APPENDIX B

USEFUL CONCEPTS

This appendix is full—like *completely* chock full—of concepts that will help you be a better and more effective prototyper and programmer. Some of these are code concepts, whereas others are methodologies. These are collected here in an appendix to make them easier for you to reference later when you look back at this book in the coming years.

Topics Covered

This appendix covers several different topics, categorized into four distinct groups. Many of these include Unity code examples, and others point you to specific parts of the book where the concept is used.

C# and Unity Coding Concepts

This section covers elements of C# coding that you might want to look back at for a refresher after you've finished the book. There are also some concepts here that, though important, didn't fit well into one of the regular chapters.

Attributes

You've seen some *Attributes* (like `[Header("Inscribed")]`) several times in this book, but there are many other useful ones that you may not have encountered yet. All of these Attributes are applied to the next thing object or field in code (e.g., `[RequireComponent(…)]` on line `// b` applies to the class AttributesEx, and `[Space(32)]` and `[Range(0,1)]` both apply to the `between0and1` field. It does not matter whether or not the Attribute is on the same line as the field. Figure B.1 and Code Listing B.1 together show the Attributes I use most frequently.

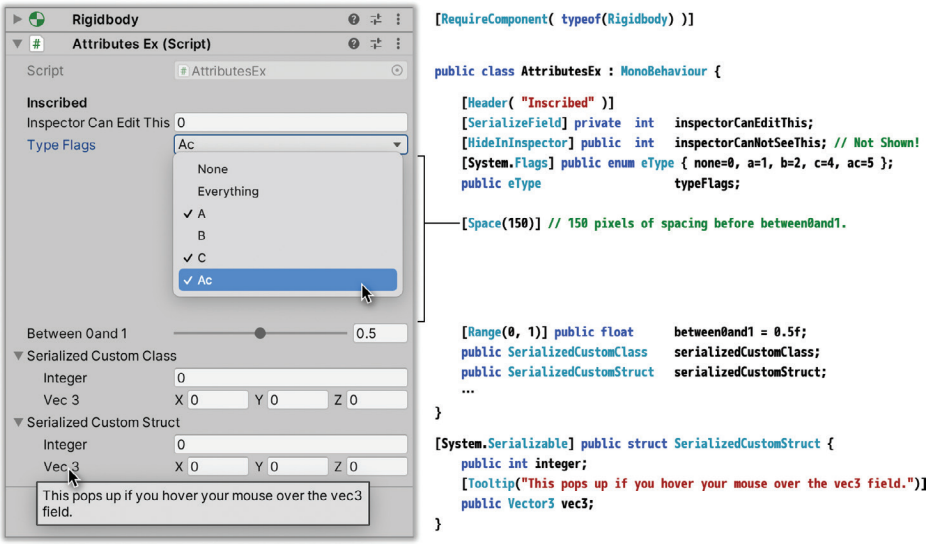


Figure B.1 The effects of the AttributesEx Attributes shown in the Unity Inspector

Code Listing B.1 Attribute Examples

```
> using UnityEngine;                                                    // a
|
> [RequireComponent( typeof(Rigidbody) )]                               // b
> public class AttributesEx : MonoBehaviour {
>     [Header( "Inscribed" )]                                           // c
>     [SerializeField]  private int    inspectorCanEditThis;            // d
>     [HideInInspector] public  int    inspectorCanNotSeeThis; // Not Shown!  // e
>
>     [System.Flags] public enum eType { none=0, a=1, b=2, c=4, ac=5 }; // f
>     public eType                     typeFlags;                       // f
>
>     [Space(150)] // 150 pixels of spacing before between0and1.        // g
>     [Range(0, 1)]       public  float between0and1 = 0.5f;            // h
>     public SerializedCustomClass    serializedCustomClass;           // i
>     public SerializedCustomStruct   serializedCustomStruct;          // j
> }
|
> [System.Serializable] public class SerializedCustomClass {           // i
>     public int     integer;
>     [Tooltip("This pops up if you hover your mouse over the vec3 field.")]  // k
>     public Vector3 vec3;
> }
|
> [System.Serializable] public struct SerializedCustomStruct {         // j
>     public int integer;
```

```
>     [Tooltip("This pops up if you hover your mouse over the vec3 field.")]         // k
>     public Vector3 vec3;
> }
```

a. All of these Attributes require **`using UnityEngine;`** to work.

b. **`[RequireComponent( typeof(Rigidbody) )]`** tells Unity that this MonoBehaviour subclass requires another to also be attached (here, **`AttributesEx`** requires **`Rigidbody`**). When you attach this script to a GameObject, a **`Rigidbody`** will automatically also be attached.

c. **`[Header( "Inscribed" )]`** creates a visible header in the Inspector to separate values.

d. **`[SerializeField]`** causes Unity to make this field visible and editable in the Inspector, even though it is a **`private`** field.

e. **`[HideInInspector]`** causes Unity to hide this field from the Inspector, even though it is **`public`** (Note: The field is still serialized; it's just not visible or editable in the Inspector).

f. **`[System.Flags]`** allows you to create an enum that works like a bitmask. As long as each of the enum values are powers of two (e.g., **a**, **b**, and **c** in **`eType`**), multiple values can be true at the same time. You can also create combination values like **ac=5**, which is equal to the sum of **a** and **c**, causing both **a** and **c** to be true when **`typeFlags`** is set to **ac**. In Unity, the **`[System.Flags]`** attribute shows check marks next to the selected bits as well as *None* and *Everything* (which select none or all of the enum values, respectively); see Figure B.1. Unity uses **`[System.Flags]`** for its LayerMasks (which are covered in the "Bitwise Boolean Operators and Layer Masks" section). There is also a good example of using the **`[System.Flags]`** attribute in Code Listing 31.8 of Chapter 31, "*Space SHMUP* — Part 1."

g. **`[Space(150)]`** creates a 150-pixel gap in the Inspector before the **`between0and1`** field.

h. **`[Range(0,1)]`** creates a slider in the Inspector for setting **`between0and1`** to any number between 0 and 1 (inclusive). The field can still be set outside this range in code.

i. **`[System.Serializable]`** makes the fields of a custom class able to be serialized by Unity, which allows the Inspector to show and edit the fields (and also allows the class and its fields to be converted back and forth to JSON using JSONUtil). Any non-serializable fields (e.g., custom classes that don't have the **`[System.Serializable]`** Attribute) will not be serialized.

j. **`[System.Serializable]`** also works on custom structs.

k. **`[Tooltip( "Tip" )]`** causes the **`"Tip"`** to appear if the mouse is hovered over this field in the Inspector for a couple seconds. I put it here to show that it works on serialized fields of custom classes and structs.

Figure B.1 shows the effects of several of the Attributes in the Unity Inspector window (the **[HideInInspector]** Attribute is not shown, because it's hidden from the Inspector).

Automatic Properties

You've used regular properties—functions that masquerade as fields—throughout this book, but I only introduced *automatic properties* in some later projects. An automatic property is one that is automatically created by C# when compiling your code without your explicitly specifying a backing variable, and they're extremely simple to implement.

Code Listing B.2 Automatic Property Examples

```
| public class AutomaticPropertiesEx : MonoBehaviour {
>     public int intPublicGetSet { get; set; }                    // a
>     public int intPrivateSet   { get; private set; }            // b
>     public int intPrivateGet   { private get; set; }            // c
| }
```

a. A standard automatic property has both a **get** and **set** clause.

b. I will frequently use automatic properties with a **private set** clause like this. This way, any member of the **AutomaticPropertiesEx** class can **set intGetOnly**, but code from other classes can only **get** the value.

c. I've not yet had a use for an automatic property with a **private get** clause and **public set** clause, but you could think of it as a drop box; any other code can **set** this value, but only the **AutomaticPropertiesEx** class can **get** it.

You cannot create a property where one clause is automatic but the other has custom code, and as with all properties, automatic properties are not shown in the Unity inspector. That being said, I still find these useful for creating values with limited access and most frequently use the all-**public** version (**// a**) and the **private set** version (**// b**) in my code.

Bitwise Boolean Operators and Layer Masks

As you learned in Chapter 21, "Boolean Operations and Conditionals," a single pipe (**|**) can be used as a non-shorting conditional OR operator, and a single ampersand (**&**) can be used as a non-shorting conditional AND operator. However, when used with **int**s, they have another important feature. **|** and **&** can be used to perform *bitwise operations* on integers, and are therefore sometimes referred to as *bitwise OR* and *bitwise AND*.

In a *bitwise operation*, the operator is used to affect the individual bits of an integer, and C# includes six different bitwise operators. The following list of them includes the effect that they would have on an 8-bit byte (a simple integral type of data that can hold numbers from 0 to 255). The operations work the same way on 32-bit ints and 64-bit long ints, but neither of those would have fit well on this page.

&	AND	`00000101 & 01000100`	returns `00000100`
|	OR	`00000101 | 01000100`	returns `01000101`
^	Exclusive OR	`00000101 ^ 01000100`	returns `01000001`
~	Complement (bitwise NOT)	`~00000101`	returns `11111010`
<<	Shift Left	`00000101 << 1`	returns `00001010`
>>	Shift Right	`01000100 >> 2`	returns `00010001`

In Unity, bitwise operations are most often used to manage LayerMasks. Unity allows developers to define up to 32 different layers, and a LayerMask is a 32-bit integer representation of which layers to consider in any physics engine or raycast operation. In Unity, the variable type **`LayerMask`** is used for LayerMasks, but it is just a wrapper for a 32-bit int with a little additional functionality. When you use a LayerMask, any bit that is a **1** represents a layer that is seen, and any bit that is a **0** represents a layer that is ignored (i.e., masked). This can be very useful if you want to check collision against only a specific layer of objects or if you want to specify a layer to ignore. (For example, the built-in layer 2, named *Ignore Raycast*, is automatically masked out for all raycast tests.)

Unity now has five reserved "Builtin" layers,[1] and all GameObjects are initially placed in the zeroth (0^{th}) layer, which is named *Default*. The remaining, non-reserved layers (numbered 3 and 6–31) are referred to as *user layers*, and giving one of these a name places it in any pop-up menu of layers (e.g., the Layer pop-up menu at the top of each GameObject Inspector).

Because the layer numbers start at zero, the bitwise LayerMask representation of the zeroth layer is a **1** in the farthest-right position of the LayerMask. (See the variable **`lmZero`** in the following code listing.) This can be a bit confusing (because the integer value of this representation is 1, not 0), so many Unity developers use the bitwise shift left operator (**<<**) to assign LayerMask values. (For example, **`1<<0`** generates the value 1, which is the zeroth layer, and **`1<<4`** generates a 1 in the proper place to mask all but the fourth physics layer.) Code Listing B.3 includes more examples.

1. For reasons of backwards compatibility, these layers are 0–Default, 1–TransparentFX, 2–Ignore Raycast, 4–Water, and 5–UI. For more info, see https://docs.unity3d.com/2020.3/Documentation/Manual/Layers.html .

Code Listing B.3 LayerMask Examples (1s Are Bolded to Make Them More Visible)

```
public class BitwiseEx : MonoBehaviour {
    void Start () {
        LayerMask lmNone = 0;    // 00000000000000000000000000000000 bitwise        // a
        LayerMask lmAll  = ~0;   // 11111111111111111111111111111111 bitwise        // b
        LayerMask lmZero = 1;    // 00000000000000000000000000000001 bitwise        // c
        LayerMask lmOne  = 2;    // 00000000000000000000000000000010 bitwise        // d
        LayerMask lmTwo  = 1<<2; // 00000000000000000000000000000100 bitwise        // e
        LayerMask lmThree= 8;    // 00000000000000000000000000001000 bitwise        // f

        LayerMask lmZeroOrTwo = lmZero | lmTwo;                                     // g
                  // Results in 00000000000000000000000000000101 bitwise

        LayerMask lmZeroThroughThree = lmZero | lmOne | lmTwo | lmThree;
                  // Results in 00000000000000000000000000001111 bitwise

        lmZero = 1 << LayerMask.NameToLayer( "TransparentFX" );                     // h
                  // Results in 00000000000000000000000000000010 bitwise

        LayerMask lmZeroOrOne = LayerMask.GetMask("Default","TransparentFX");       // i
                  // Results in 00000000000000000000000000000011 bitwise

        Debug.Log( lmZeroOrOne );                                                   // j
                  // Prints: "UnityEngine.LayerMask"
        Debug.Log( "lmZeroOrOne: " + lmZeroOrOne.ToStringPretty() );
                  // Prints: "lmZeroOrOne: 00000000000000000000000000000011"
        Debug.Log( "lmZeroOrTwo: " + lmZeroOrTwo.ToStringPretty('-') );
                  // Prints: "lmZeroOrTwo: -----------------------------1-1"
    }
}

// This is an example of an Extension Method (see later in Appendix B!)
static public class LayerMaskPrettyPrintExtension {                                 // k
    public static string ToStringPretty( this LayerMask lMask, char pad='0' ) {
        string str = System.Convert.ToString( lMask, 2 );
        if ( pad != '0' ) str = str.Replace( '0', pad );
        return str.PadLeft( 32, pad );
    }
}
```

a. When all bits are set to 0, the LayerMask will *ignore* all layers.

b. When all bits are set to 1, the LayerMask will *interact* with all layers.

c. Note that 1 is the actual integer value of the LayerMask for layer 0 (in math $2^0 = 1$).

d. Similarly, 2 is the integer value of the LayerMask for layer one ($2^1 = 2$), which demonstrates how it can get confusing to assign LayerMask values using integers. Layer one is the predefined "TransparentFX" layer in Unity.

e. Using the shift left operator (**<<**) makes a lot of sense in this case because the 1 is shifted two places to the left to create a LayerMask for the second layer.

f. Layer 3 of the LayerMask has an integer value of 8, which is 2^3 or **1<<3**.

g. A bitwise OR is used to make a LayerMask that collides with either layer 0 or layer 2.

h. The static method **LayerMask.NameToLayer()** returns a layer number—an int number, not a LayerMask—when it is passed a layer name. For example, **LayerMask.NameToLayer("TransparentFX")** returns the int 1.

i. You can also go directly from a list of layer names to a LayerMask using **GetMask()**.

j. As you can see in the comment on the next line, calling **Debug.Log()** on a LayerMask prints **"UnityEngine.LayerMask"** to the console, which is useless. So I've created an extension method here (defined at **// k**) to "pretty print" the LayerMask as a bitwise representation of the number.

k. As covered by the "Extension Methods" section of this appendix, an *extension method* like this adds functionality to an existing class without modifying the code of that class. Here, we use it to add the **ToStringPretty()** method to all **LayerMask** instances (as used at **// j**). The **System.Convert.ToString(lMask, 2)** call converts the LayerMask **lMask** to binary (the **2**) and returns a **string**, which for **lmZeroOrTwo** is **"101"**. If a padding character is passed in, any zeros in **str** are replaced with the **pad** character. The **PadLeft()** call then pads the string to at least 32 characters, prepending the **char pad** (**'0'** by default). See the results of two example calls at **// j**.

Coroutines

A coroutine is a feature of C# that enables a method to pause execution on a specific **yield** line, allow other processes to execute, and then return to execution of the paused method from exactly where it left off. In Unity, coroutines are often used when the execution of a single function could take a very long time (and make the game look like it had frozen). One example of this is the "Dice Probability Enumeration" section later in this appendix; the function to calculate all the possible outcomes of rolling many dice could take minutes or even hours to run, so pausing in the middle to let the rest of your application (and the Unity Editor) update is very helpful. You can also use coroutines as timers for tasks that you want to happen on a repeating schedule (as an alternative to using an **InvokeRepeating()** call).

Unity Example—Coroutines

This example coroutine prints the time once every second. A call to print the time in the **Update()** method would print it dozens of times per second, which is far too many.

To test this in Unity, create a C# script named *Clock* that is attached to Main Camera, and then enter the code in Code Listing B.4.

Code Listing B.4 Clock.cs — Coroutines Example

```
using System.Collections;  // This is required for IEnumerator (used at // b)
using UnityEngine;

public class Clock : MonoBehaviour {
    void Start () {
        StartCoroutine( Tick() );                                    // a
    }

    IEnumerator Tick() {                                             // b
        while (true) {                                               // c
            print(System.DateTime.Now.ToString());
            yield return new WaitForSeconds(1);                      // d
        }
    }
}
```

a. Use **`StartCoroutine()`** to start any coroutine. This is also where parameters can be passed into the coroutine method if needed (I don't use the version of **`StartCoroutine()`** where the name of the coroutine method is passed as a string because the precompiler doesn't catch method name typos in the string version).

b. All coroutines have a return type of **`IEnumerator`**.[2]

c. This infinite **`while`** loop will keep the print happening until the **`Tick()`** coroutine is exited or the whole program is stopped.

d. This **`yield`** statement tells the coroutine to wait about 1 second before continuing. I say "about" 1 second because coroutine timing is not perfectly exact.

Unlike a normal function, it is okay to use the **`while(true)`** infinite loop within a coroutine as long as there is also a **`yield`** within the **`while`** loop (as seen in Code Listing B.4).

There are a few different kinds of **`yield`** statements, including:

```
yield return null;                      // Will continue as soon as possible
yield return new WaitForSeconds(10);    // Will wait 10 seconds
yield return new WaitForEndOfFrame();   // Will wait until the next Update()
yield return new WaitForFixedUpdate();  // Will wait until the next FixedUpdate()
```

2. Of course, you don't need to know what an **`IEnumerator`** is to use coroutines, but in case you're curious, C# uses the **`interface IEnumerator`** (or **`IEnumerable`**) for any class that can be iterated over (e.g., **`List`**<>). An **`IEnumerator`** keeps track of where it is in a collection and can be asked to move on to the next iteration. For coroutines, the elements in the "collection" are the chunks of code in between **`yield`** statements, so C# uses the **`IEnumerator`** to keep track of both its place in the coroutine function and when the function is complete.

It is also possible to use **`Start()`** as a coroutine by making the return type **`IEnumerator`** instead of **`void`**; however, after the first time you **`yield`**, the timing of the remainder of the **`Start()`** coroutine will then be executed just like any other coroutine. Code Listing B.5 shows the use of **`Start()`** as a coroutine in the **`ClockStart`** class. Similarly, you can also use **`Update()`** as a coroutine, but I don't have any good examples of when that would be a good idea.

Code Listing B.5 ClockStart.cs — **`Start()`** as a Coroutine

```
public class ClockStart : MonoBehaviour {
    IEnumerator Start() {
        while ( true ) {
            print( System.DateTime.Now.ToString() );
            yield return new WaitForSeconds( 1 );
        }
    }
}
```

Delegates, Events, and UnityEvents

A *function delegate* is most simply thought of as a container for similar functions (or methods) that can all be called at once. You can see delegates used in Chapter 32, "*Space SHMUP* — Part 2," to enable a single call to the **`delegate event`**[3] **`fireEvent()`** to fire all weapons attached to a player's ship (Code Listing 32.18). Delegates are frequently used to implement *Strategy pattern* for use in game AIs. You can learn more about Strategy pattern in the "Object-Oriented Software Design Patterns" section of this appendix.

The first step of using a function delegate is to define the *delegate type* (as shown in the following line of code). The delegate type definition dictates the return type and parameter types required for any function to be assigned to an instance of this delegate type.

```
public delegate float FloatOpDelegate( float f0, float f1 );
```

This line creates a delegate definition that requires two floats as input and a single float as the return type. After the definition is set, you can define *target methods* that fit this delegate definition. Code Listing B.6 shows the declaration of a delegate type and definition of two target methods that match the delegate type (**`FloatAdd()`** and **`FloatMultiply()`**).

3. As discussed in Chapter 32, an **`event`** is a wrapper for a **`delegate`** that protects the delegate by only allowing other classes to add or remove their own methods from it (a regular delegate can be set to **`null`** by other classes, which is pretty dangerous to allow). When a delegate variable is wrapped in an event, it is usually just referred to as an "event."

Code Listing B.6 DelegateExample.cs — Initial Setup

```
| using UnityEngine;
|
| public class DelegateExample : MonoBehaviour {
>     // Create a delegate definition named FloatOpDelegate
>     // This defines the parameter and return types for target functions
>     public delegate float FloatOpDelegate( float f0, float f1 );
>
>     // FloatAdd parameter types and return type must match FloatOpDelegate
>     public float FloatAdd( float f0, float f1 ) {
>         float result = f0+f1;
>         print("The sum of "+f0+" & "+f1+" is "+result+".");
>         return( result );
>     }
>
>     // Target method types must match FloatOpDelegate, but param names can differ
>     public float FloatMultiply( float num0, float num1 ) {
>         float result = num0 * num1;
>         print("The product of "+num0+" & "+num1+" is "+result+".");
>         return( result );
>     }
| }
```

Now, a variable of the type **`FloatOpDelegate`** can be created, and either of the target methods can be assigned to it. I prefer to use the **`event`** keyword when declaring a delegate variable to add a layer of protection.[3] This event can then be called just like a function. In Code Listing B.7, the field **`foDelegate`** is a delegate event variable. Note that because **`foDelegate`** is a variable (and not a true function), its name begins with a lowercase letter.

Code Listing B.7 DelegateExample.cs — Defining and Using **`foDelegate`**

```
| public class DelegateExample : MonoBehaviour {
|     …
|     public delegate float FloatOpDelegate( float f0, float f1 );
|
|     public float FloatAdd( float f0, float f1 ) { … }
|
|     public float FloatMultiply( float f0, float f1 ) { … }
|
>     // Declare a field "foDelegate" of the type FloatOpDelegate
>     public event FloatOpDelegate foDelegate; // Declare a delegate event field
>
>     void Awake() {
>         // Assign the method FloatAdd() to foDelegate
>         foDelegate = FloatAdd;
>
```

```
>         // Call foDelegate as if it were a method; which then calls FloatAdd()
>         foDelegate( 2, 3 ); // Prints: The sum of 2 & 3 is 5.
>
>         // Assign the method FloatMultiply() to foDelegate, replacing FloatAdd()
>         foDelegate = FloatMultiply;
>
>         // Call foDelegate(2,3); it calls FloatMultiply(2,3), returning 6
>         foDelegate( 2, 3 ); // Prints: The product of 2 & 3 is 6
>     }
| }
```

Delegates can also be *multicast*, which means that more than one target method can be assigned to the delegate at the same time. This is the ability that allows a single call to one event to fire all five Weapons in the Chapter 32, "*Space SHMUP* — Part 2," prototype. There, a single call to the **`fireEvent()`** delegate event in turn calls all the **`Fire()`** methods of the various Weapons on the player's ship. If a multicast delegate has a return type that is not void (as in the **`FloatOpDelegate`** example), the return value of the *final* target method called will be returned by the call to the delegate.

Beware that if a delegate or event that has no target methods attached is called, it will throw an error. Prevent this by first checking to see whether the delegate or event is **`null`** before calling it. Code Listing B.8 demonstrates both a multicast use of **`foDelegate`** and a check to see whether **`foDelegate`** is **`null`** before calling it.

Code Listing B.8 DelegateExample.cs — Multicast Delegates and Checking for Null

```
| public class DelegateExample : MonoBehaviour {
|     …
|     void Awake() { … }
|
>     void Start() {
>         // Assign the method FloatAdd() to foDelegate
>         foDelegate = FloatAdd;
>
>         // Add the method FloatMultiply(), now BOTH are called by foDelegate
>         foDelegate += FloatMultiply;
>
>         // Check to see whether foDelegate is null before calling
>         if (foDelegate != null) {
>             // Call foDelegate(3,4); it calls FloatAdd(3,4) & FloatMultiply(3,4)
>             float result = foDelegate( 3, 4 );
>             // Prints: The sum of 3 & 4 is 7.
>             // then Prints: The product of 3 & 4 is 12.
>
>             print( "The result of the Delegate call is "+result );
>             // Prints: The result of the Delegate call is 12
```

```
>             // The result is 12 because the last target method to be called
>             //   is the one that returns a value via the delegate.
>         }
>     }
| }
```

See the "Lambda Expressions =>" section of this appendix for information on how lambda expressions are used with delegates.

UnityEvents

UnityEvents are a special form of delegate event in Unity that are able to be set easily in the Inspector pane. Please see the Observer Pattern (under "Object-Oriented Software Design Patterns") in this appendix for an extensive example of how to use **UnityEvents**.

Enums

An enum is a simple way to declare a type of variable that only has a few specific options, and it is used throughout the book. Enums in this book are usually defined outside of class definitions (either the multiline or single-line definition is fine, as shown in the following lines).

```
public enum ePetType {   // Enum type names often start with a lowercase e
    none,
    dog,
    cat,
    bird,
    fish,
    other
}

public enum eLifeStage { baby, teen, adult, senior, deceased }
```

Later, a variable can be declared using the enum's type (e.g., **public ePetType**). The various options for an enum are referred to by the enum type, a dot, and the enum value (e.g., **ePetType.dog**, as shown in Code Listing B.9).

Enums are actually integers masquerading as other values, so they can be cast to or from **int** (as shown at **// b** and **// c** in Code Listing B.9). Because it is an **int**, any enum defaults to the 0th option if not explicitly set. For example, declaring a new variable **eLifeStage age** (as in Code Listing B.9 **// a**) automatically assigns **age** the default value of **eLifeStage.baby**.

Code Listing B.9 Pet.cs — Enums Example

```
| public class Pet : MonoBehaviour {
|     public string       name = "Flash";
|     public ePetType     pType = ePetType.dog;
|     public eLifeStage   age; // By default, age=0, i.e., eLifeStage.baby      // a
|
|     void Awake() {
|         int i = (int) ePetType.cat;     // i would equal 2                    // b
|         ePetType pType = (ePetType) 4;  // pType would equal ePetType.fish    // c
|     }
| }
```

a. **age** receives the default value of **eLifeStage.baby**, which is equal to 0.

b. The code **(int)** shown on line 7 is an explicit typecast that forces **ePetType.cat** to be interpreted as an int.

c. Here, the **int** literal **4** is explicitly typecast to a **ePetType** by the code **(ePetType)**.

Enums are often used in **switch** statements (as you've seen throughout this book). For an example of defining a **public enum** within a class, see "Easing for Linear Interpolations" under "Interpolation" in this appendix.

The **[System.Flags]** attribute can be used on an enum to give it several options that can be selected at the same time (the Unity **LayerMask** is an example). For examples in Appendix B, see "Attributes" and "Observer Pattern" under "Object-Oriented Software Design Patterns." There is also an example in Code Listing 31.8 of Chapter 31, "*Space SHMUP* — Part 1."

Extension Methods

Although Unity's built-in classes are locked, C# does provide a way to add more methods to them as *extension methods*. If you create a **static** class, within that class, you can define methods that will seem (to the rest of your code) to be part of the built-in class. An excellent example of this is the **ToStringPretty()** method that we added to the **LayerMask** class at **// k** in Code Listing B.3. Code Listing B.10 repeats the **ToStringPretty()** definition and shows two calls to it.

