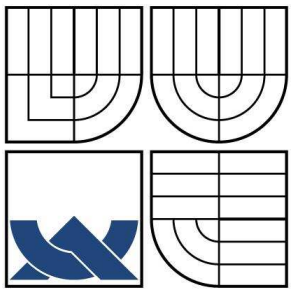
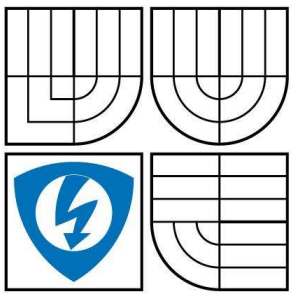


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

ÚSTAV AUTOMATIZACE A MĚŘICÍ TECHNIKY

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

DEPARTMENT OF CONTROL AND INSTRUMENTATION

OCR CÍLENĚ ZNEHODNOCENÝCH TEXTŮ

OCR OF IMAGE BASED WEB FORM PROTECTION

DIPLOMOVÁ PRÁCE

MASTER'S THESIS

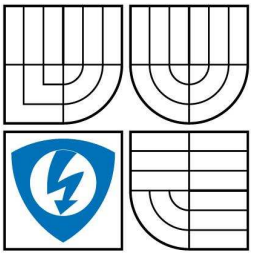
AUTOR PRÁCE — Bc. TIBOR PELUCH

AUTHOR

VEDOUCÍ PRÁCE — Ing. PETR HONZÍK, Ph.D.

SUPERVISOR

BRNO 2009



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky a komunikačních technologií

Ústav automatizace a měřicí techniky

Diplomová práce

magisterský navazující studijní obor
Kybernetika, automatizace a měření

Student:	Bc. Tibor Peluch	***ID:***	83369
Ročník:	2	***Akademický rok:***	2008/2009

NÁZEV TÉMATU:

OCR cíleně znehodnocených textů

POKYNY PRO VYPRACOVÁNÍ:

Navrhněte a vytvořte programové rozhraní vhodné pro řešení OCR (Optical Character Recognition) cíleně znehodnocených textů používaných typicky pro ochranu elektronických formulářů na webových stránkách. Program bude mít jednoduché grafické rozhraní umožňující intuitivní ovládání a bude umožňovat jednoduché rozšíření o nové algoritmy ve všech fázích zpracování obrazu a identifikace textu. Dále popište a naprogramujte základní algoritmy na všech úrovních zpracování obrazu a rozpoznání textu, aby bylo možné ověřit funkčnost programu.

DOPORUČENÁ LITERATURA:

Dle vlastního literárního průzkumu a doporučení vedoucího práce.

Termín zadání:	9.2.2009	***Termín odevzdání:***	25.5.2009

Vedoucí práce: Ing. Petr Honzík, Ph.D.

prof. Ing. Pavel Jura, CSc.
Předseda oborové rady

UPOZORNĚNÍ:

ABSTRAKT

První část práce je věnována programování aplikace pro operační systém Windows. V krátkosti jsou shrnuty hlavní rysy aplikačního systému Microsoft Foundation Class. V druhé části je realizována aplikace s grafickým uživatelským rozhraním, která umožňuje pomocí tvorby schématu zpracovávat data. Schéma je složeno z bloků, které mohou mít uživatelské rozhraní, pomocí něhož lze upravovat jejich parametry. Třetí část pojednává o implementaci bloků do dynamických knihoven a je nastíněna možnost využití zpracování dat tohoto programu jako externího modulu a možnost realtime zpracování dat, jako je například obraz a zvuk. Ověření funkčnosti aplikace je v poslední části. Aplikace je konkrétně testována na rozpoznání znehodnoceného textu pro ochranu elektronických formulářů serveru www.centrum.cz. Jsou navrženy bloky, které realizují načítání obrázku přímo z internetu, preprocessing, segmentaci, extrakci příznaků, vyhodnocení neuronovou sítí a bloky, jenž umožňují načítat a ukládat zpracovaná data na disk.

KLÍČOVÁ SLOVA

OCR, Uživatelsné rozhraní, Microsoft Foundation Class, Zpracování signálu, Winapi, Dynamická knihovna, Component Object Model

ABSTRACT

The first part of the thesis deals with programming of application in operating system Windows. Main features of application system Microsoft Foundation Class are resumed in brief here. In following part there is implemented an application with graphic user interface that makes, using schema, work with data, possible. Schema consists of blocks, they have user's interface which is able to modify parameters. The third part deals with implementation of blocks into dynamic linked libraries and there is outlined a possibility to use data of this programme as an external module and a possibility of realtime data processing e.g. picture and sound. The verification of a good functionality of this application is in the last part. The application is really tested in diagnosing of devaluated texts for protecting web forms www.centrum.cz. There were designed blocks making picture read possible just from internet, preprocessing, segmentation, feature extraction, evaluationg in neural network and blocks that make possible to read and save processed data into the disc.

KEYWORDS

OCR, User Interface, Microsoft Foundation Class Library, Signal Processing, WinApi, Dynamic link library, Component Object Model

PELUCH T. *OCR cíleně znehodnocených textů*. Brno: Vysoké učení technické v Brně. Fakulta elektrotechniky a komunikačních technologií, 2009. 74 s. Vedoucí diplomové práce Ing. Petr Honzík, Ph.D.

PROHLÁŠENÍ

Prohlašuji, že svou diplomovou práci na téma „OCR cíleně znehodnocených textů“ jsem vypracoval samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené diplomové práce dále prohlašuji, že v souvislosti s vytvořením této diplomové práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení § 152 trestního zákona č. 140/1961 Sb.

V Brně dne ....

(podpis autora)

Děkuji vedoucímu diplomové práce Ing. Petru Honzíkovi, Ph.D. za velmi užitečnou metodickou pomoc při řešení a zpracování práce.

V Brně dne …….. .....…….

podpis autora

Obsah

SEZNAM OBRÁZKŮ

Seznam tabulek

1. ÚVOD

Cíleně znehodnocený text se na internetu používá k zamezení používání formulářů robotům. Tito roboti by mohli například posílat automaticky SMS zprávy, nebo zkoušet odhalovat hesla uživatelů. Jeho použití spočívá ve vytvoření obrázku, na kterém je zobrazen text, který člověk přečíst umí, ale stroj nikoliv. Jednoduché varianty zabezpečení spočívají v pouhém převodu textu na obrázek, složitější obsahují znehodnocený text rotací a deformací písmen, popřípadě dodání dalších objektů do obrazu. Novodobé ochrany formulářů zakládají na plnění úkolů uživatelem (vybrat první, třetí a páté písmeno z textu v obrázku, nebo opsat písmena, u kterých je namalována kočka).

OCR (Optical Character Recognition) s nástupem počítačů začíná být standardní pojem a v praxi se hojně využívá. Jedná se o rozpoznání textu z obrázku. Původně se OCR začalo uplatňovat v profesích, kde bylo potřeba pomocí počítače urychlit běh procesu (například automatické dělení korespondence podle PSČ). Poté se s nástupem rychlejších počítačů tato metoda začala používat například k převodu naskenovaného textu do dokumentu.

Ačkoliv OCR cíleně znehodnoceného textu webových formulářů není ve světě žádnou novinkou, tato metoda není zdaleka dokonalá. Na každý typ znehodnocení se musí aplikovat konkrétní postup a těžko se dosahuje stoprocentní úspěšnosti.

Cílem této práce je vytvoření programu, který bude nabízet zpracování signálu od načtení obrázku z internetu, předzpracování, segmentaci, vytvoření vektoru příznaků, až po vyhodnocení. Výhoda tohoto řešení bude spočívat v ceně (je to zdarma), ve výkonu (jednotlivé bloky budou psány v jazyce C++) a v přehlednosti (uživatel bude tvořit schémata).

2. GUI

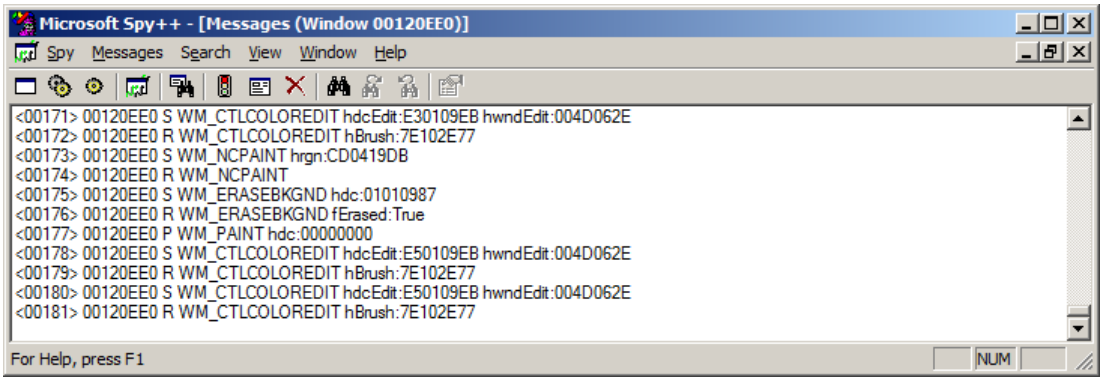
2.1 VOLBA KNIHOVNY PRO UŽIVATELSKÉ ROZHRANÍ

Jak již bylo zmíněno, tato práce bude psána v jazyce C++. Existuje nemálo knihoven, které nabízejí jednoduché tvoření okenních aplikací. V případě systému Windows, který je dnes nejrozšířenější mezi uživateli a pro který z tohoto důvodu bude výsledná aplikace napsána, bude zvoleno využití knihovny MFC (Microsoft Foundation Class). Kromě MFC lze využít například také GTK+, Qt. Začněme ale systém uživatelského rozhraní zkoumat od začátku.

Přestože je v dnešní době znalost WinApi (Windows Application Interface) zdánlivě zbytečná, je dobré znát alespoň základy, aby bylo zřejmé, jak aplikace, které jsou postavené na aplikačním systému, pracují. S použitím aplikačního systému, který zajišťuje chování aplikace, sice nutnost znalosti WinApi odpadá, ale v případě, že je třeba implementovat funkci, kterou aplikační systém nenabízí, jiný způsob, než naprogramovat to na této úrovni, není.

2.2 WINAPI

Jedná se o způsob programování, které funguje na principu doručování zpráv. Většinou jde o zprávy, které jsou generovány systémem jako reakce na vstup uživatele (pohyb myší, stisk klávesy), ale touto metodou lze realizovat i například časovače, předávání informací jednotlivých vláken, zjišťování, zda se aplikace nezacyklila apod. Ke sledování těchto zpráv lze použít například program Spyxx.exe, který je součástí vývojového prostředí Microsoft Visual C++.



Obrázek 2.1 Screenshot aplikace Spy++

2.2.1 Vstupní bod aplikace

Vstupní bod aplikace je kód, který se při spuštění aplikace provede. Ukažme si prototyp funkce konzolové aplikace:

```
int main(int argcp, char* argv[]);
```

Do této funkce vstupují dva parametry. První je počet parametrů a druhý je pole ukazatelů na tyto jednotlivé parametry zakončené hodnotou NULL. Funkce vrací hodnotu typu integer. Systém Windows při programování konzolové aplikace zatajuje některé věci, jako je například vytvoření okna a přesměrování standardních vstupů a výstupů do tohoto okna. „Černé okno“, které se na obrazovce objeví je následkem instrukcí, které se provedou ještě před vstupem do této funkce.

Abychom psali aplikaci, která se o vytvoření oken stará sama, musíme použít jiný vstupní bod:

```
int WINAPI WinMain(HINSTANCE hInstance,
                   HINSTANCE hPrevInstance,
                   LPSTR lpCmdLine,
                   int nShowCmd);
```

V tomto prototypu se vyskytují nestandardní datové typy, které jsou definovány v hlavičkovém souboru „windows.h“ (tento soubor je potřeba vložit do zdrojového kódu pro vytváření aplikací využívajících WinApi). Do funkce vstupují 4 parametry, kde význam je následující:

HINSTANCE hInstance - Handle instance. Konkrétně zde se jedná o ukazatel, který ukazuje na počáteční adresu programu v paměti. Tento parametr ale není třeba. Lze získat později pomocí funkce GetModuleHandle(NULL)

HINSTANCE hPrevInstance - Kdysi byl tento parametr využíván, protože více instancí jedné aplikace sdílelo paměť. Pomocí tohoto parametru šlo zjistit, zda již není otevřená jiná instance procesu. Dnes je tento parametr zachován jen kvůli kompatibilitě a podle dokumentace je hodnota tohoto parametru rovna NULL.

LPSTR lpCmdLine - Jedná se o pole 8-bitových znaků, ve kterém je uložen parametr, který byl uveden za názvem souboru při jeho spuštění. Oproti klasické konzolové aplikaci není dělen na jednotlivé parametry, ale celý řetězec je uložen vcelku. To znamená, že musí být zajištěno případné rozdělení textu.

int nCmdShow – v tomto parametru je uveden způsob otevření aplikace. V konzolové aplikaci se o vytvoření okna nestaráme, proto není potřeba vědět, zda má být okno vytvořeno například minimalizované. Ve WinApi aplikaci vytvoření okna realizuje programátor a tak by podle tohoto parametru měl zajišťovat způsob zobrazení okna.

Klíčové slovo WINAPI uvedené před názvem funkce je přetypované __stdcall a jedná se o způsob vkládání parametrů do zásobníku při volání funkce.

2.2.2 Registrace třídy okna

Pojem „Třída okna“ je v tomto významu poněkud matoucí. Nejedná se o třídu v jazyce C++, ale o jakousi „šablonu okna“. Programování GUI v systému Windows je založeno na doručování zpráv jednotlivým oknům, které zajišťuje operační systém. Aby bylo možné oknu doručovat zprávy, je nutné systému dát vědět, kam tyto zprávy zasílat. A aby bylo možné vytvořit více oken, které se budou chovat stejně, registruje se právě „třída okna“.

Registrace třídy okna spočívá v naplnění struktury WNDCLASS. Významy jednotlivých prvků této struktury jsou následující:

UINT style - určení vlastností okna, jako je například odesílání žádosti o překreslení při změně velikosti okna, nebo generování zprávy o dvojkliku myší v případě dvojkliku.

WNDPROC lpfnWndProc - specifikuje callback funkci, které budou doručovány zprávy

int cbClsExtra - Počet bajtů, které uživatel potřebuje alokovat spolu se třídou (pro uložení vlastních informací)

int cbWndExtra - počet bajtů, které uživatel potřebuje alokovat spolu s každým oknem

HINSTANCE hInstance - handle instance - vyplní se hodnotou parametru hInstance vstupního bodu

HICON hIcon - handle ikony, která se zobrazuje například při stisku ALT+TAB

HICON hIconSm - handle malé ikony, která bude přidružena k aplikaci (ikona zobrazující-se vlevo nahoře na okně a na taskbaru)

HCURSOR hCursor - handle kurzoru, který bude použit v případě přemístění kurzoru myši nad okno

HBRUSH hBrush - handle štětce, který bude použit při mazání obsahu okna

LPCTSTR lpszMenuName - Jméno menu, které bude zobrazeno v okně

LPCTSTR lpszClassName - jméno třídy, které registrujeme - podle tohoto názvu se později budou vytvářet okna

2.2.3 Callback funkce okna

Každé okno je v systému identifikováno svým HWND (Handle of WiNDow). Tento handle se využívá při doručování zpráv oknům. Jeho hodnota se získá při vytváření okna.

