O'REILLY®

OpenCV 3

Komputerowe rozpoznawanie obrazu w C++ przy użyciu biblioteki OpenCV

Helion

Adrian Kaehler,
Gary Rost Bradski

Tytuł oryginału: Learning OpenCV 3: Computer Vision in C++ with the OpenCV Library

Tłumaczenie: Łukasz Piwko

ISBN: 978-83-283-1656-0

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)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/opencv.zip

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/opencv
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

ROZDZIAŁ 11.

Ogólne przekształcenia obrazu

Wprowadzenie

W poprzednich rozdziałach opisaliśmy klasę przekształceń obrazu, które można przedstawić na podstawie splotu. Oczywiście istnieje wiele operacji, których w ten sposób nie da się wyrazić (tzn. przy użyciu małego okienka przesuwanego nad obrazem i wykonującego jakieś czynności). Zasadniczo przekształcenia, które można wyrażać jako sploty, mają charakter lokalny, to znaczy choć mogą zmieniać cały obraz, sposób przetworzenia konkretnego piksela zależy tylko od niewielkiej liczby otaczających go pikseli. Transformacje opisywane w tym rozdziale są pozbawione tej właściwości.

Niektóre z najbardziej przydatnych *przekształceń obrazu* są proste (np. zmiana rozmiaru) i stosuje się je bardzo często. Inne są przeznaczone do znacznie bardziej zaawansowanych zastosowań. Przedstawione przez nas w tym rozdziale przekształcenia powodują przekonwertowanie jednego obrazu w inny. Obraz wynikowy często będzie miał inny rozmiar niż oryginał albo będzie się od niego różnił pod innym względem, ale zawsze będzie zasadniczo obrazem w takim samym sensie jak grafika wejściowa. W rozdziale 12. omawiamy także operacje, które w wyniku mogą zwrócić całkiem nową reprezentację.

W komputerowym rozpoznawaniu obrazu jest kilka przekształceń, których używa się bardzo często. Biblioteka OpenCV zawiera kompletne implementacje niektórych najpopularniejszych spośród nich oraz składniki potrzebne do implementacji własnych, bardziej zaawansowanych transformacji.

Rozciąganie, kurczenie, zniekształcanie i obracanie

Najprostszym możliwym rodzajem przekształcenia jest zmiana rozmiaru obrazu w celu jego powiększenia lub zmniejszenia. Operacje te jednak nie są wcale tak banalne, jak się wydaje, ponieważ zmiana rozmiaru natychmiast implikuje pytania o sposób interpolacji (w przypadku powiększania) lub łączenia (w przypadku zmniejszania) pikseli.

Jednorodna zmiana rozmiaru

Często otrzymujemy obraz o określonym rozmiarze i chcemy go zmniejszyć lub powiększyć. Oba te zadania możemy wykonać za pomocą jednej funkcji.

cv::resize()

Funkcja `cv::resize()` zaspokaja wszystkie potrzeby programisty w dziedzinie zmiany rozmiaru obrazów. Należy jej przekazać na wejściu obraz i określić rozmiar docelowy, a funkcja wygeneruje nowy obraz o żądanym rozmiarze.

```
void cv::resize(
  cv::InputArray src,                  // obraz wejściowy
  cv::OutputArray dst,                 // obraz wynikowy
  cv::Size dsize,                      // nowy rozmiar
  double fx = 0,                       // skala wymiaru x
  double fy = 0,                       // skala wymiaru y
  int interpolation = CV::INTER_LINEAR // metoda interpolacji
);
```

Rozmiar obrazu wynikowego można określić na dwa sposoby. Pierwszy polega na podaniu rozmiaru *bezwzględnego* w argumencie `dsize`, który bezpośrednio określa, jaki rozmiar ma mieć przekształcony obraz `dst`. Drugi sposób polega na określeniu wartości *względnych*. W tym przypadku argument `dsize` ustawia się na `cv::Size(0,0)`, a następnie argumentom `fx` i `fy` przypisuje się współczynniki skalowania odpowiednio dla osi x i y[1]. Ostatni argument wskazuje metodę interpolacji. Domyślnie stosowana jest interpolacja liniowa. Pozostałe dostępne opcje są pokazane w tabeli 11.1.

Tabela 11.1. Opcje interpolacji funkcji cv::resize()

Interpolacja	Opis
`cv::INTER_NEAREST`	Najbliższy sąsiad
`cv::INTER_LINEAR`	Dwuliniowa
`cv::INTER_AREA`	Przepróbkowywanie obszaru pikseli
`cv::INTER_CUBIC`	Interpolacja dwusześcianowa
`cv::INTER_LANCZOS4`	Interpolacja algorytmem Lanczosa uwzględniająca sąsiadujące punkty w kwadracie o wymiarach 8×8

Interpolacja w tym przypadku odgrywa bardzo ważną rolę. Piksele w obrazie źródłowym są rozmieszczone w siatce całkowitoliczbowej, dzięki czemu możemy odwołać się na przykład do piksela o współrzędnych (20, 17). Podczas odwzorowywania tych całkowitoliczbowych współrzędnych w nowym obrazie mogą powstawać luki. Przyczyną powstawania takich luk może być odwzorowywanie współrzędnych całkowitoliczbowych na zmiennoprzecinkowe, które muszą zostać zaokrąglone do najbliższej liczby całkowitej, albo to, że dla niektórych lokalizacji nie ma żadnego piksela (pomyśl o podwajaniu rozmiaru obrazu przez jego rozciągnięcie — w takim przypadku co drugi piksel docelowy byłby pusty). Tego typu trudności określa się zbiorczo mianem problemów **rzutowania w przód** (ang. *forward projection*). Aby je rozwiązać, musimy podejść do tego od drugiej strony, a mianowicie przeglądamy po kolei każdy piksel obrazu docelowego i pytamy: które piksele

[1] Albo argument `dsize` musi mieć wartość `cv::Size(0,0)`, albo argumenty `fx` i `fy` muszą mieć wartość 0.

ze źródła mogą zostać wykorzystane w tym miejscu? Wybrane piksele źródłowe prawie zawsze wypadają w lokalizacjach ułamkowych, więc musimy dokonywać ich interpolacji, aby otrzymać prawidłową wartość dla naszej wartości docelowej. Domyślną metodą jest interpolacja dwuliniowa, ale do wyboru są także inne metody (wymienione w tabeli 11.1).

Najprostszym rozwiązaniem jest pobranie wartości dla piksela w obrazie rozszerzonym z najbliższego mu piksela w pierwotnym obrazie. Tak właśnie działa opcja interpolacji `cv::INTER_NEAREST`. Ewentualnie można wykorzystać sąsiednie piksele źródłowe z obszaru o wymiarach 2×2, ważąc je z uwzględnieniem ich odległości od piksela docelowego — tak działa opcja `cv::INTER_LINEAR`. Można też wirtualnie umieścić nowy piksel na starym pikselu, a następnie obliczyć średnią pokrytych wartości pikseli — tak działa metoda `cv::INTER_AREA`[2]. Jeszcze lepszy efekt wygładzania można uzyskać, wpasowując sześcienną krzywą składaną między sąsiadującymi pikselami na obszarze o wymiarach 4×4 w obrazie źródłowym, a następnie odczytując odpowiednią wartość docelową z tej krzywej — tak działa metoda interpolacji `cv::INTER_CUBIC`. Ostatnia jest interpolacja algorytmem Lanczosa, która przypomina metodę sześcienną, ale wykorzystuje informacje z obszaru wokół piksela o wymiarach 8×8[3].

Należy podkreślić różnicę między funkcją `cv::resize()` i podobnie nazwaną funkcją składową klasy `cv::Mat` — `cv::Mat::resize()`. Funkcja `cv::resize()` tworzy nowy obraz o innym rozmiarze niż oryginał, na którym odwzorowane są piksele z pierwotnej grafiki. Funkcja składowa `cv::Mat::resize()` zmienia rozmiar obrazu, którego składową wywołano, i przycina go do odpowiedniego rozmiaru. W przypadku funkcji `cv::Mat::resize()` piksele nie są interpolowane (ani ekstrapolowane).

Piramidy obrazów

Piramidy obrazów [Adelson84] znajdują bardzo szerokie zastosowanie w wielu dziedzinach komputerowego rozpoznawania obrazu. Piramida obrazów to zbiór grafik (wszystkie utworzone na bazie jednego podstawowego obrazu) o stopniowo zmniejszanym próbkowaniu, aż do osiągnięcia pewnego punktu krańcowego (punkt ten może być nawet jednopikselowym obrazem!).

W literaturze przedmiotu i praktyce wyróżnia się dwa podstawowe rodzaje piramid obrazów: Gaussa [Rosenfled80] i Laplace'a [Burt83]. Piramida Gaussa służy do zmniejszania próbkowania, a piramida Laplace'a (którą opisujemy dalej) jest używana, gdy trzeba odtworzyć obraz o większym próbkowaniu na podstawie obrazu z niższego poziomu piramidy.

cv::pyrDown()

Normalnie warstwę $(i + 1)$ w piramidzie Gaussa (oznaczymy tę warstwę G_{i+1}) tworzy się z warstwy G_i piramidy, najpierw splatając G_i z jądrem gaussowskim, a następnie usuwając każdy rząd i każdą kolumnę o parzystym numerze. Oczywiście wówczas otrzymuje się obraz, którego rozmiar

[2] Tak to przynajmniej wygląda, gdy funkcja `cv::resize()` zmniejsza obraz. W przypadku powiększania opcja `cv::INTER_AREA` daje taki sam efekt jak `cv::INTER_NEAREST`.

[3] Szczegółowy opis filtra Lanczosa wykracza poza zakres tematyczny tej książki, ale warto wiedzieć, że algorytm ten jest powszechnie wykorzystywany w przetwarzaniu obrazów cyfrowych, ponieważ zwiększa *pozorną* ostrość obrazu.

wynosi dokładnie jedną czwartą rozmiaru poprzedniego obrazu. Wielokrotne przeprowadzenie tego procesu na obrazie wejściowym G_0 pozwala otrzymać całą piramidę. Biblioteka OpenCV udostępnia metodę do generowania każdego poziomu piramidy z poprzedniego:

```
void cv::pyrDown(
  cv::InputArray src,                  // obraz wejściowy
  cv::OutputArray dst,                 // obraz wynikowy
  const cv::Size& dstsize = cv::Size() // rozmiar obrazu wyjściowego
);
```

Metoda `cv::pyrDown()` robi dokładnie to, o czym napisaliśmy wyżej, jeśli pozostawimy argument rozmiaru docelowego `dstsize` przy wartości domyślnej `cv::Size()`. Mówiąc dokładniej: domyślny rozmiar obrazu wyjściowego wynosi ((`src.cols`+1)/2, (`src.rows`+1)/2)[4]. Ewentualnie możemy przekazać argument `dstsize`, aby określić rozmiar obrazu wyjściowego. Wartość tego argumentu musi jednak spełniać pewne bardzo ścisłe warunki:

$$\left| \texttt{dstsize.width} \cdot 2 - \texttt{src.cols} \right| \leq 2$$

$$\left| \texttt{dstsize.height} \cdot 2 - \texttt{src.rows} \right| \leq 2$$

Wskutek tego ograniczenia obraz wynikowy ma rozmiar *bardzo bliski* połowie rozmiaru obrazu źródłowego. Argumentu `dstsize` używa się tylko w rzadkich przypadkach, gdy trzeba ściśle kontrolować sposób tworzenia piramidy.

cv::buildPyramid()

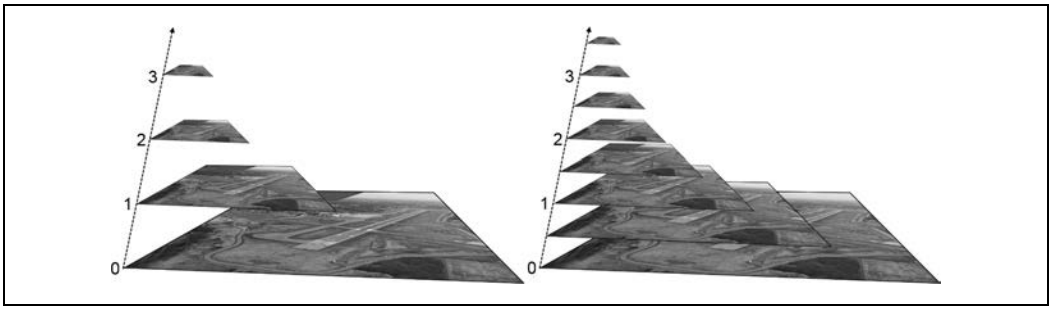
Stosunkowo często zdarzają się sytuacje, gdy trzeba utworzyć zbiór obrazów, z których każdy kolejny jest pomniejszoną wersją poprzedniego. Do tworzenia takich stosów służy funkcja `cv::buildPyramid()`:

```
void cv::buildPyramid(
  cv::InputArray src,           // obraz wejściowy
  cv::OutputArrayOfArrays dst, // obrazy piramidy
  int maxlevel                  // liczba poziomów piramidy
);
```

Argument `src` określa obraz źródłowy. Z kolei argument `dst` ma dość niezwykły typ `cv::OutputArray ↪OfArrays`, który można traktować jako wektor STL `vector<>` obiektów typu `cv::OutputArray`. Najlepszym przykładem takiej struktury jest `vector<cv::Mat>`. Natomiast argument `maxlevel` sygnalizuje, ile poziomów ma mieć tworzona piramida.

