VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

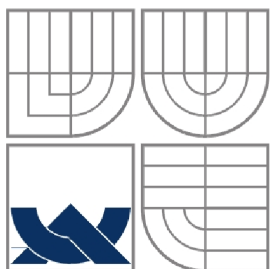
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

VIZUÁLNÍ NÁSTROJ PRO TVORBU APLIKACÍ VE ZPRACOVÁNÍ OBRAZU

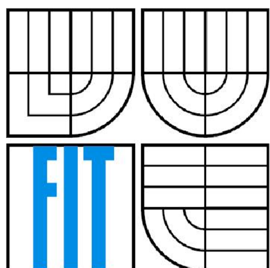
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. Lukáš Horák
AUTHOR

BRNO 2010



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

VIZUÁLNÍ NÁSTROJ PRO TVORBU APLIKACÍ VE ZPRACOVÁNÍ OBRAZU

GRAPHICAL TOOL FOR IMAGE PROCESSING APPLICATIONS CREATION

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. Lukáš Horák
AUTHOR

VEDOUCÍ PRÁCE Ing. Michal Španěl
SUPERVISOR

BRNO 2010

Vizuální nástroj pro tvorbu aplikací ve zpracování obrazu

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Ing. Michala Španěla. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……………………
Lukáš Horák
24.5.2010

Poděkování

Děkuji Ing. Michalu Španělovi za poskytnuté rady a odborný dohled při zpracování této diplomové práce.

© Lukáš Horák, 2010
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Zadání diplomové práce/8061/2009/xhorak48

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačové grafiky a multimédií   Akademický rok 2009/2010

Zadání diplomové práce

Řešitel: **Horák Lukáš, Bc.**

Obor: Počítačová grafika a multimédia

Téma: **Vizuální nástroj pro tvorbu aplikací ve zpracování obrazu**
Graphical Tool for Image Processing Applications Creation

Kategorie: Počítačová grafika

Pokyny:

1. Prostudujte základy zpracování obrazu.
2. Zorientujte se v současných knihovnách a nástrojích pro zpracování obrazu. Porovnejte jejich vlastnosti z pohledu uživatelského rozhranní a tvorby aplikací.
3. Navrhněte jednoduchý nástroj pro vizuální "skládání" aplikace zpracovávající vstupní obraz (např. vytvoření sekvence operací - bloků).
4. Experimentujte s Vaší implementací a případně navrhněte vlastní modifikace metod.
5. Porovnejte dosažené výsledky a diskutujte možnosti budoucího vývoje.
6. Vytvořte stručný plakát prezentující Vaši diplomovou práci, její cíle a výsledky.

Literatura:

- Dle pokynů vedoucího.

Při obhajobě semestrální části diplomového projektu je požadováno:

- Splnění prvních tří bodů zadání.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci ročníkového a semestrálního projektu (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Španěl Michal, Ing.**, UPGM FIT VUT

Datum zadání: 21. září 2009

Datum odevzdání: 26. května 2010

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačové grafiky a multimédií
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Jan Černocký
vedoucí ústavu

Abstrakt

Tato diplomová práce se zabývá vytvořením nástroje pro vizuální tvorbu aplikací ve zpracování obrazu, které se provádí pomocí grafického propojování funkčních bloků. Grafické propojování je realizováno pomocí tzv. node editoru, který je implementován v Qt frameworku. Qt framework je využit napříč celým projektem, pro uživatelské rozhraní a správu dynamických knihoven. Knihovna OpenCV je využita jako základ pro zpracování obrazu. Tento nástroj je možné rozšiřovat o další metody zpracování pomocí pluginů, umístěných v dynamických knihovnách. Dále interpretuje navrženou aplikaci, návrh aplikace je podpořen režimem náhledu na změnu v reálném čase. Obsahuje i historii akcí (Undo/Redo systém).

Abstract

This term project deals with creating visual tool for building image processing applications that is performed using the graphical interconnection of functional blocks. Graphical interconnection is realised by so-called node editor that is implemented in Qt framework. This framework is used across the project – for the user interface and management of dynamic libraries. OpenCV library is used as the basis for image processing, this tool can be extended to other methods of image processing using plugins included in the dynamic libraries. The tool interprets the proposed application, the proposal application is supported by the preview mode of the change in real time. The tool also contains a history of actions (Undo/Redo system).

Klíčová slova

vizuální nástroj pro vytváření aplikací, aplikace pro zpracování obrazu, dynamické knihovny, pluginy, node editor, historie akcí

Keywords

visual tool for building applications, image processing application, dynamic linked libraries, plugins, node editor, history events

Citace

Lukáš Horák: Vizuální nástroj pro tvorbu aplikací ve zpracování obrazu, diplomová práce, Brno, FIT VUT v Brně, 2010

Obsah

Obsah ... 1
1 Úvod ... 3
2 Dynamické knihovny ... 5
3 Nástroje podobného principu ... 7
3.1 DirectShow Filter Graph Editor ... 7
3.2 Blender ... 8
3.3 Pixel Bender Blender Prototype ... 9
3.4 TecnoFreak Animation System ... 10
4 Návrh řešení ... 12
4.1 Rozdělení uživatelského rozhraní a jádra aplikace ... 12
4.2 Návrh uživatelského rozhraní editoru ... 13
4.3 Rozšiřování pomocí dynamických knihoven ... 15
4.4 Přehled bloků ... 15
4.5 Nastavení a propojení funkčních bloků ... 16
4.6 Historie akcí ... 17
4.7 Interpretace navržené aplikace ... 17
4.8 Datový formát ... 18
4.9 Nástroj pro spuštění navržené aplikace ... 18
4.10 Nástroj pro validaci aplikace ... 19
5 Implementace ... 20
5.1 OpenCV ... 20
5.2 Qt framework ... 20
5.3 Výjimky ... 21
5.4 Záznam událostí ... 22
5.5 Část jádra s podporou pluginů ... 22
5.6 Zabudované kodeky ... 27
5.7 Část jádra pro realizaci programu pro zpracování obrazu ... 29
5.8 Vytvoření plánu ... 31
5.9 Validace dokumentu ... 31
5.10 Spuštění navržené aplikace ... 32
5.11 Uživatelské rozhraní ... 32
5.12 Použití NodeEditorWidget v projektu ... 37
5.13 Historie akcí ... 39
6 Vytvoření pluginů ... 42

6.1 Vytvoření nového enkodéru....42
6.2 Vytvoření nového dekodéru....42
6.3 Vytvoření továrny enkodéru či dekodéru....43
6.4 Vytvoření nového operátoru a jeho továrny....43
6.5 Vytvoření nového datového typu....44
6.6 Hlavní modul dynamické knihovny....44
7 Výsledky....46
7.1 Vyhodnocení měření stráveného času nad režií při zpracování....48
8 Závěr....51
Literatura....52
Seznam příloh....53
Příloha 1. Uživatelský manuál....54
Příloha 2. Implementace pluginu pro zpracování obrazu (ukázka)....57

1 Úvod

V dnešní uspěchané době se každý snaží zefektivnit svoji nebo cizí práci. Můžeme se pokusit o něco podobného i u návrhu a vytvoření aplikace ve zpracování obrazu.

Většinou se tyto aplikace píší v jazycích C/C++, což bývá časově náročnější. Tyto jazyky se používají především kvůli jejich rychlosti (výsledná aplikace je optimalizovaná a není interpretovaná) a paměťové náročnosti.

Tato práce se zabývá vytvořením nástroje, který usnadní vytvoření především jednoúčelových nebo experimentálních aplikací pro zpracování obrazu, které se bude provádět i dávkově. Chceme například fotografie právě získané z digitálního fotoaparátu publikovat na web, vytvoříme jejich náhledy a u originálů zmenšíme velikost se zachováním poměru stran a opatříme je vodoznakem, případně upravíme pomocí nějakého filtru. Dále můžeme vyextrahovat hrany z videa nebo se budeme snažit vyvinout vylepšené chování určitého filtru pomocí speciálně nastaveného předzpracování.

Budeme-li potřebovat rozšířit funkčnost tohoto nástroje, je nutné připsat zdrojový kód potřebné funkce. Chování výsledného programu pro zpracování obrazu se bude vytvářet, přesněji řečeno vizuálně skládat. V nástroji je k dispozici pracovní plocha, na kterou vkládáte bloky, a ty následně mezi sebou propojíte. O vstup se starají vstupní bloky, o zpracování zase bloky pro zpracování obrazu a o výstup nakonec bloky výstupní.

Představme si kytaristu a jeho kytarové efekty, přesněji, jak má tyto efekty zapojené. Vstupem je kytara, kabel z ní vede do první krabičky (v našem případě bloky pro zpracování) s efektem booster, který má ovládací potenciometry, z této krabičky vede kabel do další krabičky s efektem delay, a tak postupujeme dále až ke kytarovému kombu (zesilovač s reproduktorem), který můžeme chápat jako výstupní blok.

Nástroj, který je předmětem této práce, bude pracovat podobně, jen v počítači a ne na pódiu. Bude možné zavádět paralelní cesty, získávat data z více vstupů, ze souboru, z kamery, různě je kombinovat a ukládat data do více výstupů. Záleží na kreativitě uživatele při vytváření cílového programu. Ten se bude vytvářet vizuálně, tedy umístíme na plochu bloky, které budeme potřebovat. Poté je budeme mezi sebou je propojovat a různě nastavovat tak, aby dělaly přesně to, co budeme potřebovat. K získání zpětné vazby, k tomu, co jsme vlastně vytvořili, nám poslouží nástroj pro ladění. Tím je blok, který ve svém těle zobrazuje to, co se nachází na jeho vstupu.

Výsledek, který bude podobný jako propojení u kytaristy, půjde interpretovat přímo v nástroji a v podpůrné konzolové aplikaci, která se bude hodit především pro dávkové zpracování. Také budete moci vygenerovat zdrojový kód v jazycích C/C++, který se zkompiluje (potřeboval by ovšem ke své funkci i knihovny a pluginy tohoto nástroje, ale o tom až později).

Během času stráveného nad touto diplomovou prací se pravděpodobně nepodaří naplnit veškeré vize. Ovšem nejdůležitější milník musí být dosažen, a to, jak již název napovídá, vizuální nástroj pro tvorbu aplikací ve zpracování obrazu, tedy nástroj umožňující skládat bloky s určitým chováním a tyto bloky propojovat, čímž se docílí výsledné deterministické chování výsledné aplikace.

Tento dokument je rozdělen do několika částí. První část se věnuje teoretickému rozboru dynamických knihoven, které budou tvořit významnou část projektu již od návrhu. Následuje ukázka několika aplikací, jejichž smyslem je pomocí vizuálního skládání bloků a jejich propojení docílit jistého chování. Tento způsob práce je společný i pro tento projekt. V další části dokumentu je popsán návrh aplikace, rozdělení na jádro, uživatelské rozhraní a využití pluginů. Dále navazuje popis implementace jádra, uživatelského rozhraní, ovládacího prvku umožňující vizuální skládání a tvorba pluginů. Implementaci je věnováno nejvíce prostoru, popisuje jednotlivé fáze realizace výsledného programu. Dokument je ukončen shrnutím schopností a výsledků dosažené práce. K dokumentu je přiložen jednoduchý uživatelský manuál. Součástí zadání je prezentační plakát, který je podpořen o internetové stránky s možností si projekt stáhnout.

Aby nedocházelo k omylům, v textu se slovem aplikace myslí navrhovaná aplikace pro zpracování obrazu v nástroji, který je předmětem této diplomové práce.

2 Dynamické knihovny

Kapitola je věnována teorii a popisu dynamických knihoven. Tyto knihovny v tomto projektu zabírají patřičné místo již od návrhu. Zasvěcení čtenáři mohou touto kapitolou lehce projít.

V operačním systému jsou většinou dva hlavní typy souborů obsahující kompilovaný kód. Prvním typem je spustitelný soubor (u OS MS Windows s příponou exe) obsahující program (kód, který se vykoná ihned po spuštění tohoto souboru).

Druhým typem jsou dynamické knihovny (u OS MS Windows s příponou dll, u OS Linux s příponou so). Nejedná se však o spustitelný soubor jako takový. Dynamické knihovny obsahují tzv. exportované funkce či přístupové body knihovny. Tyto funkce lze v kódu jiného programu zpřístupnit a plně využívat.

Zpřístupnění těchto funkcí se provádí staticky či dynamicky. Dynamický přístup se využívá především u programů, které se tímto způsobem rozšiřují o určitou funkčnost. Během programu v určitém okamžiku načteme knihovnu a získáme ukazatele na potřebné funkce. Knihovna musí být načtena, dokud tyto funkce využíváme. Tento princip můžeme vidět u různých pluginů, např. pro program Adobe Photoshop, do této kategorie částečně spadají i rozšíření pro OpenGL (získává se ukazatel na rozšiřující funkce ARB, EXT...).

Statickým přístupem se rozumí princip podobný jako u statických knihoven, ale kód knihovny není přilinkován k výslednému programu. Při spuštění programu se do paměti načte tato dynamická knihovna, jejíž funkce jsou automaticky k dispozici bez složitého získání ukazatelů.

Dynamická knihovna nemusí obsahovat jen funkce, ale může obsahovat i třídy. Třídy je velice snadné využívat právě statickým přístupem.

Napřed si ukážeme, jak se k těmto funkcím přistupuje v jazyce C++ s využitím Qt frameworku. Obecný postup v jazyce C/C++ je ve všech operačních systémech stejný. Nejprve získáme ukazatel na dynamickou knihovnu, která se v případě potřeby načte do paměti. Poté získáme ukazatel na funkci z této knihovny dle jejího jména nebo jejího pořadového čísla. Dynamická knihovna musí být načtena během používání této funkce.

```
// Definice typu ukazatele na funkci z knihovny
typedef bool (*CheckVer)(int number);

// Načtení dynamické knihovny
QLibrary *lib = new QLibrary(filename, NULL);
if (lib->load() == false) return false;

// Získání ukazatele na funkci v knihovně
CheckVer checkVer = (CheckVer)lib->resolve("CheckVersion");
if (checkVer == NULL) return false;

// Volání funkce
bool result = checkVer(30);
```

Ukázka 1: Získání ukazatele na funkci z dynamické knihovny

Předchozí situaci se říká import funkcí. Všechny funkce nacházející se v dynamické knihovně však nemusí být tzv. viditelné. Kdyby nebyla funkce `CheckVersion` viditelná, příkaz pro získání pointeru na ni by vrátil hodnotu `NULL`.

Všechny exportované funkce jsou v kódu speciálně označené a ve výsledné knihovně viditelné (záleží ovšem na kompilátoru). Následující kód zajistí, že bude funkce viditelná:

```
extern "C" __declspec(dllexport) bool CheckVersion(int apiVersion);
```

Ukázka 2: Export funkce dynamické knihovny

Funkce je exportována, navíc doplněk `extern "C"` zajišťuje to, že se exportovaná funkce při překladu kompilátorem jazyka C++ bude jmenovat stejně jako v kódu, tedy `CheckVersion`. Bez tohoto doplňku by se název pravděpodobně neshodoval a obsahoval by navíc zakódované typy parametrů, typ návratové hodnoty a konvenci volání (thiscall, stdcall, atd.).

Tento přehled o využívání dynamických knihoven postačí pro porozumění textu dalších kapitol.

3 Nástroje podobného principu

Abychom si dokázali představit, jak by měl nástroj vypadat, porozhlédneme se po implementacích podobného principu.

Všechny mají společné to, že se zpracovává nějaká informace pomocí bloků s různou funkčností. U této akce musí být přítomny vstupní bloky, operátory (bloky zpracování) a výstupní bloky. Bloky jsou propojeny mezi svými piny pomocí čar nebo křivek.

U zobrazení bloků je dodržena konvence rozmístění vstupních pinů vlevo a výstupních pinů vpravo. Toto pravidlo neplatí např. u vývojových diagramů, kde se bloky navíc liší tvary.

Obrovskou výhodou těchto editorů je jejich přehlednost a nedestruktivní práce – jedná se o jednoduchou výměnu bloku bez nutnosti vytvářet vše od začátku. Stačí jen odpojit starý blok, vložit jiný a ten opět propojit, zdrojový obrázek se zaručeně nezmění. Zmíněná výhoda bude zajisté doprovázet i naši implementaci. To samé platí, i když napíšeme program pro zpracování – stačí vyměnit příkaz. Neplatí to však např. pro Adobe Photoshop, protože práce s vrstvami neumožňuje připojit filtry, které by neupravovaly zdrojová data.

3.1 DirectShow Filter Graph Editor

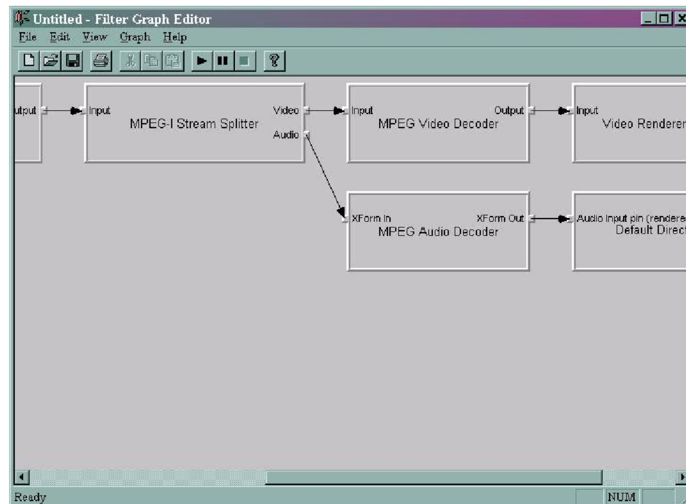
DirectShow je starší API pro operace s multimediálními daty. Operace s multimediálními daty lze nastavit pomocí kódu nebo jednodušeji pomocí Filter Graph editoru (GraphEdit [10]).

Operace s multimediálními daty začíná od multimediálního vstupu, kterým může být audio nebo video soubor, vstup z kamery, mikrofonu apod. Dále pokračují různé filtry, slučovací a rozdělovací bloky až po výstupní bloky.

Mezi klady a zápory této implementace patří:

- (+) Vstupy nalevo, výstupy napravo
- (+) Šipky zobrazující směr toku dat
- (+) Není potřeba dalších ovládacích prvků (dáno účelem využití)
- (−) Nejsou barevně odlišené piny

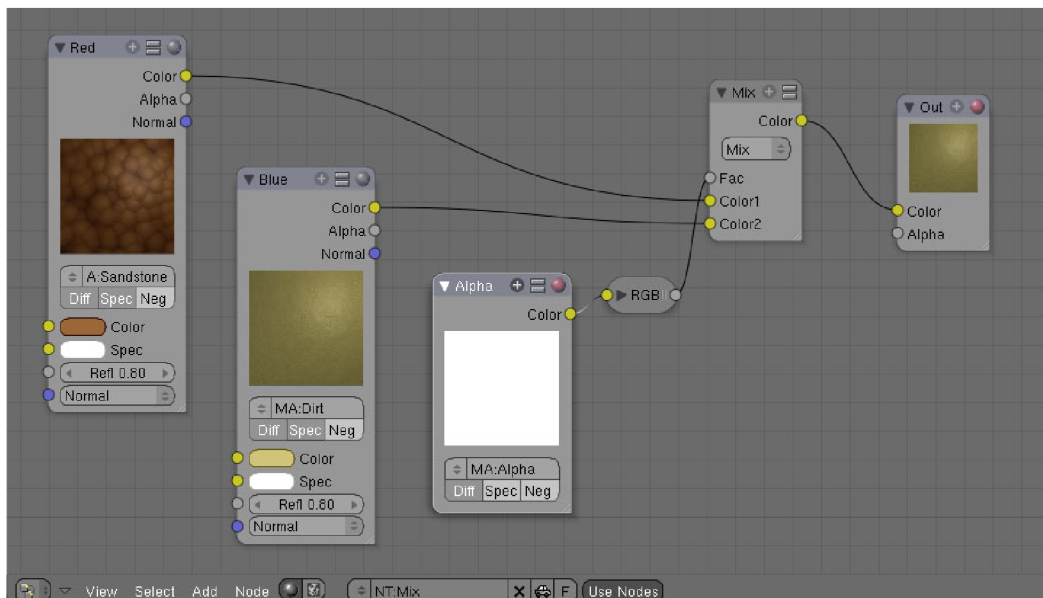
Tato aplikace je staršího data a po vydání DirectX 9 se DirectShow [8] stalo součástí Microsoft Platform SDK. Její nejpodobnější alternativou je GraphStudio [9] obohacenou o užitečné funkce, které v původním GraphEdit chybí.



