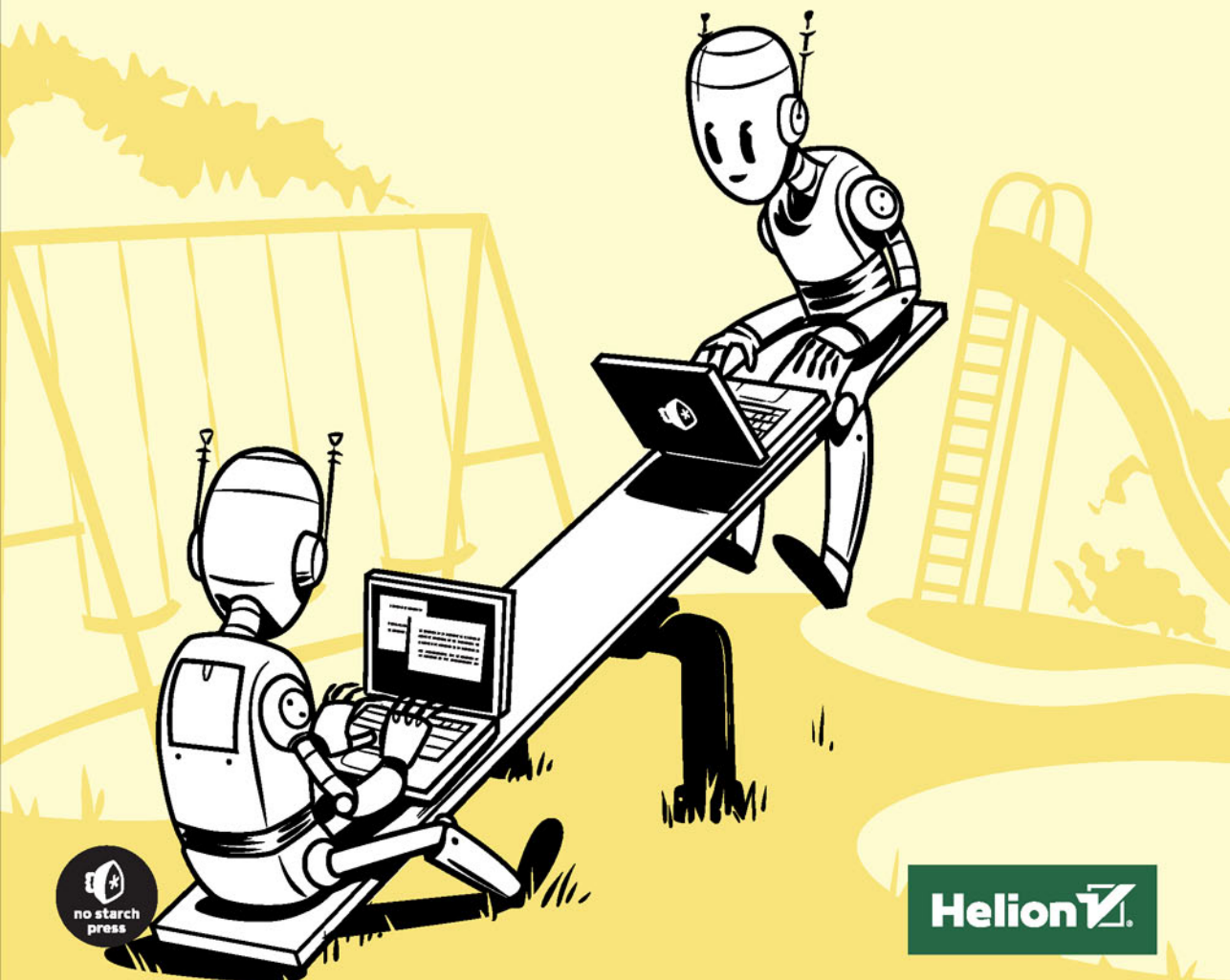


PYTHON

14 TWÓRCZYCH PROJEKTÓW
DLA DOCIEKLIWYCH PROGRAMISTÓW

MAHESH VENKITACHALAM



Helion

Tytuł oryginału: Python Playground: Geeky Projects for the Curious Programmer

Tłumaczenie: Lech Lachowski

ISBN: 978-83-283-2597-5

Copyright © 2016 by Mahesh Venkitachalam

Title of English-language original: Python Playground, ISBN 978-1-59327-604-1, published by No Starch Press.

Polish-language edition copyright © 2016 by Helion SA
All rights reserved.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

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 Wydawnictwo HELION 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 Wydawnictwo HELION nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

<http://helion.pl/user/opinie/python>

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

- [Kup książkę](#)
- [Poleć książkę](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

Spis treści

PODZIĘKOWANIA	15
WPROWADZENIE	17
Dla kogo jest ta książka?	17
Jaka jest zawartość książki?	18
Część I. Rozgrzewka	18
Część II. Symulacja życia	18
Część III. Zabawy z obrazami	18
Część IV. Grafika 3D	18
Część V. Projekty z wykorzystaniem sprzętu	19
Dlaczego Python?	19
Wersje Pythona	20
Kod zamieszczony w książce	20

Część I. Rozgrzewka

I

PARSOWANIE LIST ODTWARZANIA PROGRAMU ITUNES	25
Anatomia pliku listy odtwarzania iTunes	26
Wymagania	27
Kod	27
Znajdowanie duplikatów	28
Wyodrębnianie duplikatów	29
Wyszukiwanie utworów wspólnych dla wielu list odtwarzania	30
Gromadzenie danych statystycznych	31
Prezentowanie danych	31
Opcje wiersza poleceń	32
Kompletny kod	34
Uruchamianie programu	37
Podsumowanie	38
Eksperymenty!	38

2

SPIROGRAFY	39
Równania parametryczne	40
Równania spirografu	41
Żółwia grafika	44
Wymagania	45
Kod	45
Konstruktor Spiro	45
Funkcje konfiguracyjne	46
Metoda restart()	47
Metoda draw()	47
Tworzenie animacji	48
Klasa SpiroAnimator	48
Metoda genRandomParams()	49
Ponowne uruchamianie programu	50
Metoda update()	50
Wyświetlanie lub ukrywanie kursora	51
Zapisywanie krzywych	51
Parsowanie argumentów wiersza poleceń i inicjowanie	52
Kompletny kod	54
Uruchomienie animacji spirografu	58
Podsumowanie	60
Eksperymenty!	60

Część II. Symulacja życia

3

GRA W ŻYCIE	63
Jak to działa?	64
Wymagania	66
Kod	66
Reprezentacja planszy	66
Warunki początkowe	67
Warunki brzegowe	68
Implementacja reguł	69
Wysyłanie do programu argumentów wiersza poleceń	70
Inicjowanie symulacji	70
Kompletny kod	71
Uruchamianie symulacji „Gry w życie”	73
Podsumowanie	75
Eksperymenty!	75

4

GENEROWANIE TONÓW HARMONICZNYCH

ZA POMOCĄ ALGORYTMU KARPLUSA-STRONGA 77

Jak to działa?	79
Symulacja	79
Tworzenie plików WAV	81
Pentatonika molowa	82
Wymagania	83
Kod	83
Implementacja bufora pierścieniowego za pomocą klasy deque	83
Implementacja algorytmu Karplusa-Stronga	84
Zapisywanie pliku WAV	85
Odtwarzanie plików WAV za pomocą modułu pygame	85
Metoda main()	86
Kompletny kod	88
Uruchamianie symulacji szarpanej struny	91
Podsumowanie	92
Eksperymenty!	92

5

BOIDY: SYMULACJA STADA 93

Jak to działa?	94
Wymagania	94
Kod	95
Obliczanie pozycji i prędkości boidów	95
Ustawianie warunków brzegowych	96
Rysowanie boida	97
Zastosowanie reguł algorytmu stada	99
Dodawanie boida	101
Rozpraszanie boidów	102
Argumenty wiersza poleceń	102
Klasa Boids	103
Kompletny kod	104
Uruchamianie symulacji algorytmu stada	107
Podsumowanie	108
Eksperymenty!	108

Część III. Zabawy z obrazami

6

SZTUKA ASCII I I I

Jak to działa?	I I 2
Wymagania	I I 4

Kod	114
Definiowanie poziomów skali szarości oraz siatki	114
Obliczanie średniej jasności	115
Generowanie zawartości ASCII na podstawie obrazu	116
Opcje wiersza poleceń	117
Zapisywanie łańcuchów znaków obrazu ASCII w pliku tekstowym	117
Kompletny kod	118
Uruchamianie generatora sztuki ASCII	120
Podsumowanie	121
Eksperymenty!	121

7

FOTOMOZAIKI 123

Jak to działa?	124
Dzielenie obrazu docelowego	125
Uśrednianie wartości kolorów	125
Dopasowywanie obrazów	126
Wymagania	126
Kod	126
Wczytywanie obrazów kafelków	127
Obliczanie średniej wartości koloru obrazów wejściowych	128
Dzielenie obrazu docelowego na siatkę	128
Wyszukiwanie najlepszego dopasowania dla kafelka	129
Tworzenie siatki obrazu	130
Tworzenie fotomozaiki	131
Dodawanie opcji wiersza poleceń	132
Kontrolowanie rozmiaru fotomozaiki	133
Kompletny kod	133
Uruchamianie generatora fotomozaiki	138
Podsumowanie	138
Eksperymenty!	139

8

AUTOSTEREOGRAMY 141

Jak to działa?	142
Postrzeganie głębi w autostereogramie	142
Mapy głębi	143
Wymagania	145
Kod	145
Powtarzanie danego kafelka	145
Tworzenie kafelka z losowych kóteczek	146
Tworzenie autostereogramu	148
Opcje wiersza poleceń	149
Kompletny kod	150

