C# GAME PROGRAMMING COOKBOOK for UNITY 3D

Jeff W. Murray

CRC Press
Taylor & Francis Group
AN A K PETERS BOOK

C# GAME PROGRAMMING COOKBOOK for UNITY 3D

C# GAME PROGRAMMING COOKBOOK for UNITY 3D

Jeff W. Murray

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business
AN A K PETERS BOOK

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

© 2014 by Taylor & Francis Group, LLC
CRC Press is an imprint of Taylor & Francis Group, an Informa business

No claim to original U.S. Government works
Version Date: 20140213

International Standard Book Number-13: 978-1-4665-8142-5 (eBook - PDF)

This book is dedicated to my boys,
Ethan and William.
Be nice to the cat.

Contents

Acknowledgments

I would like to thank my wife for all the encouragement, support, and nice cups of tea. I would also like to thank my mum and dad, my brother Steve, and everyone else who knows me. Sophie cat, be nice to the boys.

Sincere thanks go to the many people who positively influence my life directly or indirectly: Michelle Ashton, Brian Robbins, George Bray, Nadeem Rasool, Christian Boutin, James and Anna, Rich and Sharon, Liz and Peter, Rob Fearon (the curator of all things shiny), everyone on Twitter who RTs my babble (you know who you are, guys!), Matthew Smith (the creator of Manic Miner), David Braben, Tōru Iwatani, and anyone who made Atari games in the 1980s.

I would like to thank everyone at AK Peters/CRC Press for the help and support and for publishing my work.

Finally, a massive thank you goes out to you for buying this book and for wanting to do something as cool as to make games. I sincerely hope this book helps your game-making adventures—feel free to tell me about them on Twitter @psychicparrot or drop by my website at http://www.psychicparrot.com.

Introduction

As I was starting out as a game developer, as a self-taught programmer my skills took a while to reach a level where I could achieve what I wanted. Sometimes I wanted to do things that I just didn't have yet the technical skills to achieve. Now and again, software packages came along that could either help me in my quest to make games or even make full games for me; complete game systems such as the Shoot 'Em-Up Construction Kit (aka SEUCK) from Sensible Software, Gary Kitchen's GameMaker, or The Quill Adventure System could bring to life the kinds of games that went way beyond anything that my limited programming skills could ever dream of building.

The downside to using game creation software was that it was tailored to create games within their chosen specific genre. If I wanted to do something outside of the limitations of the software, the source code was inaccessible and there was no way to extend or modify it. When that happened, I longed for a modular code-based system that I could plug together to create different types of games but modify parts of it without having to spend a lot of time learning how the entire system internals work—building block game development that I could actually script and modify if I needed to.

After completing my first book, *Game Development for iOS with Unity3D*, I wanted to follow up by applying a modular style of game building to Unity3D that would provide readers with a highly flexible framework to create just about any kind of game by "plugging in" the different script components. My intention was to make a more technical second book, based on C# programming, that would offer extensibility in any direction a developer might require. In essence, what you are holding in your hands right now is a cookbook for game development that has a highly flexible core framework for just about any type of game.

A lot of the work I put in at the start of writing this book was in designing a framework that not only made sense in the context of Unity but also could easily cope with the demands of different genres.

Prerequisites

You can get up and running with the required software for the grand total of zero dollars. Everything you need can be downloaded free of charge with no catches. You may want to consider an upgrade to Unity Pro at some point in the future, to take advantage of some of its advanced features, but to get started all you need to do is grab the free version from the Unity website.

Unity Free or Unity Pro (available from the Unity store at http://www.unity3d.com)

> Unity Free is completely free for anyone or any company making less than $100,000 per year—it may be downloaded for no charge at all, and you don't even need a credit card. It's a really sweet deal! We are talking about a fully functional game engine, ready to make 3D or 2D games that may be sold commercially or otherwise. There are no royalties to pay, either.
>
> Unity Pro adds a whole host of professional functionality to the engine, such as render culling and profiling. If you are a company with more than $100,000 per year of turnover, you will need a Pro license, but if you find that Unity Free doesn't pack quite enough power, you may also want to consider going Pro. You can arrange a free trial of the Pro version right from the Unity website to try before you buy. If the trial licence runs out before you feel you know enough to make a purchase, contact Unity about extending it and they are usually very friendly and helpful about it (just don't try using a trial license for 6 months at a time, as they may just figure it out!).

