Multimédia

Studijní opora

Autoři: David Bařina, Pavel Zemčík
Datum: 10. října 2025

Obsah

1 Úvod **4**

2 Multimediální data **5**
2.1 Barva a barevné modely 5
2.1.1 Barevné modely a prostory 6
2.1.2 YCbCr 6
2.2 Formát pixelu 6
2.2.1 Podvzorkování barvonosných složek 6
2.2.2 Planární a prokládané formáty pixelů 7
2.3 Formát obrazu 7
2.4 Další pojmy 11
2.5 Přehrávač videa 12
2.6 Jednoduchý videokodek 13

3 Kompresní techniky **16**
3.1 Základní metody 18
3.1.1 RLE 18
3.1.2 Kódování rozdílů 20
3.1.3 Prediktivní kódování 20
3.1.4 Vícerozměrná data 22
3.2 Entropická kódování 23
3.2.1 Unární kódování 23
3.2.2 Golombovo-Riceovo kódování 23
3.2.3 Shannonovo-Fanovo kódování 24
3.2.4 Huffmanovo kódování 24
3.2.5 Aritmetické kódování 25
3.2.6 Kontextová komprese 26
3.3 Slovníkové metody 27
3.3.1 LZ77 27
3.3.2 LZ78 28
3.3.3 LZW 29
3.3.4 Deflate 30
3.4 Shrnutí 33

4 Formáty obrazu **34**
4.1 Obecné formáty . . . 34
4.1.1 BMP . . . 34
4.1.2 TIFF . . . 37
4.2 Bezeztrátové formáty . . . 38
4.2.1 GIF . . . 38
4.2.2 PNG . . . 41
4.2.3 JPEG-LS . . . 43
4.3 Ztrátové formáty . . . 45
4.3.1 JPEG . . . 46
4.3.2 JPEG 2000 . . . 52
4.4 Shrnutí . . . 56

5 Formáty videa **57**
5.1 Pojmy a metody . . . 57
5.2 Jednoduché formáty . . . 65
5.2.1 Nekomprimované video . . . 65
5.2.2 Microsoft RLE . . . 65
5.2.3 Huffyuv . . . 66
5.2.4 MJPEG . . . 66
5.2.5 DNxHD . . . 66
5.2.6 JPEG 2000 . . . 67
5.3 Hybridní kodéry . . . 67
5.3.1 H.261 . . . 67
5.3.2 MPEG-1 Video . . . 67
5.3.3 MPEG-2 Video, H.262 . . . 68
5.3.4 MPEG-4 Visual . . . 69
5.3.5 MPEG-4 AVC, H.264 . . . 69
5.3.6 VP3, Theora . . . 69
5.3.7 VP8 . . . 70
5.3.8 Dirac . . . 71
5.4 Shrnutí . . . 71

6 Jednoduchá multimediální rozhraní **73**
6.1 OpenCV . . . 73
6.1.1 Základy . . . 73
6.1.2 Práce s obrazem . . . 74

7 Multimediální frameworky **76**
7.1 Video for Windows . . . 78
7.1.1 Přehrávač . . . 78
7.1.2 Videokodek . . . 80
7.2 DirectShow . . . 83

7.2.1 Základy 83
7.2.2 Přehrávač 86
7.2.3 Videokodek 86
7.2.4 Přehled API 89
7.3 FFmpeg 90
7.3.1 Základy 91
7.3.2 Přehrávač 94
7.3.3 Komprese videa 97
7.3.4 Videokodek 98
7.3.5 Přehled API 101
7.4 GStreamer 104
7.4.1 Základy 105
7.4.2 Přehrávač 107
7.4.3 Videokodek 110
7.4.4 Přehled API 112

Kapitola 1

Úvod

Tento text je studijní oporou k magisterskému předmětu Multimédia (MUL) na FIT VUT v Brně. Cílem opory je seznámit studenty s multimediálními formáty a API. Opora pokrývá pouze část probírané látky. Jednotlivé kapitoly odpovídají přednáškám. Důležité pojmy jsou sázeny v okraji stránky jako zde. Výklad předpokládá základní znalosti zpracování obrazu. Ty lze nalézt např. ve studijní opoře k předmětu Zpracování obrazu (ZPO). Pokud naleznete chybu, pište na adresu ibarina@fit.vutbr.cz.

◁ pojem

Kapitola 2

Multimediální data

Multimediálními daty se podle okolností mohou rozumět např. zvukové signály, statické obrazy, animace (sekvence obrazů), videa (obraz i zvuk), vícerozměrná medicínská data (statická i v pohybu), geologická data, vektorová grafika (2D nebo 3D) atp. Typicky se tedy jedná o nestrukturovaná a často velmi objemná data (v nekomprimované formě). K těmto datům mohou být přiložena metadata (např. tagy ID3 k formátu MP3).

Tato kapitola se zabývá především obrazem a sekvencemi obrazů a popisuje jejich formát a uložení v paměti. K tomu je třeba nejdříve definovat barvu, barevný model a pixel.

2.1 Barva a barevné modely

Barva je vjem světla z viditelné části spektra (vlnové délky cca od 400 do 700 nm) elektromagnetického záření, které dopadá na sítnici oka. Sítnice lidského oka obsahuje dva typy světlocitlivých receptorů (fotoreceptorů) – čípky a tyčinky. Čípky jsou trojího druhu, kde každý má odezvu na trochu jinou spektrální křivku. Proto jsou primární barvy monitorů R, G a B. Tyčinky jsou naopak citlivé jen na tmu a světlo. Na rozdíl od čípků jsou mnohonásobně citlivější (vidění v noci).

Jas neboli luminance je fotometrická[1] veličina, která měří svítivost zdroje světla na jednotku plochy. Lidské oko je na jas velmi citlivé. Pokud se hovoří o digitálním obraze, používá se pojem luma. Značí se Y a lze ji vyjádřit kombinací složek R, G a B. Definice záleží na použitém standardu. Pro doporučení ITU-R BT.601 je definice uvedena v (2.1). Tato definice říká, že lidské oko je velmi citlivé na zelenou, trochu méně na červenou a mnohem méně na modrou.

◁ jas

◁ luma

$$Y = 0{,}299R + 0{,}587G + 0{,}114B \tag{2.1}$$

[1] vztažená k lidskému vidění

2.1.1 Barevné modely a prostory

Barevný model je matematický model, který popisuje barvu. Například model RGB popisuje barvu jako trojici nezávislých složek R (červená), G (zelená) a B (modrá) s hodnotami 0 až MAX. Barevný prostor udává každé této složce (komponentě) barevnost, nejčastěji v chromatickém diagramu CIE xy. Rozsah barev, který je konkrétní barevný prostor RGB schopen pokrýt, je udán trojúhelníkem mezi vrcholy R, G a B a nazývá se gamut. Pokud je obrázek reprezentován pixely v modelu RGB a není k němu informace o barevnosti (chromatičnosti) jednotlivých složek R, G a B, nelze jej věrně zobrazit nebo vytisknout.

◁ barevný model

2.1.2 YCbCr

Barevný model YC_bC_r[2] je transformací modelu RGB. Existuje několik variant této transformace (podle použité normy, rozsahu vstupních a výstupních hodnot). V (2.2) je uveden převod použitý ve formátu JFIF pro 8bitové RGB i YC_bC_r.

◁ YC_bC_r

$$\begin{bmatrix} Y \\ C_b \\ C_r \end{bmatrix} = \begin{bmatrix} 0 \\ 128 \\ 128 \end{bmatrix} + \begin{bmatrix} +0{,}299 & +0{,}587 & +0{,}114 \\ -0{,}16875 & -0{,}33126 & +0{,}5 \\ +0{,}5 & -0{,}41869 & -0{,}08131 \end{bmatrix} \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix} \tag{2.2}$$

2.2 Formát pixelu

Formátem pixelu se rozumí použitý barevný model a způsob reprezentace jeho složek v paměti. Například pixely ve formátu RGB s 8 bity na každý kanál ($2^8 = 256$ úrovní) mohou být uloženy v modelu RGB24 (RGB888) nebo BGR24 podle pořadí jednotlivých bajtů v paměti. Formáty ARGB, RGBA nebo RGBA32 přidávají ještě další bajt pro alfakanál. Pro identifikaci formátu pixelu se používá mnoho mírně odlišných notací. Např. YUV444 nebo YCbCr444 znamená 3 bajty na pixel v nepodvzorkovaném formátu YUV nebo YC_bC_r. Pokud je použito podvzorkování barvonosných složek (C_b a C_r), tvoří několik pixelů bloky, které mají některou (barvonosnou) složku společnou.

◁ formát pixelu

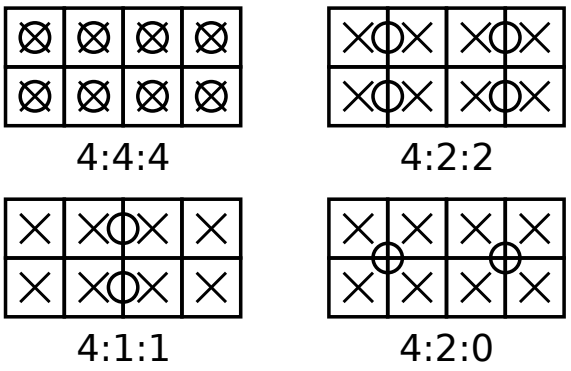
◁ RGB24

2.2.1 Podvzorkování barvonosných složek

Lidský zrak je mnohem citlivější na změny jasu Y než barvonosných složek C_b a C_r. Tyto barvonosné složky jsou proto v multimédiích (analogový či digitální televizní přenos, obrázek ve formátu JFIF/JPEG) často podvzorkovávány. Ztráta informace je přitom člověkem běžně nezpozorovatelná. Několik pixelů, které mají společnou některou barevnou složku, tvoří blok neboli makropixel.

◁ makropixel

[2] správně $Y'C_bC_r$ pro digitální obraz



Obrázek 2.1: Často používaná podvzorkování. Čtverec označuje pixel, křížek jasovou složku Y, kolečka barvonosnou složku C_b a C_r (obě v jednom kolečku).

Blok má typicky šířku 4 a výšku 2 vzorky. Podvzorkování se značí trojicí čísel ve formátu *J:a:b*, kde J udává šířku bloku (vždy 4), a je počet barvonosných složek horního řádku a b je počet těchto složek ve spodním řádku. Často používaná schémata podvzorkování jsou 4:4:4 (nepodvzorkováno), 4:2:2 (barvonosné složky horizontálně podvzorkovány na polovinu), 4:1:1 (horizontálně podvzorkovány na čtvrtinu) a 4:2:0 (horizontálně i vertikálně na polovinu). Ve formátu JFIF/JPEG se nejčastěji používá 4:2:2 a 4:2:0. Geometrické středy (centroidy) chromatických složek pixelů mohou ležet v různých polohách vůči středu jasové složky Y. Jedno z možných rozvržení ukazuje obr. 2.1. Blokem (makropixelem) může být myšlena také skupina 2×2 (4:2:0), 4×1 (4:1:1), 2×1 (4:2:2) nebo 1×1 (4:4:4) pixelů.

2.2.2 Planární a prokládané formáty pixelů

O těchto formátech má smysl hovořit pouze tehdy, když se pixel skládá z více složek, např. tři složky R, G a B. Planární (*planar*) formát obrazu nebo organizace framebufferu říká, že jednotlivé složky budou v paměti uloženy odděleně za sebou. V případě RGB24 by to znamenalo $R_0R_1R_2 \ldots G_0G_1G_2 \ldots B_0B_1B_2 \ldots$. Naproti tomu prokládaný formát (*packed*, *chunky*, *interleaved*) jednotlivé kanály pixelu prokládá. V tomto případě by RGB24 bylo uloženo jako $R_0G_0B_0R_1G_1B_1 \ldots$. Mimoto existuje semiplanární (biplanární) formát, ve kterém jsou tři složky (Y, U, V) uloženy ve dvou rovinách. Například pro NV12 je uložena nejprve složka Y, pak prokládaně složky U společně s V. Tabulky 2.1 a 2.2 ukazují přehled nejpoužívanějších formátů prokládaných a planárních pixelů.

2.3 Formát obrazu

Na statický barevný obraz lze pohlížet jako na dvourozměrnou diskrétní funkci s parametry (x, y), jejíž hodnotou je barva obrazového bodu. Ve světě počítačů je běžně používán barevný model RGB (např. elektronová děla v monitoru CRT). Barva je v RGB určena trojicí (r, g, b). Hodnota $(0, 0, 0)$ označuje černou barvu. Změna byť jediné ze složek (r, g, b) ovlivní jas bodu. V televizní technice jsou naopak pro zachování zpětné kompatibility s černobílými televizory rozšířeny deriváty modelu YUV. Pro ztrátovou kompresi je vhodné použít právě některý z derivátů tohoto

formát pixelu	pod-vzor-kování J:a:b	efektivní hloubka pixelu [bit]	hloubka makro-pixelu [bit]	pořadí komponent	velikost komponent [bit]
YUY2 YUYV422 YUYV V422	4:2:2	16	32	$Y_0U_0Y_1V_0$	8:8:8:8
UYVY UYVY422 Y422	4:2:2	16	32	$U_0Y_0V_0Y_1$	8:8:8:8
YVYU	4:2:2	16	32	$Y_0V_0Y_1U_0$	8:8:8:8
RGB RGB24		24	24	$R_0G_0B_0$	8:8:8
BGR BGR24		24	24	$B_0G_0R_0$	8:8:8
RGB565		16	16	$R_0G_0B_0$	5:6:5
RGB555		15	16	$0R_0G_0B_0$	1:5:5:5
RGBA		32	32	$R_0G_0B_0A_0$	8:8:8:8
BGRA		32	32	$B_0G_0R_0A_0$	8:8:8:8
ARGB		32	32	$A_0R_0G_0B_0$	8:8:8:8
ABGR		32	32	$A_0B_0G_0R_0$	8:8:8:8

Tabulka 2.1: Často užívané prokládané (*packed*) formáty pixelů. Model YUV ve skutečnosti značí model $\mathrm{YC_bC_r}$. Složka U tak značí C_b, složka V značí C_r. Pořadí a velikost komponent je udána v rámci makropixelu.

formát pixelu	podvzorkování J:a:b	efektivní hloubka pixelu [bit]	hloubka makropixelu [bit]	pořadí komponent	velikost komponent [bit]
IYUV I420 YUV420 YUV420P YU12	4:2:0	12	48	$4Y, U, V$	32:8:8
YVU420 YV12	4:2:0	12	48	$4Y, V, U$	32:8:8
I422	4:2:2	16	32	$2Y, U, V$	16:8:8
I444	4:4:4	24	24	Y, U, V	8:8:8
YUV411P	4:1:1	12	48	$4Y, U, V$	32:8:8

Tabulka 2.2: Často používané planární formáty pixelů. YUV je $\mathrm{YC_bC_r}$. Čísla před složkou ve sloupci pořadí jsou násobky počtu těchto složek na makropixel.

barevného modelu, ve kterém je barva popsána jako trojice (y, u, v). Výhodou je oddělení jasu barvy (složka y), na kterou je lidské oko velmi citlivé, od barvonosných informací (složky u a v), na které je lidský zrak citlivý méně. S touto znalostí se je možno na barvonosných složkách dopustit větší ztráty informací než na složce y, což vede k lepšímu poměru kvality ku velikosti zkomprimovaného obrazu.

Základním modelem, se kterým při zpracování obrazu pracujeme, je model RGB, kde je pro každou složku rezervován 1 bajt. Hodnoty celočíselných složek R, G a B se tak pohybují v intervalu $\langle 0; 255 \rangle$.

Standard JPEG File Interchange Format z roku 1992 uvádí vztahy, které umožňují převést RGB s 8 bity na kanál na variantu $\mathrm{YC_bC_r}$ definovanou tímto standardem. Hodnoty celočíselných složek Y, $\mathrm{C_b}$, $\mathrm{C_r}$ získané dle zmíněných vztahů se také pohybují v intervalu $\langle 0; 255 \rangle$. Převod RGB na YCbCr lze provést podle vztahů (2.3).

$$Y = 0{,}299R + 0{,}587G + 0{,}114B \tag{2.3a}$$
$$C_B = -0{,}1687R - 0{,}3313G + 0{,}5B + 128 \tag{2.3b}$$
$$C_R = 0{,}5R - 0{,}4187G - 0{,}0813B + 128 \tag{2.3c}$$

Převod zpět pak podle (2.4).

$$R = Y + 1{,}402(C_R - 128) \tag{2.4a}$$
$$G = Y - 0{,}34414(C_B - 128) - 0{,}71414(C_R - 128) \tag{2.4b}$$
$$B = Y + 1{,}772(C_B - 128) \tag{2.4c}$$

Výpočet výše probíhá v reálných číslech, které ovšem nelze na současných počítačích přesně reprezentovat. Pokud chceme mít jistotu, že převodem do jistého

barevného modelu a zpět nedojde ke ztrátě informací, musíme použít některý z modelů, u kterých je zaručena reverzibilita převodu. Takové modely se používají pro bezeztrátovou kompresi obrazu. Příkladem je RCT (*Reversible Color Transform*) použitý ve standardu JPEG 2000. Převod RGB na RCT lze provést podle (2.5), kde $\lfloor x \rfloor$ je zaokrouhlení x směrem dolů.

$$Y = \left\lfloor \frac{1}{4}(R + 2G + B) \right\rfloor \tag{2.5a}$$

$$C_B = B - G \tag{2.5b}$$

$$C_R = R - G \tag{2.5c}$$

Zpět pak podle (2.6).

$$G = Y - \left\lfloor \frac{1}{4}(C_B + C_R) \right\rfloor \tag{2.6a}$$

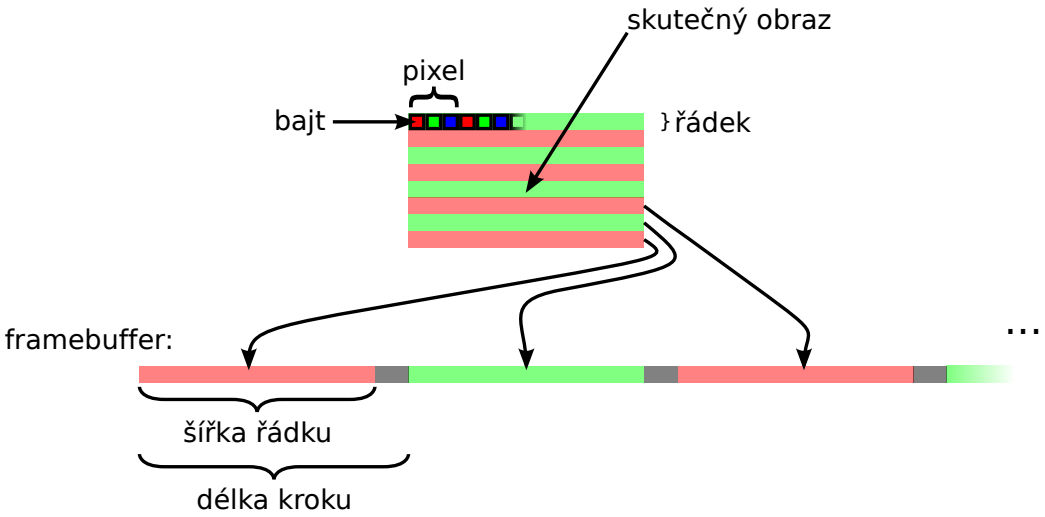
$$R = C_R + G \tag{2.6b}$$

$$B = C_B + G \tag{2.6c}$$

Formát pixelu (obrazového bodu) popisuje přesné uložení pixelu v paměti. Například jako RGB24 se označuje formát pixelu uloženého v modelu RGB, kde je pro každou jeho složku vyhrazeno přesně 8 bitů (celistvý bajt), celkem tedy 24 bitů na pixel. Na každou jeho složku připadá $2^8 = 256$ úrovní neboli kvantovacích hladin. Celkem lze vyjádřit $256^3 = 16\,777\,216$ různých barev. V paměti leží hodnoty těchto bajtů těsně za sebou v pořadí R-G-B, nikoli opačně. Jeden formát se často označuje více způsoby, např. RGB888 je to stejné jako RGB24. Naopak je třeba nezaměnit RGB24 s BGR24, kde jsou bajty v paměti uloženy v opačném pořadí. Další známé formáty pro model RGB jsou např. RGB555 (na 16 bitech), RGB565 (16 bitů), RGB32 (bez alfa kanálu, MSB nevyužit), ARGB32 a RGBA32 (alfa kanál, 32 bitů). Dále pro model YUV s podvzorkováním chromatických složek jsou to YUY2, UYVY, YVYU (horizontální), YV12 a I420 (horizontální i vertikální).

Nekomprimovaný obraz je v paměti uložen v tzv. framebufferu. Ten se skládá z pixelů v daném formátu, např. RGB24. Pixely v řádku jsou uloženy souvisle za sebou zleva doprava. Jednotlivé řádky jsou také uloženy za sebou, ale mohou být také v obráceném pořadí (odspodu nahoru), což je typické pro formát DIB v Microsoft Windows. Nemusejí být navíc uloženy souvisle. Mezi jednotlivými řádky může být z důvodu zarovnání nebo rychlého přístupu (omezení cache CPU) několik nevyužitých bajtů. Délka řádku spolu s touto nevyužitou výplní se nazývá délka kroku (*stride*). Uvedený způsob uložení ukazuje obrázek 2.2.

Parametry obrazu uloženého v paměti jsou tedy velikost formátu pixelu (pro RGB24 3 bajty), počet sloupců a řádků obrazu, krok (*stride*) v bajtech, případně pro přístup k jednotlivým kanálům také jejich velikost (pro RGB24 má každý kanál 1 bajt). Pro indikaci, že se jedná o obraz uložený odspodu nahoru, může mít nastaven záporný krok nebo výšku.



Obrázek 2.2: Souvislost dvourozměrné představy obrazu s obrazem uloženým ve framebufferu. Šedá místa framebufferu jsou nevyužita. Obraz může být ve framebufferu umístěn vzhůru nohama (spodní řádky napřed).

Příklad Následující útržek zdrojového kódu v jazyce C ukazuje výpočet adresy pixelu nebo jeho složky.

```
addr = ptr + stride * row + sizeof(pixel) * col + sizeof(channel) * ch;
```

Ukazatel `ptr` ukazuje na první nebo poslední řádek obrazu. Proměnná `stride` udává krok mezi následujícími řádky, `row` je požadovaný řádek, `col` požadovaný sloupec a `ch` složka (kanál) pixelu.

2.4 Další pojmy

Při zpracování videa má dále smysl mluvit o snímkové frekvenci. Snímková frekvence je frekvence, s jakou zobrazovací zařízení zobrazuje jednotlivé unikátní snímky, případně záznamové zařízení snímky zachycuje. Snímková frekvence (FPS, z anglického *frames per second*) se udává v jednotkách hertzích (Hz) nebo často nesprávně ve „fps“. Ve filmu, televizi a při zpracování videa existuje několik běžně používaných hodnot snímkové frekvence: 24, 25, 30, 50 a 60 Hz. Přípona „i“ znamená prokládané video (*interlaced*), přípona „p“ neprokládané (*progressive*). Často používané hodnoty jsou zmíněny níže.

60i je standard NTSC televize, po přechodu na barevné vysílání to je $60\times1000/1001 = 59{,}94$.

50i je standard pro PAL a SECAM, 50 prokládaných = 25 skutečných snímků.

30p	pro speciální typy naskenovaných klasických filmů (70 mm) + videoklipy.
24p	je neprokládaný, používaný u klasických filmů, pro převod do NTSC je film zpomalen na 23,976, pro PAL/SECAM zrychlen na 25.
25p	je odvozeno od PAL televize (50i), vhodné pro zobrazení na PC, LCD monitorech.
50p, 60p	jsou neprokládané formáty použité u HDTV.
120p	mají některé kamery (GoPro).

U digitálního obrazu a videa lze mluvit o tzv. rozlišení, což je jednoduše počet jeho sloupců a řádků. Zapisuje se ve tvaru „sloupce×řádky“ a počet obrazových bodů (pixelů) se získá právě jejich součinem. Například obraz o rozlišení 640×480 má zhruba 307 tisíc pixelů, 2048×1536 pak 3,1 Mpx. Rozlišení se udává také u zobrazovacích zařízení (např. televize). To však nemusí být shodné s rozlišením původní obrazu, který se na celé jeho ploše zobrazuje. Dochází zde k interpolaci původního obrazu do rozlišení obrazovky.

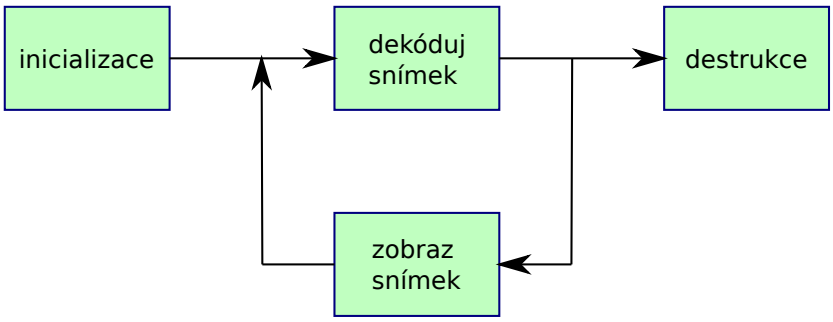
Nejčastější televizní rozlišení jsou uvedena níže. Přípona má stejný význam jako u snímkové frekvence.

480i	SDTV (NTSC, 720×480 rozdělené do dvou 240řádkových oblastí)
576i	SDTV (PAL, 720×576 rozdělené do dvou 288řádkových oblastí)
480p	EDTV (NTSC, 720×480)
720p	HDTV (1280×720)
1080i	HDTV (1280×1080, 1440×1080 nebo 1920×1080 rozdělené do dvou 540řádkových oblastí)
1080p	HDTV (1920×1080)
4K	UHDTV (3840×2160)
8K	UHDTV (7680×4320)

Poměr stran (*aspect ratio*) udává poměr horizontální a vertikální strany snímku videa. Mezi typické hodnoty patří 4:3 (původní televize a počítačové obrazovky) a 16:9 (HDTV). ◁ poměr stran

2.5 Přehrávač videa

Přehrávač videa je vhodné postavit nad některým multimediálním frameworkem. Jednoduchý přehrávač nejprve požádá o otevření multimediálního souboru (např. AVI). Dalším krokem bude nalezení vhodné stopy s videem a její otevření (inicializace kodeku). Následně je možné opakovaně volat funkci pro dekompresi jednoho snímku. Tato funkce načte z multimediálního kontejneru odpovídající fragmenty dat a volá nad nimi funkce kodeku. Pokud jsou všechny komprimované snímky klíčové, stačí zavolat funkci kodeku pro dekompresi snímku pouze jednou.



Obrázek 2.3: Dekódování videosekvence (např. přehrávač videa). Na počátku je třeba najít a inicializovat vhodný kodek (alokace pameti pro rozdílové snímky, parametry). Následně je možné opakovaně volat funkci pro dekompresi snímku.

2.6 Jednoduchý videokodek

Tato sekce popisuje příklad jednoduchého kodeku videa ve formátu RGB24 s použitím metody RLE. Na obraz (prozatím ve stupních šedi) lze metodu RLE aplikovat po řádcích. Nejprve je tedy po pixelech zleva doprava zpracován celý první řádek, pak druhý až nakonec poslední. Mezi jednotlivými řádky můžeme resetovat stav kodéru. Reset znamená, že je na výstup poslána nezbytná sekvence a předchozí symbol a počet jeho opakování jsou nastaveny na vhodné výchozí hodnoty. Barevný obraz je zpracován postupně po kanálech, např. pro formát RGB24 nejprve celá složka R, pak celá G a nakonec B. Zmíněný postup zpracování obrazu je jen jednou z mnoha možností, jak na něj RLE kompresi aplikovat. Možnou modifikací může být převod do jiného barevného modelu, prokládané zpracování barevných kanálů, zpracování pixelů v jiném pořadí než po řádcích, tedy např. v pořadí cik-cak (zig-zag), Mortonova rozkladu (Z křivky) nebo Hilbertovy křivky.

Právě popsaná implementace RLE komprese obrazu v jazyku C je k dispozici na stránkách předmětu Multimédia (MUL). Rozhraní je založeno na dvou funkcích (`compress` a `decompress`), první provede kompresi celého snímku, druhá pak jeho dekompresi. Tyto hlavní funkce procházejí obraz po pixelech a používají přitom jednoduché funkce ovládající vlastní RLE kodér a dekodér. Funkce `rle_create` inicializuje RLE (de)kodér ukazatelem na blok paměti, kam ukládat (odkud číst) komprimovaná data. Funkce `rle_reset` a `rle_reset_and_init` resetují stav (de)kodéru, tedy čítač, respektive čítač a předchozí symbol. Funkce `rle_encode` a `rle_decode` provádějí vlastní kompresi, resp. dekompresi jednoho symbolu. První jmenovaná tedy spolkne znak a může vyprodukovat do výstupu sekvenci. Naproti tomu druhá může sekvenci načíst a vždy vrátí jeden dekomprimovaný znak. Nakonec funkce `rle_flush_sequence` pošle do výstupu sekvenci předčasně (může-li vůbec co poslat). Funkce pro kompresi a dekompresi celého snímku jsou uvedeny v příkladu 2.1 a 2.2. Jako parametr obdrží obě funkce ukazatel na počátek nekomprimovaného obrázku, jeho rozměry (šířku, krok řádků a výšku) a ukazatel na komprimovaný tok dat. Funkce pro kompresi vrací velikost vyprodukovaných dat v bajtech. Tu lze srovnat

s velikostí nekomprimovaných dat, pro RGB24 (3 bajty) to bude 3×šířka×výška.

```
int compress(byte *bitmap, byte *stream, int width, int height,
    int stride)
{
    byte *ptr = stream;
    rle_state st = rle_create(&ptr);

    for(int c = 0; c < channels; c++)
    {
        for(int y = 0; y < height; y++)
        {
            rle_reset_and_init(&st, rle_init_symbol);

            for(int x = 0; x < width; x++)
                rle_encode(&st, bitmap[y*stride + x*pixel_size + c]);

            rle_flush_sequence(&st);
        }
    }

    return ptr-stream;
}
```

Příklad 2.1: Komprese jednoho snímku pomocí RLE. Funkce obdrží ukazatel na nekomprimovaný obrázek (bitmap), jeho rozměry (width, height, stride) a ukazatel pro výsledná komprimovaná data (stream). Vrací velikost komprimovaných dat.

```
int decompress(byte *bitmap, byte *stream, int width, int height,
    int stride)
{
    byte *ptr = stream;
    rle_state st = rle_create(&ptr);

    for(int c = 0; c < channels; c++)
    {
        for(int y = 0; y < height; y++)
        {
            rle_reset(&st);

            for(int x = 0; x < width; x++)
                bitmap[y*stride + x*pixel_size + c] = rle_decode(&st);
        }
    }

    return ptr-stream;
}
```

Příklad 2.2: Dekomprese jednoho snímku pomocí RLE. Funkce obdrží ukazatel pro výsledný nekomprimovaný obrázek (bitmap), jeho rozměry (width, height, stride) a ukazatel pro zdrojová komprimovaná data (stream).

Uvedené funkce pro kompresi snímku pomocí RLE lze použít na libovolný barevný (3 kanály po jednom bajtu) obrázek nebo na video, kde každý snímek komprimujeme nezávisle na ostatních (každý je klíčový). Napojení na jakýkoli multimediální framework by nemělo být složité. Např. úsek kódu pro OpenCV je uveden v příkladu 2.3. V tomto konkrétním případě mají pixely formát BGR24, což ale uvedenému kodeku nevadí.

```
Mat image = imread(imagename, CV_LOAD_IMAGE_COLOR);

byte *stream = new byte[get_max_compressed_size(image.cols,
    image.rows)];

int size = compress(image.data, stream, image.cols, image.rows,
    image.step);

decompress(image.data, stream, image.cols, image.rows, image.step);
```

Příklad 2.3: Napojení kodeku na knihovnu OpenCV. Stačí mít obrázek ve vhodném formátu a znát jeho rozměry. Blok paměti pro výsledná komprimovaná data musí být dostatečně velký (lze odhadnout).

Text této sekce se doposud věnoval spíše kompresi obrazu než videa. Pokud by se snímky videa měly komprimovat zcela nezávisle, je načrtnuté API dostačující. V tomto případě bude každý snímek klíčový. Nebude však využita podobnost po sobě jdoucích snímků videosekvence. K využití informace z okolních snímků ke kompresi snímku současného je třeba mezi jednotlivými voláními funkce `compress` udržovat určitou souvislost (kontext). Tento kontext může udržovat informace o nastavení kodéru nebo dekodéru, např. stupeň komprese, maximální vzdálenost klíčových snímků, celý předcházející klíčový snímek (vůči němu kódujeme snímek současný), další interní proměnné apod. Některá z těchto dat je možné označit za veřejné (stupeň komprese lze měnit skrze multimediální framework), jiné jsou čistě interní (předcházející klíčový snímek). Multimediální frameworky jsou navrženy tak, že umožňují jednotlivým voláním funkcí kodeku předat ukazatel na takovou datovou strukturu. Tímto způsobem je pak možné realizovat kódování pomocí rozdílových snímků (P- a B-snímky v terminologii MPEG). Bude samozřejmě potřeba upravit odpovídajícím způsobem API kodeku. Konkrétně je třeba vytvořit strukturu zaobalující všechna potřebná data (kontext), funkce pro její vytvoření a uvolnění z paměti. Dále musejí všechny funkce obdržet jako jeden z parametrů ukazatel na tuto strukturu.

Pokud vytvářený kodek neumí komprimovat či dekomprimovat libovolné vstupní či výstupní formáty videa, bude třeba vytvořit funkci, která je schopna o tomto framework informovat. Taková funkce se může jmenovat `query`. Další potřebné funkce mohou načítat/ukládat nastavení, zobrazit konfigurační dialog (platformně závislé), odhadnout velikost komprimovaného snímku apod. Kodek je tedy knihovna funkcí, z nichž nejdůležitější dvě jsou `compress` a `decompress`.

Kapitola 3

Kompresní techniky

Komprese dat je postup, který má za cíl zmenšit objem nebo datový tok dat. V multimédiích hraje komprese obzvláště podstatnou roli. Bez komprese by např. digitální fotografie o rozlišení 2048×1536 pixelů měla velikost téměř 10 MB nebo televizní vysílání ve standardu PAL datový tok 30 MB/s. Je tedy velmi žádoucí taková multimediální data komprimovat. Některá z forem komprese se vyskytuje téměř ve všech multimediálních formátech, např. v BMP, PNG, JPEG nebo částech standardu MPEG-4.

◁ ztrátová, bezeztrátová komprese

Tuto kompresi lze rozdělit na ztrátovou a bezeztrátovou. Bezeztrátová komprese zmenšuje redundanci (nadbytečnost) vstupních dat. Původní data pak lze zrekonstruovat bez zkreslení. Ztrátová komprese odstraňuje irelevantní data a to z hlediska vnímání těchto dat člověkem (psychoaukustický a psychovizuální model lidského vnímání). Při použití této komprese však nelze původní data přesně zrekonstruovat. Dochází při ní k nezvratné ztrátě informací – tedy zkreslení dekomprimovaného signálu, které však může být pro člověka nepostřehnutelné. Ztrátová komprese dosahuje řádově vyšších kompresních poměrů. V tomto případě je užitečné zmínit metody posuzující kvalitu komprimovaného obrazu (či videa nebo jiných dat). Tuto kvalitu lze posuzovat buďto objektivně (vypočte počítač) nebo subjektivně (hodnotí lidé, nákladné, časově náročné). Pro objektivní srovnání se typicky používají metody SNR, PSNR a SSIM. Tato kapitola je zaměřena na základní bezeztrátové kompresní techniky. Tyto techniky jsou použity také jako součást ztrátových kompresních metod.

◁ entropie

Jedním z nejdůležitějších pojmů z oblasti komprese dat je entropie, což je veličina udávající množství informace obsažené v jednom symbolu dat. Jednotkou této veličiny je bit. Entropii lze definovat také jako míru překvapení dekódující strany, jako informační obsah symbolu (datové zprávy) nebo jako průměrný minimální počet bitů nutný k zakódování jednoho symbolu dat. Značí se H a formálně ji lze definovat jako (3.1), kde $p(a)$ značí pravděpodobnost výskytu symbolu a z abecedy A. Tato definice však neuvažuje kontextovou informaci o výskytu symbolu (dále).

$$H = -\sum_{a \in A} p(a) \cdot \log_2 p(a) \tag{3.1}$$

Kódování je proces vzájemně jednoznačného (bijektivního) přiřazení symbolů jedné abecedy (vstupní data) symbolům abecedy druhé (kódy, kódová slova). Kódy mohou být přiřazeny jednotlivým symbolům nebo blokům vstupního toku. Například aritmetické kódování přiřazuje kód celému vstupnímu souboru dat (bitové kódy jednotlivých symbolů nelze odlišit). Takové kódování, které snižuje redundanci dat, lze nazývat kompresí. V tomto textu se pod pojmem kód rozumí jednak vlastní kódové slovo pro určitý symbol nebo soubor těchto slov (např. Huffmanův kód).

◁ kódování

Dalším důležitým pojmem je kódování s proměnnou délkou (VLC, *variable-length coding*), což je takové kódování, kde se symbolům přiřazují kódy různé délky (různého počtu bitů). V případě, že se často vyskytujícím se symbolům přiřadí krátké kódy a méně často vyskytujícím se symbolům kódy delší, jedná se o entropické kódování. Entropické kodéry mohou komprimovat data téměř optimálně vzhledem k jejich entropii. Příkladem takového kódování je Huffmanovo.

Souvisejícím pojmem je prefixový kód, což je kód s vlastností, že žádné kódové slovo není prefixem jiného kódového slova. Důsledkem této vlastnosti je to, že kód je při dekódování jednoznačný i bez oddělovačů.

Kódování a dekódování dat provádí kodér a dekodér. Kodek (z anglického *codec*, *coder–decoder*) je jejich kombinace, tedy software nebo hardware schopný zakódovat a dekódovat datový proud. Kompresi a dekompresi provádí kompresor resp. dekompresor. V kontextu této studijní opory budou pojmy kodér, dekodér a kodek užívány ve smyslu komprese dat. Je třeba nezaměňovat pojem kodek s pojmem formát. Formát je specifikace komprimovaných dat, tedy standard, a kodeky jsou konkrétními implementacemi tohoto standardu. Například kodek DivX pracuje s formátem MPEG-4.

◁ kodek

Na kompresní metody lze nahlížet podle počtu průchodů vstupních dat, které kompresor potřebuje k jejich kompresi. Jednoprůchodová metoda čte data pouze jednou. Dvouprůchodová vyžaduje ke kompresi dva kompletní průchody. Analogicky víceprůchodová metoda vyžaduje více průchodů celým souborem vstupních dat.

Mezi základní pojmy patří tzv. model dat, který kompresoru pomáhá data skutečně komprimovat (a dekompresoru dekomprimovat). Model dat může být statistický (pravděpodobnostní), kde jsou ke každému symbolu poznačeny pravděpodobnosti jejich osamoceného výskytu ve vstupním (nekomprimovaném) toku dat. Dalším typem modelu je kontextově orientovaný model dat, ve kterém jsou uloženy pravděpodobnosti výskytu jednotlivých symbolů v určitém kontextu (např. po několika předcházejících symbolech). Analogicky k modelu dat používají některé metody slovník, což je datová struktura, která obsahuje fragmenty (řetězce) nekomprimovaného souboru dat.

