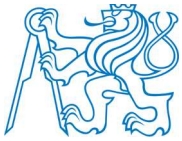


ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

Fakulta elektrotechnická
Katedra elektroenergetiky

Vytvoření aplikace pro analýzu jasů HDR fotografií

Application for luminance values extraction from HDR images

Diplomová práce

Studijní program: Elektrotechnika, energetika a management
Studijní obor: Elektroenergetika

Vedoucí práce: Ing. Marek Bálský, Ph.D.

Bc. Lubomír Nepil

Praha 2018

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Nepil** Jméno: **Lubomír** Osobní číslo: **420417**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra elektroenergetiky**

Studijní program: **Elektrotechnika, energetika a management**

Studijní obor: **Elektroenergetika**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Vytvoření aplikace pro analýzu jasů HDR fotografií

Název diplomové práce anglicky:

Application for luminance values extraction from HDR images

Pokyny pro vypracování:

1. Analýza dosavadních metod výpočtu jasů z jednotlivých digitálních fotografií s nízkým dynamickým rozsahem.
2. Softwarové možnosti rozšíření dosavadní metody na HDR sadu digitálních fotografií.
3. Vytvoření a verifikace aplikace pro HDR analýzu jasů sady digitálních fotografií.

Seznam doporučené literatury:

[1] FIŠERA, M. Digitální fotografie a zorné pole lidského oka. ČVUT v Praze, FEL, 2005. Diplomová práce.
[2] ČSN EN 13032. Světlo a osvětlení - Měření a uvádění fotometrických údajů světelných zdrojů a svítidel.
[3] HABEL, Jiří, et al. Světlo a osvětlování. Praha: FCC Public, 2013. 438 s. ISBN 978 80 86534 21 3.

Jméno a pracoviště vedoucí(ho) diplomové práce:

Ing. Marek Bálský, Ph.D., katedra elektroenergetiky FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **18.01.2018** Termín odevzdání diplomové práce: **25.05.2018**

Platnost zadání diplomové práce: **30.09.2019**

Ing. Marek Bálský, Ph.D. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Ing. Pavel Ripka, CSc. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

. | Datum převzetí zadání

Podpis studenta

© ČVUT v Praze, Design: ČVUT v Praze, VIC

Čestné prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne ……. …….

Podpis

Poděkování

Chtěl bych poděkovat panu Ing. Marku Bálskému, Ph.D. za odborné vedení práce a cenné rady, které mi pomohly tuto práci zkompletovat.

Abstrakt

Tato práce se zabývá vytvořením softwarového nástroje pro využití digitální fotografie jako nástroje pro měření jasů ve světelně technické praxi. V první části je provedena rešerše současného stavu problematiky výpočtu jasů z digitální fotografie. Druhá část se zabývá analýzou metod tvorby digitálních fotografií s vysokým dynamickým rozsahem (HDR) a v poslední části je popsána metodika tvorby softwarové aplikace pro výpočet jasů z HDR fotografie a její verifikací na základě laboratorních měření.

Klíčová slova

jas, jasoměr, měření jasu, jas scény, software, jasová analýza, HDR, digitální fotografie, digitální fotoaparát, RGB, Lab, barevný systém, barevná soustava, JavaFX, Java

Abstract

The aim of this thesis is to create a software tool that aids the use of digital photography in luminance measurement in the lighting industry. In the first section, the current state of the use of digital photography in luminance measurements is discussed. The next section follows up by analyzing the methodology of high dynamic range photography (HDR). Lastly, the development of the software tool is described and verified with the use of precise laboratory measurements.

Keywords

luminance, luminance meter, luminance measurement, scene luminance, software, luminance analysis, HDR, digital photography, digital camera, RGB, Lab, color system, JavaFX, Java

Obsah

Úvod

Technologie digitální fotografie je již mnoho let běžnou součástí našich každodenních životů. Téměř každý člověk dnes vlastní mobilní telefon, který v sobě zahrnuje digitální fotoaparát. S mobilním telefonem se již dnes dají pořizovat fotografie s kvalitou pro běžného uživatele dostačující a tato kvalita se s příchodem novějších zařízení pomalu ale jistě zlepšuje. Vlastnictví klasického digitálního fotoaparátu je dnes prakticky výhradou profesionálních fotografů či nadšenců do fotografování, kteří si pořizují kvalitní zrcadlovky.

Digitální fotoaparát však lze využít i v trochu jiném odvětví, než je běžné fotografování. K tomu je nutné si uvědomit, jak toto zařízení (fotoaparát) funguje – tedy, že vytváří digitální fotografii, která je tvořena miliony pixelů, kde každý pixel má přiřazenou číselnou hodnotu, závislou na vlastnostech světla dopadajícího na fotocitlivý senzor v daném místě. Zmíněnými vlastnostmi dopadajícího světla jsou světelně-technické veličiny, mezi něž patří i jas. Z toho tedy vyplývá, že hodnota přiřazená pixelu je nějakým způsobem úměrná jasu.

Jak a proč je číselná hodnota pixelu úměrná jasu je popsáno v [1]. Tato práce plynule navazuje na zmíněnou práci, která popisuje způsob kalibrace digitálního fotoaparátu pro měření jasů a postup, jak z pořízených snímků z kalibrovaného fotoaparátu získat hodnoty jasu v každém pixelu. V závěru [1] je výzva, která říká, že je nutné pro tuto problematiku vytvořit dedikovaný software, jehož princip je tam také naznačen.

Právě tvorbou takového softwaru se tato práce zabývá.

1 Analýza dosavadních metod výpočtu jasů z jednotlivých digitálních fotografií s nízkým dynamickým rozsahem

1.1 Jas svazku světelných paprsků

Jas svazku světelných paprsků je světelně technická veličina, na kterou bezprostředně reaguje zrakový orgán. Tato veličina je obecně určena prostorovou a plošnou hustotou světelného toku přenášeného paprsky.

Jas je určen vztahem

$$L_{OP} = \frac{d^2\Phi}{d\Omega\ dA_n} \qquad (cd \cdot m^{-2},\ lm,\ sr,\ m^2) \tag{1.1}$$

kde

- L_{OP} je jas svazku světelných paprsků ve směru osy OP svazku,
- $d\Omega$ je prostorový úhel, ve kterém se paprsky šíří
- dA_n je ploška kolmá k ose svazku paprsků, na níž se realizuje plošná hustota světelného toku

V prostředí, které světelné paprsky pohlcuje, nebo je vyzařuje či rozptyluje, se od bodu k bodu mění světelný tok přenášený svazkem paprsků, a tudíž se úměrně tomu mění i jas.

Pokud však světlo putuje bezeztrátovým prostředím vymezeným dvěma otvory o velikosti plochy dA_1 a dA_2 v libovolně umístěných stínítkách A_1 a A_2 (obr. 1) a jsou-li rozměry otvorů dA_1 a dA_2 zanedbatelné ve srovnání se vzdáleností l mezi stínítky, dostaneme z rovnice (1.2) tyto vztahy:

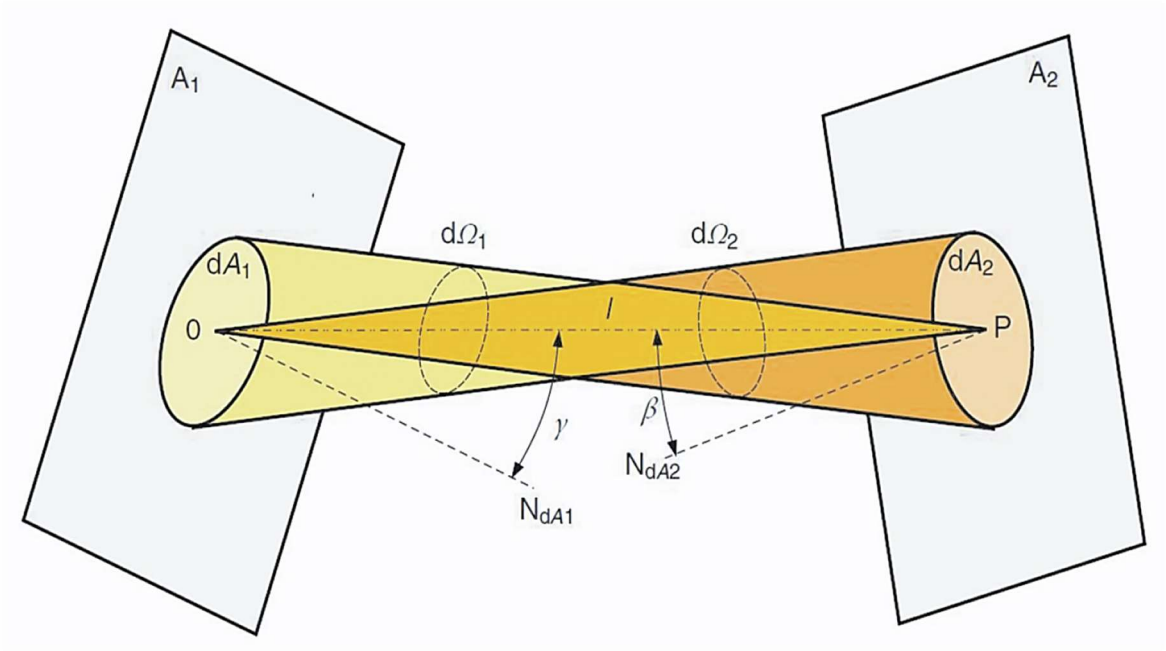
a) pro jas L_{OP} svazku paprsků sbíhajících se v prostorovém úhlu z plošky dA_1 do bodu P

$$L_{OP} = \frac{d^2\Phi}{d\Omega_1\ dA_2 \cos\beta} \qquad (cd \cdot m^{-2},\ lm,\ sr,\ m^2) \tag{1.3}$$

b) pro jas L_{OP} svazku paprsků rozbíhajících se v prostorovém úhlu $d\Omega_2 = dA_2 \cos\beta\ l^{-2}$ z bodu O

$$L_{OP} = \frac{d^2\Phi}{d\Omega_2\ dA_1 \cos\gamma} \qquad (cd \cdot m^{-2},\ lm,\ sr,\ m^2) \tag{1.4}$$

Jednotkou jasu je *kandela na čtvereční metr* ($cd \cdot m^{-2}$). [2]



Obr. 1: Svazky rozbíhajících a sbíhajících se paprsků [2]

1.2 Měření jasu

Pro měření jasu povrchů vyzařujících nebo odrážejících světlo se využívají jasoměry.

1.2.1 Jasoměr

Základní součástí jasoměru je fotočlánek, který je vysoce citlivý a spektrálně přizpůsobený. Je vybaven optickou soustavou clonících a zaměřovacích prvků, které vytyčují prostorový úhel Ω zorného pole jasoměru. Paprsky z měřené plochy projdou optickým systém a dopadají na fotočlánek, kde vytvoří normálovou osvětlenost E_N, jež je úměrná světelnému toku dopadajícímu z plochy, kterou snímáme. Osvětlenost fotočlánku způsobí generaci proudových nosičů a tekoucí proud je poté měřen citlivým měřícím zesilovačem.

Střední jas L plochy vymezené prostorovým úhlem Ω je poté dán vztahem

$$L = \frac{E_N}{\Omega} \qquad (\frac{cd}{m^2}, lx, sr) \tag{1.5}$$

Rozlišujeme dva typy jasoměrů: bodový a integrační.

- Bodovými jasoměry myslíme přístroje s malým zorným (prostorovým) úhlem, většinou v řádu jednotek úhlových minut.
- Integrační jasoměry měří průměrnou hodnotu jasu ve větším zorném poli, většinou větším než 1°.

Běžnými hodnotami zorného úhlu jsou 3°, 2°, 1° nebo 1/3°, popřípadě 6′, 2′.

Obr. 2: Jasoměr od firmy Konica Minolta [3]

Jasy ploch důležitých pro vidění a zrakovou pohodu, které se nacházejí v zorném poli uživatele interiéru se měří bodovými jasoměry v několika kontrolních bodech. Tyto body jsou umístěny tak abychom mohli posoudit rozložení jasu v zorném poli při běžném směru pohledu uživatele a obvyklé výšce očí (nejčastěji u stojící osoby 150 cm, u sedící osoby 120 cm). Měří se zejména jas pozorovaného předmětu a ploch ho obklopujících, jas vzdálených ploch (např. stěn, podlahy, stropu atd.). Také se měří jasy svazku paprsků odrážejících se od velmi jasných částí povrchů, které mohou oslňovat uživatele interiéru.

Při měření jasu je třeba dbát na to, že se jasoměrem zjišťuje střední hodnota jasu měřené plochy, kterou vymezuje optika přístroje v závislosti na vzdálenosti jasoměru od měřeného povrchu. Je proto vždy nutné dbát na to, aby měřená plocha zahrnovala pouze povrch, jehož jas se posuzuje. U dnešních jasoměrů toto není problém, neboť okolí měřené plochy pozorujeme v okuláru a měřená oblast bývá v zorném poli vyznačena kroužkem.

Podle mezinárodního doporučení [4] se jasoměry člení do čtyř tříd přesnosti označených písmeny L, A, B, C. Uvedeným třídám přesnosti odpovídají největší celkové přípustné chyby jasoměru 5 %, 7,5 %, 10 % a 15 %. [2]

1.2.2 Použití jasoměru při měření rozsáhlých scén

Z bodové charakteristiky jasoměru lze vyvodit jednu velkou nevýhodu. V případě, že chceme analyzovat jasové poměry na celé scéně a jsme tak nuceni měřit síť o velkém množství měřících bodů, stává se měření velmi časově náročným. Mnohdy jsme nuceni měřit desítky či stovky kontrolních bodů na jedné scéně. V této situaci připadá v úvahu zvážit možnosti měření jasu digitálním fotoaparátem.

Digitální fotoaparát je vybaven snímačem, který je složen z jednotlivých fotodiod. Každou z těchto diod si lze představit jako samostatný jasoměr. Proto je v digitální fotografii velký potenciál, pokud potřebujeme analyzovat jas celé scény. Pořízenou fotografii softwarově zpracujeme a v tu chvíli známe jas v každém pixelu pořízeného snímku.

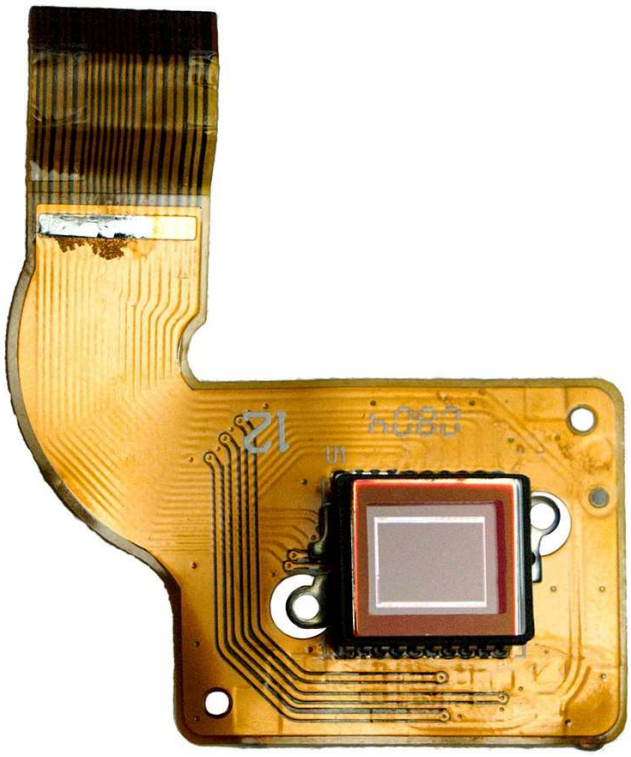
1.3 Digitální fotoaparát a digitální fotografie

Digitálním fotoaparátem obecně rozumíme zařízení, které zachycuje fotografie do digitální paměti. V dnešní době klesá rozšířenost dedikovaných digitálních fotoaparátů a pro běžné uživatele se digitálním fotoaparátem stává jejich mobilní telefon s integrovaným fotoaparátem. Samostatné digitální fotoaparáty si ale drží svojí pozici na trhu díky tomu, že dokáží pořizovat mnohem kvalitnější snímky. Tudíž pro fotografické profesionály a nadšence jsou stále nenahraditelnými.

1.3.1 Princip digitálního fotoaparátu

Srdcem celého fotoaparátu je světlocitlivá plocha (obrazový snímač), tak jako byl pro klasické fotoaparáty světlocitlivý film. Obrazové senzory slouží k převodu dopadajícího světla na elektrický náboj, který je po uzavření uzávěrky z čipu odváděn a převeden na binární signál. Vzniklý binární signál je poté pomocí mikroprocesoru upraven a převeden do některého z grafických formátu pro záznam fotografií jako např. raw, JPEG, či TIFF. Výsledný soubor je uložen na paměťové médium.

V současné době se používají dva typy snímačů – CCD a CMOS. CMOS technologie je mnohem levnější, a tedy i mnohem rozšířenější než CCD. CCD snímače jsou dnes spíše výhradou velmi kvalitních zařízení.



Obr. 3: CCD obrazový snímač na ohebném desce plošných spojů [5]

Každý fotoaparát je také vybaven optickým systémem. Typicky se jedná o objektiv s pohyblivou clonou, která usměrňuje světelné paprsky na obrazový snímač.

1.3.2 Clona

Clona je zařízení, které reguluje množství světla procházejícího objektivem fotoaparátu. Je umístěna mezi čočkami tak, aby nezastiňovala zorné pole. Velikost clony je udána pomocí clonového čísla

$$F = \frac{f}{d} \tag{1.6}$$

kde F je clonové číslo, f je ohnisková vzdálenost a d průměr otvoru clony. Obecně platí, že čím větší otvor, tím menší je clonové číslo. Množství světla dopadajícího na senzor za jednotku času je nepřímo úměrné druhé mocnině clonového čísla. Clonová čísla se proto na objektivech uvádějí v násobcích $\sqrt{2}$ (1,4 – 2 – 2,8 – 4 – 5,6 atd.). Někdy je clona uváděna jako $f / < clonové\ číslo >$.

Obr. 4: Malé a velké clonové číslo (např. 2,8 a 16) [6]

1.3.3 Doba expozice

Pomocí závěrky lze přesně nastavit dobu, po kterou bude světlo dopadat na snímač, tzv. dobu expozice neboli expoziční čas. U digitálních fotoaparátů se používá závěrka elektronická, kdy je snímač aktivní jen po určitou dobu a není třeba mechanicky zastiňovat přístup světla k senzoru. Zrcadlovky však mají klasickou štěrbinovou závěrku, která produkuje typický zvuk cvaknutí. Elektronická závěrka sama o sobě žádný zvuk nevydává, výrobci proto uměle přidávají do fotoaparátu zvukový efekt.

Obr. 5: Štěrbinová závěrka u Canon EOS 300 [7]

1.3.4 Expozice

Pro pořízení fotografie je třeba provést expozici, tedy vystavit senzor fotoaparátu světlu z fotografované scény. Stupeň expozice se udává v jednotkách *EV* (exposure value). Hodnota expozice je dána kombinací doby expozice a clonového čísla takovým způsobem, že jakákoliv kombinace těchto dvou čísel, která nám dá stejné číslo *EV* značí stejnou expozici (pro stejnou scénu), jak lze vidět v tabulce expozičních hodnot. [8]

Stupeň expozice je udáván na logaritmické stupnici o základu 2:

$$EV = \log_2 \frac{F^2}{t} \tag{1.7}$$

kde

- $F(-)$ je clonové číslo
- $t(s)$ je čas expozice

Expozice je nulová při času expozice 1 s a cloně 1.0. Zvýšení *EV* o 1 se ve fotografické terminologii nazývá změnou o jeden „stop“, a znamená snížení expozice o polovinu.

Tabulka 1:Expoziční hodnoty EV (ISO 100)

	clonové číslo												
čas (s)	1.0	1.4	2.0	2.8	4.0	5.6	8.0	11	16	22	32	45	64
60	−6	−5	−4	−3	−2	−1	0	1	2	3	4	5	6
30	−5	−4	−3	−2	−1	0	1	2	3	4	5	6	7
15	−4	−3	−2	−1	0	1	2	3	4	5	6	7	8
8	−3	−2	−1	0	1	2	3	4	5	6	7	8	9
4	−2	−1	0	1	2	3	4	5	6	7	8	9	10
2	−1	0	1	2	3	4	5	6	7	8	9	10	11
1	0	1	2	3	4	5	6	7	8	9	10	11	12
1/2	1	2	3	4	5	6	7	8	9	10	11	12	13
1/4	2	3	4	5	6	7	8	9	10	11	12	13	14
1/8	3	4	5	6	7	8	9	10	11	12	13	14	15
1/15	4	5	6	7	8	9	10	11	12	13	14	15	16
1/30	5	6	7	8	9	10	11	12	13	14	15	16	17
1/60	6	7	8	9	10	11	12	13	14	15	16	17	18
1/125	7	8	9	10	11	12	13	14	15	16	17	18	19
1/250	8	9	10	11	12	13	14	15	16	17	18	19	20
1/500	9	10	11	12	13	14	15	16	17	18	19	20	21
1/1000	10	11	12	13	14	15	16	17	18	19	20	21	22
1/2000	11	12	13	14	15	16	17	18	19	20	21	22	23
1/4000	12	13	14	15	16	17	18	19	20	21	22	23	24
1/8000	13	14	15	16	17	18	19	20	21	22	23	24	25

1.3.5 Přeexponování a podexponování snímku

Tyto termíny se používají pro charakterizování fotografií, které byly světlu vystaveny po příliš krátkou, nebo naopak příliš dlouhou dobu. Pokud byl snímač vystaven světlu moc dlouho, je příliš světlý v jasných oblastech expozice (např. na denní obloze), snímek je tedy přeexponován. Pokud naopak vystavíme snímek světlu po příliš krátkou dobu, tmavší místa snímku budou příliš tmavá. Obecně dochází ke ztrátě rozlišovací schopnosti snímku na okrajích dynamického rozsahu fotoaparátu.

