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

Bakalářská práce

Tradiční barevné korekce obrazu

Plzeň 2011

Jan Rabušic

Prohlášení

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

V Plzni dne 8. května 2011

Jan Rabušic

Poděkování

Na tomto místě bych chtěl poděkovat Ing. Petrovi Lobazovi za čas věnovaný konzultacím a za cenné rady, bez kterých by tato práce nemohla vzniknout.

Abstract

The aim of the work is to find efficient ways for basic color correction tools implementation (curves, hue/saturation, 3-way filter). The work includes analysis of the tools, their simple implementation and efficient implementation for different bit depths.

Obsah

1 Úvod **1**
1.1 Proč práce vznikla . . . 1
1.2 Přehled práce . . . 1

2 Barevné filtry: principy, užití, variace **2**
2.1 Křivky . . . 2
2.2 Odstín a sytost . . . 4
2.2.1 Co je odstín a sytost? . . . 4
2.2.2 Odstín . . . 6
2.2.3 Sytost . . . 8
2.3 3-Way Filter . . . 16
2.4 Analýza výsledků nástrojů sytosti . . . 18

3 Základní implementace **21**
3.1 Křivky . . . 21
3.2 Odstín a sytost . . . 23
3.2.1 Míchání kanálů . . . 24
3.2.2 Odstín a sytost . . . 24
3.2.3 Živost . . . 25
3.3 3-Way Filter . . . 26

4 Rozšířená implementace **28**
4.1 Křivky . . . 29
4.2 Sytost (mícháním kanálů) . . . 29
4.3 Odstín a sytost (nad prostorem HSL) . . . 30
4.4 Živost (nad prostorem HSL) . . . 34
4.5 3-Way Filter . . . 34
4.6 Další bitové hloubky . . . 35
4.7 High Dynamic Range . . . 35
4.8 Porovnání výsledků . . . 37

5 Závěr **39**

A Příloha **43**

B Příloha **47**

C Příloha **51**

1 Úvod

1.1 Proč práce vznikla

Tato bakalářská práce vyplynula ze závěru samostatné práce na Projektu 5 (Tradiční barevné korekce obrazu a videa), kterou jsem v prosinci roku 2010 odevzdal a je k této práci přiložena. Myšlenka, která stála na počátku tohoto tématu, byla prostá. Navrhnout a implementovat modul pro daný typ barevné korekce. Cílem tedy bylo psát pluginy pro existující aplikace (Adobe Photoshop, Virtualdub, apod.). Ukázalo se však, že SDK je buď extrémně komplikované, nebo poskytuje málo možností, nebo že platí obojí. Stávající software (low-end i high-end) pak trpí různými neduhy, proto byl cíl tohoto tématu pozměněn. Novým cílem je tedy vytvoření systému nového, založeného na unikátní architektuře. Výsledkem Projektu 5 mělo být rozhodnutí, zda lze pro napsání nového softwaru využít již existující open-source knihovnu GEGL. Během jejího testování bylo zjištěno několik zásadních neduhů, které vedly k závěru, že tato knihovna, ač byla ze všech známých open-source alternativ nejlepší, není pro naše účely vhodná, a že proto bude nejlepší vystavět celý software od základů. Tato práce se zabývá tvorbou filtrů základních barevných korekcí. V době implementace nebyla dosud k dispozici použitelná verze obslužného programu (který je ve fázi vývoje), na němž by se daly filtry testovat. Bylo tedy nutné k samotným filtrům napsat i obslužnou rutinu (včetně načítání a ukládání obrazových dat) a díky tomu se jedná o samostatně fungující program.

1.2 Přehled práce

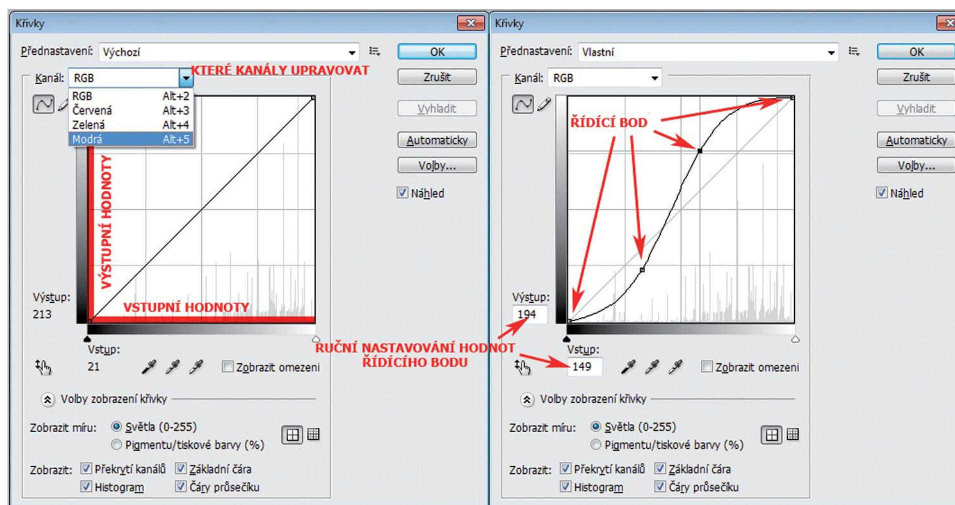
V první části této práce se zabývám principy, užitími a variacemi barevných filtrů Křivky, Odstín a sytost a 3-Way Filteru. Současně se také zmíním o výsledcích v různých barevných prostorech, respektive o tom, jaké vlastnosti barevných prostorů lze pro naše účely využít. Následující část vychází z první. Na základě rozhodnutí, jakými přístupy budu filtry realizovat, rozeberu, jak tyto přístupy implementovat "hrubou silou", neboli bez snahy využité algoritmy urychlit. V poslední části popíšu, jakými postupy lze tuto základní implementaci urychlit. Uvedu rozdíl v časové náročnosti urychlených a neurychlených implementací a implementací v různých bitových hloubkách obrazu. Rozeberu i myšlenku provádět výpočet v plovoucí desetinné čárce. Nakonec uvedu vizuální výsledky všech filtrů.

2 Barevné filtry: principy, užití, variace

2.1 Křivky

Křivky[1] jsou jedním z nejzákladnějších barevných filtrů vůbec. Pro profesionální úpravu fotografií jsou naprostou nezbytností. Jejich zásadní výhodou je jednoduchost a zároveň, díky aplikovatelnosti na většinu barevných prostorů, široká využitelnost.

Nejprve se podíváme, jak se s tímto nástrojem pracuje v jiných aplikacích. Tento nástroj, který naleznete v nejrůznějších softwarech sloužících k úpravě obrazu či videa, předvedu v Adobe Photoshop CS4. Adobe Photoshop je bitmapový grafický editor pro tvorbu a úpravy bitmapové grafiky (např. fotografií), vytvořený firmou Adobe Systems[1]. Ačkoliv se se jedná o placený software a nástroj křivky naleznete i ve freeware alternativách (např. GIMP), některé z dalších nástrojů (například živost) v jiných aplikacích nenaleznete vůbec, nebo nefungují tak, jak bychom si představovali. Další skutečností je, že freeware alternativy nejsou v mnohých směrech ideální (rychlost, uživatelské prostředí). Z těchto důvodů jsou všechny nástroje předvedeny právě v Adobe Photoshop.



Obr. 2.1: Nástroj křivky v Adobe Photoshop CS4.

Předvedení tohoto nástroje proběhne v barevném prostoru RGB, nicméně jeho univerzálnost tkví v aplikovatelnosti na jakýkoliv barevný prostor. V barevném prostoru RGB

[1] anglicky Curves

můžeme s jeho pomocí upravovat celé barevné spektrum současně (jasová korekce), nebo pouze jednotlivé kanály (barevné korekce). Spustíme-li tento nástroj, objeví se okno s mnoha volbami (obr. 2.1). Nalezneme zde například volbu, které kanály se budou upravovat, a spoustu dalších méně podstatných voleb. Hlavní částí tohoto okna je paleta, ve které je vykreslena aktuální křivka filtru. V základním stavu je pouze úhlopříčkou čtvercové palety. Barevných úprav lze docílit nadefinováním bodů, kterými musí křivka procházet. Říkejme jim řídící body. Tímto postupem měníme tvar křivky a tím i obraz samotný.

Princip filtru Křivky je tedy takový, že pro kanál, který chceme upravit, existuje křivka v 2D. Funkce, která ji definuje, vstupním hodnotám jednoznačně přiřazuje výstupní hodnoty. Pokud neprovedeme žádnou barevnou úpravu, bude rovnice této funkce $f(x) = x$. Je snad zřejmé, že křivky nepoužíváme tak, že bychom nadefinovali přechodovou funkci, nýbrž tak, že si řídícími body vytvarujeme křivku podle našich potřeb. Z těchto řídících bodů je funkce (předpis křivky) interně dopočtena.

Podle známých bodů můžeme určit křivku aproximací, nebo interpolací. Z výše uvedeného představení tohoto nástroje je viditelné, že křivka prochází zadanými body, tím pádem aproximace nepřichází v úvahu. Zajisté by šlo tvarovat křivku i aproximačně (vypočítávat například Bézierovy křivky), ale všude, kde jsem se s nástrojem křivky setkal, byly používány interpolační křivky. Zajisté to má praktické opodstatnění. Pokud by byly používány aproximační křivky, potřebovali bychom mít k nadefinování řídících bodů paletu větší, než jaká stačí pro využitelnou část křivky, jinak bychom byli v tvaru křivky omezeni. Abychom získali rovnici funkce (křivky), musíme se rozhodnout, jakou metodou budeme body interpolovat. Nejjednodušší možností je lineární interpolace. Díky ní dostaneme po částech spojitou křivku (lomená čára). Tato křivka má specifické využití jako nastavení white point a black point obrazu nebo třeba pro zvýraznění barev v Lab barevném prostoru. V praxi je užita pouze ve své nejjednodušší variantě, která je definována pouze dvěma řídícími body. Určitě by nebylo obtížné takto interpolovat i pro více řídících bodů, ale v praxi tento přístup nemá smysl, protože by v místě řídících bodů docházelo k viditelným zlomům (způsobeným právě nespojitostí křivky) a výsledek by za mnoho nestál.

Mnohem zajímavějším přístupem je nelineární interpolace. Díky ní vzniká spojitá křivka. Takto vzniklá hladká křivka je velmi hojně využívána a slouží k úpravě jasu a kontrastu (na RGB), sytosti (na a, b kanály prostoru Lab) a k mnohým dalším úpravám. V praxi používané nástroje přecházejí od lineární interpolace k nelineární při počtu řídících bodů 3 a více[2]. Nelineární interpolace lze dosáhnout různými přístupy (Lagrangeova, Newto-

[2]Pamatujme, že první dva jsou přidány okamžitě při spuštění nástroje, určují počátek a konec a nelze je odebrat.

nova, Spline interpolace). Otázkou tedy je, jakou z uvedených interpolací využít. Odpověď na tuto otázku jsem nalezl u svého předchůdce Jana Kučery v jeho bakalářské práci[2], v níž se právě touto otázkou zabýval. Lagrangeova interpolace má nevýhodu v tom, že by bylo nutné při přidání bodu přepočítat všechny polynomy a jejich následnou sumu. Newtonova interpolace sice řeší problém přidání bodu a je poměrně snadno implementovatelná, po porovnání vlastností se spline interpolací a po prozkoumání nástroje křivky v několika programech je na 99,99% jasné, že se využívá spline. Následně Jan Kučera zkouší implementace obou a v porovnání s jinými grafickými editory nachází jednoznačnou shodu ve spline.

2.2 Odstín a sytost

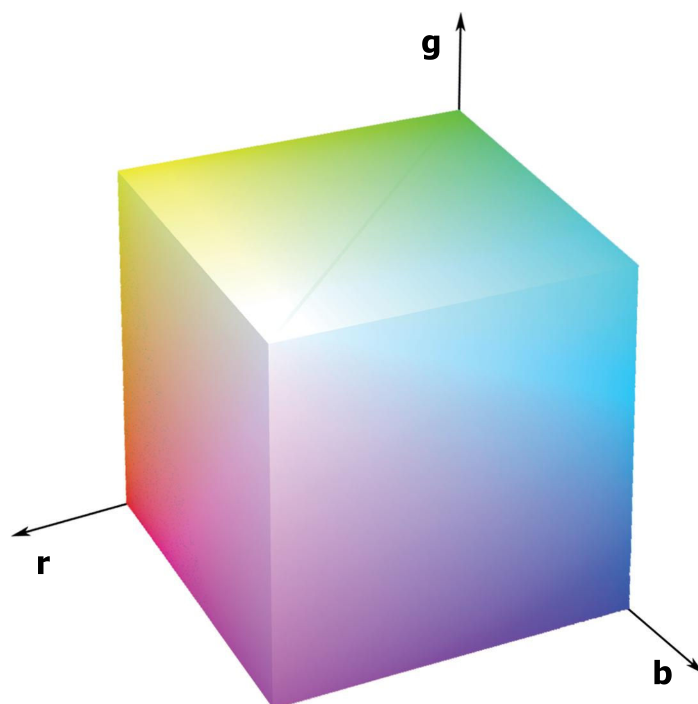
2.2.1 Co je odstín a sytost?

Nejprve by nás mělo zajímat, co to je odstín a sytost. Jsou to spolu s jasem základní vlastnosti barev, s jejichž pomocí lze tyto barvy jednoznačně a zároveň lidskému vnímání nejpřirozeněji popsat. Podle dvojí interpretace jasu pak mohou z těchto tří složek vzniknout dva různé barevné modely HSV (respektive HSB) a HSL. Vysvětlení na těchto modelech kombinuje lidské chápání a počítačovou interpretaci.

Odstín

Odstín má u obou těchto modelů stejný význam a je technicky specifikován v CIECAM02 modelu: Stupeň, se kterým lze daný barevný vjem přirovnat jako podobný, či rozdílný od vjemu, vyvolanému červenou, modrou, zelenou a žlutou barvou[3].

Vysvětlení této definice na modelu HSV je poměrně jednoduché. Model HSV vychází z modelu RGB. Představme si RGB krychli, kde jsou jednotlivé kanály vyjádřeny na osách r, g, b (obr. 2.2), její vrcholy tedy odpovídají černé, červené, zelené, modré, žluté, azurové, purpurové a bílé. Pokud tuto krychli otočíme tak, že černý i bílý bod leží na svislé ose, a zároveň tento útvar promítneme do roviny kolmé k této ose, dostaneme pravidelný šestiúhelník. Ve vrcholech tohoto šestiúhelníku nalezneme červenou, žlutou, zelenou, azurovou, modrou a purpurovou barvu. Odstín si pak můžeme představit jako středový úhel šestiúhelníku, přičemž úhel $0°$ odpovídá červené, $60°$ žluté atd.

