

USERS

INCLUYE
VERSIÓN DIGITAL
GRATIS

UNITY

DISEÑO Y PROGRAMACIÓN DE VIDEOJUEGOS

CARACTERÍSTICAS DE LA APLICACIÓN

CREACIÓN DE ESCENARIOS
Y SISTEMAS DE ILUMINACIÓN

CONTROL DE OBJETOS
E INTERFAZ DE USUARIO

ADICIÓN DE MATERIALES
Y MANEJO DEL COLOR

MODELADO Y ANIMACIÓN EN BLENDER

IMPORTACIÓN DE OBJETOS ANIMADOS

por NICOLÁS ARRIJOJA LANDA COSIO



APRENDA A DESARROLLAR INCREÍBLES JUEGOS 3D

RU
RedUSERS

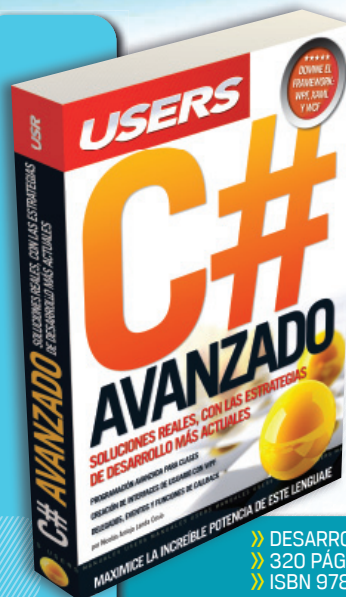
www.FreeLibros.me

CONÉCTESE CON LOS MEJORES LIBROS DE COMPUTACIÓN



ENTIENDA
EL CAMBIO,
APROVECHE SU
POTENCIAL

» DESARROLLO
» 320 PÁGINAS
» ISBN 978-987-1773-79-4



SOLUCIONES
REALES CON LAS
ESTRATEGIAS DE
DESARROLLO MÁS
ACTUALES

» DESARROLLO
» 320 PÁGINAS
» ISBN 978-987-1773-96-1



DOMINE EL
LENGUAJE LÍDER
EN APLICACIONES
CLIENTE-SERVIDOR

» DESARROLLO / INTERNET
» 320 PÁGINAS
» ISBN 978-987-1773-97-8



DISEÑO DE SITIOS
ATRACTIVOS
CON ADOBE
CREATIVE SUITE

» DESARROLLO / DISEÑO
» 320 PÁGINAS
» ISBN 978-987-1773-99-2

LLEGAMOS A TODO EL MUNDO VÍA
MÁS INFORMACIÓN / CONTÁCTENOS

usershop.redusers.com ☎ +54 (011) 4110-8700 ✉ usershop@redusers.com

* SÓLO VÁLIDO EN LA REPÚBLICA ARGENTINA // ** VÁLIDO EN TODO EL MUNDO EXCEPTO ARGENTINA

www.FreeLibros.me



UNITY

DISEÑO Y PROGRAMACIÓN DE VIDEOJUEGOS

por Nicolás Arriola Landa Cosío

Red**USERS**



TÍTULO: UNITY
AUTOR: Nicolás Arrijoja Landa Cosio
COLECCIÓN: Manuales USERS
FORMATO: 17 x 24 cm
PÁGINAS: 320

Copyright © MMXIII. Es una publicación de Fox Andina en coedición con DALAGA S.A. Hecho el depósito que marca la ley 11723. Todos los derechos reservados. Esta publicación no puede ser reproducida ni en todo ni en parte, por ningún medio actual o futuro sin el permiso previo y por escrito de Fox Andina S.A. Su infracción está penada por las leyes 11723 y 25446. La editorial no asume responsabilidad alguna por cualquier consecuencia derivada de la fabricación, funcionamiento y/o utilización de los servicios y productos que se describen y/o analizan. Todas las marcas mencionadas en este libro son propiedad exclusiva de sus respectivos dueños. Impreso en Argentina. Libro de edición argentina. Primera impresión realizada en Sevagraf, Costa Rica 5226, Grand Bourg, Malvinas Argentinas, Pcia. de Buenos Aires en I, MMXIII.

ISBN 978-987-1857-81-4

Arrijoja Landa Cosio, Nicolás
Unity. - 1a ed. - Buenos Aires : Fox Andina, 2013.
320 p. ; 24x17 cm. - (Manual users; 243)

ISBN 978-987-1857-81-4

1. Informática. I. Título

CDD 005.3



VISITE NUESTRA WEB

EN NUESTRO SITIO PUEDE OBTENER, DE FORMA GRATUITA, UN CAPÍTULO DE CADA UNO DE LOS LIBROS EN VERSIÓN PDF Y PREVIEW DIGITAL. ADEMÁS, PODRÁ ACCEDER AL SUMARIO COMPLETO, LIBRO DE UN VISTAZO, IMÁGENES AMPLIADAS DE TAPA Y CONTRATAPA Y MATERIAL ADICIONAL.

RedUSERS
COMUNIDAD DE TECNOLOGIA



redusers.com

Nuestros libros incluyen guías visuales, explicaciones paso a paso, recuadros complementarios, ejercicios, glosarios, atajos de teclado y todos los elementos necesarios para asegurar un aprendizaje exitoso y estar conectado con el mundo de la tecnología.



LLEGAMOS A TODO EL MUNDO VÍA DOCA* Y DHL**

*SOLO VÁLIDO EN LA REPÚBLICA ARGENTINA // **VÁLIDO EN TODO EL MUNDO EXCEPTO ARGENTINA

usershop.redusers.com usershop@redusers.com +54 (011) 4110-8700

Nicolás Arrioja Landa Cosío

Es catedrático de IUNIVERTECH y de la Universidad Madero, elegido el mejor profesor de los ciclos 2006-2007 y 2007-2008, se dedica a la investigación, consulta y capacitación en áreas relacionadas con la realidad virtual, la visualización científica y los videojuegos.



En 1997, desarrolló el primer videojuego de realidad virtual inmersiva en Latinoamérica, conocido como **VRaptor**. También desarrolló el primer lenguaje de programación para realidad virtual en Latinoamérica **CND-VR**, que es usado en el medio académico para enseñar de forma sencilla los conceptos de programación de gráficas 3D. Ha sido catedrático en algunas de las universidades más importantes de México durante más de diez años. En esas instituciones, enseñó desde las bases de programación en lenguaje C hasta inteligencia artificial. Ha otorgado más de 30 conferencias relacionadas con el desarrollo de la realidad virtual y los videojuegos en diversas universidades de México. Tiene una patente referida a interfaces cinemáticas para videojuegos. Diseñó el plan y el programa de estudios para la Maestría en Realidad Virtual y Video Juegos en IUNIVERTECH. Actualmente, realiza una investigación sobre Inteligencia Artificial no-Algorítmica y diseño de hardware. Es el creador de la IABA (inteligencia artificial basada en abstracciones). Tiene disposición a realizar investigación y dar conferencia en estas áreas en cualquier parte del mundo.

Dedicatoria

Este libro se lo dedico a la memoria de SGM Thomas Carter y a todos aquellos que ya no están con nosotros.

Agradecimientos

A todos mis amigos que me han apoyado durante la creación de este libro, a Nelson Prada y toda la gente de la editorial.

Prólogo



Desde el surgimiento de los primeros videojuegos modernos en la década de 1960, el creciente interés hacia ellos y las diversas aplicaciones que se les ha dado, han hecho necesaria la creación de herramientas que permitan desarrollarlos de manera más rápida y contando con un alto grado de fidelidad que los acerque al mundo real.

Actualmente existen en el mercado un sinnúmero de aplicaciones y frameworks que nos permiten trabajar de forma muy visual y que facilitan el trabajo para crear videojuegos. Este libro se enfocará en Unity, una herramienta que permite desarrollar videojuegos en 3D en diversas plataformas y darles un acabado profesional.

Unas de las mayores ventajas de Unity es que cuenta con versiones para Windows y Mac, además de que puede obtenerse una licencia gratuita para su uso.

El editor visual de Unity permite importar modelos 3D, texturas, sonidos, etc. y después trabajar con ellos. En caso de no contar con estos recursos, desde la misma aplicación se puede acceder a una tienda donde podemos encontrar una gran cantidad.

Unity es una aplicación poderosa para crear videojuegos que no requiere que seamos programadores expertos o contar con conocimientos avanzados de Física y Matemáticas para poder usarlo.

En este libro, hay instrucciones y ejercicios que “nos llevan de la mano” para crear nuestras propias escenas, colocar scripts a nuestros objetos, mover, rotar y escalar objetos, usar dispositivos de entrada, cámaras e iluminación, crear y mover terrenos, hacer simulaciones físicas a nuestro mundo virtual, incorporar efectos especiales que le den realismo a los juegos, etc. Cuenta además con ejemplos de código que pueden servirnos para realizar cosas más avanzadas y que posteriormente se incorporen a nuestros proyectos.

Te garantizo que con este libro realmente aprenderás a usar una herramienta poderosa con la cual podrás crear tus propios videojuegos. Ahora todo depende de tu empeño y creatividad. Recuerda que en un videojuego, el cielo no siempre es el límite.

María Eugenia Pérez Duarte

*Coordinadora de Ingenierías en Tecnologías y Software
Universidad Madero, México*

El libro de un vistazo

Este libro nos enseña Unity, un poderoso sistema para desarrollar y publicar nuestros propios videojuegos en 3D. Lleva de la mano al lector en los temas más importantes: scripts, iluminación, audio, efectos, física, terrenos, etc. Además, enseña a importar modelos en 3D desde Blender y propone la creación de un proyecto integrador.

*01



CONOCER UNITY

En este capítulo aprendemos como utilizar el editor de Unity, los diferentes tipos de paneles y menús que lo componen y la forma de movernos por dentro de la escena. Además vemos la adición y manipulación básica de los objetos, la creación de una escena y los conceptos más importantes.

*04



MANEJO DE CÁMARAS

La cámara nos permite ver la escena y trabajamos con sus propiedades. Creamos un script para poder manipularla y controlarla. Utilizamos los scripts que Unity provee para hacer uso de una cámara en primera persona y otra en tercera persona.

*02



TRABAJAR CON LOS OBJETOS

Nos concentramos en el uso de los objetos y la creación de scripts que manipulen sus propiedades. Cubrimos la adición de materiales y el manejo del color, la codificación de variables y la creación de las propiedades. Trabajamos con los dos tipos de movimiento.

*05



CREACIÓN DE TERRENOS

Unity provee un editor de terrenos que nos permite crear escenas en exteriores. Es un editor que tiene muchas características y con el que podemos realizar cualquier tipo de exteriores, incluyendo texturas con detalles, objetos y plantas. Está es una de las características más interesantes de Unity.

*03



CONTROL DE ENTRADAS POR EL USUARIO

Aprendemos como funciona el sistema de entradas de Unity para poder leer al teclado, mouse y controladores. La creación de un script que nos permite controlar un objeto desde el teclado y el mouse. El uso de la clase Input y sus funciones más importantes.

*06



LAS LUCES

La iluminación es fundamental en cualquier videojuego, aprendemos el uso del sistema de luces de Unity. Trabajamos con Point Light, Directional Light, Spotlight y Ambient Light. Creamos diversos scripts que nos permiten controlar las luces según lo necesite nuestro juego.

*07



FISICA Y CUERPOS RÍGIDOS

Trabajamos con el motor de física incorporado en Unity para poder llevar a cabo simulación de fenómenos físicos en nuestros juegos y trabajar con detección de colisiones en sus diversas formas. Creación de scripts para que nuestro objeto reaccione a la colisión.

*10



CÓDIGO Y HERRAMIENTAS ÚTILES

La incorporación de audio en nuestras aplicaciones y su control por medio scripts. Herramientas que podemos usar para la depuración. El código para esconder y mostrar objetos. Calcular la distancia entre objetos por medio de un script. El uso de los prefabs en el juego.

*08



EFFECTOS ESPECIALES

La creación de efectos especiales como fuego, explosiones, lluvia, etc. Hacemos uso del sistema de partículas incorporado en Unity y trabajamos con las diversas propiedades que nos permiten controlar de forma precisa como deseamos que se muestre el efecto.

*11



INTERFAZ DE USUARIO

La creación de una interfaz gráfica de usuario que podemos usar en el videojuego, para desplegar información al usuario o como mecanismo para obtener información del jugador. Vemos los elementos principales y como crear scripts que lean o muestren la información.

*09



BLENDER Y ANIMACIONES

Utilizamos Blender para modelar un personaje para videojuego. El uso de las herramientas más importantes, crear materiales y asignar texturas en Blender. Animación del personaje usando keyframes. La exportación del personaje a Unity y un script para controlarlo.

*12



PROYECTO INTEGRADOR Y DISTRIBUCIÓN

Creación de un videojuego completo que hace uso de lo aprendido anteriormente. Codificación de scripts de control para los objetos, interfaz de usuario para mostrar puntajes. Al final se construye la aplicación y queda lista para distribuirse en PCs o Internet.



INFORMACIÓN COMPLEMENTARIA



A lo largo de este manual podrá encontrar una serie de recuadros que le brindarán información complementaria: curiosidades, trucos, ideas y consejos sobre los temas tratados. Para que pueda distinguirlos en forma más sencilla, cada recuadro está identificado con diferentes iconos:



CURIOSIDADES
E IDEAS



ATENCIÓN



DATOS ÚTILES
Y NOVEDADES



SITIOS WEB

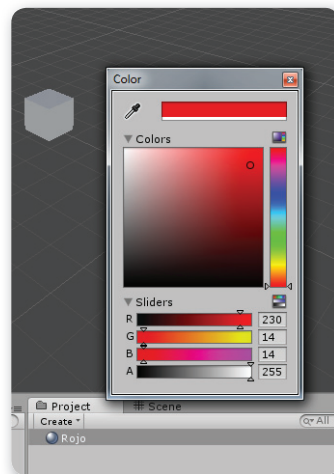
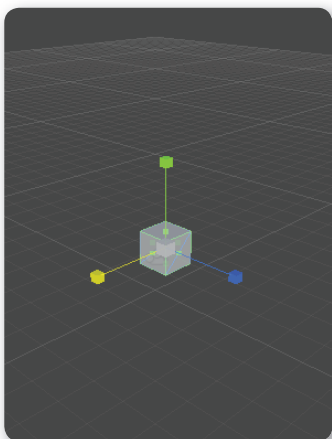
Contenido

Sobre el autor	4
Prólogo	5
El libro de un vistazo	6
Información complementaria.....	7
Introducción.....	12

*01

Conocer Unity

Características principales	14
El motor de juego.....	14
Instalación de Unity	16
Paneles principales.....	21
El panel Scene	21
El panel Game.....	26
El panel Hierarchy	32
El panel Project	33
El panel Inspector	34
Ejecutar la aplicación.....	34
Resumen	35
Actividades	36



*02

Trabajar con objetos

Programar la posición	38
Adicionar los objetos y materiales	39
Guardar la escena	42
Crear scripts.....	43
Declaración de variables	47
Crear propiedades	51
Mover los cuerpos.....	54
Movimiento por cuadro.....	54
Movimiento por tiempo	56
Escalamiento	58
La rotación.....	61
Múltiples scripts	62
Rotación por tiempo.....	62
Rotación alrededor de un punto.....	64
Proyecto de sistema solar	65
Adicionar texturas	66
Agregar el código	67
Resumen	69
Actividades	70

*03

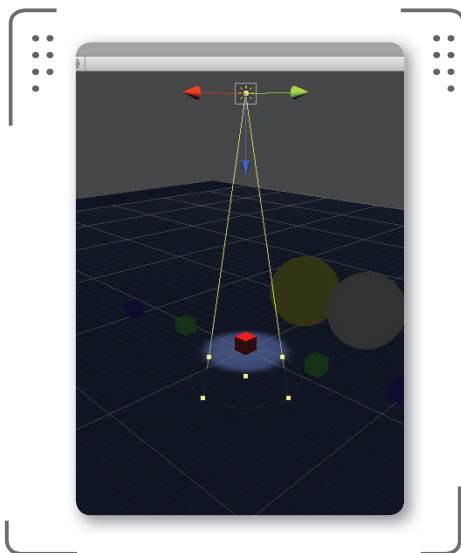
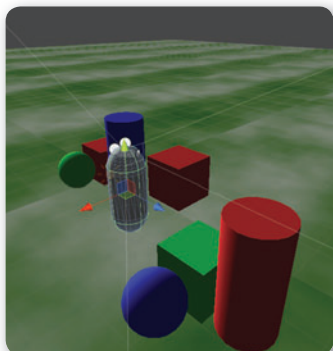
Control de entradas por el usuario

El sistema de entradas de Unity	72
Capturar la entrada del teclado	75
Capturar la entrada del mouse	86
Leer otros dispositivos	90
Resumen	95
Actividades	96

*04

Manejo de cámaras

Conocer la cámara	98
Crear una aplicación de prueba	99
Propiedades principales de la cámara	104
La cámara sigue a un objeto	115
Mover la cámara	119
Usar los scripts de Unity	121
La cámara en primera persona	123
La cámara en tercera persona	124
Resumen	125
Actividades	126



*05

Creación de terrenos

Los terrenos en Unity	128
Experimentar con los terrenos	128
Las herramientas de edición	131
Creación del terreno para un videojuego	139
Resumen	151
Actividades	152

*06

Las luces

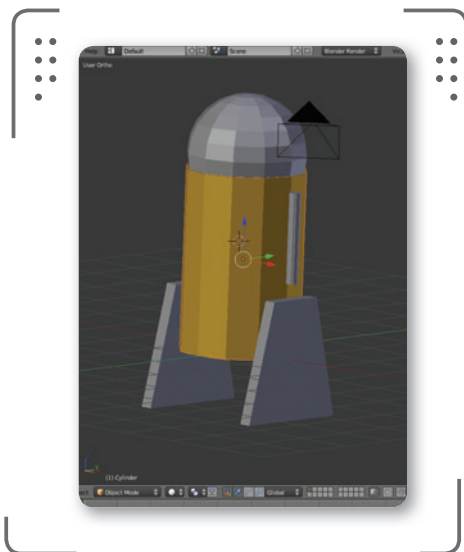
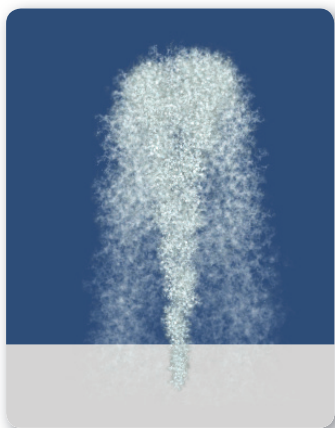
La iluminación	154
Crear la escena	154
Point Light	156
Directional Light	166
Spotlight	173
Ambient Light	180
Resumen	181
Actividades	182

07*Física y cuerpos rígidos**

El sistema de física	184
Motores de física	184
Entender los colliders.....	185
Entender el rigid body	188
Crear una aplicación de prueba	190
Usar los triggers.....	191
Usar las colisiones	193
Resumen	203
Actividades	204

08*Efectos especiales**

El sistema de partículas	206
Usos y características	206
Crear un sistema de partículas	207
Editar los diferentes tipos de datos.....	212
Más propiedades importantes.....	214
Crear un subsistema	219
El uso del editor	221



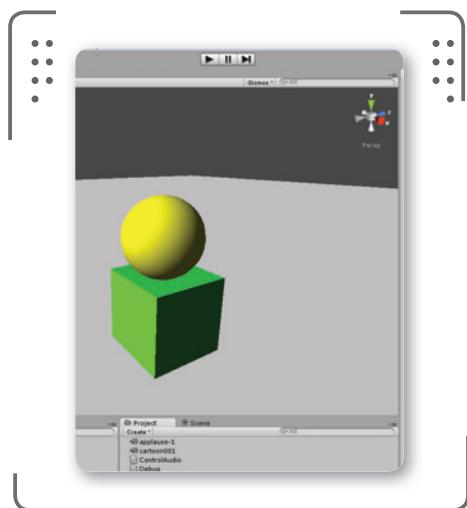
Efectos preconstruidos	222
Resumen	223
Actividades	224

09*Blender y animaciones**

Blender	226
Creación del robot	227
Materiales y texturas.....	233
Animación	238
Exportar a Unity	242
Resumen	247
Actividades	248

10*Código y herramientas útiles**

Elementos y scripts fundamentales	250
Sonido	250
Herramientas de depuración	256
Esconder objetos	259



Distancia entre los objetos.....	261
Manejo de prefabs.....	263
El uso de tag	265
Resumen	269
Actividades	270

*11

Interfaz de usuario

Elementos para crear la interfaz.....	272
Label.....	272
El uso de Box	275
El uso de Button.....	277
Pedido de texto al usuario	280
El uso del slider.....	284
El uso de toggle.....	287
Resumen	289
Actividades	290

*12

Proyecto integrador y distribución

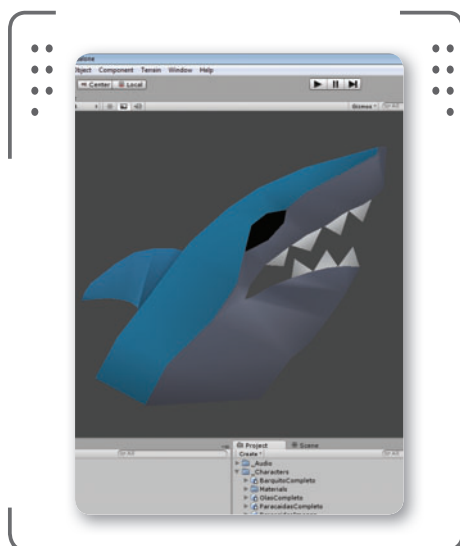
Creación y animación en Blender	292
--	------------

El barco.....	292
Las olas.....	297
El paracaídas.....	297
El pelícano	298
El tiburón	299
Programación en Unity	300
El objeto barco	300
El objeto olas	301
El objeto paracaídas	301
El objeto pelícano.....	304
El objeto tiburón	306
El objeto cámara.....	308
Creación de la escena.....	311
Construir la aplicación	314
Resumen	315
Actividades	316



Servicios al lector

Índice temático	317
Catálogo	319



» Introducción

Los videojuegos forman parte de nuestra cultura y ya tenemos generaciones que han crecido junto con ellos. Esto hace que cada día más personas estén interesadas en el desarrollo de este tipo de aplicaciones. En un inicio, el desarrollo de los juegos fue casi artesanal, complicándose más conforme la tecnología mejoraba. En un momento para crear un juego en 3D era necesario conocer DirectX, matemáticas, física y muchas cosas más que hacían difícil que una sola persona pudiera crear un buen motor de juego para sus aplicaciones.

Afortunadamente tenemos a Unity, un motor y editor de juego con características profesionales que nos permite crear nuestros propios juegos en 3D de manera sencilla. Unity se encarga de toda la parte complicada, y nosotros nos podemos concentrar en el desarrollo de la aplicación. A lo largo de la obra, se aprende cómo crear nuestros scripts para controlar la lógica de los objetos, hacemos uso del editor para crear escenas, iluminación e incluso terrenos. Aprendemos a usar el motor de física que provee Unity para que nuestro juego sea más realista y vemos la manera de incorporar efectos especiales creados por partículas. No hay que olvidar que el audio también es importante, por lo que aprenderemos a usarlo dentro de nuestras aplicaciones. Terminaremos con un proyecto integrador que nos muestre cómo podemos usar todo lo aprendido para la creación de un juego. La interacción con el usuario es necesaria por lo que también se cubre el tema de la interfaz de usuario, usando elementos gráficos, que aumentan el atractivo y la funcionalidad del juego.

El libro complementa a Unity con Blender, que es un software de modelaje y animación en 3D, el cual nos va a permitir modelar todos los objetos y personajes que nuestro juego necesite. Podemos incorporar materiales y texturas a estos objetos para posteriormente exportarlos a Unity y usarlos en nuestros juegos

Espero que este libro sea útil para todos aquellos que deseen aprender cómo crear sus propios videojuegos y que pronto se hagan realidad todas sus ideas.



Conocer Unity

Unity es una herramienta que nos permite crear videojuegos en 3D de increíble realismo, con gran facilidad. Aun teniendo una versión gratuita del software, la calidad de los juegos que podemos desarrollar es excelente, y el uso del programa es realmente muy sencillo. Estas características convierten a Unity en una excelente opción para los desarrolladores independientes y los estudios de videojuegos, cualquiera sea su presupuesto.

▼ Características principales 14

El motor de juego 14
Instalación de Unity..... 16

▼ Paneles principales 21

El panel Scene..... 21
El panel Game..... 26
El panel Hierarchy..... 32

El panel Project..... 33

El panel Inspector..... 34

▼ Ejecutar la aplicación 34

▼ Resumen..... 35

▼ Actividades..... 36





Características principales

Unity es una herramienta completa para el desarrollo de videojuegos. No solo es un motor de juegos de gran calidad y con numerosas características: también tiene un buen editor de mundos y trabaja con diferentes lenguajes de programación. Su uso es sencillo, y cualquier persona que tenga conocimientos básicos sobre programación podrá generar sus propios juegos en poco tiempo.

El motor de juego

El motor de juego tiene muchas características interesantes; incluso en su versión gratuita ofrece particularidades propias de otros más costosos. Muchas veces encontramos motores de juegos gratuitos, pero la calidad gráfica de los desarrollos luce como de hace diez años. Con Unity, por el contrario, nuestros videojuegos se ven actuales, y si usamos la versión paga, pueden tener efectos visuales propios de productos profesionales.

En el aspecto de renderizado el **pipeline** es moderno. En graficación, se conoce como pipeline al proceso secuencial por medio del cual solamente al finalizar un paso, podemos seguir con otro. Unity usa lo que conocemos como **deferred rendering pipeline**, que provee un gran desempeño y el uso de muchas luces dinámicas sin bajar demasiado el rendimiento; una buena iluminación ayuda a aumentar el realismo de nuestro mundo.

El motor incluye una colección de **shaders**, de modo que no perdamos tiempo implementando los básicos y más comunes. Los shaders son programas que se ejecutan en la tarjeta de video e indican la forma en que se dibuja un polígono. Desde luego, podemos adicionar nuestros propios shaders. También podemos colocar efectos en la



CONOCER LAS VERSIONES



Para conocer las diferencias entre la **versión gratuita** y la **Pro**, así como las licencias para las diferentes consolas, podemos visitar el sitio web de Unity, donde también encontraremos más información sobre el software y sus características: www.unity3d.com/unity/licenses.

parte del postproceso, con lo cual logramos muchos efectos ópticos, como aberraciones cromáticas, resplandores y profundidad de campo. Algunos de ellos se usan en juegos como la serie de **Gears of War**.

La licencia profesional incluye tecnologías que encontramos en motores que cuestan decenas de miles de dólares. Esto quiere decir que podemos crear videojuegos que pueden llegar a lucir tan bien como los de las consolas.

El sistema de iluminación es muy bueno, y el renderizador usa múltiples hilos para ser lo más eficiente posible. Se pueden emplear sombras suaves y duras en tiempo real. El manejo de las luces es sencillo, y para aumentar la calidad y el desempeño podemos hacer uso del **lightmapping**, ya que usa texturas en vez de cálculos de iluminación para iluminar la escena.

UNA BUENA
ILUMINACIÓN AYUDA
A AUMENTAR
EL REALISMO DE
NUESTRO MUNDO

El editor

El editor que provee Unity es muy intuitivo, y en él encontramos muchas herramientas de utilidad. Podemos aprender a usarlo rápidamente, y en muchos aspectos, veremos que es similar a cualquier otra aplicación con interfaz gráfica.

Lo primero que vemos es un buen editor de terrenos, con el que podemos crear las escenas en exteriores del juego. Se maneja como si estuviéramos esculpiendo: podemos subir y bajar el terreno, nivelarlo, etc. Una vez construida la geometría, podemos colocar las diferentes texturas y la iluminación. Los objetos tales como árboles, rocas, hierbas y otros elementos del ambiente se colocan con la misma facilidad que tiene el uso de una brocha en un programa de dibujo. También hay una herramienta especial que permite crear árboles e incluir zonas de viento.

Con el editor podemos generar los diferentes objetos, ubicarlos en el mundo, y agregar las características y los comportamientos que necesitamos; solo tenemos que colocar las propiedades y asignar los scripts. El mismo editor nos permite hacer ejecuciones de prueba para verificar el funcionamiento del juego. La edición del código de los scripts no se lleva a cabo en el editor, sino que se apoya en un programa de edición de código externo.

Otros elementos

El sistema de física es muy bueno y está basado en **PhysX** de **NVIDIA**. No es preciso ser grandes programadores o saber sobre física y simulación para usarlo. Simplemente, agregamos unas cuantas características a nuestro objeto y estamos listos para usarlo. Dentro de lo que se puede simular encontramos: **telas, cuerpos suaves, cuerpos rígidos, ragdolls**, etc. Incluso podemos encontrar un tutorial sobre la simulación de la física para autos.

El audio es un factor muy importante en la experiencia de juego. Por este motivo, Unity también nos brinda las herramientas necesarias para usarlo en nuestros desarrollos. Si bien no cuenta con un editor interno de audio, nos suministra las herramientas necesarias para tener un gran control del sonido dentro del juego.

En la programación de los scripts podemos hacer uso de tres lenguajes: **JavaScript, C# y Boo**. En este libro nos concentraremos en el manejo de **JavaScript**, pero todos los conceptos que aprendamos son válidos para **C# y Boo**. Es posible utilizar **.NET** vía **Mono**. Lo mejor de todo es que podemos tener scripts en diferentes lenguajes dentro del mismo proyecto, y no habrá conflictos. Podemos crear propiedades en nuestros scripts que se muestren y editen visualmente en el editor, lo que facilitará el desarrollo, el testeo y la experimentación.

Instalación de Unity

La instalación de Unity es muy sencilla, porque no requiere de configuraciones complicadas y prácticamente es igual que instalar cualquier otra aplicación. Unity puede usarse tanto en **Windows** como en **MacOS**, y muy pronto, también en **Linux**.

Lo primero que tenemos que hacer es dirigirnos al sitio web de Unity (**www.unity3d.com**) y descargar la aplicación.



DIRECCIÓN PARA DESCARGA



La dirección directa para realizar la descarga es: **<http://unity3d.com/unity/download>**. Una recomendación es guardar el archivo en el Escritorio, para así poder localizarlo fácilmente más adelante. Este puede tener un tamaño superior a los 400 megabytes.



Figura 1.
En el sitio web oficial de Unity hay enlaces a diversos recursos útiles.

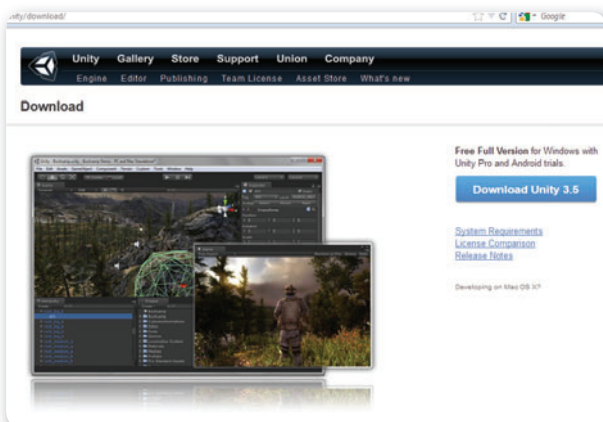


Figura 2.
El botón nos indica la versión que se va a descargar y posee un enlace a los requerimientos del sistema.

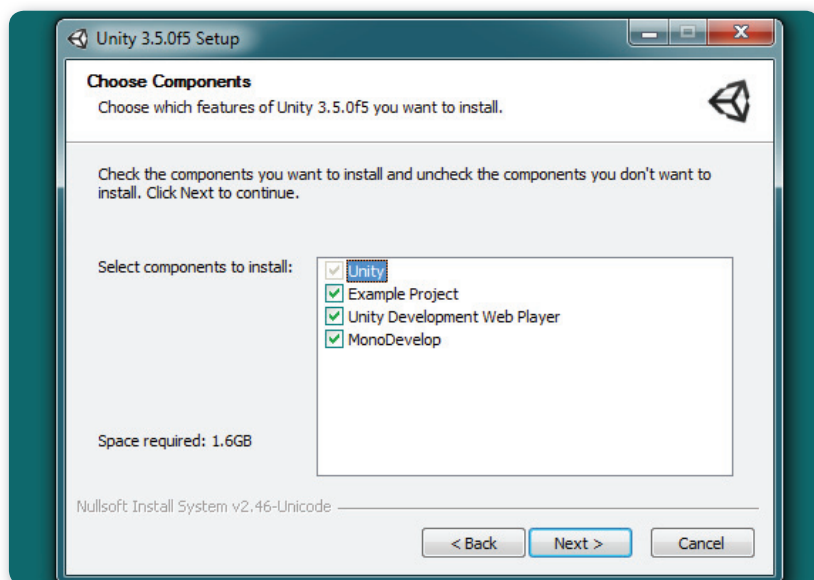


ESPACIO EN DISCO

La instalación completa de Unity requiere de 1,6 gigabytes de espacio en disco. Sin embargo, es conveniente tener más espacio disponible, pues también precisaremos lugar extra para nuestras propias aplicaciones. Los proyectos que desarrollamos pueden consumir algunos cientos de megabytes, dependiendo de la complejidad del juego y de los recursos necesarios.

El siguiente paso consiste en hacer doble clic en el instalador de Unity. Durante el desarrollo de la obra, utilizaremos la versión **3.5**, pero el proceso es similar para todas las demás. El instalador es un programa con un icono similar a un cubo en 3D formado por tres flechas, con la leyenda **UnitySetup-3.5.0**.

Si estamos de acuerdo con los términos de la licencia, los aceptamos y hacemos clic en **Next**; en caso contrario, finalizamos la instalación. Al continuar, llegaremos al cuadro de diálogo de selección de componentes, **Choose Components**, donde procedemos a verificar que las cuatro casillas estén seleccionadas: **Unity**, que es el software propiamente dicho; un **proyecto de ejemplo**, donde veremos algunas de las características del motor, y que incluye el código fuente del juego; el **Web Player**, con el que podemos jugar juegos basados en Unity directamente en el navegador; y por último, **MonoDevelop**, que es usado como editor para el código de nuestras aplicaciones (hay otros editores disponibles, pero este es bastante bueno y gratuito). Una vez que verificamos que todos los componentes están seleccionados, hacemos clic en el botón **Next**.



► **Figura 3.** Para facilitar el proceso de aprendizaje, es conveniente instalar todos los componentes.

Si llegan a aparecer otros cuadros de diálogo durante el proceso de instalación, simplemente dejamos los valores predeterminados y hacemos clic sobre el botón **Next**. Así continuamos hasta llegar al final de esta etapa. En este punto, el cuadro de diálogo muestra el número de versión instalada y nos da la opción de ejecutar Unity una vez que se termina la instalación. Dejamos esta opción seleccionada y oprimimos el botón **Finish**.



► **Figura 4.** Al finalizar, Unity se ejecutará automáticamente, y nos permitirá trabajar con el editor o jugar con el proyecto de ejemplo.



BACKUP Y VERSIONING



Es conveniente mantener **backups** de todos nuestros trabajos en Unity. Los proyectos se guardan en carpetas, lo que facilita su almacenamiento. Es importante hacer copias de todos los que desarrollemos, antes de efectuar cualquier cambio de versión. Las copias frecuentes nos permiten tener versiones pasadas que pueden ser útiles si cometemos errores, porque nos ayudan a evitar eventuales pérdidas de importante información. Este historial de versiones se conoce como **versioning**.

Con esto finaliza el instalador y se lleva a cabo la primera ejecución de Unity. Es normal que este primer arranque tome algo de tiempo y que, en algunos casos, se presenten cuadros de diálogo para indicarnos que se están realizando compilaciones. Luego se abre el editor con el nivel del proyecto de ejemplo en el fondo y una ventana con diferentes opciones. Estas nos llevan a recursos que permiten aprender las bases de Unity, donde encontramos recursos importantes como el foro, para compartir dudas y obtener ayuda de otros usuarios.



► **Figura 5.** En la ventana de bienvenida de Unity tenemos diversas opciones para aprender más.



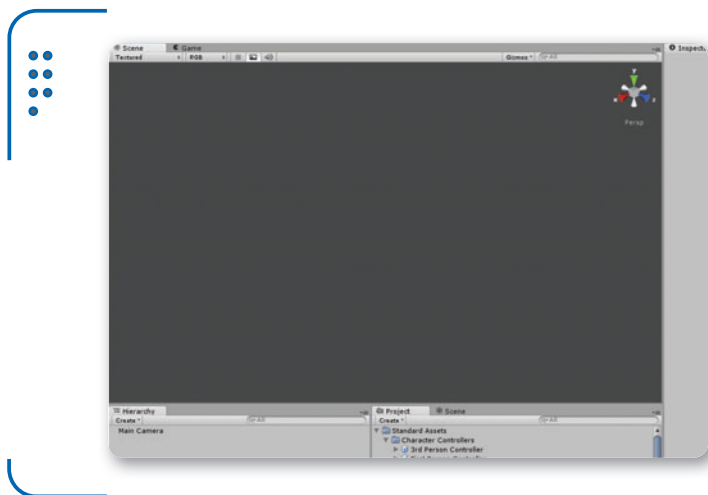
APRENDER A PROGRAMAR



Este libro toma en cuenta que el lector ya tiene, al menos, conocimientos básicos sobre programación, pero si este no es el caso, es recomendable consultar la obra **C# Guía Total del Programador**, de esta misma editorial, en la que se explican, de manera sencilla y paso por paso, los conceptos elementales y los aspectos imprescindibles para aprender a programar.

Paneles principales

Ahora es el momento de conocer la interfaz de Unity, compuesta por diferentes paneles que podemos configurar acorde a nuestras necesidades. Veremos primero las partes más importantes de la configuración inicial.



 **Figura 6.**
La configuración clásica de paneles de Unity permite trabajar rápidamente y de manera sencilla.

El panel Scene

El panel **Scene** (escena) muestra el mundo virtual, y es el lugar en el que nosotros construimos el escenario donde el juego va a desarrollarse y existir. Aquí podemos observar y editar los lugares donde van los objetos, las luces, y otros elementos. También es posible ver otros recursos, como cámara, objetos con audio y ejes coordenados. El panel **Scene** puede presentar diferentes vistas.



WIREFRAME



Hay diferentes formas de renderizado que se pueden utilizar en la red de alambre. En ellas se muestran los vértices que componen el objeto y las aristas que unen esos vértices. Es importante aclarar que los polígonos del objeto no se rellenan. Gracias a esto, es un modo de render muy rápido, aunque no muy realista para los estándares actuales.

En la parte superior encontramos una serie de menús que cambian el modo en que el panel despliega el nivel del juego. El primero se conoce como **Draw Mode**. En su modo default está establecido en **Textured**, con lo cual las superficies se muestran con una textura e iluminación muy similares a como lucirán en el juego. La segunda opción es **Wireframe**, en la que los objetos se presentan en su marco de alambre. Con el modo **Tex-Wire** obtenemos los objetos texturizados, y sobre ellos se dibuja el wireframe. El uso de **Render Paths** es más avanzado, ya que tiene efecto en el desempeño sobre las diferentes plataformas donde se ejecutará el juego. El último modo es **Lightmap Resolution**, que muestra la resolución de los lightmaps en uso.

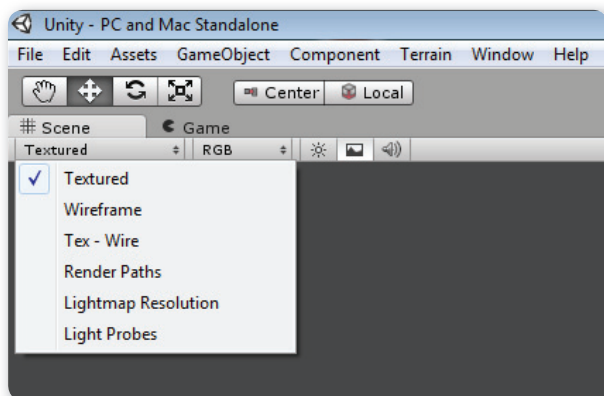


Figura 7. Desde el panel **Scene** podemos controlar el modo de dibujo para el editor de la escena.

El siguiente menú que encontramos, a la derecha del anterior, es **Render Mode** y contiene cuatro opciones. La primera es **RGB**; generalmente trabajaremos en este modo de manera predeterminada. El próximo modo



EL COLOR RGB



El modelo de color que usa Unity es RGB, en el cual el color se define por medio de tres componentes: **rojo**, **verde** y **azul**. Al combinarlos con diferentes intensidades, tenemos la posibilidad de obtener todos los colores que el ser humano puede ver. Existen otros modelos para codificar el color, pero este es el más usado en aplicaciones de videojuegos. Otro componente conocido como **canal alpha** se encarga de controlar el nivel de transparencia de la imagen.

muestra el **Alpha**: si tenemos objetos cuyos materiales estén haciendo uso del canal alpha, veremos los valores renderizados como colores. En el modo **Overdraw** podremos observar en qué lugares un objeto es dibujado sobre otro. Por último, el modo **Mipmaps** sirve para optimizar el tamaño de las texturas. Cuando usamos este modo, el color rojo indica que la textura es más grande de lo necesario. Si vemos el color azul, significa que la textura puede ser más grande. Estas indicaciones son para la posición actual de la cámara y la distancia de los diferentes objetos a ella.

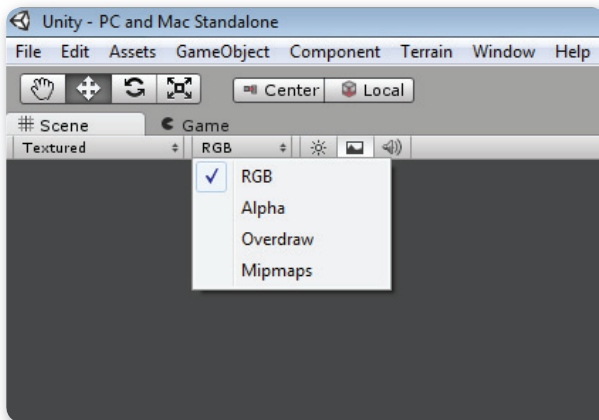


Figura 8.

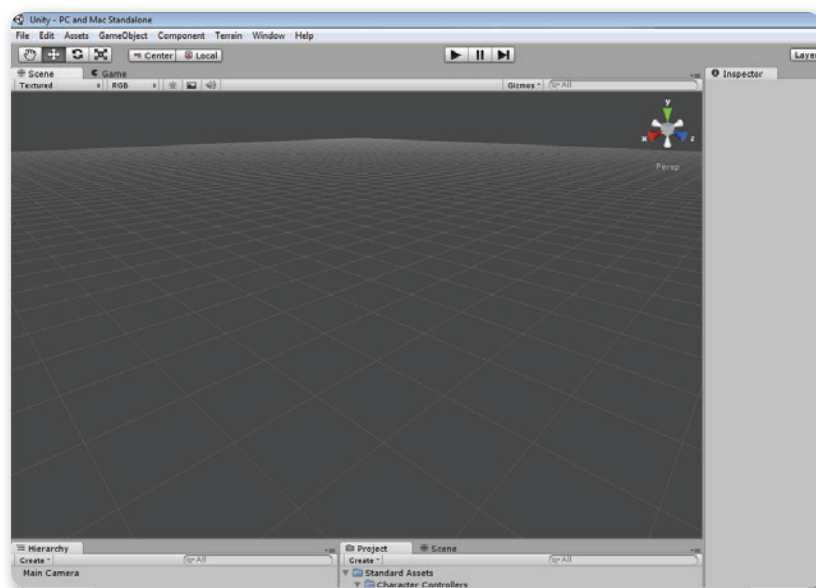
Unity ofrece distintos modos de renderizado, opción de gran utilidad para usuarios avanzados.

Encontramos luego tres botones. El primero tiene el icono de un sol; con él indicamos si la escena, no el juego en ejecución, se va a mostrar con la luz de default o con las luces que nosotros mismos hayamos adicionado. Un poco más adelante aprenderemos a hacerlo y, luego, la forma de usarlas y programarlas. En el segundo botón hay un icono que recuerda una fotografía de un paisaje, y nos permite activar o desactivar una rejilla que sirve de referencia para colocar los objetos. También podemos controlar si en el momento en que estamos editando la escena se muestran objetos del **GUI**, que es la interfaz gráfica de usuario; y **skyboxes**, cubos que presentan una textura con elementos de paisaje y que se usan

ACTIVAR LA
REJILLA SIRVE
DE REFERENCIA
PARA COLOCAR
LOS OBJETOS



para no tener que modelar todo el paisaje alrededor de la escena. Por último, hay un botón con el icono de una bocina, usado para encender o apagar las fuentes de audio que contenga la escena.



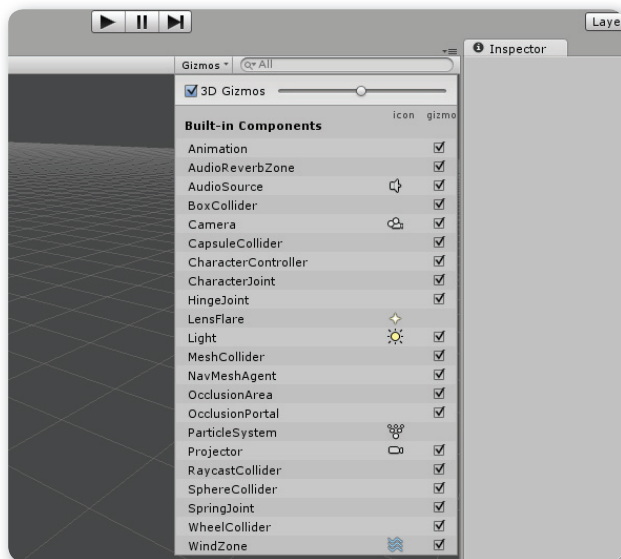
► **Figura 9.** La rejilla permite colocar de manera sencilla los diferentes objetos que componen la escena.

Los **gizmos** son pequeños iconos que aparecen en el editor de la escena y dan indicaciones sobre el tipo de objeto o recurso que representan. Hacia el extremo derecho, al lado del panel **Inspector**, encontramos el menú **Gizmos**, que nos permite seleccionar cuáles son las opciones que deseamos que aparezcan.



UNITY ASSET STORE

En **Unity Asset Store** (<http://unity3d.com/asset-store>) encontramos una gran variedad de recursos útiles que podemos comprar (también los hay gratuitos), que nos ayudarán con el desarrollo de nuestras aplicaciones, ya sea que busquemos geometrías, texturas o scripts.



► **Figura 10.** El menú **Gizmos** permite seleccionar el tamaño y elegir los gizmos que deseamos ver.

Navegar la escena

Es el momento de aprender cómo movernos dentro de la escena. Si queremos observar correctamente los diferentes movimientos, es recomendable colocar la rejilla. Para movernos por la escena, primero es necesario que esta se encuentre activa, de modo que hacemos clic con el mouse en cualquier lugar de ella que esté vacío.

El primer movimiento se lleva a cabo con las flechas del teclado. La flecha hacia arriba permite avanzar hacia el frente; este es un buen momento para intentarlo. Con la flecha hacia abajo retrocedemos,



EL COLOR DE LOS EJES



Unity tiene una clave de color para cada uno de los ejes coordenados, tanto de la escena como de los objetos. El eje **Y** tiene el color verde, el eje **X** usa el rojo y para el eje **Z** se utiliza el azul. Cuando editamos un objeto, estos colores nos permiten identificar rápidamente cada uno de sus ejes.

mientras que las flechas hacia la izquierda o derecha nos permiten deslizarnos hacia los lados.

Si movemos el mouse mientras oprimimos el botón central sin soltarlo, podemos navegar por la escena. Cuando el movimiento del mouse es hacia delante, subimos; cuando es hacia atrás, bajamos; y cuando es hacia un lado, nos deslizamos en esa dirección.

ESTAS HERRAMIENTAS
NOS PERMITIRÁN
NAVEGAR POR LA
ESCENA CON
GRAN FACILIDAD



El botón derecho del mouse nos permite cambiar la dirección hacia donde estamos viendo dentro de la escena; es similar a mover la cabeza en vez de desplazarnos en el espacio. En este caso, debemos hacer clic con el botón derecho y, sin soltarlo, mover el mouse; observaremos cómo cambia la dirección hacia donde estamos viendo.

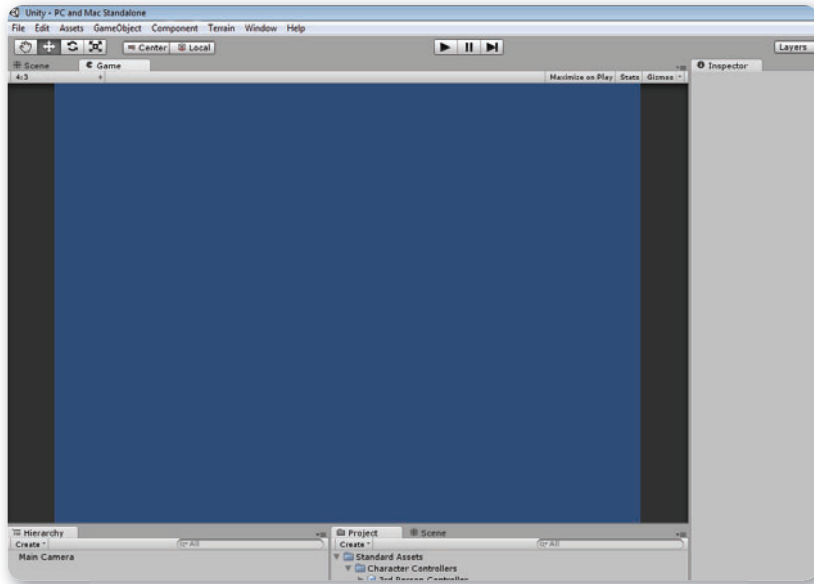
Por último, veremos la manera de **hacer zoom**. Simplemente, tenemos que mantener oprimido el botón derecho del mouse y, al mismo tiempo, la tecla **ALT**, y sin soltarlos, mover el mouse. Así

podremos alejarnos o acercarnos a aquello que estemos observando. Estas herramientas nos permitirán navegar por la escena con gran facilidad y sin problemas, pero aún hay otros elementos que también pueden ayudarnos.

En la parte superior derecha, dentro del panel **Scene**, hay un gizmo que muestra las coordenadas y sirve para orientarnos dentro de la escena. Si movemos la vista, también se actualiza, indicando hacia dónde se encuentra cada eje. Pero este gizmo no solamente hace esto: también nos permite cambiar la vista. Si hacemos clic en cualquiera de los conos que representan los ejes, nuestra vista cambiará como si estuviéramos observando la escena desde ese eje. Para regresar a una vista en perspectiva, simplemente hacemos clic en el cubo del centro.

El panel Game

En el panel **Game** (juego) podemos ejecutar el juego y verlo tal como aparecerá en pantalla. Para hacerlo, localizamos una pestaña junto a la del panel **Scene** que tiene un icono similar a la figura del **Pac-Man**. Al hacer clic sobre él, aparece en pantalla lo que capta la cámara en el interior de nuestro juego. De este modo, podemos ver la manera exacta en que el videojuego se presentará en ejecución..



► **Figura 11.** Nuestra aplicación de juego ya se ejecuta, aunque la escena todavía está vacía.

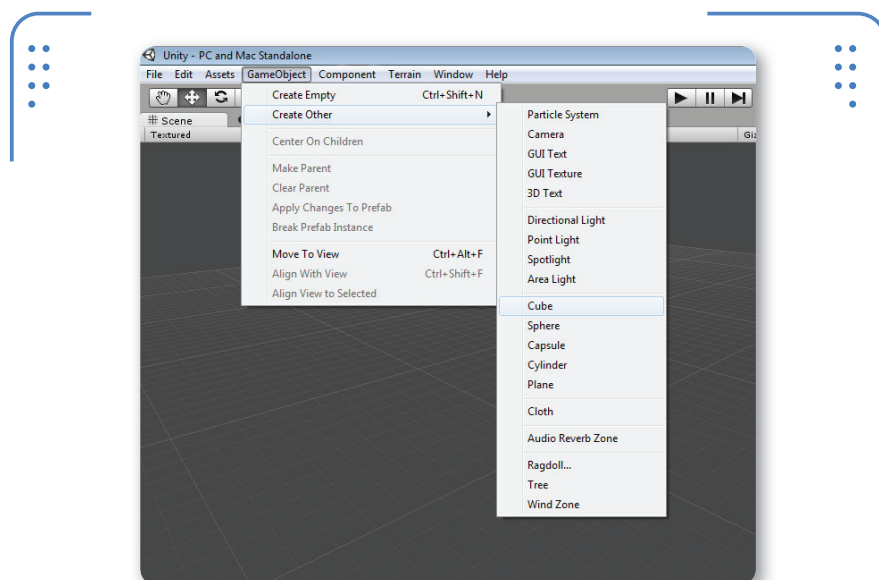
En la parte superior de la pantalla aparecen tres botones. El primero tiene el icono de **Play**, que es un triángulo. Cuando lo presionamos, el juego empieza a ejecutarse; si lo oprimimos otra vez, se detiene. Podemos pausar el juego si así lo necesitamos por medio del botón central. El último botón nos ayuda en las depuraciones, pues con él podemos avanzar el juego paso por paso.

Es importante mencionar también que tenemos una serie de menús para este panel. El primero permite cambiar la proporción de la ventana donde se renderiza el juego. Si sabemos que el desarrollo que estamos realizando va a ser usado con una determinada resolución de pantalla, podemos modificar aquí la proporción, y observarlo de la forma adecuada. Luego encontramos la opción **Maximize on Play**, la cual permite observar el juego en pantalla completa cada vez que lo ejecutamos. Con la opción **Stats** podemos observar las estadísticas de desempeño de nuestra aplicación; es útil para llevar a cabo optimizaciones en el juego. El menú **Gizmos** tiene en este panel las mismas funcionalidades que en el panel anterior.

Adicionar un objeto

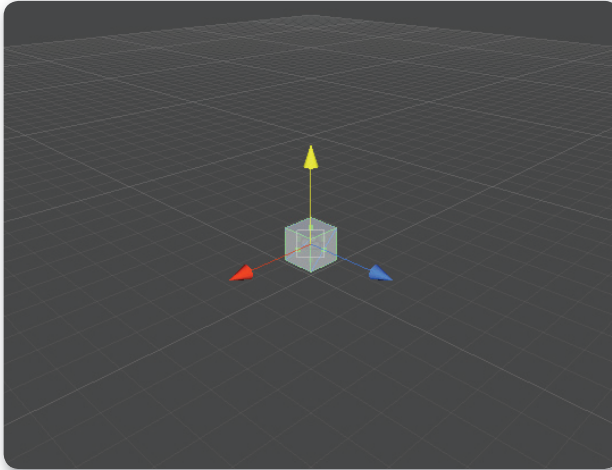
Antes de continuar aprendiendo sobre los demás paneles, vamos a adicionar objetos, ya que esto nos permite entenderlos mejor. Los objetos que agregamos a la escena se conocen como **Game Objects** y los hay de diferentes tipos. El objeto vacío no presenta ninguna geometría, pero es muy útil para colocar código o lógica de soporte. También se pueden añadir sistemas de partículas, cámaras y elementos para la interfaz de usuario. Los diferentes tipos de luces se agregan de la misma forma.

En este momento añadimos algunos objetos con geometrías sencillas y, luego, una luz para la escena. Para esto, seleccionamos el menú **Game Object** y, a partir de ahí, el submenú **Create Other**, donde aparece un menú con la lista de los posibles objetos, de la cual seleccionamos la opción **Cube**.



► **Figura 12.** El menú **Game Object** nos permite colocar diversos tipos de objetos en la escena.

Un cubo aparece en la escena y está seleccionado, por lo que muestra los ejes en su posición. Si hubiera otros objetos, solamente el seleccionado mostraría dichos ejes.



► **Figura 13.** Unity permite cambiar la posición de los objetos, y sus atributos de escala y rotación.

En la parte superior izquierda de la ventana podemos observar una serie de iconos, entre los cuales hay uno que tiene la forma de una cruz de flechas; cuando lo seleccionamos, nos permite usar la herramienta para mover el cubo, que es la que aparece originalmente. Si arrastramos las flechas que aparecen en el cubo, podemos cambiar su posición con facilidad. Al arrastrar el centro del cubo, el movimiento es libre.

Ahora hacemos clic en el botón que tiene un icono con la forma de unas flechas curvas. Esta es la herramienta que nos permite modificar la

SI ARRASTRAMOS
LAS FLECHAS QUE
APARECEN EN
EL CUBO, PODEMOS
CAMBIAR SU POSICIÓN

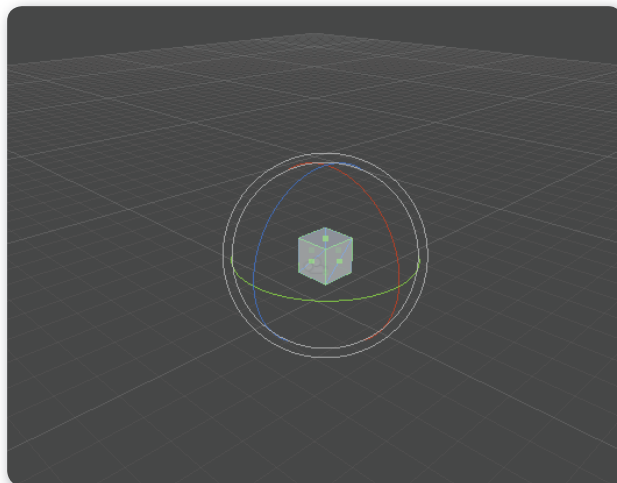
”



CAMBIAR LOS VALORES VELOZMENTE



Si deseamos cambiar con rapidez los valores de una propiedad, en vez de escribirlos, podemos hacerlo de una manera sencilla: simplemente, colocamos el puntero del mouse sobre el nombre de la propiedad y lo arrastramos. Conforme este se mueve, los valores cambian.



► **Figura 14.** En Unity tenemos la posibilidad de rotar el objeto sobre alguno de sus ejes o de manera libre.

orientación del objeto, es decir, rotarlo. Alrededor del cubo se presenta una serie de círculos: al arrastrarlos, giramos el objeto.

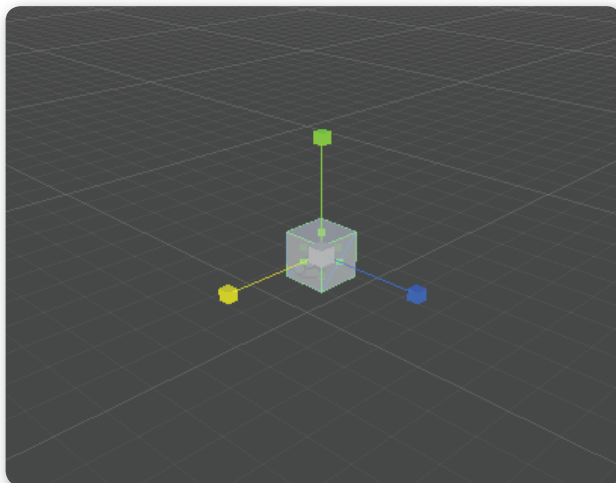
Nos falta conocer una herramienta más. En el grupo de botones existe uno con un icono que muestra un cuadro con flechas saliendo en direcciones diagonales. Esta es la herramienta de escalamiento, que permite modificar el tamaño del objeto. La modificación puede realizarse en todos los ejes simultáneamente o solo en alguno de ellos de manera aislada. Cuando seleccionamos este botón, vemos que la herramienta aparece sobre el objeto seleccionado. Al igual que con las otras herramientas, al arrastrar el mouse se llevan a cabo los cambios necesarios. Esta posibilidad es de gran utilidad a la hora de modificar las proporciones de los objetos colocados en la escena.



ENCONTRAR RÁPIDAMENTE UN OBJETO

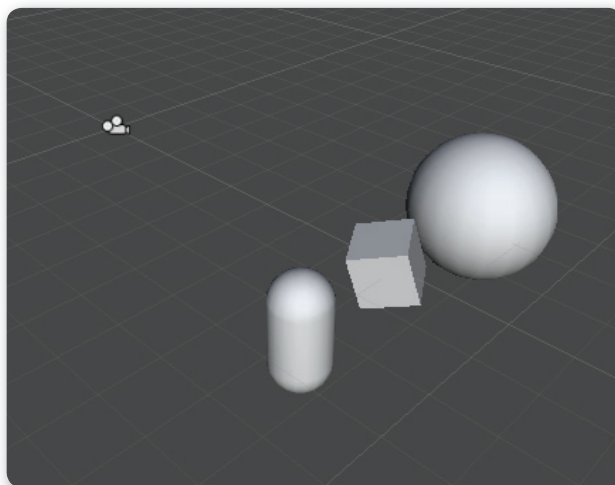


Encontrar un objeto con rapidez puede ser una necesidad frecuente a la hora de programar videojuegos. Podemos lograrlo fácilmente haciendo clic en el panel **Hierarchy**; luego, colocamos el puntero del mouse sobre el panel **Scene** y oprimimos **F**.



► **Figura 15.** Podemos escalar los objetos de manera uniforme o modificando cada eje independientemente.

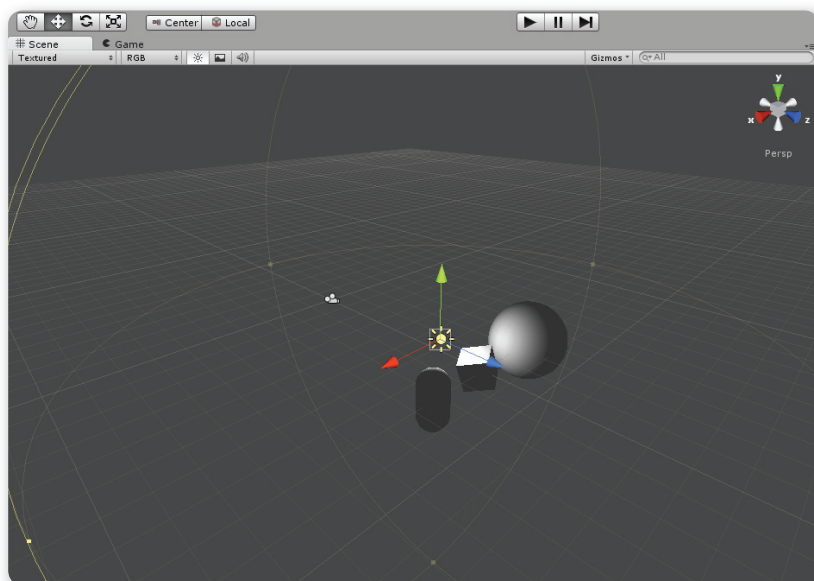
A continuación, como ejercicio, vamos a agregar una esfera, un cilindro y una cápsula, de modo que luzcan como en la **Figura 16**.



► **Figura 16.** Se puede notar el gizmo de la cámara y que el cubo está rotado gracias al uso de la herramienta.

El panel Hierarchy

Este panel es muy útil porque nos muestra un listado de todos los objetos que estamos colocando en nuestra escena. Como su nombre lo indica, **Hierarchy** (jerarquía), muestra las relaciones jerárquicas que pueden existir entre los objetos. Al hacer clic sobre el nombre del objeto, lo seleccionamos y podemos proceder a editarlo. Una vez que tenemos seleccionado el objeto, oprimimos el botón derecho, y aparece un menú contextual, desde el cual podemos duplicar, eliminar, cambiar el nombre, copiar y pegar el objeto.



► **Figura 17.** Si movemos la luz, podemos observar en tiempo real los cambios de iluminación correspondientes.

Si en la escena no aparece la cámara, podemos seleccionarla a través del panel **Hierarchy** y oprimir **F** sobre el panel **Scene**. Con esto encontramos la cámara, y por medio de las herramientas que ya conocemos, la colocamos en posición.

En el panel **Hierarchy** también encontramos un menú que nos permite crear rápidamente los objetos que podemos colocar en la escena, sin necesidad de utilizar el menú principal de Unity.

Ahora podemos añadir una luz. Si bien existen diferentes tipos de luz (que conoceremos a fondo más adelante en esta obra), en este momento trabajaremos con una de tipo **Point Light**. Esta luz es similar a la proyectada por una vela o una lámpara incandescente, pues se distribuye de manera uniforme en todas direcciones. La luz puede colocarse cerca de los tres objetos y un poco por arriba. Inmediatamente después de hacerlo, Unity deja de utilizar la iluminación predefinida y pasa a usar la nueva iluminación indicada.

AL HACER CLIC
SOBRE EL NOMBRE
DEL OBJETO, LO
SELECCIONAMOS Y
PODEMOS EDITARLO



El panel Project

En el panel **Project** (proyecto) encontramos los recursos que componen nuestro juego. Hay diferentes tipos de elementos disponibles. Si el elemento tiene el icono de Unity, significa que es una escena o un nivel. Podemos acceder a su edición haciendo doble clic en su icono.

Es conveniente tener carpetas que nos ayuden a organizar los recursos. En este panel también podemos hallar los scripts que usamos para proveer de lógica a nuestros objetos.

Más adelante aprenderemos a crear **prefabs**, objetos que podemos reutilizar. Los prefabs también aparecen en este panel, y para usarlos, simplemente tenemos que arrastrarlos hasta la escena. Otros recursos, como audio, texturas, materiales, y otros más, pueden colocarse aquí. El hecho de hacer un buen uso de este panel nos ayuda a simplificar y organizar el trabajo.

Unity nos brinda muchos scripts que podemos utilizar y algunos objetos de uso común. Estos recursos también aparecerán en este panel cuando los importe.



CONTRAER LOS COMPONENTES



Si en un momento dado tenemos tantas secciones de componentes que se dificulta la edición del objeto, podemos contraerlas haciendo clic sobre su nombre. De esta manera, sus propiedades se esconden temporalmente. Para mostrarlas otra vez, volvemos a hacer clic sobre el nombre.

El panel Inspector

El último panel que debemos conocer es **Inspector** (inspector). Para acceder a él, seleccionamos un objeto, ya sea por medio del panel **Hierarchy** o haciendo clic en la escena. El panel **Inspector** muestra sus propiedades, divididas en secciones para facilitar su uso. Conforme agreguemos nuevas características a los objetos, veremos que se añaden secciones. Dentro de ellas, están las propiedades que les corresponden. Unity permite editar sus propiedades para tener un control preciso del objeto. Podemos encontrar diferentes elementos, como cuadros de texto y casillas de verificación.

Si hemos seleccionado uno de los objetos o la cámara, en la parte de arriba veremos una sección que corresponde al componente **Transform**. Este nos da control sobre la posición, orientación y escala del objeto. Podemos colocar directamente en coordenadas la posición que deseamos para el objeto.

Si tenemos dudas sobre el trabajo de algún componente en particular, podemos resolverlas haciendo clic sobre el icono de ayuda que aparece del lado derecho en el encabezado de cada uno de ellos. Junto a él, encontramos otro icono que luce como un engranaje, donde podemos resetear algunas propiedades, el componente completo o removerlo.

Ejecutar la aplicación

Cuando editamos la cámara, ya sea por medio de las herramientas o colocando directamente sus propiedades, observamos que aparece una pequeña pantalla en el panel **Scene**. Esta muestra lo que está viendo la cámara, y es muy útil en este proceso. Debemos ubicar y orientar la cámara de tal modo que enfoque los tres objetos que hemos puesto.

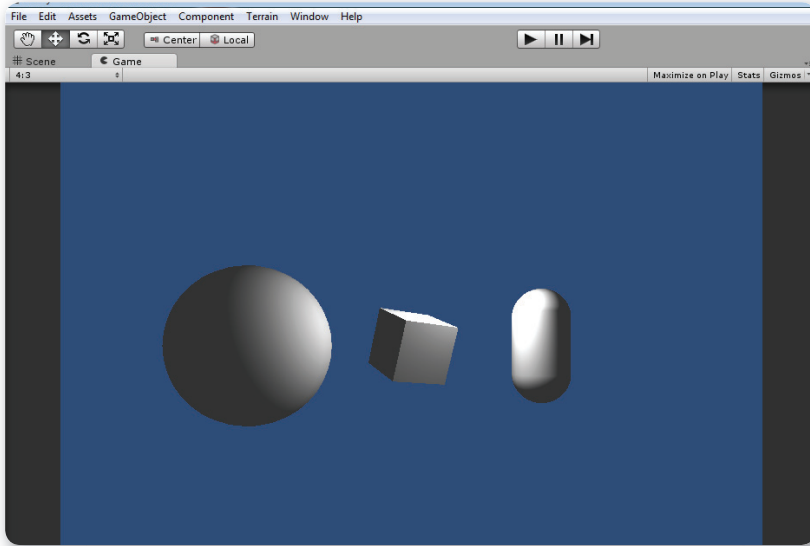


RESETEAR EL COMPONENTE



Algunas veces, cuando agregamos o editamos un objeto, queda en una posición que no deseamos. Podemos resetear el componente desde **Transform**, y entonces se colocará automáticamente en el origen. A partir de ahí, lo editamos según nuestras necesidades.

Una vez que esté lista, nos movemos al panel **Game** y observamos los tres objetos iluminados por la luz, tal como los ve la cámara. Podemos oprimir el botón **Play** para que el juego inicie. Aunque en este momento parece que nada sucede, la aplicación se está ejecutando. No aparece nada porque no hemos colocado hasta este momento ninguna lógica en particular, pero muy pronto lo haremos.



► **Figura 18.** Aun sin ninguna interacción lógica colocada, podemos observar la manera en que se ejecuta la aplicación.

En el siguiente capítulo aprenderemos a colocar lógica a los objetos para que se muevan por medio del uso de scripts.



RESUMEN

Unity nos permite crear videojuegos de una manera sencilla, ya que nos provee de un editor que facilita la generación de niveles, y tiene diferentes paneles que nos ayudan a trabajar con los objetos y recursos que vamos a utilizar. Es posible usar herramientas o colocar las propiedades a los objetos directamente. Podemos ejecutar nuestra aplicación para comprobar la lógica y ver cómo lucirá cuando quede compilada.

Actividades

TEST DE AUTOEVALUACIÓN

- 1 ¿Cuántos paneles tiene Unity?
- 2 ¿Cuál es el proceso para insertar un objeto?
- 3 ¿De qué forma podemos eliminar un objeto de la escena?
- 4 ¿Cuál es el componente que permite determinar la posición del objeto?
- 5 ¿Dónde se localizan las herramientas que permiten trasladar, rotar y escalar los objetos?
- 6 ¿En qué panel pueden verse los recursos disponibles para la aplicación?
- 7 ¿Para qué se usa el panel **Hierarchy**?
- 8 ¿Cómo podemos observar lo que la cámara está enfocando?
- 9 ¿Cómo se ven los objetos en el juego si no adicionamos ninguna luz, y cómo lucen en la escena?
- 10 ¿Cómo puede verse la escena en **wireframe**?
- 11 ¿Cuáles son las dos formas en las que se pueden agregar los objetos?
- 12 ¿Qué se necesita oprimir para que empiece a ejecutarse el juego?

ACTIVIDADES PRÁCTICAS

- 1 Mueva los objetos de las diferentes formas que hemos aprendido aquí.
- 2 Juegue con la posición y orientación de la cámara para aprender a colocarla correctamente.
- 3 Intente duplicar los objetos.
- 4 Cree la figura de una casa o un castillo con los objetos básicos que ofrece Unity.
- 5 Coloque otros tipos de luces e identifique sus diferencias.

Trabajar con objetos

En este capítulo aprenderemos a colocar scripts a nuestros objetos para que tengan lógica y dejen de ser estáticos. Conoceremos la base para poder incorporar elementos interactivos a nuestros videojuegos, que contribuirán a mejorar enormemente la experiencia del jugador. Nos concentraremos en la programación de las propiedades que se encuentran en el componente **Transform**.

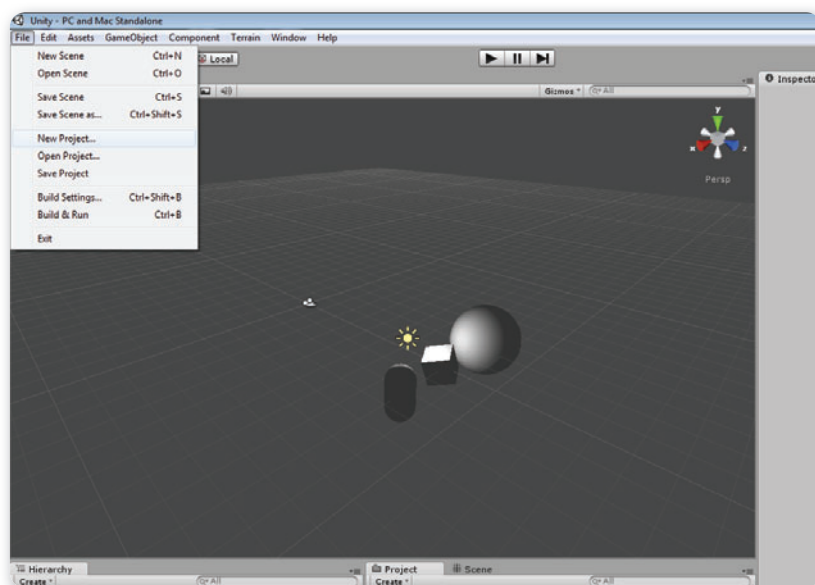
▼ Programar la posición 38	▼ La rotación..... 61
Adicionar los objetos y materiales..... 39	Múltiples scripts..... 62
Guardar la escena..... 42	Rotación por tiempo 62
Crear scripts 43	Rotación alrededor de un punto 64
Declaración de variables..... 47	
Crear propiedades..... 51	▼ Proyecto de sistema solar 65
	Adicionar texturas 66
▼ Mover los cuerpos..... 54	Agregar el código 67
Movimiento por cuadro 54	
Movimiento por tiempo..... 56	▼ Resumen 69
▼ Escalamiento 58	▼ Actividades..... 70



Programar la posición

Es momento de indicar la posición de los objetos, lo que nos permitirá moverlos de acuerdo con la lógica que deseamos cuando el juego esté en ejecución. También vamos a utilizar por primera vez materiales diferentes para cada uno de los objetos.

Empezamos por crear un nuevo proyecto, donde vamos a guardar diferentes escenas; cada una puede ser un nivel del juego. Abrimos el menú **File** y seleccionamos **New Project**.



► **Figura 1.** Empezamos por crear un nuevo proyecto para poder generar las diferentes escenas de este capítulo.



LOS PAQUETES

Unity provee de una gran cantidad de paquetes que contienen objetos y scripts que pueden sernos de utilidad. Los coloca automáticamente dentro de nuestro proyecto al seleccionarlos. Algunas empresas venden paquetes que podemos adicionar al programa.

Unity nos pregunta cuáles son los paquetes que deseamos importar. En este momento, no queremos colocar uno en particular, por lo que no seleccionamos ninguno. También vamos a colocar el nombre y la ubicación del proyecto. Usamos el directorio preestablecido y lo llamamos **Capítulo2**.

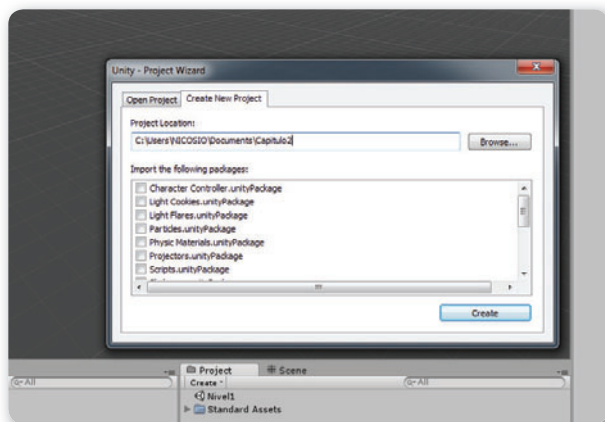


Figura 2. Colocamos el nombre y la ubicación del proyecto. No seleccionamos ningún paquete para el proyecto actual.

Adicionar los objetos y materiales

Para que nuestro ejemplo funcione de mejor manera, primero verificamos que la cámara esté en las coordenadas **(0.1,-10)**. Una vez realizado este paso, creamos un cubo y lo ubicamos en las coordenadas **(0.1,-4)**. Este es un cubo similar al del capítulo anterior, pero vamos a agregarle un material que se encargará de iluminar el objeto. Podemos utilizar materiales simples, que solo den color, u otros con texturas para que, por ejemplo, parezca que el objeto es de madera o metal; incluso, hay materiales muy avanzados que hacen uso de **shaders** complejos.

