Projekt 1

Alien Invasion

12
Das eigene Kampfschiff

In diesem und den beiden folgenden Kapiteln wollen wir ein Spiel namens *Alien Invasion* schreiben. Dazu verwenden wir Pygame, eine Zusammenstellung von leistungsfähigen Python-Modulen für den Umgang mit Grafik, Animation und sogar Ton, die es leichter macht, anspruchsvolle Spiele zu gestalten. Da sich Pygame um Aufgaben wie die Ausgabe von Bildern auf dem Monitor kümmert, brauchen Sie nicht mühselig Code dafür zu schreiben, sondern können sich auf die eigentliche Logik des Spielablaufs konzentrieren.

In diesem Kapitel richten Sie Pygame ein und erstellen ein Raumschiff, das sich in Reaktion auf die Eingaben des Spielers nach rechts und links bewegt und Geschosse abfeuert. In den nächsten beiden Kapiteln fügen Sie eine Flotte außerirdischer Raumschiffe hinzu, die es abzuschießen gilt, und nehmen Verbesserungen vor, indem Sie etwa die Anzahl der Schiffe beschränken, die dem Spieler zur Verfügung stehen, und den Punktestand ausgeben.

Anhand dieses Kapitels lernen Sie auch, wie Sie umfangreiche Projekte verwalten, die mehrere Dateien umspannen. Wir führen auch eine Menge Refactor-

ings durch und kümmern uns um die Gliederung der Dateien, um das Projekt zu strukturieren und den Code effizient zu gestalten.

Die Gestaltung eines Spiels ist eine ideale Möglichkeit, um eine Programmiersprache zu lernen und gleichzeitig Spaß dabei zu haben. Es ist ein erhebendes Gefühl, zu sehen, wie sich andere Personen in ein Spiel vertiefen, das Sie geschrieben haben. Ein einfaches Spiel zu erstellen gibt Ihnen auch einen Einblick darin, wie professionelle Programmierer Spiele entwickeln. Während Sie dieses Kapitel durcharbeiten, sollten Sie den vorgestellten Code jeweils eingeben und ausführen, um zu verstehen, wie die einzelnen Blöcke zum Spielablauf beitragen. Experimentieren Sie ruhig mit unterschiedlichen Werten und Einstellungen, um ein besseres Verständnis dafür zu gewinnen, wie Sie die Interaktion in Ihren Spielen weiterentwickeln können.

Hinweis

Das Projekt *Alien Invasion* umfasst mehrere Dateien. Daher sollten Sie auf Ihrem System einen eigenen Ordner namens *alien_invasion* anlegen und sämtliche Dateien für dieses Projekt darin speichern, damit die `import`-Anweisungen korrekt funktionieren.

Wenn Sie sich schon dazu bereit fühlen, eine Versionssteuerung zu verwenden, können Sie das bei diesem Projekt tun. Einen Überblick über Versionssteuerung erhalten Sie in Anhang D.

Das Projekt planen

Bei einem umfangreichen Projekt ist es wichtig, einen Plan aufzustellen, bevor Sie damit beginnen, Code zu schreiben. Dieser Plan hilft Ihnen dabei, sich nicht zu verzetteln, und erhöht die Wahrscheinlichkeit dafür, dass Sie das Projekt auch fertigstellen.

Wir schreiben daher zunächst eine allgemeine Beschreibung des Spielablaufs. Sie deckt zwar nicht alle Einzelheiten von *Alien Invasion* ab, vermittelt aber eine gute Vorstellung davon, wie wir beim Schreiben dieses Spiels vorgehen müssen:

In Alien Invasion steuert der Spieler ein Raumschiff, das mittig am unteren Bildschirmrand erscheint. Er kann das Schiff mit den Pfeiltasten nach rechts und links verschieben und mithilfe der Leertaste Geschosse abfeuern. Zu Spielbeginn füllt eine Flotte außerirdischer Raumschiffe den Himmel aus und bewegt sich auf dem Bildschirm seitwärts und nach unten. Der Spieler schießt, um die gegnerischen Schiffe zu zerstören. Wenn er alle abgeschossen hat, erscheint eine neue Flotte, die sich schneller bewegt als die vorhergehende. Stößt ein außerirdisches Schiff auf das Schiff des Spielers oder erreicht es den unteren Bildschirmrand, verliert der Spieler ein Schiff. Hat der Spieler drei Schiffe verloren, endet das Spiel.

In der ersten Entwicklungsphase erstellen wir ein Schiff, das sich nach rechts und links bewegen kann und Geschosse abfeuert, wenn der Spieler die Leertaste drückt. Nachdem wir dieses Verhalten eingerichtet haben, wenden wir uns den Außerirdischen zu und nehmen Verbesserungen am Spielablauf vor.

Pygame installieren

Bevor Sie damit beginnen, Code zu schreiben, müssen Sie zunächst Pygame installieren. Das Modul `pip` hilft Ihnen beim Herunterladen und Installieren von Python-Paketen. Geben Sie an einer Terminal-Eingabeaufforderung folgenden Befehl ein:

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

Dieser Befehl weist Python an, das Modul `pip` auszuführen und das Paket `pygame` in der Python-Installation des aktuellen Benutzers zu installieren. Wenn Sie einen anderen Befehl als `python` verwenden, um Programme auszuführen oder eine Terminalsitzung zu starten, etwa `python3`, müssen Sie Folgendes eingeben:

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

Hinweis

Falls dieser Befehl auf macOS nicht funktioniert, versuchen Sie ihn ohne das Flag `--user` auszuführen.

Erste Schritte für das Spielprojekt

Jetzt können wir damit beginnen, unser Spiel zu gestalten. Dazu legen wir als Erstes ein leeres Pygame-Fenster an, in das wir später die Spielelemente wie unser eigenes Schiff und die der Außerirdischen zeichnen. Außerdem sorgen wir dafür, dass das Spiel auf unsere Eingaben reagiert, legen die Hintergrundfarbe fest und laden ein Bild unseres Schiffes.

