2D UNITY

JEFF W. MURRAY

NO STARCH PRESS EARLY ACCESS PROGRAM: FEEDBACK WELCOME!

Welcome to the Early Access edition of the as yet unpublished *2D Unity* by Jeff W. Murray! As a prepublication title, this book may be incomplete and some chapters may not have been proofread.

Our goal is always to make the best books possible, and we look forward to hearing your thoughts. If you have any comments or questions, email us at **earlyaccess@nostarch.com**. If you have specific feedback for us, please include the page number, book title, and edition date in your note, and we'll be sure to review it. We appreciate your help and support!

We'll email you as new chapters become available. In the meantime, enjoy!

2D UNITY

JEFF W. MURRAY

Early Access edition, 11/20/15

ISBN-13: 978-1-59327-701-7

Publisher: William Pollock
Production Editor: Serena Yang
Cover Illustration: Josh Ellingson
Interior Design: Octopod Studios
Developmental Editor: Hayley Baker
Technical Reviewer: Mike Desjardins
Copyeditor: Anne Marie Walker
Compositor: Susan Glinert Stevens
Proofreader: Paula L. Fleming

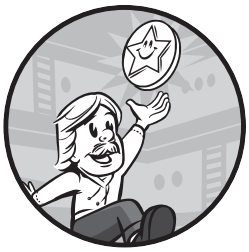
CONTENTS

The chapters in **red** are included in this Early Access PDF.

CONTENTS IN DETAIL

1

GETTING STARTED IN UNITY



Unity signaled the beginning of a major shift in who had access to high-end tools. Today, virtually anyone has access to this free and amazing development environment capable of making commercial games. In this chapter, you'll start by learning your way around Unity's main editor, also known as the integrated development environment (IDE), where almost everything happens. Then you'll explore a Unity project and learn how to manipulate objects.

Creating a Unity Project

Start Unity and you should see its splash screen followed by the login window. If you don't already have a Unity account, you can set up one now. The process is simple, so I won't outline it here; just follow the instructions

on the screen. Once you're logged in, Unity will display a project loading screen (Figure 1-1). Recently opened projects will appear in the Projects section, which makes it easy to pick up where you left off. To open a project that's not in the recent projects list, you would click Open other.

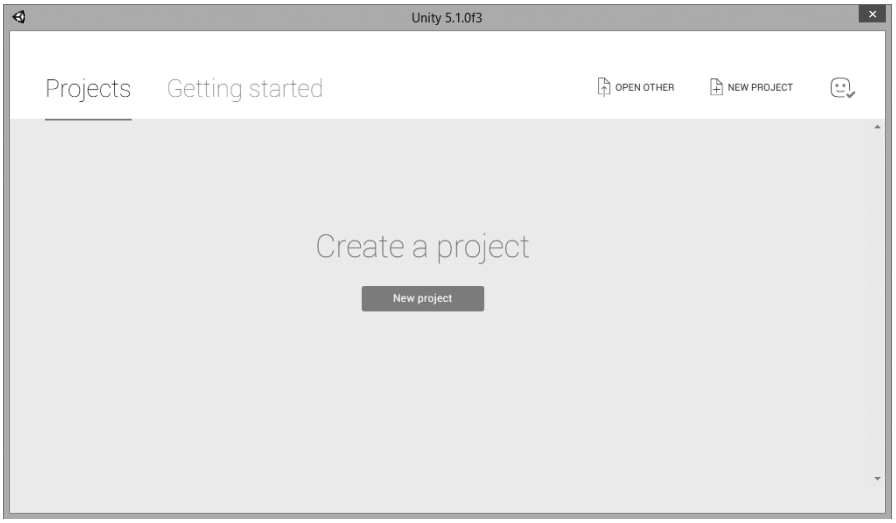


Figure 1-1: The Unity project loading screen

Let's make a new project. Click the **New Project** button to get started. The New Project window (Figure 1-2) appears and contains two choices for the type of project you want to create, 2D or 3D.

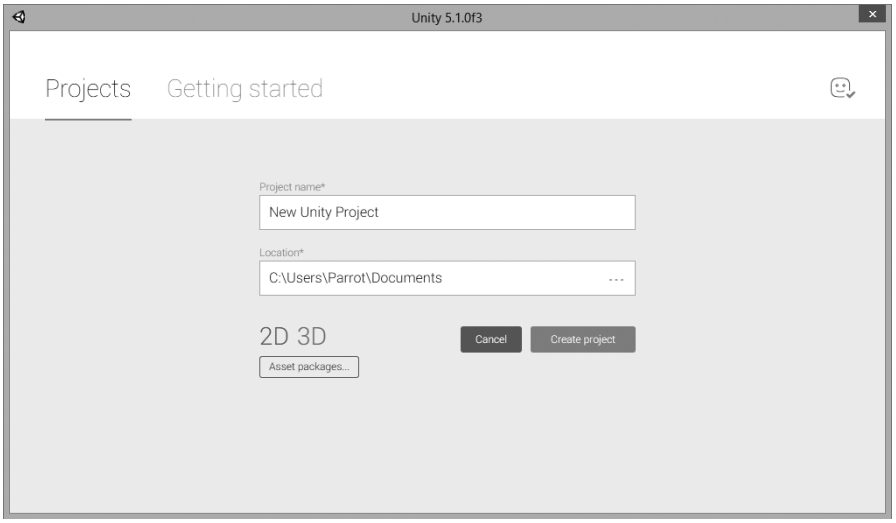


Figure 1-2: The New Project window

Enter the project name *Making2DGamesProject* in the Project name text field. Below that you'll see the default file location for your project. To change it, click the three dots to the right of the text field. Next, click the **2D** button and then click **Create project**.

The Main Editor

Before I explain each of the main editor's panels, let's change the layout. When Unity creates a project for the first time, the editor loads the default layout (Figure 1-3). The default layout is nice, but it can be a little difficult to use because it forces you to toggle between two of the panels you're going to be spending a lot of time with. Thankfully, just about everything in Unity is customizable, so you can tailor the panel layout to your liking.

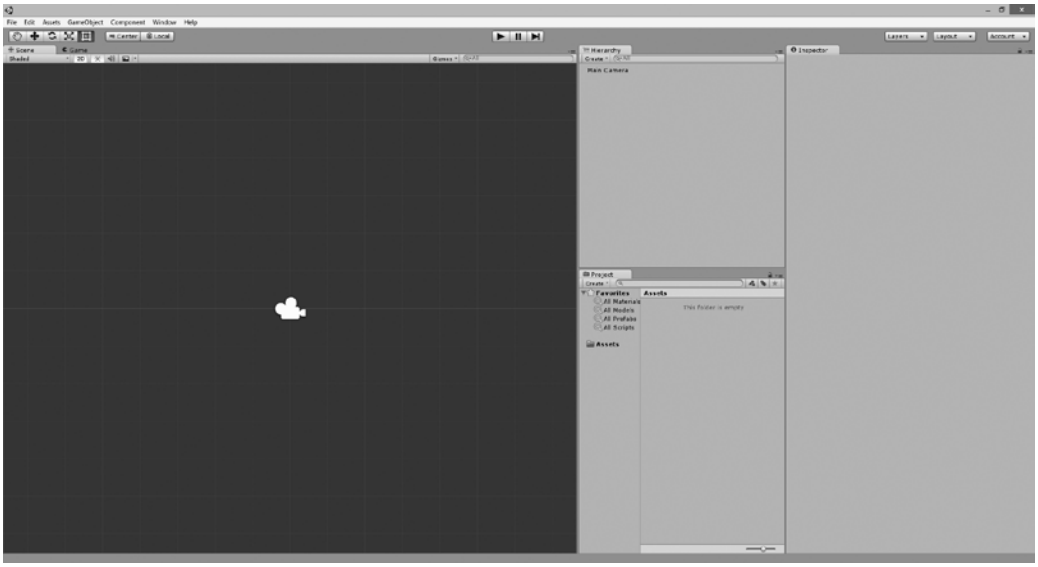


Figure 1-3: The main window of the Unity editor in a blank 2D project

A selection of preset layouts is available, but use the same 2-by-3 layout that I use so that your experience matches what you see in this book. Click the **Layout** button at the top right of the editor and change it to **2 by 3**.

NOTE *When you finish this chapter, be sure to test drive the other layout options. But for now, stick with 2 by 3 so you can see all of the main panels as I describe each one.*

The layout should now look like the one in Figure 1-4. This layout has five main panels, drop-down menus at the top left, and several tools along the top.

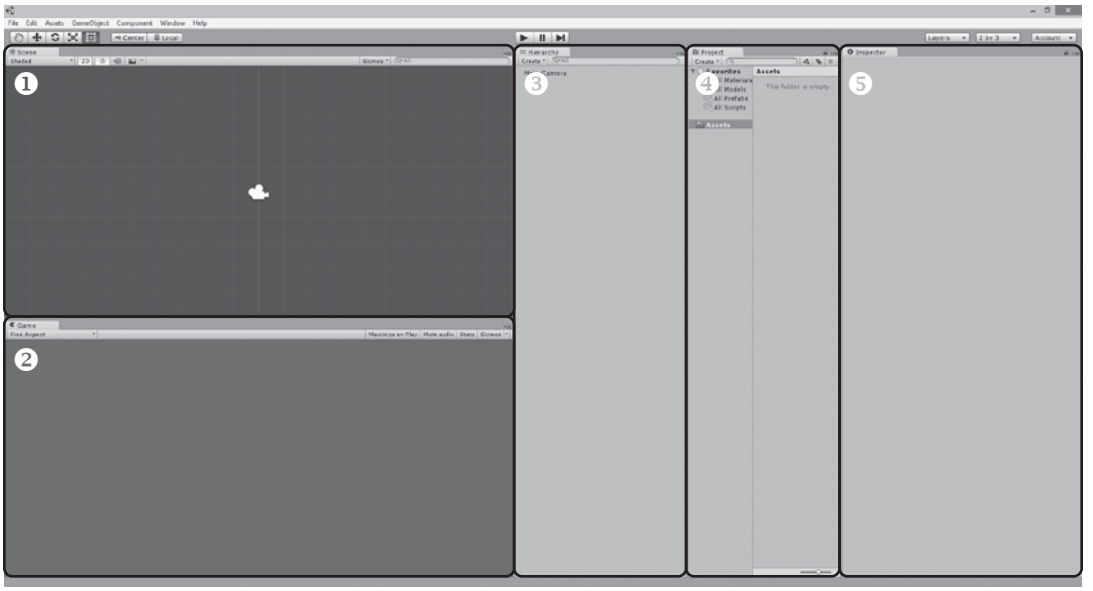


Figure 1-4: The Unity editor's 2-by-3 layout in a blank 2D project

You should see two stacked panels on the left. The Scene panel ❶ is a gateway to your game world. It provides a place to build and edit objects in a scene using a free-roaming camera and some tools.

The Game panel ❷ is where you test your game to see how it looks to the player. Press the Play button at the top of the screen to test your game at any time.

The Hierarchy panel ❸ lists every GameObject that the currently loaded scene contains. *GameObject* is the base class for the objects that make up your game's models, cameras, particle effects, sounds, and so on, and the Hierarchy lets you select, change, and create new GameObjects. The Hierarchy displays GameObjects in the order in which they were added to the scene, so the first object you add appears at the top and subsequent items appear underneath. However, you can drag and drop items to place them in any order you like.

The Project panel ❹ shows all the assets—the files and folders that contain your game's content—that belong to your Unity project. Just as with a regular file browser, you can create, copy, drag, and drop your files in this panel. The Create menu at the top left of the Project panel lets you add new assets and subfolders to the currently selected folder.

The Assets panel, a separate panel within the Project panel, shows currently selected items. The Project panel is fully searchable. The search box runs along top of the panel; you can search by asset name, type of asset, or both by clicking the drop-down arrow next to the magnifying glass.

The Inspector panel ❺ is context sensitive: when you select an item in the Project or Hierarchy panel, any editable properties will appear in the Inspector.

Anatomy of a Unity Project

Now that you know your way around Unity's interface, let's explore a Unity project. First, download the book's files from *http://www.nostarch.com/2DUnity/*. Then click **File ▸ Open**.

Browse to the *Chapter_1_Project* directory, which contains the example for this chapter. Click once to highlight the *Chapter_1_Project* folder and then click **Open**. Keep this project open for the rest of the chapter.

NOTE *If you double-click the* Chapter_1_Project *folder, the Open button may be disabled. If so, go back to the previous folder and select the project folder as I described.*

Project Directories

The files that make up a new Unity project are split into three main directories: *Assets*, *Library*, and *Project Settings*. The *Assets* folder (see Figure 1-5) contains all the graphics, sound effects, scripts, and more. The *Library* folder contains vital information that Unity uses: do not modify this folder in any way. The *Project Settings* folder tracks configuration information, such as keyboard mappings, how the panels appear in the editor, sound volume, and graphics settings.

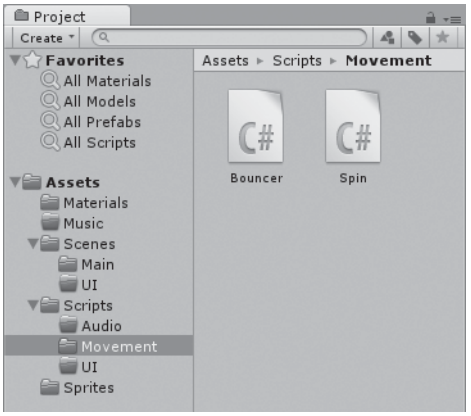


Figure 1-5: The Assets *folder as displayed in the Project panel*

The *Assets* folder is where you'll keep all the files you use in your game. A best practice is to store all of your *Assets* folder's files in subfolders with names based on what the files inside do. Doing so helps you find a file based on the function it serves in the game.

It's important to keep your project as organized as possible. By naming your folders in a logical manner and creating separate folders for different pieces of your project, you can find assets quickly. Staying organized makes it easier for you to return to a project later and helps a lot if you're collaborating with someone else.

Next, let's jump into the scene.

Navigating a Scene

The Scene panel is at the heart of the Unity editor. It's where you can visually edit everything that makes up your game's scenes.

Unity stores your GameObjects in *scenes*. You can use scenes to build levels, menus, and high-score boards.

With *Chapter_1_Project* still open, find the *Scenes* folder in Unity's Project panel. Expand the *Scenes* folder, click the *Main* folder, and look for a Unity logo icon labeled ObjectScene in the Assets section. (The icon that looks like the Unity logo represents scene data files.) Double-click the **ObjectScene** icon to open the scene.

Along the top of the Scene panel is a toolbar containing the scene control tools (Figure 1-6).

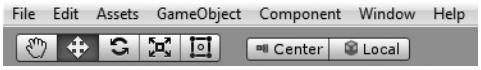


Figure 1-6: The scene control tools at the top left of the Unity editor

Starting from the left, the five tool buttons are as follows:

- **Camera Pan:** To drag and move the camera
- **Move:** To move GameObjects
- **Rotate:** To rotate GameObjects
- **Scale:** To scale GameObjects
- **Rect tool:** To use with UIs or sprites and to scale and move 2D elements in the Scene panel

Let's put these scene control tools to use. Click the **hand icon** to engage Camera Pan mode. In Camera Pan mode, you can move about the scene without affecting any objects. Some additional controls are also available for the mouse and keyboard:

- Use the arrow keys to move the scene camera. Hold down SHIFT to move faster.
- In any mode, you can access Camera Pan mode by holding down the ALT key or the middle button on your mouse to drag the camera with the mouse. You can also hold ALT-CTRL on a PC or ALT-⌘ on a Mac.
- To zoom in or out of a scene, either roll the mouse wheel or hold the ALT key and right mouse button at the same time.

Practice moving around the scene. Then continue to the next section to start moving, changing, and deleting objects.

Selecting and Manipulating Objects

Manipulating GameObjects in Unity is a breeze. We'll start by selecting objects in a scene.

Select a GameObject by clicking it. Try it now: click the **Move** icon in the scene control tools and then click any of the visible GameObjects in the Scene panel. If you're unsure about which icon to select, refer to the previous section to find the Move tool. After selecting an object, a tool handle appears on top of it, which you can use to move it (Figure 1-7).

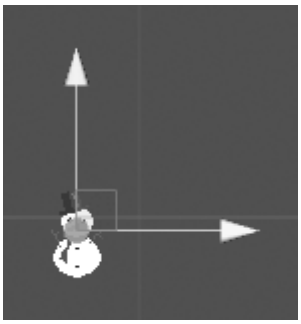


Figure 1-7: The tool handle is a visual aid that allows you to manipulate the currently selected object in the Scene panel.

Click any axis and drag it around to move the object along that axis, or click the cube in the middle of the axis to freely drag the object in any direction.

Notice that the name of the GameObject you have selected is also highlighted in the Hierarchy panel: you can select items in the Scene panel or find them in the Hierarchy and select them there. Let's explore the Hierarchy panel.

The Hierarchy Panel in Depth

The Hierarchy panel helps you organize your scenes. Without it, you'd have only a visual world to navigate, which would make accessing objects in a complex scene a nightmare.

Find the object named `ClickMe` in the Hierarchy and click it to highlight it. You probably aren't seeing the object in the Scene panel. Fortunately, this is easy to fix. Unity can automatically center the view onto the currently selected item. Move the mouse over the Scene panel and press the **F** key to center the `ClickMe` object in the Scene panel (Figure 1-8).

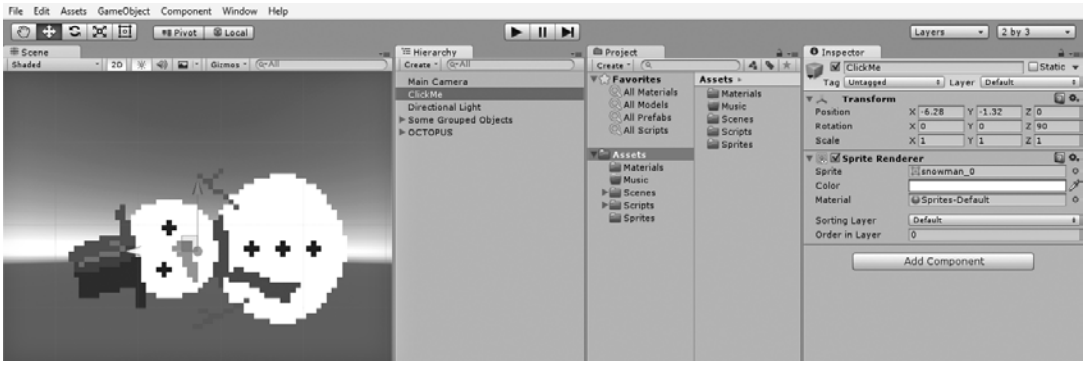


Figure 1-8: Centering the `ClickMe` GameObject makes it visible in both the Scene and Hierarchy panels.

Now press **Play** on the playback controls; some objects should move around in the scene while `ClickMe` remains static. That's because `ClickMe` doesn't have any *Components* attached to it. If you want to make your GameObjects do things, you must attach Components to them. Let's look at a more interesting GameObject that does have a Component attached to it.

Look for `ClickMeNext` in the Hierarchy and click it. `ClickMeNext` is a child object under the empty GameObject named `Some Grouped Objects`. Grouping objects this way is called *parenting*. Notice that the Hierarchy shows this parent/child relationship in the same way that a file browser shows an item within a folder.

After selecting `ClickMeNext` in the Hierarchy, look at the Inspector panel to see the Components attached to it. The Components list looks like this:

- **Transform:** The Transform Component is automatically attached to all GameObjects. It provides the GameObject's position, rotation, and scale information.
- **Sprite Renderer:** Sprite Renderer draws the sprite to the screen.
- **Spin (Script):** A *script* is a bit of code you can attach to an object to make it do something.

The spin script is a very simple script I added to rotate the snowman around its z-axis. I made the speed available to the Inspector, so you can adjust it by changing the value in the Speed field.

In the Inspector, the Speed text field shows a default value of 5. Change the Speed value in the Inspector and press **Play** to see what happens. Try a negative number too!

Let's continue with our Unity tour and look at how to rotate and scale objects.

Rotation and Scale

Click `ClickMeNext` in the Hierarchy, if it isn't still selected. Look at the Scene panel. If the snowman is not centered, hover the mouse over the Scene panel and press F to automatically center it. If nothing happens, check the Hierarchy to make sure the object is still highlighted. You may have to expand `Some Grouped Objects` to find it.

Figure 1-9: The rotation tool handle on the `ClickMeNext` object in the Scene panel

Once the object is centered in the Scene, click the **Rotate** button in the scene control tools. A tool made up of an outer circle and an inner circle with two lines intersecting at its center appears over the snowman (Figure 1-9).

Click and drag the lines inside the inner circle; either line will do. Dragging the circle around should spin the GameObject. Watch the live update of the transform's properties in the Inspector panel as you drag the circle—the numbers

in the rotation text fields should change. You can also modify the transform in the Scene panel by typing the values into the Inspector.

To the right of the Rotate button, on the scene control tools, is the Scale button. When you select Scale mode, a Scale tool handle appears on top of the selected object. You can click and drag the scale axis to resize the object, just as you did with the position and rotation handles.

Snap and Grid Settings

Lining up and spacing objects as well as making sure graphics are aligned correctly can be tricky to get right by sight alone. Unity includes a grid to help you with this process.

The Scene panel shows the default grid, which looks like faint lines in the background. Each square on the grid represents a unit. As you zoom in and out, the grid scale automatically changes (you can try this using the middle mouse wheel).

You can use the grid as a visual guide to help you line up GameObjects. Note that there's no option to change the size of the grid. The only option is to toggle it on or off in the Gizmos drop-down menu by checking or unchecking the Show Grid box.

The Snap system is also helpful for aligning objects. It helps to position them by *snapping* movement to a specified number of units. In Unity, units are an arbitrary measurement that you can set to the size you want. You'll learn more about units in Chapter 4, but for now we'll use the default value.

Usually, you can move objects anywhere you'd like by pretty much any amount. If you want to use the Snap system, hold down the CTRL key(or ⌘ key on a Mac). Turning on snapping moves objects based on the snap sizes set in the Snap settings menu. To change the Snap settings, click **Edit ▸ Snap settings** (Figure 1-10).

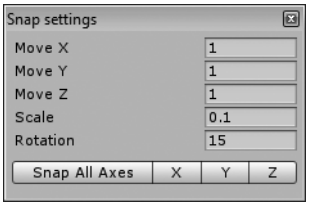


Figure 1-10: The Snap settings window

Try changing the Move X value to 10. Select an object in the Hierarchy. Next, select Move mode from the scene control tools. Hold down the snap key (CTRL, or ⌘ on a Mac) and drag the selected object around. Note how it moves just 10 units at a time. Return to the Snap settings window and change the Move X value back to 1. Try dragging the same object around; now it moves in 1-unit steps.

Snapping isn't just for positioning. Whenever the snap key is held down, snapping will apply to an object's scale and rotation, too. In the Snap settings window, you can change the snap values for scale and rotation separately. At the bottom of the Snap settings window is a row of buttons that lets you set which axis to apply snap to. Click the Snap All Axes option to apply it to all axes.

Copying, Pasting, Duplicating, and Deleting

The copy, paste, duplicate, and delete functions are a big part of everyday life as a developer. Most of these functions are available by right-clicking a GameObject in the Hierarchy. Doing so brings up a menu of common actions you can perform on a GameObject (Figure 1-11).

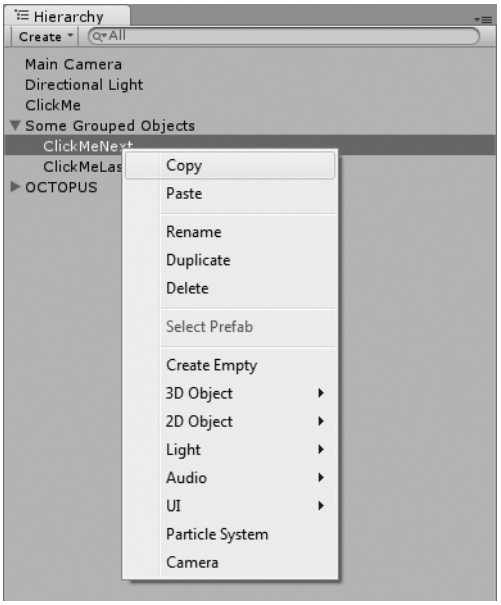


Figure 1-11: Right-click a GameObject in the Hierarchy panel to display a menu.

