Chapter 25

Pointers in Unity

A basic use of pointers is to aim at someone else's stuff and play with it. Unity has a bunch of Cubes. We should be able to aim pointers at them.

A main reason for using a fancy program like Unity is how easily it allows us to do that. There will be an empty slot, and we'll drag in the Cube we want. Inside our program, we'll have a properly aimed pointer. In Unity, that slot will be in the Inspector.

Once we can get pointers to several Cubes, we can use our old tricks to move them around.

25.0.1 Pointers to existing Cubes

The things in a Unity scene are `GameObject`'s, configured to be Lights, Cameras, and Cubes. GameObject is a regular class, which means we can declare `GameObject g1;`. It will be a pointer.

The class has a field named `transform`. It's the transform we've been using for ourself all along. If `g1` is a GameObject, we can move it with the familiar-looking `g1.transform.position = new Vector3(-7,4,0);`.

Put together, this script, on any object, will let us reach out and move some other Cube:

```
public GameObject g; // drag in a Cube

void Start() {
  g.transform.position = new Vector3(0,4,0); // just some spot
  g.GetComponent<Renderer>().material.color = new Color(1,1,0); // yellow
}
```

As usual, `g` appears in the Inspector. It's currently `null` but it says `GameObject(none)` to be a little nicer. We're suppose to find a Cube, in the panel with names, and drag it in. When our program runs, `g` will point to that Cube.

That's almost too easy. It's the reason programs like Unity exist. They let us cheat by setting up pointers without needing to write code.

Before using them more, let's check that `GameObject` is mostly a normal class. We're allowed to use `new` with it:

```
void Start() {
  GameObject g = new GameObject();
  g.transform.name="fire engine";
}
```

When we run, an empty object named fire engine appears in the panel. That's a slight cheat – Unity tracks GameObjects. It won't track our own classes, Dogs, for example.

Back to our regular examples, this one uses two pointers. You'd drag a different thing onto each:

```
public GameObject g1, g2; // drag in 2 different cubes

void Start() {
  // reach through each to change colors:
  g1.GetComponent<Renderer>().material.color = new Color(0, 1, 0); // green
  g2.GetComponent<Renderer>().material.color = new Color(0, 0, 1); // blue
}
```

Nothing special, but it's new. We weren't able to change 2 different things before, only ourself.

We can test this a little by dragging the same Cube into both slots. It's legal for 2 pointers to go to the same place. It will turn blue. `g1` turns it green, then `g2` immediately turns it blue.

We could leave `g1` null. Click the little circle on the far right of `g1` and select None from the dropdown. We'll get a standard null reference exception. `g2`'s Cube won't change – the error cut off that line.

A fun example which moves two Cubes at different speeds:

```
public GameObject c1, c2; // drag in two Cubes
int x1=-7, x2=-7;

void Update() {
  x1+=0.1f; if(x1>7) x1=-7; // move&wrap cube 1
  x2+=0.07f; if(x2>7) x2=-7; // move&wrap cube 2

  c1.transform.position = new Vector3(x1, 2, 0); // place both cubes
  c2.transform.position = new Vector3(x2, -1, 0); // using different y's
}
```

This is nothing new, except it's neat to have one script run a Cube race.

We can use the trick where a 3rd pointer chooses one of them, like the `activePet` example from the last chapter:

```
public GameObject c1, c2; // drag in two Cubes
GameObject g; // choose between c1 and c2

void Start() { g=c1; } // using 1st Cube, for now

void Update() {
  if(Input.GetKeyDown("a")) {
    if(g==c1) g=c2; // flip g between c1 and c2
    else g=c1;
  }

  // move whichever g points at:
  Vector3 v=g.transform.position;
  v.x+=0.1f;
  if(v.x>7) v.x=-7;
  g.transform.position=v;
}
```

An alternate way of getting a pointer is the `Find` command. It searches through all gameObjects in the panel, by name, and returns a pointer. This mini-program finds `"cube1"` and `"cube2"` and turns them different colors:

```
void Start() {
  GameObject c1, c2;
  c1=GameObject.Find("ball1"); // <- aim c1 at ball1
  c2=GameObject.Find("ball2"); // <- aim c2 at ball2

  c1.GetComponent<Renderer>().material.color=Color.red;
  c2.GetComponent<Renderer>().material.color=Color.blue;
}
```

Notice how `Find` uses the namespace trick. It's name is `Find` and it's in the `GameObject` namespace.

The moral here is that once you have a pointer, it doesn't matter how you got it. We can use `c1` and `c2` in the usual way.

Finally, let's use `null` some more. This next code hopes `c1` was preset. If not, it guesses some names. It uses the standard null-check:

```
public GameObject c1; // drag something in?
```

```
void Start() {
  // try to fill c1:
  if(c1==null) // they forgot to drag something in
    c1=GameObject.Find("clover");
  if(c1==null) // we couldn't find "clover"
     c1=GameObject.Find("rabbitFoot");

  // use c1 if it goes anywhere:
  if(c1!=null)
    c1.GetComponent<Renderer>().material.color = new Color(0,1,0);
  // if it's still null, we're just giving up on it
}
```

As you might guess, `Find` returns `null` is there isn't anything with that name. `null` is really the perfect not-found value.

25.1 Instantiate

Unity has a type of clone function named `Instantiate`. It takes a pointer to one gameObject, makes an identical new one, and returns a pointer.

