Poznaj GIFa

praktyczny tutorial

autor: *Piotr Kędziora*
wykonano w ramach przedmiotu Złożone Systemy Cyfrowe

Czym jest GIF?

GIF (Graphics Interchange Format) to rastrowy plik graficzny, przeznaczony głównie do potrzeb internetu. GIF pozwala nam na animację, przez łączenie danych obrazów w jednym pliku. Do zapisu obrazu używana jest kompresja LZW. W standardowym podejściu GIF obsługuje do 256 kolorów, wybranych z palety 24 bitowej (ponad 16 milionów kolorów).

Istnieją dwie wersje GIFa: 87a (z roku 1987) i ostatnia 89a (z roku 1989), którą będziemy tu opisywać.

W tym tutorialu zajmiemy się jedynie GIFem statycznym, nie interesuje nas na ten moment animacja.

Jak “podejrzeć” co znajduje się w pliku GIF?

Jeśli chcemy zobaczyć jak wygląda nasz plik, zalecam skorzystać z programu GIMP. Bardzo dobrze radzi sobie on z odczytem tych plików i pozwala na dowolne ich przybliżenie, co będzie dla nas ważne, bo tłumaczyć GIFa będziemy na podstawie prostych obrazów o małej rozdzielczości.

Zupełnie inną kwestią jest sprawdzenie jak plik wygląda z punktu widzenia programisty, czyli zobaczenie poszczególnych bajtów. Najbardziej przystępna forma to np. program HxD.

> Do pliku GIF możemy się dostać z poziomu programu w C, korzystając z funkcji `fgetc()`

Zacznijmy od początku...

Każdy plik GIF zaczyna się tak samo - to napis “GIF89a” napisany kodami ASCII, czyli pierwsze 6 bajtów naszego pliku to: 47 49 46 38 37 61

Wymiary pliku GIF

Na następnych 4 bajtach mamy zapisaną szerokość i wysokość obrazu (po 2 bajty na każdy wymiar). I tu warto wspomnieć jak jest to zapisane. Pewnie nie spodziewasz się, że dwa bajty w postaci “00 01” oznaczają, że wymiar to 256px, prawda?

Zasada jest taka, że jeśli wymiar da się zapisać na jednym bajcie (jest mniejszy niż 256px) to zapisujemy go w postaci szesnastkowej na pierwszym bajcie, drugi pozostawiając pustym (np. dla wymiary 8px nasze dwa bajty będą wyglądać tak: “08 00”). Jeśli zaś jest większy, czyli wymaga zapisanie na dwóch bajtach (wartości 256px - 65 535), to zapisujemy je do pliku w odwrotnej kolejności. Jeśli więc nasz rozmiar to 1000px, co kodujemy szestanstkowo 3E8, to w zapisujemy bajty: E8 03.

Zachęcam, by zapoznać się przy tej okazji z terminem Little endian ;)

Spakowany bajt

Teraz mamy w pliku jeden spakowany bajt. Co on oznacza? W gruncie rzeczy mówi nam, by na ten konkretny bajt spojrzeć z perspektywy bitów. Ok, czyli mamy jakby 8 miejsc, na których możemy wstawić zera czy jedynki.

Pierwszy bit zazwyczaj jest jedynką. Mówi nam o tym czy w pliku występuje globalna tablica kolorów.

Trzy kolejne bity oznaczają tzw. głębię koloru. Gdy na piksel chcemy mieć n bitów, zakodujemy tam wartość n-1, czyli dla 2: 001, dla 8: 111

Następny bit oznacza tzw. flagę sortującą, mówiącą czy tablica kolorów jest posortowana z malejącą częstotliwością (ma to pomóc dekoderowi) , jednak obecnie możemy zostawić tą wartość jako 0.

Trzy ostatnie bity naszego spakowanego bajtu to zakodowana wielkość globalnej tablicy kolorów. Jest wyliczana jako 2^{N+1}, gdzie N jest tą wartością, którą kodujemy. Wielkość globalnej tablicy kolorów to po prostu liczba kolorów, których możemy użyć w naszym GIFie.