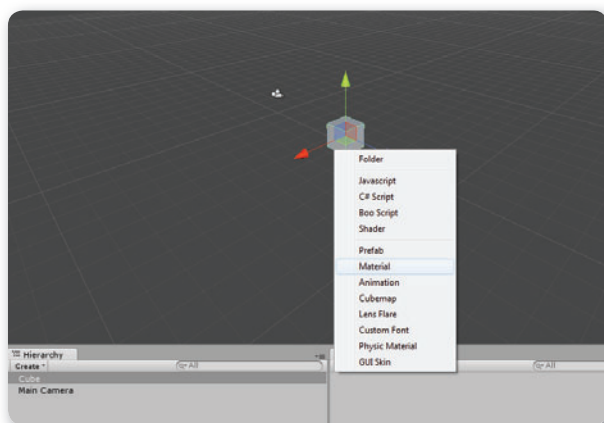


IMPORTACIÓN DE PAQUETES



Por lo general, hacemos la importación de los paquetes cuando estamos creando el nuevo proyecto. Debemos seleccionar solo los que vamos a usar realmente. Si luego descubrimos que nos hace falta otro, podemos importarlo sin tener que crear el proyecto otra vez.

Para crear el material nos dirigimos al panel **Project**, y en su menú **Create** hacemos clic con el botón del mouse. Aparece un menú de donde seleccionamos la opción **Material**. Esta acción crea un nuevo material y lo coloca en el panel **Project**.



► **Figura 3.**
Por medio del
menú Create en
el panel Project,
podemos
agregar uno
o varios
materiales.

Allí tenemos la opción de darle un nombre; en este caso, **Rojo**. Si lo seleccionamos, veremos que, del lado derecho, el panel **Inspector** muestra sus propiedades. Por default, Unity coloca un material de tipo **Diffuse**, que emplea un shader muy sencillo, pero podemos usar otra clase de shaders.

En este momento, únicamente haremos uso del color. Vemos que el panel **Inspector** muestra un rectángulo de color gris en la parte superior junto a un icono que luce como un gotero. Al hacer clic en ese rectángulo, se abre una nueva ventana, que es la de selección de color, donde elegimos el rojo para nuestro material.



COLOCAR EL COLOR PRECISO



Es posible colocar el color con exactitud en el cuadro de selección de color. Para hacerlo, contamos con múltiples herramientas. Lo mejor es experimentar con este cuadro y probar las opciones de los iconos que aparecen, pues con ellos se modifica la herramienta. En último caso, podemos indicar directamente los valores de los componentes de color.

Una vez que tenemos el material, simplemente falta asignarlo a los objetos que van a utilizarlo. La forma más sencilla de lograrlo es hacer clic sobre él en el panel **Project** y arrastrarlo sin soltar el botón del mouse hasta el objeto que lo va a usar. También podemos arrastrarlo hasta el objeto listado en el panel **Hierarchy**.

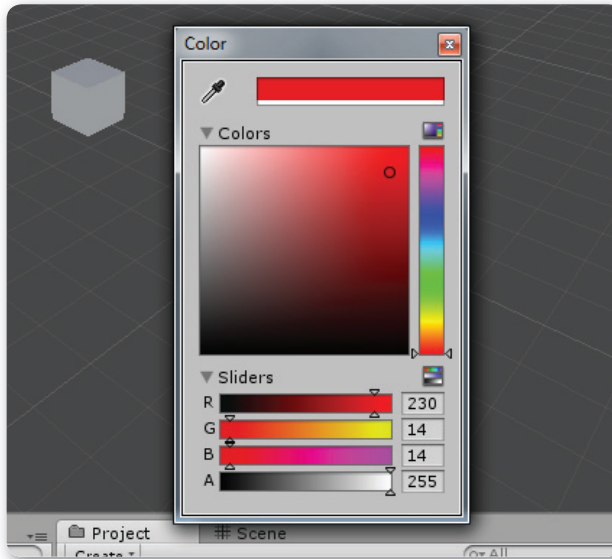


Figura 4. La ventana de selección de color nos permite definir fácilmente el tono que deseamos.

Ya tenemos la primera parte de la escena con la cámara y un objeto. Tal como aprendimos anteriormente, añadimos dos cubos más. Al primero lo situamos en las coordenadas **(-2.1,-4)** y lo llamamos **CuboAzul**. El segundo será **CuboVerde** y se ubica en las coordenadas **(2.1,-4)**. A continuación, creamos dos nuevos materiales. El primero, llamado **Azul**, tendrá la tonalidad de azul que más nos guste y lo asignaremos al cubo **CuboAzul**. El otro material será **Verde**, y lo asignaremos al cubo **CuboVerde**.

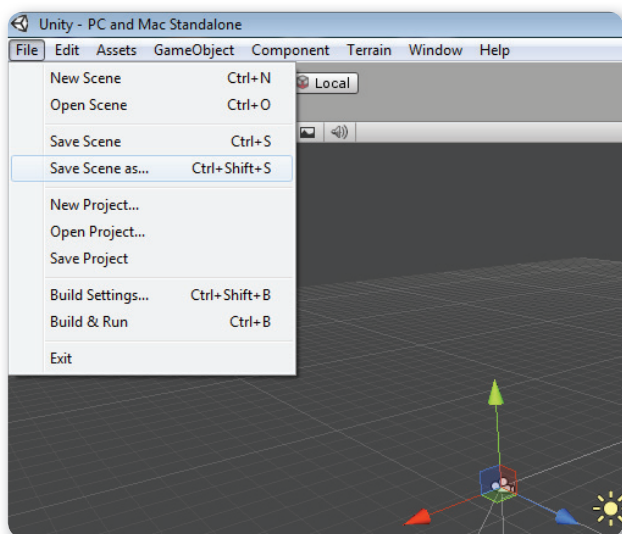
POR DEFAULT, UNITY
COLOCA UN MATERIAL
DE TIPO DIFFUSE, QUE
EMPLEA UN SHADER
MUY SENCILLO



Nuestra escena ya luce mejor, pero nos estamos olvidando de algo. Si no colocamos una luz, no podremos ver los objetos cuando se ejecute la aplicación. Para resolver este tema, adicionamos una luz de tipo **PointLight** (de punto) y la colocamos en las coordenadas **(0.2,-5.5)**. Para hacerlo, vamos al menú **GameObject** y seleccionamos **Create Other**, donde aparece una lista de objetos por crear; allí hacemos clic en **Point Light**. Veremos más a fondo el uso y manejo de las luces en el **Capítulo 6**. A continuación aprenderemos a guardar la escena..

Guardar la escena

Cuando hemos realizado cierto trabajo, es necesario guardar la escena. Las escenas son equivalentes a los niveles en el juego y podemos tener tantas como sean necesarias. La aplicación final que se distribuya al público deberá contener al menos una escena, y un proyecto estará compuesto de una o varias. Para guardarlas, vamos al menú **File** y seleccionamos **Save Scene As**. En el cuadro de diálogo que aparece colocamos como nombre de la escena **Ejemplo1**. Con esto, el trabajo ha quedado guardado.



► **Figura 5.** Es importante guardar la escena y que esta represente un nivel del juego en nuestra aplicación.

Crear scripts

Si ejecutamos nuestra aplicación, veremos que en el panel **Game** se muestra la ejecución como se verá en el juego. Podemos observar los tres cubos en su posición siendo iluminados. La forma en que se ven ha mejorado mucho en comparación con el ejemplo del capítulo anterior. Sin embargo, no hay movimiento ni interacción. Esto se debe a que no hemos colocado ninguna lógica a los objetos.

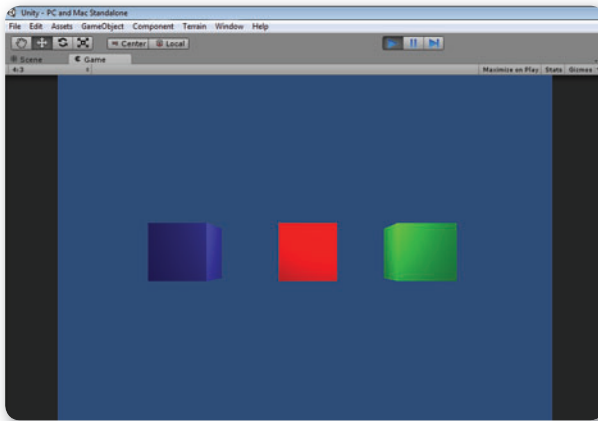


Figura 6.

Al ejecutar la aplicación, aparecen los cubos, pero no hay interacciones.

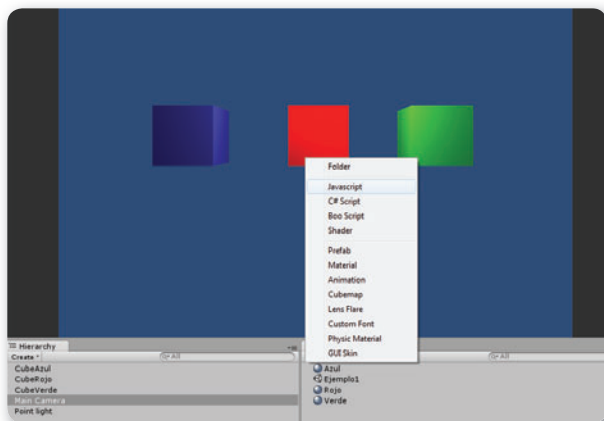
Empecemos por crear un script, que es un documento de texto donde colocamos el programa que deseamos que ejecute el objeto. En este libro, usaremos como lenguaje **JavaScript**, pero cualquier persona que tenga conocimientos sobre programación en **C#**, **C/C++**, **Java** o algún otro no debería de tener problemas para entender y escribir el código.

Para adicionar el script vamos al panel **Project** y, en **Create**, seleccionamos **JavaScript**. Esto adiciona un recurso de tipo script de Java; lo nombramos **PosicionExplicita**.



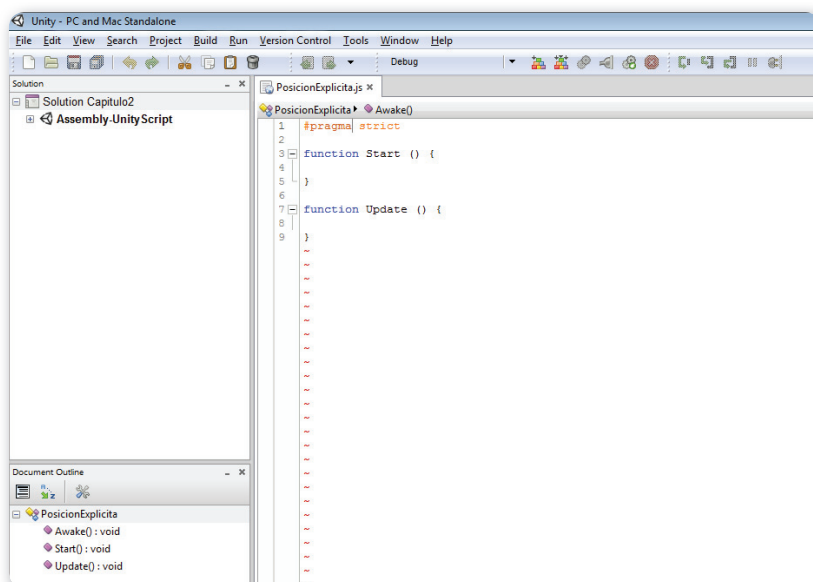
REASIGNAR MATERIALES

Es posible editar el material una vez que ha sido asignado a uno o varios objetos. Al hacerlo, todos los objetos que usen ese material mostrarán el cambio. También podemos asignar un nuevo material a un objeto si así lo necesitamos o deseamos; simplemente, lo arrastramos hasta él.



► **Figura 7.** Para adicionar un script usamos el menú **Create** del panel **Project**.

En este script colocaremos una nueva posición para uno de los cubos, diferente de la original, e incluida en la ejecución del programa. Ya tenemos el script, pero para editarlo, necesitamos hacer doble clic



► **Figura 8.** Mono Developer es la aplicación donde llevamos a cabo la edición de los scripts.

en él dentro del panel **Project**. Entonces, se ejecuta **Mono Develop** y podemos colocar ahí nuestro código. Tengamos en cuenta que puede demorar unos segundos en aparecer.

Unity tiene una serie de funciones especiales que son invocadas en ciertos momentos cuando se ejecuta la aplicación; podemos hacer uso o no de ellas según nuestras necesidades. Al principio, presenta las funciones **Start()** y **Update()**. La primera se ejecuta al iniciar el programa o cuando se crea el objeto. Es un buen punto para llevar a cabo inicializaciones o asignación de valores de inicio. Esta función solo se ejecuta una vez en el ciclo de vida del objeto. Por su parte, **Update()** es invocada en cada cuadro del juego; es decir, se puede invocar 30, 60 o más veces por segundo, dependiendo de la computadora en la que se realice la ejecución. En ella colocaremos la lógica principal del objeto, su control primario. Debemos recordar que se ejecuta constantemente, y programar pensando en esos términos. Es posible dejar las funciones vacías si no vamos a usarlas.

Empecemos por colocar el siguiente código:

```
#pragma strict
```

```
// Este programa coloca de forma explicita la posicion a un objeto
```

```
// Esta funcion se invoca automaticamente y es un buen lugar para hacer inicializaciones
```

```
function Start () {
```



UNITY TIENE
FUNCIONES
ESPECIALES QUE
SON INVOCADAS EN
CIERTOS MOMENTOS



MISMO SCRIPT EN VARIOS OBJETOS



Es posible asignar el mismo script a varios objetos; de esta manera, cada uno de ellos ejecutará una versión del script. Esta es una forma muy útil y simple de reutilizar código y agilizar el proceso de desarrollo. Un objeto también puede tener más de un script, los cuales se ejecutarán simultáneamente.

```
// Colocamos el objeto en la posicion (-2,-1.5,-2.1)
// de manera explicita en cada una de las coordenadas
transform.position.x=-2.0;
transform.position.y=-1.5;
transform.position.z=-2.1;

}

function Update () {

}
```

En este programa únicamente colocaremos una posición inicial de forma explícita. Una de las maneras de hacerlo es a través de la función **Start()**. Para colocar la posición, vamos a usar el componente **transform**, ya que en él es donde se guarda la propiedad que indica la posición del objeto. Dentro de **transform** indicamos la propiedad **position** y, luego, cada uno de sus componentes. Damos explícitamente el valor de **-2** en el eje **X**, **-1.5** en el eje **Y** y **-2.1** en **Z**; podemos indicar cualquier valor que sea necesario. Cuando se ejecute el programa, el cubo aparecerá en esta posición. Luego, simplemente guardamos el script, y cuando regresemos a Unity, se actualizará con el código nuevo. Debemos verificar que no haya ningún error de sintaxis.

El script ya está creado, pero ahora debemos asignarlo, para lo cual lo arrastramos desde el panel **Project** hasta el objeto que deseamos que lo tenga. Observaremos que el panel **Inspector** de ese objeto adiciona un componente para el script. En este ejemplo, asignamos el script al cubo de color azul.

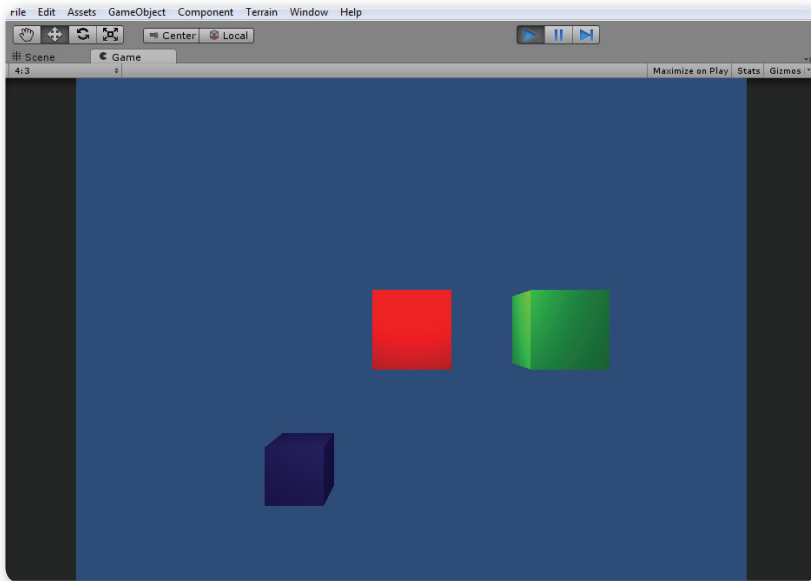
Si ejecutamos el juego en este momento, notaremos que, efectivamente, el cubo toma ahora su nueva posición.



ELIMINACIÓN DE UN SCRIPT



Si por cualquier motivo deseamos que un objeto ya no tenga determinado script, nos dirigimos al panel **Inspector** y, una vez localizado el componente del script, hacemos clic en el icono del engranaje y seleccionamos **Remove Component**. Esto elimina el componente script del objeto.



► **Figura 9.** Podemos observar que el cubo toma su nueva posición en concordancia con el código ingresado.

Declaración de variables

En el código anterior hemos colocado cada uno de los componentes de la posición por separado, pero también podemos hacerlo de manera diferente, por ejemplo, pasando un vector de tres componentes. Para realizar esta tarea, Unity incluye una clase llamada **Vector3**.

Ahora vamos a crear otro script llamado **PosicionExplicita2** y le colocamos el siguiente código:

```
#pragma strict

// Este programa coloca de forma explicita la posicion a un objeto

// Esta funcion se invoca automaticamente y es un buen lugar para hacer
// inicializaciones
function Start () {
```



```
// Colocamos el objeto en la posicion (-2.5,2.0,2.0)
// usando un vector en este caso

// Creamos la variable de tipo Vector3
var posicion = Vector3(-2.5, 2.0, 2.0);

// Asignamos a la posicion
transform.position=posicion;

}

function Update () {

}
```

PRIMERO DEBEMOS
ELIMINAR EL SCRIPT
ANTERIOR DEL CUBO
Y, LUEGO, ASIGNARLE
EL NUEVO



En este código generamos una variable llamada **posicion**. Para crearla, hacemos uso de **var** seguido del nombre y, luego, la información de su tipo. El tipo puede darse directamente, o el compilador lo deduce basado en el valor asignado. Al momento de crear la variable, estamos asignando un valor de tipo **Vector3**, el cual inicializamos con los valores **-2.5, 2 y 2**. Después de esta línea **posicion** contiene dicho vector. La propiedad **position** del componente **transform** es una variable de tipo **Vector3**, por lo que podemos hacer la asignación sin problemas.

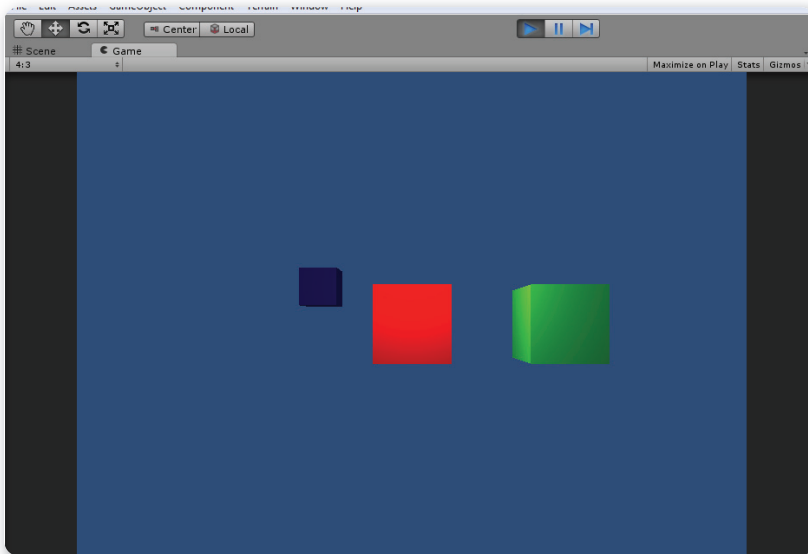
En la siguiente línea llevamos a cabo la asignación; entonces, **position** tiene los valores que hemos colocado vía la variable **posicion**. En este momento podemos probar este nuevo código. Primero debemos eliminar el script anterior del cubo azul y, luego, asignarle este script.



EVITAR LA EJECUCIÓN DE UN SCRIPT



Si lo deseamos, podemos evitar la ejecución de un script de manera temporal. Para hacerlo, vamos al panel **Inspector** del objeto y buscamos su componente de script. Veremos que, junto al nombre, hay una casilla de verificación seleccionada. Al quitar la marca, el script no se ejecutará hasta que volvamos a colocarla.



► **Figura 10.** Esta es la ejecución del segundo script, y ahora el cubo ha tomado la posición asignada por el vector.

Variables privadas

Veamos un poco más sobre la manera de usar las variables. Podemos declarar variables a nivel de función, que solo existen dentro de ella, pero también podemos declarar variables a nivel de clase. Al trabajar con JavaScript, podemos imaginar que cada script es el equivalente a una clase en los lenguajes orientados a objetos.

Crearemos variables privadas a nivel de clase que únicamente pueden ser usadas por la misma clase y son conocidas por todas sus funciones

Creemos un script, al que llamamos **VariablesPrivadas**, con este código:

```
#pragma strict
```

```
// Este programa crea variables privadas y luego las usamos para asignar la  
// posición de un objeto
```

```
// Creamos las variables
```

```
private var posX: float;
private var posY: float;
private var posZ: float;

// Esta funcion se invoca automaticamente y es un buen lugar para hacer
// inicializaciones
function Start () {

    // Asignamos los valores a las variables
    posX=1.5;
    posY=-3.0;
    posZ=-1.0;

    // Colocamos la posicion
    transform.position.x=posX;
    transform.position.y=posY;
    transform.position.z=posZ;

}

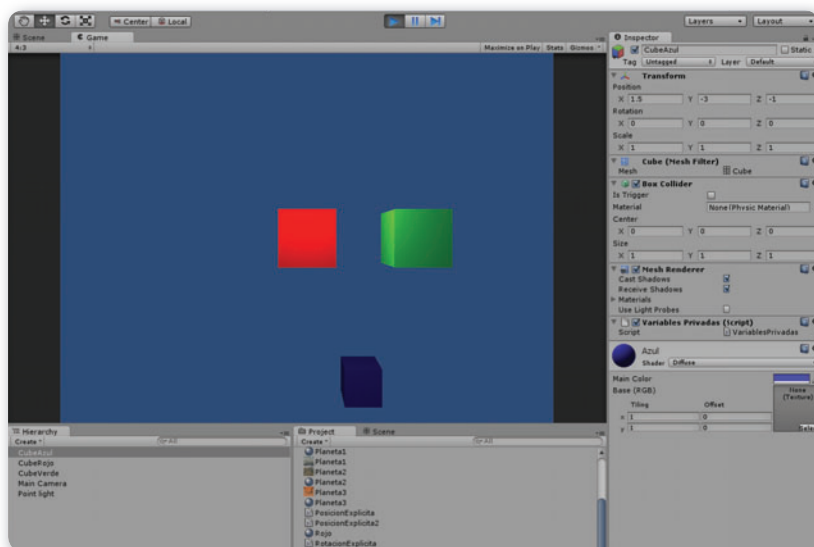
function Update () {

}
```

Comezamos declarando tres variables, una para cada eje coordenado, es decir, **X**, **Y**, **Z**, de la posición. La manera de declararlas es utilizando el modificador de acceso **private**, el cual indica que la variable va a ser privada. Luego usamos **var**, pues estamos definiendo una variable, seguido del nombre de la variable y dos puntos. Finalizamos indicando el tipo, y punto y coma.

Creamos tres variables, las cuales son reconocidas por todas las funciones que se encuentran dentro del script. En la función **Start()** procedemos a asignar los valores y, luego, colocamos la posición haciendo uso de las variables.

Podemos proceder ahora a comprobar el funcionamiento de nuestro programa, pero antes de hacerlo, eliminamos el script anterior del cubo azul y le asignamos este.



► **Figura 11.** Ahora vemos cómo podemos usar variables privadas en nuestro script.

Crear propiedades

Las propiedades proporcionan una gran flexibilidad en el trabajo y en la edición de las escenas. Actúan como variables públicas, y una de las ventajas principales que nos dan, es la integración que presentan con la interfaz del programa.

Cuando creamos una propiedad en nuestro script, Unity automáticamente la adiciona al componente de este en la interfaz. Esto nos permite colocar valores a la propiedad desde el editor, con lo cual podemos probar diferentes opciones sin tener que estar cambiando el código, como sucedería con valores explícitos o colocados en variables privadas.

Veamos un ejemplo en el cual creamos las propiedades necesarias para colocar la posición del cubo desde la interfaz de Unity. Creamos un script llamado **VariablesPropiedades** con el siguiente código:

LAS PROPIEDADES
PROPORCIONAN
FLEXIBILIDAD EN EL
TRABAJO Y EN LA
EDICIÓN DE ESCENAS



```
#pragma strict

// Este programa coloca la posicion dependiendo del valor de las propiedades

// Creamos las propiedades correspondientes
// Damos un valor de inicio

var posX: float = 1.5;
var posY: float = -2.3;
var posZ: float = 1.5;

// Esta funcion se invoca automaticamente y es un buen lugar para hacer
// inicializaciones
function Start () {

    // Colocamos la posicion
    transform.position.x=posX;
    transform.position.y=posY;
    transform.position.z=posZ;
}

function Update () {

}
```

El código no es muy diferente del anterior. Podemos observar que no estamos usando el modificador de acceso **private**, lo que coloca a la variable como pública y crea la propiedad por nosotros. Esta tiene asignado un valor de inicio, que se usa por default, aunque podemos



VALORES FINALES



Las propiedades nos permiten experimentar y refinar los valores de las variables para nuestra aplicación. Cuando tengamos esos valores bien definidos y vayamos a crear la aplicación que se libera al público, lo mejor será colocarlos explícitamente en el código como valores por default.



Mover los cuerpos

El movimiento habitual de los cuerpos se lleva a cabo cuadro por cuadro. Es decir que, en este caso, lo que hacemos es modificar la posición del objeto en cada cuadro de la animación, para provocar así la ilusión de movimiento. De todos modos, existen, básicamente, dos formas de mover los objetos: por cuadro o por tiempo.

Movimiento por cuadro

Para entender este movimiento, vamos a crear un nuevo script, al cual llamamos **MovimientoPorCuadro**, y colocamos el siguiente código:

```
#pragma strict

// Propiedad para la cantidad que deseamos que avance
var avance: float =0.1;

function Start () {

}

// Esta funcion se invoca automaticamente en cada cuadro
function Update () {

    // Se mueve "avance" unidades en el eje x
    // cada vez que se dibuja un cuadro
    transform.Translate(avance,0,0);

}
```

Empezamos por crear una propiedad que se denomina **avance**, y que representa la cantidad de unidades que se va a mover el objeto cada vez que se dibuje el cuadro en el ciclo de animación. Le damos un valor inicial de **0.1**, que es lo suficientemente pequeño como para observarlo sin problemas.

En este caso, no hacemos uso de la función **Start()**, ya que no vamos a inicializar nada. Todo nuestro trabajo se lleva a cabo en la función

Update(). Recordemos que **Update()** es invocada automáticamente cada vez que se va dibujar un cuadro, y lo que coloquemos aquí se ejecutará una y otra vez.

En **Update()** hacemos uso del método **Translate()**, el cual pertenece al componente transform. Este método modifica la posición del objeto en determinado valor, y necesita tres parámetros: el primero es la cantidad que se moverá el objeto en el eje **X**, el segundo es para el eje **Y**, y el tercero indica cuánto se mueve el objeto en el eje **Z**.

En nuestro caso, estamos moviendo la cantidad de **avance** unidades en el eje **X** cada cuadro de animación, y no nos movemos ni en **Y** ni en **Z**, por lo que allí colocamos **0**. Vamos a asignar el script al cubo rojo, el cual se moverá hacia la derecha cuando ejecutemos la aplicación.

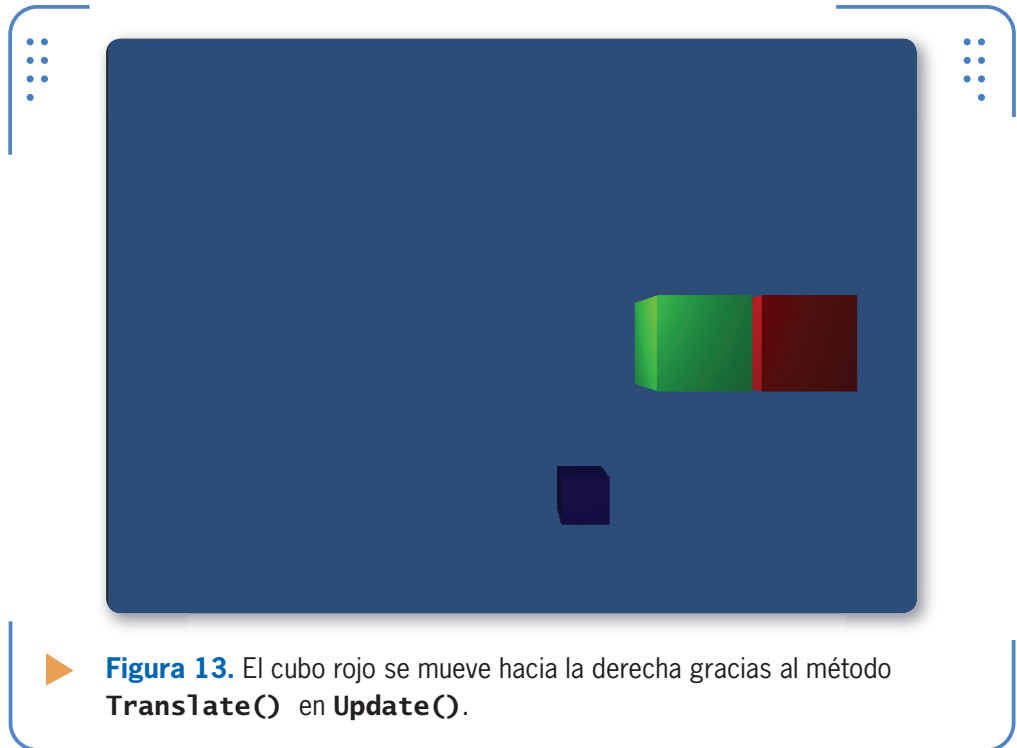


Figura 13. El cubo rojo se mueve hacia la derecha gracias al método **Translate()** en **Update()**.

Con esto ya hemos logrado mover nuestro primer objeto, pero esta no es la mejor forma de hacerlo. Esta técnica mueve el objeto cada vez que se dibuja un cuadro, pero no todas las computadoras dibujan la misma cantidad de cuadros por segundo. Esto quiere decir que, en diferentes

equipos, el cubo se moverá a distintas velocidades, lo cual no es bueno para un juego. Podemos solucionar este problema haciendo uso del movimiento por tiempo.

Movimiento por tiempo

El movimiento por tiempo nos permite corregir el problema mencionado. En vez de tomar el **avance** como la cantidad de unidades que movemos un objeto por cuadro, se toma como un valor de velocidad en unidades por segundo. Unity nos indica el tiempo que ha transcurrido entre un cuadro y otro. Al tener este valor y conocer cuánto debemos avanzar en un segundo, es muy sencillo calcular el avance que corresponde a ese cuadro para cumplir con la velocidad deseada. Con esto, no importa la velocidad de la computadora: todos los objetos se moverán siempre a la misma velocidad.

Veamos cómo programar este parámetro. Creamos un script de nombre **MovimientoPorCuerpo** con el siguiente código:

```
#pragma strict

// Propiedad de la velocidad en unidades por segundo
var velocidad:float =1.0;

function Start () {

}

// Esta funcion se invoca automaticamente en cada cuadro
function Update () {

    // Movemos el objeto en el eje Z a determinada velocidad
    transform.Translate(0,0,velocidad*Time.deltaTime);

}
```

Generamos una propiedad llamada **velocidad**, de tipo **float**, y la inicializamos con valor **1.0**, para que el objeto se mueva a una velocidad de una unidad por segundo. El valor puede cambiarse desde el editor. No efectuamos ninguna inicialización en el método **Start()**.

En el método **Update()** aplicamos el movimiento. En este ejemplo en particular, deseamos que el objeto se mueva en el eje **Z**, pero bien podría haber sido en otro eje o en varios ejes a la vez. Para llevar a cabo la traslación, hacemos uso del método **Translate()**. La cantidad que debemos trasladar el objeto es calculada por la expresión:

```
velocidad*Time.deltaTime
```

En Unity podemos conocer el tiempo que ha transcurrido entre un cuadro y otro del ciclo de animación por medio de **Time.deltaTime**. Este valor está dado en segundos, y si lo multiplicamos por la velocidad en unidades por segundo, obtendremos la cantidad que debe moverse el objeto para ese cuadro.

Ahora tenemos la posibilidad de asignar el script al cubo rojo, sin olvidarnos de eliminar el script anterior. Al momento de ejecutar el programa, seremos capaces de ver el movimiento del cubo que se realiza a la velocidad indicada.

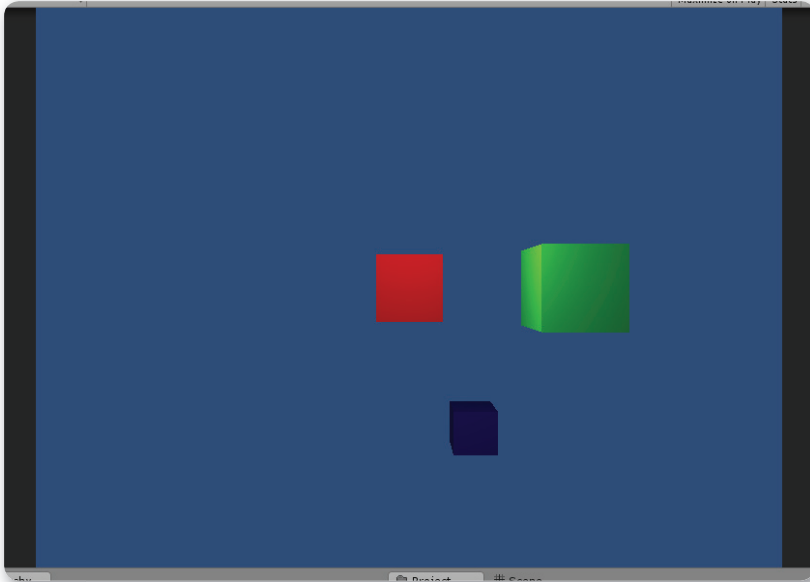


Figura 14. El cubo rojo se mueve en **Z** a una velocidad que es la misma en todas las computadoras.



Escalamiento

El escalamiento nos permite cambiar el tamaño del objeto. Como también pertenece al componente **transform**, su programación es sencilla y similar al caso anterior. Empecemos por crear una nueva escena a la que llamamos **Ejemplo dos**, donde colocamos los mismos elementos que en la anterior: tres cubos, la cámara y una luz.

En este caso, uno de los cubos cambiará su tamaño como si fuera un corazón que late. Creamos un script de nombre **Escalamiento** y le colocamos el siguiente código:

```
#pragma strict

// Este programa cambia el tamaño del objeto

// Variable para la escala
private var escalaActual:float = 1.0f;

// Propiedad para la velocidad de escala
var velEscala:float = 0.75;

function Start () {

}

function Update () {

    // Actualizamos el valor de escala
    escalaActual=escalaActual+(velEscala*Time.deltaTime);

    // Cambiamos la escala, la misma en todos los ejes
    transform.localScale= Vector3(escalaActual,escalaActual,escalaActual);

    // Verificamos si llega a su limite superior
    if(escalaActual>2.0)
    {

        // Hacemos trim al valor
        escalaActual=2.0;
    }
}
```

```
        // Invertimos la direccion del cambio de escala
        velEscala*=-1.0;
    }

    // Verificamos si llega a su limite inferior
    if(escalaActual<0.5)
    {
        // Hacemos trim al valor
        escalaActual=0.5;

        // Invertimos la direccion del cambio de escala
        velEscala*=-1.0;
    }
}
```

Empezamos por generar una variable privada que representa la escala que se tiene en un momento en particular. La inicializamos con el valor 1. Luego, creamos una propiedad llamada **velEscala**, que representa la velocidad con la que cambia la escala. En este caso, la inicializamos con un valor de **0.75**.

En la función **Update()** actualizamos el valor de **escalaActual**. Esta actualización se lleva a cabo basada en el tiempo y depende del valor de **velEscala**. Cuando **velEscala** tenga un valor positivo, **escalaActual** se incrementará; y cuando tenga un valor negativo, decrecerá.

Para colocar la escala que deseamos, tenemos que ingresarla en la propiedad **localScale** de **transform**. Esta propiedad es de tipo **Vector3**, y cada uno de los componentes es el valor de escala en cada uno de los ejes. Como queremos que la escala cambie de la misma manera en todos los ejes, damos el valor **escalaActual** en los tres.

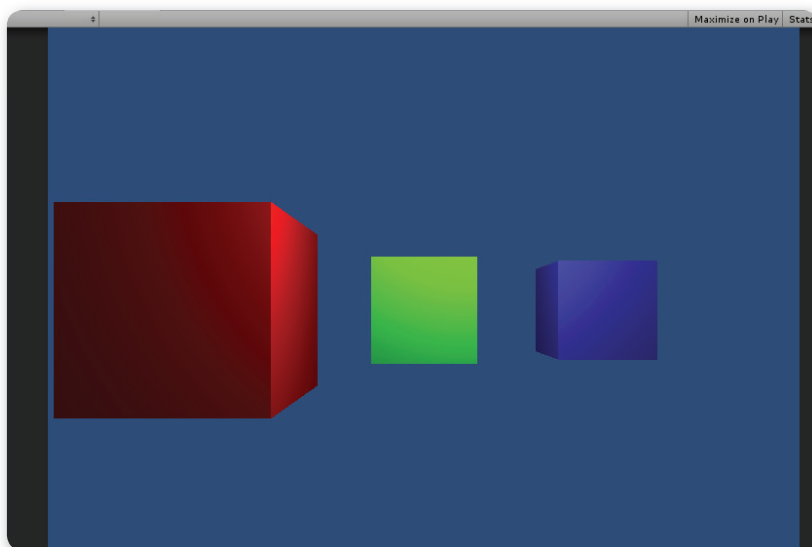


LA ESCALA Y SU COMPORTAMIENTO



Cuando usamos el valor **1**, el objeto no cambia su tamaño. El objeto crece con valores mayores que **1**, y reduce su tamaño con valores menores que **1** y mayores que **0**. Si colocamos **0**, es como si el objeto no tuviera tamaño, y valores negativos lo invierten.

El siguiente paso que podemos ver en el código consiste en programar la lógica que nos permite controlar el cambio de **escalaActual**. Cuando hacemos **trim** a un valor numérico, lo que logramos es que este no pueda exceder cierto límite o ser menor que otro. Si **escalaActual** supera **2.0**, entonces hacemos trim al valor en **2.0** e invertimos el valor de **velEscala** para que disminuya. También verificamos el caso contrario, y cuando **escalaActual** es menor que **0.5**, hacemos trim a **0.5** e invertimos **velEscala** para que incremente. Colocamos el script al cubo rojo y ejecutamos la aplicación.



► **Figura 15.** El cubo crece y se reduce según pasa el tiempo. Parece que palpita como un corazón.



COLOCACIÓN DEL SHADER



Cuando queremos colocar el shader de algún material, vamos al botón que se encuentra junto a su nombre. Al oprimirlo, aparece un menú con los diferentes tipos de shaders y las clasificaciones correspondientes. Simplemente, navegamos hasta el que necesitamos y lo seleccionamos. En el panel **Inspector** veremos las propiedades correspondientes.



La rotación

La rotación nos permite orientar el objeto en el mundo virtual. Podemos rotarlo en cualquiera de los ejes e, incluso, en dos o tres ejes simultáneamente. Empecemos por colocar la orientación de forma explícita, para lo cual creamos un script denominado **RotacionExplicita** y le asignamos el siguiente código:

```
#pragma strict

// Este programa coloca la rotacion de un objeto
// usando sus angulos de Euler por medio de los
// valores de unas propiedades

// Declaramos las propiedades

var rotX:float = 30.0;
var rotY:float = 0.0;
var rotZ:float = 0.0;

function Start () {

    // Colocamos los angulos de Euler
    transform.eulerAngles.x=rotX;
    transform.eulerAngles.y=rotY;
    transform.eulerAngles.z=rotZ;
}

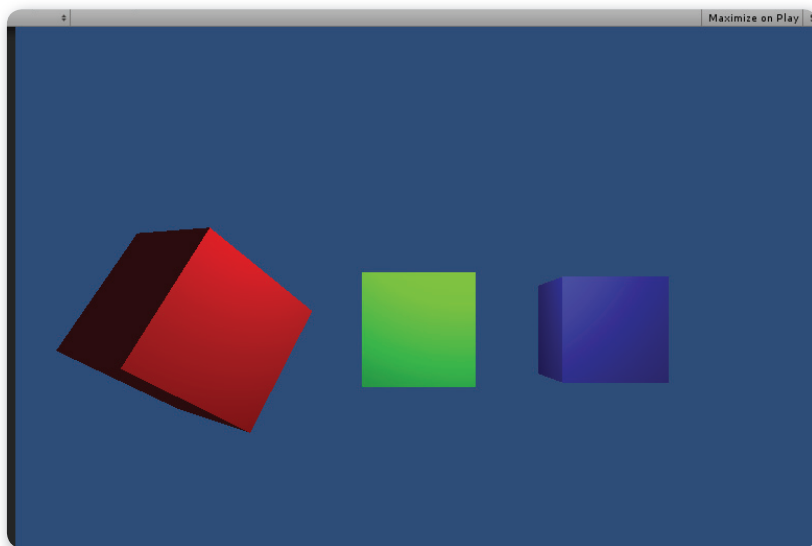
function Update () {

}
```

Iniciamos el programa creando tres propiedades, una para la rotación en cada eje. El valor dado es la rotación en grados del eje correspondiente. Para indicar la rotación, usamos sus **ángulos de Euler** por medio de la propiedad **eulerAngles** y asignando el valor correspondiente para cada uno de ellos.

Múltiples scripts

Ahora colocaremos este script al cubo rojo. Es posible colocar varios a un objeto, pero debemos cuidar de que no entren en conflicto entre ellos. El cubo tiene dos scripts, uno que lo está orientando y otro que lo hace palpar.



► **Figura 16.** El cubo no solamente ha sido rotado por nuestro script, sino que también palpita gracias al otro.

Rotación por tiempo

Esta rotación es muy sencilla: simplemente, usamos el método **Rotate()** del componente **transform**. Este método necesita tres valores que determinan la rotación en grados que vamos a darle al objeto. El primer parámetro es el valor para el eje **X**, el segundo es para el eje **Y**, y el tercero es para el **Z**. No es, como en el ejemplo anterior, el ángulo en que deseamos que se coloque, sino la cantidad de grados que queremos que el objeto rote, a partir de la orientación en que se encuentra. El elemento de rotación por tiempo es similar a la técnica de traslación, haciendo uso de **Time.deltaTime**. Ahora crearemos un script al que llamamos **RotacionPorTiempo** y le colocamos el siguiente código:

```
#pragma strict

// Este programa rota un objeto determinados grados por segundo
// alrededor del eje X, usando el valor de una propiedad

var velAngular:float = 15.0;

function Start () {

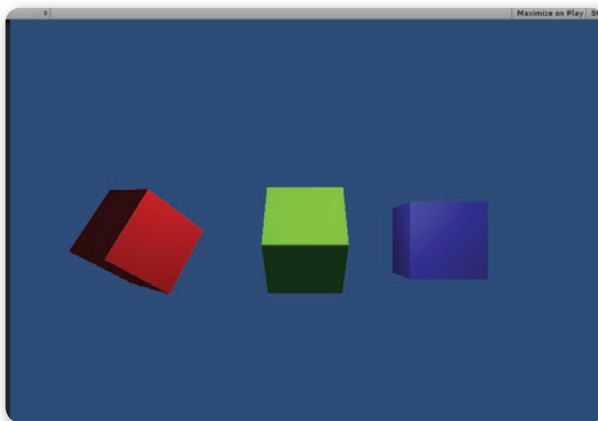
}

function Update () {

    // Rotamos determinados grados por segundo en el eje X
    // No hay rotacion en los otros dos ejes
    transform.Rotate(velAngular*Time.deltaTime,0,0);

}
```

Asignamos este script al cubo verde y ejecutamos la aplicación. El cubo rotará sobre el eje **X**.



► **Figura 17.**
El cubo verde
está rotando
sobre el eje **X**
a la velocidad
asignada.
La rotación es
por tiempo.

Rotación alrededor de un punto

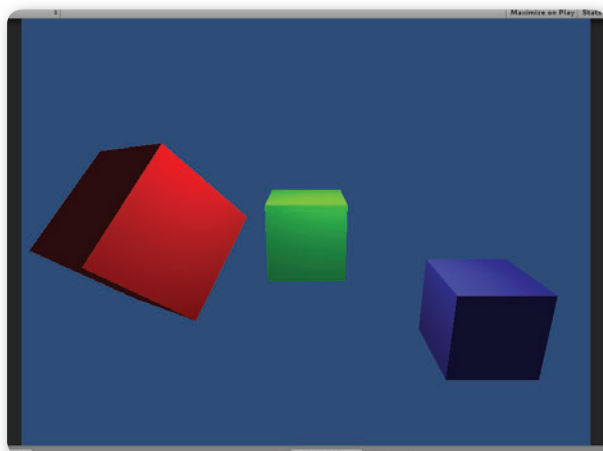
Hemos creado rotación, pero los objetos rotan alrededor de su origen; ahora veremos un truco que podemos aplicar cuando deseamos que el objeto gire alrededor de un punto particular en el espacio. Empezamos por generar un nuevo tipo de objeto usando el menú **GameObject** y luego seleccionando **Create Empty**. Esto crea un objeto vacío, que no tiene ninguna geometría para mostrar, pero que puede llevar a cabo acciones. A este objeto lo llamamos **ObjetoRotacion** y lo colocamos en la posición **(2,0,-6)**.

AL EJECUTAR LA APLICACIÓN, VEREMOS QUE EL CUBO AZUL GIRA ALREDEDOR DE SU PADRE



A continuación, dentro del panel **Hierarchy** arrastramos el cubo azul sobre **ObjetoRotacion**. Esto hace que el cubo quede como **hijo** de **ObjetoRotacion**, acción conocida como **parenting**. Si hacemos clic en el triángulo junto al nombre **ObjetoRotacion**, podemos ver y ocultar los objetos de los cuales es **padre**. Seleccionamos el cubo azul y verificamos que su posición sea **(0,0,1)**. Esta posición es ahora relativa a su padre, no al origen del mundo.

Asignamos el script **RotacionPorTiempo** a **ObjetoRotación**. No hay que asignarlo al cubo azul sino a su padre. Al ejecutar la aplicación, veremos que el cubo azul gira alrededor de su padre.

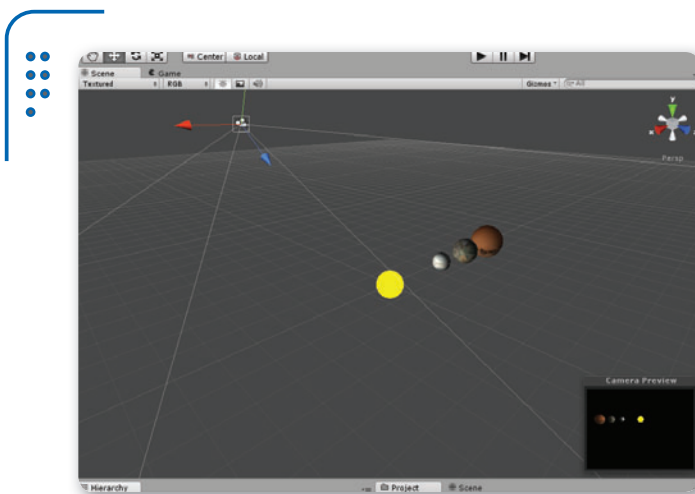


► **Figura 18.** Podemos ver que el cubo azul gira alrededor de su padre.



Proyecto de sistema solar

Realicemos ahora un proyecto integrador para aprender algunas cosas más. La idea es lograr una especie de sistema solar con planetas que giran alrededor de un sol. Para comenzar, creamos una nueva escena para nuestro proyecto actual, llamada **Ejemplo3**, a la que vamos a colocarle varios objetos con las siguientes características. Empezamos por poner la estrella como un objeto de tipo esfera en la posición **(0,0,0)** y creamos para ella un material llamado **AmarilloEstrella**. Este tendrá un shader de tipo **Self-Illumin / Bumped Diffuse**, y un color amarillo. A continuación, creamos otro objeto de tipo esfera al cual nombramos **Planeta1** y lo colocamos en **(-3,0,0)**; para este planeta indicamos una escala de **0.75** en cada uno de los ejes. El siguiente objeto también es una esfera, en este caso, **Planeta2**, que ubicamos en **(-4.7,0,0)**. La escala es de **1.1** en todos los ejes. El último planeta es una esfera que se llama **Planeta3** y se encuentra en **(-6.5,0,0)**, con escala de 1.5 en todos los ejes. Luego adicionamos una luz de tipo **PointLight**, que colocamos en la posición **(0,0,0)**, con el fin de que se emita desde la misma posición en que está la esfera que representa a la estrella. Vamos a modificar unos parámetros de la luz: el valor del rango es **14.5** y el de la intensidad es de **1.2**. La cámara se coloca en **(0,5,-10)** con una rotación en **X** de **35**. Ya tenemos listos los elementos básicos que componen la escena.



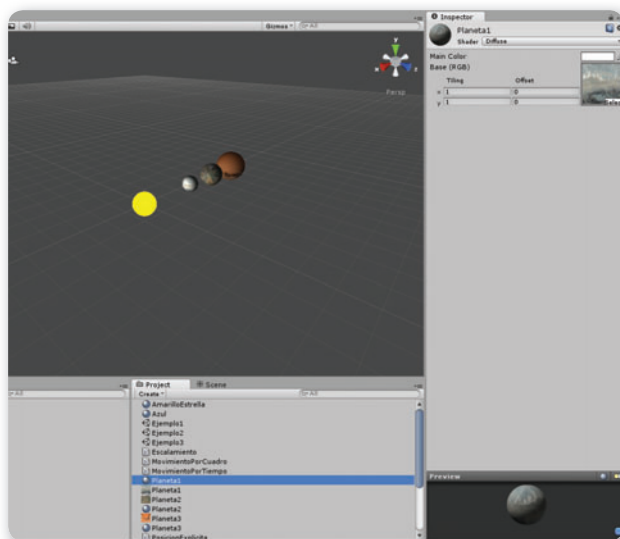
► **Figura 19.**

Así luce nuestra escena construida con los objetos que representan a los planetas y al sol.

Adicionar texturas

El siguiente paso es crear materiales que usen texturas; esto significa que podemos utilizar una imagen para que el objeto se dibuje con ella en su superficie. Primero debemos conseguir tres imágenes para las superficies de los planetas.

Una vez que tenemos las imágenes, simplemente las arrastramos desde el Escritorio o la carpeta correspondiente al panel **Project**; así serán importadas a nuestro proyecto. Luego creamos un material y le ponemos como nombre **Planeta1**. El tipo de shader que vamos a utilizar es **Diffuse**. Para colocar la textura, arrastramos la imagen que deseamos para el primer planeta desde el panel **Project** hasta el cuadro



► **Figura 20.** Creamos un material que hace uso de una textura para dibujar la superficie del objeto.



TEXTURAS PARA NUESTROS PLANETAS

El siguiente sitio web contiene una colección de imágenes gratuitas para descargar y utilizar en nuestros proyectos; podemos aplicarlas en este caso para las superficies de los planetas: **www.freebitmaps.blogspot.mx**.

que se encuentra en las propiedades del shader. Luego, le asignamos el material a la esfera que representa nuestro primer planeta. Repetimos el procedimiento para los otros dos planetas usando diferentes imágenes.

Agregar el código

Todos los planetas van a utilizar los mismos scripts: uno para la rotación y otro para el movimiento de traslación. Creamos el primer script con el nombre **RotacionPlaneta** y el siguiente código:

```
#pragma strict

// Este programa rota al planeta alrededor de su propio eje
// Usamos valores aleatorios para la velocidad de rotacion

// Variable para la velocidad angular
private var velAngular:float;

function Start () {

    // Se selecciona aleatoriamente un valor
    velAngular = Random.Range(-45,45);

}

function Update () {

    // Rotamos alrededor del eje Y
    transform.Rotate(0,velAngular*Time.deltaTime,0);

}
```

Este código es fácil de entender, lo único nuevo es el uso de **Random**, que nos permite obtener un valor aleatorio comprendido entre aquellos que pasamos como parámetros. Con esto, cada vez que ejecutemos la aplicación, obtendremos una velocidad angular aleatoria entre **-45** y **45**. Colocamos el script a todos los planetas.

El siguiente script que vamos a crear se llama **TraslacionPlaneta** y lleva el siguiente código:

```
#pragma strict

// Este programa lleva a cabo la traslacion del planeta
// alrededor del origen (estrella) usando una velocidad
// aleatoria

// Variable para la velocidad angular
private var velAngular:float;

function Start () {

    // Colocamos un valor aleatorio
    velAngular = Random.Range(-80,80);

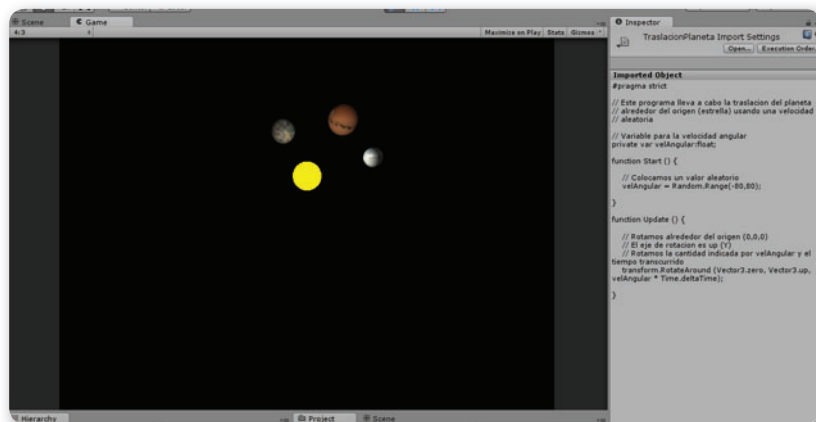
}

function Update () {

    // Rotamos alrededor del origen (0,0,0)
    // El eje de rotacion es up (Y)
    // Rotamos la cantidad indicada por velAngular y el tiempo transcurrido
    transform.RotateAround (Vector3.zero, Vector3.up, velAngular
        * Time.deltaTime);

}
```

Para realizar la rotación, usamos **RotateAround()**. El primer parámetro es una coordenada por donde pasa el eje de rotación. **Vector3.zero** es igual a **(0,0,0)**, es decir, el origen en este caso. El segundo parámetro es un vector que indica la pendiente del eje de rotación, **Vector3.up**, y equivale al eje **Y**, que se dirige hacia arriba. El último parámetro es la cantidad de grados por rotar. Ahora procedemos a asignar este script a todos los planetas, por lo que cada uno tendrá dos scripts. Si ejecutamos la aplicación, veremos el sistema solar en funcionamiento. En cada oportunidad que lo ejecutemos, los planetas se moverán de una manera diferente.



► **Figura 21.** El sistema solar se mueve con los planetas rotando y girando alrededor de la estrella.

En este capítulo hemos recorrido mucho territorio y ya conocemos cómo mover los objetos por medio del código. Ya podemos empezar a realizar acciones más complejas.



RESUMEN



Los scripts nos permiten proveer a los objetos de la lógica necesaria para nuestra aplicación. Tienen funciones preconstruidas que podemos usar para inicializar o llevar a cabo nuestra lógica. Podemos colocar los valores de forma explícita, por medio de variables o propiedades. Las variables privadas únicamente son vistas por el objeto, en tanto que las públicas actúan como propiedades. La propiedad **position** indica la posición del objeto. El método **Translate** mueve el objeto en el eje que le indiquemos. Podemos mover los cuerpos por cuadro o por tiempo. Cuando trabajamos por tiempo, usamos **Time.deltaTime**. La orientación del objeto es colocada por medio de sus **ángulos de Euler**. El método **Rotate()** rota al objeto determinada cantidad de grados. Es posible crear materiales que usen texturas.

Actividades

TEST DE AUTOEVALUACIÓN

- 1 ¿Qué es un script?
- 2 ¿Cómo creamos un script?
- 3 ¿Cómo asignamos un script?
- 4 ¿Cómo funciona una propiedad?
- 5 ¿Cuál es la sintaxis para declarar variables privadas?
- 6 ¿Cuál es la sintaxis para crear una propiedad?
- 7 ¿De qué maneras podemos colocar la posición de los objetos?
- 8 ¿De qué formas podemos colocar la orientación de los objetos?
- 9 ¿En qué propiedad se guarda la información de escala?
- 10 ¿Cómo adicionamos texturas?
- 11 ¿Cómo se trabaja basándose en tiempo?
- 12 ¿Dónde obtenemos la información del tiempo transcurrido entre cuadros?

ACTIVIDADES PRÁCTICAS

- 1 Adicione más planetas al programa.
- 2 Modifique el programa para que el tamaño del planeta también cambie.
- 3 Coloque una luna que gire alrededor de un planeta.
- 4 Haga que la cámara gire alrededor de la estrella.
- 5 Coloque un fondo con textura del espacio a la aplicación.



Control de entradas por el usuario

Hemos colocado el código que nos permite controlar la posición y orientación de los objetos; sin embargo, para programar videojuegos será necesario tener la capacidad de leer las entradas del usuario. Durante el desarrollo de este capítulo aprenderemos a capturar sus entradas, lo que permitirá la interacción del jugador con la aplicación, y enriquecerá el nivel de complejidad y el dinamismo del videojuego.

▼ El sistema de	
entradas de Unity	72
Capturar la entrada del teclado.....	75
Capturar la entrada del mouse.....	86
Leer otros dispositivos	90

▼ Resumen.....	95
▼ Actividades.....	96



El sistema de entradas de Unity

Unity tiene un sistema para capturar las entradas del usuario que es muy fácil de utilizar. Esta sencillez se debe a una cuidadosa planificación por parte de los desarrolladores, que reduce mucho los problemas que se presentan cuando se desea leer diferentes dispositivos.

Para leer las entradas del usuario, usamos, principalmente, una clase llamada **Input**. Dentro de ella encontramos una colección de variables y métodos que nos permiten tener esta información. Hay métodos para leer el teclado, el mouse, los controladores de juego e, incluso, la información de dispositivos táctiles y que contengan acelerómetros, como tablets o iPhone.

PODEMOS USAR
NUESTRA APLICACIÓN
CON CUALQUIER
TIPO DE DISPOSITIVO
DE ENTRADA

Gracias al diseño de Unity, podemos usar nuestra aplicación con cualquier tipo de dispositivo de entrada, incluso, con aquellos que no han sido creados, ya que en vez de tener funciones especiales para cada dispositivo o controlador, se abstraen los controles y se trabaja con conceptos como **Disparo1**, **Disparo2**, etc. De esta manera, programamos con el concepto y, luego, simplemente indicamos qué significa **Disparo1** o **Disparo2** en términos de control. Esto

también reduce el tiempo de desarrollo requerido, ya que no es preciso tener un código específico para la PC y otro para el **iPad**; simplemente, redefinimos **Disparo1** o **Disparo2** según el dispositivo de juego, y así evitamos cambiar las líneas de código. Dentro del editor de Unity, hay

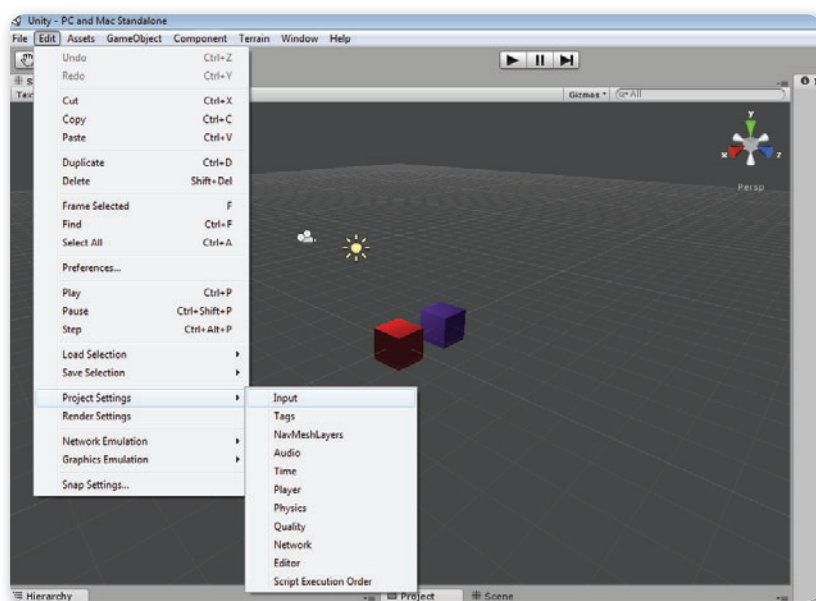


LOS ACELERÓMETROS



Los acelerómetros permiten medir la aceleración de un cuerpo. Muchos dispositivos actuales los utilizan y, dependiendo de la cantidad de acelerómetros, podremos medir el movimiento en uno, en dos o en los tres ejes. Por lo general, los acelerómetros de los dispositivos son sencillos, pero tienen suficiente sensibilidad como para poder usarlos como dispositivos de entrada en los videojuegos.

un lugar para llevar a cabo estas configuraciones: nos dirigimos al menú **Edit** y seleccionamos **Project Settings**; entre las opciones que se presentan, elegimos **Input**.



► **Figura 1.** El menú **Input** nos lleva a la configuración de los dispositivos de entrada que podemos usar con Unity.

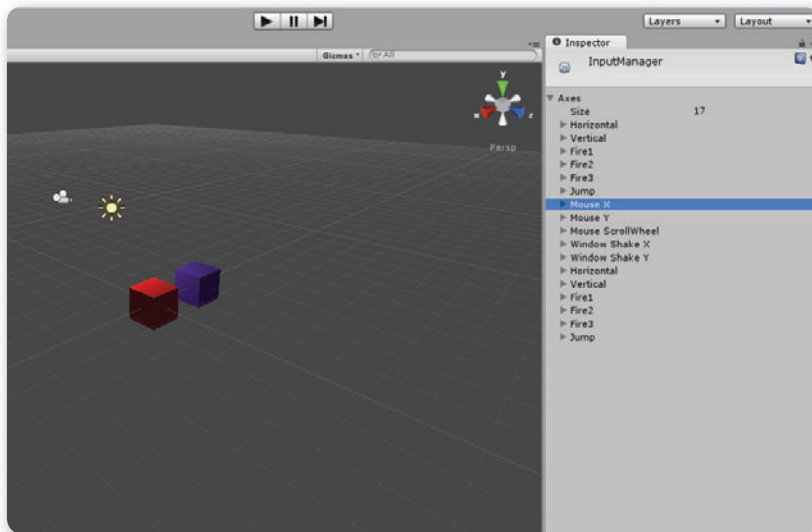
En el panel **Inspector** aparece una lista con los conceptos que podemos utilizar para programar nuestras aplicaciones. Estos son los más importantes y comunes referidos a los diferentes controles que podemos usar en los juegos.



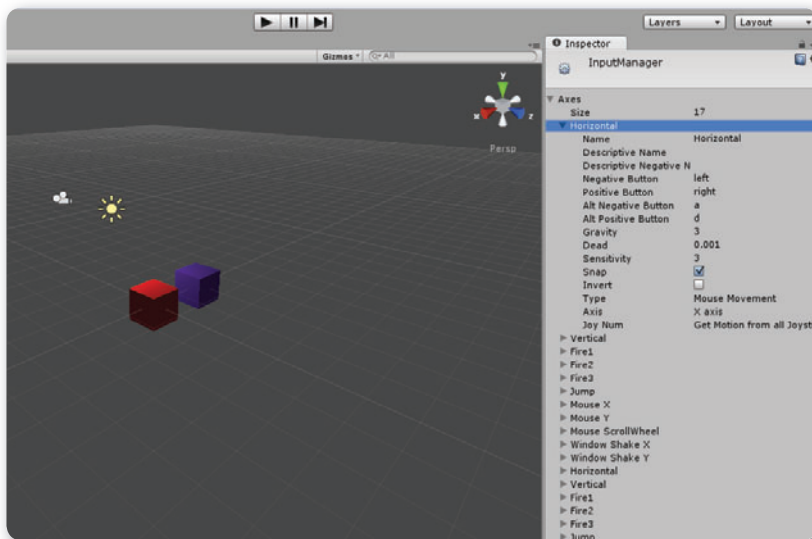
INVESTIGAR INPUT



Es recomendable investigar todos los conceptos y sus propiedades, porque esto nos permite saber qué podemos capturar de un control de juego y qué valores es posible colocar. Algunas propiedades tienen una lista de opciones que resulta interesante conocer. Cuanto más conozcamos, mayor flexibilidad y facilidad tendremos al momento de desarrollar la aplicación.



► **Figura 2.** Esta lista muestra los conceptos más importantes y comunes relacionados con los controles de juego.



► **Figura 3.** Podemos configurar el concepto para determinado dispositivo con sus propiedades.

Si expandimos uno de estos conceptos, veremos muchas propiedades que nos permitirán llevar a cabo su configuración. Aquí expresamos qué debe tomar del dispositivo controlador del juego para quedar referenciado a ese concepto.

Capturar la entrada del teclado

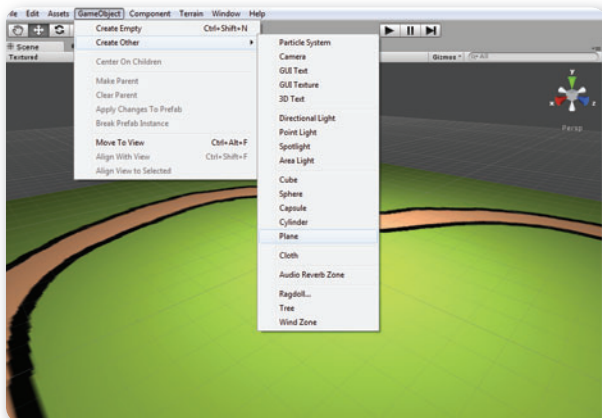
Empecemos con lo más fácil, que es capturar la entrada del teclado. Vamos a crear un pequeño juego en el que, por medio del teclado, movamos un cubo. Una vez capturadas las entradas del teclado, podemos incluir el control del juego por parte del usuario. Es posible trabajar con cualquier tecla, ya sean letras o números.

El juego consiste en una pista que debe ser recorrida por un cubo controlado por el usuario. Comencemos por crear un proyecto nuevo con una escena llamada **Ejemplo1**. La pista será una textura colocada sobre un plano, que creamos con cualquier programa de dibujo. Debe ser similar a la que se muestra en la **Figura 4**.



► **Figura 4.** Este es un ejemplo de la textura que representa la pista por donde pasará el cubo.

Ahora empezamos a crear el mundo. Vamos a utilizar un objeto de tipo plano, un cuadro que podemos colocar en nuestra escena. Para agregarlo, nos dirigimos al menú **Game Object** y seleccionamos **Other**; se abre un menú con los objetos que podemos adicionar, en donde seleccionamos **Plane**.



► **Figura 5.** El primer paso es adicionar un plano donde colocaremos la textura de la pista.

A este plano lo llamamos **Piso** y lo colocamos en la posición **(0,0,0)**. Como inicialmente no es muy grande, lo escalamos con el valor **3** en todos los ejes; ahora sí hay suficiente espacio para que el cubo pueda recorrer. Con lo que hemos aprendido en el capítulo anterior, adicionamos una textura llamada **Pista** con la imagen de la pista que hemos creado y, posteriormente, un material llamado **materialPista** que tiene la textura **Pista**, y es asignado al plano **Piso**.

Es necesario poner una luz, por lo que colocamos una de tipo **Pointlight** en la posición **(0,8,-2)**. Para que ilumine adecuadamente el piso,



LA PROPIEDAD RANGE



La propiedad **Range** en la luz indica la distancia máxima a la que llega la luz para procesar la iluminación. Al cambiar el valor, podemos observar en la escena que un mayor volumen puede ser iluminado por la luz. Con esta propiedad, es posible tener un mayor control sobre las partes de la escena que deseamos ver iluminadas por determinada fuente de luz.

debemos indicar el valor **60** en la propiedad **Range** y **0.8** en la propiedad **Intensity**. Con esto, podemos observar correctamente el piso.

Una vez que el piso está listo, vamos a colocar el cubo que será controlado por el usuario. Lo agregamos de la forma usual, y lo ubicamos en **(0,1,0)**. Creamos un material y se lo asignamos. El color del cubo puede ser nuestro tono favorito o algún otro que deseemos poner.

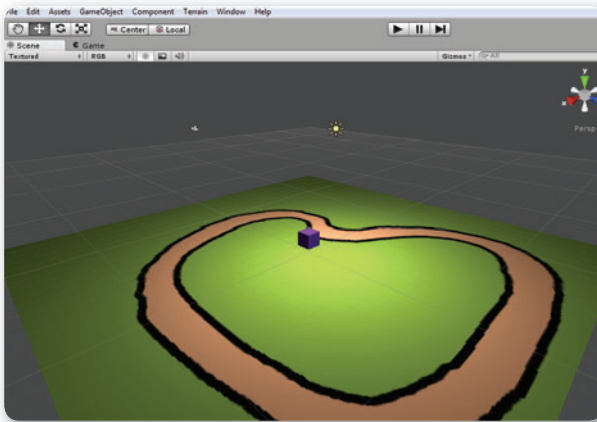


Figura 6. La escena completa luce de esta manera; ahora es necesario empezar a adicionar la lógica de control.

Empecemos por desarrollar nuestra primera versión de control. Esto nos enseña las bases sobre las cuales vamos a crear mejores maneras para interactuar con el cubo. Para hacerlo, comenzamos por generar un script de tipo JavaScript y le ponemos como nombre **DetectarTecla**. En él ponemos el siguiente código:

```
#pragma strict

function Start () {

}

// Con esta funcion controlamos el movimiento del objeto
// por medio de las teclas de flechas
// Solo se lleva a cabo la accion en el momento en que se oprime la tecla
function Update () {
```

```
// Verificamos si se ha oprimido la tecla flecha derecha
if(Input.GetKeyDown(KeyCode.RightArrow))
    transform.Rotate(0,5,0);

// Verificamos si se ha oprimido la tecla flecha izquierda
if(Input.GetKeyDown(KeyCode.LeftArrow))
    transform.Rotate(0,-5,0);

// Verificamos si se ha oprimido la tecla flecha arriba
if(Input.GetKeyDown(KeyCode.UpArrow))
    transform.Translate(0,0,1);

}
```

Ponemos el código para la detección del teclado en la función **Update**, ya que esta se invoca en cada cuadro de animación en el videojuego. Hacemos uso de la clase **Input**, dedicada a obtener las entradas de los diferentes dispositivos. El método que vamos a usar en este caso es **GetKeyDown()**, que regresa un valor **bool**, es decir, **true** o **false**, según se oprima determinada tecla o no. Un punto importante para notar es que el método regresa el valor **true** solo en el momento en que se pulsa la tecla; pero si esta continúa presionada, ya no lo hace. Recibimos el valor **true** solo una vez en todo el proceso de oprimir la tecla. Para indicar cuál es la tecla que nos interesa saber si se pulsó o no, usamos el parámetro del método.

Como lo que realmente deseamos es ejecutar un código cuando se haya oprimido una tecla determinada y recibamos un valor **bool**, nos conviene usar **if**. En la expresión por evaluar del **if** verificaremos si la tecla en cuestión se oprimió. Si observamos el código, veremos que, en



LISTADO DE TECLAS



Todos los posibles valores de teclas que podemos usar en los métodos de **Input** relacionados con el teclado se encuentran listados en **KeyCode**. Simplemente, debemos buscar cuál es la tecla que nos interesa. Encontraremos las letras, los números e, incluso, teclas especiales, como las flechas y más. En la documentación de Unity podemos consultar más información al respecto.

primer lugar, comprobamos si se presionó la tecla **flecha derecha**. Para esto, hacemos uso de **GetKeyDown**, y aplicamos como parámetro **KeyCode.RightArrow** para indicar que nos interesa verificar la tecla **flecha derecha**. Si se oprime la tecla, el valor regresado es **true** y se cumple la condición del **if**, por lo que se ejecuta la siguiente línea. Lo que deseamos que se realice es una pequeña rotación de **5** grados sobre el eje **Y**.

Podemos hacer la verificación de tantas teclas como sea necesario para nuestra aplicación. A continuación, procedemos a verificar si se ha oprimido la tecla **flecha izquierda**. Para hacerlo, usamos **KeyCode.LeftArrow**, que es el código de dicha tecla. Lo que deseamos que suceda cuando se oprime esta tecla es una rotación, pero en el sentido contrario, por eso colocamos el valor de **-5** para el eje **Y**. Esto debe permitir la rotación del cubo cuando se oprimen las teclas.

También queremos que el cubo se mueva. El movimiento se lleva a cabo cuando oprimimos la tecla **flecha arriba**. En este caso, colocamos otro **if** y usamos **GetKeyDown**. El código para la tecla **flecha arriba** es **KeyCode.UpArrow** y se coloca como parámetro. Si esto se cumple, en vez de rotar, haremos avanzar el objeto en **Z**, utilizando **Translate** con valor **1**.

RECIBIMOS EL
VALOR TRUE
SOLAMENTE EN EL
MOMENTO EN QUE
SE PULSA LA TECLA



Coordenadas de mundo vs. coordenadas de objeto

Cuando trabajamos en 3D, existen diferentes tipos de sistemas de coordenadas, y es importante tener una noción sobre ellos para comprender el comportamiento de los objetos. El primer sistema que encontramos se conoce como coordenadas de mundo, y es el



EL USO DE GETMOUSEBUTTONDOWN

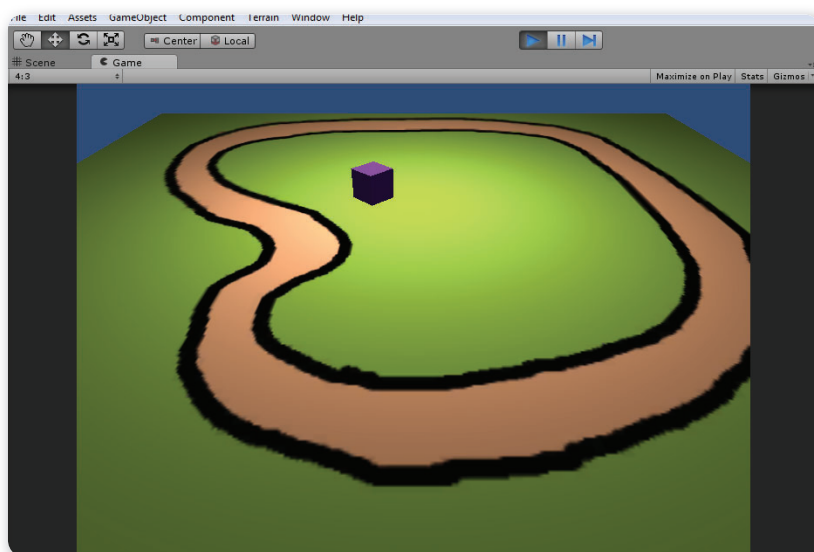


Es posible detectar cuando se oprime algún botón del mouse. Para hacerlo, usamos el método **GetMouseButtonDown**, que regresa el valor **true** en el instante en el cual se oprime el botón. Si el botón se mantiene presionado, los siguientes valores serán **false**. Este método necesita un parámetro que le indique cuál es el botón que nos interesa leer.

de la escena. Las posiciones de los objetos se definen relativas a su origen; es decir que cuando colocamos un objeto en la escena, sus coordenadas son relativas al origen del mundo.

A su vez, cada uno de los objetos que tenemos tiene su propio sistema de coordenadas, que se conoce como coordenadas de objeto. La geometría del objeto es definida en relación a estas coordenadas. Al hacer uso de estos sistemas de coordenadas, podemos trabajar con los objetos. La rotación que hemos visto se efectúa sobre el origen de dichas coordenadas. Esto rota los ejes locales, de tal manera que si hacemos uso de **Translate** sobre un objeto rotado, lo hará en la dirección de los ejes rotados, y no, en la de los ejes del mundo. Este efecto ocurrirá en nuestro ejemplo y nos permitirá guiar el objeto de una manera muy sencilla y natural.

