# 12 Un barco que dispara balas



¡Vamos a construir un juego llamado *Alien Invasion*! Utilizaremos Pygame, una colección de divertidos y potentes módulos de Python que gestionan los gráficos, la animación e incluso el sonido, facilitándote la construcción de juegos sofisticados. Con Pygame manejando tareas como dibujar imágenes en la pantalla, puedes centrarte en la lógica de alto nivel de la dinámica del juego.

En este capítulo, configurarás Pygame y crearás un cohete espacial que se mueve a derecha e izquierda y dispara balas en respuesta a las entradas del jugador. En los dos capítulos siguientes, crearás una flota de alienígenas para destruir, y luego seguirás perfeccionando el juego estableciendo límites al número de naves que puedes utilizar y añadiendo un marcador.

Mientras construyes este juego, también aprenderás a gestionar grandes proyectos que abarcan varios archivos. Refactorizaremos mucho código y gestionaremos el contenido de los archivos para organizar el proyecto y hacer que el código sea eficiente.

Hacer juegos es una forma ideal de divertirse mientras se aprende un lenguaje. Es profundamente satisfactorio jugar a un juego escrito por ti, y escribir un juego sencillo te enseñará mucho sobre cómo desarrollan juegos los profesionales. Mientras trabajas en este capítulo, introduce y ejecuta el código para identificar cómo contribuye cada bloque de código a la jugabilidad general. Experimenta con distintos valores y ajustes para comprender mejor cómo perfeccionar las interacciones en tus juegos.

## Nota

*Alien Invasion* abarca varios archivos diferentes, así que crea una nueva carpeta *alien\_invasion* en tu sistema. Asegúrate de guardar todos los archivos del proyecto en esta carpeta para que tus sentencias import funcionen correctamente.

Además, si te sientes cómodo utilizando el control de versiones, quizás quieras utilizarlo para este proyecto. Si no has utilizado antes el control de versiones, consulta el Apéndice D para obtener una visión general.

## Planificar tu proyecto

Cuando estás construyendo un gran proyecto, es importante preparar un plan antes de empezar a escribir código. Tu plan te mantendrá centrado y hará más probable que completes el proyecto.

Escribamos una descripción de la jugabilidad general. Aunque la siguiente descripción no cubre todos los detalles de *Alien Invasion*, proporciona una idea clara de cómo empezar a construir el juego:

En *Alien Invasion*, el jugador controla una nave cohete que aparece en la parte inferior central de la pantalla. El jugador puede mover la nave a derecha e izquierda con las teclas de flecha y disparar balas con la barra espaciadora. Cuando comienza el juego, una flota de alienígenas llena el cielo y se desplaza por la pantalla. El jugador dispara y destruye a los alienígenas. Si el jugador destruye a todos los alienígenas, aparece una nueva flota que se mueve más rápido que la anterior. Si algún alienígena golpea la nave del jugador o llega al fondo de la pantalla, el jugador pierde una nave. Si el jugador pierde tres naves, el juego termina.

Para la primera fase de desarrollo, haremos una nave que pueda moverse a derecha e izquierda cuando el jugador pulse las teclas de flecha y disparar balas cuando el jugador pulse la barra espaciadora. Después de configurar este comportamiento, podemos crear los alienígenas y perfeccionar la jugabilidad.

## Instalación de Pygame

Antes de empezar a programar, instala Pygame. Lo haremos del mismo modo que instalamos pytest en el Capítulo 11: con pip. Si te has saltado el Capítulo 11 o necesitas un repaso sobre pip, consulta "Instalar pytestcon pip" en la página 210.

Para instalar Pygame, introduce el siguiente comando en una terminal:

$ python -m pip install --user pygame

Si utilizas un comando distinto de python para ejecutar programas o iniciar una sesión de terminal, como python3, asegúrate de utilizar ese comando en su lugar.

## Iniciar el proyecto de juego

Empezaremos a construir el juego creando una ventana Pygame vacía. Más tarde, dibujaremos los elementos del juego, como la nave y los alienígenas, en esta ventana. También haremos que nuestro juego responda a las entradas del usuario, estableceremos el color de fondo y cargaremos una imagen de la nave.

### Crear una ventana Pygame y responder a las entradas del usuario

Crearemos una ventana Pygame vacía creando una clase que represente el juego. En tu editor de texto, crea un nuevo archivo y guárdalo como *alien\_invasion.py*; luego introduce lo siguiente:

**alien\_invasion.py**

import sys  
  
import pygame  
  
class AlienInvasion:  
 """Overall class to manage game assets and behavior."""  
  
 def \_\_init\_\_(self):  
 """Initialize the game, and create game resources."""  
❶ pygame.init()  
  
❷ self.screen = pygame.display.set\_mode((1200, 800))  
 pygame.display.set\_caption("Alien Invasion")  
  
 def run\_game(self):  
 """Start the main loop for the game."""  
❸ while True:  
 # Watch for keyboard and mouse events.  
❹ for event in pygame.event.get():  
❺ if event.type == pygame.QUIT:  
 sys.exit()  
  
 # Make the most recently drawn screen visible.  
❻ pygame.display.flip()  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 # Make a game instance, and run the game.  
 ai = AlienInvasion()  
 ai.run\_game()

En primer lugar, importamos los módulos sys y pygame. El módulo pygame contiene la funcionalidad que necesitamos para hacer un juego. Utilizaremos las herramientas del módulo sys para salir del juego cuando el jugador lo abandone.

*Alien Invasion* comienza como una clase llamada AlienInvasion. En el método \_\_init\_\_(), la función pygame.init() inicializa los ajustes de fondo que Pygame necesita para funcionar correctamente ❶. Luego llamamos a pygame.display.set\_mode() para crear una ventana de visualización ❷, en la que dibujaremos todos los elementos gráficos del juego. El argumento (1200, 800) es una tupla que define las dimensiones de la ventana del juego, que será de 1.200 píxeles de ancho por 800 píxeles de alto. (Puedes ajustar estos valores en función del tamaño de tu pantalla.) Asignamos esta ventana de visualización al atributo self.screen, para que esté disponible en todos los métodos de la clase.

El objeto que asignamos a self.screen se llama superficie. Una *surface* en Pygame es una parte de la pantalla donde se puede mostrar un elemento del juego. Cada elemento del juego, como un alienígena o una nave, es su propia superficie. La superficie devuelta por display.set\_mode() representa toda la ventana del juego. Cuando activemos el bucle de animación del juego, esta superficie se redibujará en cada pasada por el bucle, para que pueda actualizarse con cualquier cambio provocado por la entrada del usuario.

El juego se controla mediante el método run\_game(). Este método contiene un bucle while ❸ que se ejecuta continuamente. El bucle while contiene un bucle de eventos y código que gestiona las actualizaciones de la pantalla. Un *event* es una acción que el usuario realiza mientras juega, como pulsar una tecla o mover el ratón. Para que nuestro programa responda a los eventos, escribimos un bucle *event loop* a *listen* para los eventos y realizar las tareas adecuadas en función del tipo de eventos que se produzcan. El bucle for ❹ anidado dentro del bucle while es un bucle de eventos.

Para acceder a los eventos que Pygame detecta, utilizaremos la función pygame.event.get(). Esta función devuelve una lista de eventos que han tenido lugar desde la última vez que se llamó a esta función. Cualquier evento de teclado o ratón hará que se ejecute este bucle for. Dentro del bucle, escribiremos una serie de sentencias if para detectar y responder a eventos específicos. Por ejemplo, cuando el jugador haga clic en el botón de cierre de la ventana del juego, se detectará un evento pygame.QUIT y llamaremos a sys.exit() para salir del juego ❺.

La llamada a pygame.display.flip() ❻ indica a Pygame que haga visible la pantalla dibujada más recientemente. En este caso, simplemente dibuja una pantalla vacía en cada pasada por el bucle while, borrando la pantalla anterior para que sólo sea visible la nueva. Cuando movemos los elementos del juego, pygame.display.flip() actualiza continuamente la pantalla para mostrar las nuevas posiciones de los elementos del juego y ocultar las antiguas, creando la ilusión de un movimiento suave.

