Chapter 4

Input, more output, testing

This section starts with Unity3D tricks: a simple way to get input, another way to get output, and how to make the program keep running and doing things. But that won't take long, and we'll be able to use them to write some more interesting programs.

Most of this chapter will be about how we can use variables and the `n=n+1;` trick to make the computer do things.

Then, near the end, we'll move a Cube around on the screen, by adding just one extra command.

4.1 Input

Unity uses an old trick where it can show you the "box" for a variable. Typing into it is a shortcut to give that variable a starting value. Then, while the program runs, we can watch our variables change, or even cheat by hand-changing them.

This seems like cheating, but the secret is all real input is about putting a value in a variable. For example, suppose we had a drop-down with five options. There would be rules to set it up, but the end result is an `int` changing from 1 to 5. Output is the same way. If we have an auto-centered text-box with a pretty font, we'd still create a `string` and assign it to the text-box.

So changing and looking at variables directly is a completely fair way to do input/output. It's ugly and confusing if you're not a programmer. Luckily we are.

To do it we have to declare the variable outside of `Start()` with the word `public` in front. Here's a silly sample program that lets us see `x`, `w1` and `w2`. I named it `testB`, but you could just as easily reuse `testA`.

Note how the declarations are outside of Start, but are still inside the big {}s for `testB`. The indentation (even with Start) is also to help us see that:

```
Using UnityEngine;
using System.Collections;

public class testB : MonoBehaviour {

    public int x; //  outside of Start, public in front
    public string w1; //  same

    public string w2; // this one, too

    void Start() {
      x = x + 1; // just for fun
      w2 = w1 + x + "ous";
    }
}
```

Once you have this script set up (dragged onto the Directional Light) you'll see the cool part. Select the Light and look at the Inspector. If it isn't already, click the arrow to "pop open" the tab with the script. You should see new lines with `x 0`, `w1` and `w2`. You can click and type in new values (the boxes for strings are sometimes further right than you think, so keep clicking along the row.)

When you press Play (don't press Stop yet) you should see `w2` and `x` change. Pressing Stop makes them snap back (it just does.) Try entering `"ham"` for `w1`. Pressing Play again should show `w2` is `"ham1ous"`.

Not super exciting, but it's a quick, easy way to test this program for lots of different inputs and see the results. Try `3+4` for `w1`. You should see `3+41ous` for `w2`, proving it's using string-add.

Here are some notes, to explain this more:

- Using this trick to set starting values is a cheat – don't let it confuse you about how programs work. When you press Play, the Unity system jumps in and slaps those Inspector values into the variables. Then it runs Start() as normal.
- The rule about variables snapping back when you Stop is suppose to make things easier, for when you test real programs. Don't worry too much about it.
- Typing new values *during Play* won't recompute anything, yet. Entering `cow` for `w1` won't update `w2`. It would be nicer if it did, and later programs will.
- Fun fact: the system saves what you type in there. Even if you quit Unity and come back, whatever you entered for `w1` will still be there.
- Putting variables outside of `Start` with `public` in front is a normal C# rule. We'll use it for its real purpose later on. Unity happens to piggy-back on it to decide what variables to show you.

- Notice how we didn't need quotes for strings in the inspector. It already knows everything you type in `w1` is one string. We only need quotes inside the program.
- Whatever you enter for `w2` doesn't matter, which should make sense. The program only uses `w2=`, and we know that completely erases the old value. In out minds, `w2` is for output.

Here's a slightly different "Mad Lib" example. `animal`, `food` and `place` are the input variables:

```
Using UnityEngine;
using System.Collections;

public class testB : MonoBehaviour {

    public string animal, food, place;
    public string w;

    void Start() {
      w="The " + animal + " eats " + food + " in the " + place;
      print(w); // as a double-check
    }
}
```

We can enter values, Play, and check `w` to see how it looks. The neat thing is, a real program would compute `w` just like this does. The only difference is it would then copy it into a fancy text-box somewhere on the screen.

4.2 Using Update to make more interesting programs

A traditional program runs once, outputs, and is done. `Start` in Unity works this way.

There are a few tricks to making a program that runs over time. Unity uses a common one. You write a regular run-once program, but you tell the system to run it over and over. That's what `Update` is for.

The original starter program looked something like this (showing the important parts):

```
  void Start() {
  }

  void Update() {
  }
```

Anything you put inside the `Start` curly-braces will run once. Anything in the `Update` curly-braces runs over-and-over. You're allowed to use either one by itself, or both.