You'll do a lot of copying, pasting, duplicating, and deleting objects in Unity, so memorizing the keyboard shortcuts for these actions will save you a lot of time:

- **Copy:** CTRL-C or CMD-C
- **Paste:** CTRL-V or CMD-V
- **Duplicate:** CTRL-D or CMD-D
- **Delete:** DELETE or CMD-backspace

Let's try duplicating: right-click the `ClickMeNext` object in the Hierarchy, and then select **Duplicate** from the menu. A copy is automatically named `ClickMeNext 1` and is selected and ready for use.

Next, give it a more descriptive name. With the `ClickMeNext 1` GameObject already highlighted in the Hierarchy, click the `ClickMeNext 1` text to change its name, just as you would in a file browser (Figure 1-12). Change the duplicate GameObject's name to `MyCopy`.

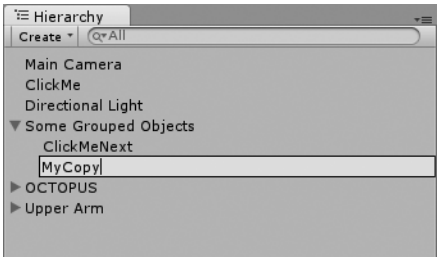


Figure 1-12: Renaming a file in the Hierarchy panel

NOTE *You can rename GameObjects in the Inspector, too. A selected GameObject's name is shown in an editable text field at the top of the Inspector.*

By default, a duplicated object appears at the same position as the original. It's difficult to select the duplicate in the Scene panel, but it's easy to accidentally select the original. In this case, it's best to select objects in the Hierarchy. With the `MyCopy` object still selected in the Hierarchy, hover the mouse over the Scene and press **F** to center the view on the sprite.

Select the Move icon in the scene control tools. Click and drag the horizontal axis on the handle to move the model to the side just enough to see the two objects separately in the scene.

Adding Components

Click **Play** to preview the current scene. Right now, `MyCopy` just remains in one location and spins. Let's make it do even more!

Click **Stop** to stop the preview. Select `MyCopy` in the Hierarchy and look at the Inspector. Find and click the **Add Component** button in the Inspector, and you should see all the available Component categories. Let's add a script to make this object move. Note that you're not actually creating a script but instead just adding an existing script to the GameObject.

In the Add Component drop-down menu, click the **Scripts** category to find the `Bouncer` script, which is a bit of code I wrote to make an object bounce. Select **`Bouncer`** to add it to the GameObject.

The `Bouncer` script makes a GameObject move back and forth along either its x- or y-axis. It uses a sine wave to calculate how much to move the object each frame; you can open it and take a look if you want, but at this stage the actual workings of the script aren't very important.

Now that you've added `Bouncer` to the GameObject, a new Component appears in the Inspector, and you can access a few of its parameters:

Wave amplitude Specifies the size of the waves (how much the object will move)

Bounce speed Specifies the rate, or how quickly the object will move

Movement direction Select either Horizontal or Vertical to specify whether to move the `GameObject` along its x- or its y-axis

Click the **Play** button in the playback controls. By default, `MyCopy` should move up and down, but only a little. While the game preview plays, change the movement direction to horizontal or vary the values in the Bounce speed or Wave amplitude boxes. The ability to edit Component values as the game is running to see instant results is one of Unity's best features.

But Unity stores values only while the game preview is stopped. If you click the Stop button on the playback controls now, any values you've changed will be lost, and the engine will reset them to their original values. If you want to retain some values you changed during playback, jot them down so you can enter them again after playback is stopped.

In the last part of this chapter, I'll describe some useful visual aids and preview functions. Find the GameObject named `OCTOPUS` in the Hierarchy, center it in the Scene, and let's wrap up!

Gizmos

Along with the `OCTOPUS` GameObject you should see two other icons (Figure 1-13), which are called *gizmos*.

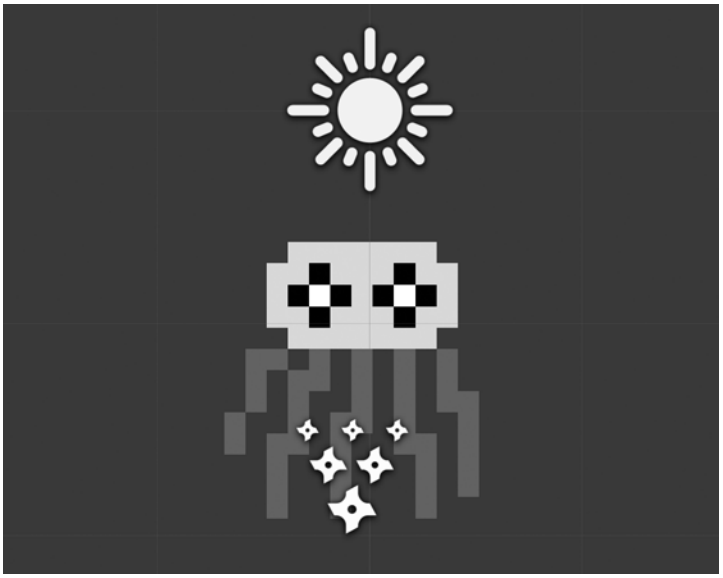


Figure 1-13: Gizmos shown in the Scene panel on the `OCTOPUS` GameObject for directional light and particle effects

What are gizmos? Well, you don't have to worry about feeding them after midnight! In Unity, gizmos are visual debugging and setup aids that make it easy to find GameObjects. Recall from "Snap and Grid Settings" on page 9 when you turned the grid on and off: the grid is a gizmo, too.

The Scene panel should show two types of gizmo for the `OCTOPUS` object: a sun and some little shurikens (yes, the traditional Japanese throwing stars). The sun represents a directional light in the scene. The directional light is a GameObject that you can move and rotate with a light Component.

The shuriken gizmo icon represents Unity's particle system. Click a shuriken in the Scene panel to select the GameObject containing the particle system Component. The gizmos will disappear and the particle system will run in its place, letting you preview the effect: you should now see a small army of octopi particles. In the Hierarchy panel, you'll see that the particle system is a child object of `OCTOPUS`. Because the particle system is the parent of `OCTOPUS`, whenever the `OCTOPUS` moves, the particles should move with it.

When you select a particle system, you should see a particle effect pop-up window in the bottom right of the Scene panel. This window lets you control the particle system so you can make and test effects. The particle pop-up window has two buttons—a Simulate/Pause toggle and the Stop button—and two text fields for setting the playback time and speed of the effect.

Two gizmos toggle buttons are located in the main editor: one in the top right of the Game panel and another in the top right of the Scene panel. When a gizmos button is highlighted, the helpful gizmo graphics will appear. A drop-down arrow next to each gizmos toggle button provides options for which gizmos will be drawn (in a checklist format) and their sizes via slider bars.

Previewing Aspect Ratio and Screen Resolution

Let's explore how GameObjects in your scene will look at different screen sizes or on different platforms. Click the Free Aspect drop-down menu at the top left of the Game panel (Figure 1-14).

The Free Aspect option lets you expand the game's view to fill the entire Game panel, whereas the other options scale the view based on a ratio, such as 5:4, 4:3 or 16:9, or a specific screen resolution. Here, the resolution is set to the default stand-alone player at 1024×768. In this context, Unity uses the term *player* to refer to the final build file. Along with screen resolution settings, you can find all the player settings in Edit ▸Project Settings ▸Player.

If you want to use screen resolutions or aspect ratios that aren't in the drop-down menu, click the + (plus) icon at the bottom of the drop-down menu to add new ones. Any resolutions or ratios you add will be saved in your project settings, but they'll only be available in this project.

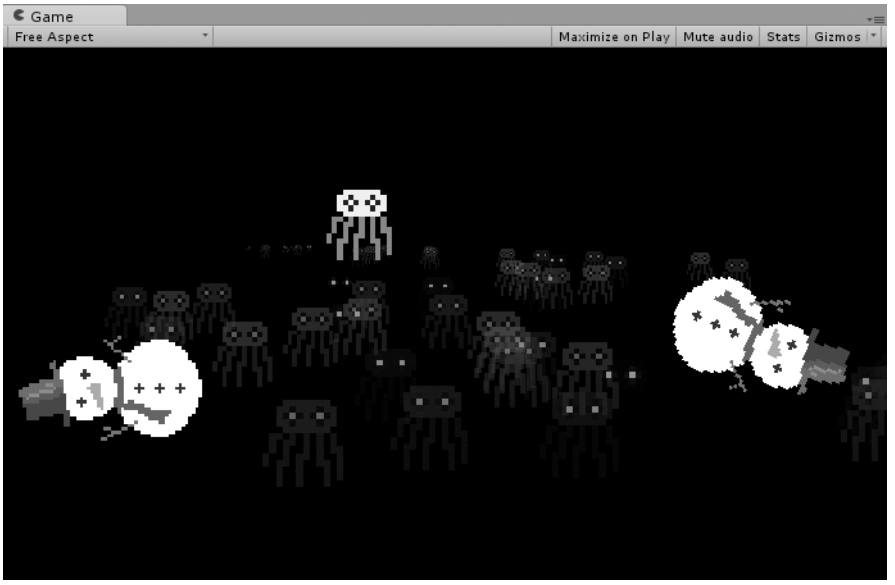


Figure 1-14: The Game preview panel showing playback and the Free Aspect drop-down menu

For this demo we don't need to change the aspect ratio, so if you changed it, set it back to Free Aspect. Then click the **Maximize on Play** button (to the right of the Free Aspect button in Figure 1-14) to toggle the button on. Click the **Play** button in the playback controls. The Game panel should expand to fill the entire editor.

The maximized play mode is great for getting a better view of your game, but because it doesn't quite fill the entire screen, you can still access the playback controls to stop, pause, or step frame by frame. When you stop playback, the Maximize on Play function will automatically return the Game panel to the size it was before playback started.

Now let's take a quick look at some of those stats to see which ones you need to keep an eye on.

Checking Your Game's Stats

With your game running, click the **Stats** button at the top right of the Game panel. The Statistics window (Figure 1-15) gives you information about how your game uses its resources, including the volume level, tris (short for *triangles*, referring to the number of triangles the renderer is drawing), and so on. It's a great place to get a bird's-eye view of your game's performance.

For example, a high number of SetPass calls (under the Graphics header) can directly affect how well your game runs. SetPass calls are calls to the graphics card to send information telling it what to draw, and they're sent every time a frame update occurs as well as when the screen needs to be updated.

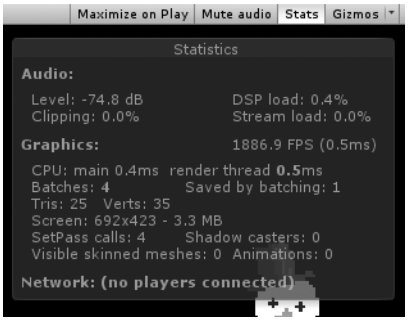


Figure 1-15: The Statistics window in the Game panel

Perhaps surprisingly, triangles are another statistic to keep an eye on. Every sprite in front of the game's camera, rendered to the screen, uses triangles, even though it's a 2D graphic. I'll discuss this in more detail later in the book, but for now let's just say that in Unity, 2D sprites are made up of flat 3D shapes that are rendered by an orthographic camera to make them look 2D. For that reason, if you're drawing numerous sprites and your game starts having performance issues, check the numbers in the Statistics window first for bottlenecks.

Many other useful stats are provided in the Statistics window, but they're beyond the scope of this book. If you want or need more specifics, the Unity documentation (Menu Help ▸ Unity Manual) provides some great information about it.

Closing Thoughts

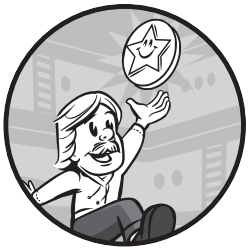
Phew, you've just learned a lot about Unity. Congratulations! If you think you need a bit more practice before moving on, play around with the editor until it makes sense to you. The best way to learn Unity is to use it.

This book's example files are available at *http://www.nostarch.com/2DUnity/* for you to experiment with, so download the projects, open them in Unity, tweak them, and dismantle them. If the editor breaks or a file stops working, you can always redownload the files and start again, so you have nothing to worry about.

The intent of this tour was to introduce you to Unity. Once you get into hands-on, practical game making, you'll be more comfortable navigating the interface. As you add sprites, gameplay logic, and Components, you'll get used to how all the elements fit together. In the next chapter, I'll show you some free graphics programs you can use to make sprites. You'll create some graphics, and then you'll go back to the editor to start making games!

2

GRAPHICS FOR YOUR GAMES



In this chapter, you won't use Unity very much. Before digging into Unity to create animations, we'll explore some options for making graphics for your games. Tons of tools make it easy to design sprites, tiles, textures, and anything else you might want to create!

I'll describe some basic aspects of 2D animation and walk through the graphics formats you'll be using in this book. I'll also discuss ways to obtain premade images and introduce two useful tools to help you create your own: GrafX2, a powerful, free graphics program, and Piskel, a sprite sheet generator. The chapter ends with two projects: you'll create a brick tile graphic and an animated player sprite that you'll use to create a platforming game in later chapters. If you don't feel very artistic, the finished images are also included in this chapter's files.

Let's get started!

Key Graphical Elements in 2D Games

Three key elements of 2D game graphics used in Unity are *textures*, *sprites*, and *sprite sheets*. You'll see them a lot. To help you follow along, here's a quick primer.

A *texture* is an image inside Unity. The texture doesn't do anything in a game until it is referenced by some code to draw it onto the screen. A *sprite* is an image in your game. It's a texture that's being drawn to the screen by a *Sprite Renderer Component*. The Sprite Renderer Component is one of Unity's built-in scripts. In this book, you'll create player sprites, object sprites, and even sprites for effects such as explosions.

2D animators create the illusion of movement by displaying a series of images. Each image in your game takes up memory, so game developers often conserve resources by using a *sprite sheet*, a single texture file made up of every image used to animate a particular object (see Figure 2-1 for an example).

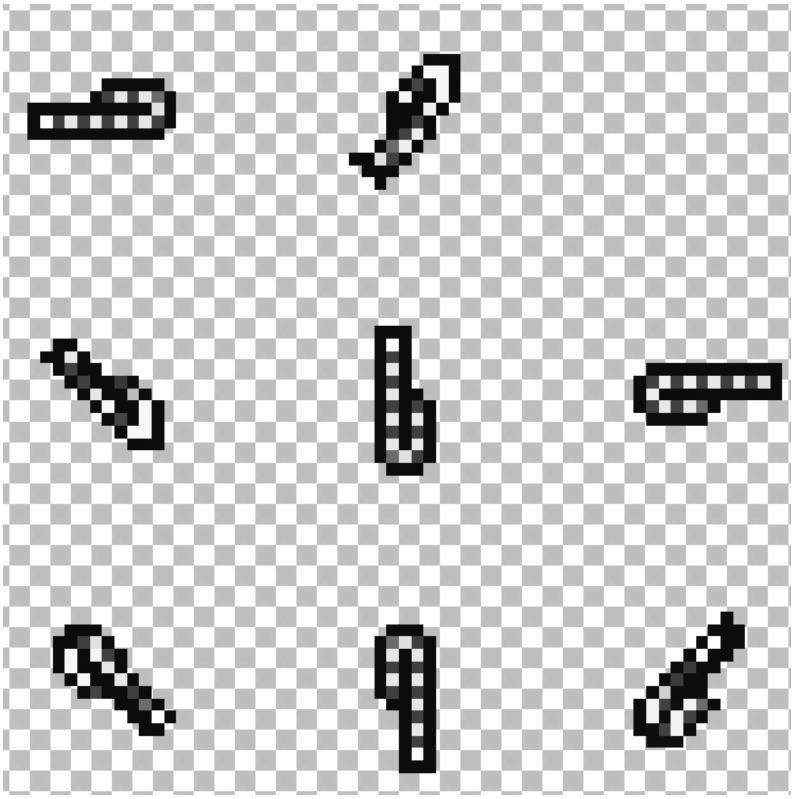


Figure 2-1: A sprite sheet for a spinning candy cane from my game Santa vs. The Meanies

In Figure 2-1, the candy cane sprites on the sprite sheet are equally spaced apart. Throughout an animation, the game displays one sprite from the sprite sheet at a time so you don't need a separate file for each one. Sprite sheets reduce the amount of memory your game will use.

In this chapter, you'll learn how to make your own sprite sheet, and in the next chapter, I'll show you how to use this sprite sheet to create an animated sprite.

NOTE *The term* sprite *dates back to the 1980s when it was first used to refer to graphics overlaid on top of background images. Images "floated" around without affecting the background image, as if they were ghosts, or sprites.*

Image Formats in Unity

Unity supports JPEG, PNG, BMP, GIF, IFF, TGA, PICT, and many other image formats, even Adobe Photoshop PSD and TIFF files. Of course, some file formats work better for certain functions than others. In this book, I'll use GIF files to make animations and convert them to PNG files when I want to import them into Unity.

The reason I use PNG files is that PNG files support *transparency*. To allow for curves and other shapes, images contain transparent pixels. Pixels can be transparent to different degrees, from slightly see-through to invisible. You can take advantage of this to achieve a wide range of visual effects, such as making images that look like glass. Player sprites usually have a transparent background, so Unity only draws the character.

When developing in Unity, you should be able to use the same image files for any platform, from desktop PCs to mobile devices and consoles. Unity usually handles any necessary platform-specific conversions automatically.

Choosing Image Size

Image size is another detail to consider when making your game graphics, because it can affect how your game runs and how much memory it will use. If you're using the free version of Unity, I recommend creating images that use a square canvas whose dimensions are a power of two (2, 4, 8, 16, 32, 64, 128, 256, and so on) to help the engine move your image data around faster.

If you're using Unity Pro to build sprite sheets for you, you shouldn't need to worry about image sizes. Normally, Unity Pro can do all the resizing for you. Make sure the images are 2048×2048 pixels or smaller to avoid any issues with older or limited hardware.

Obtaining Premade Graphics

If you'd rather focus on designing your game instead of creating art for it, you can use premade graphics or stock art. Lots of online resources are available to get graphics at a low cost or for free. You even can buy them directly within Unity!

Stock assets have their pros and cons. They are great for getting up and running quickly so you can focus on designing your game. The downside is that you're sharing assets with other people, which means you risk your game looking the same as someone else's. It's a bit of a trade-off between quality and originality, but it's your choice.

Buying Stock Assets

If you have a little money to spend, you can buy assets from the Unity Asset Store. You can pick up graphics from as low as $2, and you have hundreds of choices.

To access the Asset Store from Unity, click **Window ▸ Asset Store**. If you're not already logged in to your Unity account, you'll be prompted to do so. As soon as you're logged in, you're ready to shop. Anything you buy is added to your account for you to download as a *.unitypackage* file. Be sure to keep an eye on any licensing restrictions when you buy something, because special terms might apply.

Using Royalty-Free or Public Domain Assets

Another great way to get graphics is to download royalty-free or public domain images, which are also free. Here are some great resources:

2D Game Art for Programmers *http://www.2dgameartguru.com/*

BackYard Ninja Designs *http://www.dumbmanex.com/bynd_freestuff.html*

Lost Garden *http://www.lostgarden.com/*

Open GameArt *http://opengameart.org/*

Some amazing free assets are available to you, but make sure you read an asset's license terms carefully before using it in a game project. If you're not sure whether you can use an asset in your game, try to contact the artist directly or get some advice from a copyright expert. It's very generous of the creators to share these assets for free, so please respect their wishes. If an artist asks for credit to use their material, be sure to oblige.

Create Classic Pixel Art with GrafX2

If you haven't run off to get some premade artwork for your game by now, you must want to create your own. Awesome! For the rest of this chapter, you'll use GrafX2, a free program that's great for drawing classic pixel sprite graphics. It's very simple compared to other tools, such as Photoshop or

Gimp but it gets the job done, and it's fun to use. For example, when I designed sprites for a game called *My Nuclear Octopus* (Figure 2-2), I aimed for an arcade game vibe, and GrafX2's blocky pixels suited that style.

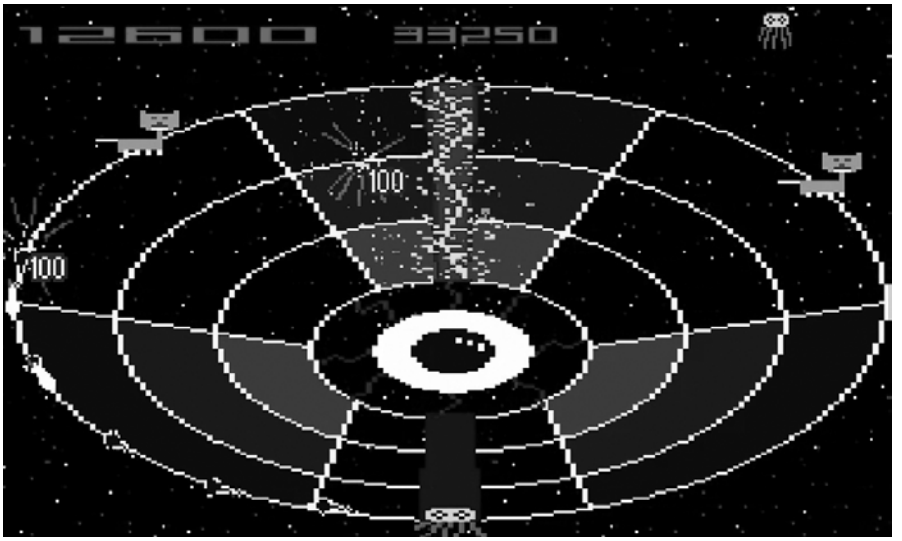


Figure 2-2: My Nuclear Octopus*, by PsychicParrot Games*

The platforming game you'll build in Chapter 9 will need some art, and in this chapter, you'll create a brick tile and a character as part of that game. In this section, you'll install GrafX2 so you can start drawing those assets.

Downloading and Installing GrafX2

GrafX2 is a neat program based on an old-school, pixel art program called Deluxe Paint, which was originally created for the Commodore Amiga 1000 in 1985. It's always fun to explore new tools, but of course you can use any program you like when you make your own graphics. Just make sure you save your images as PNGs or a similar format suitable for Unity.

To get started, download and install GrafX2 from the Download section of the GrafX2 website at *http://pulkomandy.tk/projects/GrafX2/downloads/*. You should see a list of versions available for different platforms, including Windows, Mac, and Linux. For Windows, download the installer (named something like *Graf<X2-X.X.XXX>.win32.exe*) and run the installation program. The file you need is in the *Bin* folder. Run the *GrafX.exe* file to get started.

On a Mac, download the binary file (named *GrafX2-svn<XXXX>-macosx .tgz*) and extract it in a suitable location, such as your Applications folder. Browse to wherever you extracted the file and launch the GrafX2 application from there.

Getting Started with GrafX2

After launching GrafX2, you should see its multipurpose splash screen (Figure 2-3).

Figure 2-3: The GrafX2 user interface

Click the **Anim** button to begin drawing, and you'll be greeted with an empty screen. Welcome to GrafX2!

GrafX2 starts with a blank canvas and a default 256-color palette. When you click and drag the mouse around the screen, it should draw pixels. But the default canvas size is too big, so let's fix that.

Click the icon that looks like a computer monitor (Figure 2-4) to bring up the Picture & Screen Sizes dialog (Figure 2-5). Click the **Width** field ❶, use the BACKSPACE key to delete the existing number, and enter **16**. Then click the **Height** field ❷, use the BACKSPACE key again to clear the field, and enter **16**. Finally, click the **Pixel Size** drop-down menu ❸ and select **Normal 1x1**.

Figure 2-4: The Picture & Screen Size icon is at the top left of the System tools.

Below the Pixel Size drop-down menu is a list of commonly used screen sizes. These options are for drawing large images, and because we're working on small sprites, you won't need them right now. Click **OK** to create a new canvas set to the new size.

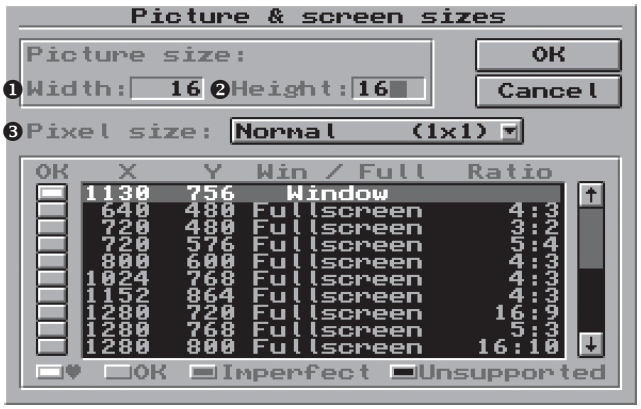


Figure 2-5: The Picture & Screen Sizes dialog.