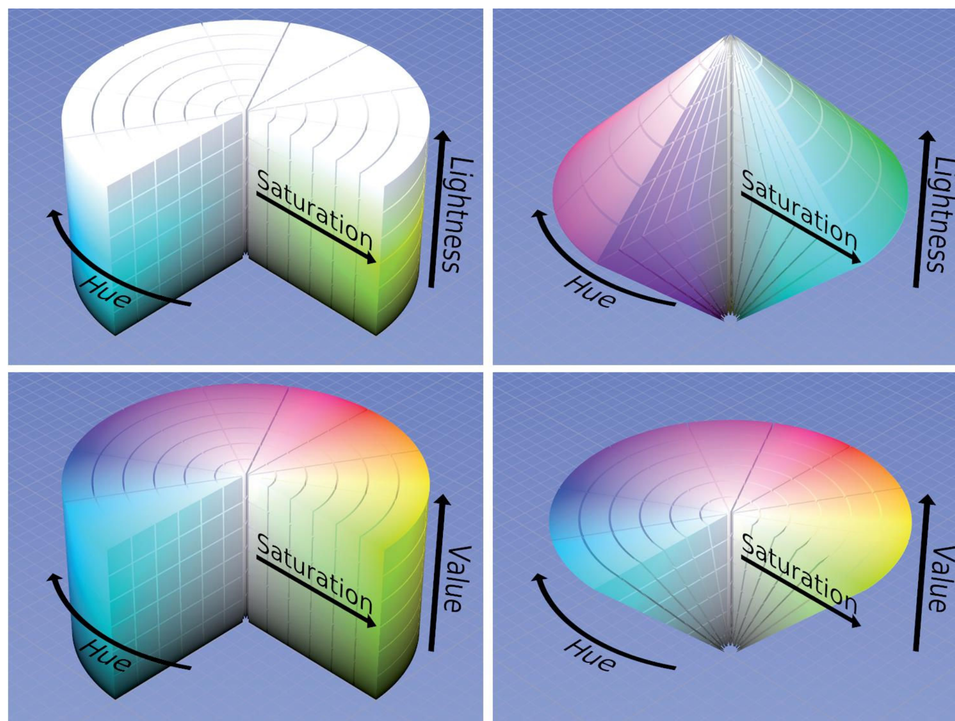


Obr. 2.2: RGB krychle.

Z hlediska vnímání lidského oka je odstín dán vlnovou délkou barvy (či směsí vlnových délek), která dopadá na sítnici. Odstín v celém rozsahu tak můžeme přirovnat k barevnému spektru.

Sytost

Sytost má opět stejný význam v modelech HSV i HSL (obr. 2.3). Je to míra, která o vzorku udává jeho čistotu barvy. Pokud je barva směsí více barev různých vlnových délek a žádná z nich není dominantní, naše oko ji bude vnímat jako šedotónovou a její sytost bude 0 (respektive 0%). Pokud bude barva obsahovat pouze světlo jedné vlnové délky, nebo budou intenzity ostatních barev vůči této zanedbatelné, budeme jí vnímat jako čistou barvu a její sytost bude 1 (respektive 100%). Pokud bychom chtěli zůstat u převodu z RGB do HSV, pak je sytost přímo úměrná rozdílu intenzit mezi nejintenzivnějším a nejméně intenzivním kanálem RGB barvy a zároveň nepřímo úměrná jasu. Modely HSV a HSL se liší právě ve výpočtu jasu. V modelu HSV jas odpovídá intenzitě nejintenzivnějšího kanálu RGB, kdežto v modelu HSL odpovídá zprůměrování nejintenzivnějšího a nejméně intenzivního kanálu RGB. Díky této rozdílné interpretaci jasu se mohou lišit sytosti vyjádřené v HSV a HSL, byť mají stejný význam.



Obr. 2.3: Barevné modely HSL (nahoře) a HSV ve válcovité a kuželovité interpretaci.

Pohled na filtr, upravující odstín a sytost zároveň, není tak zajímavý jako možnost rozebrat zvlášť přístupy úpravy odstínu a úpravy sytosti.

2.2.2 Odstín

Odstín a sytost (Hue and Saturation)

Hlavní filtr, který umožňuje úpravu odstínu a sytosti v Adobe Photoshop, je filtr Odstín a sytost[3]. Ten by se měl výstižněji jmenovat Odstín, sytost a světlost, jelikož právě tyto tři hodnoty lze jeho pomocí měnit. Jak je filtr Odstín a sytost v Adobe Photoshop realizován? Z výše uvedeného představení barevných prostorů HSV a HSL je pravděpodobné, že se obraz při zavolání tohoto filtru převede interně (tak, aby o tom uživatel nevěděl) ze stávajícího barevného prostoru do jednoho z těchto prostorů a současně uživatel dostane k dispozici okno tohoto nástroje. V tomto nástroji může uživatel posunovat třemi jezdci odpovídajícími třem kanálům prostoru HSV (HSL). První z nich je jezdec odstín. Uživatel nastaví hodnotu od -180 do +180, což velmi úzce souvisí s významem kanálu

[3] v anglické verzi Hue and saturation

Hue (ten totiž představuje úhel 0° – 360°, podrobněji bude popsáno definicí v návrhu filtrů, kapitola 3.2.2).

Obr. 2.4: Odstín pomocí nástroje odstín a sytost.

Vyvážení barev (Color Balance)

Druhým filtrem, který v Adobe Photoshop upravuje odstín barev, je Vyvážení barev. Zde můžete odstín obrazu upravit nastavením hodnot, které se přičítají, respektive odečítají od jednotlivých kanálů RGB. Alespoň tak se to může jevit běžnému uživateli. V podstatě existují dvě možnosti. První z nich je ta, že se nějaké hodnoty opravdu přičítají přímo ke kanálům RGB, ale zároveň by muselo docházet k vyvažování kanálů, aby nedocházelo ke změně světlosti obrazu. Druhá možnost znamená, že je obraz interně převeden do nějakého jiného barevného prostoru, který uchovává zvlášť informaci o světlosti pixelu a zvlášť barevnou informaci (například Lab, Luv, YC_oC_g atd.). Který přístup to je, to s určitostí říci nemůžeme. Velkou výhodou tohoto filtru je, že lze měnit hodnoty těchto offsetů pro světla, stíny a střední tóny zvlášť. Tato možnost se však poněkud odlišuje od našich představ tím, že odstín měníme několika táhly, nikoliv jako jednu transparentní veličinu. Z tohoto hlediska se Vyvážení barev blíží spíše 3-Way Filteru a bude zmíněno níže.

Další možností, jak měnit barevný odstín, by mohl být převod obrazu do barevného prostoru Lab (či CIE XYZ). Zde je samostatně oddělen kanál světlosti (L) a kanály vypovídající o poměru mezi červenou a zelenou (a) a mezi modrou a žlutou (b). Kanály a, b, by se pak daly upravovat pomocí křivek. Bohužel není znám jednoznačný přístup, který by umožnil měnit tímto způsobem odstín a zároveň zachovával nezměněnou sytost. Příčinou může být to, že RGB gamut vyjádřený v rovině a, b má sice šedé tóny v hodnotách okolo nuly a sytější hodnoty dál od nuly, ale jeho tvar není pravidelný. Faktem je, že tato metoda je pouze teoretická a v úpravě fotografií není známo její použití.

2.2.3 Sytost

Barevná sytost (Saturace) byla již představena výše. Na rozdíl od odstínu je snadno popsatelná i v barevném prostoru RGB, kde představuje rozdíl mezi hodnotou v kanálu s nejvyšší a nejnižší intenzitou (barvu 255-255-0 bude sytost 100%, pro barvu 120-120-120 bude sytost 0% a jedná se o achromatickou barvu). Tato skutečnost poskytuje větší volnost v užití různých přístupů, které se od sebe liší svými výsledky, ale i složitostí. Jednotlivé přístupy jsou seřazeny od nejjednodušších k těm nejsofistikovanějším.

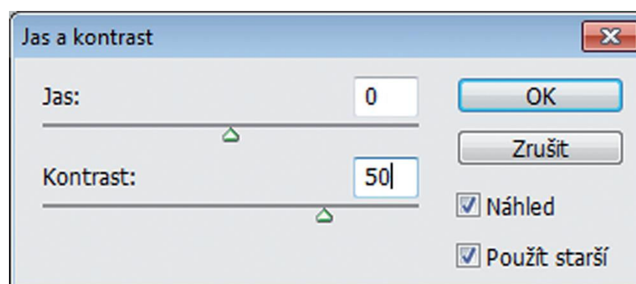
Jak u těchto metod rozhodnout, zda jsou vhodné? Určitě můžeme posoudit složitost a vizuální výsledek, ten ale může být osobním názorem zkreslen. Pokud chceme měnit pouze sytost, neměl by se měnit odstín ani světelnost. Rozhodl jsem se prozkoumat různé metody tak, že obraz před úpravou i po ní převedu do barevného prostou Lab a změny jednotlivých vzorkových barev vynesu do grafů, roviny a, b. Navíc jsou tyto vektory obarveny podle změny světlosti.

Z výše nastíněného popisu barevného prostoru Lab vyplývá, co lze z takového grafu vyčíst. V počátku grafu [0,0] se nacházejí achromatické barvy. Čím dále od středu se barvy nacházejí, tím jsou sytější. Takto můžeme posuzovat změnu sytosti celkově, ale i podle toho, jak syté byly barvy na počátku. Dále bychom chtěli, aby vektory barevných změn směřovaly přibližně ze středu. V Lab barevném prostoru však barva konstantního odstínu neleží přímo na polopřímkách směřujících ze středu, ale na mírně prohnutých křivkách. Výrazné změny směrů (obr. 2.20) však dovolují označit výsledek jako špatný.

Obr. 2.5: Originální obrazy.

Jas a kontrast (Brightness and Contrast)

Jedná se o nejjednodušší filtr k úpravě kontrastu a vůbec nejhorší filtr k jakékoli úpravě barev obrazu. Tento filtr interně pracuje ve starších verzích Adobe Photoshop jako lineárně interpolované křivky, tedy pouze posunuje černý a bílý bod obrazu. Ve většině případů je tedy zcela destruktivní vůči detailům ve světlech a stínech. V novějších verzích je realizován spojitými křivkami, a je tedy méně destruktivní. Ani tato verze však není uspokojivá, neboť zároveň se sytostí upravují i světelnost obrazu, a to je ve většině případů nevhodné.



Obr. 2.6: Nástroje Jas a kontrast.

Obr. 2.7: Výsledek zvýšení sytosti pomocí jasu a kontrastu (+50, starší verze).

Křivky (Curves)

Pokud zvýšíme kontrast pomocí křivek v modelu RGB, dostaneme výsledek podobný novější variantě Jasu a kontrastu. Malou výhodou je flexibilnější nastavení křivky a tím pádem i možnost vyhnout se ztrátě detailů. Dále se musíme vyhnout změně jasu. Toho lze docílit následujícím způsobem: původní obraz necháme ve spodní vrstvě a tuto vrstvu sduplikujeme. Teprve na nově vzniklou vrstvu použijeme úpravu kontrastu pomocí křivek a nakonec rozhodneme o volbě prolnutí svrchní vrstvy, možnost "barva". Tento režim zde funguje jako korekce jasu.

Další možností je použít místo barevného prostoru RGB prostor CMYK. Zde můžeme měnit křivky ve všech kanálech (jako v RGB), nebo pouze v kanálech CMY. V tomto případě lze získat mnohem lépe vypadající výsledek (při zvýšení sytosti nedochází tolik ke ztmavení stínů). Nevýhodou tohoto přístupu je, že modelů CMYK je celá řada a další můžeme sami definovat. Výhoda neztmavení stínů tedy funguje při určitém výtažkování černé barvy a nelze ji zcela zobecnit. Z tohoto důvodu nebudu tento přístup implementovat.

Poslední možností je upravovat obraz v barevném prostoru Lab (či jiném prostoru z této rodiny, Adobe Photoshop upřednostňuje právě Lab) a zde aplikovat lineárně interpolované křivky na kanály a, b. Je to velmi oblíbená úprava, neboť je poměrně snadná a má

celkem hezky vypadající výsledky. To je důsledek faktu, že upravujeme pouze kontrast barev, nikoliv jas.

Obr. 2.8: Výsledek zvýšení sytosti pomocí křivek, zvyšujících kontrast na kanálech RGB, a použití prolnutí "barva".

Obr. 2.9: Výsledek zvýšení sytosti pomocí křivek, zvyšujících kontrast na kanálech CMY.

Obr. 2.10: Výsledek zvýšení sytosti pomocí křivek, zvyšujících kontrast na kanálech a, b.

Obr. 2.11: Výsledek zvýšení sytosti pomocí překrytí vrstev Lab, volbou prolnutí "měkké světlo" (softlight) a krytím 50%.

Křivky lze napodobit také jinou metodou v Adobe Photoshop. Vytvoříme vrstvu z pozadí obrazu (celý obraz), tuto vrstvu budeme duplikovat a jako metodu prolnutí svrchní vrstvy

zvolíme některou z těchto variant překrytí: měkké světlo, tvrdé světlo, jasné světlo atd. Nakonec snížíme krytí této vrstvy. Tato úprava se chová, jako kdybychom použili křivky na kanály a, b, nikoliv však pouze lineárně interpolované křivky, ale jemnější spojité křivky. Volbou různých variant míchání a míry krytí tak v podstatě měníme prohnutí křivky. Tyto různé varianty a použití křivek na kanály a, b v prostoru Lab jsou vyneseny do grafů, které naleznete v kapitole 2.4 (Analýza výsledků nástrojů sytosti).

Odstín a sytost (Hue and Saturation)

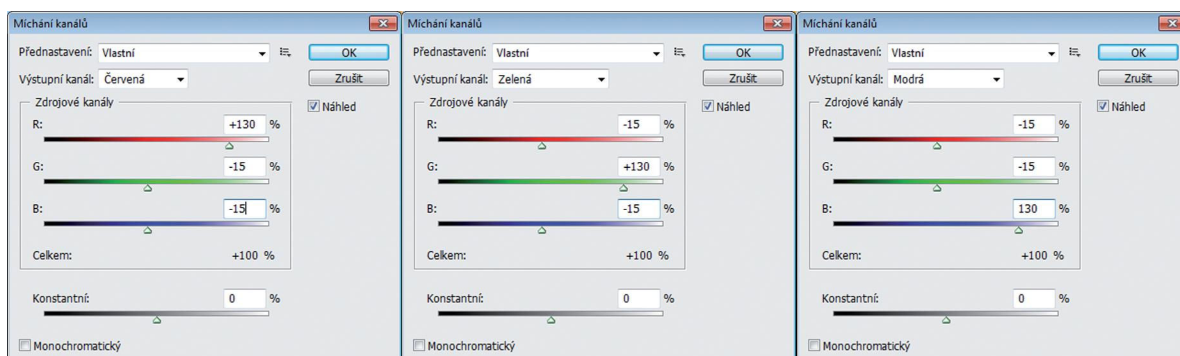
Úprava sytosti poskytnutá tímto již jednou zmíněným filtrem se zdá být mnohem jemnější než v předchozích případech. Sytost je nastavována táhlem S, které může dosahovat hodnot -100 až 100. K čemu tato hodnota slouží? Opět předpokládám, že tento nástroj pracuje nad obrazem v barevném prostoru HSL. Pak by nejspíš tato hodnota (patřičně normalizovaná) sloužila k přenásobení původní hodnoty kanálu S.