Uruchamianie generatora autostereogramu	152
Podsumowanie	153
Eksperymenty!	154

Część IV. Grafika 3D

9

ZROZUMIEĆ OPENGL	157
Tradycyjny OpenGL	159
Nowoczesny OpenGL: potok grafiki 3D	160
Prymitywy geometryczne	161
Transformacje 3D	161
Shadery	163
Bufory wierzchołków	165
Mapowanie tekstury	166
Wyświetlanie OpenGL	167
Wymagania	167
Kod	167
Tworzenie okna OpenGL	167
Ustawianie wywołań zwrotnych	168
Klasa Scene	171
Kompletny kod	176
Uruchamianie aplikacji OpenGL	181
Podsumowanie	182
Eksperymenty!	182

10

SYSTEMY CZĄSTECZEK	185
Jak to działa?	187
Modelowanie ruchu cząsteczki	188
Ustawianie maksymalnej rozpiętości	188
Renderowanie cząstek	189
Użycie blendowania OpenGL do tworzenia bardziej realistycznych iskier	190
Korzystanie z billboardingu	191
Animowanie iskier	192
Wymagania	192
Kod dla systemu cząsteczek	193
Definiowanie geometrii cząsteczek	193
Definiowanie tablicy opóźnień czasowych dla cząsteczek	194
Ustawianie początkowych prędkości cząsteczek	194
Tworzenie cieniowania wierzchołkowego	195
Tworzenie cieniowania pikseli	197
Renderowanie	198
Klasa Camera	200

Kompletny kod systemu cząsteczek	201
Kod pudełka	207
Kod dla programu głównego	209
Aktualizacja cząsteczek w każdym kroku czasowym	210
Procedura obsługi klawiatury	211
Zarządzanie główną pętlą programu	211
Kompletny główny kod programu	212
Uruchamianie programu	215
Podsumowanie	215
Eksperymenty!	216

II

RENDERING OBJĘTOŚCIOWY	217
Jak to działa?	218
Format danych	219
Generowanie promieni	219
Wyświetlanie okna OpenGL	222
Wymagania	222
Przegląd kodu projektu	223
Generowanie tekstury 3D	223
Kompletny kod tekstury 3D	225
Generowanie promieni	226
Definiowanie geometrii sześcianu kolorów	227
Tworzenie obiektu bufora ramek	230
Renderowanie tylnych ścian sześcianu	231
Renderowanie przednich ścian sześcianu	231
Renderowanie całego sześcianu	232
Procedura obsługi zmiany rozmiaru okna	233
Kompletny kod generowania promieni	233
Volume ray casting	238
Cieniowanie wierzchołkowe	240
Cieniowanie pikseli	241
Kompletny kod volume ray casting	243
Tworzenie wycinków 2D	246
Cieniowanie wierzchołkowe	248
Cieniowanie pikseli	249
Interfejs użytkownika dla tworzenia wycinków 2D	249
Kompletny kod tworzenia wycinków 2D	250
Zebranie kodu w całość	253
Kompletny kod pliku głównego	255
Uruchamianie programu	257
Podsumowanie	257
Eksperymenty!	258

Część V. Projekty z wykorzystaniem sprzętu

I2

WPROWADZENIE DO ARDUINO	263
Arduino	264
Ekosystem Arduino	266
Język	266
IDE	266
Społeczność	266
Peryferia	266
Wymagania	267
Budowa obwodu czujnika natężenia światła	267
Jak działa obwód	268
Szkic Arduino	269
Tworzenie wykresu w czasie rzeczywistym	270
Kod Pythona	270
Kompletny kod Pythona	273
Uruchamianie programu	275
Podsumowanie	276
Eksperymenty!	277

I3

LASEROWY WYŚWIETLACZ AUDIO	279
Generowanie wzorów za pomocą lasera	280
Sterowanie silniczkiem	281
Szybka transformata Fouriera	282
Wymagania	284
Konstruowanie wyświetlacza laserowego	285
Podłączanie sterownika silniczków	287
Szkic Arduino	288
Konfigurowanie cyfrowych pinów wyjścia Arduino	288
Pętla główna	289
Zatrzymywanie silniczków	291
Kod Pythona	292
Wybór urządzenia audio	292
Odczyt danych z urządzenia wejściowego	293
Obliczanie FFT strumienia danych	294
Uzyskiwanie informacji o częstotliwości z wartości FFT	294
Konwersja częstotliwości na prędkości i kierunki obracania się silniczków	295
Testowanie ustawień silniczków	296
Opcje wiersza poleceń	297
Ręczne testowanie	298
Kompletny kod Pythona	298
Uruchamianie programu	302

Podsumowanie	302
Eksperymenty!	303

I4

MONITOR POGODY OPARTY NA RASPBERRY PI 305

Sprzęt	306
Czujnik temperatury i wilgotności DHT11	306
Raspberry Pi	307
Konfigurowanie Raspberry Pi	308
Instalacja i konfiguracja oprogramowania	309
System operacyjny	309
Wstępna konfiguracja	309
Konfiguracja Wi-Fi	309
Konfigurowanie środowiska programistycznego	310
Podłączenie poprzez SSH	311
Framework WWW Bottle	312
Tworzenie wykresów za pomocą biblioteki flot	312
Wyłączanie Raspberry Pi	315
Budowanie sprzętu	315
Kod	317
Obsługa żądań danych z czujnika	317
Tworzenie wykresu danych	318
Metoda update()	321
Procedura obsługi JavaScript dla diody LED	322
Dodawanie interaktywności	322
Kompletny kod	323
Uruchamianie programu	327
Podsumowanie	328
Eksperymenty!	328

Dodatki

A

INSTALACJA OPROGRAMOWANIA 331

Instalacja kodu źródłowego dla projektów z książki	331
Instalacja w systemie Windows	332
Instalacja biblioteki GLFW	332
Instalacja prekompilowanych plików binarnych dla każdego modułu	332
Inne opcje	333
Instalacja w systemie OS X	334
Instalacja Xcode i MacPorts	334
Instalacja modułów	334
Instalacja w systemie Linux	335

B

PRAKTYCZNE PODSTAWY ELEKTRONIKI	337
Typowe komponenty	338
Podstawowe narzędzia	340
Budowanie obwodów	342
Idąc dalej	344

C

RASPBERRY PI: PORADY I WSKAZÓWKI	347
Konfigurowanie Wi-Fi	347
Sprawdzanie połączenia Raspberry Pi z siecią lokalną	348
Zapobieganie wprowadzaniu adaptera Wi-Fi w tryb uśpienia	348
Tworzenie kopii zapasowej kodu i danych z Raspberry Pi	349
Tworzenie kopii zapasowej całego systemu operacyjnego Raspberry Pi	350
Logowanie do Raspberry Pi poprzez SSH	350
Korzystanie z kamery Raspberry Pi	352
Włączanie obsługi dźwięku na Raspberry Pi	352
Zmuszenie Raspberry Pi do mówienia	352
Włączanie obsługi HDMI	353
Mobilny Raspberry Pi	353
Sprawdzanie wersji sprzętowej Raspberry Pi	354

SKOROWIDZ	355
------------------------	------------

7

Fotomozaiki

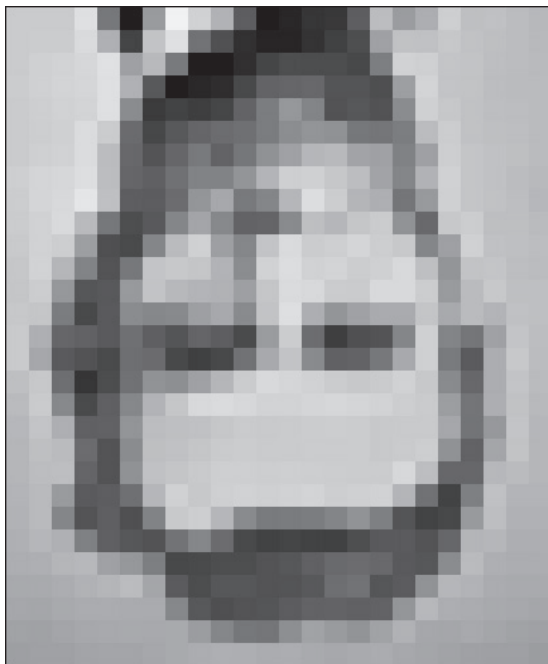


KIEDY BYŁEM W SZÓSTEJ KLASIE, ZOBACZYŁEM PEWIEN OBRAZEK, TAKI JAK TEN POKAZANY NA RYSUNKU 7.1, ALE NIE POTRAFIŁEM SIĘ ZORIENTOWAĆ, CO NA NIM JEST. PRZYPATRUJĄC MU SIĘ PRZEZ CHWILĘ, W KOŃCU ZROZUMIAŁEM. (Obróć książkę do góry nogami i przyjrzyj się obrazkowi z drugiego końca pokoju. Nikomu nie powiem).

Fotomozaika to obraz podzielony na siatkę prostokątów, z których każdy został zastąpiony przez inny obrazek, pasujący do obrazu **docelowego** (obrazu, który finalnie ma się pojawić na fotomozaice). Innymi słowy, jeśli spojrzysz na fotomozaikę z daleka, zobaczysz obraz docelowy, ale jeżeli podejdziesz bliżej, zobaczysz, że ten obraz w rzeczywistości składa się z wielu mniejszych obrazków.

Podstawą działania układanki jest sposób funkcjonowania ludzkiego oka. Cechujący się niską rozdzielczością, pikselowy obraz pokazany na rysunku 7.1 jest trudny do rozpoznania z bliska, ale gdy spojrzysz na niego z daleka, będziesz wiedzieć, co ukazuje, ponieważ będziesz dostrzegać mniej szczegółów, przez co krawędzie staną się gładkie. Fotomozaika działa na tej samej zasadzie. Z daleka obraz wygląda normalnie, ale z bliska odkrywa swoją tajemnicę — każdy „blok” jest unikatowym obrazkiem!