Obr. 6: Vlevo je snímek podexponovaný, vpravo přeexponovaný [9]

1.3.6 Dynamický rozsah scény a fotoaparátu

Dynamický rozsah scény

Tímto pojmem rozumíme poměr mezi nejsvětlejším a nejtmavším místem na scéně. Celkový rozsah lze specifikovat v jednotkách *EV*, například dynamický rozsah 10 *EV* by znamenal poměr 1:1024 mezi nejsvětlejším a nejtmavším místem na scéně.

Dynamický rozsah fotoaparátu

Definice dynamického rozsahu fotoaparátu se trochu liší. Je to poměr mezi nejsvětlejším a nejtmavším pixelem rozlišitelným na jedné fotografii. V praxi často reálný dynamický rozsah scény překročí rozsah fotoaparátu. Ilustraci tohoto problému lze vidět na obrázku, kde je dynamický rozsah scény větší než rozsah fotoaparátu. Pokud chce v tuto chvíli fotograf vyfotit fotografii bez velkého množství šumu, musí se rozhodnout, jestli vyfotí tmavší část scény, nebo světlejší část.

Obr. 7: Dynamický rozsah scény je moc velký [10]

1.4 Jasová analýza digitální fotografie

Digitální fotografie v jakémkoliv formátu je pouze dvojdimenzionálním polem binárních čísel, kde každé číslo reprezentuje jeden pixel fotografie. Toto binární číslo obsahuje informace o barvě v systému *RGB* ve formě jednotlivých barevných složek *R* (red), *G* (green) a *B* (blue). Tyto barevné složky jsou také úměrné jasu paprsků, které dopadly na příslušnou fotodiodu. Z tohoto důvodu jsme schopni využít digitální fotografii k jasové analýze scény.

1.4.1 Barevný prostor *RGB*

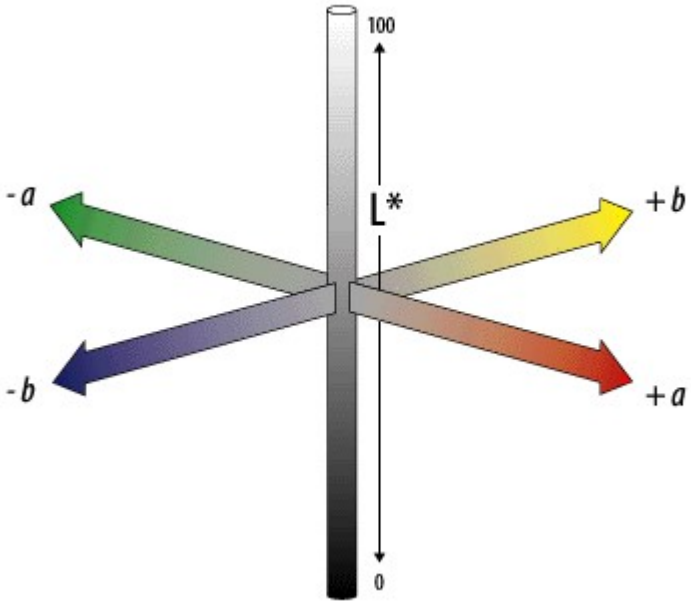
RGB systémů existuje několik, ale nejrozšířenějším z nich je systém sRGB. Tento barevný systém byl standardizován společnostmi Hewlett-Packard a Microsoft [9] pro použití na monitorech, tiskárnách a na internetu. Hodnoty jednotlivých složek jsou v rozmezí <0; 1>, např. [*R*; *G*; *B*] → [0,15; 0,45; 0,58].

V barevném systému RGB však není jasně vidět informace o jasu, k tomuto účelu je vhodný barevný prostor Lab.

1.4.2 Barevný prostor Lab

Tento barevný systém se jeví jako nejvhodnější k určení jasu. [1] Souřadnice *L* (*lightness*) je přímo úměrná jasu snímaného předmětu. Souřadnice *a* a *b* obsahují informaci o barvě. Aby nedocházelo k záměně veličiny "jas" značené "*L*" a souřadnice *L* systému Lab, bude dále tato souřadnice označována L_{Lab}.

Barevný systém Lab používaný v této práci je zkráceným názvem systému CIE L*a*b* (CIELAB). Tento barvený systém byl zaveden v roce 1976 Mezinárodní komisí pro osvětlování (CIE). K popisu barvy využívá tři základní barvy a barevný tón, sytost a jas. Vychází se z protikladů (párů barev) tzv. původních barev, tj. červená-zelená, žlutá-modrá a černá-bílá.



*Obr. 8: Grafické znázornění barevného systému L*a*b* [11]*

1.4.3 Určení hodnoty souřadnice L_{Lab} z pořízeného snímku

Jakýkoliv obraz reprezentovaný souborem bodů, které jsou popsány hodnotami *R*, *G* a *B*, lze dále zpracovat pomocí matematických algoritmů tak, že pro každý obrazový bod vypočteme hodnotu L_{Lab} v tomto bodě.

Nejvhodnějším řešením je zřejmě převedení souřadnic do standardního systému sRGB, souřadnice v tomto systému lze poté převést do systému Lab podle známých vztahů.

Pokud uvažujeme hodnoty souřadnic $R, G, B \in \langle 0, 1 \rangle$ a referenční bílé D65, můžeme nejprve aplikovat převod z barevného systému RGB na systém XYZ.

$$r = \frac{R}{12,92};\ pro\ R \le 0,04045 \tag{1.8}$$

$$r = \left(\frac{R + 0,055}{1,055} \right)^{2,4};\ pro\ R > 0,04045 \tag{1.9}$$

$$g = \frac{G}{12,92};\ pro\ G \le 0,04045 \tag{1.10}$$

$$g = \left(\frac{G + 0,055}{1,055} \right)^{2,4};\ pro\ G > 0,04045 \tag{1.11}$$

$$b = \frac{B}{12,92};\ pro\ B \le 0,04045 \tag{1.12}$$

$$b = \left(\frac{B + 0,055}{1,055} \right)^{2,4};\ pro\ B > 0,04045 \tag{1.13}$$

Z takto vypočtených hodnot *r*, *g* a *b* se určí souřadnice *X, Y, Z* pomocí vztahu:

$$[X\ Y\ Z] = [r\ g\ b] \cdot [M] \tag{1.14}$$

kde matice *M* má pro soustavu sRGB a referenční bílou *D65* následující podobu:

$$M = \begin{bmatrix} 0,412424 & 0,212656 & 0,0193324 \\ 0,357579 & 0,715158 & 0,119193 \\ 0,180464 & 0,0721856 & 0,950444 \end{bmatrix} \tag{1.15}$$

Pro výpočet souřadnic L_{Lab}, a a b použijeme následující vzorce:

$$L_{Lab} = 116 \cdot f_y - 16 \tag{1.16}$$

$$a = 500 \cdot \left(f_x - f_y \right) \tag{1.17}$$

$$b = 200 \cdot \left(f_y - f_z \right) \tag{1.18}$$

kde:

$$f_x = \sqrt[3]{x_r};\ pro\ x_r > \varepsilon \tag{1.19}$$

$$f_x = \frac{\kappa \cdot x_r + 16}{116};\ pro\ x_r \le \varepsilon \tag{1.20}$$

$$f_y = \sqrt[3]{y_r};\ pro\ y_r > \varepsilon \tag{1.21}$$

$$f_y = \frac{\kappa \cdot y_r + 16}{116};\ pro\ y_r \le \varepsilon \tag{1.22}$$

$$f_z = \sqrt[3]{z_r};\ pro\ z_r > \varepsilon \tag{1.23}$$

$$f_z = \frac{\kappa \cdot z_r + 16}{116};\ pro\ z_r \le \varepsilon \tag{1.24}$$

Obecně platí pro souřadnice referenční bílé X_r, Y_r a Z_r:

$$x_r = \frac{X}{X_r};\ y_r = \frac{Y}{Y_r};\ z_r = \frac{Z}{Z_r} \tag{1.25}$$

Avšak všimněme si, že hodnoty X, Y, Z výše zmíněného převodu do souřadnic XYZ jsou již vztaženy k referenční bílé. Proto platí:

$$x_r = X;\ y_r = Y;\ z_r = Z \tag{1.26}$$

Konstanty ε a κ jsou určené standardem CIE následovně:

$$\varepsilon = \frac{216}{24389} = 0,008856 \tag{1.27}$$

$$\kappa = \frac{24389}{27} = 903,3 \tag{1.28}$$

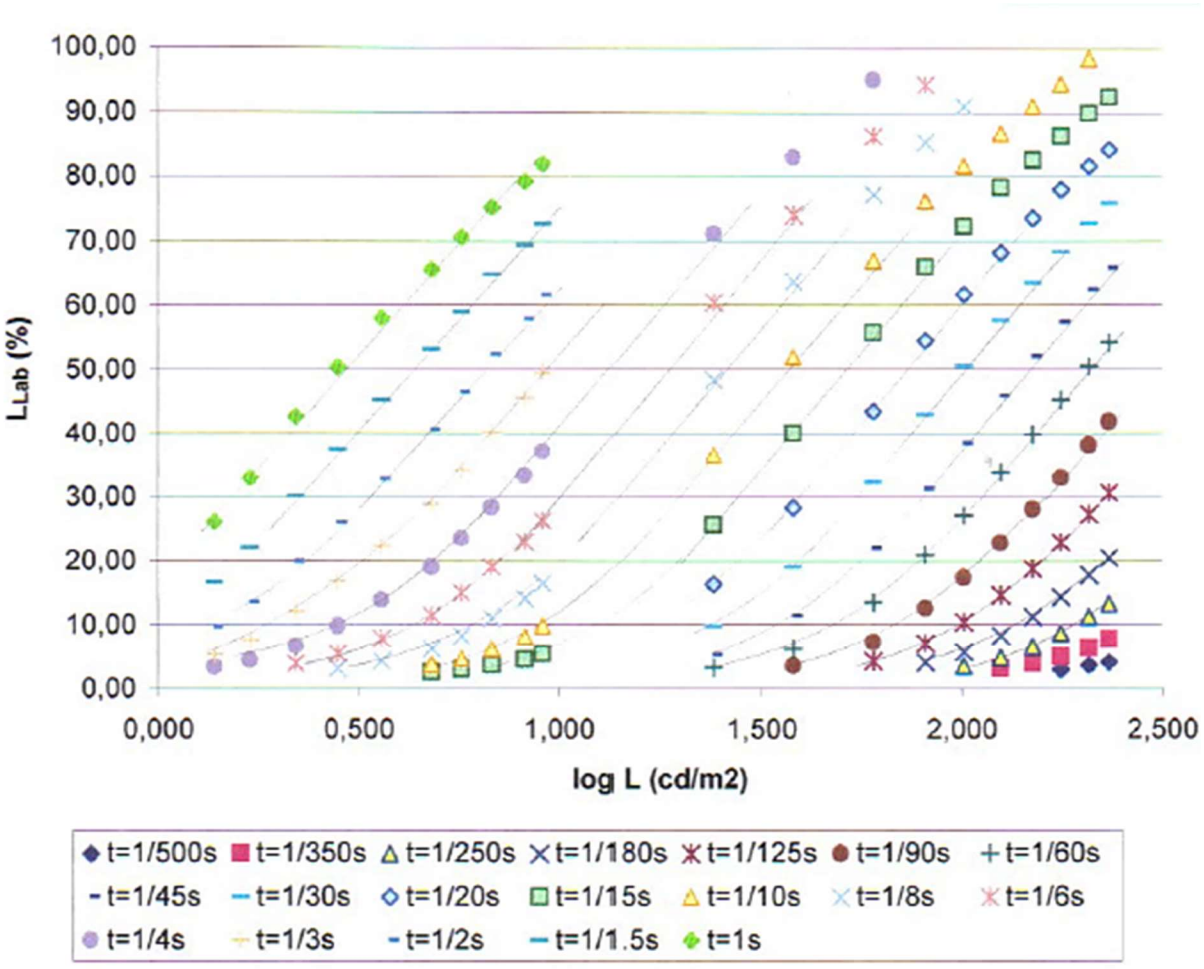
Z výše uvedených vztahů také vyplývá, že souřadnice L_{Lab} vyjadřuje pouze informaci o jasu a není nijak závislá na barvě měřeného bodu. [1]

1.4.4 Určení hodnoty jasu L z hodnoty L_{Lab}

Závislost jasu na hodnotě souřadnice L_{Lab} je dána převodními charakteristikami, které byly v [1] určeny experimentálně. Princip měření byl založen na měření vzorků o různých hodnotách jasu, které byly nasnímány digitálním fotoaparátem pro různá nastavení clon a časů expozice. Zároveň byly jasoměrem změřeny referenční hodnoty skutečných jasů na daných vzorcích.

Měření převodních charakteristik bylo rozděleno na dvě části, důvodem k tomu je obtížnost zajištění potřebně široké škály jasů na scéně. V první části se určovaly převodní charakteristiky pro „relativně malé jasy“, které jsou řádově jednotky až stovky $cd.m^{-2}$ a v druhé „relativně velké jasy“, řádově od stovek do desetitisíců $cd.m^{-2}$.

Po zpracování digitálních fotografií na hodnoty L_{Lab} dle výše uvedených vzorců vznikl soubor hodnot skutečných jasů a jim odpovídajících hodnot L_{Lab}. Z tohoto souboru byly sestaveny převodní charakteristiky.



Obr. 9: Graf naměřených hodnot L_{Lab}. pro clonové číslo 4 [1]

Z těchto charakteristik je viditelné, že křivky jsou lineární pouze v rozsahu hodnot $L_{lab} \in < 20\,;\, 80 >$. Proto se při odvozování vztahů pracuje jen s těmito hodnotami. V době publikování této práce pracujeme s následujícím vztahem, který je uveden v [12], a který je z těchto charakteristik odvozen:

$$L = \frac{F^2}{t} \cdot 0,0373 \cdot e^{0,0307 \cdot L_{Lab}} (cd \cdot m^{-2}) \tag{1.29}$$

kde:

- $F\,(-)$ je clonové číslo
- $t\,(s)$ je čas expozice
- $L_{lab} \in < 20\,;\, 80 >$ je souřadnice v barevném systému *L*a*b*

Je však nutno dodat, že koeficienty v rovnici se mohou při kalibraci jiného fotoaparátu měnit, proto jsou tyto hodnoty ve vytvářené aplikaci nastavitelné, rovnici lze poté uvést obecně s koeficienty *A* a *B*:

$$L = \frac{F^2}{t} \cdot A \cdot e^{B \cdot L_{Lab}} (cd \cdot m^{-2}) \tag{1.30}$$

2 Softwarové možnosti rozšíření dosavadní metody na HDR sadu digitálních fotografií

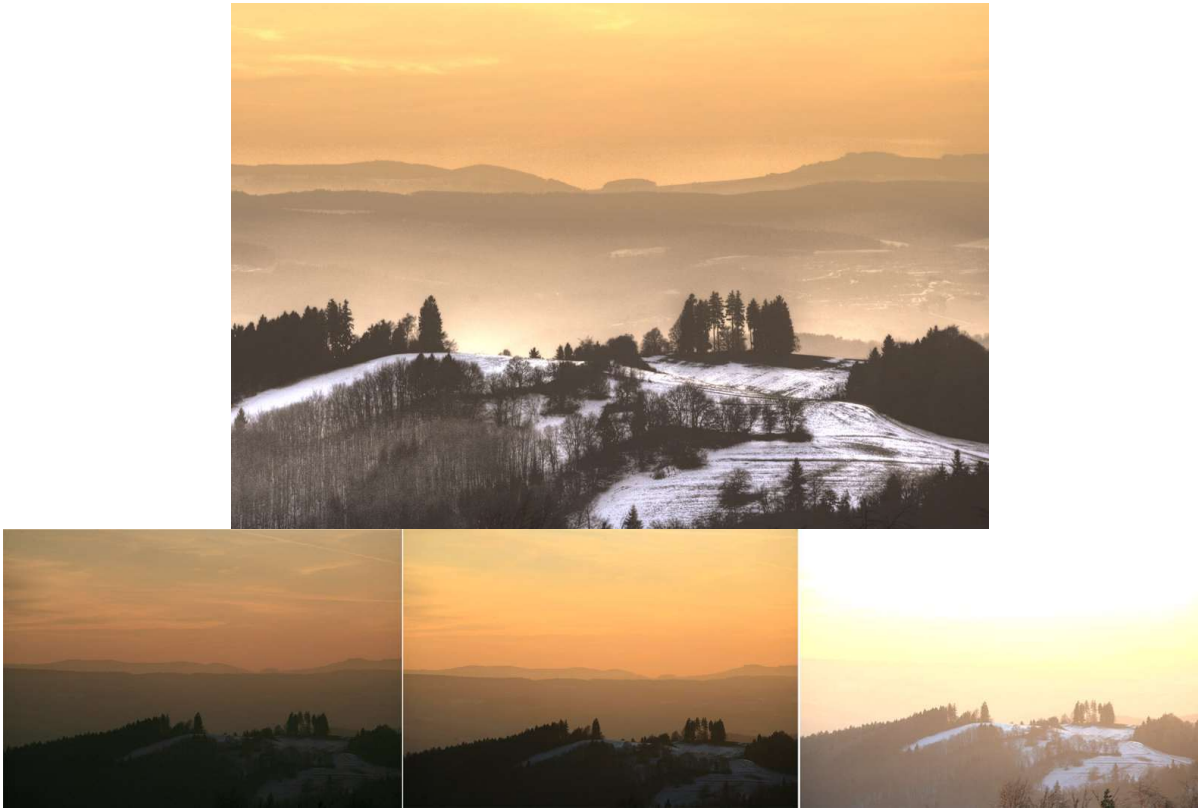
2.1 HDR fotografie obecně

V předchozí kapitole jsme se již zmínili o tématu přeexponované a podexponované fotografie. Tento problém řeší tzv. High Dynamic Range (HDR) fotografie, tedy fotografie s velkým dynamickým rozsahem. Jak již bylo zmíněno, dynamický rozsah fotoaparátu v tomto smyslu znamená poměr mezi nejtmavším a nejsvětlejším místem, které lze rozeznat na fotografii.

2.2 Metodika vytváření HDR fotografie

Základním principem vytváření HDR fotografie je pořízení několika (většinou 3 a více) různě exponovaných fotografií, které poté zkombinujeme pomocí k tomu určeného softwaru (např. Adobe Photoshop, Zoner, Photomatrix), do jedné HDR fotografie. V dnešní době již výrobci fotoaparátů a mobilních telefonů tuto funkcionalitu zabudovávají do samotných přístrojů, kdy se přístroj samotný postará o vytvoření HDR fotografie z několika snímků. [13]

Ideálním stavem je, když jsou snímky pořízeny ze stativu, aby byla na každé fotografii zachycena identická scéna. Pokud se však toto nepodaří, dnešní softwarové nástroje umí fotografie zarovnat a tento problém eliminovat.



Obr. 10 HDR fotografie složená ze 3 snímků [13]

Dále je třeba se vyhýbat jakémukoliv pohybu mezi jednotlivými snímky, např. pohybu lidí. Na výsledné kombinaci fotografií by potom byli vidět „duchové“, tedy části snímku, které se již na ostatních snímcích nevyskytují, nebo se vyskytují na jiné pozici.

Expozici lze měnit jak změnou doby expozice, tak i změnou clonového čísla. V HDR fotografii se však používá jen změna času expozice, protože se změnou clony se mění i zaostření fotografie a jednotlivé snímky by byly příliš odlišné.

2.3 Využití HDR fotografie pro měření jasu

Jedním z cílů této práce je využít HDR metodiku při výpočtu jasů z digitální fotografie. Jak bylo popsáno v kapitole 1.4, jsme při výpočtu jasu omezeni hodnotami $80 > L_{Lab} < 20$, hodnoty mimo tyto hranice odpovídají přeexponování a podexponování snímku. Pokud vezmeme v potaz tuto analogii mezi fotografií a měřením jasu, lze vyvodit závěr, že použitím HDR fotografie jsme schopni zkombinováním více fotografií při různých expozicích zvýšit jasový rozsah fotoaparátu při měření jasů, stejně jako zvyšujeme dynamický rozsah fotografie.

2.3.1 Postup vytvoření matice jasů

Digitální fotografii si můžeme představit také jako matici pixelů, kde je každý pixel reprezentován svými barevnými složkami R, G a B.

$$\begin{pmatrix} \begin{pmatrix} R_{11} \\ G_{11} \\ B_{11} \end{pmatrix} & \cdots & \begin{pmatrix} R_{1n} \\ G_{1n} \\ B_{1n} \end{pmatrix} \\ \vdots & \ddots & \vdots \\ \begin{pmatrix} R_{m1} \\ G_{m1} \\ B_{m1} \end{pmatrix} & \cdots & \begin{pmatrix} R_{mn} \\ G_{mn} \\ B_{mn} \end{pmatrix} \end{pmatrix}$$

Při převodu každého pixelu na odpovídající hodnotu jasu nejprve musíme zjistit hodnotu souřadnice L_{Lab} dle kap. 1.4. Tímto dostaneme matici hodnot L_{Lab}, kde každá hodnota přísluší odpovídajícímu pixelu na dané pozici.

$$\begin{pmatrix} L_{Lab11} & \cdots & L_{Lab1n} \\ \vdots & \ddots & \vdots \\ L_{Labm1} & \cdots & L_{Labmn} \end{pmatrix}$$

Pomocí empirického vztahu (1.31) lze poté vytvořit matici jasů výpočtem jasu pro každý pixel. Je nutné dbát na hranice dynamického rozsahu, za kterými již spočtené hodnoty jasu nechceme považovat za platné. Jsou to hodnoty $L_{Lab} < 20$, za ně do matice jasů doplníme hodnotu 0. Stejně tak v případě, kdy je $L_{Lab} > 80$, v tom případě doplníme jako hodnotu jasu 9999.

Tato konvence nám umožňuje jasně identifikovat neplatné hodnoty a také nám zkracuje dobu výpočtu celé matice, jelikož pro daný pixel výpočet provádět nemusíme a pouze dosadíme číselnou hodnotu 0 nebo 9999.

$$\begin{pmatrix} L_{11} & \cdots & L_{1n} \\ \vdots & \ddots & \vdots \\ L_{m1} & \cdots & L_{mn} \end{pmatrix}$$

Tento postup opakujeme pro každý snímek z množiny, ze které se má vytvořit HDR fotografie, respektive HDR matice jasů. Výsledkem je, že máme pro každý snímek vytvořenou matici jasů. Nyní je třeba matice sloučit do jedné.