Al final del archivo, creamos una instancia del juego y llamamos a run\_game(). Colocamos run\_game() en un bloque if que sólo se ejecuta si se llama directamente al archivo. Cuando ejecutes este archivo *alien\_invasion.py*, deberías ver una ventana vacía de Pygame.

### Control de la velocidad de fotogramas

Idealmente, los juegos deberían ejecutarse a la misma velocidad, o *frame rate*, en todos los sistemas. Controlar la velocidad de fotogramas de un juego que puede ejecutarse en varios sistemas es una cuestión compleja, pero Pygame ofrece una forma relativamente sencilla de lograr este objetivo. Haremos un reloj, y nos aseguraremos de que el reloj haga tictac una vez en cada pasada por el bucle principal. Cada vez que el bucle procese más rápido que el ritmo que definamos, Pygame calculará el tiempo correcto de pausa para que el juego se ejecute a un ritmo constante.

Definiremos el reloj en el método \_\_init\_\_():

**alien\_invasion.py**

def \_\_init\_\_(self):  
 """Initialize the game, and create game resources."""  
 pygame.init()  
 self.clock = pygame.time.Clock()  
 --snip--

Tras inicializar pygame, crearemos una instancia de la clase Clock, a partir del módulo pygame.time. Luego haremos que el reloj marque el final del bucle while en run\_game():

def run\_game(self):  
 """Start the main loop for the game."""  
 while True:  
 --snip--  
 pygame.display.flip()  
 self.clock.tick(60)

El método tick() toma un argumento: la velocidad de fotogramas del juego. Aquí estoy utilizando un valor de 60, por lo que Pygame hará todo lo posible para que el bucle se ejecute exactamente 60 veces por segundo.

## Nota

El reloj de Pygame debería ayudar a que el juego se ejecute de forma consistente en la mayoría de los sistemas. Si hace que el juego funcione de forma menos consistente en tu sistema, puedes probar con valores diferentes para la velocidad de fotogramas. Si no puedes encontrar una buena velocidad de fotogramas en tu sistema, puedes omitir por completo el reloj y ajustar la configuración del juego para que funcione bien en tu sistema.

### Configurar el color de fondo

Pygame crea una pantalla negra por defecto, pero eso es aburrido. Vamos a establecer un color de fondo diferente. Lo haremos al final del método \_\_init\_\_().

**alien\_invasion.py**

def \_\_init\_\_(self):  
 --snip--  
 pygame.display.set\_caption("Alien Invasion")  
  
 # Set the background color.  
❶ self.bg\_color = (230, 230, 230)  
  
 def run\_game(self):  
 --snip--  
 for event in pygame.event.get():  
 if event.type == pygame.QUIT:  
 sys.exit()  
  
 # Redraw the screen during each pass through the loop.  
❷ self.screen.fill(self.bg\_color)  
  
 # Make the most recently drawn screen visible.  
 pygame.display.flip()  
 self.clock.tick(60)

Los colores en Pygame se especifican como colores RGB: una mezcla de rojo, verde y azul. Cada valor de color puede ir de 0 a 255. El valor de color (255, 0, 0) es rojo, (0, 255, 0) es verde y (0, 0, 255) es azul. Puedes mezclar diferentes valores RGB para crear hasta 16 millones de colores. El valor de color (230, 230, 230)  mezcla cantidades iguales de rojo, azul y verde, lo que produce un color de fondo gris claro. Asignamos este color a self.bg\_color ❶.

Rellenamos la pantalla con el color de fondo utilizando el método fill() ❷, que actúa sobre una superficie y sólo toma un argumento: un color.

### Crear una clase de configuración

Cada vez que introduzcamos una nueva funcionalidad en el juego, normalmente crearemos también algunos ajustes nuevos. En lugar de añadir ajustes por todo el código, escribamos un módulo llamado settings que contenga una clase llamada Settings para almacenar todos estos valores en un solo lugar. Este enfoque nos permite trabajar con un solo objeto settings cada vez que necesitemos acceder a un ajuste individual. Esto también facilita la modificación de la apariencia y el comportamiento del juego a medida que crece nuestro proyecto. Para modificar el juego, cambiaremos los valores relevantes en *settings.py*, que crearemos a continuación, en lugar de buscar diferentes ajustes por todo el proyecto.

Crea un nuevo archivo llamado *settings.py* dentro de tu carpeta *alien\_invasion* , y añade esta clase inicial Settings:

**settings.py**

class Settings:  
 """A class to store all settings for Alien Invasion."""  
  
 def \_\_init\_\_(self):  
 """Initialize the game's settings."""  
 # Screen settings  
 self.screen\_width = 1200  
 self.screen\_height = 800  
 self.bg\_color = (230, 230, 230)

Para crear una instancia de Settings en el proyecto y utilizarla para acceder a nuestra configuración, tenemos que modificar *alien\_invasion.py* del siguiente modo:

**alien\_invasion.py**

--snip--  
import pygame  
  
from settings import Settings  
  
class AlienInvasion:  
 """Overall class to manage game assets and behavior."""  
  
 def \_\_init\_\_(self):  
 """Initialize the game, and create game resources."""  
 pygame.init()  
 self.clock = pygame.time.Clock()  
❶ self.settings = Settings()  
  
❷ self.screen = pygame.display.set\_mode(  
 (self.settings.screen\_width, self.settings.screen\_height))  
 pygame.display.set\_caption("Alien Invasion")  
  
 def run\_game(self):  
 --snip--  
 # Redraw the screen during each pass through the loop.  
❸ self.screen.fill(self.settings.bg\_color)  
  
 # Make the most recently drawn screen visible.  
 pygame.display.flip()  
 self.clock.tick(60)  
--snip--

Importamos Settings al archivo principal del programa. Luego creamos una instancia de Settings y la asignamos a self.settings ❶, después de hacer la llamada a pygame.init(). Cuando creamos una pantalla ❷, utilizamos los atributos screen\_width y screen\_height de self.settings, y luego utilizamos self.settings para acceder también al color de fondo al rellenar la pantalla ❸.

Cuando ejecutes ahora *alien\_invasion.py* aún no verás ningún cambio, porque lo único que hemos hecho es trasladar a otro sitio los ajustes que ya estábamos utilizando. Ahora estamos listos para empezar a añadir nuevos elementos a la pantalla.

## Añadir la imagen de la nave

Vamos a añadir la nave a nuestro juego. Para dibujar la nave del jugador en la pantalla, cargaremos una imagen y luego utilizaremos el método Pygame blit() para dibujar la imagen.

Cuando elijas el material gráfico para tus juegos, asegúrate de prestar atención a las licencias. La forma más segura y barata de empezar es utilizar gráficos con licencia libre que puedas usar y modificar, de un sitio web como [https://opengameart.org.](https://opengameart.org)

Puedes utilizar casi cualquier tipo de archivo de imagen en tu juego, pero es más fácil si utilizas un archivo de mapa de bits (*.bmp*) porque Pygame carga mapas de bits por defecto. Aunque puedes configurar Pygame para que utilice otros tipos de archivo, algunos tipos de archivo dependen de ciertas bibliotecas de imágenes que deben estar instaladas en tu ordenador. La mayoría de las imágenes que encontrarás están en formato *.jpg* o *.png*, pero puedes convertirlas a mapas de bits utilizando herramientas como Photoshop, GIMP y Paint.

Presta especial atención al color de fondo de la imagen que elijas. Intenta encontrar un archivo con un fondo transparente o sólido que puedas sustituir por cualquier color de fondo, utilizando un editor de imágenes. Tus juegos tendrán mejor aspecto si el color de fondo de la imagen coincide con el color de fondo de tu juego. Alternativamente, puedes hacer coincidir el fondo de tu juego con el fondo de la imagen.