Argument `maxlevel` może mieć dowolną wartość całkowitoliczbową nie mniejszą od zera, ponieważ wskazuje, z ilu obrazów ma się składać wygenerowana piramida. Po zakończeniu działania funkcja `cv::buildPyramid()` zwraca w argumencie `dst` wektor o długości `maxlevel+1`. Pierwszy element w tablicy `dst` jest identyczny z obrazem `src`. Drugi jest o połowę mniejszy od pierwszego, to znaczy jest taki, jaki zostałby zwrócony przez funkcję `cv::pyrDown()`. Trzeci obraz jest o połowę mniejszy od drugiego itd. (ilustracja po lewej na rysunku 11.1).

[4] Składniki *+1* są po to, aby zapewnić prawidłową obsługę obrazów o nieparzystych rozmiarach. W przypadku obrazów o rozmiarach parzystych nie mają znaczenia.



Rysunek 11.1. Piramida obrazów wygenerowana przy użyciu argumentu maxlevel = 3 (po lewej); dwie przeplatające się piramidy tworzące razem piramidę $\sqrt{2}$ (po prawej)

Piramidy wykorzystywane w praktyce często są generowane przy użyciu drobniejszych logarytmicznych współczynników skalowania. Jednym ze sposobów pozwalających wygenerować taką piramidę jest własnoręczne wywołanie funkcji `cv::resize()` odpowiednią liczbę razy, ale to dość czasochłonne rozwiązanie. Innym sposobem (w przypadku niektórych często używanych współczynników skalowania) jest wywołanie funkcji `cv::resize()` tylko raz dla każdego z zestawów obrazów, które mają zostać później splecione, a następnie przekazanie każdej z tak otrzymanych „podstaw" do funkcji `cv::buildPyramid()`. Kolejną czynnością jest właśnie złożenie otrzymanych obrazów na przemian w celu utworzenia gęstej piramidy. Na rysunku 11.1 (po prawej) pokazany jest przykład wygenerowania dwóch piramid. Pierwotny obraz został najpierw przeskalowany o współczynnik $\sqrt{2}$, a następnie przekazano go do funkcji `cv::buildPyramid()` w celu utworzenia drugiej piramidy złożonej z czterech obrazów. Po połączeniu nowego zestawu z poprzednią piramidą powstała nowa, gęstsza piramida o współczynniku skalowania $\sqrt{2}$.

cv::pyrUp()

Wybrany obraz można też przekonwertować na dwa razy większy w każdym wymiarze. Służy do tego operacja analogiczna (ale nie odwrotna!) do opisanej wcześniej operacji, której rezultatem są zmniejszone obrazy:

```
void cv::pyrUp(
  cv::InputArray src,                  // obraz wejściowy
  cv::OutputArray dst,                 // obraz wynikowy
  const cv::Size& dstsize = cv::Size() // rozmiar obrazu wynikowego
);
```

Funkcja ta najpierw powiększa obraz dwukrotnie w każdym wymiarze i nowe (parzyste) wiersze wypełnia zerami, a następnie aproksymuje „brakujące" wartości pikseli za pomocą splotu z filtrem Gaussa[5].

Analogicznie jak w przypadku funkcji `cv::PyrDown()`, jeżeli argument `dstsize` będzie miał pozostawioną wartość domyślną `cv::Size()`, wygenerowany obraz będzie dokładnie dwa razy większy (w każdym wymiarze) niż obraz `src`. Oczywiście także w tym przypadku w argumencie `dstsize`

[5] Filtr ten jest również znormalizowany do wartości 4, a nie 1. Jest to właściwe, ponieważ wstawiane wiersze przed splotem mają wszystkie piksele ustawione na zero. (Normalnie suma elementów jądra gaussowskiego wynosiłaby 1, ale w przypadku dwukrotnego skalowania piramidy — jeśli chodzi o dwa wymiary — wszystkie elementy jądra są mnożone przez 4, aby przywrócić średnią jasność po wstawieniu zerowych wierszy i kolumn).

możemy przekazać własną wartość określającą rozmiar obrazu wynikowego, ale musi ona spełniać pewne ścisłe warunki:

$$\left|\texttt{dstsize.width} \cdot 2 - \texttt{src.cols}\right| \leq (\texttt{dstsize.width} / 2)$$

$$\left|\texttt{dstsize.height} \cdot 2 - \texttt{src.rows}\right| \leq (\texttt{dstsize.height} / 2)$$

Z warunku tego wynika, że rozmiar obrazu docelowego musi być *bardzo bliski* dwukrotności rozmiaru obrazu źródłowego. Tak jak poprzednio, argument `dstsize` jest wykorzystywany tylko w stosunkowo rzadkich przypadkach, gdy trzeba dokładnie kontrolować sposób tworzenia piramidy.

Piramida Laplace'a

Wcześniej napisaliśmy, że działanie operatora `cv::pyrUp()` nie jest odwrotnością sposobu działania operatora `cv::pyrDown()`. Powinno to być oczywiste, ponieważ `cv::pyrDown()` gubi informacje. Do odzyskania pierwotnego obrazu o wyższej rozdzielczości potrzebne są informacje, które wcześniej zostały skasowane w procesie redukcji. I właśnie te dane stanowią **piramidę Laplace'a** (ang. *Laplacian pyramid*). Definicja *i*-tej warstwy tej piramidy ma postać następującej relacji:

$$L_i = G_i - UP(G_{i+1}) \otimes g_{5x5}$$

W procesie zwiększania rozmiaru przez operator `UP()` każdy piksel o współrzędnych (x, y) z pierwotnego obrazu jest odwzorowywany na piksel o współrzędnych $(2x + 1, 2y + 1)$ w obrazie docelowym. Symbolem $\otimes$ oznaczany jest splot, a wyraz g_{5x5} oznacza jądro gaussowskie o wymiarach 5×5. Oczywiście $UP(G_{i+1}) \otimes g_{5\times5}$ to definicja operatora `cv::pyrUp()` z biblioteki OpenCV. W związku z tym możemy bezpośrednio przedstawić operator Laplace'a w następującej postaci:

$$L_i = G_i - pyrUp(G_{i+1})$$

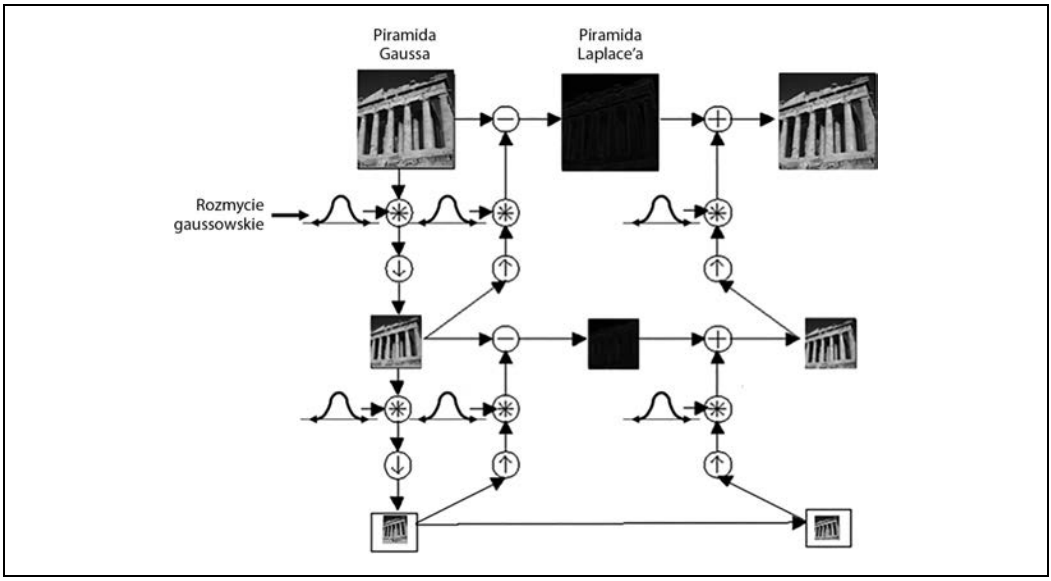
Rysunek 11.2 przedstawia piramidy Gaussa i Laplace'a w formie schematu, na którym dodatkowo widać odwrotny proces mający na celu przywrócenie pierwotnego obrazu ze zredukowanych obrazów. Zwróć uwagę, że obraz piramidy Laplace'a jest tak naprawdę aproksymacją wykorzystującą różnicę obrazów z piramidy Gaussa, co wynika z poprzedniego równania i jest widoczne na schemacie.

Mapowanie niejednorodne

W tej sekcji kierujemy uwagę na *geometryczne* operacje na obrazach, to znaczy takie przekształcenia, które biorą swój początek na przecięciu geometrii trójwymiarowej i projekcyjnej[6]. Do operacji takich zaliczają się zarówno jednolite, jak i niejednolite zmienianie rozmiaru (to drugie nazywa się **odkształcaniem** — ang. *warping*). Działania takie mogą być wykonywane z wielu powodów, na przykład można odkształcić i obrócić obraz w celu nałożenia go na ścianę na scenie lub sztucznie powiększyć zbiór obrazów szkoleniowych wykorzystywanych w treningu rozpoznawania obiektów[7].

[6] Szczegółowy opis tych przekształceń zamieściliśmy w tej sekcji, ale wracamy do nich jeszcze w rozdziale 19., przy okazji omawiania metod ich użycia w trójwymiarowych technikach wizyjnych.

[7] Ta czynność może się wydawać nieco naciągana, bo przecież równie dobrze można skorzystać z metody rozpoznawania, która jest niezależna od lokalnych zniekształceń afinicznych. Niemniej jednak metoda ta ma długą historię i niejednokrotnie okazała się niezwykle przydatna w praktyce.



Rysunek 11.2. Piramida Gaussa i jej odwrotność, czyli piramida Laplace'a

Funkcje rozciągające, ściskające, zniekształcające lub obracające obrazy nazywają się **transformacjami geometrycznymi** (wczesny przykład można znaleźć w [Semple79]). W odniesieniu do płaszczyzn wyróżnia się dwa rodzaje transformacji geometrycznych: **afiniczne**, które bazują na macierzach o wymiarach 2×3, oraz **perspektywiczne**, zwane też **homografiami**, które bazują na macierzach o wymiarach 3×3. Ten drugi rodzaj transformacji można sobie wyobrażać jako metodę obliczania tego, w jaki sposób dana płaszczyzna w trzech wymiarach jest postrzegana przez wybranego obserwatora, który może nie patrzeć na nią wprost.

Transformacja afiniczna to każda transformacja, którą można wyrazić w postaci mnożenia macierzy, po którym następuje dodanie wektora. W OpenCV ten rodzaj transformacji standardowo przedstawia się w postaci macierzy o wymiarach 2×3. Mamy następującą definicję:

$$A \equiv \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} B \equiv \begin{bmatrix} b_0 \\ b_1 \end{bmatrix} T \equiv \begin{bmatrix} A & B \end{bmatrix} X \equiv \begin{bmatrix} x \\ y \end{bmatrix} X' \equiv \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

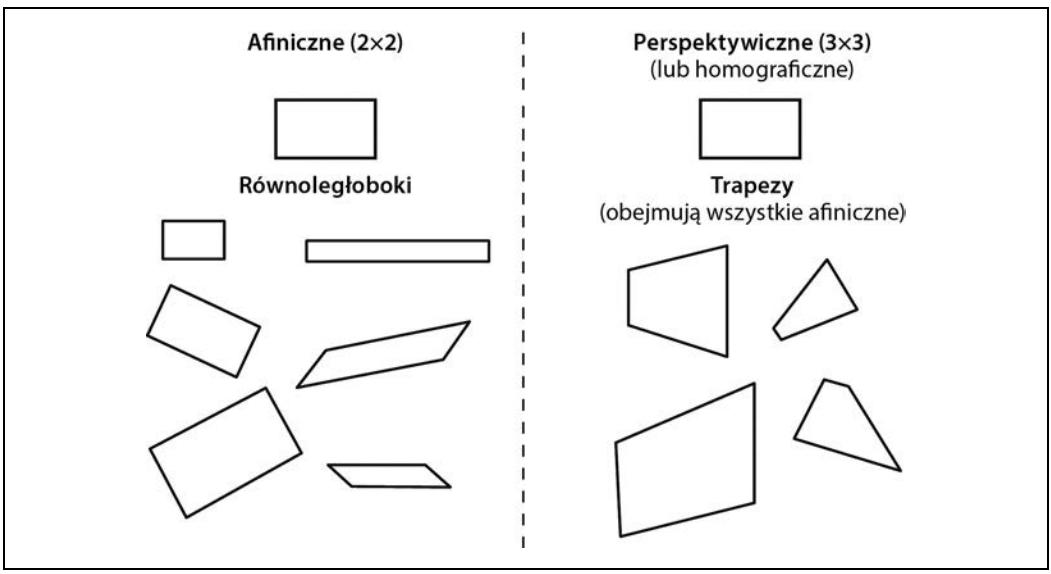
Nietrudno dostrzec, że wynik transformacji afinicznej $A \cdot X + B$ jest dokładnie taki sam jak wynik rozszerzenia wektora X do postaci X' i lewostronnego pomnożenia wektora X' przez T.

Przekształcenie afiniczne można sobie wyobrazić w następujący sposób: każdy równoległobok $ABCD$ na płaszczyźnie można odwzorować na każdy inny równoległobok $A'B'C'D'$ przez zastosowanie przekształcenia afinicznego. Jeżeli pola powierzchni tych równoległoboków są większe od zera, oznacza to, że dane przekształcenie afiniczne jest jednoznacznie zdefiniowane przez te dwa równoległoboki (ich trzy wierzchołki). Jeśli ktoś woli, to przekształcenie afiniczne może sobie wyobrażać jako rysowanie obrazu na dużym arkuszu gumy i zniekształcanie go poprzez rozciąganie i ściskanie[8] na rogach dla uzyskania różnych rodzajów równoległoboków.