Code Listing B.10 LayerMaskPrettyPrintExtension Class

```
| static public class LayerMaskPrettyPrintExtension {                          // a
|     public static string ToStringPretty(this LayerMask lMask, char pad='0') {  // b
|         string str = System.Convert.ToString( lMask, 2 );
|         if ( pad != '0' ) str = str.Replace( '0', pad );
|         return str.PadLeft( 32, pad );
|     }
```

```
| }
|
| LayerMask lmZeroOrTwo = 1<<0 | 1<<2;  // "<<0" doesn't actually change anything
| Debug.Log( "lmZeroOrTwo: " + lmZeroOrTwo.ToStringPretty() );                    // c
| // Prints: "lmZeroOrTwo: 00000000000000000000000000000101"
| Debug.Log( "lmZeroOrTwo: " + lmZeroOrTwo.ToStringPretty('-') );                 // d
| // Prints: "lmZeroOrTwo: ----------------------------1-1"
```

a. Extension methods can only be defined within a **public static** class. When a class is static, no instances are allowed to be created from it (just like a static field does not have multiple instances). The name of the static class does not matter, so I tend to name them to clarify the class I'm extending. It is also possible to extend multiple classes in one static class, but I prefer using a single static class for all extensions of a single extant class.

b. The **this** keyword on the first parameter allows the extension method to be called as if it is a method of the **LayerMask** class, as shown at **// c** and **// d**. The **pad** parameter demonstrates how to use optional parameters in extension methods. For more on optional parameters, see Chapter 24, "Functions and Parameters."

c. This line calls the **ToStringPretty()** extension method without the optional **pad** parameter, allowing **pad** to default to **'0'** (the zero character).

d. This call to **ToStringPretty()** does pass in the optional **pad** parameter.

Interfaces

An **interface** declares methods and properties that will then be implemented by a class. Think of an interface as a promise of functionality that can be kept by any class or struct. Fields cannot be required by an interface, but properties and methods can. Any class or struct that implements the interface can be referred to in code as that interface type rather than as the class that it actually is.

Implementing an interface differs from subclassing (i.e., extending a class) in several ways, one of the most interesting of which is that *a class may implement several different interfaces simultaneously,* whereas a class can only extend a single superclass. Interface names often begin with a capital **I** to distinguish them from class names.

Unity Example—Interfaces

Let's look at an example of using interfaces in Unity.

1. Create a new project in Unity named *InterfacesExample Project*.
2. In that project, create a C# script named *InterfacesExample*.
3. Attach the *InterfacesExample* script to the *Main Camera* in the scene.

4. Open the *InterfacesExample* script in Visual Studio and enter the code shown in Code Listing B.11. Please add the **`interface IMovable`** definition above the **`Interfaces Example`** class definition (which you do not need to edit for now).

Code Listing B.11 InterfacesExample.cs — The IMovable Interface

```
| using System.Collections.Generic;
| using UnityEngine;
|
> public interface IMovable {                                          // a
>     // Public Properties
>     System.Type type { get; }                                        // b
>     Vector3     loc  { get; set; }                                   // c
>
>     // Public Methods
>     void        Move();                                              // d
> }
|
| public class InterfacesExample : MonoBehaviour {…}
```

   a. Interfaces need to be declared public and usually start with a capital **`I`**.

   b. Here, the interface promises a getter property named **`type`** that will return a **`System.Type`** (the type of the class or struct that implements **`IMovable`**).

   c. A **`Vector3 loc`** property with both **`get`** and **`set`** clauses is also promised.

   d. Finally, a **`Move()`** method with no parameters that returns **`void`** is also promised.

The **`public`** keyword is not needed for the properties and methods promised by an interface because *everything* promised by an interface must be public to fulfill that promise. Next, let's look at two very different objects that implement **`IMovable`**.

5. Add the **`struct AnimalStruct`**, **`class Droid`**, and **`class TurboDroid`** to the top of the InterfacesExample script, as shown in Code Listing B.12.

Code Listing B.12 InterfacesExample.cs — Droid and TurboDroid classes and AnimalStruct

```
| public interface IMovable {                                          // a
|     // Public Properties
|     System.Type type { get; }
|     Vector3     loc  { get; set; }
|
|     // Public Methods
|     void        Move();
| }
|
```

```
> [System.Serializable]
> public class Droid : IMovable {                                    // b
>     public Vector3 myPosition;
>     public Vector3 myVelocity = Vector3.right;
>
>     public System.Type type => this.GetType();                     // c
>
>     public Vector3 loc {
>         get { return myPosition; }
>         set { myPosition = value; }
>     }
>
>     virtual public void Move() {
>         myPosition += myVelocity * Time.time;
>     }
> }
|
> [System.Serializable]
> public class TurboDroid : Droid {                                  // d
>     public bool turbo = true;
>
>     override public void Move() {
>         base.Move();
>         if ( turbo ) base.Move();
>     }
> }
|
> [System.Serializable]
> public struct AnimalStruct : IMovable {                            // e
>     public Vector3 position;
>     public Vector3 velocity;
>
>     public System.Type type {
>         get { return this.GetType(); }
>     }
>
>     public Vector3 loc {
>         get { return position; }
>         set { position = value; }
>     }
>
>     public void Move() {
>         loc += velocity * Time.time;
>     }
> }
|
| public class InterfacesExample : MonoBehaviour {…}
```

a. I repeat the **IMovable** interface definition in this code listing so that you can see how the other objects implement it (without your having to flip back to Code Listing B.11).

b. The serializable **class Droid** implements **IMovable**, providing all the properties and methods promised by **IMovable**.

c. This is a *lambda expression* property (see "Lambda Expressions =>" here in Appendix B). When **DroidInstance.type** is requested, **this.GetType()** is returned.

d. The **class TurboDroid** extends (i.e., subclasses) the **class Droid**. One of the things inherited from **Droid** is its implementation of **IMovable**. Because the **Move()** method of **Droid** is **virtual**, it can be overridden here in **TurboDroid** with a **Move()** method that calls **base.Move()** (the **Move()** method on **Droid**) twice if **turbo == true**.

e. **AnimalStruct** implements **IMovable** as a **struct**, which is a different type of data from a **class** (see "Structs" in this appendix). As you can see, it can implement interfaces and define properties and methods just like a class, but it cannot extend another **struct**.

As you can see, both classes and structs can implement the same interface, and I've intentionally made the internals of **Droid** and **AnimalStruct** quite different from each other. The magic of interfaces is this: The only thing the rest of your code needs to know about anything that implements **IMovable** is that it can be trusted to fulfill all the promises of **IMovable**. We can see why that is so powerful in Code Listing B.13.

6. It's time to implement the **InterfacesExample** class.

Code Listing B.13 InterfacesExample.cs — Working with **IMovables**

```
| public interface IMovable {…}
| …
| public struct AnimalStruct : IMovable {…}
|
| public class InterfacesExample : MonoBehaviour {
>     public List<IMovable> movables;
>     System.Text.StringBuilder sb = new System.Text.StringBuilder();                // a
|
|     void Start() {
>         movables = new List<IMovable>();
>
>         movables.Add( new AnimalStruct { velocity = Vector3.right } );             // b
>         movables.Add( new Droid() );                                               // c
>         TurboDroid turboDroid = new TurboDroid();                                  // d
>         turboDroid.turbo = true;
>         movables.Add( new TurboDroid() );
|     }
```

```
|
|     void Update() {
>         sb.Clear();
>         foreach (IMovable movable in movables) {                                    // e
>             movable.Move();                       // Move() is called
>             sb.Append( movable.type.ToString() ); // Append its type as a string
>             sb.Append( ".x=" );
>             sb.Append( movable.loc.x );           // Append the x element of loc
>             sb.Append( '\t' );                    // Append a tab
>             if (movable is TurboDroid) {                                            // f
>                 (movable as TurboDroid).turbo = !(movable as TurboDroid).turbo;
>             }
>         }
>         Debug.Log( sb );
|     }
| }
```

a. The **System.Text.StringBuilder** class is much more efficient than using **+** to concatenate several strings. See "System.Text.StringBuilder" in this appendix for more information.

b. A new instance of the **AnimalStruct** struct is created, with its **velocity** set to [0, 0, 1]. It is added to the **List** **movables** on the same line.

c. Similarly, a new **Droid** instance is created and added to **movables**.

d. We want to set **turbo=true** on the new **TurboDroid** instance, but that isn't something we can do if we're treating it as an **IMovable**. So **turboDroid** is created as a **TurboDroid** instance, **turbo** is set to **true**, and then **turboDroid** is added to **movables**.

e. Every **IMovable** is treated the same regardless of type.

f. This is quite verbose, but it shows how to use **is** and **as** with classes or structs that implement an interface. **is** returns **true** if this **IMovable** is actually a **TurboDroid** (which is only true for the last **IMovable** in **movables**). Then, if it **is** a **TurboDroid**, we treat it **as** a **TurboDroid** to toggle its **turbo** value. The effect this will have is that the **TurboDroid** will alternate moving at 1x or 2x the speed of the other **IMovable**s.

7. Save this script and return to Unity.

8. Click the *Pause* button first, and then click *Play*. This will allow you to step through one frame at a time using the *Step* button (immediately to the right of the Pause button). (When stepping each frame like this, the value of **Time.deltaTime** will always be 0.1f, so time-based code acts as if the game is running at 10 frames per second.)

You should see something similar to the following lines printed to the Console in the first few frames.

```
AnimalStruct.x=0.00    Droid.x=0.00    TurboDroid.x=0.00
AnimalStruct.x=0.02    Droid.x=0.02    TurboDroid.x=0.02
AnimalStruct.x=0.06    Droid.x=0.06    TurboDroid.x=0.10    <===<<
AnimalStruct.x=0.12    Droid.x=0.12    TurboDroid.x=0.16
AnimalStruct.x=0.20    Droid.x=0.20    TurboDroid.x=0.32    <===<<
AnimalStruct.x=0.30    Droid.x=0.30    TurboDroid.x=0.42
```

You can see in the lines marked by a left arrow (<===<<) that the **x** position of the **TurboDroid** has increased at double the pace of the other **IMovable**s because **turbo** == **true** in those frames.

Unity Makes Frequent Use of Interfaces for Observer Pattern

As you can read about in the "Object-Oriented Software Design Patterns" section of this appendix, the Observer pattern sets up a broadcaster that sends specific messages out when events occur. Over the last decade, Unity has greatly increased its use of interfaces and Observer pattern. Two excellent examples of this are responding to player input in both uGUI (Unity's graphical user interface system) and Unity's new InputSystem.[4] The following is an example of using Observer pattern to implement mouse dragging of objects in uGUI.

1. To try this code, you will need to have the Unity UI package installed. This should be the case by default, but to double check:
   - From the Unity menu bar, choose *Window* > *Package Manager*.
   - By default, this should show all the packages in your project (the menu under the mouse cursor in Figure B.2a should show *Packages: In Project*). If you **do** see the *Unity UI* package listed on the left, then you can skip the rest of these bullet points.
   - If you **do not** see *Unity UI* on the left, then you need to install it.
   - Click the *Packages: In Project* menu at the top of the Package Manager window and choose *Unity Registry*.
   - Scroll down the list on the left and select *Unity UI*.
   - Click the *Install* button in the bottom-right corner of the window to install the Unity UI package. Once it is installed, close the Package Manager window.

4. uGUI is used throughout this book, but at this point, I have chosen to use the traditional Unity **Input** class rather than the new InputSystem. InputSystem is much more robust and works better cross-platform (making input consistent on everything from Windows to macOS to PlayStation), but it takes much more work to set up and is still changing. Unity also has a new UI Toolkit that will eventually replace uGUI. UI Toolkit is more like the Cascading Style Sheets (CSS) that are common in web pages, but for now, the uGUI implementation is much more robust and mature.

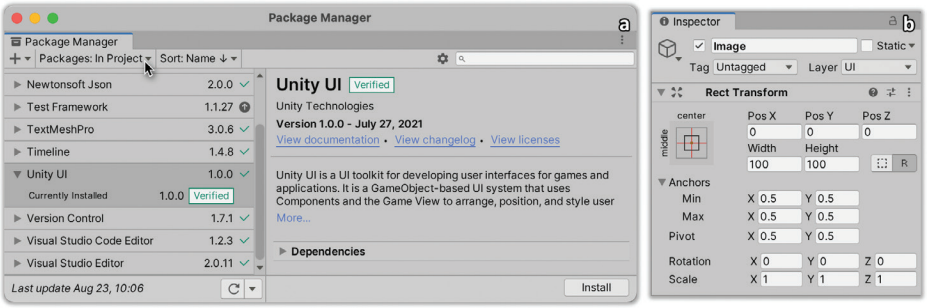


Figure B.2 The Package Manager (step 1) and Rect Transform settings for Image (step 5)

2. In the same Unity project, create a new scene and save it.
3. In that scene, create a uGUI Image by choosing *GameObject* > *UI* > *Image* from the main Unity menu. This will add a few things to the Hierarchy:
   - **EventSystem**: A GameObject that manages events for the GUI.
   - **Canvas**: A Canvas to contain uGUI GameObjects.
   - **Image**: A child of Canvas that we will use for this example.
4. This should automatically select the new Image, but if it does not:
   a. In the Hierarchy, click the *disclosure triangle* next to *Canvas*.
   b. Select the *Image* GameObject that is a child of Canvas.
5. Set the *Rect Transform* of *Image* to the settings shown in Figure B.2b. This will position the Image (a white box) in the middle of the Game pane.
6. Create a C# script named *DraggableImage* and attach it to the *Image* GameObject.
7. Open the *DraggableImage* script in Visual Studio and enter the code in Code Listing B.14.

Code Listing B.14 DraggableImage.cs

```
using UnityEngine;
using UnityEngine.EventSystems;                                    // a
using UnityEngine.UI;                                              // b

public class DraggableImage : MonoBehaviour, IBeginDragHandler,    // c
    IDragHandler, IEndDragHandler {                                // c

```

```
>     Vector2 mouseDownPosition;  // The screen position of the MouseDown
>     Vector2 beginDragPosition;  // The start position of this Image
>     Image image;                // A reference to the Image Component
|
>     public void OnBeginDrag( PointerEventData eventData ) {             // d
>         if ( image == null ) image = GetComponent<Image>();
>         image.color = Color.gray;
>         mouseDownPosition = eventData.position;
>         beginDragPosition = transform.position;
>     }
|
>     public void OnDrag( PointerEventData eventData ) {                  // e
>         Vector2 deltaFromBegin = eventData.position - mouseDownPosition;
>         transform.position = beginDragPosition + deltaFromBegin;
>     }
|
>     public void OnEndDrag( PointerEventData eventData ) {               // f
>         image.color = Color.white;
>     }
| }
```

a. **`UnityEngine.EventSystems`** contains definitions for all the interfaces implemented in this example.

b. **`UnityEngine.UI`** defines the **`Image`** class and all other uGUI classes.

c. This one single, very long line wrapped to fit on the page. After **`MonoBehaviour`** (which is the superclass extended by **`DraggableImage`**), three interfaces are added, one for each method that will be implemented.

d. When the primary mouse button is pressed on the image, **`OnBeginDrag()`** is called. First, if **`image`** is **`null`**, it is set to the **`Image`** component on this GameObject. Then, **`image.color`** is set to gray, and the starting values for **`mouseDownPosition`** and **`beginDragPosition`** are set as well.

e. **`OnDrag()`** is called (nearly) every frame between **`OnBeginDrag()`** and **`OnEndDrag()`**. In our code, it updates the position of this GameObject. We use the two different initial positions so that the relative position of the Image to the mouse stays the same.

f. When the primary mouse button is released, **`OnEndDrag()`** is called, and the color of **`image`** is reset to white.

The Unity UI package contains many of these interfaces, including an **`IDropHandler`** interface that is called on any uGUI object that is underneath a dragged GameObject when **`OnEndDrag()`** is called (e.g., with **`IDropHandler`**, you could make a bucket that you drag objects into). You can find more about this in the Unity documentation.[5]

JSON (JavaScript Object Notation) in Unity

JSON (pronounced Jay-Sahn) is used in Unity to serialize data (i.e., to turn simple data types or classes and structs that are **`[System.Serializable]`** into plain text). This is very useful for encoding data in configuration or save files and—much like the XML (eXtensible Markup Language) you can read about later in this appendix—JSON is designed to be human readable.

In the first and second editions of the book, I used XML to encode the Deck and Layout information for the *Prospector Solitaire* prototype (in Chapters 33 and 34), but in this third edition, I switched to JSON primarily because Unity now provides a **`JSONUtility`** class that makes converting to and from JSON extremely easy (you can see **`JSONUtility`** in action in Chapter 33, "*Prospector Solitaire* — Part 1").

JSON comes in both pretty and minified versions, as shown in JSON Listings B.15 and B.16. Both JSON listings show the same data (the card information for the deck of cards used in *Prospector Solitaire*), but JSON Listing B.15 has several line breaks to make it *pretty* (i.e., easier for humans to read), while JSON Listing B.16 shows the same data with any extra whitespace removed to make it as compact as possible (for faster transmission over the Internet).

I have added `//` comments in JSON Listing B.15, but be aware that actual JSON does not allow any comments. The syntax coloring in these JSON listings is the same as that for JSON files opened in Visual Studio Code, which I find to be excellent for opening and editing JSON files. (Note that although the names are similar, Visual Studio Code is a *completely different* program from the Visual Studio that installs with Unity.)

5. You can see a list of all the supported interfaces in the Unity UI package at https://docs.unity3d.com/Packages/com.unity.ugui@1.0/manual/SupportedEvents.html — accessed August 24, 2021.

JSON Listing B.15 JSON_Deck.json from *Prospector Solitaire* (in three columns)

```
{
  "decorators": [  // a
    {
      "type": "letter",
      "loc": {
        "x": -1.05,
        "y": 1.42,
        "z": 0.0
      },
      "flip": false,
      "scale": 1.25
    },
    {
      "type": "suit",
      "loc": {
        "x": -1.05,
        "y": 1.03,
        "z": 0.0
      },
      "flip": false,
      "scale": 0.4
    },
    {
      "type": "letter",
      "loc": {
        "x": 1.05,
        "y": -1.42,
        "z": 0.0
      },
      "flip": true,
      "scale": 1.25
    },
    {
      "type": "suit",
      "loc": {
        "x": 1.05,
        "y": -1.03,
        "z": 0.0
      },
      "flip": true,
      "scale": 0.4
    }
  ],
  "cards": [        // b
    {
      "rank": 1,
      "face": "",
      "pips": [     // c
        {
          "type": "pip",
          "loc": {
            "x": 0.0,
            "y": 0.0,
            "z": 0.0
          },
          "flip": false,
          "scale": 2.0
        }
      ]
    },
    {
      "rank": 2,
      "face": "",
      "pips": [
        {
          "type": "pip",
          "loc": {
            "x": 0.0,
            "y": 1.1,
            "z": 0.0
          },
          "flip": false,
          "scale": 1.0
        },
        {
          "type": "pip",
          "loc": {
            "x": 0.0,
            "y": -1.1,
            "z": 0.0
          },
          "flip": true,
          "scale": 1.0
        }
      ]
    },
    {                   // d
      "rank": 3,
      "face": "",
      "pips": [
        {
          "type": "pip",
          "loc": {
            "x": 0.0,
            "y": 1.1,
            "z": 0.0
          },
          "flip": false,
          "scale": 1.0
        },
        {
          "type": "pip",
          "loc": {
            "x": 0.0,
            "y": 0.0,
            "z": 0.0
          },
          "flip": false,
          "scale": 1.0
        },
        {
          "type": "pip",
          "loc": {
            "x": 0.0,
            "y": -1.1,
            "z": 0.0
          },
          "flip": true,
          "scale": 1.0
        }
      ]
    },

    …
    // Cards of rank 4–12
    …

    {                   // e
      "rank": 13,
      "face": "FaceCard_13",
      "pips": [
      ]
    }
  ]                     // f
}                       // g
```

a. This is the beginning of the **`decorators`** definition, which contains JSON for the two letters and two suits at opposite corners of each card.

b. The **cards** section contains JSON for each of the 13 ranks of cards in the deck: Ace (1) –King (13).

c. The **pips** of each card are the images of the suit that are in the middle of the card (1 pip for Ace and ten pips for 10). Each pip can be positioned, scaled, and flipped vertically.

d. The rank 3 card begins here and includes three pips.

e. The rank 13 card (the King) has no pips but does have a **face**, which is a reference to the King face card image. The actual images have names like "FaceCard_13S" (for the King of Spades) and so on (with "C", "D", "H", or "S" appended to match the suit).

f. This closing bracket ends the list of cards.

g. This closing brace ends the JSON document.

As you can see, pretty JSON has a lot of extra white space (spaces, line breaks, etc.). Once people are done editing JSON, it is often converted to *minified* JSON, which eliminates the extra white space. JSON Listing B.16 shows the exact same information as JSON Listing B.15, but as you can see, the minified version is much more compact.

JSON Listing B.16 A minified version of JSON_Deck.json (Showing same info as B.15)

```
{"decorators":[{"type":"letter","loc":{"x":-1.05,"y":1.42,"z":0},"flip":false,
"scale":1.25},{"type":"suit","loc":{"x":-1.05,"y":1.03,"z":0},"flip":false,
"scale":0.4},{"type":"letter","loc":{"x":1.05,"y":-1.42,"z":0},"flip":true,
"scale":1.25},{"type":"suit","loc":{"x":1.05,"y":-1.03,"z":0},"flip":true,
"scale":0.4}],"cards":[{"rank":1,"face":"","pips":[{"type":"pip","loc":{"x":0,
"y":0,"z":0},"flip":false,"scale":2}]},{"rank":2,"face":"","pips":[{"type":"pip",
"loc":{"x":0,"y":1.1,"z":0},"flip":false,"scale":1},{"type":"pip","loc":{"x":0,
"y":-1.1,"z":0},"flip":true,"scale":1}]},{"rank":3,"face":"","pips":[{"type":"pip",
"loc":{"x":0,"y":1.1,"z":0},"flip":false,"scale":1},{"type":"pip","loc":{"x":0,
"y":0,"z":0},"flip":false,"scale":1},{"type":"pip","loc":{"x":0,"y":-1.1,"z":0},
"flip":true,"scale":1}]},…
                                                        // Cards of rank 4–12
…{"rank":13,"face":"FaceCard_13","pips":[]}]}
```

The most common way to work with JSON in Unity is via the **JsonUtility** class. This is the same utility that Unity uses to serialize the Inspector, so it's very fast and robust. See Chapter 33 for examples of using **JsonUtility**. You should also look at the Unity API documentation.[6]

6. https://docs.unity3d.com/2020.3/Documentation/ScriptReference/JsonUtility.html

tip

FIELD VALUES COPIED FROM INSPECTOR COMPONENTS ARE JSON. When you right-click on any individual field shown in the Unity Inspector (including Lists and arrays) and then choose *Copy*, the information that is copied is in JSON format, so you can paste it into a standard text editor. In fact, you can past it there, modify it, copy it again, and then right-click the field in the Inspector and choose Paste to paste the modified values back onto the Component. I have found this to be especially useful when working with `[System.Serializable]` classes in the Inspector.

(Note: This only works on a single field, though that field can be a List or array of serializable classes. However, when you click the three vertical dots (⋮) in the upper-right corner of a Component and choose *Copy Component*, it is *not* copied as JSON.)

Lambda Expressions =>

A *lambda expression* is best thought of as short-hand for a simple, anonymous function (i.e., a function for which you don't need a name). You can see an example in the "Interfaces"[7] section of this appendix as well as in Pseudocode Listing 28.13 of Chapter 28, "Data-Oriented Design." Figure B.3 and the following bulleted list show the standard syntax for a lambda expression.

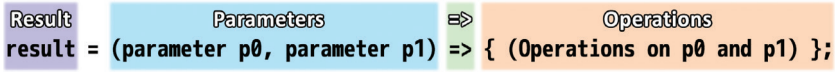


Figure B.3 Lambda Expression syntax

- **Parameters**: One or more parameters are passed in, surrounded by parentheses `()` and separated by commas. If there is only one parameter, parentheses are not necessary.
- **=>**: The "fat arrow" separates the parameters from the operations.
- **Operations**: The operations of the lambda expression are surrounded by braces `{}`. If there is only one statement, the braces are not required.
- **Result**: The result of the operations is returned to the left side of the **=** sign. If there is more than one statement in the Operations section, a `return` keyword is required to return a value (see `// b` in Code Listing B.17).

7. On line `// c` of Code Listing B.12.

You may be wondering what defines the types of the parameters and the return type of the result. This information comes from the delegate type that is targeted by the lambda expression. Code Listing B.17 shows some lambda expressions that target the **`FloatOpDelegate`** delegate type (from Code Listing B.6 of the "Delegates" section of this appendix).

Code Listing B.17 Lambda Expressions and Delegates

```
using UnityEngine;

public class LambdaExample : MonoBehaviour {
    // The FloatOpDelegate methods from the "Function Delegates" section
    public delegate float FloatOpDelegate( float f0, float f1 );                    // a

    public float FloatMultiply( float f0, float f1 ) {
        float result = f0 * f1;
        print( "The product of " + f0 + " & " + f1 + " is " + result + "." );
        return ( result );
    }

    public float FloatAdd( float f0, float f1 ) {
        float result = f0 + f1;
        print( "The sum of " + f0 + " & " + f1 + " is " + result + "." );
        return ( result );
    }


    // Lambda used as anonymous function for the delegate lambdaMultiply            // b
    private FloatOpDelegate lambdaMultiply = ( float f0, float f1 ) => {
        print( "The product of " + f0 + " & " + f1 + " is " + ( f0*f1) + "." );
        return f0 * f1;
    };

    // Simplified single-line lambda version
    private FloatOpDelegate lambdaAdd = ( float f0, float f1 ) => f0 + f1;          // c

    System.Func<float, float, float> funcAdd = ( f0, f1 ) => f0 + f1;               // d

    System.Func<float> theTime = () => Time.time;                                   // e

    private System.Action<float, float>                                             // f
        printSum = ( f0, f1 ) => print( "printSum: "+(f0 + f1) );                   // f

    public FloatOpDelegate foDelegate;

    void Start() {
        foDelegate += FloatMultiply;
```

```
>          foDelegate += lambdaMultiply;                                          // g
>
>          foDelegate( 2, 4 ); // prints "The product of 2 & 4 is 8." twice
>
>          print( "lambdaAdd: " + lambdaAdd( 2, 4 ) ); // prints "lambdaAdd: 6"       // h
>          print( "funcAdd: "   + funcAdd( 2, 4 ) );   // prints "funcAdd: 6"
>          printSum( 2, 4 );                           // prints "printSum: 6"       // i
|      }
| }
```

a. See the "Delegates" section for more info on the **FloatOpDelegate** delegate and **FloatMultiply()** and **FloatAdd()** methods listed here.

b. **lambdaMultiply** is a **FloatOpDelegate** field (just like **foDelegate** in Code Listing B.7) that has an anonymous lambda expression function assigned to it here. The anonymous lambda expression has the exact same effects as **FloatMultiply()**. Similar to delegate variable names, lambda expression names begin with a lowercase letter because they are not true functions.

c. **lambdaAdd** emulates **FloatAdd()** (except for the **print()** call). Because there is only one statement in the operations section, neither braces nor a **return** are required.

d. **System.Func<T1, T2, TResult>** is a delegate type that works for anywhere from 0 to 16 input types (**T1**, **T2**, etc.) and a single return type (**TResult**), with the return type always specified last. **FloatOpDelegate** is effectively the same as the **System.Func<float, float, float>** used here.

e. **System.Func<TResult>** is a delegate with no parameters that returns a result. Note that the parentheses between **=** and **=>** *are* required to indicate that there are no input parameters to the lambda expression.

f. **System.Action<T1, T2>** is a delegate type that works for anywhere from 1 to 16 parameters with no return type (i.e., a return type of **void**).

g. Both **FloatMultiply** and **lambdaMultiply** are added to **foDelegate**. When **foDelegate** is called, **FloatMultiply** and **lambdaMultiply** each print "The product of 2 & 4 is 8."

h. **lambdaAdd** sums 2 and 4 and returns the value 6.

i. **printSum** prints without returning a value.

