Bogusław Jackowski

GRAFIKA DYSKRETNA

CZYLI PARĘ MNIEJ LUB BARDZIEJ PRZYDATNYCH
WIADOMOŚCI O MAPACH BITOWYCH

1. Wstęp

Słowo *dyskretny* pochodzi od łacińskiego słowa *discretio – rozdzielenie*, *rozróżnienie*, chociaż w potocznym języku polskim zagubiło pierwotny sens i nabrało znaczenia związanego z zachowaniem tajemnicy. W naukach ścisłych określenie *dyskretny* nawiązuje do źródłosłowu, oznacza bowiem „coś, co składa się z kawałków" – i taki jest też sens pojęcia *grafika dyskretna*.

Czasem spotyka się określenie „grafika bitmapowa" lub – mało trafnie – „grafika rastrowa". Jak zwał tak zwał, ważniejsze, że jest to bardzo rozpowszechniona forma grafiki, bowiem praktycznie wszystkie współczesne urządzenia wyjściowe komputerów, takie jak monitory, drukarki czy plotery, są urządzeniami dyskretnymi. Ale nie chodzi tu tylko – jak mogłoby się zdawać – o grafikę ilustracyjną. Wielu użytkowników TEX-a zdziwi się pewnie, dowiedziawszy się, że „mówi prozą", czyli że używa na co dzień grafiki dyskretnej. Otóż najpopularniejsza odmiana fontów TEX-owych, czyli fonty PK, to nic innego jak czarno-biała grafika dyskretna.

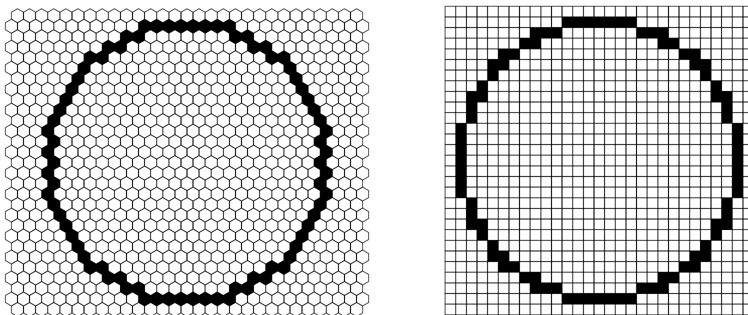
Pierwszy krok w stronę grafiki dyskretnej za nami, a skoro tak, to dlaczego by nie pójść dalej i nie skorzystać z dobrodziejstw, jakie stwarza PostScript, „znający się" na grafice dyskretnej jak nikt na świecie i mający jeszcze tę zaletę, że „dogaduje się" świetnie z TEX-em? Osobiście nie widzę żadnego powodu, zwłaszcza że korzystanie z owych dobrodziejstw jest w większości wypadków dziecinnie łatwe i wcale nie wymaga studiowania niniejszej broszurki.

Po co więc pisać broszurkę, której nie trzeba czytać? Bo czasami – jak to w życiu – trafiają się złośliwe przypadki, i wówczas bez wiedzy ani rusz. Moim celem jest opowiedzenie ogólnie o podstawach grafiki dyskretnej, o popularnych obecnie realizacjach grafiki dyskretnej, na przykład o formatach GIF, TIFF, JPEG, itp., a także o tym, co stąd wynika dla użytkowników TEX-a, w szczególności o kłopotach, na jakie można natrafić.

Innymi słowy, liczę na to, że chociaż wiedza zawarta w tej broszurce nie jest niezbędna do korzystania z grafiki dyskretnej, to jednak może się okazać przydatna.

Grafika dyskretna to pojęcie trącące zbytnią ogólnością. W praktyce przyjdzie się nam najczęściej spotykać z tworami określanymi mianem map bitowych – por. rys. 1.

Termin *mapa bitowa* wywodzi się z angielskich słów *map* i *bit*; słowo *map* oznacza po prostu *mapę*, ale też – w matematyce – *przyporządkowanie*, *odwzorowanie*, zaś słowo *bit* oznacza potocznie *kawałek*, a w informatyce – *jednostkę informacji.* Nieformalnie mówiąc, chodzi tu o przyporządkowanie elementom (kawałkom) rysunku pewnych wielkości, takich jak na przykład kolor czy stopień przezroczystości. Bardziej adekwatne byłoby więc mówienie o *mapach pikselowych*, skoro najmniejszy element informacji graficznej w przypadku grafiki dyskretnej zwie się *pikselem* (jest to akronim od angielskich słów *picture element*), ale przyjęło się określenie *mapa bitowa*, a że nie brzmi ono tragicznie źle w języku polskim, pozwolę sobie przy tym określeniu pozostać.



Rys. 1. Teoretycznie rzecz biorąc podział rysunku na „kawałki" można przeprowadzić dowolnie, nie musi to być koniecznie podział na prostokąty. Można sobie wyobrazić na przykład podział na sześciokąty ułożone podobnie jak w plastrze miodu. Ponieważ technologia komputerowa jak na razie oferuje podział obrazu na identyczne prostokąty (co nie oznacza bynajmniej, że kropki są zawsze prostokątne), w dalszym ciągu skupimy się wyłącznie na prostokątnych mapach bitowych.

Z formalnego punktu widzenia mapa bitowa to **macierz prostokątna** $V_{i,j}$, gdzie indeksy $i = 1, 2, \ldots m$, $j = 1, 2, \ldots n$ reprezentują współrzędne punktów (pikseli) danego urządzenia. Natychmiast nasuwa się pytanie: jakie wartości przyjmują elementy macierzy $V_{i,j}$ i jak je należy interpretować? Temu zagadnieniu poświęcona jest właśnie reszta broszury.

2. Grafika dyskretna a grafika obwiedniowa

Zanim jednak pogrążymy się bez reszty się w sprawach „wymagających dyskrecji", chciałbym wspomnieć o innym rodzaju reprezentacji grafiki

komputerowej, mianowicie o *grafice obwiedniowej*, zwanej czasem *grafiką wektorową*.

Otóż reprezentowanie grafiki za pomocą map bitowych, oprócz licznych zalet, ma parę wad. W praktyce uciążliwa jest z reguły duża objętość plików zawierających mapy bitowe. Ponadto grafika dyskretna związana jest z konkretnym urządzeniem, głównie z jego rozdzielczością, co wiąże się z kolei z koniecznością skalowania przy zmianie urządzenia. Skalowanie map bitowych jest dość kłopotliwe – wrócimy jeszcze do tego tematu. Podobnie inne, równie proste przekształcenia map bitowych, takie jak pochylenie, obrót o kąt różny od wielokrotności 90°, itp., sprawiają zasadnicze kłopoty.

W przypadku grafiki zawierającej dużą liczbę szczegółów (np. zdjęć) wad tych uniknąć się w zasadzie nie da. Często jednak grafika przedstawia bardzo proste obiekty, np. płaskie figury geometryczne. Wówczas możliwy jest abstrakcyjny, bardzo zwarty opis elementów graficznych, nie odwołujący się do dyskretnej struktury urządzeń wyjściowych. Aby zdefiniować płaski obiekt graficzny, wystarczy określić kształt jego brzegu czyli *obwiedni* – stąd właśnie nazwa tego sposobu opisu grafiki. W zamierzchłych czasach kontury bywały przybliżane za pomocą łamanych, czyli za pomocą ciągu wektorów – stąd określenie *grafika wektorowa*. Niektóre programy do dziś stosują ten archaiczny sposób, tak na przykład reprezentowana jest grafika w formacie WMF – *Windows Metafile*.

Istnieje wiele języków abstrakcyjnego opisu grafiki. Użytkownicy TEX-a mieli szansę zetknąć się co najmniej z dwoma: jeden to METAFONT, służący do opisu fontów TEX-owych, drugi to POSTSCRIPT, język opisu stron tekstowo-graficznych, w który wyposażone są niektóre drukarki i praktycznie wszystkie fotonaświetlarki.

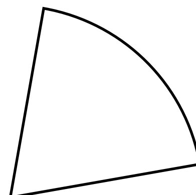
Oczywiście w ostatecznym rozrachunku grafika musi zostać sprowadzona do postaci dyskretnej, ale wygodne bywa pozostawienie tej fazy odpowiednim programom – staje się ona wówczas dla użytkownika prawie niezauważalna.

Na przykład zamiast przechowywać fonty na dysku można przechowywać jedynie źródła METAFONT-owe, a zadanie tworzenia map bitowych, czyli fontów PK dla konkretnego urządzenia, można wykonywać jedynie w razie potrzeby. Zarządzać tym procesem może odpowiedni sterownik, czyli program przetwarzający plik DVI na dane dla danego urządzenia. Po wykonaniu zadania fonty PK mogą być z dysku nawet usuwane, gdyż ponowne ich wy-

generowanie nie jest kosztowne – średniej klasy komputer osobisty generuje pełny zestaw znaków fontu CM w ciągu kilku sekund.

Podobnie jest w przypadku języka PostScript. Programy zapisane w tym języku często bywają wielokrotnie krótsze niż odpowiadająca im mapa bitowa (p. rys. 2), generowana podczas przetwarzania programu przez interpreter PostScript-u – może to być interpreter wbudowany w drukarkę lub naświetlarkę, bądź też program, taki jak np. dystrybuowany jako oprogramowanie swobodne `Ghostscript`.

```
newpath 0 0 moveto          % inicjalizacja
0 0 72 10 80 arc closepath % definicja figury
1 setlinewidth              % szerokość piórka = 1bp
stroke showpage             % rysowanie figury
```



Rys. 2. Króciutki program, zajmujący wraz z komentarzami około 200 bajtów (procent, podobnie jak w TEX-u, oznacza komentarz), opisuje bardzo prosty obiekt – wycinek koła. Załóżmy, że utworzona została czarno-biała mapa bitowa tej figury w celu wydrukowania na drukarce laserowej. Jej objętość zależeć będzie od tego, jaki format został zastosowany. Niemniej jednak nawet przy najoszczędniejszym sposobie pamiętania mapa bitowa zajęłaby objętość ośmiokrotnie większą niż program PostScript-owy. Przygotowanie mapy bitowej z rozdzielczością fotonaświetlarkową spowodowałoby dalszy, pięciokrotny wzrost objętości.

O grafice obwiedniowej można więc myśleć jako o swoistej metodzie kompresji danych: zamiast pamiętać poszczególne piksele, pamiętamy przepis na ich malowanie, a samo malowanie, czyli generowanie map bitowych zostawiamy stosownemu programowi – METAFONT-owi czy PostScript-owi.

Jak widać, wszystkie drogi prowadzą do map bitowych – wróćmy zatem do przerwanego wątku.

3. Podstawowe rodzaje map bitowych

3a. *Czarno-białe mapy bitowe*

W najprostszym przypadku $V_{i,j}$ może przyjmować dwie wartości: 0 i 1. Próżno by wszakże oczekiwać jakiejś jednolitości, na przykład że 0 oznacza kolor biały, a 1 – czarny. Tak prosto w świecie komputerów nie bywa. Czasami jest tak, czasami na odwrót, a czasami jedna z tych liczb oznacza aktualnie

wybrany kolor, a druga – obszar przezroczysty. Jak więc widać, określenie *czarno-biały* jest nieco mylące – precyzyjniejsze byłoby mówienie o grafice *dwubarwnej*, jeśli zgodzimy się, że przezroczystość będziemy rozumieć jako szczególny kolor.

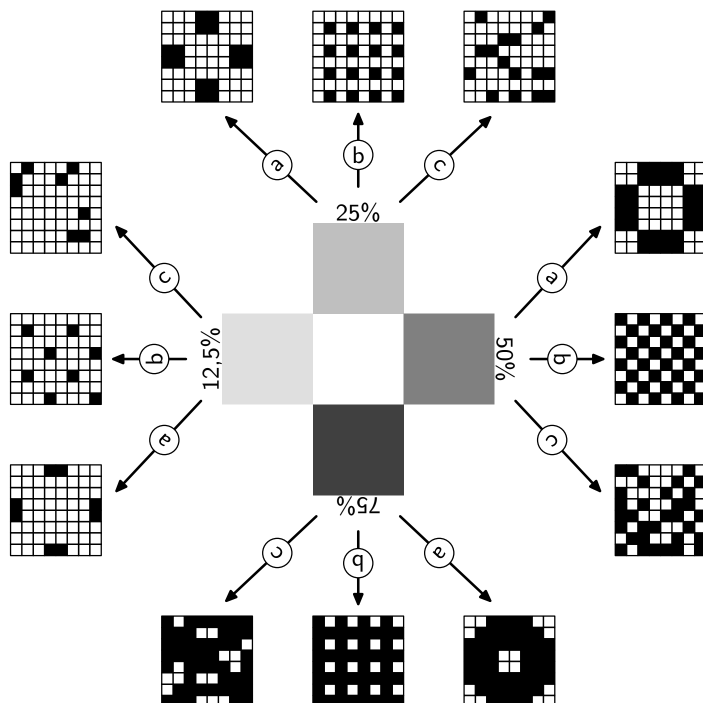
Sposób interpretacji zależy oczywiście od programu, za pomocą którego przetwarzana jest grafika. Nas na ogół interesować będzie PostScript. Jeżeli chodzi o czarno-białe mapy bitowe, to PostScript został wyposażony we wszystkie wymienione możliwości. Z niezrozumiałych przyczyn przezroczystość – tak przecież przydatna – nie należy do atrybutów pozostałych rodzajów map bitowych akceptowanych przez PostScript. A szkoda.