Ilustrace 1: DirectShow Filter Graph Editor

3.2 Blender

Blender od verze 2.42 disponuje nástrojem Node editor [1], který slouží pro vytváření shaderů a jejich kompozicí. Díky tomuto nástroji lze v Blenderu jednoduše, přehledně a hlavně rychle vytvořit kvalitní grafický výsledek renderované scény bez jakékoliv znalosti jazyka shaderů. Chování Node editoru je uživatelsky nejpřívětivější (viz níže).



Ilustrace 2: Blender Node Editor

Node editor obsahuje i několik zajímavých rysů. Jedním z nich je blok, který okamžitě zobrazuje výsledek spojených bloků přímo na pracovní ploše. Dalším rysem je rozlišení vstupních a výstupních pinů pomocí pozice a barev, které rozlišují datový typ parametru. Jak bývá zvykem, jsou tyto barevně odlišené vstupní piny viditelně umístěny vlevo, barevně odlišené výstupní piny jsou umístěny vpravo. Počet pinů je pro každý blok jiný, to závisí na jeho funkci. Dalšími zajímavými nápady je obsah parametrů jako posuvníky, vstupní textová pole apod. v těle příslušného bloku. Poslední v řadě zajímavých rysů je možnost tělo bloku minimalizovat a šetřit tak místo na obrazovce, což je vhodné ve velké propletené pavučině bloků. Samozřejmostí je posouvání, přibližování a oddalování pohledu na pracovní plochu.

V literatuře [1] je popsána i implementace této novinky, včetně procesu zpracování. Node editor je obsažen i v programech Softimage, Lightwave, Houdini a nově i v Maya. Jedná se také o 3D modelovací nástroje a node editory jsou taktéž využity pro tvorbu shaderů a kompozic.

Shrneme si klady a zápory této implementace:

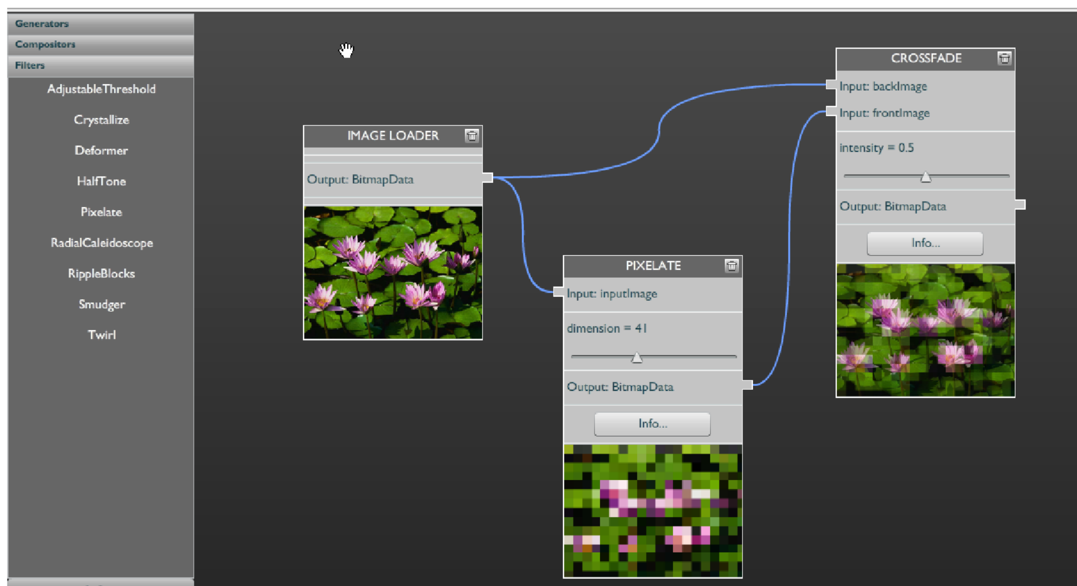
- ➕ Blok zobrazující výsledek přímo na pracovní ploše
- ➕ Rozlišení pinů barvami dle datového typu
- ➕ Parametry jako posuvníky, přepínače, atd.
- ➕ Minimalizace bloků
- ➕ Ovládací prvky v těle bloků
- ➖ Některé bloky jsou příliš velké

3.3 Pixel Bender Blender Prototype

Na internetu je dostupná zajímavá implementace NodeRendereru [2], která zpracovává pomocí funkčních bloků obraz a v každém zobrazeném bloku se zobrazuje i jeho výstup. Zmíněná vlastnost není příliš vhodná, jelikož zabírá hodně místa na editační ploše.

Velkým omezením je slabá editace propojů. Nelze upravovat již vytvořené propoje, nelze přidat další blok mezi dva propojené bloky bez odstranění jednoho z nich tak, aby se odstranil i propoj mezi nimi. Dále je nutné propojovat bloky od výstupu ke vstupu, nikoliv naopak. Na druhou stranu bloky obsahuji ovládací prvky, kterými se mění chování bloku.

Důležité je poznamenat, že tato aplikace nevytváří shadery či jejich kompozici, ale určitým způsobem zpracovává obraz, čehož chceme docílit i my.



Ilustrace 3: Pixel Bender Blender Prototype

Tato implementace má následující klady a zápory:

- ➕ Kategorizovaná paleta nástrojů
- ➕ Ovládací prvky v těle bloků
- ➖ Velké bloky vlivem náhledu v jeho těle
- ➖ Nemožnost odstraňovat nebo upravovat propoje
- ➖ Propoje lze vyvářet jen od výstupu ke vstupu
- ➖ Předdefinované vstupní obrázky
- ➖ Žádné výstupní bloky

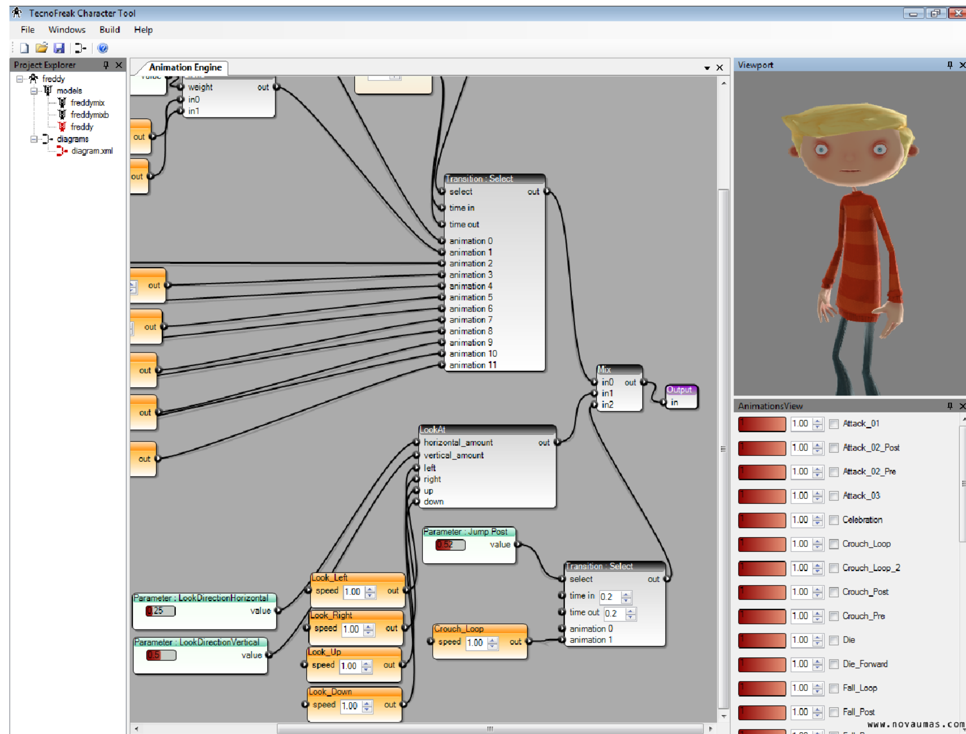
3.4 TecnoFreak Animation System

TecnoFreak Character Tool nástroj a TecnoFreak Animation System knihovna [3] jsou rozšířením pro kosterní animační systém grafického enginu Ogre3D. Propojují se zde různé animace do řídících bloků a jejich výstup se používá jako výsledná animace. Za běhu aplikace, ve které se provádí animace, se nastavují koeficienty těchto řídících bloků. Výsledkem je potom patřičná změna animace.

Typy jednotlivých bloků jsou v tomto nástroji rozlišeny různými barvami. Parametry jsou zelené, vstupní animace oranžové, řídící bloky černé a výstupní bloky fialové.

Klady a zápory tohoto nástroje jsou následující:

- ➕ Lze sdílet parametry bloku
- ➕ Barevně odlišené bloky podle kategorie
- ➖ Příliš mnoho nepřehledných propojů



Ilustrace 4: TecnoFreak Character Tool

4 Návrh řešení

Zde je popsán návrh řešení z několika pohledů: implementace, rozšiřování a ovládání.

Na úvod je třeba říci, že se aplikace bude skládat z několika částí. Nejlépe ji vystihuje následující diagram. Grafické uživatelské rozhraní využívá funkce jádra, které je středem aplikace, a pluginy jsou napojeny na jádro. Toto rozdělení přináší tu výhodu, že lze jádro použít i pro jiné nástroje.

Ilustrace 5: Diagram rozdělení aplikace na několik částí

4.1 Rozdělení uživatelského rozhraní a jádra aplikace

Již bylo řečeno, že aplikace bude rozdělena na uživatelské rozhraní a jádro. Jádro obsahuje veškeré potřebné funkce:

- Definice datových typů všech propojů mezi bloky a zapouzdření těchto datových typů
- Dekodéry realizující vstup pro vstupní bloky
- Enkodéry realizující výstup pro výstupní bloky
- Bloky realizující nějakou činnost, úpravu (dále jen operátory)
- Propojení bloků
- Provádění zpracování obrazu a zpětná vazba o provádění
- Zjištění dostupných bloků a jejich vlastností
- Načtení a uložení aplikace

Uživatelské rozhraní plní tyto funkce:

- Zobrazení bloků, jejich komponent a propojů
- Vkládání a přesouvání bloků
- Propojování
- Historie změn

Navíc aplikace bude rozšiřitelná pomocí dynamických knihoven o tyto možnosti:

- Nové datové typy (podpora pro nové bloky)
- Nové dekodéry realizující vstup pro vstupní bloky
- Nové enkodéry realizující výstup pro výstupní bloky
- Nové operátory

Důležitým požadavkem návrhu je využití v celém kódu techniky výjimek, které usnadní informování o chybách. S výjimkami souvisí i systém záznamu událostí („logování“).

4.2 Návrh uživatelského rozhraní editoru

Většina editorů se drží koncepce horního menu, horní nástrojové lišty, boční nástrojové lišty a editační plochy. Jsou i editory, kde je editační plocha rozdělena na více částí, lišty jsou přirozenou součástí editační plochy nebo je některá lišta přístupná pouze jako kontextové menu.

Koncept uživatelského rozhraní tohoto editoru je voleno s ohledem na praktické využití a urychlení práce pomocí kombinace klávesnice a myši. Grafické uživatelské rozhraní je rozděleno na čtyři základní části:

- Horní „souborové“ menu
- Horní „standardní“ nástrojová lišta
- Postranní plovoucí paleta s bloky pro vložení
- Editační (pracovní) plocha

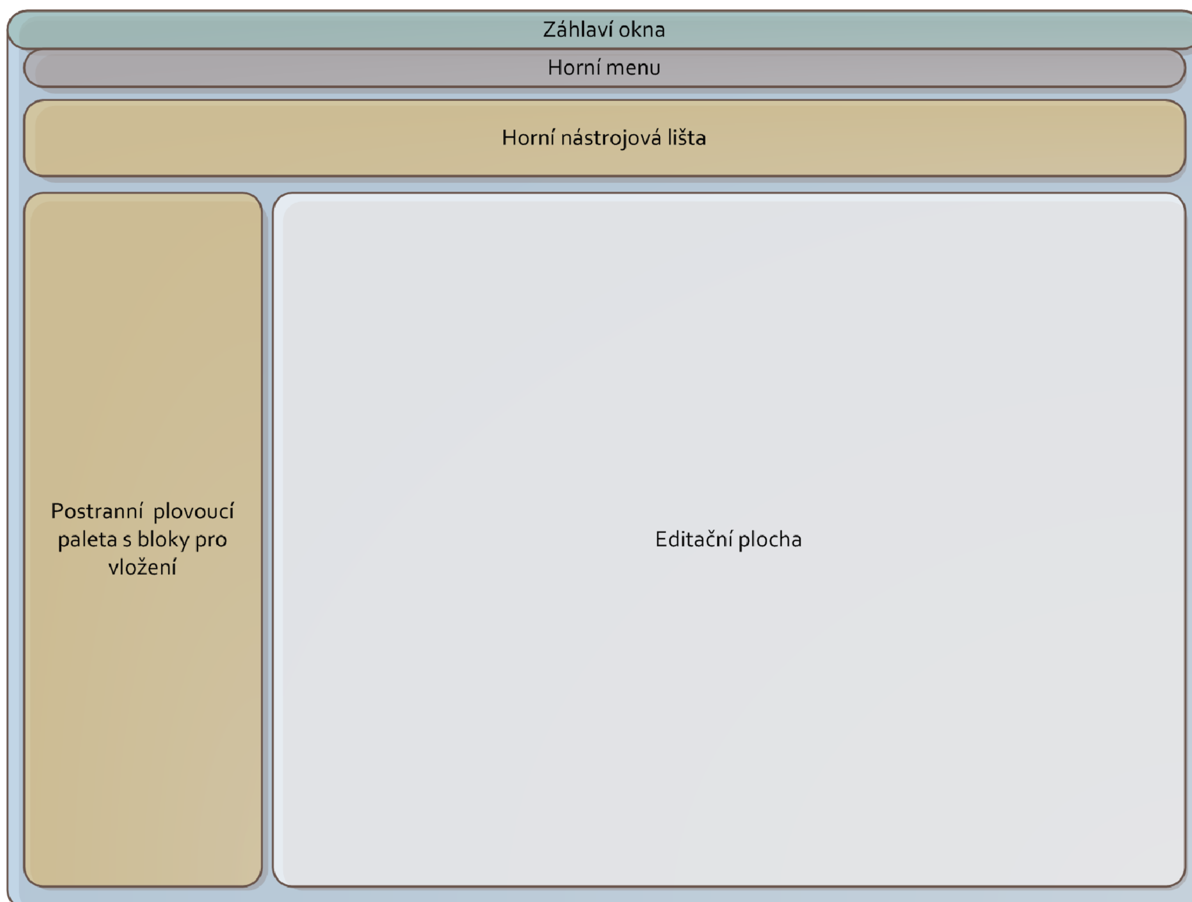
Horní nástrojová lišta je rozdělena do čtyř částí. Tato lišta je ořezanou kopií horního menu, které má více položek.

- Operace se soubory
- Akce zpět/vpřed
- Operace se schránkou
- Spouštění navrhované aplikace

Postranní plovoucí paleta obsahuje kategorizovaný kontejner s bloky, které se vkládají na pracovní plochu. Paleta bude obsahovat veškeré dostupné typy bloků.

Pracovní plocha by měla podporovat návrh aplikace pro zpracování obrazu. Jak je v kapitole 3 naznačen směr práce s editorem, měl by se využívat podobný princip. Na pracovní plochu se budou vkládat bloky z postranní palety a propojovat mezi sebou. Bloky půjde libovolně posunovat, tato vlastnost se týká i pracovní plochy, taktéž bude posuvná, což je velice přínosné.

Zobrazení bloků by se mělo držet zavedené konvence. Vstupní piny budou vlevo a výstupní vpravo. Zajímavé bude jistě i barevné rozlišení pinů dle datových typů. Nastavení parametrů bloku by mělo být možné aktivními ovládacími prvky přímo v těle bloku, přičemž bude možné minimalizovat zobrazení těchto prvků. Tím se tělo bloku zmenší, zabere méně místa a zpřehlední zobrazení.



Ilustrace 6: Návrh uživatelského rozhraní editoru

Dalším důležitým požadavkem návrhu je udržet oddělené uživatelské rozhraní a jádro aplikace. Uživatelské rozhraní bude využívat funkce jádra a nebude je žádným způsobem obcházet.

Podpůrnou funkcí uživatelského rozhraní bude vyhledávání bloků podle názvu či typu, vkládání komentářů na pracovní plochu, ladění zpracování doplněné o „watch window“, hierarchické sdružení bloků a historie editace.

Ladění zpracování bude probíhat krokováním zpracování. Přídavné okno se zobrazením aktuálních hodnot výstupu určitého propojení bude umět zobrazit i hodnotu typu obrázek, s možností přiblížení, s nástrojem kapátko apod.

4.3 Rozšiřování pomocí dynamických knihoven

Funkčnost aplikace bude možné rozšiřovat pomocí dynamických knihoven. Rozšíření se týká především nových dekodérů vstupu (jen pro načtení dat), nových enkodérů výstupu (jen pro uložení dat) a nových operátorů (případně i datové typy, viz níže). Tato jednotlivá rozšíření nazýváme pluginy.

Jedna dynamická knihovna smí obsahovat více pluginů a zároveň bude mít možnost uvnitř použít i jiné výpočetní síly než CPU, mohou s výhodou využít OpenCL, CUDA, aj.

Operátory mohou potřebovat i jiné datové typy, než které aplikace standardně nabízí, proto budou mít dynamické knihovny možnost rozšířit funkčnost o nové datové typy.

Zrekapitulujeme si, co vše půjde přidávat:

- Dekodéry vstupu
- Enkodéry výstupu
- Datové typy
- Operátory

4.4 Přehled bloků

Bloky se rozdělují na tři typy. Prvním jsou vstupní bloky. Ty využívají dekodéry pro získání dat. Vstupem může být soubor obrázku, soubor videa, adresář s obrázky nebo i výstup z kamery. Jednotlivé druhy bloků (obrázek, video, výstup z kamery) budou v aplikaci na pracovní ploše vypadat stejně, bez ohledu na to, co je vstupem. Co bude vstupem, se určuje použitým blokem. U vstupu z kamery lze dodatečnými parametry nastavit počet snímků za sekundu, rozlišení apod.

