Chapter 7

Misc Math

This section is about miscellaneous things we can do with rotations and arrows, and other various functions you don't see very often.

7.0.1 Finding angles

`Vector3.Angle(v1,v2)` tells you the angle in degrees between two arrows. It treats them as if they were both coming from the same spot, finding the angle of the V they make. It doesn't care about the direction. If one arrow is 30 degrees from another, it could be right, up, diagonal – anywhere on a 30-degree cone.

It's always positive. The largest value is 180, for 2 arrows in opposite directions.

Angle and RotateTowards and AngleAxis are related in a fun way. It turns out that for any 2 arrows coming out of the same spot, there's always one perfect axis at 90 degrees to them both. You could use it with AngleAxis to turn one arrow into the other. The amount you would need to spin is what Vector3.Angle computes.

That's actually how RotateTowards works. It looks at the two arrows, computes the 90-axis they share, and finds the Angle. It then uses AngleAxis. RotateTowards looks so smooth because it really is a simple spin around 1 diagonal axis.

A common use of Angle is checking whether something is in our "vision cone." Find the angle between your forward arrow and an arrow to the target. If the angle is too big, we can't see it:

```
Vector3 toBall = ball.position-transform.position;
float angToBall=Vector3.Angle(transform.forward, toBall);

if(angToBall<30) print("I can see it");
```

It's a little fake. For real a creature may not see in a perfect circle – they might see further sideways than up. But this is often close enough.

My favorite mistake is using the positions instead of the arrows:

```
// this code makes no sense:
float angToBall = Vector3.Angle(transform.position, ball.position);
if(angToBall<30) print("a person at 000 can see both at once");
```

What happens is fun (but not actually important). It treats the positions as arrows, each coming from 000. If finds the angle between them as if you were standing at 000.

We can get the "flat" compass angle by using the old trick of zero-ing out y. This is the trick to not count standing on hills or valleys:

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

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

Naturally, this also won't tell you the direction. 90 degrees could be 90 degrees left or right.

`Quaternion.Angle` is the same as `Vector3.Angle` except it also counts the z-roll. If two cows are aimed the same way, but one is on its side, Quaternion.Angle says 90. If the direction is 30 degrees apart and so is the z-roll, the angle is 42.2. If you're running Quaternion.RotateTowards, this is how long it will take. But you usually want Vector3.Angle.

Suppose you had a game where someone has to line up two logs. If you only need them facing the same way, check if `Vector3.Angle(log1.forward, log2.forward)` is small. If you also want to check if they're spun the same way (both mossy side up?) use the quaternion version to compare the exact rotations: `Quaternion.Angle(log1.rotation, log2.rotation)`.

7.0.2 Cross Product

Cross Product finds the axis between two arrows. This is the what Angle and RotateTowards use. You rarely need it – other functions will compute it if they need it.

Another way to think of cross product is as if 2 arrows are spokes on a wheel. Cross product finds the axle. The internet also has lots of pictures of how a cross product looks.

In Unity, `Vector3 cp = Vector3.Cross(v1, v2);` gives it. It's a Vector3, which counts as a direction.

One semi-common use is finding the +/- direction of a rotation. It turns out that an axis can go either way, depending on the direction. The cross product of forward and right is y, but it's either up or down depending on the order:

```
v = Vector3.Cross(Vector3.forward, Vector3.right); // (0,1,0)
v = Vector3.Cross(Vector3.right, Vector3.forward); // (0,-1,0)
```

That makes it so, by the left-hand rule, it's always a positive rotation from the first to the second (that's a bit sneaky, but it's real trig).

This can be used to find the +/- on a compass spin:

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

Cross product seems like a strange name for the axle arrow. If called that because the equation to make it multiples (which gives a product) the xyz's in a cross-wise pattern.

7.0.3 Normals

A **normal** is a an arrow that tells you which way a surface is facing. It comes straight out of it, with length one.

For examples, the normal of the floor is `Vector3.up`, and the normal of the right-side wall is an arrow pointing left. For a ball, the normal of any spot is an arrow from the middle, through it.

