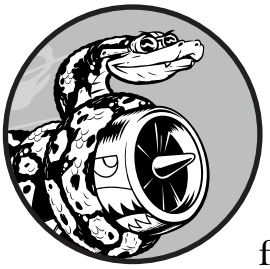
PROJECT 1

ALIEN INVASION

12

A SHIP THAT FIRES BULLETS



Let's build a game called *Alien Invasion*! We'll use Pygame, a collection of fun, powerful Python modules that manage graphics, animation, and even sound, making it easier for you to build sophisticated games. With Pygame handling tasks like drawing images to the screen, you can focus on the higher-level logic of game dynamics.

In this chapter, you'll set up Pygame, and then create a rocket ship that moves right and left and fires bullets in response to player input. In the next two chapters, you'll create a fleet of aliens to destroy, and then continue to refine the game by setting limits on the number of ships you can use and adding a scoreboard.

While building this game, you'll also learn how to manage large projects that span multiple files. We'll refactor a lot of code and manage file contents to organize the project and make the code efficient.

Making games is an ideal way to have fun while learning a language. It's deeply satisfying to play a game you wrote, and writing a simple game will help you comprehend how professionals develop games. As you work through this chapter, enter and run the code to identify how each code block contributes to overall gameplay. Experiment with different values and settings to better understand how to refine interactions in your games.

NOTE Alien Invasion *spans a number of different files, so make a new* alien_invasion *folder on your system. Be sure to save all files for the project to this folder so your* `import` *statements will work correctly.*

Also, if you feel comfortable using version control, you might want to use it for this project. If you haven't used version control before, see Appendix D for an overview.

Planning Your Project

When you're building a large project, it's important to prepare a plan before you begin to write code. Your plan will keep you focused and make it more likely that you'll complete the project.

Let's write a description of the general gameplay. Although the following description doesn't cover every detail of *Alien Invasion*, it provides a clear idea of how to start building the game:

> In *Alien Invasion*, the player controls a rocket ship that appears at the bottom center of the screen. The player can move the ship right and left using the arrow keys and shoot bullets using the spacebar. When the game begins, a fleet of aliens fills the sky and moves across and down the screen. The player shoots and destroys the aliens. If the player shoots all the aliens, a new fleet appears that moves faster than the previous fleet. If any alien hits the player's ship or reaches the bottom of the screen, the player loses a ship. If the player loses three ships, the game ends.

For the first development phase, we'll make a ship that can move right and left and fires bullets when the player presses the spacebar. After setting up this behavior, we can create the aliens and refine the gameplay.

Installing Pygame

Before you begin coding, install Pygame. The `pip` module helps you download and install Python packages. To install Pygame, enter the following command at a terminal prompt:

```
$ python -m pip install --user pygame
```

This command tells Python to run the `pip` module and install the `pygame` package to the current user's Python installation. If you use a command

other than `python` to run programs or start a terminal session, such as `python3`, your command will look like this:

```
$ python3 -m pip install --user pygame
```

NOTE *If this command doesn't work on macOS, try running the command again without the `--user` flag.*

Starting the Game Project

We'll begin building the game by creating an empty Pygame window. Later, we'll draw the game elements, such as the ship and the aliens, on this window. We'll also make our game respond to user input, set the background color, and load a ship image.

Creating a Pygame Window and Responding to User Input

We'll make an empty Pygame window by creating a class to represent the game. In your text editor, create a new file and save it as *alien_invasion.py*; then enter the following:

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()
```

First, we import the `sys` and `pygame` modules. The `pygame` module contains the functionality we need to make a game. We'll use tools in the `sys` module to exit the game when the player quits.

Alien Invasion starts as a class called `AlienInvasion`. In the `__init__()` method, the `pygame.init()` function initializes the background settings that Pygame needs to work properly ❶. At ❷, we call `pygame.display.set_mode()` to create a display window, on which we'll draw all the game's graphical elements. The argument `(1200, 800)` is a tuple that defines the dimensions of the game window, which will be 1200 pixels wide by 800 pixels high. (You can adjust these values depending on your display size.) We assign this display window to the attribute `self.screen`, so it will be available in all methods in the class.

The object we assigned to `self.screen` is called a *surface*. A surface in Pygame is a part of the screen where a game element can be displayed. Each element in the game, like an alien or a ship, is its own surface. The surface returned by `display.set_mode()` represents the entire game window. When we activate the game's animation loop, this surface will be redrawn on every pass through the loop, so it can be updated with any changes triggered by user input.

The game is controlled by the `run_game()` method. This method contains a `while` loop ❸ that runs continually. The `while` loop contains an event loop and code that manages screen updates. An *event* is an action that the user performs while playing the game, such as pressing a key or moving the mouse. To make our program respond to events, we write this *event loop* to *listen* for events and perform appropriate tasks depending on the kinds of events that occur. The `for` loop at ❹ is an event loop.

To access the events that Pygame detects, we'll use the `pygame.event.get()` function. This function returns a list of events that have taken place since the last time this function was called. Any keyboard or mouse event will cause this `for` loop to run. Inside the loop, we'll write a series of `if` statements to detect and respond to specific events. For example, when the player clicks the game window's close button, a `pygame.QUIT` event is detected and we call `sys.exit()` to exit the game ❺.

The call to `pygame.display.flip()` at ❻ tells Pygame to make the most recently drawn screen visible. In this case, it simply draws an empty screen on each pass through the `while` loop, erasing the old screen so only the new screen is visible. When we move the game elements around, `pygame.display.flip()` continually updates the display to show the new positions of game elements and hides the old ones, creating the illusion of smooth movement.

At the end of the file, we create an instance of the game, and then call `run_game()`. We place `run_game()` in an `if` block that only runs if the file is called directly. When you run this *alien_invasion.py* file, you should see an empty Pygame window.

Setting the Background Color

Pygame creates a black screen by default, but that's boring. Let's set a different background color. We'll do this at the end of the `__init__()` method.

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()
```

Colors in Pygame are specified as RGB colors: a mix of red, green, and blue. Each color value can range from 0 to 255. The color value (255, 0, 0) is red, (0, 255, 0) is green, and (0, 0, 255) is blue. You can mix different RGB values to create up to 16 million colors. The color value (230, 230, 230) mixes equal amounts of red, blue, and green, which produces a light gray background color. We assign this color to `self.bg_color` ❶.

At ❷, we fill the screen with the background color using the `fill()` method, which acts on a surface and takes only one argument: a color.

Creating a Settings Class

Each time we introduce new functionality into the game, we'll typically create some new settings as well. Instead of adding settings throughout the code, let's write a module called `settings` that contains a class called `Settings` to store all these values in one place. This approach allows us to work with just one settings object any time we need to access an individual setting. This also makes it easier to modify the game's appearance and behavior as our project grows: to modify the game, we'll simply change some values in *settings.py*, which we'll create next, instead of searching for different settings throughout the project.

Create a new file named *settings.py* inside your *alien_invasion* folder, and add this initial `Settings` class:

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)
```

To make an instance of `Settings` in the project and use it to access our settings, we need to modify *alien_invasion.py* as follows:

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()
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()
--snip--
```

We import `Settings` into the main program file. Then we create an instance of `Settings` and assign it to `self.settings` ❶, after making the call to `pygame.init()`. When we create a screen ❷, we use the `screen_width` and `screen_height` attributes of `self.settings`, and then we use `self.settings` to access the background color when filling the screen at ❸ as well.

When you run *alien_invasion.py* now you won't yet see any changes, because all we've done is move the settings we were already using elsewhere. Now we're ready to start adding new elements to the screen.

Adding the Ship Image

Let's add the ship to our game. To draw the player's ship on the screen, we'll load an image and then use the Pygame `blit()` method to draw the image.

When you're choosing artwork for your games, be sure to pay attention to licensing. The safest and cheapest way to start is to use freely licensed graphics that you can use and modify, from a website like *https://pixabay.com/*.

You can use almost any type of image file in your game, but it's easiest when you use a bitmap (*.bmp*) file because Pygame loads bitmaps by default. Although you can configure Pygame to use other file types, some file types

depend on certain image libraries that must be installed on your computer. Most images you'll find are in *.jpg* or *.png* formats, but you can convert them to bitmaps using tools like Photoshop, GIMP, and Paint.

Pay particular attention to the background color in your chosen image. Try to find a file with a transparent or solid background that you can replace with any background color using an image editor. Your games will look best if the image's background color matches your game's background color. Alternatively, you can match your game's background to the image's background.

For *Alien Invasion*, you can use the file *ship.bmp* (Figure 12-1), which is available in the book's resources at *https://nostarch.com/pythoncrashcourse2e/*. The file's background color matches the settings we're using in this project. Make a folder called *images* inside your main *alien_invasion* project folder. Save the file *ship.bmp* in the *images* folder.

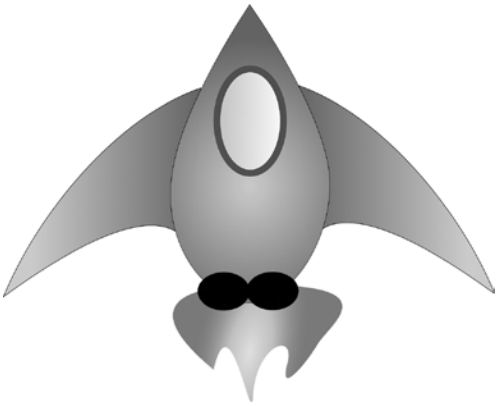


Figure 12-1: The ship for Alien Invasion

Creating the Ship Class

After choosing an image for the ship, we need to display it on the screen. To use our ship, we'll create a new `ship` module that will contain the class `Ship`. This class will manage most of the behavior of the player's ship:

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 is efficient because it lets you treat all game elements like rectangles (*rects*), even if they're not exactly shaped like rectangles. Treating an element as a rectangle is efficient because rectangles are simple geometric shapes. When Pygame needs to figure out whether two game elements have collided, for example, it can do this more quickly if it treats each object as a rectangle. This approach usually works well enough that no one playing the game will notice that we're not working with the exact shape of each game element. We'll treat the ship and the screen as rectangles in this class.

We import the `pygame` module before defining the class. The `__init__()` method of `Ship` takes two parameters: the `self` reference and a reference to the current instance of the `AlienInvasion` class. This will give `Ship` access to all the game resources defined in `AlienInvasion`. At ❶ we assign the screen to an attribute of `Ship`, so we can access it easily in all the methods in this class. At ❷ we access the screen's `rect` attribute using the `get_rect()` method and assign it to `self.screen_rect`. Doing so allows us to place the ship in the correct location on the screen.

To load the image, we call `pygame.image.load()` ❸ and give it the location of our ship image. This function returns a surface representing the ship, which we assign to `self.image`. When the image is loaded, we call `get_rect()` to access the ship surface's `rect` attribute so we can later use it to place the ship.

When you're working with a `rect` object, you can use the x- and y-coordinates of the top, bottom, left, and right edges of the rectangle, as well as the center, to place the object. You can set any of these values to establish the current position of the rect. When you're centering a game element, work with the `center`, `centerx`, or `centery` attributes of a rect. When you're working at an edge of the screen, work with the `top`, `bottom`, `left`, or `right` attributes. There are also attributes that combine these properties, such as `midbottom`, `midtop`, `midleft`, and `midright`. When you're adjusting the horizontal or vertical placement of the rect, you can just use the `x` and `y` attributes, which are the x- and y-coordinates of its top-left corner. These attributes spare you from having to do calculations that game developers formerly had to do manually, and you'll use them often.

NOTE *In Pygame, the origin (0, 0) is at the top-left corner of the screen, and coordinates increase as you go down and to the right. On a 1200 by 800 screen, the origin is at the top-left corner, and the bottom-right corner has the coordinates (1200, 800). These coordinates refer to the game window, not the physical screen.*

We'll position the ship at the bottom center of the screen. To do so, make the value of `self.rect.midbottom` match the `midbottom` attribute of the screen's rect ❹. Pygame uses these `rect` attributes to position the ship image so it's centered horizontally and aligned with the bottom of the screen.

At ❺, we define the `blitme()` method, which draws the image to the screen at the position specified by `self.rect`.

Drawing the Ship to the Screen

Now let's update *alien_invasion.py* so it creates a ship and calls the ship's `blitme()` method:

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()
--snip--
```

We import `Ship` and then make an instance of `Ship` after the screen has been created ❶. The call to `Ship()` requires one argument, an instance of `AlienInvasion`. The `self` argument here refers to the current instance of `AlienInvasion`. This is the parameter that gives `Ship` access to the game's resources, such as the `screen` object. We assign this `Ship` instance to `self.ship`.

After filling the background, we draw the ship on the screen by calling `ship.blitme()`, so the ship appears on top of the background ❷.

When you run *alien_invasion.py* now, you should see an empty game screen with the rocket ship sitting at the bottom center, as shown in Figure 12-2.

Figure 12-2: Alien Invasion *with the ship at the bottom center of the screen*

Refactoring: The _check_events() and _update_screen() Methods

In large projects, you'll often refactor code you've written before adding more code. Refactoring simplifies the structure of the code you've already written, making it easier to build on. In this section, we'll break the `run_game()` method, which is getting lengthy, into two helper methods. A *helper method* does work inside a class but isn't meant to be called through an instance. In Python, a single leading underscore indicates a helper method.

The _check_events() Method

We'll move the code that manages events to a separate method called `_check_events()`. This will simplify `run_game()` and isolate the event management loop. Isolating the event loop allows you to manage events separately from other aspects of the game, such as updating the screen.

Here's the `AlienInvasion` class with the new `_check_events()` method, which only affects the code in `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()
```

We make a new `_check_events()` method ❷ and move the lines that check whether the player has clicked to close the window into this new method.

To call a method from within a class, use dot notation with the variable `self` and the name of the method ❶. We call the method from inside the `while` loop in `run_game()`.

The _update_screen() Method

To further simplify `run_game()`, we'll move the code for updating the screen to a separate method called `_update_screen()`:

alien_invasion.py

```
    def run_game(self):
        """Start the main loop for the game."""
        while True:
            self._check_events()
            self._update_screen()

    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()
```

We moved the code that draws the background and the ship and flips the screen to `_update_screen()`. Now the body of the main loop in `run_game()` is much simpler. It's easy to see that we're looking for new events and updating the screen on each pass through the loop.

If you've already built a number of games, you'll probably start out by breaking your code into methods like these. But if you've never tackled a project like this, you probably won't know how to structure your code. This approach of writing code that works and then restructuring it as it grows more complex gives you an idea of a realistic development process: you start out writing your code as simply as possible, and then refactor it as your project becomes more complex.

Now that we've restructured the code to make it easier to add to, we can work on the dynamic aspects of the game!

TRY IT YOURSELF

12-1. Blue Sky: Make a Pygame window with a blue background.

12-2. Game Character: Find a bitmap image of a game character you like or convert an image to a bitmap. Make a class that draws the character at the center of the screen and match the background color of the image to the background color of the screen, or vice versa.

Piloting the Ship

Next, we'll give the player the ability to move the ship right and left. We'll write code that responds when the player presses the right or left arrow key. We'll focus on movement to the right first, and then we'll apply the same principles to control movement to the left. As we add this code, you'll learn how to control the movement of images on the screen and respond to user input.

Responding to a Keypress

Whenever the player presses a key, that keypress is registered in Pygame as an event. Each event is picked up by the `pygame.event.get()` method. We need to specify in our `_check_events()` method what kind of events we want the game to check for. Each keypress is registered as a `KEYDOWN` event.