Ein Pygame-Fenster anlegen und auf Benutzereingaben reagieren

Wir erstellen ein leeres Pygame-Fenster, indem wir eine Klasse zur Darstellung des Spiels schreiben. Legen Sie in Ihrem Texteditor eine neue Datei an, speichern Sie sie als *alien_invasion.py* und geben Sie Folgendes ein:

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:
            # Lauscht auf Tastatur- und Mausereignisse.
❹           for event in pygame.event.get():
❺               if event.type == pygame.QUIT:
                    sys.exit()

            # Macht den zuletzt gezeichneten Bildschirm sichtbar.
❻           pygame.display.flip()

if __name__ == '__main__':
    # Erstellt eine Spielinstanz und führt das Spiel aus.
    ai = AlienInvasion()
    ai.run_game()
```

Als Erstes importieren wir das Modul `pygame`, das alles enthält, was wir zur Entwicklung eines Spiels benötigen, sowie das Modul `sys`, das wir brauchen, damit der Spieler das Spiel beenden kann.

Am Anfang von *Alien Invasion* steht die Klasse `AlienInvasion`. In der Methode `__init__()` initialisiert die Funktion `pygame.init()` die Hintergrundeinstellungen, die Pygame benötigt, um korrekt laufen zu können (❶). Bei ❷ rufen wir `pygame.display.set_mode()` auf, um ein Fenster anzuzeigen, in das wir alle grafischen Elemente des Spiels zeichnen. Dabei übergeben wir als Argument das Tupel `(1200, 800)`, um die Abmessungen des Spielfensters auf eine Breite von 1200 Pixel und eine Höhe von 800 Pixel festzulegen. (Passen Sie diese Abmessungen an die Größe Ihres Bildschirms an.) Dieses Fenster weisen wir dem Attribut `self.screen` zu, sodass es in allen Methoden der Klasse zur Verfügung steht.

Das zu `self.screen` zugewiesene Objekt ist eine *Oberfläche*. In Pygame werden die Teile des Bildschirms, in denen ein Spielelement angezeigt wird, Oberflächen genannt. Alle grafischen Elemente, auch die außerirdischen oder Ihre eigenen Schiffe, sind Oberflächen. Die von `display.set_mode()` zurückgegebene Oberfläche

stellt das gesamte Spielfenster dar. Wenn wir die Animationsschleife des Spiels starten, wird diese Oberfläche bei jedem Durchlauf neu gezeichnet, sodass sie mit allen von den Eingaben des Benutzers ausgelösten Änderungen aktualisiert werden kann.

Gesteuert wird das Spiel durch die Methode `run_game()`. Sie enthält eine kontinuierlich ausgeführte `while`-Schleife (❸), die eine Ereignisschleife sowie Code für die Aktualisierung des Bildschirms enthält. Ein *Ereignis* ist eine Aktion, die der Benutzer während des Spiels vornimmt, z. B. die Betätigung einer Taste oder eine Mausbewegung. Damit unser Programm auf solche Aktionen reagieren kann, schreiben wir eine *Ereignisschleife* (❹), die auf Ereignisse *lauscht* und je nach der Art des aufgetretenen Ereignisses bestimmte Maßnahmen ausführt.

Um die von Pygame erkannten Ereignisse abzurufen, verwenden wir die Methode `pygame.event.get()`. Sie gibt eine Liste der Ereignisse zurück, die seit dem letzten Aufruf der Funktion stattgefunden haben. Jedes Tastatur- und jedes Mausereignis führt dazu, dass die `for`-Schleife ausgeführt wird. Innerhalb der Schleife verwenden wir eine Folge von `if`-Anweisungen, um die einzelnen Ereignisse zu identifizieren und darauf zu reagieren. Klickt der Spieler beispielsweise auf die Schließen-Schaltfläche des Spielfensters, wird das Ereignis `pygame.QUIT` erkannt, weshalb wir `sys.exit()` aufrufen, um das Spiel zu beenden (❺).

Der Aufruf von `pygame.display.flip()` bei ❻ weist Pygame an, den zuletzt gezeichneten Bildschirm sichtbar zu machen. Hierzu wird bei jedem Durchlauf der `while`-Schleife ein leerer Bildschirm gezeichnet und der alte Bildschirm gelöscht, sodass nur der neue zu sehen ist. Wenn wir Spielelemente verschieben, sorgt `pygame.display.flip()` dafür, dass die Anzeige ständig aktualisiert wird und jeweils die neuesten Positionen der Elemente darstellt und die alten verbirgt. Dadurch entsteht die Illusion einer flüssigen Bewegung.

Am Ende der Datei legen wir eine Instanz des Spiels an und rufen `run_game()` auf. Dabei stellen wir die Funktion in einen `if`-Block, sodass sie nur ausgeführt wird, wenn die Datei direkt aufgerufen wird. Wenn Sie die Datei *alien_invasion.py* jetzt ausführen, sehen Sie ein leeres Pygame-Fenster.

Die Hintergrundfarbe festlegen

Standardmäßig zeigt Pygame einen schwarzen Bildschirm an, aber das sieht langweilig aus. Daher wollen wir eine andere Hintergrundfarbe festlegen:

alien_invasion.py

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

        # Legt die Hintergrundfarbe fest.
❶       self.bg_color = (230, 230, 230)
```

```
    def run_game(self):
        -- schnipp --
            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    sys.exit()

            # Zeichnet den Bildschirm bei jedem Schleifendurchlauf neu.
❷           self.screen.fill(self.bg_color)

            # Macht den zuletzt gezeichneten Bildschirm sichtbar.
            pygame.display.flip()
```

In Pygame werden Farben im RGB-System definiert, also als eine Mischung aus Rot, Grün und Blau. Die Werte der einzelnen Farbanteile reichen dabei jeweils von 0 bis 255. Der Farbwert `(255, 0, 0)` steht für Rot, `(0, 255, 0)` für Grün und `(0, 0, 255)` für Blau. Durch die Mischung von RGB-Werten können Sie 16 Millionen Farben erzeugen. Bei der Angabe `(230, 230, 230)` werden Rot, Grün und Blau in gleichen Anteilen gemischt, was ein helles Grau ergibt. Diese Farbe weisen wir bei ❶ `self.bg_color` zu.