C# programming knowledge

> Again, to reiterate this very important point, this is *not* a book about learning how to program. You will need to know some C#, and there are a number of other books out there for that purpose, even if I have tried to make the examples as simple as possible! This book is about making games, not about learning to program.

What This Book Doesn't Cover

This is not a book about programming and it is not a book about the right or wrong way to do things. We assume that the reader has some experience with the C# programming language. I am a self-taught programmer, and I understand that there may well be better ways to do things.

This is a book about concepts, and it is inevitable that there will be better methods for achieving some of the same goals. The techniques and concepts offered in this book are meant to provide solid foundation, not to be the final word on any subject. It is the author's intention that, as you gain your own experiences in game development, you make your own rules and draw your own conclusions.

Additional material is available from the CRC Press Web site: http://www.crcpress.com/product/isbn/9781466581401.

1 Making Games the Modular Way

When I first started making games, I would approach development on a project-to-project basis, recoding and rebuilding everything from scratch each time. As I became a professional developer, landing a job at a game development studio making browser-based games, I was lucky enough to work with a guy who was innovating the scene. He was a master at turning out great games (both visually and gameplay-wise) very quickly. One secret to his success lay in the development of a reusable framework that could easily be refactored to use on all of his projects. His framework was set up to deal with server communication, input handling, browser communication, and UI among other things, saving an incredible amount of time in putting together all of the essentials. By reusing the framework, it allowed more time for him and his team to concentrate on great gameplay and graphics optimization, resulting in games that, at the time, blew the competition away. Of course, the structure was tailored to how he worked (he did build it, after all), and it took me a while to get to grips with his style of development; but once I did, it really opened my eyes. From then on, I used the framework for every project and even taught other programmers how to go about using it. Development time was substantially reduced, which left more time to concentrate on making better games.

This book is based on a similar concept of a game-centric framework for use with many different types of games, rather than a set of different games in different styles. The overall goal of this book is to provide script-based components that you can use within that framework to make a head start with your own projects in a way that reduces recoding, repurposing, or adaptation time.

In terms of this book as a cookbook, think of the framework as a base soup and the scripting components as ingredients. We can mix and match script components from different games that use the same framework to make new games, and we can share several of the same core scripts in many different games. The framework takes care of the essentials, and we add a little "glue" code to pull everything together the way we want it all to work.

This framework is, of course, optional, but you should spend some time familiarizing yourself with it to help understand the book. If you intend to use the components in this book for your own games, the framework may serve either as a base to build your games on or simply as a tutorial test bed for you to rip apart and see how things work. Perhaps you can develop a better framework or maybe you already have a solid framework in place. If you do, find a way to develop a cleaner, more efficient framework or even a framework that isn't quite so efficient but works better with your own code, and do it.

In this chapter, we start by examining some of the major programming concepts used in this book and look at how they affect the design decisions of the framework.

1.1 Important Programming Concepts

I had been programming in C# for a fairly long time before I actually sat down and figured out some of the concepts covered in this chapter. It was not because of any particular problem or difficulty with the concepts themselves but more because I had solved the problems in a different way that meant I had no real requirement to learn anything new. For most programmers, these concepts will be second nature and perhaps something taught in school, but I did not know how important they could be. I had heard about things like inheritance, and it was something I put in the to-do list, buried somewhere under "finish the project." Once I took the time to figure them out, they saved me a lot of time and led to much cleaner code than I would have previously pulled together. If there's something you are unsure about, give this chapter a read-through and see whether you can work through the ideas. Hopefully, they may save some of you some time in the long run.