A small image preview box should appear in the top left of the main GrafX2 window. Right now the window is pretty small, and it's difficult to work at this size, so let's zoom in.

Press the **M** key to enter Magnifier mode (Figure 2-6). In magnifier mode, the **+** and **–** keys on the keyboard's numeric pad zoom in or out of the image. If you're using a mouse with a mouse wheel or touch scroll, you can use the mouse wheel to control zoom.

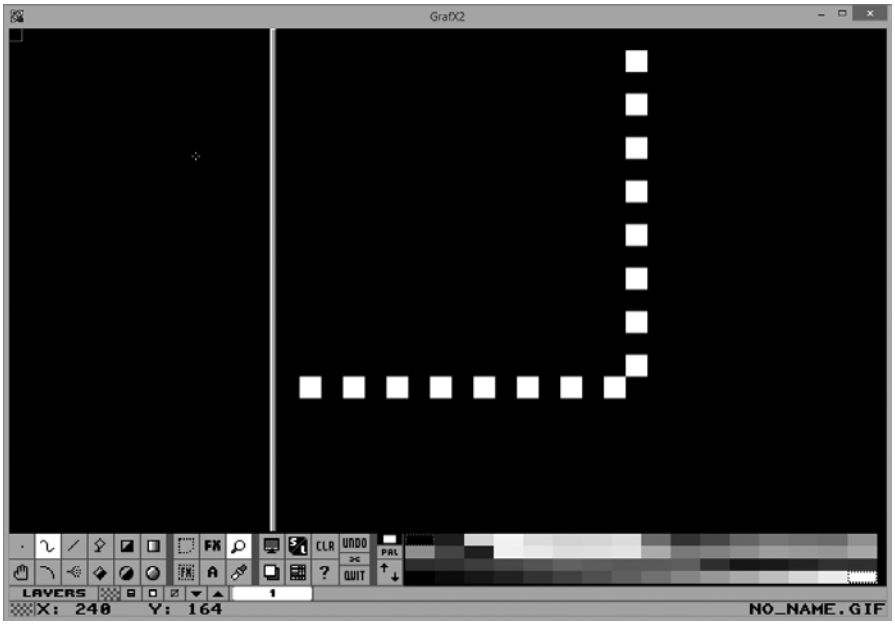


Figure 2-6: The GrafX2 interface in Magnifier mode

The magnified image should appear on the right side of the screen; the dotted border represents the edges of the sprite. You can draw anywhere inside the border using the Drawing tools in the bottom-left corner (Figure 2-7).

Figure 2-7: The Drawing tools with the Freehand Draw tool selected

You'll probably use the Freehand Draw tool the most, which is the wavy line icon on the top row of the toolbar. This tool lets you draw with the mouse. To the left of the Freehand Draw tool is the Paintbrush tool. Click it to choose brush styles, such as pixels, small squares, circles, dots, or lines. For now, just select the single-pixel image from this menu.

In GrafX2 you can select a foreground color and a background color to work with. Look at the color palette at the bottom of the screen (Figure 2-8). To set the foreground color, click any of the color blocks. To set the background color, right-click a color block. To draw using the foreground color, click inside the canvas; to use the background color, right-click inside the canvas.

Figure 2-8: The color palette

Making a Brick Tile

In this section, I'll show you how to make a simple brick tile you can use to build platforms in your games. A *tile* is a small rectangular image. Back in the day, arcade games were made of grids of tiles, as shown in Figure 2-9. This is still a common approach to game design, and I'll discuss it in depth in Chapter 8.

The tile-based level system that you'll create calls for all tiles to be the same size and same square shape. The level tiles will be 16 pixels wide by 16 pixels high.

Set the Image Size

If you followed along earlier and resized your canvas, you should be ready to go. You'll be working with a 16×16 canvas. If you need to resize your canvas, follow the steps in "Getting Started with GrafX2" on page 22.

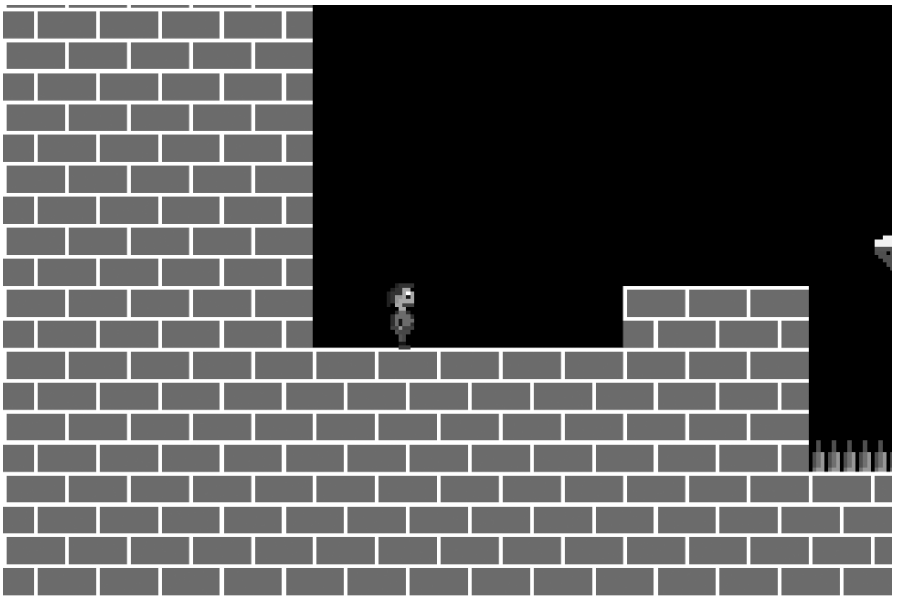


Figure 2-9: A level built of brick tiles

Draw the Brick Tile

Tile graphics are usually repeated in a pattern, so we need to make sure they fit together nicely. This feature is called *seamless tiling*. In its simplest form, seamless tiling ensures that pixels align correctly on all four sides of the image to make a pattern that's pleasing to the eye. In this project you'll create a simple brick pattern, as shown in Figure 2-10.

Figure 2-10: A simple brick pattern

Draw an Outline

The brick pattern we'll use is a traditional artistic method of representing bricks. This pattern looks like builder's bricks piled on top of one another.

To start drawing, first compose the lines that will form the cement between the bricks. Select the Lines tool from the main toolbar and draw four lines, like this:

1. Enter Magnifier mode to zoom in.
2. Draw a horizontal line across the top of the image.
3. Draw a horizontal line across the image along the 9th pixel from the top.
4. Connect both horizontal lines with a vertical line along the far left side.
5. Draw a vertical line from the bottom to the middle horizontal line along the 9th pixel from the left.

Color the Bricks

Once the lines are in place, you'll color in the spaces. Select red from the palette, and then select the Flood Fill tool (which looks like a little bucket pouring paint—see Figure 2-11) on the toolbar. Click each empty area, being careful not to click on the actual lines, to fill out the image and create seamless tiling.

Figure 2-11: The Flood Fill tool

Check Your Seamless Tiling

To make sure your fantastic brick tile works the way it's supposed to, use the Adjust tool (the hand icon underneath the Brush tool) to move your image around. By clicking the mouse and moving your image, you can see whether the edges match up and correct any mistakes. Try moving the image around so it looks like the one in Figure 2-12.

After you've checked out your fancy new brick pattern, make sure the image is back to how it was before you used the Adjust tool. That is, if it's off-center, use the Adjust tool to recenter it.

Figure 2-12: Move the image around with Adjust tool to make sure the edges match.

Save Your Work

Alright! Now you need to save the brick pattern as a PNG file (Figure 2-13). Follow these steps to save any of the drawings you create in GrafX2:

1. Click the **S** in the **Save/Load** icon.
2. Select **png** from the Format drop-down menu.
3. Click **Select drive** ❶ to change to the drive you want to save your work to.

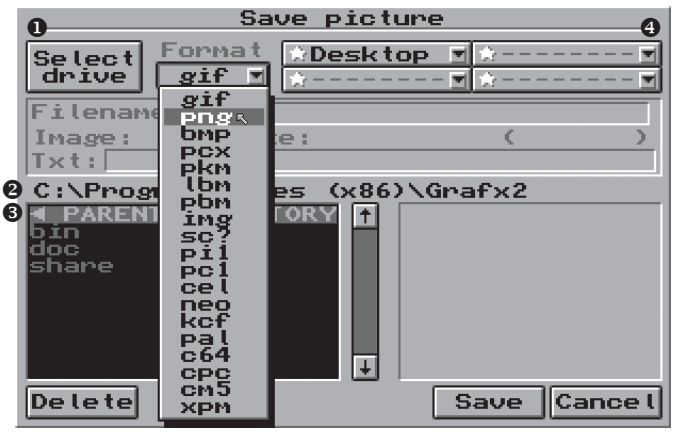


Figure 2-13: The Save Picture dialog

The current save location is shown as a file path ❷. Below the file path is a window containing files and folders at the current save location ❸. Click the files or folders in the window to navigate to where you want to save your image.

GrafX2 uses an old-school file-saving UI, which you might not be accustomed to. To access a folder you're familiar with on Windows, like *Pictures*, click **C:**, then **Users**, and then your username (which defaults to User if you haven't set it on your computer). You should see a list of folders familiar to you. Select the one you want to save to. When you click the Select drive button on a Mac, you'll see available folders in the files section in the bottom left. To access familiar folders, such as *Documents* or *Desktop*, click **Users/*username*** followed by the folder you want to save to.

To save time, you can assign save destinations to Favorite buttons, which are the small white stars in the top right of the window ❹. Right-click one and select Set to save the selected file location. Next time you need to access this folder, you can skip right to it by clicking the button. Now you can continue with the instructions:

4. Name your file *brick_tile*. Although a file extension (*.png*) appears in the filename field to start with, you don't need to add the file extension; GrafX2 will do that for you.
5. Click the **Save** button to save the image.

That's it. Let's move on to a more ambitious task: creating a player sprite!

Making an Animated Player Sprite

To create a player sprite, you'll use a simple pixel art style. When you're working at this level of resolution, too many details can make your character look strange. Give your character blocky, exaggerated features so they stand out! Look back at some of the most iconic arcade game characters, and you'll notice they have exaggerated features for the same reasons. Imagine how the original Mario sprite would look without his mustache or what Mega Man would look like with smaller eyes.

Of course, designing a character is an artistic process, and everyone has their own style and inspiration. As you follow along in this section, you'll pick up some techniques that you can use in your own graphics.

Set the Image Size

Before you start creating the sprite, you need to set up a suitable image size. Follow the same resize instructions from "Getting Started with GrafX2" on page 22 and set the width to **16** and the height to **20**. Make sure the pixel size is set to **Normal 1x1**.

Draw Your Character

In this section, you'll draw and animate Max (Figure 2-14), the star of the platforming game we'll create in later chapters. Max's walk cycle is two frames of animation, which is just enough to provide a satisfying illusion when he's moving around the screen.

GrafX2 lets you draw with a single pixel, or you can use the built-in brush shapes. For this character, let's use a circle brush to lay out his body and then draw with a single pixel to fill in the details.

Figure 2-14: The completed Max character

Draw the Body

Open the Paintbrush menu by clicking the **Brush** tool and click the circle in the fourth column on the second row down (Figure 2-15). Your brush should now be a circle. Next, select a skin color for Max from the color palette; I chose a peachy pink.

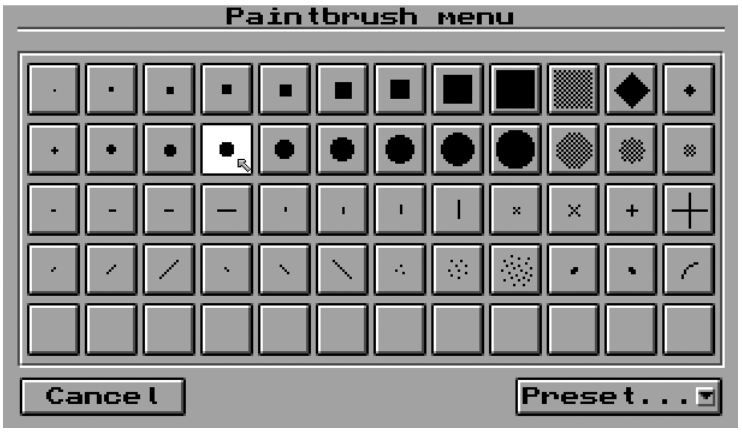


Figure 2-15: The Paintbrush tool menu

Using your circle brush, make the character's head and body by drawing two circles, as in Figure 2-16.

Now we have a basic template that we can build on: a head and the main part of the body. Treat this as a side view of the character, with his body pointing to the left. Next, we'll fill in the rest of the body, basically by doodling!

Connect the two circles to form a two-pixel-wide neck. Also, draw a two-pixel-wide leg and a foot. This produces the character's full body, shown in Figure 2-17.

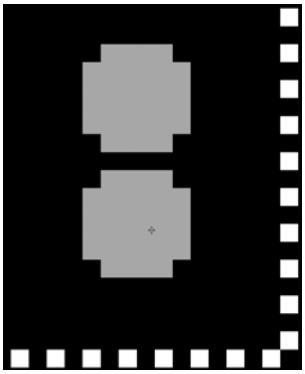


Figure 2-16: Two circle brushes will form the head and body of the Max character.

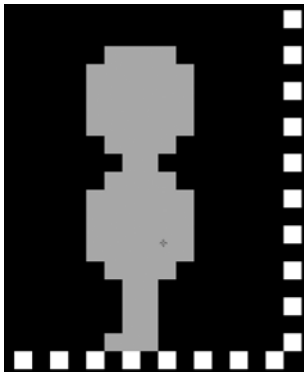


Figure 2-17: The Max character is starting to take shape.

Design a Stylish Outfit and Hairdo

Let's add some color to give Max pants, a shirt, and hair. I chose brown for the pants. If you draw a brown line across Max's waist, you can select the Flood Fill tool and click on his legs to fill in the rest of his pants. You can add a shirt using the same technique.

Click the Freehand Draw tool and draw a line of pixels just below his neck. Then choose a color for his shirt (I chose red), select the Flood Fill tool, and click on his body to fill in the shirt. This is just an easy way to color entire sections.

It's time to give this fine chap some hair. But first, let's change the shape of his face a little. Clear the far left pixel at the top of the head by right-clicking it with the Freehand Draw tool. It will look like you're deleting it, but you're just setting that pixel to the background color so it's no longer visible. For this to work correctly with transparent backgrounds, make sure the background color is set to black. If you're not sure, just right-click black in the top left of the color palette.

Next, draw a straight line at the top of the head. Figure 2-18 shows what Max looks like so far.

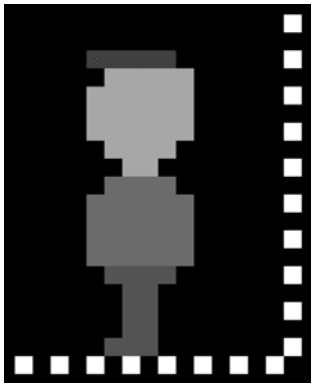


Figure 2-18: Now Max has trousers and a shirt. Note the cleared pixel just below the left of the hairline.

Doodle Some Details

Next, fill in some details in Freehand Drawing mode: add more hair, a mustache, an eye, and an arm. Just make sure you draw with black pixels if you want an element to appear black in your game. If you delete a pixel in GrafX2, it will look black in the editor, but once you export, it will be transparent in Unity. To draw Max's eye, I used a white pixel on top of a dark gray pixel. You can see these details added in Figure 2-19.

Max is looking pretty good, and you could stop here and start animating. But let's add a little extra shading to simulate lighting. Varying the shades of pixels at the edges and corners can make a huge difference and give the sprite more depth.

To simulate lighting effects, imagine a light is shining on your sprite from a fixed position. You can change the color of your pixels to shade your sprite, as though it was being affected by that light. For example, if light was coming from the top right, shadows would be cast down to the lower left. Use darker shades for pixels that are farther away from the light, as shown in Figure 2-20 for a simple sphere.

You don't have to shade everything. Sometimes all it takes to add richness is a subtle color change along the edge of a sprite. For example, I colored the left side of Max's trousers darker than the right so it looks like a shadow is wrapping around the legs, giving them more depth.

The final Max sprite design has a few extra bits of shading in the hair, face, and body (Figure 2-21). I didn't follow any strict rules to add those details; I just experimented until I was satisfied with the overall look. You don't have to be a great artist to experiment, so play around and see what you come up with!

The image file of the final, fully shaded character (*Player_Full.gif*) is in the *Chapter 2/Source Images* folder in the example files for the book, which you can download from *http://www.nostarch.com/2DUnity/*.

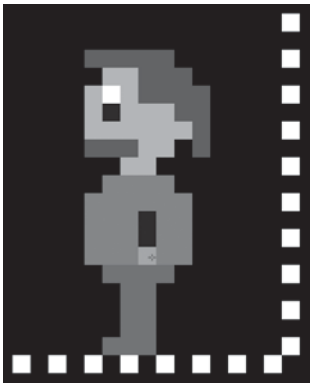


Figure 2-19: The Max character before shading

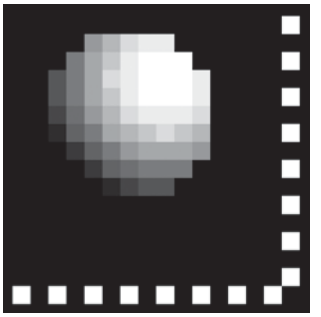


Figure 2-20: An example of a shaded sprite that simulates a light source casting from the top right

Figure 2-21: The completed Max character with some simple shading to add depth

Animate!

Next, I'll show you how I drew a simple walk animation (also known as a *walk cycle*) of just two frames. The two frames for Max's walk cycle are shown in Figure 2-22.

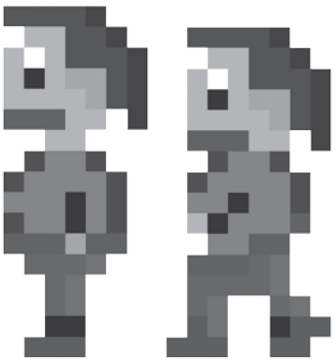


Figure 2-22: The two frames of animation for the Max character sprite

At this point, your character sprite design should be finished. If you want to make any major design changes to your sprite, it's best to do that before you start animating. Otherwise, you'll have to adjust each frame separately. (That wouldn't be so bad for this simple, two-frame walk cycle, but it's a lot more tedious when you're working with more than 50 frames!)

Add a New Animation Frame

Click the **Add Animation Frame** button, which is below the screen size button (see Figure 2-23), to have GrafX2 copy your current image into a new animation frame. Notice that the frame number has increased in the animation information bar; it should read #2 in the third row of tools. You could use the frame navigation buttons to step through the animation, but for now keep the frame number on frame #2.

Figure 2-23: The Frame tools, from left to right: Add animation frame, Delete animation frame, Step back a frame, and Step forward frame

Modify and Animate Graphics

In this second frame, we'll place Max's legs into a stepping position. Do this by using the Brush Grab tool (Figure 2-24), as follows:

Figure 2-24: The Brush Grab tool

1. Select the Brush Grab tool from the toolbar.
2. Select the leg, as shown in Figure 2-25, to make a new brush in the shape of Max's leg. The brush will move around with the mouse pointer.

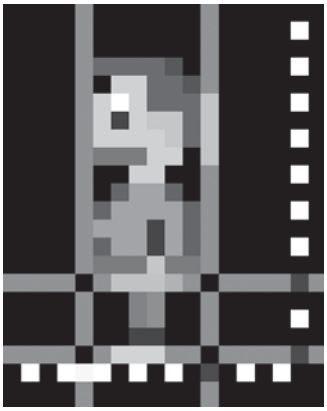


Figure 2-25: Selecting leg pixels using the Brush Grab tool

When you're drawing multiple animation frames, the Brush Grab tool makes it easy to move large chunks of pixels. It allows you to copy sections of the character to paste onto other frames, so you can rework certain areas without having to redraw the entire sprite for each frame. Next, follow these steps:

1. Move the leg-shaped brush on top of the leg in the image.
2. **Right-click** to erase the old leg. Max should end up as just a body, arm, and head.
3. Click the **Brush Effects** button on the toolbar (the FX icon with a dotted box around it). The Effects popup will appear.
4. Select **Rotate any angle** from the Shape Modifications section.
5. The Rotate tool changes the mouse pointer to a box. Drag and drop the box to rotate the new leg-shaped brush so the leg is in a stepping position.
6. When you're satisfied with the angle, right-click to confirm the rotation.
7. For the second leg, click the **Brush Effects** button again to bring up the Effects popup.
8. Select **Rotate any angle** from the Shape Modifications section of the Effects popup.
9. Rotate the brush so the leg is facing in the opposite direction (Figure 2-26).

Figure 2-26: After you rotate the legs, the pixels look pretty messy!

The only downside to this animation approach is that the pixels can get jumbled, so we'll need to tidy things up.

Tidy Up

If the character looks a bit odd, take the time to shift some pixels around to make it look better.

At this stage, you should have two animation frames drawn, and you can now watch your animation. Use the frame navigation buttons in the animation information bar to check how your animation flows. Right-click and hold on the **Go To Next Frame** button (Figure 2-27) to automatically cycle through the frames and see how the character moves.

Figure 2-27: The Go To Previous Frame and Go To Next Frame buttons

When you're working on an animation, it helps to play the animation often. Professional animators do something called *scrubbing*, which is moving forward and backward through the animation to make sure it flows well.

Save the Animation File

Save the image by clicking the **S** part of the **Save/Load** icon. Follow the same steps in "Save Your Work" on page 26, only this time save the image as a GIF file to keep all the image data intact. Saving in another format might save only the first frame, so you'd lose any extra frames.

Generate a Sprite Sheet with Piskel

In this section, I'll show you how to use Piskel, a browser-based pixel editor, to generate a sprite sheet. Piskel is an open-source app that can import GIF files containing multiple animation frames and output them all on a sprite sheet.

Unity ships with the Sprite Packer, which can copy many images onto a single sprite sheet. But the Sprite Packer doesn't work with GIF files, which is the approach I'm taking in this book. The reason I use Piskel is simply that it's much easier to export an animated GIF file into Piskel instead of importing each animation frame individually using Unity's Sprite Packer.

You can even use Piskel to make character sprites from scratch if you prefer, but the extra functionality in GrafX2 works best for me.

NOTE *Piskel is available both offline and online, where it has a lively community. You can create a public gallery to show off your creations to the world!*

Import Your Sprite

Go to the Piskel website at *http://www.piskelapp.com/* and click the **Create a Sprite** button. The Piskel editor window opens, as shown in Figure 2-28. To see what each button does, hover your mouse over it to make a tooltip appear.

Figure 2-28: On the right-hand side of the Piskel editor, there are menu functions, such as the folder icon to import images with.

To import an image, click the folder icon at the bottom of the menu along the right edge of the window. A new menu should open that contains your options for loading files. In the Import from Picture section, click **Browse Images** to bring up a file browser dialog, as in Figure 2-29.

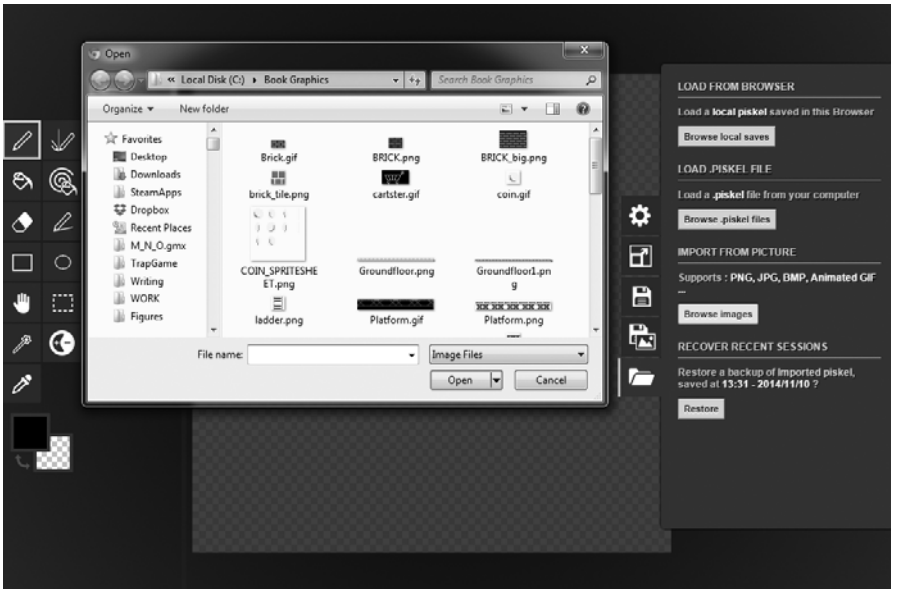


Figure 2-29: The Import from Picture feature allows you to import graphics into Piskel.

