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

Bakalářská práce

Doostřování obrázků

Plzeň, 2007 Milan Pixa

Abstrakt

Bakalářská práce se zabývá tvorbou pluginu pro aplikaci Adobe Photoshop, který bude implementovat algoritmus pro doostřování obrázků, ve vývojovém prostředí Microsoft Visuál C++ s použitím Adobe Photoshop SDK 6.0. Práce je rozdělena do dvou částí. V první se věnuje popisu a objasnění pojmů spjatých s doostřováním obrázků. Druhá část práce je zaměřená na tvorbu pluginu, především na prozkoumání tvorby uživatelského rozhraní realizovaného pomocí Win32Api.

Abstrakt

Bachelor thesis deals with the creation of plug-in for Adobe Photoshop application, that shall implement the unsharp mask algorithm, in the developing environment Microsoft Visual C++ using Adobe Photoshop SDK 6.0. The thesis is divided into two parts.The first part characterizes and clarifies image sharpening. The second part of the thesis is focused on the inspection of creating plug-in and it is specially focused on the reconnaissance of the user's creation interface realized in Win32Api.

Obsah

Obsah

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

V Plzni dne 3. května 2007

…….

Milan Pixa

1. Úvod

Na začátku práce se nejprve budu zabývat problematikou doostřování. Tomuto tématu jsou věnovány kapitoly 2. až 4. V nich se seznámíte s algoritmem použitým pro doostřování, který pro doostření využívá zvýrazňování hran obrázku. Nakonec se v této části zmíním, dle mého názoru o hlavním problému, ke kterému dochází při použití níže popsaného algoritmu pro doostření, čímž jsou okrajové pixely obrázku a popíši možná řešení.

V další části nastíním něco málo o pluginech pro Adobe Photoshop. Podrobně rozeberu uživatelské rozhraní pluginu. Zde jsem se zaměřil na implementaci rozhraní ve Win32Api s pomocí „rc“ šablon a na popis základních ovládacích prvků, které jsem použil ve své práci, spolu s jejich ovládáním.

Nakonec bude následovat uživatelská příručka pro vzniklý plugin.

2. Doostřování

Na začátek je nutno zdůraznit, že doostřování je jen matematickou operací nad daným obrazem a nemůže tedy přinést jakékoliv nové detaily obrazu, které v původním obrazu nebyly. Doostření tak vždy vede k degradaci obrazu. To vyvolává otázku proč se tedy doostřování vůbec provádí? Důvod je velmi prostý, správně doostřený obrázek působí na lidského pozorovatele subjektivně lépe a snáze mu porozumí, čehož si můžete všimnout na obrázcích 2.1 a 2.2.

Obr. 2.1. Původní obrázek před doostřením.

Obr. 2.2. Obrázek po doostření.

3. Algoritmus doostření

V programu je použit algoritmus, který nejprve rozostří obrázek, tak že rozostřené pixely neukládá zpět do původního, ale do pomocného obrázku, čímž vytvoří jeho rozostřenou kopii. Jak samotné rozostření funguje, bude popsáno v následujících odstavcích. V dalším kroku algoritmu se provede rozdíl pixelů původního a rozostřeného obrázku, pro získání obrazu obsahujícího pouze „detaily“ původního obrázku. Nakonec je tento rozdíl přičten k původnímu obrázku a tím se získá požadovaný doostřený obrázek.

Při rozostření obrázku používám Gaussovské rozostření, proto nejdříve musíme podle Gaussova vzorce:

$$\frac{1}{2\pi\sigma^2}\exp^{-\frac{x^2+y^2}{2\sigma^2}}$$

Vzorec 3.1. Gaussův vzorec k výpočtu hodnot pro Gaussovu matici.

vytvořit a vyplnit Gaussovu matici, kterou poté budeme aplikovat na matici obrázku pro rozostření. K tomu nejprve musíme z uživatelského rozhraní zjistit hodnotu parametru **σ** (sigma), podle kterého vypočítáme poloměr Gaussovy matice. Avšak samotná znalost velikosti sigmy není pro výpočet postačující, je potřeba znát také hodnotu krajních pixelů. Tu jsem si stanovil na 0,00001 pomocí několika experimentálních výpočtů rozostření pixelu, tak aby mi tato hodnota ovlivnila výsledek pro rozostření, při rozsahu pixelů obrázku od 0 do 255.

Jestliže máme výslednou velikost poloměru, můžeme vyplnit Gaussovu matici, tak že ve středu matice dosazujeme za X a Y hodnotu POLOMĚR-1 a směrem ke krajům matice se hodnoty pro X a Y snižují o jedničku.

Po vytvoření Gaussovy matice můžeme přistoupit k samotnému rozostření. Vezměme si pixel obrázku na pozici [X, Y] ten vynásobíme středem Gaussovy matice a postupně k této hodnotě budeme přičítat okolní pixely vynásobené příslušnými hodnotami z Gaussovy matice. Po přičtení všech pixelů spadajících pod poloměr Gaussovy matice, je třeba získanou hodnotu vydělit součtem hodnot z Gaussovy matice a tím získáme výslednou hodnotu pro rozostřený pixel [X, Y], který si uložíme do matice pro rozostřený obrázek na stejnou pozici, jakou měl v původním obrázku. Pro lepší pochopení ukáži výpočet na příkladu (*Obr. 3.1*):

Gaussova matice:

a	b	c
d	s	e
f	g	h

dělitel = a + b + c + d + s + e + f + g + h

```
Výsledný pixel: [x, y] = (    s * [x, y] +
                              a * [x - 1, y - 1] +
                              b * [x, y - 1] +
                              c * [x + 1, y - 1] +
                              d * [x - 1, y] +
                              e * [x + 1, y] +
                              f * [x - 1, y + 1] +
                              g * [x, y + 1] +
                              h * [x + 1, y + 1] ) / dělitel
```

Obr. 3.1. Ukázka výpočtu rozostření pixelu pomocí Gaussovy matice.

Tento postup aplikujeme postupně na všechny pixely obrázku, čímž dostaneme Gaussovsky rozostřený obrázek uložený v pomocné matici. Je nutno zdůraznit, že příklad je popsán pro šedotónový obrázek. Při práci s barevnými obrázky se musí tento postup aplikovat pro každou barevnou složku zvlášť.

Možná si teď říkáte a co například pixel na pozici [0, 0] vždyť vlevo a nahoru od něj se budeme snažit získat pixely mimo rozsah obrázku. Touto problematikou se budu zabývat později v kapitole „4. Problém okrajových pixelů”.

Nyní si popíšeme parametry ovlivňující doostření. Prvním a nejdůležitějším parametrem je poloměr Gaussovy matice, který je závislý na velikosti σ. Zde pro zvyšující se hodnoty dochází ke zvětšování poloměru, tudíž k větší míře doostření. Jak se hodnota σ projeví v doostření, můžete vidět na obrázcích 3.1 až 3.4.