Při registraci třídy okna byla uvedena funkce, do které budou doručovány všechny zprávy patřící příslušnému oknu. Její typ byl uveden jako WNDPROC. Z dokumentace, nebo z příslušného hlavičkového souboru lze vyčíst, že její prototyp bude následující:

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM lParam);
```

Význam parametrů:

HWND hWnd - identifikuje okno, kterému je zpráva doručena

UINT Msg - obsahuje zprávu, která je oknu doručena

WPARAM wParam a LPARAM lParam - Dva parametry, které upřesňují význam zprávy

Pro zasílání zprávy je nutné znát HWND okna (cíl, kam má být zpráva doručena) a samotnou zprávu. Zprávu určují tedy tři parametry, kde v 32bitovém systému má každý parametr délku právě 32bitů (dále bude předpokládán 32bitový systém). První parametr (Msg) určuje zprávu. Množství zpráv, které systém využívá, je obrovské a proto se využívá maker, které jsou nadefinovány v již zmíněném hlavičkovém souboru „windows.h“. Výčet některých základních zpráv je v tabulce Tabulka 2.1:

Tabulka 2.1 Příklady zpráv OS Windows

WM_CREATE	Okno se vytváří
WM_DESTROY	Okno bylo zničeno
WM_CLOSE	Okno se zavírá
WM_MOUSEMOVE	Pohyb myší
WM_LBUTTONDOWN	Stisknuto levé tlačítko myši
WM_KEYDOWN	Stisknuta klávesa
WM_COMMAND	Uživatel klikl na menu, nebo stiskl klávesovou zkratku
WM_NOTIFY	Na některém z potomků došlo k události

S každou zprávou jsou doručeny také parametry zprávy. V některých případech množství informace, které je předáno (2x32bitů) postačuje. Například ve zprávě WM_MOUSEMOVE jednotlivé bity parametru wParam udávají, jestli je stisknuto některé z tlačítek myši, popřípadě klávesy CTRL, SHIFT. Makrem LOWORD(lParam) lze zjistit x-ovou souřadnici a HIWORD(lParam) y-ovou souřadnici myši.

V případech, kdy těchto 64 bitů nepostačuje, může být zaslán v jednom z parametrů ukazatel na místo v paměti, kde jsou uloženy parametry zprávy.

Zprávy lze oknu zasílat dvěma způsoby. První způsob spočívá ve využití funkce:

```
LRESULT SendMessage(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM lParam);
```

Je vidět, že návratová hodnota a parametry se shodují s callback funkcí okna. Při zavolání této funkce se tedy systém postará o vyhledání okna podle jeho handlu a zavolání příslušné callback funkce se zadanými parametry.

Aby mohl být vysvětlen druhý způsob zasílání zpráv oknům, je nutné vědět, že spolu s oknem se vytváří fronta zpráv, do které se ukládají zprávy, které není třeba zaslat ihned. Při zavolání funkce SendMessage se volání callback funkce provede ihned a pokračování ve vykonávání funkce, ve které jsme zprávu zasílali, nastane až po vykonání callback funkce příslušného okna. Samozřejmostí je, že tato funkce vrátí hodnotu, která byla vrácena v callback funkci okna.

Na konec fronty zpráv lze přidat zprávu funkcí:

```
BOOL PostMessage(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM lParam);
```

Z prototypu je patrné, že funkce vrací datový typ BOOL, který indikuje úspěšnost přidání zprávy do fronty. Pro úplnost je možnost zasílat zprávy také vláknům, k čemuž slouží funkce PostThreadMessage se stejnými parametry, ale cíl není handle okna, ale „id“ vlákna. Takto zaslaná zpráva se odchytává přímo ve smyčce zpráv (viz 2.2.5).

Při používání těchto funkcí v kombinaci se zasíláním ukazatele v parametru si je třeba uvědomit, že při volání funkce PostMessage a PostThreadMessage dojde k uložení zprávy na konec fronty a program pokračuje dál. Nelze tedy tímto způsobem zaslat ukazatel na lokální proměnnou, protože v době zpracování zprávy již nemusí lokální proměnná existovat.

2.2.4 Vytvoření okna

K vytvoření okna slouží funkce:

```
HWND CreateWindow(LPCTSTR lpClassName, LPCTSTR lpWindowName,DWORD dwStyle,
                  int x, int y, int nWidth, int nHeight, HWND hWndParent,
                  HMENU hMenu, HINSTANCE hInstance, LPVOID lpParam);
```

LPCTSTR lpClassName - název registrované třídy - může být třída okna registrována voláním funkce RegisterClass, nebo některá z tříd oken nabízená systémem jako je tlačítko, seznam atd. (viz Tabulka 2.2)

LPCTSTR lpWindowName - Název okna – v případě klasického okna název, jenž je zobrazen v titulkovém pruhu vedle ikony okna

DWORD dwStyle - Styl okna - kombinace maker (použitím logického OR) určující vzhled okna a další jeho parametry (viz Tabulka 2.3)

int x - x-ová souřadnice okna

int y - y-ová souřadnice okna

int nWidth - šířka okna

int nHeight - výška okna

HWND hWndParent - handle rodiče okna

HMENU hMenu - handle menu, které bude k oknu přiřazeno

HINSTANCE hInstance - handle instance programu, ve které bylo okno registrováno

LPVOID lpParam - volitelný parametr, který bude oknu doručen při vytváření

Okno není v systému Windows chápáno pouze jako objekt, který má popisek, tlačítko „zavřít“ a nějaký obsah. Tento pojem je obecně jakýkoliv objekt (tlačítko, popisek, seznam). Protože se tyto objekty mohou nacházet uvnitř jiného okna, mají okna hierarchickou strukturu. K procházení této struktury je k dispozici funkce:

```
HWND GetWindow(HWND hWnd, UINT uCmd)
```

Hierarchické struktury je využíváno například při ničení oken (musí být zničeni také všichni potomci okna).

Systém Windows obsahuje některé předem definované základní třídy oken, které programátorovi umožňují snadno vytvořit základní ovládací prvky aplikace.

Název	Popis
Button	Tlačítko
ComboBox	Výběr položky z možností
Edit	Editační pole
ListBox	Seznam
MDIClient	Klientské okno pro MDI aplikace
ScrollBar	Posuvník
Static	Statický text

Tabulka 2.2 Některé systémové třídy oken

Styl	Význam
WS_OVERLAPPEDWINDOW	Standardní okno, které má menu, tlačítka min,max,zavřít a lze měnit jeho velikost
WS_CHILD	Okno je potomkem okna s handlem hWndParent
WS_VISIBLE	Okno bude po vytvoření ihned vidět
WS_MINIMIZE	Okno je minimalizováno

Tabulka 2.3 Některé styly oken

2.2.5 Smyčka zpráv

Základem programu pro operační systém Windows je smyčka zpráv. Jedná se o část programu, ve které dochází k odebrání zprávy z fronty zpráv a jejímu odeslání příslušnému oknu. V případě, že ve frontě není žádná zpráva (a zprávy jsou odebírány funkcí GetMessage), program přejde do stavu, kdy čeká na zprávu a tak neplýtvá časem procesoru.

Typický kód smyčky zpráv:

```
MSG msg;
while( GetMessage(&msg, NULL, 0, 0) )
{
  TranslateMessage{&msg};
  DispatchMessage{&msg};
}
```

Zpráva je zde reprezentována strukturou MSG, která obsahuje kromě již zmíněných čtyř parametrů (handle okna, zpráva a 2 parametry) také časové razítko okamžiku, kdy byla zpráva přidána do fronty (odebrání fronty nemusí nastat ihned po zařazení zprávy do fronty) a polohu myši v okamžiku jejího přidání.

Podmínkou ve smyčce je návratová hodnota funkce GetMessage. Tato funkce vrací hodnotu TRUE vždy kromě případu zachycení zprávy WM_QUIT. Důležitý parametr této funkce je adresa struktury MSG, do které budou překopírovány informace o zprávě, která se vybírá z fronty. Další parametry slouží pouze k přesnější specifikaci zpráv, které mají být vybrány (uvedené hodnoty způsobí výběr všech zpráv z fronty).

Funkce TranslateMessage slouží ke zpracování zpráv generovaných stisknutím klávesy (WM_KEYDOWN, WM_KEYUP). Jednoduše řečeno jde o transformování klávesy podle použité klávesnice (anglická klávesnice po stisku klávesy „1“ má způsobit napsání hodnoty „1“, zatímco česká hodnoty „+“). Poslední funkce DispatchMessage způsobí zavolání callback funkce okna, kterému zpráva náleží.

2.2.6 Zdroje

Programátorské prostředí Microsoft Visual Studio nabízí možnost jednoduchého vložení zdrojů (resources) do programu, jako jsou ikony, bitmapy, kurzory, dialogy, menu, akcelerátory (klávesové zkratky) atd. Výhoda spočívá v uložení přímo do spustitelného souboru a také v možnosti editace těchto zdrojů přímo v prostředí Visual C++. Tímto se podstatně zkrátí doba, kterou programátor tráví nad vytvářením programu.

2.3 MICROSOFT FOUNDATION CLASS

MFC je aplikační systém, který má definovány třídy pro práci nejen s okny. Na jednu stranu v kódu začne být mnoho věcí pro programátora neviditelných, ale využití tohoto systému přináší značné zjednodušení práce při programování.

2.3.1 CObject

Třídy, které ulehčují již zmíněnou práci nejen s okny, využívají dědičnosti. Všechny třídy MFC dědí z objektu CObject. Tato třída nabízí základní funkce, jako je serializace, získávání informace o třídě za běhu, diagnostika objektu a zajišťuje kompatibilitu se třídami kolekcí.

Diagnostika objektu se používá při ladění. Implementuje metody AssertValid, která kontroluje stav objektu a Dump, který umožňuje vypsat informace o objektu. Tato metoda se využívá při zjišťování úniků paměti (programátor neuvolnil paměť).

Další jmenované funkce spadají do tří úrovní. Nejnižší úroveň se deklaruje do třídy makrem DECLARE_DYNAMIC. Jako parametr se uvede název aktuální třídy. V implementaci se pak musí použít makro IMPLEMENT_DYNAMIC, kde se jako parametry uvedou jméno aktuální třídy a rodičovské třídy. Díky těmto makrům poté třída obsahuje metody určené k identifikaci objektu za běhu, což je užitečné při používání polymorfních objektů.

```
void function(CObject* pSomeObject)
{
  if ( pSomeObject->IsKindOf(RUNTIME_CLASS(CMyClass)) )
    { /* pSomeObject je typu CMyClass */ }
  else if ( pSomeObject->IsKindOf(RUNTIME_CLASS(CObject)) )
    { /* pSomeObject je typu CObject */ }
}
```

Další úroveň umožňuje dynamické vytváření. V deklaraci se použije makro DECLARE_DYNCREATE a v implementaci makro IMPLEMENT_DYNCREATE. Tato makra definují metodu CreateObejct, která umožňuje vytvářet objekt dynamicky. Příklad:

```
CRuntimeClass *ret=RUNTIME_CLASS(CMyClass);
CObject *newobj=ret->CreateObject();
```

Této metody se využívá při použití poslední úrovně funkcí třídy, definovanou makrem DECLARE_SERIALIZE a implementovanou IMPLEMENT_SERIALIZE.

Serializace znamená uložení dat do série za sebe. Využívá se například v případě nutnosti uložení dat na disk. V případě uložení nějakého řetězce se nejedná o velký problém, ale při ukládání nějaké hierarchické struktury může správné uložení programátorovi zabrat nemálo času. Jednoduchost využití serializace je naznačena na následujícím příkladu:

```
class COtherObject : public CObject
{
private:
  int m_nVar1;
public:
  COtherObject() { m_nVar1 = 1; }
  virtual void Serialize( CArchive& ar );
protected:
  DECLARE_SERIAL( COtherObject )
};

class CMyObject : public CObject
{
  int m_nVar1, m_nVar2;
  COtherObject* m_pOtherObject;
  CObject* m_pUnknownObject;
public:
  CMyObject() {
    m_pOtherObject = new COtherObject();
    m_nVar1 = 1; m_nVar2 = 2; }
  virtual void Serialize( CArchive& ar );
protected:
  DECLARE_SERIAL( CMyObject )
};

IMPLEMENT_SERIAL(CMyObject,CObject,1)

void CMyObject::Serialize( CArchive& ar ) {
     CObject::Serialize(ar);
     m_pOtherObject->Serialize(ar); // serializace objektu známého typu
  if ( ar.IsStoring() )
  {
    ar << m_nVar1 << m_nVar2;
    ar << m_pUnknownObject; // serializace objektu neznámého typu
  }
  else
  {
    ar >> m_nVar1 >> m_nVar2;
    ar >> m_pUnknownObject; // serializace objektu neznámého typu
  }
}
```

V uvedeném příkladu jsou dvě třídy. Je vidět, že se přepisuje virtuální metoda Serialize, ve které se zavolá nejdříve metoda Serialize implementovaná v nadřazené třídě. Je zde serializace objektu známého typu (víme, že je typu COtherObject). Poté následuje podmínka, která zjišťuje, jestli se data ukládají, nebo načítají (kvůli použití správných operátorů při práci s proudy). Touto cestou se realizuje serializace

základních datových typů a také serializace polymorfních objektů. Při rekonstrukci třídy typu CObject pomocí operátoru >> se využívá výše zmíněné dynamické vytváření. Což znamená, že při ukládání pomocí operátoru << byl do souboru spolu s třídou zapsán datový typ objektu, podle které je třída při načítání rekonstruována.

2.3.2 CCmdTarget

Třída CCmdTarget je odvozená z anglického „Command Target“, což znamená cíl příkazu. Pro šíření zpráv, které bylo vysvětleno v kapitole týkající se WinApi se využívá právě tato třída, která umožňuje vytváření tzv. „map zpráv“. Mapu zpráv si lze představit jako tabulku, ve které se definují obslužné metody třídy pro určité příchozí zprávy. Tento proces lze jednoduše zapsat pomocí maker, které jsou uvedeny v následujícím příkladu:

```
BEGIN_MESSAGE_MAP(CMyWnd, CWnd)
  ON_BN_CLICKED (ID_MY_BUTTON, OnMyButton)
  ON_BN_CLICKED (ID_MY_BUTTON2, OnMyButton2)
END_MESSAGE_MAP()
```

Teoreticky by bylo možné tento proces mapování zpráv realizovat přes virtuální metody tříd, kdy by každá zprávy měla určenou jednu metodu ve třídě, která by při příchodu zprávy byla vykonána. Ale protože množství zpráv je obrovské, byla by tato metoda příliš náročná na paměť.

2.3.3 CWnd

Třída CWnd umožňuje manipulaci s okny. Dědí z CCmdTarget, což umožňuje definovat již zmíněné mapy zpráv. Jak bylo řečeno v kapitole o WinApi, oknem se rozumí jakýkoliv objekt, který byl vytvořen funkcí CreateWindow. Její metody umožňují pouze základní operace s oknem (nebereme-li v úvahu použití metody SendMessage, kterou lze poslat libovolnou zprávu). Pro ovládání konkrétnějšího okna se využívá tříd, které jsou z této třídy odvozeny (CFrameWnd, CDialog, CView, CButton, CEdit...)

2.3.4 Dokument-pohled architektura

Aby bylo programování aplikace přehledné, oddělují se v programu data dokumentu od pohledů (způsob prezentace dat v dokumentu). MFC proto definuje třídy CView a CDocument. Aplikace poté může disponovat více pohledy na data (například textový editor může mít editor textu a HEXa editor).

2.3.5 Vlákna

Další důležitá věc při programování aplikace je rozdělení aplikace do vláken. Není to sice povinné, ale v případě, že vytvářená aplikace provádí výpočty, které trvají déle, je vhodné aplikaci rozdělit na více vláken, kde jedno vlákno obsluhuje GUI a druhé provádí výpočet. Vlákna se spravují pomocí třídy CWinThread.

2.3.6 Ostatní třídy

MFC obsahuje mnoho dalších tříd, které jsou určeny také například pro práci s vlákny, procesy, síťovými prostředky, databázemi, nabídkami, kontexty grafických zařízení apod. Všechny jsou vysvětleny v dokumentaci (www.msdn.microsoft.com). V následujícím textu tyto třídy budou použity a budou vysvětleny pouze okrajově.

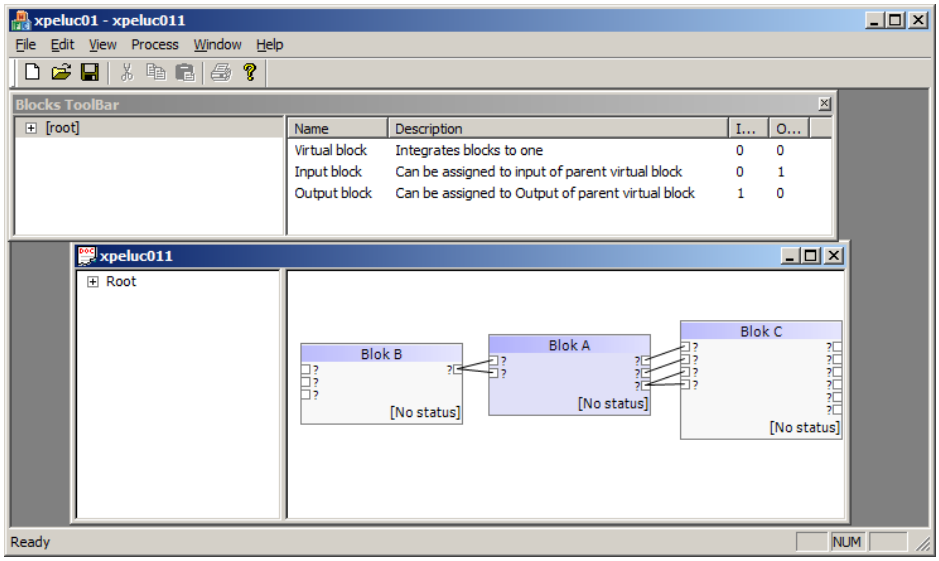
3. APLIKACE

3.1 VZHLED APLIKACE

V prostředí Microsoft Visual C++ jsou obsaženy průvodci pro vytvoření základní kostry aplikace MFC. Rozdíl v programování MDI (Multiple Document Interface) a SDI (Single Document Interface) aplikace je minimální, takže není důvod, proč nevytvořit MDI aplikaci.

Obrázek 3.1 naznačuje vzhled aplikace. V aplikaci se nachází okno dokumentu (aktivní MDI okno) a panel nástrojů s dostupnými bloky (Blocks Toolbar). Panel nástrojů bude mít stromovou strukturu, aby byl seznam bloků přehlednější. Vložení bloku do schématu bude probíhat pomocí „Drag and Drop“ (přetažení myší z panelu do dokumentu). Aby se předešlo problémům s pracovní plochou, souřadnice každého bloku budou zadány relativně - tj. procentuálně, kde se v okně nachází (například uprostřed okna: x=0,5, y=0,5). Protože velikost bloků bude konstantní a velikost plochy schématu se bude měnit podle velikosti okna, bude vhodné implementovat možnost integrace více bloků do jednoho. Poté bude možné rozdělit schéma například do oblastí načtení obrázku, preprocessing, segmentace atd. K navigaci uvnitř schématu poté poslouží levá část dokumentu, kde je znázorněna stromová struktura dokumentu.

Protože aplikace bude disponovat grafickým rozhraním pro tvorbu schémat, které budou znázorňovat tok signálů, je nutné nejdříve navrhnout systém, podle kterého se bude signál ve schématu šířit. Slovo signál bude v následujícím textu označovat informace, které si bloky předávají.

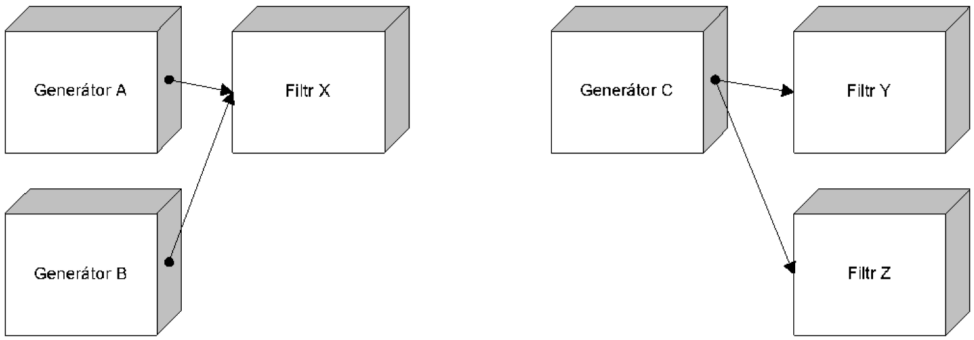


Obrázek 3.1 Vzhled aplikace

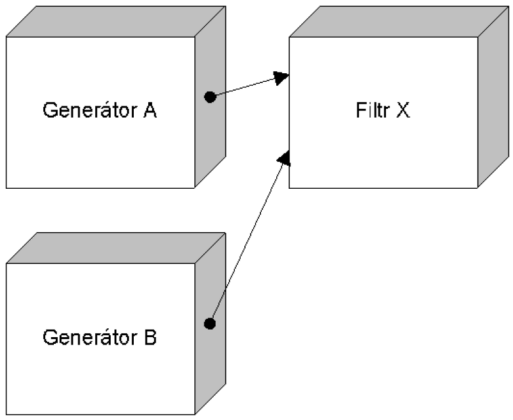
3.2 ŠÍŘENÍ SIGNÁLU

Aby bylo možné kdykoliv přistupovat k datům, která jsou na výstupu bloku, každý blok bude obsahovat paměť pro každý svůj výstupní port, ve které budou uložena výstupní data. K problematice šíření signálu lze přistupovat dvěma způsoby.

Inspirujeme-li se reálným světem, signál se šíří od generátoru, přes jednotlivé filtry až na konec schématu. Při realizaci této cesty by tedy každý blok obsahoval pro každý výstupní port ukazatel na cílový blok a port, do kterého je zapojen. V případě, že by bloky byly propojeny tímto způsobem, nastal by problém, jak zamezit zapojení dvou výstupů do jednoho vstupu (blok A bude zapojen do bloku C a blok B bude zároveň zapojen do bloku C). Další nevýhoda tohoto způsobu spočívá v problematice zapojování jednoho výstupu do více vstupů (muselo by být realizováno množinou ukazatelů pro každý výstupní port bloku). Tyto problémy naznačuje Obrázek 3.2.



Obrázek 3.2 Problémy při šíření signálu od generátoru 1

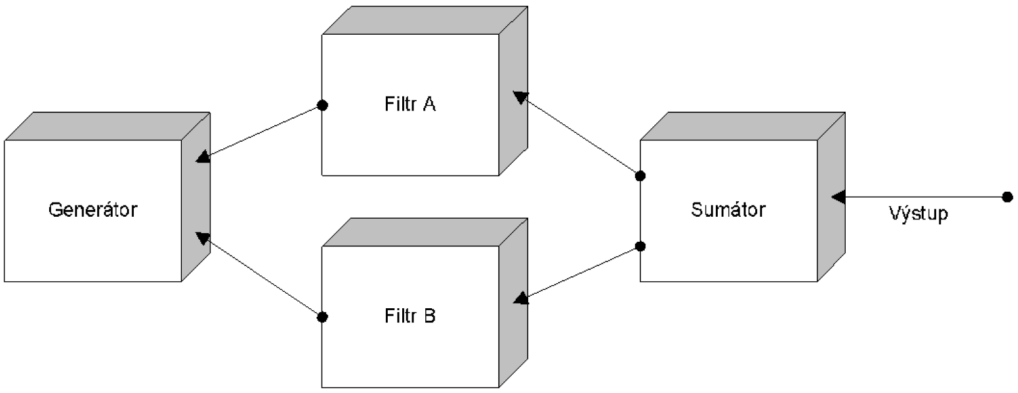


Obrázek 3.3 Problémy při šíření signálu od generátoru 2

Další implementační problém spočívá v neznalosti signálů na vstupech. Na obrázku 3.3 jsou dva generátory signálu (A,B) a jeden filtr (C) se dvěma vstupy. V okamžiku šíření signálu například z generátoru A se signál dopraví do filtru C. Ten ale nemůže signál zpracovat, protože nemá signál k dispozici signál na druhém vstupu od generátoru B. Poslední problém spočívá v nutnosti nějakým způsobem při výpočtu předávat data, která jsou na příslušném výstupním portu bloku (následující blok neví, který blok je zapojen na jeho vstup).

Druhý způsob spočívá v rekurzivním výpočtu signálů na blocích (Obrázek 3.4). Ukazatele na blok a port, do kterého je příslušný port bloku zapojen nejsou na výstupních, ale na vstupních portech. Tento způsob realizace zapojení eliminuje problém zapojení dvou vstupů do jednoho výstupu. Také lze zapojit jeden výstup do více vstupů (každý vstup může mít ukazatel na výstup, na který ukazuje již jiný blok). Rekurzivní způsob je založen na výpočtu všech bloků, které jsou zapojeny na

vstupu. Protože je znám ukazatel na blok a port, který je zapojen na vstup, lze získat data, která jsou zapojena na vstupních portech. Po výpočtu všech bloků, které jsou zapojené na vstup (každý blok takto vypočítá bloky zapojené na jeho vstup) je k dispozici na blocích, zapojených na vstup, signál. Tímto tedy odpadá i problém z prvního způsobu, kdy nebyly známy data na nějakém vstupním portu.



Obrázek 3.4 Rekurzivní způsob šíření signálu

Nevýhody spočívají v nutnosti ošetření vícenásobného výpočtu bloku a nutnost zajistit, aby bylo vypočítáno celé schéma (volba bloku, který vypočítá bloky zapojené na vstupech). Blok, který bude zapouzdřen ve třídě, bude tedy disponovat metodou, která způsobí výpočet bloků zapojených na jeho vstupech. Aby nedocházelo k několikanásobnému výpočtu, stačí do této metody přidat jako vstupní parametr číslo, které se bude měnit (například inkrementovat). Každé zpracování bloku tedy bude označeno číslem, které se po zpracování uloží. Při zahájení zpracování se tedy musí zkontrolovat, zda se číslo zpracování liší od čísla předchozího výpočtu. V případě, že je číslo zpracování stejné, je jasné, že tento blok a všechny bloky zapojené na vstupní porty tohoto bloku již byly zpracovány a nemusí se výpočet provádět znovu. Díky takto jednoduchému principu lze tedy realizovat výpočet celého schématu tak, že se vyvolá metoda pro zpracování dat všech bloků. Tento způsob ale z důvodů rozšíření a rychlosti programu není příliš vhodný, takže program bude vyžadovat od uživatele výběr množiny bloků, pro které při výpočtu celého schématu, bude zpracování probíhat. Tento způsob bude použit při realizaci aplikace. Po stisknutí pravého tlačítka na bloku lze nastavit, zda se má blok zpracovávat (volba „Execute when process“).

3.2.1 Automatické spouštění výpočtu

Program je možné spouštět manuálně, což znamená, že uživatel je značně omezen. Existující program ale lze rozšířit tak, aby se mohly bloky spouštět samy. Tímto uživatel získá možnost spouštět si simulaci například z externího programu (naprogramováním vhodného bloku), nebo zpracování signálu v „reálném čase“. O tom, jestli výpočet bude opravdu probíhat v reálném čase, lze polemizovat, protože výpočet bude mít pravděpodobně zpoždění větší, než bude pro systémy reálného času třeba, ale například pro zpracování obrazu z kamery nebo zvuku, tento model bude dostačující. Do množiny bloků, které se budou automaticky spouštět se blok přidá stiskem pravého tlačítka myši nad příslušným blokem a zaškrtnutím volby „Have own trigger“.

V tomto okamžiku ale přibude nutnost šířit signál i opačným směrem kvůli nutnosti znát množství informace, která se má generovat v generátorech signálu. Uveďme příklad. Mějme jednoduché zapojení dvou bloků - jeden je generátor signálu a druhý je blok, který signál ze vstupu předává do zvukové karty. Ovládání zvukové karty spočívá ve vytvoření bufferu, do kterého se zapisují data. Velikost tohoto bufferu může být různá a tak je vždy třeba generovat různé množství dat.

Výpočet bloků se realizuje rekurzivně a to dovoluje předávat informace blokům i opačným směrem.

3.2.2 Signál

Jedna z možností, jak reprezentovat data na výstupech jednotlivých bloků, je vytvoření univerzální třídy, schopné reprezentovat více datových typů a schopné přistupovat k prvkům jako k prvkům n-rozměrného pole (obdobně, jako v softwaru Matlab). Tato metoda se však ve výsledku jevila jako nevýhodná, protože se k datům přistupovalo po jednom prvku, což způsobovalo značné výkonové nároky. I v případě realizování metod, které k datům přistupovali po více prvcích (například čtení obrázku po řádcích), nebyl tento přístup ideální. Šlo sice přehledně reprezentovat obecné n-rozměrné pole, ale v případě, že se do dat měla uložit nějaká obecná informace, byla tato metoda nedostačující.

Protože by bylo vhodné, aby se mezi bloky předávaly obecné informace, konečné řešení je reprezentace dat pouze ukazatelem na data a proměnnou udávající velikost dat v bajtech. Tento způsob je sice pro málo zkušené programátory, kteří by mohli aplikaci využít, možná komplikovanější, ale je velmi rychlý a lze tvořit obecnější bloky pro práci s daty. Například lze vytvořit blok na uložení signálu do souboru. V případě, že do tohoto vstupu bude vstupovat signál, ve kterém bude bitmapa, uloží se na disk tak, jak je a lze ho otevřít v prohlížeči obrázků, který podporuje bitmapy.

Protože bude třeba segmentovat obrázek na více obrázků, bude třeba šířit více signálů najednou. Možnost realizace spojením více signálů do jednoho není příliš výhodná, protože by jednotlivé signály musely být složitě vyjímány a vkládány. Jednodušší řešení je nepředávat jeden ukazatel na signál, ale předávat pole ukazatelů, kde každý ukazatel identifikuje jeden signál.

3.3 IMPLEMENTACE BLOKŮ

Jak již bylo řečeno, bloky budou mít možnost integrovat více bloků do jednoho (virtuální blok). V případě nerealizování integrace by k implementaci schématu stačila pouze jedna třída, která by implementovala bloky. Při implementaci virtuálních bloků se řešení komplikuje, protože bude existovat více typů bloků, kde se každý bude chovat jiným způsobem. Naštěstí jazyk C++ disponuje možností polymorfního chování tříd.

Polymorfní chování spočívá ve vytvoření jednoho společného předka všem typům bloků. Tento předek bude obsahovat všechny metody, které bude každý blok muset být schopen realizovat (jak společné, tak metody, které se budou pro jednotlivé typy bloků lišit). Metody, které se pro každý blok liší, se označí jako virtuální. Z této třídy se odvodí potomci, kde každý bude definovat své vlastní chování pro tyto virtuální metody. Tímto lze realizovat tedy jednu třídu, která se bude schopna chovat různými způsoby.

Budou tedy implementovány následující třídy:

```
class CBlockItem;           // hlavní třída - definuje společné metody
class CBlockItem_Virtual;   // stará se o integraci více bloků do jednoho
class CBlockItem_Input;     // představuje spřažení s vstupním portem uvnitř
                            // virtuálního bloku
class CBlockItem_Output;    // představuje spřažení s výstupním portem uvnitř
                            // virtuálního bloku
class CBlockItem_Real;      // představuje reálný blok (provádějící výpočet)
```

3.3.1 CBlockItem

Tato rodičovská třída všech bloků obsahuje základní metody a členské proměnné, které obsahuje každý blok.

Členské metody:

```
GetInputCount               // vrátí počet portů na vstupu
GetOutputCount              // vrátí počet portů na výstupu
SetPos                      // nastaví pozici bloku
GetPos                      // vrátí pozici bloku
GetInConnectorTarget        // zjištění bloku a portu, který je připojen na vstupní
                            // port
SetInConnectorTarget        // nastavení bloku a portu, který je připojen na vstupní
                            // port
DeleteReferenceToBlock      // Odpojí od vstupů konkrétní blok
GetParentBlock              // vrátí ukazatel na rodičovský blok
SetParentBlock              // Nastaví ukazatel na rodičovský blok
```

Virtuální metody:

```
GetName                     // Vrátí jméno bloku
GetDescription              // Vrátí popis bloku
GetStatus                   // Vrátí stav bloku
GetInputPortName            // Vrátí jméno vstupního portu
GetOutputPortName           // Vrátí název výstupního portu
InitInstance                // Inicializace bloku při zavedení do schématu
ExitInstance                // Ukončení bloku po odebrání ze schématu
Start                       // Spuštění autotriggeru (automatického spouštění)
Stop                        // Zastavení autotriggeru (automatického spouštění)
ShowProperties              // Zobrazení okna s nastavením bloku
GetOutputPortData           // Vrátí ukazatel na data na výstupu portu bloku
GetOutputCriticalSectionPtr // Vrátí ukazatel na kritickou sekci dat na portu bloku
SetInputCount               // Nastaví počet vstupů
SetOutputCount              // Nastaví počet výstupů
GetLastProcessingTime       // Vrátí délku posledního výpočtu bloku
GetModifiedFlag             // Vrátí, zda byl nějaký z parametrů bloku změněn
SetModifiedFlag             // Nastaví příznak změny bloku
Serialize                   // Metoda k ukládání obsahu bloku