A normal is always coming from some specific surface. Suppose you have a pyramid – a base and four sides. The base's normal is an arrow pointing down. The side's normals are all pointing up at a diagonal.

There are various ways of getting normals. A raycast can get the normal of any collider it hits. This next one aims down. If it hits a tilted ramp, it will give us a tilted normal (otherwise the floor is just (0,1,0)):

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

Getting the normal of official Unity terrain is a bit of a mess. We can ask for it at any point, but we need to convert the real world xz into a 0-1 percent. This looks up the normal of a Terrain where you're standing:

```
public Terrain ground;

TerrainData gd=ground.terrainData;
Vector3 myPos=transform.position, gPos=ground.transform.position;
// convert my position into 0-1 ground x and y:
float gx=(myPos.x-gPos.x)/gd.size.x;
float gy=(myPos.z-gPos.z)/gd.size.z;
// lookup:
Vector3 gNorm=gd.GetInterpolatedNormal(gx, gy);
```

As usual, it will tend to be (0,1,0). But it will tilt left on the left side of a hill, for example. It can be used to find the uphill and downhill directions.

If you have 3 corners of any flat surface, you can use the cross-product to get the normal:

```
v1

                v2
    v0

Vector3 normal = Vector3.Cross(v1-v0, v2-v0);
```

7.0.4 Reflect

This is getting a little obscure – I've never used it. The `Reflect` command bounces an arrow off of a surface. You pretend the tip of the arrow is against the wall, and Reflect tells you which way it bounces off. It uses the wall's normal to know how to bounce.

This slightly complex code shoots a laser, using a raycast, and bounces off anything it hits named `"wall"`:

```
Vector3 laserDir;

RaycastHit RH = new RaycastHit();
bool gotaHit=false;
if(Physics.Raycast(transform.position, laserDir, out RH)) {
    if(RH.transform.name!="wall") gotaHit=true;
    else {
      // bounce off wall:
      Vector3 hitPos = RH.point;
      Vector3 wallNorm = RH.normal;
      Vector3 dir2=Vector3.Reflect(laserDir, wallNorm);
      print("hit wall, bouncing");
      // now shoot from where we bounced:
      if(Physics.Raycast(hitPos, dir2, out RH)) {
        if(RH.transform.name!="wall") gotaHit=true;
```

```
    }
  }
  if(gotaHit) print("We hit "+RH.transform.name);
}
```

7.0.5 Opposite of an angle

`Quaternion.Inverse(q);` is the proper way to write `-q`. Put another way, suppose you want to subtract q2 from q1. Write it like: `q1*Quaternion.Inverse(q2)`.

Of course, it's up to you to figure out whether to subtract it as a local or a global. You may need to use `Quaternion.Inverse(q2)*q1` it you need to subtract it as a global.

You don't need this very often since we can naturally make the opposite of most rotations anyway:

- `FromToRotation(A,B)` is opposite `FromToRotation(B,A)`.
- `Quaternion.Euler(0,-y,0)` is clearly the opposite of `Quaternion.Euler(0,y,0)`.

  But that's not true for xyz. In `Quaternion.Euler(x,y,z)` you can't just flip every xyz. To test this:

  ```
  float x,y,z; // use any values
  Quaternion q1 = Quaternion.Euler(x,y,z);
  Quaternion q2 = Quaternion.Euler(-x,-y,-z);

  print((q1*q2).eulerAngles); // not (000). These are not opposite
  Quaternion q1Inv=Quaternion.Inverse(q1);
  print((q1*q1Inv).eulerAngles); // 000
  print(q1Inv.eulerAngles); // these will be some funny numbers
  ```

- `Quaternion.LerpUnclamped(q, Quaternion.identity, 2.0);` is a longer way to flip any rotation. There's no reason to use it now that we know about Inverse. The 2.0 takes it from `q`, all the way down to 000, then past to `-q`.

