Chapter 26

Pointer examples

This chapter is more Unity-type things that show off various pointer and class tricks.

26.1 Text change

A label on the screen is just another thing you can create: `GameObject->UI->Text` makes one. It makes something called Canvas, which we can ignore, with the Text inside of it.

If you select the Game tab, on the big window, you should see a small, ugly "New Text" in the lower-left corner. That's good enough for testing, but here are some non-programming Text notes:

- Color is near the bottom (in the Text's Inspector, in the area labelled Text.) It starts as an ugly dark grey. As usual, clicking it brings up the Color Picker. Change it to pure black, or all white, or green ....
- Notice how the top now says RectTransform and has PosX and PosY. Positioning Text uses special coordinates. `(0,0)` is centered, and the screen is a few hundred across. If you want it centered on the top, try `(0,200)`. As usual, you can slide-drag the numbers (put the mouse slightly above the box.)
- The contents is the big box labelled `text`. Just so you know, it will let you enter multiple lines (they won't fit, at first.)
- To make the text larger you can slide drag where it says FontSize (in the middle of the Character sub-heading, under Text.) But it will wink out if it gets too big. Here's how to really do it:

  Go to Scene view and double-click either Canvas or your Text. This should snap your view there. Use scroll-zoom and right-button-spin or the upper-right gizmo to get a mostly head-on view. You should see a big, white

border – that's the entire "Text screen." If you click on your Text object, you should see a faint white border around it.

If the white borders are hard to see, you could turn off the fancy sky. In Scene, on the top bar of that window, there's a little mountains icon. Select the drop-down next to it and uncheck SkyBox. That should give you a nice grey background.

Your text can't be any larger than the white border around it. You can grow that using the new Width and Height values (in the Inspector, in RectTransform.) It uses the same numbers as position – about 800 makes it across the entire screen. For Height, it's fine to increase it to have plenty of room.

Again, we can run our code with the crummy, small greyish text in the corner. Making it look nicer is just for fun.

Adding text labels wouldn't have been helpful before, since we didn't know how to find them. But now we do – using a `GameObject` pointer. All we need to know is the magic command to change the `text` part. It's `GetComponent<UnityEngine.UI.Text>().text = "cow";`. The part after the `=` is just a string.

Here's the left-to-right laps program, using an on-screen lap-counter:

```
public GameObject lapText; // drag the Text object into this
int laps = 0;

Vector3 pos;

void Start() {
  pos = new Vector3(-7,0,0);
  lapText.GetComponent<UnityEngine.UI.Text>().text="laps: 0";
}

void Update() {
  pos.x+=0.1f;
  if(pos.x>7) {
    pos.x=-7;
    laps++;
    lapText.GetComponent<UnityEngine.UI.Text>().text="laps: "+laps;
  }
}
```

It's not much different from our old program – we're just slapping `laps` into the label using a pointer and the new rule for using `Text`.

26.2 More time, and APIs

Now that we know about `GetComponent<Text>().text="cow";`, we can probably change some more things. `Text` is just a class, which means it probably has more fields.

For example, there's a color picker in the Inspector under Text. If you look, you'll see dot-color works for `Text`. This turns it orange:

```
public GameObject someLabel; // drag text into here

void Start() {
  Color col = new Color(1, 0.5f, 0); // it so happens this makes orange
  someLabel.GetComponent<UnityEngine.UI.Text>().color = col;
}
```

The really neat thing about this is it's just old rules. We know classes like `Text` have fields, like `color`. And once we see `color` is just a `Color` struct we know how to use that.

Here's one more: if the Text has extra room, we can click the pictures for Left/Center/Right and watch it move (they're the first thing in the Paragraph Inspector section.) We can look that up in the code – there's an `alignment` field which says it's a `TextAnchor`. Hmmm ...

`TextAnchor` is just an enumerated type (we can look it up, or just type it and see it says `enum`.) It has values like UpperLeft and LowerCenter, which pop-up when we type TextAnchor-dot.

Here's some code using it:

```
public GameObject someLabel; // drag text into here

void Start() {
  someLabel.GetComponent<UnityEngine.UI.Text>().alignment = TextAnchor.MiddleRight;

  // in two steps:
  TextAnchor ach = TextAnchor.UpperRight;
  someLabel.GetComponent<UnityEngine.UI.Text>().alignment = ach;
}
```

The second version looks funny, declaring `TextAnchor ach;`, but it's no different than pre-computing color using `col`. If we have a class or enumerated type ... or any other type, we can use it to declare a variable.