[8] Równoległobok można nawet tak rozciągnąć, aby go odwrócić.

Jeśli danych jest kilka obrazów, z których każdy przedstawia ten sam obiekt w nieco inny sposób, to można obliczyć, jakie konkretnie przekształcenia zostały zastosowane w celu otrzymania tych różnych widoków. W takim przypadku często wykorzystuje się przekształcenia afiniczne zamiast perspektywicznych, ponieważ mają mniej parametrów, dzięki czemu łatwiej jest wykonać związane z nimi obliczenia. Wadą tego rozwiązania jest to, że prawdziwe zniekształcenia perspektywiczne można modelować wyłącznie na gruncie homografii[9], a przekształcenia afiniczne pozwalają uzyskać reprezentację niebędącą w stanie objąć wszystkich możliwych relacji między widokami. Z drugiej strony, jeśli zmiana perspektywy jest niewielka, zniekształcenie może mieć charakter afiniczny i wówczas przekształcenie afiniczne może być w zupełności wystarczające.

Za pomocą przekształcenia afinicznego można zamienić prostokąt w równoległobok. Figura może zostać ściśnięta, ale boki muszą pozostać równoległe. Istnieje też możliwość dokonania obrotu i zmiany rozmiaru. Przekształcenia perspektywiczne zapewniają bardziej elastyczne możliwości. Za ich pomocą można na przykład zamienić prostokąt w dowolny czworokąt. Rysunek 11.3 przedstawia różne schematyczne przykłady przekształceń afinicznych i perspektywicznych. Natomiast na rysunku 11.4 pokazane są podobne przykłady z użyciem zdjęcia.



Rysunek 11.3. Przekształcenia afiniczne i perspektywiczne

[9] Homografia to pojęcie matematyczne oznaczające odwzorowywanie punktów z jednej powierzchni na innej. W tym sensie jest ono zatem ogólniejsze niż znaczenie, w którym go tutaj używamy. W kontekście komputerowego rozpoznawania obrazu homografia prawie zawsze oznacza odwzorowanie punktów między dwiema płaszczyznami graficznymi, które korespondują z tym samym miejscem na płaskim obiekcie w prawdziwym świecie. Odwzorowanie takie może być reprezentowane przez pojedynczą ortogonalną macierz o wymiarach 3×3 (szerzej na ten temat piszemy w rozdziale 19.).

Przekształcenia afiniczne

Przekształcenia afiniczne stosuje się głównie w dwóch sytuacjach: gdy mamy obraz (albo obszar zainteresowania), który chcemy przekształcić, albo gdy mamy listę punktów, dla których chcemy obliczyć wynik przekształcenia. Koncepcyjnie oba te przypadki są do siebie podobne, ale różnią się pod względem praktycznej implementacji. Dlatego w bibliotece OpenCV dostępne są dwie funkcje — każda przeznaczona do użycia w innym przypadku.

cv::warpAffine() — przekształcenia afiniczne zbiorów gęstych

W pierwszym przypadku oczywistymi formatami wejściowymi i wyjściowymi są obrazy, a mniej oczywiste jest założenie funkcji, że piksele są *gęstą reprezentacją* opisywanego obrazu. Oznacza to, że funkcja zniekształcająca musi wykonywać interpolacje, tak aby obrazy wynikowe były gładkie i wyglądały naturalnie. Funkcja przekształcenia afinicznego do przekształceń struktur gęstych w OpenCV nazywa się `cv::warpAffine()`:

```
void cv::warpAffine(
  cv::InputArray src,                          // obraz wejściowy
  cv::OutputArray dst,                         // obraz wynikowy
  cv::InputArray M,                            // macierz przekształcenia 2×3
  cv::Size dsize,                              // docelowy rozmiar obrazu
  int flags = cv::INTER_LINEAR,                // interpolacja, odwrotna
  int borderMode = cv::BORDER_CONSTANT,        // ekstrapolacja pikseli
  const cv::Scalar& borderValue = cv::Scalar() // dla stałych krawędzi
);
```

Argumenty `src` i `dst` reprezentują odpowiednio tablicę wejściową i wynikową. Parametr `M` jest opisaną przez nas wcześniej macierzą o wymiarach 2×3, która określa rodzaj przekształcenia. Każdy element w tablicy docelowej jest obliczony na podstawie elementu z tablicy źródłowej pobranego z lokalizacji wyrażonej następującym wzorem:

$$dst(x, y) = src(M_{00}x + M_{01}y + M_{02}, M_{10}x + M_{11}y + M_{12})$$

Zasadniczo jednak prawa strona tego równania zwykle wskazuje lokalizację piksela, która nie jest liczbą całkowitą. W takim przypadku prawidłową wartość dla punktu $dst(x, y)$ należy znaleźć przez zastosowanie interpolacji. Kolejny argument, `flags`, wskazuje metodę interpolacji. Dostępne są metody opisane w tabeli 11.1, czyli te same co dla funkcji `cv::resize()`, oraz dodatkowo opcja `cv::WARP_INVERSE_MAP` (którą można dodać przy użyciu operatora logicznego LUB). Opcja ta służy do wykonywania odwrotnego przekształcenia z `dst` do `src`. Dwa ostatnie argumenty dotyczą ekstrapolacji krawędzi i mają takie same znaczenie jak podobne argumenty w operacjach splotu obrazów (opisanych w rozdziale 10.).

cv::getAffineTransform() — obliczanie macierzy przekształcenia afinicznego

W bibliotece OpenCV znajdują się dwie funkcje do generowania macierzy odwzorowania `M`. Pierwszej używa się, gdy dane są dwa obrazy, o których wiadomo, że łączy je relacja przekształcenia afinicznego, lub które chcemy w ten sposób aproksymować:

```
cv::Mat cv::getAffineTransform( // zwraca macierz o wymiarach 2×3
  const cv::Point2f* src,       // współrzędne *trzech* wierzchołków
  const cv::Point2f* dst        // współrzędne docelowe, trzy wierzchołki
);
```

W tej funkcji parametry `src` i `dst` są tablicami zawierającymi po trzy dwuwymiarowe (*x, y*) punkty. Wartością zwrotną jest tablica reprezentująca przekształcenie afiniczne obliczone na podstawie tych punktów.

Zasadniczo tablice punktów `src` i `dst` w funkcji `cv::getAffineTransform()` definiują dwa równoległoboki. Punkty w tablicy `src` zostaną odwzorowane przez funkcję `cv::warpAffine()`, przy użyciu otrzymanej macierzy `M`, na odpowiednie punkty w `dst`. Wszystkie pozostałe punkty zostaną pociągnięte za nimi. Po odwzorowaniu tych trzech niezależnych rogów odwzorowanie pozostałych punktów także będzie w pełni zdeterminowane.

Na listingu 11.1 przedstawiony jest kod źródłowy ilustrujący użycie tych funkcji w praktyce. W programie tym najpierw obliczamy parametry macierzowe funkcji `cv::warpAffine()`, tworząc dwie tablice po trzy elementy (reprezentujące rogi równoległoboków), a potem konwertując je na prawdziwą macierz przekształcenia za pomocą funkcji `cv::getAffineTransform()`. Następnie wykonujemy przekształcenie afiniczne i rotację obrazu. Dla naszej tablicy reprezentatywnych punktów w obrazie źródłowym, `srcTri[]`, pobieramy trzy punkty: `(0,0)`, `(0,`*`wysokość`*`-1)` oraz (*`szerokość`*`-1,0)`. Później określamy lokalizacje, w których te punkty zostaną odwzorowane w korespondującej tablicy `dstTri[]`.

Listing 11.1. Przekształcenie afiniczne

```
#include <opencv2/opencv.hpp>
#include <iostream>

using namespace std;

int main(int argc, char** argv) {

  if(argc != 2) {
    cout << "Przekształcenie afiniczne\nSposób użycia: " <<argv[0] <<" <imagename>\n" << endl;
    return -1;
  }

  cv::Mat src = cv::imread(argv[1],1);
  if( src.empty() ) { cout << "Nie udało się załadować " << argv[1] << endl; return -1; }

  cv::Point3f srcTri[] = {
    cv::Point2f(0,0),           // źródło, lewy górny
    cv::Point2f(src.cols-1, 0), // źródło, prawy górny
    cv::Point2f(0, src.rows-1)  // źródło, lewy dolny
  };

  cv::Point2f dstTri[] = {
    cv::Point2f(src.cols*0.f, src.rows*0.33f),   // cel, lewy górny
    cv::Point2f(src.cols*0.85f, src.rows*0.25f), // cel, prawy górny
    cv::Point2f(src.cols*0.15f, src.rows*0.7f)   // cel, lewy dolny
  };

  // OBLICZANIE MACIERZY PRZEKSZTAŁCENIA AFINICZNEGO
  //
  cv::Mat warp_mat = cv::getAffineTransform(srcTri, dstTri);
  cv::Mat dst, dst2;
  cv::warpAffine(
    src,
    dst,
    warp_mat,
    src.size(),
    cv::INTER_LINEAR,
```

```
      cv::BORDER_CONSTANT,
      cv::Scalar()
    );
    for( int i = 0; i < 3; ++i )
      cv::circle(dst, dstTri[i], 5, cv::Scalar(255, 0, 255), -1, cv::AA);

    cv::imshow("Test przekształcenia afinicznego", dst);
    cv::waitKey();

    for(int frame=0;;++frame) {

      // OBLICZANIE MACIERZY ROTACJI
      cv::Point2f center(src.cols*0.5f, src.rows*0.5f);
      double angle = frame*3 % 360, scale = (cos((angle - 60)* cv::PI/180) + 1.05)*0.8;

      cv::Mat rot_mat = cv::getRotationMatrix2D(center, angle, scale);
      cv::warpAffine(
        src,
        dst,
        rot_mat,
        src.size(),
        cv::INTER_LINEAR,
        cv::BORDER_CONSTANT,
        cv::Scalar()
      );
      cv::imshow("Rotated Image", dst);
      if(cv::waitKey(30) >= 0 )
        break;

    }

    return 0;
  }
```

Drugim sposobem obliczania macierzy odwzorowania `M` jest użycie funkcji `cv::getRotationMatrix2D()`, która oblicza macierz odwzorowania dla rotacji wokół dowolnego punktu w połączeniu z opcjonalnym skalowaniem. Jest to tylko jeden z możliwych rodzajów przekształcenia afinicznego, więc technika ta jest mniej ogólna niż metoda z użyciem funkcji `cv::getAffineTransform()`, ale stanowi ważny podzbiór, który ma alternatywną (i bardziej intuicyjną) reprezentację:

```
cv::Mat cv::getRotationMatrix2D( // zwraca macierz o wymiarach 2x3
  cv::Point2f center             // środek rotacji
  double angle,                  // kąt obrotu
  double scale                   // skalowanie po rotacji
);
```

Pierwszy argument, `center`, określa punkt środkowy rotacji. Dwa następne argumenty określają kąt obrotu i skalowanie. Funkcja ta zwraca macierz odwzorowania `M`, która (jak zawsze) ma wymiary 2×3 i jest typu zmiennoprzecinkowego.

Jeśli przyjmiemy definicje $\alpha = scale \cdot cos(angle)$ i $\beta = scale \cdot sin(angle)$, to macierz `M` obliczona przez niniejszą funkcję będzie miała następującą postać:

$$\begin{bmatrix} \alpha\beta(1-\alpha)\cdot center_x - \beta\cdot center_y \\ -\beta\alpha\beta\cdot center_x - (1-\alpha)\cdot center_y \end{bmatrix}$$

Łącząc obie metody obliczania macierzy odwzorowania, można na przykład wybrany obraz obrócić, przeskalować *i* przekształcić.

cv::transform() — przekształcenia afiniczne zbiorów rzadkich

Wiesz już, że do pracy ze strukturami gęstymi służy funkcja `cv::warpAffine()`. Natomiast do odwzorowań rzadkich (tzn. list indywidualnych punktów) najlepiej nadaje się funkcja `cv::transform()`. Może pamiętasz z rozdziału 5., że metoda ta ma następujący prototyp:

```
void cv::transform(
  cv::InputArray src,  // wejściowa tablica o wymiarach N×1 (Ds kanałów)
  cv::OutputArray dst, // wyjściowa tablica o wymiarach N×1 (Dd kanałów)
  cv::InputArray mtx   // macierz przekształcenia (Ds×Dd)
);
```

Generalnie `src` jest tablicą o wymiarach N×1 zawierającą D_s kanałów, gdzie N jest liczbą punktów do przekształcenia, a D_s jest wymiarem tych punktów źródłowych. Wyjściowa tablica `dst` będzie miała taki sam rozmiar, ale może mieć inną liczbę kanałów D_d. Macierz przekształcenia `mtx` ma wymiary $D_s \times D_d$ i jest stosowana do każdego elementu tablicy `src`, po czym wyniki zostają wstawione do tablicy `dst`.

Należy zauważyć, że funkcja `cv::transform()` działa na kanałowych indeksach każdego punktu w tablicy. W odniesieniu do aktualnego problemu zakładamy, że tablica ta jest w istocie dużym wektorem (N×1 lub 1×N) takich wielokanałowych obiektów. Powinniśmy pamiętać, że indeks, do którego odnosi się macierz przekształcenia, jest indeksem kanału, a nie „wektorowym" indeksem tej dużej tablicy.

