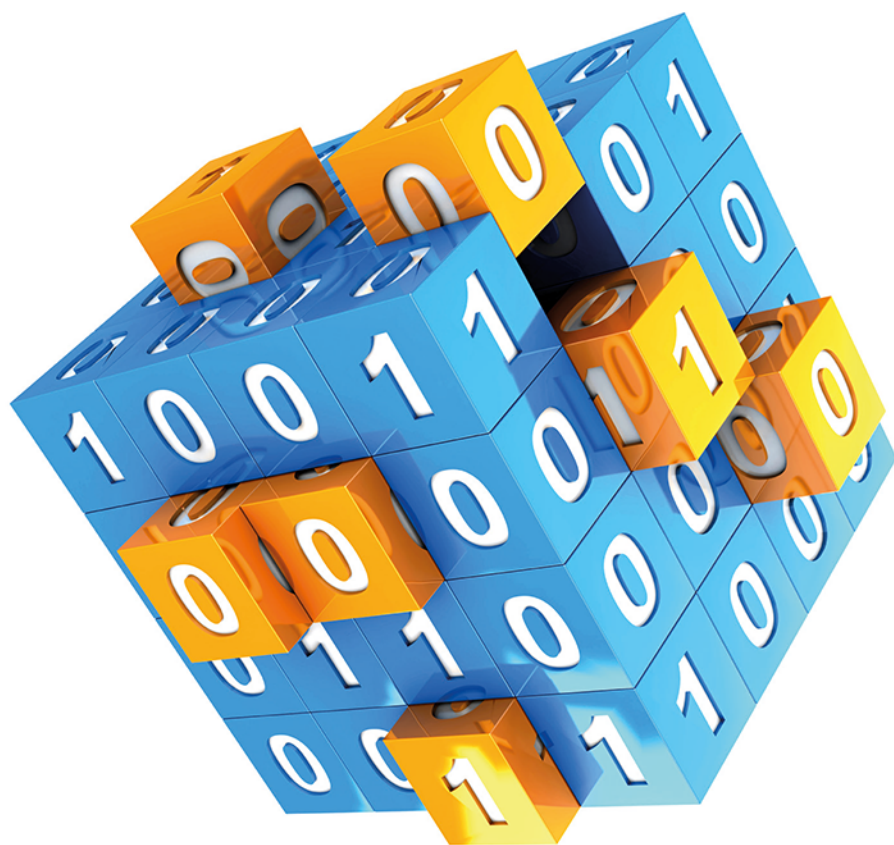


Michael Dawson

Wydanie III

PYTHON DLA KAŻDEGO

Podstawy programowania



Od zera do bohatera!

Helion 

Tytuł oryginału: Python Programming for the Absolute Beginner, 3rd Edition

Tłumaczenie: Grzegorz Pawłowski

Projekt okładki: Studio Gravite / Olsztyn; Obarek, Pokoński, Pazdrijowski, Zaprucki
Materiały graficzne na okładce zostały wykorzystane za zgodą Shutterstock Images LLC.

ISBN: 978-83-8322-774-0

© 2010 Course Technology, a part of Cengage Learning.

ALL RIGHTS RESERVED. No part of this work covered by the copyright herein may be reproduced, transmitted, stored, or used in any form or by any means graphic, electronic, or mechanical, including but not limited to photocopying, recording, scanning, digitizing, taping, Web distribution, information networks, or information storage and retrieval systems, without the prior written permission of the publisher.

Python is a registered trademark of the Python Software Foundation.

All other trademarks are the property of their respective owners.

All images © Cengage Learning unless otherwise noted.

Polish edition copyright © 2014, 2023 by Helion S.A. All rights reserved.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiegokolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz wydawca dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz wydawca nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Helion S.A.

ul. Kościuszki 1c, 44-100 Gliwice

tel. 32 230 98 63

e-mail: helion@helion.pl

WWW: <https://helion.pl> (księgarnia internetowa, katalog książek)

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

<https://helion.pl/user/opinie/pytd3v>

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:

<https://ftp.helion.pl/przyklady/pytd3v.zip>

Printed in Poland.

- Kup książkę
- Poleć książkę
- Oceń książkę

- Księgarnia internetowa
- Lubię to! » Nasza społeczność

Spis treści

O autorze	11
Wstęp	13
Rozdział 1. Wprowadzenie. Program Koniec gry	15
Analiza programu Koniec gry	15
Co warto wiedzieć o Pythonie?	16
Konfiguracja Pythona w systemie Windows	19
Konfiguracja Pythona w innych systemach operacyjnych	20
Wprowadzenie do IDLE	20
Powrót do programu Koniec gry	26
Podsumowanie	29
Rozdział 2. Typy, zmienne i proste operacje wejścia-wyjścia.	
Program Nieistotne fakty	31
Wprowadzenie do programu Nieistotne fakty	31
Użycie cudzysłowów przy tworzeniu łańcuchów znaków	32
Używanie sekwencji specjalnych w łańcuchach znaków	36
Konkatenacja i powielanie łańcuchów	40
Operacje na liczbach	42
Pojęcie zmiennych	45
Pobieranie danych wprowadzanych przez użytkownika	48
Używanie metod łańcucha	50
Stosowanie właściwych typów	54
Konwersja wartości	56
Powrót do programu Nieistotne fakty	59
Podsumowanie	61
Rozdział 3. Rozgałęzianie kodu, pętle while, projektowanie programu.	
Gra Odgadnij moją liczbę	63
Wprowadzenie do gry Jaka to liczba?	63
Generowanie liczb losowych	64
Używanie instrukcji if	67

Używanie klauzuli else	71
Używanie klauzuli elif	73
Tworzenie pętli while	76
Unikanie pętli nieskończonych	80
Traktowanie wartości jako warunków	83
Tworzenie umyślnych pętli nieskończonych	86
Korzystanie z warunków złożonych	88
Projektowanie programów	93
Powrót do gry Jaka to liczba?	95
Podsumowanie	97
Rozdział 4. Pętle for, łańcuchy znaków i krotki.	
Gra Wymieszane litery	99
Wprowadzenie do programu Wymieszane litery	99
Liczenie za pomocą pętli for	102
Stosowanie funkcji i operatorów sekwencji do łańcuchów znaków ...	105
Indeksowanie łańcuchów	107
Niemutowalność łańcuchów	111
Tworzenie nowego łańcucha	113
Wycinanie łańcuchów	116
Powrót do gry Wymieszane litery	128
Podsumowanie	131
Rozdział 5. Listy i słowniki. Gra Szubienica	133
Wprowadzenie do gry Szubienica	133
Korzystanie z list	135
Korzystanie z metod listy	140
Kiedy należy używać krotek zamiast list?	144
Używanie sekwencji zagnieżdżonych	145
Referencje współdzielone	149
Używanie słowników	152
Powrót do gry Szubienica	159
Podsumowanie	165
Rozdział 6. Funkcje. Gra Kółko i krzyżyk	167
Wprowadzenie do gry Kółko i krzyżyk	167
Tworzenie funkcji	169
Używanie parametrów i wartości zwrrotnych	172
Wykorzystanie argumentów nazwanych i domyślnych wartości parametrów	176
Wykorzystanie zmiennych globalnych i stałych	181
Powrót do gry Kółko i krzyżyk	185
Podsumowanie	196

Rozdział 7. Pliki i wyjątki. Gra Turniej wiedzy	199
Wprowadzenie do programu Turniej wiedzy	199
Odczytywanie danych z plików tekstowych	200
Zapisywanie danych do pliku tekstowego	206
Przechowywanie złożonych struktur danych w plikach	209
Obsługa wyjątków	214
Powrót do gry Turniej wiedzy	218
Podsumowanie	223
Rozdział 8. Obiekty programowe. Program Opiekun zwierzaka	225
Wprowadzenie do programu Opiekun zwierzaka	225
Podstawy programowania obiektowego	227
Tworzenie klas, metod i obiektów	228
Używanie konstruktorów	230
Wykorzystywanie atrybutów	232
Wykorzystanie atrybutów klasy i metod statycznych	236
Hermetyzacja obiektów	240
Używanie atrybutów i metod prywatnych	241
Kontrolowanie dostępu do atrybutów	245
Powrót do programu Opiekun zwierzaka	249
Podsumowanie	252
Rozdział 9. Programowanie obiektowe. Gra Blackjack	255
Wprowadzenie do gry Blackjack	255
Wysyłanie i odbieranie komunikatów	256
Tworzenie kombinacji obiektów	259
Wykorzystanie dziedziczenia do tworzenia nowych klas	263
Rozszerzanie klasy poprzez dziedziczenie	263
Modyfikowanie zachowania odziedziczonych metod	269
Polimorfizm	273
Tworzenie modułów	273
Powrót do gry Blackjack	277
Podsumowanie	287
Rozdział 10. Tworzenie interfejsów GUI. Gra Mad Lib	289
Wprowadzenie do programu Mad Lib	289
Przyjrzenie się interfejsowi GUI	291
Programowanie sterowane zdarzeniami	292
Zastosowanie okna głównego	293
Używanie przycisków	298
Tworzenie interfejsu GUI przy użyciu klasy	300
Wiązanie widżetów z procedurami obsługi zdarzeń	303
Używanie widżetów Text i Entry oraz menedżera układu Grid	305

Wykorzystanie pól wyboru	310
Wykorzystanie przycisków opcji	314
Powrót do programu Mad Lib	318
Podsumowanie	322
Rozdział 11. Grafika. Gra Pizza Panic	323
Wprowadzenie do gry Pizza Panic	323
Wprowadzenie do pakietów pygame i livewires	325
Tworzenie okna graficznego	325
Ustawienie obrazu tła	328
Układ współrzędnych ekranu graficznego	331
Wyświetlanie duszka	332
Wyświetlanie tekstu	336
Wyświetlanie komunikatu	339
Przemieszczanie duszków	342
Radzenie sobie z granicami ekranu	344
Obsługa danych wejściowych z myszy	347
Wykrywanie kolizji	350
Powrót do gry Pizza Panic	353
Podsumowanie	360
Rozdział 12. Dźwięk, animacja i rozwijanie programu.	361
Gra Astrocrash	361
Wprowadzenie do gry Astrocrash	361
Odczyt klawiatury	363
Obracanie duszka	365
Tworzenie animacji	368
Przegląd obrazów eksplozji	369
Wykorzystywanie dźwięku i muzyki	371
Planowanie gry Astrocrash	376
Utworzenie asteroidów	377
Obracanie statku	380
Poruszanie statku	382
Wystrzeliwanie pocisków	385
Regulowanie tempa wystrzeliwania pocisków	388
Obsługa kolizji	390
Dodanie efektów eksplozji	393
Dodanie poziomów gry, rejestracji wyników oraz tematu muzycznego	397
Podsumowanie	405

Dodatek A. Strona internetowa książki	407
Pliki archiwów	407
Dodatek B. Opis pakietu livewires	409
Pakiet livewires	409
Klasy modułu games	409
Funkcje modułu games	417
Stałe modułu games	418
Stałe modułu color	422
Skorowidz	423

Dźwięk, animacja i rozwijanie programu. Gra Astrocrash

W tym rozdziale poszerzysz swoje umiejętności tworzenia programów multimedialnych o obsługę dźwięku i ruchomych obrazów. Zobaczysz również, jak można pisać duży program etapami. W szczególności nauczysz się:

- odczytywać dane z klawiatury,
- odtwarzać pliki dźwiękowe,
- odtwarzać pliki muzyczne,
- tworzyć animacje,
- rozwijać program poprzez pisanie coraz bardziej kompletnych jego wersji.

Wprowadzenie do gry Astrocrash

Projekt przedstawiony w tym rozdziale, gra Astrocrash, to moja wersja klasycznej gry arkadowej Asteroids. W grze Astrocrash gracz steruje statkiem kosmicznym w ruchomym polu śmiertelnie groźnych asteroidów. Statek może się obracać i wykonywać ruch do przodu oraz, co najważniejsze, może wystrzeliwać pociski w kierunku asteroidów, aby je zniszczyć. Gracz ma jednak trochę roboty do wykonania, ponieważ asteroidy dużego i średniego rozmiaru rozpadają się na dwie mniejsze asteroidy, gdy zostaną zniszczone. I w tej samej chwili, gdy graczowi uda się unicestwić wszystkie asteroidy, pojawia się ich nowa, większa fala. Liczba punktów uzyskanych przez gracza zwiększa się wraz z każdą zniszczoną asteroidą, lecz jeśli tylko statek gracza zderzy się z unoszącą się w przestrzeni kosmicznej skałą, gra się kończy. Rysunki 12.1 i 12.2 pokazują tę grę w toku.



Rysunek 12.1. Gracz steruje statkiem kosmicznym i niszczy asteroidy w celu zwiększenia swojego wyniku punktowego. (Obraz mgławicy należy do domeny publicznej. Dzięki uprzejmości: NASA, The Hubble Heritage Team–AURA/STScI)



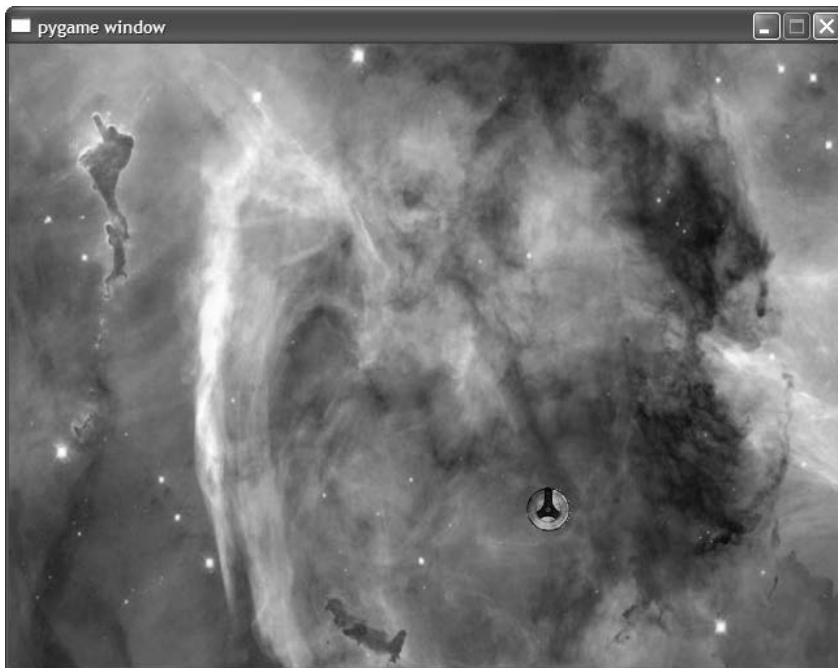
Rysunek 12.2. Jeśli asteroida uderzy w statek gracza, gra się kończy

Odczyt klawiatury

Już wiesz, jak pobierać łańcuchy znaków od użytkownika przy użyciu funkcji `input()`, ale odczyt klawiatury w celu identyfikacji pojedynczych naciśnień klawiszy to inna kwestia. Na szczęście istnieje nowy obiekt z modułu `games`, który to właśnie umożliwia.

Prezentacja programu Odczytaj klawisz

Program Odczytaj klawisz wyświetla statek kosmiczny na tle mgławicy. Użytkownik może przemieszczać statek po całym ekranie za pomocą naciśnień różnych klawiszy. Kiedy użytkownik naciska klawisz *W*, statek porusza się do góry. Gdy użytkownik naciska klawisz *S*, statek porusza się w dół. Kiedy użytkownik naciska klawisz *A*, statek porusza się w lewo. Kiedy użytkownik naciska klawisz *D*, statek porusza się w prawo. Użytkownik może również nacisnąć wiele klawiszy jednocześnie dla uzyskania łącznego efektu. Na przykład, gdy użytkownik naciska jednocześnie klawisze *W* i *D*, statek porusza się po przekątnej — w prawo, do góry. Program został zilustrowany na rysunku 12.3.



Rysunek 12.3. Statek porusza się po ekranie dzięki naciśnięciom klawiszy

Kod tego programu możesz znaleźć na stronie internetowej tej książki (<https://www.helion.pl/ksiazki/pytd3v.htm>), w folderze rozdziału 12.; nazwa pliku to `odczytaj_klawisz.py`.

Rozpoczęcie programu

Tak jak w przypadku wszystkich programów wykorzystujących pakiet `liveswires` rozpoczynam od importu potrzebnych modułów i wywołania funkcji inicjalizującej ekran graficzny:

```
# Odczytaj klawisz
# Demonstruje odczytywanie klawiatury

from liveswires import games

games.init(screen_width = 640, screen_height = 480, fps = 50)
```

Testowanie stanu klawiszy

Następnie piszę kod klasy reprezentującej statek kosmiczny. W metodzie `update()` sprawdzam, czy zostały naciśnięte określone klawisze, i zgodnie z wynikiem tych testów zmieniam pozycję statku.

```
class Ship(games.Sprite):
    """ Poruszający się statek kosmiczny. """
    def update(self):
        """ Kieruj ruchem statku na podstawie wciśniętych klawiszy. """
        if games.keyboard.is_pressed(games.K_w):
            self.y -= 1
        if games.keyboard.is_pressed(games.K_s):
            self.y += 1
        if games.keyboard.is_pressed(games.K_a):
            self.x -= 1
        if games.keyboard.is_pressed(games.K_d):
            self.x += 1
```

Wykorzystuję nowy obiekt z modułu `games` o nazwie `keyboard`. Możesz użyć tego obiektu do sprawdzenia, czy określone klawisze zostały naciśnięte. Wywołuję metodę `is_pressed()` obiektu, która zwraca wartość `True`, jeśli testowany klawisz jest naciśnięty, i wartość `False` w przeciwnym wypadku.

