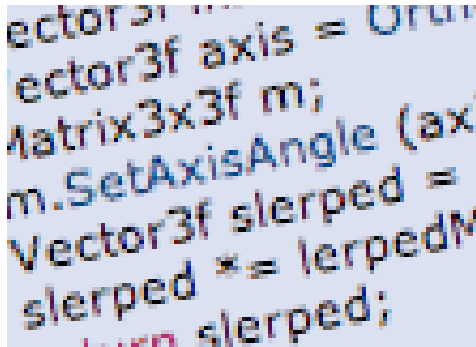


Introduction to scripting with Unity



Scripting is an essential part of Unity as it defines the behaviour of your game. This tutorial will introduce the fundamentals of scripting using Javascript. No prior knowledge of Javascript or Unity is required.

Time to complete: 2 hours.

Author: Graham McAllister

Contents

1. Aims of this tutorial
2. Prerequisites
3. Naming conventions
4. Player Input
5. Connecting variables
6. Accessing components
7. Instantiate
8. Debugging
9. Common Script Types

1. Aims of this tutorial

Scripting is how the user defines a game's behaviour (or rules) in Unity. The recommended programming language for Unity is **Javascript**, however C# or Boo can also be used. This tutorial will cover the fundamentals of scripting in Unity and also introduce key elements of the Application Programming Interface (API). You can think of the API as code that has already been written for you which lets you concentrate on your game design and also speeds up development time.

A good understanding of these basic principles is essential in order to harness the full power of Unity.

2. Prerequisites

This tutorial focuses on the scripting element of Unity, it is assumed you are already familiar with Unity's interface (if not you should read the Unity GUI tutorial).

In order to make scripting easier to understand, it is preferable to have a code editor that has syntax highlighting support for Javascript. This means that reserved words (syntax used by Javascript itself) are coloured differently than user defined words. One such editor is [SubEthaEdit](#).

NB: any text that requires the user to take an **action** begins with a '- '.

3. Naming Conventions

Before we begin, it is worth mentioning some conventions in Unity.

Variables - begin with a lowercase letter. Variables are used to store information about any aspects of a game's state.

Functions - begin with an uppercase letter. Functions are blocks of code which are written once and can then be reused as often as needed.

Classes - begin with an uppercase letter. These can be thought of as collections of functions.

Tip: When reading example code or the Unity API, pay close attention to the first letter of words. This will help you better understand the relationship between objects.

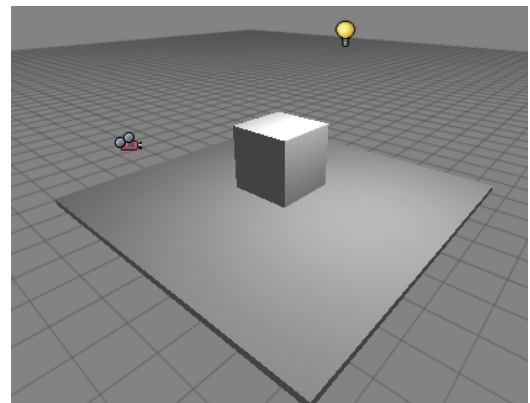
4. Player Input

For our first program we're going to allow the user to move around in a simple game world.

Setting the scene

- Start Unity.

Firstly, let's create a surface for the user to walk on. The surface we're going to use is a flattened cube shape.



- Create a cube and scale its x,y,z dimensions to 5, 0.1, 5 respectively, it should now resemble a large flat plane. Rename this object 'Plane' in the Hierarchy View.
- Create a 2nd cube and place it at the centre of this plane. If you can't see the objects in your Game View, alter the main camera so they're visible. Rename the object to Cube1.
- You should also create a point light and place it above the cubes so that they're more easily visible.
- **Save** the scene by selecting File->Save As and give the game a name.

Our first script

We're now ready to start game programming. We're going to allow the player to move around the game world by controlling the position of the main camera. To do this we're going to write a script which will read input from the keyboard, then we attach (associate) the script with the main camera (more on that in the next section).

- Begin by creating an empty script. Select Assets->Create->Javascript and rename this script to Move1 in the Project Panel.
- Double-click on the Move1 script and it will open with the **Update()** function already inserted (this is default behaviour), we're going to insert our code inside this function. Any code you insert inside the Update() function will be executed every **frame**.

