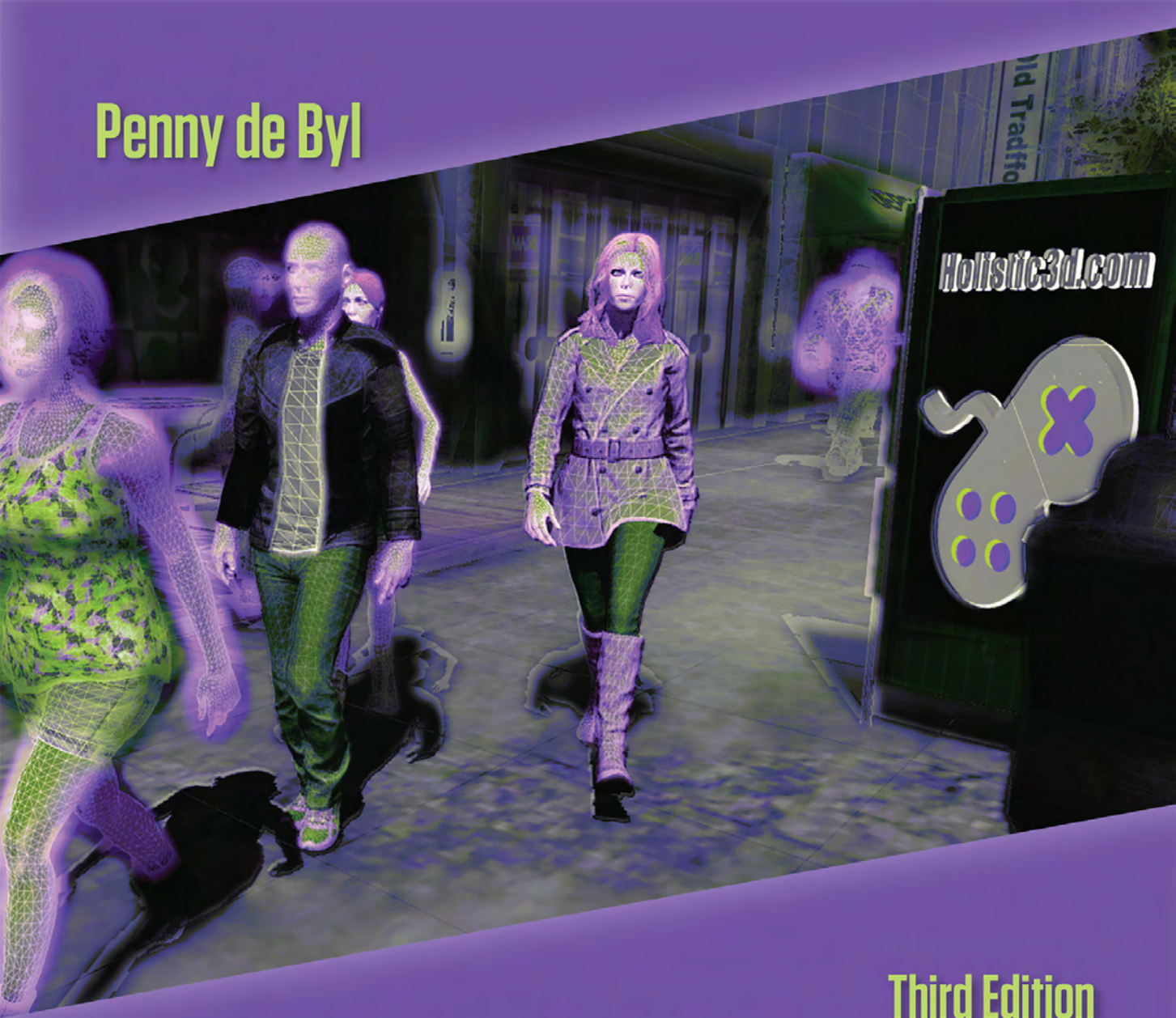
Penny de Byl



Third Edition

HOLISTIC GAME DEVELOPMENT WITH UNITY

An All-in-One Guide to Implementing Game Mechanics, Art, Design and Programming

CRC Press
Taylor & Francis Group

Holistic Game Development with Unity

Holistic Game Development with Unity

An All-in-One Guide to Implementing Game Mechanics, Art, Design and Programming

Third Edition

Penny de Byl

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business

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

Printed on acid-free paper

International Standard Book Number-13: 978-1-138-48073-5 (Hardback)
978-1-138-48062-9 (Paperback)

Library of Congress Cataloging-in-Publication Data

Names: De Byl, Penny, author.
Title: Holistic game development with Unity : an all-in-one guide to implementing game mechanics, art, design and programming/ Penny de Byl.
Description: Third edition. | Boca Raton : Taylor & Francis, 2019.
Identifiers: LCCN 2018058220 | ISBN 9781138480629 (pbk. : alk. paper) |
ISBN 9781138480735 (hardback : alk. paper)
Subjects: LCSH: Unity (Electronic resource) | Computer games--Programming. | Video games--Design.
Classification: LCC QA76.76.C672 D42 2019 | DDC 794.8/1525--dc23
LC record available at HYPERLINK "https://protect-us.mimecast.com/s/Ij6pCG6Y9jf1gp21yiQBwxB?domain=lccn.loc.gov" https://lccn.loc.gov/2018058220

Visit the Taylor & Francis Web site at
http://www.taylorandfrancis.com

and the CRC Press Web site at
http://www.crcpress.com

Contents

Preface

About This Book

I first decided to write this book in 2010 when I found existing literature for budding game designers, artists, and programmers tended to focus on only one specific vein of games development, that being either a design, artistic, or programming book. Those with artistic talents and ideas for games could not find a good resource to ease them into programming. On the other hand, programming texts tended to be dry and ignore the visual aspect.

At the time, the face of the game development industry was rapidly changing from a small number of large development teams to much more of a cottage industry consisting of small multi-skilled teams. And today, some eight years later, it is more imperative than ever that individuals are skilled in both art and programming.

Game development tools are also not what they used to be, and rapid game development tools such as Unity are making it a possibility for individuals to make complete games from scratch. It's also becoming almost impossible to write a book about software and have it published before the software is updated. Year after year we see advancements that quickly make printed material obsolete.

To address all these issues, this book is written for the artist who wants to program and the programmer who wants some pointers about using game art. In the beginning, I started writing just for artists, but soon came to realize the content was equally as relevant to those wanting to learn how to start programming games. In addition, the content inside these pages is organized by theory first, followed by application and practice in Unity. While the theory will remain relevant long into the future, the versions of Unity and how the interface changes will not, although, to Unity's credit, the interfaces from one version to another are mostly the same. What you'll find inside these pages are code and techniques that work from versions 5.6 through to 2019.

How This Book Is Organized

This book has been written with *artists who want to learn how to develop games* and *programmers who want to learn about using art in games* in mind. It approaches game development in a unique combination of teaching programming that keeps design in mind, because programming a games graphical user interface is entirely different from making it look good.

Learning about how design impacts programming and vice versa is a logical way to introduce both sides of the game development coin to game creation.

All chapters focus on sets of mechanical functions existing within games:

- Chapter 1, The Art of Programming Game Mechanics, explains the roles both art and programming play in creating games and explores the limitations of having one without the other. In addition, the complementary nature of digital art and programming is established.
- Chapter 2, Real World Mechanics, examines the branch of physics dealing with the study of motion. Motion is a fundamental idea in all of science that transcends the computer screen into virtual environments. This chapter examines kinematics, which describes motion, and dynamics, examining the causes of motion with respect to their use in computer games. It introduces the physical properties of the real world and demonstrates how a fundamental understanding of mathematics, physics, and design principles is critical in any game environment. Composition, rules of animation, and design principles are introduced in parallel with technical considerations, mathematics, and programming code that controls and defines the movement of characters, cameras, environments, and other game objects.
- Chapter 3, Animation Mechanics, studies the technical nature of 2D and 3D animated models. The reader will develop skills with respect to the programmatic control of their own artwork, models, and/or supplied assets in a game environment. Elementary mathematics, physics, and programming concepts are introduced, demonstrating the concepts of keyframes, animation states, and the development of dynamic character movement and sprite animation.
- Chapter 4, Game Rules and Mechanics, introduces common generic game mechanics such as matching, sorting, managing, and hitting. Examples of how each of these is visually represented in a game and the programming that controls them are explained in depth. Common algorithms and data structures used for each mechanic are worked through with the reader, integrating the key art assets where appropriate.
- Chapter 5, Character Mechanics, explains simple artificial intelligence algorithms to assist the reader in creating their own believable non-player characters. Animation states and techniques covered in Chapter 3 are integrated with game-specific data structures and algorithms to control the behavior of characters, from the flocking of birds to opponents that follow and interact with the player.
- Chapter 6, Player Mechanics, presents the code and artwork that will be deployed to develop graphical user interfaces and maintain player states. It includes details about the development of inventory systems, heads-up displays, and character–environment interaction.

- Chapter 7, Environmental Mechanics, reveals the fundamental concepts in creating and optimizing game environments. It covers techniques that range from adding detail to environments to make them more believable to tricks for working with large maps and weather simulations.
- Chapter 8, Mechanics for External Forces, examines issues related to developing games while keeping in mind the new plethora of input devices, social data, GPS locators, motion sensors, augmented reality, and screen sizes. Practical advice is included for using Unity to deploy games that leverage touch screens, accelerometers, and networking to the iPhone, iPad, and Android mobile devices.

The Companion Website

The website accompanying this book is http://www.holistic3d.com. It contains all of the files referred to in the workshops, finished examples, and other teaching and learning resources.

More Holistic3D Resources

Join Penny's students online on Discord at https://discord.gg/su2zar2 or Facebook at https://www.facebook.com/groups/hgdev.

