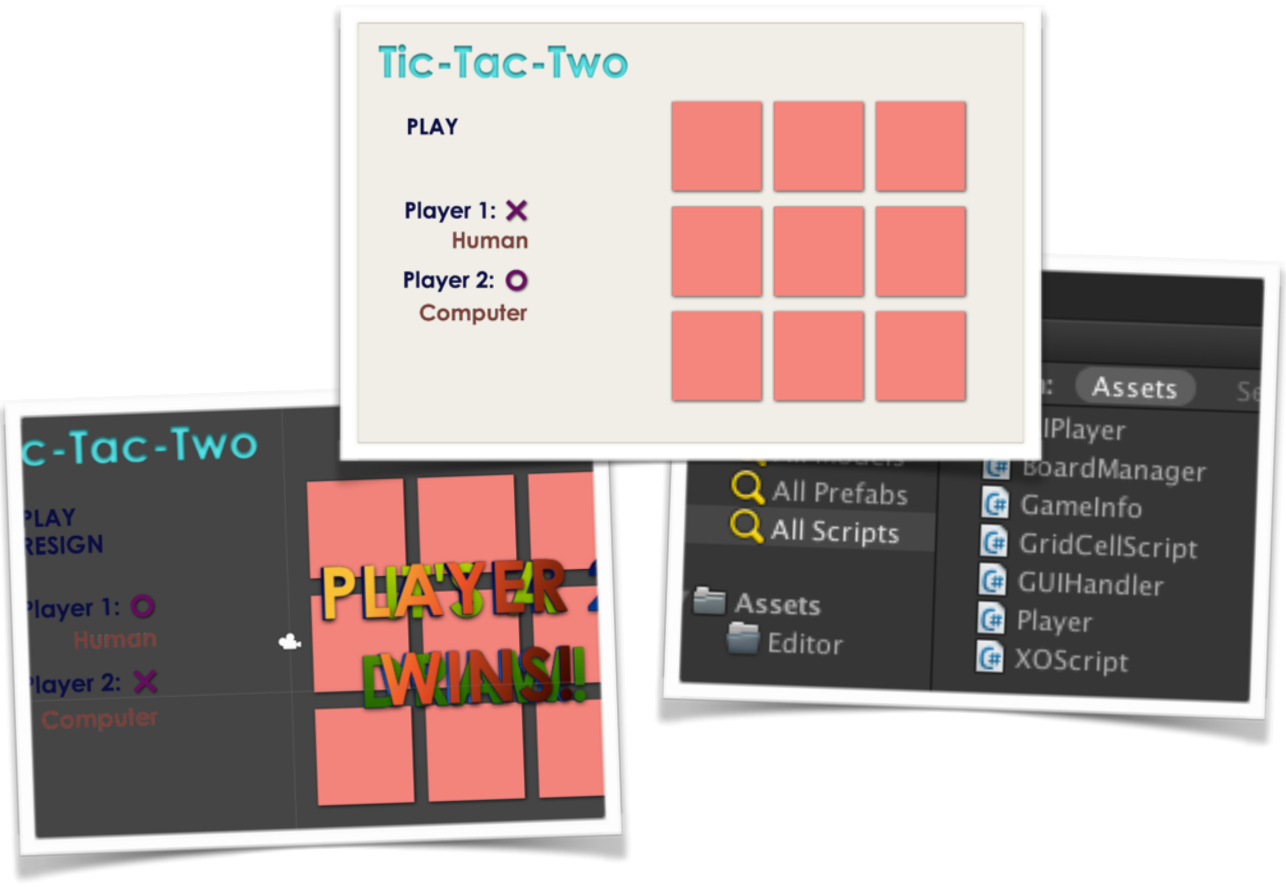
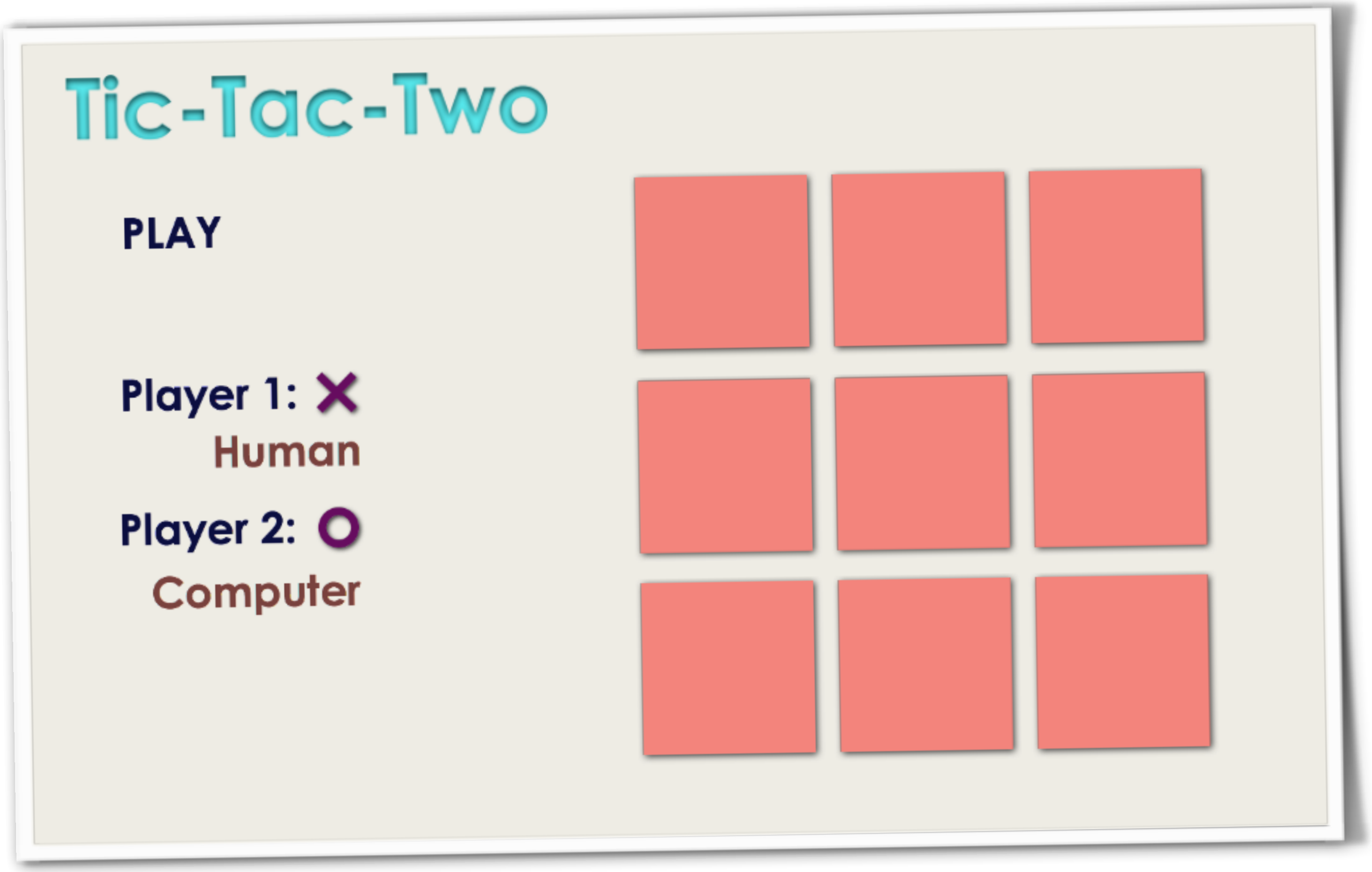
TIC-TAC-TWO

Turn-Based Game Tutorial
(a.k.a. 'Tic-Tac-Tut v2.0')



by Sean Timarco Baggaley

15 September 2014 r2.1



Preface

About the Author

Hi! My name is Sean Baggaley.

I've been working in and around the games industry since the late 1980s. My first professional work—I was a graphics artist and animator at the time—was released for the Sinclair ZX Spectrum while I was still in school. Since then, I've been paid as a bitmap graphics artist, game designer, programmer, audio engineer and done pretty much every non-management role the games industry has to offer.

In recent years, I've been working in the education sector and as a technical author, writing manuals, user guides and tutorials, mostly specialising in games and related industries.

So this tutorial took me back and, for the first time in years, it was something I was working on that didn't feel like work. Which can only be a good thing.

About the Tutorial

This tutorial came about as a result of a post on Unity's forums. Posters were asking for links to tutorials on building turn-based games, but there weren't any obvious candidates to link to. So I decided to write one. Tic-Tac-Toe is one of the oldest examples of a turn-based game, and also a very easy one to understand.

Tutorial Structure

There are two ways to write a tutorial. You can either provide step-by-step instructions for recreating a project, or you can provide the complete project, fully commented and documented up-front, along with a "how it all works" document. I've chosen the latter option. If you prefer the step-by-step format, let me know and I'll see what I can do.

Tic-Tac-Two: The Sequel

The original Tic-Tac-Two tutorial was written for a version of Unity that it now obsolete. While no major changes have been needed until recently, it was the addition of the new 2D sprite features that led to my decision to rewrite the tutorial and create a new version. Furthermore, the promise of the new Unity GUI system—not yet released at the time of writing—would also require major changes to the project anyway.

The new tutorial, which I refer to now as "Tic-Tac-Two", has been completely rewritten from the ground up. Not one line from the original version remains. However, as the new Unity GUI system has not yet been released, I have substituted my own temporary GUI instead as a placeholder. To avoid confusion, a couple of features regarding the AI player and player stats are included in the underlying code, but not accessible using the current GUI. This will be fixed in the next release.

New in this release

- An additional chapter on Finite State Machines has been added;
- Some minor changes and tweaks made here and there to improve the text.

Additional Documentation

The original release included no attempt to explain how programming works. This is remedied, to an extent, in this release by the inclusion of a preview version of an e-book I've been working on as a side project.

This preview version has been modified to cover Unity specifically. The final ebook is intended for a more general audience, so will not be exactly the same. It will certainly be a lot longer.

Unity supports three programming languages: Boo, C# and Javascript. The latter two share a common spiritual ancestor called "C" and look somewhat similar; Boo has a different ancestry, so lacks the curly braces and unusual punctuation.

This project uses C#.

The Tic-Tac-Tut game

Tic-Tac-Tut's features include:

- An AI player;
- The ability to choose select player vs. player, AI vs. AI, and any combination of both;
- The ability to play as either 'X' or 'O';

- The ability to resign the game and reset it to start afresh.

What happened to the three AI characters?

There were three AI modes in the original tutorial. However, only one AI was actually provided: the character of "Dick" was actually the same AI code, but with a 50% chance of playing a random move, while the third character, "Tom", merely picked a random empty square to play.

In this version of the tutorial, only the 'perfect' AI remains. The other modes are so easy to implement that they are now left as an exercise for the reader.

Unimplemented Features (or, Exercises for the Reader)

As the intention of this tutorial is to show how to create a turn-based game in Unity, rather than producing a complete game for sale in an app store, I have avoided unnecessary features and polish that would add little or nothing of value to its educational goals. Therefore:

- Player 1 always plays first (which is no big deal);
- There is support for game stats, but there is nothing to display them at present; this is left as an exercise for the reader;
- There are no fancy animations;
- There is no audio.

In all other respects, this is a complete implementation of the venerable game of Tic-Tac-Toe. A game that has a distinguished ancestry dating at least as far back as the Roman Empire.

Compared to the previous version, the AI / player-handling system is greatly improved: you can now have the computer play against itself, or two human players playing each other, or any combination of both types of player. You can even switch the playing pieces, so Player 1 plays as 'O'.

Enough of this procrastination: Onwards!

Introduction

Program fragments in this tutorial document are taken from the project's scripts. Comments have been removed for space and the fragments are intended purely for illustrative purposes.

Prerequisites

This tutorial requires only the plain vanilla installation of the free (also known as "Indie") version of Unity. No additional plugins or extensions required.

Prior Knowledge

This tutorial involves a little light programming.

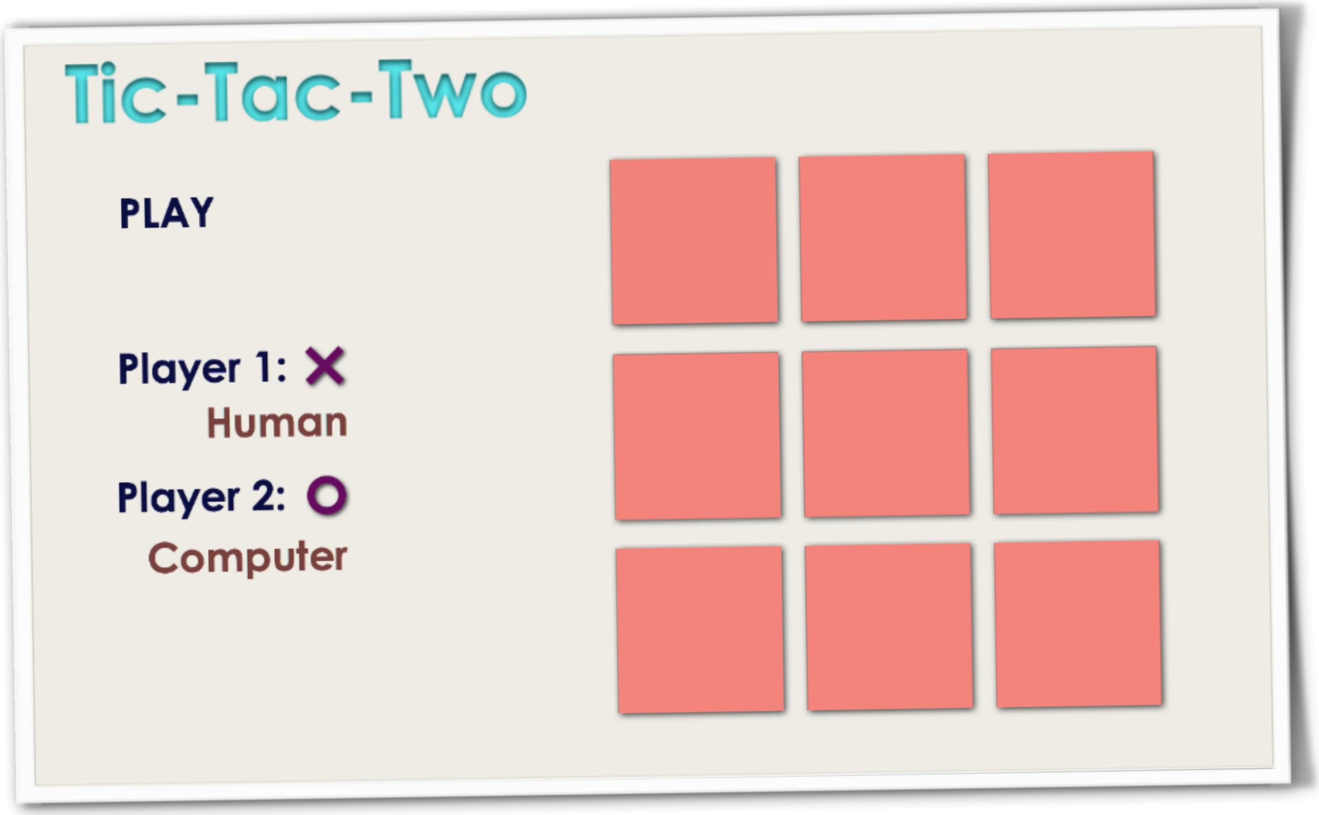
In the previous version of the tutorial, I assumed basic understanding of the C# programming language supported by Unity. This is no longer a requirement with this release.