In order to move a game object in Unity we need to alter the position property of its [transform](#), the **Translate** function belonging to the transform will let us do this. The Translate function takes 3 parameters, x, y and z movement. As we want to control the main camera game object with the cursor keys, we simply attach code to determine if the cursor keys are being pressed for the respective parameters:

```
function Update () {  
    transform.Translate(Input.GetAxis("Horizontal"), 0, Input.GetAxis("Vertical"));  
}
```

The Input.GetAxis() function returns a value between -1 and 1, e.g. on the horizontal axis, the left cursor key maps to -1, the right cursor key maps to 1.

Notice the 0 parameter for the y-axis as we're not interested in moving the camera upwards. The Horizontal and Vertical axis are pre-defined in the Input Settings, the names and keys mapped to them can be easily changed in Edit->Project Settings->Input.

- Open the Move1 Javascript and type in the above code, pay close attention to capital letters.

Attaching the script

Now that our first script is written, how do we tell Unity which game object should have this behaviour? All we have to do is to attach the script to the game object which we want to exhibit this behaviour.

- To do this, first click on the game object that you wish to have the behaviour as defined in the script. In our case, this is the Main Camera, and you can select it from either the Hierarchy View or the Scene View.
- Next select Components->Scripts->Move1 from the main menu. This attaches the script to the camera, you should notice that the Move1 component now appears in the Inspector View for the main camera.

Tip: You can also assign a script to an game object by **dragging** the script from the Project View onto the object in the Scene View.

- Run the game (press the play icon at the lower left hand corner), you should be able to move the main camera with the cursor keys or W,S,A,D.

You probably noticed that the camera moved a little too fast, let's look at a better way to control the camera speed.

Delta time

As the previous code was inside the Update() function, the camera was moving at a velocity measured in meters per **frame**. It is better however to ensure that your game objects move at the more predictable rate of meters per **second**. To achieve this we multiply the value returned from the Input.GetAxis() function by [Time.deltaTime](#) and also by the velocity we want to move per second:

```

var speed = 5.0;

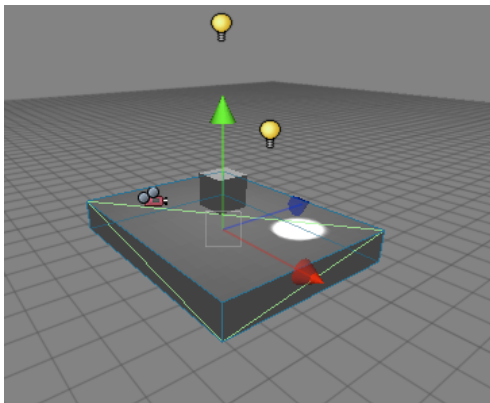
function Update () {
    var x = Input.GetAxis("Horizontal") * Time.deltaTime * speed;
    var z = Input.GetAxis("Vertical") * Time.deltaTime * speed;
    transform.Translate(x, 0, z);
}

```

- Update the Move1 script with the above code.

Notice here that the variable speed is declared outside of the function Update(), this is called an **exposed variable**, as this variable will appear in the Inspector View for whatever game object the script is attached to (the variable gets exposed to the Unity GUI). Exposing variables are useful when the value needs to be tweaked to get the desired effect, this is much easier than changing code.

5. Connecting Variables



Connecting variables via the GUI is a very powerful feature of Unity. It allows variables which would normally be assigned in code to be done via **drag and drop** in the Unity GUI. This allows for quick and easy prototyping of ideas. As connecting variables is done via the Unity GUI, we know we always need to expose a variable in our script code so that we can assign the parameter in the Inspector View.

We'll demonstrate the connecting variables concept by creating a spotlight which will follow the player (Main Camera) around as they move.

- Add a **spotlight** to the Scene View. Move it if necessary so it's close to the other game objects.
- Create a new Javascript and rename it to Follow.

Let's think what we want to do. We want our new spotlight to look at wherever the main camera is. As it happens, there's a built in function in Unity to do this, [transform.LookAt\(\)](#). If you were beginning to think 'how do I do this?' and were already imagining a lot of code, then it's worth remembering to always check the Unity **API** for a function that already exists. We could also make a good guess at looking in the 'transform' section of the API as we're interested in altering the position or rotation of a game object.

