
**VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ**
**BRNO UNIVERSITY OF TECHNOLOGY**

**FAKULTA INFORMAČNÍCH TECHNOLOGIÍ**
**FACULTY OF INFORMATION TECHNOLOGY**

**ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ**
**DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA**


# POŘIZOVÁNÍ VYSOCE KVALITNÍCH SNÍMKŮ ROVINNÝCH POVRCHŮ CHYTRÝM TELEFONEM

**CAPTURING VERY HIGH QUALITY IMAGES OF PLANAR SURFACES BY A SMARTPHONE**

**DIPLOMOVÁ PRÁCE**
**MASTER'S THESIS**


**AUTOR PRÁCE** **Bc. ADAM MASARYK**
**AUTHOR**

**VEDOUCÍ PRÁCE** **prof. Ing. ADAM HEROUT, Ph.D.**
**SUPERVISOR**



**BRNO 2021**

**Vysoké učení technické v Brně**
Fakulta informačních technologií

Ústav počítačové grafiky a multimédií (UPGM) Akademický rok 2020/2021

# Zadání diplomové práce


Student: **Masaryk Adam, Bc.**
Program: Informační technologie Obor: Vývoj aplikací
Název: **Pořizování vysoce kvalitních snímků rovinných povrchů chytrým telefonem**
**Capturing Very High Quality Images of Planar Surfaces by a Smartphone**
Kategorie: Zpracování obrazu

Zadání:

1. Nastudujte základy tvorby mobilních aplikací, zaměřte se na pořizování obrazu vestavěnou kamerou.
2. Nastudujte problematiku zpracování obrazu, zaměřte se na registraci/zarovnání rovinných obrázků a fúzi obrázků do obrázku s vysokým rozlišením (superresolution).
3. Vytvořte jednoduchou aplikaci pro smartphone, která umožní pořizovat obrázky do datové sady pro vývoj algoritmů.
4. Pořiďte datovou sadu pro vývoj algoritmů; průběžně ji rozšiřujte.
5. Experimentujte s algoritmy zpracování obrazu nad pořízenými daty, vyviňte použitelné (optimální) řešení.
6. Integrujte vyvinuté algoritmy do mobilní aplikace pro pořizování vysoce kvalitních obrázků.
7. Iterativně vylepšujte vytvořenou aplikaci pro dosažení dobré uživatelské použitelnosti a pro maximální kvalitu výstupů.
8. Zhodnoťte dosažené výsledky a navrhněte možnosti pokračování projektu; vytvořte plakátek a krátké video pro prezentování projektu.

Literatura:

- Gary Bradski, Adrian Kaehler: Learning OpenCV; Computer Vision with the OpenCV Library, O'Reilly Media, 2008
- Richard Szeliski: Computer Vision: Algorithms and Applications, Springer, 2011
- Android Developers: https://developer.android.com/index.html
- Steve Krug: Don't Make Me Think, Revisited: A Common Sense Approach to Web Usability, ISBN: 978-0321965516
- Steve Krug: Rocket Surgery Made Easy: The Do-It-Yourself Guide to Finding and Fixing Usability, ISBN: 978-0321657299

Při obhajobě semestrální části projektu je požadováno:

- Body 1 až 3, značné rozpracování bodů 4 a 5.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Herout Adam, prof. Ing., Ph.D.**
Vedoucí ústavu: Černocký Jan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2020
Datum odevzdání: 19. května 2021
Datum schválení: 30. října 2020

# Abstrakt

Cieľom tejto práce je vytvoriť mobilnú aplikáciu pre systém Android, ktorá užívateľom umožní vytvárať vysoko kvalitné fotografie rovinných predmetov. V aplikácii si užívateľ vytvorí viacero fotografií vybraného rovinného predmetu. Tieto fotografie sú následne zarovnané a potom spojené do jedného výsledného obrázku, pričom pri tomto spojení dochádza k odfiltrovaniu rôznych nedostatkov, ktoré sa môžu vo fotografiách nachádzať.



# Abstract

The aim of this thesis is to create a mobile application for Android, which allows users to create high-quality photos of planar objects. User can create multiple photographs of a selected planar object. These photographs are then aligned and combined into one final image. Various shortcomings that can be present in the photographs are filtered.



# Kľúčové slová

Android, SIFT, SURF, ORB, RANSAC, FLANN, BF-Matcher, OpenCV, fotografie, kamera, spájanie fotografií, významné body, deskriptory, MVVM, mobilná aplikácia, YUV, užívateľské rozhranie

# Keywords

Android, SIFT, SURF, ORB, RANSAC, FLANN, BF-Matcher, OpenCV, photography, camera, photo merging, key-points, descriptors, MVVM, mobile application, YUV, user interface


# Citácia

MASARYK, Adam. *Pořizování vysoce kvalitních snímků rovinných povrchů chytrým telefonem*. Brno, 2021. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce prof. Ing. Adam Herout, Ph.D.

# Pořizování vysoce kvalitních snímků rovinných povrchů chytrým telefonem

## Prehlásenie

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana prof. Ing. Adama Herouta, Ph.D. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Adam Masaryk
18. mája 2021


## Poďakovanie

Chcel by som sa poďakovať vedúcemu práce pánovi prof. Ing. Adamu Heroutovi, Ph.D., ktorý mi pri vytváraní práce bol vždy nápomocný a dával cenné rady.

# Obsah

# Kapitola 1

# Úvod

V dnešnej dobe patria mobilné zariadenia k neodmysliteľnej súčasti našich životov. Je pomerne rarita nájsť človeka, ktorý takéto zariadenie nevlastní a nevyužíva. Každé takéto zariadenie je vybavené mnohými komponentmi, medzi ktoré patrí aj kamera. Vďaka nej môžu ľudia zachytávať rôzne scény vo vysokej kvalite. Táto kvalita ale nemusí byť v niektorých prípadoch dostatočná, pretože jedna fotografia sa nemusí vydariť, ďalšia môže mať rozmazanú určitú časť alebo obsahuje rôzne nedostatky ako napríklad rôzne odrazy.

Mobilná aplikácia pre platformu Android, ktorej proces tvorby je opísaný v tomto texte, sa snaží riešiť problém, kedy užívateľ potrebuje vytvoriť fotografiu nejakého objektu v čo najvyššej kvalite bez rôznych nedostatkov tak, že objekt je odfotografovaný viac krát a vhodným spracovaním je z týchto vstupných fotografií vytvorená jedna kvalitná výsledná fotografia.

Na vytvorenie takejto aplikácie je potrebné najprv pochopiť fungovanie vybranej platformy a preto v kapitole 2 je opísaná samotná platforma, spôsob vývoja na danej platforme a možnosti vytvárania fotografií.

Keďže v aplikácií dochádza k spracovávaniu vytvorených fotografií, je nutné pochopiť rôzne prístupy spracovania. V kapitole 3 a 4 sa venujem samotnému spracovaniu obrázkov a metódam na ich zarovnávanie, ktoré je tiež súčasťou spracovania.

V kapitole 5 sa venujem už samotnému návrhu a testovaniu procesu, ktorý zo vstupných fotografií vytvorí jednu kvalitnú fotografiu. V tejto časti sa využívajú hlavne poznatky z predchádzajúcich kapitol na spracovanie a zarovnávanie obrázkov.

Po navrhnutí časti na spracovanie nasleduje časť v kapitole 6, ktorá sa venuje návrhu a testovaniu samotnej mobilnej aplikácie, ktorá na spracovanie využíva navrhnutý proces z predchádzajúcej kapitoly. V tejto časti bolo potrebné navrhnúť štruktúru samotnej aplikácie a hlavne iteratívne navrhnúť a otestovať užívateľské rozhranie.

V kapitole 7 sú spomenuté rôzne implementačné detaily jednotlivých častí mobilnej aplikácie, ktoré bolo potrebné vykonať na dosiahnutie očakávaného výsledku.

# Kapitola 2

# Platforma Android

Android je operačný systém založený na Linuxovom jadre, hlavne určený pre telefóny s dotykovou obrazovkou a tablety, no v dnešnej dobe je rozšírený aj v oblasti hodiniek, televízorov a dokonca aj ako systém v nových autách. Jedná sa o open-source systém, ktorý je vyvíjaný firmou Google. Síce samotný zdrojový kód je open-source, no v komerčných zariadeniach sa väčšinou nachádzajú predinštalované proprietárne aplikácie ako napríklad **Google Mobile Services** alebo **Google Play**. Práve táto otvorenosť je zároveň veľkou výhodou aj nevýhodou. Vďaka tomuto si môžu jednotliví výrobcovia upraviť systém podľa vlastných predstáv tak, že konečný užívateľ ani nemusí spoznať, že dané zariadenie beží na Androide. Práve tieto úpravy systému spôsobujú ťažší vývoj aplikácii, pretože sa daná aplikácia môže správať trochu inak, prípadne vôbec nefungovať na telefónoch inej značky. Najčastejšie tento problém nastáva pri lacných čínskych telefónoch.

História systému sa datuje do roku 2003, kedy *Andy Rubin*, *Rich Miner*, *Nick Sears* a *Chris White* založili firmu **Android Inc.** [1], ktorej prvotnej cieľom bolo vytvoriť operačný systém pre kamery. Neskôr si uvedomili, že trh pre takýto systém nie je veľmi veľký a rozhodli sa vytvoriť systém pre mobilné zariadenia, ktorý by konkuroval v tej dobre obrovskému **Symbianu** a **Windows Mobile**. Firma bola o 2 roky neskôr – v roku 2005 odkúpená firmou **Google**. Prvá komerčná verzia systému bola vydaná v roku 2008 na zariadení **HTC Dream**. Od tohto roku popularita systému extrémne narástla, ale s ňou aj množstvo zariadení, na ktorom tento systém beží. Vývoj systému prebieha dodnes.

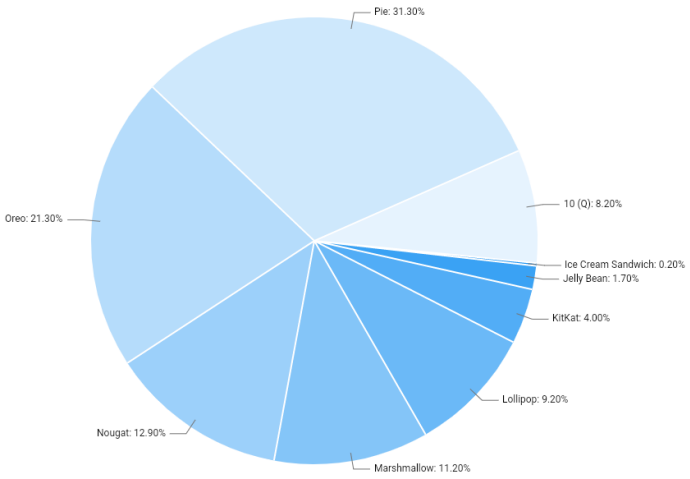
## 2.1 Verzie systému

Android vydal za svoj život mnoho verzií. Nová verzia systému je vydávaná každý rok. Najnovšia verzia v dobe písania práce je **Android 11**. Každá verzia Androidu do verzie 10 bola pomenovaná podľa cukroviniek, ktorej meno sa začínalo na určité písmeno abecedy, ktoré označovalo danú verziu.

Pri vytváraní aplikácie je nutné zvoliť najmenšiu verziu systému, ktorú bude naša aplikácia podporovať. Pri vytváraní môžeme vidieť množstvo zariadení, ktoré fungujú na danej verzii systému a podľa našej požiadavky vybrať verziu vzhľadom na to, či chceme podporovať čo najviac zariadení alebo využívať najnovšie funkcie systému.

Vydávanie novej verzie systému každý rok a open-source vývoj prináša hlavnú nevýhodu systému – veľké štiepenie na veľa verzií. Každý výrobca môže upraviť systém podľa seba. V prípade, že bude vydaná nová verzia systému a daný typ telefónu už nie je výrobcom

| Version | Codename | Percentage |
|---|---|---|
| 4.0 | Ice Cream Sandwich | 0.20% |
| 4.1 | Jelly Bean | 0.60% |
| 4.2 | Jelly Bean | 0.80% |
| 4.3 | Jelly Bean | 0.30% |
| 4.4 | KitKat | 4.00% |
| 5.0 | Lollipop | 1.80% |
| 5.1 | Lollipop | 7.40% |
| 6.0 | Marshmallow | 11.20% |
| 7.0 | Nougat | 7.50% |
| 7.1 | Nougat | 5.40% |
| 8.0 | Oreo | 7.30% |
| 8.1 | Oreo | 14.00% |
| 9.0 | Pie | 31.30% |
| 10.0 | 10 (Q) | 8.20% |




Obr. 2.1: Tabuľka a graf percentuálneho rozloženia počtu zariadení využívajúcich jednotlivé verzie systému z apríla 2020 [2].

podporovaný, tak nedostane najnovšiu aktualizáciu napriek tomu, že telefónu nič nechýba a užívateľ ho bude aj naďalej používať.

## 2.2 Architektúra systému

Na to, aby sme mohli začať vyvíjať aplikácie na Android musíme najprv pochopiť architektúru systému. Ako môžeme vidieť na obrázku 2.2, architektúra systému je rozdelená do viacerých vrstiev.

### Linux Kernel (Jadro)

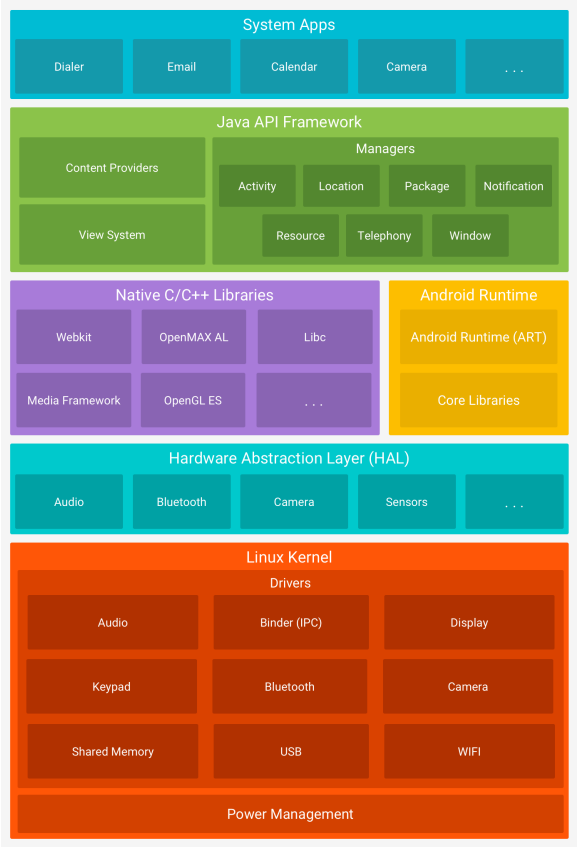
Ako som už spomínal, systém Android beží na Linuxovom jadre. Jadro systému riadi a spravuje procesy a pamäť. Výhodou Linuxového jadra je hlavne jeho bezpečnosť a uľahčenie vývoja ovládačov vďaka jeho rozšírenosti.