After looking through the existing documents and books on programming, I felt the need to explain how programming works from a non-mathematical perspective. I see programming as a form of translation: the object is to explain what you want the computer to do in terms it can understand.

Therefore, the key to understanding programming is understanding *how computers think*. And that's what the **Programming Primer** document included with this tutorial—it's in the same folder you found this document—attempts to do.

Neither document will explain the Unity Editor. Unity already comes with a bunch of free tutorials and videos explaining what all the knobs and twiddly bits do; it would be silly to duplicate all that information here.

PART 1
The Tutorial



This project combines a tutorial with a ready-made project. Rather than attempt to dictate a single "One True Path" to success—there's no such thing—my intention is to dissect the tutorial and explain:

- How it works
- Why I chose to do it this way
- Some alternative approaches

For convenience, and to avoid having quotation marks all over the place, important terms are shown in red text.

The project was approached as if I were creating a game intended for release, but doesn't include fancy menus, sound effects, or animations.

This approach means you get a behind-the-scenes view of game development, including insight into thought processes and my personal—if somewhat unusual—approach to programming and this profession.

Scripting vs. Programming

Many professional programmers often distinguish between programming and scripting. The theory is that a script targets an application (e.g. Unity, or Microsoft Access), whereas a program has no such 'container' and talks directly to the underlying operating system, such as Windows, Mac OS X, or Android.

The point is that a program is only limited by the features offered by that operating system, rather than those of a more tightly defined "container" application.

Naturally, professional programmers consider programming to be more hardcore, but the lines between the two have become increasingly blurred: The programming language known as C#, supported by Unity, is exactly the same programming language known as C# that is used to develop complete applications for Microsoft Windows and other platforms.

> In IT, a "platform" is a specific combination of hardware and operating system. E.g. an Intel processor-based computer running Microsoft Windows 8 is one platform, while an Intel processor-based computer running Apple's OS X (e.g. an iMac), is a different platform.

As a result, I tend to avoid using the term scripting as it is, to all intents and purposes, the same thing as programming. You are a programmer. Welcome!

Some Terminology

There are many programmers who talk about "Modular", "Component-based", "Object Oriented", or "Functional" programming, and other program design approaches. I find these tend to confuse programmers who are just starting out.

Fundamentally, a programmer is concerned with three things:

- Data, which we want to process in some way,
- Functions, which do the processing,
- User Interfaces (UIs) that let users control the above.

The data is stuff like 3D models, audio clips, object states, and so on. Much of the programmer's work involves deciding how to organise all this data.

Functions are the programming part. They process our data. These are what we spend most of our time writing in Unity: they contain the instructions that glue all our Unity components together and make them do cool stuff.

Finally, as we're making interactive programs, we also need to allow the user to interact with our data, accessing the functions via a convenient user interface.

That's really all there is to it.

The hardest part is often the organisational side, not the programming. You'll often spend a lot more time on the *structural scaffolding* than you will on the code that actually does stuff your game needs.

Object Oriented Programming

Unity's supported programming languages take a primarily Object Oriented Programming approach to programming.

Object Oriented Programming is an organisational tool: we divide a project up into component elements, called "objects". Each "object" includes all the necessary program functionality needed to work with the data that object represents.

For example: a "door" object might include data that defines its shape (in the form of a 3D model), its colour, and its state—e.g. "locked", "open" or "closed". When another object needs to open the door, it tells the door object to "open" by calling its "Open" function.

How that function actually works is completely hidden from the other objects, which makes it much easier to split up the programming work.

You'll see a lot of this sort of thing in Unity programming.

Classes

In Object Oriented Programming parlance, a "class" defines a type of object. The Class is the basic organisational unit when programming in Unity.

Here's the tricky part: ***a Class is not an object!***

Each Class is simply a *recipe* used to produce instances of that class, in much the same way that a cake recipe in a book defines how you make one, but is not, in itself, a cake.

If you want to create a "Door" object, you would create an instance of its class. Behind the scenes, the "Door" Class is used as a recipe to produce that new Door instance.

"Methods" vs. "Functions"

Programs usually contain a lot of functions. Each function contains a block of code that performs a specific function, hence the name. A "ShowFancyAnimation" function would contain the code needed to show a fancy animation; a "CalculateNextMove" function contains the code needed to calculate a player's next move, while a "DestroyAllHumans" function would contain the code necessary to bring about the end of civilisation as we know it and create a new robotic Utopia. I'm still working on that last one.

In Object Oriented Programming jargon, a function that is part of a class is known as a method.

As I learned to program long before this Object Oriented Programming technique became fashionable, I tend to refer to "methods" as functions regardless; in this tutorial, a "function" is therefore synonymous with a "method".

C# vs. Javascript

Unity currently supports three programming languages out of the box: Boo, C# and Javascript. Of these, C# and Javascript are the most closely related, both being part of the "C family" of programming languages.

For my tutorials, I have chosen to use the C# language.

If you have never used C# before, an important difference is its *strictness* about data types. You have to decide how you want a value to be stored. In this example, I'm defining my variable as being an integer type:

```
    int myValue = 45;
```

In Javascript, this is equivalent to:

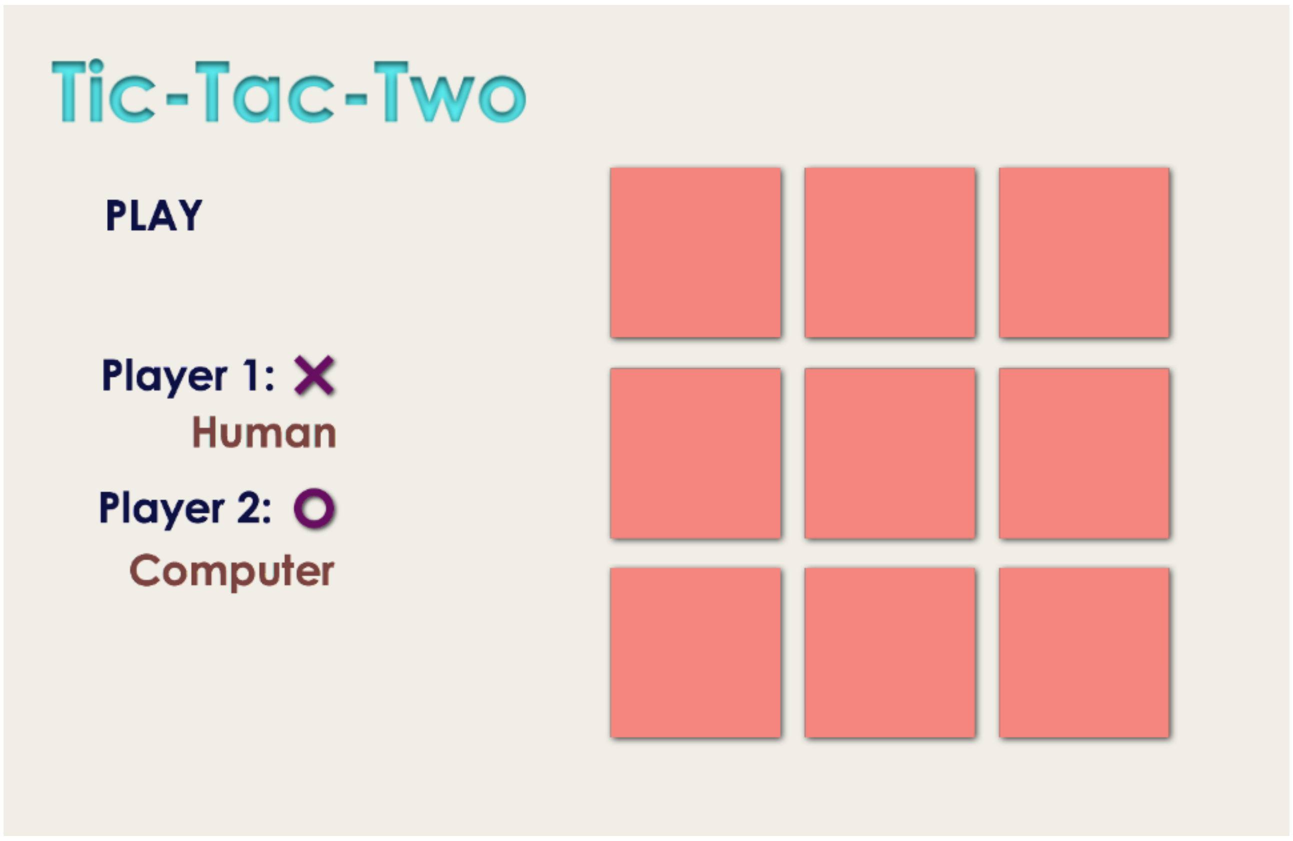
```
    var myValue = 45 as integer;
```

In C#, you have to explicitly name classes in your files, as well as a number of other details that Javascript handles on your behalf.

Using C# therefore looks like more work, but it has the benefit of making it easier to follow exactly what is going on. It's more transparent.

For a tutorial, that's a good thing.

The Game



When the game is started, the player sees the screen shown above.

At this point, you can choose which piece to play—X or O—and which player(s) should be played by the computer. You can choose to have the computer play both sides, for example.

During play, the user interface changes slightly: the PLAY button is replaced by a RESIGN button, and only the current player's information is shown.

The entire GUI for this version of the tutorial was created entirely using 2D sprites. All the pointing and clicking stuff was written by hand and does not use any of the current Unity GUI system; no third-party plugins or extensions have been used.

In the next release, the GUI will be replaced by the new Unity GUI system that was announced late last year and is currently expected in the Unity 4.6 release.

Due to time constraints, I won't be releasing the next version until after the release of Unity 5, which is expected sometime after the Unite event in late summer 2014.

Before discussing how everything is glued together, the next section will first look at all assets used in this tutorial and how they are set up in the project.

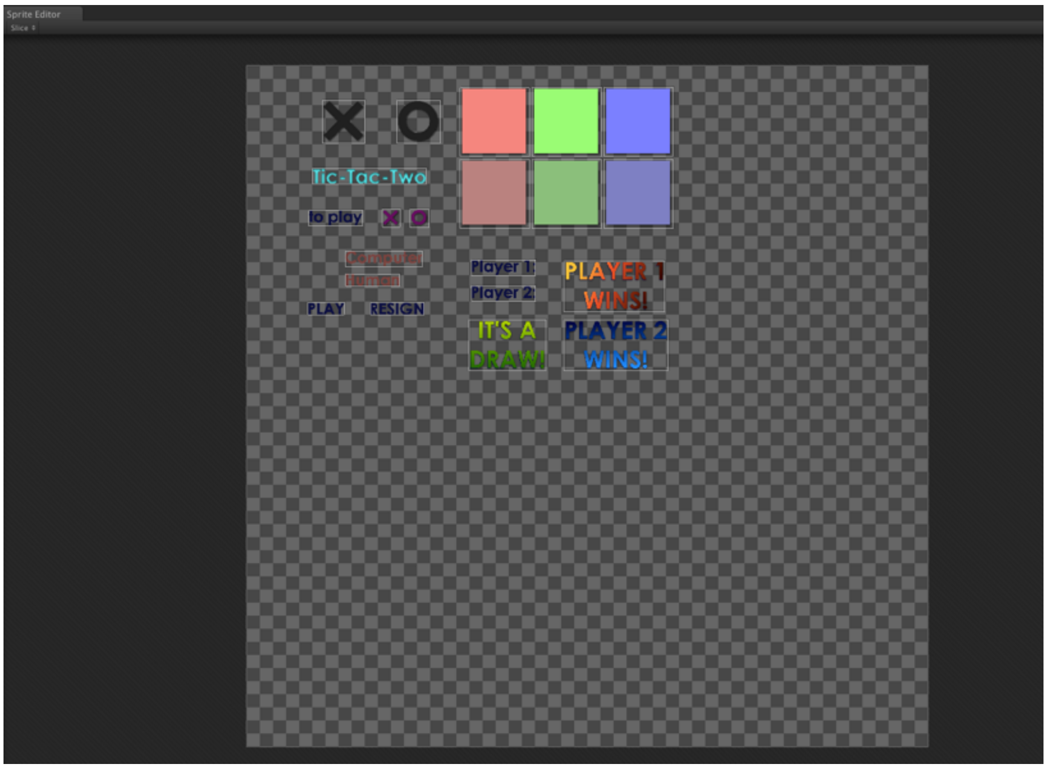
Unlike the original release, the actual graphics are all stored in a single file and chopped up using Unity's built-in Sprite Editor.

Assets Overview

Assets

Every Unity project has a number of assets. An asset can be an image file, a sound file, or even a script. (In 3D projects, you could also have 3D models, textures and materials, animation sequences, and more, but we're not using those.)

For this tutorial, all the assets were created using a Mac program called iDraw by Indeeo, Inc. This is a vector art application similar to Adobe Illustrator or CorelDraw; any such application will do for this kind of work. All the assets were saved as a single image file, which was chopped up into individual 2D sprites in Unity using the Sprite editor feature:



All scripts are specific to this project and written in C#. No additional scripts were used.

What is a sprite?

A sprite is a collection of (usually small) images. Sprites can be animated, like the birds in Rovio's *Angry Birds* games, or the character of Mario in Nintendo's original 2D *Super Mario* platform games.

In this project, the sprites do not animate, so each sprite is just a single image.

The Board

This comprises BoardCell prefabs arranged in a 3 x 3 grid.

The BoardCell prefab deserves a closer look. Each contains:

1. a BoardCell sprite,
2. an 'X' sprite, and
3. an 'O' sprite.

Elements 2 and 3 are defined as 'children' to the BoardCell sprite itself.

Furthermore, each of these sprites has a script attached to it: GridCellScript attached to to the board cell sprite, while the 'X' and 'O' sprites each have an XOScript attached.

Each of the prefab's elements also had a specific tag assigned to it in the Editor to help the scripts work; without the tags, the XOScript in particular won't know what kind of sprite it's connected to.

