Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Bakalářská práce

Barevné korekce videa

Plzeň, 2007 Jan Kučera

Prohlášení

Prohlašuji, že jsem bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 15.5.2007 Jan Kučera

Abstract

Video color correction

This bachelor thesis is dedicated to the implementation of tool curves into the video processing program VirtualDub. First part of work is devoted to behavior of tool curves. The tool is described from its usage to possible implementation. Second part of work is dedicated to creation VirtualDub filter. Last part describes realization of tool curves as a filter for program VirtualDub.

Obsah

Seznam obrázků

Seznam tabulek

1 Úvod

1.1 Proč práce vznikla

Podnětem ke vzniku této práce se stala neutěšená situace na poli možností korekce barev videa ve velice oblíbeném programu VirtualDub. Pro tento program sice existuje neskutečné množství nejrůznějších filtrů pro všechny možné operace, ale přesto se v době vzniku této práce nepodařilo nalézt nástroj známý z mnoha grafických a video editačních programů a to nástroj Křivky.

Tato práce se věnuje tomuto nástroji a to od jeho možného použití až po jeho končenou implementaci do programu VirtualDub ve formě samostatného filtru.

1.2 Stručný přehled práce

První část této práce je se věnuje křivkám, a to od fungovaní nástroje Křivky přes tvorbu křivek po ukázky použití.

Další velká kapitola je věnována SDK VirtualDubu, která zahrnuje jeho popis a také ty nejnutnější informace pro vytvoření jednoduchého filtru.

Následující kapitola se věnuje popisu vlastní implementaci filtru Křivky.

Poslední větší částí této práce před samotným závěrem je uživatelský manuál.

1.3 VirtualDub

VirtualDub je program pro zachytávání a zpracování videa pod Windows, který je licencován pod licencí GNU General Public License (GPL), což nám umožňuje jej bez problémů zdarma používat a jako bonus je možné stáhnout i zdrojový kód.

Tento program je celkem jednoduchý a proto od něj nemůžeme čekat funkce jako od známých programů pro úpravu videa, jimiž jsou například Adobe Premiere nebo Pinnacle Studio, ale zato je velice rychlý, je zdarma a existuje pro něj nespočetné množství filtrů, které jsou samozřejmě také zdarma.

2 Křivky – Curves

V oblasti zpracování digitálního obrazu a to jak statického, tedy fotografií, nebo dynamického, tj. videa, se často používá nástroj nazvaný Křivky nebo spíše anglicky Curves. Tento nástroj je velice mocný a umožňuje nejrůznější úpravy obrázku, pomocí kterých z nudného obrázku uděláme obrázek velice živý, nebo ho naopak ještě více zničíme.

2.1 Způsob práce s nástrojem

Než se vrhneme na teoretické pozadí tohoto nástroje, myslím, že bude nejlepší si ho nejdříve prakticky představit, tak jak se s ním setká běžný uživatel. Jak již bylo řečeno výše, tento nástroj lze najít v každém lepším grafickém/video editoru. Chcete-li si jej ihned vyzkoušet podívejte se do svého oblíbeného editoru a při troše štěstí ho tam jistě naleznete, pokud ne, nezoufejte, je tu VirtualDub a mnou vytvořená implementace tohoto nástroje, která vznikla jako průnik několika implementací z nejrůznějších programů.

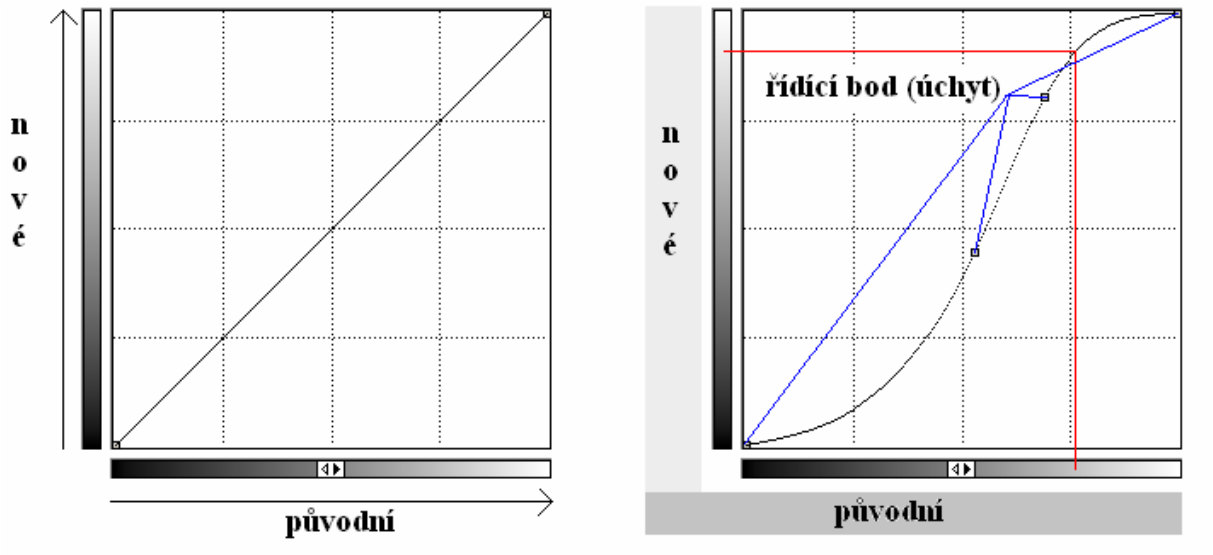
Popis použití tohoto nástroje bude proveden v RGB modelu, ale to jen z důvodu omezení programem VirtualDub, kde veškeré filtry pracují s RGB reprezentací snímků našeho videa. Toto je ale pouze omezení programu VirtualDub, jinak křivky jako nástroj fungují i v dalších barevných modelech jako CMYK, LAB. V každém z těchto modelů se dají křivky použít, ale v každém modelu se chovají trochu jinak a proto je každý model vhodný pro jiný typ korekcí.

Pomocí křivek můžeme upravovat celé barevné spektrum najednou (jasové korekce), nebo pouze jednotlivé barevné složky (barevné korekce).

Spustíme-li nástroj křivky v RGB reprezentaci, objeví se nám okno, které bude obsahovat spoustu voleb, ale také samotný nástroj křivky, který bude podobný tomu na následujícím obrázku. Na prvním obrázku je „křivka“ přímá spojnice bodů od dolního levého rohu do horního pravého rohu. Na levém a dolním okraji je panel znázorňující přechod od černé barvy po bílou nebo podle implementace pro jednotlivé složky od černé do červené, zelené, modré. Dolní panel představuje vstupní (původní) hodnoty a levý hodnoty výstupní (nové). Křivka definuje vztah mezi vstupními a výstupními hodnotami.

Pokud krajním bodům přiřadíme barvu podle panelů, tak je vidět, že dolní levý bod představuje nejtmavší bod obrázku a horní pravý bod je bodem nejsvětlejším. Křivka poté představuje přechod mezi nimi.

Jaký efekt tedy představuje křivka na obrázku 2.1? Samozřejmě, že žádný, neboť každá nová hodnota se rovná původní hodnotě.



Obrázek 2.1 Ukázka nástroje křivky

Jak se, ale dá křivka upravit? Většinou to funguje tak, že po kliknutí na křivku nebo v její blízkosti se nám vytvoří úchyt (řídící bod). Řídící bod má většinou podobu malého čtverečku, se kterým můžeme různě pohybovat nebo jej můžeme i odstranit. Pro běžnou úpravu obrázku nám vystačí použít jen několik těchto bodů, v některých implementacích bývá maximální počet omezen na zhruba 14 bodů.

Při každé změně nástroj vždy proloží body hladkou křivkou, která dala název tomuto nástroji. Zde vyvstává první vážnější problém a to jakým způsobem se křivka tvoří.