Naming Conventions

I initially covered naming conventions in Chapter 20, "Variables and Components," but they're important enough to repeat here. The code in this book follows a number of rules governing the naming of variables, functions, classes, and so on. Although none of these rules are mandatory, following them makes your code more readable not only to

others who try to decipher it but also to yourself if you ever need to return to it months later and hope to understand what you wrote. Every coder follows slightly different rules—my personal rules have even changed over the years—but the rules I present here have worked well for both me and my students, and they are consistent with most C# code that I've encountered in Unity:

1. Use *camelCase* for pretty much everything. In a variable name that is composed of multiple words, camelCase capitalizes the initial letter of each word (except for the first word, in the case of variable names).
2. Variable names should start with a lowercase letter (e.g., **`someVariableName`**).
3. Function names should start with an uppercase letter (e.g., **`Start()`**, **`FunctionName()`**).
4. Class names should start with an uppercase letter (e.g., **`GameObject`**, **`ScopeExample`**).
5. Interface names often start with a capital **`I`** (e.g., **`IMovable`**).
6. Delegate and lambda expression variable names should start with a lowercase letter (e.g., **`foDelegate`**, **`lambdaMultiply`**).
7. Private variable names often start with an underscore (e.g., **`_hiddenVariable`**). (In code written by Unity private field names sometimes start with **`M_`**.)
8. Static variable names are often all caps with *snake_case* (e.g., **`NUM_INSTANCES`**). As you can see, snake_case combines multiple words with an underscore in between them.

Object-Oriented Software Design Patterns

In 1994, the "Gang of Four" (Erich Gamma, Richard Helm, Ralph Johnson, and John Vissides) released the book *Design Patterns: Elements of Reusable Object-Oriented Software,*[8] which described various patterns that could be used in software development to create effective, reusable object-oriented code. This book uses three of those patterns and refers to a fourth.

Component Pattern

Component pattern is first covered in Chapter 27, "Object-Oriented Thinking," and it is used throughout Unity. The core idea of the Component pattern is to group closely related functions and data into a single class while at the same time keeping each class as small and focused as possible.[9]

8. Erich Gamma, Richard Helm, Ralph Johnson, and John Vissides, *Design Patterns: Elements of Reusable Object-Oriented Software* (Reading, MA: Addison-Wesley, 1994). The Factory pattern is one of many described in the book. Others include the Singleton pattern, which is used in many of the tutorials in this book.
9. The full description of the Component pattern is far more complex, but this serves for our needs.

The components that are attached to GameObjects in Unity are all based on this pattern. Each GameObject in Unity is a very small class that can act as a container for several components that each do a specific—and isolated—job. For example:

- *Transform* handles position, rotation, scale, and hierarchy.
- *Rigidbody* handles motion and physics.
- *Colliders* handle actual collision and the shape of the collision volume.

Although all of these jobs are related, they are separate enough to warrant their own component. Making each a separate component also enables easy expansion in the future: separating Colliders from the Rigidbody means that you could easily add a new kind of Collider—a ConeCollider, for instance—and Rigidbody would be able to use it without any changes to the Rigidbody code.

This is certainly important for game engine developers, but what does it mean to game designers and prototypers? The most important thing that thinking in a component-oriented way gives you is smaller, shorter classes. When your scripts are shorter, they are easier to code, easier to share with other people, easier to reuse, and easier to debug—all of which are very noble goals.

The only real negative of component-oriented design is that implementing it well takes a decent amount of forethought, which somewhat flies in the face of the prototyper's philosophy of getting things working as quickly as possible. As a result of this dilemma, Part III of this book covers both a more traditional prototyping style of just writing what works in the first several chapters and a more component-oriented approach in the last chapters. The best use of components in this book is the *Dungeon Delver* project in Chapters 35 and 36, which was written to feature component-oriented design.

The components in the Entity Component System (ECS) aspect of Unity's Data-Oriented Tech Stack (DOTS) are introduced in Chapter 28, "Data-Oriented Design," and are very different from the traditional Unity Components that are attached to GameObjects. Although the ECS components are based somewhat on the same idea of separating large scripts into smaller chunks, ECS components are extremely small and contain only data, with no functionality. Though they both use the word "components," ECS and object-oriented design patterns are antithetical to each other. For more information on DOTS and ECS, please see the "Data-Oriented Design" chapter.

Observer Pattern

The Observer pattern gives us a way to listen for events to happen in the game without continually polling for them. In the Observer pattern, a *subject* is created that will send a message when a specific event happens, and one or more *observers* can be added to that subject as listeners to be notified when that event happens. In standard C#, **`events`** and **`delegates`** are used for the subjects that observers can add listeners to. In Unity,

special **UnityEvents** are a very flexible way of implementing the Observer pattern, as shown in Code Listings B.18 and B.19.

1. Create a new scene in Unity called *_Scene_ObserverPattern* that is based on the *Basic (Built-in)* scene template.
2. Create a new script named *ObserverPattern.cs* and attach it to the *Main Camera*.
3. Open the *ObserverPatternSubject* script in Visual Studio and enter the code shown in Code Listing B.18.

Code Listing B.18 ObserverPattern_Subject.cs

```
| using System.Collections;
| using System.Collections.Generic;
| using UnityEngine;
> using UnityEngine.Events;                                                    // a
|
> [System.Flags] public enum eKeyState { none=0, down=1, up=2 };               // b
|
> [System.Serializable]
> public class KeyWatchSubject {                                               // c
>     public KeyCode                          key;
>     public eKeyState                        statesToWatch;
>     public UnityEvent<KeyCode, eKeyState> listeners;                         // d
> }
|
| public class ObserverPattern : MonoBehaviour {
>     public List<KeyWatchSubject> keyWatchSubjects;                           // e
|
|     void Update() {
>         foreach (KeyWatchSubject subject in keyWatchSubjects) {
>             if ( (subject.statesToWatch & eKeyState.down) != 0) {            // f
>                 if ( Input.GetKeyDown( subject.key ) ) {
>                     subject.listeners.Invoke( subject.key, eKeyState.down ); // g
>                 }
>             }
>             if ( ( subject.statesToWatch & eKeyState.up ) != 0 ) {           // h
>                 if ( Input.GetKeyUp( subject.key ) ) {
>                     subject.listeners.Invoke( subject.key, eKeyState.up );
>                 }
>             }
>         }
|     }
| }
```

a. **UnityEvent**s require you to import the **UnityEngine.Events** library.

b. The **[System.Flags]** attribute tells Unity to treat this enum as a bitmask (like physics layers in Unity). Instead of getting a pop-up allowing you to choose only one option, you will get a pop-up showing all options—including "None" and "Everything"—and allowing you to check whichever ones you like. Here, you want to be able to create subjects that notify on KeyDown, KeyUp, or both. When you make a **[System.Flags]** enum, you must explicitly define each member of the enum as a power of 2 for Flags to work (it is also possible to specify combinations like **downAndUp=3**, which would select *both* **eKeyState.down** and **eKeyState.up**).

c. This serializable **KeyWatchSubject** class will allow you to create subjects that will respond to key presses and can notify observers when they happen.

d. A UnityEvent can have anywhere from zero to four parameters that define what the listening functions must look like. Here, we require that the listening functions must take a **KeyCode** and an **eKeyState** as arguments by placing **KeyCode** and **eKeyState** between the angle brackets.

e. A List of **KeyWatchSubject**s allows us to respond to any key press or release.

f. The result of the bitwise AND (**&**) of **subject.statesToWatch** and the **eKeyState.down** state will be non-zero if the **down** state is one of the ones we're supposed to watch for.

g. Then, if that key was pressed down this frame, the **listeners** UnityEvent is Invoked, passing the KeyCode and whether it was pressed or released to all listeners of that UnityEvent. Unlike a standard C# delegate or event, UnityEvents can be invoked without fear of a null reference exception.

h. A similar bitwise AND (**&**) is done with **statesToWatch** and the **up** state, and if **eKeyState.up** is a state to watch for, then **listeners** will be invoked.

4. *Save* the *ObserverPattern* script and return to Unity.

5. Select *Main Camera* in the Hierarchy and look at the *ObserverPattern (Script)* component in the Inspector.

6. Add two **KeyWatchSubject**s to the **keyWatchSubjects** List with the **key** and **statesToWatch** settings shown in Figure B.4a.

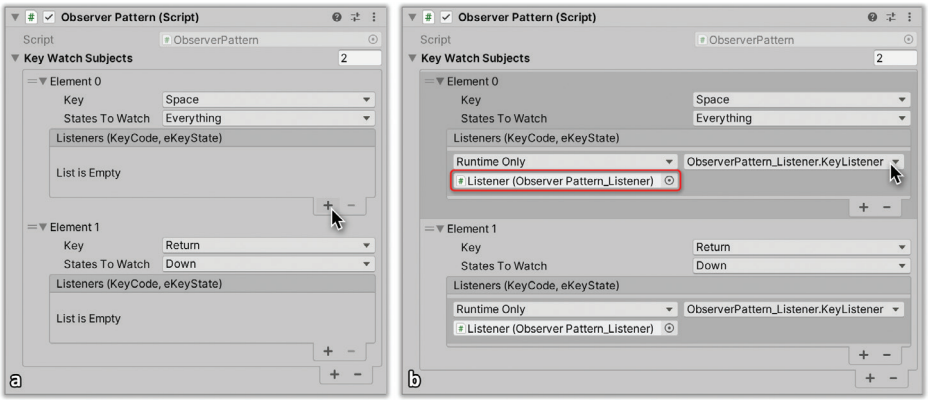


Figure B.4 UnityEvent settings for the *ObserverPattern (Script)* component of Main Camera

7. Create an empty GameObject in the Scene and name it *Listener*.
8. To make a script that will listen to the subjects, create a script named *ObserverPattern_Listener.cs* and attach it to the *Listener* GameObject.
9. Open the *ObserverPattern_Listener* script in Visual Studio and enter the code shown in Code Listing B.19.

Code Listing B.19 ObserverPattern_Listener.cs

```
| using UnityEngine;
|
| public class ObserverPattern_Listener : MonoBehaviour {
>     public void KeyListener(KeyCode key, eKeyState keyState) {                // a
>         string msg = this.name + " observed the " + key + " was ";
>         msg += ( keyState == eKeyState.down ) ? "pressed." : "released.";     // b
>         Debug.Log( msg );
>     }
| }
```

   a. To work with a UnityEvent, the listening method must be **`public`** and must have the same parameter types as the **`UnityEvent`** (in this case, a **`KeyCode`** and an **`eKeyState`**).

   b. The ternary operator is used here to return **`"pressed."`** if the **`keyState`** is **`eKeyState.down`** and **`"released."`** if it is not. We only have these two possibilities because **`KeyListener()`** will only be called by the UnityEvent on the ObserverPattern script when a key is pressed or released.

10. *Save* the *ObserverPattern_Listener* script.

11. Add listener methods to the **listeners** field of each of the **keyWatchSubjects** as shown in Figure B.4b. To do this:
    a. Click the **+** under Listeners to add an element to the **UnityEvent listeners** (under the mouse cursor in Figure B.4a).
    b. Drag the *Listener* GameObject from the Hierarchy onto the field under *Runtime Only* that is surrounded by a red rectangle in Figure B.4b. That field will now show *Listener*, though it will not yet display the text "*(ObserverPattern_Listener)*".
    c. Click the *No Function* pop-up menu (under the mouse cursor in Figure B.4b) and choose *ObserverPattern_Listener* > *KeyListener*. This tells the UnityEvent which script component and method you want to be called when the UnityEvent is invoked.
    d. Continue until your **keyWatchSubjects** settings match Figure B.4b.
12. *Save* your scene.
13. Click *Play*. If your **keyWatchSubjects** settings match mine, you should be listening for the spacebar and Return (Enter on Windows) keys to be pressed.

As you press space and Return, you will see several instances of these three messages:

```
Listener observed the Space was pressed.
Listener observed the Space was released.
Listener observed the Return was pressed.
```

This shows that the Listener GameObject was notified when the UnityEvents on the Main Camera ObserverPattern script were invoked. The new Unity InputSystem is based on this use of Observer pattern for keyboard and controller input. InputSystem requires more configuration than the Input class that is used throughout this book (making Input a better choice for prototypes), but it is very useful for large, cross-platform projects. Find out more about Unity's InputSystem at https://learn.unity.com/project/using-the-input-system-in-unity.

For an example of using standard C# **event**s to implement the Observer pattern, please see the "Using a Function Delegate Event to Fire" section of Chapter 32, "*Space SHMUP* — Part 2," where a single C# **event** named **fireEvent** is used to fire all the weapons that are attached to the player ship. I wanted to show you UnityEvents here because they are an important way that you can add observers to events in your code without changing the C# code.

Unity UI's Built-In Callbacks and Observer Pattern

The Unity UI package (that implements uGUI) contains several examples of a modified Observer pattern. If a script on a GameObject implements one of several interfaces (e.g., **IBeginDragHandler**), then the Unity EventSystem knows to call a specific method (**OnBeginDrag()** in this example) on that uGUI element whenever the player presses

the mouse button while the mouse is over that GameObject. See the "Interfaces" section of this appendix for that example.

Singleton Pattern

The Singleton pattern is the most commonly used in this book and can be found in several chapters. If you know that there will only ever be a single instance of a given class in the game, you can create a *singleton* for that class, which is a static variable of that class type that can be used to reference it from anywhere in code. Code Listing B.20 shows an example.

Code Listing B.20 Singleton Pattern

```
public class Hero : MonoBehaviour {
    static public Hero S;                                                    // a

    void Awake() {
        if (S == null) {                                                     // b
            S = this;                                                        // c
        } else {
            Debug.LogError("The singleton S of Hero has already been set!");
        }
    }
}


public class Enemy : MonoBehaviour {
    void Update() {
        Vector3 heroLoc = Hero.S.transform.position;                         // d
    }
}
```

a. The static public field **S** is the singleton for hero. I usually name all of my singletons **S**.

b. The **`if (S == null)`** statement protects you from accidentally having a second Hero instance somewhere in the code. If a second instance of Hero exists and tries to assign itself to **S**, then the error message will be thrown.

c. Because there will only ever be one instance of the Hero class, it is assigned to **S** on **`Awake()`**, when the instance is created.

d. Because **S** is both public and static, it can be referenced anywhere in code via the class name as **`Hero.S`**.

If you search online, you're likely to find a lot of hate out there for the Singleton pattern. This is largely because of two things:

1. **Singletons are unsafe in a production environment**: Singletons are static and public, meaning that ANY class or function in your entire project could potentially access them. The danger here is that some random class written by someone else could change a public field of your singleton class instance, and you would have no idea who did it!
2. **The Singleton pattern is very simple to implement and therefore often overused**: As the simplest design pattern to implement, Singleton was quickly used by a lot of people, and soon caused problems due to the preceding point. This meant that a lot of people used it in places where it wasn't really appropriate, which is part of its bad reputation.

Luckily, you can avoid the first danger in several ways. One that I like is making the singleton not only `static` but also `private`, so that only instances of this single class can access it (and because it's a singleton, there will only ever be one instance of this class). You then write `static public` accessor properties through which other classes and functions can alter fields of the singleton instance. If you find that something unknown in your code is changing a property, then you can place a debugger breakpoint in the `set` clause of the property and use the call stack in the debugger to see what method made the call to set the property.

When you write prototypes, development speed is often more important than safety, so my recommendation is that you should feel free to use singletons when you need them while prototyping but avoid using them in production code (or use modified singletons that are less problematic).

Strategy Pattern

As mentioned in the "Function Delegates" section of this appendix, the Strategy pattern is often used in AI and other areas where you might want to change behavior based on conditions yet still only call a single function delegate. In the Strategy pattern, a function delegate is created for a type of action that the class can perform (e.g., taking an action in combat), and an instance of that delegate is given different functions to call based on the situation. This avoids complicated switch statements in the code, because the delegate can be called in a single line (see Code Listing B.21).

Code Listing B.21 Strategy Pattern

```
| using UnityEngine;
|
| public class Strategy : MonoBehaviour {
>     public delegate void ctionDelegate();                                    // a
|
```

```
>     public ActionDelegate strategy;                                    // b
|     // public System.Action strategy;
|
>     public void Attack() {/* Attack code would go here */ }            // c
>     public void Wait() { /* Wait code here */ }
>     public void Flee() { /* Flee code here */ }
|
>     void Awake() {
>         strategy = Wait;                                               // d
>     }
|
|     void Update() {
>         // Possibly change strategy
>         Vector3 hPos = Hero.S.transform.position;
>         if ( (hPos - transform.position).magnitude < 100 ) {           // e
>             strategy = Attack;
>         }
>
>         // Execute the method for the current strategy
>         if (strategy != null) strategy();                              // f
|     }
| }
```

a. The **ActionDelegate** delegate type is defined. It has no parameters and a return type of void.

b. **strategy** is created as an instance of **ActionDelegate**. (As shown in the comment on the subsequent line, the built-in **System.Action** delegate could also have been used here in place of defining our own ActionDelegate.)

c. The **Attack()**, **Wait()**, and **Flee()** functions here are placeholders that are meant to show that various actions would be defined matching the parameters and return type of the **ActionDelegate** delegate type.

d. The initial strategy for this agent is to **Wait**, so **Wait** is assigned as the target method of **strategy**.

e. If the Hero singleton comes within 100 meters of this agent, it will switch its strategy to **Attack** by replacing the target method of **strategy**.

f. Regardless of which strategy is selected, **strategy()** is called to execute it. Checking that **strategy != null** before calling it is useful because calling a null function delegate (i.e., one that has no target function assigned to it) will cause a runtime error.

More Information on Design Patterns in Game Programming

Game Programming Patterns[10] by Robert Nystrom is a fantastic book that covers the most common software design patterns used in games. You can purchase a print or electronic copy of it from most online retailers, or you can just read the web version for

free on his website: http://GameProgrammingPatterns.com . It's a great resource for improving your OOP coding.

Operator Precedence and Order of Operations

Just as in algebra, some operators in C# take precedence over others. One example that you are probably familiar with is the precedence of * over + (e.g., 1 + 2 * 3 = 7 because the 2 and 3 are multiplied *before* the 1 is added to them). Here is a list of common operators and their precedence. An operator that is higher in this list will happen before one that is lower.

Table B.1 Operator Precedence

Precedence	Operator	Description
1.	`( )`	Operations grouped by parentheses always take precedence
2.	`F()`	The calling of a function
3.	`a[]`	The access of an array
4.	`i++`	Post-increment
5.	`i--`	Post-decrement
6.	`!`	NOT
7.	`~`	Bitwise NOT (complement)
8.	`++i`	Pre-increment
9.	`--i`	Pre-decrement
10.	`*`	Multiply
11.	`/`	Divide
12.	`%`	Modulus
13.	`+`	Add
14.	`-`	Subtract
15.	`<<`	Bit shift left
16.	`>>`	Bit shift right
17.	`<`	Less than
18.	`>`	Greater than
19.	`<=`	Less than or equal to
21.	`>=`	Greater than or equal to

10. Robert Nystrom, *Game Programming Patterns* (Genever Benning, 2014). ©2014 by Robert Nystrom.

Precedence	Operator	Description
22.	**==**	Equal to (the comparison operator)
23.	**!=**	Not equal to
24.	**&**	Bitwise AND
25.	**^**	Bitwise exclusive OR (XOR)
26.	**|**	Bitwise OR
27.	**&&**	Conditional, shorting AND
28.	**||**	Conditional, shorting OR
29.	**=**	Assignment

Race Conditions

Unlike many of the other topics in this section, a race condition is something that you definitely *do not* want in your code. A *race condition* occurs when your code relies on one thing happening before another, but it's possible that the two things could happen out of order and cause unexpected behavior or even a crash. In traditional computer science, race conditions are a serious consideration when designing any code that is intended for multiprocessor computers, multithreaded operating systems,[11] or networked applications (where different computers around the world could possibly end up in a race condition with each other), but it is also an issue for Unity games because they involve so many different GameObjects, each receiving **`Awake()`**, **`Start()`**, and **`Update()`** calls at roughly the same time as all the others.

Let's create an example.

Unity Example—Race Conditions

Follow these steps:

1. Create a new Unity project named *Race Condition* based on the *3D Core* template.
2. Create a C# script named *SetValues* and enter the code in Code Listing B.22.

Code Listing B.22 SetValues.cs

```
| using System.Collections;
| using UnityEngine;
|
| public class SetValues : MonoBehaviour {
>     static public int[]       VALUES;
```

11. Happily, the C# Job System that is part of Unity's Data-Oriented Tech Stack has safeguards to avoid race conditions when writing multithreaded code, but that's not what we're talking about here.

```
|
|     void Start() {
>         VALUES = new int[] { 0, 1, 2, 3, 4, 5 };
|     }
| }
```

3. Create a second script named *ReadValues* and enter the code in Code Listing B.23.

Code Listing B.23 ReadValues.cs

```
| using System.Collections;
| using UnityEngine;
|
| public class ReadValues : MonoBehaviour {
|     void Start() {
>         print( SetValues.VALUES[2] );
|     }
| }
```

4. Be sure that you have saved both scripts before returning to Unity.
5. Attach both scripts to *Main Camera* and click *Play*. When you do so, you'll receive one of two possible outputs in the Console:
   - 2
   - **NullReferenceException**: Object reference not set to an instance of an object

The thing that determines what outcome you see is which of the two **`Start()`** functions happens to be called first. If **`SetValues.Start()`** is called before **`ReadValues.Start()`**, everything works great. However, if **`ReadValues.Start()`** is called before **`SetValues.Start()`**, you get a null reference exception because **`ReadValues.Start()`** is trying to access **`SetValues.VALUES[2]`** while **`SetValues.VALUES`** is still null.

In early versions of Unity, it was extremely difficult to know which of these **`Start()`** methods would be called first. Happily, the Script Execution Order window now allows you to choose the order in which scripts execute.

6. From the Unity menu bar, choose *Edit* > *Project Settings...*, which opens the Project Settings window.
7. Click *Script Execution Order* on the left side of the Project Settings window, as shown in Figure B.5a to open the Script Execution Order Inspector (SEO Inspector). Note that there are several scripts already in the SEO Inspector because of TextMesh Pro and other Unity extensions.

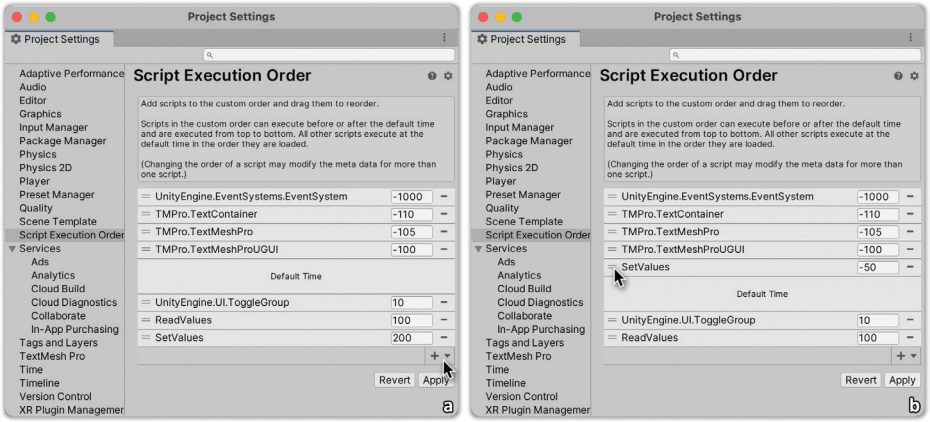


Figure B.5 The Script Execution Order Inspector

8. Add the *ReadValues* class to the SEO Inspector by clicking the **+** button under the arrow cursor in Figure B.5a.
9. Do the same to also add *SetValues* class to the SEO Inspector.

By default, ReadValues and SetValues will get the execution order values 100 and 200 as shown in Figure B.5a.

10. Click the *Apply* button in the SEO Inspector.
11. Click *Play* in Unity. This execution order will guarantee that a NullReferenceException appears in the Console.
12. Stop Unity playback.
13. Use the double-line handle on the *SetValues* bar in the SEO Inspector (under the cursor in Figure B.5b) to drag *SetValues* up above *Default Time*. Now your Inspector should look like Figure B.5b.
14. Click *Apply* in the SEO Inspector and close the SEO Inspector window.
15. Click *Play* in Unity.

Now that **`SetValues.Start()`** is guaranteed to be called before **`ReadValues.Start()`**, the "2" result is guaranteed to appear in the Console.

The double-size *Default Time* row shows when all non-specified scripts will execute.

When dealing with two scripts that both use **`Start()`**, **`Awake()`**, or any other MonoBehaviour call that is managed by Unity, the Script Execution Order Inspector is the only way to guarantee that one will run before the other.

Recursive Functions

A function designed to call itself is known as a *recursive function*. Recursive functions are introduced in the "Recursive Functions" section of Chapter 24, "Functions and Parameters," so please read the information in that chapter.

As demonstrated in Chapter 24, one simple example of a recursive function could be a function to calculate the factorial of a number (in math, 5! [5 factorial] is the multiplication of five and every other natural number below it: 5! = 5 * 4 * 3 * 2 * 1 = 120). Another fantastic example of a recursive function is the recursive Bézier curve interpolation method that is explored in the "Interpolation" section of this appendix. Please see both of these examples to learn about recursive functions.

String Interpolation – $""

While not featured extensively in this book, string interpolation is one of the easiest ways to format data into a string. You can use it to replace many instances of string conversion (e.g., **`(12345.6789f).ToString("#,##0.00")`** to output the string **`"12,345.68"`**). String interpolation using **`$""`** is also able to combine multiple variables into one string and provide formatting and alignment for them. Code Listing B.24 demonstrates some uses of string interpolation. Several comments in the code listing show the console output of the previous line.