Ways to change built-ins through code are usually called the API – Application Programming Interface. We say something is *exposed* in the API if a program can find and change it. Text color and alignment are exposed. It's not automatic, and not everything has to be – there are a few sliders that the program has no way to change – but they try to expose as much as they think

you'll need.

I want to use color and alignment for something interesting, but first I want to show the official Unity timing trick, to really make something that delays for 1 second – not just counting out 60 ticks and hoping.

The trick is to read from the built-in clock. Most systems have one. Unity's counts in seconds, starting from when you press Play. Each time you press Play, it resets to 0, and it's always in just seconds. If you press Play and wait a minute and 10 seconds, it will say 70.

To read it, just look in the built-in global `Time.time` (you might remember this from the namespace section. It's the `time` variable in the `Time` namespace.) Here's a test program showing it change in the Inspector:

```
public float theTime; // copy of time, which we can watch

void Update() {
  theTime = Time.time;

  // this would also work, but would scroll like crazy and be hard to read:
  // print( Time.time );
}
```

The trick is that, sure, `Time.time` is just a variable, but Unity is always secretly increasing it, so it really is the time.

Here's the plan for using `Time.time` to make a delay: suppose dinner is ready is 20 minutes, and it's 6:10 now. That means dinner is ready at 6:30. That's the only number we have to remember. We can walk around, check the clock occasionally, and at 6:30 we can open the oven.

In computer terms, suppose you want a 3-second delay. Look up `Time.time` now and add 3. Save that in a global. For example, if we want to wait for 3 seconds at time 15, we save 18. Then, in Update we keep checking if the time is past 18.

Here's a very simple program which prints every 3 seconds:

```
// If this is 20, we're waiting to print until time 20:
public float nextPrintTime=0;

void Update() {
  if(Time.time>=nextPrintTime) {
    print( Random.Range(0,999) ); // cheap way to print different things
    nextPrintTime = Time.time+3.0f;
  }
}
```

The last line is resetting the time. In the old version, it was `delayCounter=60;`. Now, it's like saying "OK, I just printed at time 15, the next one is 3 seconds from now, at 18."

Here's a slightly silly program using everything at once. It counts by 1's, every 1.5 seconds, and moves the text back and forth, with a color change as it gets higher:

```
public GameObject theText; // drag Text here

float nextTime; // Time.time delay trick
public float delaySecs = 1.5f; // here so we can change it
int num=0;

void Update() {
  if(Time.time>nextTime) {
    nextTime = Time.time + delaySecs;
    num++;

    // show the number:
    theText.GetComponent<UnityEngine.UI.Text>().text=""+num;

    // color it as it grows:
    Color cc;
    if(num<=3) cc = new Color(1,1,1);
    else if(num<=8) cc = new Color(1,1,0);
    else cc=new Color new Color(1,0,0);
    theText.GetComponent<UnityEngine.UI.Text>().color = cc;

    // move left/right:
    TextAlign ta;
    if(num%2==0) ta = TextAlign.MiddleLeft;
    else ta = TextAlign.MiddleRight;
    theText.GetComponent<UnityEngine.UI.Text>().alignment = ta;
  }
}
```

The thing I like about this example is on one hand, it's just a cascading if and setting a few values. The most complicated thing about it is `if(n%2==0)` to check whether `n` is even. But on the other hand it's full of gibberish like UnityEngine.UI and TextAlign.

A lot of real code is like this. What's it's doing is pretty simple, but there happen to be a lot of pre-defined classes and types making it look complicated.

26.3 Three platforms

I'd like to use one script to move three platforms back-and-forth in customizable ways. Obviously, this is going to use the trick with GameObject pointers, plus we can make some fun custom classes and use a function.

I'll assume we have a front -7 to 7 view. We'll pre-make the three platforms. Just cubes will work, but we could make them nicer: make a Cube, hand-set scale to long and flat, about`(1, 0.2, 1)`, then make 2 copies. May as well name them A, B and C.

So they don't overlap, we might stack them at about `(0,2,0)`, `(0,0,0)` and `(0,-2,0)`. Our old code ignored where the Cube started – this improved code will start Cubes moving from where we placed them. The exact spots won't matter (I put my platform A on top.)

Each platform will move left and right, but will have it's own settable left/right sides, speed and a pause at each edge. For example, we want to be able to set platform A to slowly move between -7 and 3, but set platform B to move quickly from -5 and 0, pausing for 1.5 seconds before reversing.

Each platform needs the same group of variables, so this is a good place to make a class:

```
// "blueprint" type data about platforms
// including extra stuff to make it show in Inspector:
[System.Serializable]
public class PlatformStats {
  public float minX=-7, maxX=7; // left/right of how it moves
  public float moveSpd=0.05f;
  public float edgeDelaySecs=0.5f; // pause at edge
  public GameObject g; // pointer to actual platform Cube
}
```

I'm sneaking in a small `C#` rule here. A class allows you to set starting values for variables. `minX=-7` isn't a real assignment statement. But when you `new` the class, you get that value instead of 0. You can't use this trick with structs, just because.