2.2 Tvorba Křivky

V předchozí části jsme se dozvěděli, jakým způsobem se nástroj křivky používá a co ho tvoří. Nyní je ten správný čas na zodpovězení otázky jak vytvořit křivku.

Obecně se křivky v počítačové grafice dělí na interpolační a aproximační. Bez váhání můžeme říci, že naše křivka bude jednoznačně interpolační, neboť ta vždy prochází zadanými body, kdežto křivka aproximační se jim jen snaží přiblížit, což neznamená, že jimi procházet nemusí.

Nyní, když víme, že se jedná o interpolační křivku, tak už nám zbývá malý problém, a to jakým způsobem interpolační křivku vytvořit. Máme-li n vstupních bodů, tak křivku můžeme vytvořit najednou jedním polynomem stupně n-1 (Lagrangeova interpolace), kdy se nám ale polynom může z důvodu vysokého stupně nepříjemně vlnit.

Nebo k aproximaci můžeme použít po částech polynomiální funkci - tzv. spline. Většinou se používá kubický spline, který je jak již název napovídá po částech tvořen polynomy třetího stupně.

2.2.1 Lagrangeova interpolace

Základní interpolace používaná v numerických metodách, která je navíc velice jednoduchá.

Je dáno n+1 uzlových bodů $x_0, x_1, \ldots, x_n$ z intervalu <a; b> a n+1 funkčních hodnot $f_0, f_1, \ldots, f_n \in R$; pak existuje právě jeden polynom $p \in P_n$ tak, že $p(x_i) = f_i$.

V Lagrangeově reprezentaci má tento polynom tvar:

$$L(x) = \sum_{i=0}^{n} l_i(x) f_i$$

kde polynom $l_i(x)$ se vypočítá podle vzorce:

$$l_i(x) = \prod_{j=o, j\neq i}^{n} \frac{(x - x_j)}{(x_i - x_j)}, \text{ pro i = 0...n}$$

Tato metoda má jednu velkou nevýhodu a tou je, že při přidání bodu, musíme přepočítat všechny polynomy $l_i(x)$.

2.2.2 Newtonova interpolace

Tato metoda interpolace řeší problém přidání bodů a proto ji zde také zmíním. Další její výhodou je, že snáze implementovatelná.

Předpokládejme, že máme stejné výchozí podmínky, poté Newtonův interpolační polynom předpokládejme ve tvaru:

$$N(x) = a_0 + a_1(x - x_0) + \cdots + a_n(x - x_0)\ldots(x - x_n)$$

Naším problémem je nyní nalézt koeficient a_i, tyto koeficienty nalezneme podle dále uvedeného vztahu jehož odvození zde nebudu uvádět, více detailů lze nalézt v [1] nebo [2].

Pro další krok si zavedeme:

$$a_i = S_i^i$$

a nyní pro $i = 0,\ldots,n$:

$$S_i^0 = f_i$$

pro j = 1; j ≤ i

$$S_i^j = \frac{S_i^{j-1} - S_{i-1}^{k-1}}{x_i - x_{i-j}}$$

Nyní můžeme již Newtonův polynom klidně sestrojit a pro ulehčení práce si ještě uvedeme zápis s použitím Hornerova schématu.

$$N(x) = (\ldots(a_n(x - x_n) + a_{n-1})(x - x_{n-2}) + \cdots + a_1)(x - x_0) + a_0$$

2.2.3 Spline

Základní myšlenka interpolace tohoto typu je obdobná jako u Lagrangeovy interpolace. Jediný rozdíl spočívá v tom, že místo polynomu, pomocí něhož interpolujeme danou funkci, bereme tzv. spline, což je funkce, která je po částech polynomiální.

Spline-funkce 1. řádu je po částech lineární funkce, jejímž grafem je lomená čára spojující zadané body. V praxi se uplatňuje zejména interpolace kubickou funkcí, tj. funkcí, která je po částech polynomem 3. řádu. Pro konstrukci spline funkce potřebujeme seřazené body, a také aby interpolační spline polynom byl spojitý s první i druhou derivací.

Nyní uvedu způsob výpočtu spline funkce, v tomto případě opět vynechám matematické odvození, které si případný zájemce může vyhledat v literatuře týkající se numerických metod, například zde [1].

Je dáno n+1 uzlových bodů $x_0, x_1, \ldots, x_n$ z intervalu <a; b> a n+1 funkčních hodnot $f_0, f_1, \ldots, f_n \in R$; pak:

pro $i = 0,\ldots,n-1$:

$$h_i = x_{i+1} - x_i$$

pro $i = 1, \ldots, n-1$

$$\alpha_i = \frac{h_{i-1}}{h_{i-1} - h_i}$$
$$\beta_i = 1 - \alpha_i$$
$$\gamma_i = \frac{6}{h_{i-1} + h_i}\left(\frac{f_{i+1} - f_i}{h_i} - \frac{f_i - f_{i-1}}{h_{i-1}}\right)$$

určíme:

$$M_0 = f_0'' = 0, \quad M_n = f_n'' = 0$$

zbylá M_i dopočteme jako řešení soustavy:

$$\begin{pmatrix} 2 & \beta_1 & 0 & 0 & \ldots & 0 & 0 \\ \alpha_2 & 2 & \beta_2 & 0 & \ldots & 0 & 0 \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ 0 & 0 & 0 & 0 & \alpha_{n-2} & 2 & \beta_{n-2} \\ 0 & 0 & 0 & 0 & 0 & \alpha_{n-1} & 2 \end{pmatrix} \begin{pmatrix} M_1 \\ M_2 \\ \ldots \\ M_{n-2} \\ M_{n-1} \end{pmatrix} = \begin{pmatrix} \gamma_1 - \alpha_1 f_0'' \\ \gamma_2 \\ \ldots \\ \gamma_{n-2} \\ \gamma_{n-1} - \beta_{n-1} f_n'' \end{pmatrix}$$

řešení této soustavy je velice jednoduché a můžeme ji snadno vyřešit pomocí Gaussovy eliminační metody, dále si spočteme:

$$A_i = \frac{f_{i+1} - f_i}{h_i} - \frac{(M_{i+1} - M_i)h_i}{6}$$
$$B_i = f_i - \frac{M_i h_i^2}{6}$$

nyní máme vše potřebné pro výpočet splinu a samotná rovnice úseku splinu mezi dvěma body se spočte podle následujícího vzorce:

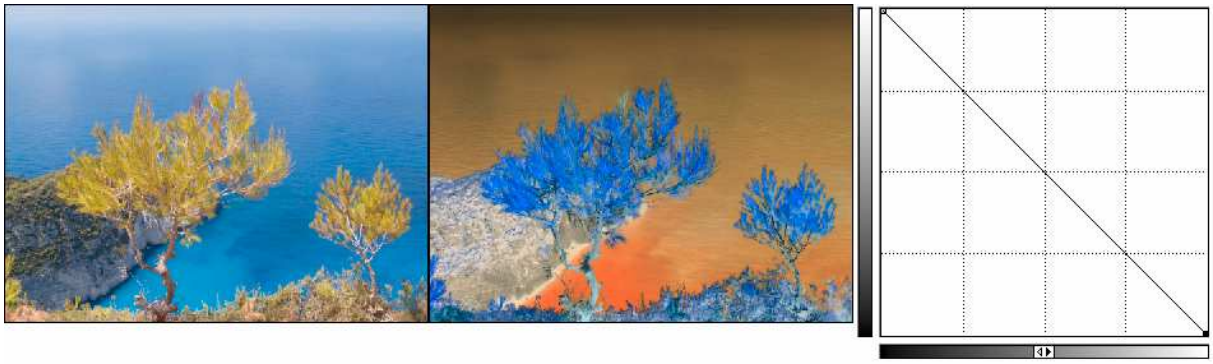
$$s_i(x) = M_i \frac{(x_{i+1} - x)^3}{6h_i} + M_{i+1} \frac{(x - x_i)^3}{6h_i} + A_i(x - x_i) + B_i$$

2.3 Ukázky použití nástroje Křivky

Nyní budou následovat ukázky několika základních použití nástroje křivky pro nejčastěji používané efekty. Část obrázku vlevo zobrazuje originální obrázek, část uprostřed obrázek po aplikaci nástroje křivky a část vpravo představuje samotnou křivku.

