Chapter 7

Misc Math

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

7.0.1 Finding angles

`Vector3.Angle(v1,v2)` tells you the angle in degrees between two arrows. It treats them as if they were both coming from the same spot, and finds the angle of the V they make.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

7.0.2 Cross Product

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

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

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

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

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

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

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

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

7.0.3 Normals

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

For examples, the normal of the floor is `Vector3.up` and the normal of the right-side wall is an arrow pointing left. If you have a left-to-right ramp (like a /-slash,) its normal would be pointing up and to the left. Even curved surfaces have normals, but in a game most round things are made of small flat edges.

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

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

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

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

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

This looks up the normal of a Terrain where you're standing:

```
public Terrain ground;

TerrainData gd=ground.terrainData;
Vector3 myPos=transform.position, gPos=ground.transform.position;
```

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

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

7.0.4 Reflect

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

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

```
Vector3 laserDir;

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

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

7.0.5 Opposite of an angle

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

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

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

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

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

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

7.0.6 Square magnitude

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

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

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

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

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

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

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

7.1 Trig you should never use

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

7.1.1 Radians

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

- 1 radian is about 57 degrees.
  For real, there are 2PI radians in a circle, which is a repeating decimal: 6.283185 .... So 90 degrees is 1.57079 ... radians. Ug.
- 0 radians is facing along +x (instead of +z).
- Radians go counter-clockwise. 0 is right, 1.57 is forward, 3.14 is left.

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

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

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

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

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

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

7.1.2 atan2

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

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

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

7.1.3 dot product

Dot product tells you the angle between two arrows, sort of. They have to be length 1 (you can normalize them,) and it tells you 1 to -1. Going the same direction is 1, 90 degrees apart gives 0, and exactly opposite gives -1.

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

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

7.1.4 Rotation matrixes

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

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

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