Obr 3.1. Doostření s hodnotou σ *0,6.*

Obr. 3.2. Doostření s hodnotou σ *2,2.*

Obr. 3.3. Doostření s hodnotou σ *13,4.*

*Obr. 3.4. Doostření s hodnotou **σ** 35,5.*

Druhým podstatným parametrem je **Míra**, který jak již jeho název napovídá, má vliv na míru doostření, ale při zachování stejného poloměru. Jeho hodnota se uvádí v procentech, ale při zpracování se používá jeho hodnota vyjádřena desetinným číslem. Proto při jeho aplikaci nedochází pouze ke zvětšení, ale i k zmenšení míry doostření, jelikož jeho rozsah začíná na 1 procentu, což odpovídá číslu 0,01. V algoritmu pro doostření se použije v místě, kde se před přičtením získaných „detailů“ k původnímu obrázku, tyto hodnoty vynásobí velikostí zadanou pro **Míru**. Vliv parametru na doostření s hodnotou **σ** 1,9 můžete porovnat na obrázcích 3.5 až 3.8.

*Obr. 3.5. Doostření s hodnotou **Míry** 30 %.*

*Obr. 3.6. Doostření s hodnotou **Míry** 100 %.*

*Obr. 3.7. Doostření s hodnotou **Míry** 320 %.*

*Obr 3.8. Doostření s hodnotou **Míry** 866 %.*

Posledním parametrem je **Práh**, který slouží pro redukci šumu vznikajícího doostřováním. Rozsah tohoto parametru je dán rozsahem hodnot pixelů, tedy v našem případě od 0 do 255. Princip používání redukce šumu v programu není nijak zvlášť složitý, pouze se porovnává rozdíl mezi pixelem původního a rozostřeného obrázku, tedy „detaily“, s hodnotou pro **Práh**. Pokud je získaný rozdíl větší nežli zadaný parametr, doostření se neprovede a příslušnému pixelu se ponechá jeho původní hodnota. Ovlivnění **Práhu** při doostření obrázku se **σ** o velikosti 11,8 a 100 procentní **Mírou** můžete vidět na obrázcích 3.9, 3.10 a 3.11.

*Obr. 3.9. Doostření s hodnotou **Práhu** 0 a **σ** o velikosti 11,8.*

*Obr. 3.10. Doostření s hodnotou **Práhu** 141 a **σ** o velikosti 11,8.*

*Obr. 3.11. Doostření s hodnotou **Práhu** 238 a **σ** o velikosti 11,8.*

Při používání standardního nástroje pro doostřování, který je v základní nabídce filtrů aplikace Adobe Photoshop, je umožněno nastavovat naráz výše uvedené parametry pro všechny barevné složky stejně. Tím může při doostřování docházet k podostření nebo přeostření některého kanálu, protože všechny barevné kanály neobsahují stejné „detaily“ obrázku, a nemůže být správně doostřena každá barevná složka zvlášť. Ve vytvořeném pluginu je však toto zohledněno a uživatel má možnost nastavit parametry pro jednotlivé kanály nezávisle na sobě.

3.1. Vyšší efektivita algoritmu

V této kapitole se podíváme na časovou náročnost výše uvedeného algoritmu, a jak snadno ji lze snížit. Abych byl více konkrétní, budeme uvažovat, že máme obrázek o rozměrech 800 x 600 pixelů a podle zadané hodnoty sigma nám vyšla Gaussova matice o rozměrech 100 x 100 pixelů.

Zaměřme se nyní na výpočet rozostřené matice, kde musíme pro každý pixel obrázku provést 100 krát 100 operací násobení mezi pixely obrázku a příslušnými pixely v Gaussově matici. Jelikož tento postup aplikujeme na každý pixel obrázku, pak finální počet operací násobení je 4,8 miliard. A snad nemusím zdůrazňovat, jak by toto číslo narůstalo pro větší a větší obrázky.

Právě zde můžeme využít jedné z výhod Gaussovského rozostření a to, že matice použita pro rozostření je symetrická podle svého jádra. Tuto vlastnost použijeme při výpočtu rozostřené matice, kde nebudeme potřebovat celou Gaussovu matici, ale pouze pole s jejím řádkem obsahujícím jádro matice.

Pak se nám výše popsaný algoritmus doostřování pozmění o to, že se obrázek nejprve rozostří pouze po řádcích pomocí našeho pole a takto „řádkově“ rozostřený obrázek ještě rozostříme stejným polem po sloupcích. Upravený postup výpočtu matice pro rozostřený obrázek je znázorněn na *Obr. 3.1.1*.

Tímto způsobem dostaneme analogicky rozostřený obrázek jako při použití celé Gaussovy matice. Zde jsme však při rozostření jednoho pixelu museli provést pouze 2 krát 100 operací násobení mezi pixelem obrázku a příslušným pixelem v poli. Při aplikaci na celý obrázek tak dostáváme jen 0,096 miliard operací násobení což je 50 krát méně než při použití celé matice.

Gaussova matice:

a	b	c
d	**s**	**e**
f	g	h

d = e = b = g
a = c = f = h

Postačující pole Gaussovy matice:

d	**s**	**e**

dělitel = **d** + **s** + **e**

O – matice obrázku
P – pomocná matice pro rozostření obrázku
R – matice rozostřeného obrázku

Výpočet jednoho pixelu při rozostření po řádcích:

P [x, y] = (**s** * **O**[x, y] +
d * **O**[x - 1, y] +
e * **O**[x + 1, y]) / dělitel

Tento postup aplikujeme na všechny pixely matice obrázku, abychom získaly matici **P**, kterou v dalším kroku použijeme při rozostření po sloupcích.

Výpočet jednoho pixelu při rozostření po sloupcích:

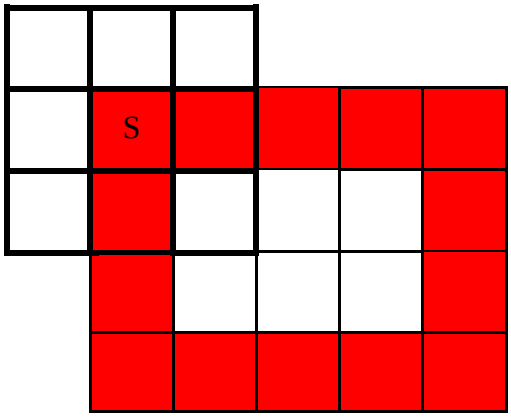
R [x, y] = (**s** * **P**[x, y] +
d * **P**[x, y - 1] +
e * **P**[x, y + 1]) / dělitel

Opět aplikujeme na všechny pixely pomocné matice pro získání výsledné rozostřené matice **R**.

Obr. 3.1.1. Postup výpočtu matice pro rozostřený obrázek s využitím symetričnosti Gaussovy matice.

4. Problém okrajových pixelů

Hlavním problémem při doostřování jsou okraje obrázku. Jak již bylo výše naznačeno při aplikaci popsaného postupu doostřování nastávají situace, kdy se dostáváme mimo rozsah obrázku. Dochází tedy k tomu, že část Gaussovy matice pro okrajové pixely obrázku zasahuje mimo jeho rozsah. Jak máme možnost vidět na obrázku (*Obr. 4.1*), při použití Gaussovy matice typu 3x3 je náš postup pro všechny červeně označené pixely obrázku nepoužitelný.



Obr. 4.1. Problém při ostření okrajových pixelů obrázku.

