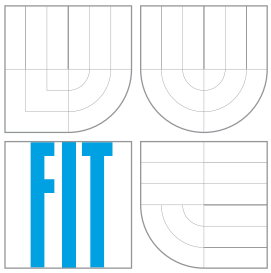
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

BEZEZTRÁTOVÁ KOMPRESE VIDEA

LOSSLESS VIDEO COMPRESSION

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

JAN KAFKA

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. DAVID BAŘINA

BRNO 2015

Abstrakt

Tato bakalářská práce se zabývá bezeztrátovou kompresí videa s použitím 3D predikce. Nejdříve je uvedena nezbytná teorie kolem úpravy obrazových dat a bezeztrátové komprese. Tato část slouží k vymezení důležitých pojmů, které jsou potřeba k samostatné tvůrčí činnosti. Následuje návrh několika variant struktury kodeku, výběr metod k implementaci a popis očekávaných vlastností. Zde lze získat první představu o způsobu fungování kodeku. Poté jsou popsány některé implementační detaily, které pomohou vytvořit přehled o vnitřní architektuře. Vlastnosti výsledného kodeku jsou nakonec otestovány a porovnány s jinými kodeky. Ukázalo se, že vlastní kodek používající 3D predikci dokáže kompresním poměrem překonat všechny ostatní kodeky v testu.

Abstract

This thesis is about the lossless video compression using 3D prediction. At first is introduced the necessary theory about image data modification. This part serves to define important terms, which are needed to start the creative work itself. Then several variants of codec's structure proposals follow, as well as the choice of methods and description of expected characteristics. Here we can get the first idea how the codec works. Then there is a description of some implementation's details, which could give a detailed insight into the codec's inner architecture. In the last part of the work, the implemented codec is tested and its characteristics are compared to another lossless codecs. The result is that the own codec, using the 3D prediction, can defeat all other tested codecs with its compression ratio.

Klíčová slova

bezeztrátová komprese, kodek, barevná transformace, prediktor, entropie, entropický kodér, multimediální framework

Keywords

lossless compression, codec, color transform, predictor, entropy, entropy encoding, multimedia frameworks

Citace

Jan Kafka: Bezeztrátová komprese videa, bakalářská práce, Brno, FIT VUT v Brně, 2015

Bezeztrátová komprese videa

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Davida Bařiny

.....
Jan Kafka
18. května 2015

Poděkování

Děkuji vedoucímu práce panu Ing. Davidu Bařinovi za odborné rady a čas.

Obsah

Kapitola 1

Úvod

Digitální multimediální obsah se stává čím dál důležitější součástí reality. A čím dál větší roli hrají audiovizuální díla. Ať už jde o možnost konzumace obsahu vytvořeného někým jiným (zpravodajství, filmy, seriály), nebo o možnost vlastní tvorby a publikování širokému spektru diváků (např. na YouTube). S rostoucím množstvím tohoto obsahu a s trendem zlepšování kvality rostou i požadavky na efektivní ukládání dat. Pro přenos po síti, pro distribuci na fyzických médiích, nebo např. pro skladování soukromých nahrávek je důležité zvolit optimální způsob, jak obrazová a zvuková data ukládat. Práce přímo se surovými daty (např. data ze snímacího čipu kamery) je nepraktická i dnes, ačkoliv jsou k dispozici úložiště vysokých kapacit s krátkou přístupovou dobou. Podle způsobu použití se tedy volí jisté metody, které pomohou zmenšit datovou náročnost multimediálního obsahu. Tomuto procesu se obecně říká komprese.

Elektronické zpracování obrazu již prošlo dlouhým vývojem. Protože jsme výchozími konzumenty většinou my, lidé, využívá se nedokonalostí našeho oka, díky kterým je možné v obrazu některé složky potlačit (nebo úplně vypustit), aniž bychom to vnímali rušivě. Této skutečnosti se využívalo už v době analogového přenosu obrazu. Výhoda digitálního zpracování je však v tom, že se pracuje s diskrétními hodnotami - s čísly, na které je možné aplikovat různé matematické postupy. Tyto postupy lze opakovat a se stejnými vstupními daty dostat vždy stejný výsledek (což u analogu z principu možné nebylo). Vznikaly tak nejen algoritmy redukující nevratně některé informace v obrazu, ale také algoritmy umožnující zpětnou rekonstrukci dat beze ztráty. Těmi se zabývá tato práce.

Kapitola Bezeztrátová komprese popisuje známou teorii potřebnou k vlastní tvůrčí činnosti. Nejdříve jsou rozebírány barevné transformace, které jsou prvním krokem při zpracování obrazových dat. Následují prediktory, odstraňující prostorovou redundanci a nakonec entropické kodéry.

Ve třetí kapitole začíná vlastní práce na kodeku. Jsou představeny dvě varianty vnitřního uspořádání a jsou popsány očekávané vlastnosti každé z nich. Následuje popis implementace stěžejních částí a napojení jádra kodeku na multimediální frameworky.

Čtvrtá kapitola se věnuje testování vytvořeného kodeku. Nejdříve je otestován samostatný kodek s různou konfigurací a je nalezena ta nejlepší. Poté je srovnán s ostatními bezeztrátovými kodeky.

Kapitola 2

Bezeztrátová komprese

Jak již bylo řečeno v úvodu, pracovat se surovými daty je nepraktické. Využívají se proto algoritmy, které zmenší objem dat a komprimovaná data lze později zpětně rekonstruovat. Pokud jsou data po zpětné rekonstrukci zcela shodná se zdrojovými daty, jde o kompresi bezeztrátovou. V ostatních případech, kdy jsou určitá data vypuštěna, jde o kompresi ztrátovou. Než bude možné přejít k samotné kompresi, je potřeba provést některé úpravy zdrojových dat. Nyní budou dopodrobna rozebírány jednotlivé metody a přístupy.

2.1 Barevné modely

Základním modelem je RGB [1], který je používán především v zobrazovací technice. Každá barva je kombinací tří základních složek červená, zelená a modrá. Obraz reprezentovaný v RGB se skládá ze tří komponent - obrazů. Každý pro jednu barevnou složku. Tyto tři obrazy se při vykreslování na monitor zkombinují a vytvoří kompozitní barevný obraz. Počet bitů, které reprezentují pixel se nazývá hloubka. Pokud budeme uvažovat 8 bitů pro každou složku, pak bude mít pixel hloubku 24 bitů, což umožní 16777216 kombinací barev. Ačkoliv je model RGB intuitivní pro zobrazovací zařízení, není příliš vhodný pro zpracování. Např. modifikace intenzity barev jednoho pixelů znamená načíst pixel z bufferu, výpočet pro všechny tři složky, modifikace všech tří složek a uložení zpět do bufferu. Z těchto i jiných důvodů používají mnohé video standardy model s jasovou složkou a barevnými rozdíly. Cílem je potlačit vzájemný vztah mezi složkami, tzv. korelaci. Z modelu RGB lze odvodit i spoustu jiných modelů a z nich zpětně získat původní hodnoty.

2.1.1 YUV

YUV model [2] stejně jako RGB používá 3 hodnoty pro definici každé barvy, avšak odděluje jasovou složku od barevných složek. Y' je jasová složka, U a V jsou barevné složky. Vztah zohledňuje rozdílnou citlivost oka na rozdílné vlnové délky a tedy barvy. Modrá se na celkovém jasu podílí nejméně, červená více a zelená nejvíce.

$$\begin{aligned} Y' &= 0{,}299R + 0{,}587G + 0{,}114B \\ U &= B - Y' \\ V &= R - Y' \end{aligned} \tag{2.1}$$

Model YUV byl použit v analogových televizních systémech PAL, NTSC a SECAM. Zatímco černobílé přijímače vykreslovaly obraz pouze pomocí jasové složky, barevné přijímače dokázaly využít i barevné složky a vykreslovat barevný obraz. To umožnilo zpětnou kompatibilitu TV vysílání. Digitální video tento model také používá, protože usnadňuje zpracování (komprese) a umožňuje zmenšit datovou náročnost podvzorkováním barevných složek. Lidské oko je citlivější na změny jasu, než na změny barvy a tak přestože bude mít obraz méně barevných informací, než jasových, celkový dojem to zásadně neovlivní. Nevýhodou je, že model YUV je ztrátový a není invertibilní.

2.1.2 LYUV

U bezeztrátových metod komprese [3] je požadavkem, aby byl proces barevné transformace dokonale reverzní. Toho lze dosáhnout pomocí invertibilních sítí. Transformaci RGB - YUV lze modifikovat tak, aby tento požadavek splnila. Závorky [] znamenají, že se hodnota zaokrouhlí na nejbližší celé číslo.

$$\begin{aligned} Y_L &= G + [0{,}299/0{,}587 \cdot R + 0{,}114/0{,}587 \cdot B] \\ U_L &= B - [0{,}587 \cdot Y_L] \\ V_L &= R - [0{,}587 \cdot Y_L] \end{aligned} \tag{2.2}$$

$$\begin{aligned} R &= V_L + [0{,}587 \cdot Y_L] \\ G &= Y_L - [0{,}299/0{,}587 \cdot R + 0{,}114/0{,}587 \cdot B] \\ B &= U_L + [0{,}587 \cdot Y_L] \end{aligned} \tag{2.3}$$

2.1.3 LOCO-I

LOCO-I je součástí standardu JPEG-LS [4]. Dekorelace barevných složek modelu RGB se provádí bezeztrátovou barevnou transformací, která obsahuje pouze operace sčítání a odčítání a je proto velmi jednoduchá.

$$\begin{aligned} C_1 &= R - G \\ C_2 &= G \\ C_3 &= B - G \end{aligned} \tag{2.4}$$

$$\begin{aligned} R &= C_1 + G \\ G &= C_2 \\ B &= C_3 + G \end{aligned} \tag{2.5}$$

2.1.4 RCT

Ve standardu JPEG 2000 [5] existují dva typy transformací. Ztrátová ICT (Irreversible Component Transformation) a bezeztrátová RCT (Reversible Component Transformation). Z pohledu této práce je zajímavější právě RCT. Symboly $\lfloor\rfloor$ znamenají zaokrouhlení výsledku na nejbližší celé číslo dolů. Všechny tři komponenty (barevné složky) by měly mít stejné vzorkovací parametry a stejnou bitovou hloubku.

$$
\begin{aligned}
Y_r &= \left\lfloor \frac{R + 2G + B}{4} \right\rfloor \\
V_r &= R - G \\
U_r &= B - G
\end{aligned}
\tag{2.6}
$$

$$
\begin{aligned}
G &= Y_r - \left\lfloor \frac{U_r + V_r}{4} \right\rfloor \\
R &= V_r + G \\
B &= U_r + G
\end{aligned}
\tag{2.7}
$$

2.2 Prediktory

Pokud vybereme z obrazu náhodně nějaký pixel, je velká šance, že okolní pixely budou mít stejnou, nebo velmi podobnou barvu [8]. Komprese obrazu je proto založena na tom, že okolní pixely mají vysoký stupeň závislosti (korelace). Tato korelace je také nazývána prostorovou redundancí. Kontext pixelu (hodnoty několika okolních pixelů) lze využít k predikci jeho hodnoty. Můžeme uvažovat pixel P, z jeho kontextu spočítáme průměr hodnot A a budeme predikovat, že pixel P bude mít hodnotu A. Podle principu komprese obrazu se v mnoha případech trefíme a jindy alespoň přiblížíme skutečné hodnotě. Rozdíl mezi skutečnou hodnotou a hodnotou predikovanou můžeme spočítat jako $\triangle = P - A$. Postup lze provádět s každým pixelem. Rozdíly jsou chyby predikce a můžeme jim přiřazovat kód proměnné délky tak, že malým číslům (většina hodnot) budeme přiřazovat kratší kód a velkým číslům (méně hodnot) delší kód. Kontext pixelu může obsahovat jen jeden, nebo dva okolní pixely, avšak lepších výsledků predikce je dosaženo, pokud je do kontextu zahrnuto více okolních pixelů. V kontextu by také měly být obsaženy pouze pixely, které již byly zpracovány. To je nutný předpoklad ke správnému dekódování. Způsobů, jak určovat kontext pixelů existuje více. Pokud pracujeme s barevnými obrázky, jejichž každý pixel má tři složky, bude potřeba provádět predikci s každou složkou zvlášť.