Process      // Postará se o zavoláni metody Process bloků na vstupu a poté zavolá
             // metodu ProcessData (rekurzivní způsob zpracování)
ProcessData  // zpracování dat v bloku
ProcessRev   // vložení zpětných dat do signálu
ClearRevMap // smazání zpětných dat ze signálu
```

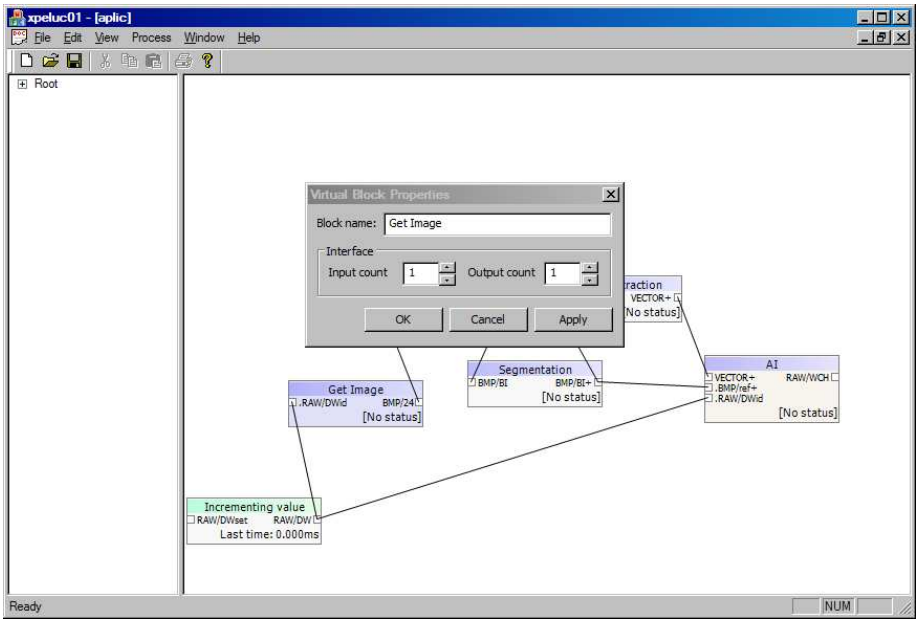
Nevirtuální metody definují společné chování. Každý blok má své umístění ve schématu, každý blok má svého rodiče, má vstupní a výstupní porty a do vstupních portů mohou být zapojeny jiné bloky. Ostatní metody jsou virtuální, což

znamená, že každá třída, která z této třídy dědí, může tyto funkce implementovat odlišně.

3.3.2 CBlockItem_Virtual

Protože ve schématu bude existovat hierarchie, je implementován typ bloku, který obsahuje další bloky udržované v seznamu. V dokumentu projektu existuje právě jedna instance tohoto bloku, který nemá rodiče (ukazatel rodiče je NULL). Ostatní tyto bloky musejí mít rodiče.

Blok nemá pojmenované vstupní a výstupní porty. K jejich pojmenování využívá pole ukazatelů na vstupní bloky (CBlockItem_Input) a výstupní bloky(CBlockItem_Output), které nesou názvy portů. O konfiguraci tohoto bloku (počet portů, název) se stará třída CBlockItem_VirtualProps, ve které je implementováno konfigurační okno. Kvůli této třídě jsou metody SetInputCount a SetOutputCount virtuální. Při jejich volání se musí v případě potřeby odpojit vstupní a výstupní bloky.

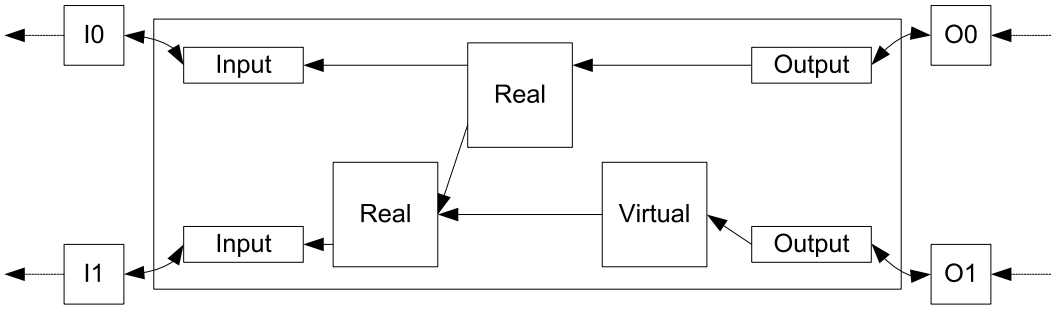


Obrázek 3.5 Konfigurační okno virtuálního bloku

Příklad vnitřní struktury virtuálního bloku je znázorněn na obrázku 3.6. Tento konkrétní virtuální blok obsahuje dva vstupy (I0, I1) a dva výstupy (O0, O1). Každý port obsahuje ukazatel na příslušnou instanci třídy vstupního, nebo výstupního bloku. V případě, že port není uvnitř bloku zapojen, obsahuje tento ukazatel hodnotu NULL.

a) V případě použitého vstupního portu musí existovat uvnitř bloku instance třídy CBlockItem_Input, ve které bude uloženo číslo portu virtuálního bloku, na který je připojen. Ukazatel na virtuální blok není třeba - vždy se jedná o rodiče bloku. Ve virtuálním bloku (v příslušném portu) je uložen ukazatel na instanci tohoto bloku. Tento blok má jeden výstup, do kterého je zapojen vstup některého z bloků uvnitř virtuálního bloku.

b) V případě použitého výstupního portu musí existovat uvnitř bloku instance třídy CBlockItem_Output, ve které bude taktéž uloženo číslo portu virtuálního bloku, na který je připojen. Opět není třeba ukazatel na virtuální blok, protože se musí jednat o rodiče bloku. Ve virtuálním bloku (v příslušném portu) je uložen ukazatel na tuto instanci bloku. Blok má pouze jeden vstup, který je zapojen do některého z bloků uvnitř virtuálního bloku.



Obrázek 3.6 Příklad vnitřní struktury virtuálního objektu

Při volání metody Process se tento objekt musí chovat odlišně. Protože virtuální bloky nemohou implementovat metodu ProcessData, která se postará o zpracování vstupních dat, metoda Process pouze zavolá metody Process bloků, jenž

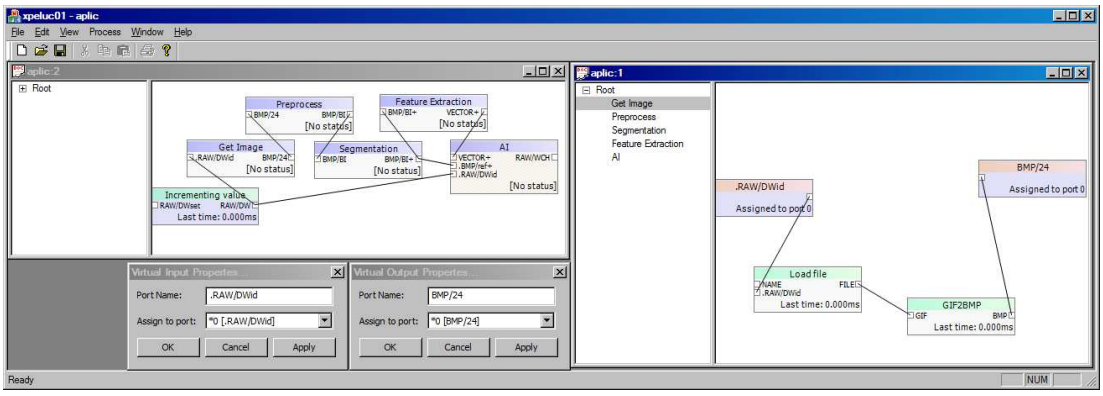
jsou připojeny uvnitř bloku k výstupním portům (jedná se o instance tříd CBlockItem_Output). Nic víc se v této metodě neděje. K zpracování bloků, které jsou připojeny na vstupu, dojde později při zavolání metody Process v instanci třídy CBlockItem_Input, které jsou spřaženy se vstupy tohoto virtuálního bloku.

3.3.3 CBlockItem_Input a CBlockItem_Output

Tyto dvě třídy slouží pouze jako cesty mezi dvěma úrovněmi hierarchie schématu. Při zpracovávání dat se starají o přechod do (ven z) virtuálního bloku. Obsahují instance tříd ke konfiguraci bloku (CBlockItem_InputProps a CBlockItem_OutputProps). V nich lze nastavovat příslušnost bloku k portu nadřazeného virtuálního bloku a název bloku (příslušného portu rodiče).

Při volání metody Process u výstupního bloku dochází pouze k pokračování zpracování na bloku, zapojeného na vstup. V případě vstupního bloku dojde ke zjištění ukazatele na blok, který je zapojen na port nadřazeného virtuálního bloku, se kterým je spřažen a na tomto bloku dojde k zavolání metody Process.

Na obrázku 3.7 jsou vidět dvě úrovně hierarchie. Jedna je kořenová (levé okno) a druhá znázorňuje obsah bloku „Get Image“. Je vidět, že názvy vstupních a výstupních portů jsou odvozovány podle názvů vstupních a výstupních bloků připojených na jednotlivé vstupní a výstupní porty virtuálního bloku.



Obrázek 3.7 Konfigurační okno vstupního a výstupního bloku

Procházení jednotlivými úrovněmi hierarchie lze tedy shrnout tak, že instance třídy CBlockItem_Virtual se stará o zanoření a instance třídy CBlockItem_Input se stará o vynoření o jednu úroveň.

3.3.4 CBlockItem_Real

Nejdůležitější bloky v celém schématu (Generátory signálu, filtry) jsou zastoupeny třídou CBlockItem_Real. Tato třída slouží jako rozhraní mezi programem a jednotlivými bloky, které jsou implementovány v dynamických knihovnách. V této třídě je tedy implementovaná metoda, která se stará o načtení dynamické knihovny a jsou zde metody, které volají metody z dynamické knihovny (např. zjištění počtu a názvů portů bloku, zjištění jména, zpracování signálu atd.).

3.4 GUI

MFC usnadňuje tvorbu aplikací jednoduchým definováním obslužných funkcí pro jednotlivé zprávy pomocí mapy zpráv. Díky tomu lze snadno realizovat obsluhu jednotlivých položek v menu, a reagování aplikace například na vstup z myši. Protože podrobně vysvětlit způsob, jak se programují aplikace pomocí aplikačního systému MFC, by bylo velmi zdlouhavé, bude naznačen postup velmi stručně.

MDI aplikace tedy znamená, že v aplikaci může být otevřeno více dokumentů najednou. Pro každý dokument je vyhrazeno jedno či více oken. Každý typ okna se nazývá pohled. Jeden dokument může mít více pohledů. Protože pro náhled na schéma postačuje jeden pohled, není třeba implementovat více pohledů pro aplikaci. Pouze z důvodů procházení stromem dokumentu do tohoto jediného pohledu je vhodné implementovat zobrazení stromové struktury celého schématu, které bude usnadňovat navigaci ve schématu. Celý pohled tedy bude znázorněn jako rozpůlené okno, kde na levé části bude stromová struktura schématu a v pravé přímí potomci bloku, vybraného ve stromu.

K přístupu k datům dokumentu je v třídě pohledu k dispozici metoda GetDocument, která vrací ukazatel na dokument.

3.4.1 Zpracování položek menu

Obsluhování položek v menu je díky MFC jednoduché. V mapě zpráv se odchytávají dva druhy zpráv. Jedna zpráva je doručována po stisknutí položky menu nebo použití klávesové zkratky (akcelerátoru). Do mapy zpráv se na odchytnutí této zprávy přidá makro ON_COMMAND.

Druhá zpráva slouží k aktualizování stavu. Aktualizace stavu položky menu probíhá v okamžiku, kdy uživatel klikne na menu a má dojít k jeho rozbalení. Aby bylo možné tuto zprávu odchytit, vytvoří se metoda s parametrem typu CCmdUI* (pomocí této instance třídy lze upravovat položku menu uvnitř metody) a do mapy zpráv se přidá makro ON_UPDATE_COMMAND_UI s příslušnými parametry. Tímto způsobem bude realizováno zablokování možnosti spuštění výpočtu v případě, že již běží, nebo zaškrtnutí některých položek v menu v případě potřeby.

3.4.2 Vykreslování schématu

Ke kreslení se využívá kontextu zařízení (třída CDC). Kontext zařízení je struktura, definující způsob vykreslování grafiky. Kontext zařízení je stavový, což znamená, že když se jednou nastaví například kreslení zeleným perem, budou se úsečky vykreslovat zeleně, dokud se tento stav nezmění. Tyto objekty (bitmapy, pera, štětce, fonty atd.) se musí odstranit z paměti, v případě, že již nejsou potřeba. Je důležité odebírat zpět tyto atributy způsobem, aby nenastal stav, kdy objekt již neexistuje, ale přesto je vybrán v kontextu. Lze to zařídit dvěma způsoby a to:

a) Výběrem nějakého globálního objektu (systémová pera, štětce atd.)
b) Při výběru objektu si uložit starý objekt a ten před odstraněním kontextu vybrat (v následujícím příkladu je to demonstrováno proměnnou pOldBitmap)

K vykreslení schématu lze využít příslušných tříd MFC. K obsluze vykreslování schématu se ve třídě pohledu přepíše virtuální metoda OnDraw. Tato metoda je volána systémem vždy, když je třeba překreslit obsah okna. Jako parametr má tato funkce ukazatel na instanci třídy CDC*, definující metody, které vykreslují

text a základní geometrické tvary (bod, úsečka, kruh) do kontextu zařízení. Protože by při kreslení přímo do okna docházelo k blikání obrazu (kvůli několikanásobnému překreslování pixelů), vykreslování se nerealizuje přímým zápisem do paměti grafického zařízení, ale nejdříve se kreslí do paměti a hotový snímek se poté překopíruje do paměti grafického zařízení.

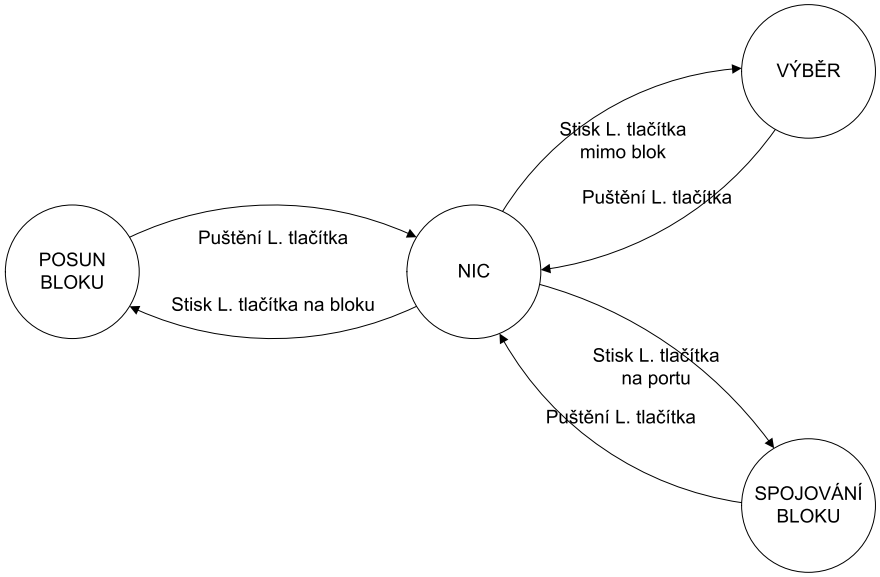
```
void CSchemeView::OnDraw(CDC* pDC)
{
  CDC dcMemory;             // kontext zařízení v paměti
  CBitmap bmp;              // bitmapa, do které se bude kreslit
  RECT rc;
  GetClientRect(&rc);       // zjištění velikosti klientské části okna
                            // vytvoření kompatibilní bitmapy s pDC s rozměry rc
  bmp.CreateCompatibleBitmap(pDC, rc.right, rc.bottom);

  // vytvoření kompatibilního kontextu s pDC
  dcMemory.CreateCompatibleDC(pDC);
  // nastavení dcMemory, aby kreslila do bmp a do pOldBitmap
  // se uloží bitmapa, která byla v kontextu vybraná předtím
  CBitmap* pOldBitmap = dcMemory.SelectObject(&bmp);
  // zde se realizuje vykreslování do kontextu dcMemory
  // překopírování obsahu kontextu v paměti do kontextu pDC
  pDC->BitBlt(0, 0, rc.right, rc.bottom, &dcMemory, 0, 0, SRCCOPY);
  // vrácení původní bitmapy, která byla v kontextu
  dcMemory.SelectObject(pOldBitmap);
}
```

Vykreslování bloku probíhá ve třídě CSchemeView postupným procházením seznamu bloků a volání metody DrawBlock s parametrem ukazatele na vykreslovaný blok, ukazatelem na instanci třídy CDC* a stavem bloku (je-li označení, či nikoliv). V metodě DrawBlock se vykreslí obdélník bloku, název, stav a jednotlivé porty s názvy. Poté, co jsou vykresleny všechny bloky, nakreslí se úsečky, které symbolizují směrování signálu jednotlivých bloků opětovným procházením seznamu s bloky a voláním metody DrawWires s parametrem ukazatele na instanci třídy CDC* a ukazatelem na instanci třídy bloku. Nakonec se nakreslí případný obdélník, který symbolizuje výběr, popřípadě úsečku, když uživatel spojuje dva bloky.

3.4.3 Ovládání myší

Systém ovládání je založen na stavovém automatu, jehož schéma je na obrázku 3.8.



Obrázek 3.8 Stavový automat ovládání myši

K implementaci ovládání myší jsou v mapě zpráv odchyceny zprávy pomocí následujících maker:

```
ON_WM_LBUTTONDOWN()     // Stisknuto levé tlačítko
ON_WM_LBUTTONUP()       // puštěno levé tlačítko
ON_WM_MOUSEMOVE()       // pohyb myši po okně
ON_WM_LBUTTONDBLCLK()   // dvojklik levého tlačítka
ON_WM_RBUTTONDOWN()     // stisknuto pravé tlačítko
```

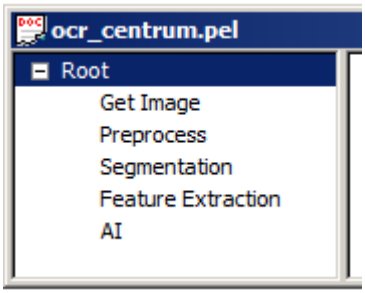
V obslužných metodách zpráv se tedy program větví přepínačem switch, kde se řeší změny stavů a akce při změně stavu (posunutí bloku, změna obdélníku výběru, změna vybraných bloků atd.).

3.4.4 Stromové zobrazení schématu

Levá část okna pohledu dokumentu znázorňuje stromový pohled na schéma (Obrázek 3.9). Je implementován ve třídě CSchemeTreeView. K naplnění tohoto stromu je implementována metoda UpdateSubTree s parametrem handlu položky ve stromovém pohledu a ukazatelem na virtuální blok, který se má aktualizovat. Tato metoda se volá rekurzivně pro potomky uvnitř metody.

Celý proces aktualizace stromu odstraňuje ze stromu položky, které ve schématu již nejsou, a přidává položky, které naopak chybí a to typu CBlockItem_Virtual, které se rozpoznávají metodou IsKindOf. Aby nedocházelo ke

kolizím s názvy, je využita vlastnost objektu CTreeView uložit spolu s položkou 32bitový parametr. Ten je nastaven jako ukazatel na objekt CBlockItem_Virtual, což poté zjednodušuje rozpoznávání objektu při kliknutí na položku.



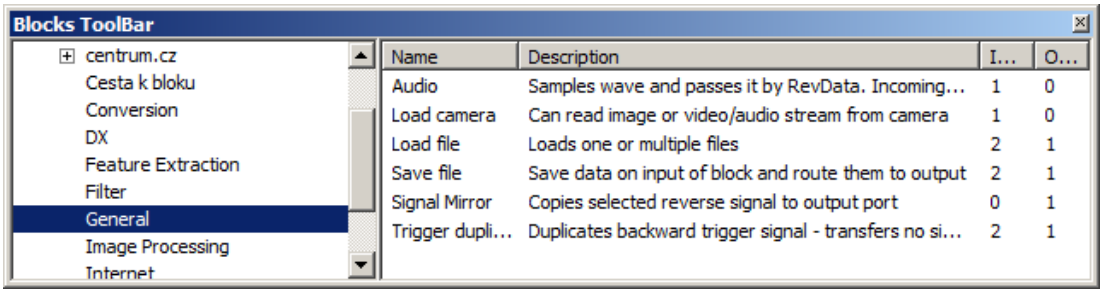
Obrázek 3.9 Zobrazení schématu ve stromové struktuře

3.4.5 Okno se seznamem bloků

Aby bylo možné jednoduše a přehledně organizovat veškeré dostupné bloky na jednom místě, v aplikaci bude existovat jedno okno, které bude nabízet dostupné bloky tříděné do stromové struktury (Obrázek 3.10). Okno se skládá ze dvou částí, kde levá část (CTreeView) znázorňuje složky s bloky a pravá část (CListView) dostupné bloky ve vybrané složce. K přemístění objektu do schématu bude využito funkce „Drag and Drop“.

O jednotlivé bloky se stará třída CBlocksDoc, jejíž jediná instance je vytvořena ve třídě CBlocksToolBar, která implementuje chování okna s těmito bloky. Třída CBlocksDoc implementuje metodu Load, která se stará o načtení všech dostupných bloků do paměti.

K reprezentaci adresářů slouží třída CBlockDirectory, která je odvozena od CObject stejně jako třída CBlockItem (třída představující obecný blok). V ní je uložen seznam objektů, které mohou být typu CObject. Protože obě tyto třídy (CBlockItem a CBlockDirectory) dědí z CObject, můžou být v tomto seznamu uloženy společně. K jejich rozlišení lze využít metodu IdKindOf. Ve třídě CBlocksDoc poté stačí mít uloženu jednu instanci třídy CBlockDirectory. Ta může obsahovat instance bloků a další instance složek.



Obrázek 3.10 Okno se seznamem dostupných bloků zařazených do stromové struktury

3.4.5.1 Drag and Drop – Drag

Aby mohly být bloky přetaženy z okna CBlocksToolBar do okna pohledu, musí být implementována funkce Drag and Drop. Okno může podporovat Drag, Drop, nebo obojí. Je třeba, aby seznam bloků umožňoval Drag. Proto se na CListView odchytává zpráva LVN_BEGINDRAG, která je doručena v případě, že uživatel začal „táhnout“ některou z položek.

Technika Drag and Drop pracuje na principu uložení dat do sdílené paměti v okně, ze kterého uživatel „táhne“ blok pryč. Protože funkce new nealokuje data do sdílené paměti, nelze druhému oknu předat pouze ukazatel na blok, který přetahujeme. Respektive lze to takto řešit pouze v jedné aplikaci, ale v případě tažení bloku do jiné aplikace (například jiné instance této aplikace) by nastala chyba, protože by se program pokoušel přistupovat do paměti, která mu nepatří.

K vyřešení tohoto problému lze využít například serializace objektu. Serializace se používá obecně u celého dokumentu k uložení dokumentu na disk. Metoda Serialize má parametr typu CArchive, který vytváří aplikační systém MFC. Konstruktor třídy CArchive je definován jako:

```
CArchive(CFile* pFile, UINT nMode, int nBufSize = 4096, void* lpBuf = NULL);
```

Jako první parametr je ukazatel na soubor, do kterého se bude zapisovat. MFC obsahuje třídu CMemFile, která se chová jako soubor, ale zapisuje jen do paměti. Díky ní lze vytvořit instanci třídy CArchive, která zapisuje data do paměti.

```
CMemFile file;
CArchive ar(&file, CArchive::store);
ar << pItem;
ar.Close();
```

K využití Drag and Drop jsou potřeba dvě struktury, které přesně definují typ dat, které jsou přetahovány. Umožňují definovat blíže, jaká data se přetahují, popřípadě definovat více formátů, ve kterých se data přetahují apod. Protože se bude implementovat pouze základní Drag and Drop, nebude uveden význam některých proměnných.

```
typedef struct FARSTRUCT tagFORMATETC {
  unsigned long cfFormat;   // identifikátor typu dat
                            // (lze registrovat jednoznačně funkcí
                            // RegisterClipboardFormat)
  DVTARGETDEVICE* ptd;      // V našem případě bude NULL
  unsigned long dwAspect;   // V našem případě bude DVASPECT_CONTENT
  long lindex;              // V našem případě bude -1
  unsigned long tymed;      // identifikuje, jak jsou data uložena.
                            // pro globální paměť bude TYMED_HGLOBAL
} FORMATETC, *LPFORMATETC;

typedef struct tagSTGMEDIUM {
  unsigned long tymed;      // typ media (opět TYMED_HGLOBAL)
  union {
    HBITMAP hBitmap;
    HMETAFILEPICT hMetaFilePict;
    HENHMETAFILE hEnhMetaFile;
    HGLOBAL hGlobal;
    LPOLESTR lpszFileName;
    Istream* pstm;
    IStorage* pstg;
  }u;                       // Handle paměti, kde jsou data uloženy
  IUnknown* pUnkForRelease; // bude NULL
} uSTGMEDIUM;
```

Data ve sdílené paměti se mohou během času podle potřeby systému přesouvat, proto se musí použít funkce GlobalLock a GlobalUnlock k přístupu do této paměti. K alokování sdílené paměti a zkopírování dat z File se použije následující kód:

```
DWORD len = (DWORD)file.GetLength();              // získá délku souboru
HANDLE hGlobal = ::GlobalAlloc(GMEM_SHARE, len);  // alokuje sdílenou paměť
BYTE* pBuf = file.Detach();                       // odebere soubor z file a vrátí
                                                  // ukazatel na data
LPVOID pData = ::GlobalLock(hGlobal);             // zamkne sdílenou paměť
memcpy(pData, pBuf, len);                         // zkopíruje data do sdílené
                                                  // paměti
GlobalUnlock(hGlobal);                            // odemkne sdílenou paměť
free(pBuf);
```

Nyní jsou data nakopírována ve sdílené paměti, takže jiné procesy k nim mají přístup. Spuštění Drag and Drop operace se provede takto:

```
COleDataSource *pSource = new COleDataSource();
pSource->CacheData(m_nClipboardFormat, &stgmed, &fmtetc);
```

```
// m_nClipboardFormat je získána funkcí RegisterClipboardFormat
pSource->DoDragDrop();
```

3.4.5.2 Drag and Drop – Drop

Aby okno mohlo být cílem Drag and Drop, musí být zaregistrováno v systému jako okno, které tuto funkci podporuje. V MFC to lze realizovat například vložením členské proměnné typu COleDropTarget do třídy okna a poté při inicializaci okna zavolat metodu COleDropTarget::Register. Tato metoda má jediný parametr a to ukazatel na instanci třídy okna CWnd*. Poté stačí již implementovat virtuální metody:

```
// uživatel právě přetáhl nějaký objekt na okno
virtual DROPEFFECT CView::OnDragEnter(COleDataObject* pDataObject,
                                      DWORD dwKeyState, CPoint point);
// uživatel se pohybuje s objektem nad oknem
virtual DROPEFFECT CView::OnDragOver(COleDataObject* pDataObject,
                                     DWORD dwKeyState, CPoint point);
// uživatel pustil tlačítko myši
virtual BOOL CView::OnDrop(COleDataObject* pDataObject,
                           DROPEFFECT dropEffect, CPoint point);
// uživatel opustil s objektem oblast okna
virtual void CView::OnDragLeave();
```

V těchto metodách lze implementovat chování Drag and Drop. Řeší se zde, zda je program schopen zpracovat obsah, který je přetahován, popřípadě jaký kurzor má mít myš při přetahováním apod. V metodě CView::OnDrop se realizuje vytvoření objektu podle dat ve sdílené paměti. Opět se vytvoří obdobným způsobem instance třídy CMemFile a CArchive a využije se operátor >> k rekonstrukci objektu. Nakonec se musí uvolnit alokovaná sdílená paměť pomocí funkce GlobalFree.

4. TVORBA EXTERNÍCH MODULŮ

V adresáři aplikace mohou být umístěny dynamické knihovny, ve kterých se mohou nacházet bloky nebo zobrazovače signálu. Kvůli přehlednosti v souborech program vyhledává všechny knihovny začínající písmenem 'b', které používá jako bloky a knihovny začínající písmenem 'v' jako zobrazovače. Proto je při vytváření knihovny zapotřebí soubor správně pojmenovat.

4.1 COMPONENT OBJECT MODEL

Aby byl program rozšiřitelný, je nutné, aby bylo možné přidávat moduly bez nutnosti kompilovat celý program znovu. Znamená to, mít možnost vytvořit externí soubory, které budou reprezentovat blok. Do dynamické knihovny nelze přímo uložit třídu (tak aby mohla být přímo externě využívána), ale pouze funkce. Tento problém řeší COM (Component Object Model.

Princip spočívá ve vytvoření čistě virtuální třídy (pure virtual), která se nazývá rozhraní [1]. Čistě virtuální třída spočívá ve vytvoření pouze deklarace třídy bez implementace. Následující ukázka je deklarace rozhraní IUnknown, ze kterého všechny třídy COM musí dědit.

```
class IUnknown
{
public:
  virtual HRESULT __stdcall QueryInterface(REFIID riid, void **ppvObject) = 0;
  virtual ULONG __stdcall AddRef( void) = 0;
  virtual ULONG __stdcall Release( void) = 0;
};
```

Protože třída nemá těla metod, nelze vytvořit instanci této třídy. Ale v případě, že bude z této třídy odvozena třída jiná a budou implementovány těla neimplementovaných metod, lze se třídu používat jako třídu IUnknown a budou se vykonávat metody třídy odvozené. Ve výsledku tedy může v externím modulu existovat funkce, která vytvoří instanci třídy odvozené od nějakého takovéhoto rozhraní a vrátí na ní ukazatel. Protože však v programu, ve kterém je knihovna načtena, není známa struktura třídy, nelze s touto třídou pracovat přímo. Lze s ní však pracovat, jako s IUnknown, protože jsou známy metody, které podporuje.

Uvedené metody sjednocují práci s moduly COM. Lépe řečeno upřesňují, jakým způsobem se s těmito moduly pracuje. Jedna z hlavních výhod je počítání referencí a automatická destrukce objektu. K tomu slouží metody AddRef a Release. Standardní chování takovéto třídy je, že obsahuje členskou proměnnou, která je při vytvoření objektu nastavena na 1. Od programátora se čeká, že v případě, že zvýší počet referencí na třídu, zavolá metodu AddRef a naopak v případě, že se snižuje počet referencí, zavolá se metoda Release. Ta se implementuje tak, že se v případě poklesu počtu referencí na nulu objekt sám zničí.

Metoda QueryInterface slouží k získání rozhraní na objekt. V aplikaci toto bude použito v případě, že blok má rozhraní pro práci s daty (zpracování dat na vstupu na výstup), rozhraní pro konfiguraci (okno s nastavením parametrů bloku), nebo například automatické spouštění simulace. Jednoduše jde o to, že objekt může a také nemusí podporovat některé věci a tato metoda slouží ke zjištění, zda objekt podporuje rozhraní a v případě, že ano, vrátí na něj ukazatel. V implementaci této metody musí programátor zajistit, aby se v instanci třídy odvozené od rozhraní, které je požadováno, uložil ukazatel na instanci této třídy. Jedině tak lze zajistit, aby mělo další rozhraní přístup k instanci této třídy.

Každé rozhraní musí mít svůj jedinečný identifikátor GUID (Globally Unique Identifier). Ke generování takovéhoto identifikátoru lze použít například program „guidgen.exe“, který je dodáván společně se softwarem Visual Studio. Podle tohoto identifikátoru se poté rozlišují jednotlivá rozhraní.

```
// {19615D2E-F617-498d-84E6-C095233E2577}
static const GUID IID_IBlock =
{ 0x19615d2e, 0xf617, 0x498d, { 0x84, 0xe6, 0xc0, 0x95, 0x23, 0x3e, 0x25, 0x77 } };

// {B951EFD6-BF6C-4c68-87D6-230EEA69D3C0}
static const GUID IID_IBlockConfig =
{ 0xb951efd6, 0xbf6c, 0x4c68, { 0x87, 0xd6, 0x23, 0xe, 0xea, 0x69, 0xd3, 0xc0 } };

// {F04B81EE-B5C6-4631-B375-E3B6082615DB}
static const GUID IID_IBlockAutoTrigger =
{ 0xf04b81ee, 0xb5c6, 0x4631, { 0xb3, 0x75, 0xe3, 0xb6, 0x8, 0x26, 0x15, 0xdb } };
```

K vytvoření instance takovéhoto objektu se obvykle používá statická metoda třídy, nebo obyčejná funkce. Výhoda statické metody spočívá v možnosti přístupu k privátním proměnným a volání privátních metod. Protože dynamická knihovna neumožňuje exportování metod, v této aplikaci se exportuje funkce:

```
__declspec(dllexport) HRESULT CreateBlockInstance(REFIID riid, void **ppv);
```

Funkce vrací typ HRESULT. Tento datový typ je ve většině případů návratový typ metod COM. Význam hodnoty se určuje podle jednotlivých bitů (popsáno v dokumentaci). Pro náš případ postačí využití makra SUCCEEDED, nebo FAILED, které vrací hodnotu typu BOOL podle toho, jestli bylo provedení metody úspěšné, nebo ne. Parametry funkce jsou REFIID riid, kterým se určuje rozhraní, které je požadováno a void **ppv, který slouží jako návratový typ funkce, kam se uloží ukazatel na vytvořenou instanci bloku.

makro	význam
S_OK	V pořádku
S_FALSE	V pořádku, ale návratová hodnota FALSE
E_NOINTERFACE	Chyba, Neexistující rozhraní
E_INVALIDARG	Chyba, špatný parametr
E_FAIL	Chyba
E_NOTIMPL	Chyba, neimplementováno
E_OUTOFMEMORY	Chyba, nedostatek paměťi

Tabulka 4.1 Některé návratové hodnoty COMu

4.2 ROZHRANÍ BLOKU

Každý blok, který bude implementován, musí disponovat přinejmenším rozhraním bloku IBlock (příloha A). Jsou to základní metody k funkci bloku, aby mohl být reprezentován v aplikaci a mohl zpracovávat data. Další funkce bloku, které jsou volitelné, blok podporovat nemusí. Protože by bylo složité pro uživatele, který bude tvořit vlastní blok implementovat všechny tyto metody rozhraní, je vhodné implementovat jednu rodičovskou třídu, která bude zprostředkovávat základní metody jako předávání názvu bloku, počtu portů, jednotlivá rozhraní atd. a uživatel pouze vytvoří třídu, která z ní bude dědit a implementuje pouze nejnutnější věci.

V programu tedy již je implementována třída CBlock, která všechny tyto základní věci realizuje. Je také dostupná třída CImpBlock, která dědí z CBlock, která ulehčuje programátorovi práci svojí přehledností (implementuje pouze nutné metody, názvy portů, bloku apod. jsou zapsány v konstantách). Podrobný návod, jak vytvořit vlastní blok, je v příloze C.

4.3 SYNCHRONIZACE

Protože je nutné rozdělit program na dvě vlákna, kde jedno je pracovní a druhé se stará o grafické rozhraní, bude k datům bloků přistupováno ze dvou různých vláken. V pracovním vlákně se budou data tvořit a ve vláknu, které obsluhuje grafické rozhraní, se budou data vizualizovat.

Každý blok má svou paměť, ve které jsou uložena data na jednotlivých portech. Z nich musejí být nějakým způsobem přenášeny informace do vizualizačního okna. K synchronizaci lze využít kritickou sekci, která zabezpečí, že k datům bude přistupovat nanejvýš jedno vlákno. Aby program pracoval efektivněji, blok bude vytvářet data následujícím způsobem:

1) Tvorba nových dat - Alokuje se potřebné místo v paměti pomocí funkce malloc a do ní se uloží výstup bloku
2) Zamknutí kritické sekce - Zamezí se přístupu ostatních vláken k datům bloku
3) Uvolnění starých dat z paměti
4) Nastavení ukazatele dat výstupu bloku na generovaná data
5) Odemknutí kritické sekce

Díky těmto pravidlům se práce s daty urychlí, protože ještě v okamžiku, kdy se vytváří data nová, může jiné vlákno ke starým datům přistupovat. Jedinou nevýhodu, kterou tento způsob má, je větší využití paměti.

4.4 BLOK - ZPRACOVÁNÍ SIGNÁLU

Hlavní funkcí bloku je zpracování signálu. Celý tento proces se děje v metodě:

```
STDMETHODIMP CImpBlock::ProcessData(const LPBLOCKDATA pInputData);
```

Metoda má jediný vstupní parametr typu LPBLOCKDATA. Jedná se ukazatel na strukturu BLOCKDATA, která je definována následovně:

```
typedef struct{
  VOID ***pDataArrayArray;          // pouziti: ppDataArray[port][vrstva]
  DWORD *dwLayersArray;             // pocet vrstev v jednotlivych portech
  DWORD **dwDataLenArrayArray;      // delka dat [port][vrstva]
}BLOCKDATA, *LPBLOCKDATA;
```

Pro alokování a uvolňování paměti musí být použit předem domluvený způsob. Lze využít funkce z rodiny malloc, popřípadě funkce new, která volá konstruktory vytvářeného datového typu. Protože se v programu předávají obecná data, je vytváření realizováno pomocí funkce malloc která vrací datový typ void*. Lze použít i funkci new, ale programátor poté musí upravit třídu CBlock, ze které dědí tato třída, v ní jsou použity funkce malloc a free.

Definice struktury signálu pro nezkušeného programátora vypadá velice obtížně, ale použití je jednoduché. Protože se jedná a konstantní parametr, není určen ke změně, a tedy pouze se z něj čte. Počet vstupních portů na bloku je znám. Počet vrstev na portu je uložen v proměnné dwLayersArray[PORT]. Počet bajtů, které jsou uloženy na dané vrstvě portu, je uložen v dwDataLenArrayArray[PORT][VRSTVA]. A nakonec pomocí pDataArrayArray[PORT][VRSTVA] lze získat data na konkrétní vrstvě portu. Data jsou typu void*, takže v případě, že programátor chce přistupovat k datům po jednotlivých bajtech, lze je přetypovat například na typ BYTE*.

K obsluhování paměti výstupních dat jsou implementovány následující metody, které usnadňují práci:

```
BOOL CBlock::DeleteBlockData();
BOOL CBlock::PrepareBlockData();
BOOL CBlock::NewBlockPortData(DWORD dwPort, DWORD dwLayers);

VOID CBlock::LockBlockData();
VOID CBlock::UnlockBlockData();
VOID CBlock::SetDataPtr(DWORD dwPort, DWORD dwLayer,
                        void* ptr, DWORD dwDataLen);
```

Jak bylo vysvětleno výše, při zápisu do výstupu bloku se musí data uzamknout. K tomuto účelu slouží metody LockBlockData a UnlockBlockData. Po zamknutí paměti se uvolní data na výstupu metodou DeleteBlockData a připraví se struktura s výstupními daty pro zápis dat metodou PrepareBlockData (alokuje se pole pro jednotlivé porty). Další metoda, která se volá, je NewBlockPortData, která se stará o vytvoření pole ukazatelů na jednotlivé vrstvy na daném portu. Tato metoda se volá pro každý výstupní port zvlášť. Do struktury se poté zapíše adresa dat pomocí metody SetDataPtr, kde se v parametru upřesní port a vrstva, kam se data zapisují. Data a jejich délka jsou v posledních dvou parametrech.

Důležité je upozornit, že při volání metody DeleteBlockData se provádí dealokace pomocí funkce free a při volání metody SetDataPtr se nevytváří kopie dat, proto je nutné alokovat paměť pomocí malloc a tyto data neuvolňovat. Data se uvolní následujícím zavoláním metody ProcessData a nebo při destrukci objektu CBlock.

4.5 BLOK - KONFIGURAČNÍ DIALOG

K nastavování parametrů bloku lze pomocí rozhraní IBlockConfig implementovat grafické rozhraní. Má jedinou metodu ShowConfig(), která je volána v případě, že uživatel požaduje otevřít konfigurační dialog. Realizace tohoto dialogu je na programátorovi. Příklad realizace konfiguračního dialogu je v příloze C.

Programátor musí zajistit synchronizaci kvůli přístupu k parametrům bloku z dvou různých vláken. Jednak se budou parametry používat z vlákna, které obsluhuje grafické rozhraní (konfigurační dialog) a jednak se budou používat v pracovním vlákně při výpočtu.

4.6 BLOK - AUTOMATICKÉ SPOUŠTĚNÍ

Aby se mohl proces spouštět automaticky podle požadavků uživatele, musí implementovat rozhraní IBlockAutoTrigger. Má pouze 2 metody a to Start a Stop.

```
class IBlockAutoTrigger : public IUnknown
{
public:
  virtual STDMETHODIMP Start(TRIGGERPROC pTrigProc, HANDLE hBlock) = 0;
  virtual STDMETHODIMP Stop() = 0;
};

typedef BOOL (WINAPI* TRIGGERPROC)(HANDLE);
```

K předání informace jakým způsobem spouštět výpočet jsou v metodě Start dva parametry. První je funkce, která se má volat v případě spuštění výpočtu a druhým je handle, který se v této funkci předá jako parametr. Funkce, která se volá, se nachází ve třídě CSchemeDoc a parametr typu HANDLE je zde přetypován na ukazatel na AUTOTRIGGERHANDLE:

```
typedef struct
{
  CDocument *pDoc;
  CBlockItem *pItem;
}AUTOTRIGGERHANDLE, *LPAUTOTRIGGERHANDLE;
```

Protože TRIGGERPROC musí být statická, aby mohla být tímto způsobem volána, nemá přístup k členským proměnným třídy. V předaném parametru je ukazatel na instanci třídy CDocument, ve kterém se blok nachází a ukazatel na instanci třídy CBlockItem, který představuje konkrétní blok. Na tomto bloku se poté spustí metoda Process, která způsobí zpracování dat obvyklým způsobem.

4.7 ZOBRAZENÍ SIGNÁLU

Po kliknutí na port bloku pravým tlačítkem myši se zobrazí nabídka dostupných zobrazovačů signálu. Rozhraní, které zobrazovač implementuje je v příloze A. Součástí aplikace jsou základní dva zobrazovače - zobrazování bitmapy (VBitmap) a zobrazování grafu (VPlot). Implementování nového zobrazovače je opět čistě na uživateli. Metoda SetData je volána vždy, když se informuje blok o změně dat, které se mají zobrazovat. V parametru této metody je ukazatel na strukturu VIEWERDATA, která je definována takto:

```
typedef struct
{
  DWORD dwLayers;         // počet vrstev
  void** ppData;          // data v jednotlivých vrstvách
  LPDWORD dwDataLenArray; // délka dat v jednotlivých vrstvách
}VIEWERDATA, *LPVIEWERDATA;
```

Ukazatel ppData obsahuje kopii jednotlivých vrstev signálu kvůli zvýšení rychlosti běhu aplikace (předpokládá se, že zobrazení dat vyžaduje zpracování dat). Protože tuto kopii vytváří stejné vlákno, jako vlákno, ve kterém se signál zobrazuje, není třeba žádné synchronizace.

Metoda Refresh je volána v okamžiku, kdy se má vykreslit obsah dat. V parametru je uvedeno, co je důvodem obnovení dat v okně. Mohou nastat dva případy parametru:

a) REFRESHTYPE_TRIGGER - Byl spuštěn jednorázový výpočet schématu (například výběrem položky Process z menu aplikace)
b) REFRESHTYPE_AUTOTRIGGER - Došlo k výpočtu pomocí automatického spuštění.

Zobrazovač by měl implementovat v menu možnost volby, na které události se bude aktualizovat.

5. APLIKACE V PRAXI

V této fázi je již možné realizovat konkrétní schéma, které bude zpracovávat data. Ačkoliv byla snaha o navrhnutí pokud možno obecného systému, následující kapitola bude zaměřena konkrétně na luštění obrázků ze serveru www.centrum.cz.

Pro přehlednější označení portů byl zvolen následující způsob:

- tečka na začátku znamená, že vstup je volitelný - není ho třeba zapojit pro správnou funkci bloku
- „RAW“ na začátku znamená, že jde o syrová data - například text, hodnota typu DWORD apod
- „DW“ znamená, že je obsažena hodnota typu DWORD
- „+“ na konci znamená, že se předpokládá vícevsrtvý signál

Samotné OCR lze rozdělit na několik fází:

a) Načtení obrázku - Získání obrázku z nějakého zdroje (soubor, internet, kamera)
b) Preprocessing - Převedení barevného modelu obrázku do binárního
c) Segmentace – Rozdělení obrázku na jednotlivé znaky
d) Extrakce příznaků - Vytvoření vektoru, který bude obsahovat informace o obrázku
e) Vyhodnocení příznaků – Učení, nebo pouze vyhodnocení příznaků například v neuronové síti
f) Postprocessing - Případné použití slovníku ke korekci výsledků
g) Uložení výsledku

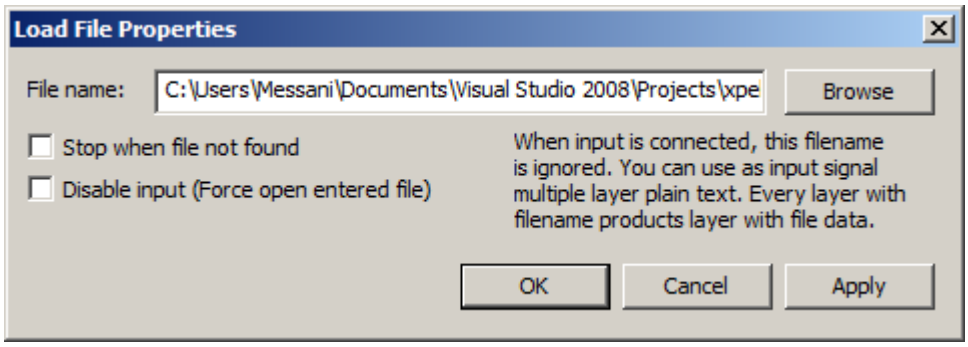
V ukázce funkčnosti programu budou provedeny všechny body kromě postprocessingu, protože na obrázcích nejsou slova, ale náhodná písmena. Také bude ukázána metoda, jak automatizovat učení v případě, že se uživatel rozhodne například změnit vektor příznaků, což znamená znova naučit neuronovou síť.

5.1 PRÁCE SE SOUBORY

5.1.1 Načtení obrázku ze souboru

Obrázek se může načítat z více zdrojů. Nejjednodušší forma získání obrázku je načtení obrázku přímo z disku. K tomuto účelu slouží modul „General/Load file“, který je umístěn v knihovně „bloadfile.dll“. Má dva vstupy: „.RAW/NAME“ a „.RAW/DWid“. První vstup umožňuje externí nastavení názvu souboru, který má být načten v unicode řetězci, ukončen znakem '\0'. Druhý vstup umožňuje vstup hodnoty DWORD, kterou lze ovlivňovat název souboru. Název souboru může obsahovat řetězec „%id%“, který je nahrazen vstupní hodnotou v portu „.RAW/DWid“.

V konfiguračním dialogu (Obrázek 5.1) jsou možnosti: zastavení výpočtu v případě, že soubor není nalezen, vypnutí vstupu s externím názvem souboru.



Obrázek 5.1 Konfigurační dialog bloku Load File

5.1.2 Načtení obrázku z internetu

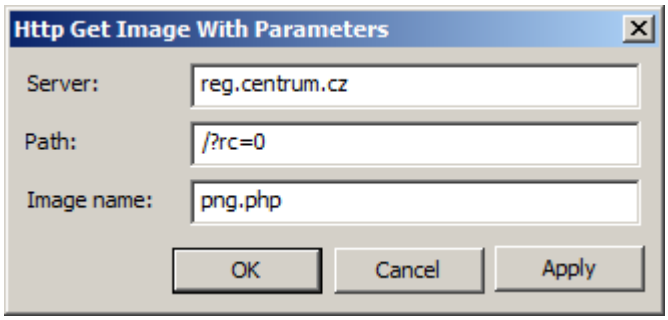
Konkrétněji jde o blok, který načte obrázek přímo ze stránky, kde se vyskytuje znehodnocený text (použita adresa: http://reg.centrum.cz/?rc=0). Je realizován v knihovně „bhttpgiwp.dll“ pod názvem „HTTP Get Image With Parameters“. Obecné načtení obrázku z internetu pomocí protokolu HTTP spočívá v odeslání požadavku GET na server a příjem a zpracování odpovědi. Protože však je zapotřebí nejdříve provést dotaz, který způsobí na straně serveru vygenerování obrázku, musí se provést celkem 2 dotazy. V konfiguračním okně (Obrázek 5.2) se nastavují tři parametry. První parametr je adresa serveru (reg.centrum.cz). Druhý

parametr je cesta ke stránce, která generuje na straně serveru obrázek (/?rc=0). Poslední parametr určuje název obrázku. Tyto tři parametry lze nastavit i externě použitím třech vstupů na bloku.

Ve zdrojovém kódu stránky představuje obrázek následující kód:

```
<img src="./png.php?x=cbb59bd50a7d1316e786067c0a80fd69&amp;y=1242305565" border="1"
alt="" />
```

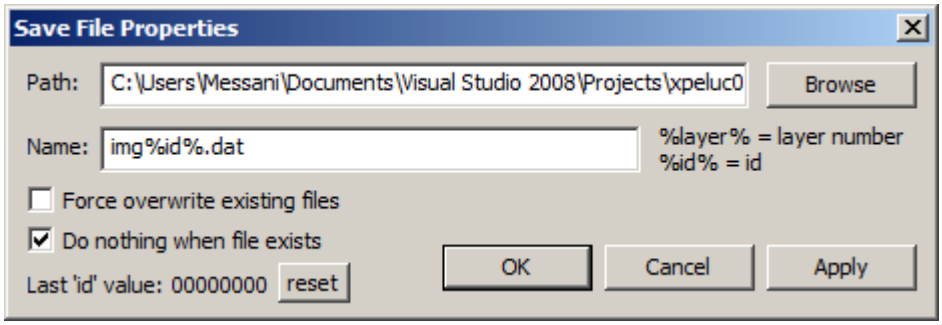
Protože parametry obrázku se mění, blok je realizován tak, že vyhledá v kódu stránky část, která je zadána uživatelem a zjistí celý odkaz. Tedy zadáme-li do třetího parametru „png.php“, program vyhledá v tomto případě celý odkaz s parametry a cestou k souboru a ten použije pro získání obrázku.



Obrázek 5.2 Konfigurační dialog bloku HTTP Get Image With Parameters

5.1.3 Uložení dat do souboru

K uložení obrázku slouží blok „Save File“ (bsavefile.dll). Princip je stejný s blokem „Load File“. Blok uloží do souboru data, které má připojeny na vstup a tyto data pošle na svůj jediný výstup. Opět má možnost vkládání číselné hodnoty do názvu souboru („%id%“) a navíc automatické vytváření hodnot bez nutnosti připojit generátor čísla na vstup. V případě existence více vrstev lze vrstvy číslovat vložením řetězce „%layer%“ do názvu souboru.



Obrázek 5.3 Konfigurační dialog bloku Save file

5.2 ZPRACOVÁNÍ OBRAZU

Po analýze bylo zjištěno, že je obraz na webovém serveru reprezentován formátem GIF. Pro zpracování je nutné, aby byla obrazová data reprezentována konkrétním formátem, se kterým se snadno pracuje. Proto byl zvolen formát BMP s 24bitovou paletou pro barevné obrázky a BMP s 8bitovou paletou pro černobílé a binární obrázky.

5.2.1 Konverze

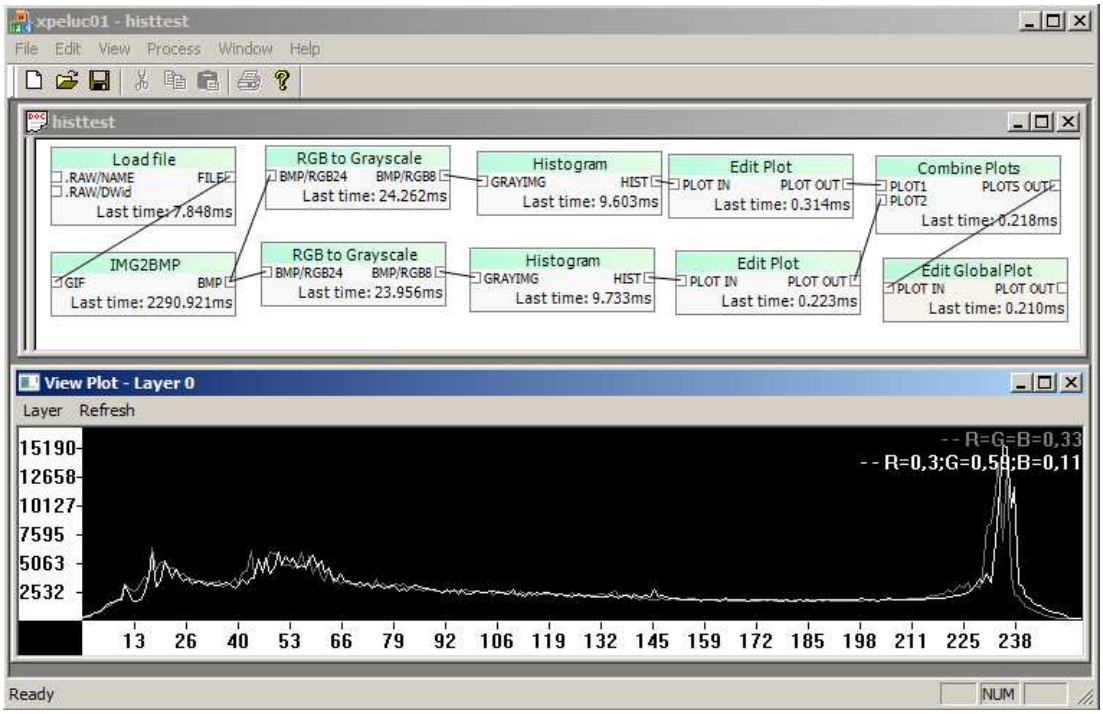
Protože GDI+ je schopné načíst obrázek z více formátů pomocí jedné funkce a exportovat ho do zvolené 24bitové bitmapy, je realizován blok s názvem „IMG2BMP“ (bimg2bmp.dll), který se o tento převod stará. Blok nemá žádné konfigurační rozhraní, pouze jeden vstup a jeden výstup.

5.2.2 Převod na černobílý snímek

Z barevného obrázku lze získat černobílý pomocí bloku „RGB to grayscale“ v knihovně „brgb2gray.dll“. Blok má jeden vstup pro 24bitovou bitmapu a jeden výstup, kde je převedený obrázek ve formátu BMP s 8bitovou paletou. Blok má konfigurační rozhraní, kde lze nastavit poměr při míchání jednotlivých barev. Lze tedy přizpůsobit převod pro některá barevná znehodnocení. Standardní koeficienty pro převod na černobílý snímek, aby byla uchována správná hodnota jasu, jsou R=0,3 G=0,59 B=0,11.

5.2.3 Práce s histogramem

K práci s histogramem je určen především blok „Histogram“, který jej generuje z obrazových dat (BMP/8bit) a pro prezentaci jsou implementovány bloky „Edit plot“, „Edit global plot“ a „Combine plots“. Pro reprezentaci histogramu byl navržen formát uvedený v příloze B. Na ukázce (Obrázek 5.4) jsou bloky, které generují dva histogramy z obrázku převedeného na černobílý pomocí různých koeficientů RGB.



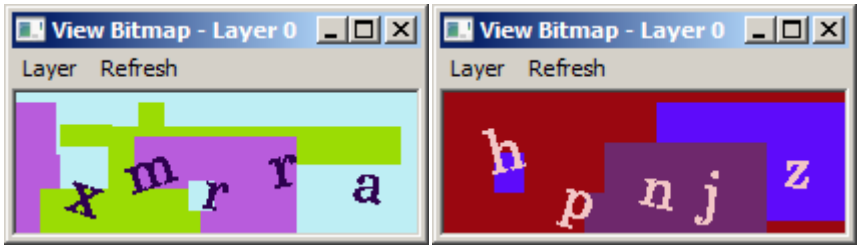
Obrázek 5.4 Ukázka vizualizace histogramu

Blok histogram generuje pouze data histogramu - tedy jen tvar křivky. Pomocí bloku „Edit Plot“ se nastaví každému grafu jeho barevný vzhled, popisek a tvar značky. Pomocí bloku „Combine Plots“ se oba tyto histogramy spojí do jednoho a blok „Edit Global Plot“ umožňuje nastavení oblasti, která má být zobrazena (v případě potřeby zobrazit detail v grafu).

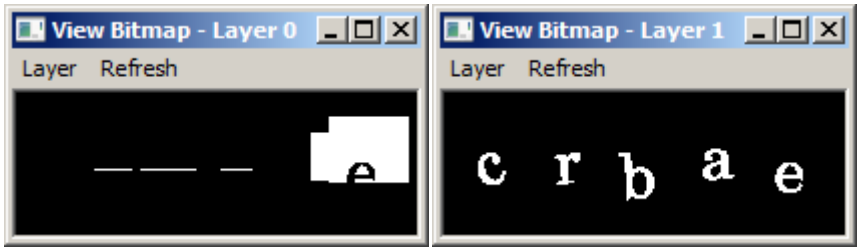
Pro OCR je důležitý ale pouze blok, který generuje histogramy, které se analyzují v dalších fázích OCR.

5.2.4 Prahování

Pro toto konkrétní znehodnocení na serveru www.centrum.cz lze využít vlastnosti, že jsou na obrázku jednobarevné objekty. Text má jednu barvu a na pozadí jsou jednobarevné obrazce (Obrázek 5.5). Vyjdeme-li z předpokladu, že text pokrývá určité procento obrazu, lze vybírat jednotlivé barvy, které pokrývají určitou plochu obrazu. Blok „Make Hist Ranges“, je implementován tak, že hledá v obrazu jednobarevné plochy, které celkově zabírají určitě procento obrazu. V konfiguračním dialogu lze tyto meze nastavit. Na obrázku 5.6 je zobrazen výstup pro konkrétní příklad, kdy jsou meze nastaveny na 1 - 15% obrazu. Jedná se o příklad, kdy blok rozpoznal jako potencionální text 2 barvy. Tento blok není obecně použitelný, proto byl zařazen do kategorie „centrum.cz/Image processing“.



Obrázek 5.5 Ukázka znehodnoceného textu z www.centrum.cz

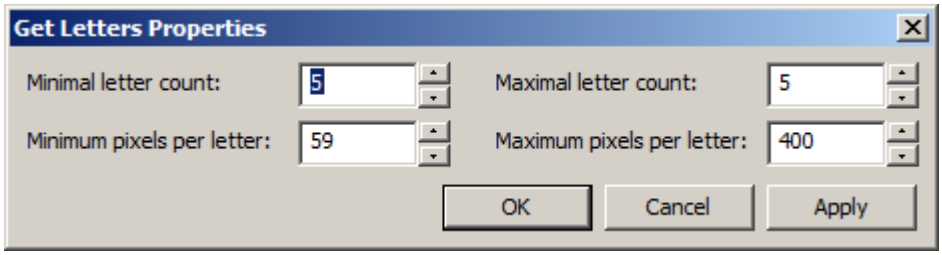


Obrázek 5.6 Ukázka vyprahovaného obrázku

5.2.5 Segmentace

Pro segmentaci byl implementován blok, který navazuje na předešlý blok, použitý pro prahování. Na obrázcích se vychází z faktu, že je znám počet znaků a každý znak má určitý minimální a maximální počet pixelů. Nastavení tohoto bloku (Obrázek 5.7) tedy spočívá v nastavení těchto mezí. Na výstupu bloku poté jsou

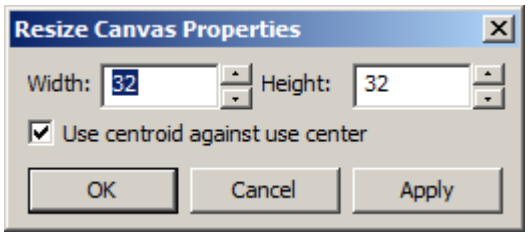
objekty, jejichž počet a množství pixelů odpovídá nastaveným parametrům. S touto metodou bylo dosaženo po testování na 400 snímcích stoprocentní úspěšnosti vyprahování snímků, avšak mohou nastat situace, kdy generátor snímku se znaky vygeneruje obrázek, který bude špatně vyhodnocen.



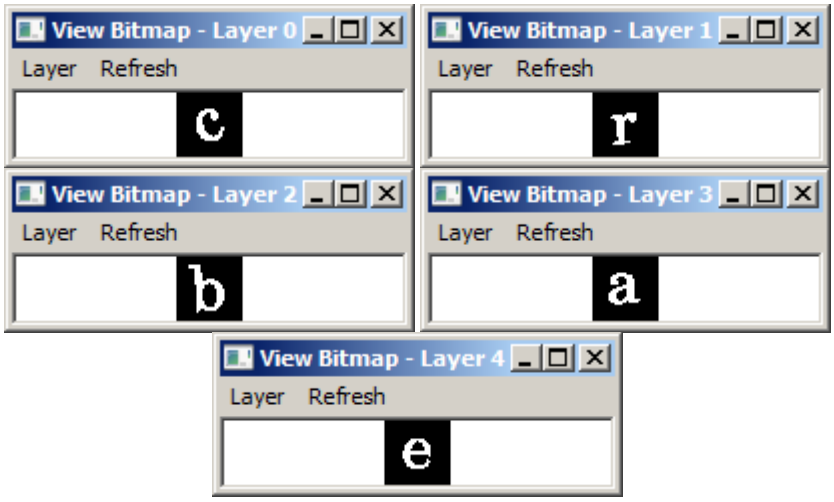
Obrázek 5.7 Nastavení bloku Get Letters

5.2.6 Změna velikosti

Ke změně velikosti obrázku bez převzorkování (pouze změna velikosti plochy obrázku) slouží blok „Resize Canvas“. Tento blok má v nastavení (Obrázek 5.8) cílovou výšku a šířku snímku. Dále lze nastavit, zda má být centrován do středu podle souřadnic, nebo podle těžiště. Příklad výstupu po zpracování obrázku je na obrázku 5.9.



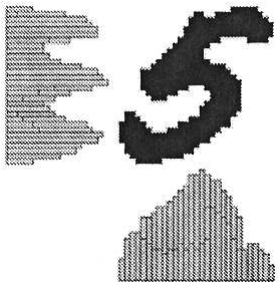
Obrázek 5.8 Nastavení bloku Resize Canvas



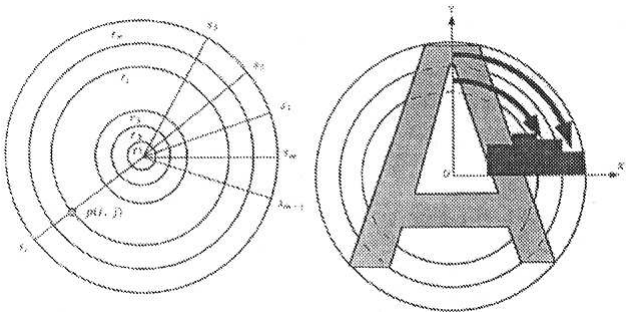
Obrázek 5.9 Jednotlivé vrstvy výstupu bloku Resize Canvas

5.3 TVORBA VEKTORU PŘÍZNAKŮ

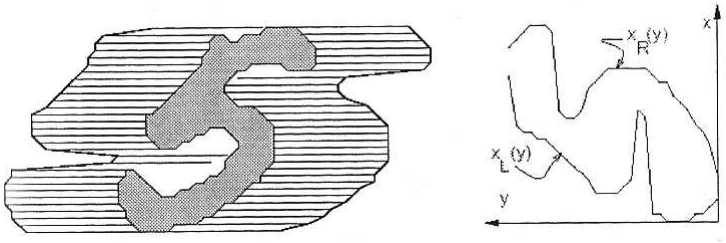
Příznakový vektor je pole reálných čísel, bude tedy reprezentován datovým typem double. Aby byl tento signál rozpoznán, jako první 4 bajty zaznamenáme do dat řetězec „VCTd“ (VeCTor of Doubles). Protože délka dat je známá, lze počet prvků v signálu jednoduše spočítat a není tedy třeba ukládat počet prvků. Ke generování příznaku vektorů jsou implementovány bloky „Gen Pixel Hist“ (Obrázek 5.10), „Gen Circle Hist“ (Obrázek 5.11) a „Gen Profile Hist“ (Obrázek 5.12). Dále byl implementován blok „Get Continous Hist“, který zjišťuje, kolikrát se jednotlivé rovnoběžky s osami protínají se znakem. Je vhodné také provést normalizaci dat (převedením celého vektoru na hodnoty v intervalu <0, 1>). Každý tento blok má jeden vstup, kam se přivede 8bitová bitmapa (popřípadě více vrstev). Ke spojení dvou vektorů slouží blok „Vector Combine“. Výsledkem použití těchto bloků je signál, kde na každé vrstvě je vektor příznaků pro jedno písmeno. Ačkoliv je příznakový vektor relativně malý, jeho použitím se pro tento konkrétní případ nedosahuje špatných výsledků. Pro čtení složitějších znaků je samozřejmě třeba použít pokročilejších metod extrakce příznaků.



Obrázek 5.10 Tvorba vektoru příznaků - histogramy pixelů v osách [2]



Obrázek 5.11 Tvorba vektoru příznaků - histogramy pixelů v kruzích [2]



Obrázek 5.12 Tvorba vektoru příznaků - profilové histogramy [2]

5.4 ROZPOZNÁNÍ ZNAKŮ

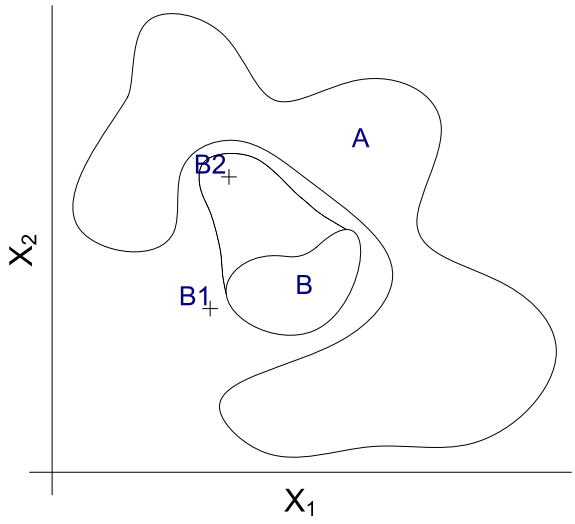
K rozpoznání znaků je pro tento konkrétní příklad implementovaná kohonenova neuronová síť. Blok má 3 vstupy, kde na první (VECTOR) je přiveden signál s příznakovým vektorem, na druhý (.BMP+) jsou přivedeny obrázky jednotlivých písmen a třetí (.RAW/WCHr) může být přivedeno správné řešení, které je uloženo v jedné vrstvě ve formě řetězce s 16bitovými znaky.

Tato kohonenova síť pracuje standardním způsobem, kdy je každý neuron představován vektorem příznaků *w* a jako vítězný vektor se volí ten, jehož

vzdálenost d od předkládaného vektoru w je nejmenší. Jde o jednoduchou variantu sítě, jejíž neurony jsou přidávány a jejich váhy se dále nemění – nejde tedy o samoorganizační mapy.

$$d_j = \sqrt{\sum_{0}^{N=1} \left(x_i - w_{ij}\right)^2} \qquad (5.1)$$

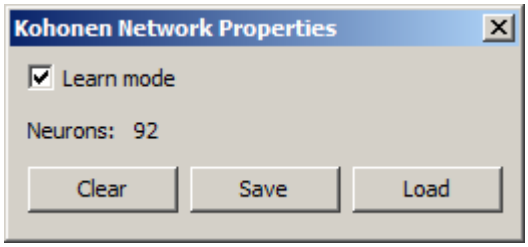
Protože se v tomto případě setkáváme s velkou variací znaků, použije se kohonenova síť, která bude mírně modifikována. Na obrázku 5.13 je graf se znázorněnými plochami, ve kterých jsou neurony znaku A, znaku B a dva vstupní vektory znaku B (B_1, B_2). V případě, že přiložíme na vstup neuronové sítě vektor B_1, její výstup je znak B, což odpovídá skutečnosti a tedy není důvod přidávat nový neuron do sítě. V případě vektoru B_2 síť rozpozná znak A. Jedná se o chybu, která se odstraní přidáním tohoto vektoru do množiny neuronů, představujících znak B. Tímto způsobem se tedy v případě potřeby bude zvětšovat počet neuronů v síti (uživatel nemusí volit žádné počáteční nastavení) a učení se projeví okamžitě.



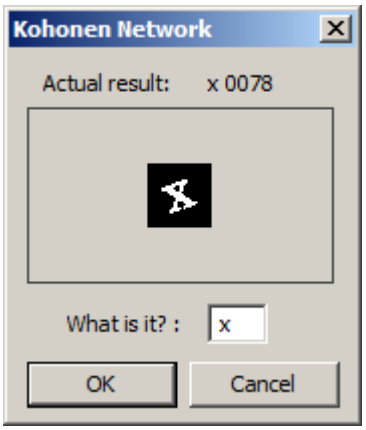
Obrázek 5.13 Modifikace kohonenovy sítě pro OCR

Obrázek 5.15 ukazuje konfigurační okno bloku, které nabízí základní operace s neuronovou síti. Volbou „Learn mode“ se určuje, zda se bude síť učit, nebo bude pouze vyhodnocovat vstupní vektory. Tlačítky „Save“ a „Load“ lze uložit a načíst aktuální stav sítě a tlačítkem „Clear“ se smažou všechny neurony. Všechny vstupní vektory musí být stejné velikosti. V případě, že se v průběhu práce se sítí změní velikost vstupního vektoru, blok automaticky otevře dialogové okno s otázkou, zda má zrušit výpočet, nebo smazat síť a změnit velikost vektoru.

V případě, že je zapnut učící režim, program se ptá formou dialogu (Obrázek 5.15) na správné řešení, ale to pouze když není na vstupu se správným řešením signál. V případě, že je, učí se podle tohoto signálu.



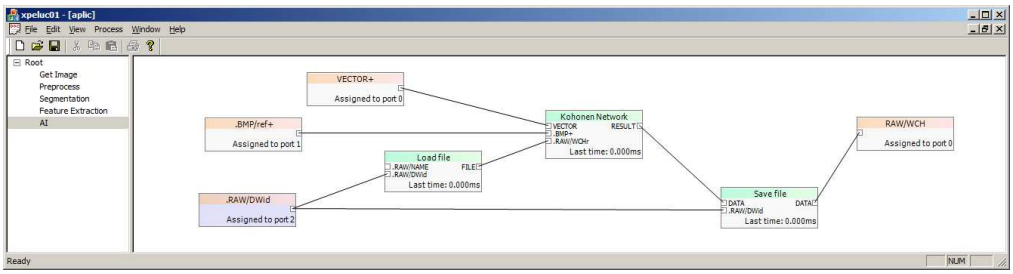
Obrázek 5.14 Konfigurační okno bloku Kohonen Network



Obrázek 5.15 Dialogové okno bloku Kohonen Network při učení

5.5 AUTOMATIZACE UČENÍ

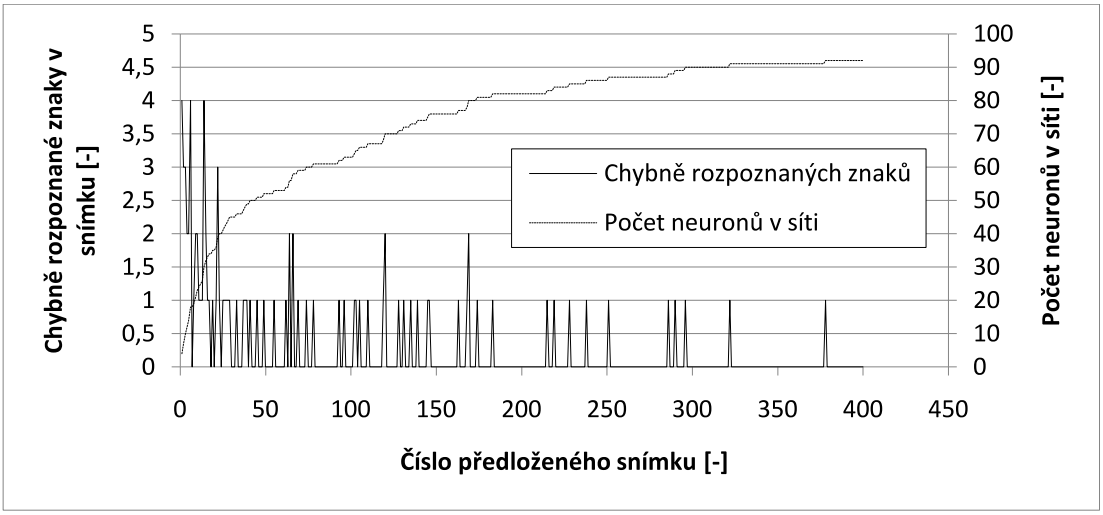
Práci usnadní schéma na obrázku 5.16. Předpokládá se, že má uživatel uloženy snímky na disku tak, aby mohly být pomocí bloku „Load file“ postupně načítány a zpracovány až po segmentaci. Na obrázku 5.16 je blok „Load File“, který se snaží načíst řešení ze souboru. Při prvním učení nenachází žádné soubory a tak na výstupu tohoto bloku není žádný signál (nesmí být zaškrtnuta volba: „Stop when file not found“). Proto se neuronová síť ptá formou dialogu, o které znaky se při učení jedná. Tento výsledek je ukládán do souborů (blok „Save File“), které při následném zpracování již blok „Load File“ načte a tak se neuronová síť dále neptá.



Obrázek 5.16 Automatizace učení

6. VÝSLEDKY PRÁCE

Na obrázku 6.1 je znázorněn průběh učení na obrázcích ze serveru centrum.cz. Na ose X je znázorněn čas, kde každá jednotka odpovídá nově předloženému obrázku s pěti znaky. Lze říci, že po učení na 300 obrázcích již OCR pracovalo velmi spolehlivě, protože během následujících 100 obrázků (500 znaků) rozpoznalo pouze dva znaky chybně (0,4% chybně rozpoznaných znaků).



Obrázek 6.1 Průběh učení

V tabulce Tabulka 6.1 je zobrazeno množství neuronů pro jednotlivé znaky po učení na 400 obrázcích. Zajímavé bylo, že autoři generátoru obrázků vyřadili použití písmen „g, i, l, o, q“. Dále je vidět, že znaky, které jsou vcelku jednoznačné, mají neuronů málo a znaky, které jsou si podobné „b-k-h“, „s-z“ jsou zastoupeny více neurony.

Znak(y)	Počet neuronů [-]
g,i,l,o,q	0
m,r,t,v	2
c,d,f,j,p,w,y	3
a,n	4
e,x	5
h,u	6
z	7
k,s	8
b	10

Tabulka 6.1 Počet neuronů na jednotlivé znaky

7. ZÁVĚR

Závěrem lze konstatovat, že cíl diplomové práce se podařilo splnit. Aplikaci lze použít nejen k realizování rozpoznání znaků z webových formulářů, ale i k mnoha jiným účelům. Jako klady hodnotím jednoduché a přehledné modelování schématu, možnost vytvořit vlastní bloky s konfiguračními dialogy a možností vícevláknového zpracování dat použitím více bloků s automatickým spouštěním. Mezi nedostatky řadím spíše jen vady na GUI, jako jsou nepřítomnost funkce „UNDO“, „COPY“ a „PASTE“, nemožnost přetahovat bloky pomocí Drag and Drop z jednoho schématu do druhého. Dále by pravděpodobně byl přínosný HEXa zobrazovač signálu.

V přiloženém CD jsou zdrojové soubory aplikace, zkompilovaná aplikace a schéma pro luštění textu ze serveru www.centrum.cz.

8. POUŽITÁ LITERATURA

[1] KRUGLINSKI, David J., SHEPHERD, George, SCOT, Wingo. *Programujeme v Microsoft Visual C++*. [s.l.] : [s.n.], 2000. 1014 s. ISBN 80-7226-362-5.

[2] YAMPOLSKIY, Roman. *Feature extraction approaches for optical character recognition*. [s.l.] : [s.n.], 2007. 228 s.

[3] NOVÁK, Petr. *Mobilní roboty*. [s.l.] : [s.n.], 2005. 256 s. ISBN 80-7300-141-1.

[4] *Builder : Informační server o programování* [online]. 2002-2003 [cit. 2008-11-15]. Dostupný z WWW: <http://builder.cz/serial91.html>.

[5] *MSDN : Visual C++ Developer Center* [online]. c2009 [cit. 2009-05-01]. Dostupný z WWW: <http://msdn.microsoft.com/cs-cz/library/60k1461a(en-us).aspx>.

[6] *Component Object Model* [online]. c2003 [cit. 2009-05-15]. Dostupný z WWW: <http://antonio.cz/com/>.

[7] *Ladislav Zezula* [online]. c2006 [cit. 2009-05-11]. Dostupný z WWW: <http://www.zezula.net/cz/prog/vytvoreni_dll.html>.

9. SEZNAM POUŽITÝCH ZKRATEK A SYMBOLŮ

COM – Component Object Model

GUI – Graphical User Interface

MDI – Multiple Document Interface

MFC – Microsoft Foundation Class

OCR – Optical Character Recognition

SDI – Single Document Interface

WinApi – Windows Application Programming Interface

10. SEZNAM PŘÍLOH

Příloha 1: Zdrojové soubory aplikace: CD, adresář: /src/

Příloha 2: Spustitelná aplikace: CD, adresář: /bin/

Příloha 3: Testovací schémata: CD, adresář: /schemes/

Příloha 4: Tento elektronický text: CD, /doc/xpeluc01.pdf

Příloha 5: Stručný uživatelský manuál: CD, /doc/manual.txt

A. DEKLARACE ROZHRANÍ

