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

Bakalářská práce

Interaktivní manipulace s bitmapovým obrazem

Plzeň 2011

Jan Kovanda

Zadání práce

Prohlášení

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

V Plzni dne .... Jan Kovanda ....

Poděkování

Na prvním místě bych chtěl poděkovat vedoucímu bakalářské práce Ing. Petru Lobazovi. Vděčím mu za to, že mě uvedl do tajů digitálního retušování a barevných korekcí obrazu. S takto získanými vědomostmi jsem mohl zpracovat tuto práci. Hlavně bych mu chtěl poděkovat za to, že kdykoliv jsem ho navštívil s problémem, s kterým jsem si neuměl poradit, vždy si našel čas, aby mi pomohl.

Rád bych poděkoval mojí rodině, která mě povzbuzovala, když mi práce nešla od ruky.

Abstract

This document discuss an inner structure of a program for advanced colour corrections of photos and video. It contains a description of the program components and its implementation. Also contains the description of the key algorithm for correct processing of a final image. The image put together from partial layers of a logical model.

It is not necessary to reimplement basic operations for an image modification. Therefore for a simplification of the development process the decision to include an open source graphical libraries for an image modification was made. After a quick search through appropriate candidates, the two most fitting are described. By testing their performance and abilities, the winner is chosen.

The winner is used as a cornerstone of a test application. The application is focussed on a creation, a loading and a right usage of user defined plugins (operations) used by the graphical library.

Keywords: inner structure, advanced colour correction, graphical library, test, user defined plugins

Abstrakt

Tento dokument pojednává o návrhu vnitřní struktury programu pro pokročilé barevné korekce fotografií a videa. Obsahuje popis jednotlivých komponent programu, návrh jejich implementace a popis klíčového algoritmu pro korektní zpracování výsledného obrazu z dílčích vrstev logického modelu.

Aby nebylo třeba psát základní operace editace obrazu, bylo navrženo použití volně dostupné grafické knihovny pro úpravu obrazu. Po rychlém výběru vhodných kandidátů jsou popsány dva nejvhodnější. Jejich testováním je na základě výkonu a schopností vybrán vítěz.

Tento vítěz je použit jako základní stavební kámen pro testovací aplikaci, která je zaměřena na vytvoření, načtení a správné použití uživatelsky definovaných pluginů (operací), se kterými je schopna grafická knihovna pracovat.

Klíčová slova: vnitřní struktura, pokročilá barevná korekce, grafická knihovna, testování, uživatelské pluginy

Obsah

1 Úvod **1**

2 Stávající možnosti barevných korekcí fotografie a videa **2**
2.1 Analýza a návrh programu 4

3 Komponenty programu **4**
3.1 Časová osa (Timeline) 4
3.1.1 Stopa (Track) 6
3.1.2 Vrstva (Layer) 7
3.1.3 Filtr (Filter) 10
3.1.4 Parametry filtru (Filter parameters) 12
3.1.5 Parametry interakce (Interaction parameters) 12
3.1.6 Klíčový snímek (Keyframe) 13
3.1.7 Ukazatel pozice v časové ose (Playhead) 14
3.2 Historie 14
3.3 Viewport 15
3.4 Systém pyramid 16
3.5 Budoucí funkce 17

4 Funkční požadavky **18**

5 Algoritmizace funkcí systému **22**
5.1 výpočetObrazu() – pseudoalgoritmus 22

6 Grafické knihovny pro manipulaci s bitmapou **27**
6.1 VIPS 27
6.2 GEGL 29

7 PluginTester **35**
7.1 Třídy programu 35
7.2 Příprava operace 36
7.3 Plugin 37
7.4 Databáze operací 38
7.5 Vrstvy 38
7.6 Algoritmus výpočtu 40
7.7 Nový plugin 40

8 Závěr **42**

Přehled zkratek **43**

Seznam obrázků **44**

Seznam tabulek **45**

Příloha A – Identifikátory **46**

1 Úvod

V dnešní době stále více lidí vlastní digitální techniku, která dokáže věrně zaznamenat události z jejich okolí. Její úroveň je dokonce taková, že fotografie a videozáznamy pořízené při automatickém nastavení jsou k nerozpoznání od reality. Může se však stát, že záznam nebyl vyhotoven za příznivých světelných podmínek a bylo by třeba získat ze stínů detaily či nemá dostatečně teplé podzimní barvy, které je třeba přidat. Náprava takových nedostatků je možná v postprodukci.

Osoba specializující se na korekce a úpravu obrazu, zkráceně foto/video korektor, má značné znalosti z tohoto oboru. Využívá nástrojů postprodukce denně a má vysoké požadavky na rychlost odezvy změn nad obrazovými daty a rychlosti celého programu. Pro svoji činnost bude používat adekvátní softwarové vybavení, což znamená produkty od společnosti Adobe, Apple, Autodesk či Avid. Tyto produkty s dlouhou historií jsou používány profesionály na celém světě.

O programu, který zvládne úpravu fotografií i videa, pojednává tato práce. Cílem je navržení programu pro barevné korekce, který bude schopen upravovat jak fotografie, tak video. Uživateli bude stačit naučit se ovládat pouze jeden program pro úpravu fotografií i videa. Díky jeho specifickému zaměření a pokročilým funkcím v oblasti barevných korekcí bude možné provádět operace, které moderní programy nepodporují.

2 Stávající možnosti barevných korekcí fotografie a videa

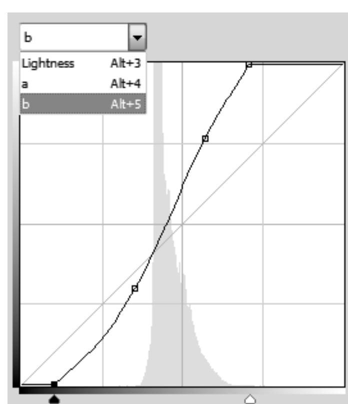
Všechny dnešní programy pro korekce obrazu lze rozdělit do dvou velkých skupin – programy pro korekci fotografií (Adobe Photoshop, Adobe Lightroom) a pro korekci videa (Iridas SpeedGrade, Autodesk Lustre). O uvedených programech se lze dočíst v [Zem10]. Mají společných několik vlastností. Jsou dosti složité a poskytují uživateli velké množství funkcí, kterými může upravovat obrazová data. Naučit se je efektivně ovládat bývá velice časově náročné a běžně může trvat i roky při každodenním používání. Nevýhodou je také vlastnost, že pokud program umí provádět skvěle korekce fotografií, nevyniká korekcí videa. To samé platí i obráceně. Z toho plyne, že pokud by chtěl korektor upravovat jak fotografie, tak video, musel by se naučit pracovat s dvěma velice odlišnými programy.

I přes jejich velké rozšíření nejsou uvedené programy dokonalé a v oblasti barevných korekcí existuje mnoho funkcí a vlastností, které postrádají. Pro dosažení požadovaného efektu může být nutné použít několik rozličných programů nebo vymyslet způsob, jak daný problém v programu obejít, což může být velice pracné. Příklady chybějících vlastností uvedu pro jeden z nejznámějších a nejrozšířenějších programů pro korekci fotografií, Adobe Photoshop.

Jedno z jeho úskalí leží v nemožnosti použití uživatelských *barevných prostorů* [Sko09, sekce 2.3]. Photoshop nám ale nedovolí použít libovolný barevný prostor. Jsou dovoleny pouze ty, které byly nainstalovány nebo vytvořeny s Photoshopem. Proto můžeme pracovat pouze s barevnými prostory RGB (anglické zkratky pro Červená, Zelená, Modrá), CMYK, Lab a odstíny šedi. Photoshop nepodporuje méně běžné barevné prostory, jako Natural Color System (NCS), HSV či YPbPr a tím přicházíme o možnost použít optimální barevný prostor pro řešení dané korekce.

Photoshop obsahuje pouze velice omezené množství nedestruktivních operací. Nedestruktivní operace je taková, že po její aplikací nejsou změněna originální obrazová data (fotografie nebo video). Uživateli je s operacemi umožněna manipulace přes *vrstvy úprav*. Např. důležitý filtr *apply image* je destruktivní. Jakmile je aplikován na obrazová data, ta jsou nenávratně změněna – nastala destrukce originálních dat, zůstala data změněná. Pro uchování originálních dat je třeba vytvářet jejich záložní kopie, což zvyšuje potřebu volné operační paměti počítače. Kromě toho nedestruktivní operace mohou být vykonávány pouze v lineárním řetězci (jedna operace po druhé) a nemohou obsahovat převody barevných prostorů mezi jednotlivými operacemi. Není proto možné používat operace v požadovaném barevném prostoru pro dosažení optimálních výsledků.

Naprosto nezbytným nástrojem pro korektora jsou křivky [Sko09, sekce 2.2.4]. Pro pracovní barevný prostor, ve kterém je obrázek otevřen, náleží každému kanálu jedna křivka. V barevném prostoru Lab můžeme nezávisle měnit křivky *L* (Lightness, Světlost), *a* a *b*. Všechny tři křivky jsou poskládány vedle sebe na obrázku 2.1.



Obrázek 2.1: Křivka *b* barevného prostoru Lab. Nezměněná křivka je úsečka, která začíná v levém dolním rohu a končí v pravém horním rohu. Manipulací křivek korektor docílí změny barevnosti obrázku.

Paleta, která je obsahuje, má pevně zvolenou velikost, kterou není možné změnit. To způsobuje komplikace při korekci křivek *a* a *b*, kde i nepatrný posun má velký dopad na změnu barevnosti obrazu. Při korekcích je běžné nepatrně měnit hodnoty křivek a hledat nejlepší kombinaci hodnot. K tomu by znatelně přispělo, kdyby mohl korektor měnit více křivek najednou. Tato vlastnost bohužel není podporována. Poslední vlastnost, kterou panel křivek postrádá, je kombinace křivek z více barevných prostorů. Nelze mít např. křivky RGB (RGB) + K křivku (CMYK). Křivky RGB by měnily barevnost, křivka K by přibarvovala černou.

Informace o barevnosti jednotlivých obrazových bodů jsou při načtení fotografie do Photoshopu zobrazeny v kanálech barevného prostoru. Běžně se provádějí korekce nad fotografií v osmibitovém barevném prostoru RGB, kde má každý obrazový bod hodnotu 0-255 (2^8). Nejtmavší místa obrazu mají hodnoty [0,0,0] (černá), nejsvětlejší [255,255,255] (bílá). Problém je ten, že nelze využívat rozšířené rozsahy barevných kanálů. Pokud bych provedl např. silné přeexponování obrazu, čímž zvýším hodnoty obrazových bodů nad hodnotu 255 a následně je stejnou mírou podexponuji (snížím hodnoty obrazových bodů na původní hodnoty), obrazové body, jejichž hodnota byla zvýšena nad 255, ztratí svoji informaci. Pokud by si program pamatoval vyšší hodnoty než 255, při podexpozici by byla data vrácena do původní podoby. Obrázky 2.2 tuto ztrátu zachycují.

(a) Originální fotografie **(b)** Přeexpozice **(c)** Podexpozice

Obrázky 2.2: Ztráta obrazových dat při provedení expozice

Není jednoduše možné vytvořit k jednomu obrázku více barevných korekcí (posloupností prováděných operací), resp. barevnou korekci nelze uložit formou *metadat*. Ta by ukládala hodnoty a parametry všech vrstev úprav a dala by se jednoduše načíst a uložit. Je přítomen i obrácený problém, tedy nelze jednu korekci použít na více fotografií. Tato nepříjemnost by dala odstranit zavedením „korekčních stylů". Styl by např. obsahoval několik přednastavených vrstev úprav, které by se jediným kliknutím myši aplikovaly na obrazová data. Takto by se snadno daly použít sady korekcí pro více fotografií.

Vrstvy úprav i zdrojové vrstvy, které obsahují fotografii, obsahují *režimy prolnutí*. Ty umožňují provádět různý typ prolínání obrazových dat dvou a více přilehlých vrstev. Uživatel si může vybrat z několika předinstalovaných režimů, ale není mu umožněno přidávat další režimy.

Jednou z důležitých vlastností každého programu je jeho uživatelská rozšiřitelnost, v případě Photoshopu použití pluginů. Plugin představuje rozšíření programu o novou funkci. Kvůli letité implementaci (první verze programu Photoshop byla vytvořena v roce 1987) nemá Photoshop dobrou podporu API. Mnoho funkcí Photoshopu proto není dostupné pro programátory. Popíšu jeden z následků tohoto omezení. Pokud by programátor vytvořil plugin pro barevnou korekci a načetl ho do Photoshopu, zobrazí se okno s náhledem dat, na které se korekce použije. V okně jsou vedle sebe zobrazeny dva obrazy: originální a po provedené korekci. Aby byl dostatečně vidět rozdíl od originálu, oba obrazy by měly být dostatečně veliké. Zastaralé API ale dovoluje pouze zobrazení náhledu o velikosti 200$\times$200 pixelů (přibližně 7$\times$7 centimetrů), na kterém není vidět všechny detaily, které aplikace filtru na obraz způsobí.

Program, řešící uvedené nedostatky, by musel sjednotit korekci fotografií i videa. Jednotné grafické uživatelské rozhraní (GUI) pro ovládání programu by zjednodušilo uživateli přestup z korekcí jednoho druhu do druhého. Zároveň musí být více zaměřený – mít menší množství více pokročilých funkcí pro barevné korekce. Uživatel pak nebude zahlcen množstvím nepotřebných funkcí a program se bude rychleji vyvíjet. Použité pluginy by fungovaly obdobně pro oboje rozhraní. Byly by vytvářeny pro použití ve videu, na

fotografii by měly efekt jako na video s jedním snímkem. Podpora uživatelsky definovaných barevných prostorů a profilů je samozřejmá.

2.1 Analýza a návrh programu

V dalších kapitolách této práce navrhnu program, který bude splňovat uvedené charakteristiky. Budu vycházet z bakalářské práce Jana Zemana [Zem10], který popisuje vzhled (GUI) programu **Dikobraz**.Ve své práci naznačuje funkci *časové osy*, která je hlavním prvkem celého programu. Na ni se v kapitole 3.1 detailně zaměřím a popíšu její části a implementaci. Kromě časové osy také popíšu, jakým způsobem bude zaznamenávána *historie* (3.2) změn provedených uživatelem. Pokud si uživatel bude chtít zobrazit výsledné korekce fotografií/videí načtených do Dikobrazu, provede tak zobrazením okna s názvem *viewport* (3.3). Když uživatel provede nad daty barevnou korekci, očekává co nejrychlejší zobrazení výsledků. To by nebylo možné při běžném způsobu ukládání výsledků algoritmu do paměti. Proto je použit *systém pyramid* (3.4), který zajistí co nejnižší rychlost odezvy Dikobrazu.

Po popisu časové osy upozorním na několik funkcí, které budou implementovány v pozdějších verzích programu. Dále zaznamenám požadavky kladené uživatelem na program, podle kterých byla navržena funkčnost časové osy. Ukážu algoritmus, kterým bude vypočítán výsledný obraz z časové osy. Po popisu implementace programu se zaměřím na optimalizaci vývoje Dikobrazu a otestuji několik grafických knihoven pro práci s obrazem. Kandidáti knihoven vhodných pro Dikobraz jsou popsáni v kapitole 6.

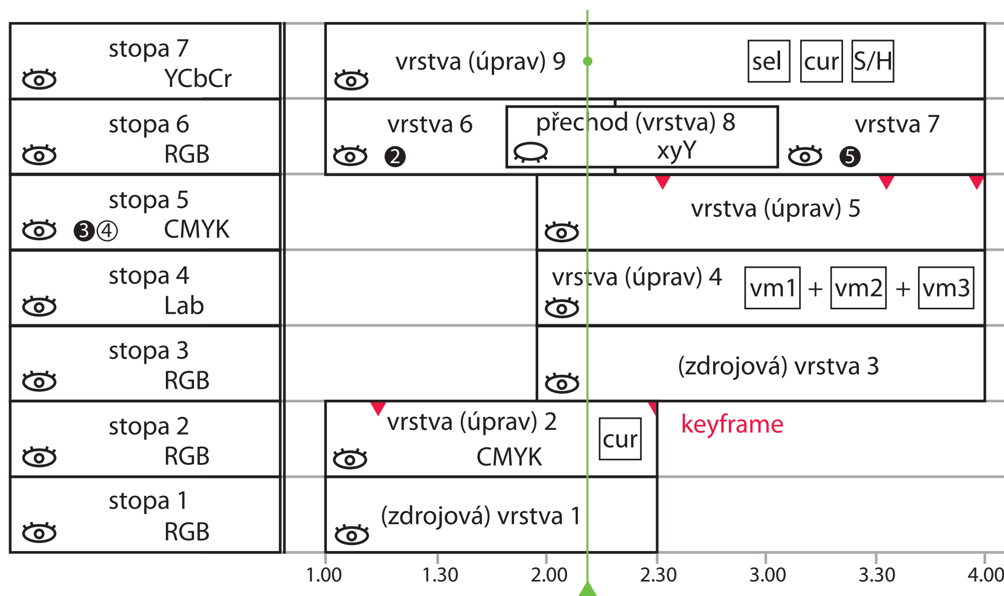
Po úvaze nad návrhem, propočtech nad zpracovávanými daty a seznámením se s vlastnostmi novodobých programů pro korekce obrazu, jsem zjistil důležitá fakta pro návrh Dikobrazu. Pro implementaci musí být použit rychlý programovací jazyk, kvůli zpracování velkých objemů dat a náročnosti korekčních metod. Pro programátorský komfort byl zvolen jazyk C++. Grafické uživatelské rozhraní bude vytvořeno úpravou komponent knihovny Qt. Ta byla vybrána pro svoji rozšířenost a platformovou nezávislost. Zároveň se zjednoduší vývoj pluginů třetích stran, jejichž GUI bude moci být vytvořeno knihovnou Qt.

3 Komponenty programu

Jako první je spuštěno **Jádro** programu, které postupně spouští jednotlivé komponenty programu. Tyto komponenty jsou časová osa, historie, pyramidový systém a viewporty. Kromě toho udržuje data potřebná pro samotný běh programu (databáze zavedených filtrů (3.1.3) – interních i externích, u kterých zároveň rozpoznává jejich typ). Pokud je požádáno časovou osou, tak provádí výpočty (5.1) nad obrazovými daty. Jeho poslední funkcí je přijímání a přeposílání požadavků příslušným komponentám. První načtenou komponentou je časová osa.

3.1 Časová osa (Timeline)

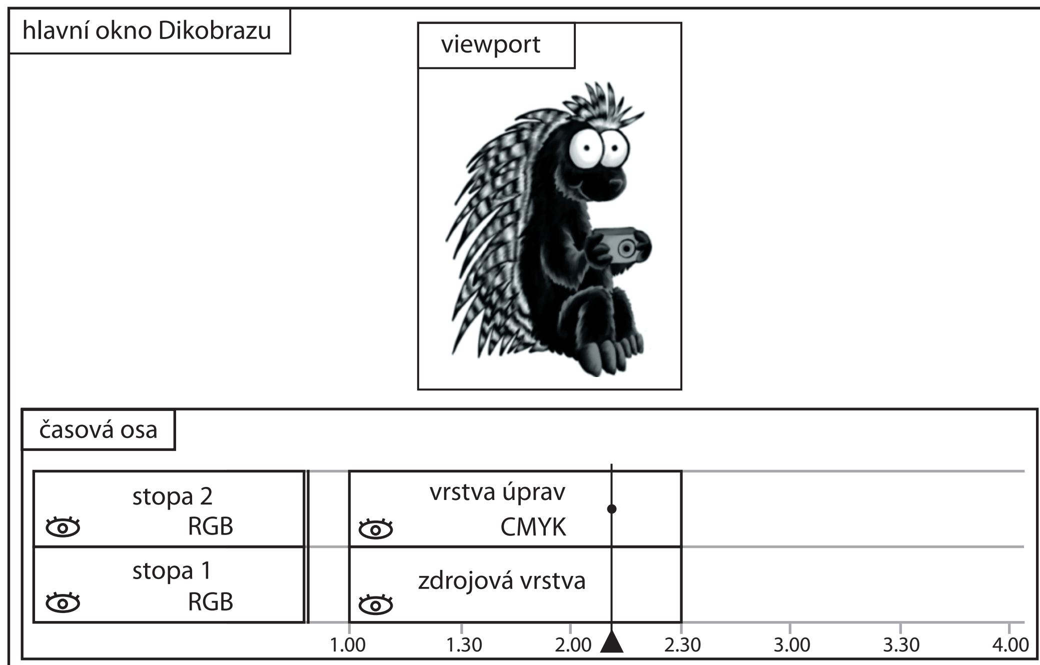
Cílem časové osy je obsáhnout funkční prvky programů pro korekci digitálních obrazů (Adobe Photoshop) a zároveň vybrat a zkombinovat ovládací prvky z oblasti programů pro zpracování videa (Autodesk Lustre, Avid MediaComposer). Dikobraz bude mít jedinou časovou osu fungující obdobně jak pro editaci fotografií, tak pro editaci videa. Následuje příklad časové osy pro editaci videa.