Due to a glitch in the Editor at the time I was working on this aspect, I ended up arranging these prefabs into a 3 x 3 cell grid by hand and named each one according to its position in the grid.

The arrangement is important as I needed to add each one manually to the Board script's grid array in the right order, from top-left to lower-right. This is a requirement for some of the functions I wrote, which assume the cells are arranged in this right order.

The board makes use of three layers of scripts:

BoardManager is the 'master' script that contains all the high-level stuff, like the "Has the game been won yet?" test. It also—unlike the previous version of the tutorial—contains most of the AI code related to calculating the scores for each cell after each turn.

At the lowest level is XOScript, which simply hides / shows the "X" or "O" graphic it is attached to. The same script is used for both symbols.

Designing Tic-Tac-Tut

Design Overview

A very popular design pattern in use today is the Model-View-Controller (or "MVC") design for an application, which I attempted to follow in the original tutorial. The idea is that you split your program into three major components: a Model component, one or more View components, and one or more Controller components.

You can see this design pattern in most desktop computer applications, including those on Windows, OS X and many of the GNU/Linux desktops as, in most applications, the "View" is directly analogous to an application's window.

In the "MVC" design pattern, the idea is that you have a **Model** that represents the core of the application: it defines what the application does. Next, you have the **View**, which is the bit of the program that defines what the application looks like. Finally, you have the **Controller**, which glues the other two components together and handles all communication between them. This is where the commands from the user (from the View component) are passed on to the part of the program that does all the work: the Model. Neither the Model, nor the View can talk to each other directly, and neither knows how the other works.

By splitting the application into these discrete components, you can thus split the development work and have three teams, one working on each component. Furthermore, the Model component is usually very easy to adapt to run on other platforms—a process known as "porting" (short for 'transporting').

Lessons Learned: MVC doesn't work well for games.

Many typical desktop applications, such as a word processor, or spreadsheet, fit perfectly into the MVC design pattern, as do many others, because most of these patterns assume only one person is interacting with the program at any one time.

Games throw a very large spanner into the works: A computer ("AI") player cannot play the game the way we humans do, by looking at the screen and looking for buttons, icons and board cells. You'd need some heavy-duty image recognition programming for that to work, and that's serious overkill.

After attempting an MVC model for the original "Tic-Tac-Tut" tutorial, I decided in this rewrite to throw all that out the window. It's just too much work for very little gain. It *can* be made to work, but it means having *multiple* View and Controller components. This is fine if you're making a big, complex game, but it is just too much for a simple two-player Tic-Tac-Toe game.

I've found the best solution is to see each script as having both a *client* element, and a *server* element. This is terminology taken from the computer network field: In a computer network, a 'server' is a computer that provides services to other computers. A web server, for example, is a computer that serves web pages up to computers connecting to it over the Internet. A *client* is a computer that *requests services from a server*. When you are browsing a website on your computer, your computer is the *client*, while the web server is the *server*.

A computer can be both a server *and* a client: a web server might, for example, request product information from yet *another* server. This could be information about a product a client wants to buy, or it might ask a social networking website for information to be displayed in a widget on the web page. In that particular situation, the web server briefly becomes a client.

This is the approach I've taken with the scripts in Tic-Tac-Two: a script might be mainly a server, mainly a client, or a mix of both. For example, the trio of scripts used by the game board itself are mostly servers—**XOScript** doesn't ask for information from any other scripts, so it is never a client. On the other hand, **BoardManager** contains most of the AI code, so it acts as both a client for **GridCellScript**, and as a server for the **AIPlayer** code.

Recycling

You'll often be told that it's a good idea to re-use scripts you've already written in other projects, because you know by then that they've been fully debugged and tested. Using such scripts saves you having to write, test and debug fresh code. It's recycling, and we all know recycling is good, right?

Well, yes… and no.

The best candidate scripts for reuse are those providing general functionality that isn't specific to your current project. In Tic-Tac-Two, the Player script, which mostly defines a Player data type (in the form of a C# class), could be reused in other games, for example.

Could we reuse the game board scripts? Not really: they're much too specialised and embody most of the game-specific logic. There's not much point reusing these in a platform game or space exploration game.

So, it's a good idea to think carefully about how you might reuse your scripts and *only if it makes sense to do so* should consider writing them in such a way as to make them easier to reuse in later projects.

If a script does not lend itself well to reuse in other projects, you can go ahead and write it with *close coupling* to other scripts it has to work with…

Close and Loose Coupling

When scripts are considered closely coupled, it means each script relies on knowing how the other works. This makes re-using such scripts in new projects tricky: you might need to include other, 'coupled' scripts as well, even if they don't add anything of use to your new projects. Close coupling therefore makes the most sense for scripts you know aren't going to be recycled in that way.

Conversely, a loosely coupled script knows nothing about how the other scripts it talks to work, so can be easily re-used in another project.

Scripts & Classes

When working with C# or Javascript, each script typically defines a single class. Therefore, when I say "script" in this tutorial, you can substitute the term "class" instead as they're essentially the same.

A major disadvantage of closely coupled scripts is that they can be harder for a team to work on at the same time. For small projects like Tic-Tac-Two, this team development aspect wasn't a major consideration as I'm a one-man operation, but it is worth remembering this when working on team projects.

A major disadvantage of *loosely* coupled scripts is that they can sometimes result in convoluted function call sequences, with the computer being told to jump from a function in one script to another where the necessary work has to be done. This is particularly common when dealing with the GUI: a gameplay script might need to start an animation sequence, which may be handled by the GUI script, so we have to write a 'stub' function in our gameplay script that simply hands control over to another function in the GUI script.

In the Tic-Tac-Two project, the various board cell scripts are closely coupled as it makes little sense to separate them.

Most of the other scripts are loosely coupled and know little, if anything, about each others' inner workings.

Simulated Difficulty

In this tutorial, I've made a deliberate choice to complicate matters slightly in order to illustrate the reality of working with loosely coupled scripts:

One of the scripts uses a slightly different format for some of its data than another, so there's a data conversion step involved when the two scripts communicate.

This simulates a common situation when using code written by someone else—for example, when working with many products bought from the Unity Asset Store. As that code's programmers will have no way of knowing what data structures you intend to use, there is often a need to convert data structures back and forth using some 'glue' code.

The tutorial also illustrates the occasional need for 'stub' functions mentioned earlier.

Refactoring

Refactoring is the process of looking for duplicated, or very nearly duplicated, code and collecting it into a single function instead. This keeps all the code for a particular process in one, easily-located (and debugged!) place.

Copy-pasted code can seem like a simple solution, but it can easily make maintaining your programs a nightmare if taken too far: instead of making a single change in a single function, you may find yourself hunting through thousands upon thousands of lines of program code to make sure you've changed *all* the duplicated code too.

Refactoring therefore saves a lot of time in the long term, even if it means taking a small hit in the short term.

Refactoring means you have less code to write, test and debug.

The best, fastest and most bug-free program code you will ever produce is the code you never have to write in the first place.

I've done quite a bit of refactoring of my scripts for this tutorial, but as a tutorial's primary purpose is to educate, I haven't been as rigorous about this as I could have been.

For example, the **BoardManager** script includes a few functions that include repetitive sequences of code. I've left these as-is because it's easier to follow the underlying algorithms this way. If I had refactored them as much as possible, the flow of the logic would be a little more difficult to follow, for little benefit.

That would be fine if this were intended as a game for publication, but it's bad for a tutorial.

Taking Turns

Unity is a real-time game engine. It doesn't have a 'turn-based' mode, but that's fine: we don't need one. All we need to do is create the *illusion* of a turn-based game, which is much the same thing.

We do this using Finite State Machines, which are covered in detail in the accompanying Programming Primer. If you haven't read it, please do so: Finite State Machines (FSMs) are what make each program—including each game—unique.

In Tic-Tac-Two, I use FSMs to make the game turn-based. The trick is simply to ensure the inactive player isn't allowed to do anything until it's their turn.

In some turn-based games, such as an old-school turn-based war-game, we don't want the inactive player to see what the active player is up to, so it's a good idea to add *transitional states* to our game's master FSM. These would be used to blank the screen and show a "Get Ready Player [x]" message.

Despite being a turn-based game, we *don't* want the graphics and animations to stop dead while the player works out his turn. For example, you might want to have the elements on the game board animate occasionally, to keep the player's attention. Or perhaps you're designing a turn-based strategy war-game or RPG, in which case you might want flames and other effects to keep animating while the player is pondering their move.

In fact, stopping the Unity engine entirely is not advisable as *everything* would grind to a halt, which is not what we want.

So the solution is to design our game such that the turn-based element is embodied by our Finite State Machines.

> Note the plural: most games have more than one FSM. The hard part of game design is actually designing your FSMs and ensuring they interact correctly, and this is easily one of the most common causes of bugs!

In Tic-Tac-Two, the FSM that handles the turn-based states lives in the **GameInfo** data structure, which is used mainly by the **GUIHandler** script to determine how the GUI should behave.

PART 2
The Scripts

Overview

Seven scripts are used in Tic-Tac-Two. All were written specifically for the tutorial, with none coming from the Standard or Pro Assets.

The scripts are grouped loosely as follows:

The User Interface:

- GUIHandler

The Game Board:

- BoardManager
- GridCellScript
- XOScript

Game Logic:

- AIPlayer
- BoardManager*

Utility Scripts:

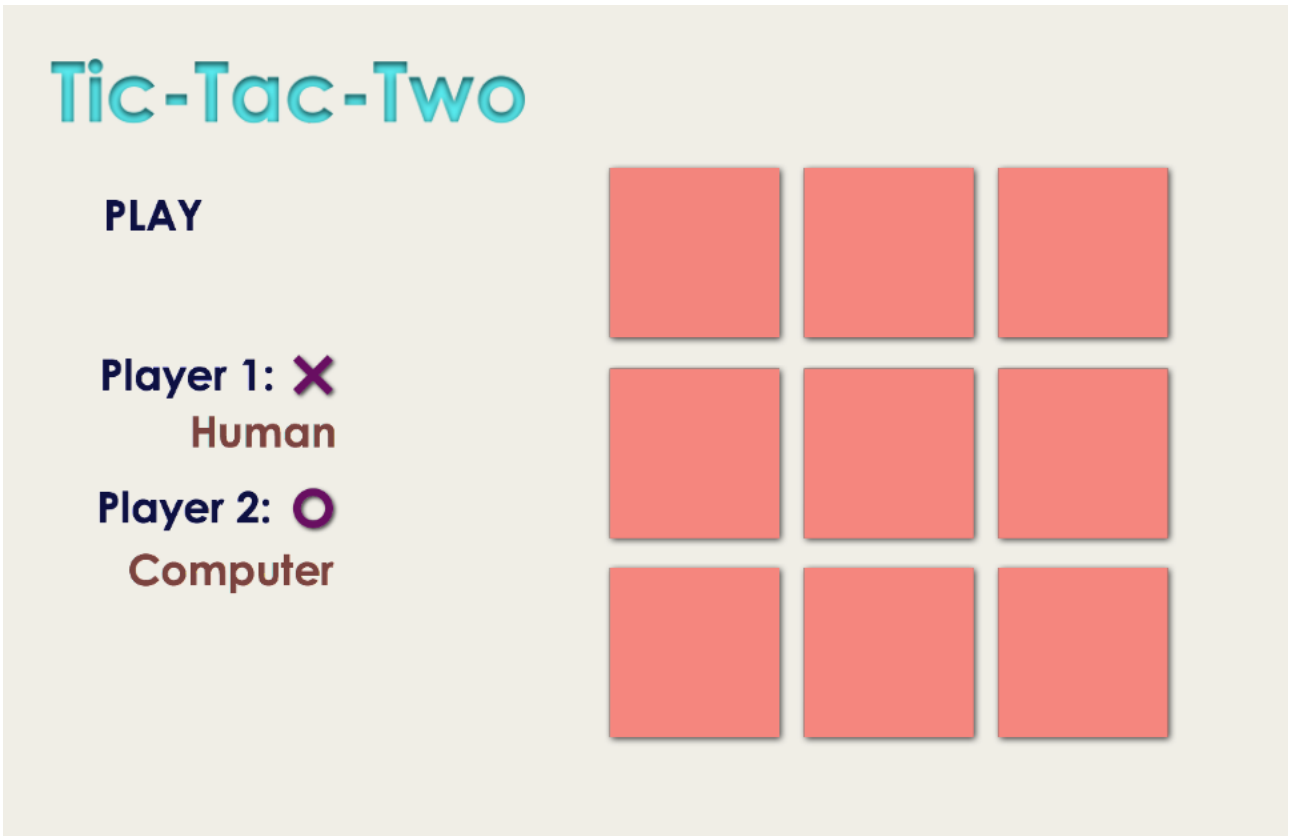
- Player
- GameInfo

** (The BoardManager script also handles some of the AI processing.)*

This section looks at each of these scripts in turn.

The GUIHandler Script

The graphical user interface in Tic-Tac-Two looks like this:



The GUIHandler script does all the user interface stuff. It checks for mouse clicks / screen taps and works out what you've clicked on, then simply shows or hides graphical elements as appropriate.

The GUIHandler script also updates the game's *state*, which tells us whether we're playing a game, or waiting for the player to set a game up.

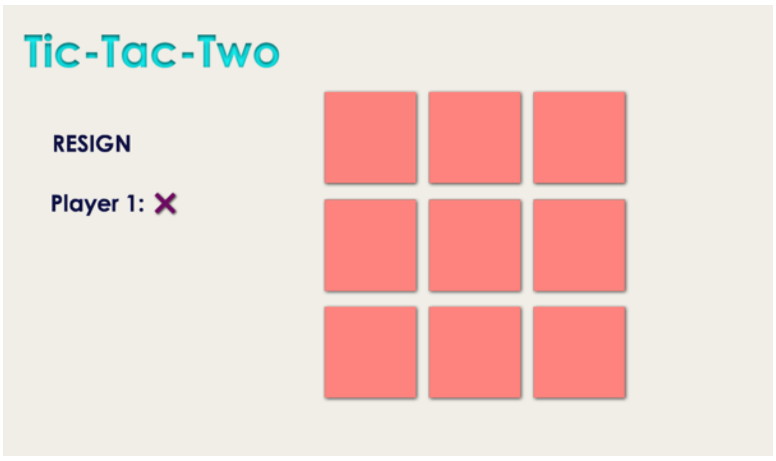
States are a major part of Unity programming, so we'll be coming back to those later.

GUIHandler also manages the game itself, which is why—unlike the old tutorial—there's no separate game management script. It's all done in GUIHandler, with some duties shared with the BoardManager and AIPlayer scripts.

When the game is started, you can click on the "X" and "O" symbols to swap them over between the players; click on either the "Human" or "Computer" entry to toggle whether the player is a human or AI player, or click "PLAY" to start playing a game.

