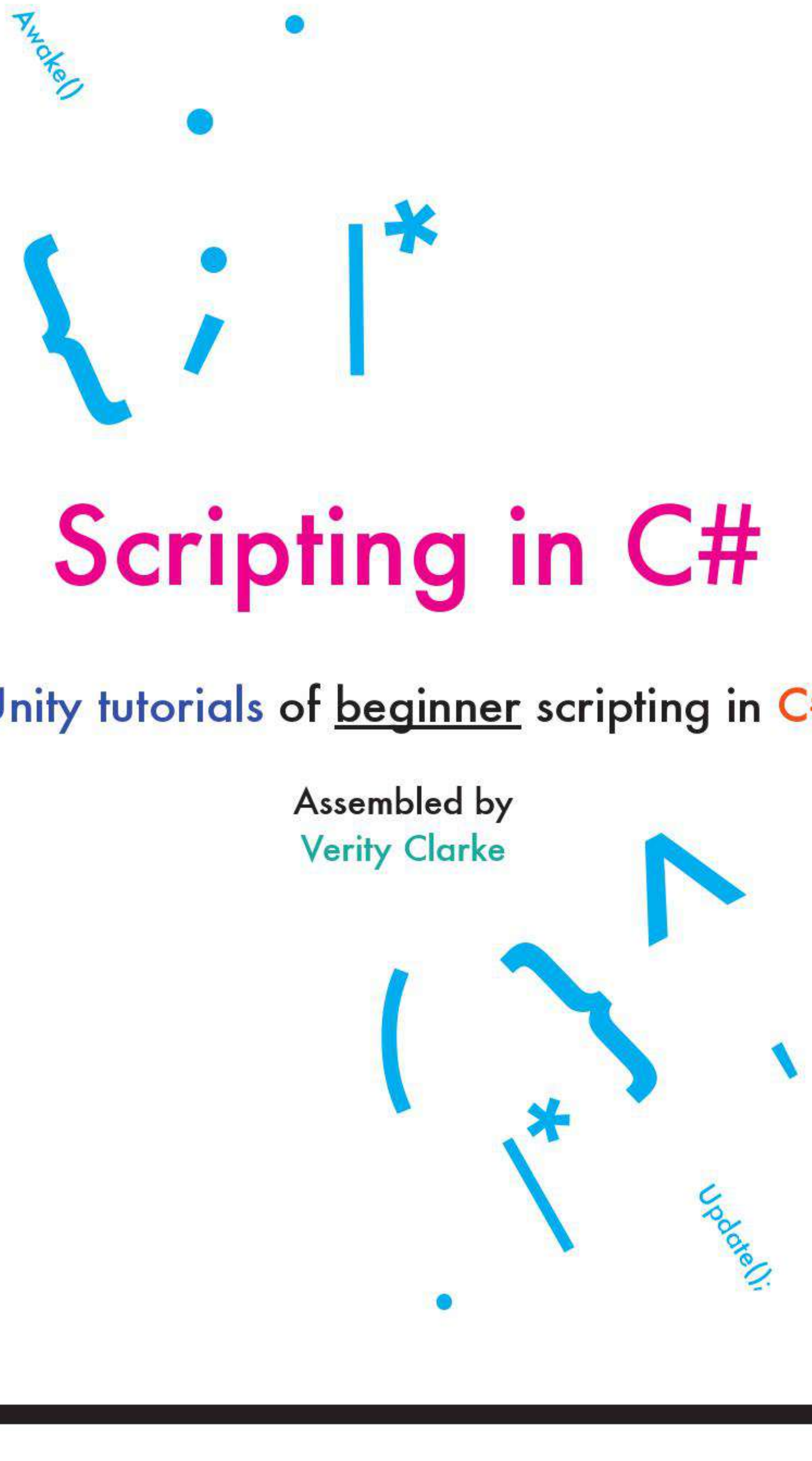


Scripting in C#

Unity tutorials of beginner scripting in C#

Assembled by

Verity Clarke

```
using UnityEngine;
using System.Collections;

public class LetterToSelf : MonoBehaviour
{
    public GameObject lazyUnAmbitiousMe;
    void Start ()
    {
        if(transform.position.y <= 5f)
        {
            Debug.Log ("Dear me in the future,

                                MAKE GAMES!
                                MAKE GAMES!
                                MAKE GAMES!

                                OK BYE (press space)");

          if(Input.GetKey(KeyCode.Space))
        {
            Destroy(lazyUnAmbitiousMe);
        }

    }
}
```

Originally this was a place for note taking, a place for revising to help me remember the C# scripting language. But as a graphic designer I can't help but organise the information and stylise it.

All of this information was found online; each chapter is simply a beginner Unity Scripting tutorial. I do not take any credit for the content of this book. All rights and reserves go to Unity and their helpful youtube videos.

You can find them at:

https://unity3d.com/learn/tutorials/modules/beginner/scripting/scripts-as-behaviour-components

You can find me at:

https://www.verityclarke.co.uk

verity_verity_verity@hotmail.com
June 2015

Chapters

https://unity3d.com/learn/tutorials/modules/beginner/scripting/scripts-as-behaviour-components

1

Scripts as Behaviour Components.

Scripts should be considered as behaviour components in unity. They can be applied to objects and can be seen in the inspector.

```
void Update ()
    {
        if(Input.GetKeyDown(KeyCode.R))
        {
            gameObject.renderer.material.color = Color.red;
        }
```

This script is saying when you press down the *'R'* key the game object will turn red.

```
void Update ()
    {
        if(Input.GetKeyDown(KeyCode.R))
        {
            gameObject.renderer.material.color = Color.red;
        }
```

Setting it to red which is part of the colour class.

Object the script is attatched too

Addressing the mesh renderer

Addressing the material attatched to the renderer

Colour of material

```
using UnityEngine;
using System.Collections;

public class ExampleBehaviourScript : MonoBehaviour
{
    void Update ()
    {
        if(Input.GetKeyDown(KeyCode.R))
        {
            gameObject.renderer.material.color = Color.red;
        }
        if(Input.GetKeyDown(KeyCode.G))
        {
            gameObject.renderer.material.color = Color.green;
        }
        if(Input.GetKeyDown(KeyCode.B))
        {
            gameObject.renderer.material.color = Color.blue;
        }
    }
}
```

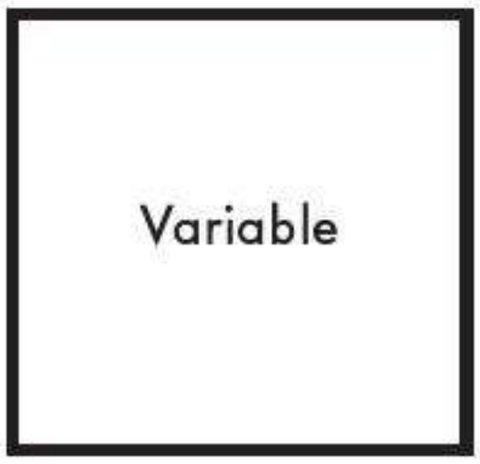
By pressing the R, G or B keys on the keyboard we can change the colour of the game object. (By looking above you can see the game object is a square if these are the results.)

https://unity3d.com/learn/tutorials/modules/beginner/scripting/variables-and-functions

2

Variables & Functions

Variables are like boxes that contain information. You need a different type of box for different types of information.

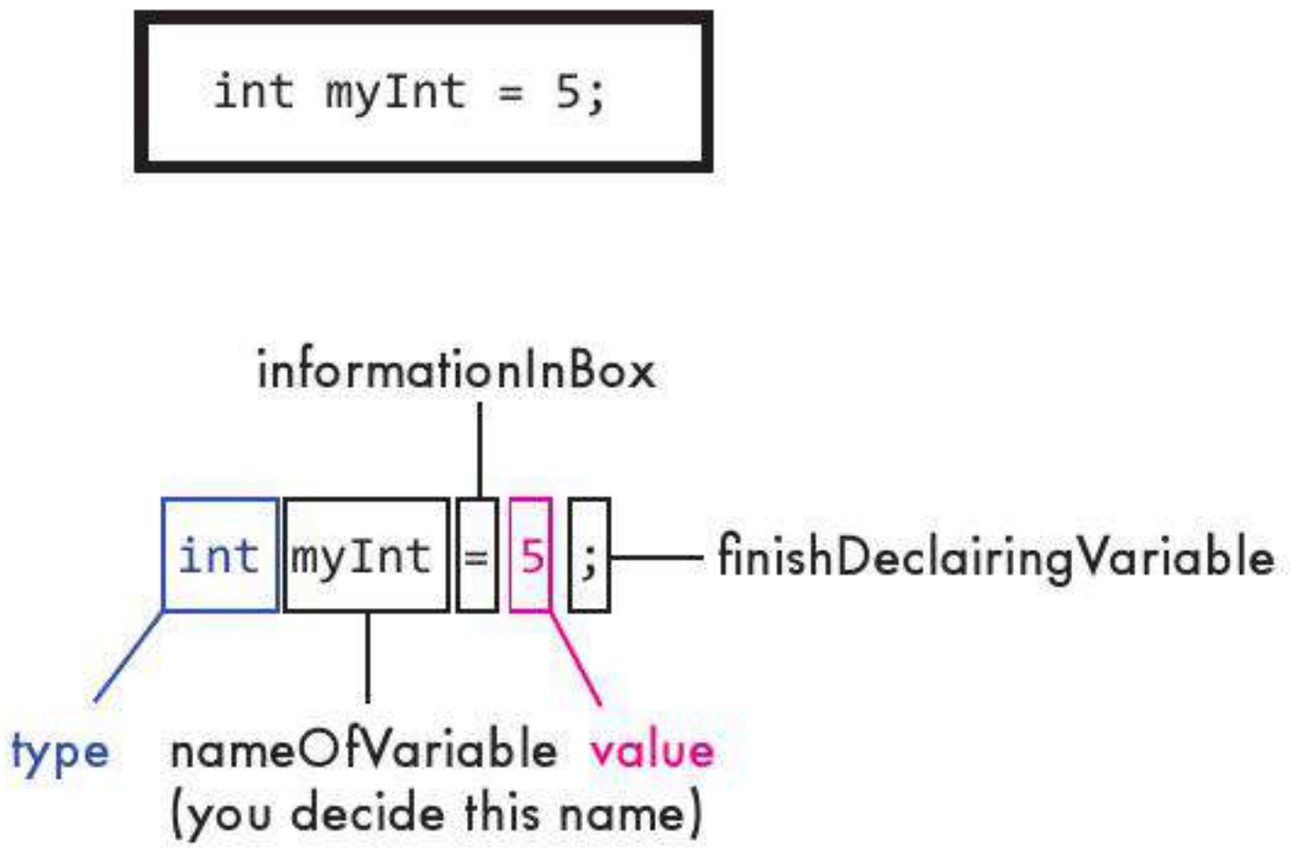


We start our variable definition by saying what sort of box we'd like.

int

Integer- meaning a whole number.

```
int myInt = 5;
```



```
public class VariablesAndFunctions : MonoBehaviour
{
    int myInt = 5;
```

To do something with this code we need to put it into a function.

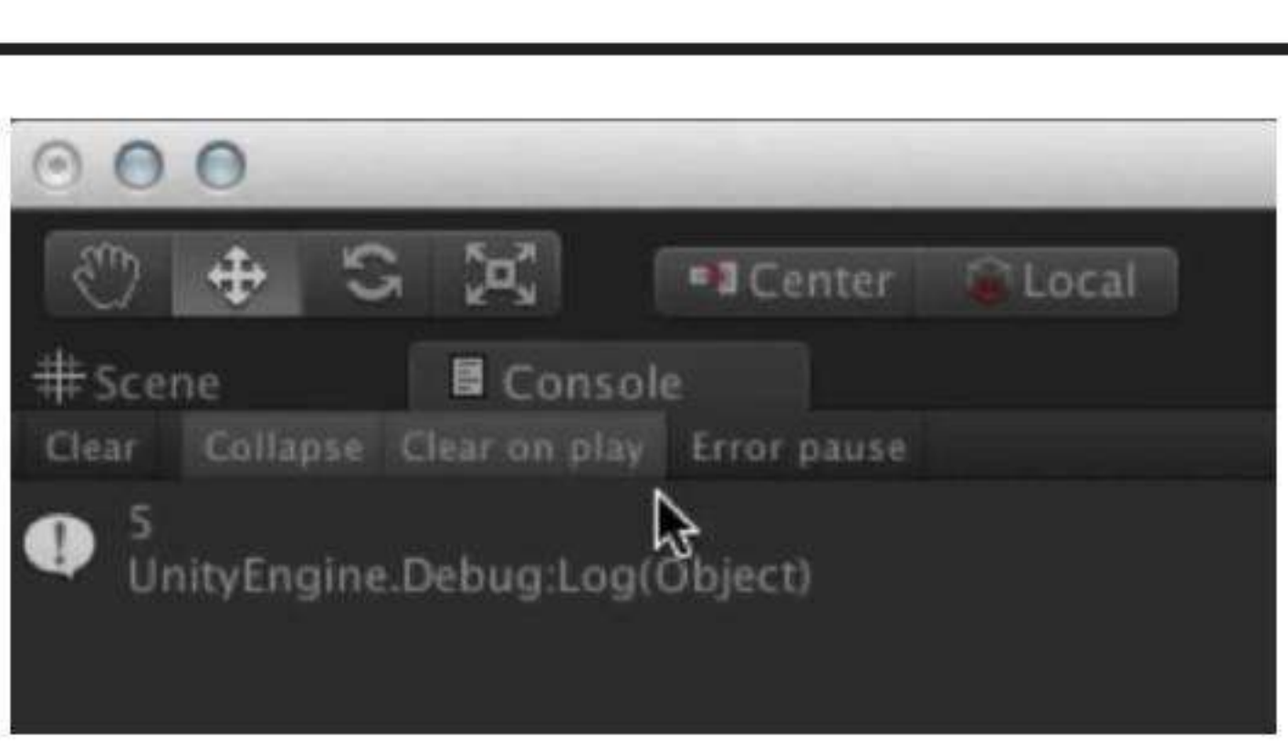
void Start ()

Start is called when the object this script is attached to enters a scene.

Debug.log ();

Get the value of any variable in your game.

```
public class VariablesAndFunctions : MonoBehaviour
{
    int myInt = 5;

    void Start ()
    {
        Debug.log(myInt)
    }
}
```



```
public class VariablesAndFunctions : MonoBehaviour
{
    int myInt = 5;

     void Start ()
     {
          Debug.log(myInt*2)
     }
}
```

10

Multiplying myInt (5) with another interger (2) will result in a value of 10.

```
public class VariablesAndFunctions : MonoBehaviour
{
    int myInt = 5;

     void Start ()
     {
          myInt=55;
          Debug.log(myInt*2)
     }
}
```

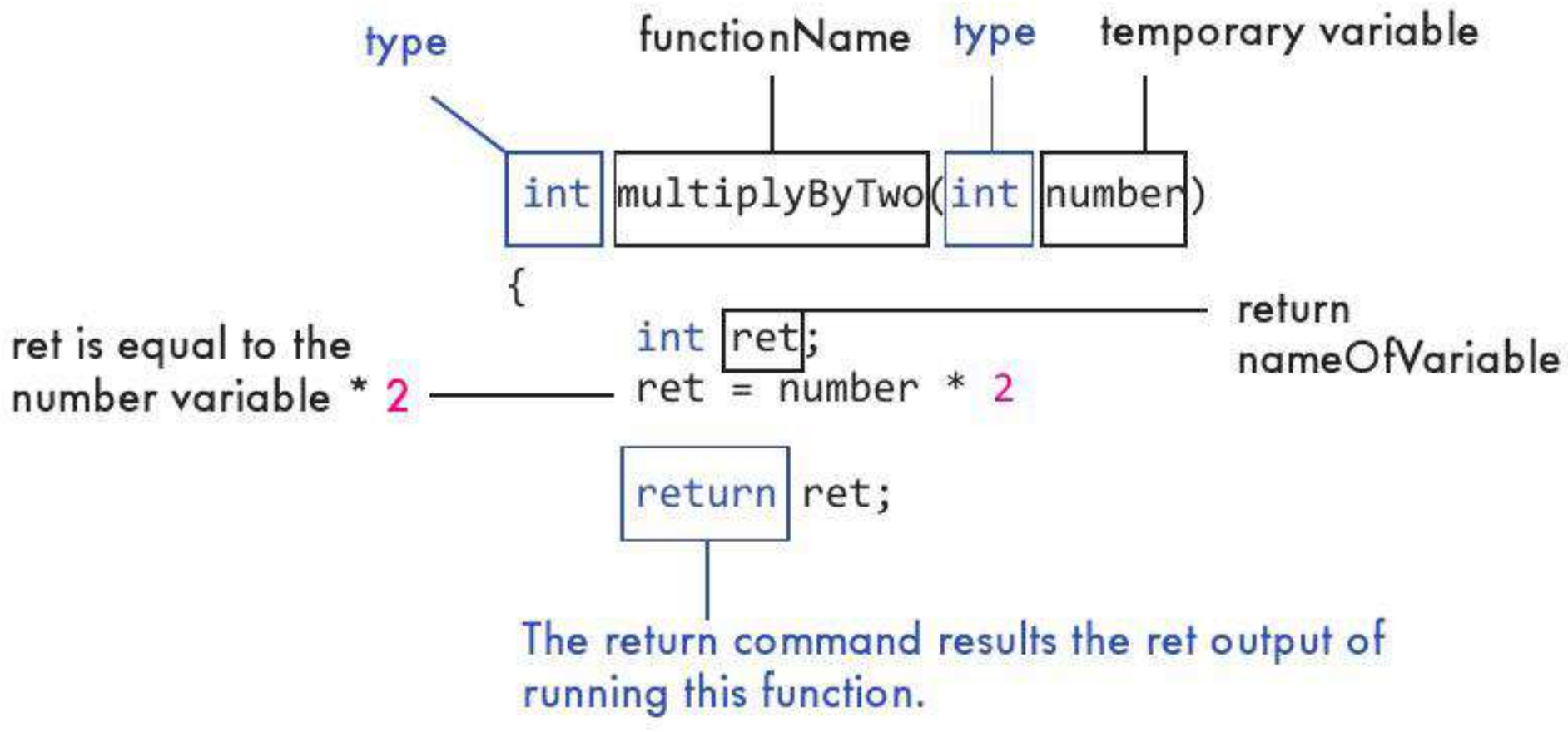
110

Although myInt (5) was initialised as a variable, the void Start () has re-assigned that to 55.

A function, also known as a method, will take the boxes we have storing information and give us boxes back, or 'return' as it's known.

void Start () is an example of a function that doesn't return anything. So it's type is void.

```
public class VariablesAndFunctions : MonoBehaviour
{
    int myInt = 5;

     void Start ()
     {
          multiplyByTwo(myInt);
          Debug.log();
     }

     int multiplyByTwo(int number)
          {
               int ret;
               ret = number * 2 ;

               return ret;
}
```



https://unity3d.com/learn/tutorials/modules/beginner/scripting/conventions-and-syntax

3

Conventions & Syntax

Coding syntax simply means the structure of the language, and some conventions are essential to learning to read and write code.

●

The dot operator is a full stop, or period, seen between words with in code. It works like writing the line of an address.

```
Debug.Log(transform.position.x);
```

Consider this line of code. Here we could consider `Debug` the country, and `Log` the city. And then within that `transform` the country, `position` the city and `x` the street we are trying to locate.

So the dot operator is effectively allowing you to separate or access elements of a compound item within Unity. A compound item: something that contains many elements.

For example transform contains position, rotation and scale. The dot operator is used to choose position, and the position contains X, Y or Z. So then we've chosen x by using the dot operator once again.

●

/

The next piece of syntax is the semi-colon. It's used to terminate statements, which is why you'll always see it at the end. How ever, not all parts of the code are statements.

For example the opening and closing of a class declaration. Or the opening and closing of functions or if statements. Anything using a curly brace does not need a semicolon at the end of it.

{

}

Indenting is not technically necessary but allows you to read it more easily, as it's used to show the functional structure of your code.

```
using UnityEngine;
using System.Collections;

public class BasicSyntax : MonoBehaviour
{
    void Start ()
    {
        Debug.Log(transform.position.x);

        if(transform.position.y <= 5f)
        {
            Debug.Log ("I'm about to hit the ground!");
        }
    }
}
```

Here you can visually trace a line going down the page of where your code blocks start and end.

Comments can be used to comment about a piece of code to leave yourself a note. Single line comments can be used by using a double forward slash.

```
// This is line is here to tell me X position of my object

Debug.Log(transform.position.x);
```

Multiple line comments can be used by using a forward slash asterisk, and then an asterisk forward slash to end the comment.

```
/* Hi there!
 * This is two lines of code
 *
 * Asterisks automatically appear with-in mono-develop
 * when you press return.
 *
 * The code also turns green to help you distinguish
 *between the differant elements of code.
 *
 */
```

Comments can be used to leave notes for other coders, as well as disable parts of code temporarily.

```
if(transform.position.y <= 5f)
        {
            Debug.Log ("I'm about to hit the ground!");
        }
```

```
/*
if(transform.position.y <= 5f)
        {
            Debug.Log ("I'm about to hit the ground!");
        }
*/
```

The compiler will ignore this if statement if it is commented out.

https://unity3d.com/learn/tutorials/modules/beginner/scripting/c-sharp-vs-javascript-syntax

4

C# vs JS Syntax

In Unity scripts are made of 1 or more classes. C# have the class declarations shown. In JavaScript however the declaration is hidden. Meaning that what you write in the script is automatically inside the class.

C#

```
using UnityEngine;
using System.Collections;

public class ExampleSyntax : MonoBehaviour
{

}
```

JS

```
#pragma strict
```

It is the equivalent of writing here in the C# script.

C#

```
int myInt = 5 ;
```

type nameOfVariable value

A variable in C# starts with the type of variable you are declaring. You then name the variable. You then have the option to give it a default value.

JS

```
var myInt : int = 5 ;
```

var nameOfVariable type value

In JavaScript a variable declaration starts by using the keyword var. We then name the variable. After that we have the option to say what type of variable it is that we are declaring by using the colon. It is good practice to decide on a variable type when declaring it. We can then optionally give it a default value.

JS

```
#pragma strict
```

The line #pragma strict at the top forces us to select a type.

C#

```
public class ExampleSyntax : MonoBehaviour
{
      int myInt= 5 ;

      int MyFunction (int number)
      {

      }
}
```

return FunctionName type parameterName

A function declaration in C# starts with the type of variable that will return from the function. Or the word void if we are not returning anything from the function. We then name the function. We then have a parameter list, also known as the arguments.

JS

```
#pragma strict

var myInt : int = 5;

function MyFunction (number : int) : int
{

}
```

function FunctionName parameterName type return

In JavaScript the function declaration starts with the key word function. We then need to name the function. Then comes the parameter list. Similarly to setting a variable type we use a colon. to set a return type for our function.

Another key differance between C# and Javascript is the default accsess modifiers. In C# the default is private. In Javascript the default accsess modifer is public.

https://unity3d.com/learn/tutorials/modules/beginner/scripting/if-statements

5

IF Statements

Often in coding you will need to make a decision based on a condition. Think of having a cup of tea, when it's first made the water is boiling. You might want to check if it's ready to drink. If the tea's temperature is greater than the hottest temperature you're willing to drink it at, then you won't drink it.

```
void TemperatureTest ()
    {
        // If the tea's temperature is greater than the
           hottest drinking temperature...
        if(coffeeTemperature > hotLimitTemperature)
        {
            // ... do this.
            print("Tea is too hot.");
        }
```

An extension of the IF statement is the IF-else statement. In out tea example the else statement might be that you drink the tea. This action will not be executed when the first condition is not satisfied. So if the tea is not too hot to drink, then we drink it.

```
void TemperatureTest ()
    {
        if(coffeeTemperature > hotLimitTemperature)
        {
            print("Tea is too hot.");
        }
         else
        {
            print("Coffee is just right.");
        }
```

Making the else statement a conditional, by using another IF, can further extend this. With our tea example if the tea was too hot we wouldn't drink it. But if we left it too long, and it had gotten cold then we might not drink it either. However if neither of these conditions are satisfied, that is, that the if neither the tea is too hot nor too cold then the tea is at an appropriate drinking temperature. You can use If statements in this way to make decision's within your code as to what it should do.

6

Loops

Loops in programming are a way to repeat lines of code. Looping, or repeating, is often referred too as iterating. There are different types of loops.

Often in Loop coding you will see angle brackets. These are symbols for operations usually pertain to work being done mathematically or with a programming language.

<

Greater than symbol

>

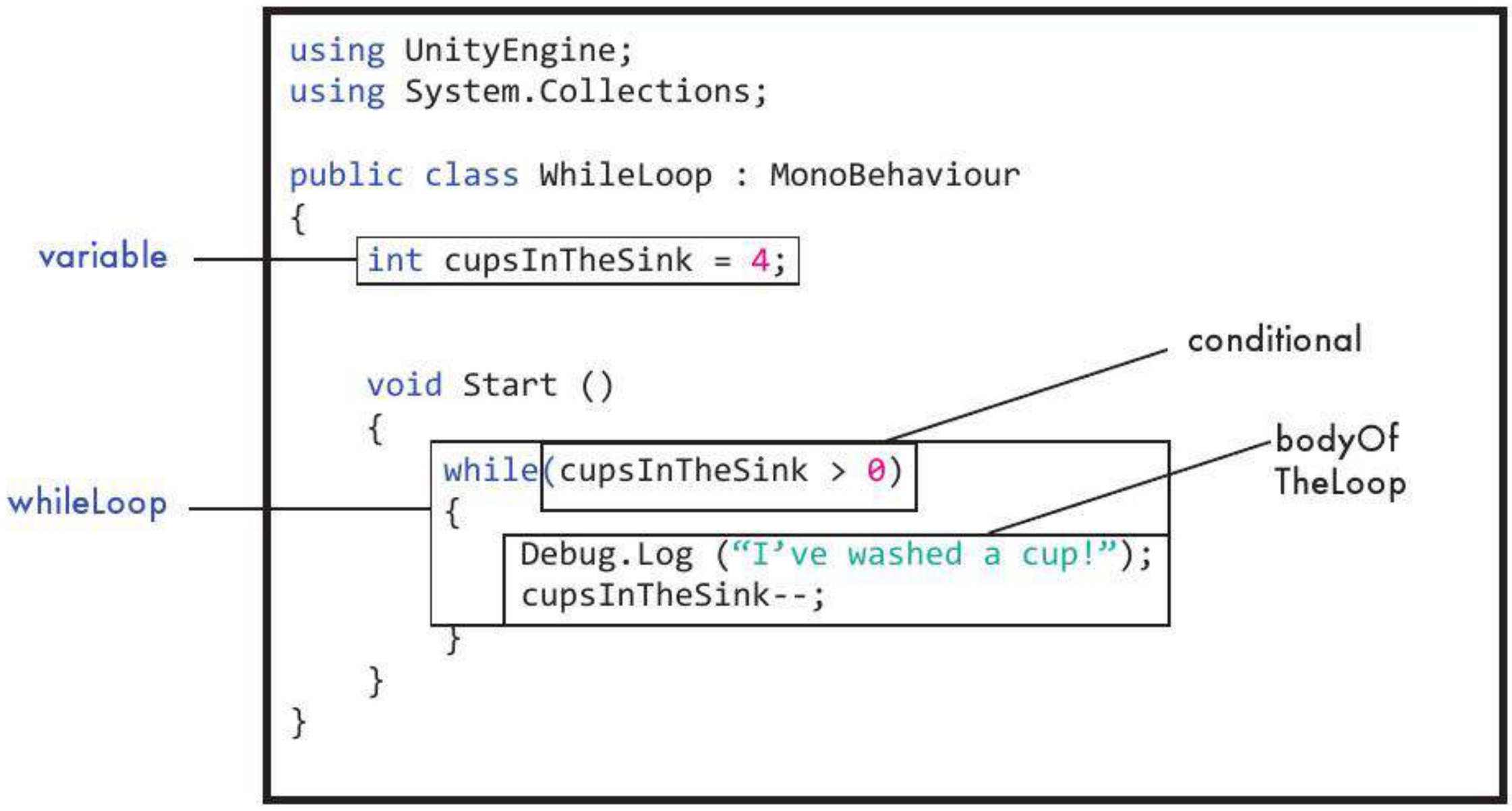
Less than symbol

Iteration

A word repeated again and again in this chapter, iteration is the act of repeating a process with the aim of approaching a desired goal, target or result. Each repetition of a process is also called an "iteration", and the results of one iteration are used as the starting point for the next iteration.

WhileLoop

In the While loop we can see that we have a variable called 'cups in the sink' which is set to 4. In the start method we have our while loop. The structure of the loop is as follows; we start with the key word while, then in parenthesis we have a conditional. The content of the loop will continue to repeat as long as the conditional value is true. The code in the braces represents the body of the loop. It is this code, which gets repeated. We will see this code will print "I've washed a cup!" then it will decrement the variable cups in the sink. The loop repeats these steps while the value of cups in the sink is greater than 0.



When we enable the script and run our scene, we see in the console that our 4 cups have been washed.

Do While Loop

The do while loop function almost identically to the while loop with one major difference, while loops test the condition before the loop body, however do while loops test the condition at the end of the body. This difference means the body of a do while loop is guaranteed to run at least once.

We can see that we have a Boolean variable named 'shouldContinue'. This variable is set to false. Next we have a do while loop. Notice the syntax of the do while loop; we start with the key word 'do', followed by opened and closed braces. Whatever code is in-between these braces makes up the body of the loop. After the body is the 'while' key word, followed by the conditional. In this case, the loop will only continue while the variable shouldContinue is equal to true. We can see the variable is set to false, so this conditional will resolve to false and the loop will exit. Finally, notice the semi-colon after the conditional. While loops don't have the semi-colon but do while loops do.

```
using UnityEngine;
using System.Collections;

public class DoWhileLoop : MonoBehaviour
{
    void Start()
    {
        bool shouldContinue = false;

        do
        {
            print ("Hello World");

        }while(shouldContinue == true);
    }
}
```

variable

doWhileLoop

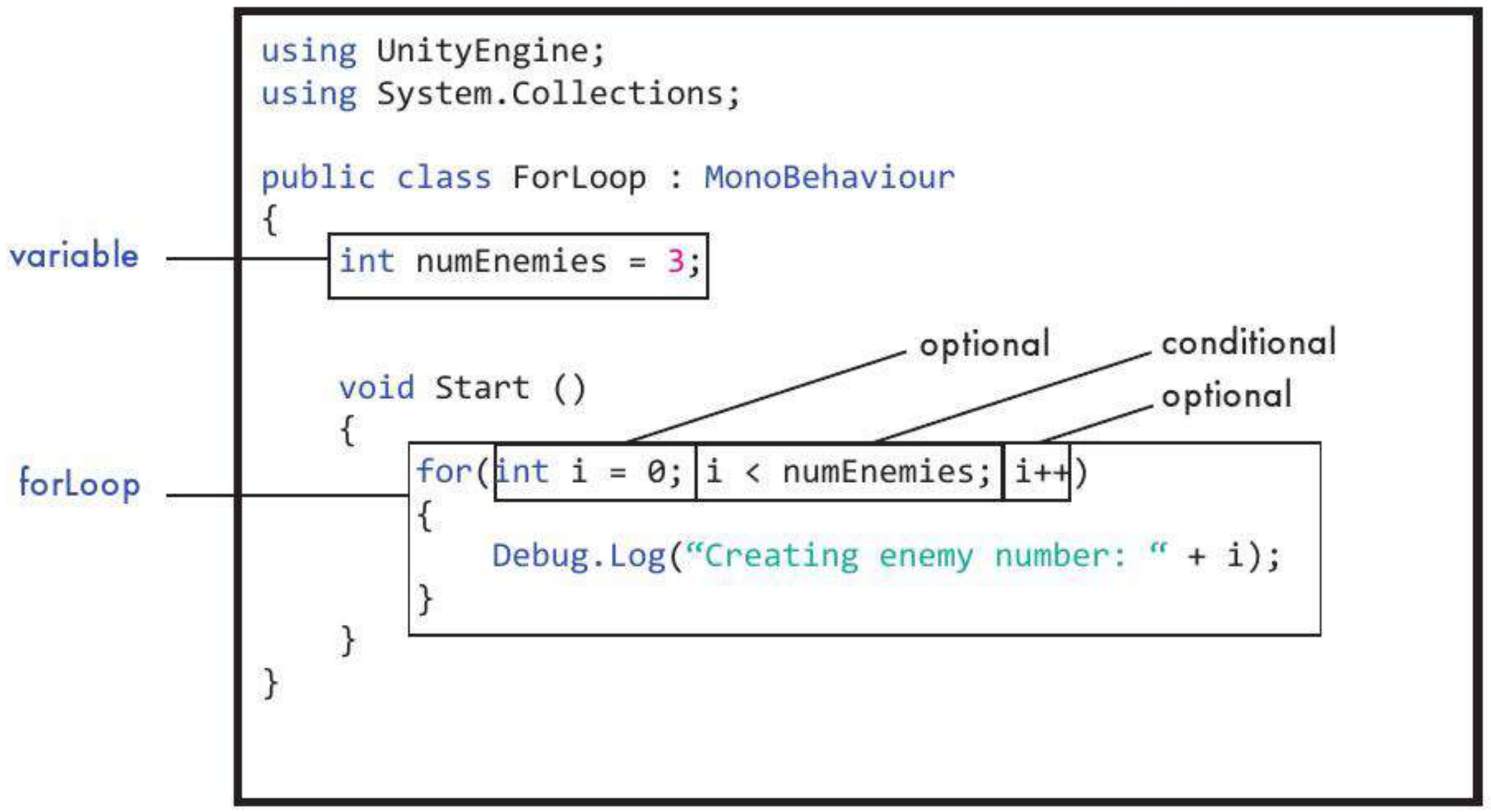
conditional

When we enable the script and run our scene, we can see that the phrase "Hello World" has printed to the console. Even though our conditional was resort to false at the beginning, the body of a do while loop always runs at least once.