Para *Alien Invasion*, puedes utilizar el archivo *ship.bmp* [(Figura 12-1](#figure12-1)), que está disponible en los recursos de este libro en [https://ehmatthes.github.io/pcc\_3e.](https://ehmatthes.github.io/pcc_3e) El color de fondo del archivo coincide con la configuración que estamos utilizando en este proyecto. Crea una carpeta llamada *images* dentro de tu carpeta principal del proyecto *alien\_invasion*. Guarda el archivo *ship.bmp* en la carpeta *images*.

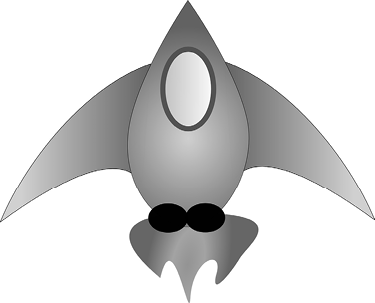


Figura 12-1: La nave para *Alien Invasion*

### Crear la clase Nave

Después de elegir una imagen para la nave, necesitamos mostrarla en la pantalla. Para utilizar nuestra nave, crearemos un nuevo módulo ship que contendrá la clase Ship. Esta clase gestionará la mayor parte del comportamiento de la nave del jugador:

**ship.py**

import pygame  
  
class Ship:  
 """A class to manage the ship."""  
  
 def \_\_init\_\_(self, ai\_game):  
 """Initialize the ship and set its starting position."""  
❶ self.screen = ai\_game.screen  
❷ self.screen\_rect = ai\_game.screen.get\_rect()  
  
 # Load the ship image and get its rect.  
❸ self.image = pygame.image.load('images/ship.bmp')  
 self.rect = self.image.get\_rect()  
  
 # Start each new ship at the bottom center of the screen.  
❹ self.rect.midbottom = self.screen\_rect.midbottom  
  
❺ def blitme(self):  
 """Draw the ship at its current location."""  
 self.screen.blit(self.image, self.rect)

Pygame es eficiente porque te permite tratar todos los elementos del juego como rectángulos (*rects*), aunque no tengan exactamente la forma de rectángulos. Tratar un elemento como un rectángulo es eficiente porque los rectángulos son formas geométricas simples. Cuando Pygame necesita averiguar si dos elementos del juego han colisionado, por ejemplo, puede hacerlo más rápidamente si trata cada objeto como un rectángulo. Este enfoque suele funcionar lo suficientemente bien como para que nadie que juegue a se dé cuenta de que no estamos trabajando con la forma exacta de cada elemento del juego. En esta clase trataremos la nave y la pantalla como rectángulos.

Importamos el módulo pygame antes de definir la clase. El método \_\_init\_\_() de Ship toma dos parámetros: la referencia self y una referencia a la instancia actual de la clase AlienInvasion. Esto dará a Ship acceso a todos los recursos del juego definidos en AlienInvasion. A continuación, asignamos la pantalla a un atributo de Ship ❶, para que podamos acceder a ella fácilmente en todos los métodos de esta clase. Accedemos al atributo rect de la pantalla mediante el método get\_rect() y lo asignamos a self.screen\_rect ❷. Esto nos permite colocar la nave en el lugar correcto de la pantalla.

Para cargar la imagen, llamamos a pygame.image.load() ❸ y le damos la ubicación de la imagen de nuestro barco. Esta función devuelve una superficie que representa el barco, que asignamos a self.image. Cuando la imagen está cargada, llamamos a get\_rect() para acceder al atributo rect de la superficie del barco, de modo que podamos utilizarla posteriormente para colocar el barco.

Cuando trabajes con un objeto rect, puedes utilizar las coordenadas *x*- y *y*- de los bordes superior, inferior, izquierdo y derecho del rectángulo, así como el centro, para colocar el objeto. Puedes fijar cualquiera de estos valores para establecer la posición actual del rect. Cuando estés centrando un elemento del juego, trabaja con los atributos center, centerx o centery de un rect. Cuando estés trabajando en un borde de la pantalla, trabaja con los atributos top, bottom, left o right. También hay atributos que combinan estas propiedades, como midbottom, midtop, midleft, y midright. Cuando estés ajustando la colocación horizontal o vertical de rect, puedes utilizar simplemente los atributos x y y, que son las coordenadas *x*- y *y*- de su esquina superior izquierda. Estos atributos te evitan tener que hacer cálculos que antes tenían que hacer manualmente los desarrolladores de juegos, y los utilizarás a menudo.

## Nota

En Pygame, el origen (0, 0) está en la esquina superior izquierda de la pantalla, y las coordenadas aumentan a medida que vas hacia abajo y hacia la derecha. En una pantalla de 1200×800, el origen está en la esquina superior izquierda, y la esquina inferior derecha tiene las coordenadas (1200, 800). Estas coordenadas se refieren a la ventana del juego, no a la pantalla física.

Posicionaremos la nave en el centro inferior de la pantalla. Para ello, haz que el valor de self.rect.midbottom coincida con el atributo midbottom de la pantalla rect ❹. Pygame utiliza estos atributos rect para posicionar la imagen de la nave de modo que quede centrada horizontalmente y alineada con la parte inferior de la pantalla.

Por último, definimos el método blitme() ❺, que dibuja la imagen en la pantalla en la posición especificada por self.rect.

### Dibujar la nave en la pantalla

Ahora actualicemos *alien\_invasion.py* para que cree una nave y llame al método blitme() de la nave:

**alien\_invasion.py**

--snip--  
from settings import Settings  
from ship import Ship  
  
class AlienInvasion:  
 """Overall class to manage game assets and behavior."""  
  
 def \_\_init\_\_(self):  
 --snip--  
 pygame.display.set\_caption("Alien Invasion")  
  
❶ self.ship = Ship(self)  
  
 def run\_game(self):  
 --snip--  
 # Redraw the screen during each pass through the loop.  
 self.screen.fill(self.settings.bg\_color)  
❷ self.ship.blitme()  
  
 # Make the most recently drawn screen visible.  
 pygame.display.flip()  
 self.clock.tick(60)  
--snip--

Importamos Ship y creamos una instancia de Ship una vez creada la pantalla ❶. La llamada a Ship() requiere un argumento: una instancia de AlienInvasion. El argumento self se refiere aquí a la instancia actual de AlienInvasion. Éste es el parámetro que da acceso a Ship a los recursos del juego, como el objeto screen. Asignamos esta instancia de Ship a self.ship.

Después de rellenar el fondo, dibujamos la nave en la pantalla llamando a ship.blitme(), para que la nave aparezca sobre el fondo ❷.

Cuando ejecutes ahora *alien\_invasion.py*, deberías ver una pantalla de juego vacía con el cohete espacial situado en la parte inferior central, como se muestra en la [Figura 12-2](#figure12-2).

|  |
| --- |
| Figura |



12-2: *Alien Invasion* con la nave

en la parte inferior central de la pantalla

## Refactorización: Los métodos \_check\_events() y \_update\_screen()

En los proyectos grandes, a menudo refactorizarás el código que has escrito antes de añadir más código. La refactorización simplifica la estructura del código que ya has escrito, facilitando su desarrollo. En esta sección, dividiremos el método run\_game(), que se está haciendo largo, en dos métodos de ayuda. Un *helper method* funciona dentro de una clase, pero no está pensado para ser utilizado por código fuera de la clase. En Python, un guión bajo inicial indica un método de ayuda.

### El método \_check\_events()

Trasladaremos el código que gestiona los eventos a un método independiente llamado \_check\_events(). Esto simplificará run\_game() y aislará el bucle de gestión de eventos. Aislar el bucle de eventos te permite gestionar los eventos por separado de otros aspectos del juego, como la actualización de la pantalla.