Code Listing B.24 StringInterpolationExample.cs

```
using UnityEngine;

public class StringInterpolationExample : MonoBehaviour {
    System.Text.StringBuilder sb;                                          // a

    void Start() {
        float inchToMM = 25.4f;
        float inchToFt = 12;
        float inchToYd = 36;
        float ftToMile = 5280;
        sb = new System.Text.StringBuilder();

        //-------- Simple expressions in String Interpolation               // b
        sb.AppendLine( $"1 foot (ft) is {inchToFt} inches (in)" );
        //1 foot (ft) is 12 inches (in)
        sb.AppendLine( $"1 mile is {ftToMile}ft or {ftToMile * inchToFt}in" );
        //1 mile is 5280ft or 63360in
        sb.Append( $"1 mile is {ftToMile:#,##0}ft" );                      // c
        sb.AppendLine( $" or {ftToMile * inchToFt:#,##0}in" );
        //1 mile is 5,280ft or 63,360in

        sb.Append( "\n\n" ); // Two line breaks

```

```
>         //———————— Numeric formatting                                    // d
>         double n = 12345.6789;
>         sb.AppendLine( $"No formatting: {n}" );
>         //No formatting: 12345.6789
>         sb.AppendLine( $"000000.000000: {n:000000.000000}" );  // Zeros
>         //000000.000000: 012345.678900
>         sb.AppendLine( $"######.######: {n:######.######}" );  // Number symbols
>         //######.######: 12345.6789
>         sb.AppendLine( $"#,###.##: {n:#,###.##}" );
>         //#,###.##: 12,345.68
>
>         sb.Append( "\n\n" ); // Two line breaks
>
>         //———————— Left and Right Alignment                              // e
>         sb.AppendLine( $"Left Aligned:  |{"<==20 chars==>", -20}|" );
>         // "Left Aligned:  |<==20 chars==>      |"  // 20 chars between pipes
>         sb.AppendLine( $"Right Aligned: |{1234.5678f, 20 : 0.##}|" );
>         // "Right Aligned: |             1234.57|"  // The float is rounded
>
>         sb.Append( "\n\n" ); // Two line breaks
>
>         sb.AppendLine("  ----8--- = -------16-------m = -------16-------mm"); // f
>         PrintUnitConversion( "Inch", inchToMM );
>         PrintUnitConversion( "Foot", inchToMM * inchToFt );
>         PrintUnitConversion( "Yard", inchToMM * inchToYd );
>         PrintUnitConversion( "Mile", inchToMM * inchToFt * ftToMile );
>
>         Debug.Log( sb.ToString() );
|     }
|
>     //———————— Combining it all together into a table of data             // g
>     void PrintUnitConversion( string uName, float asMM ) {
>         sb.Append( $"1 {uName, -8}" );
>         sb.Append( $" = {(asMM / 1000f), 16 : #,##0.000}m" );
>         sb.AppendLine( $" = {asMM, 16 : #,0.#}mm" );
>     }
|     //  ----8--- = -------16-------m = -------16-------mm
|     //1 Inch     =           0.025m =            25.4mm
|     //1 Foot     =           0.305m =           304.8mm
|     //1 Yard     =           0.914m =           914.4mm
|     //1 Mile     =       1,609.344m =       1,609,344mm
| }
```

a. For information about **`StringBuilder`**, see the next heading in this appendix.

b. As shown in these lines, any variable name or simple expression can be placed between braces to be interpolated into the string.

c. A string interpolation call cannot span two lines of code, which is why this is split into two **`$""`** interpolations. These two lines are also the first instance of using **`#`** (number symbol) and **`0`** (zero) after a colon (**`:`**) for numeric formatting.

d. The `double n` is shown here with no formatting, `0`s that force a character even if it is not needed, `#`s that do not force extra characters, and a comma in the thousands place that causes the comma separation of every three digits above the decimal. Note that the number of `#`s after the decimal limits the number of places printed, while `#`s before the decimal do not truncate the number (e.g., `$"{n:#.#}"` prints `12345.7`).

e. Placing a comma and a number after the simple expression causes the string for that expression to take up a specific number of spaces in output. If the number is negative, the text will be left-aligned, and if it is positive, it will be right-aligned.

f. This line prints the header seen after `// g`.

g. This function adds lines for several imperial to metric unit conversions. Alignment, numeric formatting, and simple expressions are all used here to generate the nicely formatted text data in the comment block that follows. Note the shorthand number format on the last line of the method.

To find out more about string interpolation, search online for "c# string interpolation".

StringBuilder

C# `strings` have immutable length (much like arrays), so any time you concatenate two strings together, a completely new string is created, and the characters from each of the two concatenated strings are copied there. This creates a lot of extra copying and memory allocation that must then be cleaned up by C# garbage collection, all of which is inefficient. The `System.Text.StringBuilder` class avoids many of these issues, so it is preferable in instances when you plan to concatenate several strings together. Though I don't use StringBuilders much in this book, I use them very frequently in my coding, especially when creating long logs while debugging. Code Listing B.25 shows an example of how to use a StringBuilder to generate tabbed output that can be pulled into a spreadsheet.

Code Listing B.25 StringBuilderExample.cs

```
using UnityEngine;

public class StringBuilderExample : MonoBehaviour {
    [Header( "Inscribed" )]
    public int gridSize = 32;                                    // a
    public float perlinZoom = 0.15f;                             // b

    [Header( "Dynamic" )]
    public bool clickToGenerate = false;

    private System.Text.StringBuilder sb;                        // c
```

```
|
|     void Start() {
>         clickToGenerate = false;
>         sb = new System.Text.StringBuilder();                                  // d
|     }
|
|     void Update() {
>         if ( !clickToGenerate ) return;
>
>         sb.Clear();                                                            // e
>         sb.Append( "Perlin Noise output for perlinZoom=" );                    // f
>         sb.AppendLine( perlinZoom.ToString("0.00") );  // The 0s are zeros    // g
>         float u;
>         int num;
>         for ( int x = 0; x < gridSize; x++ ) {
>             for ( int y = 0; y < gridSize; y++ ) {
>                 u = Mathf.PerlinNoise( x * perlinZoom, y * perlinZoom );       // h
>                 sb.Append($"{u:0.00}");  // String interpolation               // g
>                 sb.Append( '\t' );                                             // i
>             }
>             sb.AppendLine();                                                   // j
>         }
>
>         Debug.Log( sb.ToString() );                                            // k
>
>         clickToGenerate = false;
|     }
| }
```

a. **gridSize** determines the size of the output. The default value of 32 will generate a grid of output that is 32 columns by 32 rows.

b. Unity's **Mathf.PerlinNoise()** method always returns the same value for all integer numbers, so **perlinZoom** is multiplied by the **x** and **y** iterators on line **// h** to avoid lining up with integer numbers. A **perlinZoom** value of 1 would return 0.47 for *every* cell.

c. You could add **using System.Text;** to the top of this script, but because StringBuilder is the *only* thing that I use from the System.Text library, I prefer to *fully qualify* StringBuilder (i.e., to write **System.StringBuilder**) here and on line **// d** rather than import the entirety of **System.Text**.

d. Because StringBuilders can easily be cleared and reused, I often use only one instance.

e. **sb.Clear()** empties the StringBuilder, preparing it for reuse.

f. **sb.Append()** appends the string to the StringBuilder. Each string appended to the StringBuilder is inserted into to a preallocated **char[]** array, so no copying

or reallocation of memory needs to occur. If the characters added exceed the size of the array, the StringBuilder then allocates another **`char[]`** array of the same size and starts filling it with the additional characters.

g. A call to **`sb.AppendLine()`** is identical to **`sb.Append()`** except that it also adds a newline sequence (**`"\n"`** on macOS and Linux or **`"\r\n"`** on Windows) to the end.

The **`"0.00"`** in **`perlinZoom.ToString("0.00")`** tells C# to convert the float value of **`perlinZoom`** to a string with one whole number, a decimal point, and then two places after the decimal. This avoids extremely long numbers in the string output, which I find helps me read the output much more easily. You can also see the *string interpolation* version of this same string formatting at the **`// g`**, a few lines later in the code.

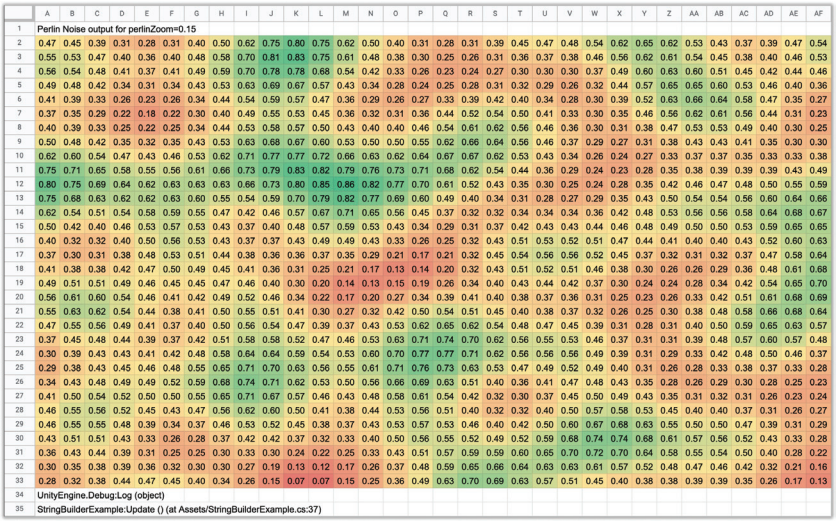
h. The **`Mathf.PerlinNoise()`** method returns a float value (usually) between 0 and 1 at the given 2D location.

i. **`sb.Append()`** can also take a single char as input.

j. **`sb.AppendLine()`** can be called with no parameter to just insert a newline.

k. When **`sb.ToString()`** is called, the **`char[]`** arrays are finally concatenated together into a string.

To test this code, attach it to a GameObject in Unity, click Play, and then click the *clickToGenerate* check box. One of the major benefits of separating log values with tabs is that you can then easily copy and paste them into Google Sheets, as you can see in Figure B.6.



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	AA	AB	AC	AD	AE	AF
1	Perlin Noise output for perlinZoom=0.15																															
2	0.47	0.45	0.39	0.31	0.28	0.31	0.40	0.50	0.62	0.75	0.80	0.75	0.62	0.50	0.40	0.31	0.28	0.31	0.39	0.45	0.47	0.48	0.54	0.62	0.65	0.62	0.53	0.43	0.37	0.39	0.47	0.54
3	0.55	0.53	0.47	0.40	0.36	0.40	0.48	0.58	0.70	0.81	0.83	0.75	0.61	0.48	0.38	0.30	0.25	0.26	0.31	0.36	0.37	0.38	0.46	0.56	0.62	0.61	0.54	0.45	0.38	0.40	0.46	0.53
4	0.56	0.54	0.48	0.41	0.37	0.41	0.49	0.59	0.70	0.78	0.78	0.68	0.54	0.42	0.33	0.26	0.23	0.24	0.27	0.30	0.30	0.30	0.37	0.49	0.60	0.63	0.60	0.51	0.45	0.42	0.44	0.46
5	0.49	0.48	0.42	0.34	0.31	0.34	0.43	0.53	0.63	0.69	0.67	0.57	0.43	0.34	0.28	0.24	0.25	0.28	0.31	0.32	0.29	0.26	0.32	0.44	0.57	0.65	0.65	0.60	0.53	0.46	0.40	0.36
6	0.41	0.39	0.33	0.26	0.23	0.26	0.34	0.44	0.54	0.59	0.57	0.47	0.36	0.29	0.26	0.27	0.33	0.39	0.42	0.40	0.34	0.28	0.30	0.39	0.52	0.63	0.66	0.64	0.58	0.47	0.35	0.27
7	0.37	0.35	0.29	0.22	0.18	0.22	0.30	0.40	0.49	0.55	0.53	0.45	0.36	0.32	0.31	0.36	0.44	0.52	0.54	0.50	0.41	0.33	0.30	0.35	0.46	0.56	0.62	0.61	0.56	0.44	0.31	0.23
8	0.40	0.39	0.33	0.25	0.22	0.25	0.34	0.44	0.53	0.58	0.57	0.50	0.43	0.40	0.40	0.46	0.54	0.61	0.62	0.56	0.46	0.36	0.30	0.31	0.38	0.47	0.53	0.53	0.49	0.40	0.30	0.25
9	0.50	0.48	0.42	0.35	0.32	0.35	0.43	0.53	0.63	0.68	0.67	0.60	0.53	0.50	0.50	0.55	0.62	0.66	0.64	0.56	0.46	0.37	0.29	0.27	0.31	0.38	0.43	0.43	0.41	0.35	0.30	0.30
10	0.62	0.60	0.54	0.47	0.43	0.46	0.53	0.62	0.71	0.77	0.77	0.72	0.66	0.63	0.62	0.64	0.67	0.67	0.62	0.53	0.43	0.34	0.26	0.24	0.27	0.33	0.37	0.37	0.35	0.33	0.33	0.38
11	0.75	0.71	0.65	0.58	0.55	0.56	0.61	0.66	0.73	0.79	0.83	0.82	0.79	0.76	0.73	0.71	0.68	0.62	0.54	0.44	0.36	0.29	0.24	0.23	0.28	0.35	0.38	0.39	0.39	0.39	0.43	0.49
12	0.80	0.75	0.69	0.64	0.62	0.63	0.63	0.63	0.66	0.73	0.80	0.85	0.86	0.82	0.77	0.70	0.61	0.52	0.43	0.35	0.30	0.25	0.24	0.28	0.35	0.42	0.46	0.47	0.48	0.50	0.55	0.59
13	0.75	0.68	0.63	0.62	0.62	0.63	0.60	0.55	0.54	0.59	0.70	0.79	0.82	0.77	0.69	0.60	0.49	0.40	0.34	0.31	0.28	0.27	0.29	0.35	0.43	0.50	0.54	0.54	0.56	0.60	0.64	0.66
14	0.62	0.54	0.51	0.54	0.58	0.59	0.55	0.47	0.42	0.46	0.57	0.67	0.71	0.65	0.56	0.45	0.37	0.32	0.32	0.34	0.34	0.34	0.36	0.42	0.48	0.53	0.56	0.56	0.58	0.64	0.68	0.67
15	0.50	0.42	0.40	0.46	0.53	0.57	0.53	0.43	0.37	0.40	0.48	0.57	0.59	0.53	0.43	0.34	0.29	0.31	0.37	0.42	0.43	0.43	0.44	0.46	0.48	0.49	0.50	0.50	0.53	0.59	0.65	0.65
16	0.40	0.32	0.32	0.40	0.50	0.56	0.53	0.43	0.37	0.37	0.43	0.49	0.49	0.43	0.33	0.26	0.25	0.32	0.43	0.51	0.53	0.52	0.51	0.47	0.44	0.41	0.40	0.40	0.43	0.52	0.60	0.63
17	0.37	0.30	0.31	0.38	0.48	0.53	0.51	0.44	0.38	0.36	0.36	0.37	0.35	0.29	0.21	0.17	0.21	0.32	0.45	0.54	0.56	0.56	0.52	0.45	0.37	0.32	0.31	0.32	0.37	0.47	0.58	0.64
18	0.41	0.38	0.38	0.42	0.47	0.50	0.49	0.45	0.41	0.36	0.31	0.25	0.21	0.17	0.13	0.14	0.20	0.32	0.43	0.51	0.52	0.51	0.46	0.38	0.30	0.26	0.26	0.29	0.36	0.48	0.61	0.68
19	0.49	0.51	0.51	0.49	0.46	0.45	0.45	0.47	0.46	0.40	0.30	0.20	0.14	0.13	0.15	0.19	0.26	0.34	0.40	0.43	0.44	0.42	0.37	0.30	0.24	0.24	0.28	0.34	0.42	0.54	0.65	0.70
20	0.56	0.61	0.60	0.54	0.46	0.41	0.42	0.49	0.52	0.46	0.34	0.22	0.17	0.20	0.27	0.34	0.39	0.41	0.40	0.38	0.37	0.36	0.31	0.25	0.23	0.26	0.33	0.42	0.51	0.61	0.68	0.69
21	0.55	0.63	0.62	0.54	0.44	0.38	0.41	0.50	0.55	0.51	0.41	0.30	0.27	0.32	0.42	0.50	0.54	0.51	0.45	0.40	0.38	0.37	0.32	0.26	0.25	0.30	0.38	0.48	0.58	0.66	0.68	0.64
22	0.47	0.55	0.56	0.49	0.41	0.37	0.40	0.50	0.56	0.54	0.47	0.39	0.37	0.43	0.53	0.62	0.65	0.62	0.54	0.48	0.47	0.45	0.39	0.31	0.28	0.31	0.40	0.50	0.59	0.65	0.63	0.57
23	0.37	0.45	0.48	0.44	0.39	0.37	0.42	0.51	0.58	0.58	0.52	0.47	0.46	0.53	0.63	0.71	0.74	0.70	0.62	0.56	0.55	0.53	0.46	0.37	0.31	0.31	0.39	0.48	0.57	0.60	0.57	0.48
24	0.30	0.39	0.43	0.43	0.41	0.42	0.48	0.58	0.64	0.64	0.59	0.54	0.53	0.60	0.70	0.77	0.77	0.71	0.62	0.56	0.56	0.56	0.49	0.39	0.31	0.29	0.33	0.42	0.48	0.50	0.46	0.37
25	0.29	0.38	0.43	0.45	0.46	0.48	0.55	0.65	0.71	0.70	0.63	0.56	0.55	0.61	0.71	0.76	0.73	0.63	0.53	0.47	0.49	0.52	0.49	0.40	0.31	0.26	0.28	0.33	0.38	0.37	0.33	0.28
26	0.34	0.43	0.48	0.49	0.49	0.52	0.59	0.68	0.74	0.71	0.62	0.53	0.50	0.56	0.66	0.69	0.63	0.51	0.40	0.36	0.41	0.47	0.48	0.43	0.35	0.28	0.26	0.29	0.30	0.28	0.25	0.23
27	0.41	0.50	0.54	0.52	0.50	0.50	0.55	0.65	0.71	0.67	0.57	0.46	0.43	0.48	0.58	0.61	0.54	0.42	0.32	0.30	0.37	0.45	0.50	0.49	0.43	0.35	0.31	0.32	0.31	0.26	0.23	0.24
28	0.46	0.55	0.56	0.52	0.45	0.43	0.47	0.56	0.62	0.60	0.50	0.41	0.38	0.44	0.53	0.56	0.51	0.40	0.32	0.32	0.40	0.50	0.57	0.58	0.53	0.45	0.40	0.40	0.37	0.31	0.26	0.27
29	0.46	0.55	0.55	0.48	0.39	0.34	0.37	0.46	0.53	0.52	0.45	0.38	0.37	0.43	0.53	0.57	0.53	0.46	0.40	0.42	0.50	0.60	0.67	0.68	0.63	0.55	0.50	0.50	0.47	0.39	0.31	0.29
30	0.43	0.51	0.51	0.43	0.33	0.26	0.28	0.37	0.42	0.42	0.37	0.32	0.33	0.40	0.50	0.56	0.55	0.52	0.49	0.52	0.59	0.68	0.74	0.74	0.68	0.61	0.57	0.56	0.52	0.43	0.33	0.28
31	0.36	0.43	0.44	0.39	0.31	0.25	0.25	0.30	0.33	0.30	0.24	0.22	0.25	0.33	0.43	0.51	0.57	0.59	0.59	0.60	0.65	0.70	0.72	0.70	0.64	0.58	0.55	0.54	0.50	0.40	0.28	0.22
32	0.30	0.35	0.38	0.39	0.36	0.32	0.30	0.30	0.27	0.19	0.13	0.12	0.17	0.26	0.37	0.48	0.59	0.65	0.66	0.64	0.63	0.63	0.61	0.57	0.52	0.48	0.47	0.46	0.42	0.32	0.21	0.16
33	0.28	0.32	0.38	0.44	0.47	0.45	0.40	0.34	0.26	0.15	0.07	0.07	0.15	0.25	0.36	0.49	0.63	0.70	0.69	0.63	0.57	0.51	0.45	0.40	0.38	0.38	0.39	0.39	0.35	0.26	0.17	0.13
34	UnityEngine.Debug:Log (object)																															
35	StringBuilderExample:Update () (at Assets/StringBuilderExample.cs:37)																															

Figure B.6 Output from the StringBuilderExample code with default values copied into Google Sheets with ColorScale Conditional Formatting applied in Sheets

Check out Chapter 28, "Data-Oriented Design," for a much more interesting use of Perlin noise.

Structs

A struct is similar to a class, but it includes no inheritance or polymorphism,[12] and as a result, all the data in a struct can be stored together in memory, in the same place, as long as each field in the struct is *blittable*. There is much more to learn about what makes data blittable, but the easiest way I have found to think about it is that data is blittable if it can be stored as one chunk in one place in memory.[13] So, it makes sense that a struct composed of only blittable data is also blittable itself. Classes are never blittable.

Because data locality (having the data you need all stored close together in computer memory) is central to *Data-Oriented Design* (DOD), structs are used extensively in DOD and especially in making the components of Entity Component Systems (ECS). Please read more about all of these concepts in Chapter 28, "Data-Oriented Design." You can also find out more about structs and how they are used in C# at the Microsoft C# Documentation[14] page, and several examples can be found by searching for "C# struct" online.

Unity Messages Beyond `Start()` and `Update()`

Unity has several messages that it sends to MonoBehaviour subclasses. The ones you're most familiar with are **`Awake()`**, **`Start()`**, **`Update()`**, **`FixedUpdate()`**, and the various collision messages, but there are many more. Here are some of the most useful ones.

12. In its most basic sense, polymorphism describes the ability of a C# subclass to either run the base class version of a method or to override that base method with its own version. For example, in Code Listing B.12, the **`virtual public void Move()`** method on **`Droid`** is overridden by the **`override public void Move()`** method on **`TurboDroid`**, and then the overriding **`Move()`** method on **`TurboDroid`** uses **`base.Move()`** to call the base class version of **`Move()`** that is defined in **`Droid`**.
13. A more accurate definition that I have found of *blittable* is: A variable type is blittable if it is stored the same way in both managed and unmanaged memory. Managed heap memory is automatically allocated by C# and then automatically reclaimed by garbage collection; managed memory is easier to work with but it becomes fragmented and is slower than unmanaged stack memory.
14. https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/struct — accessed August 25, 2021.

note

LEGEND OF SYMBOLS AFTER THE NAMES OF VARIOUS MESSAGES
Legend symbols include the following:

- ° This message is *not* called if the script is disabled (i.e., if the check box next to the script component name in the Unity Inspector is unchecked). The check box to enable/disable a script *only* appears if one of these ° messages is implemented in the script.
- ∞ This message is called both in Play mode and also every *editor frame*. Editor frames occur when the Scene or Game view is updated in the Editor. If nothing in either view is changing, it can be several seconds between editor frames.

Life-Cycle Messages

Life-cycle messages are called at specific events as the GameObject or individual script is created, made active or inactive (GameObject), enabled or disabled (script), destroyed, or reset.

- **`Awake()`**: Called when the script is instantiated
- **`Start()`°**: Called immediately before the first **`Update()`** is called on a script; as such, it always happens *after* **`Awake()`** and *before* the first **`Update()`**
- **`OnEnable()`°**: Called only during play mode when this script is enabled or the GameObject it is attached to is made active
- **`OnDisable()`°**: Called only during play mode when this script is disabled or the GameObject it is attached to is made inactive
- **`OnDestroy()`**: Called when this script or GameObject will be destroyed
- **`Reset()`**: Called in the Unity Editor when Reset is chosen from the three vertical dot (⋮) pop-up menu in the top-right of this script Component in the Inspector

Frame-Based Messages

Frame-based messages are called every time a frame is drawn to the screen.

- **`Update()`°**: Called every frame
- **`LateUpdate()`°**: Called every frame after **`Update()`** has been called on all GameObjects
- **`OnDrawGizmos()`∞**: Called every frame and editor frame (i.e., when the Scene or Game view is updated in the Editor) to draw gizmos to screen for debugging (gizmos are only drawn in the Unity Editor, not in builds)

- **`OnDrawGizmosSelected()`∞**: Called every frame and editor frame to draw gizmos if this GameObject is currently selected in the Editor
- **`OnGUI()`°**: This is called to draw Unity's IMGUI (Immediate Mode Graphical User Interface, where the GUI is redrawn from scratch at least once every frame), but it is very rarely used now. However, it could still be good for testing and quick prototypes as well as some Editor scripts (e.g., older-style **`PropertyDrawer`** scripts).

Physics-Based Messages

All physics system messages are called every physics frame (i.e., every time the physics system updates) immediately after **`FixedUpdate()`** is called. Note that *all* of these except for **`FixedUpdate()`** are still called even if the script is disabled.

- **`FixedUpdate()`°**: Called every physics frame (i.e., every time the physics system updates) if the script is enabled
- **OnCollision Messages – Collision of one Collider with another Collider**: If two Colliders touch, and both have **`isTrigger`** = **`false`**, these messages will be called. Note that the parameter passed with all of these messages is a **`Collision`** that includes information about contact points, relative velocity, etc.
  - **`OnCollisionEnter(Collision collision)`**: Called on the physics frame when the Collider of this GameObject begins touching a Collider of another GameObject
  - **`OnCollisionStay(Collision collision)`**: Called on every physics frame that the Collider of this GameObject continues touching a Collider of another GameObject (this is first called on the physics frame after **`OnCollisionEnter()`** was called, if the Colliders are still touching)
  - **`OnCollisionExit(Collision collision)`**: Called on the physics frame when the Collider of this GameObject has stopped touching a Collider of another GameObject
- **OnTrigger Messages – Collision of one Trigger with another Trigger or Collider**: If two Colliders touch, and either or both have **`isTrigger`** = **`true`**, these messages will be called. Note that the parameter passed with all of these is the *other* **`Collider`**, so there is *no* information about contact points, relative velocity, etc.
  - **`OnTriggerEnter(Collider collider)`**: Called on the physics frame when the Collider of this GameObject begins touching a Collider of another GameObject
  - **`OnTriggerStay(Collider collider)`**: Called on every physics frame that the Collider of this GameObject continues touching a Collider of another GameObject (this is initially called on the next physics frame after **`OnTriggerEnter()`** was called, if the Colliders are still touching)

  - **`OnTriggerExit(Collider collider)`**: Called on the physics frame when the Collider of this GameObject has stopped touching a Collider of another GameObject

- **Mouse Interaction**: These messages are called when the mouse interacts with a Collider of a GameObject (or an Immediate Mode GUI object). In all of these cases, the mouse overlap is checked against the Colliders in 2D screen space.
  - **`OnMouseEnter()`**: Called on the physics frame when the mouse enters a Collider of this GameObject
  - **`OnMouseOver()`**: Called on every physics frame that the mouse is over a Collider of this GameObject
  - **`OnMouseExit()`**: Called on the physics frame when the mouse exits a Collider of this GameObject
  - **`OnMouseDown()`**: Called if the player presses the primary mouse button down while the mouse is over a Collider of this GameObject
  - **`OnMouseDrag()`**: Called every physics frame that the player holds the primary mouse button down while the mouse is over a Collider of this GameObject
  - **`OnMouseUp()`**: Called if the player releases the primary mouse button while the mouse is over a Collider of this GameObject
  - **`OnMouseUpAsButton()`**: Called if the player presses and then releases the primary mouse button while the mouse is over a Collider of this GameObject. You should use **`OnMouseUpAsButton()`** when you want to be sure that the player pressed and then released the mouse button over the same GameObject.

- **Physics2D Messages**: There are also 2D versions of all the OnCollision and OnTrigger messages in the preceding list that are called by Unity's 2D physics system, but I felt it a bit redundant to list them all here. Examples include:
  - **`OnCollisionEnter2D(Collision2D collision)`**
  - **`OnTriggerStay2D(Collider2D collider)`**

Several more Unity messages can be found on the Unity API documentation page for the **`MonoBehaviour`** class.[15] You can also see a list of them in Visual Studio:

- **macOS**: From the Visual Studio menu bar, choose *Edit > Add Unity Event Functions...* or press Command-Shift-M to open a window that lists every Unity Message along with a brief description.
- **Windows**: Press Ctrl+Shift+M to open a slightly less helpful window than in macOS that lists every Unity message but has no description of them.