2.3.1 Inverze barev

Toto je asi ten nejjednodušší a nejméně používaný efekt, ale myslím, že sem patří neboť je to perfektní ukázka toho, jak vlastně křivky fungují. A jak tedy provést invertaci barev? Stačí křivku jen překlopit, tzn. že hodnotu 0 nahradíme hodnotou 255, hodnotu 1 hodnotou 254 až dojde k celkové inverzi.



Obrázek 2.2 Invertace barev pomocí nástroje křivky

Na černobílém obrázku vidíme efekt jako invertaci jasu.

Obrázek 2.3 Invertace jasu pomocí nástroje křivky

2.3.2 Úprava jasu

Pomocí křivek můžeme velice jednoduše také změnit jas obrázku. Změnu jasu lze provést dvěma způsoby.

První spočívá v přidání kontrolního bodu a jeho posunem nahoru nebo dolů, tento bod poté vyboulí křivku patřičným směrem a obrázek se ihned zesvětlí, či ztmaví. Vyboulení k hornímu okraji provede zesvětlení obrázku, kdežto vyboulení směrem k dolnímu okraji provede ztmavení.

Obrázek 2.4 Zesvětlení obrázku pomocí křivek se zachováním detailů

Zesvětlení/ztmavení můžeme také dosáhnout posunutím jednoho krajního bodu, směrem ku středu, ale pozor při této úpravě dojde ke ztrátě detailu v nejsvětlejší/nejtmavší části obrázku.

Obrázek 2.5 Zesvětlení obrázku se ztrátou detailů v nejsvětlejší části

Úprava kontrastu

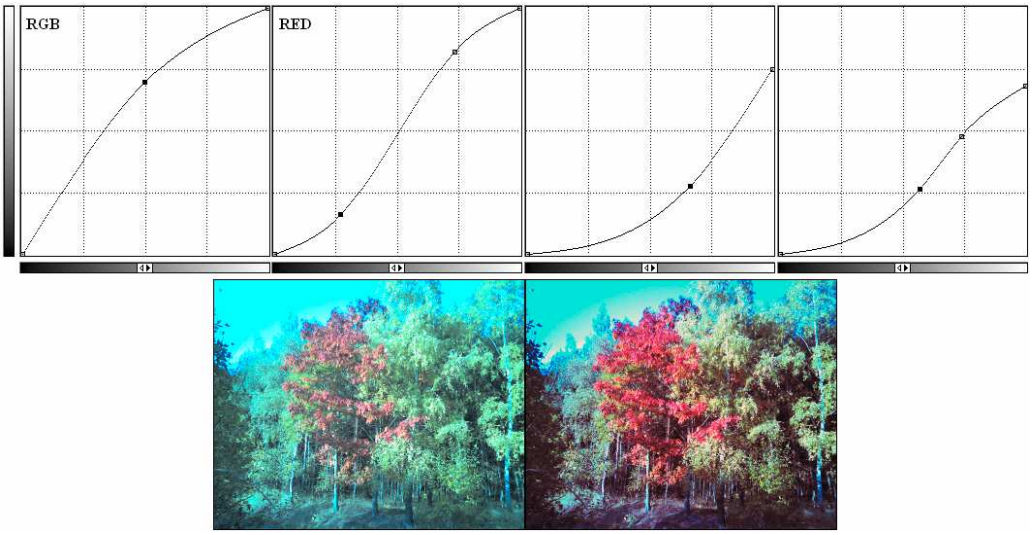
Pomocí křivek lze snadno zvýšit či snížit kontrast, a to pomocí úpravy jasu. Obecně se dá říci, že čím je křivka strmější, tím větší kontrast v daném úseku je. Chceme-li například zvýšit kontrast středních tónů, tak tmavší části ještě více ztmavíme a světlejší ještě zesvětlíme, což nám dá křivku v podobě S, zrovna jako na obrázku, naopak snížení kontrastu středních tónů je obráceně.

Obrázek 2.6 Zvýšení kontrastu ve středních tónech

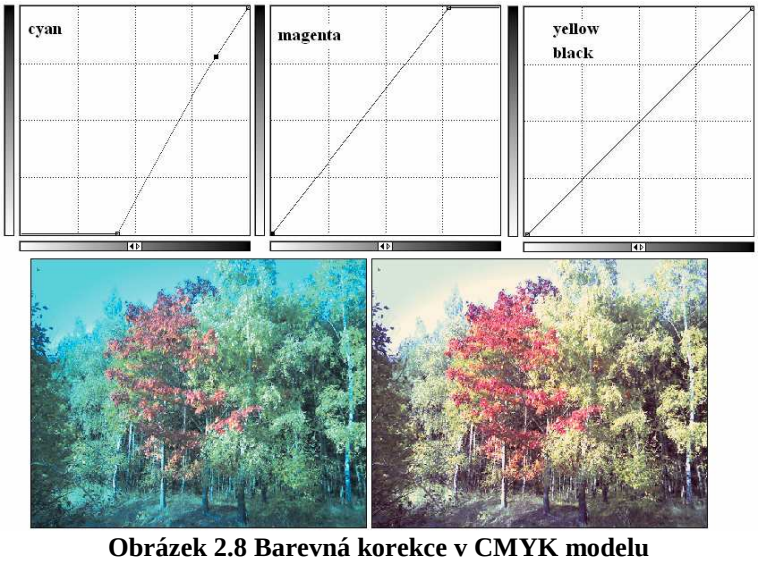
2.3.3 Barevné korekce

Toto je nejdůležitější použití křivek, pro nejrůznější barevné korekce, kdy uživatel upravuje jednotlivé barevné kanály a tím mění barevné podání obrázku. Bohužel je to také část nejnáročnější. Pokusil jsem se upravit obrázek pořízený mobilním telefonem, který má opravdu strašné barevné podání. Zde bych rád zmínil, že zdrojový obrázek (levý), jsem žádným způsobem neupravoval, takto hrozně byl integrovaným fotoaparátem opravdu pořízen. Tento obrázek jsem se pokusil upravit ve všech barevných modelech, abych tím dokázal své tvrzení, že křivky lze použít v každém z nich. Bohužel mi chybí praxe, ale přesto si myslím, že výsledné obrázky (pravé) jsou lepší než ty původní, ale konečné zhodnocení nechám raději na čtenáři.

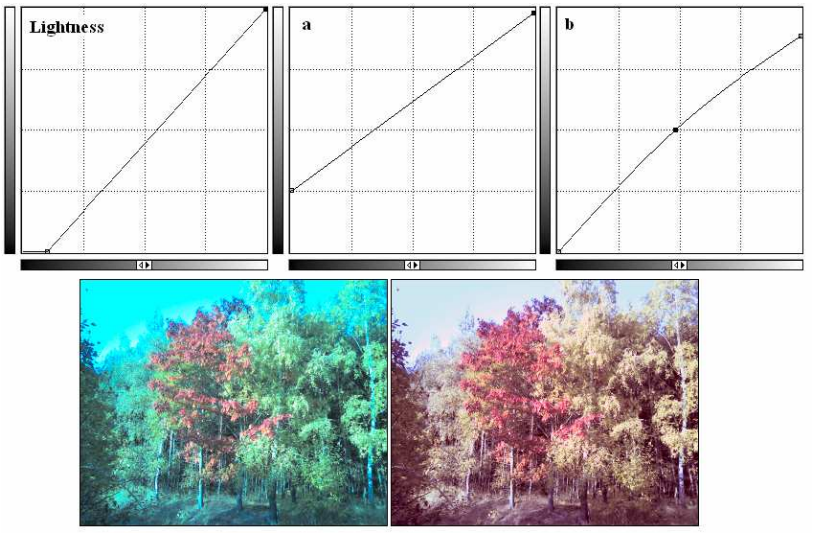
Jen pro doplnění uvedu ještě datum pořízení tohoto snímku, protože to je důležitá informace, která má vliv na barevné podání snímku. Snímek byl pořízen dne 15. října 2005.