Pro tento problém však existuje několik postupů jak ho vyřešit.

4.1. Ignorování pixelů

Mezi nejjednodušší řešení patří ignorování okrajových pixelů, kde se dostáváme mimo rozsah obrázku. Rozměry výstupního obrázku budou stejné jako vstupního. Výstupní okrajové pixely se standardně natavují na hodnotu 0, čímž na okrajích výstupního obrázku bude patrný černý okraj. Tento fakt může zavinit podstatný rozdíl v grey level statistikách daného obrázku.

4.2. Kopírování pixelů

Toto řešení spočívá, jak již název napovídá, v kopírování okrajových pixelů ze vstupního obrázku do výstupního. Výsledkem bude okraj výstupního obrázku, kde budou nezpracované pixely. I když se zdá toto řešení, jako postačující není však řešením nejlepším.

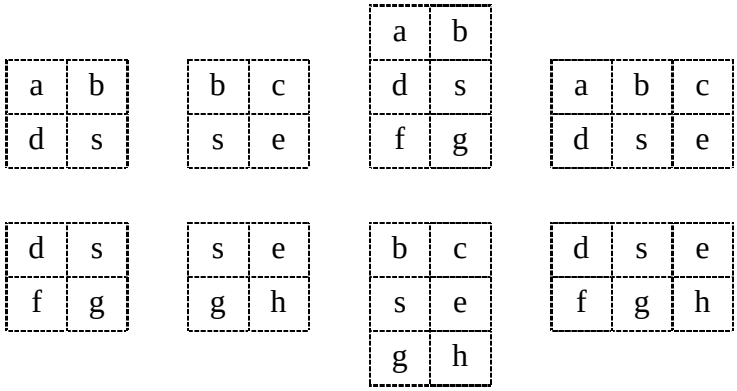
4.3. Oseknutí Gaussovy matice

Další řešení jak se vypořádat s okrajovými pixely je použití modifikovaných Gaussovských matic. Například pokud máme matici typu 3x3 :

a	b	c
d	s	e
f	g	h

Obr. 4.3.1. Matice typu 3x3

Můžeme použít jednu z následujících oříznutých matic použitelných na rohy a strany obrázků:



Obr. 4.3.2. Modifikované matice pro matici typu 3x3

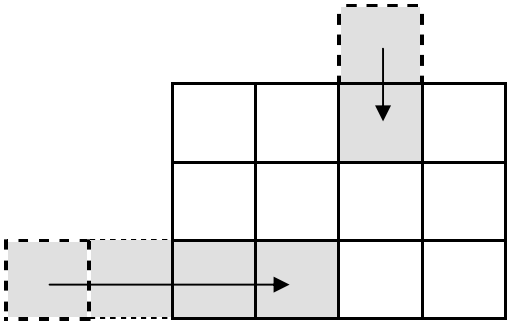
Efekt takto modifikovaných matic na obrázek bude shodný s efektem celé matice. Tato technika je sice považována za komplexní, avšak díky větší časové náročnosti není také nejlepším řešení.

4.4. Oseknutí obrázku

Další jednoduchá cesta jak se vyhnout okrajům, které nemohou být zpracovány, je jejich prosté vyloučení. Výsledkem je výstupní obrázek, který je však menší než původní. To může zapříčinit problémy při kombinování s původním obrázkem.

4.5. Reflektové indexování

Jinými slovy řečeno překlápění podle okraje zpět do obrázku. Tato metoda nedělá nic jiného, než že kontroluje každou vypočítanou souřadnici, zda je platnou souřadnicí obrázku. Pokud není, můžeme ji reflektovat zpět do obrázku.



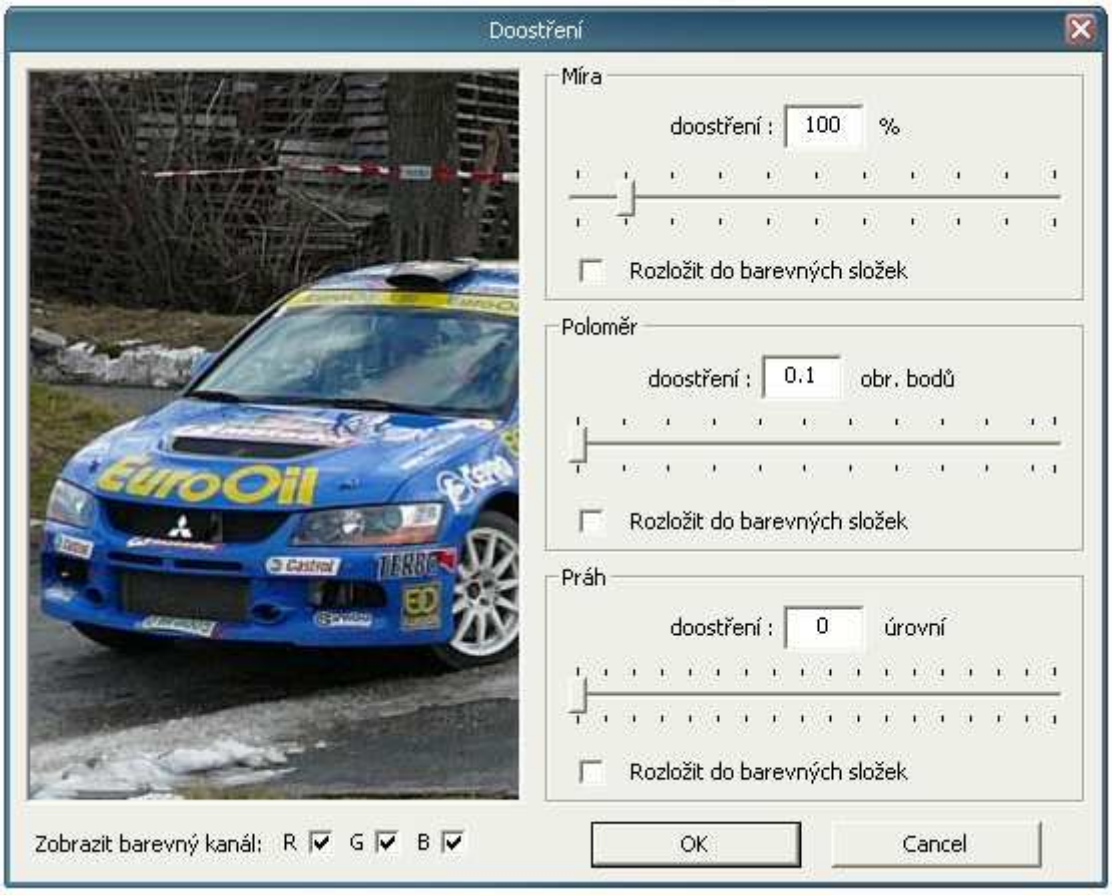
Obr. 4.5.1. Využití reflektového indexování při získávání neplatné souřadnice obrázku.

Tato technika se mi jeví jako nejoptimálnější a proto ji používám v mém programu.