When Pygame detects a `KEYDOWN` event, we need to check whether the key that was pressed is one that triggers a certain action. For example, if the player presses the right arrow key, we want to increase the ship's `rect.x` value to move the ship to the right:

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
```

Inside `_check_events()` we add an `elif` block to the event loop to respond when Pygame detects a `KEYDOWN` event ❶. We check whether the key pressed, `event.key`, is the right arrow key ❷. The right arrow key is represented by `pygame.K_RIGHT`. If the right arrow key was pressed, we move the ship to the right by increasing the value of `self.ship.rect.x` by 1 ❸.

When you run *alien_invasion.py* now, the ship should move to the right one pixel every time you press the right arrow key. That's a start, but it's not an efficient way to control the ship. Let's improve this control by allowing continuous movement.

Allowing Continuous Movement

When the player holds down the right arrow key, we want the ship to continue moving right until the player releases the key. We'll have the game detect a `pygame.KEYUP` event so we'll know when the right arrow key is released; then we'll use the `KEYDOWN` and `KEYUP` events together with a flag called `moving_right` to implement continuous motion.

When the `moving_right` flag is `False`, the ship will be motionless. When the player presses the right arrow key, we'll set the flag to `True`, and when the player releases the key, we'll set the flag to `False` again.

The `Ship` class controls all attributes of the ship, so we'll give it an attribute called `moving_right` and an `update()` method to check the status of the `moving_right` flag. The `update()` method will change the position of the ship if the flag is set to `True`. We'll call this method once on each pass through the `while` loop to update the position of the ship.

Here are the changes to `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
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--
```

We add a `self.moving_right` attribute in the `__init__()` method and set it to `False` initially ❶. Then we add `update()`, which moves the ship right if the flag is `True` ❷. The `update()` method will be called through an instance of `Ship`, so it's not considered a helper method.

Now we need to modify `_check_events()` so that `moving_right` is set to `True` when the right arrow key is pressed and `False` when the key is released:

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
```

At ❶, we modify how the game responds when the player presses the right arrow key: instead of changing the ship's position directly, we merely set `moving_right` to `True`. At ❷, we add a new `elif` block, which responds to `KEYUP` events. When the player releases the right arrow key (`K_RIGHT`), we set `moving_right` to `False`.

Next, we modify the `while` loop in `run_game()` so it calls the ship's `update()` method on each pass through the loop:

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()
```

The ship's position will be updated after we've checked for keyboard events and before we update the screen. This allows the ship's position to be updated in response to player input and ensures the updated position will be used when drawing the ship to the screen.

When you run *alien_invasion.py* and hold down the right arrow key, the ship should move continuously to the right until you release the key.

Moving Both Left and Right

Now that the ship can move continuously to the right, adding movement to the left is straightforward. Again, we'll modify the `Ship` class and the `_check_events()` method. Here are the relevant changes to `__init__()` and `update()` in `Ship`:

ship.py

```
    def __init__(self, ai_game):
        --snip--
        # Movement flags
        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
```

In `__init__()`, we add a `self.moving_left` flag. In `update()`, we use two separate `if` blocks rather than an `elif` to allow the ship's `rect.x` value to be increased and then decreased when both arrow keys are held down. This results in the ship standing still. If we used `elif` for motion to the left, the

right arrow key would always have priority. Doing it this way makes the movements more accurate when switching from right to left when the player might momentarily hold down both keys.

We have to make two adjustments to `_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
```

If a `KEYDOWN` event occurs for the `K_LEFT` key, we set `moving_left` to `True`. If a `KEYUP` event occurs for the `K_LEFT` key, we set `moving_left` to `False`. We can use `elif` blocks here because each event is connected to only one key. If the player presses both keys at once, two separate events will be detected.

When you run *alien_invasion.py* now, you should be able to move the ship continuously to the right and left. If you hold down both keys, the ship should stop moving.

Next, we'll further refine the ship's movement. Let's adjust the ship's speed and limit how far the ship can move so it can't disappear off the sides of the screen.

Adjusting the Ship's Speed

Currently, the ship moves one pixel per cycle through the `while` loop, but we can take finer control of the ship's speed by adding a `ship_speed` attribute to the `Settings` class. We'll use this attribute to determine how far to move the ship on each pass through the loop. Here's the new attribute in *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
```

We set the initial value of `ship_speed` to `1.5`. When the ship moves now, its position is adjusted by 1.5 pixels rather than 1 pixel on each pass through the loop.

We're using decimal values for the speed setting to give us finer control of the ship's speed when we increase the tempo of the game later on. However, rect attributes such as `x` store only integer values, so we need to make some modifications to `Ship`:

ship.py

```
class Ship:
    """A class to manage the ship."""

❶   def __init__(self, ai_game):
        """Initialize the ship and set its starting position."""
        self.screen = ai_game.screen
        self.settings = ai_game.settings
        --snip--

        # Start each new ship at the bottom center of the screen.
        --snip--

        # Store a decimal value for the ship's horizontal position.
❷       self.x = float(self.rect.x)

        # Movement flags
        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--
```

We create a `settings` attribute for `Ship`, so we can use it in `update()` ❶. Because we're adjusting the position of the ship by fractions of a pixel, we need to assign the position to a variable that can store a decimal value. You can use a decimal value to set an attribute of `rect`, but the `rect` will only keep the integer portion of that value. To keep track of the ship's position accurately, we define a new `self.x` attribute that can hold decimal values ❷. We use the `float()` function to convert the value of `self.rect.x` to a decimal and assign this value to `self.x`.

Now when we change the ship's position in `update()`, the value of `self.x` is adjusted by the amount stored in `settings.ship_speed` ❸. After `self.x` has been updated, we use the new value to update `self.rect.x`, which controls

the position of the ship ❹. Only the integer portion of `self.x` will be stored in `self.rect.x`, but that's fine for displaying the ship.

Now we can change the value of `ship_speed`, and any value greater than one will make the ship move faster. This will help make the ship respond quickly enough to shoot down aliens, and it will let us change the tempo of the game as the player progresses in gameplay.

NOTE *If you're using macOS, you might notice that the ship moves very slowly, even with a high speed setting. You can remedy this problem by running the game in fullscreen mode, which we'll implement shortly.*

Limiting the Ship's Range

At this point, the ship will disappear off either edge of the screen if you hold down an arrow key long enough. Let's correct this so the ship stops moving when it reaches the screen's edge. We do this by modifying the `update()` method in `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
```

This code checks the position of the ship before changing the value of `self.x`. The code `self.rect.right` returns the x-coordinate of the right edge of the ship's `rect`. If this value is less than the value returned by `self.screen_rect.right`, the ship hasn't reached the right edge of the screen ❶. The same goes for the left edge: if the value of the left side of the `rect` is greater than zero, the ship hasn't reached the left edge of the screen ❷. This ensures the ship is within these bounds before adjusting the value of `self.x`.

When you run *alien_invasion.py* now, the ship should stop moving at either edge of the screen. This is pretty cool; all we've done is add a conditional test in an `if` statement, but it feels like the ship hits a wall or a force field at either edge of the screen!

Refactoring _check_events()

The `_check_events()` method will increase in length as we continue to develop the game, so let's break `_check_events()` into two more methods: one that handles `KEYDOWN` events and another that handles `KEYUP` events:

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
```

We make two new helper methods: `_check_keydown_events()` and `_check_keyup_events()`. Each needs a `self` parameter and an `event` parameter. The bodies of these two methods are copied from `_check_events()`, and we've replaced the old code with calls to the new methods. The `_check_events()` method is simpler now with this cleaner code structure, which will make it easier to develop further responses to player input.

Pressing Q to Quit

Now that we're responding to keypresses efficiently, we can add another way to quit the game. It gets tedious to click the X at the top of the game window to end the game every time you test a new feature, so we'll add a keyboard shortcut to end the game when the player presses 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()
```

In `_check_keydown_events()`, we add a new block that ends the game when the player presses Q. Now, when testing, you can press Q to close the game rather than using your cursor to close the window.

Running the Game in Fullscreen Mode

Pygame has a fullscreen mode that you might like better than running the game in a regular window. Some games look better in fullscreen mode, and macOS users might see better performance in fullscreen mode.

To run the game in fullscreen mode, make the following changes in `__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")
```

When creating the screen surface, we pass a size of `(0, 0)` and the parameter `pygame.FULLSCREEN` ❶. This tells Pygame to figure out a window size that will fill the screen. Because we don't know the width and height of the screen ahead of time, we update these settings after the screen is created ❷. We use the `width` and `height` attributes of the screen's rect to update the `settings` object.

If you like how the game looks or behaves in fullscreen mode, keep these settings. If you liked the game better in its own window, you can revert back to the original approach where we set a specific screen size for the game.

NOTE *Make sure you can quit by pressing Q before running the game in fullscreen mode; Pygame offers no default way to quit a game while in fullscreen mode.*

A Quick Recap

In the next section, we'll add the ability to shoot bullets, which involves adding a new file called *bullet.py* and making some modifications to some of the files we're already using. Right now, we have three files containing a number of classes and methods. To be clear about how the project is organized, let's review each of these files before adding more functionality.

alien_invasion.py

The main file, *alien_invasion.py*, contains the `AlienInvasion` class. This class creates a number of important attributes used throughout the game: the settings are assigned to `settings`, the main display surface is assigned to `screen`, and a `ship` instance is created in this file as well. The main loop of the game, a `while` loop, is also stored in this module. The `while` loop calls `_check_events()`, `ship.update()`, and `_update_screen()`.

The `_check_events()` method detects relevant events, such as keypresses and releases, and processes each of these types of events through the methods `_check_keydown_events()` and `_check_keyup_events()`. For now,

these methods manage the ship's movement. The `AlienInvasion` class also contains `_update_screen()`, which redraws the screen on each pass through the main loop.

The *alien_invasion.py* file is the only file you need to run when you want to play *Alien Invasion*. The other files—*settings.py* and *ship.py*—contain code that is imported into this file.

settings.py

The *settings.py* file contains the `Settings` class. This class only has an `__init__()` method, which initializes attributes controlling the game's appearance and the ship's speed.

ship.py

The *ship.py* file contains the `Ship` class. The `Ship` class has an `__init__()` method, an `update()` method to manage the ship's position, and a `blitme()` method to draw the ship to the screen. The image of the ship is stored in *ship.bmp*, which is in the *images* folder.

TRY IT YOURSELF

12-3. Pygame Documentation: We're far enough into the game now that you might want to look at some of the Pygame documentation. The Pygame home page is at *https://www.pygame.org/*, and the home page for the documentation is at *https://www.pygame.org/docs/*. Just skim the documentation for now. You won't need it to complete this project, but it will help if you want to modify *Alien Invasion* or make your own game afterward.

12-4. Rocket: Make a game that begins with a rocket in the center of the screen. Allow the player to move the rocket up, down, left, or right using the four arrow keys. Make sure the rocket never moves beyond any edge of the screen.

12-5. Keys: Make a Pygame file that creates an empty screen. In the event loop, print the `event.key` attribute whenever a `pygame.KEYDOWN` event is detected. Run the program and press various keys to see how Pygame responds.

Shooting Bullets

Now let's add the ability to shoot bullets. We'll write code that fires a bullet, which is represented by a small rectangle, when the player presses the spacebar. Bullets will then travel straight up the screen until they disappear off the top of the screen.

Adding the Bullet Settings

At the end of the `__init__()` method, we'll update *settings.py* to include the values we'll need for a new `Bullet` class:

settings.py

```
    def __init__(self):
        --snip--
        # Bullet settings
        self.bullet_speed = 1.0
        self.bullet_width = 3
        self.bullet_height = 15
        self.bullet_color = (60, 60, 60)
```

These settings create dark gray bullets with a width of 3 pixels and a height of 15 pixels. The bullets will travel slightly slower than the ship.

Creating the Bullet Class

Now create a *bullet.py* file to store our `Bullet` class. Here's the first part of *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 decimal value.
❸       self.y = float(self.rect.y)
```

The `Bullet` class inherits from `Sprite`, which we import from the `pygame.sprite` module. When you use sprites, you can group related elements in your game and act on all the grouped elements at once. To create a bullet instance, `__init__()` needs the current instance of `AlienInvasion`, and we call `super()` to inherit properly from `Sprite`. We also set attributes for the screen and settings objects, and for the bullet's color.

At ❶, we create the bullet's `rect` attribute. The bullet isn't based on an image, so we have to build a rect from scratch using the `pygame.Rect()` class. This class requires the x- and y-coordinates of the top-left corner of the

rect, and the width and height of the rect. We initialize the rect at (0, 0), but we'll move it to the correct location in the next line, because the bullet's position depends on the ship's position. We get the width and height of the bullet from the values stored in self.settings.

At ❷, we set the bullet's midtop attribute to match the ship's midtop attribute. This will make the bullet emerge from the top of the ship, making it look like the bullet is fired from the ship. We store a decimal value for the bullet's y-coordinate so we can make fine adjustments to the bullet's speed ❸.

Here's the second part of *bullet.py*, update() and draw_bullet():

bullet.py

```
    def update(self):
        """Move the bullet up the screen."""
        # Update the decimal 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)
```

The update() method manages the bullet's position. When a bullet is fired, it moves up the screen, which corresponds to a decreasing y-coordinate value. To update the position, we subtract the amount stored in settings.bullet_speed from self.y ❶. We then use the value of self.y to set the value of self.rect.y ❷.

The bullet_speed setting allows us to increase the speed of the bullets as the game progresses or as needed to refine the game's behavior. Once a bullet is fired, we never change the value of its x-coordinate, so it will travel vertically in a straight line even if the ship moves.

When we want to draw a bullet, we call draw_bullet(). The draw.rect() function fills the part of the screen defined by the bullet's rect with the color stored in self.color ❸.

Storing Bullets in a Group

Now that we have a Bullet class and the necessary settings defined, we can write code to fire a bullet each time the player presses the spacebar. We'll create a group in AlienInvasion to store all the live bullets so we can manage the bullets that have already been fired. This group will be an instance of the pygame.sprite.Group class, which behaves like a list with some extra functionality that's helpful when building games. We'll use this group to draw bullets to the screen on each pass through the main loop and to update each bullet's position.

We'll create the group in __init__():

alien_invasion.py

```
    def __init__(self):
        --snip--
        self.ship = Ship(self)
        self.bullets = pygame.sprite.Group()
```

Then we need to update the position of the bullets on each pass through the `while` loop:

alien_invasion.py

```
    def run_game(self):
        """Start the main loop for the game."""
        while True:
            self._check_events()
            self.ship.update()
❶           self.bullets.update()
            self._update_screen()
```

When you call `update()` on a group ❶, the group automatically calls `update()` for each sprite in the group. The line `self.bullets.update()` calls `bullet.update()` for each bullet we place in the group `bullets`.

Firing Bullets

In `AlienInvasion`, we need to modify `_check_keydown_events()` to fire a bullet when the player presses the spacebar. We don't need to change `_check_keyup_events()` because nothing happens when the spacebar is released. We also need to modify `_update_screen()` to make sure each bullet is drawn to the screen before we call `flip()`.

We know there will be a bit of work to do when we fire a bullet, so let's write a new method, `_fire_bullet()`, to handle this work:

alien_invasion.py

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

class AlienInvasion:
    --snip--
    def _check_keydown_events(self, event):
        --snip--
        elif event.key == pygame.K_q:
            sys.exit()
❷       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."""
❸       new_bullet = Bullet(self)
❹       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)
        self.ship.blitme()
❺       for bullet in self.bullets.sprites():
            bullet.draw_bullet()
```

```
        pygame.display.flip()
--snip--
```

First, we import `Bullet` ❶. Then we call `_fire_bullet()` when the spacebar is pressed ❷. In `_fire_bullet()`, we make an instance of `Bullet` and call it `new_bullet` ❸. We then add it to the group `bullets` using the `add()` method ❹. The `add()` method is similar to `append()`, but it's a method that's written specifically for Pygame groups.

The `bullets.sprites()` method returns a list of all sprites in the group `bullets`. To draw all fired bullets to the screen, we loop through the sprites in `bullets` and call `draw_bullet()` on each one ❺.

When you run *alien_invasion.py* now, you should be able to move the ship right and left, and fire as many bullets as you want. The bullets travel up the screen and disappear when they reach the top, as shown in Figure 12-3. You can alter the size, color, and speed of the bullets in *settings.py*.

