to the lower-right corner:

```
Vector3 bottomToTop = cornerUL-cornerLL; // arrow up left-side
Vector3 cornerUR = cornerLR+bottomToTop; // move it to right side
```

This shows how arrows are made to be moved around. `bottomToTop` feels like the left-side arrow, like it should start from the lower-left. But it's really the side-arrow, which we happened to use the left-side to compute.

In our mental picture, adding it to the lower-`right` corner makes sense. If we needed a 1-board gap above us, we could even add it to the upper-left corner.

One last thing about this entire game board example: I cheated a little by saying someone had to enter three correct corners. Any two points would work for the lower-left and lower-right. But the upper-corner really should be that same line, rotated 90 degrees. Otherwise the board might be a funny trapesoid.

Later, we'll learn how to rotate a line. This example should really have you pick only the bottom 2 corners. So sorry.

1.2 `transform.position`+offset

We can use the point+offset trick using our current position as the point (and the offset as whatever we make up.) This seems more complicated, since `transform.position` is the point, and because it can move around, but the basic idea is simple.

For testing, our script has links to a red, green and blue cube (I'll assume you know how to make them and drag in links to Unity Inspector slots.) We'll be positioning these each frame using point+offset math.

This computes offsets a little differently for each cube, and positions them around us:

```
public Transform redCube, greenCube, blueCube; // linked to real cubes

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

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

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

}
```

Notice how the math for the offsets is nothing new. `Vector3.right` is the old shortcut for +3 on `x`. Setting `greenOff` just plugs in numbers the old way. For no reason, the blue cube re-uses the green offset, but 50% longer.

If you go to Scene view and drag yourself around (while running) they will track you. If you spin yourself, they won't spin with you, since why would they. The math only looks at our current position.

A little more advanced, suppose we have a fixed marker – the red Cube just sitting in one spot. We can use the "arrow between" and fraction-of-an-arrow tricks to place the green cube 1/2-way between us:

```
void Update() {
  Vector3 toMarker = redCube.position - transform.position;
  greenCube.position = transform.position + toMarker*0.5;
}
```

This is pretty slick. In Scene view we can drag ourself around, or the red Cube, and the green one will stay 1/2-way between us. It *looks* more complicated than it is, because of the way Update magically reruns each frame, but you can see the math part is just using a fraction of an arrow.

A little trickier, but not much, we can use the `toMarker` arrow to keep the blue Cube on the other side - it will always hide from the red marker, behind us:

```
void Update() {
  Vector3 toMarker = redCube.position - transform.position;
  greenCube.position = transform.position + toMarker*0.5;

  // new part:
  blueCube.position = transform.position + toMarker* -0.1f;
}
```

The new part is how "times -1" flips an arrow to go the other way. That's not a special rule – it's just how the scalar math works out. `toMarker*-0.1f` says to take the arrow going to to red marker cube, flip it, and take 1/10th of it.

Again, dragging around us or the red cube makes the blue one appear to always hide behind us (always at 1/10th the distance. Moving us closer and further changes it.) Blue seems smart, but we can see it's just 1 line, recomputed

each frame.

With just a little more code we can change that green cube from always being 1/2-way between to having it move along the line by itself. The idea is simple: we know we could use `toMarker*0.33f` to be 1/3rd of the distance to the red marker, or 0.8f .... Any fraction from 0 to 1 will put it at a spot along that line.

So let's change that `0.5f` to a variable running from 0 to 1 by itself. The vector math is exactly the same. This makes the green cube fly from us to the red, then snap back to us and repeat:

```
float pct = 0.0f; // we'll make this go from 0 to 1
public Transform red, blue;

void Update() {
  Vector3 toMarker = redCube.position - transform.position;

  greenCube.position = transform.position + toMarker*pct; // <- changed 0.5 to pct

  pct+=Time.deltaTime*0.5f;
  if(pct>1) pct=0; }
}
```

In the last part, I'm assuming you've seen how `deltaTime` is used and the line would look weird without it. It you haven't used it, it's the way to say "this much per second".

To make it fly the path a different way, change the last 2 lines. Going 1 down to 0 would make it fly towards us instead of away. 0 to 0.9 would make it stop a little short. But however we do it, the vector math part is happy to just use whatever percent we give it.

Errors. It's easy to forget the trick is point+offset. If you forget the starting point, you get errors. Not really errors, but things that don't look right. Suppose in the moving ball code above we forgot to use ourself as the starting position:

```
...
void Update() {
  Vector3 toMarker = redCube.position - transform.position;

  greenCube.position = toMarker*pct; // <- oops. Forgot to add the starting point
  ...
}
```

That wrong code will make the ball fly from point (0,0,0) in the direction from us to the ball (which is not towards the ball - it's using the arrow from us, slid over, which just looks funny.) If it doesn't look wrong yet, move things

around in Scene view. It tracks the angle and distance, but from the wrong spot.

Logically, an arrow always starts from some point we supply. If we add nothing it's like we told it to start from (0,0,0.)

Another non-error error is getting the arrow direction flipped around. You make an arrow using end-point minus start-point: `B-A` creates an arrow pointing from A to B.

If our code used `transform.position - redCube.position` we'd have an arrow starting at the red cube, going to us. That's fine, but we'd need to remember to start at the red cube:

```
...
void Update() {
  // a perfectly good arrow going from the red cube to us:
  Vector3 redToUs= transform.position - redCube.position;

  // correct use:
  greenCube.position = redCube.position + redToUs*pct;

  // wrong: puts green cube on other side of us, going away from red:
  greenCube.position = transform.position + redToUs*pct;
  ...
}
```

The last line isn't really wrong - earlier I wanted the blue cube to hide behind us like this does. It's mostly an example of how you need to decide and remember which way your arrows go, and whether it makes sense to have a particular arrow start from various spots.

1.2.1 Camera vector positioning

A fun trick is positioning a camera using offsets from us. It's the same as positioning a Cube near us, but with the camera it feels like a different thing, and gives an excuse to use different math.

To test, hand-moving in Scene view won't work anymore. We'll see the little camera icon move, but seeing through the camera in game view is what makes it cool. So here's some very simple movement code for testing:

```
void Update() {
  if(Input.GetKey("w")) transform.Translate(0,0, 2*Time.deltaTime); // forward
  if(Input.GetKey("a")) transform.Rotate(0,-1,0); // spin left
  else if(Input.GetKey("d")) transform.Rotate(0,+1,0); // spin right
}
```

All it does is turn with the A and D keys, and move forward with W. If you have other movement code, it would work just as well. To really test, we also need some landmarks – like terrain or walls or anything to be able to tell we're moving.

With that out of the way, now we can position a simple camera. This puts it above us and a little ahead. It doesn't change the angle, so you'd need to hand-spin it to face downward:

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

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

The same with the colored cubes, we can spin but the camera won't. It just tracks us as we move.

To make a simple zoom, we could think of one arrow from us to the closest camera position, then another going from there to the furthest camera position. That second arrow will use the multiply-by-0-to-1 trick to zoom the camera in and out:

```
public float camPct01; // the zoom amount, from 0 (nearest) to 1 (furthest)

void Update() {
  Vector3 toCam1 = Vector3.up*10;
  Vector3 toCam2 = Vector3.up*10 + Vector3.forward*3;

  theCamera.position = transform.position + toCam1 + camPct01*toCam2;
}
```

That last line should look pretty familiar: start at a point (which happens to be us,) follow an arrow from there, then follow a fraction of another arrow.

A fun way to change the 0-1 zoom percent, which has nothing to do with vector math at all, is to say it zooms to 0 as we move, then pulls back when we stop:

```
void Update() {
  bool isMoving=false;
  if(Input.GetKey("w")) {
    transform.Translate(0,0, 2*Time.deltaTime); // forward
    isMoving=true;
```

```
  }
 ...

  // zoom in if moving, out if staying still:
  if(isMoving) { camPct01-=Time.deltaTime; if(camPct01<0) camPct01=0; }
  else { camPct01+=Time.deltaTime; if(camPct01>1) camPct01=1; }

  // same camera position as before:
  Vector3 toCam1 = Vector3.up*10;
  Vector3 toCam2 = Vector3.up*10 + Vector3.forward*3;

  theCamera.position = transform.position + toCam1 + camPct01*toCam2;
}
```

1.2.2 Moving ourself

This is only mildly amusing, but we'll use it later. The old movement from before cheated by using Unity built-ins. We can move using point+offset math. We're really just computing a point nearby, then placing something there - ourself.

This moves us slowly diagonal:

```
void Update() {
  // at an angle right and a little forward:
  Vector3 mvDir = Vector3.right*3 + Vector3.forward;
  // tweak for 60 times/second:
  mvDir *= Time.deltaTime; // scalar shortcut

  transform.position = transform.position + mvDir;
}
```

Notice how we go in a straight line. We made the arrow going 3 right and 1 forward, but the final result is all that matters. There's no way the parts can "carry through" and make us move directly 3 right, then 1 up after that.

And, same as always, it doesn't care which way we're facing.

1.2.3 Averaging points

Averaging two points works. It gives you a point exactly between them. Ex: `Vector3 middle = (A+B)*0.5f;`. This works even if `A` and `B` aren't lined-up at all. The answer is midway on a line between them.

Real examples of this are `bottomMiddle = (cornerLL+cornerLR)*0.5f;` to find the center-bottom of the board. Even sneakier, the center of the board is the average of two diagonal corners: `Vector3 center + (cornerUL+cornerLR)*0.5f;`.

Writing the math as an average is easy to read, if all you need is the middle. But it's just a shortcut for point+offset-fraction:

```
Vector3 across = cornerLR-cornerLL; // arrow from left side to right
Vector3 bottomMiddle = cornerLL + across*0.5f;
```

Then you can adjust the 0.5 to whichever fraction you need.