W tym projekcie dowiesz się, jak tworzyć fotomozaiki przy użyciu Pythona. Podzielnisz obraz docelowy na siatkę mniejszych obrazków i zastąpisz każdy blok w siatce odpowiednim obrazkiem, aby utworzyć fotomozaikę oryginalnego obrazu.



Rysunek 7.1. Zastanawiający obrazek

Będziesz mógł określić wymiary siatki i zdecydować, czy obrazy wejściowe mogą być ponownie wykorzystywane w mozaice.

W tym projekcie nauczysz się wykonywać takie czynności jak:

- tworzenie obrazów za pomocą biblioteki PIL (ang. *Python Imaging Library*);
- obliczanie średniej wartości RGB obrazu;
- przycinanie obrazów;
- zastępowanie części obrazu poprzez wklejanie innego obrazu;
- porównywanie wartości RGB za pomocą pomiaru średniej odległości.

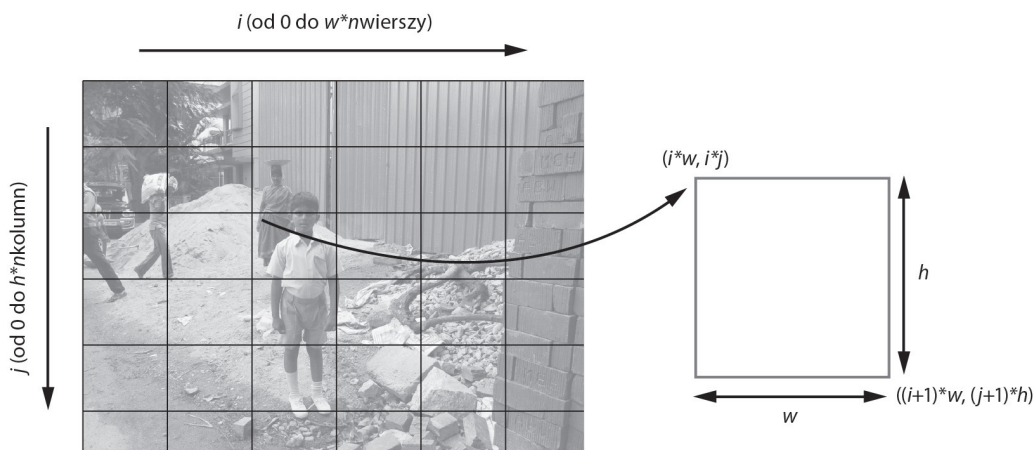
Jak to działa?

Aby utworzyć fotomozaikę, rozpoczniemy od pikselowej, mającej niską rozdzielczość wersji obrazu docelowego (ponieważ liczba obrazów kafelkowych byłaby zbyt duża w przypadku obrazu o wysokiej rozdzielczości). Rozdzielczość tego obrazu będzie określać wymiary $M \times N$ mozaiki, gdzie M jest liczbą wierszy, a N liczbą kolumn. Następnie zastąpimy każdy kafelek w oryginalnym obrazie według następującej metodologii:

1. Odczytanie obrazów kafelkowych, które zastąpią kafelki w oryginalnym obrazie.
2. Odczytanie obrazu docelowego i podzielenie go na siatkę $M \times N$ kafelków.
3. Znalezienie dla każdego kafelka najlepszych odpowiedników z obrazów wejściowych.
4. Utworzenie ostatecznej mozaiki poprzez rozmieszczenie wybranych obrazów wejściowych na siatce $M \times N$.

Dzielenie obrazu docelowego

Rozpocniemy od podzielenia obrazu docelowego na siatkę $M \times N$ kafelków zgodnie ze schematem pokazanym na rysunku 7.2.



Rysunek 7.2. Dzielenie obrazu docelowego

Obraz na rysunku 7.2 pokazuje, jak można podzielić oryginalny obraz na siatkę kafelków. Oś x reprezentuje kolumny siatki, a oś y — wiersze siatki.

Teraz przyjrzyjmy się, jak obliczyć współrzędne dla pojedynczego kafelka z tej siatki. Dla kafelka z indeksem (i, j) współrzędnymi lewego górnego rogu są $(i*w, i*j)$, a współrzędnymi prawego dolnego rogu są $((i+1)*w, (j+1)*h)$, gdzie w oraz h oznaczają odpowiednio szerokość i wysokość kafelka. Te współrzędne mogą być używane przez bibliotekę PIL do przycinania i tworzenia kafelków z tego obrazu.

Uśrednianie wartości kolorów

Każdy piksel w obrazie ma kolor, który może być reprezentowany przez wartości jego barw składowych: czerwonej, zielonej i niebieskiej. W tym przypadku używamy 8-bitowych obrazów, więc każda z tych składowych ma 8-bitową wartość z przedziału $[0, 255]$. Dla danego obrazu o całkowitej liczbie N pikseli średnia wartość RGB jest obliczana w następujący sposób:

$$(r, g, b)_{\text{śr}} = \left(\frac{r_1 + r_2 + \dots + r_N}{N}, \frac{g_1 + g_2 + \dots + g_N}{N}, \frac{b_1 + b_2 + \dots + b_N}{N} \right)$$

Należy zauważyć, że średnia wartość RGB jest również tripletem, a nie skłarem lub pojedynczą liczbą, ponieważ średnie są wyliczane osobno dla każdej składowej koloru. Obliczamy średnią wartość RGB, aby dopasować kafelki do docelowego obrazu.

Dopasowywanie obrazów

Dla każdego kafelka w obrazie docelowym musimy dopasować obraz z folderu wejściowego podanego przez użytkownika. Aby określić, czy jakieś dwa obrazy do siebie pasują, używamy średnich wartości RGB. Najbliższym dopasowaniem jest obraz z najbardziej zbliżoną średnią wartością RGB.

Najprostszym sposobem jest obliczenie odległości między wartościami RGB w pikselu w celu znalezienia najlepszego dopasowania wśród obrazów wejściowych. Możemy zastosować z geometrii następującą metodę obliczania odległości dla punktów 3D:

$$D_{1,2} = \sqrt{(r_1 - r_2)^2 + (g_1 - g_2)^2 + (b_1 - b_2)^2}$$

Obliczamy tutaj odległość między punktami (r_1, g_1, b_1) oraz (r_2, g_2, b_2) . Mając dane średnią wartość docelową RGB oraz listę średnich wartości RGB z obrazów wejściowych, można użyć przeszukiwania liniowego i obliczania odległości dla punktów 3D, aby znaleźć najbliższy pasujący obraz.

Wymagania

W tym projekcie użyjemy biblioteki Pillow do wczytywania obrazów, uzyskiwania dostępu do ich danych bazowych oraz tworzenia i modyfikowania obrazów. Wykorzystamy także numpy do manipulowania danymi obrazu.

Kod

Rozpoczniemy od wczytania obrazów kafelkowych, których będziemy używać do tworzenia fotomozaiki. Następnie obliczymy średnią wartość RGB obrazów, a potem podzielimy obraz docelowy na siatkę kafelków i poszukamy najlepszego dopasowania dla każdego z nich. Na koniec złożymy kafelki obrazów, żeby utworzyć ostateczną fotomozaikę. Aby zobaczyć pełny kod projektu, przejdź do podrozdziału „Kompletny kod”.

Wczytywanie obrazów kafelków

Najpierw wczytamy obrazy wejściowe z podanego folderu. Oto jak to zrobić:

```
def getImages(imageDir):  
    """  
    Dla danego folderu obrazów zwraca listę obiektów Image  
    """  
    files = os.listdir(imageDir) ❶  
    images = []  
    for file in files:  
        filePath = os.path.abspath(os.path.join(imageDir, file)) ❷  
        try:  
            # bezpośrednie ładowanie, abyśmy nie napotkali niedoboru zasobów  
            fp = open(filePath, "rb") ❸  
            im = Image.open(fp)  
            images.append(im)  
            # wymuszenie ładowania danych obrazu z pliku  
            im.load() ❹  
            # zamknięcie pliku  
            fp.close() ❺  
        except:  
            # pominięcie  
            print("Nieprawidłowy obraz: %s" % (filePath,))  
    return images
```

W punkcie ❶ używamy `os.listdir()`, aby zebrać w listę pliki z katalogu *imageDir*. Następnie iterujemy przez każdy plik na liście i ładujemy go do obiektu *Image* biblioteki PIL.

W punkcie ❷ używamy metod `os.path.abspath()` i `os.path.join()`, aby uzyskać kompletną nazwę pliku obrazu. Ten idiom jest powszechnie stosowany w Pythonie w celu zapewnienia, że kod będzie działał ze ścieżkami względnymi (na przykład `foo\bar`), ze ścieżkami bezwzględnymi (`c:\foo\bar`) oraz w różnych systemach operacyjnych z odmiennymi konwencjami nazewnictwa katalogów (`\` w Windowsie w porównaniu z `/` w Linuksie).

Aby załadować pliki do obiektów *Image* biblioteki PIL, moglibyśmy przekazać każdą nazwę pliku do metody `Image.open()`, ale jeśli folder fotomozaiki zawierałby setki lub nawet tysiące obrazów, takie postępowanie byłoby wysoce zasobochłonne. Zamiast tego możemy wykorzystać Pythona do otwarcia każdego obrazu kafelkowego i przekazania uchwytu pliku `fp` do biblioteki PIL przy użyciu `Image.open()`. Gdy obraz zostanie załadowany, zamykamy uchwyt pliku i zwalniamy zasoby systemowe.

W punkcie ❸ otwieramy plik obrazu za pomocą `open()`. W kolejnych liniach przekazujemy uchwyt do metody `Image.open()` i zapisujemy obraz wynikowy (`im`) w tablicy.