Używam metody `is_pressed` w ciągu instrukcji `if`, aby sprawdzić, czy którykolwiek z czterech klawiszy — *W*, *S*, *A* lub *D* — nie jest naciśnięty. Jeśli jest naciśnięty klawisz *W*, zmniejszam o 1 wartość właściwości `y` obiektu klasy `Ship`, przesuując duszka w górę ekranu o jeden piksel. Jeśli jest naciśnięty klawisz *S*, zwiększam wartość właściwości `y` obiektu o 1, przesuując duszka w dół ekranu. Jeśli jest naciśnięty klawisz *A*, zmniejszam o 1 wartość właściwości `x` obiektu, przesuując duszka w lewo. Jeśli jest naciśnięty klawisz *S*, zwiększam wartość właściwości `x` obiektu o 1, przesuując duszka w prawo.

Ponieważ wielokrotne wywołania metody `is_pressed()` mogą odczytywać jednoczesne naciśnięcia klawiszy, użytkownik może przyciskać wiele klawiszy naraz dla uzyskania łącznego efektu. Jeśli na przykład użytkownik przytrzymuje w tym samym czasie wciśnięte klawisze *D* i *S*, statek porusza się w dół i w prawo, ponieważ za każdym razem, gdy wykonywana jest metoda `update()`, wartość 1 zostaje dodana zarówno do współrzędnej `x`, jak i współrzędnej `y` obiektu klasy `Ship`.

Moduł `games` zawiera zbiór stałych reprezentujących klawisze, których możesz używać w roli argumentu w wywołaniu metody `is_pressed()`. W tym programie wykorzystuję stałą `games.K_w` reprezentującą klawisz *W*, stałą `games.K_s` odpowiadającą klawiszowi *S*, stałą `games.K_a` oznaczającą klawisz *A* oraz stałą `games.K_d` identyfikującą klawisz *D*. Schemat nadawania nazw tym stałym jest dość intuicyjny. Oto szybki sposób na wydedukowanie nazwy większości stałych reprezentujących klawisze:

- wszystkie stałe klawiatury zaczynają się od `games.K_`;
- w przypadku klawiszy alfabetycznych dodaj literę z klawisza, po zamianie na małą, na końcu nazwy stałej; na przykład stała reprezentująca klawisz *A* to `games.K_a`;
- w przypadku klawiszy numerycznych dodaj cyfrę z klawisza na końcu nazwy stałej; na przykład stała reprezentująca klawisz *1* to `games.K_1`;
- w przypadku pozostałych klawiszy często możesz na końcu nazwy stałej dodać ich nazwę pisaną samymi dużymi literami; na przykład stała reprezentująca klawisz spacji to `games.K_SPACE`.

Kompletną listę stałych klawiatury znajdziesz w dokumentacji pakietu `livewires`, w dodatku B.

Dokończenie programu

Na koniec piszę znajomą funkcję `main()`. Ładuję obraz tła przedstawiający mgławicę, tworzę statek, umieszczając go w środku ekranu, oraz wszystko uruchamiam poprzez wywołanie metody `mainloop()`.

```
def main():
    nebula_image = games.load_image("mglawica.jpg", transparent = False)
    games.screen.background = nebula_image

    ship_image = games.load_image("statek.bmp")
    the_ship = Ship(image = ship_image,
                    x = games.screen.width/2,
                    y = games.screen.height/2)
    games.screen.add(the_ship)

    games.screen.mainloop()

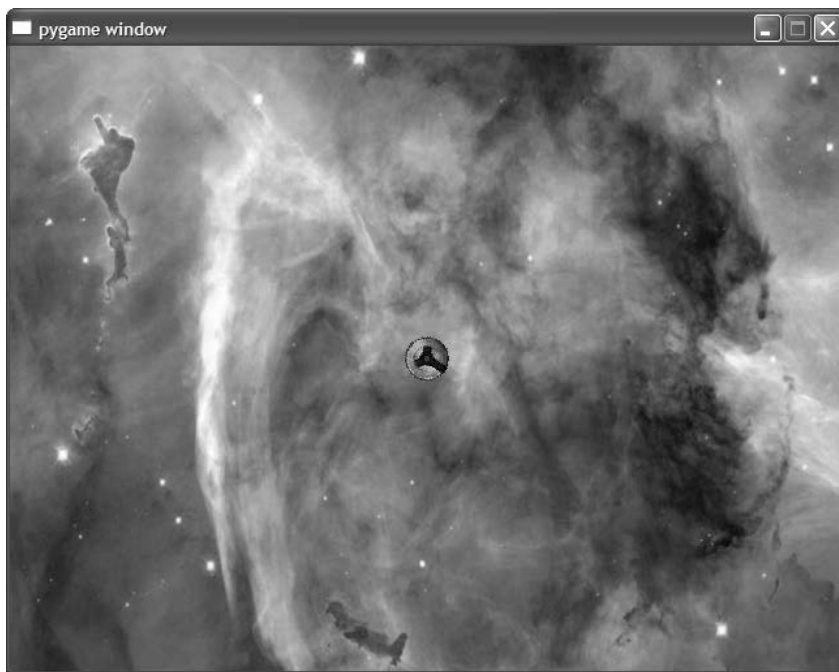
main()
```

Obracanie duszka

W rozdziale 11. dowiedziałeś się, jak przemieszczać duszki po ekranie, ale pakiet `livewires` umożliwi również ich obracanie. Duszka można obracać, wykorzystując jedną z jego właściwości.

Prezentacja programu Obróć duszka

W programie Obróć duszka, użytkownik może obracać statek kosmiczny przy użyciu klawiatury. Kiedy użytkownik naciska klawisz strzałki w górę, statek obraca się w kierunku zgodnym z ruchem wskazówki zegara. Jeśli użytkownik naciśnie klawisz strzałki w lewo, statek obróci się w kierunku przeciwnym do ruchu wskazówki zegara. Jeśli naciśnie klawisz 1, kąt położenia statku zmieni się skokowo na 0 stopni. Jeśli użytkownik naciśnie klawisz 2, statek zmieni swoją orientację na 90 stopni. Jeśli naciśnie klawisz 3, statek przyjmie położenie kątowne 180 stopni. Jeśli użytkownik naciśnie klawisz 4, kąt położenia zmieni się skokowo na 270 stopni. Program został przedstawiony na rysunku 12.4.



Rysunek 12.4. Statek kosmiczny może się obracać zgodnie z ruchem wskazówki zegara lub w kierunku przeciwnym do ruchu wskazówki zegara. Może też przeskoczyć do położenia pod z góry ustalonym kątem

Pułapka

Program Obróć duszka sprawdza, czy są wciśnięte klawisze cyfr znajdujące się u góry klawiatury, powyżej klawiszy z literami, lecz nie sprawdza stanu klawiszy klawiatury numerycznej.

Kod tego programu możesz znaleźć na stronie internetowej tej książki (<https://www.helion.pl/ksiazki/pytd3v.htm>), w folderze rozdziału 12.; nazwa pliku to `obroc_duszka.py`.

```

# Obróć duszka
# Demonstruje obracanie duszka
from livewires import games

games.init(screen_width = 640, screen_height = 480, fps = 50)

class Ship(games.Sprite):
    """ Obracający się statek kosmiczny. """
    def update(self):
        """ Obróć w zależności od naciśniętych klawiszy. """
        if games.keyboard.is_pressed(games.K_RIGHT):
            self.angle += 1
        if games.keyboard.is_pressed(games.K_LEFT):
            self.angle -= 1

        if games.keyboard.is_pressed(games.K_1):
            self.angle = 0
        if games.keyboard.is_pressed(games.K_2):
            self.angle = 90
        if games.keyboard.is_pressed(games.K_3):
            self.angle = 180
        if games.keyboard.is_pressed(games.K_4):
            self.angle = 270

def main():
    nebula_image = games.load_image("mgławica.jpg", transparent = False)
    games.screen.background = nebula_image

    ship_image = games.load_image("statek.bmp")
    the_ship = Ship(image = ship_image,
                    x = games.screen.width/2,
                    y = games.screen.height/2)
    games.screen.add(the_ship)

    games.screen.mainloop()

main()

```

Wykorzystanie właściwości angle obiektu klasy Sprite

Nowym elementem w programie jest właściwość `angle`, która reprezentuje orientację kątową duszka wyrażoną w stopniach. Możesz zwiększać lub zmniejszać wartość tej właściwości przez dodawanie lub odejmowanie przyrostów, lecz możesz też po prostu przypisać jej nową wartość w celu zmiany kąta położenia duszka.

W metodzie `update()` najpierw sprawdzam, czy wciśnięty jest klawisz strzałki w prawo. Jeśli tak jest, dodaję jedynkę do wartości właściwości `angle` obiektu, co powoduje obrócenie duszka o jeden stopień w kierunku ruchu wskazówki zegara. Następnie sprawdzam, czy wciśnięty jest klawisz strzałki w lewo. Jeśli jest tak w istocie, odejmuję jedynkę od wartości tej właściwości, powodując obrót duszka o jeden stopień w kierunku przeciwnym do ruchu wskazówki zegara.

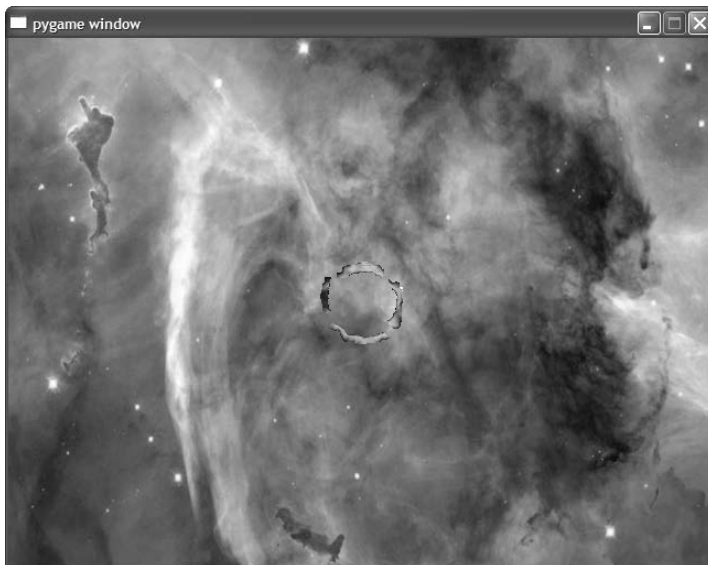
Kolejny zestaw wierszy kodu obraca statek bezpośrednio do określonego kąta położenia poprzez przypisanie nowej wartości do właściwości `angle`. Kiedy użytkownik naciska klawisz 1, kod przypisuje 0 do właściwości `angle` i duszek przeskakuje do położenia pod kątem 0 stopni (jest to jego początkowa orientacja). Kiedy użytkownik naciska klawisz 2, kod przypisuje do właściwości `angle` wartość 90 i duszek przeskakuje do położenia pod kątem 90 stopni. Gdy użytkownik naciska klawisz 3, kod przypisuje do właściwości `angle` wartość 180 i duszek przeskakuje do położenia pod kątem 180 stopni. Wreszcie, gdy użytkownik naciska klawisz 4, kod przypisuje do właściwości `angle` wartość 270 i duszek przeskakuje do położenia pod kątem 270 stopni.

Tworzenie animacji

Przemieszczanie duszków i ich obracanie sprawia, że gra staje się bardziej ekscytująca, lecz dopiero animacja wnosi w nią prawdziwe życie. Na szczęście moduł `games` zawiera klasę do obsługi animacji, stosownie nazwaną `Animation`.

Prezentacja programu Eksplozja

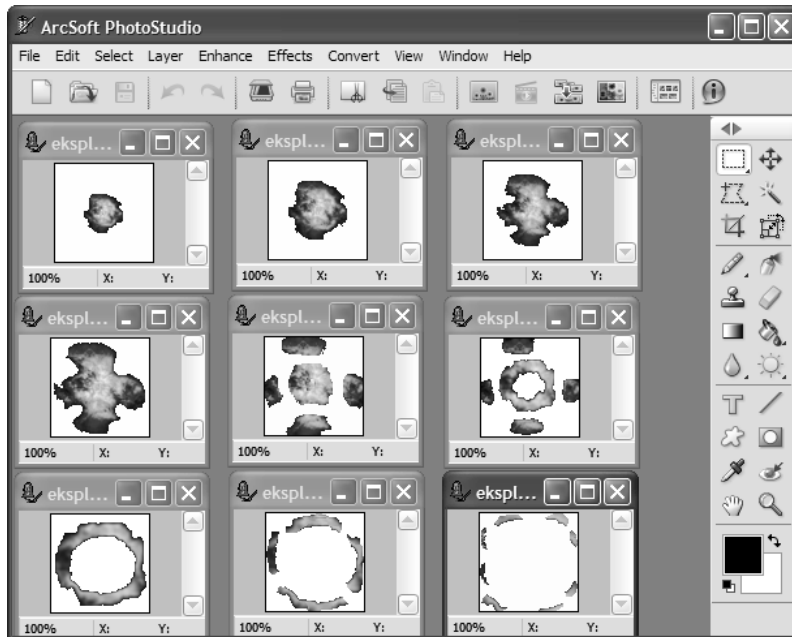
Program Eksplozja tworzy animację wybuchu w środku ekranu graficznego. Animacja jest odtwarzana w sposób ciągły, żebyś mógł się jej dobrze przyjrzeć. Kiedy skończysz już podziwiać ten bombowy efekt, możesz zakończyć program przez zamknięcie okna graficznego. Na rysunku 12.5 prezentuję migawkę programu w akcji.



Rysunek 12.5. Chociaż trudno to stwierdzić na podstawie nieruchomego obrazu, w centrum okna graficznego wykonywana jest animacja wybuchu

Przegląd obrazów eksplozji

Animacja to sekwencja obrazów (zwanymi także **ramkami** — ang. *frames*) wyświetlanych jeden po drugim. Utworzyłem sekwencję dziewięciu obrazów, które, kiedy są wyświetlane jeden po drugim, tworzą wrażenie ognistej eksplozji. Wszystkie dziewięć obrazków pokazałem na rysunku 12.6.



Rysunek 12.6. Gdy te dziewięć ramek zostanie wyświetlonych w szybkim tempie jedna po drugiej, otrzymamy wrażenie eksplozji

Kod tego programu możesz znaleźć na stronie internetowej tej książki (<https://www.helion.pl/ksiazki/pytd3v.htm>), w folderze rozdziału 12.; nazwa pliku to *eksplozja.py*.

Rozpoczęcie programu

Jak zawsze, na początku programu importuję potrzebne moduły i wywołuję funkcję inicjalizującą ekran graficzny:

```
# Eksplozja
# Demonstruje tworzenie animacji

from livewires import games

games.init(screen_width = 640, screen_height = 480, fps = 50)
```

Potem ustawiam tło ekranu graficznego:

```
nebula_image = games.load_image("mgławica.jpg", transparent = 0)
games.screen.background = nebula_image
```

Utworzenie listy plików z obrazami

Konstruktor klasy `Animation` pobiera listę nazw plików zawierających obrazy lub listę obiektów obrazu reprezentującą sekwencję obrazów do wyświetlenia. Więc w następnej kolejności tworzę listę nazw plików z obrazami, które zostały pokazane na rysunku 12.6:

```
explosion_files = ["eksplozja1.bmp",
                  "eksplozja2.bmp",
                  "eksplozja3.bmp",
                  "eksplozja4.bmp",
                  "eksplozja5.bmp",
                  "eksplozja6.bmp",
                  "eksplozja7.bmp",
                  "eksplozja8.bmp",
                  "eksplozja9.bmp"]
```

Utworzenie obiektu klasy `Animation`

W końcu tworzę obiekt klasy `Animation` i dodaję go do ekranu:

```
explosion = games.Animation(images = explosion_files,
                            x = games.screen.width/2,
                            y = games.screen.height/2,
                            n_repeats = 0,
                            repeat_interval = 5)
games.screen.add(explosion)
```

`Animation` jest klasą pochodną klasy `Sprite`, więc dziedziczy wszystkie jej atrybuty, właściwości i metody. Podobnie jak w przypadku wszystkich duszków, możesz podać współrzędne x i y , aby zdefiniować umiejscowienie animacji. W powyższym kodzie przekazuję współrzędne do konstruktora klasy, tak aby animacja była utworzona w środku ekranu.

Animacja tym się różni od duszka, że występuje w niej lista obrazów, która jest przetwarzana cyklicznie. Więc musisz dostarczyć listę nazw plików graficznych w postaci łańcuchów znaków albo listę obiektów obrazu reprezentujących obrazy, które mają być wyświetlane. Ja dostarczam listę `explosion_files` z łańcuchami reprezentującymi nazwy plików graficznych poprzez parametr `images`.