To accompany this book, there is also a YouTube channel updated constantly with new techniques and game development examples in Unity. This can be found at https://www.youtube.com/c/holistic3d. There are also several high-quality online video courses produced by the author, which you can access at a heavily discounted rate with the following coupons. These courses cover a vast range of topics from animation to procedural terrain generation to machine learning. You can find each here:

Augmented Reality:
https://www.udemy.com/augmented_reality_with_unity/?couponCode=H3DGAMEDEVBOOK
Procedural Terrain Generation:
https://www.udemy.com/procedural-terrain-generation-with-unity/?couponCode=H3DGAMEDEVBOOK
Machine Learning:
https://www.udemy.com/machine-learning-with-unity/?couponCode=H3DGAMEDEVBOOK
Shader Development:
https://www.udemy.com/unity-shaders/?couponCode=H3DGAMEDEVBOOK
Artificial Intelligence:
https://www.udemy.com/artificial-intelligence-in-unity/?couponCode=H3DGAMEDEVBOOK
Minecraft Voxel Worlds:
https://www.udemy.com/unityminecraft/?couponCode=H3DGAMEDEVBOOK

C# Programming:
https://www.udemy.com/naked_cs/?couponCode=H3DGAMEDEVBOOK
Networking:
https://www.udemy.com/unet_intro/?couponCode=H3DGAMEDEVBOOK
Animation:
https://www.udemy.com/mastering-3d-animation-in-unity/?couponCode=H3DGAMEDEVBOOK

Acknowledgments

First, I'd like to thank my editor, Sean Connelly, who has kept my project on track. His encouragement and enthusiasm for the book have been highly motivating. In addition, thanks must go to Mark Ripley of Effervescing Elephant Interactive, who acted as the initial technical editor and provided valuable insight on game programming with Unity, as well as Rachel, Ramesh, and Joy from the Unity educational team, without whose enthusiasm this third edition of the book wouldn't have materialized.

Next, I'd like to acknowledge Unity3d, who have helped shape the content of the book through their reviews and the development of a truly inspirational game development tool; as well as all of the forum contributors who have freely shared their ideas and code to answer all conceivable game development questions. The forums at http://forums.unity3d.com are an invaluable knowledge base.

Finally, I'd like to thank my family, Daniel, Tabytha, and Merlin (my labrador). Daniel has been an absolute rock. His knowledge of Microsoft Word formatting still leaves me amazed, and his proofreading and testing of all the workshops have saved so much time, which has been an invaluable contribution to this work (i.e., if the code in this book doesn't work for you—blame him! ☺). Tabytha has also been a source of inspiration, as she's now at an age where programming and mathematics are becoming fascinating to her. Her journey with Unity is just beginning. As always, Merlin has provided constant companionship and copious amounts of fur by sitting constantly under the desk at my feet, snoring and snuffling.

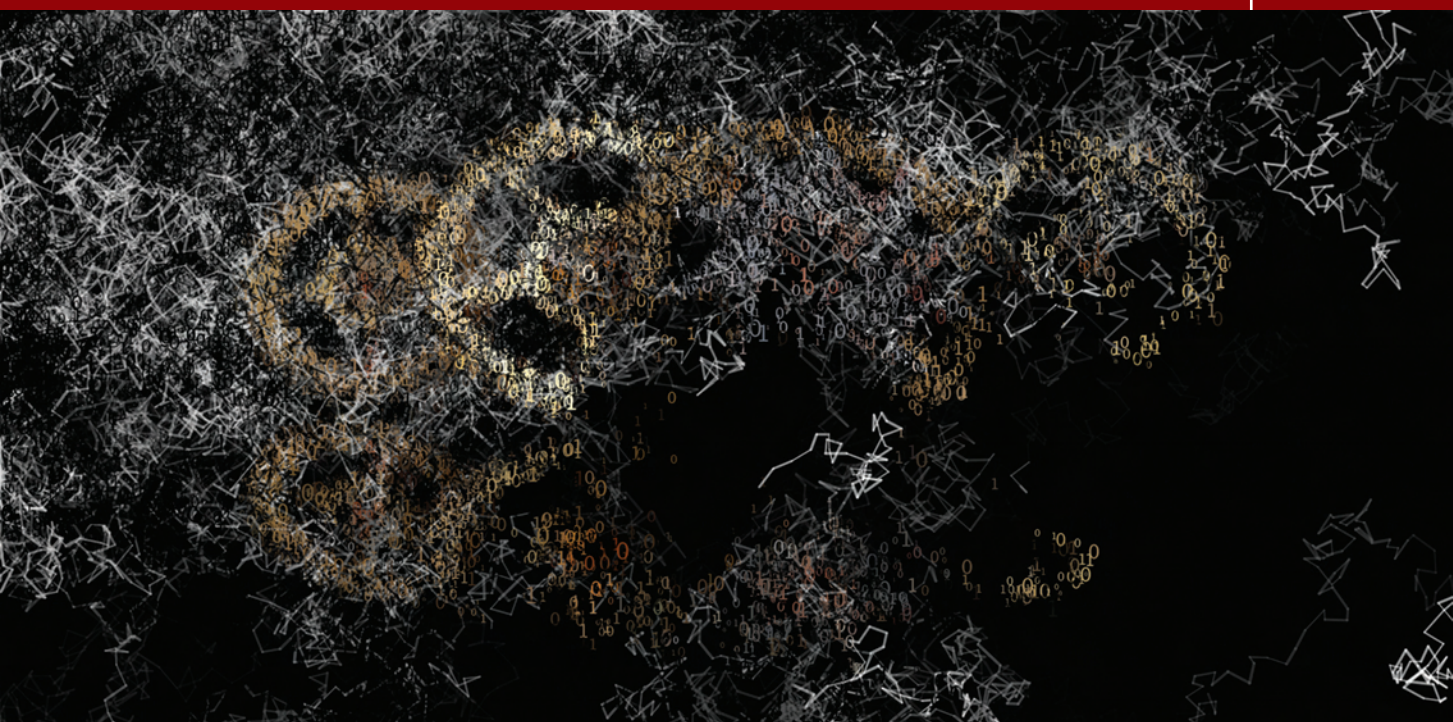
I should also thank my close friends, Kayleen and James, who've provided editorial, art directing, code testing, and rewriting support over the years; and especially Adrian, who's been a hard-working little minion for me over the final few weeks of pulling this third version together. His dedication to the project and occasional comic relief have been greatly appreciated.

To me, game development is the quintessential seam where the tectonic plates of programming and art meet: It is where both domains really start to make sense. If you are reading this, I hope you feel the same.

Author

Dr. Penny de Byl, former university professor of games and multimedia, is the founder of Holistic3D.com, an online education provider for all things games related. She has researched and taught computer science, computer graphics, animation, artificial intelligence, and mobile game development for over 25 years and has students working for Ubisoft, Apple, The Binary Mill, and Unity. Penny hosts the popular Unity development YouTube channel, Holistic3D.

CHAPTER 1



The Art of Programming Mechanics

Everyone can be taught to sculpt: Michelangelo would have had to be taught how not to. So it is with the great programmers.

Alan Perlis

1.1 Introduction

In 1979, art teacher Betty Edwards published the acclaimed *Drawing on the Right Side of the Brain*. The essence of the text taught readers to draw what they saw, rather than what they *thought* they saw. The human brain is so adept at tasks such as pattern recognition that we internally symbolize practically everything we see and regurgitate these patterns when asked to draw them on paper. Children do this very well. The simplicity in children's drawing stems from their internal representation for an object. Ask them to draw a house and a dog and you will get something you and they can recognize as a house and dog (or, more accurately, the icon for a house and dog), but something that is far from what an actual house and dog look like.

FIG 1.1 Dogs in the yard of a castle, by Tabytha de Byl, age 4.

This is evident in the child's drawing in Figure 1.1. The title of the book, *Drawing on the Right Side of the Brain*, also suggests that the ability to draw should be summoned from the side of the brain traditionally associated with creativity and that most *bad* drawings could be blamed on the left.

Different intellectual capability is commonly attributed to either the left or the right hemispheres, with the left side being responsible for the processing of language, mathematics, numbers, logic, and other such computational activities, whereas the right side deals with shapes, patterns, spatial acuity, images, dreaming, and creative pursuits. From these beliefs, those who are adept at computer programming are classified as left brained and artists as right brained. The segregation of these abilities to either side of the brain is called *lateralization*. While lateralization has been generally accepted and even used to classify and separate students into learning style groups, it is a common misconception that intellectual functioning can be separated so clearly.

In fact, the clearly defined left and right brain functions are a *neuromyth* stemming from the overgeneralization and literal isolation of the brain hemispheres. While some functions tend to reside more in one side of the brain than the other, many tasks, to some degree, require both sides. For example, many numerical computation and language activities require both hemispheres. Furthermore, the side of the brain being utilized for specific tasks can vary among people. Studies have revealed that 97% of right-handed people use their left hemisphere for language and speech processing and 70% of left-handed people use their right hemisphere.

In short, simply classifying programmers as left brainers and artists as right brainers is a misnomer. This also leads to the disturbing misconception that programmers are poor at art skills and that artists would have difficulty in understanding programming. Programming is so often generalized as a logical process and art as a creative process that some find it inconceivable that programmers could be effective as artists and vice versa.

When Betty Edwards suggests that people should use their right brain for drawing, it is in concept, not physiology. The location of the neurons the reader is being asked to use to find their creative self is not relevant. What is important is that Dr. Edwards is asking us to see drawing in a different light—in a way we may not have considered before. Instead of drawing our internalized symbol of an object that has been stored away in the brain, she asks us to draw what we see—to forget what we *think* it looks like. In the end, this symbolizes a switch in thinking away from logic and patterns to images and visual processing.

There is no doubt that some people are naturally better at programming and others at art. However, by taking Edwards' *anyone can draw* attitude, we can also say *anyone can program*. It just requires a little practice and a change of attitude.

1.2 Programming on the Right Side of the Brain

While it is true that pure logic is at the very heart of all computer programs, it still requires an enormous amount of creativity to order the logic into a program. The process is improved greatly when programmers can visualize the results of their code before it even runs. You may liken this to a scene from *The Matrix* where the characters look at screens of vertically flowing green numbers and text, but can visualize the structure and goings on in a photorealistic, three-dimensional virtual reality. To become a good computer programmer, you need to know the language of the code and be able to visualize how it is affecting the computer's memory and the results of running the program.

Learning a computer language is one key to being able to program. However, understanding how the language interacts with the computer to produce its output is even more important. Good programmers will agree that it is easy to switch between programming languages once you have mastered one. The fundamental concepts in each language are the same. In some languages, such as C, C++, C#, JavaScript, Java, and PHP, even the text and layout look the same. The basic code from each aforementioned language to print *Hello World* on the computer screen is shown in Listings 1.1 through 1.6.