Find your Max character file in the file browser and click **Open**. The Import Image popup should appear with import options (Figure 2-30).

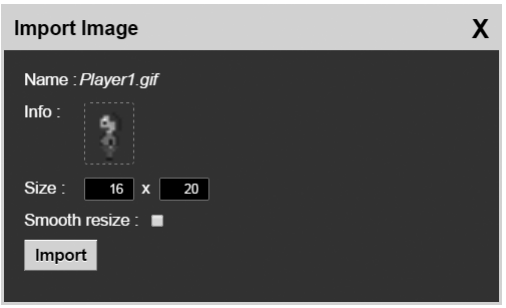


Figure 2-30: The Import Image popup

Uncheck the **Smooth resize** checkbox. The size settings should already be set to 16 by 20 pixels, and you should see a little animated preview of your character sprite. He may look fuzzy in the preview window, but once the file is imported into Unity, he should look just fine. Click **Import** to continue.

Export the Sprite Sheet as a PNG

The editor should now show your player sprite and an animation preview in the top right (Figure 2-31).

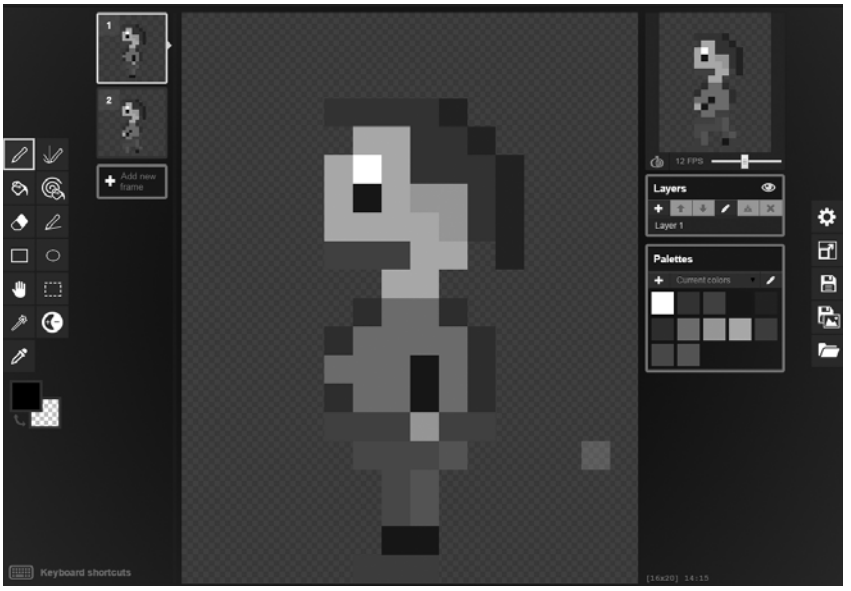


Figure 2-31: After importing the player sprite, Piskel shows the main editing area and animation previews.

Click **Export** (Figure 2-32). It's the fourth button down, the one that looks like a floppy disk with an image in front of it. The Export Image popup should open (Figure 2-33). In the **Export spritesheet** section of the Export Image menu, click the **Download PNG** button to save a completed sprite sheet to your hard drive.

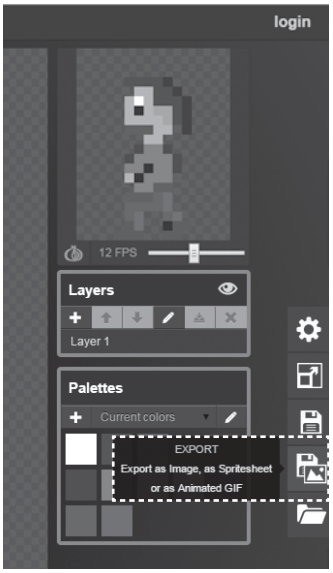


Figure 2-32: The Export Image popup

Figure 2-33: Near the top of the Export Spritesheet menu is the Download PNG button.

Closing Thoughts

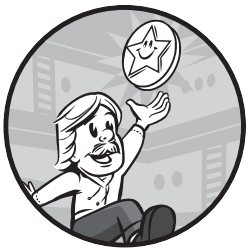
In this chapter, you looked at different ways of obtaining graphics, as well as how to create and export them yourself in GrafX2. You've learned about sprite sheets, which you can use to manage graphics more efficiently, and how to export them using Piskel.

The pixel art techniques I've introduced are just the beginning of a very rich subject. Pixel art *is* an art form, after all. Master these tools and then practice, practice, practice.

The next chapter takes you on a guided tour of Unity's 2D features and some of the main features you'll use to make your games. Oh, and don't forget to keep those graphics handy—you'll use them later in the book!

3

USING UNITY TO ANIMATE 2D SPRITES



This chapter explores how Unity handles 2D projects. You'll learn more about the 2D interface, about the camera, and how to optimize and import images. Finally, you'll animate a sprite.

Cameras

Unity draws everything in 3D, including 2D sprites: the engine automatically builds flat models and displays sprites on them, just like cardboard cutouts. Each sprite is fixed on the same position on the z-axis in 3D space, and an orthographic camera displays the sprites without perspective. When you see the sprites onscreen, they appear to exist in 2D space rather than in 3D space.

When you select the Empty 2D project type to set up Unity's editor configuration for 2D game making, Unity shields you from the complexities that occur under the hood. As a result, you don't have to work around 3D tools that may not have been intended for making 2D games.

Figure 3-1 shows the difference between how Unity deals with sprites in its 3D world (left) and how the orthographic 2D view displays them (right).

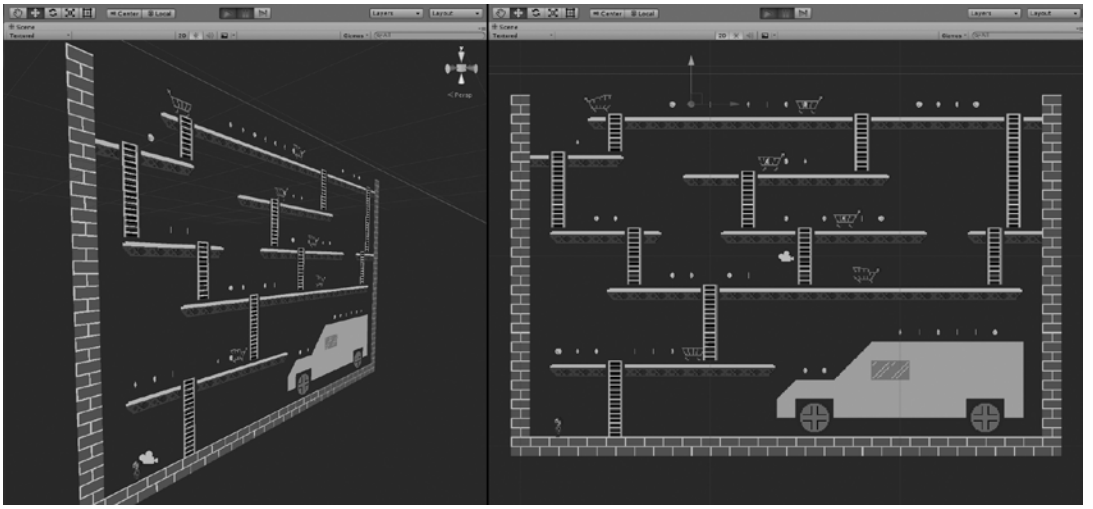


Figure 3-1: How a 2D game scene is constructed in 3D space (left) versus how the game looks through an orthographic camera (right)

Choosing Empty 2D sets the Scene view to 2D. It's possible to switch between 2D and 3D views by using the 2D button, which is the small button labeled 2D just above the Scene panel. Because you're making 2D games in this book, you won't need to use this button.

A game is like a movie in that players see all the action through a camera. Every scene needs at least one camera, and you can use multiple cameras for different views. In Unity, a camera is a GameObject with a Camera Component attached to it. Both 3D and 2D games use the same type of camera, but the Camera Component's properties are set up differently for 2D games. See Figure 3-2 to view these properties in the Inspector panel.

When Unity is set up to build a 2D project, all cameras default to 2D settings. In Figure 3-2, the Projection setting ❷ is set to Orthographic and is followed by the Size value. The Size value ❸ affects the viewing volume of the camera. Increasing the Size property zooms out of a scene, and decreasing its value zooms into the scene. You'll have the opportunity to experiment with this value later in the book when you make your own scenes.

When more than one camera is in the same scene, the Depth value ❹ decides the order in which to draw the camera views. A camera with a lower depth draws what it sees to the screen first, and cameras with higher depth values draw what they see on top of that. If you overlay cameras on top of each other in this way, you can build a screen made up of several views. For example, you may want one camera to draw a background, another camera to draw clouds on top of the background, and a third camera to draw the main game. To do this, you could use three separate cameras and adjust each camera's Depth setting to arrange them appropriately.

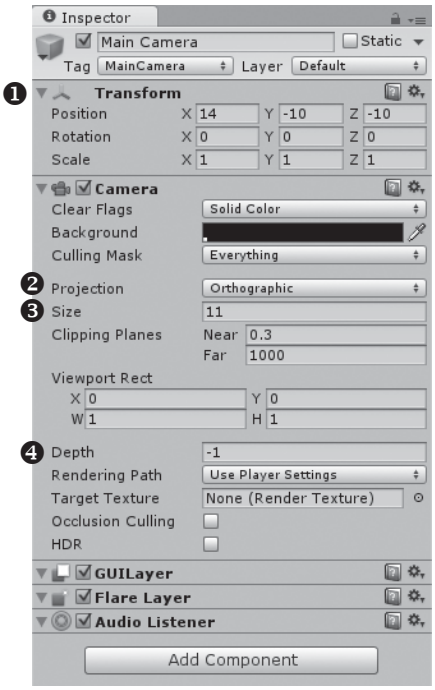


Figure 3-2: A typical Camera Component setup for a 2D game

Also, note in the Transform section ❶ in Figure 3-2, the Position along the z-axis is set to –10. If this value were set to zero, the camera would be at the same position on the z-axis as the sprites and the camera would look *past* the scene. At –10, the camera is positioned so it can *see* the scene. Keep this setting in mind: if you find your camera isn't rendering your 2D scene, check the Position z-axis.

Now that the editor set up for making a 2D game, let's look at how to get sprites into the editor and how to move, scale, and rotate them onscreen.

Importing Images

Open Unity and click the **New Project** button. Name the project *2dTour*, select a location to save the project, and then choose the **Empty 2D** project type. Click the **Create** button to get started.

After the project loads, you should see the standard **2 by 3** view (Figure 3-3); if not, select it from the Layout drop-down menu at the top right of the interface. This is the layout you'll use in this section of the book. Open an Explorer window (or Finder on a Mac) and browse to wherever you saved the brick tile image you created in Chapter 2. Or you can grab the *brick_tile.png* image from the *Images* folder in the example source files included with this book (see *http://nostarch.com/2DUnity/*).

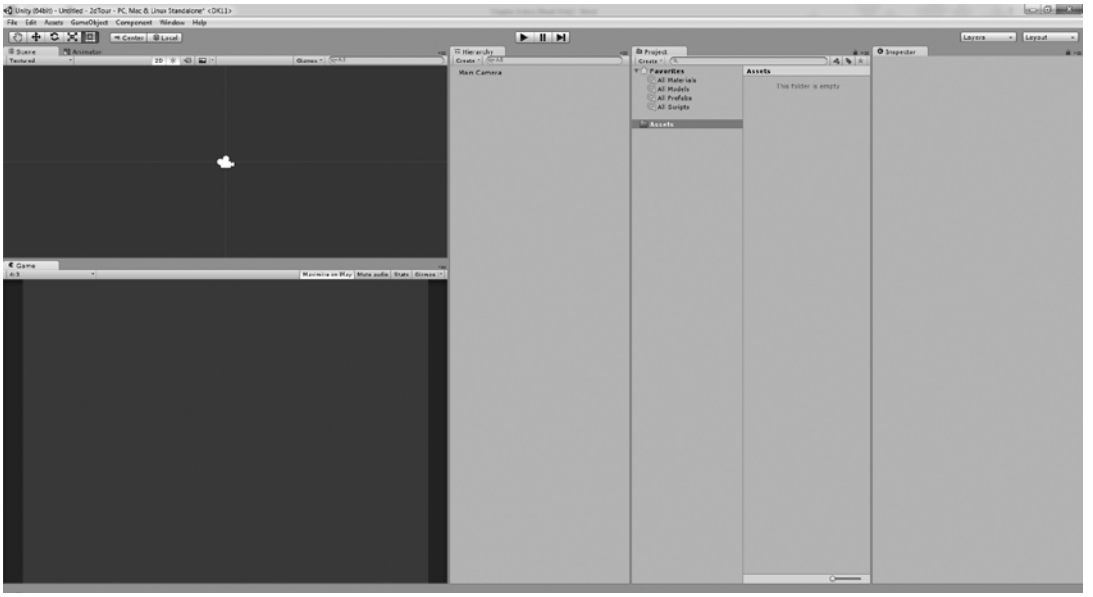


Figure 3-3: The 2 by 3 view in the Unity editor

To import the *brick_tile.png* file into Unity, drag it from the file browser window into the Project panel's *Assets* folder. Unity will import it for you. Another way to import the image is to right-click on the Project panel's Assets folder and select **Import New Asset** from the menu.

Click the newly imported `brick_tile` object in the Project panel; the Inspector panel shows all the import options for this type of asset (Figure 3-4). A preview of the sprite is shown at the bottom right of the panel.

Let's look at how the Pixels To Units setting affects the `brick_tile` sprite. In Unity, scale is unspecified. It is measured in units, which you can choose. A unit could be a pixel, a meter, a light year, or another unit measurement: it doesn't make a difference to the engine. In this case, let's set it to 16 so one of our pixel bricks is one unit. In the Game panel, the brick sprite appears in the center. It's small right now because the default Pixels To Units size is 100 pixels per unit. Although this size works for games with detailed graphics and larger images, it doesn't work for the simplistic retro graphics we'll be using in this book.

First, click the `brick_tile` item in the Project panel's Assets section. Then click and drag the item into the Hierarchy panel to add it to the scene. To make the brick sprite bigger, click the `brick_tile` item again. In the Inspector panel, change the Pixels To Units value to 16. Now, one brick will be a unit wide and a unit high.

Click the **Apply** button in the Inspector panel. The `brick_tile` sprite becomes much larger in the Game panel. Perfect! The number of units your sprites take up in the Scene also impacts how your objects react to your game's physics simulation, which I'll discuss later in the book.

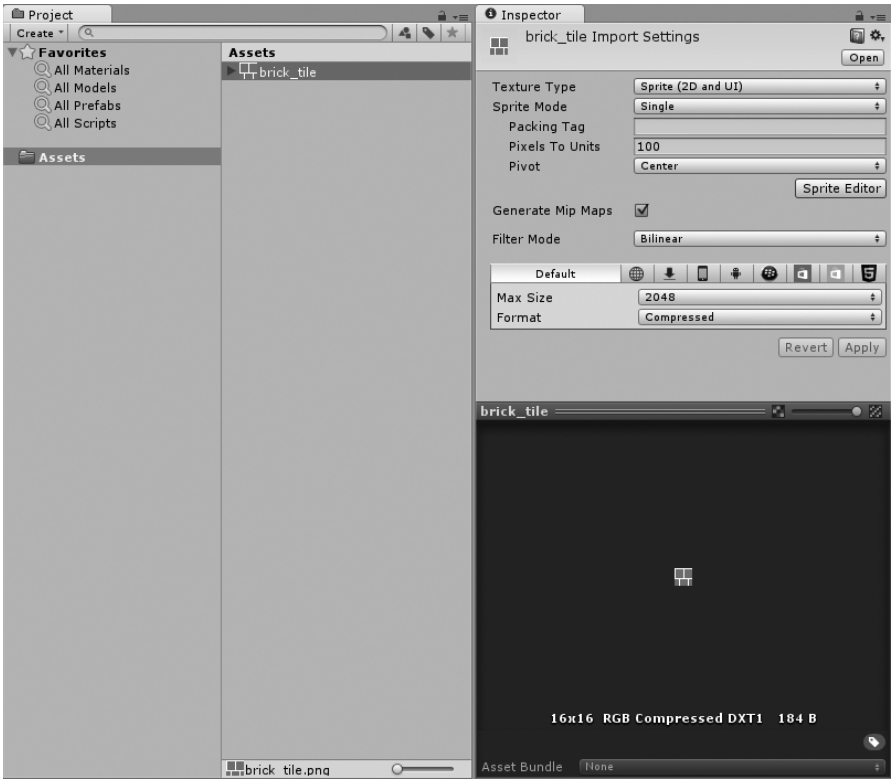


Figure 3-4: The Inspector window shows the Import Settings of the selected `brick_tile` asset.

To display the sprite in your game, click the `brick_tile` GameObject in the Hierarchy. The Inspector panel should show its Components and properties. Two Components are attached to the `brick_tile` GameObject: a Sprite Renderer and a Transform.

The Sprite Renderer Component draws the sprite onscreen. If you removed that Component, the GameObject would still exist in the Hierarchy, but the brick sprite would no longer be visible in the Game or the Scene views.

All GameObjects have the Transform Component, which stores and accesses position, rotation, and scale information. You can change Transform properties in the Inspector panel or alter them visually in the Scene panel using the Rect tool (on the right in Figure 3-5). The Rect tool acts as an all-in-one positioning, scaling, and rotation tool specifically for 2D GameObjects.

Click the **Rect** tool now; handles should appear around the `brick_tile` GameObject in the Scene panel. You can drag the handles to change the scale and rotation of the sprite, or you can click inside the sprite to drag the entire image. The Rect tool is the fastest and easiest way to modify your 2D graphics.

Figure 3-5: The Rect tool on the right is the selected tool in the Scene tools.

Although you can scale sprites using the Rect tool, when you're developing games for devices with limited memory, such as mobile phones or tablets, choosing the correct sizes for images before you import them into Unity is best for performance and memory use.

Optimizing Your Images

The `brick_tile` image you imported into the editor is 16 by 16 pixels square, which is a power of 2 ($16 = 2 \times 2 \times 2 \times 2$). If you try to import an image with dimensions that aren't powers of 2, such as the *player_spritesheet.png* you made in Chapter 2, it will import successfully, but the editor will warn you that the image can't be compressed.

Powers of 2 are very efficient as image dimensions in computer graphics. It takes less computation to manipulate the numbers that make up their underlying data, which means graphics cards can handle these images faster. As the speed of modern graphics cards increases, performance becomes less and less of a problem. But if you're targeting mobile platforms, efficient image dimensions can make a huge performance difference. Using images with dimensions that are not a power of 2 can cause your game to slow down, and because you're unable to compress these images, they'll take up a lot more memory.

The *player_spritesheet.png* you use in this book is not optimized for mobile devices. But because you'll be using just a few small images, the performance difference won't be noticeable.

NOTE *If you want to rescale* player_spritesheet.png*, load the image into a paint program (such as Paint.NET) and change the size of the canvas size to a power of 2.*

You've used the Import Settings to get sprites onto the screen, but you'll need to know more about Import Settings to do complex tasks, like animation.

Import Settings

To explore the Import Settings in more depth, use the *player_spritesheet.png* image that you created in Chapter 2 (you can also find it in the *Images* folder of the book's resources). Right-click the Project panel's Assets section in the editor. Click **Import New Asset** and then find and import the player animation file.

Recall that the image is a spritesheet containing multiple images in the same graphic, so you'll tell Unity to split the graphic into the frames of animation you want it to play.

If `player_spritesheet` isn't highlighted in the Assets section, click it to highlight it. The Inspector panel shows *player_spritesheet.png*'s Import Settings (Figure 3-6).

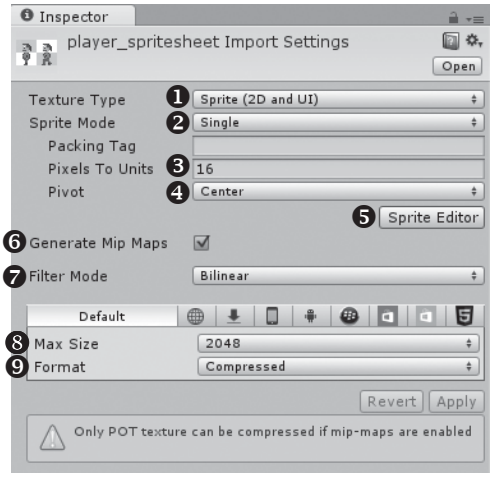


Figure 3-6: Import Settings for player_spritesheet.png

Any changes you make to these settings won't be saved until you click the Apply button, and you can undo changes by clicking Revert. But not that clicking Revert only brings back the version you last applied. The Revert and Apply buttons are grayed out until you actually make modifications.

Let's start at the top and work through the Import Settings for 2D games that you'll be using most often in this book.

Texture Type

When you've chosen to build a 2D project, the default option, and the one you want to use, is Sprite (2D and UI) Texture Type ❶. This setting is the only texture type you'll need in this book. The following settings appear only when the import texture type is set to Sprite (2D and UI).

Sprite Mode

The two options for Sprite Mode are Single or Multiple ❷. Graphics that contain a single image (such as the `brick_tile` sprite) should be set to Single, whereas graphics that contain multiple images—like the *player_spritesheet.png*—should be set to Multiple. I'll discuss the Multiple option in more detail in "Character Animation" on page 45, where I talk about player animation.

Pixels To Units

Recall that the Pixels To Units setting ❸ was explained in "Importing Images" on page 39. To give you a quick recap, this value specifies the number of pixels that make up a single unit in *Unity space*.

Pivot

The Pivot setting ❹ lets you choose where to place the pivot point for a sprite. The *pivot point* is the point at which a sprite will rotate. By default, the pivot point is in the center of the sprite, but you can choose another typical pivot point locations or you can even use the Custom option to place the pivot point manually. Having a flexible pivot system comes in handy when you need to rearrange your scenes to exactly.

The Sprite Editor Button

Clicking the Sprite Editor button ❺ brings up a visual editor for trimming unused space around a sprite, positioning a custom pivot point, or slicing up a single image into multiple sprites. I'll discuss the visual editor in more depth in "Slicing Spritesheets Automatically" on page 45.

Generate Mip Maps Checkbox

When you select the Generate Mip Maps checkbox ❻, the engine creates multiple sizes of texture automatically and uses the lower-resolution images for objects that are far away from the camera. It increases rendering speed and reduces some of the quality loss that occurs when you're trying to render large images at a small size. With the relatively simple graphics we'll use in this book, mip maps don't make much difference. Also, it's best to turn them off when you're not using them so the system won't generate and store unnecessary images. In future game projects, if you decide to move the camera in and out of a 2D scene filled with complex graphics, you may want to experiment with this setting.

Filter Mode

Graphics used in a game are usually filtered or smoothed in some way. Processing images with filters helps to make images look smoother when they're drawn at different sizes on the screen. Without filtering, images can sometimes appear pixelated or even have small artifacts left over from rescaling or image compression processes. Your filter choices are Point, Bilinear, or Trilinear ❼. I like to think of these as meaning "no smoothing," "normal smoothing," and "highest-quality smoothing," respectively. This book uses Point because it doesn't filter or smooth out the images, which is perfect for that crunchy pixel look!

Max Size and Formats

Selecting a maximum size ❽ for images will cap your images at that size automatically if any of them happen to exceed this setting.

The Format drop-down menu ❾ provides you with different methods of storing the image depending on the platform you're targeting. For desktop games, you can choose from Compressed, 16 bit, and TrueColor. When

I'm using images no bigger than a few hundred pixels (like the ones in this book), I'll always choose TrueColor for quality, even though it consumes more memory. The amount of memory these tiny images use is minimal. But if you use lots of graphics in future projects, you might need to compress some or all of them to reduce file size or memory use.

If you're producing games for console or mobile devices, you'll see a lot more image choices depending on the target device. As they're specific to those target devices, I won't be going into detail about them here.

With a better understanding of the Import Settings and some of the available options for getting your graphics into the engine, it's time to bring these settings to life. Next, you'll have Unity split up the Max character spritesheet and make Max walk!

Character Animation

Prepare to shout out "It's alive!" just like Dr. Frankenstein when you're done with this section. You'll slice the animation spritesheet you created in Chapter 2 and play it as an animation in Unity.