Rys. 3. Te same dane mogą być interpretowane w różny sposób. W przypadku dwubarwnych map bitowych możliwych kombinacji jest – jak widać – co najmniej cztery: (1) rysunek czarny, tło przezroczyste; (2) rysunek przezroczysty, tło czarne; (3) rysunek czarny, tło białe; (4) rysunek białe, tło czarne. W istocie warianty (3) i (4) to szczególne przypadki sytuacji, w której i rysunkowi, i tłu nadawany jest wybrany kolor; w przypadkach (1) i (2) można także wybrać inny kolor niż czarny.

3b. *Cieniowane mapy bitowe*

Dwubarwne mapy bitowe dają się w sposób naturalny uogólnić, można mianowicie założyć, że $V_{i,j}$ przyjmuje wartości ze zbioru 0, 1, 2, ... c, gdzie c jest pewną z góry zadaną liczbą. Tym razem mamy do czynienia z niezwyczajną w świecie komputerowym zgodnością – niemalże wszystkie programy zgadzają się na wartość c równą 255. Mało tego, prawie zawsze można bezpiecznie

zakładać, że 0 oznacza kolor czarny, a 255 – biały. Konwencję tę można łatwo przyswoić myśląc o szarości (*grayscale*) jako o mierze oświetlenia: 0 to brak oświetlenia, czyli czerń, 255 to maksymalne oświetlenie, czyli biel. Nie jest to oczywiste, ale można z taką wykładnią żyć.



Rys. 4. Sposobów rastrowania, czyli zamiany szarych pikseli na odpowiedniej wielkości tablicę pikseli czarno-białych, wymyślono bez liku, bowiem efekt wizualny w istotny sposób zależy od użytego rastra. Powyższa „róża rastrów" demonstruje wygląd tablic rastrowych mogących oddać 64 odcienie szarości: raster (a) został zaczerpnięty z adobowskiego podręcznika języka PostScript ([4], s. 316), raster (b) to popularny raster Bayera ([1]), zaś raster (c) to raster chaotyczny, zwany także stochastycznym. Przykładowe zaczernienia wynoszą 12,5%, 25%, 50% oraz 75%. Można przypuszczać, że na przykład raster Bayera, pozycjonujący kropki możliwie daleko od siebie, na drukarce laserowej będzie ciemne obszary jeszcze bardziej przyciemniał – efektywna kropka drukarki jest większa niż nominalna rozdzielczość i skutkiem tego zamiast 50% zaczernienia uzyskuje się prawie zupełną czerń.

256 odcieni szarości to nie nazbyt dużo jak na czułość oka ludzkiego, niemniej jednak w praktyce nie mamy co liczyć na więcej niż 256 odcieni. Wprawdzie PostScript umożliwia korzystanie aż z 4096 odcieni szarości, ale nie znam programu, który tę możliwość potrafiłby wyzyskać. Niektóre programy obsługujące skanery obrabiają materiał wykorzystując więcej niż 256 odcieni, ale koniec końców użytkownik otrzymuje do ręki grafikę co najwyżej 256-odcieniową.

Problem liczby odcieni szarości wiąże się ściśle z zagadnieniem odtwarzania cieniowanych map bitowych na urządzeniach dwubarwnych, takich jak na przykład fotonaświetlarki, czy czarno-białe drukarki laserowe. Rzecz w tym, że o ile na ekranie można uzyskać jaśniejszy lub ciemniejszy punkt, o tyle klisza w danym punkcie może być albo zaczerniona, albo nie. Jedynym rozwiązaniem jest zastąpienie każdego z pikseli cieniowanej mapy bitowej stosownym fragmentem czarno-białej mapy bitowej. Proces ten nazywa się rasteryzacją, a konkretny sposób zamiany szarych pikseli na tablicę pikseli czarno-białych – rastrem.

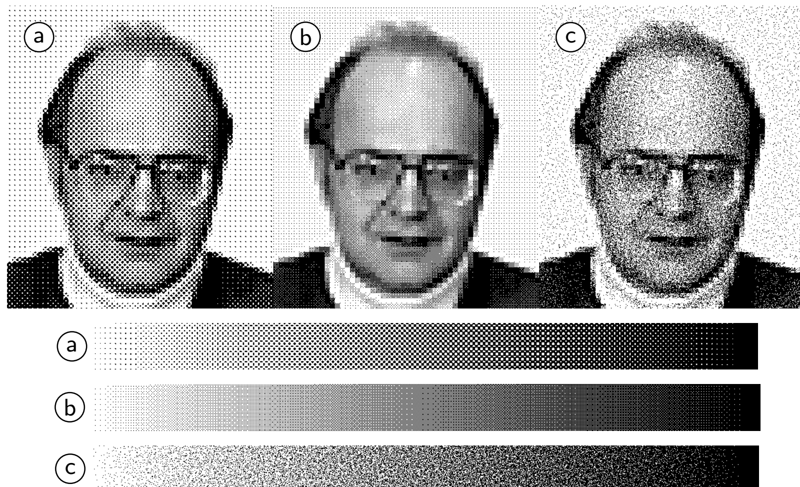
3c. *Rastry, rozdzielczość, liniatura*

Nietrudno policzyć, że aby oddać 256 odcieni szarości potrzebujemy tablicy zbudowanej z 16×16 pikseli czarno-białych. Ściśle mówiąc, tablica taka pozwala na uzyskanie 257 odcieni szarości – od 0 do $16 \times 16 = 256$, ale nieścisłości tego rodzaju nie odgrywają istotnej roli w naszych rozważaniach.

Zastanówmy się, czy drukarka laserowa ma szansę te 256 odcieni szarości dobrze oddać. Przyjmijmy rozdzielczość 300 dpi (dpi = *dots per inch* = liczba kropek na cal). Tablica 16×16 pikseli miałaby w tym wypadku rozmiar około $1{,}4 \times 1{,}4$ mm, co daje bardzo duże, zauważalne gołym okiem ziarno. Liczba tablic rastra przypadająca na jeden cal zwie się zwyczajowo liniaturą (lpi = *lines per inch* = liczba linii na cal). W tym wypadku liniatura wynosiłaby około 19 lpi. Skorzystanie z lepszej drukarki laserowej, mającej rozdzielczość 600 dpi jeszcze znacząco sprawy nie poprawia – liniatura 37 lpi to ciągle za mało, zwłaszcza że najmniejsza kropka drukarek laserowych jest większa niż ich nominalna rozdzielczość, czyli w tym wypadku $^1/_{600}$ cala.

Dopiero fotonaświetlarka, mająca bardzo precyzyjną kropkę i rozdzielczość co najmniej 1200 dpi, pozwala na uzyskanie sensownej jakości. Liniatura w tym wypadku teoretycznie powinna wynosić 75 lpi, ale fotonaświetlarki potrafią, dzięki zmyślnym algorytmom, „udawać” lepszą rozdzielczość.

W niektórych przypadkach można uzyskać zadowalającą głębię szarości przy rozdzielczości 1200 dpi i liniaturze około 100 lpi, co nominalnie odpowiada użyciu elementów rastra wielkości 12×12 pikseli i w zasadzie oznacza ograniczenie się do $12 \times 12 + 1 = 145$ odcieni szarości.



Rys. 5. Do zrastrowania zdjęć oraz tzw. klinów szarości wykorzystane zostały rastry (a), (b) i (c) przedstawione na rysunku 4, przy rozdzielczości czarno-białej mapy bitowej wynoszącej 300 dpi. Widać, że liniatura $300/8 = 37{,}5$ lpi i 64 odcienie szarości to stanowczo za mało jak na wymagania profesjonalne, ale i tak oko zostaje zdumiewająco skutecznie oszukane – fizjonomia Donalda E. Knutha jest w każdym przypadku łatwo rozpoznawalna.

Aby uzyskać naprawdę dobrą jakość, potrzebna jest liniatura 150 lpi, jaką zapewnia dopiero rozdzielczość 2400 dpi. Jaka zatem powinna być rozdzielczość źródłowej szarej mapy bitowej? Wydawać by się mogło, że taka jak liniatura, to znaczy 150 dpi. Okazuje się jednak, że w praktyce dobrze jest zastosować wyższą rozdzielczość, nawet dwukrotnie większą niż liniatura. Wspomniane już sprytne algorytmy fotonaświetlarek potrafią zrobić użytek z tej pozornie nadmiarowej informacji i podnieść jakość w sposób zauważalny.

A teraz zastanówmy się, jaka rozdzielczość byłaby potrzebna, żeby przy liniaturze 150 lpi oddać 4096 odcieni szarości. Tablica elementu rastra mu-

siałaby mieć wielkość 64×64 pikseli, co prowadzi do rozdzielczości 9600 dpi, w praktyce nieosiągalnej.

To tłumaczy, dlaczego większość programów graficznych nie obsługuje większej liczby odcieni szarości niż 256 – współczesna technologia na razie nie umożliwia skutecznego odtwarzania szerszej gamy odcieni. Nie bez znaczenia, rzecz jasna, jest też fakt, że liczby z zakresu $[0, 255]$ mieszczą się w jednym bajcie – usprawnia to znacznie obróbkę danych ze względu na przeważającą współcześnie bajtową architekturę komputerów.

3d. *Kolorowe mapy bitowe – model RGB*

Ludzkie oko ma taką dziwną właściwość, że dowolny postrzegany kolor można zastąpić mieszaniną trzech stosownie dobranych kolorów podstawowych (prawo Grassmana). Cecha ta pozwoliła na skonstruowanie kolorowych monitorów – monitory modelują barwy emitując swiatło będące mieszaniną światła czerwonego, zielonego i niebieskiego; „R", „G" i „B" to początkowe litery angielskich nazw tych kolorów podstawowych: *Red*, *Green* oraz *Blue*.

Zatem od cieniowanych map bitowych jest tylko krok do kolorowych – wystarczy, by elementy macierzy $V_{i,j}$ zdefiniować jako trójki liczb (r, g, b). Każda z liczb r, g, b przyjmuje – jak można się domyślić – wartości 0, 1, $2, \ldots$ 255. Programy interpretujące kolorowe mapy bitowe zakładają na ogół, że poszczególne składowe opisywane są tak jak w przypadku cieniowanych map bitowych – $(0, 0, 0)$ oznacza czerń, $(255, 255, 255)$ – biel. Konwencja ta, podobnie jak konwencja związana z poziomami szarości, bierze się stąd, że 0 oznacza brak emisji światła, a 255 – maksymalną emisję.

Można też myśleć o reprezentacji koloru nie jako o macierzy trójek liczb, a jako o trzech macierzach liczb z zakresu od 0 do 255 $V_{i,j}^{(r)}$, $V_{i,j}^{(g)}$, $V_{i,j}^{(b)}$, odpowiadających „poziomom szarości" barw składowych. Niektóre programy zapisują dane w ten właśnie sposób, inne wolą zapisywać kolejne trójki. PostScript jest w stanie poradzić sobie z oboma rodzajami zapisu, aczkolwiek częściej można spotykać zapis w postaci tablicy trójek.

Przy przetwarzaniu kolorowych map bitowych czasem zdarza się, że trzeba zamienić kolor RGB na szarość. Magiczna formuła opisująca tę zamianę ma postać ([4], s. 304):

$$s = 0{,}3 \times r + 0{,}59 \times g + 0{,}11 \times b \tag{3.1}$$

W praktyce znajomość wzoru (3.1) nie jest potrzebna, gdyż wszystkie programy graficzne mają stosowną „opcję”, wiedza ta jest także wbudowana w niektóre urządzenia (np. fotonaświetlarki). Warto jednak wiedzieć, co się za taką zamianą kryje.