◁ model dat

◁ slovník

Kompresní metody lze v souvislosti s výše zmíněnými pojmy rozdělit na statistické, které využívají statistický model dat, dále na kontextové, které využívají kontextový model dat, a nakonec na slovníkové, které využívají slovník.

Z hlediska vytváření modelu dat lze metody rozdělit na statické, semi-adaptivní a adaptivní (dynamické). U statické varianty je model dat stanoven předen. Není tedy třeba jej přenášet dekodéru a ke kompresi stačí pouze jediný průchod vstupním tokem. Semiadaptivní varianta vyžaduje průchody dva. Při prvním je kompresorem sestaven model dat. Ve druhém průchodu probíhá vlastní komprese. Tento typ metod se tedy adaptuje na komprimovaná dat. Nevýhodou je nutnost přenesení vytvořeného modelu do dekompresoru. Třetí variantou je adaptivní metoda, která vyžaduje pouze jediný průchod a model dat si sestavuje za běhu a to jak v kompresoru tak i v dekompresoru. Dekompresor musí při budování modelu přesně následovat kroky kompresoru.

◁ statické, adaptivní metody

Kompresní metody lze rozdělit na symetrické a asymetrické. První z nich využívají pro kompresi i dekompresi stejného algoritmu (jen v opačném pořadí). Komprese i dekomprese má tedy stejnou paměťovou i časovou složitost. Metody, u kterých tohle neplatí, jsou asymetrické.

Podle způsobu zpracování vstupních dat lze metody rozdělit na blokové a proudové. Blokové zpracovávají vstupní data po blocích, proudové symbol po symbolu. Příkladem blokové metody je bzip2 s velikostí bloku až cca 900 kB.

Konkrétní kompresní metody lze srovnat pomocí paměťové a časové náročnosti komprese a dekomprese a především pomocí účinnosti nebo u ztrátových metod kvality při daném datovém toku. Účinnost lze popsat pomocí kompresního poměru, což je podíl velikosti komprimovaných dat k velikosti dat nekomprimovaných. Je-li tento podíl menší než 1, jedná se o kompresi, v opačném případě se jedná o expanzi vstupních dat.

◁ kompresní poměr

V používaných kompresních formátech multimediálních dat lze identifikovat dva přístupy k jejich kompresi. První přístup je použit k bezeztrátové kompresi a nazývá se prediktivní, protože kompresní metoda obsahuje prediktor právě kódovaného symbolu (např. pixelu) a dále se kóduje chyba této predikce. Druhý přístup je použit naopak ke ztrátové kompresi a lze jej nazvat jako kódování transformace. Zde je na vstupní data aplikována některá transformace (např. DCT), která se dále kóduje.

3.1 Základní metody

V následujících sekcích jsou popsány jednoduché a často používané kompresní techniky. Více než samostatně se používají jako součást složitějších postupů.

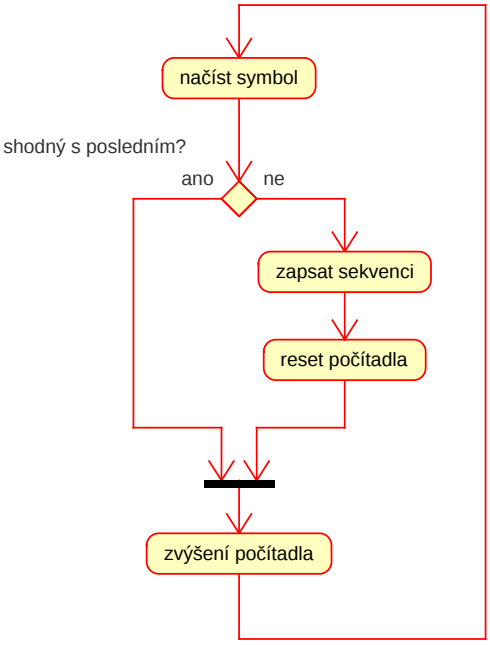
3.1.1 RLE

Metoda RLE (*run-length encoding*) je jednoduchý ale důležitý a často používaný kompresní algoritmus. Komprese spočívá v kódování sledů (posloupností, řad) stejných symbolů (znaků, pixelů) kratší sekvencí. Tato sekvence (kód) nese informaci o zakódovaném symbolu a počtu jeho opakování. Například sekvence znaků „`A A A A B C D`“ bude zakódována jako „`4×A B C D`“. Metoda má mnoho modifikací a

používá se především jako součást složitějších kompresních metod. Je použita mj. ve formátech BMP, PCX, TIFF, TGA, JPEG nebo bzip2.

Ve výstupních (komprimovaných) datech je třeba odlišit zmíněnou speciální sekvenci od ostatních symbolů abecedy. V příkladu výše musí dekodér prostě poznat, jestli se jedná o symbol „4“ nebo naopak o opakování čtyř jiných symbolů. To je možné provést tak, že se do výstupní abecedy přidá speciální „escape“ symbol, který uvozuje tyto krátké sekvence opakujících se znaků (escape sekvence). Problém nastává v případě, že další symbol není možné do abecedy přidat. Příkladem může být vstupní tok zpracovávaný po bajtech, ve kterém se mohou vyskytovat všechny symboly s ordinálními hodnotami 0–255. Možné řešení je použít jeden z méně často se vyskytujících symbolů jako uvozující escape symbol. V případě, že se ve vstupním toku takový symbol vyskytne, bude zakódován jako speciální případ escape sekvence. Jednou z možností je zakódovat ho takovou escape sekvencí, která nese informaci o jednonásobném opakování tohoto escape symbolu.

Když se ve vstupním toku vyskytne sled několika po sobě jdoucích shodných symbolů, zakóduje se tento sled krátkou escape sekvencí, která informuje o odpovídajícím počtu opakování tohoto symbolu. Naopak v případě výskytu osamoceného symbolu (ne escape symbolu), bude tento předán do výstupního toku beze změny. Optimalizací je nekódovat příliš nízký počet opakování, kdy je vytvořená escape sekvence delší než nezměněné symboly.



Obrázek 3.1: Diagram popisující jednoduchou představu o funkování RLE kodéru.

Algoritmus provádějící RLE kompresi bude číst vstupní tok symbol po symbolu. Přitom bude zjišťovat, zdali je právě přečtený symbol shodný se symbolem předchozím. V případě shody jen zvýší počítadlo opakujících se symbolů. Naopak v případě neshody pošle do výstupního toku buď osamocený (minulý) symbol nebo vhodnou escape sekvenci, dále se resetuje počítadlo (na hodnotu 1). Chování je znázorněno v diagramu na obrázku 3.1. Je patrné, že ne při každém „spolknutém“ vstupním symbolu je vyprodukován výstup. Uvedený popis kodéru však neřeší počátek a ukončení komprese. Další vadou je možnost přetečení počítadla.

3.1.2 Kódování rozdílů

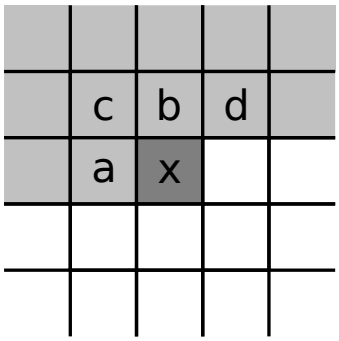
Kódování rozdílů (delta kódování, relativní kódování) je jednoduchou metodou, která předchází vlastní kompresi. Jejím účelem je zvýšit další komprimovatelnost dat. Lze ji chápat i jako jednoduchou prediktivní metodu. Používá se stejně jako v předchozím případě jako součást složitějších metod. Jádro spočívá v nahrazení vstupní hodnoty za její rozdíl proti hodnotě předcházející. Jedná se vlastně o jednoduchou predikční metodu. Snaží se predikovat (předpovědět) současný symbol tak, že jako odhad použije symbol předchozí. Na výstup pak pošle pouze jejich rozdíl (chybu predikce).

Příklad

V následujícím příkladu je sekvence hodnot z intervalu $\langle 16, 35\rangle$ se střední hodnotou 27 transformována na sekvenci z intervalu $\langle -6, 14\rangle$ se střední hodnotou 1,375. Základním předpokladem vedoucím ke kompresi je skutečnost, že následující část kompresního řetězce bude schopna zakódovat transformované hodnoty kratšími kódovými slovy. Jinak řečeno malá čísla koncentrovaná okolo 0 se v další části lépe komprimují.

```
17 16 18 32 35 35 34 28 28 → -1 +2 +14 +3 0 -1 -6 0
```

3.1.3 Prediktivní kódování



Obrázek 3.2: Prediktory predikují hodnotu pixelu x z okolních pixelů a–d. Komprese probíhá po řádcích. Světle šedé pixely jsou již známy dekodéru a mohou být použity k predikci.

Kódování rozdílů je nejjednodušší prediktivní (predikční) metodou. Prediktor se snaží predikovat právě kódovaný symbol (znak, pixel) ze symbolů již zakódovaných (tedy známých i dekódující straně). Výstupem jsou chyby predikce neboli rozdíly predikovaných hodnot od hodnot skutečných. Dekodér spočítá stejně jako kodér předpokládanou hodnotu právě dekódovaného symbolu a přičte k ní obdržený rozdíl. Podle počtu hodnot použitých k predikci se hovoří o řádu prediktoru. Kódování rozdílů používá tedy prediktor prvního řádu, protože k predikci hodnoty použije pouze jednu předchozí hodnotu. Prediktory lze aplikovat na několik bezprostředně předcházejících symbolů (1D), na několik sousedních pixelů (2D), sousedních voxelů (3D) nebo třeba koeficientů stromu DWT.

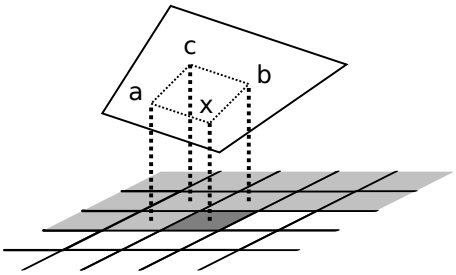
◁ prediktor

◁ řád prediktoru

Prediktory se dále dělí na lineární (vážený průměr několika okolních pixelů, kódování rozdílů, Lossless JPEG, PNG) a nelineární (medián několika okolních pixelů, MED/LOCO-I, Paeth, GAP). U lineárních se predikovaná hodnota spočte jako vážený průměr z již přenesených hodnot. U nelineárních je na již přenesené hodnoty aplikována některá nelineární funkce, např. porovnání, minimum, maximum nebo medián.

◁ lineární, nelineární prediktory

Triviální prediktor by jako odhad x na obr. 3.2 mohl použít jednu z hodnot a, b, c nebo d. Další možností je spočíst z několika těchto sousedů průměr, popřípadě medián. K výpočtu není možné použít hodnoty bílých pixelů, protože ty ještě nejsou známy dekódovací straně.



Obrázek 3.3: Proložení hodnot a, b a c rovinou. Hodnoty těchto pixelů se použijí jako výšky z odpovídajících bodů. Výška z bodu x se spočte jako $a + b - c$.

V případě hladkého okolí bodu x může být dobrým odhadem jeho hodnoty proložení okolních bodů a, b a c rovinou (3.2a). Takový odhad však nebude dávat dobré výsledky na hranách. Proložení znázorňuje obr. 3.3, kde jsou z každého pixelu vedeny kolmice o výšce odpovídající jeho hodnotě.

$$p = a + b - c \tag{3.2a}$$

$$\hat{x} = p \tag{3.2b}$$

Například nelineární prediktor MED (3.3), který je použit ve standardu LOCO-I/JPEG-LS, předpovídá hodnotu pixelu x ze tří již zpracovaných pixelů v jeho okolí. Jedná se tedy o prediktor třetího řádu. MED se nejprve pokusí detekovat hranu. V případě úspěchu použije jako predikci hodnotu pixelu, který leží na hraně. V opačném případě (hladké okolí x) proloží okolní body rovinou a jako odhad použije bod ležící na ní, tedy hodnotu p z (3.2a).

◁ prediktor MED

$$\hat{x} = \begin{cases} \min(a,b) & : \quad c \geq \max(a,b) \\ \max(a,b) & : \quad c \leq \min(a,b) \\ p & \end{cases} \tag{3.3}$$

Právě uvedenou definici lze také zapsat jako

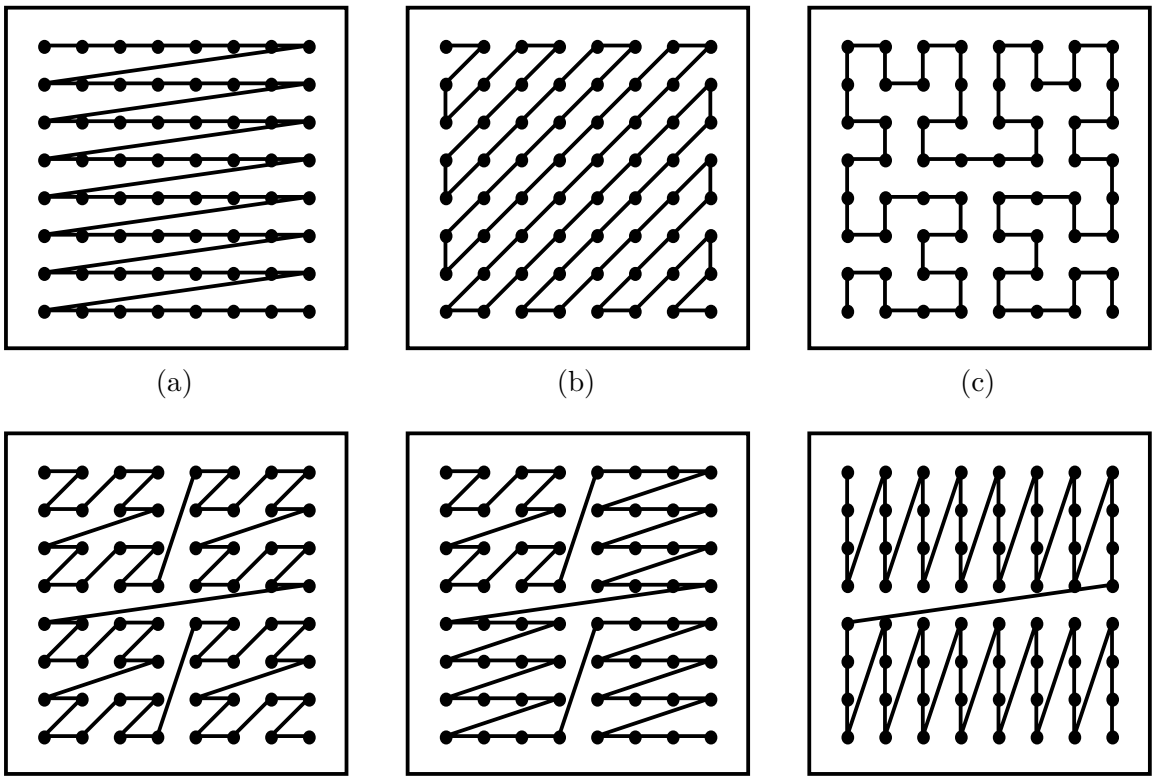
$$\hat{x} = \mathrm{med}(a,b,p). \tag{3.4}$$

Další takový nelineární editor Paeth (3.5) je použit ve standardu PNG. Ten jako predikci použije toho souseda, který je nejblíž odhadu p.

◁ prediktor Paeth

$$\hat{x} = \arg\min_{x \in a,b,c} |p - x| \tag{3.5}$$

Ve složitějším prediktoru GAP, který je použit ve standardu CALIC, je nejprve odhadnut gradient obrazu v okolí bodu x. Podle gradientu je následně detekuje přítomnost a síla hrany (slabá/silná, horizontální/vertikální). Na základě přítomnosti hrany je pak rozhodnuto o vztahu pro výpočet odhadu pixelu.



Obrázek 3.4: Průchody dvourozměrnými daty: (a) rastrový, (b) zig-zag použitý v metodě JPEG, (c) Hilbertova křivka použitá ve formátech VPx, (d) Mortonův průchod (z průchod) a (e) jeho varianta pro kódování DWT a (f) pořadí použité pro kódování DWT v algoritmu EBCOT ze standardu JPEG 2000.

3.1.4 Vícerozměrná data

Dílčí kompresní metody nejčastěji pracují s jednorozměrnými daty. Některá multimediální data jsou již svou podstatou jednorozměrná (např. zvukové signály), ostatní je nutno na jednorozměrná převést (např. dvourozměrný obraz). Tento proces se nazývá linearizace. Nejjednodušším linearizačním průchodem je rastrový (obr. 3.4a), který ◁ linearizace prochází dvourozměrný obrázek řádek po řádku. Jednotlivé řádky lze již považovat za jednorozměrný signál. Nevýhodou je ztráta informace o sousednosti původních koeficientů. Průchod lze zobecnit i pro vícerozměrná data (např. 3D prostorová data). V praxi se používají také další průchody. Zig-zag průchod (obr. 3.4b) je použit před kódováním DCT koeficientů v kompresním formátu JPEG. Mortonův průchod (obr. 3.4d) je výhodný při kompresi koeficientů DWT (diskrétní vlnkové transformace), protože projde rodičovské uzly vždy před jejich potomky. Algoritmus EBCOT ze standardu JPEG 2000 prochází tzv. bloky kódování po pruzích s výškou 4 koeficienty, v rámci těchto pruhů se koeficienty kódují po sloupcích (obr. 3.4f).

Za účelem postupného zvyšování kvality obrazu (progresivní přenos) bývá volen

ještě komplikovanější průchod daty, který dokonce snižuje dosažitelný kompresní poměr. Příkladem je algoritmus Adam7 používaný ve formátu PNG.

3.2 Entropická kódování

Entropické kódování má za úkol nahradit často se vyskytující symboly krátkými bitovými kódy. Délka těchto kódů je závislá na četnosti výskytu daného symbolu ve vstupním toku dat. Může nastat situace, kdy je symbolu přidělen kód delší, než je velikost původního symbolu. Tento případ nastává, jsou-li kratší kódy již přiděleny častěji se vyskytujícím symbolům. I přesto je komprese možná, protože k větší úspoře místa dojde právě vyjádřením častých symbolů krátkými kódy. Příkladem entropických kodérů jsou např. Golombovo-Riceovo kódování, Eliasovo gama kódování, Fibonacciho kódování, Huffmanovo kódování nebo aritmetické kódování.

3.2.1 Unární kódování

Nejjednodušší entropický kodér je unární kodér, který mapuje nezáporná celá čísla N na bitové sekvence N bitů `1` a ukončující bit `0`. Bity 0 a 1 lze samozřejmě zaměnit. Takový převod je užitečný jen, pokud jsou data seřazena vzestupně podle pravděpodobnosti jejich výskytu ve vstupním toku. Jako důsledek toho jsou častěji se vyskytujícím symbolům přiřazeny kratší bitové kódy než symbolům méně frekventovaným. Takový kód je optimální jen pro geometrické rozložení pravděpodobnosti vstupních symbolů $p(n) = 2^{-n}$. Protože přímo takové rozložení pravděpodobnosti výskytu symbolů bude velmi vzácné a protože délka kódu velmi rychle roste, je toto kódování užitečné spíše jako součást jiných entropických kodérů než přímo pro použití na kódované symboly.

N		kód
0	→	`0`
1	→	`10`
2	→	`110`
3	→	`1110`
4	→	`11110`
...		...

Tabulka 3.1: Unárního kód čísel 0–3.

3.2.2 Golombovo-Riceovo kódování

Golombovo-Riceovo (nebo jen Riceovo) kódování je speciální případ Golombova kódování. Na rozdíl od obecného Golombova kódování jej však lze na počítačích rychle a jednoduše realizovat. Kódovaní je užito např. ve standardech JPEG-LS, FLAC nebo MPEG-4 ALS. Stejně jako unární kódování mapuje i Golombův-Riceův kodér nezáporná celá čísla N na bitové kódy s proměnnou délkou. Kód je však na rozdíl od pevně daného unárního kódu parametrizovatelný parametrem $M = 2^C$ (M je mocninou 2). Metodu je tedy možné lépe adaptovat na data. V případě $M = 1$ je kód shodný s unárním kódem. Parametr M nemusí být po celou dobu kódování konstantní.

Pro získání kódu čísla N se nejprve určí hodnoty Q, R a C podle (3.6). Hodnota Q se pak kóduje unárně a hodnota R klasicky binárně na C bitech.

$$Q = \lfloor N/M \rfloor \quad R = N - Q \cdot M \quad C = \lceil \log_2 M \rceil \tag{3.6}$$

N	Q	R	kód
0	0	0	1 00
1	0	1	1 01
2	0	2	1 10
3	0	3	1 11
4	1	0	01 00
5	1	1	01 01
6	1	2	01 10

N	Q	R	kód
7	1	3	01 11
8	2	0	001 00
9	2	1	001 01
10	2	2	001 10
11	2	3	001 11
12	3	0	0001 00
...	...	...	...

Tabulka 3.2: Riceův kód čísel 0–12 pro parametr $M = 4$.

3.2.3 Shannonovo-Fanovo kódování

Shannonovo-Fanovo kódování odvozuje kódová slova přímo od pravděpodobností výskytu kódovaných symbolů (adaptuje se na data). Nejlepších výsledku dosahuje, když jsou pravděpodobnosti výskytu kódovaných symbolů záporné mocniny 2. Huffmanovo kódování však v praxi generuje o něco lepší kód. K vytvoření kódu symbolů je použit binární strom. Symboly jsou listy stromu, jehož hrany reprezentují kód symbolu (např. levá hrana znamená bit 0, pravá bit 1). Konstrukce stromu vypadá následovně.

1. Seřadit symboly sestupně podle jejich pravděpodobností.
2. Rozdělit tuto množinu (rodičovský uzel) na dvě podmnožiny (synovské uzly) tak, že obě budou mít nejlépe stejný součet pravděpodobností.
3. Rekurzivně aplikovat krok 2. na obě podmnožiny až do rozkladu na jednotlivé symboly.

3.2.4 Huffmanovo kódování

Huffmanovo kódování je velmi populární metoda použitá např. v metodách bzip2, Deflate, JPEG nebo MP3. Opět se adaptuje na kódovaná data a nejlepších výsledků dosahuje pro pravděpodobnosti, které jsou rovny záporné mocnině 2. Pro vytvoření kódů využívá také binárního stromu. Symboly jsou opět listy stromu s hranami, které udávají jejich kód. Konstrukce stromu se však poněkud liší.

1. Seřadit symboly sestupně dle jejich pravděpodobností.

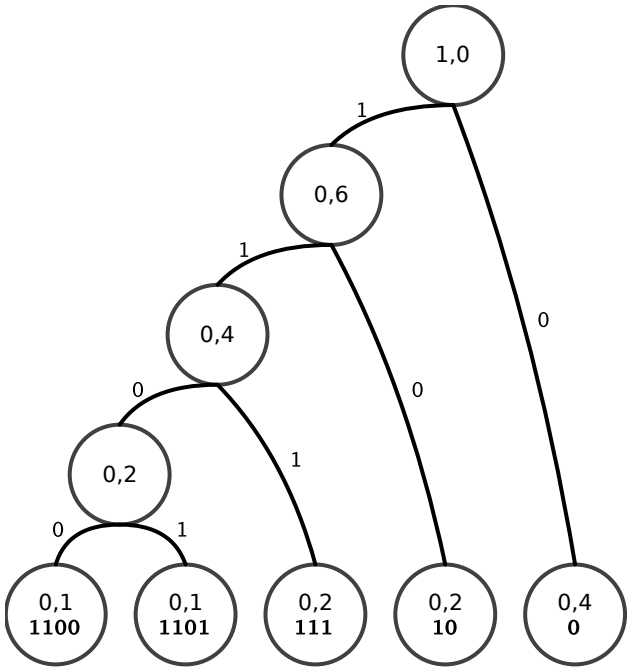
2. Vybrat 2 symboly s nejnižší pravděpodobností, spojit je do nového uzlu.

3. Dokud je co spojovat, pokračuje se opět krokem 2.

◁ Huffmanův strom

Vytvořený strom se nazývá Huffmanův strom. Při této konstrukci má kodér jistou volnost ve výběru symbolů, které bude dále spojovat. Kanonický Huffmanův kód je Huffmanův kód sestavený podle daných deterministických pravidel. Je vhodný pro přenos k dekompresoru (stačí přenést délky kódových slov). U adaptivní varianty (změna pravděpodobností výskytu symbolů v průběhu komprese) je nutné při kompresi i dekompresi opravovat strom, aby stále vyhovoval výše uvedené konstrukci. K tomu se používají hlavně dva algoritmy označované podle počátečních písmen jmen jejich tvůrců – algoritmus FGK (Faller, Gallager, Knuth) a algoritmus V (Vitter).

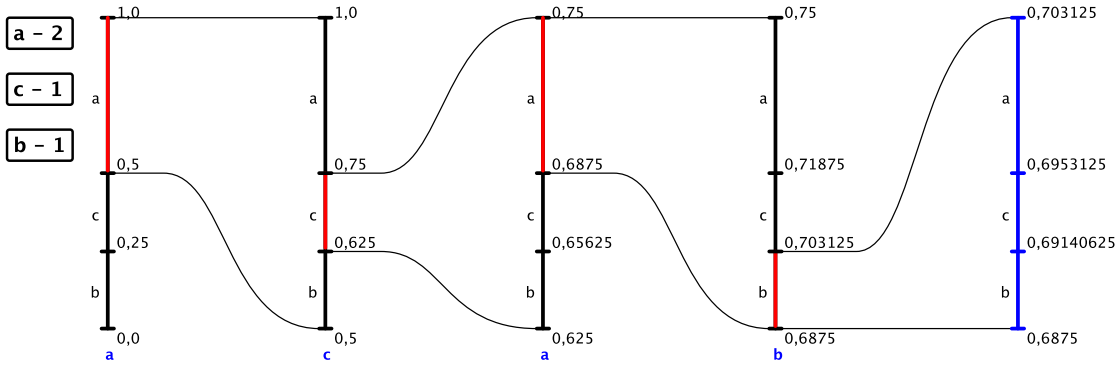
Jednoduchý příklad Huffmanova stromu sestaveného pro pět symbolů s pravděpodobnostmi 0,1, 0,1, 0,2, 0,2 a 0,4 je uveden na obr. 3.5 (symboly nejsou pro jednoduchost uvedeny, pouze jejich pravděpodobnosti). Symbol s nejvyšší četností (a odtud pravděpodobností) výskytu obdržel kód o délce 1 bit. Dva symboly s nějnižší četností výskytu obdržely naopak kódy o délce 4 bity. Vytvořený kód ale není pro toto rozložení pravděpodobností optimální, protože pravděpodobnosti nejsou mocninami dvou (jako např. 0,5 nebo 0,25). Pro optimální kód (délka kódu odpovídá entropii symbolu) je třeba použít aritmetické kódování.



Obrázek 3.5: Huffmanův strom pro 5 symbolů s pravděpodobnostmi 0,1, 0,1, 0,2, 0,2 a 0,4.

3.2.5 Aritmetické kódování

Na rozdíl od předchozích kódů vytváří aritmetické kódování optimální kódy pro libovolné pravděpodobnosti výskytu kódovaných symbolů. Metoda přidělí jeden kód celému kódovanému souboru dat, nikoli jednotlivým symbolům. Klíčovou myšlenkou

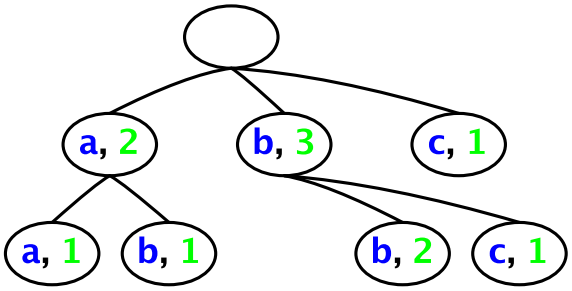


Obrázek 3.6: Aritmetické kódování řetězce `acab`. Začíná se s intervalem $\langle 0, 1)$, výsledek je libovolné číslo z intervalu $\langle 0{,}6875, 0{,}703125)$.

metody je zužování intervalu úměrně pravděpodobnosti výskytu právě kódovaného symbolu. Zužování intervalu vyžaduje další bity, takže délka kódu postupně roste. Začíná se s intervalem, který se podle pravděpodobností kódovaných symbolů neustále zužuje, a výsledkem kódování je libovolné číslo z výsledného intervalu. Ke kompresi dochází, protože symbol s vyšší pravděpodobností zúží interval méně (přidá méně bitů) než symbol s pravděpodobností nižší. Princip se běžně vysvětluje na reálném intervalu $\langle 0, 1)$, praktické implementace však musejí pracovat s celými čísly (např. 32bitový `int`). Metoda může být použita např. v kontextových kodérech, formátech JPEG[1] nebo Dirac. Implementace adaptivní varianty je značně jednodušší než v případě Huffmanova kódování (oprava Huffmanova stromu). Postačí pouhá aktualizace modelu dat, což může znamenat jen inkrementace četnosti symbolu v poli a případně inkrementaci kumulovaných četností všech následujících symbolů.

3.2.6 Kontextová komprese

Kontextové kodéry využívají nejčastěji aritmetické kódování nebo jeho modifikaci, případně Golombovo-Riceovo kódování pro rychlou implementaci. Na rozdíl od něj nekódují pravděpodobnost výskytu osamoceného symbolu, ale pravděpodobnost jeho výskytu v určitém kontextu. Kontext je zde nejčastěji několik předcházejících symbolů, okolních pixelů, okolních bitů, sousedních koeficientů apod. Podle počtu symbolů použitých k predikci se hovoří o řádu kon-

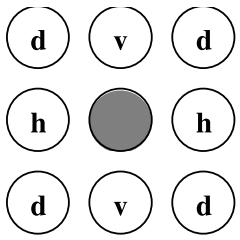


Obrázek 3.7: Kontextový model pro řetězec `aabbbc`. Byl použit kontext řádu 1 (jeden předcházející znak).

[1] běžně se nepoužívá

textu.[2] Kontext musejí tvořit již přenesené symboly, protože jinak by dekodér nemohl symbol predikovat. Kontext je tedy využit k predikci, což je zde přidělení pravděpodobnosti výskytu každému symbolu. Tato forma entropické kompresní metody je v různých variantách použita např. ve standardu MPEG-4 (CABAC, CAVLC), JPEG 2000 (EBCOT), JPEG-LS nebo metodách PPMx.

Na obr. 3.7 je zobrazen model kontextu 1. řádu pro jednoduchý řetězec `aabbbc`. Kontextem je zde jeden bezprostředně předcházející znak. Tento model predikuje např. po symbolu 'b' symbol 'b' s pravděpodobností 66 % a symbol 'c' s pravděpodobností 33 %. Na obr. 3.8 je zobrazen kontext kodéru EBCOT ze standardu JPEG 2000. Symboly h, v a d (horizontální, vertikální a diagonální sousedi) označují bity koeficientů v právě kódované bitové rovině podpásma DWT (diskrétní vlnkové transformace). Kontextem je zde tedy několik bitů, které mají v doméně DWT jistou sousednost s právě kódovaným bitem.



Obrázek 3.8: Kontext kodéru EBCOT, který je použit ve standardu JPEG 2000.

3.3 Slovníkové metody

Slovníkové metody využívají ke kódovaní částí vstupního toku dat slovník. Slovník může být jednoduchý buffer několika předchozích symbolů nebo sofistikovaná datová struktura, která šetří paměť a umožňuje rychlé vyhledávání fragmentů nekomprimovaných dat. Výstupem slovníkových metod jsou značky odkazující od tohoto slovníku.

3.3.1 LZ77

Metodu LZ77 publikovali v roce 1977 izraelští vědci Abraham Lempel a Jacob Ziv. Má mnoho modifikací a je použita např. jako základ algoritmu Deflate. Jádrem metody je pohyblivé okno (sliding window), do kterého tečou zprava doleva nekomprimovaná data. Kodér vstupující data analyzuje a na výstup produkuje kódová slova neboli značky. Dekodér načítá vyprodukované značky a na jejich základě rekonstruuje původní datový tok, který pak z pohyblivého okna vytéká. Pohyblivé okno je rozděleno na vyhledávací a předvídací část čili buffer. Vyhledávací buffer slouží jako slovník, proto se metoda řadí mezi slovníkové kompresní metody. Vyhledávací buffer je typicky dlouhý desítky kilobajtů, předvídací naproti tomu desítky bajtů. Kompresor se snaží nalézt obsah předvídacího bufferu v bufferu vyhledávacím. Na výstup pak produkuje značky s informací o místě a délce nalezené shody. Čím delší shodu se podaří nalézt, tím větší

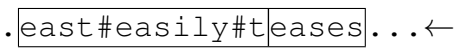
◁ pohyblivé okno

◁ vyhledávací, předvídací buffer

[2] analogicky k řádu prediktoru

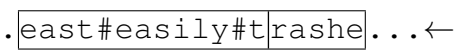
komprese je vytvořením značky dosaženo. Značku konkrétně tvoří prvky `(offset, délka, symbol)`. Prvek `offset` je pozice ve vyhledávacím bufferu, na které byl nalezen nekomprimovaný řetězec shodný s fragmentem na počátku předvídacího bufferu. Prvek `délka` je délka tohoto řetězce a `symbol` je následující symbol, který byl už neshodný.

V následujícím případě jsou nalezeny 2 shody „eas“ na pozicích 8 a 13. Pro jednoduchou implementaci se použije poslední shoda. Vytvoří se tedy značka `(13, 3, 'e')`.

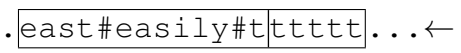


Prvky značky se zakódují na odpovídajícím počtu bitů $\lceil \log_2 S \rceil$, $\lceil \log_2 (L-1) \rceil$, $\lceil \log_2 A \rceil$, kde A je velikost abecedy.

Při nenalezení shody se generuje značka ve tvaru `(0, 0, symbol)`.



Shoda může překročit hranici vyhledávacího bufferu. V následujícím případě bude vytvořena značka `(1, 5, 't')`.



Poslední symbol předvídacího bufferu musí být vždy zakódován jako prvek značky symbol, což je také důvod pro počet bitů $\lceil \log_2 (L-1) \rceil$ prvku značky `délka`. LZ77 předpokládá, že fragmenty se vyskytují blízko u sebe, tj. že nebyla ještě vysunuta z vyhledávacího bufferu. Existuje množství vylepšení této metody, např. značky proměnné délky, důmyslné datové struktury nebo zrušení posledního pole značky.

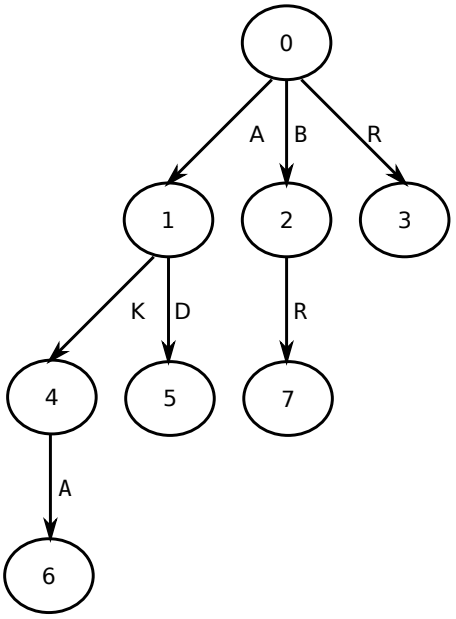
3.3.2 LZ78

Metodu LZ78 publikovali A. Lempel a J. Ziv v roce 1978. Opět má mnoho modifikací, např. algoritmus LZW. Na rozdíl od LZ77 nemá pohyblivé okno, ale opravdový slovník, ve kterém jsou fragmenty nekomprimovaného souboru identifikovány pomocí indexu neboli ukazatele do slovníku. Metoda je v praxi náročná na paměť, což lze řešit různě, např. vhodnou datovou strukturou k udržování slovníku. Tato vhodná struktura může být trie (forma datové struktury strom). V tomto případě je první položka slovníku prázdný řetězec (kořen stromu). Kodér vytváří značky ve tvaru `(index, symbol)`. Při kompresi se vyhledá nejdelší shodný řetězec obsažený ve slovníku a vygeneruje se značka s jeho indexem a následujícím symbolem, který způsobil neshodu. Každá značka přitom udává nový řetězec, který je třeba umístit do slovníku. Dekodér načítá značky, rekonstruuje nekomprimovaný soubor a přitom si buduje slovník stejně jako kodér.

◁ trie

Následuje příklad zakódování řetězce ABRAKADAKABRA. Odpovídající slovník tvořený datovou strukturou trie je vyobrazen na obr. 3.9. Výstupem komprese je 7 značek. Symbol EOF signalizuje dekodéru konec dat. Postupem času se do slovníku dostávají delší fragmenty vstupu a roste účinnost komprese.

- Na počátku je prázdný slovník (pouze kořen).
- Řetězce 'A', 'B', 'R' se zakódují jako značky `(0, 'A')`, `(0, 'B')`, `(0, 'R')`.
- Dále 'AK' a 'AD' se přidají pod uzel 'A' a zakódují se jako `(1, 'K')`, `(1, 'D')`.
- Stejně se zakóduje 'AKA', 'BR' a poslední 'A' jako `(4, 'A')`, `(2, 'R')`, `(1, EOF)`.



Obrázek 3.9: Trie pro řetězec ABRAKADAKABRA.

3.3.3 LZW

LZW je varianta LZ78, kterou vyvinul v roce 1984 americký vědec Terry Welch. Je použita např. ve formátech GIF, TIFF nebo PDF. Stejně jako u LZ78 je zde použit slovník, který je na počátku inicializován na všechny symboly vstupní abecedy. Značka má však již pouze jedno pole `(index)`. Kodér pracuje podle následujícího algoritmu.

1. Ve vstupu hledá nejdelší řetězec I obsažený ve slovníku.
2. Následující symbol x způsobí, že řetězec Ix ve slovníku již není.
3. Na výstup nyní vyšle index řetězce I, uloží řetězec Ix do slovníku, řetězec I dále bude jen symbol x.
4. Pokračuje se krokem 1.

Dekodér pracuje následovně.

1. Načte index a zapíše odpovídající řetězec I na výstup.
2. Je třeba přidat do slovníku řetězec Ix, ale symbol x ještě není znám.

3. Načte se další index, na výstup zapíše odpovídající řetězec J, jeho první znak je symbol x.

4. Nyní může dekodér uložit řetězec Ix do slovníku, řetězec J bude dále řetězec I.

5. Pokračuje se bodem 2.

Příklad

Bude se kódovat řetězec `sir#sid#`. Slovník budují kodér i dekodér ve shodě. Dekodér je však o jeden krok pozadu.

index	řetězec
0–255	znaky 0–255
256	'si'
257	'ir'
258	'r#'
259	'#s'
260	'sid'
261	'd#'

Tabulka 3.3: Slovník. Indexy 0–255 jsou zaplněny při jeho inicializaci. Další řetězce postupně přibývají v průběhu (de)komprese.

vstup x	vyhledání Ix	nalezeno	výstup (index) I	uložení Ix	nový I
's'	's'	ano			Ix='s'
'i'	'si'	ne	's'	'si'	x='i'
'r'	'ir'	ne	'i'	'ir'	x='r'
'#'	'r#'	ne	'r'	'r#'	x='#'
's'	'#s'	ne	'#'	'#s'	x='s'
'i'	'si'	ano			Ix='si'
'd'	'sid'	ne	'si'	'sid'	x='d'
'#'	'd#'	ne	'd'	'd#'	x='#'
EOF	'#EOF'	ne	'#'		

Tabulka 3.4: Kodér. Vstupem jsou jednotlivé znaky, výstupem index řetězců ve slovníku.