Aquí tienes la clase AlienInvasion con el nuevo método \_check\_events(), que sólo afecta al código de run\_game():

**alien\_invasion.py**

def run\_game(self):  
 """Start the main loop for the game."""  
 while True:  
❶ self.\_check\_events()  
  
 # Redraw the screen during each pass through the loop.  
 --snip--  
  
❷ def \_check\_events(self):  
 """Respond to keypresses and mouse events."""  
 for event in pygame.event.get():  
 if event.type == pygame.QUIT:  
 sys.exit()

Creamos un nuevo método \_check\_events() ❷ y trasladamos a este nuevo método las líneas que comprueban si el jugador ha hecho clic para cerrar la ventana.

Para llamar a un método desde dentro de una clase, utiliza la notación con puntos con la variable self y el nombre del método ❶. Llamamos al método desde dentro del bucle while en run\_game().

### El método \_actualizar\_pantalla()

Para simplificar aún más run\_game(), trasladaremos el código para actualizar la pantalla a un método independiente llamado \_update\_screen():

**alien\_invasion.py**

def run\_game(self):  
 """Start the main loop for the game."""  
 while True:  
 self.\_check\_events()  
 self.\_update\_screen()  
 self.clock.tick(60)  
  
 def \_check\_events(self):  
 --snip--  
  
 def \_update\_screen(self):  
 """Update images on the screen, and flip to the new screen."""  
 self.screen.fill(self.settings.bg\_color)  
 self.ship.blitme()  
  
 pygame.display.flip()

Trasladamos el código que dibuja el fondo y la nave y voltea la pantalla a \_update\_screen(). Ahora el cuerpo del bucle principal en run\_game() es mucho más sencillo. Es fácil ver que buscamos nuevos eventos, actualizamos la pantalla y hacemos tictac en el reloj en cada pasada por el bucle.

Si ya has construido varios juegos, probablemente empezarás dividiendo tu código en métodos como éstos. Pero si nunca has abordado un proyecto como éste, es probable que al principio no sepas exactamente cómo estructurar tu código. Este enfoque te da una idea de un proceso de desarrollo realista: empiezas escribiendo tu código de la forma más sencilla posible, y luego lo refactorizas a medida que tu proyecto se hace más complejo.

Ahora que hemos reestructurado el código para que sea más fácil añadir cosas, ¡podemos trabajar en los aspectos dinámicos del juego!

## Pruébalo tú mismo

12-1. Cielo azul: Crea una ventana Pygame con un fondo azul.

12-2. Personaje del juego: Encuentra una imagen de mapa de bits de un personaje de juego que te guste o convierte una imagen a mapa de bits. Crea una clase que dibuje el personaje en el centro de la pantalla, luego haz coincidir el color de fondo de la imagen con el color de fondo de la pantalla o viceversa.

## Pilotar la nave

A continuación, daremos al jugador la posibilidad de mover la nave a derecha e izquierda. Escribiremos código que responda cuando el jugador pulse la tecla de flecha derecha o izquierda. Nos centraremos primero en el movimiento hacia la derecha, y luego aplicaremos los mismos principios para controlar el movimiento hacia la izquierda. A medida que vayamos añadiendo este código, aprenderás a controlar el movimiento de las imágenes en la pantalla y a responder a las entradas del usuario.

### Responder a la pulsación de una tecla

Cada vez que el jugador pulsa una tecla, esa pulsación se registra en Pygame como un evento. Cada evento es recogido por el método pygame.event.get(). Tenemos que especificar en nuestro método \_check\_events() qué tipo de eventos queremos que compruebe el juego. Cada pulsación de tecla se registra como un evento KEYDOWN.

Cuando Pygame detecta un evento KEYDOWN, tenemos que comprobar si la tecla que se ha pulsado es una que desencadena una determinada acción. Por ejemplo, si el jugador pulsa la tecla de flecha derecha, queremos aumentar el valor rect.x de la nave para moverla hacia la derecha:

**alien\_invasion.py**

def \_check\_events(self):  
 """Respond to keypresses and mouse events."""  
 for event in pygame.event.get():  
 if event.type == pygame.QUIT:  
 sys.exit()  
❶ elif event.type == pygame.KEYDOWN:  
❷ if event.key == pygame.K\_RIGHT:  
 # Move the ship to the right.  
❸ self.ship.rect.x += 1

Dentro de \_check\_events() añadimos un bloque elif al bucle de eventos, para responder cuando Pygame detecte un evento KEYDOWN ❶. Comprobamos si la tecla pulsada, event.key, es la tecla de la flecha derecha ❷. La tecla de la flecha derecha está representada por pygame.K\_RIGHT. Si se ha pulsado la tecla de la flecha derecha, movemos la nave hacia la derecha aumentando el valor de self.ship.rect.x en 1 ❸.

Cuando ejecutes ahora *alien\_invasion.py*, la nave debería moverse hacia la derecha un píxel cada vez que pulses la tecla de flecha derecha. Es un comienzo, pero no es una forma eficaz de controlar la nave. Mejoremos este control permitiendo un movimiento continuo.

### Permitir el movimiento continuo

Cuando el jugador mantenga pulsada la tecla de flecha derecha, queremos que la nave siga moviéndose hasta que el jugador suelte la tecla. Haremos que el juego detecte un evento pygame.KEYUP para saber cuándo se suelta la tecla de la flecha derecha; luego utilizaremos los eventos KEYDOWN y KEYUP junto con una bandera llamada moving\_right para implementar el movimiento continuo.

Cuando la bandera moving\_right sea False, la nave estará inmóvil. Cuando el jugador pulse la tecla de flecha derecha, pondremos la bandera a True, y cuando el jugador suelte la tecla, volveremos a poner la bandera a False.

La clase Ship controla todos los atributos de la nave, así que le daremos un atributo llamado moving\_right y un método update() para comprobar el estado de la bandera moving\_right. El método update() cambiará la posición de la nave si la bandera se establece en True. Llamaremos a este método una vez en cada pasada por el bucle while para actualizar la posición de la nave.

Estos son los cambios en Ship:

**ship.py**

class Ship:  
 """A class to manage the ship."""  
  
 def \_\_init\_\_(self, ai\_game):  
 --snip--  
 # Start each new ship at the bottom center of the screen.  
 self.rect.midbottom = self.screen\_rect.midbottom  
  
 # Movement flag; start with a ship that's not moving.  
❶ self.moving\_right = False  
  
❷ def update(self):  
 """Update the ship's position based on the movement flag."""  
 if self.moving\_right:  
 self.rect.x += 1  
  
 def blitme(self):  
 --snip--

Añadimos un atributo self.moving\_right en el método \_\_init\_\_() y lo fijamos inicialmente en False ❶. Después añadimos update(), que mueve el barco a la derecha si la bandera es True ❷. El método update() será llamado desde fuera de la clase, por lo que no se considera un método ayudante.

Ahora tenemos que modificar \_check\_events() para que moving\_right se convierta en True cuando se pulse la tecla de flecha derecha y en False cuando se suelte la tecla:

**alien\_invasion.py**

def \_check\_events(self):  
 """Respond to keypresses and mouse events."""  
 for event in pygame.event.get():  
 --snip--  
 elif event.type == pygame.KEYDOWN:  
 if event.key == pygame.K\_RIGHT:  
❶ self.ship.moving\_right = True  
❷ elif event.type == pygame.KEYUP:  
 if event.key == pygame.K\_RIGHT:  
 self.ship.moving\_right = False

Aquí modificamos cómo responde el juego cuando el jugador pulsa la tecla de flecha derecha: en lugar de cambiar directamente la posición de la nave, nos limitamos a establecer moving\_right en True ❶. Luego añadimos un nuevo bloque elif, que responde a los eventos KEYUP ❷. Cuando el jugador suelta la tecla de flecha derecha (K\_RIGHT), establecemos moving\_right en False.