Listing 1.1 C

```
#include <stdio.h>
main()
{
     printf("Hello World");
}
```

Listing 1.2 C++

```
#include <iostream>
using namespace std;
void main()
{
     cout << "Hello World" << endl;
}
```

Listing 1.3 C#

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

Listing 1.4 JavaScript (in bold) embedded in HTML

```
<html>
     <head>
         <title> Hello World </title>
     </head>
     <body>
         <script type = "text/javascript">
             document.write('Hello World');
         </script>
     </body>
</html>
```

Listing 1.5 Java

```
class helloworld
{
     public static void main(String args[])
     {
         System.out.println("Hello World");
     }
}
```

Listing 1.6 PHP

```
<?php
     echo "Hello World";
?>
```

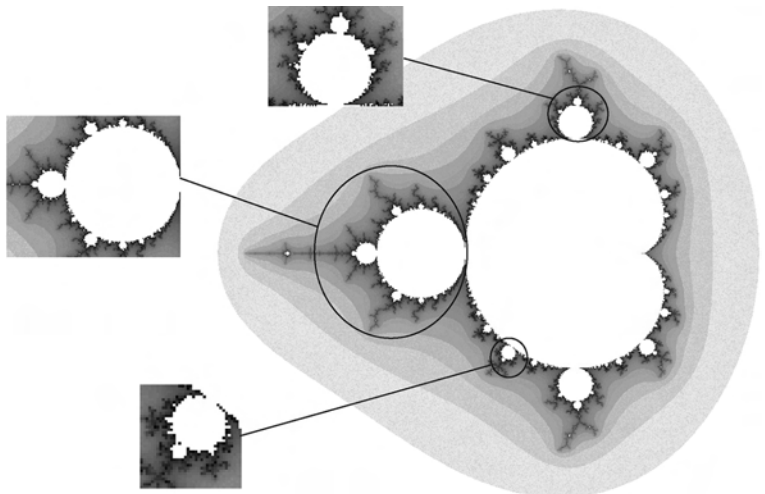


FIG 1.2 The Mandelbrot set and periodicities of orbits.

Umberto Eco, the creator of *Opera Aperta*, described the concept of art as mechanical relationships between features that can be reorganized to make a series of distinct works. This is true of programming, too. The same lines of programming code can be reorganized to create many different programs. Nowhere is this shared art/programming characteristic more obvious than in fractals.

Fractals are shapes made up of smaller self-similar copies of themselves. The famous Mandelbrot set or *Snowman* is shown in Figure 1.2. The whole shape is made up of smaller versions of itself. As you look closer, you will be able to spot tens or even hundreds of smaller snowman shapes within the larger image.

A fractal is constructed from a mathematical algorithm repeated over and over where the output is interpreted as a point and color on the computer screen. The Mandelbrot set comes from complex equations, but not all fractal algorithms require high-level mathematical knowledge to understand.

The Barnsley fern leaf is the epitome of both the creative side of programming and the algorithmic nature of art. Put simply, the algorithm takes a shape, any shape, and transforms it four times, as shown in Figure 1.3. It then takes the resulting shape and puts it through the same set of transformations. This can be repeated ad infinitum; however, around 10 iterations of this process give a good impression of the resulting image (see Figure 1.4).

Creating images with these types of algorithmic approaches is called *procedural* or *dynamic generation*. It is a common method for creating

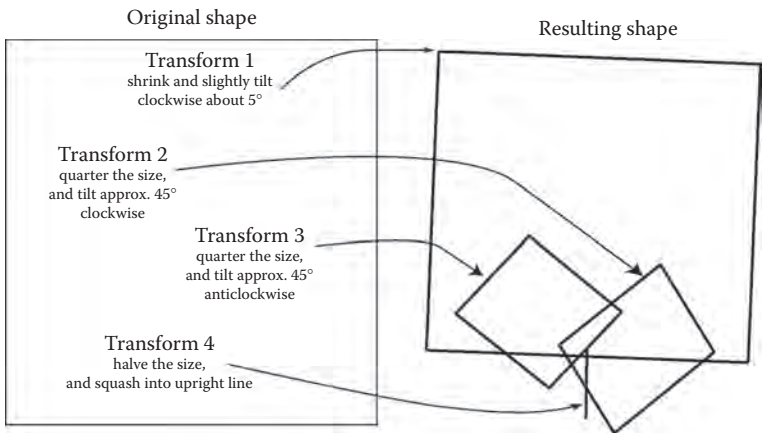


FIG 1.3 Transformations of Barnsley's fern leaf.

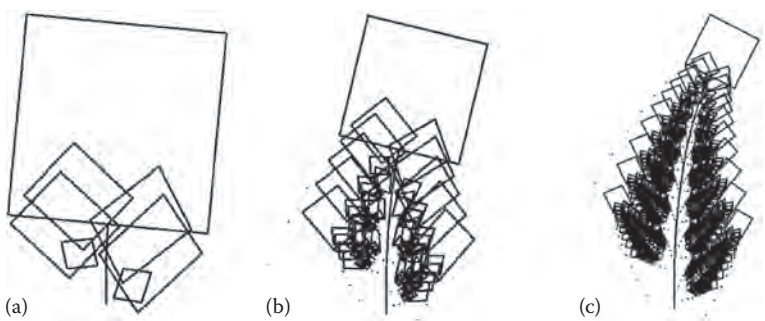


FIG 1.4 Three iterations of Barnsley's fern leaf transformations after (a) 2 iterations, (b) 5 iterations, and (c) 10 iterations.

assets such as terrain, trees, and special effects in games. Although procedural generation can create game landscapes and other assets before a player starts playing, procedural generation comes into its own *while* the game is being played. Programming code can access the assets in a game during run time. It can manipulate an asset based on player input. For example, placing a large hole in a wall after the player has blown it up is achieved with programming code. This can only be calculated at the time the player interacts with the game, as beforehand a programmer would have no idea where the player would be standing or in what direction he would shoot. The game *Fracture* by Day 1 Studios features dynamic ground terrains that lift up beneath objects when shot with a special weapon.

For Research

Procedural Generation in Unity

The Unity website has a project with numerous procedural generation demonstrations. At this point in your game-development learning journey, you may not be able to understand the underlying code, but the examples will show you what is possible and the types of things you will be able to achieve by the end of this book. The Unity project can be downloaded from https://assetstore.unity.com/packages/essentials/tutorial-projects/procedural-examples-5141.

A purpose-built programming language for creating art is *Processing*. The syntax of the code is not unlike JavaScript or C# and contains all the fundamental programming concepts you will learn about in Section 1.4. A book entitled *Creating Procedural Artworks with Processing: A Holistic Guide* by Dr. Penny de Byl that teaches these techniques is available on Amazon. The image in Figure 1.5 was created using Processing by randomly plotting circles and drawing a series of curves from a central location to each circle. Art created by Casey Reas, shown in Figure 1.6, programmed with Processing, has been displayed at the DAM Gallery in Berlin.

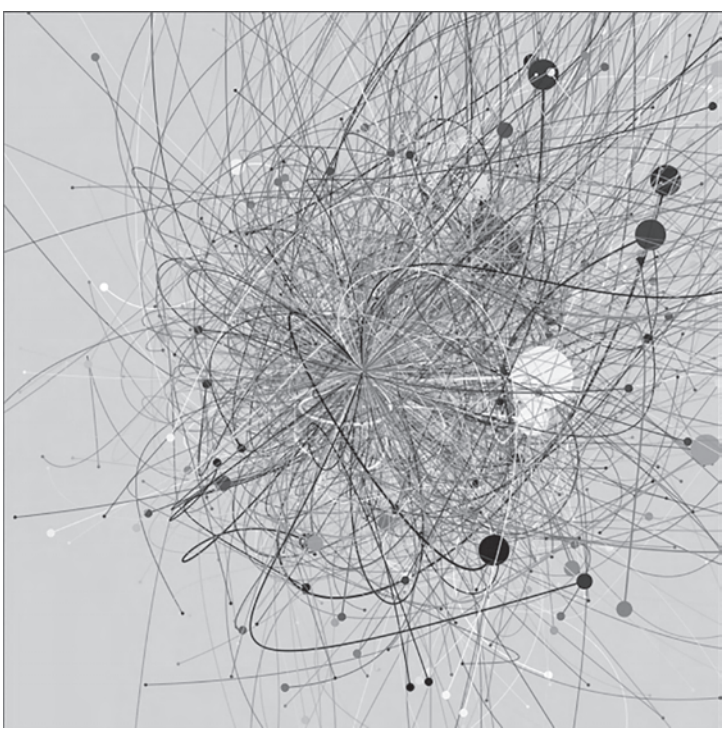


FIG 1.5 An image created with Processing.

FIG 1.6 Artwork created by Casey Reas using Processing, as exhibited at DAM Gallery, Berlin.

For Research
Getting Started with Processing
If you're interested in learning more about Processing and drawing images with programming code, you can download the open source language and find tutorials at http://processing.org.

1.3 Creating Art from the Left Side of the Brain

Most people know what they like and do not like when they see art. However, if you ask them why they like it, they may not be able to put their thoughts into words. No doubt there are some people who are naturally gifted with the ability to draw and sculpt and some who are not. For the artistically challenged, however, hope is not lost. This is certainly Betty Edwards' stance.

A logical approach to the elements and principles of design reveals rules one can apply to create more appealing artwork. They are the mechanical relationships, alluded to by Umberto Eco, that can be used as building blocks to create works of art. These fundamentals are common threads found to run through all good artwork. They will not assist you in being creative and coming up with original art, but they will help in presentation and visual attractiveness.

The elements of design are the primary items that make up drawings, models, paintings, and design. They are *point*, *line*, *shape*, *direction*, *size*, *texture*, *color*, and *hue*. All visual artworks include one or more of these elements.

In the graphics of computer games, each of these elements is as important to the visual aspect of game assets as they are in drawings, painting, and sculptures. However, as each is being stored in computer memory and processed by mathematical algorithms, their treatment by the game artist differs.

1.3.1 Point

All visual elements begin with a point. In drawing, it is the first mark put on paper. Because of the physical makeup of computer screens, it is also the fundamental building block of all digital images. Each point on an electronic screen is called a *pixel*. The number of pixels visible on a display is referred to as the *resolution*. For example, a resolution of 1024 × 768 is 1024 pixels wide and 768 pixels high.

Each pixel is referenced by its *x* and *y* Cartesian coordinates. Because pixels are discrete locations on a screen, these coordinates are always in whole numbers. The default coordinate system for a screen has the (0,0) pixel in the upper left-hand corner. A screen with 1024 × 768 resolution would have the (1023,767) pixel in the bottom right-hand corner. The highest value pixel has *x* and *y* values that are one minus the width and height, respectively, because the smallest pixel location is referenced as (0,0). It is also possible to change the default layout depending on the application being used such that the *y* values of the pixels are flipped with (0,0) being in the lower left-hand corner or even moved into the center of the screen.