5. Plugin pro Adobe Photoshop

Tato kapitola není zaměřena na podrobný popis a základní nastavení pro vytvoření vlastního pluginu, jak by se dalo očekávat. Zde bych Vás chtěl odkázat na brožuru pana Ing. Jiřího Skály „Tvorba zásuvných modulů pro Adobe Photoshop“, ve které v kapitolách 2 až 8 je popsáno vše podstatné pro vytvoření pluginu do aplikace Adobe Photoshop pomocí vývojového prostředí Microsoft Visuál C++ .

Zmínit bych se přesto chtěl o tom, že mnou vytvořený plugin patří do kategorie zásuvných modulů typu Filtr a proto jej při použití naleznete v menu Filtr\Ostření. Po spuštění se zobrazí dialog s uživatelským rozhraním na obrázku 5.1, který slouží k nastavení parametrů pro doostření. Trojice panelů Míra, Poloměr a Práh jsou po řadě určeny k nastavení míry doostření, velikosti sigmy ze které je určen poloměr a hodnoty pro redukci šumu. Každý z uvedených parametrů je možno rozložit pro nastavení jednotlivých barevných složek zvlášť. Dialog samozřejmě oplývá náhledovým okénkem zobrazujícím na části obrázku, která se do něho vejde, jak by vypadala aplikace filtru na obrázek a je umožněno měnit, jak zobrazovanou část v něm, tak i zobrazované barevné kanály.



Obr. 5.1. Dialogové okno pluginu.

Dále je nutno zdůraznit, kde naleznete důležité adresáře s hlavičkovými soubory pro tvorbu pluginu. Pokud zachováte základní nastavení při instalaci Adobe Photoshop SDK 6.0, umístí se do adresáře:

c:\Program Files\Adobe\Adobe Photoshop 6.0 SDK\

ze kterého pro nás budou důležité soubory v adresářích:

.. \Common\Includes
.. \Common\Resources
.. \PhotoshopAPI\General
.. \PhotoshopAPI\Photoshop
.. \PhotoshopAPI\Pica_sp

6. Uživatelské rozhraní pluginu

Při vytváření uživatelského rozhraní můžeme volit mezi implementací pomocí Adobe Dialog Manageru a Win32Api.

Adobe Dialog Manager je rozhraní pro správu dialogových oken v aplikacích Adobe. Dialogy vytvořené pomocí Adobe Dialog Manageru jsou nezávislé na platformě a mají konzistentní vzhled, tzv. look and feel.

Já jsem se však rozhodl použít Win32Api s pomocí šablony, která se ukládá do souboru s koncovkou „.rc“ a proto se dále budu věnovat popisu implementace dialogu tímto způsobem.

V následujících podkapitolách jsou uvedeny nejpoužívanější funkce pro některé ovládací prvky a způsob jak zpracovávat jejich nejdůležitější zprávy na nich generované různými událostmi. Jsou zde popsány jen ty ovládací prvky, které jsem využil při tvorbě pluginu.

Na začátek je nutno připomenout, že do našeho hlavního souboru „hlavni.cpp“ musíme připojit základní hlavičkový soubor <windows.h>.

6.1. Dialog

Pro vytvoření modálního dialogu uživatelského rozhraní slouží funkce DialogBox.

```
INT_PTR DialogBox(
    HINSTANCE hDllInstance,         // handl modulu (instance)
    LPCTSTR lpTemplate,             // název zdroje dialogu
    HWND hWndParent,                // handl rodičovského okna
    DLGPROC lpDialogFunc            // procedura okna dialogu
);
```

6.1.1. Handl modulu pro dialog

V našem případě se jedná o handl DLL knihovny, který získáme při inicializaci Win32 DLL knihovny, jenž musí být uvedena na začátku našeho hlavního souboru „hlavni.cpp“.

```
HANDLE      hDllInstance = NULL;

BOOL APIENTRY DllMain(HANDLE hModule, DWORD ul_reason_for_call,
    LPVOID lpReserved)
{
    if (ul_reason_for_call == DLL_PROCESS_ATTACH)
        hDllInstance = hModule;
    else
        hDllInstance = NULL;
    return true;
}
```

6.1.2. Zdroj dialogu

Do zdroje dialogu přiřadíme náš dialog, který jsme si vytvořili v rc šabloně, pomocí funkce MAKEINTRESOURCE.

```
MAKEINTRESOURCE(„název dialogu")
```

6.1.3. Rodičovské okno dialogu

Handlem (*hWndParent*) rodičovského okna se rozumí handl aplikace Adobe Photoshop, který získáme z proměnné typu *PlatformData* jenž naplníme následujícím příkazem.

```
FilterRecord *plugParam;
PlatformData *platform;

platform = (PlatformData*)((FilterRecordPtr) plugParam)->platformData;
hWndParent  = (HWND)platform->hwnd;
```

6.1.4. Obslužná funkce dialogu

Pokud dojde v dialogu k nějaké události například k zmáčknutí tlačítka, klávesy nebo kurzor myši změní polohu atd., je zavolána procedura definovaná v parametru *lpDialogFunc*, například funkce DialogProc.

```
INT_PTR CALLBACK DialogProc(
    HWND hwndDlg,               // ukazatel na dialog
    UINT uMsg,                  // typ zprávy
    WPARAM wParam,              // 1. parametr zprávy
    LPARAM lParam               // 2. parametr zprávy
);
```

Tato funkce je obvykle implementována jako jeden velký příkaz switch, s definicí obsluhy pro jednotlivé typy zpráv. Zprávy, které nás nezajímají, nemusíme obsluhovat.

První zpráva, která je generována dialogem je WM_INITDIALOG. Ta je vyvolána ještě před zobrazením dialogu. Touto cestou lze například předat obslužné funkci ukazatel na instanci třídy, jenž obsahuje proměnné, které mají mít stejnou hodnotu jako ovládací prvky dialogu.

Zpráva WM_PAINT je generována pokud potřebuje dialog obnovit svůj obsah. Všechny standardní ovládací prvky jsou obnovovány automaticky, ale pokud je v dialogu použit prvek nestandardního rázu např. náhled, je třeba zajistit jeho obnovování při příchodu této zprávy.

Zprávy generované standardními ovládacími prvky jsou obvykle typu WM_COMMAND. Údaje o ovládacím prvku, který zprávu vygeneroval, jsou zakódovány v parametru *wParam*. Příkazem LOWORD(*wParam*) získáme identifikaci ovládacího prvku a příkazem HIWORD(*wParam*) typ události, jako je kliknutí, posunutí a podobně. Je pochopitelné, že za tímto příkazem musí následovat další příkaz switch. Ten je vložen za položku WM_COMMAND prvního příkazu switch jako druhá úroveň třídění. Finální struktura obslužné funkce, by měla vypadat nějak takto:

```
INT_PTR CALLBACK DialogProc(HWND hwndDlg, UINT uMsg, WPARAM wParam,
                                    LPARAM lParam)
{
    switch (uMsg)
    {
        case  WM_INITDIALOG:
            //inicializace dialogu
            break;
        case  WM_COMMAND:
            Switch ( LOWORD (wParam))
            {
                case ID_1:
                    //obsluha ovládacího prvku s identifikací ID_1
                    if (HIWORD (wParam) == BN_CLICKED)
                    {
                        //obsluha kliknutí na ovládací prvek s identifikací ID_1
                    }
                break;
            }
    }
}
```

