Obrazy funkcyjne – projekt zaliczeniowy (*Programowanie I R*)

Tomasz Tarkowski

Uniwersytet Warszawski, Wydział Fizyki

26 marca 2025

Wstęp

W jaki sposób można wyrazić obraz (grafikę) z użyciem komputera? Z pewnością kojarzysz bitmapę. Jest ona najprostszym koncepcyjnie sposobem reprezentacji grafiki. Obraz jest dzielony na bardzo małe kwadraty zwane *pikselami* i każdemu z nich przypisuje się jakiś kolor. Sposób jest prosty, obraz zawiera różne możliwe szczegóły a pliki są... duże — zwłaszcza, gdy mówimy o bitmapach nieskompresowanych.

Czego można użyć w zamian, nie uciekając się do algorytmów kompresji, aby efektywniej przedstawiać przynajmniej niektóre obrazy? Otóż innym rozwiązaniem jest zastosowanie grafiki wektorowej. Wykorzystuje ona idee matematyczne takie jak *wektor* oraz *krzywa*. Grafika wektorowa ma sporą przewagę nad bitmapami zwłaszcza tam, gdzie obrazy posiadają jednolite kolory na dużych obszarach i kształty są łatwe do opisania przez formuły matematyczne. Jednak czy to jest koniec możliwości tego rozwiązania? Okazuje się, że nie, gdyż istnieje uogólnienie grafiki wektorowej, czyli tzw. *obrazy funkcyjne*, które mogą zostać zrealizowane z użyciem programowania funkcyjnego omówionego na jednym z wykładów.

Definicje

Zanim można będzie sformułować problem do rozwiązania, trzeba będzie wprowadzić pewne nazwy i oznaczenia. Wcześniejsza wiedza z zakresu grafiki komputerowej może być pomocna w zrozumieniu nomenklatury, ale nie jest ona wymagana. Podane w tym dokumencie wyjaśnienia powinny być wystarczające do rozwiązania zadania projektu, jednak jeśli chcesz wiedzieć więcej, to pod adresem http://conal.net/papers/functional-images/fop-conal.pdf znajduje się wersja autorska rozdziału książki, który to rozdział poświęcony jest obrazom funkcyjnym i który może być pomocny do rozszerzenia wiedzy [1].

Przez *kolor* c rozumie się trójkę liczb $c = (r, g, b)$, gdzie $r, g, b \in [0, 255]_{\mathbb{N}}$ informują odpowiednio o ilości barwnika czerwonego, zielonego i niebieskiego. Przykładami są kolor czarny $(0, 0, 0)$, biały $(255, 255, 255)$, czerwony $(255, 0, 0)$, zielony $(0, 255, 0)$, niebieski $(0, 0, 255)$ i karaibski błękit $(26, 193, 221)$. Zbiór wszystkich kolorów to C:

$$C = \{(r, g, b) \in \mathbb{N}^3 \mid r, g, b < 256\} = [0, 255]^3_{\mathbb{N}} \quad (1)$$

W tym podejściu istnieje więc $256^3 = (2^8)^3 = 2^{24}$ różnych kolorów (mówi się wtedy, że obraz ma *24-bitową głębię*). Jeśli kojarzysz czym jest transparencja to, aby wyjaśnić potencjalną wątpliwość należy dodać, że wyżej opisany model jest kodowaniem bez transparencji i tylko takim będziemy się dalej zajmować.

Co można zrobić z kolorem? Na kolorach istnieją różne operacje. Jedną z nich jest średnia ważona dwóch kolorów. Jeśli dysponujemy kolorami $c_1 = (r_1, g_1, b_1)$ oraz $c_2 = (r_2, g_2, b_2)$ to ich średnia ważona z wagą $w \in [0, 1]$ odpowiada kolorowi:

$$(\lfloor (1-w)r_1 + wr_2 \rfloor, \lfloor (1-w)g_1 + wg_2 \rfloor, \lfloor (1-w)b_1 + wb_2 \rfloor) \quad (2)$$

gdzie $\lfloor \cdot \rfloor$ jest funkcją *podłoga*. Zastosowanie podłogi wynika z faktu skończonej głębi bitowej.