For Loop

The for while loop functions similarly to the while loop but uses a different syntax.

In the for loop script we can see we have a variable called numEnemies that is set to 3. Next is the for loop which uses the following syntax. We start with the key word for then in the parenthesis we have 3 specific sections. The first section is optional and gives us a place to declare and initialise any variables we wish to use. Next we have the conditional, which must be supplied. This conditional will be evaluated before every iteration of the loop. Finally we have an optional section, which allows us to increment, or decrement any variables we would like. This section will run after every iteration of the loop. NOTE: it is a semi-colon and not a comma, which separates the different sections. Following the parenthesis is the two braces and the body for the full loop.



Due to its syntax the for loop is very useful for counting when the number of iterations is known. In this example we can see that this loop has created a variable called 'i' and set it to 0. This loop will run while 'i' is less than the variable numEnemies, which in this case equals 3. Finally, you will see that the final iteration of this loop will be increased by 1 (because of the ' i++'). In the body of the loop the words "Creating enemy number: " will be printed along with the value of 'i'.

It is worth noting that usually in programing counting begins at 0 and not 1. Therefore in this loop the variable 'i' starts at 0.

Predictably in the console view we can see that 3 enemies were created when we run the script.

For Each Loop

This loop is very useful for iterating through collections like arrays.

In this script you can see we have created a loop for 3 strings and given them some basic values. Then we have the for each loop. The for each loop basically loops through a collection item by item until it reaches the end of the collection. Once it has reached the end it stops. It uses the key word foreach followed by parenthesis. Inside the parenthesis we declare a
variable the same type as the collection or array we want to loop through. In this case we created an variable called item that is of type string. We follow this with the key word 'in' and the name of the collection we wish to loop through. Inside the body of the loop we are printing the variable item created above. Every time the loop iterates item will be the value of the next element in the collection or array. In this way it is easy to access each element of the collection without worrying how big the collection is.

Something worth noting is that you cant alter the elements of a collection using a for each loop. If you want to modify an element you will want to use a for loop.

```
using UnityEngine;
using System.Collections;

public class ForeachLoop : MonoBehaviour
{
    void Start ()
    {
        string[] strings = new string[3];

        strings[0] = "First string";
        strings[1] = "Second string";
        strings[2] = "Third string";

        foreach(string item in strings)
        {
            print (item);
        }
    }
}
```

array

forEachLoop

When we enable the script and run our scene, that even though we printed the value item each time all the strings of the array are written in the console. This is because the value of the item variable changed after every iteration.

https://unity3d.com/learn/tutorials/modules/beginner/scripting/variable-scope-and-access-modifiers

7

Scope and Access Modifiers

The scope of a variable is the area in code, which the variable can be used in. A variable is said to be local to the place in code that it can be used. Code blocks are generally what define a variables scope. And these are denoted by braces.

So for example, everything within the class in this example can be said to be local to that class. The variables pens, crayons and answer are all local to the example function and can not be used out side of that.

```
using UnityEngine;
using System.Collections;

public class ScopeAndAccessModifiers : MonoBehaviour
{
    public int alpha = 5;

    private int beta = 0;
    private int gamma = 5;
}

    void Example (int pens, int crayons)
{
    int answer;
    answer = pens * crayons * alpha;
    Debug.Log(answer);
}

    void Update ()
    {
        Debug.Log("Alpha is set to: " + alpha);
    }
}
```

You would say that the variables alpha, beta and gamma are in scope within the ScopeAndAccessModifiers class. And you would say that the pens, crayons and answer variables are in scope within the example function.

Public and private accsess modifiers.

Variables defined in the class as opposed to those declared within a function have an access modifier attributed to them.

Access Modifier

An access modifier is a key word placed before a data type when declaring a variable, and its purpose is to define where the variable or function can be seen from.

```
public class ScopeAndAccessModifiers : MonoBehaviour
{
    public int alpha = 5;

    private int beta = 0;
    private int gamma = 5;
}
```

As a general rule if other scripts need access to a variable or function then it should be public, otherwise it should be private. Declaring a variable as public means that it can be accessed from outside the class. It also means the variable is shown and editable on the component in the inspector.

In this example, my ScopeAndAccessModifiers script is applied to our prop samoflange object, and you can see that the public variable is included as a property that I can edit. This allows the user to edit the variable whilst they test the game.

Imagine for example the value controls the speed of a car. It would be nice to be able to tweak the variable whilst testing it without having to stop edit the script and play again. As such it makes sense to have this be a public variable. Note that if a variable is initialised in the class to a default value, for example if we had put alpha equals 5, then it will still be overridden by the value that's written in the inspector. However, if these values are set in functions such as Start and Awake these occur after the variable has been set in the inspector and will not be overridden.

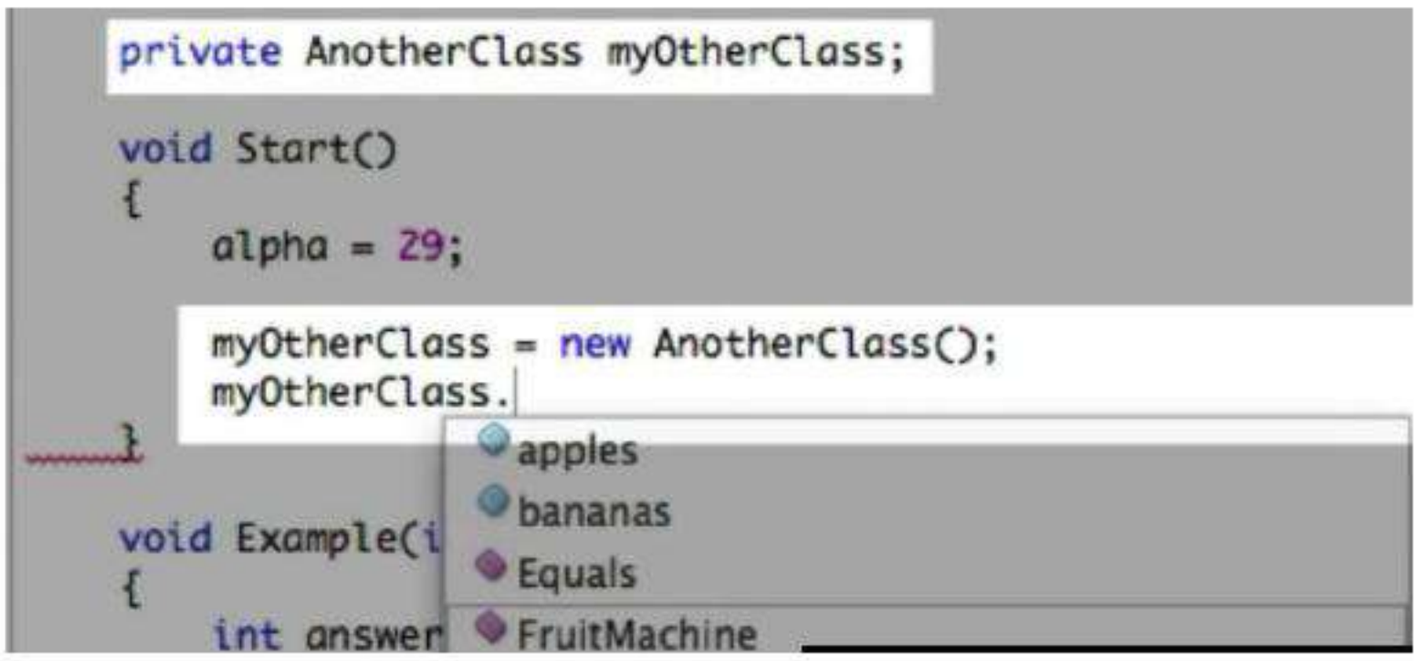
So for example, if my script had a default value of 5 but was being overridden by the inspector then setting it manually with a Start function will override the value in the inspector. Now given that the game is at run time if I tweak the value again that's overriding the script again. So bear in mind that even though the public variable is tweakable on the component it doesn't mean that you can't go back and edit it via a script during runtime.

Private variables can only be edited from within the class. In C# private is the default access modifier for any variable that doesn't have it specified. So although I've written private on beta and gamma, if I haven't they would be private regardless. It's good practice to make all member variables, those that belong to a class rather than a function private, unless they need to be public for a specific reason. Setting variables and functions to public can also mean that you access them from other scripts.

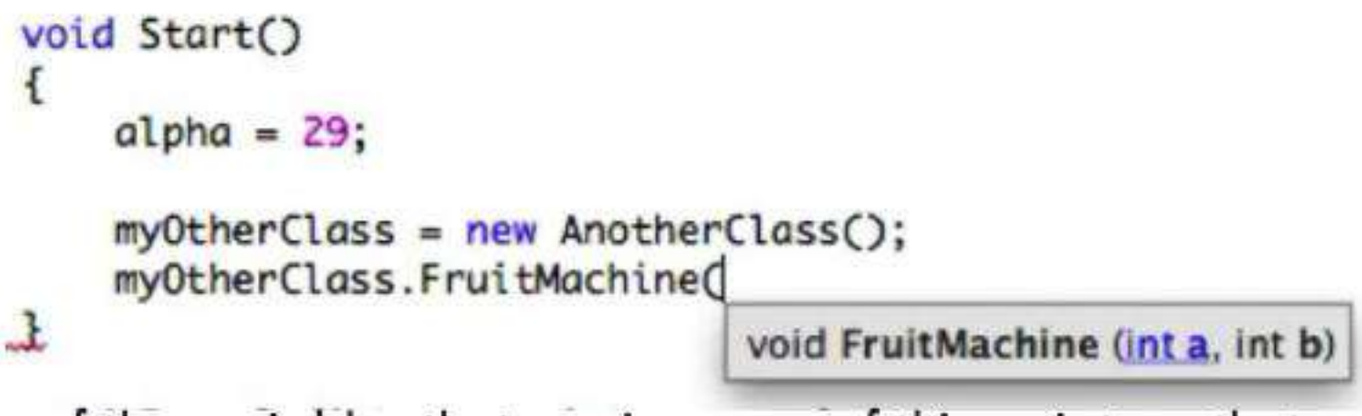
So for example I have another class here, which has 2 functions in, a public function called FruitMachine and a private called OfficeSort. I have some public variables and private variables.

```
using UnityEngine;
using System.Collections;

public class AnotherClass
{
    public int apples;
    public int bananas;


    private int stapler;
    private int sellotape;


    public void FruitMachine (int a, int b)
    {
        int answer;
        answer = a + b;
        Debug.Log("Fruit total: " + answer);
    }


    private void OfficeSort (int a, int b)
    {
        int answer;
        answer = a + b;
        Debug.Log("Office Supplies total: " + answer);
    }
}
```

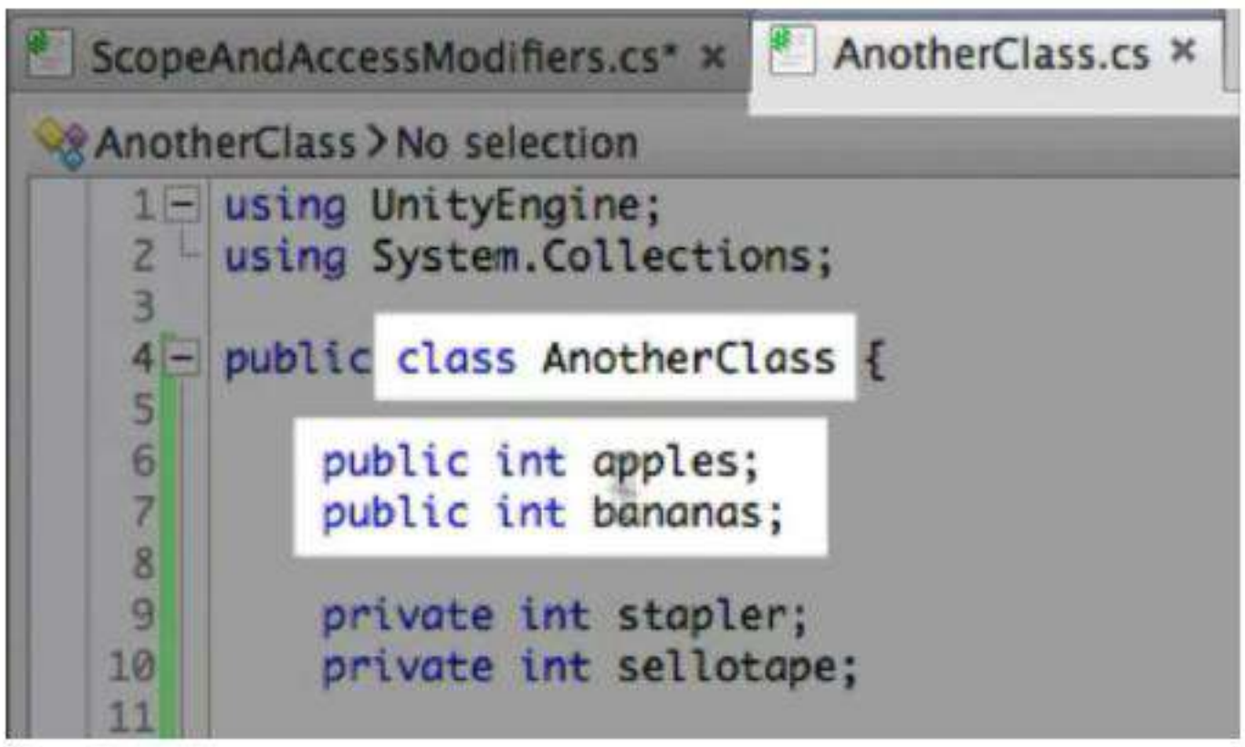
Back in my original script I can call only the public members of that script. For example, if I were to create an instance of the class AnotherClass, and you can see that when I try and access this the apples and bannana variables as well as FruitMachine are available, and you can see that in telesense in Mono Develop is telling me that I can use these.



So when I use FruitMachine and open the bracket it's then telling me what arguments need to be fed into that.



And I can feed in any of the variables that are in scope of this script, or that are in scope of AnotherClass because they are public.



So I could use any of the variables from my scope and access modifiers class and I can access the instance of another class and make use of a public variable from there, such as apples for example.

Note that because the OfficeSort and stapler and selltape are all private I cant access them by creating an instance of another class. They're for use only within the class itself.

https://unity3d.com/learn/tutorials/modules/beginner/scripting/awake-and-start

8

Awake and Start

Awake and Start are two functions that are called automatically when a script is loaded.

Awake()

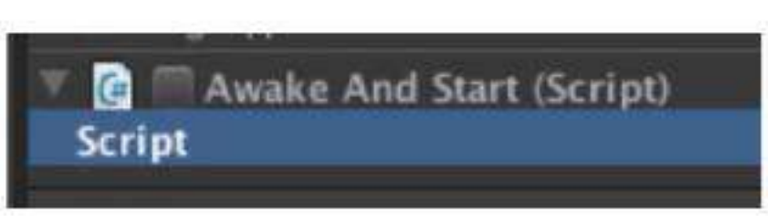
Awake is called first even if the script component is not enabled and is best used for setting up any references between scripts and initialisation. (Initialization is the process of locating and using the defined values for variable data that is used.)

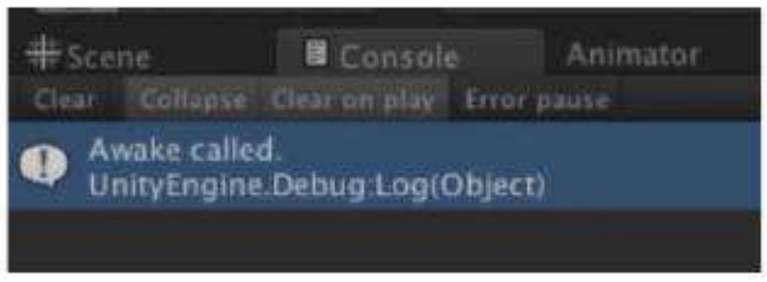
Start()

Start is called after Awake, immediately before the first update, but only if the script component is enabled. This means you can use Start for anything you need to occur when the script component is enabled. This allows you to delay any part of your initialisation code until it's really needed.

For example an enemy character could enter the game and use Awake to have ammo count assigned to him, but only get the ability to shoot, using Start at a defined time when that script component is enabled. It should be noted however that Start and Awake will only ever be called once in the lifetime of a script attached to an object. So you cannot repeat the start function by disabling and re-enabling a script.

In this script we have a script that simply has Awake and Start functions. We're using Debug.Log to show when those two things occur. For example without this script component enabled we simply see the Awake function.





But if I enable the script I will see that both functions have been called. This is useful as it allows you to initialise settings for an object before enabling that script component without the need for splitting the script in to several different scripts.

https://unity3d.com/learn/tutorials/modules/beginner/scripting/update-and-fixedupdate

9

Update and FixedUpdate

Update is probably the most commonly used function in Unity. It's called once per frame on every script that uses it. Almost everything that needs to be changed or adjusted regularly happens here.

Update();

- Called every frame
- Used for regular updates such as:
  - Moving Non-Physical Objects
  - Simple Timers
  - Receiving Input
- Update interval times vary

The movement of non-physics objects, simple timers and the detection of input are just a few things that are usually done in Update. Note that update is not called on a regular timeline. If one frame takes longer to process than the next then the time between update calls will be different.

FixedUpdate();

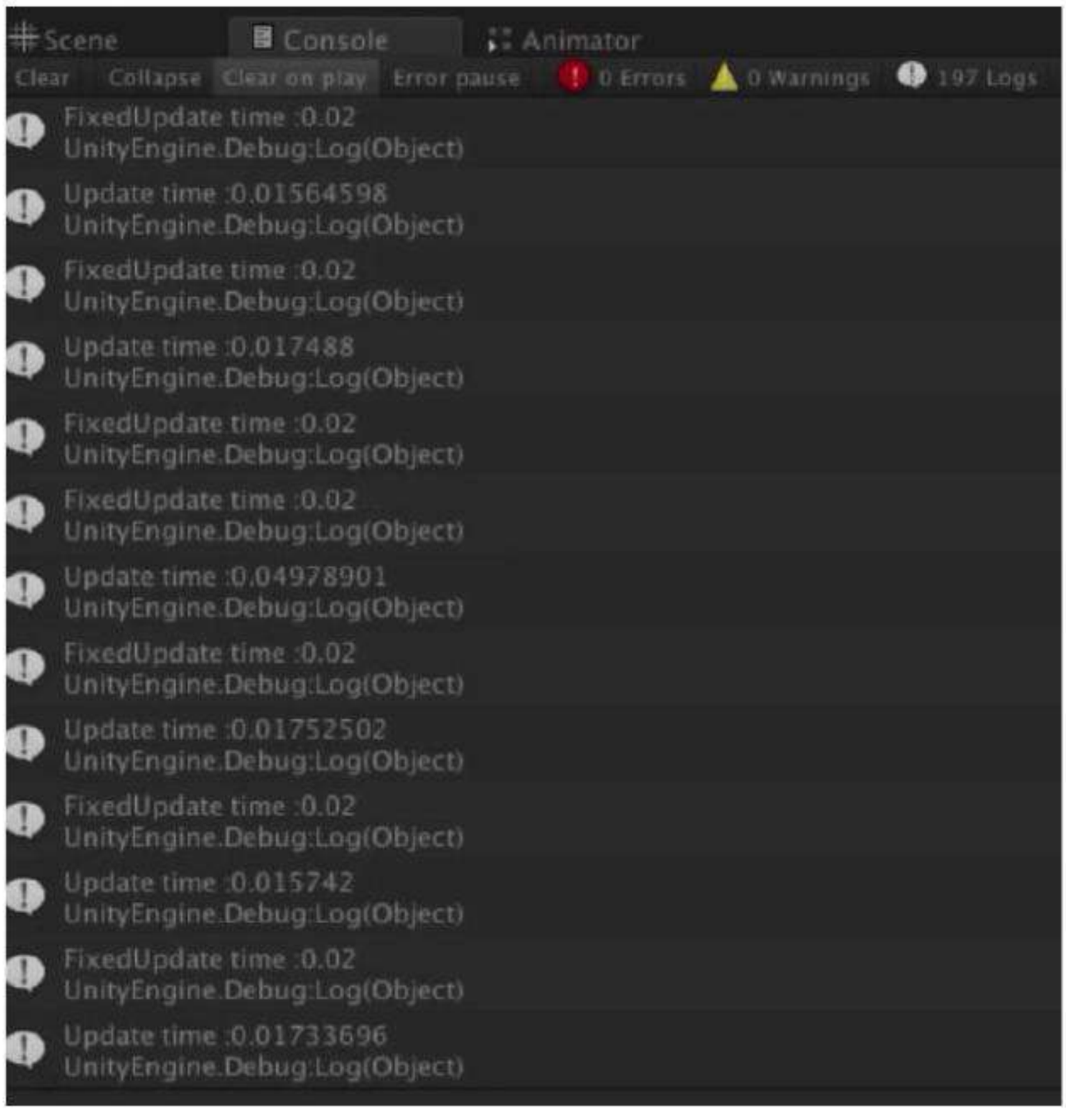
- Called every physics step
- FixedUpdate intervals are consistant
- Used for regular updates such as:
  - Adjusting physics (Rigidbody) objects

Fixed Update is a similar function to update but it has a few important differences. FixedUpdate is called on a regular timeline and will have the same time between calls. Immediately after FixedUpdate is called; any necessary physics calculations are made. As such anything that affects a rigidbody (meaning a physics object) should be executed in FixedUpdate rather than Update. When scripting physics in the FixedUpdate it's good practice to use forces for movement.

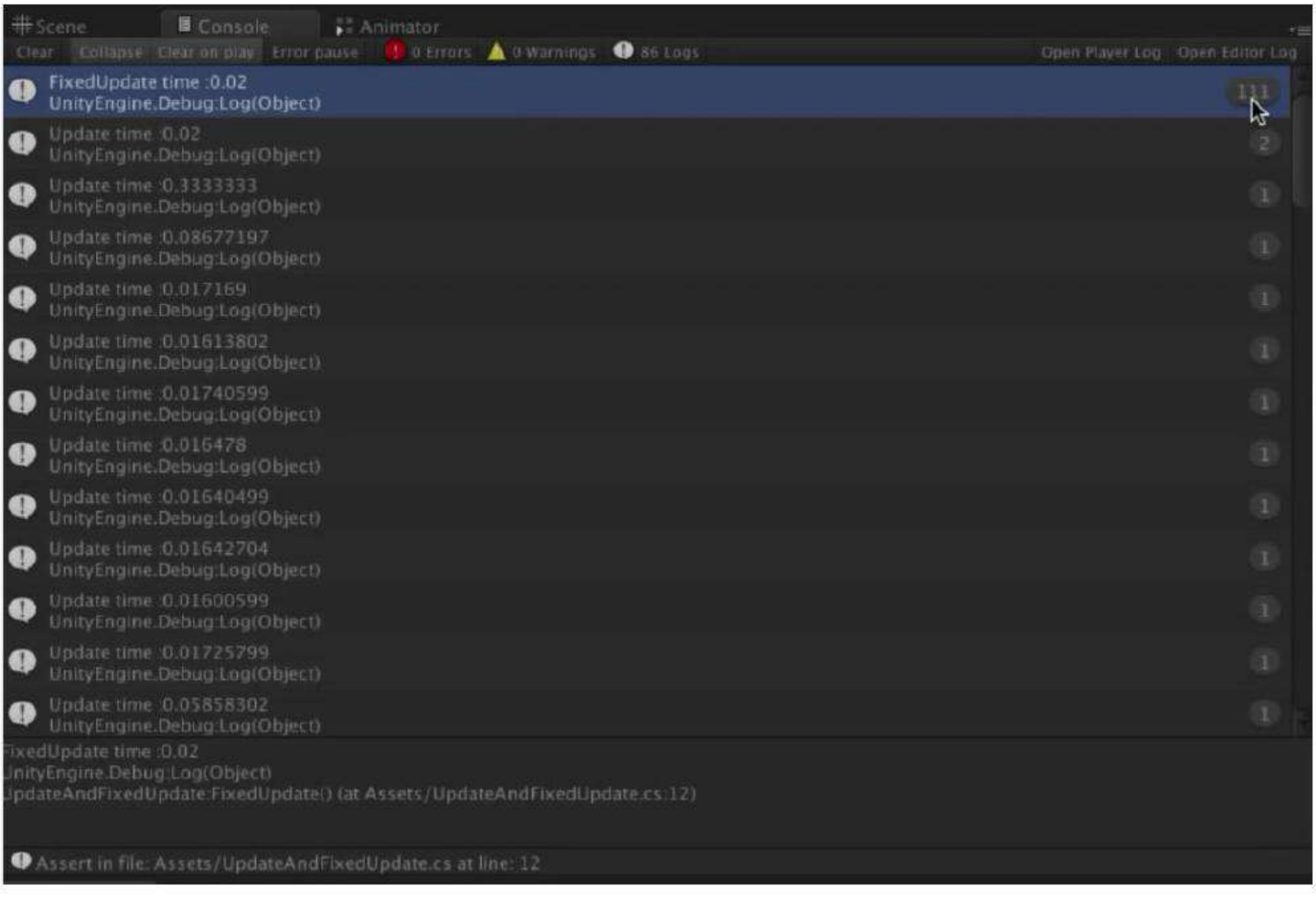
In this example we're logging to the console with each FixedUpdate and Update by adding the value of Time.deltaTime. When I press play and pause you can see that the time between FixedUpdate is always 0.02, where as the time between Update varies. This can be shown even more clearly by collapsing the console.

```
using UnityEngine;
using System.Collections;

public class UpdateAndFixedUpdate : MonoBehaviour
{
    void FixedUpdate ()
    {
        Debug.Log("FixedUpdate time :" + Time.deltaTime);
    }


    void Update ()
    {
        Debug.Log("Update time :" + Time.deltaTime);
    }
}
```

When I press play and pause you can see that the time between FixedUpdate is always 0.02, where as the time between Update varies. This can be shown even more clearly by collapsing the console.



You can see all of my FixedUpdates are collapsed into one. There have been 111 of them and they're all 0.02. Where as the Update times vary.



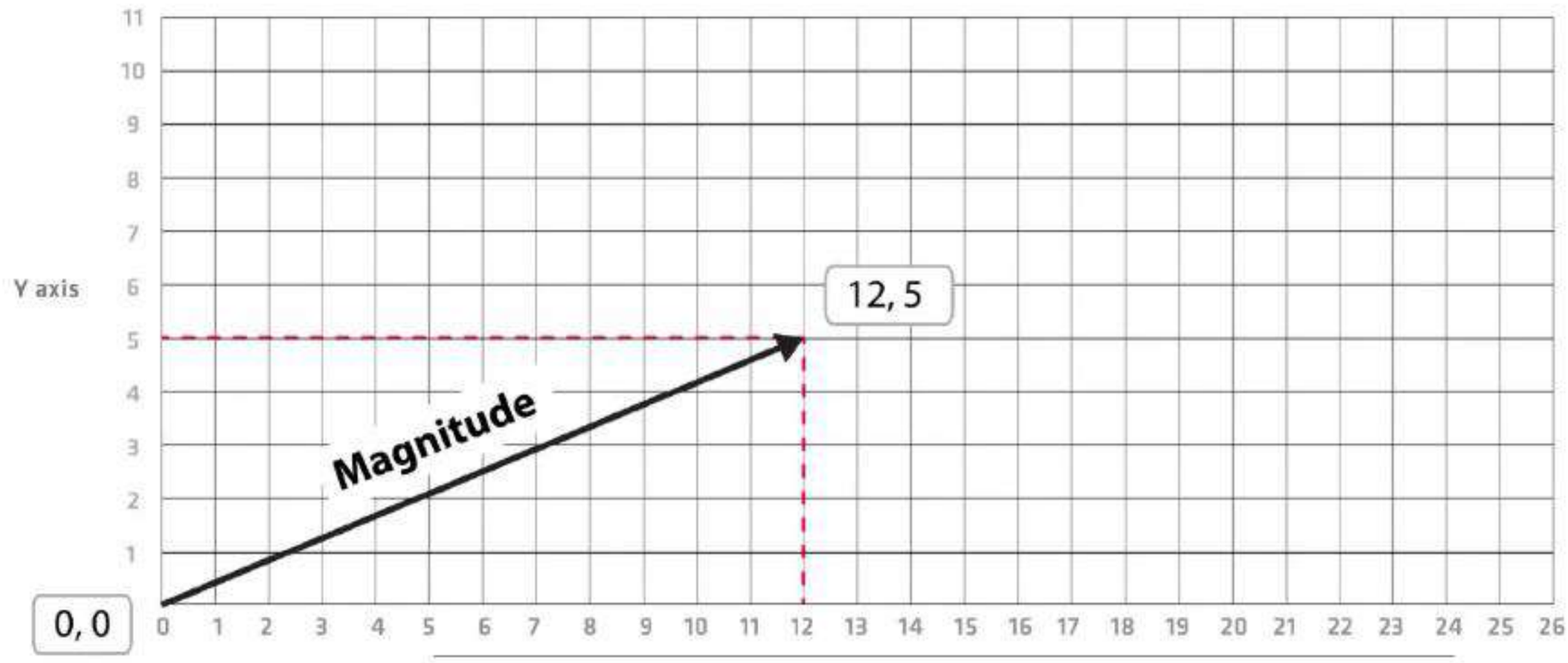
https://unity3d.com/learn/tutorials/modules/beginner/scripting/vector-maths-dot-cross-products

10

Vector Maths

In game development we find meshes, directions and all manner of other calculations, which makes them essential to understand. A vector is a line drawn between 2 points. Vectors also have a length, known as their magnitude.