Atrybut `n_repeats` obiektu określa, ile razy animacja (jako sekwencja jej wszystkich obrazów) zostanie wyświetlona. Wartość domyślna atrybutu `n_repeats` wynosi 0. Ponieważ przekazuję 0 do `n_repeats`, cykl animacji eksplozji będzie powtarzany bez końca (lub przynajmniej do momentu zamknięcia okna graficznego).

Atrybut `repeat_interval` obiektu reprezentuje opóźnienie między następującymi po sobie obrazami. Większa liczba oznacza większe opóźnienie między ramkami, skutkujące

wolniejszą animacją. Mniejsza liczba reprezentuje mniejszą zwłokę, generując szybszą animację. Ja przekazuję do atrybutu `repeat_interval` wartość 5, aby uzyskać prędkość, jaką uważam za właściwą do wygenerowania przekonującej eksplozji.

Ostatnią, lecz równie ważną czynnością jest uruchomienie programu poprzez wywołanie metody `mainloop()` obiektu `screen`:

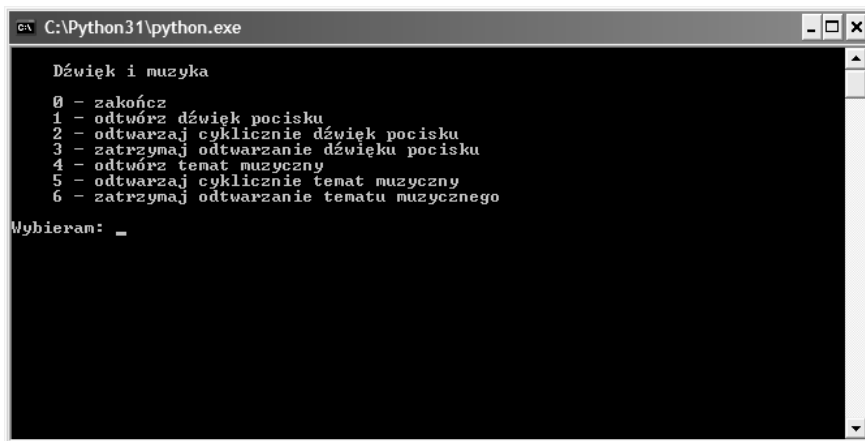
```
games.screen.mainloop()
```

Wykorzystywanie dźwięku i muzyki

Dźwięk i muzyka dodają nowy, oddziałujący na zmysły wymiar do programów. Ładowanie, odtwarzanie, powtarzanie w pętli i zatrzymywanie dźwięku i muzyki są łatwe do wykonania za pomocą modułu `games`. Chociaż ludzie mogliby się spierać na temat różnicy między dźwiękiem a muzyką, na gruncie pakietu `livewires` nie ma żadnej takiej dyskusji — istnieje w nim wyraźne rozróżnienie między tymi dwoma elementami.

Prezentacja programu Dźwięk i muzyka

Program Dźwięk i muzyka umożliwia użytkownikowi odtwarzanie, powtarzanie w pętli i zatrzymywanie efektu dźwiękowego wystrzelonego pocisku oraz tematu muzycznego z gry *Astrocraash*. Użytkownik może nawet odtwarzać obydwa te elementy jednocześnie. Rysunek 12.7 pokazuje uruchomiony program (lecz niestety nie jest w stanie wygenerować dźwięku).



Rysunek 12.7. Program umożliwia użytkownikowi odtworzenie dźwięku i fragmentu muzyki

Wskazówka

Kiedy uruchomisz ten program, będzie Ci potrzebna interakcja z oknem konsoli. Powinieneś umieścić okno konsoli w takiej pozycji, aby nie było ono zakryte przez okno graficzne. Możesz zignorować lub nawet zminimalizować utworzone przez program okno graficzne.

Kod tego programu możesz znaleźć na stronie internetowej tej książki (<https://www.helion.pl/ksiazki/pytd3v.htm>), w folderze rozdziału 12.; nazwa pliku to `dzwiek_i_muzyka.py`.

Praca z dźwiękami

Możesz utworzyć obiekt dźwiękowy do użytku programu poprzez załadowanie pliku typu WAV. Format WAV znakomicie się nadaje do zapisu efektów dźwiękowych, ponieważ może zostać użyty do zakodowania wszystkiego, co zarejestrujesz za pomocą mikrofonu.

Załadowanie dźwięku

Program rozpoczynam jak zawsze:

```
# Dźwięk i muzyka
# Demonstruje odtwarzanie plików dźwiękowych i muzycznych

from livewires import games

games.init(screen_width = 640, screen_height = 480, fps = 50)
```

Potem ładuję plik WAV, wykorzystując funkcję `load_sound()` modułu `games`.

```
# załaduj plik dźwiękowy
missile_sound = games.load_sound("pocisk.wav")
```

Funkcja przyjmuje łańcuch znaków reprezentujący nazwę pliku dźwiękowego, który ma zostać załadowany. Ładuję plik `pocisk.wav` i przypisuję nowo utworzony obiekt dźwiękowy do zmiennej `missile_sound`.

Pułapka

Za pomocą funkcji `load_sound()` można ładować tylko pliki WAV.

Następnie ładuję plik muzyczny:

```
# załaduj plik muzyczny
games.music.load("temat.mid")
```

Omówienie sposobu obsługi muzyki odłożę do momentu, gdy skończę demonstrowanie dźwięków.

Odtworzenie dźwięku

Następnie tworzę menu, z czym po raz pierwszy spotkałeś się w rozdziale 5.:

```
choice = None
while choice != "0":

    print(
        """
    Dźwięk i muzyka

    0 - zakończ
    1 - odtwórz dźwięk pocisku
    2 - odtwarzaj cyklicznie dźwięk pocisku
    3 - zatrzymaj odtwarzanie dźwięku pocisku
    4 - odtwórz temat muzyczny
    5 - odtwarzaj cyklicznie temat muzyczny
    6 - zatrzymaj odtwarzanie tematu muzycznego
    """
    )

    choice = input("Wybieram: ")
    print()

    # wyjdź
    if choice == "0":
        print("Żegnaj!")
```

Jeśli użytkownik wprowadzi 0, program pożegna użytkownika i zakończy działanie. Poniższy kod obsługuje przypadek, w którym użytkownik wprowadza 1:

```
# odtwórz dźwięk pocisku
elif choice == "1":
    missile_sound.play()
    print("Odtworzenie dźwięku pocisku.")
```

Aby odtworzyć dźwięk jeden raz, wywołuję metodę `play()` obiektu dźwiękowego. Kiedy dźwięk jest odtwarzany, zajmuje jeden z ośmiu dostępnych kanałów dźwiękowych. Aby odtworzyć dźwięk, potrzeba przynajmniej jednego otwartego kanału dźwiękowego. Kiedy zajętych jest wszystkich osiem kanałów, wywołanie metody `play()` obiektu dźwiękowego nie przyniesie żadnego efektu.

Jeśli wywołasz metodę `play()` obiektu dźwiękowego, który jest już odtwarzany, rozpocznie się odtwarzanie tego dźwięku na innym kanale, jeśli taki jest dostępny.

Cykliczne odtwarzanie dźwięku

Możesz odtwarzać dźwięk cyklicznie poprzez przekazanie liczby dodatkowych odtworzeń, jakie mają mieć miejsce, do metody `play()` obiektu. Jeśli na przykład przekażesz do `play()` liczbę 3, odpowiedni dźwięk zostanie odtworzony cztery razy (początkowe odtworzenie plus trzy powtórzenia). Możesz cyklicznie odtwarzać dźwięk bez końca po przekazaniu wartości -1 do metody `play()`.

Poniższy kod obsługuje przypadek, gdy użytkownik wprowadza 2:

```
# odtwarzaj cyklicznie dźwięk pocisku
elif choice == "2":
    loop = int(input("Ile razy powtórzyc odtwarzanie? (-1 = bez końca): "))
    missile_sound.play(loop)
    print("Cykliczne odtwarzanie dźwięku pocisku.")
```

W tym fragmencie kodu pobieram liczbę wskazującą, ile dodatkowo razy użytkownik chce usłyszeć odgłos pocisku, a następnie przekazuję tę wartość do metody `play()` obiektu dźwiękowego.

Zatrzymanie odtwarzania dźwięku

Odtwarzanie dźwięku przez obiekt dźwiękowy zatrzymuje się poprzez wywołanie metody `stop()`. Zatrzymuje ona ten konkretny dźwięk na wszystkich kanałach, na których jest odtwarzany. Jeśli wywołasz metodę `stop()` obiektu dźwiękowego, który w danej chwili nic nie odtwarza, przekonasz się, że pakiet `liveswires` jest tolerancyjny i nie poskarży się poprzez zgłoszenie błędu.

Jeśli użytkownik wprowadzi 3, poniższy kod zatrzyma odtwarzanie odgłosu pocisku (jeśli takowy jest właśnie odtwarzany):

```
# zatrzymaj odtwarzanie dźwięku pocisku
elif choice == "3":
    missile_sound.stop()
    print("Zatrzymanie odtwarzania dźwięku pocisku.")
```

Praca z muzyką

W pakiecie `liveswires` muzyka jest obsługiwana nieco inaczej niż dźwięk. Istnieje tylko jeden kanał muzyczny, więc w danym momencie jako bieżący plik muzyczny może zostać wyznaczony tylko jeden plik. Kanał muzyczny jest jednak bardziej elastyczny niż kanały dźwiękowe. Akceptuje on wiele różnych typów plików dźwiękowych, takich jak WAV, MP3, OGG i MIDI. Wreszcie, ponieważ istnieje tylko jeden kanał muzyczny, nie tworzy się nowego obiektu dla każdego pliku muzycznego. Zamiast tego masz dostęp do zestawu funkcji służących do ładowania, odtwarzania i zatrzymywania muzyki.

Załadowanie muzyki

Widziałeś kod odpowiedzialny za załadowanie pliku muzycznego w podpunkcie „Załadowanie dźwięku” punktu „Praca z dźwiękami”. Kod skorzystał z dostępu do obiektu `music` modułu `games`. To dzięki obiektowi `music` możesz załadować, odtworzyć i zatrzymać pojedynczą ścieżkę muzyczną.

Kod, którego użyłem do załadowania ścieżki muzycznej, `games.music.load("temat.mid")`, ustawia bieżącą muzykę na plik `temat.mid` typu MIDI. Muzykę ładuje się poprzez wywołanie metody `games.music.load()` i przekazanie do niej nazwy pliku muzycznego w postaci łańcucha znaków.

Masz dostępną tylko jedną ścieżkę muzyczną. Więc jeśli załadujesz nowy plik muzyczny, zastąpi on muzykę bieżącą.

Odtworzenie muzyki

Poniższy kod obsługuje przypadek, gdy użytkownik wprowadzi 4:

```
# odtwórz temat muzyczny
elif choice == "4":
    games.music.play()
    print("Odtworzenie tematu muzycznego.")
```

W rezultacie komputer odtwarza plik muzyczny, który załadowałem, temat.mid. Jeśli nie przekażesz żadnej wartości do metody `games.music.play()`, muzyka jest odtwarzana tylko raz.

Cykliczne odtwarzanie muzyki

Możesz odtwarzać muzykę cyklicznie, tyle razy, ile chcesz, po przekazaniu do metody `play()` liczby dodatkowych odtworzeń. Jeśli na przykład przekażesz wartość 3 do metody `games.music.play()`, muzyka zostanie odtworzona cztery razy (odtworzenie początkowe plus trzy powtórzenia). Możesz cyklicznie odtwarzać muzykę bez końca po przekazaniu wartości -1 do metody.

Poniższy kod obsługuje przypadek, w którym użytkownik wprowadza 5:

```
# odtwarzaj cyklicznie temat muzyczny
elif choice == "5":
    loop = int(input("Ile razy powtórzyć odtwarzanie? (-1 = bez końca): "))
    games.music.play(loop)
    print("Cykliczne odtwarzanie tematu muzycznego.")
```

W tym fragmencie kodu wczytuję liczbę dodatkowych odtworzeń tematu muzycznego, jakiej użytkownik chce posłuchać, a następnie przekazuję tę wartość do metody `play()`.

Zatrzymanie odtwarzania muzyki

Jeśli użytkownik wprowadzi opcję 6, poniższy kod zatrzyma odtwarzanie muzyki (jeśli faktycznie jest wykonywane):

```
# zatrzymaj odtwarzanie tematu muzycznego
elif choice == "6":
    games.music.stop()
    print("Zatrzymanie odtwarzania tematu muzycznego.")
```

Możesz spowodować zatrzymanie odtwarzania bieżącej muzyki poprzez wywołanie metody `games.music.stop()`, co właśnie w tym miejscu kodu robię. Jeśli wywołasz tę metodę, kiedy nie jest odtwarzana żadna muzyka, moduł `livewires` okazuje się wyrozumiały i nie generuje błędów.

Dokończenie programu

Wreszcie kończę program obsługą nieprawidłowego wyboru i oczekiwaniem na decyzję użytkownika:

```
# nieprzewidziany wybór
else:
    print("\nNiestety,", choice, "nie jest prawidłowym wyborem.")

input("\n\nAby zakończyć program, naciśnij klawisz Enter.")
```

Planowanie gry Astrocrash

Pora na powrót do projektu rozdziału — gry Astrocrash. Zamierzam pisać stopniowo coraz kompletniejszą wersję gry, aż osiągnę postać końcową, lecz nadal odczuwam potrzebę wymienienia kilku szczegółów programu, w tym głównych elementów gry, kilku niezbędnych klas i wymaganych zasobów multimedialnych.

Elementy gry

Chociaż moja gra jest oparta na klasycznej grze wideo, którą dobrze znam (a poznawałem ją etapami, drogą prób i błędów), wypisanie listy jej elementów jest nadal dobrym pomysłem:

- statek kosmiczny powinien obracać się i inicjować (lub przyspieszać) ruch do przodu w reakcji na klawisze naciśnięte przez gracza;
- statek powinien wystrzeliwać pociski po naciśnięciu przez gracza odpowiedniego klawisza;
- asteroidy powinny przelatywać przez ekran z różnymi prędkościami; mniejsze asteroidy powinny mieć generalnie wyższe prędkości niż większe;
- statek, wszystkie pociski i asteroidy powinny „przewijać się” przez brzegi ekranu — jeśli wyjdą poza granicę ekranu, powinny ukazać się po przeciwnej stronie;
- jeśli pocisk uderzy w dowolny inny obiekt na ekranie, powinien zniszczyć ten obiekt i sam siebie w efektownej, ognistej eksplozji;
- jeśli statek uderzy w dowolny inny obiekt na ekranie, powinien zniszczyć ten obiekt i sam siebie w efektownej, ognistej eksplozji;
- jeśli statek zostaje zniszczony, gra się kończy;
- jeśli zostaje zniszczona duża asteroida, powinny utworzyć się dwie asteroidy średniej wielkości; jeśli zostaje zniszczona asteroida średniego rozmiaru, powinny powstać dwie małe asteroidy; jeśli zostaje zniszczona mała asteroida, nie powstają już żadne nowe;
- za każdym razem, gdy gracz zniszczy asteroidę, jego dorobek punktowy powinien się zwiększyć; mniejsze asteroidy powinny być warte więcej punktów niż większe;
- liczba punktów uzyskanych przez gracza powinna być wyświetlana w prawym górnym rogu ekranu;
- kiedy tylko wszystkie asteroidy zostaną zniszczone, powinna zostać utworzona nowa, większa fala asteroidów.

Pomijam kilka elementów oryginału, aby zachować prostotę gry.

Klasy potrzebne w grze

Następnie sporządzam listę klas, które, jak sądzę, będą mi potrzebne:

- Ship,
- Missile,
- Asteroid,
- Explosion.

Już trochę wiem o tych klasach. Ship, Missile i Asteroid powinny być klasami pochodnymi klasy games.Sprite, podczas gdy Explosion powinna być klasą pochodną klasy games.Animation. Wiem też, że ta lista może ulec zmianie, kiedy teorię będę zamieniał w praktykę i gdy będę pisał kod gry.

Zasoby gry

Ponieważ gra zawiera dźwięk, muzykę, duszki i animację, wiem, że muszę utworzyć pewną liczbę plików multimedialnych. Oto lista, jaką udało mi się stworzyć:

- plik graficzny reprezentujący statek kosmiczny,
- plik graficzny reprezentujący pociski,
- trzy pliki graficzne, po jednym dla każdego rozmiaru asteroidy,
- seria plików graficznych do animacji eksplozji,
- plik dźwiękowy imitujący rozpędzanie statku,
- plik dźwiękowy z odgłosem wystrzeliwania pocisku,
- plik dźwiękowy imitujący eksplozję obiektu,
- plik z tematem muzycznym.

Utworzenie asteroidów

Ponieważ w grze mają występować śmiercionośne asteroidy, pomyślałem, że zacznę od nich. Choć wydaje się, że jest to dla mnie najlepszy wybór pierwszego kroku, w przypadku innego programisty może być inaczej — i jest to w porządku. Mógłbyś oczywiście wybrać inny pierwszy krok, taki jak umieszczenie na ekranie statku kosmicznego gracza. Nie istnieje jeden właściwy pierwszy krok. Najważniejszą rzeczą jest zdefiniowanie i wykonanie programów „na jeden kęs”, które, bazując jeden na drugim, wypracowują ścieżkę do kompletnego projektu.