3e. *Kolorowe mapy bitowe – model CMYK*

Nie do końca jest prawdą, że każdy kolor daje się przedstawić wiernie w modelu RGB, jak można by wnosić z poprzedniego punktu.

Model RGB dobrze sprawuje się w przypadku przedmiotów emitujących światło, na przykład monitorów.

Barwa większości przedmiotów, z którymi stykamy się na co dzień, powstaje inaczej – nie przez emisję światła barwnego, a przez pochłanianie niektórych pasm światła białego. Przedmioty, które oświetlone światłem białym (czyli mieszaniną światła czerwonego, zielonego i niebieskiego), pochłaniają kolor czerwony, postrzegamy jako jasnoniebieskie, przedmioty pochłaniające kolor zielony postrzegamy jako purpurowe, przedmioty pochłaniające kolor niebieski – jako żółte.

Podstawowe farby drukarskie zawierają pigmenty pochłaniające światło czerwone (cyjan), zielone (magenta) i niebieskie (żółć). Ponieważ farby te są przezroczyste, więc nałożenie żółci i magenty powinno spowodować pochłonięcie światła niebieskiego i zielonego, czyli powinniśmy zaobserwować, że światło rozpraszane przez tak zadrukowaną powierzchnię ma kolor czerwony. Tak jest w istocie. Podobnie jest w przypadku nałożenia cyjanu i żółci – pochłaniane jest światło czerwone i niebieskie, zatem powierzchnię postrzegamy jako zieloną.

Natomiast nie jest już tak dobrze z kolorem niebieskim – teoretycznie rzecz biorąc, nałożenie magenty i cyjanu powinno spowodować pochłonięcie czerwieni i zieleni, czyli rozpraszana powinna być składowa niebieska. Niestety, kolor, który się otrzymuje, ma odcień raczej fioletowy niż niebieski. Także pokrycie powierzchni cyjanem, magentą i żółcią bynajmniej nie daje koloru czarnego – określiłbym ten kolor w najlepszym razie jako brudnociemnobrązowy.

Okazuje się, że żadna kombinacja cyjanu, magenty i żółci nie da koloru niebieskiego. Potrzebna jest jeszcze domieszka farby czarnej. Oczywiście do uzyskania głębokiej czerni też niezbędna jest farba czarna.

Dopiero cztery farby podstawowe, zwane przez drukarzy „triadą” lub CMYK-iem (od angielskich słów *Cyan*, *Magenta*, *Yellow* i *blacK* – litera „B” jest zarezerwowana dla koloru *Blue*), pozwalają na uzyskanie rozsądnej gamy kolorów, aczkolwiek wciąż istnieją barwy, których w zadowalający sposób za pomocą „triady drukarskiej” odtworzyć się nie da. Na przykład kolor pomarańczowy zawsze wygląda brzydko – jedyny ratunek to użycie farb specjalnych, co niestety znacznie podnosi koszty. Serdecznie polecam studiowanie wzorników farb drukarskich – to może oszczędzić wielu kłopotów spowodowanych niewłaściwym doborem kolorów przy projektowaniu znaków graficznych, okładek itp.

Podobne problemy dotyczą również modelu RGB, ale szczegółowe omówienie zagadnień odtwarzania gamy kolorów i związanych z tym różnych modeli przestrzeni barwnych to tak obszerny temat, że lepiej poprzestańmy na skonstatowaniu, że problem nie jest trywialny i że można się tu natknąć na kłopotliwe sytuacje.

Wartością elementów macierzy $V_{i,j}$ w przypadku CMYK-owych map bitowych jest zatem czwórka liczb (c, m, y, k) z zakresu $[0, 255]$, przy czym w tym wypadku konwencja określająca znaczenie liczb 0 i 255 nawiązuje do drukarstwa: 0 oznacza brak farby, czyli biel (kolor papieru), 255 oznacza 100-procentowe pokrycie danym kolorem. Podobnie jak w przypadku reprezentowania kolorów za pomocą modelu RGB, również CMYK-owe mapy bitowe mogą być pamiętane albo jako macierz czwórek liczb, albo jako cztery macierze o wartościach liczbowych, ale częściej spotyka się pierwszy z tych dwóch wariantów.

Jeśli chodzi o przeliczanie współrzędnych kolorów między modelami RGB i CMYK, to problem jest bardziej skomplikowany niż w przypadku przeliczania RGB → szarość. Powód jest oczywisty – RGB jest modelem trójwymiarowym, CMYK – czterowymiarowym. Trudno w tej sytuacji oczekiwać jednoznacznej formuły, pozwalającej na przeliczanie w obie strony. Prostsza jest oczywiście zamiana CMYK → RGB. W podręczniku Adobe ([4], s. 307) podany jest następujący wzór:

$$\begin{aligned} r &= 1 - \min(1, c + k) \\ g &= 1 - \min(1, m + k) \\ b &= 1 - \min(1, y + k) \end{aligned} \tag{3.2}$$

W źródłach programu `Ghostscript` można znaleźć inny sposób przeliczania:

$$\begin{aligned} r &= (1-c)\times(1-k)\\ g &= (1-m)\times(1-k)\\ b &= (1-y)\times(1-k) \end{aligned} \tag{3.3}$$

Trudno powiedzieć, który ze wzorów jest lepszy. Z pewnością zależy to od konkretnego przypadku. Komuś, kto nie chce zdać się bezmyślnie na efekty działania programów graficznych, które takiej zamiany dokonują na zasadzie „yes-no”, pozostaje tylko metoda prób i błędów.

Znacznie trudniejsza jest zamiana RGB → CMYK. W tym przypadku co lepsze programy graficzne nie poprzestają na wyświetleniu okienka „yes-no”, lecz zostawiają użytkownikowi pole manewru. Rzecz w tym, że zasadniczo arbitralne jest określenie ilości czerni. PostScript pozwala na zdefiniowanie dwóch wielkości, mianowicie BG (*black generation*) i UCR (*undercolor removal*). Pierwsza z nich określa, jaka część barwy ma zostać „przesunięta” do czerni, druga – redukcję ilości farby nakładanej w danym punkcie, gdyż papier źle znosi jej nadmiar.

Podręcznik Adobe ([4], s. 306) podaje następujący przepis na wyznaczanie współrzędnych CMYK, wykorzystujący funkcje UCR i BG:

$$\begin{aligned} c &= \min(1, \max(0, 1-r-\mathrm{UCR}(k')))\\ m &= \min(1, \max(0, 1-g-\mathrm{UCR}(k')))\\ y &= \min(1, \max(0, 1-b-\mathrm{UCR}(k')))\\ k &= \min(1, \max(0, \mathrm{BG}(k'))) \end{aligned} \tag{3.4}$$

gdzie $k' = 1 - \min(1-r, 1-g, 1-b)$. Oczywiście nie jest to jedyna możliwa formuła, na przykład źródła `Ghostscript`-a zawierają alternatywny przepis (znaczenie k' jest takie, jak poprzednio):

$$\begin{aligned} c &= \min\left(1, \max\left(0, 1-\frac{r}{1-\mathrm{UCR}(k')}\right)\right)\\ m &= \min\left(1, \max\left(0, 1-\frac{g}{1-\mathrm{UCR}(k')}\right)\right)\\ y &= \min\left(1, \max\left(0, 1-\frac{b}{1-\mathrm{UCR}(k')}\right)\right)\\ k &= \min(1, \max(0, \mathrm{BG}(k'))) \end{aligned} \tag{3.5}$$

Można też zagadnienie uogólnić i rozważać nie pojedynczą funkcję UCR zależną od jednego parametru, lecz trójkę funkcji zależnych od trzech parametrów: $\mathrm{UCR}^{(r)}(r, g, b)$, $\mathrm{UCR}^{(g)}(r, g, b)$, $\mathrm{UCR}^{(b)}(r, g, b)$. Okazuje się, że w tę stronę poszły niektóre programy graficzne.

Dobór wielkości UCR i BG to osobny problem. Wspomniany podręcznik Adobe ani słowem nie wyjaśnia, czym się kierować w doborze funkcji UCR i BG, zostawiając to użytkownikom. Przy braku innych koncepcji można założyć, że $\mathrm{UCR} \equiv 0$ oraz $\mathrm{BG} \equiv 0$, co spowoduje, że poziom czerni zawsze będzie równy zeru, a tym samym efekty wizualne nie będą najlepsze. Dociekliwym pozostaje jak zawsze metoda prób i błędów.

Zauważmy na koniec, że problem przejścia między różnymi modelami kolorów odnosi się nie tylko do grafiki dyskretnej – w takim samym stopniu dotyczy też grafiki obwiedniowej (p. pkt 2). Jednak ze względu na wagę zagadnienia uznałem, że korzystne będzie choćby zasygnalizowanie tej problematyki w kontekście map bitowych.

3f. *Paletowe mapy bitowe*

Wprowadzenie koloru znacznie zwiększa objętość materiału graficznego – w przypadku modelu RGB trzykrotnie, w przypadku modelu CMYK – czterokrotnie. Czasami jest to niepotrzebną rozrzutnością. Zdarza się bowiem, że ilustracja zawiera faktycznie jedynie kilkadziesiąt kolorów, lub wręcz kilka, co jest częste, gdy chodzi o znaki graficzne, np. logotypy firm.

Powszechną praktyką jest wprowadzenie dodatkowej tablicy P_k, $k = 0$, $1, 2, \ldots p$, zwanej paletą, zawierającej „spis” użytych kolorów. Stosownie do użytego modelu będą to albo liczby całkowite z zakresu $[0, 255]$ (*grayscale*), albo trójki takich liczb (RGB), albo czwórki (CMYK). Tablica $V_{i,j}$ zawiera wskaźniki do tablicy P_k. Innymi słowy, kolor piksela o współrzędnych (i, j) jest określony przez wyrażenie $P_{V_{i,j}}$.

Pozwala to na znaczną redukcję objętości grafiki. Jeśli przykładowo grafika zawiera jedynie 4 kolory, to do zapamiętania wszystkich możliwych wskaźników wystarczą 2 bity, czyli w jednym bajcie mogą się zmieścić 4 wskaźniki. Stwarza to możliwość czterokrotnej redukcji w przypadku cieniowanych map bitowych, w przypadku modelu RGB – dwunastokrotnej, a w przypadku modelu CMYK – szesnastokrotnej.

Dalsza redukcja objętości plików graficznych wymaga zastosowania algorytmów kompresji danych (p. pkt 4).

Rys. 6. Porównanie tej samej ilustracji, przygotowanej z użyciem $p = 8$, 16, 32, 64, 128 oraz 256 kolorów, pokazuje, że pogorszenie jakości może być praktycznie niezauważalne przy umiarkowanej redukcji palety.

3g. *Mapy bitowe z „maską”*

Informacja o pikselu nie kończy się na jego barwie. Istotną cechą – jak już było wspomniane – jest przezroczystość. Aby uwzględnić przezroczystość, wystarczy w tablicy $V_{i,j}$ przechowywać dodatkową informację, czyli o jedną liczbę więcej, niż wynika z zastosowanego modelu: parę liczb dla bitmap cieniowanych, czwórkę dla modelu RGB i piątkę dla modelu CMYK. Programy graficzne, potrafiące tego rodzaju informację zapamiętać, wykorzystują

ją także do zabezpieczenia wyróżnionych obszarów obiektu graficznego przed modyfikacją. Zabieg ten nazywany bywa „maskowaniem”, a owa dodatkowa informacja – „maską” (lub mądrzej „kanałem alfa”).

Niestety, PostScript nie czyni z maski użytku, jedyne co potrafi, to ją pominąć. I nie wydaje się, żeby w ciągu najbliższego czasu sytuacja miała ulec zmianie. Jeszcze raz wypada powtórzyć: a szkoda.

W prostszych przypadkach ograniczenie to można obejść uciekając się do grafiki obwiedniowo-dyskretnej, bowiem PostScript-owe obiekty obwiedniowe mogą mieć przezroczyste „dziury”. Jednakże jest to rozwiązanie zaledwie częściowe, jako że nie da się w ten sposób osiągnąć półprzezroczystości.

Odnotujmy jeszcze, że niektóre dyskretne formaty graficzne zamiast maski specyfikują jeden z kolorów jako przezroczysty – tak reprezentowana jest na przykład przezroczystość w formacie GIF. Należy jednak pamiętać, że nie istnieje powszechnie akceptowany standard definiowania przezroczystości (w odróżnieniu od modeli przestrzeni barwnych, takich jak RGB czy CMYK), w związku z czym może się okazać, że dany program interpretuje grafikę niezgodnie z intencją twórcy.