W przypadku przekształceń, które są prostymi rotacjami w danej przestrzeni kanału, nasza macierz przekształcenia `mtx` będzie tylko macierzą o wymiarach 2×2 i można ją zastosować bezpośrednio do dwukanałowych indeksów tablicy `src`. W niektórych prostych przypadkach dotyczy to rotacji, rozciągania i zniekształcania. Zazwyczaj jednak do przeprowadzenia ogólnego przekształcenia afinicznego (w tym translacji i rotacji wokół dowolnego środka itd.) konieczne jest zwiększenie liczby kanałów w `src` do trzech, tak aby otrzymać bardziej typową macierz przekształcenia afinicznego o wymiarach 2×3. W tym przypadku wszystkie elementy w trzecim kanale muszą zostać ustawione na `1` (tzn. punkty muszą być podane we współrzędnych homogenicznych). Oczywiście tablica wyjściowa nadal będzie miała dwa kanały.

cv::invertAffineTransform() — odwrotne przekształcenie afiniczne

Mając dane przekształcenie afiniczne reprezentowane przez macierz o wymiarach 2×3, często trzeba wykonać operację odwrotną, w celu przywrócenia wszystkich przekształconych punktów na ich pierwotne miejsca. Można to zrobić za pomocą funkcji `cv::invertAffineTransform()`:

```
void cv::invertAffineTransform(
  cv::InputArray M,  // wejściowa macierz o wymiarach 2×3
  cv::OutputArray iM // wyjściowa macierz także o wymiarach 2×3
);
```

Funkcja ta pobiera macierz `M` o wymiarach 2×3 i zwraca inną macierz `iM` o wymiarach 2×3, która jest odwrotnością `M`. Funkcja `cv::invertAffineTransform()` tak naprawdę nie działa na żadnym obrazie, lecz jedynie dostarcza przekształcenie odwrotne. Mając macierz `iM`, można jej używać w taki sam sposób, jak używałoby się `M` w funkcjach `cv::warpAffine()` i `cv::transform()`.

Przekształcenie perspektywiczne

Aby skorzystać z większej elastyczności przekształceń perspektywicznych (zwanych też homografiami), potrzebujemy nowej funkcji, która pozwoli nam wyrazić tę szerszą klasę transformacji. Przede wszystkim musimy podkreślić, że choć rzutowanie perspektywiczne w całości definiuje pojedyncza macierz, nie jest to rodzaj przekształcenia liniowego. Wynika to z faktu, że ten rodzaj transformacji wymaga podzielenia przez ostatni wymiar (zazwyczaj Z — zobacz rozdział 19.) i w związku z tym traci po drodze jeden wymiar.

Podobnie jak w przypadku przekształceń afinicznych, operacje na obrazach (przekształcenia gęste) wykonuje się za pomocą innych funkcji niż operacje na zbiorach punktów (przekształcenia rzadkie).

cv::warpPerspective() — gęste przekształcenie perspektywiczne

Gęste przekształcenie perspektywiczne w OpenCV wykonuje się za pomocą funkcji analogicznej do funkcji gęstego przekształcenia afinicznego. Mówiąc konkretnie: prawie wszystkie argumenty funkcji `cv::warpPerspective()` pokrywają się z argumentami funkcji `cv::warpAffine()`. Drobnym, ale jakże znaczącym wyjątkiem jest to, że macierz odwzorowania musi mieć wymiary 3×3.

```
void cv::warpPerspective(
  cv::InputArray src,                          // obraz wejściowy
  cv::OutputArray dst,                         // obraz wynikowy
  cv::InputArray M,                            // macierz przekształcenia 3×3
  cv::Size dsize,                              // docelowy rozmiar obrazu
  int flags = cv::INTER_LINEAR,                // interpolacja, odwrotna
  int borderMode = cv::BORDER_CONSTANT,        // metoda ekstrapolacji
  const cv::Scalar& borderValue = cv::Scalar() // dla stałych krawędzi
);
```

Każdy element w tablicy docelowej jest obliczany na podstawie elementu z tablicy źródłowej o lokalizacji wyrażonej za pomocą następującego wzoru:

$$dst(x,y) = src\left(\frac{M_{00}x + M_{01}y + M_{02}}{M_{20}x + M_{21}y + M_{22}}, \frac{M_{10}x + M_{11}y + M_{12}}{M_{20}x + M_{21}y + M_{22}}\right)$$

Tak jak w przypadku przekształcenia afinicznego, lokalizacja wskazywana przez prawą stronę tego równania z reguły nie jest całkowitoliczbowa. Argument `flags` określa metodę interpolacji i może przyjmować takie same wartości jak analogiczny argument funkcji `cv::warpAffine()`.

cv::getPerspectiveTransform() — obliczanie macierzy rzutowania perspektywicznego

Tak jak w przypadku przekształcenia afinicznego, dla którego obliczaliśmy macierz odwzorowania w poprzednim przykładzie kodu, macierz przekształcenia możemy obliczyć przy użyciu specjalnej funkcji pobierającej listę punktów:

```
cv::Mat cv::getPerspectiveTransform( // zwraca macierz o wymiarach 3×3
  const cv::Point2f* src,            // współrzędne *czterech* wierzchołków
  const cv::Point2f* dst             // współrzędne docelowe, cztery wierzchołki
);
```

Teraz argumenty src i dst reprezentują tablice zawierające po cztery (nie trzy) punkty, dzięki czemu możemy niezależnie kontrolować, jak rogi (zazwyczaj) prostokąta z src mają być odwzorowane w (zazwyczaj) romb w dst. Nasze przekształcenie jest całkowicie zdefiniowane przez wyznaczone miejsca docelowe czterech punktów źródłowych. Jak wspominaliśmy wcześniej, w przypadku przekształceń perspektywicznych wartością zwrotną będzie tablica o wymiarach 3×3 (przykładowy kod przedstawiliśmy na listingu 11.2). Nie licząc tego, że macierz ma wymiary 3×3, i zamiany trzech punktów kontrolnych na cztery, przekształcenie perspektywiczne jest analogiczne do opisanego wcześniej przekształcenia afinicznego.

Listing 11.2. Przykład przekształcenia perspektywicznego

```
#include <opencv2/opencv.hpp>
#include <iostream>

using namespace std;

int main(int argc, char** argv) {

  if(argc != 2) {
    cout << "Zniekształcenie perspektywiczne\nSposób użycia: " <<argv[0] <<" <imagename>\n" << endl;
    return -1;
  }

  Mat src = cv::imread(argv[1],1);
  if( src.empty() ) { cout << "Nie udało się załadować " << argv[1] << endl; return -1; }

  cv::Point2f srcQuad[] = {
    cv::Point2f(0, 0),                    // źródło, lewy górny
    cv::Point2f(src.cols-1, 0),           // źródło, prawy górny
    cv::Point2f(src.cols-1, src.rows-1), // źródło, prawy dolny
    cv::Point2f(0, src.rows-1)            // źródło, lewy dolny
  };

  cv::Point2f dstQuad[] = {
    cv::Point2f(src.cols*0.05f, src.rows*0.33f),
    cv::Point2f(src.cols*0.9f, src.rows*0.25f),
    cv::Point2f(src.cols*0.8f, src.rows*0.9f),
    cv::Point2f(src.cols*0.2f, src.rows*0.7f)
  };

  // OBLICZANIE MACIERZY PERSPEKTYWY
  //
  cv::Mat warp_mat = cv::getPerspectiveTransform(srcQuad, dstQuad);
  cv::Mat dst;
  cv::warpPerspective(src, dst, warp_mat, src.size(), cv::INTER_LINEAR,
  cv::BORDER_CONSTANT, cv::Scalar());

  for( int i = 0; i < 4; i++ )
    cv::circle(dst, dstQuad[i], 5, cv::Scalar(255, 0, 255), -1, cv::AA);

  cv::imshow("Test przekształcenia perspektywicznego", dst);
  cv::waitKey();
  return 0;
}
```

cv::perspectiveTransform() — rzadkie przekształcenia perspektywiczne

Funkcja `cv::perspectiveTransform()` to specjalna procedura wykonująca przekształcenia perspektywiczne na listach punktów. Funkcja `cv::transform()` jest ograniczona do operacji liniowych, przez co nie jest w stanie prawidłowo wykonać przekształcenia perspektywicznego. Wynika to z faktu, że tego typu transformacje wymagają dzielenia przez trzecią współrzędną reprezentacji homogenicznej ($x = f \cdot X/Z$, $y = f \cdot Y/Z$). Zawsze jednak możemy skorzystać ze specjalnej funkcji `cv::perspective`
`↪Transform()`:

```
void cv::perspectiveTransform(
  cv::InputArray src,  // wejściowa tablica o wymiarach N×1 (2 lub 3 kanały)
  cv::OutputArray dst, // wyjściowa tablica o wymiarach N×1 (2 lub 3 kanały)
  cv::InputArray mtx   // macierz przekształcenia (3×3 lub 4×4)
);
```

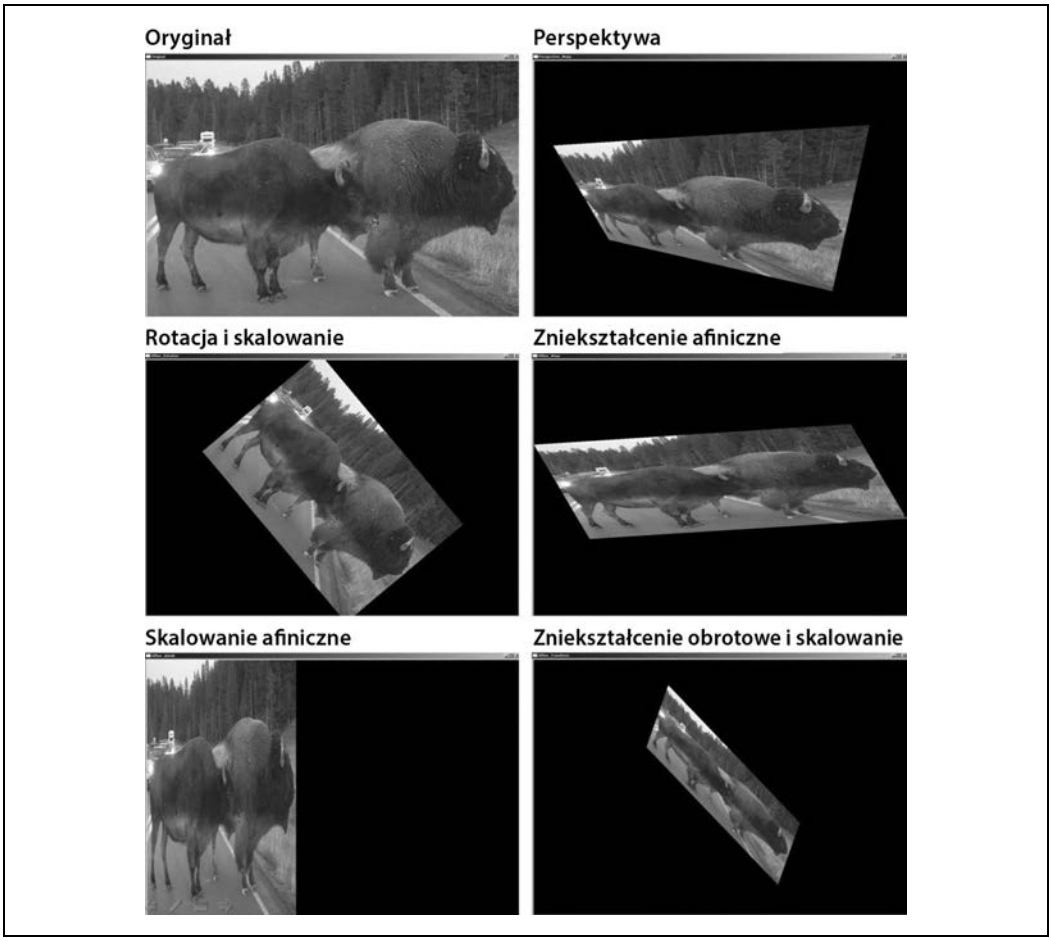
Jak zwykle argumenty `src` i `dst` reprezentują odpowiednio tablicę punktów źródłowych do przekształcenia i tablicę punktów powstałych w wyniku tej operacji. Tablice te powinny zawierać po dwa lub trzy kanały. Macierz `mtx` może mieć wymiary 3×3 lub 4×4. Jeśli jest to macierz o wymiarach 3×3, to funkcja wykonuje rzutowanie z dwóch wymiarów na dwa. Jeżeli macierz ta ma wymiary 4×4, to zostaje wykonane rzutowanie z trzech wymiarów na trzy.

W aktualnym kontekście przekształcamy zbiór punktów obrazu na inny zbiór punktów obrazu, co wygląda na odwzorowywanie z dwóch wymiarów na dwa wymiary. Nie jest to jednak do końca prawdą, ponieważ przekształcenie perspektywiczne w istocie polega na odwzorowywaniu punktów z dwuwymiarowej płaszczyzny osadzonej w przestrzeni trójwymiarowej na (inną) podpłaszczyznę dwuwymiarową. Można powiedzieć, że jest to taka sama czynność, jaką wykonuje kamera. (Bardziej szczegółowo omawiamy ten temat w dalszych rozdziałach, przy okazji omawiania kamer). Kamera pobiera punkty w trzech wymiarach i odwzorowuje je na dwa wymiary przetwornika kamery. O to zasadniczo chodzi w przypadku, gdy punkty źródłowe są podawane we współrzędnych homogenicznych. Dodajemy wymiar do tych punktów, wprowadzając wymiar Z i ustawiając wszystkie wartości w tym wymiarze na 1. Przekształcenie rzutowania jest wówczas rzutowaniem z powrotem z tej przestrzeni do dwuwymiarowej przestrzeni na wyjściu. W ten pokrętny sposób wyjaśniamy, dlaczego przy odwzorowywaniu punktów z jednego obrazu do punktów w innym obrazie potrzebna jest macierz o wymiarach 3×3.