15. At https://docs.unity3d.com/2020.3/Documentation/ScriptReference/MonoBehaviour.html under the "Messages" heading.

For more information on when these various MonoBehaviour messages are called, you can check out Unity's documentation at:

https://docs.unity3d.com/2020.3/Documentation/Manual/ExecutionOrder.html

Variable Scope

The *scope* of variables is an important concept in any programming language. A variable's scope refers to how much of the code is aware of the variable's existence. *Global* scope means that any code anywhere can see and reference the variable, whereas *local* scope means that the variable's scope is limited in some way, and it can't be seen by everything in the code. If a variable is local to a class, then only other things within the class can see it. If a variable is local to a function, then it only exists within that function and is destroyed when the function has completed.

The **ScopeExample** class in Code Listing B.26 demonstrates several different levels of scope for variables within a single class. Lettered comments after the code explain what is happening on important lines. A variable that is **maroon** in the code indicates that it is *out of scope* in that section of the code.

Code Listing B.26 ScopeExample.cs

```
using UnityEngine;

public class ScopeExample : MonoBehaviour {
    // public fields (public class variables)
    public bool                 trueOrFalse = false;                        // a
    public int                  graduationAge = 18;
    public float                goldenRatio = 1.618f;

    // private fields (private class variables)
    private bool                _hiddenVariable = true;                     // b
    private float               _anotherHiddenVariable = 0.5f;

    // protected fields (protected class variables)
    protected int               partiallyHiddenInt = 1;                     // c
    float                       anotherProtectedVariable = 1.0f;

    // static public fields (static public class variables)
    static public int           NUM_INSTANCES = 0;                          // d
    static private int          NUM_TOO = 0;                                // e

    protected bool hiddenVariableAccessor {                                 // f
        get { return _hiddenVariable; }
    }

```

```
>     void Awake() {
>         trueOrFalse = true;              // OK: assigns "true" to trueOrFalse      // g
>         print( "tOF: " + trueOrFalse );  // OK: prints "tOF: True"
>         int ageAtTenthReunion = graduationAge + 10;  // OK                         // h
>         print( "_aHV: " + _anotherHiddenVariable );  // OK: prints "_aHV: 0.5"     // i
>         NUM_INSTANCES += 1;              // OK                                     // j
>         NUM_TOO++;                       // OK                                     // k
>     }
|
|     void Update() {
>         print( ageAtTenthReunion );      // ERROR                                  // l
>         float ratioed = 1f;              // OK
>         for (int i=0; i<10; i++) {       // OK                                     // m
>             ratioed *= goldenRatio;      // OK
>         }
>         print( "ratioed: " + ratioed );  // OK: prints "ratioed: 122.9661"
>         print( i );                      // ERROR                                  // n
|     }
| }
```

a. **Public fields**: The variables **`trueOrFalse`**, **`graduationAge`**, and **`goldenRatio`** are all **`public`** *fields*. *Fields* are class instance variables, meaning that they are declared as part of the class and are visible to all functions within any instance of that class. Because these fields are *public*, they are inherited by the subclass **`ScopeExampleSubclass`** (in Code Listing B.27), which means that **`ScopeExampleSubclass`** also has a **`bool trueOrFalse`**. Public variables can also be seen by any other code that has a reference to an instance of the class. This would allow a function with the variable **`ScopeExample se`** to see and set the field **`se.trueOrFalse`**.

b. **Private fields**: These two variables are **`private`** fields. *Private* fields can only be seen by this instance of ScopeExample (meaning that an instance of ScopeExample can access and modify its own private fields, but no other instance can see them). Subclasses do not inherit private fields, so the subclass **`ScopeExampleSubclass`** does not have a **`bool _hiddenVariable`**. A function with the variable **`ScopeExample se`** would not be able to see or access the private field **`se._hiddenVariable`**.

c. **Protected fields**: A field marked **`protected`** is between public and private (in terms of access level), and all fields are **`protected`** by default unless they are explicitly marked **`private`** or **`public`**. Subclasses *do* inherit protected fields, so the subclass **`ScopeExampleSubclass`** in Code Listing B.27 does inherit the **`int partiallyHiddenInt`** that is declared in **`ScopeExample`**. However, anything outside of the **`ScopeExample`** class or subclasses thereof would not be able to see or access the protected field **`se.partiallyHiddenVariable`**.

d. **Static fields**: A `static` field is a field that is shared by the entire class, not the instances of the class. This means that **NUM_INSTANCES** is accessed as **ScopeExample.NUM_INSTANCES**. Public static fields are the closest thing to global scope that I use in C#. Any script in the codebase can access the field **public static ScopeExample.NUM_INSTANCES**, and the value of **NUM_INSTANCES** is shared by all instances of **ScopeExample**. A function with the variable **ScopeExample se** could not access **se.NUM_INSTANCES** (because it doesn't exist), but it could access **ScopeExample.NUM_INSTANCES**. The **ScopeExampleSubclass** subclass of **ScopeExample** can also access **NUM_INSTANCES**. Within an instance of **ScopeExample**, **NUM_INSTANCES** can be accessed directly (without the "**ScopeExample.**" prefix).

e. **NUM_TOO** is a **private static** field, which means that all instances of **ScopeExample** share the same value of **NUM_TOO**, but no other class can see it or access it. The subclass **ScopeExampleSubclass** cannot access **NUM_TOO**.

f. **hiddenVariableAccessor** is a read-only **public** property that allows other classes access to **_hiddenVariable**. Because there is no **set** clause, it is read-only.

g. The **// OK** comment means that this line executes without any errors. **trueOrFalse** is a **public** field of **ScopeExample**, so this method of **ScopeExample** can access it.

h. This line declares and defines a variable named **ageAtTenthReunion** that is locally scoped to the method **ScopeExample.Awake()**. This means that when the **ScopeExample.Awake()** function has finished executing, the variable **ageAtTenthReunion** will cease to exist. Also, nothing outside of this function can access or modify **ageAtTenthReunion**.

i. As a **private** field **_anotherHiddenVariable** can only be seen by methods within instances of this class.

j. Within a class, **static public** fields can be referred to by their name, meaning that the **ScopeExample.Awake()** method can reference **NUM_INSTANCES** without needing the class name before it.

k. **NUM_TOO** can also be accessed anywhere within the ScopeExample class.

l. The **// ERROR** comment means that this line will not run properly. This line throws an error because **ageAtTenthReunion** was a local variable of the method **ScopeExample.Awake()**, so it has no meaning in the **ScopeExample.Update()** method.

m. The variable **int i** is declared and defined in this **for** loop and is locally scoped to the **for** loop. This means that **i** ceases to have meaning when the **for** loop has completed.

n. This line throws an error because **i** has no meaning outside of the preceding **for** loop.

ScopeExampleSubclass in Code Listing B.27 extends the **ScopeExample** class, demonstrating how class inheritance can affect scope (**public** and **protected** fields can be seen by subclasses, but **private** fields cannot).

Code Listing B.27 ScopeExampleSubclass.cs

```
using UnityEngine;

public class ScopeExampleSubclass : ScopeExample {                              // a
    void Start() {
        print( "tOF: " + trueOrFalse );              // OK: prints "tOF: True"     // b
        print( "pHI: " + partiallyHiddenInt );       // OK: prints "pHI: 1"        // c
        print( "_hV: " + _hiddenVariable );          // ERROR                      // d
        print( "NI:  " + NUM_INSTANCES );            // OK: prints "NI:  1"        // e
        print( "NT:  " + NUM_TOO );                  // ERROR                      // f
        print( "hVA: " + hiddenVariableAccessor );   // OK: prints "hVA: True"     // g
    }
}
```

a. This line declares and defines **ScopeExampleSubclass** as a subclass of **ScopeExample**. As a subclass, **ScopeExampleSubclass** has access to the **public** and **protected** fields and methods of **ScopeExample** but not to the **private** fields or methods. Because the **Awake()** and **Update()** methods of **ScopeExample** were not explicitly declared **public** or **private**, they are by default **protected** and therefore inherited by **ScopeExampleSubclass**. Because **ScopeExampleSubclass** does not have its own **Awake()** or **Update()** functions defined, it will run the versions defined in its base class, **ScopeExample**. And, when those **ScopeExample** methods run, they will also print to the Console (or they would if there were no errors in them).

b. **trueOrFalse** is **public**, so **ScopeExampleSubclass** has inherited a **trueOrFalse** field. Additionally, because the base class (**ScopeExample**) version of **Awake()** has already run by the time **Start()** is called on **ScopeExampleSubclass**, **trueOrFalse** has already been set to **true** by the **Awake()** method.

c. **ScopeExampleSubclass** also has a **protected** **partiallyHiddenInt** field that is inherited from **ScopeExample**.

d. **_hiddenVariable** is not inherited from **ScopeExample** because it is **private**.

e. **NUM_INSTANCES** is accessible by **ScopeExampleSubclass** because as a **public** variable, it is inherited from the base class **ScopeExample**. The two classes *share the same value* for **NUM_INSTANCES**, so if an instance of each class were

instantiated, the value of **NUM_INSTANCES** would be **2** regardless of whether it was accessed from **ScopeExample** or **ScopeExampleSubclass**.

f. As a **private static** variable, **NUM_TOO** is not inherited by **ScopeExampleSubclass**. However, it's worth noting that even though **NUM_TOO** is not inherited, when **ScopeExampleSubclass** is instantiated and calls the base class version of **Awake()**—that is, the **Awake()** method that is defined in the **ScopeExample** base class—that call to the **Awake()** method *can* access **NUM_TOO** *without errors*, because the base class version is running within the scope of the **ScopeExample** base class even though it's actually running on an instance of **ScopeExampleSubclass**.

g. In our most esoteric example, **ScopeExampleSubclass** can read the **public** property **hiddenVariableAccessor**, which easily makes sense until you look a bit deeper. Inside of the **get** clause of **hiddenVariableAccessor**, it reads the **private** field **_hiddenVariable**. This is a subtle but important aspect of variable scope. Because **ScopeExampleSubclass** extends **ScopeExample**, all the **private** fields of **ScopeExample** are created for a **ScopeExampleSubclass** instance, even though the **ScopeExampleSubclass** instance can't access them directly. The **ScopeExampleSubclass** instance *can* use **public** accessors like **hiddenVariableAccessor**—which is scoped to the **ScopeExample** base class—to access **private** fields (like **_hiddenVariable**) that are also scoped to the **ScopeExample** base class. Inherited methods like the **Awake()** that **ScopeExampleSubclass** inherited from **ScopeExample** can also access **private** fields of the base class.

These many notes have included both very simple and very complex examples of variable scope. If some of it didn't make sense to you, that's okay. You can come back and read it later after you've used C# some more and have more specific scope questions.

XML

XML (eXtensible Markup Language) is a file format that is designed to be both flexible and human-readable. In the first two editions of this book, XML was used to define the cards in the deck and the layout of the tableau in the *Prospector Solitaire* game from Chapters 33 and 34. I find XML to be much more easily read by humans, but the ease of use of the integrated JSONUtility class in Unity prompted me to switch to JSON for this third edition.

XML Listing B.28 shows the XML version of the JSON_Deck.js file that you can see in JSON Listing B.15. Additional spaces have been added to the XML in B.28 to make it a little more readable, but that is okay because XML generally treats any number of spaces or line breaks as a single space. I've also included the XML_Deck.xml and XML_Layout.xml files in the UnityPackage that you download at the beginning of the *Prospector*

Solitaire project, so you can open those and compare them with the JSON versions. (The syntax coloring in XML Listing B.27 is the same as that for XML files opened in Visual Studio Code, which I find to be excellent for opening and editing XML files.)

XML Listing B.28 XML_Deck.xml Showing the Same Data as JSON_Deck.json

```
<xml>
    <!-- decorators are the suit and rank in the corners of every card. -->
    <decorator type="letter" x="-1.05" y="1.42"  z="0" flip="0" scale="1.25" />
    <decorator type="suit"   x="-1.05" y="1.03"  z="0" flip="0" scale="0.4"  />
    <decorator type="suit"   x="1.05"  y="-1.03" z="0" flip="1" scale="0.4"  />
    <decorator type="letter" x="1.05"  y="-1.42" z="0" flip="1" scale="1.25" />
    <!-- A list of all cards that defines where pips are placed. -->
    <card rank="1">
        <pip x="0" y="0"    z="0"  flip="0" scale="2" />
    </card>
    <card rank="2">
        <pip x="0" y="1.1"  z="0"  flip="0" />
        <pip x="0" y="-1.1" z="0"  flip="1" />
    </card>
    <card rank="3">
        <pip x="0" y="1.1"  z="0"  flip="0" />
        <pip x="0" y="0"    z="0"  flip="0" />
        <pip x="0" y="-1.1" z="0"  flip="1" />
    </card>
    <!-- Cards of rank 4-12 are skipped here-->
    <!-- face references the name of the Sprite to be shown on face cards. -->
    <card rank="13" face="FaceCard_13" />
</xml>
```

Even without knowing much at all about XML, you should be able to read this somewhat. XML is based on *tags* (also known as the *markup* of the document), which are the words between the two angle brackets (e.g., **<xml>**, **<card rank="2">**). Most XML *elements* have an *opening tag* (e.g., **<card rank="2">**) and a *closing tag* that contains a forward slash immediately after the opening angle bracket (e.g., **</card>**). Anything between the opening and closing tags of an element is said to be the *content* of that element (e.g., the **<pip …/>** tags between the **<card>** and **</card>** in the XML listing are the content of that **<card>**). There are also *empty-element tags*, which are tags that serve as both the opening and closing tag with no content between them. For example, in the XML listing, the tag **<pip x="0" y="1.1" z="0" flip="0" />** is an empty-element tag that requires no matching **</pip>** tag because it ends in **/>**. In general, XML files should start with **<xml>** and end with **</xml>**, so everything in the XML document is content of the **<xml>** element.

XML tags can have *attributes*, which are like fields in C#. The empty-element **<pip x="0" y="1.1" z="0" flip="0" />** that is seen in the XML listing includes **x**, **y**, **z**, and **flip** attributes.

In an XML file, anything between **<!--** and **-->** is a *comment* and is therefore ignored by any program that is reading the XML file. In XML Listing B.28, you can see that I use them the same way that I use comments in C# code.

A robust XML reader is included in C# .NET, but I have found it to be very large (it adds about 1 MB to the size of your compiled application, which is a lot if you're making something for mobile or WebGL) and unwieldy (using it is not simple). Though we no longer use it in the *Prospector Solitaire* tutorial, I've included a simple XML interpreter called **PT_XMLReader** in the ProtoTools scripts that are part of the unitypackage imported at the beginning of the later prototyping chapters. **PT_XMLReader** is not at all as robust as the .NET implementation, but it is tiny. Comments in the PT_XMLReader.cs file explain its use.

XML Documentation in C#

One excellent use of XML in Unity and C# is XML Documentation, which allows you to add documentation to your functions, classes, structs, methods, and so on that can be read and shown by Visual Studio. Code Listing B.29 shows several examples.

Code Listing B.29 C# XML Documentation in Utils.cs with XML Tags Bolded

```
/// <summary>
/// This two-dimensional array version of a Bézier curve solver is the most
///   performant I've developed so far that can handle any number of points.
/// <para>LerpUnclamped is used to allow for extrapolation.</para>
/// </summary>
/// <param name="u">The amount of interpolation [0..1]</param>
/// <param name="arr">A params array of points to interpolate</param>
/// <returns>The interpolated value</returns>
static public Vector3 Bezier( float u, params Vector3[] arr ) {…}                    // a



/// <summary>
/// This overload converts the List <paramref name="pts"/> to an array and calls
/// <see cref="Bezier(float, Vector3[])">Bezier()</see>.
/// </summary>
/// <param name="u">The amount of interpolation [0..1]</param>
/// <param name="pts">A List of points to interpolate</param>
/// <returns>The interpolated value</returns>
static public Vector3 Bezier( float u, List<Vector3> pts) {…}                        // b

/// <summary>
/// <para>While most Bézier curves are 3 or 4 points, it is possible to have
///   any number of points using this recursive function.</para>
/// <para>This has been replaced by <see cref="Bezier(float, Vector3[])">
/// Bezier()</see>, a faster array-based version.</para>
```

```
/// <para>LerpUnclamped is used to allow extrapolation.</para>
/// </summary>
/// <param name="u">The amount of interpolation [0..1]</param>
/// <param name="pts">A List of points to interpolate</param>
/// <param name="iL">Index of the left extent of the used elements of pts.
///   Defaults to 0.</param>
/// <param name="iR">Index of the right extent of the used elements of pts.
///   Defaults to -1, which is changed to the last element's index.</param>
/// <returns>The interpolated value</returns>
static public Vector3 BezierR(float u,List<Vector3> pts,int iL=0,int iR=-1) {…}      // c
```

Figure B.7 shows the effects of these XMLDoc tags when the developer hovers their mouse over the name of each function in Visual Studio (macOS). The a, b, and c labels in Figure B.7 match `// a`, `// b`, and `// c` in Code Listing B.29.

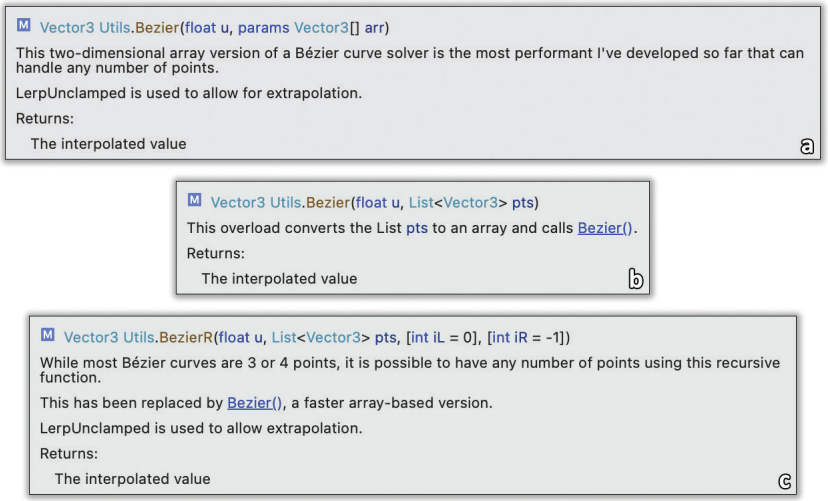


Figure B.7 XMLDoc pop-ups for the `// a`, `// b`, and `// c` functions in Code Listing B.29

Some of the tags you see here are :

- **`<summary>`**: Text between the summary tags is the primary content shown and should describe the function.
- **`<para>`**: The para tags encapsulate a paragraph, causing a new line in the pop-up.
- **`<param name="u">`**: These allow you to define help text for each parameter.
- **`<see cref="Bezier(float, Vector3[])">`**: The **`see`** tag allows you to create links to other functions within your summary description.
- **`<returns>`**: Allows you to describe the return value of the function (if it's not **`void`**).

Additional options are listed in the Microsoft documentation for XMLDoc,[16] but some of them do not work properly in Visual Studio for Mac.

Math Concepts

A lot of people cringe when they hear the word *math,* but that really doesn't need to be the case. As you'll see throughout this book, it can do some really cool things for you. In the following sections, I cover just a few cool math concepts that can help you in game development.

Cosine and Sine (Cos and Sin)

Sine and cosine are functions that convert an angle value Θ (theta) into a point along a wave shape that ranges in the y dimension from -1 to 1. They are shown in Figure B.8.

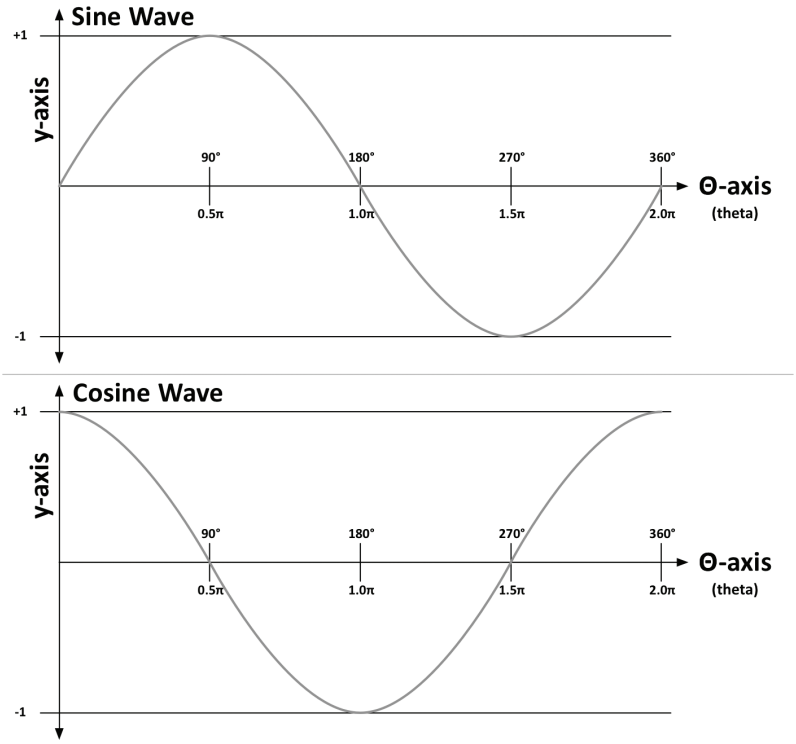


Figure B.8 The traditional representations of sine and cosine

16. https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/xmldoc/recommended-tags — accessed August 27, 2021.

However, sine and cosine are much more than just waves; they're descriptions of *the relationship of x and y when going around a circle*. I'll demonstrate what I mean with some code.

Unity Example—Sine and Cosine

The following steps will create a visualization that I think does an excellent job of showing the relationship between sine, cosine, and circular movement.

1. Create a new Unity project named *Sine and Cosine* based on the *3D Core* template and create a new Basic scene named *_Scene_Cyclic*.
2. At the top of the Scene pane, use the *Effects Toggle* pop-up to turn off the default *Skybox* (i.e., the Skybox option should not be checked, as shown in Figure B.9a). This will make the background of the Scene pane a dark gray.
3. Give the Main Camera the settings shown highlighted in Figure B.9b.
4. Click the Gizmos button in the top-right corner of the Game pane until it is highlighted. This will enable the **`OnDrawGizmos()`** call in Code Listing B.30 to draw to the Game pane (in addition to the Scene pane).
5. Create a new sphere in the scene (*GameObject* > *3D Object* > *Sphere*). Set Sphere's transform to P:[0, 0, 0], R:[0, 0, 0], S:[**0.1**, **0.1**, **0.1**] as shown in Figure B.9c (the figure does not show the Sphere (Mesh Filter), Mesh Renderer, or Sphere Collider components, but please *do not* delete them).
6. Add a *TrailRenderer* to Sphere and give it the settings shown highlighted in Figure B.9c.
   a. Select *Sphere* in the Hierarchy and choose *Component* > *Effects* > *Trail Renderer* from the menu bar.
   b. Set *Width = 0.10* (this setting is at the top-left of the graph area).
   c. Set *Time = 1*.
   d. Open the *disclosure triangle* next to *Materials* in the Sphere's TrailRenderer Inspector and click the circular target to the right of *Element 0* to select *Default-Particle* as the texture for the TrailRenderer.
7. Create a new C# script named *Cyclic*.
   a. Attach the *Cyclic* script to the *Sphere* GameObject in the Hierarchy.
   b. Open the *Cyclic* script in Visual Studio, and enter the code in Code Listing B.30.

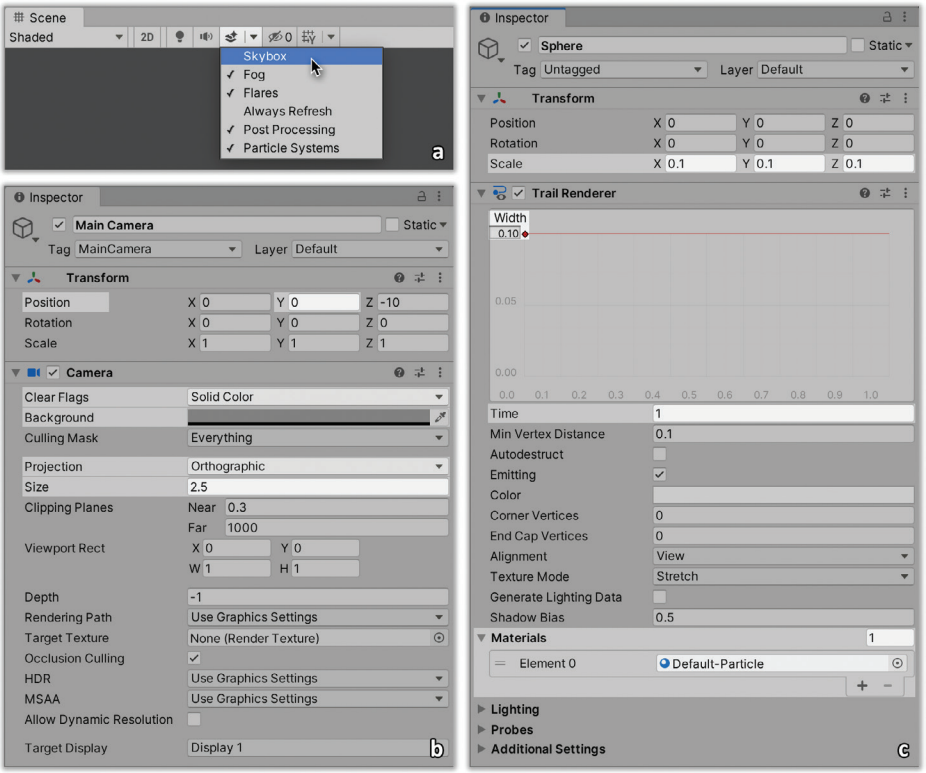


Figure B.9 Settings for the _Scene_Cyclic GameObjects

Code Listing B.30 Cyclic.cs — Exploring Sine and Cosine

```
using UnityEngine;

public class Cyclic : MonoBehaviour {
    [Header( "Inscribed" )]
    public float   theta = 0;  // The 0 here is a zero (not a theta symbol)
    public bool    showCosAsX = false;
    public bool    showSinAsY = false;

    [Header( "Dynamic" )]
    public Vector3 pos;

    void Update() {
        // Calculate radians based on time
        float radians = Time.time * Mathf.PI;

```

```
>         // Convert radians to degrees to show in the Inspector
>         // The "% 360" limits the value to the range: 0 to 359.9999
>         theta = Mathf.Round( radians * Mathf.Rad2Deg ) % 360;
>         // Reset pos
>         pos = Vector3.zero;
>         // Calculate x & y based on cos and sin respectively
>         pos.x = Mathf.Cos( radians );
>         pos.y = Mathf.Sin( radians );
>
>         // Use sin and cos if they are checked in the Inspector
>         Vector3 tPos = Vector3.zero;
>         if ( showCosAsX ) tPos.x = pos.x;
>         if ( showSinAsY ) tPos.y = pos.y;
>         // Position this.gameObject (the Sphere)
>         transform.position = tPos;
|     }
|
>     void OnDrawGizmos() {
>         if ( !Application.isPlaying ) return;// Only show when Playing
>
>         // Draw wavy rainbow lines showing the sin and cos curves
>         int inc = 10;
>         for ( int i = 0; i < 360; i += inc ) {
>             int i2 = i + inc;
>             float c0 = Mathf.Cos( i  * Mathf.Deg2Rad );
>             float c1 = Mathf.Cos( i2 * Mathf.Deg2Rad );
>             float s0 = Mathf.Sin( i  * Mathf.Deg2Rad );
>             float s1 = Mathf.Sin( i2 * Mathf.Deg2Rad );
>             Vector3 vC0 = new Vector3( c0,   -1.2f - ( i  / 360f ),  0 );
>             Vector3 vC1 = new Vector3( c1,   -1.2f - ( i2 / 360f ),  0 );
>             Vector3 vS0 = new Vector3( 1.2f + ( i  / 360f ),   s0,   0 );
>             Vector3 vS1 = new Vector3( 1.2f + ( i2 / 360f ),   s1,   0 );
>
>             Gizmos.color = Color.HSVToRGB( i / 360f, 1, 1 );
>             Gizmos.DrawLine( vC0, vC1 );
>             Gizmos.DrawLine( vS0, vS1 );
>         }
>
>         // Draw the lines and circles relative to the Sphere GameObject
>         Gizmos.color = Color.HSVToRGB( theta / 360f, 1, 1 );
>         // Show individual Sin and Cos aspects using Gizmos
>         Vector3 cosPos = new Vector3( pos.x, -1.2f - ( theta / 360f ), 0 );
>         Gizmos.DrawSphere( cosPos, 0.05f );
>         if ( showCosAsX ) Gizmos.DrawLine( cosPos, transform.position );
>
>         Vector3 sinPos = new Vector3( 1.2f + ( theta / 360f ), pos.y, 0 );
>         Gizmos.DrawSphere( sinPos, 0.05f );
>         if ( showSinAsY ) Gizmos.DrawLine( sinPos, transform.position );
>     }
| }
```

8. You can optionally set the Scene view to 2D by clicking the *2D* button at the top of the Scene pane. If you zoom in, this 2D view will give you a grid around the Sphere.
9. Click *Play*.