6.2. Tlačítko (Button)

U tlačítka se obvykle používá jen nastavení aktivity. Neaktivní tlačítko je standardně zašeděné a na nic nereaguje. Zda má být tlačítko aktivní či ne nastavíme pomocí funkce EnableWindow.

```
BOOL EnableWindow(
    HWND hWnd,          // ukazatel na ovládací prvek
    BOOL bEnable        // příznak, zda má být tlačítko aktivní
);
```

Před použitím této funkce je třeba zjistit ukazatel na ovládací prvek. K tomu slouží funkce GetDlgItem.

```
HWND GetDlgItem(
    HWND hWnd,              // ukazatel na okno dialogu
                            //= parametr hwndDlg obslužné funkce
    int nIdDlgItem          // identifikace ovládacího prvku
);
```

Pro jednoduchost je vhodné spojit oba příkazy do jedné konstrukce.

```
EnableWindow( GetDlgItem (hwndDlg, TLACITKO), true);
```

Tlačítko s identifikací TLACITKO bude aktivní. Tuto operaci lze aplikovat na libovolný ovládací prvek.

6.2.1. Obsluha zpráv tlačítka

Stisknutí tlačítka generuje zprávu WM_COMMAND, kde identifikace tlačítka a událost jsou uloženy v parametru *wParam*. Obsluha tlačítka s identifikací TLACITKO vypadá takto:

```
:
case  WM_COMMAND:
    switch (LOWORD (wParam))
    {
        case TLACITKO:
            if (HIWORD (wParam) == BN_CLICKED)
            {
                //tlačítko bylo stisknuto
            }
            break;
    }
```

6.3. Zaškrtávací políčko (Check Box)

U tohoto ovládacího prvku lze nastavit stav políčka na zaškrtnuté nebo nezaškrtnuté. Ten se nastavuje funkcí SetCheckBoxState.

```
BOOL SetCheckBoxState (
    HWND hWnd,              // ukazatel na ovládací prvek
    int nIDButton,          //identifikace ovládacího prvku
    UINT uCheck             //stav políčka
);
```

Proměnná *uCheck* může nabývat hodnot true a false.

6.3.1. Obsluha zpráv zaškrtávacího políčka

Změna stavu tlačítka generuje zprávu WM_COMMAND, kde identifikace zaškrtávacího políčka je uložena v parametru *wParam*. Jelikož aktuální stav políčka není součástí zprávy, musí být zjištěn zvláštním příkazem IsDlgButtonChecked.

```
UINT IsDlgButtonChecked(
    HWND hWnd,              // ukazatel na okno dialogu
                            // = parametr hwndDlg obslužné funkce
    int nIDButton           // identifikátor zaškrtávacího políčka
);
```

Obsluha zaškrtávacího políčka s identifikací POLICKO vypadá takto:

```
case  WM_COMMAND:
    Switch ( LOWORD (wParam))
    {
        case POLICKO:
            if( IsDlgButtonChecked(hwndDlg, POLICKO))
                // tlačítko je zaškrtnuto
            else
                // tlačítko není zaškrtnuto
            break;
    }
```

6.4. Textové pole (Edit Control)

U textového pole použijeme především operace nastavení a získání aktuálního textu v poli. Pro nastavení textu slouží funkce SetDlgItemText.

```
BOOL SetDlgItemText (
    HWND hWnd,              // handle okna dialogu
    int nIDDlgItem,         // identifikátor prvku
    LPTSTR lpString,        // text
);
```

V opačném případě kdy budeme chtít získat aktuální text z pole, použijeme funkci GetDlgItemText.

```
UINT GetDlgItemText(
    HWND hWnd,              // handle okna dialogu
    int nIDDlgItem,         // identifikátor prvku
    LPTSTR lpString,        // buffer pro text
    int nMaxCount           // maximální počet zkopírovaných znaků
);
```

6.4.1. Obsluha zpráv textového pole

Změna obsahu textového pole generuje zprávu WM_COMMAND, kde identifikace vybraného pole a událost jsou uloženy v parametru *wParam*. Jelikož součástí zprávy není aktuální text v poli, musí být zjištěn zvlášť. Pak obsluha pro textové pole s názvem TEXT_POLE bude vypadat následovně:

```
case WM_COMMAND:
    Switch ( LOWORD (wParam))
    {
        case TEXT_POLE:
            if ( HIWORD (wParam) == EN_CHANGE)
                GetDlgItemText(hWnd, TEXT_POLE, text, pocetZnaku);
                // v poli text je obsah textového pole
            break;
```

6.5. Trackbar (Slider Control)

Předtím než si popíšeme nastavení trackbaru, musíme si objasnit nastavení pro náš program, aby byl trackbar funkční. Nejprve musíme přidat pro linkování do projektu knihovnu <comctl32.lib> a vložit hlavičkový soubor <commctrl.h>.

```
// Běžné hlavičkové soubory
#include <commctrl.h>

// linkeru přidáme knihovnu přímo ve zdrojovém kódu
#pragma comment (lib, "comctl32.lib")
```

Dále musíme na začátku programu, přesněji řečeno před vytvořením dialogového okna, použít funkci InitCommonControlsEx.

```
//inicializace pro trackbar
INITCOMMONCONTROLSEX icc;
icc.dwSize = sizeof(INITCOMMONCONTROLSEX);
icc.dwICC = ICC_WIN95_CLASSES  | ICC_USEREX_CLASSES;
if ( !InitCommonControlsEx(&icc))
    return -1;
```

U trackbaru je třeba nastavit několik parametrů:

6.5.1. Nastavení rozsahu

Pro nastavení rozsahu od nuly do sta trackbaru s identifikací TB použijeme následující konstrukci.

```
SendDlgItemMessage(hwndDlg, TB, TBM_SETRANGE, true, MAKELONG(0, 100));
```

6.5.2. Nastavení značek

Značku na pozici deset pro trackbar s identifikací POSUVNIK, vložíme takto:

```
SendDlgItemMessage(hwndDlg, POSUVNIK, TBM_SETTIC, 0, (WPARAM)10);
```

6.5.3. Nastavení pozice

Nastavení aktuální pozice na nulu pro trackbar s identifikací POSUVNIK.

```
SendDlgItemMessage(hwndDlg, POSUVNIK, TBM_SETPOS, true, 0);
```

6.5.4. Obsluha zpráv trackbaru

Změna nastavení trackbaru generuje zprávu WM_HSCROLL v případě horizontální trackbaru nebo WM_VSCROLL pro vertikální trackbar, kde identifikace vybraného trackbaru je uložena v parametru *lParam*. Vygenerovanou událost (TB_THUMBPOSITION, TB_THUMBTRACK, TB_BOTTOM, TB_LINEDOWN, TB_LINEUP, TB_PAGEDOWN, TB_PAGEUP nebo TB_TOP) a aktuální hodnotu nalezneme v parametru *wParam*. Velký počet událostí, které vznikají na trackbaru jsou způsobeny různými způsoby ovládání, například myší, kurzorovými klávesami nebo tlačítky PageUp, PageDown, Home a End. Proto je obsluha trackbaru trochu odlišná od obsluh předchozích ovládacích prvků.

```
:
case WM_HSCROLL:
    switch ( LOWORD (wParam))
    {
        case TB_THUMBPOSITION:
        case TB_BOTTOM:
        case TB_TOP:
        case TB_PAGEDOWN
        case TB_PAGEUP
        case TB_LINEDOWN
        case TB_LINEUP
        case TB_THUMBTRACK:       //došlo ke změně polohy trackbaru
            int poz = (int)SendDlgItemMessage(hwndDlg, POZ, TBM_GETPOS, 0, 0);
            //aktuální hodnota trackbaru s identifikací POZ se uloží do proměnné poz
    }
break;
```