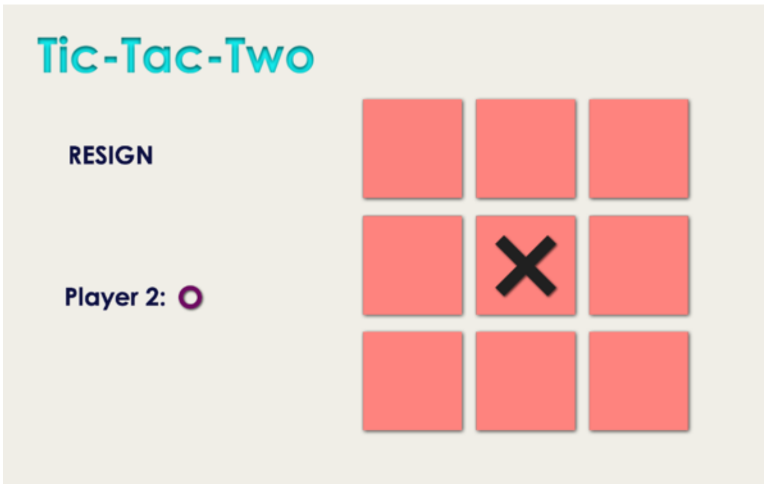
When you click PLAY, the GUIHandler script changes the game's state (which is stored in the GameInfo class defined in the script of the same name, covered later).

The change of game state means the GUI changes a bit, to look like this:



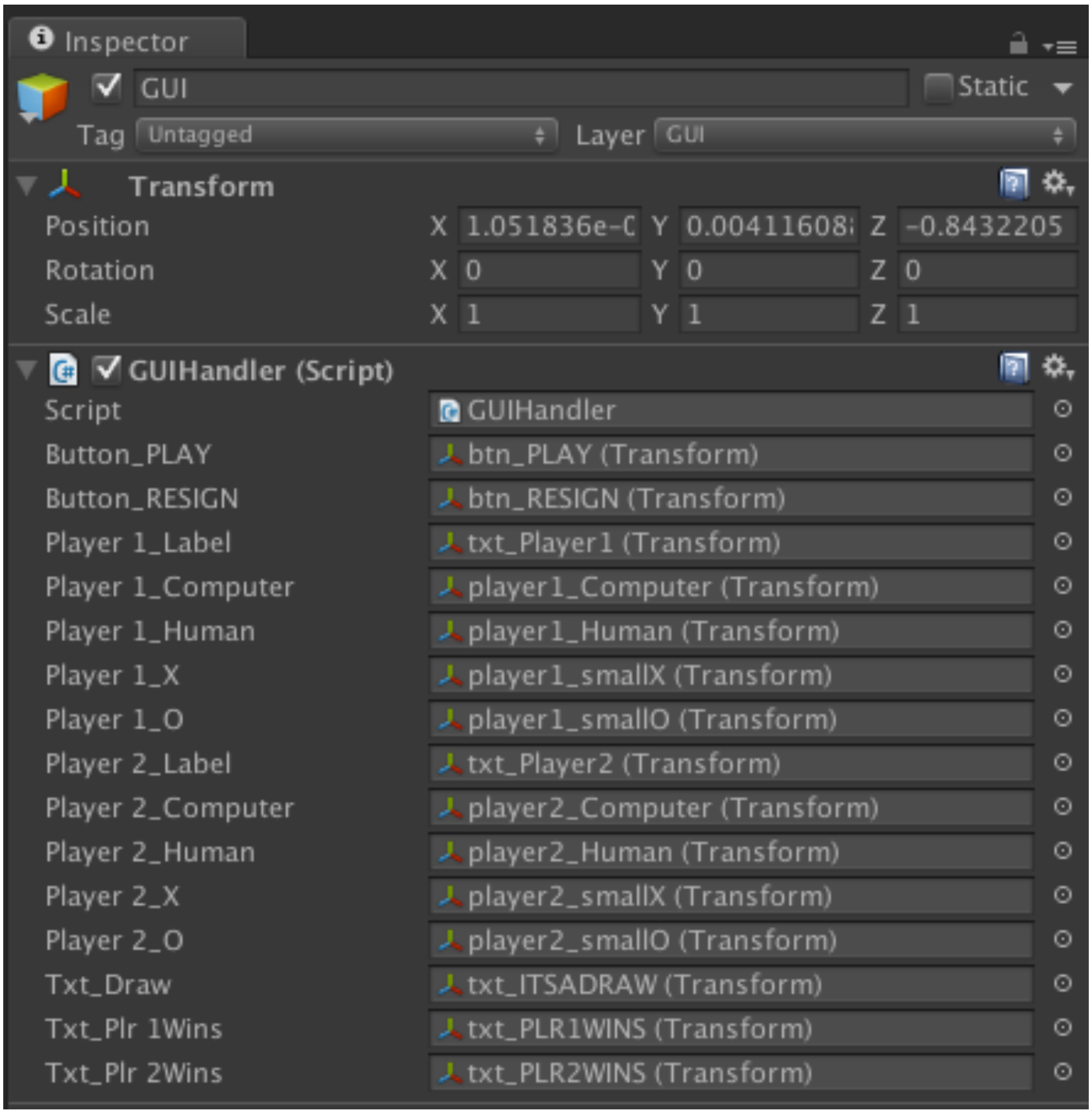
Note that the PLAY button has vanished—it's actually still in the scene; we've just hidden it.

The GUIHandler script also automatically shows the current player and the piece they're playing ("X" or "O"). After playing a move, the GUI changes as shown:



Note that the Player 1 text has been hidden and the Player 2 information is shown instead. All the script actually does is show or hide individual 2D Sprite objects; it doesn't delete them, or move them elsewhere.

At the top of the GUIHandler script itself, I declared a bunch of **public** Transform variables, each of which appears in the Unity Editor's Inspector as a slot:



GUI

Transform

	X	Y	Z
Position	1.051836e-C	0.00411608:	-0.8432205
Rotation	0	0	0
Scale	1	1	1

GUIHandler (Script)

Script	GUIHandler
Button_PLAY	btn_PLAY (Transform)
Button_RESIGN	btn_RESIGN (Transform)
Player 1_Label	txt_Player1 (Transform)
Player 1_Computer	player1_Computer (Transform)
Player 1_Human	player1_Human (Transform)
Player 1_X	player1_smallX (Transform)
Player 1_O	player1_smallO (Transform)
Player 2_Label	txt_Player2 (Transform)
Player 2_Computer	player2_Computer (Transform)
Player 2_Human	player2_Human (Transform)
Player 2_X	player2_smallX (Transform)
Player 2_O	player2_smallO (Transform)
Txt_Draw	txt_ITSADRAW (Transform)
Txt_Plr 1Wins	txt_PLR1WINS (Transform)
Txt_Plr 2Wins	txt_PLR2WINS (Transform)

As you can see, each slot is linked to its related GUI element. I put all these elements in the Hierarchy panel, under a 'GUI' GameObject, with the GUIHandler script attached to that GameObject.

If you look in the Hierarchy panel, you will see some GUI elements are shown in a faded grey. This means they're in the Scene, but are not 'active'. These are toggled on and off by the GUI. You can see them lighting up and dimming in the Hierarchy panel when you run the project and click on the various GUI elements.

Now, I could have grabbed each individual component using script code, but there's little benefit to be had from doing so; I only need to set these slots once. However, for the next set of GameObjects, I did exactly that:

```
        private Camera mainCamera;

        private GameInfo gInfo;

        private BoardManager board;

        private AIPlayer aiPlayer;
```

These are defined as *private* variables—i.e. they can only be accessed from within the GUIHandler class itself and thus do not appear in the Inspector.

Let's look at the functions themselves…

The Start() function

In the **start()** function, I set up each one of these variables. First, **gInfo** is *initialised* by creating a new *instance* of a **GameInfo** object. This sets aside some memory to store the data contained in a GameInfo object.

(If these terms are confusing you, I recommend reading the Programming Primer document included with this tutorial.)

Next, we go and find the main camera for our active Scene. This is easily done by grabbing it from **Camera.main**. This always points to the Scene's main camera. I then did a quick sanity-check to make sure we actually got a main camera. (It's always worth adding such checks as they make tracking down bugs much easier.)

Next, we use a C# language feature to find the only object matching the **BoardManager** *type* in the Scene. There should only ever be one of these. (The '<' and '>' symbols are called "angle brackets" when used like this.) The **FindObjectOfType <BoardManager> ()** function looks through the Scene for any objects defined created from the **BoardManager** class. This is faster than searching for an object by an explicit name, though the speed advantage is irrelevant here as we only do this once, during startup, anyway.

The same function is used to fetch the **AIPlayer** script too.

It's good practice to sanity-check these too, but I've left that as an exercise for the reader.

Next, we initialise—i.e. set up—the data for the two players.

This data is stored within the **gInfo** object, which includes two instances of the **Player** class.

After setting the default player values, we then do a sanity-check on the 2D sprites that contain our GUI graphics. These should be set in the Inspector, as mentioned earlier; if any are missed out, this code will spot it and print a message to the Unity debugging console.

Finally, I hide the "It's A Draw" and the two "Player [x] Wins!" graphics.

> Note how some functions are declared as `public`. This makes them accessible to the other scripts. Functions without that declaration are invisible outside the script file.

A note on localisation

I should stress that at no point is any actual text being used in the GUI: they're all just images. The computer itself has no idea what these images contain.

Experienced developers reading this may ask how I would go about localising the game—i.e. translating it—for other languages. The answer is simple: I wouldn't. Not because of the game, but because of this tutorial and its accompanying preview ebook.

However, I do use graphics software that lets me replace text in graphics programmatically. Adobe's graphics applications support such features as variables that can be set using scripts, as does *LikeThought*'s Mac-only app, *Opacity*, which has been specifically designed for this kind of work.

I generally prefer to localise at this original source material level, as it typically gives better results. It also means I can make modifications to the graphics if it turns out the translation doesn't quite fit.

Localisation is hard to get right. May day-job while working on this tutorial and its predecessor is translation, so I've seen the workflow from both ends. I think the best advice I could give anyone thinking of getting their game(s) localised is this:

Localisation is hard. Provide the translator with as much contextual information as you can. It's always better to give too much than too little. You'd be amazed at how many companies expect translators to perform miracles of mind-reading.

For the love of god, *please* don't use spreadsheet files! Spreadsheets are fine for number-crunching, but they make truly awful word processors.

GetCurrentPlayerPieceAsInt()

This function has a long, but descriptive, name and provides some 'glue' code…

The BoardManager script doesn't care about the actual players: it works directly with the "X" and "O" pieces. It therefore doesn't know which player is playing as which piece. The BoardManager script therefore uses its own data type to store the game state, independently of the GUIHandler script's own data structures. We therefore need a way to convert from one data type to the other.

The **GetCurrentPlayerPieceAsInt()** function does this conversion job for us. It turns the "piece" **enum**erated type used by the Player information class into a simple **int**eger type. (Technically, we're converting into one of the three **const int** types defined within the BoardManager script.)

Hard-coding numbers into scripts can make the program code you write a little harder to understand. Not just for others, but for you too: how likely are you to remember why you chose some random-looking number a few months or years later on?

To avoid this, I use the **const** keyword frequently to give a fixed number a meaningful name.

GameWon() & GameDrawn()

This function is called by the BoardManager script—hence the **public** keyword; without it, BoardManager wouldn't be able to 'see' the function at all.

The function is called when the BoardManager script detects that the game has been won or drawn.

As we learned earlier, the BoardManager script doesn't know which player has won: only which *piece* has won: X or O. So we first need to determine which player was playing the winning piece.

First, we ask the BoardManager script whether 'X' won. Then we look at Player 1's piece and use the results from both to work out which player won:

If 'X' won, and Player 1's piece is 'X' then we know Player 1 is the winner, otherwise it must have been Player 2.

If 'X' did *not* win, and Player 1's piece is 'O', then—again—Player 1 is the winner, otherwise it must have been Player 2.

(These tests are done using C#'s ternary operator, which is a very condensed combination of assignment and conditional test: If the condition before the question mark is 'true', then the value before the colon is assigned, otherwise the value *after* after the colon is assigned.)

Next, we run a brief animation that shows the relevant "Player [x] Won!" graphic for a few seconds before hiding it again.

Finally, we update the game statistics—stored in **gInfo**—by telling the class who won, then switch the game mode back to the main GUI, ready to set up a new game.

After these functions, we have three very similar animation functions that are run as *coroutines*—i.e. they run independently of the main workflow. Here's the first, which displays the "Player 1 Wins" graphic:

```
		private IEnumerator DoPlayer1WinsAnim ()
		{
				suspendInteractions = true;
				txt_Plr1Wins.gameObject.SetActive (true);
				yield return new WaitForSeconds (3.0f);
				txt_Plr1Wins.gameObject.SetActive (false);
				suspendInteractions = false;
		}
```

In this tutorial, we're using coroutines to show the end-game graphics. The graphic is enabled and shown for about 3 seconds, then hidden again. (I explain the **SetActive()** function later on.)

The **yield** instruction in the code above will start a counter that times-out after three seconds. If we were to run the counter directly in the function, it would 'block' the main Unity workflow and everything would stop dead until the countdown completes. For a turn-based game like Tic-Tac-Two this is less of an issue, but in a real-time game you really don't want everything to freeze while the counter runs.

Note how user interaction is explicitly suspended while the countdown runs: it's not a good idea to allow the player to carry on playing, or click on the RESIGN button, until the end-game sequence has completed.

When using C# with Unity, the **IEnumerator** keyword is required when defining a coroutine. Without it, the coroutine won't work, but you won't get an error message if you omit it, so keep this in mind.

This is very much a C# thing: if you want to program Unity in Javascript instead, the keyword isn't needed. Nor do you need to explicitly tell Javascript that you're starting a coroutine as we saw in the GameWon() function:

```
    StartCoroutine (DoPlayer1WinsAnim ());
```

This, again, is a C# requirement, because C# hides less of the complexity of programming from you than Javascript does. In Javascript, you'd just call your equivalent of DoPlayer1WinsAnim() directly, like any other function. This means the only way to tell if a function is a coroutine in Javascript is to look for the "yield" keyword somewhere within it.

NextTurn()

This function is called after a player has played their move. It flips the current player over, ready for the next turn to be played.

This is also where we check if the new current player is an AI player.

Note, too, the check to see if we're in the game setup GUI mode. This is because Unity effectively runs multiple scripts at the same time, so it's possible that the game setup GUI mode might have been activated before we've entirely finished cleaning up after a game.

CheckAndPlayAI()

This function is called by NextTurn() (above) and triggers the computer player code when needed.

LateUpdate() — the main GUI code

This is a hairy-chested beast of a function that handles *all* the graphical user interface (GUI) features in the game, both during the game setup phase, and in the game itself.