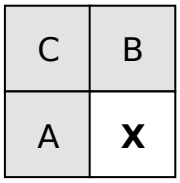
2.2.1 Lineární prediktor

Lineární prediktor využívá matematickou operaci, která následující hodnoty predikuje pomocí lineární funkce předchozích hodnot [1]. Pokud budeme uvažovat jeden vzorek (pixel) jako funkci dvou prostorových souřadnic x a y, pak můžeme lineární predikci vyjádřit např. jako vztah:

$$
f(x, y) = \left\lceil \sum_{i=1}^{m} \alpha_i f(x, y - i) \right\rceil \tag{2.8}
$$

kde m je počet vzorků, které jsou použity k predikci a α je koeficient predikce. Podle polohy vzorků se rozlišuje 1D predikce (ze současného řádku), 2D predikce (z více řádků) a 3D predikce (z více snímků). Rovnice nemůže být vypočtena pro několik prvních vzorků, protože ještě nejsou k dispozici okolní vzorky. Ty proto musí být zakódovány jinak, nebo se místo nich použije nějaká výchozí hodnota.

Složitějším lineárním prediktorem je např. Paeth [9], který je použit ve formátu PNG. Kontextem jsou tři okolní pixely A, B a C. Šedou barvou jsou označeny pixely, které má k dispozici i dekódovací strana a je možné je použít k výpočtům. Prediktor pomocí nich spočítá jednoduchou lineární funkci a poté vybere sousedící pixel, který je nejblíže vypočítané hodnotě.



Obrázek 2.1: Kontext prediktoru Paeth

```
p = a + b - c;
pa = abs(p - a);
pb = abs(p - b);
pc = abs(p - c);

if(pa <= pb && pa <= pc)
  return a;
else if(pb <= pc)
  return b;
else
  return c;
```

Kód 2.1: Algoritmus prediktoru Paeth

2.2.2 Nelineární prediktory

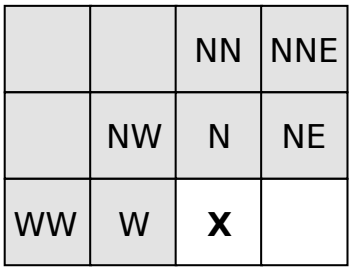
Nelineární prediktory také využívají kontext pixelu, ale predikovaná hodnota se určí na základě dané nelineární funkce. Takovým prediktorem je např. MED (Median Edge Detector), který je použit ve formátu LOCO-I/JPEG-LS [4]. Kontextem predikovaného pixelu X jsou tři okolní pixely A, B a C, stejně jako v případě Paeth. Pixel B je vybrán jako predikovaná hodnota v případech, kdy existuje vertikální hrana vlevo od pixelu X. Pixel A je vybrán v případech, kdy existuje horizontální hrana nad pixelem X. V ostatních případech, kdy není detekována žádná hrana, je predikovanou hodnotou $A + B - C$.

$$X = \begin{cases} \min(A, B) & \text{pokud } C \geq \max(A, B) \\ \max(A, B) & \text{pokud } C \leq \min(A, B) \\ A + B - C & \text{jinak} \end{cases} \tag{2.9}$$

Tento prediktor lze interpretovat i tak, že odhadovaná hodnota pixelu X je mediánem hodnot okolních pixelů A, B a $A + B - C$. Proto byl při standardizaci zvolen název „median edge detector“.

$$X = \text{median}(A, B, A + B - C) \tag{2.10}$$

Dalším příkladem nelineárního prediktoru je GAP (Gradient Adjusted Predictor) [10]. Předností tohoto prediktoru je vysoká adaptivnost, protože rozpozná slabé, normální i ostré horizontální a vertikální hrany. Kontextem predikovaného pixelu X je 7 okolních pixelů, pojmenovaných podle světových stran v angličtině.



Obrázek 2.2: Kontext pixelu v případě GAP

GAP používá k detekci hran lokální odhad gradientu g_h a g_v a tři heuristicky definované prahy (hodnoty 8, 32 a 80). Dosahuje menších chyb predikce, než MED, ale je o mnoho komplikovanější.

$$g_h = |W - WW| + |N - NW| + |N - NE| \tag{2.11}$$
$$g_v = |W - NW| + |N - NN| + |NE - NNE| \tag{2.12}$$

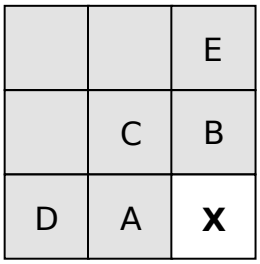
```
if(gv - gh > 80) P = W;
else if(gv - gh < -80) P = N;
else
{
  P = (W + N)/2 + (NE - NW)/4;
  if(gv - gh > 32) P = (P + W)/2;
  else if(gv - gh > 8) P = (3P + W)/4;
  else if(gv - gh < -32) P = (P + N)/2;
  else if(gv - gh < -8) P = (3P + N)/4;
}
```

Kód 2.2: Algoritmus prediktoru GAP

GED (Gradient Edge Detector) [11] je modifikace, která využívá výhod obou prediktorů MED a GAP. Kontextem predikovaného pixelu je 5 okolních pixelů.

Prediktor pracuje pouze s jedním prahem T, který může být pevně nastaven, nebo zvolen podle kontextu. Výpočet vertikálního a horizontálního gradientu je jednodušší, než u prediktoru GAP.

$$g_v = |C - A| + |E - B| \tag{2.13}$$
$$g_h = |D - A| + |C - B| \tag{2.14}$$



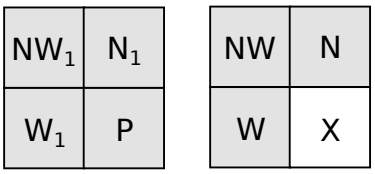
Obrázek 2.3: Kontext pixelu v případě GED

```
if(gv - gh > T)
  P = A;
else if(gv - gh < -T)
  P = B
else
  P = A + B - C
```

Kód 2.3: Algoritmus prediktoru GED

2.2.3 3D prediktor

Video je sekvence snímků, mezi kterými může být jen malý rozdíl. Nabízí se tedy možnost použít k predikci nejen současný snímek, ale i předchozí snímky. Pokud prediktor lépe odhadne hodnotu pixelu, pak je možné dosáhnout lepšího kompresního poměru a stále bez jakékoliv ztráty informace. Prediktor 3D JPEG-LS [12] byl odvozen z MED prediktoru a je navržen pro kompresi volumetrických dat z CT (Computed Tomography), MR (Magnetic Resonance), ultrazvuku apod. Tato data obsahují kromě prostorové redundance také značnou redundanci mezi snímky. Prediktor pracuje se stejným okolím jako MED, ale v současném i předchozím snímku.



Obrázek 2.4: Kontext pixelu X v případě 3D JPEG-LS

$$X = \begin{cases} \min(N, W, P) & \text{pokud } \min(NW, N_1, W_1) \geq \max(N, W, P) \\ \max(N, W, P) & \text{pokud } \max(NW, N_1, W_1) \leq \min(N, W, P) \\ \frac{2\times(W+N+P)}{3} - \frac{N_1+W_1+NW}{3} & \text{jinak} \end{cases} \quad (2.15)$$

N_1, W_1 a NW_1 jsou okolní pixely v předchozím snímku, zatímto N, W a NW jsou okolní pixely v současném snímku. X je predikovaný pixel a P je pixel na stejné pozici v předchozím snímku. Aby mohl prediktor pracovat, musí mít k dispozici právě jeden předchozí snímek.

2.3 Entropické kodéry

Entropie je významný pojem z oblasti komprese [7]. Má souvislost s kódy proměnné délky a představuje průměrnou minimální délku kódového slova. Pokud budeme dále uvažovat znaky abecedy, pak entropie rozdělení pravděpodobnosti znaků $\{p_j\}$, zapsaná jako $H(\{p_j\})$, nebo jednoduše H je:

$$H = -\sum p_j \log_2 p_j \tag{2.16}$$

Základ logaritmu určuje jednotky H. V tomto případě je základ 2, takže jednotkami jsou bity. Pro entropii $H(\{p_j, 1 \leq j \leq M\})$ platí:

$$0 \leq H \leq \log_2 M \tag{2.17}$$

kde M je celkový počet znaků. Platí, že průměrná délka kódového slova $\bar{l}$ je vždy alespoň rovna entropii H:

$$\bar{l} \geq H \tag{2.18}$$

Pokud platí, že pravděpodobnost každého znaku je rovna celé záporné mocnině dvou, pak např. Huffmanův kód dosahuje této rovnosti. V ostatních případech bude průměrná délka kódových slov $\bar{l}$ vždy větší, než entropie. Rozdíl mezi $\bar{l}$ a H se nazývá redundance. Snahou je redundanci minimalizovat a přiblížit se co nejvíce entropii.