Bei ❷ füllen wir den Bildschirm mithilfe von `fill()` mit der Hintergrundfarbe. Diese Methode nimmt nur ein einziges Argument an, nämlich eine Farbe.

Eine Klasse für Einstellungen anlegen

Jedes Mal, wenn wir unser Spiel um eine neue Funktionalität erweitern, fügen wir gewöhnlich auch neue Einstellungen hinzu. Anstatt diese Einstellungen über den Code zu verteilen, schreiben wir das Modul `settings` mit der Klasse `Settings`, um alle Einstellungen zentral zu speichern. Dadurch müssen wir nicht viele einzelne Einstellungen weitergeben, sondern nur ein einziges Einstellungsobjekt. Das vereinfacht unsere Funktionsaufrufe. Außerdem erleichtert es spätere Änderungen am Erscheinungsbild des Spiels, da wir dann nicht mehr alle Dateien nach verschiedenen Einstellungen durchsuchen, sondern nur noch gezielt einige Werte in *settings.py* umstellen müssen.

Erstellen Sie eine neue Datei namens *settings.py* innerhalb Ihres Ordners *alien_invasion* und fügen Sie folgenden ersten Entwurf der Klasse `Settings` hinzu:

settings.py

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

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

Um eine Instanz von `Settings` zu bilden und für den Zugriff auf die Einstellungen zu nutzen, ändern wir *alien_invasion.py* wie folgt ab:

alien_invasion.py

```
-- schnipp --
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):
            -- schnipp --
            # Zeichnet den Bildschirm bei jedem Schleifendurchlauf neu.
❸           self.screen.fill(self.settings.bg_color)

            # Macht den zuletzt gezeichneten Bildschirm sichtbar.
            pygame.display.flip()
-- schnipp --
```

Wir importieren `Settings` in die Hauptprogrammdatei und legen nach dem Aufruf von `pygame.init()` eine Instanz davon an, die wir `self.settings` zuweisen (❶). Wenn wir nun bei ❷ das Bildschirmobjekt erstellen, verwenden wir die Attribute `screen_width` und `screen_height` von `self.settings`. Auch beim Füllen des Bildschirms greifen wir auf die in `self.settings` festgelegte Hintergrundfarbe zurück (❸).

Wenn Sie *alien_invasion.py* jetzt ausführen, werden Sie keinen Unterschied bemerken, da wir lediglich die bereits verwendeten Einstellungen verschoben haben. Als Nächstes werden wir jedoch einige neue Elemente auf dem Bildschirm hinzufügen.

Das Bild eines Raumschiffs hinzufügen

Als Nächstes fügen wir das Raumschiff des Spielers hinzu. Um es auf dem Bildschirm anzuzeigen, laden wir ein Bild und zeichnen dieses mit der Pygame-Methode `blit()` auf den Bildschirm.

Bei der Auswahl von Grafiken für Ihre Spiele müssen Sie auf die Rechte achten. Die sicherste und billigste Möglichkeit für Anfänger besteht darin, Grafiken zu verwenden, die lizenzfrei angeboten werden und das Recht einschließen, sie zu verändern. Finden können Sie so etwas auf Websites wie beispielsweise *https://pixabay.com/*.

In einem Spiel können Sie fast jede Art von Bilddatei verwenden, aber am einfachsten ist es, Bitmap-Dateien (*.bmp*) zu nehmen, da Pygame standardmäßig Grafiken dieser Art lädt. Zwar können Sie Pygame auch für andere Dateitypen konfigurieren, aber oft müssen dazu noch zusätzliche Bibliotheken zur Bildbearbeitung auf Ihrem Computer installiert sein. Die meisten angebotenen Bilder liegen in den Formaten *.jpg* und *.png* vor, aber mit Programmen wie Photoshop, GIMP und Paint können Sie sie in Bitmaps umwandeln.

Achten Sie besonders auf die Farbe des Hintergrunds in dem ausgewählten Bild. Versuchen Sie nach Möglichkeit, eine Datei mit transparentem oder einfarbigem Hintergrund zu finden, den Sie dann in einem Bildbearbeitungsprogramm durch die gewünschte Hintergrundfarbe ersetzen können. Ihre Spiele sind optisch ansprechender, wenn die Hintergrundfarbe des Bildes der Hintergrundfarbe des Spielfelds entspricht. Alternativ können Sie auch die Spielfeldfarbe an den Hintergrund des Bildes anpassen.

Für *Alien Invasion* können Sie die Datei *ship.bmp* (Abb. 12–1) verwenden, die auf der Begleitwebsite zu diesem Buch auf *www.dpunkt.de/python3crashcourse* zu finden ist. Die Hintergrundfarbe dieses Bildes entspricht den Einstellungen, die wir in diesem Projekt vorgenommen haben. Legen Sie innerhalb des Hauptprojektordners *alien_invasion* den Ordner *images* an und speichern Sie *ship.bmp* darin ab.

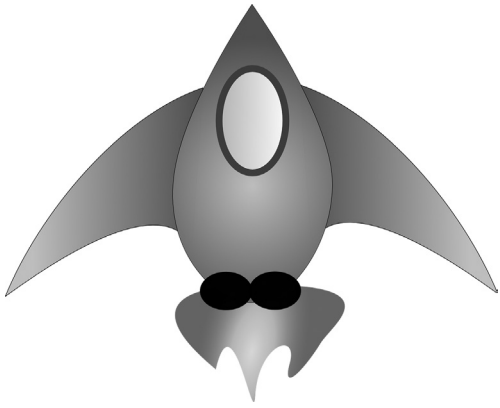

Abb. 12–1 *Das Raumschiff für Alien Invasion*

Die Klasse Ship

Um das Raumschiff im Spiel zu verwenden, schreiben wir das Modul `ship` mit der Klasse `Ship`, die das Verhalten des Spielerraumschiffs bestimmt.