Druhým typem jsou výstupní bloky, které využívají enkodéry pro ukládání dat do souborů. Výstupem může být soubor obrázku či videa. Taktéž tyto jednotlivé druhy bloků budou na pracovní ploše vypadat stejně, bez ohledu na to, co je výstupem.

Posledním typem jsou operátory, které určitým způsobem upravují vstup a výsledek úpravy posílají na svůj výstup. Každý druh bloku operátoru bude vypadat jinak, podle toho, co bude mít blok za vstupy a výstupy.

Základní implementované operátory budou:

- Blending dvou obrázků
- Změna velikosti obrázku
- Konvoluce
- Sobelův operátor
- Medián

- Převod na odstíny šedi, prahování
- Dithering
- Dilatace a eroze
- FFT
- DCT

Zajímavé mohou být i další pokročilé operátory. Prvním v řadě je speciální blok se skriptem. Tento operátor provede úpravy vstupních dat podle uživatelem zadaného skriptu a výsledek předá na výstup. Jedná se o nejrychlejší cestu k rozšíření funkčnosti aplikace. Dalšími operátory mohou být detektory pohybu, detektory barvy kůže apod.

4.5 Nastavení a propojení funkčních bloků

Tato podkapitola se týká GUI, resp. editace na pracovní ploše editoru. Již bylo zmíněno, že tělo bloku zobrazené na pracovní ploše bude obsahovat vstupy, výstupy a parametry, přičemž parametry bude možné schovat. Parametry mohou být celočíselné hodnoty, čísla s plovoucí řadovou čárkou, řetězce, barvy a výčet hodnot.

Číselné hodnoty mohou být vymezeny do různých intervalů. Číselný parametr se v těle bloku zobrazí jako textové pole s šipkami. Parametr barva bude zobrazen jako políčko vybarvené nastavenou barvou, kliknutím na toto políčko se zobrazí dialog pro výběr barvy. Výčet hodnot bude zobrazen jako prvek combo box. Pomocí těchto prvků půjdou měnit stav a hodnota parametrů. Dalším druhem parametru může být matice. Pro její zadání bude k dispozici jednoduchý editor. Textové řetězce se budou upravovat v textovém poli, delší řetězce v malém editoru.

Má-li být hodnota parametru určena výstupem předchozího bloku, podobně jako v nástroji Blender, je možné ruční zadání parametru obejít a propojit tento parametr s výstupem.

Propojení vstupů a výstupů podléhá několika omezením. Nejdůležitějším z nich je, že každý propoj musí propojovat jeden vstup a výstup. Tím se zpřehlední navrhovaná aplikace a zjednoduší se tak i implementace.

Toto omezení implikuje následující, omezující je i počet propojů připojených k jednomu vstupu. Počet musí být roven alespoň jedné.

4.6 Historie akcí

Dnešní kancelářské i jiné programy bychom si asi těžko představili bez historie akcí. Jedná se o seznam akcí, které uživatel provedl a program mu umožňuje v tomto seznamu se navracet k dřívějším změnám.

Uživatelské rozhraní by mělo disponovat i tímto rysem. Jelikož se nejedná o žádný editor, jehož uložený dokument by zabíral více než desítky kilobajtů, můžeme si dovolit historii akcí implementovat jako ukládání celého dokumentu do jedné položky seznamu. Nemusí se tak implementovat složitější zaznamenávání atomických změn a ty využít při akci zpět či vpřed pro úpravu stávajícího stavu dokumentu.

V tomto seznamu se bude posouvat zpět a vpřed (vpřed za předpokladu, že byla vrácena historie a nebyla provedena žádná změna), čímž se načte celý dokument ze seznamu historie.

Aby historie mohla být zaznamenávána, musí tuto funkci podporovat i ovládací prvek, který je pro editaci použit. Nejvhodnější by bylo využití události, která bude vyvolána pokaždé, když se provede právoplatná změna.

4.7 Interpretace navržené aplikace

Aby byla aplikace proveditelná, musí být sestaven plán, což je seznam kroků, které jsou nezbytné pro provedení alespoň jednoho cyklu běhu navržené aplikace. Plánování je oblast umělé inteligence, detailnější informace jsou uvedeny v [13].

Při sestavování plánu se prohledává plánový prostor, v našem případě se jedná o navrženou aplikaci – bloky a propoje. Plán může být sestaven pomocí dvou metod, progresivní metodou (od vstupů k výstupům) nebo regresivní metodou (od výstupů ke vstupům).

Obě metody využívají několik seznamů pro vytvoření plánu. Jedná se o seznam akcí výsledného plánu, předpoklad a nevyřešené akce, ve kterém jsou všechny nevyřešené bloky a propoje. Na počátku jsou předpokladem vstupní bloky. Jelikož pro ně neexistují nevyřešené akce, jsou ihned vloženy do seznamu akcí. Dále se postupuje tak, že se vybere blok z nevyřešených akcí, a když pro něj neexistují nevyřešené akce, vloží se do seznamu akcí, jinak se provedou kroky pro dosažení těchto nevyřešených akcí nebo se blok přeskočí. Tyto kroky se dějí, dokud není seznam nevyřešených akcí zcela prázdný.

Ve své podstatě, vytvoření plánu odhaluje zpětnovazební smyčky, které by se neměly vyskytovat. Je to dáno tím, že když se při plánování narazí na propoj vedoucí na vstup (či výstup v případě regresivního plánování) bloku, který je již vložen do plánu, nelze najít nový cíl plánu.

Dalším kritériem plánování je, aby se do plánu nedostaly neadekvátní akce, které nevedou k dosažení cíle. U regresivního plánování je obecně známo, že neadekvátní akce neplánuje. Plánování akcí pro aplikaci může obsahovat několik odpojených bloků, slepé cesty apod., a pokud se bude implementovat druhá metoda, musí se tato fakta určitým způsobem pohlídat, jinak mohou přerušit plán akcí výjimkou, že na vstupu odpojeného bloku chybí data.

4.8 Datový formát

Navržená aplikace musí být ukládána do souboru. Podle navržených funkcionalit projektu je nejlepší volbou využít formát XML [14], který umožňuje rozšiřovat entity o atributy a další entity.

Tento XML soubor by se měl skládat ze čtyř částí:

- vstupní bloky
- výstupní bloky
- operační bloky
- propoje

Každá tato část bude obsahovat entity příslušného druhu. Jádro si bude všímat atributů, které jsou základem dokumentu, editor tyto entity bude rozšiřovat o specifické atributy a další entity (např. pro uložení pozice bloku na ploše). Tuto vlastnost rozdělení XML schématu musí jádro podporovat. Jelikož jsou bloky rozděleny na tři druhy, ale i nejen pro to, všechny by měli mít své unikátní ID pro jejich snadnou identifikaci.

4.9 Nástroj pro spuštění navržené aplikace

Součástí projektu by měl být nástroj, který již navrženou aplikaci spustí. Mělo by se jednat o jednoduchý interpret pro navržené aplikace.

Jelikož jádro musí obsahovat funkce pro načtení navržené aplikace a její interpretaci, tento nástroj s výhodou využije jádro.

Důležité pro znovupoužitelnost spouštěné aplikace je nahrazení předdefinovaných vstupů. V editoru se totiž vstupy nastaví např. na specifický adresář, ale pro spuštění je důležité nastavit vstup na jiné adresáře. Nesmyslem je spustit editační nástroj, otevřít aplikaci a nastavit vstup na jiný adresář, je to příliš neefektivní způsob. Proto tento nástroj musí disponovat funkcí pro nahrazení vstupu.

4.10 Nástroj pro validaci aplikace

Zajímavým pomocným nástrojem by mohla být validace aplikace, tedy kontrola, zda je aplikace proveditelná. Není proveditelná v tom případě, že obsahuje především zpětnovazební smyčky a že k jednomu vstupnímu pinu vede více propojů.

První kontrolou je kontrola vstupních pinů a počet propojů, které k tomuto pinu vedou. Není možné, aby si pin vybíral, odkud bude data brát. Toto omezení je popsané v podkapitole 4.5.

V podkapitole 4.5 je také zmínka o omezení, že každý propoj propojuje jeden vstup a jeden výstup. Tato skutečnost musí být také zkontrolována.

Poslední kontrolou je kontrola smyček, ta je provedena pouze pomocí vytvoření alespoň částečného plánu (procházení plánového prostoru).

V případě, že funkce validace nalezla některé chyby, informuje uživatele o tom, které propoje nejsou v pořádku. Tato funkce by měla být umístěna v jádře a aplikace by ji měla využívat. Je pravděpodobné, že výše zmíněnou funkci bude využívat nástroj pro tvorbu aplikací při editaci (ověřování propojů ihned při jejich tvorbě).

Nástroj může být doplněn o přepínač, který veškeré nalezené chyby opraví odstraněním chybných propojů. I když poté bude aplikace validní, nemusí být kompletní a její spuštění nebude z tohoto důvodu možné.

5 Implementace

Implementace se drží návrhu řešení. Nástroje jsou implementovány v jazyce C++, s využitím knihovny OpenCV [11] pro zpracování obrazu. K realizaci uživatelského rozhraní a dalších funkcí je využita knihovna Qt [12].

Nástroje se skládají ze dvou částí: z jádra a uživatelského rozhraní. Návrh jádra byl zmíněn v předchozí kapitole, nyní návrh obohatíme o implementační detaily. Sled podkapitol přibližným způsobem kopíruje i plán práce na jednotlivých částích projektu.

5.1 OpenCV

OpenCV [11] je knihovna zabývající se počítačovém vidění původně vyvíjená společností Intel. K dispozici je pod licencí BSD. Knihovna je orientovaná především na zpracování obrazu v reálném čase.

OpenCV obsahuje funkce pro načtení a ukládání do často používaných obrazových formátů, dokáže získat snímky videa ze souboru s videem, získat snímky z připojené web-kamery i vytvořit video-soubor ze snímků. K tomu využívá DirectShow a FFMPEG knihovnu.

Interface knihovny verze 2.0 je napsán v jazycích C/C++. Ač je knihovna dobře dokumentovaná, někdy je problém se dorozumět s nějakou funkcí vlivem absence složitějších příkladů. Dle mého názoru je pro tento projekt vhodná. Je jednoduchá k použití, dobře optimalizovaná a na fakultě se hodně využívá.

5.2 Qt framework

Qt [12] je velice obsáhlý framework pro vytvoření aplikací především s grafickým uživatelským rozhraním. Norská firma TrollTech donedávna držela vývoj tohoto frameworku od roku 1992 ve svých rukou, nyní tuto firmu vlastní společnost Nokia a stále jej vyvíjí. Framework podporuje mnoho mobilních/embedded platforem a PC platformy, především MS Windows, Unix/Linux a operační systém MacOSX firmy Apple. Je k dispozici jak komerčně, tak i pod licencí LGPL. Framework má velice obsáhlou a kvalitní dokumentaci plnou krátkých příkladů.

Framework je psán v jazyce C++ a nejvíce aplikací využívající Qt taktéž. Qt se skládá z knihoven pro práci s GUI, SQL, OpenGL, síťovou komunikací, skripty, XML a další. Nejdůležitější je Core knihovna, která obsahuje mnoho tříd od šablon kontejnerů, přes streamy, po vícevláknovou podporu. Vše je dobře uspořádané a svým způsobem to připomíná .NET framework. Jen náhradou delegátů jsou komunikace mezi signály a sloty.

Signály a sloty řeší události, které jsou takto kompatibilní napříč všemi podporovanými operačními systémy. Tzv. MOC kompilátor (Meta-object compiler) před kompilací celého projektu projde všechny definice tříd a vygeneruje meta-objekt pro třídu označenou jako Q_OBJECT. Meta-objekt obsahuje signály a ukazatele na sloty, které se poté propojují a používají jako akce (signály) a reakce (sloty).

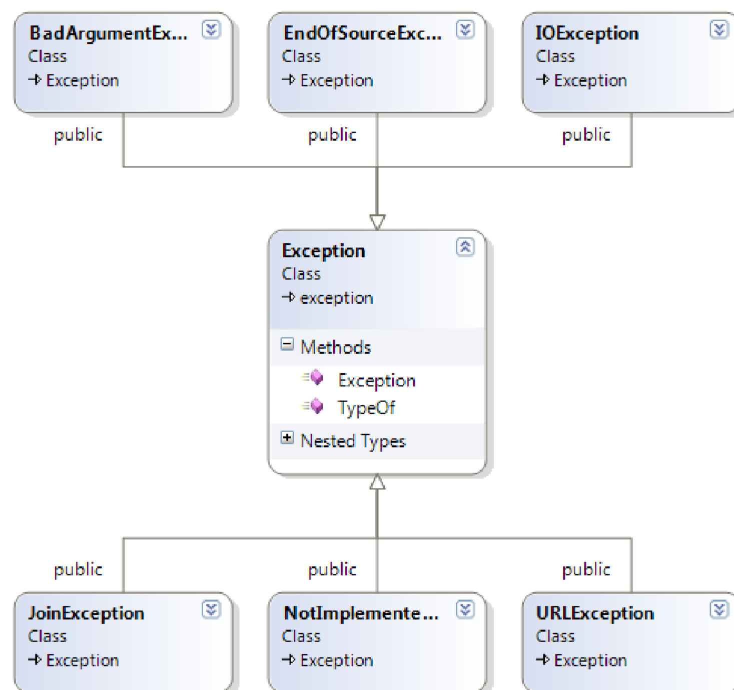
Pro návrh GUI je k dispozici praktický nástroj Qt Designer, ve kterém se po určitém počtu pokusů a omylů velice snadno pracuje, i když tam schází automatické vyplnění prázdného kontejneru určitým widgetem (Qt framework označuje prvky uživatelského rozhraní termínem widget) přes celou jeho plochu.

I přes malá nedorozumění s Qt Designerem je Qt framework velice vhodný k použití v projektu.

5.3 Výjimky

Důležitým požadavkem z návrhu je využití techniky výjimek. Ty se uplatní zejména při návrhu a běhu aplikace pro zpracování obrazu.

Výjimkami za běhu mohou být nepropojené vstupy, vyčerpání dat ze vstupů (netypická výjimka, spíše slouží k informaci a ukončení zpracování).



Ilustrace 7: Hierarchie tříd výjimek

V návrhu může jít o výjimky vyvolané při propojování dvou bloků (příklad):

- Nelze propojit vstup a výstup různých datových typů.
- Nelze propojit dva vstupy nebo dva výstupy.
- Nelze připojit více výstupů k jednomu vstupu.
- Nelze vytvořit rychlou smyčku propojením několika bloků.

5.4 Záznam událostí

Dalším požadavkem návrhu je záznam událostí. Jedná se o výpis zpráv různého charakteru do jednoho místa, tzv. logu. Zprávy jsou rozděleny na tři druhy:

- Chyby
- Obecné zprávy
- Varování

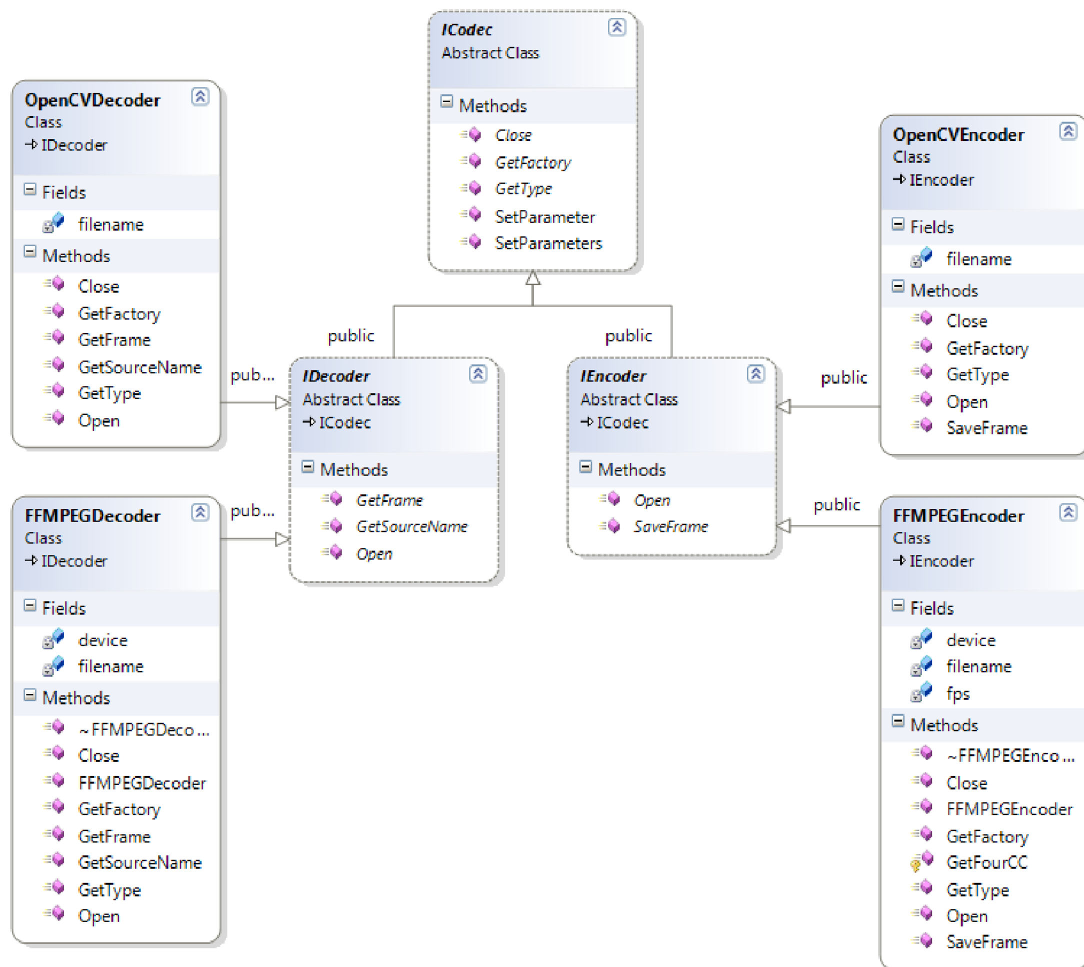
K záznamu událostí slouží tři globální objekty získané funkcemi `Errors`, `Messages` a `Warrnings` z jmenného prostoru `Diamage::Logs`, mají textový parametr identifikující modul (kategorii). Funkce vrací objekt typu `QDebug`, se kterým se komunikuje pomocí operátoru bitového posunu vlevo („<<"), tak jak je to zvykem u tříd typu `std::stream`.

5.5 Část jádra s podporou pluginů

Jádro aplikace se stará o spoustu věcí. Snad jednou z nejdůležitějších je rozšiřování funkčnosti pomocí pluginů, také definuje čtyři druhy pluginů pomocí abstraktních tříd `ICodecFactory`, `IOperatorFactory` (zahrnuje další třídy) a `IType`.