Program Astrocrash01

Program Astrocrash01 tworzy okno graficzne, ustawia tło w postaci mgławicy i tworzy osiem asteroid w losowo wybranych miejscach. Prędkość każdej asteroidy jest również

obliczana z uwzględnieniem losowości, lecz mniejsze asteroidy mogą się poruszać szybciej niż większe. Na rysunku 12.8 pokazuję program w akcji.



Rysunek 12.8. Pole poruszających się asteroid stanowi podstawę gry

Kod tego programu możesz znaleźć na stronie internetowej tej książki (<https://www.helion.pl/ksiazki/pytd3v.htm>), w folderze rozdziału 12.; nazwa pliku to *astrocrash01.py*.

Rozpoczęcie programu

Program rozpoczyna się jak większość pozostałych:

```
# Astrocrash01
# Tworzy poruszające się po ekranie asteroidy

import random
from livewires import games

games.init(screen_width = 640, screen_height = 480, fps = 50)
```

Importuję moduł `random`, aby wygenerować współrzędne x i y dla asteroid.

Klasa Asteroid

Klasa Asteroid jest wykorzystywana do tworzenia poruszających się asteroid:

```
class Asteroid(games.Sprite):
    """ Asteroida przelatująca przez ekran. """
    SMALL = 1
    MEDIUM = 2
    LARGE = 3
    images = {SMALL : games.load_image("asteroida_mala.bmp"),
              MEDIUM : games.load_image("asteroida_sred.bmp"),
              LARGE : games.load_image("asteroida_duza.bmp") }

    SPEED = 2
```

Pierwszą moją czynnością jest zdefiniowanie stałych klasowych reprezentujących trzy różne wielkości asteroid: SMALL (mała), MEDIUM (średniego rozmiaru) i LARGE (duża). Następnie tworzę słownik z rozmiarami i odpowiadającymi im obiektami obrazów asteroid. W ten sposób mogę wykorzystać stałą reprezentującą rozmiar do znalezienia odpowiedniego obiektu obrazu. Na koniec tworzę stałą klasową o nazwie SPEED, której użyję jako podstawy do obliczenia ulosowanej prędkości każdej asteroidy.

Metoda `__init__()`

W następnej kolejności zajmuję się zdefiniowaniem konstruktora:

```
def __init__(self, x, y, size):
    """ Inicjalizuj duszka asteroidy. """
    super(Asteroid, self).__init__(
        image = Asteroid.images[size],
        x = x, y = y,
        dx = random.choice([1, -1]) * Asteroid.SPEED * random.random()/size,
        dy = random.choice([1, -1]) * Asteroid.SPEED * random.random()/size)

    self.size = size
```

Wartość przekazana poprzez parametr `size` reprezentuje wielkość asteroidy i powinna być równa jednej ze stałych rozmiaru: `Asteroid.SMALL`, `Asteroid.MEDIUM` lub `Asteroid.LARGE`. Na podstawie wartości `size` pobierany jest odpowiedni obraz nowej asteroidy, który następnie zostaje przekazany do konstruktora klasy `Sprite` (ponieważ `Sprite` jest klasą nadrzędną klasy `Asteroid`). Do konstruktora klasy `Sprite` zostają również przekazane wartości `x` i `y` reprezentujące położenie kosmicznej skały, przekazane wcześniej do konstruktora klasy `Asteroid`.

Konstruktor klasy `Asteroid` generuje losowe wartości składowych prędkości nowego obiektu i przekazuje je do konstruktora klasy `Sprite`. Składowe prędkości mają wartości losowe, ale mniejsze asteroidy mogą się potencjalnie poruszać szybciej niż większe. W końcu konstruktor klasy `Asteroid` tworzy i inicjalizuje atrybut `size` obiektu.

Metoda update()

Metoda update() utrzymuje asteroidę w grze poprzez przeniesienie jej na przeciwny brzeg ekranu:

```
def update(self):
    """ Przenieś asteroidę na przeciwny brzeg ekranu. """
    if self.top > games.screen.height:
        self.bottom = 0

    if self.bottom < 0:
        self.top = games.screen.height

    if self.left > games.screen.width:
        self.right = 0

    if self.right < 0:
        self.left = games.screen.width
```

Funkcja main()

Na koniec funkcja main() ustawia tło w postaci mgławicy oraz tworzy osiem asteroid w przypadkowych miejscach ekranu:

```
def main():
    # ustaw tło
    nebula_image = games.load_image("mglawica.jpg")
    games.screen.background = nebula_image

    # utwórz 8 asteroid
    for i in range(8):
        x = random.randrange(games.screen.width)
        y = random.randrange(games.screen.height)
        size = random.choice([Asteroid.SMALL, Asteroid.MEDIUM, Asteroid.LARGE])
        new_asteroid = Asteroid(x = x, y = y, size = size)
        games.screen.add(new_asteroid)

    games.screen.mainloop()

# wystartuj!
main()
```

Obracanie statku

Aby wykonać swoje następne zadanie, wprowadzam statek kosmiczny gracza. Moim skromnym celem jest umożliwienie użytkownikowi obracania statku za pomocą klawiszy strzałek. Do pozostałych funkcji statku zamierzam zabrać się później.

Program Astrocrash02

Program Astrocrash02 stanowi rozszerzenie programu Astrocrash01. W nowej wersji tworzę w środku ekranu statek, który gracz może obracać. Jeśli gracz naciska klawisz strzałki w prawo, statek obraca się zgodnie z ruchem wskazówek zegara. Jeśli zaś gracz naciska klawisz strzałki w lewo, statek obraca się w kierunku przeciwnym do ruchu wskazówek zegara. Na rysunku 12.9 pokazuję ten program w działaniu.



Rysunek 11.9. Statek kosmiczny gracz stanowi teraz część akcji

Kod tego programu możesz znaleźć na stronie internetowej tej książki (<https://www.helion.pl/ksiazki/pytd3v.htm>), w folderze rozdziału 12.; nazwa pliku to *astrocrash02.py*.

Klasa Ship

Moim głównym zadaniem jest napisanie kodu klasy Ship reprezentującej statek kosmiczny gracza:

```
class Ship(games.Sprite):
    """ Statek kosmiczny gracza. """
    image = games.load_image("statek.bmp")
    ROTATION_STEP = 3
```

```
def update(self):
    """ Obróć statek zgodnie z naciśniętym klawiszem. """
    if games.keyboard.is_pressed(games.K_LEFT):
        self.angle -= Ship.ROTATION_STEP
    if games.keyboard.is_pressed(games.K_RIGHT):
        self.angle += Ship.ROTATION_STEP
```

Ta klasa jest podobna do klasy występującej w programie Obróć duszka z wcześniejszej części tego rozdziału, lecz istnieje kilka różnic. Po pierwsze, ładuję obraz statku i przypisuję uzyskany w ten sposób obiekt obrazu do zmiennej klasowej o nazwie `image`. Po drugie, wykorzystuję stałą klasową, `ROTATION_STEP`, do reprezentowania liczby stopni, o jaką statek się obraca.

Konkretyzacja obiektu klasy Ship

Moją ostatnią czynnością w tej nowej wersji gry jest konkretyzacja obiektu klasy `Ship` oraz dodanie go do ekranu. Tworzę nowy statek w funkcji `main()`:

```
# utwórz statek
the_ship = Ship(image = Ship.image,
                x = games.screen.width/2,
                y = games.screen.height/2)
games.screen.add(the_ship)
```

Poruszanie statku

W następnej wersji programu wprawiam statek w ruch. Gracz może nacisnąć strzałkę w górę, aby włączyć silnik statku. Dzięki temu na statek oddziałuje siła ciągu, pchając go w kierunku, jaki wskazuje przód statku. Ponieważ brak jest tarcia, statek kontynuuje poruszanie się, nie tracąc prędkości nadanej mu na początku przez gracza.

Program Astrocrash03

Kiedy gracz włącza silnik statku, program `Astrocrash03` zmienia prędkość statku w sposób zależny od położenia kąтового statku (czemu towarzyszy odpowiedni efekt dźwiękowy). Program został zilustrowany na rysunku 12.10.

Kod tego programu możesz znaleźć na stronie internetowej tej książki (<https://www.helion.pl/ksiazki/pytd3v.htm>), w folderze rozdziału 12.; nazwa pliku to `astrocrash03.py`.

Import modułu math

Moją pierwszą czynnością jest umieszczenie na początku programu instrukcji importującej nowy moduł:

```
import math, random
```



Rysunek 12.10. Teraz statek może się poruszać po ekranie

Moduł `math` zawiera znaczną liczbę funkcji i stałych matematycznych, ale niech Cię to nie przeraża. W tym programie użyję tylko kilku z nich.

Dodanie do klasy `Ship` zmiennej i stałej klasowej

Tworzę stałą klasową, `VELOCITY_STEP`, którą wykorzystam do zmiany prędkości statku:

```
VELOCITY_STEP = .03
```

Użycie większej liczby spowodowałoby szybsze przyspieszanie statku, podczas gdy mniejsza liczba sprawiłaby, że statek przyspieszałby wolniej.

Dodaję również nową zmienną klasową, `sound`, mającą reprezentować dźwięk towarzyszący przyspieszaniu statku:

```
sound = games.load_sound("przyspieszenie.wav")
```

Modyfikacja metody `update()` klasy `Ship`

Następnie dodaję nowy kod na końcu metody `update()` klasy `Ship`, aby sprawić, by statek się poruszał. Sprawdzam, czy gracz naciska klawisz strzałki w górę. Jeśli ma to miejsce, odtwarzam dźwięk przyspieszającego statku:

```
# zastosuj siłę ciągu przy naciśniętym klawiszu strzałki w górę
if games.keyboard.is_pressed(games.K_UP):
    Ship.sound.play()
```

Poza tym, gdy gracz naciska klawisz strzałki w górę, muszę zmieniać składowe prędkości statku (właściwości `dx` i `dy` obiektu klasy `Ship`). Więc jak, mając dany kąt położenia statku, mogę obliczyć wartość, o jaką powinienem zmienić każdą ze składowych prędkości? Odpowiedź daje trygonometria. Poczekaj, nie zamykaj z trzaskiem tej książki i nie uciekaj, gdzie Cię nogi poniosą, wykrzykując coś bez ładu i składu. Jak obiecałem, do tych obliczeń wykorzystam tylko dwie funkcję matematyczne w paru wierszach kodu.

Aby rozpocząć ten proces, obliczam kąt położenia statku po zamianie stopni na radiany:

```
# zmień składowe prędkości w zależności od kąta położenia statku
angle = self.angle * math.pi / 180 # zamień na radiany
```

Radian to tylko miara obrotu, podobnie jak stopień. Moduł `math` w języku Python wymaga, aby miary kątów były wyrażone w radianach (podczas gdy pakiet `liveswires` używa stopni), więc z tego powodu muszę dokonać konwersji. W obliczeniu wykorzystuję stałą `pi` modułu `math`, która reprezentuje liczbę π .

Kiedy już mam kąt położenia statku wyrażony w radianach, mogę obliczyć, o jaką wartość powinienem zmienić każdą ze składowych prędkości, wykorzystując funkcje `sin()` i `cos()` obliczające sinus i cosinus kąta. W poniższych wierszach zostają obliczone nowe wartości właściwości `dx` i `dy` obiektu:

```
self.dx += Ship.VELOCITY_STEP * math.sin(angle)
self.dy += Ship.VELOCITY_STEP * -math.cos(angle)
```

Zasadniczo `math.sin(angle)` reprezentuje procent siły ciągu powodujący zmianę prędkości statku w kierunku osi x , podczas gdy `-math.cos(angle)` reprezentuje procent siły ciągu zmieniający prędkość statku w kierunku osi y .

Pozostaje tylko zająć się granicami ekranu. Korzystam z tej samej strategii, której używałem w przypadku asteroid: statek wychodzący poza krawędź ekranu powinien wrócić po przeciwnej stronie. Prawdę mówiąc, kopiuję kod z metody `update()` klasy `Asteroid` i wklejam go na końcu metody `update()` w klasie `Ship`:

```
# przenieś statek na przeciwległy brzeg ekranu
if self.top > games.screen.height:
    self.bottom = 0

if self.bottom < 0:
    self.top = games.screen.height

if self.left > games.screen.width:
    self.right = 0

if self.right < 0:
    self.left = games.screen.width
```


Chociaż jest to skuteczne, kopiowanie i wklejanie dużych fragmentów kodu to zwykle oznaka słabości projektu. Wróć do tego kodu później, aby znaleźć bardziej eleganckie rozwiązanie.

Pułapka

Powtarzające się, duże porcje kodu powodują rozdęcie programów i sprawiają, że stają się one trudniejsze do konserwacji. Kiedy widzisz powtarzający się kod, to często pora na wprowadzenie nowej funkcji lub klasy. Pomyśl, jak mógłbyś skonsolidować kod w jednym miejscu i wywoływać go z innych części programu, w których powtarzający się kod aktualnie występuje.

Wystrzeliwanie pocisków

Następnie umożliwię statkowi wystrzeliwanie pocisków. Kiedy gracz naciska klawisz spacji, wystrzelony jest pocisk z działa statku, który leci w kierunku wskazywanym przez przód statku. Pocisk powinien niszczyć wszystko, w co uderza, ale aby nie komplikować spraw, odkładam frajdę niszczenia do jednej z późniejszych wersji programu.

Program Astrocrash04

Program Astrocrash04 pozwala graczowi na wystrzeliwanie pocisków poprzez naciśnięcie klawisza spacji, lecz jest z tym pewien problem. Jeśli gracz przytrzymuje naciśnięty klawisz spacji, ze statku wylatuje strumień pocisków w tempie około 50 na sekundę. Muszę ograniczyć tempo wystrzeliwania pocisków, lecz zostawiam ten problem do następnej wersji gry. Na rysunku 12.11 przedstawiłem program Astrocrash04 z pełnym realizmem.

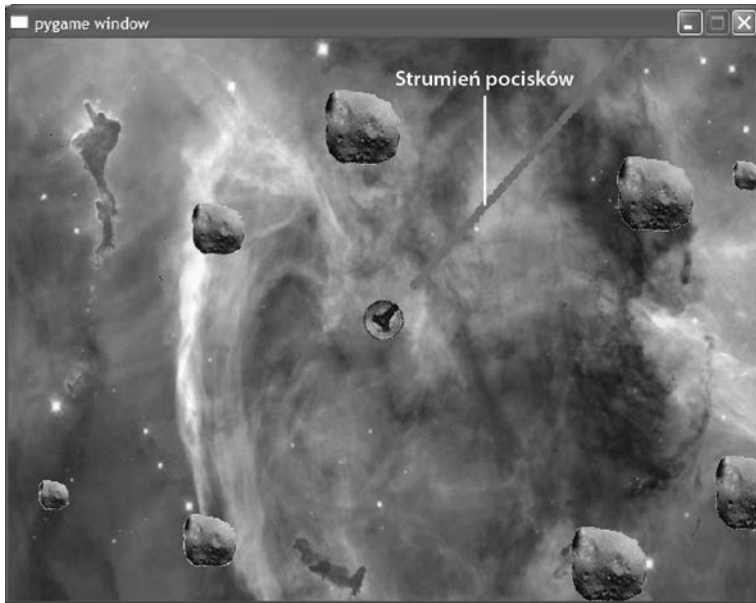
Kod tego programu możesz znaleźć na stronie internetowej tej książki (<https://www.helion.pl/ksiazki/pytd3v.htm>), w folderze rozdziału 12.; nazwa pliku to *astrocrash04.py*.

Modyfikacja metody update() klasy Ship

Modyfikuję metodę `update()` klasy `Ship` poprzez dodanie kodu, dzięki któremu statek może wystrzeliwać pociski. Jeśli gracz naciśnie klawisz spacji, tworzony jest nowy pocisk:

```
# wystrzel pocisk, jeśli jest naciśnięty klawisz spacji
if games.keyboard.is_pressed(games.K_SPACE) or True:
    new_missile = Missile(self.x, self.y, self.angle)
    games.screen.add(new_missile)
```

Oczywiście, aby skonkretyzować nowy obiekt przy użyciu wyrażenia `Missile(self.x, self.y, self.angle)`, muszę napisać taką drobną rzecz... jak klasa `Missile`.



Rysunek 12.11. Tempo wystrzeliwania pocisków jest zbyt duże

Klasa Missile

Piszę kod klasy `Missile` mającej reprezentować pociski wystrzeliwane przez statek. Zaczynam od utworzenia zmiennych i stałych klasowych:

```
class Missile(games.Sprite):
    """ Pocisk wystrzelony przez statek gracza. """
    image = games.load_image("pocisk.bmp")
    sound = games.load_sound("pocisk.wav")
    BUFFER = 40
    VELOCITY_FACTOR = 7
    LIFETIME = 40
```

Zmienna `image` ma reprezentować pocisk — pełne, czerwone kółko. Zmienna `sound` reprezentuje efekt dźwiękowy wystrzeliwania pocisku. Stała `BUFFER` definiuje odległość miejsca utworzenia nowego pocisku od statku (żeby pocisk nie został utworzony na wierzchu statku). Stała `VELOCITY_FACTOR` wpływa na szybkość lotu pocisku. Wreszcie stała `LIFETIME` określa długość czasu istnienia pocisku przed jego zniknięciem (żeby pocisk nie latał bez końca po ekranie).