2.3.1 Golombovo kódování

Golombovo kódování [13] patří mezi kódy s fixním pravděpodobnostním modelem. To znamená, že se neřídí podle četností kódovaných symbolů, ale má pevně nastavené mapování vstupních hodnot na výstupní. Je definováno pro celá nezáporná čísla (záporná čísla musí být nejdříve převedena na kladná). Nejkratší kód je přidělován číslům blízko nuly. Používá se modifikovatelný parametr m, který rozdělí číslo na dvě hodnoty. Výsledek celočíselného dělení q a zbytek po dělení r.

$$n = \left\lfloor \frac{n}{m} \right\rfloor + (n \bmod m) \tag{2.19}$$

Např. pro číslo 51 a $m = 10$ bude $q = 5$ a $r = 1$. Dekódování se provede podle vztahu:

$$n = q \cdot m + r \tag{2.20}$$

Hodnota q se vyjádří pomocí unárního kódu (posloupnost jedniček zakončená nulou) a hodnota zbytku po dělení r se vyjádří binárním kódem. Počet bitů potřebných k uložení čísla r se odvíjí od jeho velikosti a parametru m:

- Pokud je m mocnina 2, bude potřeba $\log_2(m)$ bitů
- Pokud m není mocnina 2, $b = \lceil \log_2(m) \rceil$ a:
  - Pokud $r < 2^b - m$, bude potřeba $b - 1$ bitů
  - Pokud $r \geq 2^b - m$, bude potřeba b bitů

Tento kód pracuje pouze s hodnotami, které jsou větší nebo rovny 0. Chyby predikce však mohou být i záporné. Nejdříve je tedy potřeba namapovat záporné hodnoty na kladné. Toho lze docílit mapováním záporných hodnot na kladné liché a kladných hodnot na kladné sudé. Nula zůstane beze změny.

```
if(n < 0)
  n = -2 * n - 1;
else
  n = 2 * n;
```

Kód 2.4: Mapování záporných hodnot na kladné

Teprve tyto hodnoty mohou být zakódovány. Na straně dekodéru lze poté jednoduchým testem rozpoznat, zda jde o liché, nebo sudé číslo a provést zpětné mapování.

```
if(n & 1)
  n = -((n + 1) / 2);
else
  n = n / 2;
```

Kód 2.5: Mapování na původní hodnoty

Golombovo-Riceovo kódování [14] je případ, kdy je parametr m mocnina 2. Kódování a dekódování lze díky tomu efektivně realizovat pomocí bitových posunů a masek, místo operací dělení a modulo. Následující tabulka ukazuje příklad použití tohoto kódu na hodnoty 5 - 10 s parametrem $m = 4$.

Symbol	**q**	**r**	**Výsledný kód**
5	10	01	1001
6	10	10	1010
7	10	11	1011
8	110	00	11000
9	110	01	11001
10	110	10	11010

Tabulka 2.1: Příklad Golombova-Riceova kódování

```
int q,num,m,b;
m = 4;
b = log2(m);
while(!symbolArray.empty()) {
  num = symbolArray.get();
  q = num / m; // vypocet podilu
  for(int i = 0; i < q; i++)
    putBit(1); // zapise q jednicek
  putBit(0) // zapise nulu;
  int v = 1;
  for(int i = 0; i < b; i++)
  { // zapise binarne zbytek po deleni
    putBit(v & num);
    v = v << 1; // bitovy posun
  }
}
```

Kód 2.6: Algoritmus Golombova-Riceova kódování

Pro čísla, která jsou blízko k nule je vhodné volit parametr m menší a pro velká čísla zase větší. Při nastavení m na hodnotu 1 jde v podstatě o unární způsob kódování. To je vhodné pro kódování chyb predikce rovné 0, které se zakódují jako jeden nulový bit (žádné jedničky, žádný zbytek, pouze oddělovací nula). I když je nulových chyb obvykle nejvíce, vyskytují se také chyby větší. U nich by bylo kódování s parametrem $m = 1$ neefektivní. Následující tabulka ukazuje, jak se mění délka výsledného kódu v závislosti na parametru m.

Symbol	**m=1**	**m=2**	**m=4**	**m=8**
0	0	00	000	0000
1	10	01	001	0001
2	110	100	010	0010
3	1110	101	011	0011
4	11110	1100	1000	0100
5	111110	1101	1001	0101

Tabulka 2.2: Délka kódu v závislosti na parametru m

Ideální tedy je, pokud se parametr m během kódování přizpůsobuje kódovaným hodnotám. Způsob, jakým se to provádí však musí být dosažitelný i na straně dekodéru. Nelze např. nastavovat parametry kodéru podle aktuální kódované hodnoty. Bylo by to sice ideální, ale dekódovací strana by toto nastavení nebyla schopna odvodit. Nabízí se ale možnost určovat parametr m podle již zakódovaných/dekódovaných hodnot, které tvoří určitý kontext.

2.3.2 Aritmetické kódování

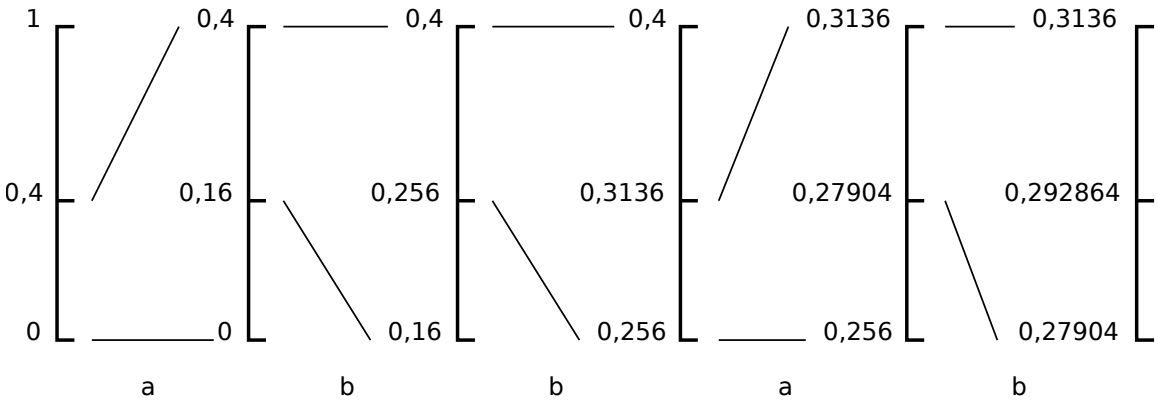
Aritmetické kódování [8] přiřazuje jeden kód celému vstupnímu souboru. Metoda pracuje s určitým intervalem hodnot. Čte vstupní soubor symbol po symbolu a používá pravděpodobnost výskytu každého symbolu ke zúžení intervalu. Z výsledného intervalu stačí vzít libovolné číslo (např. takové, které lze nejjednodušeji reprezentovat). Čím užší interval je potřeba vyjádřit, tím je kód delší. Symboly s větší pravděpodobností zužují interval méně, než symboly s menší pravděpodobností. Interval může být specifikován jeho dolní a horní hranicí.

Jako první krok je potřeba spočítat, nebo alespoň odhadnout četnost výskytu každého symbolu. Nejlepších výsledků je dosaženo, když je soubor nejdříve procházen a četnosti jsou přesně spočteny. Nicméně, pokud je možné hodnoty dobře odhadnout, není to potřeba. Algoritmus můžeme demonstrovat na příkladě, kdy jsou vstupními symboly a a b s četností $a = 2$ a $b = 3$ a tvoří řetězec *abbab*. Interval je $\langle 0; 1)$. Po získání statistik je sestavena tabulka. Interval rozdělen v poměru 2:3 podle četnosti symbolů. Vzniknou tedy dva intervaly $\langle 0; 0{,}4)$, kde patří symbol a a $\langle 0{,}4; 1)$, kde patří symbol b. Shrnuto v tabulce:

Symbol	**Četnost**	**Interval**
a	2	$\langle 0; 0{,}4)$
b	3	$\langle 0{,}4; 1)$

Tabulka 2.3: Statistiky vstupních symbolů

Tyto statistiky jsou uloženy do výstupních dat ještě před jakákoliv komprimovaná data (dekódující strana je potřebuje). Nyní se zpracovává symbol po symbolu. Prvním je a, takže se zvolí interval $\langle 0; 0{,}4)$, protože symbol patří do tohoto intervalu. Tento interval je znovu rozdělen v poměru 2:3 na intervaly $\langle 0; 0{,}16)$ a $\langle 0{,}16; 0{,}4)$. Dalším symbolem na vstupu je b, zvolí se interval $\langle 0{,}16; 0{,}4)$, opět se rozdělí v poměru 2:3 na intervaly $\langle 0{,}16; 0{,}256)$ a $\langle 0{,}256; 0{,}4)$ a pokračuje se čtením dalšího symbolu. Celý postup ukazuje následující obrázek.



Obrázek 2.5: Postup při zmenšování intervalu

Výsledkem je interval $\langle 0{,}27904; 0{,}3136)$. Z něj lze vybrat libovolné číslo, např. 0,3.

Pro zápis algoritmu pomocí pseudokódu si definujeme proměnné `Low` a `High`, představující hranice intervalu a inicializujeme `Low` na hodnotu 0 a `High` na hodnotu 1. Dále si definujeme proměnnou `Range`, která bude uchovávat aktuální velikost intervalu a funkce `HighRange(X)` a `LowRange(X)`, které vracejí krajní hodnotu intervalu pro znak X.
Optimální číslo z výsledného intervalu určí funkce `Optimum(Low,High)`.

```
Low = 0;
High = 1;
for all(symbols as s)
{
  Range = High - Low;
  Low = Low + Range * LowRange(s);
  High = Low + Range * HighRange(s);

}
Result = Optimum(Low,High);
```

Kód 2.7: Algoritmus kódování aritmetickým kodérem

Pokud je dopředu známý počet symbolů, což je přesně případ komprese snímku, stačí použít smyčku for. Jinak je potřeba definovat symbol, který je odlišný od všech ostatních a bude indikovat konec vstupu.

Proces dekódování probíhá v opačném pořadí. Začíná rekonstrukcí tabulky 2.3. Poté přečte první hodnotu, která je v našem případě 0,3. Podle tabulky patří tato hodnota do intervalu $\langle 0; 0{,}4)$, který odpovídá symbolu a, takže prvním dekódovaným symbolem je a. Poté se od čísla 0,3 odečte hodnota *Low* znaku a, což je 0 a výsledek je ještě vydělen délkou intervalu, ve kterém se nachází znak a. Získá se nová hodnota 0,75 a postup se opakuje.