1.3 Math review

The stuff above was really just examples. Just in case, sometimes it's nice to see the rules for using and combining points and offsets written out, all in one place:

- 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. `(A+B)/2` is an exception.
- For an offset `-A` is like flippping the arrow to point the other way.
- It looks better to have the point come first: point+offset. But obviously you can flip the order and the math will be the same.

I think of these rules when things break. Suppose you have `sq.position=A+B+C;` and it's working wrong. Check you have 1 point and 2 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?"

Chapter 2

Local and Global axes

2.1 Local axes

When we have a rotated object, it might be handy to imagine it has a personal x, y and z. The official name for these are its local axes.

When we want to talk about the real x,y,z, we call those the global or world axes.

To see both, select some object with a spin, go to top of the Scene window, select the Translate tool (shows red/green/blue arrows) and change the Global/Local toggle at the top of Scene. You'll see them change from the real x, y, z's to your personal ones. If you spin yourself, Global mode keeps your arrows locked, and Local mode has them spin with you.

This feature is standard on any 3D program, not just games.

The purpose of local axes is just to be more arrows we can use. If we want an arrow pointing the way we're looking – that's our local +z axis. Having them can complicate things – whenever you use an axis you have to think global or local. But they're worth it.

2.2 Using your local axis in code

`Vector3.right` is the *world* x-axis. Each object has its own personal local axes, since everything can be rotated a different way. You have to ask an object for them with `transform.right`, `transform.up` and `transform.forward`.

Those give us the exact red, green and blue arrows we see in local translate mode. We didn't think of this much before, but they're always length 1. As we spin around, the exact xyz numbers making those arrows will change, but the length works out to 1.

Fun fact: 3D systems match RGB colors with xyz: the x arrow is always red, y is always green, z is blue.

The opposite directions don't have a built-in – there's no `transform.left`, but that's not a problem since we know scalars. `-transform.right` flips it to go left.

We can use these as regular arrows. Here's a silly one putting a red cube 3 spaces to our right. If you spin yourself, the dog will spin with you:

```
public Transform redCube;

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

The magic part is how the system automatically recomputes `transform.right` as we spin, so it's always our local +x. Besides that, our code is just the same old point+offset math.

We can combine these arrows, the same as before. This puts the red cube 2 in front and 1 up (spinning with us):

```
void Update() {
  redCube.position=transform.position + transform.forward*2 + transform.up;
}
```

A fun trick with local axes is making us move the way we're facing. This adds part of our forward arrow to ourself each update :

```
void Update() {
  Vector3 mv = transform.forward*Time.deltaTime;
  transform.position += mv;
}
```

If we press Play and spin ourself a little in the Inspector, we'll see this really moves the way we're facing. Again, the magic is how the computer looks at our spin and auto-computes `transform.forward`.

The same local-axis math can be used to shoot a ball. This uses standard placement with an offset to start the ball in front of us. The new part is using our forward vector to set the ball's speed (it also assumes you know how to use Unity prefabs, and rigidbodies):

```
public Transform ballPrefab;

void Update() {
  if(Input.GetKeyDown(KeyCode.Space)) {
    Transform bb = Instantiate(ballPrefab);
```

```
    Vector3 toBall = transform.up*1.0f + transform.forward*2;
    bb.position = transform.position + toBall;

    Vector3 ballMove = transform.forward*5; // medium-slow the way we're facing
    bb.GetComponent<Rigidbody>().velocity = ballMove;
  }
}
```

The velocity of an object is just an offset – it's how it moves during 1 second. We set it once and the system figures out how much to move each frame so it adds up. `transform.forward*5` shoots the ball at 5 units per second, going the way we're facing.

One note: rigidbody's start with gravity turned on. You'll see the ball fly straight, then quickly curve away as it falls. If you want to see it fly exactly straight, find the UseGravity checkbox in the ball prefab's rigidbody and uncheck it.

We can use the usual vector math to play with the velocity arrow. This fires two balls, from our left and right, and angles them both a little inward, so they crash in front of us:

```
void Update() {
  if(Input.GetKeyDown(KeyCode.Space)) {
    Transform bLeft = Instantiate(ballPrefab);
    Transform bRight = Instantiate(ballPrefab);

    bLeft.position = transform.position + transform.left*2;
    bRight.position = transform.position + transform.right*2;

    Vector3 fmv=transform.forward*5; // forward speed, but not done yet

    // angle them both a little inward:
    bLeft.GetComponent<Rigidbody>().velocity = fmv+transform.right;
    bRight.GetComponent<Rigidbody>().velocity = fmv+transform.left;
  }
}
```

I like this example since it shows how the rules are always the same. Once we know `velocity` is just a vector offset, we can use regular offset rules. If the left ball has velocity 5 forward and 1 right, that's a regular slightly diagonal arrow, same as any other.

Local axis work just as well with moving numbers. Adding `-transform.up+transform.forward*-3` to our position gives a point a little below and 3 units behind us. Changing the -3 to -2, -1 ... moves it in closer. As usual, we can make it move by turning that into a variable. I'll have it move from -3 to 5:

```
public Transform redCube;

float dist=-3; // will go from -3 to 5 (then wrap around)

void Update() {
  Vector3 toRed = transform.up*-1 + transform.forward*dist; // <- dist changes
  redCube.position = transform.position + toRed;

  // moves dist from -3 to 5:
  dist+=2*Time.deltaTime;
  if(dist>5) dist=-3;
}
```

As we spin, the track the red cube takes will follow us.

The previous version of this had one long arrow (from the fixed cube to us) and used 0 to 1 percents to move along it. For this it seemed easier to use the length 1 `transform.forward` arrow and let the numbers stand for actual distances. Either way is fine, depending.

We can use other people's local axes. `redCube.forward` is the direction the red cube is facing. This is a simple example using our forward and its forward. We move in ours, and move the red cube using its:

```
public Transform redCube; // spin so it has a different facing than we do

void Update() {
  transform.position += transform.forward*1*Time.deltaTime;

  redCube.position += redCube.forward*0.8*Time.deltaTime;
}
```

This next one uses everyone's local axes to make a chain of cubes. red goes in front of us, green goes in front of red (based on how *red* is facing) and blue goes in front of green. Twisting any cube will shift the ones in front of it:

```
void Update() {
  redCube.position = transform.position + transform.forward*2;
  greenCube.position = redCube.position + redCube.forward*2;
  blueCube.position = greenCube.position + greenCube.forward*2;
}
```

There's no rule that `greenCube.forward` has to start at `greenCube.position`, but it's an example of the "where would that arrow make sense" idea. You could add green's forward arrow to the red cube, but it's hard to think of a good reason.

2.3 Things that assume local coords

Using local axes is so handy, some commands assume you want to use them. For example, the built-in translate command assumes you're thinking of local coordinates:

```
transform.Translate(0,0,1); // local +1z
// same as:
transform.position += transform.forward;

transform.Translate(2,3,0); // +2 to your right, +3 local up
// same as:
transform.position += transform.right*2 + transform.up*3;
```

I used this shortcut in the previous chapter in the "spin and move the way you're facing" testing movement code.

The key thing is that `Translate` is just a different word for changing your position. If you're mostly a 3D modeler, and are comfortable using the translate tool to move things, and know the Local setting is often the most useful, then this command makes perfect sense.

But if you know vector math there's no reason to ever use this. Vector math, `=` and `+=` is more flexible.

Pushing a rigidbody (increasing the speed) has an option for global or local. Suppose you have a 3D model with a facing, like a cow, and `rb` is its rigidbidy. These two commands shove it in different directions:

```
rb.AddForce(0,0,4); // cow starts flying north (real +z)
rb.AddRelativeForce(0,0,4); // cows starts flying direction its facing
```

The first one is a shortcut for adding 4 to our z-speed. Math in world coords is simple like that. Written out, it's the same as:
`rb.velocity += Vector3.forward*4;`
The second one isn't much worse. It looks up the cow's local forward:
`rb.velocity += cow.forward*4;`

The word `Relative`, in `AddRelativeForce` is just an informal way of saying Local. When I see `AddForce`, I wonder whether it's local or global, since either would make sense. But then seeing the other one, I figure "since `AddRelative` is local, just `AddForce` must be global."

Besides *relative*, Unity has some more words that are just informal ways of saying local and global. The translate command, from before, has local and global options:

```
transform.Translate(0,0,1, Space.Self); // local - moves my forward
transform.Translate(0,0,1, Space.World); // global - +1 on real z
```

It sounds nice to say "move me 0,0,1 in my local space," but it's just another way of saying to use your local axes. Likewise "world space" sounds nice enough, but it means "using the global axes."

And there's still no reason to use these if you know vector math.

To sum up: you'll see various commands with two options, probably using odd words. They're always just a choice between global or your local.

2.3.1 Looking at the numbers

This section is for if you're feeling a few things aren't quite right, and want some more explanation. If that's not true, you can safely skip ahead.

"For real," Unity uses global axes for everything. Positions are only ever stored using the real x, y, z, and movement can only be done using global coordinates. `transform.forward` doesn't really tell the system to use your local forward. It does, but the way it does is by recomputing what your local forward would be into global coordinates.

A way to look at it is like this:

```
Vector3 arrow = transform.forward*2 + transform.right;
redCube.position = transform.position + arrow;
```

`arrow` was created by thinking about our local forward and right, but then it's just a regular vector3. When we add `arrow` in the second line, the system doesn't know or care how it got made.

You don't need to see the numbers, but it might make things clearer. Here's a simple program to show 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;
}
```

You should be able to select the object, press Play, hand-spin it in Scene view, and watch how those variables change.