Metoda `__init__()`

Rozpoczynam kod konstruktora klasy od następujących wierszy:

```
def __init__(self, ship_x, ship_y, ship_angle):
    """ Inicjalizuj duszka pocisku. """
```

Możesz być zaskoczony tym, że konstruktor obiektu pocisku wymaga podania wartości współrzędnych x i y statku oraz jego kąta położenia, które są przyjmowane przez parametry `ship_x`, `ship_y` oraz `ship_angle`. Metoda potrzebuje tych wartości, aby ustalić dwie rzeczy: dokładne miejsce pierwszego pojawienia się pocisku oraz składowe jego prędkości. To, gdzie tworzony jest pocisk, zależy od miejsca statku, a to, jak pocisk leci, zależy od jego położenia kąтового.

Następnie odtwarzam efekt dźwiękowy wystrzeliwania pocisku:

```
Missile.sound.play()
```

Potem wykonuję trochę obliczeń, aby określić początkowe położenie nowego pocisku:

```
# zamień na radiany
angle = ship_angle * math.pi / 180

# oblicz pozycję początkową pocisku
buffer_x = Missile.BUFFER * math.sin(angle)
buffer_y = Missile.BUFFER * -math.cos(angle)
x = ship_x + buffer_x
y = ship_y + buffer_y
```

Uzyskuję miarę kąta położenia statku wyrażoną w radianach. Następnie obliczam współrzędne początkowe pocisku, x i y , na podstawie kąta położenia statku i wartości stałej `Missile.BUFFER`. Uzyskane wartości x i y umieszczają pocisk dokładnie przed działem statku.

Następnie obliczam składowe prędkości pocisku. Stosuję ten sam typ obliczeń, którego użyłem w klasie `Ship`:

```
# oblicz składowe prędkości pocisku
dx = Missile.VELOCITY_FACTOR * math.sin(angle)
dy = Missile.VELOCITY_FACTOR * -math.cos(angle)
```

Wywołuję konstruktor klasy `Sprite` na rzecz bieżącego obiektu:

```
# utwórz pocisk
super(Missile, self).__init__(image = Missile.image,
                               x = x, y = y,
                               dx = dx, dy = dy)
```

W końcu dodaję do obiektu klasy `Missile` atrybut `lifetime`, żeby obiekt nie pojawiał się na ekranie bez końca.

```
self.lifetime = Missile.LIFETIME
```

Metoda `update()`

Następnie piszę kod metody `update()`. Oto jego pierwsza część:

```
def update(self):
    """ obsługuje ruch pocisku. """
    # jeśli wyczerpał się czas życia pocisku, zniszcz go
    self.lifetime -= 1
    if self.lifetime == 0:
        self.destroy()
```

Powyższy kod odlicza czas życia pocisku. Zmniejszana jest wartość atrybutu `lifetime`. Kiedy osiągnie 0, obiekt klasy `Missile` dokonuje samozniszczenia.

W drugiej części metody `update()` zawarłem znajomy kod przenoszący pocisk na przeciwległy brzeg ekranu:

```
# przenieś pocisk na przeciwległy brzeg ekranu
if self.top > games.screen.height:
    self.bottom = 0

if self.bottom < 0:
    self.top = games.screen.height

if self.left > games.screen.width:
    self.right = 0

if self.right < 0:
    self.left = games.screen.width
```

Widzę, że powyższy kod został już w moim programie trzy razy powtórzony. Zdecydowanie będę go później konsolidował.

Regulowanie tempa wystrzeliwania pocisków

Jak widziałeś w poprzednim programie, statek może wystrzelić około 50 pocisków w ciągu sekundy. Nawet dla gracza, który chce wygrać, jest to trochę za dużo. Więc w tej kolejnej wersji gry nakładam ograniczenie na tempo wystrzeliwania pocisków.

Program Astrocrash05

Program `Astrocrash05` ogranicza tempo wystrzeliwania pocisków poprzez utworzenie mechanizmu odliczania, który wymusza zwłokę pomiędzy wystrzałami. Kiedy odliczanie się kończy, gracz może wystrzelić kolejny pocisk. Program został zilustrowany na rysunku 12.12.

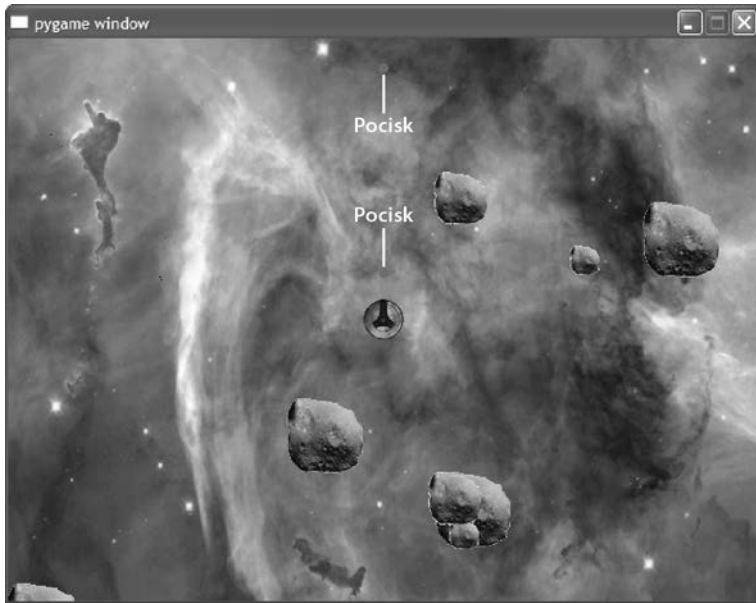
Kod tego programu możesz znaleźć na stronie internetowej tej książki (<https://www.helion.pl/ksiazki/pytd3v.htm>), w folderze rozdziału 12.; nazwa pliku to `astrocrash05.py`.

Dodanie stałej klasowej do klasy Ship

Moim pierwszym krokiem w wymuszeniu zwłoki między wystrzeliwaniem pocisków jest dodanie do klasy `Ship` stałej klasowej:

```
MISSILE_DELAY = 25
```

Stała `MISSILE_DELAY` reprezentuje czas zwłoki, jaki gracz musi odczekać między wystrzeliwaniem pocisków. Wykorzystuję ją do ponownego ustawiania odliczania, które zmusza gracza do czekania.



Rysunek 12.12. Teraz statek wystrzeliwuje pociski w rozsądniejszym tempie

Tworzenie konstruktora klasy Ship

Następnie tworzę konstruktor dla klasy:

```
def __init__(self, x, y):
    """ Inicjalizuj duszka statku. """
    super(Ship, self).__init__(image = Ship.image, x = x, y = y)
    self.missile_wait = 0
```

Metoda przyjmuje wartości reprezentujące współrzędne x i y nowego statku i przekazuje je dalej do klasy nadrzędnej klasy Ship, `games.Sprite`. W kolejnym wierszu dodaję do nowego obiektu atrybut o nazwie `missile_wait`. Wykorzystuję go do odliczania czasu zwłoki, zanim gracz będzie mógł wystrzelić kolejny pocisk.

Modyfikacja metody update() klasy Ship

Dodaję do metody `update()` klasy Ship kod, który zmniejsza wartość atrybutu `missile_wait` obiektu w ramach odliczania do 0.

```
# jeśli czekasz, aż statek będzie mógł wystrzelić następny pocisk,
# zmniejsz czas oczekiwania
if self.missile_wait > 0:
    self.missile_wait -= 1
```

Następnie zmieniam kod z poprzedniej wersji gry obsługujący wystrzeliwanie pocisków na poniższe wiersze:

```
# wystrzel pocisk, jeśli klawisz spacji jest naciśnięty i skończył się
# czas oczekiwania
if games.keyboard.is_pressed(games.K_SPACE) and self.missile_wait == 0:
    new_missile = Missile(self.x, self.y, self.angle)
    games.screen.add(new_missile)
    self.missile_wait = Ship.MISSILE_DELAY
```

Teraz, kiedy gracz naciśnie klawisz spacji, zanim statek wystrzeli nowy pocisk, musi się zakończyć odliczanie (wartość `missile_wait` musi być równa 0). Natychmiast po wystrzeleniu pocisku ustawiam atrybut `missile_wait` z powrotem na wartość `MISSILE_DELAY`, aby rozpocząć na nowo odliczanie.

Obsługa kolizji

Jak dotąd gracz może przemieszczać statek po polu asteroid, a nawet wystrzeliwać pociski, ale żaden z obiektów nie wchodzi w interakcję z innymi. Zmieniam ten stan rzeczy w kolejnej wersji gry. Kiedy pocisk zderza się z dowolnym innym obiektem, niszczy zarówno ten obiekt, jak i samego siebie. Ta sama zasada obowiązuje w przypadku kolizji statku kosmicznego z innym obiektem. Asteroidy będą w tym układzie pasywne, ponieważ nie chcę, aby zachodzące na siebie asteroidy niszczyły się wzajemnie.

Program Astrocrash06

Program `Astrocrash06` realizuje całe niezbędne wykrywanie kolizji dzięki wykorzystaniu właściwości `overlapping_sprites` klasy `Sprite`. Muszę też obsługiwać niszczenie asteroid w specjalny sposób, ponieważ kiedy są niszczone asteroidy dużej i średniej wielkości, tworzone są w miejsce każdej z nich dwie nowe, lecz mniejsze.

Pułapka

Ponieważ asteroidy są początkowo generowane w losowo wybranych miejscach, istnieje możliwość, że któraś z nich zostanie utworzona na wierzchu statku kosmicznego gracza, niszcząc statek już w momencie rozpoczęcia programu. Muszę tymczasowo pogodzić się z tą niedogodnością, ale będę musiał rozwiązać ten problem w ostatecznej wersji gry.

Program w działaniu został pokazany na rysunku 12.13.

Kod tego programu możesz znaleźć na stronie internetowej tej książki (<https://www.helion.pl/ksiazki/pytd3v.htm>), w folderze rozdziału 12.; nazwa pliku to `astrocrash06.py`.

Modyfikacja metody `update()` klasy `Missile`

Na końcu metody `update()` klasy `Missile` dodaję następujący kod:

```
# sprawdź, czy pocisk zachodzi na jakiś inny obiekt
if self.overlapping_sprites:
```



Rysunek 12.13. Teraz pociski statku kosmicznego niszczą asteroidy, ale uważaj — asteroidy mogą zniszczyć statek

```
for sprite in self.overlapping_sprites:
    sprite.die()
self.die()
```

Jeśli pocisk zachodzi na jakieś inne obiekty, zarówno w kontekście tych innych obiektów, jak i pocisku, jest wywoływana metoda `die()`. Jest to nowa metoda, którą dodam do klas `Asteroid`, `Ship` i `Missile`.

Dodanie metody `die()` do klasy `Missile`

W klasie `Missile`, podobnie jak w każdej innej klasie w tej wersji gry, jest potrzebna metoda `die()`. Metoda jest prawie tak prosta, jak tylko to możliwe:

```
def die(self):
    """ Zniszcz pocisk. """
    self.destroy()
```

Kiedy zostaje wywołana metoda `die()` obiektu klasy `Missile`, obiekt sam się niszczy.

Modyfikacja metody update() klasy Ship

Na końcu metody update() klasy Ship dodajemy następujący kod:

```
# sprawdź, czy statek nie zachodzi na jakiś inny obiekt
if self.overlapping_sprites:
    for sprite in self.overlapping_sprites:
        sprite.die()
    self.die()
```

Jeśli statek zachodzi na jakieś inne obiekty, zarówno w kontekście tych innych obiektów, jak i statku jest wywoływana metoda die(). Zwróć uwagę, że dokładnie taki sam kod pojawia się również w metodzie update() klasy Missile. Jak już wcześniej wspomniałem, kiedy widzisz zdublowany kod, powinieneś pomyśleć o tym, jak go skonsolidować. W następnej wersji gry pozbędę się zarówno tego, jak i innych fragmentów redundantnego kodu.

Dodanie metody die() do klasy Ship

Ta metoda jest identyczna jak metoda die() w klasie Missile:

```
def die(self):
    """ Zniszcz statek. """
    self.destroy()
```

Gdy zostaje wywołana metoda die() obiektu klasy Ship, obiekt sam się niszczy.

Dodanie stałej klasowej do klasy Asteroid

Do klasy Asteroid dodajemy jedną stałą klasową:

```
SPAWN = 2
```

Stała SPAWN określa liczbę nowych asteroid, jakie powstają po zniszczeniu asteroidy macierzystej.

Dodanie metody die() do klasy Asteroid

```
def die(self):
    """ Zniszcz asteroidę. """
    # jeśli nie jest to mała asteroida, zastąp ją dwoma mniejszymi
    if self.size != Asteroid.SMALL:
        for i in range(Asteroid.SPAWN):
            new_asteroid = Asteroid(x = self.x,
                                   y = self.y,
                                   size = self.size - 1)
            games.screen.add(new_asteroid)
    self.destroy()
```


Utrudnienie, jakie dodaję w tym miejscu, polega na tym, że metoda `die()` klasy `Asteroid` zawiera potencjał tworzenia nowych obiektów tej klasy. Metoda sprawdza, czy niszczona asteroida nie należy do kategorii małych asteroid. Jeśli do niej nie należy, zostają utworzone dwie nowe asteroidy, o jeden rozmiar mniejsze, w miejscu aktualnego położenia asteroidy macierzystej. Czy nowe asteroidy zostały utworzone, czy też nie, wcześniej istniejąca asteroida sama się niszczy i metoda się kończy.

Dodanie efektów eksplozji

W poprzedniej wersji gry gracz mógł niszczyć asteroidy, strzelając w nie pociskami, lecz ich destrukcji brakowało nieco wyrazu. Więc w następnym kroku dodaję do gry eksplozje.

Program `Astrocraash07`

W programie `Astrocraash07` piszę nową klasę, opartą na klasie `games.Animation`, obsługującą animowane eksplozje. Wykonuję też pewną pracę niewidoczną dla użytkownika, konsolidując redundantny kod. Nawet jeśli gracz nie doceni tych dodatkowych zmian, to i tak są one ważne. Na rysunku 12.14 pokazuję nowy program w akcji.



Rysunek 12.14. Teraz wszystkim przypadkom destrukcji w grze towarzyszą potężne eksplozje

Kod tego programu możesz znaleźć na stronie internetowej tej książki (<https://www.helion.pl/ksiazki/pytd3v.htm>), w folderze rozdziału 12.; nazwa pliku to `astrocrash07.py`.

Klasa Wrapper

Rozpaczynam od pracy na zapleczu. Tworzę nową klasę, `Wrapper`, opartą na klasie `games.Sprite`.

Metoda `update()`

Klasa `Wrapper` zawiera metodę `update()`, która po przekroczeniu przez obiekt krawędzi ekranu automatycznie przenosi go na krawędź przeciwną, tak że jego tor ruchu jakby „owija” ekran:

```
class Wrapper(games.Sprite):
    """ Duszek, którego tor lotu owija się wokół ekranu. """
    def update(self):
        """ Przenieś duszka na przeciwny brzeg ekranu. """
        if self.top > games.screen.height:
            self.bottom = 0

        if self.bottom < 0:
            self.top = games.screen.height

        if self.left > games.screen.width:
            self.right = 0

        if self.right < 0:
            self.left = games.screen.width
```

Powyższy kod widziałeś już kilkakrotnie. Powoduje owinięcie toru lotu duszka wokół ekranu. Kiedy teraz oprę pozostałe klasy występujące w tej grze na klasie `Wrapper`, jej metoda `update()` może utrzymywać instancje tych pozostałych klas w obrębie ekranu — a kod może istnieć tylko w jednym miejscu!

Metoda `die()`

Kod klasy kończę metodą `die()`, która niszczy obiekt:

```
def die(self):
    """ Zniszcz się. """
    self.destroy()
```

Klasa `Collider`

Następnie biorę się za inny redundantny kod. Zauważyłem, że zarówno klasa `Ship`, jak i `Missile` dzielą takie same instrukcje obsługujące kolizje, więc postanowiłem utworzyć

nową klasę, `Collider` (opartą na klasie `Wrapper`), reprezentującą obiekty, których tor lotu owija się wokół ekranu i które mogą się zderzać z innymi obiektami.

Metoda `update()`

Oto metoda `update()` obsługująca kolizje:

```
def update(self):
    """ Sprawdź, czy duszki nie zachodzą na siebie. """
    super(Collider, self).update()

    if self.overlapping_sprites:
        for sprite in self.overlapping_sprites:
            sprite.die()
        self.die()
```

Moją pierwszą czynnością w metodzie `update()` klasy `Collider` jest wywołanie metody `update()` jej nadklasy (czyli metody `update()` klasy `Wrapper`) w celu utrzymania obiektu w obrębie ekranu. Potem sprawdzam, czy nie występują kolizje. Jeśli nasz obiekt zachodzi na jakiegokolwiek inne obiekty, wywołuje metodę `die()` tych innych obiektów, a potem jego własną metodę `die()`.

