

Alternatywą dla ASCII stał się 16-bitowy system kodowania Unicode, na reprezentowanie do 65 tysięcy znaków w sposób jednoznaczny. nie żadne informacje dodatkowe, pozwalające rozróżnić języki czy wersje znaki. Jako ciekawostkę można podać, że w pliku zgodnym z Unicode znaków określa kolejność ich faktycznego czytania, a nie oglądanie o tekście artykułu prasowego zawierającego tekst w języku angielskim i innymi słowami lub zdaniami arabskimi).



Pojemność Unicode początkowo wydawała się ogromna, ale gdy ten system popularny i zaczęto do niego dodawać znaki chińskie, koreańskie itp., jego pojemność także się wyczerpała! Jak widać, nawet znając historię wieży Babel, eksperci się nauczyli i błędnie wymyślili nowy standard!

Ponieważ pojemność Unicode wyczerpała się, wprowadzono kodowanie UTF-16, pozwalające na kodowanie znaków za pomocą kodów o dwóch cyfrach: znaki mieszczące się w zakresie 16-bitowym są zapisywane w dwóch słowach, a znaki spoza zakresu — w dwóch słowach. Kodowanie znaków poza zakresem wykorzystuje wykrywanie znaków pierwszej jednostki kodowej, będącej właściwym znakiem i wówczas wiadomo, że sekwencję należy rozszerzoną (faktury do zrozumienia przykład opisujący ten mechanizm możesz przeczytać na stronie <https://pl.wikipedia.org/wiki=UTF-16>).

Już w tym krótkim podrozdziale można się zorientować, jak skomplikowane zagadnieniem jest kodowanie tak pozornie prostych obiektów jak słowa. W końcu problem rozwiązują tu języki przenośne typu Java, gdzie operacje prostu liczbami — co z tego, że 16-, a nie 8-bitowymi, jak kiedyś. Jeśli program interpretuje znaki w sposób uproszczony — przy założeniu, że znaki są 16-bitowe — to na pewno nie poradzi sobie z kodowaniem Unicode, który wymaga dodatkowej inteligencji (interpretacji) odczytywanych słów binarnych 2- lub 4-bajtowych!

Kodowanie obrazów

Czym są obrazy? Odpowiedź zależy od tego, kto zadaje pytanie! Dla komputera obraz miał czysto fizyczną formę i nie można go było przenieść z miejsca na inny, niż używając fizycznego transportu. Co innego **obrazy komputerowe** mają swego rodzaju abstrakcją rzeczywistych utworów artystycznych.

Obrazy komputerowe można przesyłać przez sieci komputerowe i przechowywać na urządzeniach końcowych, które z grubsza ujmując, muszą posiadać jakieś technologii „rozświetlić” punkt na ekranie i tak ustawić jego jasność, aby oko człowieka zarejestrowało na nim takie atrybuty jak kolor, jasność, kontrast. W związku z tym urządzenie fizyczne musi w jakiś sposób odwzorować przez komputer informacje i zaprezentować coś, co człowiek uzna za obraz.

wizualną. Podobnie zresztą dzieje się w przypadku innych skomplikowanych formatów „multimedialnych”, np. służących do kodowania dźwięków lub filmów.

W jaki sposób komputer może zapamiętać atrybuty obrazu? Najprościej rzecz ujmując, informacje wizualne można modelować (kodować) przy użyciu dwóch metod:

- *Grafika rastrowa* — obraz stanowi siatkę odpowiednio kolorowanych pikseli na monitorze komputera (lub na innym urządzeniu, np. drukarce). Element adresowany przez grafikę rastrową to tzw. piksel, czyli kropka na ekranie. Im więcej takich kropek, tym lepsze są możliwości oszukania ludzkiego oka i przybliżenia prezentacji komputerowej do efektów znanych ze świata rzeczywistego. Formaty grafiki rastrowej na pewno znasz ze słyszenia, np.: BMP, ICO, GIF, JPEG, PNG. Część z nich przy okazji dokonuje kompresji (o kompresji danych szerzej opowie w rozdziale 18.). W grafice rastrowej, która jest uzależniona od rozdzielczości obrazu, skalowanie może się wiązać z utratą jakości i jest wolne. Pliki obrazów mogą mieć bardzo duże rozmiary.
- *Grafika wektorowa* — używa funkcji matematycznych do modelowania obiektów (np. punktów, linii, okręgów, wieloboków). Podstawowym standardem grafiki wektorowej jest SVG, którego opis można znaleźć na stronie <https://www.w3.org/Graphics/SVG>. W przypadku grafiki wektorowej podczas rysowania obrazu i tak wszystko jest punktem na ekranie, ale istotne jest to, że odwzorowywanie na piksele i rysowanie na urządzeniu fizycznym jest przeprowadzane dopiero na sam koniec. W związku z tym algorytmy przetwarzające obrazy zapisane wektorowo zasadniczo nie są czułe na fizyczną wielkość ekranu, co zapewnia lepszą jakość grafiki niż w przypadku grafiki rastrowej. Podobnie rozmiar obrazu nie wpływa na rozmiar pliku. Niestety, aby wygenerować obraz z modelu wektorowego, potrzebna jest znacznie większa moc obliczeniowa niż w przypadku formatów rastrowych. W grafice wektorowej skalowanie nie wiąże się z utratą jakości (brak zależności od rozdzielczości obrazu) i jest szybkie.