With no rotation, these should look the same as global – `right` will be (1,0,0), `fwd` will be (0,0,1).

If you spin to face backwards `fwd` is flipped to (0,0,-1) and `right` will be flipped to (-1,0,0.)

Rotating to 45 degrees, so you're facing north-east, shows `fwd` as (0.71, 0, 0.71). X and z are equal, which seems right. The numbers look funny, but the length really is 1. You might recognize them as the sin and cosine of 45.

If you spin 45 degrees and then tilt back 45 on x, so your forward sticks up and northeast, it becomes (0.58, 0.58, 0.58). All three the same seems correct. And it really is length 1. They follow the hypotenuse rule: square them and add them up and you get 1.

Again, you don't need to know the numbers. But if you know there has to be trigonometry somewhere, and it was bothering you where, this is where. Or if you were wondering how our `transform.position` can store local and global together – now you know it only stores global. And if you see some of the numbers for local axes and they look odd – they're sins and cosines and are supposed to be odd.

2.4 Childing

We know that child objects lock onto their parents, tracking them as they move and spin. The way that's accomplished is by using the parent's local coordinates. If we play with children a little we might get a better feel for local axes and "local space."

You may have noticed that when you go to the Hierarchy panel, and drag one thing into another to make it be a child, the numbers for Position suddenly snap to new values. The object hasn't moved. The change is because it's now telling you the position in the parent's local coordinates.

The slot still says Position, but it's not. Here are some fun things to try with a child:

- Enter (0,0,0) for the position (which is really the to-parent offset.) The child will snap to the parent. That should make sense. 0 away from it in all directions is exactly where the parent is.
- Use the Inspector variable slide trick (mouse to in front of a box and the cursor changes to a double-arrow, letting you know you can slide the value.) If you slide z back-and-forth, the child moves along the parent's forward/backward.

  Try spinning the parent and trying it – changing z moves on the parent's new local z axis. Likewise with x and y. Sliding the child's x, y or z traces out each of the parent's local axis.
- Enter something not too hard like (2,0,0) for the child's position (it will move) and spin the parent. We already knew the child would spin with it, but now we can check the math. In any new spin we can find the parent's red x-local-arrow, and to will be pointing to the child, 2 spaces away.

- Drag a child from one parent a different parent. The numbers will snap again. Now it's in local coordinates from it's new parent.

  This is another way of seeing that, obviously, each object has it's own local space.

The system always remembers your real-world position. For children it also remembers an extra set – local from the parent. In code, we're allowed to use and set both, and the system updates the other. Suppose `cc` is a link to one of our children:

We can use `cc.position=` the same as always. It will move the object to that real-world spot. It recomputes the local position to the parent, based on where we put it.

The new fun part is `cc.localPosition` lets us directly set the from-parent numbers. It's a way to use raw local coordinates. If `redCube` is our child, this will put it directly in front of us:

```
public Transform redCube; // this is one of our children

void Start() {
  redCube.localPosition = new Vector3(0,0,2); // snaps to 2 in front of us
}
```

The back part fools you, since it's just raw (0,0,2). The key part is `localPosition`. It knows it's value is local coordinates from the parent. It's a little like how `transform.Translate(0,0,2)` knows to move in your forward..

Here's move-a-ball-under-me-from-back-to-front, rewritten to use a child and localPosition. It's a little simpler than before:

```
public Transform redCube; // assume this is a child of us

float zz=-3; // moves from -3 to 5

void Update() {
  redCube.localPosition = new Vector3(0, -1, zz);

  // same code as before, moving zz
  zz+=2*Time.deltaTime;
  if(zz>5) zz=-3;
}
```

Of course, using a child and `localPosition` to place objects is just a shortcut. For my child `redCube`, these two things are the same:

```
void Start() {
  redCube.localPosition = new Vector3(0,0,2);
```

```
  redCube.position = transform.position + transform.forward*2;
}
```

The thing is, they really *are* the same. In the first one, the system knows to add my position and to covert +2z parent-local into real coordinates. In the second, I do that math myself. The work is the same either way – making a child is just a time-saver.

The whole idea of "my local coordinates with me at (0,0,0)" is called Local Space. The easy way to explain what children's position numbers means is they're in the parent's local space. This isn't a new rule – just a shorter way to describe some things. If we're just using `transform.forward`, we'd call it your local axis. But if we're doing a full xyz position from us, it's nicer to say "in my local space." We'll see more tricks with it, much later.

2.4.1 `SetParent` notes

This section is a "while we're on the subject" thing. Unity has some interesting rules for setting your parent in code, and we may as well see them while we're on the subject. But there's nothing new here about local or vector math.

The basic command to make something your parent works the same as doing it by hand: nothing moves now, but you'll track your new parent. A simple use is if an arrow sticks into something. The arrow has hit, it's where it should be, and you glue it there in case the target starts moving:

```
arrow.parent = targetStuffedWithHay;
// or:
arrow.SetParent(targetStuffedWithHay);
```

To un-parent, set the parent to nothing: `arrow.parent=null;`. All normal objects are walking around with `null` for their parents.

Often you want to create a new child in a certain position. It's almost always easiest to combine steps: make it, child it, position it:

```
Transform newDogChild = Instantiate(dogPrefab);
newDogChild.parent = transform;
newDogChild.localPosition = new Vector3(0,2,0);
```

There's one rare oddball use of `SetParent` that you almost never need. It's a hack that lets you pre-set what you want your local position to be. You put you future local position in your position. Then the `setParent` command does all the work – adding `false` tells it to copy the current position into the new local position. It looks like this:

```
Transform newDogChild = Instantiate(dogPrefab);
// fake setting position. This is really our future local position:
newDogChild.position = new Vector3(0,0,2);

// "false" means to also copy our real position to be our new local position:
newDogChild.SetParent(transform, false);
```

The purpose for this is prefab UI elements – buttons and sliders. They're always children of a canvas or canvas panel, and they have a lot of tricky local settings in their RectTransform. This trick lets you preset local settings in a button prefab. When you Instantiate and child this way, you get the local settings you worked so hard to make.

But there's a better way. The instantiate command has an option to create something directly into the parent:

```
Instantiate(button1Prefab, canvasPanel2, false);
// false means to copy button1's position into it's localPosition
```

This is pretty nit-picky specific Unity stuff, but it shows a little more how you have to have a local position from something

Bonus terms: computer science and unity-slang use parent/child verbs backwards. In computer science (which is how many game programmers started out,) you use the word for what you are: `A.parent=B;` is *childing* A, or "childing A to B," or possibly "parenting B to A" (that's more awkward, but everyone would know what you mean.)

You'll often read Unity users writing how `A.parent=B;` *parents* A to B. I the computer science guys have been doing this longer and know the best way to say it. Be aware you might see it both ways.

2.5 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 did made a mistake.

If you incorrectly use `Vector3.up` instead of `transform.up` you might not notice for a while. You often only spin. Your left, right, forward and backward are all changing but your up is the same as world up.

Much later you might tip sideways and `Vector3.up*2 + transform.right` suddenly starts giving funny results, since it should have been `transform.up` the entire time.

If something expects local coordinate then using `transform.forward` gives wrong results. For examples, using it in localPosition or Translate:

```
child.localPosition = transform.forward; // ick. use Vector3.forward
transform.Translate(transform.forward); // ick. same.
```

The problem is this double-converts. In your mind you had "local (0,0,1)." These two commands like local and will convert it. `transform.forward` is only for when the comand doesn't like local and you need to hand-convert to global yourself.

Another example is with AddForce. this first one works the way you think. AddForce takes global xyz and `transform.forward` gives globals:

```
rb.AddForce(transform.forward*6); // fine, since this takes global axes
```

But this double convets:

```
rb.AddRelativeForce(transform.forward*6); // not good. double-converts
```

The point of using the `Relative` version is so we can write just (0,0,6).

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. Before the best we could do was 1/10th of an arrow, which could grow and shrink.

An earlier example had a cube hiding behind us. The same math works here:

```
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 the marker to us. We did that before, but now it moves at a constant rate (before, it took 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() {
  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+=2*Time.deltaTime;
  if(ballDist>len) ballDist=0;

}
```

If you remember, the old way used the entire arrow, and the variable was a 0 to 1 percent. This way, the variable is the real distance we want to be. Neither way is better, but this one often looks nicer. If we're 8.5 away from the marker, we have to wait while the distance slowly goes from 0 to 8.5.

The unit vector trick is also great for setting velocity. If you remember, I previously used `velocity=transform.forward*5` to fire a ball. That was really using the unit vector trick – relying on how `transform.forward` is always length 1.

If you want to shoot in a direction towards something you can do it by manually computing the unit vector. This has the space key shoot a ball towards a marker. It starts 2 in front of us and flies at 5 units/second:

```
if(Input.GetKeyDown(" ")) { // space key fires
  // standard spawn. Assume ballPrefab is set up:
  Transform ball = Instantiate(ballPrefab);
  ball.position = transform.position + transform.forward*2

  Vector3 toMarker=(marker.position-ball.position).normalized;

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

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

If you're turned away from the target this will shoot the ball back through you. And, same as before, it works better if gravity is turned off on the ball prefab.

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. A later chapter is about how to add them, take fractions or spin towards an angle instead of just snapping.

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. Remember Unity thinks +z is forward, and no rotation is looking directly north, flat along the ground.

If you have a 3D cow or gun pointing along +z, that will be fine. I like to make a testing object: make a sphere. Then add a cube above it for a hat – child the cube to that sphere and adjust local position to (0,1,0). Then add a forward arrow – child a another cube, scale it along z and place it in front:

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

A decent way to think about a rotation is which way it points us. In other words, where our local +z faces. That's part one. Part two of a rotation is the local roll on z, which won't change where we're aimed. For my little guy, it would roll the hat around.

Or, same idea, point your arm somewhere with your thumb stuck straight out. Then you can angle your thumb in a circle by rolling your arm.

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":

```
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 shows us the 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, so uses a temp 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 make rotations in different ways. Setting the x, y, z's for degrees is fine for some things. Other times we start out knowing a certain spot and want to aim ourself at it. Sometimes we want to spin around some diagonal line. Other times we want a rotation "offset" – the extra rotation to change from facing A to facing 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

We can set rotations is using x, y, z degrees. It works like aiming an airplane: heading, climb/dive and roll. Or, the same, thing: aiming the gun of an army tank.