W takim razie, gdy mamy globalną tablicę kolorów, obraz o rozdzielczości kolorystycznej 8 bitów i chcemy użyć 256 kolorów, to nasz spakowany bajt będzie miał postać 1111 0111

Tło

W kolejnym bajcie możemy zdefiniować kolor tła, czyli jak będą pokolorowane piksele, których koloru nie zdefiniujemy (piksele niepokryte przez obraz). Definiujemy go jako numer w globalnej tablicy kolorów

Pixel Aspect Ratio

By nie zagłębiać się w szczegóły, w prawie wszystkich przypadkach ta wartość jest ustawiona na 00 (ciekawskich, odsyłam tutaj i do specyfikacji)

Globalna tablica kolorów

Teraz kodujemy globalną tablicę kolorów. Zadeklarowaliśmy jej rozmiar w spakowanym bicie, teraz musimy dopełnić formalności i taką ilość kolorów zakodować. Każdy kolor będzie składał się z 3 wartości: składowej R, G i B (każda o wartości z przedziału 0-255). Tym

sposobem jesteśmy w stanie zakodować jeden z 256^3 kolorów. Ale... ograniczenie na wielkość globalnej tablicy kolorów to 256, co prowadzi nas do wniosku, że do dyspozycji mamy owszem, ponad 16 milionów kolorów, ale możemy w naszym pliku wykorzystać jedynie 256 z nich.

> tak naprawdę, to ograniczenie da się ominąć przez stosowanie lokalnych tablic kolorów, dla poszczególnych fragmentów obrazu, więcej można przeczytać na Wikipedii

Dla przykładu, załóżmy że chcemy zakodować tablicę składającą się z 4 kolorów: białego, czerwonego, niebieskiego i żółtego. Biały ma wartość RGB: `0 0 0`, czerwony `255 0 0` niebieski `0 0 256`, a zielony `0 256 0`. W naszym pliku przedstawilibyśmy to w postaci: `00 00 00 FF 00 00 00 00 FF 00 FF 00`.

Możemy więc opisać tablicę kolorów poprzez ciąg bajtów opisujących na przemian wartości red, green i blue dla każdego z kolorów.

Łatwo możemy też obliczyć wielkość globalnej tablicy kolorów: $3 \times 2^{\text{wartość wpisana w spakowany bajt}+1}$.

Deskryptor obrazu

Kolejne wartości bajtowe to tzw. deskryptor obrazu, rozpoczynający się od wartości 2C.

Następnie definiujemy położenie obrazu, który zaraz zdefiniujemy na płaszczyźnie, którą wcześniej zdefiniowaliśmy jako wymiarami. Chodzi o to, że możemy mieć plik GIF wielkości 10x10px, ale obraz może być zapisany tylko na 3x3 pikselach w prawym dolnym rogu, wtedy musimy jako kolejne bajty (po dwa) podać jakąś wartość (zapisaną podobnie jak wielkość - poprzez little endian) najpierw dla położenia “od lewej” a potem “od góry”. Zazwyczaj to po prostu `00 00 00 00`, bo chcemy by cały wymiar był pokryty obrazem. Następnie definiujemy wielkość obrazu, która w większości przypadków jest dokładnie tą samą wartością, którą podaliśmy na początku jako wymiary. Najpierw szerokość, potem wysokość.

Ostatnią wartością w deskryptorze obrazu jest spakowany bajt, który mówi nam:

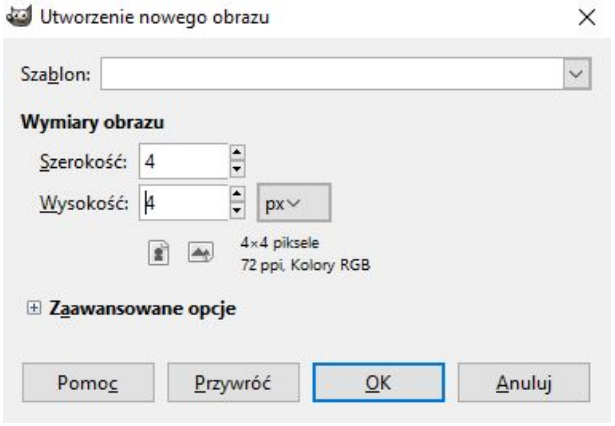
- 1. bit: czy występuje lokalna tablica kolorów dla obrazu, który definiujemy
- 2. bit: czy występuje przeplot (zazwyczaj 0)
- 3. bit: czy lokalna tablica kolorów jest posortowana według malejącej częstotliwości (gdy brak - 0)
- 4. i 5. bit - są zarezerwowane (pozostawmy jako 00)
- 6., 7. i 8 bit to wielkość lokalnej tablicy kolorów zdefiniowana jako 2^(wartość zakodowana+1)