Obrázek 3.1: Příklad časové osy obsahující všechny komponenty

Na obrázku uživatel použil sedm *stop (tracks)* (3.1.1), první byla vytvořena stopa jedna, poslední stopa sedm. Každá stopa může obsahovat *vrstvy* (3.1.2). Stopa jedna obsahuje jedinou vrstvu videa, která začíná v první minutě a končí ve druhé minutě a třiceti vteřinách. Ve druhé stopě je vložena další vrstva, ale ne zdrojová. Tentokrát je to vrstva úprav, která obsahuje *filtr* (3.1.3), který je značený jako čtverec s textem či obrázkem uvnitř. Zde je vložen filtr křivky (cur). Vrstva úprav neobsahuje data, ale podle vložených filtrů pozměňuje data vrstev v nižších stopách. Lze říct, vrstva dva je „aplikována" na vrstvu jedna. Což znamená, data z vrstvy jedna jsou vrstvou dva změněna. Třetím druhem vrstvy je přechod (vrstva osm). Ten funguje podobně jako vrstva úprav, ale pro dvě vrstvy v jedné stopě.

Dalším prvkem je *klíčový snímek* (3.1.6). Ten je značen obráceným trojúhelníkem a je obsažen ve vrstvách dva a pět. Dokáže měnit vlastnosti vrstev úprav a přechodů v čase. Pro vrstvu dva by to mohlo znamenat, že levý klíčový snímek nastaví nízkou intenzitu změny, pravý vysokou. V čase mezi oběma klíčovými snímky bude intenzita růst. Poslední prvek v ose je *ukazatel pozice v časové ose (playhead)* (3.1.7). Je značen zeleným trojúhelníkem, z kterého vychází čára. Čára zaměřuje čas (snímek) a bod ve vrstvě devět vrstvu, ze které se uživateli zobrazí snímek do viewportu. Další podrobnosti k jednotlivým částem časové osy jsou popsány dále. Každá část obsahuje na konci popisu tabulku své datové struktury. Obrázek 3.2 ukazuje časovou osu v programu.



Obrázek 3.2: Hlavní okno aplikace Dikobraz s časovou osou a jedním viewportem.

Časová osa si udržuje pole stop, jehož uspořádání odpovídá vizuálnímu pořadí stop v časové ose a vždy musí obsahovat alespoň jednu stopu. Uživatel může tažením myši měnit pořadí stop v časové ose.

3.1.1 Stopa (Track)

Jak bylo uvedeno u příkladu, stopa představuje místo, do kterého je možné zařazovat vrstvy.

Obrázek 3.3: Příklad stopy se všemi parametry

Vlevo je obrázek oka, který reprezentuje „viditelnost" stopy. Otevřené oko znamená, že je stopa viditelná pro výpočetní algoritmus z kapitoly 5.1, zavřené oko znamená opak. Uprostřed jsou dvě číslice, které představují mechanismus uživatelsky zvolených vstupních dat do vrstev a stop. Bílá číslice v tmavém kruhu znamená, že vstup dat přichází z dané stopy. Pro tento případ bude vstup dat načítán ze stopy tři. Pokud není kruh zobrazen, jsou brána data z nejbližší nižší viditelné stopy, což by pro obrázek 3.1 znamenalo stopu čtyři. Černá číslice ve světlém kruhu označuje stopu, jejíž data poslouží jako *maska*. Ta bude popsána v kapitole 3.1.2. Pokud není kruh zobrazen, není nastavena žádná maska. Zde je použita maska ze stopy čtyři. Podrobný příklad je popsán pod obrázkem 3.6. Vpravo je název barevného prostoru, ve kterém budou probíhat výpočetní operace. V příkladu je uveden barevný prostor CMYK. Barevný prostor musí být uveden vždy.

Složka je upravená stopa a neplní její funkci. Uživatel může zařadit stopy do složky a následně složku zabalit, čímž se všechny přidané stopy skryjí a nebudou zabírat místo v časové ose. Pro rozbalení a zabalení slouží znaky '+' a '-'. Stopy, patřící do složky na sobě mají identifikátor (znak, barvu), kterou se řadí k příslušné složce. Prvky datové struktury, které využívá stopa, nejsou využity. Složky nemění postup výpočtu.

Stopa si udržuje seznam vrstev, jehož uspořádání odpovídá vizuálnímu pořadí vrstev ve stopě. Uživatel může tažením myši měnit pořadí vrstev ve stopách.

Typ	Parametry	Význam
int32	trackId	unikátní identifikátor stopy
int32	inputTrackId	identifikátor dat vstupní stopy
int32	inputMaskId	identifikátor masky vstupní stopy
bool	active	viditelnost stopy
struct	guiParams	parametry stopy pro grafického uživatelského rozhraní
int32	guiHeight	výška stopy
struct	guiColor	barva stopy
string	name	pojmenování stopy
struct	interactionParams	parametry spojené s prolínáním stop, viz sekce 3.1.5
int32	colorspace	identifikátor použitého barevného prostoru
bool	folder	zda má stopa funkci stopy, či složky
Layer[]	layers	seznam vrstev pro jednu stopu

Tabulka 3.1: Datová struktura stopy

3.1.2 Vrstva (Layer)

Existují tři typy vrstev: **zdrojová vrstva** (source layer), **vrstva úprav** (layer) a **přechod** (transition). Mají stejnou sadu parametrů a jejich typ je rozlišen příznakem. Společně budou označovány jako *vrstvy* (*layers*). Je třeba jejich kombinace, aby bylo docíleno požadované barevné korekce.

Zdrojová vrstva se od ostatních liší tím, že dokáže načítat fotografie/video. Na načtená data je následně možné aplikovat další vrstvy a tím měnit původní obraz. Do zdrojové vrstvy nelze přidat klíčové snímky či filtry. Uživatel si také může vytvořit zdrojovou vrstvu, která bude jednolitě vyplněna barvou. Taková vrstva bude v Dikobrazu použita jako *nultá stopa*. Nultá stopa obsahuje nultou vrstvu, která sahá od začátku nulté stopy, až do jejího konce. Pokud by uživatel posunul ukazatel pozice v časové ose na místo, kde by nebyla žádná vrstva, budou mu do viewportu zobrazena data nulté vrstvy, neboli jednobarevná plocha.

Vrstva úprav, s příkladem na obrázku 3.4, má za úkol měnit data jiných vrstev. Znaky pro viditelnost, vstup dat a barevný prostor fungují stejně jako u stopy. Navíc obsahuje klíčové snímky a tři filtry: selektivní výběr barev (sel), křivky (cur), shadow/highlight (S/H). Jak už bylo uvedeno v kapitole 3.1, filtr určuje, jak budou změněna data nižších vrstev. Ve vrstvě je zobrazen jako obdélník, ve kterém je ikonou či slovy označen druh filtru. V okamžiku, kdy algoritmus dojde k této vrstvě, jsou zleva po sobě aplikovány všechny filtry na vstupní data. Ty přijdou v tomto případě ze stopy dva. Aby bylo možné řídit efekt filtru v závislosti na čase, je možné do vrstvy přidávat klíčové snímky. Každý z nich určuje „bod změny", ve kterém je možné měnit *parametry filtru (filter parameters)* (3.1.4).

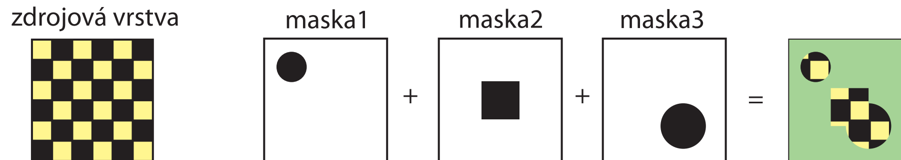
Obrázek 3.4: Příklad vrstvy úprav se všemi parametry

Zdrojová vrstva a vrstva úprav mají při vytvoření jeden skrytý klíčový snímek a nelze je v jedné stopě navzájem překrýt.

Vrstvy úprav můžou využít **masku**. Tu si lze představit jako plátno, které položím na fotografii. Fotografie je celá zakrytá a není z ní nic vidět. Poté v něm na určitých místech vyříznu díry. Těmito děrami jsou vidět části fotografie. Lze také použít inverzní masku, ve které na fotografii pokládáme části plátna a tím fotografii zakrýváme. Maska se používá tehdy, když chceme použít úpravu pouze na část obrazu. Toho často využívají vrstvy úprav. Existují dva druhy masek. *Vektorová*, vytvořená v Dikobrazu, která se do vrstvy vkládá jako filtr, nebo *bitmapová*, která je externě načítána jako zdrojová vrstva. Vektorové masky

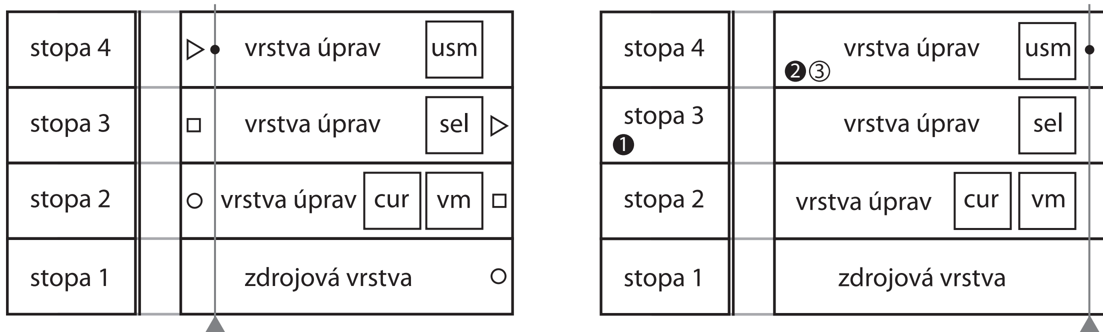
je možné ve vrstvě kombinovat. Pokud by bylo v jedné vrstvě přidáno více filtrů vektorových masek a jejich režim prolnutí [Mar07, sekce 8] je různý od režimu „normální", bude vytvořeno jejich sjednocení, průnik, atd.

Rozdíl mezi vektorovou a bitmapovou maskou je ve formátu dat. Vektorová maska tvoří většinou jednoduché obrazce, jako čtverec, kruh. V programu se ukládají pouze její parametry. Pro čtverec by to znamenalo: výška, šířka, pozice nad fotografií. Její použití může být pro fotografie a hlavně pro video. V jednotlivých snímcích je možné parametry vektorové masky měnit a plynule ji přesouvat s obrazem. Druhý typ, bitmapová maska, je šedotónový obrázek, kde je podle intenzity šedi nastavena průhlednost. Tato maska se používá ke korekci fotografií, kde je možné nastavit vlastní hodnotu masky pro každý obrazový bod fotografie. Následující příklad ukazuje kombinaci tří masek.



Obrázek 3.5: Příklad kombinace masek

Již na úvodním obrázku u časové osy (obr. 3.1) a u stopy (obr. 3.3) bylo nastíněno načítání dat z uživatelsky zadaných stop. Nyní se na tuto vlastnost zaměřím podrobněji (obrázek 3.6). Všechny vrstvy jsou zde nastaveny jako viditelné.



(a) Implicitní postup výpočtu obrazu. Vstupní / výstupní značky (kružnice, čtverec, trojúhelník) jsou pouze ilustrační, v programu nebudou vidět. Z vrstvy s kružnicí na konci jsou data poslána do vrstvy s kružnicí na počátku. Tento postup platí obdobně i pro ostatní značky.

(b) Uživatelsky změněný postup výpočtu obrazu. Vstupy do vyšších vrstev / stop jsou nastaveny z různých stop. V černém kruhu je označen vstup dat, v bílém maska. Pokud ve vrstvě není nastaven uživatelský vstup dat, funguje poslání dat implicitně.

Obrázek 3.6: Příklady datových vstupů a výstupů vrstev

Základ je stejný pro obě části obrázku. Zdrojová vrstva pouze načte snímek ze vstupních dat. V případě videa to znamená, že celá videodata nejsou načtena do paměti, pouze zobrazené (použité) snímky se čtou z disku. Fotografie je načtena celá. Vyberu si libovolný snímek z intervalu mezi začátkem a koncem zdrojové vrstvy (výběr označen šedým ukazatelem pozice v časové ose). Data jsou v jednotlivých vrstvách zpracovávána zleva doprava, posílána na nejbližší viditelnou vrstvu.

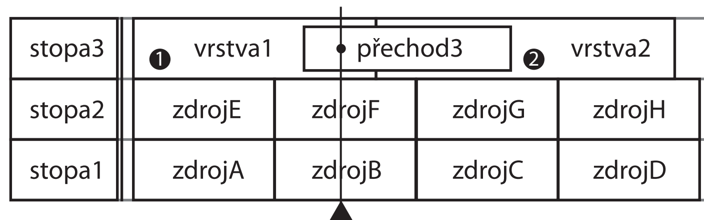
Na obrázku 3.6a jsou data načtená ve vrstvě jedna poslána do vrstvy dva, kde je na část obrazu aplikována vektorová maska (vm). Na takto označená data jsou aplikovány křivky (cur). Předtím, než budou změněná data z druhé vrstvy poslána do vrstvy třetí, jsou na konci vrstvy převedeny na bitmapu. Obdobný postup platí i pro další dvě vrstvy. Ve třetí vrstvě je aplikován filtr selektivní barva (sel). Nakonec ve čtvrté vrstvě jsou obrazová data zaostřena (usm) a zobrazena uživateli.

Na pravém obrázku uživatel změnil pořadí vykonávání operací přiřazením jiných vstupů do vrstev a stop. Vstupem může být pouze stopa na nižší pozici, čímž je zajištěna acykličnost přiřazování. Stejně jako na

obrázku 3.6a jsou data načtena a poslána do vrstvy dva. Místo jejich zpracování třetí vrstvou budou použita data načteného snímku ze stopy jedna. Čtvrtá vrstva použije data ze stopy dva a jako masku data ze stopy tři. Stopy a vrstvy mohou mít do svého vstupu přiřazenu nejvýše jednu stopu pro data (černý kruh s bílou číslicí) a nejvýše jednu stopu jako masku (bílý kruh s černou číslicí).

Posledním druhem vrstvy je přechod. Ten se používá mezi dvěma vrstvami například pro plynulou změnu obrazu mezi datovými vrstvami či pro aplikaci speciálního efektu v čase. Příkladem takového efektu je wipe-in, který způsobí postupné překrytí první vrstvy druhou vrstvou (vrstva „přijede" ze strany). Přechod používá *parametry interakce (interaction parameters)* (3.1.5) a klíčové snímky. Má dva viditelné klíčové snímky po jednom na každém konci a může překlenout nejvýše dvě vrstvy. Povolené kombinace jsou tedy vrstva-vrstva, vrstva-prázdno a prázdno-vrstva a také prázdno-prázdno. Kombinace vrstva-prázdno-vrstva nejsou pro jeden přechod povoleny. Stejně jako vrstva úprav může obsahovat masku.

Obrázek 3.7 znázorňuje použití přechodu k **tvorbě sekvencí**. Stopy jedna a dvě obsahují po čtyřech navzájem různé zdrojové vrstvy. Třetí stopa obsahuje dvě vrstvy úprav a přechod mezi nimi. Chci zobrazit snímek označený ukazatelem pozice v třetí stopě časové osy, který leží na vrstvě jedna a přechodu.



Obrázek 3.7: Př. Tvorba sekvencí

Výsledná kompozice obrazu bude skládána z několika částí. Jelikož je ukazatel pozice nastaven na přechod, ten si zjistí, jaké vrstvy překrývá a vypočítá si pro ně data. Ty potřebuje, aby na ně mohl aplikovat svoje úpravy. Nejprve se pro vrstvu jedna načtou její vstupní data ze zdrojové vrstvy B. Data první vrstvy jsou již spočítána, ještě je třeba spočítat vrstvu dva. Ta načítá data ze stopy dva, zdrojové vrstvy F. Po jejich načtení má již potřebná data pro provedení průniku dat. Výsledná data jsou zobrazena.

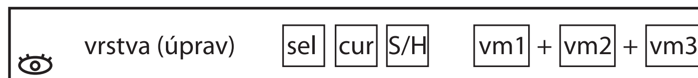
Pro všechny vrstvy platí, že stejně jako stopa, zpracovávají data v určitém barevném prostoru. Při vytvoření nové vrstvy není její barevný prostor určen a při výpočtech se přebírá barevný prostor stopy. Je možné nastavit barevný prostor explicitně a tím lokálně přepsat barevný prostor stopy.

Typ	Parametry	Význam
int32	layerId	unikátní identifikátor vrstvy
int32	trackId	identifikátor stopy, ve které je vrstva umístěna
byte	layerType	identifikátor, zda je vrstva vytvořena jako vrstva úprav, zdrojová vrstva, či přechod
int32	inputTrackId	identifikátor stopy, ze které budou načítána data. Vrstva může tento parametr pro své potřeby předefinovat
int32	inputMaskId	identifikátor stopy, ze které bude načítána maska. Vrstva může tento parametr pro své potřeby předefinovat
int32	startFrame	počáteční snímek vybrané viditelné části videa
int32	endFrame	koncový snímek vybrané viditelné části videa
bool	active	viditelnost vrstvy
struct	interactionParams	parametry spojené s prolínáním vrstev, viz sekce 3.1.5
int32	colorSpace	barevný prostor vrstvy
Filter[]	filters	seznam filtrů použitých ve vrstvě. Je řazen podle visuálního pořadí filtrů ve vrstvě. Vlevo jsou umístěny korekční filtry a filtry pro úpravu obrazu, vpravo jsou ukládány masky
int32[]	filtersTypes	seznam typů filtrů použitých ve vrstvě, je řazen stejně jako seznam filtrů. Každý filtr ze seznamu *filters* má identifikátor, který určuje jeho typ. Další informace jsou v kapitole 3.1.3
Keyframe[]	keyframes	seznam klíčových snímků. Pokud je vrstva zdrojová nebo úprav, je zde nejméně jeden klíčový snímek. V případě přechodu jsou zde nejméně dva klíčové snímky
struct	guiParams	parametry grafického uživatelského rozhraní
struct	guiColor	barva vrstvy
string	name	pojmenování vrstvy
int32	projectPositionX	horizontální souřadnice levého horního rohu vrstvy na pracovní ploše (4.2). Počátek souřadnicového systému je v levém horním rohu pracovní plochy a je uveden v pixelech
int32	projectPositionY	vertikální souřadnice levého horního rohu vrstvy na pracovní ploše. Počátek souřadnicového systému je v levém horním rohu pracovní plochy a je uveden v pixelech
string	source	úplná cesta k souboru
int32	sourceIn	počáteční snímek použitelných záběrů vybraného videa
int32	sourceOut	koncový snímek použitelných záběrů vybraného videa
struct	sourceDecodeParams	parametry dekomprese

Tabulka 3.2: Datová struktura vrstvy

3.1.3 Filtr (Filter)

Filtr představuje způsob, kterým lze provést korekci obrazových dat. Je ve vrstvě zobrazen obdélníkem, v němž je textem nebo obrázkem vyznačena operace filtru. V ukázce 3.8 jsou tyto filtry: selektivní výběr barev (sel), křivky (cur), shadow/highlight (S/H) a kombinace tří vektorových masek (vektorová maska je také filtr). Jejich aplikace na obrazová data je postupná. Pokud by měla být vrstva z ukázky spočtena, bude nejprve vytvořena vektorová maska složená ze tří částí. Poté bude aplikován filtr selektivní výběr na data vymezená maskou, na takto modifikovaná data jsou aplikovány křivky a nakonec shadow/highlight, oboje také v prostoru masky.