Mapy bitowe na przykładzie formatu BMP

Format grafiki rastrowej może posłużyć do pokazania, w jaki sposób komputery zapisują informacje o obrazie w pamięci lub pliku. Na pewno nie wystarczy prosta tabela zawierająca sekwencje kodów (np. ASCII), gdyż pliki graficzne wymagają znacznie więcej informacji.

Wrzecie przykładu spójrzmy, jak zapisywane są pliki BMP w popularnym formacie mapy bitowej (jest to tzw. DIB — *Device Independent Bitmap*)². Plik graficzny jest podzielony na sekcje informacyjne i sekcje danych o następującym układzie logicznym:

² O innym popularnym formacie (GIF) powiem w rozdziale 18., „Kodowanie i kompresja danych”.

BITMAPFILEHEADER	bmfh	# Informacje o pliku, bajty 0x00-0x0D
BITMAPINFOHEADER	bmih	# Informacje o mapie bitowej (rozmiar, kolory itp.), # bajty 0x0E-0x35
RGBQUAD	aColors[]	# Tablica kolorów, bajty 0x36-0x75
BYTE	aBitmapBits[]	# Właściwe dane obrazu

Pierwsze trzy struktury są danymi opisowymi (tzw. metadanymi) pozwalającymi na właściwą interpretację danych obrazu (`aBitmapBits[]`). W formacie mapy bitowej pierwszy bajt odpowiada lewemu dolnemu pikselowi, a ostatni bajt — prawemu górnemu (obraz jest przechowywany „do góry nogami”).

Struktura o nazwie **BITMAPFILEHEADER** zawiera informacje o pliku graficznym zapisanym na dysku lub wczytanym do pamięci (tabela 2.8).

TABELA 2.8. Nagłówek pliku BMP — informacje o pliku

POZYCJA	ROZMIAR	NAZWA	PRZEZNACZENIE
0	2	<code>bfType</code>	Ustawione na 'BM' dla plików BMP w Windowsie
2	4	<code>bfSize</code>	Rozmiar pliku w bajtach
6	2	<code>bfReserved1</code>	Zarezerwowany (0)
8	2	<code>bfReserved2</code>	Zarezerwowany (0)
10	4	<code>bfOffBits</code>	Offset (przesunięcie) w pliku, od którego zaczyna się blok danych



W formacie BMP wszelkie wartości całkowite, takie jak `bfSize`, kodowane na kilku bajtach są przechowywane w formacie zwanym *little-endian* (słowo *end* oznacza „koniec”), w którym najmniej znaczący bajt umieszczony jest jako pierwszy. Konwencja *big-endian* jest oczywiście odwrotna i warto zwrócić uwagę na ten nieważny, aby uniknąć błędów dekodowania liczb!

Kolejna struktura, o nazwie **BITMAPINFOHEADER**, opisana w tabeli 2.9, zawiera informacje o mapie bitowej (rozmiar, kolory itp.). W tej sekcji instruujemy algorytm dekodujący plik graficzny, jak ma poprawnie odczytywać nadchodzące sekwencje bajtów danych obrazu (rozmiar, kolory, informacje o ewentualnej kompresji).

Zajmijmy się teraz kolorami pikseli. Są one zapisane w tablicy kolorów przy użyciu struktury o nazwie **RGBQUAD** (tabela 2.10), gdzie każdy kolor zajmuje trzy bajty (B — niebieski, G — zielony, R — czerwony). Tablica ta jest zbędna w bitmapach 24-bitowych, gdzie każdy piksel kodowany jest 3 bajtami, odpowiadającymi bezpośrednio składowym czerwonej, zielonej i niebieskiej koloru piksela.

Z uwzględnieniem czwartego bajta (rezerwowego) każdy wpis w palecie kolorów zajmuje 4 bajty. Przykłady odwzorowania składowych RGB na „prawdziwy” kolor możesz znaleźć w sieci (np. https://pl.wikipedia.org/wiki/Lista_kolor%C3%97w).

