Creating perceptual and behavioral experiments with Unity and VR

Notes for a Short Course at Color and Imaging Conference 33

October 27, 2025

Richard F. Murray
Centre for Vision Research
York University

Contents

Chapter 1

Getting started

In these notes, I'll write down some important points about Unity that we'll cover in the course, for your reference during the course and after it's finished. The notes aren't meant to be a complete tutorial, and there are some skills, such as using Unity's graphical interface to create realistic scenes, that you can easily learn more about from other sources. (See Chapter 5 for suggestions on where to learn more.) Instead, I'll focus on basic features of Unity that will get you started, and on topics that are specific to running research experiments and are hard to find information about elsewhere.

In this chapter we'll cover the basics of how to install Unity and create a new project.

1.1 Installing Unity

Unity projects are created and managed from an app called the Unity Hub. When you "install Unity", that means downloading and installing the Unity Hub, which you can do from here:

`https://unity.com/download`

After you've installed the Unity Hub, create a Unity account and log in. You'll be given some choices about which edition to use. The Unity Personal edition will be fine. You'll be given the option to install the Unity Editor, and you should do that as well. In this course, we'll use Unity Editor version 2022.3 LTS, so choose that version. (That's the version that the calibration methods we'll discuss have been developed and tested for.) The available version numbers may have some extensions after 2022.3, such as 2022.3.10f1. Any version that begins 2022.3 will be fine. You'll be given many checkboxes that provide the option of installing add-ons to the Unity Editor. If there's a checkbox for Visual Studio Code, check that one, but you won't need any of the others. Downloading the Unity Editor may take some time, and you can check its progress by clicking the 'download' icon in the top right corner of the Unity Hub window. After the installation is done, the Unity Hub will look like Figure 1.1.

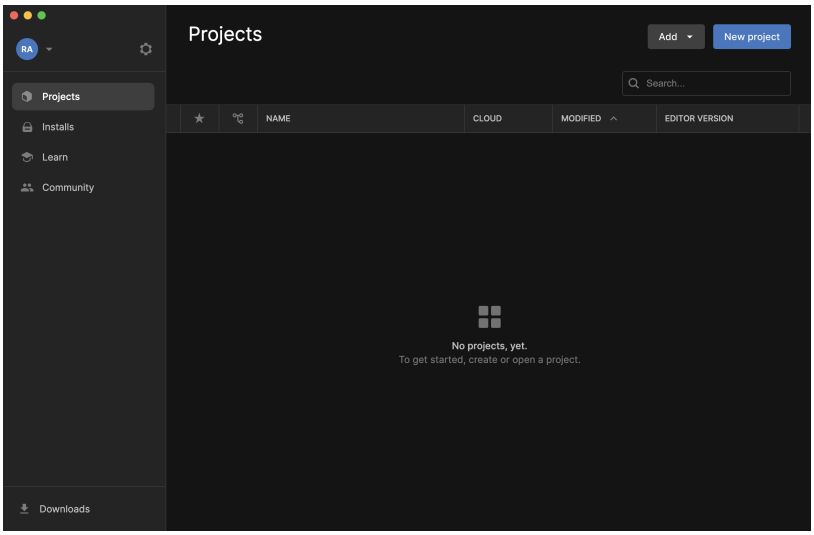

Figure 1.1: The Unity Hub.

1.2 Creating a new project

A Unity program is called a 'project'. From the Unity Hub (Figure 1.1), click the 'New project' button at the top right. The 'New project' window (Figure 1.2) will open. From the list of templates, choose 'High Definition 3D / Core'. For 'Project name', enter a name such as 'CIC test project', and for its location, enter a convenient directory such as your desktop. Leave the other options with their default values. The first time you create a High Definition 3D project, you may be told you need to download the template; click the button that does that. Then click 'Create project' at the bottom right. The Unity Hub will show a splash screen and launch a new project. It may take a few minutes for this process to finish.

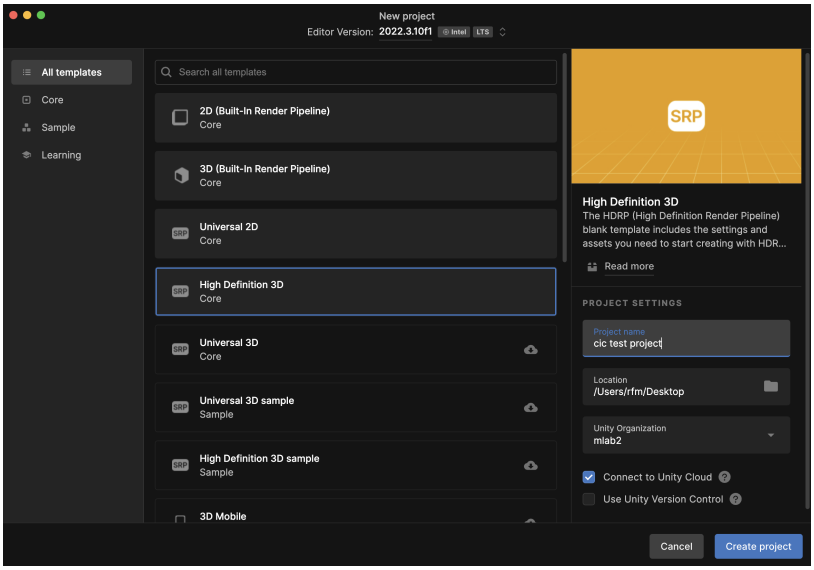

Figure 1.2: The new project window.

When the new project has launched, you'll see two windows. One is the 'HDRP Wizard' (Figure 1.3). You won't need this window for now, so you can close it.