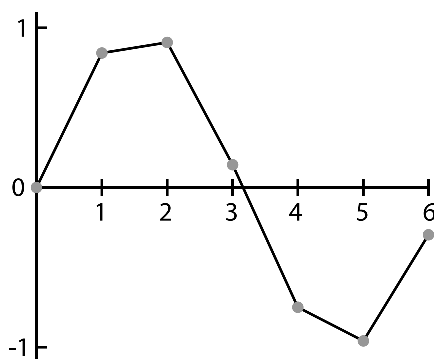


Obrázek 3.8: Ukázka vrstvy úprav obsahující filtry

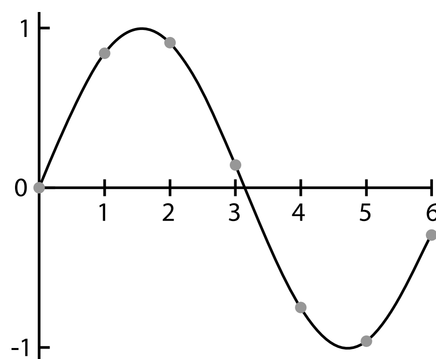
Filtrům, které jsou vytvořeny při instalaci programu a uživatel je pouze využívá, se nazývají *vnitřní filtry*. Druhým typem filtrů jsou *uživatelské filtry*. Vytváří je uživatel a lze je do Dikobrazu načítat dynamicky, za běhu programu. Při načtení filtru do programu je mu přidělen unikátní identifikátor, který představuje jeho typ. Všechny filtry typu křivky mají přidělený stejný identifikátor typu. To samé platí i pro filtry zaostření, selektivní barva nebo vektorová maska, které jsou navzájem funkčností velice odlišné. Typ je jediný rozpoznávací prvek pro odlišení filtrů. Když uživatel vytvoří novou vrstvu, do které přidá filtry různých typů, budou všechny filtry uloženy v jediném poli filtrů. Aby Dikobraz dokázal rozlišit, s jakým filtrem pracuje, zeptá se na jeho typ. Filtr mu tento identifikátor sdělí a program ví, jak s ním dále zacházet.

Každý filtr se skládá ze dvou částí. První část obsahuje obecné parametry pro daný typ filtru: globální parametry filtru pro vrstvu a *interpolátory* (popsány dále). Druhá část obsahuje vlastní data filtru a parametry interakce (jak se filtry navzájem ovlivňují).

Každý filtr může obsahovat **interpolátory**. Interpolátor určuje způsob, kterým se budou měnit hodnoty parametry filtru v průběhu celé vrstvy. Není závislý na klíčových snímcích. Má smysl ho používat pouze pro editaci videa, kde je přítomný čas. Nejběžnějším příkladem jsou po částech lineární interpolace a po částech kubická interpolace. Ukázka obou příkladů je na obrázcích 3.9.



(a) Po částech lineární interpolace. Je nejednodušší používaná. Mezi dvěmi sousedícími body je vedena úsečka



(b) Po částech kubická interpolace. Mezi každými dvěma body je snaha vytvořit takovou část, aby na sebe jednotlivé části plynule navazovali

Obrázky 3.9: Druhy interpolací

Mezi další druhy interpolátorů patří interpolace skalární hodnoty a interpolace úhlové hodnoty. Rozdíl mezi nimi ukážu na příkladu. Mám dvě celočíselné hodnoty, 10 a 350. Při skalární interpolaci budou mezihodnoty 11, 12, 13, …, 348, 349, tedy bude lineárně růst až do požadované hodnoty. Celkem získám 338 mezihodnot. Pro úhlovou interpolaci budou hodnoty obíhat po kružnici: 9, 8, …, 0, 359, 358, ..., 351. Čímž je pokryto pouze 19 mezihodnot. Tvůrce filtru si může vybrat interpolátory pro jeden parametr, či pro všechny. Všechny modely interpolátorů (lineární, exponenciální, pilovitý, ad.) budou vytvořeny ve variantách pro jeden, dva a více parametrů.

Nástroje pro **úpravy obrazu** budou realizovány filtry. První skupina jsou nekorekční nástroje: Měřítko, Otočit, Zkosit, Zdeformovat, Perspektiva, Warp, Zrcadlové převrácení. Speciální filtr *ořez* vrstvy úprav bude koncipován jako vektorová maska. Zdrojová vrstva a přechod ořezový filtr nemají. Do druhé skupiny patří nástroje pro barevnou korekci: Křivky, Selektivní výběr barvy, Světlost/Kontrast, Mixér kanálů, Ostření, Použít obraz (Apply image), Kalkulace (Calculations). Změny parametrů filtru budou ukládány do historie (3.2).

Typ	Parametry	Význam
int32	filterId	unikátní identifikátor filtru
int32	layerId	identifikátor vrstvy, v které je filtr instalován
int32	filterType	druh filtru, každému filtru je při inicializaci přiděleno vlastní číslo
bool	active	zda je funkce filtru zapnuta
struct	filterParams	parametry filtru platící globálně v rodičovské vrstvě (ne v klíčových snímcích)
struct	interpolators	interpolátory pro daný filtr

Tabulka 3.3: Datová struktura filtru

3.1.4 Parametry filtru (Filter parameters)

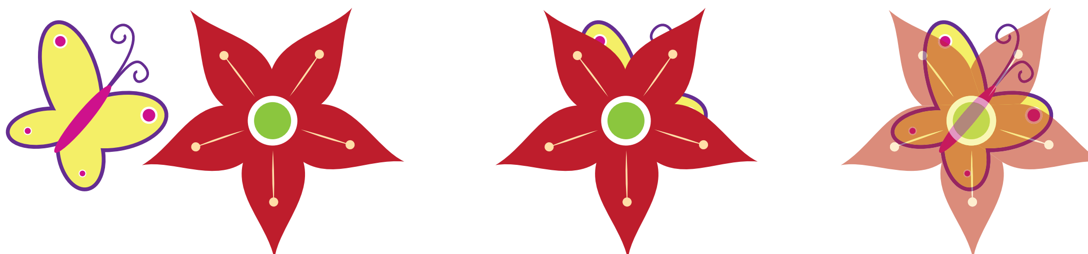
Jsou odděleny od těla filtru záměrně. Mají dvojí využití. Každá vrstva, která obsahuje alespoň jeden filtr, získává instanci (vlastní množinu) parametrů filtru, ve které jsou *globální* hodnoty pro všechny filtry. Globální hodnoty jsou sdílené všemi filtry. Změnou globálního parametru u jednoho filtru se změní onen parametr u všech filtrů. Druhé využití je jako *lokální* parametry filtru. Každý filtr obsahuje vlastní data. Změnou některého z parametrů zůstávají parametry jiných filtrů zachovány.

Typ	Parametry	Význam
int32	filterId	identifikátor filtru, ke kterému tyto parametry patří
int32	filterParamsId	unikátní identifikátor parametrů filtru
struct	interactionParams	parametry interakce filtru
…	…	vlastní parametry filtru definované uživatelem

Tabulka 3.4: Datová struktura parametru filtru

3.1.5 Parametry interakce (Interaction parameters)

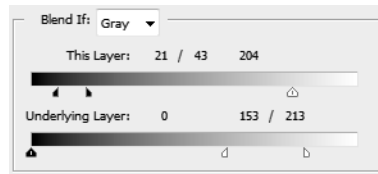
Obsahují informace pro chování při styku více objektů (např. průhlednost pro styk vrstva-vrstva a klíčový snímek-vrstva, režim prolnutí u filter-filter, …). Všechny hlavní prvky programu mají vlastní instanci. Uvedu příklad pro režim prolnutí vrstva-vrstva. Obrázek 3.10 ukazuje prolnutí dvou zdrojových vrstev. Vlevo je první vrstvou motýl, který letí na květinu z druhé vrstvy. Uprostřed motýl dosedl na květinu, ale není vidět, jelikož se díváme na květinu zespodu. Aby byl motýl viditelný, je na pravé straně snížena viditelnost květiny na 50 %. Motýl se tak zviditelnil.



Obrázek 3.10: Režim prolnutí dvou vrstev s nastavením průhlednosti

Dalším parametrem interakce může být míchání vrstev, metoda pro korekci obrazu. Pokud se dvě a více vrstev vzájemně kryje, může uživatel řídit jejich prolnutí použitím blend-if (smíchej-když) posuvníků (obr. 3.11). Vrchní posuvník nastavuje míchání pro aktuálně vybranou vrstvu, spodní posuvník pro všechny

vrstvy pod aktuální vrstvou. Čím víc posunu ukazatel v horním posuvníku, tím víc mi zmizí aktuální vrstva. Pokud budu měnit ukazatel pro spodní posuvník, tím více mi budou „prosvítat" data spodních vrstev skrz aktuální vrstvu. Příklad využití lze nalézt v [Mar07, sekce 17].



Obrázek 3.11: Ukázka blend-if posuvníků pro odstíny šedi z aplikace Adobe Photoshop

Tabulka níže obsahuje pouze reprezentativní vzorek hodnot. Další hodnoty budou doplněny ve fázi implementace.

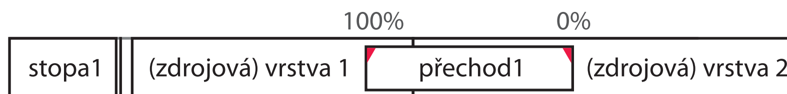
Typ	Parametry	Význam
int32	opacity	neprůhlednost komponenty. Pokud je hodnota nastavena na nula, je úplně průhledná, pokud je hodnota nastavena na sto, je úplně neprůhledná
int32	mode	režim prolínání komponenty. Může být například: Vynásob, Zesvětli, Ztmav, Překryj. Také může obsahovat uživatelsky definované režimy prolínání
struct	blend-if-params	parametry pro míchání dvou vrstev
struct	blendComp	obsahuje šest prvků: • Barevný prostor, • použitý kanál z barevného prostoru, • dvojice celočíselných hodnot pro levý ukazatel posuvníku, • dvojice celočíselných hodnot pro pravý ukazatel posuvníku.
blendComp[]	actualLayer	seznam komponent k míchání pro tuto vrstvu
blendComp[]	lowerLayers	seznam komponent k míchání pro spodní vrstvu
int32	colorSpace	barevný prostor komponenty

Tabulka 3.5: Datová struktura parametrů interakce

3.1.6 Klíčový snímek (Keyframe)

V klíčovém snímku jsou uloženy lokální parametry filtru. Pro vrstvu se třemi filtry to znamená, že každý přidaný klíčový snímek může nezávisle měnit hodnoty parametrů jednotlivých filtrů. Při vytvoření vrstvy je automaticky vytvořen jeden klíčový snímek. Ten není viditelný, protože v přítomnosti pouze jednoho klíčového snímku ve vrstvě není možné měnit parametry filtru v čase a interpolátory neúčinkují. Když není klíčový snímek viditelný, jsou nastavovány parametry filtrů ve vlastnostech vrstvy. Klíčový snímek je v GUI zobrazen jako trojúhelník směřující dolů. Každý typ vrstvy využije jinou část klíčového snímku. Zdrojová vrstva využívá pouze parametry interakce. Vrstva úprav a přechod využije jak parametry interakce, tak parametry filtrů. Pokud si uživatel bude přát přidat další klíčový snímek, přidá ho na zvolený snímek vrstvy a skrytý klíčový snímek se zviditelní. Klíčové snímky není možné přesouvat z jedné vrstvy na druhou.

Jiné využití může mít klíčový snímek ve vrstvě úprav, která obsahuje několik filtrů. Uvedu příklad na obrázku 3.4. Pokud si zobrazím data v levém klíčovém snímku, uvidím všechny parametry tří filtrů (vektorové masky nezahrnuji) a mohu je libovolně změnit. Odlišné nastavení mohu použít pro druhý klíčový snímek. Při zapnutém interpolátoru se budou hodnoty parametrů mezi klíčovými snímky v čase měnit a tím se bude měnit i výsledný obraz barevné korekce. Obrázek 3.12 ukazuje použití klíčových snímeků.



Obrázek 3.12: Použití klíčových snímků pro změnu viditelnosti přechodu v čase

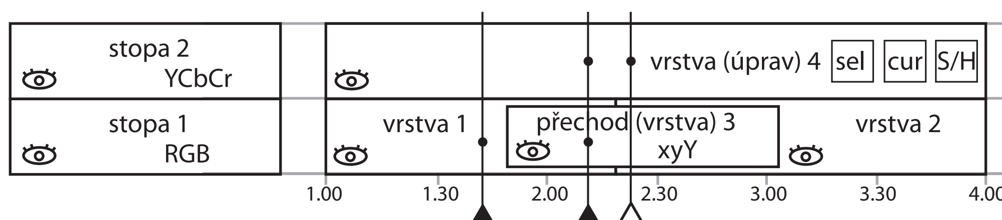
Procenta, které jsem si pro příklad zvolil, znázorňují viditelnost efektu použitého v přechodu. Na počátku přechodu (v levém klíčovém snímku) je nastavena viditelnost na 100%, což znamená, že aplikovaný efekt má plnou intenzitu. Na konci (v pravém klíčovém snímku) je efekt úplně průhledný a není viditelný. V průběhu vrstvy efekt slábne podle použitého interpolátoru. V případě lineárního interpolátoru bude v polovině vrstvy efekt pouze 50%.

Typ	Parametry	Význam
int32	keyframeId	unikátní identifikátor klíčového snímku
int32	layerId	identifikátor vrstvy, ke které je klíčový snímek přiřazen
int32	frame	snímek, na kterém je klíčový snímek umístěn
struct	interactionParams	Působí mezi daty a filtry. Příklad: Vrstva má jeden filtr, který je nastaven do režimu luminosity.
struct	filterParams[]	lokální parametry jednotlivých filtrů

Tabulka 3.6: Datová struktura klíčového snímku

3.1.7 Ukazatel pozice v časové ose (Playhead)

Ukazatel obsahuje dva prvky. Číslo snímku (framu), na který ukazuje, a odkaz na datovou pyramidu (3.4). V GUI je zobrazen jako pruh vycházející z trojúhelníku. Pozice, kde je ukazatel umístěn, označuje vybraný snímek. Černý bod, vložený na určitou vrstvu, označuje z jaké vrstvy budou zobrazena data. Číslo snímku a bod ve vrstvě určují, odkud bude viewport (sekce 3.3) čerpat obrazová data. Na obrázku 3.13 jsou zobrazeny tři ukazatele se čtyřmi body. Každý ukazatel může obsahovat více bodů, pro každou vrstvu kterou prochází jeden. Každý černý bod značí jeden viewport, který si může uživatel zobrazit. Pravý ukazatel má nevyplněný trojúhelník, čímž je označen jako aktivní (hlavní). Všechny akce, jako přidání filtru či klíčového snímku budou s aktivním ukazatelem. Ukazatel je vytvořen spolu s viewportem, který obsahuje jeho unikátní identifikátor.



Obrázek 3.13: Více ukazatelů zobrazujících několik různých míst časové osy

3.2 Historie

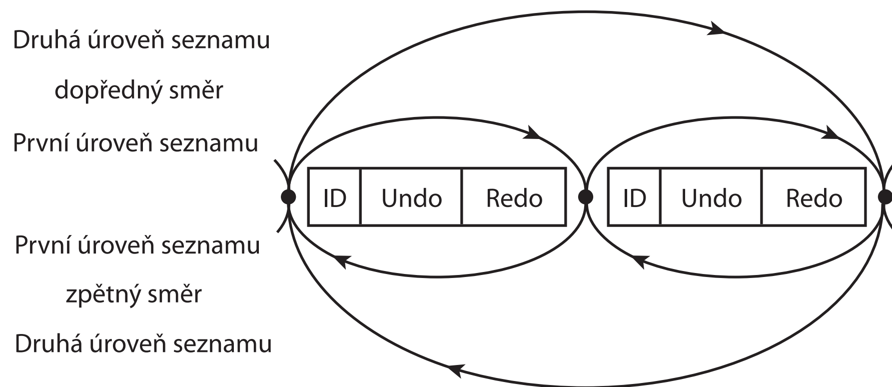
Komponenta pro uchování historie změn provedených při editaci fotografií/videa bude ukládat změny parametrů časové osy a jejích komponent. Každá změna parametru filtru, posunu vrstvy ve stopě, odebrání filtru a mnoho dalšího bude zaznamenáno do historie. Jakmile uživatel provede změnu, ta bude uložena do historie a uživateli se zpřístupní tlačítko *zpět* (undo). Pokud ho stiskne, dostane se do stavu před provedením změny. Zpět, do stavu po změně, se odtud dostane tlačítkem *dopředu* (redo). Pro lepší

kontrolu a snadnější správu jednotlivých bloků historie bude vyčleněn vlastní ucelený prostor v paměti, do kterého budou data uložena. Paměť bude plněna *příkazy*, z nichž mají všechny jednotnou strukturu:

- **ID** (Identifikátor)
- **Undo** (Parametry pro přechod k minulému nastavení)
- **Redo** (Parametry pro přechod k budoucímu nastavení)

Identifikátor je celočíselného typu a obsahuje všechny možné kódové kombinace akcí, které je možné v programu provést. Výpis možných identifikátorů je v Příloze A. Tento seznam bude rozšířen na základě implementačních požadavků.

Části Undo/Redo mají stejnou velikost. V části Undo jsou uloženy parametry filtru/…/vrstvy před změnou, v části Redo po změně. Pohyb mezi příkazy zprostředkovává obousměrný spojový seznam. Při přidání příkazu do historie (přidává se na konec) se vytvoří odkaz první úrovně oběma směry. Pokud by uživatel vracel změny, pohyboval by se po seznamu směrem doleva (k počátku). Obdobně i pro vrácením změn, směr posunu je doprava (na konec). Pokud by uživatel provedl akci, která by vyžadovala uložení více příkazů do historie naráz, budou všechny části přemostěny spojem druhé úrovně. Kdyby se chtěl uživatel vrátit zpět, budou příkazy prováděny zpětně do té doby, dokud neskončí na druhé straně mostu. Příkazy generuje GUI, jádro je zpracovává a posílá/načítá do/z Historie. Schéma 3.14 znázorňuje koncept Historie:



Obrázek 3.14: Koncept historie

Uvedu příklad na použití historie. Časová osa obsahuje vrstvu úprav s jedním filtrem. Uživatel se rozhodne vrstvu smazat. To znamená provést dva zápisy do historie v pořadí: smazání filtru a poté smazání vrstvy. Pro smazání filtru se vytvoří nový blok historie, který bude mít v části Undo uloženy parametry filtru, v části Redo informaci o smazání filtru. Obdobný blok je vytvořen pro vrstvu. Bloky jsou do historie připojeny doprava za již existující bloky. Po smazání vypadá historie jako na obrázku 3.14. Pokud by uživatel nyní stiskl tlačítko Zpět, zjistí algoritmus, že existuje most druhé úrovně a bude třeba provádět operace Undo do té doby, dokud algoritmus nedojde na začátek mostu druhé úrovně. Což znamená, že nejprve se z pravého bloku paměti provede obnovení vrstvy z části Undo, následně je z levého bloku obnoven i filtr. Algoritmus obnovování dojde na začátek mostu druhé úrovně a končí.

3.3 Viewport

Obrazová data, nad kterými byly provedeny dané úpravy, jsou zobrazeny na monitor uživateli. Místo, do kterého jsou zobrazena, se nazývá viewport. Těch může být otevřeno i více vedle sebe na jednom monitoru či mohou být rozmístěny na více monitorech. Důvod pro použití více viewportů může být např. sledování originálních obrazových dat v jednom viewportu a změněných dat v druhém.

Vstupem do viewportu je jediný obraz ve svém barevném prostoru. Pokud by obraz neměl barevný prostor přidělený, bude uživateli nabídnuta možnost přiřadit barevný prostor podle jeho volby, nebo

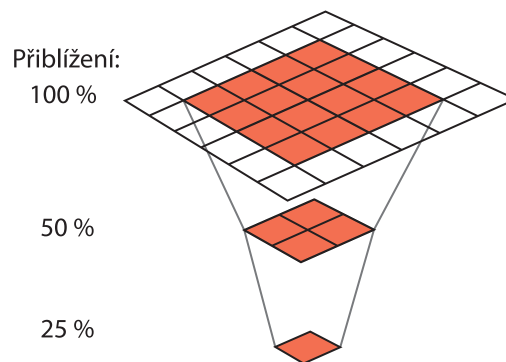
pracovat bez přidělení prostoru. Viewporty musí umět převádět mezi barevnými prostory monitoru a obrazových dat. Jak už bylo zmíněno, video či fotografie může být upravováno v různých barevných prostorech. Aby bylo možné zobrazit tato data uživateli na monitor, musí být nakonec převedeny ze svého barevného prostoru do barevného prostoru monitoru, který může být odlišný od prostoru dat. Uživatel si může nastavit i svůj barevný prostor monitoru, ve kterém by chtěl obraz vidět.

Také spravuje přiblížení. Viewport ukládá informace o použitém rozlišení, z jaké stopy jakého ukazatele pozice v časové ose je obraz zobrazen. Zobrazená část obrázku nemusí být celá, a proto jsou uloženy i souřadnice potřebné pro její zaměření. Pokud by byla data načtená do viewportu větší než jeho aktuální velikost, bude provedeno **oddálení obrazu** potřebné pro jeho celé zobrazení ve viewportu (např. 27,35 %) a použito nejbližší nižší přiblížení (pro 27,35 % to bude 25 %). Obraz bude zobrazen ve 25% přiblížení. Pokud je velikost načteného obrazu stejná nebo menší než aktuální velikost viewportu, bude zobrazen nezmenšený v celé velikosti.