### Hardware Abstraction Layer (HAL)

Táto vrstva poskytuje rozhranie vyšším vrstvám systému, vďaka čomu aj tieto vrstvy môžu pristupovať k hardvérovým komponentom systému ako napríklad kamera. Táto vrstva obsahuje viacero knižníc, kde každá táto knižnica umožňuje prístup k inej hardvérovej časti systému.

### Natívne C/C++ knižnice

Keďže chceme dosiahnuť čo najväčšiu rýchlosť systému, tak Android poskytuje natívne knižnice napísane v jazyku `C` alebo `C++`, ktoré môžeme pohodlne zavolať z nášho `Java` kódu a tým využiť čo najväčší výkon zariadenia, ktorý by za použitia `Java` kódu nebol




Obr. 2.2: Jednotlivé vrstvy architektúry systému Android [3].

možný. Väčšinou sú to knižnice na rôzne vykresľovanie grafiky, ktorá je sama o sebe náročná operácia, napríklad *OpenGL* alebo *Media Framework*.

### Android Runtime (ART)

Je to vrstva, ktorá sa stará o samotné spúšťanie užívateľských aplikácii. Je navrhnutá tak, aby dokázala spúšťať viacero inštancií virtuálnych strojov, ktoré vykonávajú *byte kód* aplikácie **DEX**. Runtime aplikuje rôzne optimalizácie kódu ako napríklad **ahead-of-time** a **just-in-time** kompiláciu a tiež vykonáva aj **garbage collection**. Do Androidu 5 sa využíval tzv. *Dalvik runtime*.

### Java API framework

Táto vrstva poskytuje stavebné komponenty, ktoré využívame pri vývoji všetkých aplikácii. Tieto komponenty sú vyvinuté v Jave a sú delené na viaceré časti, medzi ktoré patria napríklad:

- **View System** – poskytuje všetky komponenty užívateľského rozhrania ako tlačítka, okná, textové polia,
- **Resource Manager** – poskytuje grafické zdroje, lokalizačné reťazce,
- **Notification Manager** – umožňuje aplikáciam zobrazovať notifikácie v zariadení,

- **Activity Manager** – manažuje životný cyklus aplikácie,
- **Content Providers** – umožňuje aplikáciam pristupovať k dátam iných aplikácii alebo zdielať svoje vlastné dáta.

### Systémové aplikácie

Je to najvyššia vrstva architektúry systému, jedná sa o samotné aplikácie, ktoré poskytujú určitú funkcionalitu systému.

## 2.3 Štruktúra aplikácii

Aplikácie pre Android môžu byť napísané v jazyku `Java`, `Kotlin` alebo `C++`. Pomocou SDK nástrojov následne kompilujeme tento kód do jedného súboru s príponou *.apk* (Android package). Jedná sa vlastne o archív, ktorý v sebe obsahuje všetko potrebné pre beh aplikácie.

V Androide poznáme 4 základné stavebné komponenty aplikácii. Každý komponent je vlastne vstupný bod do aplikácie, cez ktorý môže užívateľ alebo systém vstúpiť do aplikácie. Každý komponent slúži na niečo iné a každý z nich má vlastný životný cyklus [4].

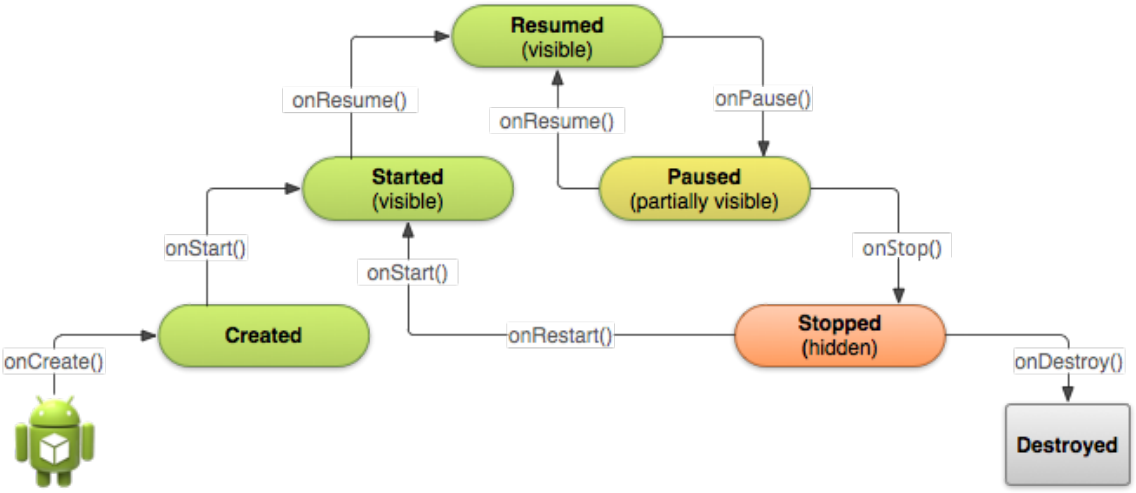
Unikátnou výhodou Androidu je, že každá komponenta môže byť spustená komponentom inej aplikácie, preto by jednotlivé komponenty mali byť nezávisle jedna od druhej. Napríklad v aplikácii pre posielanie správ môže byť jedna aktivita zoznam kontaktov a druhá aktivita na vytvorenie novej správy. Nezávisle by mali byť preto, že externá aplikácia, napríklad kamera, môže po vytvorení fotky spustiť aktivitu našej aplikácie na vytvorenie novej správy a tým umožniť rovno pridať novo vytvorenú fotografiu do správy.

### Aktivita (Activity)

Aktivita je vstupným bodom do aplikácie pre interakciu s užívateľom. Jedná sa o jednu obrazovku, ktorá obsahuje užívateľské rozhranie. Aplikácia môže obsahovať viacero aktivít, ale jednotlivé aktivity sú nezávisle jedna od druhej. Jednotlivé aktivity však spolu môžu spolupracovať.

Keďže každá aktivita má len jeden konkrétny účel, pričom dochádza k ich častému vytváraniu a zanikaniu. Aktivity majú definovaný životný cyklus, ktorý môžeme vidieť na obrázku 2.3. Aktivita môže mať 6 stavov a pri každom zmene stavu aktivity je volaná príslušná *callback* metóda, ktorá informuje o zmene stavu [5].

- **Created** – aktivita je v tomto stave iba raz a to pri prvotnom vytvorení. Aktivita sa v tomto stave iba inicializuje a ešte nie je viditeľná na obrazovke. Po skončení inicializácie automaticky prechádza do ďalšieho stavu.
- **Started** – aktivita začne byť viditeľná a interaktívna. Tento stav trvá krátko a aktivita, podobne ako v predchádzajúcom stave, automaticky prechádza do ďalšieho stavu.
- **Resumed** – stav, v ktorom aplikácia zotrváva, ak je stále viditeľná. Aktivita z tohto stavu prechádza do ďalšieho iba ak sa zmení viditeľnosť aktivity (napríklad minimalizácia).
- **Paused** – stav, ktorý indikuje, že aplikácia prestáva byť v popredí. Z tohto stavu môže byť aplikácia vrátená naspäť do popredia alebo ukončená.




Obr. 2.3: Životný cyklus aktivity [6].

- **Stopped** – aktivita v tomto stave nie je vôbec viditeľná na obrazovke. Z tohto stavu môže dôjsť k reštartovaniu aktivity alebo k jej úplnému zničeniu.
- **Destroyed** – stav, v ktorom aktivita už neexistuje a musí byť znovu vytvorená, ak ju chceme znovu použiť.

### Služba (Service)

Jedná sa o vstupný bod aplikácie, ktorý beží na pozadí a nemá žiadne užívateľské rozhranie. Využíva sa na vykonávanie dlhotrvajúcich operácii bez toho, aby rušili užívateľa a jeho interakciu s aplikáciou. Služba síce funguje v pozadí, ale stále pracuje na hlavnom vlákne aplikácie, takže náročné operácie môžu spomaliť vykresľovanie aplikácie. Jednoduchým príkladom služby je napríklad služba na prehrávanie hudby. Užívateľ chce od aplikácie na prehrávanie hudby, aby vybraná hudba hrala aj po minimalizácii aplikácie alebo pri práci s inými aplikáciami. Aktivita môže predávať správy službe tak, že sa na ňu naviaže (*bind*) alebo pomocou zámerov (*intent*). Podľa spôsobu vytvárania služieb ich delíme na:

- **Bounded** – tento typ služby sa automaticky vytvorí po prvom naviazaní aktivity a automaticky sa zruší po tom, čo sa odviažu všetky aktivity.
- **Unbouned** – tento typ služby je nutné explicitne vytvoriť a následne zrušiť.

Existujú 2 typy služieb, ktoré systém riadi odlišne:

- **Background** – užívateľ nemá tušenie o tom, že táto služba beží. Túto službu môže systém voľnejšie spravovať a prípadne ju aj ukončiť, ak potrebuje uvoľniť pamäť pre iné procesy.
- **Foreground** – služba beží v popredí a je možné identifikovať ju tak, že má notifikáciu. Systém sa bude snažiť túto službu udržať v pamäti najdlhšie, pretože jej koniec by užívateľa nepotešil. Vyššie spomenutý príklad na prehrávanie hudby by využíval tento typ služby.

### Prijímač vysielania (Broadcast receiver)

Táto komponenta umožňuje systému doručovať udalosti aplikáciam a vďaka tomu na ne môžu aplikácie patrične reagovať. Pretože táto komponenta je aj vstupným bodom do aplikácie, tak systém môže doručovať vysielania aj do aplikácii, ktoré aktuálne nie sú spustené. Výhodou tohto prístupu je, že aplikácia si môže naplánovať udalosť na presnú hodinu a v danú hodinu príjme vysielanie a preto nemusí stále bežať. Väčšina vysielaní pochádza priamo zo systému, ako napríklad upozornenie o vypnutom displeji alebo o vybitej batérii, ale aplikácie môžu definovať tiež vlastné vysielania a tým upozorňovať ostatné aplikácie. *Broadcast receivers* nemajú užívateľské rozhranie, ale môžu vytvoriť notifikáciu na upozornenie, keď nastane daná udalosť.

### Poskytovateľ obsahu (Content provider)

Vďaka tejto komponente môže aplikácia ukladať a zdieľať dáta aplikácie v úložisku zariadenia, SQL databáze alebo na webe. Ostatné aplikácie môžu cez poskytovateľa obsahu čítať, prípadne upravovať takto zdielané dáta ostatných aplikácii, ak im to poskytovateľ povoľuje. Napríklad Android má poskytovateľa obsahu, ktorý umožňuje ktorejkoľvek aplikácii, ktorá má na danú operáciu právo, čítať a upravovať telefónny zoznam osôb a ich informácie na zariadení.

## 2.4 Vytváranie komponentov aplikácie

Ak chceme vytvoriť niektorý z vyššie spomenutých komponentov (aktivita, služba alebo prijímač vysielania) použijeme na to objekt nazývaný **zámer** (*intent*). Zámer je asynchrónna správa, ktorá viaže jednotlivé komponenty dohromady.

Pri aktivitách a službách zámer definuje akciu, ktorú majú vykonať a môže tiež definovať dodatočné údaje, ktoré môže komponent potrebovať k činnosti. Napríklad pri štartovaní aktivity na zobrazenie obrázka definujeme *URI* obrázka na zobrazenie.

Pri prijímačoch vysielania je zámer vlastne správa, ktorú prijímame. Napríklad zámer pri vysielaní, ktoré indikuje vybitie batérie zariadenia obsahuje iba vopred známy reťazec, ktorý indikuje toto vybitie.

Na rozdiel od vyššie spomenutých komponentov sa poskytovateľ obsahu neaktivuje pomocou zámeru, ale aktivuje ich požiadavka od **riešiteľa obsahu** (*content resolver*). Riešiteľ obsahu spracováva všetky transakcie s poskytovateľmi obsahu, aby sa zaručila bezpečnosť.

## 2.5 Android Manifest

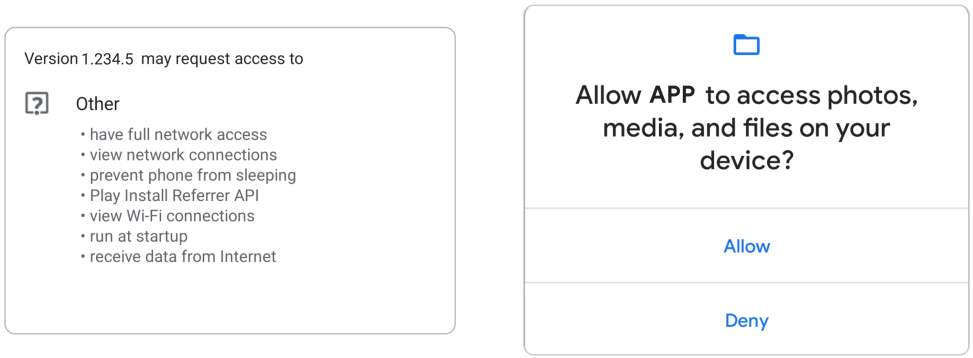
*Android Manifest* je súbor, ktorý musí byť prítomný v každej aplikácii. Vďaka tomuto súboru systém vie, aké komponenty obsahuje aplikácia. Jedná sa o `XML` súbor, v ktorom sú okrem komponentov definované aj nasledujúce veci:

- Identifikuje všetky potrebné povolenia, ktoré aplikácia potrebuje k svojmu behu.
- Deklaruje minimálnu verziu systému, na ktorom je aplikácia schopná fungovať ako je opísané v kapitole 2.1.
- Deklaruje softvérové a hardvérové požiadavky, ktoré aplikácia vyžaduje, ako napríklad kamera alebo bluetooth.

- Deklaruje *API* knižnice, voči ktorým musí byť aplikácia skompilovaná.

## 2.6 Povolenia aplikácií

Ako som popísal v kapitole 2.2, aplikácia beží vo svojom vlastnom virtuálnom stroji a tým pádom je izolovaná od ostatných aplikácii bežiacich v systéme. Každá aplikácia v systéme Android je štandardne spúšťaná s minimálnymi povoleniami. To znamená, že aplikácia nemá prístup ku komponentom systému, ktoré nepotrebuje ku správnemu fungovaniu. Vďaka tomuto prístupu sa dosahuje oveľa vyššia bezpečnosť systému a užívateľ má kontrolu nad tým, k čomu môžu aplikácie pristupovať.