4. Kompresja danych

Jeżeli skorzystanie z dobrodziejstw palety nie wchodzi w rachubę, rozmiary plików graficznych zaczynają niepokojąco rosnąć. Przypuśćmy, że potrzebujemy wydrukować kolorowe zdjęcie formatu A4. Do uzyskania dobrej jakości potrzebna jest liniatura 150 lpi, co implikuje rozdzielczość zdjęcia 300 dpi (p. s. 9). Format A4 zajmuje powierzchnię około 100 cali kwadratowych, czyli zdjęcie będzie zawierać około $100 \times 300 \times 300 = 9 \times 10^6$ pikseli. Licząc na każdy piksel po cztery bajty (model CMYK, bez maski), otrzymujemy ostatecznie objętość około 35 megabajtów.

Sporo. Bez kompresji ani rusz.

W dalszych rozważaniach dla ustalenia uwagi będziemy zakładać, że mapa bitowa, czyli macierz $V_{i,j}$, jest pamiętana jako ciąg pikseli, q_1, $q_2, \ldots$ $q_{m\times n}$, na przykład w kolejności leksykograficznej: $q_1 = V_{1,1}$, $q_2 = V_{1,2}, \ldots$ $q_n = V_{1,n}$, $q_{n+1} = V_{2,1}$, $q_{n+2} = V_{2,2}, \ldots$ $q_{2n} = V_{2,n}, \ldots$ $q_{m\times(n-1)+1} = V_{m,1}$, $q_{m\times(n-1)+2} = V_{m,2}, \ldots$ $q_{m\times n} = V_{m,n}$.

4a. *Kompresja RLE*

Najprostszy pomysł zasadza się na zauważeniu, że jeżeli dane nie są całkiem chaotyczne, to jest spora szansa, że piksel danego koloru będzie się powtarzał wiele razy pod rząd.

Wystarczy zatem powstawiać w ciąg pikseli co jakiś czas dwa rodzaje znaczników: jeden mówiący, że najbliższy piksel jest powtórzony ileś tam razy, drugi zaś informujący, że kolejne ileś tam pikseli należy traktować dosłownie.

Ten sposób kompresji nosi nazwę RLE, z angielskiego *run length encoding*), co można by tłumaczyć jako *kodowanie bieżących długości.*

Znaczniki to oczywiście zwykłe liczby, na ogół z takiego samego zakresu jak kolory, czyli $[0, 255]$. Należy się jedynie umówić, że na przykład liczby z zakresu $0 \leqslant z \leqslant 127$ oznaczają powtórzenie $z + 1$ razy piksela następującego po znaczniku, a liczby z zakresu $128 \leqslant z \leqslant 255$ oznaczają, że kolejne $z - 127$ pikseli nie zostało zakodowanych. Umówmy się jeszcze, że pierwszym elementem ciągu wynikowego jest znacznik. To wystarcza do jednoznacznej interpretacji zakodowanego ciągu.

Załóżmy dla przykładu, że mamy cieniowaną mapę bitową i że wejściowy ciąg pikseli ma następującą postać:

$$0, 0, 0, 0, 255, 255, 255, 0, 63, 0, 127, 127, 127, 127, 127, 127, 127$$

Możemy go zakodować w następujący sposób (znaczniki dla czytelności zostały podkreślone):

$$\underline{3}, 0, \underline{2}, 255, \underline{130}, 0, 63, 0, \underline{6}, 127$$

Pierwszy znacznik informuje, że wartość 0 ma zostać 4 razy powtórzona (tzn. $3 + 1$), drugi – że wartość 255 należy powtórzyć 3 razy, trzeci – że kolejne 3 piksele nie są kodowane, wreszcie czwarty – że wartość 127 ma być 7 razy powtórzona. W ten sposób długość ciągu została zredukowana o 7 bajtów. Jeśli piksele się powtarzają długimi fragmentami, to można osiągnąć prawie 64-krotną redukcję, gdyż w najlepszym razie zamiast 128 bajtów wstawiamy 2 – znacznik oraz wartość.

Współczynnik redukcji można by poprawić, gdyby znaczniki przechowywane były w dwóch bajtach. Wówczas maksymalna wartość znacznika

zamiast $2^7 = 128$ mogłaby wynosić $2^{15} = 32768$. Jeśli jednak dane noszą charakter przypadkowy, to zwiększanie zakresu znaczników prowadzi jedynie do niepotrzebnego zwiększenia objętości zakodowanych danych. Inteligentne algorytmy kompresji powinny same dobierać zakres wartości znaczników na podstawie danych wejściowych.

To spostrzeżenie nasuwa niepokojące przypuszczenie, że istnieją różne realizacje metody RLE. Zgodnie z oczekiwaniami, odmian algorytmów RLE jest niemal tyle, ile programów z nich korzystających.

Zauważmy na koniec, że chociaż rozkodowanie jest jednoznaczne, to kodować dane można w różny sposób. Dobór optymalnej strategii kodowania nie jest sprawą banalną, ale zgłębianie tego problemu odwiodłoby nas zbyt daleko od zasadniczych zagadnień.

4b. *Kompresja Huffmana*

Przyglądając się uważnie kodowaniu RLE, można się dopatrzeć sporego marnotrawstwa – wszystkie wartości pikseli są traktowane w ten sam sposób, mimo iż niektóre z nich występują częściej niż inne i w związku z tym zasługują na większą uwagę. To spostrzeżenie legło u podstaw sposobu kodowania znanego jako kompresja Huffmana (*Huffman encoding*, [3]).

W tej kompresji dane traktowane są jako strumień bitów. Wynikiem jest też strumień bitów, zawierający kody wejściowych obiektów reprezentowane jako ciągi bitów różnej długości.

Pierwsze pytanie, jakie się nasuwa, brzmi: w jaki sposób ze strumienia bitów w sposób jednoznaczny wyłuskiwać poszczególne kody? Rozwiązaniem są struktury zwane przez informatyków drzewami binarnymi. Brzmi to nader poważnie, ale w istocie sprawa jest prosta i łatwo zrozumieć o co chodzi. Rysunek 7 wyjaśnia w jaki sposób można przypisywać obiektom ciągi zer i jedynek za pomocą drzew binarnych.

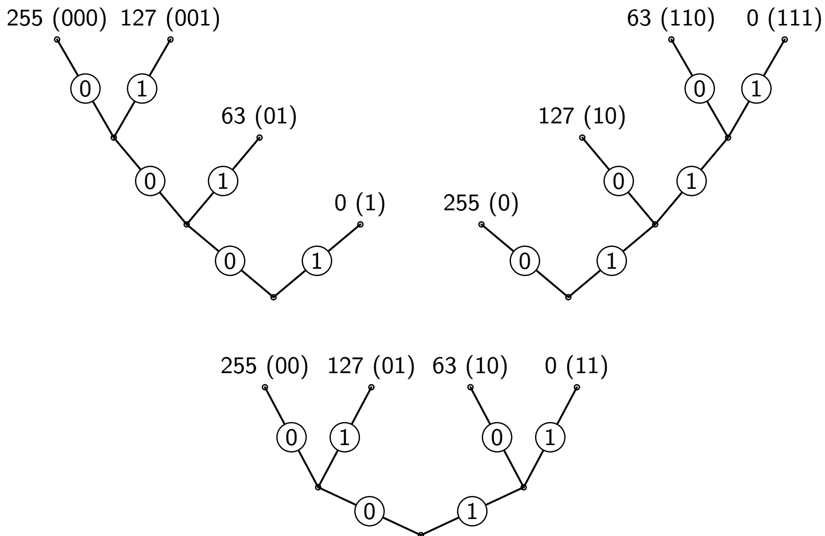
Aby dane odkodować, wystarczy wiedzieć, jak wyglądało drzewo, które posłużyło do zakodowania: ze strumienia wejściowego odczytujemy bity przesuwając się równocześnie po drzewie od czubka do zakończenia gałęzi – po dojściu do zakończenia wiemy, jaki obiekt wystąpił w ciągu wejściowym. Powtarzając postępowanie dekodujemy kolejne obiekty.

Na przykład stosując jedno z przyporządkowań pokazanych na rysunku 7, powiedzmy $255 \to (0)$, $127 \to (10)$, $63 \to (110)$ oraz $0 \to (111)$, możemy

ciąg podany w poprzednim punkcie zapisać jako ciąg zaledwie 38-bitowy, czyli 5-bajtowy:

(111)(111)(111)(111)(0)(0)(0)(111)(110)(111)(10)(10)(10)(10)(10)(10)(10)

Nawiasy oczywiście nie są częścią kodu wynikowego, zostały pozostawione jedynie dla podkreślenia podziału.



Rys. 7. Załóżmy, że w naszym strumieniu danych wejściowych występują jedynie cztery liczby, na przykład 0, 63, 127, 255. Dowolne drzewo binarne mające cztery zakończenia daje nam możliwość przypisania jednoznacznych zero-jedynkowych kodów tym liczbom: gałęzi idącej w lewo przypisujemy wartość 0, gałęzi idącej w prawo – wartość 1. Liczby umieszczamy w zakończeniach gałęzi i przypisujemy im ciąg cyfr dwójkowych (na rysunku podany w nawiasach), jaki otrzymuje się idąc od czubka drzewa do zakończenia gałęzi.

Wynik kompresji jest lepszy niż w poprzednim punkcie – i jest regułą, że kompresja Huffmana jest bardziej efektywna niż kompresja RLE, ale też mniej wygodna w implementacji.

Znaczącą kompresję uzyskaliśmy, mimo iż jeszcze nie zrobiliśmy użytku ze spostrzeżenia poczynionego na wstępie – im częściej występuje dana wartość, tym krótszy kod zero-jedynkowy powinien zostać jej przypisany.

Najkrótszy ciąg wynikowy – jak nietrudno było przewidzieć – daje przyporządkowanie 127 → (0), 0 → (10), 255 → (110) oraz 63 → (111) (lub 63 → (110) oraz 255 → (111), to nie ma znaczenia):

(10)(10)(10)(10)(110)(110)(110)(10)(111)(10)(0)(0)(0)(0)(0)(0)(0)

Tym razem otrzymaliśmy ciąg 31-bitowy, czyli 4-bajtowy. Lepiej się nie da – kto chce niech sprawdzi.

Oczywiście w praktyce funkcjonują różne odmiany metody Huffmana. Na przykład kodowanie wykorzystywane w faksach, opracowane przez Międzynarodowy Komitet Koordynacyjny ds. Telefonii i Telegrafii (CCITT), wykorzystuje interesujący wariant „skrzyżowania" metody RLE i metody Huffmana. Mianowicie słownik kodów jest ustalony raz na zawsze, ale kodowane są nie bajty, tylko różnej długości ciągi czarnych i białych pikseli (kompresja faksowa dotyczy tylko czarno-białych map bitowych), czyli znane już nam „bieżące długości".

Początkowy fragment słownika kodów dla kompresji faksowej wygląda następująco ([5]):

długość ciągu pikseli	kody huffmanowskie dla koloru białego	kody huffmanowskie dla koloru czarnego
1	000111	010
2	0111	11
3	1000	10
4	1011	011
5	1100	0011
6	1110	0010
7	1111	00011
8	10011	000101
9	10100	000100
10	00111	0000100
11	01000	0000101
12	001000	0000111
...	...	...
63	00110100	000001100111

Z metodą Huffmana można się często zetknąć w praktyce, chociaż funkcjonuje ona z reguły jako metoda uzupełniająca, w związku z czym łatwo można przeoczyć jej obecność. Korzystają z niej na przykład programy takie

jak `ARJ` czy `PKZIP`, wykorzystywana jest też na niektórych etapach kompresji stosowanej w plikach `JPG`.

4c. *Kompresja LZW*

Najbardziej popularnym algorytmem kompresji – po części pewnie z powodu wrzawy wokół rzadkiego w tej branży opatentowania – jest algorytm LZW. Nazwa pochodzi od nazwisk autorów: L – Abraham Lempel, Z – Jacob Ziv, W – Terry A. Welch. Ten ostatni nieco usprawnił opublikowany wcześniej przez Lempela i Ziva algorytm ([7], [8]), po czym firma Unisys, w której pracował Welch, uzyskała dla zmodyfikowanego algorytmu patent Urzędu Patentowego USA. Od tej pory każdy, kto chce wykorzystać w swoim oprogramowaniu algorytm LZW, musi uiścić opłatę firmie Unisys, podobno niewielką...