First we spin to face a compass direction, then we angle up/down.

Math-wise, this is a rotation on global y, then local x.. That sounds funny, but the part where we angle up/down has to be on our local x, after spinning, or it would be really hard to use.

An example: (-45, 90, 0) says to spin 90 degrees first, spinning from north to east, or 12 o'clock to 3 o'clock. Then tilt up 45 degrees, still facing east (the left-handed coordinate system means -x is up. That can be a pain to remember.)

The last part of rotation is a local z-roll (spinning your thumb.)

It seems funny, but the correct order is global y, local x, local z. Y then x aims you, then a free roll. This is how Unity reads the xyz rotation numbers in the Inspector.

It can help to play with it. Take the long-nose ball with-a-hat and spin it around. Spinning y always gives a perfect compass spin. Spinning z always just rolls the hat around. Spinning x always cranks it up and down without changing the compass direction (unless you completely flip over.)

Even if you do them in a funny order, like x, y, more x, some z, back to y . . . , Unity reads the final results in the order y, local x, local z.

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.

They have the same meaning as in the Inspector, This points you left (-90 y) and a little bit up (-10 x):

```
transform.rotation = Quaternion.Euler(-10, -90, 0);
```

Remember it's not simply setting the x, y, z degrees. It says to make a quaternion, using the Euler method with those angles.

This next example makes the redCube face straight up (no spin means facing forward, then 90 degrees up):

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

This next one keeps us facing right, while gradually rolling forwards (like we're on a barbecue spit):

```
public float zSpin=0;

void Update() {
  transform.rotation = Quaternion.Euler(0, 90, zSpin);
  zSpin+=1; // passed 360, which is fine
}
```

It's using the rule that z is always last, and is on our local axis. No matter how we're facing, the z-angle is always just a roll.

Here's a longer example using the AWSD keys to aim ourself in a small area:

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

void Update() {
  // AWSD keys aim us:
  if(Input.GetKey("a")) yy-=1;
  if(Input.GetKey("d")) yy+=1;
  if(Input.GetKey("s")) xx+=1; // positive is down
  if(Input.GetKey("w")) xx-=1;

  // limit them:
  yy = Mathf.Clamp(yy, -45, 45);
  xx = Mathf.Clamp(xx, -60, 0); // -60 is up

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

If you want to think of a rotation as x,y,z degrees, this is the way to do it. Keep your own variables, like the `xx` and `yy` here, hand move them, and use `Quaternion.Euler` to "apply" them.

It works well enough for a limited area (this only lets us aim mostly forward,) but breaks down if we can aim anywhere.

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 axis

Another way to make a rotation is to pick an arbitrary line for an axis and spin around it. You give the command 1 line, and 1 0-360 degrees.

This gradually spins us around a line running from south-west to north-east:

```
public float degrees=0;

void Update() {
  Vector3 northEastDiag = new Vector3(1,0,1); // a northeast pointing line:
  transform.rotation = Quaternion.AngleAxis(degrees, northEastDiag);
  degrees+=4;
}
```

If you watch a few full rotations you'll spot the diagonal north-east line it's spinning around.

The name might fool you a little. It says `AngleAxis`, but you don't need to give it an official axis. Give it any arrow. It counts as a direction (which means the length doesn't matter,) sticking out of your origin. We spin around it like is was an axis.

This is the same as the previous except the axis line points almost straight up. We get almost a boring y-rotation, but just slightly off:

```
public float degrees=0;

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

It's only a direction, so 10 up and 1 was just an easy way to say "mostly up."

This next example uses our local x-axis to roll something else. It places the redCube in front of us, and rolls it as if we were pushing it:

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

void Update() {
  redCube.position = transform.position + transform.forward*2;
  redCube.rotation = Quaternion.AngleAxis(degrees, transform.right);

  degrees +=4;
}
```

AngleAxis uses the left-hand rule, based on which way you point the arrow. That means if you flip the arrow to go the other way, it flips which way plus and minus go. If that example used `transform.left` it would roll the same way, but towards us instead of away.

This last one uses the arrow from us to something else, because we can. It spin us around the arrow to the red cube:

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

void Update() {
  Vector3 toRed = redCube.position - transform.position;
  transform.rotation = Quaternion.AngleAxis(degrees, roRed);

  degrees +=4;
}
```

The line to the red cube might be long or short, but we rotate around it the same no matter how long it is.

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);
```

The most common use is making look at some other object. We use the arrow pointing from us to them. This keeps us facing the red cube:

```
void Update() {
  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 computes the line, runs LookRotation and sets it as our new rotation, all in one line:

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

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

Remember `LookRotation` takes any `arrow` you supply. But `LookAt` takes a point, which to uses to create the arrow.

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

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

A really fun look-at trick is making a "flat" look-at. We often have the ground will hills. We want to y-spin to face something on the ground, but don't want tip backwards to look up at something on a hill.

a way to do that is to get an arrow to the target, 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 find the angle between a northwest arrow and a northeast arrow. We know in advance it's 90 degrees, but we're computing using between-two-arrows:

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

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

It's more interesting when we have arrows to objects. This figures out if we were looking at the red cube, what extra rotation would we need to be looking at the blue one:

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

The only use for these is when we learn how to add rotations, which we can't do yet. We'll be able to look at the first spot and use `FromToRotation` as an offset, adding it gradually to our first.

4.2.5 LookRotation's extra roll

If you remember, just pointing at something is only part of a rotation – we still get a local z-roll. `LookRotation` points us somewhere.

If you don't add anything else, it leaves 0 at zero – your head is facing up, or as much up as it can be through just spinning.

`LookRotation` has a special rule for the z-roll. You can optionally give it the direction you'd like to try your head to face, as much as possible. It probably won't be able to match it exactly. It just spins on z so your head is facing as much that way as it can.

This aims you, and tries to make your head facing right:

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

No matter what you put for the second input, you'll be aimed exactly where you wanted. It only influences the z-roll.

You almost always want to use `Vector3.up` – head facing up. If you don't write a second input, `LookRotation` uses that.

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."

You may have an irresistible urge to look at the actual numbers in a quaternion. Like Vector3's, they're a simple struct, with just 4 floats named `x`, `y`, `z` and `w`.

We can take a look at the values by setting a rotation with degrees, than looking at those row xyzw values:

```
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 a single 50 degree rotation breaks into two different numbers, both very small, and neither one on the z slot.

You can try it with a few more rotations and they make even less sense.

From our point of view, the actual values in a quaternion are random small numbers totally unrelated to the degrees we entered. But again, that's not a problem since the built-in functions do the work for us.

One funny thing – if they make no sense, why does Unity make them `public` variables? It's because there are a few people who understand quaternions. For example, you could use a copy specialized quaternion function the internet (but Unity has all of the common ones included.)

4.3.2 dot-`eulerAngles`

We can make a quaternion from degrees using `Quaternion.Euler`. We can go back to dgrees using the `eulerAngles` function. That's how the system fills in the Inspector values.

Examples:

```
void Start() {
  // this prints out starting x, y, z rotation:
  Debug.Log( transform.rotation.eulerAngles );

  // 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)
}
```

Notice how the second one changed -20 to 340. That's because it's a round trip. The angles were turned into quaternion `a1`, then translated back. Unity happens to prefer the range 0-360.

Especially because of the way it flips around the values to one's it likes, there's no real use for `eulerAngles` except testing.

4.3.3 Multiple ways to write an angle

Besides being able to add and subtract 360, there's one other way to write the same angle: as 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 spinning an extra 180 degrees on y. 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.

It seems like those up-and-over versions are bad, but sometimes you really rotate x more and more until you go up-and-over past 90.

What this means is every rotation has several ways to write it using x,y,z degrees.

4.3.4 How the Inspector handles rotation

The Inspector does one more odd thing when it shows you the x,y,z numbers. It saves the starting values that you entered, and adjusts the rotation numbers to stay close to them.

It doesn't change the rotation. It uses +/-360 and the up-and-over trick to pick the best numbers out of all the possible ways to write it.

This means your program won't see the same numbers as the Inspector. Suppose you set `transform.rotation = Quaternion.Euler(-20,0,0).` It you print it in the program, you'll see it as 340. What you see in the Inspector depends on the starting value.

If you had 0 in an Inspector slot, the system will try to adjust numbers into -180 to 180. If the "real" value was 190, it would show you -170.

But if you started with 300 degrees in that slot, it would keep the numbers between 120 and 480.

This is suppose to prevent the Inspector values from snapping as much. It's not a real problem since the Inspector isn't part of the final game anyway.

It's just one more reason to to try not to think in degrees, and use the built-in quaternion functions.

4.3.5 Gimbal lock, xyz problems

The biggest problems with using xyz-degree euler rotations is trying to get a smooth spin from one facing to another at a constant speed.

At the equator, they work great. Slowly changing y spins sideways, slowly

changing x spins up/down. But as you angle x upwards, the y-degrees-per-distance gets larger. That slows down the spin-speed.