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()

        # Lädt das Bild des Schiffes und ruft dessen umgebendes Rechteck ab.
❸       self.image = pygame.image.load('images/ship.bmp')
        self.rect = self.image.get_rect()

        # Platziert jedes neue Schiff mittig am unteren Bildschirmrand.
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 arbeitet sehr effizient, da es Sie alle Spielelemente wie Rechtecke (»rectangles«) behandeln lässt, auch wenn sie nicht wie Rechtecke geformt sind. Das ist eine praktische Vorgehensweise, da Rechtecke sehr einfache geometrische Formen sind. Wenn Pygame etwa herausfinden muss, ob zwei Spielelemente kollidiert sind, geht das viel schneller, wenn diese Objekte als Rechtecke aufgefasst werden. Das funktioniert gewöhnlich so gut, dass die Spieler gar nicht bemerken, dass das Programm nicht die wahren Formen der einzelnen Elemente betrachtet. In dieser Klasse behandeln wir daher sowohl das Raumschiff als auch den Bildschirm als Rechtecke.

Als Erstes importieren wir wieder das Modul `pygame`. Die `__init__()`-Methode von `Ship` nimmt zwei Parameter entgegen, nämlich den Selbstverweis `self` und einen Verweis auf die aktuelle Instanz der Klasse `AlienInvasion`. Dadurch erhält das Schiff Zugriff auf alle in dieser Klasse definierten Ressourcen. Bei ❶ weisen wir den Bildschirm einem Attribut von `Ship` zu, damit wir in allen Methoden dieser Klasse ohne Schwierigkeiten darauf zugreifen können. Anschließend greifen wir bei ❷ mit der Methode `get_rect()` auf das Attribut `rect` des Bildschirms zu und weisen es `self.screen_rect` zu. Dadurch können wir das Schiff an der gewünschten Stelle auf dem Bildschirm platzieren.

Um das Bild zu laden, rufen wir die Funktion `pygame.image.load()` auf (❸) und übergeben ihr den gewünschten Standort des Schiffes. Die Funktion gibt eine Oberfläche für das Schiff zurück, die wir `self.image` zuweisen. Anschließend greifen wir mithilfe von `get_rect()` auf das Attribut `rect` der Oberfläche zu, sodass wir es später zur Platzierung des Schiffes verwenden können.

Bei der Arbeit mit einem `rect`-Objekt können Sie die x- und y-Koordinaten des oberen, unteren, linken und rechten Randes sowie des Mittelpunkts verwenden. Wenn Sie einen dieser Werte festlegen, bestimmen Sie damit die aktuelle Position des Rechtecks. Um ein Spielelement zentriert zu platzieren, müssen Sie das Attribut `center`, `centerx` oder `centery` des Rechtecks verwenden. Wollen Sie es dagegen an einem Bildschirmrand ausrichten, müssen Sie das Attribut `top`, `bottom`, `left` oder `right` nutzen. Es gibt auch Attribute, die diese Eigenschaften kombinieren, z. B. `midbottom`, `midtop`, `midleft` und `midright`. Um die horizontale oder vertikale Position des Rechtecks anzupassen, können Sie einfach die Attribute `x` und `y` verwenden, bei denen es sich um die x- und y-Koordinaten der oberen linken Ecke des Rechtecks handelt. Die Nutzung dieser Attribute erspart Ihnen die Berechnungen, die Spieleentwickler früher manuell durchführen mussten.

Hinweis

In Pygame liegt der Koordinatenursprung `(0, 0)` in der oberen linken Ecke des Bildschirms, wobei die Koordinaten nach rechts bzw. unten zunehmen. Auf einem Bildschirm von 1200 x 800 Pixeln hat die untere rechte Ecke also die Koordinaten `(1200, 800)`. Diese Koordinaten beziehen sich auf das Fenster des Spiels, nicht auf den physischen Bildschirm!

Um das Schiff mittig am unteren Bildschirmrand zu platzieren, setzen wir den Wert von `self.rect.midbottom` auf den Wert, den das `midbottom`-Attribut für das Rechteck des Bildschirms hat (❹). Pygame platziert das Bild des Schiffes daraufhin so, dass es horizontal zentriert und vertikal am unteren Rand des Bildschirms ausgerichtet wird.

Bei ❺ definieren wir die Methode `blitme()`, die das Bild an der von `self.rect` angegebenen Position auf den Bildschirm zeichnet.

Das Schiff auf den Bildschirm zeichnen

Als Nächstes erweitern wir *alien_invasion.py* so, dass der Code eine Schiffsinstanz anlegt und die Methode `blitme()` für das Schiff aufruft:

alien_invasion.py

```
-- schnipp --
from settings import Settings
from ship import Ship
```

```
    class AlienInvasion:
        """Overall class to manage game assets and behavior."""

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

❶           self.ship = Ship(self)

        def run_game(self):
            -- schnipp --
            # Zeichnet den Bildschirm bei jedem Schleifendurchlauf neu.
            self.screen.fill(self.settings.bg_color)
❷           self.ship.blitme()

            # Macht den zuletzt gezeichneten Bildschirm sichtbar.
            pygame.display.flip()
    -- schnipp --
```

Hier importieren wir `Ship` und legen nach dem Erstellen des Bildschirms eine Instanz davon an (❶). Der Aufruf von `Ship()` erfordert ein Argument, nämlich eine Instanz von `AlienInvasion`. Das Argument `self` verweist hier auf die aktuelle Instanz dieser Klasse. Dieser Parameter gibt `Ship` Zugriff auf die Ressourcen des Spiels wie das `screen`-Objekt. Diese `Ship`-Instanz weisen wir `self.ship` zu.