W punkcie ❹ wywołujemy `Image.load()`, aby wymusić załadowanie danych obrazu do `im`, ponieważ `open()` jest leniwą operacją. Identyfikuje ona obraz, ale w rzeczywistości nie odczytuje wszystkich danych obrazu, dopóki nie spróbujesz go użyć.

W punkcie 5 zamykamy uchwytu pliku, aby zwolnić zasoby systemowe.

Obliczanie średniej wartości koloru obrazów wejściowych

Po wczytaniu obrazów wejściowych trzeba obliczyć ich średnią wartość koloru oraz wartość dla każdego kafelka w obrazie docelowym. Aby obliczyć obie wartości, utworzymy metodę `getAverageRGB()`.

```
def getAverageRGB(image):
    """
    Dla każdego obrazu wejściowego zwraca średnią wartość koloru jako (r, g, b)
    """
    # uzyskanie każdego obrazu kafelkowego jako tablicy numpy
    im = np.array(image) ①
    # uzyskanie kształtu każdego obrazu wejściowego
    w,h,d = im.shape ②
    # uzyskanie średniej wartości RGB
    return tuple(np.average(im.reshape(w*h, d), axis=0)) ③
```

W punkcie 1 używamy numpy do przekonwertowania każdego obiektu `Image` na tablicę danych. Zwracana tablica numpy ma kształt (w, h, d) , gdzie w jest wagą obrazu, h wysokością, a d głębokością, co w tym przypadku daje trzy jednostki (po jednej dla R, G i B) dla obrazów RGB. W punkcie 2 zapisujemy krótkę `shape`, a następnie w punkcie 3 obliczamy średnią wartość RGB poprzez przekształcanie tej tablicy na wygodniejszą formę o kształcie $(w*h, d)$, aby można było obliczyć średnią za pomocą `numpy.average()`.

Dzielenie obrazu docelowego na siatkę

Teraz musimy podzielić obraz docelowy na siatkę $M \times N$ mniejszych obrazów. Utwórzmy w tym celu odpowiednią metodę.

```
def splitImage(image, size):
    """
    Dla danego obiektu Image i wymiarów (rows, cols) zwraca listę m*n obiektów Image
    """
    W, H = image.size[0], image.size[1] ①
    m, n = size ②
    w, h = int(W/n), int(H/m) ③
    # lista obrazów
    imgs = []
    # generowanie listy wymiarów
    for j in range(m):
        for i in range(n):
            # załączanie przyciętego obrazu
            imgs.append(image.crop((i*w, j*h, (i+1)*w, (j+1)*h))) ④
    return imgs
```

Najpierw zbieramy wymiary obrazu docelowego w punkcie ❶ i wielkość siatki w punkcie ❷. W punkcie ❸ obliczamy wymiary każdego kafelka w obrazie docelowym, stosując podstawowe dzielenie.

Teraz musimy wykonać iterację przez wymiary siatki oraz wyciąć i zapisać każdy kafelek jako oddzielny obraz. W punkcie ❹ metoda `image.crop()` przycina fragment obrazu, wykorzystując jako argumenty współrzędne lewego górnego rogu kafelka i wymiary przyciętego obrazu (tak jak omówiono to w punkcie „Dzielenie obrazu docelowego” we wcześniejszej części tego rozdziału).

Wyszukiwanie najlepszego dopasowania dla kafelka

Teraz poszukajmy w folderze obrazów wejściowych najlepszego dopasowania dla kafelka. W tym celu utworzymy metodę narzędziową `getBestMatchIndex()` w następujący sposób:

```
def getBestMatchIndex(input_avg, avgs):
    """
    Zwraca indeks najlepszego dopasowania obiektu Image na podstawie odległości wartości RGB
    """

    # średnia obrazu wejściowego
    avg = input_avg

    # uzyskanie najbliższej wartości RGB dla danych wejściowych na podstawie odległości wartości RGB
    index = 0
    min_index = 0 ❶
    min_dist = float("inf") ❷
    for val in avgs: ❸
        dist = ((val[0] - avg[0])*(val[0] - avg[0]) +
                (val[1] - avg[1])*(val[1] - avg[1]) +
                (val[2] - avg[2])*(val[2] - avg[2])) ❹
        if dist < min_dist: ❺
            min_dist = dist
            min_index = index
            index += 1

    return min_index
```

Próbujemy znaleźć najbardziej zbliżone dopasowanie do średniej wartości RGB (`input_avg`) z listy `avgs`. Jest to lista średnich wartości RGB z obrazów kafelkowych.

Aby znaleźć najlepsze dopasowanie, porównujemy średnie wartości RGB obrazów wejściowych. W punktach ❶ i ❷ inicjujemy indeks najbliższego dopasowania na 0 i minimalną odległość na nieskończoność. Za pierwszym razem ten test będzie zawsze dawał wynik pozytywny, ponieważ każda odległość jest mniejsza niż nieskończoność. W punkcie ❸ tworzymy pętlę przez wartości znajdujące się na liście średnich, a w punkcie ❹ rozpoczynamy obliczanie odległości za pomocą standardowego wzoru. (Porównujemy kwadraty odległości, aby zmniejszyć czas obliczeń). Jeśli obliczona odległość jest mniejsza niż przechowywana odległość

minimalna `min_dist`, jest w punkcie ❸ zastępowana nową minimalną odległością. Na koniec iteracji otrzymujemy najbliższy dla `input_avg` indeks średniej wartości RGB z listy `avgs`. Teraz możemy użyć tego indeksu, aby wybrać pasujący obrazek kafelkowy z listy tych obrazków.

Tworzenie siatki obrazu

Aby przejść do tworzenia fotomozajki, potrzebujemy jeszcze jednej metody narzędzowej. Metoda `createImageGrid()` tworzy siatkę obrazów, która ma rozmiar $M \times N$. Ta siatka obrazu jest finalnym obrazem fotomozajki, utworzonym z listy wybranych obrazów kafelkowych.

```
def createImageGrid(images, dims):
    """
    Dla danej listy obrazów i rozmiaru siatki (m, n) utworzenie siatki obrazów.
    """
    m, n = dims ❶

    # kontrola poprawności
    assert m*n == len(images)

    # uzyskanie maksymalnej wysokości i szerokości obrazów
    # nie zakładamy, że wszystkie są równe
    width = max([img.size[0] for img in images]) ❷
    height = max([img.size[1] for img in images])

    # tworzenie obrazu wyjściowego
    grid_img = Image.new('RGB', (n*width, m*height)) ❸

    # wklejanie obrazów kafelkowych do siatki
    for index in range(len(images)):
        row = int(index/n) ❹
        col = index - n*row ❺
        grid_img.paste(images[index], (col*width, row*height)) ❻

    return grid_img
```

W punkcie ❶ zbieramy wymiary siatki, a następnie używamy metody `assert`, aby sprawdzić, czy liczba obrazów dostarczonych do metody `createImageGrid()` odpowiada rozmiarowi siatki. (Metoda `assert` sprawdza założenia w kodzie, zwłaszcza w trakcie rozwoju i testów). Teraz mamy listę obrazów kafelkowych opartą na najbliższym dopasowaniu RGB, której użyjemy do tworzenia siatki obrazu reprezentującej fotomozaikę. Niektóre z wybranych obrazów mogą nie wypełniać kafelka całkowicie ze względu na różnice w ich rozmiarach, ale nie będzie to stanowiło problemu, ponieważ najpierw wypełnimy kafelek czarnym tłem.

W punkcie ❷ i w kolejnych liniach obliczamy maksymalną szerokość i wysokość obrazów kafelkowych. (Nie przyjęliśmy żadnych założeń dotyczących wielkości wybranych obrazów wejściowych. Kod będzie działał bez względu na to,

czy są one takie same, czy różne). Jeśli obrazy wejściowe nie będą całkowicie wypełniać kafelka, przestrzenie między kafelkami będą miały kolor tła, którym jest domyślnie czarny.

W punkcie ❸ tworzymy pusty obiekt Image o wymiarach odpowiednich, żeby pomieścić wszystkie obrazy z siatki. Wkleimy do niego obrazy kafelkowe. Następnie wypełniamy siatkę obrazu. W punkcie ❹ wykonujemy pętlę przez wybrane obrazy i wklejamy je do odpowiedniej siatki za pomocą metody Image.paste(). Pierwszym argumentem przekazywanym do Image.paste() jest obiekt Image, który ma być wklejony, a drugim są współrzędne lewego górnego rogu. Teraz trzeba ustalić, w którym wierszu i w której kolumnie wkleić obrazek kafelkowy do siatki obrazu. W tym celu trzeba wyrazić indeks obrazu w kategoriach wierszy i kolumn. Indeks kafelka w siatce obrazu jest określony poprzez $N \cdot \text{row} + \text{col}$, gdzie N jest liczbą komórek w wierszu, a (row, col) to współrzędne na tej siatce. W punkcie ❺ podajemy wiersz z poprzedniego wzoru, a w punkcie ❻ kolumnę.