Ahora podemos asignar el script al cubo y ejecutar la aplicación. Al oprimir las teclas **flecha derecha** y **flecha izquierda**, el cubo rota un poco. Si mantenemos oprimida la tecla, el cubo no sigue girando. Un aspecto que debemos observar es hacia dónde avanza el cubo cada vez que pulsamos la **flecha arriba**. El cubo se mueve en la dirección hacia donde



► **Figura 7.** Ya podemos controlar el cubo y hacerlo recorrer la pista por medio de las teclas de flechas.

esté apuntado, debido a la rotación. Podemos tratar de hacerlo recorrer la pista completa. El movimiento es un poco tosco, pero se puede mejorar cambiando los valores de rotación y avance en el código.

El control que tenemos sobre el cubo es muy primitivo, pero al menos funciona. Ahora necesitamos mejorar la interacción, pues no resulta muy cómodo tener que oprimir continuamente las teclas, ni nos da mucha precisión. Por lo tanto, vamos a utilizar otro método que nos permitirá interactuar de un modo diferente.

Empezamos por eliminar el script anterior del cubo y crear uno nuevo, al cual llamamos **DetectaTeclaOprimida**. El código que colocamos en él es el siguiente:

```
#pragma strict

// Propiedades necesarias
// Velocidad del objeto
var velocidad:float =1.0;

// Velocidad de rotacion
var velAngular:float =45.0;

function Start () {

}

// Con esta funcion controlamos el movimiento del objeto
// por medio de las teclas de flechas
// La accion se lleva a cabo todo el tiempo que la tecla este
// oprimida
function Update () {
```



EL ORDEN DE LAS TRANSFORMACIONES



Un punto importante que debemos conocer es que el orden en el que llevamos a cabo las diferentes operaciones de **Transform** nos da distintos resultados. No obtenemos el mismo efecto si primero rotamos y luego trasladamos, que si primero trasladamos y después rotamos. Es interesante experimentar estas opciones.

```
// Controlamos la orientacion

// Verificamos si se ha oprimido la tecla flecha derecha
if(Input.GetKey(KeyCode.RightArrow))
    transform.Rotate(0,velAngular*Time.deltaTime,0);

// Verificamos si se ha oprimido la tecla flecha izquierda
if(Input.GetKey(KeyCode.LeftArrow))
    transform.Rotate(0,-velAngular*Time.deltaTime,0);

// Verificamos si se ha oprimido la tecla flecha arriba
if(Input.GetKey(KeyCode.UpArrow))
    transform.Translate(0,0,velocidad*Time.deltaTime);
}
```

Vamos a crear algunas propiedades que nos sirvan para colocar el valor de la velocidad con la que deseamos que el cubo se mueva. Esta velocidad está dada en unidades por segundo y le damos un valor inicial de **1.0**; la llamamos **velocidad** y es de tipo **float**, utilizada cuando deseamos guardar valores que tienen punto decimal. Luego definimos otra propiedad denominada **velAngular**, que se encarga de guardar la velocidad de rotación del cubo. El valor que le damos es de **45.0**, y sus unidades se miden en grados por segundo; también es de tipo **float**.

Dentro del método **Update** llevamos a cabo los cambios necesarios. En el ejemplo anterior, únicamente cuando se oprimía la tecla **flecha abajo** se realizaba una determinada acción. Ahora queremos que esa acción se efectúe todo el tiempo durante el cual la tecla esté oprimida. Para lograrlo, hacemos uso del método **GetKey**, perteneciente a la clase **Input**.



AYUDA CON SCRIPTS



Unity provee de información muy útil para conocer la sintaxis de los comandos e instrucciones que podemos utilizar en nuestros scripts. Para acceder a ella, simplemente seleccionamos el menú **Help** y luego **Scripting Reference**. Esta acción abre una página web desde donde podemos buscar el comando que deseamos, ver su sintaxis y, en muchos casos, consultar ejemplos de uso.

Este método regresa un valor **true** o **false**. Al invocarlo, debemos colocar como parámetro el código de la tecla que deseamos verificar si está oprimida. Si está siendo presionada en el momento en que se invoca el método, entonces regresa el valor **true**; en caso contrario, devuelve **false**. Podemos indicar la tecla de parámetro por medio de **KeyCode**.

En el script, lo primero que hacemos es verificar si se está oprimiendo la tecla **flecha derecha**; en caso de que esto ocurra, el **if** se cumple, por lo que se ejecuta la línea que está inmediatamente abajo. En este ejemplo, realizamos una rotación en el eje **Y**, cuyo valor está dado por la velocidad angular que hemos colocado y el tiempo transcurrido.

Algo similar ocurre a continuación. Verificamos si el usuario está manteniendo oprimida la tecla **flecha izquierda**. Si el **if** se cumple, procedemos a hacer una rotación. Cabe notar que, en este caso, el valor de la rotación es negado, pues deseamos que se gire en sentido contrario al anterior. También es una rotación en el eje **Y**.

Ya tenemos el control de rotación para el cubo, pero deseamos que este pueda avanzar hacia donde queramos, de manera que debemos hacer la programación correspondiente. Para el avance del cubo vamos a utilizar la tecla **flecha arriba**. En este caso, empleamos **KeyCode.UpArrow** como parámetro de **GetKey()**. Si se cumple el **if**, entonces procedemos a trasladar el cubo. La traslación se lleva a cabo sobre el eje **Z**, y la cantidad está dictada por la velocidad y el tiempo transcurrido.

Ya contamos con los controles necesarios para mover al cubo. El desplazamiento será más natural y fluido que en el caso del primer ejemplo. Solo resta asignar el script al cubo y ejecutar la aplicación.

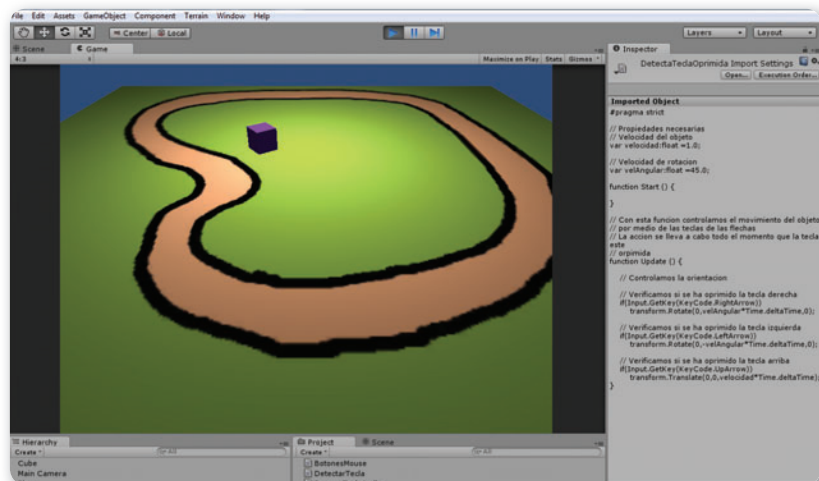
EL DESPLAZAMIENTO
SERÁ MÁS NATURAL
Y FLUIDO QUE
EN EL CASO DEL
PRIMER EJEMPLO



NÚMEROS EN KEYCODE



Cuando nos interesa hacer uso de teclas que representan números, tenemos dos opciones, ya que hay números en el teclado principal y en el teclado numérico. Las teclas de números del teclado principal empiezan con Alpha: por ejemplo, el **0** es **Alpha0**, y el **3** es **Alpha3**. Las del teclado numérico empiezan con Keypad, como **Keypad0** y **Keypad3**.



► **Figura 8.** Ahora el control del cubo es más suave y natural que en el caso anterior.

PODEMOS APOYARNOS
EN EL IF PARA LLEVAR
A CABO UN TRABAJO
CUANDO SUCEDA LO
QUE ESPERAMOS



Para terminar con el control por medio del teclado, nos hace falta ver un método que también resulta útil. En el ejemplo particular que estamos tratando, no es muy recomendable su uso, pero ver su comportamiento en acción nos permitirá entenderlo mejor. Se lo

conoce como **GetKeyUp()** y funciona de un modo ligeramente diferente del de los otros dos. El método va a regresar un valor de tipo **bool**, ya sea **true** o **false**, y al igual que los anteriores, necesita un parámetro que indique cuál es la tecla que deseamos verificar. Devuelve **true** cuando la tecla, estando previamente oprimida, se ha liberado; es decir que, cuando dejamos de pulsar la tecla, podemos llevar a cabo una acción. Si la tecla continúa presionada o nunca se pulsó, entonces el valor que obtenemos es **false**.

Al igual que en los ejemplos anteriores, podemos apoyarnos en el **if** para llevar a cabo un determinado trabajo cuando suceda lo que esperamos. Empecemos por eliminar del cubo el script anterior, y creemos uno nuevo, al cual llamaremos **DetectaTeclaArriba**, con el siguiente código:

```
#pragma strict

function Start () {

}

// Con esta funcion controlamos el movimiento del objeto
// por medio de las teclas de flechas
// Solo se lleva a cabo la accion en el momento que se suelta la tecla
function Update () {

    // Verificamos si se ha soltado la tecla flecha derecha
    if(Input.GetKeyUp(KeyCode.RightArrow))
        transform.Rotate(0,5,0);

    // Verificamos si se ha soltado la tecla flecha izquierda
    if(Input.GetKeyUp(KeyCode.LeftArrow))
        transform.Rotate(0,-5,0);

    // Verificamos si se ha soltado la tecla flecha arriba
    if(Input.GetKeyUp(KeyCode.UpArrow))
        transform.Translate(0,0,1);

}
```

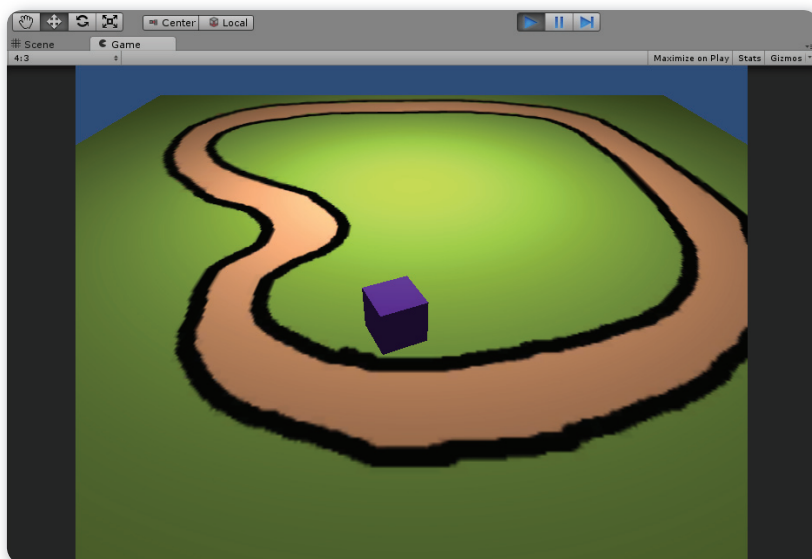
Para este ejemplo, nos basamos en la primera versión del script y efectuamos algunos cambios. Como únicamente deseamos mostrar la funcionalidad del método **GetKeyUp**, no colocamos ninguna propiedad ni variable, pero si preferimos, podemos modificar el script para utilizarlas. Colocamos el código de la forma habitual en el método **Update()**, para garantizar la verificación del estado de la tecla en cada cuadro en que se dibuje el juego.

Empezamos por usar un **if**, en cuyo interior colocamos la invocación al método **GetKeyUp()** de la clase **Input**. Primero verificamos la tecla **flecha derecha**, pasando como parámetro **KeyCode.RightArrow**. Si el **if** se cumple, el objeto rota 5 grados en el eje **Y**. No debemos olvidar que la rotación se lleva a cabo cuando dejamos de oprimir o liberamos la tecla.

Trabajamos de forma similar para verificar si el usuario ha liberado la **flecha izquierda**. El parámetro que usamos es **KeyCode.LeftArrow**; en el momento en que se cumple el **if**, el objeto rota también 5 grados, pero en la dirección contraria, por eso colocamos el valor **-5**.

Al final hacemos lo mismo pero para controlar el avance. La verificación de la tecla es con **KeyCode.UpArrow**, y cuando se cumple el **if**, en vez de rotar, hacemos una traslación de una unidad sobre el eje **Z**.

Ya podemos asignar el script al cubo. Cuando ejecutemos la aplicación y oprimamos la tecla, no sucederá nada, como tampoco si mantenemos la tecla oprimida. El cubo únicamente responderá cuando se deje de presionar la tecla.



► **Figura 9.** El cubo únicamente se mueve cuando liberamos la tecla; al oprimirla, no sucede nada.

Capturar la entrada del mouse

El mouse es otro dispositivo de entrada que podemos usar para interactuar con Unity. Vamos a aprender cómo leer los botones del mouse y luego su movimiento. La parte del movimiento puede aplicarse a otros dispositivos, por lo que la trataremos de forma separada.

En este ejemplo tendremos un cubo que rotará de diferentes maneras dependiendo del botón o botones que se opriman. Empezamos por crear una nueva escena, que lleva el nombre de **Ejemplo2**. Colocamos dos cubos, ya que más adelante agregaremos funcionalidades al segundo. El primero se llama **Cubo1**, y el segundo, **Cubo2**. Ponemos el primero en **(-2,0,0)** y el segundo en **(0,0,0)**. Para el primero asignamos el material **MaterialCubo** que creamos en el ejemplo anterior. Recordemos que los recursos presentes en el panel **Project** se comparten entre todas las escenas que tenemos en un proyecto. Luego, generamos un nuevo material llamado **MaterialRojo**, al que le colocamos rojo como color difuso; lo asignamos al segundo cubo. La escena necesita iluminación, por lo que agregamos una luz de tipo **Pointlight** en **(0,2.5,-1.8)** y ponemos el valor del rango en **10** para iluminar todo correctamente.

EL MOUSE ES OTRO
DISPOSITIVO QUE
PODEMOS UTILIZAR
PARA INTERACTUAR
CON UNITY

El siguiente punto es crear un script donde colocaremos la lógica de control. Lo llamamos **BotonesMouse** y escribimos en su interior el código provisto en la página siguiente.

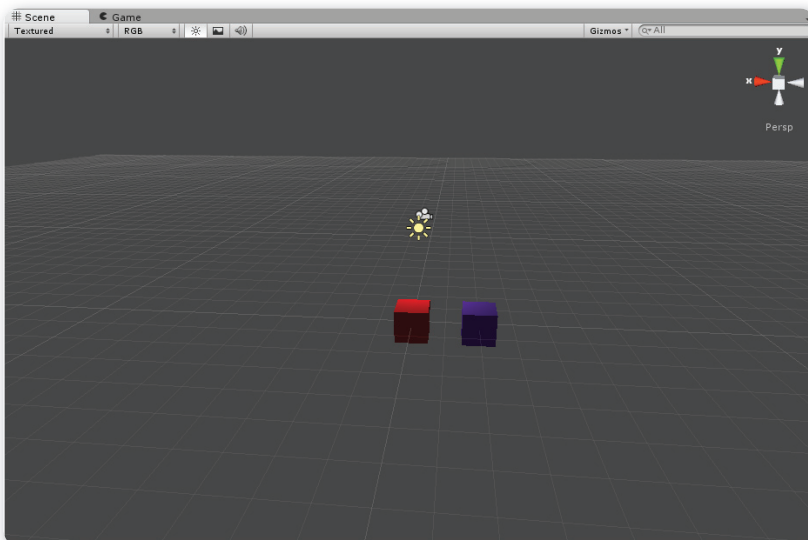


Figura 10. La nueva escena consta de dos cubos que podrán ser controlados por el mouse.


```
#pragma strict

// Propiedades necesarias
// Velocidad del objeto
var velocidad:float = 50.0;

function Start () {

}

// Al oprimir los botones del mouse se controla la
// orientacion del objeto
function Update () {

    // Verificamos si se oprime el boton izquierdo
    if(Input.GetMouseButton(0))
        transform.Rotate(0,velocidad*Time.deltaTime,0);

    // Verificamos si se oprime el boton central
    if(Input.GetMouseButton(1))
        transform.Rotate(velocidad*Time.deltaTime,0,0);

    // Verificamos si se oprime el boton derecho
    if(Input.GetMouseButton(2))
        transform.Rotate(0,-velocidad*Time.deltaTime,0);

}
```

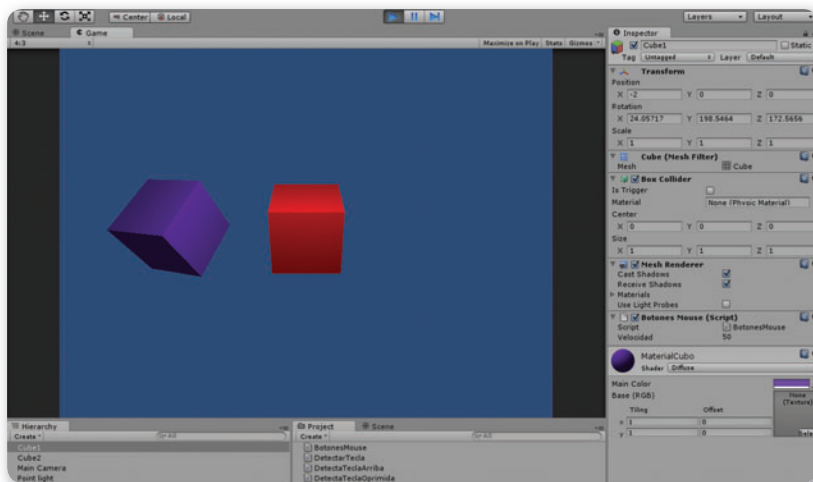
Lo primero que definimos es una propiedad llamada **velocidad**, que es de tipo **float**, a la cual inicializamos con el valor **50**. Esta velocidad se usa para la rotación y está dada en grados por segundo.

El método que debemos utilizar para leer al botón del mouse se llama **GetMouseButton**; regresa un valor de tipo **bool**, ya sea **true** o **false**. Indicamos qué botón nos interesa leer por medio del parámetro. Cuando el botón del mouse está siendo oprimido, el método regresa el valor **true**. Es decir que mientras el botón se oprima, tendremos rotación. El parámetro es un valor numérico entero. Cuando se coloca el valor **0**,

nos referimos al botón izquierdo del mouse, el valor **2** es para el botón derecho, y **1**, para el central.

El primer **if** verifica si se ha pulsado el botón **izquierdo**. Es importante notar que el método **GetMouseButton** pertenece a la clase **Input**. Si el **if** se cumple, entonces rotamos el objeto en el eje **Y** usando el valor basado en el tiempo. Lo mismo sucede con los otros dos botones. En el caso del botón **central**, hacemos la rotación en el eje **X**; y para el botón **derecho**, la rotación se hace otra vez en **Y**, pero en el sentido contrario.

Con esto finalizamos el script y lo asignamos al primer cubo creado. Al ejecutar la aplicación, simplemente debemos presionar los diferentes botones del mouse.



► **Figura 11.** Es posible controlar el movimiento del cubo por medio de los botones del mouse.



EL USO DE GETMOUSEBUTTONUP



También es posible detectar el momento en el cual alguno de los botones de mouse es liberado. Para lograrlo, usamos el método **GetMouseButtonUp**, que pertenece a la clase **Input**. En el momento en que el botón se libera, el método regresa el valor **true**; en los otros casos, regresa **false**. Como parámetro se indica el botón que nos interesa leer.

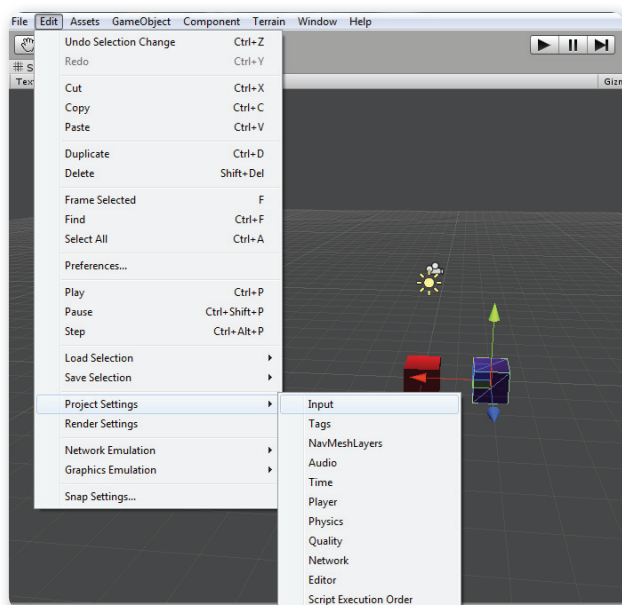
Leer otros dispositivos

La siguiente técnica nos servirá para leer diferentes tipos de dispositivos, como joysticks, controles, etc. Por una cuestión de simplicidad, vamos a usar el mouse, pues lo encontramos en todas las computadoras; sin embargo, también sirve con cualquier otro dispositivo de control.

VAMOS A USAR EL
MOUSE, YA QUE LO
ENCONTRAMOS
EN TODAS LAS
COMPUTADORAS



Lo primero que tenemos que hacer es configurar el sistema de entradas. En Unity, se abstraen las diferentes entradas, de modo que podamos programar y el código funcione con cualquier tipo de dispositivo, incluidos los que aún no estén en el mercado, en vez de tener que hacer llamadas a funciones específicas por cada dispositivo. Por ejemplo, podemos hacer un movimiento horizontal con el mouse, el joystick o un control, pero seguramente cada uno de ellos tiene su propia forma de trabajarlo, su rango y

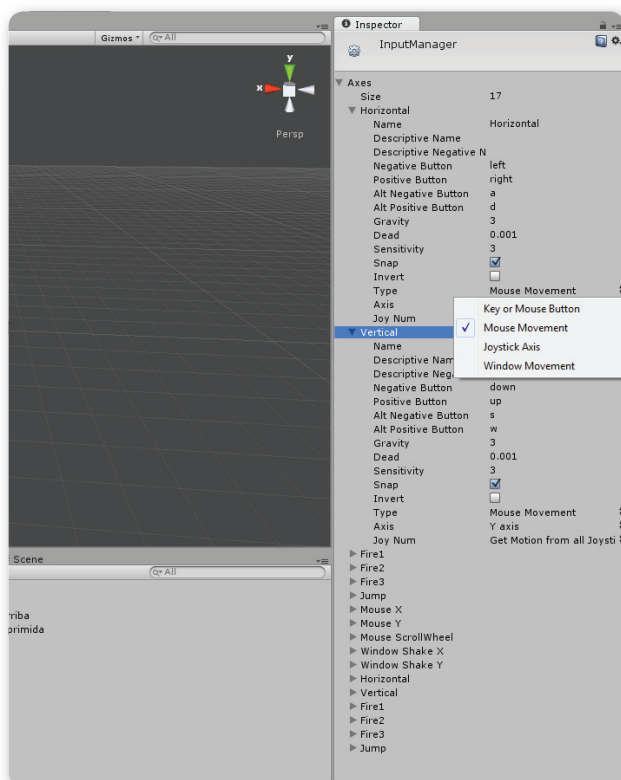


► **Figura 12.** Nos dirigimos al menú **Input** para configurar el sistema de entrada.

valores. En Unity usamos el concepto de eje horizontal y, simplemente, nuestro código se referencia a él. Al configurar el sistema de entrada, cambiamos el dispositivo si es necesario, para asegurarnos de no tener que modificar el código.

Para configurar el sistema de entrada, vamos al menú **Edit**, elegimos **Project Settings** y accedemos a **Input**. En el panel **Inspector** podemos ver las propiedades que hay que configurar. Cada una de ellas se refiere a un concepto, y podemos configurar varios valores para que funcione como deseamos.

Vamos a trabajar con los ejes horizontales y verticales por medio del mouse, de manera que debemos configurarlos. Nos dirigimos a la



► **Figura 13.** En Unity, la configuración de los parámetros de funcionamiento del mouse puede realizarse con gran facilidad.

sección **Axes**, y abrimos **Horizontal** y **Vertical**. En ambos encontramos una propiedad llamada **Type**, dentro de la cual seleccionamos **Mouse Movement**. De esta forma, indicamos que el movimiento del mouse será tomado para modificar el valor del eje.

Con la entrada configurada, podemos empezar a programar el script. Lo llamamos **Ejes** y le ponemos el siguiente código:

```
#pragma strict

// Propiedades necesarias
// Velocidad de giro del objeto
var velAngular:float = 0.05;
// Velocidad del objeto
var velocidad:float = 0.005;

function Start () {

}

function Update () {

    // Rotamos de acuerdo al movimiento del eje horizontal
    transform.Rotate(0,velAngular*Input.GetAxis("Horizontal"),0);

    // Rotamos de acuerdo al movimiento vertical del eje
    transform.Translate(0,0,velocidad*Input.GetAxis("Vertical"));

    // Si se oprime el boton de disparo subimos el objeto
```



LECTURA DE CONTROLES CON TOUCH



Unity incluye métodos que nos permiten leer la información de aquellos dispositivos cuya entrada esté basada en **touch**. En la clase **Input** encontramos el método **GetTouch**. Este regresa un objeto de tipo **Touch** que contiene información sobre el toque dado al dispositivo. El código para usarlo es un poco más complejo, pero el correspondiente para leer las propiedades de **Touch** es muy fácil de implementar.

```
if(Input.GetButtonDown("Fire1"))  
    transform.Translate(0,0.5,0);  
  
}
```

Empezamos creando dos propiedades. Una es para la velocidad angular, que es de tipo **float**; le damos el valor inicial de **0.05**. La otra es la velocidad del objeto al avanzar en el mundo, y es de **0.005**.

El método que vamos a utilizar pertenece a la clase **Input**. Va a ser un poco diferente de los anteriores, porque dará un valor numérico en vez de **true** o **false**. Este valor puede considerarse como el porcentaje que hemos movido el control en dicho eje. Esto resulta muy bueno, pues podemos hacer que el software reaccione de maneras diferentes a la cantidad de movimiento efectuado por el usuario. Si el eje del controlador se mueve en lo que consideramos el lado positivo, el valor es entre **0** y **1**. En caso de que el eje se mueva hacia el lado negativo, el valor es entre **0** y **-1**. Si el eje no se mueve, entonces el valor es **0**. Esto es lo mismo para ambos ejes.

El método usado se llama **GetAxis** y tiene un parámetro que es el nombre del eje por leer. El nombre se escribe entre comillas pues es una cadena. En el código obtenemos la cantidad de movimiento en el eje horizontal, y ese dato lo multiplicamos por la velocidad angular. El valor se usa entonces para rotar el objeto en el eje **Y**.

Hacemos algo similar para el eje vertical: lo leemos y usamos el valor del movimiento en ese

EL MÉTODO
GETAXIS TIENE UN
PARÁMETRO QUE
ES EL NOMBRE
DEL EJE POR LEER



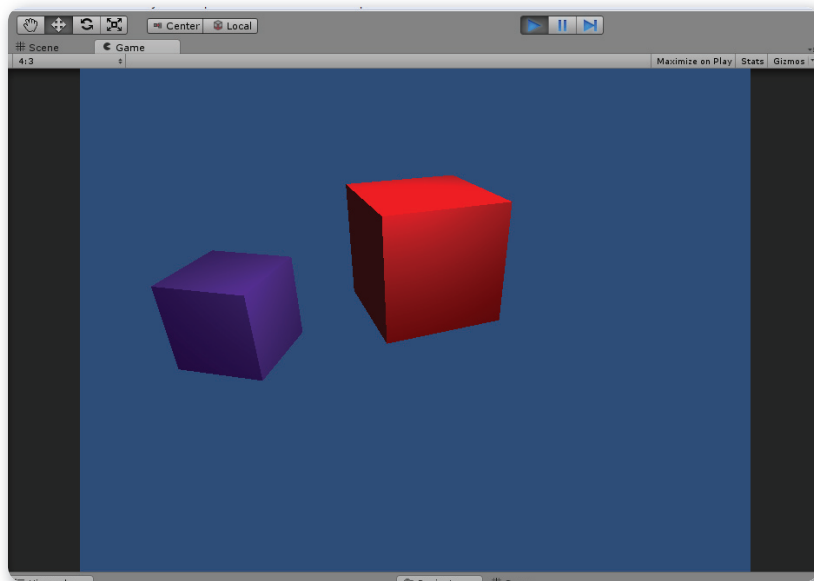
LECTURA DE LA BRÚJULA



En los dispositivos móviles podemos encontrar brújulas que indican hacia dónde está orientado el dispositivo. En Unity existe la propiedad **compass** dentro de la clase **Input**, que guarda la información actualizada de los valores de la brújula, para poder leerlos y usarlos en nuestra aplicación. Hay que notar que esta información solo está disponible en los dispositivos que físicamente tengan los circuitos de brújula en su interior.

eje multiplicado por la velocidad para calcular la traslación del objeto. Esta se lleva a cabo en el eje **Z**.

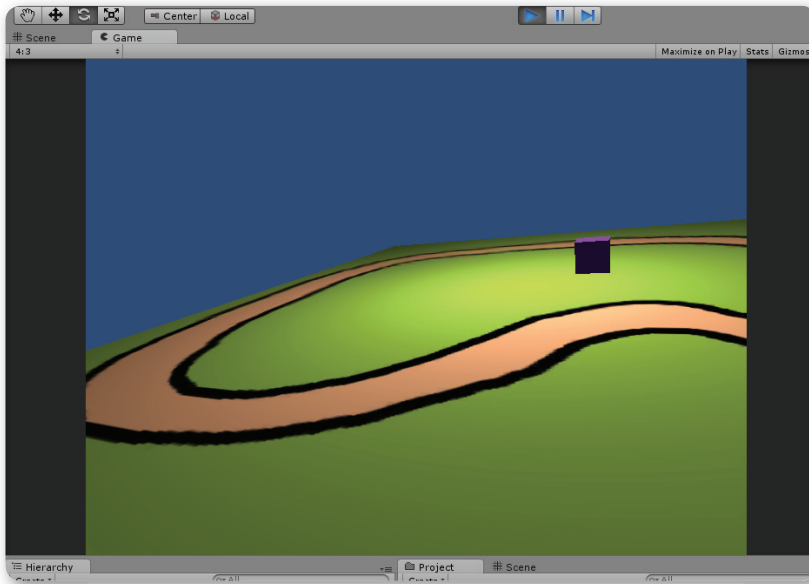
Esta forma de programar se usa no solo con los ejes del control: también puede usarse con los botones que tiene el control. Cada uno tiene un nombre, y puede ser asignado a un control, joystick o mouse.



► **Figura 14.** El cubo responde a las órdenes del mouse y efectúa las acciones como si fuera cualquier otro dispositivo de control.

En este ejemplo hacemos uso del botón conocido como **Fire1**, que equivale al botón principal del controlador. Como estamos usando el mouse, este botón equivale al botón **izquierdo**. El botón es leído por **GetButtonDown** y se pasa su nombre como parámetro, entre comillas. Si el botón se oprime, el objeto se traslada **0.5** unidades en **Y**. Ahora solo nos queda asignar el script al cubo y ejecutar la aplicación para poder controlar el cubo.

Hemos visto las diversas formas en que podemos obtener entradas del usuario y hacer que nuestro programa reaccione a ellas. Solamente debemos colocar la lógica adecuada en los **if** correspondientes o usar los valores que regresan los métodos.



► **Figura 15.** El asignar el script a la cámara provoca interesantes efectos visuales.



RESUMEN



Es posible obtener entradas de los diferentes controles que puede tener el usuario por medio de la clase **Input**, la cual nos provee de muchos métodos para utilizar. **KeyDown()** regresa un valor **true** o **false** cuando se oprime la tecla deseada. Para indicar en muchos de estos métodos cuál es la tecla que queremos, usamos **KeyCode**. El método de **GetKey()** regresa el valor **true** mientras se está oprimiendo la tecla. Si deseamos saber cuándo se la libera, recurrimos al método **KeyUp()**. Cuando es necesario leer los botones del mouse, se usa el método **GetMouseButton()**, el cual también regresa un valor de tipo **bool**: **true** cuando se oprime y **false** cuando no se está oprimiendo el valor. Para usar algunos controles, precisamos configurar **Input**. Si vamos a leer los ejes del control, se usa el método **GetAxis()**, pasando como parámetro el nombre del eje que nos interesa. En este caso, el método regresa un valor entre **-1** y **1**.

Actividades

TEST DE AUTOEVALUACIÓN

- 1 Observe todas las propiedades para configurar **Input**.
- 2 ¿Qué tipo de valores regresa **GetKeyDown()**?
- 3 Investigue sobre todos los posibles valores de **KeyCode**.
- 4 ¿Cuál es el funcionamiento de **GetKey()**?
- 5 ¿De qué manera funciona **GetKeyUp()**?
- 6 ¿Cómo se usa **GetMouseButton()**?
- 7 ¿Existen otros métodos para leer los botones del mouse?
- 8 ¿Cómo funciona **GetAxis()**?
- 9 ¿Qué valores regresa **GetAxis()**?
- 10 ¿Qué otros valores se pueden colocar como parámetros en el método **GetAxis()**?
- 11 ¿Qué tipo de controles pueden usarse con Unity?
- 12 ¿Por qué es conveniente probar el código con el mouse?

ACTIVIDADES PRÁCTICAS

- 1 Modifique el segundo ejemplo para que el cubo también se mueva para atrás.
- 2 Modifique los ejemplos para que el cubo se mueva con las teclas **A, W, S, D**.
- 3 Modifique los ejemplos para que la velocidad del cubo aumente con la **barra espaciadora** y disminuya con **CTRL**.
- 4 Modifique los ejemplos para controlar dos cubos diferentes usando las teclas numéricas.
- 5 Modifique el ejemplo final para controlar el cubo por la pista usando únicamente el mouse y avanzando al oprimir el botón **izquierdo**.



Manejo de cámaras

En este capítulo vamos a trabajar con la cámara, la cual nos permite ver los objetos que se encuentran en la escena. Al poder programarle scripts, tenemos la posibilidad de desplazarnos por la escena cuando el juego está en ejecución. Existen diferentes formas de utilizar la cámara. Conoceremos sus conceptos más importantes, que nos ayudarán a controlarla de acuerdo con nuestras necesidades.

▼ Conocer la cámara 98

- Crear una aplicación de prueba..... 99
- Propiedades principales de la cámara..... 104
- La cámara sigue a un objeto 115
- Mover la cámara..... 119
- Usar los scripts de Unity 121

La cámara en primera persona 123

La cámara en tercera persona 124

▼ Resumen..... 125

▼ Actividades..... 126

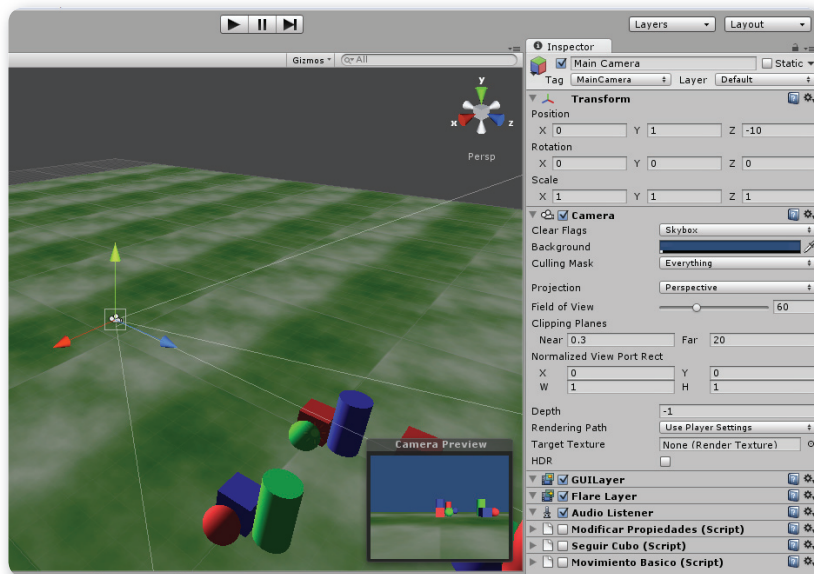


Conocer la cámara

La cámara es un objeto que nos permite observar nuestra escena. En realidad, no existe como tal, es más bien una abstracción que nos permite usar el concepto real de una cámara dentro de Unity. Se trata de una serie de transformaciones matemáticas que toman el mundo virtual como modelo matemático y lo proyectan en un plano de dos dimensiones que podemos observar en la pantalla. No vamos a entrar en detalle en cuanto a los conceptos matemáticos de la cámara, sino que nos concentraremos en la forma de usarla dentro de Unity.

En cualquiera de las aplicaciones que hemos creado, veremos que tenemos un objeto llamado **Main Camera**, creado automáticamente por Unity. Este objeto representa a la cámara principal que usamos en la aplicación. Tiene, además, varios componentes que podemos observar desde el panel **Inspector** cuando está seleccionado.

El objeto **Main Camera** tiene un componente de tipo **Transform**, que es necesario para indicar la posición de la cámara y para rotarla. La



► **Figura 1.** Para observar los componentes de la cámara, simplemente la seleccionamos y recurrimos al panel **Inspector**.

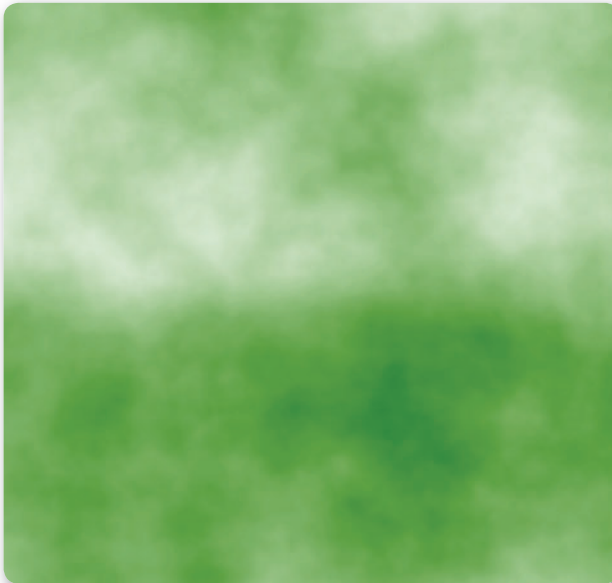
rotación de la cámara nos permite modificar la dirección hacia donde apunta. La escala no tiene efecto sobre la cámara principal.

Luego, encontramos otro componente que pertenece a la clase **Camera**. Esta clase contiene el código necesario para que la cámara funcione y presenta varias propiedades. La mejor manera de entenderlas es usándolas, por lo que empezaremos a hacerlo.

Crear una aplicación de prueba

Vamos a crear un nuevo proyecto de Unity al cual llamamos **Capítulo4**, y en él guardamos la escena como **Ejemplo1**. En ella crearemos varios objetos que nos permitirán observar no solo los cambios en las propiedades, sino también el control sobre la cámara misma.

Para empezar, creamos tres materiales, como vimos en el **Capítulo 2**, que llevan los nombres **MaterialAzul**, **MaterialRojo** y **MaterialVerde**; el color que lleva cada uno corresponde a su nombre. La aplicación tendrá un piso al cual le colocaremos una textura que deberemos crear u obtener.

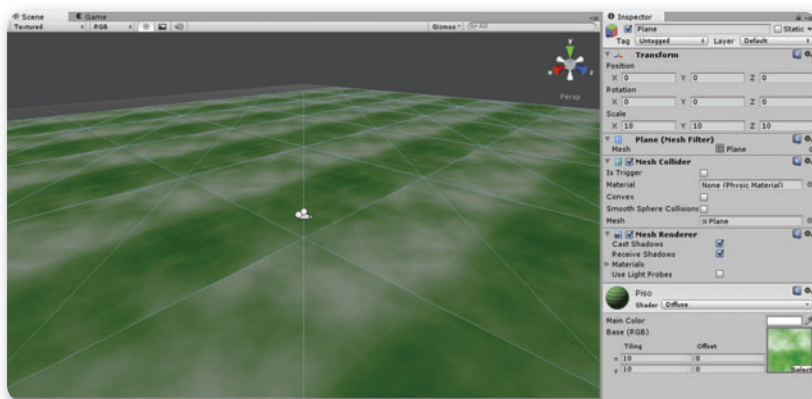


► **Figura 2.** Esta imagen es usada para la textura del piso que colocamos en nuestra aplicación de ejemplo.

La imagen se importa a Unity arrastrándola y depositándola en el panel **Project**. Entonces, creamos un nuevo material llamado **Piso** y le asignamos esta textura. Ahora ya tenemos los materiales necesarios y podemos empezar a agregar objetos.

Comenzamos con la luz. En esta ocasión, colocamos una luz de tipo **Directional Light** (en un capítulo siguiente veremos más a fondo las luces y sus diferencias), con una posición **(0,0,0)** y una rotación de **22** grados en el eje **X**. Luego, añadimos un objeto de tipo **Plane**, también en la posición **(0,0,0)**, y para que tenga un buen tamaño, colocamos la escala con valor **10** en cada uno de los ejes. A este plano le asignamos el material **Piso**.

Vamos a aprender a sacar provecho de las ventajas de edición que nos brinda Unity. Lo primero que hacemos es crear un objeto vacío



► **Figura 3.** Así luce el plano con el material que contiene la textura; ahora vamos a colocar más objetos.



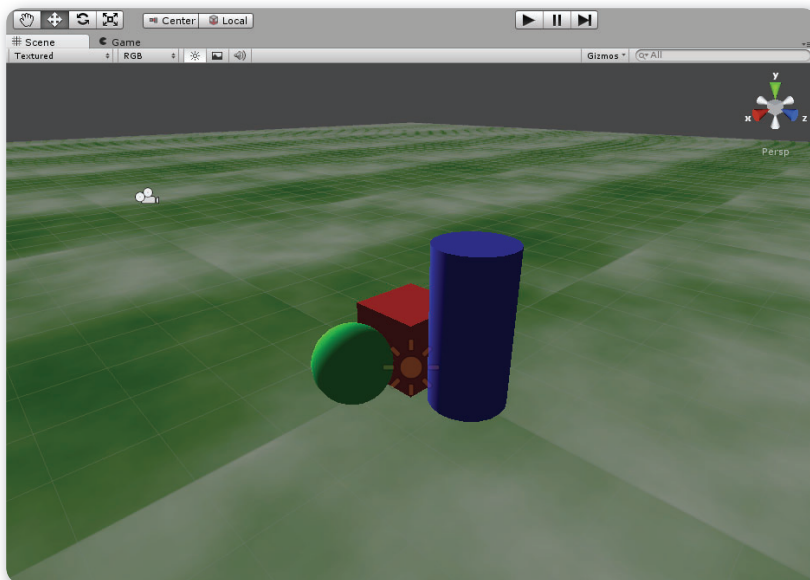
EL USO DE PARENTING

El parenting nos permite tener un objeto y, luego, referenciar a otros en términos del objeto padre. Podemos usarlo también para agrupar objetos y trabajar con ellos como una unidad. Por ejemplo, si tenemos una canasta y colocamos objetos en su interior, al mover la canasta, todos los objetos se mueven con ella. El uso de parenting es muy útil en diferentes técnicas.

desde el menú **GameObject**, y luego seleccionamos **Create Empty**. Este objeto vacío nos servirá como contenedor de otros al hacer uso de la técnica de **parenting**. A este objeto le ponemos como nombre **Objetos1** y lo ubicamos en la posición **(0,0,5,0)**.

Ahora creamos un cubo y le asignamos el material **MaterialRojo** y, en el panel **Hierarchy**, hacemos clic con el botón **izquierdo** del mouse y lo arrastramos hasta **Objetos1**. Unity crea el parenting y a la izquierda de **Objetos1** aparece un triángulo que apunta hacia abajo. En este grupo de objetos colocaremos dos más. Creamos un cilindro y le otorgamos **MaterialAzul**; por último, ponemos una esfera y le asignamos **MaterialVerde**. Ahora los tres objetos tienen establecido el parenting con relación a **Objetos1**.

Una vez que el parenting fue establecido, colocaremos los objetos en posición relativa a **Objetos1**. Es decir que, al ser el padre del grupo, este actúa como origen para las posiciones de los demás. Si movemos **Objetos1**, todos sus objetos hijo se moverán junto con él. Seleccionamos el cubo y en la propiedad **Transform** colocamos la posición **(0,0,0)**. Luego,



► **Figura 4.** Podemos ver cómo luce el grupo de objetos que creamos y ubicamos sobre la escena.

LA IMAGEN SE
IMPORTA A UNITY
ARRASTRÁNDOLA Y
DEPOSITÁNDOLA EN
EL PANEL PROJECT

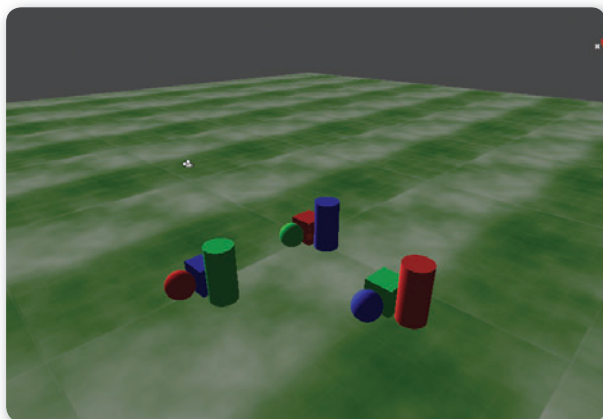


seleccionamos el cilindro y le damos posición **(0,0,5,1)**. Únicamente falta la esfera, que ubicamos en posición **(1,0,0)**. Hemos creado un grupo de tres objetos agrupados.

Ahora necesitamos crear otros dos grupos de objetos. Esto parece tedioso, pues son seis objetos más y dos vacíos, pero con Unity podemos realizarlo fácilmente. Hacemos clic sobre el nombre **Objetos1** en el panel **Hierarchy**, de tal modo que se seleccione, y luego hacemos clic derecho

con el mouse. Aparece un menú contextual de donde elegimos **Duplicate**. De este modo, creamos un nuevo objeto exactamente igual al que tenemos seleccionado.

Al objeto creado lo llamamos **Objetos2** y en su propiedad **Transform** colocamos su nueva posición. Es importante notar que debemos colocar



► **Figura 5.**
Esta es la vista que tiene la cámara de la escena que hemos construido.



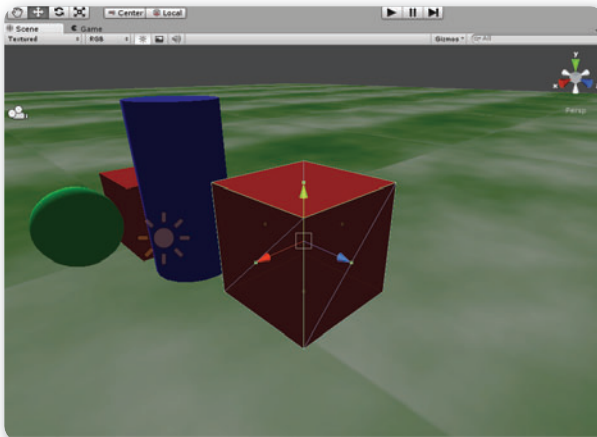
DUPLICAR OBJETOS



Unity nos permite duplicar los objetos con el fin de reducir la cantidad de pasos necesarios para crear una escena. Cuando lo hacemos, debemos considerar que el nuevo objeto se crea con el mismo nombre y en la misma posición. Es recomendable renombrarlo de inmediato y cambiarlo de posición, para evitar confusiones. Cualquiera de los dos objetos puede ser renombrado, ya que ambos son iguales.

la posición para **Objetos2** y no para uno de los objetos que están en su interior, por lo que debemos verificar en el panel **Hierarchy** que el objeto seleccionado sea **Objetos2**. La posición que le damos a este objeto es **(1.5,0.5,5)**. Con esto, ya tenemos algunos objetos a la distancia que nos servirán de referencia en los ejemplos del capítulo. Podemos cambiarles los materiales para tener más variedad visual. Una vez más, llevamos a cabo la duplicación y le damos al nuevo objeto el nombre **Objetos3**. Este tiene que ubicarse en otra posición, de modo que le asignamos los valores **(5,0.5,0)**.

Para el ejemplo que estamos desarrollando, falta únicamente un objeto. Crearemos un cubo y lo llamaremos **CuboMovimiento**. Las coordenadas donde lo colocaremos son **(0,1,3)**. El movimiento de este nuevo cubo creado será controlado por el usuario, y la cámara lo seguirá con la vista.



► **Figura 6.** Este cubo se mueve dentro de la escena, y la cámara lo sigue. Es un ejemplo de cómo podemos usar la cámara.



CONTROLAR LA ANIMACIÓN



En la cámara de tercera persona se pueden asignar las animaciones correspondientes al personaje que se va a usar. La animación **Idle** representa al personaje cuando está en espera, sin caminar; y **Walk** es el personaje caminando. En el caso de **Run**, ponemos la animación que corresponde al personaje corriendo, en tanto que **Jump** es el personaje saltando.

Propiedades principales de la cámara

Una vez que la escena fue creada, podemos proceder a guardarla y construir ejemplos que nos permitan trabajar con las diferentes propiedades de la cámara. Creamos un nuevo script denominado **ModificarPropiedades**, y dentro de él colocamos código que nos permita usar las propiedades más importantes. Empecemos por variar el color del fondo. Inicialmente, Unity tiene un color de fondo azulado, es decir que cualquier espacio de la pantalla que no sea ocupado por un objeto tendrá este color. Podemos modificar el color de fondo, al igual que las demás propiedades, por medio del panel **Inspector**, pero también es posible hacerlo de manera programática, que es como se detalla a continuación. Empezamos con el siguiente código en el script:

```
#pragma strict

// Variables para guardar el color
private var r:float = 0.0;
private var g:float = 0.0;
private var b:float = 0.0;

function Start () {

}

function Update () {

    // Cambiamos el color del fondo
    if(Input.GetKeyDown(KeyCode.Alpha1))
    {
```



CONTROLADORES DE CÁMARA



En los scripts que ofrece Unity para el control de las cámaras encontramos algunas propiedades que nos permiten configurar cómo se comporta su movimiento. Podemos indicar la velocidad al caminar, trotar y correr. También, podemos asignar los valores para el brinco y la gravedad. Al modificar estos valores, es posible hacer que la cámara funcione de un modo más personalizado.

```
// Seleccionamos un color al azar para
// cada componente de color
r = Random.value;
g = Random.value;
b = Random.value;

// Creamos una variable de tipo color
// la inicializamos con los valores de r,g,b
var micolor: Color = Color (r,g,b);

// Asignamos el color al fondo
this.camera.backgroundColor=micolor;
}

}
```

Lo que hace el script es elegir un color al azar al momento en que la tecla **1** del teclado alfanumérico es presionada.

Empezamos creando tres variables de tipo float, donde cada una de las ellas representa un componente de color.

Dentro del método **Update()** verificamos si se ha oprimido la tecla **1** del teclado alfanumérico. Como solo deseamos que el cambio de color se dé al oprimir la tecla y no al mantenerla oprimida, usamos el método **GetKeyDown()**. Cuando se cumpla la condición, le daremos valores al azar entre **0** y **1** a cada una de las variables de los componentes de color.

ES POSIBLE
MODIFICAR EL
COLOR DE FONDO
POR MEDIO DEL
PANEL INSPECTOR



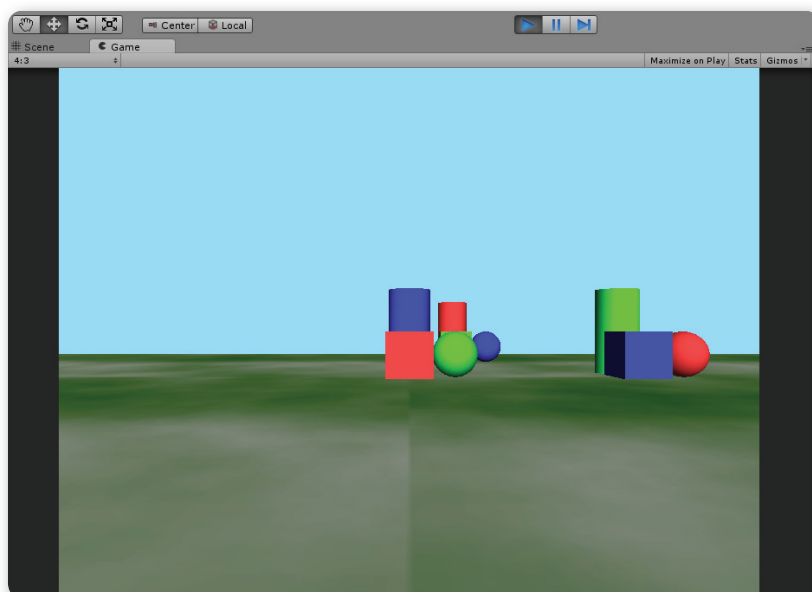
LA PROPIEDAD DE BACKGROUND COLOR



Podemos modificar esta propiedad dirigiéndonos directamente al panel **Inspector** y, allí, haciendo clic sobre el cuadro de color que se encuentra ubicado junto a su nombre, donde aparece una herramienta de selección de color. Programáticamente, pertenece a la clase **Camera** y su tipo es **Color**, por lo que debemos asignarle valores de este tipo.

No podemos asignar directamente los valores al fondo, sino que, es necesario tener un objeto de tipo **Color**, por lo que creamos uno llamado **micolor** y le asignamos los componentes correspondientes. Ahora **micolor** guarda en su interior un color elegido al azar, que es asignado al fondo por medio de la propiedad **backgroundColor**. A continuación, le asignamos el script a **Main Camera** y ejecutamos la aplicación. Cada vez que oprimamos la tecla **1**, el color del fondo cambiará.

Unity es un motor en tres dimensiones y, por eso, podemos observar en perspectiva el mundo virtual que hemos creado. Sin embargo, en



► **Figura 7.** Podemos modificar el color del fondo cada vez que hacemos clic en la tecla **1**.



LOS MODOS DE LA CÁMARA



En Unity tenemos dos modos de cámara: el de perspectiva y el ortográfico. En el primero, los objetos se ven más pequeños conforme se alejan de la cámara. En el segundo, los objetos no cambian de tamaño con la distancia, y sus dimensiones dependen de las coordenadas sobre el plano de proyección de la cámara.

ocasiones necesitamos simular una aplicación en 2D, o por algún otro motivo no deseamos que la escena se vea en perspectiva. Para hacer esto, debemos modificar la cámara para que trabaje en **modo ortográfico**.

El control del modo ortográfico se hace por medio de la propiedad **isOrthoGraphic**, que recibe un valor **true** o **false**. Cuando asignamos **true**, la cámara actúa de forma ortográfica, en tanto que cuando asignamos **false**, la cámara tendrá perspectiva.

Vamos a agregar código al script que ya tenemos, de modo que, al oprimir la tecla **2** del teclado alfanumérico, la cámara cambie de modo. Modificamos el script para que quede de la siguiente manera:

```
#pragma strict

// Variables para guardar el color
private var r:float = 0.0;
private var g:float = 0.0;
private var b:float = 0.0;

// Variable para tipo de camara
private var ortho:boolean = false;

function Start () {

}

function Update () {

    // Cambiamos el color del fondo
    if(Input.GetKeyDown(KeyCode.Alpha1))
    {

        // Seleccionamos un color al azar para
        // cada componente de color
        r = Random.value;
        g = Random.value;
        b = Random.value;

        // Creamos una variable de tipo color
        // la inicializamos con los valores de r,g,b
```

```
        var micolor: Color = Color (r,g,b);

        // Asignamos el color al fondo
        this.camera.backgroundColor=micolor;
    }

    // Cambiamos el modo ortografico/perspectiva
    if(Input.GetKeyDown(KeyCode.Alpha2))
    {

        // Se hace un toggle a la variable
        ortho=!ortho;

        // Se asigna el nuevo estado a la camara
        this.camera.isOrthoGraphic=ortho;
    }

}
```

Encontramos una variable de tipo **boolean** que se llama **ortho**. Esta funciona como un **flag** que nos permite saber cuál es el modo actual de la cámara y cambiarlo. La inicializamos con valor **false**, porque Unity tiene la cámara en modo perspectiva cuando inicia, salvo que lo indiquemos de otra manera. En la parte inferior verificamos si se ha oprimido la tecla **2** del teclado alfanumérico. Si este es el caso, lo primero que hacemos es invertir el valor de **ortho**. Esta línea hace que, cada vez que la ejecución pase por ese punto, cambie de **true** a **false**, y viceversa. El nuevo valor actualizado de **ortho** se asigna a la propiedad **isOrthographic**, perteneciente a la clase **Camera**.

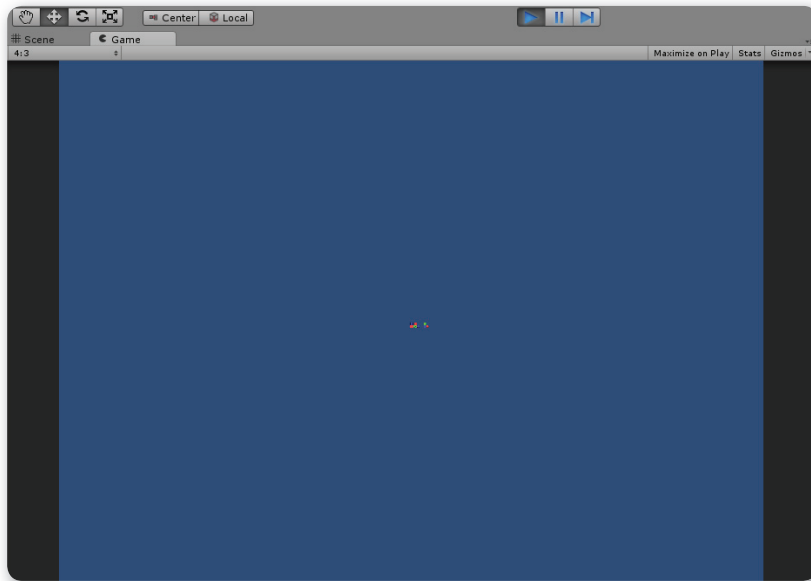


EL FLAG



El **flag** también se conoce como bandera, y generalmente, es una variable de tipo **boolean** que nos indica si algo ha sucedido o no. También puede indicarnos el estado de algo. Por ejemplo, si deseamos saber si algo ha ocurrido, le colocamos el valor **true**; si aún no sucedió, el de **false**. Esto podrá luego ser procesado sin inconvenientes por la lógica del programa.

Después de salvar el script, podemos ejecutar la aplicación y comparar los dos modos de la cámara.



► **Figura 8.** Este es el modo ortográfico, donde los objetos lucen como puntos pues sus dimensiones no son muy grandes.

Después de esto, podemos empezar a trabajar con la apertura de campo de la cámara. Esta nos indica cuántos elementos podemos observar; imaginemos que es equivalente a los lentes de una cámara. Cuando tengamos una gran apertura de campo, podremos observar más elementos, pero su tamaño será menor y lucirán más pequeños en la pantalla. Si la apertura es pequeña, podremos observar menos elementos, aunque estos lucirán más grandes. En general, una apertura de **60** grados suele funcionar muy bien y, además, es utilizada frecuentemente en aplicaciones de videojuegos.

En Unity la apertura de campo es una propiedad de la clase **Camera** y se llama **fieldOfView**. Podemos colocarle el valor en grados que deseamos.

AL TENER UNA
GRAN APERTURA DE
CAMPO, VEREMOS MÁS
ELEMENTOS, PERO SU
TAMAÑO SERÁ MENOR



Ahora vamos a ingresar nuevo código, de modo que, cuando se oprima la tecla **3** del teclado alfanumérico, se incremente el campo de visión; y cuando se oprima **4**, disminuya:

```
#pragma strict

// Variables para guardar el color
private var r:float = 0.0;
private var g:float = 0.0;
private var b:float = 0.0;

// Variable para tipo de camara
private var ortho:boolean = false;

function Start () {

}

function Update () {

    // Cambiamos el color del fondo
    if(Input.GetKeyDown(KeyCode.Alpha1))
    {

        // Seleccionamos un color al azar para
        // cada componente de color
        r = Random.value;
        g = Random.value;
        b = Random.value;

        // Creamos una variable de tipo color y
        // la inicializamos con los valores de r,g,b
        var micolor: Color = Color (r,g,b);

        // Asignamos el color al fondo
        this.camera.backgroundColor=micolor;
    }

    // Cambiamos el modo ortografico/perspectiva
    if(Input.GetKeyDown(KeyCode.Alpha2))
```

```
{  
  
    // Se hace un toggle a la variable  
    ortho=!ortho;  
  
    // Se asigna el nuevo estado a la camara  
    this.camera.isOrthoGraphic=ortho;  
}  
  
// Modificamos el angulo de apertura de campo  
// Crea un efecto de zoom  
if(Input.GetKeyDown(KeyCode.Alpha3))  
{  
    // Incrementamos la apertura de campo en 5 grados  
    this.camera.fieldOfView+=5;  
}  
  
if(Input.GetKeyDown(KeyCode.Alpha4))  
{  
    // Reducimos la apertura de campo en 5 grados  
    this.camera.fieldOfView-=5;  
}  
}
```

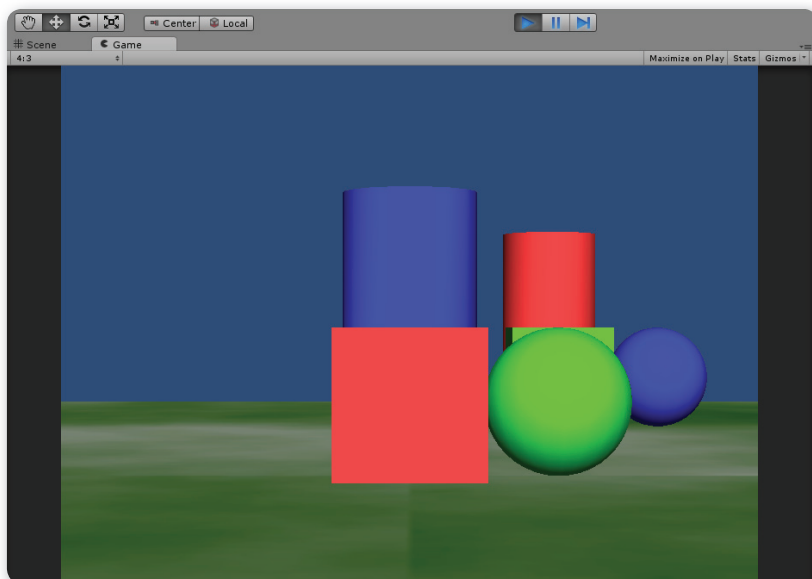
Cuando el usuario oprime la tecla **3**, simplemente incrementamos en **5** el valor de **fieldOfView**. Desde luego, podemos colocar cualquier valor que deseemos o, como en este caso, simplemente modificarlo. Si la tecla oprimida es **4**, el valor es disminuido en **5** grados. Al ejecutar la aplicación, veremos el funcionamiento de esto.



LOS PLANOS CERCANO Y LEJANO



Como método de optimización en las gráficas 3D, la cámara tiene límites en la distancia que puede observar. Este parámetro se define por medio de dos distancias a planos perpendiculares a la cámara, conocidos como **plano cercano y plano lejano**. Cualquier objeto que esté entre ellos es procesado para el render. Si el objeto está más lejos que el lejano o más cerca que el cercano, entonces no formará parte del proceso de render.



► **Figura 9.** Al asignar una apertura de campo pequeña, los objetos lucen de tamaño más grande.

Otro elemento que podemos trabajar es el valor del **plano lejano**, que nos permitirá optimizar mucho el desempeño de la aplicación, ya que, al colocar un valor adecuado para el tipo de escena que tenemos, podemos incrementar la cantidad de cuadros por segundo y usar ese poder de cómputo extra para dar más detalles al juego. Para llevar a cabo esta acción, vamos a modificar el programa de tal manera que, cuando se oprima la tecla **5** del teclado alfanumérico, el plano lejano se aleje una unidad, y cuando se oprima la tecla **6**, se acerque a la cámara una unidad. Con esto podemos observar que las partes de la escena que están más allá del plano ya no son procesadas para el render.

Modifiquemos nuestro script de la siguiente manera:

// Variables para guardar el color

```
private var r:float = 0.0;  
private var g:float = 0.0;  
private var b:float = 0.0;
```

```
// Variable para tipo de camara
private var ortho:boolean = false;

function Start () {

    this.camera.farClipPlane=20.0;
}

function Update () {

    // Cambiamos el color del fondo
    if(Input.GetKeyDown(KeyCode.Alpha1))
    {

        // Seleccionamos un color al azar para
        // cada componente de color
        r = Random.value;
        g = Random.value;
        b = Random.value;

        // Creamos una variable de tipo color
        // la inicializamos con los valores de r,g,b
        var micolor: Color = Color (r,g,b);

        // Asignamos el color al fondo
        this.camera.backgroundColor=micolor;
    }

    // Cambiamos el modo ortografico/perspectiva
    if(Input.GetKeyDown(KeyCode.Alpha2))
    {

        // Se hace un toggle a la variable
        ortho=!ortho;

        // Se asigna el nuevo estado a la camara
        this.camera.isOrthographic=ortho;
    }

    // Modificamos el angulo de apertura de campo
```



```
// para crear un efecto de zoom
if(Input.GetKeyDown(KeyCode.Alpha3))
{
    // Incrementamos la apertura de campo en 5 grados
    this.camera.fieldOfView+=5;
}

if(Input.GetKeyDown(KeyCode.Alpha4))
{
    // Reducimos la apertura de campo en 5 grados
    this.camera.fieldOfView-=5;
}

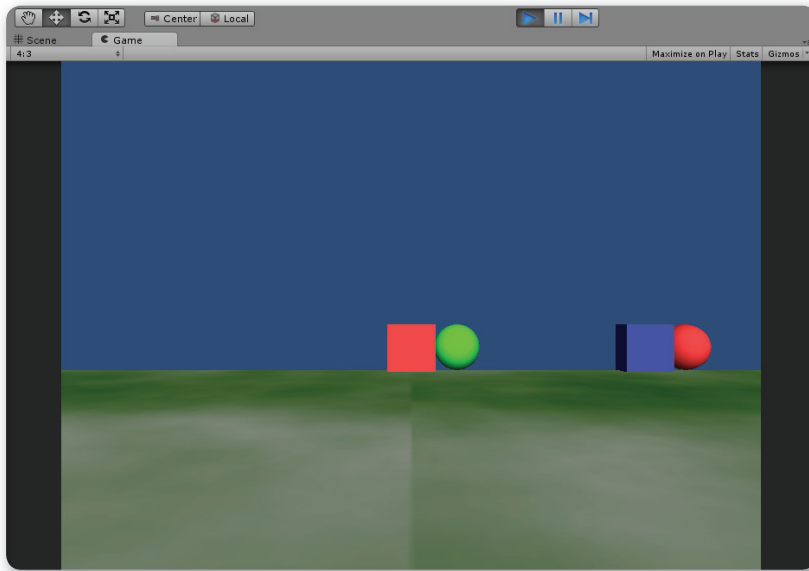
// Modificamos la distancia del plano lejano
if(Input.GetKeyDown(KeyCode.Alpha5))
{
    // Alejamos el plano lejano
    this.camera.farClipPlane++;
}

if(Input.GetKeyDown(KeyCode.Alpha6))
{
    // Acercamos el plano lejano
    this.camera.farClipPlane--;
}

}
```

En el método **Start()** estamos colocando un valor inicial al plano lejano, de modo que en la escena sea fácil observar el cambio de valor. Para colocar el valor del plano lejano, hacemos uso de la propiedad **farClipPlane**, perteneciente a la clase **Camera**. El valor colocado es la distancia en unidades del mundo a la que se encuentra el plano. Tradicionalmente, Unity lo coloca con un valor grande, y tardaríamos demasiado en ver el efecto. Luego, en el método **Update()** verificamos si se ha oprimido la tecla **5**, en cuyo caso simplemente incrementamos el valor que contenga la propiedad **farClipPlane**. Hacemos algo similar

para la tecla **6**, pero en esta oportunidad, disminuimos el valor correspondiente. Guardamos los cambios hechos al script y ejecutamos la aplicación. Vemos cómo, efectivamente, la cámara solo hace render de los objetos que están entre el plano lejano y el cercano.



► **Figura 10.** El plano lejano limita el punto hasta donde puede ver la cámara, y deja afuera aquellos objetos que están más allá.

Ya conocemos las propiedades principales de la cámara, ahora pasaremos a hacer cosas más interesantes con ella.

La cámara sigue a un objeto

Vamos a mover el cubo rojo y haremos que la cámara lo siga con la vista. Gracias a que Unity tiene una buena cantidad de métodos de ayuda, esta tarea no resultará difícil. Primero debemos colocarle el siguiente script al cubo llamado **MoverCubo**:

```
#pragma strict
```

```
// Propiedad para la velocidad del cubo
```

```
var velocidad:float=15;

function Start () {

}

function Update () {

    // Si oprimimos la tecla flecha derecha
    if(Input.GetKey(KeyCode.RightArrow))
    {
        // Trasladamos el cubo
        transform.Translate(velocidad*Time.deltaTime,0,0);
    }

    // Si oprimimos la tecla flecha izquierda
    if(Input.GetKey(KeyCode.LeftArrow))
    {
        // Trasladamos el cubo
        transform.Translate(-velocidad*Time.deltaTime,0,0);
    }

}
```

Con lo que ya hemos aprendido, no es necesario explicar este código. Vamos a crear otro script, que será asignado a la cámara. Se llamará **SeguirCubo** y tendrá el siguiente código:

```
#pragma strict

// Propiedad para guardar la transformada del
// objeto a seguir
var objetivo: Transform;

function Start () {

}

function Update () {
```

```
// Rotamos la camara en direccion al objetivo
```

```
transform.LookAt(objetivo);
```

```
}
```

Lo primero que encontramos es una propiedad llamada **objetivo**. Esta es un poco diferente de las que hemos usado anteriormente, ya que su tipo es **Transform**, por lo que nos permitirá pasar el componente **transform** de un objeto, que contiene su posición, y usarlo para lo que necesitemos en este script. Esta es una característica importante de Unity, porque nuestros scripts pueden tomar como parámetro a otros objetos del juego y, de esta manera, se incrementa y facilita la interacción entre ellos.

Una vez que tenemos las coordenadas del objeto que deseamos seguir con la mirada, podemos indicarle a la cámara que lo haga, usando el método **Update()**. Vamos a utilizar un método llamado **LookAt()**, perteneciente a la clase **Transform**. Como parámetro pasamos **Objetivo**, que contiene las coordenadas del cubo. Este método rota el objeto, que es la cámara, de tal manera que apunte hacia el objetivo.

Con ambos scripts asignados a los objetos que les corresponden, debemos colocar en la propiedad el cubo que vamos a seguir. Para lograrlo, seleccionamos la cámara y, en el panel **Inspector**, nos dirigimos al componente del script **SeguirCubo**. Vemos que ahí aparece la propiedad **Objetivo**. Ahora, desde el panel **Hierarchy** hacemos clic y arrastramos el objeto **CuboMovimiento** para colocarlo en la propiedad **Objetivo** del script; de este modo, la propiedad muestra **CuboMovimiento**. Con esto ya podemos ejecutar la aplicación.

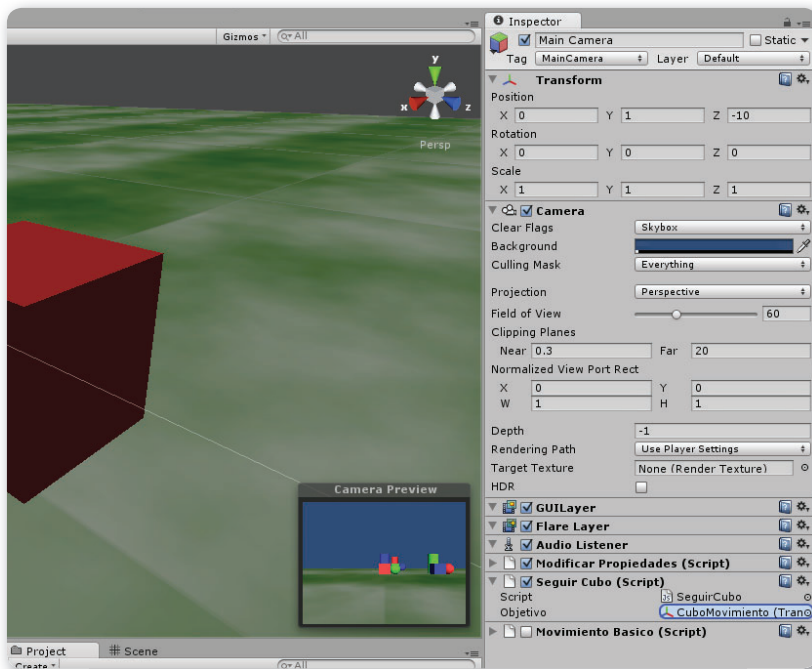
LOS SCRIPTS
PUEDEN TOMAR
COMO PARÁMETRO
A OTROS OBJETOS
DEL JUEGO



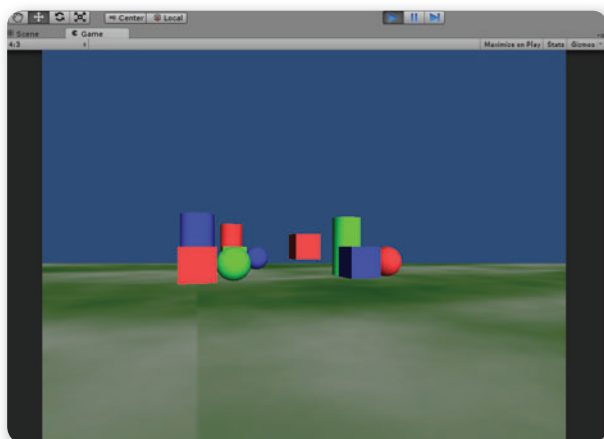
MÚLTIPLES CÁMARAS EN UNITY



En un principio, Unity presenta solo una cámara, pero nosotros podemos adicionar otras más e, incluso, hacer que una se renderice a una textura y se aplique a un objeto. La cantidad de cámaras de Unity es ilimitada, pero hay que tener siempre en cuenta el desempeño de la aplicación y la manera en que se distribuirán en la pantalla.



► **Figura 11.** El objeto **CuboMovimiento** es arrastrado a la propiedad **Objetivo** en el script de la cámara.



► **Figura 12.** La cámara sigue el movimiento del cubo, rotando hacia donde este se encuentra.

Mover la cámara

Vamos a construir algo más complejo, que nos permitirá, por primera vez, movernos libremente en la escena. Para hacerlo, creamos un nuevo script para la cámara, pero antes tenemos que deshabilitar los dos scripts anteriores. Nos movemos dentro de la escena por medio de las teclas de flechas. La **flecha arriba** nos permitirá avanzar; y la tecla **flecha abajo**, retroceder. También podemos girar con **flecha derecha** y **flecha izquierda**. El nuevo script tendrá como nombre **MovimientoBasico**, el cual le será asignado a la cámara. Le colocamos el siguiente código:

```
#pragma strict

// Propiedades necesarias
// Velocidad de la camara
var velocidad:float = 5.0;

// Velocidad de giro de la camara
var velAngular:float =45.0;

function Start () {

}

function Update () {

    // Verificamos si avanza la camara
    if(Input.GetKey(KeyCode.UpArrow))
    {

        // Trasladamos en el eje Z
        transform.Translate(0,0,velocidad*Time.deltaTime);

    }

    // Verificamos si retrocedemos
    if(Input.GetKey(KeyCode.DownArrow))
    {

        // Trasladamos en el eje Z
        transform.Translate(0,0,-velocidad*Time.deltaTime);

    }

}
```



```
// Verificamos por giro a la derecha
if(Input.GetKey(KeyCode.RightArrow))
{
    // Rotamos en el eje Y
    transform.Rotate(0,velAngular*Time.deltaTime,0);
}

// Verificamos por giro a la izquierda
if(Input.GetKey(KeyCode.LeftArrow))
{
    // Rotamos en el eje Y
    transform.Rotate(0,-velAngular*Time.deltaTime,0);
}
}
```

El código empieza por crear una propiedad para la velocidad de la cámara, llamada **velocidad**, a la que le damos un valor de inicio de **5** unidades por segundo. Encontramos otra propiedad denominada **velAngular**, que tiene un valor de **45** grados por segundo. Esta es usada para indicar con qué rapidez girará la cámara.