1.3.2 Line

On paper, a line is created by the stroke of a pen or brush. It can also define the boundary where two shapes meet. A line on a digital display is created by coloring pixels on the screen between two pixel coordinates. Given the points at the ends of a line, an algorithm calculates the pixel values that must be colored in to create a straight line. This is not as straightforward as it sounds, because the pixels can only have whole number coordinate values. The *Bresenham line algorithm* was developed by Jack E. Bresenham in 1962 to effectively calculate the best pixels to color in to give the appearance of a line. Therefore, the line that appears on a digital display can only ever be an approximation to the real line, as shown in Figure 1.7.

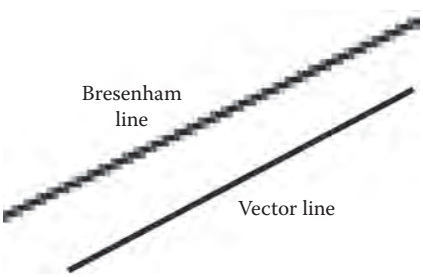


FIG 1.7 A real line and a Bresenham line.

1.3.3 Shape

A shape refers not only to primitive geometrics such as circles, squares, and triangles, but also to freeform and nonstandard formations. In computer graphics, polygons are treated as they are in geometry: a series of points called *vertices* connected by straight *edges*. By storing the coordinates of the vertices, the edges can be reconstructed using straight-line algorithms. A circle is often represented as a regular polygon with many edges. As the number of edges increases, a regular polygon approaches the shape of a circle.

Freeform objects involve the use of curves. To be stored and manipulated by the computer efficiently, these need to be stored in a mathematical format. Two common types of curves used include Bezier and non-uniform rational basis spline (NURBS).

A Bezier curve is constructed from a number of control points. The first and last points specify the start and end of the curve and the other points act as attractors, drawing the line toward them and forming a curve, as shown in Figure 1.8. A NURBS curve is similar to a Bezier curve in that it has a number of control points; however, the control points can be weighted such that some may attract more than others.

In computer graphics, a polygon is the basic building block for objects, whether in two dimensions (2D) or three dimensions (3D). A single polygon defines a flat surface onto which texture can be applied. The most efficient way to define a flat surface is through the use of three points; therefore, triangles are the polygon of choice for constructing models, although sometimes you will find square polygons used in some software packages. Fortunately for the artist, modeling software such as Autodesk's 3DS Studio Max and Blender do not require models to be handcrafted from triangles; instead, they automatically construct any objects using triangles as a base, as shown in Figure 1.9.

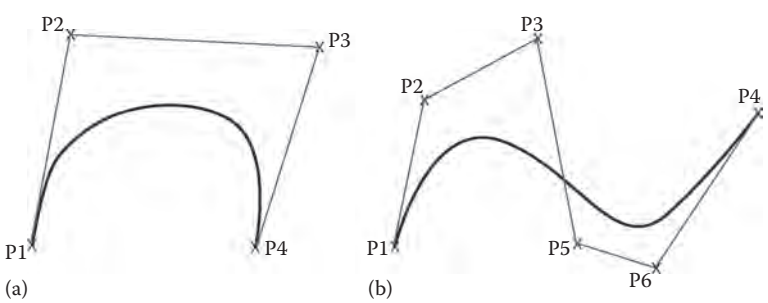


FIG 1.8 (a) A Bezier and (b) a NURBS curve.

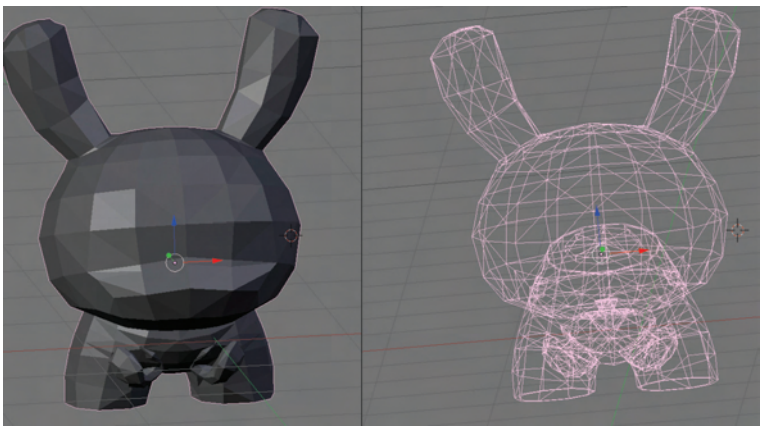


FIG 1.9 A 3D model constructed from triangles in Blender.

The wireframe model that represents a 3D object is called a *mesh*. The number of polygons in a mesh is called the *polycount*. The higher the polycount, the more triangles in the model and the more computer processing power required to render and manipulate the model. For this reason, computer game artists must find a balance between functionality and visual quality, as a high-resolution model is too costly with respect to making the game run slowly. The models must be dynamically processed and rendered in real time. In contrast, animated movie models can be of much higher quality, as they are not rendered in real time. Next time you are playing a game, take a closer look at how the models are constructed.

1.3.4 Direction

Direction is the orientation of a line. Depending on its treatment, it can imply speed and motion. A line can sit horizontal, vertical, or oblique. In computer graphics, physics, engineering, and mathematics, a *Euclidean vector* is used to specify direction. A vector stores information about how to get from one point in space to another in a straight line. Not only does it represent a direction, but also a distance, otherwise called its *magnitude*. The magnitude of a vector is taken from its length. Two vectors can point in the same direction but have different magnitudes, as shown in Figure 1.10a. In addition, two vectors can have the same magnitude but different directions, as shown in Figure 1.10b. A vector with a magnitude of 1 is *normalized*.

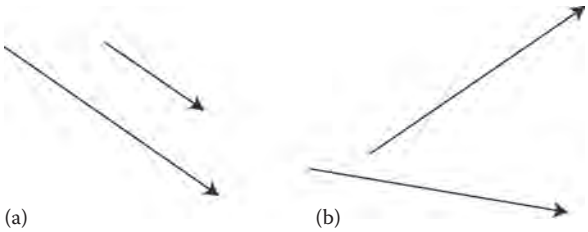


FIG 1.10 (a) Vectors with the same direction but different magnitudes and (b) vectors with the same magnitude but different directions.

Vectors are a fundamental element in 3D games as they describe the direction in which objects are orientated, how they are moving, how they are scaled, and even how they are textured and lit. The basics of vectors are explored further in Chapter 2.

1.3.5 Size

Size is the relationship of the amount of space objects take up with respect to each other. In art and design, it can be used to create balance, focal points, or emphasis. In computer graphics, size is referred to as *scale.* An object can be scaled uniformly in any direction. Figure 1.11 shows a 3D object (a) scaled uniformly by 2 (b), vertically by 3 (c), horizontally by 0.5 (d), and vertically by −1 (e).

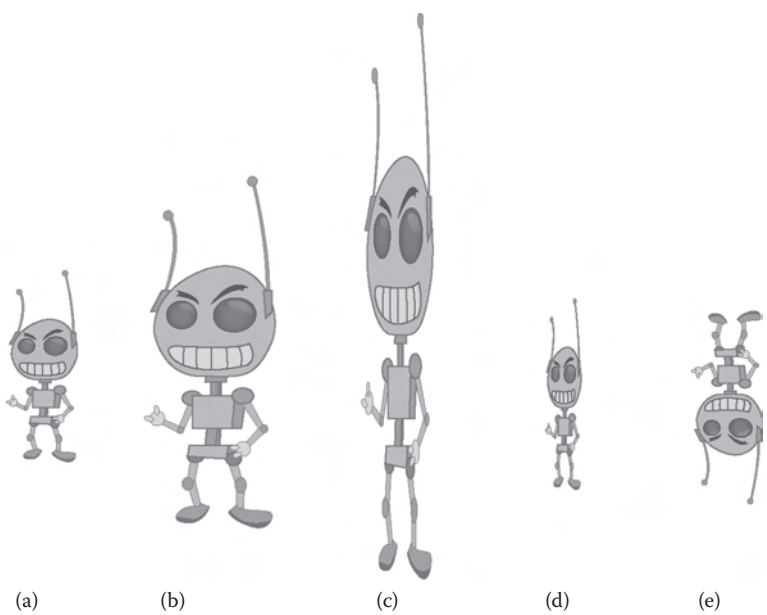


FIG 1.11 A 3D object scaled in multiple ways (a) the original object, (b) scaled uniformly by 2, (c) scaled vertically by 3, (d) scaled horizontally by 0.5, and (e) scaled vertically by −1.

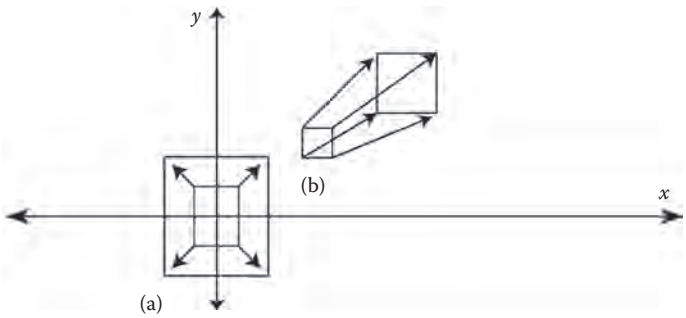


FIG 1.12 How can scaling move an object? (a) When vertices of an object around the origin are scaled, negative values become bigger negative values and the same with positive values. But the original center (0, 0) will remain (0, 0) when scaled. (b) If the vertices are all positive, then scaling them up will just make them bigger. And the final object moves location.

Note in Figure 1.11e how scaling by a negative value flips the object vertically. This can also be achieved uniformly or horizontally using negative scaling values.

Depending on the coordinates of an object, scaling will also move it. For example, if an object is centered around (0,0), it can be scaled remaining in the same place. However, if the object is away from (0,0), it will move by an amount proportional to the scale. This occurs as scaling values are multiplied with vertex coordinates to resize objects. A vertex at (0,0) multiplied by 2, for example, will remain at (0,0), whereas a vertex at (3,2) multiplied by 2 will move to (6,4). This is illustrated in Figure 1.12.