Obr. 2.4: Zľava: normálne povolenia, o ktorých nás systém informuje a prideľuje počas inštalácie; povolenie počas fungovania aplikácie, ktoré musí užívateľ potvrdiť alebo odmietnuť [7].

Android obsahuje viacero typov povolení, kde každý typ umožňuje prístup len k určitým komponentom systému:

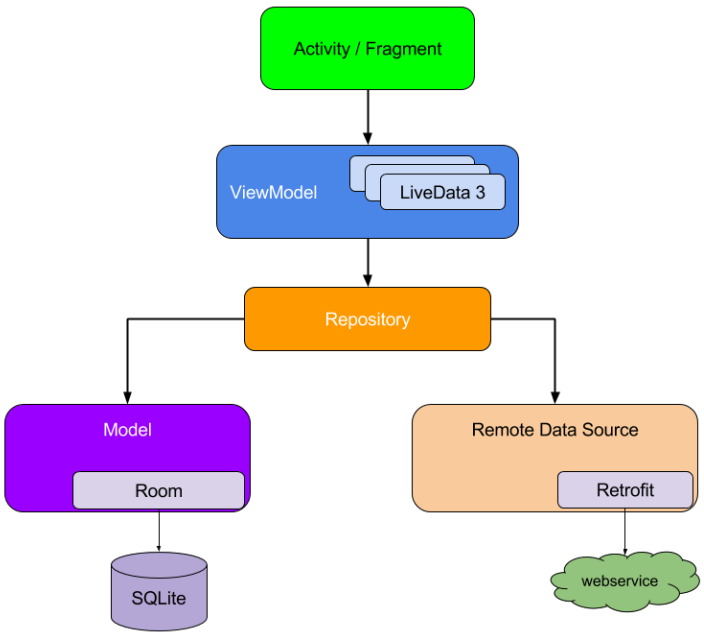
- **Normálne povolenia** – tieto povolenia umožňujú aplikáciam pristupovať k dátam mimo ich virtuálneho stroja. Sú medzi nimi povolenia, ktoré len minimálne ovplyvňujú systém alebo iné aplikácie, ako napríklad povolenie na pripojenie na internet, spúšťanie pri štarte... Tieto povolenia sú automatický pridelené pri inštalácii aplikácie, pričom systém upozorňuje na vyžadované povolenia ako vidieť na obrázku 2.4.
- **Podpisové povolenia** – ak aplikácia využíva povolenie, ktoré definovala iná aplikácia, tak systém udelí toto povolenie iba ak sa zhodujú ich kľúče, ktorými boli obidve aplikácie podpísané. Patrí sem napríklad povolenie na zapisovanie systémových nastavení.
- **Povolenia počas fungovania** – inak nazývané aj *nebezpečné* povolenia. Tieto povolenia si musí aplikácia vyžiadať od užívateľa, ktorý ich použitie môže potvrdiť alebo zamietnuť. Pri vyžiadaní sa zobrazí okno na potvrdenie alebo zamietnutie ako je možné vidieť na obrázku 2.4. Patria sem povolenia, ktoré môžu ohroziť súkromie užívateľa – napríklad prístup k polohe, kontaktom, kamere alebo k súborom na zariadení.
- **Špeciálne povolenia** – tieto povolenia môžu vytvárať a využívať len *OEM* systémové aplikácie, ktoré sú dodávané pri inštalácii systému na zariadenie.

## 2.7 Architektúra aplikácii

Pri písaní aplikácii je dôležité dodržovať princíp **oddelenia zodpovedností**. Tento princíp vraví o tom, že každá trieda by v sebe mala obsahovať iba logiku, ktorá s ňou súvisí. Príkladom sú napríklad triedy, ktoré spracovávajú užívateľské rozhranie. Tieto triedy by mali v sebe obsahovať čisto len logiku, ktorá sa týka spracovania tohto UI a nie napríklad logiku ukladania dát. Dodržiavaním tohto princípu sa vyhneme mnohým problémom pri vývoji ako napríklad problémy so životným cyklom komponentov a zaručíme si ľahšiu udržateľnosť a rozširovateľnosť kódu.

Ak by sme tento princíp nedodržali a do UI komponentov by sme dali logiku komunikácie so serverom, môže sa stať, že táto komponenta bude ukončená systémom alebo používateľom a vtedy stratíme aj pripojenie k serveru.

Aby sme sa vyhli týmto problémom, Android vývojári navrhli architektúru aplikácii nazývanú **Model-View-ViewModel**, tiež známou pod skratkou **MVVM**. Ako vyplýva z názvu, táto architektúra sa skladá z Modelu, View a View-Modelu, kde každý komponent má odlišnú úlohu. Ako môžeme vidieť na obrázku 2.5, každá komponenta je závislá len na komponente pod ňou.




Obr. 2.5: MVVM architektúra odporúčaná pri vývoji aplikácii pre Android [8].

**View** je komponentom architektúry, ktorý vidí užívateľ na obrazovke a interaguje s ním. Táto komponenta dostáva informácie od *View-Modelu* a nemá žiadne tušenie o existencií *modelu*. Nemala by obsahovať žiadnu biznis logiku. V tejto komponente dochádza často k jej zmene, napríklad pri rotácii zariadenia. Patria sem napríklad *Aktivity* a *Fragmenty*.

Komponenta architektúry **View-Model** poskytuje dáta pre *view* a obsahuje logiku na komunikáciu s *modelom*. Tento komponent nevie o existencií *view* komponentov a preto nie je afektovaný ich zmenami ako je spomínané vyššie.

**Model** komponent obsahuje všetku biznis logiku aplikácie. Patrí medzi ne napríklad lokálna databáza v zariadení. Jedna aplikácia môže mať viacero zdrojov informácii (lokálna

databáza, REST API...). Aby nedošlo k pridávaniu logiky do *View-Modelu* pri rozhodovaní, z ktorého modelu čerpať, tak sa do architektúry pridáva **repozitár**.

**Repozitár** na základe svojej logiky rozhoduje, z ktorého modelu bude čerpať dáta. V architektúre je to jediný komponent, ktorý je závislý na viacerých triedach. Táto komponenta by mala ukladať dáta do svojej *cache* a v prípade, že sú tieto dáta zastarané, mala by automaticky tieto dáta v pozadí aktualizovať. Ukladanie do *cache* zaručuje responzívnejšiu aplikáciu, keďže užívateľ nemusí čakať na dokončenie príkazu a šetrí internetové dáta.

V tejto architektúre sa na prepojenie medzi jednotlivými komponentami využíva pozorovateľný držiteľ údajov nazývaný **LiveData**. Využíva návrhový vzor pozorovateľ (*observer*) a vďaka nemu môžu jednotlivé komponenty pozorovať zmeny objektov bez vytvárania explicitnej závislosti medzi nimi. *LiveData* je závislý na životnom cykle komponentu a preto zabraňuje rôznym chybám v aplikácii. Tiež zaručuje že komponent, ktorý ho využíva, bude mať k dispozícii vždy najnovšie dáta a vykoná aj vopred definovanú automatickú aktualizáciu komponentu po prijatí nových dát [9].

## 2.8 Nástroje na vývoj

### 2.8.1 Android Studio

Android Studio je oficiálne vývojové prostredie (*Integrated Development Environment – IDE*) určené pre vývoj Android aplikácii, ktoré je založené na **IntelliJ IDEA** vývojovom prostredí. Na kompiláciu aplikácii využíva **Gradle**. Toto vývojové prostredie ponúka mnoho funkcií, ktoré uľahčujú vývoj [10]:

- Emulátor systému Android.
- Podpora vývoju `C++` akcelerovaných aplikácii.
- Nástroje na testovanie kódu.
- Nástroje, ktoré vykonávajú statickú analýzu kódu a zachytávajú mnohé chyby ako napríklad nekompatibilitu verzií, výkonnostné problémy.
- Integrácia *git* pre jednoduchšie verzovanie kódu.
- Nástroj na jednoduchú tvorbu užívateľských rozhraní.
- Automatické formátovanie kódu pre lepšiu čitateľnosť.
- Prehľadné debugovanie kódu.

### 2.8.2 Gradle

**Android Studio** využíva **Gradle** systém na kompiláciu Android aplikácie do výsledného *apk* archívu. Tento systém nie je viazaný na Android Studio, je možné ho využiť aj z príkazového riadku. Gradle umožňuje viacero funkcií:

- Vytvárať viacero verzií aplikácie z jedného zdrojového kódu.
- Vykonávať viaceré varianty kompilácie.
- Zmenšovanie zdrojov a *code obfuscation*.

- Podpisovanie skompilovaných aplikácii vopred vygenerovaným kľúčom.
- Manažovanie závislostí a ich automatické sťahovanie.

## 2.9 Vytváranie fotiek

V dokumentácii systému sa odporúča využiť jeden z dvoch možných prístupov na prácou s kamerou: CameraX [11] alebo Camera2 [12]. Pri používaní kamery na Androide verzie 6 a novšej je nutné pred použitím kamery vyžiadať povolenie od užívateľa.

Keďže každé zariadenie obsahuje iný senzor a každý výrobca si implementuje vlastnú natívnu aplikáciu na vytváranie fotiek, je potrebné nájsť riešenie, ktoré funguje na väčšine zariadení.

### 2.9.1 Camera2

Tento balíček poskytuje rozhranie pre priamy prístup ku kamerám v zariadení. Jej výhodou je priamy prístup, ktorý je využiteľný pri rôznych špecifických prípadoch. Fotky môžeme vytvárať vo formáte *JPEG*, *DNG*, prípadne čítať priamo *YUV* pixely zo senzoru. Medzi nevýhodu ale patrí napríklad nekonzistentné použitie na zariadeniach od rôznych výrobcov. Túto nevýhodu posilňuje aj fakt, že podpora tohto API je povinná od Androidu 5, no mnohí výrobcovia ho stále úplne nepodporujú ani na najnovších verziách. Z vlastnej skúsenosti na mnohých telefónoch čínskej značky stále chýba podpora *DNG* fotiek.

### 2.9.2 CameraX

Jedná sa o podpornú knižnicu, ktorá je vyvíjaná priamo firmou *Google*. Táto knižnica využíva *Camera2* a zaručuje kompatibilitu na všetkých zariadeniach až po verziu Androidu 5. Najväčšou výhodou použitia tejto podpornej knižnice je, že zaručuje konzistentnú funkcionalitu API naprieč všetkými zariadeniami rôznych výrobcov a rozličných verzií a tým rieši všetky problémy kompatibility pri použití *Camera2*. Medzi ďalšie výhody patrí aj jednoduchosť použitia a možnosť aplikovať rozšírenia, ktoré umožňujú využiť funkcie, ktoré ponúka natívna aplikácia na vytváranie fotiek. Knižnica umožňuje vytvárať fotky vo formáte *JPEG*.

# Kapitola 3

# Spracovanie obrázkov

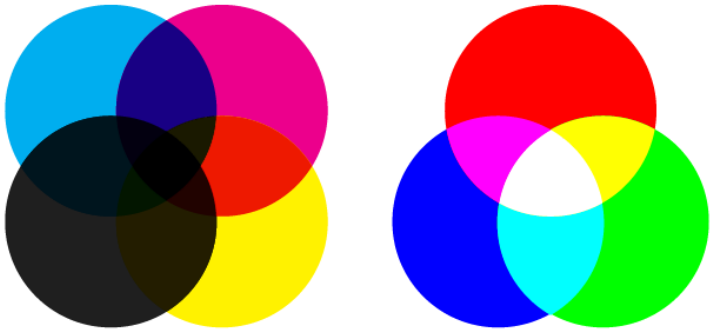
## 3.1 Farebné modely

Farebný model je špecifikovaný trojrozmerný systém, ktorý slúži na uľahčenie špecifikácie farieb. Každá farba v tomto modeli reprezentuje jeden bod v danom modeli. Na zaznamenanie farby teda potrebujeme vedieť súradnicu z každej osi tohto modelu. Farebné modely môžeme podľa spôsobu miešania farieb rozdeliť do 2 kategórii [13]:

- **Aditívne** – modely, ktoré sa využívajú najčastejšie na obrazovkách alebo displejoch. Patrí sem model RGB.
- **Subtraktívne** – tieto modely využívajú hlavne zariadenia pracujúce s pigmentom ako napríklad tlačiarne. Patrí sem model CMYK.

### RGB

Tento model patrí medzi najznámejšie a najrozšírenejšie modely v počítačovej grafike. Názov tohto modelu je odvodený od troch hlavných farieb, ktoré využíva – červená (red), zelená (green) a modrá (blue). Zmiešaním týchto troch farieb pri ich maximálnej intenzite získavame bielu farbu. V počítačoch sa najčastejšie využíva 24 bitový RGB model. To znamená, že každá zložka vie vyjadriť 256 hodnôt od 0 do 255 (8 bitov) a celý model dokáže vyjadriť 16,7 milióna farieb. Znázornenie RGB modelu môžeme vidieť na obrázku 3.1.



Obr. 3.1: CMYK farebný model (vľavo) a RGB model (vpravo) [13].

### CMYK

Ako som spomínal vyššie, tento model sa využíva hlavne pri tlači. Jeho názov označuje základné farby, ktoré využíva – azúrovú, purpurovú a žltú spolu s čiernou. Jedná sa o subtraktívny model, čo znamená, že výsledná farba sa vytvorí odstránením niektorých farieb z bieleho povrchu nanesením týchto farieb. Teoreticky nanesením všetkých farieb by mala vzniknúť čierna. Znázornenie modelu je možné vidieť na obrázku 3.1.

### YUV

Tento model bol pôvodne využívaný na prenos televízneho vysielania. Model vznikol preto, aby sa zaručila kompatibilita farebného signálu na čiernobielych televíziách. Jedná sa teda o analógový model. Znázornenie modelu môžeme vidieť na obrázku 3.2.

Hlavnou podstatou tohto modelu je, že oddeľuje jas od farebných zložiek. Zložka **Y** reprezentuje jas a má hodnoty v rozsahu od 0 do 255, zložky **U** a **V** reprezentujú farebnosť (*chromacity*), kde **U** má rozsah od -112 do 112 a **V** od -157 do 157.