Dla prostych przykładów, bez lokalnej tablicy kolorów, zazwyczaj będzie to po prostu bajt pusty.

W praktyce

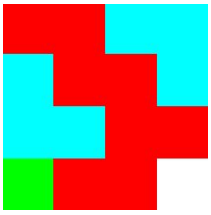
Zanim przejdziemy do najbardziej skomplikowanego fragmentu naszego opisu, powtórzmy pozyskaną na razie wiedzę, tworząc i opisując bajt po bajcie przykładowy plik GIF.

W programie GIMP tworzymy nowy obraz o wielkości 4x4 piksele:

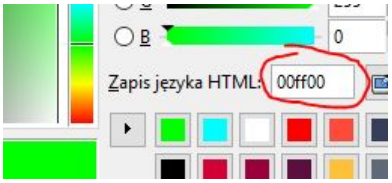


powiększamy obraz (skrót klawiszowy: Z) i tworzymy wzór, złożony z 4 kolorów. Najlepiej użyć do tego ołówka (skrót klawiszowy: N) i ustawić rozmiar na 1 piksel - dzięki temu będziemy kolorować pojedyncze piksele.

Stwórzmy taki obraz:



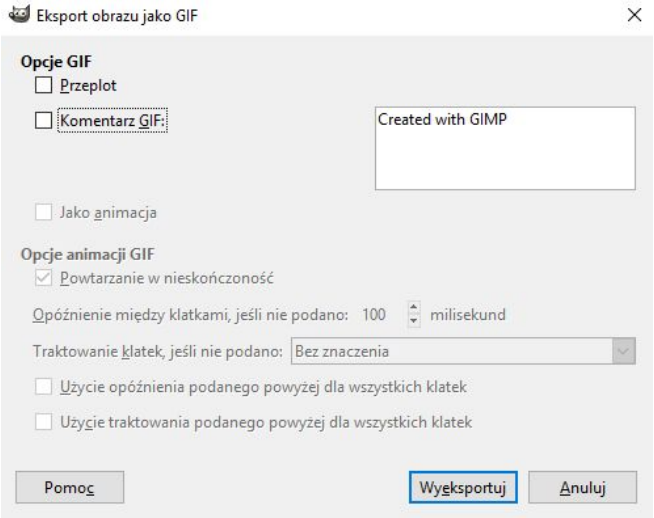
wartości koloru możemy zmienić tu:



dla naszego obrazka będą to:
`FF0000` - czerwony
`FFFFFF` - biały
`00FFFF` - cyjan
`00FF00` - zielony

Teraz zapiszmy naszego GIFa: `Plik > Wyeksportuj jako… > moj_gif.gif`

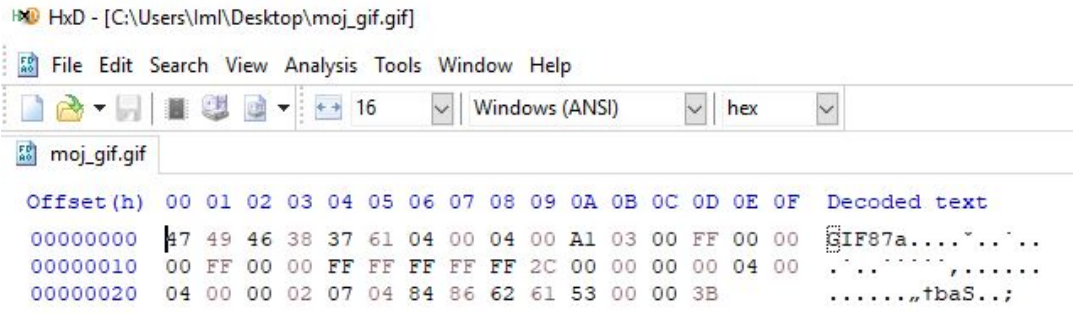
W panelu ustawień odznaczmy pole dodania komentarza:



i zatwierdzamy.