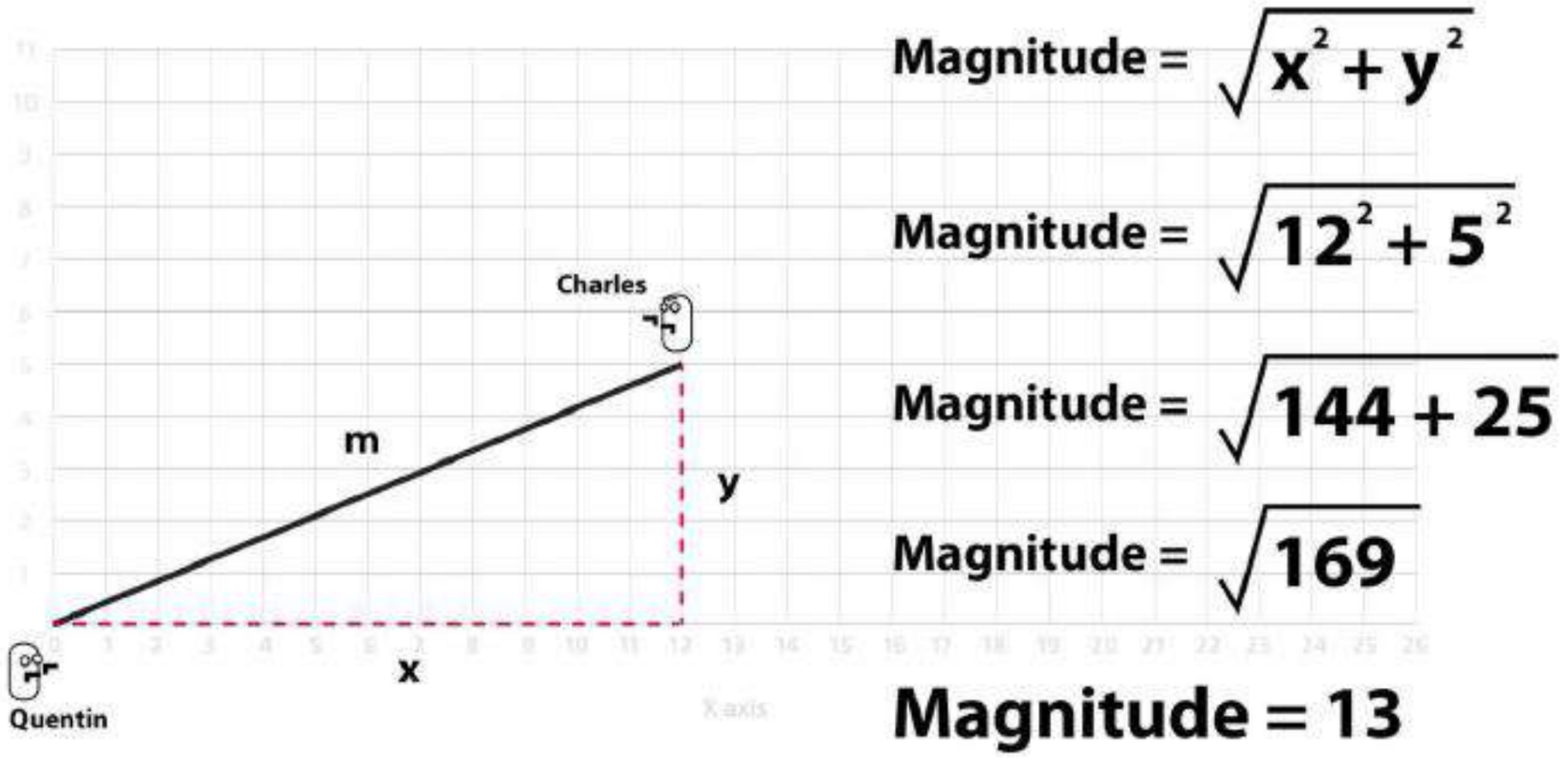
Lets start off by simply looking at 2D vectors. A 2D vector is a way of representing a point from the origin, the 0,0 point on a graph, to any point on a 2D plane. Since it is from the origin, it has an implied direction. It's made up of two components, X and Y. These represent these represent the distance from 0 along the X and Y in each axis. In this example our vector goes from the origin to position 12, 5. The length of the distance between these points is called the magnitude. This can be worked out mathematically by using Pythagoras's Theorem, which states that the square of the hypotenuse is equal to the sum of the squares on the other two sides. The hypotenuse in vector maths is the magnitude that we're trying to find.



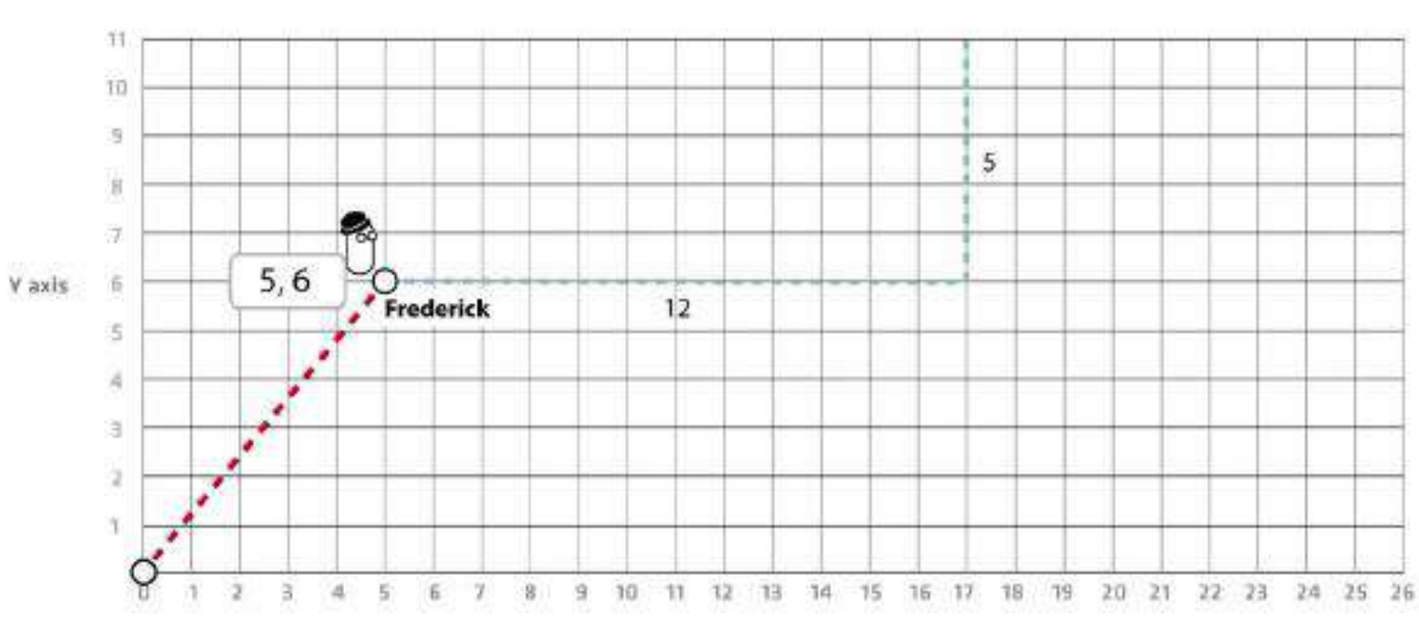
Imagine that there are two people in a field called Charles and Quentin with a lethal vendetta. Being gentlemen they've agreed to a pistol duel, but their guns only have a range of 12 units, so can they shoot one another?

2D Vectors

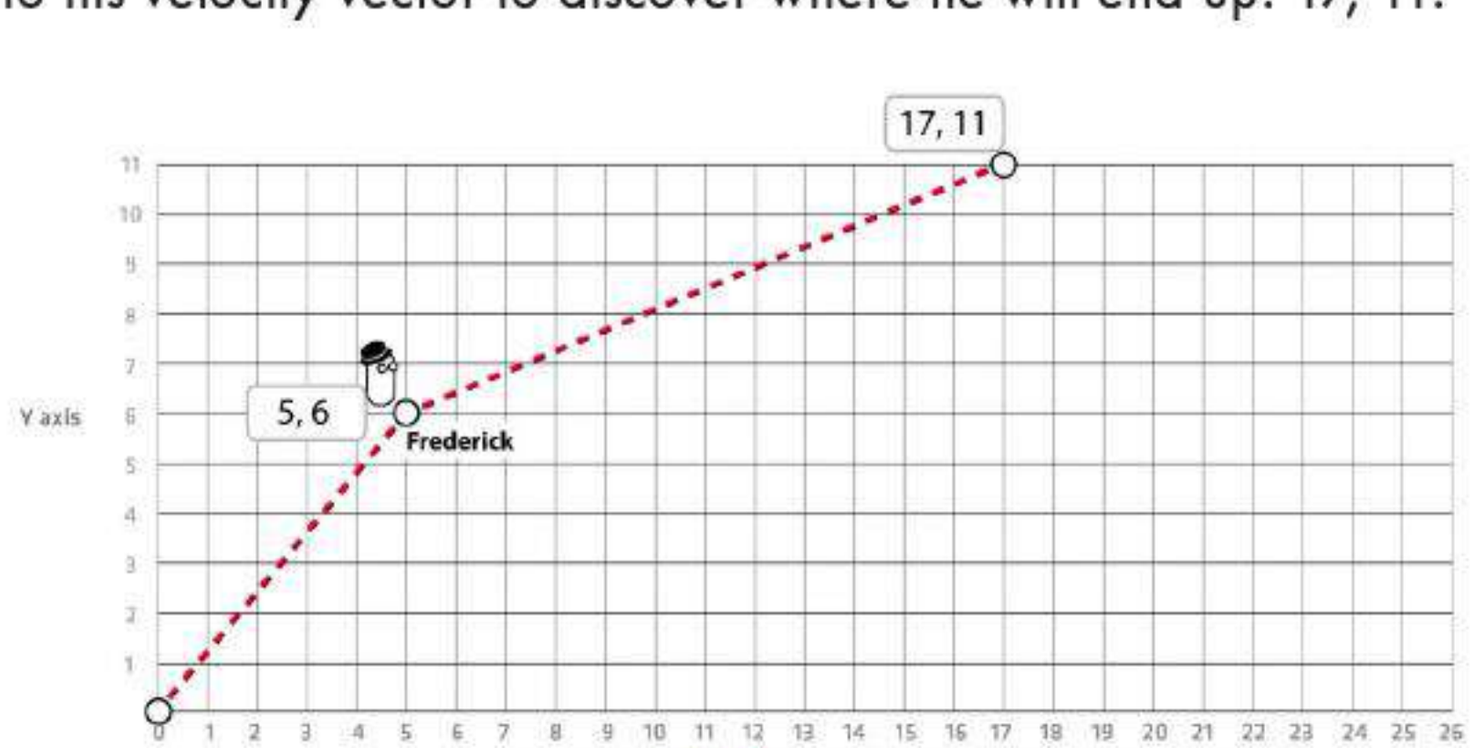
$$x^2 + y^2 = m^2$$ — Pythagoras Theorem



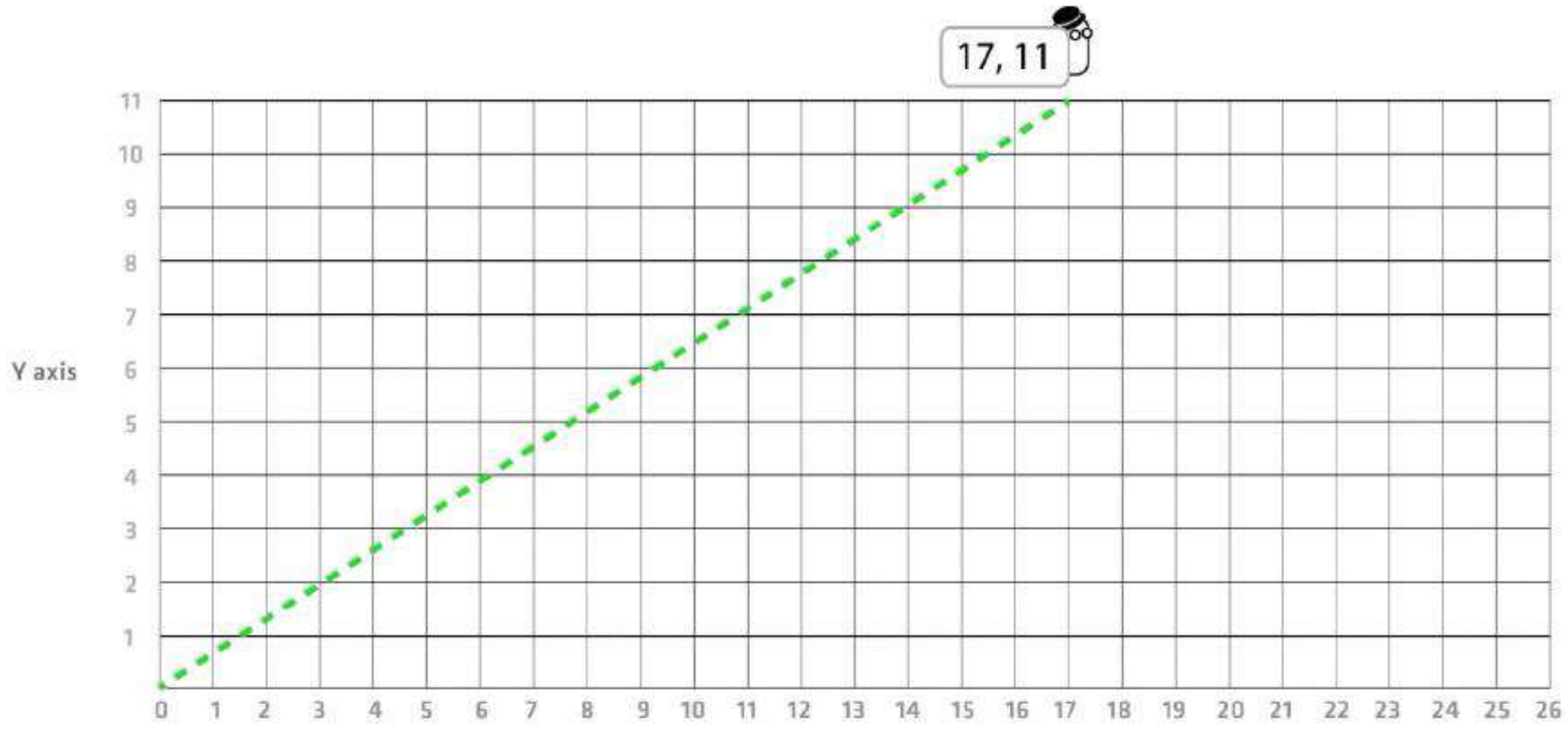
As you can see the magnitude is the square root of the sum of the X and Y positions on the grid squared, which when added together makes 169, the square root of 169 is 13. So Charles and Quentin will be unable to shoot one another, which is excellent, as violence has no place in this tutorial.



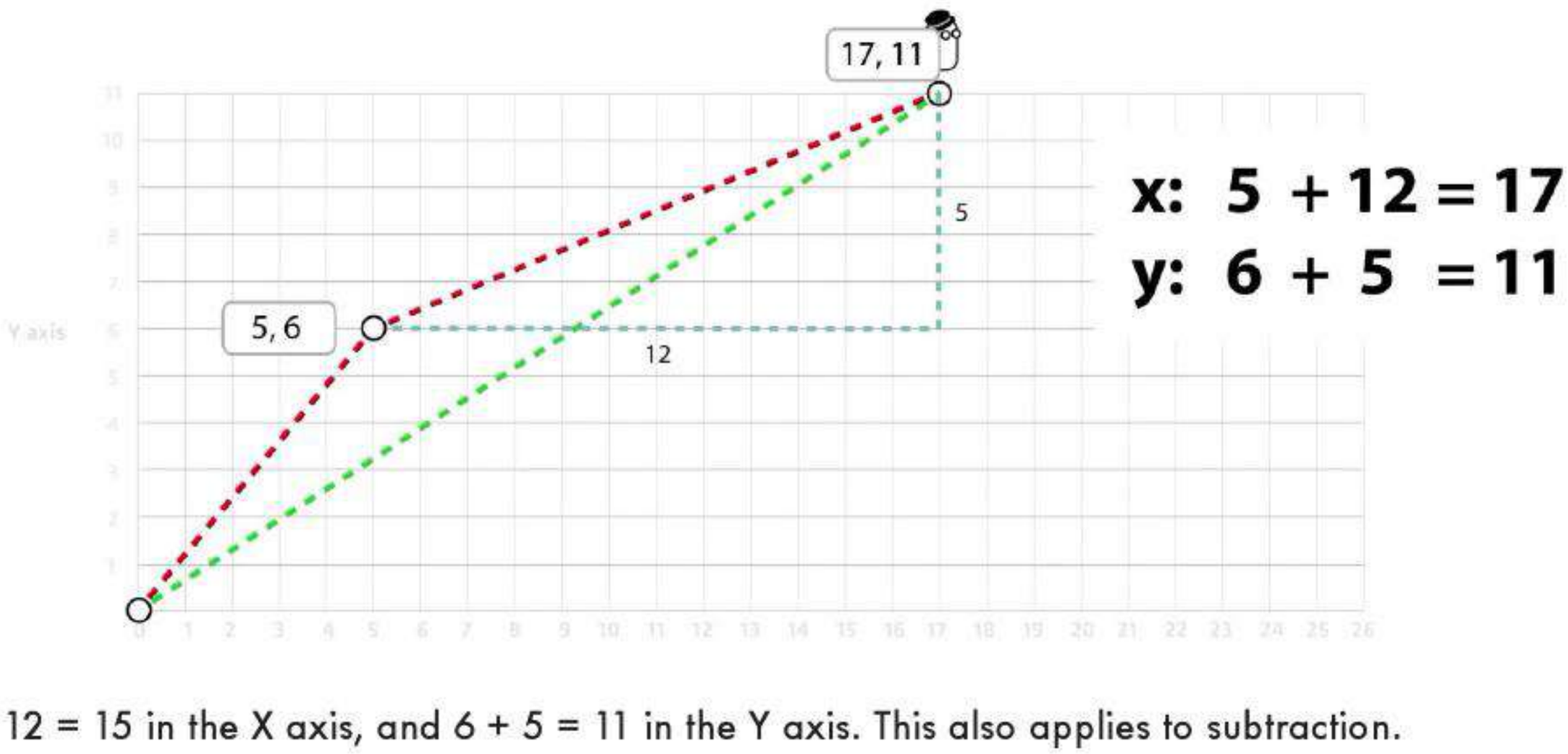
We've already learnt that vectors can be used to denote a position in space relative to the origin point. But you should also know that a moving object has a velocity, a change in position over time. This can also be expressed as a vector. In this diagram Frederick is at position 5,6 and has a velocity of 12, 5 per hour. This means he will travel in a position that is 12 further in the X axis and 5 further in the Y axis. So in order to find his new position after one hour we'd add his current position vector to his velocity vector to discover where he will end up. 17, 11.



Remember that all vectors are expressed relative to the origin, the 0 point in space. And this is no different for velocity vectors. In the same way that 5,6 is a position relative to the 0 position, 12, 5 is a velocity relative to 0 motion. So whilst he may still have the same velocity 17, 11 is Fredericks new position vector after 1 hour.

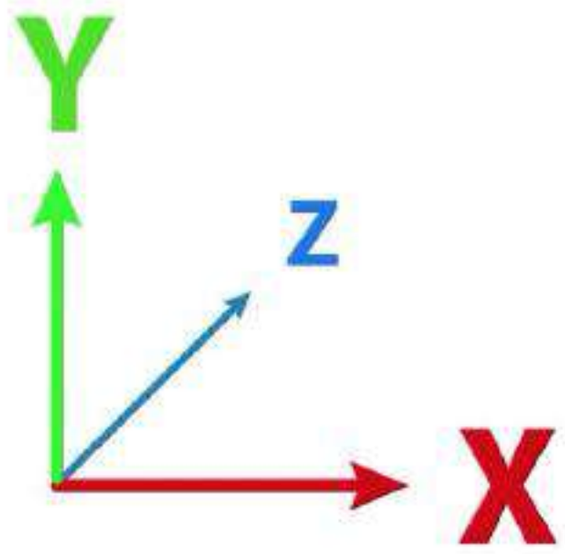


This is useful in game development tasks that involve prediction. You should note that the components of the final position are equal to the sum of the components of the two vectors.



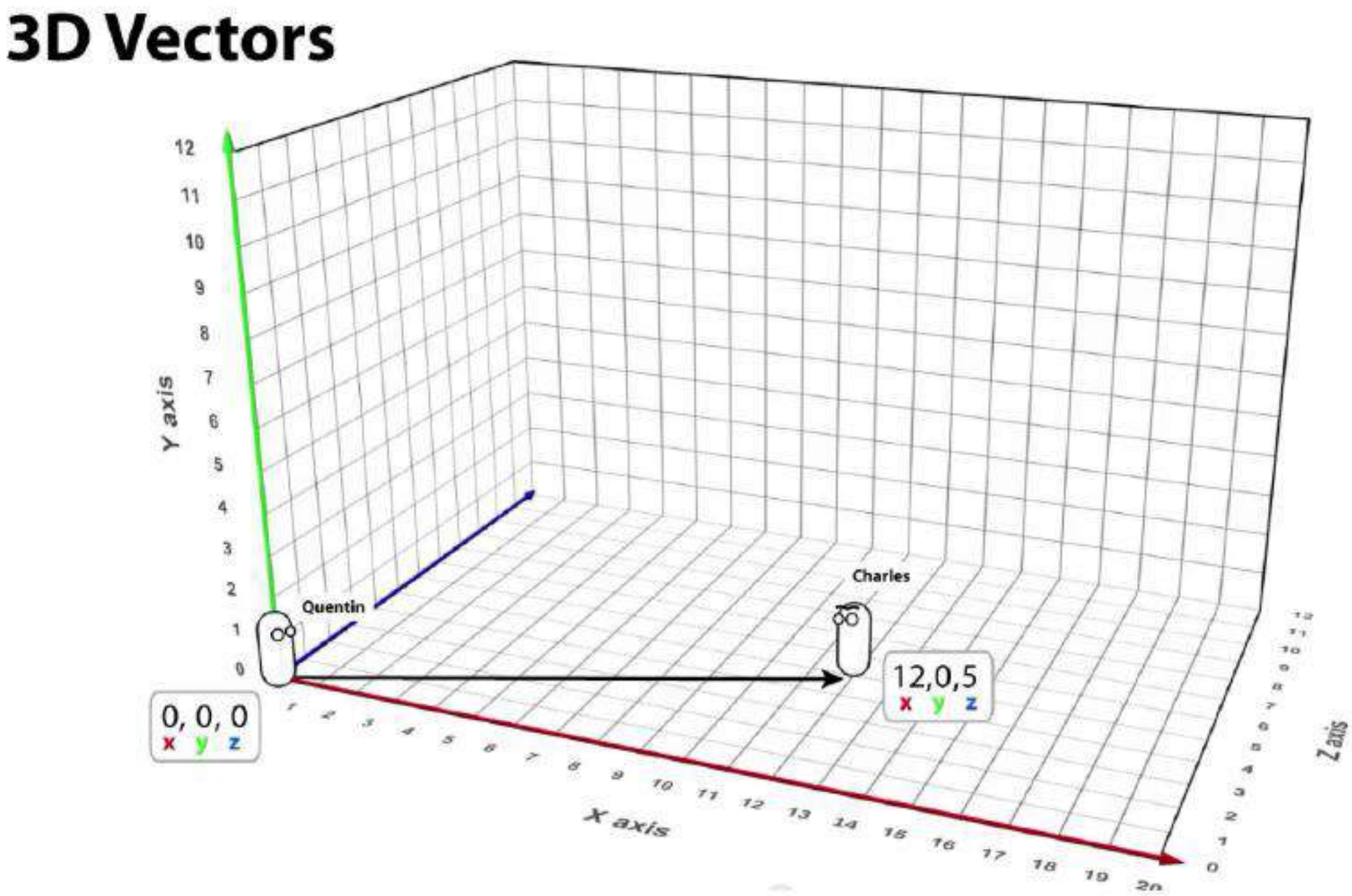
5 + 12 = 15 in the X axis, and 6 + 5 = 11 in the Y axis. This also applies to subtraction.

3 Dimentional vectors also work the same as 2D but extrapolated in the Z axis which represents depth. The X and Z axis make up a horizontal plane and Y is the direction that points up.

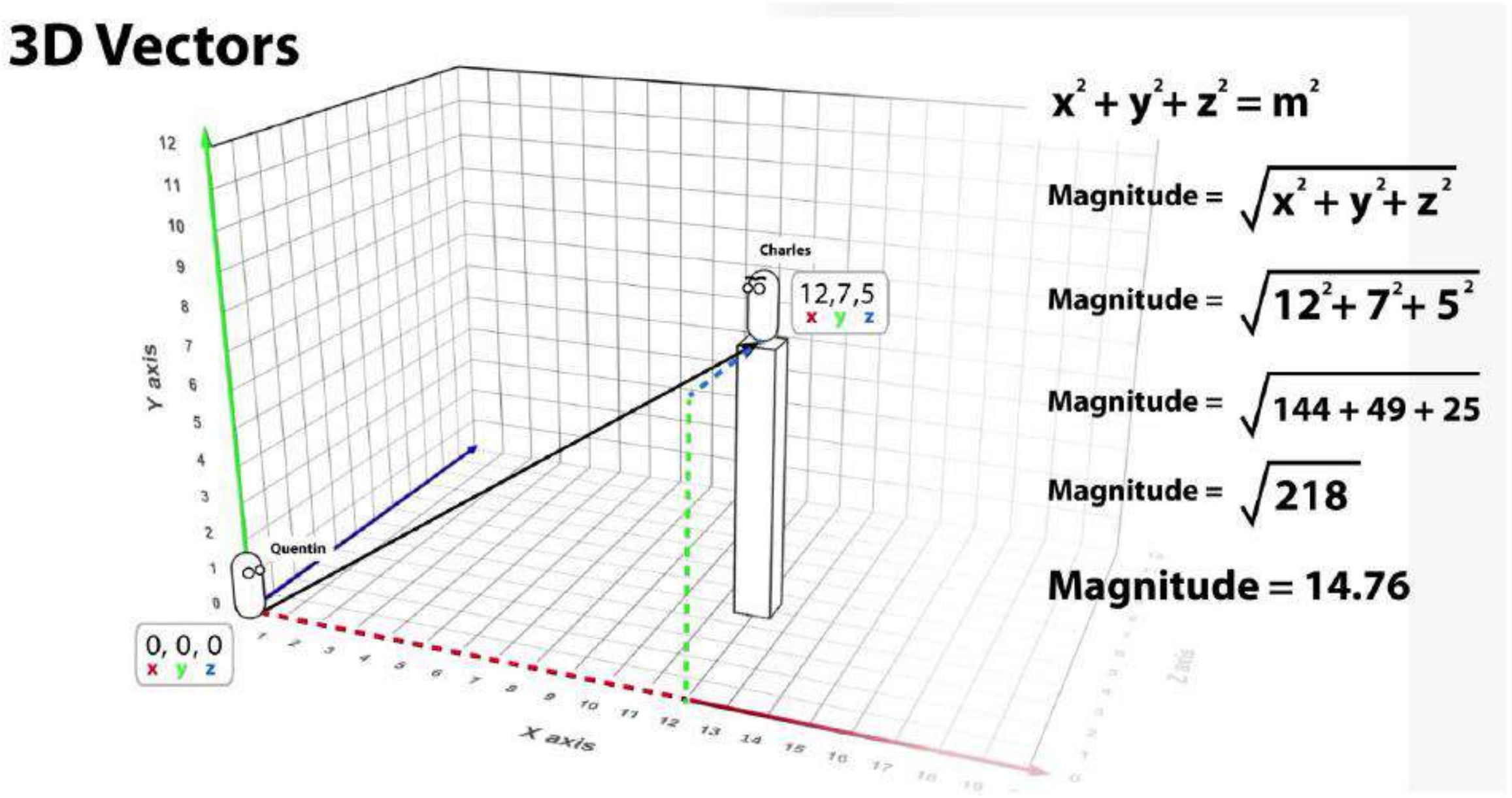


Unity works on a Left hand co-ordinate system, this means that if you hold up your left hand with your index finger pointed upwards and your thumb pointed out to form an L shape and your middle finger pointed away from you then the thumb represents the X axis, the index finger the Y axis and the middle finger the Z. This hand gesture can be helpful to remind you of the order of the axis X, Y and Z.

Given the Z represents depth, in our previous example, Charles and Quentin would have actually be stood on the X - Z plane, keeping in mind that any co-ordinates stated in 3D will be ordered X, Y, Z. Quentin would have been stood at (0, 0, 0) the 3D origin. And Charles at (12, 0, 5).



To make things more 3-dimnetional we could imagine that Charles was on a higher vantage point. Now stood upon his 7 unit tall podium, Charles is at position (12, 7, 5). The calculation to find the magnitude of the vector between the Charles and Quentin is the same as in 2D but now features the Z axis.

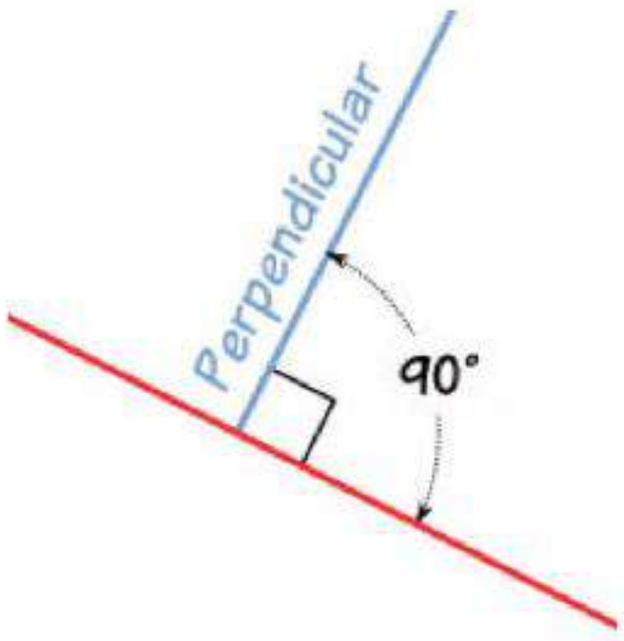


Unity makes it easy to perform calculations like this with its helper function Vector3.magnitude.

Vector3.magnitude

Vector 3 - Representation of 3D vectors and points.
magnitude - A variable that returns the length of this vector.

There are a couple of useful functions for 3D vectors, the Dot and Cross products.