Tworzenie fotomozaiki

Skoro mamy już wszystkie potrzebne narzędzia, napiszmy główną funkcję, która tworzy fotomozaikę.

```
def createPhotomosaic(target_image, input_images, grid_size, reuse_images=True):
    """
    Tworzenie fotomozaiki dla danych obrazu docelowego i obrazów wejściowych.
    """

    print('dzielenie obrazu docelowego...')
    # dzieli obraz docelowy na kafelki
    target_images = splitImage(target_image, grid_size) ❶

    print('wyszukiwanie dopasowań dla obrazu...')
    # dla każdego kafelka wybiera jeden pasujący obraz wejściowy
    output_images = []
    # dla informacji zwrotnej dla użytkownika
    count = 0
    batch_size = int(len(target_images)/10) ❷

    # obliczanie średnich obrazów wejściowych
    avgs = []
    for img in input_images:
        avgs.append(getAverageRGB(img)) ❸

    for img in target_images:
        # obliczanie średniej wartości RGB obrazu docelowego
        avg = getAverageRGB(img) ❹
        # wyszukiwanie indeksu dopasowania najbliższej wartości RGB z listy
        match_index = getBestMatchIndex(avg, avgs) ❺
        output_images.append(input_images[match_index]) ❻
        # informacja zwrotna dla użytkownika
        if count > 0 and batch_size > 10 and count % batch_size is 0: ❼
            print('przetworzono %d z %d...' % (count, len(target_images)))
            count += 1
```

```

#jeśli ustawiona jest flaga, usuwanie wybranego obrazu z danych wejściowych
if not reuse_images: ❸
    input_images.remove(match)

print('tworzenie mozaiki...')
#tworzenie obrazu fotomozaiki z kafelków
mosaic_image = createImageGrid(output_images, grid_size) ❹

#zwracanie mozaiki
return mosaic_image

```

Metoda `createPhotomosaic()` jako dane wejściowe przyjmuje obraz docelowy, listę obrazków wejściowych, rozmiar generowanej fotomozaiki oraz flagę, która wskazuje, czy obrazek może być ponownie wykorzystany. W punkcie ❶ dzieli obraz docelowy na siatkę. Gdy obraz jest podzielony, szukamy w obrazach z folderu wejściowego dopasowań dla każdego kafelka. (Ponieważ proces ten może trochę potrwać, dostarczamy użytkownikom informacji zwrotnej, aby wiedzieli, że program nadal działa).

W punkcie ❷ ustawiamy `batch_size` na jedną dziesiątą całkowitej liczby obrazów kafelkowych. Ta zmienna będzie wykorzystywana w kodzie w punkcie ❺ do aktualizacji informacji dla użytkownika. (Wybrana wartość jednej dziesiątej jest dowolna i stanowi po prostu sposób na to, by program poinformował użytkownika, że wciąż pracuje. Za każdym razem, gdy program przetwarza jedną dziesiątą obrazów, wyświetla komunikat wskazujący, że nadal działa).

W punkcie ❸ obliczamy średnią wartość RGB dla każdego obrazu z folderu wejściowego i zapisujemy tę wartość na liście `avgs`. Następnie rozpoczynamy iterację przez każdy kafelek w siatce obrazu docelowego. W punkcie ❹ obliczamy średnią wartość RGB dla każdego kafelka. Później w punkcie ❺ szukamy najbliższego dopasowania do tej wartości na liście średnich dla obrazów wejściowych. Wynik jest zwracany jako indeks, którego używamy w punkcie ❻ do pobrania obiektu `Image` i zapisania go na liście.

W punkcie ❷ dla każdej liczby `batch_size` przetworzonych obrazów wyświetlamy komunikat dla użytkownika. W punkcie ❸, jeśli flaga `reuse_images` jest ustawiona na `False`, usuwamy wybrany obraz wejściowy z listy, aby nie został ponownie wykorzystany w innym kafelku. (Najlepiej sprawdza się to wtedy, gdy masz wiele różnorodnych obrazów wejściowych do wyboru). Na koniec w punkcie ❹ łączymy obrazy, aby utworzyć finalną fotomozaikę.

Dodawanie opcji wiersza poleceń

Metoda `main()` programu obsługuje następujące opcje wiersza poleceń:

```

#parsowanie argumentów
parser = argparse.ArgumentParser(description='Tworzy fotomozaikę z obrazów
↳wejściowych')
#dodawanie argumentów
parser.add_argument('--target-image', dest='target_image', required=True)

```

```
parser.add_argument('--input-folder', dest='input_folder', required=True)
parser.add_argument('--grid-size', nargs=2, dest='grid_size', required=True)
parser.add_argument('--output-file', dest='outfile', required=False)
```

Ten kod zawiera trzy wymagane parametry wiersza poleceń: nazwę obrazu docelowego, nazwę wejściowego folderu obrazów oraz rozmiar siatki. Czwarty parametr dotyczy opcjonalnej nazwy pliku. Jeżeli nazwa pliku zostanie pominięta, fotomozaika zostanie zapisana w pliku o nazwie *mosaic.png*.

Kontrolowanie rozmiaru fotomozaiki

Ostatnią kwestią, jaką należy się zająć, jest rozmiar fotomozaiki. Jeśli bezrefleksyjnie sklejalibyśmy ze sobą obrazy wejściowe na podstawie pasujących kafelków w obrazie docelowym, moglibyśmy otrzymać ogromną fotomozaikę, która byłaby znacznie większa niż obraz docelowy. Aby tego uniknąć, zmieniamy rozmiary obrazów wejściowych, by pasowały do rozmiaru każdego kafelka w siatce. (Daje to dodatkową korzyść w postaci przyspieszenia obliczeń średniej RGB, ponieważ używamy mniejszych obrazów). Jest to obsługiwane również przez metodę `main()`:

```
print('zmiana rozmiaru obrazów...')
# dla danego rozmiaru siatki obliczanie maksymalnej szerokości i wysokości kafelków
dims = (int(target_image.size[0]/grid_size[1]),
        int(target_image.size[1]/grid_size[0])) ❶
print("maksymalne wymiary kafelka: %s" % (dims,))
# zmiana rozmiaru
for img in input_images:
    img.thumbnail(dims) ❷
```

W punkcie ❶ obliczamy wymiary docelowe na podstawie określonych rozmiarów siatki. Następnie w punkcie ❷ używamy metody `Image.thumbnail()` biblioteki PIL w celu zmiany wielkości obrazów, aby pasowały do tych wymiarów.

Kompletny kod

Kompletny kod dla tego projektu jest dostępny do pobrania na serwerze wydawnictwa Helion, pod adresem <ftp://ftp.helion.pl/przyklady/pythtp.zip>.

```
import sys, os, random, argparse
from PIL import Image
import imghdr
import numpy as np

def getAverageRGB(image):
    """
    Dla każdego obrazu wejściowego zwraca średnią wartość koloru jako (r, g, b)
    """
    # uzyskanie każdego obrazu kafelkowego jako tablicy numpy
```

```

im = np.array(image)
# uzyskanie kształtu każdego obrazu wejściowego
w,h,d = im.shape
# uzyskanie średniej wartości RGB
return tuple(np.average(im.reshape(w*h, d), axis=0))

def splitImage(image, size):
    """
    Dla danego obiektu Image i wymiarów (rows, cols) zwraca listę m*n obiektów Image
    """
    W, H = image.size[0], image.size[1]
    m, n = size
    w, h = int(W/n), int(H/m)
    # lista obrazów
    imgs = []
    # generowanie listy wymiarów
    for j in range(m):
        for i in range(n):
            # załączanie przyciętego obrazu
            imgs.append(image.crop((i*w, j*h, (i+1)*w, (j+1)*h)))
    return imgs

def getImages(imageDir):
    """
    Dla danego folderu obrazów zwraca listę obiektów Image
    """
    files = os.listdir(imageDir)
    images = []
    for file in files:
        filePath = os.path.abspath(os.path.join(imageDir, file))
        try:
            # bezpośrednie ładowanie, abyśmy nie napotkali niedoboru zasobów
            fp = open(filePath, "rb")
            im = Image.open(fp)
            images.append(im)
            # wymuszenie ładowania danych obrazu z pliku
            im.load()
            # zamknięcie pliku
            fp.close()
        except:
            # pominięcie
            print("Nieprawidłowy obraz: %s" % (filePath,))
    return images

def getImageFileNames(imageDir):
    """
    Dla danego folderu obrazów zwraca listę nazw plików obrazów
    """
    files = os.listdir(imageDir)
    filenames = []
    for file in files:
        filePath = os.path.abspath(os.path.join(imageDir, file))
        try:
            imgType = imghdr.what(filePath)
            if imgType:
                filenames.append(filePath)
        except:
            # pominięcie

```

```

        print("Nieprawidłowy obraz: %s" % (filePath,))
    return filenames

def getBestMatchIndex(input_avg, avgs):
    """
    Zwraca indeks najlepszego dopasowania obiektu Image na podstawie odległości wartości RGB
    """

    # średnia obrazu wejściowego
    avg = input_avg

    # uzyskanie najbliższej wartości RGB dla danych wejściowych na podstawie odległości wartości RGB
    index = 0
    min_index = 0
    min_dist = float("inf")
    for val in avgs:
        dist = ((val[0] - avg[0])*(val[0] - avg[0]) +
                (val[1] - avg[1])*(val[1] - avg[1]) +
                (val[2] - avg[2])*(val[2] - avg[2]))
        if dist < min_dist:
            min_dist = dist
            min_index = index
        index += 1

    return min_index

def createImageGrid(images, dims):
    """
    Dla danej listy obrazów i rozmiaru siatki (m, n) utworzenie siatki obrazów.
    """
    m, n = dims

    # kontrola poprawności
    assert m*n == len(images)

    # uzyskanie maksymalnej wysokości i szerokości obrazów
    # nie zakładamy, że wszystkie są równe
    width = max([img.size[0] for img in images])
    height = max([img.size[1] for img in images])

    # tworzenie obrazu wyjściowego
    grid_img = Image.new('RGB', (n*width, m*height))

    # wklejanie obrazów kafelkowych do siatki
    for index in range(len(images)):
        row = int(index/n)
        col = index - n*row
        grid_img.paste(images[index], (col*width, row*height))

    return grid_img

def createPhotomosaic(target_image, input_images, grid_size, reuse_images=True):
    """
    Tworzenie fotomosaiki dla danych obrazu docelowego i obrazów wejściowych.
    """