3.3.4 Deflate

Metoda Deflate je použita ve formátech ZIP (původně), ZLIB, GZIP, 7-Zip, PNG, MNG nebo PDF. Je to kombinace LZ77 a Huffmanova kódování. Na rozdíl od LZ77 mají značky jen dvě pole `(offset, délka)`. Scházející položka `(symbol)` se zapisuje do výstupního toku separátně. Komprimovaný tok se tedy skládá ze tří entit: symbolů neboli literálů, offsetů neboli vzdáleností shody a délek shody. Tyto entity se kódují pomocí Huffmanových kódu za pomoci dvou tabulek (literály/délky a

vstup (index) J	výstup J	x=J(1)	uložení Ix	nový I
's'	's'	's'	's'	J='s'
'i'	'i'	'i'	'si'	J='i'
'r'	'r'	'r'	'ir'	J='r'
'#'	'#'	'#'	'r#'	J='#'
'si'	'si'	's'	'#s'	J='si'
'd'	'd'	'd'	'sid'	J='d'
'#'	'#'	'#'	'd#'	J='#'

Tabulka 3.5: Dekodér. Vstupem jsou index řetězců, výstupem tyto řetězce.

vzdálenosti). Velikost vyhledávacího bufferu je až 32 kB. Data se komprimují nezávisle po blocích různé délky. Dekodér musí být schopen dekomprimovat bloky libovolné délky. Metoda definuje 3 režimy komprese:

1. bez komprese (blok může mít maximálně 65 535 B, užitečné pro nekomprimovatelná data a pro segmentaci souborů),

2. komprese s fixními tabulkami kódů (urychlí zpracování, sníží kompresní poměr, tabulky netřeba zapsat do výstupního toku) a

3. komprese s tabulkami zapsanými do komprimovaného toku (vyšší kompresní poměr, protože tabulky mohou být sestaveny na základě četnosti výskytu symbolů ve vstupním toku, tabulky jsou do výstupu zapsány speciální kompresí).

Přitom každý blok může být komprimován v jiném režimu komprese. Metoda je na implementaci více komplikovaná.

Na rozdíl od původní LZ77 může kompresor odložit výběr shody a zakódovat první znak předvídacího bufferu jako literál.

←...`she#needs#then#there#`|`the#new`...←

V tomto případě Deflate nevybere `(11,3)` = „the“, ale „t“ zakóduje jako literál a vybere `(20,5)` = „he#ne“. Tento odklad shody je při kompresi řízen parametrem – normální režim, režim vysoké komprese, rychlý mód. V rychlém módu je hledání odložené shody zcela vynecháno. V režimu vysoké komprese provede kompresor vždy úplné hledání, což je ale časově náročné. V normálním režimu může kompresor provést redukované hledání (jak moc je hledání redukováno je záležitostí konkrétního kompresoru).

Protože lineární hledání shody v bufferu velikosti až 32 kB je dosti časově náročné, je k hledání této shody ve specifikaci Deflate silně doporučeno použít hashovací funkci. Kompresor, který bude hledat shodu jiným způsobem, však může také produkovat platný formát dat Deflate. Toto hledání řetězců pomocí hashovací funkce je význačným rysem metody Deflate.

Podrobnosti

Každý blok dat začíná jednobitovým indikátorem posledního bloku souboru. Následují dva bity určující režim komprese (1–3, výše). Blok nemusí vždy končit na hranici bajtu. Další blok začíná bezprostředně po konci předchozího bloku. Jeho začátek tedy může být kdekoli uvnitř bajtu.

Blok režimu 1 (bez komprese) obsahuje dále délku dat v bajtech (max. 65 535). Poté následují vlastní data.

V blok režimu 2 (fixní tabulky) následují bezprostředně po tříbitové hlavičce prefixové kódy pro 1. literály/délky a 2. vzdálenosti. Prefixový kód symbolu s číslem 256 značí konec bloku (EOB). V tomto režimu se používají dvě kódové tabulky (literály/délky a vzdálenosti). V první tabulce však nejsou uloženy přímo prefixové kódy, ale jen tzv. edoc kódy. Každý edoc je na prefixový kód teprve konvertován. První tabulka obsahuje edocy 0–255 pro literály, 256 pro EOB a 257–285 pro délky. Z edoců tak lze určit, zdali se jedná o literál (symbol) nebo počátek značky `(délka, offset)`. Těch posledních 29 edoců není pro všech 256 délek (od 3 do 258) dostatek. Takže za některé z těchto edoců je třeba přidat další bity. Do výstupního toku jsou pak zapsány prefixové kódy edoců. Extra bity se přidají za prefixové kódy edoců ve výstupním toku dat. Protože jsou edocy definovány pro délky až do 258, může mít předvídací buffer velikost až do 258 symbolů. Pro zakódování vzdáleností (offsetů) je použita druhá tabulka, ve které je jich ale uvedeno pouze 30. Pro tyto jsou použity speciální 5bitové prefixy. Následují opět extra bity, které umožňují zakódovat celou délku v intervalu 1–32 768. Tudíž maximální velikost vyhledávacího bufferu je až 32 768 symbolů. Když tedy metoda Deflate nalezne shodu `(offset, délka)`, kompresor vyhledá v tabulce literálů/délek kód pro prvek `délka`. Tento kód ale není přímo kódem zapisovaným na výstup. Nazývá se edoc a v následujícím kroku je nahrazen skutečným prefixovým kódem, který bude zapsán do výstupního toku dat a za který mohou být přidány další bity. Kompresor dále vyhledá v tabulce vzdáleností (offsetů) 5bitový prefixový kód pro prvek `offset`. Tento kód bude zapsán do výstupního toku a také zde mohou být přidány další bity.

Blok v režimu 3 (vlastní tabulky kódů) opět využívá dvou tabulek (literály/délky a vzdálenosti). Tyto tabulky prefixových kódů mohou být sestaveny na základě analýzy komprimovaných dat. Není tedy přímo využito krátkých tabulek (edocy) a extra bitů jako v režimu 2, ale naivní kompresor si tímto způsobem může své tabulky prefixových kódů sestavit. Tabulky kódů je samozřejmě třeba zapsat do komprimovaného toku dat. K tomu definuje Deflate speciální postup. V tomto postupu začíná nejprve každá tabulka jako Huffmanův strom. Tato tabulka je dále přeuspořádána do určitého kanonického tvaru, ve kterém může být reprezentována pouze sekvencí délek kódů. Tato sekvence je dále podrobena algoritmu RLE, čímž je zkrácena. Takhle zkrácená sekvence je nyní zakódována dalším Huffmanovým kódováním. Výstupem je jiná tabulka Huffmanových kódů, která je opět přeuspořádána do kanonického formátu, ve kterém může být reprezentována sekvencí délek kódů. Tato další sekvence je permutována a případně zkrácena a nakonec zapsána do výstupního toku. Právě

popsané kódování Huffmanových kódů je podstatnou a význačnou částí algoritmu Deflate.

3.4 Shrnutí

Používané kombinace výše popsaných metod na multimediální data jsou uvedeny níže. Zkratka EC značí entropický kodér. V závorkách jsou uvedeny kompresní formáty, které tento řetězec používají. Vyjma metod, které kódují transformaci, je vstupní obrazová data třeba na počátku linearizovat. U transformačních metod linearizace následuje až po transformaci. Přestože jsou data linearizovaná, predikce může být vypočtena z pixelů sousedných v původním obraze.

- data (nekomprimovaná)
- data → RLE (BMP, TGA)
- data → predikce → kontextový EC (JPEG-LS)
- data → predikce → RLE → EC
- data → predikce → slovníková metoda (PNG)
- data → slovníková metoda (GIF)
- data → transformace → RLE+EC (JPEG)
- data → transformace → kontextový EC (JPEG 2000)

Algoritmy z této kapitoly jsou podrobně probírány v předmětu Kódování a komprese dat (KKO). Ke studiu oblasti komprese dat se dále podívejte na knihu David Salomon. Data Compression: The Complete Reference. 4. ilustrované vydání, Springer, 2006.

Kapitola 4

Formáty obrazu

Tato kapitola popisuje nejznámější formáty a kompresní metody rastrových obrazů. Formáty lze logicky rozdělit na několik skupin. První skupinou jsou ty formáty, které ukládají data zcela bez komprese. Dalšími skupinami jsou formáty, které využívají buďto ztrátovou nebo bezeztrátovou kompresní metodu. Formát nemusí patřit výhradně do jediné skupiny. Například JPEG 2000 smývá rozdíl mezi ztrátovou a bezeztrátovou kompresí.

4.1 Obecné formáty

Tyto formáty nelze jasně zařadit mezi ztrátové nebo bezeztrátové. Používají se buďto zcela bez komprese nebo jako kontejnery pro jiné kompresní formáty.

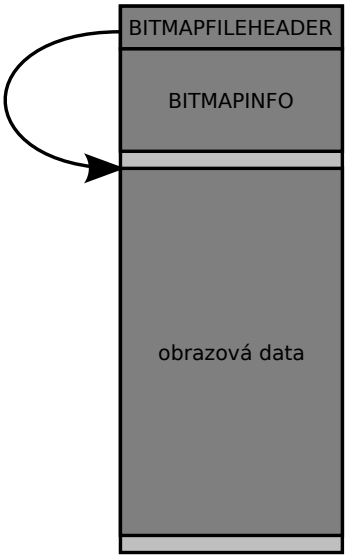
4.1.1 BMP

Souborový formát BMP je obálka pro formát DIB (*device-independent bitmap*) používaný uvnitř systémů Microsoft Windows. DIB se skládá z vlastních obrazových dat a hlavičky, která popisuje jejich formát. Uvnitř formátu je možné uložit buď nekomprimovaný obraz, obraz komprimovaný metodou RLE, případně obraz ve formátu JPEG nebo PNG (pro tiskárny). Mimo formátu DIB je v systému Windows používán také formát DDB (*device-dependent bitmap*), ten však do BMP vložit nelze. Windows API obsahuje množství funkcí pro práci s bitmapami DIB a DDB.

◁ DIB

Formát DIB podporuje barvy pixelů definované 1, 4 nebo 8bitovými indexy do palety barev a dále 16, 24 a 32bitové hodnoty pixelů v barevném modelu RGB. Normálně je obrázek v DIB uložen s řádky uspořádaným zespodu nahoru, tedy vzhůru nohama. Je možné i opačné uspořádání. Jestliže je výška obrázku (`BITMAPINFOHEADER.biHeight`) kladná, jedná se o obrázek uložený vzhůru nohama. V záporném případě je uložen shora dolů. V tomto případě ale není možná jeho komprese. Každý nekomprimovaný řádek je zarovnán na 32 bitů (doplněn nevýznamnými bity). Formát pixelu je vždy BGR, ne RGB.

Soubor s příponou .bmp začíná strukturou BITMAPFILEHEADER. Ta je bezprostředně následována hlavičkou formátu DIB (struktura BITMAPINFO, případně BITMAPV4HEADER nebo BITMAPV5HEADER) a pak vlastními obrazovými daty. BITMAPINFO se skládá ze struktury BITMAPINFOHEADER a barevné palety (pole RGBQUAD). Paleta není vložena ve všech případech, případně může obsahovat bitové masky pro atypické formáty pixelu. Ve struktuře BITMAPINFOHEADER jsou uloženy informace o rozlišení obrázku, formátu pixelu, použité kompresi apod. Vlastní obrazová data nemusejí následovat ihned za strukturou BITMAPINFO (může být vložena výplň). Struktury BITMAPV4HEADER a BITMAPV5HEADER přidávají mj. podporu alfa kanálu (průhlednost), gama korekce, barevných prostorů (např. sRGB) a barevných profilů ICC.



Obrázek 4.1: Vnitřní struktury souboru BMP. Na počátku souboru je vložena struktura BITMAPFILEHEADER, následuje BITMAPINFO. Dále v souboru se nacházejí vlastní obrazová data.

Struktury

```
typedef struct tagBITMAPFILEHEADER {
    WORD  bfType;
    DWORD bfSize;
    WORD  bfReserved1;
    WORD  bfReserved2;
    DWORD bfOffBits;
} BITMAPFILEHEADER, *PBITMAPFILEHEADER;
```

bfType je vždy „BM“.

bfSize je velikost souboru v bajtech.

bfReserved1 a bfReserved2 musejí být 0.

bfOffBits je offset (v bajtech, od začátku souboru), na kterém začínají vlastní obrazová data.

```
typedef struct tagBITMAPINFO {
    BITMAPINFOHEADER bmiHeader;
    RGBQUAD          bmiColors[1];
} BITMAPINFO, *PBITMAPINFO;
```

`bmiHeader` je struktura `BITMAPINFOHEADER` obsahující informace o rozměrech obrazu a jeho barevném formátu.

`bmiColors` obsahuje pole hodnot `RGBQUAD` (barevná paleta), případně pole indexů do současné palety. Počet položek v poli závisí na hodnotách `biBitCount` a `biClrUsed` ve struktuře `BITMAPINFOHEADER`.

```
typedef struct tagBITMAPINFOHEADER {
    DWORD biSize;
    LONG  biWidth;
    LONG  biHeight;
    WORD  biPlanes;
    WORD  biBitCount;
    DWORD biCompression;
    DWORD biSizeImage;
    LONG  biXPelsPerMeter;
    LONG  biYPelsPerMeter;
    DWORD biClrUsed;
    DWORD biClrImportant;
} BITMAPINFOHEADER, *PBITMAPINFOHEADER;
```

`biSize` je velikost struktury samotné.

`biWidth` a `biHeight` jsou rozměry obrázku.

`biPlanes` musí být 1.

`biBitCount` udává počet bitů na pixel. Může nabývat těchto hodnot.

- `0` pro JPEG nebo PNG.
- `1` pro monochromatický obrázek (v paletě jsou dvě barvy).
- `4` pro 16barevný obrázek (v paletě až 16 barev, VGA).
- `8` pro 256barevný obrázek (v paletě až 256 barev).
- `16` pro obrázek mající až 2^{16} barev. Zde je několik možností, vizte dokumentaci. V případě, kdy má `BITMAPINFOHEADER.biCompression` hodnotu `BI_RGB`, jsou pixely ve formátu RGB555. V případě `BI_BITFIELDS` jsou pixely ve formátu specifikovaném bitovými maskami v bmiColors.
- `24` pro max. 2^{24} barev, formát pixelu BGR888 (= BGR24).
- `32` pro až 2^{32} barev. Podle `BITMAPINFOHEADER.biCompression` buď ve formátu RGB24 s nevyužitým MSB nebo podle bitových masek v `bmiColors`.

`biCompression` udává použitou kompresi (pouze u DIB uložených odspodu nahoru, tj. vzhůru nohama). Může nabývat následujících hodnot.

- `BI_RGB` pro nekomprimovaný obraz.
- `BI_RLE8` pro RLE kompresi obrázků s 8 bpp. Více v dokumentaci.
- `BI_RLE4` pro RLE kompresi obrázků se 4 bpp. Více v dokumentaci.

`BI_BITFIELDS` pro nekomprimovaný obraz, formát pixelu udaný bitovými maskami v paletě. Platné pro 16 a 32 bpp.

`BI_JPEG` pro vložený JPEG. Pouze pro tiskárny.

`BI_PNG` pro vložený PNG. Pouze pro tiskárny.

`biSizeImage` udává velikost vlastních obrazových dat. Pro `BI_RGB` může být 0.

`biXPelsPerMeter` a `biYPelsPerMeter` udávají horizontální a vertikální rozlišení cílového (zobrazovacího) zařízení.

`biClrUsed` je počet skutečně použitých barev v paletě (tabulce barev). Více v dokumentaci.

`biClrImportant` je počet barev potřebných k zobrazení obrázku. Nula znamená všechny barvy.

```
typedef struct tagRGBQUAD {
    BYTE rgbBlue;
    BYTE rgbGreen;
    BYTE rgbRed;
    BYTE rgbReserved;
} RGBQUAD;
```

`rgbBlue`, `rgbGreen` a `rgbRed` jsou intenzity modré, zelené a červené.

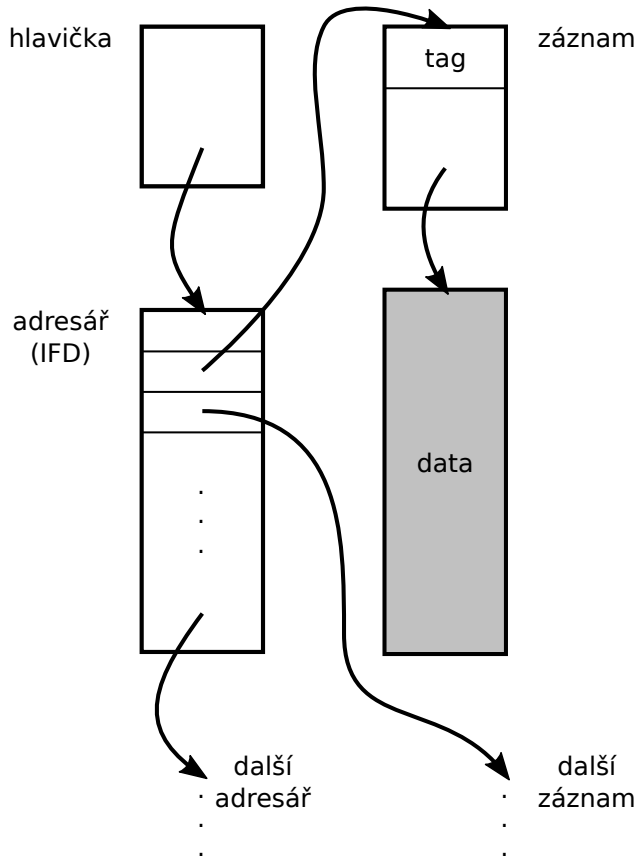
`rgbReserved` musí být 0.

4.1.2 TIFF

TIFF (Tagged Image File Format) je víceúčelový formát původně z roku 1986, poslední verze 6 je z roku 1992. Dnes za ním stojí firma Adobe. Umožňuje uložit více obrázků v jednom souboru (např. skenované stránky), fragmentovat obrazová data na proužky nebo dlaždice, uložit průhlednost a kolorimetrii obrazu (bílý bod, chromatičnost primárních barev, přenosová funkce). Dále umožňuje použít různé kompresní algoritmy (RLE, LZW, JPEG).

Struktura souboru připomíná strukturu souborového systému (složky, soubory). Za krátkou hlavičkou TIFFu následuje první IFD (*image file directory*), který je analogií složky/adresáře. Adresářů může jeden TIFF obsahovat několik. Ty pak odpovídají podsouborům (více obrázků/stran v jednom TIFFu). IFD obsahují jednotlivé záznamy (*entry*), které jsou analogií k souborům. Záznam se skládá z tagu (analogie k názvu souboru), typu prvků, počtu prvků a vlastní hodnoty nebo odkazu na hodnotu (analogie k obsahu souboru). Typem prvku záznamu může být buďto `byte` (8 bitů), `ascii` (znaky), `short` (16 bitů), `long` (32 bitů), `rational` (zlomek, 2× long), `sbyte` (znaménkový), `undefined`, `sshort`, `slong`, `srational`, `float` nebo `double`. Tag je indentifikátor obsahu záznamu. Mezi nejpodstatnější patří `Compression` (např. žádná komprese, RLE s modifikovanými Huffmanovými kódy, PackBits, LZW nebo JPEG), `ImageLength`, `ImageWidth`, `BitsPerSample`, `ColorMap` a `SamplesPerPixel`.

◁ tag



Obrázek 4.2: Struktura formátu TIFF. Jednotlivé oblasti mohou ležet kdekoli uvnitř souboru. Data (šedě) mohou být vlastní obrazová data nebo metadata.

Obrazová data mohou být v souboru fragmentována na proužky (*strips*). Příslušné tagy jsou `RowsPerStrip`, `StripOffsets` a `StripByteCounts`. Mezi typy obrazových dat, která je TIFF schopen pojmout, patří obrázky černobílé (2 úrovně), šedotónové (16 nebo 256 úrovní), paletové (16 nebo 256), RGB (16M), CMYK (2^{32}) nebo YC_bC_r (2^{24}). Další často užívané tagy jsou `Artist` (tvůrce obrázku), `Copyright`, `DateTime`, `ImageDescription`, `Make` (výrobce zařízení), `Model` (model zařízení), `Orientation`, `Software`, `Predictor` (předchozí vzorek, pouze ve spojením s LZW), `ExtraSamples` (alfa kanál).

Jednou z variant komprese používané v TIFFu je algoritmus PackBits, což je varianta RLE. PackBits pracuje na úrovni bajtů. Při dekódování se načte 8bitová znaménková hodnota n. Pokud tato leží mezi 0 a 127, zkopíruje se na výstup dalších $n+1$ bajtů beze změny. Pokud však leží mezi -127 a -1, zkopíruje se další bajt na výstup $(-n+1)$krát. Pro hodnotu -128 se neprovádí žádná operace.

◁ PackBits

4.2 Bezeztrátové formáty

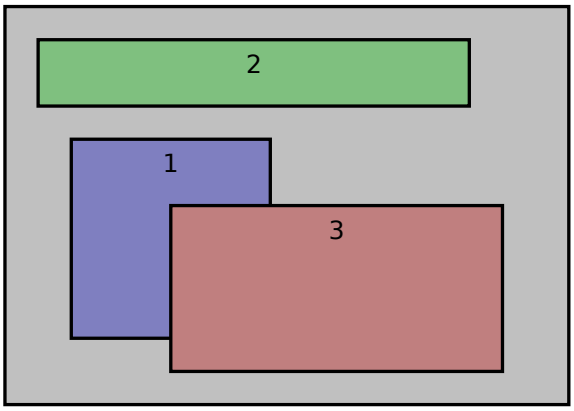
Tyto formáty používají bezeztrátovou nebo téměř bezeztrátovou kompresní metodu.

4.2.1 GIF

GIF (Graphics Interchange Format) vyvinula společnost CompuServe v roce 1987. Původní verze se označuje jako GIF87a, současný standard je GIF89a. GIF dokáže pojmout několik obrazových rámců o barevné hloubce 24 bitů. V jednom rámci však

může být obsaženo maximálně 256 různých barev. Rámce však mohou obsahovat průhledné pixely. Překrýváním rámců přes sebe je tak možno zvětšit celkovou barevnou hloubku až na plných 16 milionů barev ($2^{24} = 16\,777\,216$). Tradované tvrzení, že GIF dokáže pojmout pouze obrázky ve 256 barvách, je tedy nepravdivé nebo přinejmenším nepřesné.[1] Na druhou stranu se s takovými obrázky nelze běžně setkat a to ze dvou důvodů – 1. absence programů, které dokáží takový GIF vytvořit, a 2. často chybná implementace zobrazování těchto souborů.[2] Stejným způsobem lze dosáhnou také animace (rámce se vykreslují přes sebe s časovým zpožděním). Formát GIF využívá slovníkový kompresní algoritmus LZW, který byl dlouhá léta patentován (poslední patent vypršel v roce 2004). Proto také vznikl formát PNG, který není patenty zatížen.

Soubor ve formátu GIF neobsahuje popis obrázku jako celku. Obrázek zde tvoří tzv. logickou obrazovku. Ta je teprve pokryta překrývajícími se rámci, které teprve obsahují obrazová data. Rámce jsou obdélníkové oblasti a nemusejí vyplňovat celou logickou obrazovku. Zbývající část se jednoduše vykreslí barvou pozadí. Příklad je zobrazen na obr. 4.3. Maximální množství rámců není omezeno. Ke každému rámci může být přiřazena lokální barevná paleta. Celá logická obrazovka má pak přiřazenu globální barevnou paletu. Rámec je vykreslován svou lokální barevnou paletou. Pokud tato není přítomna, použije se ta globální. Verze 89a zavádí rozšíření, pomocí kterého lze nastavit dobu zobrazení rámce.



Obrázek 4.3: Logická obrazovka a tři obrazové rámce.

Barevné palety mohou být indexovány maximálně 8 bity ($2^8 = 256$ barev). Každá barva je popsána trojicí bajtů, které udávají pixel ve formátu RGB24. V případě nepřítomnosti lokální i globální palety by měl vykreslující program použít paletu systémovou. Pak je ovšem zobrazení závislé na programu. Barva pozadí je určena z globální palety.

Od verze 89a lze u barevných palet určit index průhledného pixelu. Jedna barva z palety tedy může být označena jako průhledná. Jedná se jen o binární hodnotu, tedy zcela průhledný a zcela neprůhledný pixel. GIF nepodporuje alfa kanál.

[1] hledejte pojem „true color GIF“

[2] programy chybně vkládaji zpoždění mezi rámce s uvedeným nulovým trváním

GIF umožňuje uložit do každého rámce pixel, který je indexem do palety barev. Varianta LZW použitá v GIFu přidává do abecedy tvořené těmito indexy ještě další symboly. Prvním z nich je „clear code“, který přikazuje dekodéru, aby vyprázdnil svůj slovník. Dalším je symbol EOF. Indexy (ukazatele) do slovníku se s jeho vzrůstající velikostí prodlužují. Jsou seskupovány do bloků (max. 255 bajtů) a ukončeny nulovým bajtem. Poslední blok musí obsahovat symbol EOF. Pixely rámce jsou komprimovány postupně po řádcích. Není zde žádná predikce z předchozích pixelů ani žádné využití jejich prostorové sousednosti.

Formát má možnost prokládat obrazová data (progresivní přenos). Prokládání rozdělí obraz na pruhy 8 řádků vysoké. Nejprve se přenese každý osmý řádek z každého pruhu, pak každý čtvrtý, atd. Celkem jsou tak při prokládání 4 průchody. Pořadí znázorňuje tabulka 4.1. Prokládaný soubor GIF je v porovnání s neprokládaným větší. Prokládání může být u každého rámce různé (zapnuto/vypnuto).

řádek sekvenčně	průchod prokládání	řádek prokládaně
1	1	1
2	4	5
3	3	3
4	4	6
5	2	2
6	4	7
7	3	4
8	4	8

Tabulka 4.1: Pořadí komprese řádků u prokládaného GIFu.

Od verze 89a jsou podporovány také animace. Animace je řada po sobě promítaných rámců, které se postupně překreslují. Mezi jejich zobrazením je vložena časová prodleva.

Od verze 89a je též možno podmínit zobrazení následujícího rámce vstupem uživatele (stisk klávesy nebo tlačítka myši). Podpora této funkce v programech je minimální.

Soubor ve formátu GIF je vnitřně rozčleněn do bloků. Některé z bloků jsou povinné, jiné nikoli. Některé z bloků je též možno opakovat (bloky vztažené k rámci). Soubor začíná signaturou (`GIF` v ASCII) a indikací verze, dále následuje popis logické obrazovky (např. rozlišení). Následně může být přítomna globální paleta barev. Dále už následuje popis rámce, lokální paleta a data rámce. Tyto bloky se samozřejmě mohou opakovat. Soubor je ukončen speciálním znakem (`0x3b`). Od verze 89a je možno vložit ještě další rozšiřující bloky. Ty mohou přidávat metainformace, nastavovat třeba dobu zobrazení rámců nebo počet opakování animace.

4.2.2 PNG

Formát PNG (Portable Network Graphics) byl vyvinut v polovině 90. let jako odpověď na patentovaný formát GIF (metoda LZW používaná v GIFu byla patentována americkou společností Unisys). PNG je tedy nezatížený patenty. Formát je rozšiřitelný a podporuje mnoho barevných typů, prokládání, alfa kanál (průhlednost pixelů) a využívá kompresní metodu Deflate (sekce 3.3.4).

Formát sestává z bloků, přičemž každý může mít různý typ i velikost. Některé z nich jsou kritické a každý dekodér je musí být schopen zpracovat. Jiné jsou pomocné (např. metainformace). Blok sestává z následujících částí – velikosti, názvu, vlastních dat a 32bitového CRC. Název bloku má čtyřpísmenný název, z něhož velikost prvního písmene udává důležitost bloku (velké písmeno pro kritické bloky, malé pro pomocné). Podobně velikost druhého písmene udává, zdali se jedná o registrovaný nebo soukromý blok (velké pro registrované skupinou PNG, malé pro privátní). Třetí písmeno je vždy velké. Poslední čtvrté písmeno je velké, jestliže by blok neměl být kopírován, malé jinak. To má smysl při zpracování/úpravě obrázku PNG aplikací, která danému typu bloku nerozumí.

Každá aplikace pracující s PNG jednoduše zpracovává bloky, kterým rozumí. Může bezpečně přeskočit všechny pomocné bloky, které nezná. Jestliže však narazí na kritický blok, který nezná, měla by nahlásit chybu a v dekódování dále nepokračovat.

Čtyři kritické bloky definované standardem PNG jsou `IHDR` (hlavička), `PLTE` (paleta), `IDAT` (obrazová data) a `IEND` (patička). Soubor PNG začíná 8bajtovou signaturou. Bezprostředně za ní následuje blok `IHDR`, který nese informace o rozměrech obrázku, počtu bitových rovin, apod. Pro paletové obrázky musí následovat blok `PLTE`. Následuje jeden nebo více bloku `IDAT` s komprimovanými pixely. Soubor pak končí blokem `IEND`.

PNG může pojmout obrázek v barevném modelu RGB s hloubkou 8 nebo 16 bitů, paletový obrázek s hloubkou 1, 2, 4 nebo 8 bitů, obrázek ve stupních šedi s hloubkou 1, 2, 4, 8 nebo 16 bitů, RGB s alfa kanálem s hloubkou 8 nebo 16 bitů a stupně šedi a alfa kanálem s hloubkou 8 nebo 16 bitů. Dovolené kombinace uvádí tabulka 4.2. Alfa kanál udává ke každému pixelu jeho průhlednost. Je to číslo α v intervalu $\langle 0, 2^{bits} - 1\rangle$ ($bits$ značí bitovou hloubku), které určuje váhu barvy vykreslovaného pixelu P a barvy pozadí B. Pixel je pak vykreslen barvou, která se získá kombinací $(1 - \alpha)B + \alpha P$. U typů pixelu, které neobsahují alfa kanál, je možno specifikovat jeden plně průhledný pixel v pomocném bloku `tRNS`.

Prokládání definované standardem PNG umožňuje dekódovací straně zobrazit obrázek nejprve v hrubém náhledu a pak jej postupně zjemňovat až do plné kvality. Užitá metoda prokládání se nazývá Adam7. Ta rozdělí obraz nejprve do bloků 8×8 pixelů. Každý tento blok je zobrazen v sedmi krocích. V prvním kroku se celý blok vykreslí barvou pixelu v levém horním rohu. V každém dalším kroku je rozlišení bloku zdvojnásobeno. V posledním kroku má každý pixel svou vlastní barvu. Pořadí pixelů je uvedeno v tabulce 4.3. ◁ Adam7

Vlastní komprese obrazu probíhá ve dvou krocích. Prvním krokem je predikce (ve

typ obrazu	kanály	bitové hloubky
paleta	indexy	1, 2, 4, 8
stupně šedi	Y	1, 2, 4, 8, 16
stupně šedi s průhledností	Y, A	8, 16
true color	R, G, B	8, 16
true color s průhledností	R, G, B, A	8, 16

Tabulka 4.2: Typy pixelů, které je formát PNG schopen pojmout. Písmena Y, R, G, B a A značí popořadě jasový, červený, zelený, modrý a alfa kanál. Indexy u palety indexují pixel ve formátu RGB24 (true color) a mohou indexovat také alfa kanál. Bitové hloubky jsou udané na jeden kanál.

1	6	4	6	2	6	4	6
7	7	7	7	7	7	7	7
5	6	5	6	5	6	5	6
7	7	7	7	7	7	7	7
3	6	4	6	3	6	4	6
7	7	7	7	7	7	7	7
5	6	5	6	5	6	5	6
7	7	7	7	7	7	7	7

Tabulka 4.3: Pořadí, ve kterém jsou zpracovávány pixely v prokládání Adam7. V prvním průchodu jsou z každého bloku zpracovány pixely označené jako 1, ve druhém 2, atd.

standardu PNG nazývaná filtrace), která nahradí každý pixel chybou proti predikované hodnotě. Druhý krok je slovníková metoda Deflate, která se aplikuje na tyto chyby predikce.

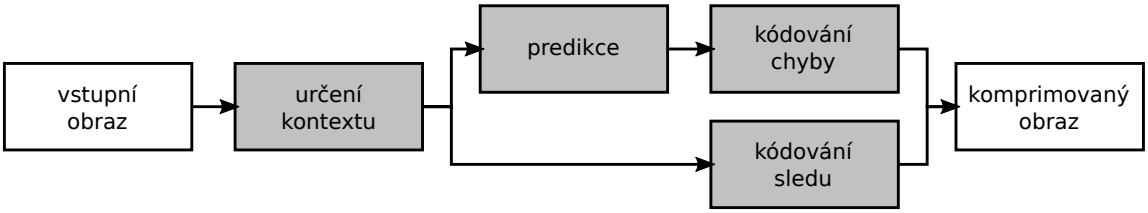
Predikce (filtrace) neprovádí žádnou kompresi dat. Namísto toho nahradí sekvenci vstupních bajtů jinou sekvencí, kterou je Deflate schopen zkomprimovat více v porovnání se sekvencí původní. Predikce je aplikována na řádky, přičemž na každý řádek může být použit odlišný prediktor. To otevírá kompresoru cestu k vyzkoušení všech prediktorů a výběru toho nejlepšího. Predikce se provádí po bajtech, které nemusejí nutně odpovídat pixelům. Například při použití barevného formátu RGB s hloubkou 8 bitů mají pixely formát RGB24. Predikce je pak aplikována separátně na kanál R, G a B, tedy po bajtech. Dále při použití 16bitové hloubky je predikce aplikována zvlášť na horní bajt a zvlášť na dolní bajt 16bitového pixelu. Prediktory definované ve standardu PNG jsou uvedeny v tabulce 4.4. Hodnota chyby predikce e se spočte ze skutečné x a predikované $\hat{x}$ hodnoty jako $e = x - \hat{x}$. Odečtení je počítáno v $\mod 256$, výsledkem je tedy opět jeden bajt.

prediktor	predikovaná hodnota
žádný	0
levý	b
horní	a
průměr	$\lfloor (a+b)/2 \rfloor$
Paeth	rovnice (3.5)

Tabulka 4.4: Prediktory. Pixely odpovídají rozložení na obr. 3.2.

4.2.3 JPEG-LS

JPEG-LS je bezeztrátový formát vytvořený jako náhrada za bezeztrátový režim formátu JPEG, který je neefektivní a většinou neimplementovaný. Nejedná se o modifikaci formátu JPEG, ale o novou metodu. Formálně je tento formát standardizován v ITU-T T.87 a ISO/IEC 14495-1 z roku 1998. Formát umožňuje komprimovat obraz bezeztrátově nebo téměř bezeztrátově.[3] JPEG-LS je založen na algoritmu LOCO-I (LOw COmplexity LOssless COmpression for Images). Tato kompresní metoda se skládá z prediktoru a následného kontextového entropického kodéru (Golombův-Riceův kód). Mimo tento řetězec může kodér zakódovat sled stejných pixelů variantou metody RLE. Zjednodušený postup znázorňuje obr. 4.4. ◁ LOCO-I



Obrázek 4.4: Zjednodušený pohled na schéma komprese. Pro každý pixel je nejprve určen kontext. Pak je rozhodnuto o použití běžného režimu (kódování chyb predikce) nebo režimu kódování sledů.

Komprese probíhá klasicky po řádcích. Pro vícekanálové obrazy (např. R, G, B) podporuje formát několik režimů prokládání – neprokládaný režim (jediný kanál), prokládání několika řádků, prokládání vzorků (složek pixelů). Pro paletové obrázky se použije jednoduše stejný postup jako pro obrázky jednokanálové.

JPEG-LS používá souborový formát založený na souborovém formátu pro metodu JPEG – JPEG Interchange Format (JIF).[4] Komprimovaný tok dat je rozdělen do segmentů. Některé z nich nesou informace potřebné k dekódování (rozměry obrazu), jiné nesou komprimovaná obrazová data.

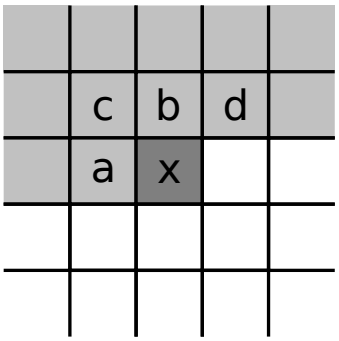
V dalších odstavcích se používá toto značení. Pro vzorek (pixel) x bude jeho hodnota ve vstupním obrázku označena jako Ix, jeho predikovaná hodnota Px a

[3] near-lossless

[4] pro obraz komprimovaný metodou JPEG se samotný JIF běžně nepoužívá

rekonstruovaná (dekomprimovaná) hodnota Rx.

Určení kontextu Nejprve je kolem právě kódovaného pixelu odhadnut gradient $D = (D_1, D_2, D_3)$ pomocí (4.1). Tento gradient zachycuje tvar obrazu okolo aktuálního pixelu x (hladkost, hranu) na obr. 4.5.



Obrázek 4.5: Prediktor MED predikuje hodnotu pixelu x ze tří okolních pixel a, b, c. Gradient D je spočten z hodnot a–d.

$$D_1 = \mathrm{R}d - \mathrm{R}b \tag{4.1a}$$
$$D_2 = \mathrm{R}b - \mathrm{R}c \tag{4.1b}$$
$$D_3 = \mathrm{R}c - \mathrm{R}a \tag{4.1c}$$

Následně se podle tohoto gradientu vybere režim komprese. Pro nulový gradient (nebo nízkou absolutní hodnotu derivací u téměř bezeztrátové komprese) se pokračuje kódováním RLE. V opačném případě se použije běžný režim.

Trojice derivací D_i udává obrovské množství kontextů pixelu x. U obrazu je též vhodné podobné kontexty spojovat. V běžném režimu komprese se proto gradient dále kvantuje na $9^3 = 729$ hodnot. Při tomto kvantování se každá derivace D_i kvantuje do celočíselných hladin Q_i v intervalu $\langle -4, 4\rangle$. Pro každé D_i je to tudíž 9 hladin Q_i, pro všechny tři derivace tak celkem $9^3 = 729$ kvantovaných gradientů. Kvantovaná trojice (Q_1, Q_2, Q_3) je následně spojena s trojicí $(-Q_1, -Q_2, -Q_3)$. Tím došlo ke spojení kvantizovaných gradientů Q s opačnými znaménky. Současně bylo množství různých kvantovaných kontextů Q redukováno na 365. Takové spojení indikováno nastavením proměnné S (`SIGN` ve specifikaci) na hodnotu -1 (jinak $+1$).