The Dot product takes two vectors and produces a scaler, a single value from them. To find the Dot product of two vectors we take their component parts, the X, Y and Z values and multiply them together to find the sum of the resultant values. This is expressed for example (Ax*Bx)*(Ay*By)*(Az*Bz). With this product you can find out

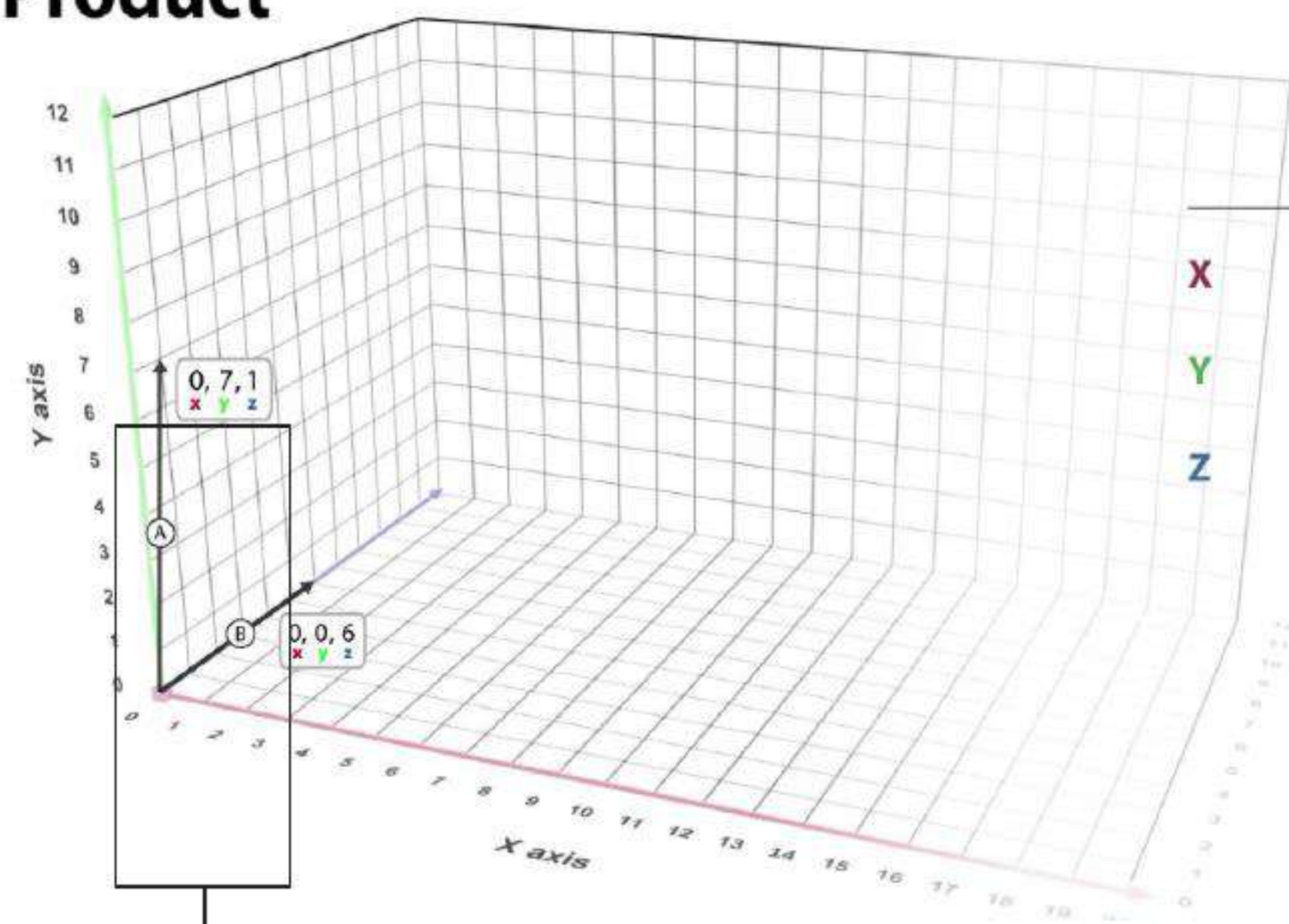
Vector A (x,y,z)

Vector B (x,y,z)

(Ax * Bx) + (Ay * By) + (Az * Bz) = Dot Product

With this product you can find out information about the 2 vectors you've specified. One example of this is finding out weather the two vectors are perpendicular to one another. If the dot product of two vectors equals 0, the vectors are perpendicular. In this example we have two vectors A and B. Vector A is at (0, 7, 1) and Vector B is at (0, 0, 6). We can see in this example that the sum of the multiplied components is 6, which means that the vectors are not perpendicular.

Dot Product

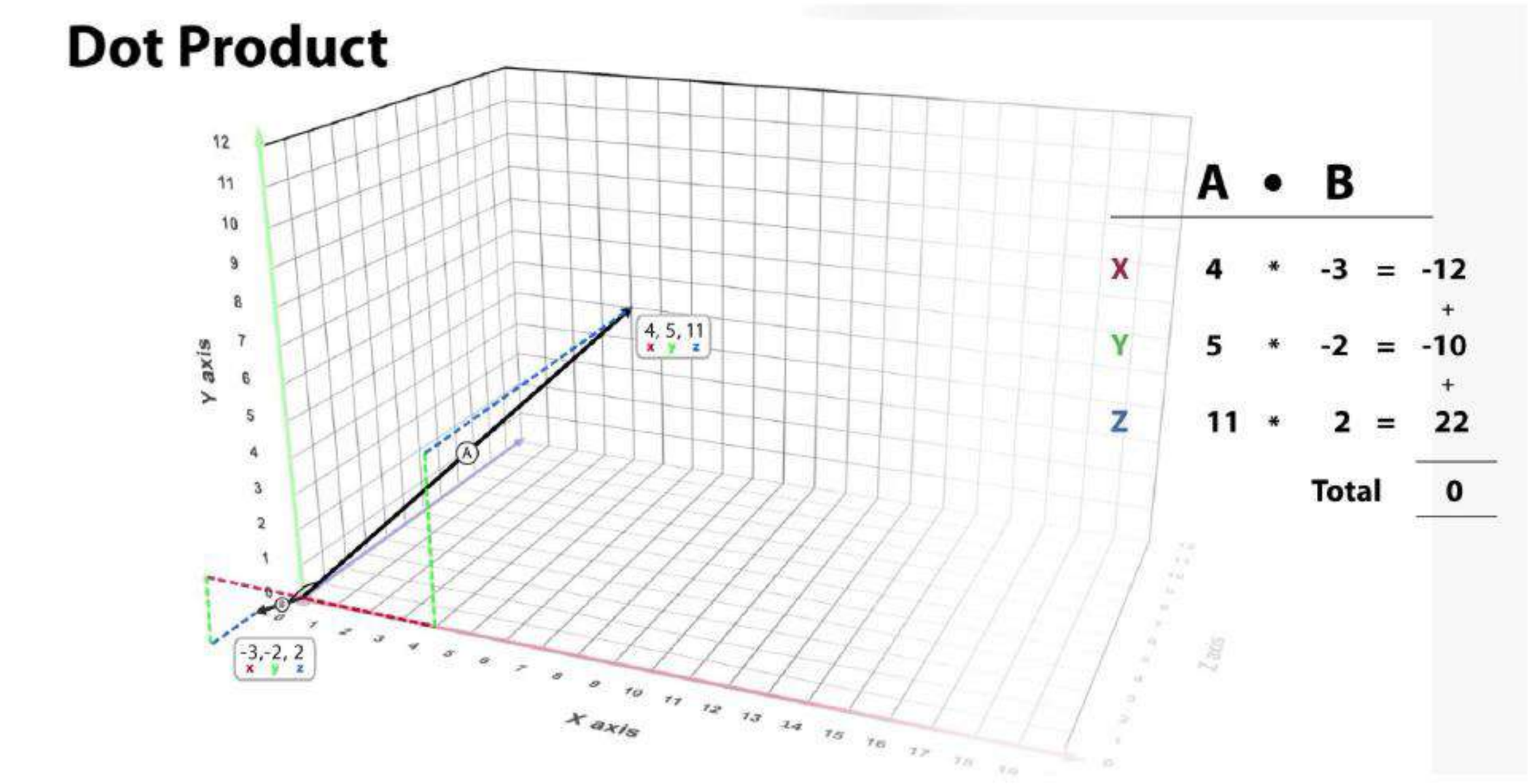


A • B				
X	0	*	0	= 0
				+
Y	7	*	0	= 0
				+
Z	1	*	6	= 6
			Total	6

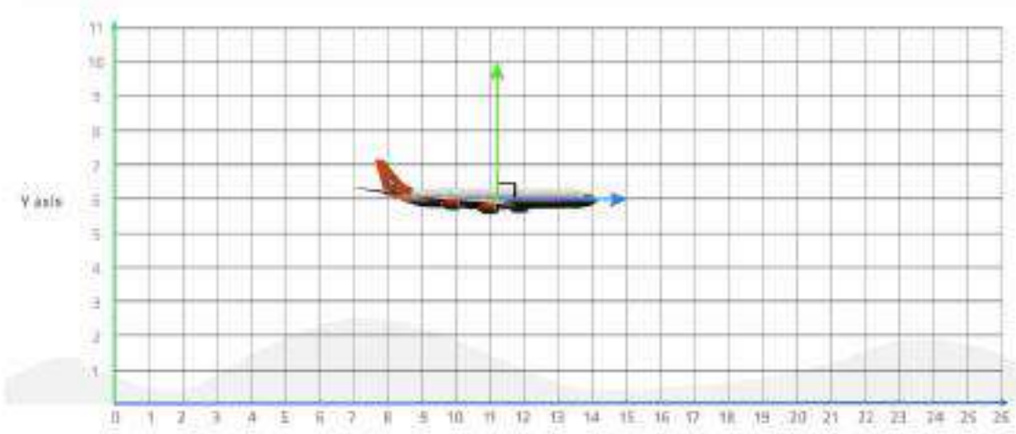
Not perpendicular

If vector A was (0, 7, 0) and B was (0, 0, 6) then we can see that the dot product equals 0 and the vectors are indeed perpendicular.

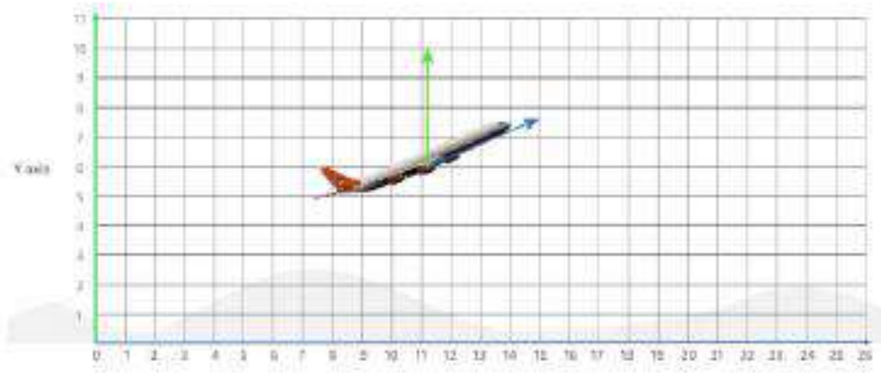
Here is another example of using the dot product in this way. This time Vector A is (4, 5, 11) and Vector B is (-3, -2, 2). Multiplying these components gives us (-12, -10, 22), the sum of which is 0. So these two vectors are also perpendicular.



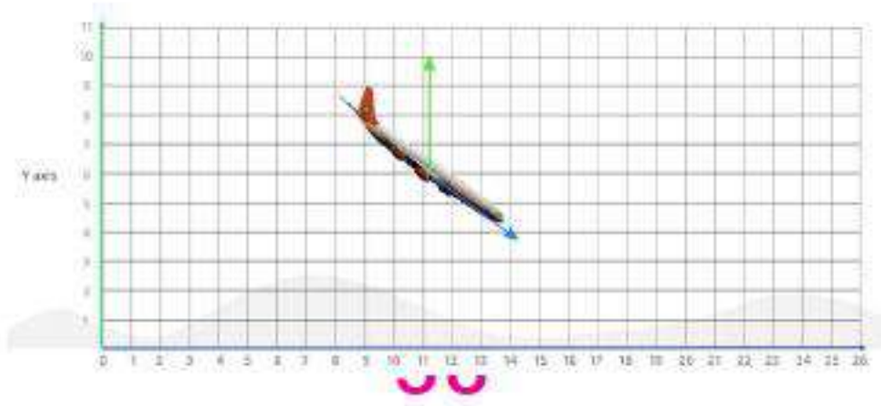
An example of using the dot product could be when creating a flight simulator. You would check the world up vector, with the forward vector of the plane. If the two were perpendicular (if the dot product equalled 0) then the plane should have the least amount of drag.



As the dot product increases in a positive value we would know that the plane is pulling up and we could add more drag.



If the dot product increases in a negative value then we know the plane is in a dive.



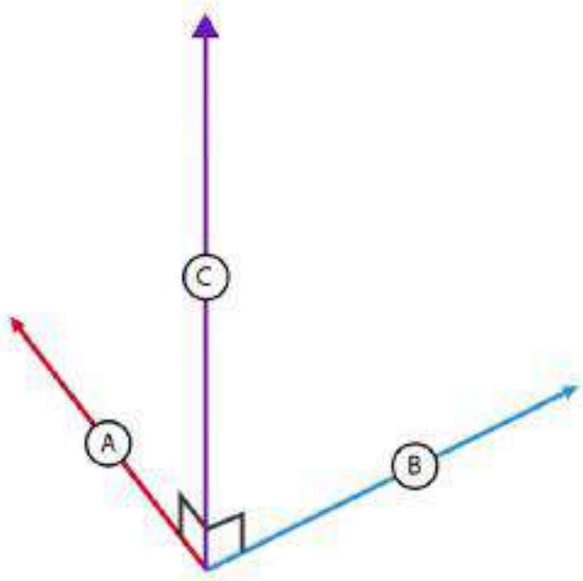
Unity has a helper function to perform dot product calculations easily.

Vector3.Dot(VectorA, Vector B)

The dot product is a float value equal to the magnitudes of the two vectors multiplied together and then multiplied by the cosine of the angle between them.

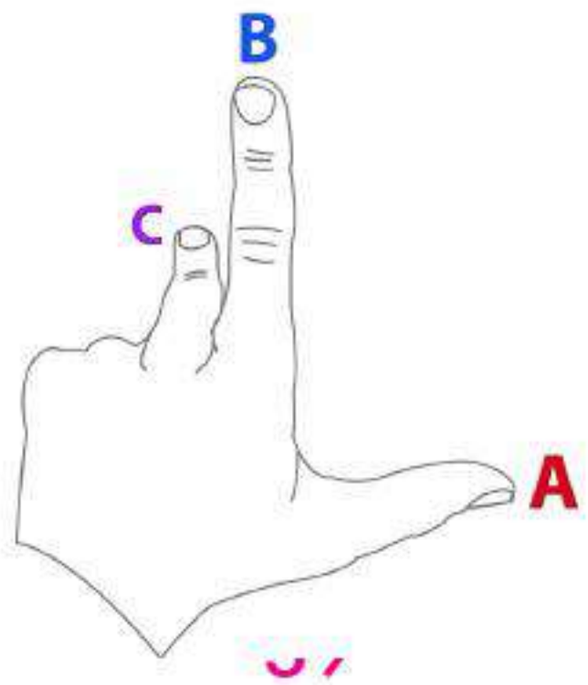
Cross Product

The cross product is a different way of combining 2 vectors. Instead of producing a scaler (a single value) the cross product produces another vector, specifically a vector that is perpendicular to the original two. For example if we took Vector A and Vector B and crossed them then the product would be vector C, one that is perpendicular to A and B. This is mathematically denoted by the caret symbol, the 'up arrow' shown here.



$$A \wedge B = C$$

Since Unity's co-ordinate system is left handed, so is its cross product. Putting your left hand into the same pose as to determine the alignment of the axis you can determine the direction of a cross product vector. In this example your thumb and index finger represent vectors A and B, the known vectors, and your middle finger represents the cross product result C. To look to the cross product in detail we would arrange our vector components A and B like this.



To look to the cross product in detail we would arrange our vector components A and B like this.

$$\begin{pmatrix} Ax \\ Ay \\ Az \end{pmatrix} \wedge \begin{pmatrix} Bx \\ By \\ Bz \end{pmatrix} = \begin{pmatrix} Cx \\ Cy \\ Cz \end{pmatrix}$$

But how does it work mathematically? To work out the cross product, we place our two vectors next to each other and then repeat this below.

$$\begin{pmatrix} Ax \\ Ay \\ Az \end{pmatrix} \begin{pmatrix} Bx \\ By \\ Bz \end{pmatrix} = \begin{pmatrix} \\ \\ \end{pmatrix}$$

$$\begin{pmatrix} Ax \\ Ay \\ Az \end{pmatrix} \begin{pmatrix} Bx \\ By \\ Bz \end{pmatrix}$$

From here we can combine the components one at a time to find the cross product. First we multiply Ay and Bz. We then subtract Az multiplied by By. This gives us the X component of the cross product. Next we do the same with Az and Bx, and Ax and Bz. To give us the Y component.

$$\begin{pmatrix} Ax \\ Ay \\ Az \end{pmatrix} \begin{pmatrix} Bx \\ By \\ Bz \end{pmatrix} = \begin{pmatrix} Ay*Bz - Az*By \\ Az*Bx - Ax*Bz \\ \end{pmatrix} = \begin{pmatrix} Cx \\ Cy \\ Cz \end{pmatrix}$$

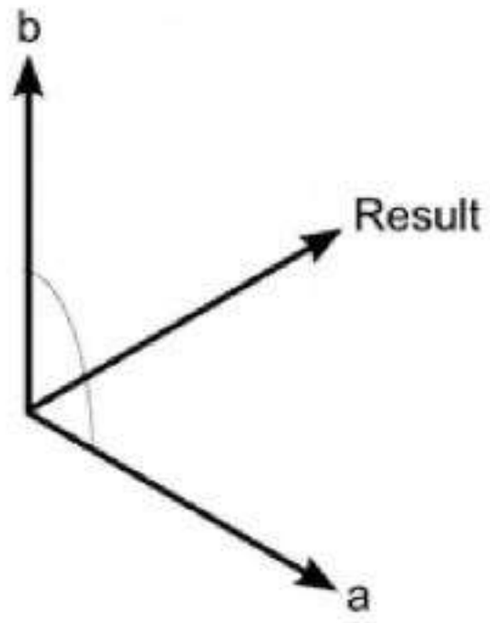
$$\begin{pmatrix} Ax \\ Ay \\ Az \end{pmatrix} \begin{pmatrix} Bx \\ By \\ Bz \end{pmatrix}$$

Finally we do the same again with Ax and By, and Ay and Bx to get the Z component. The good news is that whilst this is a detailed calculation, Unity has another helper function to perform this for you.

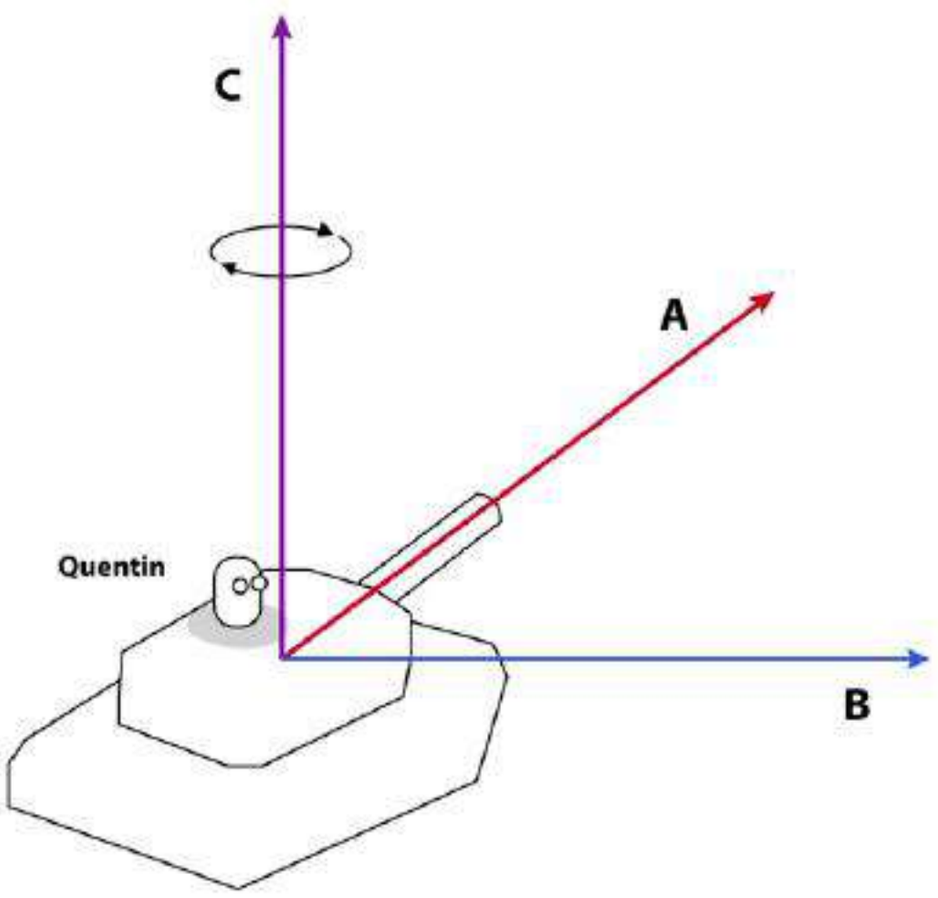
Vector3.Cross(VectorA, Vector B)

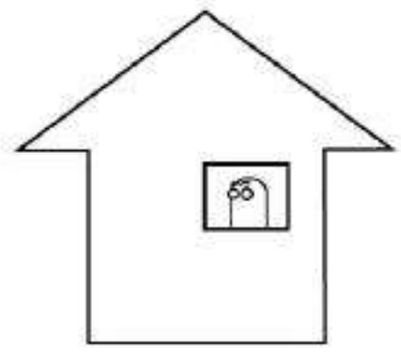
Public static function Cross

The cross product of two vectors results in a third vector which is perpendicular to the two input vectors. The result's magnitude is equal to the magnitudes of the two inputs multiplied together and then multiplied by the sine of the angle between the inputs. You can determine the direction of the result vector using the "left hand rule".



One example of using the cross product is finding the axis around which to apply torque in order to rotate a tanks turret. Given that you have the direction that the turret is currently facing, and the direction that it needs to face, you would cross these two vectors to find the axis around which to apply the rotational torque.





https://unity3d.com/learn/tutorials/modules/beginner/scripting/enabling-disabling-components

11

Enabling and Disabling Components

To enable or disable a component in Unity simply use the Enabled flag. In this example we have a reference to a light called myLight and in Start we're setting this variable using the GetComponent function to the light component attached to our object.

```
using UnityEngine;
using System.Collections;

public class EnableComponents : MonoBehaviour
{
    private Light myLight;


    void Start ()
    {
        myLight = GetComponent<Light>();
    }


    void Update ()
    {
        if(Input.GetKeyUp(KeyCode.Space))
        {
            myLight.enabled = false
        }
    }
}
```

In Update I am waiting for the key press space and then I'm setting the Enabled flag on myLight to false. I could also set this to true but it would be even more efficient to make it a toggle. So I'll set it to the inverse of myLight enabled. So by using the exclamation mark not keyword I'm effectively saying set this to what ever it's not at the moment.

```
void Update ()
{
    if(Input.GetKeyUp(KeyCode.Space))
    {
        myLight.enabled = !myLight.enabled;
    }
}
```

When you press play you can see that the space bar acts as a toggle, switching the light component on and off signified by the checking and un checking of the checkbox next to the name of the component in the inspector.

Keep in mind that because the scripts are also components you can use the dot enabled flag to disable these too.

https://unity3d.com/learn/tutorials/modules/beginner/scripting/activating-gameobjects

12

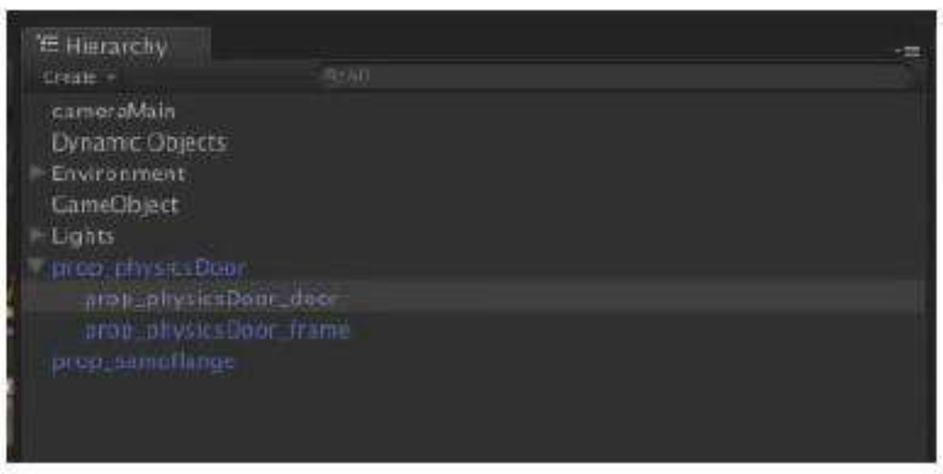
Activating GameObjects

To activate or deactivate a game object via scripting you can use the SetActive function. This will set your object on or off in terms of it being active in the scene. In this example we have a simple Start function that contains gameObject.SetActive(false). And you can see that my game object is currently active.

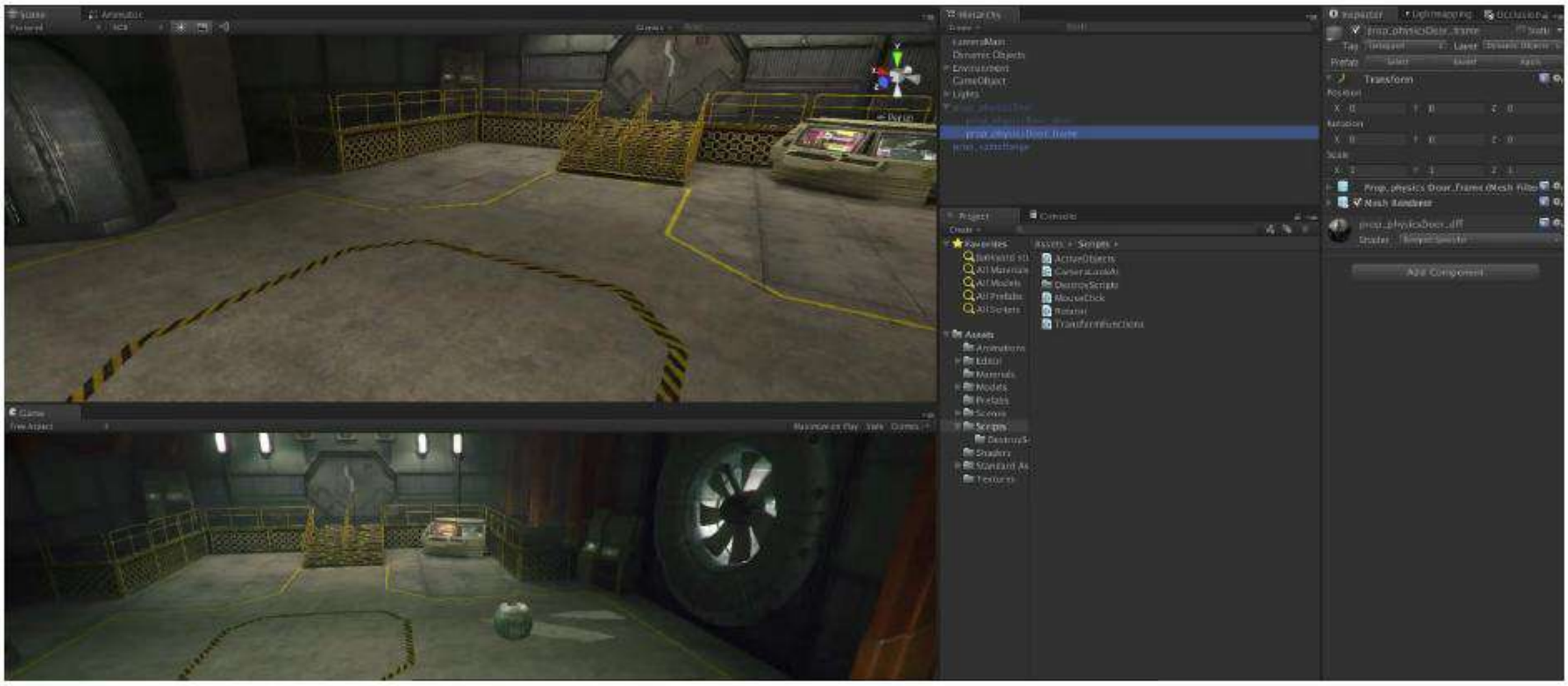
```
using UnityEngine;
using System.Collections;

public class ActiveObjects : MonoBehaviour
{
  void Start ()
  {
      gameObject.SetActive(false);
  }

}
```

If I press play the object is de-activated.

However when working with hierarchies of objects, it's important to know that a parent object can be deactivated, which will stop a child from being active in the scene. However it will remain active within its hierarchy. This is so you can disable individual objects but maintain control over groups of them using the parent object. You can see here that if I select the physics Door and select the child Door object and de-activate it in the inspector it becomes de-selected and the icon is greyed out and I've unchecked the box next to its name.



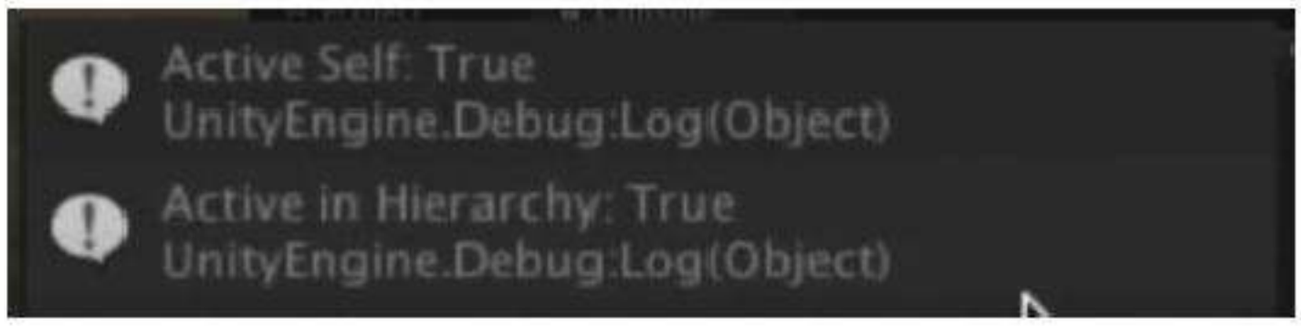
However if I deactivate the parent object instead of the child you can see that I've de-activated all of the child objects but it remains active within the hierarchy, as does the other child object, the, frame.



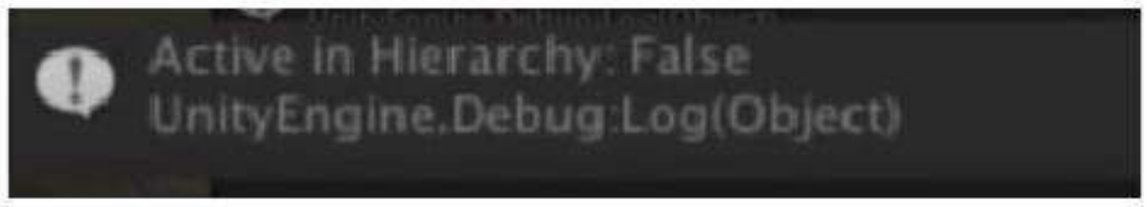
To check if an object is active within the scene or the hierarchy, you can query using the active-Self or using the activeInHierarchy states. In this 'check states' script I have a public variable for a gameobject and I'm checking that game object as to weather its activeSelf or active within the hierarchy.

```
using UnityEngine;
using System.Collections;

public class CheckState : MonoBehaviour
{
    public GameObject myObject;


    void Start ()
    {
        Debug.Log("Active Self: " + myObject.activeSelf);
        Debug.Log("Active in Hierarchy" + myObject.activeInHierarchy);
    }
}
```

We're referring to the physics door we mentioned previously and debug logging the state of those to the console. So when I press play the door itself is active and the hierarchy that it's in is also active.



However, if I deactivate the parent and press play again, the door itself is active but the hierarchy that it's in is not active, because the parent is deactivated and is therefore not active in the scene.