1.3.6 Texture

In art and design, texture relates to the surface quality of a shape or object. For example, the surface could be rough, smooth, or highly polished. In computer games, texture refers not only to the quality, but also to any photographs, colors, or patterns on the surface where the surface is defined by a polygon.

In games, textures are created using image files called *maps*. They are created in Adobe Photoshop or similar software. The image that gives an object its color is called a *texture map*, *color map*, or *diffuse coloring*. All images are mapped onto an object, polygon by polygon, using a technique called *UV mapping*. This aligns points on an image with the vertices of each polygon. The part of the image between the points is then stretched across the polygon. This process is shown on a square polygon in Figure 1.13.

To add a tactile appearance to the surface of a polygon to enhance the base texture, *bump mapping* is applied. This gives the object an appearance of having bumps, lumps, and grooves without the actual model itself being changed. Bump mapping is often applied to add more depth to an object

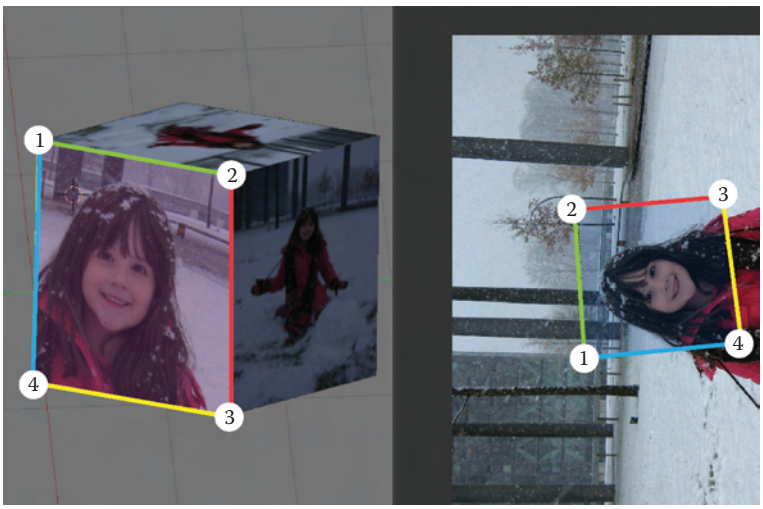


FIG 1.13 The UV mapping process. Vertices of a polygon on an object are mapped to locations on a 2D image.

with respect to the way light and shadow display on the surface. Figure 1.14 illustrates the application of a color and normal map on a soldier mesh taken from Unity.

A variety of other effects also add further texture to a surface. For example, *specular lighting* can make an object look glossy or dull, and *shaders*, small programs that manipulate the textures on a surface, can add a plethora of special effects from bubbling water to toon shading.

1.3.7 Color

In the theory of visual art involving pigments, color is taught as a set of primary colors (red, yellow, and blue) from which all other colors can be created. The color perceived by the human eye is the result of light being reflected off the surface of the artwork. When all of the light is reflected, we see white. When none of the light is reflected, we see black. The resulting color of a mixture of primaries is caused by some of the light being absorbed by the pigment. This is called a *subtractive* color model, as the pigments subtract some of the original light source before reflecting the remainder.

The light from a digital display follows an additive color model. The display emits different colors by combining the primary sources of red, green, and blue light. For this reason, color is represented in computer graphics as a three- or four-numbered value in the format (red, green, blue, and alpha). In some formats, the alpha value is not used, making color a three-value representation.

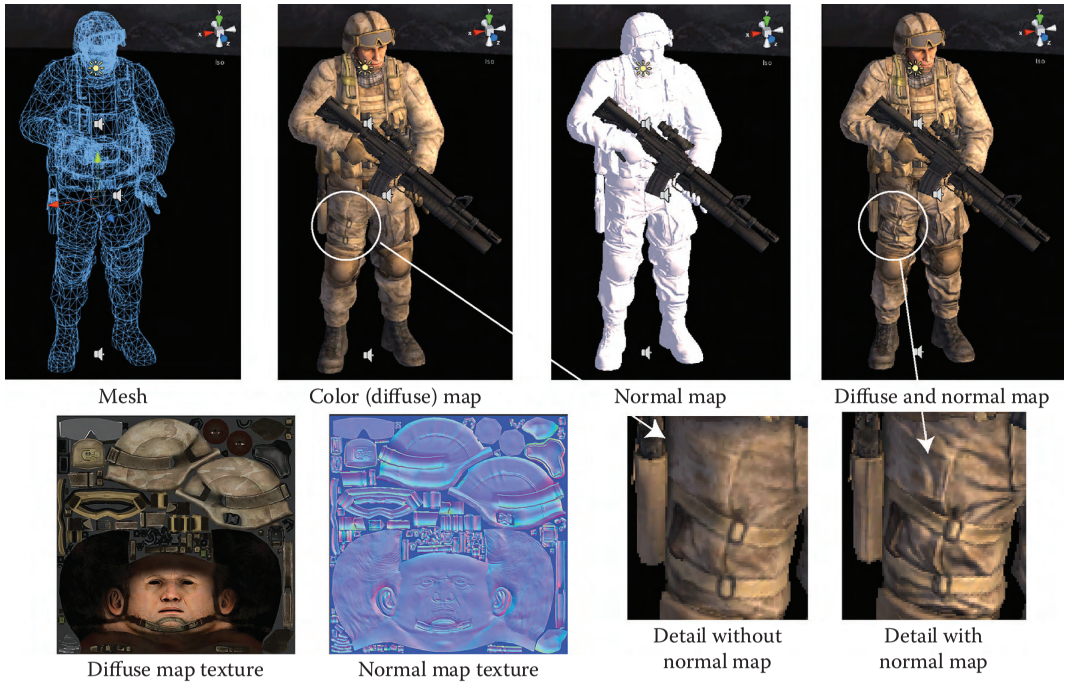


FIG 1.14 A soldier mesh with and without a color map and a normal map.

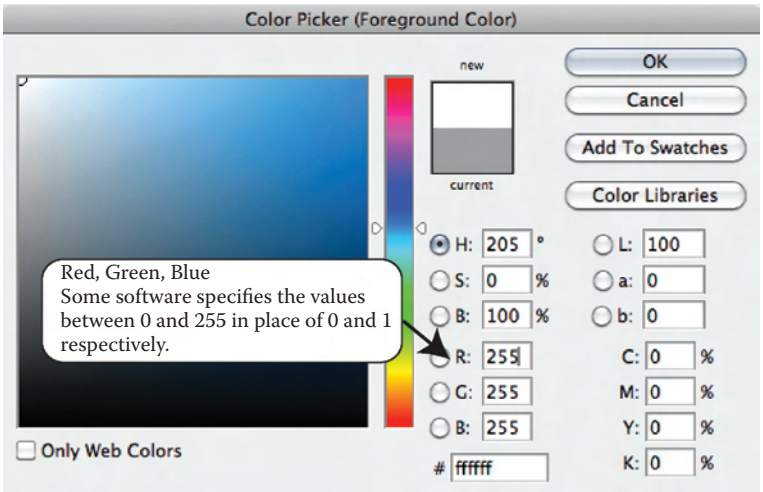


FIG 1.15 The Adobe Photoshop color picker.

Alpha represents the transparency of a color. When a surface has a color applied with an alpha of 0, it is fully transparent; when it has a value of 1, it is totally opaque. A value of 0.5 makes it partially transparent. Values for red, green, and blue also range between 0 and 1, where 0 indicates none of the color and 1 indicates all of the color. Imagine the values indicate a dial for each colored lamp. When the value is set to 0, the lamp is off and when the value is set to 1, it is at full strength—any values in between give partial brightness. For example, a color value of (1,0,0,1) will give the color red. A color value of (1,1,0,1) will give the color yellow. The easy way to look up values for a color is to use the color picker included with most software, including MS Word and Adobe Photoshop. The color picker from Adobe Photoshop is shown in Figure 1.15.

Also included with most color pickers is the ability to set the color using different color models. The one shown in Figure 1.15 includes a Hue, Saturation, and Brightness model, as well as a CMYK model. For more information on these, check out http://en.wikipedia.org/wiki/Color_model.

Note

An alternate way to set the value of a color is with values between 0 and 255 instead of between 0 and 1. It depends on the software you are using. In programming, values are usually between 0 and 1, but more commonly between 0 and 255 in color pickers.

1.4 How Game Engines Work

A game engine takes all the hard work out of creating a game. In the not so distant past, game developers had to write each game from scratch or modify older similar ones. Eventually game editor programs started to surface that allowed developers to create games without having to write a lot of the underlying code.

The game engine takes care of things such as physics, sound, graphics processing, and user input, allowing game developers to get on with the creation of high-level game mechanics. For example, in Unity, physical properties can be added to a ball with the click of a button to make it react to gravity and bounce off hard surfaces. Driving these behaviors, embedded in the engine, are millions of lines of complex code containing many mathematical functions related to real-world physics. The game developer can spend more time designing what the ball looks like and even selecting the type of material it is made from without having a background in Newtonian physics.

1.4.1 A Generic Game Engine

To understand how a game engine works, we will first look at a simple illustration of all its components. A conceptualization is shown in Figure 1.16.

The game engine is responsible for the running of a variety of components that manage all the game resources and behaviors. The *Physics Manager* handles how game objects interact with each other and the environments

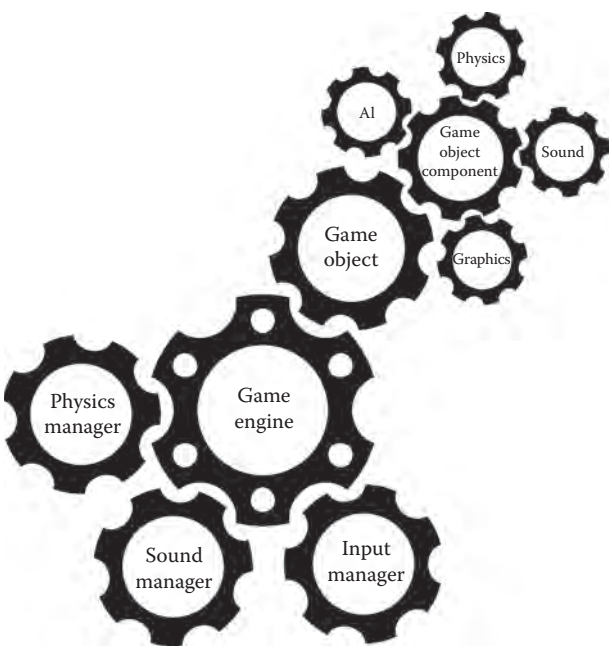


FIG 1.16 Parts of a generic game engine.

by simulating real-world physics. The *Input Manager* looks after interactions between the player and the game. It manages the drawing of graphical user interfaces and the handling of mouse clicks and the like. The *Sound Manager* is responsible for initializing and controlling how sound is delivered from the game to the player. If 3D sound is called for, it will ensure that the right sound at the right volume is sent to the correct computer speaker.