Na to, aby sme dokázali obrázky v tomto formáte lepšie spracovávať a zobrazovať, potrebujeme vedieť ako ich previesť do RGB modelu. Na prevod z RGB farebného modelu do modelu YUV a naopak použijeme následujúce rovnice [14]:

$$\begin{aligned} \mathbf{Y} &= 0.299 * R + 0.587 * G + 0.114 * B \\ \mathbf{U} &= -0.147 * R - 0.289 * G + 0.436 * B \\ \mathbf{V} &= 0.615 * R - 0.515 * G - 0.100 * B \end{aligned} \tag{3.1}$$

$$\begin{aligned} \mathbf{R} &= Y + 1.140 * V \\ \mathbf{G} &= Y - 0.395 * U - 0.581 * V \\ \mathbf{B} &= Y + 2.032 * U \end{aligned} \tag{3.2}$$

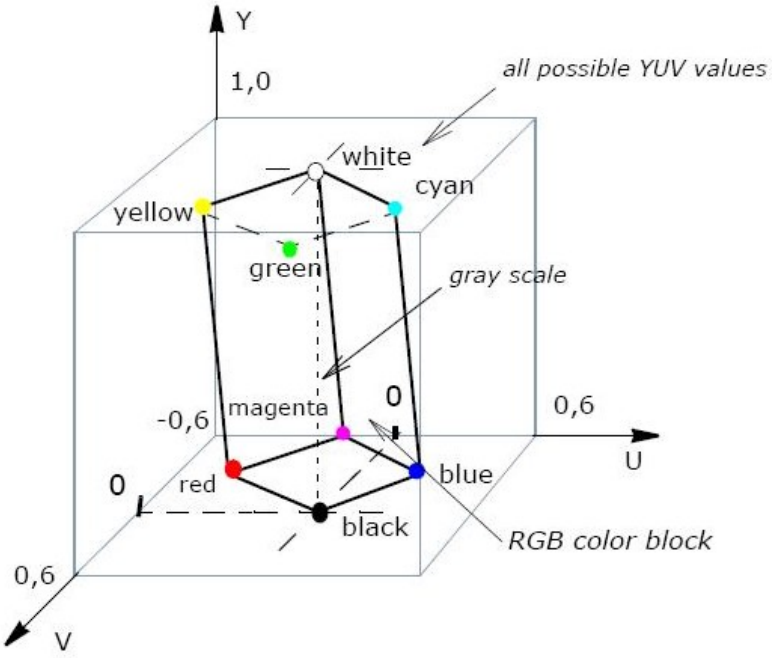
### YCbCr

Názov YUV sa v dnešnej dobe považuje za totožný s názvom modelu YCbCr, ktorý reprezentuje novší digitálny model, ktorý sa využíva kódovanie fotiek a videí. Tiež je rozdiel v ich prevode z RGB. Napríklad systém Android využíva pri vytváraní fotiek práve tento model [15].

Tento model vychádza z pôvodného YUV modelu a vznikol zmenšením/zväčšením a posunutím jeho hodnôt. Zložka **Y** má rozsah od 16 do 235 a zložky **Cb** a **Cr** majú rozsah od 16 do 240. Následujúce rovnice je možné využiť na prevod medzi RGB farebným modelom a modelom YCbCr a naopak:

$$\begin{aligned} \mathbf{Y} &= 0.257 * R + 0.504 * G + 0.098 * B + 16 \\ \mathbf{Cb} &= -0.148 * R - 0.291 * G + 0.439 * B + 128 \\ \mathbf{Cr} &= 0.439 * R - 0.368 * G - 0.071 * B + 128 \end{aligned} \tag{3.3}$$

$$\begin{aligned} \mathbf{R} &= 1.164 * (Y - 16) + 1.596 * (Cr - 128) \\ \mathbf{G} &= 1.164 * (Y - 16) - 0.813 * (Cr - 128) - 0.391 * (Cb - 128) \\ \mathbf{B} &= 1.164 * (Y - 16) + 2.018 * (Cb - 128) \end{aligned} \tag{3.4}$$




Obr. 3.2: Zobrazenie **RGB** modelu v **YCbCr** farebnom modele [13].

## 3.2 Farebné pod-vzorkovanie

Jedná sa o metódu kompresie obrázkov, ktorá využíva fakt, že ľudské oko je oveľa citlivejšie na zmeny jasu ako na farbu. Tento spôsob kompresie sa využíva aj pri **JPEG** formátoch a pri *YUV* modeloch [16].

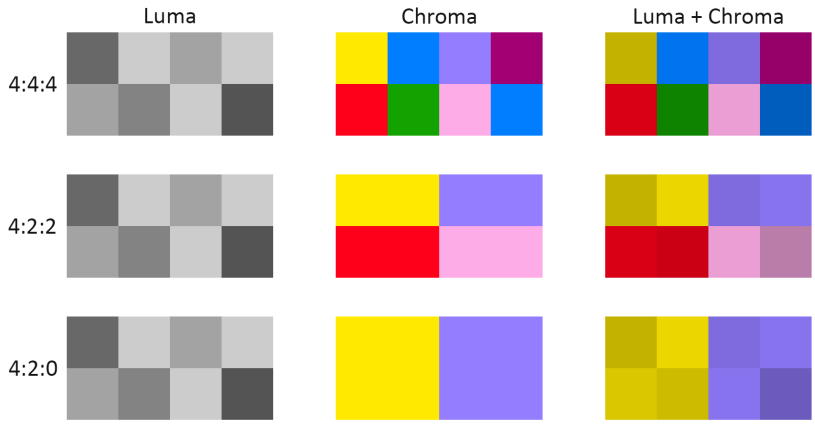
Princíp spočíva v tom, že zložky farby sú podvzorkovávane, čo znamená, že sú priemerované s okolitými pixelmi a informácia o jase ostáva nezmenená pre každý pixel. Keďže ľudské oko vníma viac zmenu jasu ako farby, tak túto zmenu skoro vôbec nepostrehne. Pri blokoch pixelov veľkosti $4 \times 2$ poznáme viaceré typy podvzorkovania, ktoré udávame v tvare 4:X:Y, kde X je počet farebných zložiek v hornom riadku a Y počet farebných zložiek v spodnom riadku. Tieto typy môžeme vidieť znázornené na obrázku 3.3:

- **4:4:4** – žiadne podvzorkovanie.
- **4:2:2** – horizontálne podvzorkovanie v každej farebnej zložke.
- **4:2:0** – horizontálne a vertikálne podvzorkovanie v každej farebnej zložke na polovicu. Tento typ podvzorkovania je využívaný v systéme Android pri vytváraní fotiek.

## 3.3 Formáty obrázkov

Keďže v práci vytvárame fotografie a tie následne musíme spracovávať na úrovni pixelov, tak je dôležité pochopiť, aký formát musíme využiť, aby sme v nevhodnosti formátu nestrácali informácie, ktoré by pomohli k lepšiemu výsledku.

Všeobecne delíme formáty obrázkov do 2 kategórii: vektorové a rastrové. V tejto práci vektorové formáty vôbec nevyužijeme. Rastrové formáty sa ďalej delia na stratové a bezstra-




Obr. 3.3: Znázornenie rôznych typov YUV pod-vzorkovania[1].

tové. Pri stratových formátoch môže obrázok pre ľudské oko vyzerať totožne s bezstratovými formátmi, ale na úrovni pixelov už je možné vidieť tieto straty [17].

### 3.3.1 JPEG

Jedná sa o najrozšírenejší formát obrázkov využívaný na internete, ale aj pri vytváraní fotiek. Jedná sa o stratový formát, čo znamená, že jeho algoritmus redukuje veľkosť súboru vymazávaním detailov z obrázka, ktoré ľudské oko nedokáže postrehnúť. Tieto detaily sú po vymazaní nenávratné stratené. Tomuto algoritmu sa dá nastaviť sila kompresie a tým manipulovať výslednú veľkosť a kvalitu obrázka. Tento formát ale nie je vhodný pre rôzne obrázky s kresbami, ikonami a podobne, pretože algoritmus funguje na princípe podobnosti farieb v susedných pixeloch. Pri kompresií sa využíva práve farebný model **YCbCr**.

Jeho hlavnou výhodou je malá veľkosť súboru po kompresii a pomerne vysoká kvalita obrázka. Nevýhodou je, že táto kvalita v detailoch nemusí byť dostatočná pri dodatočnej úprave a spracovaní obrázkov.

### 3.3.2 RAW

RAW súbory obsahujú dáta priamo zo senzoru fotoaparátu, takže nepoužívajú žiadnu kompresiu dát. Jedná sa o bezstratový formát najvyššej kvality, kde všetky informácie pri zachytávaní snímky sú uložené a je možné ich meniť aj neskôr pri úprave fotografie, ako napríklad vyváženie bielej, kontrast, expozícia... Vďaka týmto vlastnostiam je tento formát využívaný hlavne profesionálnymi fotografmi, ktorým pri upravovaní obrázka umožňuje meniť všetky vyššie spomenuté vlastnosti. V konečnej fáze po úprave obrázka ho ale ukladajú do veľkosťou menšieho formátu ako napríklad JPEG.

Hlavnou nevýhodou tohto formátu je jeho veľkosť na disku, ktorá niekoľkonásobne prevyšuje veľkosť rovnakej fotografie v JPEG formáte. tiež nie je štandardizovaný, čo znamená, že senzory kamery nám môžu od rôznych výrobcov produkovať rozličné formy súboru.

Výhodou je, že pri upravovaní tohto súboru nie sú tieto zmeny deštruktívne, ale sú len zapísané ako metadáta, ktoré sú využité pri vykresľovaní a kedykoľvek reverzibilné.

[1]Prevzaté z https://www.rtings.com/tv/learn/chroma-subsampling

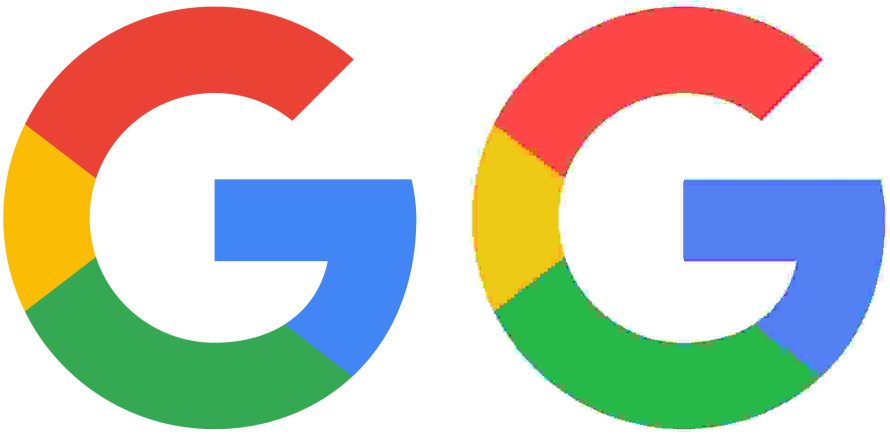
### 3.3.3 DNG

Tento formát, celým menom **Digital Negative**, je podobný formátu RAW, ale rieši jeho hlavnú nevýhodu – je presne špecifikovaný, čo znamená, že rozličné senzory kamier by mali produkovať stále rovnaký formát súboru. Bol vytvorený firmou **Adobe**. Rieši tiež jeho ďalšiu nevýhodu, ktorá je veľkosť súboru. DNG súbory dokážu byť menšie o 15 až 20% bez straty akejkoľvek kvality. Formát obsahuje aj kontrolný súčet, ktorý kontroluje, či nedošlo k poškodeniu dát súboru. Výhodou formátu je, že je neustále vyvíjaný a sú pridávané nové funkcionality.

Nevýhodou formátu ale je, že prevod medzi RAW a DNG je zdĺhavý a pri konverzií vymaže nerozpoznané metadáta z RAW súboru, čím sú nenávratne stratené. tiež zmena DNG súboru je permanentná na rozdiel od zmeny v RAW súbore.

### 3.3.4 PNG

Formát menom **Portable Network Graphics** pôvodne vznikol ako náhrada formátu GIF. Patrí medzi najrozšírenejšie formáty, pretože využíva bezstratovú kompresiu, čím zachováva kvalitu obrázkov pri menších veľkostiach ako RAW/DNG, ale nepodporuje úpravy vlastností ako tieto formáty. Na kompresiu využíva algoritmus **DEFLATE**. Tento formát tiež podporuje transparentnosť.



Obr. 3.4: Porovnanie bezstratového formátu *PNG* (vľavo) a stratového formátu s maximálnou kompresiou *JPG* (vpravo)[3].

[3]Prevzaté z https://commons.wikimedia.org/wiki/File:Google_%22G%22_Logo.svg

# Kapitola 4

# Vyhľadávanie významných bodov a zarovnanie obrázkov

Veľkou časťou tejto práce je zarovnanie viacerých obrázkov obsahujúcich totožný objekt do rovnakej polohy, aby sa na nich následne dali vykonávať ďalšie úpravy. K tomuto zarovnaniu potrebujeme poznať významné body a následne nájsť zhodné body a na základe tejto zhody vypočítať homografiu potrebnú na úpravu obrázkov. Nasledujúca kapitola čerpá z [18].

## 4.1 Významné body

Pred tým, ako si predstavíme techniky na detekciu významných bodov, musíme si vysvetliť čo sú významné body. Neexistuje presná definícia, ale významný bod si môžeme predstaviť ako zaujímavú časť obrázka, ktorá sa nejako odlišuje od svojho okolia. Medzi významné body patria napríklad hrany, body a rohy.

Znázornenie významných bodov na 2 rôznych obrázkoch môžeme vidieť na obrázku 4.1. Medzi významné body patria body, kde dochádza k nejakej zmene v hrane objektu alebo prieseku medzi viacerými okrajmi. Žiaden bod sa nenachádza na plochej, jednofarebnej stene, pretože tento bod nie je možné označiť ako významný, keďže stena rovnakej farby sa nachádza na viacerých miestach.

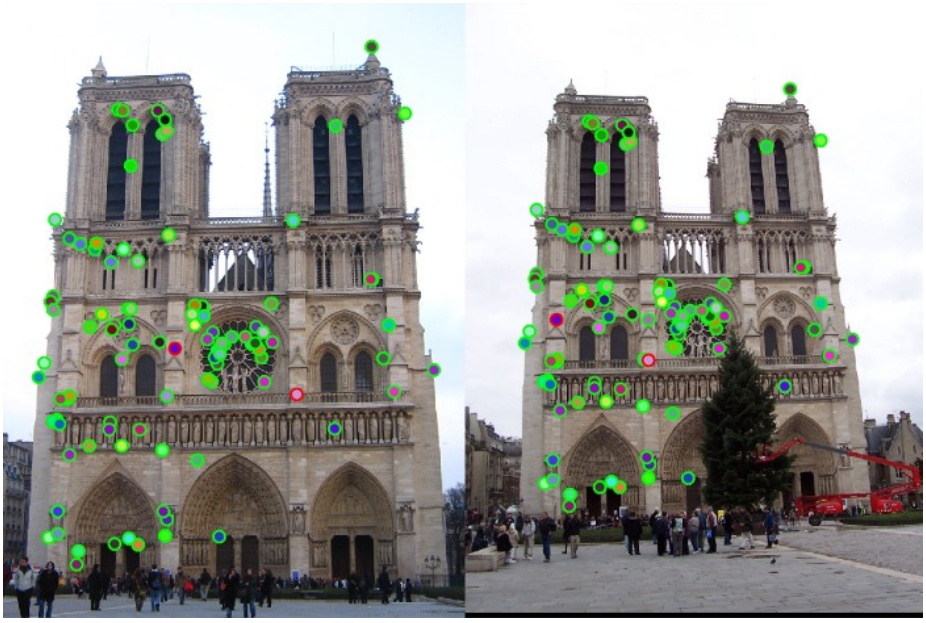
## 4.2 Deskriptory

Pri praktickom využití významných bodov nám často nestačia len samotné body, keďže tieto body často používame pri hľadaní zhodných významných bodov v rozličných obrázkoch rovnakej scény/objektu. Na to, aby sme jednoznačne našli spolu korešpondujúce body, je potrebné k nim nájsť aj ich **deskriptory**.

