Chapter 1

Vectors and offsets

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1.1 Points and Offsets

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1.1.1 Tilted checkerboard

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

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

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

```
Vector3 leftToRight = cornerLR - cornerLL;

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

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

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

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

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

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

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

000 (could be anywhere)
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1.2 `transform.position` + offset

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

  Vector3 toRed = redCube.position - transform.position;

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

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

1.2.1 Camera vector positioning

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1.2.2 Speed vectors

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

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

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

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

public Transform greenCube;

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

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

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

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

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

1.2.3 Averaging points

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

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

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

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

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

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

1.3 Math review

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

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

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

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

```
    p1+p2(B)

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

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

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

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

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

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

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

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

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

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

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

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