It is not pretty or elegant. It manages a very simple 2D GUI system, but this system really is *basic*. There are no fancy sliders, widgets, scroll bars or the like: everything is a simple on/off toggle switch of some kind.

Unlike the previous version of this tutorial, I decided against using the built-in Unity GUI systems due to the imminent release of their new system. However, as the new GUI system won't now be available until the Unity 4.6 release, I decided to roll my own using just 2D Sprites—introduced in Unity 4.3—coupled with 2D Box colliders. The next release will use the new GUI system instead.

> All the graphical user interface elements are simple 2D Sprites, each of which is attached to a 2D Box collider.

The function is split in two: one part deals with the initial game setup GUI, which lets you set human and computer players and choose which piece ("X" or "O") each player uses. The second part deals with the in-game GUI, including clicking on the individual board cells.

The heart of the GUI system I wrote involves 'ray-casting'—casting virtual lines 'into' the screen from the point you click on. Unity scans along these virtual lines for *colliders*, which is why I've attached a 2D Box collider to each of the clickable graphical elements in the game. The 2D Sprites themselves are invisible to such virtual rays, so the 2D Box colliders were added to give the rays something to 'collide' with.

So, whenever you click a mouse button, or tap a touch-screen, we cast one of these rays from that point into our Scene and see what the line hits. If it hits a 2D Box collider, we grab that collider's associated GameObject and look at its tag to work out which 2D Sprite it relates to.

Although the function looks big and scary, it really isn't. Some of its size is due to the extensive comments I've included—mainly to remind myself of what's going on. (I have a very poor memory.)

So, here's what it all does:

First, we check if the GUI should be running at all. If we're playing an end-game animation, like "Player 1 Wins!", we don't want the GUI to be enabled, so we just skip everything if this is the case.

Next, we check which GUI mode we're in: if we're in the Play mode, we only want to show the RESIGN button and the information relevant to the current player. If we're in the game set-up mode, we want to show the full user interface instead.

The **state** variables in this project, and the functions that work with them, together form one of the game's Finite State Machines. An FSM can be very simple, like the one in this tutorial, or very complex, with dozens, or even hundreds, of states.

Every game has at least one FSM.

Note how, in both cases, we simply read the game state data from **gInfo** and use that to dictate what is shown on screen. Having all this game state info in one place makes this part very easy: just show or hide each user interface element according to the game's state settings.

Note, too, that we do all this drawing / hiding stuff *separately* from the "has the user clicked on something" tests. The old "Immediate Mode" Unity GUI system works very differently: you literally decide whether to show or hide an element right at the point where you find out if the user has clicked on it, so all the showing / hiding code is interwoven through the rest of the GUI code, which can be difficult to follow.

Finally, we come to the bit that actually checks if something has been clicked on…

The test that checks for a ‘click’ or ‘tap’ action actually looks to see if we’ve received a “Fire 1” button event. This corresponds to either a left mouse button click event, a single tap on (say) a tablet or smartphone screen, or a game controller’s primary fire button, depending on the platform we’re running on.

> An **event**, in this context, is just fancy talk for functions calling other functions —often in different programs.
>
> The key here is the context: the function calls are made when a particular **event** has occurred that you want to be informed of. E.g. a Windows or OS X application will usually want to know if its user has resized its window as it‘ll need to refresh its contents.
>
> Unity relies heavily on such events too, with the “LateUpdate” event being fired off after each frame update cycle. Unity also sends an “Update” event after each frame, but before the “LateUpdate” event, so the latter can do any necessary clean-up. (For this project, it doesn’t matter which is used.)

Button Up vs. Button Down

As the comments above this section point out, we only care if the “ButtonUp” event for the “Fire1” button has been triggered, otherwise we’ll be casting rays as often as 50-60 times a second as long as the button is pressed or the finger is touching the screen. That would cause the toggles to flip back and forth ridiculously quickly until the user lets go of the button or removes their finger.

Ideally, I'd have liked to include a ‘highlighted’ image for each GUI element while the mouse is hovering over it. However, this would only work on traditional keyboard and mouse-pointer systems (known as “WIMP” GUIs), like Windows, OS X and others.

On touch-screen devices, like tablets and smartphones, there is no mouse pointer to ‘hover’ over anything, so such code would be wasted on those. As I will need to rip out this GUI entirely for the next release, it makes little sense to spend too much time on it. A 'click' sound might be worth adding, but this is simple enough to be left as an exercise for the reader. (Hey, it's *good* to be the ~~king~~ author!)

The party starts here:

```
if (Input.GetButtonUp ("Fire1")) {
     Ray ray = Camera.main.ScreenPointToRay (Input.mousePosition);
     RaycastHit2D hit = Physics2D.GetRayIntersection (ray);
     if (hit) {
          string t = hit.collider.tag;

          Player p1 = gInfo.GetPlayer (GameInfo._PLAYER1);
          Player p2 = gInfo.GetPlayer (GameInfo._PLAYER2);
```

This can be translated into Plain English™ as follows:

Has the "Fire1" button been released? If so, get the mouse pointer's location (or screen tap location) and cast a virtual ray—a line—'into' the screen from that point. If it hits a ***collider*** *object, grab that collider's* ***tag****.*

The last two lines in that fragment cache the two player data structures from **gInfo**, mainly to save on typing.

The tags for each of the GUI elements were set by hand in the Unity Editor's Inspector.

This next bit checks if we're playing a game—the exclamation mark should be read as 'not', so: "if we are NOT in the inGUI game state". If so, we check if the RESIGN button has been clicked on. This is the only button that's active while playing a game:

```
if (!(gInfo.state == GameInfo.GameStates.inGUI)) {
     if (t == "btn_RESIGN") {
          gInfo.state = GameInfo.GameStates.inGUI;
```

As we're playing a game, we also need to handle clicks on the game board here:

```
}  else if (t == "gridCell") {
```

The next line is a check to ensure the current player is actually a human. If not, any attempts to click on the board are ignored:

```
if (gInfo.GetCurrentPlayer ().PlayerIs == beings.Human) {
```

As clicking on a cell that's already got a piece in it shouldn't do anything, we then need to ensure the player has clicked on an *empty* cell:

```
GridCellScript gcScript =
     hit.collider.gameObject.GetComponent<GridCellScript> ();
if (gcScript.state == GridCellScript.CellStates.empty) {
```

Note how we actually dive right into the GridCellScript itself to get at its state. This is an example of "close coupling", which we looked at earlier.

Now that we know that the cell is empty and can be played, we work out which piece the current player is playing and tell the GridCellScript—there's one attached to each of the nine squares—which piece it should display:

```
if (gInfo.GetCurrentPlayer ().PlayingAs == pieces.O) {
     gcScript.state = GridCellScript.CellStates.O;

}  else
     gcScript.state = GridCellScript.CellStates.X;
```

The move having now been played, we can swap the players over:

```
NextTurn ();
```

Beyond this, we have the code that deals with the game setup user interface, which is what we see when we click Play in Unity and start the game.

For the sake of clarity, I've modified the formatting of some of the code fragments to make them a little easier to follow.

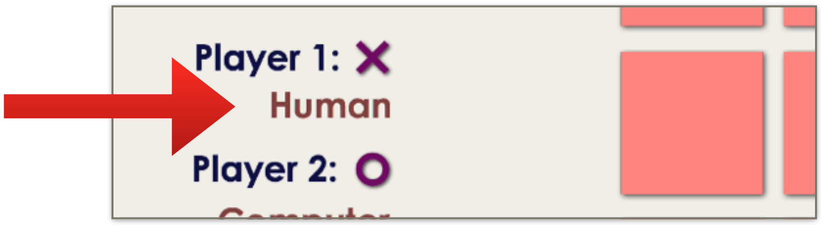
Although this section looks big and scary at first glance, it's actually mostly the same lines over and over, with only the tag check changed. (Remember, the variable **t** contains the **tag** value for the collider we hit with our ray cast earlier on).

Let's look at one of the tests:

```
    if (t == "player1_Human") {

    gInfo.SetPlayer (GameInfo._PLAYER1, beings.AI,
         p1.PlayerName, p1.PlayingAs);
    }
```

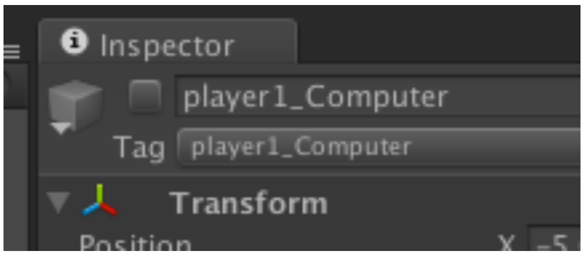
This tag is linked to the following 2D Sprite and its collider—the one with the big red arrow pointing at it:

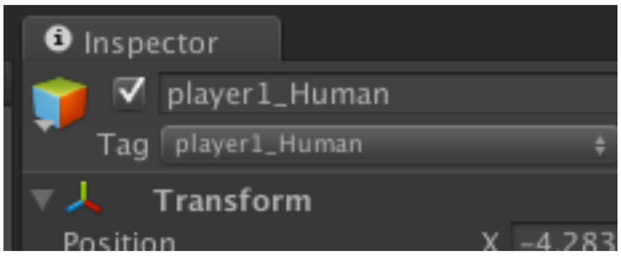


Clicking on this toggles between "Human" and "Computer" on the screen. The code simply swaps the Player information back and forth between **beings.Human** and **beings.AI** accordingly.

In the next LateUpdate cycle, we redraw the GUI based on this changed Player state, so we see "Computer" displayed instead of "Human" if the player has been set to **beings.AI**, and vice-versa.

If you look closely at the Scene view in Unity, you'll see that each toggle has two 2D Sprites (and their 2D Box colliders) superimposed on each other within the scene. This was done entirely by eye. I then went and disabled most of the 2D Sprites that will be 'off' by default. This was done by clearing the "Active" checkbox next to the GameObject's name in the Inspector. The images below show this part of the Inspector for both the **player1_Computer** and **player1_Human** sprites:





These are switched by our code using the **SetActive()** function, in the first part of the GUI code in LateUpdate():

```
player1_Computer.gameObject.SetActive (p.PlayerIs == beings.AI);

player1_Human.gameObject.SetActive (p.PlayerIs == beings.Human);
```

This function has the same effect as ticking / clearing that checkbox in the Inspector. Clearing the checkbox hides the sprite from view.

The big "Player 1 Wins!", "Player 2 Wins!" and "It's A Draw!" graphics are visible in the Scene because these were literally the last thing I added. I needed to line them up by eye not only to the centre of the board, but also to each other, so having them all visible made this easier.

These are explicitly hidden in the GUIHandler script's **Start()** function.

Many purists who don't have a social life will tell you that "close coupling" is a Very Bad Thing™, to which I reply with "It's a simple Tic-Tac-Toe game, not a major banking application; get a sense of proportion!"

Use the right tool for the job. If that means some bearded university professor with a PhD in Applied Pedantry is offended, so be it.

Most of the GUI code in this section follows the pattern seen above, simply toggling between two settings: Human / Computer, X / O, and so on.

Finally, we check if the PLAY button has been clicked on. If so, it's time to start the game. This means hiding the Player 2 info and only showing the "Player 1: [Piece]" info, without the "Human" or "Computer" bit.

Once that's done, we clear the board ready for the new game, set the correct states for a new game in **gInfo**, and check if this first turn is to be played by the computer. (If it is, we tell it to play.)

And that's it for the **GUIHandler** script. Seven curly braces later, we hit the end of the file!

Next, we'll look at the BoardManager script and its close relatives…

The BoardManager Script

This script performs multiple duties: it handles most of the game's logic—i.e. checking for wins and draws, but it also supports the computer—i.e. the Artificial Intelligence, or "AI"—player. It does this by providing a score for each square in the 3 x 3 grid after each turn. This score is used by the AIPlayer script to work out which is the best square to play.

The AI code and game logic code were developed alongside each other, so the functions aren't neatly separated in BoardManager, but I'll go over the AI-related functions in more detail shortly.

Whither Tom, Dick and Harry?

Designing an AI for a game like Tic-Tac-Toe, or, indeed, any other game, is an entire book in itself. Tic-Tac-Toe is a common starting point for such books as it's such a simple game. In fact, it's so simple that most kids outgrow it pretty quickly once they've learned the winning tactics.

I'm therefore not going to explain AI programming in huge detail, but I will explain how I've built the one for Tic-Tac-Two as it isn't quite the same as in the original project.

The most obvious difference is that there's only one AI, rather than the three ("Tom", "Dick" and "Harry") of the original. This is because the original *also* only had one AI too, despite appearances: the other two characters simply added a random element—one just played a completely random move each turn, the other played a random move roughly 50% of the time.

So only the 'proper' AI has been retained in this release. I may add a difficulty setting in the next version.

The Illusion of Intelligence

Artificial Intelligence functions for turn-based board games share some common elements. The least known is also the most interesting: AI is essentially about creating the *illusion* of "intelligence".

You may have heard about the super-computers who have beaten human chess grand masters. These computers are not, in fact, intelligent in any sense we humans would recognise. They're idiot-savants at best: they've been taught—programmed—to perform a very specific task, and nothing more. A machine that can play chess very, very well is not going to have a clue how to write a novel.

IBM's "Watson" computer is basically a machine for remembering chess openings and endgames, and for crunching through thousands of possible moves a second to work out which is the best one to play. By that, I don't mean it only looks one move ahead: the computer program will literally play *both* sides of the game for dozens of turns to see if the originating move is any good. If it isn't, it discards it, picks another possible move and studies the potential responses to that for a bit… and so on.

The result is a vast branching 'tree' of possible moves, responses and counter-moves, from which the computer program then picks one that best meets its requirements.

Human grand master chess players do this to some extent as well, but to nowhere near the same degree: it would take them hours—days, even—to play each move if they tried.

Humans therefore apply different *heuristics.*

Heuristics

When playing a board game like Tic-Tac-Toe (or chess), we apply a set of rules to each square and use it to decide whether that square should be played or not.

We normally do this by giving each square a rank of some kind, usually by looking at how advantageous it may be to us—or disadvantageous to our opponent—and rating it accordingly.

I use the word "cell" and "square" interchangeably in this project. Each of the game board's nine squares is referred to as a 'cell' in the code itself. Mainly because it's quicker to type.

The approach we use to decide on that ranking is called a **heuristic**. A heuristic is the process we apply to decide what move to play. There is no single definition of what a heuristic for a particular game would be as the term refers to the high-level set of rules and tests we want the computer's AI code to apply.