Obr. 2.12: Výsledek zvýšení sytosti pomocí nástorje Odstín a sytost (hodnota +30).

Míchání kanálů (Channel Mixer)

Dalším využitelným filtrem pro úpravu sytosti je Míchání kanálů pro barevný prostor RGB. Výsledky jsou opět velmi kvalitní, trochu odlišné od předchozích filtrů. Uživatelský

postup zde sice není tak jednoduchý jako ve filtru Odstín a sytost, naproti tomu by ale velkou výhodou mohla být výpočetní složitost (v tomto případě spíše jednoduchost). Postup předvedu na příkladu: Pro každý jednotlivý kanál zvyšte hodnotu odpovídajícího kanálu ze 100% na 150%, zbylé dva kanály nastavte na -25% (pochopitelně lze zvyšovat hodnotu až na 200%, ale je důležité, aby součet kanálů byl 100%, takže zbylé dva kanály pak budou -50%). Výhodou zde je to, že program nemusí dělat nic navíc, než zde dělá uživatel.



Obr. 2.13: Zvýšení sytosti (+30%) pomocí Míchání kanálů.

Obr. 2.14: Výsledek zvýšení sytosti (+30%) pomocí Míchání kanálů.

Živost (Vibrance)

Nejnovější nástroj (přidaný v Adobe Photoshop CS4) obraz upravuje nejšetrněji a zároveň má úplně odlišné výsledky, než jaké mají ostatní filtry, které méně syté barvy dosycují méně a více syté ještě více. Tak častokrát dochází k ”barevným přepalům”. Živost, jak se zdá, pracuje šetrněji. Více dosycuje málo syté barvy a ty více syté dosycuje méně. Jednoduché a prosté, ale jak to funguje uvnitř? K pochopení tohoto filtru bylo nezbytně nutné jej důkladně otestovat a výsledky vynést do a, b grafů (kapitola 2.4).

Na internetových diskusích jsem zjistil, že nejsem sám, kdo by chtěl tomuto filtru přijít na kloub. Vyzkoušel jsem i některé z těchto návodů (viz [4]). V principu stojí za všemi snaha vypočítat nový kanál, který vystihuje vzdálenost barevného pixelu od odpovídajícího šedého tónu. Podle metody z výše uvedeného zdroje tento kanál získám tak, že obraz nejprve zkopíruji do nové vrstvy, kterou pak převedu na odstíny šedi (podle návodu pomocí Gradient map). Tuto vrstvu prolnu se spodní vrstvou volbou ”rozdíl” (difference), výsledek zkopíruji a vytvořím z něj nový alfa kanál. Jaký je význam tohoto kanálu? Bílá barva představuje plně saturovaná místa a černá barva achromatická místa. Na tento kanál použiji invert a následně příkaz vyrovnat (rovnoměrně změní rozložení hodnot jasu v kanále tak, aby reprezentovaly celý rozsah, čímž se zvýší rozdíl úpravy mezi málo a hodně dosycenými místy). Nakonec se na základě tohoto kanálu jakožto masky provádí klasická úprava sytosti nástrojem odstín a sytost.

Obr. 2.15: Výsledek zvýšení sytosti (+100) pomocí Živosti.

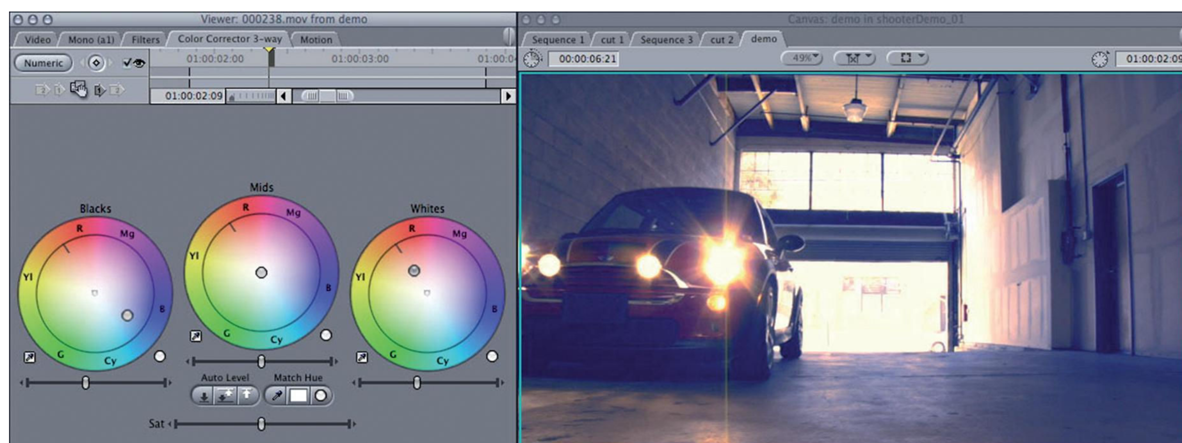
Obr. 2.16: Výsledek zvýšení sytosti pomocí návodu, napodobujícího živost (odstín a sytost - sytost +75).

2.3 3-Way Filter

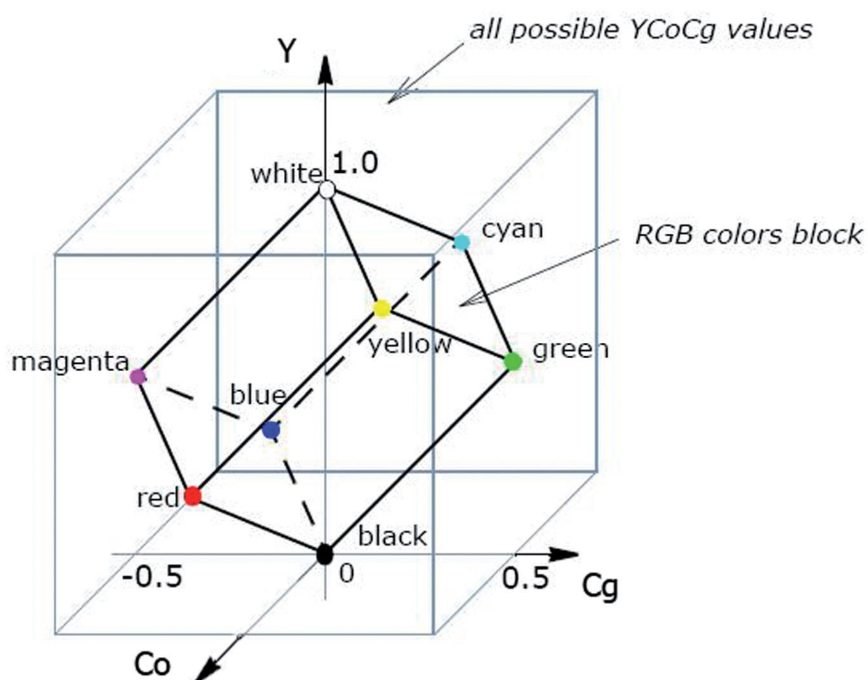
Tento filtr je ve světě úprav digitálních fotografií poměrně neznámý, je však velmi často využíván v barevné korekci videa. Jedná se o jednoduchý nástroj, kde nastavíte zvlášť pro světla, středy a stíny barvy, kterými budete obraz tónovat. V podstatě se jedná o filtr velmi podobný filtru vyvážení barev v Adobe Photoshop. Jestliže je tento filtr oblíbený pro úpravy videa, očekává se, že bude rychlý. Jak už jsem napsal při první zmínce o filtru vyvážení barev, jsou dvě stěžejní možnosti, jak tónovat obraz a zároveň co nejméně měnit jas obrazu. Podle první varianty je možné provádět úpravy přímo v RGB, ale následně musí dojít k jasové korekci. Druhá varianta vyžaduje nejprve převod obrazu do některého barevného prostoru, který má zvlášť vyjádřen jasový kanál a zvlášť barevné kanály (Lab, CIE L*u*v, CIE XYZ, YC_bC_r, ...). V tomto prostoru následně dojde k úpravě kanálů nesoucích barevnou informaci (přičemž zda se jedná o světla, středy, či stíny, je rozhodováno podle jasového kanálu). Nakonec se obraz převede zpět do RGB.

Jelikož potřebujeme již během výpočtu jasový kanál, vychází vhodnější druhá varianta. Při výběru pracovního prostoru bych se omezil na barevné prostory používané ve videu, jejich výpočet je obecně snazší (YC_bC_r). Navíc pokud není třeba držet se barevných standardů, je možné převést obraz do nejvíce zjednodušeného ekvivalentu YC_bC_r, do prostoru

YC_oC_g. Zde Y představuje pseudo jas, kanál C_o představuje oranžovou barevnost (tedy poměr mezi červenou a azurovou) a kanál C_g zelenou barevnost (poměr mezi purpurovou a zelenou, obr. 2.17).



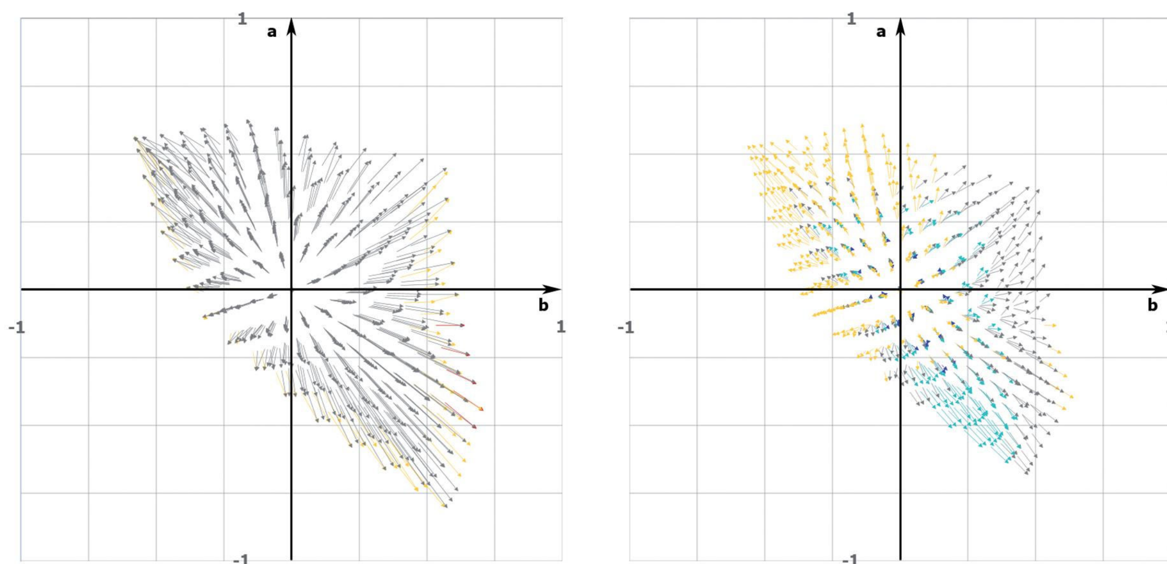
Obr. 2.17: 3-Way Filter v programu Final Cut Pro.



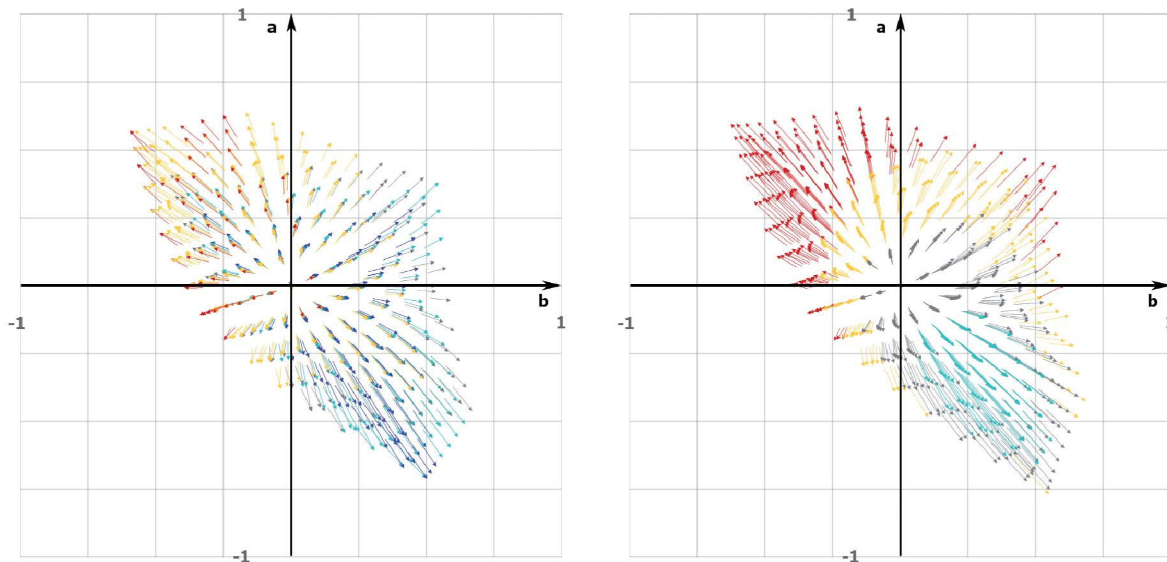
Obr. 2.18: Barevný prostor YC_oC_g (s vyznačením RGB prostoru).

