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;
```

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*. Each pair adds, and the result also has three parts. More specific: add x to x, y to y and z to z. 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)
```

It's useful, and we'll use it lots, but it's only 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;`.

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 that 3 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;`

Unity doesn't define pairwise multiplication: `A*B` isn't allowed. If you need it, which you rarely will, you can write it out in three lines. It also doesn't define scalar addition, such as `A+2`, but you also rarely need it.

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*0.5f` is an easy way to make (0.5, 0.5, 0.5).

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 for creating points. `Vector3.up*5` is a nice-looking way to write (0,5,0). Or we can combine them (scalars and addition) to make any point:

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

That's the same as `new Vector3(0,9,-4);`, except longer. But maybe it's easier to read it as "4 backwards and 9 up".

1.1 Points and Offsets

The first trick is knowing an x,y,z can be either a point, or an offset, depending on how we think about it. A point is the easy one – a spot on the map. Offsets

are arrows, which we add to points. An offset of (0,5,0) means "5 above some other spot".

`Vector3.right` and it's friends could be used as either, but are usually offsets, which is why they have those names. A typical use would be `transform.position+Vector3.right`, meaning "one space right, from me".

Suppose we're building a game board. We'll start by letting the user set the lower-left corner. It's a simple point:

```
public Vector3 cornerLL; // user will enter
```

The board squares will run sideways and forward from there. To help place them we'll make two offsets:

```
Vector3 sqSdways=new Vector3(1.2f, 0, 0); // arrow for 1 square over
Vector3 sqFwd=new Vector3(0, 0, 1.2f); // arrow for 1 square forward
```

These are offsets because of how we plan to use them. `sqSdways` isn't a spot on the map – the board may be nowhere near the point (0,0,1.2). But it's great as a single-square sideways arrow.

Using our vector math, we can compute positions of some sample squares. We start at the corner, and walk squares over:

```
s10 = cornerLL + sqSdways;
s30 = cornerLL + sqSdways*3;
```

The second one is so cool, how it clearly says "three squares sideways from the corner". It also shows a rule: an arrow times a scalar is going the same direction, but a different distance.

Adding two offsets is like following two different end-to-end arrows. This line says to start at the corner and walk 3 sideways and 2 forwards:

```
s32 = cornerLL + sqSdways*3 + sqFwd*2;
```

It puts us exactly at the corner of that particular square.

For more fun, assume we're placing 3D square models, which have their origin centered. Instead of walking to corners, we need to walk to centers – an extra 1/2 a square sideways and forwards. We can do that painlessly with another offset vector:

```
Vector3 cornerToCenter = sqSdways/2 + sqFwd/2;
```

The end result is a short diagonal arrow. Dividing by 2 is using a scaler – it's the same as `*0.5f`, but I though it looked nicer this way.

To place our squares we add that extra offset:

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

```
              o
              | fwd*2
      sdway*3 |
      -- -- --
cToc /
   LL
```

1.1.1 More calculating offsets

A board lined-up on x and z isn't really showing the advantage of offsets very well. We know sideways is just +x and forward is +z. But with a tilted board, vector math is our best friend.

Instead of only one corner, let's have someone hand-enter three corners. Depending, the board could be tilted any which way (assume they make both sides the same length, and at right angles):

```
// lower-left, upper-left, lower-right
public Vector3 cornerLL; // lower-left corner from before
public Vector3 cornerUL, cornerLR;
```

```
UL
 \    diagonal
  \  game board
   \           ---LR
    \      --/
     LL--/
```

Since we know the board is 8 squares across, each square is 1/8th of the way. Instead of entering the sideways and forward square offsets, we can compute them:

```
Vector3 sqSdways = (cornerLR-cornerLL)/8;
Vector3 sqFwd = (cornerUL-cornerLL)/8; // same
```

The new thing here is that subtracting 2 points creates an arrow. The right corner minus the left corner computes an arrow along the whole bottom.

1/8th of that arrow is our sideways 1-square arrow. Pretty slick.

The really neat thing is how finding the square centers is the same as before:

```
// sample placement (same math as before):
cornerToCenter = sqSdways/2 + sqFwd/2;

sq32 = cornerLL + cornerToCenter + sqSdways*3 + sqFwd*2;
```

One more neat arrow trick: we can compute the missing upper-right corner, using the other three. The plan is: get the arrow going up along the left side; then add that to the lower-right corner:

```
Vector3 bottomToTop = cornerUL-cornerLL; // arrow up left-side

Vector3 cornerUR = cornerLR+bottomToTop;
```

Another plan would be to start at the upper left, and march 8 squares over:

```
Vector3 cornerUR = cornerUL+sqSdways*8;
```

One last thing about this entire game board example: if we knew how to rotate an arrow, we could have them enter only the two bottom corners. We'd find the arrow between them, rotate it 90 degrees, and use that to get the other two corners.

1.2 `transform.position` + offset

Let's assume we're a Cube, with a script on us. Our location, which is a point, is in `transform.position`. We can use that, plus offsets, to place things around us.

Assume that besides us, we have three cubes: red, green and blue. No scripts, just Cubes waiting for us to reach out and position them. This simple code places them at various spots around us:

```
public Transform redCube, greenCube, blueCube;
// assume hand-linked to real cubes through Inspector

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

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

  // blue piggy-back off green:
  blueCube.position = transform.position + greenOff*1.5f;

}
```

The point+offset math is nothing new. But now we're really placing something there.