In addition to these managers are game objects. Game objects represent all the assets placed in a game environment. These include the terrain, sky, trees, weapons, rocks, nonplayer characters, rain, explosions, and so on. Because game objects represent a very diverse set of elements, they can also be customized through the addition of components that may include elements of artificial intelligence (AI), sound, graphics, and physics. The AI component determines how a game object will behave. For example, a rock in a scene would not have an AI component, but an enemy computer-controlled character would have AI to control how it attacks and pursues the player. A sound component gives a game object a sound. For example, an explosion would have a sound component whereas a tree may not. The physics component allows a game object to act within the physics system of the game. For example, physics added to a rock would see it roll down a hill or bounce and break apart when it falls. The graphics component dictates how the game object is drawn. This is the way in which it is presented to players on the screen. Some game objects will be visible and some will not. For example, a tree in a scene is a visible game object, whereas an autosave checkpoint, which may be a location in a game level, is not.

1.4.2 The Main Loop

All games run in the same way, as illustrated in Figure 1.17. There is an initialization stage in which computer memory is allocated, saved information is retrieved, and graphics and peripheral devices are checked. This is followed by the *main game loop* or *main loop*. The main loop runs continuously over and over again until the player decides to quit the game. While in the main loop, the game executes a cycle of functions that processes user input messages; checks through all game objects and updates their state, including their position; updates the environment with respect to game object positions, user interaction, and the physics system; and finally renders the new scene to the screen.

Essentially each loop renders one frame of graphics on the screen. The faster the loop executes, the smoother the animation of the game appears. The more processing that needs to be performed during the main loop, the slower it will execute. As the number of game objects increases, the amount of work the main loop has to do also increases and therefore slows down the time between frames being rendered on the screen. This time is called *frames per second* (FPS).

Game developers strive for very high FPS, and for today's computers and consoles, FPS can extend beyond 600. In some circumstances, however,

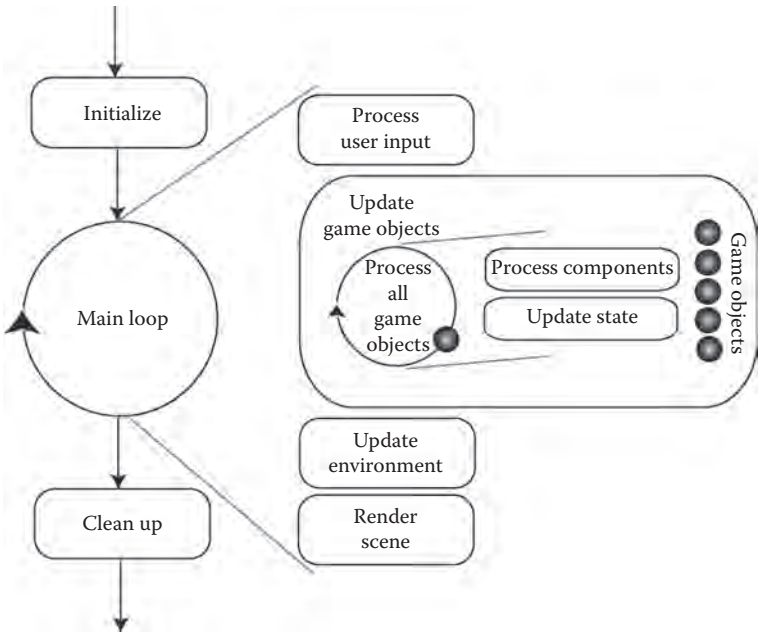


FIG 1.17 How a game runs.

such as on mobile devices with less processing power, FPS can become very low with only several game objects, and the animation will flicker, and user controls will be nonresponsive. Having said this, beginner game developers need to be aware of this issue, as even on a very powerful computer, adding a lot of highly detailed game objects can soon bring the FPS to a grinding halt. Anything below 25 FPS is considered unacceptable, and as it approaches 15 FPS the animation starts to flicker.

Unity Specifics

Game Objects

Game objects are the fundamental building blocks for Unity games. It is through the addition, modification, and interaction of game objects that you will create your own game. After adding a game object in Unity (which you will do in the next section) a variety of components can be added to give the game object different functionalities. In all, there are seven components categories. These will be thoroughly explored throughout this book. In short, they are *Mesh*, *Particles*, *Physics*, *Audio*, *Rendering*, *Miscellaneous*, and *Scripts* as shown in Figure 1.18. A game object can have all, none, or any combination of these components added. The game object exemplified in Figure 1.18 has at least one of each of these component types added.

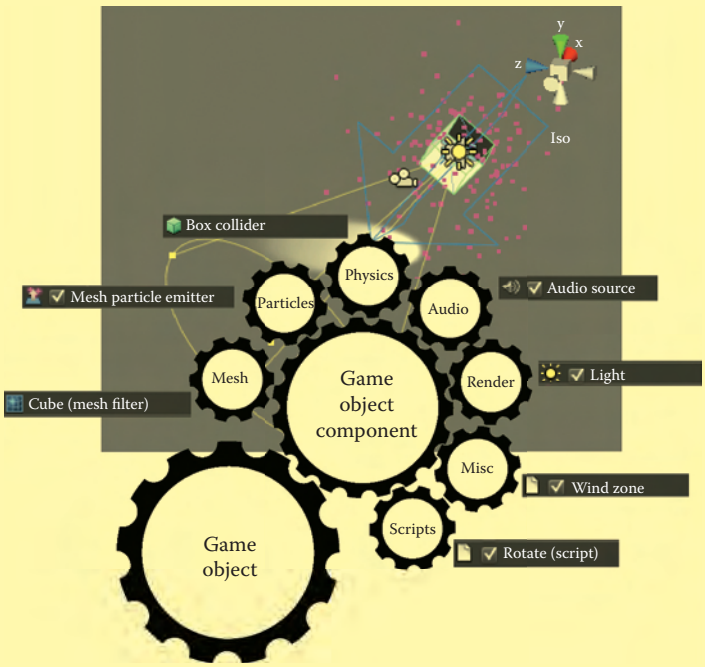


FIG 1.18 Components that can be added to a game object in Unity.

A Mesh component handles the drawing of an object. Without a Mesh component, the game object is not visible. A Particles component allows for a game object to have a particle system added. For example, if the game object were a jet fighter, a particle system could be added to give the effect of after burners. A Physics component gives the game object real world physical properties so it can be collided with and affected by gravity and other physics effects. An Audio component adds sound or sound effects to a game object. For example, if the game object were a car, the noise of a car engine could be added. A Rendering component adds special effects to a game object such as emitting light. Miscellaneous components include a variety of affects for the game object that do not fit within other categories. In Figure 1.18 the Wind Zone component is shown as a type of miscellaneous component. In brief, this causes the game object to become a source of wind for interaction within the physics

system. Last, Scripts are components that contain programming code to alter the behavior of a game object. Scripts can be used for a large variety of purposes and are fundamental to developing game mechanics and tying an entire game together.

In Unity, scripts added to game objects can be written in C#.

In addition to this traditional method of game engine use, Unity has also introduced (in 2019) methods to support multithreading and multicore processing. It is called the Entity-Component System (ECS). Such methods of programming depart heavily from the traditional object orientated or procedural methods that better facilitate beginners starting out and therefore will not be explored in this book. However, if you are interested in what's possible then check out the author's YouTube tutorials for this system at:

- https://youtu.be/Awf_Y4hBhBM
- https://youtu.be/Vg-V5G2JJNY

Unity Hands On

Getting to know the Unity3D development environment

Step 1: To begin, download Unity by visiting http://unity3D.com/ and clicking on *Download*. Unity has a free version that lacks some functionality, but never expires. The free version is still quite powerful and certainly enough for the first time game developer. Once you have downloaded the software, follow the installation instructions to get Unity up and running.

Step 2: Running Unity for the first time reveals the multi-windowed editing environment shown in Figure 1.19. The tabs and windows can be dragged around to suit your own preferences.

On the Web

Navigating the Unity Editor Interface

Visit the website for a short video demonstrating some best practices for finding your way around in the Unity Editor.

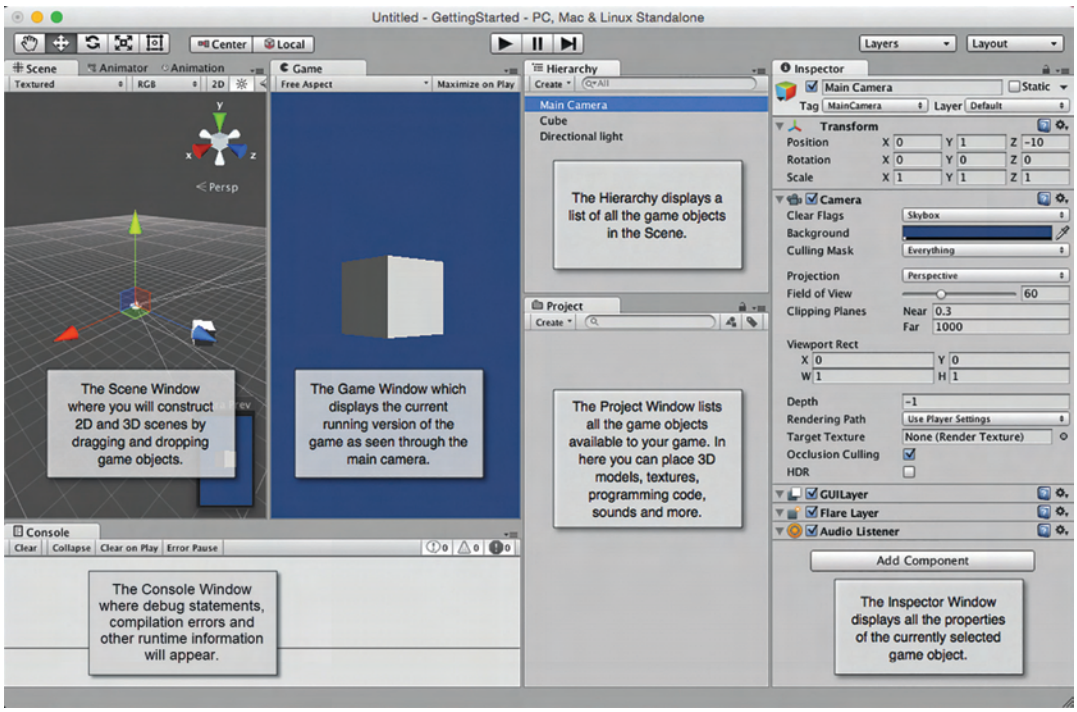


FIG 1.19 The Unity editing environment.