2.4 Analýza výsledků nástrojů sytosti

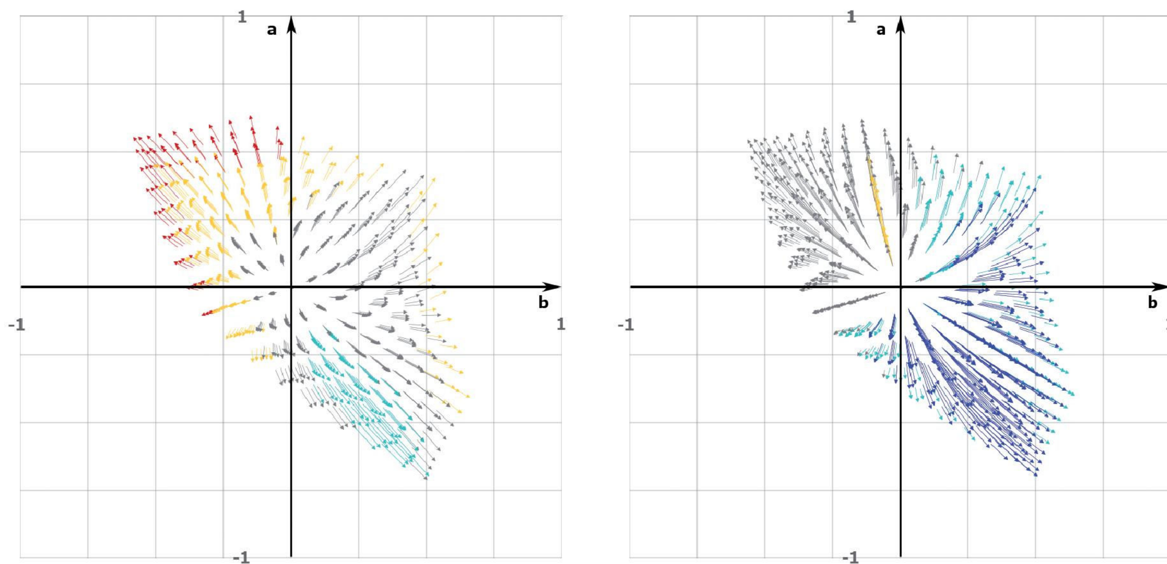
Přístupů k úpravě sytosti je velmi mnoho a není nezbytně nutné všechny implementovat. Nejprve jsem si vzal obraz, který potřeboval úpravu sytosti. O tu jsem se pokusil nástrojem odstín a sytost a ostatními filtry jsem se následně snažil dosáhnout co nejpodobnějšího výsledku. K rozhodnutí, které z těchto metod využít, mi dopomohly následující grafy. Pro jejich vytvoření jsem vygeneroval paletu základních (729) barev. Nad touto paletou jsem provedl úpravy sytosti ve stejné síle, jako v předchozím případě. Následně jsem si připravil vlastní aplikaci v jazyce C#, která porovnává obraz (převedený do barevného prostoru Lab) před úpravou a po ní a následně zjištěné změny vynese do a,b grafu pomocí xml generovaných SVG křivek (program je přiložen na DVD). Vybrané grafy jsou zde uvedeny. Kromě kanálů a,b graf vypovídá o změně L kanálu. To je vyznačeno pomocí čtyř různých barev vektorů. Pokud došlo ke změně L kanálu o -1 až +1, jsou vektory vyznačeny šedivou barvou. U snížení svělosti o 1 až 3 jsou vektory vyznačeny žlutě, o 3 a více červeně. U zvýšení světlosti o 1 až jsou vektory vyznačeny světle modře, o 3 a více tmavě modře.



Obr. 2.19, 2.20: Vlevo graf zvýšení kontrastu kanálů a,b v barevném prostoru Lab pomocí křivek posunutím počátečního a koncového bodu směrem ke středu o 30. Vpravo graf překrytí vrstev Lab s prolnutím overlay a krytím svrchní vrstvy 30%.



Obr. 2.21, 2.22: Vlevo graf překrytí vrstev Lab s prolnutím softlight a krytím svrchní vrstvy 50%. Vpravo graf míchání kanálů +20%.



Obr. 2.23, 2.24: Vlevo nástroje odstín a sytost pro zvýšení sytosti +20%. Vpravo graf živosti +100%.

O zvýšení sytosti pomocí zvýšení kontrastu na kanálech a, b (obr. 2.19) lze jednoznačně říci, že je zcela šetrné k jasovému kanálu. Jemné změny můžeme nalézt u nejkrajnějších bodů. To však není s nejvyšší pravděpodobností způsobeno naší úpravou, nýbrž následným ořezem tohoto obrazu do RGB gamutu. Tato skutečnost se může vyskytnout u všech grafů, a proto nebudeme chybám na okraji RGB gamutu přikládat velký význam.

Na dalších dvou grafech (obr. 2.20, 2.21) vidíme úpravu sytosti pomocí překrytí dvou Lab vrstev. Tyto dvě varianty se od sebe liší ve zvolené metodě prolnutí a v krytí. Výše jsem zmínil, že by tyto přístupy mohly nahradit zvyšování kontrastu na kanálech a, b, ale z našich grafů zjišťujeme, že to byl omyl. Ať už volbou prolnutí, či hodnotou krytí dochází ke změnám v jasové složce. Navíc u volby prolnutí "overlay" dochází v červených a modrých barvách k velmi chaotickým změnám.

Na dalším grafu (obr. 2.22) nalezneme zvyšování sytosti pomocí míchání kanálů. Zde si můžeme všimnout, že u zelených barev dochází ke ztmavování. Je to dáno nepříliš sofistikovaným přístupem ke korekci jasu během změn. Na druhou stranu ale můžeme vyzdvihnout, že úpravy sytosti v podstatě nikde nehýbou s odstínem.

Následující graf (obr. 2.23) zobrazuje výsledky úpravy sytosti pomocí nástroje odstín a sytost. Tentokrát se graf celkem podobá předešlému, to jest míchání kanálů. Ztmavení zelených barev zde však není tolik výrazné a také můžeme pozorovat lehké uhnutí červených barev směrem ke žluté.

Poslední graf (obr. 2.24) zobrazuje výsledky živosti. Od předchozích se značně odlišuje a můžeme konstatovat hned několik skutečností. V červených a modrých barvách dochází k výraznému zesvětlení barev. Červené barvy lehce uhýbají ke žluté a modré barvy nepatrně k červené. Nejdůležitějším zjištěním je to, že barvy, které jsou málo syté, jsou dosycovány mnohem více než u všech předchozích metod.

Grafů, které při porovnání výsledků vznikaly, bylo mnohem více. Z pochopitelných důvodů jsem zde uvedl ty nejzajímavější (tedy nejžhavější kandidáty na implementaci). Navíc jsem také vyzkoušel, zda se grafy těchto úprav nebudou lišit, pokud vyneseme pouze tmavé barvy (respektive pouze světla, pouze středy). Výsledkem byly velmi obdobné grafy, jako zde uvádím, proto jsem se rozhodl zveřejnit grafy s celým rozsahem. První tři grafy hovořily o úpravách nad barevným prostorem Lab. Jejich výsledky byly nejvíce rozdílné, a to podle toho, o jaký přístup se jednalo. Z uživatelského hlediska si myslím, že úprava sytosti v barevném prostoru Lab je poměrně perspektivní, ale z velké divergence výsledků vyvozuji, že je vhodná pouze pro profesionální uživatele. Pro ně však nemá cenu dělat automatizovaný nástroj, nýbrž jim pouze umožnit svobodné úpravy pomocí křivek atd. Z tohoto důvodu nejsou tyto tři přístupy implementovány. Ostatní jsem se rozhodl implementovat - míchání kanálů (jednoduchost), odstín a sytost (kvůli možnému následnému rozšíření o světlost) a nakonec živost (kvůli originálním výsledkům).

3 Základní implementace

Nejprve bylo nutné se rozhodnout, v jakém jazyce filtry napsat. Kvůli vysokým požadavkům na rychlost přicházely v úvahu jedině jazyky C a C++. Svou objektovostí je pro filtr křivky pohodlnější C++, ale nakonec jsem se rozhodl zůstat u ANSI C. Jako vývojové prostředí jsem zvolil Microsoft Visual Studio.

3.1 Křivky

Všechny funkce (na přidávání, ubíraní a měnění řídících bodů, inicializaci pole řídících bodů křivky, datového segmentu křivky, výpočtu dat křivky a aplikace křivka na obrazová data) jsem implementoval do souboru Curves.c. Jak jsem uvedl výše, při tvorbě filtru křivky jsem vycházel z tvorby mého předchůdce Jana Kučery. Ten použil k psaní filtru křivky jazyka C++ a objektového přístupu (spojový seznam) k přidávání, ubírání a ke změnám řídících bodů. Musel jsem tedy tyto operace implementovat jinak. Řídící body jsem umístil do pole nově definované struktury. Ta obsahuje číselnou informaci identifikující bod, x-ovou a y-ovou hodnotu. Při vytvoření nové křivky je ihned přidán počáteční a koncový bod. Při dalším přidávání, ubírání a měnění bodů dojde k jejich seřazení podle x-ové souřadnice od nejmenšího k největšímu, neboť tak je to vyžadováno pro spline interpolaci (jelikož zůstává pole po každé operaci seřazené, stačí pouze umístit na správné místo nově přidaný, respektive změněný bod).

Uchovávání řídících bodů však není zdaleka tak zajímavé jako implementace křivek samotných. Ty fungují tak, že si nejprve pro všechny možné vstupní hodnoty vypočtu hodnoty výstupní. Následně tato data aplikuji na zvolený kanál (či kanály). Při výpočtu dat křivky již přímo vycházím z funkce Jana Kučery CalcCurrentCurve(). Mnou implementovaná funkce CalcCurveData() je tedy pouze lehce pozměněná výše zmíněná funkce. Jak výpočet dat funguje? Procházím celý rozsah hodnot zvolené bitové hloubky, dokud jsou procházené x souřadnice menší než souřadnice počátečního řídícího bodu, a přiřazuji jim y souřadnice počátečního řídícího bodu. Jakmile jsou x souřadnice větší než souřadnice koncového řídícího bodu, přiřazuji zbylým bodům y souřadnice koncového řídícího bodu. Co se děje, když je x souřadnice mezi těmito řídícími body? Jsou dvě možnosti. Pokud jsou nadefinovány pouze dva řídící body křivky (počáteční a koncový), veškeré y hodnoty mezi nimi budou lineárně interpolovány. Jestliže jsme nadefinovali více než tyto dva body, bude se provádět spline interpolace. Od funkce pana Kučery se moje funkce liší ve zvolené bitové hloubce. CalcCurrentCurve() je implementována na hloubku 8 bitů (tedy rozsah 0 - 255), CalcCurveData() je implementována na hloubku 16 bitů (rozsah 0 - 65535). Důvodem je nespokojenost s hrubostí tohoto filtru v Adobe Photoshop, která je nešťastná například při použití křivek na kanály a, b prostoru Lab pro vyvážení barev.

Curve AddPoint(curve , X, Y) funkce na přidání bodu (X, Y) do vstupní křivky, výstupem křivka.

Curve DeletePoint(curve, number) funkce na smazání bodu podle čísla ze vstupní křivky, výstupem opět křivka.

Curve ChangePoint(curve, number, X, Y) funkce na změnu hodnot bodu. Bod vybrán podle čísla ze vstupní křivky, nové hodnoty X a Y a výstupem opět křivka.

Curve CreateCurve() funkce na založení nové křivky (zároveň přidá základní dva řídící body).

CurveData CreateCurveDataArray() funkce na alokaci datového segmentu křivky.

CurveData CalcCurveData(Curve curve) funkce na výpočet datové tabulky křivky (výpočet výstupních hodnot křivky pro veškeré možné vstupy).

RGB_8_bit *ApplyCurveOnRGB_8_bit(*inRGB, curve, pixelNumber) funkce na úpravu hodnot kanálů R,G a B v daném obrazu podle dané křivky.

RGB_8_bit *curveExample_8_bit(*inRGB, pixelNumber) příklad použití výše zmíněných funkcí.

Jak vůbec spline interpolace funguje?

Spline-funkce 1. řádu je po částech lineární funkce, jejímž grafem je lomená čára spojující zadané body. V praxi se uplatňuje zejména interpolace kubickou funkcí, tj. funkcí, která je po částech polynomem 3. řádu. Pro konstrukci spline funkce potřebujeme seřazené body a dále to, aby interpolační spline polynom byl spojitý s první i druhou derivací. Nyní uvedu způsob výpočtu spline funkce. V tomto případě opět vynechám matematické odvození, které si případný zájemce může vyhledat v literatuře týkající se numerických metod, například v [5].

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

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

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

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

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

určíme:

$$M_0 = f_0^n = 0,\ M_n = f_n^n = 0$$

zbylá M_i dopočteme jako řešení soustavy:

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

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

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

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

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

3.2 Odstín a sytost

Podle závěru předchozí kapitoly jsem z výše uvedených dostupných metod nakonec vybral úpravu pomocí nástroje odstín a sytost (kombinuje úpravu odstínu a sytosti zároveň, navíc je poměrně snadno rozšiřitelný, aby zvládal úpravu jasu), úpravu sytosti pomocí nástrojů míchání kanálů (pro svoji jednoduchost) a nástroje živost (pro své jedinečné výsledky u málo saturovaných barev). Všechny tyto přístupy jsou implementovány v souboru Saturate.c.

3.2.1 Míchání kanálů

Odstín realizovaný pomocí míchání kanálů jsem implementoval do funkce SaturationByChannelMixing_RGB_8_bit(). Vstupem je pole RGB struktury (v 8 bitové hloubce), počet upravovaných pixelů a hodnota, o kterou se má měnit sytost. Tato hodnota má povolený rozsah od -1 do 1 při libovolné přesnosti. Výstupem je opět pole RGB struktury.

Samotný výpočet je velmi jednoduchý, v podstatě je to identický přepis postupu, jak byl popsán výše:

$$\begin{aligned} R' &= \alpha R - \beta G - \beta B \,, \\ G' &= -\beta R + \alpha G - \beta B \,, \\ B' &= -\beta R - \beta G + \alpha B \,, \end{aligned}$$

kde $\alpha = 1 + vstupni_hodnota$; $\beta = vstupni_hodnota/2$.

Pro každou z přepočtených hodnot je nutné ověřit, zda nedošlo k přetečení maxima rozsahu neznaménkového celočíselného 8bit.

3.2.2 Odstín a sytost

Tento nástroj, kombinující úpravy sytosti a odstínu, je naimplementován do další funkce, HueSaturationOnHSL_RGB_8_bit(). Vstupem této funkce je opět pole RGB struktury (8 bitů), počet upravovaných pixelů, hodnota úpravy sytosti a navíc hodnota úpravy odstínu. Výstupem je pole RGB struktury. Minimální hodnota úpravy odstínů je 0, maximální není omezena, v praxi je použitelná do 2. Hodnota úpravy sytosti je od 0 do 360.

Do filtru vstupují obrazová data v RGB. Nejprve je nutné obrazová data převést do barevného prostoru HSL. K tomu slouží funkce rgbToHSL_8_bit() (je importována z ColorSpaceConvert.h, vstupem je pole RGB struktur a počet upravovaných pixelů, výstupem je pole HSL struktur).

