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?"