A continuación, modificamos el bucle while en run\_game() para que llame al método update() de la nave en cada pasada por el bucle:

**alien\_invasion.py**

def run\_game(self):  
 """Start the main loop for the game."""  
 while True:  
 self.\_check\_events()  
 self.ship.update()  
 self.\_update\_screen()  
 self.clock.tick(60)

La posición de la nave se actualizará después de comprobar los eventos del teclado y antes de actualizar la pantalla. Esto permite que la posición de la nave se actualice en respuesta a la entrada del jugador y garantiza que la posición actualizada se utilizará al dibujar la nave en la pantalla.

Cuando ejecutes *alien\_invasion.py* y mantengas pulsada la tecla de flecha derecha, la nave debería moverse continuamente hacia la derecha hasta que sueltes la tecla.

### Moverse tanto a la izquierda como a la derecha

Ahora que la nave puede moverse continuamente hacia la derecha, añadir el movimiento hacia la izquierda es sencillo. De nuevo, modificaremos la clase Ship y el método \_check\_events(). Aquí tienes los cambios pertinentes en \_\_init\_\_() y update() en Ship:

**ship.py**

def \_\_init\_\_(self, ai\_game):  
 --snip--  
 # Movement flags; start with a ship that's not moving.  
 self.moving\_right = False  
 self.moving\_left = False  
  
 def update(self):  
 """Update the ship's position based on movement flags."""  
 if self.moving\_right:  
 self.rect.x += 1  
 if self.moving\_left:  
 self.rect.x -= 1

En \_\_init\_\_(), añadimos una bandera self.moving\_left. En update(), utilizamos dos bloques if separados, en lugar de un elif, para permitir que el valor rect.x de la nave aumente y luego disminuya cuando se mantienen pulsadas ambas teclas de flecha. El resultado es que la nave se queda quieta. Si utilizáramos elif para el movimiento hacia la izquierda, la tecla de flecha derecha siempre tendría prioridad. Utilizar dos bloques if hace que los movimientos sean más precisos cuando el jugador puede mantener pulsadas momentáneamente ambas teclas al cambiar de dirección.

Tenemos que hacer dos añadidos a \_check\_events():

**alien\_invasion.py**

def \_check\_events(self):  
 """Respond to keypresses and mouse events."""  
 for event in pygame.event.get():  
 --snip--  
 elif event.type == pygame.KEYDOWN:  
 if event.key == pygame.K\_RIGHT:  
 self.ship.moving\_right = True  
 elif event.key == pygame.K\_LEFT:  
 self.ship.moving\_left = True  
  
 elif event.type == pygame.KEYUP:  
 if event.key == pygame.K\_RIGHT:  
 self.ship.moving\_right = False  
 elif event.key == pygame.K\_LEFT:  
 self.ship.moving\_left = False

Si se produce un evento KEYDOWN para la tecla K\_LEFT, establecemos moving\_left en True. Si se produce un evento KEYUP para la tecla K\_LEFT, establecemos moving\_left en False. Aquí podemos utilizar bloques elif porque cada evento está conectado a una sola tecla. Si el jugador pulsa ambas teclas a la vez, se detectarán dos eventos distintos.

Cuando ejecutes ahora *alien\_invasion.py*, deberías poder mover la nave continuamente a derecha e izquierda. Si mantienes pulsadas ambas teclas, la nave debería dejar de moverse.

A continuación, perfeccionaremos aún más el movimiento de la nave. Vamos a ajustar la velocidad de la nave y a limitar la distancia a la que puede moverse para que no desaparezca por los lados de la pantalla.

### Ajustar la velocidad de la nave

Actualmente, la nave se mueve un píxel por ciclo a través del bucle while, pero podemos tener un control más preciso de la velocidad de la nave añadiendo un atributo ship\_speed a la clase Settings. Utilizaremos este atributo para determinar la distancia que debe recorrer la nave en cada pasada por el bucle. Aquí tienes el nuevo atributo en *settings.py*:

**settings.py**

class Settings:  
 """A class to store all settings for Alien Invasion."""  
  
 def \_\_init\_\_(self):  
 --snip--  
  
 # Ship settings  
 self.ship\_speed = 1.5

Fijamos el valor inicial de ship\_speed en 1.5. Cuando la nave se mueve ahora, su posición se ajusta 1,5 píxeles (en lugar de 1 píxel) en cada pasada por el bucle.

Estamos utilizando un flotador para el ajuste de velocidad, para tener un control más preciso de la velocidad de la nave cuando aumentemos el ritmo del juego más adelante. Sin embargo, los atributos de rect como x sólo almacenan valores enteros, así que tenemos que hacer algunas modificaciones en Ship:

**ship.py**

class Ship:  
 """A class to manage the ship."""  
  
 def \_\_init\_\_(self, ai\_game):  
 """Initialize the ship and set its starting position."""  
 self.screen = ai\_game.screen  
❶ self.settings = ai\_game.settings  
 --snip--  
  
 # Start each new ship at the bottom center of the screen.  
 self.rect.midbottom = self.screen\_rect.midbottom  
  
 # Store a float for the ship's exact horizontal position.  
❷ self.x = float(self.rect.x)  
  
 # Movement flags; start with a ship that's not moving.  
 self.moving\_right = False  
 self.moving\_left = False  
  
 def update(self):  
 """Update the ship's position based on movement flags."""  
 # Update the ship's x value, not the rect.  
 if self.moving\_right:  
❸ self.x += self.settings.ship\_speed  
 if self.moving\_left:  
 self.x -= self.settings.ship\_speed  
  
 # Update rect object from self.x.  
❹ self.rect.x = self.x  
  
 def blitme(self):  
 --snip--

Creamos un atributo settings para Ship, de modo que podamos utilizarlo en update() ❶. Como estamos ajustando la posición de la nave en fracciones de píxel, necesitamos asignar la posición a una variable a la que se le pueda asignar un flotador. Puedes utilizar un flotador para ajustar un atributo de rect, pero rect sólo conservará la parte entera de ese valor. Para hacer un seguimiento preciso de la posición del barco, definimos un nuevo self.x ❷. Utilizamos la función float() para convertir el valor de self.rect.x en un flotador y asignamos este valor a self.x.

Ahora, cuando cambiamos la posición del barco en update(), el valor de self.x se ajusta en la cantidad almacenada en settings.ship\_speed ❸. Una vez actualizado self.x, utilizamos el nuevo valor para actualizar self.rect.x, que controla la posición del barco ❹. Sólo la parte entera de self.x se asignará a self.rect.x, pero eso está bien para mostrar la nave.

Ahora podemos cambiar el valor de ship\_speed, y cualquier valor mayor que 1 hará que la nave se mueva más rápido. Esto ayudará a que la nave responda con suficiente rapidez para derribar alienígenas, y nos permitirá cambiar el tempo del juego a medida que el jugador progresa en la partida.

### Limitar el alcance de la nave

En este punto, la nave desaparecerá de cualquiera de los bordes de la pantalla si mantienes pulsada una tecla de flecha el tiempo suficiente. Vamos a corregir esto para que la nave deje de moverse cuando llegue al borde de la pantalla. Lo haremos modificando el método update() de Ship:

**ship.py**

def update(self):  
 """Update the ship's position based on movement flags."""  
 # Update the ship's x value, not the rect.  
❶ if self.moving\_right and self.rect.right < self.screen\_rect.right:  
 self.x += self.settings.ship\_speed  
❷ if self.moving\_left and self.rect.left > 0:  
 self.x -= self.settings.ship\_speed  
  
 # Update rect object from self.x.  
 self.rect.x = self.x

Este código comprueba la posición de la nave antes de cambiar el valor de self.x. El código self.rect.right devuelve la *x*-coordenada del borde derecho de la nave rect. Si este valor es menor que el valor devuelto por self.screen\_rect.right, la nave no ha alcanzado el borde derecho de la pantalla ❶. Lo mismo ocurre con el borde izquierdo: si el valor del lado izquierdo de rect es mayor que 0, la nave no ha alcanzado el borde izquierdo de la pantalla ❷. Así te aseguras de que la nave está dentro de esos límites antes de ajustar el valor de self.x.