The worst part is when you're facing anywhere near the north or south pole. If you're aimed mostly up, you can't tilt sideways unless you snap-turn in place on y. That's called Gimbal Lock. It's also really hard to use x and y degrees to smoothly rotate anywhere past the north or south pole.

Animation in 3D models often uses xyz degrees. They solve the gimbal lock problem by limiting where you can rotate, and angling the axis so you mostly rotate around the equator.

Quaternions let us make a smooth rotation from anywhere to anywhere else. They don't have those bad north and south poles.

4.4 Quaternion setting commands

The previous commands compute and return a quaternion. There are alternate commands which assign to an existing quaternion. They do exactly the same thing – they're just alternates.

For example, these are the same:

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

There's nothing special about the second way – I don't think I've ever used it. I just don't want you to be confused if you see someone else using these shortcuts. Here are the rest:

```
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 is kind of confusing – you'd think it would be `q.SetEuler(0,90,10);`.

There are three more super-duper shortcuts: `transform.forward = Vector3.right;` is a shortcut for `transform.rotation = LookRotation(Vector3.right)`. There's no way to set the "up" direction (it always tries to put you head-up.)

You can also align your right side and up with a vector: `transform.right=v1;` and `transform.up=v1;`.

You rarely need those oddballs. Often it's because your model wasn't made facing +z and it's better to use the parent trick to fix that. We can make these later by combining a LookAt and a 90 degree spin.

Chapter 5

Combining rotations

Instead of thinking of a rotation as an absolute facing, you can think of it more like an offset: a change from where you're looking now.

We can apply a rotation to any arrow, to spin it that much. Or we can apply a rotation to another rotation, to rotate the rotation. For example: face ourselves somewhere, then add an extra twist.

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 Rotate an arrow

For rotations, we repurposed `*` to mean "apply this rotation." If `v1` is an arrow and `q` is a rotation, `q*v1` is that arrow rotated by that much.

Rotating a forward arrow is the simplest way to use it. This creates a 20 degree rotation on y, then applies it to a forward arrow. It gives us a 20-degree arrow:

```
Quaternion spin = Quaternion.Euler(0,20,0); // 20 degree y-spin
Vector3 v = spin*Vector3.forward; // tilted arrow
```

Using `Vector3.forward` is special. +z is the all-zero rotation, so a forward arrow counts as a no-rotation arrow. `spin*Vector3.forward` turns any rotation into it's arrow.

That's a useful thing to know, but the real use of rotation-times-vector is being able to use it on any vector. For example:

```
Vector3 v1 = spin*Vector3.right;
Vector3 v2 = spin*transform.right;
```

`v1` will be your right arrow, spun an extra 20 degrees. It's merely an arrow pointing at 110 degrees, but we made it in this cool way. With `v2` we don't

know which way it's facing, but it's your local +x arrow spun an extra 20 degrees around y.

A fun use is spinning an object in a circle. We can take any arrow and hit it with a gradually increasing rotation. This takes a forward arrow and slowly spins it 0-180 degrees, 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); // <- the key line

  redCube.position = transform.position + arrow;

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

I had it snap back at 180 so you can clearly see the starting position. This will start north, trace out a half circle going clockwise, then snap back to north.

The extra `*2` is for contrast. Remember `Vector3.forward*2` is merely (0,0,2) – a length two forward arrow. The `spin*(Vector3.forward*2)` applies the spin to the one long arrow.

We could instead start with an arrow pointing some other direction. Replacing the line with this next one would start the half-circle on the right, tracing a clock-wise half-circle left and down:

```
  // half-circle from right, going down and left:
  Vector3 arrow = spin*(Vector3.right*2);
```

Or we could start with just any line. This starts with a mostly-left arrow, tracing out mostly the top half of a circle:

```
  // half-circle on top, cocked a little:
  Vector3 arrow = spin*(new Vector3(-4,0,1)) ;
```

This arrow is longer. The equation rotates the actual arrow, so this long arrow stays long as it rotates.

Some notes on the rules for using these:

- The rotation has to come first, then the vector second. You'd think `v*spin` would also work, but it's just an error. It has to be `spin*v`.

- The star(`*`) isn't really a multiply. It's being re-used here as the "apply a rotation" symbol. It's a little like how `"cow"+"bell"` knows the `+` isn't a regular math plus.

  We're allowed to repurpose math symbols – it's called Operator Overloading.

- The regular math rules apply, so `v1+spin*v2` spins `v2` first, then adds `v1`. Using `spin*(v1+v2)` adds the arrows first, then spins the result.

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

```
// bottom 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. We start forward, and spin down. If we used `Vector3.up` we'd get the north half of a circle.

Spinning a point on the axis does nothing. Either of these two lines won't make it move at all:

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

// spinning right around x does nothing:
Vector3 arrow = Quaternion.Euler(degrees, 0, 0) * Vector3.right;
```

In the first one, the up-arrow really does spin around y, but it just twirls in place. Points don't have a facing, so it stays as (0,1,0) no matter how much we spin. Spinning a right or left arrow around x is the same thing – no change.

The other oddball is spinning points that aren't "on" the circle. We can take a point 5 up and 1 over, and spin it around y. It will make a tight radius 1 circle, 5 units up. In other words, the 5 up won't change. Only the 1 sideways part will spin:

```
// makes a radius-1 half-circle, 5 units above us:
Vector3 arrow = Quaternion.Euler(0, degrees, 0) * (new Vector3(1,5,0));
```

Sometimes it helps to stop imagining it as an arrow, and instead as a point, glued into a 3D grid. Then imagine the whole thing spinning. You'd see the dot at (1,5,0) up in the air, making a little ring around the y-axis.

So far we've been making rotations with the `Quaternion.Euler` method, but any way to make a rotation will work. Suppose we're angled funny and want to orbit something from our right to left, which is around our z-axis.

Using `transform.right` for the starting arrow seems right. To make the spin be around our local z, we can use angleAxis:

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

To sum up:

- Spinning a point which is on the axis won't change it. For example, spinning a forward arrow around +z.
- 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.
- Any kind of rotation can be used: one's made with `Quaternion.Euler`, or `AngleAxis` or our `transform.rotation` .... No matter how you get one, all rotations are quaternions, so can be used for this trick.

Fun fact: the computer calculates `transform.forward` using `transform.rotation*Vector3.forward`. It bends the forward arrow by your rotation.

`tranform.right` is the same idea. It's `transform.rotation*Vector3.right`.

Here's one longer example. I'd like to shoot a ball randomly forward, 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;
// random 0-360 z-spin:
dirArrow = Quaternion.Euler(0, 0, Random.Range(0,360)) * dirArrow;
```

After those two spins, dirArrow is somewhere in a 10-degree forward cone. Shooting a ball that way is a simple use of the `dirArrow` arrow.

Sometimes we want the first few shots to miss. It's easy to put a hole in the middle by changing the first angle to be 5-10. Our almost-forward arrow will be somewhere in a donut.

5.2 Combining rotations

We also use the star symbol to combine rotations. First a simple example. We know (-45,90,0) is really done in two steps, y then x. We can make each of those separately and combine them:

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

Quaternion y90thenx45 = ySpin90*xSpin45; // <-- the new line

transform.rotation = y90thenx45;
```

The rule is that the second rotation uses the new local axes. In `ySpin90*xSpin45`, the second part tilts us up on *local* x.

If you flipped them, `xSpin45*ySpin90`, you'd get a different direction. We'd tilt up on the real x, then we'd spin 90 degrees on the local y, putting us back to the ground facing exactly right, with a 45 degree roll.

It seems seriously weird that the order matters, but that's really the way it is. It's not even a quaternion thing. The way rotations are, the order matters.

Here's a real example, I want to look at something, then roll on my side. Getting a look rotation puts our head up. So we add a 90 degree roll on local z:

```
// look rotation to red cube, head up:
Quaternion qLook = Quaternion.LookRotation(redCube.position-transform.position);
// standard 90 degree z-roll:
Quaternion qzRoll90 = Quaternion.Euler(0,0,90);
// look, then local roll:
transform.rotation = qLook*qzRoll90;
```

We can see `qzRoll90` is just a simple z-roll. Like any offset, what it does depends on how we use it. Since it's second, it automatically applies to the local z.

Just for fun, this looks at something while gradually rolling. It's using the same local z-roll trick, but now with an increasing 0-360:

```
float zRoll=0;

void Update() {
  Quaternion qLook=Quaternion.LookRotation(redCube.position-transform.position);
  Quaternion qRoll = Quaternion.Euler(0,0,zRoll);
  transform.rotation = qLook*zRoll;

  zRoll+=4;
}
```

Here's a trickier one. We want to make an orbit, right-to-left (the z-axis circle we made before) and add a forward/back zig-zag. It should trace out a wavy circle.

The first part will be the same: spin a `Vector3.right` around z. To add the waves, think of the right-facing arrow. Wiggling y will make it wave. As we rotate in the circle, the local y axis rotates with us, staying 90 degrees ahead. Wiggling on local y will always be a perfect forwards/backwards:

```
public Transform ball;

float zSpin=0; // 0 to 360
// The wobble:
float ySpin=0; // goes between -20 and 20
int yDir=+1; // +1 or -1, to make it go back&forth

void Update() {
  // main circle:
  Quaternion qCircle = Quaternion.Euler(0,0,zSpin);

  // back-and-forth y-wobble:
  Quaternion qWobble = Quaternion.Euler(0,ySpin,0);

  // now combine them: orbit spin, local y-wobble
  // ...and apply that to a long Right arrow:
  Vector3 toBall = qCircle*qWobble*(Vector3.right*2); // <- key line
  ball.position = transform.position + toBall;