Now we come to the connecting variables section; what do we use as a parameter for LookAt()? Well we could hardcode a game object, however we know we want to assign the variable via the GUI, so we'll just use an exposed variable (of type Transform). Our Follow.js script should look like this:

```

var target : Transform;

function Update () {
    transform.LookAt(target);
}

```

-
- Attach the script to the spotlight and notice when the component gets added, the “target” variable is exposed.
 - With the spotlight still selected, drag the Main Camera from the Hierarchy View onto the “target” variable in the Inspector View. This **assigns** the target variable, i.e. the spotlight will now follow the Main Camera. If we wanted the spotlight to follow a different game object we could just drag in a different object (as long as it was of type Transform of course).
 - Play the game. If you watch the Scene View you should see the spotlight following the Main Camera around. You may want to change the position of the spotlight to improve the effect.
-

6. Accessing Components

As a game object can have multiple scripts (or other components) attached, it is often necessary to access other component’s functions or variables. Unity allows this via the [GetComponent\(\)](#) function.

We’re now going to add another script to our spotlight which will make it look at Cube1 whenever the jump button (spacebar by default) is pressed.

Let’s think about this first, what do we want to do:

1. Detect when the jump button has been pressed.
2. When jump has been pressed make the spotlight look at Cube1. How do we do this? Well, the Follow script contains a variable “target” whose value determines which game object the spotlight should look at. We need to set a new value for this parameter. We could hardcode the value for the cube (see the section ‘Doing it with code’ later), however we know that exposing the variable and assigning this via the GUI is a better way of doing this.

- Create a new Javascript and name it Switch. Add the following code to Switch.js:

```
var switchToTarget : Transform;

function Update () {
    if (Input.GetButtonDown("Jump"))
        GetComponent(Follow).target = switchToTarget;
}
```

Notice in particular how Follow is the parameter to GetComponent(), this returns a reference to the Follow script which we can then use to access its “target” variable.

- Add the Switch script to the spotlight and assign Cube1 to the switchToTarget parameter in the Inspector View.
- Run the game. Move around and verify that the spotlight follows you as usual, then hit the spacebar and the spotlight should focus on the Cube1.

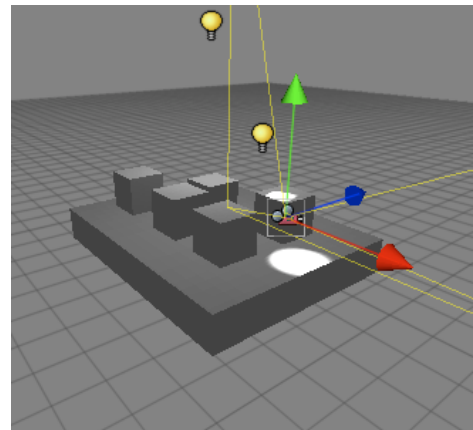
Doing it with code

Earlier in the tutorial we mentioned that it would be possible to assign the variables via code (as opposed to the Unity GUI), let’s take a look at how you would do that.

Remember this is only for comparison, assigning variables via the GUI is the recommended approach.

The problem we were interested in earlier was how do we tell the spotlight to look at Cube1 when the jump button was pressed. Our solution was to expose a variable in the Switch script which we could then assign by dropping Cube1 onto it from the Unity GUI. There are two main ways to do this in code:

1. Use the **name** of the game object.
2. Use the **tag** of the game object.



1. Game object name

A game object's name can be seen in the Hierarchy View. To use this name with code we use it as a parameter in the `GameObject.Find()` function. So if we want the jump button to switch the spotlight from Main Camera to Cube1, the code is as follows:

```
function Update () {
    if (Input.GetButtonDown("Jump"))
    {
        var newTarget = GameObject.Find("Cube").transform;
        GetComponent(Follow).target = newTarget;
    }
}
```

Notice how no variable is exposed as we name it directly in code. Check the API for more options using `Find()`.

2. Game object tag

A game object's **tag** is a string which can be used to identify a component. To see the built-in tags click on the Tag button in the Inspector View, notice you can also create your own. The function for finding a component with a specific tag is `GameObject.FindWithTag()` and takes a string as a parameter. Our complete code to do this is:

```
function Update () {
    if (Input.GetButtonDown("Jump"))
    {
        var newTarget = GameObject.FindWithTag("Cube").transform;
        GetComponent(Follow).target = newTarget;
    }
}
```

7. Instantiate

It is often desirable to **create** objects during run-time (as the game is being played). To do this, we use the [Instantiate](#) function.

Let's show how this works by instantiating (creating) a new game object every time the user presses the fire button (either the left mouse button or left ctrl on the keyboard by default).

So what do we want to do? We want the user to move around as usual, and when they hit the fire button, instantiate a new object. A few things to think about:

1. Which object do we instantiate?
2. Where do we instantiate it?

Regarding which object to instantiate, the best way of solving this is to expose a variable. This means we can state which object to instantiate by using drag and drop to assign a game object to this variable.