Step 3: After starting Unity, create a new project by selecting File > New Project. Note that the project name and directory used to save the project are one and the same; by default, this is "New Unity Project." The dialog box for creating a new project will allow you to choose 3D or 2D. For now select 3D.

Step 4: To create a simple scene, select GameObject > 3D Object > Cube from the main menu. All objects added to the game scene are called game objects in Unity. A cube will appear in the Hierarchy, Scene, Game, and Inspector windows.

Note

From this point in the text, these windows will be referenced just by their capitalized names.

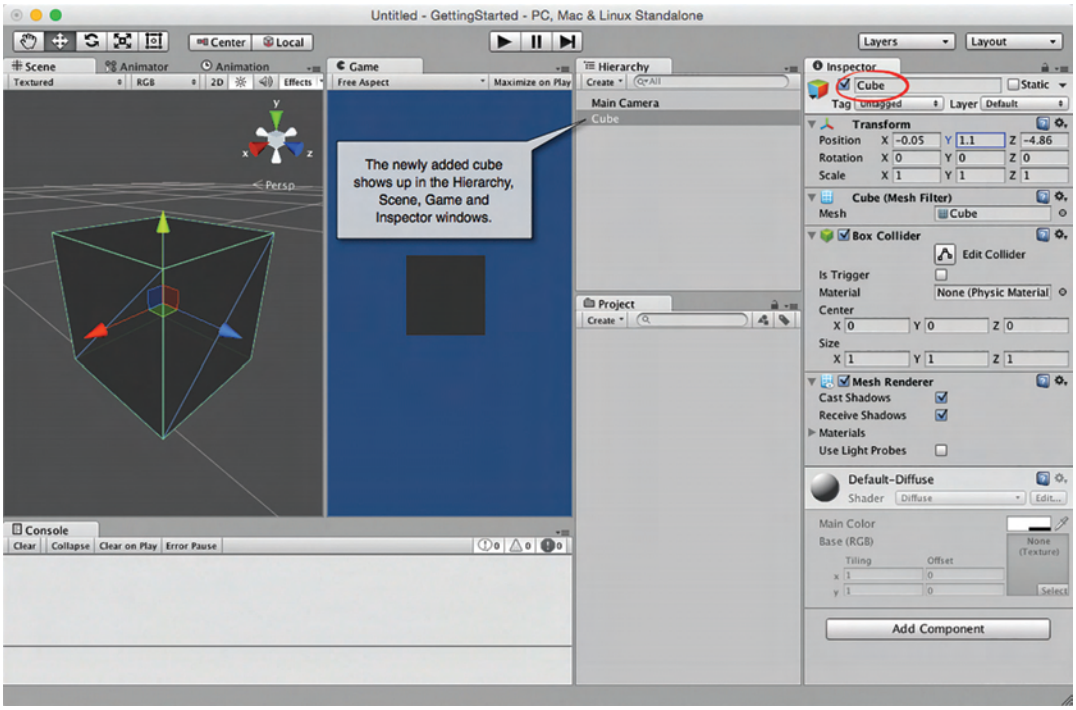


FIG 1.20 A single cube in a scene.

Step 5: If the cube appears very small, place your mouse in the Scene and use the scroll wheel to zoom in. You can also focus the scene on the cube by double-clicking it in the Hierarchy or selecting it in the Scene and pressing the F key. Note that your viewing position and angle in the Scene do not affect the look of the final game or change the attributes of any game objects. This initial Scene is shown in Figure 1.20. The Inspector shows all the properties of the cube. This includes its position, rotation, scale, the 3D mesh representing it, and a physics collider. We will look at these properties in more detail later.

Step 6: At this time, press the play button. As you have not added any functionality at this stage when running, all the game will do is display a static cube.

Note

Unity allows you to edit your game as it is running. This is great if you want to test out an idea to see how the game will react. Be careful though, because any changes you make in the editor, while play is on, will revert back to their previous value when you press stop. This can be very annoying if you've made large changes not realizing you are in play mode, as they will be wiped away as soon as you press stop. The only exceptions to this are script files, because they are edited and saved externally to the editor. Changes you make in script files are independent of the play button.

Step 7: Although lighting is a subject usually delayed for more advanced topics in game development, the author always likes to add a light to scenes to give them more depth and bring them alive. In the Scene, the cube is already shaded, as this is the default method of drawing. However, in the Game, the cube is a lifeless, flat, gray square. To add a light, select GameObject > Light > Directional Light from the main menu. A light displaying as a little sun symbol will appear in the Scene and the cube in the Game will become brighter.

Step 8: Now, because we are looking at the cube front on in the Game, it still appears as a little square. Therefore, we need to transform it for viewing. A transformation modifies the properties of position, rotation, and scale of a game object. The specifics of transformation are discussed later, but for now you can transform the cube quickly using the W key for changing position, the E key for rotating the objects, and the R key for scaling it. Before pressing any of these keys, ensure that the cube is selected in the Hierarchy window. When it is selected, it will have a green and blue wireframe displayed on it.

Step 9: In W (position) mode, the cube will be overlaid with red, green, and blue arrows. These represent the *x*, *y*, and *z* axes of the object. Clicking and dragging from any of the arrowheads will move the object along that axis. To move the object freely, click and drag from the central yellow box.

Step 10: In E (rotate) mode, the cube will have red, green, and blue circles drawn around it. Click and drag any of these circles to rotate the cube in the associated directions.

Step 11: In R (scale) mode, the red, green, and blue axes will include small cubes on the ends. Clicking and dragging any of these will change the scale of the object in the respective direction. You may also click and drag the central small cube to resize the object in all directions uniformly. Note that while you are moving, rotating, and

scaling the cube, its appearance changes in the Game window. You will also notice that values in the Transform part of the Inspector change too. Move and scale the cube so that you can see it clearly in the Game window.

Step 12: The color of a GameObject comes from an associated material. To create a material, click on Create in the Project window and select Material. New material will appear in the Project window and, when selected, its properties in the Inspector are as shown in Figure 1.21.

Step 13: To change the color of this material, click on the white box next to Main Color in the Inspector. Select the color you want from the Color Popup and then close it. The large sphere view of the material in the Inspector will change to your chosen color. To change the name of the material, click on New Material once in the Project window. Wait a moment and then click on it again slowly. This will place you in editing mode for the material name. Type in a new name and hit the Enter key.

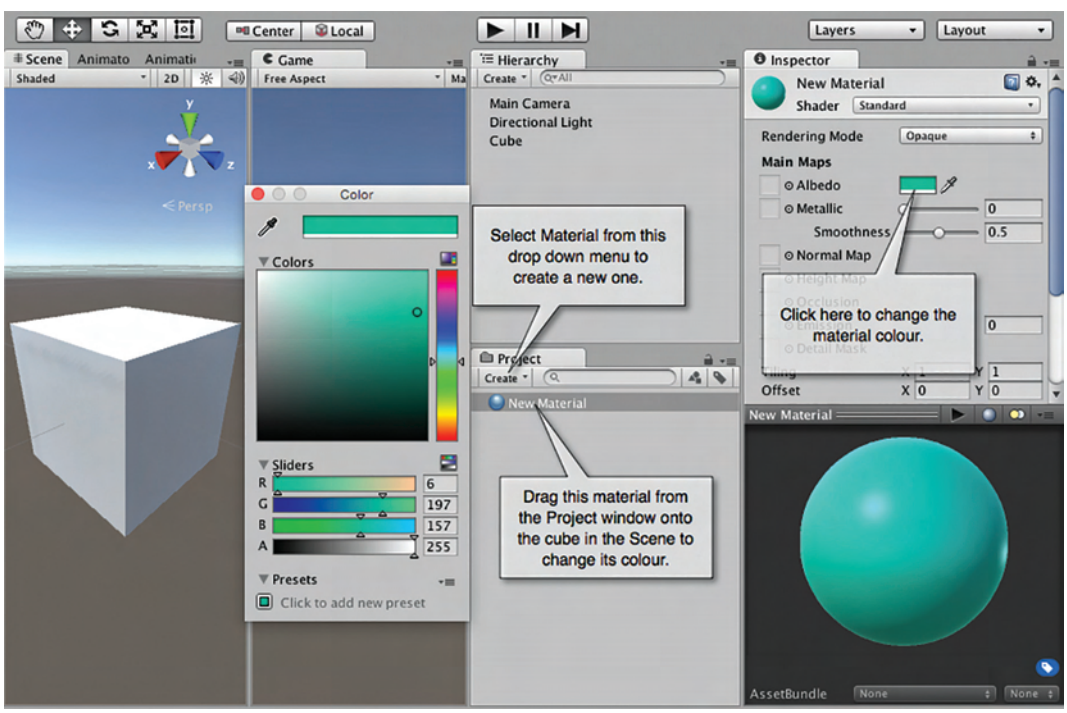


FIG 1.21 Creating a new material.

Step 14: To add material to the cube, drag and drop your material from the Project window onto the cube in the Scene. Alternatively, you can drag and drop the material from the Project window and drop it onto the cube listed in the Hierarchy window. Both will achieve the same effect. The cube will be colored.
Step 15: In the Project window, select Create and then C# Script. The C# script created will be called *NewbehaviorScript*. Change this, by slowly clicking on it, as you did with the material, to *spin*.

> **Note**
> ***Naming C# Files***
> When you create a new C# Script, you only give it a name. Do not add *.cs* on the end. Unity will do this for you automatically. However, in the Project, *spin.cs* will only appear in the list as *spin*, without the *.cs* on the end.