Obrázek 2.7 Barevná korekce v RGB modelu



Obrázek 2.8 Barevná korekce v CMYK modelu



Obrázek 2.9 Barevná korekce v LAB modelu

Bystrý čtenář si jistě všiml, že zdrojový obrázek v CMYK modelu je jiný, než zdrojové obrázky v ostatních modelech, toto je způsobeno konverzí mezi RGB a CMYK modelem.

2.3.4 Další efekty

Pomocí křivek si můžeme zobrazit také jednotlivé barevné kanály v podobě odstínů šedi, čehož dosáhneme tak, že ostatní kanály stáhneme dolů, nebo můžeme obrázek proměnit v jednolitou barvu, pomocí vodorovné křivky. Tímto by se z křivek dala udělat například prolínačka do jedné barvy. Další možností je vytvoření hrbolaté křivky, čímž dosáhneme zajímavých „uměleckých“ efektů.

Obrázek 2.10 Křivky - umělecké efekty

3 VirtualDub SDK

Autor VirtualDubu vydal velice pěkné SDK pro psaní filtrů, které je možno stáhnou na domácích stránkách tohoto programu [4]. Toto SDK obsahuje několik funkčních ukázek, takže napsání jednoduššího filtru je otázka několika mála okamžiků.

Nyní je ten správný čas si říci, co to ten filtr vlastně je a jak se vytváří. Filtr je obyčejná dll knihovna (pro potřeby VirtualDubu přejmenovaná na *.vdf), která musí exportovat dvě předdefinované následující funkce:

```
int __declspec(dllexport) __cdecl VirtualdubFilterModuleInit2(FilterModule
*fm, const FilterFunctions *ff, int& vdfd_ver, int& vdfd_compat);
void __declspec(dllexport) __cdecl
VirtualdubFilterModuleDeinit(FilterModule *fm, const FilterFunctions *ff);
```

První funkce slouží k přidání filtru do seznamu filtrů dostupných v programu VirtualDub pomocí funkce addFilter z níže uvedené struktury FilterFunctions, dalším úkolem této funkce je vyplnit položky vdfd_ver, a vdfd_compat, které určují, jakou verzi hlavičkových souborů používáme a tím také jaké verze programu VirtualDub může náš filtr používat.

Druhá funkce slouží pro odebrání filtru pomocí funkce removeFilter opět ze struktury FilterFunctions, která má jako parametr strukturu FilterDefinition vrácenou nám funkcí addfilter.

Každý filtr pracuje s obrazem ve formátu RGB32, chceme-li tudíž ve filtru použít jiný barevný model, musíme provést konverzi tam a zase zpět. Filtr má také pouze jeden vstup a právě jeden výstup, což znamená, že každému příchozímu snímku musí odpovídat jeden odchozí snímek, což pro některé aplikace to může být problém, neboť nám to neumožňuje přístup k následujícím snímkům. Dále z tohoto faktu vyplývá, že nelze měnit počet snímků, tj. filtr nemůže snímky přidávat či odebírat.

Vzhledem k faktu, že program VirtualDub byl napsán v C++, tak se předpokládá, že filtr bude psán tamtéž a proto jsou součástí SDK také hlavičkové soubory *.h, které potřebuje pro tvorbu filtru. Hlavičkových souborů je 5, a jedná se o následující soubory:

```
Filter.h
ScriptError.h
ScriptInterpreter.h
ScriptValue.h
VBitmap.h
```

3.1 Co se skrývá v hlavičkových souborech

Z výše uvedených hlavičkových souborů je pro naši práci nejdůležitější ten první, tj. Filter.h, který obsahuje několik velice zajímavých struktur a tříd, se kterými se během práce na filtru setkáme.

3.1.1 Struktura FilterDefinition

Tuto je asi nejdůležitější struktura se kterou se během práce setkáme, musí jí totiž obsahovat každý filtr, neboť tato struktura patří mezi základní pilíře nového filtru. Možná si říkáte, proč je tak důležitá. Je to způsobeno tím, že říká VirtualDubu, jakým způsobem má filtr prezentovat uživateli, a také jakým způsobem s ním může pracovat.

```
typedef struct FilterDefinition {

  struct FilterDefinition *next, *prev;   pro interní potřebu VirtualDubu
  FilterModule *module;                   pro interní potřebu VirtualDubu

  char *              name;               jméno filtru
  char *              desc;               krátký popis filtru
  char *              maker;              autor filtru
  void *              private_data;       pro interní potřebu VirtualDubu
  int                 inst_data_size;     velikost dat instance filtru

  FilterInitProc      initProc;     fce pro inicializaci dat filtru
  FilterDeinitProc    deinitProc;   fce pro vyčištění inicializovaných dat
  FilterRunProc       runProc;      fce provádějící vlastní funkci filtru
  FilterParamProc     paramProc;    fce poskytující informace pro VirtualDub
  FilterConfigProc    configProc;   fce pro zobrazení konfiguračního okna
  FilterStringProc    stringProc;   fce zobrazující krátké info o filtru
  FilterStartProc     startProc;    fce volaná před vlastním použitím filtru
  FilterEndProc       endProc;      fce volaná po použití filtru

  CScriptObject   *script_obj;      Sylia script object
  FilterScriptStrProc fssProc;      fce pro uložení filtru do Jobu
```

od verze 1.4.11 byly přidány následující položky:

```
  FilterStringProc2     stringProc2;      viz. FilterStringProc
  FilterSerialize       serializeProc;    fce pro uložení stavu filtru
  FilterDeserialize     deserializeProc;  fce pro načtení stavu filtru
  FilterCopy            copyProc;         fce klonuje nastavení filtru

} FilterDefinition;
```

Na první pohled je vidět, že struktura je to opravdu rozsáhlá a proto jsem doplnil ke každé položce krátký popisek, k čemu daná položka vlastně slouží. Červeně označené položky musí být u každého filtru vždy vyplněny, u ostatních položek můžeme použít NULL, samozřejmě kromě položky inst_data_size, kde použijeme hodnotu 0, nepoužíváme-li žádná data, jinak samozřejmě jejich velikost.