Hodnota	Interval	Symbol	Nová hodnotal
0,3	$\langle 0; 0{,}4)$	a	$(0{,}3-0)/(0{,}4-0)$
0,75	$\langle 0{,}4; 1)$	b	$(0{,}75-0{,}4)/(1-0{,}4)$
0,5833	$\langle 0{,}4; 1)$	b	$(0{,}5833-0{,}4)/(1-0{,}4)$
0,3055	$\langle 0; 0{,}4)$	a	$(0{,}3055-0)/(0{,}4-0)$
0,7637	$\langle 0{,}4; 1)$	b	-

Tabulka 2.4: Postup při dekódování

V pseudokódu si definujeme další funkce a proměnné. `Code` představuje číslo, ze kterého se dekóduje. `Range(s)` vrací délku intervalu symbolu `s`. `FindInterval(Code)` najde příslušný interval odpovídající hodnotě `Code` a vrátí jeho index. `SymbolFrom(Index)` vrátí symbol podle hodnoty `Index`. Dekódované znaky jsou zapisovány na výstup `Output`. Algoritmus bude předpokládat, že tabulka 2.3 je již sestavena a počet symbolů je dopředu známý (byl součástí dat).

```
Code = data;
for (symbols)
{
  Index = FindInterval(Code);
  s = SymbolFrom(Index);
  Output.append(s);
  Code = (Code - LowRange(s)) / Range(s);
}
```

Kód 2.8: Algoritmus dekódování aritmetickým kodérem

Pokud by počet symbolů nebyl známý, musel by být opět mezi symboly jeden navíc, indikující konec algoritmu. Takový symbol by musel být i součástí tabulky 2.3 a jeho pravděpodobnost by měla být co nejmenší (např. 1 / celkový počet symbolů). Aritmetický kodér může pracovat buď s přesným modelem dat, který získá analýzou, nebo může vycházet z nějakého základního modelu a ten aktualizovat během své činnosti. Poté odpadá nejen nutnost kódování a dekódování tabulky, ale také dvojitý průchod kódovanými daty.

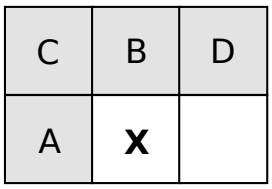
Při implementaci aritmetického kodéru lze narazit na problémy, protože počítat s nekonečnými reálnými čísly je možné pouze v teoretické rovině. Algoritmus se dá ale přepsat s použitím celočíselné aritmetiky [15]. Protože jsou pravděpodobnosti rovny četnosti výskytu symbolů, lze je normalizovat na počet symbolů a s ním pracovat. Dolní mez (`Low`) představuje součet četností všech menších symbolů a horní mez (`High`) představuje součet dolní meze a četnosti aktuálně kódovaného symbolu. Jako interval k dělení lze použít 32-bitovou proměnnou, ze které se však využije jen 31 bitů, aby se předešlo přetečení. Při kódování spodní a horní mez konvergují k sobě a je proto nutné provádět úpravy intervalů promocí E1, E2 a E3 škálování.

2.3.3 Kontextové kódování

Kontextové kodéry [8] mohou využívat aritmetické, nebo Golombovo-Riceovo kódování. Nekódují však pravděpodobnost výskytu samotného symbolu, ale pravděpodobnost jeho výskytu v nějakém kontextu. Kontextem může být např. několik předcházejících symbolů, okolní pixely apod. Řád kontextu udává, kolik symbolů je k predikci použito. Kontext musí

tvořit symboly s ohledem na dekódovací stranu, která musí být schopna daný symbol predikovat. Kontextem je celé nezáporné číslo a může sloužit k výběru modelu pro kontextový entropický kodér. Ten pak může pracovat s několika pravděpodobnostmi výskytu symbolů. Větší počet kontextů vede potenciálně k lepším výsledkům, avšak jen do určité míry. V typickém snímku je zpracovávaný pixel závislý jen na několika okolních pixelech a pracovat s příliš velkým kontextem může vést naopak k horším výsledkům.

Kontext se dá také využít ke korekci predikovaných hodnot. Příkladem může být standard LOCO-I/JPEG-LS [4]. Kromě klasické predikce probíhá navíc určování kontextu. Nejdříve jsou spočítány tři rozdíly $g_1 = D - B$, $g_2 = B - C$ a $g_3 = C - A$ z hodnot okolních pixelů.



Obrázek 2.6: Kontext JPEG-LS

Tyto rozdíly vyjadřují lokální gradient, což je charakter okolí predikovaného pixelu (hladké, nebo s ostrými přechody) a má blízký vztah ke statistickému chování chyby predikce. Rozdíly jsou poté kvantovány do 9 regionů a každý region je indexován od -4 do 4. Kontexty jsou získány za předpokladu, že platí symetrie

$$P\left(e|C_t = [q_1, q_2, q_3[\right) = P\left(-e|C_t = [-q_1, -q_2, -q_3[\right) \tag{2.21}$$

Pokud je první nenulový kvantovaný gradient záporný, změní se znaménka všech ostatních kvantovaných gradientů a k celé trojici se asociuje záporné znaménko. V opačném případě kladné. Celkový počet kontextů je $((2 \times 4 + 1)^3 + 1)/2 = 365$. Po výpočtu predikce se k hodnotě podle znaménka přičte, nebo odečte hodnota kontextu a je spočítána chyba predikce, která je rovněž upravena podle znaménka kontextu. Celý postup komprese je shrnut v bodech.

1. Výpočet gradientů g_1, g_2, g_3
2. Kvantování hodnot gradientů na hodnoty kontextu q_1, q_2, q_3
3. Výběr režimu podle kontextu
4. Výpočet predikce
5. Korekce predikce
6. Výpočet chyby predikce
7. Korekce chyby predikce
8. Aktualizace hodnot pro další pixel

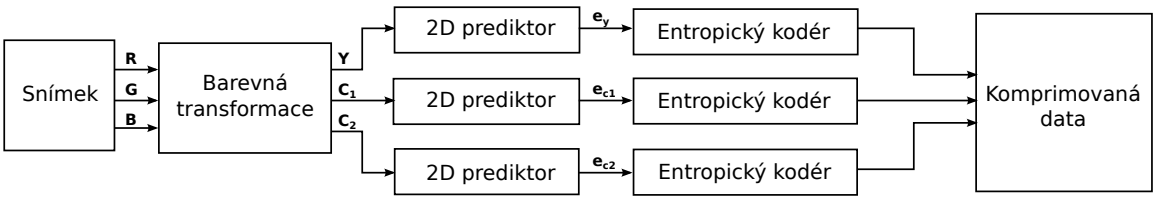
Kapitola 3

Návrh a implementace

Tato kapitola popisuje vnitřní uspořádání kodeku a způsob softwarové implementace. Metod je možné použít více v různé kombinaci. Kodek nemusí být jen napevno implementovaný algoritmus. Součástí kodeku může být i konfigurační dialog, případně může být nastavován parametry příkazového řádku. Lze tak implementovat několik variant barevné transformace, prediktorů a entropických kodérů. Uživateli je poté poskytnuta možnost volby jednotlivých postupů, případně i nastavení parametrů.

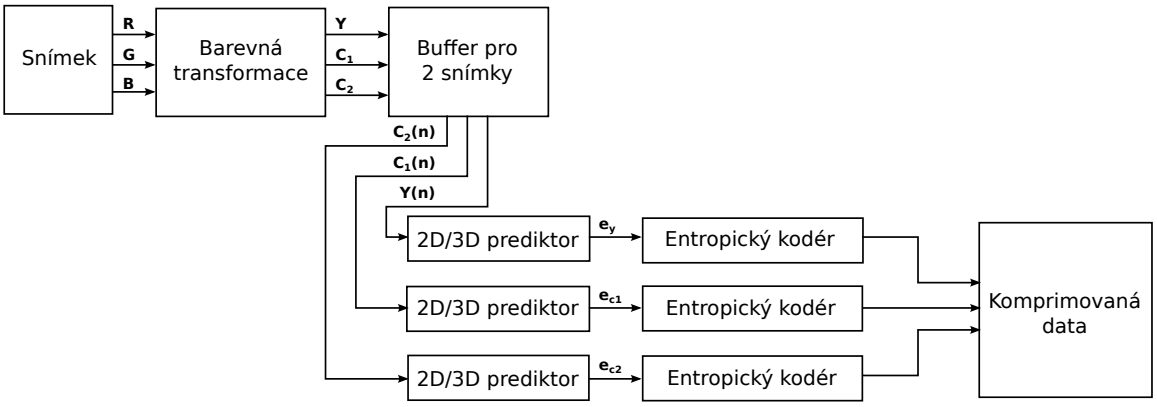
3.1 Struktura kodeku

Kodek musí obsahovat kodér i dekodér. Protože dekodér používá pouze inverzní postup, bude zde popsán a rozebrán zejména kodér. V prvním uspořádání je na každý snímek nejprve aplikována barevná transformace a dekorelované složky jsou zpracovávány zvlášť pomocí tří nezávislých prediktorů. Tyto prediktory pracují pouze s aktuálním snímkem a jsou typu 2D. Chyby predikce jsou kódovány nezávislými entropickými kodéry a kódová slova jsou prokládána. Výsledkem jsou komprimovaná data. Jde o jednoduché uspořádání, jehož implementace je ze všech variant nejjednodušší. Je očekávána nejvyšší rychlost komprese, avšak kompresní poměr spíše průměrný.



Obrázek 3.1: Struktura kodéru - varianta 1

Ve druhém uspořádání je mezi barevnou transformaci a prediktory vložen buffer pro dva snímky. Ten bude obsahovat již dekorelované složky pixelů z aktuálního a jednoho předchozího snímku. Predikce tak může být rozšířena na 3D. Výhodou pozice bufferu je, že se při opakovaném přístupu ke snímku nemusí znovu provádět barevná transformace. 3D prediktor může buffer využít, jakmile obsahuje jeden kompletní předchozí snímek. Je očekávána nižší rychlost komprese, ale lepší kompresní poměr.



Obrázek 3.2: Struktura kodéru - varianta 2

3.2 Jádro kodeku

Jádro kodeku bylo implementováno v jazyce C++11. Tvoří jej třída `JKvideocodec`, která obsahuje dvě hlavní metody `encode_frame` a `decode_frame` pro kompresi a dekompresi jednoho snímku. Ve třídě se nacházejí ještě další metody, které slouží k alokování paměti pro buffery, nastavení parametrů apod. Implementace se nachází v souborech `core.c` a `core.h`. Následuje popis stěžejních částí, které bylo při implementaci nutno řešit.

3.2.1 Reprezentace snímků v paměti

Nekomprimované snímky poskytuje framework ve framebufferu. Každý pixel je uložen jako trojice 8-bitových hodnot R, G a B (formát RGB24). Pixely jsou poskládány za sebou a stejně tak celé řádky. Jejich pořadí však může být různé. V případě formátu DIB v OS Windows jsou uloženy odspodu nahoru, ale např. framework FFmpeg na Linuxu používá pořadí řádků shora dolů. Mezi řádky může být několik nevyžitých bajtů z důvodu zarovnání nebo optimalizace pro cache CPU. Celková délka řádku s touto výplní se nazývá délka kroku (stride). Protože je buffer jednorozměrný, adresu (index) konkrétního pixelu a konkrétní barevné složky je potřeba vypočítat.