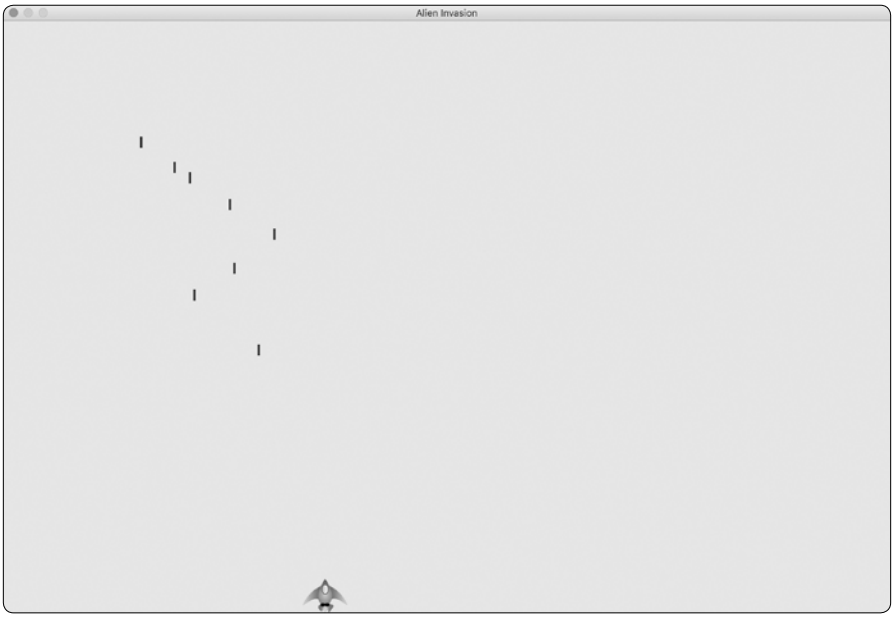


Figure 12-3: The ship after firing a series of bullets

Deleting Old Bullets

At the moment, the bullets disappear when they reach the top, but only because Pygame can't draw them above the top of the screen. The bullets actually continue to exist; their y-coordinate values just grow increasingly negative. This is a problem, because they continue to consume memory and processing power.

We need to get rid of these old bullets, or the game will slow down from doing so much unnecessary work. To do this, we need to detect when the `bottom` value of a bullet's `rect` has a value of 0, which indicates the bullet has passed off the top of the screen:

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()
```

When you use a `for` loop with a list (or a group in Pygame), Python expects that the list will stay the same length as long as the loop is running. Because we can't remove items from a list or group within a `for` loop, we have to loop over a copy of the group. We use the `copy()` method to set up the `for` loop ❶, which enables us to modify `bullets` inside the loop. We check each bullet to see whether it has disappeared off the top of the screen at ❷. If it has, we remove it from `bullets` ❸. At ❹ we insert a `print()` call to show how many bullets currently exist in the game and verify that they're being deleted when they reach the top of the screen.

If this code works correctly, we can watch the terminal output while firing bullets and see that the number of bullets decreases to zero after each series of bullets has cleared the top of the screen. After you run the game and verify that bullets are being deleted properly, remove the `print()` call. If you leave it in, the game will slow down significantly because it takes more time to write output to the terminal than it does to draw graphics to the game window.

Limiting the Number of Bullets

Many shooting games limit the number of bullets a player can have on the screen at one time; doing so encourages players to shoot accurately. We'll do the same in *Alien Invasion*.

First, store the number of bullets allowed in *settings.py*:

settings.py

```
        # Bullet settings
        --snip--
        self.bullet_color = (60, 60, 60)
        self.bullets_allowed = 3
```

This limits the player to three bullets at a time. We'll use this setting in `AlienInvasion` to check how many bullets exist before creating a new bullet in `_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)
```

When the player presses the spacebar, we check the length of `bullets`. If `len(self.bullets)` is less than three, we create a new bullet. But if three bullets are already active, nothing happens when the spacebar is pressed. When you run the game now, you should be able to fire bullets only in groups of three.

Creating the _update_bullets() Method

We want to keep the `AlienInvasion` class reasonably well organized, so now that we've written and checked the bullet management code, we can move it to a separate method. We'll create a new method called `_update_bullets()` and add it just before `_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)
```

The code for `_update_bullets()` is cut and pasted from `run_game()`; all we've done here is clarify the comments.

The `while` loop in `run_game()` looks simple again:

alien_invasion.py

```
        while True:
            self._check_events()
            self.ship.update()
            self._update_bullets()
            self._update_screen()
```

Now our main loop contains only minimal code, so we can quickly read the method names and understand what's happening in the game. The main loop checks for player input, and then updates the position of the ship and any bullets that have been fired. We then use the updated positions to draw a new screen.

Run *alien_invasion.py* one more time, and make sure you can still fire bullets without errors.

TRY IT YOURSELF

12-6. Sideways Shooter: Write a game that places a ship on the left side of the screen and allows the player to move the ship up and down. Make the ship fire a bullet that travels right across the screen when the player presses the spacebar. Make sure bullets are deleted once they disappear off the screen.

Summary

In this chapter, you learned to make a plan for a game and learned the basic structure of a game written in Pygame. You learned to set a background color and store settings in a separate class where you can adjust them more easily. You saw how to draw an image to the screen and give the player control over the movement of game elements. You created elements that move on their own, like bullets flying up a screen, and deleted objects that are no longer needed. You also learned to refactor code in a project on a regular basis to facilitate ongoing development.

In Chapter 13, we'll add aliens to *Alien Invasion*. By the end of the chapter, you'll be able to shoot down aliens, hopefully before they reach your ship!

13

ALIENS!

In this chapter, we'll add aliens to *Alien Invasion*. We'll add one alien near the top of the screen and then generate a whole fleet of aliens. We'll make the fleet advance sideways and down, and we'll get rid of any aliens hit by a bullet. Finally, we'll limit the number of ships a player has and end the game when the player runs out of ships.

As you work through this chapter, you'll learn more about Pygame and about managing a large project. You'll also learn to detect collisions between game objects, like bullets and aliens. Detecting collisions helps you define interactions between elements in your games: for example, you can

confine a character inside the walls of a maze or pass a ball between two characters. We'll continue to work from a plan that we revisit occasionally to maintain the focus of our code-writing sessions.

Before we start writing new code to add a fleet of aliens to the screen, let's look at the project and update our plan.

Reviewing the Project

When you're beginning a new phase of development on a large project, it's always a good idea to revisit your plan and clarify what you want to accomplish with the code you're about to write. In this chapter, we'll:

- Examine our code and determine if we need to refactor before implementing new features.
- Add a single alien to the top-left corner of the screen with appropriate spacing around it.
- Use the spacing around the first alien and the overall screen size to determine how many aliens can fit on the screen. We'll write a loop to create aliens to fill the upper portion of the screen.
- Make the fleet move sideways and down until the entire fleet is shot down, an alien hits the ship, or an alien reaches the ground. If the entire fleet is shot down, we'll create a new fleet. If an alien hits the ship or the ground, we'll destroy the ship and create a new fleet.
- Limit the number of ships the player can use, and end the game when the player has used up the allotted number of ships.

We'll refine this plan as we implement features, but this is sufficient to start with.

You should also review your existing code when you begin working on a new series of features in a project. Because each new phase typically makes a project more complex, it's best to clean up any cluttered or inefficient code. We've been refactoring as we go, so there isn't any code that we need to refactor at this point.

Creating the First Alien

Placing one alien on the screen is like placing a ship on the screen. Each alien's behavior is controlled by a class called `Alien`, which we'll structure like the `Ship` class. We'll continue using bitmap images for simplicity. You can find your own image for an alien or use the one shown in Figure 13-1, which is available in the book's resources at *https://nostarch.com/pythoncrashcourse2e/*. This image has a gray background, which matches the screen's background color. Make sure you save the image file you choose in the *images* folder.

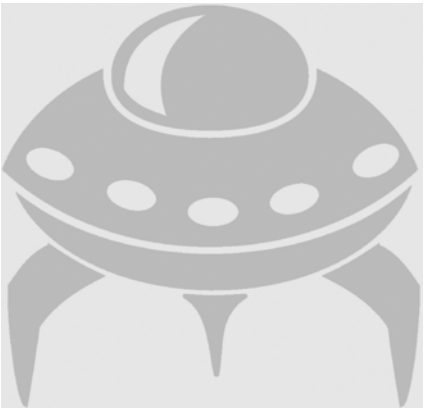


Figure 13-1: The alien we'll use to build the fleet

Creating the Alien Class

Now we'll write the `Alien` class and save it as *alien.py*:

alien.py

```
import pygame
from pygame.sprite import Sprite

class Alien(Sprite):
    """A class to represent a single alien in the fleet."""

    def __init__(self, ai_game):
        """Initialize the alien and set its starting position."""
        super().__init__()
        self.screen = ai_game.screen

        # Load the alien image and set its rect attribute.
        self.image = pygame.image.load('images/alien.bmp')
        self.rect = self.image.get_rect()

        # Start each new alien near the top left of the screen.
❶       self.rect.x = self.rect.width
        self.rect.y = self.rect.height

        # Store the alien's exact horizontal position.
❷       self.x = float(self.rect.x)
```

Most of this class is like the `Ship` class except for the aliens' placement on the screen. We initially place each alien near the top-left corner of the screen; we add a space to the left of it that's equal to the alien's width and a space above it equal to its height ❶ so it's easy to see. We're mainly

concerned with the aliens' horizontal speed, so we'll track the horizontal position of each alien precisely ❷.

This `Alien` class doesn't need a method for drawing it to the screen; instead, we'll use a Pygame group method that automatically draws all the elements of a group to the screen.

Creating an Instance of the Alien

We want to create an instance of `Alien` so we can see the first alien on the screen. Because it's part of our setup work, we'll add the code for this instance at the end of the `__init__()` method in `AlienInvasion`. Eventually, we'll create an entire fleet of aliens, which will be quite a bit of work, so we'll make a new helper method called `_create_fleet()`.

The order of methods in a class doesn't matter, as long as there's some consistency to how they're placed. I'll place `_create_fleet()` just before the `_update_screen()` method, but anywhere in `AlienInvasion` will work. First, we'll import the `Alien` class.

Here are the updated `import` statements for *alien_invasion.py*:

alien_invasion.py

```
--snip--
from bullet import Bullet
from alien import Alien
```

And here's the updated `__init__()` method:

alien_invasion.py

```
    def __init__(self):
        --snip--
        self.ship = Ship(self)
        self.bullets = pygame.sprite.Group()
        self.aliens = pygame.sprite.Group()

        self._create_fleet()
```

We create a group to hold the fleet of aliens, and we call `_create_fleet()`, which we're about to write.

Here's the new `_create_fleet()` method:

alien_invasion.py

```
    def _create_fleet(self):
        """Create the fleet of aliens."""
        # Make an alien.
        alien = Alien(self)
        self.aliens.add(alien)
```

In this method, we're creating one instance of `Alien`, and then adding it to the group that will hold the fleet. The alien will be placed in the default upper-left area of the screen, which is perfect for the first alien.

To make the alien appear, we need to call the group's `draw()` method in `_update_screen()`:

alien_invasion.py

```
    def _update_screen(self):
        --snip--
        for bullet in self.bullets.sprites():
            bullet.draw_bullet()
        self.aliens.draw(self.screen)

        pygame.display.flip()
```

When you call `draw()` on a group, Pygame draws each element in the group at the position defined by its `rect` attribute. The `draw()` method requires one argument: a surface on which to draw the elements from the group. Figure 13-2 shows the first alien on the screen.

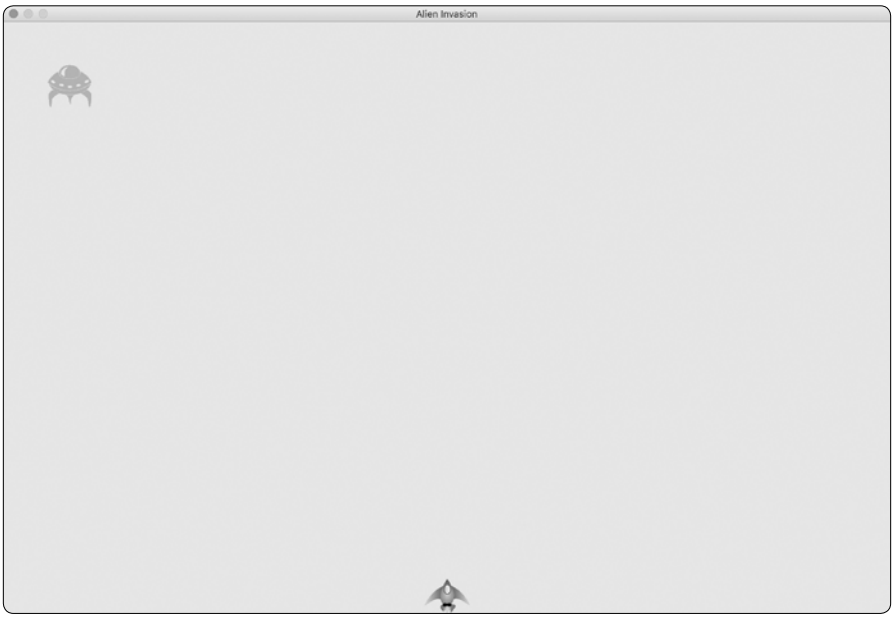


Figure 13-2: The first alien appears.

Now that the first alien appears correctly, we'll write the code to draw an entire fleet.

Building the Alien Fleet

To draw a fleet, we need to figure out how many aliens can fit across the screen and how many rows of aliens can fit down the screen. We'll first figure out the horizontal spacing between aliens and create a row; then we'll determine the vertical spacing and create an entire fleet.

Determining How Many Aliens Fit in a Row

To figure out how many aliens fit in a row, let's look at how much horizontal space we have. The screen width is stored in `settings.screen_width`, but we need an empty margin on either side of the screen. We'll make this margin the width of one alien. Because we have two margins, the available space for aliens is the screen width minus two alien widths:

```
available_space_x = settings.screen_width - (2 * alien_width)
```

We also need to set the spacing between aliens; we'll make it one alien width. The space needed to display one alien is twice its width: one width for the alien and one width for the empty space to its right. To find the number of aliens that fit across the screen, we divide the available space by two times the width of an alien. We use *floor division* (`//`), which divides two numbers and drops any remainder, so we'll get an integer number of aliens:

```
number_aliens_x = available_space_x // (2 * alien_width)
```

We'll use these calculations when we create the fleet.

NOTE *One great aspect of calculations in programming is that you don't have to be sure your formulas are correct when you first write them. You can try them out and see if they work. At worst, you'll have a screen that's overcrowded with aliens or has too few aliens. You can then revise your calculations based on what you see on the screen.*

Creating a Row of Aliens

We're ready to generate a full row of aliens. Because our code for making a single alien works, we'll rewrite `_create_fleet()` to make a whole row of aliens:

alien_invasion.py

```
    def _create_fleet(self):
        """Create the fleet of aliens."""
        # Create an alien and find the number of aliens in a row.
        # Spacing between each alien is equal to one alien width.
❶       alien = Alien(self)
❷       alien_width = alien.rect.width
❸       available_space_x = self.settings.screen_width - (2 * alien_width)
        number_aliens_x = available_space_x // (2 * alien_width)