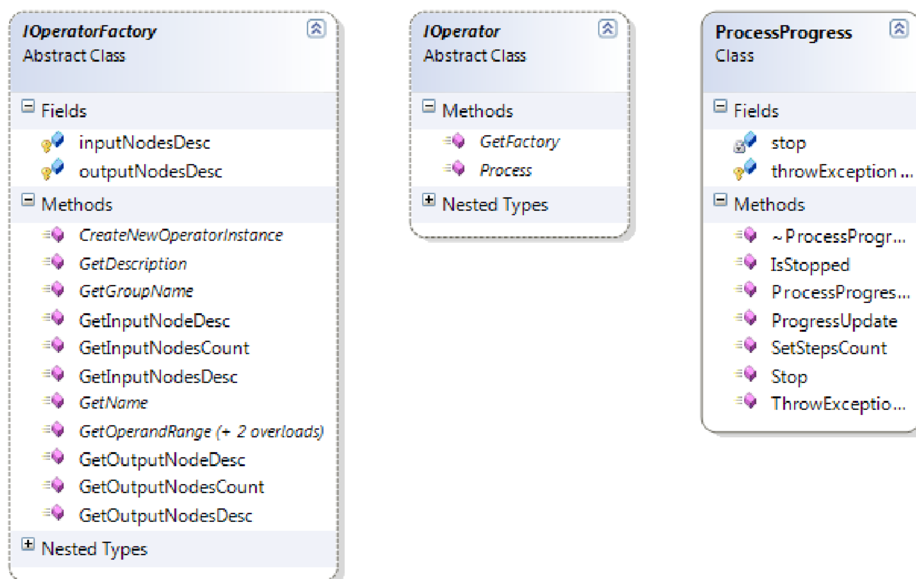
`IDecoder` a `IEncoder` mají společného předka `ICodec`. Zaměřují se na získání dat ze zdroje či uložení výsledku. Obsahují metody pro otevření a uzavření, nastavení pokročilých parametrů a získání či uložení snímku. Pokročilé parametry mohou být využity pro nastavení rozlišení snímku z kamery, počet snímků za sekundu, kodek použitý pro uložení videa apod. V původním návrhu není obsažena abstraktní třída `ICodecFactory`, která slouží pro vytváření instancí tříd `IDecoder` a `IEncoder` jádrem.

Jádro obsahuje již specializované třídy pro uložení a načtení obrazových formátů standardně přístupných v OpenCV knihovně (`OpenCVDecoder`, `OpenCVEncoder`), formátů videa (`FFMPEGDecoder`, `FFMPEGEncoder`) a získávání obrázků ze vstupního zařízení (`DeviceInput`).



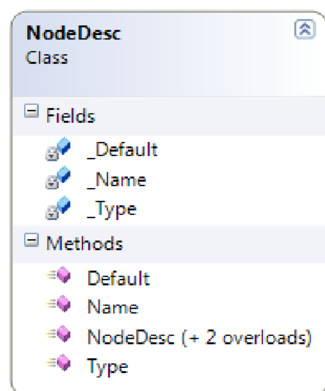
Ilustrace 8: Hierarchie tříd kodeků

`IOperatorFactory` je abstraktní třída pro vytvoření instance třídy `IOperator` (operátor), která zpracovává obraz. `IOperatorFactory` obsahuje metody pro získání názvu a popisu operátoru, popisu jeho vstupů a výstupů. Abstraktní třída `IOperator` obsahuje metodu `Process`, která určitým způsobem zpracuje obraz (např. provede konvoluci, medián). Metoda může vyvolat výjimku, jejímž zdrojem může být špatný parametr či jiné podměty. V takovém případě se celé zpracování přeruší. S metodou lze komunikovat pomocí instance třídy `ProcessProgress`, která informuje o stavu operace. Tato třída je navržena pro komunikaci uživatelského rozhraní s procesem zpracování.



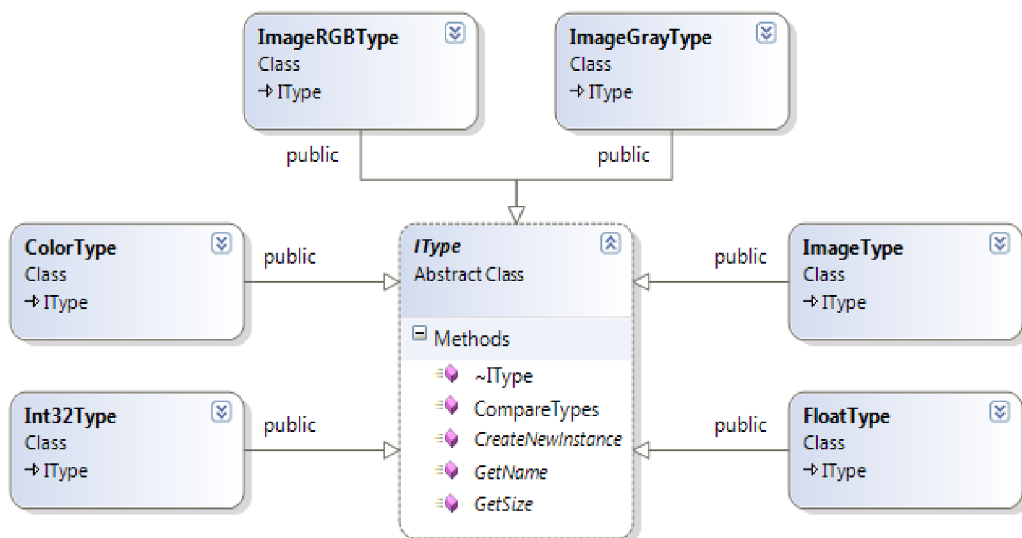
Ilustrace 9: Třídy IOperatorFactory, IOperator, IProcessProgress

Třída `IOperatorFactory` pro popis vstupů a výstupů využívá třídu `NodeDesc`. Třída uchovává název a datový typ operandu, případně jeho výchozí hodnotu. Proto jde třída inicializovat dvěma způsoby: dvojicí název a datový typ nebo dvojicí název a výchozí hodnota.



Ilustrace 10: Třída NodeDesc

Abstraktní třída `IType` je využívána pro definici datových typů vstupů a výstupů operátorů. Jádro definuje základní typy, jako jsou číselné typy, datový typ řetězec, barva a obrázek. Třída `IType` je zavedena ze dvou důvodů. První a nejdůležitější je sémantická kontrola propojů a druhá má původ v pluginech pro zpracování obrazu (`IOperatorFactory`, `IOperator`), které si takto mohou definovat své vlastní datové typy (např. přidání datového typu „region of interest“ či jiné).

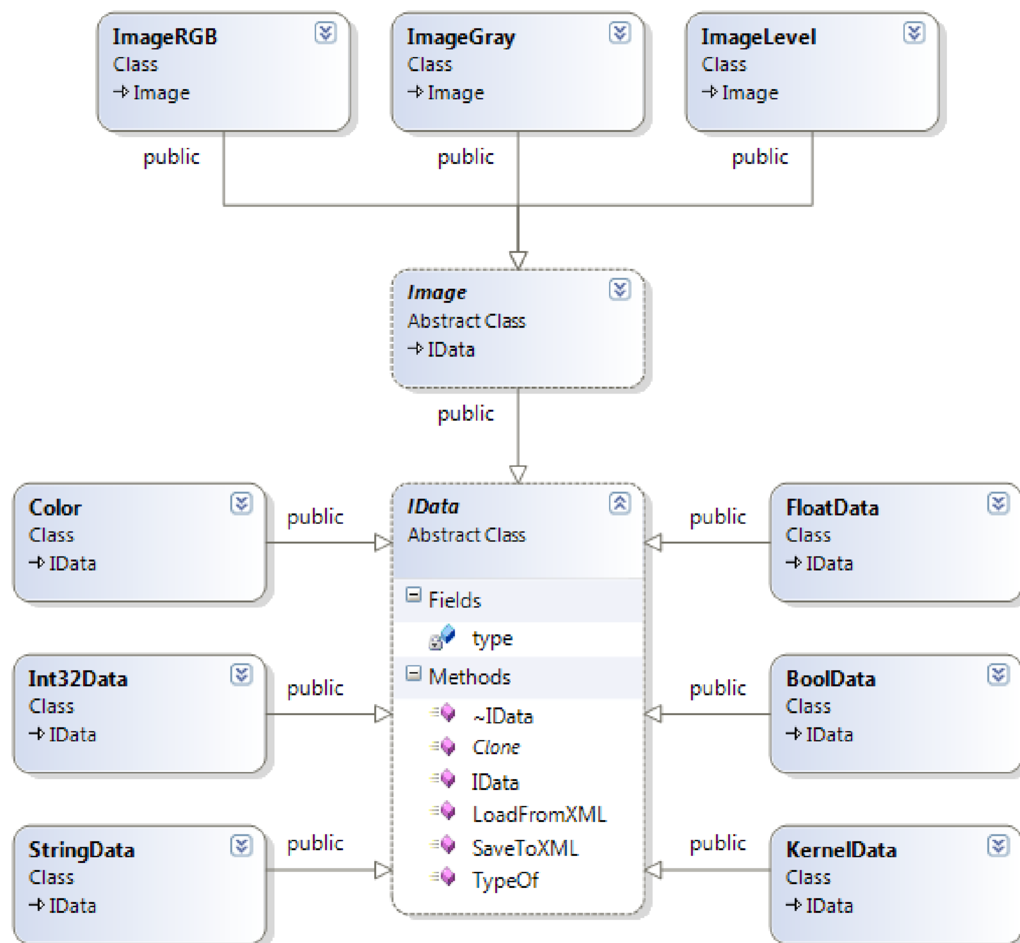


Ilustrace 11: Hierarchie tříd IType (jen některé)

`IType` má metodu pro vytvoření nové instance tohoto datového typu, jedná se tedy také o továrnu. Tyto instance jsou třídy s předkem `IData`, která nese data a informaci o použitém typu. Využívá se zejména pro předávání dat mezi bloky. K této třídě patří i třída `IDataPtr`, která vykonává funkci tzv. shared pointeru a stará se o bezstarostné uvolnění instance třídy `IData`. Postarání se o instanci tímto způsobem zamezuje únikům paměti či vícenásobnému uvolnění téže paměti.

Třídy `Int32Data`, `FloatData` a další číselné reprezentace, které jsou odvozeny od třídy `IData`, neslouží pro ukládání pouze jednoho čísla, ale obsahují dynamický vektor těchto čísel. Tuto vlastnost lze využít např. u operátorů, kde má být vstupem vektor čísel (např. velikost filtru v pixelech).

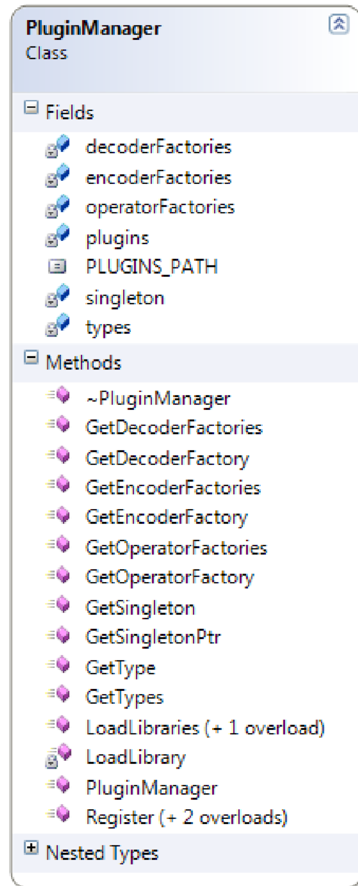
Podpůrnými třídami jádra jsou `Image` a `Color`, které vyjadřují svou podstatu názvy. Abstraktní třída `Image` je předkem pro třídy `ImageRGB, ImageGrey` a `ImageLevel`. Vstupní piny bloků, u kterých nezáleží, zda bude vstupní obrázek barevný nebo v odstínech šedi, jsou určeny jako datový typ `Image`, což je velice důležité pro definici vstupů a výstupů operátoru. Pokud vstupní piny bloku budou určeny jako datový typ `ImageRGB`, obrázek v odstínech šedi nepůjde připojit (to samé opačně pro `ImageGrey`). `ImageLevel` je speciální případ, nelze pro něj použít datový typ `Image`, aby nešel zaměnit s ostatními formáty. `ImageLevel` je obrázek s různým počtem úrovní hodnot (binární a více) a získá se z ostatních pomocí převodních bloků. Vnitřní implementace třídy `Image` využívá OpenCV reprezentaci dat.



Ilustrace 12: Hierarchie tříd IData (jen některé)

Druhou velice důležitou funkcí jádra je management pluginů. K tomuto účelu slouží singleton třídy `PluginManager`, ta vyhledá veškeré pluginy (metody `LoadLibraries` a `LoadLibrary`), inicializuje je, zaregistruje je (metoda `Register`) a poskytuje k nim přístup přes jejich názvy (metody `GetXFactory`, `GetXFactories`, kde `X` jsou `Decoder`, `Encoder`, `Operator` a `Type`, viz ilustrace 13). Při uvolnění instance této třídy se veškeré inicializované pluginy uvolní.

`PluginManager` využívá Qt framework pro vyhledání dynamických knihoven s pluginy v souborovém systému a jejich zavedení do programu.



Ilustrace 13: Třída PluginManager

5.6 Zabudované kodeky

Jak již bylo výše uvedeno, jádro obsahuje specializované třídy pro uložení a načtení obrazových formátů (`OpenCVDecoder`, `OpenCVEncoder`), formátů videa (`FFMPEGDecoder`, `FFMPEGEncoder`) a získávání obrázků ze vstupního zařízení (`DeviceInput`). Hierarchii těchto tříd můžete vidět na ilustraci 8: Hierarchie tříd kodeků.

5.6.1 Kodeky obrazových formátů

Implementace tříd `OpenCVDecoder` a `OpenCVEncoder` sídlí společně v souboru `ImageCodecs.cpp`, kde je ve vektoru uvedeno, jaké přípony souboru podporují právě tyto třídy. Také je tam umístěna implementace tříd `OpenCVDecoderFactory` a `OpenCVEncoderFactory`, které jsou důležité pro enumeraci a vytvoření instancí tříd `OpenCVDecoder` a `OpenCVEncoder`. Tyto továrny jsou inicializovány v konstruktoru třídy `PluginManager`.

Životní cyklus dekodéru je znázorněn v následujícím kódu:

```
Decoder *decoder = PluginManager::GetSingleton()->GetDecoder("jpg");
decoder->Open(filename);
Image *image = decoder->GetFrame();
decoder->Close();
delete decoder;
```

Ukázka 3: Životní cyklus dekodéru

Pro nový dekodér je důležité přepsat metody `Open`, `GetFrame` a `Close`. V metodě `Open` této specializované třídy je pouze uloženo jméno souboru, v metodě `GetFrame` se pomocí funkce `cvLoadImage` načte obrázek do datové struktury, a v metodě `Close` se neděje vůbec nic. Je to z toho důvodu, že si zmíněná funkce sama otevře soubor, vrací ihned data a neponechává soubor nadále otevřený.

Podobný životní cyklus má i enkodér, pouze místo metody `GetFrame` má metodu `SaveFrame`. Metoda `SaveImage` využívá funkci `cvSaveImage`, která vytvoří nebo otevře soubor, zapíše do něj data a soubor zavře – metody se tedy chovají obdobně jako výše popsané metody třídy `OpenCVDecoder`.

5.6.2 Kodeky video formátů

Třídy kodeků video formátů, `FFMPEGDecoder` a `FFMPEGEncoder` jsou umístěny v jediném souboru `VideoCodecs.cpp`. Také jsou zde umístěny podporované přípony souborů a továrny pro tyto třídy. Metody `Open`, `GetFrame`, `SaveFrame` a `Close` pracují odlišně, než jak to bylo u obrazových formátů.

V dekodéru metoda Open skutečně otevře soubor pomocí funkce `cvCaptureFromAVI`, `GetFrame` volá funkci `cvQueryFrame` a metoda `Close` to celé dokončí pomocí funkce `cvReleaseCapture`.

Enkodér v metodě `Open` si jen poznamená cestu k výstupnímu souboru z toho důvodu, že nezná rozlišení snímků, které se budou vkládat. Proto s otevřením čeká na první snímek v metodě `SaveFrame`, otevření se realizuje pomocí funkce `cvCreateVideoWriter`. Všechny snímky se ukládají pomocí funkce `cvWriteFrame` a metoda `Close` vše uzavře pomocí funkce `cvReleaseVideoWriter`.

5.6.3 Enkodér pro web-kameru

Jedná se o speciální případ enkodéru, který získává snímky z web-kamery. Také má vlastní továrnu, ale není zaregistrována v instanci třídy `PluginManagera`, takže se k ní musí přistupovat přes její vlastní singleton.

Metoda `Open` inicializuje přístup k datům pomocí funkce `cvCaptureFromCAM`, metoda `GetFrame` získá snímek s využitím funkcí `cvGrabFrame` a `cvRetrieveFrame`. Metoda `Close` vše uvolní pomocí metody `cvReleaseCapture`.

Při využití více web-kamer je situace složitější, v OpenCV je doporučeno, aby před voláním `cvRetrieveFrame` bylo zavoláno napřed `cvGrabFrame` pro všechny použité web-kamery. To je ošetřeno použitím statického vektoru inicializovaných kamer a určité režie v metodách `Open`, `GetFrame` a `Close`.

5.7 Část jádra pro realizaci programu pro zpracování obrazu

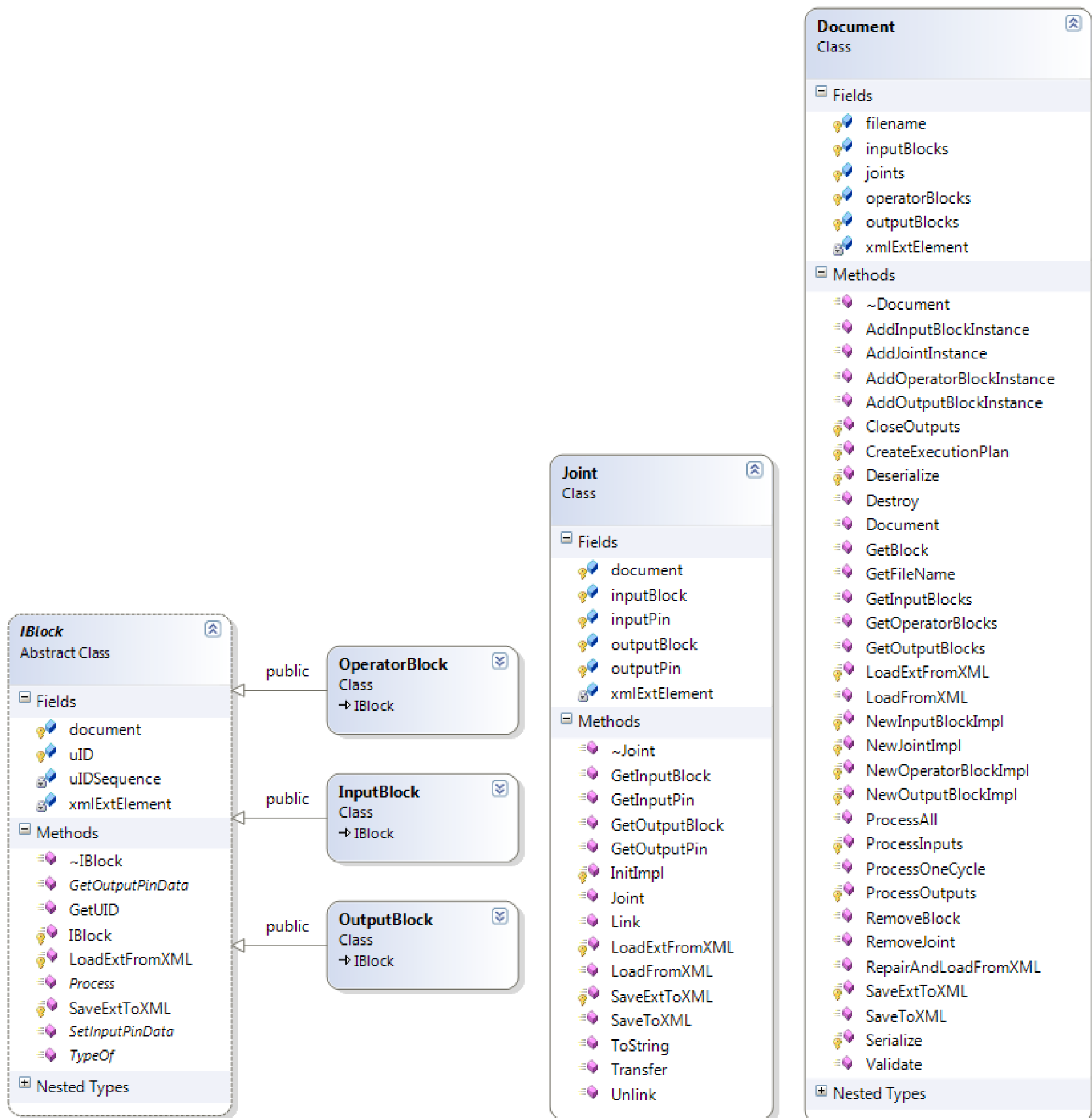
V další fázi projektu je aplikace obohacena i o management propojení. Jádro bude obsahovat třídu `Document` s kolekcemi instancí tříd `InputBlock`, `OutputBlock`, `OperatorBlock` a `Joint`. Třída `Document` reprezentuje navrženou aplikaci pro zpracování obrazu.