Rysunek 11.4 pokazuje wyniki przekształceń afinicznych i perspektywicznych przedstawionych na listingach 11.1 i 11.2. W przykładach tych transformacji poddaliśmy prawdziwe obrazy. Można je porównać z prostymi schematami pokazanymi na rysunku 11.3.

Ogólne odwzorowania

Omawiane do tej pory przekształcenia afiniczne i perspektywiczne to specjalne przypadki ogólniejszego procesu. Zasadniczo obie te transformacje bazują na tej samej zasadzie: pobierają piksele z jednego miejsca w obrazie źródłowym i odwzorowują je w innym miejscu w obrazie docelowym. Istnieją też inne operacje o takiej samej strukturze. W tej sekcji opisujemy kilka kolejnych przekształceń tego typu, a następnie przedstawiamy dostępne w OpenCV narzędzia do implementacji własnych ogólnych przekształceń opartych na odwzorowaniach.



Rysunek 11.4. Odwzorowania perspektywiczne i afiniczne obrazu

Odwzorowania biegunowe

W rozdziale 5. krótko nadmieniliśmy o dwóch funkcjach, `cv::cartToPolar()` i `cv::polarToCart()`, za pomocą których można konwertować tablice punktów kartezjańskich *x, y* na tablice punktów biegunowych r–θ (lub odwrotnie).

Między funkcjami odwzorowania biegunowego oraz funkcjami przekształceń perspektywicznych i afinicznych występuje drobna niespójność stylistyczna. Funkcje odwzorowania biegunowego do reprezentacji wektorów dwuwymiarowych wymagają podania par tablic jednokanałowych, a nie dwukanałowych. Źródłem tej różnicy jest sposób, w jaki te różne funkcje są tradycyjnie używane, i nie ma to związku z żadnymi wewnętrznymi mechanizmami ich działania.

Choć funkcje `cv::cartToPolar()` i `cv::polarToCart()` są wykorzystywane przez bardziej skomplikowane procedury (np. opisana dalej `cv::logPolar()`), mogą być też przydatne same w sobie.

cv::cartToPolar() — konwersja współrzędnych kartezjańskich na biegunowe

Do zamiany współrzędnych kartezjańskich na biegunowe służy funkcja `cv::cartToPolar()`:

```
void cv::cartToPolar(
  cv::InputArray x,           // wejściowa jednokanałowa tablica współrzędnych x
  cv::InputArray y,           // wejściowa jednokanałowa tablica współrzędnych y
  cv::OutputArray magnitude,  // wyjściowa jednokanałowa tablica promieni wodzących
  cv::OutputArray angle,      // wyjściowa jednokanałowa tablica kątów
  bool angleInDegrees = false // true oznacza stopnie, false — radiany
);
```

Dwa pierwsze argumenty, `x` i `y`, są tablicami jednokanałowymi. Jednak nie reprezentują one tylko list punktów, lecz **pole wektorowe**[10], w którym składnik x danego punktu reprezentuje odpowiednia wartość z tablicy `x`, a składnik y — odpowiednia wartość z tablicy `y`. Natomiast wyniki tej funkcji znajdują się w tablicach `magnitude` i `angle`. Każdy punkt w `magnitude` reprezentuje długość wektora w tym punkcie z tablic `x` i `y`, a każdy punkt w `angle` reprezentuje kąt skierowany tego wektora. Kąty zapisane w `angles` domyślnie są podawane w radianach, to znaczy w przedziale $[0, 2\pi)$. Jeżeli jednak argumentowi `angleInDegrees` zostanie przypisana wartość `true`, to kąty w `angle` będą zapisywane w stopniach w przedziale [0, 360). Ponadto należy zauważyć, że kąty są obliczane (w przybliżeniu) według wzoru `atan2(y,x)`, w związku z czym kąt 0 odpowiada wektorowi skierowanemu w kierunku $\hat{x}$.

Jeśli zastanawiasz się, gdzie można by zastosować tę funkcję, wyobraź sobie, że obliczyłeś już pochodne x i y obrazu za pomocą funkcji `cv::Sobel()` albo przy użyciu funkcji splotu za pośrednictwem funkcji `cv::DFT()` lub `cv::filter2D()`. Jeżeli pochodne x zostały zapisane w obrazie `dx_img`, a pochodne y — w obrazie `dy_img`, to można sporządzić histogram wykrywania kątów krawędzi, to znaczy można zebrać wszystkie kąty, dla których moduł lub moc piksela na krawędzi przekracza pewien określony próg. Aby wykonać takie obliczenia, najpierw trzeba utworzyć dwa nowe obrazy docelowe (i na przykład nadać im nazwy `img_mag` i `img_angle`) na pochodne kierunkowe, a następnie posłużyć się funkcją `cvCartToPolar( dx_img, dy_img, img_mag, img_angle, 1 )`. Później moglibyśmy sporządzić histogram na podstawie danych z `img_angle`, które pobieralibyśmy pod warunkiem, że odpowiedni „piksel” w `img_mag` jest powyżej ustalonego przez nas progu.

Funkcje rozpoznawania obrazu i graficzne omawiamy w rozdziale 22. Opisany tu proces stanowi podstawę obliczania ważnej składowej rozpoznawania obiektów **— HOG** (ang. *histogram of oriented gradients*).

cv::polarToCart() — konwersja współrzędnych biegunowych na kartezjańskie

Funkcja `cv::cartToPolar()` zamienia współrzędne biegunowe na kartezjańskie.

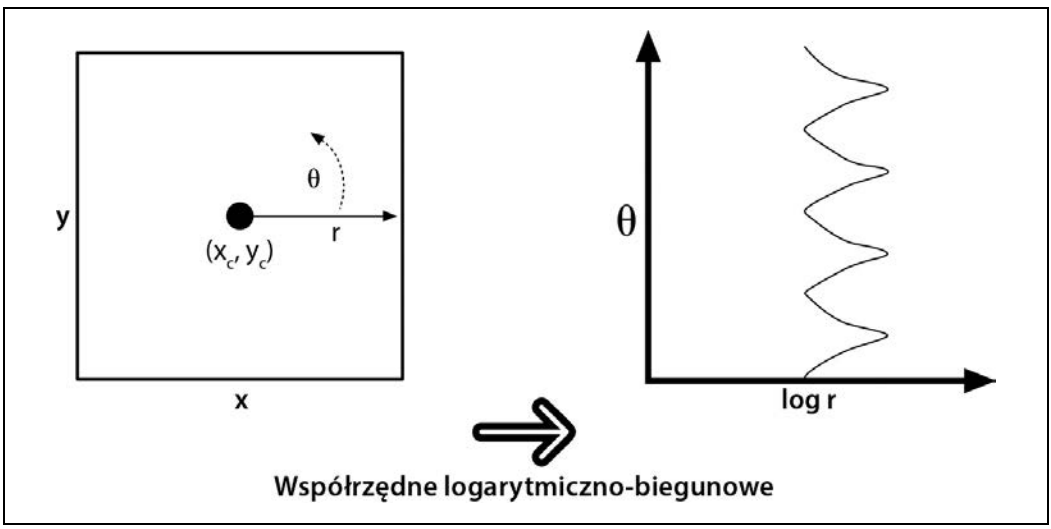
```
void cv::polarToCart(
  cv::InputArray magnitude,   // wyjściowa jednokanałowa tablica promieni wodzących
  cv::InputArray angle,       // wyjściowa jednokanałowa tablica kątów
  cv::OutputArray x,          // wejściowa jednokanałowa tablica x
  cv::OutputArray y,          // wejściowa jednokanałowa tablica y
  bool angleInDegrees = false // true oznacza stopnie, false — radiany
);
```

[10] Jeśli ktoś nie jest obeznany z pojęciem pola wektorowego, to może je sobie wyobrazić jako dwuskładnikowy wektor powiązany z każdym punktem „obrazu”.

Operacja zamiany współrzędnych biegunowych na kartezjańskie również jest często przydatna. Przyjmuje praktycznie takie same argumenty jak `cv::cartToPolar()`, tylko że tym razem tablice odległości od bieguna i kątów są danymi wejściowymi, a `x` i `y` — wyjściowymi.

Współrzędne logarytmiczno-biegunowe

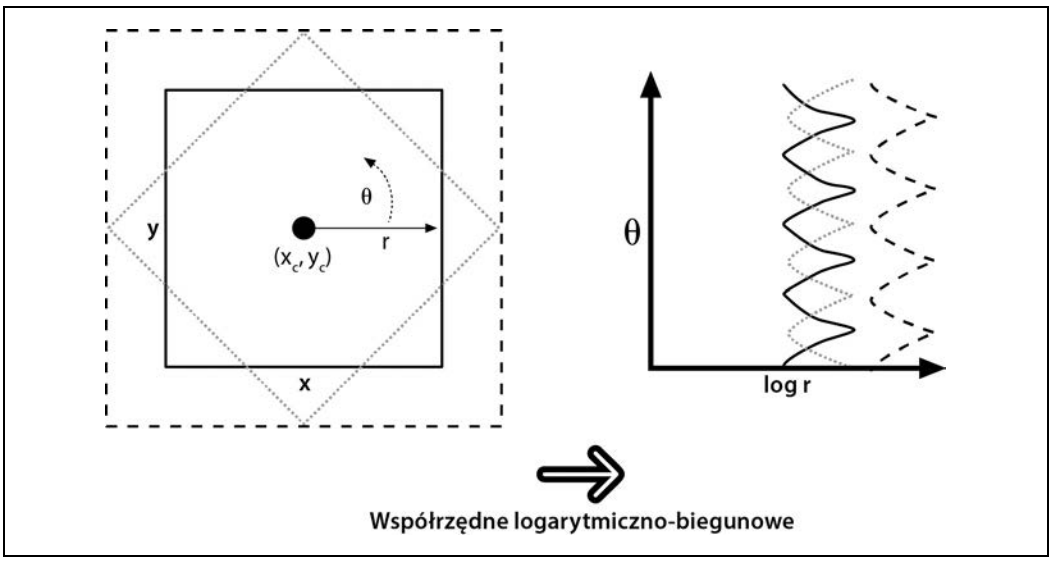
W odniesieniu do obrazów dwuwymiarowych przekształcenie logarytmiczno-biegunowe [Schwartz80] polega na zamianie współrzędnych kartezjańskich na logarytmiczno-biegunowe: $(x, y) \leftrightarrow re^{i\theta}$, gdzie $r = \sqrt{x^2 + y^2}$, a $\theta = atan2(y, x)$. Aby następnie wyodrębnić współrzędne biegunowe do przestrzeni (ρ, θ) względnej w odniesieniu do pewnego punktu centralnego (x_c, y_c), należy obliczyć logarytm w taki sposób, że $\rho = \log\left(\sqrt{(x - x_c)^2 + (y - y_c)^2}\right)$ oraz $\theta = atan2(y - y_c, x - x_c)$. Na potrzeby obróbki obrazów — gdy trzeba „upchnąć" interesujące nas dane w dostępnej pamięci graficznej — do ρ zazwyczaj stosuje się współczynnik skalowania m. Po lewej na rysunku 11.5 widać kwadratowy obiekt, a po prawej — jego wersję zakodowaną w przestrzeni logarytmiczno-biegunowej.



Rysunek 11.5. Przekształcenie logarytmiczno-biegunowe odwzorowuje współrzędne (x,y) na (log(r),0). W tym przykładzie pokazaliśmy kwadrat przedstawiony w układzie współrzędnych logarytmiczno-biegunowych

Można się zastanawiać, po co w ogóle coś takiego robić. Konwersja na współrzędne logarytmiczno-biegunowe ma jednak korzenie w obserwacji ludzkiego wzroku. W oku znajduje się niewielki, ale bardzo gęsty obszar fotoreceptorów (**dołek środkowy siatkówki oka**), poza którym gęstość receptorów raptownie spada (w tempie wykładniczym). Spójrz na dowolny punkt na ścianie i w polu widzenia w odległości na wyciągnięcie ręki ustaw palec. Następnie cały czas patrząc w ten wybrany punkt na ścianie, powoli przesuwaj palec w bok. Zwróć uwagę, jak szybko obraz palca staje się nieostry podczas przesuwania go sprzed zasięgu dołka środkowego siatkówki. Ta anatomiczna struktura ma też pewne interesujące właściwości matematyczne (ich opis wykracza poza zakres tematyczny tej książki) dotyczące zachowywania kątów przecięcia linii.

Dla nas najważniejsze jest to, że przy użyciu przekształcenia na współrzędne logarytmiczno-biegunowe można stworzyć dwuwymiarowe niezmienne reprezentacje widoków obiektu przez przesunięcie środka ciężkości przekształconego obrazu do ustalonego punktu na płaszczyźnie logarytmiczno-biegunowej. Na rysunku 11.6 po lewej pokazane są trzy figury, które chcemy rozpoznać jako kwadraty. Trudność polega na tym, że każda z nich wygląda całkiem inaczej od pozostałych. Jedna jest znacznie większa od pozostałych dwóch, a inna jest przekręcona. Po prawej stronie ilustracji widać wynik zamiany na współrzędne logarytmiczno-biegunowe. Zwróć uwagę, że różnice w rozmiarze na płaszczyźnie (x,y) zostały zamienione na przesunięcia względem osi $\log(r)$ na płaszczyźnie logarytmiczno-biegunowej, a różnice w rotacji skutkują przesunięciami względem osi θ. Jeśli weźmiemy przekształcony środek każdego kwadratu na płaszczyźnie logarytmiczno-biegunowej i przeniesiemy go do jakiegoś ustalonego punktu, to wszystkie kwadraty na tej płaszczyźnie okażą się identyczne. Oznacza to rodzaj niezmienności w odniesieniu do rotacji i skalowania w dwóch wymiarach[11].