Przez *punkt* p rozumie się punkt w przestrzeni dwuwymiarowej $p \in \mathbb{R}^2$. Gdy mówimy o obrazach funkcyjnych wygodnie jest operować na zbiorze $\mathbb{R}^2$, natomiast w momencie, gdy trzeba obraz zapisać w postaci bitmapy, można zamiast punktu mówić o *pikselu* $p \in \mathbb{N}^2$.

Obraz funkcyjny to odwzorowanie, które każdemu punktowi przypisuje kolor:

$$I_C : \mathbb{R}^2 \to C \quad (3)$$

Gdy chcemy obraz funkcyjny zapisać w postaci bitmapy o wymiarach $w \times h$ (w pikseli szerokości i h pikseli wysokości) to można rozumieć to jako zwężenie funkcji do podzbioru pikseli $[0, w-1]_{\mathbb{N}} \times [0, h-1]_{\mathbb{N}}$:

$$I_C|_{[0,w-1]_{\mathbb{N}} \times [0,h-1]_{\mathbb{N}}} : [0, w-1]_{\mathbb{N}} \times [0, h-1]_{\mathbb{N}} \to C \quad (4)$$

$$I_C|_{[0,w-1]_{\mathbb{N}} \times [0,h-1]_{\mathbb{N}}}(p) = I_C(p) \quad (5)$$

Zwróć uwagę, że obydwie współrzędne piksela są numerowane od zera. Co więcej, zwyczajowa notacja zakłada, że piksel $(0, 0)$ znajduje się w lewym górnym rogu a piksel $(w-1, h-1)$ — w prawym dolnym rogu obrazu.

Okazuje się, że obraz funkcyjny ma szersze znaczenie, aniżeli przedstawione wyżej. Dlatego odwzorowanie I_C będziemy dalej nazywać *właściwym obrazem funkcyjnym* a samo określenie *obraz funkcyjny* rozszerzymy na odwzorowanie $I_{\mathcal{X}} : \mathbb{R}^2 \to \mathcal{X}$ dla dowolnego zbioru $\mathcal{X}$.

Jakie więc jeszcze istnieją obrazy funkcyjne poza właściwymi? W obszarze naszych zainteresowań są jeszcze dwa: region oraz mieszanka. Poprzez *region*, zwany też *wycinanką*, rozumie się obraz funkcyjny $I_{\mathbb{B}}$, gdzie $\mathbb{B} = \{\text{fałsz}, \text{prawda}\}$. Regiony mogą być przydatne do składania obrazów funkcyjnych. Poprzez *mieszankę* rozumie się natomiast obraz funkcyjny I_U, gdzie $U = [0, 1]_{\mathbb{R}}$. Mieszanka służy do mieszania kolorów w proporcjach zadanych parametrem ze zbioru U.

Przypomnijmy sobie teraz pewną notację stosowaną w teorii mnogości. Zbiór wszystkich funkcji ze zbioru A w zbiór B można oznaczyć jako B^A. Tak więc:

$$\mathcal{X}^{\mathbb{R}^2} = \{f \subseteq \mathbb{R}^2 \times \mathcal{X} \mid \forall p \in \mathbb{R}^2\ \exists! x \in \mathcal{X} \colon (p, x) \in f\} \quad (6)$$

jest zbiorem wszystkich możliwych obrazów funkcyjnych, natomiast:

$$\mathcal{C}^{\mathbb{R}^2},\ \mathbb{B}^{\mathbb{R}^2},\ \mathcal{U}^{\mathbb{R}^2} \quad (7)$$

są odpowiednio zbiorami wszystkich możliwych właściwych obrazów funkcyjnych, regionów oraz mieszanek.