  // move them:
  zSpin+=2;
  ySpin+=3*yDir;
  if(ySpin>20) { ySpin=20; yDir=-1; }
  else if(ySpin<-20) { ySpin=-20; yDir=+1; }
}
```

Obviously, this is something where you want to add comments explaining the plan. The key line `qCircle*qWobble*Vector3.right` isn't easy to just look at and know what it does.

5.3 Visualizing rotation combinations

I wrote that in `q1*q2` the second rotation uses the local axis of the first. Well ... that's one way to look at it. There are others.

The most important thing is the order matters. If you combine rotations A, B and C, you have to choose between `A*B*C` or `B*A*C` ... 6 combinations. All do different things.

A lot of times, we're thinking of one thing as the base rotation, and the rest as ways to adjust it. Here are the ways to think of it:

- `base*local`. Multiplying after the starting spin applies the new one as local.
- `world*base`. Multiplying before the starting spin applies the new one using world axes.

- Left-to-right `base*local1*local2*local3`. Each one after the first uses the local axis so far.
- Right-to-left `world3*world2*world1*base`. You can think of the right one as the base spin, and then go right-to-left and apply each one using world coordinates.

That seems really confusing, that `A*B*C` could mean left-to-right local, or right-to-left world. For real, you start with a plan. If your plan uses local rotations, add them to the back. If it's global rotations, add them to the front.

In case you were wondering, this is the way real rotation math works. Unity is just copying it, and not making it any more complicated than it was already.

Examples:

This is the cone with a hole in it example, rewritten to think in local axis. Assume we want to shoot the ball in almost our local forward. First we take real forwards and spin to face my forward. The next step is rolling 0-360 on local z. We'll still be facing forward, but the local y-axis will be in a random direction. Then we can cock 10-20 degrees on local y.

In other words, point your finger forward, spin randomly in place, and cock it a little sideways. Here's the code:

```
// 360 roll on z, for the "miss direction":
float missDir = Random.Range(0,360);
Quaternion qCone = Quaternion.Euler(0,0,missDir);

// sideways cock 10-20 degrees:
float missAngle = Random.Range(10.0f, 20.0f);
Quaternion qCock = Quaternion.Euler(0, missAngle, 0);

// combine them into the final direction:
Quaternion qShoot = transform.rotation*qSpin*qCock;

ball.position = transform.position+qShoot*(Vector3.forward*2);
```

That second-to-last line is thinking in local, right to left: face my way, random roll on local z, cock using randomly-facing local y. In case you were wondering.

Because of the rules, we can also think of this right-to-left as world: small y-rotate, random 360 spin on world z, then apply the world rotation to get my facing. But I think this is easier to imagine as left-to-right local. Applying a series of local rotations sounds more complicated, but it's more natural.

Here's an easier one using the world trick. We want to start just any facing, keep it, and have the script spin me around global y. We can do that by putting the y-spin first:

```
Quaternion startSpin; // saved value
float yDegs=0;

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

void Update() {
  yDegs+=3;

  Quaternion ySpin = Quaternion.Euler(0,yDegs,0);
  transform.rotation = ySpin*startSpin; // <- key line, ySpin is first = global
}
```

Read the last line as: `startSpin` is the base, with world `ySpin` done to it.

Let me just say again this is some of the hardest stuff to use, but no one is going to know you took a few tries to get something working. The keys are: remember `q1*q2` is different from `q2*q1` for rotations. For each rotation decide if you want it on global, or local so far, and put it in front or in back, depending. Remember you can mix&match both types. And write down your plan in a comment, since the line combining them will make no sense without it.

Chapter 6

Moving

Unity has some built-in commands to help gradually move, or spin. Some are things we couldn't do before, but most are just shortcuts.

6.1 Moving a point

We can already use offsets and normalization to move at a certain speed. Unity has some built-ins to help with this.

First a review. This uses our old unit arrows and distance to move us at a constant rate towards some other object:

```
public Transform goHere;
public float moveSpeed=2.0f;

void Update() {
  Vector3 toTarget = goHere.position - transform.position;
  Vector3 unitToTarg = toTarget.normalized;
  float targDist = toTarget.magnitude;
  float mvAmt = moveSpeed*Time.deltaTime; // units to move this frame

  Vector3 newPos;
  // don't overshoot:
  if(mvAmt>=targDist) newPos=goHere.position;
  else {
    newPos=transform.position + unitToTarg*mvAmt;
  }
  transform.position = newPos;
}
```

The line in the `else` is where most of the work is. We have a unit (length 1) vector to our target. We scale that arrow by how much we want to move (`mvAmt`) and add it.

Before I moved some other cube. This code is moving ourself. It works the same but notice the new way it avoids an overshoot. If we're closer than 1 move, we just place ourself on the target.

6.1.1 MoveTowards

`MoveTowards` is a shortcut for most of that. It takes where you are, where you want to go, and how much to move. Here's a rewrite of the code above using MoveTowards:

```
public Transform goHere;
public float moveSpeed=2.0f;

void Update() {
  float mvAmt = moveSpeed*Time.deltaTime; // units to move this frame

  Vector3 newPos;
  newPos = Vector3.MoveTowards(transform.position, goHere.position, mvAmt);

  transform.position = newPos;
}
```

Obviously, the inside of `MoveTowards` is all the math in the first version, including "don't overshoot."

Because of the no-overshoot part, it's safe to run `MoveTowards` every frame, no matter what. Once you reach the target, `MoveTowards` does nothing – not even any jiggle. Another part of the program can move your target, doing no other work, and you'll start moving that way.

The command looks so neat that it's easy to forget it's still just a 1-time small move. Like anything else, it has to be in Update if you want it to keep moving you each frame.

The third input is the amount to move. The easiest thing to do is to keep it constant, which can be robotic-looking. You can easily change the speed as it moves to get more realism, or to make it prettier.

This example uses a simple equation to make it go faster when you're further away. The numbers aren't important, just that they're based on distance. We'd add these new lines at the start of Update:

```
moveSpeed = ((goHere.position - transform.position).magnitude)/3;
moveSpeed = Mathf.Clamp(moveSpeed, 1, 4);
```

Previously, it moved at a constant 2/second. Now it moves twice as fast when it's far away (the equation has 12+ units away be max speed) and slows, down to 1, as it gets closer. Sort of like a rubber-band effect.

We could do the opposite and have it start slow, quickly increasing to a speed of 2:

```
moveSpeed=0.3f;

// each frame, try to get up to full speed:
if(moveSpeed<2.0f) moveSpeed+=1.0f*Time.deltaTime;
```

When we reach the target, or change to a new one, we'd could reset the speed to 0.3, causing us to accelerate when we start moving again.

6.1.2 Lerp

Another common way to move is what we've been doing with an all-the-way arrow and a 0-1 percent. The technical term for taking a 0-1 percent between two points is Linear Interpolation, shortened to "lerp."

Unlike MoveTowards, we have to know the start point. When we moved a green cube towards a red cube the start point was always us. If we move ourself, we have to save the start point:

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

This code uses math we already know to move us to the target:

```
public Transform goHere;
public float secondsPerTrip=3.0f;

Vector3 savedStart; // where I started my move
float tripPercent=99; // 0 to 1. 99=not moving

void beginMove() {
  savedStart = transform.position;
  tripPercent=0.0f
}

void performMove() {
  // move pct from 0 to 1:
  tripPercent += Time.deltaTime/secondsPerTrip;
  if(tripPercent>=1) { tripPercent=1; }

  Vector3 wholeMove = goHere.position - savedStart;
```

```
  transform.position = savedStart + wholeMove*tripPercent; // <- key line
}

void Update() {
  if(tripPercent<1) performMove();
  else if(Input.GetKeyDown(" ")) beginMove();
}
```

The inside of `performMove` should look very familiar. The rest is to set it up. Unlike `MoveTowards`, the target can't move (if it does, we'll snap to the same spot on the new between-arrow.) We need to reset the start and percent each time.

The built-in Lerp function replaces three lines in performMove:

```
  tripPercent += Time.deltaTime/secondsPerTrip;
  transform.position = Vector3.Lerp(savedStart, goHere.position, tripPercent);
```

It computes the line, makes sure percent can't go past 1, and uses the percent to find the point.

The big difference is moving with a percent takes the same speed to go there. If the points are nearby, it crawls, taking 3 seconds. If they're far apart, it zooms to take 3 seconds.

We could change that by moving the percent slower for further apart objects, but that's a pain. MoveTowards is usually easier.

There's another, completely different way people use Lerp. Each frame you move a fixed percent of the total distance closer. It gives a fast-then-slow movement. It's totally unrealistic, but looks cool and is easy to do.

Here's how the trick looks with regular math:

```
public Transform goHere;

void Update() {
  Vector3 toTarget = goHere.position - transform.position;
  transform.position = transform.position + toTarget*0.05f;
}
```

Each frame it recomputes the new arrow to the target, and moves us 5% of the way along it. We never actually reach it, but we quickly get so close it won't matter.

Here it is written using a `Lerp`:

```
public Transform goHere;

void Update() {
  transform.position = Vector3.Lerp(transform.position, goHere.position, 0.05f);
}
```

When you see the first input matching where you assign it, it's this fast-then-slow method.

6.2 Gradually rotating

Our previous method of making a moving rotation was hand-moving the degrees and remaking the quaternion. A better way is directly rotating a quaternion, using special functions.

This lets us get smooth motion, from any angle to any other, even using two rotations that were made different ways. It's the main reason we like quaternions so much.

6.2.1 Spinning an arrow

Unity has a fun shortcut for spinning an arrow. It works like `MoveToward`. You give it the current arrow, the new arrow, and how much you want to spin this frame.

This example puts a ball to our right, and spins it to point straight up:

```
public Transform ball;
Vector3 toBall = Vector3.right*4; // this is what we will be spinning

void Update () {
  toBall=Vector3.RotateTowards(toBall, Vector3.up*4, 1.5f*Time.deltaTime, 0);
  ball.position = transform.position+toBall

  // reset, for testing:
  if(Input.GetKeyDown(KeyCode.A)) toBall=Vector3.right*4;
}
```

You should be able to see it's not just moving from right to up – it's really spinning the arrow, making the ball move in a curve (if we wanted straight movement we'd use MoveTowards.)

The third input is the spin amount. It's in *radians*, not degrees. 90 degrees is 1.57 radians. That's why it has a speed of 1.5 but goes so fast – that means to rotate almost 90 degrees per second. I'm not sure why it uses radians. 'Real" math uses radians for all angles, but Unity uses degrees to everything else.

`RotateTowards` can also change the length of the arrow. In this case, the starting and ending arrows are both length 4, but they didn't have to be. The 4th input is how fast it grows/shrink the arrow. As soon as it gets to the correct length, it stops. If it doesn't change size, the 4th input won't matter. I think 0 looks good for those cases.

If your math is wrong, the rotation and size change will finish at different times. It could rotate all the way then finish shrinking, or shrink to the final size halfway through the rotation. For example:

```
toBall = Vector3.right*2;
  ...
toBall=Vector3.RotateTowards(toBall, Vector3.up*4, 1.5f*Time.deltaTime, Time.deltaTime);
```

For about a second this spins to face up while shrinking to length 3. Then, while pointing up, takes another second to shrink down to length 2. Doubling the 4th input would fix that.

6.2.2 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(0,90,0);

void Update() {
  Quaternion qq = Quaternion.RotateTowards(transform.rotation, qWant, 60*Time.deltaTime);
  transform.rotation = qq;
}
```

This spins us from our current facing to due east, spinning at 60 degrees/second.

This (and RotateTowards) takes the most direct line, and has no problem going diagonal. You could be facing any direction and this will rotate you by the most direct route.

Since these are rotations, it also takes into account the z-roll. This will just spin us in place from head up to head down:

```
// start exactly forward:
void Start() { transform.rotation = Quaternion.identity;}

void Update() {
  Vector3 qWant=Quaternion.Euler(0,0,-180);
  transform.rotation=Quaternion.RotateTowards(transform.rotation, qWant, 30*Time.deltaTime);
}
```

We can also do both at once. This flips us on our back while slowly tilting us a little bit up:

```
// start exactly forward:
void Start() { transform.rotation = Quaternion.identity;}
```

```
void Update() {
  Vector3 qWant=Quaternion.Euler(-10,0,-180);
  transform.rotation=Quaternion.RotateTowards(transform.rotation, qWant, 30*Time.deltaTime);
}
```

These always both finish at once, since they count as one big rotation. This one quickly rolls 180 on z while slowly tilting the 10 on x.

6.2.3 Rotation lerp

Rotations have there 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. This is because `q*0.5` or `(q1+q2)/2` don't work at all.

To get a rotation 1/2-way between `q1` and `q1`, 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);
```

Both of the old moving-Lerp tricks work.
`Quaternion.Lerp(startRotation, endRotation, pct0to1);` will spin you from one to the other as the percent goes to 1. RotateTowards is probably simpler.

The percent-based Lerp will give a fast-then-slow spin. This zooms us to facing up, easing into it as we get near:

```
void Update() {
    transform.rotation=Quaternion.Lerp(transform.position, Vector3.up, 0.03f);
  }
}
```

There's an alternate version named `Quaternion.Slerp`. It does the same thing as Lerp – you can use either one. Slerp uses a better but slightly slower math equation, but I've never seen a difference.

Like vector Lerp, Quaternion Lerp only uses 0-1 percents. If you really need 150% of your angle, `LerpUnclamped` lets you use any percent. Exs:

```
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
```

This can solve some very specific math problems, like needing to double an angle (use 2 for the percent,) or find the opposite angle (use -1.)

6.3 Time.deltaTime

This is a common trick that lets us use nice speeds for movement. The problem is we prefer to think of movement as speed-per-second, but these commands don't work that way. `Time.deltaTime` is a way to fix that.

I'll show you how to use it first, then the explanation:

Whenever you run a move command every frame, the easiest thing is to use amount-per-second for the speed. Use the per-second movement, multiplied by `Time.deltaTime`. This moves at 5 units/second:

```
Vector v1 = Vector3.MoveTowards(v1, target, 5*Time.deltaTime);
```

If we run this every Update, it moves a little each time, adding up to 5 each second.

The actual value of `Time.deltaTime` is how many seconds it's been since the last time Update ran, which is always something tiny like 0.0166 seconds. You can remember it since *delta*, in math, means how much something changed. It's the change in time.

To see how it works, suppose you have exactly 60 evenly-spaced Updates every second. `Time.deltaTime` will always be 1/60th – 0.0166. Each Update you'll be adding 1/60th of 5, which is what we want to add to 5/second.

Next suppose one frame gets skipped. The next Update will have double the `deltaTime`. We'll move twice as much that update, making up for the time we missed.

For rotations, this spins at 60 degrees per second because of the 3rd input:

```
Quaternion qq = Quaternion.RotateTowards(qq, qWant, 60*Time.deltaTime);
```

If we wanted to make it spin faster, increasing at a rate of 40 degree per second, we'd use `Time.deltaTime` for the speed increase:

```
// this adds 40 every second:
rSpeed += 40*Time.deltaTime;

Quaternion qq=Quaternion.RotateTowards(qq,qWant, rSpeed*Time.deltaTime);
```

When you see things using `Time.deltaTime`, it's easiest to not think of it as a variable. Look at what's next to it, and that's speed/second.

One last note, it's still just a 1-time thing. The trick works because we do it every Update. For example, this would move us a tenth of a unit each time we pressed "a," and no more:

```
if(Input.GetKeyDown("a"))
  transform.position = Vector3.MoveTowards(transform.position, dest, 6*Time.deltaTime);
```

But if we changed it to `GetKey`, which is true as long as the key is held, it would run every Update and would give a speed of 6/second.

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.

A funny thing is 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");
```

Both inputs need to be arrows, and they usually only make sense if they really are coming from the same spot. Suppose we use the positions by mistake: this computes a wrong angle between us and the ball:

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

`transform.position` isn't an arrow. But if you say it is one, it counts as the arrow from (0,0,0) to you. So this 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`. Any two can create the third. That's one of the main purposes of cross-product.

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, 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 know three of the 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 – that's 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 cross product and angle between you and the normal, then spinning that much past it.

7.0.5 Opposite of an angle

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

It cancels out the original rotation: `q*qInverse` is rotation (0,0,0).

To be cute, you can also use a no-limit Lerp to get it. Start at your angle and go 100% past 0: `qInv = Quaternion.LerpUnclamped(q, Quaternion.indentity, 2.0);`. But using Inverse is probably easier to read.

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

If you're using trig to do things, it tends to take more testing to get it to work, and there's almost always an easier way. Almost. But just in case, here's some of the trig you should be avoiding.

7.1.1 Radians

Real trig functions, like sin, cosine ... use radians instead of degrees. They are 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, and so on.

Unity has a builtin `Mathf.Rad2Deg` conversion, but it's just 57. It won't do anything about starting in the wrong place or going backwards. `Mathf.Deg2Rad` is just 1/57.

If you really need to convert a trig angle in radians to a unity angle in degrees, it's:

```
degs=-rads*Mathf.Rad2Deg+90;

rads=-degs*Mathf.Deg2Rad+Mathf.PI/2;
```

There's also a shorter but more confusing method involving switching x and y.

7.1.2 atan2

A standard trig trick is turning a line's slope into its angle. If a line has a slope of 1, it's at 45 degrees. Arc-tangent does that, but it crashes on 90 degrees (the slope is infinity) and only works for half a circle. The standard computer trig trick is a rewrite using x and y called `atan2`. It gives a correct full-circle angle.

In Unity `Mathf.Atan2(transform.forward.z, transform.forward.x)` gives your "flat" y-spin. But in radians, not degrees. Using Angle is almost always easier.

7.1.3 dot product

Dot product is the trig version of angle. It tells you the angle between two arrows except: they have to be length 1 (you can normalize them,) and it goes from 1 to -1, which is odd. An exact match gives 1, 90 degrees apart gives 0, and exactly opposite gives -1.

It's actually the cosine of the angle between them, so 45 degrees give 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

Sometimes we have a problem where we think "if I was always sitting at 000 facing forward, this would be easy." That's the local space trick.

We've already seen the easy version: solve a problem as if you were facing forward, then twist it to the real angle. Shooting a ball in our forward cone was like that.

The advanced version of the trick is the really useful one. You start with a problem involving you and some other objects. First you bring them into your local space, then solve it using easy math, then bring the answers back into real world space.

8.1 Local Space

Local space is the official name for coordinates using your xyz axis, with you as 000.

So far, we've been using local space informally, for example placing a ball ahead of us using `transform.forward*2`. In our minds, the ball is at (0,0,2) in our local space. Or when you child an object, the coordinates you see are in the parent's local space.

The main trick to using local space is we know the real xyz axes are straight, and our local axes are diagonal lines, but forget about that. Instead, imagine we select and center ourself, and spin the camera to line up our local xyz's. Now, to us, our local coordinates are a perfect, real xyz.

When you're in local space, it's a world where you are always at (0,0,0), pointing exactly forwards. This makes a lot of math very easy.

The reason we don't have to think about the real xyz is we'll use one equation, at the end, to convert back.