Here's an example showing how they work together. I'm putting them each on one line, to save space:

```
void Start() { print("begin now"); }

void Update() { print("x"); }
```

If you press Play, wait a little, then Stop, you"ll see one "begin now" followed by x's blasted down the page (you may have to scroll up.) If you only see a single `x`, you probably have **collapse** checked on the Console, which won't show duplicate lines. Just uncheck it, at the top.

Notice how the inside of `Update` doesn't have anything about repeating. It's completely normal. The system just knows it should run `Update` over and over.

Here's a program taking advantage of Update(). It adds one to `x` over-and-over and lets us watch:

```
public int x;

void Update() { x=x+1; }
```

This is fundamentally no different than when we had several `x=x+1;`'s in a row inside Start(). We're just using a trick to make it run lots, spread over time.

If you know games, you may notice it's going too fast. Update should be running 60 times each second. A quirk of Unity is when you run things this way it goes as fast as possible, usually a few hundred times per second. That's fine for our purposes.

If you don't know games, each time Update runs is usually called a `frame` or a `tick`.

We can use the Update trick to improve the Mad Lib program. Moving it there makes it so changes while running immediately update the answer (this version is a little different than before, but the same idea):

```
  public int eatAmt;
  public string animal, food;
  public string w; // output

  void Update() { // <- in Update instead of Start
    eatAmt = eatAmt + 1;
    w = "The " + animal + " eats " + eatAmt + " " + food + ".";
}
```

I had `eatAmt` keep increasing just so we can see this really is running over-and-over. It's not super exciting, but we can now change `animal` or `food`, while running, and have `w` change.

Here's a little more interesting program. It computes hours and seconds based on minutes:

```
public float minutes; // input
public float hours, seconds; // outputs

void Update() {
  hours = minutes/60.0f;
  seconds = minutes*60.0f;
}
```

If we type (while this is running) 120 into the minutes box, we should see hours snap to 2, and seconds snap to 7200.

The Update trick works with any other "change me" assignments. This next program makes `y` go down very slowly, and computes 5% interest on `x`:

```
public float x, y; // <- shortcut, but we still get a box for each

void Update() {
  y = y - 0.01f; // <- floats let us use a very small number
  x = x * 1.05f;
}
```

At first `x` won't change, since anything times 0 is 0. But if you change it to 1, it will go up slowly, then faster and faster, like compound interest. That's not really useful yet – just showing how the "change me" assign tricks from the last chapter work here.

4.3 declare = shortcut and Inspector vars

We can use the declare-and-equals trick with `public` "Inspector" variables, but there are some tricks. This will start `x` at 100, for real and in the Inspector:

```
public int x=100;

void Start() {
  x=x+1;
}
```

But what happens if you change Inspector `x` to -999? The rule is: the Inspector wins, and `x` starts at -999, even though the code still says it starts

at 100. That's because Unity jumps in and changes it to -999 before running Start.

This is horribly, horribly, horribly confusing if you try to read the program and forget. For real, variables never just magically change their values. But it makes testing programs much easier – what you see in the Inspector is always what you get.

One way not to be confused by this rule is not to use the assign trick. But I do it anyway, to give the Inspector that starting value. If you use Unity for real, this trick is handy.

4.4 Replacing constants with variables

Here's a program that goes up by 1 each tick:

```
public float x;
public float xSpeed = 1.0f;

void Update() { x = x + xSpeed; }
```

The old version used `x=x+1;`. The new version uses `x=x+xSpeed;`, but `xSpeed` is always 1, so it's the same.

It seems overly complicated, but there's one advantage: we can change `xSpeed` while we're running.

Press Play, let `x` go up by 1 for a while, then type 0.1 for `xSpeed` – `x` will go up 1/10th as fast. Try -1 and `x` starts going down. 0 for `xSpeed` makes it stop (it's still adding, just adding 0.)

Here's an even more interesting version. It starts out increasing by 1, but then increases by a little less each time (it has one new line):

```
public float x;
public float xSpeed = 1.0f;

void Update() {
  x = x + xSpeed;
  xSpeed = xSpeed - 0.005f; // slow down a tiny bit
}
```

`x` will go up, slow down, almost pause, then go back down. That's kind of neat, for a 2-line program.