Všechny *Proc položky jsou vlastně ukazatele na funkce, které VirtualDubu poskytujeme, na těchto funkcích záleží možnosti našeho filtru, čím více jich naimplementujeme, tím větší možnosti bude mít náš filtr, přičemž jak již bylo řečeno výše jediná povinná funkce je funkce s prototypem jménem FilterRunProc. Funkce mají následující prototypy:

```
int  FilterInitProc(FilterActivation *fa, const FilterFunctions *ff);
void FilterDeinitProc(FilterActivation *fa, const FilterFunctions *ff);
int  FilterRunProc(const FilterActivation *fa, const FilterFunctions *ff);
long FilterParamProc(FilterActivation *fa, const FilterFunctions *ff);
int  FilterConfigProc(FilterActivation *fa, const FilterFunctions *ff, HWND
hWnd);
void FilterStringProc(const FilterActivation *fa, const FilterFunctions
*ff, char *buf);
int  FilterStartProc(FilterActivation *fa, const FilterFunctions *ff);
int  FilterEndProc(FilterActivation *fa, const FilterFunctions *ff);
bool FilterScriptStrProc(FilterActivation *fa, const FilterFunctions *,
char *, int);
void FilterStringProc2(const FilterActivation *fa, const FilterFunctions
*ff, char *buf, int maxlen);
int  FilterSerialize(FilterActivation *fa, const FilterFunctions *ff, char
*buf, int maxbuf);
void FilterDeserialize(FilterActivation *fa, const FilterFunctions *ff,
const char *buf, int maxbuf);
void FilterCopy(FilterActivation *fa, const FilterFunctions *ff, void
*dst);
```

3.1.2 Třída FilterActivation

Instance této třídy je poskytována, každé funkci a obsahuje především ukazatel na naše data potřebná pro funkci filtru, těmito daty může být buď struktura nebo objekt, mezi další důležité položky patří ukazatele na zdrojovou a cílovou bitmapu. Celá definice je následující:

```
class FilterActivation {
public:
    FilterDefinition *filter;
    oid *filter_data;                  ukazatel na naše data
    VFBitmap &dst, &src;               ukazatelé na bitmapy
    VFBitmap *__reserved0, *const last;
    unsigned long x1, y1, x2, y2;

    FilterStateInfo *pfsi;             třída s informacemi o stavu filtru
    IFilterPreview *ifp;               rozhraní pro náhled filtru

    FilterActivation(VFBitmap& _dst, VFBitmap& _src, VFBitmap *_last) :
dst(_dst), src(_src), last(_last) {}
    FilterActivation(const FilterActivation& fa, VFBitmap& _dst, VFBitmap&
_src, VFBitmap *_last);
};
```

3.1.3 Třída FilterStateInfo

Tato struktura obsahuje nejrůznější informace o právě zpracovávaném snímku videa.

```
class FilterStateInfo {
public:
    long    lCurrentFrame;              číslo aktuálního výstupního snímku
    long    lMicrosecsPerFrame;         počet mikrosekund cílového snímku
    long    lCurrentSourceFrame;        číslo aktuálního zdrojového snímku
    long    lMicrosecsPerSrcFrame;      počet mikrosekund zdrojovéhosnímku
    long    lSourceFrameMS;             timestamp zdrojového snímku
    long    lDestFrameMS;               timestamp cílového snímku
};
```

3.1.4 Struktura FilterFunctions

Tato struktura obsahuje nejrůznější funkce, které nám poskytuje přímo VirtualDub, s prvními dvěmi funkcemi jsme se již setkali výše, jsou to funkce pro přidání/odebrání filtru do VirtualDubu.

```
struct FilterFunctions {
    FilterDefinition *(*addFilter)(FilterModule *, FilterDefinition *, int
fd_len);
    void (*removeFilter)(FilterDefinition *);
    bool (*isFPUEnabled)();
    bool (*isMMXEnabled)();
    void (*InitVTables)(struct FilterVTbls *);
```

Předchozí funkce nám umožňují vyvolání výjimky MyError, ale zároveň mohou být volány pouze v těchto funkcíh: runProc, initProc a startProc.

```
    void (*ExceptOutOfMemory)();             od verze VirtualDubu 1.4
    void (*Except)(const char *format, ...);  od verze VirtualDubu 1.4
```

Následující funkce můžeme volat kdykoli:

```
    long (*getCPUFlags)();                   od verze VirtualDubu 1.4
    long (*getHostVersionInfo)(char *buffer, int len);    od verze 1.4d
};
```

3.2 Další použití filtrů

Filtry pro VirtualDub je možno použít nejen přímo v programu VirtualDub, ale i ve spoustě jeho modifikací, jejichž název většinou slovo Dub obsahuje, příkladem nám může být například program NanDub, v těchto klonech není s filtry žádný problém. Ten může nastat až při používání filtrů v programu Avisynth, který nepodporuje všechny možnosti, jako VirtualDub, chceme-li proto psát univerzální filtry, musíme na to dávat pozor, dobrou zprávou je, že VirtualDub SDK obsahuje upozornění na případné nekompatibility.

4 Vlastní tvorba filtru

První základní otázkou bylo, v jakém jazyce se filtr pokusím naimplementovat, nejdříve jsem doufal v Borland Delphi, a to hlavně kvůli tvorbě GUI, ale po důkladnějším rozvážení a hlavně pohlédnutí hlavičkových souborů jsem tuto variantu zavrhl. A tak jsem se rozhodl použít jazyk autora VirtualDubu a to C++, zároveň s jeho vývojovým prostředím jímž je Visual Studio.

Protože dnes světem vládne objektově orientované programování, tak jsem se rozhodl používat objekty, toto mé rozhodnutí mi přineslo několik problémů, které se naštěstí vyřešily po důsledném přečtení SDK. Jde o to, že při opakovaném spuštění konfigurace filtru dojde k vytvoření nové instance objektu, který představuje data filtru, to znamená, že u tohoto objektu musíme vytvořit konstruktor, který jej zduplikuje.

Během práce jsem narazil na čtyři hlavní okruhy problémů, prvním se týkal křivek, a to od základního problému jak uchovávat jednotlivé body až po problém vygenerování křivky, další sadou problémů se stala tvorba oblastí, tj. úseků videa, které budou obsahovat vlastní křivky, třetím, ale ne zanedbatelným problémem se stalo GUI, neboť tento nástroj je celý postaven na grafickém ovládání a posledním problémem bylo, které funkce uvedené ve FilterDefiniton struktuře implementovat.

4.1 Tvorba křivky

Pro řídící bod jsem si vytvořil třídu FixedPoint, která obsahuje pouze vstupní a výstupní hodnotu bodu a několik užitečných funkcí, které se týkají především vykreslování křivky. Hned nato nastal první problém a to jak uchovávat řídící body, tvořící jednotlivé křivky? Nejprve jsem uvažoval o obyčejném poli, kde bych měl pro každou barvu předem připraveno pole o určitém počtu bodů, neboť tento počet je velmi jasně omezen. V mnoha programech je tímto omezením pouhých 14 bodů, nebo bych mohl uživateli například povolit vložit každý bod extra což by představovalo maximálně 256 bodů, což by byla velikost mého pole, ale při představě 256 bodů pro každou barvu, které jsou 4 a pro x oblastí to dělá dosti velké paměťové požadavky.

Nehledě na další problém, kterým je vlastní pohyb bodu uživatelem, tento bod se nesmí dostat před bod předcházející ani na jeho stejnou vstupní souřadnici (v některých programech je toto řešeno, že při takovémto přetažení bodu bude tento odstraněn), já jsem to vyřešil pouze tak, že bodem lze pohnout jen na určitou vzdálenost od bodu předchozího či následujícího. Je vidět, že se nám to pěkně komplikuje a to jsme ještě neřešili problém odebrání bodu, neboť

potřebuji mít body vždy seřazeny za sebou a to kvůli generování křivky, tento problém by se dal snadno řešit při poli pro 256 prvků, ale zase je tu ta jeho velikost.

Proto jsem se rozhodl udržovat body ve dvoucestném seznamu a to ze dvou důvodů, za prvé kvůli paměťové náročnosti vůči poli s 256 prvky a také z důvodu snadného přístupu k předchozímu a následujícímu bodu, přičemž jde také velice snad identifikovat, že bod je krajní. Kvůli tomuto rozhodnutí jsem modifikoval třídu FixedPoint a přidal jsem do ní ukazatele na předchozí a následující prvek, přičemž samotný seznam jsem implementoval jako třídu s názvem FixedPointDoubleLinkedList, která zapouzdřuje všechny důležité funkce, jako je přidávání prvku, odebrání prvku, a mnoho dalších funkcí. Tímto jsem vytvořil úložiště pro základní body křivek, tyto úložiště jsem vložil do objektu CurveSegment a tímto se přesuneme do následující části.