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

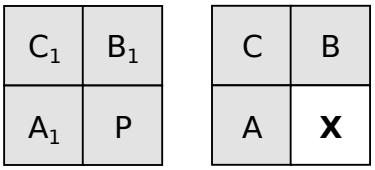
Kód 3.1: Výpočet adresy pixelu a barevné složky

Tento způsob adresování každého pixelu však není příliš efektivní. Pixely jsou ze snímku načítány postupně a tak je výhodnější pouze inkrementovat ukazatel a na konci každého řádku doplnit rozdíl do stride.

3.2.2 Vlastní verze prediktoru – MED 3D

Součástí implementace je i vlastní verze 3D prediktoru. Vycházel jsem z následujících předpokladů. Jestliže jsou v obrazu místa, která se mezi snímky nemění, lze pixely v těchto místech predikovat přesně a dosáhnout nulové chyby predikce. 3D prediktor může porovnávat snímky pouze v časové oblasti a tak nemusí být citlivý na hrany a složité textury, jako 2D prediktory. Pokud je v daném místě shoda mezi snímky, pak je predikce správná bez ohledu na charakter obrazu v samostatných snímcích. Pokud shoda není, znamená to, že v daném místě obrazu došlo k nějaké změně (např. se pohnul nějaký objekt, nebo nastal střih). Protože součástí implementace není žádná kompenzace pohybu ani detekce střihu,

může vést predikce z předchozího snímku v těchto případech k větším chybám, než predikce z okolí v současném snímku. Vlastní verze proto vychází z MED prediktoru a doplňuje jej o 3D větev. Okolí predikovaného pixelu X (A, B, C) se nejdříve porovná se stejným okolím v předchozím snímku (A_1, B_1, C_1). Pokud jsou obě okolí shodná, pak je predikovanou hodnotou pixel P z předchozího snímku, který je na stejné pozici, jako predikovaný pixel X. Jinak probíhá MED predikce pouze z okolí v současném snímku.



Obrázek 3.3: Kontext pixelu X v případě MED 3D

$$X = \begin{cases} P & \text{pokud } A = A_1, B = B_1, C = C_1 \\ \min(A, B) & \text{pokud } C \geq \max(A, B) \\ \max(A, B) & \text{pokud } C \leq \min(A, B) \\ A + B - C & \text{jinak} \end{cases} \tag{3.1}$$

Pokud jsou snímky velmi podobné, převažovat bude predikce z předchozího snímku. Jindy mohou být snímky značně odlišné a převažovat bude MED predikce. V ideálním případě bude 3D větev predikovat všechny statické části obrazu a dynamicky se měnící části přenechá MED větvi. V praxi však bude záležet na kvalitě zdrojového materiálu. Účinnost tohoto řešení může značně ovlivnit např. šum v obrazu.

3.2.3 Buffery pro prediktory

Aby mohly prediktory porovnávat okolí právě zpracovávaného pixelu, musí být k dispozici již zpracované pixely s dekorelovanými složkami `y`, `c1` a `c2`. K tomuto účelu jsou implementovány dva buffery. Každý buffer je jednorozměrné pole struktur `PX`.

```
typedef struct pixel
{
  int y;
  int c1;
  int c2;
  int err_y;
  int err_c1;
  int err_c2;

}PX;
```

Kód 3.2: Struktura PX představující dekorelovaný pixel

K okolním pixelům se přistupuje pomocí ukazatelů (jeden pro každý pixel). Ukazatele na okolní pixely se na začátku inicializují a po každém použití se inkrementují. Buffer je vždy o 1 řádek a o 1 sloupec větší, než zpracovávaný snímek a pixely mimo rozsah snímku jsou inicializovány na výchozí hodnotu (všechny položky nastaveny na 0). U prediktorů tak není potřeba řešit přístup mimo rozsah snímku. MED a Paeth prediktory pracují pouze s jedním

bufferem, zatímco 3D prediktory s oběma. Po každém zpracovaném snímku je první buffer vyměněn s druhým bufferem (prohodí se ukazatele). První buffer je přepisován aktuálně zpracovávaným snímkem a ve druhém bufferu se nachází snímek předchozí. Úplně první snímek musí být vždy celý zpracováván MED nebo Paeth prediktorem, protože předchozí snímek ještě není k dispozici.

3.2.4 Adaptivní nastavení Golombova-Riceova kodéru

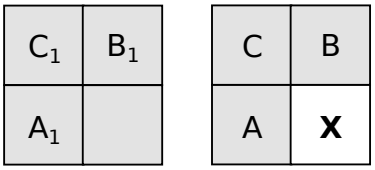
Položky `err_y`, `err_c1` a `err_c2` jsou chyby predikce posunuté do kladných hodnot a slouží pro adaptivní nastavení parametru Golom. Riceova kodéru. To probíhá pomocí tří okolních pixelů `A`, `B` a `C`. Z těchto tří chyb predikce je spočítán aritmetický průměr a parametr kodéru každé složky je nastaven podle předem definované tabulky `optimal`. Tato tabulka obsahuje optimální hodnoty dělitele pro každý symbol.

```
M_Y = optimal[(A.err_y + B.err_y + C.err_y) / 3];
M_C1 = optimal[(A.err_c1 + B.err_c1 + C.err_c1) / 3];
M_C2 = optimal[(A.err_c2 + B.err_c2 + C.err_c2) / 3];
```

Kód 3.3: Adaptivní nastavení Golomb. Rice kodéru

3.2.5 Adaptivní model dat aritmetického kodéru

Implementovaný aritmetický kodér pracuje s adaptivním modelem dat. Před zahájením kódování jsou všechny četnosti symbolů v tabulce inicializovány na hodnotu 1 a po zakódování každého symbolu se daná četnost zvýší o 1. Dekódovací strana stejným způsobem inicializuje model a provádí dekódování s aktualizacemi četností. Modely dat jsou vytvořeny dva, každý pro jeden kontext. Kontextem jsou tři okolní pixely v aktuálním a předchozím snímku.



Obrázek 3.4: Kontext aritmetického kodéru

Kontext nabývá hodnoty 1, pokud $A_1 = A$, $B_1 = B$ a $C_1 = C$. Jinak nabývá hodnoty 0. Protože jde o stejné okolní pixely, jako v případě MED 3D prediktoru, byl tento prediktor upraven tak, aby měnil i hodnotu kontextu. Kontext určuje, která ze dvou tabulek se má použít. Aktualizace četností pak probíhá pouze v jedné tabulce. Celý proces je ještě modifikován tak, aby model uvažoval jen symboly, které budou skutečně zakódovány. Při pevném nastavení by model počítal vždy s maximální možnou chybou predikce a inicializoval by i četnosti symbolů, které zakódovány nebudou. Kódování má tedy dva průchody. V prvním průchodu probíhá výpočet kontextu a chyby predikce a ukládání těchto dvojic hodnot do pomocného bufferu. Zároveň se hledá největší chyba predikce pro každý kontext zvlášť. Na konci prvního průchodu jsou největší chyby použity jako krajní hodnoty pro inicializaci tabulek a jsou také uloženy do výstupního streamu (každá hodnota na 2 byty), aby je měla k dispozici i dekódovací strana. Poté teprve probíhá kódování. Kodér podle kontextu volí jednu nebo druhou tabulku.

3.2.6 Klíčové snímky

V případě použití 3D prediktoru musí kódování probíhat postupně a snímky jsou na sobě závislé. Dekódování musí probíhat stejným způsobem, od prvního snímku po poslední. Problém může nastat při přehrávání, pokud je potřeba posun před nebo za aktuální pozici ve videu (seek). Protože všechny snímky na sebe navazují, začít dekódovat od jiného než prvního snímku nelze. Proto jsou do videa řazeny klíčové snímky, které jsou celé predikovány 2D prediktorem. Od každého klíčového snímku je možné zahájit dekódování a interval mezi klíčovými snímky je možné nastavit. Pokud je u všech snímků použit 2D prediktor, pak jsou všechny snímky klíčové.

3.2.7 Vícevláknové zpracování

Kodek umožňuje rozložit kompresi i dekompresi do 4 vláken. Obraz je rozdělen na 4 části a každému vláknu je přidělena jedna část. Např. snímek s rozlišením 1280×720 px je rozdělen na 4 snímky s rozlišením 640×360 px. Tyto snímky mohou být zpracovány paralelně.

Obrázek 3.5: Rozdělení obrazu pro více vláken

Rozdělené snímky nemusí mít stejnou velikost, pokud rozměry nejsou dělitelné dvěma. Při kompresi produkuje každé vlákno svůj datový proud. První vlákno zapisuje data přímo do cílového bufferu, zatímco ostatní vlákna zapisují data do svých nezávislých bufferů. Datové proudy je na konci komprese potřeba spojit do jednoho a zároveň je nutné uložit informace o velikosti každého proudu. Velikost není dopředu známá a nemusí být stejná u všech vláken. Informace o velikosti každého proudu potřebuje dekódovací strana. Při dekompresi předá každému vláknu ukazatel na místo, odkud má začít číst data. Vlákna provádí dekódování a zapisují původní hodnoty pixelů do společného výstupního bufferu. Protože jsou adresové prostory čtení/zápisu dat vždy odlišné, není potřeba řešit kritické sekce a stačí pouze synchronizovat dokončení provádění všech vláken.

3.3 Obálky pro multimediální frameworky

Pro sjednocení práce s kodeky byly v operačních systémech zavedeny multimediální frameworky. Jsou to softwarové komponenty, které umožňují jednotný přístup k multimediálnímu obsahu, jeho kódování, dekódování atd. Nejstarším multimediálním frameworkem na Windows je VfW (Video for Windows) [16], vyvynutý jako reakce na framework QuickTime od firmy Apple a uvedený v roce 1992. Přišel s vlastním kontejnerem AVI (Audio Video

Interlave). Nástupcem se stal framework DirectShow, který je založen na COM (Component Object Model). Zjednodušuje přehrávání, konverze formátů, nahrávání a současně poskytuje přístup k nižším vrstvám architektury pro aplikace, které požadují vlastní řešení. Zamýšleným nástupcem je multimediální framework MF (Media Foundation) [17], který již nyní existuje na Windows Vista a novějších OS. Po nějaký čas je plánována existence obou frameworků a pozvolný přechod na MF. Co se týče Unixu, zde je nejvýznamnějším multimediálním frameworkem FFmpeg [18]. Jde o svobodný software skládající se z několika knihoven. Framework může být zkompilován i pod jinými OS, včetně Windows.

Kodek se skládá z jádra, které implementuje dané algoritmy a obálek, které jádro propojí s konkrétním frameworkem. Kodek tak tvoří zásuvný modul, nebo patch. Framework při kompresi poskytuje jádru kodeku nekomprimované snímky a na výstupu kodeku očekává komprimovaná data. Při dekompresi naopak poskytuje komprimovaná data a očekává rekonstruované snímky na výstupu. Řeší také alokování paměti a ukládání dat do souboru.