        # Create the first row of aliens.
❹       for alien_number in range(number_aliens_x):
            # Create an alien and place it in the row.
            alien = Alien(self)
❺           alien.x = alien_width + 2 * alien_width * alien_number
            alien.rect.x = alien.x
            self.aliens.add(alien)
```

We've already thought through most of this code. We need to know the alien's width and height to place aliens, so we create an alien at ❶ before we perform calculations. This alien won't be part of the fleet, so don't add it to the group `aliens`. At ❷ we get the alien's width from its `rect` attribute and store this value in `alien_width` so we don't have to keep working through the `rect` attribute. At ❸ we calculate the horizontal space available for aliens and the number of aliens that can fit into that space.

Next, we set up a loop that counts from 0 to the number of aliens we need to make ❹. In the main body of the loop, we create a new alien and then set its x-coordinate value to place it in the row ❺. Each alien is pushed to the right one alien width from the left margin. Next, we multiply the alien width by 2 to account for the space each alien takes up, including the empty space to its right, and we multiply this amount by the alien's position in the row. We use the alien's `x` attribute to set the position of its rect. Then we add each new alien to the group `aliens`.

When you run *Alien Invasion* now, you should see the first row of aliens appear, as in Figure 13-3.

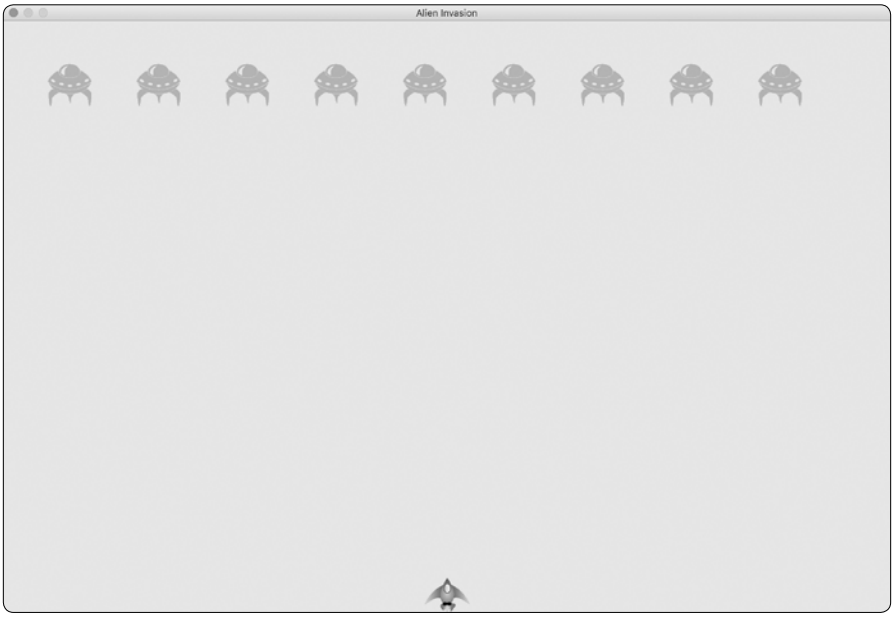


Figure 13-3: The first row of aliens

The first row is offset to the left, which is actually good for gameplay. The reason is that we want the fleet to move right until it hits the edge of the screen, then drop down a bit, then move left, and so forth. Like the classic game *Space Invaders*, this movement is more interesting than having the fleet drop straight down. We'll continue this motion until all aliens are shot down or until an alien hits the ship or the bottom of the screen.

NOTE *Depending on the screen width you've chosen, the alignment of the first row of aliens might look slightly different on your system.*

Refactoring _create_fleet()

If the code we've written so far was all we need to create a fleet, we'd probably leave `_create_fleet()` as is. But we have more work to do, so let's clean up the method a bit. We'll add a new helper method, `_create_alien()`, and call it from `_create_fleet()`:

alien_invasion.py

```
    def _create_fleet(self):
        --snip--
        # Create the first row of aliens.
        for alien_number in range(number_aliens_x):
            self._create_alien(alien_number)

    def _create_alien(self, alien_number):
        """Create an alien and place it in the row."""
        alien = Alien(self)
        alien_width = alien.rect.width
        alien.x = alien_width + 2 * alien_width * alien_number
        alien.rect.x = alien.x
        self.aliens.add(alien)
```

The method `_create_alien()` requires one parameter in addition to `self`: it needs the alien number that's currently being created. We use the same body we made for `_create_fleet()` except that we get the width of an alien inside the method instead of passing it as an argument. This refactoring will make it easier to add new rows and create an entire fleet.

Adding Rows

To finish the fleet, we'll determine the number of rows that fit on the screen and then repeat the loop for creating the aliens in one row until we have the correct number of rows. To determine the number of rows, we find the available vertical space by subtracting the alien height from the top, the ship height from the bottom, and two alien heights from the bottom of the screen:

```
available_space_y = settings.screen_height - (3 * alien_height) - ship_height
```

The result will create some empty space above the ship, so the player has some time to start shooting aliens at the beginning of each level.

Each row needs some empty space below it, which we'll make equal to the height of one alien. To find the number of rows, we divide the available space by two times the height of an alien. We use floor division because we can only make an integer number of rows. (Again, if these calculations are off, we'll see it right away and adjust our approach until we have reasonable spacing.)

```
number_rows = available_height_y // (2 * alien_height)
```

Now that we know how many rows fit in a fleet, we can repeat the code for creating a row:

alien_invasion.py

```
    def _create_fleet(self):
        --snip--
        alien = Alien(self)
❶       alien_width, alien_height = alien.rect.size
        available_space_x = self.settings.screen_width - (2 * alien_width)
        number_aliens_x = available_space_x // (2 * alien_width)

        # Determine the number of rows of aliens that fit on the screen.
        ship_height = self.ship.rect.height
❷       available_space_y = (self.settings.screen_height -
                             (3 * alien_height) - ship_height)
        number_rows = available_space_y // (2 * alien_height)

        # Create the full fleet of aliens.
❸       for row_number in range(number_rows):
            for alien_number in range(number_aliens_x):
                self._create_alien(alien_number, row_number)

    def _create_alien(self, alien_number, row_number):
        """Create an alien and place it in the row."""
        alien = Alien(self)
        alien_width, alien_height = alien.rect.size
        alien.x = alien_width + 2 * alien_width * alien_number
        alien.rect.x = alien.x
❹       alien.rect.y = alien.rect.height + 2 * alien.rect.height * row_number
        self.aliens.add(alien)
```

We need the width and height of an alien, so at ❶ we use the attribute `size`, which contains a tuple with the width and height of a `rect` object. To calculate the number of rows we can fit on the screen, we write our `available_space_y` calculation right after the calculation for `available_space_x` ❷. The calculation is wrapped in parentheses so the outcome can be split over two lines, which results in lines of 79 characters or less, as is recommended.

To create multiple rows, we use two nested loops: one outer and one inner loop ❸. The inner loop creates the aliens in one row. The outer loop counts from 0 to the number of rows we want; Python uses the code for making a single row and repeats it `number_rows` times.

To nest the loops, write the new `for` loop and indent the code you want to repeat. (Most text editors make it easy to indent and unindent blocks of code, but for help see Appendix B.) Now when we call `_create_alien()`, we include an argument for the row number so each row can be placed farther down the screen.

The definition of `_create_alien()` needs a parameter to hold the row number. Within `_create_alien()`, we change an alien's y-coordinate value when it's not in the first row ❹ by starting with one alien's height to create empty space at the top of the screen. Each row starts two alien heights below

the previous row, so we multiply the alien height by two and then by the row number. The first row number is 0, so the vertical placement of the first row is unchanged. All subsequent rows are placed farther down the screen.

When you run the game now, you should see a full fleet of aliens, as shown in Figure 13-4.

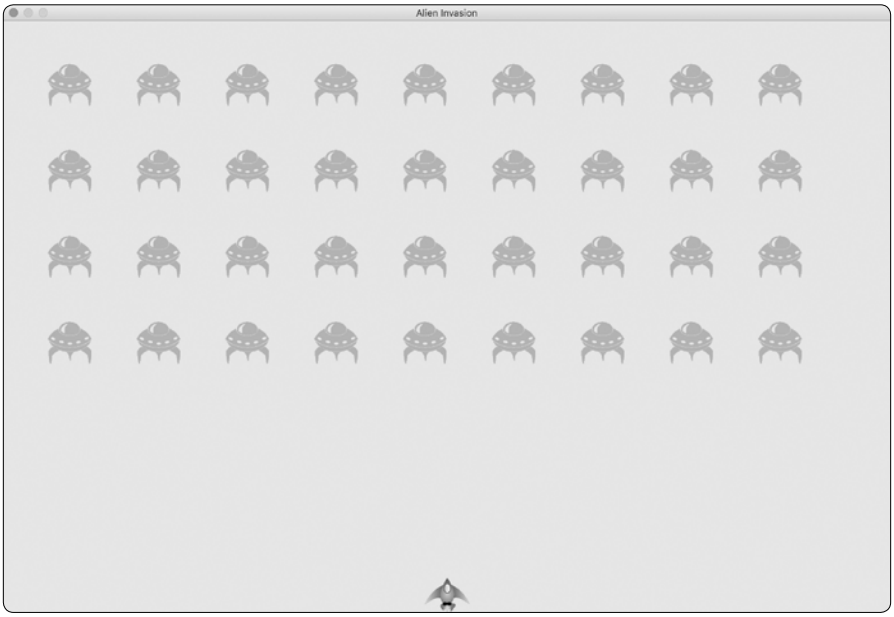


Figure 13-4: The full fleet appears.

In the next section, we'll make the fleet move!

TRY IT YOURSELF

13-1. Stars: Find an image of a star. Make a grid of stars appear on the screen.

13-2. Better Stars: You can make a more realistic star pattern by introducing randomness when you place each star. Recall that you can get a random number like this:

```
from random import randint
random_number = randint(-10, 10)
```

This code returns a random integer between –10 and 10. Using your code in Exercise 13-1, adjust each star's position by a random amount.

Making the Fleet Move

Now let's make the fleet of aliens move to the right across the screen until it hits the edge, and then make it drop a set amount and move in the other direction. We'll continue this movement until all aliens have been shot down, one collides with the ship, or one reaches the bottom of the screen. Let's begin by making the fleet move to the right.

Moving the Aliens Right

To move the aliens, we'll use an `update()` method in *alien.py*, which we'll call for each alien in the group of aliens. First, add a setting to control the speed of each alien:

settings.py

```
    def __init__(self):
        --snip--
        # Alien settings
        self.alien_speed = 1.0
```

Then use this setting to implement `update()`:

alien.py

```
    def __init__(self, ai_game):
        """Initialize the alien and set its starting position."""
        super().__init__()
        self.screen = ai_game.screen
        self.settings = ai_game.settings
        --snip--

    def update(self):
        """Move the alien to the right."""
❶       self.x += self.settings.alien_speed
❷       self.rect.x = self.x
```

We create a settings parameter in `__init__()` so we can access the alien's speed in `update()`. Each time we update an alien's position, we move it to the right by the amount stored in `alien_speed`. We track the alien's exact position with the `self.x` attribute, which can hold decimal values ❶. We then use the value of `self.x` to update the position of the alien's `rect` ❷.

In the main `while` loop, we already have calls to update the ship and bullet positions. Now we'll add a call to update the position of each alien as well:

alien_invasion.py

```
        while True:
            self._check_events()
            self.ship.update()
            self._update_bullets()
            self._update_aliens()
            self._update_screen()
```

We're about to write some code to manage the movement of the fleet, so we create a new method called `_update_aliens()`. We set the aliens' positions to update after the bullets have been updated, because we'll soon be checking to see whether any bullets hit any aliens.

Where you place this method in the module is not critical. But to keep the code organized, I'll place it just after `_update_bullets()` to match the order of method calls in the `while` loop. Here's the first version of `_update_aliens()`:

alien_invasion.py

```
    def _update_aliens(self):
        """Update the positions of all aliens in the fleet."""
        self.aliens.update()
```

We use the `update()` method on the `aliens` group, which calls each alien's `update()` method. When you run *Alien Invasion* now, you should see the fleet move right and disappear off the side of the screen.

Creating Settings for Fleet Direction

Now we'll create the settings that will make the fleet move down the screen and to the left when it hits the right edge of the screen. Here's how to implement this behavior:

settings.py

```
        # Alien settings
        self.alien_speed = 1.0
        self.fleet_drop_speed = 10
        # fleet_direction of 1 represents right; -1 represents left.
        self.fleet_direction = 1
```

The setting `fleet_drop_speed` controls how quickly the fleet drops down the screen each time an alien reaches either edge. It's helpful to separate this speed from the aliens' horizontal speed so you can adjust the two speeds independently.

To implement the setting `fleet_direction`, we could use a text value, such as `'left'` or `'right'`, but we'd end up with `if-elif` statements testing for the fleet direction. Instead, because we have only two directions to deal with, let's use the values 1 and –1, and switch between them each time the fleet changes direction. (Using numbers also makes sense because moving right involves adding to each alien's x-coordinate value, and moving left involves subtracting from each alien's x-coordinate value.)

Checking Whether an Alien Has Hit the Edge

We need a method to check whether an alien is at either edge, and we need to modify `update()` to allow each alien to move in the appropriate direction. This code is part of the `Alien` class:

alien.py

```
    def check_edges(self):
        """Return True if alien is at edge of screen."""
        screen_rect = self.screen.get_rect()
```

```
❶         if self.rect.right >= screen_rect.right or self.rect.left <= 0:
              return True

      def update(self):
          """Move the alien right or left."""
❷         self.x += (self.settings.alien_speed *
                         self.settings.fleet_direction)
          self.rect.x = self.x
```

We can call the new method `check_edges()` on any alien to see whether it's at the left or right edge. The alien is at the right edge if the `right` attribute of its `rect` is greater than or equal to the `right` attribute of the screen's `rect`. It's at the left edge if its `left` value is less than or equal to 0 ❶.

We modify the method `update()` to allow motion to the left or right by multiplying the alien's speed by the value of `fleet_direction` ❷. If `fleet_direction` is 1, the value of `alien_speed` will be added to the alien's current position, moving the alien to the right; if `fleet_direction` is –1, the value will be subtracted from the alien's position, moving the alien to the left.

Dropping the Fleet and Changing Direction

When an alien reaches the edge, the entire fleet needs to drop down and change direction. Therefore, we need to add some code to `AlienInvasion` because that's where we'll check whether any aliens are at the left or right edge. We'll make this happen by writing the methods `_check_fleet_edges()` and `_change_fleet_direction()`, and then modifying `_update_aliens()`. I'll put these new methods after `_create_alien()`, but again the placement of these methods in the class isn't critical.

alien_invasion.py

```
      def _check_fleet_edges(self):
          """Respond appropriately if any aliens have reached an edge."""
❶         for alien in self.aliens.sprites():
              if alien.check_edges():
❷                 self._change_fleet_direction()
                  break

      def _change_fleet_direction(self):
          """Drop the entire fleet and change the fleet's direction."""
          for alien in self.aliens.sprites():
❸             alien.rect.y += self.settings.fleet_drop_speed
          self.settings.fleet_direction *= -1
```

In `_check_fleet_edges()`, we loop through the fleet and call `check_edges()` on each alien ❶. If `check_edges()` returns `True`, we know an alien is at an edge and the whole fleet needs to change direction; so we call `_change_fleet_direction()` and break out of the loop ❷. In `_change_fleet_direction()`, we loop through all the aliens and drop each one using the setting `fleet_drop_speed` ❸; then we change the value of `fleet_direction` by multiplying its current value by –1. The line that changes the fleet's direction isn't part of the `for` loop. We want to change each alien's vertical position, but we only want to change the direction of the fleet once.

Here are the changes to `_update_aliens()`:

alien_invasion.py

```
def _update_aliens(self):
    """
    Check if the fleet is at an edge,
      then update the positions of all aliens in the fleet.
    """
    self._check_fleet_edges()
    self.aliens.update()
```

We've modified the method by calling `_check_fleet_edges()` before updating each alien's position.

When you run the game now, the fleet should move back and forth between the edges of the screen and drop down every time it hits an edge. Now we can start shooting down aliens and watch for any aliens that hit the ship or reach the bottom of the screen.

TRY IT YOURSELF

13-3. Raindrops: Find an image of a raindrop and create a grid of raindrops. Make the raindrops fall toward the bottom of the screen until they disappear.

13-4. Steady Rain: Modify your code in Exercise 13-3 so when a row of raindrops disappears off the bottom of the screen, a new row appears at the top of the screen and begins to fall.

Shooting Aliens

We've built our ship and a fleet of aliens, but when the bullets reach the aliens, they simply pass through because we aren't checking for collisions. In game programming, *collisions* happen when game elements overlap. To make the bullets shoot down aliens, we'll use the method `sprite.groupcollide()` to look for collisions between members of two groups.

Detecting Bullet Collisions

We want to know right away when a bullet hits an alien so we can make an alien disappear as soon as it's hit. To do this, we'll look for collisions immediately after updating the position of all the bullets.

The `sprite.groupcollide()` function compares the rects of each element in one group with the rects of each element in another group. In this case, it compares each bullet's `rect` with each alien's `rect` and returns a dictionary containing the bullets and aliens that have collided. Each key in the

dictionary will be a bullet, and the corresponding value will be the alien that was hit. (We'll also use this dictionary when we implement a scoring system in Chapter 14.)

Add the following code to the end of `_update_bullets()` to check for collisions between bullets and aliens:

alien_invasion.py