```

```

print('dzielenie obrazu docelowego...')
# dzieli obraz docelowy na kafelki
target_images = splitImage(target_image, grid_size)

print('wyszukiwanie dopasowań dla obrazu...')
# dla każdego kafelka wybiera jeden pasujący obraz wejściowy
output_images = []
# dla informacji zwrotnej dla użytkownika
count = 0
batch_size = int(len(target_images)/10)

# obliczanie średnich obrazów wejściowych
avgs = []
for img in input_images:
    avgs.append(getAverageRGB(img))

for img in target_images:
    # obliczanie średniej wartości RGB obrazu docelowego
    avg = getAverageRGB(img)
    # wyszukiwanie indeksu dopasowania najbliższej wartości RGB z listy
    match_index = getBestMatchIndex(avg, avgs)
    output_images.append(input_images[match_index])
    # informacja zwrotna dla użytkownika
    if count > 0 and batch_size > 10 and count % batch_size is 0:
        print('przetworzono %d z %d...' %(count, len(target_images)))
        count += 1
    # jeśli ustawiona jest flaga, usuwanie wybranego obrazu z danych wejściowych
    if not reuse_images:
        input_images.remove(match)

print('tworzenie mozaiki...')
# tworzenie obrazu fotomozaiki z kafelków
mosaic_image = createImageGrid(output_images, grid_size)

# zwracanie mozaiki
return mosaic_image

# Zebranie kodu w funkcji main()
def main():
    # Argumentami wiersza poleceń są sys.argv[1], sys.argv[2], ...
    # sys.argv[0] to nazwa samego skryptu, która może być ignorowana

    # parsowanie argumentów
    parser = argparse.ArgumentParser(description='Tworzy fotomozaikę z obrazów
                                                wejściowych')

    # dodawanie argumentów
    parser.add_argument('--target-image', dest='target_image', required=True)
    parser.add_argument('--input-folder', dest='input_folder', required=True)
    parser.add_argument('--grid-size', nargs=2, dest='grid_size', required=True)
    parser.add_argument('--output-file', dest='outfile', required=False)

    args = parser.parse_args()

    ##### DANE WEJŚCIOWE #####

    # obraz docelowy
    target_image = Image.open(args.target_image)

```

```

#obrazy wejściowe
print('wczytywanie folderu wejściowego...')
input_images = getImages(args.input_folder)

#sprawdzanie, czy znalezione zostały jakieś prawidłowe obrazy wejściowe
if input_images == []:
    print('Nie znaleziono żadnych obrazów wejściowych w %s. Zamykanie
    ↪ programu.' %
        (args.input_folder, ))
    exit()

#tasowanie listy w celu uzyskania bardziej urozmaiconych danych wyjściowych?
random.shuffle(input_images)

#rozmiar siatki
grid_size = (int(args.grid_size[0]), int(args.grid_size[1]))

#dane wyjściowe
output_filename = 'mosaic.png'
if args.outfile:
    output_filename = args.outfile

#ponowne wykorzystanie dowolnego obrazu z danych wejściowych
reuse_images = True

#zmiana rozmiaru danych wejściowych, aby pasowały do rozmiaru pierwotnego obrazu?
resize_input = True

##### KONIEC DANYCH WEJŚCIOWYCH #####

print('rozpoczęcie tworzenia fotomozaiki...')

#jeśli obrazy nie mogą być ponownie używane, należy upewnić się, że m*n <= num_of_images
if not reuse_images:
    if grid_size[0]*grid_size[1] > len(input_images):
        print('rozmiar siatki jest mniejszy niż liczba obrazów')
        exit()

#zmiana rozmiaru danych wejściowych
if resize_input:
    print('zmiana rozmiaru obrazów...')
    #dla danego rozmiaru siatki obliczanie maksymalnej szerokości i wysokości kafelków
    dims = (int(target_image.size[0]/grid_size[1]),
            int(target_image.size[1]/grid_size[0]))
    print("maksymalne wymiary kafelka: %s" % (dims,))
    #zmiana rozmiaru
    for img in input_images:
        img.thumbnail(dims)

#tworzenie fotomozaiki
mosaic_image = createPhotomosaic(target_image, input_images, grid_size,
                                  reuse_images)

#zapisywanie mozaiki
mosaic_image.save(output_filename, 'PNG')

print("dane wyjściowe zostały zapisane w %s" % (output_filename,))

```

```
print('zrobione.')

# Standardowy kod do wywoływania funkcji main(),
# aby rozpocząć program.
if __name__ == '__main__':
    main()
```

Uruchamianie generatora fotomozaiki

Oto przykładowe uruchomienie programu:

```
$ python photomosaic.py --target-image test-data/cherai.jpg --input-folder
test-data/set6/ --grid-size 128 128
wczytywanie folderu wejściowego...
rozpoczęcie tworzenia fotomozaiki...
zmiana rozmiaru obrazów...
maksymalne wymiary kafelka: (23, 15)
dzielenie obrazu docelowego...
wyszukiwanie dopasowań dla obrazu...
przetworzono 1638 z 16384...
przetworzono 3276 z 16384...
przetworzono 4914 z 16384...
tworzenie mozaiki...
dane wyjściowe zostały zapisane w mosaic.png
zrobione.
```

Rysunek 7.3a przedstawia obraz docelowy, a rysunek 7.3b pokazuje fotomozaikę. Zbliżenie fotomozaiki można zobaczyć na rysunku 7.3c.



Rysunek 7.3. Przykładowe uruchomienie generatora fotomozaiki

Podsumowanie

W ramach tego projektu dowiedziałeś się, jak utworzyć fotomozaikę, mając dany obraz docelowy oraz zbiór obrazów wejściowych. Fotomozaika oglądana z daleka wygląda jak pierwotny obraz, ale z bliska można zobaczyć poszczególne obrazy, które tworzą tę mozaikę.

Eksperymenty!

Oto kilka sposobów na dalsze eksperymentowanie z fotomozaikami:

- Napisz program, który tworzy pikselową wersję dowolnego obrazu, podobną do tej z rysunku 7.1.
- Za pomocą kodu zaprezentowanego w tym rozdziale tworzyliśmy fotomozaikę, wklejając dopasowane obrazy bez żadnych przerw pomiędzy nimi. Bardziej artystyczna prezentacja może zawierać jednolite przerwy w postaci kilku pikseli obramowania wokół każdego obrazu kafelkowego. W jaki sposób można utworzyć takie przerwy? (Podpowiedź: uwzględnij te przerwy podczas obliczania końcowych wymiarów obrazu oraz w trakcie wklejania w metodzie `createImageGrid()`).
- Ten program większość czasu poświęca na wyszukiwanie dla kafelków najlepszego dopasowania spośród obrazów z folderu wejściowego. Aby przyspieszyć działanie programu, trzeba sprawić, żeby metoda `getBestMatchIndex()` działała szybciej. Nasza implementacja tej metody opierała się na prostym przeszukiwaniu liniowym listy średnich (traktowanych jako trójwymiarowe punkty). To zadanie jest związane z ogólnym problemem **wyszukiwania najbliższego sąsiada**. Jednym ze szczególnie efektywnych sposobów wyszukiwania najbliższego punktu jest **przeszukiwanie drzewa kd**. Biblioteka `scipy` posiada klasę złożoną o nazwie `scipy.spatial.KDTree`, która pozwala tworzyć drzewo `kd` i odpytywać je o najbliższe dopasowania punktów. Spróbuj zastąpić przeszukiwanie liniowe drzewem `kd`, używając `SciPy`. (Zobacz <http://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.KDTree.html>).

Skorowidz

A

- adapter SparkFun, 281, 287
- adres IP, 305
- AJAX, 312, 322
- algorytm
 - Karplusa-Stronga, 78, 79, 83, 91
 - implementacja, 84
 - stada, 93, 95, 99
 - symulacja, 95, 107
 - volume ray casting, 218, 221, 223, 238
- ALSA, 352
- Arduino, 263, 264, 266, 288
 - adapter SparkFun, *Patrz:* adapter SparkFun
 - język programowania, 266
 - peryferia, *Patrz:* tarcza
 - społeczność, 266
- Arduino Uno, 264, 265
- Audacity, 82
- automat
 - komórkowy, 63
 - skończony, 158, 171, 173
- autostereogram, 141, 142
 - postrzeganie głębi, 142, 143
 - tworzenie, 148

B

- biblioteka
 - dowiązanie, 157, 167
 - flot, 305, 311, 312, 322
 - GLFW, 167, 222, 255
 - instalacja, 332
 - glutils.py, 170
 - jQuery, 314

- matplotlib, 28, 32, 66, 87, 98, 272, 273
- numpy, 95, 100
- OpenGL, *Patrz:* OpenGL
- PIL, 45, 115, 146, 181, 224
- Pillow, 45, 126
- pyttsx, 352
- scipy, 99
- billboarding, 191
- blendowanie, 200
 - alfa, 190, 199
 - OpenGL, 190
- blinkers, *Patrz:* światła uliczne
- blok try, 272
- boid, 93
 - animacja, 97, 98
 - położenie, 95, 97, 98
 - prędkość, 95, 97, 98, 101
- bootloader, 265
- breadboard, 338
- bryła widzenia, 163, 219
- bufor
 - cykliczny, *Patrz:* bufor pierścieniowy
 - głębi, 161, 170
 - pierścieniowy, 79, 80
 - próbek, 79, 80
 - ramek, *Patrz:* FBO
 - wierzchołków, 165
- buforowanie podwójne, 171

C

- centroid, 101
- cieniowanie
 - pikseli, 161, 163, 164, 176, 197, 222, 238, 249
 - wierzchołkowe, 160, 163, 164, 174, 195, 227, 240, 248, 249

Conway John, 63
CSS, 318
czcionka nieproporcjonalna, 113
czujnik
 światła, 267
 temperatury i wilgotności, *Patrz:* DHT11