Initially, the Sphere doesn't move, but colored dots move along paths below and to the right of the sphere (you may need to zoom out to see it all). The dot on the right follows the wave defined by **`Mathf.Sin(theta)`**, and the dot below follows the **`Mathf.Cos(theta)`** wave.

If you check the box next to **`showCosAsX`** in the *Sphere:Cyclic (Script)* Inspector, Sphere will start moving in the X direction following a cosine wave. You can see how the X motion of Sphere is connected directly to the cosine motion of the bottom wave.

Uncheck **`showCosAsX`** and check **`showSinAsY`**. Now you can see how the Y motion of Sphere is connected to the sine wave. If you check both **`showCosAsX`** and **`showSinAsY`**, Sphere will move in the circle defined by combining X = cos(Θ) and Y = sin(Θ). A full circle is 360°, or 2π radians (i.e., 2 * **`Mathf.PI`**).

This connection is also shown in Figure B.10, which uses similar colors to those in the Unity example.

This is important because it means that *you can use sine and cosine for all sorts of circular or cyclic behavior*! These properties of sine and cosine are used in the Chapter 32, "*Space SHMUP* — Part 2," project to define wavy movement for the Enemy_1 enemy type and to adjust the linear interpolation easing of the Enemy_2 type (see the "Interpolation" section in this appendix for information about linear interpolation and easing).

Dice Probability Enumeration

Chapter 11, "Math and Game Balance," covered Jesse Schell's Rule 4 of probability: Enumeration can solve difficult math problems. Here is a quick Unity program that can enumerate all the possibilities for any number of dice with any number of sides. However, beware that each die you add *drastically* increases the number of calculations that must be done (e.g., 5d6 [five six-sided dice] take six times longer to calculate than 4d6 and 36 times longer to calculate than 3d6).

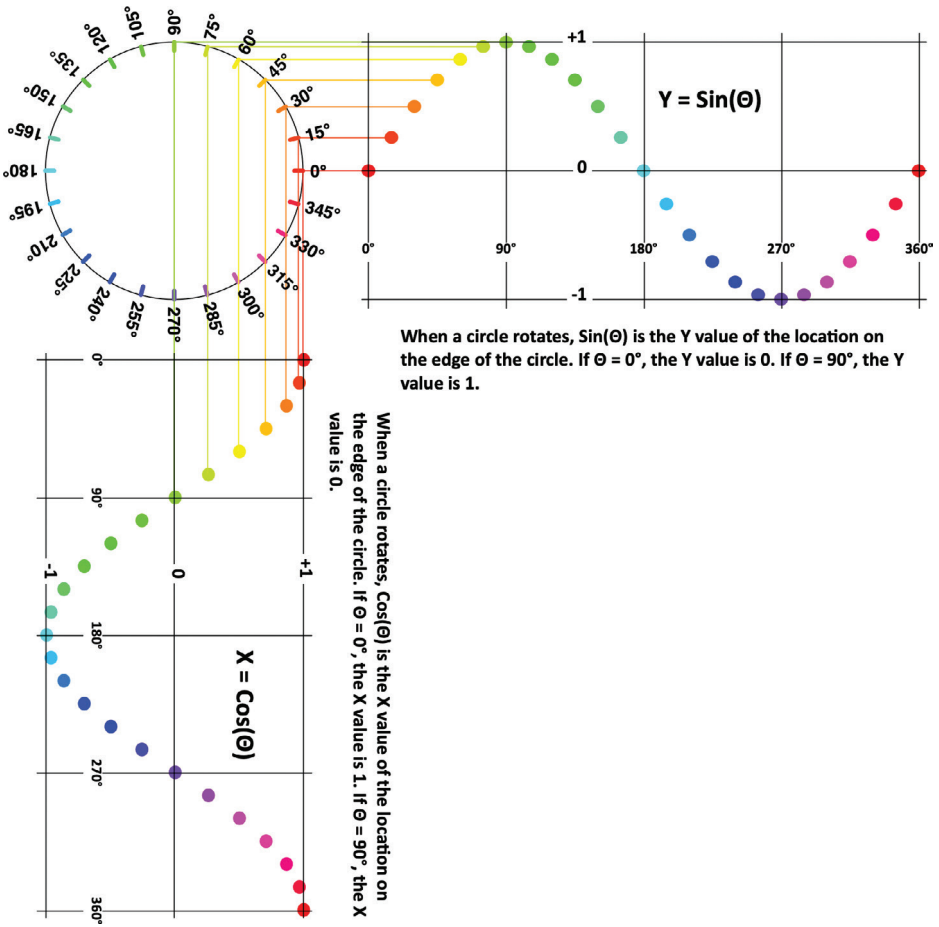


Figure B.10 The relationship of sine and cosine to a circle

Unity Example—Dice Probability

Follow these steps to create a program that will enumerate all possibilities for any number of identical dice with any number of sides. The default for this code is 2d6 (two six-sided dice). With these default values, the program will run through all possible rolls of the two dice (e.g., 1|1, 1|2, 1|3, 1|4, 1|5, 1|6, 2|1, 2|2, ... 6|5, 6|6) and track the sum of the dice for each possibility.

1. Start a new Unity project named *Dice Probability* based on the *3D Core* template.
2. Create a new C# script named *DiceProbability* and attach it to the *Main Camera* in the Scene pane. (The Main Camera settings will be handled in **`Awake()`** this time.)
3. Click the *Gizmos* button in the top-right corner of the Game pane until it is highlighted. This will enable the **`OnDrawGizmos()`** call in Code Listing B.31 to draw to the Game pane.

4. Open *DiceProbability* in Visual Studio and enter the code in Code Listing B.31.

Code Listing B.31 DiceProbability.cs

```
using System.Collections;
using UnityEngine;

public class DiceProbability : MonoBehaviour {
    [Header("Inscribed")]
    public int      numDice = 2;
    public int      numSides = 6;
    [Tooltip("Check this box in the inspector to begin rolling")]
    public bool     checkToCalculate = false;
    [Tooltip("The CalculateRolls() coroutine yields after maxIterations rolls")]
    public int      maxIterations = 10000;
    public float    width = 16;
    public float    height = 9;

    [Header("Dynamic")]
    public int[]    dice;  // An array of the values of each die
    public int[]    rolls; // An array storing how many times a roll has come up
    // ^ For 2d6 this would be [ 0, 0, 1, 2, 3, 4, 5, 6, 5, 4, 3, 2, 1 ],
    //    meaning that a 2 was rolled once while a 7 was rolled 6 times.

    void Awake() {
        // Set up the main camera to properly display the graph
        Camera cam = Camera.main;
        cam.clearFlags = CameraClearFlags.SolidColor;
        cam.backgroundColor = Color.black;
        cam.orthographic = true;
        cam.orthographicSize = 5;
        cam.transform.position = new Vector3(8, 4.5f, -10);
    }

    void Update() {
        if (checkToCalculate) {
            StartCoroutine( CalculateRolls() );
            checkToCalculate = false;
        }
    }

    void OnDrawGizmos() {
        float minVal = numDice;
        float maxVal = numDice*numSides;

        // If the rolls array is not ready, return
        if (rolls == null || rolls.Length == 0 || rolls.Length != maxVal+1) {
            return;
        }
```

```
>
>         // Draw the rolls array
>         float maxRolls = Mathf.Max(rolls);
>         float heightMult = 1f/maxRolls;
>         float widthMult = 1f/(maxVal-minVal);
>
>         Gizmos.color = Color.white;
>         Vector3 v0, v1 = Vector3.zero;
>         for (int i=numDice; i<=maxVal; i++) {
>             v0 = v1;
>             v1.x = ( (float) i - numDice ) * width * widthMult;
>             v1.y = ( (float) rolls[i] ) * height * heightMult;
>             if (i != numDice) {
>                 Gizmos.DrawLine(v0,v1);
>             }
>         }
>     }
|
>     public IEnumerator CalculateRolls() {
>         // Calculate max value (the maximum possible value that could be rolled
>         //  on the dice (for example, for 2d6 maxValue = 12)
>         int maxValue = numDice*numSides;
>         // Make the array large enough to hold all possible values
>         rolls = new int[maxValue+1];
>
>         // Make an array with an element for each die. All are preset to a value
>         //  of 1 except for the first die which is set to 0 (to make the
>         //  method RecursivelyAddOne() work properly)
>         dice = new int[numDice];
>         for (int i=0; i<numDice; i++) {
>             dice[i] = (i==0) ? 0 : 1;
>         }
>
>         // Iterate on the dice.
>
>         int iterations = 0;
>         int sum = 0;
>
>         // Usually, I avoid while loops because they can lead to infinite loops,
>         //  but because this is a coroutine with a yield in the while loop, it's
>         //  not as big of a problem.
>         while (sum != maxValue) {
>             // ^ sum will == maxValue when all dice are at their maximum value
>
>             // Increment the 0th die in the dice Array
>             RecursivelyAddOne(0);
>
>             // Sum all the dice together
>             sum = SumDice();
```

```
>             // and add 1 to that position in the rolls array
>             rolls[sum]++;
>
>             // add to iterations and yield every 10,000 iterations (by default)
>             iterations++;
>             if (iterations % maxIterations == 0) {
>                 yield return null;
>             }
>         }
>         print("Calculation Done");
>
>         // Use StringBuilder to build complex strings (see earlier in Appendix B)
>         System.Text.StringBuilder sb = new System.Text.StringBuilder();
>         sb.AppendLine( "Dice Rolls for " + numDice + "d" + numSides + ":" );
>         for (int i=numDice; i<=maxValue; i++) {
>             // ToString("#,###") adds commas to separate each three digits
>             sb.AppendLine( i + "\t" + rolls[i].ToString("#,##0") );
>         }
>
>         int totalRolls = 0;
>         foreach (int i in rolls) {
>             totalRolls += i;
>         }
>         sb.AppendLine( "\nTotal Rolls: " + totalRolls.ToString("#,##0") );
>
>         print(sb);
>     }
|
>     // This is a recursive method, meaning that it calls itself. You can read
>     //  about recursive methods more later in this appendix.
>     public void RecursivelyAddOne(int ndx) {
>         if (ndx == dice.Length) return; // We've exceeded the length of dice
>         //  Array, so just return
>
>         // Increment the die at position ndx
>         dice[ndx]++;
>         // If this exceeds the capacity of the die...
>         if (dice[ndx] > numSides) {
>             dice[ndx] = 1;               // then set this die to 1...
>             RecursivelyAddOne(ndx+1);    // and increment the next die
>         }
>         return;
>     }
|
>     public int SumDice() {
>         // Sum the values of all the dice in the dice array
>         int sum = 0;
>         for (int i=0; i<dice.Length; i++) {
>             sum += dice[i];
```

```
>         }
>         return(sum);
>     }
| }
```

5. To use the DiceProbability enumerator, click *Play* and then select *Main Camera* in the Hierarchy pane.
6. In the *Main Camera:Dice Probability (Script)* Inspector, you can set **numDice** (the number of dice) and **numSides** (the number of sides for each die) and then click **checkToCalculate** to calculate the probability of any specific number coming up on those dice (Unity must be playing for **checkToCalculate** to do anything).

Unity enumerates all the possible results and then outputs the results to the Console pane as well as a graph in the Game pane (as long as you have Gizmos turned on in the Game pane).

Try it first with the default setting of 2 dice of 6 sides each (2d6) and you'll get these results in the console (you will have to click the message in the console to see more than the first two lines):

```
Dice Rolls for 2d6:
2       1
3       2
4       3
5       4
6       5
7       6
8       5
9       4
10      3
11      2
12      1

Total Rolls: 36
```

7. In the Inspector, try setting **numDice**=8 and **numSides**=6. Then check **checkToCalculate**.

You'll see that this takes a lot longer to calculate and that the results (and the curve graph) progressively update each time the coroutine yields (see the "Coroutines" section in this appendix). To speed this up, try setting **maxIterations**=100,000.

maxIterations is the number of die rolls that the code will calculate before the coroutine yields and allows Unity to show you the results. The more **maxIterations** you allow, the faster the overall calculation will complete because the code will calculate more rolls in between showing you results. Choosing fewer **maxIterations** will show you results more frequently, but that will drastically slow down the overall time to calculate.

Now any time you want to know the probability of something like rolling a 13 or 28 on 8d6, you can figure it out through enumeration. The following are some salient lines from the console output.

```
Dice rolls for 8d6:
8     1
9     8
…
12    330
13    792
14    1,708
…
27    133,288
28    135,954  // 28 (the average roll) is 171.66 times more likely than a 13
29    133,288
…
47    8
48    1

Total Rolls: 1,679,616
```

This means that the probability of rolling a 13 on 8D6 is 792 / 1,679,616 = 11 / 23,328 ≈ 0.00047 ≈ 0.05%.

You could also modify this code to roll the dice a specified number of times and choose random rolls each time. With a high number of rolls, this would give you a practical probability instead of the theoretical probability that is currently produced (see Jesse Schell's Rule 9 of probability in Chapter 11, "Math and Game Balance").

Using Data-Oriented Design to Improve the DiceProbability Code

The code in Code Listing B.31 was written for the first edition of this book, and it's horribly inefficient. On the other hand, even as inefficient as it is, it's millions of times faster than calculating the dice by hand, so for what it is, it's fine. However, after you've read Chapter 28, "Data-Oriented Design," I challenge you to come back to this code and rewrite it from a DOD mindset. I'm certain that a multithreaded version would be significantly faster, though you may need to schedule the jobs in multiple batches to avoid halting other operations while this was calculating (which is why I used a coroutine in the original code).

Dot Product

Another extremely useful math concept is the dot product. A dot product of two vectors is the result of multiplying the respective X, Y, and Z components of each vector together and adding the results, as shown in Code Listing B.32.

Code Listing B.32 Dot Product Calculation

```
> Vector3 a = new Vector3( 1, 2, 3 );
> Vector3 b = new Vector3( 4, 5, 6 );
> float dotProduct = a.x*b.x + a.y*b.y + a.z*b.z;                    // a
>    // dotProduct =   1*4   +   2*5   +   3*6
>    // dotProduct =   4     +   10    +   18
>    // dotProduct =   32
> dotProduct = Vector3.Dot(a,b); // This is the best way to do it in Unity      // b
```

a. A manual calculation of the dot product of **`Vector3`**s **a** and **b**.

b. The same calculation performed using the built-in static method **`Vector3.Dot()`**.

At first, dot products might not seem very important, but they have an *extremely* useful property: The float that the dot product *a*•*b*[17] returns is equivalent to *a*.magnitude * *b*.magnitude * Cos(Θ) where Θ is the angle between the two vectors, as shown in Figure B.11.

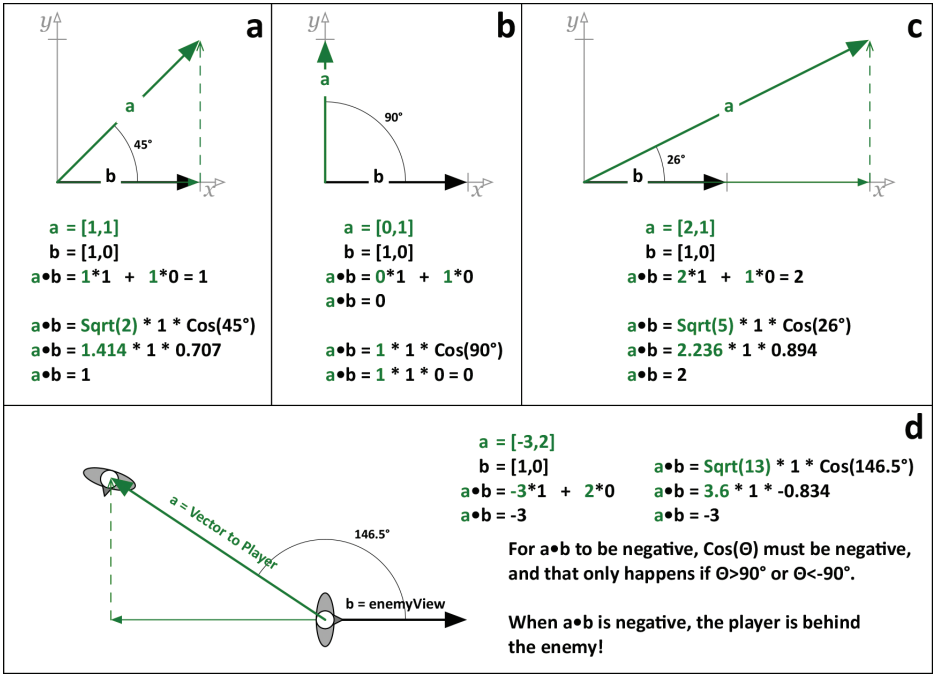


Figure B.11 Dot product examples (decimal numbers are approximate values)

17. The • symbol is used here (and commonly in math) to represent the dot product, as opposed to the * that represents standard multiplication of floats and the × that represents a cross product of two vectors.

Figure B.11a shows a standard example of a dot product. In this example, the unit vector[18] *b* is pointing along the X axis. *b* has the coordinates [1, 0], and vector *a* has the coordinates [1, 1]. The *a* vector can be thought of as two parts: the part parallel to *b* (the X coordinate of *a*, shown with the thin green line on top of *b*) and the part perpendicular to *b* (the Y coordinate of *a*, shown with the dashed green line). The length of the part of *a* that is parallel to *b* is known as the *projection* of *a* onto *b* and is the result of the dot product *a*•*b*. The dot product takes the whole of a [1, 1], which has a length equal to the square root of 2 (≈1.414) and tells us how much of that vector is parallel to *b*. As mentioned previously, there are two different ways to calculate a dot product, both of which are shown in Figure B.11a and both of which give us the result of 1. This means that the length of vector *a* when projected onto unit vector *b* is 1.

Figure B.11b shows that when two vectors are completely perpendicular, their dot product is zero. So here the projection of *a* onto *b* is zero.

Figure B.11c shows the projection of a longer vector *a* onto *b*. Here again, both versions of the dot product calculation give us the same correct result.

Figure B.11d shows how a dot product can also be used to tell whether an enemy is facing the player character (which can be useful in stealth games). Here, vector *a* is [–3,2] and *b* is [1,0]. The dot product *a*•*b* is –3. If the enemy is looking in the *b* direction, and the dot product of projecting the vector to the player *a* onto unit vector *b* is negative, then that means that the player is behind the enemy. Though all the examples in Figure B.11 show *b* pointing along the X axis, the dot product still works perfectly regardless of the direction in which *b* is pointing, as long as *b* is a unit vector.

You can use dot products in several other places as well, and its use is very common in computer graphics programming (for instance, dot products are used to determine whether a triangle is facing toward a light when calculating diffuse lighting).

Interpolation

An interpolation refers to any mathematical blending between two values. When I was working as a contract programmer after college, I feel that one of the major reasons I got a lot of job offers was because the motion of elements in my graphics code looked smooth and *juicy* (to use Kyle Gabler's term[19]). This was accomplished through the use of various forms of interpolation, easing, and Bézier curves, all of which I present in this section of the appendix.

18. A *unit vector* is a vector with a magnitude of 1 (i.e., a length of 1).

19. "Juice It or Lose It" is a great 2012 talk by Matrin Jonasson and Petri Purho about adding juiciness to games. You can try the link https://www.youtube.com/watch?v=Fy0aCDmgnxg or just search for "juice it or lose it."

Linear Interpolation

A *linear interpolation* is a way of mathematically defining a new value or position by stating that it is in between two existing values. All linear interpolations follow the same formula:

p01 = (1-u) * p0 + u * p1

The result of interpolating the points p0 and p1 is usually called p01. In code, this would look something like Code Listing B.33.

Code Listing B.33 Basic Two-Point Interpolation

```
> Vector3 p0 = new Vector3( 0, 0, 0 );
> Vector3 p1 = new Vector3( 1, 1, 0 );
> float   u  = 0.5f;
>
> Vector3 p01 = (1 -u) * p0 +  u * p1;
> Debug.Log( p01 );  // prints: ( 0.5, 0.5, 0 ), the point half-way between p0 & p1
```

In Code Listing B.33, a new point **p01** is created by interpolating between the points **p0** and **p1**. The value **u** ranges in value between 0 and 1. This can happen with any number of dimensions, though you most often interpolate **Vector2**s and **Vector3**s in Unity.

Time-Based Linear Interpolations

In a time-based linear interpolation, you are guaranteed that the interpolation will complete in a specific amount of time because the **u** value is based on the amount of time that has passed divided by the total desired duration of the interpolation.

Unity Example—Time-Based Linear Interpolation

To create a Unity example, do the following:

1. Create a new Unity project named *Interpolation* using the *3D Core* template.
2. Create a new scene based on the Built-in template and save the scene as *_Scene_Interp*.
3. Create a cube in the hierarchy (*GameObject* > *3D Object* > *Cube*).
   a. Select *Cube* in the Hierarchy pane and attach a TrailRenderer to it (*Component* > *Effects* > *Trail Renderer*).
   b. Open the *Materials* array of the TrailRenderer and set *Element 0* to the built-in material *Default-Particle*. (Click the circle to the right of *Element 0* to see *Default-Particle* in the list of available materials.)
4. Create a new C# script in the Project pane named *Interpolator*. Attach it to *Cube* and then open it in Visual Studio to enter the code in Code Listing B.34.

Code Listing B.34 Interpolator.cs

```
using UnityEngine;

public class Interpolator : MonoBehaviour {
    [Header( "Inscribed" )]
    public Vector3 p0 = new Vector3( 0, 0, 0 );
    public Vector3 p1 = new Vector3( 3, 4, 5 );
    public float   timeDuration = 1;
    [Tooltip( "Set checkToStart to true to start moving." )]
    public bool checkToStart = false;

    [Header( "Dynamic" )]
    public Vector3 p01;
    public bool  moving = false;
    public float timeStart;

    void Update() {
        if ( checkToStart ) {
            checkToStart = false;

            moving = true;
            timeStart = Time.time;
        }

        if ( moving ) {
            float u = ( Time.time - timeStart ) / timeDuration;
            if ( u >= 1 ) {  // If u >= 1, the move ends
                u = 1;
                moving = false;
            }

            // Here is the standard linear interpolation function
            p01 = ( 1 - u ) * p0 + u * p1;

            transform.position = p01;
        }
    }
}
```

5. Switch back to Unity and click *Play*. In the *Cube:Interpolator (Script)* component, check the box next to **checkToStart**, and Cube will move from **p0** to **p1** in 1 second. If you adjust **timeDuration** to another value and then check **checkToStart** again, you can see that Cube always moves from **p0** to **p1** in **timeDuration** seconds. You can also change the location of **p0** or **p1** while Cube is moving, and it will update accordingly.

Linear Interpolations Using Zeno's Paradox

Zeno of Elea (ca. 490–430 BCE) was a Greek philosopher who proposed a set of several paradoxes that had to do with the philosophical impossibility of everyday, common-sense motion.

In Zeno's Dichotomy Paradox, the question is posed of whether a moving object can ever reach a stationary point. Imagine that a frog is hopping toward a wall. Every hop, it covers half of the remaining distance to the wall. No matter how many times the frog hops, it will still have covered only half of the distance remaining to the wall after its last hop, so it will *never* reach the wall.

Ignoring the philosophical implications (and the complete lack of rationality) of this, you can actually use a similar concept along with linear interpolation to create a smooth motion that eases toward a certain point. This is used throughout this book to make cameras that smoothly follow various points of interest (first discussed in Chapter 30, "*Mission Demolition*").

Unity Example—Zeno's Paradox Interpolation

Continuing the same *Interpolation* project from before:

1. Add a sphere to the scene (*GameObject* > *3D Object* > *Sphere*) and move it somewhere away from Cube.
2. Create a new C# script in the Project pane named *ZenosFollower* and attach it to Sphere.
3. Open *ZenosFollower* in Visual Studio and enter the code in Code Listing B.35.

Code Listing B.35 ZenosFollower.cs

```
| using UnityEngine;
|
| public class ZenosFollower : MonoBehaviour {
>     [Header("Inscribed")]
>     public GameObject               poi; // Point Of Interest
>     [Range(0, 1)] public float      u = 0.1f;
|
>     [Header("Dynamic")]
>     public Vector3                  p0;
>     // p0 is on a separate line so that we don't get three Dynamic headers
>     public Vector3                  p1, p01;
|
>     void FixedUpdate () {
>         // Get the position of this and the poi
>         p0 = this.transform.position;
>         p1 = poi.transform.position;
>
>         // Interpolate between the two
```

```
>         p01 = (1-u)*p0 + u*p1;
>
>         // Move this to the new position
>         this.transform.position = p01;
>     }
| }
```

4. Save the code and return to Unity.
5. Set the **poi** of *Sphere:ZenosFollower* to be *Cube* (by dragging *Cube* from the Hierarchy pane into the **poi** slot of the *ZenosFollower (Script)* Inspector on the *Sphere*).
6. Save your scene!

Now, when you click *Play*, the sphere moves toward the cube. If you select the cube and check the **checkToStart** box, the sphere will continue to follow the cube through its motions. You can also move the cube around in the Scene window manually and watch the sphere follow.

Try changing the value of **u** in the *Sphere:ZenosFollower* Inspector. Lower values will make it follow more slowly whereas higher values will make it follow more rapidly. A value of *0.5* would make the sphere cover half of the distance to the cube every frame, which would exactly mimic Zeno's Dichotomy Paradox (but in practice, this follows far too closely). It is true that with this specific code, the sphere will never get to exactly the same location as the cube, and the movement is not very controllable, but this is meant to just be a very quick, simple following script.