Let's start by setting up the Import Settings for *player_spritesheet.png*.

1. Select `player_spritesheet` in the Assets section.
2. In the Inspector panel, change Sprite Mode to **Multiple** to tell Unity you want multiple sprites from this single image.

The default Filter Mode will make your images fuzzy. For your tiny sprites, you'll need to use a mode that doesn't smooth out the image:

3. Click the **Filter Mode** drop-down menu and change it to **Point**.
4. Click the **Apply** button.

Next, you need to tell the editor how to slice up this image to create multiple sprites, which is done in the Sprite Editor. Click the **Sprite Editor** button to open it.

Slicing Spritesheets Automatically

When you open the Sprite Editor, a window appears with a preview of your image and a few buttons surrounding the window. A **Slice** button and **Trim** button are at the top left and are initially grayed out. To the right of these buttons are the **Revert** and **Apply** buttons. These apply changes or revert to a saved version when you want to undo a change.

First, you need to tell Unity how you want to slice the image:

1. Click the **Slice** button to bring up the Slice menu (Figure 3-7). Set the default slice Type to **Automatic**. In Automatic, the editor will guess where the edges of sprites are.

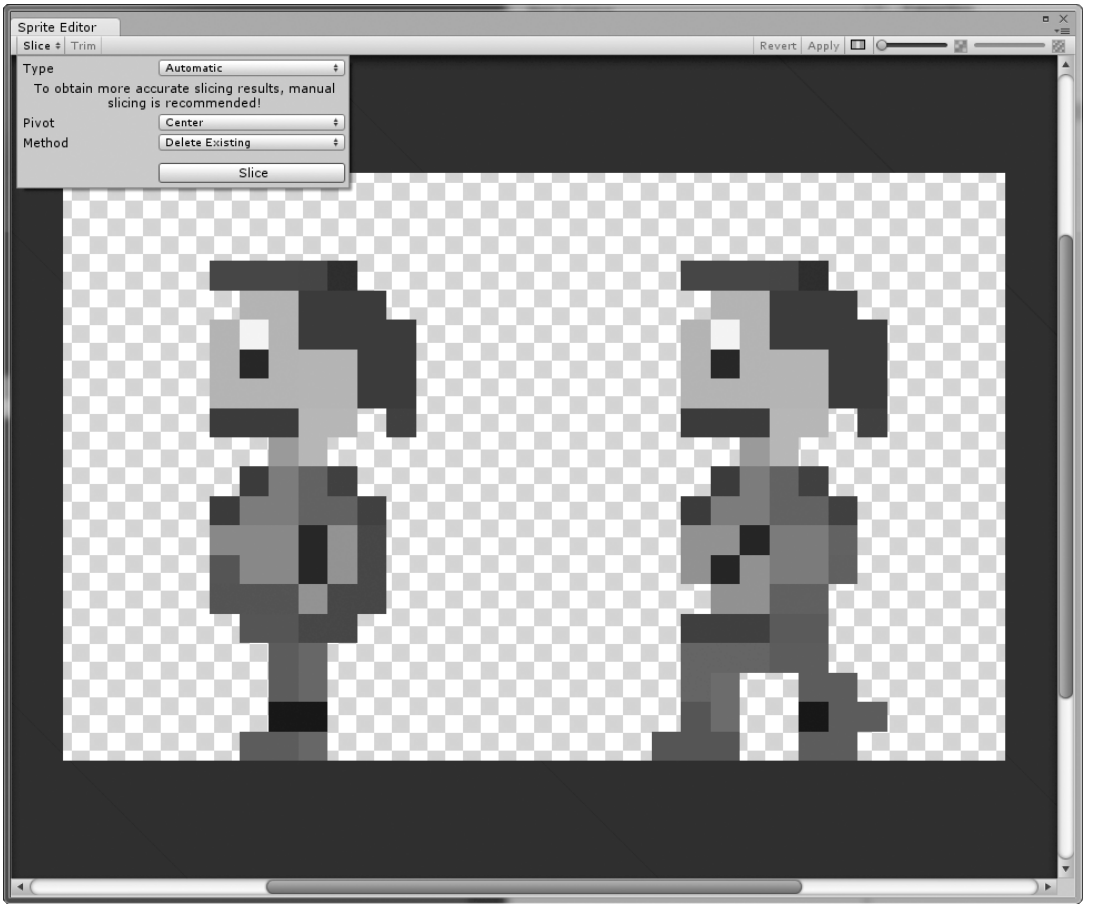


Figure 3-7: The Sprite Editor and the Slice menu

2. Click the **Slice** button again. Some lightly colored rectangles (which can sometimes be difficult to see) should appear around the areas that Unity thinks are your sprites. These rectangles are the slices.
3. Click the **Apply** button. The slices will be used to generate two new sprites that will become visible in the Project panel's Assets section. In the Project browser, a drop-down arrow appears next to the `player_spritesheet`. When you click the arrow, two new sprites should be displayed from the images in your spritesheet.
4. Close the Sprite Editor.

To make sure the images have been sliced correctly, click either one of the new sprites in the Assets section to see a preview in the Inspector.

Now let's combine these individual images into an animation.

Create an Animation File for Your Character

Drag and drop the `player_spritesheet` from the Assets section into the Scene panel. Usually, you can drag items into either the Scene panel or Hierarchy panel to add them to the scene, but this is one of the rare cases in which the outcome will be different for each panel. So make sure you drag the `player_spritesheet` item directly into the Scene panel, not the Hierarchy.

From the Create New Animation dialog that opens, choose where to save the animation (Figure 3-8). By default, the filename is *New Animation.anim*. Change it to *playerwalk.anim* and click **Save**. The default save location for animations is your project's *Assets* folder. Later in the book, you'll learn better ways to organize and manage your files. For now, as long as the file is stored in the project files, Unity will be able to use it.

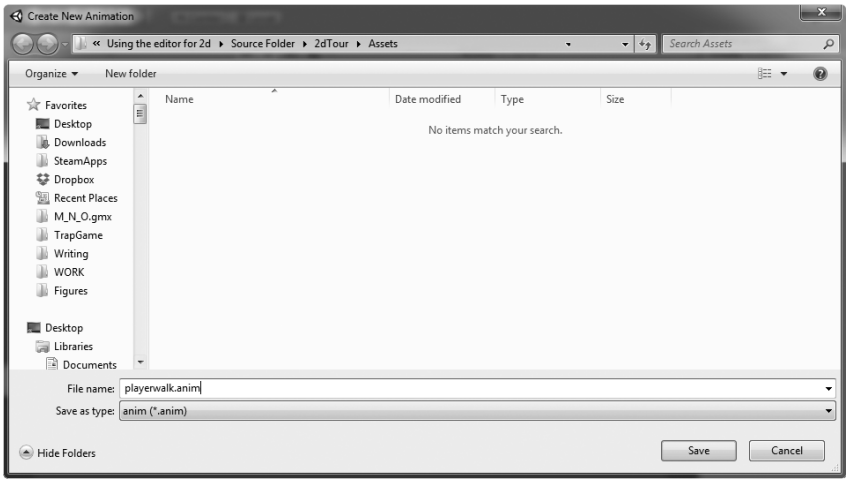


Figure 3-8: The Create New Animation save dialog

Now you should be able to see `player_spritesheet` in the Scene panel, the Hierarchy panel, and the Game preview. Click the **Play** button to start the game, and you'll see Max walk!

Automatic slicing can be very helpful, but notice that Max jitters forward one pixel in the second frame. The reason is that automatic slicing didn't slice the image quite right. Max looks okay when he isn't moving, but he might look strange when he walks around a game level. To fix this, you'll need to keep his body in the same place between frames, which is a job for the manual slicing method in the Sprite Editor.

Slicing Spritesheets Manually

Using the manual slicing system requires some extra time, but it's often the best way to make sure your animations display properly and the images are sliced correctly.

Click `player_spritesheet` in the Assets section to bring up its Import Settings in the Inspector. Next, click the **Sprite Editor** button. Figure 3-9 shows that the automatic slices you made earlier are tight around the two images.

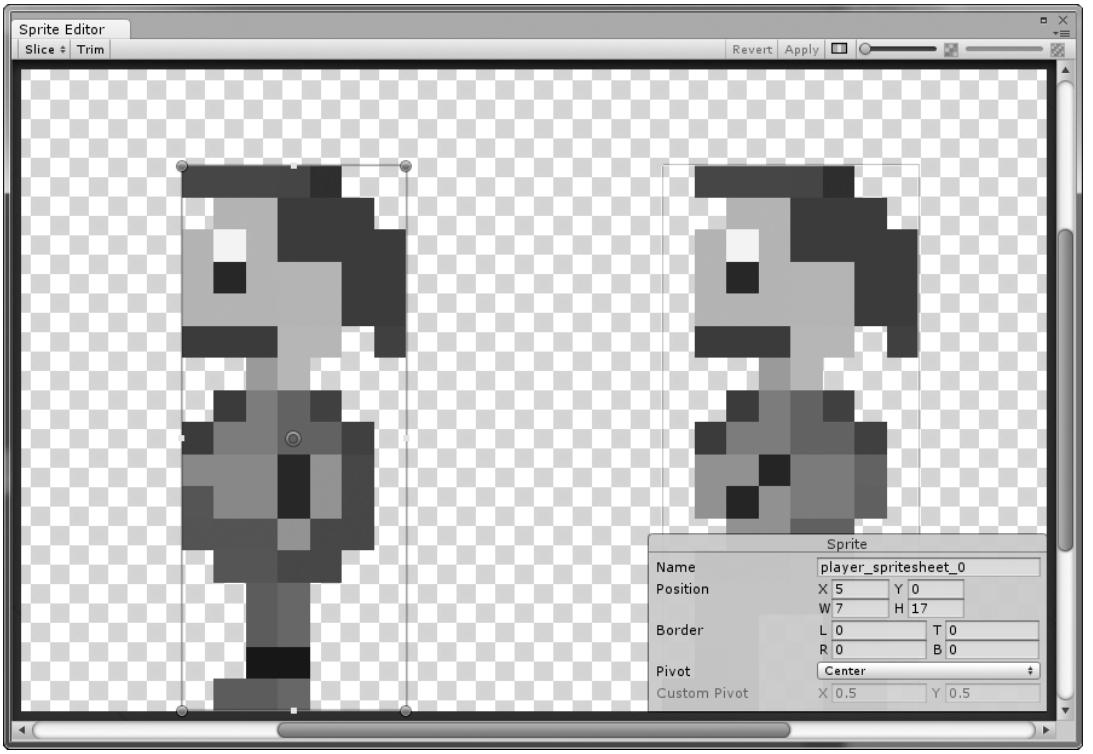


Figure 3-9: The Sprite Editor showing slices around the two sprites in player_spritesheet.png

From the top left of the Sprite Editor, click the **Slice** button. Change the Type from Automatic to **Grid** in the drop-down list. Next, click anywhere on the right image (the one of Max in the stepping pose) to select the slice. Handles will appear around it.

The Sprite information panel at the bottom right of the Sprite Editor provides some useful data about slice parameters. Similar to the Inspector panel, you can enter values to rename the slice, view or set its Position and size, set a Border, or select the position of the slice's Pivot point.

The Sprite information panel shows you that the width of the image slice on the right side of the spritesheet is 8 pixels. Click the image slice on the left and you'll see in the information panel that its width is 7 pixels. This slice is one pixel narrower. To prevent the sprite from jittering between frames, you want to make the boxes around each image the same width.

Let's make the left slice a bit wider to match the width of the right slice. We can do that in the Sprite information panel:

1. With the left slice selected, click the **Position X** box and enter `4`. The slice rectangle around the image should move one pixel to the left.
2. Click the **W** box and enter `8`. The right side of the slice rectangle will move a pixel to the right.

NOTE *It's also possible to set these values visually by dragging and dropping the boxes. I chose to enter number values because it's easier to explain.*

Click the **Apply** button at the top right of the Sprite Editor to save the changes you've made to the slice.

Click the **Play** button in the Unity editor. Now when Max walks, he should stay in the same place without the single-pixel wobble. Awesome!

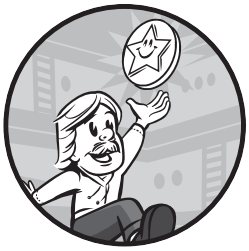
Closing Thoughts

In this chapter, you discovered how Unity handles 2D projects. You imported a spritesheet into Unity, sliced it up, and created an animation in Unity. You also corrected a small glitch caused by the automatic slicing tool by slicing the sprites manually. You'll use all of this information to make the platform game coming up. Everything you've learned so far will also prepare you for creating great games in general.

In the next chapter, you'll learn to program in C# and explore some of the core concepts you'll need to be a full-fledged Unity game developer.

4

INTRODUCTION TO PROGRAMMING



In this chapter, we'll start to breathe some life into our Unity games with code. I'll introduce you to some C# programming basics and object-oriented programming. By the end of the chapter, you'll have programmed your own brick-breaking game using Unity scripts, loops, and variables.

You won't come away from this chapter knowing everything there is to know about programming. Programmers become skilled through a mix of knowledge and, more important, practice. I started programming in the 1980s by typing and running game source code from magazines, which inspired me to create my own games. Many of my early games didn't work well, but I kept practicing and eventually learned how to make my own games.

This chapter is part reference and part tutorial. I'll teach you how the example code works; there's more to programming than typing someone else's code, so I hope you'll be inspired to practice and learn more after completing this book.

This chapter is divided into three small projects. To begin your journey, you'll open the example Unity project for this chapter and make a ball bounce around the screen. In the second project, you'll make a player-controlled bat hit a ball, and in the third project, you'll create bricks to make a brick-breaking game. In each project, you'll learn some technical skills for making games and have a little fun along the way!

What Is C#?

Unity supports two programming languages: JavaScript and C#. This book uses C# (pronounced C sharp). Both languages have their pros and cons, but C# is the language used by most Unity-powered game studios and is my language of choice.

NOTE *C# was originally named Cool, which stood for C-like Object Oriented Language. It is rumored that the name was changed before launch due to copyright reasons, but I wonder if it was just too cool for all those business application developers to take seriously!*

C# is an *object-oriented* programming language, which means it's based on the principle of constructing code and data using objects. In programming, *objects* are scripted Components that are combined to do something, and making those combinations is what programmers mean when they refer to object-oriented development.

Getting Started

The three projects in this chapter are all in a single Unity project file. Click **File ▸ Open Project**, browse to *Projects/Example Files/*, and select the example project for this chapter. If you ever get stuck or would rather follow along with the full code, you can look at the finished project files.

In the Project panel's *Scenes* folder, you should see three scene files named *Part 1*, *Part 2*, and *Part 3*. Double-click the scene named *Part 1*. With *Part 1* open, notice that there's just a ball in the scene. When you click **Play**, it doesn't move. You'll program it to move!

Bouncing a Ball

Let's open a script to get started. Scripts are easy to identify in your Project panel because Unity displays a small C# icon next to them along with a *.cs* file extension.

In the Project panel, open the *Scripts_1* folder and double-click the *SimpleBallControl* script. It will open in MonoDevelop, which is the default program for writing and editing scripts in Unity.

NOTE *As of version 5.2, the Unity installer software offers Microsoft Visual Studio as a free alternative script editing program for PC users only. The script development processes shown in this book are the exact same for either Visual Studio or MonoDevelop, but I've chosen to stick to MonoDevelop.*

Libraries

Let's look at the first two lines of the program:

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

These two lines are automatically added whenever you create a script. They tell the Unity engine about any additional code *libraries* that the script needs access to. Libraries are built-in collections of code that handle important technical information. They let your scripts talk to the engine and provide Unity-specific classes like `GameObject` and `Transform`, as well as other important elements you'll need for your games.

Notice the semicolon at the end of both lines. The semicolon is commonly known as a *terminator*. In C#, the semicolon tells the engine where a *statement* ends. A statement is a single piece of code (normally a single line of code). You must end each statement with a semicolon so the engine knows where one statement ends and another begins.

Let's move on and discuss class declaration.

Classes and Inheritance

C# code is split into *classes*, which are essentially chunks of code. Note that the script is not the class; a class is a section of code inside a script. A single script could contain many different classes. To keep things simple, each script file in this book will contain just one class.

Take a look at the third line of code in the *SimpleBallControl* script:

```
public class SimpleBallControl : MonoBehaviour {
```

This is a *class declaration*, where you decide how the class can be accessed from other scripts, declare what type of class it is, and name the class. The word `public` tells the engine that this part of the script is accessible from "outside" this script, which is a process known as *scoping*. I'll discuss scope in more detail later in "Variable Scope" on page 55.

After `public`, the word `class` tells the engine that you're making a class. Now you need to name your class. The class name is mandatory, because it separates the chunk of code between the curly brackets from any other code you'll write.

The name of the class here is `SimpleBallControl`, which is the same as the filename for the script file. When you create a script, the skeleton code that Unity adds will automatically assume the script's filename as its class name, but you can change it if you want to. It isn't essential for the filename and

class name to match, but it's always useful to name script files descriptively so you know what's inside them just from reading names in Unity's project file browser.

After `public class SimpleBallControl` is a colon and another class name, `MonoBehaviour`. `MonoBehaviour` is one of Unity's built-in classes that handles important Unity functionality. Including `MonoBehaviour` like this makes `SimpleBallControl` have all of the functionality of `MonoBehaviour` in addition to whatever functionality you'll program into that class. This is called *inheritance*. In your class declaration, you can tell Unity that your new class should use, or inherit, the properties of another class.

As an analogy for inheritance, imagine that your favorite breakfast cereal is called Unity-O's. Now imagine a new product comes on the market called Peanut Butter Unity-O's. The peanut butter version inherits all of the properties and behaviors of your favorite breakfast cereal: it has the same consistency, crunch, and is fully compatible with milk. The only difference is that it includes peanut butter.

You can do the same with classes, essentially setting up your new class (`SimpleBallControl`) to have all the functions and properties of the parent class (`MonoBehaviour`) without your having to retype all of the content from the `MonoBehaviour` class. Inheriting from `MonoBehaviour` also allows you to use your scripts as Components.

At the end of the class declaration is a curly bracket. Curly brackets wrap up chunks of code. You tell the engine where the code starts with a { and where it ends with a }. For example, the curly bracket after the class declaration indicates where the code for the class starts, and the curly bracket at the end of the code indicates where the code for the class ends.

Variables

When programming, you'll spend a lot of time manipulating data. For example, to get the ball to bounce around your screen, you'll modify data about its position and its speed using *variables*, which store data.

The *SimpleBallControl* script uses several variables to make the ball move. To tell the engine about a variable, you declare it in a similar way as the class. Let's add our first variable. Add the following variable delcaration below the `SimpleBallControl` class delcaration:

```
public int moveSpeed = 10;
```

I've named this variable `moveSpeed`. When naming variables, you should use names that reflect what the variable holds. Names like `thing1` and `thing2` won't help you solve problems in the code. Instead, it's best to be descriptive and name variables in relation to their use, such as `bounceCounter` for counting bounces. Notice that I start variable names with a lowercase letter and capitalize every word thereafter. This pattern is called *camel case*, and makes it easy to glance at my code and identify where my variables are being used.

This variable also has a scope (`public`) and a data type (`int`), which I will discuss in greater detail in the following sections.

NOTE *You may have noticed that this line of code is indented. In programming, indentation is important because it's used to group code together. When you're writing scripts in MonoDevelop, indentation is added automatically. Just type the code, and when you press ENTER on the keyboard, the code will be indented.*

Variable Scope

When you declare a variable or a class, you decide how accessible it is to other scripts or other parts of the engine. This is known as *scope*. `moveSpeed` is set to a public scope, which means other scripts with access to this class can access this variable. For example, later in the book you'll use a game control script that will need to find out the name of the current level. The level-builder script stores the level name as a public variable so the game control script can access it to display it on the screen.

If your class is derived from `MonoBehaviour`, when the script is attached to a GameObject, you'll also be able to access public variables in the Inspector, right inside Unity. This is ideal for scripts where you may need to tweak variables to try out different values as the game is running, because you can change them in the Inspector without having to pause the preview.

So far you've seen a public class and a public variable, but there's also *private* scope, which means the variable will be accessible only within the class it is declared inside. In addition to public and private, variables and classes can be *static*, which means the variable belongs to the class but is available to any other class.

An example of a static variable might be one that stores a score. A class that contains all the code for running the game could hold the player's score in a static variable. Other classes can access the score or modify it. The static variable will always belong to the game control class, but it could be modified elsewhere by other scripts.

Let's consider a more complicated example. Let's say you have 10 GameObjects that all have the same script attached to them. The script has a public variable in it. Each public variable is dedicated to the GameObject the script belongs to, say health. If you hit 1 of those GameObjects, it loses some health, but the other 9 aren't affected. But if that public variable were static, hitting 1 of the 10 GameObjects would decrease the health of all of them because they share the same public static variable.

Data Types

After you set the scope of your variable, you then need to tell the computer what you'll store in the variable. This process is called *typing*. You can think of variables as storage boxes. Just like there are different types of boxes to store different types of items, there are different types of variables to store different kinds of data.

You'll see a few types of variables in this book, but for now I'll just cover the three main ones: numbers, strings, and Booleans. Let's look at numbers first.

Numbers

I use two types of number variables in this book: integers and floats. *Integers* (`int`) are whole numbers without decimals—for example, 1, 2, 3, and so on. *Floats* include decimals—for example, 3.14159265359. The variable `moveSpeed` that we defined earlier is an integer. The `moveSpeed` integer-type variable will set the speed of the ball.

Add the following line after the `moveSpeed` variable declaration to set up another integer variable called `bounceCounter`:

```
public int bounceCounter;
```

The `bounceCounter` variable will count how many times the ball bounces.

The method for setting floats is a little different from setting integers. You write the scope, followed by the keyword `float`, followed by your variable's name. Use an equal sign (`=`), followed by the letter `f`, to set its value. The *SimpleBallControl* script needs four float-type variables. Add the following four variable declarations under the `bounceCounter` variable:

```
private float boundaryLeft = -17f;
private float boundaryRight = 17f;
private float boundaryTop = 13f;
private float boundaryBottom = -13f;
```

These variables specify the screen's boundaries. The four boundary variables are all declared as public so you can update them in Unity's Inspector pane without having to open the script. Note that in these variables, I'm referring to positions in units, not pixels.

Modifying Number Variables

Now let's learn how to manipulate float and integer variables. To add or subtract from a variable, you can use this shorthand:

```
bounceCounter += 1;
bounceCounter -= 1;
```

This construct might look a little strange if you've never seen it before, but it's very common in programming. Basically, you're setting a variable to its current value plus or minus a number. In this example, we're adding (`+=`) and subtracting (`-=`) 1 from `bounceCounter`, but you could add or subtract any number.

If you just need to add or subtract 1, you can use another convenient shorthand. In the example project's *SimpleBallControl* script, I keep track of how many times the ball bounces by using `++` to add 1 to the `bounceCounter` variable each time the ball changes direction:

```
bounceCounter++;
```

You'll see this in action later in the chapter. You can decrease a variable by 1 using similar shorthand:

```
bounceCounter--;
```

Next, we'll look at how to store and modify text.

Strings

In the example game, I create a string to display how many times the ball bounces. A *string* stores text or symbols. Strings can hold all kinds of text, from player names to the names of planets in a space game. Here's what a simple greeting looks like:

```
string myGreetingString = "Hello world.";
```

To set a string, first write `string`, followed by a name for your string variable. Then, using `=` , set the string by adding quotes around text you want the string to display.

You can also join strings, which is called *concatenation*. In the last chunk of code in the ball bounce script, I combine a string and an integer to display the current number of bounces:

```
        string bounceString = "Bounces " + bounceCounter;
```

Here, I declare a string named `bounceString` and set it to the text "Bounces." The next part of the code adds the value of the integer variable `bounceCounter` to the string. The engine will automatically convert `bounceCounter` into a string for you and join the two strings together to produce something like `Bounces 3` on the screen.

Booleans

There is another type of variable known as a Boolean variable, which has just two possible states. A Boolean may be set to either true or false. In the next section, you'll look at game logic that works in a similar way to how a Boolean variable works: the result of a statement can either be true or false.

Why the funny name? The Boolean type gets its name from George Boole, who developed an algebraic system of logic in the mid-19th century. As he wouldn't have had a computer back then, there was no way he could have known just how important Booleans would be for videogame programming! Game developers use them in just about every project.

Game Logic

All right, let's use what you've learned so far! You'll add four `if` statements to the *SimpleBounceControl* script to make the ball bounce off the walls. An `if` statement checks conditions and runs code based on the result.