Nachdem wir den Hintergrund ausgefüllt haben, zeichnen wir das Schiff mithilfe von `ship.blitme()` auf den Bildschirm, sodass es vor dem Hintergrund angezeigt wird (❷).

Wenn Sie jetzt *alien_invasion.py* ausführen, sehen Sie ein fast leeres Spielfeld mit unserem Raumschiff unten in der Mitte am Bildschirmrand (siehe Abb. 12–2).

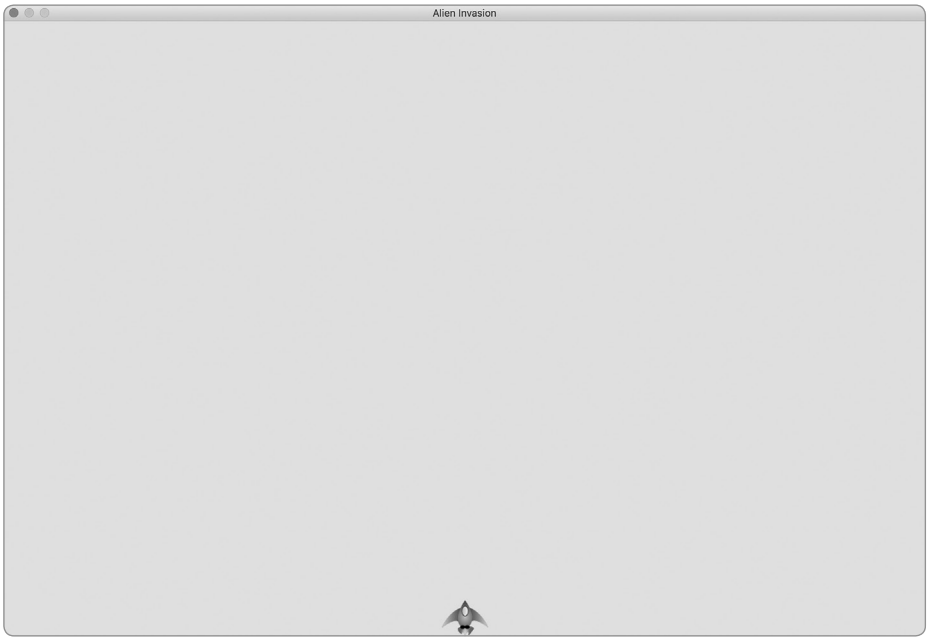

Abb. 12–2 *Alien Invasion mit einem Schiff mittig am unteren Bildschirmrand*

Refactoring: Die Methoden _check_events() und _update_screen()

Bei größeren Projekten führen Sie häufig ein *Refactoring* an dem bereits geschriebenen Code durch, bevor Sie weiteren Code hinzufügen. Dabei vereinfachen Sie die Struktur des vorhandenen Codes, sodass es leichter wird, darauf aufzubauen. In diesem Abschnitt zerlegen wir die Methode `run_game()`, die schon ziemlich lang geworden ist, in zwei *Hilfsmethoden. Eine Hilfsmethode funktioniert innerhalb einer Klasse, ist aber nicht dazu bestimmt, von einer Instanz aufgerufen zu werden. Ein einzelner führender Unterstrich kennzeichnet in Python eine Hilfsmethode.*

Die Methode _check_events()

Wir verlagern den Code für den Umgang mit Ereignissen in eine eigene Methode, die wir `_check_events()` nennen. Dadurch vereinfachen wir nicht nur `run_game()`, sondern separieren auch die Ereignisschleife, sodass wir Ereignisse unabhängig von anderen Aspekten des Spiels wie etwa der Aktualisierung des Bildschirms handhaben können.

Das folgende Listing zeigt die neue Methode `_check_events()` und die Änderungen am Code von `run_game()` in der Klasse `AlienInvasion`:

alien_invasion.py

```
    def run_game(self):
        """Start the main loop for the game."""
        while True:
❶           self._check_events()
            # Zeichnet den Bildschirm bei jedem Schleifendurchlauf neu.
            -- schnipp --

❷   def _check_events(self):
        """Respond to keypresses and mouse events."""
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                sys.exit()
```

Bei ❷ erstellen wir die neue Methode `_check_events()`und nehmen die Codezeilen, die prüfen, ob der Spieler geklickt hat, um das Spiel zu beenden, darin auf.

Um eine Methode aus einer Klasse heraus aufzurufen, wird die Punktschreibweise mit der Variablen `self` und dem Namen der Methode verwendet (❶). Auf diese Weise rufen wir `_check_events()` in der `while`-Schleife in `run_game()` auf.

Die Methode _update_screen()

Um `run_game()` weiter zu vereinfachen, verschieben wir den Code zur Aktualisierung des Bildschirms in die Methode `_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):
        -- schnipp --