D

dane
 audio, 280, 284
 analiza, 294
 wczytywanie, 292, 293
 kontenerowe, 83
 wolumetryczne 3D, 218
 rzut perspektywiczny, *Patrz:* rzut
 perspektywiczny danych
 wolumetrycznych 3D
 tworzenie wycinków 2D, 223, 246
deklaracja typu dokumentu, *Patrz:* DTD
depth map, *Patrz:* mapa głębi
DFT, 282
DHT11, 305, 306, 315, 327
dioda, 340
 anoda, 340
 katoda, 340
 LED, *Patrz:* LED
discrete Fourier transform, *Patrz:* DFT
document type declaration, *Patrz:* DTD
DTD, 27
duplikat, 27
 wyodrębnianie, 29
 wyszukiwanie, 28
dzielnik napięcia rezystancyjny, 268
dźwięk, 282
 amplituda, 81, 82, 283
 częstotliwość, 77, 80, 283, 284
 podstawowa, 78
 próbkiowania, 81
 składowa, 284, 294
 względna, 283
 odległość, *Patrz:* interwał muzyczny
 rozdzielczość, 81
 składowa harmoniczna, 80
 współczynnik tłumienia, 80

E

EAGLE, 342
epitrochoida, 42
espeak, 352

F

fala, 283, *Patrz też:* dźwięk
fast Fourier transform, *Patrz:* FFT
FBO, 221, 226
 tworzenie, 230
 zmiana rozmiaru okna, 233
FFT, 282, 283, 284, 341
 liczba próbek, 284
 obliczanie, 284, 293, 294
filtr dolnoprzepustowy, 80
filtrowanie liniowe, 182
flaga logiczna, 46, 47, 176
format
 EPS, 52
 Gerber, 344
 PNG, 51, 52
 WAV, 79, 81
 XML, 26
fotomozaika, 123
 rozmiar, 133
 tworzenie, 130, 131
fotorezystor, 267, 268, 275, 338
Fouriera transformata, *Patrz:* DFT, FFT
fragment shader, *Patrz:* cieniowanie pikseli
frame buffer object, *Patrz:* FBO
funkcja
 konfiguracyjna, 46
 ord, 295
 parametr, *Patrz:* parametr
 plot, 318
 routingu, 312

G

geometria
 3D, *Patrz:* grafika 3D
 rzutowa, 161, 162
GLFW, 157, 253, 332
glider, *Patrz:* szybowiec
GLSL, 163, 174, 217
GPIO, 306, 307
GPU, *Patrz:* procesor graficzny
gra
 Pac-Man, 65
 w życie, 63
 Gosper Glider Gun, 75
 plansza, 66
 reguły, 64, 69
 stan początkowy, 67, 68
 symulacja, 70, 73
 warunki brzegowe, *Patrz:* warunek brzegowy

grafika

3D, 160

blendowanie, 161

cieniowanie, *Patrz:* cieniowanie

inicjowanie, 171

raseteryzacja, 161

renderowanie, 171

rysowanie, 171

transformacja, *Patrz:* transformacja

współrzędne, *Patrz:* współrzędne

renderowanie, 157

skalowanie, 159

wektorowa, 52

H

HDMI, 307, 308, 353

hipotrochoida, 42, 281

Hunter John, 28

I

IC, *Patrz:* układ scalony

ICSP, 265

IDE, 266

in-circuit serial programming, *Patrz:* ICSP

integrated circuit, *Patrz:* układ scalony

interfejs

GPIO, 306, 307

użytkownika, *Patrz:* UI

interwał

muzyczny, 82

powtarzania, 149

iteracja, 30

iTunes, 25

J

język

Arduino, *Patrz:* Arduino język programowania

GLSL, 157

K

kamera, 352

kanał alfa, 190, 191, 200

katalog lista plików, 127

KiCad, 342

klasa

deque, 270

Image, 224

ParticleSystem, 200

RayCastRender, 238

RayCube, 226, 233, 238

RenderWin, 253

Scene, 171

SliceRender, 247, 248, 249

kod ASCII, *Patrz:* ASCII

kolor, 50

OpenGL, *Patrz:* OpenGL kolor

kondensator, 339

pojemność, 340

konstruktor, 45

kontener deque, 83

inicjowanie, 85

krzywa

epitrochoida, 42

hipotrochoida, 42

rysowanie, 39, 45, 47, 48

kula, 161

kursor

kształt, 46, 53

ukrywanie, 51, 52

wyświetlanie, 51

L

laser, 279

generowanie wzoru, 280, 281

wskaźnik, *Patrz:* wskaźnik laserowy

LDR, *Patrz:* fotorezystor

LED, 306, 315, 322, 327, 339

liczba, 50

licznik, 51

light-dependent resistor, *Patrz:* fotorezystor

list comprehension, *Patrz:* wyrażenie listowe

lista

łańcuchów znaków, 116, 117

odtwarzania, 25, 26, 28

tworzenie, 30

właściwości, 27

lokalizator zasobów, *Patrz:* URL

low-pass filter, *Patrz:* filtr dolnoprzepustowy

luminancja, 115

lutowie, 341

lutownica, 341

L

łańcuch znaków, 116
konwersja na liczbę całkowitą, 295
podział, 272
zapisywanie w pliku tekstowym, 117

M

macierz rzutowania, 164, 240
mapa głębi, 143, 144, 145, 148
mapowanie tekstury, 166
metoda
 addToDeq, 271
 appendleft, 271
 assert, 130
 choice, 67, 68
 concatenate, 225
 datetime, 52
 draw, 45
 ellipse, 147
 encode, 31
 get_device_count, 292
 getData, 321
 glActiveTexture, 174
 glDrawArrays, 174
 glDrawElements, 232
 glEnableVertexAttribArray, 172
 glFwSwapBuffers, 171
 glGetUniformLocation, 172
 glUniformMatrix4fv, 174
 glutils.ortho, 248
 glutils.perspective, 240
 glVertexAttribPointer, 173
 highlight, 323
 image.crop, 117, 129
 listdir, 224
 lookAt, 170
 mpl_connect, 101
 numpy.array, 32
 os.listdir, 127
 os.path.abspath, 127, 224
 os.path.join, 127, 224
 parser.add_mutually_exclusive_group, 33
 pdist, 99
 perspective, 170
 plot, 32
 plt.show, 67
 PWM, 289

randint, 50
range, 272
readPlist, 28
render, 171
renderCube, 231, 232
reshape, 95
scene.step, 171
set_data, 272
setTimeout, 321
setup, 53
split, 272
squareform, 99
subplot, 32
uniform, 50
update, 45
volume ray casting, *Patrz:* algorytm volume ray casting
 zip, 95
mikrokomputer jednoukładowy, 265
mikrokontroler AVR, 265
model boidów, 93
 efekt rozpraszania, 102
 reguły, 94, 101
 warunek brzegowy, *Patrz:* warunki brzegowe
modelview, *Patrz:* widok modelu
moduł
 Adafruit_DHT, 317
 argparse, 33, 52, 87
 Axes, 273
 collections, 83
 espeak, 352
 Figure, 273
 Image, 181
 kamery, 352
 math, 95
 Pillow, 45
 pyaudio, 292
 pygame, 85
 pyglfw, 332
 random, 50, 67
 tkinter, 52
 turtle, 44
 wave, 83, 85
monitor portu szeregowego, 266
mostek H, 287
multimetr, 340

N

największy wspólny dzielnik, 43
narzędzie
 ALSA, 352
 rsync, 349
NWD, 43

O

obiekt
 animowanie, 45, 48
 bufora ramki, *Patrz:* FBO
 canvas, 52
 Image, 127
 JSON, 318
 RayCube, 239
 rysowanie, 45
 set, 30
 trójwymiarowy, 191
 VAO, 166, 193
 VBO, 172
obraz, 51
 ASCII, 116
 jako dwuwymiarowa tablica, 148
 kafelek, 112, 115, 124, 129, 145
 brzegowy, 116
 jasność, 114, 115
 kolor, 125, 126, 128, 129
 odstęp, 149
 współrzędne, 116, 125
 miara jasności, *Patrz:* luminancja
 rozdzielczość, 52, 123, 124
 wczytywanie, 126
obwód
 drukowany, 339, 342
 elektroniczny, 339
 schemat, 342
okrąg, 40
OpenGL, 157, 158, 160
 kolor, 221
 kontekst, 167
 okno, 167, 222, 233, 253
 wyświetlanie, 167
OpenGL Shading Language, *Patrz:* shader GLSL,
 Patrz: język GLSL
operacja o stałym czasie wykonywania, 83
operator
 %, 69, 295
 *, 30
 modulo, 69, 295

opornik, *Patrz:* rezystor
oscylloskop, 341
overtone, *Patrz:* ton harmoniczny

P

pakiet RPi.GPIO, 310
parametr, 40
 inicjowanie, 46
 losowy, 49
parser, 33, 52
pas trójkątów, 166, 248, 249
PCB, *Patrz:* obwód drukowany
pentatonika, 82
pętla, 100
 for, 29, 49
 while, 170
plik
 dźwiękowy, 79
 nazwa unikatowa, 52
 ścieżka, *Patrz:* ścieżka
 WAV, *Patrz:* format WAV, *Patrz też:* format VAW
 odtworzenie, 85
 tworzenie, 85
 zapisywanie, 51
p-list, *Patrz:* lista właściwości
płytką prototypową, 315, 338, 342
polecenie
 ping, 348
 przychodzące, 305
port
 Ethernet, 307
 micro USB, 307
 szeregowy, 270, 273
 USB, 265, 306, 307
potok graficzny
 3D, 160
 programowalny, 159
 stałofunkcyjny, 159
półprzewodnik, 338
półton, 82
printed circuit board, *Patrz:* obwód drukowany
procesor graficzny, 157, 165
program
 Arduino, *Patrz:* szkic
 EAGLE, *Patrz:* EAGLE
 graficzny turtle, 44
 KiCad, 342
 PuTTY, 350
 uruchamianie, 37
 ponowne, 50

programator ICSP, *Patrz:* ICSP programator
property list, *Patrz:* lista właściwości
prymityw, 161
 3D, 159
 GL_POINTS, 189
 GL_TRIANGLE_STRIP, 247
 GL_TRIANGLES, 221
 kolejność wierzchołków, 222
 OpenGL, 161
 rysowanie, 222
przełącznik elektroniczny, 340
PyOpenGL, 157, 222