Obraz v HSL je pak upravován přičtením zadané konstanty k hodnotám kanálu H a vynásobením hodnot kanálu S další zadanou konstantou. Výsledek těchto úprav musí projít opět korekcí, aby nepřetekl povolený rozsah. Následně je pak obraz převeden zpět do barevného prostoru RGB pomocí funkce hslToRGB_8_bit().

Implementace převodní funkce rgbToHSL_8_bit() vychází z definice barevného prostoru HSL[6]. Pro kanál L (neboli světelnost) platí:

$$L_{HSL} = \frac{1}{2}(M + m) ,$$

kde M je hodnota maximálního a m je hodnota minimálního RGB kanálu. Pro kanál S je potřeba znát nejprve takzvanou chromu, pro kterou platí:

$$C = M - m ,$$

pak tedy kanál S bude:

$$S_{HSL} = \begin{cases} 0, & \text{je-li } C = 0 \\ \frac{C}{1-|2L-1|}, & \text{jindy.} \end{cases}$$

Nakonec pro kanál H bude platit:

$$H' = \begin{cases} \text{nedefinováno}, & \text{je-li } C = 0 \\ \frac{G-B}{C} \bmod 6, & \text{je-li } M = R \\ \frac{B-R}{C} + 2, & \text{je-li } M = G \\ \frac{R-G}{C} + 4, & \text{je-li } M = B, \end{cases}$$

$$H_{HSL} = 60H'$$

Pro implementaci funkce hslToRGB_8_bit() stačí invertovat výše zmíněné instrukce. Pokud někoho zajímají tyto zpětné převody, nalezne je například v [6].

3.2.3 Živost

Tento nástroj jsem naimplementoval ve funkci Vibrance_RGB_8_bit(). Z výše rozebraných výsledků tohoto filtru v Adobe Photoshopu je patrné, že jeho účelem je zvyšování (respektive snižování) sytosti málo dosycených barev. Implementace v podání Adobe Systems je bohužel neznámá, mnohé si však můžeme domyslet, a tak jsem se rozhodl vytvořit filtr ryze po svém. Z předchozí implementace jsme zjistili, že převod do prostoru HSL umožní jasné, snadné a transparentní úpravy sytosti (na kanálu S).

Jakmile tedy převedeme obraz z RGB do HSL, zbývá vyřešit otázku, jakou funkci použít k úpravě kanálu S. Z výše uvedených požadavků vyplývá, že potřebujeme funkci, která nízké hodnoty zvyšuje více a pro kterou dále platí, že čím jsou hodnoty vyšší, tím méně je zvyšuje, až nakonec nejsou zvyšovány vůbec. Pokud se podíváme mezi nejběžnější

funkce, našim požadavkům ideálně vyhovuje mocninná funkce s exponentem o hodnotách (0, 1) a s oborem hodnot také na intervalu (0, 1).

$$S'_{HSL} = S^{hodnota}$$

Nastává poněkud neobvyklá situace, neboť snižováním hodnoty konstanty dochází k dosycování konstanty a naopak. Není to žádná překážka, jen je třeba s tím počítat, až budeme tento filtr spojovat s GUI a hodnoty táhla přizpůsobovat rozsahu. Po řadě testů jsem vymezil rozmezí hodnot konstanty, pro které se funkce chová tak, jak potřebujeme. Může tedy nabývat hodnot 0,6 pro maximální dosycení až 1,4 pro maximální odsycení.

3.3 3-Way Filter

Tento nástroj jsem umístil do samostatného zdrojového souboru ThreeWayFilter.c a je realizován funkcí ThreeWayFilter_RGB_8_bit(). Vstupem této funkce je pole RGB struktur, počet pixelů a hodnoty úprav. Hodnot je celkem šest, neboť pro světla, středy a stíny máme vždy po dvou proměnných. Implementace tohoto filtru byla stejně jako v předchozím případě v podstatě "naslepo". Z výše napsaného rozboru filtru jsem se rozhodl pro úpravu nad určitým barevným prostorem, který by měl jasovou složku oddělenou. Jako tento prostor jsem nakonec zvolil prostor YC_oC_g, který je zjednodušenou variantou prostoru YC_bC_r. Jeho hlavní výhodou je bezkonkurenčně nejsnazší převod z RGB a podle některých zdrojů[7] je uváděn pouze jako způsob kódování prostoru RGB. Zde kanál C_o představuje oranžovou barevnost (tedy poměr mezi červenou a azurovou) a kanál C_g zelenou (tedy poměr mezi purpurovou a zelenou). Pokud chceme obraz tónovat, stačí přičítat, respektive odečítat zadané konstantní hodnoty k těmto dvěma kanálům. My však nezůstaneme u pouhé konstanty ke všem bodům obrazu. Naším cílem je nastavovat tónování pro světla, středy a stíny zvlášť. Pixel tedy můžeme posuzovat podle jeho jasové složky Y. Nestačí však rozdělit tento rozsah na tři intervaly, ale na pět. Je totiž bezpodmínečně nutné mezi světla a středy a mezi středy a stíny vložit intervaly, na nichž bude přechod od jeden̆ch konstant k druhým pozvolný. V těchto dvou intervalech vystačíme s lineární interpolací tak, jak je naznačeno v grafu níže.

Princip úpravy kanálů je tedy velmi jednoduchý. Samotný převod do prostoru YC_oC_g a zpět do RGB je realizován funkcemi rgbToYCoCg_8_bit() a ycocgToRGB_8_bit(). Rovnice převodu z RGB[8]:

$$\begin{aligned} Y &= (R + 2G + B) >> 2\,, \\ C_o &= (R - B) >> 1\,, \\ C_g &= (-R + 2G - B) >> 2\,. \end{aligned}$$

Rovnice převodu do RGB:

$$
\begin{aligned}
R &= Y + C_o - C_g \,, \\
G &= Y + C_g \,, \\
B &= Y - C_o - C_g \,.
\end{aligned}
$$

V tomto převodu je však třeba hlídat, aby nedocházelo k přetečení rozsahu.

4 Rozšířená implementace

Naším cílem je, aby tyto filtry fungovaly jako nástroj v bitmapovém grafickém editoru pro tvorbu a úpravu bitmapové grafiky. V dnešní době veškeré obdobné editory umožňují barevnou korekci spolu s realtimovým náhledem na aktuální výsledek korekce. Zároveň jsme tlačeni ze strany výrobců monitorů jejich stále vyšším a vyšším rozlišením. Naším cílem je dostat tyto úpravy na co nejkratší čas u co největších obrazů s co nejmenší možnou chybou. Už předem asi vidíte, že není možné dosáhnout absolutního úspěchu a že nejspíše budeme muset dělat určité kompromisy. Nejprve tedy předvedu urychlení na hloubce 8 bitů na kanál, poté budu diskutovat o implementací v 10, 12, 14, 16 bitech a nakonec se budu zamýšlet nad hloubkou 32 bitů a možností užití plovoucí desetinné čárky.

Jak budeme jednotlivé filtry urychlovat? Můžeme kombinovat následující postupy: zjednodušení rovnic, použití celočíselných výpočtů tam, kde to jde (a nenapáchá to velké škody), použití celočíselných porovnání (místo porovnání s plovoucí desetinnou čárkou) a nakonec nejzásadnější urychlení, použití vyhledávacích tabulek (zkratka používaná v angličtině LUT neboli look up table). V čem tkví výhoda užití vyhledávacích tabulek? Určitý výpočet, nebo pouze část výpočtu (například dělení, které je velmi časově náročné) předpočteme dopředu pro veškeré možné vstupní hodnoty. V těle funkce je pak výpočet nahrazen načtením výsledné hodnoty z položky tabulky odpovídající vstupu.

Mějme výpočet $y = \alpha * vstupni_hodnota$, pak nahrazení LUT bude vypadat následovně:

```
// vypocet hodnot LUT
for(i = 0; i <= rozsah_vstupni_hodnoty; i++){
  lut[i] = alpha * i;
}

// vypocet pomoci LUT
for(i = 0; i < pocet_iteraci; i++){
  y[i] = lut[vstupni_hodnota[i]];
}
```

Je tedy zcela patrné, že použití tabulky bude výhodné, dokud bude vstupních hodnot znatelně méně, než je počet iterací tohoto výpočtu ve funkci. Pokud uvažujeme, že vstupem bude jeden z kanálů (8 bitů) a filtrem budeme upravovat obraz o rozlišení 1920 x 1080 (2,1 milionu pixelů), budeme tedy výpočet provádět pro 256 vstupních hodnot místo 2,1 milionů výpočtů (je tedy velmi pravděpodobné, že i výpočet 65 tisíc vstupů pro hloubku 16 bitů bude mnohem rychlejší než počítat každý pixel jednotlivě). Pochopitelně musíme

brát v úvahu i velikost tabulky.

4.1 Křivky

Zde je situace poměrně jednoduchá. Křivky jako takové jsou navrženy tak, aby vypočítaly nejprve data křivky, což odpovídá celému rozsahu kanálu (16 bitů). Takto je vlastně předpočtena jakási vyhledávací tabulka. Na tomto místě bych rád vznesl myšlenku, podle které se řídím i ve všech následujících zjednodušeních. Pokud se rozhodnu některý výpočet nahradit LUT, předpokládám, že počet iterací není závratný, a daný výpočet při předpočítávání LUT budu provádět v plovoucí desetinné čárce (kromě těch případů, u kterých by ani při použití celočíselných výpočtů nedocházelo k chybě). Je to poměrně jasně daný postup, který má zabránit nárůstu zbytečných chyb. Z tohoto důvodu můžeme konstatovat, že křivky můžeme prozatím opustit.

4.2 Sytost (mícháním kanálů)

Ani zde není situace složitá. V kapitole implementace (3.2.1) jsem uvedl rovnice, kterými je filtr realizován. Každý kanál závisí na hodnotě konstanty, která slouží k úpravě sytosti, a na každém z ostatních tří kanálů. Pokud bychom chtěli použít k celému výpočtu kanálu zjednodušení pomocí jedné LUT, potřebovali bychom předpočítat tabulku, jejímž vstupem by byly všechny možné kombinace tří kanálů RGB.

```
// vypocet hodnot LUT
for(i = 0; i <= rozsah_hodnoty_X; i++){
  for(j = 0; j <= rozsah_hodnoty_Y; j++){
    for(k = 0; k <= rozsah_hodnoty_Z; k++){
      poradi_polozky = i * (rozsah_hodnoty_Y + 1) * (rozsah_hodnoty_Z + 1)
                       + j * (rozsah_hodnoty_Z + 1) + k;
      lut[poradi_polozky] = alpha * i + beta * j + beta * k;
}
```

Pokud bychom zjednodušili výpočet pořadí iterace (pro rozah 8 bitového čísla), vypadal by výpočet obrazových bodů takto:

```
// vypocet pomoci LUT
for(i = 0; i <= pocet_iteraci; i++){
```

```
    poradi_polozky = (x[i] << 16) + (y[i] << 8) + z[i]
    x_nove[i] = lut[poradi_polozky];
}
```

To by však znamenalo 16,8 milionů vstupů při hloubce 8 bitů. Taková tabulka pochopitelně žádné urychlení nepředstavuje, neboť vstupy dokonce převyšují výstupy, nehledě na paměťovou náročnost. Musíme tedy urychlit výpočet jiným způsobem, přesto však u vyhledávacích tabulek zůstaneme. Z rovnic je patrné, že se zde několikrát opakuje výpočet $\alpha * X$ a $\beta * X$, kde α a β jsou konstanty a X je obecně jeden z kanálů. Stačí tedy znát hodnotu filtru a můžeme si pro tyto dva výpočty předpočítat dvě tabulky s 256 vstupy, které nahradí celkem 9 násobení v každé iteraci.

```
// vypocet hodnot LUT alpha
for(i = 0; i <= rozsah_kanalu; i++){
    lut_a[i] = alpha * i;
}

// vypocet hodnot LUT beta
for(i = 0; i <= rozsah_kanalu; i++){
    lut_b[i] = beta * i;
}

// vypocet pomoci LUT
for(i = 0; i < pocet_iteraci; i++){
    R[i] =  lut_a[r[i]]-lut_b[g[i]]-lut_b[b[i]];
    G[i] = -lut_b[r[i]]+lut_a[g[i]]-lut_b[b[i]];
    B[i] = -lut_b[r[i]]-lut_b[g[i]]+lut_a[b[i]];
}
```

Kromě použití LUT zde také dohlédneme na to, aby všechny korekce rozsahu používaly celočíselné porovnávání.

4.3 Odstín a sytost (nad prostorem HSL)

Zde je již situace poněkud složitější. Úprava kanálů v prostoru HSL je v podstatě triviální, ale horší je to s převody mezi prostory x HSL a RGB. Pokud se vrátíme k cíli těchto urychlení, můžeme využít to, že chceme mít opravdu rychlý náhled tohoto nástroje, jinde tolik rychlost nepotřebujeme. Z toho vyplývá jasný fakt, že stačí převést obraz z RGB do HSL

pouze jednou. Zavoláním se nástroje se obraz nejprve převede do prostoru HSL a následně se bude volat funkce HueSaturationOnHSL_HSL_8_bit(), která bude mít na vstupu pole HSL struktur a výstupem bude pole RGB struktur. Z hlediska rychlého náhledu nás tedy zajímá doba běhu této funkce. Zajisté by bylo možné urychlit i převod z RGB do HSL, pokud nám však nebude vadit mírná prodleva, získáme tak HSL obrazové body nezatížené chybou (respektive pouze chybou zaokrouhlovací na konci výpočtu). To je z našeho pohledu velmi důležité, neboť nás při převodu zpět čeká ještě řada výpočtů, která by tuto chybu velmi zvýšila.