1.1.1 Manager and Controller Scripts

I am a strong believer in manager and controller scripts. I like to try and split things out into separate areas; for example, in the *Metal Vehicle Doom* game, I have race controller scripts and a global race controller script. The race controller scripts are attached to the players and track their positions on the track, waypoints, and other relevant player-specific race information. The global race controller script talks to all the race controller scripts attached to the players to determine who is winning and when the race starts or finishes. By keeping this logic separate from the other game scripts and contained in their own controller scripts, it makes it easier to migrate them from project to project. Essentially, I can take the race controller and global race controller scripts out of the game and apply them to another game, perhaps one that features a completely different type of gameplay—for example, alien characters running around a track instead of cars. As long as I apply the correct control scripts, the race logic is in place, and I can access it in the new game.

In the framework that this book contains, there are individual manager and controller scripts dealing with user data, input, game functions, and user interface. We look at those in detail in Chapter 2, but as you read this chapter, you should keep in mind the idea of separated scripts dedicated to managing particular parts of the game structure. It was important to me to design scripts as standalone so that they may be used in more than one situation. For example, our weapon slot manager will not care what kind of weapon is in any of the slots. The weapon slot manager is merely an interface between the player and the weapon, taking a call to "fire" and responding to it by telling the weapon in the currently selected weapon slot to fire. What happens on the player end will not affect the slot manager just as anything that happens with the weapon itself will not affect the slot manager. It just doesn't care as long as your code talks to it in the proper way and as long as your weapons receive commands in the proper way. It doesn't even matter what type of object the slot manager is attached to. If you decide to attach the weapon slot manager to a car, a boat, a telegraph pole, etc., it doesn't really matter just as long as when you want them to fire, you use the correct function in the slot manager to get it to tell a weapon to fire.

Since our core game logic is controlled by manager and controller scripts, we need to be a little smart about how we piece everything together. Some manager scripts may benefit from being static and available globally (for all other scripts to access), whereas others may be better attached to other scripts. We deal with these on a case-by-case basis. To get things started, we will be looking at some of the ways that these manager scripts can communicate with each other.

As a final note for the topic in this section, you may be wondering what the difference is between managers and controllers. There really isn't all that much, and I have only chosen to differentiate for my own sanity. I see controllers as scripts that are larger global systems, such as game state control, and managers as smaller scripts applied to gameObjects, such as weapon slot management or physics control. The terms are applied loosely, so don't worry if there appear to be inconsistencies in the application of the term in one case versus another. I'll try my best to keep things logical, but that doesn't mean it'll always make sense to everyone else!

1.1.2 Script Communication

An important part of our manager- and component-based structures is how our scripts are going to communicate with each other. It is inevitable that we will need to access our scripts from a multitude of other areas of the game, which means we should try to provide interfaces that make the most sense. There are several different ways of communicating between scripts and objects in Unity:

1. **Direct referencing manager scripts via variables set in the editor by the Inspector window.**
   The easiest way to have your scripts talk to each other is to have direct references to them in the form of public variables within a class. They are populated in the Unity editor with a direct link to another script.
   Here is an example of direct referencing:

```
 public void aScript otherScript;
```

   In the editor window, the Inspector shows the otherScript field. We drag and drop an object containing the script component that we want to talk to. Within the class, function calls are made directly on the variable, such as

```
otherScript.DoSomething();
```

2. **GameObject referencing using SendMessage.**
SendMessage is a great way to send a message to a gameObject and call a function in one of its attached scripts or components when we do not need any kind of return result. For example,

```
SomeGameObject.SendMessage("DoSomething");
```

SendMessage may also take several parameters, such as setting whether or not the engine should throw an error when there is no receiver, that is, no function in any script attached to the gameObject with a name matching the one in the SendMessage call. (SendMessageOptions). You can also pass one parameter into the chosen function just as if you were passing it via a regular function call such as

```
SomeGameObject.SendMessage("AddScore",2);
SomeGameObject.SendMessage("AddScore",
SendMessageOptions.RequireReceiver);
SomeGameObject.SendMessage("AddScore",
SendMessageOptions.DontRequireReceiver);
```