We can use Inverse to make Quaternion.LookRotation(q1,q2) (the one we have now only works for vectors). We can make it by "subtracting" q1 from q2:

```
// sample rotations:
Quaternion q1=Quaternion.Euler(30,-60,10);
Quaternion q2=Quaternion.Euler(20,-45,22);

// q2-q1, as a global rotation:
Quaternion q1to2global=Quaternion.Inverse(q1)*q2;
// test using q1 + q12:
```

```
print((q1to2global*q1).eulerAngles); // (20,-45,22)

// q2-q1, on q1's local axes:
Quaternion q1to2Local=q2*Quaternion.Inverse(q1);
// test using q1+q12:
print((q1*q1to2Local).eulerAngles); // (20,-45,22)
```

7.0.6 Square magnitude

There's a built-in function that gives you the *square* of the length of a line. If `v` has length 3, `v.sqrMagnitude` is 9.

That seems insanely pointless. Why would you need your length squared? And if you did, wouldn't it just be easier to write `len*len`?

It's just a trick to speed up the math, and does nothing else useful. Here's the explanation:

The formula to find the length of an arrow is `Mathf.Sqrt(x*x+y*y+z*z)`. In other words, when you run `v.magnitude` the first step gets 9, then the second step square roots it to get 3.

When you use sqrMagnitude, you're saying to save time by only running step 1, and you'll take it from there.

Here's how that can save time: Suppose you want to find the nearest enemy, from you. Find everyone's `sqrMagnitude` and use the smallest. We don't care about the actual distance, so we may as well compare the one that's faster to compute.

The other use: suppose you want to find everything 3 away from you. Take everyone's `sqrMagnitude` and check for 9 or less.

The important thing is, `sqrMagnitude` does nothing useful – it can sometimes be a smidge faster, but makes you think harder to write the program.

7.1 Trig you should almost never use

In school we learned that angles and rotations use trig. They do, but it's almost all built-in to the system. There's rarely a need to use it in 3D.

7.1.1 Radians

Real trig functions, like sin, cosine ... use radians instead of degrees. They're different in three ways:

- 1 radian is about 57 degrees.
  For real, there are 2PI radians in a circle, which is a repeating decimal: 6.283185 .... So 90 degrees is 1.57079 ... radians. Ug.

- 0 radians is facing along +x (instead of +z).
- Radians go counter-clockwise. 0 is right, 1.57 is forward, 3.14 is left.

What this means is if you have a trig angle of 2, that's 114 degrees, but you're not done. It also starts on +x and goes backwards. In Unity math it's really -24 degrees.

Unity has a built-in `Mathf.Rad2Deg`, but it's just 57. It doesn't take care of the +90 counter-clockwise part. Likewise `Mathf.Deg2Rad` is just the number 1/57.

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

```
degs=-rads*Mathf.Rad2Deg+90; // radians to unity degrees

rads=-degs*Mathf.Deg2Rad+Mathf.PI/2; // unity degrees to radians
```

7.1.2 atan2

In trig, arc-tangent turns a slope into the angle (in radians). But it won't work for 90 degrees and gives the answer+180 for anything aiming left.

A computer trick fixes that. Instead of computing the slope, we give it x and y. This would give us the angle we're facing:

```
Vector3 fwd=transform.forward;
float ang=Mathf.Atan2(fwd.z, fwd.x);
```

That gives `ang` in radians, so we're not done. Ugg.

atan2 is useful in a few special circumstances, but for most problems it's simpler to use other commands.

7.1.3 dot product

Dot product tells you the angle between two arrows, sort of. They have to be length 1 (you can normalize them,) and it gives the cosine of the angle, which can be converted to radians. It also can't tell left from right, the same as Vector3.Angle.

Vector3.Angle is just better.

7.1.4 Rotation matrixes

Another standard computer way of representing rotations is a 4x4 grid, called a Rotation Matrix. In Unity, a `Matrix4x4`.

Graphics cards use these, so you might see them in Shaders. Some of the older GUI items use them. The camera has one.

But quaternions are better for most things. There's no reason to use a rotation matrix unless you're required to.