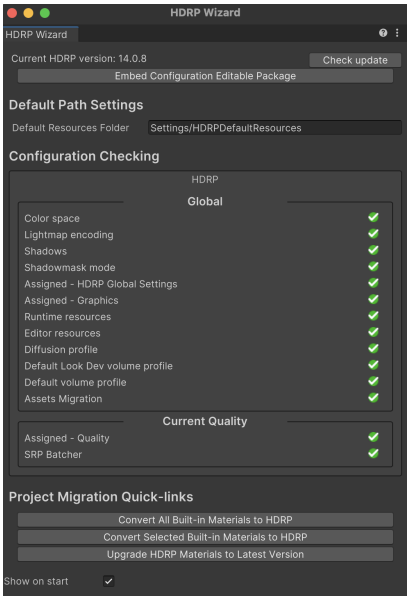

Figure 1.3: The HDRP Wizard.

The other window that will open is the Unity Editor (Figure 1.4), which will be our main interface for creating experiments.

1.3 Creating a scene

Creating an experiment in Unity has two phases. First, you'll use the Unity Editor (Figure 1.4) to design a scene that contains lights, objects, a camera, and various other components. Then, you'll write scripts using the C# programming language to give those components behaviours, such as moving through the scene, or responding to a user's actions with the keyboard and mouse. After making a first draft of the experiment, you'll usually alternate between these two phases several times, fine-tuning the components and their behaviours until they're ready to go.

In the course, I'll demonstrate how to use the Unity Editor to create scenes for an experiment. You can also find references for readings on this topic in Chapter 5. In the next two chapters, I'll describe some parts of C# and the Unity scripting interface that we'll need to write scripts for the components of the scene.

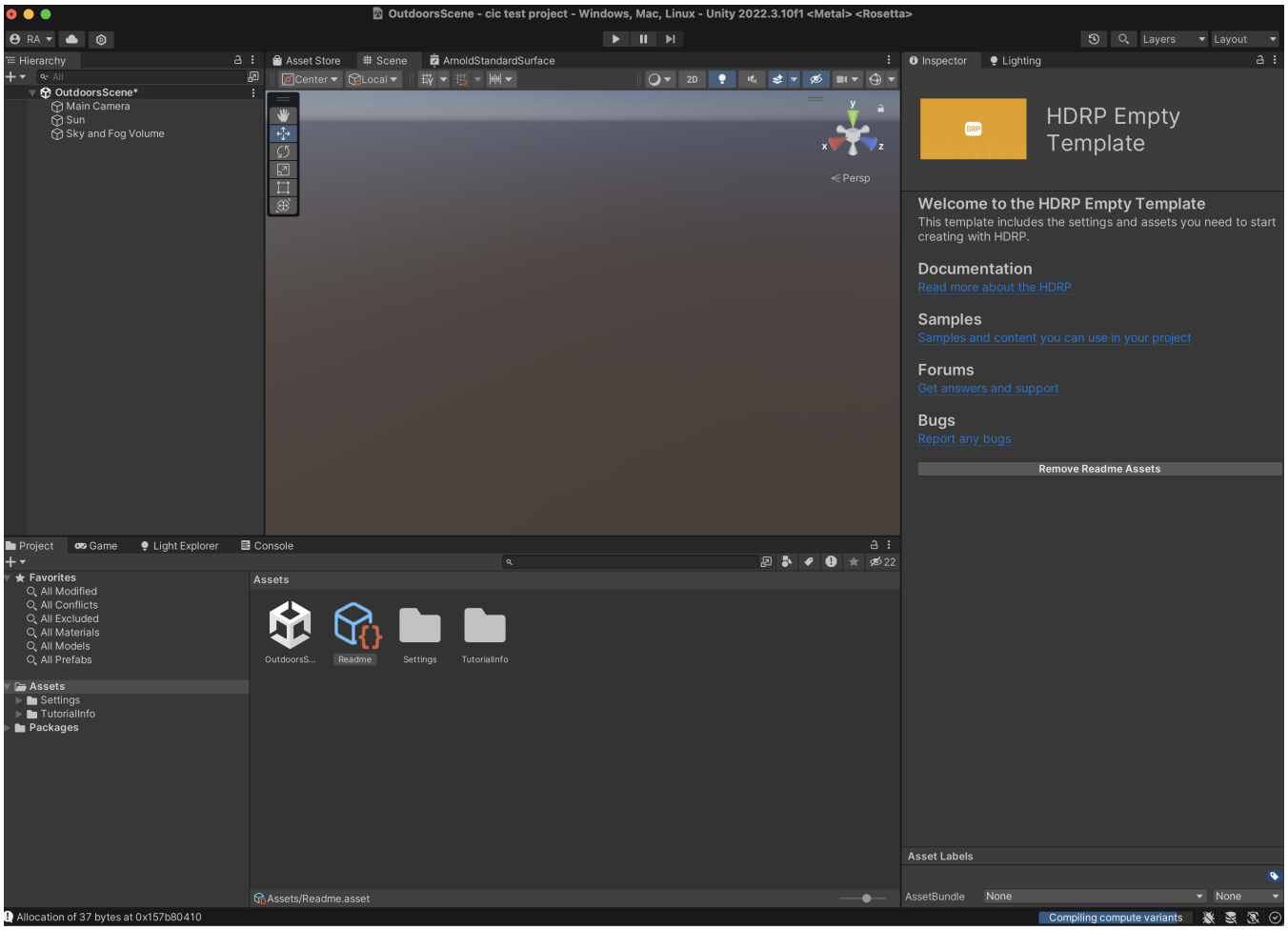

Figure 1.4: The Unity Editor.

Chapter 2

Coding in C#

In these notes, I'll assume that you have some programming experience, though not necessarily with C#. That means you're familiar with basic features that are common to most programming languages: variables, if statements, for loops, functions, and so on. Instead of giving a complete description of C#, I'll cover the main programming constructs that you'll need in order to start creating experiments. The scripts that you'll write for Unity using C# will be fairly simple, and you'll probably find that it's relatively easy to learn enough about C# for your applications.