    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()
```

Damit haben wir den Code, der den Hintergrund und das Raumschiff zeichnet und auf dem Bildschirm darstellt, in `_update_screen()` verlagert. Der Rumpf der Hauptschleife in `run_game()` ist dadurch viel einfacher geworden. Es ist jetzt leicht zu erkennen, dass wir darin nach neuen Ereignissen Ausschau halten und den Bildschirm bei jedem Schleifendurchlauf aktualisieren.

Wenn Sie bereits eine Reihe von Spielen geschrieben haben, werden Sie Ihren Code wahrscheinlich schon von Anfang an in solche Methoden unterteilen. Sollten Sie aber noch nie ein Projekt wie dieses in Angriff genommen haben, ist Ihnen möglicherweise nicht klar, wie Sie Ihren Code gliedern sollen. Die hier gezeigte Vorgehensweise, zunächst einmal funktionierenden Code zu schreiben und ihn dann bei zunehmender Komplexität umzustrukturieren, gibt Ihnen auch eine Vorstellung davon, wie die Entwicklung in der Praxis vonstattengeht: Sie beginnen damit, Ihren Code auf so einfache Weise wie möglich zu schreiben, und führen dann, wenn das Projekt immer vielschichtiger wird, Refactorings durch.

Nachdem wir den Code jetzt so umstrukturiert haben, dass wir ihn leichter erweitern können, wollen wir uns um die dynamischen Aspekte des Spiels kümmern.

Probieren Sie es selbst aus!

12-1 Blauer Himmel: Erstellen Sie ein Pygame-Fenster mit einem blauen Hintergrund.

12-2 Spielfigur: Suchen Sie ein Bitmap-Bild einer Spielfigur, die Ihnen gefällt, oder konvertieren Sie ein Bild ins Bitmap-Format. Erstellen Sie eine Klasse, die die Figur in die Mitte des Bildschirms zeichnet, und passen Sie die Hintergrundfarbe des Bildes an die des Bildschirms an oder umgekehrt.

Das Schiff bewegen

Als Nächstes wollen wir dafür sorgen, dass der Spieler das Schiff nach rechts und links bewegen kann. Dazu schreiben wir Code, der darauf reagiert, dass der Spieler die Tasten mit den nach rechts bzw. links weisenden Pfeilen drückt. Dabei konzentrieren wir uns erst auf die Bewegung nach rechts und wenden anschließend die gleichen Prinzipien auf die Bewegung nach links an. Dabei lernen Sie, wie Sie die Bewegung von Bildern auf dem Bildschirm steuern können.

Auf Tastenbetätigungen reagieren

Wenn der Spieler eine Taste drückt, wird dies in Pygame als Ereignis registriert. Da die Methode `pygame.event.get()` jegliche Ereignisse erfasst, müssen wir in der Methode `_check_events()` genau angeben, auf welche Art von Ereignissen sie lauschen soll.

Ein Tastendruck wird als `KEYDOWN`-Ereignis registriert. Allerdings müssen wir bei einem solchen Ereignis noch prüfen, ob die betätigte Taste tatsächlich diejenige war, die ein bestimmtes Ereignis auslösen soll. Beispielsweise wollen wir den Wert

des Attributs `rect.x` erhöhen, um das Schiff nach rechts zu bewegen, wenn der Benutzer die Taste mit dem nach rechts weisenden Pfeil drückt:

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:
                    # Bewegt das Schiff nach rechts.
❸                   self.ship.rect.x += 1
```

Im Rumpf von `_check_events()` ergänzen wir die Ereignisschleife um einen `elif`-Block für den Fall, dass Pygame ein `KEYDOWN`-Ereignis erfasst (❶). Wir prüfen, ob es sich bei der gedrückten Taste (`event.key`) um die Taste mit dem Rechtspfeil handelt (❷), die durch `pygame.K_RIGHT` dargestellt wird. Wenn ja, bewegen wir das Schiff nach rechts, indem wir den Wert von `self.ship.rect.x` um 1 erhöhen (❸).

Wenn Sie *alien_invasion.py* jetzt ausführen, bewegt sich das Schiff jedes Mal, wenn Sie die Taste mit dem nach rechts weisenden Pfeil drücken, ein Pixel nach rechts. Das ist schon einmal ein Anfang, aber nicht besonders wirkungsvoll, um die Bewegung eines Raumschiffs zu steuern. Daher wollen wir als Nächstes kontinuierliche Bewegungen zulassen.

Kontinuierliche Bewegung

Wenn der Spieler die Pfeiltaste gedrückt hält, soll sich das Schiff kontinuierlich nach rechts bewegen, bis die Taste wieder losgelassen wird. Mit dem Ereignis `pygame.KEYUP` können wir erkennen, wann eine Taste losgelassen wird. Um für kontinuierliche Bewegung zu sorgen, verwenden wir die Ereignisse `KEYDOWN` und `KEYUP` sowie ein Flag namens `moving_right`.

Solange dieses Flag den Wert `False` hat, ist das Schiff nicht in Bewegung. Drückt der Spieler die Rechtspfeiltaste, so setzen wir das Flag auf `True`. Beim Loslassen der Taste erhält das Flag wieder den Wert `False`.

Da die Klasse `Ship` für alle Merkmale des Schiffes zuständig ist, fügen wir ihr das Attribut `moving_right` sowie die Methode `update()` hinzu, die den Status dieses Flags prüft. Hat das Flag den Wert `True`, ändert die Methode die Position des Schiffes. Wir rufen diese Methode bei jedem Durchlauf der `while`-Schleife einmal auf, um die Position des Schiffes zu aktualisieren.

An der Klasse `Ship` müssen wir dazu folgende Änderungen vornehmen:

ship.py

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

    def __init__(self, ai_game):
        -- schnipp --
        # Platziert jedes neue Schiff mittig am unteren Bildschirmrand.
        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):
        -- schnipp --
```

Wir ergänzen die Methode `__init__()` um das Attribut `self.moving_right` und setzen es zunächst auf `False` (❶). Außerdem fügen wir die Methode `update()` hinzu, die das Schiff nach rechts verschiebt, wenn das Flag den Wert `True` hat (❷). Da `update()` durch eine Instanz von `Ship` aufgerufen wird, handelt es sich bei ihr nicht um eine Hilfsmethode.

Als Nächstes ändern wir `_check_events()`, sodass `moving_right` auf `True` gesetzt wird, wenn der Benutzer die Rechtspfeiltaste drückt, und auf `False`, wenn er sie wieder loslässt:

alien_invasion.py