2.3.2 Algoritmus kombinování jasových matic

Základní myšlenka pro zkombinování více jasových matic je nahrazení neplatných hodnot 0 a 9999, hodnotami platnými, pokud existují v jiné matici na stejném indexu. Jednoduchá implementace v pseudokódu je znázorněna níže.

```
//mejme 3 stejne velke matice jasu
List jasoveMatice = {matice1, matice2, matice3};
//zvolme si referencni matici, do ktere budeme doplnovat platne hodnoty
referencniMatice = matice1;
//iterujme pres referencni matici
for(x = 0; x < sirkaMatice; x++){
    for(y = 0; y < vyskaMatice; y++){
        //pokud je jas na souradnici x,y neplatnou hodnotou -> hledej platnou hodnotu
        //v ostatnich dvou maticich
        if(!jePlatnaHodnota(referencniMatice[x][y])) {
            //iterujme pres list matic, referencni matici preskakujeme
            for(i = 1; i < pocetMatic; i++){
                //pokud je v aktualni matici na stejne pozici platna hodnota, vymen ji
                //za neplatnou v referencni matici
                aktualniMatice = jasoveMatice(i);
                aktualniJas = aktualniMatice[x][y];
                if(jePlatnaHodnota(aktualniJas){
                    referencniMatice[x][y] = aktualniJas;
                }
            }
        }
    }
}
```

Fragment kódu 1: Algoritmus 1

Tento algoritmus je však trochu zjednodušený a trpí určitými nedostatky. Jedním z nich je to, že při hledání vhodné hodnoty jasu na náhradu neplatné hodnoty, při každém nalezení platné hodnoty dojde k výměně. To znamená, že při nalezení neplatné hodnoty v referenční matici na pozici x, y a přítomnosti platné hodnoty v maticích 2 a 3 dojde nejprve k nahrazení neplatné hodnoty hodnotou jasu z matice dva a poté se ta samá hodnota vymění za platnou hodnotu z třetí matice.

Toto lze vyřešit tak, že budeme hodnoty, které by mohly potenciálně nahradit neplatnou hodnotu ve vztažné matici ukládat do pole a poté z těchto hodnot vybereme tu nejvhodnější. Nejvhodnější hodnotou jasu myslíme hodnotu L_{Lab}, nikoli jasu, která je co nejblíže 50, jelikož v této oblasti závislosti dle obr. 9 nejvíce odpovídají přímce a budou tedy i více odpovídat realitě.

Bude tedy výhodnější pracovat zpočátku s maticemi L_{Lab} hodnot, sestavit optimalizovanou matici a až poté vypočítat jasovou matici. Nutno dodat, že při výpočtu jasu je nutné znát údaje o expozici snímku, tudíž je nutné udržet pořádek v tom, ke které expozici patří daná hodnota L_{Lab}.

```
//mejme 3 stejne velke matice Llab
List LlabMatice = {matice1, matice2, matice3};
//zvolme si referencni matici, do ktere budeme doplnovat platne hodnoty
referencniMatice = matice1;
//iterujme pres referencni matici
for(x = 0; x < sirkaMatice; x++){
    for(y = 0; y < vyskaMatice; y++){
        //pokud je jas na souradnici x,y neplatnou hodnotou -> hledej platnou hodnotu
        //v ostatnich dvou maticich
        if(!jePlatnaHodnota(referencniMatice[x][y])) {
            //iterujme pres list matic, referencni matici preskakujeme
            //definujme list hodnot L_lab, do ktereho budeme ukladat potencialni nahrady
            List nahrady;
            for(i = 1; i < pocetMatic; i++){
                //pokud je v aktualni matici na stejne pozici platna hodnota,
                //pridej ji do pole potencialnich nahrad
                aktualniMatice = LlabMatice(i);
                aktualniLlab = aktualniMatice[x][y];
                if(jePlatnaHodnota(aktualniLlab)) {
                    nahrady.add(aktualniMatice[x][y]);
                }
            }
            //pokud list s potencialnimi nahradami neni prazdny, vyber z nej
            //tu nejvhodnejsi a nahrad s ni neplatnou hodnotu
            if(!nahrady.isEmpty()){
                referencniMatice[x][y] = vyberNejvhodnejsi(nahrady);
            }
        }
    }
}
```

Fragment kódu 2: Algoritmus 2

Představme si však situaci, kdy máme více fotografií, každou o různé expozici – hodnoty L_{Lab} se v této situaci mezi jednotlivými snímky dost mění. Můžeme tedy mít několik snímků, kde na stejném indexu budou platné hodnoty L_{Lab}, např. 40 v prvním snímku, 50 v druhém a 60 ve třetím. Která z těchto hodnot bude za použití předchozího postupu ve finálním snímku? Bude v něm ta ze snímku, který bereme jako referenční a do něhož doplňujeme hodnoty z ostatních snímků pouze tam, kde najdeme pixel s hodnotou L_{Lab} mimo námi nastavené hranice. V této situaci tedy vidíme platnou hodnotu, kterou již zdánlivě není potřeba nahrazovat. Nahradit ji však může být žádoucí, pokud se v ostatních snímcích nachází hodnota L_{Lab} bližší 50.

Algoritmus pro implementování této myšlenky je znázorněn níže. Pro každý pixel z referenční matice hodnot L_{Lab} se snažíme najít hodnotu bližší k 50, nežli je ta v referenční matici. V případě, že takovou hodnotu najdeme, doplníme ji do referenční matice, v případě, že takovou hodnotu nenajdeme, v referenční matici zůstává původní hodnota.

```
//mejme 3 stejne velke matice Llab
List LlabMatice = {matice1, matice2, matice3};
//zvolme si referencni matici, do ktere budeme doplnovat platne hodnoty
referencniMatice = matice1;
//iterujme pres referencni matici
for(x = 0; x < sirkaMatice; x++){
        for(y = 0; y < vyskaMatice; y++){
           //pro kazdou hodnotu hledame v ostatnich maticich hodnotu blizsi 50
           //definujme list hodnot Llab, do ktereho budeme ukladat potencialni nahrady
           List nahrady;
           for(i = 1; i < pocetMatic; i++){
                 //pokud je v aktualni matici na stejne pozici platna hodnota,
                 //pridej ji do pole potencialnich nahrad
                 aktualniMatice = LlabMatice(i);
                 aktualniLlab = aktualniMatice[x][y];
                 if(jePlatnaHodnota(aktualniLlab)) {
                     nahrady.add(aktualniMatice[x][y]);
                 }
           }
                //pokud list s potencialnimi nahradami neni prazdny, vyber z nej
                //tu nejvhodnejsi a nahrad s ni stavajici hodnotu
                if(!nahrady.isEmpty()){
                    referencniMatice[x][y] = vyberNejvhodnejsi(nahrady);
                }
            }
        }
    }
```

Fragment kódu 3: Algoritmus 3

Tato varianta je samozřejmě výpočetně mnohem náročnější než algoritmus předchozí, který hledá náhradu pouze pro neplatné hodnoty. Můžeme ale dosáhnout mnohem větší přesnosti měření jasu.

3 Vytvoření a verifikace aplikace pro HDR analýzu jasů sady digitálních fotografií

Hlavním cílem této práce je vytvoření softwaru pro výpočty jasu z fotografií. Výpočet byl dosud na katedře elektroenergetiky prováděn ve Wolfram Mathematica, toto řešení je však pomalé, ne příliš uživatelsky přístupné a často se potýkalo s problémy zpětné kompatibility při přechodu na novější verzi tohoto softwaru.

Od nové aplikace se očekává, že bude výpočty provádět v mnohem rychlejším čase. Velkým neduhem řešení ve Wolfram Mathematica je absence grafického uživatelského rozhraní. Navržená aplikace bude včetně uživatelského rozhraní, které si klade za cíl umožnit jednoduché a precizní odečítání naměřených hodnot. Další funkcionalita bude umožňovat nastavování parametrů výpočtu, jako jsou například koeficienty v rovnici (1.32).

Je důležité podotknout, že ač je práce psána v češtině, veškerý kód je psán v angličtině, jak je při psaní softwaru zvykem. Jako kompromis budou ukázky kódu vložené do textu okomentovány česky. Stejně tak byla angličtina použita jako jazyk grafického rozhraní.

3.1 Návrh softwarového řešení

Jednou z počátečních otázek při návrhu řešení bylo, zda se bude jednat o desktopovou nebo webovou aplikaci. U webové aplikace se nabízely dvě možnosti – provádět výpočty na straně klienta anebo na serveru, s tím, že by muselo velké množství dat (fotografií) putovat na server pro zpracování a poté také zpět na stranu klienta pro zobrazení, což je hlavní důvod, proč se toto řešení ukázalo jako nejméně schůdné.

Webová aplikace s veškerými výpočty prováděnými na straně klienta již v dnešní době není nic neobvyklého. Nebyl by problém ani s infrastrukturou pro zpracování fotografií, v ekosystému jazyka JavaScript existuje několik knihoven, které tuto problematiku řeší. Výhodami tohoto řešení by byla poněkud flexibilnější tvorba uživatelského rozhraní, a hlavně také možnost užívání aplikace bez jakékoliv instalace pouhým načtením webové stránky v prohlížeči. Nevýhodou však je starost se zajištěním domény a hostingu a také samozřejmě nutnost připojení k internetu. Nutnost připojení k internetu by byla problémem v případě, kdybychom chtěli výpočty provádět v terénu ihned po pořízení fotografií. Toto jsou hlavní důvody, proč byla nakonec vybrána poslední možnost, a to tedy cesta tvorbou desktopové aplikace.

Desktopová aplikace, tedy program, který běží na počítači a startuje z pevného disku, v tomto případě nepotřebuje připojení k internetu, což je nespornou výhodou v naší situaci z výše uvedeného důvodu. Mírnou nevýhodou je nutnost přítomnosti veškerých souborů, které jsou potřeba k běhu programu, na pevném disku. Ohledně rychlosti výpočtu se jedná o velmi dobré řešení, program lze optimalizovat tak, aby dnešní procesory využívaly všechna svá vlákna a výpočty potom trvají mnohem kratší dobu. Není pravdou, že by vícevláknové řešení nešlo realizovat i v klientské aplikaci, která běží v prohlížeči, avšak u desktopových aplikací je s tímto mnohem delší zkušenost a lepší dostupnost k tomu vhodných technologií.

3.1.1 Výběr technologií

V dnešní době je k dispozici nespočet možností pro tvorbu desktopových aplikací, jednak z hlediska programovacích jazyků a také z hlediska knihoven pro tvorbu grafických uživatelských rozhraní (dále jen GUI – *graphical user interface*), kterých pro každý jazyk existuje několik. Mezi

dnes nejrozšířenější programovací jazyky pro tvorbu desktopových aplikací patří C#, C++, Java nebo i Python.

Vybrán byl nakonec jazyk Java, jehož velkou výhodou je platformní nezávislost. Program psaný v Javě běží na jakémkoliv operačním systému díky tomu, že se kompiluje do bytekódu, který poté běží na JVM (Java Virtual Machine). JVM je virtuální stroj, který byl vyvinut pro mnoho platforem (např. Windows, Linux). Ačkoli Java nemusí být nutně z výše zmiňovaných jazyků tím nejrychlejším (tento přívlastek se často spojuje s C++), pro naše řešení bude postačující. Verze Javy použitá v naší aplikaci je Java 9, která na rozdíl od předchozí verze 8 podporuje formát fotografií TIFF, do té doby bylo pro podporu tohoto formátu nutno využívat externích knihoven.

Pro jazyk Java existuje několik standardních knihoven pro tvorbu GUI, nejpopulárnější z nich jsou Swing a JavaFX. Swing je již poměrně stará technologie, jež není již vyvíjena a podporována a JavaFX je jejím přímým nástupcem. Pro naši potřebu byla proto zvolena knihovna JavaFX.

Grafické rozhraní však netvoří celou aplikaci, slouží nám pouze k uživatelské interakci. Ještě je potřeba provádět výpočty, v našem případě pro jasovou analýzu fotografií. Pro zpracování fotografií má Java také několik standardních knihoven, které umožňují čtení fotografií, operace s barevnými pixely, převody mezi různými barevnými systémy a další operace převáděné při práci s obrázky.

3.1.2 Knihovna JavaFX

Jak již bylo zmíněno, JavaFX je platforma oficiálně podporovaná společností Oracle, která se stará o vývoj Javy jako takové. Jejím účelem je rychlá a snadná tvorba uživatelských rozhraní, ať již pro vývoj desktopových, či webových aplikací. Poskytuje veškeré standardní komponenty, které patří do uživatelských rozhraní. Obecná struktura JavaFX aplikace je na obr. 11.

Obecně JavaFX aplikace obsahuje jedno nebo více oken (stages), každé okno má přiřazenou scénu (Scene). Každá scéna může být tvořena stromem propojených ovládacích prvků (Scene Graphs), rozložení a jiných komponent.

Stage

Stage (pódium) je prakticky vnějším oknem celé aplikace. Desktopová aplikace může mít několik oken otevřených zároveň, např. pro dialogy či různá upozornění. Každé okno je v aplikaci reprezentováno Java objektem Stage.

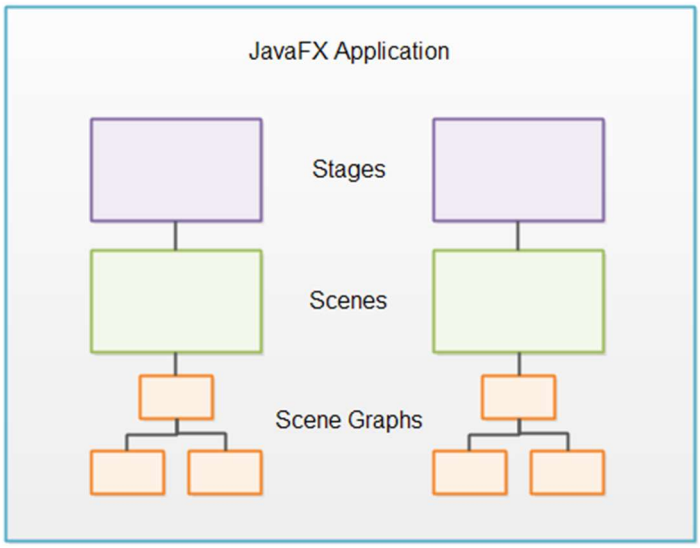
Scene

Abychom mohli cokoliv v okně zobrazit, potřebujeme k tomu scénu. Stage dokáže zobrazit v každém okamžiku jen jednu scénu, ale je možné scény měnit za běhu programu. Doslovným překladem těchto dvou prvků dostaneme analogii pódia a scény v divadle. Pódium v divadle lze též přearanžovat mnohokrát během představení a vždy se jedná o jinou scénu.

Scéna je v programu reprezentována objektem Scene.

Scene graph

Všechny vizuální komponenty (ovládací prvky, rozložení atd.) musí být pro zobrazení připojeny ke scéně a tato scéna musí být připojena k pódiu, jen tak bude možné celou scénu vidět. Celý tento strom komponent připojených ke scéně se nazývá Scene Graph.



Obr. 11: Struktura JavaFX aplikace [14]

Node

Prvek Node označuje jednotlivé uzly ve stromu. Všechny uzly dědí od JavaFX třídy *javafx.scene.Node*. Existují dva typy uzlů v stromu – *větve* a *listy*. Větev je uzel, který obsahuje další potomky, kdežto list již žádné potomky nemá.

Ovládací prvky

JavaFX ovládací prvky jsou běžné komponenty uživatelského rozhraní, které nám poskytují jakékoliv možnosti ovládání naší aplikace. Například se jedná o tlačítka, textová pole, nadpisy atd.

Aby byl ovládací prvek aktivní, musí být připojen ke stromu objektu Scene. Jsou většinou zahrnuty v nějaké komponentě rozložení, která se stará o vzájemné rozpoložení jednotlivých ovládacích prvků.

V této práci jsou použity následující ovládací prvky:

- Button
- CheckBox
- Label
- Menu
- MenuBar
- TextField

Rozložení

Rozložení (*Layouts*) jsou komponenty, které do sebe zabalují jiné komponenty a určují jejich uspořádání. Také se někdy označují jako rodičovské komponenty (*parent components*), jelikož obsahují potomky, a také proto, že všechny Java třídy reprezentující rozložení dědí z třídy *javafx.scene.Parent*. Příkladem je komponenta HBox, kde jsou jednotliví potomci uspořádáni horizontálně.

Stejně jako ovládací prvky musí i rozložení být připojeny ke stromu scény jakéhokoliv objektu Scene.

V této práci jsou použity tyto prvky rozložení:

- Group
- HBox
- VBox
- ScrollPane
- GridPane

Komponenty rozložení lze také vnořovat do sebe. To se může ukázat jako užitečné např. pro vytvoření horizontálních řádků komponent, které nejsou uspořádány v mřížovitě a kde každý řádek má jiné parametry. K tomuto lze využít několik HBox komponent uvnitř VBox komponenty. [14]

3.1.3 Knihovny používané pro výpočty

K provádění výpočtů nám stačí velmi málo, co se knihoven týče. Dá se říct, že bychom si v případě nutnosti vystačili i s úplně elementárními prvky, které existují prakticky v každém programovacím jazyce, jako jsou datové typy, řetězce vícedimenzionální pole, metody, smyčky, podmínky atd. Existuje však několik standardních Java knihoven, které nám značně usnadní práci.

Java.IO

Jak již vyplývá z názvu, java.io je knihovna zabývající se vstupem a výstupem dat, čtením a zapisováním dat z datových proudů (*Stream*) a také zpracováním souborů. Pro naši potřebu je důležitá druhá poslední funkcionalita. Abychom vůbec mohli v aplikaci nahrát fotografii do proměnné a potom s ní pracovat, musíme k tomu využít této knihovny, konkrétně třídy *File*. Při nahrávání fotografií jednoduše vytvoříme objekt typu *File*, který bude v programu představovat daný soubor (fotografii). V této práci využíváme k vytvoření objektu *File* JavaFX třídy *FileChooser*, která umožňuje otevřít klasický dialog pro nahrání souborů.

```
File file = fileChooser.showOpenDialog(primaryStage);
```

Abstract Window Toolkit (AWT)

Java AWT je knihovna která původně sloužila k tvorbě uživatelských rozhraní, než ji nahradil Swing a poté JavaFX. V této knihovně se však nachází několik tříd, které jsou i dnes stále užitečné při práci s grafikou a fotografiemi. Velmi stěžejní je pro nás třída *BufferedImage*, která dědí od abstraktní třídy *Image*. Tato třída představuje samotný obrázek a data o něm. Obrázek je zde reprezentován jeho barevným modelem a rastrem obrazových dat.

My tuto třídu využíváme tak, že za pomocí třídy *javax.imageio.ImageIO* a její statické metody *read* přečteme výše zmiňovaný objekt typu *File* a tato metoda nám vrátí objekt typu *BufferedImage*.

```
BufferedImage image = ImageIO.read(file);
```

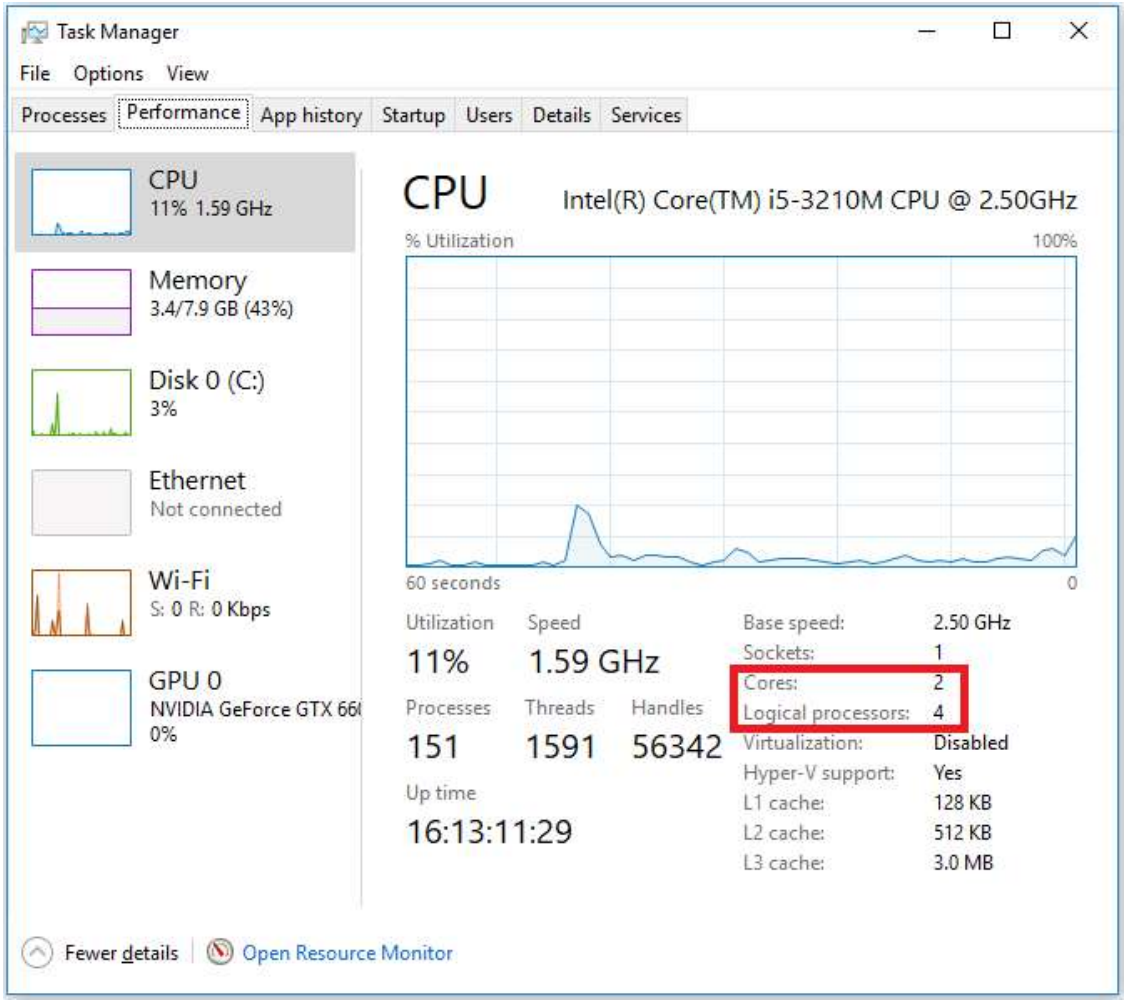
BufferedImage má několik metod, které nám velmi usnadňují manipulaci s fotografiemi. Jednou z nich je instanční metoda *getRGB(int x, int y)*, která nám vrací RGB data o pixelu v sRGB barevném systému. Stejně tak existuje i metoda *setRGB(int x, int y, int rgb)*, jež nám umožňuje přepsat libovolný pixel jiným o udané barvě. Nutno podotknout, že RGB informace je zde ve formátu jednoho celého čísla, pro získání jednotlivých barevných složek je nutné jej rozdělit. V hexadecimálním tvaru je toto číslo ve formátu *RRGGBB*, tedy prvních 8 bitů je hodnota červené složky, dalších 8 bitů je hodnota zelené složky a posledních 8 bitů tvoří modrou složku. Každá složka je hodnota mezi 0 a 255.