25.1.1 Instantiate as a copy

Here's some simple code making 10 random copies of a Cube. Warning, warning, warning: do not use this to copy yourself. The copy will run its copy of the script, copying itself, and so on forever. `c1` should point to something without this script on it:

```
public GameObject c1; // drag in a Cube to copy

void Start() {
  for(int i=0; i<10; i++) { // basic 10-times loop
    GameObject bb = Instantiate(c1);
    // bb is now aimed at the freshly made copy of c1

    Vector3 pos; pos.z=0;
    pos.y=Random.Range(-5.0f, 5.0f);
    pos.x=Random.Range(-7.0f, 7.0f);

    bb.transform.position = pos;
  }
}
```

In a normal program we'd need to save those 10 pointers somewhere, or else we're simply making time-wasting garbage. Here we're taking advantage of how

Unity tracks all gameObjects.

This next one is mostly the same, except it makes them 1-at-a-time, when we press space. Then, to use one extra pointer, the previous one also turns red:

```
public GameObject ball1; // drag in some other item
// the most recently created ball:
GameObject previousBall;

void Update() {
  if(Input.GetKeyDown(KeyCode.Space) {
    // change the most recent ball's color:
    if(previousBall!=null)
      previousBall.GetComponent<Renderer>().material.color=Color.red;

    GameObject newBall = Instantiate(ball1);
    // move to random spot on screen:
    float x=Random.Range(-7.0f, 7.0f);
    float y=Random.Range(-4.0f, 4.0f);
    newBall.transform.position = new Vector3(x, y, 0);

    previousBall=newBall; // update to the new most recent ball
  }
}
```

Notice how `ball1` never changes. Its job is to remember the "master ball" for the Instantiate copy command. Meanwhile `previousBall` is a typical moving pointer, temporarily remembering one ball.

Instantiate is also overloaded. We're allowed to give it a position where we want the new item:

```
public GameObject ball1;

void Update() {
  if(Input.GetKeyDown(KeyCode.Space) {
    Vector3 newBallPos; // pre-make target position
    newBallPos.x=Random.Range(-7.0f, 7.0f);
    newBallPos.y=Random.Range(-4.0f, 4.0f);
    newBallPos.z=0;

    // clone ball1 and set the position:
    GameObject bb = Instantiate(ball1, newBallPos);
  }
}
```

That's a typical time-saving overload. All it does extra is run the `position=` line which we ran before. But 1 line saved for something we do often is a useful shortcut.

25.1.2 Proper Unity use of Instantiate

This entire section has nothing to do with programming. But if you use Instantiate with Unity for real it's probably good to know.

Suppose we want lots of explosions using Instantiate. We need a master explosion, but it should be hidden – off the edge of the world or something – and also frozen somehow. That's a pain, so Unity provides a nice way.

Take any Cube and drag it down into the Project window. Unity copies it there, with a corner-cube icon. It counts as a Cube, but it's not visible anywhere and its scripts won't run. It's the perfect master object for copying. Unity calls that a *prefab*.

Any of the scripts above work with these. Find a prefab (cube icon which you made in the Project panel) and drag it into the `gameObject` slot. Your script is now making new objects out of thin air.

The whole process looks like this: suppose you know how to make explosions. Make one (using a particleSystem), then drag it into Project to make a prefab from it. Delete the original. Anyone with `public GameObject blast;` can now get a link to the master explosion, for copying.

As long as we're here, we may as well see the other use of Unity prefabs. Once you have a prefab, you can quickly make real copies of it by dragging into the scene. More than that, the copies are linked. Changing the prefab (the original, in the Project panel) changes all copies.

But don't let that confuse you about how programs work. In a program, copies are copies – there's no link to the original.

25.1.3 Pointers for "caching"

You may have noticed `GetComponent<Renderer>().material.color` is a 3-stage lookup. It calls a function to get our renderer, then the material, then finally the color. Material is a class, which means we can get a pointer to one. We can create a shortcut pointer to our own material:

```
Material myMat; // will be a short-cut to our Material

void Start() {
  // create the shortcut, using a partial look-up:
  myMat = GetComponent<Renderer>().material;
}
```

```
void Update() {
  // 1% chance to turn red or blue:
  if(Random.Range(0,100)==1) myMat.color = Color.red;
  if(Random.Range(0,100)==1) myMat.color = Color.blue;
}
```

The old stuff is still there. Start has `GetComponent<Renderer>().material`, which it saves, and Update has dot-color. `Material myMat` is a pointer, since Material is a class.

Not super useful, but it's pretty neat that we can do it. We can't get a shortcut to color, since Color is a struct. You can never have a pointer to a struct.

We can do the same trick with not-us. Suppose we have a link to the floor, and often want to change the floor's color. We can save the floor's material as a shortcut:

```
public GameObject floor; // drag the floor into this
Material floorMat; // saved shortcut for changing floor color

void Start() {
  // set up floor color shortcut:
  floorMat = floor.GetComponent<Renderer>().material;
}

void Update() {
  // increase the floor's red and wrap it around:
  Color cc = floorMat.color; // using the shortcut
  cc.r+=0.02f; if(cc.r>1) cc.r=0;
  floorMat.color=cc; // using the shortcut
}
```

I think `floor.GetComponent<Renderer>()` is pretty neat.

25.2 New'ing Inspector variables

You may have noticed that our classes in the Inspector don't need `new`'s. Unity does it automatically. But we're allowed to use `new` if we want:

```
// recall we need this line to make it display:
[System.Serializable]
public class Dog {
  public string name;
  public int age;
}
```

```
public Dog dg; // now dg is in the Inspector

void Start() {
  dg.age=7; // legal, it's been new'd magically
}

void Update() {
  if(Input.GetKeyDown("a"))
    dg=new Dog(); // new blank Dog
}
```

Pressing the A key does the usual thing: it creates a fresh Dog, abandoning the old one. The Inspector looks like it's only erasing the age, but it's really tracking that fresh Dog.

Put another way, do anything you want with classes visible in the Inspector. It won't restrict anything – it merely tracks and displays the current one.