```
class IBlock : public IUnknown
{
public:
                        // získání počtu vstupů bloku
  virtual STDMETHODIMP GetInputCount(DWORD *pdwTarget) = 0;
                        // získání počtu výstupů bloku
  virtual STDMETHODIMP GetOutputCount(DWORD *pdwTarget) = 0;
                        // získání názvu bloku
  virtual STDMETHODIMP GetName(CString& rstrName) = 0;
                        // získání popisu bloku
  virtual STDMETHODIMP GetDescription(CString& rstrName) = 0;
                        // získání názvu portu na daném vstupu
  virtual STDMETHODIMP GetInputPortName(DWORD dwPort, CString& rstrName) = 0;
                        // získání názvu portu na daném výstupu
  virtual STDMETHODIMP GetOutputPortName(DWORD dwPort, CString& rstrName) = 0;
                        // inicializace bloku (volá se při umístění bloku do schématu)
  virtual STDMETHODIMP InitInstance() = 0;
                        // úklid paměti bloku (při smazání bloku ze schématu)
  virtual STDMETHODIMP ExitInstance() = 0;
                        // Zpracování reverzních dat (vysvětleno později)
  virtual STDMETHODIMP ProcessRev(DWORD dwProcessId, REVDATAMAP &rRevDataMap) = 0;
                        // Úklid reverzních dat (vysvětleno později)
  virtual STDMETHODIMP ClearRevMap(REVDATAMAP &rRevDataMapClean) = 0;
                        // Zpracování dat na vstupu na výstup
  virtual STDMETHODIMP ProcessData(const LPBLOCKDATA pInputData) = 0;
                        // získání ukazatele na výstup bloku na daném portu
  virtual STDMETHODIMP_(const PORTDATA) GetOutputPortData(DWORD dwPort) const = 0;
                        // získání cesty k bloku v nabídce dostupných bloků
  virtual STDMETHODIMP GetBlockPath(CString &rstrPath) = 0;
                        // Serializace (volá se při ukládání schématu na disk)
  virtual STDMETHODIMP_(void) Serialize(CArchive &ar) = 0;
                        // vrácení kritické sekce, která zamyká data
  virtual STDMETHODIMP_(LPCRITICAL_SECTION) GetCriticalSectionPtr() = 0;
                        // získá, zda byly parametry bloku změněny
  virtual STDMETHODIMP_(BOOL) GetModifiedFlag() = 0;
                        // nastaví, zda byly oarametry bloku změněny
  virtual STDMETHODIMP_(VOID) SetModifiedFlag(BOOL bModified) = 0;
};

class IBlockConfig : public IUnknown
{
public:
                        // požaduje se zobrazení konfiguračního okna
        virtual STDMETHODIMP ShowConfig() = 0;
};

class IBlockAutoTrigger : public IUnknown
{
public:
                        // požaduje se start automatického spouštění
        virtual STDMETHODIMP Start(TRIGGERPROC pTrigProc, HANDLE hBlock) = 0;
                            // požaduje se ukončení automatického spouštění
        virtual STDMETHODIMP Stop() = 0;
};

class IViewer : public IUnknown
{
public:
        virtual STDMETHODIMP SetData(LPVIEWERDATA pData) = 0;
        virtual STDMETHODIMP ShowViewer(CDocument* pDocument, CWnd* pMainWnd) = 0;
        virtual STDMETHODIMP Refresh(DWORD dwType) = 0;
        virtual STDMETHODIMP GetName(CString &rstrName) = 0;
        virtual STDMETHODIMP GetDescription(CString &rstrDescription) = 0;
};
```

B. STRUKTURA DAT S GRAFEM

offset	struktura	datový typ	délka	popis
0x00	PLOTFILEHEADER	char[4]	4b	musí být řetězec "plot" pro data křivky "glbl" pro data grafu
				následuje struktura PLOTHEADER
				0x00000000 značí konec souboru - nic nenásleduje
0x04	PLOTHEADER	DATATYPE	4b	typ dat, které následují
0x08	.	ITEMTYPE	4b	datový typ dat, které následují
0x0c	.	DWORD	4b	počet prvků, které následují
				V případě, že DATATYPE je DATATYPE_EOP,
				nenásledují data, ale opět PLOTFILEEHADER
0x10				data... následovaná strukturou PLOTHEADER

```
enum ITEMTYPE {
  ITEMTYPE_BYTE,      // hodnoty typu BYTE
  ITEMTYPE_WORD,      // hodnoty typu WORD
  ITEMTYPE_DWORD,     // hodnoty typu DWORD
  ITEMTYPE_FLOAT,     // hodnoty typu FLOAT
  ITEMTYPE_DOUBLE     // hodnoty typu DOUBLE
};
```

deklarace	popis	výskyt	délka [byte]
enum DATATYPE {			
DATATYPE_EOP = 0,	konec záznamu o křivce	p g	0
DATATYPE_XVALUES,	pole hodnot X-ových souřadnic	p	ph*dh
DATATYPE_YVALUES,	pole hodnot Y-ových souřadnic	p	ph*dh
DATATYPE_XLABEL,	název X-ové osy (neimplementováno)	g	?
DATATYPE_YLABEL,	název Y-ové osy (neimplementováno)	g	?
DATATYPE_LEGEND,	legenda	p	?
DATATYPE_COLOR,	barva	p	1*sizeof(RGBQUAD)
DATATYPE_MARK,	značka ('.', 'x', '+', 'o', '-')	p	1
DATATYPE_GRAPHNAME,	název grafu (neimplementováno)	g	?
DATATYPE_XRANGE,	rozsah na X-ové ose	g	2*sizeof(DOUBLE)
DATATYPE_YRANGE	rozsah na Y-ové ose	g	2*sizeof(DOUBLE)
};			

výstyt: p = plot, g = global
délka: ph = počet hodnot, dh = délka jedné hodnoty v bajtech

C. PŘIDÁNÍ BLOKU

Demonstrujme přidání modulu, který vynásobí všechny pixely hodnotou, která bude zadána uživatelem. Předpokládá se, že je otevřen projekt aplikace ve Visual Studiu.

Do řešení (solution) se přidá nový projekt (File - Add - New Project). Vybereme typ MFC DLL a nazveme ho „multiply“. Typ knihovny zvolíme jako „Regular DLL using shared MFC DLL“ (v případě, že bychom chtěli knihovnu použít externě bez nainstalovaného Visual Studia, bude potřeba knihovny MFC s projektem staticky slinkovat). Do projektu přidáme z adresáře „global“ soubory „Block.h“, „Block.cpp“, „iblock.h“. Dále nakopírujeme do adresáře projektu a přidáme do projektu soubory podle požadavků (* jsou označeny soubory, které nakopírujeme v tomto případě):

```
* imp_block.h, imp_block.cpp               // podpora zpracování dat - musí být vždy
  imp_autotrigger.h, imp_autotrigger.cpp   // podpora automatického spouštění
* imp_config.h, imp_config.cpp             // podpora konfigurace bloku přes GUI
* imp_configwindow.h, imp_configwindow.cpp // podpora třída pro práci s konfig. oknem
```

V souboru „bmultiply.def“ připíšeme název funkce, která vytváří instanci bloku.

```
LIBRARY      "bmultiply"
EXPORTS
  CreateBlockInstance ; Tento řádek přidáme(komentáře značeny středníkem)
```

Do třídy CImpBlock přidáme členskou proměnnou typu float, kterou se budou násobit pixely:

```
float m_fMultiply;
```

V souboru „ imp_block.cpp“ nastavíme informace o bloku:

```
const LPWSTR g_pName = L"Multiply pixels";     // název
const LPWSTR g_pDesc = L"Some description..."; // popis
const LPWSTR g_pPath = L"User blocks";         // kategorie
#define __BLOCKHAVECONFIGWINDOW                // blok má konfigurační okno
const DWORD g_dwInPortCount = 1;               // jeden vstup
const DWORD g_dwOutPortCount = 1;              // jeden výstup
const WCHAR *g_pInputNameArray[] = {L"BMP8"};  // název vstupního portu
const WCHAR *g_pOutputNameArray[] = {L"BMP8"}; // název výstupního portu
```

V konstruktoru CImpBlock inicializujeme proměnnou m_fMultiply na 1:

```
m_fMultiply = 1;
```

Vyplníme metodu Serialize, aby se uložila hodnota m_fMultiply:

```
STDMETHODIMP_(void) CImpBlock::Serialize(CArchive &ar)
{
  if ( ar.IsStoring() ) // ukladame
    ar << m_fMultiply;
  else                  // nahravame
    ar >> m_fMultiply;
}
```

Vyplníme metodu zpracování dat:

```
STDMETHODIMP CImpBlock::ProcessData(const LPBLOCKDATA pInputData)
{
  std::deque<std::pair<LPVOID, DWORD> > mapData;
  for ( DWORD dwLayer = 0; dwLayer < pInputData->dwLayersArray[0]; dwLayer++ )
    if ( pInputData->dwDataLenArrayArray[0][dwLayer] > sizeof(BITMAPFILEHEADER) ) {
      BITMAPFILEHEADER *bfh = (BITMAPFILEHEADER*)pInputData->pDataArrayArray[0][dwLayer];
      if ( 0x4d42 == bfh->bfType ) {
        BITMAPINFOHEADER *bih = (BITMAPINFOHEADER*)&bfh[1];
        if ( bih->biBitCount == 8 && bih->biCompression == 0 && bih->biPlanes == 1 ) {
          BYTE* pPixelsSrc = (BYTE*)&((BYTE*)&bih[1])[sizeof(RGBQUAD)*256];
          BYTE* pDataTgt = (BYTE*)malloc(pInputData->dwDataLenArrayArray[0][dwLayer]);
          DWORD dwHeaderLen = (int)pPixelsSrc-(int)bfh;
          memcpy(pDataTgt, bfh, dwHeaderLen);
          BYTE *pPixelsTgt = &pDataTgt[dwHeaderLen];
          for ( int i = 0; i < bih->biHeight * bih->biWidth; i++ ) {
            float fData = pPixelsSrc[i] * m_fMultiply;
            pPixelsTgt[i] = (fData < 0)?(0):( (fData>255) ?(255):((UCHAR)fData) );
          }
          mapData.push_back(std::pair<LPVOID, DWORD>(pDataTgt,
                            pInputData->dwDataLenArrayArray[0][dwLayer]));
        }
      }
  }
  LockBlockData();
  DeleteBlockData();
  PrepareBlockData();
  NewBlockPortData(0, mapData.size());
  DWORD dwUsedLayers = 0;
  while ( !mapData.empty() ) {
    SetDataPtr(0, dwUsedLayers++, (*mapData.begin()).first, (*mapData.begin()).second);
    mapData.pop_front();
  }
  UnlockBlockData();
  return S_OK;
}
```

V záložce Resource View v projektu „multiply“ přidáme dialog - název mu necháme IDD_DIALOG1. Tlačítko „Apply“ pojmenujeme jako IDC_APPLY, editační pole jako IDC_VALUE. V CImpBlockConfigWindow::DoDataExchange do kritické sekce vložíme řádek pro výměnu kopírovaní parametru do proměnné a naopak:

```
void CImpBlockConfigWindow::DoDataExchange(CDataExchange* pDX)
{
  EnterCriticalSection(&m_pImpBlock->m_cs);
  CDialog::DoDataExchange(pDX);
  DDX_Text(pDX, IDC_VALUE, m_pImpBlock->m_fMultiply); // Tento řádek vložíme
  LeaveCriticalSection(&m_pImpBlock->m_cs);
  if ( pDX->m_bSaveAndValidate )
    m_pImpBlock->SetModifiedFlag(TRUE);
}
```

V případě, že jsme do prvku dali tlačítko Apply, můžeme odkomentovat makro v mapě zpráv k odchycení zprávy stisku tohoto tlačítka (metoda pro zpracování je v tomto souboru již implementována)

```
BEGIN_MESSAGE_MAP(CImpBlockConfigWindow, CDialog)
  ON_BN_CLICKED(IDC_APPLY, &CImpBlockConfigWindow::OnBnClickedApply) // Toto odkomentovat
END_MESSAGE_MAP()
```

Po těchto pár krocích stačí jen blok zkompilovat a blok bude plně funkční.