Cuando ejecutes ahora *alien\_invasion.py*, la nave debería dejar de moverse en cualquiera de los bordes de la pantalla. Esto está muy bien; todo lo que hemos hecho es añadir una prueba condicional en una sentencia if, ¡pero parece como si la nave chocara contra una pared o un campo de fuerza en cualquiera de los bordes de la pantalla!

### Refactorización de \_check\_events()

El método \_check\_events() aumentará su longitud a medida que sigamos desarrollando el juego, así que vamos a dividir \_check\_events() en dos métodos distintos: uno que se ocupe de los eventos de KEYDOWN y otro que se ocupe de los eventos de KEYUP:

**alien\_invasion.py**

def \_check\_events(self):  
 """Respond to keypresses and mouse events."""  
 for event in pygame.event.get():  
 if event.type == pygame.QUIT:  
 sys.exit()  
 elif event.type == pygame.KEYDOWN:  
 self.\_check\_keydown\_events(event)  
 elif event.type == pygame.KEYUP:  
 self.\_check\_keyup\_events(event)  
  
 def \_check\_keydown\_events(self, event):  
 """Respond to keypresses."""  
 if event.key == pygame.K\_RIGHT:  
 self.ship.moving\_right = True  
 elif event.key == pygame.K\_LEFT:  
 self.ship.moving\_left = True  
  
 def \_check\_keyup\_events(self, event):  
 """Respond to key releases."""  
 if event.key == pygame.K\_RIGHT:  
 self.ship.moving\_right = False  
 elif event.key == pygame.K\_LEFT:  
 self.ship.moving\_left = False

Creamos dos nuevos métodos ayudantes: \_check\_keydown\_events() y \_check\_keyup\_events(). Cada uno necesita un parámetro self y un parámetro event. Los cuerpos de estos dos métodos están copiados de \_check\_events(), y hemos sustituido el código antiguo por llamadas a los nuevos métodos. El método \_check\_events() es más sencillo ahora con esta estructura de código más limpia, lo que facilitará el desarrollo de otras respuestas a la entrada del jugador.

### Pulsar Q para salir

Ahora que respondemos eficazmente a las pulsaciones de teclas, podemos añadir otra forma de salir del juego. Resulta tedioso hacer clic en la X de la parte superior de la ventana del juego para terminar la partida cada vez que pruebas una nueva función, así que añadiremos un atajo de teclado para terminar la partida cuando el jugador pulse Q:

**alien\_invasion.py**

def \_check\_keydown\_events(self, event):  
 --snip--  
 elif event.key == pygame.K\_LEFT:  
 self.ship.moving\_left = True  
 elif event.key == pygame.K\_q:  
 sys.exit()

En \_check\_keydown\_events(), añadimos un nuevo bloque que finaliza el juego cuando el jugador pulsa Q. Ahora, al probar, puedes pulsar Q para cerrar el juego en lugar de utilizar el cursor para cerrar la ventana.

### Ejecutar el juego en modo pantalla completa

Pygame tiene un modo de pantalla completa que puede gustarte más que ejecutar el juego en una ventana normal. Algunos juegos se ven mejor en el modo de pantalla completa, y en algunos sistemas, el juego puede funcionar mejor en general en el modo de pantalla completa.

Para ejecutar el juego en modo de pantalla completa, realiza los siguientes cambios en \_\_init\_\_():

**alien\_invasion.py**

def \_\_init\_\_(self):  
 """Initialize the game, and create game resources."""  
 pygame.init()  
 self.settings = Settings()  
  
❶ self.screen = pygame.display.set\_mode((0, 0), pygame.FULLSCREEN)  
❷ self.settings.screen\_width = self.screen.get\_rect().width  
 self.settings.screen\_height = self.screen.get\_rect().height  
 pygame.display.set\_caption("Alien Invasion")

Al crear la superficie de la pantalla, pasamos un tamaño de (0, 0) y el parámetro pygame.FULLSCREEN ❶. Esto le dice a Pygame que calcule un tamaño de ventana que llene la pantalla. Como no sabemos de antemano la anchura y la altura de la pantalla, actualizamos estos parámetros después de crear la pantalla ❷. Utilizamos los atributos width y height del objeto rect de la pantalla para actualizar el objeto settings.

Si te gusta cómo se ve o se comporta el juego en modo pantalla completa, mantén estos ajustes. Si te gusta más el juego en su propia ventana, puedes volver al planteamiento original en el que fijamos un tamaño de pantalla específico para el juego.

## Nota

Asegúrate de que puedes salir pulsando Q antes de ejecutar el juego en modo de pantalla completa; Pygame no ofrece ninguna forma predeterminada de salir de un juego en modo de pantalla completa.

## Resumen rápido

En la siguiente sección, añadiremos la posibilidad de disparar balas, lo que implica añadir un nuevo archivo llamado *bullet.py* y hacer algunas modificaciones en algunos de los archivos que ya estamos utilizando. Ahora mismo, tenemos tres archivos que contienen una serie de clases y métodos. Para tener claro cómo está organizado el proyecto, vamos a revisar cada uno de estos archivos antes de añadir más funcionalidad.

### alien\_invasion.py

El archivo principal, *alien\_invasion.py*, contiene la clase AlienInvasion. Esta clase crea una serie de atributos importantes que se utilizan a lo largo del juego: los ajustes se asignan a settings, la superficie de visualización principal se asigna a screen, y también se crea una instancia de ship en este archivo. El bucle principal del juego, un bucle while, también se almacena en este módulo. El bucle while llama a \_check\_events(), ship.update(), y \_update\_screen(). También marca el reloj en cada pasada por el bucle.

El método \_check\_events() detecta eventos relevantes, como pulsaciones de teclas y desbloqueos, y procesa cada uno de estos tipos de eventos a través de los métodos \_check\_keydown\_events() y \_check\_keyup\_events(). Por ahora, estos métodos gestionan el movimiento de la nave. La clase AlienInvasion también contiene \_update\_screen(), que redibuja la pantalla en cada pasada por el bucle principal.

El archivo *alien\_invasion.py* es el único que necesitas ejecutar cuando quieras jugar a *Alien Invasion*. Los otros archivos, *settings.py* y *ship.py*, contienen código que se importa en este archivo.

### ajustes.py

El archivo *settings.py* contiene la clase Settings. Esta clase sólo tiene un método \_\_init\_\_(), que inicializa los atributos que controlan la apariencia del juego y la velocidad de la nave.

### nave.py

El archivo *ship.py* contiene la clase Ship. La clase Ship tiene un método \_\_init\_\_(), un método update() para gestionar la posición de la nave, y un método blitme() para dibujar la nave en la pantalla. La imagen de la nave se almacena en *ship.bmp*, que está en la carpeta *images*.

## Pruébalo tú mismo

12-3. Documentación de Pygame: Ya hemos avanzado lo suficiente en el juego como para que quieras echar un vistazo a parte de la documentación de Pygame. La página de inicio de Pygame está en [https://pygame.org,](https://pygame.org) y la página de inicio de la documentación está en [https://pygame.org/docs.](https://pygame.org/docs) De momento, hojea la documentación. No la necesitarás para completar este proyecto, pero te ayudará si después quieres modificar *Alien Invasion* o crear tu propio juego.

12-4. Cohete: Haz un juego que comience con un cohete en el centro de la pantalla. Permite al jugador mover el cohete hacia arriba, abajo, izquierda o derecha utilizando las cuatro teclas de flecha. Asegúrate de que el cohete nunca se mueve más allá de ningún borde de la pantalla.

12-5. Teclas: Crea un archivo Pygame que cree una pantalla vacía. En el bucle de eventos, imprime el atributo event.key cada vez que se detecte un evento pygame.KEYDOWN. Ejecuta el programa y pulsa varias teclas para ver cómo responde Pygame.