`Joint` je třída sloužící k propojení bloků. Metoda této třídy k propojení kontroluje sémantiku obou konců, v případě nezdaru vyvolá výjimku. Další metoda této třídy slouží k přenosu dat z jednoho konce na druhý, přenáší se specializovaná instance již zmíněné třídy `IData`. Metoda získá data z výstupního bloku a zkopíruje je na vstupní blok.

`DecoderBlock`, `EncoderBlock` a `OperatorBlock` jsou třídy pro zaobalení vstupu, výstupu a operátoru. Mají společného předka `IBlock`. Potomci této třídy přepisují metodu `Process`, která provede zpracování dat.

Třída `DecoderBlock` má metodu `Init`, které se předává jako parametr URI ke vstupu. Toto URI rozlišuje soubor obrázku (lze zadat více souborů oddělených středníkem), soubor videa, adresář s obrázky a vstupní zařízení (web-kamera).

Třída `OperatorBlock` má odlišnou metodu `Init`, které se předává továrna operátoru. Metoda `Init` si takto vytvoří svou vlastní instanci operátoru a pomocí továrny vytvoří vlastní vektory vstupů a výstupů a inicializuje je na výchozí hodnoty.



Ilustrace 14: Hierarchie tříd IBlock, třídy Joint a Document

Třída `Document` obsahuje metodu pro provedení činnosti navržené aplikace. Z počátku ověří závislosti a vytvoří tak plán postupu od vstupních bloků směrem k výstupním, který se následně vykoná. Vytvoření plánu je detailně popsáno v podkapitole 5.8 Vytvoření plánu. Celá akce provedení činnosti může probíhat v jiném vlákně než uživatelské rozhraní, proto lze informovat uživatele o průběhu a uživatel má možnost zrušit provádění.

Jelikož je aplikace navržena tak, že je uživatelské rozhraní implementačně odděleno od jádra, je nutné rozšířit třídy `Document`, `InputBlock`, `OutputBlock`, `OperatorBlock` a `Joint` pro

použití v něm. Nově vzniklé třídy jsou rozšířeny o informace o pozici, celkovém vzhledu a stavu na pracovní ploše (zobrazení bloku jako minimalizovaného).

Celý dokument je ukládán ve formátu XML, což přináší flexibilitu při výše zmíněném rozšiřování těchto tříd.

5.8 Vytvoření plánu

Vytvoření plánu není jednoduchá operace. Současná implementace je progresivní typ plánování, kdy se postupně prochází cesta od vstupních bloků směrem k výstupním. Metoda třídy `Document`, `CreateExecutionPlan`, vrací posloupnost akcí pro provedení činnosti a také vrací seznam „nevyřešených" propojů, které pravděpodobně tvoří rychlou smyčku.

Nejprve se naplní seznam nevyřešených propojů a také bloků, ze kterých jsou vyloučeny vstupní bloky přidané rovnou do posloupnosti akcí.

Po tomto inicializačním kroku se prochází smyčkou tak dlouho, dokud nejsou seznamy nevyřešených bloků a propojů prázdné. V této smyčce se řeší, zda jsou pro jistý nevyřešený blok již všechny vstupní propoje v seznamu akcí a zda pro jistý nevyřešený propoj je jeho vstup (tedy výstup bloku) také v seznamu akcí. V kladně vyhodnocených případech se blok či propoj dostane ze seznamu nevyřešených do posloupnosti akcí.

Pokud smyčka nic nepřidá do plánu po celou dobu procházení všech nevyřešených akcí, smyčka se ukončí. Toto opatření zabrání neukončení cyklu plánování v případě, kdy jsou v dokumentu slepé cesty (nejsou propojeny se vstupním nebo výstupním blokem).

5.9 Validace dokumentu

Plán provedení všech akcí pro získání výsledku grafické aplikace souvisí s validací dokumentu. Při generování plánu se totiž kontrolují rychlé smyčky, které nejsou povoleny.

Validace se využívá na kontrolu celého dokumentu, jestli je vše v pořádku. Nejdříve se kontroluje, jestli ke vstupním pinům vede vždy alespoň jeden propoj (injektivní propojení) – více propojů vedoucích k jednomu pinu je nepřípustné. Dále se vygeneruje plán, který mimo jiné vrací propoje, které tvoří smyčku. Veškeré nesrovnalosti se vkládají do výsledné zprávy.

Metoda `Validate` třídy `Document` umožňuje případné nesrovnalosti i opravit. V případě injektivního propojení se všechny nepotřebné propoje odstraní tak, aby zůstal pouze jeden. U smyčky se odstraní všechny propoje, které tuto smyčku vytváří. Tím se dokument neopraví do spustitelné formy, ale je to nejrychlejší cesta k uvedení do provozu.

5.9.1 Nástroj pro validaci dokumentu

Pro validaci dokumentu je vytvořen nástroj pro příkazovou řádku, který využívá knihovnu jádra. Nástroj načte dokument a ověří jeho správnost. Pokud je výsledkem ověření, že dokument je invalidní, je možné pomocí dodatečného parametru tento dokument opravit. Postup opravy a validace je popsán v podkapitole 5.9.

5.10 Spuštění navržené aplikace

Do plánu akcí, vytvořeného plánováním (viz podkapitola 5.8), se vkládají bloky a propoje. Jelikož nejsou tyto třídy návrhově vůbec spojeny, je nutné je pro tuto chvíli sjednotit. Třída `Joint` má metodu `Transfer` a třída `IBlock` má metodu `Process`. Pro účel spojení jsou navrženy třídy `ExecutionPlanItem`, `ExecutionPlanJointItem` a `ExecutionPlanOperatorItem`. První z nich slouží jako předek těm následujícím a obsahuje abstraktní metodu Execute, která v implementaci zavolá pro třídu `Joint` metodu `Transfer` a obdobně u třídy `IBlock` metodu `Process`.

Při psaní této zprávy jsem se zamyslel nad touto implementací a chtěl bych říct, že třídy `Joint` a `IBlock` mohly být ve vztahu pomocí předka a abstraktní metody `Process` (nebo s jiným názvem) a vyhnul bych se tak tomuto řešení.

5.10.1 Nástroj pro spuštění navržené aplikace

Aby spuštění vytvořené aplikace pro zpracování obrazu nebylo odkázáno jen na editor a bylo možné znovu využít tuto aplikaci (i dávkově), byl vytvořen nástroj (využívající knihovnu jádra) pro příkazovou řádku.

Aplikaci se pouze sdělí nové vstupy. To se provádí pomocí parametrů při spuštění – vždy se sdělí ID vstupu a vstupní data, resp. cesta k nim. Detailnější popis použití je uveden v manuálu v příloze.

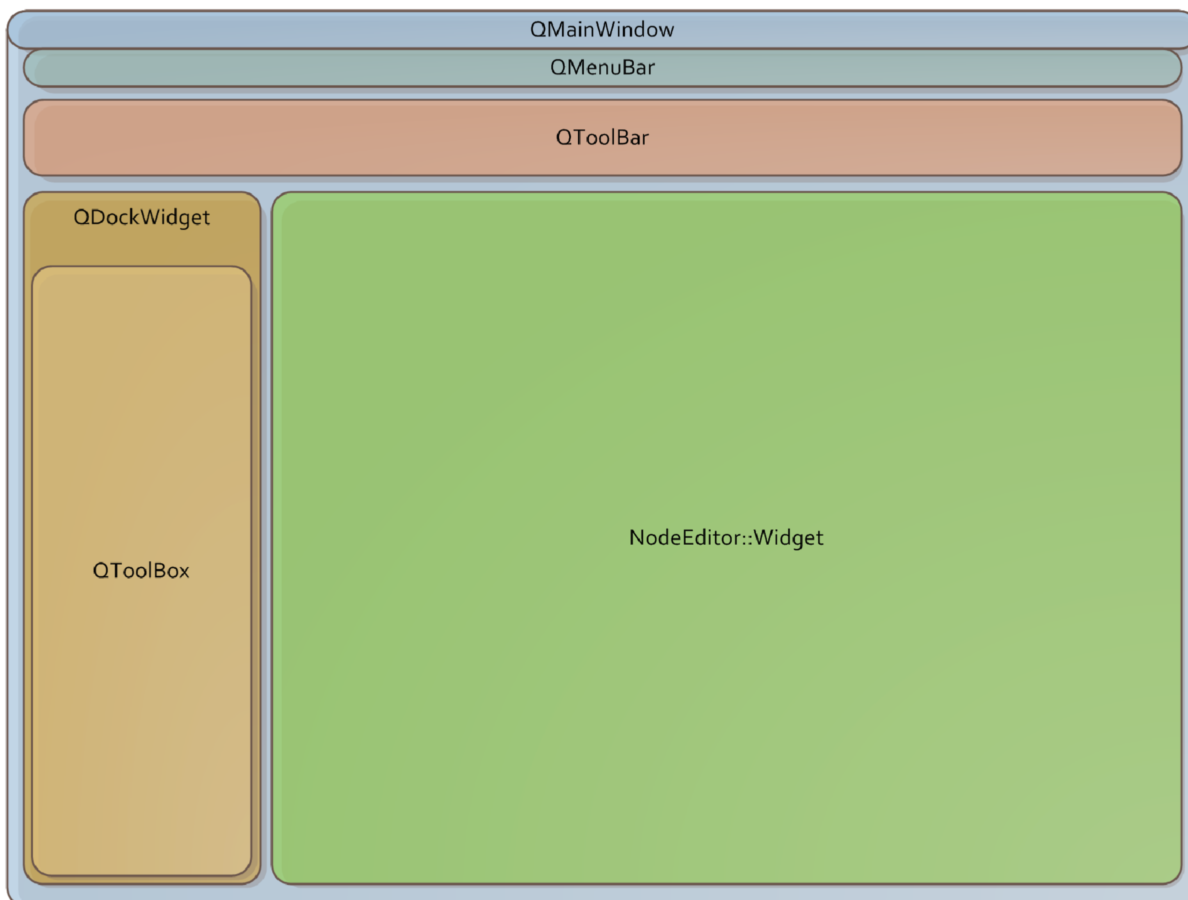
5.11 Uživatelské rozhraní

V následujícím textu jsou rozebrány dvě důležité části uživatelského rozhraní. První část se zabývá rozvržením hlavního okna aplikace, ve kterém se nachází pracovní plocha, což je část druhá. Této druhé části je věnováno více podkapitol, protože není dostupný žádný podobný widget a musel být celý implementován téměř od začátku.

5.11.1 Rozvržení okna aplikace

Dnešní přístup k řešení uživatelských rozhraní je především s využitím grafiky, klávesových zkratek a ovládání myší, a to se zavedenými konvencemi. Tato aplikace nebude výjimkou.

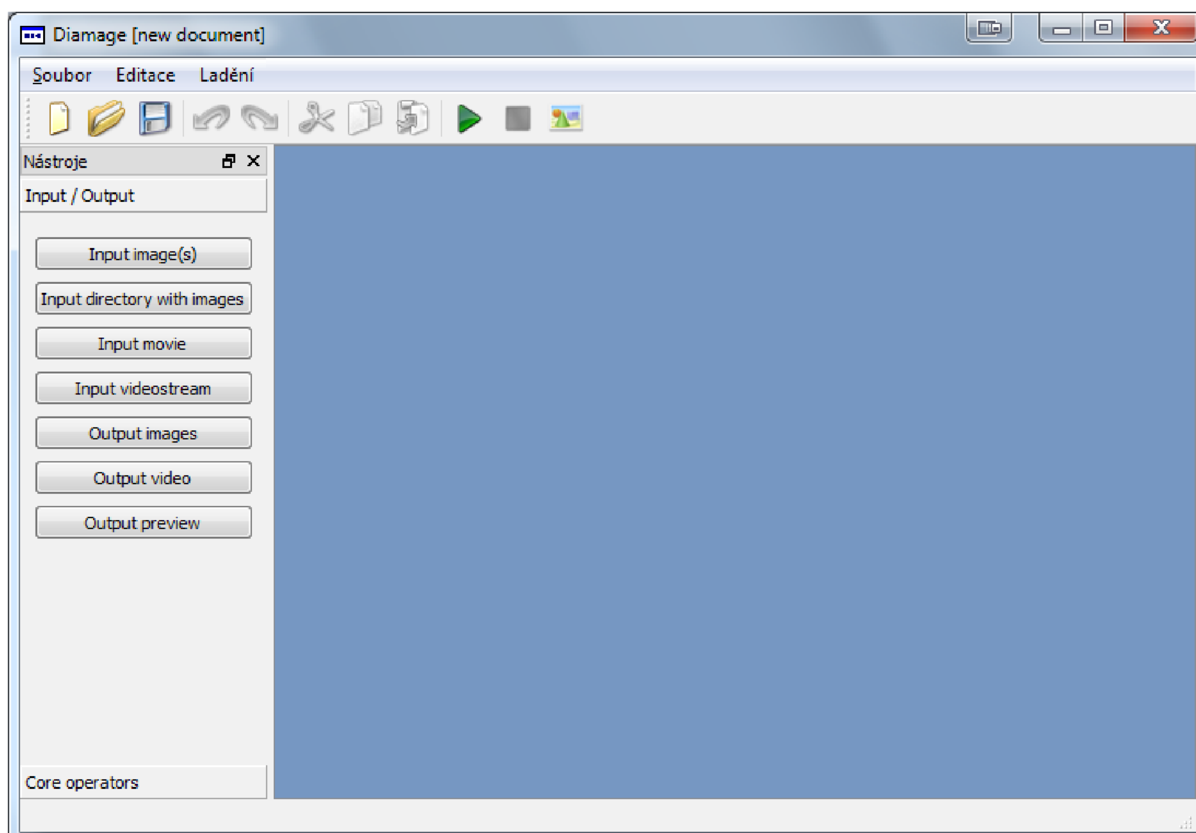
Základní rozvržení se zaměřuje na co největší pracovní plochu, postranní panel a horní nástrojovou lištu s menu. Horní nástrojová lišta má společné příkazy s menu, a slouží hlavně pro uživatele se sklonem využívat více myš než klávesové zkratky.



Ilustrace 15: Schéma rozvržení ovládacích prvků editoru

Postranní panel slouží pro zobrazení výčtu všech bloků, které jdou vložit na pracovní plochu. Tento výčet je kategorizován. První kategorii tvoří vstupní a výstupní bloky, které jsou zabudovány v jádře. Ostatní kategorie jsou tvořeny automaticky. Platí, že jedna kategorie je vytvořena ze všech pluginů dostupných v určité dynamické knihovně (přesněji řečeno, každá továrna operátoru definuje název kategorie, do které operátor spadá). Kliknutím na položku v tomto panelu se vloží odpovídající blok na pracovní plochu a umístí se vedle stisknuté položky.

Hlavní okno umožňuje řídit běh navrhované aplikace. K tomu slouží tlačítka pro spuštění běhu, který lze zastavit tlačítkem pro ukončení běhu nebo se zastaví automaticky. Tento proces běží v druhém vlákně, díky tomu lze během zpracování upravovat parametry. Běh se ukončí automaticky po přehrávání vstupního videa nebo po zpracování všech vstupních obrázků. Další mód běhu je automaticky aktualizovaný náhled. Díky němu je jednoduché se rychle dopracovat k vytouženému výsledku upravením parametrů.



Ilustrace 16: Realizace uživatelského rozhraní hlavního okna

5.11.2 Pracovní plocha uživatelského rozhraní

V této podkapitole jsou popsány původní nápady, jak implementovat pracovní plochu, od kterých se dostaneme k aktuální implementaci. Vytvoření pracovní plochy může být provedeno pomocí dvou způsobů, ty se odlišují přístupem k vykreslování.

První způsob je vykreslování přes buffer, do něj se vykreslí veškeré bloky a propoje, které jsou aktuálně viditelné, a tento buffer se následně vykreslí na obrazovku. Princip je výhodný v tom, že lze využívat přiblížení a oddálení pohledu. Nevýhodou je časté vykreslování ovládacích prvků (ovládací prvky v bloku...) do bufferu, vykreslování bufferu a interakce.

Druhý způsob je zaplnění pracovní plochy upravenými ovládacími prvky. Výhodou je jednodušší implementace. Nevýhodou je absence přiblíženi a oddálení, pravděpodobná nemožnost kontroly některých interakcí v Qt a obrovský počet ovládacích prvků.

5.11.3 Vykreslování uživatelského rozhraní

Qt widgety poskytují metodu pro vykreslení, ale je velkým problémem je donutit reagovat na podněty uživatele. Proto současná implementace kombinuje oba výše popsané způsoby, využívá Qt widget `QGraphicsView`, který funguje jako vykreslování scény, do níž se mohou vkládat i ovládací prvky bez složitého propojování interakce prvku a vykreslování, což je velice důležitá vlastnost.

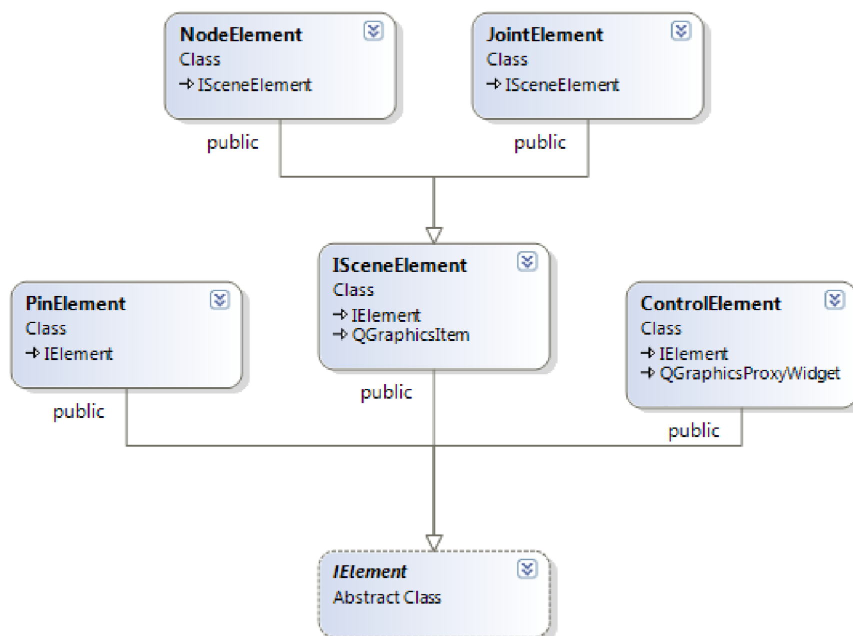
Pro vykreslování byl vyvinut nový widget `NodeEditorWidget`, který je založený právě na `QGraphicsView`. K tomuto řešení vedla dva týdny trvající cesta z výše popsaného prvního způsobu implementace.