Kluczowym chwytem w kompresji LZW jest budowanie słownika najczęściej powtarzających się słów, ale w inny sposób niż przy kompresji Huffmana – w tym wypadku słownik budowany jest *a vista* w trakcie przetwarzania strumienia danych wejściowych. Wynikiem jest ciąg numerów haseł w słowniku, czyli wskaźników (kodów).

Najłatwiej prześledzić zasadę działania algorytmu biorąc jako model kompresję tekstów, dlatego na moment odejdziemy od map bitowych, pamiętając wszak, że tekst to dla komputera nic innego jak strumień liczb jednobajtowych.

Idea kompresji daje się streścić następująco: w każdym momencie trzymamy w „rękach” ostatnio przeczytany znak i oraz poprzedzający ten znak najdłuższy możliwy ciąg znaków stanowiący „hasło”, tzn. znajdujący się w słowniku, przy czym zaczynamy od słownika, w którym znajdują się wszystkie pojedyncze znaki (musimy z góry wiedzieć, co się może pojawić na wejściu). Jeśli hasło przedłużone o ostatnio przeczytany znak znajduje się w słowniku, to tę pozycję słownikową uznajemy za nowe hasło; jeśli przedłużenie dałoby hasło nie należące do słownika, to wskaźnik dotychczasowego hasła wysyłamy do strumienia wyjściowego, hasło przedłużone o ostatnio przeczytany znak dodajemy do słownika, a ostatnio przeczytany znak (na pewno jest w słowniku) uznajemy za nowe hasło; następnie wczytujemy kolejny znak i powtarzamy postępowanie.

Strumień wyjściowy zawiera jedynie wskaźniki, czyli numery haseł w słowniku, ale samego słownika *nie zawiera* – algorytm dekodujący bę-

dzie w stanie ten słownik odtworzyć. Oczywiście musi być znana też wielkość słownika, czyli liczba bitów przeznaczonych na pojedyncze wskaźniki.

Stosując półformalną notację, algorytm kompresji LZW można zapisać nieco precyzyjniej:

Kompresja LZW – kodowanie

dane: *strumień wejściowy* q_1, q_2, $q_3, \ldots$ *(składający się ze znaków a, b, c, ...) oraz słownik* S_0, $S_1, \ldots S_s$ *(zawierający jednoznakowe słowa a, b, c, ...)*

zmienne: H *– słowo składające się ze znaków poprzedzających znak ostatnio przeczytany (włącznie z tym znakiem), znajdujące się w słowniku (tzn.* $H = S_K$ *dla pewnego* K*)*

wynik: *strumień wskaźników (kodów), tzn. liczb z zakresu* $[0, s]$ *oraz ewentualnie wielkość słownika* s

inicjalizacja:

$S_0 \leftarrow a$, $S_1 \leftarrow b$, $S_2 \leftarrow c, \ldots$ $H \leftarrow q_1$

główny algorytm:

dopóki *w strumieniu wejściowym są znaki* **wykonuj:**

pobierz kolejny znak q_i $(i \geqslant 2)$ *ze strumienia wejściowego*

jeśli $H\,q_i = S_K$ *dla pewnego* K **to:**

$H \leftarrow S_K$

w przeciwnym razie:

wyślij do strumienia wyjściowego liczbę K *taką, że* $S_K = H$

wpisz do słownika hasło $H\,q_i$ *(na końcu listy haseł)*

$H \leftarrow q_i$

powtarzaj

zakończenie: *(po wyczerpaniu strumienia wejściowego)*

wyślij do strumienia wyjściowego liczbę K *taką, że* $S_K = H$

Zaprezentowana tu wersja algorytmu kodującego zawiera jedno ważne uproszczenie natury technicznej, bez uwzględnienia którego rzeczywiste implementacje algorytmu by nie działały. Otóż przy dłuższych danych w słowniku w pewnym momencie nieuchronnie zabraknie miejsca, bowiem wysłanie kodu do strumienia wyjściowego wiąże się zawsze z uszczupleniem

wolnego miejsca w słowniku. Ta uciążliwa cecha, jak za chwilę zobaczymy, okaże się kluczem do odkodowania.

Ale w praktyce coś trzeba z tym fantem zrobić. „Prawdziwy” algorytm LZW wstawia do ciągu wyjściowego specjalny znacznik, rozpoznawany przez algorytm dekodujący, po czym przywraca stan początkowy wszystkim strukturom danych i kompresja w pewnym sensie zaczyna się od początku. Można oczywiście użyć ogromnego słownika, ale wówczas w niektórych przypadkach strumień wynikowy będzie zawierał liczby z niepotrzebnie dużego zakresu.

Zobaczmy na bardzo prostym przykładzie jak działa kodowanie. Poniższa tabela opisuje stan zmiennych w kolejnych krokach na końcu każdego obrotu pętli oraz – po zakończeniu wykonywania pętli – tuż przed zatrzymaniem algorytmu. Dla uproszczenia przyjęte zostało, że strumień danych wejściowych zawiera jedynie litery a, b i c i że słownik zawierać może co najwyżej 256 elementów, czyli że na wskaźniki przeznaczony został jeden bajt.

Kompresja LZW – symulacja kodowania dane wejściowe: *abacccab*		
krok 0	$q_1 = b$ $H = a$	$S_0 = a,\ S_1 = b,\ S_2 = c$ wynik: *nic*
krok 1	$q_2 = b$ $H = b$	$S_0 = a,\ S_1 = b,\ S_2 = c,\ S_3 = ab$ wynik: 0 (tj. a)
krok 2	$q_3 = a$ $H = a$	$S_0 = a,\ S_1 = b,\ S_2 = c,\ S_3 = ab,\ S_4 = ba$ wynik: 0 1 (tj. $a\ b$)
krok 3	$q_4 = c$ $H = c$	$S_0 = a,\ S_1 = b,\ S_2 = c,\ S_3 = ab,\ S_4 = ba,\ S_5 = ac$ wynik: 0 1 0 (tj. $a\ b\ a$)
krok 4	$q_5 = c$ $H = c$	$S_0 = a,\ S_1 = b,\ S_2 = c,\ S_3 = ab,\ S_4 = ba,\ S_5 = ac,\ S_6 = cc$ wynik: 0 1 0 2 (tj. $a\ b\ a\ c$)
krok 5	$q_6 = c$ $H = cc$	$S_0 = a,\ S_1 = b,\ S_2 = c,\ S_3 = ab,\ S_4 = ba,\ S_5 = ac,\ S_6 = cc$ wynik: 0 1 0 2 (tj. $a\ b\ a\ c$)
krok 6	$q_7 = a$ $H = a$	$S_0 = a,\ S_1 = b,\ S_2 = c,\ S_3 = ab,\ S_4 = ba,\ S_5 = ac,\ S_6 = cc,\ S_7 = cca$ wynik: 0 1 0 2 6 (tj. $a\ b\ a\ c\ cc$)
krok 7	$q_8 = b$ $H = ab$	$S_0 = a,\ S_1 = b,\ S_2 = c,\ S_3 = ab,\ S_4 = ba,\ S_5 = ac,\ S_6 = cc,\ S_7 = cca$ wynik: 0 1 0 2 6 (tj. $a\ b\ a\ c\ cc$)
krok 8		$S_0 = a,\ S_1 = b,\ S_2 = c,\ S_3 = ab,\ S_4 = ba,\ S_5 = ac,\ S_6 = cc,\ S_7 = cca$ wynik: 0 1 0 2 6 3 (tj. $a\ b\ a\ c\ cc\ ab$)

Uważniejsze przyjrzenie się symulacji kodowania pozostawiam czytelnikowi.

Zastanówmy się, w jaki sposób mogłoby przebiegać dekodowanie. Słownika wprawdzie nie znamy, ale wiemy, jak powstawał. Wiemy też, że pierwszy element zakodowanego strumienia zawiera wskaźnik do pojedynczego znaku, zatem pierwszy znak potrafimy odcyfrować.

Odnotowaliśmy także, że wysłaniu liczby do strumienia wyjściowego towarzyszy powiększenie słownika o kolejną pozycję. Rzućmy jeszcze raz okiem na to, jak to się dzieje: hasło będące kandydatem do wysłania jest przedłużane przez dopisywanie na końcu kolejno czytanych znaków tak długo, jak długo po przedłużeniu otrzymujemy słowo znajdujące się w słowniku; jednak w końcu przychodzi taki znak, który doczepiony na końcu dotychczas uzbieranego hasła dałby słowo, którego w słowniku nie ma; w tym momencie dotychczasowy kandydat jest „wypychany" do strumienia wyjściowego przez znak, który się był pojawił, a w słowniku ląduje słowo dopiero co wysłane z doczepionym na końcu znakiem, który spowodował zamieszanie.

Obserwacja ta pozwala na odtwarzanie słownika krok po kroku podczas dekodowania. Załóżmy bowiem, że odkodowaliśmy wskaźniki $r_1, r_2, \ldots r_{n-1}$, a w kolejce czeka na odkodowanie wskaźnik r_n:

$$\underbrace{\boxed{a_1\, a_2 \ldots a_{k_a}}}_{r_1}\ \underbrace{\boxed{b_1\, b_2 \ldots b_{k_b}}}_{r_2}\ \ldots\ \underbrace{\boxed{c_1\, c_2 \ldots c_{k_c}}}_{r_{n-1}}\ \underbrace{\boxed{d_1 \vdots d_2 \ldots d_{k_d}}}_{r_n}$$

Znakiem odpowiedzialnym za „wypchnięcie" wskaźnika r_{n-1} był pierwszy znak słowa kryjącego się pod kodem r_n, czyli znak d_1, co oznacza, że w momencie wysłania kodu r_{n-1} do strumienia wynikowego słownik został poszerzony o słowo $c_1\, c_2 \ldots c_{k_c}\, d_1$.

Oczywiście słowo o wskaźniku równym r_{n-1} znajduje się w dotychczas odtworzonym słowniku (skorośmy je odkodowali).

Jeśli mamy szczęście i słowo o wskaźniku równym r_n także jest już w słowniku, to znamy jego pierwszy znak, czyli znamy też słowo $c_1\, c_2 \ldots c_{k_c}\, d_1$, a więc potrafimy odtworzyć stan słownika algorytmu kodowania w momencie wysyłania liczby r_{n-1} do strumienia wynikowego.

A jeśli w słowniku nie ma słowa o wskaźniku r_n? Zauważmy, że słowo o wskaźniku $r_n - 1$ z pewnością znajduje się w słowniku, gdyż hasła dokładane są do słownika systematycznie i umieszczane w pierwszym wolnym

miejscu. Stąd wniosek, że wskaźnik r_n odwołuje się do *pierwszej wolnej pozycji w słowniku*, czyli do słowa $c_1\,c_2\,\ldots\,c_{k_c}\,d_1$, które – jak już wiemy – musiało zostać wpisane w to miejsce w trakcie kodowania. W takim razie wiemy, że r_n jest kodem słowa $c_1\,c_2\,\ldots\,c_{k_c}\,d_1$. Gdybyśmy tylko znali literę d_1! Ale litera d_1 jest znana, zachodzi bowiem równość

$$c_1\,c_2\,\ldots\,c_{k_c}\,d_1 = d_1\,d_2\,\ldots\,d_{k_d}$$

z której wynika, że d_1 to po prostu c_1. Wynika też, że $d_1 = d_{k_d}$, ale to już jest mniej ważne.

I to wszystko. Zapisany w tej samej półformalnej konwencji algorytm dekodowania przedstawia się następująco:

Kompresja LZW – dekodowanie

dane: *strumień wejściowy* $r_1, r_2, r_3, \ldots$ *składający się z liczb z zakresu* $[0, s]$ *(wskaźników)*
słownik $S_0, S_1, \ldots S_s$ *zawierający ciągi znaków* a, b, $c, \ldots$
zmienne: K *– ostatnio przeczytany wskaźnik*

inicjalizacja:
$S_0 \leftarrow a$, $S_1 \leftarrow b$, $S_2 \leftarrow c, \ldots$
$K \leftarrow r_1$ *(w tym momencie K jest kodem pojedynczego znaku)*

główny algorytm:
dopóki *w strumieniu wejściowym są liczby* **wykonuj:**
  wyślij S_K do strumienia wyjściowego (hasło S_K jest znane)
  pobierz kolejny wskaźnik r_i ($r_i \geqslant 2$) ze strumienia wejściowego)
  jeśli *hasło S_{r_i} jest znane* **to:**
    dodaj do słownika (na końcu listy haseł) konkatenację postaci: S_K pierwszy_znak(S_{r_i})
  w przeciwnym razie:
    (r_i wskazuje na pierwszą wolną pozycję w słowniku)
* $S_{r_i} \leftarrow S_K$ pierwszy_znak(S_K)
  $K \leftarrow r_i$
powtarzaj

zakończenie *(po wyczerpaniu strumienia wejściowego)*
  wyślij S_K do strumienia wyjściowego

Gwiazdką została opatrzona instrukcja, do której algorytm odwołuje się w momencie natrafienia na kod, którego jeszcze nie ma w słowniku. Poniższa tabela zawiera zapis symulacji odkodowania dopiero co zakodowanego ciągu znaków. Krok 4, w którym algorytm robi użytek z instrukcji oznaczonej gwiazdką, został także wyróżniony gwiazdką.