3.4 Systém pyramid

Změny parametrů filtru či posun ukazatele pozice jsou při korekci snímku prováděny velice často. Proto je třeba, aby byly co nejrychleji zobrazeny uživateli do viewportu. Kromě toho korektor běžně používá funkci přiblížení/oddálení obrazu ve viewportu. Ve vysokém přiblížení jsou upravovány detaily snímku, v nízkém přiblížení se upravuje celkový vzhled snímku. Korektor mění přiblížení často a pokud by musel být snímek počítán znovu pro každou změnu přiblížení, stala by se tato operace příliš časově náročnou, tudíž nepoužitelnou. Aby byla obě kritéria splněna, je zaveden způsob ukládání dat do pyramid.

Název je odvozen od podobnosti s obrácenou pyramidou. Vrchní patro pyramidy uchovává obrazová data ve 100% přiblížení, nižší patro má pouze 50% přiblížení, což znamená pouze polovinu velikosti vrchního patra. Tímto způsobem lze zmenšovat rozlišení, dokud jsou k dispozici obrazová data. Podrobnosti o pyramidovém systému a použití při oddálení obrazu ve viewportu lze nalézt v [Sko09, sekce 3.1.3]. Následující obrázek ukazuje třípatrovou pyramidu. Jednotlivá patra mohou obsahovat data obrázku pro dané přiblížení. Pokud uživatel pracuje pouze v jednom patře pyramidy a ostatní nepoužije, nebudou ani ty naplněny daty.



Obrázek 3.15: Princip pyramidového modelu. Pokud prostřední patro pyramidy má mít pouze 50% přiblížení, je třeba snížit velikost dat z vrchního parta pyramidy. Snížení se provede netriviálním zprůměrováním hodnot čtyř přilehlých pixelů (obrazových bodů). 25% vrstva průměruje již 16 pixelů z nejvrchnějšího patra.

Uživatel zaměří ukazatelem pozice a bodu ve vrstvě snímek, který chce zobrazit do viewportu. Program spustí algoritmus 5.1, který naplní pyramidy z obrázku 3.16 daty. Plnění probíhá od zaměřené vrstvy směrem dolů. Obrazová data jsou do viewportu načtena z příslušné pyramidy ve chvíli, kdy je algoritmem naplněna. Pyramidy pro levý (aktivní) ukazatel jsou naplněny a data z pyramidy ve stopě tři jsou zobrazena ve viewportu. Pokud by nyní uživatel změnil parametry filtru selektivní barvy (sel) ve vrstvě

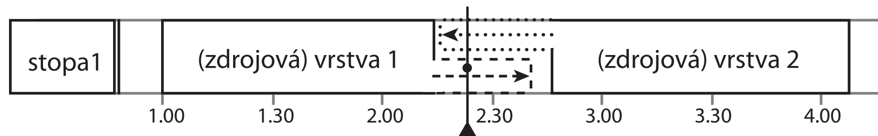
úprav dva, budou data z pyramidy ve třetí stopě *zneplatněna* a bude je třeba obnovit. Algoritmus se podívá do nižší stopy a zjistí, že ta obsahuje pyramidu s *platnými* daty. Použije tato data a po aplikaci parametrů filtru je uloží do pyramidy ve třetí stopě. Tato nová data jsou již platná a budou zobrazena uživateli do viewportu.

Obrázek 3.16: Použití pyramid v časové ose. Ta obsahuje zdrojovou vrstvu a dvě vrstvy úprav. Na obě vrstvy úprav je na různých snímcích zaměřen ukazatel pozice. Od zaměřeného snímku níže obsahuje každá stopa pyramidu. Levý ukazatel využívá tři pyramidy pro mezivýpočty a zobrazení, pravý ukazatel pouze dvě

Výjimku tvoří případ, kdy je otevřeno více viewportů nad jedním snímkem. Přidání viewportu lze docílit označením požadovaného ukazatele a vrstvy, kterou zobrazuje. V takovém případě viewporty sdílejí stejnou pyramidu. Pokud by se navíc jejich pohledy kryly, místo překryvu bude spočteno pro první viewport a druhý viewport toto místo pouze převezme a nemusí ho počítat.

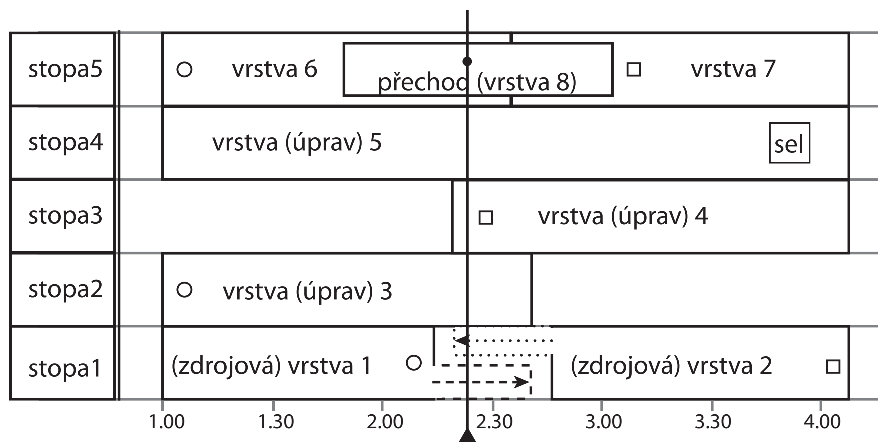
3.5 Budoucí funkce

Jak bylo ukázáno na obrázku 3.7, aktuální návrh časové osy není příliš kompaktní. Pro prolnutí dvou vrstev bylo třeba tří stop, dvou vrstev úprav a přechodu. Proto byl navržena pozměněná funkčnost časové osy s **překrýváním zdrojových vrstev** na obrázku 3.18. Nyní pro prolnutí stačí pouze dvě zdrojové vrstvy. Čárkovaný a tečkovaný obdélník značí skrytá video data (4.3), která nahrazují přechod.



Obrázek 3.17: Ukázka překrývání vrstev

Také by bylo nutné změnit aktuální vstupy ze stop, jelikož by nebylo jednoznačné, která data vrstvy načítá. V obrázku 3.18 jsou na začátku a na konci vrstev zobrazeny obrazce nahrazující nynější vstupy ze stop. Kružnice na konci vrstvy jedna navazuje na vrstvy tři a šest. Obdobně pro čtverec. Použitím tohoto modelu časové osy by se také změnil algoritmus výpočtu.



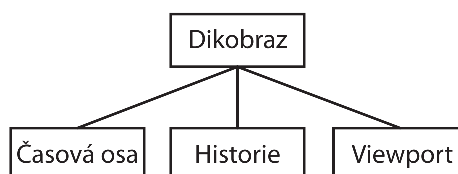
Obrázek 3.18: Příklad budoucí časové osy

Je třeba, aby Dikobraz umožňoval rozšíření vlastní funkčnosti bez zásahu do zdrojového kódu. Toho lze docílit dynamickým načítáním modulů – **pluginů**, neboli také filtrů třetích stran. Proto je třeba zajistit podporu pro takováto rozšíření v jádře programu. I když by se takto daly připojit jakékoliv funkce, pro aktuální verzi se bude rozšíření vztahovat pouze na filtry (modifikátory vrstev, např. křivky, selektivní barvy, inverze aj.).

Výpočetní operace nad obrazovými daty by mohly být urychleny použitím **paralelního zpracování** dat **grafickou kartou** počítače. Aktuální výběr urychlovací metody závisí na požadované přenositelnosti a možnostech grafické karty počítače. V případě, že by karta nepodporovala zpracování obrazu, bude výpočet probíhat přes CPU.

4 Funkční požadavky

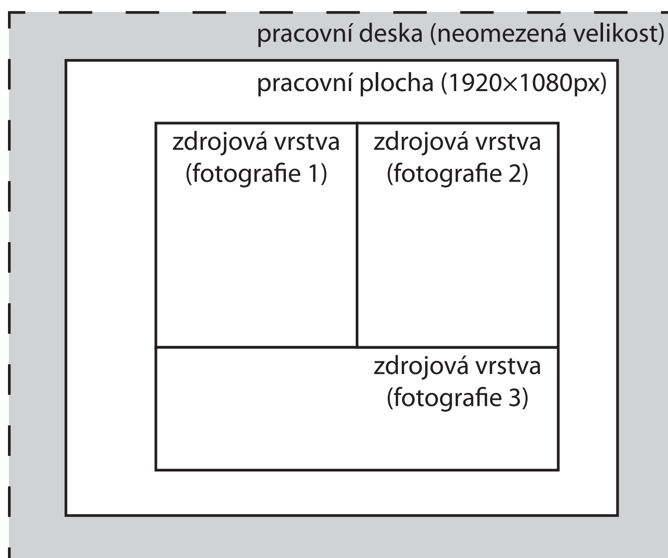
Při návrhu komponent jsem vycházel z požadavků kladených uživatelem na program. Tyto požadavky jsem zařadil do skupin a zapsal. Každý požadavek je označen řetězcem, který označuje jeho funkci. Ke každému řetězci na levé straně je vpravo popsáno chování programu. Následující graf zobrazuje zkrácení identifikátoru požadavku



Obrázek 4.1: Strom funkcí pro zkrácení identifikátorů

Dikobraz.

Projekt.Nový — Při založení nového projektu je uživatel vyzván ke zvolení velikosti *pracovní plochy*, např. si vybere rozlišení Full HD 1920×1080 pixelů. Pracovní plocha je prostor, do kterého budou umísťována všechna data zdrojových vrstev. Načtené fotografie a videa ve zdrojových vrstvách můžou mít velikost menší, než je velikost plochy. Uživatel může jednotlivé zdrojové vrstvy přesouvat libovolně po pracovní ploše. Její velikost lze později změnit přes menu. Změna je jednorázová akce, neukládá se do historie změn. Kromě pracovní plochy se vytvoří i časová osa, s prázdnou stopou. Příklad se třemi zdrojovými vrstvami je na obrázku 4.2.



Obrázek 4.2: Uživatel chce např. provést fotomontáž, kdy ze tří oddělených fotografií složí fotografii jedinou. Vytvoří si pracovní plochu o velikost 1920×1080 pixelů a vloží do časové osy tři zdrojové vrstvy fotografií a na pracovní ploše je posune tak, aby tvořili požadovaný tvar. Na každou z nich může uživatel aplikovat vlastní vrstvy úprav. Šedý prostor okolo plochy, *pracovní deska*, má neomezenou velikost a je přítomna proto, aby uživatel mohl vložit do pracovní plochy fotografii či video větší, než je její velikost. Data, která přesáhnou pracovní plochu nejsou viditelná – uživatel je tažením může posunout dovnitř, či vně pracovní plochy.

Projekt.Otevřít — Načte do programu projekt Dikobrazu, tj. nastavení programu a podobu časové osy. Platí pouze pro načítání do neupravovaného projektu. Pokud byl stávající projekt změněn, bude uživateli nabídnuto okno s možnostmi pro uložení aktuální projektu do souboru.

Projekt.Změna — Pokud byla v aktuálním projektu provedena změna (změna parametrů filtru, posun zdrojové vrstvy, přidání stopy, …), bude v záhlaví programu u názvu projektu přidána hvězdička '*'.

Projekt.Uložit — Uživatel chce uložit aktuální projekt (veškeré nastavení časové osy, velikost pracovní plochy) do souboru na disk.

Projekt.Uložit.OK — Uloží stávající projekt. Hvězdička značící změnu bude odstraněna.

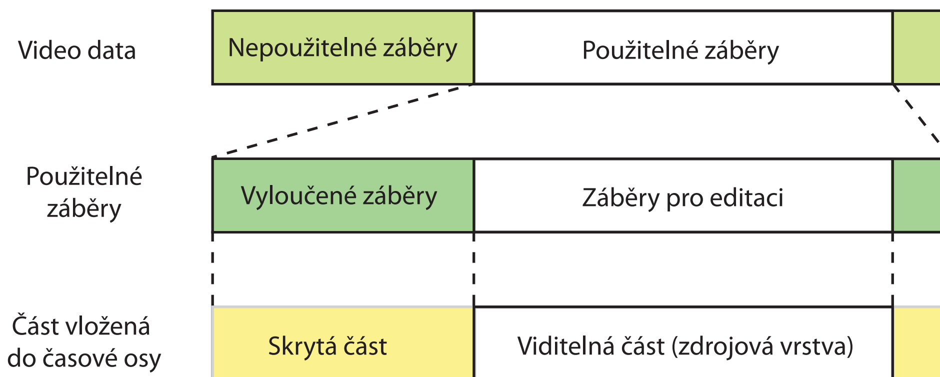
Projekt.Uložit.Zahodit — Změny provedené od posledního uložení stávajícího projektu jsou zahozeny; stávající projekt není uložen.

Projekt.Uložit.Zpět — Zruší dialogové okno pro uložení projektu bez provedení jakékoliv akce.

Projekt.Uložit.Nový	Projekt ještě nebyl uložen. Otevře se okno ve kterém uživatel zvolí jméno a místo pro uložení projektu.
Projekt.Uložit.Uložený	Projekt již byl uložen. Uživatel provede změnu v projektu a znovu ho bude chtít uložit. Přepíše se poslední umístění souboru projektu novým (aktuálním) souborem.
Konec	Ukončení programu. Pokud byla provedena změna, bude uživatel vyzván k uložení aktuálního projektu.

ČasováOsa.

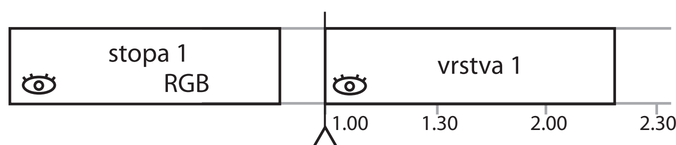
Stopa.Přidat	Bude přidána nová stopa na vrchol do zásobníku stop časové osy.
Vrstva.Přidat	Uživatel má na výběr vložení jedné ze tří druhů vrstev. Ve chvíli vložení vrstvy je vrstva *prázdná*, což pro dané druhy vrstev znamená: • Zdrojová vrstva obsahuje jeden skrytý klíčový snímek. • Vrstva úprav obsahuje jeden skrytý klíčový snímek a neobsahuje filtry. • Přechod přechod obsahuje dva klíčové snímky se shodným nastavením, každý na jedné straně. Neobsahuje filtry. • Každá vrstva je úplně neprůhledná a její režim prolínání je nastaven na „normální“.
VrstvaDat.Přidat	Vloží do vybrané stopy data k úpravě. Uživatel otevře okno pro výběr souborů obrazových dat. V případě, že vybere video soubor, bude z něho moci vybrat použitelná data. Z těch může určit, jaká část bude v časové ose viditelná a která skrytá. Tuto proceduru znázorňuje obrázek 4.3. Detaily této akce jsou popsány v [Zem10, sekce 2.1.1].



Obrázek 4.3: Vložení videa do časové osy

Dále si uživatel zvolí, v jakém rozlišení budou obrazová data do projektu importována. Rozměry vstupních dat budou na toto rozlišení změněny. Běžný je postup, kdy si místo práce s videem s vysokým rozlišením, které se zpracovávalo programem velice dlouho, uživatel zvolí zmenšenou velikost videa, nad kterou bude provádět korekce. Po dokončení úprav nad touto pracovní verzí zamění obsah za video v plném rozlišení a nechá si vygenerovat výsledné video s aplikovanými korekcemi off-line.

Informace o délkách jednotlivých úseků jsou uloženy v příslušné vrstvě (tabulka 3.2). V případě vybrání obrázku bude vložen celý. Počátek dat bude na pozici aktivního (hlavního) ukazatele pozice v časové ose 3.1.7. Tato funkce je přístupná, pouze pokud je vybrána stopa. Výsledek akce ukazuje obrázek 4.4.



Obrázek 4.4: Vložení vrstvy na pozici aktivního (hlavního) ukazatele

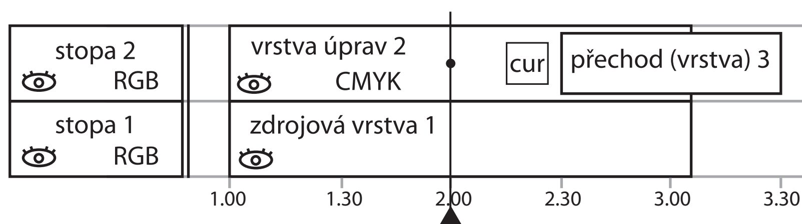
VrstvaÚprav.Přidat	Vloží do vybrané stopy vrstvu úprav. Počátek vrstvy bude na pozici aktivního (hlavního) ukazatele pozice v časové ose. Tato funkce je přístupná, pouze pokud je vybrána stopa. Uživatel může vložit vrstvu úprav bez použití zdrojové vrstvy. Úpravy pak budou aplikovány na nultou stopu.
Přechod.Přidat	Vloží do časové osy přechod. Možností je mnoho. Uživatel může označit dvě vrstvy, nad kterými bude vytvořen přechod. Nebo stačí označit pouze jednu vrstvu, druhý konec přechodu bude nad prázdným místem ve stopě. Pokud bude chtít uživatel vložit přechod na prázdné místo, posune na něj aktivní ukazatel a vytvoří ho tam. Tato funkce je přístupná, pouze pokud je vybrána stopa.
Vrstva.Viditelnost	Uživatel může stiskem značky oka měnit viditelnost vrstvy pro výpočetní algoritmus z kapitoly 5.1. Otevřené oko znamená, že je vrstva „aktivní" a bude zahrnuta v algoritmu, Pokud je oko zavřené, vrstva v algoritmu zahrnuta nebude.
Vrstva.Průhlednost	Uživatel může pro každou vrstvu změnit procentuální hodnotu průhlednosti vrstvy. Tato funkce je přístupná, pouze pokud je vybrána datová vrstva nebo vrstva úprav.
Filtr.Přidat	Vloží do vybrané vrstvy úprav filtr. Tato funkce je přístupná, pouze pokud je vybrána vrstva úprav.
KlíčovýSnímek.Přidat	Vloží do vybrané vrstvy úprav či přechodu na pozici aktivního (hlavního) ukazatele pozice v časové ose klíčový snímek se základními parametry všech filtrů. Pokud byl ve vrstvě do té doby pouze jeden (neviditelný) klíčový snímek, tak se zobrazí. Tato funkce je přístupná, pouze pokud je vybrána vrstva úprav či přechod a aktivní ukazatel je uvnitř vrstvy.
KlíčovýSnímek.Duplikát	Zduplikuje klíčový snímek a všechny jeho nastavení. Tato funkce je přístupná, pokud je vybrán klíčový snímek a aktivní ukazatel je nastaven na vrstvu, ve které je označený klíčový snímek. V každý časový okamžik může být v každé vrstvě úprav či přechodu na každém snímku pouze jeden klíčový snímek. Proto vrstva s deseti snímky může obsahovat pouze deset klíčových snímků. Pokud by se uživatel snažil vložit (zduplikovat) na obsazené místo další klíčový snímek, bude tato akce odmítnuta.
Čas.Přiblížit	Přiblíží časovou osu. Funguje na principu nástroje *lupa*. Zvýší se tím úroveň detailů časové osy.
Čas.Oddálit	Oddálí časovou osu. Funguje na principu nástroje *lupa*. Sníží se tím úroveň detailů časové osy.

Ukazatel.Hlavní	Pokud je v časové ose pouze jeden ukazatel pozice v časové ose, je automaticky vždy aktivní (hlavní). Pokud existuje více jak jeden ukazatel pozice, může uživatel vybrat jeden z nich, který se stane aktivním (hlavním). S tím budou reagovat další funkce (přidej klíčový snímek, filtr, ad.). Uživatel může nastavit jiný ukazatel jako hlavní. Předešlý hlavní ukazatel se stane neaktivním. Vždy může být pouze jeden aktivní (hlavní) ukazatel.
Skrýt	Okno s časovou osou je skryto tlačítkem zavřít, nebo aktivací položky v menu.
Zobrazit	Okno s časovou osou je zobrazeno (odkryto) příslušným tlačítkem, nebo položkou v menu. Okno se zobrazí se stejným nastavením, jako bylo skryto.
Ukotvit	Okno časové osy je možné ukotvit na jednu ze čtyř stěn hlavního okna programu. Tažením může být okno „překotveno" na jinou stranu.
Uvolnit	Okno časové osy je možné uvolnit ode stěny tlačítkem na okně, či tažením. Tuto akci je možné provést pouze tehdy, je-li okno časové osy ukotveno.
Historie.	
Zpět	Vrátí se o jeden krok zpět v historii úprav.
Vpřed	Posune se o jeden krok dopředu v historii úprav.
Viewport.	
Zobraz	Uživatel zvolí vrstvu nad kterou prochází ukazatel pozice v časové ose a vloží do ní zaměřovací bod. Otevře se viewport a aktivuje se algoritmus pro výpočet obrazu. Výsledný obraz se z pyramidy zobrazí uživateli.
Obraz.Přiblížit	Přiblíží obrazová data. Jsou zobrazena data z vrchnějšího patra pyramidy.
Obraz.Oddálit	Oddálí obrazová data. Jsou zobrazena data ze spodnějšího patra pyramidy.
Obraz.Exportovat	Obrazová data zobrazená do viewportu je možné exportovat na disk jako soubor obrazu.