Step 16: Double-click on it and a code/text editor will open for entering code. In the code editor type:

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

public class spin : MonoBehaviour {

    // Use this for initialization
    void Start () {

    }

    // Update is called once per frame
    void Update () {
        transform.Rotate(Vector3.up * 10);
    }
}
```

The code must be EXACTLY as it appears here. Ensure that you have the correct spelling and capitalization; otherwise, it may not work. For large spaces, for example, before the word *transform*, insert a tab. When you are done, save your code in the text editor.

> **Note**
> The code in Step 15 contains the Unity function *Update()*. Whatever code you place inside *Update()* will be run once each main loop. Therefore the code in Step 15 will run over and over again for the entire time the play button is down.

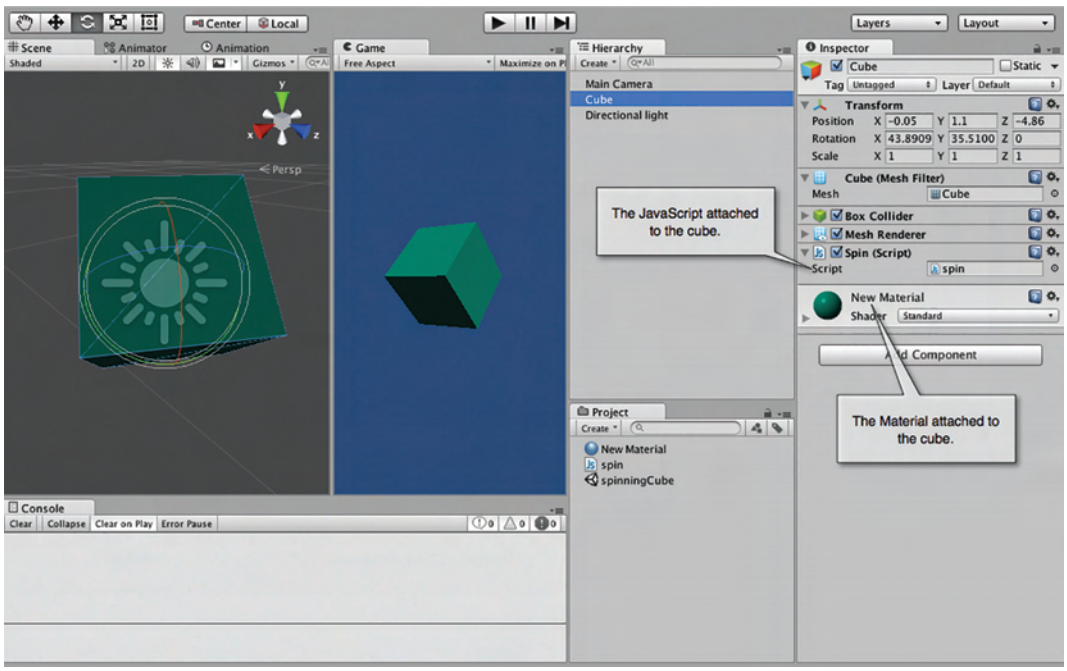


FIG 1.22 The cube with the spin script attached.

Step 17: Return to Unity and drag and drop the spin code from the Project window onto the cube in the Hierarchy window. You will notice that if you select the cube, the spin script will appear as a component added to the cube in the Inspector, as shown in Figure 1.22.

Step 18: Press the play button and watch as the cube spins.

Step 19: To save the application, select File > Save Scene from the main menu. In the dialog that pops up, give the scene a name such as *spinningCube*. Next, select File > Save Project from the main menu.

Each project you create can have multiple scenes. Scenes are saved inside projects. For example, an entire game application would be a single project. However, inside the game there would be multiple scenes, such as Main Menu, Help Screen, Map View, and 3D View. Single scenes can also be imported from one project to another.

1.5 A Scripting Primer

Hard-core programmers do not strictly consider scripting as programming, as it is a much simpler way of writing a program. Scripting languages have all the properties of programming languages; however, they tend to be less verbose and require less code to get the job done. For example, the

JavaScript and Java shown in Listings 1.4 and 1.5, respectively, demonstrate how the scripting language JavaScript requires much less code to achieve the same outcome as Java. The major difference between programming and scripting is that programming languages are compiled (built into a program by the computer) and then run afterward, and a scripting language is interpreted by another program as it runs.

When a program is compiled, it is turned into machine code the computer can understand. Compilation checks through the code for errors and then builds it into a program. Do not worry if you get lots of errors when you start programming. It is a normal part of the learning process. Some of the error messages will also seem quite vague and ambiguous for what they are trying to tell you. However, a quick search on Google will often reveal their true meaning.

C# is used in this book as the primary means of programming. Its syntax and constructs are closely related to C++, C, and Java; therefore, it is ideal to learn as a beginning language. It also provides a very powerful and yet simple way to develop game mechanics in Unity and many other game-editing environments.

Several fundamental constructs in programming are required to fully understand a programming language. These are variables, operations, arrays, conditions, loops, functions, and objects. However, the most fundamental and important concept that underlies everything is logic.

1.5.1 Logic

At the heart of all computers are the electronic switches on the circuit boards that cause them to operate. The fact that electricity has two states, on or off, is the underlying foundation on which programs are built. In simplistic terms, switches (otherwise known as relays) are openings that can either be opened or be closed, representing on and off, respectively. Imagine a basic circuit illustrated as a battery, electric cable, light switch, and light bulb, as shown in Figure 1.23.

When the switch is open, the circuit is broken, and the electricity is off. When the switch is closed, the circuit is complete, and the electricity is on. This is exactly how computer circuitry works. Slightly more complex switches called

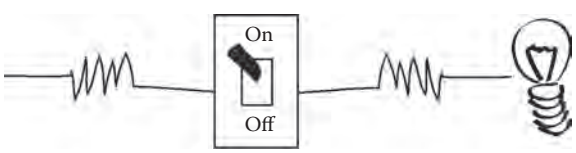


FIG 1.23 A very basic electric circuit.

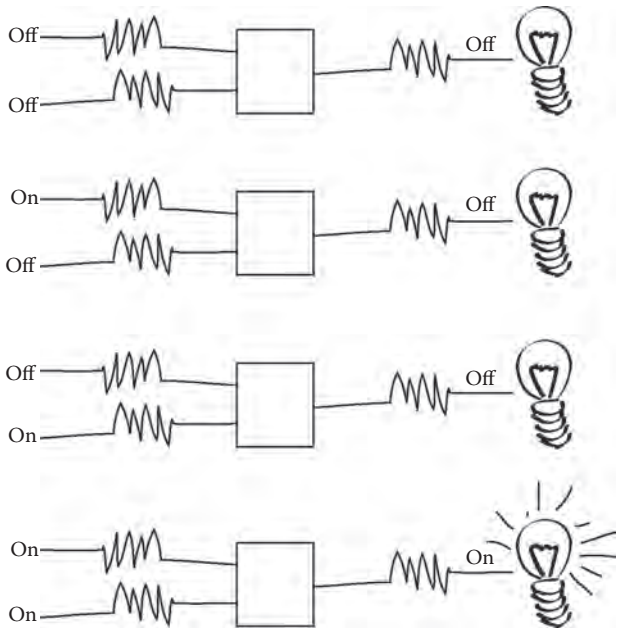


FIG 1.24 A conceptualization of an AND logic gate.

logic gates allow for the circuit to be opened or closed, depending on two incoming electrical currents instead of the one from the previous example.

In all, there are seven logic gate configurations that take two currents as input and a single current as output. These gates are called AND, OR, NOT, NAND, NOR, XOR, and XNOR. The AND gate takes two input currents; if both currents are on, the output current is on. If only one or none of the input currents is on, the output is off. This is illustrated in Figure 1.24.

In computer science, mathematics, physics, and many more disciplines the operation of combining two currents or signals into one is called *Boolean algebra (named after nineteenth-century mathematician George Boole)* and logic gate names (i.e., AND, OR) are called *Boolean functions*, although the *on* signal in computer science is referred to as TRUE, or the value 1, and *off* is FALSE, or 0. Using this terminology, all possible functions of the AND gate can be represented in *truth tables*, as shown in Table 1.1. It should be noted from Table 1.1 that truth tables can be written in a number of formats using 1s and 0s or TRUEs and FALSEs.

Each Boolean function has its own unique truth table. They work in the same way as the AND function does. They have two input values of TRUE and/or FALSE and one output value of TRUE or FALSE.

TABLE 1.1 Truth Table for the Boolean Function AND

Format 1			Format 2		
Input		***Output***	***Input***		***Output***
A	**B**	**A AND B**	**A**	**B**	**A AND B**
1	1	1	TRUE	TRUE	TRUE
1	0	0	TRUE	FALSE	FALSE
0	1	0	FALSE	TRUE	FALSE
0	0	0	FALSE	FALSE	FALSE

If you are just learning about programming for the first time right now, these concepts may seem a bit abstract and disconnected from real programming. However, as you learn more about programming, you will realize how fundamental knowing these truth tables is to a holistic understanding. It will become much clearer as you begin to write your own programs.

Quick Reference

Boolean Algebra

Function	Boolean Algebra Syntax	C#	Truth Table		
AND	A • B	A && B	**Input**		**Output**
			A	**B**	**A AND B**
			0	0	0
			0	1	0
			1	0	0
			1	1	1
OR	A + B	A || B	**Input**		**Output**
			A	**B**	**A OR B**
			0	0	0
			0	1	1
			1	0	1
			1	1	1
NOT	Ā	!A	**Input**		**Output**
			A		**NOT**
			0		1
			1		0

Quick Reference

Boolean Algebra (Continued)

Function	Boolean Algebra Syntax	C#	Truth Table		
NAND	$\overline{A \cdot B}$	`!(A && B)`	**Input**		**Output**
			A	**B**	**A NAND B**
			0	0	1
			0	1	1
			1	0	1
			1	1	0
NOR	$\overline{A+B}$	`!(A || B)`	**Input**		**Output**
			A	**B**	**A NOR B**
			0	0	1
			0	1	0
			1	0	0
			1	1	0
XOR	$A \oplus B$	`(A && !B) || (!A && B)`	**Input**		**Output**
			A	**B**	**A XOR B**
			0	0	0
			0	1	1
			1	0	1
			1	1	0
XNOR	$A \odot B$	`(!A && !B) || (A && B)`	**Input**	**Output**	**Output**
			A	**B**	**A XNOR B**
			0	0	1
			0	1	0
			1	0	0
			1	1	1

On the Website

Interactive Boolean Algebra Logic Gates

Navigate to the book's website for an interactive Unity version of Boolean Algebra.

1.5.2 Comments

Comments are not actually programming code. They are, however, inserted lines of freeform text totally ignored by the compiler. They allow you to insert explanations about your code or little reminders of what your code does.