En el método **Update()** verificamos si se ha oprimido alguna tecla en particular. En caso de que se haya presionado la **flecha arriba**, llevamos a cabo una traslación en el eje **Z**. La cantidad por trasladar depende de la velocidad y el tiempo transcurrido. Algo similar ocurre con la **flecha abajo**, pero en este caso, la traslación es en la dirección contraria, para realizar el retroceso. Estos movimientos de traslación se hacen por medio del método **Translate()**, que pertenece a **transform**.

Una vez realizado este paso, verificamos la tecla **flecha derecha** para



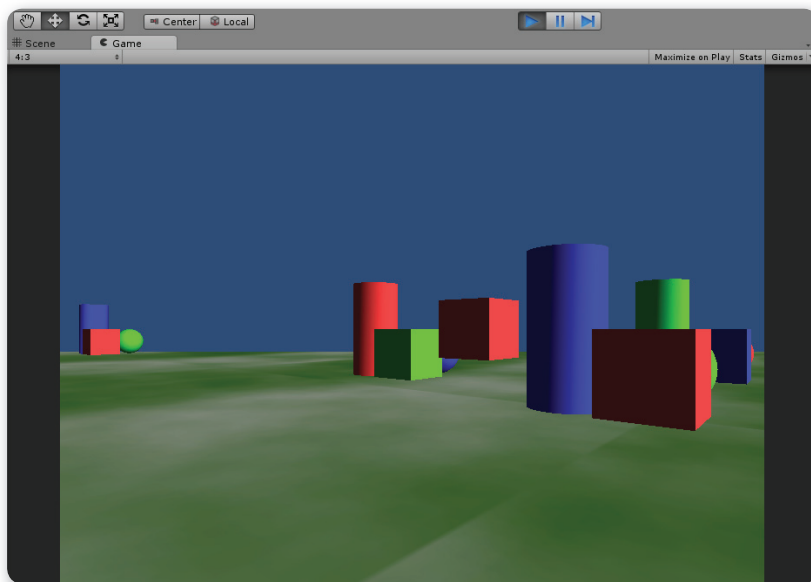
COMPONENTES CON LAS CÁMARAS



Cuando estamos creando nuestras propias cámaras, podemos agregar otros componentes de Unity dentro de un parenting, para así adquirir nuevas funcionalidades. Por ejemplo, podemos adicionar una luz, y de esta forma, generar una linterna. También podemos adicionar componentes para comportamientos de física, colisiones y otros elementos que necesitemos.

poder girar la cámara. La rotación es sobre el eje **Y**, y simplemente rotamos la cantidad dada por la velocidad angular y el tiempo transcurrido. La rotación se efectúa con el método **Rotate()**. De una manera similar, hacemos la rotación en el sentido contrario, para cuando se pulse la **flecha izquierda**.

Con esto ya podemos asignar el script a la cámara, sin olvidarnos de deshabilitar los scripts previos, para que no interfieran.



► **Figura 13.** Con la implementación del nuevo script podemos movernos dentro de la escena y recorrerla.

Usar los scripts de Unity

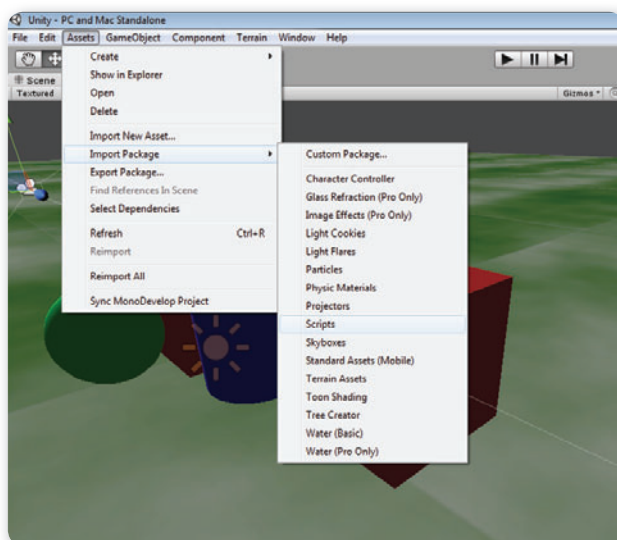
Hacer el código para una cámara como la de los videojuegos profesionales puede ser muy complejo. No solo necesitamos avanzar y girar la cabeza, sino que también hay que producir colisiones y saltos, agacharse, tomar carrera, etc. Esto es algo que podemos hacer programando, pero también tenemos la posibilidad de usar scripts que Unity provee por default. Hay un script para la cámara en primera persona, y otro para la cámara en tercera persona.

Una cámara en primera persona nos muestra la escena tal como si nosotros estuviéramos ahí viéndola con nuestros ojos, por eso su nombre. Lo que vemos es lo que ven los ojos del personaje que estamos controlando. En la cámara en tercera persona somos un espectador que sigue desde cierta distancia al personaje principal; podemos ver sus movimientos y algo de la escena alrededor de él.

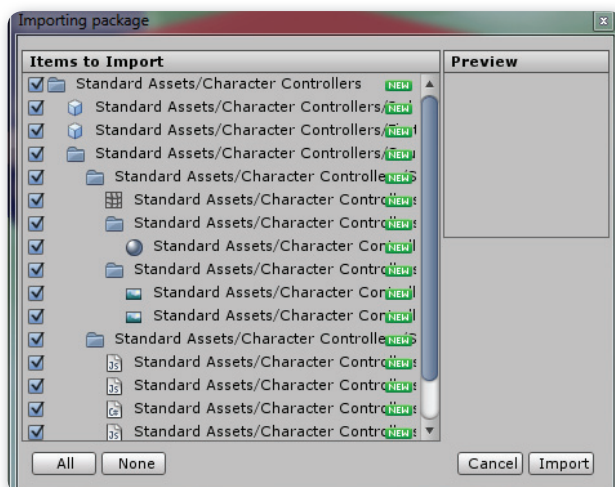
Importar los scripts

Comenzamos con la importación de scripts. Esto puede hacerse para todos los demás tipos de scripts, no solo para los controladores de cámaras. Para importar un script de Unity nos dirigimos al menú **Assets** y seleccionamos **Import Package**. Aparece un menú con las diferentes opciones de paquetes que podemos importar, de donde seleccionamos **Character Controller**.

Se presenta un cuadro de diálogo que nos permite seleccionar los elementos que deseamos importar; en este caso, dejamos todas las casillas de verificación seleccionadas y oprimimos **Import**.



► **Figura 14.** Por medio de los menús indicamos que vamos a importar el paquete con scripts para la cámara **Character Controller**.



► **Figura 15.** En el cuadro de diálogo dejamos todas las opciones seleccionadas y pulsamos el botón **Import**.

Unity empieza a importar el paquete con los scripts para la cámara y, luego de unos segundos, vemos que en el panel **Project** aparece una nueva carpeta llamada **Standard Assets**; dentro de ella hay otra llamada **Character Controller**, donde encontramos los controladores de cámara.

La cámara en primera persona

Para colocar la cámara en primera persona, simplemente tomamos el objeto que dice **First Person Controller** y lo arrastramos hasta la escena. Aparece una cápsula con una cámara arriba. Si ejecutamos la aplicación, veremos que podemos movernos e, incluso, girar la cámara con el

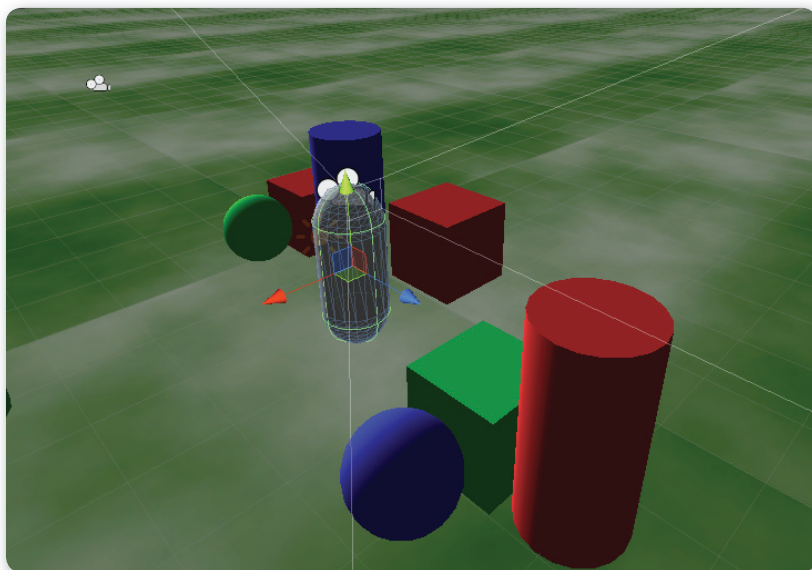


MEJORAR EL ASPECTO DEL ESPACIO



La cámara puede utilizar **skybox** en vez de un color de fondo. En este caso, se usa una serie de texturas para dibujar el fondo de la escena. Con ellas podemos representar paisajes lejanos u otros elementos que hagan lucir más natural el ambiente. Este tipo de texturas se pueden conseguir en Internet o, incluso, podemos hacerlas nosotros mismos.

mouse. No es posible atravesar los objetos que ya existen, pero podemos saltar pulsando la tecla **barra espaciadora**.



► **Figura 16.** El controlador de la cámara en primera persona tiene una cápsula y una cámara en la parte superior.

La cámara en tercera persona

En este primer paso, debemos eliminar el controlador de la cámara en primera persona del panel **Hierarchy**. Luego, arrastramos el objeto **3rd Person Controller** del panel **Project** a la escena. Movemos al personaje para pruebas que Unity provee con el editor y lo colocamos sobre el plano.

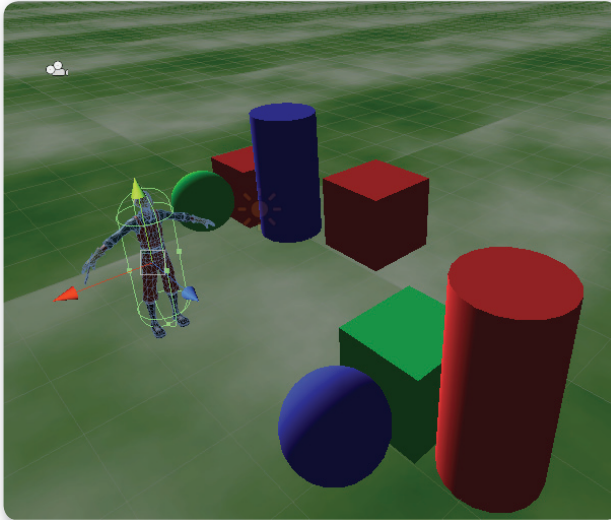


OTROS EFECTOS



Hemos visto cómo el uso de la apertura de campo sirve para crear un efecto de zoom. Si la cámara se mueve constantemente como en un automóvil, al ampliar la apertura de campo, se incrementa la sensación de velocidad. Una apertura de campo pequeña puede inducir la sensación de claustrofobia o de espacios pequeños. La experimentación nos llevará a lograr efectos muy interesantes.

Luego ejecutamos la aplicación y vemos al personaje en movimiento, seguido por la cámara.



► **Figura 17.** El personaje debe estar bien apoyado sobre el plano, y para controlarlo debemos usar las teclas de flechas del teclado.

Con esto ya conocemos el manejo de las cámaras y podemos proceder a hacer aplicaciones interactivas más interesantes. Ya sea programando nuestras propias cámaras o usando las de Unity, tenemos la capacidad de crear nuestros propios juegos.



RESUMEN

La cámara nos permite observar la escena, y tiene muchas propiedades que pueden servirnos para modificar la forma en que luce el juego y su desempeño. Es posible cambiar el color del fondo y, con la apertura de campo, indicar cuánto puede observar la cámara. El plano lejano nos permite limitar la distancia hasta donde ve la cámara. Por medio de traslaciones y rotaciones, movemos la cámara en la escena. Existen funciones que nos apoyan para seguir con la vista a otros objetos. Unity incluye dos componentes con controladores para las cámaras en primera y tercera persona.

Actividades

TEST DE AUTOEVALUACIÓN

- 1 ¿Para qué se usa la propiedad **backgroundColor**?
- 2 ¿Cómo se crea un objeto de la clase **Color**?
- 3 ¿Qué es el modo ortográfico?
- 4 ¿Qué propiedad controla el modo de la cámara?
- 5 ¿Qué es la apertura de campo?
- 6 ¿Cuál es la propiedad que controla la a apertura de campo?
- 7 ¿Qué es el plano lejano?
- 8 ¿Cuál es la propiedad que controla al plano lejano?
- 9 ¿Cómo se crea una propiedad que acepte un objeto?
- 10 ¿Cómo funciona el método **LookAt()**?
- 11 ¿Cómo funciona la cámara en primera persona?
- 12 ¿Cómo funciona la cámara en tercera persona?

ACTIVIDADES PRÁCTICAS

- 1 Haga el script para una cámara, que modifique la dirección con el mouse.
- 2 Haga el efecto de zoom oprimiendo solamente una vez la tecla **z**.
- 3 Realice el juego de la pista, pero para recorrerlo con una cámara programada por usted.
- 4 Haga un script que modifique el color del fondo tal como cambia el color del cielo durante el día, desde el amanecer hasta la noche.
- 5 Realice un laberinto y recórralo usando la cámara en primera y en tercera persona.



Creación de terrenos

Un aspecto muy importante de Unity es su capacidad para crear y renderizar terrenos, pues muchos de nuestros juegos pueden ocurrir en escenarios exteriores, y el hecho de que esta herramienta esté integrada nos ahorra mucho trabajo y mejora el desempeño de la aplicación. En este capítulo aprenderemos a utilizar el editor de terrenos.

▼ Los terrenos en Unity 128

Experimentar con los terrenos 128

Las herramientas de edición 131

▼ Creación del terreno

para un videojuego 139

▼ Resumen..... 151

▼ Actividades..... 152



Los terrenos en Unity

El editor de terrenos funciona de manera similar a un programa de dibujo en 2D y también al modelado con arcilla. Por medio de un conjunto de diferentes tipos de brochas, podemos modelar el terreno, ya que estas nos sirven para subir, bajar o nivelar las distintas partes de la superficie. Otras brochas se usan para dar textura; este es un factor fundamental para que el terreno se vea de forma natural. También es posible ubicar vegetación, como árboles y pasto, por medio del uso de brochas. Por último, podemos incorporar objetos de utilería en la escena, como piedras.

Experimentar con los terrenos

Para trabajar con los terrenos en esta sección, necesitamos crear un nuevo proyecto de Unity al que llamaremos **Capítulo5**. Durante

PODEMOS CONFIGURAR
EL TAMAÑO Y LA
RESOLUCIÓN DEL
TERRENO USANDO
SET RESOLUTION

este proceso, el programa muestra un cuadro de diálogo donde nos pregunta qué paquetes deseamos importar. En esta ocasión, a diferencia de lo que hicimos en proyectos anteriores, vamos a seleccionar la opción **Character Controller.package** y después continuaremos con la creación del proyecto.

Unity muestra luego un proyecto vacío en el cual solo encontramos la cámara. Vamos a crear un objeto de tipo terreno que contendrá todo el código necesario para poder ser editado

y renderizado. Para hacerlo, nos dirigimos al menú **Terrain** y, allí, seleccionamos la opción **Create terrain**.



EXPERIMENTAR CON LOS TERRENOS



En Unity la mejor manera de aprender a usar los terrenos es por medio de la experimentación de las diferentes herramientas. De este modo, obtendremos la habilidad necesaria para crear los terrenos destinados a nuestras aplicaciones. La edición de detalles finos requiere de cierta práctica. Al experimentar, también podemos descubrir técnicas que nos ayuden a desarrollar mejores trabajos con el tiempo.

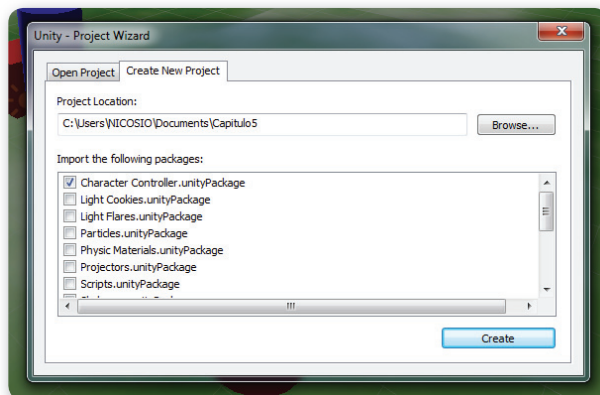


Figura 1.
Para utilizar la cámara en primera persona, es necesario importar el paquete seleccionado.

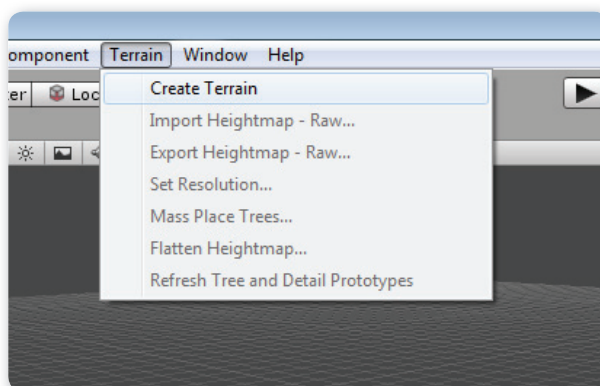


Figura 2.
Para crear el terreno debemos seleccionar el menú **Create Terrain**.

En el editor aparece un plano diferente del que hemos usado anteriormente, ya que es un terreno llano. Tenemos que hacer algunas configuraciones antes de empezar a editar, porque de lo contrario, será

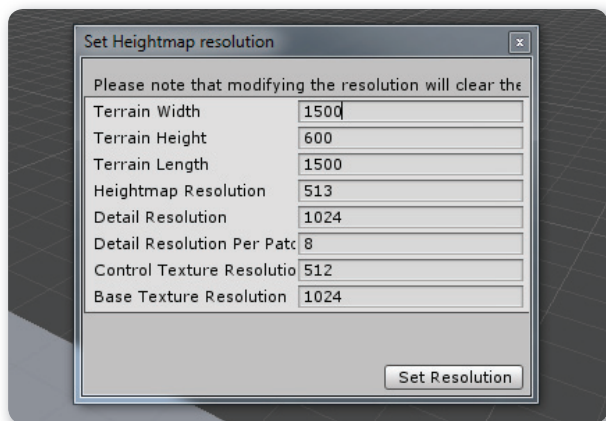


POSICIONAR LA CÁMARA



Al subir el terreno, es posible que la cámara ya no lo pueda ver. Esto se debe a que no es reposicionada de manera automática cuando el terreno sube, sino que debemos moverla manualmente. Es conveniente darle un valor en **Y** un poco mayor que la altura a la que subimos el terreno. Si no encontramos la cámara en el editor o el terreno, podemos usar la tecla **F** sobre la escena, una vez que seleccionamos el objeto en el panel **Hierarchy**.

más difícil adecuar el escenario a un tamaño determinado. Para esto, nos dirigimos al menú **Terrain** y seleccionamos **Set Resolution**.



► **Figura 3.** Aparece un cuadro de diálogo que permite configurar la resolución y el tamaño del terreno.

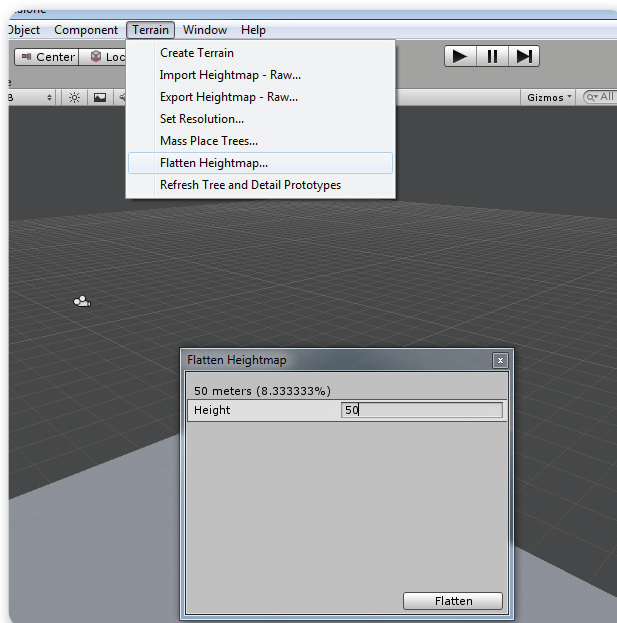
Al usar **Set Resolution**, se abre un cuadro de diálogo por medio del cual podemos configurar el tamaño y la resolución del terreno. La propiedad **Terrain Width** permite indicar el ancho que tendrá. La unidad de medida es equivalente al metro y en este ejemplo ponemos un valor de **1000**. En la propiedad **Terrain Length** colocamos el largo del terreno, en este caso, **1000**. Así tendremos un terreno de un kilómetro cuadrado,

que es lo suficientemente grande para nuestra experimentación. La propiedad **Terrain Height** indica la altura máxima que puede llegar a tener.

El aspecto del terreno puede ser modificado al elevar o bajar algunas zonas; sin embargo, Unity no puede colocar la altura por debajo de cero. Esto puede parecer un factor limitante, pero si subimos la altura del terreno, podremos luego bajar partes de él sin problemas. Para hacerlo, nos dirigimos otra vez al menú **Terrain**, pero en esta ocasión seleccionamos **Flatten Heightmap** y

colocamos el valor **50**. Esto quiere decir que el terreno estará elevado **50** metros por encima de **0** y, entonces, podremos bajar partes de él sin ningún inconveniente. Una vez modificada la altura, reposicionamos la cámara y continuamos trabajando.

EL ASPECTO DEL
TERRENO PUEDE
SER MODIFICADO
AL ELEVAR O BAJAR
ALGUNAS ZONAS



► **Figura 4.** Subir el terreno nos permite tener más libertad de edición, y es recomendable hacerlo antes de modelar cualquier sector.

Las herramientas de edición

Una vez que hemos creado la base del terreno e hicimos las modificaciones de altura necesarias, podemos empezar con el proceso de edición. Por el momento, vamos a usar tres herramientas y, más adelante, otras tres. Una de ellas cambia la altura del terreno, la otra lo nivela, y la última ayuda a suavizarlo. Junto con cada una tenemos



ESCRITURA DE SCRIPTS

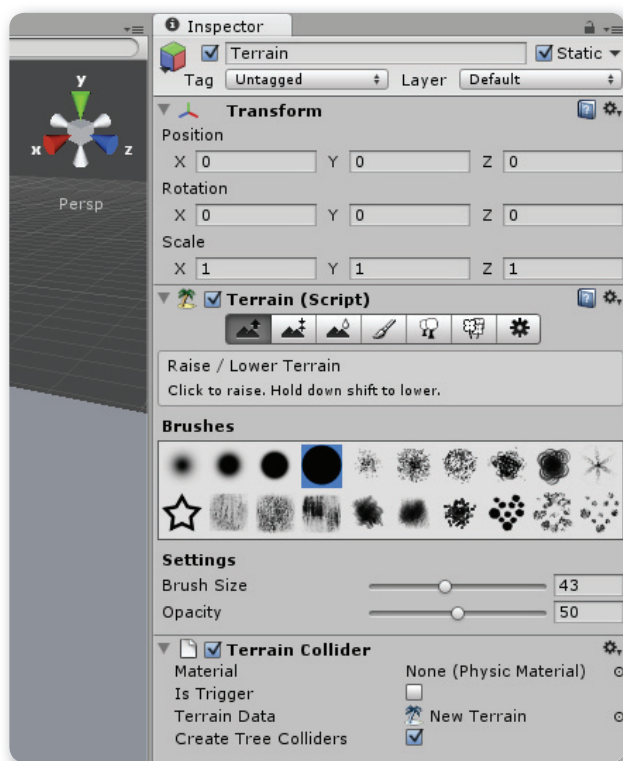


Las herramientas de edición para el terreno son un script que se ejecuta dentro de Unity; este es el que hace las modificaciones al terreno inicial. La escritura de scripts que se ejecuten en el editor de Unity es un tema más avanzado, pero es posible hacerlo, y permite adicionar funcionalidades y nuevas herramientas.

disponible un conjunto de brochas que nos permiten aplicar diferentes texturas. Las herramientas también tienen propiedades que modifican la manera en que trabajan.

Cuando seleccionamos una herramienta y apoyamos el cursor del mouse sobre el terreno, en el panel **Scene** vemos que el puntero del mouse presenta un área de color azulado. El puntero indica el lugar del terreno que se verá afectado, y la intensidad del color nos da idea de la fuerza de la deformación que se producirá. Este puntero también modifica su forma de acuerdo con la brocha que estemos utilizando y los valores de sus propiedades.

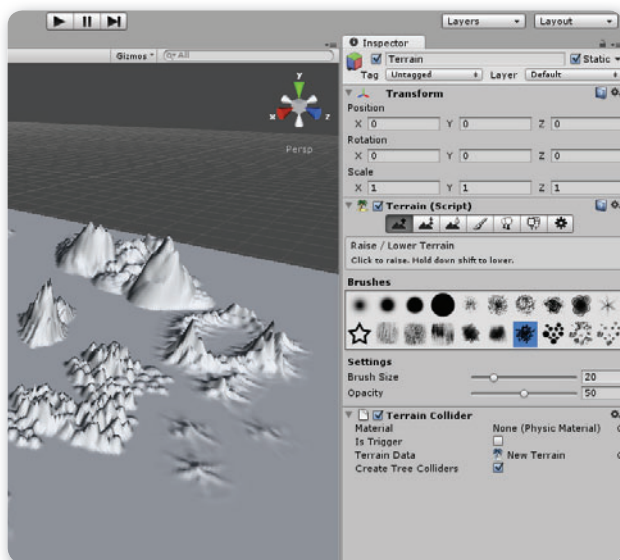
Hacemos clic en el terreno dentro del panel **Scene**. El panel **Inspector** muestra entonces las herramientas y todos los elementos necesarios para llevar a cabo la edición.



► **Figura 5.** En el panel **Inspector** encontramos las herramientas para edición de terrenos.

Subir y bajar terrenos

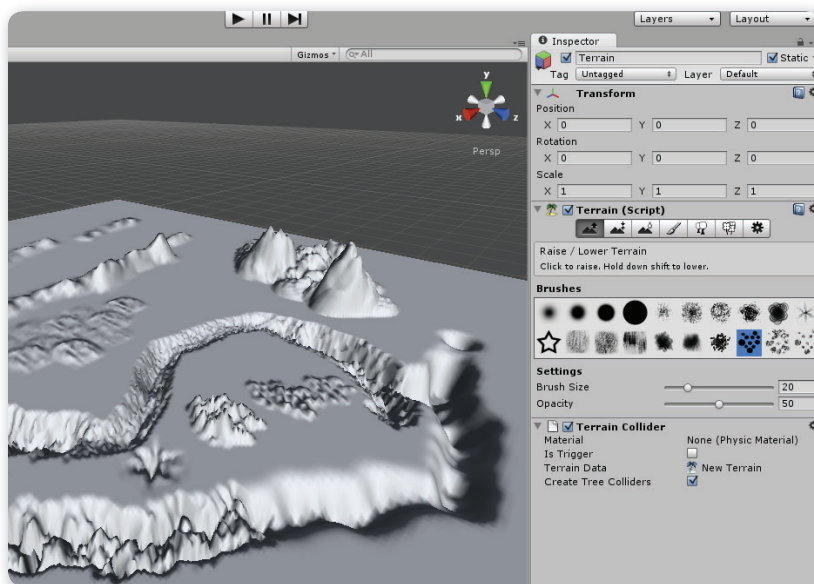
Empezaremos a usar la primera herramienta que nos permite modificar la altura de una parte del terreno. Para esto, seleccionamos el icono que tiene la figura de una montaña con una flecha que apunta hacia arriba. En la sección de brochas, elegimos la primera y pasamos el mouse sobre el terreno en el panel **Scene**. Para empezar a hacer que el terreno suba, debemos mover el mouse y oprimir el botón izquierdo al mismo tiempo. Si volvemos a pasar el mouse por esa zona manteniendo presionado el botón, el terreno continuará subiendo. Es conveniente intentarlo con diferentes brochas. De esta manera, podremos crear montañas y modificar la apariencia de su superficie.



► **Figura 6.** Con diferentes brochas obtenemos distintos tipos de modificaciones sobre el terreno.

Para lograr que el terreno baje, hacemos lo mismo, pero oprimimos al mismo tiempo la tecla **SHIFT**.

Al usar diferentes brochas, obtendremos distintas texturas en el aspecto del terreno. Con esta herramienta podremos crear cráteres, cañones, depresiones, etc. También puede resultar útil al momento de bajar la altura de alguna montaña o una parte de ella.



► **Figura 7.** Hacer depresiones es simple, y podemos incorporarlas a nuestra escena para que luzca más real.

La herramienta tiene dos propiedades: **Brush Size** (tamaño del pincel) y **Opacity** (opacidad). **Brush Size** indica el tamaño de la brocha elegida e influye directamente en la cantidad de terreno que se verá afectado. Como ejercicio, podemos ir recorriendo el valor de **10** en **10** y ver las diferentes formas que podemos obtener. El valor posible para el tamaño de la brocha va desde **1** hasta **100** y puede tener punto decimal. Usar brochas grandes permite trabajar más rápido, en tanto que usar las pequeñas brinda la posibilidad de editar con más detalle.



SIN CUEVAS



Al momento de escribir este libro, Unity solamente puede crear terrenos que no tengan concavidades, es decir que no podemos generar cuevas o formaciones geológicas similares. Si necesitamos este tipo de superficies, lo mejor es hacerlas en un software de modelado 3D y, luego, importarlas a Unity. Cualquier otro tipo de terreno es fácil de modelar.

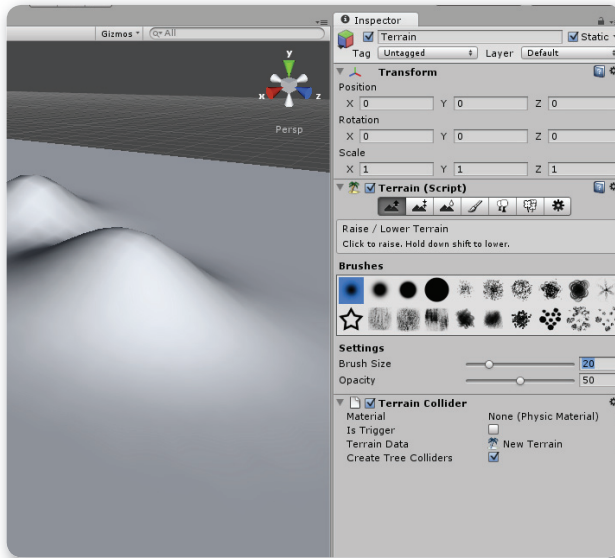


Figura 8. Esta imagen muestra el efecto que se produce al cambiar el tamaño de la brocha durante el proceso de edición del terreno.

Opacity, por otro lado, indica la fuerza que tiene el efecto sobre el terreno. Si el nivel de opacidad elegido es bajo, el terreno se modificará menos que si elegimos un nivel de opacidad mayor; en este caso los cambios serán más notables.

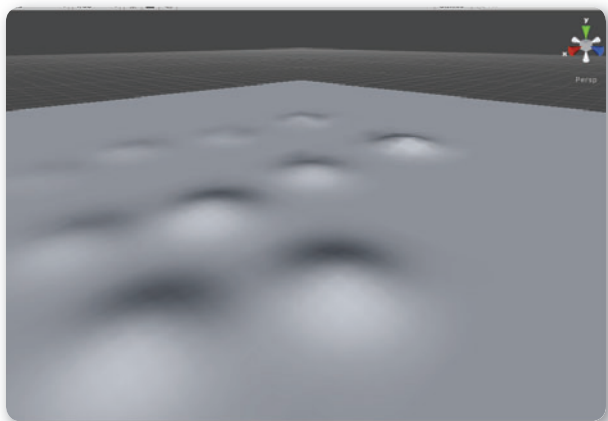
Ahora, como ejercicio, vamos a comprobar el resultado de mantener estable el tamaño de la brocha y cambiar el valor de la opacidad, que va de **0** a **100**. Vemos que con **0**, el efecto no se lleva a cabo, en tanto que con **100** obtenemos su máxima fuerza.



APLANAR TODO EL TERRENO



Si hemos llevado a cabo la modificación del terreno, pero en un momento vemos que no nos gusta y deseamos empezar otra vez, en lugar de tratar de aplanarlo a mano, la manera más simple de hacerlo es por medio de la herramienta **Flatten Heightmap**. Este modo es mucho más rápido y nos ahorra tiempo. Solo hay que colocar otra vez el valor de la altura.



► **Figura 9.**
Podemos observar la forma en que diferentes valores de opacidad afectan al terreno.

Pintar a una altura

La segunda herramienta que veremos nos permitirá llevar a cabo deformaciones del terreno. La diferencia que tiene con las demás es que permite colocar un límite a la altura: no es posible sobrepasar ese tope, por lo que se llega a generar una especie de plano. El control de la altura límite se logra por medio de la propiedad **Height**, en la que indicamos hasta dónde deseamos que llegue la brocha, en un rango de **0 a 600**. Vamos a seleccionar esta herramienta y a colocar diferentes valores para observar cómo se limitan las alturas.

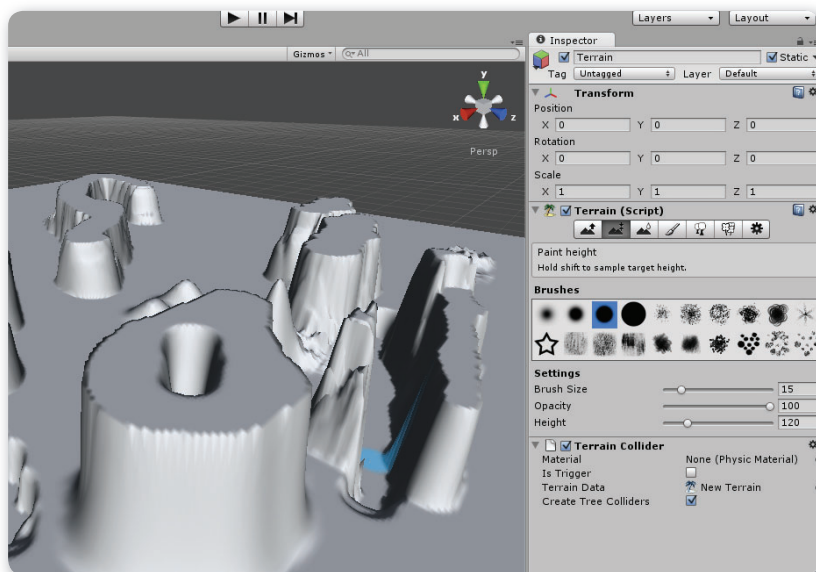
Como anteriormente hemos decidido colocar el terreno a una altura de **50** metros, si ahora deseamos que esté por arriba, debemos poner un valor mayor. En caso de que queramos hacer un agujero, los valores que ingresemos deberán ser menores que **50** con el objetivo de hacer que el terreno baje. Para experimentar, pongamos valores inferiores y veamos qué es lo que sucede al hacerlo.



CUIDAR LA ALTURA



Si al crear el terreno colocamos una altura determinada, esta será la referencia para que la herramienta de altura dibuje por arriba o por debajo de ese valor. En caso de que no luzca tan alto como debería, tendremos que verificar la diferencia entre la altura del terreno y la colocada en la propiedad.



► **Figura 10.** Las brochas nos permiten adecuar fácilmente la altura del terreno, para lograr el efecto deseado.

Suavizar el terreno

Los terrenos que construimos pueden tener diferentes texturas. Algunas veces nos enfrentamos con superficies que nos gustan en su forma general pero que son demasiado rugosas, y necesitamos suavizarlas. También puede suceder que, al ejecutar la aplicación, tengamos un terreno que resulta difícil de navegar y que la mejor opción sea nivelarlo ligeramente. Unity provee de una brocha para suavizado para realizar esta tarea sin tener que repetir todo el trabajo.



VOLCÁN Y FOSAS



Cuando pintamos con altura, es muy sencillo hacer elementos de terreno como volcanes. Primero generamos la montaña y, luego, pintamos con una altura menor, de manera de crear el cráter del volcán. Para hacer una fosa, colocamos un valor menor que la altura del piso alrededor de la fosa. Esta herramienta también es útil para generar cañones, o para colocar terrenos planos destinados a calles y otras estructuras.

EN UN TERRENO
DIFÍCIL DE NAVEGAR,
LA MEJOR OPCIÓN
PUEDE SER NIVELARLO
LIGERAMENTE

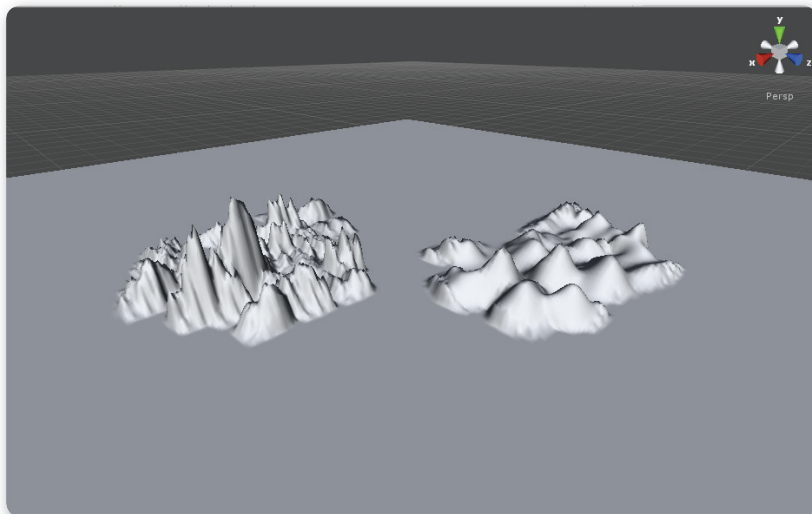


En el panel **Inspector** encontramos un icono con una figura de montañas y una gota de agua. Al pasar el puntero del mouse sobre el terreno y oprimir el botón izquierdo, este se suaviza de acuerdo con la textura de la brocha y sus propiedades.

En esta herramienta únicamente tenemos dos propiedades. **Brush Size** indica el tamaño de la brocha; cuanto más grande sea, mayor será el área sobre la que se realiza el suavizado. El rango

de valores para esta propiedad es de **1** a **100**. Por su parte, **Opacity** es equivalente a la que ya vimos con anterioridad. El valor que colocamos indica cuán fuerte se aplicará el efecto sobre el terreno. Su rango de valores varía desde **0** hasta **100**; los más pequeños suavizarán menos el terreno, y los más grandes lo suavizarán más.

Ahora podemos experimentar con esta propiedad. Creamos dos grupos de montañas con una textura que nos permita hacerlas escarpadas y, luego, con la herramienta de suavizado modificamos un grupo de ellas. El cambio es evidente.



► **Figura 11.** Podemos observar el cambio en la textura del terreno al hacer uso de la herramienta de suavizado.



Creación del terreno para un videojuego

Veamos ahora una metodología sencilla que puede servirnos para crear los terrenos de nuestros videojuegos. Empezamos tomando una hoja limpia sobre la cual dibujamos el mapa del terreno. Podemos trabajar con diferentes colores para dar una idea de las alturas que vamos a utilizar. También podemos indicar, por medio de marcas, algunas características en particular. Para este caso, haremos una isla y usaremos un mapa parecido al que se muestra en la **Figura 12**.

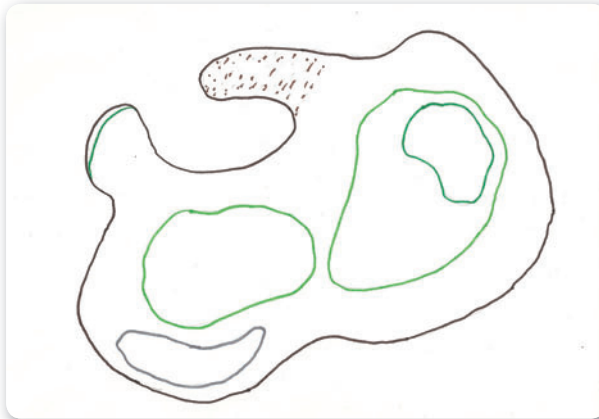


Figura 12.

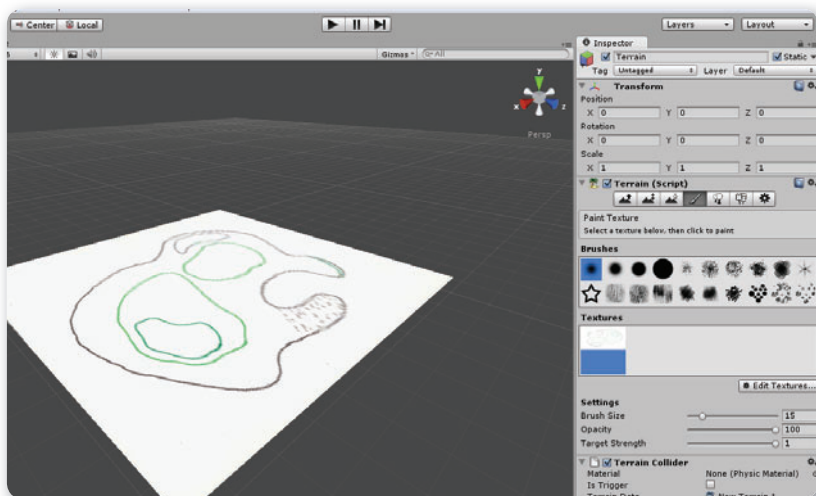
En este mapa vemos las características más importantes del terreno que deseamos crear.

El mapa representa una isla. Tiene dos montes, uno de ellos con una sección de más altura. En la parte superior habrá una depresión, y abajo, una pequeña bahía. Marcamos el terreno para indicar cierta zona de rugosidad. Ahora creamos una nueva escena, y seguimos los pasos ya vistos para generar el terreno y colocarlo a una altura de **50** unidades.

Vamos a utilizar la imagen del mapa como guía para la edición. Lo primero que hacemos es adicionar una luz de tipo **Directional Light**, que es útil para observar las variaciones del terreno. Luego importamos la imagen del mapa como textura. Para hacerlo, nos dirigimos al terreno y, en las herramientas, seleccionamos el botón que tiene el icono de una brocha. En la parte de abajo de las brochas veremos un botón con el icono de un engranaje. Al oprimirlo, aparece un menú donde

seleccionamos **Add Texture**. Inmediatamente se abre un cuadro de diálogo donde podemos seleccionar la textura que deseamos, que en este caso, es el mapa de la isla. En la primera propiedad, llamada **Splat**, hay un icono con forma de círculo; al hacer clic en él, aparecen las texturas que tiene el proyecto, entre las cuales seleccionamos el mapa. Ahora debemos indicar el tamaño que tiene la textura en el terreno. Deseamos que lo cubra totalmente, por lo que colocamos el valor **1000** en las propiedades **Tile Size X** y **Tile Size Y**. Oprimimos el botón **Apply** y veremos el mapa sobre el plano del terreno.

Como siguiente paso, vamos a subir el terreno de acuerdo con el mapa; la zona que pertenece al mar no se debe levantar. Para esto,



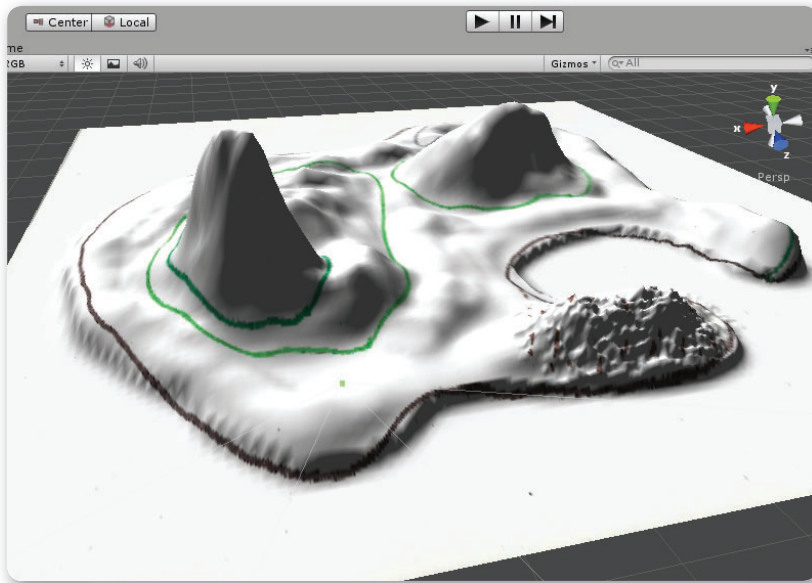
► **Figura 13.** El mapa se encuentra ahora en el terreno y nos sirve de guía para la edición de las alturas.



DESCARGA GRATUITA DE TEXTURAS

En ocasiones, necesitamos texturas gratuitas para nuestras aplicaciones. En muchos sitios no tienen costo para fines no comerciales, pero siempre es bueno verificar los términos de la licencia de uso. Un sitio que ofrece texturas muy bien clasificadas y que podemos usar para este caso es **cg textures** (www.cgtextures.com).

podemos usar todas las herramientas que ya conocemos. Al finalizar, tendremos un cuadro similar al de la **Figura 14**.



► **Figura 14.** El terreno fue editado según la idea general dada por el mapa.

Ahora buscamos en Internet tres texturas diferentes para aplicar a la isla: césped, roca y arena; con ellas le daremos más vida y detalle a la isla. Los nombres que les asignamos en Unity, una vez importadas, son: **Césped**, **Roca** y **Arena**, respectivamente.

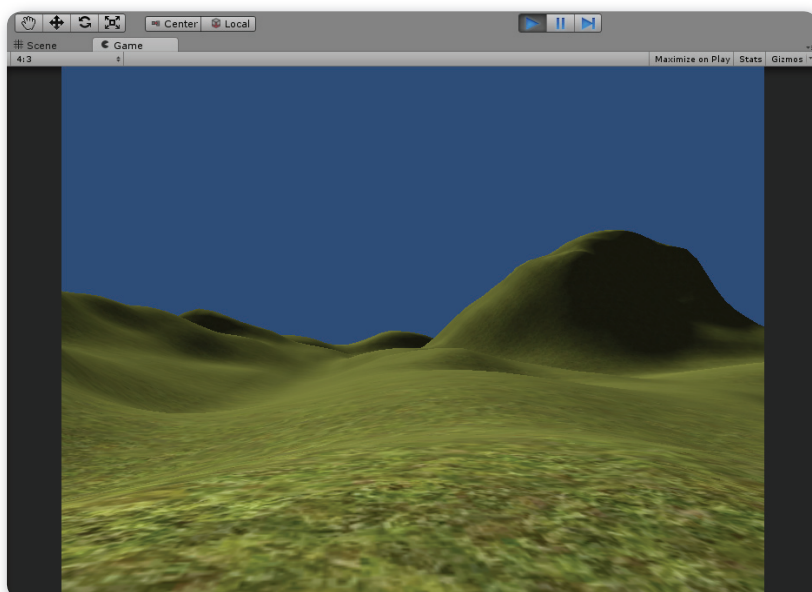
Vamos a hacer un cambio en la isla. La textura **Césped** será la de base, y **Roca**, junto con **Arena**, serán las texturas de detalle. Esto quiere decir que, en vez de tener el mapa, tendremos **Césped** como textura principal para el suelo de la isla. Este cambio se hace seleccionando la herramienta de pintar textura, la que tiene el icono de una brocha. En el botón de las texturas con el icono del engranaje seleccionamos **Edit Textures**. Cuando aparece otra vez el cuadro de diálogo, hacemos clic sobre el icono en forma de círculo. De la

POR MEDIO DE
MARCAS, PODEMOS
INDICAR ALGUNAS
CARACTERÍSTICAS
EN PARTICULAR



lista de texturas que tenemos, seleccionamos **Césped**. En las propiedades **Tile Size X** y **Tile Size Y** ponemos el valor **10**, que podremos modificar posteriormente para adecuar el tamaño a una vista natural en la isla.

Este es un buen momento para agregar un controlador de cámara en primera persona. Debemos ponerlo sobre la superficie de la isla, porque es necesario que podamos movernos por ella.

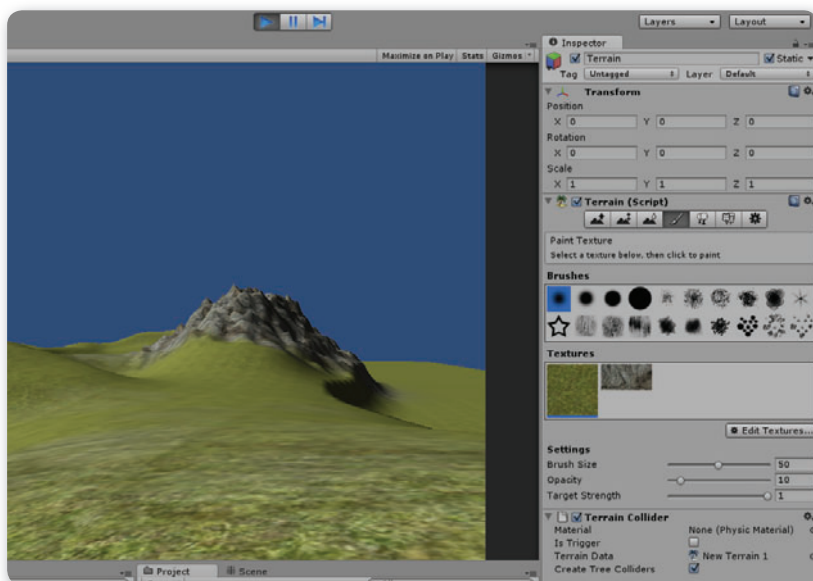


► **Figura 15.** Con el controlador en primera persona podemos recorrer la isla y ver las texturas.

Ya tenemos la textura base, ahora podemos colocar las de detalle. Estas serán pintadas sobre la base, y podemos ponerlas donde deseamos o donde necesitamos poner otra textura. Las texturas de detalle se pintan con las brochas, de manera muy similar a pintar con un programa de dibujo. Las texturas se mezclarán suavemente para dar una sensación de naturalidad. Podemos usar tantas texturas como sean necesarias, y es buena idea colocar algunas especiales en determinados lugares para ayudar a aumentar el realismo del juego. Cuando realizamos esta tarea, podemos usar las diferentes brochas para pintar, cada una de una manera diferente, y también es posible modificar las

propiedades para cambiar la forma en que la brocha actúa y así tener un control más preciso sobre el trabajo que estamos realizando.

En este momento vamos a agregar la textura **Roca** para pintar las cumbres de las montañas y la zona cercana a la bahía. Con la herramienta de pintar textura seleccionada, nos dirigimos al botón que tiene el icono en forma de engranaje y elegimos **Add Texture**. Aparece un cuadro de diálogo donde seleccionamos la textura **Roca**. Podemos dejar los valores de default de **15** unidades. Regresamos a las herramientas y



► **Figura 16.** Con más de una textura, la isla luce mucho mejor, y el ambiente se ve más real.



LA PINTURA DE DETALLE Y EL DESEMPEÑO

La pintura de detalle también puede disminuir el desempeño de la aplicación cuando se usa en exceso. Por motivos de rendimiento, Unity no presenta toda la pintura de detalle simultáneamente: solo presenta aquella que está a una determinada distancia. Por esta razón, es posible que, al editar la escena, no veamos la pintura de detalle donde pensamos que la estamos colocando. Solo hay que mover la cámara e ir editando zona por zona.

vemos que apareció esta segunda textura. Para usarla, debemos hacer clic. Seleccionamos una brocha que nos agrade y le damos un tamaño de **10** unidades. Luego procedemos a pintar esa textura en los lugares donde creamos que va a lucir bien.

Para la textura **Arena** procedemos del mismo modo y la colocamos en diferentes zonas de la isla, para acentuar el realismo. Con tres texturas, la isla se ve mucho mejor.

Colocar zonas con agua

La isla necesita estar rodeada de agua, y Unity incluye un paquete de agua básico. Para usarlo, lo importamos con el menú **Assets** y luego seleccionamos **Import Package**. Aparece la lista de paquetes, de la cual elegimos **Water (Basic)**; dejamos seleccionadas todas las opciones y procedemos a importar.

EL AGUA QUE UNITY
PROVEE ES DE NIVEL
BÁSICO Y ES MUY
ÚTIL PARA REALIZAR
COSAS SENCILLAS

El agua que Unity provee por default es de nivel básico y es muy útil para realizar cosas sencillas. Hay otros tipos de agua con mejores shaders que nos brindan la oportunidad de hacerla lucir de manera mucho más realista. Nosotros podemos programar esos shaders, obtener algunos con la versión 0, incluso, comprarlos de algún proveedor y adicionarlos a nuestro proyecto.

El paquete que hemos importado se encuentra en la carpeta **Standard Assets** del panel **Project**. Allí encontramos una nueva carpeta llamada **Water (Basic)**, dentro de la cual hay dos **prefabs** que podemos utilizar. Cada uno de ellos está diseñado para cambiar la manera en que se ve la iluminación del agua. Para que el agua luzca con iluminación diurna, usamos **Daylight Simple Water**, y para que luzca con iluminación nocturna, **Nighttime Simple Water**.

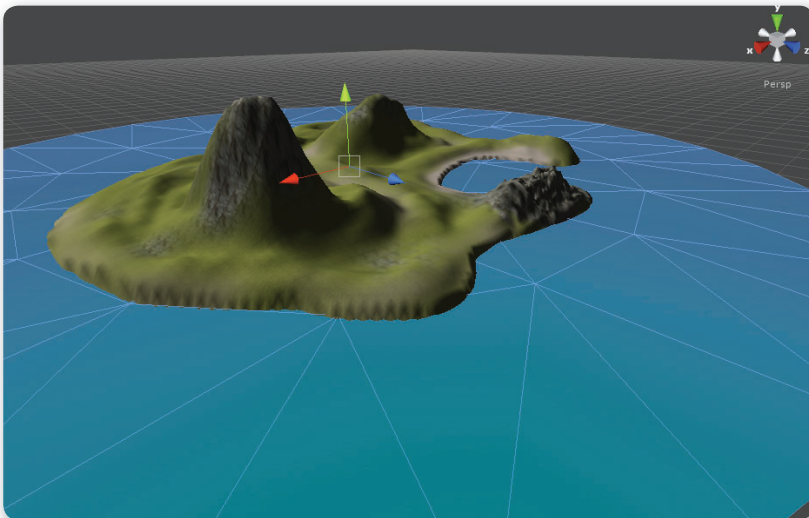


CONSEJOS PARA LA EDICIÓN



Una manera sencilla de hacer montañas es mantener oprimido el botón del mouse. Si lo hacemos repetidamente y de forma rápida, también se obtiene un aspecto diferente del terreno, que puede ser útil para crear formas irregulares. Cuando usamos distintas brochas en la misma montaña, logramos efectos interesantes.

Tomamos **Daylight Simple Water** y lo arrastramos a la escena. Aparece un círculo, en cuya superficie se renderiza el efecto del agua. Donde sea que lo coloquemos, tendremos un área que luce como agua. Podemos colocar tantos de estos objetos como necesitemos para mostrar este elemento. En nuestro caso, solo precisamos uno, pero es necesario hacerlo más grande y ubicarlo en un lugar adecuado. Colocamos el agua de manera centrada para que cubra toda la superficie alrededor de la isla, pero también es necesario darle una altura ligeramente mayor que la del terreno, para que esta se vea. Ubicamos el agua en la posición **(500,60,500)**, al igual que hacemos con



► **Figura 17.** Nuestra isla ya tiene agua alrededor, lo que aporta mayor realismo al videojuego.

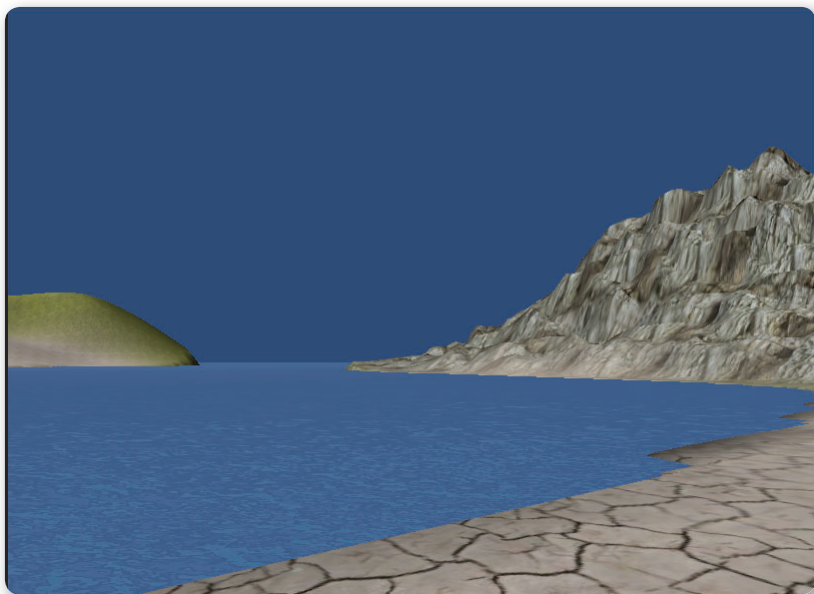


MODIFICAR EL AGUA



El agua que ofrece Unity tiene varias propiedades que podemos ajustar para lograr un mejor efecto de acuerdo con nuestras necesidades. Es posible cambiar el color y las texturas que se usan para el renderizado. Incluso, podemos variar la escala de las ondas y su velocidad. Esto nos ayuda mucho, pero necesitamos experimentar con estos valores para aprender a usarlos correctamente.

los demás objetos. En este momento, es muy pequeña y no se ve, por lo que vamos a modificar su escala. Al ser un plano, solo cambiaremos la escala en los ejes **X** y **Z**. Damos el valor **700** para la escala en **X**, **1** para la escala en **Y** y **700** también para la escala en **Z**. Ahora ya podemos ver el agua alrededor de la isla.



► **Figura 18.** El agua que hemos colocado en la escena luce muy bien vista desde la bahía.

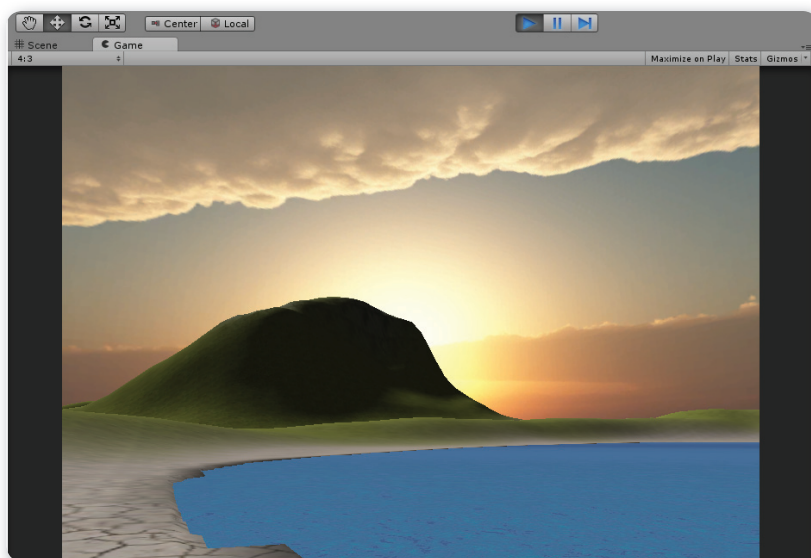
Adicionar un skybox

El **skybox** nos permite tener la ilusión de una escena que se extiende muy lejos. También nos ayuda a presentar una imagen para el cielo que cubre todos los lugares donde miramos. Se trata de un cubo que siempre se renderiza como fondo y tiene seis texturas. Desde el interior de ese cubo, las texturas se ven de manera continua, y hacia donde giremos la cámara, veremos el cielo o el paisaje que represente.

Para hacer uso del skybox debemos empezar por importar el paquete que lo contiene. Nos dirigimos al menú **Assets** y seleccionamos **Import Package**; luego, de la lista de opciones seleccionamos **Skyboxes** y dejamos

marcadas todas las opciones que presente el cuadro de diálogo. Al terminar la importación, encontraremos la carpeta **Skyboxes** adentro de la carpeta **Standard Assets** del panel **Project**.

Para colocar el skybox, vamos al menú **Edit** y seleccionamos **Render Settings**. En el panel **Inspector** aparecen las propiedades de renderizado, una de las cuales se llama **Skybox Material**. Lo que debemos hacer ahora es arrastrar uno de los skyboxes que se encuentran en la carpeta **Skyboxes** hacia la propiedad **Skybox Material**. Si no se muestra nada en el panel **Scene**, no hay problema. Simplemente, ejecutamos la aplicación y este debe aparecer.



► **Figura 19.** El uso de **Skybox** incrementa mucho el realismo de la escena y ayuda a definir el humor general del nivel.



EL SKYBOX EN LA ESCENA



Es posible que el skybox no aparezca en el panel **Scene** una vez que lo hayamos adicionado a las propiedades de render. Para poder observarlo, debemos hacer clic en el icono del panel **Scene** que luce como una foto donde se ven montañas. De este modo se verá el skybox. Podemos ocultarlo haciendo clic otra vez.

Agregar elementos al terreno

Para que el terreno se vea mejor y más realista, debemos añadirle árboles, rocas, hierbas y otros objetos. Estos elementos pueden ser creados e importados a Unity, o podemos hacer uso de los que incorpora el software. Para este caso, vamos a importar un nuevo paquete llamado **Terrain Assets**, que contiene ejemplos de hierbas y una palmera. Esto se hace de la forma tradicional. Vamos al menú **Assets** y elegimos **Import Package**. De la lista de opciones seleccionamos **Terrain Assets** y dejamos

marcadas todas las opciones disponibles. Después de importarlo, aparecerá una carpeta llamada **Terrain Assets** dentro de la carpeta **Standard Assets** del panel **Project**.

El agregado de árboles es sencillo, y para hacerlo necesitamos una herramienta que trabaja con brochas. Vamos al terreno y seleccionamos la que tiene el icono que luce como un par de árboles. Luego, nos dirigimos al botón con el icono en forma de engranaje y seleccionamos **Add Tree**. Aparece un cuadro de

LA PINTURA DE
DETALLE NOS PERMITE
ADICIONAR HIERBA,
ROCAS Y OTRO TIPO
DE OBJETOS



diálogo en el que hacemos clic en el icono que tiene forma de círculo. Allí vemos las geometrías que podemos usar para los árboles; en este caso, seleccionamos la palmera.

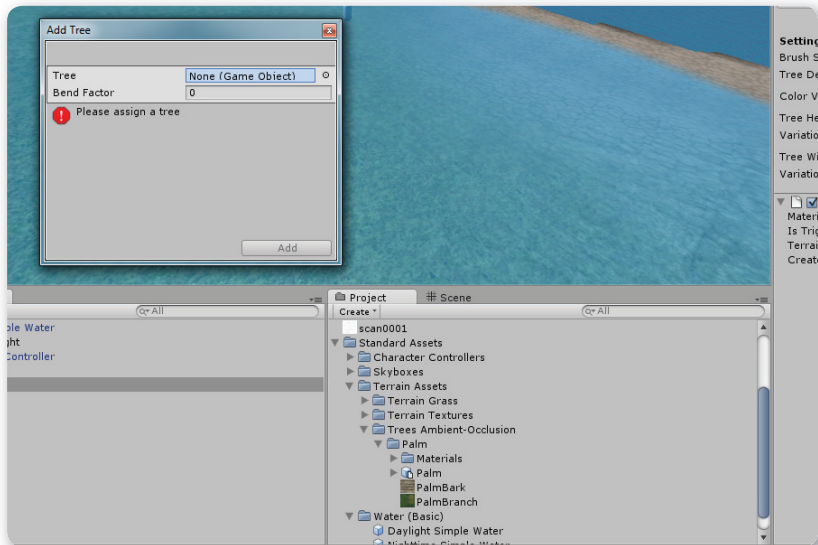
Si cuando hacemos clic en el icono, la palmera no aparece en la lista de objetos, tendremos que añadirla manualmente. En ese caso, cerramos el cuadro de diálogo en donde se muestran las geometrías y, sin el cuadro **Add Tree**, vamos al panel **Project** y buscamos la carpeta **Palm** dentro de la carpeta **Trees Ambient-Occlusion**, que se encuentra en **Terrain Assets**. Allí veremos un prefab llamado **Palm**; lo arrastramos hasta la propiedad **Tree** del cuadro de diálogo **Add Tree**.



MEJORAR EL DESEMPEÑO



El hecho de pintar árboles hace lucir bien el terreno, pero tiene una desventaja: puede reducir el desempeño de la aplicación. Si la geometría de los árboles es compleja y tenemos muchos, el programa tal vez muestre un desempeño menor. Lo mejor, entonces, es usar los árboles de una forma planificada y evitar los excesos. De esta manera, tendremos un terreno realista sin perjudicar el rendimiento.



► **Figura 20.** En caso de tener problemas, podemos adicionar la palmera manualmente.

Una vez que agregamos la palmera, podemos proceder a plantar palmeras por toda la isla. Dentro de las propiedades de esta herramienta encontramos una conocida como **Tree Density**, que indica la densidad de árboles que vamos a plantar. Las demás propiedades nos permiten controlar las dimensiones de los árboles así como sus variaciones; estas últimas son valores para modificar el alto o el ancho de los árboles dentro de cierto rango, de modo que no todos se vean iguales. También podemos aplicar variaciones en el color.

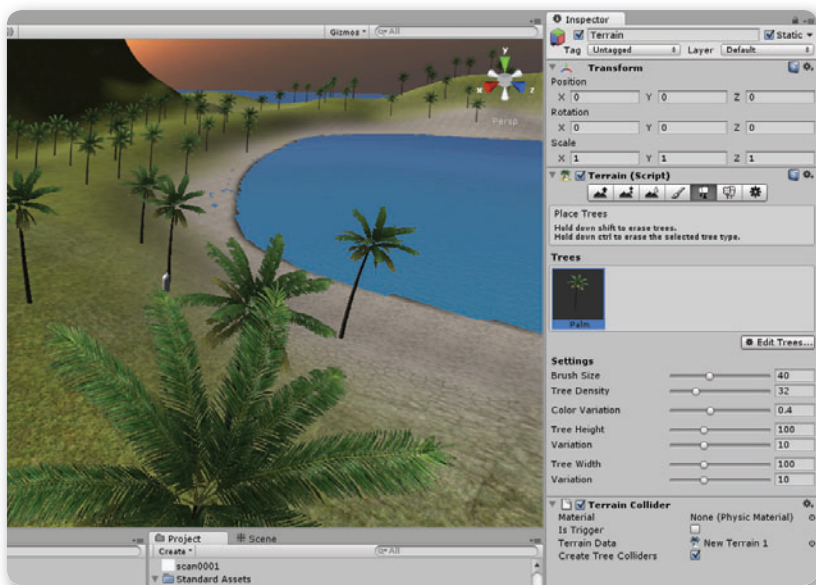
La pintura de detalle nos permite adicionar hierba, rocas y otro tipo de objetos. Para usar esta opción en el terreno, seleccionamos la



EDITORES DE TERRENO



Existen editores de terreno que podemos utilizar para crear superficies más avanzadas y que es posible importar desde Unity. Este tipo de editores tienen generadores automáticos, herramientas de erosión y muchas otras características. Un ejemplo de ellos es **Bryce**.



► **Figura 21.** Podemos plantar palmeras por toda la isla, con diferentes densidades.

LOS AMBIENTES
EXTERIORES SON
POPULARES Y BRINDAN
UN MAYOR REALISMO A
NUESTROS JUEGOS

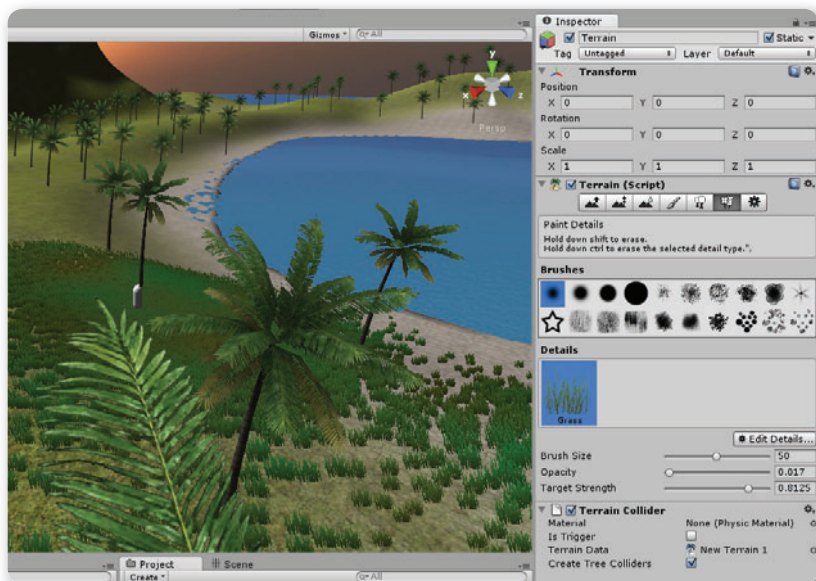


herramienta que tiene un icono con flores. La manera de usarla es muy similar a la forma en que utilizamos las otras herramientas. Debemos

agregar así la textura que necesitaremos para la hierba. Para hacerlo, nos dirigimos al botón con el icono que parece un engranaje y seleccionamos **Add Grass Texture**. Aparecerá un cuadro de diálogo, donde procederemos a hacer clic en el icono con forma de círculo. Al ver las texturas disponibles, buscamos la que se llama **Grass** o **Grass2**. Luego, simplemente dibujamos la hierba sobre el terreno del modo que más nos guste. Esta herramienta también tiene propiedades que nos ayudan a ajustarla

y configurarla según necesitemos.

Los ambientes exteriores son siempre muy populares y brindan un mayor realismo a nuestros juegos, permitiendo darle al jugador más opciones y variar los espacios donde los juegos se llevan a cabo.



► **Figura 22.** Con la pintura de detalle podemos colocar hierba y otros elementos en la escena.

Una buena planificación siempre es necesaria para que el terreno sea realmente funcional.

Hemos aprendido las bases de la creación de terrenos en Unity y sus herramientas. Ahora ya podemos tener niveles que luzcan mucho mejor y sean más sofisticados.



RESUMEN

En este capítulo aprendimos a crear terrenos y usar las herramientas de edición. Unity provee todo lo necesario para realizar esta tarea. Debemos crear el terreno del tamaño necesario y colocarlo a una altura que sea fácil de modificar. A partir de eso, podemos usar la herramienta para cambiar la altura, ya sea elevando o disminuyendo la altitud. Esta y las demás herramientas trabajan con brochas. Otra herramienta nos permite elevar el terreno a una altura en particular. Si este se ha vuelto muy rugoso, podemos suavizarlo. La primera textura se coloca en todo el terreno, pero podemos usar las texturas de detalle, que son pintadas en diferentes lugares. También es posible colocar agua y el **skybox** mejora mucho la calidad del escenario al presentar un paisaje a la distancia. Podemos pintar árboles y objetos en detalle, como hierba o rocas.

Actividades

TEST DE AUTOEVALUACIÓN

- 1 ¿De qué manera se coloca el tamaño del terreno?
- 2 ¿De qué forma se indica la altura del terreno?
- 3 ¿Qué ventaja da colocar el terreno a una altura determinada?
- 4 ¿Qué herramienta se usa para elevar al terreno?
- 5 ¿Qué tecla se usa para bajar la altura del terreno?
- 6 ¿Cuántos tipos de brochas existen?
- 7 ¿Qué hacen las propiedades de las brochas?
- 8 ¿Cómo se puede elevar el terreno a una altura específica?
- 9 ¿Qué se puede hacer para suavizar el terreno?
- 10 ¿Cómo se usan las texturas de detalle?
- 11 ¿Cómo se colocan árboles?
- 12 ¿Cómo se colocan hierbas?

ACTIVIDADES PRÁCTICAS

- 1 Diseñe un nuevo nivel que ocurra en un cañón desértico.
- 2 Diseñe un nuevo nivel que ocurra en un valle.
- 3 Coloque lagos que se encuentren en diferentes alturas.
- 4 Modifique el nivel para que transcurra en la noche.
- 5 Intente crear un acantilado.

Las luces

Durante este capítulo conoceremos los distintos tipos de luces y sus aplicaciones prácticas, y aprenderemos sobre los conceptos más importantes para su adecuado control. Las diversas luces posibilitan una iluminación variada en cada caso. Una buena combinación en la aplicación de sus diferentes tipos y el aprovechamiento de los efectos disponibles contribuyen a mejorar la iluminación y, por ende, la experiencia del jugador. Por el contrario, una iluminación incorrecta puede romper el encanto del juego.

▼ La iluminación	154
Crear la escena	154
Point Light.....	156
Directional Light.....	166
Spotlight.....	173

Ambient Light.....	180
--------------------	-----

▼ Resumen.....	181
----------------	-----

▼ Actividades.....	182
--------------------	-----



La iluminación

Unity permite realizar un manejo avanzado de la iluminación, y en su versión profesional podemos sacar provecho de esta posibilidad. Para hacer uso de los sistemas de iluminación avanzados es necesario conocer las matemáticas de los modelos de iluminación y programar los shaders de manera adecuada. Afortunadamente, también tenemos un sistema sencillo que nos permite hacer cosas muy interesantes y que, en muchas ocasiones, es suficiente para una gran parte de los proyectos.

De forma básica, trabajamos con cuatro tipos de luz: **Ambient Light**, **Point Light**, **Directional Light** y **Spotlight**. Cada una tiene ciertas características que la hacen apropiada para un efecto en especial. En los capítulos anteriores hemos trabajado con luces, pero ahora es el momento de conocerlas más a fondo.

Crear la escena

Vamos a crear una escena sencilla, que nos permita observar la manera en que trabajan las luces disponibles. Empezamos por generar un nuevo proyecto al que llamamos **Capítulo6**, y guardamos la escena con el nombre **Ejemplo1**. Esta consta de un plano y una serie de cubos, que tendrán diferentes colores para poder observar el modo en que el tono de la luz altera la forma en que percibimos el del material. También habrá dos esferas que nos permitirán observar mejor el efecto del sombreado; este puede cambiar dependiendo de la dirección o la posición de la luz.

Para poder observar bien el comportamiento de las luces, lo mejor es tener diferentes objetos y varios materiales. Empezamos por crear seis: **MaterialAmarillo**, **MaterialAzul**, **MaterialBlanco**, **MaterialRojo**, **MaterialVerde** y



LOS COLORES DE LA LUZ



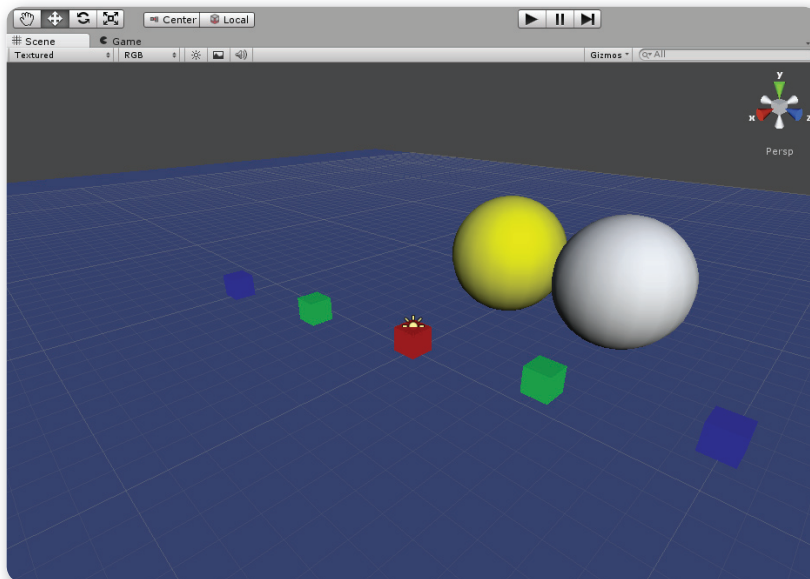
La luz tiene un color formado por tres componentes, al igual que el material. El color final se calcula componente por componente, viendo cuánto aportan a él tanto la luz como el material mismo. Una vez obtenido el color, puede ser escalado dependiendo de la cantidad de iluminación que recibe el vértice, polígono o píxel en particular. La técnica puede variar dependiendo del modelo de iluminación y el código correspondiente al shader.