5 Algoritmizace funkcí systému

5.1 výpočetObrazu() – pseudoalgoritmus

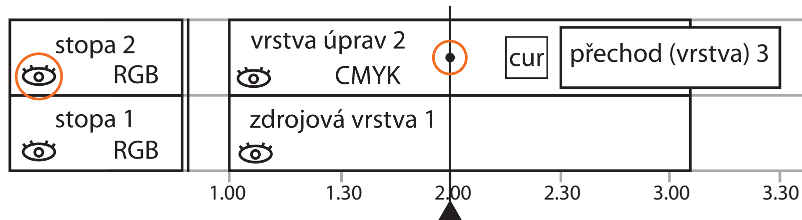
Tato metoda, založená na principu **rekurzivního sestupu**, provádí výpočet požadovaného obrazu z časové osy. Impuls k vykonání vydá uživatel, když si nechá zobrazit viewport na určité vrstvě. Následující kód má na levé straně zobrazenou řádku. Po blocích kódu následuje jeho popis. Funkčnost každého bloku bude naznačena na obrázku 5.1.



Obrázek 5.1: Ukázkový příklad jako průvodce zdrojovým kódem. Časová osa obsahuje dvě stopy a tři vrstvy. V první stopě je zdrojová vrstva, ve druhé je vrstva úprav s filtrem křivky (cur) a přechod typu vrstva-prázdno

```
1   vstup: viewport
2
3   track:= viewport→trackId
4
5   if (track neaktivni)
6
7     A:= vypocetObrazu(predchozi trackId)
8     KONEC
```

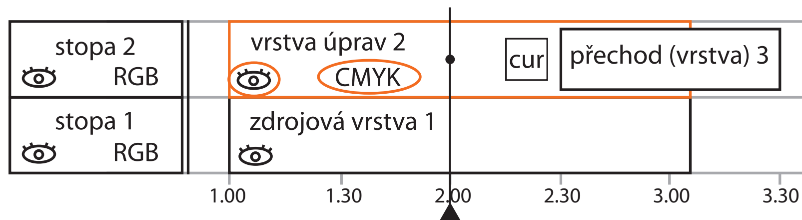
Vstupem je příslušný ukazatel pozice spřažený s viewportem. Z něho se zjistí stopa. Pokud je stopa neaktivní, je do obrazu A přiřazena návratová hodnota rekurzivně volané metody s parametrem identifikátoru předchozí stopy.



Obrázek 5.2: Uživatel si zvolí, že chce zobrazit snímek z druhé minuty vrstvy úprav. Proto na vrstvu vloží zaměřovací bod. Vstupem do programu je tedy viewport a jeho ukazatel pozice v časové ose. Z viewportu se zjistí, že stopa, ze které se čte, je stopa 2. Stopa je aktivní (oko je otevřené) a algoritmus nevykoná činnost větve *if*

```
9   else if (track aktivni) {
10    actualLayer := pozice layer pod ukazatelem
11    if (actualLayer neaktivni nebo neexistuje)
12      A:= vypocetObrazu(track→inputTrackId)
13      goto transition
14
15    if (actualLayer→colorspace = default)
16      colorspace:= track→colorspace
17
18    else
19      colorspace:= actualLayer→colorspace
```

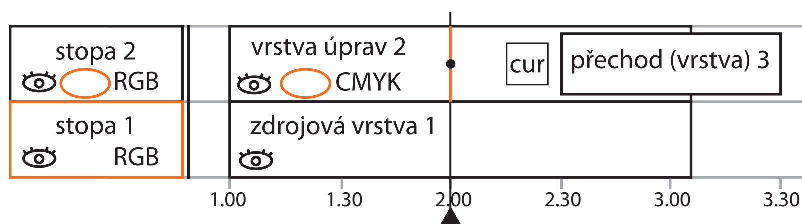
Aktuální stopa je aktivní. Aktuální vrstvu zjistíme z parametrů trackId a playhead→frame uložených ve viewportu. V dané stopě projdu všechny vrstvy a z parametrů startFrame a endFrame zjistím, v jaké vrstvě leží hledaný snímek. Pokud je nalezená vrstva neaktivní, nebo neexistuje, rekurzivně zavolám metodu s identifikátorem stopy, ze které čerpám data. Po dokončení rekurze provedu skok na aplikaci přechodu. Pokud je vrstva aktivní, nastavím proměnnou barevného prostoru. Ta má buď hodnotu barevného prostoru vrstvy, nebo, pokud ta je neurčená, barevného prostoru stopy.



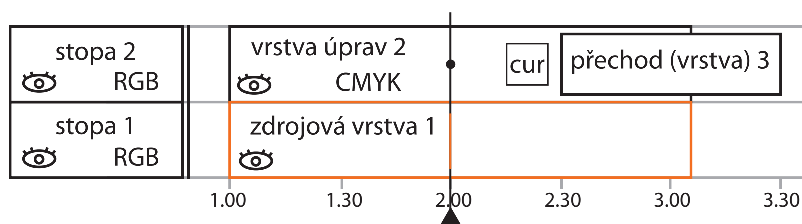
Obrázek 5.3: Stopa 2 je aktivní a v tom případě algoritmus označí aktuální vrstvu, která leží pod ukazatelem pozice. Jelikož je aktuální vrstva aktivní, načte se její barevný prostor, který uživatel lokálně přepsal na CMYK

```
20    actualFrame = viewport→playhead→frame
21    if (actualLayer is zdrojova vrstva)
22       A:= obrazek from actualFrame
23
24    else {
25       if (actualLayer→inputTrackId = default)
26          inputTrackId := track→inputTrackId
27
28       else
29          inputTrackId := actualLayer→inputTrackId
30
31       if (inputTrackId = default)
32          inputTrackId := predchozi trackId
33
34       A:= vypocetObrazu(inputTrackId)
35    }
```

Pokud je aktuální vrstva zdrojová, načti do obrazu A obrazová data ležící na pozici snímku. Pokud není zdrojová, zjisti, jestli aktuální vrstva či stopa obsahují uživatelsky nastavený vstup dat a ulož ho do lokální proměnné. Pokud není uživatelsky nastavený, zjisti identifikátor předchozí stopy a zaznamenej ho. Nad lokální proměnnou zavolej rekurzivně metodu.



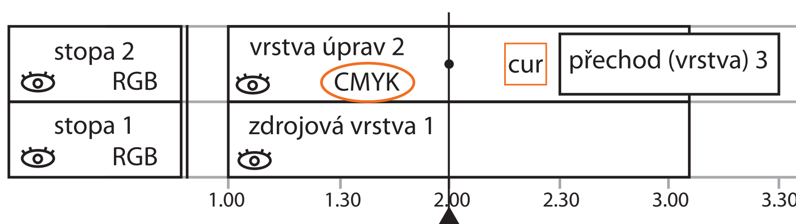
Obrázek 5.4: Uloží se aktuální snímek zaměřený ukazatelem pozice a bodem ve vrstvě úprav. Jelikož není aktuální vrstva zdrojová, algoritmus bude hledat, odkud mají přijít data. Ani aktuální vrstva, ani stopa, která ji obsahuje, nemají nastaveny uživatelské vstupy. Proto je nalezen identifikátor předchozí stopy, což je stopa 1. Metoda výpočetObrazu() je rekurzivně zavolána nad stopou 1. Postup zpracování je popsán na obrázku 5.5



Obrázek 5.5: Stopa 1 je aktivní a aktuální vrstvou se stává zdrojová vrstva. Přijme se barevný prostor RGB ze stopy a jelikož je stopa poslední (další stopy už nejsou), vrátí metoda obrazová data načtená pod ukazatelem pozice převedená do barevného prostoru RGB

```
36    preved A do colorspace
37
38    if (actualLayer neprázdná) {
39        // filtry musí být zpracovány v takovém pořadí, v kterém jsou aktuálně zobrazené ve vrstvě.
40        opakuj pro vsechny aktivni filtry z~aktuální vrstvy:
41        interpoluj interactionParams z actualLayer→filterId
42          (predchozi keyframe, nasledujici keyframe, actualFrame)
43        interpoluj filterParams z actualLayer→filterId
44          (predchozi keyframe, nasledujici keyframe, actualFrame)
45
46        B:= aplikuj filtr (filterParams)
47        A:= interaction (A, B, interactionParams)
48    }
```

V této chvíli máme obraz uložený v proměnné A. Pokud aktuální vrstva není *prázdná* (4, spočti hodnoty parametrů interakce a parametrů pro všechny filtry ve vrstvě. Pro každý z nich ulož do proměnné obrazu B obrazová data interpolovaných parametrů filtru aplikovaných na aktuální snímek. Do hlavního obrazu A následně ulož průnik obrazů A a B řízený interpolovanými parametry interakce.



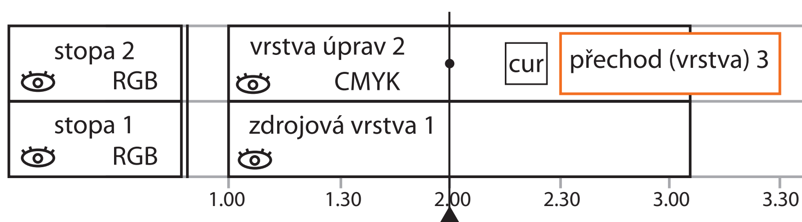
Obrázek 5.6: Z rekurze byla vrácena obrazová data A v barevném prostoru RGB. Aktuální vrstva ale pracuje v barevném prostoru CMYK, a proto do něj budou data převedena. Aktuální vrstva obsahuje jeden filtr a nezměněné parametry interakce. Proto bude do obrazu B uložen výsledný obraz po aplikaci filtru na data A. Nakonec je provedeno sjednocení obou obrazů do A podle parametrů interakce. Další blok kódu nebude vykonán, protože předchozí vrstva (zdrojová vrstva 1) má nezměněné parametry interakce a není pro algoritmus zajímavá

```
49    B := vypocetObrazu(predchozi trackId)
50    preved B do colorspace
51
52    if (actualLayer→interactionParams = default)
53      interactionParams = track→interactionParams
54
55    else
56      interpoluj interactionParams z actualLayer (predchozi keyframe, nasledujici keyframe, frame)
57
58    A:= interaction(A, B, interactionParams)
59    goto transition
60  }
```

Může nastat případ, že aktuální stopa má nastavenou např. průhlednost, a proto musí být zahrnuta do výpočtu předchozí stopa. Pokud by byly parametry nezměněny, není třeba tuto část vykonávat – jakékoliv změny pod touto stopou nebudou propagovány výše a v této chvíli pro výpočet nejsou zajímavé. Obraz B můžeme přepsat daty vrácenými metodou s parametrem identifikátoru předchozí vrstvy. Převedeme B do lokálního barevného prostoru. Pokud nejsou parametry interakce aktuální vrstvy změněny oproti standardním hodnotám, ulož do lokální proměnné parametry z aktuální stopy. V opačném případě proveď interpolaci parametrů interakce z aktuální vrstvy. Do hlavního obrazu A následně ulož průnik obrazů A a B řízený interpolovanými parametry interakce.

```
61  transition:
62
63  if (transition pod actualFrame existuje a je aktivni)
64     proved vetev na radce 9 pro druhou odpovidajici vrstvu, vysledek uloz do B
65     preved A a B do transition→colorspace (if default, pak do track→colorspace)
66     interpoluj interactionParams z transition (predchozi keyframe, nasledujici keyframe, frame)
67
68     // a případně aplikovat filtry, jsou-li na transition
69     A:= transition(A, B, interactionParams)
```

Pokud je aktuální snímek nad aktivním přechodem, je třeba provést celý výpočet znovu pro vrstvu, nad kterou stojí druhý konec přechodu. Výsledek je uložen do obrazu B. Převede A i B do barevného prostoru přechodu, nebo, pokud má nezměněné hodnoty, do prostoru stopy. Proveď interpolaci parametrů interakce z přechodu. Do hlavního obrazu A následně ulož průnik obrazů A a B řízený interpolovanými parametry interakce. Výsledný obraz A zobraz do viewportu.



Obrázek 5.7: I když časová osa obsahuje přechod, není nad ním nastaven ukazatel pozice v časové ose. Tato část kódu bude přeskočena, výsledek algoritmu uložen do pyramidy a zobrazen uživateli do viewportu

6 Grafické knihovny pro manipulaci s bitmapou

Aby nebylo třeba programovat všechny základní operace pro manipulaci s obrazem, hledal jsem a poměřoval vlastnosti a schopnosti různých zavedených grafických knihoven. Vyhledávaná kriteria jsou:

- Použit jazyk C či C++.
- Existuje programátorská dokumentace.
- Operace knihovny nad fotografiemi s rozlišením 12 Mpx musí být rychlé.
- Sofistikovaný systém pro ukládání mezivýsledků operací.
- Schopnost načítat a ukládat soubory v různých barevných prostorech.
- Jak vývoj, tak komunita uživatelů musí být aktivní.
- Existuje jako samostatná grafická knihovna.
- Poskytuje rozhraní pro komunikaci program-knihovna.
- Přeložitelnost a spustitelnost kódu na platformách GNU/Linux a Windows.
- Podpora nelineární cesty zpracování obrazu.

Uvedu knihovny, které nesplnily uvedené požadavky:

- *Krita* je více zaměřena na vektorové a bitmapové kreslení a ne na operace pro manipulaci s obrazem.
- Populární *GIMP* není distribuován jako knihovna a nedovoluje plnohodnotnou úpravu obrazu v barevných prostorech CMYK a Lab.
- *Vigra* je příliš specializovaná na počítačové vidění a není zaměřena na interaktivní používání.
- *VLX* nemá dostatečnou dokumentaci a podporuje pouze RGB obrázky.
- *CImg* má nedostatečnou dokumentaci a vše včetně GUI je uloženo v jediném hlavičkovém souboru o rozsahu 40 000 řádků zdrojového kódu.
- *ImageMagick/GraphicsMagick* pracuje pouze jednoprůchodově, po dokončení běhu není možné nalezený postup použít znovu. Není zaměřena na interaktivní používání.
- *GIL* je zpravována firmou Adobe a je velice dobře napsána. Její vývoj není aktivní a naučit se ji ovládat je dosti náročné.

Nakonec jsem vybral dva kandidáty, jejichž specifikace splňovala dané podmínky. Jsou jimi grafické knihovny **VIPS** a **GEGL**. Obě knihovny jsou vyvíjené v systémem GNU/Linux. Zaměřil jsem se na testování parametrů každé z nich.

6.1 VIPS

Knihovna VIPS (http://www.vips.ecs.soton.ac.uk) vyniká zpracováním obrázků větších, než je velikost operační paměti počítače. Automaticky pracuje ve více vláknech a podporuje mnoho barevných prostorů včetně prostorů uživatelsky definovaných. Použitím specifické datové struktury potřebuje pro svůj běh pouze malé množství počítačové paměti. Je psaný v jazyce C, existuje i C++ API.

Na vývoji VIPSu pracuje aktivně osm vývojářů. Aktualizace programu a novinky jsou zveřejňovány přibližně jedenkrát měsíčně na internetových stránkách projektu. Na mailling listu VIPSu je nízký, ale stálý provoz.

Již jsem zmínil, že jsem se snažil zkompilovat zdrojové soubory VIPSu pod systémem Windows. Funkční verze pro Windows jsou vytvářeny odděleně a k dispozici je pouze již vytvořený spustitelný instalační soubor, který nainstaluje jak VIPS, tak i jeho GUI s názvem *nip2*. Po prostudování pokynů na stránkách vývojářů je třeba k vytvoření verze použitelné pro vývoj na systému Windows několik dalších prvků. Podrobnosti k instalaci jsem objevil v jednom z vláken maillistu vývojářů VIPSu [Ken10]. Programátor se v něm ptá, jak zkompilovat VIPS pod Windows. V odpovědi mu hlavní vývojář VIPSu John Cupitt píše, že k překladu a vývoji pod Windows by bylo třeba linkovat zdrojový kód VIPSu proti knihovně msvcrt.dll. Dále je třeba přidat a udržovat aktuální verze všech knihoven, které bude msvcrt.dll potřebovat. Přidává poznámku, že jelikož je VIPS složen ze 26 malých projektů, bude třeba přidat a aktualizovat i je. Pokud bude vývoj probíhat nástrojem MinGW (Minimalist GNU for Windows), tak bude třeba přeložit a nastavit nástroj pro spojování knihoven pkg-config. Na závěr John Cuppit píše:

> „Doufám, že tyto potíže (s překladem pod Windows) budou odstraněny do několika let.“

Na archivní stránce vývojářů VIPSu [Cup08] je návod na kompilaci, podle kterého je možné přeložit několik starších verzí VIPSu. S překladem jsem neuspěl a tak jsem se rozhodl testovat knihovnu v systému GNU/Linux.

Dokumentace k API je obecná a nastiňuje funkci jednotlivých komponent VIPSu. K operacím nad fotografiemi jsou napsány manuálové stránky, jejichž názvy i obsah jsou nepřehledné. Výuková dokumentace ukazující možnosti knihovny (průvodce programem) je dobře udělaná, seznámí uživatele začátečníka s možnostmi knihovny.

V dokumentaci je popsán i postup, jak v grafickém rozhraní *nip2* psát uživatelské pluginy ve speciální VIPS syntaxi. Takto vytvořené pluginy je možné ukládat, načítat i zavést do VIPSu dynamicky. Vytvářet komplikované filtry (přistupovat k jednotlivým bytům, provádění bitového posuvu) by bylo velice obtížné. Správnost napsaného kódu lze kontrolovat pouze rovnou ve VIPSu otestováním jeho funkčnosti.

VIPS používá k uložení obrazových dat vlastní formát, do kterého ukládá i hlavičku délky 64 bytů s užitečnými informacemi o datech (výška, šířka, offset, rozlišení, počet barevných kanálů, počet bytů použitých pro jeden kanál, barevný prostor). Jeden obrázek může celkem obsahovat až 2^{32} kanálů. Počet bytů v kanálu je určeno použitým datovým typem. Možnosti jsou: 8, 16 nebo 32bitové znaménkové i neznaménkové celé číslo, čísla s plovoucí desetinou čárkou a čísla komplexní. Na konci obrazových dat mohou být uložena doplňková XML data, která obsahují metadata obrazu, jako ICC profil (formát obsahující detailní informace o barevném profilu) či jeho historii. Doplňková data jsou uložena v bitové posloupnosti (little/big endian) podle hardwaru hostujícího počítače. Pro pole v hlavičce jsou připraveny metody pro čtení. Tyto metody také zaručují nezávislost na hardware počítače. V případě potřeby dojde k bitovému skládání – pokud je třeba, tak se při čtení změní pořadí čtení bitů.

VIPS podporuje mnoho barevných prostorů i bitových hloubek. Použitelné barevné prostory v testované verzi jsou bitmapa, 16bitové odstíny šedi, 8 a 16bitové RGB, až 64bitový Lab, XYZ a UCS. Barevný prostor CMYK dokáže rozpoznat, ale nesprávně manipuluje s jednotlivými barevnými kanály. V dokumentaci k VIPSu jsem se dočetl, že implementované barevné prostory jsou zastaralé a uživateli je doporučeno používat vlastní ICC profily. Při přidání externího ICC profilu k fotografii byla interpretace obrazu správně změněna.

Data načteného obrázku jsou uložena ve struktuře *VImage*, která se skládá z pospojovaných dlaždic. Daty se plní pouze ty dlaždice, které jsou vyžádány k zobrazení do viewportu *nip2*.

Testování

Pro otestování kritérií rychlosti zpracování korekcí obrazu, přiblížení obrazu a ukládání mezivýsledků do paměti jsem provedl několik testů. Všechny budou provedeny v prostředí *nip2* s fotografií ve formátu JPEG s rozlišením 4000$\times$4000 pixelů (přibližně 15,3 Mpx).