Kompresja LZW – symulacja dekodowania dane wejściowe: 0, 1, 0, 2, 6, 3		
krok 0	$K = 0$	$S_0 = a,\ S_1 = b,\ S_2 = c$ wynik: *pusty*
krok 1	$K = 1$	$S_0 = a,\ S_1 = b,\ S_2 = c,\ S_3 = ab$ wynik: a
krok 2	$K = 0$	$S_0 = a,\ S_1 = b,\ S_2 = c,\ S_3 = ab,\ S_4 = ba$ wynik: $a\ b$
krok 3	$K = 2$	$S_0 = a,\ S_1 = b,\ S_2 = c,\ S_3 = ab,\ S_4 = ba,\ S_5 = ac$ wynik: $a\ b\ a$
krok 4*	$K = 6$	$S_0 = a,\ S_1 = b,\ S_2 = c,\ S_3 = ab,\ S_4 = ba,\ S_5 = ac$ wynik: $a\ b\ a\ c$
krok 5	$K = 3$	$S_0 = a,\ S_1 = b,\ S_2 = c,\ S_3 = ab,\ S_4 = ba,\ S_5 = ac,\ S_6 = cca$ wynik: $a\ b\ a\ c\ cc$
krok 6		$S_0 = a,\ S_1 = b,\ S_2 = c,\ S_3 = ab,\ S_4 = ba,\ S_5 = ac,\ S_6 = cca$ wynik: $a\ b\ a\ c\ cc\ ab$

Na pierwszy rzut oka mogłoby się wydawać, że z tym algorytmem LZW to wiele hałasu o nic – taki skomplikowany algorytm, a skutki bynajmniej nie powalają na kolana. Wprawdzie udało się skompresować dane (na wejściu jest 8 liter, na wyjściu – 6 liczb), ale zysk to niewielki.

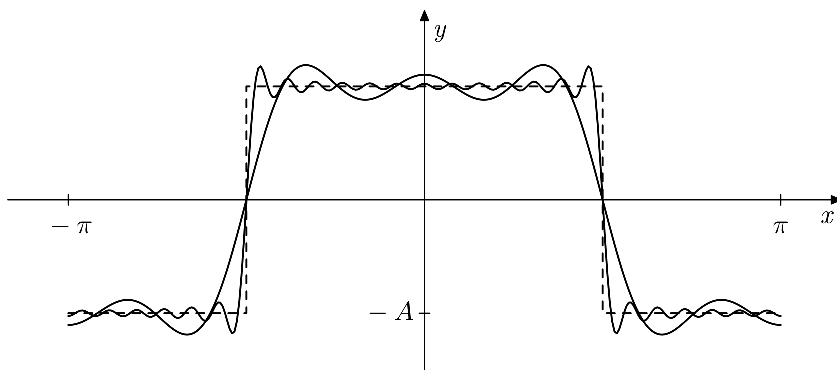
Rzecz w tym, że algorytm LZW długo się „rozpędza" i w słowniku muszą się znaleźć dziesiątki haseł, zanim kompresja zacznie naprawdę działać. Jest to na tyle skuteczna metoda, że od czasu, kiedy została opublikowana (1984), mimo licznych prób nie udało się jej w sposób istotny poprawić. A ponadto metody kompresji słownikowej zbliżone do kompresji LZW są tak szeroko rozpowszechnione, że do atrybutów ogłady towarzyskiej zaliczyłbym znajomość zasad, na których oparty jest algorytm LZW.

Lepszy współczynnik kompresji dają jedynie tzw. metody stratne, które uzyskują zdumiewająco dobre efekty kosztem pominięcia – z reguły trafnie – części informacji.

4d. *Kompresja JPEG*

Jedną z szerzej rozpowszechnionych metod kompresji stratnej jest metoda JPEG, opublikowana przez organizację Joint Photographic Experts Group (od której wzięła swą nazwę).

Pomysł kompresji JPEG sięga korzeniami do matematycznej techniki, znanej jako analiza fourierowska, a polegającej na przybliżaniu funkcji za pomocą wielomianów trygonometrycznych.



Rys. 8. Funkcję „schodkową”, zaznaczoną na rysunku linią przerywaną, można przedstawić w postaci nieskończonego szeregu Fouriera jako:

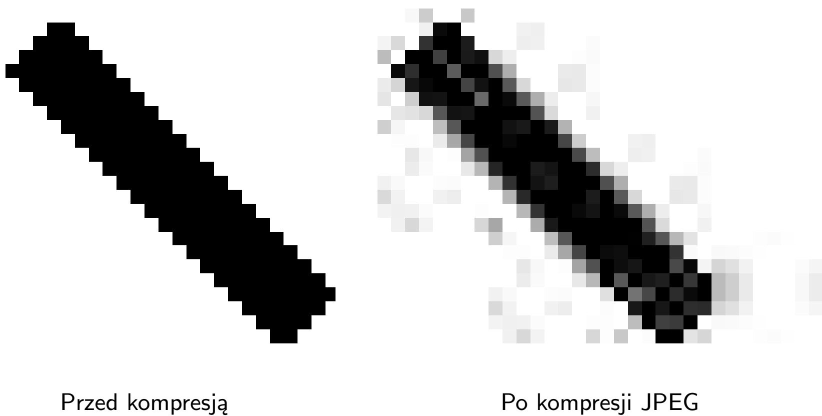
$$y(x) = \frac{A}{2} + \frac{2A}{\pi}\left(\cos x - \frac{1}{3}\cos 3x + \frac{1}{5}\cos 5x - \frac{1}{7}\cos 7x + \ldots\right)$$

Ograniczając się do skończonej liczby wyrazów szeregu (na ilustracji linia ciągła), otrzymuje się przybliżenie z charakterystycznymi „falkami”, tym drobniejszymi, im więcej wyrazów zostało uwzględnionych. Im gładsza funkcja, tym mniej widoczne stają się artefakty wynikające z niedokładności aproksymacji.

Przybliżanie z nieskończoną dokładnością wymaga nieskończonego wielomianu, który matematycy lubią nazywać szeregiem. Operacja obliczania współczynników szeregu Fouriera nosi nazwę przekształcenia (lub transformacji) Fouriera.

Użyteczność rozwijania w szereg Fouriera bierze się stąd, że współczynniki mają łatwo zrozumiałe intuicyjnie znaczenie. Jeśli daną funkcję potraktować jako wykres drgań, to rozwinięcie w szereg Fouriera odpowiada rozbiciu drgania na składowe harmoniczne, a współczynniki Fouriera to po prostu amplitudy poszczególnych składowych.

W otaczającym nas świecie jakoś tak jest, że w wielu zjawiskach wyższe składowe harmoniczne odgrywają mniejszą rolę, w związku z tym można je na ogół pominąć. Na ogół, ale nie zawsze. Rysunek 8 pokazuje, kiedy przybliżenie fourierowskie się nie sprawdza: jeśli istnieją punkty, w których następuje gwałtowna zmiana funkcji, to w okolicach takich miejsc pojawiają się zaburzenia. Efekty przypominające „falki” z rysunku 8 można często zobaczyć na grafice poddanej kompresji JPEG. Rysunek 9 przedstawia powiększony fragment grafiki, wykazujący zakłócenia tego rodzaju.



Rys. 9. Kompresja JPEG jest zalecana w przypadku obrazów o kolorach zmieniających się w sposób ciągły. Gwałtowna zmiana koloru powoduje powstanie artefaktów, pogarszających zdecydowanie jakość obrazu. Źródłem tych artefaktów są matematyczne własności transformaty Fouriera. Przyjrzawszy się uważniej zaburzeniom, zauważymy, że wyraźnie mają one strukturę „kafelkową”, biorącą się z zastosowanego w metodzie JPEG podziału obrazu na kwadraty o rozmiarach 8×8 pikseli.

Podobieństwo zastosowanej w metodzie JPEG dyskretnej transformacji cosinusowej (DCT – *Discrete Cosine Transform*) do rozwijania w szereg Fouriera nie jest bynajmniej powierzchowne, aczkolwiek na pierwszy rzut oka analogie mogą być trudne do zauważenia, zwłaszcza że analiza fourierowska dotyczy przebiegów ciągłych.

Transformacja DCT obliczana jest dla kwadratów („kafelków”) wielkości 8×8 pikseli. Poszczególne składowe kolorów podlegają kompresji niezależnie. Oznaczmy przez $q_{x,y}$ wartość koloru piksela o współrzędnych

(x, y), przy czym umawiamy się, że współrzędne liczymy lokalnie w każdym z kafelków i że możliwymi wartościami x, y są 0, 1, 2,... 7. Przy tych założeniach przekształcenie DCT dla pojedynczego kafelka opisuje następujący wzór:

$$Q_{u,v} = \frac{1}{4} C_u C_v \sum_{x=0}^{7} \sum_{y=0}^{7} \big(q_{x,y} - 128\big) \cos \frac{\pi u (2x+1)}{16} \cos \frac{\pi v (2y+1)}{16} \tag{4.1}$$

gdzie $C_u = 1/\sqrt{2}$ dla $u = 0$, a w przeciwnym razie $C_u = 1$; podobnie jest określona wartość C_v. Oczywiście zakładamy, że $q_{x,y} \in [0, 255]$. Kolory poszczególnych pikseli można odtworzyć ze wzoru bliźniaczo podobnego do wzoru (4.1).

$$q_{u,v} = 128 + \frac{1}{4} \sum_{u=0}^{7} \sum_{v=0}^{7} C_u C_v Q_{u,v} \cos \frac{\pi u (2x+1)}{16} \cos \frac{\pi v (2y+1)}{16} \tag{4.2}$$

Pozornie pomienialiśmy siekierkę na kijek: macierz 8×8 liczb całkowitych zamieniliśmy na macierz 8×8 liczb rzeczywistych.

Kompresji ni śladu.

W sukurs przychodzi interpretacja przekształcenia Fouriera. Na elementy macierzy $Q_{u,v}$ można patrzeć jako na amplitudy składowych harmonicznych. Należy więc oczekiwać, że w przypadku danych wykazujących regularność decydować będą składowe niskookresowe, czyli że wyraźnie większe amplitudy powinny się pojawiać w pobliżu punktu $(0, 0)$.

Analiza teoretyczna jest oczywiście poza naszym zasięgiem, ale zawsze możemy posłużyć się przykładem. Załóżmy, że mamy do czynienia z grafiką z łagodnymi przejściami kolorów. Tym samym możemy zakładać, że nad kafelkami – małymi przecież fragmentami mapy bitowej – funkcja $q_{x,y}$ układa się dość regularnie. Założenie to jest oparte na powszechnym przekonaniu, że większość zjawisk w przyrodzie daje się opisywać za pomocą funkcji gładkich. Typowe przykłady takiego założenia można znaleźć w wielu podręcznikach fizyki, gdzie obficie wykorzystywane jest założenie, że na odpowiednio krótkim odcinku funkcja liniowa dobrze przybliża funkcję gładką. My nie będziemy żądać aż tak wiele – okazuje się, że wystarcza rozkład kolorów zbliżony do funkcji wielomianowej niskiego stopnia.