If you want to see this really work, put a Rigidbody on us, add a bouncy material, make a floor, a ramp ... and let us knock around. The colored cubes will track us perfectly. Even when we spin, they stay locked in.

For a variant, let's say someone enters one long arrow, coming from us, where we want the colored cubes evenly spaced. We can do that with scalars:

```
public cubeLine;
// ex: (2,8,0) stacks the cubes up, leaning right, with gaps

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

The scalar determines where they go on the line.

A cool idea that gives us: what if we take just one cube and slide the scalar from 0 to 1 (in the code)? It will move along the line:

```
public Vector3 cubeLine;
float pct = 0.0f; // we'll make this go from 0 to 1
public Transform greenCube;

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

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

Stepping back, writing code to shoot out along any line seems like it might involve lots of math. But thinking of it as offsets and scalars makes it easy.

To jazz that up, we can use the trick of computing an arrow between any two points. Instead of entering `cubeLine`, let's say the red cube is the other end. Where ever we are, we want the green cube to shoot from us to the red one.

The only change is computing `cubeLine` as the red cube's position minus ours:

```
// shoot green cube from us to the red one:
float pct = 0.0f;
public Transform redCube, greenCube;

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;
}
```

This next one is simpler. Multiplying by a negative scalar flips an arrow. We'd like the green cube to hide behind us (from the red cube). We'll can take that same arrow from us to red, and take it times -0.1:

```
// green cube hiding behind us from red cube:
  Vector3 toRed = redCube.position - transform.position;

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

1.2.1 Camera vector positioning

Instead of placing a green cube nearby, we can use the same tricks to place the camera. It will appear to be tracking us.

This code puts the camera above us, and a little ahead:

```
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;
}
```

The camera would need to be aimed downwards (a 90 degree spin on x). If we have the rigidbody bouncey set-up from before, the camera-tracking looks pretty cool. But if you go to Scene view and watch, it's no different than when we made the cubes track us.

We can give it a camera zoom by scaling the offset. We'll add a `zoomPct` variable and have up/down arrow keys change it:

```
public float zoomPct=1; // 1=full length
// zoomAmt -- to camera arrow multiplied by this

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

  // arrow keys zoom in/out:
```

```
  if(Input.getKey(KeyCode.UpArrow) zoomPct-=0.01f;
  if(Input.getKey(KeyCode.DownArrow) zoomPct+=0.01f;
}
```

Real zooms don't go straight to our center, like this does. They have a little offset – like going to a spot above our shoulder or something. That's easy to make by using two arrows, and scaling only the second one:

```
  |
  | offset#2, zooms in/out
  /
 / offset#1, always this long
Player
```

The code is pretty simple: add offset#1, then offset#2 times the zoom percent:

```
  Vector3 toCam1 = Vector3.up*8+Vector3.forward*3;
  Vector3 toCam2 = Vector3.up*10;

  theCamera.position = transform.position + toCam1 + toCam2*zoom;
```

Mathwise, it's just a point plus two offsets, which we're done before. We just never had a reason to scale only one of the arrows.

1.2.2 Moving with vectors

Vectors are a nice way of storing "this is how much I move each frame". They say which direction, and how fast. We like to say the arrow is how much we move each second. Each update we'll move a fraction of that.

This moves the green cube along whatever vector the user enters:

```
public Vector3 greenMvArrow;
public Transform greenCube;

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

I like how the += feels like what it does. `x+=0.1f;` moves `x` a little. `greenCube.position += someVector3;` does the same thing, except in 3D space.

Of course, we could move ourself, using the same method, with `transform.position += mvArrow/60;`

We we moving the green cube before, using a more complicated method. We needed to know both end points (us and the red cube) and the percent. It always stayed on the line, even if one end moved. Since we always knew what

percent of the way we were, it was easy to stop when we got there, and go back to the start.

This method is simpler, for when we're happy to just go in a direction.

One note: I'm assuming Update runs 60 times/second. If you know about `Time.deltaTime` (or want to look up that trick), we'd definitely use that instead of 60.

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. If 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;
```

The code is a little longer, but we can adjust the 0.5 to whatever we need. Average is really just a shortcut for this, when we're sure we'll only want 50% of the way and won't need to adjust it later.

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 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, `(A+B)/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 you have `sq.position=A+B+C;` and it's working wrong. Check whether A counts as a point, and B and C are 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?"

Finally, two fun errors. It's easy to accidentally use an offset by itself, forgetting to to add the starting point:

```
greenCube.position = toRed*pct; // <- oops
// Forgot to start at our position
```

This is like moving the line to start at (000). It's the same length and direction, but in the wrong place. It tends to be really confusing since 000 usually isn't any special spot in your game. Depending where you are, the line can be nearly correct or completely off the screen.

If you get a line that appears to shift in funny way, check that you added the starting point.

It's also easy to flip the arrow direction by mistake. It's the end point minus the start point. If you see `transform.position - redCube.position`, that's as arrow going *to* you, *from* the red cube. Nothing wrong with that, but if you follow it from you, it goes away instead of towards.