Java Concurrency

V dnešním počítači má každý procesor dvě a více fyzických jader a jeví se proto operačnímu systému jako dva samostatné procesory. To umožňuje procesoru vykonávat více procesů najednou, čehož výsledkem je větší výpočetní výkon. Procesory často mají i funkci zvanou *Hyper-Threading*, která způsobuje, že se procesor tváří, jako by měl dvojnásobný počet jader. Dvoujádrový procesor se

potom operačnímu systému prezentuje jako čtyřjádrový – má 4 virtuální jádra (logické procesory). Tento procesor má stále jen 2 CPU, avšak využívá vnitřní logiky k zrychlení běhu programu.

Aby byl náš program schopen využít co nejvíce jader procesoru, popř. všech virtuálních jader, musíme k tomu v případě Javy využít k tomu připravenou knihovnu zvanou *Concurrency*. Základním pojmem v této knihovně je vlákno neboli *thread*. Každé vlákno je spojeno s instancí objektu *Thread*. Pokud využíváme více vláken, jedná se o tzv. *multithreading*. Úkolem operačního systému je potom co nejefektivněji rozdělit existující vlákna či procesy mezi jednotlivá jádra procesoru.



Obr. 12: Počet fyzických a počet virtuálních jader procesoru

Výpočty, které má za úkol provádět námi vytvářený program jsou z velké části založené na čtení pixelu z existující fotografie, provedení výpočtů s daným pixelem a uložení výsledku matice. V tomto případě lze jednoduše zpracovávanou fotografii rozdělit na segmenty – tak, aby každé vlákno zpracovávalo svůj samostatný segment. Tomuto postupu se také někdy říká „rozděl a panuj“. Výsledky výpočtů se potom budou zapisovat do výsledné matice, ke které mají všechna vlákna přístup.

Častým problémem při *multithreadingu* jsou konflikty mezi jednotlivými vlákny, tzv. *thread interference*. Když například dvě vlákna pracují nad sdílenou proměnnou, může se stát, že první vlákno si přečte hodnotu dané proměnné, provede změny a uloží je do původní proměnné, zatímco druhé vlákno si přečetlo onu proměnnou ve stejném okamžiku jako první vlákno. Druhé vlákno

provede změny a také je uloží do původní proměnné. První vlákno provedlo výpočet rychleji a změny uložilo do proměnné dříve. Druhé vlákno poté svými změnami přepsalo výsledek výpočtu prvního vlákna, které o svoje změny tedy přišlo. Tyto problémy se v *Java Concurrency* řeší tzv. synchronizací, kterou dosáhneme toho, že ke sdílenému objektu přistupuje vždy jen jedno vlákno, a to druhé čeká.

Těmto problémům se však v našem případě můžeme snadno vyhnout. Zdrojová fotografie, se kterou všechna vlákna pracují, je pouze ke čtení, a výsledky výpočtů se zapisují do separátní matice. Každé vlákno má svůj přidělený segment zdrojové fotografie a také segment matice výsledků. Tím pádem se při správném postupu nestane, že by vlákna zapisovala výsledky do segmentu, který patří jinému vláknu.

K implementaci výše zmíněnému postupu „rozděl a panuj“ existuje v *Java Concurrency* abstraktní třída *RecursiveTask*. Princip použití této třídy je na následujícím fragmentu pseudokódu.

```
if (ukol je dostatecne maly)
        proved ho klasickym zpusobem
else
    rozdel ukol na nekolik mensich casti
    proved vsechny dilci casti a vrat vysledek
```

Fragment kódu 4: Rozdělení práce s třídou RecursiveTask

Pro vykonání všech dílčích úkolů se zavolá metoda *invoke()* třídy *ForkJoinPool* a jako parametr se jí předá námi požadovaný úkol, což je třída, která je potomkem zmíněné třídy *RecursiveTask*. *ForkJoinPool* představuje jakýsi „fond vláken“, anglicky *thread pool*. Když vytvoříme instanci této třídy, získáme vlastně několik aktivních vláken, které si mezi sebou rozdělí práci, kterou jsme předali metodě *invoke()*. Tato třída počká, než každé vlákno dokončí svůj úkol a poté nám vrátí výsledek.

3.2 Realizace výpočetní části

Úkolem výpočetní části programu je vzít nahrané fotografie a provést převod každého pixelu na hodnotu jasu. Tyto vypočtené hodnoty jsou poté posílány do uživatelského rozhraní pro jejich zobrazení.

Všechny důležité metody pro výpočty jsou pro jednoduchost agregovány do jedné třídy, která slouží pouze jako procesor výpočtů a neuchovává si v sobě žádný stav. Tato třída byla proto nazvána *ImageProcessor*. Postupně zde nyní budou uvedeny všechny důležité metody z této třídy, tyto metody vesměs jen aplikují vzorce z kapitoly 1.4.

3.2.1 Převod pixelu ze soustavy *RGB* do *L*a*b*

```
//prevadi RGB souradnice v rozmezi <0,255> na XYZ souradnice v rozmezi <0,1>
public double rgbToXyz(int R, int G, int B) {

    //normalizuj RGB hodnoty <0,255> na hodnoty <0,1>
    double r = R / 255.f;
    double g = G / 255.f;
    double b = B / 255.f;
    /*aplikuj sRGB -> XYZ prevodni rovnice
      X, Y a Z jsou vztazeny k standardnimu iluminantu D65/2° */
    if (r <= 0.04045) {
        r = r / 12.92;
    } else {
        r = Math.pow(((r + 0.055) / 1.055), 2.4);
    }
    if (g <= 0.04045) {
        g = g / 12.92;
    } else {
        g = Math.pow(((g + 0.055) / 1.055), 2.4);
    }
    if (b <= 0.04045) {
        b = b / 12.92;
    } else {
        b = Math.pow(((b + 0.055) / 1.055), 2.4);
    }

    double resultY;
    resultY = r * 0.2126 + g * 0.7152 + b * 0.0722;
    return resultY;
}
```

Fragment kódu 5: Metoda rgbToXyz

Výše uvedená metoda má za úkol převést barevný pixel z barevného systému RGB do systému XYZ, který je nutným mezikrokem při převodu do systému Lab. Do argumentu metody jsou předávány tři hodnoty jednotlivých barevných složek. Tyto hodnoty jsou v rozmezí od 0 do 255, tedy 8 bitů. Proto je třeba je nejprve převést na normalizovaný rozsah 0 až 1. Po normalizaci lze již aplikovat známé vzorce (1.33) až (1.34). Všimněme si, že metoda neprovádí výpočet všech složek X, Y a Z, nýbrž počítá jen složku Y. To je z toho důvodu, že naším cílem je získat souřadnici L ze systému Lab a abychom se k ní dostali, stačí nám složka Y. Výpočet ostatních složek by se prováděl navíc a zbytečně by zpomaloval chod programu.

```
//konstanty
private static double eps = 0.008856;
private static double k = 903.3;

//prevadi Y ze systemu XYZ na L v systemu Lab
public double xyzToLab(double resultY) {

    double fy;

    //vypocti souradnici L
    if (resultY > eps) {
        fy = Math.pow(resultY, 1.0 / 3.0);
    } else {
        fy = (k * resultY + 16.0) / 116.0;
    }

    double result;

    result = (116.0 * fy) - 16.0;

    return result;

}
```

Fragment kódu 6: Metoda xyzToLab

Druhá metoda sloužící k převodu pixelu na hodnotu L_{Lab} má na starost převzít výsledek výpočtu předchozí metody *rgbToXyz* a vyvodit z ní potřebnou hodnotu L_{Lab}, která se pohybuje mezi 0 a 100. Opět zde vidíme aplikované známé vzorce (1.35), (1.36) a (1.37). V ukázce kódu je ještě vidět definice konstant, které jsou ve skutečnosti deklarovány již na začátku třídy. Tyto konstanty představují ε a κ zmiňované v rovnicích (1.38) a (1.39).

K úplnosti by ještě bylo dobré tyto dvě metody spojit, jelikož jsou vždy používány obě najednou. K tomu slouží následující metoda.

```
//prevede pixel typu int na hodnotu Llab <0,100> typu double
public double convertPixelRgbToLab(int pixel) {
    Color color = new Color(pixel);
    return xyzToLab(rgbToXyz(color.getRed(), color.getGreen(), color.getBlue()));

}
```

Fragment kódu 7: Metoda convertPixelRgbToLab

Tato metoda také akceptuje jako parametr pixel typu *int*, kde jsou všechny barevné složky zakódované do 24 bitů tohoto čísla. Toto je z důvodu přímé návaznosti na metodu *getRGB(int x, int y)* třídy *BufferedImage* z knihovny *AWT*, která vrací data o pixelu právě v tomto formátu. K rozebrání tohoto čísla na jednotlivé složky využíváme třídy *java.awt.Color*, která nám umožňuje pomocí metod *getRed*, *getGreen* a *getBlue* získat barevné složky jako celé číslo v rozmezí 0 a 255.

Jednotlivé složky předáme metodě *rgbToXyz* jako parametry a výsledek této metody předáme metodě *xyzToLab*, jejíž výsledek, tedy hodnotu L_{Lab}, vrátíme.

3.2.2 Sestavení L_{Lab} matice a matice jasů L

Sestavení celé matice L_{Lab} z dané fotografie je vcelku jednoduché, vlastně jen zavoláme výše popsanou metodu na každý jeden pixel fotografie.

```
/* ze zadane fotografie vypocita matici LLab */
public double[][] constructLlabMatrix(BufferedImage img) {

    //uloz rozmery fotografie
    int cols = img.getWidth();
    int rows = img.getHeight();

    //vytvor prazdnou matici o rozmerech fotografie
    double[][] LlabMatrix = new double[rows][cols];

    //napln matici hodnotami Llab
    for (int j = 0; j < rows; j++) {
        for (int i = 0; i < cols; i++) {
            LlabMatrix[j][i] = convertPixelRgbToLab(img.getRGB(i, j));
        }
    }
    return LlabMatrix;
}
```

Fragment kódu 8: Metoda constructLlabMatrix

Metoda *constructLlabMatrix* si bere jako parametr již několikrát zmiňovaný objekt třídy *BufferedImage*, který představuje fotografii. Postup metody je následující:

1. Uložíme rozměry fotografie.
2. Vytvoříme prázdnou matici typu *double* o těchto rozměrech.
3. Iterujeme přes celou fotografii.
4. Každý pixel zkonvertujeme na hodnotu L_{Lab}.
5. Pixel uložíme do matice.

Všimněme si zvláštního indexování u smyček. Konvencí bývá index vnější smyčky značit *i*, a index vnořené smyčky *j*. V našem případě je to obráceně, a to z toho důvodu, že dvojdimenzionální pole má indexování ve smyslu *[řádek][sloupec]*. V případě fotografie je však intuitivnější indexovat pixely dle kartézského systému os *X* a *Y* a tomu odpovídá i volání metody *getRGB(i, j)*, kde je na prvním místě index souřadnice *X*, tedy sloupce *i*, a na druhém místě souřadnice *Y* – řádku *j*.

Nyní, když máme sestavenou matici L_{Lab}, můžeme na jejím základě sestavit matici jasů L.

```
//ze zadane matice LLab, expozice a koeficientu A a B vytvori matici jasu L
public double[][] constructLuminanceMatrix(double[][] LlabMatrix,
                                           double aperture, double exposure,
                                           double Acoeff, double Bcoeff) {
    //uloz rozmery LLab matice
    int rows = LlabMatrix.length;
    int cols = LlabMatrix[0].length;

    //vytvor prazdnou matici jasu o stejnych rozmerech
    double[][] luminanceMatrix = new double[rows][cols];

    for (int j = 0; j < rows; j++) {
        for (int i = 0; i < cols; i++) {

            //Llab hodnoty <= 20 && >= 80 nahrazeny 0 a 9999
            if (LlabMatrix[j][i] <= 20) {
                luminanceMatrix[j][i] = 0;

            } else if (LlabMatrix[j][i] >= 80) {
                luminanceMatrix[j][i] = 9999;

            } else {
                //vypocti jas dle vzorce
                luminanceMatrix[j][i] =
                        (aperture * aperture / exposure) *
                                Acoeff * Math.exp(Bcoeff * LlabMatrix[j][i]);

            }
        }
    }
    return luminanceMatrix;
}
```

Fragment kódu 9: Metoda constructLuminanceMatrix

Tato metoda přebírá jako parametr matici L_{Lab}, clonové číslo F (aperture), čas expozice t v sekundách (exposure) a také koeficienty A a B. Všechny tyto parametry se dosadí do vzorce (1.40) a z něj dostaneme hodnotu jasu. Tato metoda se také stará o validaci hodnot L_{Lab}, pokud je hodnota $\leq$ 20 nebo $\geq$ 80, nastaví hodnotu jasu na 0, respektive 9999.

3.2.3 Využití více vláken při výpočtech jasu

Uvedené metody z přechozí podkapitoly pro výpočet matic L_{Lab} a jasu L efektivně využívají pouze jedno vlákno, a proto mají určitou výkonovou rezervu. Jak bylo zmíněno v kapitole 3.1.3 – program lze s využitím knihovny *Java Concurrency* optimalizovat pro využití více vláken. Princip „rozděl a panuj“ lze aplikovat jak na výpočet matice L_{Lab}, tak i na výpočet matice jasů, jeho implementování využíváme třídu *RecursiveTask*. Která rekurzivně vytváří nové instance sama sebe, dokud není práce rozdělena na požadovaný počet dílů.

Naše třída starající se o výpočet L_{Lab} matice se jmenuje *ConstructLlabMatrixForkTask*. Její definice je následující:

```
public class ConstructLlabMatrixForkTask extends RecursiveTask<double[][]>
```

Tato třída je potomkem zmíněné abstraktní třídy *RecursiveTask*. Generický parametr ve špičatých závorkách značí typ výsledku, který dostaneme vykonáním tohoto úkolu. V našem případě je to dvojdimenzionální pole – matice.

Všechny proměnné v této třídě a oba její konstruktory jsou vidět níže.

```
private static ImageProcessor processor;
private static BufferedImage src;
private static double[][] dst;
private static int rows;
private static int cols;
private static int threadCount;
private int startX;
private int endX;
private int startY;
private int endY;


private ConstructLlabMatrixForkTask(int startX, int endX, int startY, int endY) {
    this.startX = startX;
    this.endX = endX;
    this.startY = startY;
    this.endY = endY;
}

public ConstructLlabMatrixForkTask(ImageProcessor processor, BufferedImage src) {
    this.threadCount = Runtime.getRuntime().availableProcessors();
    this.processor = processor;
    this.src = src;
    this.rows = src.getHeight();
    this.cols = src.getWidth();
    this.dst = new double[rows][cols];
}
```

Fragment kódu 10: Třída ConstructLlabMatrixForkTask – proměnné a konstruktory

Prvních 6 proměnných je statických, tyto proměnné jsou sdíleny mezi jednotlivými instancemi. Každá instance potřebuje objekt typu *ImageProcessor* pro provádění výpočtů. Dále jsou zde dvě proměnné pojmenovány *src* a *dst*, tedy *source* (zdroj) a *destination* (cíl). Zdroj zde představuje zpracovávaná fotografie, ze které čteme jednotlivé pixely a cíl je matice, do které zapisujeme hodnoty L_{Lab}. Další dvě proměnné *rows* a *cols* jen značí rozměry fotografie. Poslední statickou proměnnou je *threadCount*, tedy počet vláken, která budou využívána.

V ukázce kódu můžeme vidět dva konstruktory, tedy metody vyvolané při vytváření objektu. Druhý z nich má nastaven veřejný přístup (*public*) a je volán při vytváření první instance této třídy. Jako parametry mu jsou předány instance tříd *ImageProcessor* a *BufferedImage*, ty jsou uloženy do statických proměnných, aby k nim mohly přistupovat i další instance třídy *ConstructLlabMatrixForkTask*. Dále je v tomto konstruktoru nastavena hodnota *threadCount* pomocí příkazu,

```
this.threadCount = Runtime.getRuntime().availableProcessors();
```

který, jak již název vypovídá, vrací počet virtuálních procesorů, které jsou operačnímu systému dostupné. Také jsou inicializovány zbylé statické proměnné *rows*, *cols* a *dst*, poslední zmíněné je přiřazena prázdná matice o rozměrech zdrojové fotografie.

První konstruktor, který má přístup nastaven na *private* a tedy může být volán jen z této třídy, slouží k vytváření dílčích úkolů, které zpracovávají vlastní segment zdrojové fotografie. Proto jsou mu jako parametry předávány čtyři proměnné typu *int*, které značí souřadnice X a Y, jimiž je daný segment vymezen. Tyto proměnné jsou v metodě uloženy do příslušných instančních proměnných.

O rozdělení fotografie na jednotlivé segmenty se stará následující metoda.

```
private RecursiveTask[] createSubtasks() {
    //vytvor pole s tolika Task instancem kolik je virtualnich procesoru
    ConstructLlabMatrixForkTask[] tasks =
            new ConstructLlabMatrixForkTask[threadCount];

    //vyska fotografie je delitelna threadCount bez zbytku -> rozdel fotografii
    //vertikalne
    if (rows % threadCount == 0) {
        startX = 0;
        endX = cols;
        int sectionHeight = rows / threadCount;
        //vytvor novy ForkTask, pridel mu segment obrazku a pridej ho do pole
        for (int i = 0; i < threadCount; i++) {
            startY = i * sectionHeight;
            endY = (i + 1) * sectionHeight;
            tasks[i] = new ConstructLlabMatrixForkTask(startX, endX, startY, endY);
        }
    //pouze sirka fotografie je delitelna threadCount bez zbytku -> rozdel fotografii
    //horizontalne
    } else if (cols % threadCount == 0) {
        startY = 0;
        endY = rows;
        int sectionWidth = cols / threadCount;

        for (int i = 0; i < threadCount; i++) {
            startX = i * sectionWidth;
            endX = (i + 1) * sectionWidth;
            tasks[i] = new ConstructLlabMatrixForkTask(startX, endX, startY, endY);
        }
    //ani jeden rozmer neni delitelny threadCount bez zbytku -> posledni segment vetsi
    } else {
        startX = 0;
        endX = cols;
        int sectionHeight = rows / threadCount;

        for (int i = 0; i < threadCount; i++) {
            startY = i * sectionHeight;
            //v posledni iteraci vytvor segment z toho, co zbylo
            if (i == threadCount - 1) {
                endY = rows;
            } else {
                endY = (i + 1) * sectionHeight;
            }
            tasks[i] = new ConstructLlabMatrixForkTask(startX, endX, startY, endY);
        }
    }
    return tasks;
}
```

Fragment kódu 11: Třída ConstructLlabMatrixForkTask – metoda createSubtasks

Nejprve musíme vytvořit prázdné pole typu *ConstructLlabMatrixForkTask*, jehož velikost bude stejná jako počet vláken procesoru, protože přesně na tolik částí chceme zpracování celé fotografie rozdělit. Další postup je závislý na rozměrech fotografie. Musíme si uvědomit, co by se stalo, kdybychom chtěli rozdělit fotografii na *n* segmentů např. vertikálně, a výška fotografie by nebyla bezezbytku dělitelná *n*. Došlo by k zaokrouhlení výšky segmentu, protože by nebyla celým číslem. Součet všech segmentů o dané výšce by však byl menší, nebo větší než výška fotografie. Mohlo by se stát, že by některé řádky nebyly zpracovány, jelikož by nepatřily žádnému ze segmentů. Abychom se tomuto vyhnuli, kontrolujeme podmínku, zda je výška fotografie, tedy počet řádků (*rows*) dělitelná počtem vláken (*threadCount*) bez zbytku a pokud ano, rozdělíme fotografii vertikálně. Pokud tato podmínka neplatí pro počet řádků, ale platí alespoň pro počet sloupců, dělíme fotografii horizontálně.

V případě, že neplatí tato podmínka ani u jednoho z rozměrů fotografie, rozdělíme ji tak, že poslední segment bude větší nebo menší než všechny ostatní. Konkrétně to řešíme tak, že vytváříme segmenty o stejné velikosti až do poslední iterace smyčky, kdy nastavíme velikost segmentu tak, aby pokryl zbývající část fotografie.

Vzhledem k tomu, že počty logických jader procesorů jsou sudá čísla (4, 6, 8, 12), nemělo by k poslední situaci docházet často, protože rozlišení digitálních fotografií jsou většinou dělitelná 4 nebo 8. Je však dobré mít jistotu pro situace s procesory, kdy je počet virtuálních jader 6 nebo 12.

Vytvoření segmentu obnáší vytvoření instance třídy *ConstructLlabMatrixForkTask* pomocí konstruktoru, který akceptuje jako parametry hraniční souřadnice segmentu. Tuto instanci přidáme do vytvořeného pole nazvaného *tasks*. Po rozdělení celé fotografie na segmenty z metody vrátíme referenci na toto pole.