Pokud se podíváme do funkce HueSaturationOnHSL_HSL_8_bit(), zjistíme, že zde lze nahradit násobení kanálu S použitím LUT (vstupem bude hodnota kanálu S).

```
// vypocet hodnot LUT
for(i = 0; i <= rozsah_kanalu_S; i++){
  lut[i] = hodnota * i;
}

// vypocet pomoci LUT
for(i = 0; i < pocet_iteraci; i++){
  S_nove[i] = lut[S[i]];
}
```

Následně je nutné urychlit převod z HSL do RGB, který je totiž pro výstup na obrazovku nezbytný. Pseudokód původní funkce:

```
RGB_8_bit* hslToRGB(HSL *imageHSL, long pixelNumber){
    // pro vsechny body
    for (i = 0; i < pixelNumber; i++){
      //rgb pro achromaticke barvy
      if (imageHSL[i].saturation == 0.0){
          newRGB[i].red = (unsigned short)(imageHSL[i].light * 255.0);
          newRGB[i].green = (unsigned short)(imageHSL[i].light * 255.0);
          newRGB[i].blue = (unsigned short)(imageHSL[i].light * 255.0);
      } else {
          //rgb pro chromaticke barvy
          double max, min, sv, h, fract, vsf, mid1, mid2;
          int sextant;

          //normalizace odstinu na rozsah 0-6
          h = imageHSL[i].hue / 60.0;
```

```
//vypocet urovne nejvyssiho z kanalu, hodnoty 0-1
max = (imageHSL[i].light < 0.5) ? (imageHSL[i].light *
(1 + imageHSL[i].saturation)) : ((imageHSL[i].light +
imageHSL[i].saturation) - (imageHSL[i].light * imageHSL[i].saturation));
//vypocet urovne nejnizsiho z kanalu, hodnoty 0-1
min = (imageHSL[i].light + imageHSL[i].light) - max;
sv = (max - min) / max;
//sextant slouzi k rozhodnuti do ktere sestiny barva patri
sextant = (int)h;
//fract udava uhel barvy v dane sestine sestiuhelniku
fract = h - sextant;
vsf = max * sv * fract;
//mid1,2 jsou dve ruzne moznosti vysledku stredniho kanalu
mid1 = m + vsf;
mid2 = v - vsf;
//prideleni hodnot kanalu podle prislusne sestiny
switch (sextant)
{
case 0:
  newRGB[i].red = (unsigned short)(max * 255.0);
  newRGB[i].green = (unsigned short)(mid1 * 255.0);
  newRGB[i].blue = (unsigned short)(min * 255.0);
  break;
case 1:
  newRGB[i].red = (unsigned short)(mid2 * 255.0);
  newRGB[i].green = (unsigned short)(max * 255.0);
  newRGB[i].blue = (unsigned short)(min * 255.0);
  break;
case 2:
  newRGB[i].red = (unsigned short)(min * 255.0);
  newRGB[i].green = (unsigned short)(max * 255.0);
  newRGB[i].blue = (unsigned short)(mid1 * 255.0);
  break;
case 3:
  newRGB[i].red = (unsigned short)(min * 255.0);
  newRGB[i].green = (unsigned short)(mid2 * 255.0);
  newRGB[i].blue = (unsigned short)(max * 255.0);
  break;
case 4:
  newRGB[i].red = (unsigned short)(mid1 * 255.0);
```

```
            newRGB[i].green = (unsigned short)(min * 255.0);
            newRGB[i].blue = (unsigned short)(max * 255.0);
            break;
         case 5:
            newRGB[i].red = (unsigned short)(max * 255.0);
            newRGB[i].green = (unsigned short)(min * 255.0);
            newRGB[i].blue = (unsigned short)(mid2 * 255.0);
            break;
         }
      }
   }
   return newRGB;
}
```

Co zde lze urychlit? V prvé řadě lze veškeré výpočty provádět celočíselně místo plovoucí desetinné čárky. Pomůže nám tedy, když vstupem bude RGB v 8-bitovém unsigned integer. Následně tedy musíme ve všech výpočtech počítat s jiným rozsahem než doposud. Tedy například místo výrazu

```
v = (imageHSL[i].light < 0.5) ? (imageHSL[i].light *
(1 + imageHSL[i].saturation)) : ((imageHSL[i].light +
imageHSL[i].saturation) - (imageHSL[i].light * imageHSL[i].saturation));
```

musíme nahradit výrazem

```
v = (imageHSL[i].light < 128) ? (imageHSL[i].light +
((imageHSL[i].light * imageHSL[i].saturation >> 8))) : (imageHSL[i].light
+ imageHSL[i].saturation - ((imageHSL[i].light * imageHSL[i].saturation >> 8)));
```

Zde bitový posun o 8 vpravo odpovídá dělení 256. Tato úprava je potřebná kvůli vyššímu rozsahu (neboť pokud násobíme 1 * 1, dostáváme 1, to je korektní vůči mezím; pokud násobíme 256 * 256, dostáváme 65536, což je 256 krát větší než požadovaná mez).

Dále je potřeba vyjádřit kanál H tak, aby jej bylo možné zapsat do rozsahu unsigned 8-bit. Aby byla zachována myšlenka, že úhel Hue je dělitelný 6, rozhodl jsem se pro rozsah kanálu 0-240. Pokud chceme zjistit, do jaké šestiny daná barva spadá, budeme muset kanál Hue dělit 40 místo původních 60. Toto dělení jsem se rozhodl urychlit předpočtením této operace do LUT.

Další úpravou je vynechání zbytečných výpočtů, a to mid1 a mid2. Tyto hodnoty se pro každou iteraci mění a k samotnému dopočtení výsledku je vždy potřeba pouze jedna

z těchto hodnot. Proto nejsou vypočítávány ihned, ale až po volbě konkrétního switche.

Na závěr díky celému výpočtu v integeru není třeba konečný výsledek přenásobovat a výsledek je přirozeně v 8bitový unsigned integer.

4.4 Živost (nad prostorem HSL)

Stejně jako odstín a sytost vychází živost z úpravy nad prostorem HSL. Urychlení živosti tedy bude obdobné jako v předchozím případě. Nejprve je nutné od funkce na úpravu živosti oddělit převod z barevného prostoru RGB do prostoru HSL. Pak je potřeba použít LUT místo výpočtu mocninné funkce v přepočtu kanálu S.

```
// vypocet hodnot LUT
for(i = 0; i <= rozsah_kanalu_S; i++){
  lut[i] = pow(i, hodnota);
}

// vypocet pomoci LUT
for(i = 0; i < pocet_iteraci; i++){
  S_nove[i] = lut[S[i]];
}
```

Nakonec je třeba použít urychlený převod z HSL do RGB (viz kapitola 4.3).

4.5 3-Way Filter

Stejně jako v předchozích dvou případech dochází u 3-Way Filteru nejprve k převodu do jiného barevného prostoru (zde YC_oC_g). K samotnému náhledu tedy není třeba převod do tohoto prostoru, pouze převod zpět. Z funkce tedy opět odebereme převod do YC_oC_g a bude volán před začátkem tohoto filtru pouze jednou. Samotný přepočet kanálů C_o a C_g je realizován přičtením konstant k tomuto kanálu. V jasech, kde se mísí světla - středy a středy - stíny, musíme tuto konstantu získat lineární interpolací. Ta je ovšem poněkud časově náročnější, a tak je urychlena předpočtením do LUT (zvlášť pro C_o a C_g). Převod z YC_oC_g do RGB už je natolik zjednodušen, že více urychlit nejspíš nejde.

4.6 Další bitové hloubky

Chceme-li používat tyto filtry v praxi, nelze se spokojit pouze s úpravami v 8bitové hloubce. V Adobe Photoshop je podporována hloubka 8, 16 a částečně 32 bitů. Rozvíjející se technologie zobrazovacích zařízení nyní začíná podporovat hloubku 10 bitů na kanál a fotoaparáty, fotící do formátu RAW, ukládají data v hloubce 12 bitů na kanál. Lepší skenery umožňují skenování v bitové hloubce 14 bitů. Pokud tyto informace shrneme, dá se konstatovat, že je potřeba tyto filtry implementovat do hloubky 8, 10, 12, 14, 16 a možná 32 bitů.

Pokud budeme uvažovat pouze o hloubkách 10, 12, 14 a 16 bitů na kanál, lze říci, že implementace zde nebude příliš odlišná od 8bitové. Je třeba přizpůsobit se odpovídajícímu bitovému rozsahu a nahradit všechny 8bitové LUT. Jelikož jsou všude použity jednorozměrné vyhledávací tabulky, nebude problém použít tabulky odpovídající danému rozsahu. Pro největší 16bitovou tabulku je potřeba 65535 iterací výpočtů, což je výrazně méně než u obrazu s dvěma miliony obrazových bodů. U dvourozměrných LUT by již použití dvou desetibitových vstupů znamenalo 1,05 milionu iterací, což by z hlediska urychlení nemělo valný význam, o dalších bitových hloubkách nemluvě. Zde by bylo nutné používat menší tabulky (maximálně 8bitové) a požadované hodnoty získávat lineární interpolací sousedních hodnot. Naštěstí tento přístup není nikde nutný.

Všechny filtry jsem naimplementoval do odpovídajících bitových hloubek, pouze filtr křivky je pro všechny bitové hloubky naimplementován stejně jako 16bitový. Vstupem jsou hodnoty v odpovídající bitové hloubce rozšířené na 16 bitů (bitovým posuvem vlevo) a výsledné hodnoty jsou ořezávány do odpovídajících bitových hloubek (bitovým posuvem vpravo). Poslední, co zbývá vyřešit, je hloubka 32 bitů. K čemu je potřeba a jak na ni?

4.7 High Dynamic Range

Snahou člověka je zachycovat svět kolem sebe realisticky. Postupem času se kvalita a realističnost záznamu zlepšovala díky vývoji použitých technologií. Problém, který se u záznamu obrazu vyskytl, byl v tom, že v reálném světě nalezneme velké rozdíly v množství světla ve scéně.

Podle Reinhard a kol.[9] je rozdíl množství světla v nočních scénách a ve scénách na sněhu za slunného dne až deset řádů absolutního rozsahu, rozdíl množství světla v osvětlených

a ve stinných částech jedné scény pak může být i více než čtyři řády rozsahu. Tento rozdíl v rámci jedné scény se nazývá dynamický rozsah scény a cílem HDR je jej celý přenést do pořízeného snímku. První, kdo se s tímto problém setkal již v 19. století, byl Hippolyte Bayard, který přišel s nápadem vytvořit kompozici pomocí více negativů. Největší vlna vývoje přišla až v 80. a 90. letech 20. století spolu s rozvojem digitální fotografie. Problémem tehdejší (i dnešní) digitální fotografie je právě omezená schopnost zachytit na snímcích dostatečně velký dynamický rozsah. Současný formát digitálních snímků zvládne pojmout maximálně dva řády rozsahu. Volbou clonového čísla a času expozice při fotografování volíme, kterou část dynamického rozsahu scény přeneseme do snímku a kterou ztratíme (bude podexponována do černé nebo přeexponována do bílé). Proto bylo již v devadesátých letech minulého století navrženo přejít na jiný formát snímků na formát schopný zachytit velký dynamický rozsah High Dynamic Range. Hlavní myšlenka nového formátu spočívá v tom, že místo klasických bitmap, které uchovávají hodnotu světlosti pixelu v položce typu byte poskytující rozsah 0255, použijeme položku typu float (tedy reprezentace pomocí plovoucí desetinné čárky), která svým rozsahem předčí i největší možné rozdíly v reálných scénách. Určitou daní za velký rozsah je pak omezená přesnost zobrazení.[10].

Hlavní výhoda HDR je tedy v tom, že nám umožňuje uchovávat hodnoty menší než 0 a větší než 1 s velkou přesností (neboli nezobrazitelné barvy). Pokud chceme použít celočíselnou reprezentaci, musíme vhodně zvolit, jaký rozsah hodnot bude odpovídat zobrazitelnému spektru a jaký bude "mimo". Když bychom zvolili příliš úzký rozsah zobrazitelných hodnot, může docházet k příliš velkým zaokrouhlovacím chybám. Když naopak zvolíme příliš velký rozsah hodnot pro zobrazitelné spektrum, mohlo by docházet k tomu, že by rozsah nevystačil. Tato varianta by tedy vyžadovala hlubší analýzu toho, jakých hodnot můžou různé obrazy v HDR (v plovoucí desetinné čárce) nabývat. Předpokládám, že právě z tohoto důvodu pracuje Adobe Photoshop (a další známé nástroje) s HDR v plovoucí desetinné čárce, anebo je nám nabídnuto "namapovat" obraz do hloubky 8, 16 bitů.

K urychlení výpočtů s plovoucí desetinnou čárkou je teoreticky možné použít opět LUT, ale velikost takové tabulky by mohla být maximálně někde okolo 16 bitů. Předpočítat větší než 16bitové tabulky a vyhledat hodnoty v ní by bylo časově náročnější než vypočítávat operace přímo pro každý obrazový bod. Chtějme tedy předpočítávat 16bitovou tabulku. Hodnoty této tabulky by byly vypočítávány pro operace, ve kterých by bylo vstupem číslo s plovoucí desetinnou čárkou, tvořené zleva 16 různými bity a doplněné dalšími 16 nulovými bity. Tabulka by tedy pokrývala znaménkový bit, 8 bitů exponentu a 7 bitů mantisy. Zbylých 16 bitů mantisy by bylo nutné interpolovat. Tato interpolace by se neobešla bez dvou operací násobení, součtu a bitového posuvu o 16 vpravo. Jelikož většina dosud použitých tabulek slouží k náhradě jednoduchého výpočtu násobení (jednou je zastoupeno dělení a jednou mocninná funkce), ztrácí takto složitě realizovaná vyhle-

dávací tabulka smysl. Výhodné by mohlo být použití leda pro mocninnou funkci, tam je na druhou stranu diskutabilní přesnost (neboť ze 7 bitů mantisy bychom interpolací chtěli získat 23 bitů).

Shrnutím předchozího zjištění poměrně jasně vyplývá, že použití vyhledávacích tabulek nebude pro 32 bitů na kanál tím vhodným urychlením. Využití vyhledávacích tabulek pro plovoucí desetinnou čárku by mohlo být použito pro poloviční délku, tedy half. Jednoznačně bychom však museli přejít k používání výpočtů a porovnání s plovoucí desetinnou čárkou a tím bychom běh filtru také značně zpomalili. Pokud se podíváme do konkurenčního softwaru, zabývajícího se úpravou HDR obrazu, zjistíme, že náhledy již zdaleka nefungují hladce, nebo (v Adobe Photoshop) nejsou některé nástroje vůbec podporovány. Ze shrnutí vyplývá, že realizace filtrů v 32bitové hloubce není buďto urychlována vůbec, nebo možná využívá k zvýšení rychlosti GPU, čímž se dostává mimo rozsah této práce.