Przyjrzyjmy się mieszaniu właściwych obrazów funkcyjnych z użyciem mieszanek. Niech mix będzie *funkcją mieszania*:

$$\mathrm{mix} \colon \mathcal{U}^{\mathbb{R}^2} \times \mathcal{C}^{\mathbb{R}^2} \times \mathcal{C}^{\mathbb{R}^2} \to \mathcal{C}^{\mathbb{R}^2} \quad (8)$$

Funkcja mieszanie przyjmuje zatem mieszankę i dwa właściwe obrazy funkcyjne i zwraca trzeci właściwy obraz funkcyjny:

$$\mathrm{mix}(I_U, I_{C1}, I_{C2}) = I_C \quad (9)$$

Zwrócony obraz funkcyjny spełnia warunek:

$$I_C(p) = I_U(p) \cdot I_{C1}(p) + (1 - I_U(p)) \cdot I_{C2}(p) \quad (10)$$

Oznacza to, że otrzymany właściwy obraz funkcyjny został zmieszany z dwóch innych według proporcji określonej przez mieszankę. Szczególnym przypadkiem jest użycie mieszanki tożsamościowo równej 0 (lub 1). Wtedy otrzymany właściwy obraz funkcyjny jest kopią jednego z właściwych obrazów funkcyjnych podanych jako argumenty wywołania funkcji mieszania mix.

Teraz przyjrzyjmy się prawdopodobnie najtrudniejszemu pojęciu spośród dotychczas opisywanych. Niech lift będzie *funkcją uniesienia punktowego*:

$$\mathrm{lift} \colon D^{A_1 \times \cdots \times A_n} \times A_1^{\mathbb{R}^2} \times \cdots \times A_n^{\mathbb{R}^2} \to D^{\mathbb{R}^2} \quad (11)$$

W tym momencie nie określamy czym dokładnie są zbiory D oraz A_i. Funkcja lift przyjmuje jako swoje argumenty zestaw funkcji h oraz f_i i zwraca funkcję g:

$$\mathrm{lift}(h, f_1, \ldots, f_n) = g \quad (12)$$

która spełnia warunek:

$$g(p) = h(f_1(p), \ldots, f_n(p)) \quad (13)$$

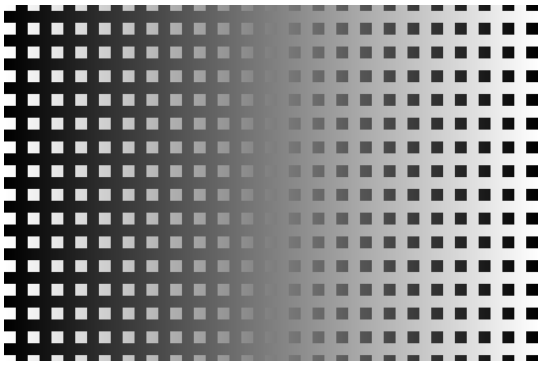
Zwróćmy uwagę, że następujący zapis uznajemy za prawidłowy:

$$\mathrm{lift}(h, f_1, \ldots, f_n)(p) \quad (14)$$

i oznacza on $g(p)$. Co więcej, w szczególnym przypadku $n = 0$, funkcja h nie przyjmuje argumentów, tzn. $h = d = \mathrm{const}$ i wtedy $\mathrm{lift}(h)(p) = d$.

Aby lepiej zrozumieć funkcję lift, przyjrzyjmy się przypadkowi $n = 2$:

$$\mathrm{lift}(h, f_1, f_2) = g \quad (15)$$
$$h \colon A_1 \times A_2 \to D \quad (16)$$
$$f_1 \colon \mathbb{R}^2 \to A_1 \quad (17)$$
$$f_2 \colon \mathbb{R}^2 \to A_2 \quad (18)$$
$$g \colon \mathbb{R}^2 \to D \quad (19)$$
$$g(p) = h(f_1(p), f_2(p)) \quad (20)$$



Rysunek 1: Przykładowy obraz wygenerowany z użyciem biblioteki Pillow.

Oznacza to, że funkcja lift umieszcza rezultat działania funkcji f_1 i f_2 jako argumenty faktyczne funkcji h. Jeśli więc $f_1(p) = a_1$ oraz $f_2(p) = a_2$ a $h(a_1, a_2) = d$ i $\mathrm{lift}(h, f_1, f_2) = g$ to wtedy $g(p) = d$.

