Chapter 8

Unity `if` examples

Now that we can "think" with ifs, we can improve the moving, coloring, and resizing examples from before. Even if you have no interest in making games, this is a pretty good way to practice writing `if`'s.

Here's the old "move across the screen" code with only `x` changing:

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

void Update() {
  x = x + xSpd;
  transform.position = new Vector3(x, 0, 0);
}
```

Instead of shooting past the right side, we could have it wrap around like the old Asteroids game. We just need to figure out how to say "when it goes past the right side, return to the left."

That's two separate things to figure out. Going to the left side is easy, we've done it before: `x=-7;`. You might think that can't be right – putting `x=-7;` in Update will keep us glued to the left edge. But that's the other part – how to do it only at the correct time.

Checking past the right side is `if(x>7)`. Combining them is: `if(x>7) x=-7;`. Move with wrap-around looks like this:

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

void Update() {
  x = x + xSpd;
  if(x>7) x=-7; // new wrap-around to left side
```

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

That new line is pretty short. A chapter ago it would have been a really bad example – why would we want to change from 7.1 to -7? But now that little `if` does exactly what we need.

It probably looks a little "snappy." If you're interested you could tweak the +7 and 7 numbers for the sides until it looks really smooth. Turning those into variables would let us adjust them as they run. Again, only if you want to see it smooth:

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

public float xMin=-7.0f, xMax=7.0f; // the new left and right side variables

void Update() {
  x = x + xSpd;
  if(x>xMax) x=xMin; // replaced -7 and 7 with variables

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

Once you get used to this stuff, that new line is easier to read. It's directly telling you that past the max jumps to the min.

We can use the same trick to change the size. This grows from normal to double-wide then snaps back and repeats (remember 1 means normal-size):

```
public float xSz=1.0f;

void Update() {
  xSz += 0.01f;
  if(xSz>2.0f) xSz=1.0f; // wrap

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

Wrapping-around color is about the same. To make it more interesting I'll have it go down (remember color numbers are from 0 to 1.) For fun I'll keep red at 1, so this goes from yellow (green and red) to just red:

```
public float g=1.0f; // start full green

void Update() {
  g -= 0.01f; // go down to 0
```

```
  if(g<0) g=1.0f; // wrap 0 back to 1

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

Notice it checks for *less than* 0, since it's going down.

8.1 bouncing

The other fun thing we can do when we hit the edge is bounce. This will flip the speed when we hit the right edge:

```
public float x=-7.0f;
public float xSpd=0.1f; // <- speed is now a variable, so we can change it

void Update() {
  x += xSpd;
  if(x>7) xSpd = -0.1f; // start moving left

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

Again, pretty cool for just one `if`. It looks even cooler if you have a wall and tweak the edge so it looks like it's bouncing off it.

An improved version will bounces off both sides, and keep whatever speed it started with (by using `xSpd*=-1` to flip the speed):

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

void Update() {
  x += xSpd;

  if(x>7) { xSpd *= -1; }
  if(x<-7) { xSpd *= -1; }

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

You should be able to Play and see `xSpd` flip between +0.1 and -0.1 (or whatever number you set it to) as it moves back and forth.

There's one cool bug: if the cube starts completely out-of-bounds, like x=12, it will just wiggle in place – `xSpd` gets flipped every Update. Fixing stuff like that is part of the fun of programming

Size change with bouncing is the same, except the numbers are smaller (1 to 2.) This version grows as a square by using the number for x and y:

```
public float sz=1.0f, spd=0.005f;

void Update() {
  sz+=spd;
  if(sz>2 || sz<1) spd*=-1; // flip speed when passes min or max
  transform.localScale = new Vector3(sz, sz, 1);
}
```

For fun, we can rewrite our movement bouncing using a different plan. We'll make a new variable `dir`, will be always be +1 or -1. Then our speed will be how fast in that direction:

```
public float x=-7.0f;
public float xSpd=0.1f; // always positive
public int dir=+1; // +1 is right, -1 is left

void Update() {
  if(dir==1) { // we're going right:
    x += xSpd; // add the speed
    if(x>7) dir=-1;
  }
  else { // we're going left:
    x -= xSpd; // subtracting the speed
    if(x<-7) dir=+1;
  }

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

Notice how I snuck in an example of an `if` inside of an `if`.

8.1.1 Speed increase

In older movement examples, I had the speed get faster and faster without limit. Now we can use an `if` to add a limit. Or to reset the speed, or whatever else can can think of.