It should also be noted that when a child is inactive because of its parents is inactive using setActive to true will not make that child object active. It can only become active again it can only be made active again when once its parent is made active.

So basically if the parent is deactivated then everything inside (the children) will also be deactivated.

https://unity3d.com/learn/tutorials/modules/beginner/scripting/translate-and-rotate

13

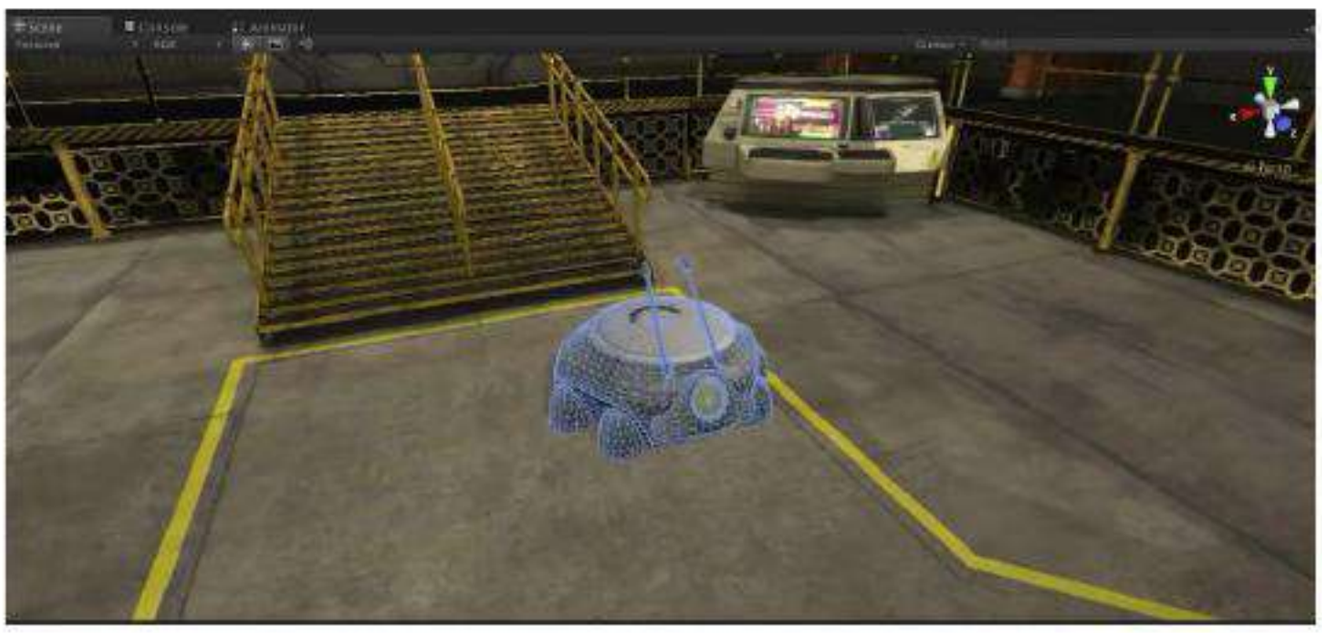
Translate and Rotate

Translate and rotate are two functions that are commonly used to change the position and rotation of a gem object. In this example we begin by looking at translate. You can see that the argument of translate takes a vector3. So this example simply translates down the Z axis, so you can see we've used 0 for X and Y, by 1 every frame because it's inside Update.

transform.Translate(amount in each axis to move by)

```
using UnityEngine;
using System.Collections;

public class TransformFunctions : MonoBehaviour
{

      void update ()
      {
            transform.Translate(new Vector3(0, 0, 1));
      }
}
```

It's applied to our floorbot object. If I press play it moves very quickly because it's updating every frame.



Now what we'd normally with a translate operation is to multiply is by Time.DeltaTime. This means that it will be moved in meters per second rather than meters per frame.

```
      void update ()
      {
            transform.Translate(new Vector3(0, 0, 1) * Time.DeltaTime);
      }
}
```

Then instead of specifically saying Vector3(0, 0, 1) we can use Vector3.forward as it's a shortcut to this. And then we can multiply by another value which we can establish as a separate variable.

```
using UnityEngine;
using System.Collections;

public class TransformFunctions : MonoBehaviour
{

	public float moveSpeed = 10f;

	void update ()
	{
		transform.Translate(Vector3.forward * moveSpeed * Time.DeltaTime);

	}
}
```

That way we can control it by using the variable inside the inspector. But what if we don't want this to happen every frame and we want it to be based on a key press? Meaning that it only occurs when I press the up arrow?

Input.GetKey

Returns true while the user holds down the key identified by name.

```
void update ()
{
	if(Input.GetKey(KeyCode.UpArrow))
	transform.Translate(Vector3.forward * moveSpeed * Time.DeltaTime);

}
```

Then I could do the exact same checking for the down arrow. This time I've used a negative value of Vector3.forward in order to move backwards when I move down.

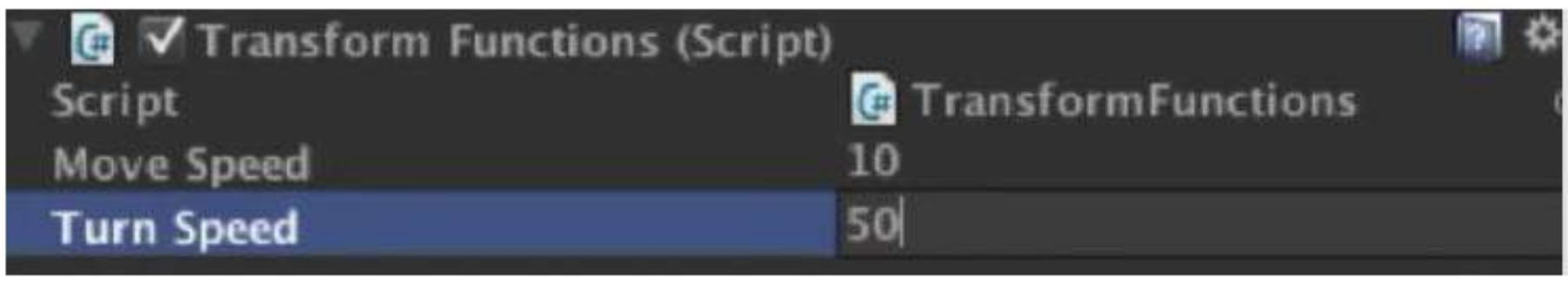
```
void update ()
{
	if(Input.GetKey(KeyCode.UpArrow))
	transform.Translate(Vector3.forward * moveSpeed * Time.DeltaTime);

	if(Input.GetKey(KeyCode.DownArrow))
	transform.Translate(-Vector3.forward * moveSpeed * Time.DeltaTime);

}
```

Transorm.rotate works in a very similar way. Again taking in a Vector3 into its argument. This time we're using the Vector3 shortcut Vector3.up. This represents the axis around which to turn.

transform.Rotate(axis around which to rotate, amount to rotate by);

```
      using UnityEngine;
using System.Collections;

public class TransformFunctions : MonoBehaviour
{

      public float moveSpeed = 10f;
      public float turnSpeed = 50f;

      void update ()
      {
            if(Input.GetKey(KeyCode.UpArrow))
            transform.Translate(Vector3.forward * moveSpeed * Time.DeltaTime);

            if(Input.GetKey(KeyCode.DownArrow))
            transform.Translate(-Vector3.forward * moveSpeed * Time.DeltaTime);

            if(Input.GetKey(KeyCode.LeftArrow))
            transform.Rotate(Vector3.up * -turnSpeed * Time.DeltaTime);

            if(Input.GetKey(KeyCode.RightArrow))
            transform.Rotate(Vector3.up * turnSpeed * Time.DeltaTime);

      }
}
```

We'll add in our variable called turnSpeed at 50f. When I press play I can move the object around and rotate it using the left and right arrows. Because I set these are public variables I can adjust them without having to go back and edit the scripts.



It should be noted that these functions work on the local axis rather than the world axis. So where I'm using Vector3.forward or Vector3.up - this is relative to the axis of the object it's applied to. It's also notable that if you want move an object with a collider, so something that's going to be interactive with physics then you should not use translate or rotate. You should use the physics functions instead. The only time you would want to effect transform is if you had a rigid body that is kinematic.

https://unity3d.com/learn/tutorials/modules/beginner/scripting/look-at

14

Look At

LookAt can be used to make a game object's forward direction point at another transform in the world. In this example our samoflange prop is falling and bouncing off of our hoverpad. Our camera is pointing at the hover pad.

But what if we want to make it look at the falling object? We can use the LookAt function inside of update in order to achieve this. In this script you can see that we make a reference to the object that we wish to look for. A variable called 'target' of type transform. We then use the transform.LookAt function to tell our objects to look at the target. Now we'll simply apply this to our camera and drag on the samoflange object as the object to track. Now when I press play our camera is constantly facing the object.

15

Linear Interpolation

When making games it can sometimes be useful to linearly interpolate between two values. This is done with a function called Lerp. Linearly interpolating is finding a value that is some percentage between two given values. For example, we could linearly interpolate between the numbers 3 and 5 by 50% to get the number 4. This is because 4 is 50% of the way between 3 and 5.

In Unity there are several Lerp functions that can be used for different types. For the example we have just used, the equivalent would be the Mathf.Lerp function and would look like this:

```
// In this case, result = 4
float result = Mathf.Lerp (3f, 5f, 0.5f);
```

The Mathf.Lerp function takes 3 float parameters: one representing the value to interpolate from; another representing the value to interpolate to and a final float representing how far to interpolate. In this case, the interpolation value is 0.5, which means 50%. If it was 0, the function would return the 'from' value and if it was 1 the function would return the 'to' value.

Other examples of Lerp functions include Color.Lerp and Vector3.Lerp. These work in exactly the same way as Mathf.Lerp but the 'from' and 'to' values are of type Color and Vector3 respectively. The third parameter in each case is still a float representing how much to interpolate. The result of these functions is finding a colour that is some blend of two given colours and a vector that is some percentage of the way between the two given vectors.

Let's look at another example:

```
Vector3 from = new Vector3 (1f, 2f, 3f);
Vector3 to = new Vector3 (5f, 6f, 7f);

// Here result = (4, 5, 6)
Vector3 result = Vector3.Lerp (from, to, 0.75f);
```

In this case the result is (4, 5, 6) because 4 is 75% of the way between 1 and 5; 5 is 75% of the way between 2 and 6 and 6 is 75% of the way between 3 and 7.

The same principle is applied when using Color.Lerp. In the Color struct, colours are represented by 4 floats representing red, blue, green and alpha. When using Lerp, these floats are interpolated just as with Mathf.Lerp and Vector3.Lerp.

Under some circumstances Lerp functions can be used to smooth a value over time. Consider the following piece of code:

```
void Update ()
{
    light.intensity = Mathf.Lerp(light.intensity, 8f, 0.5f);
}
```

If the intensity of the light starts off at 0 then after the first update it will be set to 4. The next frame it will be set to 6, then to 7, then to 7.5 and so on. Thus over several frames, the lights intensity will tend towards 8 but the rate of it's change will slow as it approaches its target. Note that this happens over the course of several frames. If we wanted this to not be frame rate dependent then we could use the following code:

```
void Update ()
{
    light.intensity = Mathf.Lerp(light.intensity, 8f, 0.5f * Time.deltaTime);
}
```

This would mean the change to intensity would happen per second instead of per frame.

Please note that when smoothing a value it is often best to use the SmoothDamp function. Only use Lerp for smoothing if you are sure of the effect you want.

https://unity3d.com/learn/tutorials/modules/beginner/scripting/destroy

16

Destroy

The destroy function can be used to remove game objects or components from game objects at runtime. This can also be done with a time delay by using its second argument, a float number.

Destroy(GameObject/ Component,optional delay);

To destroy a game object, for example, we could simply refer to the gameobject that the script is attatched to. In this example our prop samoflange object has the script attatched to it.

```
using UnityEngine;
using System.Collections;

public class DestroyBasic : MonoBehaviour
{
    public GameObject other;
    void Update ()
    {
        if(Input.GetKey(KeyCode.Space))
        {
            Destroy(other);
        }
    }
}
```

And when the space bar is pressed the game object will be destroyed.

Now a problem could be that you may be using this script for various purposes so it doesn't make sense to destroy the object, as the script component will be removed too, as it's attached to the object. So instead you'll likely use a reference to another object.

Here we've set up a public variable called Other to refer to another object. Then in the inspector we'll drag on another object to use, such as the door and press play the door gets removed from the scene.

You can also use the destroy command to destroy to remove components, rather than entire game objects. For this we use the GetComponent function inside the destroy parenthesis to refer to a component. In this example I'll destroy the mesh renderer component so that the object is no longer visibly rendered.

```
using UnityEngine;
using System.Collections;

public class DestroyComponent : MonoBehaviour
{
    void Update ()
    {
        if(Input.GetKey(KeyCode.Space))
        {
            Destroy(GetComponent<MeshRenderer>());
        }
    }
}
```

This make the object still in the game and will have all of its other components except the mesh renderer that we removed. All of these examples can include a number as the second argument in order to create a time delay.

For example if I had 3 as a floating point number, as my second argument.

```
using UnityEngine;
using System.Collections;

public class DestroyComponent : MonoBehaviour
{
    void Update ()
    {
        if(Input.GetKey(KeyCode.Space))
        {
            Destroy(gameObject, 3f);
        }
    }
}
```

When I press play there will be a 3 second delay, then my object will be destroyed. This works the same for destroying components.




https://unity3d.com/learn/tutorials/modules/beginner/scripting/get-button-and-get-key

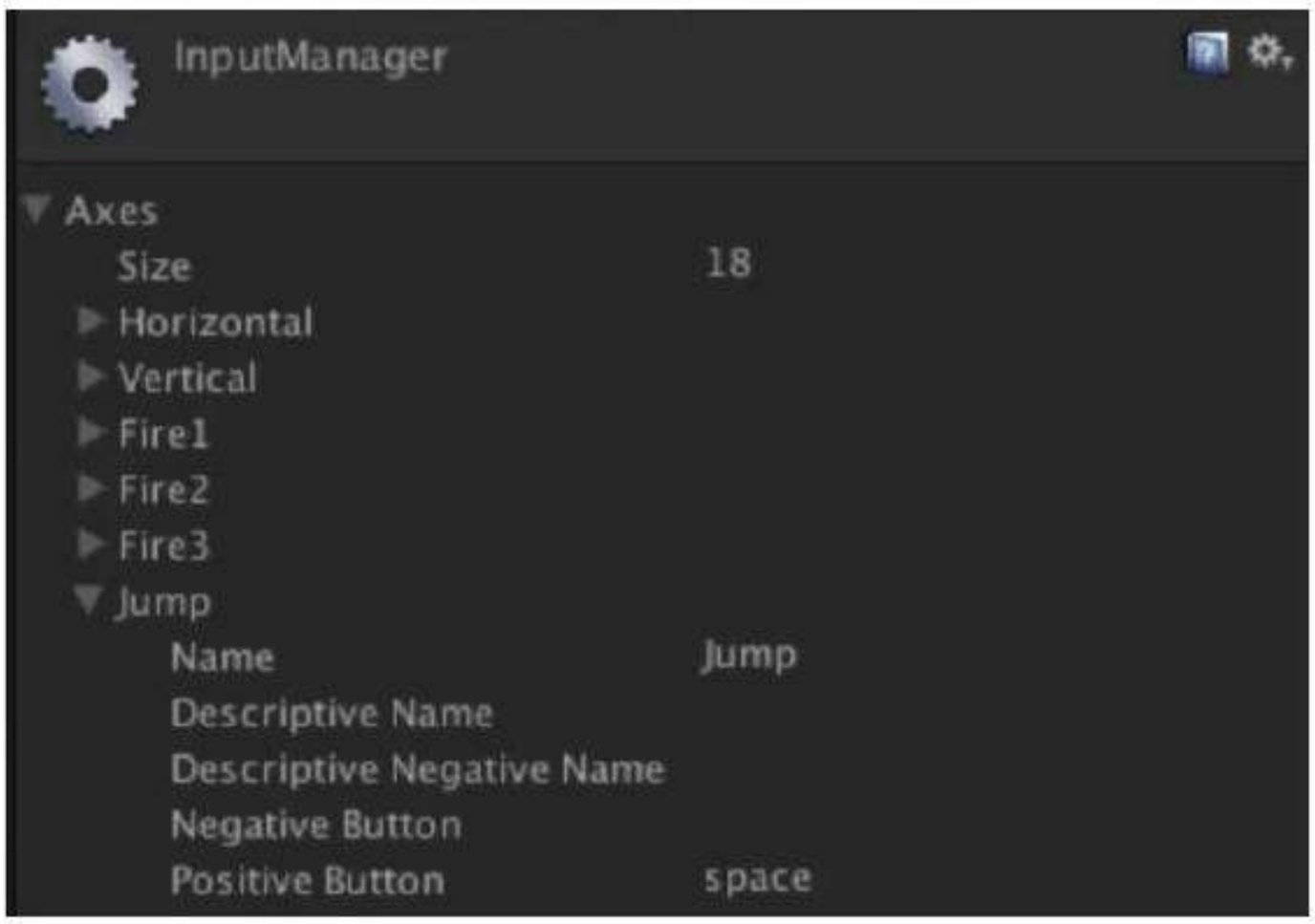
17

GetButton and GetKey

In Unity GetKey and GetButton are ways of receiving input from keys or joy stick buttons via Unity's input class. The core differences between the two is that GetKey specifically names keys for you using key codes. For example, the space bar is represented by KeyCode.Space.

```
Input.GetKeyDown(KeyCode.Space);
```

This works just fine for keyboard but it's recommended that you use GetButton instead and specify your own controls. The input manager gives you the ability to name an input and specify a key or button for it and can be accessed by choosing Edit> Project settings > Input.



Then when calling it you can reference a name using a string. For example Jump. This is a default input represented by the space bar, but we could put in a different key or button code in order to change the input that represents Jump. Then when calling this button you can reference the name using the string Jump. To find out what to write in your positive button check the reference in the documentation.

http://docs.unity3d.com/Manual/class-InputManager.html

When using GetKey or GetButton these inputs have three states that return a Boolean. True or False. First is GetKey or GetButton. This will register True or False depending on weather the button is being pressed or not.

GetButton/Down/Up

"Jump"

GetButtonDown: False

GetButton: False

GetButtonUp: False

So currently the button is not being pressed so the GetButton is returning false. When we first press the key it then returns true on the first frame, then as we progress through the frames, holding down the button GetButtonDown returns to false. GetButton still equals true so we can check if the button is being held.

GetButtonDown: True

GetButton: True

GetButtonUp: False

GetButtonDown: False

GetButton: True

GetButtonUp: False

Then we release the button GetButtonUp shows true but only again on the first frame

GetButtonDown: False

GetButton: False

GetButtonUp: True

GetButtonDown: False

GetButton: False

GetButtonUp: False

As we continue, all values return to false. You should note that GetKey behaves exactly the same way, only the code is written a little differently.

In order to check for a button we use the string of the title that was placed in the input manager, Jump.

```
void Update ()
{
    bool down = Input.GetButtonDown(“Jump”);
    bool held = Input.GetButton(“Jump”);
    bool up = Input.GetButtonUp(“Jump”);
}
```

But when looking for a very specific key then we can use key codes.

```
void Update ()
{
    bool down = Input.GetKeyDown(KeyCode.Space);
    bool held = Input.GetKey(KeyCode.Space);
    bool up = Input.GetKeyUp(KeyCode.Space);
}
```

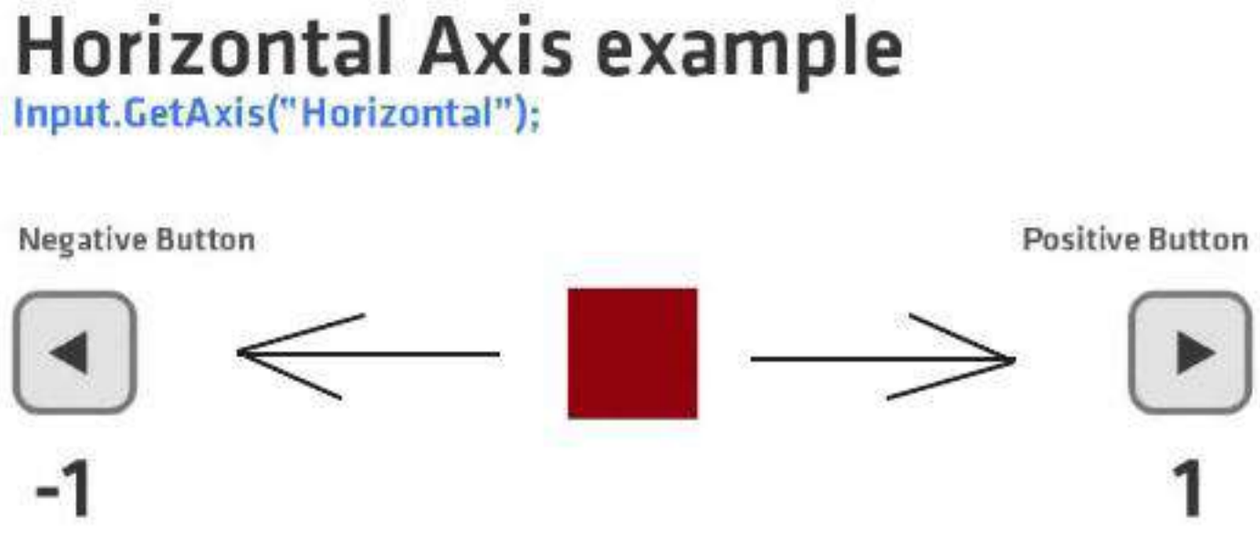
Because this only pertains to the exact key it’s recommended that you use GetButton and specify in the inputs in the input manager.

18

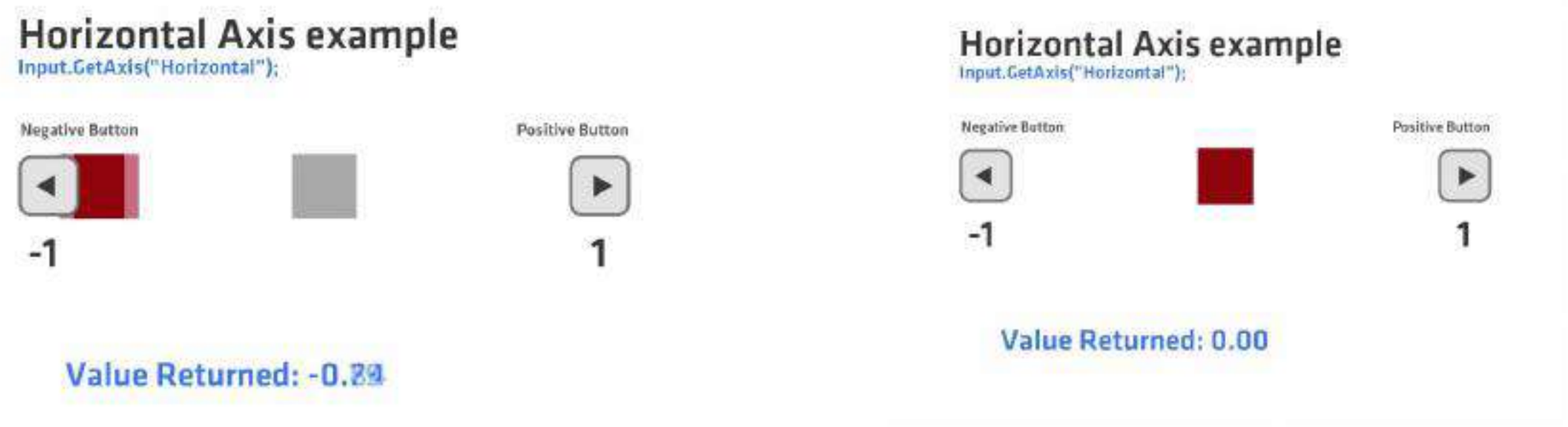
GetAxis

Input.GetAxis works in a similar fashion to GetButton and GetKey but with some fundamental differences. GetButton and GetKey both return a bool, the button is either pressed or not pressed. Whilst GetAxis returns a float value between -1 and positive 1. We set up our axis in the input manager, which can be accessed via Edit> Project settings > Input.

With a button press you'd only consider the positive button value. But with an axis we should consider both positive and negative buttons. As well as gravity, sensitivity, dead and snap.



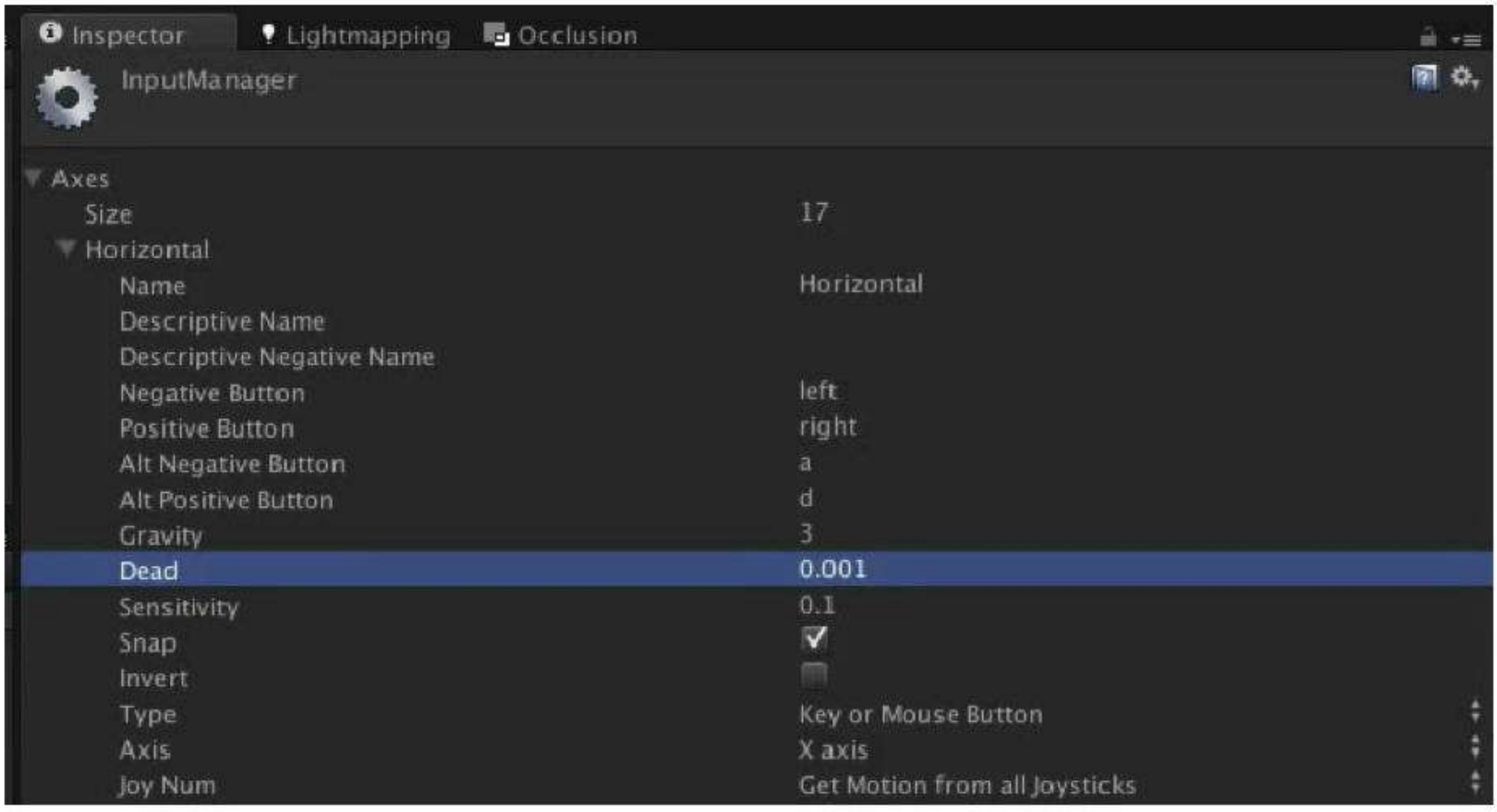
This is the standard behaviour of GetAxis Horixontal. Remember that GetAxis returns a float; as such it is a sliding scale between -1 and 1. The gravity of the axis affects how fast this scale returns to 0 after the button has been released. The higher the gravity the faster the return so if we increase the gravity from 3 to 100 we will see this axis returns to 0 more quickly.



(When you move left then left go the square snaps back really quickly.) Like wise if you lower the gravity lower than the default value of 3 then the axis will return to 0 more slowly.

Sensitivity is the opposite of gravity. And controls how quickly the return value of the input reaches 1 or -1. The higher the number, the more responsive, the lower the number, the more smooth.

If we were using a joy stick to represent our axis then we wouldn't necessarily want to feel an effect from very small amounts of joystick movement. To avoid this we have a dead zone. The higher the dead value is the larger the dead zone and further the joy stick has to be moved in order for GetAxis to return a non 0 value.



The snap option allows you to return 0 if both positive and negative buttons are held. To receive a value from horizontal or vertical axis simply add an instance of Input.GetAxis("Horizontal") or Input.GetAxis("Vertical") to your code as seen here.

```
using UnityEngine;
using System.Collections;

public class DualAxisExample : MonoBehaviour
{
    public float range;
    public GUIText textOutput;


    void Update ()
    {
        float h = Input.GetAxis("Horizontal");
        float v = Input.GetAxis("Vertical");
        float xPos = h * range;
        float yPos = v * range;
    }
}
```

You can also use Input.GetAxisRaw to only return whole numbers and nothing in-between; this is useful for 2D games that need precise controls rather than smoothed values. Note that this does not require the use of gravity of sensitivity.

https://unity3d.com/learn/tutorials/modules/beginner/scripting/on-mouse-down

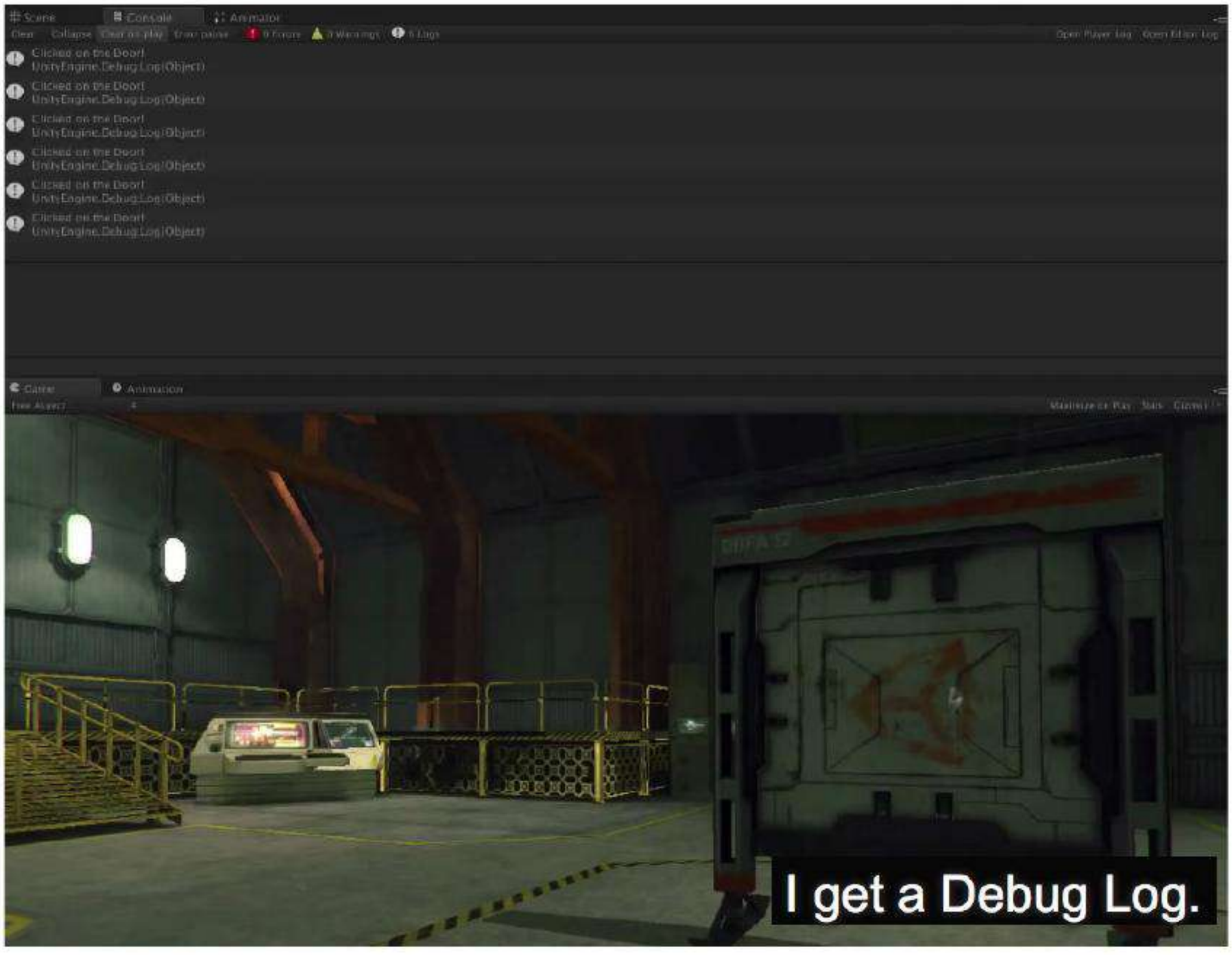
19

OnMouseDown

OnMouseDown and its related functions can detect a click on a collider or a GUI text element. In this example we have a door object that has a collider and a rigid body attached. The script that I've written has an OnMouseDown function and when that object is clicked we will DebugLog meaning that we'll print to the console "Clicked on the door".

```
using UnityEngine;
using System.Collections;

public class MouseClick : MonoBehaviour
{
    void OnMouseDown ()
    {
        Debug.Log ("Clicked on the door!");
    }
}
```

So when I press play, check my console and click the door in the game I get a Debug.Log.