V dalším kroku je těchto 365 kvantovaných gradientů namapováno na celá čísla v intervalu $\langle 0, 364\rangle$, celkem tedy na 365 celočíselných hodnot. Toto mapování není ve standardu určeno. Je tedy závislé na implementaci.

Predikce V bežném režimu komprese se hodnoty pixelů převedou na chyby prediktoru. Použitý prediktor detekuje okolo predikovaného pixelu hrany. Jestliže není hrana nalezena, pak se predikovaná hodnota vypočte jako $\mathrm{R}a + \mathrm{R}b - \mathrm{R}c$, což odpovídá proložení roviny bodem x. Taková predikce tedy předpokládá hladkost obrazu v okolí x. Jestliže je však nalezena vertikální hrana, odpovídá predikce hodnotě vzorku Rb na hraně. Stejně tak je predikováno Ra v případě horizontální hrany. Predikovaná hodnota se vypočte jako (4.2). Protože predikce musí být spočtena také dekodérem, počítá se ze rekonstruovaných hodnot pixelů Ra, Rb a Rc.

$$\mathrm{P}x = \begin{cases} \min(\mathrm{R}a, \mathrm{R}b) & : \quad \mathrm{R}c \geq \max(\mathrm{R}a, \mathrm{R}b) \\ \max(\mathrm{R}a, \mathrm{R}b) & : \quad \mathrm{R}c \leq \min(\mathrm{R}a, \mathrm{R}b) \\ \mathrm{R}a + \mathrm{R}b - \mathrm{R}c \end{cases} \tag{4.2}$$

Dalším krokem je korekce této predikované hodnoty. Jejím účelem je vycentrovat rozložení chyb predikce do cílového intervalu. Tento krok je založen na akumulované hodnotě doposud vypočtených chyb predikce pro pixely se stejným kontextem. Tato korekce je tedy mj. závislá na kontextu Q.

Za použití skutečné $\mathrm{I}x$ a predikované $\mathrm{P}x$ hodnoty pixelu se vypočte chyba predikce E. Tento převod (4.3) je také závislý na kontextu Q přes znaménko S.

$$E = S \cdot (\mathrm{I}x - \mathrm{P}x) \tag{4.3}$$

Pro téměř bezeztrátový režim komprese je chyba predikce ještě kvantována. V tomto režimu musí tedy kodér spočíst rekonstruovanou hodnotu pixelu $\mathrm{R}x$, což je normálně činnost dekodéru. To dělá, protože tato hodnota je dále užívána ke kompresi. U bezeztrátového režimu je $\mathrm{R}x$ shodná s $\mathrm{I}x$, takže takový výpočet není třeba.

Kódování chyby predikce Chyby predikce jsou kódovány variantou Golombova-Riceova kódu. Před vlastním kódováním je třeba převést chyby predikce na nezáporná celá čísla.

$$E_M = \begin{cases} 2 \cdot |E| & : \quad E \geq 0 \\ 2 \cdot |E| - 1 & : \quad E < 0 \end{cases} \tag{4.4}$$

Pro téměř bezeztrátový režim je tento převod mírně odlišný. Tyto nezáporné hodnoty jsou dále kódovány Golombovými-Riceovými kódy s limitem maximální délky. Parametr kódu je závislý na kontextu a aktualizuje se vždy, když je v tomto kontextu zakódována nová hodnota.

Režim kódování sledů Jestliže je gradient kontextu nulový (malý u téměř bezeztrátového režimu), použije se režim RLE. Kompresor hledá a počítá až do konce řádku pixely $\mathrm{I}x$ shodné s předchozím zakódovaným pixelem $\mathrm{R}a$. U téměř bezeztrátového režimu se hodnoty pixelů mohou mírně lišit. Když je sled přerušen neshodným pixelem $\mathrm{I}x$, bude tento zakódován běžným režimem komprese (jen s drobnými odlišnostmi). Délky sledů se kódují rozšířenou formou Golombova kódu.

4.3 Ztrátové formáty

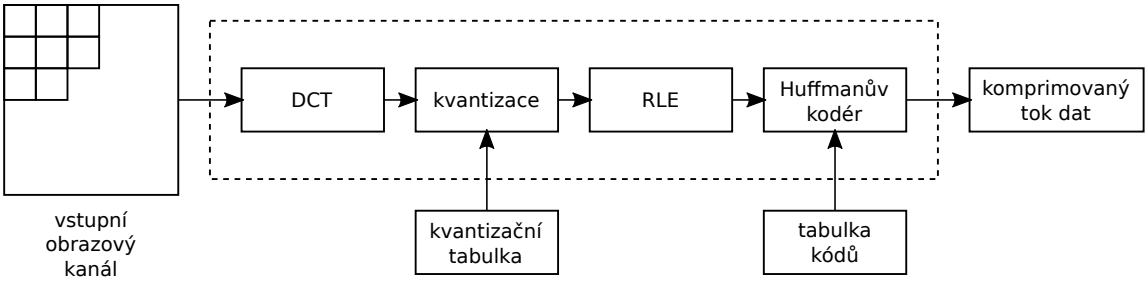
Tyto formáty používají ztrátovou kompresní metodu. Nicméně některé z nich specifikují také bezeztrátovou kompresi.

4.3.1 JPEG

JPEG (Joint Photographic Experts Group) je název výboru, který stojí za standardem pro kompresi obrazu ISO/IEC 10918-1 nebo též doporučením CCITT/ITU-T T.81. Tento standard z roku 1992 se běžně označuje zkratkou JPEG. Stejně tak se nesprávně touto zkratkou označuje formát souboru, který takovýto kompresní formát zaobaluje. JPEG je formát primárně určený ke ztrátové kompresi obrazu. Standard však definuje také režim pro bezeztrátovou kompresi.[5] Pro ztrátovou kompresi podporuje JPEG sekvenční a progresivní přenos obrazových dat. Mimoto JPEG definuje hierarchický režim, kdy je obrázek uložen v několika různých rozlišeních. Ten může komprimovat obrázek ztrátově i bezeztrátově. Standard definuje také formát JIF, do kterého je možné data komprimovaná metodou JPEG uložit. Ten se však sám o sobě nepoužívá. Namísto toho se používají s JIF kompatibilní souborové formáty JFIF a Exif, případně JFIF s vloženými segmenty Exif. Tyto soubory mají příponu `jpeg` nebo `jpg`.

◁ JFIF
◁ Exif

Standard JPEG definuje čtyři kompresní postupy (procesy). Základní postup (baseline process), který musejí být schopny provést všechny aplikace pracující s formátem JPEG, umožňuje komprimovat 8bitová vstupní data (na každou složku). Pracuje s Huffmanovým kódováním a ke kompresi umožňuje použít až 2 tabulky Huffmanových kódů pro koeficienty AC a 2 tabulky pro koeficienty DC. Data jsou komprimována pouze sekvenčně. Tímto způsobem je možno zkomprimovat až 4 složky (např. Y, C_b a C_r). Rozšířený postup (extended process) rozšiřuje možnosti základního také na 12bitová data. Umožňuje využít progresivní zpracování a až 4 tabulky pro koeficienty AC a 4 pro DC. Další je bezeztrátový postup, který se v praxi nepoužívá. A posledním je hierarchický postup, který se taktéž nepoužívá. Následující text se věnuje pouze základnímu postupu (baseline).



Obrázek 4.6: Zjednodušené schéma kodéru JPEG. Stejné schéma se aplikuje na každý obrazový kanál.

Vstupní obraz je zpracováván v barevném modelu YC_bC_r. Jednotlivé složky lze horizontálně i vertikálně podvzorkovat. Barvonosné (chromatické) složky C_b a C_r se typicky podvzorkovávají v jednou nebo obou směrech na polovinu (4:2:2 a 4:2:0). Důležitá podvzorkování znázorňuje obr. 2.1. Tyto tři složky barevného modelu jsou

[5] nepoužívá se

dále zpracovávány odděleně. Převod z modelu 8bitového RGB na 8bitový YC_bC_r je popsán v (4.5).

$$Y = 0{,}299R + 0{,}587G + 0{,}114B \tag{4.5a}$$
$$C_b = -0{,}1687R - 0{,}3313G + 0{,}5B + 128 \tag{4.5b}$$
$$C_r = 0{,}5R - 0{,}4187G - 0{,}0813B + 128 \tag{4.5c}$$

Obrázek 4.7: Blokový efekt při vysokém stupni komprese.

Každá složka je rozsekána na bloky 8×8 vzorků. To je hlavní důvod vzniku tzv. blokového efektu (blokových artefaktů) u obrázků s vysokým stupněm komprese (obr. 4.7). K blokovému efektu přispívá menší mírou také podvzorkování barvonosných složek. Blok 8×8 vzorků je základní datovou jednotkou formátu JPEG. Pokud některá složka obrazu nemá rozměry přesně v násobcích bloků, bude při kompresi na velikost bloku rozšířena. Doporučený způsob rozšíření je duplikace nejbližšího řádku/sloupce. Na každém takovém bloku 8×8 je následně spočtena jeho diskrétní kosinová transformace (DCT).

◁ blokový efekt

$$\lambda_u = \begin{cases} 1/\sqrt{2} & : \quad u = 0 \\ 1 \end{cases} \tag{4.6}$$

$$\left\{ g_{v,u,y,x} = \lambda_v \lambda_u \frac{1}{4} \cos\left[\frac{v\pi}{8}\left(y + \frac{1}{2}\right)\right] \cos\left[\frac{u\pi}{8}\left(x + \frac{1}{2}\right)\right] \right\}_{0 \le v,u < 8} \tag{4.7}$$

$$S_{v,u} = \sum_{y=0}^{7} \sum_{x=0}^{7} s_{y,x}\, g_{v,u,y,x} \tag{4.8}$$

Výsledkem je matice 64 koeficientů. Transformovaný blok má v levém horním rohu (souřadnice $0, 0$) koeficient udávající posun vůči 0 neboli průměr bloku, tedy tzv. stejnosměrnou složku signálu. Odtud se nazývá koeficient DC (analogie ke stejnosměrnému proudu, anglicky *Direct Current*). Tento koeficient se dále kóduje odlišně proti zbytku transformace. Zbylých 63 koeficientů se označuje jako koeficienty AC (analogicky ke střídavému proudu, *Alternating Current*). Udávají váhy, se kterými je v bloku přítomna odpovídající dvourozměrná kosinusoida. Právě popsaný postup je plně invertibilní, nedochází ke ztrátě dat. Nejvíce energie (nejvyšší amplitudy koeficientů) bývá soustředěno kolem nízkých frekvencí (okolí koeficientu DC). Vyšší frekvence se vyskytují především na hranách objektů.

◁ koeficienty AC, DC

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

Tabulka 4.5: Příklad kvantizační tabulky Q pro 8bitovou jasovou složku Y.

17	18	24	47	99	99	99	99
18	21	26	66	99	99	99	99
24	26	56	99	99	99	99	99
47	66	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99

Tabulka 4.6: Příklad kvantizační tabulky Q pro 8bitovou barvonosnou složku.

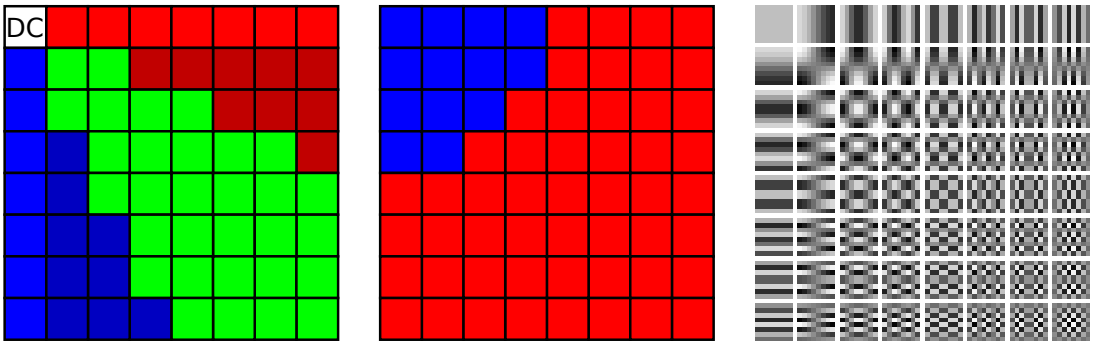
Následujícím krokem je kvantování koeficientů. K tomu musí kodér i dekodér obdržet tzv. kvantizační tabulku Q. To je tabulka s hodnotami, kterými má být při kompresi podělen korespondující koeficient DCT. Následně je každý koeficient zaokrouhlen na nejbližší celé číslo. Příklad takové tabulky je v tab. 4.5. Nejvyšší hodnoty jsou v kvantizačních tabulkách koncentrovány u vysokých frekvencí, kde je možné dopustit se větší ztráty dat.

◁ kvantizační tabulka

Při dekompresi jsou těmito hodnotami koeficienty naopak násobeny. Protože byly hodnoty koeficientů zaokrouhleny na celá čísla, je rekonstruovaný koeficient kvantovaný do několika hladin. Kvantizační tabulka udává, jak velké ztráty dat se kompresor dopustí. Vyšší hodnoty znamenají hrubší kvantování, tedy větší ztrátu informace a tedy méně kvalitní dekomprimovaný obraz. Hodnoty v tabulce jsou tudíž vytvářeny s ohledem na cílovou kvalitu a lidský psychovizuální model. Pro jasovou složku Y a chromatické složky C_b a C_r se používá odlišná tabulka.[6] Důvodem k tomu je větší citlivost lidského oka na jas Y.

Kvantovací tabulky Q jsou parametrem metody JPEG, které udávají míru a kvalitu komprese. Uživatel samozřejmě pro nastavení míry komprese nemusí specifikovat 64 koeficientů každé kvantizační tabulky. Namísto toho je použitým programem spočtena každá z kvantizačních tabulek na základě jediného parametru q, který udává kvalitu obrazu neboli míru komprese. Jestliže je zvolena vhodná kvantovací tabulka, bude

[6] je možné použít odlišnou tabulku pro každý kanál



(a) Směry hran (b) Frekvence (c) Bázové funkce

Obrázek 4.8: Význam koeficientů DCT v bloku 8×8 pixelů. V (a) je naznačena orientace funkce cosinus. Červeně jsou vybarveny vertikální hrany (světle jsou čistě vertikální), modře horizontální (světle čistě horizontální) a zeleně mřížka vytvořená kombinací vertikální a horizontální funkcí. V (b) je zobrazeno rozdělení frekvencí. Modře jsou vyznačeny nízké frekvence, červeně vysoké. Bázové funkce jsou zobrazeny v (c).

většina koeficientů DCT po kvantizaci nulová. Zbylé budou koncentrovány v levém horním rohu (nízké frekvence).

$$S^q_{v,u} = \text{round}(S_{v,u}/Q_{v,u}) \tag{4.9}$$

Matice kvantovaných koeficientů je dále linearizována tzv. zig-zag průchodem. ◁ zig-zag
Tento průchod prochází nejdříve koeficienty s nižšími frekvencemi. Ty mají typicky vyšší amplitudu a tudíž je u nich větší šance, že zůstanou i po kvantizaci nenulové. Naopak koeficienty ke konci průchodu budou s vyšší pravděpodobností nulové a není třeba je kódovat. Namísto toho se za poslední nenulový koeficient umístí symbol EOB (End Of Block). Řada linearizovaných koeficientů je dále podrobena variantě RLE kódování nul. Mimo řetězec RLE stojí pouze koeficient DC, který je kódován diferenciálně (rozdílově) proti koeficientu DC z předchozího bloku. Tento rozdílný krok je proveden, protože průměrné hodnoty (koeficienty DC) sousedních bloků jsou velmi blízké.

$$d = S^{q,n}_{0,0} - S^{q,n-1}_{0,0} \tag{4.10}$$

Posledním krokem jejich komprese je Huffmanovo nebo aritmetické kódování. Obě kódování probíhají za pomoci dodaných tabulek. Aritmetické je cca o 5–10 % účinnější.

Protože aritmetické kódování (QM kodér) se u JPEGu prakticky nepoužívá, věnuje se následující text pouze Huffmanovu kódování. Koeficienty DC se kódují za pomoci odlišných tabulek Huffmanových kódů než koeficienty AC. Sled koeficientů

AC (zlinearizovaných průchodem zig-zag) obsahuje jen několik málo nenulových čísel. Mezery mezi nimi vyplňují sledy nul. Jak už bylo zmíněno výše, za posledním nenulovým koeficientem následuje dlouhý sled koncových nul, který je nahrazen symbolem EOB. Efektivní kódování těchto sledů nul je motivací k využití metody RLE.

Rozdílová hodnota koeficientu DC se kóduje podle tabulky 4.7. Ve skutečnosti není třeba takovou tabulku mít umístěnou v paměti. Kategorii lze pro danou hodnotu koeficientu jednoduše spočítat. Číslo kategorie se zakóduje kódem z použité tabulky Huffmanových kódů. Tento kód je následován stejným počtem bitů, jako je hodnota kategorie. Tyto další bity udávají pořadí hodnoty uvnitř sloupce rozsah zmíněné tabulky. Např. rozdílová hodnota koeficientu -2 bude zakódována Huffmanovým kódem kategorie 2, za kterým následuje binárně `01`, tedy 2. hodnota ve sloupci rozsah. První bit udává vlastně znaménko. Za kategorii 0 se již žádné další bity nepřipojují. Aby se předešlo konfliktu se synchronizačními značkami `0xff`, netvoří žádný z použitých Huffmanových kódů pouze bity 1. Pokud by se bajt `0xff` vyskytnul v zápise hodnoty za kódem kategorie, musí být následován vloženou hodnotou `0x00`. Sestavení tabulky Huffmanových kódů je záležitostí kodéru. Často se však používají předpočítané tabulky, což má výhodu v menší výpočetní náročnosti komprese.

kategorie	rozsah hodnot
0	0
1	$-1, 1$
2	$-3, -2, 2, 3$
3	$-7 \ldots -4, 4 \ldots 7$
4	$-15 \ldots -8, 8 \ldots 15$
5	$-31 \ldots -16, 16 \ldots 31$
6	$-63 \ldots -32, 32 \ldots 63$
7	$-127 \ldots -64, 64 \ldots 127$
8	$-255 \ldots -128, 128 \ldots 255$
9	$-511 \ldots -256, 256 \ldots 511$
10	$-1023 \ldots -512, 512 \ldots 1023$
11	$-2047 \ldots -1024, 1024 \ldots 2047$

Tabulka 4.7: Kategorie a rozsahy koeficientů DC.

Před kódováním každého nenulového koeficientu AC hledá kodér sled Z předcházejících nul. Pokud je nalezen sled délky více než 15 nul, je sled 15 nul zakódován symbolem ZRL. Dalším krokem je výpočet kategorie S pro amplitudu kódovaného nenulového koeficientu. Princip je stejný jako u kategorií koeficientů DC. Kategorie 0 není využita, protože koeficienty jsou již nenulové (nulové se kódují pomocí RLE). Kategorie jsou vypsány v tabulce 4.8. Kategorie S a délka sledu Z se nyní použíjí jako indexy do tabulky Huffmanových kódů. Za kódem opět následují bity, které udávají pořadí ve sloupci rozsah (tedy upřesňují hodnotu koeficientu). Kód pro $Z = 0$ a $S = 0$

má význam EOB. Kód pro $Z = 15$ a $S = 0$ je ZRL, což lze chápat jako sled 15 nul, který je následován koeficientem s amplitudou 0, tedy celkem 16 nul.

kategorie S	rozsah hodnot
1	$-1, 1$
2	$-3, -2, 2, 3$
3	$-7 \ldots -4, 4 \ldots 7$
4	$-15 \ldots -8, 8 \ldots 15$
5	$-31 \ldots -16, 16 \ldots 31$
6	$-63 \ldots -32, 32 \ldots 63$
7	$-127 \ldots -64, 64 \ldots 127$
8	$-255 \ldots -128, 128 \ldots 255$
9	$-511 \ldots -256, 256 \ldots 511$
10	$-1023 \ldots -512, 512 \ldots 1023$

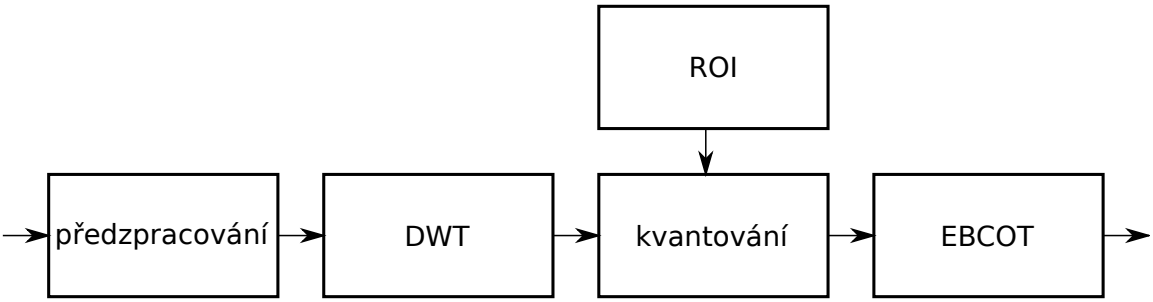
Tabulka 4.8: Kategorie a rozsahy koeficientů AC.

Standard JPEG (T.81) definuje základní kontejner JIF pro data komprimovaná metodou JPEG. JIF využívá uspořádání bajtů big endian. Soubory jsou rozděleny do segmentů, což je blok max. 65 535 B. Každý segment začíná značkou čili markerem, který identifikuje jeho obsah. Marker je dvoubajtová hodnota, ve které první bajt má vždy hodnotu `0xff`. Druhý bajt je vždy v rozsahu `0x01` až `0xfe`. Hodnota `0x00` je vyhrazena pro použití skutečné hodnoty `0xff` (255) uvnitř segmentu. Další hodnoty `0xff` před začátkem markeru jsou ignorovány. Důležité markery jsou `APP0` (hlavička JFIF), `APP1` (Exif), `SOF0` (začátek obrázku v základním režimu), `SOS` (začátek komprimovaných dat), `RSTm` (restartovací značky), `DHT` (definice Huffmanových tabulek) a `DQT` (definice kvantizačních tabulek). Datový tok zkomprimovaný metodou JPEG je však možné vkládat do různých kontejnerů (např. MJPEG, PDF, TIFF). Nejčastěji se vkládá do samostatného souborového formátu JFIF, který rozšiřuje JIF především o hlavičku a náhledy. Tyto soubory pak mají příponu `jpeg` nebo `jpg`. Další možností je použít formát Exif, který je taktéž rozšířením formátu JIF. Exif je značně složitější než JFIF – vkládá do formátu JIF formát TIFF, do kterého ukládá nejrůznější informace o snímku (např. výrobce, model fotoaparátu), náhled obrázku atd. Exif a JFIF nejsou kompatibilní, často se však používá hack, ve kterém se jako kontejner použije JFIF a za hlavičku se vloží ještě segment s daty Exif. Výbor JPEG definoval ještě další formát SPIFF, který se však neujal.

◁ segment
◁ marker

4.3.2 JPEG 2000

Formát JPEG 2000 se měl stát náhradou za JPEG. Byl vytvořen stejným výborem v roce 2000. Jádro formátu (část 1) je standardizováno mezinárodním standardem ISO/IEC 15444-1 neboli doporučením ITU-T T.800. Proti JPEGu přináší řadu vylepšení, vyšší kompresní poměr při stejné kvalitě, vyšší odolnost k chybám atd. Komprese i dekomprese je však výpočetně náročnější. JPEG 2000 smývá rozdíl mezi ztrátovou a bezeztrátovou kompresní metodou. Jádrem tohoto formátu je diskrétní vlnková transformace (DWT). Následující text popisuje pouze jádro tohoto formátu (část 1).



Obrázek 4.9: Schéma komprese metodou JPEG 2000. Předzpracování znamená transformaci barevného modelu obrazu a jeho rozsekání na dlaždice. Následně se na každé dlaždici spočítá diskrétní vlnková transformace (DWT). Koeficienty DWT se pak kvantují a některé z nich mohou být předsunuty před ostatní (ROI). Posledním krokem je kompresní algoritmus EBCOT, který kóduje DWT po bitových rovinách.

Ke ztrátové kompresi používá formát JPEG 2000 stejně jako JPEG modelu $\mathrm{YC_bC_r}$. Převod z RGB lze provést podle (2.2). Tato transformace vstupních složek se nazývá ICT (Irreversible Color Transform). Pro bezeztrátovou kompresi se použije transformace RCT (Reversible Color Transform) (4.11), kterou lze vypočíst aritmetikou celých čísel (integer). Stejně jako u $\mathrm{YC_bC_r}$ se jedná o derivát modelu YUV. Po převodu do $\mathrm{YC_bC_r}$ může následovat podvzorkování barvonosných složek. Složky jsou dále zpracovávány nezávisle.

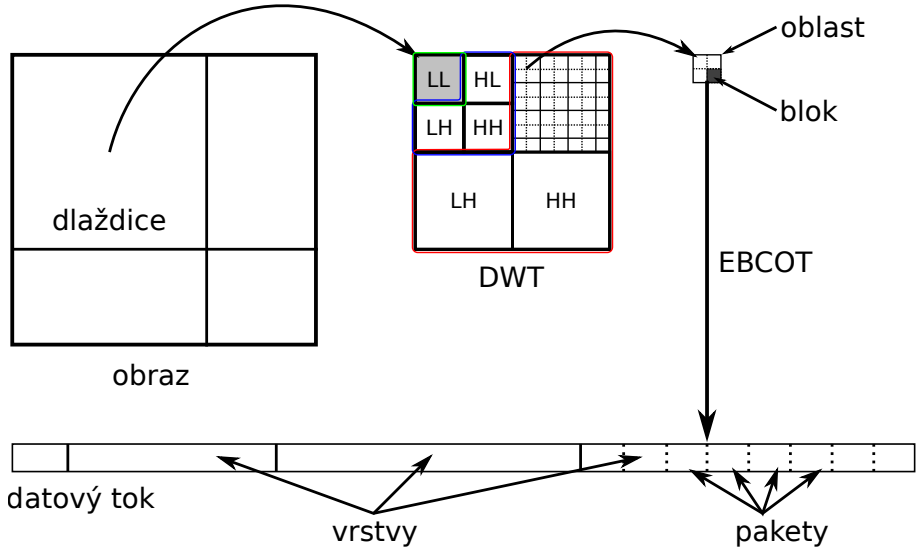
$$Y = \left\lfloor \frac{R + 2G + B}{4} \right\rfloor \tag{4.11a}$$

$$C_b = B - G \tag{4.11b}$$

$$C_r = R - G \tag{4.11c}$$

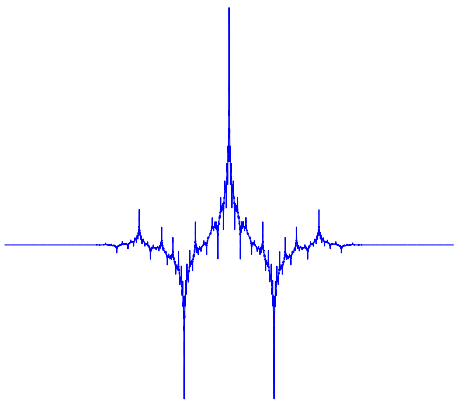
Pro snížení výpočetní náročnosti je možné vstupní obraz rozsekat do dlaždic, které jsou dále zpracovávány nezávisle. Dlaždice jsou obdélníkové oblasti stejných rozměrů. Jejich velikost je definována v segmentu markeru `SIZ`.

◁ dlaždice



Obrázek 4.11: Každá barevná složka je rozsekána na dlaždice. Na každé z nich je spočtena DWT. Jednotlivá rozlišení jsou u DWT znázorněna barevně. Každá úroveň rozkladu obsahuje tři podpásma (HL, LH, HH). Poslední úroveň sestává pouze z podpásma LL. Podpásma jsou rozdělena na oblasti a bloky kódování. Každý blok je po bitových rovinách zkomprimován algoritmem EBCOT. Komprimovaný tok je přeorganizován do vrstev a následně zaobalen do paketů.

Na každé dlaždici je pak spočtena diskrétní vlnková transformace. Výsledkem transformace je několik úrovní rozkladu a tak i několik rozlišení. Každá úroveň (rozlišení) se skládá ze tří podpásem HL, LH a HH. Na nejvyšší úrovni rozkladu (nejmenším rozlišení) se nachází zbývající podpásmo LL. Jednotlivé úrovně rozkladu odpovídají jednotlivým rozlišením, která jsou s každou další úrovní v každém rozměru poloviční. Pro ztrátovou kompresi se používá vlnka CDF 9/7, pro bezeztrátovou CDF 5/3 (lze počítat v celočíselné aritmetice). Další část standardu (část 2) umožňuje použít libovolnou diskrétní vlnku.



Obrázek 4.10: Vlnka CDF 5/3 použitá pro bezeztrátovou kompresi.

◁ rozlišení

◁ kvantování

Kvantování (kvantizace) je proces, při kterém je redukována přesnost koeficientů DWT. Pro bezeztrátovou kompresi je krok kvantování Δ_b roven 1 (žádné kvantování). V případě ztrátové varianty komprese se kvantovací krok spočte pro každé podpásmo b podle (4.12), kde R_b je nominální dynamický rozsah podpásma b (dále), ϵ_b je exponent a μ_b je mantisa. Pár (ϵ_b, μ_b) je určen v segmentech markerů `QCD` nebo `QCC`. Volba těchto hodnot závisí na kodéru. Hodnoty mohou být pro každé podpásmo jiné nebo se určí

jen pro podpásmo LL nejvyšší úrovně rozkladu a ostatní podpásma se odvodí podle jednoduchého vztahu.

$$\Delta_b = 2^{R_b - \epsilon_b}\left(1 + \frac{\mu_b}{2^{11}}\right) \tag{4.12}$$

Bitová hloubka R_I je počet bitů použitý k reprezentaci vzorků původního obrazu (uloženo v markeru `SIZ`). Nominální dynamický rozsah R_b se spočte jako součet rozsahu R_I a logaritmu zisku podpásma (1 pro LH a HL, 2 pro HH, 0 pro LL).

$$R_b = R_I + \log_2(gain_b) \tag{4.13}$$

Oblast zájmu (regions of interest, ROI) je část obrazu, která je komprimována prioritně vůči ostatním méně důležitým částem (pozadí). Při dekompresi je tedy ROI známa dříve než pozadí. To může být užitečné třeba při načítání velkých obrázků po síti. Kompresor může vyznačit jako ROI například všechny obličeje na fotografii. Při dekompresi jsou nejprve dekódovány tváře spolu s hrubým náhledem na pozadí, později je pozadí upřesněno. Pokud je komprimovaný datový tok utnut uprostřed přenosu, jsou tváře dekomprimovány ve výrazně vyšší kvalitě než zbytek obrazu. ◁ ROI

Jádro formátu JPEG 2000 využívá k vyznačení ROI algoritmus Maxshift. Druhá část standardu pak definuje ještě další možnost vyznačení ROI pomocí obdélníků a elips. Principem algoritmu Maxshift je posunout magnitudy koeficientů popředí (ROI) nad koeficienty pozadí o s bitových rovin. To lze interpretovat jako násobení magnitud koeficientů hodnotou 2^s. Dekodéru je známa velikost posunu s a umí tedy rozlišit, které koeficienty jsou posunuty. Kompresní algoritmus EBCOT kóduje koeficienty po bitových rovinách (od MSB po LSB) a zpracuje tedy všechny koeficienty popředí ještě před koeficienty pozadí. Posun s musí být větší nebo roven počtu bitových rovin maximálního koeficientu pozadí. ◁ Maxshift

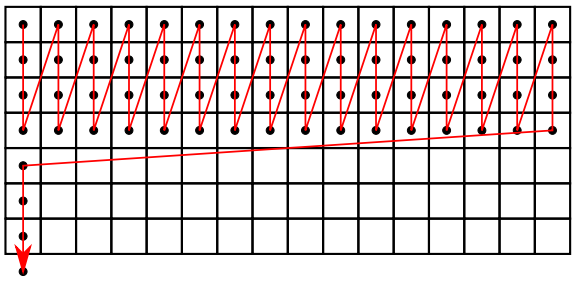
$$s \geq \max(M_b) \tag{4.14}$$

Jednotlivá podpásma jsou dále rozdělěna na obdélnikové oblasti (precinct). Oblasti odpovídají prostorovým částem obrazu a jsou tak využitelné při sekvenčním pořadí progresivního přenosu. Velikostí oblasti je v důsledku omezena velikost paketů (dále). Informace o oblastech jsou uloženy v segmentech `COD` nebo `COC`. Oblasti se dále dělí na bloky kódování.

Pro účely kódování koeficientů algoritmem EBCOT jsou podpásma rozčleněna do obdélníkových oblastí zvaných bloky kódování (*code-blocks*). Velikost těchto bloků je v každém směru mocninou dvou je stejná pro všechny úrovně a podpásma DWT. Pokud blok přesahuje hranice podpásma, kódují se z něj jen ty koeficienty, které do podpásma ještě spadají. Profil 0 omezuje velikost bloku na 32×32 nebo 64×64 koeficientů.

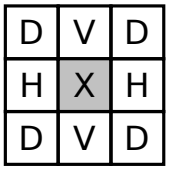
Datový tok (pakety) je rozdělen na tzv. vrstvy (layers). Účelem je postupné zvyšování kvality s každou další dekódovanou vrstvou. Vrstvy se dále skládají z paketů. Každá vrstva obsahuje další bity z bloků kódování. ◁ vrstva

Algoritmus EBCOT kóduje za pomoci aritmetického kodéru (MQ) koeficienty bloků do toku bitů. Kódování probíhá po bitových rovinách od nejvýznamnějšího (MSB) po nejméně významný bit (LSB). Kódování každé bitové roviny probíhá v pořadí ve třech průchodech – propagace významnosti, upřesnění magnitudy a úklid (*cleanup*). Na nejvýznamnější bitové rovině se začíná úklidem (zatím není žádný významný koeficient). Při propagaci významnosti se kódují bity koeficientů, u kterých je velmi pravděpodobné, že se stanou významné (bity s významnými sousedy). Upřesnění magnitudy upřesní bity již dříve významných koeficientů. Při úklidu se zakódují všechny zbývající bity. V rámci bitové roviny se bity zpracovávají po proužcích výšky 4 bity, v rámci proužku pak po sloupcích (obr. 4.12). Bity jsou v jednotlivých průchodech kódovány kontextovým aritmetickým kodérem s kontextem zobrazeným na obr. 4.13. Proud bitů produkovaný tímto algoritmem lze v mnoha bodech přerušit. Dekódovaný obraz pak kvalitativně odpovídá počtu použitých bitů. Pro bezeztrátový režim se zpracují všechny bity. ◁ EBCOT



Obrázek 4.12: Pořadí průchodu bitovou rovinou bloku kódování.

Všechna komprimovaná data obrazu jsou nakonec zformována do paketů. Paket je spojitý segment, který obsahuje data z jedné konkrétní dlaždice, vrstvy (kvalita), barevné složky, rozlišení a oblasti (pozice). Pakety jsou uvozeny svou hlavičkou. Pořadí, v jakém se pakety v datovém toku objevují, může být určeno čtyřmi osami – vrstva (kvalita), barevná složka, rozlišení a oblast (pozice). Toto pořadí může být v průběhu datového toku měněno.



Obrázek 4.13: Kontext používaný algoritmem EBCOT. Bity H, V a D značí bity v bitové rovině koeficientů v daném podpásmu DWT – horizontální, vertikální a diagonální soused.

Datový tok je uložen do nízkoúrovňového formátu, který má shodnou syntaxi s formátem JIF ze sekce 4.3.1. Jsou však použity odlišné markery. Nejvýznamnějšími markery jsou `SOC`, `SOT`, `SOD`, `SIZ`, `COD`, `COC` a `SOP`. Tento datový tok je pak zaobalen do souborového formátu. První část standardu definuje souborový formát JP2 s příponou `.jp2`. Ten umožňuje přiložit metadata ve formě XML (např. XMP).

4.4 Shrnutí

Výše popsaný přehled formátů není úplný. Chybí zde např. RAW, PCX, PSD (Photoshop), JPEG XR, WebP, skupina formátů Netpbm, dále formáty pro HDR apod. Některé formáty jsou standardizovány některou organizací (často ITU-T nebo ISO/IEC), jiné mají alespoň dostupnou specifikaci (dokument). Pouze formát BMP/DIB je oficiálně popsán pouze dokumentací na MSDN. Z formátu TIFF vychází několik dalších formátů, které již standardizovány jsou. Formáty shrnuje tabulka 4.9.

formát	standard	komprese		
		žádná	bezeztrátová	ztrátová
TIFF	ne (jen specifikace)	ano	ano	ano[a]
GIF	ne (jen specifikace)	ne	ano	ne
BMP	ne (jen dokumentace)	ano	ano	ne
JPEG	ITU-T T.81, ISO/IEC 10918-1	ne	ne[b]	ano
PNG	RFC 2083 (verze 1), ISO/IEC 15948	ne	ano	ne
JPEG-LS	ITU-T T.87, ISO/IEC 14495-1	ne	ano	ne[c]
JPEG 2000	ISO/IEC 15444-1, ITU-T T.800	ne	ano	ano
WebP	RFC 6386 (VP8)	ne	ano	ano

Tabulka 4.9: Přehled obrazových formátů. Formát JPEG je ukládán do kontejneru JFIF nebo Exif, formát WebP do RIFF. Formát JPEG definuje také bezeztrátovou kompresi, která je ovšem neefektivní a nepoužívá se.

[a] Ztrátová komprese pouze metodou JPEG.
[b] Definována, ale nepoužívá se.
[c] Formát však umožňuje téměř bezeztrátovou (*near-lossless*) kompresi.

Kapitola 5

Formáty videa

V této kapitole jsou nejprve vysvětleny pojmy a základní principy komprese videa. Výklad se zabývá především hybridními kodéry. Poté jsou podrobněji probrány jednotlivé formáty (např. MPEG).

5.1 Pojmy a metody

Digitální video je možno chápat jako diskrétní signál třech volných proměnných (x, y, t), což jsou prostorové souřadnice a čas. Hodnotou každého vzorku takového signálu je barva. Ta bývá často vyjádřena jako trojice (R, G, B) nebo (Y, C_b, C_r). Z důvodu extrémní paměťové náročnosti[1] nekomprimované formy takového signálu se však s videem pracuje jako se sekvencí dvourozměrných snímků. Snímek (*frame*) je základní jednotkou videa. To znamená, že každý multimediální framework, aplikace či formát pracuje primárně nebo výhradně se snímky, nikoli pixely či naopak většími bloky videa.

◁ snímek

Způsob uložení videa na disk udávají formáty videa. Může se jednat o nekomprimovaný, bezeztrátový nebo ztrátový formát. Formáty mohou být standardizovány některou organizací (často ITU-T a ISO/IEC). Videokodek je software schopný zakódovat a dekódovat datový proud podle některého takového formátu. Tyto datové proudy jsou uloženy v kontejnerových formátech spolu s dalšími informacemi (titulky, kapitoly). Zakódované datové proudy se spolu s audiem a dalšími informace ukládají do multimediálních kontejnerů.