```
    def _check_events(self):
        """Respond to keypresses and mouse events."""
        for event in pygame.event.get():
            -- schnipp --
            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
```

Bei ❶ ändern wir die bisherige Reaktion des Spiels auf eine Betätigung der Rechtspfeiltaste. Anstatt die Position des Schiffes direkt zu ändern, setzen wir lediglich `moving_right` auf `True`. Bei ❷ fügen wir einen neuen `elif`-Block hinzu, der auf `KEYUP`-Ereignisse reagiert und `moving_right` wieder auf `False` setzt, wenn die Rechtspfeiltaste (`K_RIGHT`) losgelassen wird.

Schließlich ändern wir noch die `while`-Schleife in `run_game()`, sodass darin jetzt bei jedem Durchlauf die Methode `update()` für das Schiff aufgerufen wird:

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()
```

Wir aktualisieren die Position des Schiffes, nachdem wir auf Tastaturereignisse gelauscht haben, aber bevor wir den Bildschirm aktualisieren. Dadurch können wir die Position als Reaktion auf Eingaben des Spielers verändern und stellen sicher, dass beim Zeichnen des Schiffes auf den Bildschirm die neue Position verwendet wird.

Wenn Sie jetzt *alien_invasion.py* ausführen und die Taste mit dem nach rechts weisenden Pfeil gedrückt halten, bewegt sich das Schiff kontinuierlich nach rechts, bis Sie die Taste wieder loslassen.

Bewegung nach rechts und links

Da sich das Schiff jetzt kontinuierlich nach rechts bewegen kann, ist es ganz einfach, ihm auch noch die Bewegung nach links beizubringen. Dazu müssen wir abermals die Klasse `Ship` und die Methode `_check_events()` abwandeln. Die Methoden `__init__()` und `update()` in `Ship` ändern wir dazu wie folgt:

ship.py

```
def __init__(self, ai_game):
    -- schnipp --
    # Bewegungsflags
    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__()` fügen wir das Flag `self.moving_left` hinzu. Statt `elif` verwenden wir in `update()` zwei getrennte `if`-Blöcke. Wenn der Spieler bei einem Wechsel der Bewegungsrichtung eine kurze Zeit lang beide Tasten gedrückt hält, wird der Wert von `rect.x` für das Schiff dadurch erst erhöht, dann verringert, sodass das Schiff stehen bleibt. Hätten wir für die Bewegung nach links einen `elif`-Block verwendet, so hätte die Rechtspfeiltaste in einem solchen Fall dagegen Priorität. Durch die Verwendung von zwei `if`-Blöcken erreichen wir daher eine Bewegung, die besser der Absicht des Spielers entspricht.

Außerdem müssen wir zwei Änderungen an `_check_events()` vornehmen:

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
```

Bei einem `KEYDOWN`-Ereignis für `K_LEFT` setzen wir `moving_left` auf `True`, bei einem `KEYUP`-Ereignis auf `False`. Hier können wir `elif`-Blöcke verwenden, da jedes Ereignis nur mit einer Taste verknüpft ist. Wenn der Spieler beide Tasten gleichzeitig drückt, werden zwei getrennte Ereignisse erkannt.

Wenn Sie *alien_invasion.py* jetzt ausführen, können Sie das Schiff kontinuierlich nach rechts und nach links bewegen. Halten Sie beide Tasten gedrückt, hält das Schiff an.

Die Bewegungssteuerung können wir aber noch verbessern. Als Nächstes passen wir die Geschwindigkeit des Schiffes an und schränken seinen Bewegungsspielraum ein, sodass es nicht hinter dem Bildschirmrand verschwindet.

Die Geschwindigkeit des Schiffes anpassen

Zurzeit bewegt sich das Schiff um ein Pixel pro Durchlauf durch die `while`-Schleife. Um die Geschwindigkeit regeln zu können, fügen wir der Klasse `Settings` das Attribut `ship_speed` hinzu. Damit legen wir fest, wie weit das Schiff bei jedem Schleifendurchlauf verschoben wird. In *settings.py* sieht unser neues Attribut wie folgt aus:

settings.py

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

    def __init__(self):
        -- schnipp --

        # Schiffseinstellungen
        self.ship_speed = 1.5
```

Den Anfangswert von `ship_speed` setzen wir auf `1.5`. Wenn wir das Schiff bewegen, wird seine Position jetzt um 1,5 statt nur um 1 Pixel verschoben.

Für die Geschwindigkeitseinstellung verwenden wir Fließkommawerte, da wir damit eine feinere Kontrolle der Schiffsgeschwindigkeit bekommen, wenn wir das Spieltempo später erhöhen. Da `rect`-Attribute wie `x` jedoch nur Integerwerte speichern können, müssen wir noch einige Änderungen an `Ship` vornehmen:

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
        -- schnipp --

        # Platziert ein neues Schiff mittig am unteren Bildschirmrand.
        -- schnipp --

        # Speichert einen Fließkommawert für den Schiffsmittelpunkt.
❷       self.x = float(self.rect.x)

        # Bewegungsflags
        self.moving_right = False
        self.moving_left = False

    def update(self):
        """Update the ship's position based on movement flags."""
        # Aktualisiert den Wert für den Mittelpunkt des Schiffs,
        # nicht des Rechtecks.
        if self.moving_right:
❸           self.x += self.settings.ship_speed
        if self.moving_left:
            self.x -= self.settings.ship_speed

        # Aktualisiert das rect-Objekt auf der Grundlage von self.x.
❹       self.rect.x = self.x

    def blitme(self):
        -- schnipp --
```

Wir erstellen das Attribut `settings` für `Ship`, sodass wir es in `update()` verwenden können (❶). Da wir die Position des Schiffes nun um Bruchteile von Pixeln verschieben, müssen wir diese Position einer Variablen zuweisen, die Fließkommazahlen aufnehmen kann. Sie können zwar einen Fließkommawert angeben, um ein Attribut eines `rect`-Objekts festzulegen, gespeichert wird dabei aber nur der Integeranteil dieses Werts. Um die Position des Schiffes genau festzuhalten, defi-

nieren wir das neue Attribut `self.x`, das Fließkommawerte aufnehmen kann (❷). Mit der Funktion `float()` wandeln wir den Wert von `self.rect.x` in einen Fließkommawert um und speichern das Ergebnis in `self.x`.