Test1: Jak jsou data ukládána do struktury *VImage*? VIPS by měl vynikat jak rychlostí zpracování velkých fotografií, tak nízkou spotřebou paměti. Zkusím zjistit, co je na tom tvrzení pravdy. Do viewportu načtu fotografii ve 100% velikosti. Jelikož je fotografie větší než velikost viewportu, uvidím pouze její část. Budu pohybovat s posuvníky viewportu a zobrazovat si jiné části fotografie. Po otevření viewportu byla zobrazena levá horní oblast fotografie. Pokud jsem trochu pohnul libovolným posuvníkem, k aktuálnímu zobrazení ve viewportu se přidaly pouze nové bloky fotografie. Pokud jsem posunul posuvník na původní místo, znovu se mi zobrazil levý horní roh fotografie, tentokrát nebylo třeba nic dopočítávat, vše je již uloženo v paměti. Test byl úspěšný, zobrazení fotografie a dopočty dalších částí byly rychlé. Stejně tak pouze zobrazené a dopočtené části byly uloženy do paměti ve struktuře *VImage*.

Test2: Zkoumám práci knihovny s oddáleným obrazem, na kterém je aplikováno ostření, náročná korekce obrazu, s rozsahem padesáti pixelů. Potřebuji totiž zjistit, jakým způsobem jsou aplikovány korekční operace na různé hodnoty přiblížení/oddálení obrazu. Nechám si do viewportu zobrazit korekci nad fotografií ve 100% velikosti a následně oddálím obraz na 6 %. Po ukončení testu jsem zjistil, že po načtení fotografie ve 100 % byla provedena korekce pouze na části obrazu, kterou jsem viděl. Po oddálení zobrazení na 6 % byla viditelná celá fotografie a korekční změna aplikována postupně na celou fotografii. Při libovolné změně přiblížení fotografie byl obraz z viewportu celý přepočítán. Z toho plyne několik poznatků:

- Operace nad oddáleným obrazem je velice pomalá. Jelikož se při barevných korekcích často pracuje s oddáleným obrazem, byla by práce s ním nepoužitelná.
- Dále jsem zjistil, že pokud je zpracovávaná operace náročná na výpočetní čas procesoru, tak ho vytěžuje na 100 % bez možnosti zastavení probíhajícího výpočtu obrazu.
- Pokud je obraz při změně rozlišení přepočítáván, vypadá to, že VIPS nepoužívá ukládání mezivýpočtů do paměti. Pro potvrzení tohoto tvrzení jsem provedl další test.

Test3: Je využívána paměť pro ukládání mezivýpočtů (cache)? Z dokumentace neplyne, jestli si každá korekční metoda udržuje vlastní *VImage*, což by umožňovalo používat cache, či je použit jiný systém pro ukládání mezivýsledků. Pokud by VIPS nevyužíval cache, musel by po každé změně parametru jakékoliv korekční operace znovu provést výpočet a případné zobrazení výsledného obrazu, což by znamenalo vysoké výpočetní časy. Na fotografii aplikuji operaci USM (Unsharp Mask, operace ostření obrazu) s rozsahem padesát pixelů a následně aplikuji operaci barevné inverze (vytvoří negativ fotografie). Její aplikace je mnohem méně časově náročná, než operace USM. Zobrazím obrázek s aplikovanou maskou v 6% velikosti. Výsledky této akce znám z předchozího testu. Nyní budu chtít zobrazit operaci inverze. Pokud by byl obrázek s USM uložen v paměti cache, časová náročnost pro provedení inverze by byla v poměru k aplikaci USM zanedbatelná. Výsledek testu potvrdil, že čas pro provedení operace inverze byl delší, než čas pro aplikaci USM. Z toho plyne, že VIPS nepoužívá cache.

Po ukončení testování jsem ze zjištěných výsledků rozhodl, že **VIPS není použitelný** pro Dikobraz. Hlavním důvodem je neexistence struktury pro ukládání mezivýpočtů (např. pyramidový systém). Každá změna parametrů filtru či změna přiblížení obrazu ve viewportu musela být počítána od začátku. Druhý kandidát, knihovna GEGL, tento nedostatek nemá.

6.2 GEGL

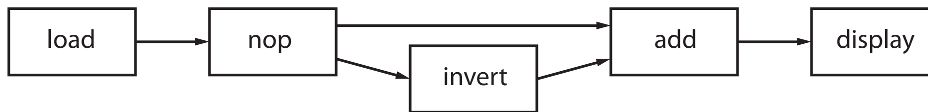
Knihovna GEGL (http://gegl.org) pracuje na principu vytvoření acyklického stromu operací (grafu), kde jednotlivé operace, představující vrcholy stromu, jsou vzájemně pospojovány. GEGL dokáže zpracovat i obrazy větší, než je velikost paměti počítače. Informace, které už se do paměti nevejdou, jsou ukládány do swapovacích souborů na pevný disk. Všechny operace jsou nedestruktivní, data zdrojového obrazu nejsou změněna. Pokud by byl GEGL nainstalován na počítač s podporou více vláken, budou operace paralelizovány. GEGL je třeba mít nainstalovaný spolu s knihovnou BABL (http://gegl.org/babl), která se GEGLu stará o převod obrazových dat mezi barevnými prostory a interpretaci barev. K převodu může docházet požadavkem operace GEGLu. Obě knihovny jsou psané v jazyce C. GEGL je vyvíjen od roku 2000 s tím, že uprostřed této doby byl vývoj na několik let pozastaven. Poslední tři roky je však vývoj velice aktivní.

Na vývoji knihovny pracuje aktivně nejméně pět vývojářů. Tuto dolní hranici jsem odvodil z příspěvků na kódové úložiště *git*. Na stránkách GEGLu je uvedeno celkem 55 jmen těch, kteří se za celou dobu života GEGLu podíleli na jeho vylepšování. Knihovna BABL je vyvíjena souběžně s GEGLem stejným týmem vývojářů. Aktualizace programu a novinky jsou zveřejňovány přibližně jednou za půl roku na internetových stránkách projektu. Na mailling listu GEGLu je nízký, ale stálý provoz.

Instalace pod Windows je dost náročná. Na stránkách vývojářů není mnoho pomocných informací k průběhu instalace. Poté, co jsem doinstaloval mnoho věcí ručně, jsem narazil na cyklickou závislost balíčků, kterou jsem nepřekonal. Instalaci pod GNU/Linux jsem provedl ručně podle instrukcí na stránkách GEGLu.

Stručná dokumentace popisující jednotlivé veřejné metody je na stránkách GEGLu. Tyto metody se dělí na dvě skupiny. První obsahuje popis metod API (inicializace GEGLu, založení stromu operací, vytvoření vrcholu stromu, atd.) a druhá metody pro operaci s obrazovými daty (USM, křivky, selektivní výběr barvy, aj.). Dokumentace k tvorbě programů s využitím GEGLu téměř není kromě dvou příkladů na stránkách vývojářů. Další informace jsem zjišťoval pročítáním vývojářského maillistu. Podrobnou funkci programu mi prozradilo až zkoumání zdrojového kódu. Dokumentace k BABLu obsahuje všechny doposud implementované BABL formáty barevných prostorů a ukázkový příklad převodu z jednoho barevného prostoru do jiného.

Pro popis funkce knihovny použiji příklad. Je vytvořen strom operací zobrazený na obrázku 6.1. Po vytvoření acyklického stromu je zavolána metoda *process* na výstupní vrchol. V průběhu této metody jsou postupně provedeny všechny operace v pořadí od kořene stromu k výstupnímu vrcholu. Po dokončení operace můžou být data zobrazena uživateli.



Obrázek 6.1: Příklad vytvořeného acyklického grafu operací

Načtení obrázku umožňuje jako jediná operace *load*, která dokáže načíst formáty obrázků JPEG, PNG a BMP. Pokud by na vstup přišel obrázek nepodporovaného typu, proběhne konverze do zpracovatelného formátu knihovnou *magick*. Operace *load* vždy načte celý obrázek do paměti nebo do swapovacího souboru – nelze načíst pouze vybranou část. *Load* načítá obrázek vždy jako lineární 32bitové RGBA s plovoucí desetinnou čárkou, což dává 128 bitů pro barevný pixel (s alfa kanálem). Lineární znamená, že je změněna gamma obrázku na hodnotu jedna. Při převodu do nelineárního barevného prostoru je gamma vrácena na původní hodnotu. Oproti fotografii v barevném prostoru osmibitového RGB se zvýší jeho objem dat $5,\overline{3}\times$.

Jako další je zařazena operace *nop*. Ta zde funguje pouze jako „roura". Neprovádí žádnou operaci, pouze uchovává data z předchozího vrcholu. Bylo ji nutné zařadit, jelikož z operace *load* nelze rozdělit výstup dat. Z „roury" již rozdělit výstup lze. Z obrázku 6.1 je patrné, že výstup z vrcholu *nop* bude využit dvakrát. Jednou pro operaci *invert*, podruhé pro operaci *add*. Načtený obrázek z vrcholu *load* je třeba někde udržovat v paměti. Všechny vrcholy grafu, u kterých není zakázáno ukládat data do paměti, obsahují paměť pro ukládání mezivýpočtů. Touto pamětí je pyramidový model založený na podobném principu jako systém pyramid programu Dikobraz z kapitoly 3.4. V GEGLu má každá pyramida tři patra podle použitého oddálení fotografie, 100%, 50% a 25%.

Vrchol *invert* vytváří z dat negativ. Operace *add* přijímá hodnoty ze dvou vstupních obrazových dat a výsledný obraz je součtem hodnot stejnolehlých obrazových bodů. Nakonec zavoláme metodu *process* nad vrcholem *display* a teprve v té chvíli začne celý výpočet. Výsledkem by měla být jednobarevná bílá plocha, přičemž nezáleží na vstupních datech. Samotný průběh výpočtu není přerušitelný.

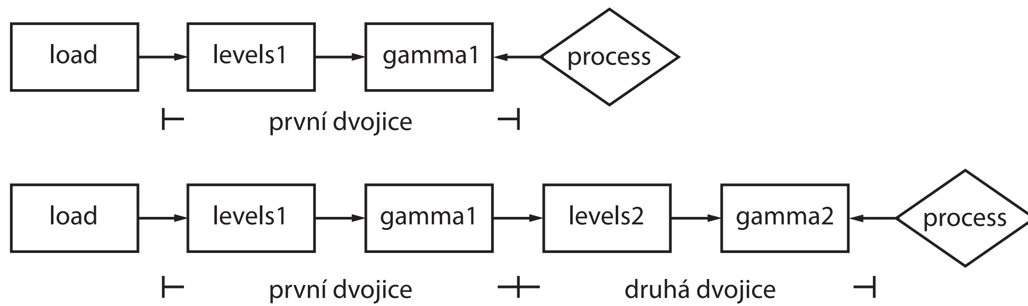
Pokud bych chtěl zpracovat pouze část obrazových dat, musím místo metody *process* použít metodu *gegl_node_blit* (zkráceně *blit*). Ta v aktuální verzi GEGLu nemá zapnutou podporu pro ukládání dat do pyramid (tato informace je vepsána do zdrojového kódu), může je pouze načítat. Předtím, než je možné začít pracovat s obrázkem *blitem*, je nutné zavolat metodu *process* na kořen stromu, aby byla naplněna jeho pyramida daty. Na určení oblasti, která se má zobrazit, se používá objekt GeglRectange s parametry (levý horní roh[x,y], délky stran obdélníku[šířka, výška]). Operace rozmazání či ořez modifikují zpracovávanou velikost oblasti požadované uživatelem.

Pro lepší pochopení funkce GEGL operací jsem prozkoumal jejich zdrojový kód. Vstupem do operace jsou obrazová data ve stejném formátu jako výstup z předešlé operace. Také lze nastavit vlastní vstupní i výstupní barevný prostor pro operaci. Toho se využívá v případě, že algoritmus vyžaduje přesný formát dat. Příkladem může být operace, která na vstupu očekává data v barevném prostoru osmibitového RGB s rozsahem hodnot [0-255]. Pokud bych poslal na vstup RGB s plovoucí desetinou čárkou s rozsahem [0.0-1.0] nastala by v algoritmu chyba nebo by byla data poškozena. Pokud je vstupní/výstupní barevný prostor stejný jako výstupní prostor předchozí operace, konverze prostoru neproběhne. Většina operací je přizpůsobena pro práci s daty v lineárním 32bitovém barevném prostoru RGBA s plovoucí desetinou čárkou a ty co nejsou, si na vstupu převedou data na požadovaný formát.

Testování

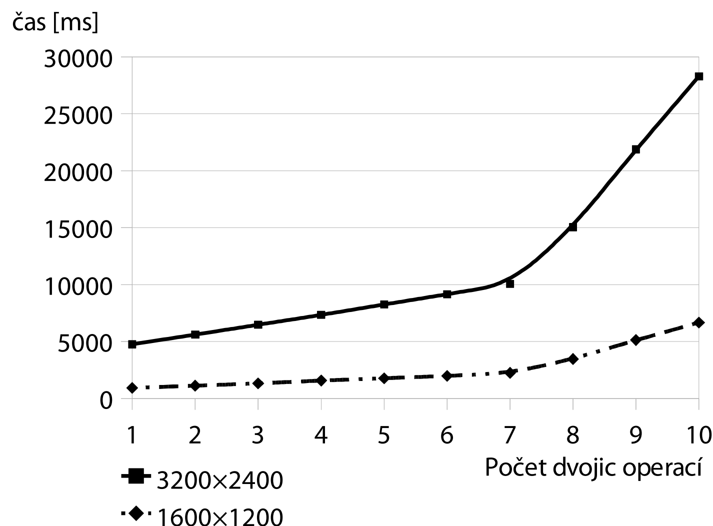
Test1: Zjišťuji růst výpočetního času v závislosti na počtu operací lineárně zařazených ve stromu. Nikde v dokumentaci není psáno, v jakém čase program běží. I když bych očekával, že poroste lineárně. Strom

vytvořím pro jeden, dva, …, až deset lineárně spojených párů operací *levels* a *gamma* s ukázkou na obrázku 6.2.



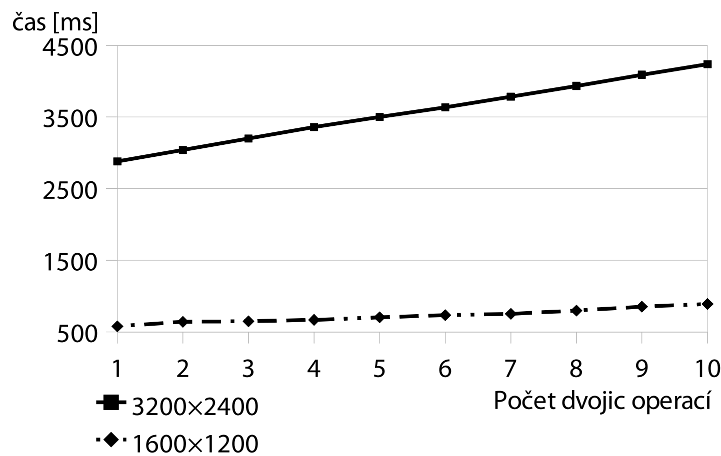
Obrázek 6.2: Dva stromy pro jednu a dvě dvojice operací *levels* a *gamma*

Každý ze stromů budu testovat padesátkrát pro pět různých rozlišení jedné fotografie. Z časů pro každý strom spočtu průměr, který zanesu do grafu. Výsledek testu nedopadl dle očekávání. Zaznamenané hodnoty časů pro dvě největší rozlišení jsou zobrazeny v grafu 6.3. Další nižší rozlišení jsem do grafu nezanášel, jelikož je jejich rozdíl nepatrný.



Graf 6.3: Výpočetní složitost dvojic operací levels-gamma

Průměry časů rostou přibližně až do sedmi dvojic lineárně, po nich se lineárně zvyšují mnohem rychleji. Tyto výsledky nejsou v pořádku a nevypovídají o pravém chování GEGLu. Abych se ujistil o správnosti těchto výsledků, spustil jsem test znovu a získal jsem shodné výsledky. Proto jsem vytvořil druhý test, jehož zadání a vypracování je stejné, jako v předchozím případě. Tentokrát jako dvojice použiji operace *levels* a *invert*. Operace *invert* je implementačně mnohem jednodušší, než *gamma*. Výsledek testu je zobrazen v grafu 6.4 a ukazuje lineární závislost času na počtu vrcholů grafu.



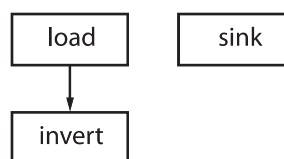
Graf 6.4: Výpočetní složitost dvojic operací levels-invert

Kromě potvrzení, že je čas zpracování lineární, jsem zjistil, že je vysoce závislý na použité operaci.

Test2: Budu zkoumat funkčnost pyramidového modelu pro ukládání mezivýsledků v acyklickém stromu. Použitím pyramid by se měla podstatně zvýšit rychlost výpočtu obrazu. Strom bude tvořit jediná operace *load*, na kterou zavolám dvakrát po sobě *process*. Když jsem poprvé zavolal *process*, byla data načtena do pyramidy. Tato operace trvala nezanedbatelný čas v závislosti na velikosti vstupních dat. Následně jsem zavolal *process* podruhé a čas provedení operace byl nulový (nezměřitelný), protože se data od předchozího volání *process* nezměnila (pyramida je naplněna platnými daty). Tento test dokázal, že pyramidový systém funguje.

Nyní provedu stejný test, jen místo volání metody *process* budu volat *blit*. Jelikož *blit* neumí ukládat data do pyramid, musím poprvé zavolat na vrchol *load* metodu *process*, která uložila data do pyramidy. Následné volání metody *blit* na vrchol *load* proběhne v nulovém čase. Výsledek testu je tedy stejný jako u metody *process*.

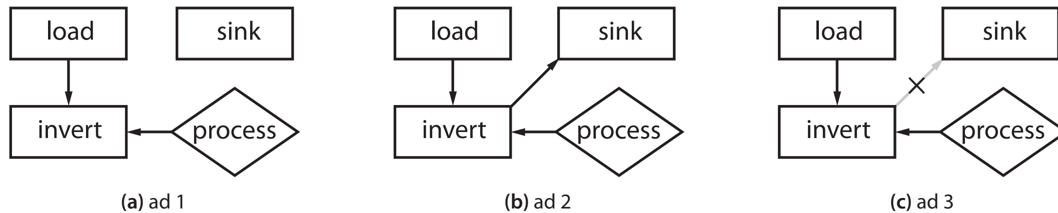
Test3: Udržení dat v pyramidách při změnách spojení stromu operací. Strom a jeho operace je možné přepojovat za běhu programu. Potřebuji vědět, jaká data jsou uchovávána a jaká musí být spočtena znovu. Použiji operace *load*, *invert* a *sink*. Operace *sink* je zkrácená forma jména *buffer-sink*, který funguje jako úložiště dat v paměti počítače. Lze do něj pouze data zapisovat, pro čtení z něj se používá jiná operace. Operace *load* je spojena s *invert*, *sink* je nepřipojený (obr. 6.5). Operací *load* budu načítat fotografii s rozlišením 3200×2400 pixelů ve formátu JPEG. V průběhu testu budu měnit připojení vrcholů grafu, každá změna je zobrazena na schématu 6.6.



Obrázek 6.5: Schéma spojení stromu operací

Postup testování:

1. Poprvé budu volat *process* na operaci *invert*.
2. Připojím výstup z *invert* do vstupu *sink*. Zavolám *process* na *invert*.
3. Odpojím *sink* od *invert* a na *invert* zavolám *process*.



Obrázky 6.6: Změna spojení ve stromě

V bodě 1 došlo k alokaci místa v pyramidě a načtení dat. V druhém bodě došlo k připojení komponenty grafu k volané komponentě, do pyramidy byla nahrána nová data. V bodě 3 došlo k odpojení komponenty od grafu, ale data v pyramidě se nezměnila a čas je nulový (neměřitelný). Test ukázal, že připojením komponenty ke grafu se data uložená v pyramidě stanou neplatná a musí být znovu přepočítána. Odpojení od grafu nevadí.

Test4: Měřím časy vykonávání jednotlivých operací metodou *process*. V předchozích testech jsem zjistil, že ukládání do pyramid funguje. Nyní změřím, kolik času ušetřím jejich používáním. Strom je lineární s pospojovaným řetězem metod. Jeho reprezentace je ukázána na obrázku 6.7. Vrcholy *invert1* a *invert2* vykonávají operaci *invert*. Mají odlišné názvy pro jejich rozpoznání. Operací *load* budu načítat fotografii s rozlišením 3200×2400 pixelů ve formátu JPEG.