MaterialSuelo. Los materiales que llevan nombre de color tienen ese tono en particular. Para el **MaterialSuelo** podemos colocar uno formado por diferentes componentes de color, como violeta o cian.

Con los materiales listos, procedemos a colocar los objetos. Creamos un plano llamado **Piso** y lo ponemos en la posición **(0,0,0)**. Es necesario escalar este plano de modo que cubra un área suficientemente grande para la escena. Por lo tanto, le colocamos una escala de **10** en los ejes **X** y **Z**; el **Y** puede permanecer con el valor **1**. El material **MaterialSuelo** es asignado al objeto **Piso**.

Generamos un cubo y lo ubicamos en la posición **(2,0.5,-10)**; luego, le asignamos **MaterialAzul** como material. El siguiente cubo se encuentra en **(1,0.5,-5)** y como material deber llevar a **MaterialVerde**. Agregamos otro más con **MaterialRojo** y lo colocamos en **(0,0.5,0)**. Se necesita un cuarto cubo en la posición **(-1,0.5,5)** con **MaterialVerde** asignado, y otro que lleva **MaterialAzul** y está en **(-2,0.5,10)**.

Ahora precisamos dos esferas. La primera se coloca en **(-6,2.5,0)**, lleva una escala de **5** en todos los ejes y **MaterialAmarillo**. La segunda también



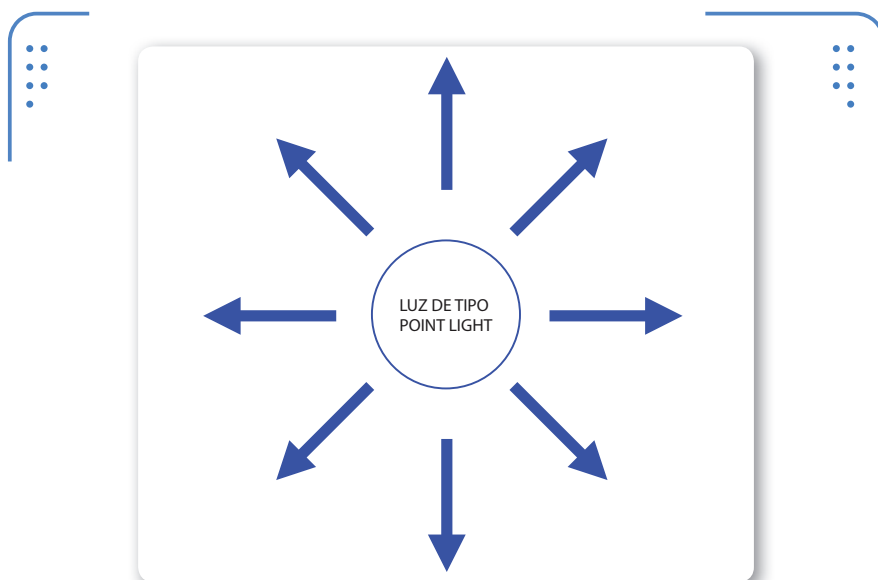
► **Figura 1.** El escenario de prueba para la iluminación está listo para ser utilizado en nuestro proyecto.

posee una escala de **5**, pero con **MaterialBlanco**, y tiene una posición cercana a la anterior, en **(-6,2.5,5)**.

La cámara se ubica en la posición **(21,3.3,0)**. Es necesario rotarla para ver el conjunto de los objetos; el valor de rotación que le colocamos es **(0,90,0)**. Para notar bien la iluminación en este ejemplo, es preciso establecer el color negro de fondo, lo cual puede hacerse directamente en el editor de Unity usando el panel **Inspector**.

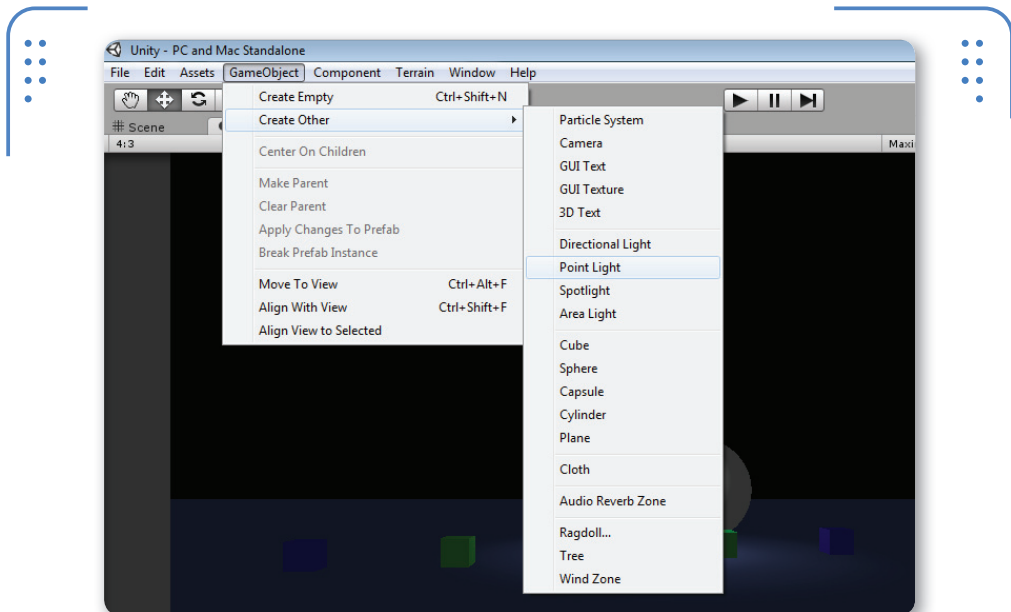
Point Light

La primera luz que vamos a conocer es muy sencilla y se conoce como **Point Light** (luz de punto). Para entender su comportamiento, podemos imaginar una vela que produce luz y cuyos rayos se emiten en todas las direcciones. La iluminación de los objetos depende de la posición de la vela: si la movemos, los objetos se iluminan de forma diferente.



► **Figura 2.** La luz de tipo **Point Light** emite los rayos en todas las direcciones del espacio en nuestra escena.

Procederemos a crear entonces una luz de tipo **Point Light**. Para hacerlo, nos dirigimos al menú **GameObject**, seleccionamos **Create Other** y,



► **Figura 3.** Por medio del menú **GameObject** creamos la luz de tipo **Point Light**.

de las opciones que ofrece elegimos la que dice **Point Light**. Al terminar, podremos observar mediante un pequeño experimento, la manera en que esta luz trabaja y se desempeña.

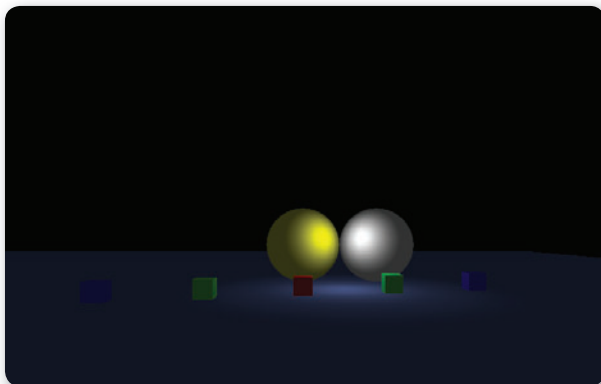
Para realizarlo, sencillamente colocamos una luz entre las esferas. Una vez que lo hayamos hecho, podremos observar cómo la luz se distribuye en todas direcciones. La posición que elijamos deberá ser **(-2,3.5,2.5)**, y el valor de la propiedad **Range** deberá ser **10**.



LA ILUMINACIÓN Y EL FONDO

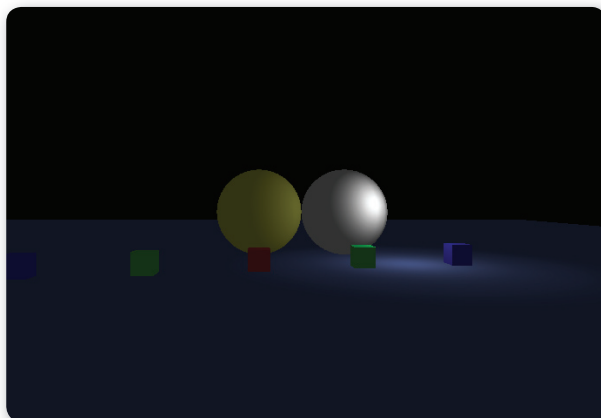


Si en nuestra escena usamos el fondo vacío de Unity, debemos tener cuidado con su color. Uno que no esté de acuerdo con el tipo de iluminación que estamos empleando puede romper con la armonía de la escena y hacer que la aplicación luzca mal. Por lo general, el color del fondo puede ser un tono oscuro de la luz más significativa de la escena. El fondo totalmente negro también funciona bien en la mayoría de los casos.



► **Figura 4.**
Esta es la primera posición de la luz, y aquí vemos cómo se distribuye entre las esferas.

Ahora vamos a mover la luz y ver el efecto que produce en la escena. La nueva posición es **(-2,3.5,7.5)**. La luz se encuentra a la derecha, y las esferas están iluminadas de ese lado. Con esto, comprobamos que la posición es muy significativa para este tipo de luz. Podemos experimentar con la rotación y descubrir que este parámetro no afecta a este tipo de luz.



► **Figura 5.**
Al mover la luz de tipo **Point Light**, cambia la forma en que la escena se ilumina. La rotación no tiene efecto sobre ella.

Empecemos a codificar un script para manejar de forma programática las propiedades más importantes de este tipo de luz. La primera con la que trabajaremos es **Color**; esta controla el color que proyecta la luz, y la manejamos de forma similar a como hicimos con el color del fondo.

Creamos un script y lo llamamos **LuzPunto**. El código inicial es el siguiente:

```
#pragma strict

// Este script modifica las propiedades de la luz punto

function Start () {

}

function Update () {

    // Modificamos el color de la luz
    if(Input.GetKeyDown(KeyCode.Alpha1))
    {

        // Colocamos un valor al azar a cada componente de color
        var r: float = Random.value;
        var g: float = Random.value;
        var b: float = Random.value;

        // Creamos una variable de tipo Color y le asignamos los valores
        var micolor: Color = new Color(r,g,b);

        // Asignamos el color a la luz
        this.light.color=micolor;

    }

}
```

El código principal se encuentra dentro de la función **Update()**. Para empezar, verificamos si se ha oprimido la tecla **1** del teclado alfanumérico. Luego creamos tres variables, que representan los componentes de color, y les asignamos valores aleatorios. A continuación, creamos una variable de tipo **Color** llamada **micolor**, a la que le asignamos esos componentes. Para finalizar, asignamos el color al color de la luz. La luz de tipo **Point Light** es un objeto que tiene varios componentes. Uno de ellos es de tipo **Transform** y otro es de tipo **Light**.

Por esta razón, para acceder a la propiedad **Color** primero tenemos que acceder al componente **Light** del objeto que tiene el script. Una vez finalizado este código, lo asignamos a la luz **Point Light**. Al ejecutar el programa y oprimir la tecla **1**, obtendremos un color al azar para la luz. Esta es una buena oportunidad para ver cómo el color de la luz modifica el que es percibido en el material. La esfera blanca es la que mejor representa el color que tiene la luz en ese momento.

La siguiente propiedad con la que vamos a trabajar es **Intensity**, que controla la intensidad de la luz.

Vamos a adicionar código para controlar la intensidad. Con la tecla **3** podemos incrementarla, y con la **4**, disminuirla. El código queda de la siguiente forma:

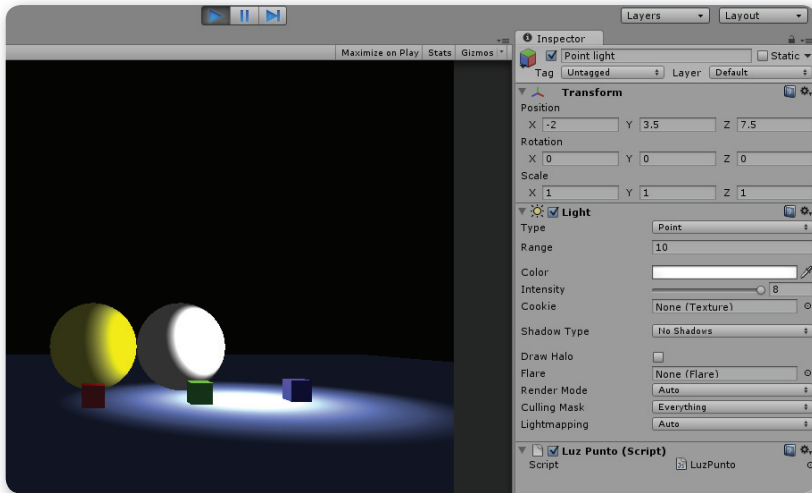
```
function Start () {  
  
}  
  
function Update () {  
  
    // Modificamos el color de la luz  
    if(Input.GetKeyDown(KeyCode.Alpha1))  
    {  
        // Colocamos un valor al azar a cada componente de color  
        var r: float = Random.value;  
        var g: float = Random.value;  
        var b: float = Random.value;  
  
        // Creamos una variable de tipo Color y le asignamos los valores  
        var micolor: Color = new Color(r,g,b);  
  
        // Asignamos el color a la luz  
        this.light.color=micolor;  
    }  
  
    // Modificamos la intensidad de la luz  
    if(Input.GetKeyDown(KeyCode.Alpha3))  
    {  
        
```

```
// Incrementamos la intensidad de la luz
this.light.intensity+=0.1;

}

if(Input.GetKeyDown(KeyCode.Alpha4))
{
    // Bajamos la intensidad de la luz
    this.light.intensity-=0.1;
}

}
```



► **Figura 6.** Podemos observar la intensidad de la luz con su máximo valor. Así se obtiene una saturación de color.



CONSIDERACIONES SOBRE LA INTENSIDAD



Hay que tener cuidado con los valores de la intensidad, porque en algunos casos, si es muy alta, el color se puede saturar y producir efectos no deseados. El valor de la intensidad va desde 0 hasta 8, y admite decimales. La intensidad predefinida es 1 y, en ocasiones, es aconsejable incrementarla ligeramente.

El código es muy sencillo: cuando se oprime la tecla **3**, el valor que tenga la intensidad se incrementa en **0.1**, y cuando se oprime la tecla **4**, disminuye en **0.1**.

La próxima propiedad que vamos a aplicar es **Range** (rango) e indica la distancia a la que puede iluminar esa luz en particular. Con ella podemos controlar muy bien el volumen de espacio que ilumina. Cualquier objeto dentro de este volumen será iluminado por la luz, en tanto que los que están por fuera de él no serán iluminados. En el caso de la luz **Point Light**, el volumen tiene la forma de una esfera. La propiedad puede tener un valor numérico de tipo **float**.

Ahora vamos a adicionar código a nuestro script:

```
#pragma strict

// Este script modifica las propiedades de la luz de tipo Point Light

function Start () {

}

function Update () {

    // Modificamos el color de la luz
    if(Input.GetKeyDown(KeyCode.Alpha1))
    {

        // Colocamos un valor al azar a cada componente de color
        var r: float = Random.value;
        var g: float = Random.value;
```



OPTIMIZAR LA ILUMINACIÓN



El cálculo de iluminación puede ser muy intensivo y, por eso, es necesario optimizar el cómputo para que la aplicación se ejecute lo mejor posible. Una de las formas de hacerlo es limitar la distancia a la que puede iluminar la luz. De esta manera, no se pierde tiempo calculando la iluminación de objetos que, de todas formas, recibirían muy poca iluminación. **Range** puede utilizarse para adquirir más control sobre la iluminación y mejorar el desempeño.

```
        var b: float = Random.value;

        // Creamos una variable de tipo Color y le asignamos los valores
        var micolor: Color = new Color(r,g,b);

        // Asignamos el color a la luz
        this.light.color=micolor;

    }

    // Modificamos la intensidad de la luz
    if(Input.GetKeyDown(KeyCode.Alpha3))
    {

        // Incrementamos la intensidad de la luz
        this.light.intensity+=0.1;

    }

    if(Input.GetKeyDown(KeyCode.Alpha4))
    {

        // Bajamos la intensidad de la luz
        this.light.intensity-=0.1;

    }

    // Modificamos el rango de la luz
    if(Input.GetKeyDown(KeyCode.Alpha5))
    {

        // Incrementamos el rango de la luz
        this.light.range+=1.0;

    }

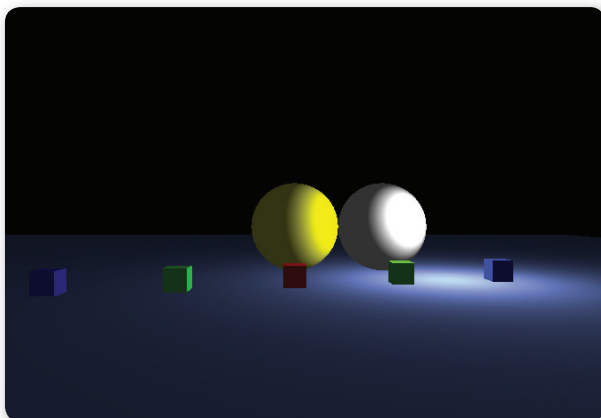
    if(Input.GetKeyDown(KeyCode.Alpha6))
    {

        // Disminuimos el rango de la luz
        this.light.range-=1.0;

    }

}
```

Cuando se oprime la tecla **5** del teclado alfanumérico, incrementamos el rango en **1**. En el caso de oprimir la tecla **6**, el rango disminuye. La propiedad pertenece al componente **light** de la luz de tipo **Point Light**.



► **Figura 7.**
Al incrementar el rango, más objetos se ven iluminados y los más cercanos reciben más luz.

La posición de la luz también se puede controlar por medio de código. Como la luz de tipo **Point Light** tiene un componente **Transform**, podemos moverla de la misma manera en que hemos aprendido a hacer con los objetos. El código del script queda de la siguiente forma:

```
function Start () {  
  
}  
  
function Update () {
```



DIBUJO DE HALOS



En ocasiones, colocar un halo alrededor de la fuente de luz incrementa el realismo de la escena y nos ayuda a ajustar la luz con más detalle. Ya que las luces no son un objeto físico, el halo indica su presencia. Para mostrarlo, debemos dirigirnos al panel **Inspector** con la luz seleccionada. En el componente **Light** encontramos una propiedad de tipo **bool** llamada **Draw Halo**. Si la marcamos, el halo se activa.

// Modificamos el color de la luz

```
if(Input.GetKeyDown(KeyCode.Alpha1))  
{
```

// Colocamos un valor al azar a cada componente de color

```
var r: float = Random.value;  
var g: float = Random.value;  
var b: float = Random.value;
```

// Creamos una variable de tipo Color y le asignamos los valores

```
var micolor: Color = new Color(r,g,b);
```

// Asignamos el color a la luz

```
this.light.color=micolor;
```

```
}
```

// Modificamos la intensidad de la luz

```
if(Input.GetKeyDown(KeyCode.Alpha3))  
{
```

// Incrementamos la intensidad de la luz

```
this.light.intensity+=0.1;
```

```
}
```

```
if(Input.GetKeyDown(KeyCode.Alpha4))
```

```
{
```

// Disminuimos la intensidad de la luz

```
this.light.intensity-=0.1;
```

```
}
```

// Modificamos el rango de la luz

```
if(Input.GetKeyDown(KeyCode.Alpha5))  
{
```

// Incrementamos el rango de la luz

```
this.light.range+=1.0;
```

```
}
```

```
if(Input.GetKeyDown(KeyCode.Alpha6))
```

```
{
```

// Disminuimos el rango de la luz

```
        this.light.range-=1.0;
    }

    // Movemos la luz
    if(Input.GetKey(KeyCode.RightArrow))
    {
        // Trasladamos la luz
        transform.Translate(0,0,0.1);
    }

    if(Input.GetKey(KeyCode.LeftArrow))
    {
        // Trasladamos la luz
        transform.Translate(0,0,-0.1);
    }
}
```

Directional Light

Otro tipo de luz que podemos utilizar es la **Directional Light** (luz direccional). Aunque comparte algunas características con los demás tipos, tiene ciertas cualidades propias. Podemos entender fácilmente su comportamiento si nos imaginamos un plano infinito a gran distancia que emite rayos de luz, todos paralelos entre sí, es decir que todos llevan la misma dirección. Podemos pensar en la **Directional Light** como una luz similar a la del Sol. Esta característica permite iluminar los objetos de manera uniforme.

Para ver su efecto, eliminamos la anterior y adicionamos una **Directional Light**. En este momento no importa la posición que tenga,

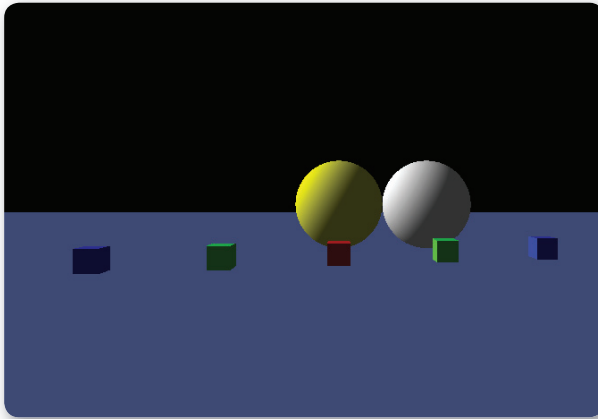


LOS RAYOS DE LUZ



Cuando hablamos de rayos de luz en las explicaciones, estamos usando una abstracción o analogía para entenderlo con más facilidad. En realidad, no hay rayos de luz, sino modelos matemáticos que definen el comportamiento de los diferentes tipos de luz. Sin embargo, es más sencillo explicarlo en términos de rayos de luz que en términos de ecuaciones matemáticas. En Internet encontraremos mucha información sobre los diferentes modelos de iluminación.

ya que la iluminación será la misma en cualquier lugar. Si debemos experimentar con la dirección, por lo que vamos a rotarla. Al hacerlo, veremos los efectos que tiene sobre la escena. Si observamos las esferas, es fácil comprobar que los rayos de luz son paralelos.



► **Figura 8.**
En las esferas es fácil contemplar el comportamiento de la luz **Directional Light**.

Vamos a experimentar modificando la posición de la luz. Vemos que la iluminación de la escena no cambia, lo que indica que, para este tipo de luz, lo que importa es la orientación y no, su posición. Empecemos a controlar la luz por medio de un script. Lo llamamos **LuzDireccional** y su código es el siguiente:

```
#pragma strict

// Este script modifica las propiedades principales de la luz de tipo
// Directional Light

function Start () {

}

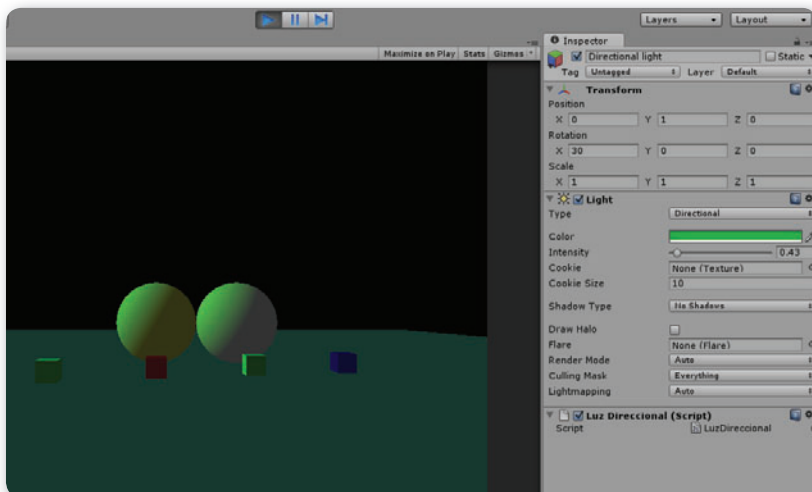
function Update () {
```



```
// Modificamos el color de la luz
if(Input.GetKeyDown(KeyCode.Alpha1))
{
    // Colocamos valores al azar para cada componente de color
    var r: float = Random.value;
    var g: float = Random.value;
    var b: float = Random.value;

    // Creamos una variable de tipo Color y le asignamos los valores
    var micolor: Color = new Color(r,g,b);

    // Asignamos el color a la luz
    this.light.color=micolor;
}
}
```



► **Figura 9.** El color de la luz **Directional Light** puede modificarse desde el editor o por medio de un script.

El color es una propiedad que se comparte con las otras luces. Podemos colocar el color directamente en el editor de Unity haciendo uso del panel **Inspector**. El color también puede cambiarse programáticamente, y eso es lo que hacemos en el script. Se selecciona un color al azar y se lo asigna a la luz; esto sucede cuando se oprime la tecla **1** del teclado alfanumérico.

Ahora agregamos más código al script:

```
#pragma strict

// Este script modifica las propiedades principales de la luz de tipo
// Directional Light

function Start () {

}

function Update () {

    // Modificamos el color de la luz
    if(Input.GetKeyDown(KeyCode.Alpha1))
    {

        // Colocamos valores al azar para cada componente de color
        var r: float = Random.value;
        var g: float = Random.value;
        var b: float = Random.value;

        // Creamos una variable de tipo Color y le asignamos los valores
        var micolor: Color = new Color(r,g,b);
```



DIRECTIONAL LIGHT Y LA PROPIEDAD RANGE



En este tipo de luz no tenemos la propiedad **Range** para manejar el rango, pues la luz se distribuye de manera uniforme en toda la escena, sin importar cuán grande sea. Este es un punto importante, pues no podemos usarla de forma local. Una luz de tipo **Directional Light** siempre va a iluminar todos los objetos de la escena, y puede darnos resultados no deseados o inesperados en otros lugares del escenario.

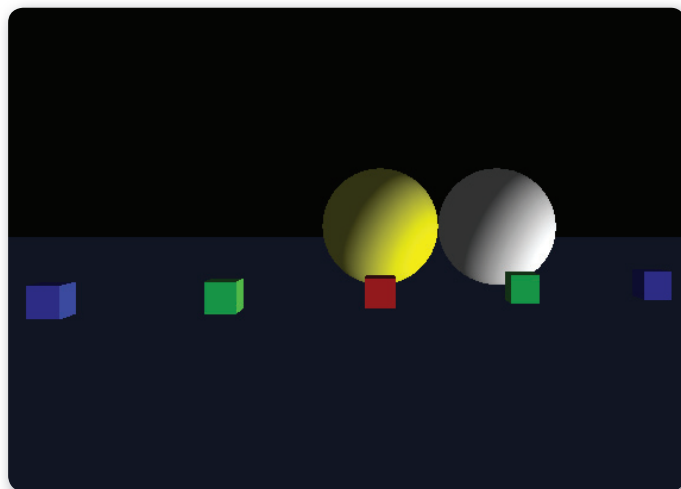
```
// Asignamos el color a la luz
this.light.color=micolor;

}

// Cambiamos la direccion de la luz
if(Input.GetKey(KeyCode.Alpha2))
{
    // Rotamos en el eje Y
    transform.Rotate(0,1,0);
}

}
```

Al asignar el script a la luz y oprimir la tecla **2** del teclado alfanumérico, observaremos el efecto que tiene la rotación sobre la iluminación de la escena. En este caso, únicamente estamos rotando en **Y**, pero es posible rotar en cualquier eje e, incluso, en varios a la vez.



► **Figura 10.** La rotación de la luz afecta la forma en que se ilumina la escena.

Esta luz no tiene la propiedad **Range**, pero sí, **Intensity** (intensidad), que nos permite cambiar su potencia. Con esto podemos ajustar más precisamente el tipo de luz que tenemos. Los valores que podemos asignar a la propiedad **Intensity** se encuentran en el rango de **0** a **8**. Al igual que en el caso de las otras propiedades, podemos asignarla directamente en el editor o controlarla de manera programática. Agregamos el siguiente código al script:

```
#pragma strict

// Este script modifica las propiedades principales de la luz de tipo
// Directional Light

function Start () {

}

function Update () {

    // Modificamos el color de la luz
    if(Input.GetKeyDown(KeyCode.Alpha1))
    {

        // Colocamos valores al azar para cada componente de color
        var r: float = Random.value;
        var g: float = Random.value;
        var b: float = Random.value;

        // Creamos una variable de tipo Color y le asignamos los valores
        var micolor: Color = new Color(r,g,b);

        // Asignamos el color a la luz
        this.light.color=micolor;

    }

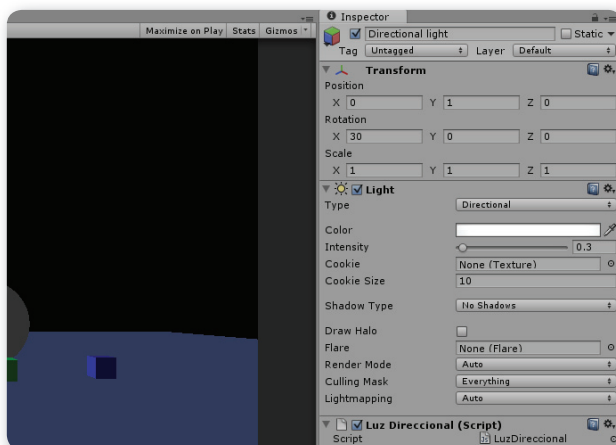
    // Cambiamos la direccion de la luz
    if(Input.GetKey(KeyCode.Alpha2))
    {
```

```
// Rotamos en el eje Y
transform.Rotate(0,1,0);
}

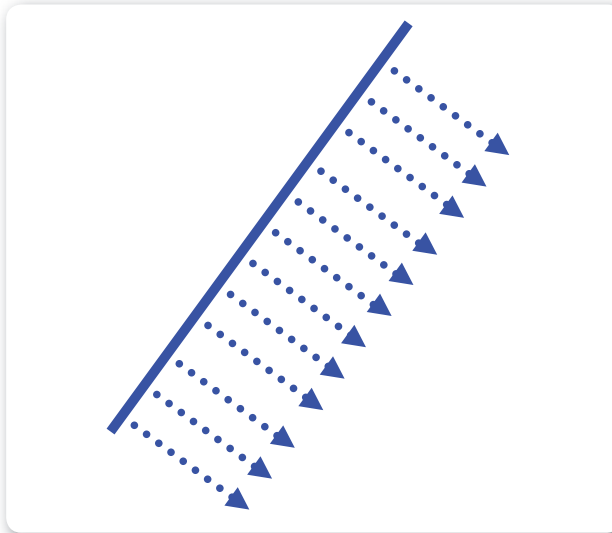
// Modificamos la intensidad de la luz
if(Input.GetKeyDown(KeyCode.Alpha3))
{
    // Incrementamos la intensidad
    this.light.intensity+=0.1;
}

if(Input.GetKeyDown(KeyCode.Alpha4))
{
    // Disminuimos la intensidad
    this.light.intensity-=0.1;
}
}
```

El código es similar al que ya hemos utilizado; solo usamos las teclas **3** y **4** para incrementar y disminuir la intensidad, respectivamente.



► **Figura 11.** Al no tener rango, la intensidad es una herramienta útil para controlar la iluminación.



► **Figura 12.** La luz **Directional Light** emite todos los rayos paralelos y en la misma dirección.

Spotlight

El tercer tipo de luz que vamos a utilizar es **Spotlight** (luz de tipo spot), similar a la que se emplea en los comercios y museos. Es una luz que podemos colocar en una posición y que ilumina hacia una dirección en particular. Tiene la característica de iluminar en forma de cono; es decir que tenemos un cono de luz. Lo que se encuentre dentro del volumen de ese cono se iluminará, y lo esté por fuera no recibirá luz.



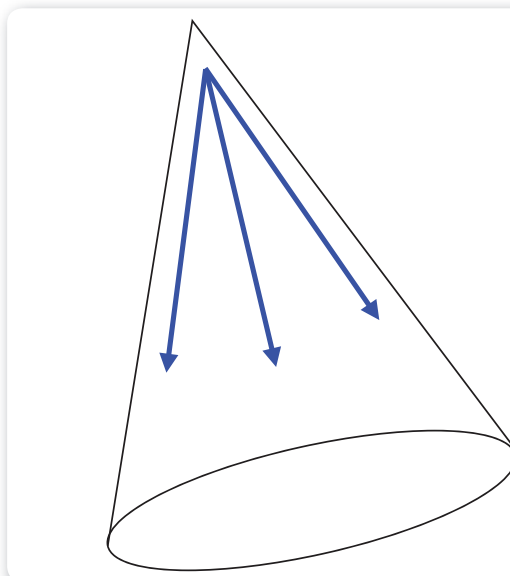
SELECCIONAR UNA LUZ DE TIPO AMBIENT LIGHT



La luz de ambiente es de gran ayuda para colocar las tonalidades de las sombras y permitir que se vean los objetos aun en condiciones donde no son iluminados por otras luces. El color de la luz de ambiente ayuda mucho a sentar el humor de la escena. Los colores cálidos o fríos dan diferentes sensaciones. El gris oscuro es un buen color neutro. Si tenemos muchas luces de un color en particular, podemos usar un tono oscuro de ese color.

Son importantes tanto la posición como la orientación, ya que un cambio en cualquiera de estas propiedades modificará la forma en que se ilumina la escena. Este tipo de luz también tiene otras propiedades que podemos utilizar para controlarla.

Para trabajar con ella, empecemos por eliminar las luces anteriores. A continuación, adicionamos una luz de tipo **Spotlight** y les colocamos valores a sus propiedades; luego las veremos con más detalle.



► **Figura 13.** La luz de tipo **Spotlight** tiene un cono que ilumina los objetos presentes en su interior.

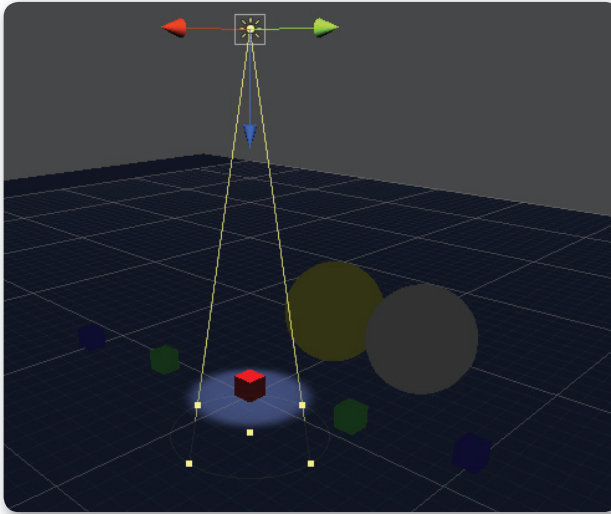


LAS SOMBRAS



Unity tiene un sistema de iluminación avanzado en su versión profesional que nos permite integrar diferentes clases de sombras. En la versión gratuita, las luces nos dan sombreados sobre el objeto iluminado, pero estos no proyectan ninguna sombra sobre los otros objetos. Si necesitamos tener sombras en nuestra aplicación, es necesario actualizar a la versión profesional, comprando la licencia correspondiente. Los costos se encuentran en el sitio web de Unity.

Colocamos la luz en la posición **(0,15,0)** para darle una ubicación cercana a los objetos. La rotación que le asignamos es **(90,0,0)**. En esta ocasión, el valor para **Range** es **17** y la propiedad que se llama **Spot Angle** debe tener un valor de **25**. A la intensidad le damos el valor máximo de **8**.



► **Figura 14.** Observamos la diferencia en la iluminación que produce este tipo de luz.

Empecemos a manejar las propiedades principales de esta luz. No explicaremos **Color**, ya que se maneja de manera similar a los otros tipos de luz. Creamos un nuevo script que asignaremos a **Spotlight** y que lleva por nombre **LuzSpot**:

```
#pragma strict
```

```
// Este script modifica las propiedades principales de la luz puntual
```

```
function Start () {
```

```
}
```

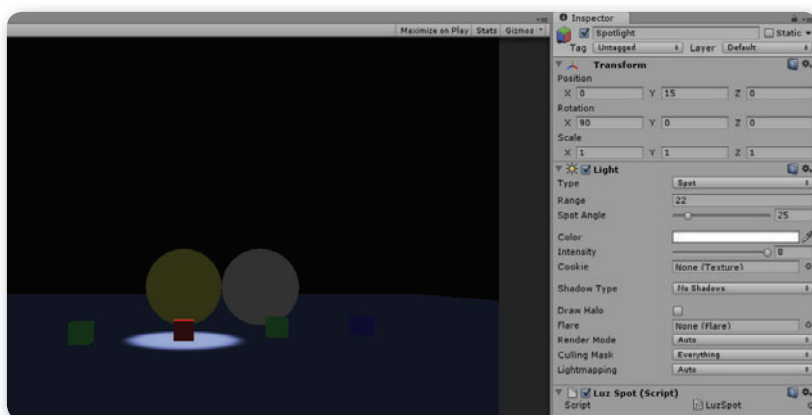
```
function Update () {
```



```
// Modificamos el rango de la luz
if(Input.GetKeyDown(KeyCode.Alpha1))
{
    // Incrementamos el rango
    this.light.range+=1.0;
}

if(Input.GetKeyDown(KeyCode.Alpha2))
{
    // Disminuimos el rango
    this.light.range-=1.0;
}
}
```

La primera propiedad con la que trabajamos es **Range**. Cuando se oprime la tecla **1**, se incrementa en **1**; y cuando se oprime la tecla **2** del teclado alfanumérico, se reduce. En este tipo de luz, **Range** equivale a la altura del cono de luz de **Spotlight**. Si la luz tiene un buen ángulo pero no ilumina correctamente al objeto, es posible que el rango sea un poco corto. Al ejecutar la aplicación, podemos observar que, manteniendo las demás propiedades constantes y modificando el rango, cambia la forma en que la escena se ilumina.

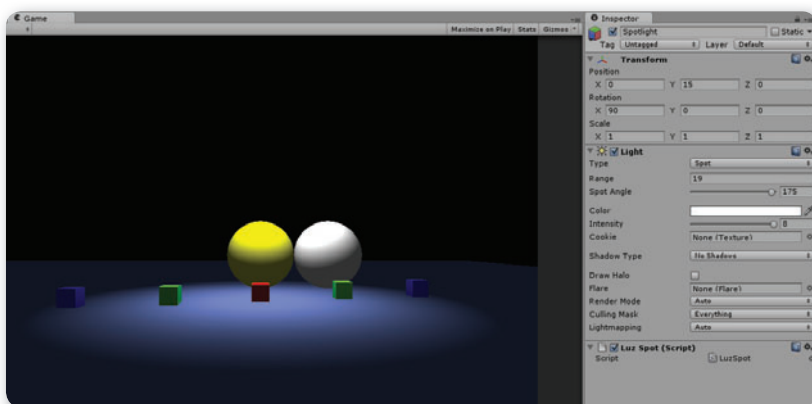


► **Figura 15.** Con la propiedad **Range** controlamos la altura del cono de luz para **Spotlight**.

La siguiente propiedad con la que vamos a trabajar se llama **Spot Angle**, y controla el ángulo interno del cono de luz. Esto significa que, por medio de la propiedad **Spot Angle**, controlamos cuán ancho o delgado será el cono de luz. El valor está dado en grados, y puede ir desde **1** hasta **179**. Este se asigna simplemente a la propiedad. Al adicionar el siguiente código, podemos controlar el ángulo mientras el programa se ejecuta:

```
function Start () {  
  
}  
  
function Update () {  
  
    // Modificamos el rango de la luz  
    if(Input.GetKeyDown(KeyCode.Alpha1))  
    {  
        // Incrementamos el rango  
        this.light.range+=1.0;  
    }  
  
    if(Input.GetKeyDown(KeyCode.Alpha2))  
    {  
        // Disminuimos el rango  
        this.light.range-=1.0;  
    }  
  
    // Modificamos el angulo del cono  
    if(Input.GetKey(KeyCode.Alpha3))  
    {  
        // Incrementamos el angulo  
        this.light.spotAngle++;  
    }  
  
    if(Input.GetKey(KeyCode.Alpha4))  
    {  
        // Disminuimos el angulo  
        this.light.spotAngle--;  
    }  
  
}
```

La propiedad **spotAngle** pertenece a la clase **Light** y, por eso, la encontramos dentro del componente **light** de la luz de tipo **Spotlight**. En este script la estamos incrementando o disminuyendo en **1** según la tecla que se oprima. Es posible asignar valores directamente o calcularlos mediante algún tipo de lógica. Lo único que debemos cuidar es el rango que vamos a usar.



► **Figura 16.** El ancho del cono puede ser controlado por medio de la propiedad **Spot Angle**.

Al cambiar la rotación de la luz, podemos controlar hacia dónde se dirige y elegir qué deseamos que ilumine. Es posible rotarla en cualquier eje y en la cantidad que necesitemos. Al agregar el siguiente código, controlamos estos factores:

```
#pragma strict

// Este script modifica las propiedades principales de la luz

function Start () {

}

function Update () {
```

```
// Modificamos el rango de la luz
if(Input.GetKeyDown(KeyCode.Alpha1))
{
    // Incrementamos el rango
    this.light.range+=1.0;
}

if(Input.GetKeyDown(KeyCode.Alpha2))
{
    // Reducimos el rango
    this.light.range-=1.0;
}

// Modificamos el angulo del cono
if(Input.GetKey(KeyCode.Alpha3))
{
    // Incrementamos el angulo
    this.light.spotAngle++;
}

if(Input.GetKey(KeyCode.Alpha4))
{
    // Reducimos el angulo
    this.light.spotAngle--;
}

// Rotamos la luz
if(Input.GetKey(KeyCode.LeftArrow))
{
    // Rotamos en el eje X
    transform.Rotate(0.1,0,0);
}

if(Input.GetKey(KeyCode.RightArrow))
{
    // Rotamos en el eje X
    transform.Rotate(-0.1,0,0);
}

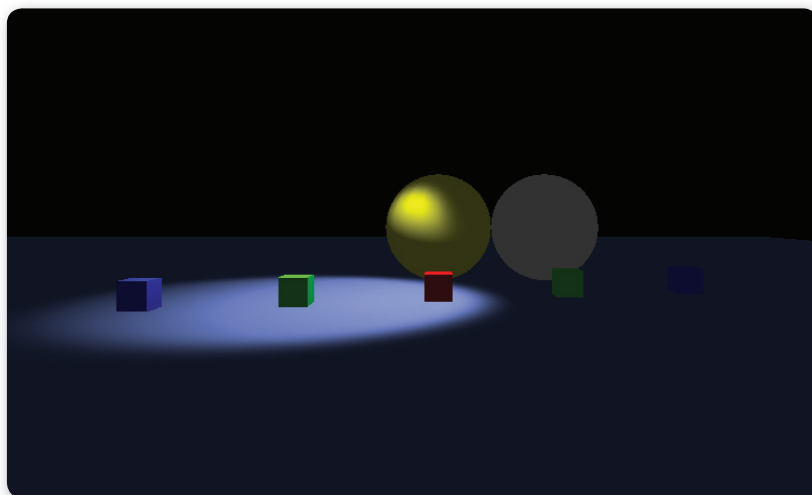
if(Input.GetKey(KeyCode.UpArrow))
{
```



```
// Rotamos en el eje Y
transform.Rotate(0,-0.1,0);
}

if(Input.GetKey(KeyCode.DownArrow))
{
    // Rotamos en el eje Y
    transform.Rotate(0,0.1,0);
}
}
```

La luz se rota por medio de las flechas del teclado, con las que podemos controlarla y dirigirla hacia diferentes posiciones.



► **Figura 17.** La luz de tipo **SpotLight** puede ser dirigida a un lugar en particular con solo rotarla.

Ambient Light

La última luz que vamos a conocer en este capítulo es la **Ambient Light** (luz de ambiente); es la que encontramos normalmente dispersa en toda la escena. En el mundo real, la luz puede entrar por una ventana, y todos los objetos reflejan un poco de ella. El piso refleja luz y es lo que nos permite observar la parte inferior de la mesa sin que exista una luz

abajo que la ilumine. La luz de ambiente es el total de la iluminación que recibimos por todos los reflejos de luz de todos los objetos. Hacer esto en la computadora para aplicaciones de tiempo real como los videojuegos sería muy intensivo en términos de cómputo. Por esta razón, hay que seleccionar previamente el color de **Ambient Light**.

Para colocar esta luz, seleccionamos el menú **Edit** seguido de la opción **Render Settings**. En el panel **Inspector** encontraremos una propiedad que se llama **Ambient Light**, donde elegimos el color de la luz.

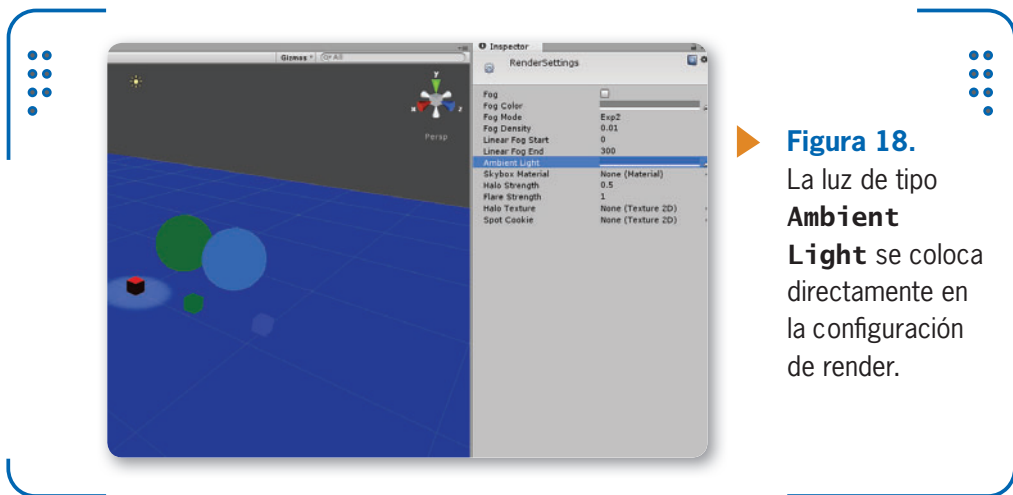


Figura 18. La luz de tipo **Ambient Light** se coloca directamente en la configuración de render.

A lo largo de este capítulo hemos visto los elementos más importantes de las luces que podemos utilizar, ya sea directamente en el editor o por medio de scripts. Este aspecto requiere de experimentación y práctica para dominar el manejo de los distintos tipos de luz y crear el ambiente que deseamos en la escena.



RESUMEN

Unity tiene diferentes tipos de luz que podemos utilizar en nuestras escenas. La luz de tipo **Directional Light** emite todos los rayos paralelos, y su orientación es importante para una correcta iluminación en la escena. En la **Point Light** los rayos de luz salen en todas las direcciones; para esta luz, el factor más importante es la posición. La luz **Spotlight** maneja tanto la posición como la orientación, y también son importantes las dimensiones del cono de luz. Por último, la luz de tipo **Ambient Light** ilumina la escena de forma general.

Actividades

TEST DE AUTOEVALUACIÓN

- 1 ¿Cuáles son los tipos básicos de luz en Unity?
- 2 ¿Cuál es la característica principal de la luz de tipo **Point Light**?
- 3 ¿Cuál es la característica principal de la luz **Directional Light**?
- 4 ¿Cuál es la característica principal de **Spotlight**?
- 5 ¿Qué es la luz de tipo **Ambient Light**?
- 6 ¿Cómo se coloca el color de una luz?
- 7 ¿Cómo funciona la propiedad **Range** de la luz?
- 8 ¿Cómo funciona la propiedad **Intensity** de la luz?
- 9 ¿Qué propiedad se usa para controlar el ángulo del cono de luz?
- 10 ¿Con qué propiedad podemos controlar la altura del cono de luz?

ACTIVIDADES PRÁCTICAS

- 1 Haga un script que cambie los rayos de luz de manera similar a como cambian durante el día.
- 2 Haga un script que cambie el color de la luz de forma similar a como sucede en el mundo real, del amanecer al anochecer.
- 3 Elabore un script que simule las luces giratorias de las patrullas.
- 4 Realice un script que encienda y apague una luz al oprimir una tecla
- 5 Haga un script para una luz de tipo **Spotlight** que siga constantemente a un objeto.

Física y cuerpos rígidos

Una parte importante de muchos videojuegos modernos es el uso de simulación física. La simulación nos permite aplicar las leyes de la física a nuestro mundo virtual, incrementa el realismo del juego y nos ayuda a interactuar de una manera natural con los objetos que encontramos en ese mundo.

▼ El sistema de física 184

Motores de física 184

Entender los colliders 185

Entender el rigid body 188

▼ Crear una aplicación

de prueba 190

Usar los triggers 191

Usar las colisiones 193

▼ Resumen..... 203

▼ Actividades..... 204



El sistema de física

Crear un sistema de física para un motor de juegos no es una tarea sencilla. Es necesario tener conocimientos sobre física clásica y matemáticas, y también precisamos tener un manejo avanzado de programación, pues debemos programar todo de una manera muy eficiente para que la aplicación no sufra en cuanto a su desempeño.

Motores de física

Existen en el mercado motores de física de diferentes costos, desde los de código abierto hasta los que cuestan miles de dólares.

En un principio, los motores de física se ejecutaban solo en la **CPU**, y esta es una de las razones por las cuales tienen que ser muy eficientes.

Con el tiempo, empezó a surgir la aceleración por hardware en la parte de física. El hecho de tener un **API** (*Application Programming Interface*, o interfaz de programación de aplicaciones) y el hardware necesario para ejecutarlo ayuda mucho, ya que libera a la CPU de una gran cantidad de procesamiento y mejora notablemente el desempeño del motor.

Una API importante es **PhysX**, desarrollada inicialmente por **Ageia** y ahora propiedad de **NVIDIA**. Su gran ventaja es que se ejecuta con aceleración por hardware. Es una API muy bien diseñada, eficiente y que se puede adaptar sin mucha dificultad a nuestros propios desarrollos. Aun con estas ventajas, es necesario tener buenos conocimientos de programación y entender con claridad la física clásica.

Unity utiliza PhysX como motor de física, y lo presenta con una interfaz muy simple, de manera que podamos utilizar todo lo relacionado con un motor de física avanzado sin tener que trabajar en exceso con la programación. Podemos hacer uso de la física de una manera similar a



EL SITIO WEB DE PHYSX



Para conocer más sobre PhysX (www.geforce.com/hardware/technology/physx), debemos visitar su sitio web. Allí encontraremos una descripción completa con ejemplos, videos y más información sobre la tecnología que se utiliza. También podemos ver una lista que ejemplifica el tipo de juegos profesionales que están usando PhysX.

las opciones que hemos estado usando hasta el momento. Tenemos componentes, propiedades y handlers que nos permiten utilizar todo esto sin problemas.

Entender los colliders

Una parte muy importante de la simulación física es la colisión. Hacer una colisión precisa sobre una superficie compleja como la de un personaje de videojuegos es complicado y requiere de muchos cálculos. Por eso, en vez de utilizar la geometría real del objeto, hacemos uso de diferentes geometrías más sencillas que nos dan efectos similares sin el costo en desempeño que esto significa.

Es debido a la forma que tienen estas geometrías, que la verificación de la colisión es muy sencilla y rápida. De manera básica, Unity tiene cuatro tipos de **colliders**: **box collider** (de caja), **sphere collider** (de esfera), **capsule collider** (de cápsula) y **mesh collider**. El collider es un objeto que lleva a cabo la lógica de la colisión. Posee una geometría que no se ve, pero que se usa en los cálculos de colisión; lo que hace es envolver a la geometría del objeto con la geometría del collider que más se le asemeje o convenga. El **mesh collider** sirve para detectar colisiones, pero es más complejo, ya que en lugar de usar una geometría sencilla hace uso de una copia de la geometría del objeto y se verifica la colisión contra cada uno de los triángulos que la componen. Si bien brinda la posibilidad de obtener una colisión más exacta, la desventaja que tiene es que requiere de mayor procesamiento.

UNA PARTE MUY
IMPORTANTE DE
LA SIMULACIÓN
FÍSICA ES LA
COLISIÓN



EL COLLIDER DE TIPO WHEEL



El collider de tipo **Wheel** es el más avanzado y es de un tipo especial, que se usa generalmente para los vehículos y representa una rueda. Además de la detección de colisiones, incorpora la física necesaria para las ruedas y los modelos de fricción que se pueden usar con los vehículos. Si nuestra aplicación requiere simulación física para los vehículos, podemos recurrir a este collider. Hay que tomar en cuenta que es necesario configurar correctamente sus propiedades para que la simulación se comporte de manera realista.

Box Collider

De manera general, los objetos que creamos en Unity tienen el componente collider incluido. Podemos eliminarlo e, incluso, reemplazarlo por algún otro en caso de que sea necesario. El **Box Collider** es, simplemente, una caja que puede envolver a nuestra geometría. Podemos modificar sus dimensiones para ajustarlo lo más posible al tipo de geometría que tenemos. Los colliders no solo van a tener dimensiones, también poseen una propiedad llamada **Center**, que nos permite ajustar su posición con relación a la geometría. El origen en el espacio del objeto de la geometría es la coordenada **(0,0,0)** de la propiedad **Center**. Al cambiar los valores de esta propiedad, modificamos la posición del collider con relación a la geometría.

Sphere Collider

Este collider es una esfera que envuelve a la geometría. Su cálculo de colisión es extremadamente rápido, pero no es muy conveniente para objetos muy delgados, pues hay mucho espacio entre la superficie del collider y la del objeto, y esto da lugar a falsos positivos de colisión. El tamaño se determina al colocar el radio de la esfera haciendo uso de la propiedad **Radius**.

Capsule Collider

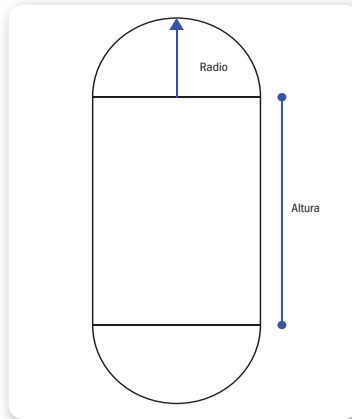
El **Capsule Collider** tiene la forma de una cápsula. Para delimitar su tamaño, cuenta con dos propiedades: **Radius** indica el radio de las secciones semiesféricas y el radio del cilindro; y **Height** determina la altura del cilindro. Este tipo de collider es muy usado en los personajes de videojuegos, y en controladores en primera y tercera persona.



COLLIDERS Y RIGID BODIES

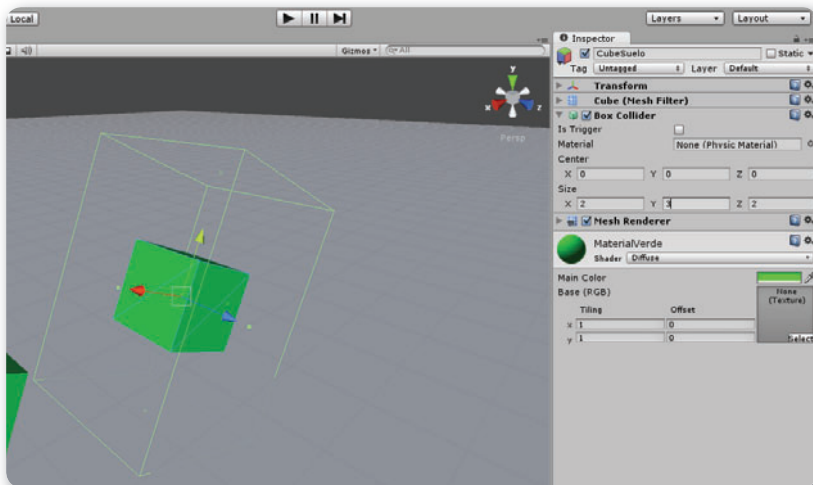


El collider sin más componentes se usa, principalmente, para objetos estáticos que tenemos en la escena. Si el objeto no se mueve y no planeamos que reaccione con física, es decir que solo se podrá colisionar con él, entonces se le pone el collider. Por el contrario, si el objeto va a tener movimiento y reaccionará con las leyes de la física, debemos adicionar también un **rigid body**.

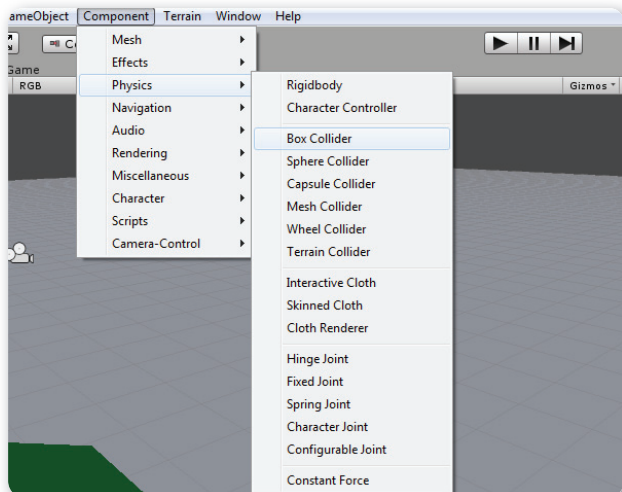


► **Figura 1.** Este diagrama muestra la forma en que luce el **Capsule Collider** y las propiedades que definen su tamaño.

Si necesitamos agregar algún tipo de collider, lo primero que debemos hacer es seleccionar el objeto al cual se lo vamos a colocar. Luego, nos dirigimos al menú **Component** y, de ahí, al submenú **Physics**. Al hacer clic, encontramos listados de los diferentes tipos de colliders; simplemente, pulsamos en el que precisamos. Es posible que, al adicionar el collider, Unity nos pregunte si queremos **Reemplazar con** o **Adicionar** uno nuevo.



► **Figura 2.** En esta figura encontramos un **Box collider** que envuelve al objeto; sus propiedades se encuentran en el panel **Inspector**.



► **Figura 3.** Adicionar colliders es muy fácil; simplemente, hay que hacer la selección del componente adecuado.

Entender el rigid body

Necesitamos otro componente que nos permita tener una reacción similar a la obtenida por objetos reales frente a estas colisiones. Este componente es **rigid body** y cuando lo adicionamos a un objeto, este usa el motor de física para reaccionar frente a la gravedad. Reacciona a las fuerzas ya sean lineales o de torca, y esto quiere decir que tiene simulación de física. El objeto no solo sirve para colisiones, también podemos adicionar fuerzas y otros comportamientos. Por ejemplo, la bala de un cañón recibe una fuerza determinada para salir disparada. El uso que le demos depende de nosotros.

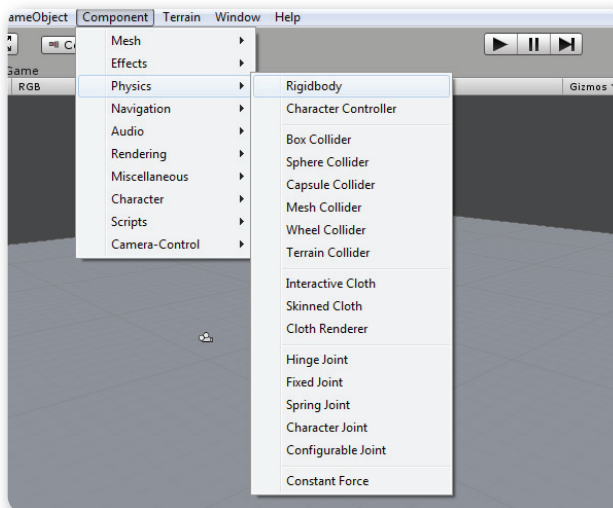


COLOCAR EL VALOR DE LA MASA



El correcto uso de la propiedad **Mass** requiere de práctica. Puede llegar a ser un proceso tedioso al principio, pero hacerlo correctamente ayuda mucho a mejorar la calidad del producto. No debemos olvidarnos de verificar la magnitud de las fuerzas. Algunas veces, el problema no son las masas, sino las fuerzas que actúan sobre los objetos. Es recomendable que la masa más grande no sea mayor a cien veces la más pequeña.

Solamente debemos adicionar el componente **Rigidbody** a los objetos que necesitan la parte de simulación física. Es muy simple hacerlo. Primero seleccionamos el objeto y luego nos dirigimos al menú **Component**. En los submenús que aparecen, seleccionamos **Physics**, ya que este componente pertenece a la clasificación de física. Luego hacemos clic en **Rigidbody**. En el panel **Inspector** veremos que aparece el componente. Ahora este objeto llevará a cabo simulación de acuerdo con las leyes de la física.



► **Figura 4.** Adicionar el componente **Rigidbody** es muy sencillo. Podemos encontrarlo en **Physics**, dentro del menú **Component**.

Veamos algunas de las propiedades más importantes que tiene el componente **Rigidbody**. Estas son significativas, ya que no todos los objetos deben tener el mismo comportamiento. Por ejemplo, cubos de diferentes materiales deben tener masas distintas. El juego no lucirá real si un automóvil tiene la misma masa que una manzana y reacciona de igual forma a una colisión que a un disparo.

La propiedad **Mass** es la primera que encontramos y representa la masa del objeto. En este caso, no tenemos unidades específicas, por lo que necesitaremos algo de experimentación.

LA GRAVEDAD
ES UN COMPONENTE
CRUCIAL DE LAS
SIMULACIONES
FÍSICAS



En el mundo real, cuando le aplicamos una fuerza a un objeto, este se mueve, pero después, empieza a perder velocidad y se detiene. Esto sucede debido a la fricción entre el objeto y la superficie de rodamiento o el aire. Para simular este comportamiento, debemos colocar un valor dentro de la propiedad **Drag**. Si ponemos **0**, el objeto no tendrá resistencia al aire o fricción y se moverá de manera indefinida. Con valores mayores, el objeto detiene rápidamente su movimiento. Hay una propiedad similar, pero relacionada con el movimiento angular (es decir, el de rotación), que se llama **Angular Drag**.

La gravedad es un componente importante de las simulaciones físicas y, por defecto, Unity la coloca activada en **Rigidbody**. Puede ocurrir que, por alguna razón, el objeto no necesite esto. La propiedad **Use Gravity** es de tipo **Boolean**, y cuando está seleccionada, la gravedad actúa sobre el objeto; en caso contrario, no lo afecta.



Crear una aplicación de prueba

Ahora que ya conocemos los conceptos básicos para usar física en Unity, podemos generar una aplicación de ejemplo que nos permita experimentar con este fenómeno. Creamos un nuevo proyecto con el nombre **Capítulo7** y guardamos la escena como **EjemploTrigger**.

Para empezar, colocamos un plano en la posición **(0,0,0)** y le damos una escala de **(10,1,10)**, lo que creará un piso sobre el cual ubicaremos diferentes objetos. Luego, ponemos una luz de tipo **Directional Light** y la giramos de modo tal que el piso se ilumine adecuadamente. Es necesario crear tres materiales, a los que llamaremos: **MaterialAmarillo**, **MaterialRojo** y **MaterialVerde**; les colocamos el color que indica su nombre.

A continuación, creamos un cubo que lleve por nombre **CubeSuelo**, ubicado en la posición **(0,1,0)**. Le damos una rotación de **(0,0,25)** y le asignamos el material **MaterialVerde**. Simplemente, verificamos que

el cubo tenga un collider de tipo **Box Collider**. Como este cubo no reaccionará a la física, no le colocamos un **Rigidbody**; el cubo nos sirve solo para la detección de colisiones. Creamos otro cubo de nombre **CubeSuelo2** y lo ponemos en la posición **(3,1,0)**, con la misma rotación que el anterior y el mismo material.

A continuación, crearemos otro cubo, que será nuestro primer objeto en hacer uso de un **Rigidbody**. Lo colocamos en la posición **(0,6.26,0)** y no le damos ninguna rotación. Lo llamamos **CuboAire** y le adicionamos un componente de tipo **Box Collider**. Dejamos las propiedades con sus valores predefinidos y le damos el color **MaterialRojo**. Ahora, ya podemos ver la simulación de física. Arrancamos la aplicación, y **CuboAire**, al tener el **Rigidbody**, empezará a caer por la gravedad y luego colisionará con uno de los cubos verdes.

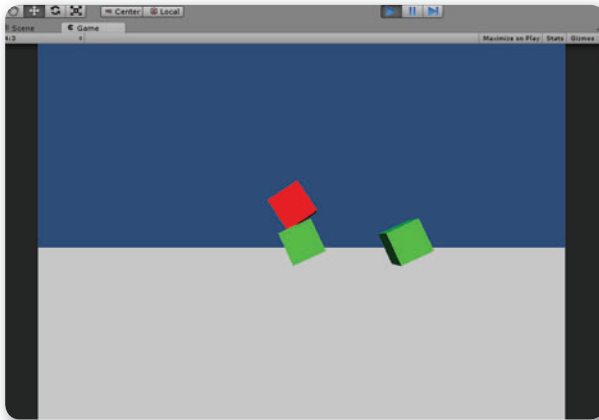


Figura 5.

Ahora el cubo rojo cae y, luego, colisionará con uno de los cubos verdes gracias al uso de **Rigidbody**.

Usar los triggers

Los **triggers** son eventos que ocurren cuando se detecta una colisión. Actúan fuera del motor de física y son útiles para detectar colisiones con objetos como puertas, que dan bonos, etc. Para indicar que un objeto usará triggers, debemos marcar la propiedad **Is Trigger** del componente **Box Collider**. Hacemos esto para el objeto **CubeSuelo2**.

Al tener un evento, podemos usar un handler, que es una función, y realizar una acción cuando esa colisión se lleve a cabo. Para esto, creamos un script al que llamamos **ColisionTrigger** con el siguiente código:


```
#pragma strict

function Start () {

}

function Update () {

}

// Handler para el inicio de la colision por medio del trigger
function OnTriggerEnter (objetoCol : Collider) {

    // Trasladamos el objeto en X
    transform.Translate(1,0,0);
}
```

El handler que usamos es **OnTriggerEnter()**, que tiene un parámetro de tipo **Collider** llamado **objetoCol**. Dentro de este objeto, tenemos la información sobre el objeto que colisionó con el que tiene el trigger activado. En el handler, podemos colocar cualquier código que necesitemos. En este caso, como solo estamos experimentando, lo que hacemos es trasladar el objeto con el trigger una unidad en el eje **X** local.

Agregamos otro cubo, que será el objeto que colisionará. Lo llamamos **CubeAire2**, lo ubicamos en la posición **(2.81,6.3,0)** y le colocamos **MaterialRojo**. Asignamos este script al objeto **CubeSuelo2** y ejecutamos la aplicación. Veremos que, al momento de colisionar, el cubo se mueve.

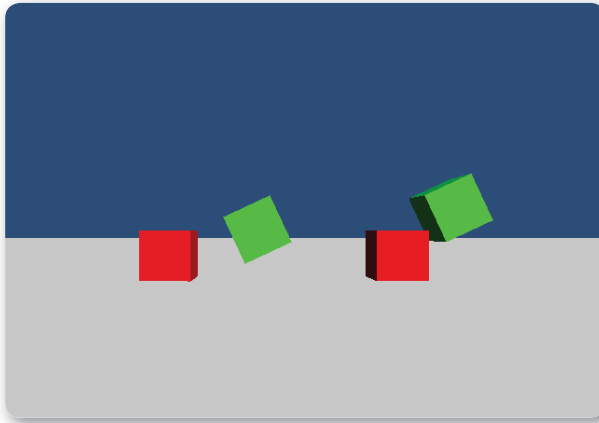


¿TE RESULTA ÚTIL?

Lo que estás leyendo es el fruto del **trabajo de cientos de personas** que ponen todo de sí para lograr un **mejor producto**. Utilizar versiones "**pirata**" desalienta la inversión y da lugar a publicaciones de **menor calidad**.

NO ATENTES CONTRA LA LECTURA. NO ATENTES CONTRA TI. COMPRA SÓLO PRODUCTOS ORIGINALES.

Nuestras publicaciones se comercializan en kioscos o puestos de voceadores; librerías; locales cerrados; supermercados e internet (usershop.redusers.com). Si tienes alguna duda, comentario o quieres saber más, puedes contactarnos por medio de usershop@redusers.com



► **Figura 6.**
El cubo ha reaccionado a la colisión por medio del código del trigger.

Aunque el trigger es muy sencillo de utilizar, tiene el inconveniente de que no usa el motor de física. Sin embargo, es muy útil cuando necesitamos detectar la colisión contra un objeto, sin que este tenga que reaccionar con simulación física.

Usar las colisiones

Con todo esto que vimos, ya podemos empezar a usar colisiones junto con el motor de física. Necesitamos modificar la escena y hacer algunos cambios. Empecemos por deshabilitar el script anterior y, luego, vamos a insertar un controlador en tercera persona, que ponemos en la posición **(0,1,-14)** y apuntando directamente hacia nuestros cubos.

Agregamos un nuevo cubo, llamado **CubeAire3**, al que le asignamos el **MaterialAmarillo**, y colocamos en la posición **(4.3,15,0)**. Este cubo deberá

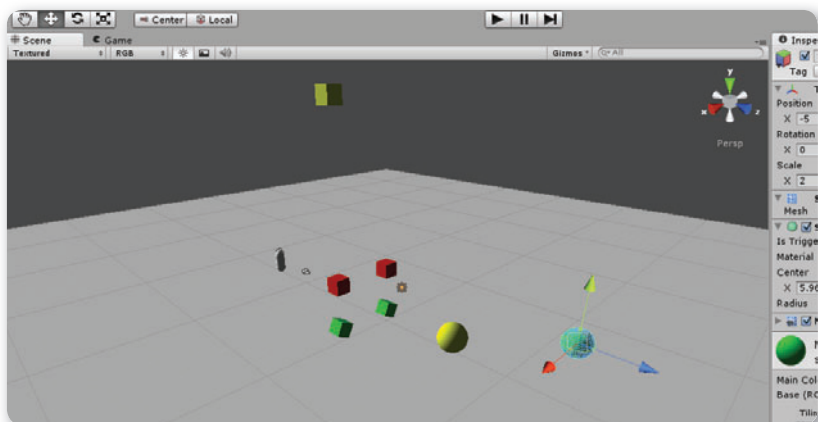


OTROS HANDLERS PARA EL USO DE TRIGGERS



Hemos visto cómo usar **OnTriggerEnter()**, que se invoca cuando se entra en la colisión, pero también tenemos otros handlers que pueden ser útiles. El handler **OnTriggerExit()** se invoca cuando el colisionador ya no toca el objeto, y es útil para llevar a cabo código cuando terminó la colisión. Por su parte, **OnTriggerStay()** se invoca mientras la colisión se siga dando. La elección del handler que debemos usar depende de la lógica de nuestra aplicación, aunque **OnTriggerEnter()** es buena opción si únicamente deseamos detectarla.

tener un **Rigidbody**, ya que deseamos que la gravedad actúe sobre él. Luego, también le colocamos una esfera con nombre **SphereAmarilla** en la posición **(0,1,6)**, con una escala de **(2,2,2)**, y le asignamos **MaterialAmarillo**. Añadimos otra esfera denominada **SphereVerde** y que llevará por material **MaterialVerde**. La escala es de **(2,2,2)** y su posición es **(-5,1,12)**.



► **Figura 7.** La escena ahora cuenta con todos los objetos necesarios.

Empecemos con una colisión sencilla, creando un script llamado **Colision**. Este será asignado al objeto que deseamos que lleve la lógica cuando sea colisionado, que en este caso es **CubePiso**. El código que le colocamos es el siguiente:

```
#pragma strict

function Start () {

}

function Update () {

}

// Handler para la colision
function OnCollisionEnter(objetoCol : Collision) {
```

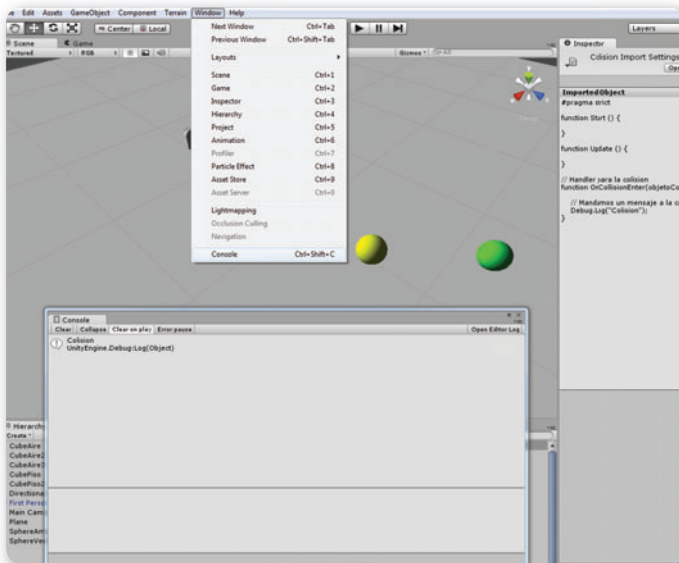
// Mandamos un mensaje a la consola

```
Debug.Log("Colision");
```

```
}
```

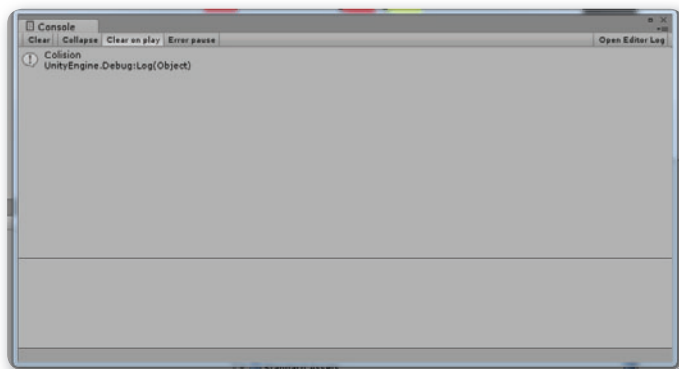
Cuando la colisión ocurre, se invoca el handler **OnCollisionEnter()**. Este tiene un parámetro de tipo **Collision** al que llamamos **objetoCol**, que contiene información sobre el objeto con el cual se produce la colisión. Un aspecto muy importante en el interior del código es el método **Log**, perteneciente a la clase **Debug**. Por medio de este método, podemos ayudarnos mucho en el proceso de depuración, porque nos permite imprimir mensajes en la consola. La consola es una ventana especial que Unity utiliza para comunicarse con nosotros. Los mensajes también aparecen en la barra de estado, en la parte inferior del editor de Unity.

Dentro del handler, podemos colocar cualquier código que sea necesario para la lógica de la aplicación. En este caso, lo que hacemos es mandar un mensaje. El método **Log()** requiere de un parámetro, que es una cadena con el mensaje que se va a mostrar. Es posible imprimir también los valores de variables y propiedades usando el método **ToString()** cuando corresponda.



► **Figura 8.** La consola y el método **Log()** son de gran utilidad al depurar la aplicación.

Ahora podemos ejecutar la aplicación, con la consola abierta. En el momento en que el cubo rojo colisiona con **CubePiso**, el mensaje de **Log()** aparece en la consola.



► **Figura 9.** El mensaje producido por **Log** en la colisión se muestra en la consola.

Es el momento de aprender algo que nos será de mucha utilidad y que consiste en identificar el objeto con el que se ha colisionado. Esto es necesario, pues un personaje no reacciona igual al colisionar contra una bala que al hacerlo contra un objeto que dé puntos extras. En este ejemplo, **CubePiso2** será colisionado por dos cubos, pero solo reaccionará ante la colisión de **CubeAire3**. Antes de colocar el siguiente código, hay que verificar que **CubePiso2** no tenga ningún script asignado. Luego, creamos un nuevo script que llamamos **ColisionNombres** y le colocamos el código de la página siguiente.



EL USO DE LA CONSOLA



La consola es una ventana donde se despliega información desde Unity. Podemos mostrarla desde el menú **Window** y luego seleccionar **Console**. Otra forma es a través de la combinación de teclas **CTRL + SHIFT + C**. En la parte superior de la consola tenemos botones que nos permiten usarla. Podemos modificar su tamaño para facilitar la parte de depuración. Los contenidos se mantienen aunque esté cerrada.

```
#pragma strict

function Start () {

}

function Update () {

}

// Handler para la colision
function OnCollisionEnter(objetoCol : Collision) {

    // Verificamos si el objeto con el que colisionamos es CubeAire3
    if(objetoCol.gameObject.name=="CubeAire3")
        // Trasladamos el objeto en -1 en Z
        transform.Translate(0,0,-1);

}
```

El script es asignado a **CubePiso2**. Sabemos que dentro de **objetoCol** se guarda la información del objeto con el cual se colisionó. Dentro de **objetoCol** tenemos un **gameObject** que representa el objeto con el que se colisiona y que tiene una propiedad llamada **name**. Haciendo uso de ella, podemos reconocer con qué objeto se ha colisionado.