Metoda `die()`

Następnie tworzę metodę dla opisywanej klasy, ponieważ wszystkie obiekty klasy `Collider` robią to samo, kiedy kończą swoje istnienie — tworzą eksplozję i niszczą się same.

```
def die(self):
    """ Zniszcz się i pozostaw po sobie eksplozję. """
    new_explosion = Explosion(x = self.x, y = self.y)
    games.screen.add(new_explosion)
    self.destroy()
```

W tej metodzie tworzę obiekt klasy `Explosion`. To nowa klasa, której obiektami są animacje eksplozji. Wkrótce ujrzysz tę klasę w pełni jej blasku.

Modyfikacja klasy `Asteroid`

Modyfikuję klasę `Asteroid` w taki sposób, aby była oparta na klasie `Wrapper`:

```
class Asteroid(Wrapper):
```

Teraz klasa `Asteroid` dziedziczy metodę `update()` po klasie `Wrapper` i dlatego wycinam jej własną metodę `update()`. Redundantny kod zaczyna znikać!

Jedyną rzeczą, jaką jeszcze robię w tej klasie, jest zmiana ostatniego wiersza jej metody `die()`. Wywołanie `self.die()` zastępuję wierszem:

```
super(Asteroid, self).die()
```

Odtąd zawsze, kiedy zmienię metodę `die()` klasy `Wrapper`, klasa `Asteroid` automatycznie zbierze wynikające z tego korzyści.

Modyfikacja klasy Ship

Modyfikuję klasę Ship tak, aby była oparta na klasie Collider:

```
class Ship(Collider):
```

W metodzie update() klasy Ship dodaję wiersz:

```
    super(Ship, self).update()
```

Mogę teraz wyciąć kilka dalszych fragmentów redundantnego kodu. Ponieważ kolizje obsługuje metoda update() klasy Collider, wycinam kod wykrywania kolizji z metody update() klasy Ship. A ponieważ metoda update() klasy Collider wywołuje metodę update() klasy Wrapper, z metody update() klasy Ship wycinam także kod obsługujący powracanie obiektu na ekran. Wycinam również z klasy Ship metodę die(), ponieważ dziedziczy ją ona po klasie Collider.

Modyfikacja klasy Missile

W trakcie modyfikacji klasy Missile, zmieniam nagłówek klasy w taki sposób, aby klasa była oparta na klasie Collider:

```
class Missile(Collider):
```

W metodzie update() klasy Missile dodaję wiersz

```
    super(Missile, self).update()
```

Podobnie jak w przypadku klasy Ship, mogę teraz wyciąć z klasy Missile redundantny kod. Ponieważ kolizje obsługuje metoda update() klasy Collider, wycinam kod wykrywania kolizji z metody update() klasy Missile. A ponieważ metoda update() klasy Collider wywołuje metodę update() klasy Wrapper, z metody update() klasy Missile wycinam także kod obsługujący powracanie obiektu na ekran. Wycinam również z klasy Missile metodę die(), ponieważ dziedziczy ją ona po klasie Collider.

Wskazówka

Aby pomóc sobie w zrozumieniu tych wszystkich zmian, jakie opisuję, zajrzyj do kompletnego kodu wszystkich wersji programu Astrocrash zamieszczonego na stronie internetowej tej książki www.courseptr.com/downloads.

Klasa Explosion

Ponieważ chcę tworzyć animowane eksplozje, napisałem klasę Explosion opartą na klasie games.Animation.

```
class Explosion(games.Animation):
    """ Animacja eksplozji. """
    sound = games.load_sound("eksplozja.wav")
    images = ["eksplozja1.bmp",
             "eksplozja2.bmp",
```

```
"eksplozja3.bmp",
"eksplozja4.bmp",
"eksplozja5.bmp",
"eksplozja6.bmp",
"eksplozja7.bmp",
"eksplozja8.bmp",
"eksplozja9.bmp"]
```

Definiuję zmienną klasową `sound` reprezentującą efekt dźwiękowy towarzyszący eksplozji. Definiuję także zmienną klasową `images`, przypisując do niej listę nazw plików zawierających dziewięć ramek służących do animacji eksplozji.

Następnie piszę konstruktor klasy `Explosion`.

```
def __init__(self, x, y):
    super(Explosion, self).__init__(images = Explosion.images,
                                   x = x, y = y,
                                   repeat_interval = 4, n_repeats = 1,
                                   is_collideable = False)

    Explosion.sound.play()
```

Konstruktor klasy `Explosion` przyjmuje wartości reprezentujące współrzędne ekranowe eksplozji poprzez parametry `x` i `y`. Kiedy wywołuję konstruktor nadklasy (`games.Animation`), przekazuję te wartości znów jako parametry `x` i `y`, aby animacja została odtworzona dokładnie tam, gdzie chcę. Do konstruktora klasy nadrzędnej przekazuję również poprzez parametr `images` listę nazw plików graficznych, `Explosion.images`. Parametrowi `n_repeats` nadaję wartość 1, aby animacja była odtwarzana tylko raz. A parametrowi `repeat_interval` nadaję wartość 4, aby szybkość animacji była taka jak należy. Parametrowi `is_collideable` nadaję wartość `False`, aby mogące się przydarzyć zachodzenie innych duszków na animację eksplozji nie liczyło się jako kolizja.

Na koniec odtwarzam efekt dźwiękowy eksplozji za pomocą wywołania metody `Explosion.sound.play()`.

Sztuczka

Pamiętaj, że do konstruktora klasy `games.Animation` możesz przekazać albo listę nazw plików, albo listę obiektów obrazu reprezentującą ramki animacji.

Dodanie poziomów gry, rejestracji wyników oraz tematu muzycznego

Do gry trzeba dodać jeszcze kilka elementów, aby mogła być postrzegana jako kompletna. Moim ostatnim posunięciem jest dodanie poziomów gry — co oznacza, że wtedy, gdy gracz zniszczy wszystkie asteroidy znajdujące się na ekranie, pojawia się nowa, liczniejsza grupa tych obiektów. Dodaję również funkcjonalność rejestracji dorobku punktowego gracza oraz wzmagający napięcie temat muzyczny, aby było można w pełni przeżywać grę.

Program Astrocrash08

Oprócz poziomów gry, rejestracji zdobytych punktów i tematu muzycznego dodają trochę kodu, którego efekty mogą być mniej oczywiste dla gracza, lecz który ma pomimo to istotne znaczenie dla kompletności programu. Na rysunku 12.15 pokazuję moją ostateczną wersję tej gry.



Rysunek 12.15. Ostateczny szlif umożliwia kontynuowanie gry tak długo, jak na to pozwalają umiejętności gracza

Kod tego programu możesz znaleźć na stronie internetowej tej książki (<https://www.helion.pl/ksiazki/pytd3v.htm>), w folderze rozdziału 12.; nazwa pliku to *astrocrash08.py*.

Import modułu color

Pierwszy dodatek jest dosyć prosty. Z pakietu *livewires* oprócz modułu *games* importuję moduł *color*:

```
from livewires import games, color
```

Potrzebuję modułu *color*, aby komunikat „Koniec gry” mógł zostać wyświetlony w ładnym, jaskrawoczerwonym kolorze.

Klasa Game

Pod koniec programu dodaję klasę `Game` — nową klasę, służącą do utworzenia obiektu reprezentującego samą grę. Tworzenie obiektu mającego reprezentować grę może się z początku wydawać nieco dziwnym pomysłem, ale nabierze ono sensu, gdy się nad tym dłużej zastanowisz. Sama gra mogłaby z pewnością stanowić obiekt z takimi metodami jak `play()`, służąca do rozpoczęcia gry, `advance()`, umożliwiająca podniesienie gry na kolejny poziom, oraz `end()`, pozwalająca zakończyć grę.

Decyzja projektowa o reprezentowaniu gry przez obiekt ułatwia innym obiektom przesyłanie do gry komunikatów. Na przykład w sytuacji, gdy na aktualnym poziomie gry zostaje zniszczona ostatnia asteroida, mogłaby przesłać do gry komunikat z żądaniem przejścia do następnego poziomu. Albo wtedy, gdy zostaje zniszczony statek, mógłby przesłać do gry komunikat, że powinna się zakończyć.

Kiedy będę omawiał klasę `Game`, zapewne zauważysz, że duża część kodu zawartego w funkcji `main()` została włączona do tej klasy.

Metoda `__init__()`

Pierwszą rzeczą, jaką robię w klasie `Game`, jest zdefiniowanie konstruktora:

```
class Game(object):
    """ Sama gra. """
    def __init__(self):
        """ Inicjalizuj obiekt klasy Game. """
        # ustaw poziom
        self.level = 0

        # załaduj dźwięk na podniesienie poziomu
        self.sound = games.load_sound("poziom.wav")

        # utwórz wynik punktowy
        self.score = games.Text(value = 0,
                                size = 30,
                                color = color.white,
                                top = 5,
                                right = games.screen.width - 10,
                                is_collideable = False)
        games.screen.add(self.score)

        # utwórz statek kosmiczny gracza
        self.ship = Ship(game = self,
                         x = games.screen.width/2,
                         y = games.screen.height/2)
        games.screen.add(self.ship)
```

Atrybut `level` reprezentuje aktualny numer poziomu gry. Atrybut `sound` odpowiada za efekt dźwiękowy towarzyszący podniesieniu poziomu gry. Atrybut `score` reprezentuje wynik punktowy gry — to obiekt klasy `Text`, który ukazuje się w prawym górnym rogu ekranu. Właściwość `is_collideable` tego obiektu ma wartość `False`, co oznacza, że wynik

nie występuje w żadnych kolizjach — więc statek gracza nie zderzy się z wynikiem i nie dojdzie do eksplozji! W końcu `ship` to atrybut reprezentujący statek kosmiczny gracza.

Metoda `play()`

Następnie definiuję metodę `play()`, która rozpoczyna grę.

```
def play(self):
    """ Przeprowadź grę. """
    # rozpocznij odtwarzanie tematu muzycznego
    games.music.load("temat.mid")
    games.music.play(-1)

    # załaduj i ustaw tło
    nebula_image = games.load_image("mgławica.jpg")
    games.screen.background = nebula_image

    # przejdź do poziomu 1
    self.advance()

    # rozpocznij grę
    games.screen.mainloop()
```

Metoda ta ładuje temat muzyczny i odtwarza go w niekończącej się pętli. Ładuje obraz mgławicy i ustawia go jako tło. Następnie wywołuje własną metodę obiektu klasy `Game` o nazwie `advance()`, która podnosi grę na kolejny poziom. (Wszystkiego o metodzie `advance()` dowiesz się w następnym podpunkcie). Na koniec metoda `play()` wywołuje metodę `games.screen.mainloop()`, aby całą grę wprowadzić w ruch!

Metoda `advance()`

Metoda `advance()` podnosi grę na kolejny poziom. Zwiększa numer poziomu, tworzy nową falę asteroid, wyświetla krótko na ekranie numer poziomu oraz odtwarza dźwięk obwieszczający zmianę poziomu gry.

Moja pierwsza czynność w tej metodzie jest dość prosta — zwiększam numer poziomu:

```
def advance(self):
    """ Przejdź do następnego poziomu gry. """
    self.level += 1
```

Następnie przechodzę do najciekawszej części metody — utworzenia nowej fali asteroid. Każdy poziom rozpoczyna się od liczby asteroid równej jego numerowi. Więc pierwszy poziom rozpoczyna się od jednej asteroidy, drugi — od dwóch itd. Mimo że utworzenie grupy asteroid jest proste, muszę uzyskać pewność, że żadna nowa asteroida nie zostanie utworzona na wierzchu statku kosmicznego. Inaczej statek wybuchnie w momencie rozpoczęcia nowego poziomu gry.

```
# wielkość obszaru ochronnego wokół statku przy tworzeniu asteroid
BUFFER = 150

# utwórz nowe asteroidy
```



```
for i in range(self.level):
    # oblicz współrzędne x i y zapewniające minimum odległości od statku
    # określ minimalną odległość wzdłuż osi x oraz wzdłuż osi y
    x_min = random.randrange(BUFFER)
    y_min = BUFFER - x_min

    # wyznacz odległość wzdłuż osi x oraz wzdłuż osi y
    # z zachowaniem odległości minimalnej
    x_distance = random.randrange(x_min, games.screen.width - x_min)
    y_distance = random.randrange(y_min, games.screen.height - y_min)

    # oblicz położenie na podstawie odległości
    x = self.ship.x + x_distance
    y = self.ship.y + y_distance

    # jeśli to konieczne, przeskocz między krawędziami ekranu
    x %= games.screen.width
    y %= games.screen.height

    # utwórz asteroidę
    new_asteroid = Asteroid(game = self,
                            x = x, y = y,
                            size = Asteroid.LARGE)
    games.screen.add(new_asteroid)
```

Stała `BUFFER` reprezentuje wielkość bezpiecznej strefy, jaką chcę mieć wokół statku.

Następnie uruchamiam pętlę. W trakcie każdej iteracji tworzę nową asteroidę w bezpiecznej odległości od statku.

Wartość zmiennej `x_min` wyznacza minimalną odległość miejsca utworzenia nowej asteroidy od statku obliczoną wzdłuż osi `x`, podczas gdy zmienna `y_min` reprezentuje minimalną odległość miejsca utworzenia nowej asteroidy od statku, obliczoną wzdłuż osi `y`. Wprowadzam zmienność poprzez wykorzystanie modułu `random`, ale suma wartości `x_min` i `y_min` zawsze jest równa stałej `BUFFER`.

Wartość zmiennej `x_distance` to odległość miejsca utworzenia nowej asteroidy wzdłuż osi `x`. Jest losowo wybraną liczbą, która jednak daje pewność, że nowa asteroida zostanie utworzona w odległości od statku co najmniej równej `x_min`. Natomiast zmienna `y_distance` reprezentuje odległość miejsca utworzenia nowej asteroidy obliczoną wzdłuż osi `y`. Jest losowo wybraną liczbą, która jednak daje pewność, że nowa asteroida zostanie utworzona w odległości od statku co najmniej równej `y_min`.

Wartość zmiennej `x` to współrzędna `x` nowej asteroidy. Obliczam ją poprzez dodanie liczby `x_distance` do wartości współrzędnej `x` statku kosmicznego. Potem upewniam się, że wartość `x` nie usytuuje asteroidy poza ekranem, wymuszając „obieganie” ekranu¹ za pomocą operatora modulo. Z kolei wartość zmiennej `y` to współrzędna `y` nowej asteroidy. Obliczam ją poprzez dodanie liczby `y_distance` do wartości współrzędnej `y` statku kosmicznego. Potem upewniam się, że wartość `y` nie umieszcza asteroidy poza ekranem,

¹ Można sobie wyobrazić ekran jako powierzchnię walca powstałego przez sklejenie jego prawej i lewej krawędzi — *przyj. tłum.*

wymuszając „obieganie” ekranu² za pomocą operatora modulo. Następnie wykorzystuję tak obliczone wartości x i y do utworzenia nowiuteńkiej asteroidy.

Zauważ, że w konstruktorze klasy `Asteroid` pojawił się nowy parametr, `game`. Zapamiętaj, że ponieważ każda asteroida musi mieć możliwość wywołania metody obiektu klasy `Game`, każdy obiekt klasy `Asteroid` musi zawierać referencję do obiektu klasy `Game`. Więc przekazuję obiekt `self` do parametru `game`, który zostanie wykorzystany przez konstruktor klasy `Asteroid` do zdefiniowania atrybutu reprezentującego grę.

Moje ostatnie czynności w metodzie `advance()` to wyświetlenie nowego numeru poziomu oraz odtworzenie dźwięku obwieszczającego podwyższenie poziomu gry:

```
# wyświetl numer poziomu
level_message = games.Message(value = "Poziom " + str(self.level),
                               size = 40,
                               color = color.yellow,
                               x = games.screen.width/2,
                               y = games.screen.width/10,
                               lifetime = 3 * games.screen.fps,
                               is_collideable = False)
games.screen.add(level_message)

# odtwórz dźwięk przejścia do nowego poziomu (nie dotyczy pierwszego poziomu)
if self.level > 1:
    self.sound.play()
```

Metoda `end()`

Metoda `end()` wyświetla dużymi, czerwonymi literami komunikat „Koniec gry” na środku ekranu przez mniej więcej pięć sekund. Potem gra się kończy i ekran graficzny zostaje zamknięty.

```
def end(self):
    """ Zakończ grę. """
    # pokazuj komunikat 'Koniec gry' przez 5 sekund
    end_message = games.Message(value = "Koniec gry",
                                size = 90,
                                color = color.red,
                                x = games.screen.width/2,
                                y = games.screen.height/2,
                                lifetime = 5 * games.screen.fps,
                                after_death = games.screen.quit,
                                is_collideable = False)
    games.screen.add(end_message)
```

² Tym razem można sobie wyobrazić ekran jako powierzchnię walca powstałego przez sklejenie jego dolnej i górnej krawędzi — *przyp. tłum.*

Dodanie do klasy Asteroid zmiennej i stałej klasowej

Dokonuję w klasie Asteroid kilku zmian związanych z dodaniem poziomów gry i rejestrowaniem wyniku. Dodaję stałą klasową POINTS:

```
POINTS = 30
```

Stała ta będzie odgrywać rolę wartości bazowej służącej do wyznaczenia liczby punktów określających wartość asteroidy. Rzeczywista wartość punktowa będzie modyfikowana zgodnie z wielkością asteroidy — mniejsze asteroidy będą warte więcej punktów niż większe.