Jako FourCC (*four character code*, 4CC) kód se označuje čtyřbajtový identifikátor kodeku (a tedy formátu), jenž se používá například v kontejnerech AVI nebo Matroska (zde zaobalen). Jeden formát videa může být podle použitého kodeku označen více různými FourCC kódy. Není dokonce výjimkou, že jeden kodek používá pro jediný formát FourCC kódů více. Například různé verze kodeku DivX používají pro formát

◁ FourCC

[1] například 2 hodiny FullHD videa ve formátu RGB24 při 60 fps zabírají teoreticky cca $2{,}687 \times 10^{12}$ bajtů = 2,444 terabajtů

MPEG-4 ASP kódy `DIVX` i `DX50`. Výběr tohoto kódu tedy záleží na použitém kodeku a tudíž i frameworku.

Přenosová rychlost či datový tok (anglicky *bitrate*) udává počet bitů komprimovaného videa, které jsou potřeba k jeho přehrání za jednotku času (např. za sekundu nebo delší segment). Měří se v bitech za sekundu (bps, kbps, Mbps). Pokud je tato rychlost (vztažená na segment videa) po celé trvání video stopy neměnná, hovoří se o konstantní přenosové rychlosti (*constant bitrate*, CBR). V tomto případě se kvalita videa v jeho průběhu mění. Náročnější scény vyžadují vyšší datový tok, který se jim však v případě pevného toku nedostává. U nenáročných scén pak dochází dokonce k plýtvání datovým tokem. Pokud se naopak přenosová rychlost v průběhu videa mění, označuje se jako proměnná přenosová rychlost (*variable bitrate*, VBR). Této strategie se využívá pro zachování fixní kvality (náročnější části videa budou mít vyšší datový tok). Speciální případem VBR je průměrná přenosová rychlost (*average bitrate*, ABR). Ta se snaží dosáhnout dlouhodobě stejného průměrného datového toku. Typicky však vyžaduje víceprůchodovou kompresní metodu. V prvním průchodu je zjištěna náročnost jednotlivých částí videa. V druhém probíhá vlastní komprese, při které je náročnějším částem přidělen vyšší datový tok.

◁ bitrate

◁ CBR, VBR, ABR

Postprocessingem videa se myslí jeho úprava těsně před zobrazením. Často se jedná o úpravu přímo při přehrávání (v reálném čase). Příkladem aplikovaných filtrů mohou být:

◁ postprocessing

- odstranění blokových artefaktů (*deblocking*),
- odstranění prokládání (*deinterlacing*),
- odstranění zvlnění kolem hran (*deringing*),
- odstranění šumu (*denoising*),
- odstranění poblikávání (*deflickering*),
- změna rozlišení (*scaling*) a poměru stran (*aspect ratio*),
- korekce barev (barevné křivky),
- změna formátu pixelů,
- změna snímkové frekvence (*frame rate*),
- přidání efektů starých filmů (zrnění, praskání, šumu),
- vložení loga,
- případně další filtry zlepšující dojem ze sledování videa (doostření, rozmazání, přechodové efekty, zlepšení kontrastu, úprava jasu, apod.)

Je třeba nezaměňovat postprocessing videa s tzv. „loop“ či „in-loop“ filtry. Ty provádějí podobnou filtraci (často konkrétně deblocking). Jsou však nedílnou součástí kompresních a dekompresních algoritmů. Dekodér by bez takového filtru nemohl video dekódovat. Naopak postprocessing je volitelný a probíhá až po dekompresi snímků videa.

◁ loop filtr

Jak bylo uvedeno výše, jakousi základní jednotkou videa je jeden snímek. Rychlost, s jakou se snímky při přehrávání zobrazují, se nazývá snímková rychlost, snímková frekvence (*frame rate*) či pouze FPS (*frames per second*). Jednotkou by měl být Hz (hertz) nebo s^{-1}. V řadě případů se lze setkat s nesprávným značením jednotky jako „fps.“ Často používané hodnoty této veličiny jsou:

◁ snímková rychlost

- 24 (film),
- 25 (televize PAL),
- 60 (HDTV).

Z hlediska komprese videozáznamu se rozlišuje několik typů snímků. Základní je rozlišení na snímky klíčové (*key-frame*, *intra-frame*) a rozdílové (*delta-frame*, *inter-frame*). Klíčové snímky jsou samostatné, nezávislé na okolních snímcích videa. Rozdílové snímky jsou kódovány jako rozdíly vůči jiným snímkům v jejich okolí. K jejich dekódování je tedy třeba dekódovat ještě další snímky, což s sebou nese vyšší paměťové a výpočetní nároky. Snímky, vůči kterým jsou rozdílové snímky kódovány, se nazývají referenční.

◁ klíčové, rozdílové snímky

Pokud je rozdílový snímek kódován vůči referenčnímu snímku, který jej předchází, nepřináší tato závislost při běžném sekvenčním přehrávání další problémy. V moderních kompresních standardech se však jako referenční používají také snímky, které ve videu následují teprve až po odkazujícím rozdílovém snímku. V tomto případě by při sekvenčním zpracování snímků nemohl dekodér rozdílový snímek dekomprimovat, protože se tento snímek v době svého dekódování odkazuje na snímky ještě neznámé (budoucí). Proto se před přenosem k dekódující straně mění pořadí snímků videa tak, že všechny rozdílové snímky následují až po svých závislostech. Této technice se říká změna pořadí snímků (*frame reordering*). Změna pořadí s sebou přináší další paměťové nároky (dekodér je nucen držet v paměti několik snímků, které v daný moment ještě nepotřebuje).

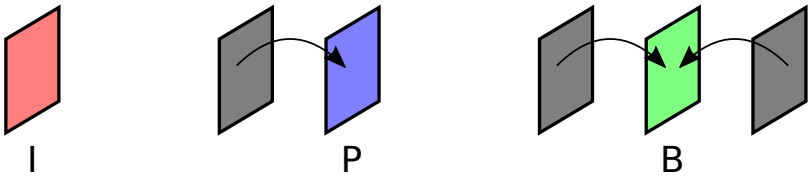
◁ frame reordering

Protože změnou pořadí snímků ztratí dekódující strana informaci o jejich původním pořadí, která je k přehrání nezbytná, jsou snímky opatřeny tzv. časovými razítky. V terminologii standardu MPEG se nazývají PTS (*presentation time stamp*) a DTS (*decoding time stamp*). PTS udává pořadí, v jaké mají být snímky při přehrávání zobrazovány. Naproti tomu DTS říká, v jakém pořadí musejí být dekódovány. Pokud není použito rozdílových snímků, které jako svůj referenční snímek používají snímek z budoucnosti, budou tyto hodnoty shodné.

◁ PTS, DTS

Použití rozdílových snímků přináší ještě jednu nevýhodu. Tou je menší odolnost vůči chybám přenosu. Pokud se při přenosu vyskytne chyba, poškodí odpovídající

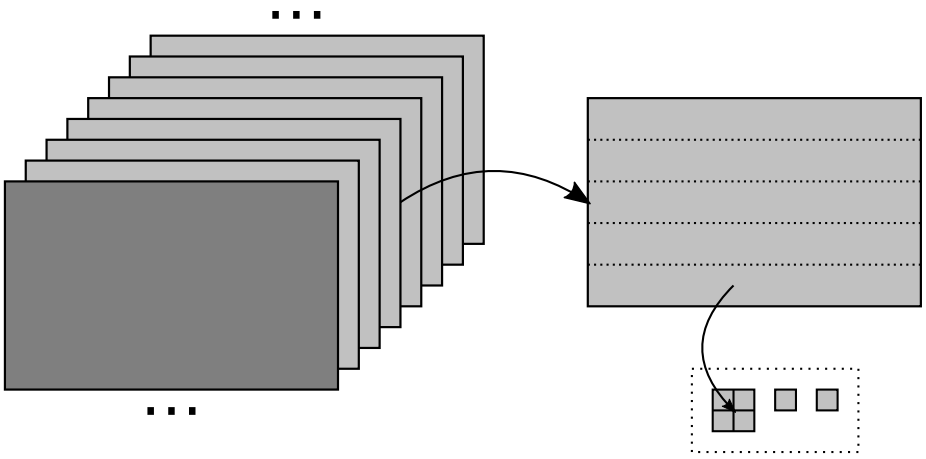
snímek. Při výhradním použití klíčových snímků by se hned následující snímek zobrazil správně. Při použití rozdílových snímků se však chyba přenosu propaguje na další snímky.



Obrázek 5.1: Znázornění závislosti pro I/P/B-snímky. I-snímek není závislý na žádném jiném snímku. Makrobloky P- a B-snímku odkazují na další snímky.

V terminologii rodiny standardů MPEG se klíčové snímky označují jako I-snímky (intra-frame). Rozdílové snímky, které se odkazují pouze na předchozí snímek, jsou zde P-snímky (predicted frame). Nakonec rozdílové snímky, které se odkazují i do budoucnosti, jsou označovány jako B-snímky (bidirectionally predictive frames). V různých standardech existují ještě mnohé další typy snímků. Ve standardu MPEG-1 se objevuje ještě D-snímek (nese pouze DC koeficienty). Rodina standardů VP používá odlišnou terminologii a definuje tzv. golden a altref snímky. Bezeztrátový kodek Lagarith používá nulové snímky (null-frame), které indikují nulovou změnu proti předchozímu snímku.

◁ I/P/B-snímek



Obrázek 5.2: Hierarchie videa. Video je logicky rozděleno na skupiny snímků (tmavě počáteční klíčový snímek). Snímky jsou dále rozřezány na řezy, které obsahují jednotivé makrobloky. Makrobloky se dělí v každém kanálu na bloky tvořené maticí vzorků.

Terminologie ztrátové komprese videa vychází v mnohém především ze standardu H.261. Následující hierarchické dělení videa pochází právě odtud. Celé video se skládá ze sekvence snímků. Ty se seskupují do tzv. skupin snímků (*group of pictures*, GOP). Každá taková skupina začíná klíčovým snímkem. Dále už následují pouze neklíčové. Další klíčový snímek otevírá novou skupinu. Jednotlivé snímky se dále logicky dělí na

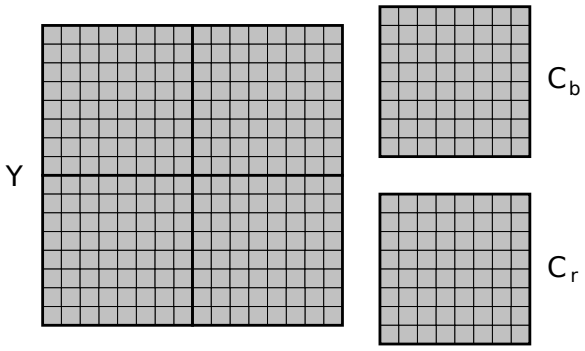
tzv. řezy (*slices*). Tyto řezy jsou kódovány nezávisle. To zvyšuje odolnost datového toku vůči chybám a otevírá cestu k paralelizaci zpracování. Řezy se dále skládají z tzv. makrobloků. Makroblok vzniká rozřezáním jasové složky (komponenta Y) snímku na bloky velikosti 16×16 vzorků. Moderní standardy umožňují použít i jiné velikosti. Součástí makrobloku jsou ale také odpovídající části barvonosných složek (komponent C_b a C_r). Při typicky používaném podvzorkování 4:2:0 mají obě tyto části velikost 8×8 vzorků. Typický makroblok tak tvoří tři části – 16×16 vzorků jasové složky a dvě prostorově odpovídající části 8×8 vzorků barvonosných složek. Makroblok se dále dělí na bloky velikosti 8×8 vzorků. Tyto bloky jsou tentokrát nezávislé na složkách (komponentách) snímku. Zmíněný typický makroblok tedy sestává ze čtveřice (2×2) bloků v jasové složce a dvou dalších bloků v barvonosných složkách. To je celkem 6 bloků na jeden makroblok. Makroblok je jakási elementární jednotka standardů MPEG. Používá se především při popisu P- a B-snímků. Rozdělení na bloky je zde použito pouze za účelem kódování pomocí DCT. Některé standardy (např. H.264) dělí makroblok ještě na další oblasti. Makroblok nebo jeho oblasti tvoří základní jednotku pro popis pohybu mezi snímky (tzv. kompenzace pohybu).

◁ řez

◁ makroblok

◁ blok

V předchozí odstavcích bylo uvedeno dělení snímků na klíčové a rozdílové. Způsob jejich kódování se podstatně liší. Klíčové snímky využívají při kompresi pouze podobnosti v prostorové oblasti snímku. Rozdílové využívají navíc podobnosti v časové oblasti videa. Klíčové snímky jsou řádově větší než rozdílové. Dále bude vysvětleno kódování klíčových a rozdílových snímků.



Obrázek 5.3: Znázornění typického makrobloku (pro podvzorkování 4:2:0). Na kanálu Y se makroblok skládá ze čtveřice bloků o velikostech 8×8 vzorků. Na obou barvonosných kanálech tato čtveřice prostorově odpovídá jen jednomu bloku.

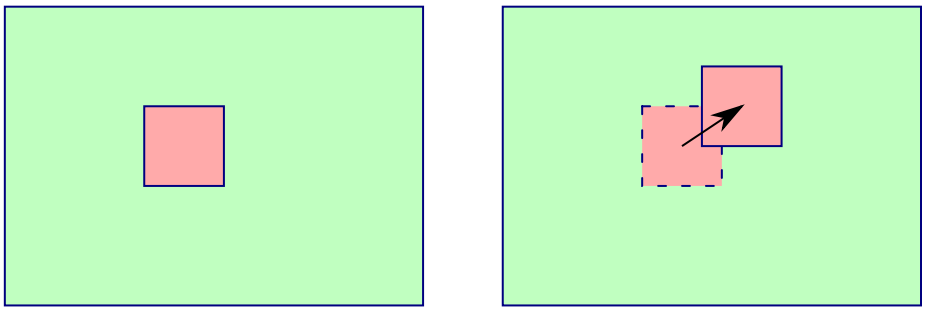
Kódování klíčových snímků neboli intra-kódování pracuje podobně jako komprese obrazu ve standardu JPEG. Snímek je rozdělen na makrobloky a dále na bloky. Ve standardu JPEG by se jednotlivé bloky 8×8 komprimovaly přímo. Ve standardech komprese videa předchází však tento krok ještě predikce makrobloku.[2] Ta probíhá pro celý makroblok z již dekódovaných okolních makrobloků. Podle použitého standardu a velikosti makrobloku může být použito několik režimů predikce makrobloku či jeho oblastí. Typicky se celý makroblok vyplní nejbližším sloupcem makrobloku vlevo, řádkem makrobloku nad, případně kombinací tohoto sloupce a řádku. Dále se v každém bloku kóduje pouze chyba této predikce. Postup je téměř shodný s postupem v metodě JPEG. Prvním krokem je DCT celého bloku. Touto transformací se získá 1 koeficient DC a 63 koeficientů AC. Tyto jsou dále kvantovány

◁ intra-kódování

◁ predikce makrobloku

[2]Do standardů MPEG zavádí intra-predikci až MPEG-4 AVC (H.264).

kvantizační maticí Q. Na rozdíl od metody JPEG jsou však prvky matice ještě před kvantováním násobeny hodnotou q, která se může pro každý makroblok lišit. Ve standardu JPEG byla kvantizační matice pro všechny makrobloky shodná. Matici je samozřejmě v průběhu komprese možné měnit. Následuje linearizace zigzag průchodem, RLE nul a kódování pomocí kódů s proměnnou délkou (VLC), případně varianty aritmetického kódování. Koeficienty DC mohou být v závislosti na standardu zpracovány další transformací nebo zapsány pouze rozdílově jako v metodě JPEG. Pro jasovou a barvonosné složky se používají odlišné tabulky kódů VLC.



Obrázek 5.4: Pohybový vektor pro makroblok v levém snímku odkazuje na část referenčního snímku (vpravo). Pohybový vektor může být udán s přesností na celé pixely, půlpixely nebo čtvrtpixely.

Kódování rozdílových snímků se nazývá inter-kódování. Funguje odliště od výše uvedeného postupu. Přesto uvnitř používá některé části kompresní metody JPEG. Základem popisu rozdílových snímků je popis jejich transformace ze snímků referenčních. Tato transformace se nazývá kompenzace pohybu (*motion compensation*, MC). Jednotlivé kompresní standardy umožňují různé popisy této transformace. Globální kompenzace pohybu (*global motion compensation*, GMC) ji popisuje jako afinní transformaci celého snímku. Podle počtu použitých bodů (*warp-point*) je možné provést složitější transformaci, což je ovšem také výpočetně náročnější. Bloková kompenzace pohybu (*block motion compensation*, BMC) popisuje transformaci pro každý makroblok rozdílového snímku nezávisle. Každý makroblok (případně jeho oblasti) má přiřazen jeden (P-snímek) či dva (B-snímek) pohybové vektory (*motion vector*, MV). Pohybový vektor určuje, kde se v referenčním snímku nachází oblast, která je kódovanému makrobloku rozdílového snímku nejvíce podobná. Při pohybu objektů ve scéně tak pohyblivé vektory sledují pohybující se objekt. Hledání pohybových vektorů se nazývá odhad pohybu (*motion estimation*, ME). Pohybový vektor může být určen s přesností na pixely (celá čísla), poloviny pixelů (half-pel) nebo dokonce čtvrtiny pixelů (Qpel). Jemnější přesnost s sebou nese nutnost interpolace neexistujících vzorků a vyšší výpočetní náročnost. K popisu vektoru je pak také třeba více bitů. Samotné hledání pohybových vektorů je velmi výpočetně náročné. K řešení této úlohy existuje mnoho rychlých algoritmů, které však nemusejí najít nejpodobnější oblast. Odkazovaná oblast v referenčním snímku samozřejmě nemusí odpovídat jeho makroblokům (častý omyl). V některých standardech se používá ještě kompenzace pohybu s prekrývajícími se

◁ inter-kódování

◁ kompenzace pohybu

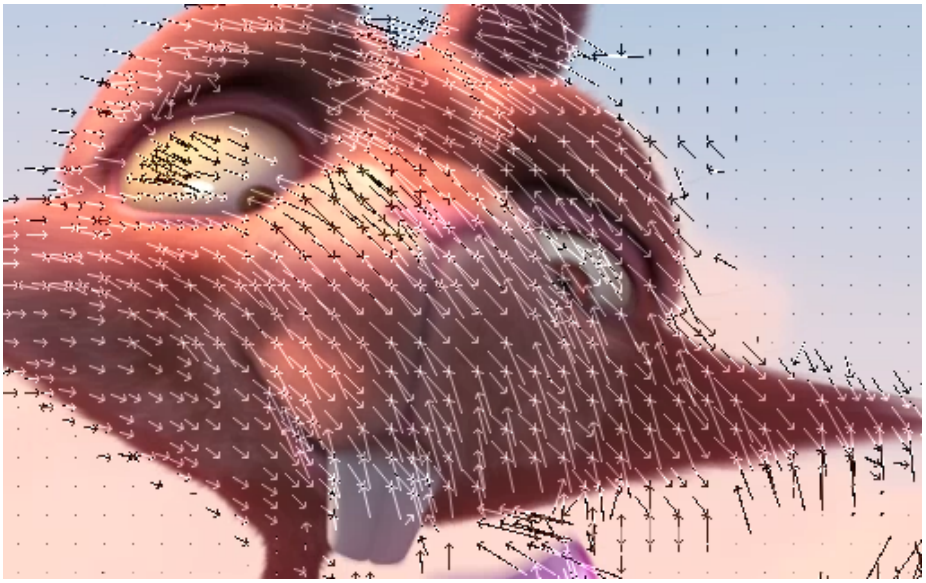
◁ GMC

◁ pohybový vektor

◁ odhad pohybu

bloky (*overlapped block motion compensation*, OBMC). Tato metoda funguje podobně jako BMC. Na rozdíl od ní jsou ale makrobloky virtuálně rozšířeny a každému makrobloku pak náleží více pohybových vektorů. Pohybové vektory jsou kódovány jako rozdíly vůči okolním pohybovým vektorům. Pohybové vektory se udávají pouze na jasové složce. Pro barvonosné složky makrobloku se použijí vektory z odpovídající jasové složky (předpokládají se stejné). Popis makrobloku pouze pomocí kompenzace pohybu není samozřejmě dostatečný. Rozdíl takto predikované hodnoty a skutečné hodnoty kódovaného rozdílového snímku se zakóduje po blocích pomocí DCT. Postup je vzásadě stejný s postupem metody JPEG popsané v odstavci o intra-kódování.

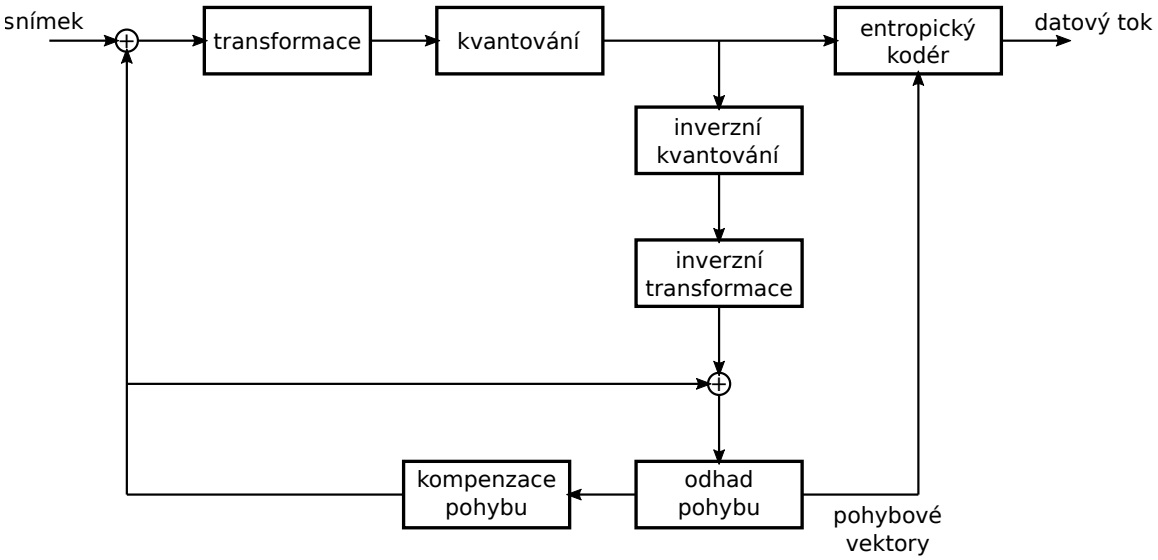
◁ OBMC



Obrázek 5.5: Pohybové vektory zobrazené ve snímku z filmu Big Buck Bunny. Pohybuje se pouze objekt v části snímku (nulové vektory naznačeny tečkou).

Moderní ztrátové kompresní formáty videa kombinují oba výše popsané postupy (intra- a inter-kódování). Využívají tak i prostorovou i časovou redundanci videa. Takovéto kódování se často označuje jako hybridní. Hybridní kodér tedy kóduje některé snímky pomocí intra-kódování (prostorová redundance) a jiné pomocí inter-kódování (časová redundance). Skutečnost je ještě o něco komplikovanější, protože v rámci rozdílových snímků je možné použít na jednotlivé makrobloky také intra-kódování. Takovýto kodér musí tedy predikovat kódovaný rozdílový snímek ze snímků okolních (ne nutně předchozích). Predikovaný snímek se „odečte“ od snímku kódovaného a dále se pracuje pouze s chybou predikce. Do výstupního toku se zakóduje tato chyba (DCT nebo DWT, VLC nebo aritmetické kódování) a pohybové vektory. Kodér i dekodér musejí v paměti udržovat několik snímků, které mohou být použity k predikci (kompenzace pohybu). Tento přístup činí kompresi i dekompresi paměťově náročnými. Počet snímků, které mohou být použity k predikci lze omezit velikostí GOP. Některé formáty (VP8) umožňují predikci z omezeného počtu snímků, čímž podstatně srážejí paměťovou náročnost.

◁ hybridní kódování



Obrázek 5.6: Zjednodušené schéma typického hybridního kodéru. Transformace probíhá na rozdílu vůči predikovanému snímku. Do entropického kodéru vstupuje chyba predikce spolu s příslušnými pohybovými vektory. Kodér dále provádí část činnosti dekodéru – rekonstrukci snímku. Referenční snímky jsou uchovány pro hledání pohybových vektorů při kompresi dalších snímků.

Výše načrtlý postup popisuje všechny kompresní metody vycházející z formátu H.261, tj. především rodinu MPEG. Jednotlivé formáty se liší např. v použité transformaci (DWT je použita ve standardu Dirac), velikosti bloků, typech snímků, použitém entropickém kodéru apod.

◁ prokládání

Při prokládání snímků videa se nepřenáší celý obraz najednou. V jednom okamžiku jsou k dispozici buď pouze sudé nebo pouze liché řádky. Hovoří se tak o půlsnímcích. Televizní systém PAL přenášel například 50 půlsnímků za sekundu. Při zobrazování analogovou televizí takové prokládání nevadilo. Problém nastává u digitálního videa, kde se v jednom okamžiku zobrazuje celý snímek. Ze dvou půlsnímků je tedy třeba složit jeden snímek. Naivní metodou je jejich prokládání. Zde jsou však při pohybu patrné artefakty. K odstranění prokládání existuje mnoho algoritmů. Metody komprese digitálního videa se musejí s prokládáním nějak vypořádat. Prvním standardem, který prokládané video řeší je MPEG-2. Ten zavádí dva druhy snímků. První druh má liché a sudé řádky ve zvláštních snímcích (*field*). Druhý druh má liché i sudé řádky v jednom snímku prokládaně (*frame*). Řádky prokládaných snímků se před zpracováním kompresním postupem reorganizují. Toto přeskládání probíhá na úrovni makrobloku pouze u jasové složky (barvonosné jsou podvzorkovány).

◁ standardizace

Většina kompresních formátů je standardizována některou mezinárodní organizací. Na tomto poli jsou největšími autoritami ISO/IEC, ITU-T (dříve CCITT), případně SMPTE. Standardizuje se pouze syntaxe datového toku (*bitstream*) a funkce dekodéru. Funkce kodéru nejsou standardizovány, kodér musí pouze vytvořit platný datový

tok. To otevírá cestu různým implementacím (rychlost, kompresní poměr). Mnohé standardy specifikují tzv. profily a úrovně (*level*). Jsou to jakési záchytné body, po které je možné jednotlivé standardy implementovat. Profily specifikují použité algoritmy (např. použití B-snímků), úrovně udávají technická omezení (např. maximální rozlišení). Kodér nemusí z použitého profilu použít žádnou vlastnost; nesmí ale použít žádnou vlastnost, která v profilu není. Dekodér naopak musí z použitého profilu implementovat naprosto vše; nemusí ale implementovat žádnou vlastnost, která v profilu není.

◁ profil

5.2 Jednoduché formáty

Tyto formáty komprimují jednotlivé snímky nezávisle. V podstatě se tedy jedná pouze o kompresi obrazu aplikovanou na videosekvenci.

5.2.1 Nekomprimované video

Nejzákladnějším formátem videa jsou nekomprimované (RAW) snímky. Identifikují se FourCC kódem `0x00000000` (nejedná se o text). Formát odpovídá nekomprimované bitmapě DIB (nejčasteji BGR24). Kodek má v obou směrech (komprese i dekomprese) triviální roli – pouhou kopii bloku paměti. Nevýhodou je vysoký datový tok a velikost videa.

5.2.2 Microsoft RLE

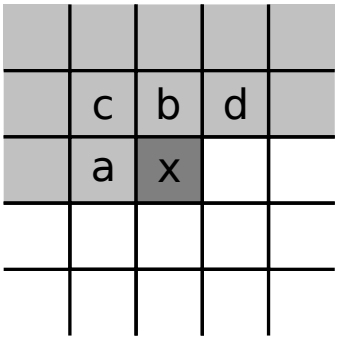
Formát MS RLE odpovídá bitmapě DIB komprimované metodou RLE. Ačkoli MSDN specifikuje tuto kompresi pouze pro snímky bitové hloubky 4 a 8 (tj. paleta), alespoň některé frameworky (FFmpeg) ji implementují i pro vyšší hodnoty (16, 24, 32 bitů na pixel). Vlastní metoda je variantou RLE. Kódová slova tvoří dvojice bajtů A, B. Jejich význam je uveden v tabulce 5.1. Formát identifikuje FourCC kód `mrle`.

A	B	význam
0	0–2	escape kód (konec obrázku)
0	3–255	dalších B bajtů kopírovat
1–255	0–255	dalších A bajtů hodnotu B

Tabulka 5.1: Významy dvojice bajtů u MS RLE. Každý snímek by měl končit escape sekvencí konce obrázku.

5.2.3 Huffyuv

Huffyuv je rychlý bezeztrátový kodek zveřejněný pod licencí GPL. Jeho následníkem je kodek Lagarith. V kompresní metodě lze spatřit klasický postup bezeztrátové komprese obrazu. Prvním krokem je převedení pixelů do barevného modelu $(R-G, G, B-G)$ (též LOCO-I). Druhým krokem je použití některého prediktoru: levý soused a, proložení rovinou p, medián $\mathrm{med}(a, b, p)$. Proložení rovinou se spočte jako $p = a + b - c$. Posledním krokem je statické Huffmanovo kódování. Každý kanál má vlastní Huffmanovu tabulku. FourCC kódem je `HFYU`.



Obrázek 5.7: Okolní pixely a, b, c právě kódovaného pixelu x, které využívají prediktory ve formátu Huffyuv.

Lagarith používá jako prediktor pouze medián. Dále následuje ještě RLE a nakonec aritmetické kódování. Navíc má možnost použít tzv. nulové snímky (*null frames*), které říkají, že současný snímek je shodný s předchozím.

5.2.4 MJPEG

Formát Motion JPEG (MJPEG) nemá jednotný standard. Existuje několik jeho variant v závislosti na použitém kodeku. Zastupuje jej tedy mnoho FourCC. Z toho plyne problém jejich vzájemné nekompatibility. Je typický pro IP a webové kamery. Používá pouze klíčové snímky, které jsou komprimovány metodou JPEG. Nejedná se tedy o hybridní kodér. Existuje také Motion JPEG 2000, který je již specifikován standardem.

5.2.5 DNxHD

Formát DNxHD vyvinutý americkou společností Avid byl v roce 2008 standardizován jako SMPTE VC-3 (formálně SMPTE ST 2019-1). Je určen pro profesionální použití (postprodukce, televizní průmysl) a použití uvnitř kontejneru MXF. Nesetkáte se s ním tedy u koncových uživatelů. Vlastní formát podporuje video ve formátu 4:2:2 (8bitové YCbCr, 10bitové YCbCr) a 4:4:4 (10bitové RGB). Každý snímek je komprimován nezávisle na ostatních (tedy jako klíčový). Kompresní postup je silně podobný metodě JPEG. Vstupní raster je nejprve rozdělen na makrobloky (16×16 na složce Y), ty jsou po komponentách dále rozděleny na bloky 8×8 vzorků. Dalším krokem je DCT každého bloku. Koeficient DC není kvantován a je zapsán rozdílově proti předcházejícímu. Na počátku každého řádku makrobloků je (na rozdíl od JPEG) tato predikce resetována. Na koeficienty AC je dále aplikována kvantizace, která se však od JPEGu liší. Kvantizační tabulky jsou pevně dány standardem. Nicméně každá

kvantizační tabulka je vynásobena součinitelem, který je udán v hlavičce každého makrobloku. Následuje průchod zig-zag, RLE sledů nul a Huffmanovo kódování. Používají se tři tabulky Huffmanových (VLC) kódů: amplitudy, sledy RLE-0, DC koeficienty. Tabulky těchto kódů jsou na rozdíl od JPEGu také pevně specifikovány ve standardu. Formát počítá s uložením progresivního i prokládaného videa (možnost separovat řádky prokládaného videa do separátních bloků). Formát podporuje přidání kontrolního součtu CRC ke každému rámci.

5.2.6 JPEG 2000

JPEG 2000 se používá stejně jako předchozí formát pouze v profesionální sféře (např. distribuce filmů do kin). Každý snímek je kódován nezávisle (klíčový) metodou JPEG 2000. Podrobně vizte odpovídající sekci v předchozí kapitole.

5.3 Hybridní kodéry

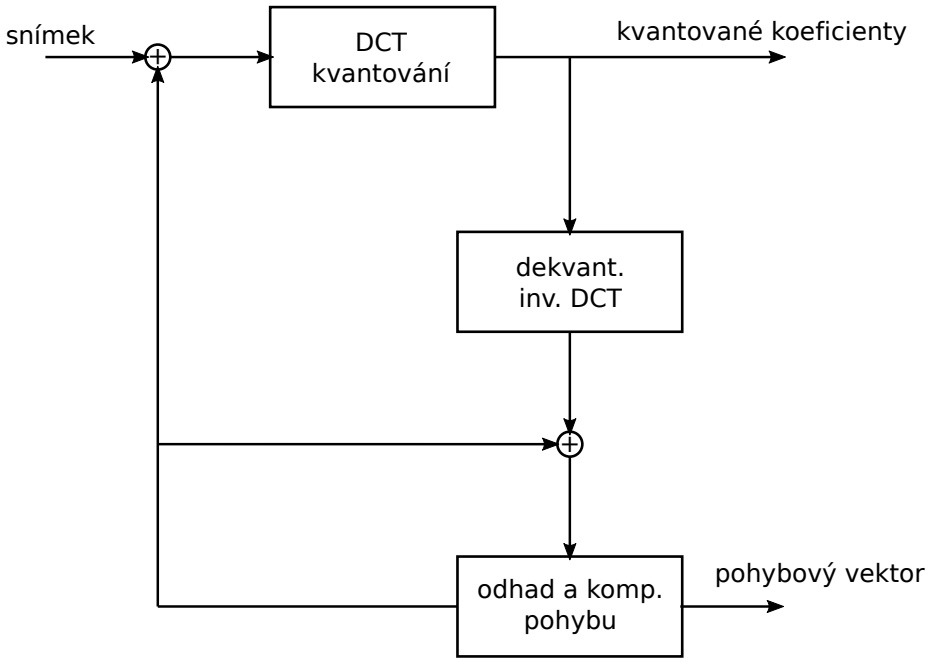
Hybridní kodéry videa nekomprimují jednotlivé snímky nezávisle, ale využívájí také souvislosti mezi nimi. To jim umožňuje dosáhnou vyšších kompresních poměrů proti kodérům, které využívaly pouze prostorové redundance v rámci jednotlivých snímků.

5.3.1 H.261

H.261 je standard ITU-T z roku 1988. Specifikace má pouze 29 stran. Jedná se o jednoduchého zástupce hybridního kodéru. Formát je navržen pro rozlišení CIF a QCIF a formát obrazu YC_bC_r 4:2:0. Makroblok má pevné rozměry 16×16 na složce Y, 8×8 na C_b a 8×8 na C_r. Pro kódování intra-snímků se využívá DCT po blocích 8×8. Postup je shodný s kompresní metodou JPEG. Kódování inter-snímků je založeno na kompenzaci pohybu. Na jeden makroblok se vytvoří jeden pohybový vektor, který udává posun vůči předchozímu snímku. Pohybové vektory mají přesnost na celé pixely. Formát využívá in-loop filtr.

5.3.2 MPEG-1 Video

MPEG-1 je komplexní standard ISO, který byl v roce 1993 vytvořen v rámci skupiny MPEG (Motion/Moving Picture Experts Group). Komprese videa je specifikována v části 2 (Video) tohoto standardu, formálně ISO/IEC 11172-2. Jedná se o hybridní kodér a vychází především z H.261 a JPEG. Používán je například ve formátu VCD; může být použit na SVCD a Video DVD. Byl opět navržen na formát obrazu YC_bC_r 4:2:0. Obsahuje klasické I/P/B-snímky a D-snímky (jen DC koeficienty, pro rychlé převíjení). GOP má typicky velikost 15–18 snímků. Blok má klasickou velikost 8×8, makroblok stejně jako u H.261 16×16 pro složku Y. I-snímky se komprimují jako JPEG (DCT). P-snímky využívají blokovou kompenzaci pohybu proti předcházejícímu



Obrázek 5.8: Zjednodušené schéma H.261. Jedná se o typický hybridné kodér.

I- nebo P-snímku. Pohybové vektory se tvoří pro každý makroblok s přesností na půlpixel (half-pel). P-snímky obsahují jeden pohybový vektor na makroblok, B-snímky dva. Kvůli použití B-snímků a tedy změně pořadí snímků musí být v kontejneru ke snímkům poznačen DTS (*decoding time stamp*). MPEG-1 umoňuje použít pouze konstantní datový tok (CBR). Mezi kodeky implementující tento standard patří např. libavcodec.

5.3.3 MPEG-2 Video, H.262

Komprese videa specifikována v části 2 (Video) standardu MPEG-2, formálně ISO/IEC 13818-2 a také ITU-T H.262. Standard pochází z roku 1995 a vylepšuje kompresi videa specifikovanou v MPEG-1. Má širokou podporu a používá se na SVCD, DVD, DVB, HD DVD, Blu-ray a digitální TV (DVB-T). Přináší podporu pro prokládané (*interlaced*) video (řádky složky Y každého bloku jsou přeskládány). Podporuje podvzorkování 4:2:2, 4:2:0 a 4:4:4 (takže makrobloky mají různé složení). MPEG-2 používá I/P/B-snímky, ale vypouští D-snímky (díky vyššímu výpočetnímu výkonu lze místo D-snímků použít koeficienty DC z I-snímků). Na makroblok opět připadá 1 pohybový vektor (2 v případě B-snímků). Standard je složitější než MPEG-1, má 243 stran. Specifikuje profily a levely (např. SP nemá B-snímky). Nejčastěji se používá profil ASP (Advanced Simple Profile). Na rozdíl od MPEG-1 umožňuje MPEG-2 použít variabilní datový tok (VBR). Mezi kodeky implementující tento standard patří např. libavcodec.

5.3.4 MPEG-4 Visual

Komprese videa specifikována v části 2 (Visual) standardu MPEG-4 z roku 1999, formálně ISO/IEC 14496-2. Formát opět vylepšuje předešlé standardy a je založen především na ITU-T H.263. Byl navržen pro formát YC_bC_r 4:2:0 a využívá klasické I/P/B-snímky. Na rozdíl od předchozí standardů vypouští in-loop filtr. Standard je značně složitý, má 517 stran, popisuje uložení komplexní scény (3D tvar) a opět specifikuje profily. Profil SP je bez B-snímků. Profil ASP umožňuje využít B-snímky, řeší prokládání, pohybové vektory mají čtvrtpixelovou přesnost (Qpel) a je možné využít též globální kompenzaci pohybu (GMC). Právě Qpel a GMC se liší od předchozích standardů. Má mnoho implementací (např. DivX, Xvid, 3ivx, libavcodec) a tudíž i mnoho FourCC kódů.