R

rampa, 112, 114
Raspberry Pi, 305, 306, 327, 347
 HDMI, 353
 kamera, 352
 konfiguracja, 309
 pakiet RPi.GPIO, 310
 kopia zapasowa
 plików, 349, 350
 systemu operacyjnego, 350
 logowanie, 350, 351
 mobilny, 353
 model, 307
 pamięć, 307
 podłączenie do sieci lokalnej, 305, 348
 SSH, *Patrz:* SSH
 system operacyjny, 308, 309
 wersja sprzętowa, 354
 WiFi, 309, 347
 tryb uśpienia, 348
 Wi-Fi, 308
 wyjście audio, 352
 wyłączanie, 315
RCA, 307, 308
rendering
 3D, 218
 objętościowy, 217, 218
Reynolds Craig, 93
rezystancja, 339
rezystor, 267, 315, 339
RGB, 147, 190
 wartość średnia, 112, 125, 126
RGBA, 190
ring buffer, *Patrz:* bufor pierścieniowy
RMS, 341
root mean square, *Patrz:* RMS

równanie
 mieszania, 190
 parametryczne, 40, 41
rzut
 ortograficzny, 163, 248
 perspektywiczny, 163
 danych wolumetrycznych 3D, 219
 płaszczyzna bliższa, 163, 219
 płaszczyzna dalsza, 163, 219
 pole widoku, 163

S

SDL, 85
serial monitor, *Patrz:* monitor portu szeregowego
serwer Bottle, 305, 312, 318
shader, 157, 162, 163, 174, 191, 210, 239
 GLSL, 217
 kodu kompilowanie, 165
shield, *Patrz:* tarcza
silniczek
 kierunek obracania, 289, 290, 295, 296, 298
 podłączanie, 288
 prędkość, 289, 290, 295, 296, 298
 sterowanie, 281, 289, 290, 291, *Patrz też:*
 adapter SparkFun
 modulacja szerokości impulsów, 289
 zatrzymywanie, 291
Simple DirectMedia Layer, *Patrz:* SDL
skala muzyczna, 82
skanowanie
 CT, 217, 219
 MRI, 217, 219
sketch, *Patrz:* szkic
słownik, 27, 86
słowo kluczowe
 in, 29, 164
 layout, 175
spirograf, 39, 54, 281
 model, 41
 okresowość, 43
SSH, 311, 327, 350
 sesja, 311
stado, 93, 99
system cząsteczek, 185, 193, 201
 animacja, 192
 iskra, 190, 192
 model matematyczny, 186, 187
 renderowanie, 189, 193
 równanie ruchu, 188
 tablica opóźnień czasowych, 194

sześcian kolorów, 220, 223
 geometria, 227
 renderowanie, 231, 232, 240
 rysowanie, 226
szkic, 266, 288
 czujnik światła, 269
 przesyłanie do Arduino, 270
 tworzenie, 266
sztuka ASCII, 111, 112
szybowiec, 64, 68, 73

Ś

ścieżka
 bezwzględna, 127
 względna, 127, 224
średnia kwadratowa, *Patrz:* RMS
światła uliczne, 64, 73

T

tablica, 66
 definiowanie, 67
 dwuwymiarowa, 95, 115
 jednowymiarowa, 95
 numpy, 32, 82, 95, 115, 222, 224
tarcza, 266
tekst zamiana na mowę, 352
tekstura, 191, 200
 3D, 219, 223, 239
 adresowanie, 219
 generowanie, 223
 wymiały, 225
 jednostka, 174, 176, 199
 mapowanie, *Patrz:* mapowanie tekstury
 OpenGL, 225, 239
 współrzędne, 175, 176
texture mapping, *Patrz:* mapowanie tekstury
timer, 45, 50, 170, 339
ton harmoniczny, 78, 282
topik, 341
torus, 65
transformacja, 161, 162
 generyczna, 162
 perspektywiczna, 239
 rzutowania, 162, *Patrz też:* rzut
 widoku modelu, 162
transformata Fouriera
 dyskretna, *Patrz:* DFT
 szybka, *Patrz:* FFT

translacja, 162
 tranzystor, 339, 340
 MOSFET, 287, 340
triangle strip, *Patrz:* pas trójkątów

U

UI, 52
układ
 Broadcom BCM2835, 307
 elektroniczny, 338
 scalony, 339, 340
 ATmega328, 265
uniform resource locator, *Patrz:* URL
URL, 27
USB, 265, 307

V

VAO, 247
VBO, 247
vertex buffer, *Patrz:* bufor wierzchołków
vertex shader, *Patrz:* cieniowanie wierzchołkowe
view frustrum, *Patrz:* bryła widzenia

W

wartość alfa, 190, 191, 210
warunek brzegowy, 65, 94, 116
 kafelkowy, 96
 toroidalny, 65, 68, 96
wektor
 jednostkowy prędkości, 95
 normalny, 191, 222
 trójwymiarowy, 188
widok modelu, 162, 164, 170
widzenie rozbieżne, 141, 142
wiersz poleceń, 86, 132
 argument, 33, 52
wskaźnik laserowy, 279, 280, 285
współczynnik wypełnienia, 289
współrzędne
 3D mapowanie, 163
 jednorodne, 161, 162
 trójwymiarowe, 188
wyjątek, 224, 272
wykres, 31, 32, 87, 273, 276, 311, 312, 318
 etykieta, 32
 w czasie rzeczywistym, 270, 271
widmowy, 77

wyrażenie listowe, 53
wywołanie zwrotne, 168
wzmacniacz napięcia, 340

Z

zamiana tekstu na mowę, 352
zdarzenie
 klawiatury, 169, 222, 249
 myszki, 102, 168
 wciśnięcia przycisku, 101
 wywołanie zwrotne, 168, 169
 zmiany rozmiaru okna, 169
złącze composite video, *Patrz*: RCA
zmienna, 33

znacznik, 26
 dict, 27
 key, 27
 plist, 27
 xml, 27
znak
 %, 69, 295
 *, 30
 <, 100
 łańcuch, *Patrz*: łańcuch znaków

Ż

żądanie AJAX, 322

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION



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

Zmień swoją stronę WWW
w działający bankomat!

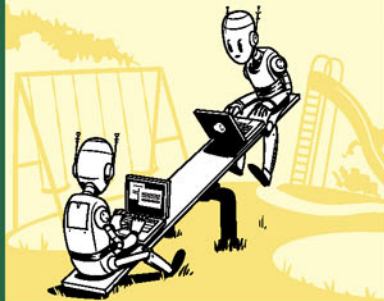
Dowiedz się więcej i dołącz już dzisiaj!

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

GRUPA WYDAWNICZA



Helion SA



PYTHON? SPRÓBUJ SIĘ Z NIM POBawiĆ!

Python jest ciekawym językiem programowania o dużych możliwościach. Dzięki niemu w prosty sposób można rozwiązać złożone problemy tego świata. Umożliwia przy tym pisanie czytelnego i łatwego w konserwacji kodu. Opanowanie składni i podstawowych koncepcji programistycznych w Pythonie nie jest trudne, jednak potem przychodzi moment, aby wypróbować go w prawdziwym programowaniu.

Niniejsza książka nauczy Cię wykorzystywać ten język do rozwiązywania nietrywialnych problemów, z którymi muszą się mierzyć programiści. Książka składa się z czternastu zabawnych i inspirujących projektów, dzięki którym odkryjesz niuanse programowania i nauczysz się pracy z kilkoma bibliotekami Pythona. Co ważniejsze, nauczysz się analizy problemu, dowiesz się, jak opracować algorytm do jego rozwiązania, a następnie jak zaimplementować rozwiązanie. Wykorzystasz Pythona do tworzenia muzyki, symulacji rzeczywistych zjawisk, a także zmusisz do współpracy z płytkami Arduino i Raspberry Pi — a wszystko w ramach świetnej, wciągającej zabawy!

Sprawdź, jak wykorzystać Pythona do:

- generowania spirografowych wzorów
- tworzenia muzyki na komputerze
- przekładania obrazów na sztukę ASCII
- tworzenia realistycznych animacji za pomocą biblioteki OpenGL
- wizualizacji 3D danych z obrazowania medycznego CT i MRI
- zbudowania internetowego systemu monitorowania pogody z wykorzystaniem Raspberry Pi

Mahesh Venkitachalam — jest inżynierem oprogramowania z dwudziestoletnim doświadczeniem w programowaniu. Od lat rozwija aplikacje służące naukowcom do wizualizacji 3D. Pracuje również nad podzespołami elektronicznymi, które udostępnia amatorom elektroniki na zasadach open source. Jest pasjonatem technologii, prowadzi popularny blog o elektronice i programowaniu — electronut.in. Mieszka w Indiach.

Helion

księgarnia internetowa



<http://helion.pl>

zamówienia telefoniczne



0 801 339900



0 601 339900

Informatyka w najlepszym wydaniu

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
<http://helion.pl>

Sprawdź najnowsze promocje:
• <http://helion.pl/promocje>
Książki najchętniej czytane:
• <http://helion.pl/bestsellery>
Zamów informacje o nowościach:
• <http://helion.pl/nawosci>

sięgnij po WIĘCEJ



KOD KORZYŚCI

ISBN 978-83-283-2597-5



9 788328 325975

cena: 59,00 zł