Now, an AI has to calculate an actual rating for each playable square on the board one-by-one. Humans rarely do this: we'll look for *patterns*, because pattern-matching is what humans do best. We can spot a row of two identical pieces on a 3 x 3 grid at a glance. A computer has to be programmed to inspect each square on the board to see if such a situation exists. It can't "see" the board visually the way we do—as the Programming Primer document explains, *everything is numbers* as far as the computer is concerned.

For example, in Tic-Tac-Toe, the most common pattern to look for is a row, column or diagonal containing two of our own pieces, with an empty square available that will let us complete the triplet and win.

We know that if we play a third piece on that row, we'll win.

This trumps a similar pair of enemy pieces, because our win makes their potential win moot: they'll have lost before they get to play anyway.

Similarly, Tic-Tac-Toe has another 'golden rule' we can add to our heuristic: always start by playing the centre square. Doing so gives you a piece on both diagonals, a vertical column, and a horizontal row. No other square gives you that from the start, so there is a tactical advantage to playing this square as soon as possible.

In fact, that advantage can be so strong that I've actually decided to make the computer player a little *less* likely to play it: my AI code tosses a virtual coin and, only if that coin is 'heads' does that centre square get given a high ranking.

Why?

Because the number one purpose of a game is to be *fun*. A computer player that is effectively impossible to beat when it plays first is no fun at all for most players. Increasing the score for the centre square is therefore only done once, at the start of play, and it's the only time random chance is used by the AI player.

> The computer processor is never actually aware it's playing Tic-Tac-Toe. It's a machine for moving, processing and comparing numbers. It is we programmers who must create that spark of intelligence through the software we write.

You can add the code to make that centre square more likely to be chosen by the AI player pretty easily.

So, without further ado, let's look at the script in more detail:

Data structures

BoardManager defines a number of **public const**ant values. These cannot be changed by the program code and serve mainly to make that code easier to understand.

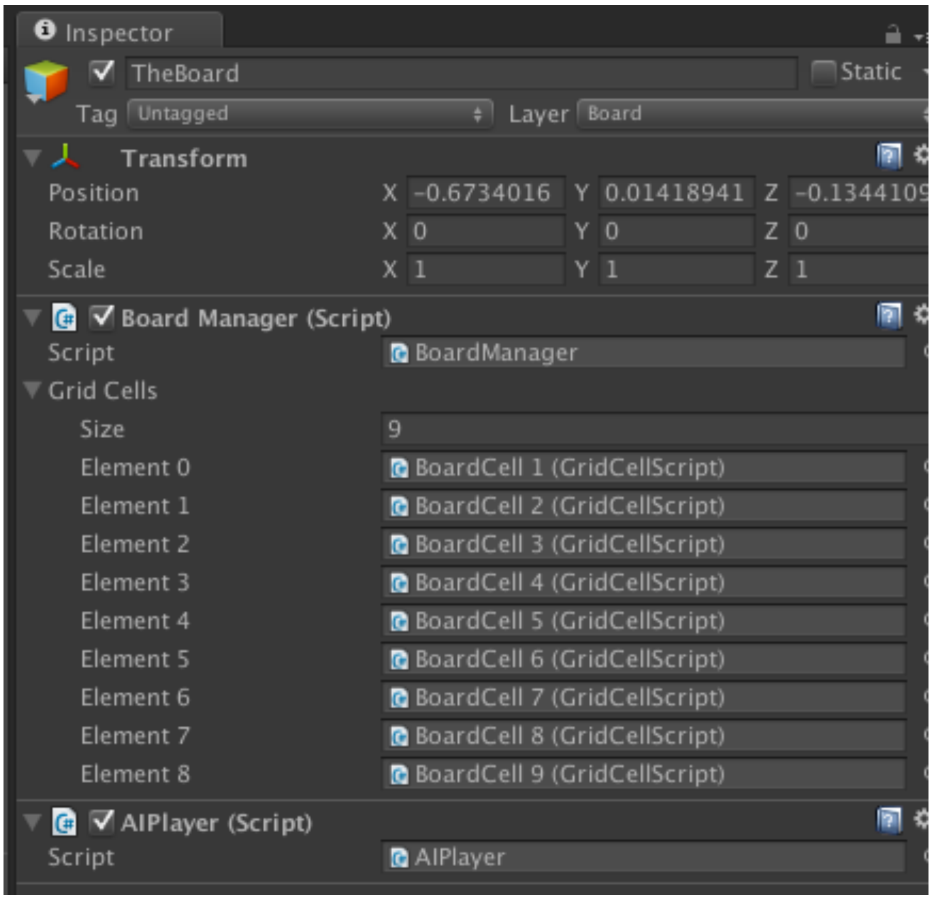
It is considered bad form to use 'magic' numbers in your code as it can be difficult to understand, or remember, why a specific number was chosen. Constants like these let you make those numbers "self-documenting". In this context, it means the numbers are replaced by descriptive labels explaining their purpose. When the code is compiled, the label is simply replaced by the number itself.

The constants defining the game board's size are self-explanatory, but RANDOMFACTOR is perhaps a little less so. We'll come back to this one shortly in the **InitScoreboard()** function.

The final three constants define integer values representing a board square's three possible *states*: Empty, showing an 'O', or showing an 'X'.

(Note that all these constants are defined as **public**, making them visible to the other scripts in the project too.)

Next, we have an array of GridCellScripts called *gridCells*, which represents our game board's current state. Each entry needs to point to one of the squares on the 3 x 3 game board, so I set these up by hand in the Inspector:



TheBoard
Tag Untagged Layer Board

	X	Y	Z
Position	-0.6734016	0.01418941	-0.134410
Rotation	0	0	0
Scale	1	1	1

Board Manager (Script)

Script	BoardManager
Grid Cells	
Size	9
Element 0	BoardCell 1 (GridCellScript)
Element 1	BoardCell 2 (GridCellScript)
Element 2	BoardCell 3 (GridCellScript)
Element 3	BoardCell 4 (GridCellScript)
Element 4	BoardCell 5 (GridCellScript)
Element 5	BoardCell 6 (GridCellScript)
Element 6	BoardCell 7 (GridCellScript)
Element 7	BoardCell 8 (GridCellScript)
Element 8	BoardCell 9 (GridCellScript)

AIPlayer (Script)

Script AIPlayer

As you can see from the above, the BoardManager script is attached to a GameObject in the scene called **TheBoard**, which contains all the individual squares as ‘child’ objects. Also attached here is the AIPlayer script, which we’ll look at later.

Note the **Grid Cells** entry. This appears because of the gridCells definition:

```
    public GridCellScript[] gridCells;
```

In Unity, any variable declared as ‘public’ will be shown in the Inspector. Arrays like gridCells appear like the example above, except, to begin with, they only have one entry. I typed “9” into the “Size” box to create nine slots, then dragged each individual “BoardCell [number]” element under the **TheBoard** GameObject in the order shown.

This ensures I have the board squares ordered correctly.

This array points directly to the individual GridCellScript component of each board cell prefab. We use this script to get at the square’s current state, and to change it if necessary.

Next up is a variable that stores a pointer to the GUIHandler script. We need to call functions inside that script from this one, so this pointer makes it possible to do that.

An **enum**erated data type appears next. This defines a *list of constants*. In this case, the constants are used to represent compass directions—North, Northeast, East, etc. (I added a “D_” prefix to stop MonoDevelop trying to autosuggest all sorts of pointless alternatives for the single-letter directions.)

pieceToPlayNext is used to tell us whose turn it is. In fact, it only tells us which piece—X or O—is yet to play its move. BoardManager knows nothing at all about which player is playing as which piece. In fact, BoardManager knows nothing about the players at all. It cares only for the X and O pieces.

Finally, we have an integer array called **scoreBoard**. This contains our AI scores—the rankings we discussed earlier. The values stored in this array are the three constants we saw earlier: EMPTYCELL, O_CELL and X_CELL.

At last, we come to the functions!

Start()

This is the entry point function for all Unity scripts. It's where I do all the initialisation of the BoardManager script's data structures.

First, I grab a pointer to the **GUIHandler** script. There should be one somewhere in the scene. If there isn't, I logged an error to the debug console.

Next, I initialise the **scoreBoard** array. Note the use of the BOARDDIMENSIONS constant to define how big it should be. The **new** instruction is used to tell the computer to allocate enough RAM memory to store the nine ints needed to hold the scoreboard data.

An array is a class in its own right. If I didn't use C#'s **new** instruction, the **scoreBoard** variable would be 'null'—i.e. containing no data at all. Merely defining a complex data structure like this is therefore not enough on its own: it must also be *instantiated* by having RAM allocated to it.

Finally, the function I'm about to describe is called to initialise the scoreboard with the starting scores for each square…

InitScoreboard()

This function resets the scores in the **scoreBoard** array, ready for a new game.

It first sets a value for **pieceToPlayNext**. As the script does not yet know at this stage which piece is playing first, I set it to EMPTYCELL (i.e. zero).

Next, the score board is reset by looping through each square on the board and setting its score to zero. The only exception is for square number four—the centre square—which is given a higher value, but only 50% of the time. (Note that this is why I'm using a **for** loop, rather than a **foreach** loop: I need a counter to keep track of which square we're on.

This is our first use of the RANDOMFACTOR constant:

Random.value looks like a variable, but it isn't; it's actually a special function called a "getter" pretending to be a variable. (You can find out more about these in the Programming Primer document.) What this function does is give us a random floating point value between 0 and 1. I'm using this to simulate a virtual coin-toss: if the random number is greater than or equal to RANDOMFACTOR, it's a 'heads'; if not, it's a 'tails'. If I get 'heads', the score for this square is set to 5, otherwise, it's set to zero like all the others.

This function is also used by the next one…

ClearBoard()

This time, a **foreach** loop is used to step through each of the gridCell entries and set their state to *empty*.

Finally, **InitScoreboard()** is called to zap the grid squares' corresponding scores as well.

The Programming Primer document explains the differences between the **for** and **foreach** constructs in more detail.

Now we get to an important utility function:

GetBoardAsInts()

This one performs two functions.

The first is—as the function's name suggests—to return an array of ints containing the game board's status. This array avoids the need to expose the **gridCells** array and all the board cell scripts as well to other scripts. Instead, we only ever pass this array of basic integers. This is where three of our earlier integer constants are used: EMPTYCELL, O_CELL and X_CELL. It's also why those constants are defined as public, rather than private.

The second feature is less obvious, but this proved the best place to put it: it's where we calculate the new value for **pieceToPlayNext**.

The calculation is a little odd because the sequence of function calls that precedes a new turn means this function is called *before* the GUIHandler's own **NextTurn()** function. So the values we're basing our calculations on are one turn out of step at this point.

Next, we move on to a function that will be called by the AIPlayer script…

AIPlayMoveAt()

The AIPlayer script calls this function when it has decided which cell to play. This function simply drills down to that cell's own scripts and tells it to update its state accordingly, by either enabling the "X" or "O" graphic over the square.

Finally, we call the **NextTurn()** function to signal the end of the player's turn.

HasGameBeenWon()

In the original Tic-Tac-Tut project, the tests used to answer the question implied by the function's name were long-winded and inefficient. They did the job, but it could be done more quickly. Much more quickly…

The original script used a series of loops to check horizontal, vertical and diagonal lines, testing each cell on the board, one-by-one, and incrementing counters to track how many pieces were in each of the lines.

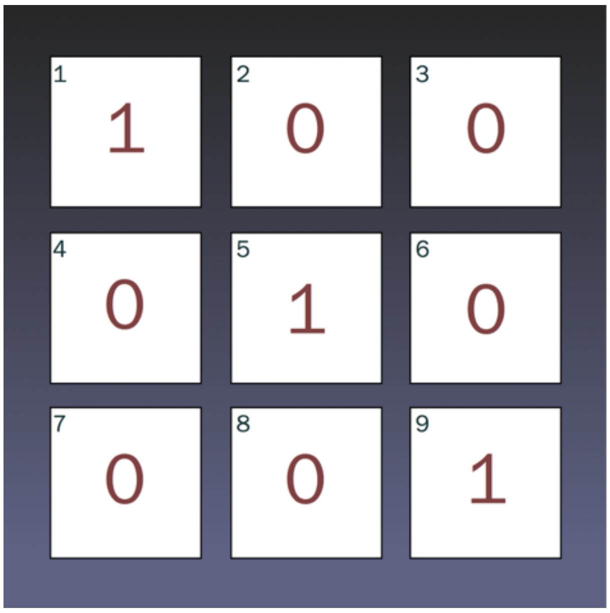
There were three horizontal lines, three vertical lines and two diagonals to test. That's 8 loops and tests in all. This new function works it all out with just *two* very simple loops.

How does it work?

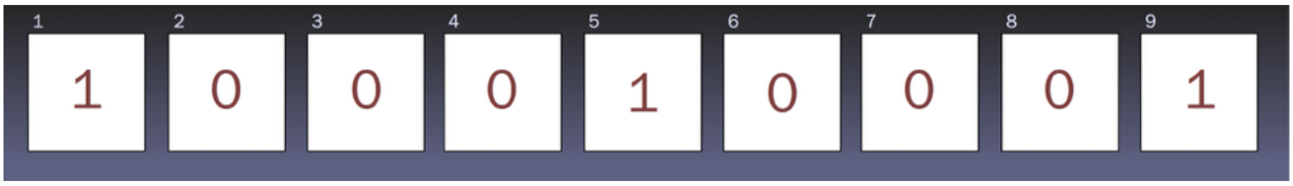
If you look back up the script, you'll see the **GetBoardAsInts()** function. This is used during play to maintain an array of integers that tell us the current state of the game board. To check this board for a win condition, we could just loop through all possible combinations of winning rows, columns and diagonals, looking for one with three identical pieces. If we find one, that piece has won; if no such lines are found, there's no winner yet. But there is another way, and it's entirely built around the fact that computers actually 'think' in numbers…

Supposing we represent the board using a single binary digit—or "bit"—per square? That poses an obvious problem: there are *three* states for each cell, not just two: there's the 'empty' state to consider, as well as the X and O pieces. It turns out this isn't a big deal:

When checking for winning lines, *we don't care about the other piece*. If we define a binary 1 as "X" and binary 0 to mean "anything else", then we need only look for this:



That works fine in binary. Of course, the computer doesn't see that square grid arrangement. All it sees is a single variable with the nine bits at one end of it, like this:



Note that I've numbered each bit to show which square on the grid it corresponds with. When looking at computer memory, bits are actually numbered from right to left, but that's not important for this example.

We can't fit 9 bits into an 8-bit byte, and RAM storage is arranged in blocks of bytes, so that means a minimum of 16 bits of storage, of which only the right-most 9 bits are of interest to us; the remainder are unused. (All the unused bits are set to 0.)

In C#, an **int**eger variable needs four bytes of RAM for storage—i.e. it holds 32 bits, so there are 23 bits to the left of the bit labelled "1" in the diagram above that we're ignoring.