Converting local space to global should look familiar. Multiply the local coords by our rotation and add our position:

```
Vector3 localOffset = new Vector3(1,0,2); // sample local-space coords
Vector3 pos = transform.position + transform.rotation*localOffset;
```

Converting a world point into our local space is new. But it's just the opposite: subtract our position and apply the opposite of our rotation:

```
// convert ball.position into our local space:
Vector3 toBall=ball.position-transform.position;
Quaternion qMyInv = Quaternion.Inverse(transform.rotation); // my opposite rotation
Vector3 ballLocal = qMyInv*toBall;
```

A fun question is asking whether (1,0,2) is world space or local space. Like everything else, it depends what you were thinking when you made it. Whenever you have a point, you got it a certain way, or made it for a certain reason. That tells you whether it counts as global or local.

8.2 Local space tricks

A common, obvious use of local space is asking whether a ball is to our left or right. We just have to convert the ball's position into our local space:

```
// standard global to local formula:
qToMyLocal = Quaternion.Inverse(transform.rotation);
Vector3 localBall = qToMyLocal*(ball.position-transform.position);

if(localBall.x>0) print("ball is to my right");
if(localBall.z<0) print("ball is behind me");
```

Notice how "bringing the ball into local space" just means we use temp `localBall` to hold the local space numbers.

Here's a quick check: suppose `localBall` is (1.3, 0, 6.2). That means we can start from us, go 1.3 on our right arrow, 6.2 on our forward arrow, and we're at the ball.

That example only had to bring things to local space – it didn't need to convert anything back to global. Here's an example of the full trick – bring it to local, do the math, bring it back out:

We'd like to suck the ball to our forward/backward axis – if it's 3 ahead of us and 1 up, we want to suck it so it's just 3 ahead. The math for that using real xyz is hard. Here it is using local space:

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

```
// Now do local space math. We want to kill the ball's x and y:
Vector3 suckTo = new Vector3(0,0,localBall.z);
localBall = Vector3.MoveTowards(localBall, suckTo, 3*Time.deltaTime);

// now convert back to global and put the ball there:
Vector3 ballPos2=transform.position+transform.rotation*localBall;
ball.position = ballPos2;
```

It looks a little long. But the first is just converting to local space, and the last is converting to global. Once you use this trick a few times, that stuff just goes there and your eyes just brush past it. The middle part is the real work, which is easy now that we're in local space.

This problem is a little made-up. If we wanted to suck the ball directly towards us, we could use normalized `toBall` from a few chapters ago. We wouldn't need local space. But you get oddball stuff like this a lot. Whenever the explanation of a problem involves a local axis, the local space trick might help.

Another use for thinking in local space is not having to use `transform.forward` anymore. When we use that, we're thinking local +1 z, and then immediately translating it into global. If you like local space, we can build a position in local, then turn it into global.

A really simple example is putting something in a random box in front of us:

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

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

We could have built the same thing using `transform.forward` and `transform.right`, but this is a little shorter. It's also nicer if you have complicated placement math and have to adjust x, y and z a lot.

If seems like this would be extra-hard if we want the ball to avoid a dog or something – we'd be test-converting into global a bunch to compare. The usual trick is to bring the dog into local space with us.

Here's one more example of the full (to local, math, back to global) trick:

Suppose we want to keep a rigidbody ball locked onto a diagonal line. The line is marked out by an empty gameObject as one end, with its +z along our line. We can use the local trick: bring to ball to the line's local, fix it (make sure x and y are both 0) and then convert back to global and put the ball there:

```
public Transform theLine; // the start of a line, going in +z
Quaternion qToLineLocal; // saved copy for global to local conversion
public Transform ball;

void Start() {
  // to bring the ball into line's local space
  // computing once, since the line won't move:
  qToLineLocal = Quaternion.Inverse(theLine.rotation);
}

void FixedUpdate() {
  // standard world to local formula:
  Vector3 localBallPos = qToLineLocal*(ball.position-theLine.position);

  // adjust ball to be exactly on the line:
  localBallPos.x=0; localBallPos.y=0;
  if(localBall.z<0) localBall.z=0;

  // standard local to global formula:
  Vector3 ballPos = theLine.position+theLine.rotation*localBallPos;
  ball.GetComponent<Rigidbody>().MovePosition(ballPos);
}
```

There's probably more tweaking to make this particular example work (my ball sometimes bounced extra-hard off the floor.)

8.2.1 Converting rotations, offsets

That math was for points. Sometimes you want to use local space with rotations. It works mostly the same way. Multiply by your rotation to convert local to world; by the inverse to convert world to local:

```
Quaternion qLocalBall;
qLocalBall=Quaternion.Inverse(transform.rotation)*ball.rotation;
// do your math ...

// change back to world:
Quaternion qWorldBall = transform.rotation*qLocalBall;
```

A local rotation of (0,0,0) is pointing along your +z. It means the ball is pointing the same way you are. Local rotation (0,10,0) means to you, the ball seems to be looking a little to the right.

The usual tricks apply, for example you might create a local rotation, then convert to global. Or bring a rotation into local, just to check it.

Converting arrows and offsets to and from local is also done by multiplying by your rotation or the inverse – don't add or subtract your position.

But it works out the same. You probably got the offset by subtracting your position. And you probably intend to add the offset to your position at the end. So this really means to make sure you don't double-add or subtract your position.

8.3 Built-in shortcuts

Unity has a 6 shortcuts for converting to/from local space of a Transform. If you're comfortable with rotation math, learning these is probably more trouble than it's worth. But it's nice to see what they want to provide.

The names are a little funny. I think – I'm not sure – these were the standard names of the commands when Unity borrowed them.

Three of them are different ways to go from local space to world space.

`transform.TransformDirection(v);` is just `transform.rotation*v`. It converts a local arrow to a global arrow.

Notice how it uses the technical term *direction*. That tells you the output is an arrow on global space, and not a point. To use it, you need to add it to something, often yourself.

`transform.TransformVector(v);` is the same, but also multiples by your scale. Which seems like an odd thing to multiply by.

The idea is, suppose you use this to place something touching your surface. If you double your scale, you have to double the offset to still be touching that spot. If that's what you plan to do, this command is perfect.

The math is: `transform.rotation*(transform.lossyScale*v);`. LossyScale is your actual scale, counting your scale and all of your parents'.

The last one is the same except it adds your position at the end. It converts a local into a global point. The command is `transform.TransformPoint(v);`. The math is: `transform.position+transform.rotation*(transform.lossyScale*v);`

How can you tell from the name that that imultiplies by the scale? I don't know – I guess the people who use this just know. Plus most things have scale (1,1,1), so it won't matter anyway.

Notice how all of these take a local position/offset as input. They're the same. A local position is in your local space, which uses you as (0,0,0). An offset from you in your local coordinates is just the long way of writing that.

The other three go from global to local. They convert the input to a point in your local-space. Remember the main use of that is making math easier, then converting back to world space to use it.

`transform.InverseTransformDirection` simply converts a global arrow into your local space. As a review, this uses it to check whether the ball is

to your left or right:

```
Vector3 toBall = ball.position - transform.position; // world
toBall = transform.InverseTransformDirection(toBall); // now local
if(toBall.x>0) Debug.Log("ball is to my right");
```

I suppose you can remember it since the math way to do it uses Inverse:
`toBall = Quaternion.Inverse(transform.rotation)*toBall;`

`InverseTransformVector` is the same, but divides by your scale. It's the opposite of `TransformVector`.

Suppose a basic 1x1 barbecue grill has certain hotspots, and grills scale to any size. This command makes it easier to bring a world offset from a scaled grill into the original 1x1 size, where you can do easy math based on hotspot positions.

Sure, you could just divide by the scale yourself. But if you already know this command the built-in divide is probably pretty nice.

`InverseTransformPoint` assumes the input is a point – not just a global offset. It subtracts you first, then runs inverse transform vector (bring to local, divide by scale.)

And again, these are all just short equations, which you could write yourself. Or maybe seeing the equations helps you use and understand these.

Chapter 9

Misc examples

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

9.1 Getting your y-spin

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

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

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

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

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

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

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

9.2 Using a y-spin

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

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

```
float curYspin;

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

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

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

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

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

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

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

9.3 Align with ground

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

9.4 Orbit camera

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

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

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

Simple code to do that:

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

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

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

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

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

This would go before we use `qCam`:

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

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

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

make the distance to the camera longer or shorter.

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

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

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

9.5 Drawing a line between

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

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

The code:

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

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

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

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

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

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

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

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

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

9.6 Consistant LookAt

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

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

```
float zz=4;

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

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

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

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

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

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

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

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

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

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

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

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

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

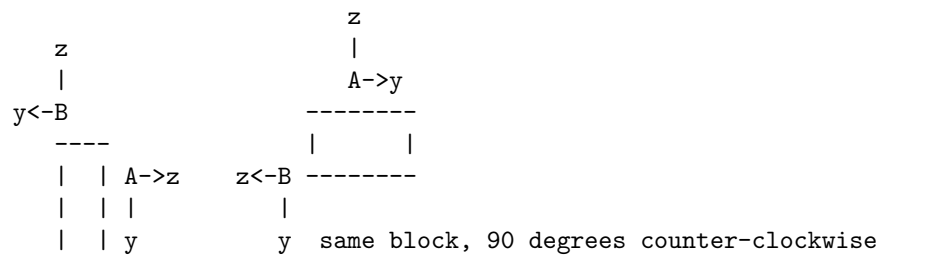
This RoateTowards trick will also help a moving object gradually tilt to the ground angle. Compute your ground-tilt angle as normal, then use the second-to-last line above to ease into it.

9.7 Connect two blocks

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

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

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



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

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

```
public Transform cube1, cube2;

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

}
```

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

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

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

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

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

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

```
public Transform cube1, cube2;

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

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

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

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

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

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

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

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

```
public Transform cube1, cube2;

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

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

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

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