TABELA 2.9. Nagłówek pliku BMP — informacje o obrazie

POZYCJA	ROZMIAR	NAZWA	PRZEZNACZENIE
34	4	biSize	Rozmiar struktury BITMAPINFOHEADER w bajtach
35	4	biWidth	Szerokość obrazu w pikselach (od lewej do prawej)
36	4	biHeight	Wysokość obrazu w pikselach (od dołu do góry)
37	2	biPlanes	Liczba warstw kolorów
38	2	biBitCount	Liczba bitów na piksel (inaczej mówiąc, informacja o kolorach). Wartości: 1 (obraz czarno-biały), 4 (16 kolorów), 8 (256 kolorów), 24 (16,7 mln kolorów)
39	4	biCompression	Typ zastosowanej kompresji (typowo 0 — brak, 1 — kompresja RLE 8-bitów/piksel itp.). O kompresji RLE powiem w rozdziale 18., „Kodowanie i kompresja danych”
40	4	biSizeImage	Rozmiar rysunku
41	4	biXPelsPerMeter	Rozdzielcość pozioma
42	4	biYPelsPerMeter	Rozdzielcość pionowa
43	4	biClrUsed	Liczba kolorów w paletie (jeśli została ustawiona na 0, to liczba kolorów jest liczona na podstawie biBitCount)
44	4	biClrImportant	Liczba ważnych kolorów w paletie (jeśli została ustawiona na 0, to wszystkie kolory są ważne)

TABELA 2.10. Paleta kolorów RGB w pliku BMP

POZYCJA	ROZMIAR	NAZWA	PRZEZNACZENIE
54	1	rgbBlue	Wartość składowej niebieskiej koloru (0 – 255)
55	1	rgbGreen	Wartość składowej zielonej koloru (0 – 255)
56	1	rgbRed	Wartość składowej czerwonej koloru (0 – 255)
57	1	rgbReserved	Zarezerwowany (0)

Dla plików BMP 1-, 4- i 8-bitowych kolor piksela obrazu w danym rodzaju pliku kodowany jest za pomocą wymienionej liczby bitów jako indeks w tabeli kolorów będącej częścią pliku BMP. Algorytm komputerowy, na podstawie bieżąco odczytywanego bajta danych zapisanych w sekcji aBitmapBits, danych o obrazie i jego informacie, jest w stanie odwzorować kolor piksela widzianego na ekranie. Przykładowo, dla palety czarno-białej w tablicy palety kolorów trzeba zapisać tylko dwie wartości: {0,0,0,0} i {0xff,0xff,0xff,0x0} — kolor czarny i kolor biały.

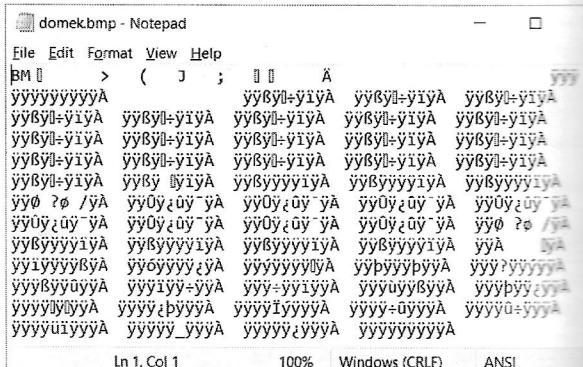
Reguły zapisu bloku danych definiujących obraz są ściśle związane z formatem plików:

- plik 24-bitowy nie wymaga mapy kolorów i każde 3 kolejne bajty obrazu reprezentują jeden piksel;
- plik 8-bitowy rozróżnia paletę o maksymalnej liczbie 256 kolorów — bajt obrazu jest indeksem do elementu mapy;
- plik 4-bitowy zawiera mapę kolorów o maksymalnie 16 kolorach — bajt danych obrazu zawiera zatem dwa indeksy do elementów mapy (pierwszego i drugiego);
- plik 1-bitowy zawiera 2-elementową mapę kolorów (obraz czarno-biały), czyli każdy bajt zawiera aż osiem indeksów do elementów mapy;
- długość każdej linii obrazu jest wyrównywana do pełnych 4 bajtów.

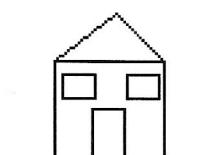
Na opis formatu BMP poświęciłem trochę czasu, ale mam nadzieję, że już orientujesz się, na czym polega realizacja złożonych struktur informacyjnych opartych na interpretowaniu sekwencji bajtów. Program odczytujący lub dekoder sprzętowy muszą wiedzieć, że mają do czynienia z określonym formatem zapisu, i odpowiednio interpretować kolejno odczytywane bajty lub bloki danych. Pomyłka w interpretacji formatu prowadzi do różnorodnych efektów ubocznych — zobacz np., skończy otwarcie pliku graficznego BMP w programie Notatnik, poprawnie otwierającym zwykłe pliki tekstowe (na dodatek w kilku standardach kodowania). Program ten dosłownie „zgłępieje” po załadowaniu mapy bitowej (rysunek 2.3).