3.3.1 Knihovna pro Video for Windows

Součástí frameworku je Video Compression Manager (VCM) poskytující rozhraní pro videokodeky. Zásuvný modul do tohoto frameworku je ve formě dynamicky linkované knihovny (DLL), která se do systému nainstaluje a zaregistruje. Do zdrojových kódů je třeba includovat hlavičku `<vfw.h>` a sestavovat s `winmm.lib`. Základem je funkce `DriverProc`, která přijímá zprávy od systému a vykonává konkrétní akce.

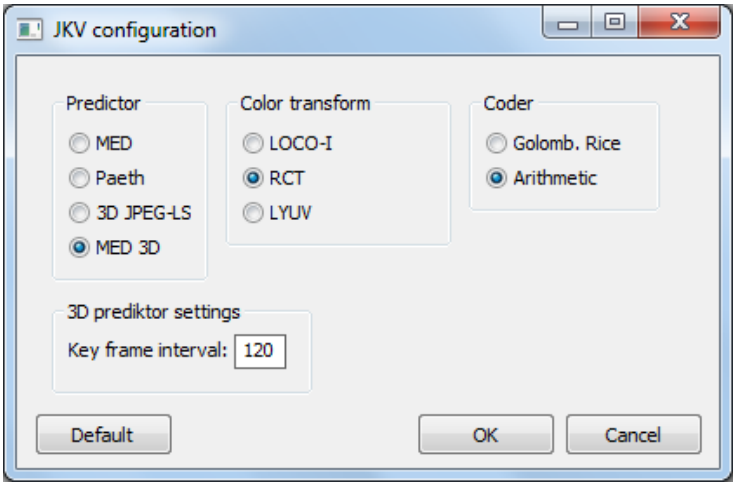
```
#include <vfw.h>

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);
  }
}
```

Kód 3.4: Kostra kodeku pro VfW

Na zprávu `ICM_COMPRESS` zareaguje kodek kompresí předaného snímku a na zprávu `ICM_DECOMPRESS` jeho dekompresí. Funkce `Compress` a `Decompress` jsou uživatelské funkce, které řeší vlastní kompresi a dekompresi. Kodek by měl reagovat i na celou řadu dalších zpráv, např. `ICM_COMPRESS_BEGIN` (připravit se na kompresi, alokovat potřebné zdroje), `ICM_COMPRESS_END` (konec komprese, uvolnění zdrojů), `ICM_COMPRESS_QUERY` (kontrola podpory vstupního a výstupního formátu), `ICM_CONFIGURE` (zobrazení konfiguračního dialogu) a další. Součástí je GUI konfigurační dialog, který umožňuje nastavit typ prediktoru, barevné transformace, kodéru a interval mezi klíčovými snímky.



Obrázek 3.6: GUI konfigurační dialog

3.3.2 Filtr pro DirectShow

Framework pracuje s grafem filtrů. Filtry lze rozdělit na zdrojové, transformační a renderovací. Rozdělení odpovídá základním bázovým třídám (base classes), které jsou součástí SDK. Kodek spadá do transformačních filtrů, kterých je více druhů (`CBaseFilter`, `CTransformFilter`, `CTransInPlaceFilter` a `CVideoTransformFilter`). Při implementaci vlastního filtru, např. kodeku je vhodné dědit z třídy `CTransformFilter` a implementovat minimálně metody `CheckInputType` (vyjednání vstupního toku), `GetMediaType` (generování seznamu podporovaných formátů pro výstup), `CheckTransform` (ověření kompatibility vstupního a výstupního typu), `DecideBufferSize` (nastavení požadavků na buffer) a `Transform` (vlastní transformace dat). Třída vlastního dekodéru může vypadat následovně.

```
class JKDec : public CTransformFilter
{
public:
  JKDec();
  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);
};
```

Kód 3.5: Kostra dekodéru pro DirectShow

Kapitola 4

Testování kodeku

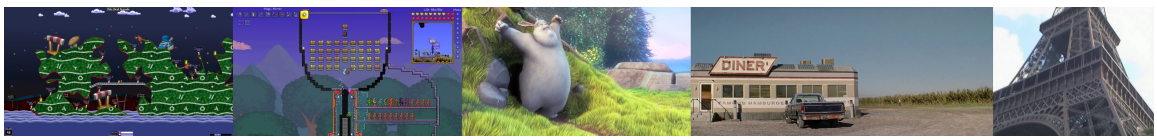
Tato kapitola se zabývá testováním vytvořeného kodeku. K tomuto účelu byla vytvořena testovací sada několika videí, která mají různé vlastnosti (rozlišení, délka, obsah). Nejdříve je otestován samotný kodek. V každém testu je zvolena určitá konfigurace a výsledky jsou porovnávány mezi sebou. Cílem je vyhodnotit vlastnosti metod barevné transformace, prediktorů a kodérů a jejich vliv na velikost výsledného souboru. Tyto základní testy slouží pro nalezení nejlepší konfigurace kodeku. Následně je provedeno srovnání s ostatními běžně používanými bezeztrátovými kodeky `Huffyuv`, `Lagarith` a `FFV1`. Porovnává se zejména kompresní poměr, ale také rychlost komprese.

4.1 Testovací sada

Testovací sada obsahuje několik videí ve formátu RGB24. Jejich vlastnosti shrnuje následující tabulka.

Název videa	**Rozlišení**	**Snímková frekv. [Hz]**	**Délka [s]**	**Velikost [kB]**
BigBuckBunny	640×360	24	596	9 670 003
BigBuckBunnyHD	1280×720	24	596	38 658 315
EiffelTower	720×576	25	60	1 822 544
Film	640×272	24	240	2 940 611
Worms	640×360	30	240	4 863 755
WormsHD	1280×720	30	240	19 481 604
Terraria	640×360	30	240	4 863 755
TerrariaHD	1280×720	30	240	19 443 965

Tabulka 4.1: Testovací sada



Obrázek 4.1: Miniatury videí - zleva Worms, Terraria, BigBuckBunny, Film, EiffelTower

Testovací videa pocházejí z různých zdrojů. Worms a Terraria jsou nahrávky z počítačové hry, BigBuckBunny je počítačová animace, Film je úryvek z hraného filmu a EiffelTower je nahrávka z Mini-DV kamery. Většina testovacích videí obsahuje alespoň v některých místech redundanci mezi snímky, kde se mohou pozitivně projevit 3D prediktory. Video EiffelTower je výjimkou, protože bylo natáčeno při chůzi z ruky a obraz obsahuje šum.

4.2 Srovnání barevných transformací

Nejdříve byly porovnány implementované barevné transformace LYUV, LOCO-I a RCT mezi sebou. Prediktor byl nastaven na MED 2D a kodér na Golomb. Riceův.

Název videa	**LYUV [kB]**	**LOCO-I [kB]**	**RCT [kB]**
Film	829 111	772 311	**768 746**
BigBuckBunny	3 765 175	3 606 535	**3 589 032**
BigBuckBunnyHD	11 801 140	11 124 166	**11 094 127**
Worms	2 122 049	2 056 657	**2 043 411**
WormsHD	7 045 149	6 803 564	**6 769 536**
Terraria	1 485 221	**1 423 886**	1 424 481
TerrariaHD	4 569 645	**4 341 829**	4 348 771
EiffelTower	557 671	**503 228**	503 650

Tabulka 4.2: Srovnání barevných transformací

Ve většině případů podávala barevná transformace RCT nejlepší výsledky. Pouze ve třech případech byla lepší LOCO-I. Lze si také všimnout, že rozdíly mezi LOCO-I a RCT jsou poměrně malé, zatímco LYUV transformace je hodnotami vzdálenější. Do dalších testů byla tedy vybrána transformace RCT.

4.3 Srovnání prediktorů

Následuje srovnání implementovaných prediktorů MED, Paeth, 3D JPEG-LS a MED 3D. Barevná transformace byla nastavena na RCT a kodér na Golomb. Riceův. Nejdříve byly porovnány prediktory MED a Paeth, které pracují pouze s jedním snímkem.

Název videa	**MED [kB]**	**Paeth [kB]**
Film	**768 746**	1 065 003
BigBuckBunny	**3 589 032**	4 306 814
BigBuckBunnyHD	**11 094 127**	14 162 055
Worms	**2 043 411**	2 244 393
WormsHD	**6 769 536**	7 334 858
Terraria	**1 424 481**	1 740 530
TerrariaHD	**4 348 771**	5 750 022
EiffelTower	**503 650**	643 387

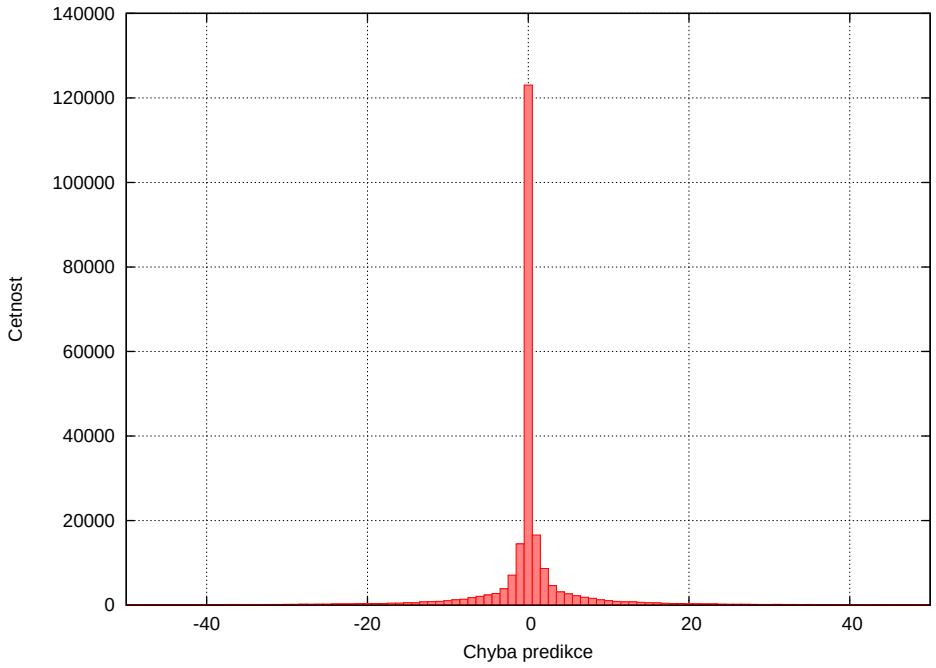
Tabulka 4.3: Srovnání 2D prediktorů

Prediktor MED dosahoval ve všech případech nejlepších výsledků. Použití samostatného prediktoru Paeth se neukázalo jako příliš vhodné. Pokud by byl tento prediktor použit v kombinaci s dalšími čtyřmi základními prediktory, jako je tomu ve formátu PNG, výsledky by byly pravděpodobně lepší. V dalším testu byl tedy vybrán MED jako zástupce nejlepšího 2D prediktoru a byl srovnán s 3D prediktory.