4.8 Porovnání výsledků

Porovnání rychlostí běhu filtrů probíhalo na notebooku MSI s dvoujádrovým procesorem Intel(R) Core(TM)2 Duo T7250 (2,0GHz) a s pamětí 2GB RAM na 64bitovém systému Windows 7 Professional. Každý filtr byl stokrát spuštěn a z těchto sto měření časů byla vybrána nejmenší hodnota, abychom zamezili zkreslením okolními vlivy. Ty mohou představovat jiné běžící procesy (ačkoliv nebyly během testů spuštěny jiné aplikace, nemůžeme vyloučit, že bude náš výpočet pozastaven během některého systémového procesu). Měření probíhalo na dvou různých velikostech obrazu: 2500 x 4000 (10,0 milionů bodů) a 1920 x 1080 (FULL HD rozlišení = 2,1 milionu bodů). Tyto hodnoty byly vyneseny do následující tabulky pro operace v hloubce 8 bitů:

Filtr	Urychlení	Doba běhu (ms) 10M	Doba běhu 2,1M
Křivky	ano	370	92
Sytost (míchání kanálů)	ne	662	135
Sytost (míchání kanálů)	ano	366	75
Odstín, sytost (HSL)	ne	2613	521
Odstín, sytost (HSL)	ano	431	85
Živost (HSL)	ne	3110	618
Živost (HSL)	ano	433	86
3-Way Filter	ne	826	164
3-Way Filter	ano	425	91

a do druhé tabulky pro operace v hloubce 16 bitů:

Filtr	Urychlení	Doba běhu (ms) 10M	Doba běhu 2,1M
Křivky	ano	373	93
Sytost (míchání kanálů)	ano	368	75
Odstín, sytost (HSL)	ano	436	89
Živost (HSL)	ano	499	98
3-Way Filter	ano	445	99

5 Závěr

V této bakalářské práci jsem nejprve uvedl různá využití nástroje křivky, princip jeho fungování a použití tohoto nástroje v Adobe Photoshop. Dále jsem vysvětlil pojmy odstín a sytost, jejich význam v barevných prostorech HSV a HSL a význam barevných prostorů HSV a HSL z hlediska lidského vnímání a zároveň počítačové interpretace, jejich drobné rozdíly. Poté jsem uvedl, jak lze přistupovat k úpravám odstínu a sytosti, a to jak pomocí již existujících, k tomu určených nástrojů (například v editoru Adobe Photoshop), tak pomocí speciálních postupů na nástrojích, které s odstínem a sytostí na první pohled téměř nesouvisí. Na základě hlubší analýzy výsledků těchto přístupů jsem vybral nejlepší zástupce, které má cenu implementovat (jsou jimi Sytost pomocí míchání kanálů, Odstín a sytost, Živost). Dále jsem rozebral, k čemu slouží takzvaný 3-Way Filter.

V další kapitole jsem analyzoval, jak tyto funkce nejjednodušeji naimplementovat jako funkce napsané v jazyce ANSI C. U křivek jsem vyšel z již odzkoušených implementací mého předchůdce Jana Kučery. U odstínu pomocí míchání kanálů jsem vyšel z pracovního postupu, jakým lze efekt úpravy odstínů uživatelsky docílit pomocí jednoduchého nástroje míchání kanálů. U odstínu a sytosti jsem vyšel z definice barevného prostoru HSL (stejně tak pro živost) a vizuální výsledky těchto tří nástrojů jsem opět důkladně zanalyzoval a vyhodnotil. Nakonec u 3-Way Filteru byla potřeba rozlišovat během výpočtu jas, proto jsem vyšel z definice barevného prostoru YC_oC_g, který byl pro naše účely nejvhodnější.

V poslední kapitole bylo mým cílem tyto nástroje co nejvíce urychlit. Poté zvážit, pro jaké bitové hloubky se vyplatí je implementovat, a své rozhodnutí podložit. Nakonec byly jednotlivé nástroje změřeny z hlediska časové náročnosti a výsledky vyneseny do tabulek.

Zde nebude na škodu porovnat, co bylo naším cílem a co bylo nakonec splněno. Snažili jsme se, aby nástroje fungovaly co možná nejrychleji a aby bylo možné sledovat náhled na tyto úpravy v reálném čase. To vyžadovalo, aby se operace prováděly do 100 ms pro určitou velikost obrazu. Toto bylo docíleno pro velikost obrazu 1920 x 1080, což odpovídá standardům FULL HD monitoru. Z mého pohledu je toto rozlišení dostatečné, pokud uvažujeme o rozlišení náhledu. Pokud by chtěl někdo oponovat, že složením monitorů můžeme získat náhled mnohem větší, byl obraz testován i na rozlišení 2500 x 4000. Zde se sice nepodařilo dostat hodnoty pod 300 ms, ale dle mého názoru by toto rozlišení patrně nebylo obsluhováno pouze výše uvedenou testovací sestavou. Z tohoto důvodu značím jako stěžejní výsledky pro obraz v rozlišení FULL HD. Pro tento obraz byl úkol splněn (dokonce nejen pro 8bitové, ale i 16bitové obrazy).

Z analýzy chyb (pro 8bitovou hloubku) uvedené v příloze C bylo zjištěno, že největší

chyba nastává při použití nástroje odstín a sytost na prostoru HSL. Nejhorší případ nastává v 1.,3. a 5. sextantu odstínu při světlostech do 128, kde nejvyšší možná chyba 3,25 (oproti výpočtu v double) nastává pro střední z kanálů RGB, maximální z kanálů může být zatížen chybou až 2,25 a minimální z kanálů chybou 1,75. Velmi podobná je situace u živosti, ta také závisí na převodu z HSL do RGB, ale na rozdíl od předchozího případu úprava kanálu S nezatěžuje lineární chybou, nýbrž mocninou. Pro velmi málo saturované barvy by podle analytického výpočtu chyby docházelo až k chybě okolo 60. Zde nezbylo než vyzkoušet, k jakým chybám může dojít ve skutečném obraze. Na obrazech B.1, B.2, B.3 je srovnání umělého obrazu, který obsahuje takovéto hodnoty a úpravu živosti, v Adobe Photoshop a v našem filtru. Z výsledku poměrně jasně vyplývá, že k žádným znatelným odchylkám nedochází (až na sílu filtru samotného). V nástroji sytost (mícháním kanálů) dochází k maximální chybě 1,5 na kanál. Výpočet nástroje křivky probíhá v plovoucí desetinné čárce, chyba je zde maximálně zaokrouhlovací, tedy 0,5 na kanál. Ve 3-Way filtru může dojít na kanálech R a B k chybě 2,5 a na kanálu G k chybě 1,5. Pokud bychom chtěli využívat tyto filtry k přesným koloristickým operacím, byly by tyto chyby znepokojivé, z hlediska manuálních nástrojů na úpravu obrazu jsou to však celkem přijatelné hodnoty. Pokud by uživatel využíval maximální hodnoty filtrů a nebyl by s výsledkem spokojen, může využít přesnějších výpočtů ve vyšší bitové hloubce.

Z uvedených výsledků je patrné, že mnou vytvořené filtry lze bez problémů využít v připravovaném bitmapovém editoru. Naším dalším cílem může být vyřešení práce s HDR obrazem. Ta v dnešní době vyžaduje mnoho nástrojů, které by bylo vhodné komplexněji shrnout a v lepším případě i zefektivnit.

Literatura

[1] *Wikipedie, Otevřená encyklopedie.*
Adobe Photoshop.
Dostupné z: http://cs.wikipedia.org/wiki/Adobe_Photoshop

[2] Jan Kučera, *Barevné korekce videa.*
Bakalářská práce 2007.

[3] Mark Fairchild, *Color Appearance Models: CIECAM02 and Beyond*
Tutorial slides for IS&T/SID 12th Color Imaging Conference.

[4] *DigitalPhoton.net, Color vibrance in Photoshop*
Dostupné z: http://www.digitalphoton.net/color-vibrance-in-photoshop/

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

[6] *Wikipedia, The Free Encyclopedia.*
HSL and HSV.
Dostupné z: http://en.wikipedia.org/wiki/HSL_and_HSV

[7] *Multimedia Wiki, YCoCg.*
Dostupné z: http://wiki.multimedia.cx/index.php?title=YCoCg

[8] *Intel, Software Documentation.*
Color space convert documentation.
Dostupné z: http://software.intel.com/en-us/

[9] Reinhard E., Stark M., Shirley P., Ferwerda J., *Photographic tone reproduction for digital images.*
ACM Trans. Graph. (special issue SIGGRAPH 2002) 21, 3, 267276.

[10] Michal Havlena, *Pořizování HDR dat.*
Diplomová práce.
Dostupné z: http://mida.wz.cz/hdr/files/Text.pdf

A Příloha

Výtah z Projektu 5

Úkol projektu

Mým úkolem je prozkoumat grafickou opensource knihovnu GEGL. Zjistit její možnosti, popsat její funkce a na základě zjištěných skutečností rozhodnout, zda (a případně nakolik) je vhodná pro použití do našeho projektu.

Úvod do knihovny GEGL

GEGL (Generic Graphics Library) je knihovna, která slouží k nedestruktivní úpravě obrazů. Jedná se o svobodný software, který se dá redistribuovat a měnit podle podmínek uvedených v GNU LGPL a GNU GPL licencích (viz. gegl.org). Zpracování obrazu je založeno na průchodu grafem, jehož uzly tvoří jednotlivé dílčí úkony (načítání obrazu do bufferu, dílčí úpravy obrazu, ukládání apod.). K urychlení zpracování lze v této knihovně pracovat s obrazovými daty pomocí bufferů (GegelBuffer) větších než RAM. Díky knihovně BABL, kterou GEGL používá, nabízí podporu široké škály barevných modelů a formátů uložení pixelů jak na vstupu, tak na výstupu. Přestože vstup i výstup podporuje knihovna ve formátech 8bitový a 16bitový integer a 32bitový float, zpracování a manipulace s daty probíhá pouze v 32bitovém float. Díky podporovaným knihovnám zvládá GEGL zpracovat data z různých formátů vstupních souborů, jako jsou PGN, JPEG, SVG, EXR, RAW, FFmpeg, V4L a jiné. K uchování formátů podporovaných knihovnou BABL je zde použit GeglBuffer, který mimo jiné podporuje tzv. Mipmapping. Zpracování probíhá po soustech zpracovávají se podoblasti a závislosti mezi nimi. K samotné úpravě obrazu využívá své operace (Gaussian blur, Bilateralfilter, Invert, a další) a operace, které mu poskytují externí podporované knihovny (SVG). Kromě korekce barev zvládá GEGL například renderovací operace (Perlinnoise) nebo práci s textem pomocí knihoven pango a cairo.

Operace, funkčnost, použitelnost GEGL

Jak už jsem v úvodu zmínil, GEGL sice zvládá načítat a ukládat data kromě float v 8bitovém a 16bitovém integer, ale veškeré operace sloužící k úpravě obrazu jsou prováděny ve float. Je to velmi zásadní a pro naše potřeby největší nedostatek. Pokud se podrobněji podíváme, jak je operace v GEGL realizována, zjistíme, že na vstup operace přichází buffer. Ten může být v jakémkoliv formátu, který BABL nabízí, může zde být různý počet kanálů, každý kanál může mít různý význam (jednou tři kanály představují RGB, jindy mohou představovat Lab) a data pixelů mohou být uložena ve formátu 8bitový integer, 16bitový integer a 32bitový float. Před každou operací nastává fáze prepare, která naformátuje data vstupního a výstupního bufferu na určitý barevný model a formát uložení dat (většinou volí barevný model RGBA a formát uložení pixelů vždy 32bitový float). Můžeme se jen domnívat, proč šli autoři knihovny právě touto cestou. Velmi pravděpodobné je, že měli málo času do vydání na to, aby jednotlivé operace napsali pro všechny rozmanité formáty dat. Pro naše účely však v tomto řešení vidíme spoustu nedokonalostí.

Obrazová data uložená v souboru jsou v naprosté většině v integer, nikoliv ve float. Z této skutečnosti vyplývá, že v každé triviální úpravě obrazu bude docházet k dvojímu zbytečnému převodu (z int na float a naopak).

Operace s float jsou časově náročnější než v integer. Mnoho operací neobsahuje dělení a často je použití float zbytečné. Použití výchozího barevného modelu RGBA není ve většině operací vhodné, GEGL používá pro většinu operací barevný model RGBA, což není vhodné z více důvodů. Různé barevné modely se používají proto, že výsledky upravujících operací (kontrast, světlost, roztažení histogramu atd.) jsou pro různé barevné modely různé (úprava kontrastu v RGB se bude pro většinu obrazů lišit od úpravy kontrastu v Lab) a některé úpravy v prostoru Lab jsou v RGBA nenapodobitelné. Pokud budu chtít například operací kontrast upravit obraz ve stupních šedi, narazím na další nevýhodu defaultního použití RGBA. Kromě dvojího zbytečného převodu se bude kontrast provádět pro tři kanály místo pro jeden. Poslední nevýhodou GEGL je, že jednotlivé operace nejsou příliš optimalizované. Například je zde úplná absence použití LookUp tabulek, které by urychlily některé operace, například gama korekci.

Pokud se rozhodneme pro použití této knihovny a její přepsání, musíme počítat se značným nárůstem velikosti kódu, který jsme zaznamenali již pro jednoduchou operaci invert. Daleko větší nárůsty můžeme očekávat u složitějších operací, kde budeme zavádět výpočty pomocí LookUp tabulek. Dále, pokud se rozhodneme pro přepsání knihovny GEGL, se musíme rozhodnout, v jakém formátu budeme předávat informace o formátu obrazových dat (barevný prostor a bitová hloubka). Zatím jsem pro názornost provedl pouze zjednodušený přepis operace invert, ve které jsem provizorně tyto proměnné nadefinoval (a

pro testování větvení je měním přímo v této operaci). Musíme se tedy rozhodnout mezi dvěma cestami:

a) Rozhodovat o modelech na základě GeglBuffer formátu, který je předáván jako buble objekt spolu s bufferem. Dosud se mi však nepodařilo prokázat, zda jde gegl_buffer_get_format volat i uvnitř operací na zjištění formátu vstupu. Jeho nespornou výhodou je však to, že je již v GEGL pevně zabudován. Nevýhodou je jeho svázání s použitím knihovny BABL k převodům mezi barevnými prostory a jejich spravováním. Knihovna BABL sice nabízí celkem rozmanité množství barevných prostorů, ale převody mezi nimi často nejsou možné vůbec, nebo jsou realizovány několika dílčími převody. Pokud bychom chtěli používat knihovnu BABL, je velmi pravděpodobné, že bychom ji museli dovybavit lepšími převody a přizpůsobit si tvorbu nových modelů.

b) Vytvořit novou datovou strukturu splňující naše potřeby, tu předávat atributem operací a její hodnotu stále uchovávat. Tato možnost má také své nevýhody. Hlavní je ta, že GEGL je v současné době poměrně robustní knihovna a přidání této struktury rozhodně nebude snadné, troufám si říci, že bude vyžadovat úpravy napříč celou knihovnou. Pokud bychom chtěli mezi barevnými prostory přecházet, museli bychom ji stejně asociovat s formáty a převody knihovny BABL alespoň do té doby, než se nám podaří naimplementovat převody vlastní.