Niech zatem przykładowo rozkład odcieni szarości nad danym kafelkiem opisuje tablica:

	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*
0	255	250	243	234	222	205	179	96
1	219	216	212	207	198	186	165	96
2	182	182	181	179	174	166	151	96
3	146	149	151	151	150	147	138	96
4	109	115	120	124	127	127	124	96
5	73	81	89	96	103	108	110	96
6	36	47	58	69	79	88	96	96
7	0	14	28	41	55	69	83	96

Transformacja DCT przeprowadza powyższą tablicę w macierz współczynników transformacji cosinusowej:

	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*
0	10,37	43,25	–74,03	39,88	–35,12	21,85	–15,74	7,06
1	422,67	191,34	–47,74	42,79	–22,40	19,31	–10,22	5,34
2	0,07	0,02	–0,09	–0,05	0,07	0,08	–0,04	–0,09
3	44,22	19,90	–5,31	4,55	–2,06	1,74	–1,24	0,86
4	0,63	0,16	0,30	–0,25	–0,88	–0,08	–0,26	0,47
5	12,72	5,89	–1,32	1,19	–0,77	0,54	–0,31	0,20
6	–0,16	–0,05	0,21	0,13	–0,16	–0,19	0,09	0,23
7	2,76	1,29	–0,90	–0,11	0,38	0,04	–0,47	0,07

Na pierwszy rzut ten galimatias liczb wygląda mało zachęcająco. Ale po uważniejszym przyjrzeniu się dojdziemy do wniosku, że nasza supozycja okazała się słuszna: duże amplitudy występują w pobliżu narożnika $(0, 0)$.

I tu dochodzimy do bardzo ważnego momentu algorytmu kompresji JPEG, mianowicie do *kwantyzacji amplitud*. Na tym właśnie etapie dane są upraszczane, czyli *tracona jest informacja*: wartości amplitud są zaokrąglane do najbliższej wielokrotności pewnej ustalonej z góry liczby całkowitej. W świetle tego, co zostało powiedziane o własnościach przekształcenia Fouriera, można przypuszczać, że niewielkie zmiany amplitud zaowocują niedostrzegalnymi zmianami obrazu po zastosowaniu transformacji odwrotnej. W istocie możemy sobie pozwolić na całkiem spore zmiany amplitud.

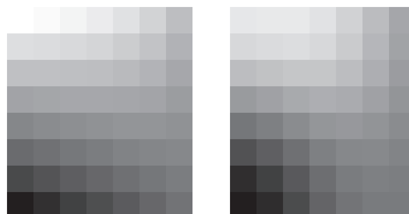
Poniższa tablica zawiera amplitudy zaokrąglone do wielokrotności liczby 100:

	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*
0	0	0	–100	0	0	0	0	0
1	400	200	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0

Ta brutalna zmiana amplitud nie wpływa aż tak bardzo na wynik transformacji odwrotnej, jak można by się spodziewać:

	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*
0	229	231	231	223	204	177	150	133
1	211	215	217	211	195	170	145	130
2	178	184	189	189	178	159	137	124
3	135	143	154	160	156	143	127	116
4	88	99	116	129	132	126	116	107
5	45	59	80	100	110	111	105	100
6	12	28	53	77	94	99	97	94
7	0	11	38	65	85	93	93	90

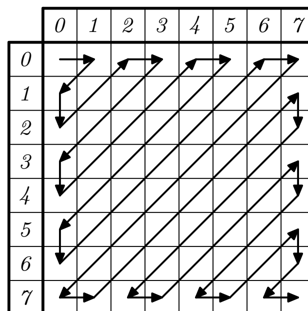
Liczby bezlitośnie ujawniają wszystkie niedostatki metody, ale porównanie wyglądu kafelka oryginalnego (po lewej) z uzyskanym z zaokrąglonych amplitud przez zastosowanie odwrotnej transformacji DCT (po prawej) pokazuje, że istota zmian kolorystyki została zachowana:



Kwantyzacja decyduje o kompresji JPEG: zamiast pamiętać 64 poziomów szarości, możemy zapamiętać 3 amplitudy – dałoby to kompresję około dwudziestokrotną. W praktyce kompresja bywa nie aż tak duża, bo nie stosuje się tak drastycznego zaokrąglania głównych amplitud, natomiast różne amplitudy są kwantyzowane w różnym stopniu – mniej istotne nawet ze współczynnikami większymi niż 100. Przykładowa tablica, określająca do wielokrotności jakich liczb mają być zaokrąglane poszczególne amplitudy (zaczerpnięta z [6]), wygląda następująco:

	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*
0	16	12	14	14	18	24	49	72
1	11	10	16	24	40	51	61	12
2	13	17	22	35	64	92	14	16
3	22	37	55	78	95	19	24	29
4	56	64	87	98	26	40	51	68
5	81	103	112	58	57	87	109	104
6	121	100	60	69	80	103	113	120
7	103	55	56	62	77	92	101	99

To już prawie wszystko. Jeszcze należy wspomnieć, że amplitudy są poddawane kompresji Huffmana, przy czym przed kompresją nie są ustawiane leksykograficznie, ale wzdłuż zygzaka, uwzględniającego w przybliżony sposób „ważność" amplitud:



Jak widać, nawet przedstawienie zasady kompresji JPEG wymagało dotknięcia, choćby pobieżnego, technicznych detali. Sukces formatu JPEG jest między innymi efektem uwzględnienia wielu szczególnych przypadków. Właściwie kompresja JPEG to zestaw kilku metod kompresji, dobieranych

w zależności od konkretnego przypadku i zadanych parametrów. Zainteresowani technicznymi aspektami metody JPEG mogą znaleźć jej drobiazgowy opis na przykład w publikacjach [5], [6] oraz [2].

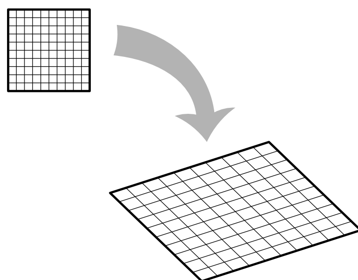
5. Strefy zwiększonego ryzyka

W punkcie 2 wspomniałem, że poddawanie map bitowych geometrycznym przekształceniom napotyka na kłopoty zasadniczej natury. Każdy, kto zabiera się za przekształcanie map bitowych, powinien mieć świadomość, że to niebezpieczne rejony. Potrzebne jest spore doświadczenie i specjalizowane oprogramowanie.

Ale co zrobić, jak i na jednym, i na drugim nam nie zbywa, a tu trzeba na przykład powiększyć ilustrację? Na szczęście PostScript został wyposażony w możliwości, z których w trudnych sytuacjach można skorzystać. Warto jednak wiedzieć, czym to grozi.

5a. *Przekształcenia afiniczne i nie tylko*

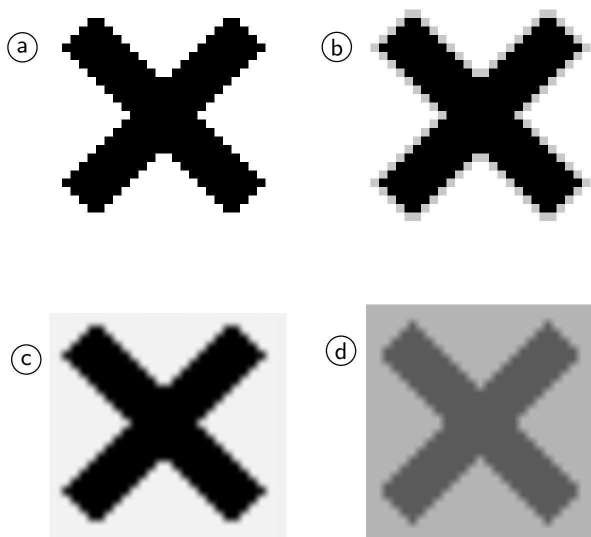
Skalowanie wszystkich obiektów (grafiki dyskretnej, obwiedniowej, tekstu) to chleb powszedni PostScript-u. W istocie PostScript może dokonywać na wszystkich tych obiektach tzw. przekształceń afinicznych, czyli takich przekształceń płaszczyzny, które przeprowadzają kwadraty w równoległoboki:



Póki mapa bitowa, którą przekształcamy, ma odpowiednio dużą rozdzielczość (co to znaczy „odpowiednio”, zależy od konkretnego przypadku), póty możemy polegać na algorytmach PostScript-owych – wynik będzie poprawny. Jeśli natomiast rozdzielczość jest zbyt mała, to może się pojawić wyraźna struktura pikselowa.

Czasem jest to efekt zamierzony, jak na przykład na rysunku 9 (s. 28), ale w ogólnym przypadku raczej jest to zjawisko niepożądane.

Programy graficzne stosują trick, zwany *antyaliasingiem*, po to, by piksele stały się jak najmniej zauważalne. Ta dziwna nazwa bierze się stąd, że „piłokształtny” wygląd linii ukośnych, odtwarzanych na urządzeniach dyskretnych, jest określany jako *alias*, czyli coś, co jest zamiast tego, co być powinno. Wydaje mi się, że w tym wypadku można nie bawić się w wierne tłumaczenie terminu angielskiego i używać dobrze oddającego istotę rzeczy określenia „wygładzanie”.



Rys. 10. Wygładzanie, zwane antyaliasingiem, polegające na dostawianiu na granicy kolorów pikseli o kolorze pośrednim, działa w zasadzie tylko wówczas, gdy piksele są małe. Figura (b) to po prostu figura (a) poddana wygładzaniu (oryginalna wielkość ok. 2 mm przy rozdzielczości 300 dpi). Po powiększeniu nadal wyraźnie widać strukturę pikselową. Można sobie zażyczyć, by POSTSCRIPT przy skalowaniu zastosował wygładzanie oparte na tej samej zasadzie (w świecie POSTSCRIPT-owym zwane „interpolacją”). Figura (c) ilustruje wynik powiększania z interpolacją – wyszło całkiem nieźle, jak na tak duże powiększenie. W wygładzeniu brzegów pomaga dodatkowo raster. Ale na niektórych urządzeniach POSTSCRIPT-owych może się pojawić niepożądany efekt w postaci „rozpłynięcia się” koloru po całym obrazie, na skutek czego biel zostaje zastąpiona szarością. Najlepiej interpolacja działa w przypadku ilustracji kolorowych i cieniowanych, co widać na figurze (d) (takiej samej wielkości jak figura (a)), która również została powiększona z wykorzystaniem interpolacji. Ilustracje o gładkich przejściach barwy mogą być powiększane w znacznym zakresie bez utraty jakości.

Trick polega na dostawieniu przy „zębach piły” pikseli o kolorze pośrednim między kolorem obiektu a kolorem tła (p. rys. 10). Oczywiście jeśli piksele są za duże, to oka oszukać się nie da, ale w przypadku małych pikseli, takich jak na przykład ekranowe, wygładzanie daje znakomite efekty i pamiętać o nim powinni w pierwszym rzędzie wszyscy parający się przygotowywaniem publikacji elektronicznych.

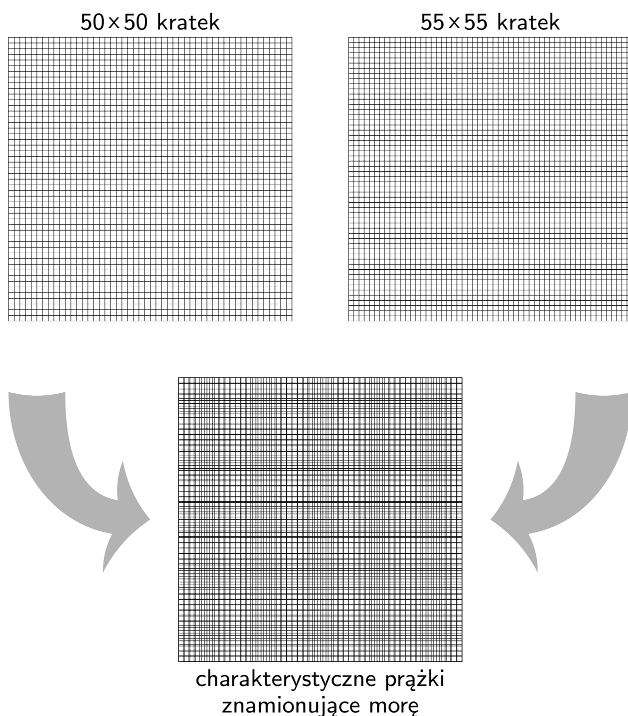
Podobnej techniki używa PostScript przy przekształcaniu map bitowych. Używając specjalnego parametru boole'owskiego (`/Interpolate`) można PostScript poinformować, że dana mapa bitowa ma być transformowana z użyciem wygładzania. Zabieg ten jest godny polecenia w przypadku grafiki cieniowanej i barwnej, zwłaszcza jeżeli przejścia kolorów są w miarę gładkie, a zmiana kształtu oryginału nie jest nazbyt zdecydowana. Niestety starsze interpretery PostScript-u mogą nie znać wygładzania. Wówczas pozostają już tylko specjalizowane programy...