## Disparando balas

Ahora vamos a añadir la posibilidad de disparar balas. Escribiremos un código que dispare una bala, representada por un pequeño rectángulo, cuando el jugador pulse la barra espaciadora. Las balas se desplazarán en línea recta por la pantalla hasta que desaparezcan de la parte superior de la pantalla.

### Añadir la configuración de las balas

Al final del método \_\_init\_\_(), actualizaremos *settings.py* para incluir los valores que necesitaremos para una nueva clase Bullet:

**settings.py**

def \_\_init\_\_(self):  
 --snip--  
 # Bullet settings  
 self.bullet\_speed = 2.0  
 self.bullet\_width = 3  
 self.bullet\_height = 15  
 self.bullet\_color = (60, 60, 60)

Estos valores crean balas de color gris oscuro con una anchura de 3 píxeles y una altura de 15 píxeles. Las balas se desplazarán ligeramente más rápido que la nave.

### Creación de la clase Bala

Ahora crea un archivo *bullet.py* para almacenar nuestra clase Bullet. Aquí tienes la primera parte de *bullet.py*:

**bullet.py**

import pygame  
from pygame.sprite import Sprite  
  
class Bullet(Sprite):  
 """A class to manage bullets fired from the ship."""  
  
 def \_\_init\_\_(self, ai\_game):  
 """Create a bullet object at the ship's current position."""  
 super().\_\_init\_\_()  
 self.screen = ai\_game.screen  
 self.settings = ai\_game.settings  
 self.color = self.settings.bullet\_color  
  
 # Create a bullet rect at (0, 0) and then set correct position.  
❶ self.rect = pygame.Rect(0, 0, self.settings.bullet\_width,  
 self.settings.bullet\_height)  
❷ self.rect.midtop = ai\_game.ship.rect.midtop  
  
 # Store the bullet's position as a float.  
❸ self.y = float(self.rect.y)

La clase Bullet hereda de Sprite, que importamos del módulo pygame.sprite. Cuando utilizas sprites, puedes agrupar elementos relacionados en tu juego y actuar sobre todos los elementos agrupados a la vez. Para crear una instancia de bala, \_\_init\_\_() necesita la instancia actual de AlienInvasion, y llamamos a super() para heredar correctamente de Sprite. También establecemos atributos para los objetos pantalla y ajustes, y para el color de la bala.

A continuación creamos el atributo rect ❶ de la bala . La bala no se basa en una imagen, así que tenemos que construir una rect desde cero utilizando la clase pygame.Rect(). Esta clase necesita las coordenadas *x*- y *y*- de la esquina superior izquierda del rect, y la anchura y altura del rect. Inicializamos el rect en (0, 0), pero lo moveremos a la ubicación correcta en la línea siguiente, porque la posición de la bala depende de la posición del barco. Obtenemos la anchura y la altura de la bala a partir de los valores almacenados en self.settings.

Ajustamos el atributo midtop de la bala para que coincida con el atributo midtop ❷ del barco . Esto hará que la bala emerja de la parte superior de la nave, haciendo que parezca que la bala se dispara desde la nave. Utilizamos un flotador para la coordenada *y* de la bala, de modo que podamos realizar ajustes finos en la velocidad de la bala ❸.

Aquí tienes la segunda parte de *bullet.py*, update() y draw\_bullet():

**bullet.py**

def update(self):  
 """Move the bullet up the screen."""  
 # Update the exact position of the bullet.  
❶ self.y -= self.settings.bullet\_speed  
 # Update the rect position.  
❷ self.rect.y = self.y  
  
 def draw\_bullet(self):  
 """Draw the bullet to the screen."""  
❸ pygame.draw.rect(self.screen, self.color, self.rect)

El método update() gestiona la posición de la bala. Cuando se dispara una bala, se desplaza hacia arriba en la pantalla, lo que corresponde a un valor de coordenada *y* decreciente. Para actualizar la posición, restamos la cantidad almacenada en settings.bullet\_speed de self.y ❶. A continuación, utilizamos el valor de self.y para fijar el valor de self.rect.y ❷.

El ajuste de bullet\_speed nos permite aumentar la velocidad de las balas a medida que avanza el juego o según sea necesario para perfeccionar su comportamiento. Una vez disparada una bala, nunca cambiamos el valor de su coordenada *x*, por lo que viajará verticalmente en línea recta aunque la nave se mueva.

Cuando queremos dibujar una bala, llamamos a draw\_bullet(). La función draw.rect() rellena la parte de la pantalla definida por la rect de la bala con el color almacenado en self.color ❸.

### Almacenar viñetas en un grupo

Ahora que tenemos una clase Bullet y los ajustes necesarios definidos, podemos escribir código para disparar una bala cada vez que el jugador pulse la barra espaciadora. Crearemos un grupo en AlienInvasion para almacenar todas las balas activas, de modo que podamos gestionar las balas que ya han sido disparadas. Este grupo será una instancia de la clase pygame.sprite.Group, que se comporta como una lista con algunas funciones adicionales que resultan útiles cuando se construyen juegos. Utilizaremos este grupo para dibujar las balas en la pantalla en cada pasada por el bucle principal y para actualizar la posición de cada bala.

En primer lugar, importaremos la nueva clase Bullet:

**alien\_invasion.py**

--snip--  
from ship import Ship  
from bullet import Bullet

A continuación crearemos el grupo que contiene las viñetas en \_\_init\_\_():

**alien\_invasion.py**

def \_\_init\_\_(self):  
 --snip--  
 self.ship = Ship(self)  
 self.bullets = pygame.sprite.Group()

Luego tendremos que actualizar la posición de las viñetas en cada pasada por el bucle while:

**alien\_invasion.py**

def run\_game(self):  
 """Start the main loop for the game."""  
 while True:  
 self.\_check\_events()  
 self.ship.update()  
 self.bullets.update()  
 self.\_update\_screen()  
 self.clock.tick(60)

Cuando llamas a update() en un grupo, el grupo llama automáticamente a update() para cada sprite del grupo. La línea self.bullets.update() llama a bullet.update() por cada bala que colocamos en el grupo bullets.

### Disparar balas

En AlienInvasion, tenemos que modificar \_check\_keydown\_events() para disparar una bala cuando el jugador pulse la barra espaciadora. No necesitamos modificar \_check\_keyup\_events() porque no ocurre nada cuando se suelta la barra espaciadora. También tenemos que modificar \_update\_screen() para asegurarnos de que cada bala se dibuja en la pantalla antes de llamar a flip().

Habrá un poco de trabajo que hacer cuando disparemos una bala, así que vamos a escribir un nuevo método, \_fire\_bullet(), para manejar este trabajo:

**alien\_invasion.py**

def \_check\_keydown\_events(self, event):  
 --snip--  
 elif event.key == pygame.K\_q:  
 sys.exit()  
❶ elif event.key == pygame.K\_SPACE:  
 self.\_fire\_bullet()  
  
 def \_check\_keyup\_events(self, event):  
 --snip--  
  
 def \_fire\_bullet(self):  
 """Create a new bullet and add it to the bullets group."""  
❷ new\_bullet = Bullet(self)  
❸ self.bullets.add(new\_bullet)  
  
 def \_update\_screen(self):  
 """Update images on the screen, and flip to the new screen."""  
 self.screen.fill(self.settings.bg\_color)  
❹ for bullet in self.bullets.sprites():  
 bullet.draw\_bullet()  
 self.ship.blitme()  
  
 pygame.display.flip()  
--snip--

Llamamos a \_fire\_bullet() cuando se pulsa la barra espaciadora ❶. En \_fire\_bullet(), creamos una instancia de Bullet y la llamamos new\_bullet ❷. Luego lo añadimos al grupo bullets utilizando el método add() ❸. El método add() es similar a append(), pero está escrito específicamente para grupos Pygame.