2.1 Variables and data types

C# is a strongly typed language. That means we have to define every variable as having a specific data type before we use it. For example, if we want to use a variable called `trial_num` to keep track of the trial number in an experiment, we'll have to have a line like the following before we use that variable.

```
int trial_num;
```

This declares a new variable of type 'int', meaning integer. Also notice the semicolon: most statements in C# end with a semicolon. Optionally, we can also give the variable an initial value.

```
int trial_num = 1;
```

After the variable has been declared this way, we can use it without declaring it again, for example by increasing its value by one.

```
trial_num = trial_num + 1;
```

Here are examples of a few other useful data types.

```
float reaction_time1 = 2.34f;  // floating point, with 4 bytes of precision
double reaction_time2 = 3.45;  // floating point, with 8 bytes of precision
bool correct_response = true;  // Boolean; can be true or false
char keypress = 'x';           // character; note the single quotes
string subject = "jfk";        // string; note the double quotes
```

After a variable has been declared, its type is fixed. Trying to assign a value of 128 to the string variable 'subject', for example, would generate an error. This is quite different from some other languages, such as MATLAB and Python, where we can use variables without declaring them, and a variable's data type can change over the course of a program.

For example, this code generates an error in C#.

```
double x = 1;
int y = x;
```

C# will do some type conversions automatically, but not this one, because information can be lost when we convert a floating point number to an integer. To do this, we have to *cast* the data type.

```
double x = 1;
int y = (int)x;
```

We put the target data type in parentheses, just before the value to be converted. This works for some simple conversions, but not for others, such as strings to integers, where the conversion is more complex. Other methods are available for those conversions, but we won't need them for now.

We can also define *arrays* of these basic data types, such as an array of integers.

```
int[] trialNums = new int[5];
```

We can optionally assign initial values to an array as well.

```
int[] trialNums = {10, 20, 30, 40, 50};
```

After this declaration, trialNums[0], trialNums[1], and so on up to trialNums[4] are integer variables that we can use just like any other integer variable. The numbers inside the brackets are *indices* into the array. The indices start at zero, so an array with n elements has indices from 0 to $n - 1$.

For arithmetic, addition is denoted by `+`, subtraction is `-`, multiplication is `*`, and division is `/`. Parentheses control the order of operations.

```
double x = 1.5 + ((2.3 - 3.4)/(4.1 * 5.5));
```

C# has shortcuts for some common arithmetic operations. One shortcut increments a variable by some amount.

```
x += 5;
```

This is a shortcut for

```
x = x + 5;
```

Incrementing a variable by one is so common that there's also a shortcut for that.

```
x++;
```

This is a shortcut for

```
x = x + 1;
```

2.2 A small detour: .NET Fiddle

It's helpful to be able to experiment with code when learning a language, and there's a website where you can do that with C#.

https://dotnetfiddle.net

The web interface starts with the following code.

```
using System;

public class Program
{
    public static void Main()
    {
        Console.WriteLine("Hello World");
    }
}
```

If you click the 'Run' button at the top of the page, this code will run, and the text 'Hello World' will appear in the lower part of the window. Later on we'll look more closely at the lines that start with 'using' and 'public', but for now you just need to know that you can replace the line that begins with 'Console' with one or more lines of your own, and then run the code. For example, try replacing the 'Console' line with the following lines, and then run the program.

```
int trial_num = 1;
Console.WriteLine(trial_num);
```

You should see the value of `trial_num`, namely the integer 1, appear in the lower window. From here, you can experiment with the code and answer questions you may have about how C# works. What happens if you try to print the variable without declaring it, or before assigning it a value? What happens if you try to assign it a value that has the wrong data type? Experimenting with code this way is an important part of learning a new programming language.

At the left of the .NET Fiddle window, you can choose the version of the C# compiler to use. Unity uses version 9, which at the time of writing is the default version in .NET Fiddle.

2.3 The if statement

You can use an if statement to test whether a condition is true, and if it is true, then run some code.

```
if (a > 0)
    b = 1;
```

Here we assume that we've already declared the variables `a` and `b`. We test whether `a > 0`, and if it is, then we assign a value to `b`. By convention, we indent the code that is run if the condition is true, but that's just a convention that makes code easier to understand, and the program will run just the same if we don't indent. (This is quite different from Python, where indentation is an important part of the language.) By default, the if statement applies to just the one next line of code. If we want two or more lines of code to be run if the condition is true, we have to enclose them in set braces { }.

```
if (a > 0)
{
    b = 1;
    c = 2;
}
```

After the if clause, we can also have one or more optional 'else if' clauses, and finally one optional 'else' clause.

```
if (a > 0)
{
    b = 1;
    c = 2;
}
else if(a == 1)
    b = 10;
else if(a == 2)
    b = 20;
else
    b = -1;
```

The 'else if' and 'else' statements can also have multiple lines of code grouped by set braces.

Here are a few comparisons that result in Boolean values, which we can use in an if statement.

```
a > 0
a < 0
a >= 0
a <= 0
a == 0  # test for equality; note two equals signs
a != 0  # test for inequality
```

Logical 'and' is denoted by &&, logical 'or' is ||, and logical 'not' is !. So for example,

```
if ( a >= 15 && !( a >= 27 ) )
    b = -1;
```

2.4 The while loop

We can use a while loop to repeat code for as long as a condition is true.

```
while (a < 10)
    a = a + 1;
```

This code also assumes that we've declared the variable `a`. As with the if statement, if we want multiple lines to be repeated in the loop, we put them in set braces.

```
while (a < 10)
{
```

```
        a = a + 1;
        b = b - 1;
    }
```

Any Boolean expression that can be used in an if statement can also be used as the condition at the top of a while loop.

Within a loop, the 'continue' statement immediately returns to the top of the loop, and the 'break' statement exits the loop. The following code is a roundabout way of printing the integers 11 to 15.

```
int a = 0;
while (true)
{
    a++;
    if (a <= 10)
        continue;
    Console.WriteLine(a);
    if (a >= 15)
        break;
}
```

C# also has a do-while loop, which evaluates the loop code at least once before checking the condition.

```
do
{
    a = a + 1;
    b = b - 1;
}
while (a < 10);
```

Note that the do-while loop requires a semicolon after the while conditon.