6.5.5. Nelineární pohyb trackbaru

Nelineárního pohybu využijeme v případech, kdy potřebujeme zjemnit pohyb jezdce pro nastavení některých hodnot v daném rozsahu pro určitý parametr, neboli zvětšit oblast na trackbaru pro nastavení kýžených hodnot. Tato vlastnost není podporována samotným ovládacím prvkem, ale musíme si ji zajistit sami. Jak lze nelineárního pohybu docílit popíši na konkrétním případu v pluginu. Zde je této vlastnosti využito pro trackbar sloužící k nastavení hodnoty sigmy, kde je vhodné přesnější nastavení nižších hodnot. Aby se pohyb trackbaru jevil, jako nelineární s jemnějším natavením hodnot v prvním polovině rozsahu pro sigmu, musíme po získání polohy jezdce tuto hodnotu nejprve přepočítat pomocí nějaké funkce a výsledek teprve budeme považovat za hodnotu sigma, kterou budeme zpracovat a zobrazovat jako aktuální polohu. V našem případě je vhodné použít exponenciální funkci, jejíž průběh odpovídá výše zmíněnému kritériu, čili z počátku narůstá pomaleji a pro zvyšující se hodnoty roste rychleji. Funkci použijeme dle vzorce 6.5.5.1.

$$\sigma = a^{x}$$

Vzorec 6.5.5.1. Vzorec pro exponenciální funkci.

Kde *x* představuje hodnotu polohy jezdce a za základ funkce *a* je vhodné dosadit hodnotu 1,9, kterou jsem po porovnání několika exponenciál o různých základech shledal jako optimální.

Pokud však dojde ke změně hodnoty sigma jiným prvkem nežli trackbarem a tato změna se má projevit i na poloze jezdce, musíme opět pomocí upravené stejné exponenciální funkce provést přepočet pro získání skutečné polohy jezdce. Získanou hodnotu teprve nastavíme, jako jeho polohu. Upravený tvar funkce můžete vidět ve vzorci 6.5.5.2.

$$x = \frac{\log \sigma}{\log a}$$

Vzorec 6.5.5.2. Modifikovaný vzorec pro exponenciální funkci.

Nakonec bych chtěl upozornit, že rozsahy trackbaru a hodnot pro sigmu nejsou stejné. Proto při počátečním nastavování rozsahu trackbaru nejsou za minimum a maximum dosazovány minimální a maximální hodnoty sigmy, ale musíme je přepočítat pomocí vzorce 6.5.5.2.

6.6. Obrázek (Picture Control)

Tomuto prvku lze velice jednoduše v „rc“ šabloně přiřadit bitmapový obrázek, který je taktéž uložen ve zdrojích, a to v případě, kdy se během programu tento obrázek nemění a je použit například pro vylepšení vzhledu programu. Pokud však má tento prvek plnit funkci náhledu zpracovávaného obrázku, který během programu samozřejmě mění svůj vzhled, musíme nejprve inicializovat proměnnou typu HBITMAP. V té budeme uchovávat obrázek náhledu, který bude obsahovat část zpracovávaného obrázku.

```
HBITMAP CreateCompatibleBitmap(
    HDC hdc,                    // kontext zařízení
    int nWidth,                 // šířka náhledu
    int nHeight                 // výška náhledu
);
```

Pro vytvoření kontextu zařízení bitmapy použijeme funkci GetDC.

```
HDC GetDC (
    HWND hWnd                    // ukazatel na okno dialogu
);
```

Dále je potřeba do vytvořeného ukazatele bitmapy vložit pixely obrázku, které chceme zobrazit, pomocí funkce SetDIBits.

```
int SetDIBits(
    HDC hdc,                    // kontext zařízení
    HBITMAP hBitmap,            // ukazatel na bitmapu
    UINT uStartScan,            // první řádek obrázku uloženého
                                // v poli lpvBits od kterého se začne načítat
    UINT cScanLines,            // počet řádků pro načtení
    CONST VOID *lpvBits,        // pole s pixely bitmapy
    CONST BITMAPINFO *lpbmi,    // info o bitmapě
    UINT fuColorUse             // použitý barevný prostor
);
```

Za parametry *uStartScan* a *cScanLines* bychom měli přiřadit hodnoty nula a výška náhledu, aby došlo k načtení celého obrázku.

Důležitým parametrem je **lpvBits*, kterému předáme pole s pixely obrázku pro náhled. V tomto poli musejí být obrazová data uloženy prokládaně, tedy nejprve jsou uloženy všechny barevné kanály prvního pixelu, pak všechny kanály druhého pixelu atd. Zde bych chtěl upozornit na opačné uspořádání barevných kanálů. Ty jsou uloženy v pořadí B (modrý kanál), G (zelený kanál), R (červený kanál) a alfa kanál, který nemusíme vyplňovat.

Dalším důležitým parametrem je **lpbmi* což je struktura obsahující základní informace o naší bitmapě. Pro její naplnění doporučuji vytvořit si vlastní metodu, ve které je nutno naplnit položky uvedené v následující konstrukci.

```
PBITMAPINFO CreateBitmapInfo(){

    PBITMAPINFO bInf = NULL;
    WORD cClrBits = 32;                                  // bitová hloubka monitoru

    bInf = (PBITMAPINFO) LocalAlloc ( LPTR, sizeof(BITMAPINFOHEADER) +
          + sizeof(RGBQUAD) * (1<< cClrBits));

    bInf->bmiHeader.biSize = sizeof(BITMAPINFOHEADER);
    bInf->bmiHeader.biWidth = sirkaNahledu;              // šířka obrázku pro náhled
    bInf->bmiHeader.biHeight = vyskaNahledu;             // výška obrázku pro náhled
    bInf->bmiHeader.biPlanes = 1;                        // musí být nastaveno na 1
    bInf->bmiHeader.biBitCount = 32;                     // počet bitů na pixel
    bInf->bmiHeader.biCompression = BI_RGB;              // komprese obrázku
    bInf->bmiHeader.biClrImportant = 0;

    return bInf;
}

PBITMAPINFO bInf = CreateBitmapInfo();
```

Parametr *bInf->bmiHeader.biClrImportant* specifikuje počet barevných indexů požadovaných pro zobrazení bitmapy. Jestliže tuto hodnotu nastavíme na nulu, jsou požadovány všechny barvy.

Ještě nesmíme opomenout při plnění bitmapy nastavit parametr *fuColorUse* na hodnotu DIB_RGB_COLORS a bitmapa pro náhled je vytvořena.

