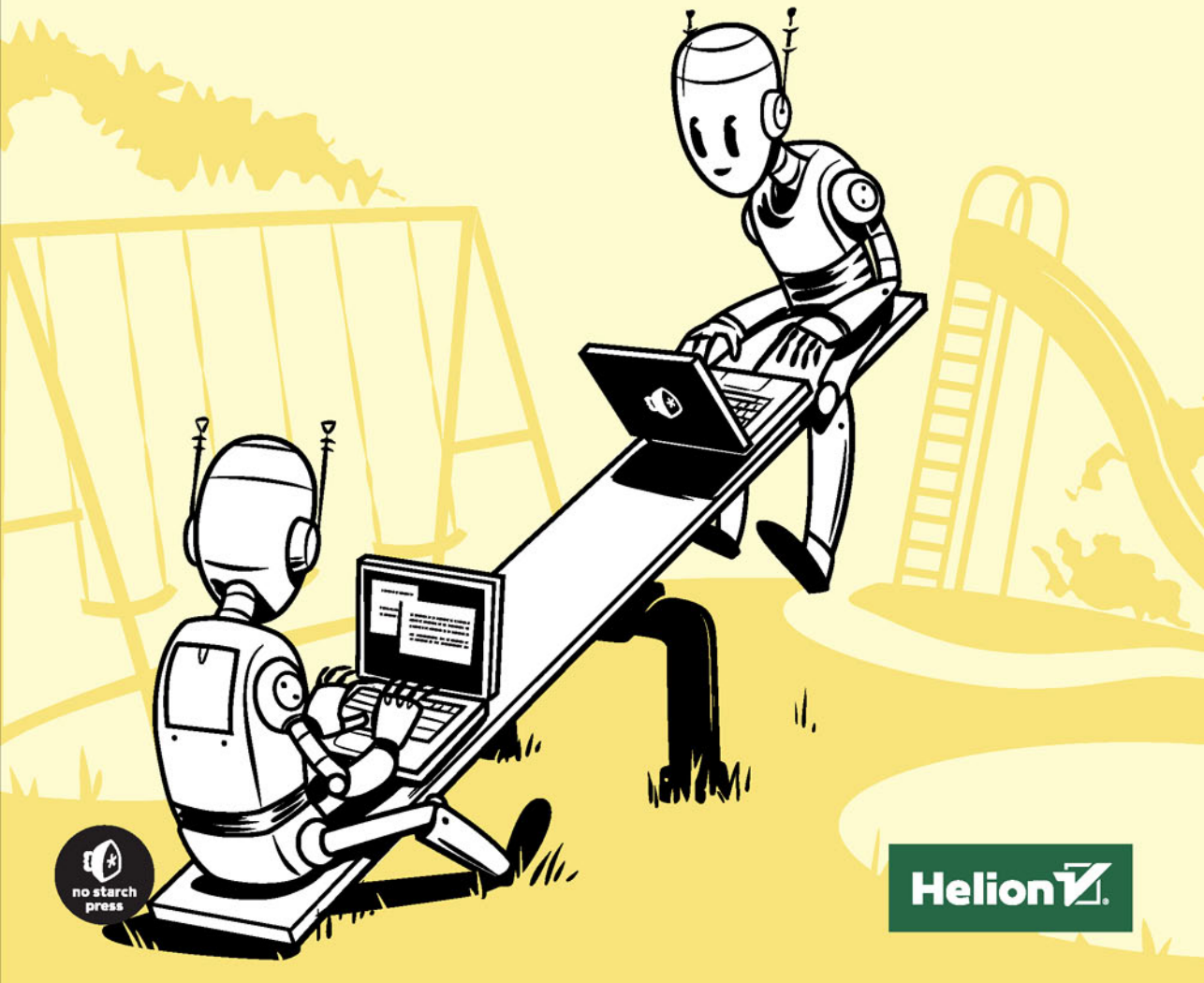
PYTHON

14 TWÓRCZYCH PROJEKTÓW DLA DOCIEKLIWYCH PROGRAMISTÓW

MAHESH VENKITACHALAM



no starch press

Helion

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

Tłumaczenie: Lech Lachowski

ISBN: 978-83-283-2597-5

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

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)