Obrázek 6.7: Schéma spojení stromu operací pro metodu *process*

Postup testování:

1. Zavolám *process* na operaci *load*.
2. Zavolám *process* na operaci *invert1*.
3. Zavolám poprvé *process* na *invert2*.
4. Zavolám *process* na *gamma*.
5. Znovu zavolám *process* na *invert2*.
6. Nakonec zavolám *process* na *sink*.

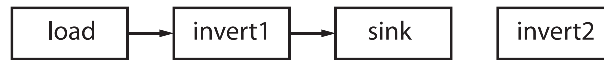
Výsledky testu:

1. Čas pro načtení dat do pyramidy vrcholu *load* neměřím, provedl jsem *process* abych odstranil čas potřebný pro načtení fotografie do paměti.
2. Nad vstupními daty z pyramidy vrcholu *load* byla provedena inverze a došlo k uložení dat do pyramidy *invert1*, čas trvání byl 10 sekund.
3. V pyramidě vrcholu *invert1* byla platná data. Nad těmito daty byla provedena znovu inverze a nová data byla uložena do pyramidy *invert2*, čas trvání byl 8,7 sekundy.
4. Postup je obdobný jako v předchozím bodu s časem 9,7 sekundy.
5. Jelikož byla data stále uchována v pyramidě vrcholu *invert2* a nebyla změněna, nebylo třeba nic počítat ani ukládat. Čas byl nulový.
6. Očekával bych, že byla data z pyramidy vrcholu *gamma* uložena do *sinku*. Ale překvapivě byla operace provedena okamžitě, čas byl neměřitelný. Místo toho, aby byl naplněn *sink*, byl pouze přesměrován jeho ukazatel do paměti na již spočtená data pyramidy vrcholu *gamma*.

Z výsledků testu plyne jedna potíž, totiž velice vysoké časy pro zpracování operací. Hlavní problém tkví v nevhodné implementaci operací, ve kterých není zavedena žádná optimalizace. Taková implementace operací není využitelná pro moje potřeby, pro reálné použití budu muset operace předělat. Optimalizaci operací popisuji v kapitole 7.3.

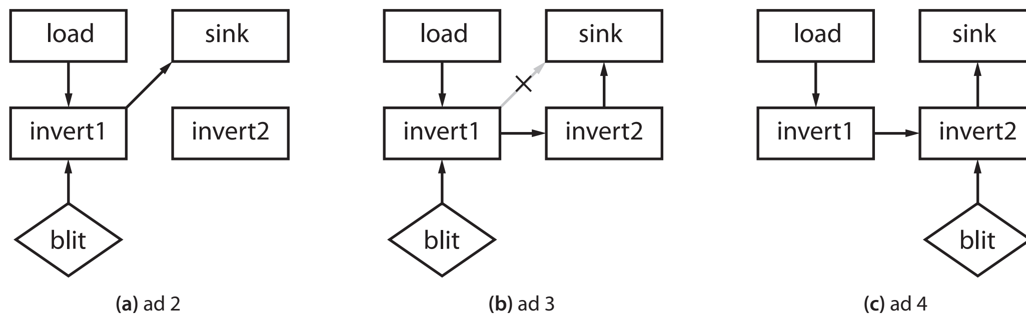
Test5: Měřím časy vykonávání jednotlivých operací metodou *blit*. Ve většině případů není třeba počítat celý obraz. Stačí pouze ta část, kterou má uživatel zobrazenou na monitoru. Tím by operace blit značně

snížila čas zpracování. Strom je lineární s částečně pospojovaným řetězem metod. Jeho reprezentace je ukázána na obrázku 6.8. Používám operace *load*, *invert1*, *invert2* a *sink*. Operací *load* budu načítat fotografii s rozlišením 3200×2400 pixelů ve formátu JPEG. Aby byly výsledky testu srovnatelné s předchozími výsledky, bude *blit* pokrývat celou plochu fotografie.



Obrázek 6.8: Schéma spojení stromu operací pro metodu *blit*

Postup testování. Jeho jednotlivé kroky jsou zobrazeny na obrázku 6.9:
1. Zavolám *process* na vrchol *load*.
2. Zavolám poprvé *blit* na operaci *invert1*.
3. Spojím výstup z *invert1* do vstupu *invert2* a výstup z *invert2* do vstupu *sink*. Původní cesta z *invert1* do *sink* bude tímto zákrokem zrušena. Znovu zavolám *blit* na *invert1*.
4. Nakonec zavolám *blit* na *invert2*.



(a) ad 2 **(b)** ad 3 **(c)** ad 4

Obrázky 6.9: Změna spojení ve stromě pro zpracování metodou *blit*

Výsledky testu:
1. Jelikož *blit* neumí ukládat data do pyramid, použil jsem operaci *process*, abych načetl data do pyramidy.
2. Došlo k alokaci místa v pyramidě vrcholu *invert1*, aplikace inverze na vstupní data a načtení dat, celkový čas byl 7,7 sekundy.
3. Jelikož došlo k připojení vrcholu grafu k volanému vrcholu, byla do pyramidy nahrána nová data. Čas obnovení dat a jejich načtení byl 3,7 sekundy.
4. Bylo třeba od začátku dopočítat celá obrazová data a výpočetní čas byl veliký, celých 10 sekund. Proč byl čas tak veliký? Problém leží přímo v implementaci metody *blit*, která pracuje neoptimálně s daty z pyramid vrcholů stromu.

Výsledky testu dokazují, že *blit* může mít lepší časy, než metoda *process*. Na druhou stranu jeho nedostatečná práce s pyramidami činí jeho použití problematické.

Hodnocení

Velkou nevýhodou GEGLu je neefektivní práce jeho operací. S načtenou fotografií je automaticky pracováno v předurčeném barevném prostoru, který zvyšuje velikost zpracovávaných dat $5{,}\overline{3}\times$ a tím zvyšuje spotřebu paměti a čas na zpracování acyklického stromu operací. Všechny operace pracují pouze v jediném barevném prostoru, do kterého si vstupní data převedou. Samotný převod zpomaluje provádění korekcí. Tyto vlastnosti lze zlepšit přepsáním operací GEGLu a změnou určité části jádra programu, která má na starost konverzi barevných prostorů. Vytvoření aplikace, která využívá změněnou implementaci operací, popisuji v další kapitole.

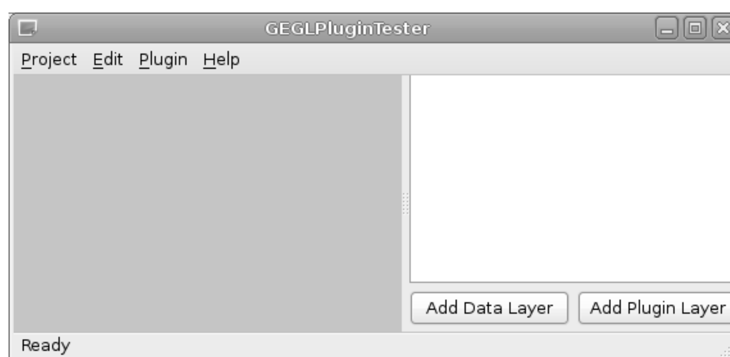
7 PluginTester

Cílem je navrhnout a implementovat aplikaci pro interaktivní testování filtrů pro barevné korekce. Využiji proto znalosti nabyté při práci s knihovnou GEGL a vytvořím program, ve kterém bude možné testovat uživatelsky vytvořené GEGL operace. Základní fakta a požadavky programu:

- Knihovna GEGL je psána v jazyce C. Proto i program bude muset být psán v jazyce z rodiny C.
- Pro GUI použiji platformově nezávislou grafickou knihovnu Qt, která využívá jazyk C++. I když bude vývoj probíhat v Linuxu, platformová nezávislost usnadní vývoj pod Windows.
- Pojem *interaktivní testování filtrů* skrývá několik požadavků. Kromě zřejmého, že uživatel může měnit hodnoty jednotlivých filtrů při běhu programu, může také načítat filtry za běhu programu.
- Zrychlit operace GEGLu.

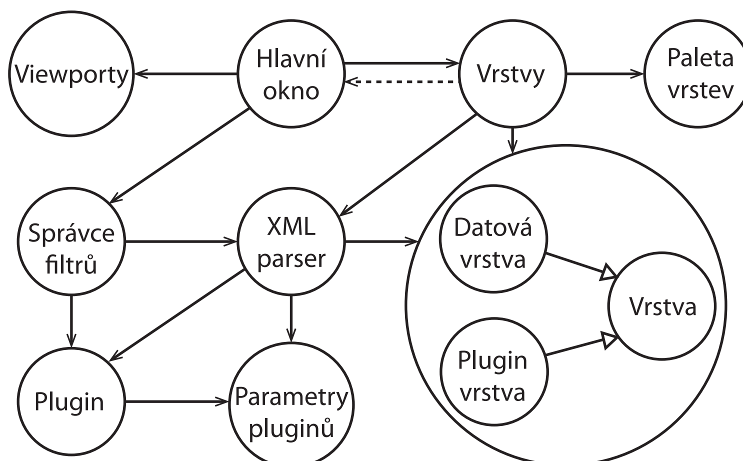
7.1 Třídy programu

První věc, kterou uživatel při spuštění programu uvidí, je *hlavní okno* aplikace s otevřeným novým projektem.



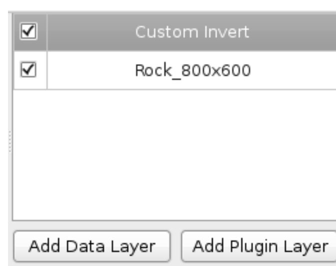
Obrázek 7.1: Hlavní okno aplikace

Okno funguje jako opěrný bod GUI – udržuje instance Správce pluginů a Vrstev. Pro striktní oddělení funkčních částí programu na sebe tyto dvě části „nevidí“ a pokud si potřebují poslat data, provedou tak přes hlavní okno. Posílání je na obrázku 7.2 vyznačeno plnou čarou, příjem čárkovanou. Další funkcí hlavního okna je zobrazování zpracovaných dat z vyrovnávací paměti vrstvy do viewportů.



Obrázek 7.2: Zjednodušené schéma komunikace programu

Pravou stranu plochy obhospodařuje třída Vrstvy. Oblast je rozdělena na dvě části. Vrchní, ve které jsou vrstvy, a spodní s tlačítky pro jejich přidání do palety. Obě tyto grafické části spravuje *Paleta vrstev* (7.3). Vrstvy řídí Paletu vrstev a tvoří datové úložiště pro všechny *Datové vrstvy* a *Vrstvy pluginů*.



Obrázek 7.3: Paleta vrstev s dvěma vrstvami

Datová vrstva udržuje data obrázku načteného z datového úložiště a vrstva pluginu provádí manipulaci se zdrojovými daty. Přidání datové vrstvy do palety provede uživatel stiskem tlačítka *přidat datovou vrstvu* (Add Data Layer) a je možné tak provést okamžitě po vytvoření nového projektu. Datová vrstva je první, kterou uživatel musí vložit do palety vrstev. Dokud ji nevloží, není mu umožněno vložit vrstvu pluginu. Pokud již vrstva dat existuje, může uživatel stiskem tlačítka *přidat vrstvu pluginu* (Add Plugin Layer) vybrat plugin, okolo kterého se vytvoří vrstva pluginu a vloží se do palety.

7.2 Příprava operace

Vlastní plugin je tvořen postupně z více částí. Nejprve uživatel naprogramuje operaci, z té vytvoří XML reprezentaci, kterou načte v programu.

Každá operace obsahuje základní dvě statické metody komunikačního rozhraní. Metoda *process* obsahuje výkonnou část operace. Vstupními parametry jsou *GeglOperace*, z které lze získat uživatelské proměnné použité v operaci, vstupní vyrovnávací paměť, která obsahuje vstupní data do operace, výstupní vyrovnávací paměť, do které se ukládají vypočtené výsledky hodnot pixelů, počet pixelů, které se mají v jedné iteraci vypočítat a nakonec obdélník, který vymezuje oblast obrazu, která je v dané iteraci počítána. Poslední dva parametry jsou přítomné kvůli práci s obrazem. Celý obraz je virtuálně rozdělen na dlaždice o určitém rozměru. Při výpočtu je daná operace prováděna pro každou dlaždici. Toto rozdělení je výhodné pro vícevláknové zpracování obrazu. Druhou metodou je *gegl_chant_class_init*, která obsahuje data o operaci. Hlavní je unikátní **identifikátor operace** ve tvaru *gegl:...*, např. *gegl:color_separation*. Nakonec obsahuje popis funkce operace a do jaké kategorie se řadí.

Přeloženou dynamickou knihovnu uživatel uloží do instalační složky GEGL operací. Aby bylo možné nově vytvořenou operaci v programu načíst, je třeba pro ni vytvořit **konfigurační XML** soubor popisující její chování. Důvod pro jeho vytvoření je, že za běhu programu není možné potřebné informace z operace získat. Příklad takového souboru je ukázán v kapitole 7.7. Formát dat má přesnou následující strukturu:

1. Jméno souboru dynamické knihovny.
2. Název operace zobrazený v GUI.
3. GEGL identifikátor operace.
4. Absolutní, nebo relativní cesta k souboru dynamické knihovny.
5. Zda-li je operace vstupní (export do png), výstupní (generátor obrazových dat) či vstupně-výstupní, které provedou operaci nad vstupními daty a změněné je pošlou dál.

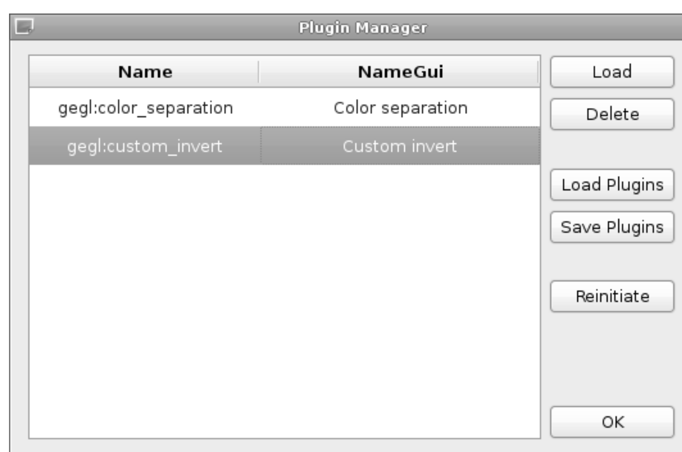
   Následují parametry operace. Každý parametr obsahuje název, kterým se zobrazí v GUI aplikace a stejný identifikátor, jaký je použit v operaci. Pokud žádné parametry neobsahuje (např. operace inverze), je deaktivován příznak dalších parametrů. Existuje sedm typů parametrů, z nichž každý má odlišné detaily.

   (a) *Číslo (Number)* může být celočíselné, nebo s plovoucí desetinou čárkou. Také obsahuje minimální, maximální a počáteční hodnotu parametru.

(b) *Aux* značí vstup dat z jiného vrcholu grafu. Pokud má operace aux, bude mít i běžný vstup dat. Další parametr po aux může mít přidaný příznak *podpora (support)*. Pokud by nebyla aux vyplněna, bude brána hodnota z následujícího parametru a naopak.
(c) *Pointer (Ukazatel)* může ukazovat na struktury, pole, aj.
(d) *Color (Barva)* je provizorní struktura, bude nahrazena objektem GeglColor, implementovaným v následujících verzích knihovny GEGL. První je řetězec označení barevného prostoru takový, který využívá knihovna BABL. Dále je barevná hloubka a nakonec řetězec číselných hodnot oddělených čárkami bez mezer představující jednotlivé hodnoty barvy.
(e) *String (Řetězec)* pro přenos libovolných hodnot oddělených čárkami bez mezer.
(f) *Boolean (Logický)* může nabývat hodnot ano a ne.

7.3 Plugin

Pro vytvoření vrstvy pluginu je třeba mít připravený plugin. Ten může být vytvořen dvěma způsoby. Prvním způsobem je tvorba jako vnitřní plugin aplikace. Tyto pluginy jsou vytvořeny programátorem a jeho vytvoření nemůže uživatel ovlivnit, může ho jen používat. Program obsahuje vnitřní pluginy *inverze* a *světlost a kontrast*. Druhý způsob je načtení nového pluginu vytvořeného uživatelem. Po spuštění programu a otevření nového projektu uživatel aktivuje záložku Plugin→Manager. Zobrazí se panel Správce pluginů (obrázek 7.4).



Obrázek 7.4: Okno správce pluginů s dvěma načtenými pluginy

Stiskem tlačítka Load (Načíst) může uživatel vybrat vytvořený XML soubor s popisem operace a načíst ho. Třída XML parser soubor přečte a vytvoří plugin, který je vložen do Správce pluginů. Pokud by uživatel chtěl vložit plugin se jménem (GEGL identifikátorem), který už ve správci existuje, plugin nebude vložen. Tlačítko Delete (Vymazat) odstraní plugin z programu. Tlačítko Load Plugins (Načíst Pluginy) dokáže načíst ze souboru skupinu pluginů, které do něho byly uloženy tlačítkem Save Plugins (Uložit Pluginy). Tlačítkem OK se okno Správce pluginů zavře. O tlačítku Reinitiate (Znovunačtení) pojednává odstavec 7.4. Zjednodušená struktura pluginu je následující:

1. *GEGL identifikátor* operace, který byl již zmíněn.
2. *GeglNode*, vrchol acyklického grafu, v kterém je udržována instance operace.
3. Všechny *parametry* jsou převedeny do programové podoby a uschovány v poli.
4. Poslední je *pole předpočítaných hodnot*. Má délku 257 bytů. Pozice 0-255 jsou hodnoty pixelů, poslední byte značí, jestli je pole naplněno validními daty.

Při testování operací knihovny GEGL jsem zjistil, že všechny operace jsou velice pomalé a v aktuální podobě nepoužitelné. Operace, které můžou být formou pluginu vloženy do programu, musí být upraveny. V rámci optimalizace rychlosti operací budou všechna data zpracovávána v osmibitovém barevném prostoru RGBA. Aby se při přechodu z 32bitového RGBA (ve kterém implicitně pracuje většina operací)

na osmibitové RGBA nesnížila přesnost provádění operace, musí být její návrh pečlivý. Snížením počtu bitů pro zpracování dojde ke zvýšení výkonu aplikace. Původní algoritmus výpočtu plýtval výpočetním časem pro získání hodnoty každého výstupního pixelu. Příklad z operace *úrovně*:

```
outPixel[x] = (inPixel[x] − inOffset) * outRozsah / inRozsah + outOffset;
```

outPixel a inPixel jsou výstupní/vstupní pole obrazových dat. Zbývající proměnné jsou pouze konstanty a v průběhu výpočtu se nemění. To znamená, že výstupní pixel závisí na jediné proměnné a pro jeho výpočet je třeba použít čtyři matematické operace. Čas potřebný pro výpočet jediné standardní dlaždici velikosti 128×64 pixelů spočtu jako 128×64×4 (kanály RGBA)×čas všech čtyř operací.

Nezanedbatelné zvýšení výkonu poskytuje pole předpočítaných hodnot, které bylo zavedeno pro zvýšení efektivity GEGL operací. Při vytvoření pluginu je pole prázdné. Jakmile je poprvé zavolána metoda operace *process*, je do ní pole posláno odkazem. Kontrola validity dat selže a pole je naplněno validními daty. Pro každou buňku pole je vypočtena hodnota s použitím vstupního pixelu. Pokud bych použil příklad výše:

```
for(int i = 0; i <= 255; i++)
  pole[i] = (i − inOffset) * outRozsah / inRozsah + outOffset;
```

Nakonec nahodím bit validity na pozici 256. Výstupní pole pak plním:

```
outPixel[x] = pole[inPixel[x]];
```

Výsledný čas výpočtu bude v tomto případě 128×64×4 (kanály RGBA)×nalezení hodnoty v matici. Když je dopočítána tato dlaždice, pokračuje se s výpočtem další. Za vstupu se zjistí, že jsou data validní a není třeba pole znovu plnit daty.

7.4 Databáze operací

Jedním z požadavků je načítání pluginů za běhu programu. Příkladem může být vytvoření nové operace, kterou bude chtít uživatel otestovat. Načte plugin do programu, vytvoří posloupnost vrstev a bude pozorovat výsledek. Zjistí, že je třeba operaci pozměnit. Pozmění ji, vytvoří dynamickou knihovnu a bude ji chtít načíst do programu. Jenže to má háček.