3. **Static variables.**
The static variable type is useful in that it extends across the entire system; it will be accessible in every other script. This is a particularly useful behavior for a game control script, where several different scripts may want to communicate with it to do things such as add to the player's score, lose a life, or perhaps change a level.
An example declaration of a static variable might be

```
private static GameController aController;
```

Although static variables extend across the entire program, you can have private and public static variables. Things get a little tricky when you try to understand the differences between public and private static types—I was glad to have friends on Twitter that could explain it all to me, so let me pass on what I was told:

Public static
A public static variable exists everywhere in the system and may be accessed from other classes and other types of script.
Imagine a situation where a player control script needs to tell the game controller script whenever a player picks up a banana. We could deal with it like this:

1. In our gamecontroller.cs game controller script, we set up a public static:

```
public static GameController gateway;
```

2. When the game controller (gamecontroller.cs) runs its Start() function, it stores a reference to itself in a public static variable like this:

```
gateway = this;
```

3. In any other class, we can now access the game controller by referring to its type followed by that static variable (GameController.gateway) such as

```
GameController.gateway.GotBanana();
```

Private static
A private static variable exists within the class it was declared and in any other instances *of the same class*. Other classes/types of script will not be able to access it.

As a working example, try to imagine that a script named player.cs directly controls player objects in your game. They all need to tell a player manager script when something happens, so we declare the player manager as a static variable in our player.cs script like this:

```
private static PlayerManager playerManager;
```

The playerManager object only needs to be set up once, by a single instance of the player class, to be ready to use for all the other instances of the same class. All player.cs scripts will be able to access the same instance of the PlayerManager.

4. **The singleton design pattern.**
In the previous part of this section, we looked at using a static variable to share a manager script across the entire game code. The biggest danger with this method is that it is possible to create multiple instances of the same script. If this happens, you may find that your player code is talking to the wrong instance of the game controller.

A *singleton* is a commonly used design pattern that allows for only one instance of a particular class to be instantiated at a time. This pattern is ideal for our game scripts that may need to communicate (or be communicated with) across the entire game code. Note that we will be providing a static reference to the script, exactly as we did in the "Static Variables" method earlier in this section, but in implementing a singleton class, we will be adding some extra code to make sure that only one instance of our script is ever created.

1.1.3 Using the Singleton Pattern in Unity

It is not too difficult to see how useful static variables can be in communication between different script objects. In the public static example cited earlier, the idea was that we had a game controller object that needed to be accessed from one or more other scripts in our game.

The method shown here was demonstrated on the Unity public wiki* by a user named Emil Johansen (AngryAnt). It uses a private static variable in conjunction with a public static function. Other scripts access the public function to gain access to the private static instance of this script, which is returned via the public function so that only one instance of the object will ever exist in a scene regardless of how many components it is attached to and regardless of how many times it is instantiated.

A simple singleton structure:

```
public class MySingleton
{
     private static MySingleton instance;

     public MySingleton ()
```

* http://wiki.unity3d.com/index.php/Singleton.

```
        {
            if (instance != null)
            {
                Debug.LogError ("Cannot have two instances of singleton.");
                return;
            }

            instance = this;
        }

        public static MySingleton Instance
          {
            get
            {
                if (instance == null)
                {
                     new MySingleton ();
                }

                return instance;
            }
        }
}
```

The singleton instance of our script may be accessed anywhere, by any script, simply with the following syntax:

```
MySingleton.Instance.MySingletonMember;
```

1.1.4 Inheritance

Inheritance is a complex concept, which demands some explanation here because of its key role within the scripts provided in this book. Have a read through this section, but don't worry if you don't pick up inheritance right away. Once we get to the programming, it will most likely become clear.

The bottom line is that inheritance is used in programming to describe a method of providing template scripts that may be overridden, or added to, by other scripts. As a metaphor, imagine a car. All cars have four wheels and an engine. The types of wheels may vary from car to car, as will the engine, so when we say "this is a car" and try to describe how our car behaves, we may also describe the engine and wheels.