I therefore defined two **uint** variables called **OCells** and **XCells**.

I then stored the binary representation of the board for each piece in its respective variable.

The **uint** keyword is short for **unsigned integer**. This merely tells the compiler that the value we're storing here will never be a negative number. Although it's not strictly necessary to use this data type, it does reduce the potential for some particularly obscure bugs. It also means we can play around with all of the bits.

Which is why I'm using the **uint** data type.

Next, I create a counter. This will be used to determine which bit in each of OCells and XCells I'm going to store the a 1 in if I find an associated piece on that square. Note that the counter starts at '8' and counts down, as I want to store the board squares in the OCells and XCells in order from left to right: top row (left to right), middle row, then bottom row.

Tic-Tac-Toe's board is symmetrical, so it doesn't actually matter whether the counter starts from 8 and counts down, or starts at 0 and counts up as all complete rows, columns and diagonals will be spotted either way.

Now we come to the first loop, which looks at each cell in the **board** array in turn. If the integer value for that cell is *O_CELL*, I set the appropriate bit in

OCells. I do the same test for the 'X' pieces as well and set XCells appropriately.

This line needs some explanation:

```
        OCells |= (1u << i);
```

The vertical bar before the equals sign represents a **bitwise OR** operation. This comes from Boolean Logic, which is concerned with binary—as in "true/false", or "yes/no"—operations.

The Programming Primer document that accompanies this tutorial explains more about all this in the **Working with Digits** chapter.

Now, near the top of this function is this line:

```
uint [] winStates = { 448u, 56u, 7u, 292u, 146u, 73u, 273u, 84u };
```

This is a list—an array, in fact—of what appear, at first glance, to be random numbers. Far from it: each one represents a specific "win" state to look for. Each number represents a three-in-a-row state. The first number is the decimal representation of the binary number **111 000 000**.

These numbers should be thought of as representing a 3 x 3 grid, so that first number represents this:

111
000
000

If we apply this number to the *OCells* variable using a **bitwise AND** operation, and we get the same number back, that means the top row of the game board contains three 'O' cells in a row. I.e: it's a win for 'O'.

The same happens if we apply the same test to *XCells*.

So we end up with this:

```
    foreach (uint state in winStates) {
            if ((OCells & state) == state) {
                    winner = O_CELL;
            }
            if ((XCells & state) == state) {
                    winner = X_CELL;
            }
    }
```

All we do here is grab each *state* entry in the *winStates* array and check each in turn against OCells and XCells, always looking for the AND operation to return the same number as *state*.

If you're not clear on why this works, do read the *Programming Primer* document, where each bitwise operator is explained in more detail.

If one of these tests is positive, we set the *winner* variable accordingly.

It's important to note that this algorithm works because we are checking for a winner after each turn, and we know that it's a turn-based game, so it's impossible for both players to play a winning move at the same time.

It's worth comparing the equivalent function in the old version of this tutorial with this one: this version is a *lot* shorter. Understanding how computers think really does help.

RefreshScores()

This function is called after each turn and does two jobs. It first checks if the game has been won, or drawn.

If the game has been won or drawn, this function also calls the appropriate functions in GUIHandler to start the game state transition and show the "Game Won" or "Game Drawn" messages.

If the game is not yet over, the function updates the scoreboard array for the game board, calculating the score for each remaining playable cell. For this, we call **GetScoreFor()**.

GetBestMove()

GetBestMove(), is called by the **AIPlayer** script to find out what the best move is to play. The function looks through the board cells and keeps track of the best score it has found so far. The cell with the highest—i.e. best—score is the one the AI player chooses.

There's also a bit of virtual coin-tossing to make sure it doesn't always pick the first cell if there are multiple cells with the same best score.

Working out what the best move *is*, however, is the job of the first three functions: **GetScoreFor(), CalcScore()**, and **SetCellsCountersFor()**.

The scoring process is broken up into separate functions to make it easier to follow the logic. I have a lousy memory, so breaking operations up into smaller, manageable, easy to follow chunks just makes life easier.

Ideally, a function should do one thing, and one thing only. There are a few functions where I bend this rule a little—such as RefreshScores()—but I don't do it often.

GetScoreFor()

This is the top-level AI scoring function. It calculates the score for the specified cell number.

First, we check if the cell is playable—i.e. if it's empty.

Next, we check if this is the centre cell. If it is, and it hasn't been played yet, we nudge the AI player into playing it. (Note that we also use a random number check to avoid having the AI player *always* play this on their first turn. This deliberately makes the AI a little more fallible. A game is, after all, supposed to be fun.)

At this point, we know the cell is playable, so it's time to work out its score.

For each horizontal, vertical and diagonal row:

1. Count the number of friendly and enemy pieces;
    1.1. If there are two friendly pieces in a row, and we can play a third piece to complete it, we give a high score to that playable cell;
    1.2. If there are two enemy pieces in a row, and we can play a third piece to prevent our opponent winning with three in a row, we give *that* cell a high score.
2. Otherwise, apply a score based on the number of friendly / enemy cells in the row.

For each cell, then, we need to scan along four rows: vertical, horizontal, and the two diagonals (top-left to bottom-right, and top-right to bottom-left).

GetScoreFor() uses **SetCellsCountersFor()** to count the number of pieces in each vertical, horizontal and diagonal row. For each of the four rows we want to check, we call SetCellsCountersFor() with:

* the piece ID (so we know which piece is "friendly");
* the cell ID (0-8, corresponding to the cell on the board);
* our two cell-counter variables;
* our game board;
* the direction we want to scan in, using an **enum**eration in the form of compass points.

The ref keyword

There are two ways to pass information through a function call: either by copying each parameter, or by passing a reference to it. The first option is the default in C#.

Most of the time, this is what we want. Sometimes, however, we want the called function to make changes to a variable our calling function is using. We can either do this by making the called function **return** that information, or we can give that called function *explicit permission to change our calling function's data directly*. This is what the **ref** keyword does.

When **GetScoreFor()** calls **SetCellsCountersFor()**, each call uses the **ref** keyword before some of the parameters passed to the latter function.

We want to update two counters simultaneously, but a function can only return one datatype, so we'd have to use a custom **struct**ure datatype, such as a **Vector2**, to pass the information around. This is a bit of a hassle for such a simple function, and it makes the logic a little harder to follow too.

Using the **ref** keyword means we give the called function permission to change our calling function's own variables directly, so when **SetCellsCountersFor()** changes that variable, it's actually changing the copy inside **GetScoreFor()**. Thus we don't need to return the value explicitly. (This also has the advantage that we can, if needed, use the function's **return** keyword to return something else, such as an error / success status.)

We also pass the game board's state as a **ref**erence too, though this is mainly to avoid having to copy the array each time. Although such copying doesn't really impact performance, it can potentially cause problems when running on low-end smartphones, which often have very limited RAM for running applications in. We should never assume that we have lots of resources available, nor should we assume that ours will be the only program running on a device.

The actual score calculation is done in **CalcScore()**, which we'll come to shortly. First, let's look more closely at the cell counting…

SetCellsCountersFor()

This is an unusual function in that its logic requires it to call *itself*. In programming jargon, this is known as recursion. This is why we pass a direction into the function as a parameter, but we'll get to this in a moment.

First, as we know which piece type—X or O—is the friendly piece, we use this knowledge to work out what the enemy ('foe') piece type is. We use the ternary operator once again for this: it can be read as: *"If **friendPiece** is X_CELL, set to O_CELL, else set to X_CELL."*

Next, we have a sanity-check to make sure we're not about to look at an out-of-bounds cell in the **board** array. Such checks can really help with debugging. It's also a simple enough test that it won't affect performance.

Next, we check the requested cell and see if there's a piece on it. If so, we update the relevant counter.

This is the "Set Cells Counter…" part of the function done with. Now we get to the recursion part.

The big **switch** / **case** list just checks the **dir** variable and works out which cell needs to be scanned next. And then calls **SetCellsCounterFor()** on that cell.

Note that each of the **case** entries also includes a limit check to make sure we don't end up going off the board and out of bounds. For our traditional 3 x 3 cell Tic-Tac-Toe board, the tests aren't that complicated and should be easy enough to follow.

Note that the diagonal directions are only tested if scanning in that direction makes sense. On a traditional Tic-Tac-Toe board, both these diagonals *must* pass through the centre cell. Therefore, there's no point in testing the diagonals for the bottom-centre cell.

In theory, this function could be expanded to handle a Tic-Tac-Toe board of *any* size, which is why I wrote it this way.

Those bounds checks are the reason the function doesn't keep calling itself in an infinite loop: after one or two calls, we'll hit the edge of the board and the logic in the **SetCellsCountersFor()** function will fall through to the **return** keyword right at the end.

This passes control back to the point immediately after the function was called. If we scan twice in a particular direction, that means we land on the **break** instruction immediately after the recursive call. So we fall through to the **return** keyword once again. This continues until we end up back in **GetScoreFor()**.

The stack

All this is possible thanks to a temporary storage queue known as a stack.

When a function calls another, all its variables are first *pushed* onto a dedicated temporary storage area in the computer's RAM. This area behaves much like a stack of paper, where the last bit of data put on top is the first bit of data taken off, hence its name: *the stack*.

The next step is to store the current position in the function—known as the *program counter*—on that stack too. This way, the computer knows how to get back to where it was.

Finally, the called function is executed by simply setting the *program counter* to the address of its first instruction. (We're back to those little boxes of numbers described in the accompanying Programming Primer.)

When the called function *returns control* to the calling function, the computer *pops* that stored data back off the stack. This includes the original location for the program counter, so the computer knows where to continue fetching its instructions from.

CalcScore()

This is the final function involved in the AI scoring.

It uses the number of friendly and enemy pieces found in the horizontal, vertical or diagonal row to return a score.

Time for me to put my hand up here and be honest: the numbers really are mostly plucked out of thin air. What matters most is the order of the tests, and making sure that if two 'friend' pieces are found in the same row as this empty cell we're testing, we give a very high score to make sure the AI player chooses to play it and win the game.

The basic logic goes like this:

1. Do we have one of our own pieces on this row? If so, bump the score up a little. We won't win on this turn by playing this cell, but it's better to build up an existing row if we have already started it.
2. Similarly, if this row has an enemy cell, we should increase the score a little to encourage a blocking move if (1) doesn't override it.
3. We have more than one (i.e. two) friendly pieces on this row, which means we can complete it and win the game! Give it a very, very high score!
4. Failing (3), we should try to stop our opponent winning the game themselves by blocking their two-piece line.

WasWinnerX()

This is a simple utility function used by GUIHandler to work out which player has actually won.

The reason it's needed is because BoardManager has no idea which player is actually playing as which piece—X or O—so GUIHandler needs to know which piece won before it can make that calculation itself.

And that's it for BoardManager. Just some smaller scripts left to go…

The AIPlayer Script

This is one of the shortest scripts in the project. The reason being that most of the heavy lifting for the computer player is built into the **BoardManager** script.

Start()

This function simply gets a reference to the **BoardManager** script, so we can call some of its functions.

PlayAIMove()

As the function's name suggests, this is where the computer player's move is made. It actually just calls BoardManager's **GetBestMove()** function and passes its result to **DoMove()**, the next function in the script.

Note that **DoMove()** is a **coroutine**, which means it is run in its own separate process.

In C#, coroutines need to be called via **StartCoroutine()**. In Javascript, this isn't necessary and such functions are called like any other.

DoMove()

This function pretends to think for about 1.5 seconds, before playing the move.

It's the call to the **WaitForSeconds()** timer function that forces us to make this a coroutine as we can't call this function directly without bringing the entire Unity engine to a juddering halt while it waits for the timer to complete.

The reasons for this delay are twofold: humans have a minimum response time and if the computer player played its move instantly, it would appear to a human player that the computer had played their move at the same time as the human player did. This just feels weird.

The second reason is psychological: it gives the illusion that the computer player actually *needs* to think about its move. (In fact, the computer has already chosen which cell it'll play before the timer even starts!)

The result of the delay is to provide an illusion that we're playing a person, rather than a faceless, emotionless machine that can crush our fragile egos if it were allowed to play a 'perfect' game at full speed!

Only once the timer has completed does the function call BoardManager's **AIPlayMoveAt()** function. That function, in turn, hands control to the **GUIHandler** script, which proceeds to show the computer player's piece on the board itself.

The Player Script

This script defines two publicly-accessible **enum**erations and a **class**. These are used to define a *player* entity in our game. Everything we need to know about each player is stored in a **Player** class. As we have two players in our game, we create two *instances* of this class, one for each player, in the **GUIHandler** script.

What is a class?

A class is an combination of a data structure and supporting functions. By combining both data and code in the structure, we can create 'intelligent' data structures that can do all sorts of neat tricks.

There's more on classes and **object orientation** in the accompanying **Programming Primer** ebook.

Unlike most of the other scripts in this project, the Player script defines a standalone class, named **Player**, that does not inherit features from Unity's **Monobehaviour** class.

When a class inherits from Monobehaviour, it combines that class' features with its own, effectively extending it. Monobehaviour includes all the functionality associated with Unity, such as the usual **Update()**, **LateUpdate()**, and **Start()** functions.

As **Player** doesn't inherit from Monobehaviour, it doesn't get those functions either. It's a completely independent class and would work fine in any C# application, whether it ran on Unity or not.

The Player data structure

The Player class defines the following features for a player entity:

* **PlayingAs** – which tells us which piece, 'O' or 'X', the player is playing as;
* **PlayerName** – which stores the name for the player. This feature isn't used in this version of the tutorial, but will be used in the next release;
* **PlayerIs** – which defines whether this player is a human or computer (AI) player;
* **score** – the player's score;
* **won** – how many games the player has won;
* **lost** – how many games the player has lost;
* **drawn** – how many games the player has drawn;
* **totalGames** – the total number of games the player has played so far.

Sir Not Appearing In This Tutorial

Like **PlayerName**, the statistical data – **score**, **won**, **lost**, **drawn**, and **totalGames** – is not used in Tic-Tac-Two either, but will be used in the next version.

Properties: Getting and Setting

Three of the class' variables use **getters** and **setters**, making them what C# defines as **properties** of the class. These are **PlayingAs**, **PlayerName**, and **PlayerIs**. Each of these variables has a hidden **private** variable associated with it that actually stores the information. This private variable is never, ever, accessed by any of the other scripts, because a private variable is invisible to them.

Instead, the scripts access the data via **getters** and **setters**, which are mini-functions that define how each variable is stored and retrieved.

These functions are implied, not explicit: from the perspective of, say, the **GUIHandler** script, a Property like **PlayingAs** appears exactly like any other class' member variable.

When we try to read the data in **PlayingAs**, we're actually calling its **get** function, which simply returns either **pieces.X** or **pieces.O**, depending on whether its associated private variable, **_isX**, is set to *true* or *false*. (It's set to *false* by default.)