Nyní přiřadíme vytvořený ukazatel bitmapy do ovládacího prvku. K tomu nám poslouží již známá funkce SendDlgItemMessage a pro prvek s identifikací *NAHLED* použijeme konstrukci:

```
SendDlgItemMessage(hwndDlg, NAHLED, STM_SETIMAGE, IMAGE_BITMAP,
                                    (LPARAM) hBitmap);
```

Nakonec nesmíme zapomenout na uvolnění vytvořeného kontextu zařízení.

```
ReleaseDC(hwndDlg, hdc);
```

6.6.1. Obrázky v barevném prostoru CMYK

Při zpracování obrázků v barevném prostoru CMYK se dostáváme k jednomu z nedostatku ve Win32Api, které s tímto barevným prostorem nedokáže pracovat, ale

dokáže zobrazit pouze obrázky v RGB prostoru. S tímto problémem se můžeme vypořádat pomocí funkce colorServices, kterou nám nabízí plugin host. Ta poskytuje služby související s barvami a nejčastěji je právě používána pro konverzi barevných kanálů.

```
OSErr colorServices(ColorServicesInfo *info)
```

Použití této funkce pro konverzi z CMYK do RGB režimu předvedu v následujícím příkladu.

```
ColorServicesInfo csInfo;
csInfo.infoSize = sizeof(csInfo);                    // z důvodu číslování verzí

csInfo.reserved = NULL;                              //
csInfo.reservedSourceSpaceInfo = NULL;               // nutno nastavit na NULL
csInfo.reservedResultSpaceInfo = NULL;               //

csInfo.selector = plugIncolorServicesConvertColor;   // použitá služba konverze
csInfo.sourceSpace = plugIncolorServicesCMYKSpace;   // počáteční barevný prostor
csInfo.resultSpace = plugIncolorServicesRGBSpace;    // požadovaný prostor

uint8 pixelCMYK[4];
uint8 pixelRGB[3];

for (int a = 0; a < 4; a++)
    info.colorComponents[a] = pixelCMYK[a];          // zkopírování barvy pro konverzi

if (gFilterRecord->colorServices(&info) == noErr)    // kontrola úspěšného převodu
    for (int b = 0; b < 3; b++)
        pixelRGB[b] = (uint8)info.colorComponents[b];    // !! čtení složek již v RGB
                                                         // musíme přetypovat na uint8 !!
```

Nevýhodou funkce je vysoká časová náročnost, způsobená pomalou konverzí barev Photoshopem.

6.7. Myš

Myš není většinou potřeba nastavovat, snad kromě kurzoru. Jeho změny lze využít například u náhledu. Kurzor můžeme změnit příkazem SetCursor.

```
HCURSOR SetCursor(
    HCURSOR hCursor          //ukazatel na kurzor
);
```

Před použitím tohoto příkazu musíme nejprve vytvořit nový kurzor pomocí funkce LoadCursor.

```
HCURSOR LoadCursor(
    HINSTANCE hInstance,    // ukazatel na aplikaci, v které je kurzor uložen
    LPCTSTR lpCursorName    // jméno kurzoru nebo jeho zdroje
);
```

Pokud si vystačíte s předdefinovanými kurzory ve Windows, můžeme použít následující konstrukci pro nastavení kurzoru.

```
SetCursor (LoadCursor(NULL,MAKEINTRESOURCE(„Identifikace")));
```

Typ kurzoru	Identifikace
Standardní šipka	IDC_ARROW
Textový kurzor	IDC_IBEAM
Kříž	IDC_CROSS
Čtyřsměrná šipka	IDC_SIZEALL
Ruka	IDC_HAND
Šipka s otazníkem	IDC_HELP
Hodiny	IDC_WAIT

6.7.1. Obsluha zpráv myši

U myší se jedná o zprávy zaměřené na stav tlačítek a na pohyb myši. Stisknutí levého tlačítka se vygeneruje zpráva WM_LBUTTONDOWN a po jeho uvolnění WM_LBUTTONUP. Pozice, na které k události došlo, je uložena v parametru *lParam*. Ostatní tlačítka myši se chovají obdobně.

Při změně pozice kurzoru, je vygenerována zpráva WM_MOUSEMOVE a to tehdy, pokud je kurzor nad prvkem, u kterého je myš sledována. Jak lze sledovat myš, je popsáno v další kapitole na příkladu pohybu náhledu.

6.8. Pohyb náhledu

Pohybem náhledu se míní pohyb zpracovávaného obrázku v něm, neboli zobrazení jiné části obrázku do náhledu. Pří řešení tohoto pohybu lze využít sledování myši, kdy po stisku tlačítka nad prvkem náhledu si uložíme aktuální polohu kurzoru a po pohybu myši se stisknutým tlačítkem odečteme získanou předchozí polohu od nové aktuální polohy kurzoru. O tento rozdíl, pak posuneme souřadnice části obrázku, která má být načtena do bitmapy náhledu.

Jelikož ovládací prvek pro obrázky negeneruje zprávy, které by nás informovaly o stavu myši nad tímto prvkem, musíme si zajistit vlastní sledování. K

tomuto účelu využijeme takzvaného „subclassingu“, což znamená „napíchnutí se“ na proceduru okna libovolného prvku na dialogu.

Nejprve v handleru zprávy WM_INITDIALOG zavoláme funkci SetWindowLongPtr pro nastavení procedury okna na vlastní funkci. Je důležité uložit si původní hodnotu vrácenou voláním této funkce do proměnné typu WNDPROC. Pro prvek s názvem *NAHLED* použijeme následující konstrukci.

```
WNDPROC staryProc;

switch (uMsg)
{
    case  WM_INITDIALOG:
        staryProc = (WNDPROC)SetWindowLongPtr(GetDlgItem(hwndDlg, NAHLED),
                    GWLP_WNDPROC, (LONG_PTR)WindowProcPreview);
        :
        break;
}
```

Funkce WindowProcPreview může být společná pro více prvků stejného typu. Podívejme se, jak vypadá v našem případě:

```
LRESULT CALLBACK WindowProcPreview(
    HWND hWnd,
    UINT uMsg,
    WPARAM wParam,
    LPARAM lParam
){

    int x, y;

    switch ( uMsg )
    {
        case WM_GETDLGCODE:
            return DLGC_WANTARROWS;


        case WM_SETCURSOR:                  // nastavení kurzoru při najetí nad náhled
            SetCursor(LoadCursor(NULL,IDC_HAND));    // do podoby ruky
            return true;


        case WM_LBUTTONDOWN:                        // stisknuto levé tlačítko myši
            SetCapture(hWnd);                       // pro zasílání zpráv i mimo prvek
            SetCursor(LoadCursor(NULL,IDC_SIZEALL)); // změna kurzoru na čtyřsměr. šipku
            staraPozMysi = MAKEPOINTS(lParam);      // zjistím počáteční polohu kurzoru
            return true;
```

```
        case WM_LBUTTONUP:                          // uvolněno levé tlačítko myši
            ReleaseCapture();                       // uvolnění zasílaní zpráv mimo prvek
            return true;


        case WM_MOUSEMOVE:// došlo k pohybu myši
            if(wParam == MK_LBUTTON)        // posun pouze při stisknutém levém tlačítku
            {
                SetCursor(LoadCursor(NULL,IDC_SIZEALL));        // změna kurzoru

                novaPozMysi = MAKEPOINTS(lParam);   // zjistím novou polohu kurzoru

                x = staraPozMysi.x - novaPozMysi.x;     // výpočet posunu po ose X
                y = novaPozMysi.y - staraPozMysi.y;     // výpočet posunu po ose Y
                staraPozMysi = novaPozMysi;             // uložím novou polohu kurzoru

                kontrolaPosun(x, y);                    // procedura kontrolující souřadnice
                nactiNahled();                          // procedura načte náhled
            }
            return true;
    }

    return CallWindowProc(staryProc, hWnd, uMsg, wParam, lParam);
}
```

Handler zprávy WM_GETDLGCODE slouží k informování systému, že chceme do našeho prvku posílat některé další zprávy, především zprávy myši. Hodnotou DLGC_WANTARROWS si o ně jednoduše řekneme.