```
    def _update_bullets(self):
        """Update position of bullets and get rid of old bullets."""
        --snip--

        # Check for any bullets that have hit aliens.
        #   If so, get rid of the bullet and the alien.
        collisions = pygame.sprite.groupcollide(
                self.bullets, self.aliens, True, True)
```

The new code we added compares the positions of all the bullets in `self.bullets` and all the aliens in `self.aliens`, and identifies any that overlap. Whenever the rects of a bullet and alien overlap, `groupcollide()` adds a key-value pair to the dictionary it returns. The two `True` arguments tell Pygame to delete the bullets and aliens that have collided. (To make a high-powered bullet that can travel to the top of the screen, destroying every alien in its path, you could set the first Boolean argument to `False` and keep the second Boolean argument set to `True`. The aliens hit would disappear, but all bullets would stay active until they disappeared off the top of the screen.)

When you run *Alien Invasion* now, aliens you hit should disappear. Figure 13-5 shows a fleet that has been partially shot down.

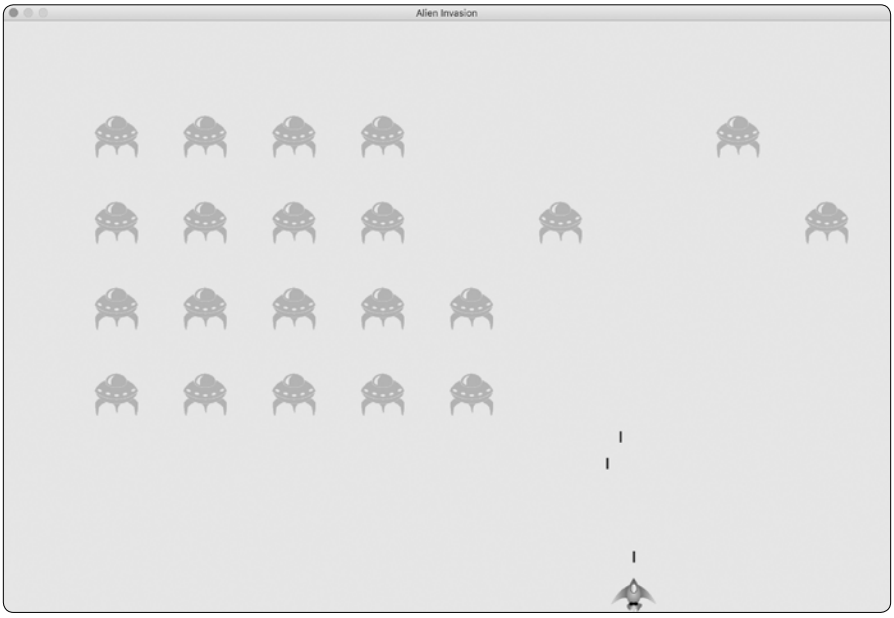


Figure 13-5: We can shoot aliens!

Making Larger Bullets for Testing

You can test many features of the game simply by running the game. But some features are tedious to test in the normal version of the game. For example, it's a lot of work to shoot down every alien on the screen multiple times to test whether your code responds to an empty fleet correctly.

To test particular features, you can change certain game settings to focus on a particular area. For example, you might shrink the screen so there are fewer aliens to shoot down or increase the bullet speed and give yourself lots of bullets at once.

My favorite change for testing *Alien Invasion* is to use really wide bullets that remain active even after they've hit an alien (see Figure 13-6). Try setting `bullet_width` to 300, or even 3000, to see how quickly you can shoot down the fleet!

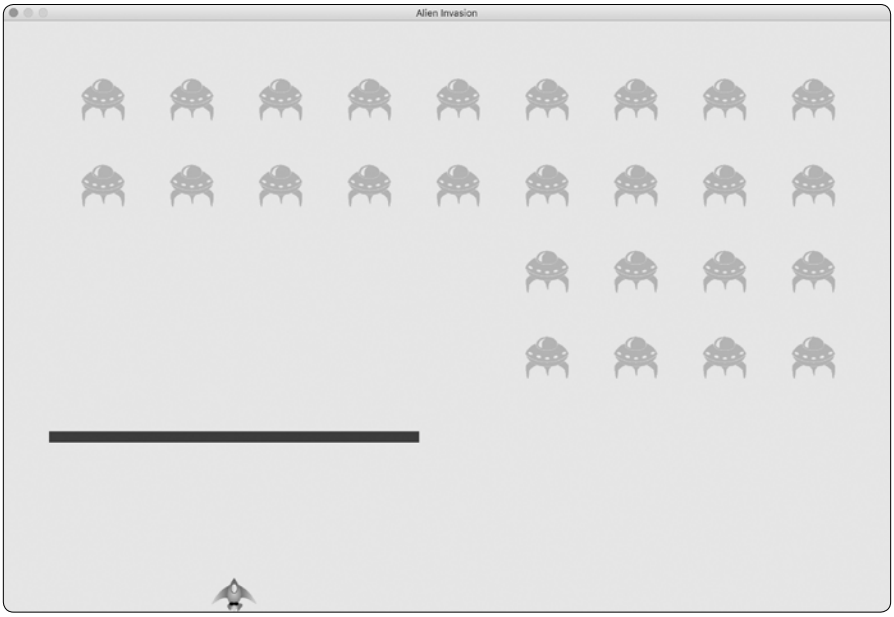


Figure 13-6: Extra-powerful bullets make some aspects of the game easier to test.

Changes like these will help you test the game more efficiently and possibly spark ideas for giving players bonus powers. Just remember to restore the settings to normal when you're finished testing a feature.

Repopulating the Fleet

One key feature of *Alien Invasion* is that the aliens are relentless: every time the fleet is destroyed, a new fleet should appear.

To make a new fleet of aliens appear after a fleet has been destroyed, we first check whether the `aliens` group is empty. If it is, we make a call to `_create_fleet()`. We'll perform this check at the end of `_update_bullets()`, because that's where individual aliens are destroyed.

alien_invasion.py

```
    def _update_bullets(self):
        --snip--
❶       if not self.aliens:
            # Destroy existing bullets and create new fleet.
❷           self.bullets.empty()
            self._create_fleet()
```

At ❶, we check whether the `aliens` group is empty. An empty group evaluates to `False`, so this is a simple way to check whether the group is empty. If it is, we get rid of any existing bullets by using the `empty()` method, which removes all the remaining sprites from a group ❷. We also call `_create_fleet()`, which fills the screen with aliens again.

Now a new fleet appears as soon as you destroy the current fleet.

Speeding Up the Bullets

If you've tried firing at the aliens in the game's current state, you might find that the bullets aren't traveling at the best speed for gameplay. They might be a little slow on your system or way too fast. At this point, you can modify the settings to make the gameplay interesting and enjoyable on your system.

We modify the speed of the bullets by adjusting the value of `bullet_speed` in *settings.py*. On my system, I'll adjust the value of `bullet_speed` to 1.5, so the bullets travel a little faster:

settings.py

```
        # Bullet settings
        self.bullet_speed = 1.5
        self.bullet_width = 3
        --snip--
```

The best value for this setting depends on your system's speed, so find a value that works for you. You can adjust other settings as well.

Refactoring _update_bullets()

Let's refactor `_update_bullets()` so it's not doing so many different tasks. We'll move the code for dealing with bullet and alien collisions to a separate method:

alien_invasion.py

```
    def _update_bullets(self):
        --snip--
        # Get rid of bullets that have disappeared.
        for bullet in self.bullets.copy():
            if bullet.rect.bottom <= 0:
                self.bullets.remove(bullet)

        self._check_bullet_alien_collisions()

    def _check_bullet_alien_collisions(self):
        """Respond to bullet-alien collisions."""
        # Remove any bullets and aliens that have collided.
```

```
        collisions = pygame.sprite.groupcollide(
                self.bullets, self.aliens, True, True)

        if not self.aliens:
            # Destroy existing bullets and create new fleet.
            self.bullets.empty()
            self._create_fleet()
```

We've created a new method, `_check_bullet_alien_collisions()`, to look for collisions between bullets and aliens, and to respond appropriately if the entire fleet has been destroyed. Doing so keeps `_update_bullets()` from growing too long and simplifies further development.

TRY IT YOURSELF

13-5. Sideways Shooter Part 2: We've come a long way since Exercise 12-6, Sideways Shooter. For this exercise, try to develop Sideways Shooter to the same point we've brought *Alien Invasion* to. Add a fleet of aliens, and make them move sideways toward the ship. Or, write code that places aliens at random positions along the right side of the screen and then sends them toward the ship. Also, write code that makes the aliens disappear when they're hit.

Ending the Game

What's the fun and challenge in a game if you can't lose? If the player doesn't shoot down the fleet quickly enough, we'll have the aliens destroy the ship when they make contact. At the same time, we'll limit the number of ships a player can use, and we'll destroy the ship when an alien reaches the bottom of the screen. The game will end when the player has used up all their ships.

Detecting Alien and Ship Collisions

We'll start by checking for collisions between aliens and the ship so we can respond appropriately when an alien hits it. We'll check for alien and ship collisions immediately after updating the position of each alien in `AlienInvasion`:

alien_invasion.py

```
    def _update_aliens(self):
        --snip--
        self.aliens.update()

        # Look for alien-ship collisions.
❶       if pygame.sprite.spritecollideany(self.ship, self.aliens):
❷           print("Ship hit!!!")
```

The `spritecollideany()` function takes two arguments: a sprite and a group. The function looks for any member of the group that has collided with the sprite and stops looping through the group as soon as it finds one member that has collided with the sprite. Here, it loops through the group `aliens` and returns the first alien it finds that has collided with `ship`.

If no collisions occur, `spritecollideany()` returns `None` and the `if` block at ❶ won't execute. If it finds an alien that has collided with the ship, it returns that alien and the `if` block executes: it prints *Ship hit!!!* ❷. When an alien hits the ship, we'll need to do a number of tasks: we'll need to delete all remaining aliens and bullets, recenter the ship, and create a new fleet. Before we write code to do all this, we need to know that our approach for detecting alien and ship collisions works correctly. Writing a `print()` call is a simple way to ensure we're detecting these collisions properly.

Now when you run *Alien Invasion,* the message *Ship hit!!!* should appear in the terminal whenever an alien runs into the ship. When you're testing this feature, set `alien_drop_speed` to a higher value, such as 50 or 100, so the aliens reach your ship faster.

Responding to Alien and Ship Collisions

Now we need to figure out exactly what will happen when an alien collides with the ship. Instead of destroying the `ship` instance and creating a new one, we'll count how many times the ship has been hit by tracking statistics for the game. Tracking statistics will also be useful for scoring.

Let's write a new class, `GameStats`, to track game statistics, and save it as *game_stats.py*:

game_stats.py

```
class GameStats:
    """Track statistics for Alien Invasion."""

    def __init__(self, ai_game):
        """Initialize statistics."""
        self.settings = ai_game.settings
        self.reset_stats()

    def reset_stats(self):
        """Initialize statistics that can change during the game."""
        self.ships_left = self.settings.ship_limit
```

We'll make one `GameStats` instance for the entire time *Alien Invasion* is running. But we'll need to reset some statistics each time the player starts a new game. To do this, we'll initialize most of the statistics in the `reset_stats()` method instead of directly in `__init__()`. We'll call this method from `__init__()` so the statistics are set properly when the `GameStats` instance is first created ❶. But we'll also be able to call `reset_stats()` any time the player starts a new game.

Right now we have only one statistic, `ships_left`, the value of which will change throughout the game. The number of ships the player starts with should be stored in *settings.py* as `ship_limit`:

settings.py

```
        # Ship settings
        self.ship_speed = 1.5
        self.ship_limit = 3
```

We also need to make a few changes in *alien_invasion.py* to create an instance of `GameStats`. First, we'll update the `import` statements at the top of the file:

alien_invasion.py

```
import sys
from time import sleep

import pygame

from settings import Settings
from game_stats import GameStats
from ship import Ship
--snip--
```

We import the `sleep()` function from the `time` module in the Python standard library so we can pause the game for a moment when the ship is hit. We also import `GameStats`.

We'll create an instance of `GameStats` in `__init__()`:

alien_invasion.py

```
    def __init__(self):
        --snip--
        self.screen = pygame.display.set_mode(
            (self.settings.screen_width, self.settings.screen_height))
        pygame.display.set_caption("Alien Invasion")

        # Create an instance to store game statistics.
        self.stats = GameStats(self)

        self.ship = Ship(self)
        --snip--
```

We make the instance after creating the game window but before defining other game elements, such as the ship.

When an alien hits the ship, we'll subtract one from the number of ships left, destroy all existing aliens and bullets, create a new fleet, and reposition the ship in the middle of the screen. We'll also pause the game for a moment so the player can notice the collision and regroup before a new fleet appears.

Let's put most of this code in a new method called `_ship_hit()`. We'll call this method from `_update_aliens()` when an alien hits the ship:

alien_invasion.py

```
    def _ship_hit(self):
        """Respond to the ship being hit by an alien."""
```

```
            # Decrement ships_left.
❶           self.stats.ships_left -= 1

            # Get rid of any remaining aliens and bullets.
❷           self.aliens.empty()
            self.bullets.empty()

            # Create a new fleet and center the ship.
❸           self._create_fleet()
            self.ship.center_ship()

            # Pause.
❹           sleep(0.5)
```

The new method `_ship_hit()` coordinates the response when an alien hits a ship. Inside `_ship_hit()`, the number of ships left is reduced by 1 at ❶, after which we empty the groups `aliens` and `bullets` ❷.

Next, we create a new fleet and center the ship ❸. (We'll add the method `center_ship()` to `Ship` in a moment.) Then we add a pause after the updates have been made to all the game elements but before any changes have been drawn to the screen, so the player can see that their ship has been hit ❹. The `sleep()` call pauses program execution for half a second, long enough for the player to see that the alien has hit the ship. When the `sleep()` function ends, code execution moves on to the `_update_screen()` method, which draws the new fleet to the screen.

In `_update_aliens()`, we replace the `print()` call with a call to `_ship_hit()` when an alien hits the ship:

alien_invasion.py

```
    def _update_aliens(self):
        --snip--
        if pygame.sprite.spritecollideany(self.ship, self.aliens):
            self._ship_hit()
```

Here's the new method `center_ship()`; add it to the end of *ship.py*:

ship.py

```
    def center_ship(self):
        """Center the ship on the screen."""
        self.rect.midbottom = self.screen_rect.midbottom
        self.x = float(self.rect.x)
```

We center the ship the same way we did in `__init__()`. After centering it, we reset the `self.x` attribute, which allows us to track the ship's exact position.

NOTE *Notice that we never make more than one ship; we make only one ship instance for the whole game and recenter it whenever the ship has been hit. The statistic* `ships_left` *will tell us when the player has run out of ships.*

Run the game, shoot a few aliens, and let an alien hit the ship. The game should pause, and a new fleet should appear with the ship centered at the bottom of the screen again.

Aliens that Reach the Bottom of the Screen

If an alien reaches the bottom of the screen, we'll have the game respond the same way it does when an alien hits the ship. To check when this happens, add a new method in *alien_invasion.py*:

alien_invasion.py