Rysunek 11.6. Logarytmiczno-biegunowe przekształcenie obróconych i przeskalowanych kwadratów. Rozmiar zamienia się na przesunięcie względem osi log(r), a rotacja — na przesunięcie względem osi θ

[11] W rozdziale 22. szczegółowo opisujemy techniki rozpoznawania. Na razie wystarczy zauważyć, że wykonywanie transformacji na współrzędne logarytmiczno-biegunowe całego obiektu nie byłoby dobrym pomysłem, ponieważ przekształcenia takie są wrażliwe na dokładne położenie punktów centralnych. W przypadku rozpoznawania obiektów prawdopodobnie lepszym rozwiązaniem byłoby wykrycie zbiorów kluczowych punktów (np. rogów albo zarysów), przycięcie tych widoków, a następnie wykorzystanie środków tych punktów kluczowych jako środków logarytmiczno-biegunowych. Przy użyciu tych lokalnych przekształceń logarytmiczno-biegunowych można następnie stworzyć lokalne właściwości, które są (częściowo) niezmienne w odniesieniu do rotacji i skalowania i które można powiązać z obiektem wizualnym.

cv::logPolar()

Funkcja OpenCV do wykonywania przekształceń logarytmiczno-biegunowych to `cv::logPolar()`:

```
void cv::logPolar(
  cv::InputArray src,           // obraz wejściowy
  cv::OutputArray dst,          // obraz wynikowy
  cv::Point2f center,           // środek przekształcenia
  double m,                     // współczynnik skalowania
  int flags = cv::INTER_LINEAR // tryby interpolacji i wypełniania
      | cv::WARP_FILL_OUTLIERS
);
```

Argumenty `src` i `dst` to typowe obrazy wejściowy i wyjściowy. Parametr `center` określa punkt centralny (x_c, y_c) przekształcenia logarytmiczno-biegunowego. Parametr `m` reprezentuje współczynnik skalowania, który należy zdefiniować, aby interesujące nas właściwości zdominowały dostępny obszar obrazu. Parametr `flags` służy do wybierania metody interpolacji. Do wyboru są standardowe metody interpolacji dostępne w bibliotece OpenCV (tabela 11.1). Różne metody można łączyć z jednym lub dwoma następującymi znacznikami: `CV::WARP_FILL_OUTLIERS` (wypełnienie punktów, które w przeciwnym razie byłyby niezdefiniowane) oraz `CV::WARP_INVERSE_MAP` (obliczanie odwrotnego odwzorowania ze współrzędnych logarytmiczno-biegunowych na kartezjańskie).

Na listingu 11.3 przedstawiony jest przykładowy program z użyciem współrzędnych logarytmiczno-biegunowych, w którym pokazujemy transformacje w obie strony. Rysunek 11.7 przedstawia efekty zastosowania tych operacji do zdjęcia.

Rysunek 11.7. Przykład zastosowania przekształcenia na zdjęciu łosia, ze środkiem transformacji w białym punkcie po lewej stronie; po prawej widać efekt przekształcenia

Listing 11.3. Przykład przekształcenia logarytmiczno-biegunowego

```
#include <opencv2/opencv.hpp>
#include <iostream>

using namespace std;
```

```
int main(int argc, char** argv) {

  if(argc != 3) {
    cout << "LogPolar\nSposób użycia: " <<argv[0] <<" <imagename> <M value>\n" <<"<M value>~30
    ↪zazwyczaj wystarcza\n";
    return -1;
  }

  cv::Mat src = cv::imread(argv[1],1);

  if( src.empty() ) { cout << "Nie udało się załadować " << argv[1] << endl; return -1; }

  double M = atof(argv[2]);
  cv::Mat dst(src.size(), src.type()), src2(src.size(), src.type());

  cv::logPolar(
    src,
    dst,
    cv::Point2f(src.cols*0.5f, src.rows*0.5f),
    M,
    cv::INTER_LINEAR | cv::WARP_FILL_OUTLIERS
  );
  cv::logPolar(
    dst,
    src2,
    cv::Point2f(src.cols*0.5f, src.rows*0.5f),
    M,
    cv::INTER_LINEAR | cv::WARP_INVERSE_MAP
  );
  cv::imshow( "log-polar", dst );
  cv::imshow( "inverse log-polar", src2 );

  cv::waitKey();

  return 0;
}
```

Odwzorowania arbitralne

Czasami interpolację wykonujemy programowo, to znaczy mamy pewien algorytm odwzorowania i go po prostu stosujemy. Zdarzają się jednak przypadki, gdy wolimy odwzorowanie przeprowadzić samodzielnie. Zanim zaczniemy zgłębiać tajniki metod obliczania (i stosowania) takich odwzorowań, najpierw przyjrzymy się funkcji stosowania odwzorowań, na której bazują wszystkie te pozostałe metody.

Jednym z typowych zastosowań funkcji `cv::remap()` jest rektyfikacja (korekta zniekształceń) obrazów skalibrowanych i stereo. W rozdziałach 18. i 19. opisujemy funkcje konwertujące obliczone zniekształcenia spowodowane przez kamerę na parametry `mapx` i `mapy`.

Krótko mówiąc: funkcja OpenCV, której nam teraz trzeba, to `cv:remap()`.

cv::remap() — ogólne odwzorowanie obrazu

```
void cv::remap(
  cv::InputArray src,                    // obraz wejściowy
  cv::OutputArray dst,                   // obraz wynikowy
  cv::InputArray map1,                   // docelowy x dla piksela źródłowego
  cv::InputArray map2,                   // docelowy y dla piksela źródłowego
```

```
    int interpolation = cv::INTER_LINEAR,          // interpolacja, odwrotność
    int borderMode = cv::BORDER_CONSTANT,          // metoda ekstrapolacji
    const cv::Scalar& borderValue = cv::Scalar() // dla stałych krawędzi
);
```

Dwa pierwsze argumenty tej funkcji to odpowiednio obraz wejściowy i wynikowy. Następne dwa argumenty, `map1` i `map2`, wskazują docelowe wartości współrzędnych *x* i *y* wyznaczających lokalizację danego piksela. Przy ich użyciu definiuje się własne ogólne odwzorowanie. Tablice te powinny mieć taki sam rozmiar jak tablica źródłowa i wynikowa oraz muszą być typu `CV::S16C2`, `CV::F32C1` lub `CV::F32C2`. Odwzorowania niecałkowitoliczbowe również są dozwolone. W takim przypadku funkcja `cv::remap()` automatycznie wykonuje obliczenia interpolacyjne.

Następny argument to `interpolation`. Jego wartość wskazuje funkcji `cv::remap()`, jak dokładnie ma przeprowadzić interpolację. Dostępne są wszystkie metody wymienione w tabeli 11.1 z wyjątkiem `cv::INTER_AREA`, która nie została zaimplementowana dla funkcji `cv::remap()`.

Renowacja obrazów

Wiele obrazów zawiera różne usterki. Podczas wykonywania zdjęć obiektyw może być zabrudzony albo zachlapany wodą, a stare zdjęcia bywają podrapane lub w inny sposób uszkodzone. Ratunkiem w takich sytuacjach może być zastosowanie techniki rekonstrukcji (ang. *inpainting*) [Telea04], czyli metody retuszu polegającej na pobieraniu informacji o kolorze i teksturze na krawędziach zniszczonego obszaru, a następnie powielaniu i mieszaniu takich samych pikseli w tym obszarze. Na rysunku 11.8 pokazany jest przykład, w którym technika ta została zastosowana do pozbycia się napisu ze zdjęcia.

Rysunek 11.8. Po lewej: obraz zniszczony przez napis; po prawej: obraz wyretuszowany za pomocą techniki inpaintingu

Inpainting

Metoda inpaintingu sprawdza się tylko wówczas, gdy zniszczony obszar nie jest zbyt gruby, to znaczy na jego obwodzie pozostało wystarczająco informacji na temat tekstury i koloru, aby można było dokonać uzupełnienia brakującej części. Rysunek 11.9 pokazuje, co się dzieje, gdy uszkodzony obszar jest za duży.

Rysunek 11.9. Technika inpaintingu nie jest czarodziejską różdżką do odzyskiwania całkowicie utraconych tekstur. Na zdjęciu po lewej całkowicie wymazano gniazdo nasienne pomarańczy, przez co na zdjęciu po prawej owoc został uzupełniony teksturą miąższu

Oto prototyp funkcji `cv::inpaint()`:

```
void cv::inpaint(
  cv::InputArray src,          // obraz wejściowy, 8 bitów, 1 lub 3 kanały
  cv::InputArray inpaintMask, // 8 bitów, 1 kanał, niezerowe piksele
  cv::OutputArray dst,         // obraz wynikowy
  double inpaintRadius,        // promień obszaru do uwzględnienia wokół piksela
  int flags                    // może mieć wartość NS lub TELEA
);
```

Argument `src` reprezentuje 8-bitowy jednokanałowy obraz w skali szarości lub trzykanałowy obraz kolorowy, który ma być poddany operacji retuszu. Argument `inpaintMask` to 8-bitowy jednokanałowy obraz o takim samym rozmiarze jak `src`, w którym zniszczone obszary (np. napis widoczny po lewej stronie na rysunku 11.8) zostały zaznaczone niezerowymi pikselami. Wszystkie pozostałe piksele w tym obrazie są ustawione na `0`. Wynik zostanie zapisany w obrazie `dst`, który musi mieć taki sam rozmiar i taką samą liczbę kanałów jak `src`. Argument `inpaintRadius` określa promień obszaru wokół każdego „wmalowywanego" piksela, jaki ma zostać wliczony w jego kolor wynikowy. Jak widać na rysunku 11.9, piksele wewnętrzne znajdujące się w odpowiednio dużym obszarze mogą w całości przyjmować kolory od innych „wmalowanych" pikseli, które znajdują się bliżej krawędzi. Prawie zawsze argument ten powinien mieć niewielką wartość, rzędu `3`, ponieważ zbyt duży promień spowoduje powstanie widocznego zamazania. I wreszcie za pomocą parametru `flags` można eksperymentować z dwiema metodami inpaintingu: `cv::INPAINT_NS` (metoda Naviera-Stokesa) i `cv::INPAINT_TELEA` (autorem tej metody jest A. Telea).

Usuwanie szumów

Kolejnym poważnym problemem dotyczącym obrazów jest szum. Jego głównym źródłem w wielu przypadkach są niedostatki oświetlenia przy wykonywaniu zdjęcia. W słabym oświetleniu aparat musi zastosować wzmocnienie sygnału, w wyniku czego dochodzi także do wzmocnienia szumu. Niedoskonałości tego typu mają zwykle charakter rozproszonych punktów o zbyt dużej lub zbyt małej jasności, choć zdarzają się też odbarwienia na kolorowych obrazach.

W OpenCV zaimplementowany jest algorytm usuwania szumów o nazwie **FNLMD** (ang. *Fast Non-Local Means Denoising*), który powstał na podstawie prac Antoniego Buadesa, Bartomeu Colla i Jeana-Michela Morela [Buades05]. Podczas gdy działanie prostych algorytmów odszumiających z reguły opiera się na uśrednianiu wartości indywidualnych pikseli na podstawie wartości pikseli sąsiednich, centralną koncepcją algorytmu FNLMD jest wyszukiwanie *podobnych pikseli* w innych częściach obrazu i uśrednianie wartości w odniesieniu do nich. Podobne do siebie piksele to nie takie, które mają zbliżone kolor lub intensywność, lecz takie, które znajdują się w podobnym otoczeniu. Pomysł ten opiera się na spostrzeżeniu, że większość obrazów zawiera powtarzalne struktury, więc nawet jeśli dany piksel zostanie zniszczony przez szum, można go odzyskać na podstawie innych pikseli, które uniknęły tego losu.

Identyfikacja podobnych pikseli odbywa się przy użyciu okienka $B(p, s)$ ustawionego centralnie nad pikselem p i mającego rozmiar s. Po ustawieniu takiego okna nad punktem, który chcemy zmienić, możemy porównać jego zawartość z zawartością analogicznego okna ustawionego nad innym pikselem q. Kwadrat odległości między punktami $B(p,s)$ i $B(q,s)$ obliczamy według następującego wzoru:

$$d^2\left(B(p,s),B(q,s)\right)=\frac{1}{3(2s+1)}\sum_{c=1}^{3}\sum_{j\varepsilon B(0,s)}\left(I_c(p+j)-I_c(q+j)\right)^2$$

We wzorze tym c jest indeksem koloru, $I_c(p)$ reprezentuje intensywność obrazu w kanale c w punkcie p, a suma nad j dotyczy elementów plamy. Na podstawie kwadratu odległości przypisuje się każdemu pikselowi obrazu wagę względem piksela, który jest właśnie modyfikowany. Wagę tę obliczamy według następującego wzoru:

$$w(p,q)=e^{-\frac{\max\left(d^2-2\sigma^2,0.0\right)}{h^2}}$$

W tej funkcji symbol σ oznacza spodziewaną wartość odchylenia standardowego szumów w obrazie (w jednostkach intensywności), natomiast h reprezentuje ogólny parametr filtrowania określający, jak szybko plamy będą stawać się nieistotne wraz ze wzrostem ich kwadratu odległości od modyfikowanej plamy. Generalnie rzecz biorąc, wzrost wartości h pociąga za sobą wzrost ilości usuniętego szumu, ale kosztem utraty pewnych szczegółów obrazu. Mniejsza wartość h pozwala lepiej zachować obraz, lecz pozostawia też więcej szumu.