Ke správné činnosti `QGraphicsView` je nutné vytvořit instanci třídy `QGraphicsScene.` Ta k reprezentaci scény využívá BSP strom, díky kterému jsou vykreslování a interakce rychlé. Do scény se vkládají veškeré grafické entity a ovládací prvky, pořadí vkládání koresponduje s pořadím ve vykreslování a interakci.

Propoje mezi bloky by se měly vykreslovat prioritně, ale do vykreslovacího procesu nelze zasahovat přímo. Proto je jediným řešením využití tzv. skupin (`QGraphicsItemGroup`). Prvky k vykreslení jsou rozděleny do dvou skupin – první tvoří propoje a druhou skupinu tvoří bloky.

Grafické interpretace bloků a propojů jsou potomky třídy `QGraphicsItem`, mají přepsanou metodu pro vykreslování a určení hranic entity. Třída grafické interpretace bloku (`NodeElement`) obsahuje kontejner na piny (třída `PinElement`), ke kterým se přichytávají propoje. Třída `NodeElement` obsahuje i kontejner na ovládací prvky (třída `ControlElement`).

`ControlElement` dědí z třídy `QGraphicsProxyWidget`, jak nám již název napovídá, bude se jednat o zprostředkovatele komunikace mezi instancí třídy `QGraphicsScene` a widgetem. Díky této třídě se vyřeší problémy popisované v prvním způsobu implementace v předchozí podkapitole 5.11.2. Třída si automaticky vytváří widgety podle datového typu určitého vstupu bloku.



Ilustrace 17: Hierarchie tříd prvků Node editoru

Pro reprezentaci propojení bloků slouží třída `JointElement`, která vykresluje křivku mezi piny. Tato třída se od třídy bloků liší tím, že pozice v grafu scény je vždy v počátku souřadného systému, jen obálka se mění. Ta je velice často mimo počátek. Je to jednodušší způsob implementace, jelikož se při přesouvání bloku nebo jiné akci nemusí aktualizovat hodnota pozice, jen se z pozic pinů vytvoří nová obálka a reprezentace scény se aktualizuje.

5.11.4 Interakce pracovní plochy s uživatelem

Tato podkapitola uzavře vnitřní popis widgetu `NodeEditorWidget`. Grafická reprezentace byla již popsána a nyní zbývá komunikace s uživatelem. Ta je tvořena přepsáním metod `wheelEvent`, `mouseMoveEvent`, `mousePressEvent` a `mouseReleaseEvent`.

Zmíněné metody navzájem spolupracují a musí se mezi nimi udržet kontext. Výjimku tvoří metoda `wheelEvent`, která slouží pouze pro obsluhu měřítka zobrazení (přiblížení/oddálení pohledu).

Kontextem se myslí především akce, která se právě vykonává. Editační akce jsou posun pohledu, přesun bloku, odpojení pinu a propojování dvou pinů. Každá metoda nejdříve ověří, zda by neměla být akce přeposlána widgetu, a v takovém případě ji přepošle nebo vykoná určenou akci.

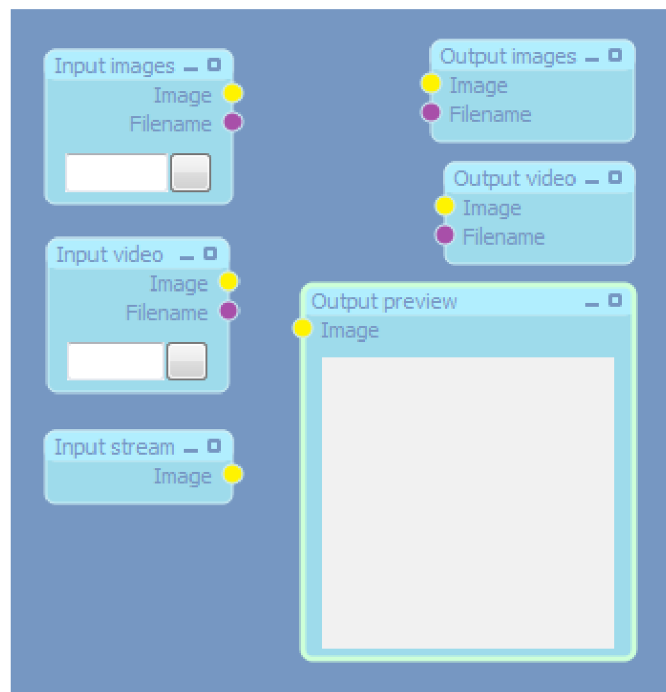
První v řadě je vždy `mousePressEvent`. Podle prvku na pracovní ploše, který se nachází pod kurzorem myši, je vybrána akce a připraví se kontext. Kontext není potřebný v případě, že jde o stisknutí tlačítek na horní liště bloku (tlačítka pro minimalizaci a skrytí ovládacích prvků bloku).

Při pohybu myší nad pracovní plochou se volá metoda `mouseMoveEvent` a při existenci kontextu na akci se zobrazuje vizuální návrh úpravy propoje nebo se přesouvá blok. Při činnosti, kdy se obsah pracovní plochy takto mění, se volá metoda pro její překreslení.

Při uvolnění tlačítka myši je aktivována událost `mouseReleaseEvent`. Pokud je akcí vytvoření propoje, tato skutečnost se zapíše do dokumentu.

5.12 Použití NodeEditorWidget v projektu

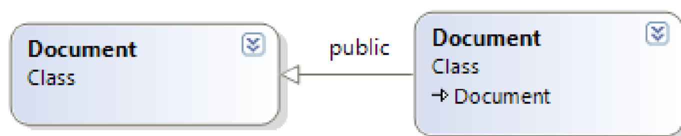
Nyní je na řadě propojení widgetu s jádrem tak, aby widget sloužil k editaci a zobrazení dokumentu. K tomu poslouží specializace tříd `Document`, `InputBlock`, `OutputBlock`, `OperatorBlock` a `Joint`, které budou obsahovat pozici a celkový vzhled bloku na pracovní ploše. Uvedené specializované třídy se jmenují stejně jako jejich předci, jen jsou umístěny v jiném jmenném prostoru. Při zmínce tříd těchto názvů je budeme považovat za specializované. Editorem se považuje `NodeEditorWidget`.



Ilustrace 18: Přehled vstupních a výstupních bloků

Bloky ve svém těle mají umístěné widgety, které reagují na akce uživatele. Proto tyto třídy obsahují sloty (viz 5.2 Qt framework) - z toho důvodu dědí i třídu `QObject` a třída je označena makrem `Q_OBJECT`.

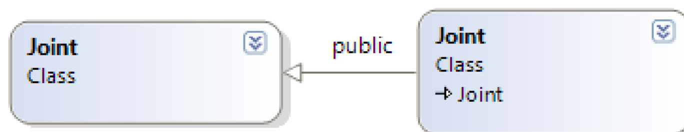
Třída `Document` má přímý přístup k vytvořenému widgetu pro editaci, přepisuje metody načtení a uložení dodatečných dat. Aby byly bloky a propoje správně vytvořeny i s reprezentací v editoru, přepisuje metody pro vytvoření instancí těchto tříd. Dále zavádí historii změn, která je detailně popsána v podkapitole 5.13. V ilustraci 19 lze vidět předka nově implementované třídy `Document` – `Diamage::Processing::Document`.



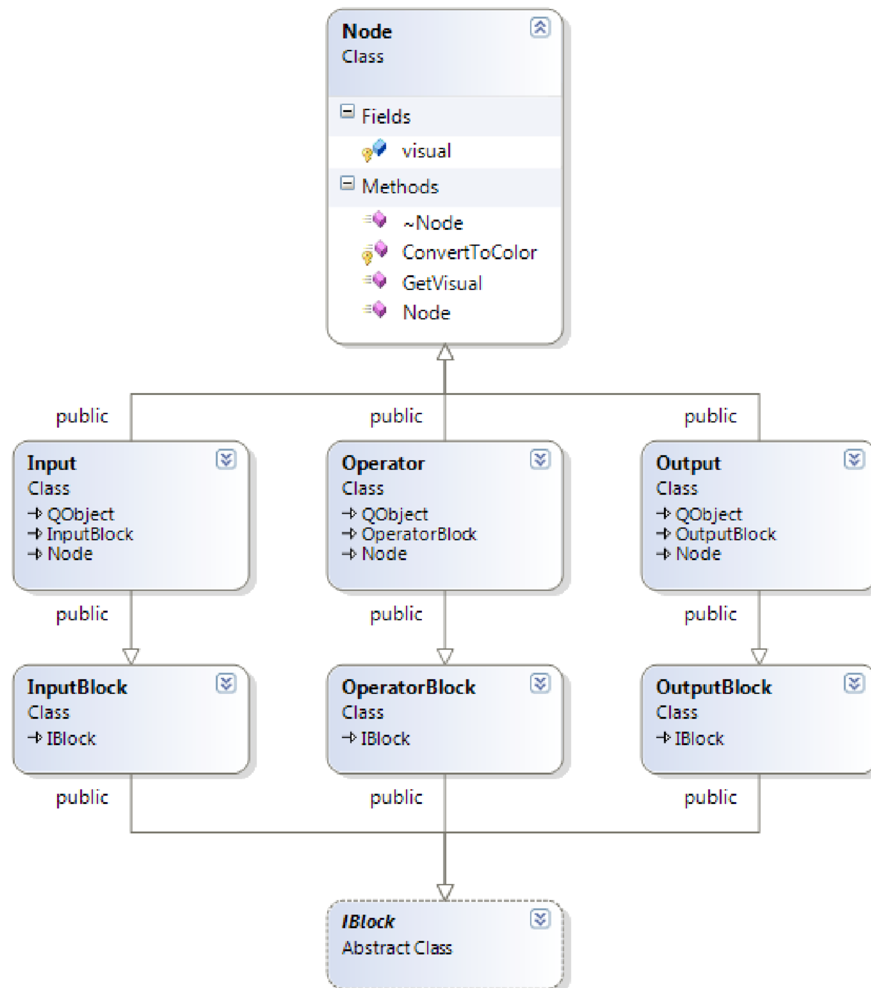
Ilustrace 19: Hierarchie nové třídy Document

Třídy `Input` a `Output` si jsou podobné. Obě mají metodu `InitVisual` pro vytvoření reprezentace v editoru. Navíc obě využívají tříd `ControlElement` pro komunikaci s uživatelem. U vstupního bloku jsou to tlačítka pro výběr souboru, adresáře apod. V případě výstupního bloku se jedná především o blok s náhledem ve vlastním těle.

Třída `Operator` má také metodu pro vytvoření reprezentace v editoru a metodu pro vytvoření widgetů pro ovládání. Tyto widgety jsou vytvářeny podle vstupních parametrů operátoru.



Ilustrace 20: Hierarchie třídy Joint



Ilustrace 21: Hierarchie tříd Input, Operator a Output

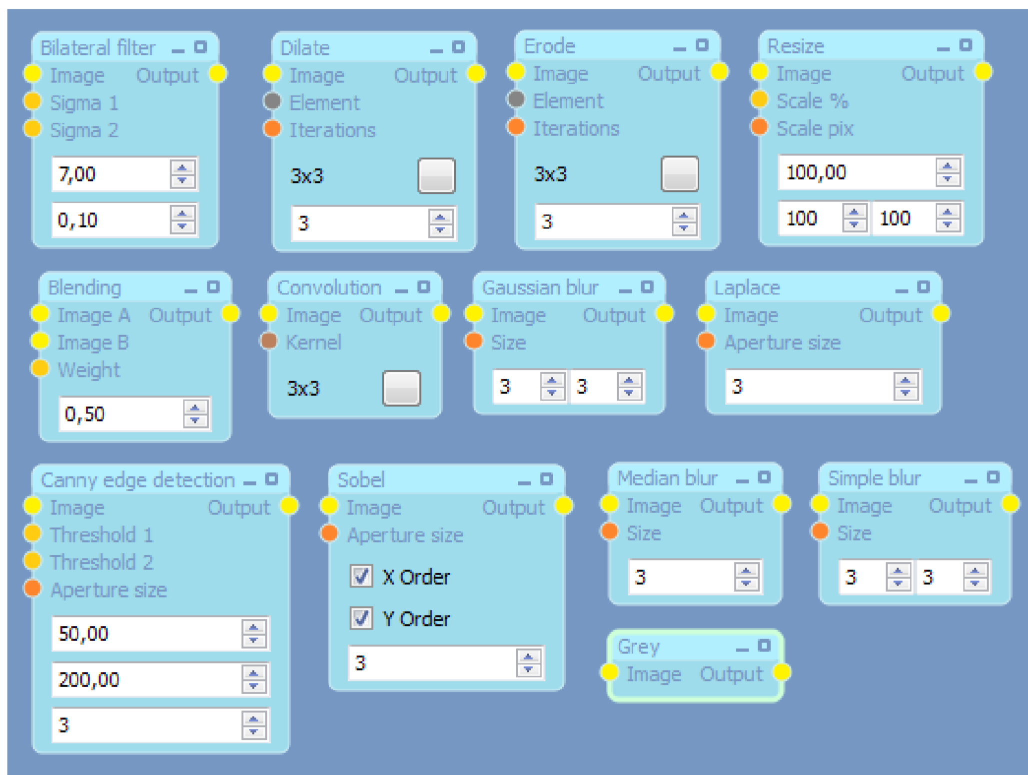
Třídy `Input`, `Operator` a `Output` mají společného předka `Node`, který slouží pro sjednocení a zároveň přidává instanci třídy vizuální reprezentace. Dále rozšiřuje využití XML o pozici a stav zobrazení.

Propoje jsou reprezentovány třídou `Joint`. V tomto případě jen propojuje vizuální reprezentaci a svého předchůdce. Z diagramu tříd v ilustraci 20 lze vidět předka této třídy `Joint` – `Diamage::Processing::Joint`.

5.13 Historie akcí

Historie akcí v editorech je velice nepostradatelným nástrojem. Princip historie je snad v každém editoru stejný, jen se může lišit ovládání. Většinou se využívá systém ovládání „jeden krok

zpět/vpřed“ nebo je k dispozici „více kroků zpět/vpřed“ (např. internetové prohlížeče, Adobe Photoshop).



Ilustrace 22: Přehled zobrazení implementovaných bloků operátorů

Historie akcí je implementována i v nástroji pro tvorbu aplikací s neomezeným počtem kroků (omezením je jen dostatečná paměť). Každá akce je zaznamenána do historie. Akcí může být posun bloku, připojení a změna hodnot ovládacích prvků.

Historie je implementována v třídě `Document` (viz podkapitola 5.12) pomocí listu, do kterého se vkládá celý aktuální stav dokumentu. Optimálním řešením by bylo vkládat pouze provedené změny, což by bylo na realizaci náročnější. Dokument se při akci zpět nebo vpřed celý vyprázdní a načte se jeho stav z historie.

K uložení stavu dokumentu slouží metoda `MakeHistoryNote`, která odstraní stavy novější, než bude tento stav (několikrát se provedla akce zpět a zavolala se metoda `MakeHistoryNote`, přirozeně se provedené akce zpět musí odstranit a uvolnit).

Akce zpět/vpřed je realizována metodami `Undo/Redo`. Ty využívají metod `CanDoUndo/CanDoRedo` pro zjištění, zda se tato akce může provést a obnoví starší/novější stav.

Jelikož jsou některé akce pro zaznamenání do historie vytvořeny samotným widgetem (přesun, propojení) a některé jsou odezvou ovládacích prvků v bloku, je využita složitější síť signálů a slotů

pro jejich zaznamenání. V každém případě je pro zaznamenání vždy volána metoda `DocumentChanged` vytvořeného widgetu. Ta emituje signál `DocumentChangedSignal`, který je propojený se slotem `DocumentChanged` ve třídě hlavního okna `FrmMain`. V tomto slotě je volána již zmíněná metoda `MakeHistoryNote`, uživatelské rozhraní se přizpůsobí novým podmínkám, případně se překreslí výstupní náhledy.

6 Vytvoření pluginů

Jak jsem již zmiňoval, jádro je vytvořeno tak, aby se mohla rozšiřovat jeho funkčnost pomocí pluginů. Pro rozšíření počtu podporovaných grafických formátů slouží abstraktní třídy `ICodecFactory` spolu s `IDecoder` a `IEncoder`. Ke zvýšení počtu funkcí naopak slouží abstraktní třídy `IOperatorFactory` spolu s `IOperator`. Pokud operátor bude potřebovat jako parametr neexistující datový typ, je možné přidat tento datový typ implementací abstraktních tříd `IType` a `IData`. Jakmile tyto třídy implementujeme potřebným způsobem, stačí vytvořit funkci pro zaregistrování těchto rozšíření do jádra aplikace a vše následně zkompilovat jako dynamickou knihovnu.

Pokud jsme se rozhodli vytvořit nový kodek a přejeme si vytvořit enkodér i dekodér, musíme implementovat tyto dvě třídy a pro každou z nich implementovat nové třídy `ICodecFactory`. V případě nového operátoru je nutné pro něj implementovat jeho podpůrnou třídu (továrnu) `IOperatorFactory`.

6.1 Vytvoření nového enkodéru

Při vytváření enkodéru je nutné implementovat tyto metody:

1. `GetType` určující, zda enkodér pracuje s obrázky, videem nebo zařízením (`CT_IMAGE`, `CT_MOVIE`, `CT_VIDEOSTREAM`).
2. `GetFactory`, která vrací instanci třídy `ICodecFactory`, která vytvořila tento enkodér.
3. `SetParameter` pro nastavení jednoho parametru (využitelné zejména u videa).
4. `SetParameters` pro nastavení více parametrů najednou.
5. `Open` pro otevření výstupu.
6. `SaveFrame` pro uložení jednoho snímku do výstupu.
7. `CloseSource` pro uzavření výstupu.

6.2 Vytvoření nového dekodéru

Při vytváření dekodéru je nutné implementovat tyto metody:

8. `GetType` určující, zda enkodér pracuje s obrázky, videem nebo zařízením.
1. `GetFactory`, která vrací instanci třídy `ICodecFactory`, která vytvořila tento dekodér.
2. `SetParameter` pro nastavení jednoho parametru (využitelné zejména u videa).
3. `SetParameters` pro nastavení více parametrů naráz.

4. `Open` pro otevření vstupu.
5. `GetFrame` pro získání jednoho snímku ze vstupu.
6. `CloseSource` pro uzavření vstupu.

6.3 Vytvoření továrny enkodéru či dekodéru

Každý enkodér a dekodér musí mít svou továrnu, proto je nutné také implementovat její metody:

1. `GetExtensions`, která vrací vektor podporovaných přípon souborů.
2. `GetType`, která určuje, zda jde o kodek pracující s obrazovými soubory, video soubory nebo s tokem videa ze zařízení.
3. `GetKind`, která určuje, zda jde o enkodér nebo dekodér.
4. `CreateNewCodecInstance`, která vrátí novou instanci kodeku rovnou k použití.