Každá třída, která je potomkem *RecursiveTask* musí implementovat metodu *compute()*.

```
protected double[][] compute() {
    //pokud volano poprve, musime rozdelit fotografii na segmenty
    if (endX == 0) {
        ForkJoinTask.invokeAll(createSubtasks());
        return dst;
    } else {
        return process();
    }
}
```

Fragment kódu 12: Třída ConstructLlabMatrixForkTask - metoda createSubtasks()

Tato metoda vyhodnotí, zda se jedná o první instanci třídy *ConstructLlabMatrixForkTask* a nikoli o již vytvořený segment, pokud ano, zavolá metodu *createSubtasks(),* která rozdělí zdrojovou fotografii na jednotlivé segmenty a předá její výsledek, tedy pole objektů *ConstructLlabMatrixForkTask* metodě *ForkJoinTask.invokeAll*, která rekurzivně volá metodu *compute()* všech objektů tohoto pole. Pokud se však nejedná o první instanci této třídy a již o vytvořenou podúlohu, tak se zavolá metoda *process()*, která provede výpočet daného segmentu a vrátí matici *dst* obohacenou o výsledky tohoto výpočtu.

Metoda *process()* pouze volá metodu *constructLlabMatrix* z třídy *ImageProcessor*, která je variací stejnojmenné metody zmíněné na straně 41. Tato metoda se od zmíněné liší jen tím, že zpracovává pouze segment fotografie vytyčený hranicemi, které jsou jí předány jako parametry. Původní metoda zpracovává fotografii celou.

```
private double[][] process() {
    return processor.constructLlabMatrix(src, dst, startX, endX, startY, endY);
}
```

Fragment kódu 13: Třída ConstructLlabMatrixForkTask - metoda process()

V tuto chvíli je již třída kompletní a můžeme ji využít k sestavení matice L_{Lab} vytvořením fondu vláken, která se postarají o provedení celého úkolu.

Sestavení matice jasů L s využitím více vláken vypadá téměř identicky. Třída k tomu určená se jmenuje *ConstructLuminanceMatrixForkTask* a od výše rozebírané třídy se liší pouze v proměnných, které přibyly. Pro výpočet jasů je nutno ještě definovat koeficienty A a B, čas expozice t a clonové číslo F. Tyto parametry jsou také předány v konstruktoru a uloženy do statických proměnných, jelikož jsou využívány všemi instancemi.

3.2.4 Vytvoření HDR matice jasů

V této sekci je nejprve nutné představit dvě entity, které jsme vytvořili pro reprezentaci fotografií. První z nich je třída nazvaná *UnmergedImage* a druhá *MergedImage*. Jak z názvu vypovídá, první

zmiňovaná představuje nahranou fotografii, kterou lze použít k vytvoření HDR snímku, reprezentovaného třídou *MergedImage*. Obě třídy vlastně jen zaobalují jednotlivé matice jasů a L_{Lab} a mimo jiné obsahují *set* a *get* metody pro přístup k těmto proměnným. Obě třídy dědí z interface nazvaného *MyImage*, které slouží k tomu, abychom mohli vytvořit proměnnou typu *MyImage*, a uložit do ní instanci kterékoliv z implementací tohoto interface.

Třída, která byla vytvořena pro spojení několika fotografií a jejich matic jasů do jedné je pojmenována *ImageMerger*. Obsahuje metodu *MergeImages*, jejíž výstupem je instance zmíněné třídy *MergedImage*, která obsahuje matici L_{Lab} a matici jasů, které vznikly sloučením více matic z třídy *UnmergedImage*.

Tato metoda je vidět na následující straně. Jako parametr jí je předán list objektů *UnmergedImage*, ve kterém jsou uloženy nahrané fotografie. Metoda nejprve vytvoří novou instanci třídy *MergedImage*, kterou bude poté vracet. Dále jsou pomocí metody *initializeImages* vypočteny všechny L_{Lab} a jasové matice všech nahraných fotografií, některé však již mohly být vypočteny, proto se provede výpočet jen těch, které ještě neexistují. Je zvolena referenční fotografie a vytvoří se kopie jejích matic, abychom je mohli upravovat.

Nyní již implementujeme známý algoritmus č. 3, který byl vysvětlen na straně 32.

Vytvoří se list nazvaný *potentialReplacements*, do kterého budeme ukládat fotografie, které mají na stejné pozici platnou hodnotu L_{Lab}, která by mohla být blíže 50, nežli aktuální hodnota v referenční matici. Následují tři smyčky – první dvě iterují přes jednotlivé indexy matice, třetí z nich iteruje mezi jednotlivými fotografiemi. Pokud má aktuální fotografie na daném indexu platnou L_{Lab} hodnotu, přidáme ji do listu *potentialReplacements*. Po doběhnutí třetí smyčky se zkontroluje, zda byla vůbec ve všech fotografiích nalezena jediná platná hodnota a pokud ano, z těchto hodnot se vybere ta nejblíže k 50 pomocí metody *closestTo50*, která vrací referenci na fotografii, která tuto hodnotu obsahuje. Do referenčních matic se poté uloží hodnoty z této reference. List *potentialReplacements* se před posunem na další index musí vyprázdnit metodou *clear()*.

Po doběhnutí všech smyček se do vytvořené instance *MergedImage* uloží vytvořené matice a tato instance se vrátí.

```
private UnmergedImage closestTo50(List<UnmergedImage> contenders, int j, int i) {
    UnmergedImage closest = contenders.get(0);

    for (int a = 1; a < contenders.size(); a++) {
        double currentValue = contenders.get(a).getlLabMatrix()[j][i];
        if (Math.abs(closest.getlLabMatrix()[j][i] - 50)
                > Math.abs(currentValue - 50)) {
            closest = contenders.get(a);
        }
    }
    return closest;
}
```

Fragment kódu 14: Třída ImageMerger – metoda closestTo50

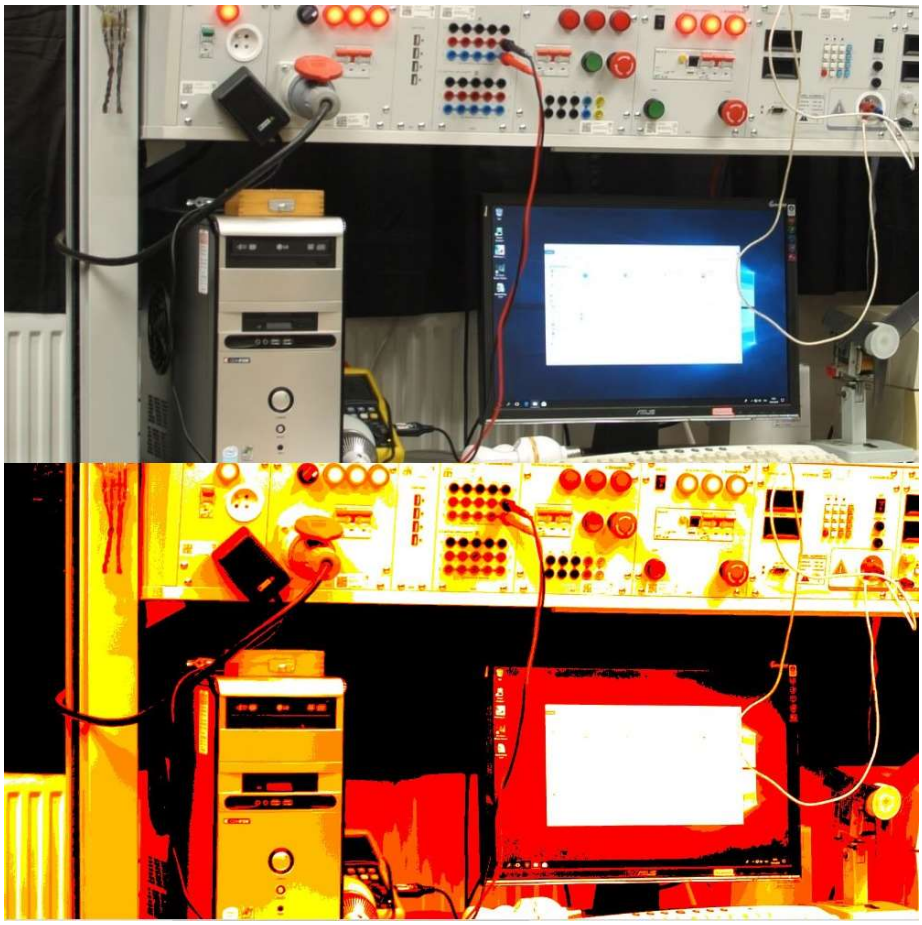
```
public MergedImage MergeImages(List<UnmergedImage> images) {

    MergedImage merged = new MergedImage(images);
    //vypocti zbyle matice z nahranych fotografii
    initializeImages(images);
    //zvolme referencni fotografii
    UnmergedImage baseImage = images.get(0);

    //vytvor kopii obou matic abys nezasahoval do puvodnich
    double[][] mergedLlabMatrix = Arrays.stream(baseImage.getlLabMatrix())
            .map(double[]::clone)
            .toArray(double[][]::new);
    double[][] mergedLuminanceMatrix = Arrays.stream(baseImage.getLuminanceMatrix())
            .map(double[]::clone)
            .toArray(double[][]::new);

    //zavedme list pro ukladani potencialnich nahrad
    ArrayList<UnmergedImage> potentialReplacements = new ArrayList<>();

    for (int j = 0; j < mergedLlabMatrix.length; j++) {
        for (int i = 0; i < mergedLlabMatrix[0].length; i++) {
            //iteruj mezi jednotlivymi fotografiemi a hledej vhodnejsi nahrady
            for (UnmergedImage img : images) {
                double lLabValue = img.getlLabMatrix()[j][i];
                if (validLValue(lLabValue)) {
                    potentialReplacements.add(img);
                }
            }
            //vyber z potencialnich nahrad tu nejblize LLab = 50
            //pote uloz do referencni
            if (!(potentialReplacements.size() == 0)) {
                UnmergedImage replacingImage =
                        closestTo50(potentialReplacements, j, i);
                if(replacingImage != baseImage) {

                    mergedLlabMatrix[j][i] =
                            replacingImage.getlLabMatrix()[j][i];

                    mergedLuminanceMatrix[j][i] =
                            replacingImage.getLuminanceMatrix()[j][i];
                }
            }
            potentialReplacements.clear();
        }
    }
    merged.setlLabMatrix(mergedLlabMatrix);
    merged.setLuminanceMatrix(mergedLuminanceMatrix);
    return merged;
}
```

Fragment kódu 15: Třída ImageMerger – metoda mergeImages

3.2.5 Vytváření jasové mapy

Po kalkulaci jasové matice je účelné zpracovanou fotografii rekonstruovat jako jasovou mapu – tedy fotografii, kde barva každého pixelu je dána pouze jeho vypočteným jasem. Pro jasovou mapu lze udělat analogii s displejem termokamery, kde je barva pixelů také odvozená od měřené hodnoty, nikoli však jasu ale teploty. Tato mapa se poté zobrazí v GUI místo originální fotografie, jelikož je v ní snazší orientace při odečítání jasů.

Ukázku jasové mapy a původní fotografie můžeme vidět na následujícím obrázku. Světlejší barva značí vyšší jas.



Obr. 13: Jasová mapa a její předloha

Pro uložení hotové mapy a také informace o jejích barvách a hraničních hodnotách jasů jednotlivých barev byla vytvořena třída *ReconstructedImage*, která je potomkem standardní třídy *BufferedImage* pouze rozšířená o pole barev a pole desetinných čísel. Tato desetinná čísla jsou hodnoty jasu, které definují intervaly, pro jejichž jas byla použita daná barva. Zmíněná třída také obsahuje konstantní pole pěti defaultních barev, ze kterých je sestaven snímek na obr. 13.

K samotnému vytváření jasové mapy byla zavedena třída *ColorMapper*, která obsahuje dvě statické metody, které vytvoří zpracovanou instanci třídy *ReconstructedImage*. První z těchto metod je vidět na další straně.

```
public static ReconstructedImage reconstructImage(MyImage image) {
    //pokud nejsou matice vypocteny, proved jejich vypocet
    if (!image.isInitialized()) {
        cache.initializeImageMatrices(image, false);
    }
    //vezmi defaultni barvy
    Color[] colors = ReconstructedImage.getDefaultColors();

    double[][] luminanceMatrix = image.getLuminanceMatrix();

    //vytvor intervaly
    double[] intervalThreshholds = calculateThreshholds(image, colors.length);

    int height = image.getlLabMatrix().length;
    int width = image.getlLabMatrix()[0].length;

    //vytvor prazdnou fotografii
    ReconstructedImage reconstructedImg =
            new ReconstructedImage(width, height, BufferedImage.TYPE_INT_RGB);

    //vypln fotografii pixely
    for (int j = 0; j < height; j++) {
        for (int i = 0; i < width; i++) {
            //uloz jas
            double luminance = luminanceMatrix[j][i];
            Color color;
            //nastav barvu na zaklade hodnoty jasu
            if (luminance == 0) {
                color = Color.BLACK;
                reconstructedImg.setRGB(i, j, color.getRGB());
            } else if (luminance == 9999) {
                color = Color.WHITE;
                reconstructedImg.setRGB(i, j, color.getRGB());
            } else {
                for (int k = 0; k < intervalThreshholds.length; k++) {
                    if (luminance >= intervalThreshholds[k]) {
                        color = colors[k];
                        reconstructedImg.setRGB(i, j, color.getRGB());
                        break;
                    }
                }
            }
        }
    }
    reconstructedImg.setLuminanceThreshholds(intervalThreshholds);
    return reconstructedImg;
}
```

Fragment kódu 16: Třída ColorMapper – metoda reconstructImage 1

Metoda *reconstructImage* má dvě varianty, první z nich ukazuje fragment kódu 16. Tato varianta používá výše zmíněné defaultní barvy a rozsah změřených jasů je rozdělen do 5 stejně velkých intervalů. Tyto intervaly jsou vytvořeny metodou *calculateThreshholds*, která jednoduše vezme rozdíl mezi maximálním a minimálním změřeným jasem (pouze platné hodnoty) a rozdělí ho do pěti stejně velkých intervalů. Hranice těchto intervalů poté vrátí v poli typu *double*.

Vytvoří se nová instance typu *ReconstructedImage* a do ní jsou postupně vkládány barevné pixely. Nejprve se zkontroluje, zda je daná hodnota jasu platná. Pokud není, přeexponovanému pixelu se nastaví bílá barva a podexponovanému barva černá. Ostatním pixelům je nastavena barva ze zadaného pole barev, která má v poli stejný index jako interval jasu, do kterého spadá daný jas.

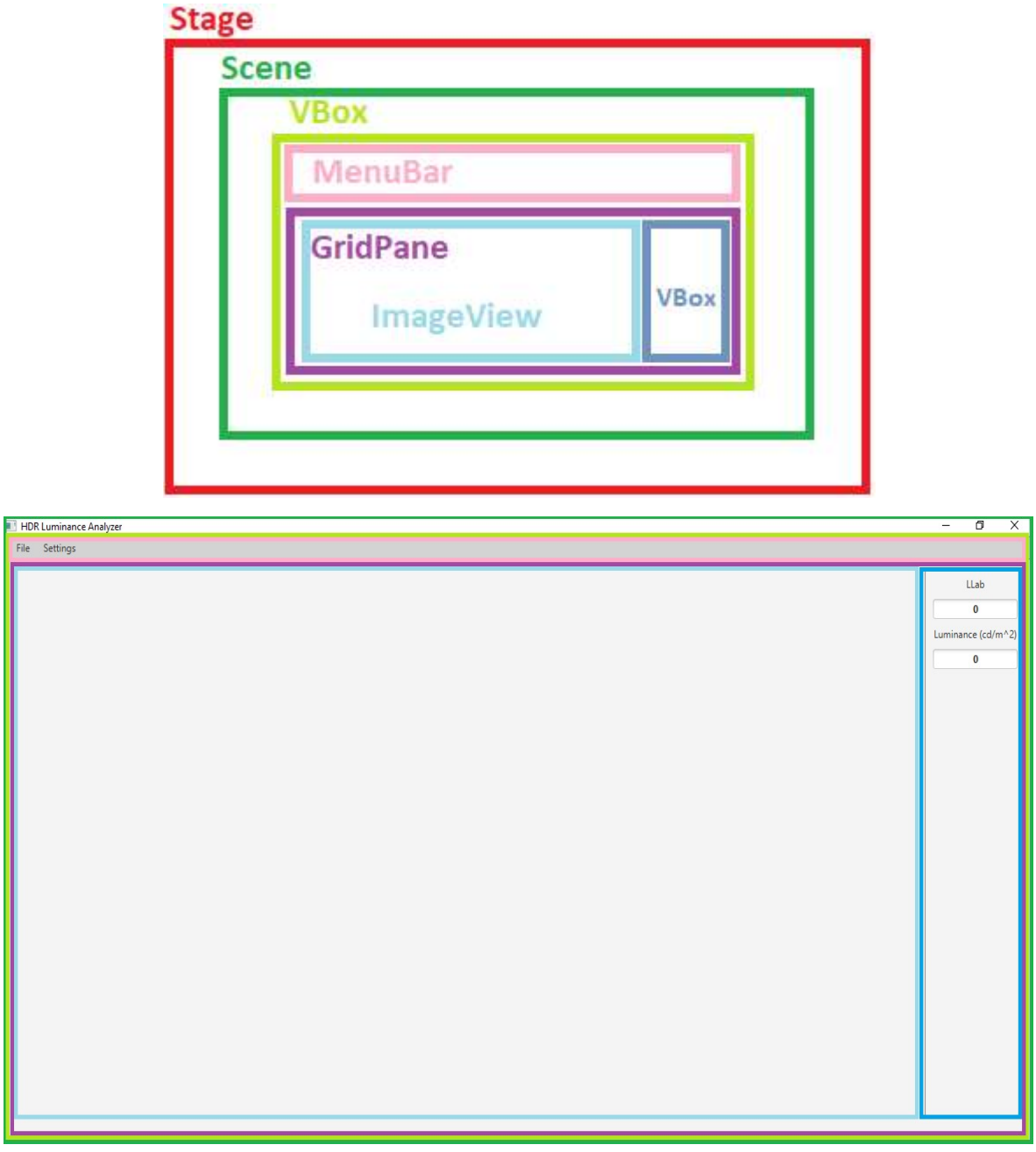
Druhá varianta této metody se liší pouze v získání barev a intervalů jasu, ty jsou jí tentokrát předány jako parametr. Tato verze je použita v případě, že tyto parametry nastavíme v GUI.

3.3 Realizace uživatelského rozhraní

V této sekci budou popsány hlavní prvky uživatelského rozhraní a jejich funkce.

3.3.1 Základní strukturální rozložení GUI

Základní struktura prvků rozložení je ilustrována na obr. 14.



Obr. 14: Základní struktura GUI

V první části obr. 14 je vidět struktura kontejnerů v úvodní obrazovce uživatelského rozhraní. Druhá část je screenshot aplikace s barevně vyznačenými prvky korespondujícími s kontejnery v první části. Jak již bylo popsáno v 3.1.2, kořenovým prvkem uživatelského rozhraní je vždy *Stage*, to musí však vždy mít i svoji scénu, tyto dva prvky proto na screenshotu splývají a je tudíž vyznačena jen scéna. Naše scéna obsahuje prvek rozložení *VBox*, který obsahuje dva vertikálně uspořádané prvky. Jedním z nich je *MenuBar*, což je klasická horní lišta, kterou známe z většiny Windows aplikací. Druhým je *GridPane*, což je prvek rozložení, který umožňuje mřížovité uspořádání. V tomto *GridPane* se nachází dva prvky – na levé straně je to *ImageView*, což je pole kde se zobrazuje zvolená fotografie a na pravé straně je to *VBox*, který obsahuje další prvky pro interakci s fotografií.

Hlavním vstupním bodem do aplikace je metoda *start* v třídě, která je potomkem abstraktní třídy javafx.application.Application. Naše implementace této třídy se jmenuje *Main*. Metodu *start* lze vidět níže, pro stručnost z ní bylo vyjmuto formátování prvků a také obsahuje pouze kód týkající se prvků představených na obr. 14, další prvky budou rozebrány později.

```
private final ImageView imv = new ImageView();

public void start(Stage primaryStage) {

    VBox root = new VBox();
    Scene scene = new Scene(root, 1200, 1000, Color.WHITE);

    //menu
    MenuBar menu = new MenuBar();
    Menu menuFile = new Menu("File");
    Menu menuSettings = new Menu("Settings");
    menu.getMenus().addAll(menuFile, menuSettings);

    //grid
    GridPane gridpane = new GridPane();

    ScrollPane scrollPane = new ScrollPane(imv);

    //Vbox napravo
    final Label luminance = new Label("Luminance (cd/m^2)");

    final TextField lumTextField = new TextField("0");

    final Label llabLabel = new Label("LLab");

    final TextField llabTextField = new TextField("0");

    final Label coords = new Label("");

    VBox vertBox = new VBox(10);
    vertBox.getChildren().addAll(llabLabel, llabTextField, luminance, lumTextField,
        coords);

    //vypln grid
    gridpane.add(scrollPane, 0, 0);
    gridpane.add(vertBox, 1, 0);
    root.getChildren().addAll(menu, gridpane);

    primaryStage.setScene(scene);
    primaryStage.show();
}
```

Fragment kódu 17: Třída Main – metoda start

Jak můžeme vidět, metodě *start* je předána jako parametr instance třídy *Stage*. O vytvoření této instance se stará JavaFX samotné, po startu programu ji injektuje do této metody. Tato metoda je převážně tvořena definicemi jednotlivých prvků a vytváření jejich vzájemných vazeb.

Nejprve je vytvořen kořenový *VBox*, který byl výše znázorněn světle zelenou barvou, ten je při vytváření objektu třídy *Scene* předán jejímu konstruktoru. Dále mu jsou mimo jiné předány rozměry a barva pozadí. Následně je vytvořena horní lišta třídy *MenuBar* a její jednotlivé položky „File“ a „Settings“, které jsou třídy *Menu*. Tyto položky jsou poté do lišty přidány pomocí metody *addAll*. Poté je vytvořen kontejner *GridPane*, který je vyznačen fialovou barvou.