Similarly we can make it do something more exciting. For example adding a force to the door object when it's clicked. So in this example I am adding a force in the inverse direction of its forward, so the door will be fired away from its Z axis. I am also switching on gravity, so it will fall down when it's clicked.

```
using UnityEngine;
using System.Collections;

public class MouseClick : MonoBehaviour
{
    void OnMouseDown ()
    {
        rigidbody.AddForce(-transform.forward * 500f);
         rigidbody.useGravity = true;
    }
}
```

Similarly we can make it do something more exciting. For example adding a force to the door object when it's clicked. So in this example I am adding a force in the inverse direction of its forward, so the door will be fired away.

https://unity3d.com/learn/tutorials/modules/beginner/scripting/getcomponent

20

GetComponent

In Unity a script is considered as a custom component, and often you will need to access other scripts attached to the same game object or even on other game objects. You can access other scripts and components using GetComponent.

UsingOtherComponents.cs* × AnotherScript.cs × YetAnotherScript.cs ×

No selection

```
1  using UnityEngine;
2  using System.Collections;
3
4  public class YetAnotherScript : MonoBehaviour
5  {
6      public int numberOfPlayerDeaths = 3;
7  }
8
```

UsingOtherComponents.cs* × AnotherScript.cs × YetAnotherScript.cs ×

No selection

```
1  using UnityEngine;
2  using System.Collections;
3
4  public class YetAnotherScript : MonoBehaviour
5  {
6      public int numberOfPlayerDeaths = 3;
7  }
8
```

In this example both AnotherScript and YetAnotherScript both have public variables within them. We'd like to be able to affect those variables in our UsingOtherComponents script.

UsingOtherComponents.cs* × AnotherScript.cs × YetAnotherScript.cs ×

UsingOtherComponents › Awake ()

```
1  using UnityEngine;
2  using System.Collections;
3
4  public class UsingOtherComponents : MonoBehaviour
5  {
6      public GameObject otherGameObject;
7
8      private AnotherScript anotherScript;
9      private YetAnotherScript yetAnotherScript;
10
11
12     void Awake ()
13     {
14         anotherScript = GetComponent<AnotherScript>();
15         yetAnotherScript = otherGameObject.GetComponent<YetAnotherScript>();
16     }
17
18
19     void Start ()
20     {
21         Debug.Log("The player's score is " + anotherScript.playerScore);
22         Debug.Log("The player has died " + yetAnotherScript.numberOfPlayerDeaths + " times");
23     }
24 }
```

We have 3 variables here, one to store anotherGameObject, and two to store references to the other scripts. Note that the references to the other scripts are simply variables whose type is the name of the script. This is because what we are actually referencing is an instance of the class defined in the script.

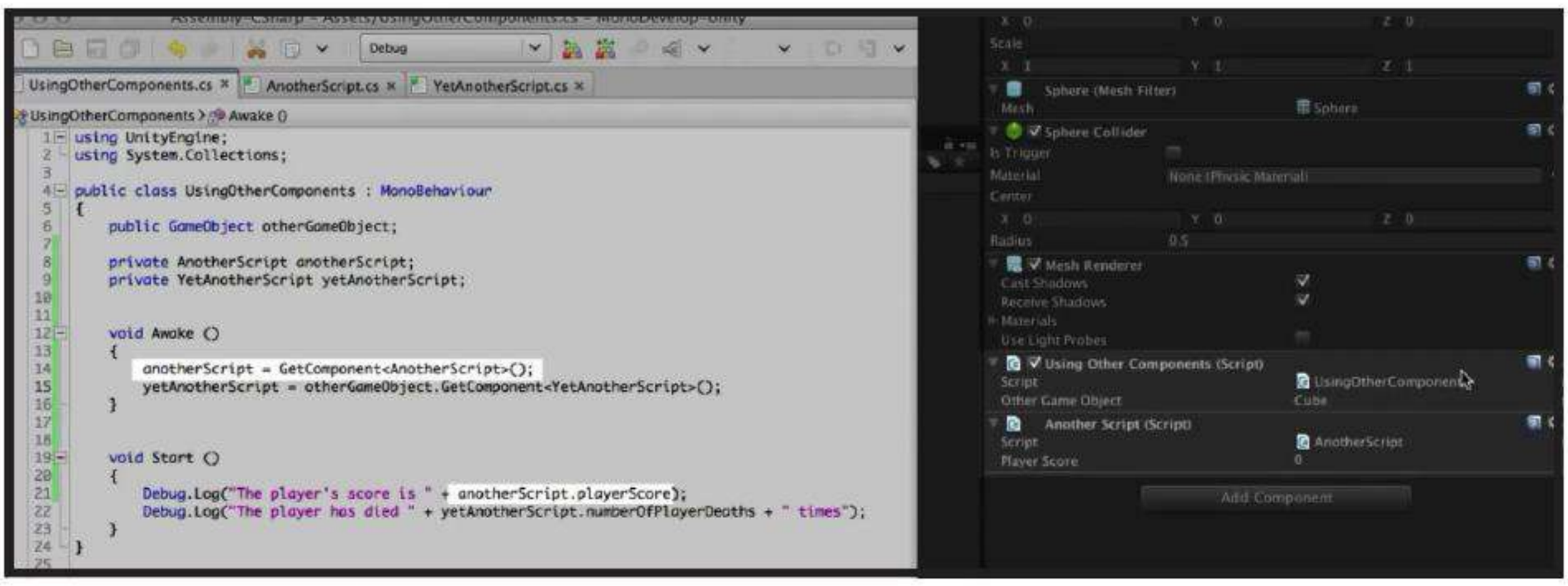
In the Awake function we initialise our variables. The GetComponent function uses a slightly different style of call to what we are used to. We use a pair of angle brackets before the normal brackets. These angle brackets are for taking a type as a parameter; in our case the type will be AnotherScript.

```
GetComponent<Type>();
```

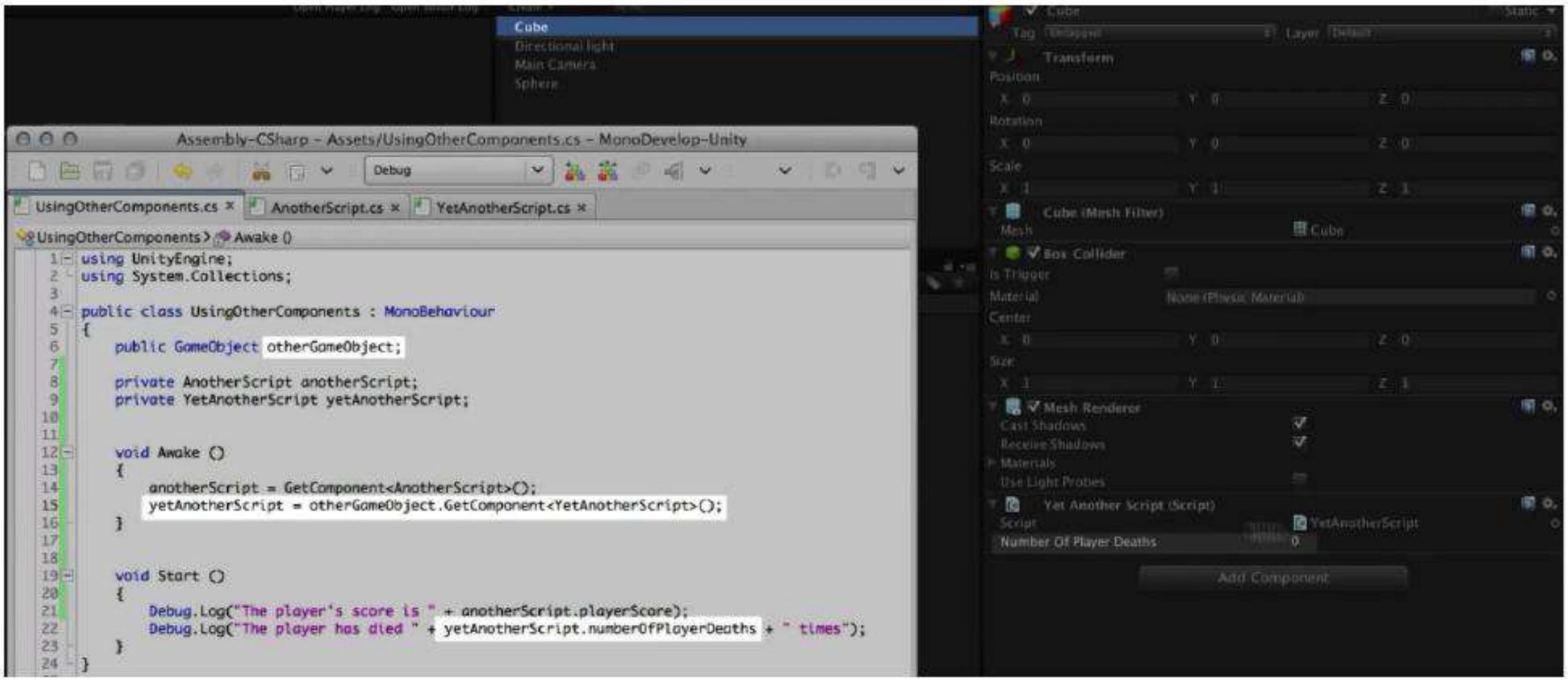
We can also call GetComponent to address components on other game objects that we have references to, like otherGameObject.

```
GetObjectReference.GetComponent<Type>();
```

GetComponent will return a reference to any component of the type specified on the game object it is called upon.



So in this example I'm addressing my anotherScript, which is on the same object as UsingOtherComponents, my main script. So I can say GetComponent and address the player score directly. But if I want to address the number of player deaths I'll use otherGameObject.GetComonent to address the other script on the cube. That way I can affect the number of player deaths directly.



Whilst GetComponent is most commonly used for accessing other scripts, it can also be used to access other components. that are not exposed by the API.

For example the collider that you normally access by typing Collider is a non specific collider. A sphere collider has different properties to a box collider for example. And if you want to access these properties in scripts then you can do so by using GetComponent. For example I can address the box collider on the OtherGameObject by saying
boxCol = otherGameObject.GetComponent<BoxCollider>

```
using UnityEngine;
using System.Collections;

public class UsingOtherComponents : MonoBehaviour
{
    public GameObject otherGameObject;

    private AnotherScript anotherScript;
    private YetAnotherScript yetAnotherScript;
    private BoxCollider boxCol;

    void Awake ()
    {
        anotherScript = GetComponent<AnotherScript>();
        yetAnotherScript = otherGameObject.GetComponent<YetAnotherScript>();
        boxCol = otherGameObject.GetComponent<BoxCollider>();
    }
```

I can then use my boxCol variable to address any of the parameters on that component on the OtherGameObject. So for example I can address the size and set it to 3 when the scene starts.

Some thing to note about GetComponent is that it's expensive in terms of processing power and should be called as infrequently as possible. It's often good practice to call it in the Awake or Start functions, or only once when it's first needed.

https://unity3d.com/learn/tutorials/modules/beginner/scripting/data-types

21

Data types

When working with code you will inevitably need to work with many types of variables. Ultimately all variables have what is called a data type. The two primary data types are value types and reference types. Variables like integers, floats, doubles, Booleans and characters are all value types. Additionally there are complex variable types known as Strucks. Strucks are value data types that simply contain one or more other variables. The two most common Strucks in Unity are Vector3 and Quaternion. The list of reference types is much simpler. Basically any variable that is an object to a class is a reference type too.

Value

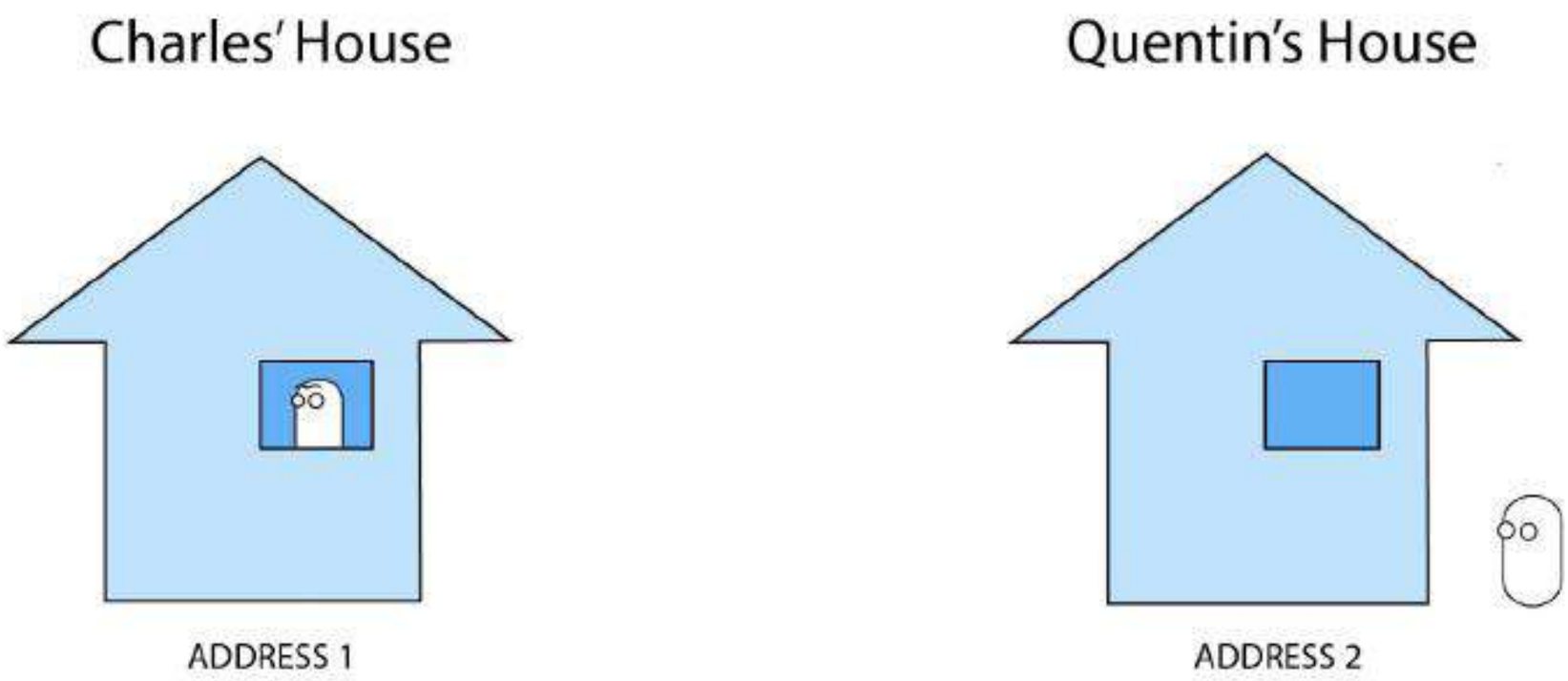
- int
- float
- double
- bool
- char
- Structs
  - Vector3
  - Quaternion

The two most common classes and thus the two most common reference types in Unity are transform and game object. You might be wondering what the difference is between value and reference types. Simply put value type variables actually contain some value, while reference type variables contain a memory address where the value is stored. The result is that if the value type changes only that specific variables is affected. If a reference type changes however, all variable that contain that memory address will also be affected

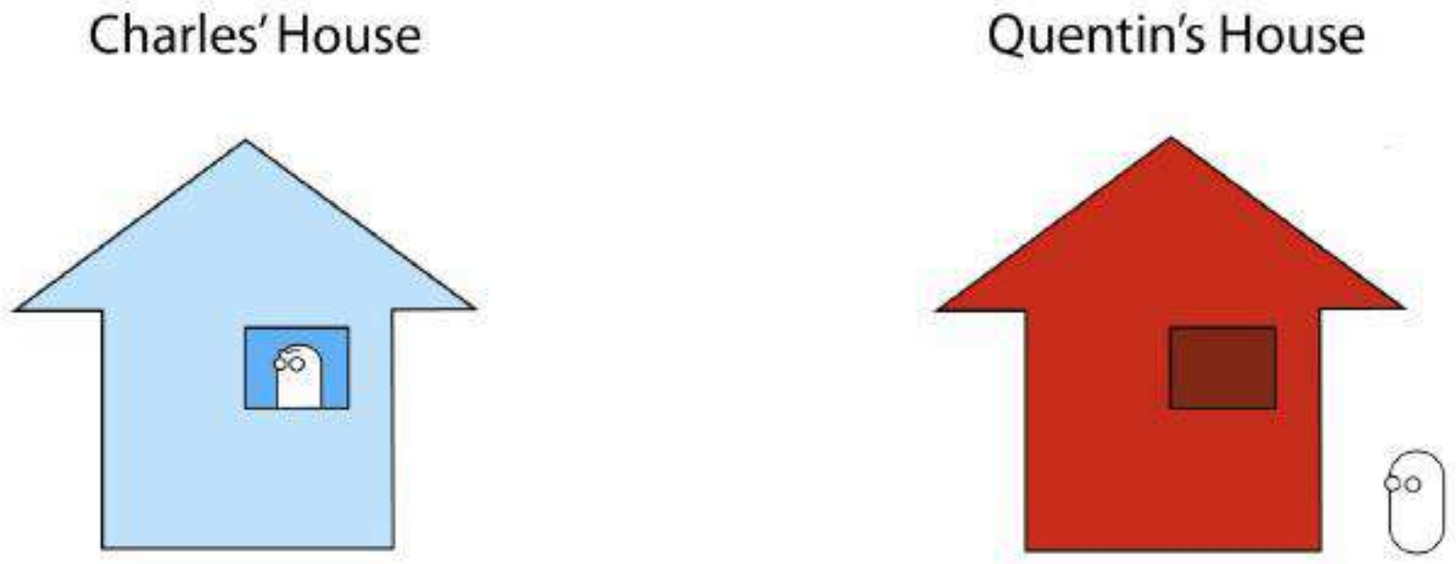
Reference

- Classes
  - Transform
  - GameObject

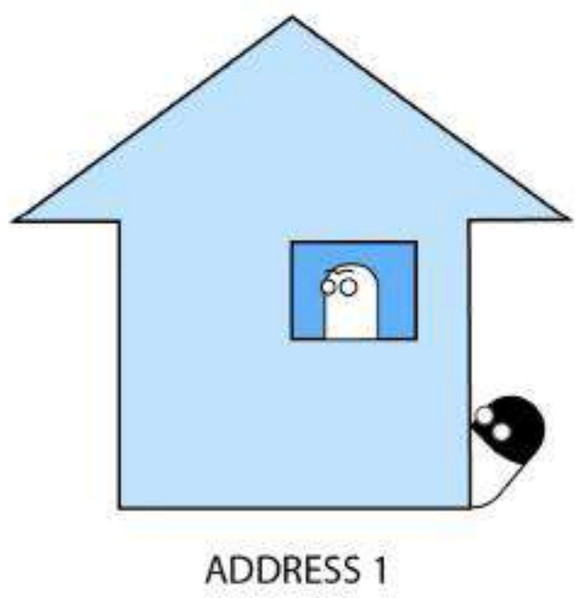
As an analogy consider this. Charles has a nice blue house full of his favourite things. Clinton sees his house online and is jealous. He then makes an exact copy and fills it with duplicates of everything that Charles owns.



This is like assignment for value types, you make a copy of the variable. Quentin doesn't own Charles' house and nothing he changes in his house will affect the original, not even painting it red.



If Quentin had Charles address he wouldn't need to copy the house. He could simply visit any time he likes to see its contents instead. This is like assignment for reference types.



You make a note of the memory address at the value you need and return to that address when you need to know the value of the variable.

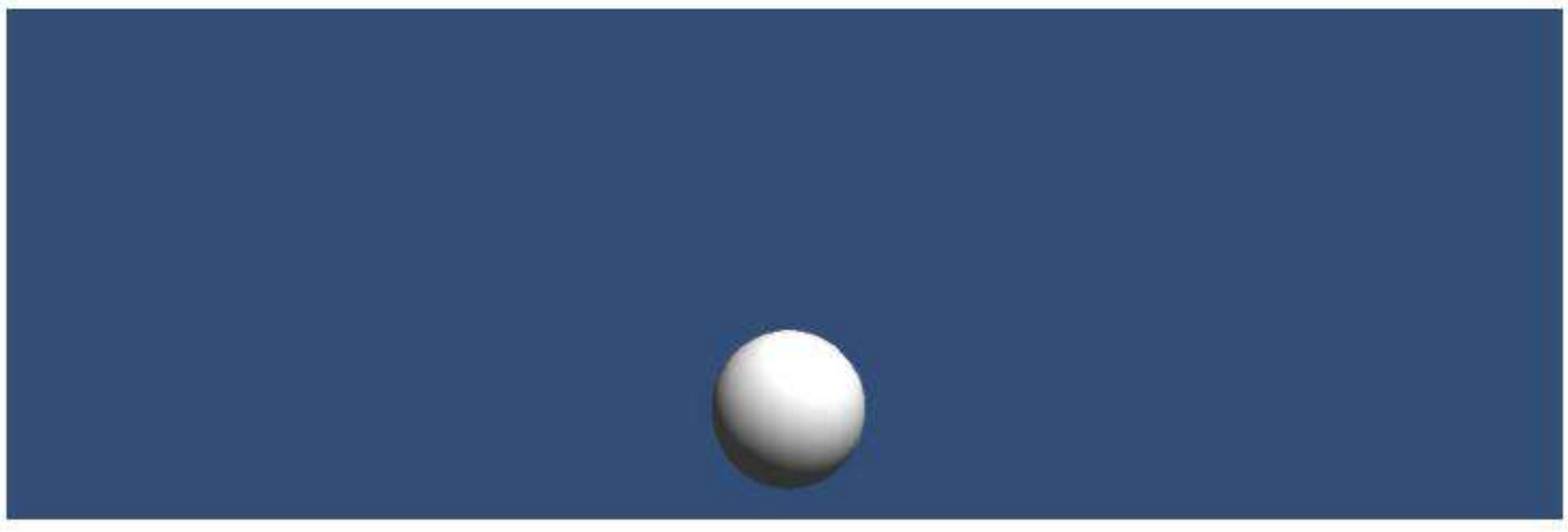
Let's take a look at this in action. We have here a scene with a sphere in it. The sphere has a script attached to it named DataType script. Inside this script we first create a Vector3 variable named pos and assigned to it the value that transforms that position. We then set pos equal to a new Vector3 with the value of 0, 2, 0. Since we know that pos is a Vector3 structs and that structs are value types, we know that only pos is affected by this line of code. The transform position is unaffected.

DatatypeScript.cs ×

No selection

```
1  using UnityEngine;
2  using System.Collections;
3
4  public class DatatypeScript : MonoBehaviour
5  {
6      void Start ()
7      {
8          Vector3 pos = transform.position;
9          pos = new Vector3(0, 2, 0);
10     }
11 }
12
```

If we go back in the Unity we can run our scene and confirm that the sphere does not move. Value type contains their own copy of data and changing them only affects that specific variable.



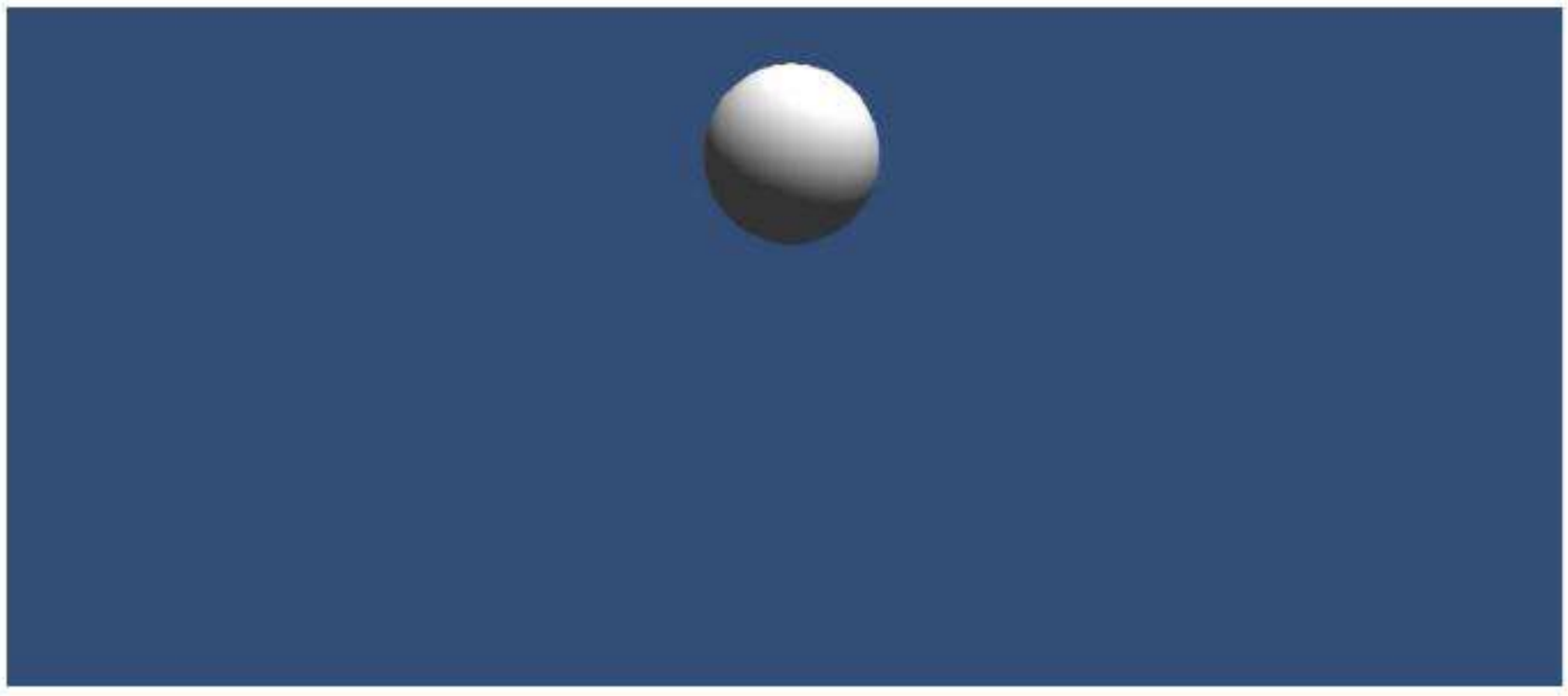
Lets try another approach, let's start by creating a transform variable named tran, and setting it equal to the objects transform. Then, lets set trans position to a new Vector3 with the value 0, 2, 0. Since transform is a class we know that tran is a reference type. Further more since we assigned the objects transform to the variable tran using the assignment operator (=) we know that they point to the same memory address. The result is that changing one changes the other.

DatatypeScript.cs* ×

DatatypeScript ▸ Start ()

```
1  using UnityEngine;
2  using System.Collections;
3
4  public class DatatypeScript : MonoBehaviour
5  {
6      void Start ()
7      {
8          Transform tran = transform;
9          tran.position = new Vector3(0, 2, 0);
10     }
11 }
12
```

When we run our scene we should notice immediately that the sphere has moved upwards even though we never directly changed the spheres transform.