Deskriptor je vlastne popis alebo odtlačok určený jednému významnému bodu. Tento deskriptor by mal byť v ideálnom prípade nemenný pri transformácii, rotácii alebo škálovani obrázka, aby sme dokázali rovnaký významný bod nájsť pomocou tohto deskriptora aj v inom obrázku, ktorý zachytáva ten istý objekt. Kvalita deskriptora je hlavným parametrom, ktorý rozhoduje, či dokážeme rovnaký bod nájsť a priradiť zhodnému bodu v inom obrázku. Deskriptory delíme do dvoch kategórii:

- **Lokálne deskriptory** – využívajú na popis významného bodu iba jeho bezprostredné okolie.

- **Globálne deskriptory** – popisujú celý obrázok, ale ich nevýhodou, že zmena v časti obrázku zmení celý deskriptor.



Obr. 4.1: Vyhľadanie významných bodov na 2 odlišných obrázkoch zachytávajúcich ten istý objekt[2].

## 4.3 Detekcia významných bodov

Hlavnou úlohou detektorov významných bodov je nájsť také invariantné významné body medzi rozličnými obrázkami, vďaka ktorým môžme efektívne zosúladiť lokálne štruktúry naprieč týmito obrázkami. Aby sme dosiahli tento cieľ je nutné, aby detektory spĺňali nasledovné vlastnosti:

- **Robustnosť** – detektor by mal byť schopný detegovať rovnaké významné body bez ohľadu na škálovanie, rotáciu, posuny a rôzne deformácie a kompresie obrázkov.
- **Opakovateľnosť** – detektor by mal detegovať rovnaké významné body naprieč rôznymi obrázkami rovnakej scény alebo objektami z rôznych uhlov pohľadu.
- **Presnosť** – všetky nájdene body by mali byť presne lokalizované
- **Všeobecnosť** – všetky detegované významne body by sa mali dať použiť v rôznych aplikáciach.
- **Efektívnosť** – algoritmus by mal nájsť významné body v obrázku pomerne rýchlo, aby bola možná podpora detekcie v reálnom čase.
- **Množstvo** – mal by detegovať takmer všetky významné body v obrázkoch, pričom hustota týchto bodov by mala reprezentovať informáciu o obsahu obrázku.

[2]Prevzaté z https://medium.com/data-breach/introduction-to-feature-detection-and-matching-65e27179885d

V reálnom svete ale neexistuje ideálny detektor. Existuje viacero typov detektorov, pričom každý z nich sa zameriava na konkrétny typ využitia a preto sa musia robiť kompromisy – nie každá vlastnosť je rovnako dôležitá.

### 4.3.1 Harris Corner Detection

V praxi jeden z najčastejšie používaných algoritmov na detekciu hrán v obrázkoch patrí *Harrisov detektor*, ktorý v roku 1988 vymysleli *Chris Harris* a *Mike Stephens* [19]. Vychádza z *Moravecovho detektora*, ktorý funguje tak, že po obraze sa pohybuje štvorec určitej veľkosti a každých 45° vypočíta zmenu intenzity v obraze a tým deteguje roh. *Harris* a *Stephens* vylepšili algoritmus tak, že namiesto posuvného okna sa gradienty počítajú pomocou derivácie.

Výstupom algoritmu je množina významných bodov bez deskriptorov, čo znamená, že je invariantný voči rotácii, posunutiu a šumu, no nie je invariantný voči škálovaniu a preto je pri vyhľadávaní významných bodov v obrázkoch reálnych scén nepoužiteľný.

### 4.3.2 SIFT

**Scale Invariant Feature Transform** je technika, ktorá bola vytvorená *D. G. Lowem* v roku 2004 [20]. Jej výstupom sú významné body a ich deskriptory, čo znamená, že táto technika je invariantná voči rotácii, posunutiu, šumu a aj voči škálovaniu. Vďaka týmto výhodám je prakticky využiteľná aj v našom nasadení. Táto technika bola do roku 2020 licencovaná a preto bolo potrebné pri komerčnom použití zakúpiť licenciu na použitie od jej autorov. V roku 2020 ale patent[3] vypršal. Patrí medzi jednu z najznámejších techník na detekciu významných bodov a ich deskriptorov. Získanie významných bodov z obrázkov prebieha pomocou **DoG** (*Difference of Gaussians*). Významné body sú vyberané ako extrémy *DoG* funkcie.

Každému bodu záujmu sa použitím rôznych rozmerov a okolia okolo bodu záujmu odhadne lokálna orientácia využitím lokálnych vlastností obrázka, čím sa zaručí invariancia voči rotácii.

Následne je vypočítaný deskriptor pre každý zistený bod, ktorý je založený na lokálnych informáciach obrázka vo vybranom rozmere. *SIFT* deskriptor vytvára histogram gradientov orientácie bodov, ktoré sú v okolí významného bodu. V tomto histograme následne hľadá najvyššiu hodnotu orientácie a tiež všetky hodnoty, ktoré sú v rozmedzí 80% od najvyššej a tieto hodnoty použije ako dominantné orientácie významného bodu.

Deskriptor v tejto technike je navrhnutý tak, aby bol odolný voči zmene lokácie, orientácie a rozmeru. Nie je explicitne odolný voči afinným transformáciám, ale ukázalo sa, že je odolný voči deformáciám, ktoré sú spôsobené napríklad zmenou perspektívy. Vďaka týmto charakteristikám vykazuje vynikajúcu presnosť oproti konkurenčným technikám.

Hlavnou nevýhodou tejto techniky je to, že konštrukcia vektora vlastností je veľmi komplikovaná a preto je ovplyvnený aj výpočtový čas deskriptorov, čo znamená, že technika je pomerne pomalá.

### 4.3.3 SURF

**Speeded-Up Robust Features** je technika detekcie významných bodov spolu s ich deskriptormi vytvorená *H. Bayom* [21] ako efektívna alternatíva *SIFT-u*. Je oveľa rýchlejšia a

[3] https://patents.google.com/patent/US6711293

robustnejšia ako *SIFT*. Táto rýchlosť je dosiahnutá napríklad tým, že vo fáze detekcie bodov záujmu sa namiesto využitia Gaussovských derivácii využívajú jednoduché 2D boxové filtre. Táto technika je tiež licencovaná a preto je potrebné pri komerčnom použití zakúpiť licenciu na použitie od jej autorov.

Hlavnou výhodou *SURF* deskriptoru oproti *SIFT* deskriptoru je menší čas výpočtu vďaka využitiu 64 dimenzionálneho vektoru vlastností na opis lokálnych vlastností, na rozdiel od *SIFT-u*, ktorý využíva 128 dimenzionálny vektor.

Aj keď sa rýchlosť výpočtu výrazne znížila, tak *SIFT* sa stále viac hodí pri obrázkoch, ktoré sú posunuté, otočené, škálované alebo majú rôzne deformácie svetla. Nevýhodou *SURF-u* sú tiež prípady, v ktorých je otočenie obrázkov veľmi extrémne alebo zmena uhla pohľadu medzi obrázkami je veľmi veľký. Podobne ako *SIFT*, *SURF* tiež nie je plne afinne invariantný.

### 4.3.4 ORB

**Oriented FAST and Rotated BRIEF** je technika detekcie vyznamných bodov a ich deskriptorov, ktorá bola v roku 2011 vytvorená v `OpenCV` laboratóriách [22]. Hlavným zámerom za vytvorením *ORB-u* bolo vytvoriť alternatívu k *SIFT* a *ORB* algoritmom, ktorý je oveľa rýchlejší, má vyššiu presnosť a zároveň je voľne použiteľný aj bez zakúpenia licencie na použitie.

Ako hovorí názov, algoritmus vznikol fúziou *FAST* detektoru významných bodov a *BRIEF* deskriptora, ktorým vykonali mnoho modifikácii, aby dosiahli spomínané ciele. V prvom kroku použije *FAST* na detekciu významných bodov, na ktoré následne aplikuje *Harrisovú rohovú mieru (Harris corner measure)*, aby z týchto bodov našlo $N$ najlepších. Tu nastáva ale problém, že *FAST* nepočíta orientácie bodov a preto nezaručuje odolnosť voči rotácii obrázka. Autori prišli s modifikáciou, kde sa počíta ťažisko *patch* oblasti násobené intenzitou s rohom umiestneným v strede. Smer vektora od tohto rohového bodu k ťažisku dáva výslednú orientáciu.

Následne sa vytvoria deskriptory k významným bodom pomocou *BRIEF* algoritmu. Samotný *BRIEF* má zlé výsledky pri rotácii. Aby sa zvýšila presnosť, tak najprv vypočítajú maticu rotácie pomocou orientácie *patch* oblasti a následne sú *BRIEF* deskriptory pootočené pomocou tejto matice.

## 4.4 Porovnanie techník na detekciu významných bodov

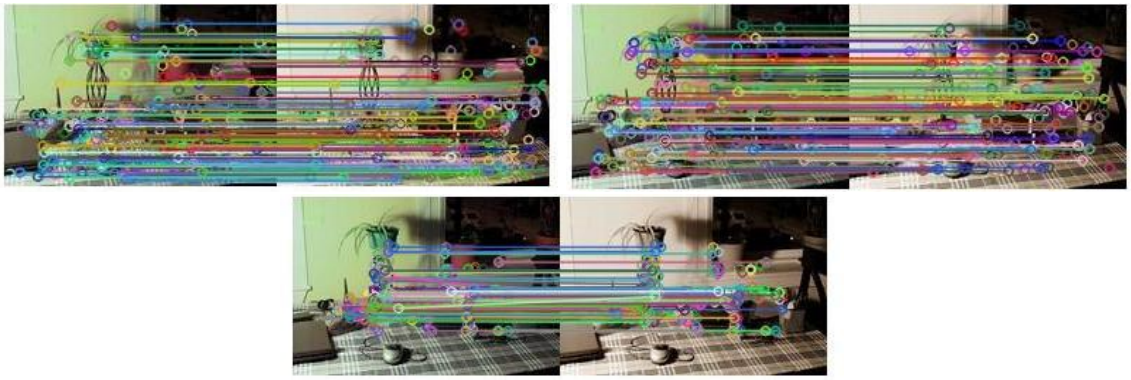
V článku vydanom v roku 2017 [23] porovnali autori 3 techniky na detekciu významných bodov: *SIFT*, *SURF* a *ORB*. Zamerali sa na ich výkonnosť a odolnosť voči rotácii, škálovaniu a rôznym deformitám ako napríklad *fish-eyes*.

### 4.4.1 Intenzita

V tejto časti autori testovali obrázky, ktoré mali rozličné intenzity a zloženie farieb. Výsledky testu sú v tabuľke 4.1 a znázornene na obrázku 4.2. Ako môžeme vidieť, *SIFT* má najväčší počet zhôd významných bodov, zatiaľ čo *ORB* najmenej. Naopak ale *ORB* mal najmenší čas výpočtu, zatiaľ čo *SIFT* najväčší.

| | Čas (sek) | Výz. b. 1 | Výz. b. 2 | Zhody | Percento zhody |
|---|---|---|---|---|---|
| SIFT | 0.13 | 248 | 229 | 183 | 76.7 |
| SURF | 0.04 | 162 | 166 | 119 | 72.6 |
| ORB | 0.03 | 261 | 267 | 168 | 63.6 |

Tabuľka 4.1: Výsledky porovnania obrázkov s rozličnými intenzitami [23].



Obr. 4.2: Zhody významných bodov v obrázkoch s rozličnými intenzitami. Zľava: *SIFT*, *SURF*, *ORB* [23].

### 4.4.2 Rotácia

V tejto časti testovali obrázok, ktorý bol rotovaný o 45 stupňov. Výsledky sú v tabuľke 4.2. Podobne ako v predchádzajúcom teste *SIFT* dosahuje najväčšie percento zhôd významných bodov, zatiaľ čo *ORB* vyžaduje najmenší výpočtový čas.

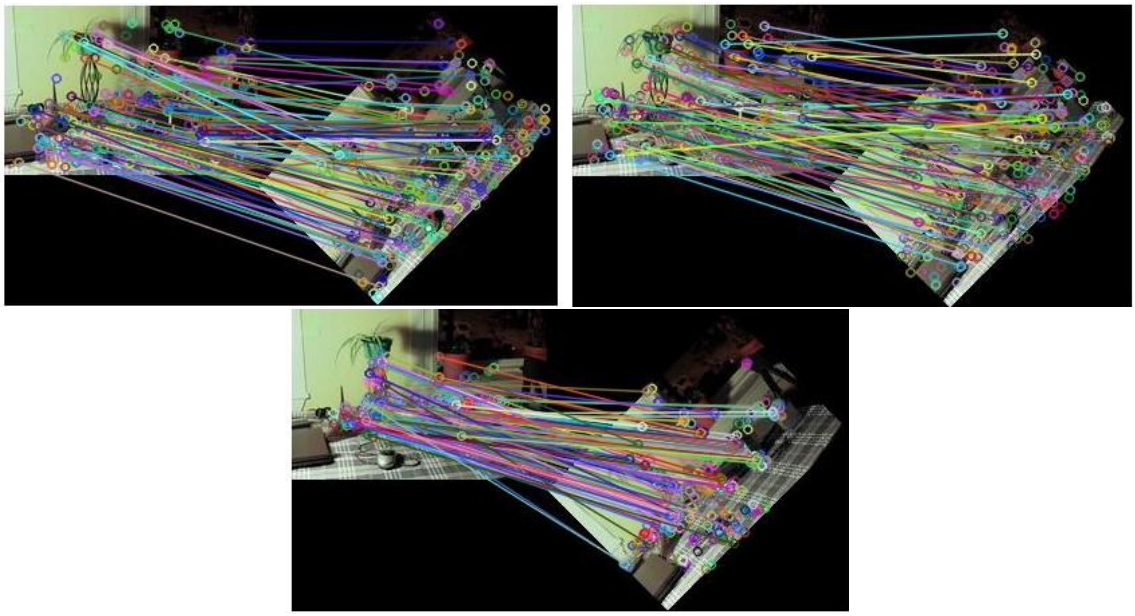
Tabuľka 4.3 reprezentuje percento zhody významných bodov naprieč obrázkami 4.3, ktoré sú otočené o zadané uhly. Z tejto tabuľky môžeme vidieť, že jedine pri 90 stupňoch poskytujú *SURF* a *ORB* lepšie percento zhody, zatiaľ čo pri všetkých ostatných hodnotách prevažuje *SIFT*.

| | Čas (sek) | Výz. b. 1 | Výz. b. 2 | Zhody | Percento zhody |
|---|---|---|---|---|---|
| SIFT | 0.16 | 248 | 260 | 166 | 65.4 |
| SURF | 0.03 | 162 | 271 | 110 | 50.8 |
| ORB | 0.03 | 261 | 423 | 158 | 46.2 |

Tabuľka 4.2: Výsledky porovnania obrázkov rotovaných o 45 stupňov [23].

| Uhol → | 0° | 45° | 90° | 135° | 180° | 225° | 270° |
|---|---|---|---|---|---|---|---|
| SIFT | 100 | 65 | 93 | 67 | 92 | 65 | 93 |
| SURF | 99 | 51 | 99 | 52 | 96 | 51 | 95 |
| ORB | 100 | 46 | 97 | 46 | 100 | 46 | 97 |

Tabuľka 4.3: Tabuľka percent zhôd pri porovnaní obrázkov rotovaných o rôzne stupne [23].