```
    def _check_aliens_bottom(self):
        """Check if any aliens have reached the bottom of the screen."""
        screen_rect = self.screen.get_rect()
        for alien in self.aliens.sprites():
❶           if alien.rect.bottom >= screen_rect.bottom:
                # Treat this the same as if the ship got hit.
                self._ship_hit()
                break
```

The method `_check_aliens_bottom()` checks whether any aliens have reached the bottom of the screen. An alien reaches the bottom when its `rect.bottom` value is greater than or equal to the screen's `rect.bottom` attribute ❶. If an alien reaches the bottom, we call `_ship_hit()`. If one alien hits the bottom, there's no need to check the rest, so we break out of the loop after calling `_ship_hit()`.

We'll call this method from `_update_aliens()`:

alien_invasion.py

```
    def _update_aliens(self):
        --snip--
        # Look for alien-ship collisions.
        if pygame.sprite.spritecollideany(self.ship, self.aliens):
            self._ship_hit()

        # Look for aliens hitting the bottom of the screen.
        self._check_aliens_bottom()
```

We call `_check_aliens_bottom()` after updating the positions of all the aliens and after looking for alien and ship collisions ❷. Now a new fleet will appear every time the ship is hit by an alien or an alien reaches the bottom of the screen.

Game Over!

Alien Invasion feels more complete now, but the game never ends. The value of `ships_left` just grows increasingly negative. Let's add a `game_active` flag as an attribute to `GameStats` to end the game when the player runs out of ships. We'll set this flag at the end of the `__init__()` method in `GameStats`:

game_stats.py

```
    def __init__(self, ai_game):
        --snip--
        # Start Alien Invasion in an active state.
        self.game_active = True
```

Now we add code to `_ship_hit()` that sets `game_active` to `False` when the player has used up all their ships:

alien_invasion.py

```
    def _ship_hit(self):
        """Respond to ship being hit by alien."""
        if self.stats.ships_left > 0:
            # Decrement ships_left.
            self.stats.ships_left -= 1
           --snip--
            # Pause.
            sleep(0.5)
        else:
            self.stats.game_active = False
```

Most of `_ship_hit()` is unchanged. We've moved all the existing code into an `if` block, which tests to make sure the player has at least one ship remaining. If so, we create a new fleet, pause, and move on. If the player has no ships left, we set `game_active` to `False`.

Identifying When Parts of the Game Should Run

We need to identify the parts of the game that should always run and the parts that should run only when the game is active:

alien_invasion.py

```
    def run_game(self):
        """Start the main loop for the game."""
        while True:
            self._check_events()

            if self.stats.game_active:
                self.ship.update()
                self._update_bullets()
                self._update_aliens()

            self._update_screen()
```

In the main loop, we always need to call `_check_events()`, even if the game is inactive. For example, we still need to know if the user presses Q to quit the game or clicks the button to close the window. We also continue updating the screen so we can make changes to the screen while waiting to see whether the player chooses to start a new game. The rest of the function calls only need to happen when the game is active, because when the game is inactive, we don't need to update the positions of game elements.

Now when you play *Alien Invasion*, the game should freeze when you've used up all your ships.

TRY IT YOURSELF

13-6. Game Over: In Sideways Shooter, keep track of the number of times the ship is hit and the number of times an alien is hit by the ship. Decide on an appropriate condition for ending the game, and stop the game when this situation occurs.

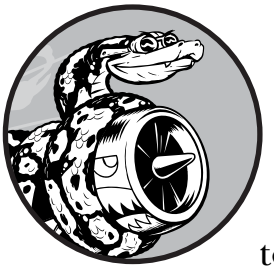
Summary

In this chapter, you learned how to add a large number of identical elements to a game by creating a fleet of aliens. You used nested loops to create a grid of elements, and you made a large set of game elements move by calling each element's `update()` method. You learned to control the direction of objects on the screen and to respond to specific situations, such as when the fleet reaches the edge of the screen. You detected and responded to collisions when bullets hit aliens and aliens hit the ship. You also learned how to track statistics in a game and use a `game_active` flag to determine when the game is over.

In the next and final chapter of this project, we'll add a Play button so the player can choose when to start their first game and whether to play again when the game ends. We'll speed up the game each time the player shoots down the entire fleet, and we'll add a scoring system. The final result will be a fully playable game!

14

SCORING



In this chapter, we'll finish the *Alien Invasion* game. We'll add a Play button to start a game on demand or to restart a game once it ends. We'll also change the game so it speeds up when the player moves up a level, and implement a scoring system. By the end of the chapter, you'll know enough to start writing games that increase in difficulty as a player progresses and show scores.

Adding the Play Button

In this section, we'll add a Play button that appears before a game begins and reappears when the game ends so the player can play again.

Right now the game begins as soon as you run *alien_invasion.py*. Let's start the game in an inactive state and then prompt the player to click a Play button to begin. To do this, modify the `__init__()` method of `GameStats`:

game_stats.py

```
    def __init__(self, ai_game):
        """Initialize statistics."""
        self.settings = ai_game.settings
        self.reset_stats()

        # Start game in an inactive state.
        self.game_active = False
```

Now the game should start in an inactive state with no way for the player to start it until we make a Play button.

Creating a Button Class

Because Pygame doesn't have a built-in method for making buttons, we'll write a `Button` class to create a filled rectangle with a label. You can use this code to make any button in a game. Here's the first part of the `Button` class; save it as *button.py*:

button.py

```
import pygame.font

class Button:

❶   def __init__(self, ai_game, msg):
        """Initialize button attributes."""
        self.screen = ai_game.screen
        self.screen_rect = self.screen.get_rect()

        # Set the dimensions and properties of the button.
❷       self.width, self.height = 200, 50
        self.button_color = (0, 255, 0)
        self.text_color = (255, 255, 255)
❸       self.font = pygame.font.SysFont(None, 48)

        # Build the button's rect object and center it.
❹       self.rect = pygame.Rect(0, 0, self.width, self.height)
        self.rect.center = self.screen_rect.center

        # The button message needs to be prepped only once.
❺       self._prep_msg(msg)
```

First, we import the `pygame.font` module, which lets Pygame render text to the screen. The `__init__()` method takes the parameters `self`, the `ai_game`

object, and `msg`, which contains the button's text ❶. We set the button dimensions at ❷, and then set `button_color` to color the button's `rect` object bright green and set `text_color` to render the text in white.

At ❸, we prepare a `font` attribute for rendering text. The `None` argument tells Pygame to use the default font, and `48` specifies the size of the text. To center the button on the screen, we create a `rect` for the button ❹ and set its `center` attribute to match that of the screen.

Pygame works with text by rendering the string you want to display as an image. At ❺, we call `_prep_msg()` to handle this rendering.

Here's the code for `_prep_msg()`:

button.py

```
    def _prep_msg(self, msg):
        """Turn msg into a rendered image and center text on the button."""
❶       self.msg_image = self.font.render(msg, True, self.text_color,
                self.button_color)
❷       self.msg_image_rect = self.msg_image.get_rect()
        self.msg_image_rect.center = self.rect.center
```

The `_prep_msg()` method needs a `self` parameter and the text to be rendered as an image (`msg`). The call to `font.render()` turns the text stored in `msg` into an image, which we then store in `self.msg_image` ❶. The `font.render()` method also takes a Boolean value to turn antialiasing on or off (antialiasing makes the edges of the text smoother). The remaining arguments are the specified font color and background color. We set antialiasing to `True` and set the text background to the same color as the button. (If you don't include a background color, Pygame will try to render the font with a transparent background.)

At ❷, we center the text image on the button by creating a `rect` from the image and setting its `center` attribute to match that of the button.

Finally, we create a `draw_button()` method that we can call to display the button onscreen:

button.py

```
    def draw_button(self):
        # Draw blank button and then draw message.
        self.screen.fill(self.button_color, self.rect)
        self.screen.blit(self.msg_image, self.msg_image_rect)
```

We call `screen.fill()` to draw the rectangular portion of the button. Then we call `screen.blit()` to draw the text image to the screen, passing it an image and the `rect` object associated with the image. This completes the `Button` class.

Drawing the Button to the Screen

We'll use the `Button` class to create a Play button in `AlienInvasion`. First, we'll update the `import` statements:

alien_invasion.py

```
--snip--
from game_stats import GameStats
from button import Button
```

Because we need only one Play button, we'll create the button in the `__init__()` method of `AlienInvasion`. We can place this code at the very end of `__init__()`:

alien_invasion.py

```
    def __init__(self):
        --snip--
        self._create_fleet()

        # Make the Play button.
        self.play_button = Button(self, "Play")
```

This code creates an instance of `Button` with the label *Play*, but it doesn't draw the button to the screen. We'll call the button's `draw_button()` method in `_update_screen()`:

alien_invasion.py

```
    def _update_screen(self):
        --snip--
        self.aliens.draw(self.screen)

        # Draw the play button if the game is inactive.
        if not self.stats.game_active:
            self.play_button.draw_button()

        pygame.display.flip()
```

To make the Play button visible above all other elements on the screen, we draw it after all the other elements have been drawn but before flipping to a new screen. We include it in an `if` block, so the button only appears when the game is inactive.

Now when you run *Alien Invasion*, you should see a Play button in the center of the screen, as shown in Figure 14-1.

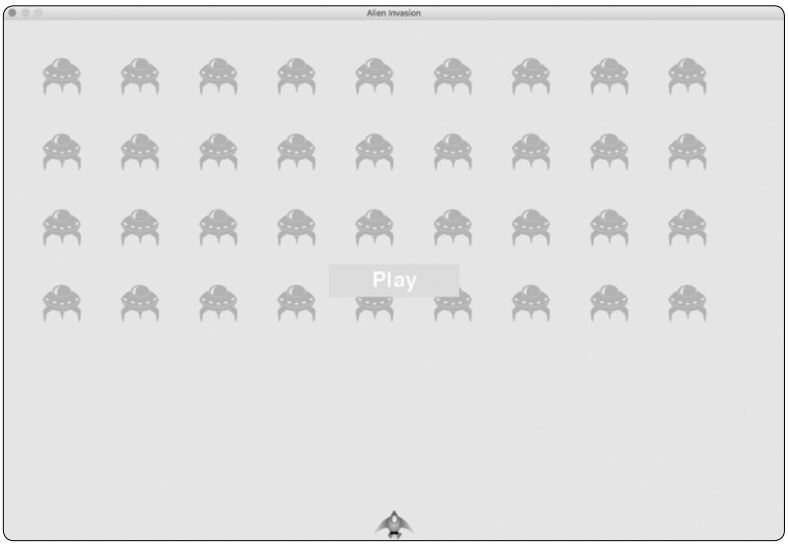


Figure 14-1: A Play button appears when the game is inactive.

Starting the Game

To start a new game when the player clicks Play, add the following `elif` block to the end of `_check_events()` to monitor mouse events over the button:

alien_invasion.py

```
    def _check_events(self):
        """Respond to keypresses and mouse events."""
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                --snip--
❶           elif event.type == pygame.MOUSEBUTTONDOWN:
❷               mouse_pos = pygame.mouse.get_pos()
❸               self._check_play_button(mouse_pos)
```

Pygame detects a `MOUSEBUTTONDOWN` event when the player clicks anywhere on the screen ❶, but we want to restrict our game to respond to mouse clicks only on the Play button. To accomplish this, we use `pygame.mouse.get_pos()`, which returns a tuple containing the mouse cursor's x- and y-coordinates when the mouse button is clicked ❷. We send these values to the new method `_check_play_button()` ❸.

Here's `_check_play_button()`, which I chose to place after `_check_events()`:

alien_invasion.py

```
    def _check_play_button(self, mouse_pos):
        """Start a new game when the player clicks Play."""
❶       if self.play_button.rect.collidepoint(mouse_pos):
            self.stats.game_active = True
```

We use the `rect` method `collidepoint()` to check whether the point of the mouse click overlaps the region defined by the Play button's `rect` ❶. If so, we set `game_active` to `True`, and the game begins!

At this point, you should be able to start and play a full game. When the game ends, the value of `game_active` should become `False` and the Play button should reappear.

Resetting the Game

The Play button code we just wrote works the first time the player clicks Play. But it doesn't work after the first game ends, because the conditions that caused the game to end haven't been reset.

To reset the game each time the player clicks Play, we need to reset the game statistics, clear out the old aliens and bullets, build a new fleet, and center the ship, as shown here:

alien_invasion.py

```
    def _check_play_button(self, mouse_pos):
        """Start a new game when the player clicks Play."""
        if self.play_button.rect.collidepoint(mouse_pos):
            # Reset the game statistics.
❶           self.stats.reset_stats()
            self.stats.game_active = True

            # Get rid of any remaining aliens and bullets.
❷           self.aliens.empty()
```

```
            self.bullets.empty()

            # Create a new fleet and center the ship.
❸           self._create_fleet()
            self.ship.center_ship()
```

At ❶, we reset the game statistics, which gives the player three new ships. Then we set `game_active` to `True` so the game will begin as soon as the code in this function finishes running. We empty the `aliens` and `bullets` groups ❷, and then create a new fleet and center the ship ❸.

Now the game will reset properly each time you click Play, allowing you to play it as many times as you want!

Deactivating the Play Button

One issue with our Play button is that the button region on the screen will continue to respond to clicks even when the Play button isn't visible. If you click the Play button area by accident after a game begins, the game will restart!

To fix this, set the game to start only when `game_active` is `False`:

alien_invasion.py

```
    def _check_play_button(self, mouse_pos):
        """Start a new game when the player clicks Play."""
❶       button_clicked = self.play_button.rect.collidepoint(mouse_pos)
❷       if button_clicked and not self.stats.game_active:
            # Reset the game statistics.
            self.stats.reset_stats()
            --snip--
```

The flag `button_clicked` stores a `True` or `False` value ❶, and the game will restart only if Play is clicked *and* the game is not currently active ❷. To test this behavior, start a new game and repeatedly click where the Play button should be. If everything works as expected, clicking the Play button area should have no effect on the gameplay.

Hiding the Mouse Cursor

We want the mouse cursor to be visible to begin play, but once play begins, it just gets in the way. To fix this, we'll make it invisible when the game becomes active. We can do this at the end of the `if` block in `_check_play_button()`:

alien_invasion.py

```
    def _check_play_button(self, mouse_pos):
        """Start a new game when the player clicks Play."""
        button_clicked = self.play_button.rect.collidepoint(mouse_pos)
        if button_clicked and not self.stats.game_active:
            --snip--
            # Hide the mouse cursor.
            pygame.mouse.set_visible(False)
```

Passing `False` to `set_visible()` tells Pygame to hide the cursor when the mouse is over the game window.

We'll make the cursor reappear once the game ends so the player can click Play again to begin a new game. Here's the code to do that:

alien_invasion.py

```
    def _ship_hit(self):
        """Respond to ship being hit by alien."""
        if self.stats.ships_left > 0:
            --snip--
        else:
            self.stats.game_active = False
            pygame.mouse.set_visible(True)
```

We make the cursor visible again as soon as the game becomes inactive, which happens in `_ship_hit()`. Attention to details like this makes your game more professional looking and allows the player to focus on playing rather than figuring out the user interface.

TRY IT YOURSELF

14-1. Press P to Play: Because *Alien Invasion* uses keyboard input to control the ship, it would be useful to start the game with a keypress. Add code that lets the player press P to start. It might help to move some code from `_check_play_button()` to a `_start_game()` method that can be called from `_check_play_button()` *and* `_check_keydown_events()`.

14-2. Target Practice: Create a rectangle at the right edge of the screen that moves up and down at a steady rate. Then have a ship appear on the left side of the screen that the player can move up and down while firing bullets at the moving, rectangular target. Add a Play button that starts the game, and when the player misses the target three times, end the game and make the Play button reappear. Let the player restart the game with this Play button.

Leveling Up

In our current game, once a player shoots down the entire alien fleet, the player reaches a new level, but the game difficulty doesn't change. Let's liven things up a bit and make the game more challenging by increasing the game's speed each time a player clears the screen.

Modifying the Speed Settings

We'll first reorganize the `Settings` class to group the game settings into static and changing ones. We'll also make sure that settings that change

during the game reset when we start a new game. Here's the `__init__()` method for *settings.py*:

settings.py