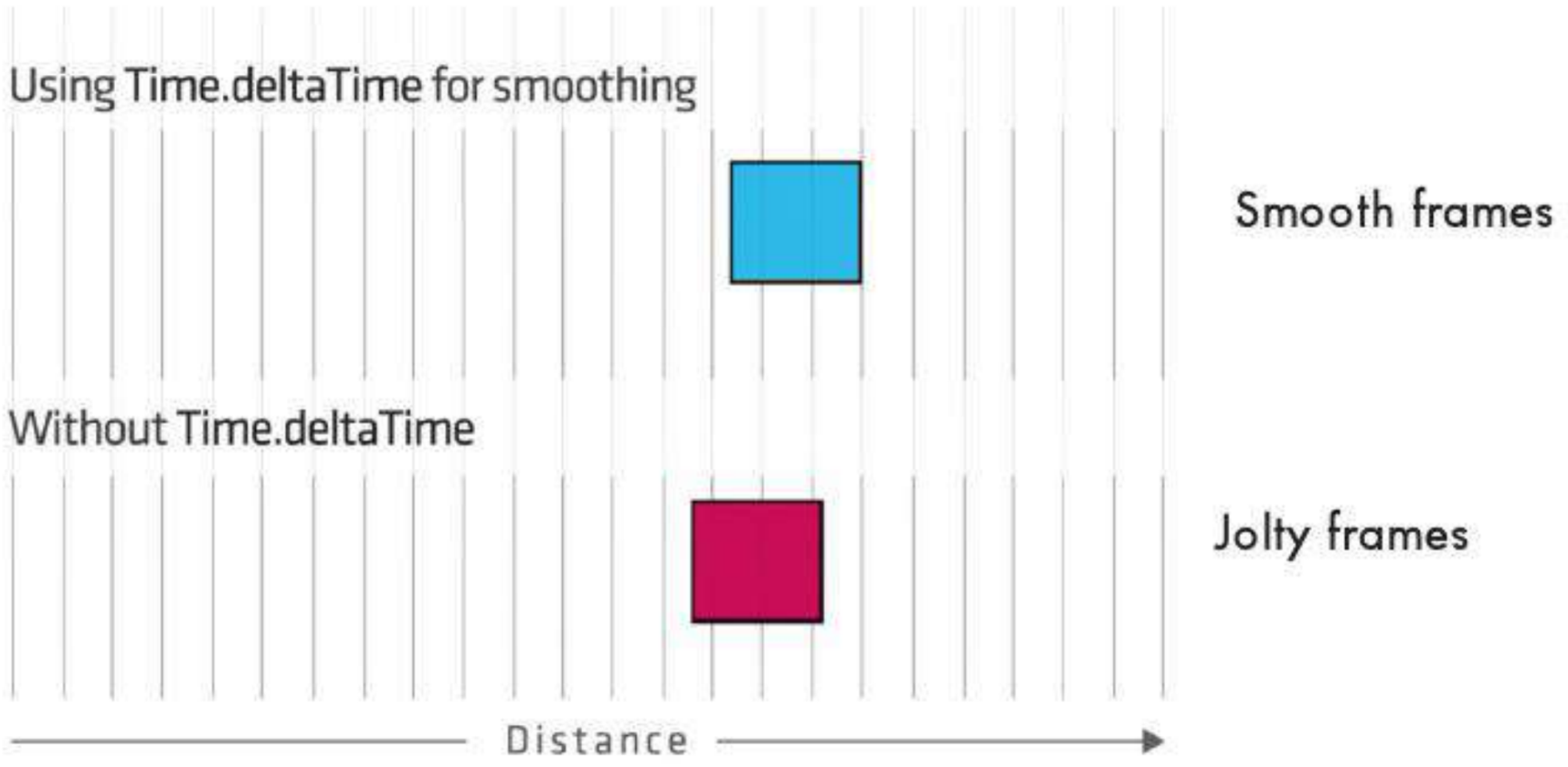
We were able to indirectly change it by using a reference variable.

https://unity3d.com/learn/tutorials/modules/beginner/scripting/delta-time

22

Delta Time

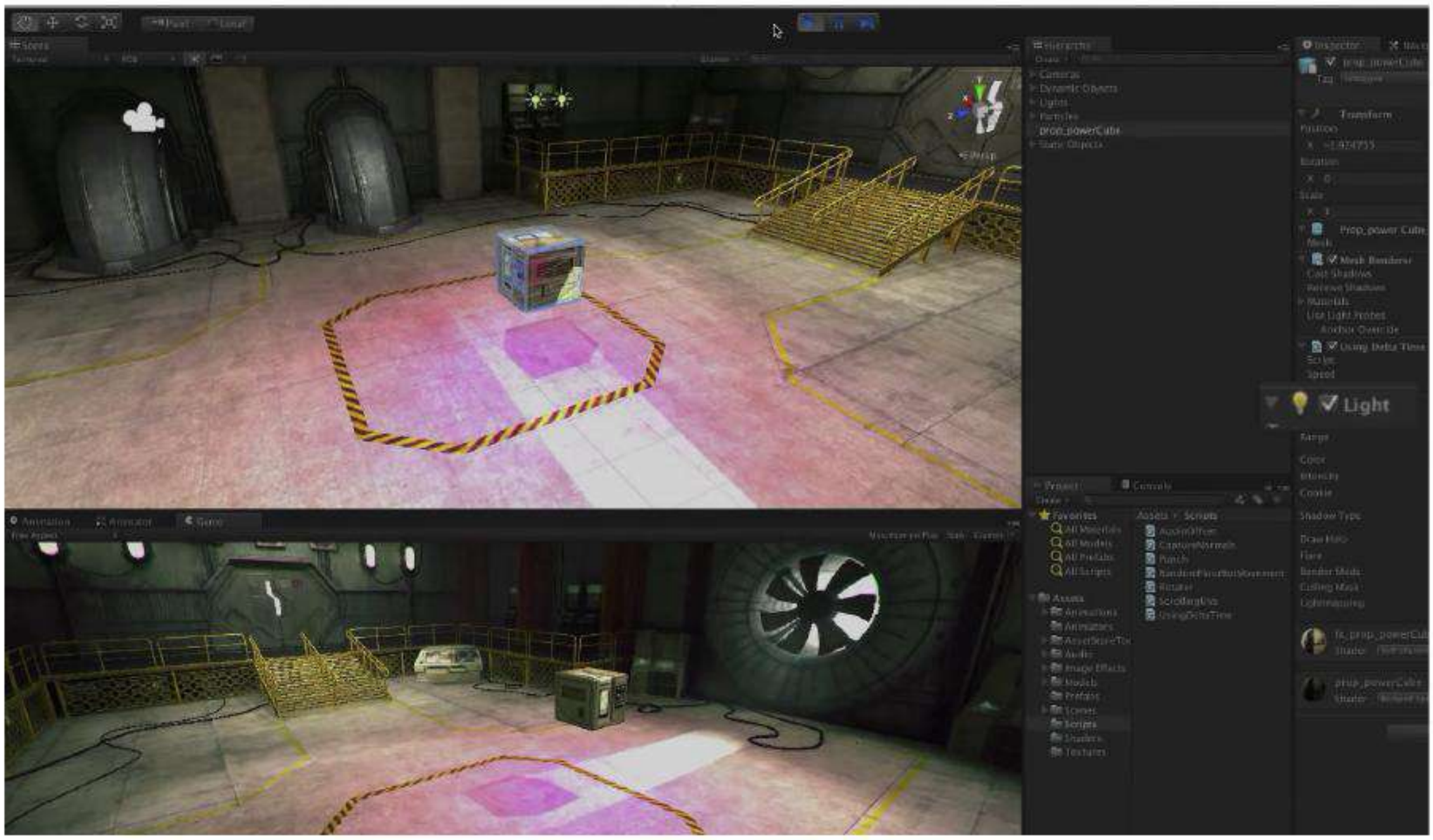
The term Delta time means a difference between two values. The deltaTime property of the time class is essentially the time between each update or fixed update unction call. This can be used to smooth out values used for movement and other incremental calculations. The time between frames is not constant. So, if for example, you're moving an object each frame by a fixed amount then the overall effect might not be smooth. This is because the amount of time it takes to complete a frame will vary, despite the distance of movement remaining constant. If you modify the change using time.DeltaTime then any frames that take a longer time will see a bigger change.



The overall effect of this is that over a period of time the change will appear smooth. When the scene starts our count down variable is reduced by the amount of time, in seconds, that it takes to complete each frame. And when this reaches 0 we are enabling a light component on our power cube game object.

UsingDeltaTime.cs

```
UsingDeltaTime > No selection
1  using UnityEngine;
2  using System.Collections;
3
4  public class UsingDeltaTime : MonoBehaviour
5  {
6      public float speed = 8f;
7      public float countdown = 3.0f;
8
9
10     void Update ()
11     {
12         countdown -= Time.deltaTime;
13         if(countdown <= 0.0f)
14             light.enabled = true;
15
16         if(Input.GetKey(KeyCode.RightArrow))
17             transform.position += new Vector3(speed * Time.deltaTime, 0.0f, 0.0f);
18     }
19 }
```



(After 3 seconds the light comes on)

At the same time we can control the cubes movement. Here we are modifying the X-axis when the right arrow key is pressed by adding an amount defined by the variable 'speed' and modified by Time.deltaTime. The movement is smoothed so that the speed remains constant, even if the frame rate varies. The effect of using Time.deltaTime in this way is that it gives us a change in values per second rather than per frame.

```
using UnityEngine;
using System.Collections;

public class UsingDeltaTime : MonoBehaviour
{
    public float speed = 8f;
    public float countdown = 3.0f;


    void Update ()
    {
        countdown -= Time.deltaTime;
        if(countdown <= 0.0f)
            light.enabled = true;

         if(Input.GetKey(KeyCode.RightArrow))
            transform.position += new Vector3(speed * Time.deltaTime, 0.0f, 0.0f);
    }
}
```

(Moves the cube forward.)

https://unity3d.com/learn/tutorials/modules/beginner/scripting/classes

23

Classes

In Unity each script contains a definition for a class. If we were to extend our metaphor where variables are boxes and functions of machines then classes would be the factories that these boxes and machines are in. You may have noticed that scripts from other tutorials have the key word 'class' near the top. Unity automatically puts this line in the script for you when you create a new C# script. This is implicit in JavaScript, meaning that you wont see the definition but Unity still treats the contents of this script as a class. This class shares the same name as the script it is in.

```
public class SingleCharacterScript : MonoBehaviour
```

This is very important, and if you change the name of one you must change the name of the other. It is often best not to change the name of scripts or classes unless you absolutely have too. So try to name your script sensibly when you create it. The class is a container for variables and functions, and provides amongst other things a nice way to group things that work together. They are an organisational tool in something known as Object Orientated Programming, or OOP for short.

OOP

Object Oriented Programming

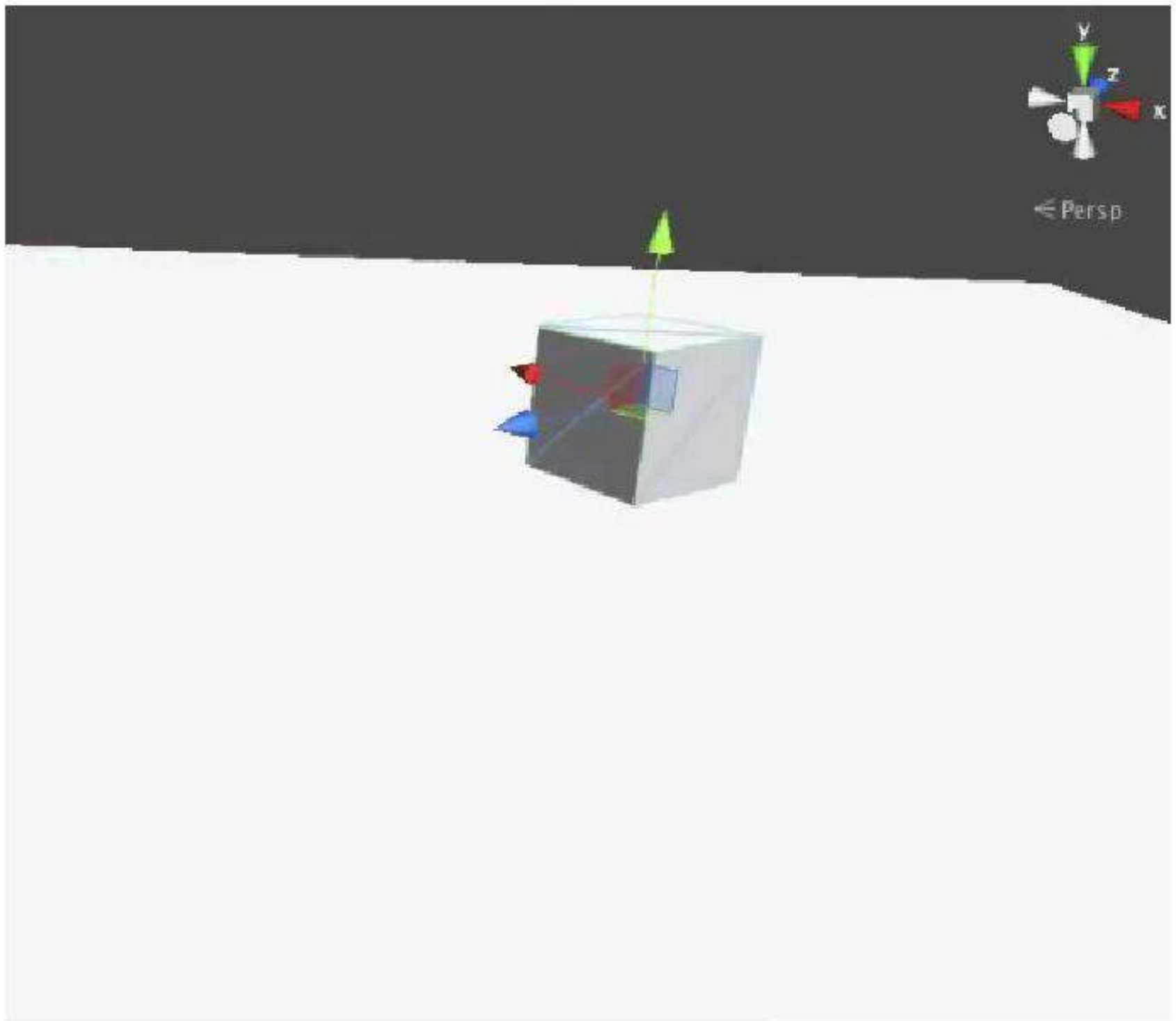
The solution, and core of OOP is to split the script up into multiple scripts, each one taking a single role. Classes should there fore be dedicated to one task. In this example we have a script that handles a number of different tasks, so we should split it up into three shorter scripts to make it easier to manage.

This script for example, handles an inventory, movement and shooting. When attached to a cube we can drive it around, we can shoot some objects and we can keep track of objects in out inventory. This is all contained with in a single character script, what we should be doing is organising this into inventory, moving and shooting as three separate classes.

SingleCharacterScript.cs | Inventory.cs | Shooting.cs | MovementControls.cs

SingleCharacterScript) No selection

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class SingleCharacterScript : MonoBehaviour
{
    public class Stuff
    {
        public int bullets;
        public int grenades;
        public int rockets;

        public Stuff(int bul, int gre, int roc)
        {
            bullets = bul;
            grenades = gre;
            rockets = roc;
        }
    }

    public Stuff myStuff = new Stuff(10, 7, 25);
    public float speed;
    public float turnSpeed;
    public Rigidbody bulletPrefab;
    public Transform firePosition;
    public float bulletSpeed;

    void Update ()
    {
        Movement();
        Shoot();
    }

    void Movement ()
    {
        float forwardMovement = Input.GetAxis("Vertical") * speed * Time.deltaTime;
        float turnMovement = Input.GetAxis("Horizontal") * turnSpeed * Time.deltaTime;

        transform.Translate(Vector3.forward * forwardMovement);
        transform.Rotate(Vector3.up * turnMovement);
    }

    void Shoot ()
    {
        if(Input.GetButtonDown("Fire1") && myStuff.bullets > 0)
        {
            Rigidbody bulletInstance = Instantiate(bulletPrefab, firePosition.position, firePosition.rotation) as Rigidbody;
            bulletInstance.AddForce(firePosition.forward * bulletSpeed);
            myStuff.bullets--;
        }
    }
}
```



In another scene, which looks almost identical, the cube has scripts which handle each of these things individually. This makes these scripts easier to manage, easier to read and more efficient to write. Let's take a look at the use of a class.

In this example we have our inventory class, within that we have a subclass called Stuff. We have created the class here and it contains 3 integer variables. One for bullets, one for grenades, and one for rockets. We then create an instance of this class, down here.

```
using UnityEngine;
using System.Collections;

public class Inventory : MonoBehaviour
{
    public class Stuff
    {
        public int bullets;
        public int grenades;
        public int rockets;
        public float fuel;

        public Stuff(int bul, int gre, int roc)
        {
            bullets = bul;
            grenades = gre;
            rockets = roc;
        }

        public Stuff(int bul, float fu)
        {
            bullets = bul;
            fuel = fu;
        }

        // Constructor
        public Stuff ()
        {
            bullets = 1;
            grenades = 1;
            rockets = 1;
        }
    }

    // Creating an Instance (an Object) of the Stuff class
    public Stuff myStuff = new Stuff(50, 5, 5);

    public Stuff myOtherStuff = new Stuff(50, 1.5f);

    void Start()
    {
        Debug.Log(myStuff.bullets);
    }
}
```

We give it a data type, which is effectively the name of the class, and then we give it a name and we say new, and again reuse the name of the class. The brackets on the end of the name of the class imply that a constructor is being used. A constructor is a function that you can use to give default values to the variables in your class, amongst other tasks they can perform.

For example, we might set up a default constructor. Here we have given default values to each of the integer variables inside our class. This means instead of being initialised at 0 they will be given a default value of 1.

```
// Constructor
public Stuff ()
{
    bullets = 1;
    grenades = 1;
    rockets = 1;
}
```

What we can also do is write our own constructor that allows us to set parameters for those variables. So for example, we might say, public stuff, and then inside the brackets create some parameters. Then what we might do is assign our variables to these parameters. This means that when we create an object or an instance of our class (down here) we can now use the brackets to define what those default values will be. So we know if we put in an integer then a comma, an integer then a comma, we know exactly what those variables with in this instance of the class will be.

```
public Stuff(int bul, int gre, int roc)
    {
        bullets = bul;
        grenades = gre;
        rockets = roc;
    }
```

For example we may have 50 bullets, monodevelop can help figure out what you're doing by telling you as you write them. And lets say we've got 5 grenades and we've got 5 rockets.

A few things to note about constructers, first off the name of the constructer is always the name of the class. Constructers never have a return type, not even void. A class can have multiple different constructors, but only one of them gets called when an object is initialised. For example our first constructor set up values for these three integers. But what if we had more values. Let's put in a float for how much fuel we're carrying. Now currently this constructer does not setup a value for that. But we could set up another constructer to do that job for us. And then if we use the parameters setup in this constructer, what we'll see is that this constructer will be used to setup our new object. So if I make one that has, for example the bullets and then a float for the fuel.

```
public Stuff(int bul, float fu)
{
    bullets = bul;
    fuel = fu;
}
```

Now that we have a separate constructor what I can do is create an instance and use a different constructor based upon what parameters I put into that instance. So I can say new Stuff and this time I can say, for example, 50 bullets and 1.5 amount of fuel. Now this instance of the class is going to use this constructor because the parameters match.

```
public Stuff(int bul, float fu)
{
    bullets = bul;
    fuel = fu;
}
```

```
public Stuff myOtherStuff = new Stuff(50, 1.5f);
```

Where as my other instance of the class will use this constructor because those parameters match.

```
public Stuff(int bul, int gre, int roc)
{
    bullets = bul;
    grenades = gre;
    rockets = roc;
}
```

```
public Stuff myStuff = new Stuff(50, 5, 5);
```

So you can do a lot of different things with constructors and you can have multiple different constructors for the same class. Classes are pretty handy for organising data when creating games and it's recommended that you think through how your scripts are going to be laid out, before you go and write one huge class that has loads of different things in it.

https://unity3d.com/learn/tutorials/modules/beginner/scripting/instantiate

24

Instantiate

Instantiate is a function used to make clones of game objects. This is commonly used in the context of cloning a prefab. A prefab is simply a preconfigured object saved in the project's assets. One such example might be firing rockets from a Bazooka. Each of the rockets would need to be instantiated in to the game world so it could be fired. In this example we are using Fire1 to trigger an insatiate function. The most basic form of instantiate simply takes one parameter, the object that we wish to clone. In this example we have created a public variable called rocketPrefab and we're passing this into the instantiate command.

Instantiate(GameObject or Prefab to create an instance or 'clone' of)

UsingInstantiate.cs ×

UsingInstantiate › rocketPrefab

```
1  using UnityEngine;
2  using System.Collections;
3
4  public class UsingInstantiate : MonoBehaviour
5  {
6      public Rigidbody rocketPrefab;
7
8      void Update ()
9      {
10         if(Input.GetButtonDown("Fire1"))
11         {
12             Instantiate(rocketPrefab);
13         }
14     }
15 }
```

How ever this means that the prefab will be instantiated at its default position, which is 0. And in our example that position is the centre of the world, and the centre of where the bazooka is at the moment.

Those rockets appear at the centre of the Bazooka. Instead we want to fire it like a rocket. So it needs to appear to be coming from the barrel of our bazooka. For this we need a slightly different version of instantiate. This version of Instantiate takes three parameters, the object to be instantiated, in our case the rocket prefab, and then a position and rotation to give our new clone of the prefab.

UsingInstantiate.cs ×

UsingInstantiate > Update ()

```
1  using UnityEngine;
2  using System.Collections;
3
4  public class UsingInstantiate : MonoBehaviour
5  {
6      public Rigidbody rocketPrefab;
7      public Transform barrelEnd;
8
9      void Update ()
10     {
11         if(Input.GetButtonDown("Fire1"))
12         {
13             Instantiate(rocketPrefab, barrelEnd.position, barrelEnd.rotation);
14         }
15     }
16 }
```

Instantiate(Prefab to clone, Position, Rotation to use)

For this I have created an empty Game object that's been positioned in front of the bazookas barrel. We are using this by creating a public variable called BarrelEnd and we used this component's position and rotation values as those for the new clone of our rocket prefab. So by dragging and dropping this new empty location to my script in the inspector, I can receive the transform, position and rotation by making use of the barrelEnd variable.

So when I press play and when I press fire the rocket appears in the right place but the grenades just drop.

We need to be able to affect the object that's created by our instantiation, i.e. the clone of the rocket, as standard instantiate returns a type called Object. But in order to fire our rocket we're going to change that type to a rigidbody. To do this we use as RigidBody after the instantiate call and store the returned value in a rigid body variable, which we'll call rocketInstance.

UsingInstantiate.cs*

UsingInstantiate > Update ()

```
using UnityEngine;
using System.Collections;

public class UsingInstantiate : MonoBehaviour
{
    public Rigidbody rocketPrefab;
    public Transform barrelEnd;

    void Update ()
    {
        if(Input.GetButtonDown("Fire1"))
        {
            Rigidbody rocketInstance;
            rocketInstance = Instantiate(rocketPrefab, barrelEnd.position, barrelEnd.rotation) as Rigidbody;
        }
    }
}
```

Type reference;
reference = Instantiate(Prefab to clone, Position, Rotation) as Type;

So here we setup a rigidbody variable called rocketInstance, and we set that variable equal to our instantiate function's return. Then given that this is now a rigidbody we can use our variable rocketInstance and address anything in the rigidbody class, for example, addForce. And again we can make use of the transform barrelEnd to grab the direction of which we want to add a force. And we can multiply by a certain amount.

UsingInstantiate.cs*

UsingInstantiate > Update ()

```
using UnityEngine;
using System.Collections;

public class UsingInstantiate : MonoBehaviour
{
    public Rigidbody rocketPrefab;
    public Transform barrelEnd;

    void Update ()
    {
        if(Input.GetButtonDown("Fire1"))
        {
            Rigidbody rocketInstance;
            rocketInstance = Instantiate(rocketPrefab, barrelEnd.position, barrelEnd.rotation) as Rigidbody;
            rocketInstance.AddForce(barrelEnd.forward * 5000)
        }
    }
}
```

When we press play the rockets are fired off into the distance by adding a force to the clone that I create.

Bear in mind that when you create many clones within a game those things will still exist within your scene. So if doing something like this you may want to consider writing a script that removes them from the world after a certain amount of time.

We have a rocket destruction script, which simply removes the object after 1 and a half seconds.

UsingInstantiate.cs × | RocketDestruction.cs ×

RocketDestruction > Start ()

```
1  using UnityEngine;
2  using System.Collections;
3
4  public class RocketDestruction : MonoBehaviour
5  {
6      void Start()
7      {
8          Destroy (gameObject, 1.5f);
9      }
10 }
11
```

https://unity3d.com/learn/tutorials/modules/beginner/scripting/arrays

25

Arrays

Arrays are a way of storing a collection of data of the same type together. Let's say we need to store 5 integers. Rather than storing them individually as myIntA, myIntB, myIntC etc we can store them in an array. We declare an array in a similar way to declaring an individual variable. However after the type of the variable we use open and closed square brackets.

int []

It's important to make the distinction that an array is not a type but a collection of variables of a certain type. Before an array can be used we need to know it's length, i.e., how many elements will be stored in it. Elements are what we items in an array. To specify the length we use the 'new' key word followed by the type of array and the number of elements in square brackets.

```
type[] nameOfArray = new type[number of elements];
```

We can now initialise our array in the Start function. We can access and initialise elements of the array by using the name of the array followed by square brackets containing the index of the element. The index of an array is simply an integer with the first index being 0. When thinking of an elements index, think about how many steps away from the first element the one your trying to access is. So to access the first element use the index 0 since it's 0 steps away from the first. For the second element use the index 1and so on.

```
nameOfArray[index of element to accsess] = value;
```

```
using UnityEngine;
using System.Collections;

public class Arrays : MonoBehaviour
{
	int[] myIntArray = new int [5];

	void Start ()
	{
		myIntArray[0] = 12;
	}
}
```

Now I'm going to initialise the 5 elements in my array.

```
using UnityEngine;
using System.Collections;

public class Arrays : MonoBehaviour
{
	int[] myIntArray = new int [5];

	void Start ()
	{
		myIntArray[0] = 12;
		myIntArray[1] = 76;
		myIntArray[2] = 8;
		myIntArray[3] = 937;
		myIntArray[4] = 903;
	}
}
```

How ever, if I want to I can do it all in one line. This means I can initialise my array at the same time as declaring it. This means using braces as we normally would for a function, loop or IF statement. Between the braces we put the value for our elements in the order that we'd like them to appear in the array. This will create an array of 5 ints just as before, starting with 12 then 76 and so on.

type[] nameOfArray = {value, value, value, value, value};

```
using UnityEngine;
using System.Collections;

public class Arrays : MonoBehaviour
{
	int[] myIntArray = {12, 76, 8, 937, 903};

	void Start ()
	{

	}
}
```

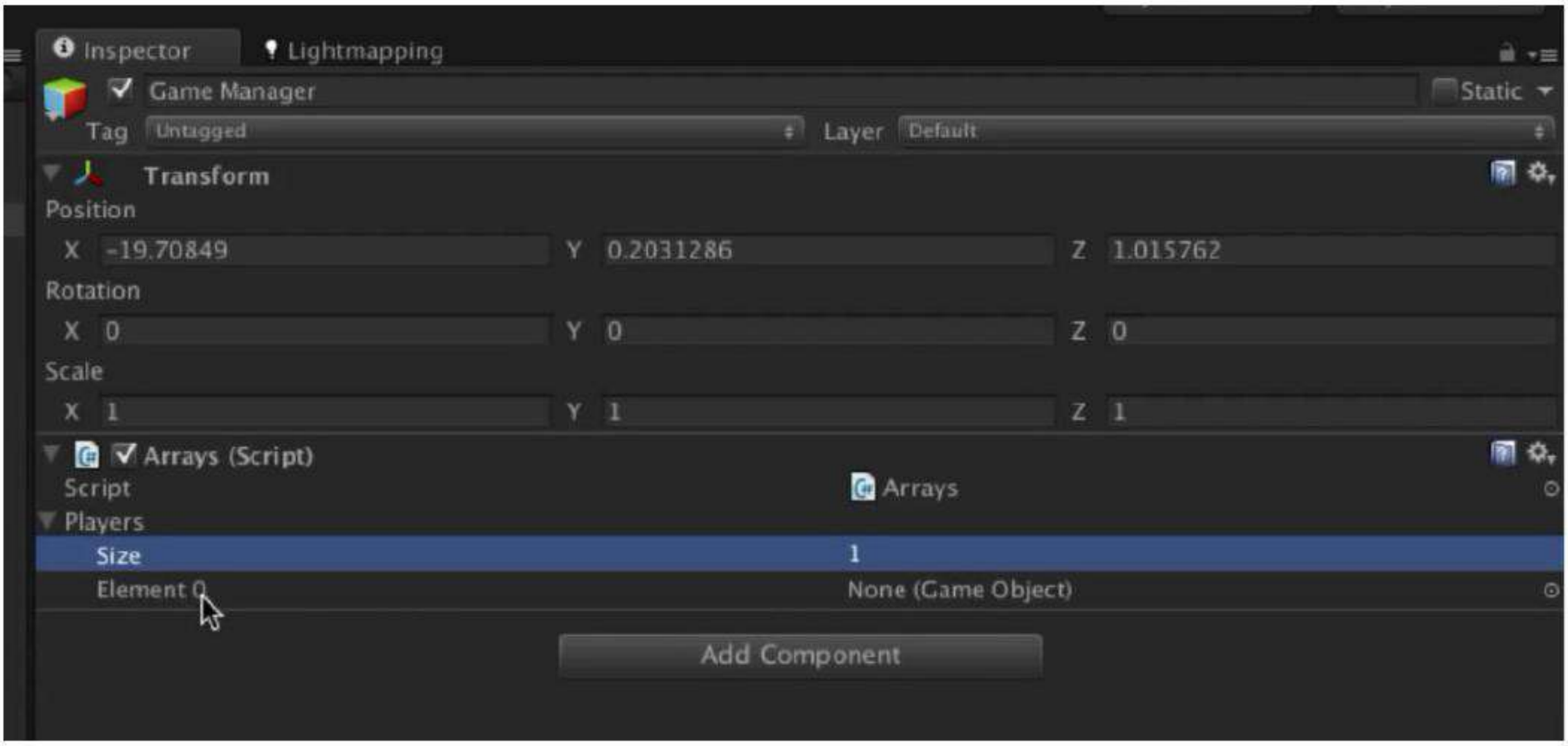
Note we didn't specifically state the length this time. This is defined by the number of elements within our curly braces. Arrays declared in this way can still be accessed in the same way as demonstrated in our first example, by using:

nameOfArray[index of element to accsess] = value;

```
using UnityEngine;
using System.Collections;

public class Arrays : MonoBehaviour
{
	int[] myIntArray = {12, 76, 8, 937, 903};

	void Start ()
	{
		myIntArray[4] = 57;
	}
}
```

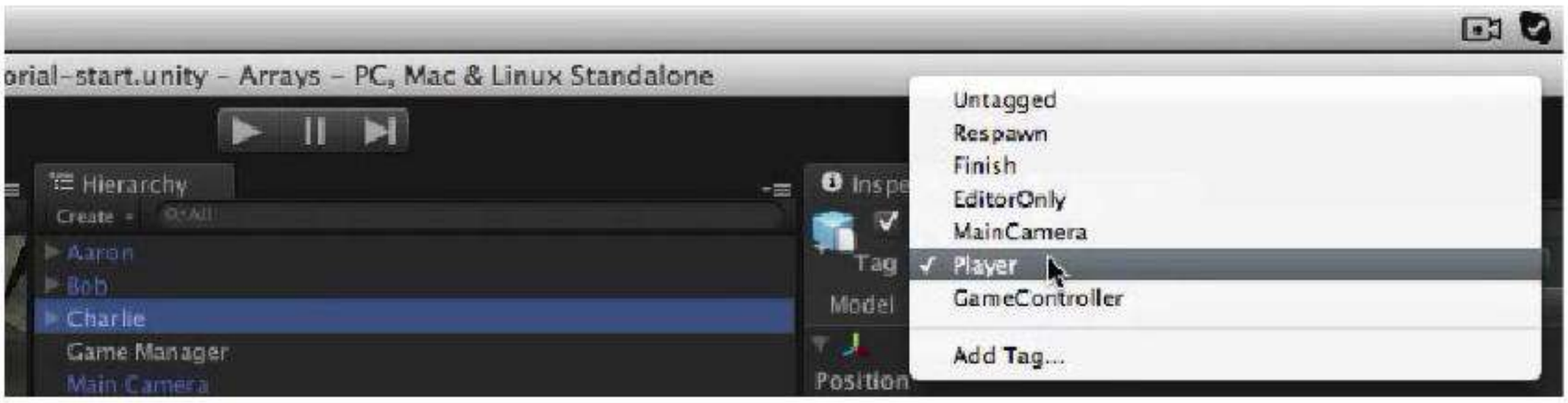
There are a few things to note about arrays. If you make an array public you will be able to see it in the inspector and allocate values to it.

In the inspector I can see where it's assigned to an object my players array is showing publicly in the inspector and I can assign a length by using the size value. So if I type in a value of 1 in to size I get 1 element in my array on to which I can drop a game object, because that's the type of this array. How ever we can use the script to initialise it. Unity has some functions to help us do this.



We want our players array to store all of the players in our scene. The function FindGameObjectsWithTag returns an array of all the game objects in the scene with a specified tag. Since our player game objects all have the player tag we can pass this in and get all of the players in the scene. We can use the array that this function returns to initialise our new public players array. So simply by doing this we have assigned everything that this ("player") function has returned to our players array. That means that it's collected the objects Aaron, Bob and Charlie as they all contain the tag Player.