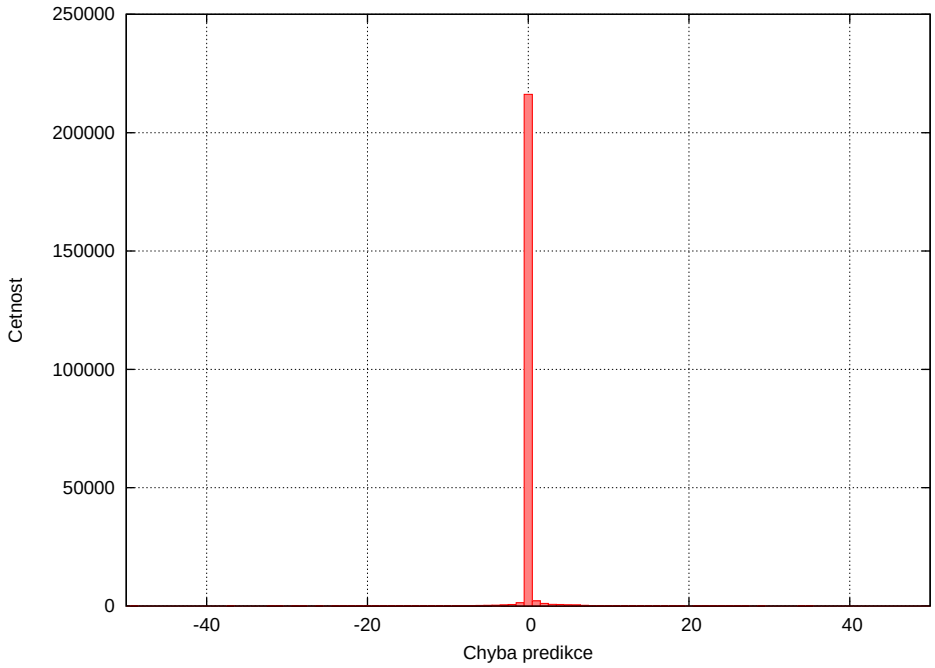
Název videa	**MED [kB]**	**3D JPEG-LS [kB]**	**MED 3D [kB]**
Film	768 746	635 608	**583 001**
BigBuckBunny	3 589 032	3 057 987	**2 690 512**
BigBuckBunnyHD	11 094 127	9 853 079	**8 708 241**
Worms	2 043 411	1 600 118	**1 013 040**
WormsHD	6 769 536	5 399 713	**3 599 209**
Terraria	1 424 481	1 186 006	**943 318**
TerrariaHD	4 348 771	3 823 297	**3 192 173**
EiffelTower	503 650	569 543	**502 579**

Tabulka 4.4: Srovnání a 3D prediktory

Téměř ve všech případech přinášejí oba 3D prediktory zlepšení, které se pohybuje od desetin procent (video EiffelTower) po více než polovinu (video Worms). Lepší výsledky podával prediktor MED 3D. Pro srovnání následují dva histogramy chyb predikce z 5. snímku videa Worms.



Obrázek 4.2: Chyby predikce - MED



Obrázek 4.3: Chyby predikce - MED 3D

Je vidět, že MED 3D prediktor produkuje podstatně méně chyb predikce různých od nuly. Rozdíl mezi 4. a 5. snímkem je malý a shodné části obrazu bylo možné predikovat s mnohem větší přesností.

Ačkoliv byl prediktor 3D JPEG-LS navržen pro volumetrická data, dokázal přinést zlepšení i u těchto testovacích videí. V případě videa EiffelTower se však objevil horší výsledek o 11,56 %, zatímco MED 3D přinesl zlepšení, i když jen o 0,21 %. Prediktor 3D JPEG-LS tedy selhává v případech, kdy jsou mezi snímky velké rozdíly a chyby predikce v těchto situacích spíše zhoršuje. Protože obsah videa není dopředu známý, nelze se spoléhat na to, že bude obsahovat dostatek redundance mezi snímky, aby měl 3D prediktor výhodu. Při srovnání predktorů 3D JPEG-LS a MED 3D by se mohlo zdát, že MED 3D využívá poměrně málo informace z předchozího snímku, pokud z něj predikuje jen hodnoty naprosto shodné mezi snímky. V tomto případě se to však ukázalo jako výhoda. Díky tomu, že MED 3D prediktor dokáže přepínat mezi 3D a 2D predikcí podle aktuálních hodnot okolních pixelů, nezhoršil chyby predikce oproti čistě 2D variantě.

Z výsledků se dá tedy usoudit, že MED 3D prediktor je nejen lepší ale také spolehlivější. Lze jej použít na libovolný video materiál a výsledky budou vždy lepší, nebo při nejhorším stejné, jako při použití MED prediktoru.

4.4 Srovnání entropických kodérů

Golomb. Riceův kodér byl srovnán s aritmetickým kodérem. Barevná transformace byla nastavena na RCT a prediktor na MED 3D.

Název videa	**Golomb. R. [kB]**	**Aritmetický [kB]**
Film	583 001	**373 381**
BigBuckBunny	2 690 512	**2 086 248**
BigBuckBunnyHD	8 708 241	**6 366 847**
Worms	1 013 040	**455 936**
WormsHD	3 599 209	**1 387 944**
Terraria	943 318	**464 777**
TerrariaHD	3 192 173	**1 229 950**
EiffelTower	**502 579**	504 795

Tabulka 4.5: Srovnání entropických kodérů

Aritmetický kodér je lepší téměř ve všech případech. Zlepšení se pohybuje od 22,46 % do 61,47 %. Největší přínos má tento kodér v případech, kdy musí kódovat velké množství nul. To nastává zejména u videí, kde se může ve velkém měřítku projevit 3D prediktor. Zatímco Golomb. Riceův kodér nemůže přidělovat kódy kratší, než 1 bit, aritmetický kodér toho schopen je a navíc pracuje s kontextem. Horší výsledek o 0,44 % se objevil pouze u videa EiffelTower. Zde se objevuje větší rozpětí chyb predikce a adaptivní model aritmetického kodéru zde nepracuje tak efektivně, jako u jiných videí. Díky velkému přínosu v ostatních případech však byl z testované dvojice zvolen aritmetický kodér jako nejlepší.

4.5 Srovnání s ostatními kodeky

Předchozí testy ukázaly, jaká je nejlepší konfigurace vlastního kodeku (barevná transformace RCT, prediktor MED 3D, aritmetický kodér). Nyní budou výsledky srovnány s kodeky Lagarith 1.3.27, Huffyuv 2.1.1 a FFV1, který byl součástí ffdshow tryouts rev. 4532. Všem kodekům byla ponechána výchozí konfigurace, až na barevný model, který byl ve všech případech nastaven na RGB a u FFV1 byl ještě změněn kodér na aritmetický. Výsledky tak budou lépe srovnatelné s vlastním kodekem. Ten je v tabulkách pod označením JKV.

Název videa	**Huffyuv [kB]**	**Lagarith [kB]**	**FFV1 [kB]**	**JKV [kB]**
Film	975 587	700 617	544 896	**373 381**
BigBuckBunny	4 960 976	3 464 919	2 967 456	**2 086 248**
BigBuckBunnyHD	15 639 592	10 462 689	8 721 213	**6 366 847**
Worms	2 836 922	1 735 659	1 306 547	**455 936**
WormsHD	9 270 077	5 496 255	3 782 533	**1 387 944**
Terraria	2 110 189	1 197 086	973 053	**464 777**
TerrariaHD	6 713 093	3 165 051	2 504 196	**1 229 950**
EiffelTower	677 834	496 946	**392 101**	504 795

Tabulka 4.6: Srovnání s jinými kodeky - velikost souboru

Název videa	**Huffyuv**	**Lagarith**	**FFV1**	**JKV**
Film	3,01:1	4,19:1	5,39:1	**7,87:1**
BigBuckBunny	1,94:1	2,79:1	3,25:1	**4,63:1**
BigBuckBunnyHD	2,47:1	3,69:1	4,43:1	**6,07:1**
Worms	1,71:1	2,80:1	3,72:1	**10,66:1**
WormsHD	2,10:1	3,54:1	5,15:1	**14,03:1**
Terraria	2,30:1	4,06:1	4,99:1	**10,46:1**
TerrariaHD	4,66:1	6,14:1	7,76:1	**15,80:1**
EiffelTower	2,68:1	3,66:1	**4,64:1**	3,61:1

Tabulka 4.7: Srovnání s jinými kodeky - kompresní poměr

Nejmenšího kompresního poměru dosáhl kodek Huffyuv. Za ním následují kodeky Lagarith a FFV1. Nejlepších výsledků, až na jeden případ, dosáhl vlastní kodek. Rozdíl oproti druhému nejlepšímu kodeku FFV1 je největší u videa Worms (65,1 %). Nejlepší kompresní poměr byl ale dosažen u videa TerrariaHD, kde činí 15,80:1. Horší výsledky se objevují jen u videa EiffelTower, kde vlastní kodek překoná pouze Huffyuv o 25,52 %. Proti kodeku Lagarith je horší o 1,55 % a proti FFV1 o 22,32 %. Z výsledků vyplývá, že vlastní kodek dokáže přinést velkou úsporu u videí, kde je hodně redundance mezi snímky. Má však ještě určitou rezervu v případech, kdy převažuje 2D predikce. To nastává právě u videa EiffelTower. Kodeky Lagarith a FFV1 pracují pouze s 2D predikcí a tu mají dokonaleji propracovanou, zatímco vlastní kodek se zaměřuje hlavně na 3D predikci.

Kodeky byly srovnány také podle rychlosti kódování. Během komprese byl sledován údaj fps (frames per second - snímková frekvence). Z minimálních a maximálních hodnot byl poté spočítán aritmetický průměr. Konfigurace kodeků nebyla od předchozích testů změněna. Vlastní kodek a Lagarith jsou jediné dva kodeky v testu, které podporují vícevláknové zpracování.