O řádek níže je vytvořena instance třídy *ScrollPane*, což je dosud nezmíněný prvek, který bude použit k zabalení objektu třídy *ImageView*, který slouží pro zobrazení fotografie. Je použit z toho důvodu, že pokud je zobrazovaná fotografie větší, než je prostor vyhraněný *ImageView*, objeví se vpravo a pod fotografií posuvné lišty. Tyto lišty nám umožní se posouvat po fotografii, což bude také možné přidržením levého tlačítka myši a tažením do libovolného směru.

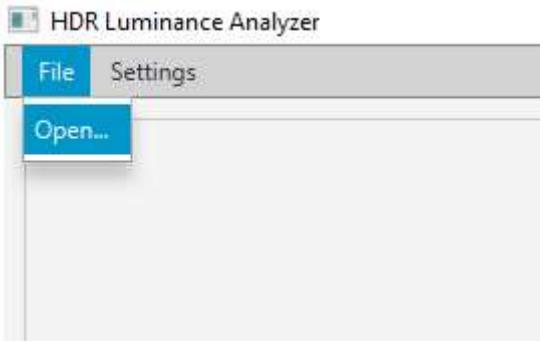
Na dalších řádcích se vytvářejí prvky *VBoxu* na pravé straně (tmavě modrá z obr. 14). Jedná se o dva nadpisy pro textová pole, která slouží pro zobrazení aktuální hodnoty jasu a L_{Lab}. Třetí nadpis slouží k zobrazení aktuální souřadnice kurzoru myši a je viditelný až když se nahraje první fotografie. Poté je vytvořen samotný *VBox* a tyto prvky jsou do něj umístěny.

V poslední části jsou do *GridPane* umístěni jeho dva potomci – *ScrollPane* a *VBox* na pravé straně. Samotný *GridPane* je poté přidán do kořenového *VBoxu* spolu s vrchním menu. Scéna je potom propojena se *Stage* a pomocí metody *show* je *Stage* zobrazeno.

3.3.2 Ovládací prvky

Nahrání fotografie

Úplně základním požadavkem na naše GUI je, abychom mohli nahrát fotografii pro zpracování. K tomu slouží tlačítko „Open“ v horním menu pod položkou „File“. Tento způsob otvírání souboru je nepsaným standardem mezi většinou desktopových aplikaci (MS Office, Adobe Photoshop aj.), a proto nebyl důvod ho nepoužít i v našem případě.



Obr. 15: Tlačítko „Open"

Po klinutí na tuto položku se otevře klasické okno pro výběr souboru, které nám umožní otevřít fotografii. Položku „Open“ je však nejprve nutné přidat do menu. V metodě *start* vytvoříme proměnnou typu *MenuItem* a přiřadíme ji záložce „File“, viz fragment kódu 18. K samotnému otevření okna s prohlížečem souborů slouží třída *FileChooser*. Je vytvořena její instance a té je přiřazen nový *ExtensionFilter*, kterým specifikujeme, jaký typ souboru lze nahrát podle jeho přípony.

Položce „Open“ také musíme říci, co má provést, pokud se na ni klikne. To se provede přiřazením event handleru pomocí metody *setOnAction* a lambda výrazu. V tomto lambda výrazu nejprve

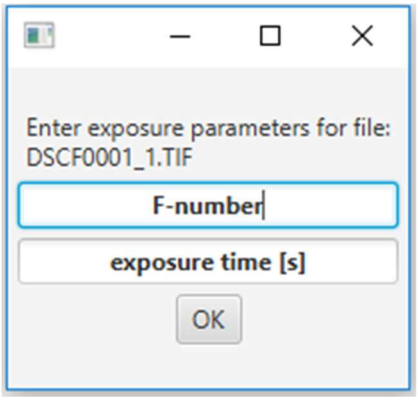
vytvoříme proměnnou typu *File* a do ní uložíme fotografii, kterou vybereme po otevření okna pro výběr souboru. Toto okno se otevře po zavolání metody *showOpenDialog*, tato metoda patří naší instanci třídy *FileChooser* a jako parametr jí musíme předat naši instanci *Stage*.

Ve chvíli, kdy máme vybraný soubor uložený do proměnné *file*, ho můžeme předat metodě *loadImg*, která se postará o další postup zpracování fotografie.

```
public void start(Stage primaryStage) {

    ...

    //menu
    Menu menuFile = new Menu("File");
    MenuItem openItem = new MenuItem("Open...");
    menuFile.getItems().add(openItem);

    ...

    //fileChooser
    FileChooser fileChooser = new FileChooser();
    List<String> extensions = Arrays.asList("*.jpg", "*.jpeg", "*.tif", "*.tiff",
            "*.png", "*.bmp");
    fileChooser.getExtensionFilters().add(
            new FileChooser.ExtensionFilter("Image files", extensions));
    openItem.setOnAction(event -> {
        File file = fileChooser.showOpenDialog(primaryStage);

        if (file != null) {
            try {
                loadImg(file, vertBox);
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    });

    ...

}
```

Fragment kódu 18: Třída Main – metoda start: nahrání souboru

Metoda *loadImg* (viz fragment kódu 19Fragment kódu 19) má mimo jiné za úkol získat od uživatele po nahrání fotografie parametry expozice pomocí následujícího vyskakovacího okna.



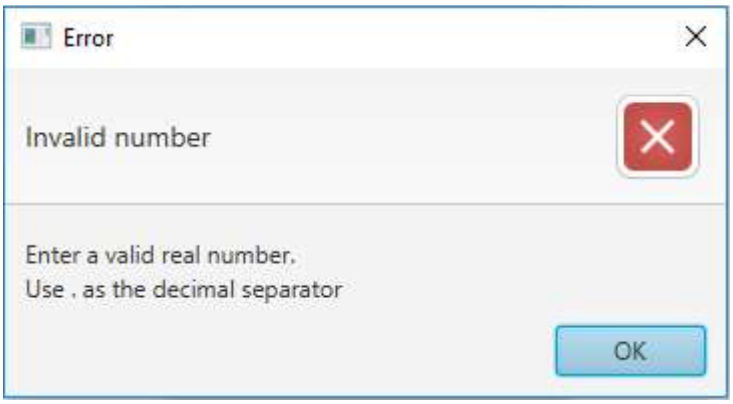
Obr. 16: Vyskakovací okno – zadání expozice

Pro vytvoření vyskakovacího okna je nutné vytvořit novou instanci *Stage*. Jak je vidět ze screenshotu, okno obsahuje dvě textová pole, jeden nadpis a jedno tlačítko. Všechny tyto objekty jsou definovány ve zmíněné metodě. Tyto prvky jsou poté agregovány do jednoho *VBoxu*, ten je poté předán nově vytvořené scéně a ta je nastavena do *Stage*, které je poté zobrazeno metodou *show*.

```
private void loadImg(File file, VBox rightMenu) throws IOException {

        final String fileName = file.getName();
        final BufferedImage image = ImageIO.read(file);

        Stage exposureInputStage = new Stage();

        Label exposureLabel = new Label("Enter exposure parameters for file:\n"
        + fileName);
        TextField fNumberTextField = new TextField("F-number");
        TextField exposureTimeTextField = new TextField("exposure time [s]");

        Button okBtn = new Button("OK");
        okBtn.setOnAction( e -> {
            String fNumberText = fNumberTextField.getText();
            String exposureTimeText = exposureTimeTextField.getText();

            if (isValidDouble(fNumberText) && isValidDouble(exposureTimeText)) {
                double fNumber = Double.parseDouble(fNumberText);
                double exposureTime = Double.parseDouble(exposureTimeText);

                UnmergedImage toBeAdded = new UnmergedImage(image, exposureTime,
                                                fNumber, fileName);
                imgDataCache.getImageList().add(toBeAdded);

                exposureInputStage.close();
                setupImageHandlingButtons(rightMenu);
                addImageMenuItem(toBeAdded, rightMenu);
            } else {
                alertInvalidNumber();
            }
        });

        VBox vBox = new VBox(exposureLabel, fNumberTextField, exposureTimeTextField,
                             okBtn);

        Scene scene = new Scene(vBox, 200, 130);
        exposureInputStage.setScene(scene);
        exposureInputStage.show();
}
```

Fragment kódu 19: Třída Main – metoda loadImg

Nyní rozeberme, co se stane po stisku tlačítka „OK“. Jeho chování je nastaveno, stejně jako u předchozího tlačítka „Open“, pomocí metody *setOnAction* a lambda výrazu. Nejprve je uložen text z obou textových polí do proměnné typu *String* (řetězec). Poté je provedena kontrola, zda jsou zadané řetězce platnými desetinnými čísly, o to se stará metoda *isValidDouble*. Pokud řetězce neprojdou touto kontrolou, je vyvoláno další vyskakovací okno (obr. 17), pomocí metody *alertInvalidNumber*. Po zavření tohoto okna je možné zadat parametry znovu.



Obr. 17: Upozornění na špatně zadané číslo

V případě, že vstup kontrolou projde, jsou oba řetězce uloženy do proměnných typu *double*. Dále je vytvořena instance třídy *UnmergedImage*, do které je uložena nahraná fotografie, která byla na začátku metody uložena do proměnné typu *BufferedImage*. Jako další parametry jsou konstruktoru *UnmergedImage* předány právě zadané parametry expozice a také název fotografie.

Přidání položky fotografie do menu vpravo

Dalším postupem je uložení této instance *UnmergedImage* do listu fotografií, který patří instanci třídy *ImageDataCache*. Tato třída slouží jednak jako úložiště nahraných fotografií, ale také obsahuje několik metod k manipulaci těchto fotografií. Poté je zavolána metoda *close* pro uzavření *Stage* vyskakovacího okna. Také jsou zavolány další dvě metody, kterým je jako parametr předán odkaz na pravostranné menu (*VBox*), který byl také předán metodě *loadImg* jako parametr, jelikož byl tento *VBox* definován jako lokální proměnná v metodě *start*.

Nejprve si ukažme druhou z těchto dvou metod – *addImageMenuItem*.

```
private void addImageMenuItem(UnmergedImage addedImage, VBox rightMenu) {

    Label imgName = new Label(addedImage.getImgName());

    CheckBox checkBox = new CheckBox();
    checkBox.setOnAction(e -> {
        addedImage.setSelected(checkBox.isSelected());
    });

    Button detailsBtn = new Button("Details");
    detailsBtn.setOnAction(e -> {
        showImageDetails(addedImage);
    });

    HBox hBox = new HBox(checkBox, detailsBtn);

    ImageMenuItem imgMenuItem = new ImageMenuItem(addedImage, imgName, hBox);
    rightMenu.getChildren().add(imgMenuItem);
}
```

Fragment kódu 20: Třída Main – metoda addImageMenuItem

Tato metoda má za úkol do *VBoxu*, k němuž obdržela referenci jako parametr, přidat položku *ImageMenuItem*, která je potomkem *VBox*. Tato třída byla vytvořena kvůli tomu, abychom měli přístup k referenci na fotografii, která k dané položce patří. Jedná se tedy vlastně o *VBox* s jednou instanční proměnnou navíc typu *UnmergedImage*. Na obr. 18 můžeme vidět, jak hotový objekt *ImageMenuItem* vypadá.

Obr. 18: Objekt ImageMenuItem v menu napravo

Metoda *addImageMenuItem* nejprve vytvoří *Label*, který obsahuje název nahrané fotografie. Dále vytvoří *CheckBox* (zaškrtávací políčko), který slouží k označení fotografie pro další manipulaci. Při zaškrtnutí se stav políčka propíše do *boolean* proměnné dané instance *UnmergedImage*, abychom mohli jednoduše zjistit, zda je fotografie označena. Posledním elementem je tlačítko „Details“, které při stisknutí zobrazí ostatní parametry fotografie, konkrétně její rozlišení a parametry expozice. Funkčnost tohoto tlačítka zajišťuje metoda *showImageDetails*, která podobně jako metoda *loadImg* vytvoří novou *Stage* a zobrazí ji, viz obr. 19.

Všechny tři prvky jsou umístěny do instance *ImageMenuItem* a přidány do pravého menu.

Obr. 19: Detaily fotografie

Přidání tlačítek pro manipulaci s fotografiemi

Druhá z metod volaných při stisku tlačítka „OK“ po zadání hodnot expozice se jmenuje *setupImageHandlingButtons*. Jejím úkolem je přidat dvě tlačítka do menu vpravo. Tato tlačítka nesou názvy „View“ a „Delete“. První z tlačítek slouží k tomu, abychom dali povel na zpracování vybraných fotografií a následné zobrazení v *ImageView*. Druhé z nich slouží k vymazání označených položek z menu a také k odstranění fotografií z paměti.

Metoda nejdřív zkontroluje, zda již nebyla tato tlačítka do menu přidána. K tomuto slouží *boolean imageButtonsCreated*, který je instanční proměnou třídy *Main*. Pokud je kontrola úspěšná, dojde k vytvoření tlačítka „View“ a nastavení jeho chování při stisknutí, opět metodou *setOnAction*. Při jeho stisku se tedy vyberou všechny označené fotografie a uloží se do listu. Pokud je list prázdný, nic se nestane. Pokud obsahuje pouze jednu fotografii, tato fotografie se uloží do proměnné pojmenované *displayedImage*, která patří dané instanci třídy *Main*. Pokud je nalezeno více označených fotografií, tyto fotografie se sloučí do jedné pomocí metody *mergeAllImages* z třídy *ImageDataCache*. Tato metoda pouze zavolá metodu *mergeImages* její instance známé třídy *ImageMerger* a vrátí jako výsledek objekt třídy *MergedImage*. Ten se poté nastaví do proměnné *displayedImage*. Ještě před vytvořením HDR matice jasů se však zkontroluje, zda mají všechny fotografie stejné rozměry. Pokud tomu tak není, dojde k upozornění podobnému tomu na obr. 17.

Nakonec se zavolá metoda *refreshView*, která zajistí, že se všechny změny projeví i vizuálně. V této metodě se zavolá metoda *reconstructImage* třídy *ColorMapper*, která hodnoty jasu z aktuálního *displayedImage* namapuje na barevné pixely a vznikne tak rekonstruovaná fotografie v barevné škále odpovídající hodnotám jasu. Tato fotografie se poté předá objektu *ImageView*, který tvoří okno v levé části obrazovky.

```
private void setupImageHandlingButtons(VBox rightMenu){
    if(!imageButtonsCreated) {
        Button displayBtn = new Button("View");

        displayBtn.setOnAction(e -> {
            List<UnmergedImage> selectedImgs = imgDataCache.getSelectedImages();
            if (!selectedImgs.isEmpty()) {
                if (selectedImgs.size() == 1) {
                    this.displayedImage = selectedImgs.get(0);
                } else {
                    if(imgDataCache.imgDimensionsEqual()) {
                        this.displayedImage = imgDataCache.mergeAllImages();
                    }else{
                        alertDimensionsNotEqual();
                    }
                }
                refreshView();
            }
        });

        Button deleteBtn = new Button("Delete");

        deleteBtn.setOnAction(e -> {
            List<UnmergedImage> selectedImgs = imgDataCache.getSelectedImages();
            if (selectedImgs != null) {
                //vezmi polozky z praveho menu
                List<ImageMenuItem> imgMenuItems = rightMenu.getChildren().stream()
                        .filter(elem -> elem instanceof ImageMenuItem)
                        .map(item -> (ImageMenuItem) item)
                        .collect(Collectors.toList());
                //odstran vybrane polozky z praveho menu
                imgMenuItems.stream()
                        .forEach(a -> {
                            UnmergedImage img = a.getImageReference();
                            if (selectedImgs.contains(img)) {
                                rightMenu.getChildren().remove(a);
                                imgDataCache.getImageList().remove(img);
                            }
                        });
                refreshView();
            }
        });
        HBox box = new HBox(displayBtn, deleteBtn);
        rightMenu.getChildren().add(box);
        imageButtonsCreated = true;
    }
}
```

Fragment kódu 21: Třída Main – metoda setupImageHandlingButtons

Druhým přidávaným tlačítkem je tlačítko „Delete“, které má prostou funkci odstranit z menu a z paměti označené fotografie. Opět jsou nejdříve vybrány označené fotografie. Poté jsou vybrány z pravého menu vybrány pouze položky třídy *ImageMenuItem*, využitím funkcionality *streamů*. Každá z těchto položek je poté zkontrolována, zda se fotografie, ke které položka patří, nachází v listu

označených fotografií. Pokud tomu tak je, je daná položka odstraněna a z listu všech fotografií v třídě *ImageDataCache* je tato fotografie také odstraněna. Na konci je opět zavolána metoda *refreshView*.

Obě tlačítka jsou uspořádána horizontálně pomocí *HBoxu* a vložena do pravého menu. Výsledek této metody je vidět na obr. 20. Proměnná *imageButtonsCreated* je nastavena na *true*, aby při nahrání další fotografie nedošlo k přidání těchto tlačítek znovu.

Obr. 20: Pravé menu s tlačítky View a Delete

3.3.3 Odečítání hodnot jasu

Na obr. 20 je vidět screenshot pravého menu, dvě z položek jsou zaškrtnuté. Tyto fotografie se po stisku tlačítka „View“ sloučí do jedné a poté se jejich verze v jiné barevné škále zobrazí v levé části okna programu. Ještě bychom potřebovali, aby byla zobrazená fotografie interaktivní. Interaktivní v tom smyslu, že po přejetí myší nad libovolným pixelem se v pravém menu v textových polích „LLab“ a „Luminance“ zobrazí hodnota L_{Lab}, respektive jasu L. Ještě se pod existuje nadpis pro zobrazení aktuálních souřadnic, ten se však zobrazí až při vypočtení všech matic zobrazované fotografie. Důležité je zmínit, že jak matice, tak i fotografie mají počáteční souřadnici [0, 0] v levém horním rohu

K tomuto se musíme vrátit zpět k metodě *start()*, kde byla tyto textová pole definována, viz fragment kódu 22. Třída *ImageView* k tomuto účelu poskytuje metodu *setOnMouseMoved*, která se volá při každém pohybu myši v okně *ImageView*. My tedy potřebujeme, aby při každém pohybu myši byla do textových polí nahrána aktuální hodnota jasu a L_{Lab}. Jako parametr tedy předáme této metodě event handler vyjádřený lambda výrazem.

V tomto lambda výrazu při každém pohybu myši zkontrolujeme, zda má daná fotografie vypočtenou matici L_{Lab} i matici jasu. Toto je provedeno *if* podmínkou a instanční metodou nazvanou *isInitialized* rozhraní *MyImage*, která vrací proměnnou typu *boolean*. Poté můžeme jednotlivým textovým polím předat hodnoty jasu, L_{Lab} a aktuálních souřadnic. Aktuální souřadnice kurzoru myši vůči fotografii získáme voláním metod *e.getX()* a *e.getY()*. Písmeno *e* (od event) zde představuje zdroj

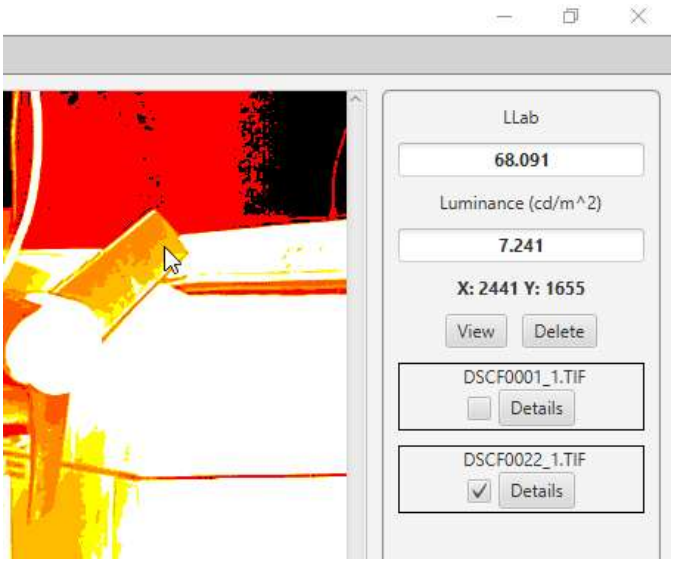
události (eventu), který ji vyvolal, v tomto případě je zdrojem naše instance třídy *ImageView*. Tyto dvě metody tedy patří této instanci.

Souřadnice si uložíme do proměnných a můžeme je předat objektu *Label* metodou *setText*. Zobrazíme tak v tomto nadpisu aktuální souřadnici kurzoru. Metody *getX* a *getY* však vrací desetinné číslo, proto je třeba je zaokrouhlit dolu (matice je indexována od 0, ale *ImageView* od 1 zaokrouhlením posledního indexu nahoru bychom sahali na neexistující index), a přetypovat na *int*.

```
private final ImageView imv = new ImageView();
private MyImage displayedImage;

public void start(Stage primaryStage) {

    final Label luminance = new Label("Luminance (cd/m^2)");
    final TextField lumTextField = new TextField("0");

    final Label llabLabel = new Label("LLab");
    final TextField llabTextField = new TextField("0");

    final Label coords = new Label("");

    imv.setOnMouseMoved(e -> {
        if (this.displayedImage.isInitialized()) {

            int x = (int) Math.floor(e.getX());
            int y = (int) Math.floor(e.getY());

            coords.setText("X: " + x + " Y: " + y );

            llabTextField.setText(String.format("%.3f",
                    this.displayedImage.getlLabMatrix()[y][x]));

            lumTextField.setText(String.format("%.3f",
                    this.displayedImage.getLuminanceMatrix()[y][x]));
        }
    });
}
```

Fragment kódu 22: Třída Main – metoda start: zobrazení hodnot jasu

Oběma textovým polím poté předáme jejich příslušné hodnoty jasu L a L_{Lab} také metodou *setText*. Aktuální hodnoty získáme voláním get metody objektu typu *MyImage* pro příslušnou matici. Umístěním souřadnic v obráceném pořadí (první index v matici značí řádek, druhý sloupec) do hranatých závorek obdržíme hodnotu na dané pozici. Na dalším obrázku můžeme vidět výsledek této operace.



Obr. 21: Sejmutí hodnoty jasu

Kurzor zde ukazuje na pixel v zobrazeném snímku v barevné škále úměrné hodnotám jasu. V tuto chvíli jsme schopni změřit jas na každém pixelu snímku, pokud je na daném pixelu hodnota L_{Lab} v požadovaném rozmezí a jas je tedy definován.