FixedUpdate() is used instead of **Update()** in ZenosFollower to make the behavior consistent across all computers. If **Update()** had been used, then depending on the processor load on your computer at any given time, the Sphere would follow closer or further away because more or fewer **Update()** calls would happen each second as the frame rate experienced natural variations. For the same reason, using **Update()** would also cause the sphere to follow much closer on fast machines than on slow ones. **FixedUpdate()** makes behavior consistent across all machines at all times because it is *always* called 50 times per second.[20]

20. **FixedUpdate()** is called 50 times per second because the default value for **Time.fixedDeltaTime** is 0.02 (1/50th of a second), but you can change the frequency at which **FixedUpdate()** is called by adjusting **Time.fixedDeltaTime**. This is especially useful if you've changed **Time.timeScale** to something like 0.1 (slowing down Unity to 1/10th regular speed). At a **Time.timeScale** of 0.1, **FixedUpdate()** would only be called every 0.2 seconds in real time, leading to visibly stuttering physics movement. Any time you alter **Time.timeScale**, you should also alter **Time.fixedDeltaTime** by the same amount, so for a **Time.timeScale** of 0.1, you want a **Time.fixedDeltaTime** of 0.002 to still experience 50 **FixedUpdate()** calls for every real-time second. This makes slow-mo effects look smooth.

Interpolating More Than Just Position

You can interpolate almost any kind of numeric value. In Unity, this means that you can very easily interpolate values like scale, rotation, and color among others.

Unity Example—Interpolating Various Attributes

You can either do this in the same scene as the previous interpolation examples or in a new project:

1. Create a new scene named *_Scene_Interp2* and create two new cubes in its Hierarchy named *c0* and *c1*.
2. Make two new materials (*Assets* > *Create* > *Material*) named *Mat_c0* and *Mat_c1*, each based on the Standard shader.
3. Drag each material onto its respective cube to apply the material.
4. Select *c0* and set its *position*, *rotation*, and *scale* to anything you want (as long as it's visible on screen and the *scale X*, *Y*, and *Z* values are positive). Under the *c0:Mat_c0* section of the Inspector, you should set the *Albedo* color to whatever you want.
5. Do the same for the transform of *c1* and the color of Mat_c1, making sure that c1 and c0 have different positions, rotations, scales, and colors from each other.
6. Add a third cube to the scene, set its position to P:[0, 0, 0], and name it *Cube01*.
7. Create a new C# script named *Interpolator2* and attach it to Cube01. Enter the code in Code Listing B.36 into the Interpolator2 script.

Code Listing B.36 Interpolator2.cs

```
using UnityEngine;

public class Interpolator2 : MonoBehaviour {
    [Header("Inscribed")]
    public Transform      c0;
    public Transform      c1;
    public float          timeDuration = 1;
    [Tooltip( "Click the checkToStart checkbox to start moving" )]
    public bool           checkToStart = false;

    [Header("Dynamic")]
    public Vector3        p01;
    public Color          c01;
    public Quaternion     r01;
    public Vector3        s01;
    public bool           moving = false;
    public float          timeStart;

    private Material      mat, matC0, matC1;

```

```
>     void Awake() {
>         mat = GetComponent<Renderer>().material;
>         matC1 = c1.GetComponent<Renderer>().material;
>         matC0 = c0.GetComponent<Renderer>().material;
>     }
|
|     void Update () {
>         if (checkToStart) {
>             checkToStart = false;
>
>             moving = true;
>             timeStart = Time.time;
>         }
>
>         if (moving) {
>             float u = ( Time.time-timeStart )/timeDuration;
>             if (u>=1) {
>                 u=1;
>                 moving = false;
>             }
>
>             // This is the standard linear interpolation function
>             p01 = (1-u)*c0.position + u*c1.position;
>             c01 = (1-u)*matC0.color + u*matC1.color;
>             s01 = (1-u)*c0.localScale + u*c1.localScale;
>             // Rotations are treated differently because Quaternions are tricky
>             r01 = Quaternion.SlerpUnclamped(c0.rotation, c1.rotation, u);
>
>             // Apply these to this Cube01
>             transform.position = p01;
>             mat.color = c01;
>             transform.localScale = s01;
>             transform.rotation = r01;
>         }
|     }
| }
```

8. Save the script and return to Unity.
9. Drag *c0* from the Hierarchy pane into the **c0** field of the *Cube01:Interpolator2 (Script)* Inspector. Also drag *c1* from the Hierarchy into the **c1** field of the *Interpolator2* script.
10. Click *Play* and then click the **checkToStart** check box in the *Cube01:Interpolator2* Inspector. You'll see that Cube01 now interpolates much more than just position.

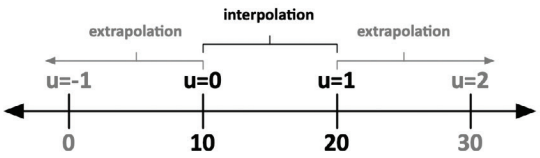
In this code, rotation is handled by the **Quaternion.SlerpUnclamped()** method ("Slerp" is short for the Spherical Linear intERPolation that is used for rotations in Unity). Among other things, we use the built-in Unity method because quaternions involve a lot of complex math (like, literally complex, in that a quaternion comprises three imaginary

numbers and one real number). Rather than independently interpolate each of the X, Y, Z, and W values of the quaternion, a Slerp attempts to choose the most direct path around the surface of a sphere from one rotation to another. The standard **`Quaternion.Slerp()`** method only accepts a **u** value from [0..1], but **`SlerpUnclamped()`** allows *extrapolation*.[21]

Linear Extrapolation

An interpolation blends two points using a **u** value that ranges from 0 to 1. If you allow the **u** value to go beyond this range, you get *extrapolation* (so named because instead of interpolating between two values, it now extrapolates data outside of the original two points).

Given the initial two points on a number line of 10 and 20, an extrapolation of **u**=2 would work as shown in Figure B.12.




```
p01 = (1-u)*10 + u*20
   u = 2
p01 = (1-2)*10 + 2*20
p01 =  (-1)*10 + 2*20
p01 =      -10 + 40
p01 =          30
```

Figure B.12 An example of extrapolation

Unity Example—Linear Extrapolation

To see this in code, make the bolded code additions in Code Listing B.37 to Interpolator2. In addition to extrapolation, this additional code also allows the movement to loop.

Code Listing B.37 Interpolator2.cs

```
| public class Interpolator2 : MonoBehaviour {
|     [Header("Inscribed")]
|     public Transform   c0;
|     public Transform   c1;
>     public float       uMin = 0;
>     public float       uMax = 1;
|     public float       timeDuration = 1;
```

21. If you look at the documentation for Vector3, it also has a **`Lerp()`** method (short for Linear intERPolation) that can interpolate between Vector3s, but I almost never use that function because it clamps the values of u to the range [0..1] and doesn't allow extrapolation. In Unity 5, a **`Vector3.LerpUnclamped()`** method was added that does not clamp to the 0 to 1 range. I do use the unclamped version, but I still think it's important for you to learn how to Lerp on your own, which is why you didn't use **`Vector3.LerpUnclamped()`** in the code for this section.

```
    public bool        loopMove = false; // When true, causes the move to repeat
    [Tooltip( "Click the checkToStart checkbox to start moving" )]
    public bool        checkToStart = false;
    …

    void Update () {
        …
        if (moving) {
            float u = (Time.time-timeStart)/timeDuration;
            if (u>=1) {
                u=1;
                if (loopMove) {
                    timeStart = Time.time;
                } else {
                    moving = false;  // This line is now within the else clause
                }
            }

            // Adjust u to the range from uMin to uMax
            u = (1-u)*uMin + u*uMax;
            // ^ Look familiar? We're using a linear interpolation here too!

            // This is the standard linear interpolation function
            p01 = (1-u)*c0.position + u*c1.position;
            …
        }
    }
}
```

Save the script and return to Unity. Now, if you click *Play* in Unity and then click the **checkToStart** box on Cube01, you'll get the same behavior as you saw when testing Code Listing B.36. However, try changing **uMin** to –1 and **uMax** to 2 in the *Cube01:Interpolator2 (Script)* Inspector. Click **checkToStart**, and you'll see that the color, position, rotation, and scale all extrapolate and go beyond the original range that you set.[22] You can now also check **loopMove** to repeat the interpolation endlessly.

Easing for Linear Interpolations

The interpolations you've been doing so far are pretty nice, but they also have a very mechanical feel to them because they start abruptly, move at a constant rate, and then stop abruptly. Happily, several different *easing* functions can be used to make this movement more interesting. This is most easily explained with a Unity example.

22. You might receive a warning in the console telling you that "BoxColliders does[sic] not support negative scale or size." Don't worry about this. The extrapolation of scale could cause negative scaling here, but we're not worried about collision detection in this example.

Unity Example—Interpolation Easing

Continue the *Interpolation* project by following these steps:

1. Create a new C# script named *Easing* and open it in Visual Studio to add the code in Code Listing B.38. As you do so, note that **`Easing`** *does not* extend **`MonoBehaviour`**.

Code Listing B.38 Easing.cs

```
using UnityEngine;

public class Easing {  // Be sure to remove ": MonoBehaviour" from this line! // a

    public enum eType {                                                    // b
        linear, easeIn, easeOut, easeInOut, sin, sinIn, sinOut
    }

    static public float Ease (float u, eType type, float val=2) {          // c
        float u2 = u;

        switch (type) {
            case eType.linear:  // val is ignored for Easing.eType.linear
                u2 = u;
                break;

            case eType.easeIn:
                u2 = Mathf.Pow(u, val);
                break;

            case eType.easeOut:
                u2 = 1 - Mathf.Pow( 1-u, val );
                break;

            case eType.easeInOut:
                if ( u <= 0.5f ) {
                    u2 = 0.5f * Mathf.Pow( u*2, val );
                } else {
                    u2 = 0.5f + 0.5f * ( 1 - Mathf.Pow( 1-(2*(u-0.5f)), val ) );
                }
                break;

            case eType.sin:
                // Try val values of 0.16f and -0.2f for Easing.eType.sin     // c
                u2 = u + val * Mathf.Sin( 2*Mathf.PI*u );
                break;

            case eType.sinIn:   // val is ignored for SinIn
                u2 = 1 - Mathf.Cos( u * Mathf.PI * 0.5f );
                break;

```

```
>             case eType.sinOut:  // val is ignored for SinOut
>                 u2 = Mathf.Sin( u * Mathf.PI * 0.5f );
>                 break;
>         }
>
>         return( u2 );
>     }
| }
```

a. The **Easing** class is *not* a subclass of **MonoBehaviour**.

b. The **eType** enum is defined within the **Easing** class, so **eType** variables outside of this class would be declared as **Easing.eType** (as shown in Code Listing B.39). Within the **Easing** class, only **eType** is needed.

c. **val** is an optional float parameter of the static **Ease()** function. It is used in various ways as a modifier for many easing types. For example, for **easeIn**, it is used as the power to which **u** is raised; if **val** is 2, then **u2** = u^2, if **val** is 3, then **u2** = u^3. For **sin** easing, **val** is the multiplier for the amplitude of the sine curve that is added to the line (see Figure B.13 for some examples).

This **Easing** class holds all the easing functions for **u**, making it easy to import them into any of your projects. The various easing curves are shown and described in Figure B.13, except for **sinIn** and **sinOut**, which are sine-based, less flexible versions of **easeIn** and **easeOut**.

2. Save the *Easing* script, open the *Interpolator2* script, and make the modifications shown in Code Listing B.39.

Code Listing B.39 Interpolator2.cs

```
| public class Interpolator2 : MonoBehaviour {
|     [Header("Inscribed")]
|     …
|     public bool         loopMove = true; // Causes the move to repeat
>     public Easing.eType easingType = Easing.eType.linear;
>     public float        easingVal = 2;
|     [Tooltip( "Click the checkToStart checkbox to start moving" )]
|     public bool         checkToStart = false;
|     …
|
|     void Update () {
|         …
|         if (moving) {
|             …
|             // Adjust u to the range from uMin to uMax
|             u = (1-u)*uMin + u*uMax;
|             // ^ Look familiar? We're using a linear interpolation here too!
```

```
|
>             // The Easing.Ease function modifies u to change tweak movement
>             u = Easing.Ease(u, easingType, easingVal);
|
|             // This is the standard linear interpolation function
|             p01 = (1-u)*c0.position + u*c1.position;
|             …
|         }
|     }
| }
```

3. Save the *Interpolator2* script in Visual Studio and switch back to Unity.
4. In the *Cube01:Interpolator2 (Script)* Inspector, set **uMin** = *0* and **uMax** = *1*. Also check **loopMove** to set it to **true**.
5. Save your scene!
6. Click *Play* and click **checkToStart**. Now, because **loopMove** is checked as well, Cube01 continuously interpolates between **c0** and **c1**.

Try playing around with the different settings for **easingType**. The **easingVal** value only affects the **easeIn**, **easeOut**, **easeInOut**, and **sin** easing types. For the **sin** type, try an **easingVal** of *0.16* as well as one of *–0.2* to see the flexibility of this sine-based easing type.

In Figure B.13 you can see a graphical representation of various easing curves. In this figure, the horizontal dimension represents the initial **u** value, while the vertical dimension represents the eased **u** value (**u2**). You can see that in every example when **u**=0, **u2** also equals 0 and when **u**=1, **u2** also equals 1. Because of this, if the linear interpolation is time based, the value will *always* move completely from p0 to p1 in the same amount of time regardless of the easing settings.

The *Linear* curve shows no easing (i.e., **u2** = **u**). In each of the other curves shown, the **u2** = **u** line remains as a dashed diagonal line to show normal, linear behavior. If the vertical component of a curve is ever below the dashed diagonal, the movement is lagging behind a linear curve. Conversely, if the vertical component of the curve is ever above the dashed diagonal, the eased curve is ahead of where the linear movement would have been. The slope of the curve represents the speed of the interpolation at that point: A 45° slope is the same speed as the linear interpolation, while a shallower slope is slower and a steeper slope is faster.

The *EaseIn* curve starts slowly and then moves faster toward the end (**u2** = **u***u). This is called "easing in" because the first part of the motion is "easy" and slow before it then speeds up.

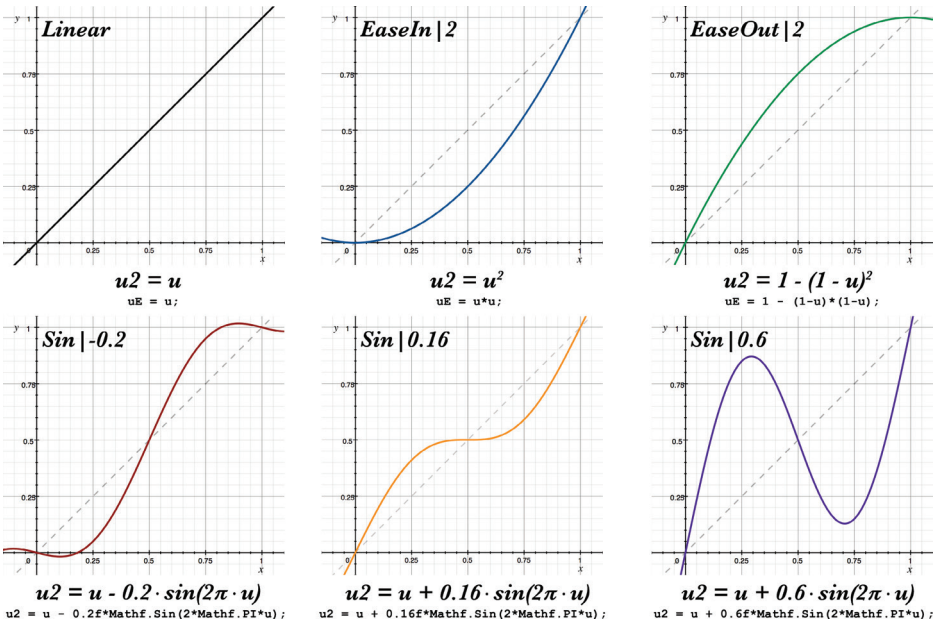


Figure B.13 Various easing curves and their formulae. In each case, the number after the pipe (**|**) represents the **`val`** (or **`easingVal`**) value.

The *EaseOut* curve is the opposite of the *EaseIn* curve. With this curve, the movement starts quickly and then slows at the end. This is commonly known as "easing out."

The three *Sin* curves on the bottom of the diagram all follow the same formula (**`u2 = u + val * sin(u*2π)`**), where **`val`** is a floating-point number (the variable `val` or `easingVal` in the code). The multiplication of **`u*2π`** inside of the **`sin()`** ensures that as **`u`** moves from 0 to 1, it passes through a full sine wave (moving center, up, center, down, and back to center). If **`val`**=0, the sine curve has no effect on the curve (i.e., the curve remains linear). As **`val`** gets further from 0 (either positively or negatively), it has more of an effect.

The curve *Sin|–0.2* is an ease-in-out with a bounce. A **`val`** of *–0.2* adds a negative sine wave to the linear progression, causing a moving object to back up a bit from p0, move quickly toward p1, overshoot a bit, and then settle at p1. A **`val`** closer to zero (e.g., *Sin|–0.1*) would cause the object to ease-in to full speed at the center point and then slow as it approached p1 without the extrapolation bounce at either end.

In the curve *Sin|0.16,* a slight sine curve is added to the linear **`u`** progression, causing the curve to get ahead of linear, stop briefly in the middle, and then hurry to catch up at the end. If you're moving an object, this brings it to the center point, slows it in the middle to "feature" it for a while, and then moves it out.

The curve *Sin|0.6* is the easing curve that is used by Enemy_2 in Chapter 32, "*Space SHMUP* — Part 2." In this case, enough positive sine wave has been added to cause the object to shoot past the center point to a point about 80% of the way to p1, then move back to a point 20% of the way to p1, and then finally move to p1.

Note that the easing functions only exhibit the behavior we want for a **u** value in the range [0..1] and wouldn't handle extrapolation of **u** values well.

You can see some more examples of easing functions demonstrated with animations at http://easings.net, though their names for the functions and implementations differ from mine (I didn't know standard functions already existed when I created mine).

Bézier Curves

I use Bézier curves enough in my work that—even though they are a form of interpolation—they deserve their own section here. A Bézier curve is a linear interpolation between *more than two points*. Just as with a normal linear interpolation, the base formula is p01 = (1-u) * p0 + u * p1. The Bézier curve just adds more points and more calculations. But Bézier curves are one of the wonderful cases where math can lead to something simply beautiful. To gain an idea of what I'm talking about, I highly recommend watching the award-winning video "The Beauty of Bézier Curves" by Freya Hölmer.[23]

In any Bézier curve, the interpolated point will start at the first point and end at the last point but never touches the points in between. Because of this, the first and last points are often referred to as "end points," while the points in between are called "control points."

Three-Point and Four-Point Bézier Curves

Given three points: p0, p1, and p2

```
p01 = (1-u) * p0  +  u * p1
p12 = (1-u) * p1  +  u * p2
p012 = (1-u) * p01  +  u * p12
```

As is demonstrated in the preceding equations, for the three points p0, p1, and p2, the location of the point of the Bézier curve is calculated by first performing a linear interpolation between p0 and p1 (the resulting point is called p01), then performing

23. https://youtu.be/aVwxzDHniEw or search "the beauty of bezier curves" on YouTube. Freya Hölmer is a brilliant Unity developer, and I recommend watching any of her videos! – accessed March 27, 2022.

a linear interpolation between p1 and p2 (called p12), and finally performing a linear interpolation between p01 and p12 to obtain the final point, p012. A graphical representation of this is shown in Figure B.14.

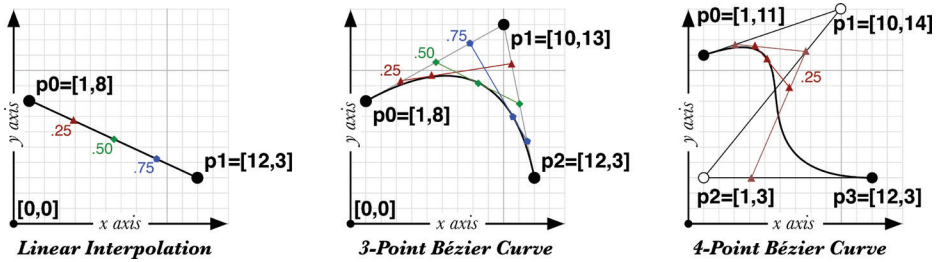


Figure B.14 A linear interpolation and three-point and four-point Bézier curves

A four-point curve requires more calculations to accommodate all four points:

```
p01 = (1-u) * p0  +  u * p1
p12 = (1-u) * p1  +  u * p2
p23 = (1-u) * p2  +  u * p3
p012 = (1-u) * p01  +  u * p12
p123 = (1-u) * p12  +  u * p23
p0123 = (1-u) * p012  +  u * p123
```

Four-point Bézier curves are used in many drawing programs as a way of defining very controllable curves, including Adobe Flash,[24] Illustrator, and Photoshop; the Affinity suite (Designer, Photo, and Publisher); The Omni Group's OmniGraffle; and many others. In fact, the curve editor used in Unity for animation and audio processing uses a form of four-point Bézier curves.

Unity Example—Bézier Curves

Follow these steps to create a Bézier curve example in Unity. When writing code, I don't use the accented *é* in Bézier because code is usually written with only the original ASCII characters (which lack accents).

1. Create a new scene in your Unity project and save it as *_Scene_Bezier*.

24. This is a complete aside, but, hey, thanks for reading the footnotes! I deeply lament the loss of both Macromedia Director (and its web plugin, Shockwave) and Macromedia Flash. Each of them was an amazing tool for non-programmers to make interactive content. Director enabled the CD-ROM era of games, and of course, Flash was responsible for most of the interesting web games and animation throughout the 2000s. Yes, of course, there were security issues with how the plugins worked, but I have not found anything since that allows creative designers to so quickly mock up a game idea. Plus, art creation in Flash was just *fun*!

2. Add four cubes to the Hierarchy named *c0*, *c1*, *c2*, and *c3*.
   a. Set the *transform.scale* of all cubes to S:[**0.5**, **0.5**, **0.5**].
   b. Position the cubes in various positions around the scene and adjust your Scene view so that you can see all of them.
3. Add a sphere to the scene.
   a. Attach a *TrailRenderer* to the sphere.
   b. Open the *Materials* array of the TrailRenderer and set *Element 0* to the built-in material *Default-Particle*.
4. Create a new C# script named *Bezier* and attach it to Sphere. Open *Bezier* in Visual Studio and enter the code in Code Listing B.40 to demonstrate a four-point Bézier curve in Unity.

Code Listing B.40 Bezier.cs

```
using UnityEngine;

public class Bezier : MonoBehaviour {
    [Header("Inscribed")]
    public float        timeDuration = 1;
    public Transform    c0, c1, c2, c3;
    [Tooltip( "Click the checkToStart checkbox to start moving" )]
    public bool         checkToStart = false;

    [Header("Dynamic")]
    public float        u;
    public Vector3      p0123;
    public bool         moving = false;
    public float        timeStart;

    void Update () {
        if (checkToStart) {
            checkToStart = false;
            moving = true;
            timeStart = Time.time;
        }

        if (moving) {
            u = (Time.time-timeStart)/timeDuration;
            if (u>=1) {
                u=1;
                moving = false;
            }

            // 4-point Bezier curve calculation
            Vector3 p01, p12, p23, p012, p123;

```

```
>             p01 = (1-u)*c0.position + u*c1.position;
>             p12 = (1-u)*c1.position + u*c2.position;
>             p23 = (1-u)*c2.position + u*c3.position;
>
>             p012 = (1-u)*p01 + u*p12;
>             p123 = (1-u)*p12 + u*p23;
>
>             p0123 = (1-u)*p012 + u*p123;
>
>             transform.position = p0123;
>         }
|     }
| }
```

5. Save the *Bezier* script and return to Unity.
6. Assign each of the *four cubes* to their *respective fields* in the *Sphere:Bezier (Script)* Inspector.
7. Click *Play* and then click the **`checkToStart`** check box in the Inspector.

Sphere will trace a Bézier curve between the four cubes. It's important to note here that Sphere only ever touches c0 and c3. It is influenced by but does not touch c1 and c2. This is true of all Bézier curves. The ends of the curve always touch the first and last points, but no points in-between are ever touched. If you're interested in looking into a kind of curve where the midpoints *are* touched, look up "Hermite spline" online (as well as other kinds of splines).

A Recursive Bézier Curve Function

As you saw in the previous section, the additional calculations for adding more control points to a Bézier curve are pretty straightforward conceptually, but it takes a while to type all the additional lines of code. The upcoming code listing uses a recursive function to handle any number of points without any additional code. It's a little conceptually complex, so let's start by considering how it should work.

To interpolate a standard three-point Bézier curve, you start with three points: [p0, p1, p2]. However, to interpolate them, you need to first break it down into two smaller interpolations [p0, p1] and [p1, p2]. You interpolate each of these and return the interpolated points p01 and p12. Finally, you interpolate between p01 and p12 to get the final point p012.

The **`BezierR()`** (**`R`** for recursive) function will do the same thing, recursively breaking the problem down to smaller and smaller lists of points until each branch gets to a list of just one point and then returning those points up the recursion chain, interpolating on the way up.

> **note**
>
> In the past, I used this recursive version of the Bézier curve function in many projects, but since I've learned about Data-Oriented Design, I've found a non-recursive way to calculate any number of points that is even more efficient, so I've added a data-oriented version of the function after this section.
>
> However, this is still an excellent example of using a recursive function to solve a problem, and it offers you a chance to see an example of transitioning a function from OOP to DOD, so I'm keeping it in this edition.

In the first edition of this book, the recursive Bézier function created a new **`List<Vector3>`** with each recursion, but that was very inefficient because it wasted both memory and processing power with each creation of a new List. In fact, interpolating a four-point Bézier curve with the first-edition version of the recursive function would create *fourteen* additional lists. (Please refer to Figure B.15 and Code Listing B.41 as you read the next few paragraphs.)

Rather than make a ton of new Lists, the second edition version of the recursive function (shown in Code Listing B.41) just passes the same List into each of its recursions along with two integers—**`iL`** and **`iR`**—as you can see in the **`BezierR()`** function declaration.

```
static public Vector3 BezierR(float u, List<Vector3> pts, int iL=0, int iR=-1) {…}
```

The integers **`iL`** and **`iR`** are each an *index* into the List **`pts`**, meaning that **`iL`** and **`iR`** are references to elements of **`pts`**. If **`iL`** is 0, then it's pointing to the 0^{th} element of **`pts`**. If **`iR`** is 3, then it's pointing to the third element of **`pts`**. **`iL`** and **`iR`** are optional parameters. If neither are passed in (i.e., **`BezierR()`** is called with just **`u`** and **`pts`** arguments), then **`iL`** will be 0, and **`iR`** will start as –1 but will thereafter be given the index of the final element of **`pts`**.