This is one of the Big Ideas in programming: they're your variables, and they mean what you say they mean. `xSpeed` is how fast `x` changes, because we made it that way – because of the first line. `xSpeed` of 1.0 means `x` is going up quickly, 0 means it isn't changing, negative means `x` is going down. Those

numbers would mean different things for some other variable, but they mean this for `xSpeed`.

In the second line `xSpeed=xSpeed-0.005f;` means "make `x` slow down." It's also just a regular line that subtracts a little from a variable. But it means "slow down" in this program.

4.5 Math shortcuts

`numOfCats = numOfCats + 1;` adds 1 to how many cats we have, but it seems like a waste having to write `numOfCats` twice.

There's a family of shortcuts for modifying a variable, so you only have to write it once.

`numOfCats += 4;` is a shortcut for `numOfCats = numOfCats + 4;`. The `+=` counts as one symbol, so it can't have spaces or be `=+`. You can think of it as saying "numOfCats gets increased by 4."

It also works for floats and strings:

```
x+=0.1f;  // same as x=x+0.1f;
w+="ly"; // same as w=w+"ly";
// no shortcut to add to the front of a string
```

The same shortcut works with other math symbols: `-=`, `*=` and `/=`:

```
numOfCats *=2; // numOfCats = numOfCats * 2;
cows -=6; // cows = cows - 6;
q /=2; // q = q/2;
```

These tricks are very common. You almost never see someone write this out the long way. But they're only shortcuts – they don't even run any faster than if you use the long way. Use them when you get sick of writing the same variable twice.

There's one tricky part – the entire right side is always computed first. Ex:

```
x *= 1+1; // same as: x = x * 2;
cows *= n+1; // same as: cows = cows * (n+1);
```

You usually don't use the shortcut for anything complicated like this. I just write these out the long way.

There's a super shortcut for just adding or subtracting 1. That seems excessive, but we do it a lot. `x++;` adds 1 to `x`. `x--;` subtracts 1 from `x`. There isn't even an `=`. Just `x++;` all by itself adds one:

```
// all do the same thing:
x++;
x += 1;
x = x + 1;
```

The official name of `++` is **increment** (coming from the root word, increase.) And `--` is called a **decrement**.

4.6 Fun 3D stuff

With just a little extra work, we can take a program that moves `x` around, and have it also move a Cube on the screen around, using `x`. All we need is a rule to apply `x` to the position of some Cube.

Here are the steps, then an explanation later. Make sure it isn't in Play mode for any of this:

- Add a Cube: click GameObject (top bar) slide down to 3DObject, then select Cube (or any shape, really.) Don't worry about where it is or whether you can see it now.
- The camera should already be in the right spot. But check and change it if isn't. Select the camera, and look in the Inspector, at the top part labeled Transform. It should have (0, 1, -10) for the x,y,z under Position. And (0, 0, 0) for the x,y,z in Rotation. If it doesn't, type those values in yourself.
- Use this script (as usual, I'm leaving out the `using` lines on top.) It's a small rewrite of the changing-`x` code. If you reuse it, change `x` and `xSpeed` to those new values in the Inspector:

  ```
  public float x=-7.0f;

  public float xSpeed=0.1f;

  void Update() {
    x = x + xSpeed;

    // this is the new part:
    transform.position = new Vector3(x, 0, 0);
  }
  ```

- Find the Cube you made (in the Hierarchy panel) and drag this script onto it (select the Cube, drag the script into the blank space at the bottom of the Inspector.) It should look like the usual script – should see `x` and `xSpeed` variables displayed.

  If it's on the Directional Light, Remove it from that one.

- Play should make the Cube move from left to right (then off the edge forever.)

Here's the explanation of the parts:

The things in the Scene are officially called `GameObjects`. The GameObject->3DObject menu has simple testing shapes, like a Cube or a Sphere. Any of them are good enough.

When you make it, the system puts it "in front" of you, which could be anywhere. Since our program will position the object, it doesn't matter where it starts.

Way back, I mentioned how scripts are intended to be on their Cube, which they move around. We're using that rule now. The last line of that script says "move me," the Cube it's on counts as the "me".

This really is the entire movement command:

```
transform.position = new Vector3(x, y, z); // moves us to (x,y,z)
```

If you look in the Cube's Inspector, in the Transform block, in the Position slot, you'll see it changing.

For now, it's "just because." All we need to know about it is that x, y and z can be any `float` or float variable.

There's no natural size of the screen. It depends on the position of the camera and how it's aimed and even the particular size of your Scene/Game area. With the camera in the starting position, the left side is about x=-7, which is why the code starts us there. The right side is about x=+7.