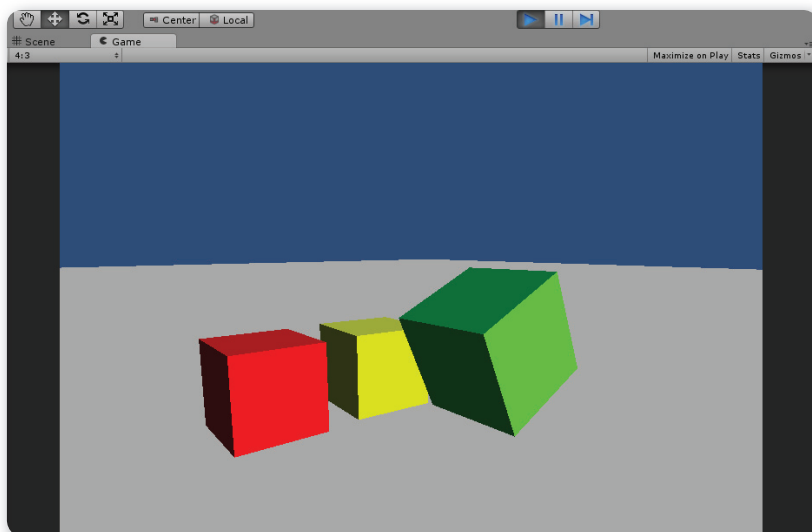
Dentro del handler de la colisión, utilizamos **if** para comparar el nombre del objeto con el que se colisiona y **CubeAire3**. Solamente cuando la colisión se produzca contra **CubeAire3**, el objeto se moverá en **Z**. De esta forma, podríamos usar varios **if** para que la reacción fuera diferente con distintos objetos.



EL USO DE TAGS



Puede resultar complicado hacer **if** para todos los nombres posibles; en ese caso, hacemos uso de **tags** (etiquetas). Podemos crear un tag y colocárselo a muchos objetos. De esta forma, en la colisión, en vez de comparar contra el nombre, hacemos la comparación contra el tag. Esto ayuda mucho a reducir el código y a hacer más flexible la aplicación.



► **Figura 10.** El cubo verde solamente reacciona a la colisión con el cubo amarillo, e ignora al rojo.

Estas técnicas son útiles para las colisiones entre objetos, pero, cuando queremos usar la detección de colisiones de los controladores en primera persona, tercera, etc., es necesario usar otra técnica diferente.

Colisiones con controlador

En el siguiente ejemplo, utilizamos el controlador en primera persona y llevamos a cabo la detección de colisiones. La esfera amarilla será destruida cuando el jugador colisione con ella. Necesitamos hacer un script que se agregue al controlador de primera persona; luego usaremos un handler especial y detectaremos si el objeto colisionado es la esfera que nos interesa. Si ese es el caso, entonces la destruiremos.

Para empezar, creamos un nuevo script y lo llamamos **ColisionControlador**. Colocamos el siguiente código:

```
#pragma strict
```

```
function Start () {
```

```
}

function Update () {

}

// Verificamos por colision con el controlador
function OnControllerColliderHit(objetoCol: ControllerColliderHit){

    // Verificamos si colisionamos con el objeto SphereAmarilla
    if(objetoCol.gameObject.name=="SphereAmarilla")
        // Destruimos el objeto
        Destroy(objetoCol.gameObject);

}
```

Usamos **OnControllerColliderHit()**, un handler especial que se usa para la detección de colisiones en todos aquellos objetos que tengan controladores, no solamente los controladores en primera y tercera persona que hemos visto. Para todos los demás objetos, lo recomendable es utilizar los handlers que hemos visto anteriormente. Este handler también tiene un objeto como parámetro, al cual llamamos **objetoCol**; es de tipo **ControllerColliderHit** y, de manera similar a los casos anteriores, guarda información sobre el objeto con el que se ha colisionado. En el interior del handler, podemos colocar cualquier código que sea necesario para llevar a cabo nuestra lógica.

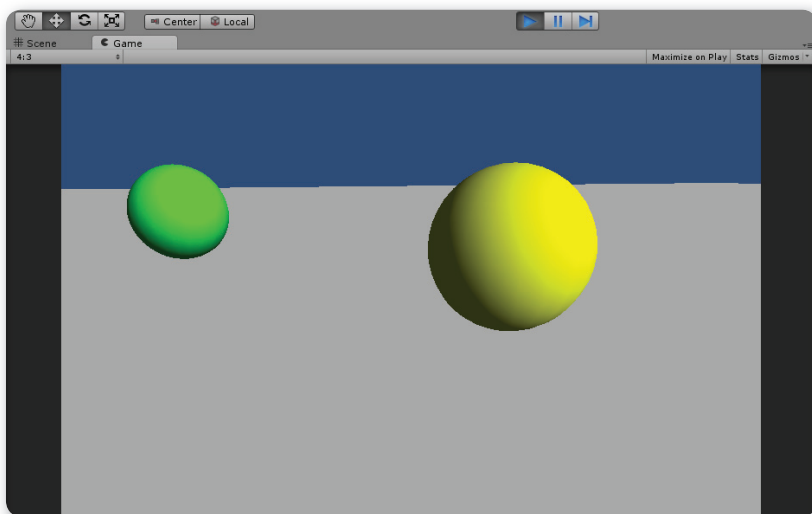
Empezamos por verificar si el objeto con el que hemos colisionado se llama **SphereAmarilla**. Solo cuando colisionemos con **SphereAmarilla**, se producirá la destrucción del objeto.

Destruir objetos es algo muy útil y común en los videojuegos, sobre todo, en el caso de objetos que ya no van a ser necesarios. Esta destrucción ayuda a mejorar el desempeño de la aplicación. Por ejemplo, los objetos que se van recolectando a lo largo del juego pueden ser destruidos una vez que el jugador los ha tomado. Otro ejemplo es el de los juegos de tipo **Shooter** (tirador), en que los personajes que han muerto son eliminados del nivel. Si creamos muchos objetos, es necesario que en algún momento los eliminemos. Si creamos, por ejemplo, las balas, en muy poco tiempo tendremos miles de objetos

extras que estarán consumiendo recursos, por lo que es recomendable ir eliminando aquellos objetos que ya no aporten funcionalidad al juego.

Para destruir un objetos vamos a usar el método **Destroy()**. Este necesita un parámetro, y es allí donde colocamos el objeto por destruir. Como queremos destruir el objeto con el que hemos colisionado, asignamos **objetoCol.gameObject** como parámetro.

Le colocamos el script al controlador en primera persona y ejecutamos la aplicación. Podemos movernos por la escena y colisionar con diversos objetos, pero, en el momento en que lo hagamos contra la esfera amarilla, esta será destruida.



► **Figura 11.** En el momento en que colisionemos con la esfera amarilla, esta será destruida y desaparecerá de la escena.



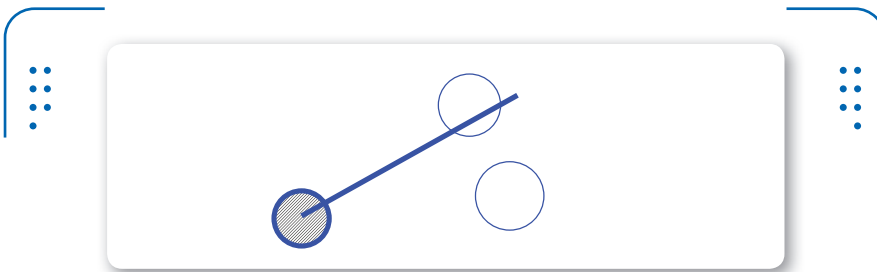
LA DESTRUCCIÓN DE LOS OBJETOS



Cuando usamos el método **Destroy()** para destruir objetos, esto sucede solo en tiempo de ejecución. Quiere decir que, si creamos un objeto por medio del editor de Unity y luego lo destruimos cuando se ejecuta el juego, desaparecerá de la escena, pero al momento de finalizar el juego, aparecerá otra vez en el editor. No hay que preocuparse por que **Destroy()** altere la escena en el editor.

Colisiones por medio de RayCast

Existe otro mecanismo de colisión que es por medio de lanzamiento de rayos. Estos son, en realidad, una abstracción o un modelo matemático. Podemos entenderlo como un segmento de línea que va en una dirección en particular. Si algún objeto se interpone en esta línea que llamamos rayo, decimos que ha ocurrido una colisión. Este mecanismo de colisión nos permite detectar objetos sin tener que hacer contacto físicamente con ellos por medio de un collider. Puede ser útil, por ejemplo, en el caso de un rifle. Para saber si el disparo acertó, lanzamos un rayo desde el rifle, en la dirección en que apunta, y como tamaño le damos la distancia máxima de disparo. Si un objeto se interpone, consideramos que la bala acertó. Cada vez que necesitemos esta detección a distancia, podemos hacer uso de los rayos.



► **Figura 12.** El objeto lanza un rayo en determinada dirección. Si algún objeto se interpone, consideramos que la colisión se efectuó.

Ahora procedemos a crear este efecto en Unity. Lo que haremos será disparar un rayo; cuando haya una colisión con la esfera verde, esta será trasladada, de tal forma que el jugador nunca llega a tocarla. El script es asignado al controlador en primera persona, y le ponemos por nombre **ColisiónRayCast**. El código que adicionaremos es el siguiente:

```
#pragma strict

function Start () {

}

function Update () {
```

```
// Creamos un objeto de tipo RaycastHit
var colision : RaycastHit;

// Disparamos el rayo y verificamos si hay colision
if(Physics.Raycast (transform.position, transform.forward, colision, 3))
{
    // Verificamos que el objeto con el que se colisiona es SphereVerde
    if(colision.collider.gameObject.name=="SphereVerde")
    {
        // Movemos el objeto con el que se colisiona
        colision.collider.transform.Translate(transform.forward);
    }
}
}
```

El código es sencillo y todo trabaja en la función **Update()**, pues nos interesa que se verifique la colisión en cada cuadro de animación; sin embargo, también podemos usarlo en funciones o casos particulares.

Lo primero que encontramos es la creación de una variable de tipo **RaycastHit**, que contendrá la información de la colisión; trabajaremos con ella para acceder al objeto con el que colisionamos. Luego tenemos un **if** para verificar si hubo o no colisión. Para saberlo, invocamos el método **Raycast()**, que regresa un valor **boolean**. Si hubo colisión, se devuelve el valor **true**; en caso contrario, obtenemos el valor **false**, y el **if** no se cumple.

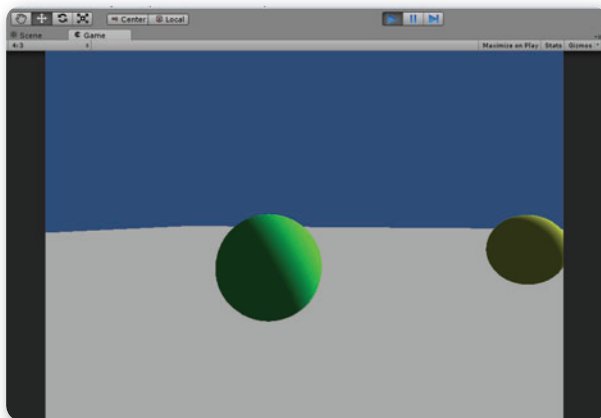
El método **Raycast()** pertenece a la clase **Physics** y necesita de cuatro parámetros. El primero es de tipo **Vector3** e indica la posición donde se inicia el rayo. Como deseamos que comience donde se encuentra el controlador en primera persona, lo colocamos en **transform.position**. El segundo parámetro es también de tipo **Vector3** y se usa para indicar la dirección en la que el rayo se dirige, y como queremos que vaya en la misma dirección en que el jugador está observando, usamos **transform.forward**. El tercer parámetro es el objeto que va a contener la información de colisión, motivo por el cual pasamos a **collision**. Por último, debemos indicar el tamaño que va a tener el rayo en unidades de mundo.

En nuestro código, se crea un rayo con tres unidades de tamaño. Si se colisionó, procedemos a verificar que dicha colisión se haya dado con

el objeto que tiene por nombre **SphereVerde**. Verificamos la propiedad **name**, que se encuentra en el **gameObject** del collider. En caso de colisión, trasladamos el objeto por medio del método **Translate** y pasamos como parámetro **transform.forward**, para que se mueva en la misma dirección que estamos observando. **Translate** forma parte de **transform**, que está contenido en el collider de **colision**.

Ahora podemos ejecutar la aplicación y, al acercarnos a la esfera verde, veremos cómo esta se aleja.

De este modo, hemos aprendido las bases sobre colisiones y los mecanismos más importantes de colisión. Solamente debemos adicionar lógica para hacer que cosas interesantes sucedan al colisionar.



► **Figura 13.**
La esfera se aleja al detectarse la colisión por medio del uso del rayo.



RESUMEN

Unity incorpora un sistema profesional para la física, conocido como PhysX. Para usarlo, debemos adicionar los componentes de física relacionados. La detección de colisiones se hace por medio de colliders, y los hay de varios tipos. Si deseamos que el objeto reaccione a la colisión, debemos usar el handler apropiado, dependiendo del mecanismo de detección que hemos seleccionado. Para los controladores tenemos que usar un handler especial. La parte relacionada con la simulación física pertenece al componente **Rigidbody**. Este tiene varias propiedades que podemos configurar para lograr un mayor realismo en el comportamiento del objeto. Otro mecanismo de detección es por medio del uso de rayos. En cualquier handler de colisiones, siempre tenemos como parámetro un objeto que nos da información sobre el objeto con el que se ha colisionado.

Actividades

TEST DE AUTOEVALUACIÓN

- 1 ¿Cuál es el API de física usado por Unity?
- 2 ¿Qué es un collider?
- 3 ¿Cuáles son los tipos básicos de colliders?
- 4 ¿Cuál es el uso de la propiedad **Center** de los colliders?
- 5 ¿Qué es **Rigidbody**?
- 6 ¿Para qué se usa la propiedad **Mass**?
- 7 ¿Qué es un trigger?
- 8 ¿Cómo se usa el handler **OnCollisionEnter()**?
- 9 ¿Para qué se usa el método **Log()**?
- 10 ¿Para qué se usa el método **Destroy()**?
- 11 ¿Cómo se activa la gravedad en la simulación física?
- 12 ¿Cómo funciona la detección de colisiones por rayos?

ACTIVIDADES PRÁCTICAS

- 1 En la aplicación de la pista, coloque objetos que, cuando sean recogidos, aumenten la velocidad del cubo.
- 2 Investigue sobre los materiales físicos y observe cómo modifican la simulación física.
- 3 En la aplicación de la pista, cree un objeto vacío con **Box Collider** y, por medio de un rayo, impida que el jugador pase por la parte interior de la pista para acortar camino.
- 4 Haga la aplicación de la pista para dos jugadores y, por medio de colisiones, detecte quién gana la carrera.
- 5 Haga un sube y baja (balancín) que trabaje de forma perpetua al resetear las posiciones de los objetos que caen después de colisionar.

Efectos especiales

Unity nos provee de buenas herramientas para lograr efectos especiales, con las que podemos incorporar explosiones, humo, lluvia y muchas cosas más a nuestros juegos. Como cualquier otra herramienta del programa, su uso es sencillo. En los videojuegos modernos, los efectos especiales se han vuelto cada vez más importantes, porque ayudan, no solo a dar realismo, sino también a mantener el interés del jugador. La creación de efectos especiales requiere un buen conocimiento de las bases y mucha experimentación. Únicamente la práctica nos permitirá obtener resultados cada vez más reales.

▼ El sistema de partículas206

Usos y características..... 206

▼ Crear un sistema

de partículas.....207

Editar los diferentes

tipos de datos.....212

Más propiedades importantes.....214

Crear un subsistema..... 219

El uso del editor..... 221

Efectos preconstruidos..... 222

▼ Resumen.....223

▼ Actividades.....224





El sistema de partículas

La base de muchos efectos especiales es el sistema de partículas. Unity siempre se ha caracterizado por tener un buen sistema, pero luego de la reciente actualización, trabaja de un modo mucho mejor. Estos cambios no fueron mera cuestión de desempeño: las propiedades han quedado mejor organizadas, y con nombres y valores más descriptivos, por lo son mucho más fáciles de utilizar.

Usos y características

Un sistema de partículas puede utilizarse para crear efectos como fuego, explosiones, humo, lluvia, fuentes, cascadas, y más. El sistema tiene uno o varios emisores, es decir, puntos en el espacio a partir de los cuales surgen las partículas. Además, los emisores no son solo puntos: son volúmenes de los que surgen las partículas. La partícula es un objeto muy simple. Podemos verlo como un cuadro que contiene una textura y una posición; es decir, su geometría es sumamente sencilla. Las partículas tienen una propiedad conocida como **billboarding**. Podemos

UN SISTEMA DE
PARTÍCULAS PUEDE
UTILIZARSE PARA
CREAR EFECTOS
DE TODO TIPO

pensar en cada una de ellas como en un elemento que compone el efecto. Por ejemplo, en el fuego, la partícula puede representar una llama pequeña, pero al haber cientos o miles de ellas, el resultado será un fuego intenso. En la lluvia, cada partícula puede ser una de las gotas.

La forma en que surgen estas partículas depende de las propiedades del sistema. Cada una tendrá un tiempo de vida durante el cual puede presentar diferentes características. Por ejemplo, la partícula puede modificar su color según pasa

su vida. Si seguimos con el ejemplo del fuego, puede tener un color amarillo al inicio, pasar luego por rojo, después por el naranja oscuro, y terminar en un tono de gris. Otros elementos, como la dirección en la que sale y su velocidad, pueden controlarse. Las partículas pueden ser modificadas por fenómenos físicos, como la gravedad o las colisiones. Esto depende mucho del tipo de efecto que vayamos a crear. Si deseamos tener una fuente, es necesario colocar la gravedad e, incluso, detectar colisiones para que la caída del agua parezca real.

Muchos elementos sencillos pueden realizarse con un emisor, pero los mejores efectos se logran al usar varios, cada uno de ellos, especializado en una etapa del efecto. En Unity, podemos colocar varios emisores al mismo tiempo y, de esta forma, crear efectos profesionales. El programa provee también de una serie de efectos creados previamente que podemos modificar según nuestra necesidad.

En Unity, el sistema de partículas está compuesto de **módulos**, cada uno de los cuales tiene diversas propiedades que controlan diferentes elementos del sistema. Los módulos pueden ser encendidos y apagados según sea necesario y dependiendo realmente de qué tipo de funcionalidades deseamos para el sistema. Esta forma de organización mejora con relación al sistema anterior. Algunas propiedades tienen editores propios, aspecto que permite ejercer mayor control sobre los valores por utilizar. Esto es fundamental, pues no todos los valores son estáticos a lo largo del tiempo, como ya hemos comentado. Entonces, estos editores nos permiten indicar el cambio del valor de la propiedad según pasa el ciclo de vida de la partícula.

Crear un sistema de partículas

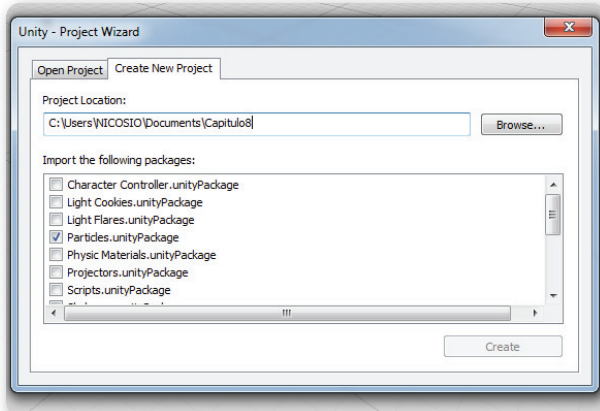
Para este capítulo, vamos a crear un proyecto al cual llamamos **Capítulo8**. Cuando lo estamos haciendo y el programa lista las opciones de paquetes para importar, seleccionamos la opción denominada **Particles.unityPackage**, que incluye muchos de los efectos ya creados que utilizaremos más adelante.



TORNADOS



Un punto importante en los efectos con partículas es la rotación, en especial, si deseamos crear un efecto similar a un tornado o cualquier otro movimiento circular. El módulo **Rotation over Lifetime** nos permite dar una velocidad angular a las partículas y, de esta forma, hacerlas rotar. Podemos usar un valor constante, curva o aleatorios.



► **Figura 1.**
Al crear el proyecto, seleccionamos el paquete de partículas, para importarlo y usar en nuestra aplicación.

Para colocar un sistema de partículas, seleccionamos el menú **GameObject** y la opción **Create Other**. Aparece una lista de alternativas de la que seleccionamos **Particle System**.

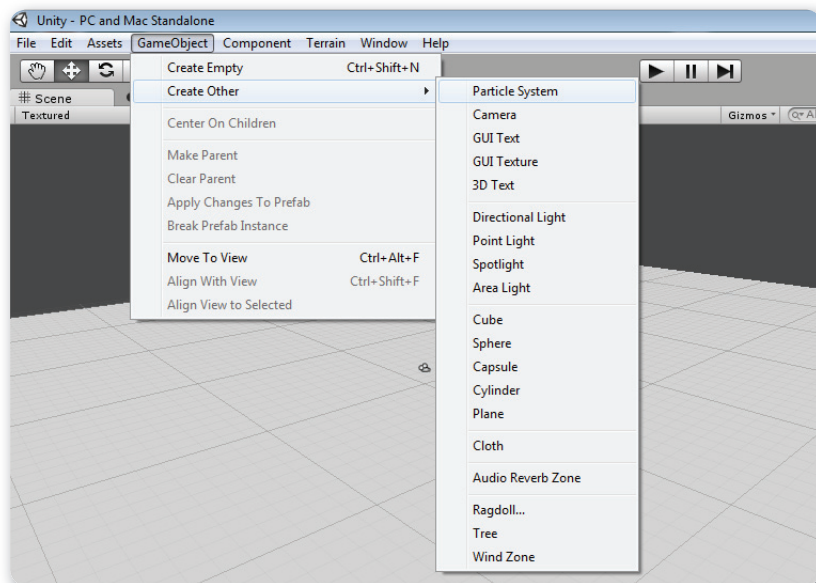
Con esto, creamos un sistema de partículas que aparece en el editor y que empieza a llevar a cabo la emisión de partículas. En este momento, el sistema tiene valores por defecto con una emisión muy sencilla. Si observamos el editor de Unity, descubrimos varios elementos nuevos. En primer lugar, dentro del panel **Scene** vemos un pequeño cuadro de diálogo que tiene unas herramientas básicas para el control de la simulación del sistema de partículas. En ese panel, cuando el sistema de partículas está seleccionado, se lleva a cabo una simulación que nos permite ver el funcionamiento del sistema sin tener que iniciar la aplicación. En la parte superior de este cuadro de diálogo, encontramos dos botones. **Pause** coloca la simulación en pausa y hace



BILLBOARDING



El billboard es una técnica usada en las gráficas computacionales en 3D. Consiste en tener una geometría, generalmente un cuadro conocido como **Quad**, en el cual se coloca algún tipo de textura. Luego, la geometría es rotada de tal forma que su normal apunte directamente en dirección a la cámara. Si esta cambia de posición, la rotación del billboard se recalcula para que siempre apunte a la cámara. Esta es una forma sencilla y económica, en términos computacionales, para generar diversos tipos de efectos.



► **Figura 2.** La creación de un sistema de partículas es similar a la de cualquier otro objeto.

que esta se detenga momentáneamente hasta que oprimamos el botón que, en este caso, dirá **Simulate. Stop** también detiene la simulación. La diferencia con **Pause** consiste en detener la simulación y enviarla otra vez al tiempo 0; la pausa no regresa el tiempo.

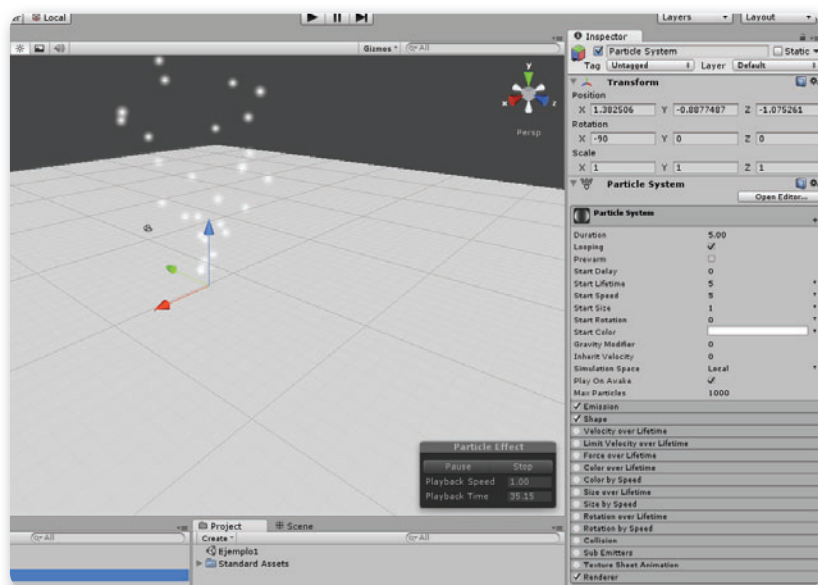
Debajo de los botones encontramos dos propiedades. La primera se llama **Playback Speed** e indica la velocidad con la que se efectúa la simulación. Con una velocidad de **1**, tenemos tiempo real. Si colocamos



CAMBIO EN EL TAMAÑO

Los tamaños grandes en las partículas pueden dar la sensación de suavizado, por lo que son útiles para algunos tipos de humos. El tamaño puede modificarse por medio de una curva a lo largo del tiempo, y podemos difuminar la partícula conforme pasa el tiempo. Un ejemplo de su uso podría ser el caso de la estela de humo de un misil. Para controlarlo, usamos el módulo **Size over Lifetime**, que tiene una propiedad llamada **Size**, sobre la cual trabajaremos.

valores mayores, el proceso se ejecuta más rápidamente, mientras que al colocar valores menores, la simulación se ejecuta con más lentitud. En la propiedad **Playback Time**, tenemos el tiempo en segundos que ha transcurrido desde el inicio de la simulación del sistema de partículas. Es muy útil, pues nos permite observar la forma en que se comporta el sistema según corre el tiempo. Puede ser de gran utilidad al depurar nuestro efecto y ver si todo se coordina tal como esperamos.



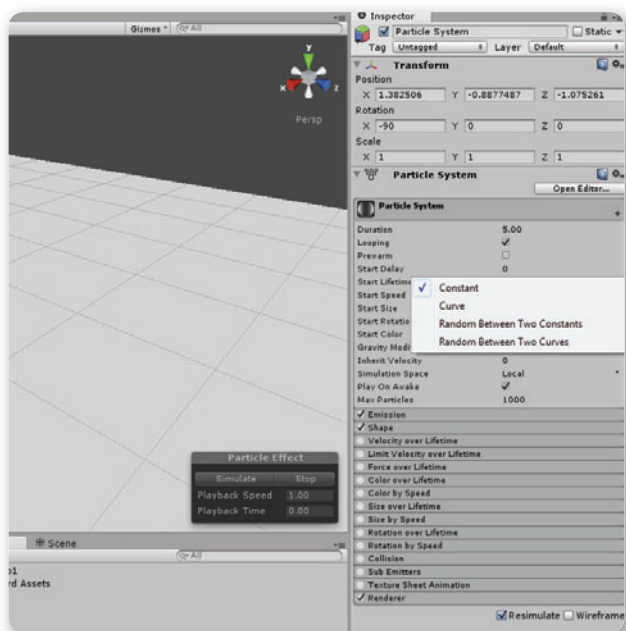
► **Figura 3.** Aquí podemos observar el sistema de partículas y algunas de las herramientas que usamos para su edición.

Comencemos a explorar el sistema de partículas y sus diferentes propiedades, ya que de esta manera conoceremos con facilidad sus capacidades y potenciales. La primera que encontramos se llama **Duration**, y simboliza el tiempo en segundos durante el cual el sistema de partículas estará emitiendo; por defecto, el valor es **5**. Esta propiedad se ve afectada por otra de nombre **Looping**, que permite que el sistema se repita y emita constantemente. Cuando el sistema está emitiendo de forma permanente, podemos considerar el valor de **Duration** como la duración de un ciclo del sistema.

Algunos efectos necesitan ser creados mientras el jugador los observa, y otros deben aparecer como si ya existieran desde antes que el jugador los vea. Cuando queremos que el sistema se muestre como si ya estuviera funcionando desde antes, usamos la propiedad **Prewarm**.

Si deseamos que el sistema de partículas no arranque de inmediato con la emisión de las partículas, colocamos un valor mayor que **0** para la propiedad **Start Delay**. Esta requiere un valor expresado en segundos. Es conveniente experimentar con el sistema de partículas al mismo tiempo que vemos las propiedades, para comprenderlas mejor. No hay que olvidar que es posible resetear el tiempo del sistema, para así observar su comportamiento desde el inicio con el nuevo valor.

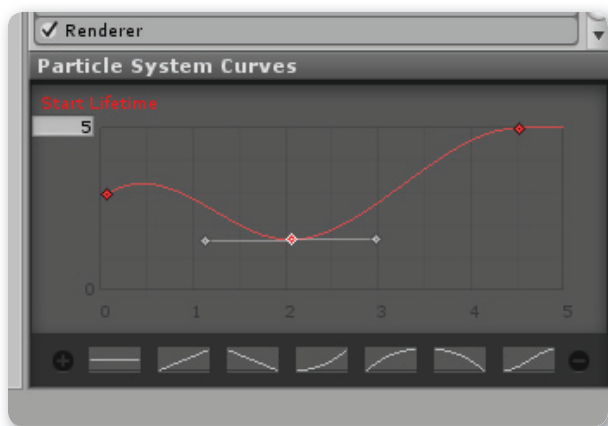
Las partículas tienen un tiempo de vida que se controla usando la propiedad **Start Lifetime**. La partícula hace una cuenta regresiva y, cuando llega al valor **0**, muere. Esta propiedad puede aceptar valores de diferentes tipos, y si hacemos clic en el icono del triángulo que se encuentra a la derecha, podemos verlos.



► **Figura 4.** Algunas propiedades pueden tener diferentes tipos de valores, los cuales se muestran en esta figura.

Editar los diferentes tipos de datos

Un tipo de valor que se puede colocar en las propiedades es **Curve**. Es una función con relación al tiempo del ciclo del sistema de partículas que nos indica un valor determinado en cierto momento. Supongamos que deseamos que el valor **Start Lifetime** sea diferente en diversos momentos del ciclo. Cuando seleccionamos el valor **Curve**, en la parte inferior aparece un editor de curvas. En el eje horizontal está el tiempo; no hay que olvidar que este representa el tiempo del ciclo del sistema de partículas. En el eje vertical, se representa el valor de la propiedad. En la parte inferior, hay botones que nos permiten colocar un tipo de curva predefinido. Por medio del mouse, podemos modificar los valores de los puntos de control en la curva. Para adicionar nuevos puntos, debemos simplemente hacer doble clic donde sea que los necesitemos.



► **Figura 5.** Con la práctica podemos aprender a utilizar el editor de curvas de manera muy sencilla.

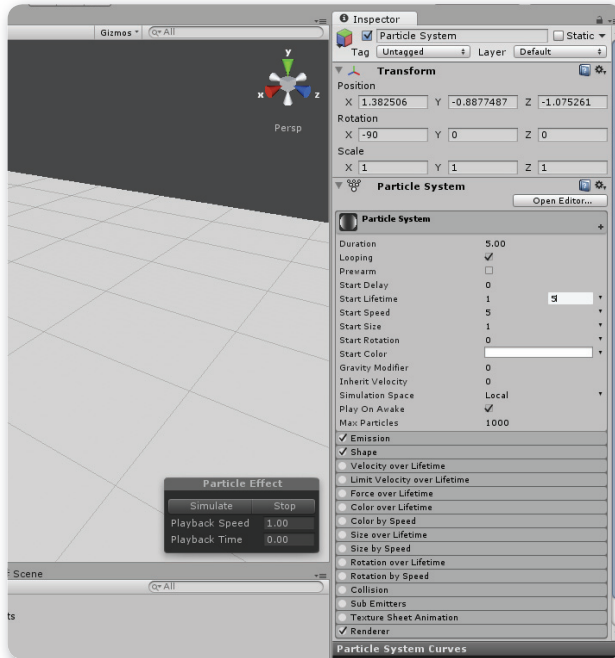


MÁS SOBRE BILLBOARDING



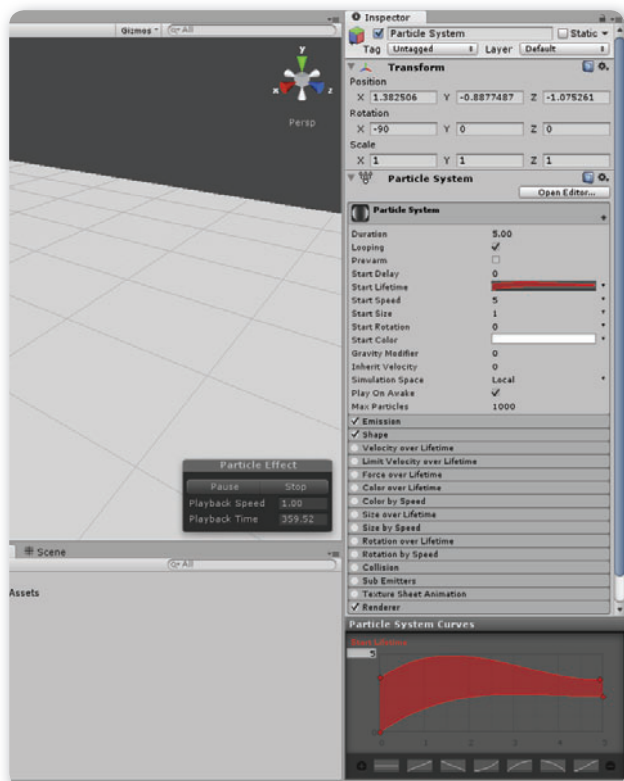
Para conocer más sobre las diferentes maneras en que se puede implementar el billboarding y las matemáticas que lo sostengan, podemos consultar el sitio web www.lighthouse3d.com/opengl/billboarding. En este se indica cómo llevar a cabo billboarding haciendo uso de **OpenGL** y **C++**. Los conceptos pueden transferirse a cualquier lenguaje y API gráfica. En Unity, no necesitamos programar el billboarding, pues el sistema de partículas lo hace por nosotros.

Otro tipo de dato que se usa en algunas propiedades se conoce como **Random between two constants** (aleatorio entre dos constantes). En este caso, la propiedad presenta espacio para escribir dos valores que definen el rango en el cual se seleccionará el valor aleatorio. El valor de la izquierda es el menor, y el de la derecha, el mayor.



► **Figura 6.** Al utilizar un valor aleatorio, indicamos el rango de valores posibles.

El último tipo que podemos utilizar es **Random between two curves** (aleatorio entre dos curvas). En este caso, lo que tenemos es un editor de curvas con dos curvas colocadas, cada una de las cuales puede ser editada y manipulada de manera independiente. Una de ellas representa el valor mayor del rango, y la otra, el menor. El valor aleatorio se encuentra en el área comprendida entre las dos curvas. Para un momento en particular del ciclo del sistema de partículas, podemos tener el valor de ambas y, de este modo, definir el rango sobre el cual el valor aleatorio es seleccionado.



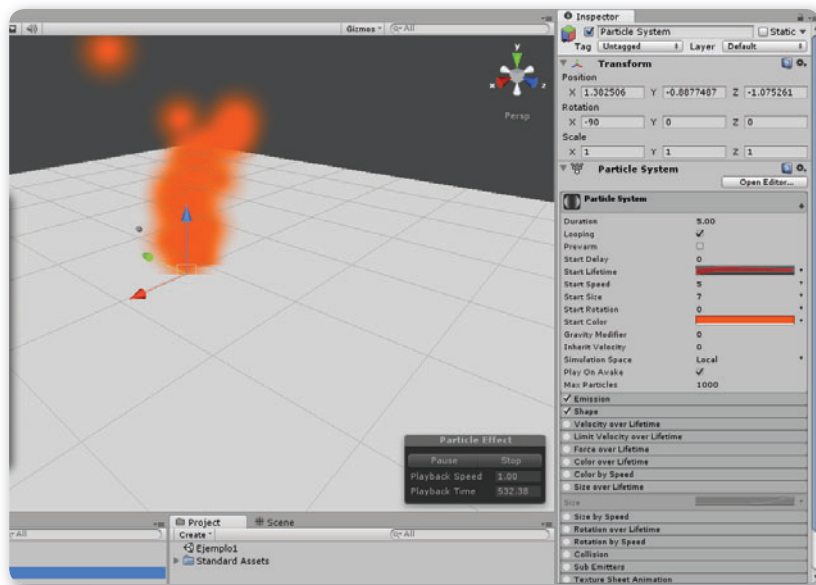
► **Figura 7.** En el editor, podemos tener dos curvas que van a definir el rango de valores aleatorios con relación al tiempo.

Más propiedades importantes

La siguiente propiedad que debemos conocer es **Start Speed**, con la que es posible definir la velocidad inicial de la partícula cuando esta es emitida. Al igual que con la propiedad anterior, podemos colocar distintos tipos de valores. Si en este momento decidimos incrementar el valor, veremos que en la simulación las partículas inmediatamente modifican su velocidad.

Las partículas, por default, tienen el valor **1** en su tamaño. En algunos efectos necesitaremos partículas grandes, y, en otros, será mejor utilizar algunas más pequeñas. La propiedad que se encarga de controlar este parámetro es **StartSize**.

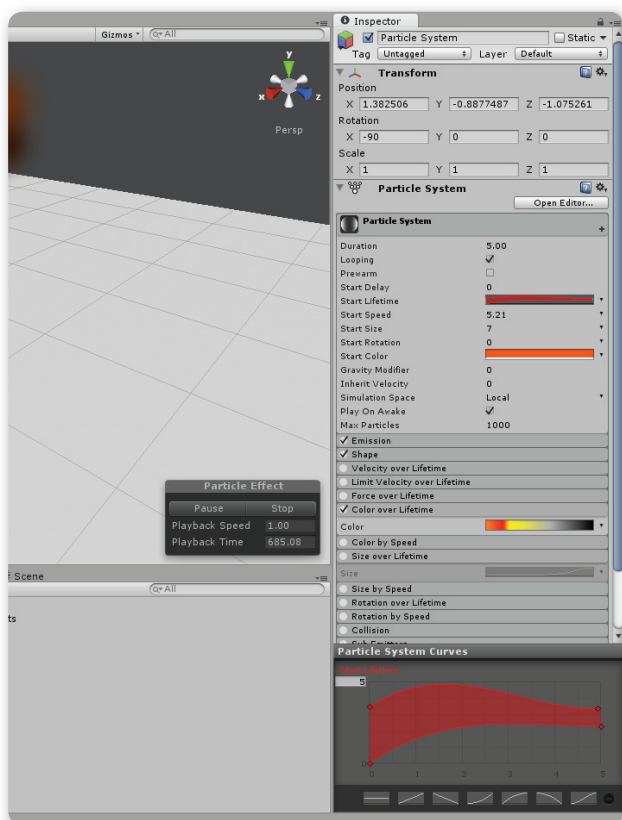
Hay dos colores importantes en el tiempo de vida de una partícula: el primero es el color en el momento de ser emitida, y el segundo, el color que tendrá durante el resto de su vida. Para colocar el color de emisión, hacemos uso de **Start Color**. Al hacer clic en esta propiedad, aparece un cuadro de diálogo con selector de color, donde elegimos el deseado. Podemos utilizar un color definido o un degradado. En este momento, por ejemplo, podemos colocar un color rojo-naranja y observar en pantalla cómo se modifican las partículas.



► **Figura 8.** Con la propiedad **Start Color** podemos modificar el color de emisión de las partículas.

Ahora vamos a cambiar el color de las partículas a lo largo del tiempo. Primero activamos el módulo **Color over lifetime** haciendo clic en el botón de tipo radio que se encuentra del lado izquierdo. En su interior, hay una propiedad llamada **Color**. En su tipo seleccionamos **Gradient**, lo cual nos permite crear un degradado, en el que podemos adicionar colores y modificar sus posiciones a lo largo del tiempo. El eje horizontal del editor es el tiempo, empezando del lado izquierdo y finalizando con la vida de la partícula en el lado derecho. Ahora podemos colocar

diferentes tonos de rojo, naranja y grises hacia el final. Con esto, tendremos un efecto parecido al humo. Todos estos cambios se observan directamente en la simulación del sistema de partículas.

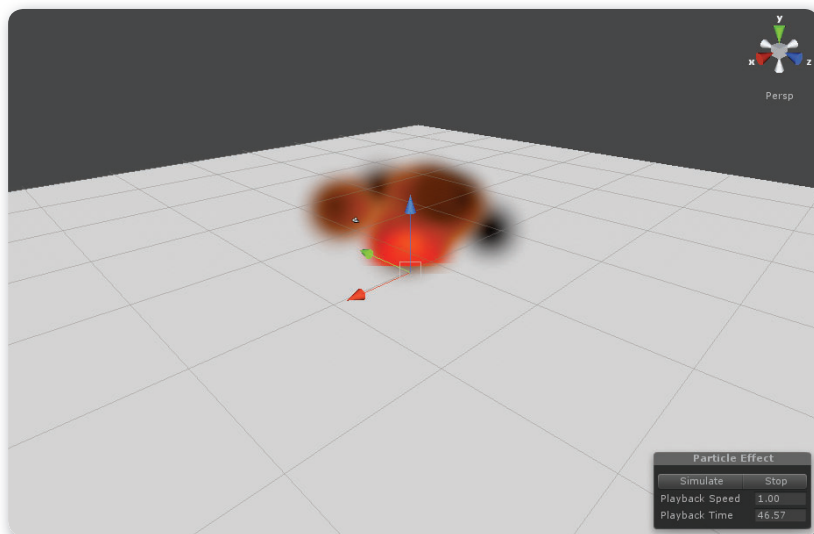


► **Figura 9.** Es posible indicar cuáles son los colores de la partícula a lo largo de su ciclo de vida.

Con esta nueva versión del sistema de partículas, resulta más fácil integrarlas al sistema de física. Uno de los elementos que podemos utilizar es la gravedad. En algunos casos, deseamos que esta afecte a las partículas. Por ejemplo, el sistema de partículas que representa una fuente requiere de la gravedad para lucir real. Este parámetro se controla por medio de la propiedad **Gravity Modifier** o modificador de gravedad. Este valor es un escalar sobre la gravedad del sistema. El

valor por defecto es **0**, e indica que las partículas no se verán afectadas por la gravedad. Hay que tener cuidado con esta propiedad, pues si las partículas no tienen una velocidad muy alta al ser emitidas, incluso valores pequeños pueden afectarlas fuertemente. En el experimento que estamos llevando a cabo, colocamos **0.2** en **Gravity Modifier**. Veamos cómo se alteran las partículas con este valor.

Otra propiedad que resulta importante es **Max Particles**, con la cual indicamos la cantidad máxima de partículas que puede tener el



► **Figura 10.** Esta imagen muestra claramente la manera en que la gravedad afecta al sistema de partículas.



SISTEMAS CONTINUOS Y DE UN SOLO TIRO



Algunos efectos creados con sistemas de partículas tienen que ser continuos; por ejemplo, una fogata que esté encendida constantemente necesita tener la propiedad **Looping** seleccionada. En el caso de una explosión, el sistema de partículas solo debe ejecutarse una vez, por lo que deseleccionamos la propiedad **Looping**. Otros factores que debemos considerar son el tiempo de vida de las partículas y sus velocidades. De esta manera, lograremos el efecto correcto.

sistema. Es una forma de controlar la densidad. Para algunos efectos, necesitaremos una mayor cantidad de partículas, y esta es la manera de hacerlo. Para tener un mayor control sobre la emisión, usamos el módulo **Emission**, que solamente tiene dos propiedades. La primera es **Rate** e indica la cantidad de partículas que se emiten en un segundo. Podemos colocar cualquier valor numérico positivo y, en conjunto con **Max**

Particles, tener un buen control de la emisión. La segunda propiedad es **Burst** y permite que, cada cierto tiempo, tengamos un brote extra de partículas.

LA PROPIEDAD BURST
PERMITE QUE, CADA
CIERTO TIEMPO,
TENGAMOS UN BROTE
EXTRA DE PARTÍCULAS

Anteriormente mencionamos que es posible cambiar la forma del emisor de partículas. Por defecto, Unity coloca la emisión en forma de cono. Cuando usamos este tipo de forma, podemos controlar el ángulo de emisión, es decir, cuánto se esparce y el radio correspondiente. La siguiente es la forma de hemisferio, y es la que tendría una media naranja. Con ella, podemos controlar el radio y, también, si la emisión se lleva a cabo

desde la superficie del hemisferio o no. La forma de esfera funciona de manera similar, pero en este caso tenemos la forma de una naranja completa, y la emisión se realiza en todas las direcciones. Por lo general, los emisores de esta forma están separados del piso, de modo de observar la emisión en todas las direcciones. El siguiente emisor tiene la forma de una caja y es muy útil cuando la emisión debe lucir como una columna. Con este tipo de emisor podemos cambiar el tamaño que tiene la caja en cada uno de los ejes. Si deseamos hacer un muro de fuego, esta forma es una buena opción. El último tipo de forma se conoce como **Mesh**. En este caso, debemos tener una geometría cuya superficie será



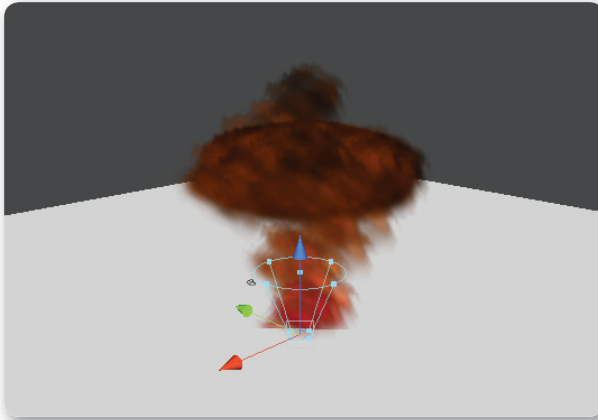
EL USO DE LAS RÁFAGAS



Una propiedad importante en nuestros efectos es **Bursts**, que nos permite crear ráfagas o emisiones repentinas de partículas a lo largo del ciclo. Estas son partículas extras que se adicionan al sistema. El hecho de tener una ráfaga de partículas extras en un momento dado nos permite crear efectos tales como explosiones en forma de hongo. Solo hay que darle la orden dentro del ciclo e indicar la cantidad de partículas que deseamos que tenga.

usada como emisor de partículas. Un ejemplo de su uso podría ser en el caso de un objeto que está ardiendo.

El módulo **Renderer** se encarga de todo lo relacionado con el dibujo de la partícula. En él lo que nos interesa es la propiedad **Material**, en la que podemos indicar cuál es la textura que se va a utilizar con la partícula. Este material debe crearse previamente. El cambio de la textura modifica mucho la experiencia visual del sistema. Por ejemplo, al sistema que hemos estado creando podemos colocarle el material **Dust Material 1** y, luego, observar la emisión.



► **Figura 11.**
El cambio de material en la partícula puede modificar mucho el sistema que estamos creando, y hasta mejorarlo.

Crear un subsistema

El sistema de partículas que hemos creado nos permite tener un buen efecto, pero algunas veces se necesita más de un sistema para lograr algo mejor. En este caso, lo que hacemos es crear uno o varios sistemas de partículas que tendrán parenting con relación al original. Estos sistemas extras sirven cuando precisamos partículas que luzcan o se comporten de manera diferente. Hay diversas maneras de hacer esto. En este caso, nos dirigimos al módulo principal del sistema de partículas, llamado **Particle System**, y, donde está su nombre, hacemos clic con el botón derecho. Aparece un menú del cual seleccionamos la opción **Create Particle System**; se crea un nuevo sistema de partículas que tiene parenting con relación al original. El parenting puede observarse fácilmente en el panel **Hierarchy**.



► **Figura 12.**
El nuevo sistema ya puede observarse como subsistema del principal.

Para experimentar, haremos que el subsistema se mueva como si el viento soplara sobre las partículas. Para lograrlo, adicionamos una fuerza a las partículas. Cuando usamos fuerzas, debemos tener en cuenta que se aplican únicamente al sistema o subsistema con el que estemos trabajando. Si deseamos que la fuerza se aplique a todos los subsistemas, entonces tenemos que agregarla en cada uno de ellos. La fuerza es controlada por medio del módulo **Force over lifetime**. Al activarlo, vemos que tiene varias propiedades. En este momento nos interesan las marcadas como **X**, **Y**, **Z**. En ellas, colocamos la cantidad de fuerza que recibe la partícula en cada uno de los ejes. Para este ejemplo, ponemos **5** en **X**. Si observamos inmediatamente el panel **Scene**, veremos que las partículas se mueven con rapidez en la dirección de ese eje. Este efecto da la ilusión de que el viento está soplando y se las lleva.

Los tipos que podemos colocar en la fuerza son valores constantes, pero también es posible colocar curvas, valores aleatorios e, incluso,



SELECCIÓN CORRECTA DE PROPIEDADES



Un punto importante que debemos tener en cuenta al momento de hacer uso de las diferentes propiedades en el sistema de partículas es la compatibilidad que existe entre dichas propiedades. Hay algunas que solamente tienen efecto cuando otras tienen un determinado valor o han sido seleccionadas. Un ejemplo es la propiedad **Start Delay**, que no puede usarse si la propiedad **Prewarm** ha sido seleccionada.

aleatorios entre dos curvas. La forma de editar estos valores ya la hemos visto a lo largo del capítulo. El uso de curvas puede ayudarnos a generar en el jugador la sensación de que las partículas que observa tienen algún tipo de aceleración.

El uso del editor

Unity nos provee de una pequeña herramienta que puede ayudarnos con la edición. Su uso es similar al que hemos visto con el panel **Inspector**, pero el sistema y los subsistemas aparecen listados de manera horizontal, y tiene la ventaja de que permite hacer las ediciones un poco más rápido. Para activar el editor, hacemos clic en el botón **Open Editor**, en el módulo principal del sistema de partículas. Al hacerlo, se abre una nueva ventana que es el editor, donde vemos dos secciones. En el lado izquierdo están listados los sistemas, empezando por el principal y seguido de los subsistemas. Podemos observar sus módulos y propiedades de manera similar al panel **Inspector**. En el lado derecho, está el editor de curvas. La selección entre el panel **Inspector** y la ventana de edición es personal, ya que ambos sirven para realizar la misma función. La ventaja que presenta el editor es la capacidad de hacer crecer la ventana y que la edición de las curvas sea más sencilla. La desventaja es que requiere espacio en Unity y oculta otros paneles que podemos precisar. Para regresar al sistema de edición tradicional, simplemente cerramos la ventana del editor.

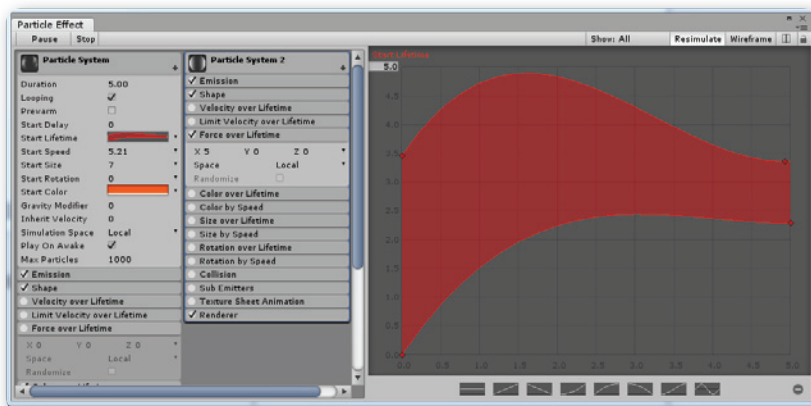
LA VENTAJA QUE
PRESENTA EL EDITOR
ES PERMITIR QUE LA
EDICIÓN DE CURVAS
SEA MÁS SENCILLA



LA VELOCIDAD DE LAS PARTÍCULAS



Hemos visto que podemos colocar una velocidad inicial para las partículas, pero no siempre es necesario tener el mismo valor. Es bueno poder variar la velocidad de la partícula según transcurre el tiempo, ya sea para acelerarla o detenerla. Esto puede hacerse mediante el uso de un sistema de fuerzas, pero es más fácil implementarlo con el módulo **Velocity over Lifetime**. Únicamente debemos colocar la velocidad en cada uno de los ejes, y como podemos utilizar la curva para los valores, resulta fácil de indicar.



► **Figura 13.** El editor del sistema de partículas nos permite editar el sistema como un todo y navegar fácilmente entre los subsistemas.

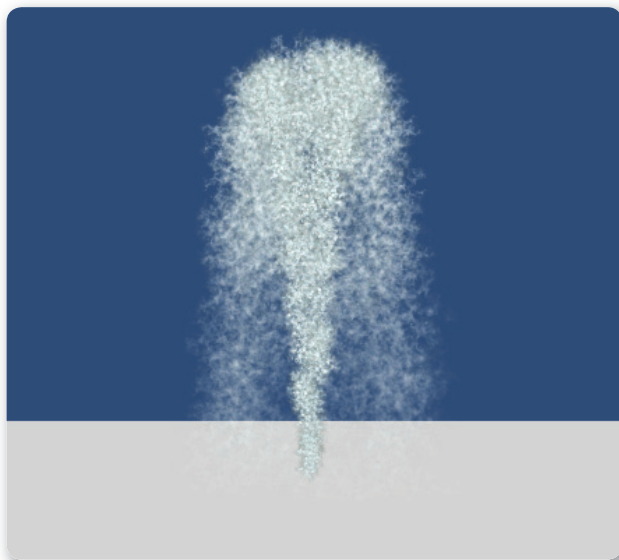
Efectos preconstruidos

Veamos ahora algunos de los efectos preconstruidos que Unity incluye en el paquete de partículas; podemos utilizarlos en nuestras aplicaciones. Es bueno conocerlos y verlos por lo menos una vez, pues es posible modificarlos para crear nuestros propios efectos, o pueden darnos ideas sobre cómo crearlos.

PODEMOS MODIFICAR
LOS EFECTOS
PRECONSTRUIDOS
PARA CREAR LOS
NUESTROS

Estos efectos fueron importados al inicio, cuando creamos la aplicación, y podemos encontrarlos en el panel **Project**. En él vemos la carpeta **Standard Assets** y, dentro de ella, **Particles**. Los efectos están organizados dentro de carpetas también, y una de ellas es utilizada para guardar los recursos necesarios de estos efectos. Además, encontramos una carpeta donde se guardan emisores de partículas con el sistema de partículas anterior.

En caso de que el sistema de partículas tenga el esquema anterior de edición, podemos observar propiedades similares a las que hemos visto. Algunas de ellas, como puede ser el color de la partícula, se repiten varias veces. No deberíamos de tener problemas para editarlas. Aun así, la edición es más sencilla con el nuevo esquema.



► **Figura 14.** La fuente de agua es uno de los efectos que Unity ofrece, y luce de manera muy realista.

Al observar estos sistemas, podemos tratar de replicarlos nosotros mismos en el nuevo sistema.

En este capítulo, hemos visto los elementos más importantes del sistema de partículas de Unity. Ya podemos construir nuestros propios efectos, pero esto requiere de práctica y, sobre todo, de experimentación con los diferentes módulos y propiedades.



RESUMEN

Unity incluye un nuevo sistema de partículas que nos permite crear diferentes efectos. Está dividido en módulos, cada uno de los cuales se encarga de controlar una función específica. Los módulos tienen propiedades en su interior. Algunas nos permiten colocar valores directos, curvas, valores aleatorios y degradados de color. El tiempo en el sistema de partículas se mide en segundos y se agrupa en ciclos. Algunas propiedades no se pueden usar en combinación con otras. Es posible tener parenting en los sistemas de partículas para lograr efectos más avanzados.

Actividades

TEST DE AUTOEVALUACIÓN

- 1 ¿Cómo se usa **Curve** para crear un ciclo de vida tres veces más grande?
- 2 ¿De qué manera se puede colocar una mayor velocidad al final de la vida de las partículas?
- 3 ¿Qué se usa para asignar diversos colores a las partículas?
- 4 ¿Qué debe hacerse para modificar la gravedad?
- 5 ¿Para qué se utiliza **Curve**?
- 6 ¿Qué es el valor aleatorio entre dos curvas?
- 7 ¿Cómo funciona el degradado de colores?
- 8 ¿Qué características tiene el editor de partículas?
- 9 ¿Es posible hacer uso de parenting con partículas?
- 10 ¿Qué ventaja tiene el editor de partículas a los valores del panel **Inspector**?
- 11 ¿De qué forma se puede cambiar el material de la partícula?
- 12 ¿Qué propiedades permiten rotar las partículas?

ACTIVIDADES PRÁCTICAS

- 1 Haga un sistema de partículas que luzca como un tornado.
- 2 Agregue un sistema de partículas a un objeto en movimiento para que parezca un misil.
- 3 Realice un sistema de partículas que se asemeje a una fuente.
- 4 Haga un sistema para un efecto de lluvia.
- 5 Haga un sistema para un efecto de lava.

Blender y animaciones

Hasta ahora vimos varios ejemplos en este libro, sin embargo, los objetos que hemos utilizado no son más que geometrías básicas. En los videojuegos profesionales, solemos tener modelos animados, por lo que en este capítulo veremos cómo incorporarlos en nuestros proyectos de Unity apoyándonos en el modelador 3D Blender (**www.blender.org**). Además, aprenderemos a añadirles materiales y texturas, para que los modelos se vean reales.

▼ Blender.....	226
Creación del robot	227
Materiales y texturas.....	233
Animación	238
Exportar a Unity.....	242

▼ Resumen.....	247
▼ Actividades.....	248



Blender

Blender es un software de tipo **Open Source** (de código libre), que nos permite modelar y animar objetos en 3D. Ya tiene mucho tiempo desde su aparición, lo que ha permitido que sus herramientas evolucionen y, hoy, sea comparable en cuanto a capacidades con muchas de sus contrapartes comerciales. El manejo de Blender puede aprenderse con gran facilidad, y el cambio que tuvo recientemente en su interfaz lo hace aún más simple.

En Windows la instalación es similar a la de cualquier otro software. Al ejecutar Blender, vemos una pantalla de inicio, que puede variar

BLENDER ES UN
SOFTWARE DE CÓDIGO
LIBRE QUE NOS
PERMITE MODELAR Y
ANIMAR OBJETOS EN 3D

dependiendo de la versión que se esté utilizando, que nos permite acceder a nuestros documentos más recientes, como también a sitios web que puedan resultarnos de utilidad.

La interfaz tal vez luzca un poco diferente de lo que estamos acostumbrados a ver en Windows, sin embargo, su manejo es sencillo. Haciendo clic con el mouse fuera de la pantalla de inicio, entramos en el editor 3D. La parte central muestra el mundo 3D donde modelamos los objetos. En la parte izquierda, encontramos un panel con

herramientas que nos permiten trabajar con el objeto seleccionado o el que se ha creado. Del mismo lado, hay otro panel con más herramientas, que cambian según el elemento seleccionado. En la zona inferior vemos un **timeline** (línea de tiempo) que usaremos para crear las animaciones.

Aunque Blender tiene herramientas poderosas de modelaje y animación 3D, que nos permitirían crear personajes complejos, nos



MOVER LOS VÉRTICES



El movimiento libre del mouse algunas veces no es adecuado para colocar los vértices en la posición que deseamos. En estos casos, es mejor usar el **widget** que provee Blender, y que muestra los ejes coordinados sobre el vértice, por medio de flechas de colores. Al usar la flecha, el vértice se moverá solo a lo largo de ese eje en particular. Esto nos ayuda a colocar lo que estemos editando de forma más precisa, ya que funciona también con aristas y caras.

concentraremos en desarrollar un personaje sencillo que será animado y, posteriormente, exportado a una aplicación en Unity de manera muy simple y en pocos pasos. Controlar la animación desde Unity puede requerir de muy pocas líneas de código o, también, puede hacerse mediante la creación de un controlador avanzado, si fuera necesario.

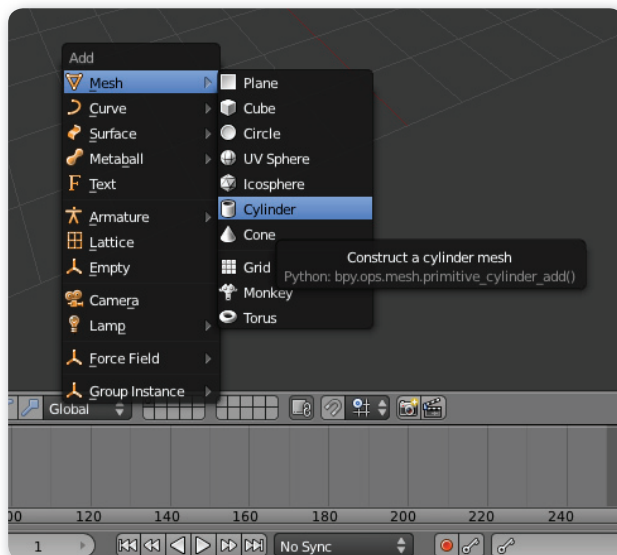


► **Figura 1.** Al ejecutar el software de modelado y animación 3D Blender, aparece el editor con su pantalla de inicio.

Creación del robot

El personaje animado que vamos a crear es un pequeño robot. Si bien se compone de objetos sencillos, también es posible utilizar otros más complejos. Para empezar, ejecutamos Blender y entramos en el editor. Por default, Blender crea un cubo, pero como nosotros no lo usaremos, vamos a eliminarlo. Para hacerlo, primero lo seleccionamos, con lo cual mostrará un marco de color naranja a su alrededor. Entonces procedemos a oprimir la tecla **X**; aparece un cuadro de diálogo que nos permite confirmar o rechazar la eliminación. Una vez que liberamos el espacio, podemos adicionar los objetos que forman el cuerpo del robot. Con el cursor sobre el área de edición, oprimimos la combinación de teclas **SHIFT + A**. Al hacerlo, aparece el menú **Add**, que muestra las

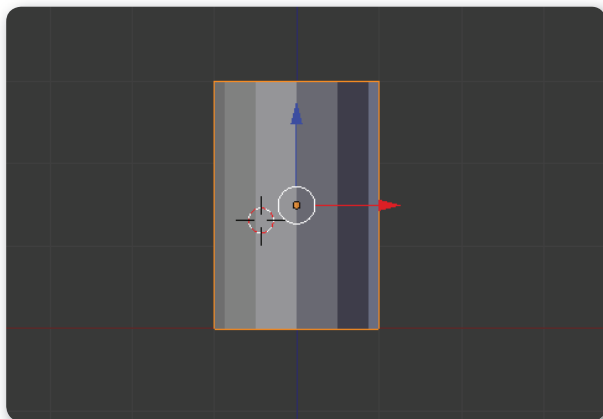
clasificaciones de objetos por añadir. Lo que necesitamos adicionar es un objeto de tipo **Mesh**, que tiene una geometría construida por vértices, del tipo que maneja Unity. Al ubicarnos sobre esta opción, se muestra una nueva lista de donde seleccionamos el objeto **Cylinder**, que usaremos para el cuerpo del robot.



► **Figura 2.** El menú **Add** muestra muchas opciones, de las cuales seleccionamos el objeto **Cylinder**.

Un cilindro aparece en la pantalla. Para editarlo, primero vamos a colocar la cámara con la vista frontal, oprimiendo la tecla **1** del teclado numérico, seguida de la tecla **5**. Así tendremos una vista frontal ortogonal que resultará muy útil. En el panel **Object Tools**, ubicado del lado izquierdo, encontramos una sección llamada **Add Cylinder**, que se utiliza para determinar las características del cilindro que estamos creando. Colocamos valor **12** para los vértices, **1** para el radio y **3** para la profundidad. Luego, una vez que el cilindro haya sido creado, procedemos a ubicarlo en la posición adecuada.

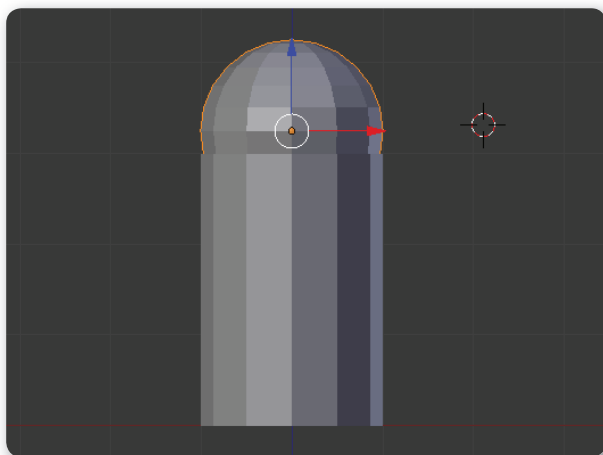
Necesitamos abrir un nuevo panel donde se encuentran las propiedades de transformación. Para hacerlo, colocamos el cursor sobre el área de edición y oprimimos la tecla **N**. En la parte superior



► **Figura 3.**
El cilindro es colocado en su posición, y lo usaremos para el cuerpo del robot.

se encuentran las coordenadas de locación que debemos utilizar para colocar el cilindro en **(0,0,1.5)**.

De manera similar agregamos los demás objetos que componen el robot. Ahora necesitamos agregar una esfera como cabeza, y por este motivo, siguiendo los mismos pasos, ponemos una **UV Sphere**. Para esta figura, también existe una sección en el panel izquierdo que nos permite colocar sus características. Tendrá **12** anillos, **12** segmentos y un tamaño de **1**. La esfera será colocada en la posición **(0,0,3.25)**. Podemos hacerlo fácilmente utilizando el panel **Transform**.



► **Figura 4.**
En la parte superior del cilindro queda la esfera que representa la cabeza del robot.

LA CREACIÓN DE LAS PIERNAS DEL ROBOT REQUIERE DEL USO DE DIFERENTES HERRAMIENTAS

La creación de las piernas del robot es un proceso más complejo, para el que necesitamos usar diferentes herramientas. La base es un cubo, cuyas dimensiones vamos a modificar, y al que luego le editaremos los vértices para que luzca diferente. La segunda pierna, simplemente, será una clonación de la primera (más adelante veremos cómo realizarla). Empezamos creando un cubo y lo colocamos en las coordenadas **(1.1,0,0)**. En el panel **Transform**, debajo de la ubicación, aparece **Scale**. Colocamos la escala de **0.1** en **X**, **1** en **Y** y **1** en **Z**, lo que genera una caja delgada con la que podemos trabajar. Cuando la caja está seleccionada, se muestra con un marco naranja, y al oprimir la tecla **TAB**, entramos en modo de edición de objeto. En este modo podemos modificar la geometría del objeto al editar sus vértices, aristas o caras. Los elementos seleccionados son editados y se muestran en color naranja. Al oprimir la tecla **A**, se seleccionan o deseleccionan todos los elementos del

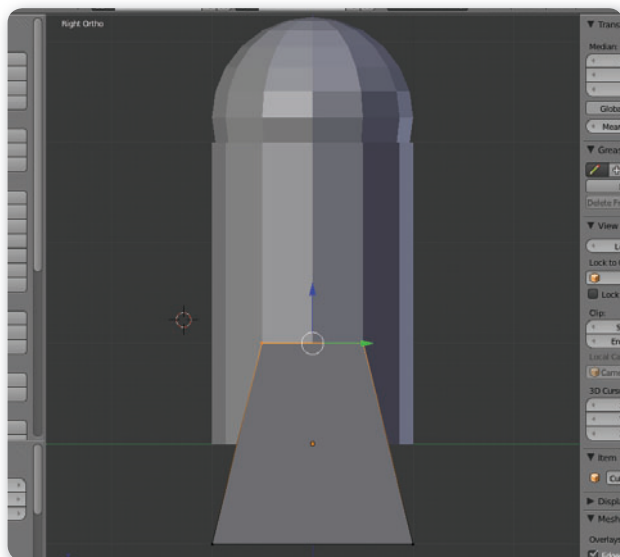
objeto. Debemos presionarla hasta que todos los elementos estén deseleccionados. En la parte superior del timeline, encontramos una serie de botones que nos ayudan en la edición. Por default, Blender trabaja con la edición de vértices, por lo que podemos seleccionarlos colocando el cursor sobre el vértice que deseamos y oprimiendo el botón derecho del mouse. En este caso, es más fácil editar con la vista de la cámara en el lado derecho de la pantalla, y lo hacemos pulsando la tecla **3** del teclado numérico. Con esta vista, seleccionamos uno de los vértices superiores y lo movemos. Cuando esté en la posición deseada, hacemos clic con el botón izquierdo. Editamos los vértices superiores de la caja hasta que se vean como se muestra en la **Figura 5**.



DESCARGAR BLENDER



Blender se puede utilizar en diversas plataformas, entre ellas, Windows. La descarga es gratuita, y en el sitio también podemos encontrar muchos recursos y tutoriales. Al momento de descargarlo, debemos asegurarnos de estar descargando el archivo adecuado para nuestro sistema operativo. En Internet encontraremos muchos recursos para este software. El sitio web oficial de Blender es **www.blender.org**.



► **Figura 5.** Al editar los vértices, logramos darle forma a la pierna del robot.

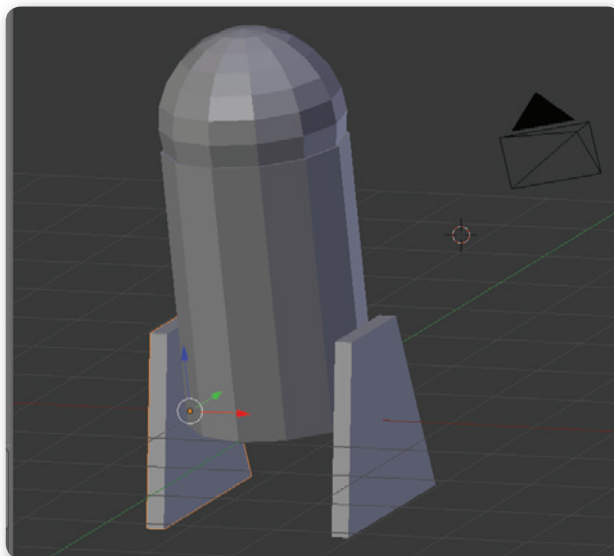
Crear otra pierna que luzca igual siguiendo los mismos pasos nos tomaría demasiado tiempo, pero tenemos la opción de clonar o duplicar el objeto. En primer lugar, salimos del modo de edición de objeto presionando la tecla **TAB**. La pierna tiene que estar seleccionada, y en el panel de la izquierda buscamos un botón que dice **Duplicate Objects**. Al hacer clic en él, creamos un objeto igual al seleccionado, y, si movemos el mouse, podemos colocarlo en otra posición. Es más sencillo ubicarlo indicando directamente las coordenadas en el panel **Propiedades de transformación**. Para la segunda pierna, la posición es **(-1.1,0,0)**.



UBICAR LA GEOMETRÍA



Muchas veces ocurre que la geometría desdoblada no se adapta exactamente a lo que deseamos. Podemos moverla usando la tecla **G** y deslizando el mouse. La tecla **S** nos permite cambiar la escala de la geometría, y la tecla **R**, rotarla. Para confirmar el cambio, presionamos el botón izquierdo del mouse, y, si deseamos eliminarlo, antes de confirmar, simplemente oprimimos la tecla **ESC**.



► **Figura 6.** El robot ahora tiene dos piernas gracias a la herramienta de duplicación.

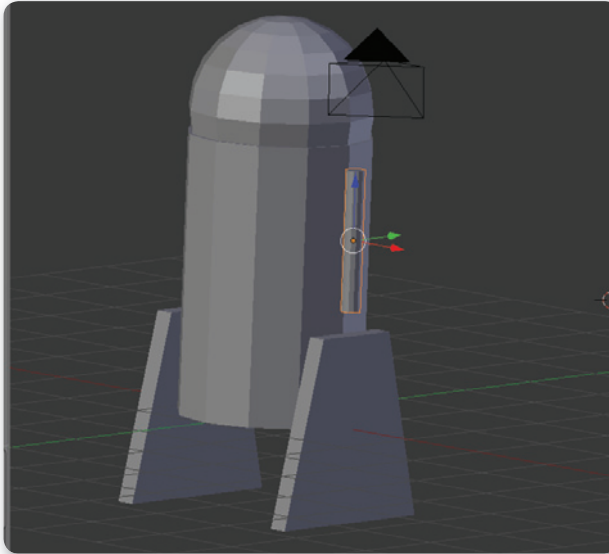
El cuerpo del robot ya casi está listo. Vamos a adicionarle un elemento extra, que consiste en un brazo con el cual llevará a cabo su ataque. Para crearlo, usamos una geometría en forma de cilindro y la colocamos a un lado del cuerpo. Posteriormente, este brazo tendrá su propia animación. Este cilindro no será tan grande como el cuerpo del robot, por lo que debemos colocarle **12** en la cantidad de vértices. El radio tiene que ser pequeño, porque deseamos que el brazo sea delgado. Le asignamos valor **0.1**. La longitud del brazo es de **1.5**, pero luego, dentro de la animación, modificaremos este valor. El brazo estará a un costado, en la posición **(1.1,0,2)**.



CUADROS LLAVE Y LA ANIMACIÓN



Cuando creamos una animación, no siempre es necesario colocar manualmente todos los cuadros. Lo que hacemos es indicar solo los más importantes, llamados **keyframes** (cuadros llave); posteriormente, Blender interpolará por nosotros los demás. Esto reduce mucho el tiempo necesario para desarrollar una animación.



► **Figura 7.** El brazo del robot ha sido colocado a un costado del cuerpo, por arriba de la pierna.

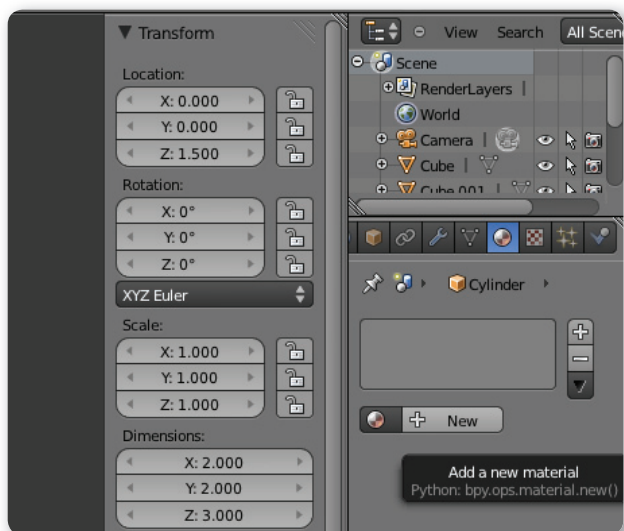
El modelo del robot está listo, pero es necesario hacerlo lucir mejor, por lo que vamos a colocarle materiales y una textura.

Materiales y texturas

Blender permite usar materiales y texturas en los modelos que se crean. Para nuestro robot, utilizaremos ambos recursos y los exportaremos a Unity junto con la geometría del robot. El cuerpo, las piernas y el brazo tendrán materiales, y la cabeza llevará una textura.

La colocación de materiales es sencilla, y estos pueden ser exportados a Unity sin inconvenientes. Empezaremos por ponerle un material al cuerpo del robot, para lo cual lo seleccionamos. Debemos utilizar una herramienta que nos permita asignar los materiales. Para encontrarla, vamos al panel que se encuentra del lado derecho de la pantalla, donde aparece una jerarquía de objetos en la parte superior, e inmediatamente abajo, una tira de iconos. Estos nos permiten acceder a las herramientas y verlas clasificadas según diversos conceptos. Para navegar por esta herramienta, debemos hacer un arrastre con el botón central del mouse.

Buscamos un icono en forma de esfera llamado **Material**. Hacemos clic con el botón izquierdo del mouse, y, en la parte inferior del panel, aparecen las herramientas que usaremos para los materiales.



► **Figura 8.** Es muy fácil encontrar y utilizar las herramientas necesarias para colocar materiales.

Los pasos para agregar un material son sencillos: utilizamos un botón llamado **New**. Al hacer clic en él, aparecen nuevas herramientas, incluido un visor, donde podemos visualizar la forma en que luce el material.