2.5 The for loop

The syntax of a for loop is different in C# than in most other languages, and similar to C/C++. Here's an example that sums the numbers 0 to 10.

```
k = 0;
for (i = 0; i <= 10; i++)
    k = k + i;
```

The top line of a for loop has three statements, separated by semicolons, inside parentheses. The first statement is run before the loop begins, so here we set `i = 0`. The second statement is run to decide whether to make another pass through the loop. Here we check `i <= 10`, and if that statement is true, then we'll make another pass through the loop. The third statement is run after each pass through the loop. Here we run `i++`, to increment the loop variable. As a result, this for statement sets `i` to the values 0 to 10, and repeats the loop code for each value.

This code assumes that we've already declared the variables `i` and `k`. We only used `i` as a loop variable, and after the loop is done we don't need it anymore. In order to keep loop variables from proliferating, we can also write a for loop as follows.

```
k = 0;
for (int i = 0; i <= 10; i++)
    k = k + i;
```

Here we declare the loop variable `i` inside the parentheses. This way, the variable exists for the duration of the loop, and ceases to exist when the loop is done.

As with a while loop, multiple lines of code are grouped together with set braces, and we can use the continue and break statements to change the sequence of events as we go through the loop.

2.6 Functions

Another way in which C# is strongly typed is that when we define a new function, we specify the data types of the input and output arguments.

```
double calc(int a, int b)
{
    double c = 0.5*(3*a + 2*b);
    c = c + 10;
    return c;
}
```

Here we declare that the input arguments are integers, and the return argument is a floating point number. Passing arguments of the wrong type will generate an error. The code that defines the function is given inside set braces, and the return statement gives the return argument.

We can define a function like 'calc' in .NET Fiddle as follows.

```
using System;

public class Program
{
    public static void Main()
    {
        double x = calc(1,2);
        Console.WriteLine(x);
    }

    static double calc(int a, int b)
    {
        double c = 0.5*(3*a + 2*b);
        c = c + 10;
        return c;
    }
}
```

We place the definition of our new function just after the definition of the Main() function. Also note that here we've added a keyword 'static' to the beginning of the first line of the function definition. In the next section, we'll see why that's necessary in this case.

2.7 Classes and objects

A *class* is a data type that can contain both variables and functions. The variables in a class are usually called *fields*, and the functions are usually called *methods*. An *object* is an instance of a class.

Here's an example. C# has a class called Random, which we can use to generate random numbers. In order to do that, we create a variable of type Random. To create a variable from a class we use the following syntax.

```
using System;
Random rng = new Random();
```

First, the statement 'using System' makes the class Random available to us. We'll discuss statements like this at length in the next section, on namespaces. For now, our main interest is the second line, where we declare a variable 'rng' of type Random, and assign it a new instance of the Random class.

If an analogy helps, note that this is similar to how we created an array of integers in Section 2.1.

```
int[] trialNums = new int[5];
```

In fact, an array is a special kind of class in C#. Here too, we have a class type (int[]), and we declare a variable (trialNums) to have that type. In order to have the variable actually contain an instance of the type, we have to initialize it with the 'new' operator.

Returning to our example with Random, that class has a method Next(), which generates a random number. We can use the method as follows.

```
int x = rng.Next(0, 100);
```

Notice the syntax: we use the variable name, then a period, then the method name and any arguments. This line assigns x a random integer from 0 to 99. For the Next() method, the first argument is the lower bound, which is inclusive, and the second argument is the upper bound, which is exclusive.

This syntax highlights the fact that we're calling the Next() method for *this specific instance* of Random(), namely the variable rng. When we call a random number generator, that moves the generator along to the next random number, so it changes the state of the generator. If we have another instance of Random, say rng2, then calling rng.Next(0, 100) will change the state of rng, but not rng2.

In addition to methods (functions) and fields (variables), classes can also have *properties*. A property is like a function in some ways, and like a variable in other ways. We can use and set the value of a property as if it was a variable, but actually the class may be doing substantial calculations behind the scene when we do this. We'll see an example in the next paragraph.

The picture so far is that we create an object, which is an instance of a class, and then we use the methods, fields, and properties of that object. That's often true, but there is an exception to that approach. There are some methods, fields, and properties that it doesn't really make sense to connect to more than one object. For example, if we want to check the time that has elapsed since our program started running, that's the same for all objects. In this case we can use a *static* method, field, or property, which is associated with a class itself, rather than individual objects that are instances of the class. For example, the class UnityEngine.Time has a static property 'time' that returns the elapsed time since the program started.

```
float elapsed_time = UnityEngine.Time.time;
```

Here we use the property 'time' of the class Time itself, without first creating an object that's an instance of Time.

How do we know when to use a class this way? If you read the documentation for the class Time, it tells you that the 'time' property is static.

https://docs.unity3d.com/ScriptReference/Time-time.html

That means you should call Time.time directly, instead of creating an instance of Time. So when reading documentation, it's worth watching for the keyword 'static'.

In the previous section, we declared our function 'calc' to be static, because the .NET Fiddle app calls the Main() method without creating an instance of the class, so only static methods can be called from Main().

Now we can revisit the idea of a 'property' of a class. The syntax of the property Time.time suggests that it's a variable, because we didn't need parentheses, as we would if we had to call a function Time.time(). And yet clearly Time.time is not just a variable, because C# has to check a system clock in order to return the elapsed time. That's why we say properties are somewhat like both functions and variables. We can use them as if they're variables, but the class usually does some kind of computation in order to produce the value of the property.

Depending on your previous programming experience, you may or may not have used classes before. C# is an object-oriented language, which means it uses classes and objects extensively. In fact, almost everything in a C# program is defined as part of a class, so it's worth learning this part of the language thoroughly.

Now we can understand the default code in .NET Fiddle a bit better (see Section 2.2). We haven't discussed how to define new classes, but from the keywords in that code you can probably see that it defines a class called Program, and within that class it defines a method called Main(), which is the method that runs when we click the Run button. Here too, everything is defined as part of a class, even the program itself.