Użycie specjalizowanych programów jest z wielu względów niewygodne, ale pozwala przekształcać mapy bitowe nieafinicznie (nieliniowo), czego PostScript nie potrafi. Tego rodzaju zabiegi są czasami potrzebne na przykład do usunięcia deformacji z wadliwie wykonanej fotografii, czy też do zabiegów specjalnych. Jednak nie polecałbym nadużywania przekształceń nieafinicznych, chociaż doświadczenie poucza, że nuworysze w branży grafiki komputerowej przepadają za tego rodzaju efektami.

5b. *Zmora mory*

Największym wrogiem początkujących adeptów grafiki komputerowej jest mora. Słowo to ma niewinny źródłosłów, pochodzi bowiem od francuskiego słowa *moire*, a oznacza jedwabną tkaninę, mieniącą się charakterystycznymi wzorami, przypominającymi prążki interferencyjne. Istotnie za ten efekt odpowiedzialna jest interferencja. Gdy nakładają się dwa bodźce o podobnej częstotliwości, wtedy nasze zmysły rejestrują dodatkowy bodziec o częstotliwości równej różnicy częstotliwości bodźców źródłowych (rys. 11).

Chociaż interferencja wywołuje ciągłe zmiany bodźca, to jednak zmysł wzroku wykazuje tak silną tendencję do wyostrzania krawędzi, że wzory interferencyjne stają się bardzo wyraźne, wręcz wydaje się, że mamy do czynienia ze skokowymi zmianami sygnału, chociaż obejrzenie wzoru przez lupę nie wykazuje obecności żadnych gwałtowniejszych zmian. Dopiero spojrzenie z pewnego oddalenia ujawnia morę.



Rys. 11. Nałożenie na siebie dwóch kwadratów tej samej wielkości, ale podzielonych raz na 50×50, a raz na 55×55 kwadratów daje wyraźny rytm, powtarzający się w pionie i w poziomie pięciokrotnie, co wynika z zależności $55 - 50 = 5$. Ta właściwość mory jest wykorzystywana w technikach pomiarowych.

W grafice komputerowej zjawisko to pojawia się co i rusz, i to w przeróżnych kontekstach, najczęściej przykrych. Prawie zawsze daje się we znaki w przypadku obróbki grafiki poddanej uprzednio rasteryzacji (por. s. 8). Wówczas próby transformowania takiej grafiki kończą się z reguły powstaniem mory, gdyż rytm przeskalowanego rastra kłóci się z rytmem pikseli urządzenia. Jedynym ratunkiem jest usunięcie rastra za pomocą profesjonalnego programu graficznego, i dopiero potem poddawanie grafiki przekształceniom.

Należy jednak pamiętać, że liniatura użytego rastra determinuje efektywną rozdzielczość grafiki, gdyż usunięcie rastra odbywa się poprzez

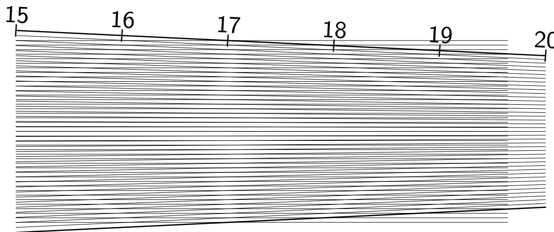
„rozmazanie” obrazu. Programy graficzne są w stanie nieco poprawić ostrość, ale wiele więcej niż umożliwia liniatura uzyskać się nie da.

Nie tylko rastry są utrapieniem. Także faktura muru czy dachu, tkanin, drobne ornamenty i inne elementy tego rodzaju, interferując z rastrem nałożonym podczas druku, mogą spowodować morę.

Przy druku barwnym należy zachować szczególną ostrożność, gdyż klisze do wszystkich czterech kolorów „triady drukarskiej” (p. s. 11 – 12) powinny mieć raster o tej samej częstotliwości. Żeby uniknąć mory, każdy z kolorów musi mieć raster ustawiony pod innym kątem, co wprawdzie teoretycznie nie jest w stanie zlikwidować interferencji, ale może ją uczynić praktycznie niewidoczną.

Rozsądnym kompromisem okazały się następujące kąty rastra: cyjan 15°, magenta 75°, żółć 0°, czerń 45°. Ponieważ kolor żółty jest słabo widoczny, jego interferencja z cyjanem i magentą nie przeszkadza, aczkolwiek daje się dostrzec na diapozytywach.

Ale mora nie tylko nieszczęścia wróży – daje się ona „oswoić” i bywa niezwykle przydatna. Znalazła zastosowanie na przykład w technice pomiarowej do wykonywania bardzo subtelnych pomiarów. Z bliższego nam podwórka – na linijkach drukarskich często umieszczany jest przymiar pozwalający ustalić liniaturę rastra właśnie na podstawie efektu mory (rys. 12).



Rys. 12. Przymiar w kształcie „miotełki” może posłużyć do pomiaru częstotliwości regularnego wzoru, np. liniatury rastra. Skala pokazuje liczbę „włosków miotełki” na centymetr w danym pionie. Jeśli na przymiar zostanie nałożony równolegle do jego osi regularny wzór – w tym przypadku wiązka równoległych linii – to utworzy się charakterystyczna struktura interferencyjna, przypominająca rodzinę hiperbol, z wyraźnie zarysowanym centrum. To miejsce wskazuje częstotliwość badanego wzoru. W tym wypadku odczytujemy, że na jeden centymetr przypada około 17 linii.

6. Krótki przegląd typowych formatów graficznych

Formaty plików graficznych powielają ten sam schemat: składają się z części informacyjnej podającej rozmiary mapy bitowej, model kolorów, rodzaj kompresji itp., oraz z części zawierającej mapę bitową. Czasem może występować wiele elementów danego rodzaju – formaty GIF i TIFF pozwalają na przechowywanie wielu map bitowych w jednym pliku.

Każdy z popularnych formatów można albo scharakteryzować w kilku zdaniach, albo opisać ze wszystkimi szczegółami, bo w sumie o szczegóły tu chodzi. Jeśli jednak zważyć, że pełny opis na przykład formatu TIFF liczy przeszło 100 drobno zadrukowanych stron formatu A4, to przyjdzie się zgodzić, że drugi wariant raczej nie wchodzi w rachubę.

6a. *PCX*

W swoim czasie był to bardzo popularny format, co było w prostej linii skutkiem popularności programu `Paintbrush`, który zapisywał pliki w tym formacie. PCX może zawierać czarno-białe, paletowe, cieniowane oraz kolorowe (model RGB) mapy bitowe. Dane są kompresowane odmianą metody RLE, charakterystyczną dla formatu PCX.

6b. *TARGA*

Format ten jest wart odnotowania choćby z tego względu, że `POV`, znakomity program do kreowania realistycznej grafiki (dystrybuowany jako oprogramowanie swobodne), zapisuje dane w tym formacie. Format TARGA uwzględnia paletowe, cieniowane oraz kolorowe (model RGB) mapy bitowe. Dane mogą zawierać maskę (kanał alfa). Tylko kompresja RLE jest wykorzystywana.

6c. *GIF*

Dzięki Internetowi GIF stał się jednym z najpopularniejszych ostatnimi czasy formatów. Akceptuje jedynie paletowe mapy bitowe, ale dzięki zwartości nagłówka i zastosowanej kompresji LZW pliki mogą być bardzo małe (kilkaset, a nawet kilkadziesiąt bajtów). Istnieje możliwość zdefiniowania koloru przezroczystego, a także przechowywania kilku obrazów wraz z informacją dla ewentualnego programu animującego, taką jak np. czas trwania poszczególnych klatek, liczba powtórzeń, itp.

6d. *JPEG*

Jeśli chodzi o kompresję – najistotniejszy aspekt formatu JPEG – to wszystko zostało powiedziane w punkcie 4. W tym miejscu można jedynie dorzucić, że format JPEG dopuszcza także kompresję bezstratną i że obsługuje cieniowane i kolorowe mapy bitowe (RGB i CMYK, aczkolwiek jeśli chodzi o CMYK, to starsze programy miewają kłopoty z jego zrozumieniem).

6e. *TIFF*

Jest to niewątpliwie jeden z najbogatszych formatów graficznych, praktycznie zawierający wszystkie inne formaty. Może zawierać wiele map bitowych, dopuszcza wszystkie omawiane modele grafiki: czarno-białą, cieniowaną, paletową, kolorową (RGB i CMYK), możliwe jest także przechowywanie maski. Obsługiwane są praktycznie wszystkie rodzaje kompresji, między innymi LZW i JPEG. Mimo szerokiego rozpowszechnienia i stosunkowo niezłej dokumentacji zdarza się, że niektóre popularne programy graficzne danej odmiany TIFF-a nie interpretują poprawnie.

6f. *EPS*

Wyjątkiem wśród formatów jest oczywiście grafika PostScript-owa, gdyż programy zapisane w bardzo bogatym języku mogą być niezwykle skomplikowane. W szczególności PostScript może interpretować niemalże wszystkie wymienione formaty. Nie jest to możliwość „wirtualna” – z `Ghostscript`-em dystrybuowane jest oprogramowanie, napisane w języku PostScript, pozwalające przetwarzać pliki GIF, JPEG, PCX, ostatnio doszlusował format TIFF. Z tu wymienionych brakuje jedynie formatu TARGA.

7. Podsumowanie

Jak widać, tematyka jest szeroka i pełne jej zgłębienie to niewątpliwie lata pracy. Tym samym zbyt szczegółowe opisywanie formatów graficznych mija się z celem, gdyż wszystkie formaty są modyfikowane w miarę upływu czasu, niekiedy tak szybko, że trudno za zmianami nadążyć. W każdym konkretnym wypadku lepiej nie ufać dokumentacji drukowanej i poszukać najświeższej wersji opisu w Internecie – stało się to naprawdę bardzo łatwe.

W związku z tym jedyne, co ma sens, to zaznajamianie się z podstawową problematyką związaną z modelami kolorów czy algorytmami kompresji,

które nie zmieniają się aż tak szybko jak specyfikacja niektórych formatów. Mam nadzieję, że broszurka ta posłuży jako wstęp do dalszej, samodzielnej już, penetracji w tym kierunku.

WZMIANKOWANA LITERATURA

[1] B. E. Bayer, *An optimum method for two-level rendition of continuous-tone pictures*, Int. Conf. Commun., 1973, s. (26-11) – (26-15).

[2] J. Fokker, *Functional specification of JPEG decompression, and an implementation for free*, 1995 (`http://www.cs.ruu.nl./~jeroen`).

[3] D. A. Huffman, *A method for the construction of minimum redundancy codes*, Proc. IRE, 40 (9), 1952, s. 1098 – 1101.

[4] *POSTSCRIPT Language reference manual*, Addison-Wesley, wydanie II, 1991.

[5] *Tag image file format, revision 6.0*, Aldus Corporation, 3 czerwca, 1992 (`ftp://ftp.sgi.com/graphics/tiff`).

[6] G. K. Wallace, *The JPEG still picture compression standard*, 1991 (`ftp://ftp.uu.net/graphics/jpeg/wallace.ps.gz`).

[7] T. A. Welch, *A technique for high performance data compression*, IEEE Computer, 17 (6), czerwiec 1984, s. 8 – 19.

[8] J. Ziv, A. Lempel, *A universal algorithm for sequential data compression*, IEEE Transactions on Information Theory, Vol. IT-23, No. 3, maj 1977, s. 337 – 343.

SPIS TREŚCI

1. Wstęp 2
2. Grafika dyskretna a grafika obwiedniowa 3
3. Podstawowe rodzaje map bitowych 5
   3a. *Czarno-białe mapy bitowe* 5
   3b. *Cieniowane mapy bitowe* 6
   3c. *Rastry, rozdzielczość, liniatura* 8
   3d. *Kolorowe mapy bitowe – model RGB* 10
   3e. *Kolorowe mapy bitowe – model CMYK* 11
   3f. *Paletowe mapy bitowe* 14
   3g. *Mapy bitowe z „maską”* 15
4. Kompresja danych 16
   4a. *Kompresja RLE* 16
   4b. *Kompresja Huffmana* 18
   4c. *Kompresja LZW* 21
   4d. *Kompresja JPEG* 26
5. Strefy zwiększonego ryzyka 33
   5a. *Przekształcenia afiniczne i nie tylko* 33
   5b. *Zmora mory* 35
6. Krótki przegląd typowych formatów graficznych 38
   6a. *PCX* 38
   6b. *TARGA* 38
   6c. *GIF* 38
   6d. *JPEG* 38
   6e. *TIFF* 39
   6f. *EPS* 39
7. Podsumowanie 39
   Wzmiankowana literatura 40