Navíc se musí v konstruktoru této třídy vytvořit popis parametrů kodeku, které se získávají pomocí metod `GetParametersDesc`, `GetParameterDesc` a `GetParametersCount`.

6.4 Vytvoření nového operátoru a jeho továrny

Nový operátor se vytváří z předka `IOperator` a musí k sobě mít i vlastní továrnu `IOperatorFactory`. U této nové třídy odvozené od `IOperator` stačí přepsat dvě metody:

1. `GetFactory` vracející instanci vlastní továrny.
2. `Process`, které se předají instance vstupních a výstupních operandů – vstupní operandy se zpracují a výstup procesu se vloží na výstup.

Detailnější implementaci naleznete např. v souboru `GreyOperator.h`. Ale obraz řekne více než tisíc slov, proto bude následovat ukázka získání vstupu, inicializace výstupu a provedení převodu do odstínů šedi v příloze. Ale nejdříve je pro pochopení nutné vysvětlit, jak vytvořit továrnu.

Továrna neslouží jen k vytvoření instance operátoru, ale i k enumeraci vstupů a výstupů. To se děje v konstruktoru továrny. Následuje výpis metod pro přepsání:

1. `GetName` pro získání názvu továrny/operátoru.
2. `GetGroupName` pro získání názvu skupiny operátorů, do které tato továrna/operátor spadá.
3. `GetDescription` pro získání detailnějšího textového popisu továrny/operátoru.
4. `GetOperandRange`, která nastavuje minimum, maximum a krok pro editaci hodnot číselných parametrů.

5. CreateNewOperatorInstance, která vrací novou instanci operátoru.

Důležitou součástí je již zmíněný konstruktor, ve kterém se vyplní vektory instancemi tříd NodeDesc. U výstupů NodeDesc popisují jen datový typ, u vstupů mohou přidat i výchozí (počáteční) hodnoty parametrů. Třída OperatorBlock (viz podkapitola 5.7) si vytvoří z těchto informací vlastní vektory vyplněné výchozími hodnotami.

6.5 Vytvoření nového datového typu

Nový datový typ souvisí s implementací dvou abstraktních tříd IType a IData. Již bylo zmíněno, že třída IType slouží jako továrna a k porovnání datových typů jako sémantická kontrola. Diagram těchto tříd naleznete v podkapitole 5.5 Část jádra s podporou pluginů. Nejprve se zaměříme na její přepsání:

1. GetName metoda slouží k informování o názvu datového typu (např. String).
2. GetSize metoda slouží k informování o velikosti datového typu. Je-li velikost nepředvídatelná, smí vracet nulu.
3. CompareTypes se přepisuje jen za předpokladu, že je nutné nějakým způsobem upravit porovnání.
4. CreateNewInstance pro vytvoření nové instance třídy IData.

Metoda CompareTypes, jak je tomu u typů ImageType, ImageGrayType, ImageRGBType a ImageLevelType, může sloužit jako k porovnání příbuzných typů (ImageType zastupuje všechny zmíněné tři další třídy, ale mezi sebou jsou nekompatibilní).

Třída IData má méně metod k přepsání (ne všechny jsou povinné):

1. Clone je povinná metoda k přepsání, vytváří novou instanci s kopií obsažených dat.
2. LoadFromXML slouží pro načtení obsažených dat z části XML dokumentu.
3. SaveToXML slouží pro uložení obsažených dat do určité části XML dokumentu. Tyto dvě metody slouží pro podporu uložení a načtení navrhované aplikace.

6.6 Hlavní modul dynamické knihovny

Dynamická knihovna může obsahovat pro tuto aplikaci více pluginů. A to z toho důvodu, aby operátory, které používají svůj nový vlastní datový typ, byly umístěny ve stejné dynamické knihovně, což usnadní psaní pluginů.

Dynamická knihovna musí mít pro použití v této aplikaci jeden přístupový bod:

```
extern "C" PLUGIN_EXPORT void *GetDllPlugins(float apiVersion) throw()
{
   Diamage::Logs::Messages("NewPlugins.dll") << "New plugins is loading";
   if (apiVersion < PLUGIN_API_VERSION) {
      Diamage::Logs::Messages("NewPlugins.dll")
         << "Could not load plugins, incorrect plugin API version";
      return NULL;
   }
   return new Plugin();
}
```

Ukázka 4: Přístupový bod dynamické knihovny obsahující pluginy

Výše popsaná funkce ověří, zda je používáno jádro potřebné verze (makro `PLUGIN_API_VERSION`) a poté vytvoří instanci třídy `IPlugins`. Ta zaregistruje veškeré pluginy obsažené v dynamické knihovně, v singletonu třídy `PluginManager`. Registrace se provádí v přepsané metodě `Register`. Další metoda pro přepsání je metoda `GetName`(), která vrací název pluginu.

Takto napsaná a sestavená dynamická knihovna je „čitelná“ jádrem aplikace. Implementace nové dynamické knihovny je časově nenáročná, i když obsahuje hodně částí k implementaci. Jako vzor může posloužit knihovna `CorePlugins`, která se metodou „copy&paste“ snadno stane novou knihovnou pluginů.

7 Výsledky

Výsledkem implementace jsou tři nástroje, které splňují zadání diplomové práce. Jedná se o vizuální nástroj pro tvorbu aplikací ve zpracování obrazu, nástroj pro validaci takové aplikace a nástroj pro spuštění aplikace. Vše je implementováno v jazyce C++, využívá se OpenCV knihovna a Qt framework. Projekt je odladěn v prostředí MS Windows, zvolený jazyk a knihovny zaručují snadné sestavení i na jiných dnes používaných operačních systémech.

Společnou součástí všech uvedených nástrojů je jádro tvořené knihovnou. Toto jádro podporuje systém zásuvných modulů (pluginů) pro rozšíření. Rozšířením mohou být nové vstupní a výstupní formáty, nové metody zpracování obrazu a nové datové typy. Jádro obsahuje i reprezentaci navrhované aplikace, umožňuje její editaci, validaci a provádění. Do validace patří automatická sémantická kontrola a kontrola jiných zakázaných případů propojení.

Vizuální nástroj má jednoduché ovládání a umožňuje propojovat funkční bloky pro docílení výsledného chování aplikace pro zpracování obrazu. K tomuto účelu byl vytvořen grafický ovládací prvek, Node editor widget, který slouží k vizuálnímu propojování bloků. Prvek umožňuje barevně odlišit piny a vkládat další ovládací prvky (widgets) přímo do těla bloku.

Vizuální nástroj umožňuje spustit navrhovanou aplikaci přímo v něm nebo využít režimu náhledu v reálném čase – aplikace ihned reaguje na změny v návrhu a také je zobrazí. Mimo standardní operace jako jsou uložení a otevření aplikace ze souboru obsahuje nástroj i historii akcí, tedy podporu příkazů akce zpět (undo) a akce vpřed (redo). Vše je doplněno o pohodlné klávesové zkratky.

Pro režim náhledu v reálném čase slouží výstup, který ve svém těle zobrazuje připojená data jako malý náhled. Dvojitým kliknutím na toto tělo se otevře okno se zobrazením úplné velikost.

Nástroj pro validaci aplikace je konzolový program, informuje o chybách v aplikaci v tom smyslu, že není spustitelná – obsahuje sémantické a jiné chyby). Dokáže aplikaci i opravit, ale výsledkem často bývá neúplná aplikace, která už je validní, ale již nemusí být spustitelná.

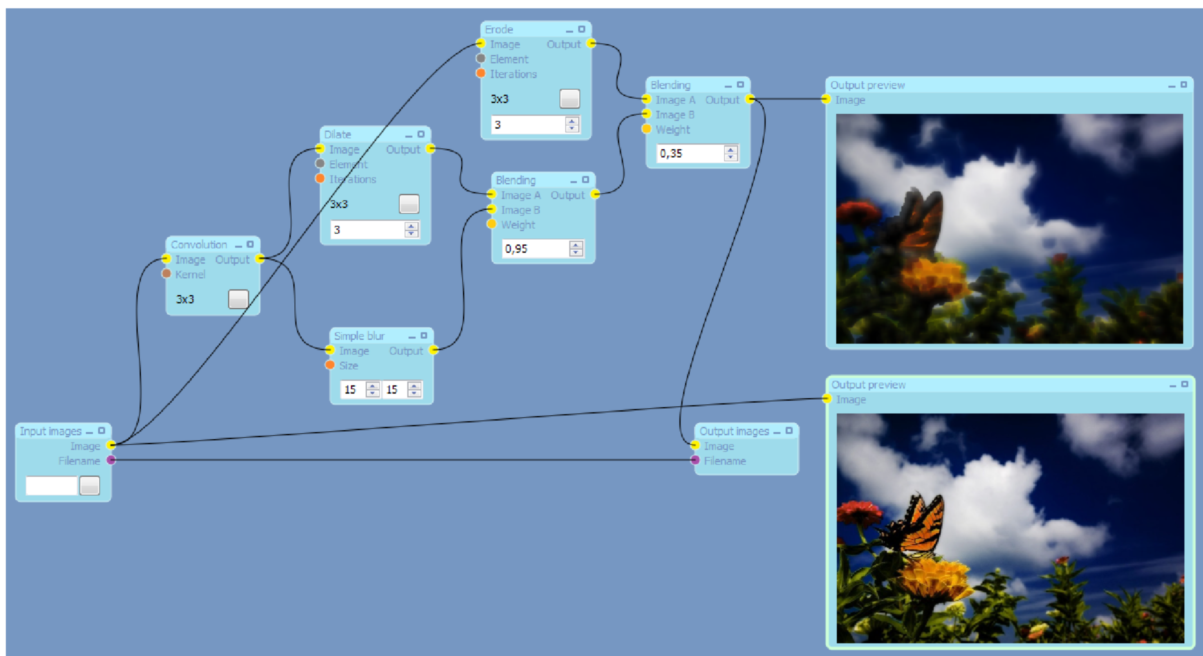
Posledním z nástrojů je konzolový nástroj pro spuštění aplikace. Ten umožňuje dávkové zpracování pomocí navržené aplikace. Tento nástroj je vhodnou volbou pro rychlé použití hotové aplikace.

Základní dostupné bloky jsou:

- Vstupní bloky (obraz, video, kamera)
- Výstupní bloky (obraz, video)
- Bilaterální filtr
- Blending

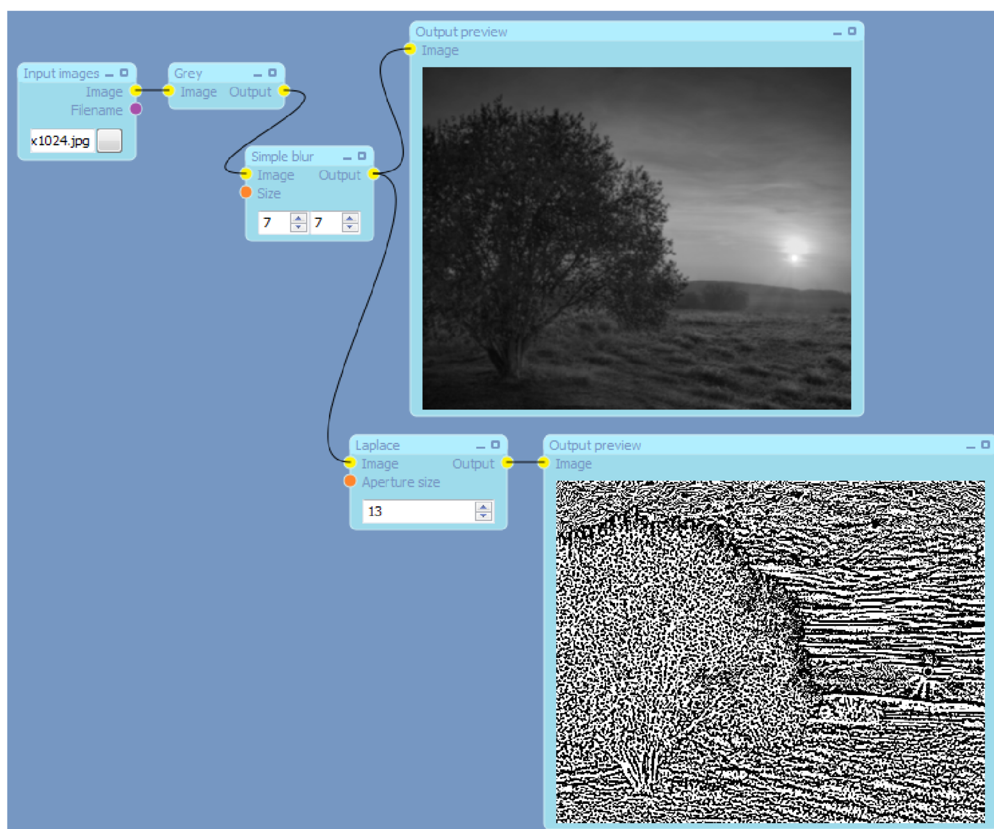
- Cannyho hranový detektor
- Konvoluce
- Dilatace
- Eroze
- Gaussovské rozostření
- Převod na stupně šedi
- Laplaceův operátor
- Medián
- Změna velikosti
- Jednoduché rozostření
- Sobelův operátor

Implementace obsahuje tři spustitelné soubory a dvě dynamické knihovny. Zdrojový kód je rozložen v 75 souborech (bez generovaných). Pro prezentaci projektu jsem vytvořil webovou prezentaci a plakát o formátu A3. Na internetových stránkách http://www.riotix.com/diamage je tento projekt ke stažení.



Ilustrace 23: Ukázka vytvoření jednoduché aplikace

Ke konci vývoje nástroje pro editaci byl proveden neoficiální jednoduchý průzkum a zpětná vazba mezi vybranými uživateli. Jejich výsledkem bylo přidání režimu náhledu v reálném čase, který původně v návrhu nebyl, ale zvýšil komfort. Z průzkumu vyplývá, že je projekt zajímavý a má určitý potenciál.



Ilustrace 24: Využití laplaceova operátoru na obrázku s vysokým rozlišením

Možností pro budoucí vývoj je několik. Aby se vyplatilo nástroj jej použít na tvorbu mnohem více aplikací a mohl konkurovat i jiným nástrojům, musí mít mnohem větší počet pluginů, než které jsou v současné době implementovány. S tím souvisí i rozšíření třídy `IType` o abstraktní metodu `ShowEditor`, ta by zobrazila editor pro nový datový typ, který v současné době nelze editovat pomocí nástroje pro návrh. Důležitou změnou je nahrazení současného plánování regresivním plánováním, aby se opravdu do plánu provádění navržené aplikace nedostávaly slepé cesty apod.

Další věci, které by mohly pomoci experimentujícím uživatelům, jsou nástroje pro ladění, které byly navrženy v kapitole 4, a také operátor vykonávající skript.

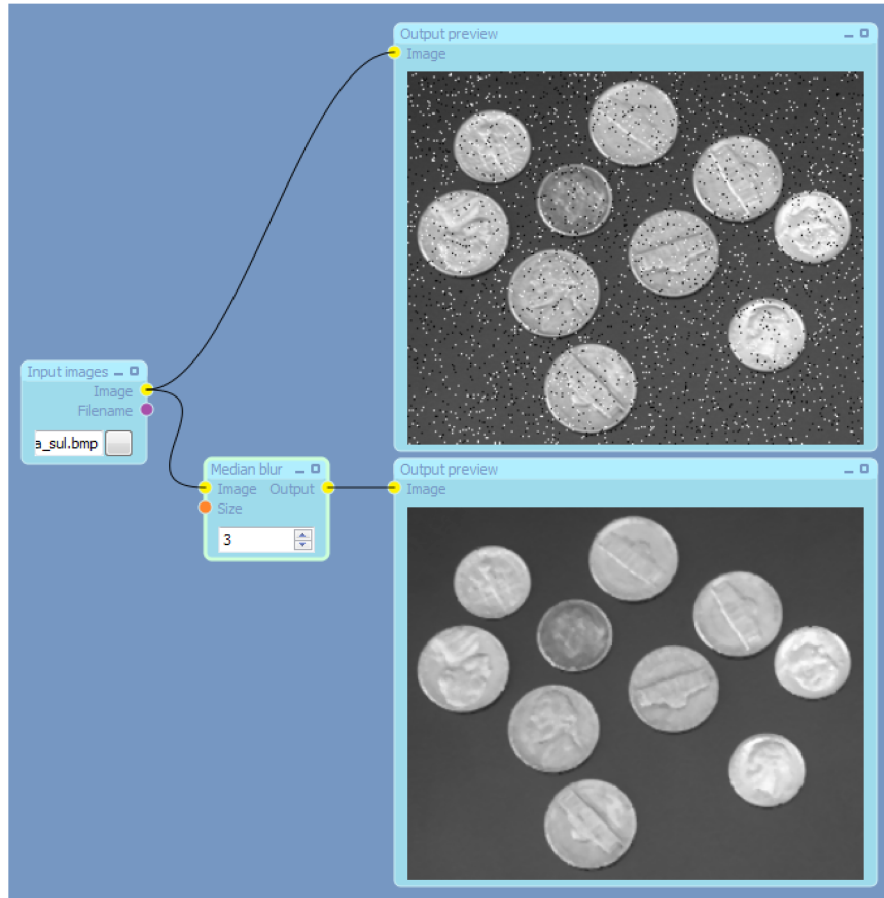
7.1 Vyhodnocení měření stráveného času nad režií při zpracování

Předmětem testu je vyhodnocení, jakou režii má jádro při zpracování. To znamená, že se porovnávají dva časy:

1. Jak dlouho trvá běh aplikace vytvořené v nástroji.

2. Jak dlouho trvá běh stejné aplikace, která je napsaná přímo v C++.

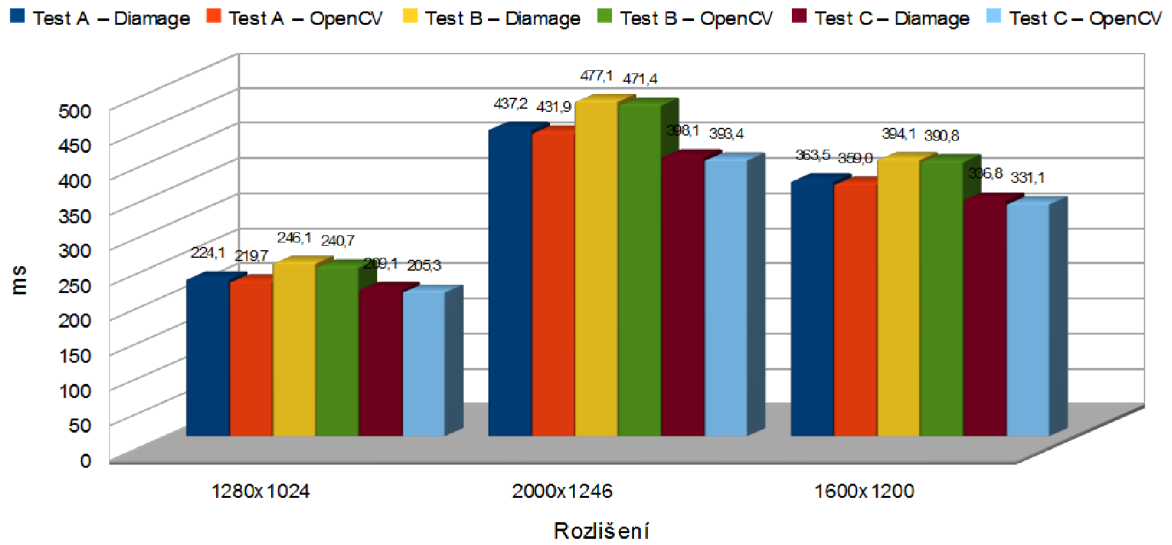
V případě prvního času oproti druhému je započtena režie, která zahrnuje vytvoření plánu, předávání dat mezi bloky, režie třídy `IDataPtr`, režie spouštění jednotlivých operací atd.