Zwykle korzyści z uwzględniania plam bardzo oddalonych (w pikselach) od aktualizowanego piksela są coraz mniejsze, ponieważ liczba takich plam wzrasta kwadratowo wraz z dozwolonym dystansem. Dlatego normalnie definiuje się cały obszar, zwany **oknem wyszukiwania**, i w aktualizacji biorą udział tylko plamy znajdujące się w tym oknie. Wówczas aktualizacja bieżącego piksela jest obliczana

na podstawie zwykłej średniej ważonej wszystkich pozostałych pikseli w oknie wyszukiwania z wykorzystaniem zanikających wykładniczo wag[12]. Dlatego algorytm ma w nazwie człon „nielokalny". Plamy uwzględniane w retuszu danego piksela są tylko luźno skorelowane z lokalizacją tego piksela.

Implementacja algorytmu FNLMD w bibliotece OpenCV obejmuje kilka różnych funkcji, z których każda ma zastosowanie w nieco innej sytuacji.

Podstawowy algorytm FNLMD — funkcja cv::fastNlMeansDenoising()

```
void cv::fastNlMeansDenoising(
  cv::InputArray src,          // obraz wejściowy
  cv::OutputArray dst,         // obraz wynikowy
  float h = 3,                 // parametr rozkładu wagi
  int templateWindowSize = 7, // rozmiar plam używanych do porównywania
  int searchWindowSize = 21    // maksymalna odległość plamy do uwzględnienia
);
```

Pierwsza z czterech funkcji implementujących algorytm FNLMD to `cv::fastNlMeansDenoising()`, która reprezentuje jego dokładną implementację. Tablica wynikowa `dst` jest obliczana z tablicy wejściowej `src` przy użyciu obszaru `templateWindowSize` i parametru rozkładu `h` oraz z uwzględnieniem plam w oknie `searchWindowSize`. Obraz może być jedno-, dwu- lub trzykanałowy oraz musi być typu `cv::U8`[13]. Tabela 11.2 zawiera wykaz niektórych wartości dostarczonych przez autorów algorytmu, na które można ustawiać parametr rozkładu `h`.

Tabela 11.2. Zalecane wartości dla funkcji cv::fastNlMeansDenoising() i obrazów w skali szarości

Szum: σ	Rozmiar plamy: s	Okno wyszukiwania	Parametr rozkładu
$0 < \sigma \leq 15$	3×3	21×21	$0.40 \cdot \sigma$
$15 < \sigma \leq 30$	5×5	21×21	$0.40 \cdot \sigma$
$30 < \sigma \leq 45$	7×7	35×35	$0.35 \cdot \sigma$
$45 < \sigma \leq 75$	9×9	35×35	$0.35 \cdot \sigma$
$75 < \sigma \leq 100$	11×11	35×35	$0.30 \cdot \sigma$

Algorytm FNLMD dla kolorowych obrazów — funkcja cv::fastNlMeansDenoisingColor()

```
void cv::fastNlMeansDenoisingColored(
  cv::InputArray src,          // obraz wejściowy
  cv::OutputArray dst,         // obraz wynikowy
  float h = 3,                 // parametr rozkładu wagi jasności
  float hColor = 3,            // parametr rozkładu wagi koloru
  int templateWindowSize = 7, // rozmiar plam używanych do porównywania
  int searchWindowSize = 21    // największa brana pod uwagę odległość plamy
);
```

[12] W tym miejscu należy zwrócić uwagę na jeden drobiazg: waga wkładu piksela p w dotyczące go obliczenia wynosi $w(p,p) = e^0 = 1$. Generalnie oznacza to zbyt wysoką wagę względem innych podobnych pikseli, przez co wartość p zmienia się tylko odrobinę. Z tego względu wagę p określa się jako największą wartość z wag pikseli w obszarze $B(p,s)$.

[13] Choć obraz ten może mieć kilka kanałów, funkcja ta nie jest najlepszym narzędziem do pracy z obrazami kolorowymi. W takim przypadku lepiej sprawdza się funkcja `cv::fastNlMeansDenoisingColored()`.

Drugi wariant algorytmu FNLMD służy do pracy z kolorowymi obrazami i przyjmuje tylko obrazy typu `cv::U8C3`. Choć zasadniczo za pomocą tego algorytmu można by było mniej lub bardziej bezpośrednio przetwarzać obraz w formacie RGB, w praktyce lepszym pomysłem jest jego przekonwertowanie na inną przestrzeń kolorów przed rozpoczęciem obliczeń. Funkcja `cv::fastNlMeansDenoising ↪Colored()` najpierw konwertuje obraz na przestrzeń kolorów LAB, następnie wykonuje algorytmu FNLMD, a na koniec z powrotem konwertuje wynik na format RGB. Największa zaleta tego rozwiązania wiąże się z faktem, że w obrazie kolorowym, w efekcie, są trzy parametry rozkładu. Natomiast w reprezentacji RGB jest mało prawdopodobne, aby trzeba było ustawiać którykolwiek z nich na osobną wartość. W przestrzeni LAB przypisywanie komponentowi jasności innego parametru rozkładu niż komponentom kolorów jest czymś naturalnym. Funkcja `cv::fastNlMeansDenoising ↪Colored()` to umożliwia. Argument `h` reprezentuje parametr rozkładu jasności, podczas gdy nowy parametr `hColor` dotyczy kanałów kolorów. Generalnie wartość parametru `hColor` jest nieco mniejsza od wartości `h`. W większości przypadków odpowiednia jest wartość `10`. Tabela 11.3 zawiera wykaz niektórych wartości, na które można ustawić parametr rozkładu `h`.

Tabela 11.3. Zalecane wartości dla funkcji cv::fastNlMeansDenoising() i kolorowych obrazów

Szum: σ	Rozmiar plamy: s	Okna wyszukiwania	Parametr rozkładu: h
$0 < \sigma \leq 25$	3×3	21×21	$0.55 \cdot \sigma$
$25 < \sigma \leq 55$	5×5	35×35	$0.40 \cdot \sigma$
$55 < \sigma \leq 100$	7×7	35×35	$0.35 \cdot \sigma$

Algorytm FNLMD dla filmów — funkcje cv::fastNlMeansDenoisingMulti() i cv::fastNlMeansDenoisingColorMulti()

```
void cv::fastNlMeansDenoisingMulti(
  cv::InputArrayOfArrays srcImgs, // sekwencja kilku obrazów
  cv::OutputArray dst,            // obraz wynikowy
  int imgToDenoiseIndex,          // indeks obrazu do odszumienia
  int temporalWindowSize,         // liczba obrazów do użycia (nieparzysta)
  float h = 3,                    // parametr rozkładu wagi
  int templateWindowSize = 7,     // rozmiar plam porównawczych
  int searchWindowSize = 21       // maksymalna odległość plamy
);
void cv::fastNlMeansDenoisingColoredMulti(
  cv::InputArrayOfArrays srcImgs, // sekwencja kilku obrazów
  cv::OutputArray dst,            // obraz wynikowy
  int imgToDenoiseIndex,          // indeks obrazu do odszumienia
  int temporalWindowSize,         // liczba obrazów do użycia (nieparzysta)
  float h = 3,                    // parametr rozkładu wagi
  float hColor = 3,               // parametr rozkładu wagi dla koloru
  int templateWindowSize = 7,     // rozmiar plam porównawczych
  int searchWindowSize = 21       // maksymalna odległość plamy
);
```

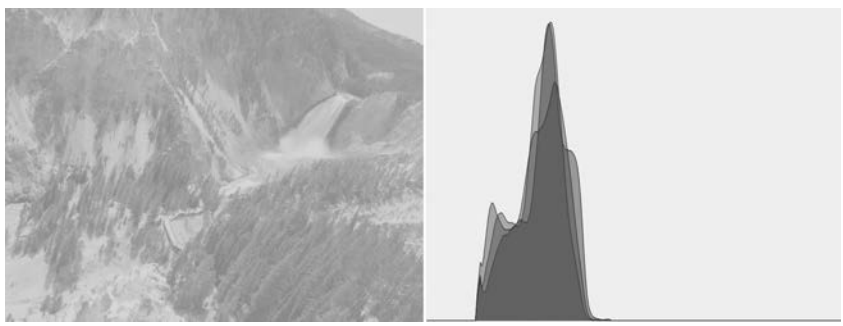
Trzeci i czwarty wariant implementacji omawianego algorytmu służą do pracy z sekwencjami obrazów, na przykład zapisanymi w postaci filmu wideo. Jeśli chodzi o sekwencje obrazów, to naturalnie można się spodziewać, że informacje przydatne przy odszumianiu mogą znajdować się także w innych klatkach niż bieżąca. W większości aplikacji szum zmienia się z obrazu na obraz, podczas gdy sygnał z dużym prawdopodobieństwem pozostaje podobny lub nawet identyczny. Funkcje `cv::fastNlMeansDenoisingMulti()` i `cv::fastNlMeansDenoisingColorMulti()` pobierają tablicę obrazów, `srcImgs`, a nie pojedyncze obrazy. Ponadto należy podać informację, który obraz z sekwencji ma być poddany odszumianiu. Służy do tego parametr `imgToDenoiseIndex`. Dodatkowo należy dostar-

czyć okno wskazujące, ile obrazów z sekwencji ma być wykorzystanych do odszumiania. Parametr ten musi mieć wartość nieparzystą, a implikowane okno zawsze jest wycentrowane na obrazie `imgTo↪DenoiseIndex`. (Gdyby więc parametr `imgToDenoiseIndex` ustawiono na `4`, a `temporalWindowSize` na `5`, to przy odszumianiu zostałyby wykorzystane obrazy 2, 3, 4, 5 i 6).

Wyrównywanie histogramu

Kamery i czujniki obrazu muszą nie tylko uchwycić naturalny kontrast sceny, ale także dostosowywać parametry ekspozycji do aktualnego poziomu oświetlenia. W standardowym aparacie ilość światła docierającego do czujników reguluje się za pomocą migawki i przysłony. Natomiast zakres wartości kontrastu w danym obrazie często przekracza możliwości dostępnego zakresu dynamicznego czujnika. W efekcie konieczne jest poszukiwanie kompromisu między rejestrowaniem ciemnych obszarów (np. cieni), które wymagają dłuższego czasu ekspozycji, i jasnych obszarów, które wymagają krótszego czasu ekspozycji, aby zapobiec nasyceniu białych plam. Często zdarza się tak, że nie jest możliwe skuteczne pogodzenie tych dwóch wymogów w jednym obrazie.

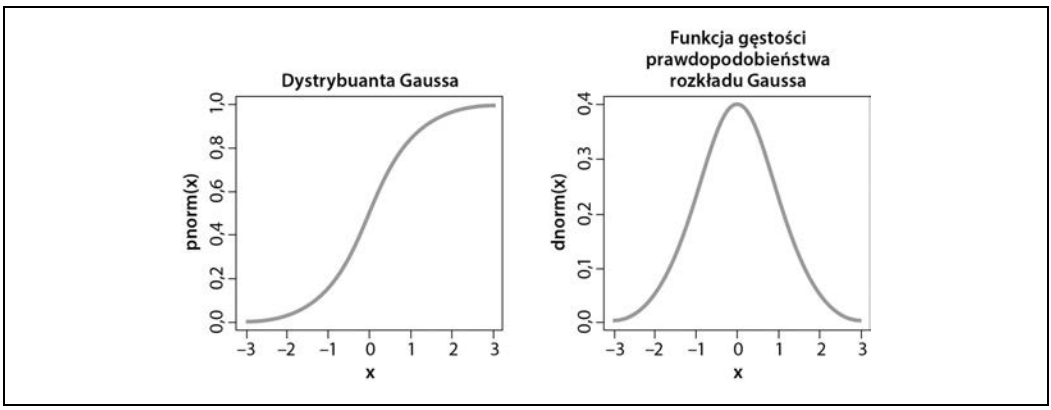
Gdy zdjęcie jest już zrobione, nic nie da się zrobić z nastawieniami czujnika, ale można spróbować poprawić zakres dynamiczny obrazu, aby zwiększyć jego kontrast. Jedną z najczęściej wykorzystywanych technik tego typu jest **wyrównywanie histogramu** (ang. *histogram equalization*)[14]. Na rysunku 11.10 widać, że przyczyną słabej jakości obrazu po lewej jest niewystarczające zróżnicowanie zakresu wartości. Dobitnie jest to widoczne na histogramie wartości intensywności pokazanym po prawej. Ponieważ jest to obraz 8-bitowy, jego zakres intensywności mieści się w przedziale od `0` do `255`, a na histogramie wyraźnie widać, że wszystkie wartości są stłoczone w pobliżu środka przedziału. Technika wyrównywania histogramu polega właśnie na rozszerzaniu zakresu takich wartości.