These relationships may be shown in a hierarchical order:

Car

    -Wheels
    -Engine

Now try to picture this as a C# script:

Car class

    Wheels function
    Engine function

If we were building a game with lots of cars in it, having to rewrite the car class for each type of car would be silly. A far more efficient method might be to write a base class and populate it with virtual functions. When we need to create a car, rather than use this base class, we build a new class, which inherits the base class. Because our new class is inherited, it is optional whether or not we choose to override the Wheels or Engine function to make them behave in ways specific to our new class. That is, we can build "default" functions into the base class, and if we only need to use a default behavior for an engine, our new class doesn't need to override the engine function.

The base class might look something like this:

```
public class BaseCar : MonoBehavior {

        public virtual void Engine () {
             Debug.Log("Vroom");
        }

        public virtual void Wheels () {
             Debug.Log("Four wheels");
        }
}
```

There are two key things to notice in the above script. One is the class declaration itself and the fact that this class *derives* from MonoBehavior. MonoBehavior is itself a class—the Unity documentation describes it as "the base class every script derives from"—this MonoBehavior class contains many engine-specific functions and methods such as Start(), Update(), FixedUpdate(), and more. If our script didn't derive from MonoBehavior, it would not inherit those functions, and the engine wouldn't automatically call functions like Update() for us to be able to work with. Another point to note is that MonoBehavior is a class that is built in to the engine and not something we can access to edit or change.

The second point to note is that our functions are both declared as *virtual* functions. Both are public and virtual. Making virtual functions means that the behavior in our base class may be overridden by any scripts that derive from it. The behavior we define in this base class could be thought of as its default behavior. We will cover overriding in full a little further on in this section.

Let's take a look at what this script actually does: If we were to call the Engine() function of our BaseCar class, it would write to the console "Vroom." If we called Wheels, the console would read "Four wheels."

Now that we have our BaseCar class, we can use this as a template and make new versions of it like this:

```
public class OctoCar : BaseCar {
        public override void Wheels () {
             Debug.Log("Eight wheels");
        }
}
```

The first thing you may notice is that the OctoCar class derives from BaseCar rather than from MonoBehavior. This means that OctoCar inherits the functions and methods belonging to our BaseCar script. As the functions described by BaseCar were virtual, they may be overridden. For OctoCar, we override Wheels with the line:

```
public override void Wheels () {
```

Let's take a look at what this script actually does: In this case, if we were to call the Engine() function on OctoCar, it would do the same as the BaseCar class; that is, it would write "Vroom" to the console. It would do this because we have inherited the function but have not overridden it, which means we keep that default behavior. In OctoCar, however, we have overridden the Wheels() function. The BaseCar behavior of Wheels would print "Four wheels" to the console, but if we call Wheels() on OctoCar, the overridden behavior will write "Eight wheels" instead.

Inheritance plays a huge part in how our core game framework is structured. The idea is that we have basic object types and specific elaborated versions of these objects inheriting the base methods, properties, and functions. By building our games in this manner, the communication between the different game components (such as game control scripts, weapon scripts, projectile controllers, etc.) becomes universal without having to write out the same function declarations over and over again for different variations of the script. For the core framework, our main goal is to make it as flexible and extensible as possible, and this would be much more difficult if we were unable to use inheritance.

The subject doesn't stop here, but that is as far as we will be going here. Each method will be useful in different areas of game development. For example, in cases where a component requires a reference to another object such as a material or a texture, it makes sense to set this via the Unity editor. In cases where we are developing classes that are not applied to gameObjects, we may use a static variable to make it accessible to all other scripts. For the sake of sanity, it is probably likely that we will never use a static object reference without using it as a singleton.

1.1.5 Where to Now?

Think about how your objects work together in the game world and how their scripts need to communicate with each other. I find that it helps to make flow diagrams or to use mind-mapping software to work things out beforehand. A little planning early on can reduce a whole lot of code revisions later on, so try to plan ahead. That said, it is perfectly okay for your structure to grow and evolve as you develop it further. The framework shown in this book ended up going through many revisions along the way before reaching its final form.