`iL` represents the leftmost element of **`pts`** that is being considered by any recursion of **`BezierR()`**, and **`iR`** represents the rightmost element. So, for a four-point List **`pts`**, **`iL`** will start at 0, and **`iR`** will start at 3 (the index of the last point in **`pts`**). Each time the **`BezierR()`** function recurs, it branches and sends a range of fewer points to the next level. Rather than create new Lists, the new version of **`BezierR()`** adjusts **`iL`** and **`iR`** to look at a smaller section of the overall List **`pts`**. Eventually, a terminal case is reached in each branch where **`iL`** and **`iR`** both point to the same element of **`pts`**, and then the value of that element is returned up the chain as a Vector3, and the series of actual interpolations takes place as the chain of recursion unwinds.

Figure B.15 shows the series of recursive calls that are made by a single four-point call to the **`BezierR()`** function. The green arrows trace the calls, and the red dashed arrows show the returns. You can see that either **`iL`** or **`iR`** is respectively incremented or decremented with each call, and the elements of **`pts`** that are black in the diagram are those that are in the range from **`iL`** to **`iR`** for each call to **`BezierR()`**.

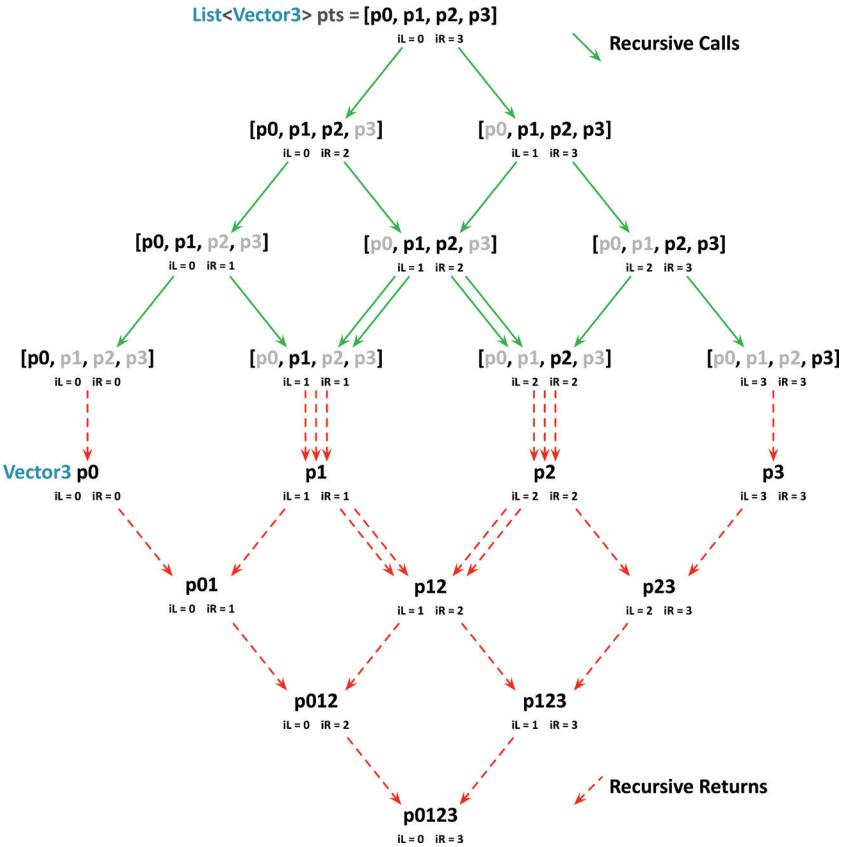


Figure B.15 The path of calls (solid green arrow) and returns (dashed red arrow) when recursively interpolating a four-point Bezier curve

Unfortunately, this method of forking recursion will calculate the value of P12 twice (it has two green arrows pointing to the [p1, p2] interpolation and two red arrows returning the p12 value). This is an inefficiency in this version of solving a Bézier curve that is avoided by the data-oriented **`BezierArray()`** method that is discussed in the next section.

Code Listing B.41 is an excerpt from the Utils.cs class that you import in some of the later projects of Part III. It shows a recursive **`BezierR()`** function to calculate a Bézier curve for any number of points.

Code Listing B.41 Utils.cs

```
using System.Collections;
using System.Collections.Generic;    // Needed for the List<>s in these functions
using UnityEngine;

public class Utils {
    …  // There are many lines in Utils prior to the BezierR methods

//============================ Bézier Curves ============================          // a
    /// <summary>
    /// While most Bézier curves are 3 or 4 points, it is possible to have
    ///   any number of points using this recursive function.
    /// LerpUnclamped is used to allow extrapolation.
    /// </summary>
    /// <param name="u">The amount of interpolation [0..1]</param>
    /// <param name="pts">A List of Vector3 points to interpolate</param>
    /// <param name="iL">Index of the left extent of the used elements of pts.
    ///   Defaults to 0.</param>
    /// <param name="iR">Index of the right extent of the used elements of pts.
    ///   Defaults to -1, which is changed to the last element's index.</param>
    static public Vector3 BezierR(float u,List<Vector3> pts,int iL=0,int iR=-1){
        // If iR is the default -1 value, set iR to the last element in pts        // b
        if (iR == -1) iR = pts.Count-1;                                            // c

        // If we are only looking at one element of pts, return it                 // d
        if (iL == iR) return( pts[iL] );

        // Call BezierR again with all but the leftmost used element of pts
        Vector3 leftVal = BezierR(u, pts, iL, iR-1);                               // e
        // And call BezierR again with all but the rightmost used element of pts
        Vector3 rightVal = BezierR(u, pts, iL+1, iR);                              // f

        // The result is the Lerp of these two recursive calls to BezierR
        Vector3 res = Vector3.LerpUnclamped( leftVal, rightVal, u );               // g
        return( res );
    }

    // This version allows an Array or a series of Vector3 arguments as input
    static public Vector3 BezierR( float u, params Vector3[] arr ) {               // h
        return( BezierR( u, new List<Vector3>( arr ) ) );
    }

    …
    // Utils includes many versions of the BezierR function for other data types
}
```

a. I tend to use large, obvious headings like this throughout the Utils script because it is so large. On the following line, you can see XML Documentation for the BezierR function (the `///` comments with tags like `<summary>` and `<param>`). Visual Studio can read these and give you this information if you mouse over the name of the function anywhere in code. See "XML Documentation" in this appendix for more information.

b. The **`BezierR()`** function takes as input a float **`u`** and a **`List<Vector3> pts`** of points to interpolate. It also has two optional parameters, **`iL`** and **`iR`**, which represent the indices in the **`pts`** List of the leftmost (**`iL`**) and rightmost (**`iR`**) element being considered by this recursion of **`Bezier()`**. See Figure B.15 for more information.

c. If the **`iR`** value passed in is –1 (or if no value is passed in), **`iR`** is set to the index of the last element in **`pts`**.

d. As mentioned in Chapter 24, "Functions and Parameters," the *terminal case* of a recursive function is the one that stops the recursive calling of the function and begins returning values up the chain. The terminal case of **`BezierR()`** is reached when **`iL`** == **`iR`**. If **`iL`** == **`iR`**, then both the left and right indices into the List **`pts`** are pointing at the same Vector3 element. When this occurs, the Vector3 to which they both point is returned.

e. This is one of the two recursive calls to **`BezierR()`**. Here, the **`iR`** index is decremented by 1. So, if this initial call to **`BezierR()`** is considering elements 0–3 of **`pts`**, the recursion called here will only consider elements 0–2. This allows us to consider fewer elements of **`pts`** on the next recursion without creating a new List.

f. This is the other recursive call to **`BezierR()`**. Here **`iL`** index into **`pts`** is incremented by 1. This has the effect of passing all but the first element of **`pts`** into the next recursion.

g. The results of the recursive calls on lines **`// e`** and **`// f`** have been stored in two Vector3s: **`leftVal`** and **`rightVal`**. These two Vector3s are interpolated by **`Vector3.LerpUnclamped()`**, and the result is returned up the recursion chain.

h. As covered in Chapter 24, "Functions and Parameters," the **`params`** keyword allows the **`arr`** array parameter to accept either a Vector3 array or a series of individual Vector3 parameters separated by commas (after the first float argument). This is used on line **`// d`** of Code Listing B.42.

Code Listing B.42 shows a test class that uses the params version of **BezierR()** in two different ways.

Code Listing B.42 BezierRTester.cs Test Class for **Utils.BezierR()**

```
using UnityEngine;

public class BezierRTester : MonoBehaviour {
    public Vector3 p0 = new Vector3( 1, 0, 0 );
    public Vector3 p1 = new Vector3( 1, 1, 0 );
    public Vector3 p2 = new Vector3( 0, 1, 0 );
    public Vector3 p3 = new Vector3( 0, 0, 0 );

    void Start() {
        System.Text.StringBuilder sb = new System.Text.StringBuilder();       // a
        Vector3[] points = new Vector3[] { p0, p1, p2, p3 };
        Vector3 p0123;
        for ( float u = 0; u <= 1; u += 0.1f ) {
            sb.Append( "As array: " );
            p0123 = Utils.BezierR( u, points );                               // b
            sb.Append( p0123.ToString() );
            sb.Append( "\tAs floats: " );                                     // c
            p0123 = Utils.BezierR( u, p0, p1, p2, p3 );                       // d
            sb.AppendLine( p0123.ToString() );                                // e
        }
        Debug.Log( sb.ToString() );
    }
}
```

a. For information on the StringBuilder, see the "System.Text.StringBuilder" section of this appendix.

b. Here, the array **points** is passed into the array **arr** parameter of **Utils.BezierR()**, which is nothing unexpected.

c. The **\t** is a tab.

d. The **params** keyword of the **BezierR()** overload definition on line **// h** of Code Listing B.41 allows the series of **Vector3** parameters (i.e., **p0, p1, p2, p3**) to automatically be converted into a **Vector3[]** array and assigned to the **arr** parameter of that **BezierR()** function.

e. The output from the **StringBuilder sb** shows that both versions of calls to **BezierR()** result in the same values:

```
As array: (1.0, 0.0, 0.0)  As floats: (1.0, 0.0, 0.0)
As array: (1.0, 0.3, 0.0)  As floats: (1.0, 0.3, 0.0)
As array: (0.9, 0.5, 0.0)  As floats: (0.9, 0.5, 0.0)
…                                                    // and so on
```

Lines `// b` and `// d` will both call the overload of **`BezierR()`** with the **`arr`** array (declared on line `// h` of Code Listing B.41). Then on the single line of that overload, the **`arr`** array is converted to a **`List<Vector3>`**, and when **`BezierR()`** is called with a **`List<Vector3>`** as the second argument, it calls the original version of **`BezierR()`** declared on line `// b` of Code Listing B.41.

A Data-Oriented Bézier Function

Reusing the same List rather than creating new Lists with every recursion significantly improved the Bézier function in Utils.cs from the first to second edition of the book, but as I started learning about Data-Oriented Design, it occurred to me that the **`BezierR()`** method was wasting a lot of time calling itself recursively, including calculating some values twice (like the p12 example explained after Figure B.15).

The version that I now use of the **`Bezier()`** function for any number of points uses a two-dimensional array to hold all the points and then combines them until they are a single value. Figure B.16 shows how the array approach works (with four points on the left and a Bézier interpolation of ints on the right):

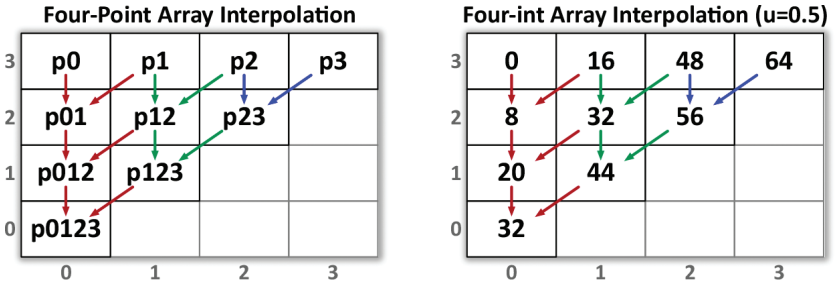


Figure B.16 Two visualizations of how a 2D array will be used to interpolate a Bézier curve

As you can see in the figure, when four points are passed into the array version of a Bézier solver, the following steps happen:

A. All four points are initially placed in the 3rd row of the array.

B. Each pair of points on the 3rd row is interpolated into positions 0–2 on the 2nd row.

C. Each pair of points on the 2nd row is interpolated into positions 0–1 on the 1st row.

D. The remaining pair of points on the 1st row is interpolated into the [0, 0] cell.

The right half of Figure B.16 shows this process on some simple integers. The pairs of values on the 3rd row are all interpolated using a **u** value of 0.5, with the results placed in the 2nd row. The process continues until the final interpolated value is placed in cell [0, 0].

Code Listing B.43 shows how this can be done in code using **`params`** to handle any number of **`Vector3`** points. There is also an overload that handles Lists.

Code Listing B.43 **`BezierArray()`** Code from Utils.cs

```
| public class Utils {
|     …
|
| //============================ Bézier Curves ============================
|     …
|
>     /// <summary>
>     /// This two-dimensional array version of a Bézier curve solver is the most
>     ///   performant I've developed so far that can handle any number of points.
>     /// LerpUnclamped is used to allow for extrapolation.
>     /// </summary>
>     /// <param name="u">The amount of interpolation [0..1]</param>
>     /// <param name="arr">A params array of points to interpolate</param>
>     /// <returns>The interpolated value</returns>
>     static public Vector3 BezierArray( float u, params Vector3[] arr ) {      // a
>         int numPoints = arr.Length;
>         Vector3[,] arr2D = new Vector3[numPoints, numPoints];                 // b
>         // Put the initial points in the last row of arr2D
>         for ( int i = 0; i < numPoints; i++ ) {                               // c
>             arr2D[numPoints - 1, i] = arr[i];
>         }
>
>         // Loop over the rows
>         Vector3 vecL, vecR;
>         for ( int row = numPoints - 1; row > 0; row-- ) {                     // d
>             for ( int col = 0; col < row; col++ ) {                           // e
>                 vecL = arr2D[row, col];
>                 vecR = arr2D[row, col + 1];
>                 arr2D[row - 1, col] = (1-u)*vecL + u*vecR;                    // f
>             }
>         }
>
>         return arr2D[0, 0];                                                   // g
>     }
|
|     …
| }
```

a. The **params** keyword allows the series of **Vector3** parameters (i.e., **p0**, **p1**, **p2**, and **p3**) to automatically be converted into a **Vector3[]** array and assigned to **arr**.

b. A two-dimensional array **arr2D** is created that is large enough to hold everything. In the example of a four-point Bézier curve, a 4x4 array is created.

c. Each of the elements are copied from **arr** into the last (i.e., 3rd) row of **arr2D**.

d. A **row** value is created that is initialized to **numPoints-1**, the number of the last row, to which we just copied all the points from **arr**. The iterator clause of the **for** loop will decrease row, running a loop for **row=3**, **2**, and **1** (but not **0**).

e. In a second **for** loop, a **col** value is initialized to 0 and counts up, running a loop for **col=0**, **1**, and **2** (but not **3**).

f. Pairs of points from the current row are interpolated, and the result is placed into the next row down. For example, when the points at [3, 0] and [3, 1] are interpolated, the result is placed in [2, 0].

g. When the nested **for** loops complete, the value at [0, 0] is the final interpolated point that **BezierArray()** should return.

This array version has several advantages over the recursive version of Bézier solving:

- Only one function is ever called, so we avoid the overhead of multiple function calls.
- High data locality! Unity's Vector3 is a struct, so its x, y, and z values are all stored in the same place. As a result, the values of the Vector3s in **arr2D** are very closely packed in memory, speeding up access times. (To read more about data locality, see Chapter 28, "Data-Oriented Design.")
- Each value is only calculated once, as opposed to the p12 example shown in Figure B.15 where the recursive calls repeat calculations.
- It's much more straightforward to explain. The diagram in Figure B.16 is probably much easier to understand than the one for **BezierR()** in Figure B.15.

A version of the array-based Bézier function is included in the ProtoTools **Utils** class that is part of the initial unitypackage for the *Prospector* and *Dungeon Delver* projects, though in that package, it is simply called **Bezier()** rather than **BezierArray()**, because I want it to be clearly seen as the default version of the function.

The Utils C# script includes several overloads of the **Bezier()** function for different types (e.g., **Vector3**, **Vector2**, **float**, and **Quaternion**). It also includes more overloads that use the **params** keyword to allow any number of points to be passed in as arguments to the **Bezier()** function as well as a **List** version.

Several good implementations of Bézier curves and Splines (a way of making curves that connect into a long, smooth path) are available in the Unity Asset Store.

YOUR CODE CAN ALWAYS BE BETTER, AND THAT'S OKAY!

One of the big takeaways that I want you to get from these Bézier function examples is that code can *always* be improved and refactored to be more efficient or readable (though at the extreme end, those two goals are often mutually exclusive; assembly language can be extremely efficient, but it's not easy to read, even for most programmers). In retrospect, the array-based version of the Bézier functions seems much more straightforward and obvious, but that's not the order in which I came up with them. (And, in fact, I could create a version of the **`Bezier()`** function that used a single-dimensional array, which would be even more efficient than the two-dimensional array version.)

As you continue to develop and refactor your projects, don't get hung up on the inefficiencies or deficiencies of your old code, and don't worry about throwing out old code or old work. Keep pushing forward and keep improving your code progressively; that's how you become a better programmer.

Pen-and-Paper Roleplaying Games

Many good pen-and-paper roleplaying games (RPGs) are out there. The most popular is still probably *Dungeons & Dragons* by Wizards of the Coast (*D&D*), which is now in its fifth edition. Since the third edition, *D&D* has been based on the *d20* system, which uses a single twenty-sided die in place of the many complex rolls of myriad dice that were common in prior systems. I like *D&D* for a lot of things, but I have found that my students often get bogged down in combat when attempting to run *D&D* as their first system; it has a lot of very specific combat rules (especially in the fourth edition). Even in experienced roleplaying groups, the beginning of any *D&D* combat usually marks the end of any story progression for the rest of the game session. After the single battle is over, there's no more time in the session.

My personal recommendation for a first RPG system is *FATE* by Evil Hat Productions, especially the streamlined *FATE Accelerated* (FAE) system. *FAE* is a simple system that allows players to contribute directly to the narrative much more than other systems allow. (Most other systems give the *game master* running the game exclusive power over any events that happen.) You can learn about the core version of *FATE* at the website http://faterpg.com , and you can read the free *FATE* system reference document (SRD) at http://fate-srd.com . For info on *FATE* Accelerated, and to get a pay-what-you-wish 50-page eBook with all the info you need to get started, check out http://www.evilhat.com/home/fae/ .[25]

25. You can also directly download the 50-page FAE eBook from http://www.evilhat.com/home/wp-content/uploads/FAE.zip . All these FATE-related links were accessed August 27, 2021.

Tips for Running a Good Roleplaying Campaign

Running a roleplaying campaign can do wonders for your abilities as both a game designer and storyteller. Here are some tips that I've found to be very useful when my students start running campaigns:

- **Start simple:** A lot of different roleplaying systems are out there, and they vary greatly in the complexity of their rules. As described in the preceding section, I recommend starting with a simple system like the *FATE Accelerated* system by Evil Hat Productions. After you've played a few games with that system, you can move on to more complex systems like *D&D* or *Pathfinder*. The fifth edition of *D&D* has a relatively straightforward core rulebook with many supplemental rulebooks to add as you get deeper into the system.
- **Start short:** Rather than starting with the first episode of a campaign that you expect to take a full year of play to complete, try starting with a simple mission that can be wrapped up in a single night of play. This gives your group a chance to try out their characters and the system and see if they like both. If not, it's easy to change to something else, and it's much more important that the players enjoy their first experience roleplaying than that you kick off an epic campaign.
- **Help the players get started:** If the players in your campaign have little or no prior experience roleplaying, creating their characters for them is a very good idea. This gives you the chance to make sure that the characters have complementary abilities and stats so that they'll combine into a good team. A standard roleplaying party is composed of the following characters (these examples are from a fantasy world, but they can be adapted to anywhere; in a future setting, the wizard would be a hacker):
  - A **warrior** to absorb enemy damage and fight up close (a.k.a., a **tank**)
  - A **wizard** to do long-range damage and detect magic (a.k.a., a **glass cannon**)
  - A **thief** to disarm traps and make powerful sneak attacks (a.k.a., a **blaster**)
  - A **cleric** to detect evil and heal the other party members (a.k.a., a **controller**)

  If you're going to create characters for your players, you should get early buy-in from them by asking them to tell you about the kind of play experience they would like and the kind of abilities that they want their character to have. Early buy-in and interest is one of the keys to getting your players past the rough patches that can happen at the beginning of a campaign.
- **Plan for improvisation:** Your players will frequently do things that you don't expect. The only way to plan for this is to prepare yourself for flexibility and improvisation. Be ready with things like maps of generic spaces, a list of names that could be used for NPCs (non-player characters) that the party may or may not encounter, and a few generic enemies or monsters of various difficulties that you can conjure at will. The more you have ready beforehand, the less time you'll have to spend looking through your rulebook in the middle of the game.

- **Be willing to make rulings:** If you can't find the answer to a question in the rules after five minutes of looking, just make a ruling using your best judgment and agree with the players that you'll look it up after the game session is over. This keeps the game from bogging down due to esoteric rules.
- **It's also the players' story:** Remember to allow the players to go off the beaten path. If you've prepared too narrow a scenario, you might be tempted to not let them do so, but that would run the risk of killing their enjoyment of the game.
- **Remember that constant optimal challenge isn't fun:** In the discussion of *flow* in Chapter 8, "Design Goals," you read that if players are always optimally challenged, they get exhausted quickly. This is also true in RPGs. Boss fights should always optimally challenge your players, but you should also have fights where the players win easily (this helps demonstrate to them that their characters are actually getting stronger as they level up) and sometimes even fights that the players need to flee from to survive (this is usually not expected by players, and can be really dramatic for them). Unlike most systems, FAE has a really intriguing game mechanic that makes giving up and fleeing a much better choice than fighting to lose, which is another reason I really like it.

If you keep these tips in mind, it should help your roleplaying campaigns be a lot more fun for both you and your players.

User Interface Concepts

This section covers how to use various (e.g., Microsoft, Sony, Bluetooth) gamepad controllers on your Windows, macOS, or Linux machine and information about how to enable right-click on macOS computers.

Complex Game Controller Input

In general, the Unity Input Manager's defaults handle basic input like the left analog stick and basic button presses of a single controller pretty well, but if you want to use more complex input on a modern controller or if you want to create a game that manages input from several players, I strongly recommend that you look to something other than the built-in Unity Input Manager that you've seen throughout this book. Here are some of the ones that I've seen give people the best results:

- **InControl by Gallant Games**: http://www.gallantgames.com

  InControl is my personal favorite input manager. It does a great job of mapping dozens of controllers to some basic inputs (left stick, right stick, D-pad, two bumpers, two triggers, and four action buttons) that are generalized across all the controllers. If you want controllers to "just work," this to me seems like your best bet. You can find it by searching "InControl" on the Unity Asset Store.

- **Rewired by Guavaman Enterprises**: https://guavaman.com/projects/rewired/

  Rewired is the input manager that I've seen used most by other developers. It was even used by Will Winn Games to manage up to 12 simultaneous local controllers for their fantastic pirate game *Plunder Panic*. Rewired was also used by Digital Mania to develop their four-player brawler *Warshmallows*, which you can see featured in Figure 10.2 of Chapter 10, "Game Testing." Rewired requires more setup than InControl, but as a result, I think it's more flexible. You can find it by searching "Rewired" on the Unity Asset Store.

  Additionally, to prep for my work with Digital Mania, I created a controller watcher that works with Rewired to show the status of all four controllers on screen as uGUI elements. This allows you to see what was happening with controller input in any screen recordings you do for playtesting. Check the book website http://book.prototools.net for a link to download this tool.

- **Unity Input System**: https://docs.unity3d.com/Manual/com.unity.inputsystem.html

  Unity has recently developed a much more powerful and robust Input System package to provide more functionality than the traditional Input Manager. I did not use it in this book because there is considerable setup time to get basic input working. However, the new Input System is much more reliable across multiple platforms and controller types, so I expect to use it in the future.

Although configuring the old Unity Input Manager to handle multiple controllers of different types on various game platforms is possible, it's a massive time commitment to do so, and I highly recommend paying the $40 for either InControl or Rewired to make that hassle their problem instead of yours.

Input Manager Mapping for Various Controllers

Although most of the games included in this book use a mouse or keyboard interface, I'm guessing you might want to eventually hook up a gamepad controller to your games. Unfortunately, this is more complex than you would expect. In general, if you can connect a Bluetooth gamepad controller to your computer, then you will be able to get "Horizontal" and "Vertical" axis input from the left analog stick, but the rest is unfortunately different for various controllers on various operating systems, with each type of controller (PlayStation, Xbox, Switch, etc.) appearing differently to the operating system, and each operating system also interpreting them differently. This is the reason that Rewired and InControl are so popular as well as the reason that Unity has developed their new Input System.

Previous editions of this book included the mappings for a wired Xbox 360 controller when plugged into macOS, Windows, and Linux (as examples of the differences, Joystick Axis 5 is up and down on the right analog stick on Linux and Windows but it is the left trigger on macOS; and Axis 6 is the right trigger on macOS and Linux but it is

left and right on the D-pad on Windows). Additionally, recent changes to macOS have invalidated most of the drivers for Xbox 360 controllers that I can find online. Rather than get your hopes up by including that diagram in this edition, I'm just going to recommend again that you look to InControl, Rewired, or Unity's new Input System if you want to develop a game that will work with various gamepad controllers across multiple operating systems.

Right-Click on macOS

Throughout this book, I often ask you to right-click on something. However, many people don't know how to right-click on a Macintosh because it's not the default setting for macOS trackpads and mice. There are actually several ways to right-click, and the one you use depends on how new your Mac is and how you prefer to interact with your machine.

Control-Click = Right-Click

Near the bottom-left corner of all modern macOS keyboards is a *control* key. If you hold down the *control* key and then left-click (your normal click) on anything, macOS treats it as a right-click.

Use Any PC Mouse

You can use almost any PC mouse that has two or three buttons on macOS. I personally use a Logitech *MX Anywhere 2* or a Razer *Orochi*.

Set Your macOS Mouse to Right-Click

If you have a macOS mouse made in 2005 or later (e.g., the Apple Mighty Mouse or Apple Magic Mouse), you can enable right-click by following these steps:

1. Open *System Preferences* from the Apple menu in the top-left corner of your screen. Then click the *Mouse* settings icon.
2. Select the *Point & Click* tab at the top of the screen.
3. Check the box next to *Secondary click*.
4. Choose *Click on right side* from the pop-up menu directly below *Secondary Click*.

This makes a click on the left side of the mouse left-click and a click on the right side right-click.

Set Your macOS Trackpad to Right-Click

As with the Apple Mouse, you can configure any Apple laptop trackpad (or the Bluetooth Magic Trackpad) to right-click.

1. Open *System Preferences* from the Apple menu in the top-left corner of your screen. Then click the *Trackpad* settings icon.
2. Choose the *Point & Click* tab at the top of the window.
3. Check the box next to *Secondary Click*.
4. If you choose *Click or tap with two fingers* from the pop-up menu directly below *Secondary Click*, it will make tapping with one finger the standard left-click and tapping with two fingers the right-click. Other right-click trackpad options are also available.