```
    def __init__(self):
        """Initialize the game's static settings."""
        # Screen settings
        self.screen_width = 1200
        self.screen_height = 800
        self.bg_color = (230, 230, 230)

        # Ship settings
        self.ship_limit = 3

        # Bullet settings
        self.bullet_width = 3
        self.bullet_height = 15
        self.bullet_color = 60, 60, 60
        self.bullets_allowed = 3

        # Alien settings
        self.fleet_drop_speed = 10

        # How quickly the game speeds up
❶       self.speedup_scale = 1.1

❷       self.initialize_dynamic_settings()
```

We continue to initialize those settings that stay constant in the `__init__()` method. At ❶, we add a `speedup_scale` setting to control how quickly the game speeds up: a value of 2 will double the game speed every time the player reaches a new level; a value of 1 will keep the speed constant. A value like 1.1 should increase the speed enough to make the game challenging but not impossible. Finally, we call the `initialize_dynamic_settings()` method to initialize the values for attributes that need to change throughout the game ❷.

Here's the code for `initialize_dynamic_settings()`:

settings.py

```
    def initialize_dynamic_settings(self):
        """Initialize settings that change throughout the game."""
        self.ship_speed = 1.5
        self.bullet_speed = 3.0
        self.alien_speed = 1.0

        # fleet_direction of 1 represents right; -1 represents left.
        self.fleet_direction = 1
```

This method sets the initial values for the ship, bullet, and alien speeds. We'll increase these speeds as the player progresses in the game and reset them each time the player starts a new game. We include `fleet_direction` in this method so the aliens always move right at the beginning of a new game. We don't need to increase the value of `fleet_drop_speed`, because when the aliens move faster across the screen, they'll also come down the screen faster.

To increase the speeds of the ship, bullets, and aliens each time the player reaches a new level, we'll write a new method called `increase_speed()`:

settings.py

```
    def increase_speed(self):
        """Increase speed settings."""
        self.ship_speed *= self.speedup_scale
        self.bullet_speed *= self.speedup_scale
        self.alien_speed *= self.speedup_scale
```

To increase the speed of these game elements, we multiply each speed setting by the value of `speedup_scale`.

We increase the game's tempo by calling `increase_speed()` in `_check_bullet_alien_collisions()` when the last alien in a fleet has been shot down:

alien_invasion.py

```
    def _check_bullet_alien_collisions(self):
        --snip--
        if not self.aliens:
            # Destroy existing bullets and create new fleet.
            self.bullets.empty()
            self._create_fleet()
            self.settings.increase_speed()
```

Changing the values of the speed settings `ship_speed`, `alien_speed`, and `bullet_speed` is enough to speed up the entire game!

Resetting the Speed

Now we need to return any changed settings to their initial values each time the player starts a new game; otherwise, each new game would start with the increased speed settings of the previous game:

alien_invasion.py

```
    def _check_play_button(self, mouse_pos):
        """Start a new game when the player clicks Play."""
        button_clicked = self.play_button.rect.collidepoint(mouse_pos)
        if button_clicked and not self.stats.game_active:
            # Reset the game settings.
            self.settings.initialize_dynamic_settings()
            --snip--
```

Playing *Alien Invasion* should be more fun and challenging now. Each time you clear the screen, the game should speed up and become slightly more difficult. If the game becomes too difficult too quickly, decrease the value of `settings.speedup_scale`. Or if the game isn't challenging enough, increase the value slightly. Find a sweet spot by ramping up the difficulty in a reasonable amount of time. The first couple of screens should be easy, the next few challenging but doable, and subsequent screens almost impossibly difficult.

TRY IT YOURSELF

14-3. Challenging Target Practice: Start with your work from Exercise 14-2 (page 285). Make the target move faster as the game progresses, and restart the target at the original speed when the player clicks Play.

14-4. Difficulty Levels: Make a set of buttons for *Alien Invasion* that allows the player to select an appropriate starting difficulty level for the game. Each button should assign the appropriate values for the attributes in `Settings` needed to create different difficulty levels.

Scoring

Let's implement a scoring system to track the game's score in real time and display the high score, level, and number of ships remaining.

The score is a game statistic, so we'll add a `score` attribute to `GameStats`:

game_stats.py

```
class GameStats:
    --snip--
    def reset_stats(self):
        """Initialize statistics that can change during the game."""
        self.ships_left = self.ai_settings.ship_limit
        self.score = 0
```

To reset the score each time a new game starts, we initialize `score` in `reset_stats()` rather than `__init__()`.

Displaying the Score

To display the score on the screen, we first create a new class, `Scoreboard`. For now, this class will just display the current score, but eventually we'll use it to report the high score, level, and number of ships remaining as well. Here's the first part of the class; save it as *scoreboard.py*:

scoreboard.py

```
import pygame.font

class Scoreboard:
    """A class to report scoring information."""

❶   def __init__(self, ai_game):
        """Initialize scorekeeping attributes."""
        self.screen = ai_game.screen
        self.screen_rect = self.screen.get_rect()
        self.settings = ai_game.settings
        self.stats = ai_game.stats

        # Font settings for scoring information.
❷       self.text_color = (30, 30, 30)
❸       self.font = pygame.font.SysFont(None, 48)
```

```
        # Prepare the initial score image.
❹       self.prep_score()
```

Because `Scoreboard` writes text to the screen, we begin by importing the `pygame.font` module. Next, we give `__init__()` the `ai_game` parameter so it can access the `settings`, `screen`, and `stats` objects, which it will need to report the values we're tracking ❶. Then we set a text color ❷ and instantiate a font object ❸.

To turn the text to be displayed into an image, we call `prep_score()` ❹, which we define here:

scoreboard.py

```
    def prep_score(self):
        """Turn the score into a rendered image."""
❶       score_str = str(self.stats.score)
❷       self.score_image = self.font.render(score_str, True,
                self.text_color, self.settings.bg_color)

        # Display the score at the top right of the screen.
❸       self.score_rect = self.score_image.get_rect()
❹       self.score_rect.right = self.screen_rect.right - 20
❺       self.score_rect.top = 20
```

In `prep_score()`, we turn the numerical value `stats.score` into a string ❶, and then pass this string to `render()`, which creates the image ❷. To display the score clearly onscreen, we pass the screen's background color and the text color to `render()`.

We'll position the score in the upper-right corner of the screen and have it expand to the left as the score increases and the width of the number grows. To make sure the score always lines up with the right side of the screen, we create a `rect` called `score_rect` ❸ and set its right edge 20 pixels from the right edge of the screen ❹. We then place the top edge 20 pixels down from the top of the screen ❺.

Then we create a `show_score()` method to display the rendered score image:

scoreboard.py

```
    def show_score(self):
        """Draw score to the screen."""
        self.screen.blit(self.score_image, self.score_rect)
```

This method draws the score image onscreen at the location `score_rect` specifies.

Making a Scoreboard

To display the score, we'll create a `Scoreboard` instance in `AlienInvasion`. First, let's update the `import` statements:

alien_invasion.py

```
--snip--
from game_stats import GameStats
from scoreboard import Scoreboard
--snip--
```

Next, we make an instance of `Scoreboard` in `__init__()`:

alien_invasion.py

```
    def __init__(self):
        --snip--
        pygame.display.set_caption("Alien Invasion")

        # Create an instance to store game statistics,
        #   and create a scoreboard.
        self.stats = GameStats(self)
        self.sb = Scoreboard(self)
        --snip--
```

Then we draw the scoreboard onscreen in `_update_screen()`:

alien_invasion.py

```
    def _update_screen(self):
        --snip--
        self.aliens.draw(self.screen)

        # Draw the score information.
        self.sb.show_score()

        # Draw the play button if the game is inactive.
        --snip--
```

We call `show_score()` just before we draw the Play button.

When you run *Alien Invasion* now, a 0 should appear at the top right of the screen. (At this point, we just want to make sure the score appears in the right place before developing the scoring system further.) Figure 14-2 shows the score as it appears before the game starts.

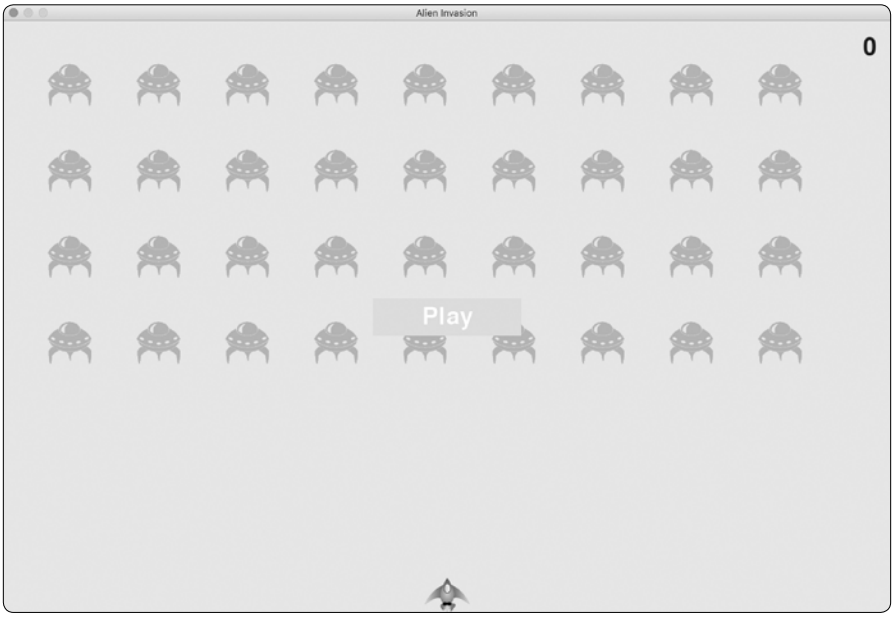


Figure 14-2: The score appears at the top-right corner of the screen.

Next, we'll assign point values to each alien!

Updating the Score as Aliens Are Shot Down

To write a live score onscreen, we update the value of `stats.score` whenever an alien is hit, and then call `prep_score()` to update the score image. But first, let's determine how many points a player gets each time they shoot down an alien:

settings.py

```
    def initialize_dynamic_settings(self):
        --snip--

        # Scoring
        self.alien_points = 50
```

We'll increase each alien's point value as the game progresses. To make sure this point value is reset each time a new game starts, we set the value in `initialize_dynamic_settings()`.

Let's update the score each time an alien is shot down in `_check_bullet_alien_collisions()`:

alien_invasion.py

```
    def _check_bullet_alien_collisions(self):
        """Respond to bullet-alien collisions."""
        # Remove any bullets and aliens that have collided.
        collisions = pygame.sprite.groupcollide(
                self.bullets, self.aliens, True, True)

        if collisions:
            self.stats.score += self.settings.alien_points
            self.sb.prep_score()
        --snip--
```

When a bullet hits an alien, Pygame returns a `collisions` dictionary. We check whether the dictionary exists, and if it does, the alien's value is added to the score. We then call `prep_score()` to create a new image for the updated score.

Now when you play *Alien Invasion*, you should be able to rack up points!

Resetting the Score

Right now, we're only prepping a new score *after* an alien has been hit, which works for most of the game. But we still see the old score when a new game starts until the first alien is hit in the new game.

We can fix this by prepping the score when starting a new game:

alien_invasion.py

```
    def _check_play_button(self, mouse_pos):
        --snip--
        if button_clicked and not self.stats.game_active:
            --snip--
            # Reset the game statistics.
            self.stats.reset_stats()
```

```
            self.stats.game_active = True
            self.sb.prep_score()
            --snip--
```

We call `prep_score()` after resetting the game stats when starting a new game. This preps the scoreboard with a 0 score.

Making Sure to Score All Hits

As currently written, our code could miss scoring for some aliens. For example, if two bullets collide with aliens during the same pass through the loop or if we make an extra-wide bullet to hit multiple aliens, the player will only receive points for hitting one of the aliens. To fix this, let's refine the way that bullet and alien collisions are detected.

In `_check_bullet_alien_collisions()`, any bullet that collides with an alien becomes a key in the `collisions` dictionary. The value associated with each bullet is a list of aliens it has collided with. We loop through the values in the `collisions` dictionary to make sure we award points for each alien hit:

alien_invasion.py

```
def _check_bullet_alien_collisions(self):
    --snip--
    if collisions:
❶       for aliens in collisions.values():
            self.stats.score += self.settings.alien_points * len(aliens)
        self.sb.prep_score()
    --snip--
```

If the `collisions` dictionary has been defined, we loop through all values in the dictionary. Remember that each value is a list of aliens hit by a single bullet. We multiply the value of each alien by the number of aliens in each list and add this amount to the current score. To test this, change the width of a bullet to 300 pixels and verify that you receive points for each alien you hit with your extra-wide bullets; then return the bullet width to its normal value.

Increasing Point Values

Because the game gets more difficult each time a player reaches a new level, aliens in later levels should be worth more points. To implement this functionality, we'll add code to increase the point value when the game's speed increases:

settings.py

```
class Settings:
    """A class to store all settings for Alien Invasion."""

    def __init__(self):
        --snip--
        # How quickly the game speeds up
        self.speedup_scale = 1.1
```

```
        # How quickly the alien point values increase
❶       self.score_scale = 1.5

        self.initialize_dynamic_settings()

    def initialize_dynamic_settings(self):
        --snip--

    def increase_speed(self):
        """Increase speed settings and alien point values."""
        self.ship_speed *= self.speedup_scale
        self.bullet_speed *= self.speedup_scale
        self.alien_speed *= self.speedup_scale

❷       self.alien_points = int(self.alien_points * self.score_scale)
```

We define a rate at which points increase, which we call `score_scale` ❶. A small increase in speed (1.1) makes the game more challenging quickly. But to see a more notable difference in scoring, we need to change the alien point value by a larger amount (1.5). Now when we increase the game's speed, we also increase the point value of each hit ❷. We use the `int()` function to increase the point value by whole integers.

To see the value of each alien, add a `print()` call to the `increase_speed()` method in `Settings`:

settings.py

```
    def increase_speed(self):
        --snip--
        self.alien_points = int(self.alien_points * self.score_scale)
        print(self.alien_points)
```

The new point value should appear in the terminal every time you reach a new level.

NOTE *Be sure to remove the `print()` call after verifying that the point value is increasing, or it might affect your game's performance and distract the player.*

Rounding the Score

Most arcade-style shooting games report scores as multiples of 10, so let's follow that lead with our scores. Also, let's format the score to include comma separators in large numbers. We'll make this change in `Scoreboard`:

scoreboard.py

```
    def prep_score(self):
        """Turn the score into a rendered image."""
❶       rounded_score = round(self.stats.score, -1)
❷       score_str = "{:,}".format(rounded_score)
        self.score_image = self.font.render(score_str, True,
                self.text_color, self.settings.bg_color)
        --snip--
```

The `round()` function normally rounds a decimal number to a set number of decimal places given as the second argument. However, when you pass a negative number as the second argument, `round()` will round the value to the nearest 10, 100, 1000, and so on. The code at ❶ tells Python to round the value of `stats.score` to the nearest 10 and store it in `rounded_score`.

At ❷, a string formatting directive tells Python to insert commas into numbers when converting a numerical value to a string: for example, to output `1,000,000` instead of `1000000`. Now when you run the game, you should see a neatly formatted, rounded score even when you rack up lots of points, as shown in Figure 14-3.

Figure 14-3: A rounded score with comma separators

High Scores

Every player wants to beat a game's high score, so let's track and report high scores to give players something to work toward. We'll store high scores in `GameStats`:

game_stats.py

```
    def __init__(self, ai_game):
        --snip--
        # High score should never be reset.
        self.high_score = 0
```

Because the high score should never be reset, we initialize `high_score` in `__init__()` rather than in `reset_stats()`.

Next, we'll modify `Scoreboard` to display the high score. Let's start with the `__init__()` method:

scoreboard.py