I'll go back to the simple wrap-around Cube. This starts slow and speeds up. The new part is a maximum speed, using a simple `if`:

```
// wrap-around, increasing speed, to a maximum:
public float x=-7.0f, xSpd=0; // speed _starts_ at 0

void Update() {
```

```
  x = x + xSpd;
  if(x>7) x=-7; // wrap-around

  xSpd += 0.001f; // very slow speed increase
  if(xSpd>0.8f) xSpd = 0.8f; // speed limit

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

We could just as easily have it slow down to a minimum. We'd start `xSpd` high, subtract a tiny bit, and use a "can't get below this" `if`:

```
  xSpd += -0.001f; // a little slower
  if(xSpd<0.05f) xSpd = 0.05f; // lowest speed limit
```

You might remember the trick where I started the speed negative and always increased it. The Cube went backwards, slowed and then reversed direction. We can use the same trick to make gravity: always pull the y-speed a little down and bounce off the bottom:

```
public float y=3.0f; // nearer the top, giving it room to fall
public float ySpd=0; // any speed is fine, but 0=not moving seems obvious

void Update() {
  y+=ySpd;
  ySpd -= 0.003f; // gravity

  // bounce off bottom:
  if(y<-3) {
    ySpd*=-1;
    y=-3;  // just in case, don't let it fall below the floor
  }

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

This trick always fools me. When you're going up, `ySpd-=0.003f;` slows you down. When you're going down, the same line makes you go faster. That's how gravity really works, but it seems weird. Than the bounce is the same bounce from before.

We can also move left and right, bouncing off the sides. It's the same lines as before, and makes the cube bounce around in realistic arcs:

```
public float x=2.0f, y=3.0f; // set these to anywhere interesting
public float xSpd=0.1f; // always goes this fast, left or right
public float ySpd=0.1f; // a little bit up, for fun
```

```
void Update() {
  x += xSpd; y+=ySpd;
  ySpd -= 0.003f; // gravity

  // bounce off left, right and bottom:
  if(x>7 || x<-7) xSpd*=-1;
  if(y<-3) { ySpd*=-1; y=-3; } // bounce off bottom

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

Moving onto color, we can modify the yellow-to-red to go faster and faster, then reset. This looks funny since the speed is a negative number which gets "faster" when we subtract from it:

```
public float g=1.0f, gSpd=-0.01f; // green

void Update() {
  g += gSpd; // going down to 0/red
  if(g<0) g=1.0f; // wrap around back to 1/yellow

  gSpd-=0.0005f; // a little faster
  if(gSpd<-0.03f) gSpd=-0.01; // reset to starting speed when too fast

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

That last `if` is a standard wrap-around, but it feels funny to wrap around the speed.

8.2 Counting

We can do some interesting things if we count how many laps the cube makes. Making a lap counter is easy. Make an int variable, and add 1 on each reset:

```
// partial code for a lap counter:
public int laps=0;

void Update() {
  x += xSpd;
  if(x>7) {
    x=-7;
    laps=laps+1;
    // do special lap stuff here
  }
```

If we watch in the Inspector, `laps` will go up each time, but nothing else interesting will happen.

The next part is using the counter to do something. I'd like it to turn red after 3 laps, and green after 7 laps, then back to white on lap 8. All of that goes in the "do special lap stuff here" area:

```
// change color after laps 3, 7 and 8:
public float x=-7.0f, xSpd=0.1f;
public int laps=0;

void Update() {
  x += xSpd;

  if(x>7) {
    x=-7;
    laps += 1;

    // color change checks:
    if(laps==3)
      GetComponent<Renderer>().material.color = new Color(1,0,0); // red
    if(laps==7)
      GetComponent<Renderer>().material.color = new Color(0,1,0); // green
    if(laps==8)
      GetComponent<Renderer>().material.color = new Color(1,1,1); // white
  }

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

This is probably the largest body of an `if` we've had yet. But I think it seems natural. When you get past the end you should: 1) wrap around, 2) count the lap, and 3) use `if`s to check for the three special laps.

The color-changing `if`'s are a little new, but not really. We've used the line to change color before, just never guarded by an `if`.

A possibly confusing thing about this is what color is the 4th lap? The code doesn't say. Of course it's red since we turn it red on 3 and stays that way until we change it again. Another confusion is I said it resets after the 8th lap, but the code simply turns it white. That's because there's no "reset" command. Since it started out as white, turning it white feels like a reset.

A little tricker use of a lap counter, we can do something every 3rd lap, using modulo. `laps%3` (the remainder after dividing by 3,) will be 1,2,0,1,2,0 . . . after each lap. So `(laps%3==0)` is true every 3rd lap.

This sets the speed to "slow", every third lap:

```
  if(x>7) {
    x=-7;
    laps++;

    // set speed for next lap:
    if(laps%3==0) xSpd=0.03f; // slow
    else xSpd=0.1f; // otherwise normal speed
  }
```

This has a cool off-by-one bug. We set the speed for the *next* lap. When the the third lap *ends* we set it slow for the 4th lap. Ooops. Then the 7th, 10th and every 3rd after that is slow.

Another interesting thing: this always sets the speed, even when we don't need to. If the lap before us was fast and we should be fast, the current speed is fine. But checking for that is a pain. The way I wrote it is simpler and works just as well. "Changing" the speed to the same value it was before won't cause any harm.

8.3 Delay counter

If we want to wait for a second or two, we can do that with an `int` counter, an `if` and a plan.

The simplest plan is having the counter be how many Updates we should skip, count down, and zero means we're done waiting. If we want to wait for 60 Updates (about a second,) we set the counter to 60.

This moves a Cube in big steps, with an 80 update delay in-between each move. It should hop from left to right:

```
public float x=-7.0f;

public int delayCounter=0; // how many Updates to skip

void Update() {

  if(delayCounter>0) { // still waiting:
    delayCounter -= 1;
  }
  else {
    x = x + 1;
    transform.position = new Vector3(x, 0, 0);

    delayCounter=80; // wait for one more second
  }
}
```

The movement code is now guarded by an `if`. That's what `if`'s do – they say that we *might* do something. An `if-else` is good for this: either we're waiting on the countdown, or we're not. You could watch `delayCounter` spin down to 0 (the true part of the `if` is running,) then see the Cube pop forward and the counter reset (the `else` ran one time.)

We don't have to restart the delay immediately. It's possible to normally have no delay, `delayCounter` is almost always 0, and we set it only sometimes. This version moves with wrap-around, with a delay at the start of each round:

```
public float x=-7.0f;

public int delayCounter=0; // how many Updates to skip

void Update() {
  if(delayCounter>0) { // still waiting:
    delayCounter--; // subtract 1 shortcut
  }
  else {
    x += 0.1f; // standard small move
    if(x>7) { // wrap and signal for a pause:
      x=-7;
      delayCounter=80;
    }
    transform.position = new Vector3(x, 0, 0);
  }
}
```

Stepping back a little, I like to think about how `delayCounter=80;` is just assigning some variable. It's only the delay command because we wrote the `if`'s to make it work that way. We didn't need to look for a special delay command – we made it ourselves.

I also like to think about how we know `x=80;` says to put the Cube way off the right edge. But's it's the same assignment statement as setting the delay.

It might look funny to have the delay part come first. In the intro to `if`'s I wrote about flipping the true/false. I would look like this:

```
  if(delayCounter<=0) { // not waiting
    x+=0.1f;
    if(x>7) {
      x=-7;
      delayCounter=80;
    }
    transform.position = new Vector3(x, 0, 0);
  } // end of movement
```

```
  else delayCounter--;
}
```

Even if you don't think this looks nicer, the trick where we check "not waiting" first by flipping `(>0)` to `(<=0)` can come in handy.

This next example uses a delay with a size change. The basic code makes the Cube be size 1, 2, 3, 4 then back to 1. A delay counter makes it do that once every second. We should see it gradually pop larger:

```
public float sz=1.0f;
public int delay=0;

void Update() {
  if(delay>0) delay--;
  else {
    // sz goes 1,2,3,4 then back to 1:
    sz+=1;
    if(sz>4) { sz=1; delay=120; }
    else delay=60;

    transform.localScale = new Vector3(sz,sz,sz); // all-around larger
  }
}
```

Notice how I got tricky with setting the delay. When it wraps around to 1 I give a longer 120-tick delay, otherwise a smaller 60 ticks.

For fun, we could write that 60/120 a different way: always give a delay of 60, with an extra 60 for a wrap-around:

```
  if(delay>0) delay--;
  else {
    sz+=1;
    delay=60; // base delay
    if(sz>4) { sz=1; delay+=60; } // extra delay for wrapping

    transform.localScale = new Vector3(sz,sz,sz); // all-around larger
  }
}
```

What I like about this is the best way depends on how you think about it: two different delay times; or a base delay with a bonus for wrapping.

Finally, I mentioned counting down was the simplest plan for a delay. Another plan would be counting up to the total. We'd need an extra variable. To wait for 20 Updates we'd set the total to 20 and the counter to 0. Here it is with the movement wrap-around:

```
  int x=-7;

  int delayCount=0; // during a delay this counts up to the total, from 0
  int delayTotal=0; // no delay right now

  void Update() {
    if(delayCount<delayTotal) // delayCount goes up, delayTotal stays the same
      delayCount++;
    else {
      x+=0.1f;
      if(x>7) {
        x=-7;
        delayTotal=80; // how long to wait
        delayCount=0; // reset the count
    }
    transform.position = new Vector3(x, 0, 0);
  }
}
```

This is more complicated, which usually means worse. One possible advantage is if you want to show the delay as a percent – like one of those bars that fills in. We can't compute that with a single count-down, since we didn't save where it started.

8.4 State variables

In the second version of bouncing, with `dir`, my idea was the program has two stages – moving right and moving left. `dir`'s job was to remember which stage we were on. I used -1 and +1 as easy-to-remember values, but it could have used 1=left and 2=right.

Sometimes we have a program with more complicated stages, and we formalize the idea of a stage variable. Here's the whole trick:

Draw out a picture with your stages and the rules for going to the next. Number them – the exact numbers don't matter; the numbers are like their names.

To program it, make an `int` to remember which stage you're on. Inside, use a big "do only one of these" cascading if-else-if-else. Make one part for each stage. Whenever you want to go to a stage, like stage 3, write `stage=3`

Here's an example. The comments under `stage` explain what it does:

```
public int stage=0;
// stage 0 = move right fast until we get to x=6
// stage 1 = grow until we get to size 2
// stage 2 = slowly move up to y=4
```

```
// stage 3+ = done, do nothing

float x=-7, y=0; // position, for stage 0 and 2
float sz=1.0f; // Cube size, for stage 1

void Update() {

  if(stage==0) { // move right until past 6
    x += 0.1f;
    if(x>=6) stage=1; // done moving right
   }
  else if(stage==1) { // grow until size 2
    sz += 0.02f;
    if(sz>=2.0f) stage=2; // done growing
  }
  else if(stage==2) { // move up until past 4
    y += 0.03f;
    if(y>=4) stage=3; // done going up. Stage 3 is nothing
  }
  else if(stage==3) {} // stage 3 is do nothing

  // simpler to do these each time, even if some stages don't need them:
  transform.position = new Vector3(x, y, 0);
  transform.localScale = new Vector3(1, sz, 1);
}
```

Again, the trick is it only does one part each Update. For stage 0 it only runs those two lines: move a little right, past 6 means go to the next stage. Even better, when you finally set `stage` to 1, the *next* frame does the first grow – no double-changes.

It might seem funny having a number for the stage. Using a string looks nicer at first, but is really worse, but still makes a nice example:

```
public string stage="move right";
// will be "move right" "grow" "move up" or "done"

...
  if(stage=="move right") {
    x+=0.1f;  if(x>=6) stage="grow";
   }
  else if(stage=="grow") {
    sz+=0.02f;  if(sz>=2.0f) stage="move up";
  }
  else if(stage=="move up") {
    y+=0.03f;  if(y>=4) stage="done";
  }
```

The main problem is it's so easy to misspell a word, and there's no dropdown to help. If you change stage to `go right` by mistake, this whole thing just stops.

Back to regular state-machine examples, this one changes color in stages, using a delay counter so it's not a blur. Instead of stopping at the end, it goes back to stage 0:

```
int colStage=0;
// 0 = red for a long time
// 1 = bright blue quickly
// 2 = yellow for a medium time, then back to 0

int delay=100; // wait a little before starting, for no reason

void Update() {
  // whole thing is in a basic delay if-else:
  if(delay>0) delayCount--; // delayed: count down and wait
  else {
    // state machine ifs:
    if(colStage==0) {
      // red
      GetComponent<Renderer>().material.color = new Color(1,0,0);
      delay=150;
      colStage=1;
    }
    else if(colStage==1) {
      // bright blue:
      GetComponent<Renderer>().material.color = new Color(0.5f,0.5f,1);
      delay=30;
      colStage=2;
    }
    else if(colStage==2) {
      // yellow
      GetComponent<Renderer>().material.color = new Color(1,1,0);
      delay=70;
      colStage=0; // back to red
    }
  }
}
```

This spends almost all of it's time subtracting from `delay`. Each stage runs for only one update: it changes color, sets the new delay, and goes to the next stage (for when the delay is finally over.)

The important thing is how we're still using our extra made-up `stage` variable to remember which step we're on.

Not really important for these examples, but the official name for this sort of thinking is a `State Machine`. You can find state diagrams (circles with arrows,) or GUI's (Unity's built-in Mechanim animation system is a drag&drop state machine.)