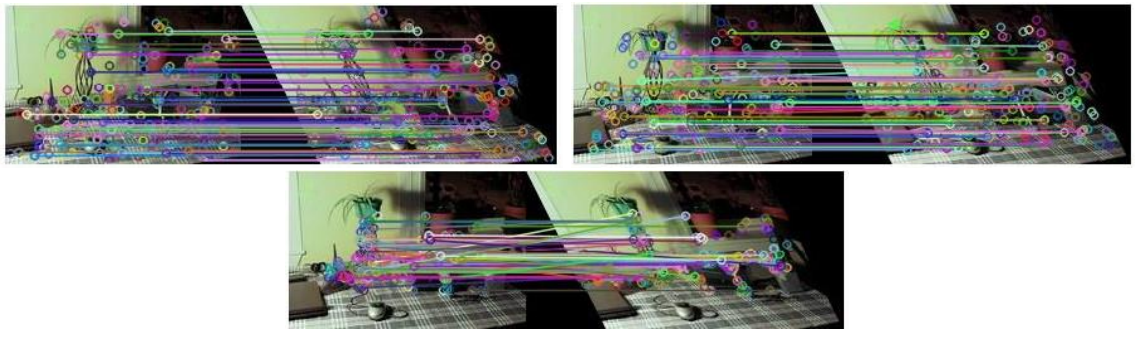
Obr. 4.3: Zhody významných bodov v otočených obrázkoch. Zľava: *SIFT*, *SURF*, *ORB* [23].

### 4.4.3 Skosenie

Originálny obrázok bol skosený hodnotou 0.5, aby sa zistila reakcia techník na skosenie obrázkov. Skosenie môžeme vidieť na obrázku 4.4. Výsledky sú v tabuľke 4.4. Výsledok testu je podobný ako doteraz – *SIFT* najväčšie percento zhody a *ORB* najrýchlejší.

| | Čas (sek) | Výz. b. 1 | Výz. b. 2 | Zhody | Percento zhody |
|---|---|---|---|---|---|
| SIFT | 0.133 | 248 | 229 | 150 | 62.89 |
| SURF | 0.049 | 162 | 214 | 111 | 59.04 |
| ORB | 0.026 | 261 | 298 | 145 | 51.88 |

Tabuľka 4.4: Výsledky porovnania skosených obrázkov [23].



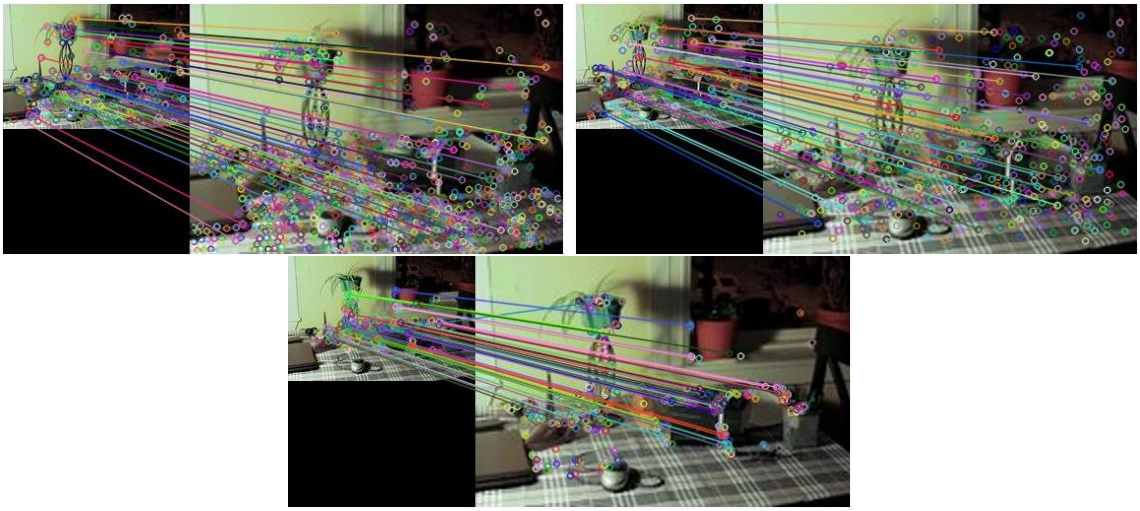
Obr. 4.4: Zhody významných bodov v skosených obrázkoch. Zľava: *SIFT*, *SURF*, *ORB* [23].

### 4.4.4 Škálovanie

V tejto časti došlo k dvojnásobnému zväčšeniu obrázku 4.5 a následne testovali presnosť zhôd. Výsledky sú v tabuľke 4.5. V tejto časti jednoznačne dominuje *ORB* rýchlosťou aj počtom zhôd. Najhoršie v obidvoch meraniach je na tom *SIFT*.

| | Čas (sek) | Výz. b. 1 | Výz. b. 2 | Zhody | Percento zhody |
|---|---|---|---|---|---|
| SIFT | 0.25 | 248 | 1210 | 232 | 31.8 |
| SURF | 0.08 | 162 | 581 | 136 | 36.6 |
| ORB | 0.02 | 261 | 471 | 181 | 49.5 |

Tabuľka 4.5: Výsledky porovnania dvojnásobne zväčšených obrázkov [23].



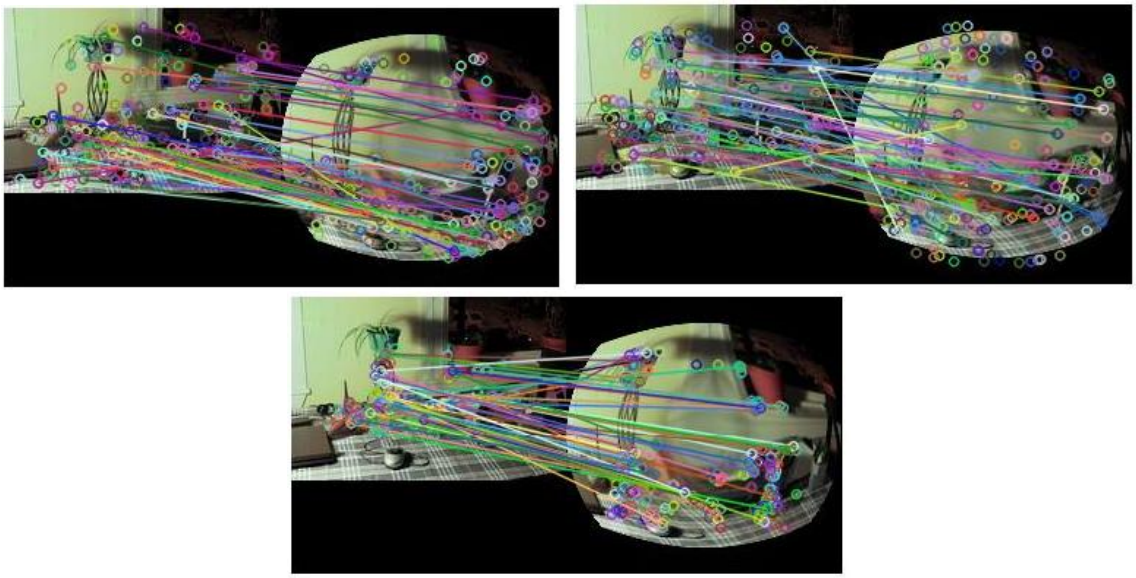
Obr. 4.5: Zhody významných bodov v obrázkoch s rozličným rozmerom. Zľava: *SIFT*, *SURF*, *ORB* [23].

### 4.4.5 Fish-eye

V tomto teste sa autori zamerali na obrázky 4.6, ktoré boli deformované *fish-eye* efektom, ktorý sa často vyskytuje v lacných kamerách. Výsledky sú v tabuľke 4.6. Výsledok je podobný ako v predchádzajúcich testoch: *SIFT* dosiahol najväčšie percento zhôd, zatiaľ čo *ORB* bol najrýchlejší.

| | Čas (sek) | Výz. b. 1 | Výz. b. 2 | Zhody | Percento zhody |
|---|---|---|---|---|---|
| SIFT | 0.132 | 248 | 236 | 143 | 59.09 |
| SURF | 0.036 | 162 | 224 | 85 | 44.04 |
| ORB | 0.012 | 261 | 282 | 125 | 46.04 |

Tabuľka 4.6: Výsledky porovnania zdeformovaných obrázkov efektom *fish-eye* [23].



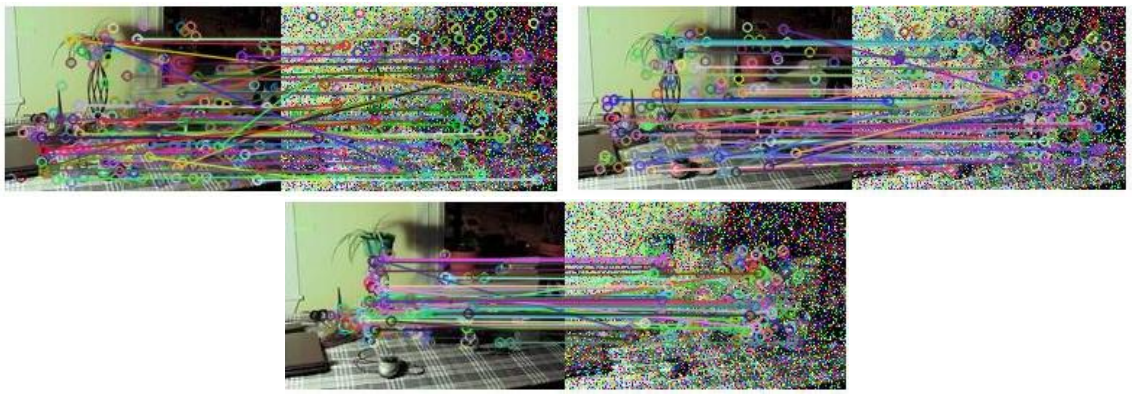
Obr. 4.6: Zhody významných bodov vo *fish-eye* obrázkoch. Zľava: *SIFT*, *SURF*, *ORB* [23].

### 4.4.6 Šum

Pri testovaní šumu v obrázkoch autori pridali 30% *salt and pepper* šum do obrázku 4.7, aby zistili ako ovplyvní testované metódy. Z tabuľky 4.7 môžeme vidieť, že *SIFT* a *ORB* vykazujú najväčšie percento zhôd, pričom *ORB* je zase najrýchlejšia metóda.

| | Čas (sek) | Výz. b. 1 | Výz. b. 2 | Zhody | Percento zhody |
|---|---|---|---|---|---|
| SIFT | 0.115 | 248 | 242 | 132 | 53.8 |
| SURF | 0.059 | 162 | 385 | 108 | 39.48 |
| ORB | 0.027 | 261 | 308 | 155 | 54.48 |

Tabuľka 4.7: Výsledky porovnania obrázkov s pridaným šumom [23].



Obr. 4.7: Zhody významných bodov v obrázkoch so šumom. Zľava: *SIFT*, *SURF*, *ORB* [23].

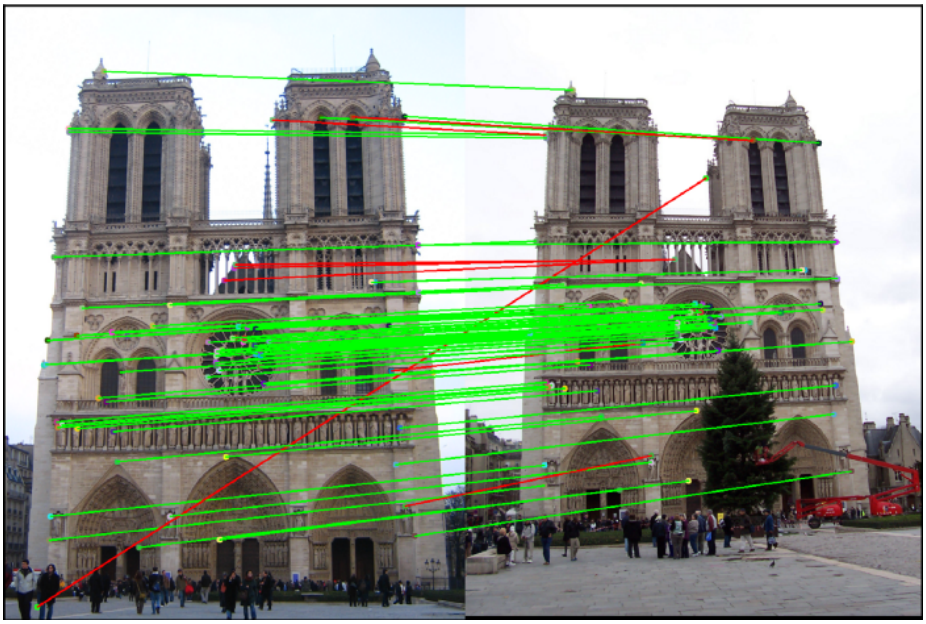
### 4.4.7 Vyhodnotenie testov

Z jednotlivých testov je možné vydedukovať, že najrýchlejší algoritmus je jednoznačne *ORB*, ktorý vo väčšine prípadov dokonca prekonáva v presnosti *SURF* algoritmus. Čo sa týka percenta zhôd, tak vo väčšine vyhráva *SIFT*. Táto presnosť je však na úkor rýchlosti, ktorá môže byť v menej výkonných alebo mobilných zariadeniach rozhodujúca.

Ako je vidieť zo všetkých obrázkov, významné body *ORB* algoritmu bývajú koncentrované v strede objektu, čo môže spôsobiť určité nepresnosti na rozdiel od *SURF* alebo *SIFT* algoritmu, kde tieto významné body sú distribuované po celom obrázku.