You'll add these `if` statements to the `Update` function, so find the `Update` function in the `SimpleBounceControl` script. A *function* is a chunk of code that does something. A function might count coins, update a player's position, or shoot lasers out of a cow's eyes. Enter the following code to make the `Update` function look like this:

```
// Update is called once per frame
void Update () {
     myRB.velocity = moveDirection * moveSpeed;
     // Move right
     if❶ (moveDirection.x == 1❷ &&❸ myTransform.position.x >= boundaryRight) {
            moveDirection.x = -1;
            bounceCounter++;
     }
```

Lines that start with two forward slashes (`//`) don't do anything—they're *comments*. Programmers usually write comments between lines of code to explain how the code works. Comments may be for the benefot of other programmers or for the benefit of thep rogrammer herself when returning to the code in the future. It's good practice to write lots of code comments. Here, one comment tells us how often the `Update` function is called and another indicates where the code deals with moving the ball to the right.

This code checks the position and direction of the ball to see if it's hit the right-hand wall and should bounce off it. To do that, the code compares values using `if` statements. `if` statements can compare variables to see when things change, when limits are reached, or when items are picked up. They are used in most programs and games.

To make a function, you first have to tell the engine what the function is and how it should work. This is known as the *function declaration*. A declaration starts with what, if anything, this new function will return as output. If the function doesn't return output, we use the keyword `void`. If the function should return output, all we have to do is declare the function with the type of data it will return. In this case, `Update` doesn't return anything, so it is declared as a `void` function.

To write an `if` statement, use `if` ❶, followed by the condition you want to check wrapped in parentheses. In this case, we're comparing two pairs of variables. Look at the first pair at ❷. To compare two variables, you use two equal sign operators (`==`), which means *is equal to*. Note that this is different from the single equal sign (`=`), which means *set to*. In this code, the first condition is `moveDirection.x == 1`, which checks the direction of the ball. If `moveDirection.x` is equal to 1, the ball is moving to the right.

The period in `moveDirection.x` is called *dot notation*; it's a way of accessing properties or functions of an object or class. In this case, the statement asks `moveDirection` to tell us the value of the `x` variable.

You can combine the conditions of `if` statements using *logical operators*. For example, you might want to compare two or more conditions, which is what we need to do in this example. To compare two statements and run

some code if both are true, you can use the AND expression, `&&`. Essentially, `&&` means *if this statement AND this statement are true, then run the code between the curly brackets.*

The logical operator `&&` at ❸ adds a second condition to the statement, `myTransform.position.x >= boundaryRight`. This part of the conditional statement asks if the x position of the ball is greater than or equal to the value in `boundaryRight`. Basically, it asks if the ball has hit the right side of the screen.

To summarize, we're asking, "Is the ball moving to the right, *and* has it hit the right edge of the screen?" If these conditions are true, then the code between the `if` statement's curly brackets runs.

Most `if` statements use curly brackets, just like class declarations, to indicate what code belongs to the `if` statement. Inside the curly brackets, the first line of code reverses the direction of the ball along the x-axis by setting `moveDirection.x` to –1.

After the direction of the ball is changed, the last line of code in the body of the `if` statement increments the `bounceCounter` variable so we can keep track of how many times the ball bounces. The entire `if` statement ends with a curly bracket to enclose those two lines of code. Now we need to do the same thing for the left, top, and bottom boundaries.

After the `if` statement we just wrote, add the following code to check if the ball bounces off the left side:

```
// left
if (moveDirection.x == -1 && myTransform.position.x <= boundaryLeft){
     moveDirection.x = 1;
     bounceCounter++;
}
```

This code checks if `moveDirection.x` is –1 this time (instead of 1), which indicates if the ball is moving to the left. Then it asks *has the ball hit the left side of the screen*? If both conditions are met, `moveDirection.x` is set to 1 to move the ball to the righ and 1 is added to the `bounceCounter` variable.

Next, add this code to make the ball bounce off the top and bottom of the screen:

```
// top
if (moveDirection.y == 1 && myTransform.position.y >= boundaryTop){
     moveDirection.y = -1;
     bounceCounter++;
}
// bottom
if (moveDirection.y == -1 && myTransform.position.y <= boundaryBottom) {
     moveDirection.y = 1;
     bounceCounter++;
}
```

The only change in this code is that it now checks the y-axis instead of the x-axis, using `moveDirection.y` and `myTransform.position.y` to track and change the ball's vertical movement.

Save the script and go back to the Unity editor. Click **Play** and the ball will bounce around the screen. You just created your first program! Take a moment to bask in the ball-bouncing glory.

Now let's make the ball-bouncing program into a game.

Controlling a Moving Bat

In this project, you'll add player control and some other feature to the ball-bouncing program. In the Project panel's *Scenes* folder, double-click the scene named *Part 2*.

The scene contains three walls, a ball, and a bat (Figure 4-1). The bat doesn't have the code to make it do anything yet, and the ball doesn't spin quite right. You'll add the code to move the bat and fix the ball spin.

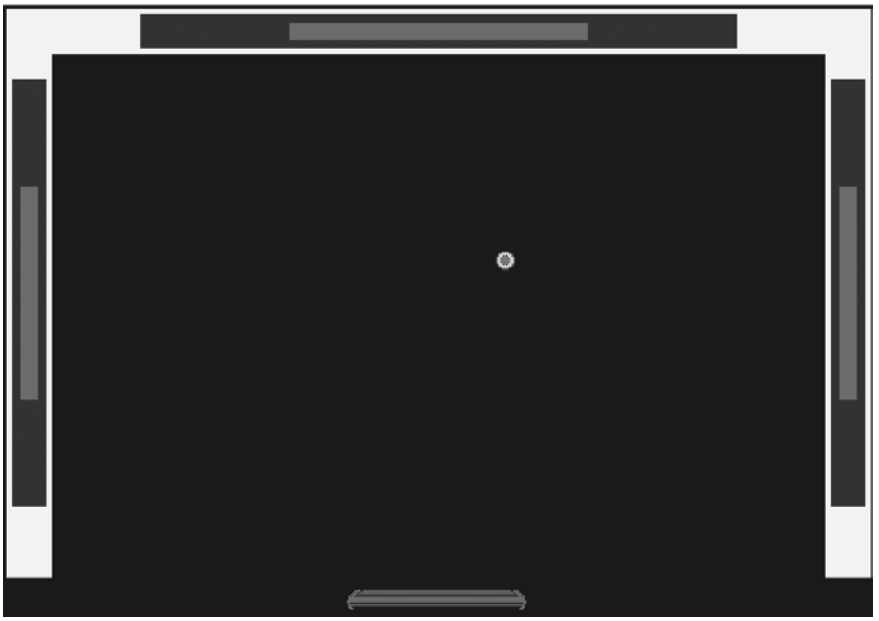


Figure 4-1: A simple bat and ball game, only the bat doesn't move yet—that's your job!

In the Project panel, click the *Scripts_2* folder and then double-click the *spinBall* script to get started.

More About Objects

As mentioned previously, C# is an object-oriented programming language, which means that chunks of code, or classes, are thought of as objects. For something to happen, these objects need to communicate with each other, especially in games, which rely on many scripts to make the game work. You'll need to write code that communicates between objects. For example, you have to write code to update a score display or make an explosive object explode.

Most elements in Unity are objects—Components, instances of classes, GameObjects, and so on. In "Modifying Number Variables" on page 56, you learned how to use variables to store data. You can also use variables to reference objects so your scripts can communicate with them and tell them what to do.

Next, we'll write some code that will change the way that the ball looks as it bounces around the screen. You'll write a short script that communicates with the ball's `Transform` Component to make the ball look like it's spinning through the air.

When you open the *spinBall* script, you'll see a skeleton structure. The script doesn't do anything yet, but you'll get to work on this. First, you need to add variables to hold references to the ball's `Transform` and `Rigidbody2D` Components. These are the two objects your script needs to communicate with to create the spinning effect.

Let's set up some variables inside our class declaration. Enter the following code at the beginning of your script:

```
using UnityEngine;
using System.Collections;

public class SpinBall : MonoBehaviour {
     private Transform❶ myTransform;
     private Rigidbody2D❷ myRB;
```

The two variables at ❶ and ❷ are used later in the code to set up references to other objects that the script needs to work.

Next, you'll add code to two different functions. The `SpinBall` class you're working on contains two functions by default, `Start` and `Update`. The game engine will call both functions automatically. But you can only have one of them in a class at any time: you can't have multiple `Start` functions for example, or the engine wouldn't know which one it was supposed to call when the game starts.

First, let's look at the `Start` function. Add this code to the `Start` function's skeleton:

```
     // Use this for initialization
     void Start () {
          myTransform = GetComponent<Transform❶> ();
          myRB = GetComponent<Rigidbody2D❷> ();
     }
```

The two variables inside the `Start` function store references to other objects so the script can communicate with them, and the built-in command `GetComponent` sets up the object references.

You tell `GetComponent` the type of Component you're looking for, and `GetComponent` returns a reference to whichever Component it finds. To use this commands, you set the Component type you're looking for between angle brackets, and then you add an empty set of parentheses at the end to

let the engine know you're calling a function. `myTransform` uses `GetComponent` to find the `Transform` Component attached to the same GameObject as this script ❶, and `myRB` looks for a `Rigidbody2D` Component ❷.

Now you have variables you can use to interact with the `Transform` or `Rigidbody2D` Components.

The Game Loop

Let's turn to our game loop. *Game loop* is a term for code that runs continuously during game play to update the game's state (player score, health, enemies, and so on) and react to player input. In Unity, that loop is handled by the `Update` and `FixedUpdate` functions.

There's one major difference between `Update` and `FixedUpdate`. `Update` is called when the screen updates, and `FixedUpdate` is called when the game's physics engine updates. The difference is in the timing: screen updates occur a certain number of times per second, and the intervals between each call can vary wildly depending on the performance of the machine the program's running on. The physics engine, on the other hand, is updated at fixed intervals, which makes `FixedUpdate` perfect for code that needs to happen at set times. Code that isn't particularly time-sensitive (like code used to update an onscreen display, for example) is better suited to `Update`, as screen update time varies depending on the frame rate of the game.

Unity calls many functions as it runs through its regular update loop. The Unity documentation has an in-depth list of the different kinds of functions and when they're called. You can find the full list in the help files under Execution Order or in the online documents at *http://docs.unity3d.com/Manual/ExecutionOrder.html*.

For this example, you'll modify the skeleton code slightly. Change the `Update` function to `FixedUpdate` by renaming it in the function declaration. Then add the following code so the function looks like this:

```
// Update is called once per frame
void FixedUpdate () {
     float rotateAmount = myRB.velocity.x❶;
   ❷myTransform.Rotate (0, 0, -rotateAmount❸);
}
```

In the `FixedUpdate` function, the ball rotates and its velocity is set. Because I wanted these updates to happen at the same time on any system, I put them in `FixedUpdate`. If you changed the function name to `Update`, the code would still work, but the ball would rotate at a different speed.

The variable `rotateAmount` stores the ball's velocity ❶. You'll use this value to decide how much to rotate the ball. The rotation will change its direction based on the ball's movement to create a simple rotation effect. Notice that `rotateAmount` doesn't have a scope declared. This is because scope is declared inside the `FixedUpdate` function. When you create a variable inside a function, it only exists when the function is called, and it stops existing when the function ends.

To get the ball's velocity, you'll communicate with its `Rigidbody2D` Component—the Component that deals with movement and speed. Using the variable `myRB`, you can use all the accessible properties and functions of the `Rigidbody2D` Component. To access the horizontal velocity from the `Rigidbody2D` Component, you'll use dot notation: `myRB.velocity.x`.

Earlier, in the `Start` function, you set up `myTransform` to reference the ball's `Transform` Component. You can now use this variable to access the publicly available properties and functions of a `Transform` Component ❷. One of the items you can access on a `Transform` is the `Rotate` function. The `Rotate` function might be a bit confusing at first, because it's set up for 3D rotation, but for now, all you need to know is that you rotate the ball along its z-axis ❸ to make it spin.

Save the script. Go back to the Unity editor and click **Play** in the scene controls. You should see the ball rotating as it moves. That's much better. A little polish makes all the difference!

Move the Bat

Now, you'll make the bat follow the mouse pointer. Open the *batControl* script from the *Scripts_2* folder. You'll get the mouse position and set the bat's position to match the mouse's position. To do that, you'll need to use vectors.

To help you understand vectors, imagine you're standing in a room in the dark and someone is explaining where the door is by telling you how much to move to the side and how much to move forward or backward. Let's say you're told to move three steps to the right and two steps forward. You could represent these instructions as a vector: (3, 2). The amount to move horizontally is a positive number, which means you should go right; a negative number would mean go left. The next number is also a positive number, which means you should move forward; a negative number would mean step backward.

Vectors can be used to hold information about positions in your game world. The format of a vector is (***x***, ***y***), which just happens to be how we describe two-dimensional space to the game engine. Vectors are part of a larger family of variables known as *data structures*. Structures store groups of data (such as x- and y-coordinates) in a way that uses only a single variable rather than having to store each piece of data separately. If you think of variables as a box, this type of box has several compartments in it.

Near the top of the *batControl* script, inside the class declaration, are three variables you need to declare. Two of these are `Vector2` variables you'll use to set up the bat position, one to find the position of the mouse and another to set the position of the bat. The variable type you need is called `Vector2`.

```
private Transform myTransform;
private Vector2 myPosition;
private Vector2 mousePosition;
```

`myPosition` holds the position of the bat in a `Vector2` variable.

Now, jump down to the `Update` function. Here you'll add code to make the bat interactive:

```
    void Update () {
        // get the position of the bat from the transform
❶       myPosition = myTransform.position;
        // get the mouse position (converted from pixels to world units)
❷       mousePosition = Camera.main.ScreenToWorldPoint (Input.mousePosition);
        // set myPosition to have the same x position of the mouse
❸       myPosition.x = mousePosition.x;
        // update the transform's position to move it where it should be
        myTransform.position = myPosition;
    }
```

When you get the position of the mouse pointer, the game engine returns a value in pixel/screen space. The position of the bat needs to be in game world units, so we have to make a little conversion using the built-in function `ScreenToWorldPoint`. `ScreenToWorldPoint` can be called on a camera and will return converted coordinates based on the camera's position and rotation, combined with the pixel coordinates that you have to pass in as a parameter. I used a shortcut to get the main camera in the scene—instead of providing an exact reference I just used `Camera.main` to let Unity figure it out for me. The returned converted mouse position from `ScreenToWorldPoint` is stored in the variable `mousePosition`. You only need the x position of the mouse pointer, which you'll grab using dot notation: `mousePosition.x`.

Click Save and go back to Unity. Click **Play** in the scene controls. The bat should now move with the mouse so you can stop the ball from going out of the play area. Welcome to your own version of a classic arcade game!

NOTE *If you've been snooping around the project, you'll find that this scene has a different ball bouncing script that uses collisions rather than ball positions to decide when to change direction. Because you needed the ball to detect collisions against the bat this time around, I snuck in a more complex version. I won't explain the script here, but the collision system it uses will appear in Chapter 8 when we build the player script for the platforming game.*

Breaking Bricks!

In Unity, find the Project panel's *Scenes* folder and double-click the *Part 3* scene. This is the third and final mini-project for this chapter. In this project, you'll add some bricks to create a brick-breaking game. This scene also adds a score display. Figure 4-2 shows the final game.

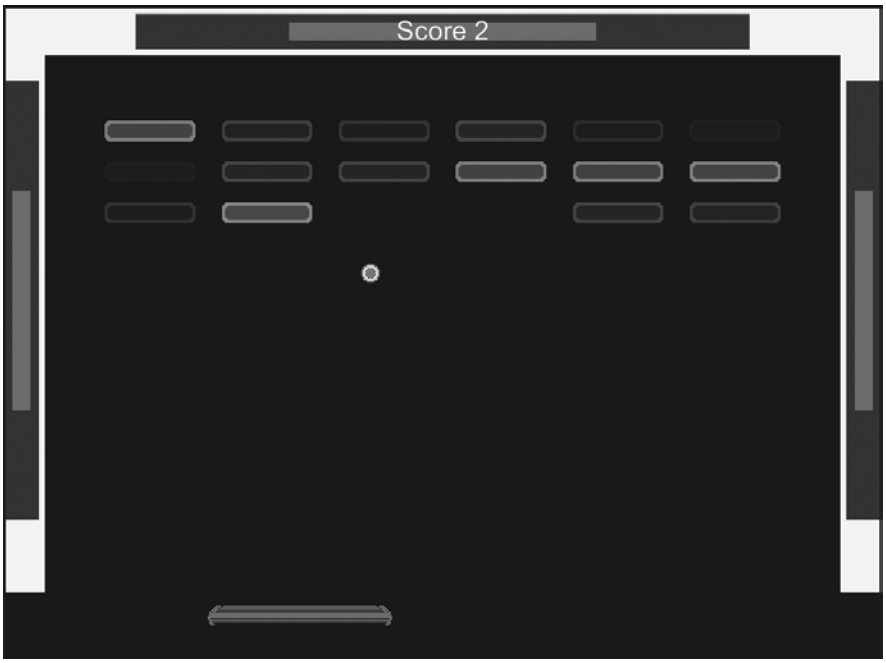


Figure 4-2: The brick-breaking game needs scripts to add these bricks to it!

Use a Loop to Make Bricks

In the *Part 3* scene, there's not much of a game yet. The score display never increases, and the ball just bounces off the walls like it did in *Part 2*. To make the game more interesting, you'll use a loop (specifically, a `for-next` loop) to add bricks for the ball to break. In the Project panel's *Scripts_3* folder, double-click the *GameController* script.

The function you'll build here will be called by the GameController class's `Start` function. Let's talk about that `Start` function for a moment.

The `Start` function is called automatically before anything is updated and before graphics are rendered to the screen. This makes it the perfect place for setup code, such as code that creates dynamic-level objects (like breakable bricks). Because `Start` is called before graphics are drawn to the screen, the level will be built before the player sees it happen. When the game starts, the player will already see bricks on the screen.

Let's move on to loops. You'll find lots of loops in programming. Loops give you the ability to repeat a piece of code without having to enter it over and over. In this script, you'll split the code into two loops. The first keeps track of rows and the second keeps track of columns, because that's how the bricks will be presented (Figure 4-3).

Figure 4-3: Bricks appear as three rows and six columns.

A C# loop contains three parts separated by semicolons:

- The initialization
- The condition
- The afterthought

A loop will look something like this:

```
for (initialization; condition; afterthought) {
    // Here, between the curly brackets, is the code that gets repeated
}
```

The *initialization* sets up a variable to keep track of how many times the code is looped until the *condition* is met. The condition states what you're testing. After each loop through the code, the *afterthought* runs. The afterthought usually increases or decreases the initialized variable that's counting the number of completed loops. The `BuildLevel` function will use two loops (called *nested loops*, because one is inside the other) to create the bricks and lay them out on the level.

Just after the curly bracket that closes the `Update` function, add the following code:

```
void BuildLevel() {❶
   ❷for (int by = 1; by<totalRows; by++) {
        ❸for (int bx = 1; bx<totalColumns; bx++) {
             ❹MakeBrick (bx, by);
             ❺totalBricks++;
          }
     }
}
```

The `BuildLevel` function doesn't need to return anything, so its type is `void`. Because we don't need to pass anything into it, the `BuildLevel` function has just an empty set of parentheses to tell the engine that it's a function ❶. Then a curly bracket signifies the start of the code that will run whenever the function is called.

Following the curly bracket are two nested loops. In the first loop, the variable `by` decides which row to draw ❷, and in the second, the variable `bx` counts the columns ❸. (`b` stands for brick, `x` for x-axis, and `y` for y-axis.)

Now you can use the `bx` and `by` variables to create and position the bricks with the `MakeBrick` function at ❹. Don't worry too much about how the `MakeBrick` function works just yet; I'll come back to this later in the section.

The code at ❺ increments the variable `totalBricks`. `totalBricks` counts how many bricks have been made. The *GameController* script needs to know how many bricks are in the scene so it can add more when the ball has broken all of the bricks. Because the bricks are added to the scene in the new function, it's the best place to count how many are made. Finally, two curly brackets close the loops.

Although the `BuildLevel` code is ready to go, it won't do anything until the `BuildLevel` function is actually called from another function. To call `BuildLevel`, add the following line to the `Start` function, after the line `bricksDestroyed = 0;`:

```
        BuildLevel ();
```

You need to call `BuildLevel` in one more place—when the number of bricks that have been destroyed is greater than or equal to the total number of bricks in the level. Destroying all of the bricks in the level can happen at any time, so you'll need a condition in the `Update` function that will check the number of destroyed bricks regularly and react accordingly. In this case, when the bricks are all destroyed, the `Update` function needs to call `BuildLevel` to make a new set.

Add the following code inside the `Update` function:

```
    if (bricksDestroyed >= totalBricks) {
        BuildLevel ();
    }
```

When all of the bricks have been broken, `BuildLevel` will be called to make more bricks for the player to destroy.

Save the script and then return to Unity. Every time you save a script, the engine will automatically recompile it, so there might be a slight delay. Games containing large numbers of scripts can take several seconds to recompile. However, because this script features just a few small scripts, the compile delay should be minimal.

Click the **Play** button in the scene controls and do some brick breaking. Enjoy!

Next, I'll introduce you to another key concept in programming, the *array*. Data storage is an important part of game development, and for small-scale data storage, game developers usually opt to use arrays. You'll use an array to add a little extra shine to the brick-breaking game.

Color Your Bricks with Arrays

When you need to store more than just a single number or string, you might need to use an array. An array is a type of variable used for storing multiple objects in an easily accessible way. Those objects might be numbers, strings, instances of classes—any type of object the Unity engine allows you to access.

For example, let's say I want to store the layout for a level made of tiled blocks. Using a grid of 10×10 tiles (100 tiles in total), I also want to use a

number to indicate each tile type. Rather than making 100 variables to store these numbers in, I can make a single variable to hold an array of 100 numbers. Because numbers, strings, objects, and structures can be stored in a variable, they can also be stored in an array.

Many kinds of arrays exist, but for the purposes of this book I'll use two array types, the built-in array and the `Generic List`. In a nutshell, built-in arrays are ideal for storing items that you want to access via the Inspector in Unity. `Generic Lists` are useful for storing any kind of data that only needs to be accessible via code (not the Inspector). A `Generic List` has the smallest impact on performance and memory use.

Although the brick-breaking game is fun, the bricks look a bit dull. Let's do something about that by building a script that uses random colors to add some pizzazz. We'll store the colors in a `Generic List` array and have the script pick a color at random to pass to the brick's `Sprite Renderer` Component. Then the `Sprite Renderer` will color the brick.

Find the *Scripts_3* folder in the Project panel and double-click the *ColorBrick* script file to open it. At the top of the script, under the line that reads `using System.Collections;`, add this line to access the `System.Collections.Generic` library:

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

Right now, the class `ColorBrick` is empty. Enter the following code to add the variable declarations for the array and one for the reference to the brick's `SpriteRenderer` Component inside the class:

```
    private SpriteRenderer myRenderer;
    public List<Color> colorList;
```

Every sprite in Unity uses the `SpriteRenderer` Component to draw its image. This Component has a public color property you can access to tint sprites, which is perfect for what you want to do here. To access the `SpriteRenderer`, you need a variable containing a reference to it, which in this code is called `myRenderer`.

Next, you declare the `Generic List`. A `Generic List` takes a scope (here, it's `public` so you can easily add color either inside the Inspector panel or in the code), followed by the word `List`. The type of object you'll be storing in the list goes inside angle brackets. Then you need to name the `Generic List`—in this case, it's `colorList`—and add a semicolon to end the line.

Now, move down to the `Start` function and add the following code:

```
        colorList.Add(new Color(255,0,0,255));
```

To add items to a `Generic List`, you write the list's name followed by a period and the `Add` command. After `Add`, use parentheses around the item you want to add to the list. In this case, it's a color. `Color` is a built-in structure. To create a color, you must use the `new` keyword to tell Unity you're

creating a new structure. Then, enter `Color` followed by four numbers wrapped in parentheses. These numbers are the parameters that the `Color` function uses to describe a color. The first three numbers represent red, green, and blue values, which is a standard way of representing colors on computers. The fourth number is an *alpha value*. The alpha value determines how transparent the sprite should be. The `Color` structure takes values between 0 and 255 for each color you want to add.

In the preceding code line, I set the red value to 255 to make this color pure red. Because you don't want any of the bricks to be transparent, the alpha value is also set to 255.