Dobrze, plik GIF stworzony, teraz przyjrzyjmy się jego strukturze.
Otwórzmy program HxD i otwórzmy w nim nasz plik.

Otrzymamy taki wynik:



widzimy tu 46 bajtów, co zgadza się z tym co widzimy we właściwościach pliku.

Teraz spróbujmy rozpoznać poszczególne części:

```
47 49 46 38 37 61 - zapisane kodami ASCII “GIF 89a”
04 00 - szerokość obrazu
04 00 - wysokość obrazu
A1 - spakowany bajt: 1010 0001
   -  1 - obecna jest globalna tablica kolorów
   -  010 - stosujemy głębię koloru 3 bitową
   -  0 - tablica kolorów nie jest posortowana
   -  001 - wielkość tablicy kolorów jest równa 2^(N+1)=2^(1+1)=4
03 - kolor tła (jego numer w globalnej tabeli kolorów)
00 - pixel aspect ratio
FF 00 00 - poszczególne wartości globalnej tablicy kolorów,
00 FF 00   najpierw czerwony, potem zielony, cyjan i biały
```

```
00 FF FF
FF FF FF
2C - znacznik początku deskryptora obrazu
00 00 - położenie obrazu od lewej
00 00 - położenie obrazu od góry
04 00 04 00 - wielkość obrazu
00 - spakowany bajt
```
i jak na razie tyle wiemy, resztą zajmiemy się za chwilę.

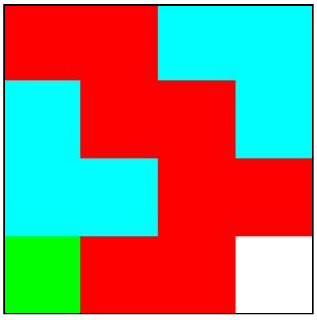
fragment który rozpoczyna się od 02, a kończy przed 3B to...

Dane obrazu

Weźmy naszą globalną tablicę kolorów:

#0	FF 00 00
#1	00 FF 00
#2	00 FF FF
#3	FF FF FF

i spójrzmy na nasz obraz jako na tablicę indeksów w globalnej tablicy:



0	0	2	2
2	0	0	2
2	2	0	0
1	0	0	3

ten sposób bardzo dobrze opisany jest na Wikipedii

idąc dalej, nasz obraz możemy zapisać jako jednowymiarową tablicę wartości:

0	0	2	2	2	0	0	2	2	2	0	0	1	0	0	3

to 16 wartości do zapisania.

Format GIF proponuje nam bezstratną kompresję LZW w celu zmniejszania ilości danych do zapisu.

Spróbujmy powoli zrozumieć ideę.

Najpierw stwórzmy **słownik**, czyli taki zbiór powtarzających się indeksów. Dodajmy do niego wszystko co mamy w naszej globalnej tablicy kolorów, a następnie znacznik Clear Code i End of Information Code:

#0	0
#1	1
#2	2
#3	3
#4	Clear Code
#5	End of Information Code

Teraz na przykładzie spróbuję wyjaśnić sposób działania algorytmu LZW, który dalej opiszę formalnie:

1. zaczynamy z pustym kodem
2. na początku zapisujemy #4 (Clear Code) jako wartość w naszym kodzie
3. mamy ciąg danych 0, 0, 2, 2, 2, 0, 0, 2, 2, 2, 0, 0, 1, 0, 0, 3
4. bierzemy pierwszą wartość
   0, 0, 2, 2, 2, 0, 0, 2, 2, 2, 0, 0, 1, 0, 0, 3
5. sprawdzamy czy jest w słowniku - jest, więc bierzemy kolejną wartość
   0, 0, 2, 2, 2, 0, 0, 2, 2, 2, 0, 0, 1, 0, 0, 3
6. sekwencji 0, 0 nie ma w słowniku, więc dopisujemy go, a do kodu dodajemy indeks w słowniku, jaki ma sekwencja bez ostatniej wartości - czyli 0~~0~~ (wykreślamy ostatnie 0 więc patrzymy na indeks jednego 0) i dopisujemy do kodu #0
7. obecnie nasz kod to #4#0, zaś słownik:

#0	0
#1	1
#2	2
#3	3
#4	Clear Code
#5	End of Information Code
#6	0, 0

8. rozpoczynam od ostatnio analizowanej wartości i patrzę na kolejną:
   0, 0, 2, 2, 2, 0, 0, 2, 2, 2, 0, 0, 1, 0, 0, 3
9. sprawdzam czy mam sekwencję 0, 2 w słowniku; nie mam, więc dopisuję, a do kodu dodaję indeks słownikowy wartości 0, czyli #0

10. idę dalej, ostatnio analizowana wartość i jej następca to teraz:
    0, 0, 2, 2, 2, 0, 0, 2, 2, 2, 0, 0, 1, 0, 0, 3
11. jej również nie ma w słowniku - dopisuję, a do kodu dodaję wartość #2, bo to indeks słownikowy sekwencji mniejszej o ostatnią wartość
12. obecnie nasz kod to #4#0#0#2, a słownik:

#0	0
#1	1
#2	2
#3	3
#4	Clear Code
#5	End of Information Code
#6	0, 0
#7	0, 2
#8	2, 2

13. analizuję kolejne wartości:
    0, 0, 2, 2, 2, 0, 0, 2, 2, 2, 0, 0, 1, 0, 0, 3
14. tym razem sekwencję 2, 2 mam już w słowniku, więc biorę kolejną wartość:
    0, 0, 2, 2, 2, 0, 0, 2, 2, 2, 0, 0, 1, 0, 0, 3
15. sekwencji 2, 2, 0 nie ma w słowniku, więc dopisuję ją, a do kodu dodaję indeks słownikowy sekwencji mniejszej o ostatnią wartość, czyli dla 2, 2 jest to #8
16. analizuję dalej:
    0, 0, 2, 2, 2, 0, 0, 2, 2, 2, 0, 0, 1, 0, 0, 3
17. sprawdzam, czy mam taką wartość w słowniku; mam więc, biorę kolejną wartość:
    0, 0, 2, 2, 2, 0, 0, 2, 2, 2, 0, 0, 1, 0, 0, 3
18. jej nie mam, także dodaję do słownika a do kodu indeks odpowiadający w słwoniku sekwencji 0, 0, czyli #6
19. dalej:
    0, 0, 2, 2, 2, 0, 0, 2, 2, 2, 0, 0, 1, 0, 0, 3 -> rozszerzam
    0, 0, 2, 2, 2, 0, 0, 2, 2, 2, 0, 0, 1, 0, 0, 3 -> dopisuję do słownika 2, 2, 2 a do kodu #8
    0, 0, 2, 2, 2, 0, 0, 2, 2, 2, 0, 0, 1, 0, 0, 3 -> dopisuję do słownika 2, 0, a do kodu #2
    itd...
20. Na końcu zostanie Ci wartość 3, dla której sekwencję 0, 0, 3 wpiszesz do słownika, ale jej samej nie użyjesz. W kodowaniu LZW, to co pozostaje, przepisujemy do kodu jako indeks w słowniku, czyli w naszym przypadku na końcu dodajemy #3
21. Ostatnim krokiem jest dopisanie wartości End of Information, która w naszym słowniku ma wartość #5

Polecam, by zrobić sobie to kodowanie samodzielnie na kartce.

Twój kod powinien wyglądać:
#4#0#0#2#8#6#8#2#6#1#6#3#5

Mamy więc do zakodowania 13 wartości, a nie jak poprzednio, 16. Na ten moment nie jest to może spektakularny zysk, ale przy większych obrazach jest on znacznie bardziej zauważalny.