Při inicializaci knihovny GEGL, která proběhne jedenkrát před použitím jakékoliv GEGL funkce, je vytvořena databáze všech operací, jejichž soubory se nacházejí v jeho instalační složce. Aby mohly být operace zaindexovány, jsou volány jejich metody *gegl_chant_class_init*, z kterých jsou vybrány požadované informace. V databázi je možné indexovat pouze operace s unikátními identifikátory operací. Jakmile je databáze vyrobena, je pouze možné vytvářet instance načtených operací – přidávání ani mazání operací není povoleno. Proto jsem zkusil GEGL ukončit a inicializovat a tím ho donutit, aby znovu vytvořil databázi operací. Nakonec jsem zjistil, že GEGL není možné v průběhu programu úplně ukončit. I po zavolání metody *gegl_exit*, která ukončí činnost GEGLu a odstraní alokovaný prostor, není uvolněna databáze operací.

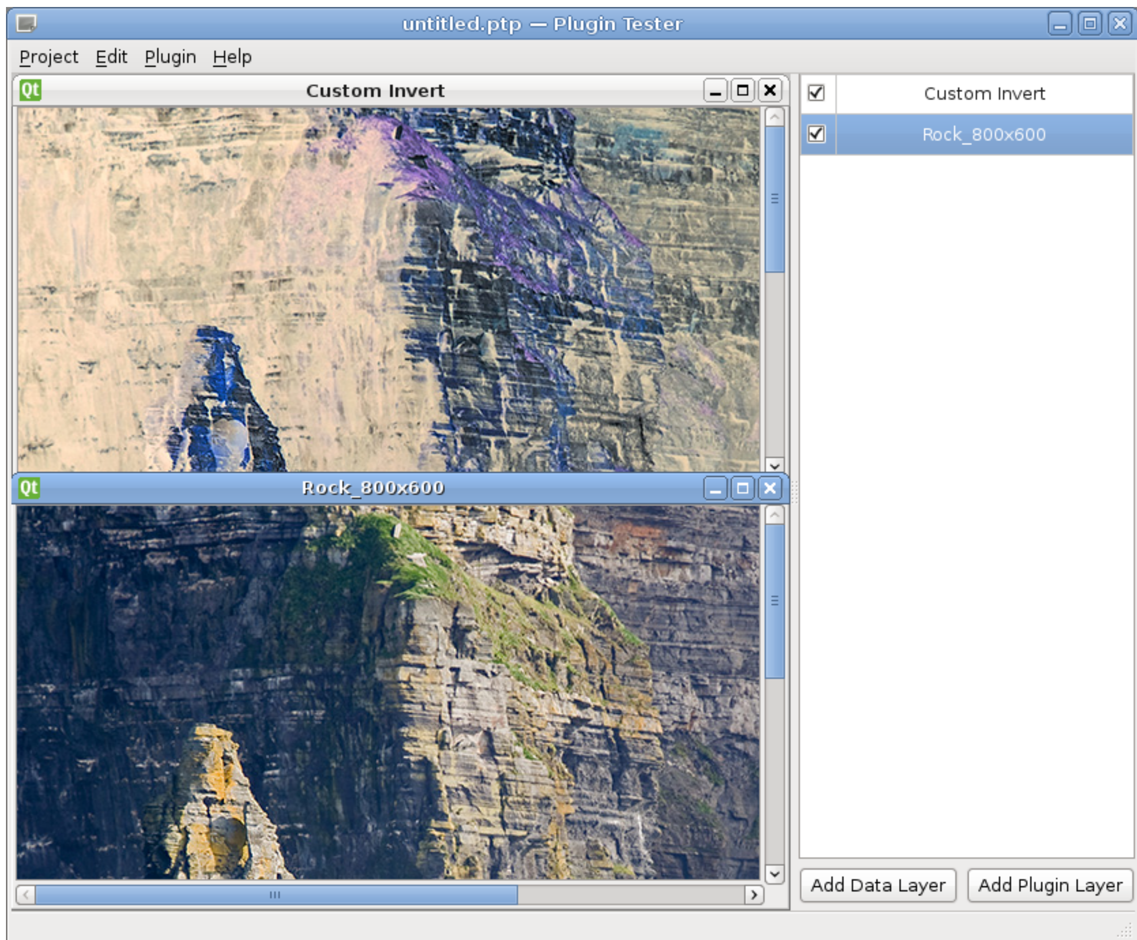
Proto jsem ve Správci pluginů vytvořil tlačítko *Reinitiate (Znovunačtení)*. Jeho aktivací jsou všechny aktuální pluginy a vrstvy uloženy do dočasných souborů a program restartován. Restart je pouze zdánlivý. Do statické proměnné je uložen příznak reinicializace a po sobě se provedou dvě programové operace – spuštění programu od začátku a ukončení stávajícího. Zpětně jsou načtena všechna data z dočasných souborů a nově změněná uživatelská operace je načtena s novým načtením databáze operací GEGLu. Nevýhodou této operace je ztráta dočasně uložených dat vrstev, ale protože je aplikace zaměřena pouze na testování, tato ztráta není podstatná.

7.5 Vrstvy

Nyní již má uživatel vše připravené pro vytvoření vrstev. Každá vrstva obsahuje identifikátor vrstvy použitý ve chvíli, když je otevřeno více oken s nastavením vrstev, zaškrtávací políčko pro změnu viditelnosti

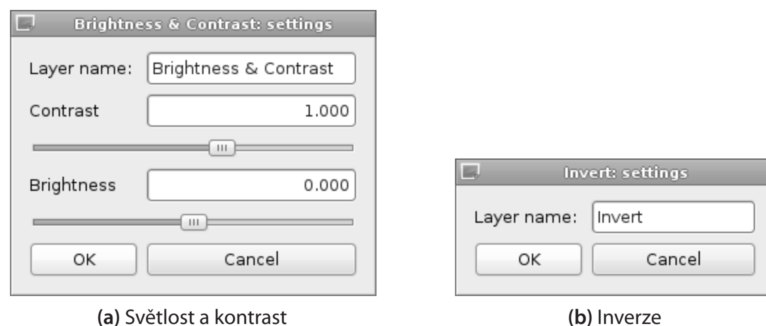
vrstvy (pokud je políčko zaškrtnuto, vrstva je viditelná) a název vrstvy. Dále každá vrstva obsahuje dočasnou paměť, do které jsou uložena obrazová data. Funkci této paměti ukážu na příkladu.

Použiji příklad z obrázku 7.3. Uživatel vložil datovou vrstvu, ve které vybral obrázek Rock_800×600. Data obrázku se načtou do dočasné paměti vrstvy, z kterých jsou automaticky zobrazena na plochu. Pro datové vrstvy se stává tato dočasná paměť trvalou, mají životnost vrstvy. Na tuto vrstvu aplikoval vrstvu úprav *Custom invert*, která je pouze upravená verze běžné metody inverze. Paleta vrstev funguje jako zásobník se dnem dole, což znamená, že nově přidané vrstvy budou přibývat na vrchol palety vrstev. Pokud uživatel klikne pravým tlačítkem myši na vrstvu inverze, objeví se kontextové menu s nabídkou akcí. Na výběr je *Show Viewport (Zobraz Viewport)* a *Delete Layer (Smaž vrstvu)*. Pokud vybere zobrazení, algoritmus vezme data z datové vrstvy a aplikuje na ně inverzi. Výsledek zobrazí na plochu. Obrázek 7.5 ukazuje výsledek.



Obrázek 7.5: Zobrazení datové vrstvy a vrstvy pluginu

Pokud by uživatel přidal na vrchol zásobníku další vrstvu pluginu *světlost a kontrast*, mohl by měnit světlost a kontrast předchozí vrstvy, inverze. Dvojklikem myši na libovolnou vrstvu se zobrazí paleta s nastavením vrstvy. Paleta je vytvořena dynamicky při vložení vrstvy do palety vrstev. Data pro její tvorbu jí poskytují parametry filtru. Datové vrstvy můžou pouze změnit název vrstvy, vrstvy pluginů mají různé palety nastavení podle použitých parametrů. Na obrázku 7.6 je vidět rozdíl mezi vrstvami pluginů *světlost a kontrast* a *inverze*.



(a) Světlost a kontrast

(b) Inverze

Obrázek 7.6: Palety nastavení pluginů

7.6 Algoritmus výpočtu

Pokud uživatel použije akci *Show Viewport (Zobraz Viewport)*, provede program následující kroky k výpočtu výsledného obrazu.

1. Algoritmus začíná od nejvyšší vrstvy a postupuje směrem dolů, dokud nenajde vrstvu, která provedla volání.
2. Od této vrstvy postupuje dolů a hledá vrstvu, ze které by mohl získat obrazová data. Taková vrstva může být každá vrstva dat, či vrstva pluginů, která má naplněnou dočasnou paměť a data jsou validní (neproběhla změna posloupnosti vrstev od posledního naplnění dočasné paměti). Také jsou vynechávány všechny neviditelné vrstvy.
3. Nalezená zdrojová data jsou převedena na zdrojový vrchol acyklického GEGL grafu.
4. Od této vrstvy postupuje algoritmus **zpět**, směrem ke startovní vrstvě. Každou vrstvu (pluginu), kterou tímto způsobem najde, připojí k již nalezené části grafu. Jako v předchozím případě vynechává neviditelné vrstvy. Cesta algoritmu končí, až se dostane do počáteční vrstvy.
5. Zde vytvoří dočasnou paměť a naplní ji daty, které získá zavoláním metody *process* pro výpočet obrazu z acyklického grafu.
6. Zbývá jen data zobrazit použitím metod knihovny Qt. Zobrazení je ale zpomaleno, jelikož knihovna Qt používá ve všech svých komponentách barevný prostor ARGB. Aby bylo možné data zobrazit, musím je převést z prostoru RGBA do ARGB.

7.7 Nový plugin

Pro testování algoritmu výpočtu jsem vytvořil operaci *světlost a kontrast*. Obecný popis jsem již poskytl v sekci 7.2, a proto vypíšu jen informace specifické pro tuto operaci. Proměnné, s kterými pracuje, jsou *brightness (světlost)*, *constrast (kontrast)* a pole předpočítaných hodnot. Třída operace je *bodový filtr*, pracuje s jednotlivými pixely a nevyužívá jejich okolí. Metoda *prepare (připrav)* nastaví konverzi vstupních i výstupních dat do formátu nelineárního RGBA s osmibitovou hloubkou zobrazení. V metodě *process* je naplněno pole předpočítaných hodnot:

```
for (i = 0; i <= 255; i++)
   pixelValue = (((i — 128.0) * contrast) + (brightness * 255.0) + 128);
```

Počáteční hodnota brightness je nula, contrast je jedna. Přesto, že jsou vstupní data osmibitová (unsigned char), je třeba kvůli operaci (i - 128.0) zvýšit rozsah proměnné pixelValue. Pokud je hodnota pixelValue záporná, uloží se nula, pokud překračuje 255, uloží se 255. V ostatních případech se uloží celočíselná vypočtená hodnota. Po naplnění pole je nahozen příznak validity dat. V další části se podle hodnot vstupních pixelů vyhledává v předpočítaném poli a ukládá se do výstupního pole. Operace nepočítá s alfa kanálem, jeho hodnoty jsou pouze překopírovány do výstupního pole. Aby byl plugin kompletní, je třeba k operaci vytvořit konfigurační XML soubor, jehož struktura byla popsána v kapitole 7.2 a má následující podobu:

```
1   <?xml version="1.0" encoding="UTF-8"?>
2   <operation xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
3
4       <pluginFileName>brightness-contrast.so</pluginFileName>
5
6       <nameGui>Brightness & Contrast</nameGui>
7
8       <nameOperation>gegl:brightness-contrast</nameOperation>
9
10      <pathToPlugin mode="relative">./config</pathToPlugin>
11
12      <inputOutput hasInput="true" hasOutput="true" />
13
14      <params>
15
16          <param nameGui="Contrast" nameOperation="contrast">
17              <details type="number">
18                  <numberType>Double</numberType>
19                  <minRange>-5</minRange>
20                  <maxRange>5</maxRange>
21                  <defaultValue>1</defaultValue>
22              </details>
23          </param>
24
25          <param nameGui="Brightness" nameOperation="brightness">
26              <details type="number">
27                  <numberType>Double</numberType>
28                  <minRange>-3</minRange>
29                  <maxRange>3</maxRange>
30                  <defaultValue>0</defaultValue>
31              </details>
32          </param>
33
34      </params>
35  </operation>
```

8 Závěr

Velká část návrhu struktury programu pro barevné korekce je založena na hlavním prvku GUI, časové ose. Z jejího návrhu byly odvozeny všechny ostatní prvky, zejména algoritmus pro výpočet obrazu. Kritickým parametrem pro běh programů pro barevné korekce je rychlost odezvy na podněty uživatele. Už velice nízké časy v řádech desítek milisekund mohou rozhodnout o jeho použitelnosti. Aby bylo možné dodržet tato přísná kritéria, bude třeba podřídit vše rychlosti aplikace.

Testoval jsem grafické knihovny, které by značně zrychlily vývoj aplikace a snížily množství práce. Jejich použití by bylo na pracné operace (převody do barevných prostorů) a základní operace (načtení a export souborů různých formátů). První knihovna, VIPS, má pokročilé funkce a dokáže pracovat s velkými obrázky, avšak pro interaktivní operace s obrazem nemá uzpůsobenou datovou strukturu ani rychlost odezvy. Knihovna GEGL (a BABL) některé z nevýhod odstraňují, ale další přidávají. Použitím pyramidového systému a acyklického grafu je pro program příznivé. Operace ale nejsou vytvořeny pro více kanálů a jsou pomalé.

Programem pro testování pluginů (operací) GEGLu jsem vytvořil nástroj pro snadné ozkoušení funkčnosti pokročilých grafických operací. Zároveň jsem se snažil zjistit, kolik úsilí by stála úprava GEGLu, aby mohla být knihovna použita v programu pro barevné korekce. Zpracování operace jsem zrychlil optimalizací algoritmu a použitím pole předpočítaných hodnot. Snadno jsem rozšířil operace i pro práci s vícekanálovými prostory. Konečné rozhodnutí ale padlo v neprospěch knihovna GEGL. Pro to, aby byla použita pro program barevných korekcí, by bylo třeba značně změnit její kód. Jednodušším řešením proto bude napsat si vlastní kód.

Přehled zkratek

GUI	Graphical User Interface; grafické uživatelské rozhraní
EXIF	Exchangeable Image File Format; formát metadat, obsahuje informace o obrázku
CPU	Central Processing Unit; procesor počítače
API	Application Programming Interface; rozhraní pro programování aplikací
ICC profil	standard pro zobrazení barev obrázku vytvořený mezinárodním konsorciem
JPEG	formát souboru se ztrátovou kompresí

Reference

[Cup08] John Cuppit. *Postup kompilace VIPSu na operačním systému Windows pro starší verze knihovny [online].* URL: http://www.vips.ecs.soton.ac.uk/index.php?title=Build_on_windows&oldid=2122, únor 2008.

[Ken10] J. Kenneally. *Informace pro kompilace VIPSu na operačním systému Windows [online].* URL: https://www.jiscmail.ac.uk/cgi-bin/webadmin?A2=ind1010&L=VIPSIP&D=0&P=51, říjen 2010.

[Mar07] Dan Margulis. *Professional Photoshop: The Classic Guide to Color Correction.* Peachpit Press, Berkeley (California), 5th edition, 2007.

[Sko09] Věra Skorkovská. *Systém pro barevné korekce digitálních fotografií.* Bakalářská práce, Západočeská universita, Fakulta aplikovaných věd, Plzeň, 2009.

[Zem10] Jan Zeman. *Barevné korekce obrazu a videa.* Bakalářská práce, Západočeská universita, Fakulta aplikovaných věd, Plzeň, 2010.

Seznam obrázků

2.1 Křivka *b* barevného prostoru Lab . . . 2
2.2 Ztráta obrazových dat při provedení expozice . . . 3
3.1 Příklad časové osy obsahující všechny komponenty . . . 5
3.2 Hlavní okno aplikace Dikobraz s časovou osou a jedním viewportem. . . . 6
3.3 Příklad stopy se všemi parametry . . . 6
3.4 Příklad vrstvy úprav se všemi parametry . . . 7
3.5 Příklad kombinace masek . . . 8
3.6 Příklady datových vstupů a výstupů vrstev . . . 8
3.7 Př. Tvorba sekvencí . . . 9
3.8 Ukázka vrstvy úprav obsahující filtry . . . 11
3.9 Druhy interpolací . . . 11
3.10 Režim prolnutí dvou vrstev s nastavením průhlednosti . . . 12
3.11 Ukázka blend-if posuvníků pro odstíny šedi z aplikace Adobe Photoshop . . . 13
3.12 Použití klíčových snímků pro změnu viditelnosti přechodu v čase . . . 14
3.13 Více ukazatelů zobrazujících několik různých míst časové osy . . . 14
3.14 Koncept historie . . . 15
3.15 Princip pyramidového modelu . . . 16
3.16 Použití pyramid v časové ose . . . 17
3.17 Ukázka překrývání vrstev . . . 17
3.18 Příklad budoucí časové osy . . . 18
4.1 Strom funkcí pro zkrácení identifikátorů . . . 18
4.2 Příklad pracovní plochy se třemi zdrojovými vrstvami fotografií . . . 19
4.3 Vložení videa do časové osy . . . 20
4.4 Vložení vrstvy na pozici aktivního (hlavního) ukazatele . . . 21
5.1 Ukázkový příklad jako průvodce zdrojovým kódem . . . 22
5.2 Uživatel vybere vrstvu, kterou chce zobrazit . . . 23
5.3 Je nalezena vrstva pro zpracování a nastaven barevný prostor . . . 23
5.4 Nalezení vstupu dat ke zpracování . . . 24
5.5 Zpracování zdrojové vrstvy . . . 24
5.6 Aplikace filtrů a průběh interakce mezi daty . . . 25
5.7 Zpracování přechodu . . . 26
6.1 Příklad vytvořeného acyklického grafu operací . . . 30
6.2 Dva stromy pro jednu a dvě dvojice operací *levels* a *gamma* . . . 31
6.3 Výpočetní složitost dvojic operací levels-gamma . . . 31
6.4 Výpočetní složitost dvojic operací levels-invert . . . 32
6.5 Schéma spojení stromu operací . . . 32
6.6 Změna spojení ve stromě . . . 33
6.7 Schéma spojení stromu operací pro metodu *process* . . . 33
6.8 Schéma spojení stromu operací pro metodu *blit* . . . 34
6.9 Změna spojení ve stromě pro zpracování metodou *blit* . . . 34
7.1 Hlavní okno aplikace . . . 35
7.2 Zjednodušené schéma komunikace programu . . . 35
7.3 Paleta vrstev s dvěma vrstvami . . . 36
7.4 Okno správce pluginů s dvěma načtenými pluginy . . . 37
7.5 Zobrazení datové vrstvy a vrstvy pluginu . . . 39
7.6 Palety nastavení pluginů . . . 40

Seznam tabulek

3.1 Datová struktura stopy . . . 7
3.2 Datová struktura vrstvy . . . 10
3.3 Datová struktura filtru . . . 12
3.4 Datová struktura parametru filtru . . . 12

3.5 Datová struktura parametrů interakce 13
3.6 Datová struktura klíčového snímku 14

Příloha A – Identifikátory

0001	Stopa přidána
0002	Stopa odebrána
0003	Stopa přesunuta
0004	Stopa přejmenována
0005	Stopa – přidat masku
0006	Stopa – změněn barevný prostor
0007	Stopa – změna parametrů interakce
0008	Stopa – změněn vstup
0009	Stopa – aktivní on/off
0010	Vrstva přidána (je zvolen její typ)
0011	Vrstva odebrána
0012	Vrstva přesunuta
0013	Vrstva přejmenována
0014	Vrstva – přidat masku
0015	Vrstva – změna velikosti
0016	Vrstva – zarážka on/off
0017	Vrstva – změna parametrů interakce
0018	Vrstva – změna parametrů přechodu
0019	Vrstva – změněn barevný prostor
0020	Vrstva – změněn vstup
0021	Vrstva – aktivní on/off
0022	Vrstva – změna vstupního souboru
0023	Vrstva – změna začátku a konce vstupních dat
0024	Vrstva – změněn snímek označující start vstupních dat
0025	Vrstva – změna parametrů dekomprese
0026	Vrstva – změna pozice na pracovní ploše
0027	Keyframe přidán
0028	Keyframe odebrán
0029	Keyframe přesunut
0030	Keyframe – změněny parametry interakce
0031	Filtr přidán
0032	Filtr odebrán
0033	Filtr – změněny parametry filtru (vlastní data & parametry interakce)
0034	Filtr – aktivní on/off
0035	Viewport přidán (přidá se playhead)
0036	Viewport odebrán (odebere se playhead)
0037	Playhead posunut
0038	Playhead – nastavena stopa k sledování
0039	Interpolátor změněn