Next, enter the following lines of code to add blue, green, and yellow to the `colorList`:

```
colorList.Add(new Color(0,0,255,255));
colorList.Add(new Color(0,255,0,255));
colorList.Add(new Color(255,255,0,255));
```

Now, add a line of code to grab a reference to the `SpriteRenderer` and store it in the `myRenderer` variable declared earlier:

```
myRenderer = GetComponent<SpriteRenderer> ();
```

Everything is ready to use: you've set up the variables, added some colors to the array, and grabbed a reference to the `SpriteRenderer`.

The next step is to make the script pick a random color from the list:

```
int colorIndex = Random.Range(0, colorList.Count);
```

`Random.Range` is a built-in function that takes minimum and maximum number values, and then picks a random number between the two. Here, the minimum number is zero because the first item in the array starts at 0 and the maximum number is provided by `colorList.Count`. A list's `.Count` property returns the number of items that the list contains, meaning that the `Random.Range` function should return a number between zero and the number of colors in `colorList`. That return value then gets stored as an integer in `colorIndex`.

To get an item from a list, you need to put it into a variable of the same type. Add this line:

```
Color theBrickColor = (Color) colorList [colorIndex];
```

`theBrickColor` is a Color type variable, which is used to hold the color the code gets using the `colorIndex` number. After the equal sign is a type in parentheses, telling the engine the next object will be this type. Because the engine doesn't always know what kind of object is coming next, the type makes sure that `theBrickColor` variable gets the right value. To access a color in `colorList`, write `colorIndex` (which is an integer) inside square brackets.

`theBrickColor` should now hold a randomly chosen color. The final step is to use the `SpriteRenderer` to color the brick. Add this line:

```
myRenderer.color = theBrickColor;
```

The `.color` property of the `Sprite Renderer` Component referenced by the variable `myRenderer` is set to the `Color` object in `theBrickColor`.

And that's how you use an array to set a sprite to be tinted to a random color. Click **Play** in the scene controls to see all the colorful bricks!

The ability to store different types of objects or values in `Generic Lists` makes them perfect for all kinds of uses. Arrays will come in handy for your future game development.

Closing Words

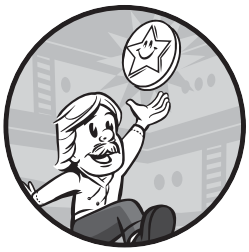
Believe it or not, that's really about all the programming knowledge you'll need to make games. The reality is that code is just a tool we use to tell computers how we want to solve problems. Anyone can learn to program, and with practice, solving game-related problem gets easier and easier.

Like most arts, programming requires knowledge of basic principles to help your imagination flourish. In this chapter, you made some fun things happen in the brick-breaking game, and you explored principles of programming such as variables, functions, and classes.

You've learned a lot and you're doing great, so don't be afraid to do battle with programming! In the next chapter, you'll set up some game play using your new programming skills. You'll continue to build those skills, and you'll be coding your own games in no time.

5

PROGRAMMING PLAYER CONTROLS AND GAME PHYSICS



So far, you've created an animated player sprite and learned some programming basics. In this chapter, I'll show you how to program Max so you can control him using the keyboard. You'll add some gameplay by creating simple objects and learn about game physics and *collision detection*, an important concept for developing any kind of game. Collision detection is when you check whether two objects in your game world are touching each other; it's the bread and butter of game development. You'll learn how Unity handles collisions so you can apply collision code and components to your own games. This is an exciting chapter!

You'll make a simple game where the player dodges bricks as they fall from the sky (see Figure 5-1). The goal is to avoid the falling bricks for as long as possible. The catch? The bricks fall faster and faster as the game goes on.

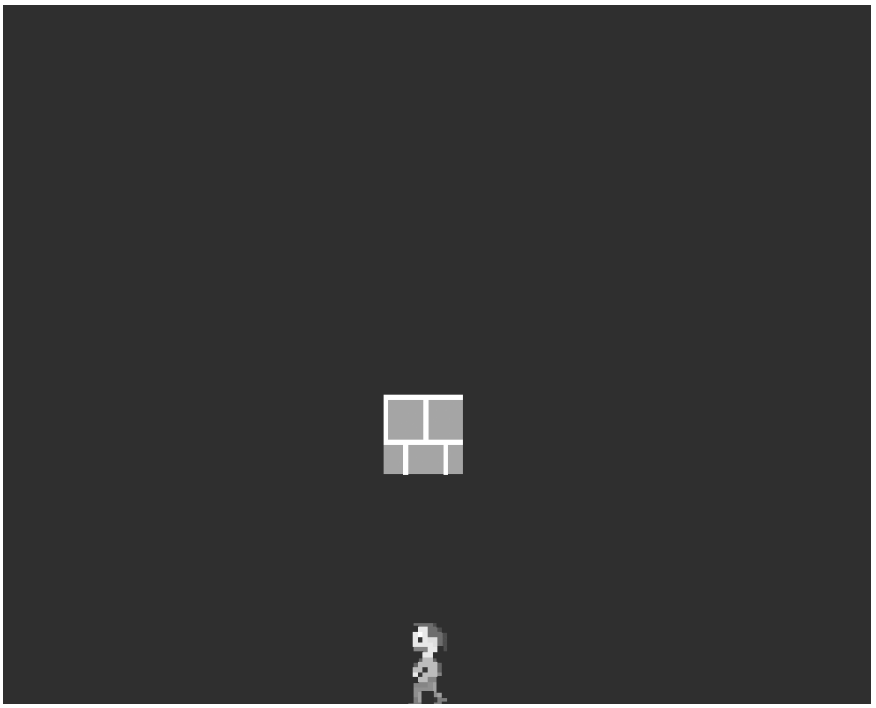


Figure 5-1: The simple brick-dodging game we'll build in this chapter

All of the action will take place in a single scene. The player needs to move left or right so poor Max doesn't get bonked on the head. We won't create any fancy scoring, levels, or user interfaces just yet. This game is purely for the fun of it and will get you started with the basics of creating a playable demo.

Dodging Falling Bricks

Open Unity and use Open Project to find the example from the source files provided in *Examples/Chapter 5/*. Recall that organization is key to your projects, so let's start with a quick look at the project structure. Figure 5-2 shows the Project panel with four appropriately named folders for *Graphics*, *Prefabs*, *Scenes*, and *Scripts*.

As you move through the project, try to keep track of where Unity saves your files. Unity tends to save new files you create in the *Assets* folder, so be sure to move them into the proper folders as you make them. In the example project files, the graphics you'll need are in the *Graphics* folder already. You'll need to be aware of save locations for all other files you create.

Let's create a new scene for all the action to happen in. To do this, go to **File ▸ New Scene**. Unity prompts you to save the changes on the current scene, but because we haven't made any changes, just click the **Don't Save** button to continue.

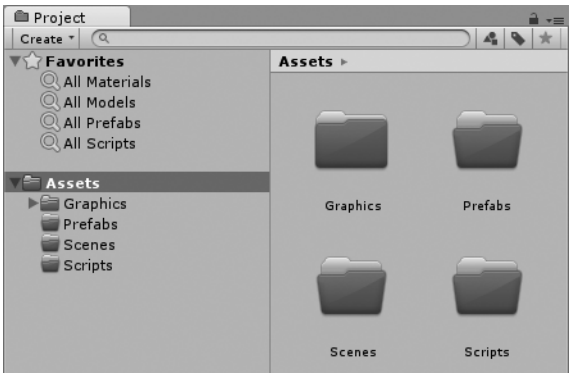


Figure 5-2: An orderly project structure

Unity makes a new, empty scene, but it isn't saved to a file until you tell it to. To save the file, go to **File ▸ Save Scene**. Click the *Scenes* folder and then enter *myGame* into the File name box. Click the **Save** button (Figure 5-3) to save the scene.

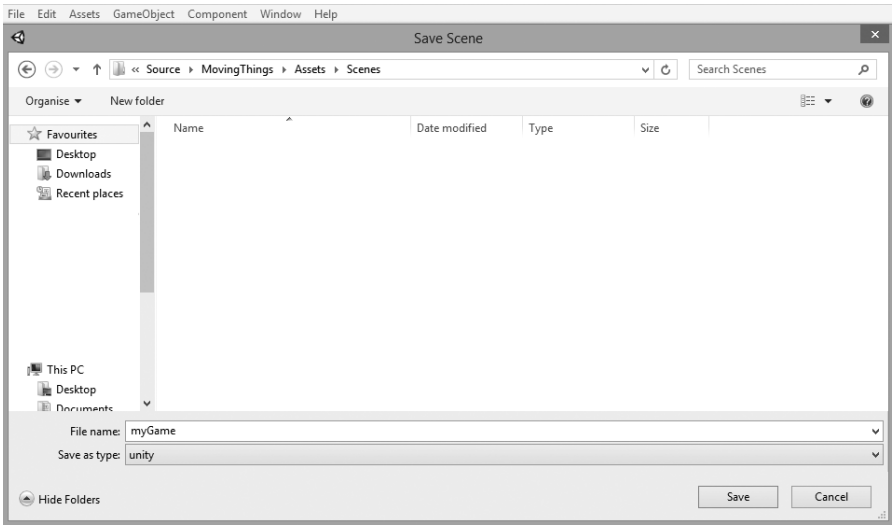


Figure 5-3: Saving a new scene into the Scenes *folder*

The project is ready to go. Let's turn those graphics into objects, sprites, and animations.

Add the Player Sprite to the Scene

The first step is to add the player sprite. In the *Graphics* folder, click and drag the `player_spritesheet` object and drop it into the Scene (*not* the Hierarchy; it has to be in the Scene panel for the animation to work right). A Create New Animation window should appear. Name it *playerwalk.anim* and click **Save**.

The Max character sprite should now be in the Scene. Click **Play** to watch his walk animation. Unity automatically named your sprite `1`, which is not the most useful description, so you need to rename it in the Inspector. Select Max and then rename the sprite `Player` using the text field at the top of the Inspector.

Next, let's create a script to make him move around.

Programming Player Controls

Make sure the player sprite is selected in the Hierarchy. Then, in the Inspector, click the **Add Component** button (Figure 5-4) and select **New Script**.

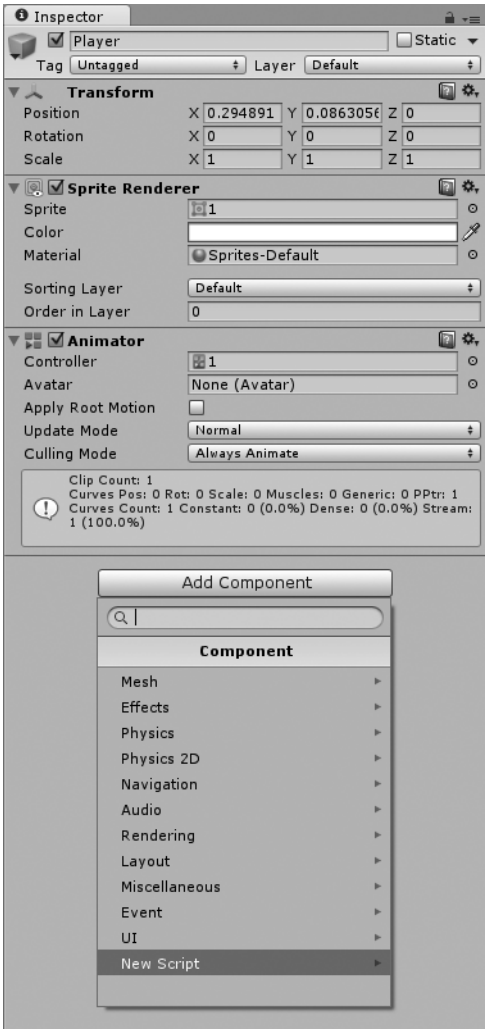


Figure 5-4: The Add Component pop-up menu

Name the script *PlayerControl*. The default script type should be C Sharp (C#), but double-check that C Sharp is selected in the Language drop-down menu. Click the **Create and Add** button. The Inspector should now show a new Component—your script!

NOTE *Unity editor calls the language C Sharp, which is the way C# is pronounced. Both C Sharp and C# refer to the same language.*

Double-click the script name in the Inspector. Your script editor program (MonoDevelop or Visual Studio) opens to a skeleton script (Figure 5-5), which is the same as the skeleton C# script you saw in Chapter 4.

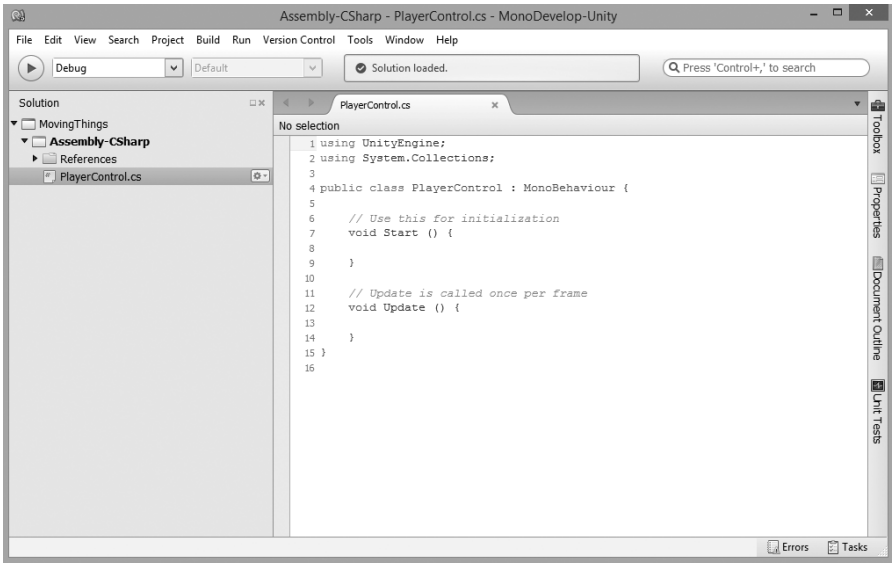


Figure 5-5: MonoDevelop and the PlayerControl.cs *script*

As mentioned in earlier chapters, Components are attached to GameObjects to make them do things. We'll make a script that will become a Component on the `player` GameObject (your sprite) so you can control it using keyboard input. To do that, we need to access the player's Transform Component.

The Transform Component holds position, rotation, and scale information about your GameObject. Transforms are attached to every GameObject automatically, and to access them, you'll need to store a reference to the Transform in a variable. At the top of the script, just after the class declaration, enter the following code to declare a variable that holds the Transform reference:

```
public class PlayerControl : MonoBehaviour {
private Transform myTransform;
private float gameWidth = 6;
```

The line `private Transform myTransform;` sets up a variable that holds the Transform reference. Because we don't need to access the variable anywhere else, you can declare it as a `private` variable.

The `gameWidth` variable is a float that is used to set how far left or right the player can move. The middle of the game play area is at zero, so `-gameWidth` indicates where the left boundary of the play area is, and `gameWidth` indicates where the right boundary is. For this game, the play area is 6 units wide. Later in this chapter, you'll write code to check the player position.

The size of the graphics in the game world impacts how the play area is used. In this case, all the graphics' Pixels To Units ratios are set to 16 (as in previous chapters). That means you can convert the play area of 6 units wide into its pixel equivalent by calculating 6 units × 16 pixels wide, resulting in a play area that is 96 pixels wide. How those 96 pixels are displayed onscreen depends on the camera settings. But this calculation of actual pixels in the game world can be useful, for example, for creating correctly sized background graphics in a graphics program like GrafX2.

The next step is to grab a reference to the `Player` Transform Component and store it in the variable `myTransform`. To make sure that the `myTransform` variable is set up before you try to access it, grab the reference in the `Start` function. Add the following line of code on the line just after `void Start ()`:

```
myTransform = GetComponent<Transform> ();
```

`myTransform` will now hold whatever the `GetComponent` command returns. The `GetComponent` command will search for the Component you specify in angle brackets <>, and it will only look for Components attached to the GameObject that this script is attached to, which in this case is `Player`. Remember that `GetComponent` is a function, so you need to include the parentheses at the end of the command to call it. Now when the scene starts and the GameObject is initialized, `myTransform` should be ready to access the Transform Component.

Next, you'll add some code to the `Update` function. Recall from Chapter 4 that the `Update` function is where you can put code that needs to be called constantly throughout the game. Code in `Update` is called automatically by the game engine every time it goes through its normal update loop.

On the next line, inside the `Update` function's curly brackets, you'll check for player input. Enter these lines into the `Update` function:

```
❶float moveInputAmount = Input.GetAxis ("Horizontal");
❷if (moveInputAmount>0 && myTransform.position.x < gameWidth)
 {
    ❸myTransform.Translate (new Vector2(0.1f, 0));
 }
```

`Input.GetAxis` checks what Unity calls a *virtual axis*, which you can use for checking keyboard input (or joystick or controller input). The default input settings provide two axes named `Horizontal` and `Vertical`. You can access

axes values using the `GetAxis` function, as shown at ❶. `GetAxis` returns a value between –1 and 1. If the value is 1, the right key is pressed. If the value is –1, the left key is pressed. If no key is pressed, the value is 0.

Figure 5-6 shows a visualization of the `Horizontal` axis. In plain English, the code at ❶ asks Unity for the horizontal input amount and stores it in a variable named `moveInputAmount`.

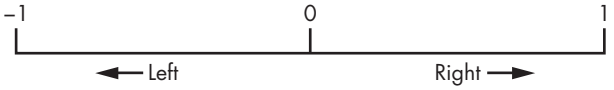


Figure 5-6: The default `Horizontal` axis setup has left input as a negative number and right input as positive. This same logic can be applied to the `Vertical` axis by replacing left and right with down and up, respectively.

The first condition at ❷ asks what the state of the `Horizontal` axis is, as stored in `moveInputAmount`. It checks whether the value returned by `GetAxis` is greater than zero. If that condition is true, the user has pressed the right key and the player should move right.

The second part of the condition (everything after `&&`) checks the player's `x` position against the variable `gameWidth` to see whether the player is in the game play area, because we want to keep the player within those boundaries. If both conditions are met, the code between the curly brackets runs, and the player moves to the right. Note that if the second condition is false, the player is up against the edge of the screen, and the code between the brackets is skipped.

Now that we've detected user input and made sure that it's safe to move Max, let's investigate the actual movement code. In the body of the condition ❸, I use the Transform Component, whose reference is held in the variable `myTransform`, to move the player. To do so, you can call a function called `Translate`. In vector math, the term *translate* just means to change or move a vector. The `Translate` function takes three parameters: `x`, `y`, and `z`. You don't need to worry about the z-axis, which is for 3D movement. You only need to move along the x- or y-axis to go left or right.

When you use `Translate`, the x, y, and z values set up a vector that tells Unity how much the player should move from its current position. To go right, I give `Translate` an x value of 0.1. The reason the value is so small is that I don't want the player to move too quickly. Passing higher values to the `Translate` function will make the player zoom off the screen.

Let's look at the code for the player to move left. Enter the following code right after the code you just added to the `Update` function:

```
        if (moveInputAmount<0 && myTransform.position.x > -gameWidth)
        {
            myTransform.Translate (new Vector2(-0.1f, 0));
        }
```

In the first condition, we check the value of `moveInputAmount` for a negative value instead of a positive one. In the second part of the condition, the transform's `x` position is compared to `-gameWidth`. If the result from `moveInputAmount` is less than zero and the player has enough room to move left, the code in the curly brackets runs, and `Translate` moves `Player` to the left because we've passed –0.1 to it for its `x` value.

That's all the code it takes to move the player left or right, though it won't work in the game until the GameObject has Components attached to it that allow the `Player` object to work as part of the physics engine.

Game Physics

For an object to behave in a realistic way, a whole lot of math needs to be done. The *physics engine* handles the math to simulate your game world, and the game engine uses that information to make objects move. The physics engine also handles events, like collisions.

In this book, you'll be using one of Unity's two physics engine libraries, Box2D. Because Unity has two physics engines, there are two sets of physics and collision commands, which can be confusing at first. To avoid any confusion, just keep in mind that all 2D physics collisions Components, or functions, have 2D in their name, such as `RigidBody2D`, `BoxCollider2D`, and so on.

Unity's implementation of the Box2D engine provides a number of Components. Here are the main ones you'll use in this book:

`RigidBody2D` Enables GameObjects to act under the control of the physics engine. For any objects to react to each other, they must have a `RigidBody2D` Component.

`SphereCollider2D` Provides the GameObject with a simple sphere-based collision shape.

`BoxCollider2D` Provides a simple box shape as the collision shape.

`EdgeCollider2D` Creates platforms in a 2D platform game. It provides a collision shape based on a collection of points along a line. For example, imagine a line along the top of a platform that a player walks on.

Unity's 2D physics system has quite a bit more functionality, but I won't cover it here. As you become more advanced, the Unity documentation is a great place to learn about new and wonderful features. If the basic collision or physics systems discussed here aren't working well for your game, check out other options under the `RigidBody2D` or `Collider2D` classes in the game engine documentation.

In our falling bricks game, the player and brick GameObjects need a `RigidBody2D` Component attached so they can react to collisions. In addition, the player and brick GameObjects will have a collider Component attached.

Setting Up Physics and Collisions

We'll add gravity to the game so bricks fall from the sky, and to add more excitement, we'll make the bricks explode when they hit the ground. But first, let's make sure the player is set up with the right physics and colliders.

Add Physics to the Player

Switch back to Unity if you're still in your script editor. Select Max in the Scene. In the Inspector, click **Add Component** and select **Physics 2D ▸ Rigidbody 2D**. This tells the physics engine that the player is a physics object that needs to be controlled by the physics system, but it still needs a collision component. Without collision, the physics engine will apply gravity to the player, and the player object will drop off the screen. Let's prevent that from happening.

Click **Add Component** again and select **Physics 2D ▸ Box Collider 2D**. The `BoxCollider2D` Component automatically matches the size of the sprite. Now Max should act like a solid object in the game.

Add the Ground

With the `myGame` scene open, right-click in the Hierarchy and select **2D Object ▸ Sprite**. Name this new sprite `GroundSprite`. With `GroundSprite` still selected in the Hierarchy, look in the *Graphics* folder in the Project panel and find the `Sandy` sprite. Drag `Sandy` into the Inspector's Sprite field to set the `GroundSprite`'s sprite to the sand tile.

Before adding a collider, we need to move `GroundSprite` down to the bottom of the screen so it looks like the ground that the player is standing on. In the Inspector, enter the following numbers in the Position fields:

X: 0

Y: –5.1

Z: 0

The bottom of the play area should now be nice sandy ground. The only problem is that the bricks will just go right through it, so let's prevent that by adding a 2D collider. With `GroundSprite` still selected, click the **Add Component** button in the Inspector. Select **Physics2D ▸ Box Collider 2D** to add the `BoxCollider2D` to the GameObject.

With the ground set up, click **Play** to see the player move around a bit and click **Stop** when you're ready to continue with the next step, in which you'll set up the bricks.

Create the Brick Object Prefab

Creating a prefab is the easiest way to reuse GameObjects in different scenes. *Prefabs* are like templates of GameObjects that you can drag into a scene; they retain the same properties as the original GameObject. For

example, you might build a character with a head, a body, and limbs. Instead of rebuilding the same character in every game scene, you can put the head, body, and limbs into a prefab. Whenever you load a new level, you can add the prefab to the scene (called *instantiation*), and it should appear as you originally made it.

Instead of creating lots of bricks, we'll create a prefab brick object and tell a script to create instances of the prefab.

First, you'll add the brick object to the scene. From your *Assets/Graphics* folder, drag the `brick_tile` object into the Hierarchy. The brick sprite will appear in the center of the Scene preview, and `brick_tile` should be listed in the Hierarchy. Next, you'll add the physics and collision Components you need.

Make sure the `brick_tile` is selected in the Hierarchy panel so you can see it in the Inspector. Click the **Add Component** button and select **Physics2D ▸ RigidBody2D**. Next, add a `BoxCollider2D` Component. Click the **Add Component** button again and select **Physics2D ▸ BoxCollider2D**.

The brick should now have a collision Component and physics. When you click **Play**, the brick will fall and hit the ground. Click the **Stop** button before continuing.

When the brick hits the ground, it just sits there. Let's write a script to make it disappear when it hits the ground. This will make the game playable. (We'll revisit the brick collision later to add a nice explosion particle effect.)

When the physics engine detects a collision, it automatically makes a call to all Components attached to the affected GameObjects. By adding a function to a GameObject's script to "catch" the call, you can create collision-based events. In this case, you'll destroy the brick GameObject when it hits the ground.

Click **Add Component**. At the bottom of the menu, click **New Script**. Make sure you select **C Sharp** as the language and name the script *Hazard*. Create the script by clicking the **Create and Add** button.