4.2 Tvorba oblastí

Jak již bylo řečeno výše, oblasti jsou uchovávány ve třídě CurveSegment, přičemž zde nastal opět problém uchovávání těchto tříd neboť oblastí může být několik. Tentokrát jsem se dlouho nerozmýšlel a rovnou jsem sáhl k dvojcestnému seznamu a to především z důvodu jeho implementace v předchozí kapitole. Tato volba se ukázala být velice dobrou, neboť při aktivním náhledu filtru se může uživatel posouvat kam se mu chce a toto mi šetří čas prohledávání celého seznamu, neboť se posouvám jen jedním směrem. To znamená, že jsem třídu CurveSegment opět doplnil o několik dalších vlastností jako jsou ukazatelé na předchozí a následující segment, ukazatelé na aktuální bod a aktuální křivku a několik dalších položek. Dvojcestný seznam byl opět implementován pomocí třídy s výstižným názvem CurveSegmentDoubleLinkedList. Tato třída v sobě opět zahrnuje spoustu funkcí od obyčejné správy prvků až po uložení třídy do řetězce.

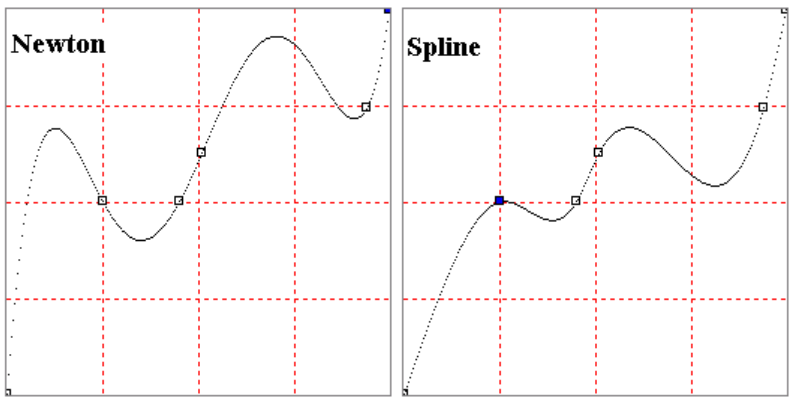
Zde musím zmínit, že přidávání prvků, tj. segmentů, nebylo zdaleka tak jednoduché jako implementovat přidávání bodů, neboť každá oblast má nějaký začátek a konec a je třeba ji správně vložit tak aby se jednotlivé oblasti v žádném případě nepřekrývaly a přitom šly pěkně za sebou. Zde jsem musel řešit velké množství problémových situací, například nově vkládaná oblast zasahuje do existující, nebo nová oblast je uvnitř existující oblasti a nejhorší případ, kdy nová oblast je přes několik oblastí a další částečně zasahuje. Naštěstí se mi toto podařilo na konec vyřešit, takže zadávání nové oblasti funguje tak, že v případě, že se oblasti překrývají, dojde k oříznutí původní oblasti tak, aby nová oblast odpovídala přesně vstupu uživatele. S vkládáním oblastí také souvisí problém změny oblasti, tu jsem ale vyřešil velice

šalamounsky, neboť vytvořím novou oblast obsahující stejné řídící body křivek jako oblast měněná, tu odstraním a duplikovanou oblast vložím, čímž dojde ke správnému oříznutí případných konfliktů.

Vzhledem k faktu, že VirtualDub je schopen uchovat pro filtr pouze jeden objekt či strukturu, tak jsem si vytvořil ještě jednu třídu s názvem FilterData, což se může zdát zbytečné, protože by mě mohlo bohatě stačit doplnit třídu CurveSegmentDoubleLinkedList o potřebné položky, ale pro lepší přehlednost jsem to udělal takto. Třída FilterData tedy obsahuje CurveSegmentDoubleLinkedList, několik položek a dvě velice zajímavá pole bodů která obsahující vlastní křivky a to jak křivky pro zobrazení na obrazovce tak křivky pro filtrování. Zde by mohlo dojít k malému zmatku a proto to raději vysvětlím. Udržuji za prvé pole o 1024 prvcích, které slouží pro vykreslování křivky v nástroji křivky, a pole o 768 prvcích, které se použije při filtrování. Možná se divíte proč je jedno pole větší, je to tím, že pro nástroj potřebuji čtyři křivky a to RGB, R, G, B, kdežto pro filtr potřebuji jen výsledné křivky pro jednotlivé barvy tj. R, G, B. Ha, říkáte si, jaké výsledné křivky? Během zkoumání nástroje křivky v různých programech jsem vyzkoumal, že nejdříve se aplikují křivky na jednotlivé barevné kanály (R, G, B) a poté až křivka RGB, tzn. že červená filtrovací křivka je vytvořena jako RGB obraz červené složky. V kódu by to vypadalo takto:

```
redFilterRow[i] = rgb[redRow[i]].
```

Pro čtenáře zajímající se o vlastní implementaci výpočtu křivky mám dobrou zprávu, je to tady. Jak jsem zmínil v teoretické části, jsou v podstatě dva způsoby tvorby křivky, a to buď pomocí Newtonovy metody a nebo pomocí splinu. Po porovnání vlastností těchto metod a po prozkoumání nástroje křivky v několika programech je na 99,99% jasné, že se využívá spline. Přesto mi to nedalo a implementoval jsem jak Newtona tak spline a ačkoliv by se mohlo zdát, že pro tyto účely bude Newtonova metoda postačující, neboť počet použitých bodů bývá velice nízký, tak opak je pravdou jak je dobře vidět na následujícím obrázku.



Obrázek 4.1 Porovnání Newtonovi a Spline interpolace

4.3 GUI

Po vytvoření všech objektů přišla chvíle pro vytvoření uživatelského rozhraní, vzhledem k faktu, že program je psán v C++ a jedná se o DLL knihovnu, tak jsem byl nucen psát celý dialog ve WinApi. Tvorba dialogu by se dala rozdělit do dvou částí, a to vytvoření vlastního dialogu a jeho zprovoznění. První část je velice jednoduchá a rychlá neboť Visual Studio má docela pěkný resource editor, ve kterém člověk snadno vytvoří dialog podle svého přání, neboť se jedná o vizuální editor.

Horší je to s druhou částí neboť mít dialog plný kontrolek, které vůbec nereagují na uživatelský vstup, není to pravé ořechové. Bohužel ve WinApi nejsou události tak jak jsme zvyklí z vyšších programovacích jazyků, jako je například C#, Delphi, kde máme nejrůznější události, například událost OnClick. Ve WinApi jsou veškeré události řešeny pomocí zpráv, které nám zasílá operační systém a na nás je, jakým způsobem na ně budeme reagovat. Na druhou stranu pomocí zpráv také nastavujeme vlastnosti jednotlivých kontrolek. Pro zpracování zpráv nám slouží callback funkce, kterou jsme předali jako parametr při vytváření dialogu. Tato funkce musí reagovat na velké množství zpráv, které značí nejrůznější události a to od kliknutí na tlačítko přes pohyb myši až například po změnu v editu či listboxu.

Nebudu zde popisovat všechny zprávy, neboť program jich na spoustu reaguje a ještě na víc nereaguje. Nehledě k faktu, že tyto informace se dají velice dobře dohledat ať již v přiložených zdrojových kódech, či například na MSDN nebo bývají zmíněny v jakémkoliv tutoriálu na Internetu, například zde[5]. Raději se zde proto zmíním jen o tom z mého pohledu nejdůležitějším, což jsou věci, které mi dalo velkou práci zjistit. Dále bych na tomto místě rád upozornil, že zde nebudu uvádět plnou syntaxi těchto konstrukcí, neboť toto není WinApi tutoriál a při znalosti jména příkazu či zprávy lze informace snadno a rychle vyhledat na internetu či v literatuře.