Wenn wir die Position des Schiffes in `update()` verschieben, wird der Wert von `self.x` nun jeweils um den in `settings.ship_speed` gespeicherten Betrag geändert (❸). Nach der Aktualisierung von `self.x` nutzen wir dessen neuen Wert, um das Attribut `self.rect.x` zu aktualisieren, das die Position des Schiffes bestimmt (❹). Dabei wird in `self.rect.x` nur der Integeranteil von `self.x` gespeichert, was aber für die Anzeige des Schiffes ausreicht.

Jetzt können wir den Wert von `ship_speed` ändern, wobei jede Einstellung größer als 1 dafür sorgt, dass sich das Schiff schneller bewegt. Das ist hilfreich, um das Schiff schnell genug reagieren zu lassen, sodass es die gegnerischen Schiffe abschießen kann, und ermöglicht uns außerdem, das Spieltempo bei fortschreitendem Spielverlauf zu erhöhen.

Hinweis

Auf macOS kann es vorkommen, dass sich das Schiff selbst bei hohen Geschwindigkeitswerten sehr langsam bewegt. Dieses Problem können Sie dadurch lösen, dass Sie das Spiel im Vollbildmodus ausführen, worum wir uns in Kürze kümmern werden.

Den Bewegungsbereich des Schiffes einschränken

Wenn Sie eine der Pfeiltasten zu lange gedrückt halten, verschwindet das Schiff hinter dem Bildschirmrand. Das wollen wir korrigieren, sodass das Schiff anhält, wenn es den Rand erreicht. Dazu ändern wir die Methode `update()` in `Ship`:

ship.py

```
    def update(self):
        """Update the ship's position based on movement flags."""
        # Aktualisiert den Wert für den Mittelpunkt des Schiffs,
        # nicht des Rechtecks.
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

        # Aktualisiert das rect-Objekt auf der Grundlage von self.x.
        self.rect.x = self.x
```

Dieser Code prüft erst die Position des Schiffes, bevor er den Wert von `self.x` ändert. Dabei gibt `self.rect.right` die x-Koordinate für den rechten Rand des Rechtecks um das Schiff zurück. Solange dieser Wert kleiner als der von `self.screen_rect.right` ist, hat das Schiff den rechten Rand des Bildschirms noch nicht erreicht (❶). Das Gleiche gilt für den linken Rand: Solange der Wert für den linken

Rand des Rechtecks größer als 0 ist, befindet sich das Schiff noch nicht am linken Bildschirmrand (❷). Damit stellen wir sicher, dass wir den Wert von `self.x` nur dann ändern, wenn sich das Schiff innerhalb dieser Grenzen befindet.

Wenn Sie *alien_invasion.py* jetzt ausführen, hält das Schiff an beiden Bildschirmrändern an. Das ist schon eine tolle Sache: Wir haben lediglich eine Bedingung in einer `if`-Anweisung geändert, und schon sieht es so aus, als ob das Schiff gegen eine Wand oder ein Kraftfeld stößt, wenn es den Bildschirmrand berührt!

Refactoring von _check_events()

Je weiter wir das Spiel entwickeln, umso länger wird die Methode `_check_events()`. Daher wollen wir sie in zwei getrennte Methoden für `KEYDOWN`- bzw. `KEYUP`-Ereignisse zerlegen:

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
```

Wir erstellen hier die beiden neuen Hilfsmethoden `_check_keydown_events()` und `_check_keyup_events()`, die jeweils einen `self`- und einen `event`-Parameter benötigen. Die Rümpfe dieser Methoden haben wir dem Code von `_check_events()` entnommen. Außerdem haben wir den alten Code durch Aufrufe der beiden neuen Methoden ersetzt. Diese klare Codestruktur macht die Methode `_check_events()` einfacher, wodurch sich weitere Reaktionen auf Eingaben des Spielers leichter hinzufügen lassen.

Beenden mit Q

Es kann ziemlich nervtötend sein, jedes Mal auf das X oben im Spielfenster zu klicken, um das Spiel nach dem Test eines neuen Elements zu beenden. Daher wollen wir die Möglichkeit hinzufügen, das Spiel durch Drücken der Taste Q zu beenden:

alien_invasion.py

```
def _check_keydown_events(self, event):
    -- schnipp --
    elif event.key == pygame.K_LEFT:
        self.ship.moving_left = True
    elif event.key == pygame.K_q:
        sys.exit()
```

Dazu fügen wir in `_check_keydown_events()` einen neuen Block ein, der das Spiel beendet, wenn der Spieler Q drückt. Jetzt können Sie beim Testen einfach diese Taste betätigen, um das Spiel abzubrechen, anstatt den Cursor zur Schließen-Schaltfläche bewegen zu müssen.

Das Spiel im Vollbildmodus ausführen

Pygame ermöglicht auch einen Vollbildmodus. Manche Spiele sehen darin einfach besser aus als in einem regulären Fenster, und auf macOS können Sie im Vollbildmodus auch eine bessere Leistung erzielen.

Um das Spiel im Vollbildmodus auszuführen, nehmen Sie die folgenden Änderungen an `__init__()` vor:

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")
```

Beim Erstellen der Bildschirmoberfläche übergeben wir die Größe `(0, 0)` und den Parameter `pygame.FULLSCREEN` (❶). Dadurch weisen wir Pygame an, eine bildschirmfüllende Fenstergröße zu ermitteln. Da wir die Höhe und Breite des Bildschirms nicht im Voraus kennen, aktualisieren wir diese Einstellungen im `settings`-Objekt mit den Attributen `width` und `height` des Bildschirmrechtecks, nachdem wir den Bildschirm angelegt haben (❷).