The point of this class is we can easily create variables for all three platforms by declaring 3 of them. These are `public`, to put them in the Inspector. GUI-magic will let us pop each open and closed:

```
public PlatformStats PA, PB, PC;
```

A new thing is how our real link to the platform – `public GameObject g;` – is inside the class. We've never put a `GameObject` as a field in a class before, but there's no reason we can't. It's good here so we get one link for each platform

– we'd drag our pre-made platform-Cubes into `PA.g`, `PB.g` and `PC.g`.

We need the usual extra variables to run the movement: a `Vector3` for the current position, one to remember if we're moving left or right, and one to time the hit-an-edge delays. We don't need these to be in the Inspector, but still need one set per platform, so it seems fine to group them into another class:

```
// running data about platforms
class PltfrmActiveData {
  public Vector3 pos; // current position
  public float waitUntilTime=-99; // for end-pause delay
  public int mvDir=+1; // +1 = moving right; -1= moving left
}
```

Then we make one set for each platform:

```
PltfrmActiveData padA, padB, padC; // not in Inspector
```

The Inspector variables are `new`'d and filled already, but we'll have to create and set these in Start:

```
void Start () {
  // standard required new, for each:
  padA = new PltfrmActiveData();
  padB = new PltfrmActiveData();
  padC = new PltfrmActiveData();

  // start them all moving right, just because:
  padA.mvDir = padB.mvDir = padC.mvDir = +1;

  // copy starting positions from the real linked platforms:
  padA.pos = PA.g.transform.position;
  padB.pos = PB.g.transform.position;
  padC.pos = PC.g.transform.position;
}
```

The last three lines are a mouthful, but it's a good trick. `PA.g` is a link to the real platform A, which means `PA.g.transform.position` is where we hand-placed platform A before the game started. `padA.pos` is our standard variable controlling how it moves.

So those lines say to start each platform's position at the actual spot we hand-placed them. A nice bonus is our code doesn't have to decide how high up each platform is – we did that when we arranged them in Unity.

Since we have to run the same code for all three platforms, a function seems like a good idea. It needs all the data for that platform, which is now just two variables. This looks messy, but it's the same old move code, with a possible edge-delay added:

```
void movePlatform(PlatformStats P, PltfrmActiveData pa) {
  // are we still paused on an edge?
  if(Time.time<pa.waitUntilTime) return;

  pa.pos.x += P.moveSpd*pa.mvDir; // move at our speed in our direction

  if(pa.pos.x<P.minX) { // off left edge?
    pa.pos.x=P.minX;
    pa.mvDir=+1;
    if(P.edgeDlySecs>0) pa.waitUntilTime = Time.time+P.edgeDlySecs;
  }
  else if(pa.pos.x>P.maxX) { // off right edge?
    pa.pos.x=P.maxX;
    pa.mvDir=-1;
    if(P.edgeDlySecs>0) pa.waitUntilTime = Time.time+P.edgeDlySecs;
  }
  // position the actual platform:
  P.g.transform.position=pa.pos;
}
```

Notice how this counts on `pa` being a pointer. It only reads from `P`, but changes the inside of `pa` – moving the position and such.

Update merely needs to call that function for each platform:

```
void Update() {
  movePlatform(PA, padA);
  movePlatform(PB, padB);
  movePlatform(PC, padC);
}
```

That's it.

It's part coding tricks (because of the classes and function, a fourth platform is easy to add,) and part GUI tricks (we can easily adjust the edges and speeds, even while it's running.)

I kept it simplish, but we could grow it to have a separate delay at each edge, an option to wrap or bounce, a different speed each way .... But even with this you can get interesting platform motion.

26.4 More prefab use

The fun thing about prefabs is they don't feel like simple copies – using one feels like making something pop into existence. It's extra fun to do that with "physics" objects.

For this we need a falling ball from before: create a Sphere, add a Rigidbody component, possibly color it, drag it into Project to make a prefab of it, delete the original.