Rozhodování mezi těmito variantami není pouze mou záležitostí a mělo by se určitě týkat i dalších členů týmu, zejména p. Jirkovského, který se zabývá barevnými prostory, jejich popisem a převodem mezi nimi. Určitě bude vhodné, aby prozkoumal knihovnu BABL, zda by šla pro jeho záměry využít. Pokud ne, musíme uvažovat i o možnosti vytvoření nové knihovny. Jelikož je GEGL úzce spjat s knihovnou BABL, její výměna za novou knihovnu zaštiťující správu barevných prostorů by mohla přinést řadu nečekaných problémů. Ty by pak bylo velmi těžké řešit, jelikož není k dispozici žádná dokumentace, která by podrobněji popisovala jádro GEGL. Ani vývojáři GEGL mi žádnou podrobnější neposkytli, nebo prostě žádná taková není. Jejich reakce jsou zatím takové, že se nás snaží odradit od rozepisování operací podle obrazového formátu.

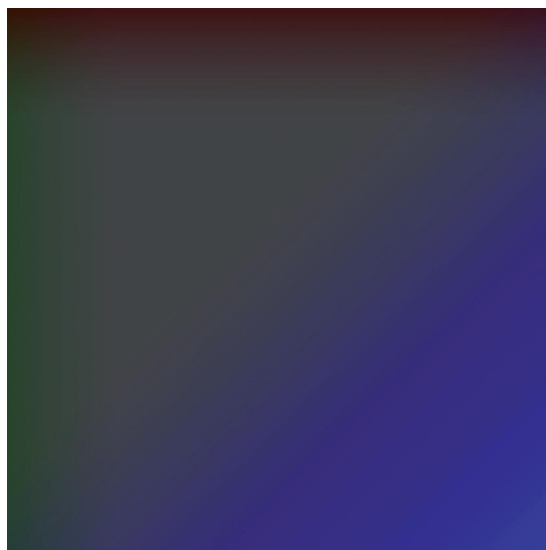
Závěr

Knihovnu GEGL nemůžeme pro naše účely ponechat bez úprav. Je zapotřebí přepsat jednotlivé operace, aby fáze prepare nejprve zjistila formát dat. Podle toho by se provedl nejprve formát vstupu a výstupu operace. Dále by se podle toho, o jaký formát se jedná, provedla odpovídající varianta fáze process. Dalším cílem přepisu by bylo, tam, kde to jde, použít k výpočtu LookUp tabulky. Aby vše fungovalo podle našich představ, museli

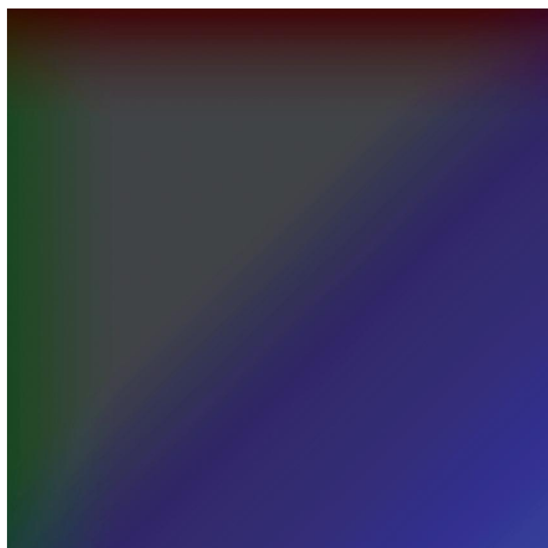
bychom vše řídit podle formátu obrazových dat. Zde se budeme muset rozhodnout pro jednu z výše uvedených variant, jak tento formát předávat. Ať už se rozhodneme pro jakoukoliv, bude to vyžadovat rozsáhlé a náročné úpravy (zejména pro nedostatečnou dokumentaci knihoven GEGL a BABL). Knihovna GEGL je svižně vyvíjený projekt, který má bohužel trochu odlišné cíle, než jsou ty naše. Jedná se patrně o nejvhodnější knihovnu, to však z důvodu, že dnes neexistuje jiná aktivně vyvíjená knihovna v tomto zaměření.

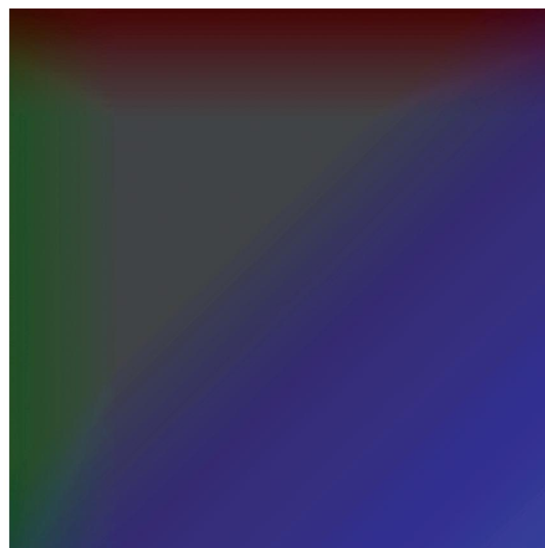
B Příloha

Vizuální výsledky



Obr. C.1: Originální obraz.





Obr. C.2, C.3: Vlevo živost v Adobe Photoshop (+75), vpravo živost pomocí vlastního filtru (0,6 - maximální dosycení).

Obr. C.4: Originální obraz.

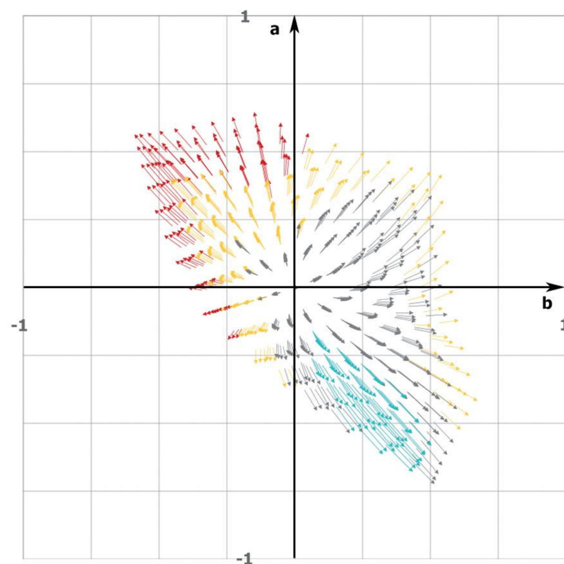
Obr. C.5, C.6: Vlevo živost pomocí vlastního filtru, hodnota 0,6 - maximální dosycení), vpravo hodnota 1,4 maximální odsycení.

Obr. C.7, C.8: Sytost pomocí filtru sytost (míchání kanálů), vlevo hodnota +0,3, vpravo -0,3.

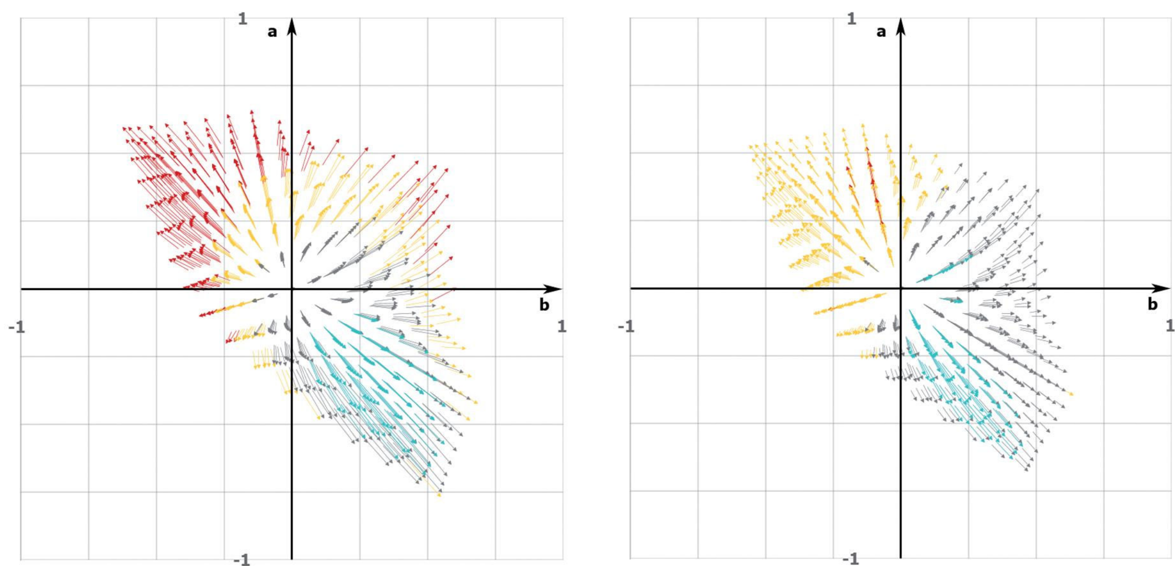
Obr. C.9, C.10: Sytost pomocí filtru odstín a sytost, vlevo hodnota 1,3, vpravo 0,7.

Obr. C.11, C.12: Vlevo odstín pomocí filtru odstín a sytost, hodnota +40 (odpovídá +60 ve Photoshopu), vpravo hodnota 80 (odpovídá 120).

Obr. C.13, C.14: Vlevo 3-Way Filter (Co={10,-15,-40},Cg={0,-15,40}, vpravo křivky (body X1=16640, Y1=6400, X2=48640, Y2=57600.



Obr. C.15: Graf sytosti pomocí odstín sytost (vlastní filtr).



Obr. C.16, C.17: Vlevo graf sytosti pomocí míchání kanálů, vpravo živost (vlastní filtry).

C Příloha

Analýza chyb urychlené implementace

Jelikož jsme při urychlování metod v mnoha případech přešli od výpočtu v double, k výpočtu v integer, musíme zjistit, jakých chyb se tím dopouštíme.

První nástroj, který musíme přezkoušet je odstín pomocí míchání kanálů. Je zde pouze jeden výpočet pro každý kanál, před kterým jsme provedli zaokrouhlení.

$$R' = \alpha\, R - \beta\, G - \beta\, B\,,$$

každý kanál je zatížen chybou $0,5 + 0,5 + 0,5 = 1,5$

Analýza nástroje Odstín a sytost prostoru HSL je poněkud složitější. Převodem z RGB do HSL nám žádná chyba nevzniká, výpočty jsou prováděny v plovoucí desetinné čárce a chyba. První chyba tedy vznikne, když výsledné HSL uložíme v integeru. Kanál H je zatížen chybou h, kanál S chybou s a kanál L chybou l. Platí $h_0 = s_0 = l_0 = 0,5$. Během úprav dochází ke změně kanálu S a tím i k možnému zvýšení chyby $s_1 = 1,5$ ($s_1 = 2s_0 + 0,5$; kde 0,5 je další zaokrouhlovací chyba)
V převodu HSL do RGB pochopitelně dochází ke kvantizaci chyb. Výpočet kanálu RGB_{max} se nám větví podle hodnoty jasu daného pixelu. Pro hodnoty L do 128 platí:

$$RGB_{max} = L + l + \left((L + l)\,(S + s)\,/256 \right)$$

pokud budeme uvažovat maximální hodnoty pro L a S, dostáváme výslednou chybu $rgb_{max} = 1,5l + s$
Pro hodnoty L nad 128 platí:

$$RGB_{max} = L + l + S + s - \left((L + l)\,(S + s)\,/256 \right)$$

tedy po dosazení maxim vychází chyba $rgb_{max} = s + l - (s + l) = 0$ Pro RGB_{min} platí výpočet:

$$RGB_{min} = L + l + L - RGB_{max} - rgb_{max}$$

tedy dostáváme, že chyba pro L menší 128 bude $rgb_{min} = 2l - (1,5l + s) = 0,5l - s$ a pro L větší, rovno 128 bude $rgb_{min} = 2l - 0 = 2l$
Při výpočtu RGB_{mid} mohou nastat opět v zásadě dvě varianty:

$$RGB_{mid} = RGB_{max} + rgb_{max} - V$$

nebo

$$RGB_{mid} = RGB_{max} + rgb_{max} + V$$

kde

$$V = ((RGB_{max} + rgb_{max} - RGB_{min} - rgb_{min}) * fract) /256$$

kde fract je zatížen chybou h a nabývá hodnot 0-255. Z toho vyplývá, že v nejhorším případě může být chyba $v = rgb_{max} + rgb_{min} + h$, pak tedy chyba $rgb_{mid} = rgb_{max} - v$, nebo $rgb_{mid} = rgb_{max} + v$, budou po dosazení:

$$rgb_{mid} = rgb_{max} - rgb_{max} - rgb_{min} - h = -rgb_{min} - h$$

nebo

$$rgb_{mid} = rgb_{max} + rgb_{max} + rgb_{min} + h = 2rgb_max + rgb_{min} + h$$

Nyní můžeme přistoupit ke konkrétním dosazením.
Pokud je L < 128 a barva se nachází v 1.,3.,nebo 5. sextantu, bude maximální chyba následující:

$$\begin{aligned} rgb_{max} &= 1,5l + s = 2,25 \\ rgb_{min} &= 0,5l + s = 1,75 \\ rgb_{mid} &= h + 2,5l + s = 3,25 \end{aligned}$$

Pokud je L < 128 a barva se nachází v 2.,4.,nebo 6. sextantu, bude maximální chyba následující:

$$\begin{aligned} rgb_{max} &= 1,5l + s = 2,25 \\ rgb_{min} &= 0,5l + s = 1,75 \\ rgb_{mid} &= h + 0,5l + s = 2,25 \end{aligned}$$

Pokud je L <= 128, pro všechny odstíny bude maximální chyba následující:

$$\begin{aligned} rgb_{max} &= 0 \\ rgb_{min} &= 2l = 1 \\ rgb_{mid} &= h + 2l = 1,5 \end{aligned}$$

Ve filtru živost dochází k identickým převodům jako v nástroji odstín a sytost. Jedinou změnou je tedy chyba generovaná přímo úpravou v HSL. Zde se pro většinu barev jedná o chybu srovnatelnou a mnohdy mnohem menší než u předchozího případu. Horší výsledky mohou nastat u barev velmi blízkých achromatickým barvám, zde se může chyba razantně zvýšit. To je ovšem poměrně jasný důsledek toho, že jsou tyto málo syté barvy

razantně dosycovány, což je vlastně princip tohoto filtru.

Pokud ve 3-Way filteru budeme uvažovat o tom, že při převodu z RGB bude každý z kanálů Y$C_o$$C_g$ zatížen zaokrouhlovací chybou 0,5, při úpravě kanálů C_o a C_g bude ke každému z těchto dvou kanálů přičtena zaokrouhlovací chyba 0,5. Pak nejhorší možná chyba kanálů R a B bude po zpětném převodu tvořena součtem chyb všech tří kanálů Y$C_o$$C_g$, tedy chybou 2,5. Kanál G bude zatížen chybami kanálů Y a C_g, tedy chybou 1,5.