5.3.5 MPEG-4 AVC, H.264

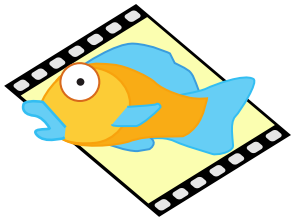
Komprese videa je v MPEG-4 specifikována ještě jednou a to v části 10 (Advanced Video Coding). Formálně se jedná o ISO/IEC 14496-10 a ITU-T H.264. Standard tohoto hybridního kodéru pochází z roku 2003 a přináší mnoho nových vlastností. Používá se na Blu-ray, HD DVD a DVB (stanice v rozlišení HD). Byl navržen pro formáty 4:0:0 (monochromatický), 4:2:0, 4:2:2 a 4:4:4. Definuje I/P/B-snímky, používá DCT a WHT (Walshova–Hadamardova transformace) pro DC koeficienty z DCT. U intra-snímkového kódování zavádí standard H.264 před DCT také predikci transformovaného bloku. Dále dovoluje použít bloky transformace (DCT) různých velikostí. Makroblok je zde rozdělen na oblasti. Bloková kompenzace pohybu se provádí pro bloky 4×4 až 16×16. Z toho plyne, že lze použít více pohybových vektorů na jeden makroblok. Pro predikci P/B-snímku lze využít až 16 referenčních snímků (namísto 1/2 u předchozích standardů). Navíc se zde u B-snímků používá vážená predikce (váha, ke kterému referenčnímu snímku je kódovaný blok blíž). Na rozdíl od předchozích standardů, které využívaly pouze VLC kódy (Huffmanovy), lze v MPEG-4 AVC použít buďto entropický kodér CAVLC (VLC kódy) nebo CABAC (aritmetické kódování). Proti části 2 standardu MPEG-4 přidává ještě in-loop filtr. Standard je opět značně složitý a má 680 stran. Mezi významné kodeky implementující tento standard patří libavcodec (pouze dekomprese), x264 (pouze komprese), DivX H.264 nebo CoreAVC (pouze dekomprese). MPEG-4 AVC používá FourCC kódy `AVC1` nebo `H264`.

5.3.6 VP3, Theora

Standard VP3 je z roku 2004, má krátkou specifikaci (cca 1000 rádků). Není ovšem známa jeho oficiální specifikace, pouze zpětně vytvořený popis formátu. Za standardem stojí společnost On2. Stejně jako v předchozích případech se jedná o dalšího zástupce hybridních kodérů založených na DCT. Standard byl navržen na formát obrazu YC_bC_r 4:2:0. Na rozdíl od standardů z rodiny MPEG nedefinuje I/P/B-snímky. Namísto toho

používá tzv. golden a inter-snímky. Golden snímky mají roli I-snímků, inter-snímky pak P-snímků. B-snímky nejsou použity. Také další terminologie je mírně odlišná. Blok 8×8 se zde nazývá fragment. Skupina 4×4 (=16) fragmentů utváří superblok (kanály stále nezávisle). Makroblok je tvořen klasicky 4 bloky v Y, a po jednom v C_b a C_r. Bloky v superbloku se zpracovávají v pořadí daném Hilbertovou křivkou. Golden snímky se komprimují obdobně jako JPEG. U inter-snímků lze makroblok kódovat 8 režimy, např. klasickou BMC (pohybový vektor). VP3 používá FourCC kódy `VP30` a `VP31`.

Formát VP3 byl s malými rozšířeními v roce 2004 otevřen pod názvem Theora. Theora je určena pro kontejner Ogg a její vývoj zastřešuje organizace Xiph.org Foundation. Pro formát platí totéž, co pro VP3. Navíc existuje referenční implementace libtheora. Theora využívá FourCC kód `theo`.

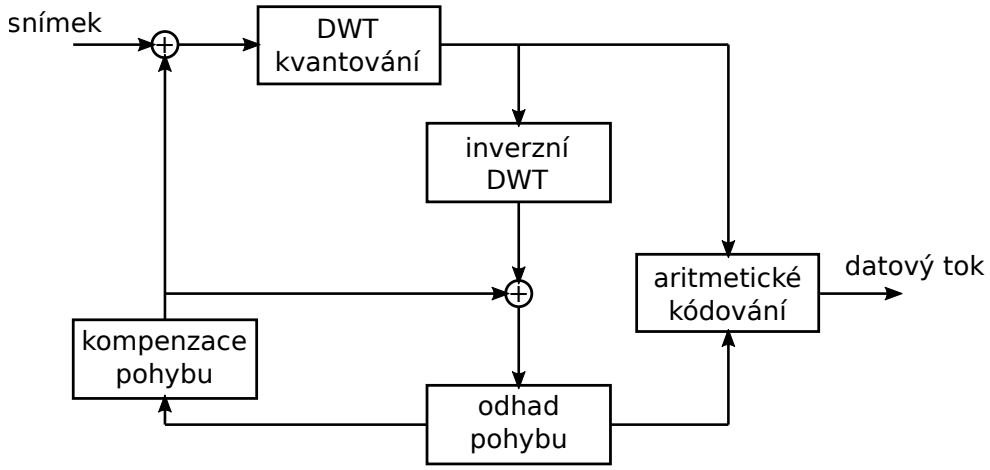


5.3.7 VP8

VP8 je další formát z rodiny VP od společnosti On2. V roce 2010 byl však spolu s touto společností odkoupen a otevřen Googlem. Google zamýšlí jeho použití pro video v HTML5. Z technického hlediska se jedná o konkurenci formátu H.264. VP8 je hybridní kodér, který používá DCT a WHT (pro DC koeficienty z DCT pro Y). Nedefinuje B-snímky, zavádí však golden snímky a altref snímky. Inter-snímky mohou být zapsány proti kterémukoli předchozímu snímku z GOP. Roli referenčního snímku může hrát poslední golden, poslední altref nebo přímo předcházející snímek. Pro dekompresi je třeba mít paměť pouze pro 4 snímky (golden, altref, předchozí a současný). Na rozdíl od předchozí standardů definuje přesnou DCT za použití celočíselné aritmetiky (bijektivní mapování). Je tedy přesně známa hodnota dekomprimovaného pixelu. Standard je navržen pouze 8bitový formát YC_bC_r 4:2:0. Každý snímek se dělí klasicky na makrobloky ($16 \times 16Y$). Ty se zpracovávají v rastrovém průchodu. Makrobloky se dělí na 4×4 podbloky (16 Y, 4 U, 4 V). Podbloky se též zpracovávají v rastrovém pořadí. Transformace (DCT) se aplikuje na podbloky 4×4 vzorky. Pohybové vektory mají čtvrtpixelovou přesnost (Qpel). Vektory pro barvonosné podbloky se spočtou jako průměr vektorů ze 4 odpovídajících podbloků v Y. VP8 využívá in-loop filtr, který odstraní blokové artefakty. Protože nejsou použity B-snímky, není třeba měnit jejich pořadí. Jako entropický kodér je použita varianta aritmetického kodéru, tzv. *boolean entropy coder*. Specifikace je tvořena pomocí úseků zdrojových kódu v C, má 303 stran, ale je čtivá. Formát má podporu ve webových prohlížečích a referenční implementaci libvpx.

5.3.8 Dirac

Formát Dirac je dalším zástupcem hybridních kodérů. Na rozdíl od všech předchozích není založen na DCT, ale na DWT (diskrétní vlnková transformace). Standard pochází z roku 2008 a jeho vývoj je zastřešen společností BBC. Jedná se o otevřený standard se dvěma otevřenými (open source) referenčními implementacemi. Knihovna Schrödinger je napsána v C (optimalizovaný kodek) a knihovna dirac-research (původně Dirac) v C++. Standard je určen pro formáty YCC s podvzorkováním 4:4:4, 4:2:2, 4:2:0 a řeší také prokládání. Pro DWT je zde na výběr 7 vlnek. Jinak pracuje jako ostatní hybridní kodéry (např. z rodiny MPEG) – využívá I/P/B-snímky a blokovou kompenzaci pohybu (pohybové vektory). Schéma je na obr. 5.9.



Obrázek 5.9: Schéma hybridního kodéru formátu Dirac. Namísto typické DCT je použita DWT. Jinak se schéma podstatně neliší.

Jako entropický kodér je možné použít kontextový aritmetický kodér nebo exp-Golombovy kódy. Podmnožina formátu byla standardizována jako VC-2 (též Dirac Pro) a specifikuje pouze I-snímky. Díky podstatě DWT nevzniká po dekompresi blokový efekt. Specifikace má 134 stran a použitý FourCC kód je `BBCD`.

5.4 Shrnutí

Výše popsaný přehled kompresních formátů videa není vyčerpávající. Chybějí zde např. další formáty z rodiny VP, formát RealVideo, Indeo, rodina WMV a další. Některé formáty jsou standardizovány mezinárodní organizací, jiné mají jen specifikaci. Některé ale nemají oficiálně ani tu, jsou k nim veřejně dostupné pouze zdrojové kódy, případně jen vytvořená videa. Situaci chronologicky shrnuje tabulka 5.2.

formát	standard	komprese	
		bezeztrátová	ztrátová
H.120	ITU-T H.120	ne	ano
H.261	ITU-T H.261	ne	ano
MS RLE	ne (jen dokumentace)	ano	ne
Motion JPEG	ne	ne	ano
MPEG-1 Video	ISO/IEC-11172-2	ne	ano
MPEG-2 Video, H.262	ISO/IEC 13818-2, ITU-T H.262	ne	ano
H.263	ITU-T H.263	ne	ano
MPEG-4 Visual	ISO/IEC 14496-2	ne	ano
Huffyuv	ne (není specifikace)	ano	ne
FFV1	ne (jen specifikace)	ano	ne
MPEG-4 AVC, H.264	ISO/IEC 14496-10, ITU-T H.264	ne[a]	ano
Lagarith	ne (není specifikace)	ano	ne
VP3	ne (není specifikace)	ne	ano
Theora	ne (jen specifikace)	ne	ano
VP8	RFC 6386	ne	ano
Dirac, VC-2	(jen specifikace), SMPTE ST 2042[b]	ano	ano
HEVC, H.265	ISO/IEC 23008-2, ITU-T H.265	ne[c]	ano
VP9	RFC[d]	ne	ano
VC-3	SMPTE ST 2019	ne	ano

Tabulka 5.2: Přehled formátů videa.

[a] Umožňuje také bezeztrátovou kompresi intra-snímků.
[b] Je standardizována pouze podmnožina formátu s názvem VC-2.
[c] Umožňuje také bezeztrátovou kompresi intra-snímků.
[d] V době psaní této studijní opory ještě není známo číslo RFC.

Kapitola 6

Jednoduchá multimediální rozhraní

Tato kapitola popisuje jednoduchá API, které mohou být použita při práci s multimédii. Nejedná se však o plnohodnotné multimediální frameworky, které jsou blíže rozebrány až v následující kapitole.

6.1 OpenCV

OpenCV je multiplatformní knihovna pro manipulaci s obrazem. Je vyvíjena převážně Intelem a zaměřena na algoritmy počítačového vidění v reálném čase. Umožňuje také základní práci s videem. Knihovnu je možné využít z prostředí jazyků C, C++ a s generátorem rozhraní SWIG také v dalších jazycích (Python, Octave).

6.1.1 Základy

Následující text se bude věnovat rozhraní v C++, nikoli v C. Dokumentace lze nalézt na oficiálních stránkách.[1] Ve vlastní aplikaci je třeba includovat potřebné hlavičkové soubory. Minimálně to bude `<cv.h>`, pak `<highgui.h>`. Všechny třídy leží ve jmenném prostoru `cv`. Jádrem rozhraní v C++ je třída `Mat`, která může reprezentovat statický obraz (snímek videa). Načtení obrázku v podporovaném formátu se provede pomocí funkce `imread`, zápis pak pomocí `imwrite`. Obrázek bude načten v barevném modelu BGR24 (tři kanály, datový typ `uchar`). K zobrazení slouží funkce `imshow` (otevře okno).

K vlastním obrazovým datům lze přistoupit pomocí ukazatele `.data`. Krok mezi řádky (*stride*) je uložen v položce `.step`. Na obrazová data je tedy možné přímo zavolat funkce `compress` nebo `decompress` z kapitoly 2.6. Takové použití je demonstrováno v následujícím úseku kódu.

[1] `http://opencv.willowgarage.com/`

```
Mat image = imread(imagename, CV_LOAD_IMAGE_COLOR);
imshow("original", image);

byte *buff = new byte[get_max_compressed_size(image.cols, image.rows)];
compress(image.data, buff, image.cols, image.rows, image.step);
decompress(image.data, buff, image.cols, image.rows, image.step);

imshow("decompressed", image);
waitKey();
```

Příklad 6.1: Načtení barevného obrázku, jeho následná komprese a dekomprese. Bylo vynecháno ošetření chyb.

K práci s videem slouží třída `VideoCapture`. Jako jediný parametr konstruktoru je třeba předat název požadovaného souboru. Snímek videa lze získat například pomocí operátoru ».

```
VideoCapture cap(filename);
Mat frame;

while(1)
{
    cap >> frame;
    if(!frame.data)
        break;
    imshow("frame", frame);
    waitKey();
}
```

Příklad 6.2: Úsek kódu, který postupně dekóduje všechny snímky videa. Práce se snímkem je stejná jako v případě statického obrazu. Opět bylo vynecháno ošetření chyb.

Stejná třída zaobaluje také práci se zachytávacími zařízeními (webkamera). V tomto případě se jako parametr konstruktoru předá číslo zařízení (v případě jediné webkamery to bude 0).

```
VideoCapture cap(0);
```

Příklad 6.3: Úsek kódu, který otevře zachytávací zařízení s indexem 0. Zpracování je dále shodné s předchozím příkladem.

6.1.2 Práce s obrazem

K převodu barevných modelů slouží funkce `cvtColor`. Ta jako parametr kód, který uvádá požadovanou transformace (např. `CV_BGR2YCrCb` či `CV_BGR2GRAY`).

```
cvtColor(image, image, CV_BGR2GRAY);
```

K rozmazání obrazu lze použít funkci `GaussianBlur`, která použije filtr tvaru Gaussova okna. Jako parametry je možné zadat velikost okna a standardní odchylky ve směru os.

```
GaussianBlur(image, image, Size(), 5);
```

K detekci hran je v knihovně implementován Cannyho detektor. Parametry funkce `Canny` jsou např. prahy hystereze.

```
Canny(image, image, 0, 30);
```

K výpočtu DCT slouží funkce `dct`. Převod zpět je možné zajistit buď voláním `idct` nebo opět `dct` ovšem s parametrem `DCT_INVERSE`. Zobrazení je vhodné provést logaritmicky.

```
dct(image, image);
```

Kapitola 7

Multimediální frameworky

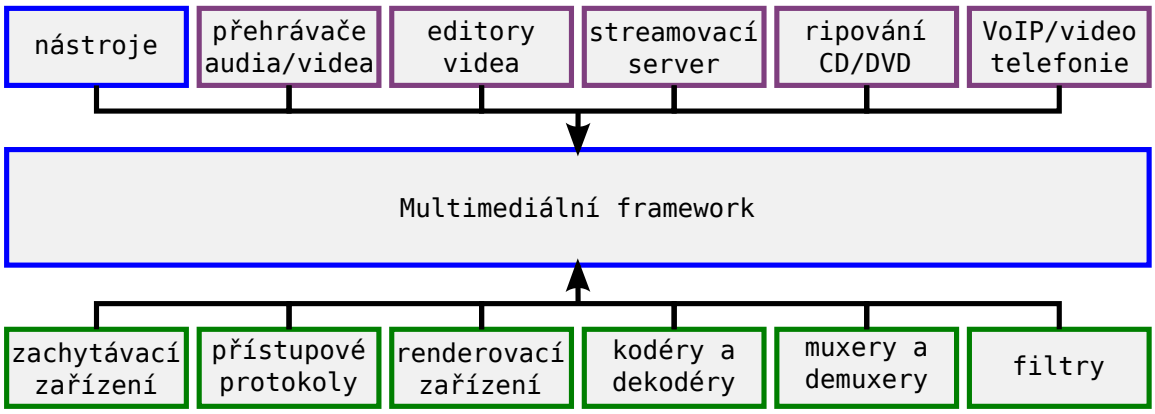
Tato kapitola obsahuje přehled některých známých multimediálních frameworků. Jsou zde probrány frameworky Video for Windows, DirectShow, FFmpeg a GStreamer. Přehled samozřejmě není kompletní. Mezi další významné frameworky patří např. Media Foundation, xine, Phonon, QuickTime, VLC nebo Helix DNA Client.

Před vlastním přehledem je nejprve nutné vysvětlit, co to vlastně multimediální framework je. Jedná se o soubor knihoven a nástrojů pro práci s multimediálními daty, nejčastěji videem a audiem. Na tento framework můžeme nahlížet z pohledu aplikace, která v sobě chce jeho služby využít. Aplikací může být třeba audio nebo video přehrávač, editor videa, streamovací server apod. Například přehrávač může po frameworku žádat otevření souboru s videem a postupnou dekompresi některých snímků. Editor videa bude chtít snímky také zkomprimovat a uložit do kontejneru. Klient pro konference VoIP bude požadovat otevření zvukové karty a kompresi zachytávaného zvuku. Příkladů může být spousta.

Framework v sobě zaobaluje určitou minimální funkcionalitu, kterou je možno dále rozšiřovat (většinou instalací zásuvných modulů). V tomto případě se na framework nahlíží z pohledu tohoto modulu, což může být např. nový kodek, postprocessingový filtr, modul pro přístup ke zvukovému subsystému, modul umožňující využít hardwarovou akceleraci při dekódování videa apod. Zásuvné moduly rozšiřující funkcionalitu frameworku budou zřejmě využívat odlišné části API frameworku. Celou situaci shrnuje blokový diagram na obrázku 7.1.

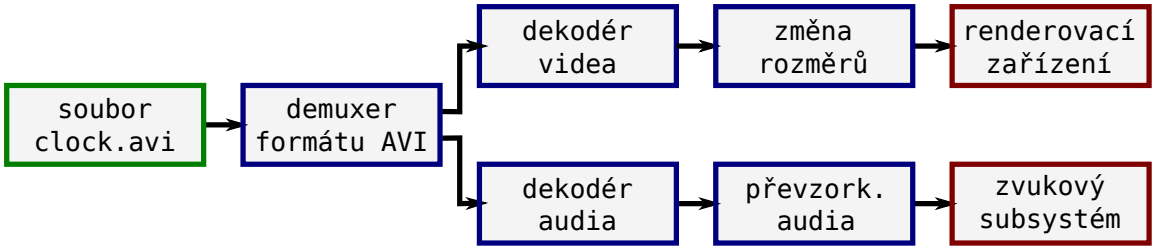
Rozšíření funkcionality frameworku často vyžaduje administrátorská oprávnění (není-li framework nainstalován jen v adresáři určitého uživatele). Některé frameworky (např. GStreamer) umožňují uživateli načíst rozšiřující moduly i bez instalace do systému. U některých to není z principu možné, protože přidání nové funkcionality vyžaduje znovupřeložení frameworku. To je případ frameworku FFmpeg.

Frameworky mohou mít uvnitř fixní graf toku dat (*pipeline*). To je případ nejstaršího zde probíraného frameworku – Video for Windows. To samozřejmě použití frameworku velmi omezuje (aplikace jej může použít jen k přesně definovaným úkonům). Většinou však lze tok dat (graf filtrů) uvnitř frameworku ovlivnit. Třeba DirectShow nebo GStreamer umožňují klientské aplikaci postavit si graf téměř libo-



Obrázek 7.1: Vztah multimediálního frameworku, klientských aplikací a rozšiřujících modulů. Framework sám o sobě poskytuje jen omezenou funkcionalitu. Tu lze ovšem rozšiřovat instalací nových částí (např. kodek nebo filtr s video efekty). Mezi aplikace využívající framework se řadí také ty nástroje, které jsou šířeny spolu s frameworkem (např. konzolové nástroje umožňující stavbu grafu filtrů).

volně a poskládat tak jakoukoli aplikaci (přehrávač audia, rekomprese videozáznamu nebo zachytávaní síťového vysílání). Jednoduchý graf filtrů je zobrazen na obr. 7.2.



Obrázek 7.2: Graf filtrů pro přehrávač videa. Zeleně je označen zdrojový filtr (soubor z disku), modře transformační filtry (demuxer, dekodéry, konvertory formátů) a červeně výstupní filtry (obrazovka, zvuková karta).

Filtry zde zobrazené lze rozdělit do tří skupin – zdroje datového toku, transformace a cílové filtry. Takové rozdělení lze víceméně nalézt u všech frameworků (přímo např. u DirectShow). Jednotlivé filtry jsou propojeny přes vstupy/výstupy, které se v závislosti na frameworku nazývají např. pady nebo piny. Filtr může mít dle svého určení žádný, jeden nebo více vstupů a výstupů. Aby bylo možné dva filtry propojit, musejí se shodnout na typu datového toku. Nelze např. napojit dekodér videa na vstup zvukové karty. Některé framewoky umějí formát dat v omezené míře konvertovat. Lze tak automaticky provádět změnu vzorkovací frekvence u audia nebo formátu pixelu u videa. Jiné frameworky (VfW) však tohle neumožňují. V takovém případě je nutné, aby se navazující filtry na typu dat shodly (jinak se nespojí).

7.1 Video for Windows

Video for Windows (VfW) je multimediální framework pro systémy Microsoft Windows. Vyvinul jej Microsoft jako reakci na framework QuickTime od firmy Apple. První ještě 16bitová verze byla uvedena v listopadu roku 1992. Framework přišel s vlastním souborovým formátem (multimediálním kontejnerem) Audio Video Interleave (AVI). Jeho nástupcem se stal framework DirectShow. Nás teď bude zajímat pouze jeho část nazývaná Video Compression Manager (VCM), která poskytuje rohraní pro videokodeky. Analogicky existuje Audio Compression Manager (ACM). Dokumentaci k veškerému API naleznete na MSDN.[1] Dnes je tento framework označen ze strany Microsoftu za zastaralý. Jeho použití je nicméně velmi jednoduché a každý dnes používaný kodek pro něj implementuje modul.

7.1.1 Přehrávač

Rozhraní AVIFile umožňuje aplikaci pracovat s formátem AVI, což znamená hlavně číst a ukládat datové stopy (audio, video stopy). Odstiňuje přitom volání kompresoru, resp. dekompresoru. Aplikace tedy postupně žádá framework o dekomprimované snímky (na určité pozici) a framework sám najde a použije potřebný VfW kodek. Nelze přitom použít DirectShow kodeky (naopak v DirectShow použít kodeky pro VfW lze). Práci s funkcemi AVIFile je nutné započít voláním `AVIFileInit` a ukončit s `AVIFileExit`. Otevření souboru AVI provede `AVIFileOpen`. Nyní je možné procházet a otevírat jetnotlivé stopy (toky dat uvnitř kontejneru). K otevření slouží funkce `AVIFileGetStream`, které lze jako jeden z parametrů předat typ požadované stopy (např. `streamtypeVIDEO` pro video stopu). Informace o otevřeném toku je možné získat pomocí `AVIStreamInfo`, která vyplní strukturu `AVISTREAMINFO`. Z té lze pak vyčíst informace o snímkové frekvenci (podíl `dwRate/dwScale`) a počet snímků (`dwLength`). K získání informací o formátu snímků slouží `AVIStreamReadFormat`, která vyplní strukturu `BITMAPINFOHEADER` (více v kapitole 4.1.1). Z té je možno vyčíst hlavně rozměr snímku, bitovou hloubku a použitou kompresi (FourCC kód). Zobrazení FourCC kódu pomocí `printf` jako sekvence čtyř znaků je vidět níže.

```
DWORD dwFCC;

printf("%c%c%c%c",
    (dwFCC>>000)&0xff, (dwFCC>>010)&0xff,
    (dwFCC>>020)&0xff, (dwFCC>>030)&0xff);
```

Dekompresi jednotlivých snímků započne `AVIStreamGetFrameOpen`, která v parametru obdrží cílový formát. Framework VfW neumí konvertovat barevné modely (formáty pixelu), naopak DirectShow konverzi provádí automaticky. Takže pokud třeba kodek umí dekomprimovat pouze do RGB8 (paleta) a aplikace požaduje BGR24 (`biCompression = BI_RGB` a `biBitCount = 24`), funkce

[1] `http://msdn.microsoft.com/`

`AVIStreamGetFrameOpen` selže. Nyní už je možné dekomprimovat pomocí `AVIStreamGetFrame` jednotlivé snímky (funkce obrží jako parametr číslo požadovaného snímku). Pro Microsoft Windows (formát DIB) je typické uložení snímku zdola nahoru (vzhůru nohama). Orientace shora dolů (normálně) je indikována zápornou výškou (`biHeight`). Převrácení snímku lze provést stejným trikem jako je popsán v sekci 7.3.4. Krok (*stride*) mezi následujícími řádky je vždy šířka zarovnaná ná nejbližší vyšší násobek 32 bitů. Po převrácení bude krok záporný.

Funkce

Důležité funkce jsou shrnuty níže. Pro překlad aplikace je třeba vložit hlavičkový soubor `<vfw.h>` a linkovat s `vfw32.lib`).

`AVIFileInit`
: Inicializuje knihovnu AVIFile.

`AVIFileExit`
: Ukončí práci s knihovnou.

`AVIFileOpen`
: Otevře soubor AVI.

`AVIFileRelease`
: Zavře soubor.

`AVIFileGetStream`
: Vrátí ukazatel na objekt reprezentující vybranou stopu (např. video stopa) uvnitř souboru AVI.

`AVIStreamInfo`
: Vyplní strukturu s informacemi o stopě (trvání, snímková frekvence).

`AVIStreamFormatSize`
: Zjistí velikost nutnou pro uložení struktury o použitém formátu stopy.

`AVIStreamReadFormat`
: Vyplní strukturu s informacemi o použitém formátu stopy (rozlišení, barevná hloubka, FourCC kód).

`AVIStreamGetFrameOpen`
: Připraví dekompresor k dekódování jednotlivých snímků. Je třeba mít nainstalovaný formátu odpovídající VfW kodek.

`AVIStreamGetFrame`
: Dekomprimuje a vrátí ukazatel na požadovaný snímek.

`AVIStreamGetFrameClose`
: Ukončí dekompresi, uvolní prostředky.

`AVIStreamOpenFromFile`
: Otevře vybranou stopu (např. video) rovnou ze souboru AVI.

`AVIFileCreateStream`
: V existujícím souboru vytvoří novou stopu.

`AVIStreamSetFormat`
: Nastaví formát stopy.

`AVIStreamStart`
: Vrátí počáteční číslo snímku pro danou stopu.

`AVIStreamEnd`
: Vrátí číslo posledního snímku.

`AVIStreamRead`
: Přečte požadovaný počet nedekomprimovaných dat stopy.

`AVIStreamWrite`
: Zapíše data do stopy.

`AVIStreamRelease`
: Uzavře otevřenou stopu, uvolní prostředky.

Struktury

Následují důležité struktury.

`AVISTREAMINFO`
: Obsahuje informace o stopě, tj. především typ (audio, video), FourCC kód kompresoru pro uložení dat, snímková frekvence, číslo počátečního snímku a počet snímků.

`BITMAPINFOHEADER`
: Obsahuje informace o snímku, tj. především rozměry a typ použití komprese (FourCC kód) a barevnou hloubku (bity na pixel).

Zdrojový kód ukázkového přehrávače lze stáhnou na stránkách předmětu.

7.1.2 Videokodek

Tato sekce se týká rozhraní Video Compression Manager (VCM). Zásuvný modul (kodek) do tohoto frameworku je tzv. instalovatelný ovladač. Jedná se od dynamicky linkovanou knihovnu (DLL), kterou je nutné do systému Windows nainstalovat (vyžaduje administrátorská oprávnění). Stačí tedy sestavit jeden zásuvný modul a ten správně zaregistrovat. Při překladu je třeba vložit `<vfw.h>` a sestavovat s `winmm.lib`. Vstupem každého instalovatelného ovladače je funkce `DriverProc`, která přijímá zprávy od systému a vykonává odpovídající akce. Příkladem takové zprávy je žádost o kompresi (`ICM_COMPRESS`) či dekompresi `ICM_DECOMPRESS` jednoho snímku. Kostra kodeku je uvedena v příkladu 7.1.

```
#include <vfw.h>

LRESULT WINAPI DriverProc(DWORD dwDriverId, HDRVR hdrvr, UINT msg,
    LONG lParam1, LONG lParam2)
{
    switch(msg)
    {
        case ICM_COMPRESS:
            return Compress((ICCOMPRESS*)lParam1, (DWORD)lParam2);

        case ICM_DECOMPRESS:
            return Decompress((ICDECOMPRESS*)lParam1, (DWORD)lParam2);
    }
}
```

Příklad 7.1: Struktura ovladače pro framework Video for Windows. Jádrem je funkce `DriverProc` reagující na zprávy `ICM_COMPRESS` a `ICM_DECOMPRESS`. Funkce `Compress` a `Decompress` jsou uživatelské funkce, ve kterých je provedena vlastní komprese, resp. dekomprese snímku.

Funkce `DriverProc` buď zprávu zpracuje nebo ji předá výchozímu handleru `DefDriverProc`. Zprávy, na které by kodek měl reagovat, a očekávaná akce jsou uvedeny níže.

`ICM_ABOUT`
: Zobrazit dialog s informacemi o kodeku.

`ICM_COMPRESS`
: Zkomprimovat obdržený snímek do připraveného bufferu.

`ICM_COMPRESS_BEGIN`
: Připravit se na kompresi podle předaných parametrů a alokovat potřebné datové struktury podle obdrženého formátu videa. Může být obdržena i uprostřed komprese.

`ICM_COMPRESS_END`
: Konec komprese, uvolnit alokované prostředky.

`ICM_COMPRESS_GET_FORMAT`
: Vrátí informace o výstupním (komprimovaném) formátu videa.

`ICM_COMPRESS_GET_SIZE`
: Podle požadovaného vstupního a výstupního formátu videa vrátí maximální velikost zkomprimovaného snímku (nutné pro alokaci paměti pro výstup).

`ICM_COMPRESS_QUERY`
: Vrátí informaci, zdali je podporován předaný vstupní a případně výstupní formát.

`ICM_CONFIGURE`
: Zobrazí konfigurační dialog (např. kvalita videa).

`ICM_DECOMPRESS`

Dekomprimuje obržený snímek.

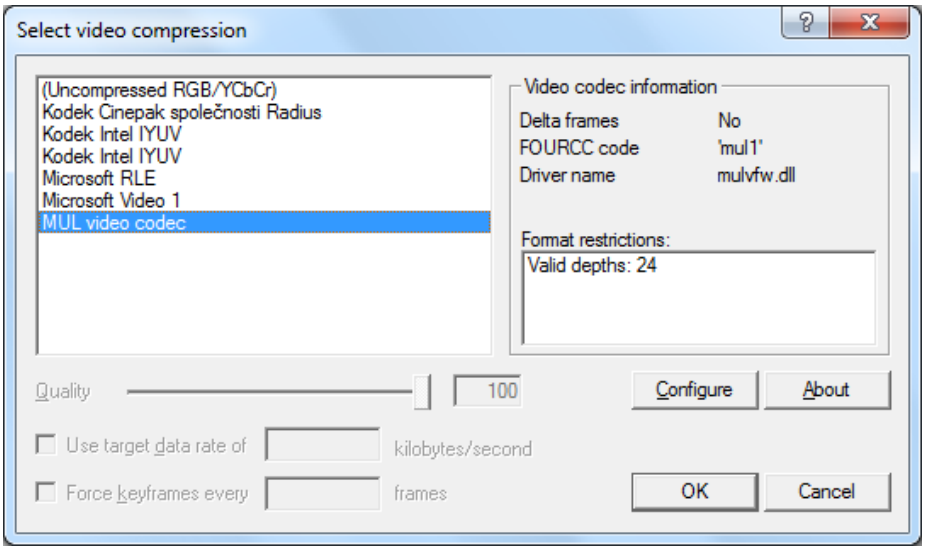
`ICM_DECOMPRESS_BEGIN`
: Připravit se na dekompresi podle předaných parametrů, alokovat potřebné prostředky. Může být časově náročné.

`ICM_DECOMPRESS_END`
: Konec dekomprese, uvolnit prostředky.

`ICM_DECOMPRESS_GET_FORMAT`
: Vrátí informace o výchozím formátu pro dekomprimovaná data pro daný vstupní formát.

`ICM_DECOMPRESS_QUERY`
: Vrátí informaci, zdali podporuje daný vstupní (komprimovaný) formát, případně v kombinaci s požadovaným výstupním (dekomprimovaným) formátem.

`ICM_GETINFO`
: Vrátí informace o kodeku (název).



Obrázek 7.3: Screenshot dialogu s výběrem VfW kodeků v programu VirtualDub. Jedná se o 32bitovou verzi programu a tedy i kodeku.

Typický scénář komprese může vypadat takhle: `ICM_GETINFO`, `ICM_COMPRESS_QUERY`, `ICM_COMPRESS_GET_FORMAT`, `ICM_COMPRESS_GET_SIZE`, `ICM_COMPRESS_BEGIN`, `ICM_COMPRESS`, `ICM_COMPRESS`, `ICM_COMPRESS`, ..., `ICM_COMPRESS_END`. V reakci na `ICM_GETINFO` je třeba vrátit FourCC kód a název kodeku. Pro `ICM_COMPRESS_QUERY` je především nutné ověřit, zdali se jedná o podporovaný formát videa (RGB24). Pro `ICM_COMPRESS_GET_FORMAT` se provede ověření formátu a vrátí se FourCC kód. Pro `ICM_COMPRESS_GET_SIZE` se vrátí maximální velikost komprimovaného snímku. Při `ICM_COMPRESS_BEGIN` se ověří formát videa

a kodek se připraví na kompresi. Při `ICM_COMPRESS` se zkomprimuje obdržený snímek a vrátí se informace o tom, jestli byl vytvořen klíčový nebo rozdílový snímek. Konečně při `ICM_COMPRESS_END` se komprese ukončí, což může být v podstatě prázdná operace.

Jednoduchý kodek pro tento framewok je ke stažení na stránkách předmětu. Ve skutečnosti se jedná o obálku nad dříve navrženým RLE kodekem. To také ukazuje vhodnou architekturu kodeku. Vlastní kodek bude ve skutečnosti na frameworku nezávislá knihovna. Pro každý žádaný framework je pak napsána jen obálka, která volá funkce skutečného kodeku.

7.2 DirectShow

DirectShow (zkráceně DShow, DS, původně ActiveMovie) je framework firmy Microsoft, který se stal nástupcem frameworku Video for Windows. Je založen na objektovém modelu COM (Component Object Model). DirectShow je velmi obecný framework, protože graf filtrů uvnitř může být v podstatě libovolný. Graf se skládá ze tří typů filtrů – zdrojové, transformační a renderovací. Zdrojové jsou zdrojem datového toku, např. soubor na disku nebo zachytávací zařízení (webkamera). Transformační provádějí transformaci dat, např. dekódování komprimovaného videa, převod barevného modelu, filtrace audia apod. V renderovacích pak datový tok končí, např. výstup na obrazovku. Oproti frameworku VfW má DirectShow mnohé výhody. Mimo grafu filtrů je to např. možnost automatické konverze barevných modelů. Pro vývoj je nutno nainstalovat příslušné SDK (dnes Windows SDK, dříve DirectX SDK). Framework je zpětně kompatibilní se svým předchůdcem Video for Windows, jehož kodeky VCM jsou zde obaleny transformačním filtrem AVI Decompressor, případně AVI Compressor. Analogicky je zde ACM Wrapper Filter pro kodeky ACM. Součástí SDK je mimo jiné také nástroj GraphEdit, který umožňuje renderovací graf vizualizovat a poskládat jednoduše pomocí myši. Microsoft jej dnes (stejně jako VfW) označuje za zastaralý ve prospěch Media Foundation ze systému Windows Vista.

7.2.1 Základy

Jádrem frameworku je graf filtrů. Tyto filtry lze rozdělit na zdrojové, transformační a renderovací. Toto rozdělení odpovídá základním (bázovým) třídám (*base classes*), které jsou součástí SDK.[2] Například transformační filtry vycházejí ze třídy `CBaseFilter`. Transformační filtry jsou zde ve skutečnosti vícerého druhu – obecná třída `CBaseFilter`, transformace vstupu na výstup (třída `CTransformFilter`), transformace v místě neboli *in-place* (`CTransInPlaceFilter`) a třída `CVideoTransformFilter` určená pro dekodéry videa. Pokud tedy bude třeba implementovat vlastní transformační

[2]Pokud mají být tyto třídy využity, je nutné je po instalaci SDK přeložit jako statické knihovny, které se poté přilinkují k aplikaci.

filtr (např. video efekt, kodek), je možné podědit některou z těchto tříd. V případě `CTransformFilter` bude nutné implementovat alespoň metody `CheckInputType`, `GetMediaType`, `CheckTransform`, `DecideBufferSize` a `Transform` (více v sekci 7.2.3).

Graf filtrů se tedy skládá z filtrů trojího typu. Ty jsou mezi sebou spojeny piny. Aby se mohly takto filtry propojit, musejí podporovat stejný „transport“ (typicky přes lokální paměť) a shodnout se na typu dat. Data pak tečou z výstupního pinu jednoho filtru do vstupního pinu filtru následujícího. Zdrojové filtry produkují tok dat, který dále pokračuje přes filtry transformační až k filtrům renderovacím. Některé filtry mohou tok dat větvit nebo spojovat (splittery, muxery, overlaye). DirectShow definuje dva mechanizmy pro lokální paměťové transporty – modely „push“ a „pull“. V modelu push produkuje zdrojový filtr neustále data, která pak doručuje filtru následujícímu, který je pasivně přijímá, zpracovává a posílá dále. V modelu pull je zdrojový filtr napojen na filtr (typicky parser), který od něj data požaduje. Zdrojový filtr na to reaguje doručením dat, sám od sebe však nic neprodukuje.

Vlastní data (např. komprimovaný či nekomprimovaný snímek videa) jsou uložena v bufferech (jednoduše pole bajtů). Každý buffer je obalen objektem COM, který se nazývá mediální snímek (*media sample*, rozhraní `IMediaSample`). Tyto snímky jsou spravovány tzv. alokátory (rozhraní `IMemAllocator`). Každý pin má některý alokátor přiřazen. Alokátor jednoduše vytvoří několik (konečný počet) mediálních snímků a alokuje pro ně buffery. Pak je na požádání schopen tyto snímky přidělovat (metoda `GetBuffer`). Pokud jsou všechny právě přiděleny, čeká metoda na uvolnění některého z nich. Filtr tedy postupně žádá o snímky, plní je daty a posílá filtru následujícímu, který je zpracuje a snímek uvolní. Uvolnění umožní jeho opětovné použití prvním filtrem pro další data.

Všechny filtry v grafu se nacházejí v jednom ze stavů: zastaven (stopped), pozastaven (paused) nebo spuštěn (running). Smyslem pozastavení je naplnit graf filtrů daty, aby bylo jeho spuštění okamžité. Přechod mezi stavy je řízen správcem grafu filtrů (*filter graph manager*). Aplikace volá pouze metody `Run`, `Pause` a `Stop` objektu rozhraní `IMediaControl`. Správce pak zavolá odpovídající metody všech filtrů ve správném pořadí. Přechod mezi stavy zastaveno a spuštěno vždy probíhá přes stav pozastaveno.

Makra

Při práci s VfW či DS mohou být užitečná následující makra.

`SIZE_VIDEOHEADER`
: Udává velikost struktury `VIDEOINFOHEADER`.

`HEADER`
: Vrací adresu struktury `BITMAPINFOHEADER` uvnitř `VIDEOINFOHEADER`.