```
    def __init__(self, ai_game):
        --snip--
        # Prepare the initial score images.
        self.prep_score()
❶       self.prep_high_score()
```

The high score will be displayed separately from the score, so we need a new method, `prep_high_score()`, to prepare the high score image ❶.

Here's the `prep_high_score()` method:

scoreboard.py

```
    def prep_high_score(self):
        """Turn the high score into a rendered image."""
❶       high_score = round(self.stats.high_score, -1)
        high_score_str = "{:,}".format(high_score)
❷       self.high_score_image = self.font.render(high_score_str, True,
                self.text_color, self.settings.bg_color)

        # Center the high score at the top of the screen.
        self.high_score_rect = self.high_score_image.get_rect()
❸       self.high_score_rect.centerx = self.screen_rect.centerx
❹       self.high_score_rect.top = self.score_rect.top
```

We round the high score to the nearest 10 and format it with commas ❶. We then generate an image from the high score ❷, center the high score `rect` horizontally ❸, and set its `top` attribute to match the top of the score image ❹.

The `show_score()` method now draws the current score at the top right and the high score at the top center of the screen:

scoreboard.py

```
    def show_score(self):
        """Draw score to the screen."""
        self.screen.blit(self.score_image, self.score_rect)
        self.screen.blit(self.high_score_image, self.high_score_rect)
```

To check for high scores, we'll write a new method, `check_high_score()`, in `Scoreboard`:

scoreboard.py

```
    def check_high_score(self):
        """Check to see if there's a new high score."""
        if self.stats.score > self.stats.high_score:
            self.stats.high_score = self.stats.score
            self.prep_high_score()
```

The method `check_high_score()` checks the current score against the high score. If the current score is greater, we update the value of `high_score` and call `prep_high_score()` to update the high score's image.

We need to call `check_high_score()` each time an alien is hit after updating the score in `_check_bullet_alien_collisions()`:

alien_invasion.py

```
def _check_bullet_alien_collisions(self):
    --snip--
    if collisions:
        for aliens in collisions.values():
            self.stats.score += self.settings.alien_points * len(aliens)
        self.sb.prep_score()
        self.sb.check_high_score()
    --snip--
```

We call `check_high_score()` when the `collisions` dictionary is present, and we do so after updating the score for all the aliens that have been hit.

The first time you play *Alien Invasion*, your score will be the high score, so it will be displayed as the current score and the high score. But when you start a second game, your high score should appear in the middle and your current score at the right, as shown in Figure 14-4.

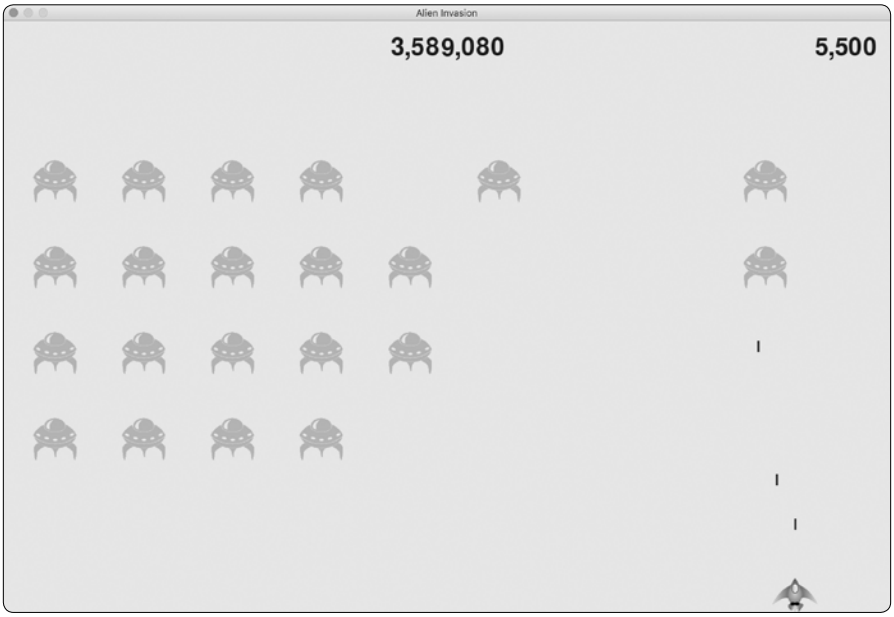


Figure 14-4: The high score is shown at the top center of the screen.

Displaying the Level

To display the player's level in the game, we first need an attribute in `GameStats` representing the current level. To reset the level at the start of each new game, initialize it in `reset_stats()`:

game_stats.py

```
def reset_stats(self):
    """Initialize statistics that can change during the game."""
    self.ships_left = self.settings.ship_limit
```

```
        self.score = 0
        self.level = 1
```

To have Scoreboard display the current level, we call a new method, prep_level(), from __init__():

scoreboard.py

```
    def __init__(self, ai_game):
        --snip--
        self.prep_high_score()
        self.prep_level()
```

Here's prep_level():

scoreboard.py

```
    def prep_level(self):
        """Turn the level into a rendered image."""
        level_str = str(self.stats.level)
❶       self.level_image = self.font.render(level_str, True,
                self.text_color, self.settings.bg_color)

        # Position the level below the score.
        self.level_rect = self.level_image.get_rect()
❷       self.level_rect.right = self.score_rect.right
❸       self.level_rect.top = self.score_rect.bottom + 10
```

The prep_level() method creates an image from the value stored in stats.level ❶ and sets the image's right attribute to match the score's right attribute ❷. It then sets the top attribute 10 pixels beneath the bottom of the score image to leave space between the score and the level ❸.

We also need to update show_score():

scoreboard.py

```
    def show_score(self):
        """Draw scores and level to the screen."""
        self.screen.blit(self.score_image, self.score_rect)
        self.screen.blit(self.high_score_image, self.high_score_rect)
        self.screen.blit(self.level_image, self.level_rect)
```

This new line draws the level image to the screen.

We'll increment stats.level and update the level image in _check_bullet_alien_collisions():

alien_invasion.py

```
    def _check_bullet_alien_collisions(self):
        --snip--
        if not self.aliens:
            # Destroy existing bullets and create new fleet.
            self.bullets.empty()
            self._create_fleet()
            self.settings.increase_speed()

            # Increase level.
            self.stats.level += 1
            self.sb.prep_level()
```

If a fleet is destroyed, we increment the value of `stats.level` and call `prep_level()` to make sure the new level displays correctly.

To ensure the level image updates properly at the start of a new game, we also call `prep_level()` when the player clicks the Play button:

alien_invasion.py

```
def _check_play_button(self, mouse_pos):
    --snip--
    if button_clicked and not self.stats.game_active:
        --snip--
        self.sb.prep_score()
        self.sb.prep_level()
        --snip--
```

We call `prep_level()` right after calling `prep_score()`.

Now you'll see how many levels you've completed, as shown in Figure 14-5.

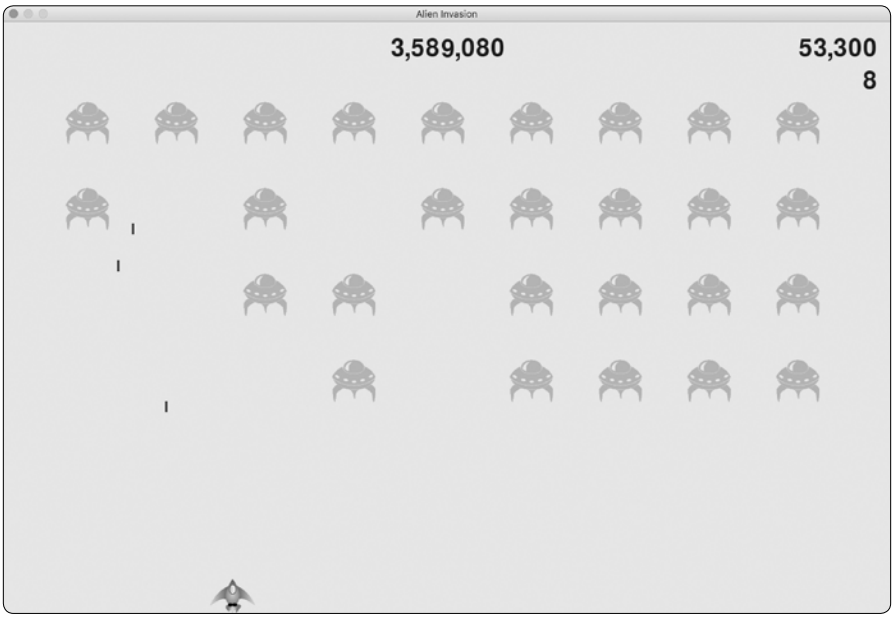


Figure 14-5: The current level appears just below the current score.

NOTE *In some classic games, the scores have labels, such as Score, High Score, and Level. We've omitted these labels because the meaning of each number becomes clear once you've played the game. To include these labels, add them to the score strings just before the calls to* `font.render()` *in* `Scoreboard`.

Displaying the Number of Ships

Finally, let's display the number of ships the player has left, but this time, let's use a graphic. To do so, we'll draw ships in the upper-left corner of

the screen to represent how many ships are left, just as many classic arcade games do.

First, we need to make `Ship` inherit from `Sprite` so we can create a group of ships:

ship.py

```
import pygame
from pygame.sprite import Sprite

❶ class Ship(Sprite):
    """A class to manage the ship."""

    def __init__(self, ai_game):
        """Initialize the ship and set its starting position."""
❷       super().__init__()
        --snip--
```

Here we import `Sprite`, make sure `Ship` inherits from `Sprite` ❶, and call `super()` at the beginning of `__init__()` ❷.

Next, we need to modify `Scoreboard` to create a group of ships we can display. Here are the `import` statements for `Scoreboard`:

scoreboard.py

```
import pygame.font
from pygame.sprite import Group

from ship import Ship
```

Because we're making a group of ships, we import the `Group` and `Ship` classes.

Here's `__init__()`:

scoreboard.py

```
    def __init__(self, ai_game):
        """Initialize scorekeeping attributes."""
        self.ai_game = ai_game
        self.screen = ai_game.screen
        --snip--
        self.prep_level()
        self.prep_ships()
```

We assign the game instance to an attribute, because we'll need it to create some ships. We call `prep_ships()` after the call to `prep_level()`.

Here's `prep_ships()`:

scoreboard.py

```
    def prep_ships(self):
        """Show how many ships are left."""
❶       self.ships = Group()
❷       for ship_number in range(self.stats.ships_left):
            ship = Ship(self.ai_game)
❸           ship.rect.x = 10 + ship_number * ship.rect.width
❹           ship.rect.y = 10
❺           self.ships.add(ship)
```

The `prep_ships()` method creates an empty group, `self.ships`, to hold the ship instances ❶. To fill this group, a loop runs once for every ship the player has left ❷. Inside the loop, we create a new ship and set each ship's x-coordinate value so the ships appear next to each other with a 10-pixel margin on the left side of the group of ships ❸. We set the y-coordinate value 10 pixels down from the top of the screen so the ships appear in the upper-left corner of the screen ❹. Then we add each new ship to the group `ships` ❺.

Now we need to draw the ships to the screen:

scoreboard.py

```
    def show_score(self):
        """Draw scores, level, and ships to the screen."""
        self.screen.blit(self.score_image, self.score_rect)
        self.screen.blit(self.high_score_image, self.high_score_rect)
        self.screen.blit(self.level_image, self.level_rect)
        self.ships.draw(self.screen)
```

To display the ships on the screen, we call `draw()` on the group, and Pygame draws each ship.

To show the player how many ships they have to start with, we call `prep_ships()` when a new game starts. We do this in `_check_play_button()` in `AlienInvasion`:

alien_invasion.py

```
    def _check_play_button(self, mouse_pos):
        --snip--
        if button_clicked and not self.stats.game_active:
            --snip--
            self.sb.prep_score()
            self.sb.prep_level()
            self.sb.prep_ships()
            --snip--
```

We also call `prep_ships()` when a ship is hit to update the display of ship images when the player loses a ship:

alien_invasion.py

```
    def _ship_hit(self):
        """Respond to ship being hit by alien."""
        if self.stats.ships_left > 0:
            # Decrement ships_left, and update scoreboard.
            self.stats.ships_left -= 1
            self.sb.prep_ships()
            --snip--
```

We call `prep_ships()` after decreasing the value of `ships_left`, so the correct number of ships displays each time a ship is destroyed.

Figure 14-6 shows the complete scoring system with the remaining ships displayed at the top left of the screen.

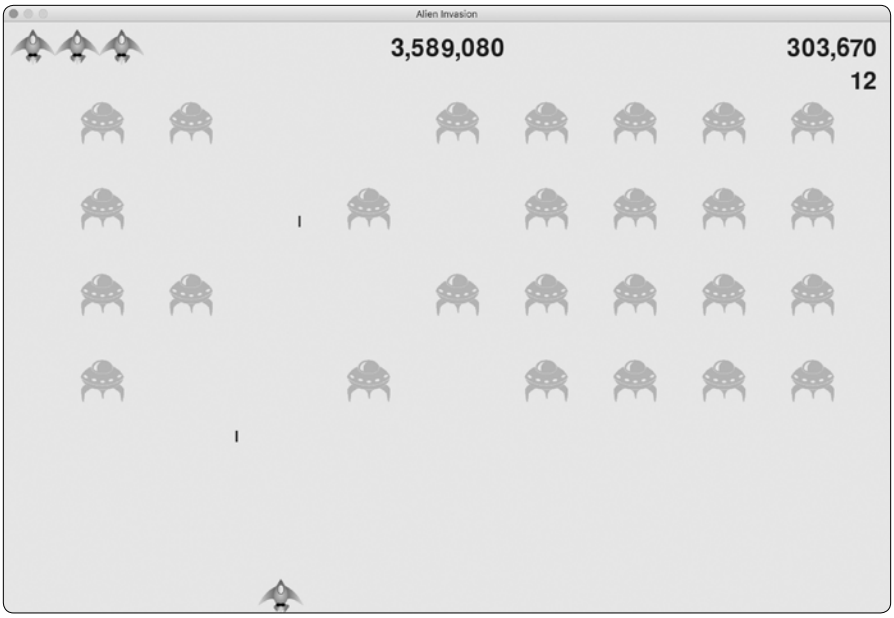


Figure 14-6: The complete scoring system for Alien Invasion

TRY IT YOURSELF

14-5. All-Time High Score: The high score is reset every time a player closes and restarts *Alien Invasion*. Fix this by writing the high score to a file before calling `sys.exit()` and reading in the high score when initializing its value in `GameStats`.

14-6. Refactoring: Look for methods that are doing more than one task, and refactor them to organize your code and make it efficient. For example, move some of the code in `_check_bullet_alien_collisions()`, which starts a new level when the fleet of aliens has been destroyed, to a function called `start_new_level()`. Also, move the four separate method calls in the `__init__()` method in `Scoreboard` to a method called `prep_images()` to shorten `__init__()`. The `prep_images()` method could also help simplify `_check_play_button()` or `start_game()` if you've already refactored `_check_play_button()`.

NOTE *Before attempting to refactor the project, see Appendix D to learn how to restore the project to a working state if you introduce bugs while refactoring.*

(continued)

14-7. Expanding the Game: Think of a way to expand *Alien Invasion*. For example, you could program the aliens to shoot bullets down at the ship or add shields for your ship to hide behind, which can be destroyed by bullets from either side. Or use something like the `pygame.mixer` module to add sound effects, such as explosions and shooting sounds.

14-8. Sideways Shooter, Final Version: Continue developing Sideways Shooter, using everything we've done in this project. Add a Play button, make the game speed up at appropriate points, and develop a scoring system. Be sure to refactor your code as you work, and look for opportunities to customize the game beyond what was shown in this chapter.

Summary

In this chapter, you learned how to implement a Play button to start a new game, detect mouse events, and hide the cursor in active games. You can use what you've learned to create other buttons in your games, like a Help button to display instructions on how to play. You also learned how to modify the speed of a game as it progresses, implement a progressive scoring system, and display information in textual and nontextual ways.