This script randomly fires them from the left, in a little arc. For fun, I'll shoot a few, wait, shoot a few more, in 5 waves. I'm using the new timer trick to wait:

```
public GameObject ballPF; // dragged ball with rigidbody prefab

int waveNum=0;
float nextWave=-99; // next wave fires after this time

void Update() {
  // stops after firing 5 waves:
  if(waveNum>=5) return;

  if(Time.time<nextWave) return; // delay between each wave
  // make 2 to 5 sec delay for the next wave:
  nextWave=Time.time + Random.Range(2.0f, 5.0f);

  // make 3-5 at random heights, random speed, random size:
  Vector3 bPos; bPos.x=-7; bPos.z=0; // always on left side
  Vector3 bSpd; bSpd.z=0;
  int count = Random.Range(3,5+1); // fire 3-5 balls
  for(int i=0;i<count;i++) { // standard "this many times" loop
    GameObject bb = Instantiate(ballPF);

    bPos.y=Random.Range(-3.0f, 1.0f); // not too high
    bb.transform.position = bPos;

    float bSz = Random.Range(0.3f, 0.7f); // random size
    bb.transform.localScale = new Vector3(bSz, bSz, bSz);

    bSpd.x=Random.Range(5.0f, 12.0f); // arcing speed
    bSpd.y=Random.Range(3.0f, 8.0f); // to the right
    bb.GetComponent<Rigidbody>().velocity = bSpd;
  }

  waveNum++;
}
```

Nothing new here, but a lot at once. Some notes about the design:

- It might have looked a little nicer if the body of the `for` loop was moved into a function `fireOneBall();`. It wouldn't make the program smaller, but just breaking a long thing into parts is fine.

- Not being a function let me cheat a little with the variables. Notice how before the loop I pre-made position and speed, pre-setting the two parts that will be the same for all balls.

Instead of blasting out the balls in each wave all at once, a wave could quickly spit 4 balls one-at-a-time, wait longer, and repeat (you can really see the difference).

There's nothing new here, just being clever with variables and `if`'s:

```
int ballsThisWave=0;

void Update() {
  if(waveNum>5) return;
  if(Time.time<nextTime) return;
  ballsThisWave++;
  if(ballsThisWave<4) nextTime = Time.time+0.1f; // short delay (same wave)
  else { // this wave is done, reset and wait for next:
    nextTime = Time.time+Random.Range(2.0f, 5.0f);
    waveNum++;
    ballsThisWave=0; // reset
  }
  // same shoot one ball code goes here:
}
```

`ballsThisWave` is like the loop counter, sort of, except it goes up by 1 every 0.1 seconds.

An interesting-looking thing is creating objects at our position. This moves us back and forth along the bottom of the screen, and lets the space bar fire balls from us:

```
public GameObject ballPF; // prefab for the ball
Vector3 pos; // where we are

void Start() {
  pos = new Vector3(0,-3,0); // bottom center
}

void Update() {
  if(Input.GetKey("a")) pos.x +=-0.2f;
  if(Input.GetKey("s")) pos.x +=+0.2f;
  pos.x = Math.Clamp(pos.x, -7, 7);
  transform.position = pos;

  if(Input.GeyKeyDown(KeyCode.Space)) {
    GameObject bb = Instantiate(ballPF);
```

```
    Vector3 ballPos = pos; // <- start new ball where we are now
    ballPos.y+=1.0; // a little above us
    bb.transform.position = ballPos;

    // shoot straight up, random speed for no reason:
    Vector3 ballSpd = new Vector3(0, Random.Range(8.0f, 10.0f) ,0);
    bb.GetComponent<Rigidbody>().velocity = ballSpd;
  }
}
```

The lines moving us aren't the most efficient – if we aren't pressing a key it checks the edges and "moves" us by 0. But they're short and easy to read.

26.5 Multiple scripts

Way, way back I explained the typical game-making plan where each object that needs to think should have it's own script. It would be fun to have the balls we shoot shrink and then wink out of existence. We can do that with a second script, on the balls.

This script would go on the prefab ball. Each time we make a ball, it gets a fresh copy of this script on it, starting right then. The way it fits in is tricky, but the script is simple by itself:

```
float sz; // current size (x, y, z are always the same)
float shrinkStart; // what time to start shrinking

void Start() {
  // copy our real size as the starting size:
  sz = transform.localScale.x; // x, y and z are all the same
  // wait a few seconds before we shrink:
  shrinkStart = Time.time + Random.Range(3.0f, 6.0f);
}

void Update() {
  if(Time.time<shrinkStart) return;
  sz-=0.01f;
  transform.localScale = new Vector3(sz,sz,sz);
  if(sz<=0)
    Destroy(gameObject); // <- new command. Delete ourself
}
```

`Destroy` isn't a `C#` command. It's like the opposite of an Instantiate – it removes you from the Scene. You can use it to destroy things besides yourself, if you have a pointer there.

If you've shot 4 balls, there will be a copy of this on each ball, with different variables. The Unity system runs all 5 Updates each frame.