Since the screen is only 14 across, a speed of 0.1 feels about right.

To get a feel how that last line works, we can change things around:

- Start x at -8 (in the Inspector.) That should make the Cube come in from off-screen.
- Change `xSpeed` to 0.005 (half speed) or 0.5. From before, we know this will just make `x` go slower or faster. But it's pretty cool seeing the Cube do that.
- In the last line, change the middle 0 to a 3: `Vector3(x,3,0);`. It should go across more near the top of the screen. So x is across and y is up. `Vector3(x,-4,0);` should make it go more near the bottom.
- Change `x` to 7 and `xSpeed` to -0.1. We know this will just make `x` start at 7 and go down, but now it also makes the Cube go from right to left.
- Change the last line to `Vector3(0,x,0);` (`x` in the second slot instead of the first.) That should make the Cube go from bottom to top.

We already know the middle number is up/down. The variable happens to be named `x`, but variable names don't matter.

- Change the last line to `Vector3(0,0,x);`. The Cube should come right at us, then fly underneath us. The last slot is for how far/close it is from us.

I think of this as 2 parts. The new part is the exact command to set position and what the slots in `Vector3( _, _, _)` mean and what the numbers are for the edges of the screen. That only seems tricky because it's new.

The second part is doing the real work, which is just making `x` move around, the same as we did before.

Most cool-looking programs break down that way. Once you understand the rules for using some neat thing, the rest is regular programming to make the numbers change how they should.

4.7 more cute cube tricks

Before we knew how to move Cubes, we could make more than one variable change at once, but we didn't have a reason. Now, with Cubes, we do.

If we change `x` and `y` at the same time, the Cube can move at an angle:

```
public float x=-8.0f; // just off the left side
public float y=-4.0f; // near the bottom

public float xSpeed=0.1f;
public float ySpeed=0.1f;

void Update() {
  x = x + xSpeed;
  y = y + ySpeed;

  transform.position = new Vector3(x, y, 0);
}
```

That should make the cube fly from the lower-left to the upper-right.

We can change the angle by changing either speed. Or have it go lower-right to upper-left (by making `x` go from 7 backwards.) And so on.

We already know how to make `x` go up, slow, then back down. We can reuse that for a Cube that goes right, slows, then goes back left:

```
public float x=-7.0f;
public float xSpeed=0.1f;

void Update() {
```

```
  x = x + xSpeed;
  xSpeed = xSpeed - 0.0002f; // <- new line

  transform.position = new Vector3(x, 0, 0);
}
```

In the old version, we didn't really worry about the exact value where `x` turned around and started getting smaller. Now we'd like it to turn around somewhere around 7. So we may have to tweak the starting speed, or the slowdown.

I use just trial and error for that.

We can use the "variables are better than constants" rule again, by turning that -0.0002 slowdown into a variable. I'm going to name the variable `xAccel`:

```
public float x=-7.0f;
public float xSpeed=0.1f;
public float xAccel=-0.0002f; // new. x acceleration variable

void Update() {
  x = x + xSpeed;
  xSpeed = xSpeed + xAccel; // using the new variable here

  transform.position = new Vector3(x, 0, 0);
}
```

Now we can just adjust the slowdown in the Inspector, instead of having to go back to the code each time.

A fun trick is to start `xSpeed` at 0, and `xAccel` at a very small positive number. The cube will speed up from a standstill. Or, set `xSpeed` negative and `xAccel` positive – it will go left, slow, then right.

4.7.1 Colors

If we want to change the Cube's color, we have to know the command, and what the values mean. This makes the Cube turn red:

```
GetComponent<Renderer>().material.color = new Color(1, 0, 0);
```

The first part is very strange-looking, but it works and changing color is fun enough to be worth it. The pop-up will even help us. Note the `R` in Renderer is capital, and those really are a less-than and greater-than (they're being used as angle-brackets.)

Like before, we need to know what the 3 values in parens mean. They stand for red, green and blue, in that order, and go from 0 to 1. For example, `Color(0,0.3f,0);` makes dark green.