Tworzenie obrazów

Do tworzenia obrazów można wykorzystać jedną z bibliotek graficznych, np. Pillow [2]. Przykładowy kod użycia tej biblioteki znajduje się poniżej w treści kodu źródłowego 1. Wynik działania programu można zobaczyć na rysunku 1.

Kod źródłowy 1: Użycie biblioteki Pillow.

```
1  from PIL import Image
2
3  width, height = 450, 300
4
5  def color(p):
6      w, h = p[0], p[1]
7      t = w / width
8      c0 = int(255 * t)
9      c1 = 255 - c0
10     if (w - 0) % 20 < 10 and (h + 5) % 20 < 10:
11         return (c1, c1, c1) # Siec kwadratow.
12     else:
13         return (c0, c0, c0) # Tlo.
14
15 img = Image.new("RGB", (width, height)) # Tworzy pusty obraz.
16 pixels = img.load() # Udostepnia piksele obrazu.
17
18 for w in range(width):
19     for h in range(height):
20         pixels[w, h] = color((w, h)) # Przypisuje pikselowi jakis kolor.
21
22 img.save("output.bmp") # Zapisuje obraz.
```

W przykładzie widać mechanizm tworzenia pliku BMP. Z użyciem *Image.new()* można stworzyć nową bitmapę z dostępem do poszczególnych jej pikseli realizowanym poprzez metodę *load()*. Piksele indeksuje się z użyciem dwóch liczb naturalnych według konwencji opisanej wcześniej, tzn. piksel $(0, 0)$ znajduje się w lewym górnym rogu obrazu. Pikselowi można przypisać kolor z 24-bitową głębią. Obraz można zapisać do pliku BMP o wybranej nazwie dzięki metodzie *save()*.

Zadanie

Twoim zadaniem jest zaimplementowanie opisanych dalej funkcji i napisanie w ich oparciu program tworzący

obrazy przedstawione na rysunku 2 — należy przedstawić kod tworzący wszystkie 12 paneli tego rysunku począwszy od (a) a skończywszy na (l). Funkcje stwórz w sposób możliwie najbardziej zgodny z podanym opisem. W implementacji funkcji nierzadko należy wykorzystać funkcje lambda. Funkcje *compose* oraz *lift* powinny być użyte w implementacji innych funkcji. Przydatna w implementacji innych funkcji będzie również *translate*. Kod tworzący rysunki powinien możliwie najdokładniej odtworzyć pierwowzór zaprezentowany na rysunku 2. Zauważ, że nie musisz definiować typu danych odpowiadającemu punktowi $p \in \mathbb{R}^2$ — możesz w zamian użyć krotki rozmiaru dwa, tzn. pary, tak jak pokazano to w kodzie źródłowym 1. Rozwiązanie zadania w formie pojedynczego pliku o rozszerzeniu *.py* należy przesłać na adres e-mail Tomasz.Tarkowski@fuw.edu.pl. Do rozwiązania nie należy dołączać żadnych innych plików (w szczególności nie należy dołączać plików graficznych). Uwaga: Zadanie może być bardzo wymagające do pomyślnego ukończenia. Z drugiej strony, wykonane samodzielnie, może być bardzo pouczające.

compose

Funkcja *compose(*fs)*, która przyjmuje zmienną liczbę argumentów, realizuje złożenie funkcji, np. *compose(f1, f2, f3)* należy interpretować jako $f_3 \circ f_2 \circ f_1$, tzn. jako funkcję h spełniającą warunek $h(x) = f_3(f_2(f_1(x)))$, przy założeniu, że *f1*, *f2* i *f3* są interpretowane jako odpowiednio f_1, f_2 i f_3. W szczególnym przypadku, gdy *compose()* jest wywoływane bez argumentów, należy zwrócić funkcję identycznościową, tzn. równoważną *lambda x: x*.

lift

Funkcja *lift(h, *fs)* realizuje uniesienie punktowe.