Try to break up your scripts into smaller components to help with debugging and for flexibility. Also, try to design for reusability; hard-coding references to other scripts reduces the portability of the script and increases the amount of work you will need to do in the future when you want to carry it over into another project.

2 Building the Core Game Framework

In this chapter, we look at the structure of the base project, to which we will be adding all of the game script components, and the thinking behind it. This chapter will give context to the design decisions made in the core framework as well as provide a good base for understanding how the individual code recipes work together and why they may sometimes have dependencies on other components.

The framework for this book (as seen in Figure 2.1) features six main areas:

1. Game Controller
   The game controller acts like a central communication script for all of the different parts of the game. In a way, it is the main glue that holds together the various components that make up the game.

2. Scene Manager
   This script deals with the loading and saving of scenes. Scenes are what Unity calls its level files.

3. UI Manager
   The UI manager draws all of the main in-game user interface elements, such as score display and lives.

4. Sound Manager
   The sound manager handles sound playback. The manager holds an array containing the AudioClips of possible sounds to play. When the game first begins, the

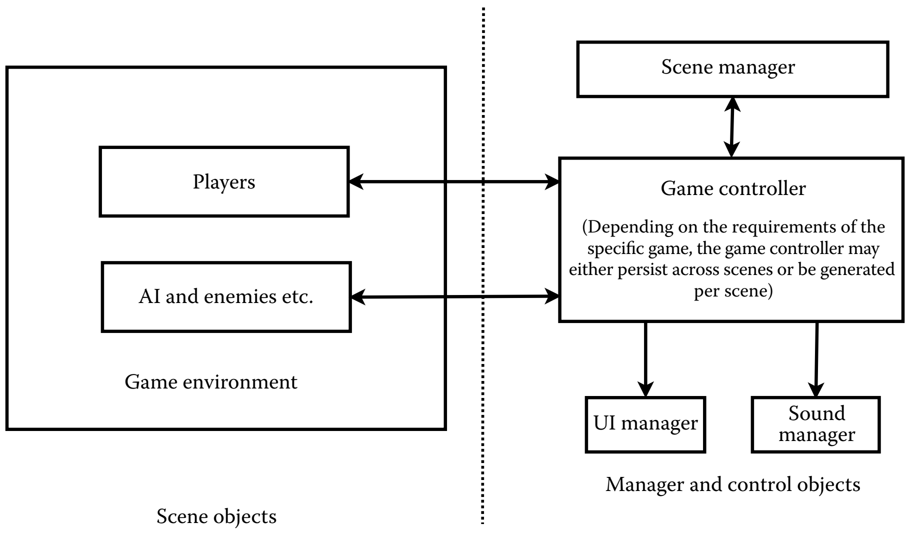
Scene manager
Players
Game controller
(Depending on the requirements of the specific game, the game controller may either persist across scenes or be generated per scene)
AI and enemies etc.
Game environment
UI manager
Sound manager
Manager and control objects
Scene objects

Figure 2.1 Diagram of the game framework.

sound manager creates one individual AudioSource per sound in the array. When we need to play a sound, the index number referring to a sound's place in the array is passed to the manager. It uses the sound's individual AudioSource to play the source, which helps to alleviate problems that may be caused by playing too many sound files from a single source.

5. Players
   Players may already be present in the game scene or may be instantiated by the game controller. The players have a relatively complex structure that will be explored later in Chapter 3.

6. AI and Enemies
   As with players, AI-controlled characters or enemies may already be present in the scene at load time or they may be created at runtime by the game controller or other script.

Although you may choose to work with your own framework, understanding the one used in this book will help in seeing how the components work alongside each other and how they work together. To maximize flexibility, the base classes need to be able to accommodate a host of different configurations. For example, a player control script should allow for input to be passed in from another script so that control systems may easily be switched out. The main player script does not need to know where the input is coming from, but it needs to provide a common interface for an input script to communicate with. The input script may take its input from the keyboard, mouse, artificial intelligence, networked input, etc., and just as long as a common interface is used to communicate with the player controller; it doesn't matter to the player controller as to where the input actually comes from.