2.8 Namespaces

C# code is organized using *namespaces*. A namespace contains definitions for a group of classes. Here's a slightly modified version of default code for .NET Fiddle, which we saw in Section 2.2.

```
public class Program
{
    public static void Main()
    {
        System.Console.WriteLine("Hello World");
    }
}
```

We define a class Program, that contains a method Main(), that implements our program, which is a single function call. C# has a namespace called System, which contains many fundamental classes. One such class is Console, which is used for managing input and output. Console has a static method WriteLine(), which takes a string as an argument and writes it to the console. As you can see in this example, we call the method by giving the namespace System, then a period, then the class name Console, then another period, then the method name and its arguments.

Specifying all these components can be inconvenient. Instead, we can put the statement 'using System' as the first line of the program, and then we can use classes from System without specifying the namespace each time. That's what we did in Section 2.2, without completely explaining it at the time: we put 'using System' as the first line, and then we were able to call Console.WriteLine() instead of System.Console.WriteLine().

Namespaces can also be nested. Unity has a namespace called UnityEngine, and you'll usually see 'using UnityEngine' at the top of a Unity script. UnityEngine contains a sub-namespace called Rendering, which contains a sub-namespace called HighDefinition, which defines classes that are used with the High-Definition Render Pipeline. It contains a class called Exposure that we can use to control the exposure setting when rendering. We can refer to this class as UnityEngine.Rendering.HighDefinition.Exposure, but that's quite inconvenient, especially if we use other classes from that namespace as well. If we include a suitable 'using' statement,

```
using UnityEngine.Rendering.HighDefinition;
```

then we can simply refer to the class as Exposure.

To summarize: classes are defined in namespaces, and in order to have convenient access to those classes, we can put 'using' statements in our programs.

2.9 A simple program

Now that you know the basics of C#, one quick way of understanding the language more thoroughly is to see it in use. The code below is a simple sorting algorithm called 'bubblesort', written in C#.

The bubblesort algorithm works as follows. Suppose we have a list of elements to be sorted. We compare the first and second elements of the list, and if they're in the wrong order, then we switch their places. Then we compare the second and third elements, and if necessary, switch them. We continue like this for the third and fourth elements, and so on, until finally we compare and possibly switch the second-last and last elements. After this pass through the list, the list might not be completely sorted, but it should be closer to being sorted than it was before. So we repeat this procedure many times. If we ever make it through the whole list without switching any two elements, then we know the list must be completely sorted. This procedure is guaranteed to eventually sort any list. Although it's not a particularly efficient sorting algorithm, it is a nice introductory exercise in coding that uses many of the features of C# that we've covered.

```
using System;

// create an array of random integers from 0 to 99
int n = 16;                     // choose the number of integers
int[] x = new int[n];           // create an array of integers
Random rng = new Random();      // create a random number generator
for (int i = 0; i < n; i++)     // let i range from 0 to n-1
    x[i] = rng.Next(0, 100);    // get a random integer from 0 to 99

// sort the array of integers
while (true)                            // loop indefinitely
{
    int switches = 0;                   // initialize a counter
    for (int i = 0; i < n - 1; i++) // let i range from 0 to n-2
        if (x[i] > x[i + 1])            // wrong order?
        {
            int tmp = x[i];             // if so, switch the integers
            x[i] = x[i + 1];
            x[i + 1] = tmp;
            switches++;                 // increment the switch counter
        }

    // if no switches, then we're done
    if (switches==0)
        break;
}

// print the sorted integers
for (int i = 0; i < n; i++)
    Console.WriteLine(x[i]);
```

To run this program and experiment with it, you can copy it into .NET Fiddle (see Section 2.2). You should put the line 'using System' at the very top, and the rest of the code inside the function Main().

Chapter 3

Scripting

3.1 Introduction

In this chapter I'll describe some scripting methods for experiment-related tasks, such as checking for keyboard input and saving data in a text file. In the course, we'll create experiments that use these methods.

3.2 Creating a new script

To create a new C# script, right-click the mouse in the Assets view (a panel in the middle of the bottom half of the Unity Editor window), and a menu will appear. Hover over the 'Create' option, and a submenu will appear. Choose the option 'C# script', and name the new script icon that appears in the Assets view. You can use a name like 'NewScript'. If this is the main script where you'll run your experiment from the Start() and Update() methods (see the next section for more information), then the script must be attached to a Unity game object. The camera is a reasonable choice. Drag and drop the script from the Assets view onto the 'Main Camera' entry in the Hierarchy view (a panel in the top left corner of the Unity Editor window).

To see whether the drag-and-drop operation was successful, click the 'Main Camera' entry in the Hierarchy view, and look at the Inspector view (a panel in the top right corner of the Unity Editor window). If you scroll down to the bottom of the Inspector view, you should see a sub-panel whose title is the name of your script. If you don't see this, drag and drop the script from the Assets view to the 'Main Camera' entry in the Hierarchy view again.

After you've created the script, you can double-click its icon in the Assets view to open the script in an editor. Depending on which editor your system opens, such as Visual Studio, you may have to do some configuration such as logging in to a Microsoft account in order to use the editor.

3.3 Start() and Update()

After you've created a new script, it has placeholders for two functions: `Start()` and `Update()`. Almost all your code will be called from these two functions. `Start()` is called at the beginning of your program, so this is where you'll put initialization code. `Update()` is called once per video frame, which is typically around 60 times per second. This is where you'll put the code that runs your experiment.

3.4 Input devices

The UnityEngine.Input class contains methods for getting information from input devices such as the keyboard and mouse. You can find documentation on Input and its methods here:

https://docs.unity3d.com/ScriptReference/Input.html

Instead of creating an instance of Input, we call its static methods directly.