```
using UnityEngine;
using System.Collections;

public class Arrays : MonoBehaviour
{
	GameObject [] Players

	void Start ()
	{
		players = GameObject.FindGameObjectsWithTag("Player");
	}
}
```

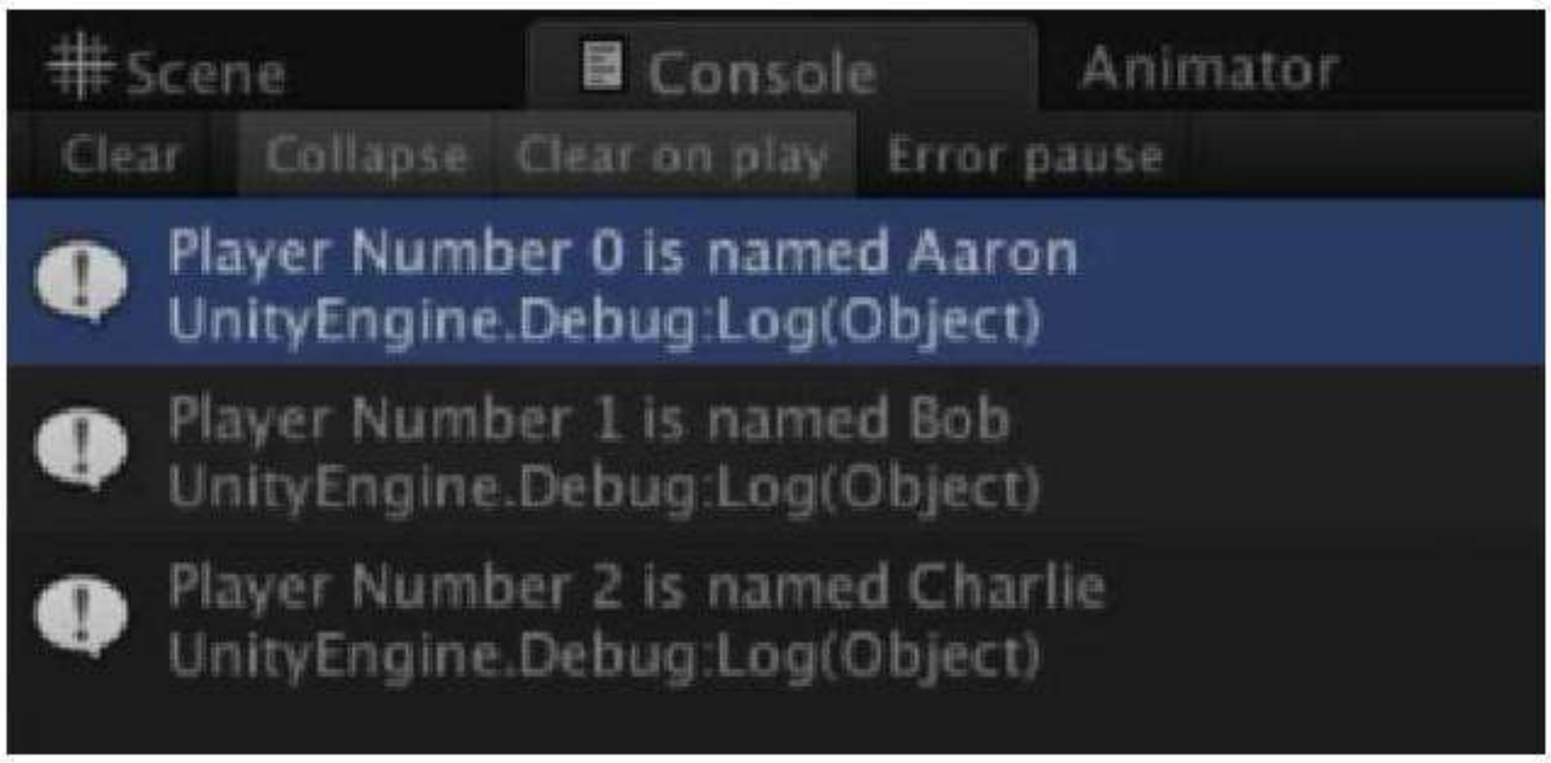


Another great thing about arrays is that they work really well with loops. Lets say that we want to log the name of all of the players in our scene. We will use a For loop to iterate through each element of the array.

Arrays have a property for their length. This will return the number of elements in an array. This means that if we start our loops iterator at 0 and continue to loop whilst it's less than the length of the array, we can perform a number of actions equal to the number of elements in our array. What's more our iterate variable 'i', will also be equal to the index number in our array. This will print out a sentence that says 'Player number 0 is names Aaron'. It does this by taking the text Player Number followed by the 'i' variable, representing the number, or index, and finally it retrieves the player's name by looking at a particular index in the 'players' gameobjects array.

```
using UnityEngine;
using System.Collections;

public class Arrays : MonoBehaviour
{
    public GameObject[] players;

    void Start ()
    {
        players = GameObject.FindGameObjectsWithTag("Player");

        for(int i = 0; i < players.Length; i++)
        {
            Debug.Log("Player Number "+i+" is named "+players[i].name);
        }
    }
}
```

When we press play we can see that our console has logged 3 messages.



One for each item in the array until it has reached the end of the array elements. You can also see that the array is assigned to our public variable in the inspector as the script initialised it.

https://unity3d.com/learn/tutorials/modules/beginner/scripting/invoke

26

Invoke

The invoke functions enable you to schedule a function call after some specified time delay. This allows us to build a useful method call system that is time sensitive. Here we have a scene in Unity with an empty game object. The game object has an invoke script attached to it. In the invoke script we have a public GameObject name target. We also have a method named SpawnObject. The SpawnObject method simply instantiates the target object at the position 0, 2 ,0. In the start method we call the function Invoke. The Invoke function takes 2 parameters; a string containing the name in the method that we wish to call, and an amount of time to delay in seconds. We can see from this line of code that after 2 seconds the spawn object method will be called. It is worth noting that only methods which take no parameters and have return type of void to can be called using invoke.

```
using UnityEngine;
using System.Collections;

public class InvokeScript : MonoBehaviour
{
    public GameObject target;


    void Start()
    {
        Invoke ("SpawnObject", 2);
    }

    void SpawnObject()
    {
        Instantiate(target, new Vector3(0, 2, 0), Quaternion.identity);
    }
}
```

When we run our scene we can see that after 2 seconds an orb is instantiated and falls to the ground. This is useful for calling methods a single time. But what if we wanted to call a method repeatedly? This can easily be done using the invoke repeating function.

The invoke repeating script we can see that things are laid out almost exactly as the invoke script.

```
using UnityEngine;
using System.Collections;

public class InvokeRepeating : MonoBehaviour
{
    public GameObject target;


    void Start()
    {
        InvokeRepeating("SpawnObject", 2, 1);
    }

    void SpawnObject()
    {
        float x = Random.Range(-2.0f, 2.0f);
        float z = Random.Range(-2.0f, 2.0f);
        Instantiate(target, new Vector3(x, 2, z), Quaternion.identity);
    }
}
```

We still have a game object named target and we still have a spawned object method. The spawn object method in this script still instantiates the target object, but now it does in a random X & Z position.

In the start method we have a call to the invoke repeating function. This function takes 3 arguments; a string containing the name of the method that we wish to call, the delay in seconds before it calls the method, and a delay in seconds between each subsequent call of the method. We can see this line of code the spawn object will be called after 2 seconds, then be called again every 1 second.

When we run our scene we will see after 2 seconds an orb gets instantiated. Then after 1 second another orb is created. This will keep going infinitely and fill our entire scene with orbs.

Obviously it is important to note how to stop a method we have called with the invoke repeating function. In order to stop all instances of an invoke call in this script we can use the CancelInvoke method.

CancelInvoke();

If we wish to stop a specific invoke we can pass in a string containing the method we wish to stop.

```
using UnityEngine;
using System.Collections;

public class InvokeRepeating : MonoBehaviour
{
    public GameObject target;


    void Start()
    {
        InvokeRepeating("SpawnObject", 2, 1);

        CancelInvoke("SpawnObject");
    }

    void SpawnObject()
    {
        float x = Random.Range(-2.0f, 2.0f);
        float z = Random.Range(-2.0f, 2.0f);
        Instantiate(target, new Vector3(x, 2, z), Quaternion.identity);
    }
}
```

27

Enumerations

When scripting in Unity sometimes we want a variable that is one of a set of constants. Consider the points of a compass. We could describe these using an integer where 0 was North, 1 was East 2 was South and 3 was West. This wouldn't be very easy to read or write in code since it means remembering what each number referred too. Instead we can create something called an Enumeration. These are often referred to as 'e-nums' are a special data type that has a specific subset of possible values.

Here we have a script named EnumScript. Let's create an enumeration to represent the different directions of a compass. We can create the enumeration either inside or outside of the class. We would place an enumeration in the class only if that class needed access to it. We would place the enumeration outside of the class if other classes also needed access to it. For this example we will be creating it inside the class, the syntax when declaring an enumeration is to start with the e-num keyword.

enum

Next we give it a name. For our example we will call the enumeration Direction. Note that we're using a capital letter for Direction. That's because we are effectively declaring a type and not a variable. Then we need to list the various constants of the enumeration separated by commas in braces. First we want North, then we want East, South and finally West. Unlike when you normally close braces this is the end of a statement, and so needs a semi-colon. Each of these constants declared in this enumeration has a value which at default is an integer starting at 0. This is incremented through the set. So North has a value of 0, East has a value of 1, South has a value of 2 and West has a value of 3.

```
enum Direction {North, East, South, West};
```

Both the type of value and the value themselves can be overridden if required. If we were to declare the enum with North equals 1, then East would have a value of 2, South a value of 3 and West a value of 4. Alternativly we could declare a value for each of the constants. Such as North equals 10 East equals 11, South equals 15 and West equals 27.

```
enum Direction {North = 10, East = 11, South = 15, West = 27};
```

We can also change the type of the constants in an enumeration. The constants can be changed to any integral type. We change the type putting a call in after the name of our enumeration and putting the type afterwards. For example we could have direction called in short. This would mean that the type of enumeration was short instead of int. One of the reasons for changing the type of an enum is optimization. But we generally won't worry about it.

```
enum Direction : short {North, East, South, West};
```

Once we have our enum declared we can then make variables of the enum's type. Let's make a direction variable called myDirection. We now can treat this like any other variable. But where as an integer might have a value of 5, or a float might have a variable of 97.3 our myDirection variable can have direction.north, direction.east, direction.south and direction.west. Let's set my direction to direction.north in the start function. Our enum can also be passed into and returned from functions. We simply use the name of the enumeration (in our case direction) where we would use a type. To illustrate this let's make a small function to reverse a given direction. Given North is should return South and so on. The return type for our function is Direction and we'll call the function ReverseDirection. We'll give the function a direction parameter, which we'll call dir for now. Now all we need to do is compare dir to our various direction constants and return the opposite. We only need to check for North, East, South and West because those are the only possible values.

```
using UnityEngine;
using System.Collections;

public class EnumScript : MonoBehaviour
{
    enum Direction{North, East, South, West};


void Start ()
    {
        Direction myDirection;

        myDirection = Direction.North;
    }

    Direction ReverseDirection (Direction dir)
    {
        if(dir == Direction.North)
            dir = Direction.South;
        else if(dir == Direction.South)
            dir = Direction.North;
        else if(dir == Direction.East)
            dir = Direction.West;
        else if(dir == Direction.West)
            dir = Direction.East;

        return dir;
    }
}
```

https://unity3d.com/learn/tutorials/modules/beginner/scripting/switch

28

Switch Statements

When making decisions in code it is common to make an IF statement or a series of IF else statements. An alternative to this is can be the switch statement. A switch statement is a more streamlined conditional that is used when you want to compare a single variable to a series of constants. A common use of switches is to make decisions based on an enumeration. Consider a game where the characters conversation options were based on their intelligence. You could set these options using a series of If else statements but that could get cumbersome. Instead we can use a switch statement to more easily implement our solution.

Here we have a script called ConversationScript. In this script we will implement a function that will allow our character to give a greeting based on their intelligence. First we will create an integer. For our characters intelligence. We will set this variable equal to 5. For the purpose of this example let's say that intelligence ranges from 5 being the highest to 1 being the lowest. Next we will create a function called greet which will use a switch statement to output a greeting based on the characters intelligence. The syntax for a switch statement starts with the keyword switch. Followed by the variable, which it will be comparing in parenthesis. In this case we will be comparing our intelligence variable. We follow that with some open and closed braces. Inside the braces we need to define a series of cases. These cases are the constants that we are comparing our variable against. To create a case we use the keyword case followed by our constant and a colon. As you can see we are starting by defining what the character will say if they have an intelligence of 5.

```
using UnityEngine;
using System.Collections;

public class ConversationScript : MonoBehaviour
{
    public int intelligence= = 5;

void Greet ()
    {
            switch (intelligence)
            {
              case 5:
            }
        }
}
```

Let's bring out a greeting that's somewhat of higher intelligence by print. As I am not a particularly high intelligence myself I'm going to guess.

```
print ("Why hello there good sir! Let me teach you about Trigonometry!");
```

After the instructions for each case we need to include something called a jump statement. The most commonly used jump statement for switches is break. This is a keyword that jumps the code execution out of the switch so that none of the subsequent cases are executed. Think of break working like the return keyword, but where as return is commonly used in functions, break is used in switches and loops. After encountering a break, code will continue execution after the block of code is in. In our situation, since the break is inside of a switch, after it is encountered execution will continue just after the switch. Let's put in a couple more cases with phrases of de-creasing intelligence.

```
using UnityEngine;
using System.Collections;

public class ConversationScript : MonoBehaviour
{
    public int intelligence= = 5;

void Greet ()
    {
          switch (intelligence)
          {
            case 5:
           print ("Why hello there good sir! Let me teach you about Trigonometry!");
           break;
            case 4:
           print ("Hello and good day!");
           break;
            case 3:
           print ("Whadya want?");
           break;
            case 2:
           print ("Grog SMASH!");
           break;
            case 1:
           print ("Ulg, glib, Pblblblblb");
           break;
           default:
           print ("Incorrect intelligence level.");
           break;
          }
    }
}
```

For our final case we want to catch everything that doesn't have a case of it's own. To do this, instead of using case we use another keyword default. Since this should be for any case that has not already been covered it does not need a value. Other than that we treat is as if it as we would any other case. Think of default as the else in an if else statement. Just like else default covers any situation not caught by a proceeding conditional.

29

MonoDevelop's Debugger

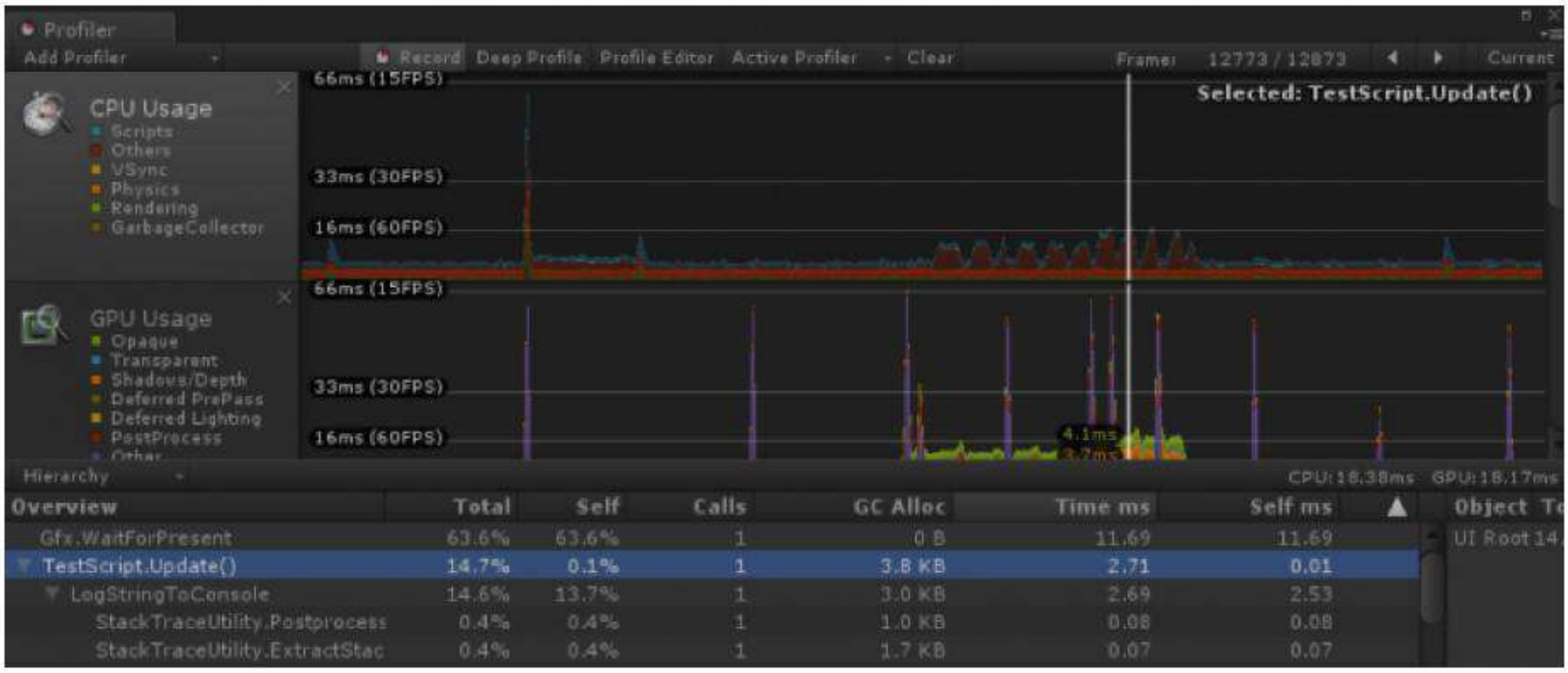
Debug.Log (or print) can be a great tool for outputting messages to the console when things happen. But what if you want to check the state of a bunch of variables at a certain point in your application?

One way of doing that would be to write a huge Debug.Log message like this:

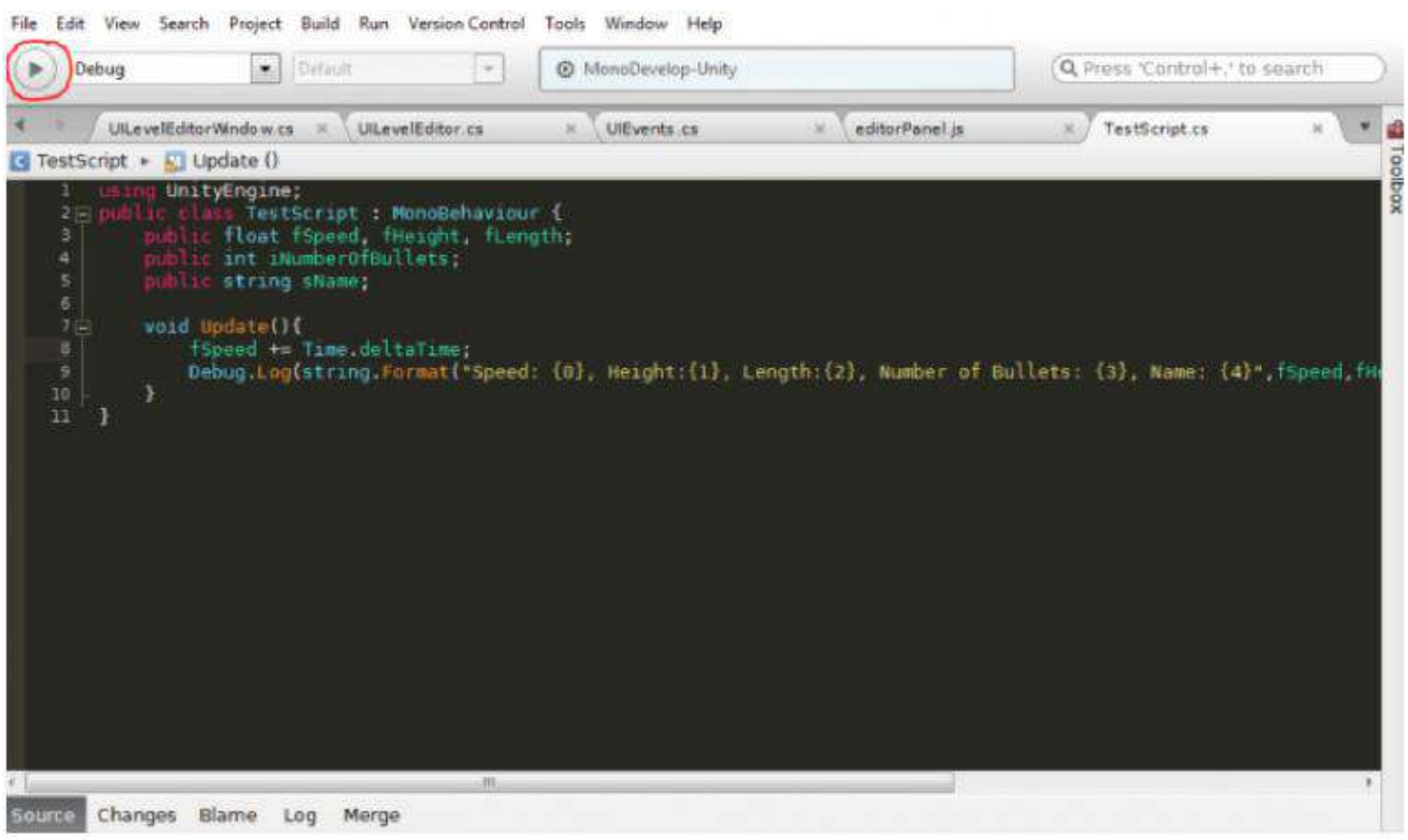
```
using UnityEngine;
public class TestScript : MonoBehaviour

{
    public float fSpeed, fHeight, fLength;
    public int iNumberOfBullets;
    public string sName;

    void Update()
     {
        fSpeed += Time.deltaTime;

      Debug.Log(string.Format("Speed:{0}, Height:{1}, Length:{2},
      Number of Bullets: {3}, Name: {4}",fSpeed,fHeight,fLength,
     iNumberOfBul lets,sName));}
}
```

Which will result in your console window looking something like this...

Not to mention the spikes your frame-rate will suffer because it will write to disk each frame....



This example is very basic, but I'm sure if you have been programming with Unity for long enough you will have used Debug.Log to output something at some point. I'd recommend using it for most situations, however there are some instances when a Debug.Log just wouldn't cut it. To start using the Debugger, all you have to do is press the "Play" button in the top left of MonoDevelop's window:



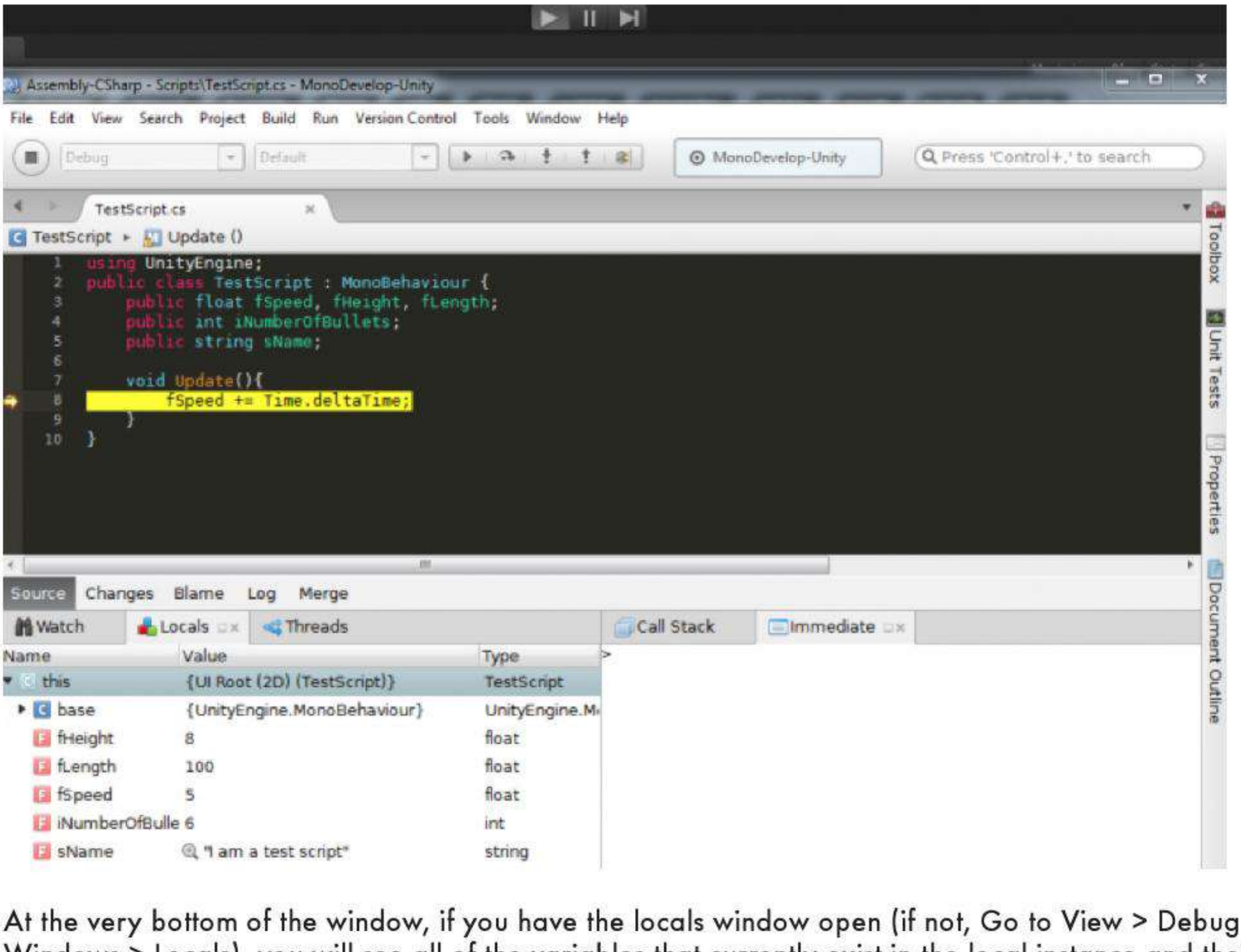
If you press that, MonoDevelop will prompt you to "Attach to a Process". Just select your instance of unity and press attach. The dialogue will vanish and you will be left with a "Stop" button instead of a play button.

To start debugging, press your mouse to the far left edge of the editor (next to the line number) and a red dot will appear, you would have just created a breakpoint!

A breakpoint

```
1  using UnityEngine;
2  public class TestScript : MonoBehaviour {
3      public float fSpeed, fHeight, fLength;
4      public int iNumberOfBullets;
5      public string sName;
6
7      void Update(){
8          fSpeed += Time.deltaTime;
9      }
10 }
```

This will not do anything for now, however, if you now go to Unity and press play in your editor window something great will happen...



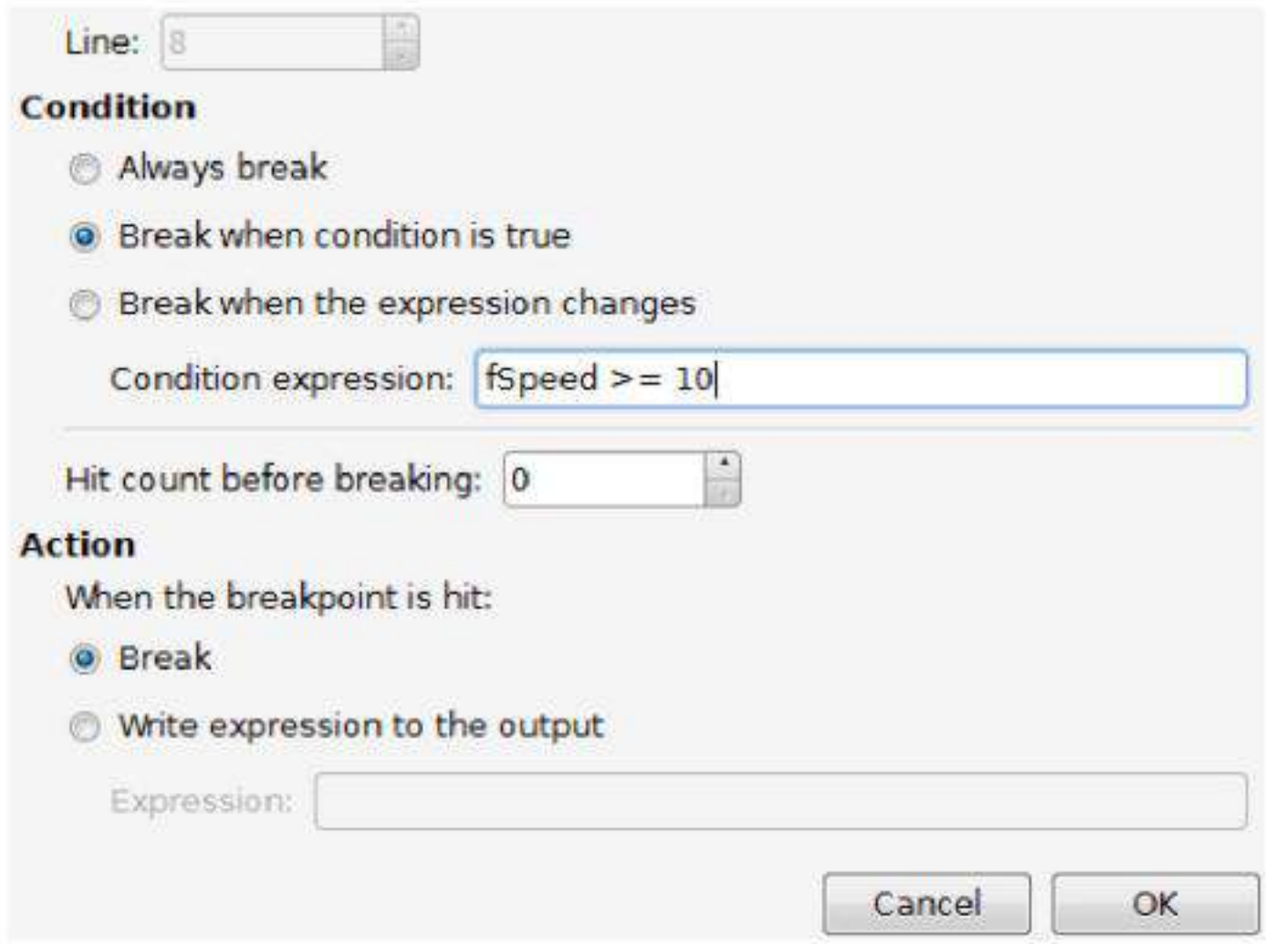
At the very bottom of the window, if you have the locals window open (if not, Go to View > Debug Windows > Locals), you will see all of the variables that currently exist in the local instance and their values at the time of the breakpoint being hit.

To continue the applications execution, just press the "Play" button in MonoDevelop

Your script will continue its execution (and Unity's editor will no longer be frozen). Of course in this instance, the script will hit the breakpoint again on the next frame. So just left click the breakpoint in MonoDevelop and hit the Play button again so it doesn't execute the breakpoint again.

It gets better!

With breakpoints, you can make them stop the application running when certain conditions are met. For example, imagine you want to check what the values are when the fSpeed variable reaches 10. To do that, press the Stop button in MonoDevelop, and Right click on your breakpoint, then press Breakpoint Properties.



Set the condition to "Break when condition is true" and set the "Condition Expression" to "fSpeed >= 10" and then press OK.

Re-attach the editor to Unity and press the Play button in Unity, when the condition is met the breakpoint will fire and stop the application.

A note about using Condition Breakpoints: They cause performance issues as it has to validate the expression each time it is run.