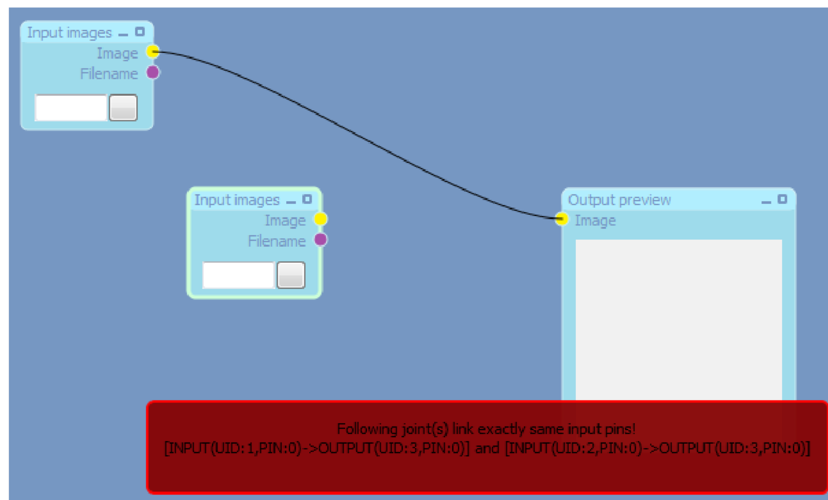
Ilustrace 25: Použití mediánu pro odstranění šumu typu pepř a sůl

Testovaly se tři aplikace se třemi různými vstupními obrázky v sérii opakování stokrát na počítači s procesorem Intel Core 2 Quad Q9300 @ 2500MHz. První aplikací byla z ilustrace 23, ostatní aplikace jsou modifikací. V druhé aplikaci byl blok jednoduchého rozostřením nahrazen blokem gaussovského rozostření. Třetí aplikace neobsahuje konvoluci.

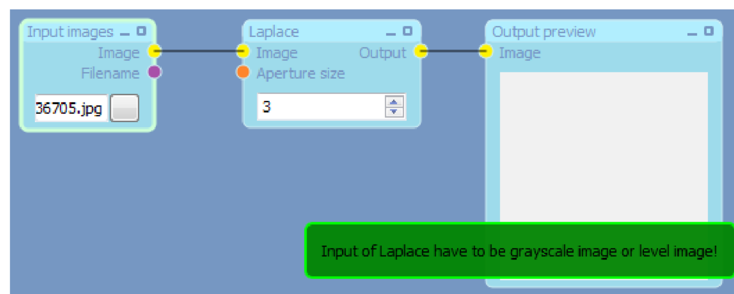
Očekáváním je, že by čas režie (rozdíl dvou výše popsaných časů) u první a druhé aplikace měly být teoreticky stejný, u třetí aplikace by měl být nižší. Tento předpoklad není jednoduché potvrdit, protože je rozdíl mezi časy velmi malý. Je velice pravděpodobné, že je výše zmíněné očekávání správné, dle získaných výsledků. Výsledky časů jsou k dispozici v grafu na ilustraci 26.



Ilustrace 26: Graf výsledků



Ilustrace 27: Zobarzení chyby při vytvoření dvou propojů u jednoho vstupního pinu



Ilustrace 28: Zobrazení chyby při aktualizaci náhledu v reálném čase

8 Závěr

V rámci diplomové práce bylo mým úkolem zorientovat se v současných nástrojích a knihovnách pro zpracování obrazu, navrhnout jednoduchý nástroj pro vizuální skládání aplikace a experimentovat s implementací.

Porovnáním a zorientováním se v několika nástrojích pro vizuální skládání jsem získal mnoho představ, jakým směrem by se mohl vzhled a chování aplikace vydat. Směr, jaký jsem popsal v kapitolách 4 a 5 se mi jeví jako nejpřijatelnější pro začínajícího i zkušeného uživatele a považuji ho zároveň i za přehledný. Grafický styl, chování a některé další prvky jsou velice podobné Node editoru v Blenderu. Jelikož jsem se rozhodl pro objektový jazyk, kladl jsem velký důraz na objektový návrh.

Výhodou je i rozšiřitelnost díky využití dynamických knihoven. Tento nástroj jsem navrhnul jako funkčně soběstačný a přidáváním pluginů jeho použitelnost roste. Pluginy mohou využívat funkcí OpenCV knihovny nebo i vlastní funkce, které jsou napsány pro GPU. Přidáním zmíněného skriptovacího bloku (operátoru) využití této aplikace vzroste, jelikož usnadní psaní nových bloků pro zpracování obrazu.

V druhé fázi jsem implementoval uživatelské rozhraní, funkce pro zpracování obrazu (operátory) a dokončil chybějící prvky jádra. Uživatelské rozhraní využívá ovládací prvek (widget) pro propojování bloků, který jsem musel vytvořit, protože neexistuje dostupná alternativa pro Qt framework. Prvek jsem navrhl obecně, je možné jej použít i v jiných projektech využívajících právě tento framework, na kterém je celá moje práce založená. Vizuální nástroj pro vytvoření aplikací ve zpracování obrazu využívá tento prvek a propojuje jej s funkcemi jádra. Je tak umožněno i provádění aplikace v tomto nástroji.

Jádro projektu jsem během vývoje vizuálního nástroje doplnil o provedení zpracování obrazu, o režim náhledu v reálném čase a validaci aplikace. Také jsem provedl drobné úpravy, na které jsem při návrhu nemyslel.

Moje představa o dalším vývoji nástroje je jednoznačná, především by potřeboval doplnit další pluginy a umožnit tvorbu pluginů pomocí skriptovacího jazyka (pravděpodobně jazyka Python). Zvýšení počtu pluginů zvyšuje využitelnost nástroje. S pluginy souvisí i přidání některých rysů do jádra a editoru, které momentálně nejsou k dispozici. Experimentující uživatelé nebo vývojáři by jistě přivítali možnost jednoduše ladit jejich operátory a případně nějaký zajímavý nástroj, který by lépe ukazoval výsledek než je dosavadní okno po rozkliknutí bloku s náhledem.

Přínosem práce je nástroj pro jednoduché vytvoření aplikace ve zpracování obrazu, v dávkovém zpracování obrazu pomocí takto vytvořené aplikace a v testování různých filtrů a jejich parametrů.

Literatura

[1] Kolektiv autorů: *Blender Composite Nodes* [online]. Dostupný na WWW: http://www.blender.org/development/release-logs/blender-242/blender-composite-nodes/ (prosinec 2009)

[2] Gladman, Simon: *Pixel Bender Blender Prototype* [online]. Dostupný na WWW: http://flexmonkey.blogspot.com/search/label/PixelBender (prosinec 2009)

[3] Novaumas Pau: *TecnoFreak Animation System* [online]. Dostupný na WWW: http://sourceforge.net/projects/tecnofreakanima/ (prosinec 2009)

[4] Vašek, Milan: *Pár rad pro začínající 3D grafiky* [online]. Dostupný na WWW: http://www.milanvasek.com/blog/?page_id=201 (prosinec 2009)

[5] Chalupa, Radek: *1001 tipů a triků pro Visual C++*. Brno, Computer Press, 2003, ISBN 80-7226-842-2

[6] Sobell, Mark: *Mistrovství v Linuxu*. Brno, Computer Press, 2007, ISBN 978-80-251-1726-2

[7] Pokorný, Pavel: *Blender – naučte se 3D grafiku*. Praha, BEN – technická literatura, 2009, ISBN 978-80-7300-244-2

[8] Microsoft Corporation: *DirectShow* [online], 2010. Dostupný na WWW: http://msdn.microsoft.com/en-us/library/dd375454(v=VS.85).aspx (květen 2009)

[9] Jánoš, I.: *Monogram GraphStudio* [online]. Dostupný na WWW: http://blog.monogram.sk/janos/tools/monogram-graphstudio/

[10] Microsoft Corporation: *Overview of GraphEdit* [online]. Dostupný na WWW: http://msdn.microsoft.com/en-us/library/ms787460.aspx

[11] Bradski, G., Kaehler, A.: *Learning OpenCV.* O'Reilly, 2008, ISBN 978-0-596-51613-0

[12] Blanchette, J., Summerfield, M.: *C++ GUI programming with Qt 4 (2nd edition).* Prentice-Hall, 2008, ISBN 0-13-235416-0

[13] Barták, R.: *Plánování a rozvrhování* [online]. Dostupný na WWW: http://ktiml.mff.cuni.cz/~bartak/planovani/index.html (květen 2010)

[14] Kolektiv autorů: *Extensible markup language (5th edition)* [online]. Dostupný na WWW: http://www.w3.org/TR/REC-xml/ (květen 2010)

Seznam příloh

Příloha 1. Uživatelský manuál

Příloha 2. Implementace pluginu pro zpracování obrazu (ukázka)

Příloha 3. CD se zdrojovými kódy, dokumentací a plakátem

Příloha 1. Uživatelský manuál

Diamage nástroj

Menu & toolbar

- New (Ctrl + N, 1. tlačítko) – Vytvoří nový dokument (aplikace pro zpracování obrazu).
- Open... (Ctrl + O, 2. tlačítko) – Otevře dokument ze souboru.
- Save (Ctrl + S, 3. tlačítko) – Uloží dokument.
- Save as... – Uloží dokument do jiného souboru.
- Undo (Ctrl+Z, 4. tlačítko) – Vrátí se v historii o krok zpět.
- Redo (Ctrl+Y, 5. tlačítko) – Skočí v historii o krok vpřed.
- Cut (Ctrl+X, 6. tlačítko) – Vyjme vybraný prvek do schránky.
- Copy (Ctrl+C, 7. tlačítko) – Zkopíruje vybraný prvek do schránky.
- Paste (Ctrl+V, 8. tlačítko) – Vloží prvek ze schránky.
- Remove (Del) – Odstraní vybraný blok nebo propoj.
- Run (F5, 9. tlačítko) – Spustí navrhovanou aplikaci pro zpracování obrazu.
- Stop (10. tlačítko)– Ukončuje běh navrhované aplikace.
- Interactive preview (11. tlačítko) – Při změně v dokumentu (aplikaci) se aktualizuje vzhled výstupu.
- About (F1) – Zobrazí okno s informacemi o programu.

Blocks toolbar

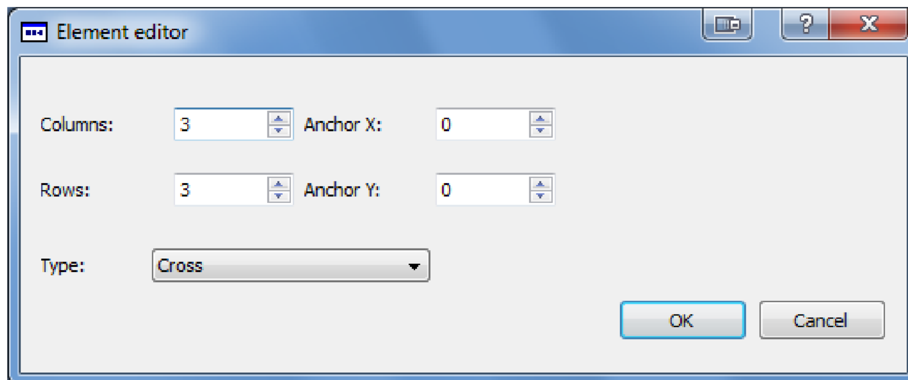
- Tlačítko seskupení – Zobrazí bloky dostupné v určitém seskupení.
- Tlačítko bloku – Vloží blok do dokumentu.

Pracovní plocha

- Kolečko myši – přiblížení/oddálení
- Pravé tlačítko myši + posuv – posun pohledu na plochu
- Levé tlačítko myši
  - Výběr bloku nebo propoje
  - Minimalizace bloku
  - Maximalizace bloku

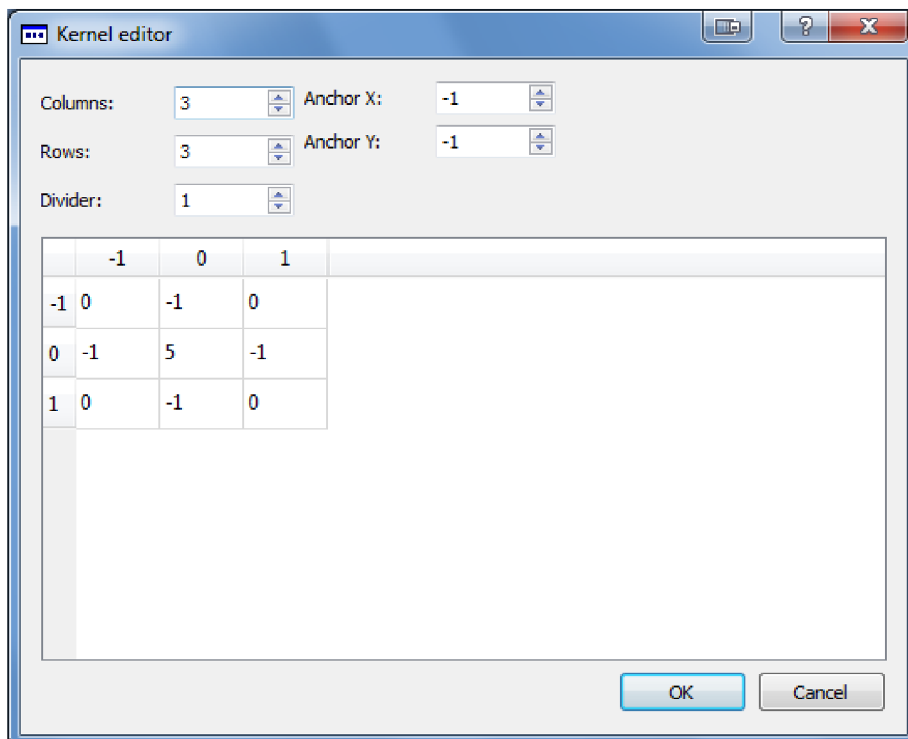
- Interakce s prvky v bloku

Element editor



- Columns, Rows – Velikost elementu
- Anchor X, Anchor Y – Ukotvení elementu
- Type – Typ elementu (obdélník, kosodélník, elipsa)

Kernel editor



- Columns, Rows – Velikost matice
- Divider – číslo, které dělí všechny prvky matice
- Anchor X, Anchor Y – Ukotvení jádra

Validator nástroj

Nástroj provede kontrolu navržené aplikace. Parametry nástroje jsou:

```
validator <file> [-r]
```

- `<file>` – navržená aplikace, která se má kontrolovat.
- `-r` – nástroj aplikaci i opraví (nepovinný přepínač).

DiamageCmd nástroj

Nástroj slouží pro spuštění aplikace z příkazové řádky. Pokud je potřeba zaměnit vstup, který je nastaven již od návrhu, slouží k tomu parametry. Parametry pro spuštění jsou:

```
diamagecmd <file> <ID:source>
```

- `<file>` – navržená aplikace, která se má spustit.
- `<ID:source>` – pár `ID` a zdroj dat slouží pro záměnu vstupu. `ID` je `ID` vstupu (lze zjistit), zdroj dat je cesta k souboru, souborům nebo adresáři.
    - `file://PATH_TO_FILE` – zdroj je tvořen jedním souborem.
    - `file://PATH_TO_FILE1;...;file://PATH_TO_FILEn` – zdroj je tvořen n soubory, které se dávkově zpracují.
    - `dir://PATH_TO_DIR` – zdroj je tvořen všemi obrazovými soubory v definovaném adresáři.

Příloha 2. Implementace pluginu pro zpracování obrazu (ukázka)

```
class GreyOperator : public IOperator {
public:
   /// Get operator descriptor and factory
   virtual IOperatorFactory *GetFactory() {
      return GreyOperatorFactory::singleton;
   }


   /// Process input and return result in output
   /// @param input Input node operands
   /// @param output Output node operands
   ///           (have to be initialized before call this method)
   /// @param result Result and progress informations
   /// @returns Returns true if sucess, otherwise returns false.
   virtual void Process(const NodeOperands &input,
         NodeOperands &output, Diamage::Core::ProcessProgress *result)
   {
      // Get input operands
      Diamage::Types::ImageRGB *inputImage =
            (Diamage::Types::ImageRGB*)input[0].getPointer();
      int width = inputImage->GetWidth();
      int height = inputImage->GetHeight();


      // Init output
      ImageGray *outputImage = static_cast<ImageGray*>(
            ImageGrayType::GetType()->CreateNewInstance());
      outputImage->Initialize(width, height);
      output[0] = IDataPtr(static_cast<IData*>(outputImage));


      try {
         cvCvtColor(inputImage->GetDataStruct(),
               outputImage->GetDataStruct(), CV_BGR2GRAY);
      } catch (...) {
         int err = cvGetErrStatus();
         if (err < 0) {
            THROW_EXCEPTION(cvErrorStr(err));
         } else {
            THROW_EXCEPTION("Grey operator reaches assert!");
         }
      }
   }
};
```

```
class GreyOperatorFactory : public Diamage::Core::IOperatorFactory {
protected:
   static GreyOperatorFactory *singleton;
public:
   /// Constructor
   GreyOperatorFactory() {
      this->singleton = this;
      this->inputNodesDesc.push_back(NodeDesc(ImageRGBType::GetType(), "Image"));
      this->outputNodesDesc.push_back(NodeDesc(ImageGrayType::GetType(), "Output"));
   }


   /// Destructor
   virtual ~GreyOperatorFactory() { this->singleton = NULL; }


   /// Get operator name
   virtual const String &GetName() {
      static const String name = "Grey";
      return name;
   }
   /// Get group name
   virtual const String &GetGroupName() {
      static const String name = "Core operators";
      return name;
   }
   /// Get operator description
   virtual const String &GetDescription() {
      static const String desc = "Converts color image to grey image";
      return desc;
   }


   /// Get operand range
   /// @param operand Operand name
   /// @param min Minimal value
   /// @param max Maximal value
   /// @param step Step size
   virtual void GetOperandRange(const String &operand,
                              int64 &min, int64 &max, int64 &step)
   { min = 0; max = 0; step = 0; }
   /// Get operand range
   /// @param operand Operand name
   /// @param min Minimal value
   /// @param max Maximal value
   /// @param step Step size
   virtual void GetOperandRange(const String &operand,
                              uint64 &min, uint64 &max, uint64 &step)
   { min = 0; max = 0; step = 0; }
   /// Get operand range
   /// @param operand Operand name
   /// @param min Minimal value
   /// @param max Maximal value
   /// @param step Step size
   virtual void GetOperandRange(const String &operand,
                              double &min, double &max, double &step)
   { min = 0; max = 0; step = 0; }


   /// Create instance of plugin operator
   virtual Diamage::Core::IOperator *CreateNewOperatorInstance() {
      return new GreyOperator();
   }
};
```