When we try to set the data for **PlayingAs**, its associated **set** function is used to set the private **_isX** variable accordingly.

Note that **value** is an automatically generated variable that represents the value we're trying to set. This is why you don't see it explicitly declared anywhere.

Properties are discussed in more detail in the accompanying **Programming Primer** document.

The GameInfo Script

Like the **Player** script we've just looked at, this script also defines a standalone class.

The **GameInfo** class is used to contain the game state. I.e. it keeps track of the two players, whether we're in the main GUI, or are currently in a game, and so on.

Unlike the **Player** script, **GameInfo** is a bit more involved…

Constants and Enumerations

First, we define some publicly-accessible constants:

_PLAYER1 and _PLAYER2 are self-explanatory: we use these instead of hard-coding "0" or "1", making the program's logic a little easier to follow.

> **Note:** I like to use a leading underscore symbol for constants and private variables. To distinguish the two, I always write constants entirely in capital letters.

Next, we define a publicly-accessible **enum**eration, called **GameStates**. This defines the three states the game can be in: *inGUI*, *playing_plr1*, and *playing_plr2*. These tell us, respectively, that we're currently in the GUI (i.e. setting up a game), or waiting for either Player 1 or Player 2 to play their turn.

With these defined, we now move onto the data structure itself. First comes the **state** variable (which is a **Property**, hence the **get** and **set** functions. This stores the game's current state.

Next, we define the two players. These structures were defined in the **Player** scripts we saw earlier, so I won't go into detail here. Note the use of the **new** instruction to create each of these. Classes need to have memory explicitly set aside for their storage, and this is what the **new** instruction does. This is called *object instantiation*. It creates an *instance* of an object in memory. The class definition we see in the script is like a plan for a building: it isn't until we *instantiate* it—i.e. construct the building based on that plan—that we've created an object of that class.

The **new** instruction calls what's known as a *constructor* function within the class. You can have more than one of these, but we have just the one in this example: **GameInfo()**. Note that it has the same name as the class, which is a requirement for a class constructor.

Constructors allow us to set the data in the class with default values. More complex constructors even let us pass values in directly from the **new** instruction, so we can set specific values for some of the variables in the class.

GameInfo()

This constructor function sets default values for all the data structures in the class. This mostly consists of setting up each of the players with default values, so Player 1 is set as a Human, with a name of "Player One", and playing as "O".

All the statistical variables—score, won, lost, etc.—are also reset here. Those variables aren't actually used in this version of the tutorial, but will be used in the next release.

GetPlayer()

This utility function simply returns the requested player's **Player** object. This is used by other scripts to make them easier to read—and to type!

StartPlaying()

Called when a new game is started. Player 1 always plays first, so we just change the **state** variable accordingly.

GetCurrentPlayer()

A utility function that returns the **Player** object for the current player only. It returns **null** if a game is not currently in progress.

GetPlayerPiece()

Returns the piece being played by the selected player. This function is used to work out who actually won a game, as the **BoardManager** script never sees the Player data itself.

SetPlayer()

A utility function used to set the player data for a specified player. Used by the GUIHandler script's GUI section to update the player's state when it is changed by the user. (E.g. when the player is switched from Human to Computer.)

UpdatePlayerScore()

Another utility function that lets us update the score for the specified player. Used by the next function, **GameWon()**.

GameWon()

When a game has been won, we pass the winning player to this function, which then proceeds to update the scores and other statistical variables. Although we never see these in this release, we will see them in the next one as it will have a more complete GUI.

Finally…

GameDrawn()

This function is very similar to **GameWon()**, and is called if a game has been drawn. Unlike **GameWon()**, however, we don't pass any parameters as the same changes are made to each player.

GridCellScript

This script works very closely with **XOScript** to handle the game board display. There is one **GridCellScript** attached to each board cell prefab on the game board.

Each board cell prefab contains:

1. A blank board cell 2D Sprite;
   1.1. An instance of **GridCellScript** attached to the above;
2. A large "X" 2D Sprite;
   2.1. An instance of **XOScript** attached to the above;
3. A large "O" 2D Sprite.
   3.1. An instance of **XOScript** attached to the above;

The "X" and "O" sprites are hidden by default, so the board appears clear.

GridCellScript and **XOScript** work in tandem to allow us to show or hide the "X" and "O" elements over the blank board cell image.

GridCellScript is the control script for the prefab itself, so I'll focus on it first…

First, we define a **CellStates** enumeration, which allows us to identify each state using words rather than numbers. As usual, this is mainly for readability.

Next come some private variables for the class: **_state** is used to hold the cell's current state: Empty, displaying an O, or displaying an X.

Next we have an array for XOScript components. There will only be two for now, but it's possible to extend the tutorial to handle games with more than two players, so there's no point fixing the array's size here.

The cell's associated XOScript components will be set up shortly, in the **Start()** function.

Next we have a **bool**ean variable that is used to flag a state change.

This is used to trigger code in the **LateUpdate()** function that is called every cycle. As we don't want to keep redrawing every cell for every cycle—we only need to update the board once, at each change of state—this lets us ensure the drawing is only done when it's actually necessary.

The reason we don't want to update every single cycle is because, on mobile devices, this can drain the battery more quickly. Mobile devices rely heavily on battery power, so it is good practice to avoid putting unnecessary pressure on it.

Finally, we define the **state** variable for the class, which is a **Property**, like its counterpart in **GameInfo**.

Here, we see an example where using a **Property setter** pays off: when the state is changed, we set **stateHasChanged**, ready for the **LateUpdate()** code. Next, we check if the state has changed to either 'X' or 'O'—if not, we're just resetting the game board.

If we're not just wiping the board, we use **SendMessageUpwards()** to tell the **BoardManager** script to run its **RefreshScores()** function. This saves us having to obtain an explicit reference to the BoardManager script. Any script that sits above this one in the project hierarchy will be sent this same message; if the script has a **RefreshScores()** function defined in it—and assuming it's declared as **public**—then that function will be run.

Start()

This function gathers up the **XOScript**s that are attached to this prefab. They're attached as components to the "X" and "O" sprites, rather than directly to this script. The result should be an array with two **XOScript** references, one for each of the two sprites.

Finally, it sets the cell's current state to *empty*, as this is the default state.

LateUpdate()

This function handles the updating of the game board:

If **stateHasChanged** is *true*, it checks the current state and uses it to show or hide each of the attached "X" and "O" sprite images accordingly. The actual showing or hiding is done within **XOScript**.

Finally, we reset **stateHasChanged** to *false* to stop the board being updated every single game cycle.

XOScript

This script is used to show or hide its attached image—either an 'X' sprite or an 'O' sprite—when called upon to do so by **GridCellScript**.

There are no public variables for this class. The only variable is a private one, named _**isX**. This is set up in the Start() function…

Start()

The 'X' and 'O' sprites in the board cell prefab are each tagged in the Inspector as either "cellX" or "cellO", respectively. We use this information to work out what image the script is attached to. (Hence the script's name.)

If the tag of the attached sprite is "cellX", we know this is an 'X' sprite, so we set _**isX** accordingly. If the tag doesn't match, we know it must be an 'O' sprite by a process of elimination.

This is all the information the script needs in order to function.

HideX() and ShowX()

These two functions are called from GridCellScript's **LateUpdate()** function. They first check if the script is attached to an 'X' image. If so, we either show or hide the sprite by enabling or disabling its renderer.

HideO() and ShowO()

These are almost identical to HideX() / ShowX(), above. The only difference is the test on _**isX**, which is inverted—the exclamation mark means 'not'.

And that's it for the scripts!

PART 3

The Making Of Tic-Tac-Two

As this tutorial doesn't follow the traditional step-by-step format, it's worth giving a broad overview of how the development process went…

What follows is a discussion of how *I* write programs. This does not mean you should follow it slavishly, as what works for me might not work at all for you. I have a very poor short-term memory, so much of my process has developed over the years to work around this rather irritating problem.

My Development Process

I tend to add a new feature, test it, refine it, then move onto the next section, so I don't follow a formal project plan. This is something a programmer with many years' experience can do: like any learned skill, after a while some processes happen at a subconscious, rather than conscious, level. My approach is therefore *iterative*: like building with Lego, I can see the end result in my mind's eye, so I know exactly what I'm aiming for, but I tend to thrash out the low-level details as I go.

I also tend to work alone, so I find I have little use for formal design documents; what paperwork I do create generally consists of a long To-Do list, the occasional (typed) note, and not much else.

Comments

One of my more obvious programming tics is comments: I write lots of them. My reasoning goes like this: "I've already got a text editor open right in front of me, so why open up another just to jot down my thoughts on what I'm programming?" I also have terrible handwriting that even I find hard to read, so I've long since given up on pen and paper notes as they are often worse than useless.

The script excerpts I've included in this tutorial have had most of these comments removed as the document itself already explains the code.

Experience & Hindsight

I've been working in this field, on and off, since the 1980s and I'm still learning new things every day. I used to plan and sketch out games in notebooks during the early stages of my career, but I don't do that as often now.

If you've never built a game before, you won't have the benefit of hindsight just yet. Trust me: you'll get that benefit very quickly!

There now follows a very important warning:

Don't take anything in this tutorial as gospel!

There is no One True Path to Enlightenment. There is no single answer to "How do I go about making a game?"

What I describe is what works for *me*. My brain is happiest working like this. Your brain may beg to differ and demand you find another approach to designing and developing your own projects.

Find out what works for you. In the creative fields, there's no such thing as one size fits all. This is especially true if you intend to work with other people: you need to find a system that works for *all* of you, and that's hard. This is why you often hear of teams and partnerships falling apart. It takes time to find the exact combination of personalities, skills and talents that 'click' together and work as one.

Treat this tutorial as a guide, not a formal template.

The Game Board

First, I decided to aim for an initial prototype that supported two players, but had no AI. Human vs. Human only, then.

A Spot Of R&D

For the original tutorial, this took a few hours as I decided to take the opportunity to experiment and see how different board sizes affect the gameplay. My R&D produced some interesting results: for the Tic-Tac-Toe rules to work, the board **must** be a 3x3 grid. As the game is won by completing a complete row, column or diagonal, it's effectively unplayable at larger sizes as games invariably end in a draw. You need to adjust the winning sequence length as a factor of board size to restore playability.

> Tic-Tac-Toe is such a simple game, and so easy to play with just a piece of chalk or charcoal on any convenient surface, there's a strong case to be made for it being the gaming equivalent of a fossil—the oldest, most primitive, living relative of more complex games like Nine Men's Morris, Go and even Connect Four.

None of this was, however, relevant to the main aim of the tutorial, so the code in this tutorial, and especially the AI, mostly assumes a traditional 3x3 grid is being used if this makes the code easier to write and to follow.

The Game UI

The game's user interface was originally going to be built on the long-awaited new Unity GUI system. This was announced well over a year ago at a Unite conference, but it was delayed. After a while, it became clear that it would be a while yet before it was released, so I decided to go ahead with a version built on the 2D game features released in Unity 4.3.

> It has since been announced that the new GUI system will appear in the Unity 4.6 release, but as Unity 5 is itself just around the corner, I have decided to wait for that before creating the next major revision of this tutorial.

This meant rolling my own GUI system based on basic 2D sprites. As I expected the new GUI system to appear shortly, I decided against making this GUI overly complicated. I pared it down to the basics instead.

This resulted in choosing just a single AI, rather than the three AI 'characters' I invented for the original release. The next release will see a "difficulty level" slider added to make up for this.

Also new in this 2nd version of the tutorial is the addition of Human vs. Human and Computer vs. Computer modes, which were requested in the support forum. The ability to set which piece ('X' or 'O') each player would play was also added.

As these are all options that can be simply toggled from one setting to the other, this made the GUI itself relatively easy from a design perspective. I created a mock-up of the game in *Indeeo*'s *iDraw*, which is a vector art / illustration app for the Mac. (It's similar to Adobe Illustrator, or CorelDraw.)

This also gave me the opportunity to replace the original tutorial's rather poor graphics with something a bit more polished and cleaner.

Once the mock-up was done, I arranged all the various graphical elements onto a single image and exported it as a PNG file for for Unity. Here, I opened the image up in the new Sprite Editor and chopped it up into little bits to create all the 2D Sprite elements I needed.

Every 2D Sprite that was clickable in the GUI needed a 2D Box Collider component added to it as the ray-casting method I decided to use only 'sees' collider components; it doesn't see anything else at all.

The board was built up of 9 manually positioned board cell prefabs, which were constructed of a 2D 'cell' sprite image, overlaid by two 2D sprites, one containing an 'X' image, the other an 'O'. The latter two had XOScript components added, while the empty board cell sprite had a GridCellScript attached instead.

Finally, I manually arranged all the board cells and the GUI sprite elements by eye. Once they were roughly in the right positions, I entered manual coordinates in the Inspector to snap them into line.

The next step was to write the GUI code itself, the results of which can be found in **GUIHandler**. This is not my proudest moment: it works, and it's actually very simple in concept, but it's not at all pretty as there are many, many tests and nested chunks of code.

In effect, what you see here is a very minimalist GUI system with all its guts exposed. I can't wait to replace it in the next version.

AI and game logic

I had originally intended to just copy and paste the AI and game logic code from the original tutorial, but I felt its attempt to support the Model-View-Controller design pattern was not very successful. In the end, not one line of code from the original tutorial made it into this version. It's a complete rewrite.

The temporary GUI system used in this release dictated some changes. For example, I originally intended to retain the three AI 'characters', but felt this would add little to the tutorial while requiring a lot of time and effort on the GUI side. I decided to drop it until the next release.

Similarly, you may have noticed that the **Player** script covered earlier includes some storage for game statistics data, but that this isn't currently in use. Again, the cut-down GUI meant adding these features would have required far more code than it was worth given the minimal benefit.

The Beginning

Despite all the verbiage in this document, the Tic-Tac-Two project it describes is quite simple. If you remove all the comments in the code, what's left is surprisingly short. Writing this documentation has taken me far more time than writing the project itself.

A lot more, in fact: the original tutorial had only 40-odd pages of documentation. I've more than doubled that with this document alone, while the accompanying ebook adds a further 70+ pages!

That Programming Primer ebook is now destined to follow its own path: I've made the decision to move away from my day-job of translation, so I can focus on writing instead. This will include revising, expanding and selling that ebook as a standalone book in its own right, with a version customised for inclusion in the next version of this tutorial. That alone will be over 200 pages long.

Thank you!

I hope this project has helped you understand how to create a simple turn-based game in Unity.

I can be contacted through the dedicated Tic-Tac-Tut support forum at:

http://forum.unity3d.com/threads/tic-tac-tut-unity-genre-tutorial-support-thread.139516/

Or via my website at: www.bangbangclick.com

Thanks for reading!