## 4.5 Hľadanie zhodných významných bodov

Hľadanie zhodných významných bodov je súčasťou mnohých aplikácii v počítačovom videní ako napríklad zarovnanie obrázkov alebo kalibrácia kamery. Úlohou tejto časti je nájsť zhodné významné body, ktoré boli lokalizované niektorou metódou uvedenou vyššie naprieč dvomi rozličnými obrázkami, ktoré zachytávajú rovnakú scénu alebo objekt. Na to, aby sme spoľahlivo a čo najpresnejšie dokázali hľadať zhodné významné body, je nutné zvoliť kvalitný detektor významných bodov a deskriptorov na konkrétny prípad. Na obrázku 4.8 môžeme vidieť čiarou prepojené zhodné významné body v dvoch rozličných obrázkoch budovy. V knižnici `OpenCV` je možné využiť dva algoritmy na hľadanie zhodných bodov [24]: *Brute-Force* a *FLANN*.



Obr. 4.8: Zhodné významné body z dvoch rozličných obrázkov rovnakej budovy prepojené čiarou[5].

[5]Prevzaté z https://medium.com/data-breach/introduction-to-feature-detection-and-matching-65e27179885d

### Brute-Force Matcher

Princíp tohto algoritmu je jednoduchý. Vyberie deskriptor jedného významného bodu z prvého obrázka a vypočíta vzdialenosť od každého bodu v druhom obrázku. Potom použije ten, ktorý má najmenšiu vzdialenosť. Ako vyplýva z princípu fungovania, tento algoritmus nájde vždy najoptimálnejšie riešenie, ale za cenu nižšej výkonnosti, keďže musí vypočítať vzdialenosť medzi každým bodom v prvom aj druhom obrázku.

Na výpočet vzdialenosti sa pri *SIFT* a *SURF* algoritmoch odporúča využívať *Euklidovskú* vzdialenosť *L2*, zatiaľ čo pre techniky ako *ORB*, *BRISK* a *BRIEF* sa odporúča *Hammingova* vzdialenosť.

### FLANN Matcher

**Fast Library for Approximate Nearest Neighbors** je zbierka algoritmov, ktoré sú optimalizované na rýchle hľadanie susedov v dátach. Je oveľa rýchlejší ako *Brute-Force*, ale ako jeho názov hovorí, najbližších susedov odhaduje len približne a nemusí nám poskytnúť vždy ten najlepší výsledok ako *Brute-Force*.

## 4.6 Nájdenie homografie medzi obrázkami

Nájdenie homografie je dôležitým krokom pri zarovnávaní obrázkov, pretože na jej základe vykonávame transformácie obrázkov a tým získavame výsledný zarovnaný obrázok. Homografia je vlastne perspektívna transformácia, pri ktorej mapujeme body z jednej roviny na druhú. Homografia je reprezentovaná maticou veľkosti $3 \times 3$. Aplikovaním matice na bod $(x, y)$ v prvom obrázku dostaneme bod $(x', y')$ v druhom obrázku [25]:

$$s \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = H \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}. \tag{4.1}$$

Nájsť maticu homografie naprieč dvomi obrázkami, z ktorých máme už nájdené aspoň 4 zhodné významné body, môžeme pomocou algoritmu *RANSAC*. Potom touto maticou môžeme jednotlivé obrázky zarovnať.

### RANSAC

**Random sample consensus** je algoritmus pomerne starý a to z roku 1981 [26]. Jedná sa o iteratívnu metódu, ktorá sa vo vstupných dátach, ktoré sú tvorené množinou príslušných bodov, snaží nájsť matematický model. Algoritmus hľadá najlepšie riešenie až kým nedosiahne stanovený počet iterácii alebo nedosiahne zadané percento presnosti. Čím vyšší počet iterácii, tým máme väčšiu pravdepodobnosť na nájdenie lepšieho výsledného modelu.

Ako vyplýva z názvu, algoritmus na výpočet vyberá náhodne $n$ hodnôt zo všetkých vstupných hodnôt, preto je najlepšie mať čo najviac vstupných dát. Algoritmus vychádza z predpokladu, že vstupné dáta sa dajú rozdeliť do:

- *Inliers* – vstupné dáta, ktoré patria do nájdeného modelu.
- *Outliers* – vstupné dáta, ktoré nepatria do nájdeného modelu. Sú to napríklad chybové hodnoty spôsobené šumom alebo zlým meraním.

# Kapitola 5

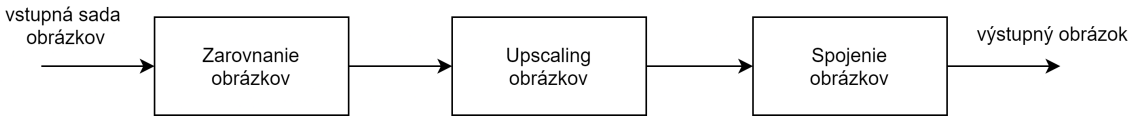
# Návrh a testovanie procesu vytvárania kvalitných fotiek

V tejto prvej časti návrhu sa budem venovať návrhu procesu spracovania obrázkov tak, aby sme dostali vysoko kvalitné výsledné obrázky a testovaním ich výslednej kvality.

Na dosiahnutie čo najkvalitnejšieho výsledku je potrebné použiť čo najkvalitnejšie prístupy a formáty. Tieto najkvalitnejšie prístupy sú ale za cenu rýchlosti, ktorá môže byť kľúčová pri použití na mobilných zariadeniach. Znázornenie tohto procesu spracovania môžeme vidieť na obrázku 5.1 a je navrhnutý následovne:

- vstupné obrázky na ich spojenie je potrebné ich najprv zarovnať, aby sa zachytený objekt nachádzal v rovnakom mieste naprieč všetkými vstupnými obrázkami,
- po zarovnaní obrázkov môžeme týmto obrázkom zvýšiť rozlíšenie na zlepšenie kvality,
- tieto zarovnané a prípadne zväčšené obrázky spojíme nejakou metódou, ktorá využije dáta zo všetkých obrázkov a vytvorí tak čo najkvalitnejší výsledok a pri tom odstráni rôzne nedokonalosti.

Vstupom bude sada obrázkov rovnakého rovinného objektu, ako napríklad karta alebo časť obalu, ktorá prejde určitými krokmi spracovania a výsledkom bude jeden obrázok v čo najvyššej kvalite. Pre najpresnejšie výsledky je najlepšie, aby daný rovinný objekt bol zachytený na jednofarebnom pozadí bez vzorov, pretože toto môže ovplyvniť detekciu významných bodov, prípadne nafotiť objekt tak, že na celej fotke je vidieť len ten objekt. Aby nedochádzalo ku strate kvality už v prvom kroku, bolo by najlepšie, keby spracovávané obrázky využívajú nejaký bezstratový formát ako napríklad *PNG*. Táto sada vstupných obrázkov nasleduje do ďalšej časti spracovania – zarovnanie. Ukážkové vstupné dáta môžeme vidieť na obrázku 5.2.




Obr. 5.1: Znázornenie poradia spracovania obrázkov.


Obr. 5.2: Ukážka vstupných fotiek rovinného objektu (karty) v bezstratovom formáte *PNG*. Karta je odfotená z rôznych pozícii za použitia blesku.

## 5.1 Zarovnanie obrázkov

Na zarovnanie rovnakého objektu zachyteného na viacerých fotografiách je nutné vykonať 3 kroky: nájsť význačné body v každom obrázku rovnakou metódou, nájsť spoločné významné body naprieč obrázkami a následne vypočítať homografiu potrebnú na transformovanie obrázka a tým ho zarovnať.

Pri využívaní detektorov významných bodov budem vychádzať hlavne z kapitoly 4.3. V tejto kapitole sme sa dozvedeli, že ak chceme detegovať body invariantné voči väčšine transformáciám, je nutné využiť metódy, ktoré k významným bodom vytvárajú aj deskriptory. Pri používaní detektorov vychádzam z porovnania v kapitole 4.4. Z kapitoly sme sa dozvedeli, že *SIFT* patrí k najpresnejším, *ORB* k najrýchlejším a *SURF* niekde v strede. *SURF* ale bohužiaľ použiť nemôžem, pretože je nutné zakúpiť licenciu na jeho použitie. Pri vyhľadávaní významných bodov je potrebné na vstupných obrázkoch dodržať to, aby to neboli jednoduché objekty, na ktorých sa ťažko hľadajú významné body.

Podobne je to aj pri hľadaní spoločných významných bodov (opísané v kapitole 4.5). *BFMatcher* by mal poskytnúť najlepší výsledok za cenu rýchlosti, zatiaľ čo *FLANN* je rýchlejší za cenu presnosti. Práve touto výkonnosťou je potrebné sa zaoberať na mobilných zariadeniach, aby celý proces netrval nadmerne dlho a dostali sme čo najlepšiu kvalitu.

Po tom ako nájdeme zhodné významné body v obrázkoch, je vhodné ešte odfiltrovať najhoršie zhody. To sa dá vykonať spôsobom, že si jednotlivé zhody zoradíme podľa vzdialenosti medzi jednotlivými bodmi a vymažeme určité percento hodnôt s najväčšou vzdialenosťou.

Posledným krokom je nájdenie homografie na základe spoločných významných bodov pomocou algoritmu *RANSAC*. Po nájdení tejto homografie ju môžeme aplikovať na jednotlivé obrázky a tým vykonať zarovnanie.

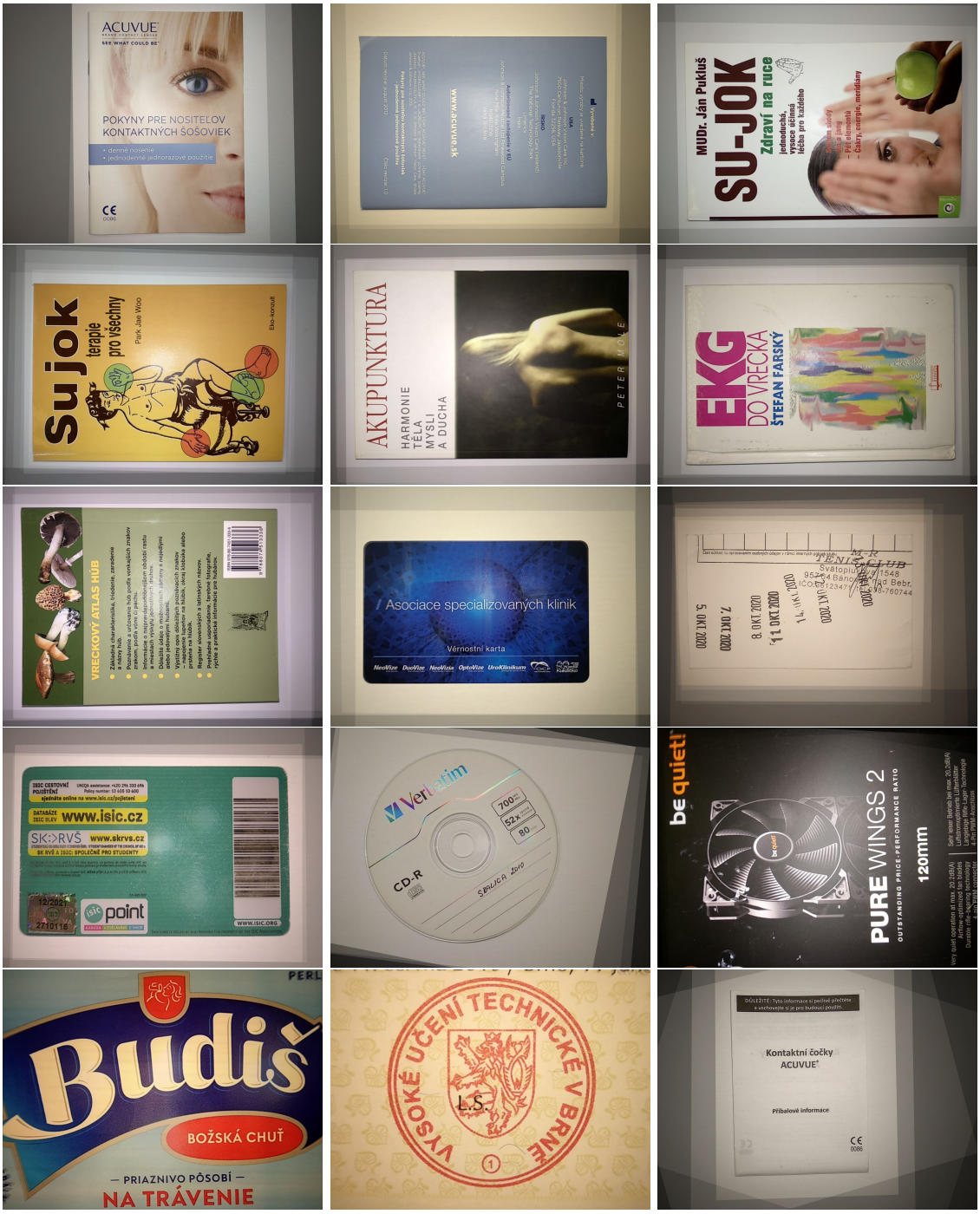
### 5.1.1 Experimentovanie so zarovnávaním

Na experimentovanie som si vytvoril skripty v jazyku `Python`, aby sa mi jednoduchšie testovalo a prípadne upravovalo. Všetky testy prebiehali na notebooku s procesorom *Intel Core i5 6300HQ* a 8GB pamäte na *Ubuntu 20.04 LTS*. Na testovanie času som použil knižnicu `time`. Všetky vstupné obrázky boli vytvorené v bezstratovom formáte *PNG* v rozlíšení $1440 \times 1080$.