Název videa	**Lagarith [fps]**	**Huffyuv [fps]**	**FFV1 [fps]**	**JKV [fps]**
Film	**215**	164	49	85
BigBuckBunny	**95**	91	29	58
BigBuckBunnyHD	**34**	25	9	18
Worms	**128**	86	33	74
WormsHD	**34**	24	10	22
Terraria	**210**	125	45	76
TerrariaHD	**67**	35	14	23
EiffelTower	**82**	66	20	29

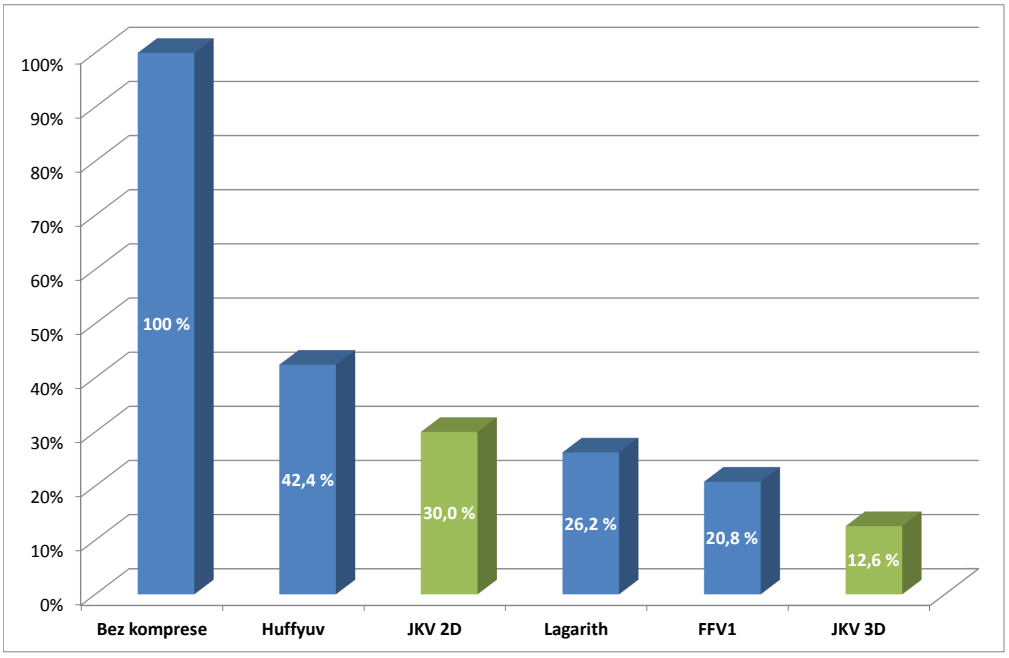
Tabulka 4.8: Srovnání s jinými kodeky - rychlost komprese

Testování probíhalo v programu VirtualDub 1.10.4 a na této testovací sestavě.

- CPU: Intel Core-i5 3210M @ 2,5 GHz
- RAM: 8 GB DDR3 1333 MHz
- OS: Windows 7 Pro 64-bit

Nejrychlejším kodekem je jednoznačně Lagarith. Za ním následuje Huffyuv a vlastní kodek. FFV1 skončil až na posledním místě. Je vidět, že využití více vláken má smysl i u bezeztrátové komprese. Přestože vlastní kodek používá náročnější 3D predikci, díky rozdělení zpracování do vláken dokáže kódovat rychleji, než kodek FFV1. Z výsledků lze také vysledovat, že se rychlost kódování mění podle náročnosti scény a v případě vlastního kodeku ještě rozhoduje velikost rozdílu mezi snímky. U videa Worms, kde se uplatňuje 3D predikce ve velkém měřítku, dosahuje vlastní kodek více než dvojnásobné rychlosti komprese, než FFV1. U videa EiffelTower je rozdíl jen 31,03 %. Rychlost komprese také nemusí znamenat nejlepší kompresní poměr. U videa EiffelTower byl kodek FFV1 sice nejpomalejší, ale dosáhl zde nejlepšího kompresního poměru. Při SD rozlišení jsou všechny kodeky kromě FFV1 použitelné pro kompresi v reálném čase (pokud se bude uvažovat 30 Hz jako spodní hranice). Při HD rozlišení již pouze Lagarith díky jeho dobré optimalizaci na více vláken a SSE instrukce.

Následuje ještě graf průměrné velikosti souborů u všech kodeků napříč testovacími videi. Hodnoty jsou uvedeny v procentech. 100 % představuje průměrnou velikost nekomprimovaných videí. Čistě 2D predikce není u vlastního kodeku příliš účinná a dokáže svými výsledky překonat pouze kodek Huffyuv. S použitím 3D predikce se však vlastní kodek dostal na nejmenší průměrnou velikost souboru, která činí 12,6 % velikosti nekomprimovaného videa. Proti druhému nejlepšímu kodeku FFV1 přináší zlepšení o 39,42 %.



Obrázek 4.4: Průměrná velikost souborů v procentech

Kapitola 5

Závěr

Tato práce se zabývala bezeztrátovou kompresí videa. Nejdříve byla shrnuta potřebná teorie kolem barevných transformací, prediktorů a entropických kodérů. Poté byly popsány dva návrhy struktury kodeku a očekávání od každé z nich. Kodek byl implementován pro multimediální frameworky Video for Windows a DirectShow.

Už během prvních fází implementace byly prováděny experimenty s 3D prediktory, protože referenční 3D JPEG-LS nepodával ve všech případech lepší výsledky, než 2D prediktory. Nejdříve byly hledány jiné prediktory, nakonec se však povedlo upravit MED prediktor doplněním o 3D větev a podařilo se dosáhnout nejen lepších výsledků, ale také lepší spolehlivosti.

Vlastnosti kodeku byly nakonec otestovány. Nejdříve byla nalezena nejlepší konfigurace, což vyžadovalo otestovat barevné transformace, prediktory a entropické kodéry. Poté byl kodek s touto konfigurací srovnán s bezeztrátovými kodeky Lagarith, Huffyuv a FFV1. Nejdůležitější bylo porovnat kompresní poměr, kde až na jeden případ zvítězil vlastní kodek se značným náskokem oproti druhému nejlepšímu FFV1. V případě testovacího videa EiffelTower se však podařilo překonat pouze kodek Huffyuv a ve srovnání s FFV je vlastní kodek horší cca o pětinu. Zde je vidět určitá rezerva pro zlepšování. Kodeky Lagarith a FFV1 jsou dobře vyladěny pro kompresi každého snímku zvlášť, zatímco vlastní kodek se zaměřil hlavně na snížení redundance mezi snímky. Zlepšení by mohlo přinést použití korekce chyb predikce pomocí kontextu, jako je např. ve formátu LOCO-I/JPEG-LS.

Kodeky byly porovnány také s ohledem na rychlost komprese. Nejrychlejší se ukázal být kodek Lagarith, následovaný Huffyuv, vlastním kodekem a na posledním místě skončil FFV1. Zde se ukázalo, že úprava vlastního kodeku pro více vláken měla smysl. Přesto však jsou i zde značné rezervy pro zlepšování. Některé operace by mohly být optimalizovány SSE instrukcemi, nebo celý kodek by mohl být akcelerován pomocí GPU.

Vytvořený kodek byl uvolněn pod licencí LGPL a je k dispozici na adrese
https://sourceforge.net/projects/jklosslessvideocodec/.

Literatura

[1] GONZALEZ, Rafael C. a Richard E. WOODS. *Digital Image Processing*. 3rd ed. Upper Saddle River: Pearson, 2008. ISBN 01-316-8728-X.

[2] POYNTON, C.A. *Digital Video and HDTV: Algorithms and Interfaces*. USA: Morgan Kaufmann Publishers, 2003. ISBN 9781558607927.

[3] VLEUTEN, René J. a Sebastian EGNER. Lossless and Fine-Granularity Scalable Near-Lossless Color Image Compression. [online]. (Endhoven, The Netherlands),[cit. 2014-12-15]. Dostupné z: http://www.win.tue.nl/wic2004/28.pdf

[4] WEINBERGER, M. J., G. SEROUSSI a G. SAPIRO. The LOCO-I Lossless Image Compression Algorithm: Principles and Standardization into JPEG-LS. In: *IEEE Transactions on Image Processing*. 2000, s. 1309 - 1324. ISSN 1057-7149.

[5] ATHANASSIOS, Skodras, Charilaos CHRISTOPULOS a Touradj EBRAHIMI. The JPEG 2000 still image compression standard. In: *Signal Processing Magazine*. IEEE, 2001, s. 36-58. 18.5.

[6] POYNTON, C. A. *A technical introduction to digital video*. New York: J. Wiley, 1996. ISBN 04-711-2253-X.

[7] GIBSON, Jerry D., Toby BERGER, Tom LOOKABAUGH, LINDBERGH a Richard L. BAKER. *Digital Compression for Multimedia*. USA: Morgan Kaufmann Publishers, 1998. ISBN 1558603697.

[8] SALOMON, D. *Data Compression: The Complete Reference*. London: Springer-Verlag, 2007. 4. ISBN 9781846286025.

[9] PNG (Portable Network Graphics) Specification. *W3C* [online]. 1996 [cit. 2015-03-01]. Dostupné z: http://www.w3.org/TR/PNG-Filters.html

[10] WANG, Kun a Gao LIQUN. A new approach of adaptive edge detection based on GAP predictor. In: *Control and Decision Conference*. China: IEEE, 2009, s. 657-660.

[11] AVRAMOVIC, A. Lossless compression of medical images based on gradient edge detection. In: *Telecommunications Forum (TELFOR), 2011 19th*. Belgrade: IEEE, 2011, s. 1199-1202. ISBN 978-1-4577-1499-3.

[12] AIT-AOUDIA, Samy, Benhamida FATMA-ZOHRA a Yousfi MOHAMED-AZZEDDINE. Lossless compression of volumetric medical data. In: *Computer and Information Sciences - ISCIS 2006*. Berlin: Springer, 2006, s. 563-571.

[13] GOLOMB, S. W. Run-length encodings. In: *IEEE Transaction on Information Theory*. IEEE, 1966, s. 399.

[14] RICE, Robert F. a James R. PLAUNT. Adaptive Variable-Length Coding for Efficient Compression of Spacecraft Television Data. In: *IEEE transactions on communication technology*. New York: Institute of Electrical and Electronics Engineers, 1971, s. 889-897. ISSN 0018-9332.

[15] BODDEN, Eric, Malte CLASEN a Joachim KNEIS. Arithmetic Coding revelated: A guided tour from theory to praxis. *Sable Technical Report*. 2007, č. 5.

[16] Video for Windows. *MSDN - the Microsoft Developer Network* [online]. [cit. 2015-03-01]. Dostupné z: https://msdn.microsoft.com/en-us/library/dd757708

[17] Microsoft Media Foundation. *MSDN - the Microsoft Developer Network* [online]. [cit. 2015-03-01]. Dostupné z: https://msdn.microsoft.com/en-us/library/ms694197

[18] FFmpeg. *A complete, cross-platform solution to record, convert and stream audio and video* [online]. [cit. 2015-03-01]. Dostupné z: http://ffmpeg.org/

Příloha A

Obsah DVD

- Zdrojové kódy kodeku ve formě projektu do MS Visual Studio
- Návod k instalaci
- Textová dokumentace kodeku
- Programová dokumentace kodeku
- Zdrojový kód dokumentace v LaTeXu
- Testovací video Worms