The method `GetKeyDown()` returns true or false to indicate whether a key has just been pressed down during the current video frame.

```
using UnityEngine;
bool keyDown = Input.GetKeyDown(KeyCode.A);
```

We pass a value from the KeyCode class to indicate which key we want to check. (KeyCode is also in the UnityEngine namespace.) In this example we check the A key. By passing other arguments, we can check other keys on the keyboard, or buttons on a controller.

```
bool buttonDown = Input.GetKeyDown(KeyCode.JoystickButton1);
```

You can find all the key codes provided by KeyCode here:

https://docs.unity3d.com/ScriptReference/KeyCode.html

`GetKeyDown()` returns true just once, when a key is first pressed. Sometimes we want a function that returns true for as long as the key is pressed, and for that we can use `GetKey()`.

```
bool keyPressed = Input.GetKey(KeyCode.A);
```

The Input class also has a method for checking the mouse position.

```
Vector3 mousePos = Input.mousePosition;
```

This returns a value of type `Vector3`, which is a data type with three floating point components. In this example, we can access the mouse coordinates as `mousePos.X` and `mousePos.Y`. This data type also has `mousePos.Z`, but for a mouse position that component is always zero.

We can check the mouse buttons as well.

```
bool mouseClick = Input.GetMouseButtonDown(0);
```

Here we pass the argument 0, which checks the left mouse button. To check the right button, pass 1, and to check the middle button, pass 2.

The UnityEngine.Input namespace is deprecated, but it works fine, and it's easy to use, so that's what we'll use here. You can read about its replacement, UnityEngine.InputSystem, to learn about a newer namespace that has some more advanced features.

3.5 Unity objects

In a script, we use C# code to control the appearance and behaviour of objects in the rendered scene. In order to do this, we need a variable that corresponds to each object. There are a few ways of establishing this connection between variables and objects.

One approach is to create a new object in the script.

```
GameObject cube = GameObject.CreatePrimitive(PrimitiveType.Cube);
```

GameObject and PrimitiveType are defined in the UnityEngine namespace. When this line runs, a new cube appears in the environment at the origin, and the variable 'cube' contains the information we'll need

to manipulate the cube using the tools described below.

Another approach is to create a new object manually, for example by using the menu and choosing GameObject / 3D Object / Cube, which places a new cube in the scene. Then we declare a public variable of type GameObject in the script.

```
public GameObject cube;
```

If the script contains a declaration like this, then we can use the Hierarchy view and choose the object (such as Main Camera) that we linked the script to. Then, in the Inspector view, under the section with the name of the script, there will be a new feature listed with the variable name we declared to be public, e.g., 'cube' in this example. We can drag the cube we created manually from the Hierarchy view to the box next to this feature. This establishes the link between the variable and the cube. When we run the program, the variable will contain the information we'll need to manipulate the cube.

A third approach is to use C# functions to search for an object that we created manually. We won't need this approach for the projects in this course, but you can read about it in the scripting chapters of Geig (2022), which I refer to in Chapter 5.

Once we have a variable that corresponds to a Unity object, we can use that variable to control the object's properties. A GameObject variable has a field 'transform', which is a class of type Transform. You can change the object's position as follows.

```
cube.transform.localPosition = new Vector3(1, 0, 0);
```

Here we create a Vector3 value with the (x, y, z) position we want to assign to the object.

We can also rotate the object.

```
cube.transform.localRotation = Quaternion.Euler(0, 90, 0);
```

Here we give the rotation angles around the x, y, and z axis. In this example we rotate the object 90° around the y axis. It's important to know that when we use Quaternion.Euler(), Unity makes the z rotation first, then the x rotation, then the y rotation. (To see why this matters, examine the difference between rotating an object 90° around the x axis and then 90° around the y axis, and reversing the order of the rotations.)

We can also change the size of an object.

```
cube.transform.localScale = new Vector3(2, 2, 2);
```

Here we give the scale factors by which to change the object size in the x, y, and z directions.

These are just a few of the ways we can manipulate Unity objects from scripts, and we'll see more examples in the course projects.

3.6 Random numbers

Many experimental designs rely on random numbers, for example to choose which stimulus to show on a given trial. The class UnityEngine.Random has a static method Range() that generates a pseudo-random number. This method is *overloaded*, which means that there are several versions of the method, and a different version is called depending on the type of arguments we pass. If we pass two integers, then the method returns a random integer within the given range.

```
using UnityEngine;
```

```
int k = Random.Range(1, 100);
```

Here the lower limit is inclusive and the upper limit is exclusive, so this code produces a number from 1 to 99. If we pass two floating point numbers, we get a random floating point number from the given range.

```
float x = Random.Range(1f, 100f);
```

Note that we can append the letter f to a number to indicate that its type is float. When the arguments are floating point numbers, the lower and upper limits are inclusive: the random number that's returned can range from the first number, up to and including the second number.

We can get an integer from the current time, and use it to seed this random number generator.

```
int rngseed = (int)System.DateTime.Now.Ticks;
Random.InitState(rngseed);
```

Note the type conversion with (int); see Section 2.1 for a reminder of how this works.

We saw in Section 2.7 that the System namespace also defines a Random class that generates random numbers. This is a good example of why namespaces are useful: the two classes have the same name, but even if we're using both namespaces we can resolve the ambiguity by referring to them as UnityEngine.Random and System.Random. The two classes work differently, so it's important to know which one we're using. For example, with System.Random we can create several independent random number generator objects, whereas every time we use UnityEngine.Random.Range(), we're using a single random number generator.

3.7 Writing data to a text file

In an experiment, we usually need to save data into a file, for example recording the subject's responses. A text file is often a good format to use, as it's flexible, can be read by most data analysis programs, and won't be obsolete anytime soon.

First we'll see how to create a string that contains data from variables.

```
int trial_num = 1;
int stimulus_code = 2;
double response_time = 3.45;
string dataline = $"{trial_num},{stimulus_code},{response_time}";
```

This is called *string interpolation*. We put a dollar sign $ before the first quote, and inside the string we include variable names inside set braces. The values of the variables are substituted for the variable names and set braces, so after this code the variable 'dataline' contains the string "1,2,3.45".