RYSUNEK 2.3.

Plik graficzny otwarty w edytorze tekstowym Notepad (Windows)



Tymczasem ten sam plik, poprawnie otwarty w trybie podglądu graficznego, zawiera prosty obrazek:



Narysowałem dość nieporadnie mały domek, zapewne wybudowany na podstawie zgłoszenia, gdyż już na pierwszy rzut oka widać, że ma powierzchnię mniejszą niż 35 m² i jest parterowy...

Ale żarty na bok. Jest to obrazek w opisany wcześniej formacie BMP i sprawdźmy kilka wybranych bajtów, by przekonać się, czy faktycznie mamy do czynienia z tym formatem!

Sprawdźmy zatem, co zawierają wybrane bajty zapisane w tym pliku, odczytując np. pierwsze 6 bajtów, czyli elementy opisane jako BITMAPFILEHEADER i rozmiary obrazka w pikselach zawarte w sekcji BITMAPINFOHEADER. W tym celu napisałem prosty program w Pythonie, który przy okazji pokazuje podstawowe zasady odczytu i interpretacji sekwencji bajtów pliku otwartego w trybie binarnym ([odczytBMP.py](#)).

```
import sys
print("**** Odczyt pliku BMP w trybie binarnym ***")
plik=open("domek.bmp", "rb")
dane = plik.read() # Czytamy cały plik
dlugosc=len(dane)
print("Otwarto plik: ",plik.name)
print("Tryb otwarcia: ",plik.mode)
print("Plik zajmuje", dlugosc, "bajtów na dysku")
plik.seek(0) # Ustawiamy się na pozycję 1. (bfType)
print("6 początkowych bajtów:")
for i in range(0,6):
    c=plik.read(1)
    print(" Bajt nr:", i, ":", c, "hex:", c.hex())
plik.seek(2) # Ustawiamy kurSOR na 3. bajt, czyli parametr bfSize (4 bajty)
bfSize= plik.read(4)
print("bfSize binarnie:", bfSize)

# Dekodujemy wartość całkowitą z ciągu bajtów, używając funkcji int.from_bytes o następującej składni:
# int.from_bytes (ciąg binarny, endianness)
# endianess = 'big' lub 'little' (konwencja big lub little endian)
# Można użyć przenośnej konstrukcji sys.byteorder, która sama wykryje poprawną wartość:
bfSizeDecymalnie=int.from_bytes(bfSize, byteorder=sys.byteorder)
print("bfSize decymalnie:", bfSizeDecymalnie)

plik.seek(18) # Ustawiamy kurSOR na 19. bajt, czyli parametr biWidth (4 bajty)
biWidth=plik.read(4) # Szerokość obrazu
plik.seek(22) # Ustawiamy kurSOR na 23. bajt, czyli parametr biHeight (4 bajty)
biHeight=plik.read(4) # Wysokość obrazu
print("biWidth=", int.from_bytes(biWidth, byteorder=sys.byteorder), "pikseli")
print("biHeight=", int.from_bytes(biHeight, byteorder=sys.byteorder), "pikseli")
plik.close()
```

Program wyświetli następujące informacje:

```
*** Odczyt pliku BMP w trybie binarnym ***
Otwarto plik: domek.bmp
Tryb otwarcia: rb
Plik zajmuje 770 bajtów na dysku
6 początkowych bajtów:
```

```
Bajt nr: 0 : b'B' hex: 42
Bajt nr: 1 : b'M' hex: 4d
Bajt nr: 2 : b'\x02' hex: 02
Bajt nr: 3 : b'\x03' hex: 03
Bajt nr: 4 : b'\x00' hex: 00
Bajt nr: 5 : b'\x00' hex: 00
bfSize binarnie: b'\x02\x03\x00\x00'
bfSize decimalnie: 770
biWidth= 74 pikseli
biHeight= 59 pikseli
```

Wartość bfType to oczywiście zajmujący dwa bajty 'BM'.

Wartość bfSize to cztery bajty: 02 03 00 00 (wartości szesnastkowe). Tak jak wcześniej podałem, wartości całkowite kodowane przy użyciu kilku bajtów są przechowywane w formacie zwanym *little-endian*, w którym najmniej znaczący bajt umieszczony jest jako pierwszy. Jak zatem można wyłuskać z tak sformatowanej liczby jej „prawdziwą” wartość bfSize wynoszącą 0x0302, czyli binarnie 001100000010 albo 770 dziesiętnie?

Okazuje się, że w Pythonie nie jest to trudne. W przykładowym programie pozałem prostą metodę zakładającą użycie funkcji `int.from_bytes`. Dzięki poprawnej konwersji widzisz, że odczytana długość pliku (770 bajtów) jest faktycznie zakodowana w polu bfSize.

Co było do udowodnienia!