Spis treści

Część I. Rozgrzewka

Część II. Symulacja życia

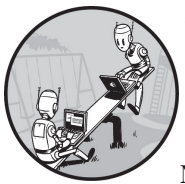
Część IV. Grafika 3D

Część V. Projekty z wykorzystaniem sprzętu

Dodatki

7

Fotomozaiki

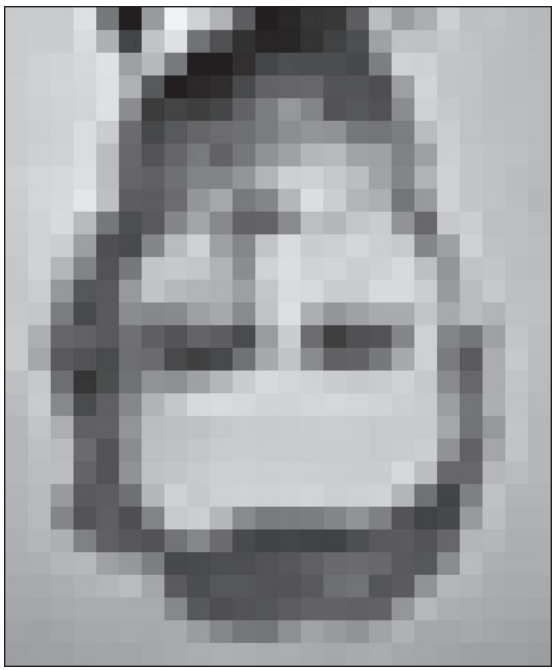


KIEDY BYŁEM W SZÓSTEJ KLASIE, ZOBACZYŁEM PEWIEN OBRAZEK, TAKI JAK TEN POKAZANY NA RYSUNKU 7.1, ALE NIE POTRAFIŁEM SIĘ ZORIENTOWAĆ, CO NA NIM JEST. PRZYPATRUJĄC MU się przez chwilę, w końcu zrozumiałem. (Obróć książkę do góry nogami i przyjrzyj się obrazkowi z drugiego końca pokoju. Nikomu nie powiem).

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. Podzielisz 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.

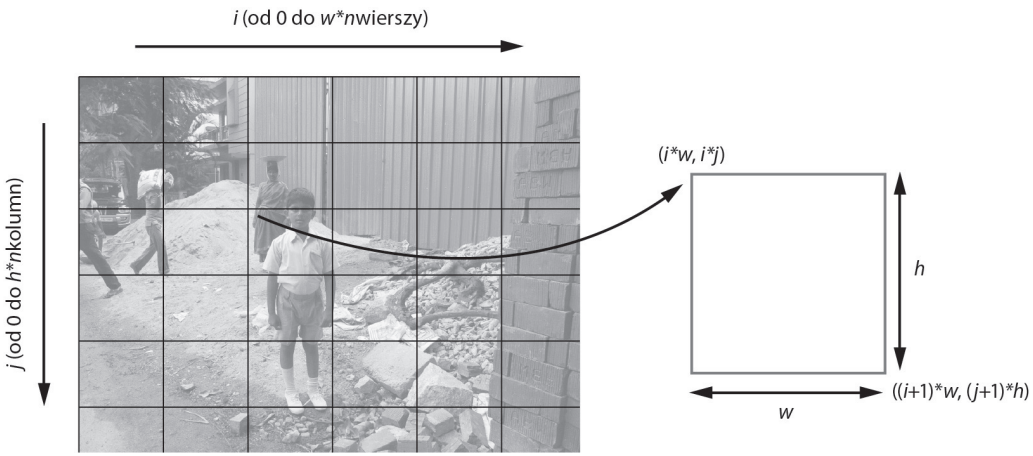
Jak to działa?

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

Rozpoczniemy 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:

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

Należy zauważyć, że średnia wartość RGB jest również tripletem, a nie skalarem 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 ❺ 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) ❶
    # uzyskanie kształtu każdego obrazu wejściowego
    w,h,d = im.shape ❷
    # uzyskanie średniej wartości RGB
    return tuple(np.average(im.reshape(w*h, d), axis=0)) ❸
```

W punkcie ❶ 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 ❷ zapisujemy krotkę `shape`, a następnie w punkcie ❸ 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] ❶
    m, n = size ❷
    w, h = int(W/n), int(H/m) ❸
    # 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))) ❹
    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 ❺ 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 fotomozaiki, potrzebujemy jeszcze jednej metody narzędziowej. Metoda `createImageGrid()` tworzy siatkę obrazów, która ma rozmiar $M \times N$. Ta siatka obrazu jest finalnym obrazem fotomozaiki, 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 działać 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 ❻ 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***row* + *col*, gdzie *N* jest liczbą komórek w wierszu, a (*row*, *col*) to współrzędne na tej siatce. W punkcie ❹ podajemy wiersz z poprzedniego wzoru, a w punkcie ❺ 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: ❽
            input_images.remove(match)

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

    # 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 ❼ 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 ❼ dla każdej liczby `batch_size` przetworzonych obrazów wyświetlamy komunikat dla użytkownika. W punkcie ❽, 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 ❾ 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.tumbnail()` 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 fotomozaiki 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 *k*d**. Biblioteka `scipy` posiada klasę złożoną o nazwie `scipy.spatial.KDTree`, która pozwala tworzyć drzewo *k*d i odpytywać je o najbliższe dopasowania punktów. Spróbuj zastąpić przeszukiwanie liniowe drzewem *k*d, używając SciPy. (Zobacz *http://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.KDTree.html*).

Skorowidz

D

E

F

G

H

I

J

K

L

Ł

M

R

S

Ś

T

U

V

W

Z

Ż

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
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/nowosci
Informatyka w najlepszym wydaniu
ISBN 978-83-283-2597-5
9 788328 325975
cena: 59,00 zł
no starch press