To write a string like this to a text file, you can use the System.IO.StreamWriter class.

```
using System.IO;

filename = "data.txt";

using (StreamWriter writer = new StreamWriter(filename, true))
    writer.WriteLine(dataline);
```

The method WriteLine() writes a newline to the file after the string, so if we call it several times, each string appears on a separate line.

There's also a StreamReader class that you can use to read data from a file. This is less common in experiments, but if it's something you need to do, see Chapter 5, where you can find links to online documentation.

3.8 More methods

Here are a few more useful methods. I've given the full name for each function call, but if you've included a 'using' statement such as 'using UnityEngine', then you don't need to specify the namespace (see Section 2.8).

We can print a value in the Unity Editor's console, which is in a tab called 'Console' in the bottom half of the window. Showing values in the console is useful for debugging.

```
float response_time = 3.42f;
UnityEngine.Debug.Log(response_time);
```

We can find the number of seconds since the experiment has started.

```
float t = UnityEngine.Time.time;
```

We can create a string with the current date and time.

```
string timestamp = System.DateTime.Now.ToString("yyyy-MM-dd HH:mm:ss");
```

We can save a screenshot of the currently rendered view.

```
UnityEngine.ScreenCapture.CaptureScreenshot("screenshot.png");
```

After we've started running a program from the Unity Editor, we can end it as follows.

```
UnityEditor.EditorApplication.isPlaying = false;
```

Alternatively, if we've compiled the program and are running it without the Unity Editor, we end it this way.

```
UnityEngine.Application.Quit();
```

To decide which method to use to end the program, we can check whether the program is running from within the Unity Editor.

```
bool inEditor = UnityEngine.Application.isEditor;
```

3.9 Organizing code in separate files

If you create a C# script in the Assets view, and define a new class in that file, then the class is available to other C# scripts in your project. In the course, we'll see examples of how this is a useful way of organizing

code when your projects get more complex, or when you write code that you want to use in more than one project.

Chapter 4

Configuring Unity for VR

This chapter describes how to configure Unity 2022.3 to display rendered images in a VR headset, and how to get input from VR controllers.

4.1 Project settings

In the menu bar, choose Window / Package Manager, which opens the Package Manager window (Figure 4.1). In the top-left corner of this window, there is a drop-down menu whose default selection is "Packages: In Project". From that menu, choose "Unity Registry" instead. In the list of packages on the left side of the window, scroll down and choose "XR Interaction Toolkit". Information about this package will appear in a panel to the right. In that panel, click Install. A progress bar will appear and track progress throughout the installation.

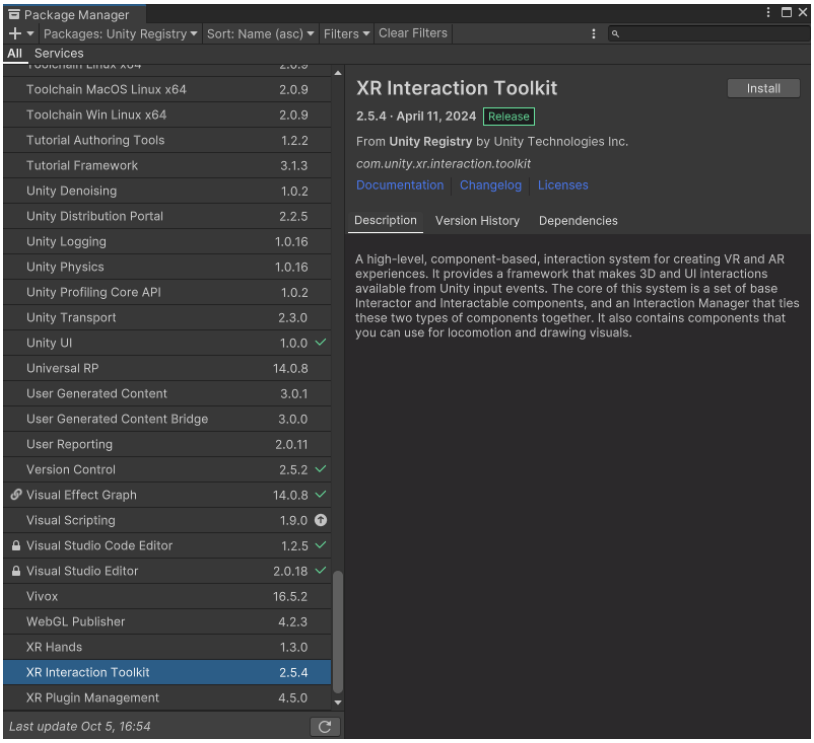

Figure 4.1: Installing the XR Interaction Toolkit through the Package Manger.

Unity will show a warning that begins "This package is using the new input system..." It will ask whether you want to "enable the backends," which will require starting the editor. Click Yes.

Unity will ask whether you want to save the changes you've made to the scene. Click Save. The Unity Editor will restart. After the editor restarts, close the Package Manager window.

In the menu bar, choose Edit / Project Settings..., which opens the Project Settings window (Figure 4.2). In the list of settings on the left, click "XR Plugin Management." (NOT "XR Plug-in Management";

the only difference is a hyphen.) In the panel to the right, click "Install XR Plugin Management". Unity will show a progress bar.