První pro mne problematickou věcí se stalo, že když máme dialog a na něm více kontrolek, tak veškeré zprávy přijímá jen dialog, ale my bychom potřebovali například získávat vstup klávesnice nad jednou určitou kontrolkou. Toto se zařídí celkem jednoduše, a to tak, že si vytvoříme novou callback funkci pro tuto kontrolku a zaregistrujeme ji pomocí příkazu SetWindowLongPtr a parametru GWL_WNDPROC, když už jsme u toho, tak se nám může stát, že budeme potřebovat všechen vstup z klávesnice a Windows ne a ne nám ho poskytnout, tak toto vyřešíme také velice snadno a to správnou odpovědí na zprávu WM_GETDLGCODE.

Když už máme všechen vstup takto zajištěný, tak se může stát, že potřebujeme něco vykreslovat, pro mě je to v podstatě graf, ale může to být cokoliv. Po pracné implementaci pomocí základních příkazů zjistíme k naší velké nelibosti, že nám naše WinApi grafika, u které předpokládáme, že by měla být velice rychlá, problikává, ale co s tím? Samozřejmě, že je zde řešení, i když není tak jednoduché jako v předchozím případě s klávesnicí. Musíme použít doublebuffering, který je v novějších jazycích řešen v podobě property. A tak tedy doubleffering vyřešíme tak, že naši grafiku vytvoříme na bitmapu, kterou poté teprve vykreslíme. Tento postup je trochu složitější a proto zde uvedu celý kód:

```
PAINTSTRUCT x;
HDC hDC, memDC;
HBITMAP memBM, oldBM;

DC = BeginPaint(hwnd, &x);
emDC = CreateCompatibleDC(hDC);
emBM = CreateCompatibleBitmap(hDC, CurveWindowSize, CurveWindowSize);
ldBM = (HBITMAP)SelectObject(memDC, memBM);
```

vlastní kód kreslení na memDC

```
BitBlt(hDC, 0, 0, CurveWindowSize, CurveWindowSize, memDC, 0, 0, SRCCOPY);
SelectObject(memDC, oldBM);
DeleteObject(memBM);
DeleteDC(memDC);
EndPaint(hwnd, &x);
```

Nyní když víme, jak napsat neproblikávající aplikaci můžeme narazit na problém, že bychom potřebovali u kontrolky odkaz na nějaká vlastní data, například při odpovědi na zprávu WM_PAINT, tj. při vykreslování, ale jak je tam dostat? Řešení je opět velice jednoduché, neboť naštěstí každá kontrolka má možnost uložení uživatelských dat a to pomocí již známého příkazu SetWindowLongPtr, ale pozor, je nutno zadat různé parametry při ukládání dat do kontrolky (GWLP_USERDATA) či celého dialogu (DWLP_USER).

4.4 FilterDefinition aneb co implementovat

V kapitole týkající se SDK VirtualDubu jsme si ukázali strukturu FilterDefinition, kterou musí obsahovat každý filtr, zároveň jsme si také uvedli prototypy funkcí, které tato struktura obsahuje. Pozorný čtenář si jistě pamatuje, že stačí implementovat pouhou jednu funkci a to funkci RunProc, což je vlastní filtrovací funkce. Bohužel při implementaci pouze této jediné funkce by mohla být funkčnost filtru velice omezená a proto je v podstatě potřeba implementovat téměř všechny funkce. Pro důkaz tohoto tvrzení jsem na následující stránku umístil vyplněnou strukturu FilterDefinition z mého filtru.

```
typedef struct FilterDefinition {

  NULL, NULL,               pro interní potřebu VirtualDubu
  NULL,                     pro interní potřebu VirtualDubu

  "Curves",                 jméno filtru
  "Another Curves Filter.", krátký popis filtru
  "Jan Kučera",             autor filtru
  NULL,                     pro interní potřebu VirtualDubu
  sizeof(FilterData),       velikost dat instance filtru

  CurvesInitProc,           fce pro inicializaci dat filtru
  CurvesDeinitProc,         fce pro vyčištění inicializovaných dat
  CurvesRunProc,            fce provádějící vlastní funkci filtru
  NULL,                     fce poskytující informace pro VirtualDub
  CurvesConfigProc,         fce pro zobrazení konfiguračního okna
  CurvesStringProc,         fce zobrazující krátké info o filtru
  CurvesStartProc,          fce volaná před vlastním použitím filtru
  CurvesEndProc,            fce volaná po použití filtru

  &curves_obj,              Sylia script object
  CurvesFssProc,            fce pro uložení filtru do Jobu
```

od verze 1.4.11 byly přidány následující položky:

```
  CurvesStringProc2,        viz. FilterStringProc
  NULL,                     fce pro uložení stavu filtru
  NULL,                     fce pro načtení stavu filtru
  CurvesCopyProc            fce klonuje nastavení filtru

} FilterDefinition;
```

U zde uvedené struktury je vidět, že kromě funkcí na uložení a načtení stavu filtru jsem implementoval všechny, tyto dvě funkce jsem neimplementoval, protože mají v budoucnosti sloužit k uložení/načtení aktuálního stavu filtru.

5 Uživatelská příručka

V této kapitole si ukážeme jak filtr dostat do programu VirtualDub a také jakým způsobem ho ovládat. Obě tyto akce jsou velice jednoduché a tak k nim můžeme bez dalšího zbytečného vysvětlování přejít.

5.1 Zprovoznění filtru ve VirtualDubu

Ke zprovoznění filtru ve VirtualDubu máme dvě možnosti, přičemž obě jsou velice jednoduché a zvládne je opravdu každý uživatel.

První možnost spočívá ve zkopírování filtru do adresáře Plugins v adresáři programu VirtualDub, tato možnost je vhodná především tehdy, když budeme filtr chtít používat častěji, neboť filtr bude načten a přidán do seznamu použitelných filtrů automaticky při každém startu programu VirtualDub.

Druhá možnost je ta, že v dialogu Add Filter zvolíme možnost Load, jestliže používáme jinou jazykovou mutaci, tak se názvy mohou trochu lišit, a ve standardním dialogu otevřít jej vybereme.

5.2 Ovládání filtru

Vzhledem k faktu, že se u filtru předpokládá zveřejnění na internetu, tak jsem vytvořil GUI v anglickém jazyce. Věřím že toto nebude na překážku, neboť GUI filtru je podle mého názoru natolik intuitivní, že by případné lokalizace byly zcela zbytečné.

Okno filtru je rozděleno do tří samostatných částí, přičemž jedna část je věnována ovládání samotné křivky, druhá část je věnována úsekům videa, kde mohou být různé křivky a za třetí část můžeme považovat spodní lištu tlačítek.