El material que vamos a colocar en el cuerpo del robot tiene color amarillo. Para darle el color al material, usamos el componente difuso únicamente; es posible usar otros componentes, pero, para este

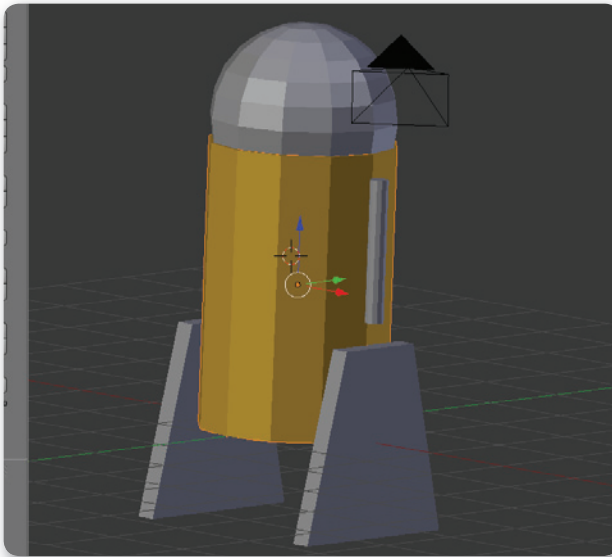


MINICURSO BLENDER



Para profundizar en el uso de las herramientas básicas de Blender, podemos acceder a una serie de videos publicados en www.youtube.com/user/nicosioired. El playlist donde se encuentran se llama: **Curso Blender en Español**, y esta puede ser una buena opción para entender el manejo de este programa fácil y rápidamente.

proyecto, no son necesarios. Buscamos la herramienta **Diffuse** (difuso), la cual presenta un rectángulo de color blanco. Al hacer clic en él, se abre un cuadro de diálogo para seleccionar el color. Podemos hacer la selección directamente en el círculo de colores, o determinar los valores de rojo, verde y azul que lo componen. Seleccionamos un tono amarillo y, luego de asignarlo al cuerpo del robot, podemos observar el resultado.



► **Figura 9.** La asignación de color al material es sencilla, y podemos observar el resultado en el objeto.

Podemos seguir los mismos pasos para los demás elementos del robot. Solo recordemos que es necesario seleccionar previamente el objeto, antes de asignarle el material. Para las piernas, creamos un material de color rojo, y en el brazo usamos uno azul.

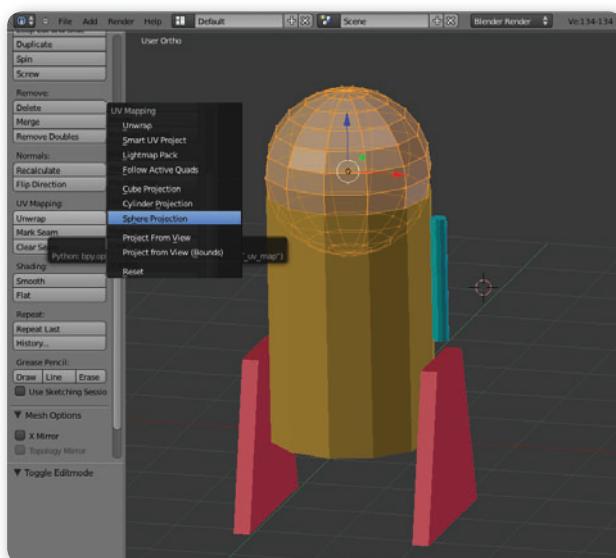
En la cabeza, en vez de usar un material, vamos a aplicar una textura, de modo que primero tendremos que crearla. Para hacerlo, podemos usar cualquier programa de dibujo y, luego de crear la imagen que deseemos, la salvamos en formato .JPG o .PNG.

La imagen que usaremos para la textura tiene un tamaño de **256** por **256** píxeles. No es necesario usar texturas más grandes para un modelo tan sencillo como el que tenemos. Dibujamos una cara, y para facilitar

su colocación, ubicamos la mayor parte de ella en el cuadrante superior izquierdo de la imagen. A esta imagen, le ponemos el nombre **CaraRobot**.

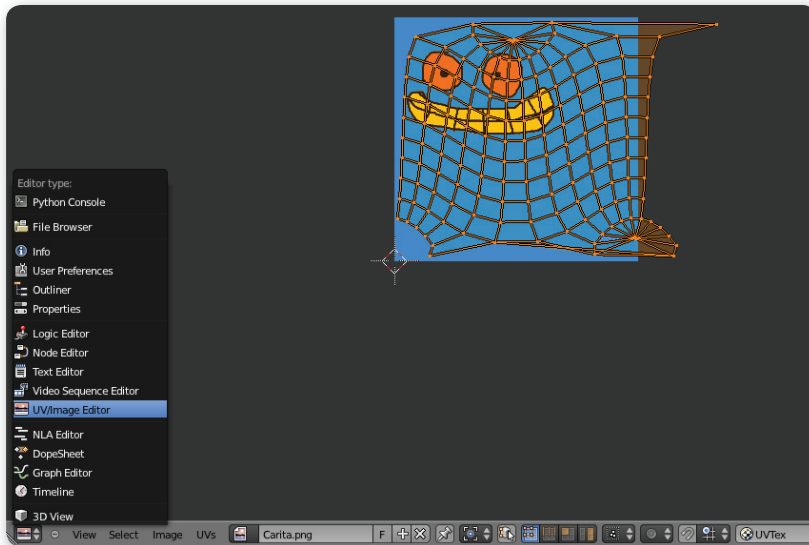
Asignar la textura requiere de varios pasos; empezamos por seleccionar la esfera. Es necesario entrar en el modo de edición del objeto cada vez que vamos a trabajar con sus texturas, por lo que presionamos la tecla **TAB**. Una vez en modo de edición, todos los vértices de la esfera deben ser seleccionados, y, para hacerlo, pulsamos la tecla **A** hasta que la esfera completa tenga color naranja. Con los vértices seleccionados, llevamos a cabo un proceso conocido como **Unwrap**, que consiste en aplanar la geometría 3D a un plano 2D para poder seleccionar qué partes de la textura corresponden a determinada cara del objeto. En el panel del lado izquierdo, hay una sección llamada **UV Mapping** (mapeo UV); hacemos clic en el botón **Unwrap** y aparece un menú con las diferentes versiones disponibles. La opción que más nos sirve para el tipo de geometría que estamos usando es **Sphere Projection** (proyección esférica), por lo que la seleccionamos.

Parecerá que no ha ocurrido nada, pero en realidad, para poder trabajar con **Unwrap**, debemos cambiar de editor. En la parte superior



► **Figura 10.** El algoritmo de Unwrap que mejor funcionará para nuestros fines es **Sphere Projection**.

del timeline, hay un menú que tiene, en su lado izquierdo, un botón con el icono de un cubo. Al hacer clic sobre él, se muestra otro menú con diferentes tipos de editores, de los cuales seleccionamos la opción que dice **UV/Image Editor** (UV/editor de imágenes). Aparece entonces la esfera desdoblada y colocada sobre un plano. Podemos asignar la textura correspondiente desde el menú **Image**, ubicado en la misma barra de menú que el selector de editores. En él, seleccionamos **Open Image** y elegimos la imagen **CaraRobot** generada previamente. Esta se muestra detrás de la geometría y queda asignada como textura.



► **Figura 11.** Con este editor podemos asignar la textura en la posición necesaria.



EL USO DE UNWRAP

Unwrap es una de las herramientas más útiles que ofrece Blender, pues permite colocarle una textura al objeto 3D fácilmente. Podemos colocar los vértices y las caras en las zonas exactas de la textura que deseamos que se muestren en el objeto. Para hacerlo, existen diferentes algoritmos y, dependiendo de la geometría del objeto, tendrán una utilidad particular. Lo mejor es experimentar con ellos, aunque también tenemos la opción de editar los vértices en forma manual para obtener una mejor precisión.

Una vez asignada la textura, regresamos a **3D View**, que seleccionamos en **Editor Type** y, después, salimos del modo de edición. Es muy posible que la textura no se muestre, debido al tipo de render que se está usando en ese momento. Por eso, será necesario indicar que deseamos **Textured**, que se selecciona en la barra de menú por encima del timeline. Su menú se presenta con una esfera y, al hacerle clic, muestra los diferentes tipos de render posibles. Elegimos el que tiene una textura, cuyo icono es una esfera blanca con rojo. Una vez que entramos en este modo, observaremos la textura aplicada en la cabeza del robot. En caso de que no nos guste cómo se ve, podemos repetir la operación para la asignación de texturas y mover la geometría desdoblada a una posición más adecuada. Con esto, ya tenemos la geometría del robot completa, y solo nos queda pasar a la animación.

Animación

Existen diferentes mecanismos para llevar a cabo la animación de las geometrías en 3D. En este caso, usaremos rotaciones y escalas en los objetos que lo componen para animarlo; otros métodos más avanzados usan deformaciones en las geometrías. Antes de efectuar cualquier tipo de animación, es muy importante planificar cómo deseamos que luzca y cuántos cuadros se necesitarán.

Para el caso de nuestro robot, tendremos tres animaciones, aunque siempre es posible agregar más. Una animación de espera cubrirá los cuadros de animación del **1** al **20**. La segunda cubrirá del cuadro **21** al **40**, y representará la manera de caminar del robot. En la última animación, el robot disparará, e iniciará en el cuadro **41** terminando en el **60**.

Comenzamos por crear la primera animación: el robot esperando. Esta es muy sencilla, ya que el robot únicamente moverá la cabeza de un lado



MODOS DE OBJETO

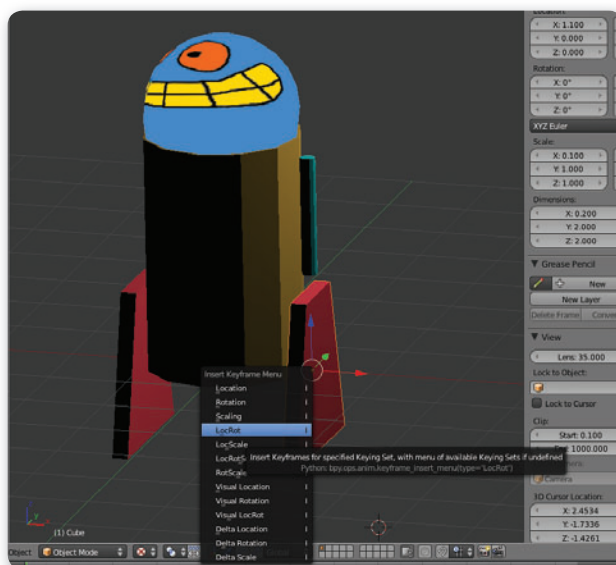


Los objetos pueden tener diferentes modos. El objeto puede estar o no seleccionado. La selección del objeto se hace por medio de clic con el botón derecho del mouse. Una vez seleccionado, podemos entrar en su modo de edición presionando la tecla **TAB** y regresamos al modo de selección pulsando esa tecla otra vez. Para conocer más sobre estos modos, conviene consultar la documentación sobre Blender.

a otro, por lo que debemos colocar el rango de la animación. En la parte inferior del timeline, vemos un **TextBox** que dice **Start** y otro que dice **End**; en ellos, colocamos los valores **1** y **20**, respectivamente. Junto a este TextBox hay otro que muestra cuál es nuestro cuadro actual.

Seleccionamos la cabeza y nos colocamos en el cuadro **1**. Nos dirigimos al panel **Transform** y, en la sección de rotación, escribimos el valor **-45.0** en el eje **Z**. Esto mueve la cabeza del robot. Cuando ya está todo en la posición deseada, grabamos ese cuadro como un **keyframe** (cuadro llave) que se usará para calcular la animación. Los cuadros llave son los cuadros de animación donde sucede una parte importante del movimiento, y se usan como base para crear la interpolación del movimiento en la animación.

Para grabar esa posición en el cuadro llave, procedemos a colocar el cursor sobre el área de edición y oprimir la tecla **I**; al hacerlo, aparece un menú con las opciones para grabar, entre las cuales encontramos: **Rotation**, **Location**, **Scaling**, **LocRot**, **RotScale**, **LocRotScale** y todas las demás combinaciones posibles. En esta ocasión seleccionamos **Rotation**. Si hubiéramos cambiado la posición y rotado, entonces la selección sería



► **Figura 12.** Es necesario indicar cuál de las propiedades o qué combinación es la que deseamos grabar.

EXISTEN DIFERENTES MECANISMOS PARA LLEVAR A CABO LA ANIMACIÓN DE LAS GEOMETRÍAS EN 3D



LocRot, para indicar que nos interesa guardar la locación y la rotación. Así queda guardado ese cuadro de animación. Nos movemos al cuadro **5** y colocamos otra vez el valor **-45.0** en el eje **Z** para la rotación; volvemos a grabar el cuadro de animación. Ahora vamos al cuadro **10** y ponemos como valor de rotación en **Z** el valor **0**. Para el cuadro **15** ponemos el valor **45.0** en el mismo campo y, por último, en el cuadro **20**, el valor es **0**. Todos estos valores crean el movimiento de la

cabeza, que es un poco asimétrico a propósito. Si deseamos observar cómo luce la animación, simplemente oprimimos el botón **PLAY**, ubicado en la misma barra que las posiciones de los cuadros de animación.

La siguiente animación será la que hace que el robot camine. En ella, el robot moverá la cabeza y las dos piernas. Cuando necesitamos animar más de un componente del personaje, lo que hacemos es grabar cada uno por separado; es decir, primero hacemos la grabación de la cabeza, luego de una pierna, y después de la otra. Hacemos estas animaciones de acuerdo con las siguientes tablas:

VALORES PARA LA CABEZA		
CUADRO	ROTACIÓN EN X	ROTACIÓN EN Z
21	0	0
30	30	30
40	0	30

Tabla1. Información de animación para la cabeza.



CAPACITACIÓN EN BLENDER



En el sitio web de Blender existe una sección especializada con links a muchas páginas y documentos que podemos utilizar para aprender más sobre este programa. También hay cursos en video y muchos libros que podemos consultar. Blender es un software muy utilizado, por lo que será sencillo encontrar diversos recursos en Internet. El sitio web oficial de recursos educativos y ayuda es: www.blender.org/education-help.

VALORES PARA LA PIERNA DERECHA	
CUADRO	ROTACIÓN EN X
21	0
25	-30
30	0
35	30
40	0

Tabla 2. Información de animación para la pierna derecha.

VALORES PARA LA PIERNA IZQUIERDA	
CUADRO	ROTACIÓN EN X
21	0
25	30
30	0
35	-30
40	0

Tabla 3. Información de animación para la pierna izquierda.

La última animación consiste en hacer que el robot dispare. En este caso, tendremos movimiento de la cabeza y del brazo. Este rotará y

VALORES PARA LA CABEZA		
CUADRO	ROTACIÓN EN X	ROTACIÓN EN Z
41	0	30
50	-30	30
60	-30	30

Tabla 4. Información de animación para la cabeza.

VALORES PARA EL BRAZO		
CUADRO	ROTACIÓN EN X	ESCALA EN Z
41	0	1
50	90	3
60	0	1

Tabla 5. Información de animación para el brazo.

cambiará de tamaño, por lo que cuando se graben los cuadros de este objeto, será necesario usar **RotScale**, ya que esta opción guarda tanto la rotación como la escala. La animación corre del cuadro **41** al **60**, y las siguientes tablas muestran los valores que debemos usar.

Ya tenemos las tres animaciones que necesitamos. Ahora es preciso exportar el modelo y las animaciones a Unity.

Exportar a Unity

Primero necesitamos crear un nuevo proyecto de Unity al cual llamaremos **Capítulo9**. Le colocamos un plano en las coordenadas **(0,0,0)**, con escala en **X** y **Z** con valor **10**, y la escala en **Y** con valor **1**. Este será el piso sobre el cual el robot se moverá. Luego, ponemos una luz de tipo **Directional Light** y la rotamos a gusto para iluminar bien la escena.

Una vez terminado el primer paso, importamos la textura que corresponde a la cara del robot en Unity, usando la técnica que hemos aprendido anteriormente. Si nuestro objeto tuviera más texturas, tendríamos que importarlas a todas.



LAS ESFERAS



Existen diferentes algoritmos para la creación de esferas, y Blender permite crear dos tipos de ellas: UV Sphere e Icosphere. La esfera UV está diseñada para aplicar texturas fácilmente, y su geometría es similar a la división de la Tierra en meridianos y paralelos. La icoesfera es diferente, pues su geometría se crea a partir de la subdivisión progresiva del icosaedro, lo que produce una geometría conocida como geodésica. Aunque es más sencillo aplicar texturas a la esfera UV, también podemos asignarlas a la icoesfera.

Regresamos a Blender y verificamos que no estemos en modo de edición de algún objeto. Luego oprimimos la tecla **A** para seleccionar todos los objetos que componen el robot; es muy importante que todos estén seleccionados. En el menú **File** elegimos **Export** y, de este modo, se mostrarán varias opciones, de las cuales elegimos **AutoDesk FBX (.fbx)**. Eso nos lleva a una nueva pantalla donde podemos seleccionar el nombre del archivo y la carpeta de destino. Por una cuestión de agilidad, podemos grabarlo en el escritorio y ponerle el nombre **Robot.fbx**. En el lado derecho, vemos un panel donde aparecen las propiedades para exportar el archivo; comprobamos que la opción **Selected Objects** esté marcada y que esté seleccionada también **Enable Animation**. Una vez que dichas opciones estén marcadas, hacemos clic en el botón **Export FBX**.

SI NUESTRO
OBJETO TUVIERA
MÁS TEXTURAS,
TENDRÍAMOS QUE
IMPORTARLAS A TODAS

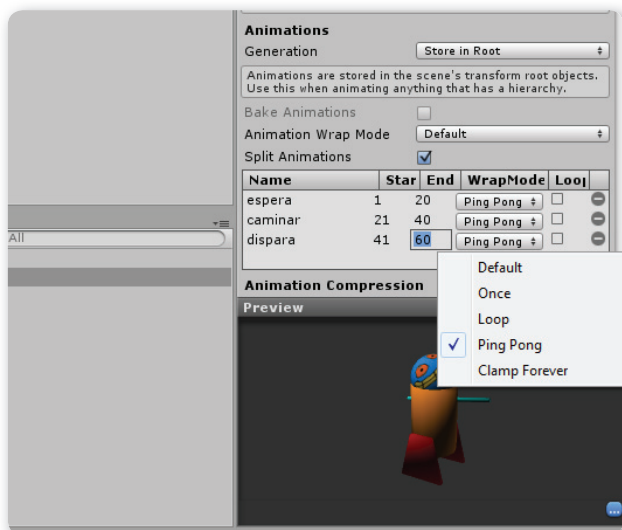


Este es un buen momento para guardar el trabajo en Blender, si no lo hemos realizado con anterioridad.

Para importar el robot, arrastramos el archivo **Robotito.fbx** desde el escritorio hacia el panel **Project** de Unity. Al soltarlo, Unity llevará a cabo la importación y lo mostrará con un icono en forma de cubo azul. Si lo seleccionamos, en el panel **Inspector** tendremos las propiedades de importación. Debemos colocar algunos valores para que el proceso se lleve a cabo correctamente. Lo primero que verificamos es la escala, que está en la propiedad **Scale Factor** y debe tener el valor **1**. En la sección de animaciones marcamos la propiedad **Split Animations** y completamos la tabla que allí se encuentra con la información de las animaciones creadas en Blender. Los datos se muestran en la **Tabla 6**.

VALORES PARA LA PIERNA IZQUIERDA			
NAME	START	END	WRAPMODE
espera	1	20	Ping Pong
caminar	21	40	Ping Pong
dispara	41	60	Ping Pong

Tabla 6. Información de animación para la pierna izquierda.



► **Figura 13.** Al importar el robot, es necesario indicar la información de la animación.

Por último, debemos instanciar el robot, arrastrándolo desde el panel **Project** hasta la escena. Una vez allí, podremos verlo.

Código de animación

Luego de importar el robot, podemos crear un script y asignárselo. Esto nos permitirá controlar programáticamente su animación. El control de la animación puede ser muy sencillo o muy complejo, ya que, en realidad, se pueden hacer controles de animación muy avanzados en Unity. Empecemos por algo simple. Creamos un script con nombre **ControlRobot** y le colocamos el siguiente código:

```
#pragma strict

function Start () {

}

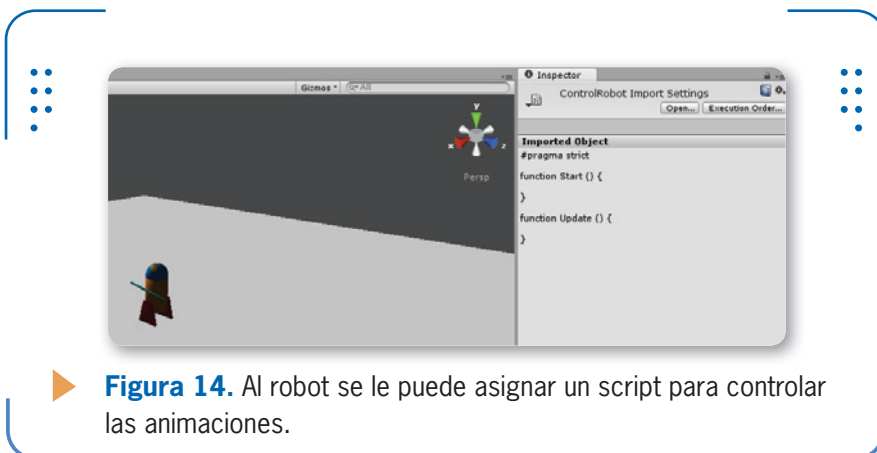
function Update () {
```

```
if(Input.GetKey(KeyCode.Alpha1))
{
    animation.Stop();
    animation.Play("espera");
}

if(Input.GetKey(KeyCode.Alpha2))
{
    animation.Stop();
    animation.Play("caminar");
}

if(Input.GetKey(KeyCode.Alpha3))
{
    animation.Stop();
    animation.Play("dispara");
}
}
```

Es necesario asignar el script al robot para que funcione. El control general de las animaciones se encuentra dentro del objeto **animation**, y este tiene varios métodos para controlarlas. El primero que usamos es **Stop()**, que detiene cualquier animación que se esté ejecutando. Otro



► **Figura 14.** Al robot se le puede asignar un script para controlar las animaciones.

método muy importante es **Play()**, que usamos cuando queremos que una animación en particular empiece a reproducirse. El método necesita un parámetro que es el nombre de la animación por mostrar. Debe estar escrito exactamente igual que en la tabla con los datos sobre la configuración de la importación. Si los nombres no coinciden, entonces la animación no se mostrará. Cabe notar que el nombre de la animación se pasa como cadena, de modo que no hay que olvidar las comillas.

En el script esperamos que se oprima la tecla **1**, **2** o **3** del teclado alfanumérico. Lo primero que sucede es que se detiene la animación anterior y, luego, empieza la que hemos seleccionado. Podemos ejecutar ahora la aplicación y experimentar con esto.

Es tiempo de hacer un script un poco más avanzado, por lo que creamos uno llamado **ControlRobot2** y le colocamos el siguiente código:

```
#pragma strict
var velocidad:float=1.0;
var velAngular:float=60.0;

function Start () {

}

function Update () {

    if(Input.GetKey(KeyCode.UpArrow))
    {
        transform.Translate(0,0,velocidad*Time.deltaTime);
        animation.Play("caminar");
    } else if(Input.GetKey(KeyCode.DownArrow))
    {
        transform.Translate(0,0,-velocidad*Time.deltaTime);
        animation.Play("caminar");
    } else if(Input.GetKey(KeyCode.LeftArrow))
    {
        transform.Rotate(0,-velAngular*Time.deltaTime,0);
        animation.Play("caminar");
    }
```

```
}else if(Input.GetKey(KeyCode.RightArrow))
{
    transform.Rotate(0,velAngular*Time.deltaTime,0);
    animation.Play("caminar");

}

}else if(Input.GetKey(KeyCode.Space))
{
    animation.Play("dispara");
} else
{
    animation.Play("espera");
}

}
```

Con este script hacemos algo más interesante, pues el robot tiene la animación de caminar cuando lo movemos por la escena. Si no se oprime ninguna tecla, se muestra la animación de espera; y si se oprime la **barra espaciadora**, la animación que se ejecuta es la de disparo. Este script, aunque sencillo, muestra la manera en que podemos controlar las animaciones dependiendo de las acciones que lleve a cabo el personaje. Podemos asignar el script al robot, pero antes es necesario deshabilitar el anterior.

Hemos aprendido las bases de la animación y la importación de geometrías y objetos a Unity desde Blender. Ya podemos realizar proyectos que luzcan mucho mejor y sean más entretenidos.



RESUMEN



En Unity es posible importar geometrías y animaciones creadas con software de modelaje 3D. Un software muy utilizado es Blender, el cual permite crear objetos de manera sencilla. Tenemos diversas herramientas de edición y, también, es posible asignarles materiales a los objetos. La asignación de texturas es sencilla con la herramienta **Unwrap**. La animación se crea usando keyframes, que muestran los momentos importantes de la animación. El formato más sencillo para exportar es **FBX**, que Unity lee directamente. Al importar, es necesario colocar en Unity la información de las animaciones. El objeto **animation** contiene los métodos necesarios para controlar la animación.

Actividades

TEST DE AUTOEVALUACIÓN

- 1 ¿De qué forma se eliminan los objetos en Blender?
- 2 ¿Cómo se adiciona un objeto en Blender?
- 3 ¿Cuáles son los tipos de esferas en Blender?
- 4 ¿Cuáles son los modos en los que se puede trabajar con un objeto?
- 5 ¿Cómo se seleccionan o deseleccionan los objetos en Blender?
- 6 ¿De qué manera se duplica un objeto?
- 7 ¿Cómo se asignan los materiales?
- 8 ¿Cómo se asignan las texturas?
- 9 ¿Cuál es la función de **Unwrap**?
- 10 ¿Cuáles son los pasos para animar en Blender?
- 11 ¿Qué formato de archivo se usa para exportar a Unity?
- 12 ¿Cuáles son los pasos para importar en Unity?

ACTIVIDADES PRÁCTICAS

- 1 Diseñe, modele, anime e importe otro personaje.
- 2 Adicione un sistema de partículas al ataque del robot.
- 3 Cree la animación de correr para el robot.
- 4 Cree la animación de saltar para el robot.
- 5 Agregue un personaje que se controle de manera independiente.



Código y herramientas útiles

Unity se caracteriza por proveernos de muchas herramientas útiles y fáciles de usar para crear videojuegos. Si bien ya hemos visto las principales, todavía nos falta conocer algunas más y los ejemplos de código que podemos utilizar en nuestros proyectos, con el objetivo de hacer scripts más avanzados y de manera más sencilla. En este capítulo, nos concentraremos en estos temas.

▼ Elementos y scripts

fundamentales250

Sonido 250

Herramientas de depuración..... 256

Esconder objetos..... 259

Distancia entre los objetos 261

Manejo de prefabs..... 263

El uso de tag..... 265

▼ Resumen.....269

▼ Actividades.....270





Elementos y scripts fundamentales

Agregar algunos elementos clave incorpora dinamismo y complejidad a nuestros videojuegos. Este es el caso de la adición de sonidos al juego, que aumenta enormemente el realismo. Además, es importante conocer los códigos indispensables que podemos incorporar en nuestros proyectos con el objetivo de lograr distintas funcionalidades de manera simple.

Sonido

El sonido en los videojuegos es un elemento importante que nos ayuda enormemente a generar realismo, y a hacer que la experiencia del juego sea mucho más interesante y entretenida. Con Unity, podemos incorporar sonidos a los diferentes objetos de la escena. Este puede reproducirse constantemente o podemos hacer que suene cuando sucede cierta condición en particular. Para trabajar con el sonido en Unity, debemos entender tres conceptos importantes: la **importación**, el **Listener** y el **Audio Source**.

La importación de sonido no es muy diferente de importar otro tipo de elemento. Primero, necesitamos tener el archivo de sonido guardado en algún lugar del disco duro. El escritorio es un buen sitio, ya que ofrece un acceso sencillo para la importación. Teniendo ya un proyecto creado, arrastramos el archivo de audio al panel **Project**. Cuando llevamos a cabo esta acción, aparece un nuevo objeto, que será nuestro recurso de audio, el cual podemos identificar fácilmente por su icono en forma de bocina. Repetimos el procedimiento para cuantos recursos



LA CLASE PREFABUTILITY



Existe una clase en Unity que nos brinda soporte para realizar muchas operaciones con los prefabs: se llama **PrefabUtility** y, con ella, podemos lograr resultados muy interesantes. En la documentación de Unity, encontramos el listado de sus funciones. Algunas son avanzadas, como la que permite desconectar la instancia de su relación con el prefab, para poder modificarla sin afectar a las demás instancias existentes.

sean necesarios. Los recursos de audio deben estar en cualquier formato usual de sonido, como **MP3** o **WAV**.

El siguiente elemento es el **Listener**, que representa a la persona que escucha lo que sucede y que, en la mayoría de los casos, suele ser el personaje principal. Este elemento es necesario porque los sonidos suenan diferente dependiendo de su posición con relación a la fuente que los emite. Unity nos permite reproducir sonido en 3D, que no sonará igual si la fuente se encuentra a la derecha, a la izquierda o en cualquier otra posición. El volumen también depende de la distancia entre el **Listener** y la fuente que lo genera. Casi siempre, tenemos solo un **Listener** por escena, que es colocado en la cámara del personaje principal. La cámara que Unity ha creado por nosotros ya contiene un objeto de este tipo.

Los objetos que deseamos que lleven a cabo la emisión de sonido deben tener un componente conocido como **Audio Source**. En él se coloca el clip de audio que queremos y, basado en sus parámetros de configuración, llevará a cabo la reproducción correspondiente. El clip puede ser fijo o podemos cambiarlo programáticamente, lo cual es más recomendable porque nos brinda mucha más flexibilidad.

Veamos cómo colocar sonido en la aplicación. Empecemos por crear un nuevo proyecto, llamado **Capítulo10**, y salvemos la escena como **Ejemplos**. De modo similar al aprendido en capítulos anteriores, creamos un plano ubicado en **(0,0,0)** y adicionamos una luz de tipo **Directional Light**, que puede tener la orientación que más nos guste. El primer objeto que vamos a poner es un cubo, para el cual creamos un material de color rojo al que llamamos **MatRojo**, ubicado en la posición **(0,1,0)**.

EL SONIDO EN LOS
VIDEOJUEGOS ES
UN ELEMENTO
IMPORTANTE QUE
GENERA REALISMO

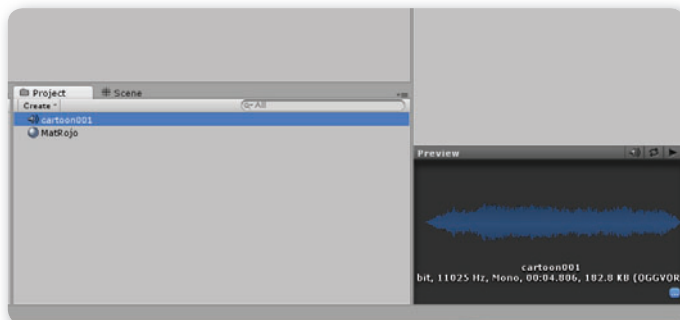


LA CLASE RIGIDBODY



Otra clase que podemos utilizar para tener más control sobre la aplicación es **RigidBody**. Es importante conocerla, porque tanto sus variables como sus funciones son interesantes y nos permiten tener más control sobre el comportamiento físico del objeto. Por ejemplo, podemos agregar fuerzas y torque o modificar la velocidad. Incluso, es posible congelar la rotación o modificar el centro de masa y la densidad.

Para importar el recurso de audio, procedemos seleccionándolo del panel **Project**. Unity nos permite reproducirlo para verificar su calidad y contenido, además de darnos la posibilidad de observar algunas de sus propiedades básicas en el panel **Inspector**.



► **Figura 1.** La importación del recurso de audio es muy sencilla y aparece en el panel **Project**.

A continuación, seleccionamos el cubo. Le vamos a agregar un componente de tipo **Audio Source** para que pueda reproducir un sonido cuando lo necesitemos, basado en la lógica del script. Dicha adición se lleva a cabo por medio del menú **Component**, del cual seleccionamos **Audio** y, en la lista de opciones, **Audio Source**. De este modo, el cubo ya tiene el componente y, dentro de este, encontramos diversas propiedades.

En este ejemplo, habrá un segundo cubo que colisionará con el que ya tenemos. Para reconocerlos, cambiamos el nombre del actual a **CubePrincipal** y añadimos uno nuevo en la posición **(0,5,0)**, al cual le podemos asignar el mismo material. Al nuevo cubo le colocamos un



FUNCIONES SOBRE EL AUDIO



Podemos utilizar diversas funciones que modifican la forma en que se reproduce el audio. **Volume** se usa para indicar la amplitud con relación a la distancia. **Pan** indica cuán a la derecha o a la izquierda se escucha el sonido con relación a la distancia. **Spread** se refiere a la dispersión del audio en grados, y **Low-Pass** es un filtro usado en frecuencia contra distancia.

componente de tipo **RigidBody** para hacer que, debido a la gravedad, este caiga y colisione con el principal.

Adicionamos un nuevo recurso de audio que se reproducirá cuando se presione la **Barra espaciadora**. Con esto, demostraremos la capacidad de usar un **AudioSource** para más de un sonido. Creamos ahora un script llamado **ControlAudio** y le colocamos el siguiente código:

```
#pragma strict

// Representan al audio que deseamos reproducir
var efecto: AudioClip;
var efecto2: AudioClip;

function Start () {

}

function Update () {

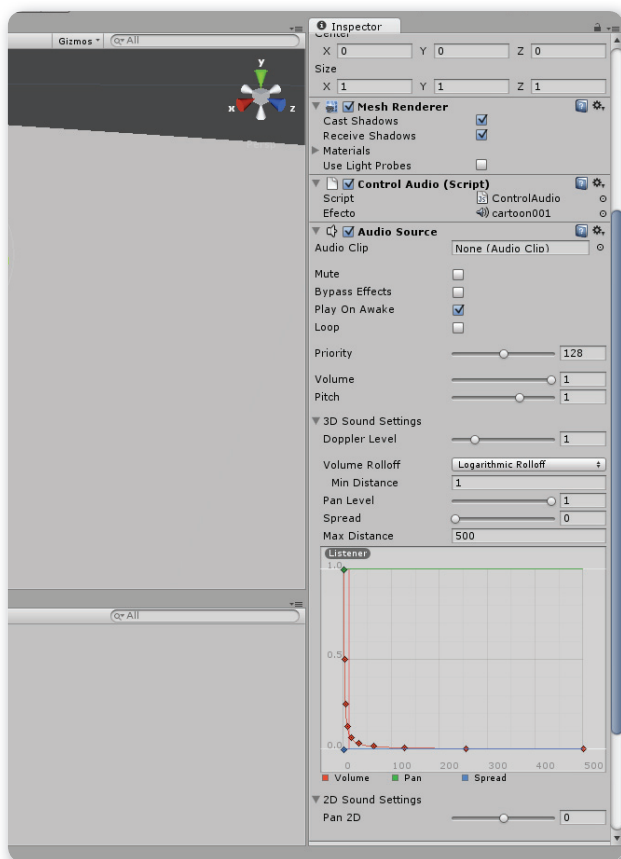
    if(Input.GetKeyDown(KeyCode.Space))
    {
        // Verificamos que el AudioClip tenga un objeto asignado
        if(efecto2)
        {
            // Asignamos efecto como audio a reproducir
            audio.clip=efecto2;

            // Reproducimos el audio
            audio.Play();
        }
    }

}

function OnCollisionEnter(collision : Collision)
{
    // Verificamos que el AudioClip tenga un objeto asignado
    if(efecto)
```

```
{  
  
    // Asignamos efecto como audio a reproducir  
    audio.clip=efecto;  
  
    // Reproducimos el audio  
    audio.Play();  
  
}  
  
}
```



► **Figura 2.** El componente **AudioSource** debe ser colocado para poder reproducir el sonido.

El programa inicia declarando dos propiedades de tipo **AudioClip**, las cuales nos permitirán asignar los recursos de audio y que estos sean reproducidos por el script. Dentro del método **Update()** verificamos si se ha presionado la tecla **Barra espaciadora** por medio de la función **GetKey()**; en caso de que así suceda, procedemos a usar un **if** para verificar si se ha asignado un recurso de audio a **efecto2**. Si tratamos de reproducir un **AudioClip** vacío, podemos recibir errores. El **Audio Source** es referenciado vía el objeto audio, por lo que en su propiedad de clip asignamos **efecto2**. Para reproducir el recurso de audio, simplemente invocamos el método **Play()** de audio.

Cuando ocurra la colisión, el recurso de audio **efecto2** se reproducirá, por lo que hacemos uso del handler **OnCollisionEnter()**. En su interior colocamos un código similar, primero verificando mediante el **if** si el efecto contiene un recurso de audio, para posteriormente asignar el recurso al clip y reproducirlo por medio de **Play()**.

Una vez que este código ha sido salvado, lo asignamos al objeto **CubePrincipal**. No hay que olvidar que, dentro del panel **Inspector**, debemos colocar los recursos de audio en las propiedades que corresponden al script. Al ejecutar la escena, escucharemos los sonidos que son reproducidos por el cubo.

Si el volumen es demasiado bajo, debemos modificar la propiedad **Volume Rolloff**, que indica la manera en que se desvanece el sonido con relación a la distancia. Aquí seleccionamos el valor **Custom Rolloff** y, en la gráfica ubicada en la parte inferior de la pantalla, editamos la curva que muestra cómo desciende el volumen.

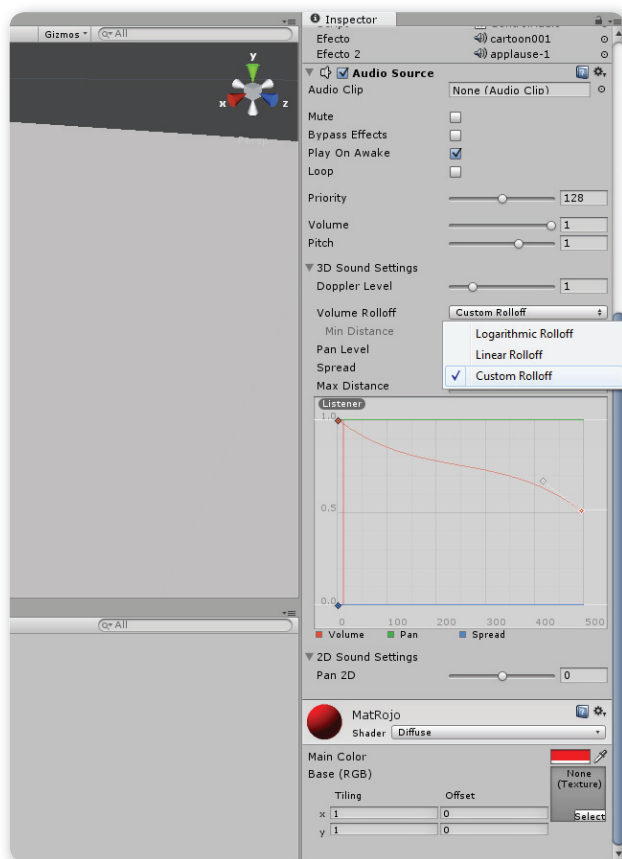
SI EL VOLUMEN
ES DEMASIADO BAJO,
DEBEMOS MODIFICAR
LA PROPIEDAD
VOLUME ROLLOFF



PARTÍCULAS EN PREFABS



Cuando necesitamos crear efectos en varias partes de la escena o hacer que el efecto esté relacionado con un objeto determinado, lo mejor es crear un prefab con el sistema de partículas. La forma de conseguirlo es igual que con un objeto: debemos crear el sistema, luego un prefab y colocarlo allí. Lo mismo que hemos aprendido para instanciar el prefab se utiliza en este caso, y el sistema aparecerá donde sea necesario, con cuantas copias requiera el diseño del juego.



► **Figura 3.** Es posible modificar la manera en que se comporta el volumen dentro de la aplicación.

Herramientas de depuración

A veces necesitamos algunas herramientas que nos ayuden a depurar la aplicación, en especial, cuando trabajamos con elementos que no se observan fácilmente, como **Bounding Box**. En esta ocasión, veremos cómo es posible utilizar la clase **Gizmos** para mostrar elementos en el panel **Scene** y que estos nos ayuden a crear la aplicación. El código que vamos a utilizar se ejecuta cuando estamos editando y no cuando la aplicación se corre. Empecemos por crear un nuevo script llamado **Debug** al que le colocamos el siguiente código:

```
#pragma strict

var objetivo : Transform;

function OnDrawGizmosSelected () {

    // Verificamos que el objeto ha sido asignado
    if(objetivo) {

        // Indicamos el color que deseamos
        Gizmos.color = Color.blue;

        // Indicamos las posiciones para trazar la linea
        Gizmos.DrawLine (transform.position, objetivo.position);

        // Indicamos el color para la siguiente figura
        Gizmos.color = Color.white;

        // Dibujamos la esfera de radio 1.5
        Gizmos.DrawWireSphere (transform.position, 1.5);

    }

}
```

Para probar este script, necesitamos un objeto en la escena, que pasamos por medio de una propiedad llamada **objetivo**. Para pasarlo como parámetro, en el editor arrastramos el objeto a la propiedad correspondiente del script dentro del panel **Inspector**. Hacemos uso de una función especial denominada **OnDrawGizmosSelected()**, que permite el dibujo de gizmos, los cuales usamos para depurar más fácilmente la aplicación. Se mostrarán en el panel **Scene**. El código de **OnDrawGizmosSelected()** no requiere que se ejecute el juego para que se procese. Todos los gizmos se apoyan en la clase **Gizmos**.

En este ejemplo, haremos uso de dos gizmos. El primero permite dibujar una línea entre dos posiciones. Es muy útil, por ejemplo, si deseamos hacer algún tipo de **ray casting**, para verificar hacia qué objeto se está dirigiendo el personaje, etc. **Gizmos** tiene una propiedad llamada **color**, que se utiliza para indicar el color con el que se va a dibujar. Todos se dibujan con el color indicado hasta que se vuelva a

asignar un nuevo color a la propiedad; en el programa vamos a usar azul. Para dibujar la línea podemos recurrir a la función **DrawLine()**, que requiere de dos posiciones. La primera que colocamos es la que representa nuestra propia ubicación, y la segunda es la del objeto llamado objetivo. También es posible colocar posiciones explícitamente pasando un objeto de tipo **Vector3**.

Ahora agregaremos un código para otro gizmo muy útil, que nos permite dibujar una esfera. Si algunas veces necesitamos depurar el **bounding sphere** de un objeto para las colisiones, una manera simple de hacerlo es por medio de este gizmo. Al dibujar una esfera con el mismo radio que el **bounding sphere**, podemos observar fácilmente cuándo se lleva a cabo o no la colisión. Para ayudar a diferenciar el gizmo, colocamos el color en blanco y, después, usamos la función **DrawWireSphere()**, que precisa de dos parámetros. El primero de ellos es la posición donde se encuentra el centro de la esfera; es bueno ponerla

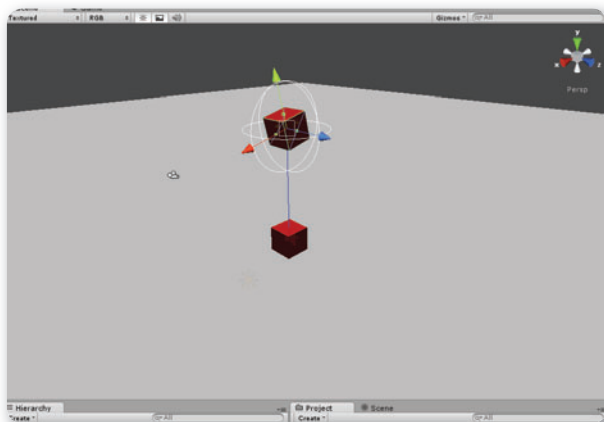


Figura 4.

Los gizmos se han dibujado y nos ayudan a depurar la aplicación.



PROPIEDADES DE AUDIO SOURCE

La propiedad **Audio Clip** representa al recurso de audio que deseamos que se reproduzca. Este puede ser pasado directamente desde el editor como una propiedad si deseamos relacionar el **Audio Source** solo a ese sonido. Si queremos que reproduzca cualquier sonido posible, entonces es mejor hacerlo programáticamente. Las demás propiedades nos permiten controlar la manera en que ese sonido se reproduce.

en la posición del objeto que deseamos englobar con ella. El segundo parámetro es el radio de la esfera. De esta forma, podríamos colocar más gizmos, ya sea para el objeto del script o para algún otro recibido por medio de la propiedad.

Asignamos el script al segundo cubo y colocamos **CubePrincipal** en la propiedad **Objetivo**, haciendo uso del panel **Inspector**. En la escena, veremos que se dibuja una línea de color azul entre los dos objetos y una esfera alrededor del cubo.

Esconder objetos

Otro código que puede resultarnos útil es el que sirve para esconder los objetos. En este caso, el objeto continuará ejecutando toda su lógica, pero no se llevará a cabo el dibujo de su geometría. Podemos utilizarlo para lograr diferentes efectos dentro del juego. El objeto puede empezar a dibujarse otra vez en el momento que lo indiquemos. Solo la parte del render se ve afectada; todo lo demás, incluidas las colisiones, se llevará a cabo. Creamos un nuevo script denominado **Esconder** y le asignamos el siguiente código:

```
#pragma strict

function Start () {

}

function Update () {

    // Obtenemos el objeto Renderer para trabajar con el
    var r:Renderer = GetComponent(Renderer);

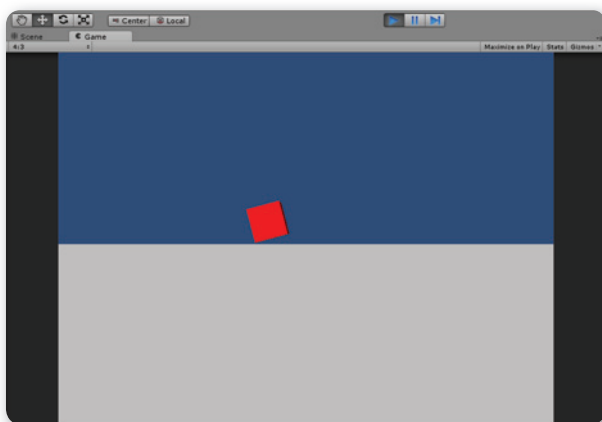
    if(Input.GetKeyDown(KeyCode.Alpha1))
    {
        // Deshabilitamos el render del objeto
        r.enabled = false;
    }

    if(Input.GetKeyDown(KeyCode.Alpha2))
```

```
{  
  
    // Habilitamos el render del objeto  
    r.enabled = true;  
  
}  
  
}
```

El código de esta función puede parecer complejo, pero en realidad es fácil de entender. Existe un componente que tienen los objetos y se llama **Renderer**, que es el encargado de llevar a cabo las acciones de dibujo del objeto. Cuando deseamos modificar los elementos relacionados con el dibujo, hacemos uso de él. El componente no puede ser llamado directamente, por lo que es necesario obtenerlo. Para obtener cualquier componente, utilizamos **GetComponent()**. Esta función requiere de un parámetro que es el componente por buscar del objeto donde se ejecuta el script. La función regresa un objeto que es la referencia al componente, y debemos tener una variable que lo reciba. Por esta razón, creamos la variable **r**, que es de tipo **Renderer**, y si todo funciona bien, seremos capaces de modificar las propiedades de **Renderer** por medio de **r**.

En el código vamos a ocultar el cubo y a mostrarlo otra vez, dependiendo de la tecla que el usuario oprima. Cuando presiona **1** del teclado alfanumérico, colocamos el valor **false** en la propiedad **enabled** de **Renderer**, para hacer que el cubo ya no se dibuje. Como comentamos



► **Figura 5.**
El cubo no se dibuja, pero continúa interactuando con los demás objetos.

anteriormente, esto solo modifica la parte de dibujo, pero todos los demás elementos del cubo siguen funcionando. Si colocamos el valor **true** en la misma propiedad, el cubo se vuelve a mostrar, y esto sucede cuando se oprime el **2** del teclado alfanumérico.

Ahora podemos asignar el script a **CubePrincipal** y ejecutarlo. Tras hacerlo, observaremos que el cubo se oculta y aparece otra vez. Incluso, cuando está oculto, la colisión con el otro cubo se realiza.

Distancia entre los objetos

Un factor que resulta sumamente útil es conocer la distancia que existe entre los objetos, o entre ellos y el jugador. Mucha de la lógica del programa puede depender de este tipo de cálculos. El cálculo de la distancia puede usarse para llevar a cabo transiciones en máquinas de estados finitos. Este es un mecanismo sencillo de control, y de esta manera podremos modificar el comportamiento de los enemigos, reconocer si el objeto o jugador está dentro de algún área, simular sensores o alguna otra función que necesitemos que se ejecute.

El cálculo de la distancia se realiza usando la fórmula de distancia entre dos puntos, aunque hay un método para hacerlo que nos ofrece Unity que reduce la cantidad de código por escribir. Veamos un ejemplo de cómo utilizarlo creando un nuevo script que lleve por nombre **ObtenerDistancia**. En el script colocamos el siguiente código:

```
#pragma strict

// Propiedad para el objeto del que deseamos conocer la distancia
var objetivo: Transform;

// Variable para guardar la distancia
private var distancia: float;

function Start () {

}

function Update () {
```

```
// Calculamos la distancia dando las dos posiciones
distancia = Vector3.Distance(objetivo.transform.position,
transform.position);

// Verificamos el valor de la distancia
if( distancia < 1.0)
{
    // Mandamos el objetivo hacia arriba
    objetivo.transform.Translate(0,5,0);
}

}
```

Veamos el código. En primer lugar, encontramos una propiedad de tipo **Transform**, por medio de la cual recibimos el objeto con el que deseamos calcular la distancia. La distancia será calculada entre el objeto que tiene el script y el que pasamos vía la propiedad.

Para calcular la distancia, en vez de colocar mucho código, simplemente invocamos la función **Distance()**, perteneciente a la clase **Vector3**. La función necesita dos parámetros de tipo **Vector3**, que representan las coordenadas o puntos entre los que deseamos encontrar la distancia. La función regresa un valor de tipo **float** que es la distancia calculada en unidades de mundo. En el programa, simplemente, verificamos la distancia, y, si esta es menor que **1.0**, movemos el objeto contenido en **objetivo 5** unidades hacia arriba. El ejemplo da la idea general de cómo podemos hacer que los objetos reaccionen a otros a la distancia. Asignamos el script al objeto **CubePrincipal** y ponemos **cube** en su propiedad **objetivo**. Ahora podemos ejecutar el programa y observar su comportamiento.



LOS GIZMOS



Existen más gizmos que podemos utilizar en nuestras aplicaciones, y que se explican en detalle en la documentación de Unity. Por ejemplo, tenemos **DrawSphere**, que dibuja una esfera sólida; **DrawWireCube**, que coloca un cubo de alambre; y **DrawCube**, que lo dibuja sólido. Todos deben usarse en la función **OnDrawGizmosSelected()**.

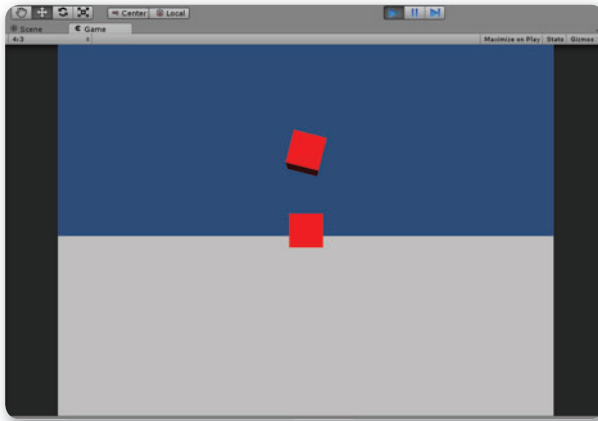


Figura 6.

El cubo se coloca arriba otra vez cuando la distancia es menor que **1.0**.

Manejo de prefabs

El concepto de prefabs es de gran utilidad en Unity, y aunque ya los hemos comentado brevemente en capítulos anteriores, lo veremos más a fondo en este. El prefab está diseñado para ahorrar trabajo, ya que nos permite crear un modelo de objeto y, luego, hacer fácilmente muchas instancias de él. Pensemos, por ejemplo, en un juego donde tengamos varios enemigos que lucen iguales y que usan los mismos scripts. En vez de tener que crear en forma manual cada uno de los enemigos, lo que hacemos es crear un prefab y, luego, crear tantos enemigos como sean necesarios. La reducción de trabajo y el ahorro de tiempo son muy grandes. Cuando trabajamos con prefabs, debemos planificar correctamente todo lo que queremos que haga, ya que realizar cambios en uno de ellos una vez que lo hemos creado es engorroso. Por lo general, lo que hacemos es crear primero la geometría y, luego, colocar las animaciones y scripts necesarios. Hay que probar el funcionamiento y, una vez seguros de que tendrá todo lo requerido, se procede a su creación.

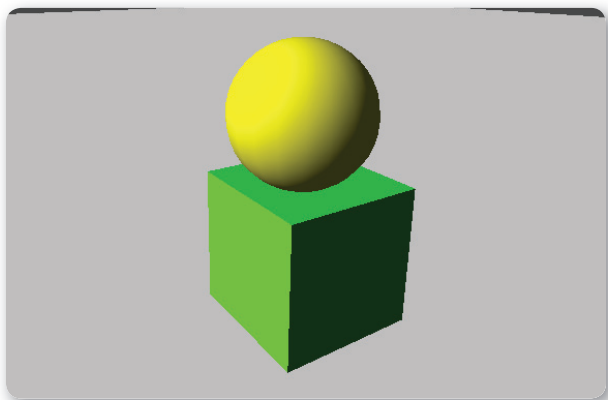
UN PREFAB PERMITE
CREAR UN MODELO DE
OBJETO Y LUEGO HACER
FÁCILMENTE MUCHAS
INSTANCIAS DE ÉL

Vamos a construir un prefab, que consistirá en un cubo y una esfera. Luego, trabajaremos con el cubo y sus instancias. Empezamos por generar dos materiales nuevos que usaremos en la geometría. El primer material es verde y se llama **MatVerde**; al otro le

colocamos color amarillo y lo nombramos **MatAmarillo**. Luego, creamos un cubo. En este momento, no importa el lugar donde lo hagamos; simplemente, es necesario poder manipularlo con facilidad. A este cubo le ponemos como nombre **CubeObjeto** y le asignamos el material **MatVerde**. Después, creamos una esfera y la colocamos exactamente arriba del cubo, sin olvidarnos de asignarle el material **matAmarillo**.

Ya que deseamos manejar un único objeto en vez de dos, realizamos un parenting, tal como lo vimos en el **Capítulo 2**. Usando el panel **Hierarchy**, colocamos la esfera como hijo de **CubeObjeto**. Al hacer el parenting, simplificamos el proceso y podemos hacer el prefab, aunque contenga diferentes objetos. Una vez que finalizamos, le adicionamos a **CubeObjeto** un componente **RigidBody**.

Procedemos a crear el prefab, el cual en un principio está vacío, y al que posteriormente le agregaremos todos los elementos que contendrá. Para esto, vamos al panel **Project** y hacemos clic derecho sobre su menú.



► **Figura 7.**
El objeto con su parenting es utilizado para crear el prefab.

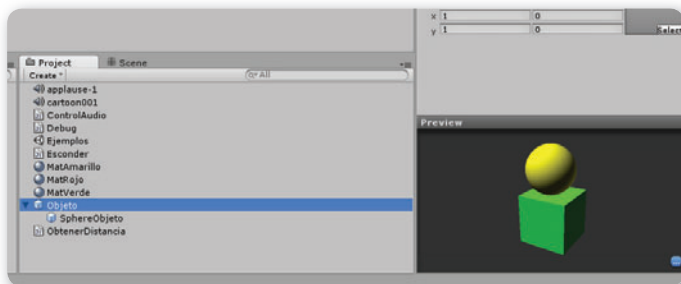


EL QUATERNION



El **quaternion** es un número hipercomplejo. En este punto no necesitamos conocer su matemática, pero sí, sus aplicaciones. Es muy utilizado para representar rotaciones, pues no tiene problemas para hacerlo aunque sean complejas. Otra ventaja es que resulta muy sencillo hacer interpolaciones con él. Unity lo usa internamente como medio para representar todas las rotaciones. Es bueno conocer las funciones que nos ofrece la clase **Quaternion**, y podemos hacerlo mediante la documentación de Unity.

Aparecen una serie de opciones entre las que seleccionamos **Prefab**. Veremos que en el mismo panel se crea un nuevo elemento con el icono de un cubo; lo llamamos **Objeto**. Para adicionar la geometría al prefab, simplemente arrastramos **CubeObjeto** desde el panel **Hierarchy** hasta el panel **Project**, en el icono del prefab. Desde ahora, cada vez que creamos un objeto desde este prefab, tendrá la misma geometría que hemos usado, por lo que será aún más fácil crear múltiples objetos.

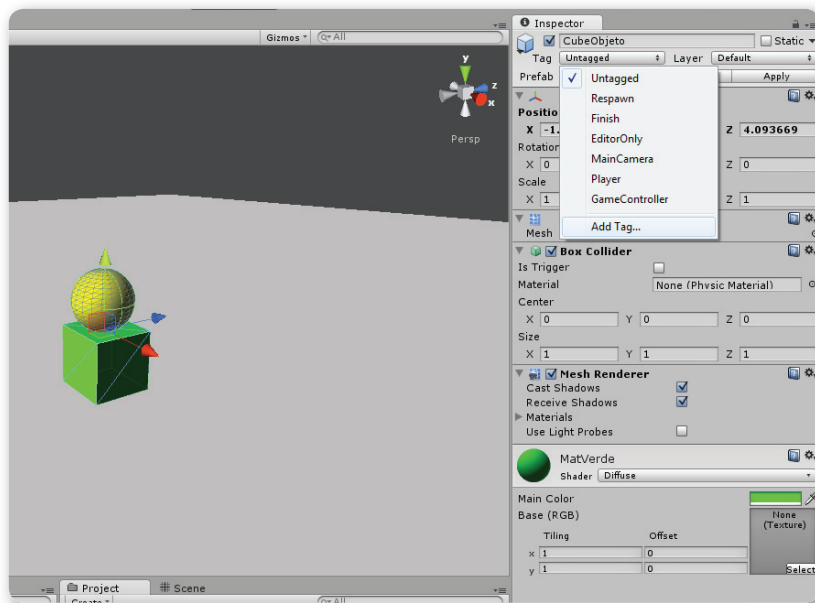


► **Figura 8.** En el prefab, podemos observar la geometría que colocamos.

El uso de tag

Existe un elemento en Unity llamado **tag** (etiqueta) que se usa para clasificar los objetos, y nos ayuda mucho a mejorar y simplificar la lógica del programa. Por ejemplo, podemos usar un tag para todos los elementos del juego que sean armas o municiones. De esta forma, por medio de un script, en vez de tener muchos **if** que realicen lo mismo para estos objetos diferentes, tendremos un único **if** para todos los objetos que tengan el mismo tag. Esto es muy útil, en especial, cuando tenemos una gran variedad de objetos. Los tags también nos ayudan a ordenar los objetos de manera lógica.

Para colocar un tag, primero debemos crearlo. Seleccionamos **CubeObjeto** y, en la parte superior del panel **Inspector**, encontramos una propiedad llamada **Tag**, la cual, por defecto, tiene el valor **Untagged**. Al hacer clic en dicha propiedad, aparece una lista de los tags existentes y, sobre el final, la opción de agregar un nuevo tag por medio de **Add Tag**. Para crearla, hacemos clic sobre esa opción.



► **Figura 9.** Es sencillo colocarle el tag al objeto o crear uno nuevo si es necesario.

PODEMOS USAR EL ADMINISTRADOR DE TAGS PARA CREARLOS, MODIFICARLOS O ELIMINARLOS

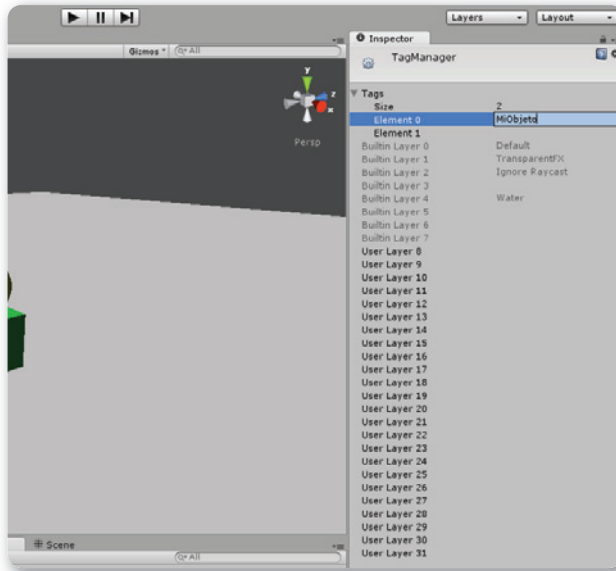


como tag. Al asignar el tag a un prefab, todos los objetos que se creen a partir de él tendrán el mismo tag.

El panel **Inspector** cambia y nos muestra el administrador de tags, que usamos para crearlos, modificarlos o eliminarlos. También nos permite administrar las diferentes capas del juego. En la parte superior aparecen los tags; en caso de que no se muestren, hacemos clic en donde dice

Tag. Como hasta este momento no hemos creado ninguno, la lista solo muestra un elemento donde podemos hacer la adición. Colocamos en este elemento **MiObjeto**, que es el texto que identifica a ese tag en particular. De manera similar podemos adicionar más tags.

Necesitamos seleccionar otra vez el prefab **Objeto**, de modo que el panel **Inspector** muestre todas las propiedades. Nos dirigimos nuevamente a la parte superior, donde debe aparecer **MiObjeto** en el listado; lo seleccionamos y queda asignado



► **Figura 10.** Creamos un nuevo tag haciendo uso del administrador de tags.

Creamos un nuevo script al que llamamos **CrearPrefabs** y le colocamos el siguiente código:

```
#pragma strict

// Propiedad para el prefab que deseamos utilizar
var objetoPrefab: Rigidbody;

function Start () {

}

function Update () {

    // Al oprimir Q creamos un objeto
    if(Input.GetKeyDown(KeyCode.Q))
    {

        // Se instancia el objeto en una posicion aleatoria
        Instantiate(objetoPrefab,Vector3(Random.Range(-10,10), 1, ►
```

```
        Random.Range(-10,10)), Quaternion.identity);
    }

    if(Input.GetKeyDown(KeyCode.W))
    {
        // Variable para la instancia
        var temporal : Rigidbody;

        // Creamos el objeto y lo guardamos en temporal
        temporal =Instantiate(objetoPrefab,Vector3(Random.
        Range(-10,10), 1, Random.Range(-10,10)), Quaternion.identity);

        // Usamos temporal para modificar las propiedades del objeto
        temporal.velocity = transform.TransformDirection (Vector3.
        forward * 10);
    }
}
```

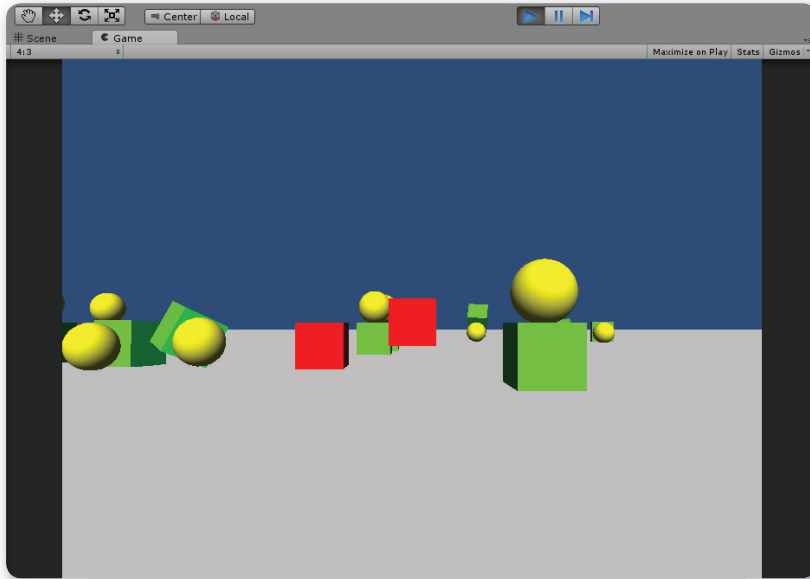
En primer lugar, encontramos una propiedad que guarda un **RigidBody**, por medio de la cual pasaremos el prefab que deseamos instanciar. El código instancia objetos basados en el prefab. Cuando se da la condición de que la tecla **Q** es oprimida, se instancia en un lugar seleccionado aleatoriamente; y cuando se presiona la **W**, se instancia y se le asigna una fuerza al objeto creado.

INSTANTIATE(),
REGRESA UN OBJETO
DE TIPO RIGIDBODY QUE
ES UNA REFERENCIA A
LA INSTANCIA

Para instanciar, hacemos uso de la función **Instantiate()**. El primer parámetro es el **RigidBody** del objeto por instanciar; el segundo es la posición donde se va a instanciar, que en este caso se da con valores aleatorios; y el último es la orientación del objeto, que se da por medio de un quaternion. Usamos la identidad para que el objeto no tenga ninguna rotación al crearse.

La función **Instantiate()** regresa un objeto de tipo **RigidBody**, el cual es una referencia a la instancia. Al tener esta referencia, podemos controlar el objeto creado por **Instantiate()**. Esto lo aprovechamos en la segunda instanciación, porque luego, por medio de la variable temporal, adicionamos una velocidad. Hay que notar que **velocity** es una propiedad de **RigidBody**.

Ahora ya podemos asignar el script a **CubePrincipal** y colocar el prefab en la propiedad **ObjetoPrefab** del script. Al ejecutar el programa, podemos crear muchos objetos.



► **Figura 11.** Con el script, podemos instanciar muchos objetos del prefab.



RESUMEN



Para utilizar el sonido, necesitamos importarlo, colocar un **Listener** y agregar el componente **Audio Source** a los objetos que van a producirlo. Por medio del objeto audio y el método **Play()**, podemos reproducir el sonido, sin olvidarnos antes de asignar el recurso de audio a la propiedad **clip**. El uso de gizmos nos ayuda a depurar los programas: los gizmos se mostrarán en el panel **Scene** aun cuando la aplicación no se ejecute. El componente **Renderer** permite acceder a diversos métodos y propiedades que nos facilitan el control de la forma en que se dibuja el objeto. A través de la clase **Vector3** y su método **Distance()**, calculamos la distancia entre dos objetos. Los prefabs nos permiten crear objetos de la misma clase, de manera muy sencilla y rápida.

Actividades

TEST DE AUTOEVALUACIÓN

- 1 Explique la adición de recursos de audio en Unity
- 2 ¿Qué es el componente **Listener**?
- 3 ¿Qué es el componente **Audio Source**?
- 4 ¿De qué tipo son las propiedades para recursos de audio?
- 5 ¿Con qué método se reproduce el sonido?
- 6 ¿Qué es un gizmo?
- 7 ¿Qué método se usa para trabajar con los gizmos?
- 8 ¿Qué tipos de gizmos hay?
- 9 ¿Para que se usa el componente **Renderer**?
- 10 ¿De qué forma se esconde o muestra un objeto?
- 11 ¿Cómo se calcula la distancia entre dos objetos?
- 12 ¿Qué son los prefabs?

ACTIVIDADES PRÁCTICAS

- 1 Cree varios objetos que produzcan sonidos.
- 2 Investigue cómo modificar el volumen programáticamente.
- 3 Piense la forma de colocar música de fondo en la aplicación.
- 4 Coloque un gizmo de tipo **Icon** en un objeto vacío.
- 5 Haga un script que instancie dos prefabs diferentes.

Interfaz de usuario

Durante el transcurso de este capítulo, nos concentraremos en los aspectos relacionados con la interfaz de usuario, en lugar de enfocarnos en los objetos que componen el juego. Aprenderemos a utilizar los elementos que Unity provee, para crear una interfaz amigable e intuitiva de manera simple y eficaz.

▼ Elementos para crear

la interfaz272

Label272

El uso de Box.....275

El uso de Button277

Pedido de texto al usuario.....280

El uso del slider284

El uso de toggle287

▼ Resumen.....289

▼ Actividades.....290

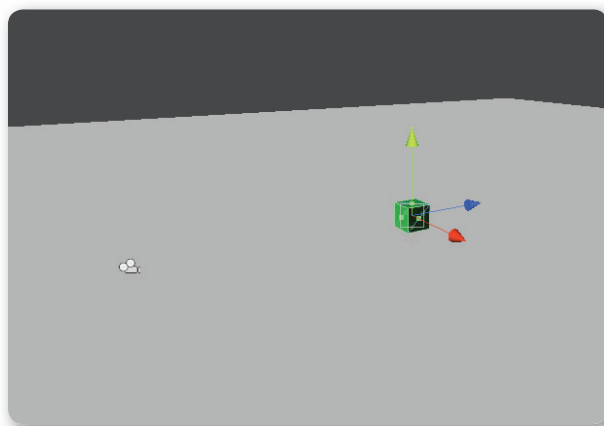


Elementos para crear la interfaz

La interfaz de usuario es muy importante porque, con ella, Unity le comunica al jugador información útil, y este puede dar información al juego, llevar a cabo configuraciones y muchas acciones más. Unity provee de varios elementos con los cuales podemos construir la interfaz. Veremos aquellos más importantes, tanto de entrada como de salida.

Label

El primer elemento que vamos a utilizar es **Label** (etiqueta), que nos permite escribir un mensaje en la pantalla. Para mostrar cómo usarlo, lo primero que tenemos que hacer es crear un nuevo proyecto al cual llamaremos **Capítulo11**. En él tendremos un cubo con el cual vamos a interactuar por medio de la interfaz. Empezamos por colocar un plano en la posición **(0,0,0)** y modificamos su escala a **(10,1,10)**; este nos ayuda a tener una referencia visual. Continuamos con la creación de un material denominado **MatVerde** y le asignamos ese verde. Luego creamos un cubo que ubicamos en la posición **(0,1,0)** y le asignamos el material recién creado. Para finalizar con la creación de la escena, adicionamos una luz de tipo **Directional Light** y la rotamos de modo que ilumine agradablemente la escena.



► **Figura 1.**
La escena con la que trabajaremos consta de un cubo que será controlado.

A continuación, creamos un script al que llamamos **ControlInterfaz** y lo asignamos al cubo. Cualquier objeto puede tener código que muestre elementos de interfaz. Lo mejor es tener un objeto que se especialice en el manejo total de la interfaz, aunque es posible que varios objetos lo hagan; la decisión depende del diseño que tengamos para nuestra aplicación.

Para poder trabajar con los diferentes elementos que componen la interfaz, tenemos que recurrir a una función especial llamada **OnGUI()**. Esta forma parte del **framework** de Unity y se especializa en la lógica relacionada con la interfaz gráfica de usuario. Se invoca en cada cuadro de animación, por lo que podemos contar con que los elementos de la interfaz se ejecutarán rápidamente. Además de usar esta función, también nos apoyaremos en la clase **GUI**, en cuyo interior están todos los elementos de la interfaz de usuario.

El primer elemento que conoceremos se conoce como **Label** (etiqueta), y se usa para imprimir información en la pantalla. Al igual que todos los elementos que veremos en este capítulo, **Label** trabaja en 2D y no forma parte del espacio de la escena, sino que, simplemente, se dibuja sobre ella. En el script que hemos creado colocamos el siguiente código:

LABEL TRABAJA
EN 2D Y SE USA
PARA IMPRIMIR
INFORMACIÓN EN
LA PANTALLA



```
#pragma strict
```

```
// Variable para el texto introducido por el usuario
```

```
private var texto:String = "Tu nombre ";
```

```
function Start () {
```

```
}
```

```
function Update () {
```

```
}
```

```
function OnGUI () {
```

```
// Variables para la posicion
```



```
var x:int = transform.position.x;
var y:int = transform.position.y;
var z:int = transform.position.z;

// Variables para la rotacion
var xr:int = transform.rotation.eulerAngles.x;
var yr:int = transform.rotation.eulerAngles.y;
var zr:int = transform.rotation.eulerAngles.z;

// Etiqueta para el texto
GUI.Label (Rect (10, 10, 200, 30), «Hola «+texto);

// Etiqueta para la posicion
GUI.Label (Rect (10, 50, 120, 30), x.ToString()+", "+y.ToString()+", "+z.
ToString());

}
```

Empezamos con la creación de una variable con acceso privado llamada **texto** y de tipo **String**. La inicializamos con el texto **Tu nombre**. Esta variable será usada junto con otro elemento, pero la declaramos en este momento pues es requerida por una de las etiquetas que vamos a tener que utilizar.

Posteriormente, en el código encontramos la función **OnGUI()**. Estamos declarando e inicializando seis variables dentro de ella. Tres se usan para guardar la posición del objeto, y las otras tres tienen los valores de rotación del objeto.



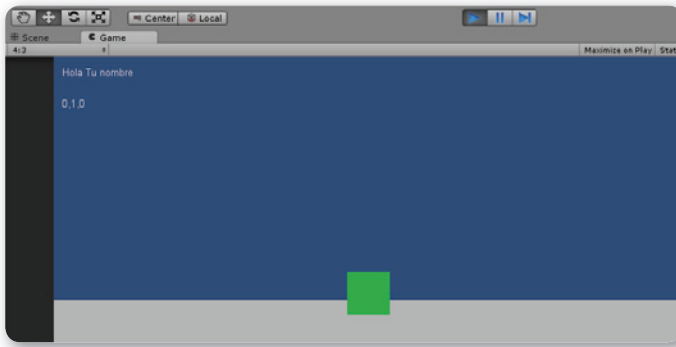
EL USO DE RECT



Rect es una estructura que define a un rectángulo y se usa para definir áreas en la pantalla que tienen esta forma. Cuando creamos el objeto, su constructor recibe cuatro parámetros. Los dos primeros son las coordenadas **X** e **Y** de la esquina superior izquierda del rectángulo, el tercer parámetro es el ancho, y el cuarto es la altura. De esta manera, podemos definir un área de la pantalla que usará varias de las funciones de GUI para saber dónde deben de dibujarse los objetos.

Para crear la etiqueta hacemos uso de la función **Label**, que pertenece a la clase **GUI**. Esta función requiere de dos parámetros: el primero es un rectángulo que indica dónde se encuentra la etiqueta, y el segundo es la cadena que deseamos mostrar. Esta puede ser colocada explícitamente, o podemos llevar a cabo una concatenación o usar alguna cadena que se creará con algún otro método. En este caso, concatenamos "Hola" y el contenido de la variable **texto**. La segunda cadena que creamos aparece un poco más abajo y mostrará las coordenadas del objeto. Hay que notar que las variables **x**, **y**, **z** hacen uso de la función **ToString()** para poder concatenar y ser mostradas.

Una vez que asignamos el script al cubo, podemos ejecutar la aplicación, y notaremos la aparición de ambas etiquetas en la pantalla.



► **Figura 2.** Podemos ver las dos etiquetas que se muestran en pantalla.

El uso de Box

Unity posee otro elemento que nos permite desplegar información en la pantalla, su uso es similar al de **Label** y se conoce como **Box** (caja). Veamos el código necesario para usarlo:

```
#pragma strict
```

```
// Variable para el texto introducido por el usuario  
private var texto:String = "Tu nombre";
```

```
function Start () {  
  
}  
  
function Update () {  
  
}  
  
function OnGUI () {  
  
    // Variables para la posicion  
    var x:int = transform.position.x;  
    var y:int = transform.position.y;  
    var z:int = transform.position.z;  
  
    // Variables para la rotacion  
    var xr:int = transform.rotation.eulerAngles.x;  
    var yr:int = transform.rotation.eulerAngles.y;  
    var zr:int = transform.rotation.eulerAngles.z;  
  
    // Etiqueta para el texto  
    GUI.Label (Rect (10, 10, 200, 30), «Hola «+texto);  
  
    // Etiqueta para la posicion  
    GUI.Label (Rect (10, 50, 120, 30), x.ToString()+",""+y.ToString()+",""+z.  
    ToString());  
  
    // Colocamos una caja  
    GUI.Box (Rect(10,90, 120, 30), xr.ToString()+",""+yr.ToString()+",""+zr.  
    ToString());  
  
}
```

Cuando usamos la función **Box()** perteneciente a la clase **GUI**, se dibuja en pantalla un rectángulo, en cuyo interior aparece el texto que deseamos mostrar. El primer parámetro debe ser un rectángulo que indica la posición y el tamaño de la caja. En nuestro ejemplo, la esquina superior izquierda se coloca en las coordenadas **(10,90)**, y la caja tiene

120 de ancho por **30** de altura, algo bastante común para la tipografía que Unity usa por default. El texto que mostramos es una concatenación de los valores de rotación del objeto. Hasta este momento, la rotación en todos los ejes es de **0**. Posteriormente, controlaremos la rotación utilizando la interfaz de usuario. Ejecutemos el programa y veamos la forma en que se muestra la caja.