Rysunek 11.10. Obraz po lewej ma niski kontrast, co potwierdza histogram intensywności widoczny po prawej

[14] Wyrównywanie histogramu to stara technika matematyczna, której zastosowanie w przetwarzaniu grafiki zostało opisane w różnych podręcznikach [Jain86, Russ02, Acharya05], referatach konferencyjnych [Schwarz78], a nawet w odniesieniu do obrazowania żywych tkanek [Laughlin81]. Jeśli zastanawiasz się, dlaczego techniki wyrównywania histogramu nie opisujemy w rozdziale poświęconym histogramom (rozdział 13.), to nie robimy tego, ponieważ technika ta bezpośrednio nie wykorzystuje żadnych histogramowych typów danych. Wprawdzie histogramy są wykorzystywane wewnętrznie, ale z punktu widzenia użytkownika nie są one potrzebne.

Matematyczne podstawy techniki wyrównywania histogramu opierają się na odwzorowywaniu jednego rozkładu (danego histogramu wartości intensywności) na inny rozkład (szerszy i najlepiej jednostajny). Innymi słowy chodzi o to, aby wartości *y* pierwotnego rozkładu jak najrównomierniej rozciągnąć w rozkładzie docelowym. Jak wiadomo, istnieje dobry sposób, by tak rozprowadzić wartości, polegający na wykorzystaniu funkcji zwanej **dystrybuantą**. Na rysunku 11.11 przedstawiamy przykład zastosowania tej funkcji w odniesieniu do nieco wyidealizowanego przypadku gaussowskiego rozkładu gęstości, choć należy pamiętać, że technikę tę można zastosować w odniesieniu do każdego rodzaju rozkładu. Jest to po prostu skumulowana wartość gęstości prawdopodobieństwa oryginalnego rozkładu od jego ujemnej do dodatniej granicy.



Rysunek 11.11. Wynik obliczenia dystrybuanty (po lewej) dla rozkładu Gaussa

Za pomocą dystrybuanty oryginalny rozkład można zmienić na bardziej wyrównany (rysunek 11.12), wyszukując po prostu każdą wartość *y* w rozkładzie oryginalnym i znajdując dla niej odpowiednie miejsce w rozkładzie wyrównanym. W przypadku rozkładów ciągłych uzyskuje się dokładne wyrównanie, natomiast w przypadku rozkładów cyfrowych/dyskretnych rezultaty mogą być dalekie od tego ideału.

Po zastosowaniu tej techniki do zdjęcia z rysunku 11.10 otrzymujemy wyrównany histogram rozkładu intensywności, czego efektem jest poprawa jakości obrazu widoczna na rysunku 11.13.

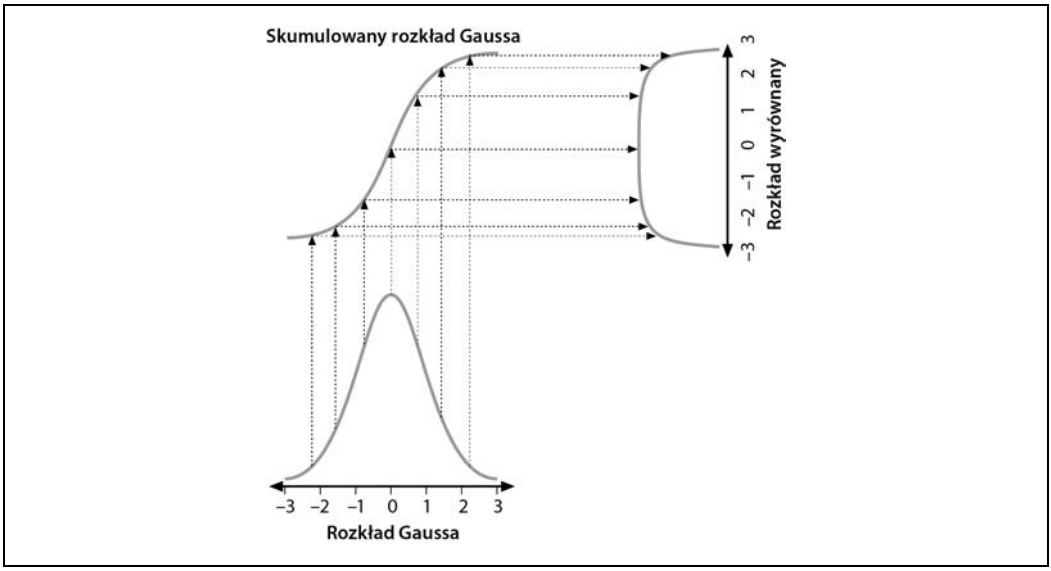
cv::equalizeHist() — wyrównywanie kontrastu

W bibliotece OpenCV opisana technika jest zaimplementowana w postaci jednej zgrabnej funkcji:

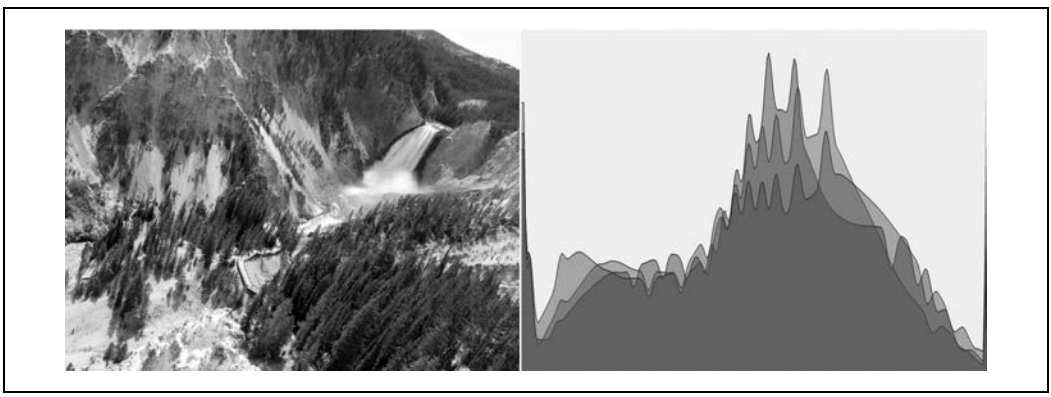
```
void cv::equalizeHist(
  const cv::InputArray src, // obraz wejściowy
  cv::OutputArray dst       // obraz wynikowy
);
```

W funkcji `cv::equalizeHist()` źródłowa tablica `src` musi być jednokanałowym 8-bitowym obrazem. Wynikowy obraz `dst` będzie miał takie same parametry. Obrazy kolorowe można retuszować, rozdzielając je na pojedyncze kanały i każdy z nich przetwarzając osobno[15].

[15] W praktyce efekt osobnego wyrównywania histogramu poszczególnych kanałów rzadko jest zadowalający. Często lepszym rozwiązaniem jest konwersja obrazu na bardziej odpowiednią do takich operacji przestrzeń kolorów (np. LAB) i przeprowadzenie wyrównywania histogramu tylko w kanale jasności.



Rysunek 11.12. Wyrównywanie rozkładu Gaussa przy użyciu dystrybuanty



Rysunek 11.13. Efekt wyrównania histogramu — zakres został rozszerzony

Podsumowanie

W tym rozdziale przedstawiliśmy różne metody przekształcania obrazów. Opisaliśmy techniki skalowania, a także przekształcenia afiniczne i perspektywiczne. Pokazaliśmy, jak zamieniać reprezentacje w kartezjańskim układzie współrzędnych na współrzędne biegunowe. Wszystkie opisane przez nas funkcje łączy to, że pobierają pewien obraz i zamieniają go w inny za pomocą jakiejś globalnej operacji odnoszącej się do całej grafiki. Przedstawiliśmy nawet funkcję, za pomocą której można wykonywać ogólne odwzorowania i w stosunku do której wiele innych funkcji opisanych w tym rozdziale można traktować jako jej szczególne przypadki.

Ponadto nieco miejsca poświęciliśmy niektórym algorytmom wykorzystywanym w retuszu fotografii, takim jak inpainting, odszumianie i wyrównywanie histogramu. Za pomocą tych algorytmów można poprawiać obrazy z aparatu fotograficznego i kamery, choć często znajdują też zastosowanie w implementacji innych technik wizualnych dotyczących ziarnistych lub w inny sposób uszkodzonych danych wideo.

Ćwiczenia

1. Znajdź i załaduj obraz twarzy zwróconej przodem, z otwartymi oczami i zajmującej większość powierzchni obrazu. Napisz program znajdujący źrenice oczu.

Laplasjan „lubi" jasne centralnie położone punkty z ciemną otoczką, a źrenice mają cechy odwrotne. Wystarczy więc wykonać inwersję i splot z odpowiednio dużym laplasjanem.

2. Spójrz na wykresy przedstawiające, jak funkcja konwersji na współrzędne logarytmiczno-biegunowe zamienia kwadrat w falowaną linię.
   a. Narysuj wyniki konwersji na współrzędne logarytmiczno-biegunowe dla przypadku, gdy punkt środkowy znajduje się w jednym z rogów kwadratu.
   b. Jak wyglądałoby koło po przekształceniu na współrzędne logarytmiczno-biegunowe, gdyby punkt centralny znajdował się wewnątrz tego koła i blisko jego krawędzi?
   c. Narysuj wynik transformacji dla przypadku, gdy punkt centralny znajduje się tuż za krawędzią koła.
3. W wyniku transformacji na współrzędne logarytmiczno-biegunowe kształty o różnych kątach rotacji i rozmiarach są przenoszone do przestrzeni, w której te właściwości zamieniają się w przesunięcia względem osi θ i $\log(r)$. Transformacja Fouriera jest niezmienna względem translacji. Jak wykorzystać te informacje w celu wymuszenia na kształtach o różnych rozmiarach i kątach rotacji automatycznego zwracania ekwiwalentnej reprezentacji w dziedzinie logarytmiczno-biegunowej?
4. Narysuj obrazy przedstawiające duży, mały, duży obrócony i mały obrócony kwadrat. Każdy z nich poddaj przekształceniu do dziedziny logarytmiczno-biegunowej. Napisz dwuwymiarowy przesuwacz, który pobiera punkt centralny z reprezentacji logarytmiczno-biegunowej i tak przesuwa kształty, aby były do siebie maksymalnie podobne.
5. Załaduj obraz i poddaj go przekształceniu perspektywicznemu, a następnie rotacji. Czy można te dwa przekształcenia wykonać za jednym razem?
6. Inpainting sprawdza się bardzo dobrze w przypadku usuwania napisów z wzorzystych obszarów. Co by się stało, gdyby napis zasłaniał prawdziwą krawędź obiektu na obrazie? Sprawdź to.
7. Wypróbuj działanie techniki wyrównywania histogramu na kilku własnych obrazach i opisz wyniki.
8. Wyjaśnij różnicę między wyrównywaniem histogramu obrazu a usuwaniem szumu z obrazu.

Skorowidz

A

B

C

D

E

F

G

H

I

J

K

L

Ł

M

N

O

P

Q

R

S

Ś

T

U

W

Y

Z

O'REILLY®

OpenCV: poznaj i stosuj algorytmy przetwarzania obrazów!

Komputerowe rozpoznawanie obrazów przechodzi dziś fazę burzliwego rozwoju. Przyczyniają się do tego ogromna popularność cyfrowych aparatów fotograficznych, wielka liczba grafik zgromadzonych w obszernych internetowych bazach danych, a przede wszystkim coraz doskonalsze algorytmy przetwarzania obrazu. W rozwijaniu tej technologii wielką rolę odegrała biblioteka OpenCV, usprawniając pracę setek tysięcy ludzi. OpenCV 3.x ułatwia efektywne rozwijanie projektów dzięki opartej na języku C++ spójnej architekturze, która doskonale działa na wielu platformach.

Ta książka, przeznaczona dla osób znających język C++, jest praktycznym wprowadzeniem do otwartej biblioteki OpenCV w wersji 3.x. Zawiera też podstawowe informacje na temat komputerowego rozpoznawania obrazu, co powinno ułatwić efektywne posługiwanie się tą biblioteką. Sama biblioteka OpenCV została przedstawiona w sposób umożliwiający bardzo szybkie rozpoczęcie pracy. Książka ułatwia naturalne zrozumienie działania algorytmów, dzięki czemu projektowanie i debugowanie aplikacji nie powinno sprawiać problemów — a to powoduje, że stanowi świetne przygotowanie do zgłębienia bardziej zaawansowanych zagadnień komputerowego rozpoznawania obrazu i uczenia maszynowego.

Dzięki książce opanujesz:

- przegląd biblioteki OpenCV i zawarte w niej funkcje
- pracę z plikami obrazów, filmów i danych oraz przekształcanie obrazów
- ważniejsze algorytmy do pracy na obrazach
- punkty kluczowe: wykrywanie i filtrowanie
- trójwymiarowe widzenie, ruch, określanie pozycji
- uczenie maszyn w OpenCV

Adrian Kaehler — jest naukowcem i założycielem start-upów. Zajmuje się uczeniem maszynowym, modelowaniem statystycznym i komputerowym rozpoznawaniem obrazu. Pracuje w Intel Corporation i w Laboratorium Sztucznej Inteligencji Uniwersytetu Stanforda. Współzakładał Silicon Valley Deep Learning Group.

Gary Rost Bradski — jest naukowcem i konsultantem. Zajmuje się robotyką, uczeniem maszynowym i komputerowym rozpoznawaniem obrazów. Pracuje w Laboratorium Sztucznej Inteligencji Uniwersytetu Stanforda. Współtworzył takie biblioteki jak Open Source Computer Vision Library, Machine Learning Library i Probabilistic Network Library (PNL).

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

sięgnij po WIĘCEJ

KOD KORZYŚCI

ISBN 978-83-283-1656-0

9 788328 316560

Informatyka w najlepszym wydaniu

cena: 149,00 zł