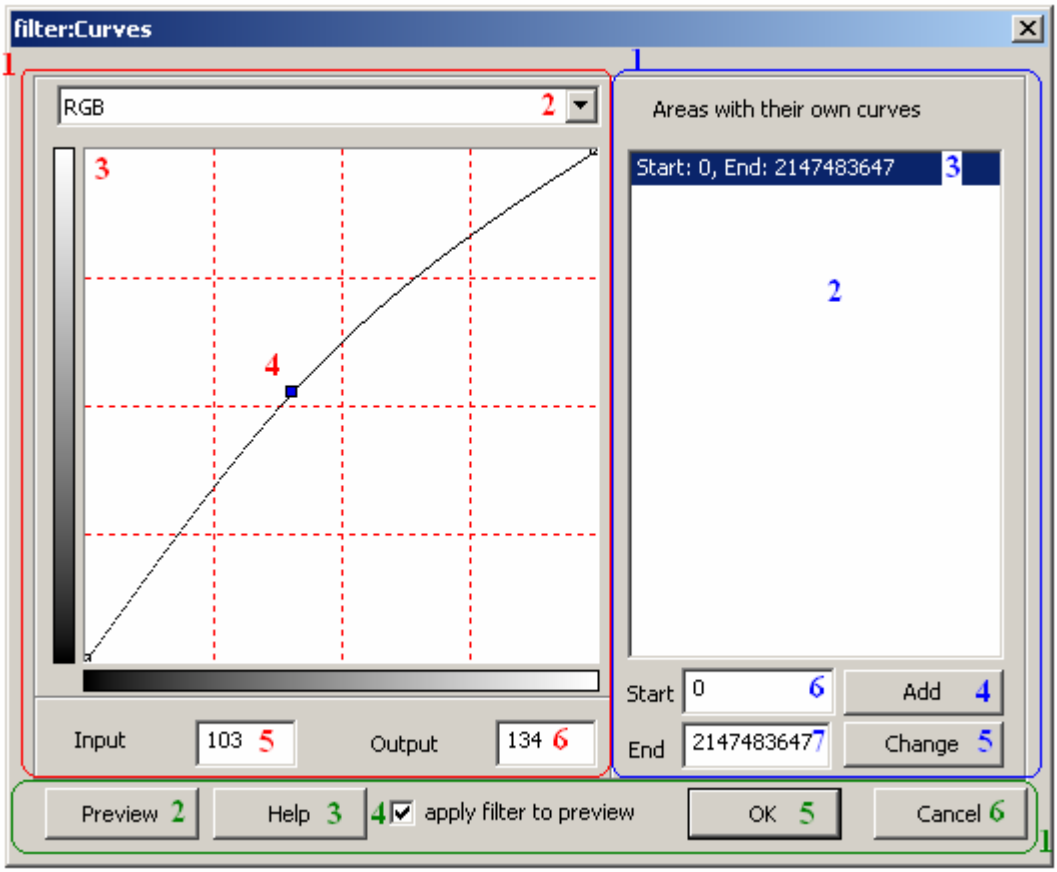
Na následující stranu jsem umístil screenshot okna filtru, na kterém jsem barevně označil všechny tři výše uvedené oblasti. Celá oblast má vždy číslo jedna a každý významný ovládací prvek, má číslo následující, pojďme si je nyní popsat důkladněji.

5.2.1 Část křivky

Jak již nadpis prozrazuje, toto je hlavní část našeho filtru, neboť právě zde uživatel upravuje křivku, která bude ovlivňovat výsledný obraz. Tato část je označena červeně.

1. Celá část křivky
2. Rozbalovací menu, obsahující jednotlivé barevné kanály
3. Vlastní oblast křivky, sem se dají přidávat pevné body kliknutím levým tlačítkem myši

4. Vybraný bod, s tímto bodem můžeme pohybovat a to ať myší nebo klávesnicí pomocí kurzorových šipek, nebo jej můžeme odstranit stiskem klávesy Delete, z bodu se lze také přepnout na následující bod pomocí klávesy Tab
5. Editační pole zobrazující nám zdrojovou hodnotu, tuto hodnotu lze také přepsat, v případě že není vybrán žádný bod, je toto pole neaktivní a zobrazuje pozici kurzoru myši nad oblastí křivek
6. Jako předchozí jen s tím rozdílem, že zde vidíme cílovou hodnotu



Obrázek 5.1 Screenshot dialogu filtru

5.2.2 Část oblastí

Toto je velice zajímavá část se kterou se v souvislosti s nástrojem křivky asi běžně nesetkáme, jde o to, že se nám může stát, že potřebujeme upravit jen nějaký úsek videa, a protože VirtualDub neumožňuje klipy, ale aplikuje vždy filtr na celý snímek najednou, tak toto je řešení. Tato část je označena modře.

1. Celá část oblastí
2. Seznam vytvořených oblastí, kliknutím lze změnit aktuální oblast, jejíž křivka se nám zobrazuje v části křivky
3. Aktuální oblast
4. Tímto tlačítkem přidáme oblast, která bude od snímku uvedeného v poli 6 až po snímek v poli 7

5. Změna délky oblasti podle hodnot v 6, 7
6. Editační pole určující nám počáteční snímek, na který se bude filtr aplikovat, pozor VirtualDub čísluje snímky od 0
7. Editační pole určující do kterého snímku včetně se bude filtr aplikovat, číslo zde uvedené je maximální počet snímků podporovaný tímto filtrem

5.2.3 Tlačítková lišta

Toto je velice zajímavá část se kterou se v souvislosti s nástrojem křivky asi běžně nesetkáme, jde o to, že se nám může stát, že potřebujeme upravit jen nějaký úsek videa, a protože VirtualDub neumožňuje klipy, ale aplikuje vždy filtr na celý snímek najednou, tak toto je řešení. Tato část je označena zeleně.

1. Celá část tlačítek
2. Tlačítko pro zobrazení/skrytí náhledu během úprav křivky
3. Toto tlačítko zobrazí krátkou hypertextovou nápovědu, pokud tato existuje v adresáři ve kterém je filtr nakopírovaný
4. Zaškrtávací políčko pro změnu náhledu z původního na upravené video
5. Toto tlačítko ukončí dialog, přičemž budou v nastavení filtru uloženy veškeré změny zde provedené
6. Toto tlačítko ukončí dialog a v závislosti na předchozím stavu provede jednu z následujících akcí, byl-li již filtr nastaven dříve, tak se toto nastavení zachová, jinak dojde k odstranění filtru ze seznamu filtrů

6 Závěr

Úkolem této práce bylo vytvoření implementace nástroje křivky v podobě filtru pro program VirtualDub. Musím říci, že tato implementace se opravdu povedla a filtr je nejen funkční, ale i celkem rychlý, neboť je jen zhruba o dvě procenta pomalejší než interní filtry. Rychlost jsem měřil tak, že jsem pomocí křivek udělal invertaci barev a porovnal jsem ji s interním filtrem invert. Pro úplnost doplním informace o videu, jednalo se o video o rozlišení 576*432 pixelů a délka vida byla 10292 snímků. Výsledky jsou vidět v následující tabulce číslo 6.1.

	Filtr (čas v sekundách)		Rozdíl	
	Křivky	Invertace	v sekundách	v procentech
1. měření	733,315	717,913	15,402	2,15%
2. měření	733,074	717,692	15,382	2,14%
3. měření	733,224	718,103	15,121	2,11%
Průměr	733,204	717,903	15,301	2,13%

Tabulka 6.1 Srovnání filtru Křivky s integrovaným filtrem invertace barev

Přemýšlivý čtenář může namítat, zda-li nebude filtr pomalejší při použití nějaké složitější křivky. Na toto musím odpovědět nikoli, neboť při vlastní aplikaci filtru již na tvaru křivky nezáleží, jediné co může způsobit velice malý rychlostní propad je velké množství oblastí.

Literatura

[1] Mošová, Vratislava. *Numerické metody* [on-line]. Olomouc 2003 Dostupný z URL: <http://www.inf.upol.cz/skripta/texty/numericke%20metody.pdf>

[2] Přikryl Petr; Brandner Marek. *Numerické metody II*, 1. vydání Plzeň: Západočeská univerzita, 2000. ISBN 80-7082-699-1

[3] Margulis, Dan. *Professional Photoshop*. 4th edition, Wiley Publishing, 2002. ISBN 0-7645-3695-8

[4] Lee, Avery. *Welcome to virtualdub.org! - virtualdub.org* [on-line]. 2004 Dostupný z WWW: <http://www.virtualdub.org>

[5] Chalupa, Radek *Učíme se WinAPI* [on-line]. 2002 Dostupný z WWW: <http://www.builder.cz/serial91.html>