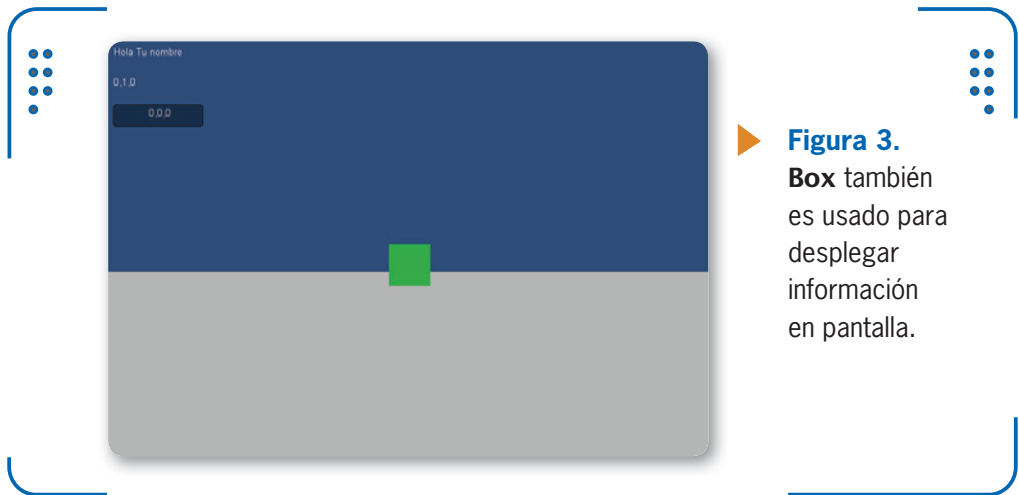


Figura 3. **Box** también es usado para desplegar información en pantalla.

El uso de Button

Hemos estado utilizando elementos de salida, pero también tenemos elementos de la interfaz gráfica de usuario que pueden usarse para introducir datos. El primero de los que vamos a utilizar es **Button** (botón), que resulta muy útil porque permite indicarle cosas al programa mediante un simple clic, como controlar objetos, designar configuraciones, tener control sobre los objetos de un inventario, y mucho más. El control se despliega en la pantalla como un objeto 2D. A diferencia de muchos lenguajes de programación que hacen uso de handlers para ejecutar un código determinado, en Unity tenemos una función para el botón que regresa **true** si se ha oprimido, o **false** en caso contrario. Esto tiende a facilitar las cosas, y el código es más claro para las personas que no tienen mucha experiencia en programación. Por lo general, usaremos la invocación a la función **Button** dentro de un **if** para colocar, en el bloque de código que corresponde, aquello que deseamos que se ejecute cuando se oprime el botón.

En nuestro ejemplo, deseamos controlar el movimiento del cubo por medio de botones. Vamos a poner cuatro, ordenados como una cruz. Cada uno controlará el movimiento en una dirección determinada, y se llevarán a cabo en el eje **X** y en el **Y**. Para lograrlo, modificamos el código del script de modo que se vea de la siguiente manera:

```
#pragma strict

// Variable para el texto introducido por el usuario
private var texto:String = "Tu nombre ";

// Variable para la cantidad de avance
private var avance:int = 1;

function Start () {

}

function Update () {

}

function OnGUI () {

    // Variables para la posicion
    var x:int = transform.position.x;
    var y:int = transform.position.y;
    var z:int = transform.position.z;

    // Variables para la rotacion
    var xr:int = transform.rotation.eulerAngles.x;
    var yr:int = transform.rotation.eulerAngles.y;
    var zr:int = transform.rotation.eulerAngles.z;

    // Etiqueta para el texto
    GUI.Label (Rect (10, 10, 200, 30), «Hola «+texto);

    // Etiqueta para la posicion
```

```

GUI.Label (Rect (10, 50, 120, 30), x.ToString()+" "+y.ToString()+" "+z.
ToString());

// Colocamos un box
GUI.Box (Rect(10,90, 120, 30), xr.ToString()+" "+yr.ToString()+" "+zr.
ToString());

// Colocamos boton y verificamos si se oprime
if(GUI.Button(Rect(40,140,60,30),"Arriba"))
    transform.Translate(0,avance,0);    // Movemos el cubo hacia arriba

// Colocamos boton y verificamos si se oprime
if(GUI.Button(Rect(10,170,60,30),"Izquierda"))
    transform.Translate(-avance,0,0);    // Movemos el cubo hacia la izquierda

// Colocamos boton y verificamos si se oprime
if(GUI.Button(Rect(70,170,60,30),"Derecha"))
    transform.Translate(avance,0,0);    // Movemos el cubo hacia la derecha

// Colocamos boton y verificamos si se oprime
if(GUI.Button(Rect(40,200,60,30),"Abajo"))
    transform.Translate(0,-avance,0);    // Movemos el cubo hacia abajo

}

```

Comenzamos por adicionar otra variable en el script, para indicar la cantidad que haremos avanzar al cubo en una dirección determinada.



EL USO DE TOSTRING()

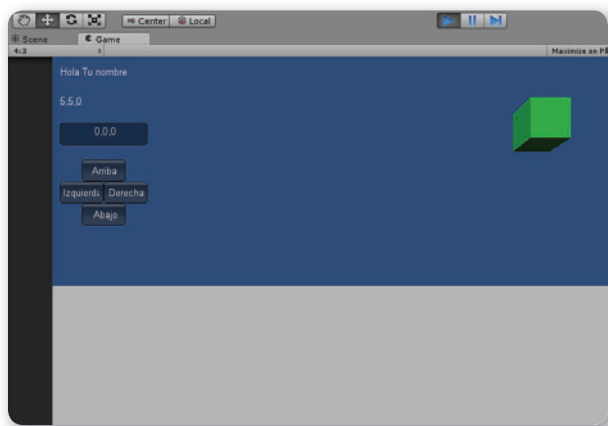


La mayoría de los objetos que usamos no son de tipo **String**, pero, como en el caso de los números, a veces necesitamos mostrarlos; **ToString()** contiene el código para representar ese valor dentro de una cadena. Son muchas las clases que permiten hacer uso de **ToString()** y, así, obtener impresiones de su estado. Es probable que en los juegos no lo utilicemos mucho, pero, para obtener información de depuración, es de gran utilidad.

La inicializamos con el valor **1**. Luego, colocamos el botón que se encarga de mover el cubo hacia arriba. Para crearlo, usamos la función **Button()**, que pertenece a la clase **GUI**. Esta función necesita dos parámetros: el primero es **Rectangle**, que representa la posición y el tamaño del botón; y el segundo es **String**, donde colocamos el texto que deseamos mostrar en el interior del botón. Este texto puede colocarse explícitamente tal como aparece en el script o puede crearse programáticamente. El botón tendrá su esquina superior izquierda en la posición **(40,140)**, y posee **60** pixeles de ancho por **30** de alto.

Como la función **Button** regresa un valor **true** o **false**, la colocamos en el **if**; el código que se ejecutará cuando se oprima el botón es una traslación en el eje **Y** dada la cantidad guardada en **avance**. De modo similar se crean los otros tres botones, cada uno trasladando hacia la dirección que le corresponde.

Con esto, podemos proceder a ejecutar la aplicación. Si oprimimos los botones, podremos observar la forma en que el cubo se mueve. También es muy interesante ver la manera en que se muestra el valor actualizado en la etiqueta.



► **Figura 4.**
Ya tenemos control sobre el cubo por medio de la interfaz de usuario.

Pedido de texto al usuario

Los botones son buenos para llevar a cabo una interacción sencilla y rápida con el usuario, pero en ocasiones, necesitamos algo más avanzado. Nuestra aplicación puede requerir que se introduzca texto

por alguna razón. Puede necesitar un **password**, un nombre del usuario, el nombre que se le dará a un personaje o, simplemente, el texto que deseamos enviar en una comunicación. Para lograr esto, haremos uso de un elemento que actúa de manera similar a los **TextBox** empleados en otros frameworks. En Unity, este elemento se crea por medio de la función **TextField()**, la cual pertenece a la clase **GUI**, al igual que las demás funciones con las que hemos trabajado.

Esta función requiere de más parámetros y regresa una cadena después de ser procesada. La cadena es el texto que contiene en su interior, es decir, el introducido por el usuario.

Los parámetros de **TextField()** son importantes. El primero es un objeto de tipo **Rect**, en el cual colocaremos la posición de la esquina superior izquierda del **TextField** y las dimensiones que va a tener en pantalla. Hay que tener en cuenta que estas dimensiones solo son usadas para delimitar el tamaño en la pantalla, no la cantidad de caracteres que puede contener el **TextField**. Todo **TextField** necesita tener una variable de tipo **String** que será asignada a él; con ella se controla y obtiene el contenido del **TextField**. Debemos pasar esta variable como segundo parámetro, ya que es usada para dibujar el contenido del **TextField** en cada invocación a **OnGUI**. El tercer parámetro es la cantidad de caracteres que puede contener el **TextField**. En este caso, es importante colocar un valor de tipo entero. Es muy común que usemos la misma variable asignada al **TextField** como la variable que recibe el valor de este; así es más fácil mantener su contenido actualizado.

Modificaremos el script para adicionar el **TextField**. No olvidemos que, anteriormente, hemos colocado una etiqueta que mostrará lo que sea que el usuario escriba en él. El código es el siguiente:

```
#pragma strict

// Variable para el texto introducido por el usuario
private var texto:String = "Tu nombre ";

// Variable para la rotacion
private var rotacion:float =0.0;
```

LA CADENA ES EL
TEXTO QUE CONTIENE
EN SU INTERIOR, ES
DECIR, EL INTRODUCIDO
POR EL USUARIO



```
// Variable para la cantidad de avance
```

```
private var avance:int = 1;
```

```
// Variable para saber si hay o no movimiento
```

```
private var movimiento:boolean =false;
```

```
function Start () {
```

```
}
```

```
function Update () {
```

```
}
```

```
function OnGUI () {
```

```
// Variables para la posicion
```

```
var x:int = transform.position.x;
```

```
var y:int = transform.position.y;
```

```
var z:int = transform.position.z;
```

```
// Variables para la rotacion
```

```
var xr:int = transform.rotation.eulerAngles.x;
```

```
var yr:int = transform.rotation.eulerAngles.y;
```

```
var zr:int = transform.rotation.eulerAngles.z;
```

```
// Etiqueta para el texto
```

```
GUI.Label (Rect (10, 10, 200, 30), «Hola «+texto);
```

```
// Etiqueta para la posicion
```

```
GUI.Label (Rect (10, 50, 120, 30), x.ToString()+" "+y.ToString()+" "+z.  
ToString());
```

```
// Colocamos un box
```

```
GUI.Box (Rect(10,90, 120, 30), xr.ToString()+" "+yr.ToString()+" "+zr.  
ToString());
```

```

// Colocamos boton y verificamos si se oprime
if(GUI.Button(Rect(40,140,60,30),"Arriba"))
    transform.Translate(0,avance,0);    // Movemos el cubo hacia arriba

// Colocamos boton y verificamos si se oprime
if(GUI.Button(Rect(10,170,60,30),"Izquierda"))
    transform.Translate(-avance,0,0); // Movemos el cubo hacia la izquierda

// Colocamos boton y verificamos si se oprime
if(GUI.Button(Rect(70,170,60,30),"Derecha"))
    transform.Translate(avance,0,0);   // Movemos el cubo hacia la derecha

// Colocamos boton y verificamos si se oprime
if(GUI.Button(Rect(40,200,60,30),"Abajo"))
    transform.Translate(0,-avance,0);  // Movemos el cubo hacia abajo

// Peticion de un texto
texto = GUI.TextField(Rect(250,10,200,30),texto,25);

}

```

La variable de tipo **String** llamada **texto** que está en la parte superior del script es la que quedará asignada al **TextField**. Le asignamos como valor inicial **"Tu nombre"**, el cual será mostrado como texto por default.

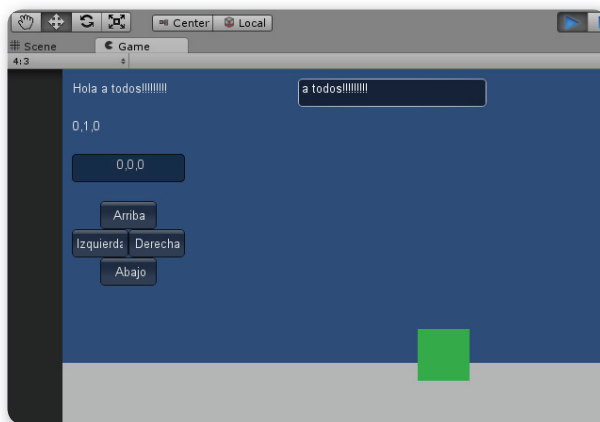


Figura 5.
El texto se captura en el **TextField** y se presenta en la etiqueta.

Más abajo, en la función **OnGUI()** adicionamos el código necesario para pedir el texto. Hacemos la invocación a **TextField**, que se dibuja en las coordenadas **(250,10)**; la cantidad de caracteres que podrá contener es de **25**. Hay que notar que la variable **texto** se usa tanto para recibir el valor que regresa **TextField**, como para indicar que se dibuja en su interior. Con esto ya podemos ejecutar la aplicación y experimentar con **TextField**.

El uso del slider

Otro elemento que nos resulta muy útil para introducir información es el **slider**, porque con él podemos indicar un valor de forma gráfica. Consiste en una barra y un indicador. La barra representa el rango de valores posibles, y la posición del indicador, el valor actual seleccionado. Los sliders son utilizados cuando hacemos configuraciones en la aplicación; por ejemplo, el valor del volumen puede ser controlado por un slider. Podemos usarlo en otro tipo de aplicaciones, siempre que deseemos dar al usuario una forma sencilla de seleccionar un valor. Las ventajas que tiene el uso del slider son varias, porque puede usarse en cualquier plataforma, no requiere de teclado, su uso es intuitivo y sencillo, y los jugadores seguramente lo habrán usado previamente en otras aplicaciones.

En nuestra aplicación de ejemplo, haremos uso del slider para controlar la rotación del cubo, es decir, según la posición del indicador, tendrá cierto ángulo. La rotación se llevará a cabo en el eje **Y**. Cuando usamos un slider, debemos tener una variable asignada a él, de modo que podamos leer fácilmente el valor y que, también, el slider se dibuje con el indicador en la posición correcta. Cuando deseamos colocar un slider, usamos la función **HorizontalSlider()**, perteneciente a la clase **GUI**,



LAS DIMENSIONES DE PANTALLA



El hecho de colocar las coordenadas de los elementos de la interfaz de forma explícita puede ser sencillo, pero, si los usuarios van a utilizar el juego con diferentes resoluciones, lo mejor es poner las coordenadas en términos de la resolución de pantalla que se está usando, generalmente, de forma porcentual. Para lograr esto, lo primero que necesitamos es conocer las dimensiones, que se encuentran en la clase **Screen**, dentro de las propiedades **Screen.width** y **Screen.height**.

al igual que los demás elementos que hemos visto. La función necesita cuatro parámetros y regresa un valor. El primer parámetro es un objeto de tipo **Rect** que utilizamos para indicar la posición donde se dibuja el slider dentro de un rectángulo imaginario; también usamos este objeto para indicar el tamaño de dibujo del slider. Cabe notar que el tamaño de dibujo no tiene relación con el rango de valores que representa el slider, sino que este se coloca de forma independiente. Al momento de definir el tamaño del slider, hay que tomar en cuenta su rango de valores: si este es muy alto y el slider pequeño, el indicador no tendrá muy buena resolución; y cambios pequeños en el indicador nos llevarán a cambios grandes en el valor indicado. El segundo parámetro es la variable relacionada con el slider, la que guarda el valor asignado; se lo usa para dibujar el indicador en la posición correspondiente. Es recomendable que esta variable sea modificada solo por el slider mismo; en caso contrario, se corre el riesgo de asignarle un valor por fuera del rango. Si es necesario modificar la variable de manera programática, debemos colocar código de seguridad para asegurarnos de que se encuentre en el rango. El tercer parámetro es el valor mínimo del rango, y el cuarto, el valor máximo. Cualquier valor que regrese el slider, según la posición del indicador, estará dentro de estos últimos dos valores. El slider devuelve un valor determinado por la posición del indicador dentro del rango. Es recomendable que la variable que reciba el valor sea la misma que hemos asignado al slider, para así garantizar consistencia entre el valor y el dibujo de ella.

EL SLIDER
PUEDE USARSE
EN CUALQUIER
PLATAFORMA, Y SU
USO ES INTUITIVO



CREACIÓN DE VENTANAS



Si deseamos crear una interfaz más compleja y que se muestre por medio de ventanas, podemos recurrir a la función **Window()**. Simplemente, hay que notar que tiene una función asociada donde colocamos los elementos que va a contener, y que puede ser cualquier elemento válido de **GUI**. El uso de ventanas puede ser atractivo, pero realmente debemos pensar si son útiles dentro del juego o si el uso de la interfaz normal es mejor.

Veamos el código que se adicionará al script:

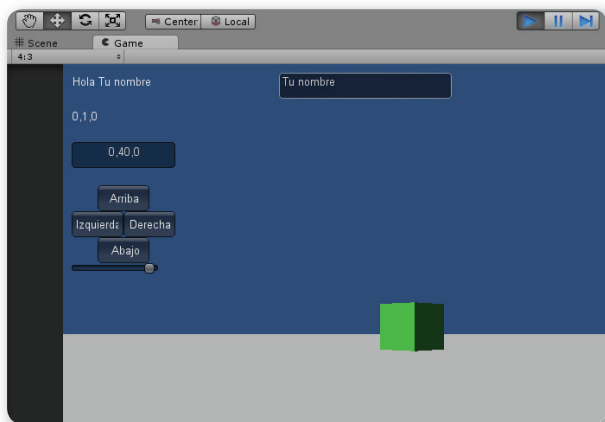
// Slider para el valor de rotacion

```
rotacion= GUI.HorizontalSlider(Rect(10,230,100,30), rotacion, -45.0, 45.0);
```

// Colocamos la rotacion en el eje Y

```
transform.rotation.eulerAngles.y=rotacion;
```

Lo primero que encontramos es la declaración de la variable **rotacion**, de tipo **float**, que inicializamos en **0.0**. Esta variable será asignada al slider y representa el ángulo con el cual rotará el cubo. Ya dentro de la función **OnGUI()**, procedemos a colocar el slider que se encuentra en la posición **(10,230)**; su tamaño es de **100** por **30**, y su rango va desde **-45.0** hasta **45.0**. Debemos notar que el slider tiene un tamaño de **100**



► **Figura 6.**
El slider permite controlar la rotación del cubo de manera sencilla e intuitiva.



INTRODUCIR TEXTO

La introducción de texto por medio de **TextField()** es sencilla, pero no todos los dispositivos donde podemos ejecutar la aplicación tienen acceso a un teclado. Para los que no lo tienen, podemos crear nuestro propio teclado por medio de botones, y hacer que este se muestre en pantalla para permitir la inserción de texto. Este tipo de técnica es muy utilizada en las consolas. Tiene la ventaja de que puede usarse en todos los dispositivos, pero la desventaja de no ser tan rápida como un teclado tradicional.

pixeles, y el rango cubre **90** valores, por lo que tenemos suficiente resolución para seleccionar un valor adecuado. El valor del indicador queda guardado en la variable **rotación**. En la línea de código inferior, simplemente asignamos el valor de rotación al eje **Y**, y con esto nuestro cubo queda rotado. Recordemos que la caja escribe los valores de rotación, de modo que podemos leer en ella el valor que hemos seleccionado por medio del slider.

Ahora, podemos ejecutar la aplicación y observar el comportamiento del slider en el cubo.



► **Figura 7.** Es posible utilizar los diversos controles que hemos colocado en la **interfaz**.

El uso de toggle

El último elemento que trataremos en este capítulo se conoce como **toggle** y funciona de manera similar al control **CheckBox** de otras APIs. Al usarlo, aparece un recuadro en pantalla, con un texto descriptivo, y al



CAPTURAR MÚLTIPLES LÍNEAS



El uso de **TextField** es útil, pero está limitado a una sola línea de texto, y aunque esto es suficiente para la mayoría de las aplicaciones, algunas veces es necesario capturar textos largos que utilizan múltiples líneas. Para estos casos, hacemos uso de la función **TextArea()** de la clase **GUI**. Su uso es muy sencillo, ya que los parámetros que utiliza son los mismos que **TextField()** y tiene los mismos significados.

hacer clic, queda seleccionado; si volvemos a hacer clic, se deselectiona. En el script que estamos creando, colocamos el valor **0** para la variable **avance** cuando el toggle no esté seleccionado, y el valor **1** cuando lo esté. De esta forma, el cubo se podrá mover únicamente con la selección.

Para usar este elemento, debemos colocar la función **Toggle()**, perteneciente a la clase **GUI**, en el script. Esta función regresa un valor de tipo **boolean**, que contiene **true** cuando está seleccionado y **false** cuando no lo está. La función necesita tres parámetros. El primero es un objeto de tipo **Rect** que utilizamos para indicar la posición y el tamaño del elemento; el segundo es una variable asignada de tipo **boolean**, que se usa para dibujar correctamente el estado del toggle; el tercer parámetro es un **String**, que utilizamos para colocar el texto descriptivo del toggle.

Agreguemos el toggle al script:

```
// Toggle para el movimiento
movimiento = GUI.Toggle(Rect(10,260,100,30),movimiento, "Permitir
avance");

// Verificamos si hay movimiento o no
if(movimiento==true)
    avance=1;
else
    avance=0;
```

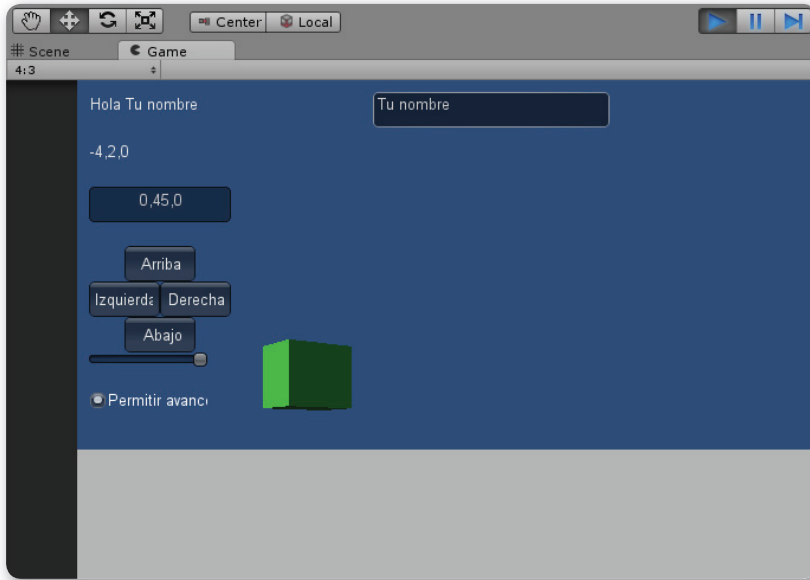
En la parte superior, inicializamos la variable **movimiento**, que es de tipo **boolean**, con **false**. Una vez dentro de la función **OnGUI()**, invocamos a **Toggle()**, que queda colocado en la posición **(10,260)**. La variable que le asignamos es **movimiento** y mostrará el texto **"Permitir avance"**. El valor



BOTÓN PARA MANTENER OPRIMIDO



El botón que hemos utilizado en el capítulo solamente responde al evento **click**, y no podemos saber si el usuario aún lo sigue oprimiendo. Para determinadas funcionalidades, esto puede ser útil, ya que podemos necesitar que cierto código se ejecute mientras se mantenga pulsado el botón. Con Unity, podemos usar la función **RepeatButton()** para cumplir este objetivo; su uso es idéntico al del botón tradicional. No hay que olvidar usar un **if**.



► **Figura 8.** Podemos observar la forma en que trabaja **toggle**.

regresado por la función queda guardado en dicha variable, que se usa para modificar el valor de avance según sea **true** o **false**. Es momento de ejecutar la aplicación y hacer uso de **toggle**.

Ya conocemos los elementos más importantes para crear interfaces de usuario que podemos utilizar en nuestros juegos, tanto para obtener información como para hacer la configuración. Un buen diseño de interfaz mejora de manera notable la experiencia del usuario.



RESUMEN

Unity nos provee de herramientas para diseñar la interfaz gráfica de usuario para nuestras aplicaciones. Para colocar los elementos, es necesario crear un script y asignarlo a un objeto. Dentro del script se usa la función **OnGUI()**, la cual es necesaria para colocar el código de la interfaz de usuario. Los elementos usan un objeto de tipo **Rect** para colocar su posición y tamaño. Algunos requieren de una variable que se les asigne para regresar un valor y poder dibujarse correctamente. Hay elementos que nos permiten mostrar información, y otros que podemos utilizar para introducirla.

Actividades

TEST DE AUTOEVALUACIÓN

- 1 ¿Cuál es el uso de la función **OnGUI()**?
- 2 ¿Qué clase contiene los elementos para crear interfaces de usuario?
- 3 Describa los parámetros usados en **Rect**.
- 4 ¿Qué formas existen para mostrar información en la pantalla?
- 5 ¿Cuál es el uso de **ToString()**?
- 6 ¿Cómo se crea un botón en Unity?
- 7 ¿Qué elemento es equivalente a un **CheckBox**?
- 8 ¿Qué usos se le pueden dar a un slider?
- 9 ¿Por qué es bueno usar la misma variable para recibir y mostrar información?
- 10 ¿Qué tipo de variable se usa para un toggle?
- 11 ¿Cuántos tipos de sliders hay?
- 12 ¿Por qué el botón requiere de un **if**?

ACTIVIDADES PRÁCTICAS

- 1 Coloque botones para mover el cubo en el eje **Z**.
- 2 Cree otro objeto con su propio script que coloque elementos en la interfaz.
- 3 Adicione un slider vertical para girar al cubo en el eje **Y**.
- 4 Cree una interfaz de configuración.
- 5 Genere una interfaz de usuario para alguno de los proyectos realizados en los capítulos anteriores.



Proyecto integrador y distribución

En este capítulo llevaremos a cabo un proyecto de integración de los temas que hemos aprendido a lo largo del libro y, posteriormente, lo dejaremos listo para distribuir. Usaremos scripts, audio, iluminación e importación de personajes animados desde Blender. Para lograr la distribución, es necesario seleccionar la plataforma que deseamos, ya sea computadora, consola o la Web. Al final, tendremos un juego creado con Unity y Blender.

▼ Creación y animación

en Blender292

El barco292

Las olas297

El paracaídas297

El pelícano.....298

El tiburón.....299

▼ Programación en Unity300

El objeto barco300

El objeto olas.....301

El objeto paracaídas301

El objeto pelícano304

El objeto tiburón.....306

El objeto cámara308

Creación de la escena311

▼ Construir la aplicación.....314

▼ Resumen.....315

▼ Actividades.....316





Creación y animación en Blender

El proyecto consiste en un juego en el que tenemos un barco que el usuario puede mover al oprimir las flechas del teclado. Cerca del barco habrá dos tiburones que nadarán recorriendo el mar; ellos formarán parte de los antagonistas. Desde el cielo, caerán paquetes en paracaídas que serán desplazados por el viento. El barco deberá recogerlos sin que los tiburones lleguen a ellos. El juego tendrá otro antagonista, que será un pelícano que, al volar, tiene la posibilidad de colisionar contra el paquete. Cuando esto suceda, el paquete caerá bruscamente al mar, y hará mucho más difícil que el barco logre obtenerlo.

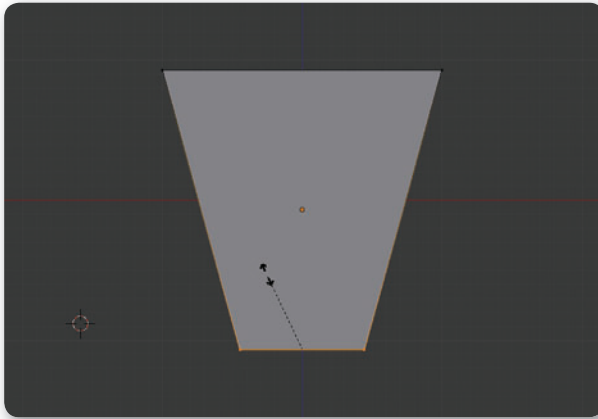
El escenario tendrá otro objeto con forma de olas que servirá para darle una mejor presentación al juego. El fondo del escenario será de color azul cielo, y, en la parte superior, tendremos elementos de **GUI** para mostrar la información pertinente.

Ahora debemos ver más a fondo cada uno de los objetos que componen el juego y lo que deben hacer. Los objetos tienen que ser modelados y animados en Blender, del modo aprendido en el **Capítulo 9**. Mostraremos la manera de crear y animar el barco, y con los demás conocimientos que ya hemos incorporado, procederemos a generar los demás objetos siguiendo los mismos pasos.

El barco

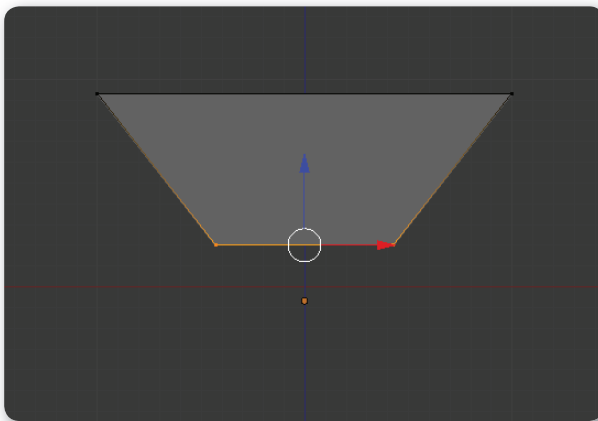
Empezamos creando un proyecto nuevo en Blender y, luego, borrando el cubo que aparece al principio. Luego, nos colocamos con una vista en modo ortogonal oprimiendo la tecla **5** del teclado alfanumérico seguida de la tecla **1**, para quedar con una vista frontal. Creamos un nuevo cubo, lo seleccionamos y entramos al modo de edición pulsando **TAB**.

Una vez que estamos dentro del modo de edición, oprimimos la tecla **A**, con el objetivo de deseleccionar todo y luego seleccionamos los cuatro vértices inferiores del cubo. Para facilitar esta acción, si es necesario, podemos cambiar la vista. Pulsamos la tecla **S** para activar el comando de escala, y enseguida oprimimos **X**, para limitar la escala en el eje **X**. Nos movemos con el mouse y reducimos la escala de la base del cubo hasta tener un trapecio invertido.



► **Figura 1.**
Al cambiar la escala, creamos la figura que necesitamos para el barco.

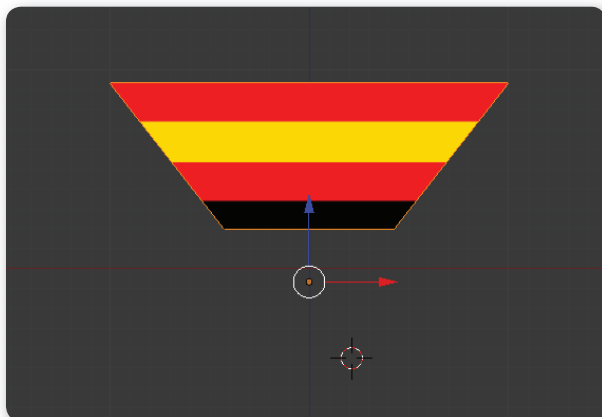
El trapecio es demasiado alto para que luzca del modo en que deseamos, por lo que hacemos clic en la flecha azul del gizmo de traslación, que está en la pantalla, y arrastramos hacia arriba, hasta darle la proporción correcta al casco del barco.



► **Figura 2.**
Al mover los vértices hacia arriba, modificamos la forma de la figura.

Aprovechando que estamos en modo de edición, adicionamos la textura. Lo primero que debemos hacer es llevar a cabo un unwrap y luego colocar la textura que deseamos utilizar.

Al terminar la asignación, salimos del modo de edición y damos al objeto una escala de **0.25** en el eje **Y**.

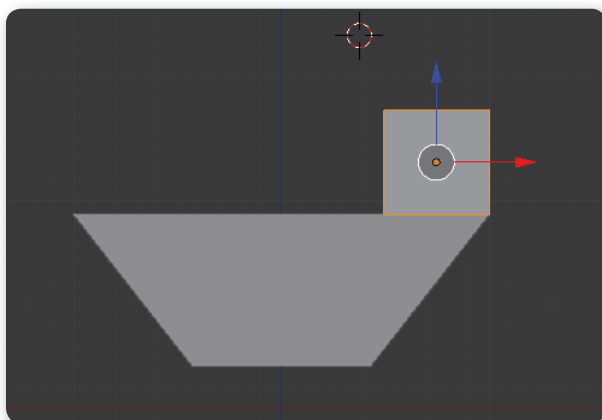


► **Figura 3.**
El casco del barco tiene asignada la textura que deseamos usar.

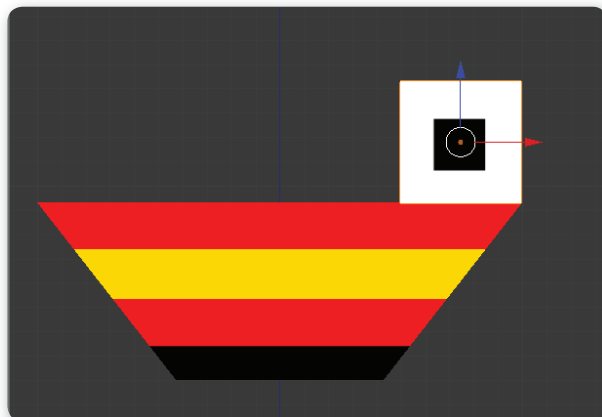
El siguiente elemento que debemos crear es la cabina del barco y, para hacerlo, usamos un cubo al que le damos el valor **0.25** en todos sus ejes. Luego lo trasladamos hasta el lugar en que se pose sobre la tapa del casco en su lado derecho.

Al igual que hicimos con el casco, deberemos adicionar una textura para esta cabina. La ventaja de usar un cubo es que el unwrap y la asignación de texturas son sencillos.

El último elemento que necesitamos colocar es la chimenea del barco, la cual también se crea a partir de un cubo. Para hacerlo correctamente, modificamos la escala del cubo y le asignamos valor **0.1** para los ejes **X** e



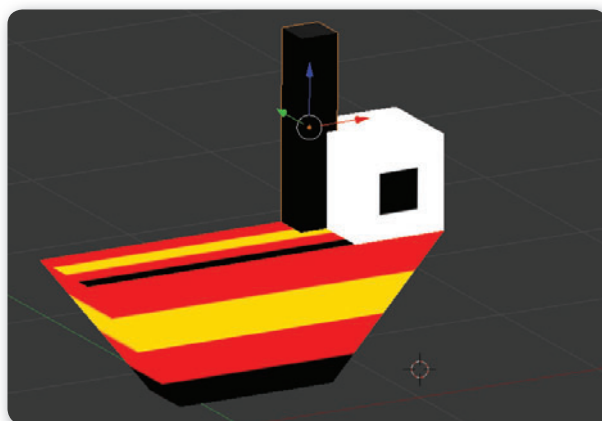
► **Figura 4.**
Un nuevo cubo, pero más pequeño, es la forma que representa a la cabina.



► **Figura 5.**
El barco luce mucho mejor luego de haberle aplicado las texturas elegidas.

Y; el eje **Z** tendrá **0.5** como valor. Una vez que tenemos la forma correcta, movemos la chimenea junto a la cabina para que se vea bien. El último paso es asignarle a la chimenea un material de color negro.

La animación del movimiento del barco es sencilla: un movimiento hacia arriba y hacia abajo. Para hacerlo, necesitamos únicamente diez cuadros de animación, y la realizamos por medio de la traslación. El movimiento es sencillo: en el primer keyframe, el barco se coloca a una altura de **1.6** unidades; para el segundo keyframe, que está en el cuadro **5**, elevamos el barco a **2.1** unidades; el último tiene una posición de **1.6** unidades otra vez. Esto le da al barco el movimiento de subir y bajar,

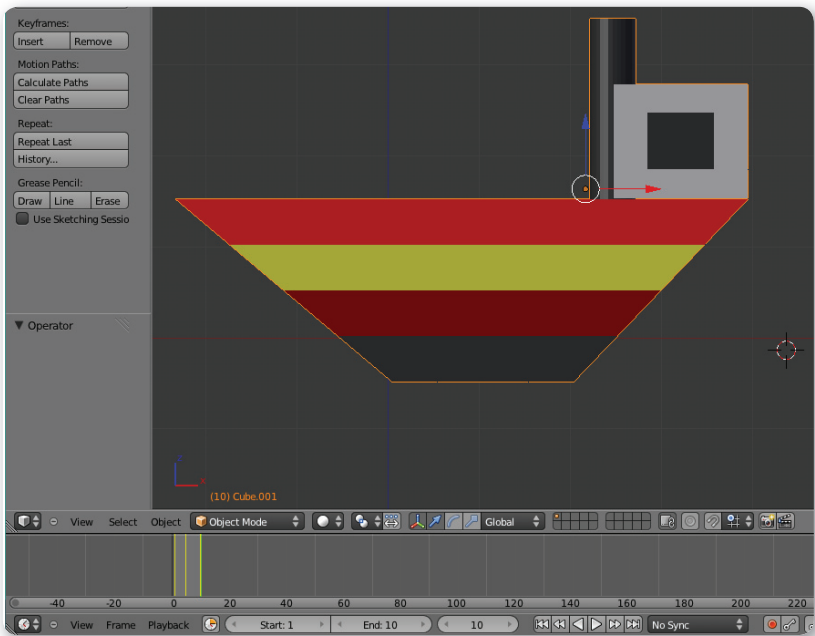


► **Figura 6.**
La chimenea es colocada en su lugar, junto a la cabina.

como si fuera impulsado por las olas del mar. La embarcación está compuesta por tres objetos que debemos unir en uno solo para facilitar la animación. Entonces, seleccionamos los tres objetos y oprimimos **CTRL + J**. Los valores de animación que deberán utilizarse son los que pueden observarse en la **Tabla 1**.

INFORMACIÓN DE ANIMACIONES			
CUADRO	EJE X	EJE Y	EJE Z
1	2.2	0	1.6
5	2.2	0	2.1
10	2.2	0	1.6

Tabla 1. Estos valores de coordenadas son las correspondientes a la animación del barco.



► **Figura 7.** El proceso de animación es sencillo y puede realizarse de manera veloz.

Las olas

El siguiente objeto que es necesario modelar es la ola. No hace falta modelar una ola muy grande, ya que una pequeña repetida varias veces servirá igual. En la ola que modelaremos, podemos colocar cuatro cúspides y asignarle un material de color azul un poco oscuro. Al igual que en el caso del barco, podemos crear una animación para que el juego luzca mejor. Esta puede ser un vaivén de izquierda a derecha y de arriba hacia abajo, que tome desde el cuadro **1** hasta el **25**; la llamamos **Movimiento**. Al importar el objeto, la animación puede tener el **WrapMode** en **Loop**, de modo que se repita constantemente.

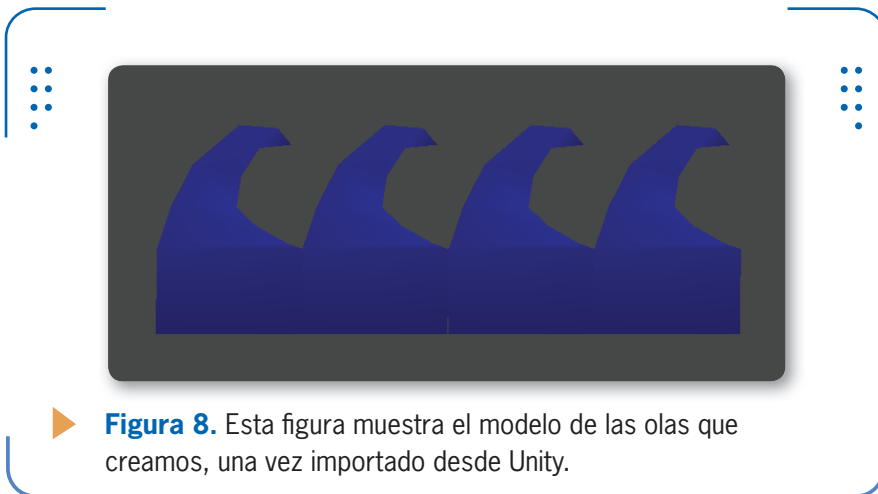


Figura 8. Esta figura muestra el modelo de las olas que creamos, una vez importado desde Unity.

El paracaídas

Este objeto es importante porque contiene la carga que el barco debe rescatar. El modelo es un poco más complejo que los anteriores, ya que es necesario modelar más, y usa más objetos. En primer lugar, debe tener un paracaídas al que le damos color blanco. Este va a tener dos o más cuerdas que llegarán hasta la carga, de un material con color que contraste, como gris o café. En la parte inferior, donde se unen todas las cuerdas, colocamos la carga, representada por un cubo al que podemos colocarle un material o algún tipo de textura, ya sea de madera o de metal, algo que lo haga verse real.

Al tener varios objetos, podemos animarlos en forma independiente. La animación que vamos a crear debe corresponder al movimiento de

un objeto que cae en paracaídas, y que puede girar con suavidad de derecha a izquierda. El cubo puede tener un movimiento pendular. Con esto lograremos una animación atractiva y, al ser sencilla, podemos usar pocos cuadros, ya que este tipo de movimiento puede repetirse constantemente. Alrededor de **10 a 20** cuadros de animación serán más que suficientes en este caso.



► **Figura 9.**
Este es solo uno de los ejemplos de la manera en que podemos modelar el paracaídas.

El pelícano

Este pájaro necesita tres materiales. Para las plumas del cuerpo y las alas, un material en tono blanco; para las patas y el pico, otro en un tono amarillo o naranja; y para el ojo, uno negro. Podemos animar el movimiento del cuerpo, las patas, el pico e, incluso, las alas. Dicha animación puede llamarse **Movimiento**, al igual que en los otros objetos, y puede usar alrededor de **30** cuadros. El **WrapMode** puede ser **PingPong** o **Loop**, dependiendo de la forma en que se crea la animación.



POSICIONES DIFERENTES



Dependiendo del tamaño con que se crearon los modelos en Blender, es posible que se necesiten posiciones diferentes de las que se indican en el capítulo. Sin embargo, esto no supone ninguna dificultad; lo importante es tener una escena similar y que brinde el mismo funcionamiento. Las figuras nos ayudan a guiarnos para lograr este resultado y, también, tenemos la libertad de hacer las modificaciones y mejoras al juego que deseemos.



► **Figura 10.**
El pelicano
modelado,
al momento
de ser
importado
desde Unity

El tiburón

El tiburón requerirá de cuatro materiales para lucir bien. Uno blanco para los dientes y uno negro para el ojo; en la parte inferior se le puede colocar otro de color gris; y en la superior, uno azul o gris azulado. No es necesario modelar todo el cuerpo del tiburón, ya que las olas solamente dejarán ver la cabeza.

La animación deberá mostrar al tiburón nadando. Si lo deseamos, podemos animar las quijadas también. El movimiento no debe ser suave, porque queremos darle una personalidad agresiva.

Con esto ya tenemos todos los objetos que usaremos para el juego. Ahora hay que importarlos desde Unity.



► **Figura 11.**
Esta figura
muestra el
modo en que
lucirá el tiburón,
una vez que
sea importado
desde Unity.



Programación en Unity

Lo primero que necesitamos es crear una nueva aplicación a la cual llamamos **Tiburón**. Luego, procedemos a importarla tal como aprendimos en el **Capítulo 9**. Una vez importadas todas las geometrías, será necesario colocar cada una en la escena con el fin de darles los scripts correspondientes y, luego, crear los prefabs que correspondan.

La escena necesita una luz de tipo **Directional Light**, de color blanco, ubicada en la posición **(0.3,0.25,-3.40)**. La rotación es de **(0,0,0)**, y la intensidad, de **0.5**.

El objeto barco

Una vez que el barco está en escena, lo seleccionamos y le adicionamos un **Box Collider**, que no debe cubrirlo a la perfección. No tiene que quedar mucho espacio entre el barco y el límite del **Box Collider**. Después, creamos un script denominado **ControlBarco** y le adicionamos el siguiente código:

```
// Propiedad para la velocidad del barco
var velocidad:float;

function Update () {

    // Movemos el barco en el eje X
    transform.Translate(-Input.GetAxis("Horizontal")*velocidad,0,0);

    // Evitamos que el barco salga fuera de rango
    if(transform.position.x<-15)
        transform.position=Vector3(-15,-11,12);
    else if(transform.position.x>15)
        transform.position=Vector3(15,-11,12);

}
```

El código es sencillo. En primer lugar, tenemos una propiedad llamada **velocidad**, que indica la rapidez con que se moverá el barco. No olvidemos asignar el script al barco y colocarle **0.1** como valor. Si lo deseamos, podemos hacer que sea más o menos rápido. En la función

Update(), verificamos si se ha usado el movimiento en el eje horizontal del control, que equivale también a las flechas derecha e izquierda del teclado, y basándonos en esto, trasladamos el barco.

Como no deseamos que el barco exceda cierto límite de movimiento, se coloca un código que lo verifique y permita delimitarlo en su posición. Cuando todo está listo, creamos un prefab llamado **BarcoCompleto** y le asignamos el barco con los componentes que hemos adicionado al prefab.

El objeto olas

El prefab que corresponde a las olas es el más sencillo. En realidad, no hay que adicionar nada al objeto; simplemente, creamos un prefab llamado **OlasCompleto** y le asignamos la geometría de las olas. No precisará ningún script.

El objeto paracaídas

El paracaídas es el objeto más complicado que tenemos. Empezamos por crear el prefab **ParacaidasImagen**, al cual agregamos el modelo del paracaídas después de haber colocado un componente **RigidBody**, un **BoxCollider** y un script llamado **ControlParacaidas**. Para este script es necesario conseguir cuatro sonidos: uno para la colisión con el pelícano, otro para la colisión con el tiburón, el siguiente para cuando el paquete toca el mar, y el último cuando aterriza debidamente en el barco. El código del script es el siguiente:

```
var choqueTiburon: AudioClip;
var choqueAgua: AudioClip;
var choqueBarco: AudioClip;

private var chocado:boolean;

// Inicializamos
function Start()
{
    // Obtenemos la camara
    controlador=GameObject.Find("MainCamera");
```

```
// Indicamos que no ha chocado
chocado=false;
}

function Update () {

    // Verificamos si estamos en algun lugar para destruir el paracaidas
    if(chocado==false)
    if(transform.position.y<-15 || transform.position.x<-19.5 || transform.
    position.x>19.5)
    {
        // Lo que sucede si el paracaidas cae al agua
        audio.PlayOneShot(choqueAgua);
        controlador.GetComponent("ControlGeneral").ParacaidasCaido();
        Destroy(gameObject,1);
        chocado=true;
    }

    // Simulamos el viento
    // El rango varia dependiendo de la masa
    if(Random.Range(1,100)<10)
        rigidbody.AddForce(Vector3(Random.Range(-200,200),0,0));
}

// Verificamos por colisiones
function OnCollisionEnter(collision : Collision)
{
    // Contra el tiburon
    if(collision.gameObject.name=="TiburonFrontal" || collision.gameObject.
    name=="TiburonPosterior")
    {
        controlador.GetComponent("ControlGeneral").Paracaidas
        Comido();
        particleEmitter.emit=true;
        audio.PlayOneShot(choqueTiburon);
        Destroy(gameObject,1);
    }

    // Contra el barco
```

```
else if(collision.gameObject.name=="BarquitoCompleto")
{
    //Aqui paracaidas salvado
    audio.PlayOneShot(choqueBarco);
    controlador.GetComponent("ControlGeneral").Paracaidas
    Salvado();
    Destroy(gameObject,0.1);
}

// Contra el pelicano
else if(collision.gameObject.name=="PelicanoCompleto")
{
    rigidbody.drag=0.5;
    audio.PlayOneShot(choquePelicano);
}
}
```

En el script, lo primero que hacemos es encontrar la cámara y guardar una referencia a ella en la variable **controlador**. Esto es necesario porque la cámara tendrá un script que controlará la lógica general del juego, y tenemos que comunicarnos con ella para darle información y actualizar la interfaz de usuario. Para lograrlo, usamos la función **Find()**, que pertenece a la clase **GameObject** y que necesita como parámetro el nombre del objeto que deseamos encontrar.

En la función **Update()** procederemos a verificar si el paracaídas se ha salido fuera de rango o si ha colisionado con el agua. En el caso de que alguna de estas opciones suceda, se reproducirá el sonido correspondiente al choque con el agua y se invocará la función **ParacaidasCaído()**, que pertenece al script de la cámara; posteriormente, se completa la destrucción del objeto.



MÚSICA PARA JUEGOS



Un buen lugar para encontrar diferentes tipos de música que podemos utilizar en nuestros juegos, incluido un pequeño sistema de búsqueda por estilo y por tema, es el sitio del compositor Kevin MacLeod. Allí nos dirigimos a la sección **Royalty-Free Music**. La música es un elemento importante para generar la ambientación y no debemos olvidarla. La dirección del sitio web es: <http://incompetech.com/music/royalty-free>.

Aprovechamos la función **Update()** para colocar un movimiento aleatorio vía una fuerza, que simule al viento. Para hacerlo, usamos el **RigidBody** que se ha colocado y, por medio de la función **AddForce**, indicamos una fuerza en **X** que puede ir de **-200** a **200**.

En la función **OnCollisionEnter()**, verificamos por colisiones contra los objetos del juego. En primer lugar, verificamos si la colisión se ha llevado a cabo con alguno de los tiburones. Habrá dos tiburones en el juego, y es necesario verificar la colisión contra ambos. Para que esto funcione, el **BoxCollider** debe tener la suficiente profundidad como para pasar por la zona de ambos animales. Cuando se colisiona con alguno de ellos, se invoca la función **ParacaidasComido()**, que pertenece al script de la cámara, y se activa el emisor de partículas al mismo tiempo que se reproduce el sonido correspondiente a la colisión con el tiburón; un segundo después se destruye el objeto. El emisor de partículas es sencillo, puede mostrar una pequeña explosión o salpicadura. Lo importante es colocar colores que contrasten con los objetos que tenemos. Cuando la colisión se da contra el barco, se reproduce, en primer lugar, el sonido correspondiente y, luego, se invoca la función **ParacaidasSalvado()** del script de la cámara; al final se destruye el objeto. El **RigidBody** del paracaídas debe tener el valor **5** en la propiedad **Drag**, porque la usaremos para dar un buen efecto de caída cuando colisione contra el pelícano. En esta colisión, simplemente, modificamos el valor de **Drag** a **0.5** y reproducimos el sonido correspondiente. Como este objeto va a reproducir audio, no debemos olvidarnos de agregar un componente **AudioSource**.

El objeto pelícano

A la geometría del pelícano, debemos añadirle un componente de tipo **BoxCollider**, que debe cubrirlo perfectamente, de manera ajustada. El ave volará de derecha a izquierda, y viceversa. Cada vez que llegue al extremo de un lado, girará y esperará un tiempo para regresar con una velocidad y altura seleccionadas al azar.

El pelícano necesita un script denominado **ControlPelicano**, con el siguiente código:

```
private var velocidad: float;  
private var xMax: float;
```

```
private var xMin: float;
private var velExtra:float;
private var enEspera:boolean;

// Inicializamos valores
function Start()
{
    xMax=20;
    Xmin=-20;
    velocidad=-1.0;
    velExtra=5;
    animation["Movimiento"].speed=1;
}

function Update () {

    // Si no esta en espera avanzamos
    if(!enEspera)
        transform.Translate(Vector3(velocidad*Time.deltaTime,0,0));

    // Verificamos que estamos en el limite
    if(transform.position.x>20)
    {
        velocidad=Random.Range(1.5, 1+velExtra);
        transform.position=Vector3(19.5,Random.Range(-5,10),11);
        transform.localScale=Vector3(-1,1,1);
        Espera();
    }

    // Verificamos el otro limite
    if(transform.position.x<-20)
    {
        velocidad=-(Random.Range(1.5, 1+velExtra));
        transform.position=Vector3(-19.5,Random.Range(-5,10),11);
        transform.localScale=Vector3(1,1,1);
        Espera();
    }
}
```

```
function Espera()
{
    // Hacemos esperar al pelicano
    enEspera=true;
    yield WaitForSeconds((Random.Range(5,15)));
    enEspera=false;
}
```

Empezamos inicializando las variables con los valores convenientes. Posteriormente, podemos modificarlos para ajustarlos de manera favorable. Observaremos que también se está colocando la velocidad con la que va a mostrarse la animación. En este caso, colocamos el valor **1**.

En la función **Update()**, verificamos si el pelícano no se encuentra en estado de espera, en cuyo caso lo hacemos avanzar de acuerdo con la velocidad. Tenemos dos **if** que nos ayudan a verificar si el ave ha llegado a uno de los límites. En caso de que esto suceda, se selecciona una velocidad nueva al azar y se coloca también en una nueva posición con la altura seleccionada por un valor aleatorio. Para hacer que el pelícano dé la vuelta, le ponemos una escala de **1** o **-1**, dependiendo de la dirección a la que se dirija. Este cambio de escala hace que el pelícano gire a la derecha o a la izquierda; también la velocidad tiene un cambio de signo que nos permite variar la dirección. En ambos casos, hay un tiempo de espera. Para crearlo, invocamos la función **Espera()**, y, en su interior, colocamos la variable **enEspera** con valor **true**, para evitar el movimiento del ave. Al usar **yield WaitForSeconds**, hacemos que el script entre en pausa durante una cantidad determinada de segundos, sin que el resto de la aplicación se detenga. Al terminar la espera, que puede ser de **5** a **15** segundos, colocamos en la variable **enEspera** el valor **false**, para que el movimiento del pelícano continúe.

El objeto tiburón

A la geometría del tiburón hay que adicionarle un **BoxCollider**; en este caso, es mejor que cubra solo la parte desde las mandíbulas hasta la aleta, no todo el cuerpo, ya que únicamente las mandíbulas son peligrosas. El tiburón también tiene un script que lo va a controlar, y lo llamamos **ControlTiburon**. Su código es el siguiente:

```
var offset=0;
private var velocidad: float;
private var xMax: float;
private var xMin: float;
private var velExtra:float;

// Inicializamos variables
function Start()
{
    xMax=20;
    Xmin=-20;
    velocidad=-1.0;
    velExtra=5;
    animation["Movimiento"].speed=1;

    // Colocamos su posicion
    transform.position=Vector3(Random.Range(-18,18),-11-(offset/2),
    11+offset);
}

function Update () {

    // Avanzamos el tiburon
    transform.Translate(Vector3(velocidad*Time.deltaTime,0,0));

    // Verificamos que estamos en el limite
    if(transform.position.x>20)
    {
        velocidad=Random.Range(0.5, 1+velExtra);
        transform.position=Vector3(19.5,-11-(offset/2),11+offset);
        transform.localScale=Vector3(-1,1,1);
        animation["Movimiento"].speed=Random.Range(0.5,2);
    }

    if(transform.position.x<-20)
    {
        velocidad=-(Random.Range(0.5, 1+velExtra));
        transform.position=Vector3(-19.5,-11-(offset/2),11+offset);
        transform.localScale=Vector3(1,1,1);
        animation["Movimiento"].speed=Random.Range(0.5,2);
    }
}
```

```
}  
  
}
```

El tiburón tiene un comportamiento similar al del pelícano, ya que, al llegar al límite derecho o izquierdo, da la vuelta. En el script, empezamos por colocarles los valores iniciales a las variables, y le damos a la animación una velocidad de **1**. La variable **offset** se usa para identificar al tiburón y colocarlo en el lugar que corresponde. No tendremos problemas para entender la función **Update()**. Lo primero es hacer avanzar al tiburón. Si se detecta que ha llegado a cualquiera de los límites, se selecciona una nueva velocidad y se lo coloca en una posición adecuada, de una manera muy similar a lo que ocurre con el pelícano. Por medio de la escala, lo ponemos viendo hacia una dirección u otra; luego, asignamos la velocidad de la animación con un valor aleatorio, para darle un poco de dinamismo al juego.

El objeto cámara

La cámara también tendrá un script, ya que la usaremos como objeto de control. Creamos uno llamado **ControlGeneral**, al que le asignamos el siguiente código:

```
var paracaidas:GameObject;  
var guiSkin: GUISkin;  
private var tiempo:float;  
private var deltaTiempo:float;  
private var maxtiempo:float;  
private var valorDrag:float;
```



CREAR TIEMPOS DE ESPERA



Una instrucción importante es **yield WaitForSeconds()**, debido a que con ella podemos hacer que un script en particular se quede detenido durante cierta cantidad de segundos. Hacemos uso de esta instrucción en el proyecto del capítulo; su uso es sencillo, pues solo necesita de un parámetro: la cantidad de segundos que deseamos detener el script. **Yield** también puede usarse para trabajar con **corutinas**, que pueden darnos funcionalidades extras.

```
private var perdidos:int;
private var comidos:int;
private var salvados:int;
private var estado:int;
private var hayAudio:boolean;

// Inicializamos valores
function Start()
{
    maxtiempo=5;
    tiempo=maxtiempo;

    deltaTiempo=0.1;
    valorDrag=5;
    perdidos=0;
    comidos=0;

    estado=0; //0-juego, 1-juego terminado
    hayAudio=false;
}

function Update () {

    // Para el conteo del tiempo
    if(estado==0)
    {
        tiempo-=Time.deltaTime;

        if(tiempo<=0)
        {
            tiempo=maxtiempo;
            CreaParacaidas();
        }

        if(perdidos+comidos>15)
            estado=1;
    }

    if(estado==1)
        Despedida();

    audio.mute=hayAudio;
}
```

```
function OnGUI()
{
    // Colocamos la informacion
    GUI.skin=guiSkin;
    if(estado==0)
        GUI.Box(Rect((Screen.width/2)-250,10,500,50),"Salvados: "+
        salvados+", Comidos: "+comidos+", Perdidos: "+perdidos);
    else
        GUI.Box(Rect((Screen.width/2)-250,10,500,50),"Lograste
        salvar "+salvados+" amigos.");

        hayAudio= !GUI.Toggle(Rect(Screen.width-100, Screen.height-40,
        100, 30), hayAudio, "Audio");
}

function CreaParacaidas()
{

    var temp =Instantiate(paracaidas,Vector3(Random.Range
    (-14,15),14,14.5),Quaternion.identity);
    var tempRB = temp.GetComponent(Rigidbody);
    tempRB.drag=valorDrag;

    DecrementaTiempo();
    DecrementaDrag();

}

public function DecrementaTiempo()
{
    maxtiempo-=deltaTiempo;
    if(maxtiempo<1.5)
        maxtiempo=1.5;
}

public function DecrementaDrag()
{
    valorDrag-=0.1;
    if(valorDrag<1.5)
        valorDrag=1.5;
}
```

```
}

// Funciones invocadas por otros objetos
public function ParacaidasCaido()
{
    DecrementaDrag();
    DecrementaTiempo();
    perdidos++;
}

public function ParacaidasComido()
{
    DecrementaDrag();
    DecrementaTiempo();
    comidos++;
}

public function ParacaidasSalvado()
{
    salvados++;
}

function Despedida()
{
    yield WaitForSeconds(5);
}
```

El script funciona como una pequeña máquina de estados finitos, en la cual el estado **0** es el juego activado, y el estado **1** es el juego terminado. Hacemos uso de un pequeño contador de tiempo para llevar a cabo la creación del paracaídas. El script también se encarga de la presentación de la interfaz gráfica, para lo cual recibimos el diseño para la tipografía por medio de una propiedad de tipo **GUISkin**. Las funciones que se invocan en los otros scripts, nos ayudan a modificar las variables que guardan los resultados del juego y que, luego, son desplegados.

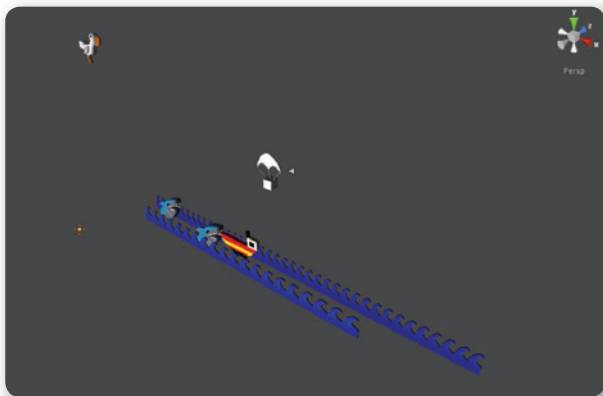
Creación de la escena

Ahora, con los prefabs ya creados, podemos borrar los objetos importados, y usar los prefabs para la construcción de la escena.

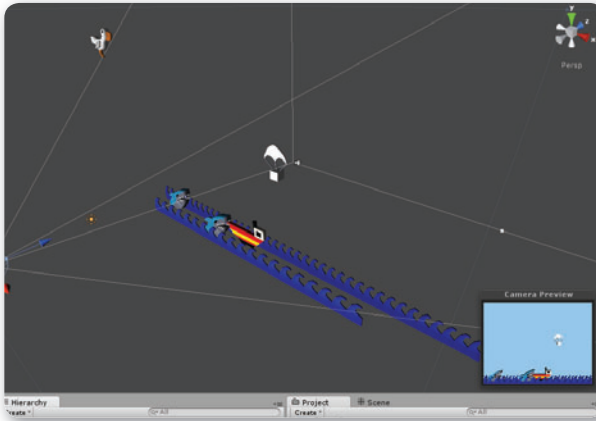
Empezamos por colocar la banda frontal de olas, que está compuesta por cinco olas en fila y agrupadas por parenting. La posición del padre debe ser **(-15,-11,10)**. Luego, de manera similar, creamos la banda de olas posterior, que tiene un parenting y, en su interior, una fila de **10** olas. La posición donde se encuentra el parenting es **(-19,-13,14)**. Luego ponemos los objetos en la escena.

Ahora creamos un barco desde su prefab, al cual ubicamos en la posición **(0,-11,12)**. Después, verificamos que la propiedad **Velocidad** tenga el valor **0.1** en su script. Creamos el primer tiburón y lo llamamos **TiburónFrontal**; el nombre es importante porque el script identifica la colisión con el nombre. En su propiedad **Offset** debe llevar el valor **0**. Las coordenadas donde lo colocamos son **(-2.85,-11,11)**. El siguiente tiburón se llama **TiburónPosterior** y debe tener valor **2** en la propiedad **Offset**. Las coordenadas donde lo ubicamos son **(-13.7,-12,13)**. Para el pelícano usamos el nombre **PelicanoCompleto** y lo ponemos, inicialmente, en **(-21,10.35,11)**. Después colocamos el objeto paracaídas, llamado **ParacaidasCompleto**, y ponemos los recursos de audio correspondientes en las propiedades. La posición del objeto es **(6.41,0,15)**.

Es importante que la cámara se coloque en la posición **(0,0,-10)** sin ningún tipo de rotación. En el componente **Camera** de **MainCamera**, hay una propiedad llamada **Background**. En ella seleccionamos un color azul cielo para el fondo de la escena. El script asignado a la cámara es **ControlGeneral**; en la propiedad **Paracaidas**, ponemos el prefab que corresponde al paracaídas, y en la otra propiedad colocamos el objeto **GUISkin**, que pronto crearemos.



► **Figura 12.**
La escena que hayamos construido tendría que lucir de un modo similar al de la figura.

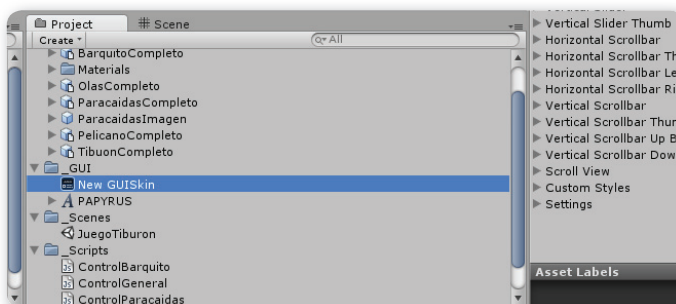


► **Figura 13.**
La cámara suele colocarse con un color de fondo.

Creación de un GUISkin

Un **GUISkin** nos permite cambiar la tipografía de los elementos que usamos para la interfaz gráfica. Para agregarlo, vamos al panel **Project** y seleccionamos el menú **Create**, donde encontramos la opción **GUISkin**. Una vez creada, hacemos clic sobre ella y, en el panel **Inspector**, vemos una lista de los componentes. Seleccionamos **Box**, y aparecen las propiedades que podemos modificar; aquí cambiamos el componente a nuestro gusto. Ya que hemos modificado el **Box** como preferimos, procedemos a asignar este **GUISkin** a la propiedad en el script de la cámara.

Una vez que hayamos terminado, podremos ejecutar la aplicación y tendremos la posibilidad de jugar con el barco.



► **Figura 14.** El **GUISkin** puede agregarse seleccionando la opción adecuada en el menú **Create**, dentro del panel **Project**.



Figura 15.

El juego se ejecuta con todos los objetos comportándose de la manera esperada.

➤ Construir la aplicación

El último paso cuando terminamos un juego es construir la aplicación para poder distribuirlo. La distribución para consolas requiere de licencias especiales, pero la versión gratuita de Unity nos permite construir diversas opciones para la PC y la Web.

Para compilar, vamos al menú **File** y elegimos la opción **Build Settings**. Se abre una ventana donde vemos un listado; tenemos que verificar que nuestra escena esté seleccionada. Del lado izquierdo está la lista

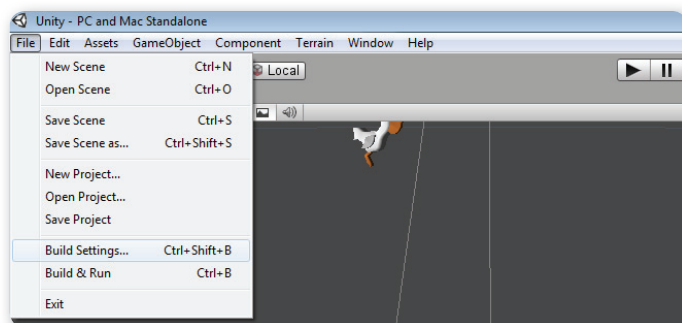
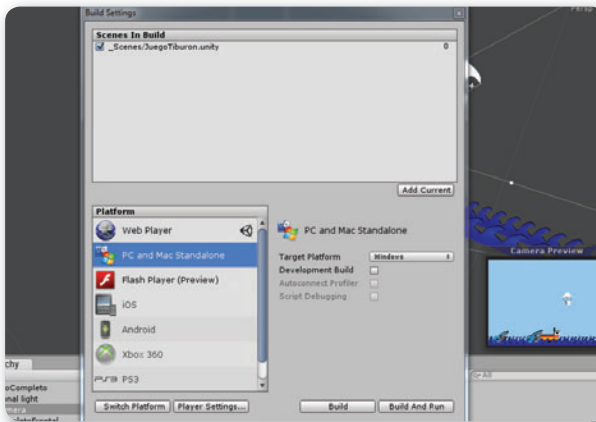


Figura 16. Es importante colocar la configuración antes de construir el juego.

de plataformas, de donde elegimos la opción **PC and Mac Standalone**. Podemos seleccionar la versión de Windows que queramos. Cuando ya esté todo listo, oprimimos el botón **Build**.

Aparecerá un cuadro de diálogo para seleccionar el lugar donde se guardará la compilación. Podemos hacerlo en el mismo folder donde estamos creando el proyecto. Después de la construcción, aparece un ejecutable con ese nombre y una carpeta que también tiene la misma denominación, y donde se han guardado los recursos del juego. Cuando distribuyamos el juego, deberemos entregar tanto el ejecutable como esta carpeta.



► **Figura 17.** Esta ventana nos permite configurar la distribución deseada.

De esta manera, hemos creado una aplicación completa de Unity y ya somos capaces de desarrollar nuestros propios videojuegos.



RESUMEN

Para crear un juego sencillo, debemos planificar cada uno de los objetos y qué es lo que deben hacer. Estos pueden ser modelados y animados en Blender y, posteriormente, importados desde Unity. Lo mejor es darle al objeto todas sus características, incluyendo el script, y después asignarlo a un prefab. Cuando ya tenemos los prefabs listos, procedemos a crear la escena. Un buen lugar para tener el control general de la aplicación es la cámara. Una vez que la aplicación funciona como deseamos, la construimos seleccionando una plataforma en particular.

Actividades

TEST DE AUTOEVALUACIÓN

- 1 ¿Por qué es mejor animar en Blender?
- 2 ¿Cuál es la ventaja de utilizar prefabs?
- 3 ¿Cómo se puede invocar una función de otro objeto?
- 4 ¿Cuál es la importancia de nombrar a los objetos?
- 5 ¿Por qué es bueno tener un script de control general?
- 6 ¿Qué ventaja da colocar un script en la cámara?
- 7 ¿Cómo se puede modificar la tipografía del GUI?
- 8 ¿Qué se necesita para pasar un prefab a una propiedad?
- 9 ¿Cómo se pausa la ejecución de un script?
- 10 ¿Qué pasos se deben dar para el diseño básico de un juego?
- 11 ¿Cómo se publica un juego en Unity?
- 12 ¿Cómo se publica un juego para la Web?

ACTIVIDADES PRÁCTICAS

- 1 Agregue un pelícano más.
- 2 Adicione otro tipo de antagonista para el juego.
- 3 Publique la versión del juego para la Web.
- 4 Mejore el diseño de la interfaz de usuario.
- 5 Coloque un fondo musical al juego.

Índice temático

#	3rd Person Controller 124	F	Efectos preconstruidos 222
A	Acelerómetros 72 Add Texture 140, 143 Add Tree 148 Ambient Light 154, 173 Angular Drag 190 Ángulos de Euler 61, 69 Apertura de campo 109, 124 Audio Source 251		Escalamiento 30, 58 First Person Controller 123 Flatten Heightmap 130, 135
B	Blender 12, 226, 230, 292 Bool 78, 84, 95 Box collider 186, 191, 300 Brush Size 134, 138 Burst 218	G	Game Object 28 GetAxis 93 GetKeyDown() 93 Gizmos 256 Gradient 215 Gravity Modifier 216
C	Cadena 93, 195, 246, 275 Capsule Collider 186 Colliders 185 Collision 195, 202 Color over lifetime 215 Colores de la luz 215 Consola 195 Create Particle System 219 Create terrain 128 Custom Rolloff 255	H	Herramientas de depuración 256 Hierarchy 32
D	Debug 195, 256 Destroy() 200 Diffuse 40, 66, 235 Directional Light 100, 139, 154, 166 Duration 210	I	Iluminación 87, 144, 154 Import Package 122, 144 Inspector 34, 104, 147 Intensity 77, 160
		K	Keyframe 232, 239
		L	Label 272 Listener 250 Location 239 LocRot 239 LocRotScale 239 Looping 210, 217
		M	Main Camera 98, 106 Mass 189 Max Particles 217 Maximize on Play 27

M	Mesh collider	185
	Mipmas	23
	MonoDevelop	18
	Motor de juego	14
	Motores de física	184
	Mover la cámara	119
	Movimiento por cuadro	54
	Movimiento por tiempo	56

N	Navegar la escena	25
	Nighttime Simple Water	144

O	Object Tools	228
	Open Source	226
	Otros dispositivos	90
	Overdraw	23

P	Paneles principales	21
	Parenting	64, 100
	Pipeline	14
	Plane	100
	Plano cercano	111
	Plano lejano	111
	Point Light	154
	Position	48
	Prefabs	144, 263
	Prewarm	211
	Project Settings	73

Q	Quaternion	264
----------	------------------	-----

R	Radius	186
	Random	67
	Range	76
	RayCast	201
	Render Settings	147

R	Rigid Body	188
	Rotation	239
	RotScale	239

S	Scaling	239
	Scene	21, 32
	Set Resolution	130
	Shaders	14
	Sistema de entradas	72
	Sistema de física	16, 184, 216
	Sistema de partículas	206
	Skybox Material	147
	Skyboxes	23
	Slider	284
	Sphere Collider	186
	Sphere Projection	236
	Split Animations	243
	Spot Angle	177
	Spotlight	173
	Start Delay	211

T	Tags	265
	Terrain Height	130
	Terrain Length	130
	Terrain Width	130
	Time.deltaTime	57
	Toggle	287
	Transform	34
	Tree Density	149
	Trim	60

U	Unwrap	236
	Update()	145
	UV Mapping	236

V	Vector3	47
	Volume Rolloff	255

CONÉCTESE CON LOS MEJORES LIBROS DE COMPUTACIÓN



Este manual es ideal para todos aquellos que quieran iniciarse en la programación de microcontroladores. A través de esta obra, podrán aprender sobre los microcontroladores PIC 16F y 18F, hasta llegar a conectar los dispositivos de forma inalámbrica, entre muchos otros proyectos.

- » HARDWARE / ELECTRÓNICA
- » 320 PÁGINAS
- » ISBN 978-987-1773-56-5

LLEGAMOS A TODO EL MUNDO VÍA **OCA** * Y **PdL** **
MÁS INFORMACIÓN / CONTÁCTENOS

usershop.redusers.com +54 (011) 4110-8700 usershop@redusers.com

* SÓLO VÁLIDO EN LA REPÚBLICA ARGENTINA // ** VÁLIDO EN TODO EL MUNDO EXCEPTO ARGENTINA

www.FreeLibros.me



UNITY



Presentamos un libro ideal para todos aquellos que deseen crear sus propios videojuegos en 3D mediante las herramientas de programación que ofrece Unity. A lo largo de la obra, aprenderemos a crear scripts para controlar la lógica de los objetos, y haremos uso del editor para generar escenas, efectos de iluminación y terrenos. También usaremos el motor de física que provee Unity para que el juego sea más realista, y veremos cómo incorporar efectos especiales. Además, sin dejar de lado la importancia del audio y la interacción con el usuario, reforzaremos estos aspectos a través de elementos gráficos que aumenten el atractivo y la funcionalidad de nuestro juego. A través de explicaciones sencillas, el lector dominará el programa volcando todos los conceptos aprendidos en un proyecto integrador que le permitirá construir su propio videojuego.



Unity no es solo un motor de juegos de gran calidad; también nos permite trabajar con diferentes lenguajes de programación.



* EN ESTE LIBRO APRENDERÁ:

- **Conocer Unity:** cómo utilizar el editor, y cuáles son sus diferentes paneles y menús. La adición y manipulación de objetos, y la creación de escenas.
- **Objetos y entradas de usuario:** creación de scripts para manipular objetos, materiales y colores. Entradas para leer el teclado, el mouse y los controladores.
- **Manejo de cámaras:** creación de scripts para manipular y controlar la cámara. Uso de una cámara en primera persona y otra en tercera.
- **Creación de terrenos:** trabajo con el editor de terrenos para crear escenas en exteriores. Desarrollo de texturas con detalles, objetos y plantas.
- **Interfaz de usuario:** generación de una interfaz gráfica de usuario, elementos principales, y creación de scripts para leer y mostrar información.
- **Blender y animaciones:** modelado de personajes, creación de materiales y texturas. Animación de personajes, exportación a Unity y script de control.



» SOBRE EL AUTOR

Nicolás Arrijo Landa Cosío es catedrático de IUNIVERTECH y de la Universidad Madero, en México. Se dedica a la investigación, consulta y capacitación en áreas relacionadas con la realidad virtual, la visualización científica y los videojuegos. Actualmente, realiza una investigación sobre Inteligencia Artificial no-Algorítmica y diseño de hardware.

» NIVEL DE USUARIO

Básico / Intermedio

» CATEGORÍA

Desarrollo