`DIBSIZE`
: Spočte velikost potřebnou pro uložení nekomprimované bitmapy v daném for-

mátu.

`DIBWIDTHBYTES`
: Spočítá krok mezi řádky, jako parametr vyžaduje strukturu `BITMAPINFOHEADER`, alternativně můžete použít následující makro `CALC_BI_STRIDE`.

```
#define CALC_BI_STRIDE(width,bitcount) \
    (((((width) * (bitcount)) + 31) & ~31) >> 3)
```

Protože vytvářený filtr nebude mít k dispozici standardní vstup a výstup, je možné pro ladění využít funkci `OutputDebugString` a její výstup zachytávat např. pomocí DebugView od Sysinternals.

Identifikátory

Multimediální formáty se v DirectShow identifikují pomocí tzv. GUID. GUID je 128bitový identifikátor. Starší FourCC kód je naproti tomu pouze 32bitový (4 čitelné znaky).

`MAKEFOURCC`
: Vytvoří FourCC kód (32bitové neznaménkové celé číslo) ze čtyř separátních znaků (char).

`FCC`
: Vytvoří FourCC kód ze znakového literálu.

`DEFINE_GUID`
: Bezpečně definuje a exportuje identifikátor GUID, musí následovat až po vložení hlavičkového souboru `<initguid.h>`. V případě GUID vytvořeného z FourCC kódu lze použít následující makro `DEFINE_FOURCCGUID`.

```
#define DEFINE_FOURCCGUID(name, dwFCC) DEFINE_GUID(name, dwFCC, \
    0x0000, 0x0010, 0x80, 0x00, 0x00, 0xAA, 0x00, 0x38, 0x9B, 0x71)
```

Následující tři definice jsou ekvivalentní a vytvoří FourCC kód „`YUY2`“.

```
DWORD fccYUY2 = MAKEFOURCC('Y','U','Y','2');
DWORD fccYUY2 = FCC('YUY2');
DWORD fccYUY2 = '2YUY';
```

FourCC kódy a identifikátory GUID lze vzájemně převádět. Ke konverzi slouží třída `FOURCCMap`. Pro zpětnou kompatibilitu GUID je pro FourCC kódy vyhrazen rozsah `XXXXXXXX-0000-0010-8000-00AA00389B71`. Následující dvě definice GUID jsou ekvivalentní.

```
FOURCCMap fccMap(FCC('YUY2'));
GUID g1 = (GUID)fccMap;
GUID g2 = (GUID)FOURCCMap(FCC('YUY2'));
```

7.2.2 Přehrávač

Pro následující aplikaci je třeba vložit hlavičkový soubor `dshow.h` a sestavovat se statickou knihovnou `strmiids.lib`. Na počátku aplikace se voláním funkce `CoInitialize` inicializuje knihovna objektů COM. Poté se funkcí `CoCreateInstance` vytvoří objekt rozhraní `IGraphBuilder`, který reprezentuje správce grafu filtrů. Voláním metody `QueryInterface` se lze dostat k objektům rozhraní `IMediaControl` a `IMediaEvent`. Správce grafu filtrů je možno požádat o automatickou stavbu grafu pro určený soubor pomocí metody `RenderFile`. Graf lze následně spustit voláním `IMediaControl::Run`. Nyní je vhodné počkat na konec přehrávání pomocí `IMediaEvent::WaitForCompletion`. V úseku kódu níže jsou vynechána ošetření chyb.

```
#include <dshow.h>
#include <tchar.h>

void main(void)
{
    IGraphBuilder *pGraph = NULL;
    IMediaControl *pControl = NULL;
    IMediaEvent   *pEvent = NULL;

    HRESULT hr = CoInitialize(NULL);

    hr = CoCreateInstance(CLSID_FilterGraph, NULL,
        CLSCTX_INPROC_SERVER, IID_IGraphBuilder, (void **)&pGraph);
    hr = pGraph->QueryInterface(IID_IMediaControl, (void **)&pControl);
    hr = pGraph->QueryInterface(IID_IMediaEvent, (void **)&pEvent);

    hr = pGraph->RenderFile(_T("clock.avi"), NULL);
    hr = pControl->Run();
    long evCode;
    pEvent->WaitForCompletion(INFINITE, &evCode);

    pControl->Release();
    pEvent->Release();
    pGraph->Release();
    CoUninitialize();
}
```

Příklad 7.2: Kostra jednoduché aplikace přehrávající pomocí DirectShow video `clock.avi`.

7.2.3 Videokodek

Následující text se věnuje jednoduchému dekodéru videa z kapitoly 2.6. Na počátku bude třeba deklarovat pro filtr GUID (zde CLSID). Ten je možno vytvořit z FourCC pomocí makra `DEFINE_FOURCCGUID` uvedeného výše.

```
static const DWORD FOURCC_MUL1 = FCC('mul1');
DEFINE_FOURCCGUID(CLSID_MUL1, FOURCC_MUL1);
```

Dále bude nejvhodnější cestou podědit třídu `CTransformFilter`. Je nutné implementovat minimálně abstraktní metody (uvedené v sekci 7.2.1). Jejich význam je popsán dále. V konstruktoru nové třídy (dekodéru) je vhodné alokovat a inicializovat privátní data.

```
class CMULDec : public CTransformFilter
{
public:
    CMULDec();
    HRESULT CheckInputType(const CMediaType *mtIn);
    HRESULT GetMediaType(int iPosition, CMediaType *pMediaType);
    HRESULT CheckTransform(const CMediaType *mtIn,
        const CMediaType *mtOut);
    HRESULT DecideBufferSize(IMemAllocator *pAlloc,
        ALLOCATOR_PROPERTIES *pProp);
    HRESULT Transform(IMediaSample *pSource, IMediaSample *pDest);
};
```

Metoda `CheckInputType` slouží k vyjednání podporovaných dat na spojení s předchozím filtrem (vstupní tok). Jakmile je do vstupního pinu předchozí filtr napojen, je napojován pin výstupní. Nejprve je zavolána metoda `GetMediaType`, která má za úkol vygenerovat seznam navrhovaných formátů pro výstupní pin. Pro ověření, jestli je konkrétní výstupní typ kompatibilní s typem vstupním, je volána metoda `CheckTransform`. Dalším krokem je nastavení vlastností alokátoru pro výstupní pin, tj. především velikost a počet bufferů. Výstupní pin zavolá metodu `DecideBufferSize`, která jako parametr obdrží ukazatel na alokátor a strukturu `ALLOCATOR_PROPERTIES` s požadavky následujícího filtru (výstupní tok). V této metodě je třeba provést úpravu vlastností a následné volání metody `IMemAllocator::SetProperties` výstupního alokátoru. Jádrem filtru je metoda `Transform`, která provede vlastní transformaci dat. V tomto případě se bude jednat o dekompresi jednoho snímku videa. Metoda obdrží ukazatele na vstupní a výstupní snímek (`IMediaSample`). Voláním metod `GetPointer` se filtr dostane k vlastním datům bufferu. Na ně se zavolá vlastní funkce `decompress` z kapitoly 2.6.

```
HRESULT CMULDec::Transform(IMediaSample *pSource, IMediaSample *pDest)
{
    HRESULT hr;
    BYTE *pBufferIn, *pBufferOut;
    hr = pSource->GetPointer(&pBufferIn);
    if(FAILED(hr))
        return hr;
    hr = pDest->GetPointer(&pBufferOut);
    if(FAILED(hr))
        return hr;
```

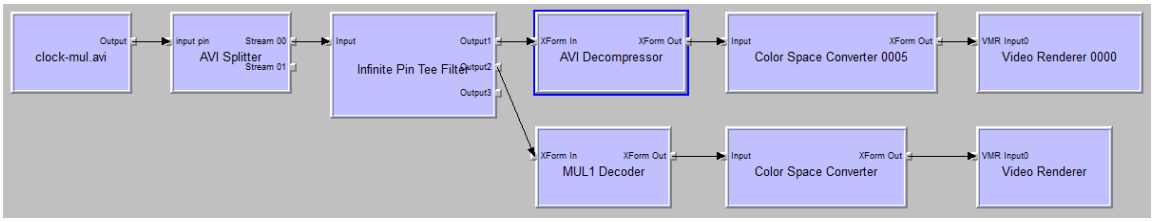
```
    CMediaType &mt = m_pInput->CurrentMediaType();
    ASSERT(*mt.FormatType() == FORMAT_VideoInfo);
    BITMAPINFOHEADER *pBmi = HEADER(mt.Format());

    DWORD cbDest = decompress(
        pBufferOut,
        pBufferIn,
        pBmi->biWidth,
        abs(pBmi->biHeight),
        DIBWIDTHBYTES(*pBmi) );

    KASSERT((long)cbDest <= pDest->GetSize());
    pDest->SetActualDataLength(cbDest);
    pDest->SetSyncPoint(TRUE);

    return S_OK;
}
```

Nyní ještě zbývá přidat podporu pro funkcionalitu COM, což je nezbytné pro zabalení filtru do knihovny DLL a její vystavení pro další aplikace. Jediný skutečně nutný krok je přidání statické metody třídy s názvem `CreateInstance` (může se jmenovat libovolně). Odkaz na ni se pak vloží do globálního pole `g_Templates` typu `CFactoryTemplate`.

```
CUnknown * WINAPI CMULDec::CreateInstance(LPUNKNOWN pUnk, HRESULT *pHr)
{
    CMULDec *pFilter = new CMULDec();
    if(pFilter == NULL)
        *pHr = E_OUTOFMEMORY;
    return pFilter;
}

CFactoryTemplate g_Templates[] = {{
    _T(MUL1_NAME),
    &CLSID_MUL1,
    CMULDec::CreateInstance,
    NULL,
    &sudDBVDecoder
}};
```

Nakonec bude nutné implementovat funkce pro registraci filtru (přidají se `DllRegisterServer` a `DllUnregisterServer`). Knihovnu je třeba sestavovat (linkovat) se statickými knihovnami `strmbasd.lib`, `msvcrtd.lib` a `winmm.lib` (pro sestavení Debug). Instalace dekodéru do systému (je třeba administrátorských oprávnění) se provede příkazem `regsvr32 muldec.dll`, odinstalace pak `regsvr32 /u muldec.dll`. Výsledek lze spatřit na obrázku 7.4.

Ve výkladu bylo vynecháno mnoho podstatných detailů. Kompletní zdrojové kódy filtru lze stáhnou na stránkách předmětu.



Obrázek 7.4: Graf filtrů, na jehož vstupu se nachází video ve formátu z kapitoly 2.6. Následuje větvení komprimované video stopy. První větev je dekódována kodekem pro Video for Windows (VCM), druhá právě vytvořeným dekodérem pro DirectShow. Výstup by měl být totožný. Je zde patrný rozdíl proti VfW, který neumožňoval automatickou konverzi barevných modelů (formátů pixelu). DirectShow toho řeší zařazením filtru Color Space Converter.

7.2.4 Přehled API

Struktury

`AM_MEDIA_TYPE`
: Popisuje formát mediálních dat; např. pro `FORMAT_VideoInfo` nese strukturu `VIDEOINFOHEADER` obalující `BITMAPINFOHEADER`.

`VIDEOINFOHEADER`
: Hlavička snímku (obaluje `BITMAPINFOHEADER`).

`BITMAPINFOHEADER`
: Hlavička obrázku.

`ALLOCATOR_PROPERTIES`
: Popisuje alokátor, tj. počet a velikost bufferů.

Funkce

`DllMain`
: Vstupní bod do dynamicky sestavované knihovny (DLL). V implementaci základních tříd DirectShow se jmenuje `DllEntryPoint`.

`DllRegisterServer`
: Komponenta by se měla zaregistrovat do systému.

`AMovieDllRegisterServer2`
: Registruje a odregistruje DirectShow filtry.

`DllUnregisterServer`
: Opak k funkci `DllRegisterServer`.

`FreeMediaType`
: Uvolní strukturu informací o formátu dat uvnitř struktury `AM_MEDIA_TYPE`.

Třídy a rozhraní

`FOURCCMap`
: Třída umožňující převod mezi identifikátory GUID a FourCC.

`IMediaSample`
: Rozhraní objektu COM, přes který lze číst a nastavovat vlastnosti mediálních snímků (buffery).

`IMemAllocator`
: Rozhraní pro alokaci mediálních snímků (přesouvání dat mezi piny).

`CTransformFilter`
: Třída pro implementaci transformačních filtrů.

`CTransInPlaceFilter`
: Transformační filtry pracující s daty v místě, tzv. *in-place*.

`CMediaType`
: Obálka nad strukturou `AM_MEDIA_TYPE`, která popisuje formát mediálního snímku.

`CBaseFilter`
: Obecná abstraktní třída pro implementaci transformačních filtrů.

`CTransformInputPin` a `CTransformOutputPin`
: Reprezentuje vstupní resp. výstupní pin filtru.

7.3 FFmpeg

FFmpeg[3] je jeden z nejvýznamnějších frameworků unixového světa. Jedná se vlastně o několik knihoven, z nichž nejdůležitější jsou libavcodec (audio a video kodeky) a libavformat (kontejnery). Využívají jej např. MPlayer, VLC media player, xine, Avidemux, ffdshow a další. Jedná se o nízkoúrovňový framework s rozhraním pro jazyk C. Součástí balíku FFmpeg jsou (mimo vlastních knihoven) i konzolové nástroje pro snadné použití z příkazové řádky. Knihovny i nástroje jsou krátce popsány níže. V roce 2011 se část vývojářů od projektu FFmpeg odtrhla a vytvořila nový projekt s názvem Libav.[4] Pro frameworky VfW a DShow existuje obálka nad knihovnami FFmpeg s názvem ffdshow.

[3] `http://ffmpeg.org/`
[4] `http://libav.org/`

Nástroje

`ffmpeg`
: Slouží pro překódování multimediálních souborů.

`ffserver`
: Streamovací server.

`ffplay`
: Jednoduchý přehrávač založený na SDL.

`ffprobe`
: Nástroj, který zobrazí informace o multimediálních souborech.

Knihovny

`libavutil`
: Obsahuje pomocné funkce (generátor náhodných čísel, logování zpráv, ošetření chyb), funkce pro konverzi barevných prostorů, datové struktury, matematické konstatny a funkce (zaokrouhlování, logaritmus, převody číselných typů, racionální čísla), definice formátů pixelu, typů snímků, typů datových proudů atd.

`libavcodec`
: Obsahuje kodéry a dekodéry pro audio a video formáty.

`libavformat`
: Obsahuje muxery a demuxery pro různé kontejnerové formáty.

`libavdevice`
: Poskytuje napojení na různé multimediální frameworky (Video4Linux2, Video for Windows, DirectShow, ALSA, OSS, PulseAudio).

`libavfilter`
: Filtry (split, crop, ass, hflip, frei0r, unsharp), grafy filtrů.

`libswscale`
: Rychlá změna rozlišení a převod barevných modelů.

`libswresample`
: Rychlá změna vzorkovací frekvence a formátu audia.

7.3.1 Základy

Následující příkaz zobrazí podporované kontejnerové formáty. Příznak „D“ značí demuxer (splitter) a „E“ značí muxer.

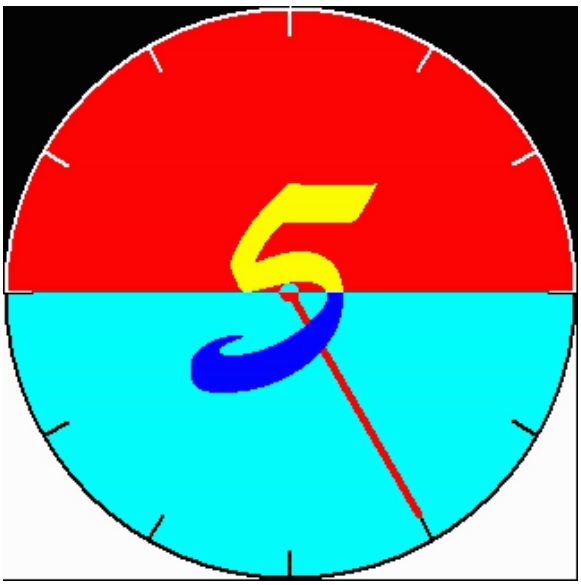
```
ffmpeg -formats
```

Analogicky lze pomocí volby `-codecs` zobrazit dostupné kodeky („D“ zde značí dekodér, „E“ kodér), `-filters` filtry, `-pix_fmts` formáty pixelu nebo `-protocols` protokoly (např. RTMP).

Nástroj `ffplay` slouží k rychlému přehrání videa. Následující příkaz zobrazí pomocí knihovny SDL okno a přehraje v něm žádané video.

```
ffplay clock.avi
```

Přidáním volby `-vf vflip` se na video aplikuje filtr, který každý snímek vertikálně překlopí. Filtrům lze předávat parametry, např. `-vf crop=256:256:0:0` použije filtr pro ořezání videa, kde výstupní (ořezané) video bude mít rozměry 256×256 pixelů a ořez bude počínat na souřadnicích $(0, 0)$ původního (vstupního) videa. Z filtrů je také možné sestavit graf. Prosté zřetězení (spojení) filtrů se provede pomocí čárky: `-vf "transpose, negate"` – video nejprve transponuje, poté neguje barvy. Takto zřetězeným filtrům se říká řetězec filtrů.



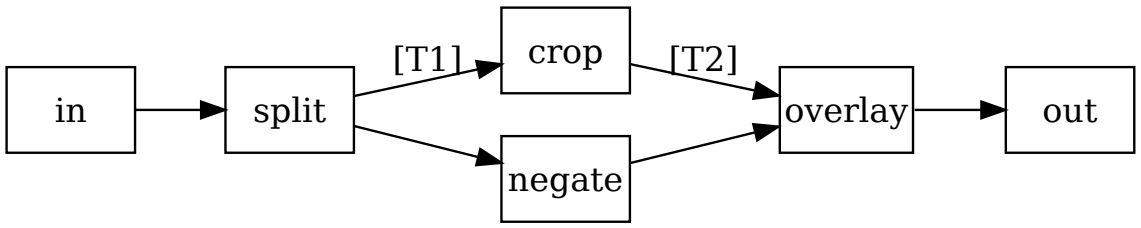
Obrázek 7.5: Výstup grafu (obr. 7.6). Snímek je v jedné větvi negován. Jeho spodní polovina je pak překryta druhou ořezanou větví.

Z jednotlivých řetězců (větví grafu) lze sestavit graf filtrů. Jednotlivé řetězce se od sebe oddělují pomocí středníku. Některé filtry mají několik vstupů a výstupů, ty se anglicky označují jako *pad* (podložka). Pokud je třeba sestavit větvený graf filtrů, je třeba tyto pady pojmenovat. Názvy padů se uzavírají do hranatých závorek a zapisují se před (vstupní) a za (výstupní) název filtru. Není třeba pojmenovat všechny pady. Nepojmenované jsou propojovány automaticky podle následujícího pravidla. První nepojmenovaný výstupní pad je připojen na první nepojmenovaný vstupní pad následujícího filtru řetězce. Například filtr `split` (větvení videa) má jeden vstup a dva výstupy, filtr `overlay` (překryv/spojení videa) má naopak dva vstupy a jeden výstup. Řetězec `nullsrc, split [L1], [L2] overlay, nullsink` tedy nejprve rozvětví vstup na dva výstupy (první s názvem „L1“ a druhý nepojmenovaný), pak překryje video ze dvou vstupů (první s názvem „L2“, druhý nepojmenovaný). Po automatickém propojení bude druhý nepojmenovaný výstup filtru `split` napojen na druhý nepojmenovaný vstup filtru `overlay`. Graf filtrů je platný (validní) pouze v případě, že jsou všechny pady (vstupy a výstupy) někam napojeny.

Následující příkaz otevře video, rozvětví ho, v první větvi provede ořez horní poloviny (zůstane spodní), ve druhé větvi provede negaci a překryv s ořezanou spodní polovinou.

```
ffplay -vf "[in]␣split␣[T1],␣negate,␣[T2]␣overlay=0:H/2␣[out];␣[T1]␣
    crop=iw:ih/2:0:ih/2␣[T2]" clock.avi
```

Lépe vše osvětluje graf na obrázku 7.6 získaný pomocí nástroje `graph2dot`.



Obrázek 7.6: Graf filtrů FFmpegu, který provede rozvětvení, úpravu a opětovné sloučení videa.

Přepínačem `-f` lze udat vstupní zařízení (např. `alsa`, `oss`, `video4linux`). Zobrazení vstupu z webkamery se provede následovně.

```
ffplay -f video4linux2 /dev/video0
```

K zobrazení informací o videu (datové toky, formáty audia/videa) slouží příkaz `ffprobe`. Jako parametr postačí název souboru.

```
ffprobe clock.avi
```

Uvedený příkaz zobrazí výstup podobný následujícímu.

```
Input #0, avi, from 'clock.avi':
  Duration: 00:00:12.00, start: 0.000000, bitrate: 55 kb/s
    Stream #0:0: Video: msrle ([1][0][0][0] / 0x0001), pal8, 321x321, 1
        fps, 1 tbr, 1 tbn, 1 tbc
    Stream #0:1: Audio: truespeech ("[0][0][0] / 0x0022), 8000 Hz, 1
       channels, s16, 8 kb/s
```

K překódování videa slouží příkaz `ffmpeg`. Syntaxe volání je následující.

```
ffmpeg [globalni volby] [[volby pro vstup] [-i vstup]]... {[volby pro
   vystup] vystup}...
```

Například triviální použití pro konverzi do kontejnerového formátu Matroska s videokodekem FFV1 a audiokodekem FLAC bude vypadat takto.

```
ffmpeg -i clock.avi -c:v ffv1 -c:a flac output.mkv
```

Pro výstupní video je možné nastavit různé parametry. Zde bude mít výstup datový tok videa 64 kbit/s.

```
ffmpeg -i input.avi -b:v 64k output.avi
```

Záznam videa z webkamery a zvuku ze zařízení OSS lze provést následovně.

```
ffmpeg -f oss -i /dev/dsp -f video4linux2 -i /dev/video0 output.mpg
```

7.3.2 Přehrávač

V následující sekci bude vysvětlena základní práce s kontejnerem a tvorba triviálního přehrávače videa. Pro práci s videem byl použit framework FFmpeg verze 0.10. Dokumentaci k veškerému API lze nalézt na oficiálních stránkách frameworku FFmpeg.[5]

Pro použití API frameworku je nutné překladač informovat, kde leží příslušné hlavičkové soubory a knihovny. V systémech unixového typu k tomu lze využít nástroj `pkg-config`. Volání překladače by mohlo vypadat zhruba následovně.

```
cc -I/usr/include/libavformat app.c -lavformat -o app
```

Rozhraní knihoven FFmpeg je v čistém C. Pokud chceme volat funkce z C++, je třeba obalit příslušné `#include` konstrukcí `extern "C"`. K použití funkcí z jedné z knihoven frameworku FFmpeg, např. libavformat, je třeba vložit (includovat) příslušný hlavičkový soubor. Názvy hlavičkových souboru jsou shodné s názvy knihoven (bez prefixu „lib“), např. `#include <avformat.h>`.

Prvním krokem po spuštění aplikace by mělo být volání funkce `av_register_all`, která zaregistruje všechny dostupné muxery, demuxery, kodeky a protokoly. Následně se voláním funkce `avformat_open_input` otevře vstupní tok (kontejner, např. soubor AVI). Funkce vytvoří strukturu `AVFormatContext`, která zaobaluje informace o multimediálním kontejneru. Nyní je možné načíst z kontejneru základní informace (délka trvání, datový tok). To provede funkce `avformat_find_stream_info`. Zavoláním `av_dump_format` se na standardní chybový výstup vypíše několik základních informací o kontejneru a v něm obsažených datových tocích (stopách). Pro korektní uvolnění prostředků je nakonec nutno zavolat `avformat_close_input`. Celý příklad 7.3 je možno vidět dále.

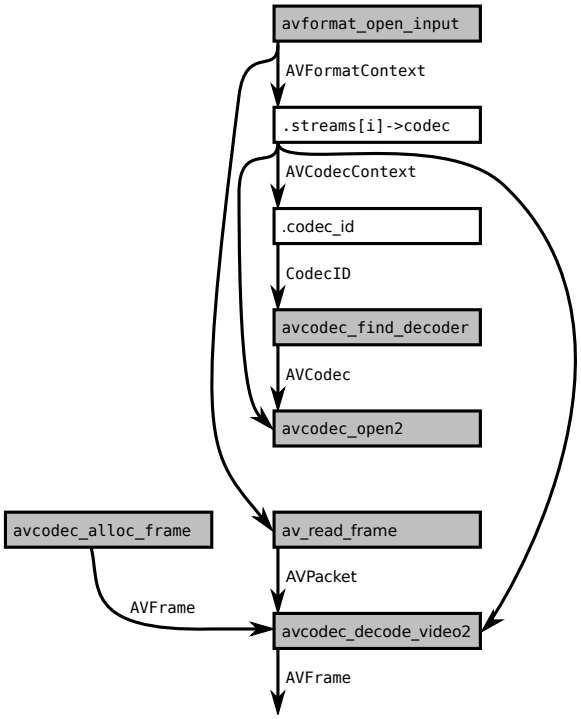
```
#include <avformat.h>

int main(int argc, const char *argv[])
{
    av_register_all();

    const char *filename = argc>1 ? argv[1] : "clock.avi";

    av_log(NULL, AV_LOG_INFO, "Opening␣%s...\n", filename);

    AVFormatContext *pFormatCtx = NULL;

    if( avformat_open_input(&pFormatCtx, filename, NULL, NULL) )
        abort();

    if( avformat_find_stream_info(pFormatCtx, NULL) < 0 )
        abort();
```

[5] http://ffmpeg.org/

```
    av_dump_format(pFormatCtx, 0, filename, 0);

    avformat_close_input(&pFormatCtx);

    return 0;
}
```

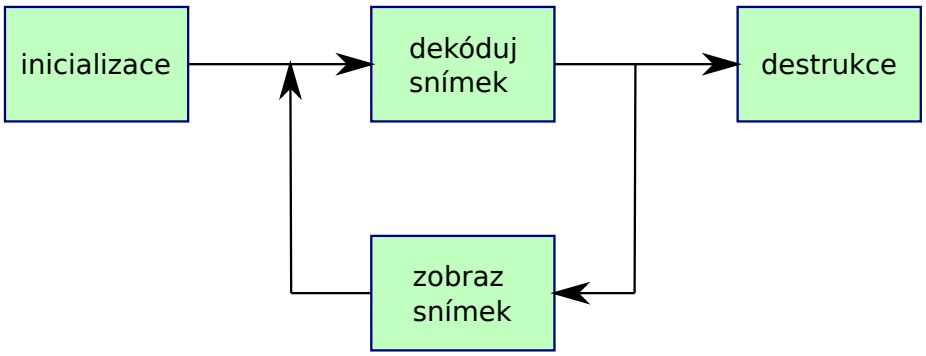
Příklad 7.3: Jednoduchý program, který otevře vstupní soubor a vypíše o něm základní informace.

Další text se věnuje pouze extrakci a zobrazení videosnímků. Pro přehrávání není volání `avformat_find_stream_info` a `av_dump_format` nezbytné. Po volání `avformat_open_input` jsou ve struktuře `AVFormatContext` k dispozici informace o datových tocích (stopách) uvnitř kontejneru. Jednotlivé stopy lze procházet pomocí pole `streams` této struktury. Každá stopa je zde typu `AVStream`. Uvnitř této struktury je důležitá především položka `codec`, která ukazuje na strukturu typu `AVCodecContext`. `AVCodecContext` je obálka informací pro kodek, nikoli však vlastní kodek. Z této struktury je nyní podstatná položka `codec_type`, pomocí které je možno rozlišit audio (`AVMEDIA_TYPE_AUDIO`) a video (`AVMEDIA_TYPE_VIDEO`) stopy. V předchozím příkladu 7.3 je vstupní datový tok (kontejner) přístupný přes `pFormatCtx`. Videostopu je možné identifikovat postupným procházením všech stop uvnitř tohoto kontejneru. Namísto ručního procházení lze použít také funkci `av_find_best_stream`.



Obrázek 7.7: Graf volání funkcí a jejich závislosti při dekompresi videa. Funkce `av_read_frame` a `avcodec_decode_video2` se volají opakovaně pro každý snímek.

Nyní je třeba pro informace uvnitř struktury `AVCodecContext` nalézt odpovídající dekodér, to provede `avcodec_find_decoder`. V případě úspěchu je možno přistoupit k inicializaci dekodéru voláním `avcodec_open2`. Následně lze přistoupit k vlastnímu dekódování snímků. Snímek reprezentuje struktura `AVFrame`. Pro právě dekódovaný snímek je potřeba tuto strukturu alokovat, to provede `avcodec_alloc_frame`. Dále je potřeba struktury `AVPacket`, do které budou



Obrázek 7.8: Jádrem přehrávače je smyčka, ve které se dekódují jednotlivé snímky. Před jejich dekódováním je nutné alokovat prostředky a inicializovat kodek. Po skončení přehrávání je možné prostředky uvolnit.

z videostopy načtena surová (zkomprimovaná) data odpovídající jednomu snímku. Vlastní dekódování bude probíhat ve smyčce (dokud bude co přehrávat). V každé iteraci smyčky se pomocí `av_read_frame` načte jeden paket (`AVPacket`). Ten se vzápětí pomocí `avcodec_decode_video2` dekóduje do struktury `AVFrame`. Odtud je možné jej vykreslit na obrazovku (nebo jinak zpracovat). Kritická část kódu je v příkladu 7.4 níže.

```
AVPacket pkt;

while( av_read_frame(pFormatCtx, &pkt) == 0 )
{
    if( pkt.stream_index == videoStream)
    {
        int frameFinished = 0;
        if( avcodec_decode_video2(pCodecCtx, pFrame, &frameFinished, &
            pkt) < 0 )
            abort();
        if(frameFinished)
        {
            // ...
        }
    }
    av_free_packet(&pkt);
}
```

Příklad 7.4: Smyčka dekódující snímky u triviálního přehrávače video snímků. Po dekódování je možno provést zpracování (zobrazení) – naznačeno tečkami.

Protože dekódovaný snímek pravděpodobně nebude ve formátu, který je možno přímo odeslat do renderovacího zařízení, je nutné provést konverzi jeho formátu. Vhodným formátem může být snímek s pixely v BGR24, což udává v případě FFmpegu položka `PIX_FMT_BGR24` výčtu `PixelFormat`. Konverzi provedeme ihned po

dekódování snímku voláním `sws_scale` z knihovny libswscale. Použití je v příkladu 7.5.

```
Mat img(pCodecCtx->height, pCodecCtx->width, CV_8UC3,
    (void *)pFrameRGB->data[0], pFrameRGB->linesize[0]);
struct SwsContext *c = NULL;

// ...

c = sws_getCachedContext(c, pCodecCtx->width, pCodecCtx->height,
    (PixelFormat)pFrame->format, pCodecCtx->width, pCodecCtx->height,
    PIX_FMT_BGR24, SWS_BICUBIC, NULL, NULL, NULL);

sws_scale(c, (const uint8_t * const*)pFrame->data, pFrame->linesize,
    0, pFrame->height, pFrameRGB->data, pFrameRGB->linesize);

imshow("image", img);
```

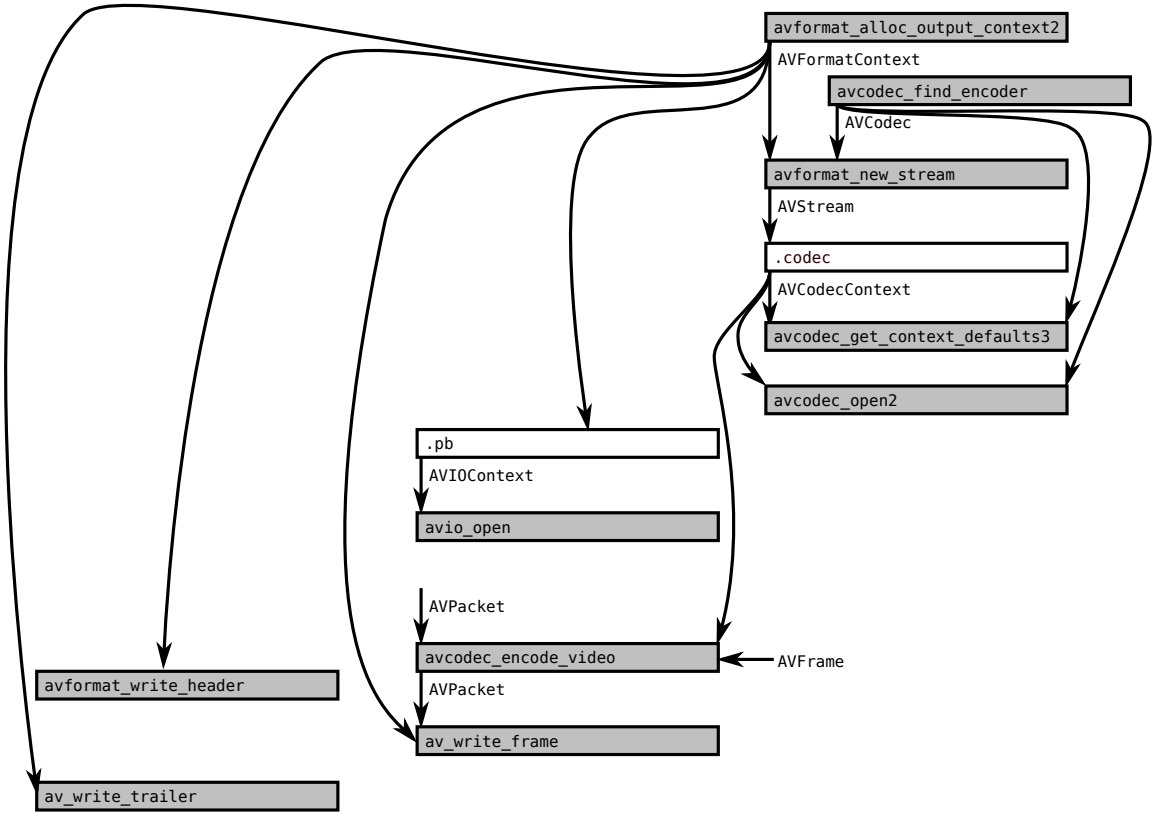
Příklad 7.5: Převod snímku na RGB a zobrazení pomocí OpenCV. Dekódovaný snímek je uložen v `pFrame`, snímek v BGR24 bude uložen do `pFrameRGB`. OpenCV předpokládá pixely v pořadí BGR, s typem `CV_8UC3` to bude přímo formát BGR24.

Kompletní zdrojové kódy uvedených dvou příkladů lze nalézt na stránkách předmětu.

7.3.3 Komprese videa

Pokud bude nutno dekomprimovaný snímek opět zkomprimovat a uložit do kontejneru, lze postupovat následovně. Nejprve je třeba pomocí volání `avformat_alloc_output_context2` alokovat novou strukturu `AVFormatContext`, která bude reprezentovat výstupní kontejner (např. soubor AVI). Požadovaný videokodek se vybere z výčtu `CodecID` (např. `CODEC_ID_FFV1`). Pomocí volání `avcodec_find_encoder` se z něj vytvoří struktura `AVCodec`. Voláním `avformat_new_stream` se v kontejneru (alokovaná struktura `AVFormatContext`) vytvoří nová stopa komprimovaná vybraným kodekem (vytvořený `AVCodec`). Tímto krokem vznikne struktura `AVStream`, která mimo jiné obsahuje položku `.codec`. To je struktura `AVCodecContext`, která umožňuje nastavovat parametry komprese. Tato struktura se nejprve vyplní pro daný kodek výchozími hodnotami (voláním `avcodec_get_context_defaults3`). Poté lze některé parametry kodeku upravit (rozměry videa, formát pixelu). Po tomto kroku je již možné kodek otevřít a inicializovat jeho kontextem (vyplněná `AVCodecContext`) pomocí funkce `avcodec_open2`. Správnost popsaných operací lze ověřit výpisem pomocí `av_dump_format`. Výstup bude podobný následujícímu úseku.

```
Output #0, avi, to 'output.avi':
    Stream #0:0: Video: ffv1, yuv444p, 321x321, q=2-31, 200 kb/s, 90k
        tbn, 1 tbc
```



Obrázek 7.9: Graf volání funkcí a jejich závislosti při kompresi videa. Funkce `avcodec_encode_video` a `av_write_frame` se volají opakovaně pro každý snímek.

Pokud je vše v pořádku, lze přistoupit ke skutečnému vytvoření kontejneru (otevření souboru AVI). K tomu slouží funkce `avio_open`, která jako parametr obdrží ukazatel na položku `.pb` struktury `AVFormatContext` a jméno výstupního souboru. Před uložením snímků je třeba zapsat hlavičku stopy pomocí `avformat_write_header`. Stejně tak je třeba zavolat po uložení všech potřebných snímků funkci `av_write_trailer`. Mezi těmito voláními lze ukládat jednotlivé komprimované snímky. K tomu slouží funkce `av_write_frame`, která jako parametr očekává již připravený paket (`AVPacket`). Ke kompresi snímku (`AVFrame`) do paketu slouží funkce `avcodec_encode_video`. Ta komprimuje dříve zvoleným kodekem a proto je předtím nutné snímek převést do formátu, kterému tento kodek rozumí. Pro kodek FFV1 to bude např. formát `PIX_FMT_YUV444P`. Konverzi provede již v předchozí sekci zmiňovaná funkce `sws_scale`.

7.3.4 Videokodek

Tato sekce popisuje napojení jednoduchého kodeku z kapitoly 2.6 do frameworku FFmpeg. Zmíněný kodek pracuje s obrazem, který je v paměti orientován „vzhůru

nohama.“ Pro jeho obrácení bez skutečného kopírování dat může být užitečná následující funkce. Trik spočívá v nastavení ukazatele na obraz na jeho poslední řádek a udání záporné velikosti kroku mezi řádky.

```
static void flip(AVCodecContext *avctx, AVPicture *picture)
{
    picture->data[0] += picture->linesize[0] * (avctx->height-1);
    picture->linesize[0] *= -1;
}
```

Přidání kodeku (zde pouze dekodéru) do frameworku FFmpeg znamená znovupřeložení všech změnou dotčených částí, což budou knihovny libavcodec a libavformat. Do knihovny libavcodec se přidá vlastní dekodér. Do knihovny libavformat se přidá pouze mapování nového FourCC kódu na nově přidaný kodek. V případě systémů unixového typu je k aktualizaci knihoven nainstalovaných v systému třeba administrátorských oprávnění. Nic ale uživateli nebrání v přeložení a instalaci frameworku FFmpeg do jeho domovského adresáře. Správnou funkci dekodéru lze rychle ověřit voláním nástroje `ffplay` na předpřipravené video.