Aby móc zmieniać poziom gry, program musi rozpoznać moment, w którym wszystkie asteroidy występujące na aktualnym poziomie gry zostały zniszczone. Dlatego też śledzę i rejestruję całkowitą liczbę asteroid za pomocą nowej zmiennej klasowej, total, którą definiuję w początkowej części kodu klasy:

```
total = 0
```

Modyfikacja konstruktora klasy Asteroid

W konstruktorze dodaję wiersz zwiększający wartość zmiennej Asteroid.total:

```
Asteroid.total += 1
```

Chcę teraz, aby każda asteroida miała możliwość przesyłania komunikatu do obiektu klasy Game, więc dostarczam do każdego obiektu klasy Asteroid referencję do obiektu klasy Game. W konstruktorze klasy Asteroid przyjmuję obiekt klasy Game poprzez utworzenie nowego parametru:

```
def __init__(self, game, x, y, size):
```

Parametr game przyjmuje obiekt klasy Game, który następnie wykorzystuję do utworzenia atrybutu w nowym obiekcie klasy Asteroid:

```
self.game = game
```

Tak więc każdy nowy obiekt klasy Asteroid ma atrybut game, który jest referencją do samej gry. Dzięki temu atrybutowi obiekt klasy Asteroid może wywołać metodę obiektu klasy Game, taką jak advance().

Modyfikacja metody die() klasy Asteroid

Wprowadzam kilka dodatków do metody die() w klasie Asteroid. Najpierw zmniejszam wartość zmiennej Asteroid.total:

```
Asteroid.total -= 1
```

Następnie zwiększam wynik punktowy na podstawie wartości stałej Asteroid.POINTS oraz rozmiaru asteroidy (mniejsze asteroidy są warte więcej punktów niż większe). Chcę mieć również pewność, że wynik będzie zawsze wyrównywany do prawej strony, więc ustawiam na nowo właściwość right obiektu score na 10 pikseli od prawej krawędzi ekranu.

```
self.game.score.value += int(Asteroid.POINTS / self.size)
self.game.score.right = games.screen.width - 10
```

Kiedy tworzę każdą z dwóch nowych asteroid, muszę przekazać referencję do obiektu klasy Game, czego dokonuję poprzez zmodyfikowanie pierwszego wiersza wywołania konstruktora klasy Asteroid:

```
new_asteroid = Asteroid(game = self.game,
```

Pod koniec metody die() klasy Asteroid badam wartość zmiennej Asteroid.total, aby sprawdzić, czy wszystkie asteroidy zostały zniszczone. Jeśli rzeczywiście tak jest, ostatnia asteroida wywołuje metodę advance() obiektu klasy Game, która przenosi grę na następny poziom oraz tworzy nową grupę asteroid.

```
# jeśli wszystkie asteroidy zostały zniszczone, przejdź do następnego poziomu
if Asteroid.total == 0:
    self.game.advance()
```

Dodanie stałej klasowej do klasy Ship

Wprowadzam kilka dodatków do klasy Ship. Tworzę stałą klasową VELOCITY_MAX, którą wykorzystuję do ograniczenia maksymalnej prędkości statku kosmicznego gracza:

```
VELOCITY_MAX = 3
```

Modyfikacja konstruktora klasy Ship

Podobnie jak obiekt klasy Asteroid, obiekt klasy Ship musi mieć dostęp do obiektu klasy Game, aby mógł wywołać jego metodę. Tak jak to zrobiłem w przypadku klasy Asteroid, modyfikuję konstruktor klasy Ship:

```
def __init__(self, game, x, y):
```

Nowy parametr, game, przyjmuje jako swoją wartość obiekt klasy Game, której potem używam do utworzenia atrybutu obiektu klasy Ship:

```
self.game = game
```

Więc każdy obiekt klasy Ship ma atrybut game, który jest referencją do samej gry. Dzięki temu atrybutowi obiekt klasy Ship może wywołać metodę obiektu klasy Game, taką jak end().

Modyfikacja metody update() klasy Ship

W metodzie update() klasy Ship, ograniczam poszczególne składowe prędkości obiektu tej klasy, dx i dy, wykorzystując stałą klasową VELOCITY_MAX:

```
# ogranicz prędkość w każdym kierunku
self.dx = min(max(self.dx, -Ship.VELOCITY_MAX), Ship.VELOCITY_MAX)
self.dy = min(max(self.dy, -Ship.VELOCITY_MAX), Ship.VELOCITY_MAX)
```

Powyższy kod daje pewność, że `dx` i `dy` nigdy nie będą miały wartości mniejszej niż `Ship.VELOCITY_MAX` oraz większej niż `Ship.VELOCITY_MAX`. Aby to osiągnąć, skorzystałem z funkcji `min()` i `max()`. Funkcja `min()` zwraca wartość minimalną dwóch liczb, podczas gdy funkcja `max()` zwraca wartość maksymalną dwóch liczb. Ograniczam prędkość statku, aby uniknąć potencjalnych problemów, łącznie z wpadaniem statku na swoje własne pociski.

Dodanie do klasy `Ship` metody `die()`

Kiedy statek gracza zostaje zniszczony, gra się kończy. Dodaję do klasy `Ship` metodę `die()`, która wywołuje metodę `end()` obiektu klasy `Game` w celu zakończenia gry.

```
def die(self):
    """ Zniszcz statek i zakończ grę. """
    self.game.end()
    super(Ship, self).die()
```

Funkcja `main()`

Teraz, skoro mam klasę `Game`, funkcja `main()` staje się całkiem krótka. Wszystko, co mam w tej funkcji do zrobienia, to utworzenie obiektu klasy `Game` oraz wywołanie metody `play()` tego obiektu, aby uruchomić grę.

```
def main():
    astrocrash = Game()
    astrocrash.play()

# wystartuj!
main()
```

Podsumowanie

W tym rozdziale rozszerzyłeś swoją wiedzę o programowaniu multimedialnym na obszar dźwięku, muzyki i animacji. Dowiedziałeś się, jak ładować i odtwarzać pliki dźwiękowe i muzyczne oraz jak zatrzymywać ich odtwarzanie. Zobaczyłeś również, jak tworzy się animacje. Nauczyłeś się również techniki tworzenia dużych programów poprzez pisanie coraz bardziej kompletnych, roboczych wersji finalnego produktu. Zobaczyłeś, jak można każdorazowo stawiać sobie jeden nowy cel do realizacji, budując w ten sposób swoją drogę do pełnego programu. Na koniec zobaczyłeś, jak wszystkie te nowe informacje i techniki zostały wykorzystane przy tworzeniu rozgrywanej w szybkim tempie gry akcji z efektami dźwiękowymi, animacją i własną muzyką.

Sprawdź swoje umiejętności

1. Ulepsz grę Astrocrash poprzez utworzenie nowego rodzaju śmiertelnie niebezpiecznego gruzu kosmicznego. Nadaj temu nowemu typowi kosmicznych śmieci pewną cechę, która odróżni je od asteroid. Na przykład zniszczenie takiego obiektu mogłoby wymagać dwóch uderzeń pocisku.
 2. Napisz wersję gry Simon Says, w której gracz ma za zadanie powtarzanie coraz bardziej skomplikowanych, przypadkowych sekwencji kolorów i dźwięków przy użyciu klawiatury.
 3. Napisz własną wersję innej klasycznej gry wideo, takiej jak Space Invaders lub Pac-Man.
 4. Wykreuj swoje własne programistyczne wyzwanie, lecz co najważniejsze, nigdy nie rezygnuj z mobilizowania się do dalszej nauki.
-

Skorowidz

A

abstrakcja, 171
aktualizacja
 zmiennej, 79
algorytm, 93
anagram, 129
animacja, 361, 368, 370
argument, 23
 nazwany, 176, 178
 pozycyjny, 178
 wyjątku, 218
ASCII-Art, 36
atrybut, 227, 232
atrybut klasy, 236, 238
 Animation, 415
 Message, 342
atrybuty prywatne, 241

B

blok kodu, 70
błąd, 23
błąd logiczny, 54, 55

C

ciało pętli, 78
cudzysłów, 32
 podwójny, 34
 pojedynczy, 34
 potrójny, 35

D

DBMS, Database
 Management System, 90
definiowanie
 funkcji, 170
 klasy, 229
 konstruktora, 301
 metody, 229
dekorator, 239
dodanie
 obiektu do ekranu, 335,
 338, 341
 pary klucz-wartość, 156
dokumentowanie funkcji,
 171
domyślne wartości
 parametrów, 177, 179
dostęp
 do atrybutów, 235, 238,
 245
 do atrybutów
 prywatnych, 242
 do elementów
 zagnieżdżonych, 146
 do elementu łańcucha,
 110
 do kodu źródłowego, 19
 do metod prywatnych,
 243
 do pliku binarnego, 210
 do pliku tekstowego, 203
 do wartości słownika,
 153

 do właściwości, 248
 do zagnieżdżonych
 krotek, 148
 do zamarynowanych
 obiektów, 213
 do znaku łańcucha, 109
 sekwencyjny, 107
 swobodny, 107
duszek, sprite, 332, 335
dziedziczenie, 263, 265
dzielenie
 całkowite, 44
 zmiennoprzecinkowe, 44
dzwonek systemowy, 39
dźwięk, 361, 371

E

ekran, 341, 344
ekran graficzny, 327, 331
element, 153
elementy interfejsu GUI, 292
etykieta, 296, 298

F

falsz, 85
funkcja, 22
 __additional_cards(),
 285
 __init__(), 285, 354–357,
 379, 399
 __pass_time(), 250
 advance(), 400

funkcja

append(), 142
 ask_number(), 189
 ask_yes_no(), 189
 capitalize(), 53
 change_global(), 184
 check_catch(), 355
 check_drop(), 358
 close(), 208
 computer_move(), 192
 congrat_winner, 195
 count(), 144
 create_widgets(), 318, 319
 die(), 391–395, 403
 display(), 173
 display_board(), 190
 display_instruct(), 188
 dump(), 212
 eat(), 250
 end(), 402
 end_game(), 356
 get(), 155, 159
 human_move(), 192
 handle_caught(), 356
 index(), 144
 init(), 348
 input(), 49, 60
 insert(), 144
 instructions(), 170
 int(), 58
 is_pressed(), 364
 items(), 159
 keys(), 159
 legal_moves(), 190
 len(), 106, 125, 137
 load(), 212
 lower(), 53
 main(), 195, 222, 287,
 359, 405
 mainloop(), 322, 349,
 365
 new_board(), 189
 next_block(), 221, 223
 next_line(), 220

next_turn(), 195
 open_file, 220
 pieces(), 189
 play(), 251, 286
 pop(), 144
 print(), 22, 34, 35
 randint(), 66
 random.randrange(),
 111
 randrange(), 66
 range(), 102, 103
 read_global(), 184
 read(), 208
 readline(), 204, 208
 readlines(), 205, 208
 remove(), 142
 replace(), 53
 reverse(), 144
 shelve.open(), 212
 sort(), 143
 strip(), 53
 swapcase(), 53
 talk(), 250
 tell_story(), 321
 title(), 53
 update(), 345, 354,
 380–395, 404
 upper(), 52
 values(), 159
 welcome(), 221
 winner(), 191
 write(), 207
 writelines(), 207

funkcje

konwersji typów, 58
 modułu games, 417
 modułu pickle, 212

G

generowanie

błędu, 23
 liczb losowych, 64
 losowej pozycji, 130

gra

Astrocraash, 361, 376
 Blackjack, 255, 277
 Jaka to liczba, 63, 95
 Kółko i krzyżyk, 167,
 185
 Mad Lib, 289, 323
 Pizza Panic, 323, 353
 Szubienica, 133, 159
 Turniej wiedzy, 199,
 218
 Wymieszane litery, 99,
 128
 graficzny interfejs
 użytkownika, GUI, 15,
 289, 300
 granice ekranu, 344
 grupowanie obiektów, 262
 GUI, Graphical User
 Interface, 16, 289, 300
 GUI toolkit, 291

H

hermetyzacja, 174
 hermetyzacja obiektów, 240

I

IDLE, 20
 import modułu, 65, 96, 276
 indeksowanie
 krotek, 126
 list, 137
 łańcuchów, 107
 inicjalizacja
 atrybutów, 234
 ekranu graficznego, 327
 zmiennych, 163
 instalacja w systemie
 Windows, 19
 instrukcja, 23
 break, 87
 continue, 87

if, 67, 68, 71
 przypisania, 46
 try, 215

J

języki
 obiektowe, 17
 wysokiego poziomu, 17

K

klasa, 227, 229
 Animation, 370, 415
 Application, 301
 Asteroid, 379, 392, 403
 BJ_Card, 281
 BJ_Dealer, 284
 BJ_Deck, 281
 BJ_Game, 284
 BJ_Hand, 282
 BJ_Player, 283
 Card, 259
 Chef, 357
 Collider, 394
 Critter, 249
 Explosion, 396
 Game, 399
 Hand, 261
 Keyboard, 417
 Label, 312
 Message, 340, 414
 Missile, 386, 390, 396
 Mouse, 416
 Music, 417
 Pan, 354
 Pizza, 355
 Screen, 410
 Ship, 381–396, 404
 Sprite, 333–336, 342, 367, 411
 Text, 338, 413
 Wrapper, 394

klasy
 bazowe, 265, 270
 modułu games, 327, 409
 pochodne, 266, 272
 programu Blackjack, 279

klauzula
 elif, 73, 75
 else, 71, 73, 218
 except, 215

klawisz, 419–422
 Enter, 61
 Tab, 38

klient funkcji, 240
 klucz, 154

kod, 23
 kod samodokumentujący, 48

kolizje, 350, 352, 390
 kombinacje obiektów, 259
 komentarz, 27, 59, 96
 komunikat, 256, 339

konfiguracja
 w innych systemach, 20
 w systemie Windows, 19

konfigurowanie programu, 188

konkatenacja
 krotek, 127
 list, 137
 łańcuchów, 41

konkretyzacja obiektu, 230
 konstruktor, 230, 232, 249

konwersja
 łańcuchów na liczby, 57
 wartości, 56

kończenie programu, 28, 97
 krotka, 99, 120
 funkcja len(), 125
 indeksowanie, 126
 jako warunek, 122
 konkatenacja, 127
 niemutowalność, 127

operator in, 125
 wycinanie, 126
 wypisywanie, 123
 zawierająca elementy, 122

kryptografia, 68

L

liczby
 całkowite, 44, 57
 losowe, 64
 pseudolosowe, 64
 zmiennoprzecinkowe, 44

liczenie
 co pięć, 104
 do przodu, 103
 do tyłu, 105

lista, 133
 funkcja len(), 137
 indeksowanie, 137
 konkatenacja, 137
 mutowalność, 138
 operator in, 137
 przypisanie wartości
 elementowi, 138
 przypisanie wartości
 wycinkowi, 139
 usuwanie elementu, 139
 usuwanie wycinka, 139

lista mailingowa, 18
 literał, 23

Ł

ładowanie
 dźwięku, 372
 muzyki, 374
 obrazu, 330, 333
 łańcuch, 23, 28, 105–115
 łańcuch dokumentacyjny, 171

M

marynowanie, 209, 210
 menedżer układu Grid, 305, 306
 metoda, *Patrz* funkcja
 metody, 229
 instancji, 229, 230
 klasy bazowej, 271
 klasy Mouse, 416
 klasy Music, 417
 klasy Screen, 410
 klasy Sprite, 412, 413
 klasy Text, 414
 listy, 140, 144
 łańcucha, 52, 53
 obiektu dźwiękowego, 418
 obiektu pliku, 208
 obiektu screen, 328
 prywatne, 241, 243
 słownika, 159
 statyczne, 236, 239
 moduł, 274
 color, 338, 398, 422
 decimal, 45
 games, 327, 353, 418–422
 karty, 277
 math, 382
 pakietu livewires, 409
 pickle, 211
 random, 65, 96
 shelve, 212
 tkinter, 295, 297, 301, 318
 modyfikowanie
 metod, 269
 okna głównego, 296
 mutowalność, 111
 mutowalność list, 138
 muzyka, 371, 374
 mysza, 348

N

nadklasa, superclass, 271
 nazwa zmiennej, 47
 niemutowalność
 krotek, 127
 łańcuchów, 111
 notacja z kropką, 66
 numery pozycji
 dodatnie, 109
 ujemne, 110