3.3.4 Přizpůsobení jasové mapy

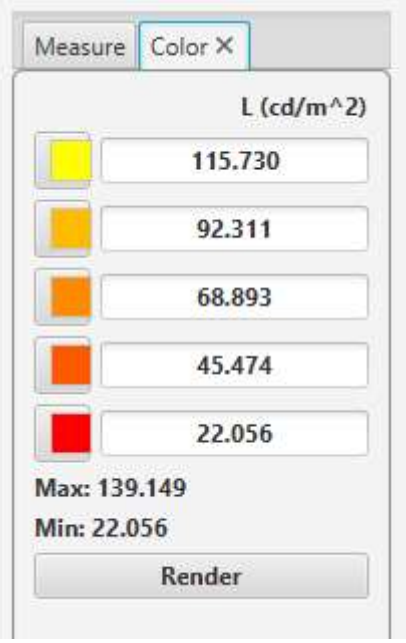
Jasová mapa je vytvářena metodami třídy *ColorMapper*, jak bylo zmíněno v sekci 3.2.5. Při nahrání fotografie je použita metoda, která byla v této sekci ukázána. Ta využívá defaultních barev a rozsah jasů je rozdělen na 5 stejně velkých intervalů.

Tato varianta může postačující, když do každého intervalu spadá podobné množství pixelů. Pokud však máme fotografii, kde 90 % hodnot tvoří nízké jasy spadající do jednoho intervalu a zbytek jasy vyšší, které rozdělíme mezi zbývající 4 intervaly, dostaneme prakticky jednobarevný snímek s několika málo místy jiné barvy. Tento problém můžeme vidět v přílohách 1 a 2.

Příloha 1 ukazuje jasovou mapu snímku s nízkým dynamickým rozsahem a lineárně rozdělenými intervaly. V této situaci jsou tyto intervaly postačující, jednotlivé barvy jsou zastoupeny poměrně rovnoměrně. V druhé příloze je případ HDR fotografie s lineárně rozdělenými intervaly, zde však drtivou většinu hodnot tvoří jasy v posledním intervalu, proto je na mapě vidět jasná převaha červené barvy. Příloha 3 ukazuje stejný snímek, v tomto případě s nelineárně rozdělenými intervaly. Přítomnost několika barevných hladin pro nejnižších cca 5 % jasů značně zvýšila čitelnost mapy.

Jednoduchým řešením tohoto problému je dát uživateli možnost si nastavit intervaly jasů a barvy dle vlastního uvážení. Pro implementování této funkcionality bylo rozděleno pravé menu na dvě záložky. V jedné z nich byly ponechány všechny funkce pro odečítání jasu, tato záložka byla pojmenována „Measure“. Druhá ze záložek obsahuje prvky pro výběr barvy a zadání hodnoty jasových intervalů a nese název „Color“.

Pro vytvoření záložky existuje v *JavaFX* třída *Tab*, která musí být umístěna ve speciálním kontejneru zvaném třídy *TabPane*. V třídě *Main* byly proto vytvořeny 3 instanční proměnné – dvě záložky třídy *Tab* a jeden *TabPane*. Výslednou podobu pravého menu lze vidět na dalším obrázku.



Obr. 22: Pravé menu s možností úpravy jasové mapy

V třídě *Main* existuje metoda *createColorVBox*, která se stará o sestavení všech prvků, které vidíme na záložce „Color“ na obr. 22.

```
private final Tab measure = new Tab("Measure");
private final Tab colors = new Tab("Color");
private TabPane tabPane = new TabPane();
private Button renderBtn = new Button("Render");
private Label maxL = new Label();
private Label minL = new Label();

private VBox createColorVBox(Color[] colors) {
    Label luminanceLabel = new Label("L (cd/m^2)");
    VBox colorVbox = new VBox();
    colorVbox.getChildren().add(luminanceLabel);

    for (int i = 0; i < colors.length; i++) {
        HBox row = new HBox();

        ColorPicker picker = new ColorPicker(colors[colors.length - 1 - i]);
        TextField luminanceThreshhold = new TextField();

        row.getChildren().addAll(picker, luminanceThreshhold);
        colorVbox.getChildren().add(row);
    }

    renderBtn.setOnAction(e -> {
        List<TextField> textFields = extractLuminanceThreshholdTextFields();
        double[] threshholds = new double[textFields.size()];

        for (int i = 0; i < textFields.size(); i++) {
            String text = textFields.get(i).getText();

            if (isValidDouble(text)) {
                threshholds[i] = Double.parseDouble(text);
            } else {
                alertInvalidNumber();
                break;
            }
        }
        if (!validateThreshHoldValues(threshholds)) {
            alertInvalidValues();
        } else {
            java.awt.Color[] extractedColors = extractSelectedColors();

            reconstructedImage.setLuminanceThreshholds(threshholds);
            reconstructedImage.setColors(extractedColors);
            reconstructedImage =
                    ColorMapper.reconstructImage(displayedImage, reconstructedImage);
            imv.setImage(SwingFXUtils.toFXImage(reconstructedImage, null));
        }
    });

    colorVbox.getChildren().addAll(maxL, minL, renderBtn);
    return colorVbox;
}
```

Fragment kódu 23: Třída Main – vytvoření menu jasové mapy

Fragment kódu 23 ukazuje definování objektů, které tvoří záložku „Color“. Všechny tyto objekty jsou uspořádány vertikálně pomocí kontejneru *VBox*. Pro výběr barvy slouží objekt typu *ColorPicker*. Kliknutím na barevný obdélník se otevře okno s paletou barev. První *for* smyčka v metodě *createColorVBox* vytvoří pro každou barvu *HBox*, který obsahuje objekt *ColorPicker* a *TextField*. Zmíněný *TextField* slouží pro zobrazení a zadání hodnoty jasu, která je spodní hranicí intervalu pro danou barvu.

Tlačítko „Render“ slouží k potvrzení zadaných hodnot jasu a následnému vykreslení mapy v daných barvách a se zadanými intervaly. Pomocí metody *setOnAction* je mu opět nastaven event handler tvořený lambda výrazem. Tento event handler se postará o přečtení hodnot z textových polí a jejich validaci pomocí k tomu určených metod. Pokud hodnoty projdou validací, jsou ještě uloženy hodnoty nastavených barev a obě tato pole jsou předána aktuální instanci třídy *ReconstructedImage*, která je právě zobrazována. Poté je do této instance také uložen výsledek metody *reconstructImage* (její druhé varianty, nikoli té v kap. 3.2.5). Výsledná jasová mapa je zobrazena v *ImageView*.

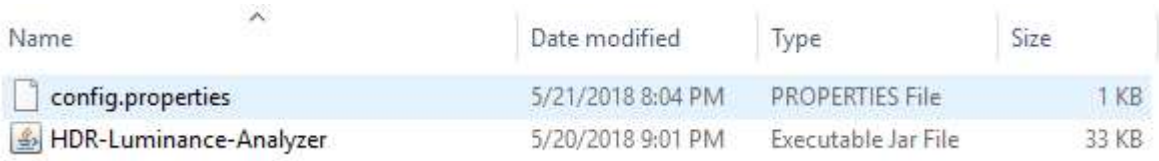
3.4 Instalace programu

V této sekci bude popsáno, jak program spustit na jakémkoliv počítači.

Všechny zdrojové soubory Java aplikace jsou standardně zabaleny do souboru s příponou „.jar“. Tato přípona není tak známá, jako např. „.exe“ soubory u nativních aplikací pro systém Windows, avšak funkci plní totožnou. Pro spuštění aplikace je na daném operačním systému nutné mít nainstalova *Java SE Runtime Environment* (JRE) verze 9 a vyšší. V době publikování této práce je aktuální verzí dostupnou ke stažení na stránkách společnosti Oracle JRE 10.0.1.

Instalační soubor JRE lze stáhnout pro všechny dnes běžně užívané operační systémy Linux, MacOS a Windows. Otevřením instalačního souboru se spustí jednoduchý installer, který ničím nepřekvapí žádného uživatele. Po instalaci by již mělo být možné dvojitým kliknutím na „.jar“ soubor aplikaci spustit.

Jedinou podmínkou chodu aplikace je přítomnost souboru „config.properties“ ve stejném adresáři, jako se nachází „.jar“ soubor. Tento soubor obsahuje koeficienty výpočtu a aplikace k němu proto potřebuje přístup.



Obr. 23: Adresář aplikace s konfiguračním souborem

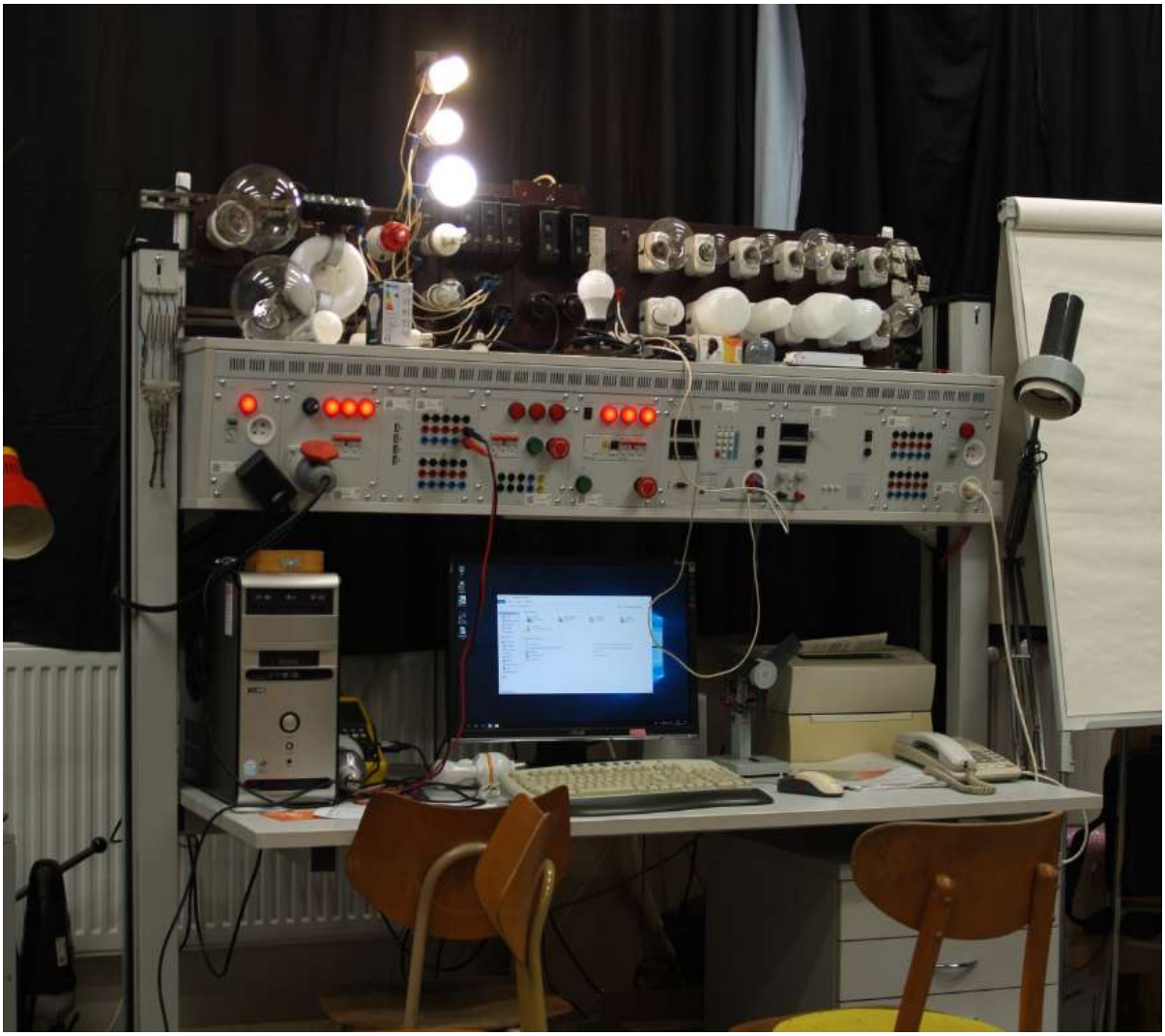
Java aplikace lze také spouštět přes příkazový řádek. Nejprve je nutné se dostat do složky, kde se nachází „.jar“ soubor (na systémech Windows i Linux příkazem „*cd ADRESA_SLOŽKY*“) a poté příkazem „*java -jar NÁZEV_SOUBORU.jar*“ aplikaci spustíme.

3.5 Verifikace programu

Tato kapitola se bude zabývat dokumentací postupu při ověření správnosti funkce softwaru a srovnání výsledků s výsledky naměřenými.

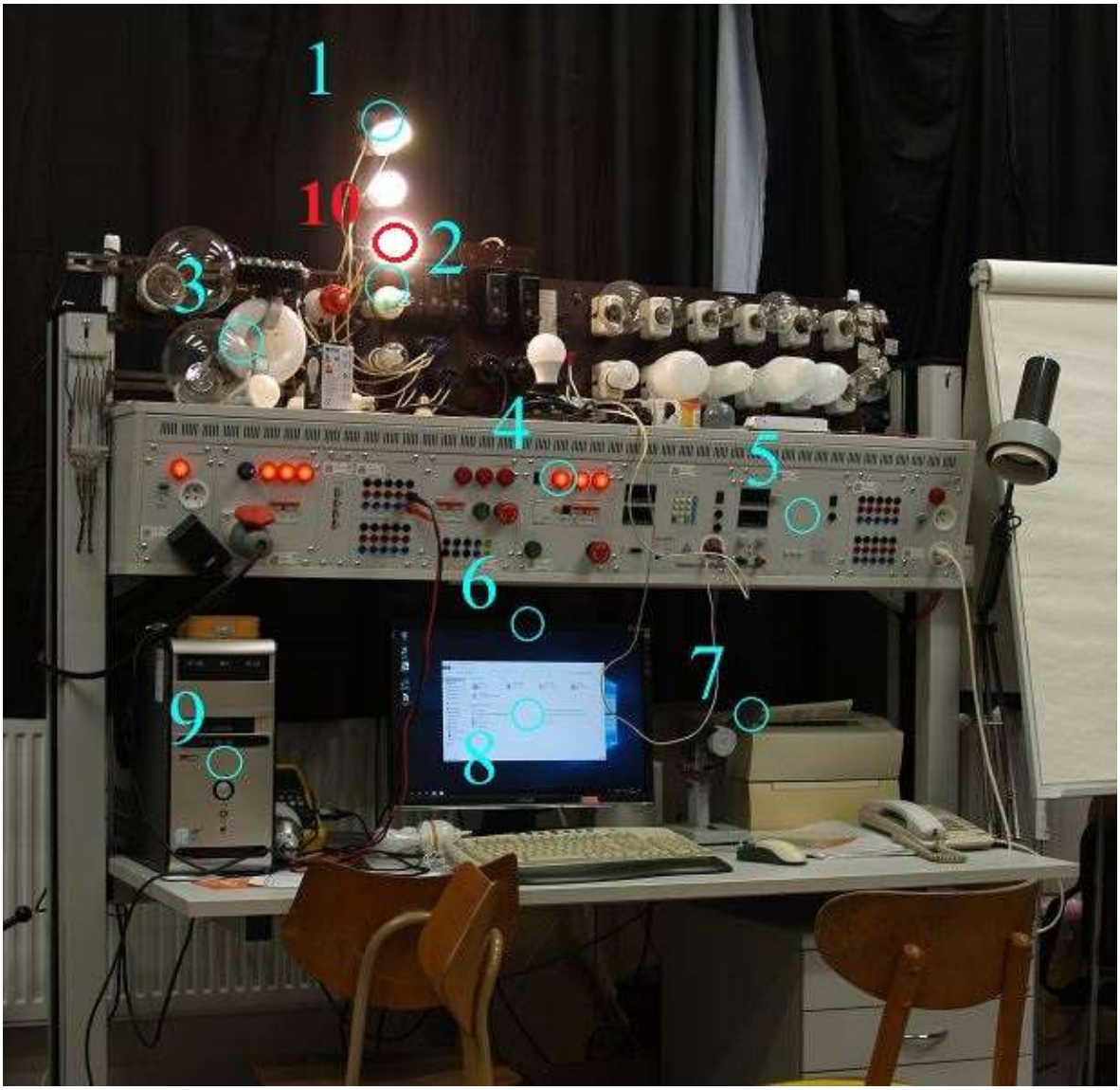
3.5.1 Průběh měření

Pro měření bylo nutné vybrat scénu s velkým rozsahem jasů, abychom dobře prověřili dynamický rozsah fotoaparátu při měření jasu. Jako tuto scénu jsme vybrali pracoviště v laboratoři světelné techniky na katedře elektroenergetiky. Tuto scénu můžeme vidět na následující fotografii.



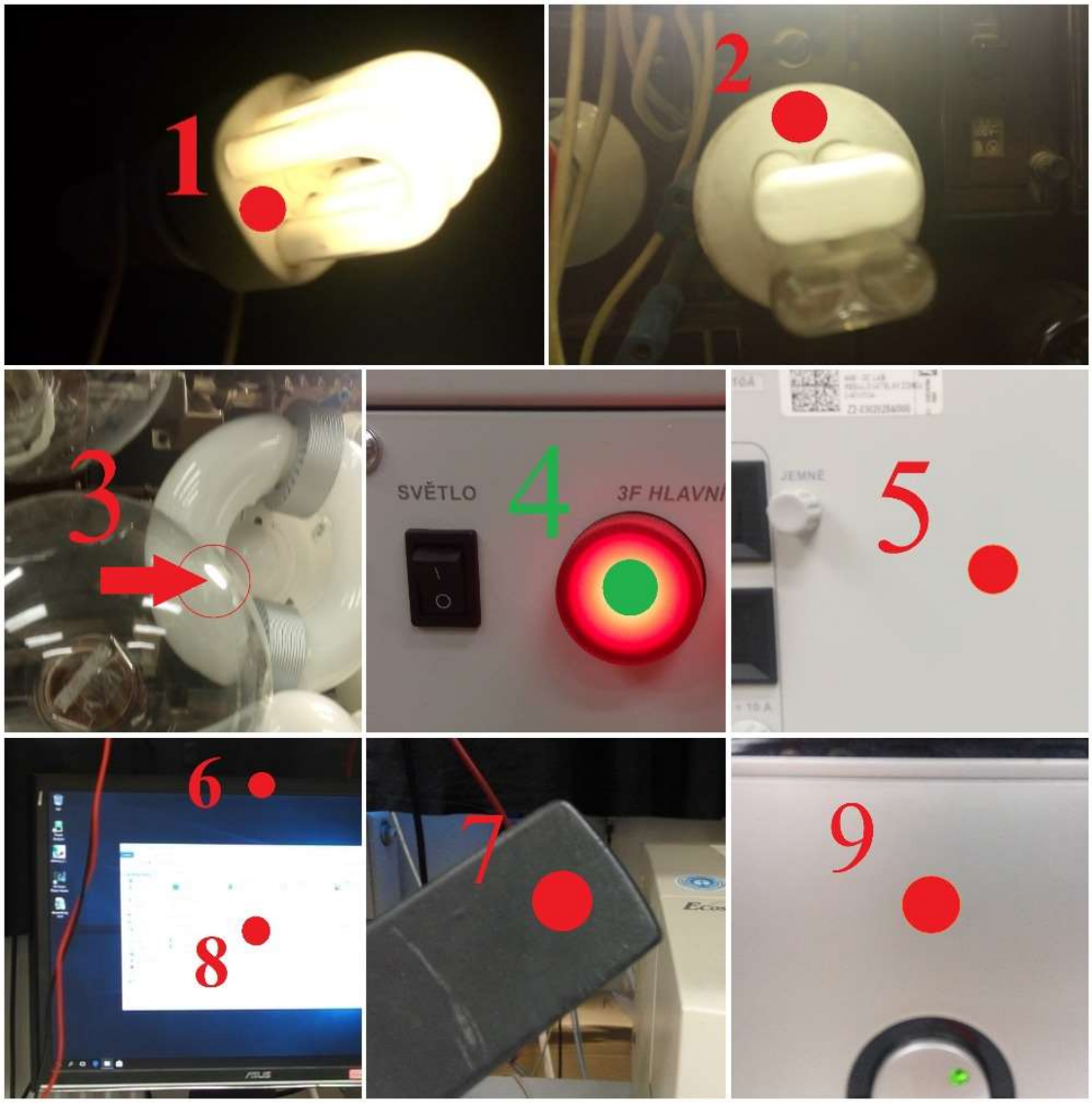
Obr. 21: Fotografovaná scéna

Na této scéně jsme vybrali 10 měřících bodů tak, abychom dobře pokryli celou škálu jasů, které se zde nachází. Tyto body jsou vyznačené na následující fotografii.



Obr. 22: Měřící body

Nedostatkem tohoto vyznačení je příliš velká plocha, která je zakroužkována. V těchto plochách je jas hodně proměnlivý, snad až na č. 10. Body byly měřeny s mnohem menším zorným úhlem, proto bylo nutné ještě k většině bodů vytvořit detailní záběr. Koláž těchto detailních záběrů najdeme na další straně.



Obr. 23: Detail měřených bodů

Samotné měření sestávalo z dvou částí. V první části jsme s jasoměrem *LMT L 1009*, umístěným na stativu, měřili různé body na scéně, abychom dobře pokryli jasový rozsah scény. Nakonec bylo vybráno zmíněných 10 bodů. Naměřené hodnoty jasu v těchto bodech obsahuje následující tabulka.

Tabulka 2: Změřené hodnoty jasu

Číslo vzorku	1	2	3	4	5	6	7	8	9	10
Jas (cd/m^2)	9700	434	304	182,5	51	3,4	7,9	135,6	33,7	42000

Druhou částí měření bylo pořízení snímků fotoaparátem *Fujifilm Finepix S2 Pro*. Tento fotoaparát je stejným fotoaparátem použitým v [1], který byl zkalibrován pro měření jasu. Důležité bylo zachovat stejnou polohu fotoaparátu a jasoměru, stativ byl tedy ponechán na stejné pozici a jasoměr byl pouze vyměněn za fotoaparát. Pro eliminaci vlivu otřesů jsme použili samospoušť.