Při posouvání náhledu, bychom neměli zapomenout kontrolovat změnu souřadnic, tak aby nedošlo k posunu náhledu mimo rozsah obrázku.

7. Uživatelská příručka

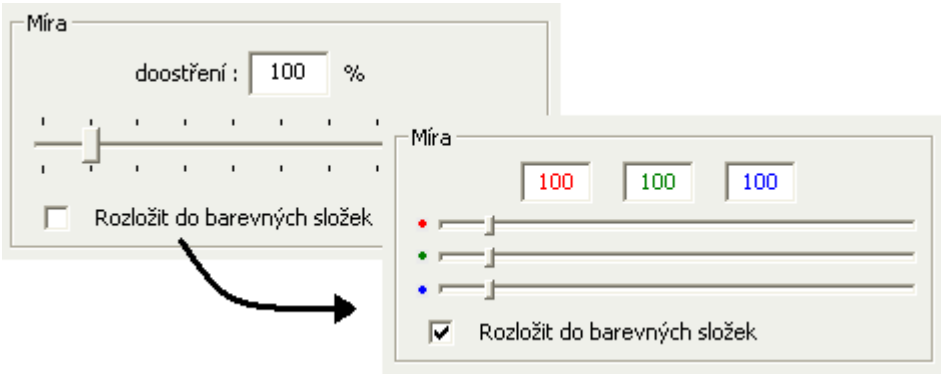
Nejprve musíte nakopírovat soubor Doostrovani.8bf do adresáře Adobe Photoshopu se jménem Plug-Ins\Filters nebo pokud používáte českou verzi programu, pak jej zkopírujte do adresáře Zásuvné moduly\Filtry. Plugin po novém spuštění Photoshopu naleznete v menu Filtr\Ostření\Doostřit… Po spuštění pluginu se zobrazí dialogové okno (Obr. 5.1), které můžete kdykoli ukončit pomocí tlačítka Cancel, křížku v levém horním rohu nebo klávesy Esc.

V levé polovině dialogu naleznete okno s náhledem obrázku. Po najetí kurzoru do prostoru vymezeného pro náhled se kurzor myši změní do podoby ruky, čímž Vás informuje o možnosti pohybovat obrázkem v náhledu. Pohybu docílíte přidržením levého

tlačítka myši a jejím pohybem. Při pohybu náhledu mění kurzor svou podobu do tvaru čtyřsměrné šipky.

Pod oknem náhledu se nachází skupina zaškrtávacích políček, jejichž počet je závislý na barevném prostoru obrázku. Tyto políčka slouží pro nastavení zobrazování barevných kanálu v náhledu. Při zaškrtnutém políčku u příslušné složky je složka zobrazována.

V pravé polovině dialogu se nachází trojice panelů pojmenovaných Míra, Poloměr a Práh sloužící k nastavení vlastností pro doostření obrázku. V panelu s názvem Míra můžete pomocí posuvníku nebo změnou hodnoty v textovém poli nastavit procentuální velikost doostření. Po upravení této hodnoty dojde ke zpracování obrázku, který je pak nově upravený zobrazen v náhledu. V dolní části panelu je zaškrtávací políčko sloužící k rozdělení nastavení hodnoty míry pro jednotlivé barevné kanály (Obr. 7.2). Toto zaškrtávací políčko s obdobnou funkcí je umístěno v každém ze tří panelů. V dalším panelu Poloměr můžete stejným způsobem nastavit hodnotu pro sigma z Gaussova rozostření použitého pro doostření obrázku. Posledním panelem Práh lze opět stejným postupem natavit hodnotu pro korekci šumu vzniklého při doostřování.



Obr. 7.2. Rozložení míry pro jednotlivé barevné kanály.

V dolní části pravé poloviny se ještě nachází dvojice tlačítek Ok a Cancel pomocí nichž můžeme potvrdit nebo zrušit provedení doostření obrázku.

8. Závěr

Nejprve jsem se při zpracovávání této práce musel seznámit s algoritmem doostřování a s problémy, které při něm nastávají. Dle mého názoru se nejedná o nijak složitou záležitost a jeho jednoduchá implementace mě v celku překvapila. Samozřejmě, že má i svá úskalí, ale ty lze velice efektivně řešit.

Největší překážkou bylo vytvořit uživatelské rozhraní v prostředí Win32Api, které pro mne bylo doposud naprosto neznámé. Po dlouhém hledání a pročítání různých

tutoriálů na toto téma se mi podařilo porozumět problematice tvorby dialogů v prostředí Win32Api. Jelikož se jedná o celkem rozsáhlé, zajímavé a v oblasti pluginu o méně prokoumané téma je mu věnována velká část práce. Tím jsem samozřejmě nechtěl naznačit, že tvorba pluginu pomocí SDK 6.0 by byla nezajímavá či jednoduchá. I v tomto směru se v mém případě jednalo o něco nového, avšak s touto problematikou mi velice pomohla brožuru pana Ing. Jiřího Skály „Tvorba zásuvných modulů pro Adobe Photoshop“.

Výsledkem programové části mé bakalářské práce je plugin pro Adobe Photoshop, který je schopen doostřovat obrázky s využitím Gaussovského rozostření. Obrázky mohou být v barevném režimu RGB a CMYK při osmy bitech na kanál. Při doostřování může obsluha využít odlišného doostření jednotlivých barevných kanálů podle různých kritérií pro každý kanál.

V případě dalšího upravování pluginu by bylo vhodné vyřešit převod pixelů z barevného prostoru CMYK pro náhledové okénko, jiným efektivnějším způsobem nežli pomocí funkce *colorServices*. Zde vidím slabinu ve velké časové naročnosti.

Použitá literatura

[1] SKÁLA, Jiří. *Tvorba zásuvných modulů pro Adobe Photoshop*. Studentský sborník KIV FAV ZČU. Plzeň: KIV, 2006.

[2] URL <*http:// www.builder.cz*> [cit. 2007-04-14]

[3] URL <*http:// www.codeproject.com*> [cit. 2007-04-14]

[4] ŽÁRA, J., BENEŠ, B., SOCHOR, J., FELKEL, P. *Moderní počítačová grafika*. 2. vydání. Brno: Computer Press, c2004. ISBN 80-251-0454-0.