A script named *Hazard* should appear in the Inspector. Click the script to open your script editor. You'll see the familiar template script to start. To make bricks disappear when they hit the ground, simply add the following code after the class declaration (you don't need to change the `Start` or `Update` functions):

```
void OnCollisionEnter2D() {
    Destroy (gameObject);
}
```

The function `OnCollisionEnter2D` is one of Unity's built-in functions. Unity calls `OnCollisionEnter2D` whenever a collision involving the associated GameObject occurs.

To remove the sprite from the scene, destroy the GameObject using the `Destroy` function. Notice that this is done by passing the `gameObject` to the `Destroy` function. This tells the `Destroy` function to destroy whatever GameObject it's attached to. That's it for the brick script!

Save the script and then close or minimize your script editor to return to Unity. Create a prefab by dragging the `brick_tile` GameObject out of the Hierarchy and dropping it into the *Prefabs* folder in the Project panel. This should add a new file to the project in the *Prefabs* folder. As long as the `brick_tile` prefab exists in that folder, you can delete the original one in the Hierarchy. Right-click the `brick_tile` GameObject in the Hierarchy and choose **Delete** from the drop-down menu.

Creating a Game Controller Script

Because you just deleted the `brick_tile` GameObject, when you click the Play button, nothing happens. Let's build a game controller script to create falling bricks. In game development, a *game controller* script refers to code that keeps track of the game's core logic, and it's best to keep all of the game's logic in this script. For example, the game controller script could be code that keeps track of the score or code that keeps track of the game's state, like whether the game is active or paused. Figure 5-7 shows a regular structure I use in its simplest form: separate scripts are attached to players, enemies, and other scene objects. The game controller script is central to this structure.

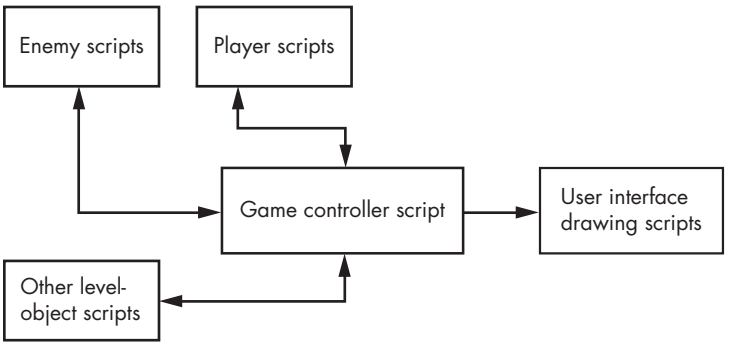


Figure 5-7: A typical componentized scene structure based on a game controller script

Always attach the game controller script to its own GameObject called `GameController` so it's easy to find in the Hierarchy. First, create the GameObject by right-clicking in the Hierarchy. From the pop-up menu, click **Create Empty**.

Why choose an empty GameObject? Empty GameObjects are a great place to put scripts that have no associated sprites in a scene. For example, this *GameController* script will only spawn new brick objects. The script isn't a physical object in the game. To keep the project organized, it gets its own GameObject so it won't get mixed in with a bunch of other Components.

The other important reason to attach your game controller script to an empty GameObject is that scripts that derive from the `Monobehaviour` class—like the game controller—need to be attached to a GameObject in the Scene to access `Monobehaviour` functionality, such as the `Start`, `Update`, or `FixedUpdate` functions.

Rename the new GameObject to `GameController`. Its default name is `GameObject`, so find it in the Hierarchy and either left-click its name to rename it or highlight `GameObject` and change its name in the Inspector.

Now let's get back to more programming. Make sure the `GameController` GameObject is selected, and then click the **Add Component** button in the Inspector panel. Choose **New Script** from the menu. In the Name field, type **`GameController`**. Make sure that **C Sharp** is selected in the Language drop-down menu and click the **Create and Add** button.

Double-click the script to open the script editor and edit the code. The *GameController* script starts out as the regular template you saw earlier in this book. It will need several variables to work. Inside the public class declarations, declare all those variables like this:

```
public class GameController : MonoBehaviour {

     public GameObject hazardPrefab;
     public float gameAreaWidth = 5;
     public float startingHeight = 6;
     public float timeBetweenDrops = 1f;
     public float dropGravity = 1f;
     private Vector2 theNewObstaclePosition;
```

I'll discuss each variable's purpose as it comes up in the script. For now, let's just zip ahead to the `Start` function. `Start` gets called only once when the GameObject is initialized, so it's a good place to take care of anything that needs to be set up before the main update functions are called.

In the `Start` function, add an `Invoke` statement to make the first brick drop after two seconds:

```
     void Start () {
          Invoke ("DropBrick", 2);
     }
```

`Invoke` takes a function name in quotation marks as one of its arguments, followed by the number of seconds before the function is called. It's like having a timer built into your code. In this case, a function called `DropBrick` will be called two seconds after the `Start` function is called.

You don't need to add anything to the `Update` function, so you can ignore it. On the line after the `Update` function, add the following code to create the `DropBrick` function:

```
void DropBrick() {
     // set up a Vector2 to use for positioning the new brick
     theNewObstaclePosition.x = ❶Random.Range (-gameAreaWidth, gameAreaWidth);
     theNewObstaclePosition.y = ❷startingHeight;
```

First, we declare a variable named `theNewObstaclePosition.x`, which we'll use to make bricks randomly appear along the top of the screen. It's a `Vector2` type variable, which means we can use it to store position data (like x- and y-coordinates) to use with the `Transform` Component. In this case, we'll

use it to build a position vector with a random x value using the `Random.Range` function ❶. `Random.Range` returns a randomly generated value in the range that you pass to it. Here, we pass in the width of the game (held in the variable `gameAreaWidth`) in negative and positive form (Figure 5-8).

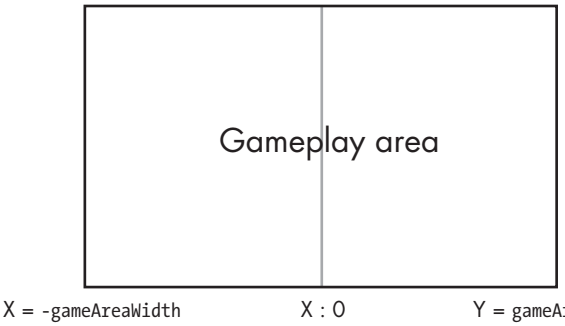


Figure 5-8: The gameplay area and how `gameAreaWidth` is used to find the left and right sides of it

To set the height of this new brick, we use the variable `startingHeight` ❷, which holds a value to set the height bricks should fall from. I set this value to 6 in the variable declarations, but I didn't use any kind of scientific process to get that number. I tested a few different values until it looked right.

The next part of the `DropBrick` function will actually create a brick in the game. Add this code to the `DropBrick` function:

```
    theNewGameObject = (GameObject) Instantiate (hazardPrefab,
theNewObstaclePosition, Quaternion.identity);
```

To create a brick, this code uses the `Instantiate` function. When `Instantiate` is used to add a new GameObject to the scene, it returns a reference to the new object. The variable named `theNewGameObject` will hold that returned reference.

NOTE *The `Instantiate` function makes a copy of the prefab you pass into it, adding the copy to the scene for you to use in your game. Game developers often refer to this process as* spawning.

Notice that immediately before the `Instantiate` call is a `(GameObject)`. The game engine doesn't know what type of object `Instantiate` will return, so you have to tell it. In this case, it's a GameObject, but it's possible for `Instantiate` to return a `Transform` if that's what your code calls for. Make sure that the variable you're putting the reference into is of the same type. In this case, we want to use the new object's GameObject Component in the next line of the code, so we enter it as `(GameObject)`.

In the next line, we call `GetComponent` on the new GameObject to access its `RigidBody2D` Component:

```
theNewGameObject.GetComponent<Rigidbody2D> ().gravityScale = dropGravity;
```

We need to access the `Rigidbody2D` Component so we can scale the way the brick reacts to gravity. As the game progresses, we'll want the bricks to fall faster. To achieve this, we'll increase this `gravityScale` value so gravity has a greater effect on the falling bricks.

Let's add more to the `DropBrick` function. Enter these lines to set up future brick drops:

```
    // make sure any previous calls to drop a brick are cancelled out
❶CancelInvoke ("DropBrick");

    // schedule a new brick drop at timeBetweenDrops
❷Invoke ("DropBrick", timeBetweenDrops);

    // speed up the drop time to make bricks drop sooner
❸if (timeBetweenDrops > 0.5f)
    ❹timeBetweenDrops -= 0.05f;
```

To make sure there's only one active call set up for the `DropBrick` function, we use the `CancelInvoke` command ❶. `CancelInvoke` takes the name of the function and removes all calls scheduled by `Invoke`, ensuring that only one will be waiting in the queue. This just makes all the events easier to manage.

After the queue is cleared by `CancelInvoke`, there's a new call to `Invoke` at ❷ that schedules a call to the `DropBrick` function at the time held in the variable `timeBetweenDrops`.

We want `timeBetweenDrops` to decrease as the game progresses so that bricks drop more frequently and the game gets more difficult. To do this, an `if` statement ❸ checks whether the value of `timeBetweenDrops` is greater than 0.5. If it's more than 0.5, `timeBetweenDrops` decreases by 0.05 ❹. Once `timeBetweenDrops` is less than 0.5, we stop subtracting from it; otherwise, the game would become way too hard to play.

Finally, let's add the code to increase `dropGravity` to make the bricks fall faster, which also makes the game harder as it goes on:

```
    dropGravity += 0.1f;
}
```

The closing curly bracket ends the function. That's all the code you need to make it rain bricks! Don't forget to save your script before you return to Unity.

Next, you need to tell Unity which prefab to use for instantiating bricks by adding a reference to the `brick` prefab in the Inspector panel. Click the `GameController` in the Hierarchy panel. The Inspector should show the Component and its publically available properties, which will look something like Figure 5-9. The Hazard Prefab field will display `None (GameObject)`, meaning that there's no reference set up yet for the brick object. Let's fix that now.

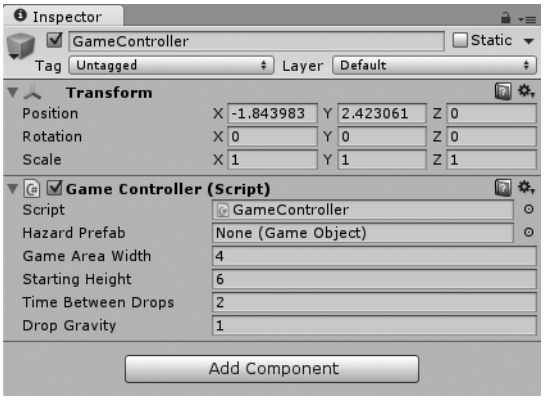


Figure 5-9: The Inspector shows the `GameController` Component.

Click the *Prefabs* folder in the Project panel to highlight it. The asset browser should show its contents. Click and drag the `brick` prefab into the Hazard Prefab field in the Inspector. Drop it there to change the entry from `None (GameObject)` to `brick (GameObject)`.

Time to give the game a go! Click **Play** to preview the game. You should be able to move the player left and right to avoid the falling bricks. Congratulations on completing the game!

Adding Polish

In game development, once all the main features of a game are working, the final stages include the *polish* phase. This phase can include fixing bugs, but it's more about adding finishing touches, such as inserting transitions between levels, smoothing out movement, or generally improving the flow of the game.

At this stage, the game we've been working on feels a bit rough. The way the bricks just disappear when they hit the ground doesn't make much sense: why are they disappearing? Are they leaving this dimension? Being stolen by aliens? By adding a simple particle effect, the bricks will smash and break apart when they hit the ground. Although this isn't a critical addition to the game, it is a nice feature to give the game a little polish.

Create a Smashing Brick Particle Effect

Unfortunately, the particle system is still intended to be used primarily in 3D games. For that reason, it's not set up to use the 2D sprite system, and any image you want to use in a particle effect needs to be set up as a 3D texture. To do this, let's grab the brick image you already have and duplicate it.

Select `brick_tile` in the Project panel, and press CTRL-D (CMD-D on a Mac) to duplicate the file. Click the new `brick_tile 1` object to view its Import Settings in the Inspector (Figure 5-10).

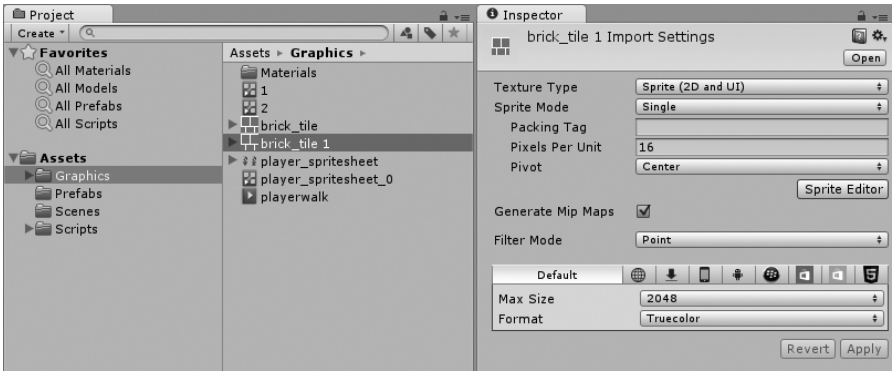


Figure 5-10: The Import Settings for the texture selected in the Project panel are shown in the Inspector.

Click the Texture Type drop-down menu and change its value from Sprite (2D and UI) to Texture; then click the **Apply** button.

Add the Particle Effect

Next, you'll add the particle effect. Right-click anywhere in the Hierarchy panel to show the drop-down menu; then select **Particle System**. The standard particle texture is a little white blob: you should see lots of them in the Scene and Game panels. You need to change the texture to that brick texture you just duplicated.

In the *Graphics* folder in the Project panel, click and drag the new `brick_tile 1` texture on top of the `Particle System` object in the Hierarchy. When you drop the texture onto the `Particle System`, those white blobs will turn into the brick pattern.

What actually happened was that the texture you dragged onto the GameObject was automatically placed into the Material used by the particle system. In the Inspector, you'll see the Particle System Component. Scroll down to find the Renderer section and click it to see its options. In this section is a Material field, and it's set to the brick material! Unity put it there for you.

Figure 5-11 shows the settings I used for my particle effect: copy them into yours. But feel free to experiment. Play around with the numbers in the Inspector to get creative!

Next, to make sure the particle effect goes away when the brick GameObjects are destroyed, you'll need to add a script to the `Particle System` object.

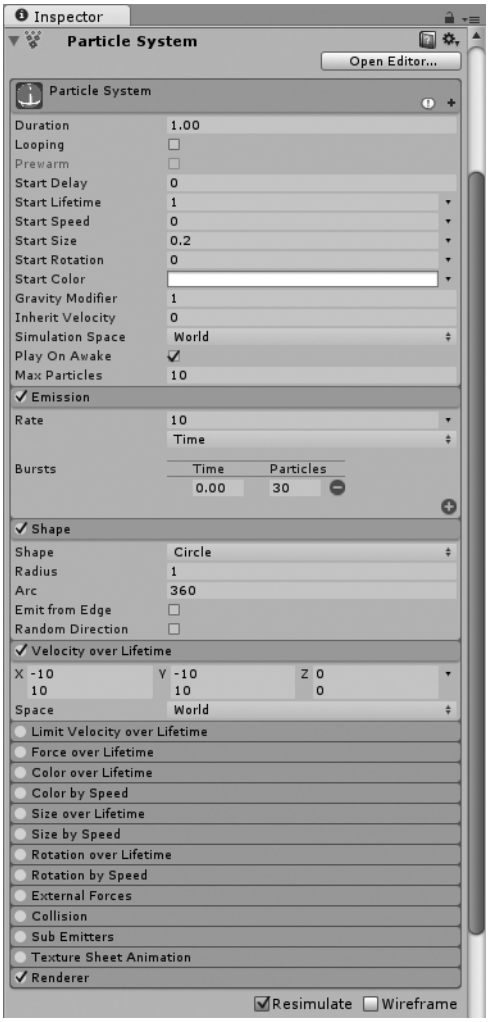


Figure 5-11: The Particle System settings I used to make an exploding brick effect

Add a Script

Select the Particle System from the Hierarchy. In the Inspector, click the **Add Component** button. In the drop-down menu, select **New Script**, name the new script *DestroyInTime*, and click the **Create and Add** button. Double-click the script to open it in the script editor.

You'll create a function to destroy the particle effect that this Component is attached to. Add a float variable called `destroyTimeSecs` to use as the timer to the body of the `DestroyInTime` class declaration:

```
public float destroyTimeSecs = 1;
```

Next, add this code to the Start function:

```
        Invoke ("DestroyThis", destroyTimeSecs);
```

Now when the Start function is called, this Invoke statement will call the DestroyThis function in the time set by destroyTimeSecs.

The DestroyThis function has a simple statement that tells Unity to destroy the GameObject that this script is attached to. Add this code after the Start function's closing bracket:

```
        void DestroyThis () {
             Destroy (gameObject);
        }
```

It's important to note the difference between the capitalized GameObject and gameObject. When you use the lowercase versions of transform or gameObject, as we do here, you're referring to the Transform or GameObject that the script is attached to. When you use the capitalized version, you're saying "the type of object is a GameObject" or "the type of object is a Transform." The capitalized versions of GameObject or Transform tell the engine what type of object you are either creating or expecting as a return value from another function and don't refer to a particular instance of either.

The full script looks like this:

```
using UnityEngine;
using System.Collections;

public class DestroyInTime : MonoBehaviour {

     public float destroyTimeSecs = 1;

     // Use this for initialization
     void Start () {
          Invoke ("DestroyThis", destroyTimeSecs);
     }

     void DestroyThis () {
          Destroy (gameObject);
     }
}
```

Double-check that your code looks right. Then save your script and head back to Unity.

Drag Particle System from the Hierarchy into the *Prefabs* folder in the Project panel to make a new prefab. Right-click the Particle System object in the Hierarchy and then click **Delete** to remove it from the scene. Don't

worry! The prefab has all the necessary information, so the game will be able to add new effects to the scene as needed. Add the particle effect to the *Hazard* script.

Now you need to use the `Instantiate` command to trigger the explosion when the brick is destroyed. Earlier in this chapter, you added the *Hazard* script to the brick so that the brick would be destroyed when it hit something.

In the *Prefabs* folder in the Project panel, click the `brick` prefab so the Inspector panel changes to show the properties of `brick`. You should see the *Hazard* script as a Component in the Inspector. Double-click the script to open it in the script editor program.

Add the following variable to hold a reference to the new particle effect prefab, just before the `Start()` function:

```
public Transform particlePrefab;
```

The collision system triggers a function named `OnCollisionEnter2D` when the brick hits something. You'll instantiate the particle effect there, just before the `Destroy` statement that you wrote earlier. The full `OnCollisionEnter2D` function should look like this:

```
void OnCollisionEnter2D() {
    Instantiate (particlePrefab, transform.position, Quaternion.identity);
    Destroy (gameObject);
}
```

Save the script and then return to Unity. A small spinning icon appears in the bottom right of the editor window (Figure 5-12) to indicate that the scripts are being compiled. It disappears quickly, so if you didn't see it, don't worry.

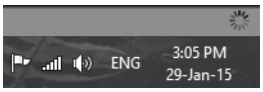


Figure 5-12: The spinning icon appears in the bottom right when the editor is busy compiling scripts.

The next task is to tell the *Hazard* script where to find the particle effect by creating a reference to it in the Inspector.

In the *Prefabs* folder in the Project panel, click the `brick` prefab to show its properties in the Inspector. You'll see the `hazard` Component again. Find the `Particle System` prefab back in the *Prefabs* folder and then drag and drop it into the Particle Prefab field on the `hazard` Component. The Inspector for the `brick` prefab should now look something like Figure 5-13.

Test the game by clicking the **Play** button in the scene tools. Now, whenever a brick hits the ground or the player, your particle effect should occur when the brick disappears. It's a small effect that adds to the overall feel of the game: it makes more sense for bricks to explode on impact than just disappear!

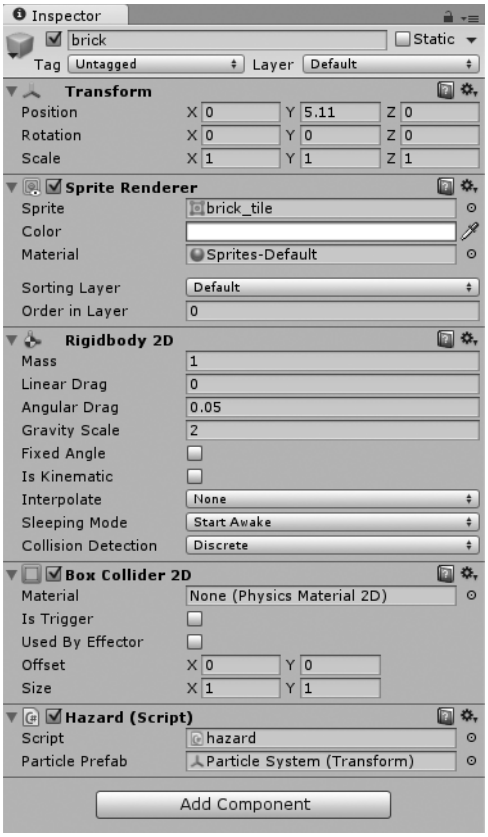


Figure 5-13: The `brick` prefab in the Inspector panel. Note the `Hazard (Script)` Component and its Particle Prefab field.

Flip the Player

You're almost done polishing your brick-dodging game. Right now, the player looks a bit silly because he's always facing left, even when he moves to the right. A little extra programming can fix this. By modifying the scale of the `Transform` Component either in the Inspector or through code, you can flip sprites around.

Select the player sprite if it isn't already selected. Then look in the Inspector for the Transform section. Change the `x` value of Scale from 1 to –1. The sprite flips around to face the right side of the screen. Change the value of scale back to 1, and the sprite flips to face left again. You can access this value in code as well, which is exactly what the *PlayerControl* script will do to change the sprite's direction whenever it needs to.

In the *Scripts* folder (Figure 5-14) in the Project panel, double-click the *PlayerControl* script to open it in the script editor.

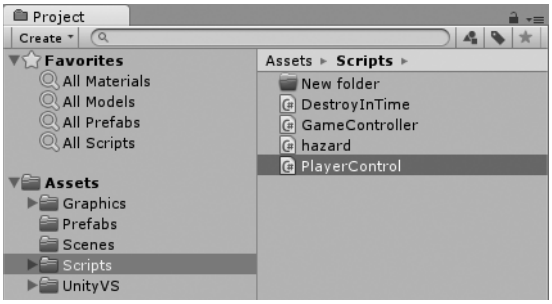


Figure 5-14: The PlayerControl *script in the* Scripts *folder*

Scroll down to the `Update` function to add two lines of code. You'll place the first one inside the curly brackets to move right. Add the following just after `myTransform.Translate (0.1f, 0, 0);`:

```
        myTransform.localScale = new Vector3(-1, 1, 1);
```

The second line sets the `x` scale in the opposite direction, which goes between the curly brackets to move left:

```
        myTransform.localScale = new Vector3(1, 1, 1);
```

Your updated `Update` function should now look like this:

```
// Update is called once per frame
void Update () {
     // move right
     if (Input.GetAxis("Horizontal")>0 && myTransform.position.x < gameWidth)
     {
          myTransform.Translate (0.1f, 0, 0);
          myTransform.localScale = new Vector3(-1, 1, 1);
     }
     // move left
     if (Input.GetAxis("Horizontal")<0 && myTransform.position.x > -gameWidth)
     {
          myTransform.Translate (-0.1f, 0, 0);
          myTransform.localScale = new Vector3(1, 1, 1);
     }
}
```

Test the game to make sure the character faces the same direction that he moves when you press the left or right arrow keys. Flipping a sprite through code is pretty neat, huh?

Closing Thoughts

You've completed a brick-dodging game. Good job! You learned how to program some player controls and learned basic collision handling. You know how to add a player character and move it around the screen, as well as how to apply some simple physics from the 2D physics engine to create falling blocks.

You also practiced building a very basic game structure. The game controller controls all the main game functions. I use this structure for all my games, and it works for just about anything.

Organizing your project structures can make future game development and debugging more straightforward. When problems occur, it's easier to pinpoint where they're happening when you use separate specialized scripts rather than large scripts that lump many purposes together.

In Chapter 6, I'll introduce you to Unity's graphical user interface. We'll look at the available user interface elements and some helpful tools for building interfaces.