As for where to instantiate it, for now we'll just create the new game object wherever the user (Main Camera) is currently located whenever the fire button is pressed.

The Instantiate function takes three parameters; (1) the object we want to create, (2) the 3D position of the object and (3) the rotation of the object.

The complete code to do this is as follows (Create.js):

```
var newObject : Transform;

function Update () {
    if (Input.GetButtonDown("Fire1")) {
        Instantiate(newObject, transform.position, transform.rotation);
    }
}
```

Don't forget that transform.position and transform.rotation are the position and rotation of the transform that the script is attached to, in our case this will be the Main Camera.

However, when an object is instantiated, it is usual for that object to be a **prefab**. We'll now turn the Cube1 game object into a prefab.

- Firstly, let's create an empty prefab. Select Assets->Create->Prefab. Rename this prefab to Cube.
- Drag the Cube1 game object from the Hierarchy View onto the Cube prefab in the Project view. Notice the prefab **icon** changes.

Now we can create our Javascript code.

- Create a new Javascript and name it Create. Insert the above code.
- Attach this script to the Main Camera and assign the Cube prefab to the newObject variable of Main Camera.
- Play the game and move around as usual. Each time the fire button is clicked (LMB or left ctrl) and you should notice a new cube appearing.

8. Debugging

Debugging is the skill of finding and fixing human errors in your code (ok let's call them mistakes!). Unity provides help via the [Debug](#) class, we'll now look at the `Debug.Log()` function.

Log

The `Log()` function allows the user to send a message to the Unity Console. Reasons for doing this might include:

1. To prove that a certain part of the code is being reached during run-time.
2. To report the status of a variable.

We'll now use the `Log()` function to send a message to the Unity Console when the user clicks the fire button.

- Open the Create script and add the following line after the 'Instantiate' code inside the 'if' block:

```
Debug.Log("Cube created");
```

- Run the game and click the fire button, you should see a line appear at the bottom of the Unity GUI saying "Cube created", you can click on this to examine the Unity Console.

Watch

Another useful feature for debugging is exposing a private variable. This makes the variable visible in the Inspector View when the **Debug** mode is selected, but it cannot be edited.

To demonstrate this, we'll expose a private variable to count the number of cubes that we instantiate.

- Open the Create script again and add two lines:

- (1) Add a private variable called `cubeCount`
- (2) Increment this variable whenever a cube is instantiated.

The complete code is as follows (Create.js):

```
var newObject : Transform;
private var cubeCount = 0;

function Update () {

    if (Input.GetButtonDown("Fire1")) {
        Instantiate(newObject, transform.position, transform.rotation);
        Debug.Log("Cube created");
        cubeCount++;
    }

}
```


-
- Run the game and click the fire button to create some cubes. Notice in the Inspector View how the cubeCount variable is incremented whenever a new cube is instantiated. Notice also how the number appears greyed out, this denotes that it's a read-only variable (cannot be edited).
-

9. Common script types

Whenever a new Javascript is created, by default it contains an Update() function. This section will discuss other common options available, simply replace the name of the Update() function with one from the list below.

FixedUpdate()

Code placed inside this function is executed at regular intervals (a fixed framerate). It is common to use this function type when applying forces to a Rigidbody.

```
// Apply a upwards force to the rigid body every frame

function FixedUpdate () {
    rigidbody.AddForce (Vector3.up);
}
```

Awake()

Code inside here is called when the script is initialized.

Start()

This is called before any Update() function, but after Awake(). The difference between the Start () and Awake() functions is that the Start() function is only called if the script is enabled (if its checkbox is enabled in the Inspector View).

OnCollisionEnter()

Code inside here is executed when the game object the script belongs to collides with another game object.

OnMouseDown()

Code inside here is executed when the mouse hovers over a game object which contains a GUIElement or a Collider.

```
// Loads the level named "SomeLevel" as a response
// to the user clicking on the object

function OnMouseDown () {
    Application.LoadLevel ("SomeLevel");
}
```

OnMouseOver()

Code inside here is executed when the mouse hovers over a game object which contains a `GUIElement` or a `Collider`.

```
// Fades the red component of the material to zero
// while the mouse is over the mesh

function OnMouseOver () {
    renderer.material.color.r -= 0.1 * Time.deltaTime;
}
```

Check the Unity [API](#) for more information on all of these functions.

Summary

This tutorial has introduced the essential scripting concepts in Unity. You should now read other Unity tutorials or try experimenting yourself!