rotate

Funkcja *rotate(image, phi)* przyjmuje obraz funkcyjny oraz kąt podany w radianach i zwraca obraz funkcyjny obrócony o ten kąt. Orientację obrotu należy wydedukować z rysunku 2.

translate

Funkcja *translate(image, vector)* przyjmuje obraz funkcyjny oraz wektor i zwraca obraz funkcyjny przesunięty o ten wektor. Zwrot przesunięcia należy wydedukować z rysunku 2.

scale

Funkcja *scale(image, s)* przyjmuje obraz funkcyjny oraz współczynnik *s* i zwraca obraz funkcyjny przeskalowany o ten współczynnik.

cond

Funkcja *cond(region, image1, image2)* korzysta z regionu (wycinanki), który wycina i klei dwa obrazy właściwe, tworząc w ten sposób nowy. Dla punktów *p*, takich że *region(p) == True*, brany jest obraz *image1*, zaś dla pozostałych — *image2*. Wskazówka: W implementacji skorzystaj z funkcji *lift*.

lerp

Funkcja *lerp(blend, image1, image2)* korzysta z mieszanki, która wyznacza proporcje mieszania kolorów dwóch obrazów właściwych, tworząc w ten sposób nowy. Dla każdego punktu *p* argument *blend* zwraca parametr mieszania oznaczony jako *w*, który informuje, ile w finalnym obrazie dla punktu *p* będzie koloru z obrazu *image1*, a ile z obrazu *image2*. Wskazówka: W implementacji skorzystaj z funkcji *lift*.

darken

Funkcja *darken(image, blend)* zwraca ściemnioną wersję obrazu właściwego *image* w oparciu o mieszankę *blend*. Dla szczególnego przypadku *blend* tożsamościowo równego 1 oznacza stworzenie czarnego obrazu, a dla przypadku 0 — oznacza kopię *image*.

lighten

Funkcja *lighten(image, blend)* zwraca rozjaśnioną wersję obrazu właściwego *image* w oparciu o mieszankę *blend*. Dla szczególnego przypadku blend tożsamościowo równego 1 oznacza stworzenie białego obrazu, a dla przypadku 0 — oznacza kopię *image*.

weighted_mean

Funkcja *weighted_mean(color1, color2, w)* zwraca średnią ważoną kolorów.

constant

Funkcja *constant(x)* przyjmuje argument *x* i zwraca stały obraz funkcyjny I_χ, gdzie każdy piksel ma wartość:

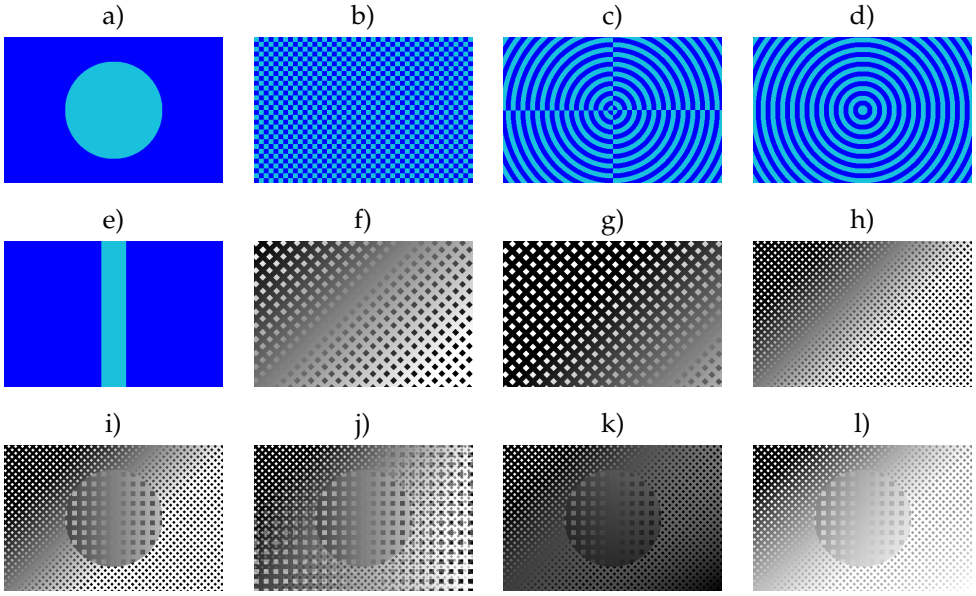
- logiczną *True* w przypadku obrazu funkcyjnego będącego regionem dla argumentu *x == True*,
- koloru „karaibski błękit" w przypadku obrazu funkcyjnego będącego obrazem właściwym dla argumentu *x == Color_Caribbean_blue* lub
- zmiennoprzecinkową 0,42 w przypadku obrazu funkcyjnego będącego mieszanką dla argumentu *x == 0.42*.