Now we can use the same tricks to move the color. This slowly colors the Cube from black to red (if the Cube is off the screen, you may have to manually enter 0,0,0 for the position in Transform in the Inspector.):

```
public float r = 0;
public float rSpeed = 0.005f; // how fast it changes

void Update() {
  r = r + rSpeed;

  GetComponent<Renderer>().material.color = new Color(r,0,0);
}
```

We can try the usual tricks:

Start `r` at 1 and have `rSpeed` be negative (so it starts red and goes to black.) Use the second slot – `Color(0,r,0)` – to make green change, or `Color(0,0,r)` to change blue. Or use something besides 0's – `Color(1,r,1)` goes from purple to white.

Another interesting one is putting the variable in two places – `Color(r,r,0)`. The same variable controls red and green (which happens to make yellow.)

We can change red, green and blue at once by making a variable for each. These particular numbers change them at different speeds, making kind of an interesting pattern. You're probably sick of seeing the long way, so I'll start using the `+=` shortcut:

```
public float r=0.0f, g=1.0f, b=1.0f;
public float rSpeed = 0.002f, gSpeed = -0.002f, bSpeed=-0.001f;

void Update() {
  r += rSpeed; g += gSpeed; b += bSpeed;

  GetComponent<Renderer>().material.color = new Color(r, g, b);
}
```

4.7.2 Scale

To make us twice as big, use `transform.localScale = new Vector3(2,2,2);`. As usual, we need to know what the numbers mean.

You may have guessed from the `Vector3`, that they stand for x, y and z. 3D programs don't use just one number for size. They let you stretch it wider, taller and deeper (along the x-axis, y-axis and z-axis) in different amounts. If you want to just make the whole thing bigger or smaller, you have to use the same number for all 3.

Here are a few uses of it:

```
transform.localScale = new Vector3(1, 2, 1); // twice as tall
transform.localScale = new Vector3(1, 0.5f, 1); // half as tall
transform.localScale = new Vector3(2, 1, 1); // twice as wide
transform.localScale = new Vector3(5, 0.2f, 1); // long, sideways pole
transform.localScale = new Vector3(0.1f, 0.1f, 0.1f); // 1/10th size mini-Cube
```

The numbers aren't officially the size – they're a multiplier. But Cubes' base size is 1x1x1, so for Cubes, the scale is also the size.

This next program makes the Cube get wider and taller, in sort of a fun pattern. It starts not as wide, but it gets wider faster than it gets taller. I used `xSizeSpeed` for how fast the x-size changes since it was the best variable name I could think of:

```
float xSize=1.0f, ySize=2.0f; // our x and y scale
float xSizeSpeed = 0.05f, ySizeSpeed = 0.02f; // how fast scale changes

void Update() {
  xSize += xSizeSpeed; ySize += ySizeSpeed;

  transform.localScale = new Vector3( xSize, ySize, 1);
}
```

As usual, can enter a small negative number for one of the speeds, to see it get smaller (negative size means inside-out, which works, but looks funny.) Or 0 for one speed to not have that size change.

This, and the color change, and the movement are really the same thing - just some variables that get bigger or smaller, then one special Unity command.

But that's the point. A lot of things that look different are just the same simple thing, when we program them.

4.8 Errors, funny stuff

When using the assign shortcuts it's easy to forget the `=` in `+=`, or to add an extra `=`, or an extra `x`. Examples:

```
x+3;  // oops! meant to write x+=3;
// Only assignment, call, increment, decrement, and new
// object expressions can be used as a statement

x ++ 2; // meant to use x+=2;
// Unexpected symbol ‘2’
```

Some mistakes aren't errors, they just do wrong stuff or look funny:

```
x += x+3; // add x+3 to x. Yikes!! Meant to write x+=3;

x = x+= 3; // Works! Adds 3 to x, but meant to use x+=3;

x = x ++; // Also works! Add 1. But meant to use x++;
```

I try to remember that `+=` and `++` are shortcuts. Using them should make a line shorter, not longer.

A script changing color needs to be on a real colorable shape, like a Cube. On a Light it gives the run-time error *"Missing Component Exception: there is no Renderer ....* It's looking for the color slot and can't find it.

The scripts that move or change size won't give errors on a non-Cube, but you won't see anything happen.

Leaving out the `public` in front of a variable isn't an error. You won't see a slot for it in the Inspector, but it will work as normal. This prints 10, 11, 12 ...:

```
int x; // forgot public, but not an error

void Start() { x=10; }

void Update() { x=x+1; print(x); }
```

And, to repeat, remember the inspector wins. If you write `public int x;` an inspector slot with a 0 appears. If you later change it to `public int x=7;`, the 0 will still be in the inspector and overrides the 7.