Další text předpokládá zdrojové kódy frameworku v aktuálním adresáři. Do podadresáře `libavcodec` se nejprve zkopírují zdrojové kódy z kapitoly 2.6, tedy `mulcodec.h` a `mulcodec.c`. Dále se v `libavcodec` vytvoří soubor `mul1.c`, který bude sloužit jako obálka přidávaného kodeku pro FFmpeg. Tento soubor bude obsahovat pouze jednu datovou strukturu typu `AVCodec`, která se odkáže na lokální funkce obalující skutečná volání kodeku.

```
AVCodec ff_mul1_decoder =
{
    .name           = "mul1",
    .long_name      = NULL_IF_CONFIG_SMALL("MUL video codec using RLE"),
    .type           = AVMEDIA_TYPE_VIDEO,
    .id             = CODEC_ID_MUL1,
    .capabilities   = CODEC_CAP_LOSSLESS | CODEC_CAP_DR1,
    .init           = mul1_decode_init,
    .decode         = mul1_decode_frame,
    .close          = mul1_decode_close,
};
```

V položce `.type` je nezbytné dát najevo, že se jedná o kodek videa. Položka `.id` je klíčová, udává totiž unikátní identifikátor kodeku uvnitř frameworku FFmpeg. Do `.capabilities` se nastaví relevantní `CODEC_CAP_*`. Hodnota `CODEC_CAP_LOSSLESS` zde říká, že kodek je bezeztrátový. Hodnota `CODEC_CAP_DR1` umožňuje kodeku použít funkci `get_buffer`. Do `.init`, `.decode` a `.close` se nastaví ukazatele na lokální funkce, které je třeba dále implementovat.

Ve stejném souboru se vytvoří podle prototypů v deklaraci `AVCodec` odkazované lokální funkce. Nejpodstatnější z nich je funkce pro dekódování snímku. Ta v podstatě pouze obaluje volání skutečné funkce `decompress` přidávaného kodeku. Kodek na

rozdíl od frameworku FFmpeg očekává obrácenou orientaci snímku (řádky odspodu nahoru), což je typické pro systém Microsoft Windows. Tento problém lze obejít jednoduchým trikem, kdy se nastaví ukazatel na snímek až na jeho poslední řádek a udá se k němu záporný krok (*stride*). Alternativně lze volání `decompress` obalit voláním výše uvedené funkce `flip`. V úseku kódu níže je ještě vynechána alokace snímku, který by se měl nakonec z paměti také uvolnit. Funkce vrací počet skutečně použitých bajtů.

```
static int mul1_decode_frame(AVCodecContext *avctx,
    void *outdata, int *outdata_size, AVPacket *avpkt)
{
    AVFrame *picture = outdata;
    *outdata_size = sizeof(AVFrame);

    // ...

    return decompress(
        picture->data[0] + picture->linesize[0] * (avctx->height-1),
        avpkt->data,
        avctx->width,
        avctx->height,
        picture->linesize[0] * -1
    );
}
```

Identifikátor `CODEC_ID_MUL1` se přidá do výčtu CodecID v `libavcodec/avcodec.h`. Přitom je třeba přidat jej nakonec (jinak se rozbije binární kompatibilita knihoven) a s hodnotou, která jako řetězec ASCII znaků připomíná název kodeku (pro zamezení konfliktů s jinými nově přidanými kodeky).

```
CODEC_ID_MUL1       = MKBETAG('M','U','L','1'),
```

Nyní ještě zbývá framework o kodeku informovat tak, aby byl pro odpovídající tok videa automaticky zavolán. K tomu je třeba dekodér nejprve zaregistrovat. Toho se docílí modifikací funkce `avcodec_register_all` v `libavcodec/allcodecs.c`. Uvnitř této funkce se na vhodné místo přidá následující řádek.

```
REGISTER_DECODER (MUL1, mul1);
```

Dále se upraví tabulka mapování FourCC kódů (ve formátu RIFF) na výše zavedené ID kodeku, tj. `ff_codec_bmp_tags` v `libavformat/riff.c`. Před konec se přidají následující řádky.

```
{ CODEC_ID_MUL1,         MKTAG('M', 'U', 'L', '1') },
{ CODEC_ID_MUL1,         MKTAG('m', 'u', 'l', '1') },
```

Aby se kodek přeložil, je nutné modifikovat `Makefile` v adresáři `libavcodec`. Přidá se zde následující závislost.

```
OBJS-$(CONFIG_MUL1_DECODER)            += mul1.o mulcodec.o
```

Nyní je možné FFmpeg zkonfigurovat a přeložit. Jako parametry skriptu `configure` lze předat požadované volby (pro přehled `-help`), následně se zavolá `make`.

7.3.5 Přehled API

Výčty

`AVMediaType` (avutil)
: Typ datového toku neboli stopy (audio, video, titulky); např. `AVMEDIA_TYPE_VIDEO` nebo `AVMEDIA_TYPE_AUDIO`.

`PixelFormat` (avutil)
: Formát pixelu; např. `PIX_FMT_RGB24` nebo `PIX_FMT_YUV444P`.

`CodecID` (avcodec)
: Unikátní identifikátor kodeku (kodéru nebo dekodéru) vnitř frameworku; např. `CODEC_ID_MSRLE` nebo `CODEC_ID_FFV1`.

Struktury

`AVFormatContext` (avformat)
: Zaobaluje multimediální kontejner.

`AVStream` (avformat)
: Reprezentuje datový tok uvnitř multimediálního kontejneru.

`AVCodecContext` (avcodec)
: Nese kontext kodeku neboli informace pro kodek (rozměry videa, ID kodeku), ne však vlastní kodek.

`AVCodec` (avcodec)
: Vlastní kodek (kodér, dekodér).

`AVFrame` (avcodec)
: Zaobaluje snímek audia nebo videa.

`AVPicture` (avcodec)
: Zaobaluje snímek videa. Obsahuje pouze obrazová data. Strukturu `AVFrame` lze přetypovat na `AVPicture`.

`AVPacket` (avcodec)
: Obaluje surová data přečtená z (k zapsání do) dané stopy (audio nebo video stopa) ve vstupním kontejneru. Při dekompresi je tato data třeba dále dekódovat. Při kompresi se naopak přímo zapíší do kontejneru.

`SwsContext` (swscale)
: Nese informace a data nutná ke změně velikosti a formátu pixelu snímku.

Funkce

`av_register_all` (avformat)
: Zaregistruje všechny kodeky, muxery, demuxery a protokoly.

`avformat_open_input` (avformat)
: Otevře vstupní kontejner (např. soubor) a přečte z něj hlavičku.

`avformat_find_stream_info` (avformat)
: Načte ze vstupního kontejneru různé informace. Užitečné např. pro následné vypsání funkcí `av_dump_format`.

`av_dump_format` (avformat)
: Zobrazí informace o kontejneru a dostupných stopách uvnitř něj (audio, video).

`avcodec_find_decoder` (avcodec)
: Podle ID kodeku najde odpovídající dekodér.

`avcodec_find_encoder` (avcodec)
: Podle ID kodeku (např. `CODEC_ID_FFV1`) vrátí požadovaný kodér.

`avcodec_open2` (avcodec)
: Inicializuje kontext kodeku (informace pro kodek) pro použití s daným kodekem.

`avcodec_close` (avcodec)
: Uvolní prostředky uvnitř kontextu kodeku alokované s funkcí `avcodec_open2`.

`avcodec_alloc_frame` (avcodec)
: Alokuje strukturu `AVFrame`, kterou je třeba dále uvolnit voláním funkce `av_free`.

`avpicture_get_size` (avcodec)
: Spočte velikost nutnou pro uložení obrazových dat o daných rozměrech a v daném formátu pixelu. Užitečné pro alokaci bufferu.

`avpicture_fill` (avcodec)
: Podle parametrů vyplní strukturu `AVPicture` (může být `AVFrame`). Parametr ptr ukazuje na oblast paměti, kam budou obrazová data v daném formátu pixelu skutečně uložena. S touto oblastí je pak možno dále pracovat (např. vykreslit na obrazovku).

`av_read_frame` (avformat)
: Přečte z kontejneru jeden paket. Pro video odpovídá paket jednomu snímku. Data z paketu musejí být dále dekódována. Alokovaný paket je pak třeba uvolnit voláním `av_free_packet`.

`avformat_write_header` (avformat)
: Zapíše do kontejneru hlavičku datové stopy.

`av_write_frame` (avformat)
: Zapíše do kontejneru paket.

`av_write_trailer` (avformat)
: Zapíše do kontejneru zakončení stopy.

`av_new_packet` (avcodec)

Alokuje nový paket.

`av_free_packet` (avcodec)
: Uvolní alokovaný paket.

`avcodec_decode_video2` (avcodec)
: Z paketu dekóduje jeden snímek videa.

`avcodec_encode_video` (avcodec)
: Zkomprimuje jeden snímek videa do předaného bufferu. Obrázek musí být ve formátu udaném `avctx.pix_fmt`.

`avformat_close_input` (avformat)
: Uvolní strukturu `AVFormatContext`.

`sws_getCachedContext` (swscale)
: Vytvoří strukturu pro změnu rozměrů a formátu pixelu. Zkusí využít již existující. Potřebné pro volání funkce `sws_scale`.

`sws_freeContext` (swscale)
: Uvolní alokovanou strukturu vytvořenou voláním `sws_freeContext`.

`sws_scale` (swscale)
: Změní rozměry a formát pixelu snímku.

`av_malloc` (avutil)
: Alokuje blok paměti.

`av_free` (avutil)
: Uvolní alokovanou paměť.

`av_log` (avutil)
: Loguje informace.

`avformat_alloc_output_context2` (avformat)
: Pro výstupní formát alokuje strukturu `AVFormatContext`.

`av_find_best_stream` (avformat)
: Vrátí z kontejneru požadovanou datovou stopu, např. video stopu. Alternativně lze všechny stopy procházet ručně.

`avformat_new_stream` (avformat)
: Přidá do kontejneru novou datovou stopu.

`avcodec_get_context_defaults3` (avcodec)
: Nastaví parametry v předaném `AVCodecContext` na výchozí hodnoty daného kodeku.

`avio_open` (avformat)
: Vytvoří a inicializuje strukturu potřebnou pro přístup k danému kontejneru.

`avio_close` (avformat)
: Uzavře a uvolní strukturu potřebnou pro přístup ke kontejneru.

7.4 GStreamer

GStreamer (gst) je další významný multiplatformní framework založený na grafu filtrů. Využívají jej (některé volitelně) např. přehrávače Amarok, Banshee, Gnash, Kaffeine, Listen, Songbird nebo IM klient Empathy. Je založen na objektovém systému GLib 2. Zásuvné moduly (pluginy) pro tento framework jsou dynamické knihovny a jsou načítány za běhu. Informace o těchto pluginech jsou cachovány v tzv. registru GStreameru (`~/.gstreamer-$GST_MAJORMINOR/registry-$ARCH.bin`). Pluginy jsou hledány mj. v adresáři specifikovaném proměnnou prostředí `GST_PLUGIN_PATH` a jsou načítány automaticky. Grafu filtrů se zde říká *pipeline*. Základními stavebními kameny pipeline jsou tzv. elementy (např. dekodér, kodér, demuxer, zdroj dat, video výstup). Elementy jsou obsaženy v pluginech. Plugin po svém načtení typicky zaregistruje několik elementů, které implementuje (např. kodér a dekodér jednoho formátu).

Elementy jsou jednoduše černé krabičky, které mohou mít několik vstupů i výstupů. Například do dekodéru audia tečou na vstupu komprimovaná data a na výstupu z něj vychází dekomprimovaný zvukový signál. Demuxer má na vstupu tok z kontejnerového formátu, na výstupu několik stop ukrytých uvnitř (audio, video). Zdrojové (*source*) elementy dodávají do pipeline tok tak (např. ze souboru na disku). Cílové (*sink*) elementy slouží jako koncové body pro toky dat (např. zvuková karta). Elementy se mezi sebou propojují přípojnými body, tzv. pady (z angl. *pad*). Pady mohou být vstupní (sink) nebo výstupní (source, src). Data tečou vždy ze zdrojového (src) do cílového (sink). Jaká data elementy přijímají na vstupu nebo jsou schopny produkovat na výstupu určují tzv. datové formáty neboli schopnosti padu (*capabilities*). Data jsou zde identifikována pomocí typů MIME (např. `audio/x-vorbis`, `audio/x-raw-float`). Na základě těchto schopností (datových formátů) je framework schopen propojovat elementy automaticky. Součástí frameworku jsou také nástroje pro příkazovou řádku – `gst-inspect` a `gst-launch`. Příkaz `gst-inspect` slouží k zjišťování informací o pluginech a elementech. Bez parametrů vypíše jejich přehled. Pokud se mu jako parametr předá název pluginu či elementu, vypíše o něm podrobné informace. Příkaz `gst-launch` slouží k sestavení a spuštění pipeline z příkazového řádku. Veškerou dokumentaci ke GStreameru lze nalézt na oficiálních stránkách.[6] V následujícím textu je všude použit GStreamer verze 0.10.

[6] `http://gstreamer.freedesktop.org/`

7.4.1 Základy

Příkaz `gst-inspect` bez parametrů vypíše dohromady seznam všech pluginů a elementů.

```
gst-inspect-0.10
```

Tento seznam je ve formátu – „plugin: element: popis elementu.“ Jeden z řádků výstupu může být například element `alsasrc` v pluginu `alsa`.

```
alsa:  alsasrc: Audio source (ALSA)
```

Bližší informace se získají předáním jména pluginu nebo elementu jako parametru příkazu. Následující příkaz vypíše informace o pluginu.

```
gst-inspect-0.10 alsa
```

Výstup budem podobný následujícímu (zkráceno).

```
Plugin Details:
  Name:                 alsa
  Description:          ALSA plugin library
  Filename:             /usr/lib64/gstreamer-0.10/libgstalsa.so
  Version:              0.10.35
  License:              LGPL

  alsasink: Audio sink (ALSA)
  alsasrc: Audio source (ALSA)
  alsamixer: Alsa mixer
```

Informace o elementu lze vypsat, předá-li se jako parametr jeho název.

```
gst-inspect-0.10 alsasrc
```

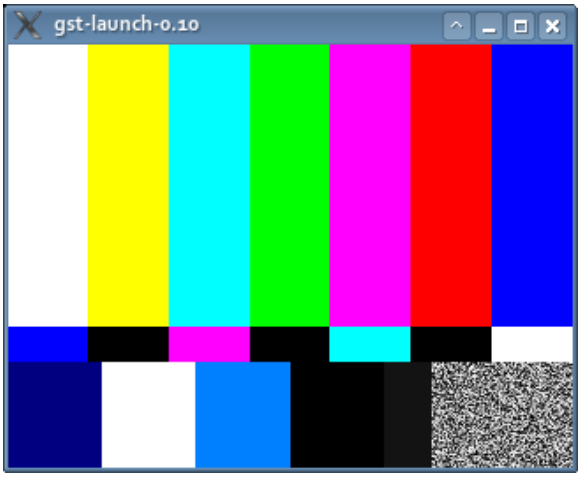
Příkaz vypíše následující (zkráceno).

```
Factory Details:
  Long name:    Audio source (ALSA)
  Class:        Source/Audio
  Description:  Read from a sound card via ALSA

Pad Templates:
  SRC template: 'src'
    Availability: Always
    Capabilities:
      audio/x-raw-int
             endianness: { 1234, 4321 }
                 signed: { true, false }
                  width: 32
                  depth: 32
                   rate: [ 1, 2147483647 ]
               channels: [ 1, 2147483647 ]

Element Properties:
```

```
  name                : The name of the object
                        flags: readable, writable
                        String. Default: null Current: "alsasrc0"
  device              : ALSA device, as defined in an asound
     configuration file
                        flags: readable, writable
                        String. Default: "default" Current: "default"
```

Ve výpisu se objeví popis a třída elementu (zde `Source/Audio`), výstupní (src) a vstupní (sink) pady a jejich formáty dat (zde `audio/x-raw-int` u zdrojového padu), vlastnosti (*properties*) elementu (zde `name` a `device`) apod.

Příkaz `gst-launch` slouží k sestavení a spuštění pipeline z příkazové řádky. Jako parametr obdrží textový řetězec popisující graf filtrů. Jednotlivé elementy jsou zde propojovány vykřičníkem. Vlastnosti elementů mohou být nastavovány za názvem elementu (ve formátu `vlastnost=hodnota`), oddělují se mezerou. Za jménem elementu lze pomocí tečky oddělit jméno padu, kterého se spojení týká. Mezi vykřičníky lze také zapsat datové formáty (*capabilities*), přes které se má spojení omezit (filtrovat). Schopnosti se skládají z typu MIME následovaného čárkou oddělovanými vlastnostmi. Jednotlivé datové formáty se od sebe oddělují středníky.



Obrázek 7.10: Výstup jednoduché pipeline sestávající z elementů `videotestsrc` a `ximagesink`.

Nejjednodušším příkladem bude pipeline sestávající z elementu `playbin`. Ve skutečnosti se nejedná o obyčejný element, ale o koš (angl. *bin*) elementů.

```
gst-launch-0.10 playbin uri=file:///tmp/clock.avi
```

Příkaz vytvoří pipeline, kterou ihned spustí. Spuštěním se otevře okno, ve kterém se zobrazuje přehrávané video. Do zvukového subsystému je přitom přehráván zvuk. Uvnitř elementu `playbin` byl automaticky podle přehrávaného souboru vytvořen vhodný graf filtrů.

Následující pipeline sestává z elementu `videotestsrc`, který generuje testovací obraz, a elementu `ximagesink`, který vytvoří okno (X Window System). Výsledek je zobrazen na obr. 7.10.

```
gst-launch-0.10 videotestsrc ! ximagesink
```

Následující příkaz sestaví pipeline z elementů `filesrc` (čte data ze souboru), `decodebin` (sestaví v sobě vhodný graf pro dekódování), `colorspace` (umožňuje

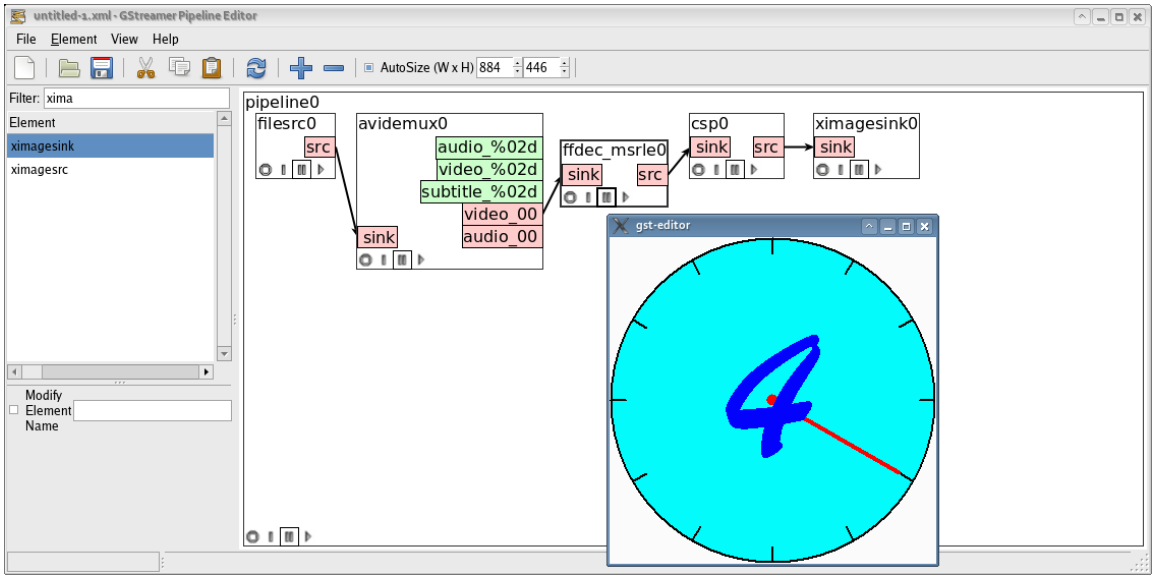
konvertovat barevné modely) a `ximagesink`. Výstup bude stejný jako v případě použití elementu `playbin`.

```
gst-launch-0.10 filesrc location=/tmp/clock.avi ! decodebin !
    colorspace ! ximagesink
```

Dalším krokem je ruční zařazení elementů demuxování kontejneru AVI a dekódování videostopy (zde formát Microsoft RLE). O demuxování se postará element `avidemux`. Pro dekódování byl použit element `ffdec_msrle` z balíčku *gst-ffmpeg-plugins*.

```
gst-launch-0.10 filesrc location=/tmp/clock-rle.avi ! avidemux !
    ffdec_msrle ! colorspace ! ximagesink
```

Pro snadnou vizuální stavbu a ovládání grafu filtrů slouží nástroj `gst-editor`.[7] Ukázku je možné vidět na obr. 7.11.



Obrázek 7.11: Nástroj `gst-editor` zobrazující graf filtrů pro přehrávání videostopy (formát Microsoft RLE) z kontejneru AVI.

7.4.2 Přehrávač

K překladu vlastní aplikace je vhodné využít nástroj `pkg-config`, který je schopen vrátit fragment příkazové řádky pro překlad (kompilaci) i sestavení (linkování) aplikace vůči daným knihovnám. K jednoduchým příkladům níže bude stačit použít pouze knihovnu `gstreamer-0.10`. Ve zdrojovém kódu je nejprve třeba vložit hlavičkový soubor `<gst/gst.h>`. Protože je GStreamer postaven nad knihovnou GLib, vloží se tímto také `<glib.h>`. Dále je tedy možné používat také funkce z GLib.[8] Než

[7] `http://code.google.com/p/gst-editor/`
[8] `http://developer.gnome.org/glib/stable/`

bude možné použít knihovny GStreamer, musí být na počátku funkce `main` zavolána funkce `gst_init`, která knihovnu inicializuje (načte registry). Níže je zdrojový kód triviální aplikace.

```
#include <gst/gst.h>

int main(int argc, char *argv[])
{
    gst_init(&argc, &argv);

    g_print("Started...\n");

    return 0;
}
```

Příklad 7.6: Aplikace, která inicializuje knihovnu GStreamer a pomocí funkce `g_print` z knihovny GLib vypíše na standardní výstup text.

Nejdůležitějším datovým typem GStreameru je pravděpodobně `GstElement`. Jedná se o abstraktní třídu, ze které vycházejí všechny elementy v pipeline. Dokonce celá pipeline, což je třída `GstPipeline`, dědí z `GstElement`. Další důležitou třídou poděděnou z `GstElement` je koš elementů `GstBin`. Samotný `GstElement` dědí v důsledku ze třídy `GObject` knihovny GLib, což mu mj. poskytuje funkcionalitu počítání referencí (a automatické uvolňování paměti) a čtení/nastavování vlastností. Každý element (`GstElement`) je vždy v jednom ze stavů `GST_STATE_VOID_PENDING` (žádný), `GST_STATE_NULL` (zastaven), `GST_STATE_READY` (připraven k pozastavení), `GST_STATE_PAUSED` (pozastaven) nebo `GST_STATE_PLAYING` (spuštěn). Stavy lze zjišťovat a nastavovat pomocí funkcí `gst_element_get_state` a `gst_element_set_state`. Význam je analogický stavům popsaným v sekci 7.2.1 o DirectShow.

Pro snazší orientaci a odkazování je možné elementy pojmenovat. Název elementu může být získán pomocí funkce `gst_element_get_name` a nastaven pomocí `gst_element_set_name`. Všechny elementy mají pady (src nebo sink, vysvětleno výše), což jsou objekty typu `GstPad`. Pomocí padů se elementy vzájemně propojují. Spoji pak tečou data obalená typem `GstBuffer`. Odkaz na existující pad může být získán pomocí funkce `gst_element_get_static_pad`. Nový pad může být vytvořen buď s funkcí `gst_element_request_pad` (ze šablony) nebo `gst_element_get_request_pad` (podle názvu).

Ke spojení dvou elementů skrze pady slouží `gst_element_link`. K propojení řetězce mnoha elementů je rychlejší použít `gst_element_link_many`. Pokud je nutné omezit datové formáty spojení (*capabilities*), je k dispozici funkce `gst_element_link_filtered`, která mezi parametry obdrží také odkaz na objekt třídy `GstCaps`, která mediální formáty (např. `video/x-raw-rgb`) popisuje. Pokud je třeba propojovat přímo jednotlivé pady elementů, jsou k dispozici ještě funkce `gst_element_link_pads` a `gst_element_link_pads_filtered`.

K vytváření instancí elementů slouží funkce `gst_element_factory_make`,

která jako parametr obdrží název šablony elementu (např. `videotestsrc` nebo `ximagesink`), ze které má být element instanciován. Druhým parametrem je požadovaný název elementu (může být `NULL` pokud má unikátní název vygenerovat GStreamer). Funkce `gst_element_factory_make` je vlastně spojením dvou funkcí – `gst_element_factory_find` (vrátí šablonu elementů) a `gst_element_factory_create` (vytváří instance ze šablony). Šablona je typu `GstElementFactory`.

Samotné elementy je výhodné umístit do pipeline (`GstPipeline`). Ta poskytuje funkci hodin a správu sběrnice zpráv. K vytvoření nové pipeline slouží funkce `gst_pipeline_new`.

Všechny vytvářené objekty je třeba uvolnit pomocí `gst_object_unref` nebo jiné specializované funkce. Zařazením elementů do pipeline budou tyto elementy uvolněny automaticky při destrukci samotné pipeline.

Následuje jednoduchý příklad, který zkonstruuje pipeline sestávající ze dvou propojených elementů `videotestsrc` a `ximagesink`.

```
int main(int argc, char *argv[])
{
    gst_init(&argc, &argv);

    GstElement *pipeline = gst_pipeline_new("pipeline");
    GstElement *source = gst_element_factory_make("videotestsrc",
        "source");
    GstElement *sink   = gst_element_factory_make("ximagesink",
        "sink");

    gst_bin_add_many(GST_BIN(pipeline), source, sink, NULL);
    gst_element_link(source, sink);
    gst_element_set_state(pipeline, GST_STATE_PLAYING);

    // ...

    gst_element_set_state(pipeline, GST_STATE_NULL);
    gst_object_unref(GST_OBJECT(pipeline));

    return 0;
}
```

Příklad 7.7: Aplikace, která vytvoří pipeline a vloží do ní dva elementy. Pipeline je následně spuštěna (přepnuta do stavu `GST_STATE_PLAYING`). Výpustkou je naznačeno čekání nebo smyčka zpráv. Následně se pipeline zastaví a uvolní. Ošetření chyb bylo pro stručnost vypuštěno.

Správu smyčky zpráv poskytuje knihovna GLib. Smyčka samotná je zastřešena datovým typem `GMainLoop`. O vytvoření nové smyčky se postará funkce `g_main_loop_new`, ke spuštění slouží `g_main_loop_run`, k ukončení `g_main_loop_quit` a k uvolnění prostředků `g_main_loop_unref`. Při přehrávání je možné použít tuto smyčku k čekání na zprávu o konci datového toku

(GST_MESSAGE_EOS). Následně se aplikace může ukončit. Zprávy je možné přijímat pomocí sběrnice zpráv, kterou poskytuje samotná pipeline (třída GstPipeline). Sběrnice je typu GstBus a vlastní zprávy jsou pak GstMessage. K získání odkazu na sběrnici lze využít funkci gst_pipeline_get_bus. K přidání vlastní callback funkce na sběrnici slouží gst_bus_add_watch. Callback funkce slouží k asynchronímu chytání zpráv ve vytvořené smyčce zpráv (výše).

Zkrácený příklad je uveden níže. Nezkrácenou verzi lze nalézt na stránkách předmětu.

```
static GMainLoop *loop;

static gboolean bus_callback(GstBus *bus, GstMessage *message,
    gpointer data)
{
    if(GST_MESSAGE_TYPE(message) == GST_MESSAGE_EOS)
        g_main_loop_quit(loop);
    return TRUE;
}

int main(int argc, char *argv[])
{
    gst_init(&argc, &argv);

    GstElement *pipeline = gst_pipeline_new("pipeline");
    GstBus *bus = gst_pipeline_get_bus(GST_PIPELINE(pipeline));
    gst_bus_add_watch(bus, bus_callback, NULL);
    gst_object_unref(bus);

    // ...

    loop = g_main_loop_new(NULL, FALSE);
    g_main_loop_run(loop);

    // ...
}
```

Příklad 7.8: Zkrácená aplikace, která vytvoří pipeline, přidá na její sběrnici callback funkci a odstartuje smyčku zpráv. Tímto způsobem lze počkat na konec datového toku a ukončit přehrávač.

7.4.3 Videokodek

Pro vytvoření prvního pluginu GStreameru je vhodné postupovat podle příručky „Plugin Writer's Guide“ v dokumentaci. Zde bude demonstrován plugin dekódující video ve formátu ze sekce 2.6. API pro tvorbu pluginů se od API pro tvorbu aplikací liší (je širší).

Pro vysvětlení je třeba nejdříve prohloubit znalosti z předchozích sekcí. Připojné body elementů, tzv. pady, mají vždy jeden ze dvou směrů – *source* nebo *sink*. Pad

směru sink (cílový) je uvnitř pluginu zdrojem dat. Do src (zdrojový pad) jsou pak elementem generována nová data. Z vnějšího podledu tečou data vždy ze src do sink padu. Pady mají dále tři dostupnosti (*availabilities*) – vždy (*always*), někdy (*sometimes*) a na požádání (*on request*). Pady s dostupností „vždy“ existují jednoduše vždy. Pady s dostupností „někdy“ jsou pluginem vytvořeny, pokud je k tomu důvod (např. ve vstupním kontejneru se objeví příslušná datová stopa). Pady s dostupností „na požádání“ jsou vytvořeny, pokud jsou požadovány (např. další výstup elementu „tee“, který větví vstupní datový tok). Formáty dat definují, jaká data mohou padem protékat. V případě dekodéru formátu ze sekce 2.6 to na vstupním padu bude nově zavedený typ MIME `video/x-mul`, na výstupním pak `video/x-raw-bgr`, k čemuž slouží makro `GST_VIDEO_CAPS_BGR`. Pro přidání nového typu MIME pro FourCC nalezený uvnitř kontejneru RIFF (AVI) je třeba upravit balíček *gst-plugins-base-libs*. Příslušný patch je k dispozici na stránkách předmětu.

Nejjednodušší cestou k implementaci dekodéru videa bude pravděpodobně specializace třídy `GstBaseVideoDecoder`, která dědí ze třídy `GstElement`.

Pady elementu se vytvářejí z tzv. šablon. K definici této šablony slouží makro `GST_STATIC_PAD_TEMPLATE`, které vytvoří objekt typu `GstStaticPadTemplate`. Dekodér bude mít jeden pad směru sink a jeden směru src.

```
static GstStaticPadTemplate sink_factory = GST_STATIC_PAD_TEMPLATE(
    "sink",
    GST_PAD_SINK,
    GST_PAD_ALWAYS,
    GST_STATIC_CAPS("video/x-mul"));

static GstStaticPadTemplate src_factory = GST_STATIC_PAD_TEMPLATE(
    "src",
    GST_PAD_SRC,
    GST_PAD_ALWAYS,
    GST_STATIC_CAPS(GST_VIDEO_CAPS_BGR));
```

K vlastní definici nového elementu slouží makro `GST_BOILERPLATE`. Jako parametry obdrží název nového objektu, prefix jeho funkcí a jméno rodičovské třídy (`GstBaseVideoDecoder`). Dále lze implementovat jednotlivé funkce.

Třída `GstBaseVideoDecoder` dovoluje překrýt následující virtuální metody.

`start` (volitelně)
: Volána na počátku zpracování dat. Vhodné místo pro alokaci prostředků.

`stop` (volitelně)
: Volána při ukončení zpracování. Vhodné místo pro uvolnění prostředků.

`scan_for_sync` (volitelně)
: Čeká na synchronizační bod mezi snímky.

`parse_data` (požadováno při vstupu nerozčleněném do paketů)
: Rozčleňuje příchozí data do snímků.

`set_format`
: Upozorňuje element na nový datový formát na vstupním padu.

`reset` (volitelně)
: Reset po změně pozice v datovém toku (*seek*).

`finish` (volitelně)
: Volána pro vyprodukování zbývajích dat (např. při konci datového toku).

`handle_frame`
: Dekódování jednoho snímku.

Funkce překrývající výše uvedené virtuální metody budou mít stejný postfix jako název překrývané funkce. Funkce s postfixem `_base_init` má za úkol přidat pady a a vyplnit detaily jako je název elementu, jeho třída (zde `Codec/Decoder/Video`) a popis. Funkce s postfixem `_init` inicializuje vlastní instanci elementu. Zde je důležité nastavit položku `packetized` třídy `GstBaseVideoDecoder` na `TRUE`, což zajistí zpracování vstupních dat rovnou na snímcích. Ke zpracování snímku slouží funkce s postfixem `_handle_frame`. Uvnitř je třeba nejprve alokovat pomocí `gst_base_video_decoder_alloc_src_frame` nový buffer pro výstupní (dekomprimovaný) snímek. Následuje volání funkce `decompress` ze sekce 2.6. Odeslání dat do výstupního padu nakonec zajistí `gst_base_video_decoder_finish_frame`. Pro volání funkce `decompress` je ještě třeba znát velikost kroku mezi řádky obrazu. K tomu slouží `gst_video_format_get_row_stride`. Poslední nutnou funkcí dekodéru je ta s postfixem `_start`. Zde je vhodný moment pro zjištění rozměrů obrazu ze vstupního padu. K tomu lze využít funkci `gst_pad_get_negotiated_caps`. Je však nutné ošetřit stav, kdy ještě není na vstupu napojen element dodávající data.

Dále je třeba zaobalit element dekodéru do pluginu. K tomu slouží makro `GST_PLUGIN_DEFINE`, kterému se předá odkaz na vstupní funkci pluginu. V té je nutné zavolat gst_element_register. Protože je třída `GstBaseVideoDecoder` z balíku *gst-plugins-bad-libs*, který obsahuje nestabilní API, není možné využít k práci s ním nástroj `pkg-config`. V případě Autotools je namísto toho nutné do příslušného `Makefile.am` přidat odkaz přímo na soubor pro libtool, např. `/usr/lib64/libgstbasevideo-0.10.la`.

7.4.4 Přehled API

Struktury

`GstObject`
: Základní třída objektové hierarchie GStreameru. Je vystavěna nad třídou `GObject` systému GLib. Přidává vlastnost „name“. Všechny ostatní třídy GStreameru (např. `GstElement`) z ní dědí.

`GstElement`
: Abstraktní třída pro všechny elementy, které mohou být vloženy do pi-

peline. Název elementu může být získán nebo nastaven pomocí funkcí `gst_element_get_name` a `gst_element_set_name`. Elementy obsahují přípojné body, tzn. pady. Každý element se nachází v některém ze stavů výčtu `GstState`.

`GstElementFactory`
: Třída reprezentující šablonu elementu (každý element je instanciován ze šablony). K vytváření instancí slouží funkce `gst_element_factory_find` a `gst_element_factory_create`, případně zastřešující funkce `gst_element_factory_make`.

`GstPad`
: Přípojný bod, pomocí kterého se elementy propojují. Každý pad má směr výčtu `GstPadDirection` (`GST_PAD_SRC` nebo `GST_PAD_SINK`), formáty dat (`GstCaps`) a popisující šablonu (`GstPadTemplate`), ze které je vytvářen. Pro propojování slouží funkce `gst_pad_link`, `gst_element_link`; pro vytváření ze šablony pak `gst_pad_new_from_template`.

`GstCaps`
: Struktura popisující formáty dat, které mohou pady protékat. Sestávají z pole struktur `GstStructure`. K vytváření slouží `gst_caps_new_simple`.

`GstStructure`
: Obecná struktura obsahující názvy a hodnoty.

`GstBin`
: Třída pro elementy, které mohou obsahovat další elementy, tj. koš elementů.

`GstEvent`
: Události (např. konec toku dat nebo skok na určitou pozici, tj. *seek*), které tečou v pipeline ve směru nebo v protisměru toku dat. K některým událostem je připojen buffer.

`GstPipeline`
: Nejvrchnější koš elementů, který poskytuje funkcionalitu hodin a správy sběrnice zpráv. Vytvoří se funkcí `gst_pipeline_new`. Sběrnice zpráv `GstBus` lze obdržet voláním `gst_pipeline_get_bus`.

`GstBus`
: Asynchronní sběrnice zpráv. Aplikace může zaregistrovat asynchronní callback funkci voláním `gst_bus_add_watch_full` nebo `gst_bus_add_watch`.

`GstMessage`
: Zaobaluje zprávu, která je vysílána elementy v pipeline a která je pak předávána aplikaci použitím sběrnice `GstBus`.

Funkce

`gst_init`
: Inicializuje knihovnu GStreamer.

`gst_element_factory_make`
: Vytvoří element z názvu šablony.

`gst_element_factory_find`
: Vrátí šablonu elementu (`GstElementFactory`) podle názvu.

`gst_element_factory_create`
: Ze šablony (`GstElementFactory`) vytvoří element.

`gst_object_unref`
: Dekrementuje počet odkazů na objekt. Při nulovém počtu je objekt z paměti uvolněn.

`gst_object_set_name`
: Nastaví objektu jméno (při `NULL` přiřadí unikátní).

`gst_object_get_name`
: Vrátí jméno objektu.

`gst_pipeline_new`
: Vytvoří novou pipeline s daným jménem.

`gst_bin_add_many`
: Přidá do koše elementů několik elementů.

`gst_element_link_pads`
: Propojí dva elementy přes pojmenované pady.

`gst_element_set_state`
: Nastaví stav elementu (`GstState`, např. `GST_STATE_PAUSED`).

`gst_element_get_request_pad`
: Podle jména (např. „`src_%d`“) vrátí pad, který je k dispozici „na pořádání.“

`gst_pad_get_name`
: Vrátí jméno padu.

`gst_element_get_compatible_pad`
: Vyhledá v elementu pad kompatibilní s daným formátem dat (`GstCaps`).

`gst_pad_link`
: Propojí dva pady (vždy zdrojový a cílový).

`gst_caps_get_structure`
: Vrátí jednu ze struktur (`GstStructure`) uvnitř formátu dat (`GstCaps`).

`gst_caps_get_size`
: Vrátí počet struktur uvnitř formátu dat.

`gst_structure_get_int`
: Načte ze struktury celé číslo.

`gst_caps_new_simple`
: Ze struktury vytvoří formát dat.

`gst_element_link_filtered`
: Propojí zdrojový a cílový pad za použití filtru definovaném pomocí `GstCaps`.

`gst_caps_unref`

Sníží počet odkazů na formát dat.

`gst_caps_new_full`
: Z několika struktur vytvoří formát dat.

`gst_bin_add`
: Přidá do koše elementů jeden element.

`gst_element_get_static_pad`
: Podle jména vrátí již existující pad elementu.

`gst_pipeline_get_bus`
: Vrátí sběrnici zpráv (`GstBus`) dané pipeline.

`gst_bus_add_watch`
: Přidá ke sběrnici callback funkci.

Literatura

[1] Mallat, S.: *A Wavelet Tour of Signal Processing : The Sparse Way. With contributions from Gabriel Peyré.* Academic Press, třetí vydání, 2009, ISBN 978-0-12-374370-1.

[2] Richardson, I. E. G.: *H.264 and MPEG-4 Video Compression: Video Coding for Next-generation Multimedia*. Wiley, 2003, ISBN 978-0-470-84837-1.

[3] Salomon, D.: *Data Compression: The Complete Reference*. Springer-Verlag, Čtvrté vydání, 2007, ISBN 978-1-84628-602-5.

[4] Taubman, D. S.; Marcellin, M. W.: *JPEG2000: Image Compression Fundamentals, Standards and Practice*. Springer, 2002, ISBN 978-1-4613-5245-7.