O

obiekt, 230
 klasy Card, 261
 klasy Ship, 382
 klasy Sprite, 342
 klasy Text, 338
 modułu games, 327
 mouse, 350
 screen, 328, 341
 obiekty
 graficzne, 331
 programowe, 225
 obliczanie
 liczby sekund, 61
 wagi, 61
 obsługa
 danych wejściowych, 347
 kolizji, 352, 390
 wyjątków, 214–217
 zdarzeń, 292, 303
 odbieranie komunikatów, 256
 odczyt klawiatury, 363
 odczytywanie
 danych, 200
 wartości zmiennej, 183
 z pliku, 203, 211
 z wiersza, 204
 zawartości pliku, 205

odmarynowanie, 211
 odtwarzanie
 dźwięku, 373, 374
 muzyki, 375
 okno
 główne, root window, 293
 graficzne, 325
 konsoli, 16
 Python Shell, 21
 OOP, object-oriented programming, 17, 225
 opcja Edit with IDLE, 27
 operacje na liczbach, 42
 operator
 in, 107, 125, 137, 154
 logiczny and, 91
 logiczny not, 90
 logiczny or, 92
 moduło, 44
 operatory
 matematyczne, 44
 porównania, 69, 70
 operatory
 rozszerzonego
 przypisania, 59
 sekwencji, 105
 otrzymywanie
 informacji, 173
 wartości, 175
 otwieranie pliku, 202

P

pakiet
 livewires, 325, 347, 384, 398, 407
 pygame, 325
 para klucz-wartość, 156–158
 parametr, 173
 lifetime, 415
 pozycyjny, 177
 self, 230

- pętla
 for, 100–102
 nieskończona, 79
 nieskończona umyślna, 86
 while, 76, 78, 129
 zdarzeń, 292, 296, 298, 300
- piksel, 327
- plik
 analizator_komunikatow.
 ↪py, 105
 astrocrash01.py, 378
 astrocrash06.py, 390
 dlugowiecznosc.py, 306
 dostep_swobodny.py, 108
 duszek_pizzy.py, 332
 dzwiek_i_muzyka.py, 372
 eksplozja.py, 369
 fundusz_powierniczzy_zly.py, 54
 globalny_zasieg.py, 182
 glupie_lancuchy.py, 41
 gra_w_karty.py, 259
 gry.py, 275
 haslo.py, 67
 instrukcja.py, 170
 inwentarz_bohatera.py, 121
 inwentarz_bohatera2.py, 124
 inwentarz_bohatera3.py, 135
 karty.py, 277
 komputer_nastrojow.py, 74
 koniec_gry2.py, 33
 krajacz_pizzy.py, 116
 kwiz.txt, 219
 leniwe_przyciski2.py, 301
 licznik.py, 102
 licznik_klikniec.py, 303
 maitre_d.py, 84
 manipulacje_cytatami.py, 51
 metkownica.py, 297
 najlepsze_wyniki.py, 141
 najlepsze_wyniki2.py, 147
 nieuchwytna_pizza.py, 350
 nowe_okno_graficzne.py, 326
 obraz_tla.py, 329
 obroc_duszka.py, 366
 odczytaj_klawisz.py, 363
 odczytaj_to.txt, 200
 opiekun_zwierzaka.py, 249
 osobisty_pozdrawiacz.py, 48
 patelnia_w_ruchu.py, 347
 pizza_panic.py, 353
 pizza_w_ruchu.py, 343
 pobierz_i_zwroc.py, 172
 pozdrawiacz.py, 45
 przegrana_bitwa-dobry.py, 82
 przegrana_bitwa-zly.py, 80
 py3e_software.zip, 407
 py3e_source.zip, 407
 rzut_koscmi.py, 65
 sprezysta_pizza.py, 345
 szubienica.py, 160
 translator_slangu.py, 153
 trzylatek.py, 77
 uczestnik_funduszu_powierniczego_dobry.py, 56
 udzielony_odmowiony.
 ↪py, 71, 88
 wybor_filmow.py, 311
 wybor_filmow2.py, 315
 wybredny_licznik.py, 86
 plik_wygrales.py, 340
 wymieszane_litery.py, 128
 wysoki_wynik.py, 337
 zabawne_podziekowania.
 ↪py, 37
 zadania_tekstowe.py, 43
 zwariowany_lancuch.py, 101
 zwierzak_z_klasa.py, 237
 zwierzak_z_konstruktozem.py, 231
 zyczenia_urodzinowe.py, 177
- pliki
 dźwiękowe, 361
 muzyczne, 361
 tekstowe, 200, 209
 z obrazami, 370
- pobieranie
 danych, 48
 wartości, 156
- podświetlanie składni, 24
- poła wyboru, 310, 312, 314
- polimorfizm, 273
- ponowne wykorzystanie kodu, 176
- powielanie łańcuchów, 40, 42
- powrót do początku pętli, 87
- poziomy gry, 397
- półka, shelf, 212
- prawda, 85
- prędkość, 343, 384
- procedura obsługi zdarzeń, 292, 303
- program
 Analizator komunikatów, 105
 Astrocrash01, 377

- program
 - Astrocraash02, 381
 - Astrocraash03, 382
 - Astrocraash04, 385
 - Astrocraash05, 388
 - Astrocraash06, 390
 - Astrocraash07, 393
 - Astrocraash08, 398
 - Długowieczność, 305
 - Dostęp swobodny, 108
 - Duszek pizzy, 332
 - Dźwięk i muzyka, 371
 - Ekskluzywna sieć, 88
 - Eksplozja, 368
 - Globalny zasięg, 182
 - Głupie łańcuchy, 40
 - Gra w karty, 259
 - Gra w karty 2.0, 264
 - Gra w karty 3.0, 269
 - Hasło, 67
 - Instrukcja, 169
 - Inwentarz bohatera, 120
 - Inwentarz bohatera 2.0, 124
 - Inwentarz bohatera 3.0, 135
 - Komputer nastrojów, 73
 - Koniec gry, 15
 - Kółko i krzyżyk, 194
 - Krajacz pizzy, 116
 - Leniwe przyciski, 298
 - Leniwe przyciski 2, 300
 - Licznik, 102
 - Licznik kliknięć, 303
 - Mad Lib, 289, 318
 - Maitre D', 83
 - Manipulacje cytatami, 51
 - Metkownica, 296
 - Najlepsze wyniki, 140
 - Najlepsze wyniki 2.0, 145
 - Nieistotne fakty, 31
 - Nieuchwytna pizza, 350
 - Nowe okno graficzne, 326
 - Obraz tła, 329
 - Obróć duszka, 366
 - Obsłuż to, 214
 - Odczytaj klawisz, 363
 - Odczytaj to, 200
 - Opiekun zwierzaka, 225, 249
 - Osobisty pozdrawiacz, 48
 - Patelnia w ruchu, 347
 - Pizza w ruchu, 342
 - Pobierz i zwróć, 172
 - Pogromca Obcych, 257
 - Pozdrawiacz, 45
 - Prosta gra, 274
 - Prosty interfejs GUI, 293
 - Prywatny zwierzak, 241
 - Przegrana bitwa, 80
 - Rzut kośćmi, 64
 - Sprężysta pizza, 344
 - Symulator trzylatka, 77
 - Translator slangu komputerowego, 152
 - Turniej wiedzy, 199
 - Uczestnik funduszu powierniczego, 54
 - Udzielony-odmówiony, 71
 - Wybór filmów, 311
- program
 - Wybór filmów 2, 315
 - Wybredny licznik, 86
 - Wygrałeś, 339
 - Wymieszane litery, 99
 - Wysoki wynik, 336
 - Zabawne podziękowania, 37
 - Zadania tekstowe, 42
 - Zamarynuj to, 209
 - Zapisz to, 206
 - Zwariowany łańcuch, 100
 - Zwierzak z atrybutem, 233
 - Zwierzak z klasą, 237
 - Zwierzak z konstruktorem, 231
 - Zwierzak z właściwością, 245
 - Życzenia urodzinowe, 177
- programowanie
 - obiektowe, OOP, 17, 225
 - sterowane zdarzeniami, 292
 - w trybie interaktywnym, 21
 - w trybie skryptowym, 24
- projektowanie
 - klas, 279
 - programów, 93
- prywatność obiektu, 244
- przechwytywanie sygnałów wejściowych, 349
- przeciążanie operatora, 56
- przekazywanie wartości, 23
- przemieszczanie duszków, 342
- przesłanianie metod, 269, 345
- przesłonięcie zmiennej globalnej, 184
- przesuwanie kursora, 38
- przesyłanie komunikatu, 258
- przycisk, 298, 299
- przyciski opcji, 315–317
- pseudokod, 93, 95, 186, 280
- punkt tabulacji, 38
- puste krotki, 122
- pusty wiersz, 28

R

radian, 384
 ramka, frame, 297, 369
 referencja do widżetu, 312
 referencje współdzielone,
 145, 151
 rejestracja wyników, 397
 reprezentowanie danych, 186
 rodzaje cudzysłowu, 34
 rozgałęzianie kodu, 63, 77
 rozpakowanie sekwencji,
 147
 rozszerzanie klasy, 263
 rozszerzanie klasy
 pochodnej, 266
 rozszerzenie
 .bat, 294
 .py, 25

S

sekwencja niemutowalna,
 111
 sekwencje
 specjalne, 36, 40
 zagnieżdżone, 145, 146
 składowe prędkości, 384
 słownik, 133, 152, 159
 sortowanie, 143
 specyfikacja typu wyjątku,
 215
 sprawdzanie
 klucza, 154
 wartości zmiennej, 79
 stała, 114, 160, 181
 stała klasowa, 383, 388, 392,
 403
 stałe modułu
 color, 422
 games, 418–422
 stan
 klawiszy, 364
 pola wyboru, 314

struktura słownika, 159
 sygnał
 dzwonka systemowego,
 39
 wejściowy, 349
 system zarządzania bazą
 danych, 90

Ś

śledzenie programu, 81
 środowisko
 programowania, 20

T

tekst, 336
 testowanie stanu klawiszy,
 364
 Tkinter, 293
 tło, 330
 tryb
 interaktywny, 21, 26
 skryptowy, 24
 tryby dostępu do pliku, 203,
 210, 212
 tworzenie
 algorytmów, 93
 anagramu, 129
 animacji, 368
 atrybutów prywatnych,
 241
 atrybutu klasy, 238
 duszka, 335
 etykiety, 298
 funkcji, 169
 interfejsu GUI, 289,
 291, 300
 klasy, 229, 259
 klasy bazowej, 265, 270
 klasy pochodnej, 345
 kombinacji obiektów,
 259

komentarzy, 59, 96
 konstruktora, 232
 krotek, 120
 listy, 135
 łańcuchów, 32, 35, 52
 menu, 251
 metod, 229
 metod prywatnych, 243
 metod statycznych, 239
 modułów, 273, 274
 nowego łańcucha, 113,
 115, 130
 obiektu, 230, 302
 okna głównego, 295
 okna graficznego, 325
 pętli, 96, 102, 163
 pół wyboru, 312
 procedury obsługi
 zdarzeń, 304
 przycisków, 299
 przycisków opcji, 316
 pustego łańcucha, 129
 pustej krotki, 122
 ramki, 297
 sekwencji
 zagnieżdżonych, 146
 skutku ubocznego, 193
 słowników, 153
 stałych, 114, 160
 warunków, 69
 widżetów, 302, 308
 wielu obiektów, 232
 właściwości, 246
 wycinków, 118, 137
 zmiennych, 46
 typ danych, 54
 typy
 liczbowe, 44
 sekwencji, 105
 wyjątków, 216

U

- układ współrzędnych, 331
- umieszczenie widżetu, 306
- umieszczanie na półce, 212
- UML, Unified Modeling Language, 258
- umyślne pętle
 - nieskończone, 86
- uruchamianie programu, 24
- ustawienie tła, 330
- usuwanie
 - elementu listy, 139
 - pary klucz-wartość, 158
 - wycinka listy, 139
- używanie
 - cudzysłówów, 32
 - domyślnych wartości parametrów, 179
 - etykiet, 296
 - klas pochodnych, 272
 - klucza, 154
 - komentarza, 27
- używanie
 - konstruktorów, 230
 - krotek, 123, 144
 - list, 144
 - metod łańcucha, 50
 - parametrów zwrotnych, 172
 - przycisków, 298
 - sekwencji specjalnych, 36
 - sekwencji
 - zagnieżdżonych, 145
 - słowników, 152
 - stałych, 185
 - widżetów, 305
 - zmiennych globalnych, 181, 185
 - znaku kontynuacji wiersza, 42

W

- wartości
 - mutowalne, 151
 - początkowe, 96
 - zwrotne, 23, 172, 174
- wartość
 - False, 82, 85
 - None, 118
 - True, 82, 85
- wartownik, 79
- warunek, 69
 - prosty, 88
 - złożony, 88
- wcięcie, 70
- wczytywanie wierszy, 205
- wiązanie widżetów, 303
- widoczność wskaźnika myszy, 349
- widżet, 296
 - Button, 298, 299
 - Entry, 305, 308
 - Frame, 297
 - Label, 298
 - Text, 305, 308
- właściwości
 - klasy Mouse, 416
 - klasy Screen, 410
 - klasy Sprite, 336, 411
 - klasy Text, 339, 414
 - obiektu mouse, 350
 - obiektu screen, 328
- właściwość, 245, 246
 - angle, 367
 - bottom, 346
 - mood, 250
 - still_playing, 285
- wprowadzanie danych, 60
- współrzędne
 - myszy, 348
 - punktu, 331

- wstawianie
 - tekstu, 309
 - znaku cudzysłowu, 38
 - znaku nowego wiersza, 38
- wycinanie
 - krotek, 126
 - łańcuchów, 116
- wycinek, 118
- wycinki list, 137
- wyjątek, 214
- wyjątek AttributeError, 242
- wyjście z pętli, 87
- wykrywanie kolizji, 350, 352
- wypisywanie
 - krotki, 123
 - wartości, 34
 - znaku lewego ukośnika, 38
- wyrażenie, 44
- wysyłanie komunikatów, 256
- wyświetlanie
 - duszka, 332
 - komunikatu, 339
 - menu, 141
 - obiektu, 236
 - tekstu, 336
 - wartości zmiennej, 60
 - wyników, 142, 148
- wywołanie funkcji, 171
- wywoływanie metody, 230
 - klasy bazowej, 271
 - statycznej, 240

Z

- zagnieżdżanie wywołań funkcji, 58
- zakres, 181
- zamykanie pliku, 202

- zapisywanie
 - do pliku, 206, 210
 - łańcuchów, 207
 - programu, 24
- zdarzenia, 292
- złożone struktury danych,
 - 209
- zmiana
 - parametru
 - mutowalnego, 193
- wartości zmiennej
 - globalnej, 184
- zmienna, 45, 46
 - globalna, 181
 - lokalna, 181
 - typu Boolean, 313
- znak
 - cudzysłowu, 38
 - kontynuacji wiersza, 42
 - kratki, 27
- lewego ukośnika, 38
- nowego wiersza, 38
- tabulacji, 38
- zachęty, 21
- Zunifikowany Język Modelowania, UML, 258
- zwracanie
 - informacji, 174
 - wartości, 175

PROGRAM PARTNERSKI

— GRUPY HELION —



1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

<http://program-partnerski.helion.pl>

GRUPA
Helion 

PYTHON DLA KAŻDEGO

Chcesz się nauczyć programować? Świetna decyzja! Wybierz język obiektowy, łatwy w użyciu, z przejrzystą składnią. Python będzie wprost doskonały! Rozwijany od ponad 20 lat, jest dojrzałym językiem, pozwalającym tworzyć zaawansowane aplikacje dla różnych systemów operacyjnych. Ponadto posiada system automatycznego zarządzania pamięcią, który zdejmuje z programisty obowiązek panowania nad tym skomplikowanym obszarem.

Jeżeli zdecydowałeś się na naukę programowania w tym języku, to książka, którą trzymasz w rękach, będzie strzałem w dziesiątkę! Siegnij po nią i przekonaj się, jak skonfigurować swoje środowisko pracy i rozpocząć przygodę z Pythonem. Z kolejnych rozdziałów dowiesz się, co to są typy proste, zmienne, instrukcje warunkowe, pętle oraz listy. Ponadto nauczysz się tworzyć listy i funkcje oraz obsługiwać wyjątki i korzystać z plików. Gdy już zbudujesz fundamenty wiedzy na temat języka Python, przejdziesz do bardziej zaawansowanych zagadnień. Programowanie obiektowe, tworzenie grafiki oraz graficznego interfejsu użytkownika, tworzenie animacji i efektów dźwiękowych to tylko niektóre z poruszanych tematów. Książka ta jest obowiązkową lekturą dla wszystkich osób, które zamierzają opanować język Python!

Dzięki tej książce:

- poznasz elementy i składnię języka Python
- zaznajomisz się z typowymi konstrukcjami języka
- nauczysz się programować obiektowo
- stworzysz animację i efekty dźwiękowe
- pokochasz język Python

Twój przewodnik po języku Python!

	KOD KORZYŚCI Siegnij po więcej! ▶	
 helion.pl	ISBN 978-83-8322-774-0	
 HELION SA ul. Kościuszki 1c 44-100 Gliwice tel.: 32 230 98 63 helion@helion.pl	 9 788383 227740	
Cena: 99,00 zł		