El método bullets.sprites() devuelve una lista de todos los sprites del grupo bullets. Para dibujar todas las balas disparadas en la pantalla, hacemos un bucle a través de los sprites en bullets y llamamos a draw\_bullet() en cada uno de ellos ❹. Colocamos este bucle antes de la línea que dibuja la nave, para que las balas no empiecen encima de la nave.

Cuando ejecutes ahora *alien\_invasion.py*, deberías poder mover la nave a derecha e izquierda y disparar tantas balas como quieras. Las balas suben por la pantalla y desaparecen cuando llegan arriba, como se muestra en la [Figura 12-3](#figure12-3). Puedes modificar el tamaño, el color y la velocidad de las balas en *settings.py*.

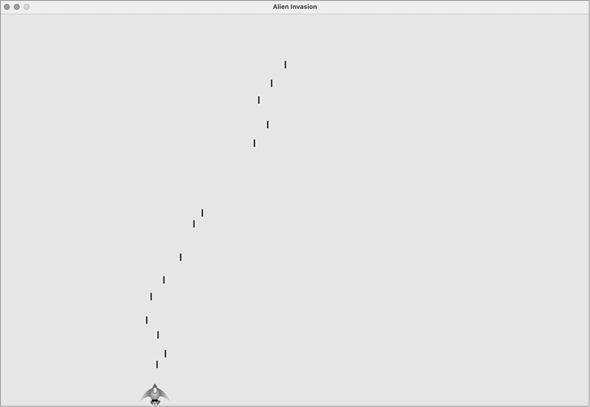


Figura12-3: La nave después de disparar una serie de

balas

### Borrar balas antiguas

De momento, las balas desaparecen cuando llegan arriba, pero sólo porque Pygame no puede dibujarlas por encima de la parte superior de la pantalla. En realidad, las balas siguen existiendo, sólo que sus valores de coordenadas *y* son cada vez más negativos. Esto es un problema porque siguen consumiendo memoria y capacidad de proceso.

Tenemos que deshacernos de estas viejas balas, o el juego se ralentizará por hacer tanto trabajo innecesario. Para ello, necesitamos detectar cuándo el valor bottom de la rect de una bala tiene un valor de 0, lo que indica que la bala ha pasado de la parte superior de la pantalla:

**alien\_invasion.py**

def run\_game(self):  
 """Start the main loop for the game."""  
 while True:  
 self.\_check\_events()  
 self.ship.update()  
 self.bullets.update()  
  
 # Get rid of bullets that have disappeared.  
❶ for bullet in self.bullets.copy():  
❷ if bullet.rect.bottom <= 0:  
❸ self.bullets.remove(bullet)  
❹ print(len(self.bullets))  
  
 self.\_update\_screen()  
 self.clock.tick(60)

Cuando utilizas un bucle for con una lista (o un grupo en Pygame), Python espera que la lista mantenga la misma longitud mientras se ejecute el bucle. Eso significa que no puedes eliminar elementos de una lista o grupo dentro de un bucle for, así que tenemos que hacer un bucle sobre una copia del grupo. Utilizamos el método copy() para configurar el bucle for ❶, lo que nos deja libertad para modificar el grupo original bullets dentro del bucle. Comprobamos cada viñeta para ver si ha desaparecido de la parte superior de la pantalla ❷. Si es así, la eliminamos de bullets ❸. Insertamos una llamada a print() para mostrar cuántas balas existen actualmente en el juego y comprobamos que se eliminan cuando llegan a la parte superior de la pantalla ❹.

Si este código funciona correctamente, podemos observar la salida del terminal mientras disparamos balas y ver que el número de balas disminuye a cero después de que cada serie de balas haya despejado la parte superior de la pantalla. Después de ejecutar el juego y comprobar que las balas se eliminan correctamente, elimina la llamada a print(). Si la dejas, el juego se ralentizará significativamente porque se tarda más tiempo en escribir la salida en el terminal que en dibujar los gráficos en la ventana del juego.

### Limitar el número de balas

Muchos juegos de disparos limitan el número de balas que un jugador puede tener en la pantalla a la vez; al hacerlo, animan a los jugadores a disparar con precisión. Haremos lo mismo en *Alien Invasion*.

Primero, almacena el número de balas permitidas en *settings.py*:

**settings.py**

# Bullet settings  
 --snip--  
 self.bullet\_color = (60, 60, 60)  
 self.bullets\_allowed = 3

Esto limita al jugador a tres balas a la vez. Utilizaremos este ajuste en AlienInvasion para comprobar cuántas balas existen antes de crear una nueva bala en \_fire\_bullet():

**alien\_invasion.py**

def \_fire\_bullet(self):  
 """Create a new bullet and add it to the bullets group."""  
 if len(self.bullets) < self.settings.bullets\_allowed:  
 new\_bullet = Bullet(self)  
 self.bullets.add(new\_bullet)

Cuando el jugador pulsa la barra espaciadora, comprobamos la longitud de bullets. Si len(self.bullets) es inferior a tres, creamos una nueva bala. Pero si ya hay tres balas activas, no ocurre nada cuando se pulsa la barra espaciadora. Cuando ejecutes el juego ahora, sólo deberías poder disparar balas en grupos de tres.

### Creación del método \_update\_bullets()

Queremos mantener la clase AlienInvasion razonablemente bien organizada, así que ahora que hemos escrito y comprobado el código de gestión de las balas, podemos trasladarlo a un método aparte. Crearemos un nuevo método llamado \_update\_bullets() y lo añadiremos justo antes de \_update\_screen():

**alien\_invasion.py**

def \_update\_bullets(self):  
 """Update position of bullets and get rid of old bullets."""  
 # Update bullet positions.  
 self.bullets.update()  
  
 # Get rid of bullets that have disappeared.  
 for bullet in self.bullets.copy():  
 if bullet.rect.bottom <= 0:  
 self.bullets.remove(bullet)

El código de \_update\_bullets() está cortado y pegado de run\_game(); lo único que hemos hecho aquí es aclarar los comentarios.

El bucle while de run\_game() vuelve a parecer sencillo:

**alien\_invasion.py**

while True:  
 self.\_check\_events()  
 self.ship.update()  
 self.\_update\_bullets()  
 self.\_update\_screen()  
 self.clock.tick(60)

Ahora nuestro bucle principal sólo contiene un código mínimo, por lo que podemos leer rápidamente los nombres de los métodos y comprender lo que ocurre en el juego. El bucle principal comprueba la entrada del jugador, y luego actualiza la posición de la nave y las balas que se hayan disparado. A continuación, utilizamos las posiciones actualizadas para dibujar una nueva pantalla y marcar el reloj al final de cada pasada por el bucle.

Ejecuta *alien\_invasion.py* una vez más, y asegúrate de que aún puedes disparar balas sin errores.

## Pruébalo tú mismo

12-6. Tirador lateral: Escribe un juego que coloque una nave en el lado izquierdo de la pantalla y permita al jugador mover la nave arriba y abajo. Haz que la nave dispare una bala que recorra la pantalla hacia la derecha cuando el jugador pulse la barra espaciadora. Asegúrate de que las balas se borran cuando desaparecen de la pantalla.

## Resumen

En este capítulo, aprendiste a hacer un plan para un juego y conociste la estructura básica de un juego escrito en Pygame. Aprendiste a establecer un color de fondo y a almacenar los ajustes en una clase separada, donde puedes ajustarlos más fácilmente. Viste cómo dibujar una imagen en la pantalla y dar al jugador el control sobre el movimiento de los elementos del juego. Creaste elementos que se mueven solos, como balas que vuelan por la pantalla, y eliminaste objetos que ya no son necesarios. También aprendiste a refactorizar periódicamente el código de un proyecto para facilitar el desarrollo continuo.

En el Capítulo 13, añadiremos extraterrestres a *Alien Invasion*. Al final del capítulo, serás capaz de derribar alienígenas, ¡esperemos que antes de que lleguen a tu nave!