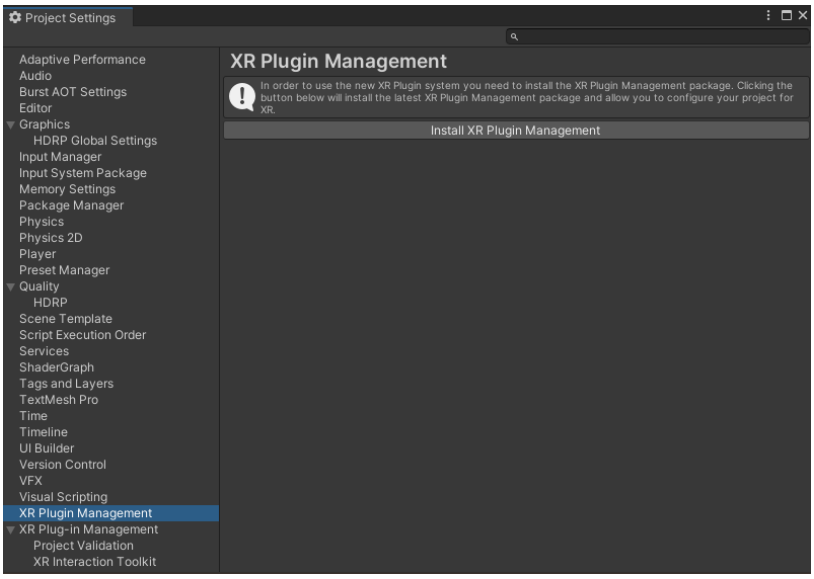

Figure 4.2: Installing XR Plugin Management through the Project Settings.

After that installation has finished, in the list of settings on the left, click "XR Plug-in Management" (now with a hyphen; Figure 4.3). If you're using an Oculus headset, then in the panel to the right, check the box next to "Oculus". Unity will show a progress bar. After that installation is complete, close the Project Settings window.

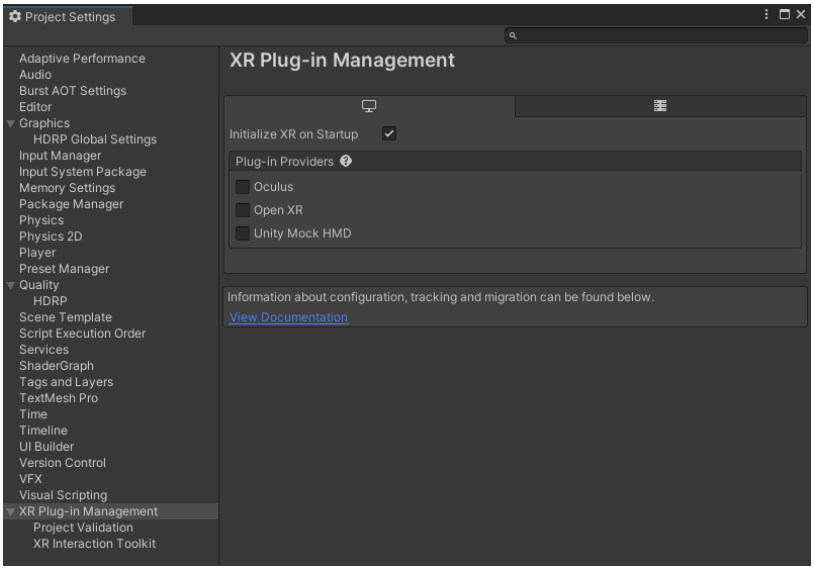

Figure 4.3: Choosing a plug-in provider through the Project Settings.

In the Hierarchy view, right-click, and in the menu that pops up, choose XR / XR Origin (VR). This will create an object called "XR Origin (XR Rig)" in the Hierarchy view, which is the VR camera object.

In the Hierarchy view, delete the object "Main Camera", which is the camera for flat-panel displays.

4.2 VR controllers

To get input from VR controllers, you can use the UnityEngine.XR namespace. Here's a script that shows how, by reporting the right joystick position in the console when the first button on the right controller is pressed.

```
using UnityEngine;
using UnityEngine.XR;

public class MainScript : MonoBehaviour
{
    InputDevice device;

    void Start()
    {
    }

    void Update()
    {
        if (!device.isValid)
        {
            Debug.Log("attempting to get input device");
            device = InputDevices.GetDeviceAtXRNode(XRNode.RightHand);
        }

        bool buttonOK = device.TryGetFeatureValue(CommonUsages.primaryButton,
                                                  out bool buttonValue);
        if (buttonOK && buttonValue)
        {
            bool axisOK = device.TryGetFeatureValue(CommonUsages.primary2DAxis,
                                                    out Vector2 axisValue);
            if(axisOK)
                Debug.Log($"({axisValue.x}, {axisValue.y})");
        }
    }
}
```

You can also check whether the button is pressed using the deprecated UnityEngine.Input system.

```
bool b = Input.GetKeyDown(KeyCode.JoystickButton0);
```

The UnityEngine.XR method returns 'true' as long as the button is pressed, whereas the UnityEngine.Input method returns 'true' only when the button is first pressed.

You can also use the UnityEngine.XR approach to get other features of the controller, such as its position in space.

```
bool posOK = device.TryGetFeatureValue(CommonUsages.devicePosition,
                                       out Vector3 pos);
if (posOK)
    Debug.Log($"({pos.x}, {pos.y}, {pos.z})");
```

See the help for CommonUsages for a full list of the properties you can check.

Chapter 5

Learning more

To learn the basics of the interactive Unity Editor and C# scripting, the first eight chapters of the following book are quite helpful.

Geig, M. (2022). *Sams teach yourself: Unity game development in 24 hours, fourth edition*. Sams Publishing.

It covers the Built-In Render Pipeline (BRP) instead of the High-Definition Render Pipeline (HDRP) that we use in this course. The HDRP has much better graphics, and the BRP will probably be deprecated at some point, so I recommend continuing to learn about and use the HDRP. However, at an introductory level, the differences are not profound, and you'll still probably find Geig's book to be useful.

Unity also provides thorough online documentation in its user manual.

https://docs.unity3d.com/Manual/index.html

In particular, note that in the list of topics at the left, there's a subsection on scripting.

For more complete information about the scripting tools available through C#, you can also consult Unity's scripting reference.

https://docs.unity3d.com/ScriptReference

Online documentation is maintained for each version of Unity, so be sure to check the version number of the documentation you're using, at the top left of the page.

There are also many online tutorials and discussions for Unity, so if you don't find what you're looking for in the documentation, a web search is usually worthwhile. Just make sure to get information for a recent version of Unity, and for the High Definition Render Pipeline.