Bylo pořízeno celkem 21 fotografií této scény s clonovým číslem 4. Čas expozice byl pro každý snímek jiný, nejkratší čas byl 1/1500 *s*, nejdelší 1/1,5 *s*. Tyto snímky bylo ještě potřeba převést z formátu RAF do TIFF.

3.5.2 Měření jasů pomocí aplikace

Nyní máme k dispozici 21 snímků, které můžeme otevřít v naší aplikaci a odečíst jednotlivé jasy, pokud jsou na daném pixelu definovány. Ukažme si v následující tabulce, jak vypadají hodnoty odečtené z jednotlivých fotografií, na posledním řádku jsou uvedeny změřené hodnoty pro porovnání. Všechny hodnoty u každého bodu byly vždy odečítány na stejném pixelu. Červená políčka značí neplatné hodnoty, tedy pixely, u kterých nebyla hodnota L_{Lab} v požadovaném rozsahu.

Tabulka 3: Hodnoty jasu vypočtené programem

Číslo fotografie	**t (s)**	**Číslo bodu**									
		1	2	3	4	5	6	7	8	9	10
		Jas (cd/m^2)									
1	1/15					42,31				27,62	
2	1/20				117,78	43,00			113,12	29,84	
3	1/30				139,30	49,38			125,02	34,87	
4	1/45				178,91	56,02			134,46		
5	1/60			149,55	196,15				125,68		
6	1/90		473,94	206,21	284,39				140,33		
7	1/125		468,72	198,49	390,18				148,64		
8	1/180		470,39	224,53	556,89						
9	1/250		486,73		400,78						
10	1/350		501,29		534,54						
11	1/500		563,81		586,14						
12	1/750										
13	1/1000	5637,22									
14	1/1500	6251,71									
16	1/10					40,20				27,56	
17	1/8					44,81		8,97		29,77	
18	1/6							8,15		30,58	
19	1/4							7,46			
20	1/3						4,02	7,30			
21	1/2						3,31	6,97			
22	1/1,5						3,34	7,10			
Změřená hodnota →		9700	434	304	182,5	51	3,4	7,9	135,6	33,7	42000

Jako první si všimneme velkého počtu červených polí, to ukazuje, jak malý dynamický rozsah má samotný fotoaparát. V těchto místech byla daná fotografie přeexponovaná, nebo podexponovaná. Pozice č. 10 nebyla dle očekávání zachycena ani na jednom snímku, takto vysoký jas je s fotoaparátem velmi obtížně zachytit. Stejně tak bod 1, který sice zachycen byl, ale jen na dvou fotografiích, a to velmi nepřesně.

Rychlou analýzou naměřených hodnot můžeme také usoudit, že hodnoty jasu do 500 cd/m^2 jsme změřili relativně přesně, s výjimkou dvou bodů. Jedním z nich je bod č. 3, zde se v nejlepším případě

odchylujeme od změřené hodnoty přibližně o 26 %. To můžeme odůvodnit velkou proměnlivostí jasu v tomto místě, která se projevila tím, že dva sousední pixely mají rozdílné hodnoty i o 200 cd/m^2. Uvedené jasy v bodě 3 jsou změřeny na pozici [1308, 841], kde byla na fotografii 6 změřena hodnota 206 cd/m^2, avšak pokud se posuneme o pixel vlevo na [1307, 841], jsme na hodnotě 392 cd/m^2. Když si uvědomíme, že zorné pole jasoměru při měření mělo záběr minimálně několik desítek pixelů, dojdeme k původu chyby. Jasoměr vyrobil průměrnou hodnotu jasu v jeho zorném úhlu, kdežto my jsme jeho zorný úhel rozdělili na jednotlivé pixely, které mají různé hodnoty jasu.

Na obr. 24 vidíme měřené místo v barevné škále odpovídající jasu. Podexponovaná místa jsou zbarvena černě, přeexponovaná bíle a rozsah změřených jasů je rozdělen lineárně na 5 intervalů. Žlutá barva značí interval nejvyšších jasů, červená zase nejnižších 20 %.

Obr. 24: Gradient jasu v bodě č. 3

Druhým zvláštním úkazem je měřící bod č. 4. Zde jsme naměřili širokou škálu jasů při různých expozicích. Toto chování je zřejmě ovlivněno optickými vlastnostmi krytu, který je umístěn na kontrolce. Přesněji si však tento problém nedokážeme vysvětlit.

Ze zbylých kontrolních bodů, které nám poskytly uspokojující výsledky můžeme vyhodnotit relativní chyby a také jejich závislost na hodnotě L_{Lab}. Tabulka 4 obsahuje výčet těchto hodnot. Sloupce L_M a L_C představují změřené, respektive vypočtené hodnoty jasu, $\Delta L_{\%}$ je relativní chyba jasu vůči měřené hodnotě a ΔL_{Lab50} je vzdálenost příslušné hodnoty L_{Lab} od 50.

Červeně zvýrazněné řádky jsou ty, kde je hodnota L_{Lab} nejblíže 50 z daného souboru hodnot v měřeném bodě, tedy kde ΔL_{Lab50} je nejblíže 0. Tuto hodnotu považujeme za nejpřesnější, jelikož se nachází ve středu lineární části převodní charakteristiky. Podle této podmínky se také vybírá hodnota jasu daného pixelu při slučování více fotografií do HDR snímku.

Když se nyní podíváme na tyto hodnoty v tabulce, zjistíme, že ne vždy má hodnota, kterou považujeme za nejpřesnější, nejmenší chybu. Náš předpoklad byl splněn v měřících bodech 8 a 6, s přimhouřením oka i v bodě 2. V bodě 9 byla však chyba námi vybrané hodnoty ze všech nejvyšší a hodnota s nejmenší chybou se již blížila hranici $L_{Lab} = 20$.

Takovéto výkyvy můžeme přisoudit jak chybě měření jasoměru, tak i nekonzistenci senzoru fotoaparátu, či variaci světelných vlastností scény mezi měřením jasoměrem a pořizování jednotlivých snímků. Dalším faktorem může být nepřesná kalibrace.

Tabulka 4: Relativní chyby naměřených hodnot

Měřící bod	L_M (cd/m2)	L_C (cd/m2)	$\Delta L_{\%}$	L_{Lab}	ΔL_{Lab50}
2	434	473,94	9,20%	70,93	20,93
		468,72	8,00%	59,87	9,87
		470,39	8,39%	48,10	1,90
		486,73	12,15%	38,52	11,49
		501,29	15,50%	28,52	21,49
		563,81	29,91%	21,49	28,51
5	51	42,31	17,04%	50,59	0,59
		43,00	15,68%	41,75	8,25
		49,38	3,17%	33,05	16,95
		56,02	9,84%	23,95	26,05
		40,20	21,17%	62,13	12,13
		44,81	12,15%	72,93	22,93
6	3,4	4,02	18,26%	26,36	23,64
		3,31	2,68%	33,21	16,79
		3,34	1,76%	42,89	7,11
7	7,9	8,97	13,52%	20,53	29,47
		8,15	3,20%	26,80	23,20
		7,46	5,54%	37,12	12,88
		7,30	7,57%	45,79	4,21
		6,97	11,81%	57,46	7,46
		7,10	10,15%	67,45	17,45
8	135,6	113,12	16,58%	73,25	23,25
		125,02	7,80%	63,31	13,31
		134,46	0,84%	52,47	2,47
		125,68	7,32%	40,90	9,10
		140,33	3,49%	31,28	18,72
		148,64	9,62%	22,46	27,55
9	33,7	27,62	18,04%	36,70	13,30
		29,84	11,44%	29,85	20,15
		34,87	3,48%	21,72	28,28
		27,56	18,23%	49,83	0,17
		29,77	11,66%	59,62	9,62
		30,58	9,27%	69,86	19,86

Závěr

Předchozí kapitola ukázala, že dokážeme měřit nízké a středně vysoké jasy s přesností do 20 %. To je přesnost postačující pro orientační měření, kdy potřebujeme analyzovat jasové poměry na celé scéně. Výměnou za přesnost konvenčních jasoměrů získáváme rychlost měření a to, že jsme schopni po vyfocení fotografie zjistit jas každého pixelu.

Schopnost zjistit jas v každém pixelu je obrovskou výhodou oproti bodovým jasoměrům, u kterých jsme nuceni při analyzování celé scény měřit síť bodů. Tato síť může buď obsahovat velké množství bodů, jejichž měření zabere mnoho času, anebo může být řidší a méně časově náročná na proměření. Technologie jasové analýzy pomocí digitální fotografie nemusí v tomto smyslu dělat žádné kompromisy, jediným stisknutím spouště máme k dispozici síť o několika milionech měřících bodů v podobě pixelů.

Hlavní aplikací této metody prozatím určitě nejsou přesná laboratorní měření, či přímá měření jasu světelných zdrojů. Využití však může najít při analyzování jasových poměrů interiérů či venkovních scén, kde se nejčastěji vyskytují jasy o hodnotách do 500 cd/m^2. Měření vyšších jasů by bylo teoreticky možné, kdybychom využili ještě nižší nastavení expozice fotoaparátu. Při našem měření jsme se drželi konstantního clonového čísla 4, které lze však ještě snížit.

Vytvořená softwarová aplikace splňuje naše očekávání ohledně rychlosti výpočtu. Výpočetní výkon je samozřejmě dán hardwarem počítače, na kterém aplikace běží. Zpracování fotografie do jasové matice trvá mnohem kratší dobu než samotné nahrání fotografie a zadání parametrů expozice, řádově jsou to desetiny až jednotky sekund. Při kombinování fotografií pro HDR jsou to také jednotky sekund.

Vzhledem k tomu, že jsou fotografie nahrávány do paměti, je aplikace poměrně náročná na využití RAM. Jako opatření proti problémům s nedostatkem paměti je dobré fotografie převádět např. do formátu JPEG. Sice tím snížíme kvalitu fotografie, ale na hodnoty jasu to bude mít minimální vliv a velikost souboru se výrazně sníží.

Grafické rozhraní je jednoduché a odečítání hodnot jasu pouhým přejetím kurzoru po pixelu je intuitivní. Stejně tak označení fotografie pomocí zaškrtávacího políčka pro její zobrazení, či pro sloučení více fotografií. Díky možnosti úpravy koeficientů výpočtu v grafickém rozhraní je aplikace částečně odolná vůči potenciálním změnám v kalibraci.

Existují ještě určité funkce, které by aplikace mohla mít a které nebyly implementovány v rámci této práce. Například by bylo vhodné přidat možnost označení plochy v poli fotografie za účelem vypočtení průměrného jasu na této ploše. Další potenciální funkcí by mohl být export vybraných hodnot jasu do textového souboru, např. CSV.

Tvorba aplikace byla v kapitole 3 rozebrána poměrně detailně z toho důvodu, aby se na ni dalo snadno navázat, např. by v rámci dalšího výzkumu této problematiky na katedře elektroenergetiky mohlo dojít k nutnosti upravení či rozšíření aplikace.

Vylepšením metod kalibrace a navázáním spolupráce s katedrami zabývajícími se softwarem v rámci fakulty by mohlo vést ke zdokonalení aplikace do podoby, kdy by se mohla využívat ke komerčním účelům.

Literatura

[1] FIŠERA, M. *Digitální fotografie a zorné pole lidského oka*. Praha, 2005. Bakalářská práce. ČVUT FEL.

[2] HABEL, Jiří. *Světlo a osvětlování*. Praha: FCC Public, 2013. ISBN 978-80-86534-21-3.

[3] KONICA MINOLTA. LS-150 Luminance Meter. *Konica Minolta* [online]. [cit. 2018-05-21]. Dostupné z: https://sensing.konicaminolta.us/products/ls-150-luminance-meter/

[4] Mezinárodní doporučení CIE: Illuminance Meters and Luminance Meters. 1983.

[5] C-M. *Ccd-sensor.jpg* [online]. 23. 5. 2007 [cit. 2018-05-21]. Dostupné z: https://commons.wikimedia.org/w/index.php?curid=2150801

[6] MOHYLEK. *Apertures.jpg* [online]. 17. 8. 2006 [cit. 2018-05-21]. Dostupné z: https://commons.wikimedia.org/w/index.php?curid=1065235

[7] JÓZEFOWICZ, Remigiusz. *MigawkaSzczelinowa.jpg* [online]. 15. 4. 2007 [cit. 2018-05-21]. Dostupné z: https://commons.wikimedia.org/w/index.php?curid=1065235

[8] FULTON, Wayne. *EV - Exposure Value* [online]. [cit. 2018-05-21]. Dostupné z: https://www.scantips.com/lights/evchart.html

[9] STOKES, Michael, Matthew ANDERSON, Srinivasan CHANDRASEKAR a Ricardo MOTTA. *A Standard Default Color Space for the Internet - sRGB* [online]. 5. 11. 1996 [cit. 2018-05-21]. Dostupné z: https://www.w3.org/Graphics/Color/sRGB.html

[10] PIHAN, Roman. *Dynamický Rozsah (Kontrast) Scény* [online]. 2013 [cit. 2018-05-21]. Dostupné z: http://www.fotoroman.cz/tech1/exposure_contrast.htm

[11] ADOBE SYSTEMS INCORPORATED. *CIE Lab* [online]. 2000 [cit. 2018-05-21]. Dostupné z: http://dba.med.sc.edu/price/irf/Adobe_tg/models/cielab.html

[12] BÁLSKÝ, Marek. *Luminance values extraction from digital images*. 2016. ČVUT FEL.

[13] RYBÁŘ, Jan. *Co to je HDR fotografie* [online]. 8. 11. 2016 [cit. 2018-05-21]. Dostupné z: https://www.fotoguru.cz/co-to-je-hdr/

[14] JENKOV, Jakob. *JavaFX Tutorial* [online]. 16. 3. 2018 [cit. 2018-05-21]. Dostupné z: http://tutorials.jenkov.com/javafx/index.html

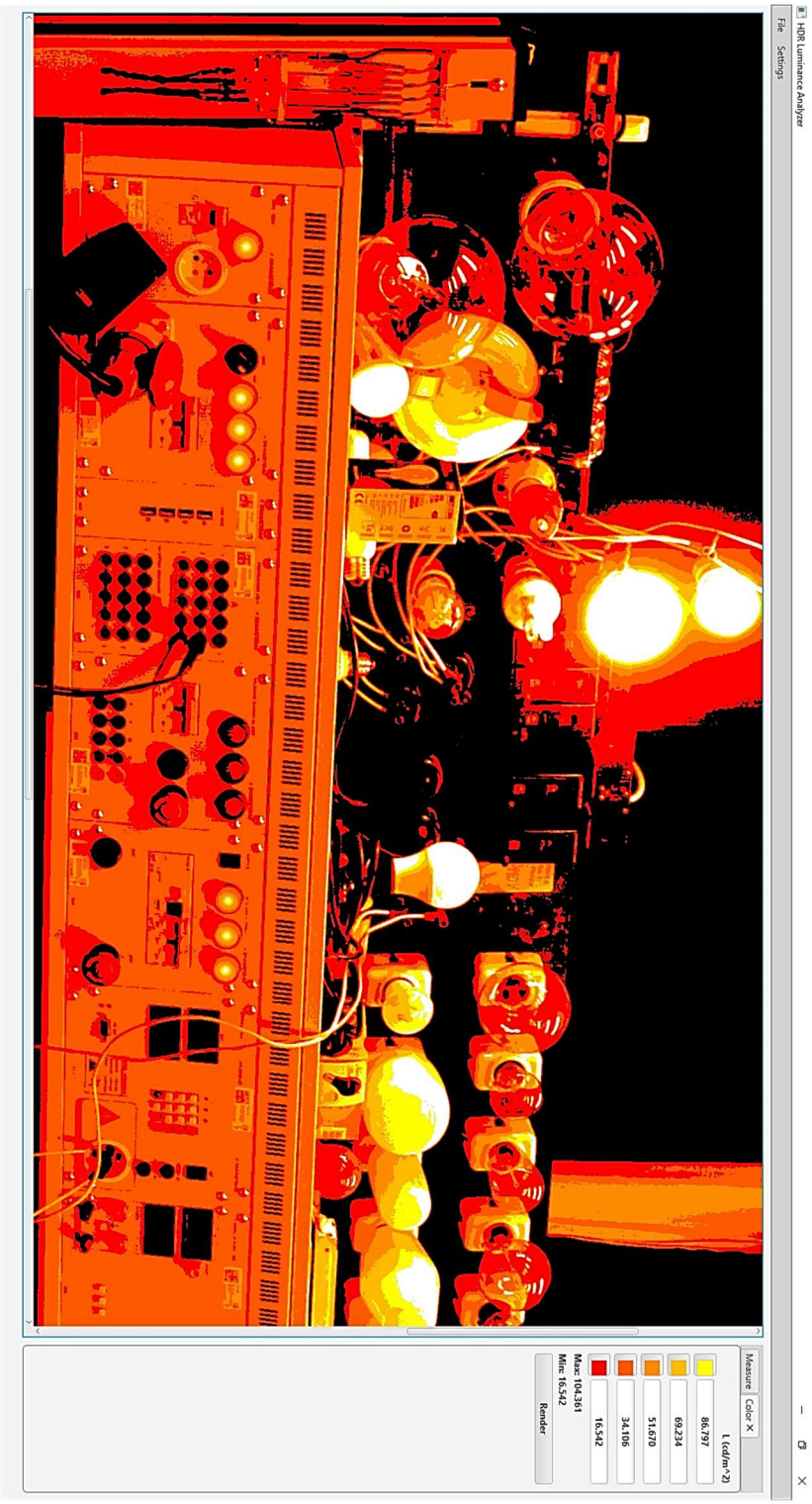
[15] *ČSN EN 13032: Světlo a osvětlení - Měření a uvádění fotometrických údajů světelných zdrojů a svítidel*. 2012.

[16] *Lighting Design Glossary* [online]. [cit. 2018-05-21]. Dostupné z: https://www.schorsch.com/en/kbase/glossary/luminance.html

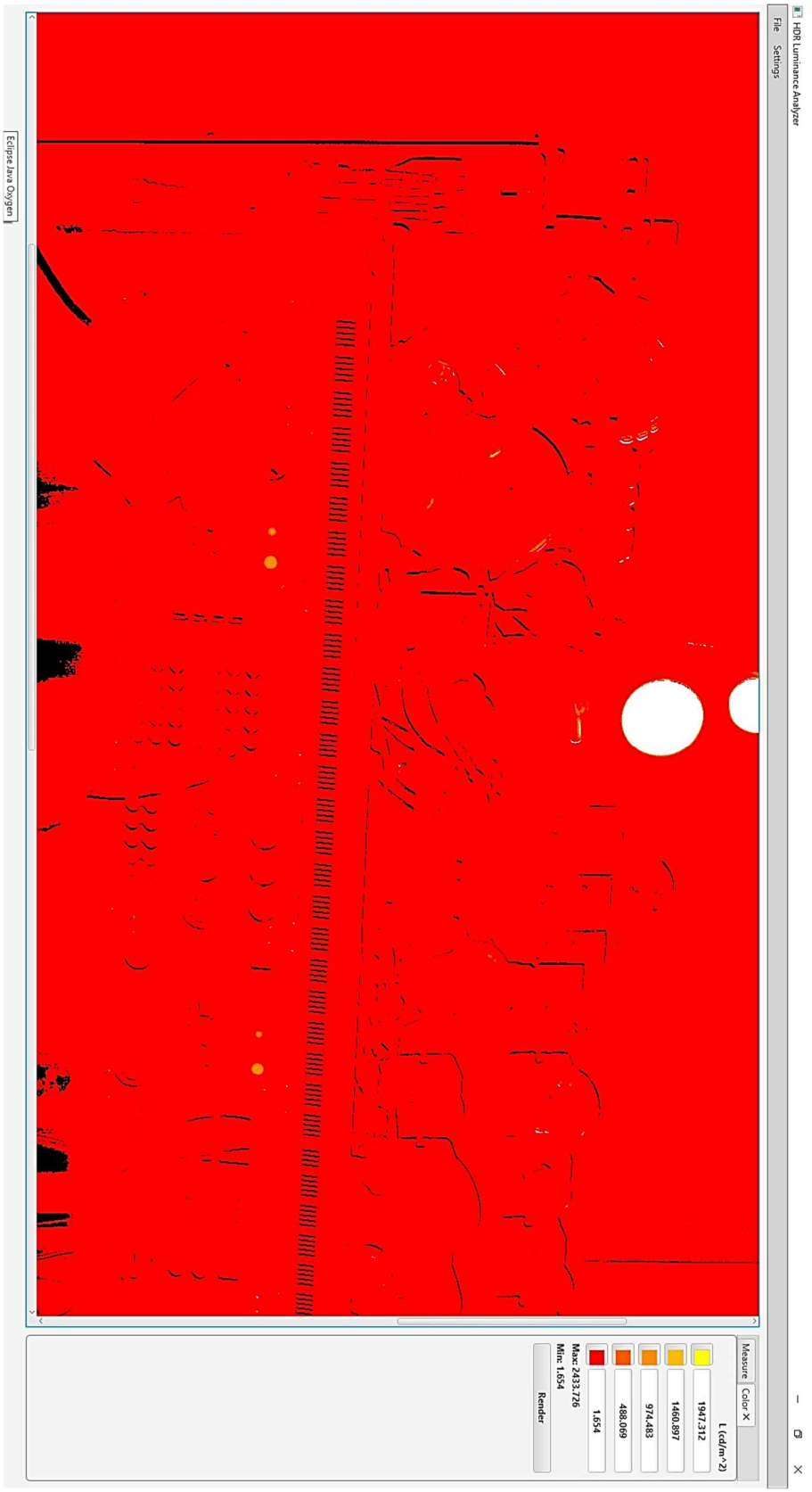
[17] JCC2011. *Etendue-Free space.png* [online]. [cit. 2018-05-21]. Dostupné z: https://commons.wikimedia.org/wiki/File:Etendue-Free_space.png

Přílohy

Příloha 1 – Jasová mapa fotografie s malým dynamickým rozsahem a lineárně rozdělenými intervaly



Příloha 2 – Jasová mapa HDR fotografie s lineárně rozdělenými intervaly



Příloha 3 – Jasová mapa HDR fotografie s nelineárně rozdělenými intervaly