Flexible Code Size

Teraz, jeśli już wiesz jak przeprowadzić taką kompresję, dodajmy jedno "utrudnienie". Otóż GIF korzysta z tzw. "Flexible Code Size", czyli zmiennej ilości bitów, na których zapisujemy nasze wartości. Już tłumaczę.

Możemy przyjąć, że będziemy zapisywać pierwszą wartość na np. 3 bitach, drugą na 4, a trzecią na 6. Mając więc do zakodowania wartości 1, 2, 3 i naszą zmienną ilość bitów, zapiszemy je jako 0010010000011. Podobnie będziemy chcieli zapisać nasze zakodowane wartości z LZW. Skąd jednak wiedzieć, ile bitów poświęcać na każdą wartość?

Zaczynamy od wartości Start Code Size wedle tabeli:

Start Code Size	numery w słowniku		
	kolory	clear code	EOI code
3	#0-#3	#4	#5
4	#0-#7	#8	#9
5	#0-#15	#16	#17
6	#0-#31	#32	#33
7	#0-#63	#64	#65
8	#0-#127	#128	#129
9	#0-#255	#256	#257

Dla naszego przykładowego GIFa, będziemy mieli więc początkową wartość Start Code Size równą 3. Na takiej ilości bitów zapiszemy pierwsze wartości.

W miarę posuwania się dalej z kodowaniem, będziemy chcieli rozszerzyć ilość bitów, na których zapisujemy. Dla wygody więc stwórzmy jakby równoległą do naszego kodu tablicę z wartościami odpowiadającymi ilości bitów, na których te wartości będziemy zapisywać.

Pierwsze wartości będziemy zapisywać na current code size = Start Code Size. Jednak, ta wielkość będzie się zmieniać. Za każdym dodaniem nowej wartości do słownika (a przed wpisaniem wartości do kodu) sprawdzamy na jakim miejscu w tym słowniku ta wartość się znajdzie. Jeśli będzie to miejsce numer $2^{\text{current code size}}$ to zwiększamy current code size o jeden i

zawsze po wpisaniu wartości do kodu, wpisujemy w równoległej tablicy aktualny current code size.

Dla naszego przykładu, zwiększymy więc current code size z wartości 3 na wartość 4, gdy do słownika wpiszemy wartość o indeksie #8, czyli 2^3.

Zapis do bajtów

Powiedzmy, że po takim kodowaniu, mamy już gotową tablicę code:

code	4	0	0	2	8	6	8	2	6	1	6	3	5
code size	3	3	3	4	4	4	4	4	4	4	4	4	4

zapis do bajtów odbywa się w następujący sposób:
mamy 100 000 000 0010 1000 0110 ...
zapisujemy to po kolei (spacje oddzielają poszczególne bajty):
->100
->000100
->0 00000100
->00100 00000100
->1 00000100 00000100
->10001 00000100 00000100
->0 11010001 00000100 00000100
itd...

Jeśli na koniec jakiś bajt nie jest wypełniony “do końca”, dodajemy z przodu zera.

Jeszcze zanim wpiszemy powyższe bajty, warto dodać, że zawsze tą sekwencję poprzedzamy wartością Start Code Size pomniejszoną o jeden i zapisaną heksadecymalnie (jako jeden bajt) i ilością bajtów w bloku danych (również jako jeden bajt). To oznacza, że dla większych plików trzeba będzie zapisywać wiele takich bloków wedle sekwencji:

start code size	ilość bajtów bloku		...			ilość bajtów bloku		...			ilość bajtów bloku		...		...

Na koniec

Ostatnie, co musimy zrobić to dodać wartość 00 jako znak końca danych obrazu i 3B jako zakończenie - ten znacznik zawsze jest ostatnim, jaki mamy w pliku GIF

Zapraszam do eksperymentów, próbujcie swoich sił i twórzcie GIFy!

pomocne źródła:

1. https://www.w3.org/Graphics/GIF/spec-gif89a.txt
2. http://giflib.sourceforge.net/whatsinagif/index.html
3. https://en.wikipedia.org/wiki/GIF