circle

Funkcja *circle(center, radius, inner, outer)* zwraca obraz funkcyjny koła o środku *center* i promieniu *radius*. Obraz funkcyjny zwraca wartość *inner* wewnątrz koła i na jego obwodzie oraz *outer* poza nim.

checker

Funkcja *checker(width, value1, value2)* zwraca obraz funkcyjny szachownicy o szerokości kratki *width* oraz zwracanych wartościach zadanych przez pozostałe dwa argumenty.



Rysunek 2: Obrazy do zadania. Opisy poszczególnych paneli: a) koło o promieniu 100 pikseli w kolorze karaibskiego błękitu umieszczone na środku obrazu wypełnionego kolorem niebieskim, b) szachownica o boku długości 10 pikseli, c) szachownica biegunowa z czteroma panelami, d) naprzemienne koncentryczne pasy, e) pasek o szerokości 50 pikseli, f) obraz z rysunku 1 obrócony o $\pi/4$, g) obraz z poprzedniego panelu dodatkowo przesunięty o wektor $(100, 200)$ pikseli, h) obraz z poprzedniego panelu dodatkowo przeskalowany o czynnik 0,5, i) obraz stworzony z użyciem wycinanki w kształcie koła, obrazu z rysunku 1 oraz z obrazu z poprzedniego panelu, j) obraz stworzony z użyciem mieszanki opisanej funkcją *lambda p: (p[0] / width + p[1] / height) / 2.*, obrazu z poprzedniego panelu oraz z obrazu z rysunku 1, k) obraz z panelu (i) ściemniony z użyciem tej samej funkcji lambda, l) obraz z panelu (i) rozjaśniony z użyciem tej samej funkcji lambda. Wszystkie obrazy mają wymiary 450 na 300 pikseli.

polar_checker

Funkcja *polar_checker(center, width, n, value1, value2)* zwraca obraz funkcyjny „szachownicy biegunowej" o środku w punkcie *center*, zawierającej *n* paneli o szerokością kątowej $2\pi/n$ każdy. W implementacji należy m.in. odwołać się do zdefiniowanej wcześniej funkcji *checker* i przekazać jej argument *width*. W rozwiązaniu wystarczy założyć, że *n* jest parzyste.

rings

Funkcja *rings(center, width, value1, value2)* zwraca obraz funkcyjny koncentrycznych naprzemiennych pasów szerokości *width* o środku w punkcie *center*.

vertical_stripe

Funkcja *vertical_stripe(center, width, value1, value2)* zwraca obraz funkcyjny w postaci paska *value1* o szerokości *width* otoczonego przez *value2* wycentrowanego wokół *center*.

Podziękowania

Treść projektu została zainspirowana jednym z zadań z przedmiotu „Języki i narzędzia programowania I" oferowanego na Wydziale Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego oraz pracą Conala Elliotta [1].

Bibliografia

[1] C. Elliott. "Functional images". W: *The Fun of Programming*. Red. J. Gibbons i O. de Moor. "Cornerstones of Computing" series. Houndmills, Basingstoke, Hampshire RG21 6XS; 175 Fifth Avenue, New York, NY 10010: Palgrave Macmillan, 2003. URL: http://conal.net/papers/functional-images/.

[2] *Pillow (PIL Fork)*. URL: https://pillow.readthedocs.io/.