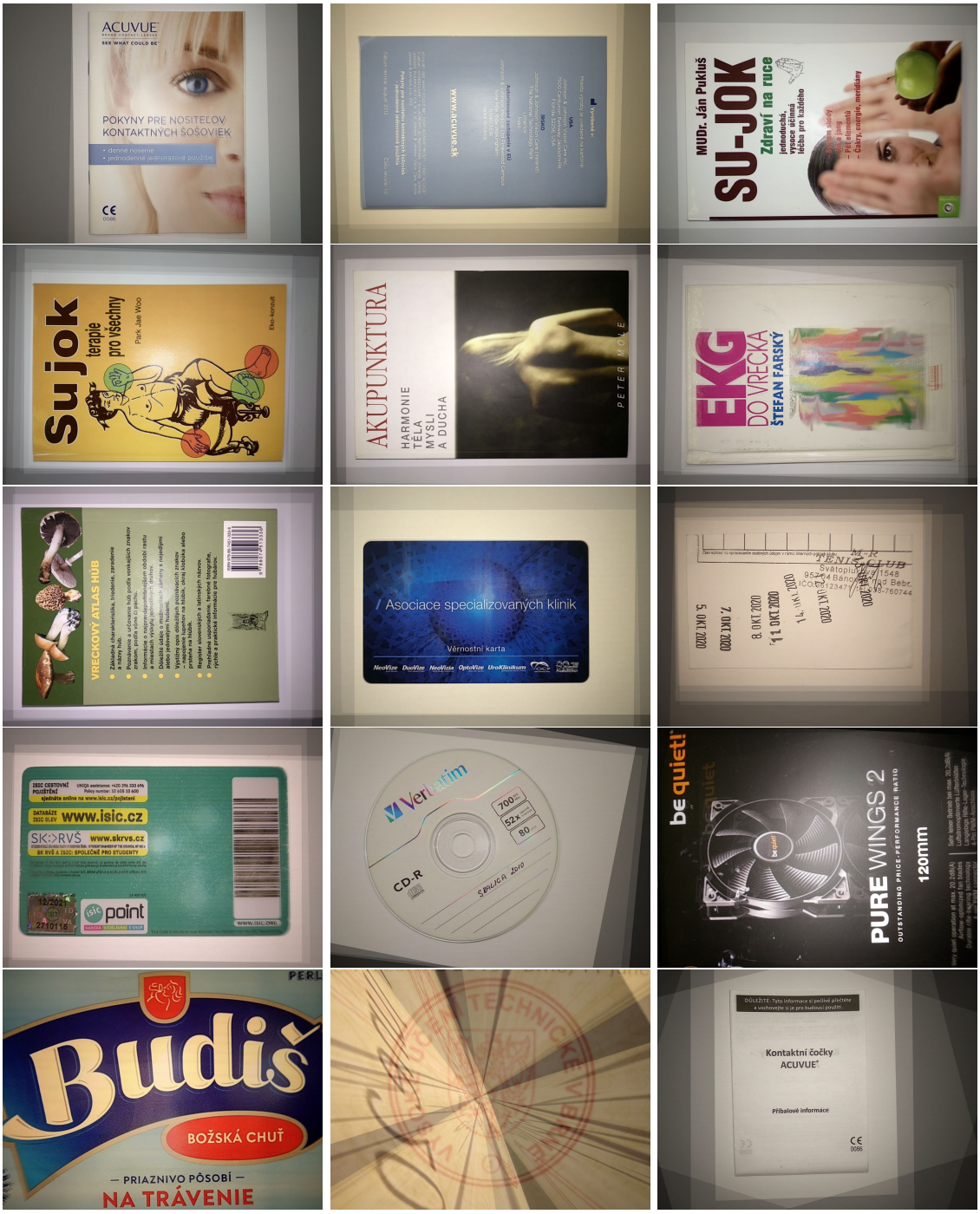
Pri testoch detektorov som použil ich odporúčané predvolené nastavenia a na hľadanie spoločných významných bodov som použil *BFMatcher*, aby som dostal čo najlepšiu kvalitu výstupu. Na hľadanie homografie bol použitý algoritmus *RANSAC*. Pri filtrovaní zhodných významných bodov bolo ponechaných 70% párov bodov s najmenšou vzdialenosťou. Pri hľadaní významných bodov vždy použijem prvý obrázok ako referenčný, ktorý sa nemení a všetky ostatné obrázky počítajú významné body voči nemu.

Meranie výkonnosti bolo spustené pred hľadaním významných bodov v obrázku, ktorý chceme zarovnať a ukončené po aplikovaní homografie, takže výsledky v sebe obsahujú aj čas potrebný na nájdenie spoločných významných bodov pomocou *BFMatcher* algoritmu, nájdenie homografie pomocou *RANSAC* a aplikovanie tejto homografie. Testy boli spustené nad rôznymi sadami obrázkov, kde každá sada zachytávala iný objekt. Tieto časy boli potom spriemerované.

Výsledky výkonnosti sú v tabuľke 5.1. Z hľadiska výkonnosti sa potvrdili výsledky z kapitoly 4.4. *ORB* je časovo omnoho rýchlejší a oveľa konzistentnejší. *SIFT* pri niektorých sadách produkuje výrazne vyššie priemerné časy oproti iným sadám.




Obr. 5.3: Ukážka spojených obrázkov zarovnaných pomocou *SIFT* obrázkov do jedného výsledného.




Obr. 5.4: Ukážka spojených obrázkov zarovnaných pomocou *ORB* do jedného výsledného.


Obr. 5.5: Vystrihnuté časti spojených zarovnaných obrázkov znázorňujúce chybu pri zarovnávaní pomocou *ORB* (ľavý stĺpec fotiek). *SIFT* v tomto prípade vykonal správne zarovnanie fotiek (pravý stĺpec). V prípade karty (posledný riadok) nevykonal ani *SIFT* úplne najlepšie zarovnanie.

| Názov | Priemerný čas zarovnania (sek) | |
|---|---|---|
| | ***SIFT*** | ***ORB*** |
| Kniha 1 | 0.5188 | 0.0423 |
| Kniha 2 | 0.8579 | 0.0442 |
| Kniha 3 | 0.5800 | 0.0434 |
| Kniha 4 | 0.7306 | 0.0455 |
| Kniha 5 | 0.7218 | 0.0441 |
| Kniha 6 | 0.4210 | 0.0411 |
| Kniha 7 | 0.6418 | 0.0445 |
| Kniha 8 | 1.8417 | 0.0498 |
| Karta 1 | 0.5690 | 0.0426 |
| Karta 2 | 0.7735 | 0.0455 |
| Karta 3 | 1.4308 | 0.0458 |
| CD | 0.5152 | 0.0420 |
| CD obal | 0.5658 | 0.0440 |
| Kindle | 6.4495 | 0.0486 |
| Papier | 0.4818 | 0.0416 |
| Obal | 1.8467 | 0.0524 |
| Etiketa | 0.9207 | 0.0496 |
| Razítko | 0.8431 | 0.0806 |

Tabuľka 5.1: Porovnanie priemerných časov zarovnania jedného snímku zo sady obrázkov rôznymi metódami.

Pri porovnaní kvality výsledkov som sa rozhodol použiť iný prístup ako autori testov opísaných v kapitole 4.4, pretože počet zhodných bodov reálne neznamená naozajstnú zhodu. Preto som sa rozhodol obrázky takto zarovnané spojiť do jedného obrázku takým spôsobom, že každý obrázok ma redukovaný *alfa* kanál, aby umožnil viditeľnosť aj iných obrázkov takto skladaných cez seba. Tieto spojené obrázky som následne ručne skontroloval a hľadal rôzne nedostatky v zarovnaní. Spojené obrázky môžeme vidieť na obrázkoch 5.3 a 5.4. Pri pohľade na tieto spojené obrázky bolo vidno nedostatky pri zarovnávaní pri *ORB* metóde. Ako vidíme na obrázku 5.5, pri *ORB* presvitajú časti obrázkov, ktoré sa nachádzajú mimo zarovnania ostatných obrázkov. V prípade razítka dokonca došlo k úplne zlému zarovnaniu. *SIFT* dáva oveľa kvalitnejšie výsledky zarovnania, ale tiež nie je úplne perfektné. V poslednom riadku fotiek na obrázku 5.5, *SIFT* pri zarovnaní karty tiež nevykonal úplne najlepšie zarovnanie. Tento nedostatok by išiel odstrániť použitím dostatočného počtu fotiek daného objektu a šikovnej metódy na spájanie obrázkov, ktorá by takéto nevyhovujúce zarovnania odstránila. Ďalšou možnosťou je ručné porovnanie jednotlivých snímkov a vyradenie takto nevyhovujúcich zarovnaných snímkov objektu.

## 5.2 Upscaling obrázkov

V tomto kroku je možnosť vykonať *upscaling* zarovnaných obrázkov na dvoj alebo trojnásobok. Keďže kamery v dnešných telefónoch robia snímky už v pomerne vysokých rozlíšeniach, tak nepovažujem tento krok za nutný. Ak by ale užívateľ aplikácie mal záujem mať výslednú fotku vo vyššom rozlíšení, tak sa môže vykonať *upscaling* zarovnaných obrázkov pomocou

*bilineárnej* alebo *bikubickej* interpolácie. *Upscaling* sa vykonáva v tomto kroku, pretože v prípade chýb môže dôjsť k ich odfiltrovaniu v spájaní.

## 5.3 Spojenie obrázkov

Po tom, čo už máme jednotlivé fotky objektu zarovnané a prípadne zväčšené, je potrebné vykonať posledný krok, ktorým je vytvorenie výsledného obrázku spojením všetkých vstupných fotografií do jednej. Vstupné obrázky boli zarovnané pomocou *SIFT*.

Prvá metóda spojenia, ktorá sa ponúka, je vytvorenie výsledného obrázku pomocou spriemerovania jednotlivých pixelov všetkých vstupných zarovnaných obrázkov. Výsledok takéhoto spojenia môžeme vidieť na obrázku 5.6. Ako je vidieť, toto spojenie nie je úplne najlepšie, pretože nedochádza k filtrovaniu prípadne zle zarovnaných obrázkov alebo šumu, ako napríklad odraz blesku.

Vylepšením tejto metódy je napríklad orezaný priemer (*truncated mean*), pri ktorom tiež dochádza k spriemerovaniu jednotlivých pixelov, ale až po tom, čo sú jednotlivé pixely v rovnakej pozícii v každom vstupnom zarovnanom obrázku usporiadané podľa intenzity a z tohto usporiadania je odstránené zadané percento najvyšších a najnižších hodnôt. Toto odstránenie extrémnych hodnôt spôsobí odstránenie nedostatkov, ako napríklad odraz blesku. Vďaka tomu máme výborne osvetlený objekt z každého uhla bez odrazu blesku.


Obr. 5.6: Spojenie obrázkov pomocou spriemerovania hodnôt (ľavý stĺpec) a orezaného spriemerovania (pravý stĺpec). Môžeme vidieť ostrejšie detaily a odstránenie fľakov od blesku.

# Kapitola 6

# Návrh a testovanie mobilnej aplikácie

V tejto časti sa budeme venovať návrhu samotnej aplikácie pre Android. Základnou myšlienkou aplikácie je, že užívateľ otvorí aplikáciu, vytvorí úlohu do ktorej nafotí objekt z rôznych uhlov a následne spustí spracovávanie. Keďže toto spracovávanie bude trvať dlhšiu dobu v závislosti od počtu fotiek, je vhodné, aby dostal notifikáciu pri ukončení spracovania. tiež by mal mať možnosť výsledné fotky z aplikácie zdielať, prípadne niekam uložiť.

Keďže proces spracovávania ma viacero medzikrokov a musíme vykonať jeden pred druhým a po jednotlivých krokoch prezrieť a overiť výsledok, potrebujeme v aplikácii rozdeliť fotografie na vstupné, zarovnané a výstupné. Vstupné fotografie sa dajú zachytiť priamo z kamery zariadenia. Potom užívateľ spustí zarovnanie týchto fotiek. Pred spustením nového zarovnávania fotiek je nutné, aby sa predchádzajúce fotografie zmazali, aby nedošlo k ich miešaniu. Ďalej užívateľ zarovnané fotografie skontroluje, vymaže nevhodné a na zvyšných spustí proces spájania. Na skontrolovanie zarovnania potrebuje nejaký šikovný spôsob porovnania jednotlivých fotiek v užívateľskom rozhraní. Výsledkom spojenia bude jedna fotka, ale užívateľ môže chcieť vytvoriť viacero takýchto výsledných fotografií rôznymi spôsobmi spájania, filtrovania a mal by dokázať jednotlivé výsledné fotografie porovnať, aby si vedel vybrať tu najkvalitnejšiu.

Keďže spracovanie fotografií je výpočtovo náročné a tiež pracujeme na mobilných zariadeniach, ktoré nemajú k dispozícii taký výkon a pamäť ako stolové počítače, je nutné väčšinu spracovania vykonať v jazyku `C++`, ktorý sa preloží do natívneho kódu a vďaka tomu poskytne oveľa väčšiu rýchlosť spracovania ako jazyk `Java`.

## 6.1 Prehľad existujúcich riešení

Tato kapitola popisuje podobné už existujúce aplikácie. Aplikácie, ktoré riešia konkrétne tento problém, sa mi nepodarilo nájsť. Našiel som ale niekoľko aplikácii, ktoré sa snažia vytvárať kvalitné verzie vstupných fotografií.

Aplikácie *Remini*[1] a *EnhanceFox*[2] využívajú umelú inteligenciu na zlepšenie kvality fotiek, napríklad odstránením šumu z fotografií. Vstupom je len jedna fotografia, na ktorej sa pracuje. V mojom testovaní ale výsledky týchto aplikácii neboli až tak dobré, pretože dosť často dochádzalo k strate detailov fotografií. Tiež nedokážu odfiltrovať nedostatky, ako

[1] https://play.google.com/store/apps/details?id=com.bigwinepot.nwdn.international
[2] https://play.google.com/store/apps/details?id=com.changpeng.enhancefox

napríklad blesk, keďže nemajú viaceré fotografie, z ktorých by mohli čerpať. Použitie týchto aplikácií tiež nie je veľmi dobré, pretože nútia užívateľa najprv registrovať sa a obsahujú veľké množstvo reklám.

Aplikácia *TensorZoom*[3] je vlastne demo aplikácia, ktorá pomocou knižnice *TensorFlow* implementuje neurónovú sieť, ktorá dokáže obrázok zväčšiť až 4× v každom smere. Toto zväčšovanie funguje tak, že obrázok je rozdelený na určitý počet regiónov pomocou mriežky a zväčšovanie konkrétnej mriežky začne až po maximálnom priblížení na daný región. Predpokladám, že je to takto urobené z dôvodu, že výpočet nie je práve najrýchlejší. Kvalita výstupu je ale oveľa lepšia ako v predchádzajúcich aplikáciach. Táto aplikácia podporuje iba zväčšovanie obrázkov na väčšie rozlíšenie a neposkytuje žiadne možnosti filtrovania.

## 6.2 Štruktúra aplikácie

V tejto kapitole sa budem venovať návrhu štruktúry mobilnej aplikácie. Popíšem tu výber verzie systému, použitý architektonický vzor a tiež aj návrh databázy.

### 6.2.1 Voľba verzie systému

Pri výbere minimálnej podporovanej verzie systému Android som sa musel riadiť podľa viacerých kritérií. Prvým kritériom bolo, aby vybraná verzia mala podporu pre *Camera2 API* opísané v kapitole 2.9.1. Táto podpora je povinná od verzie 5.0. V danej kapitole som tiež spomínal nedodržiavanie implementácie API od rôznych výrobcov, preto nie je vhodné voliť vždy hraničnú verziu.

Druhým kritériom je dostatočný výkon zariadenia. Spracovanie fotografií je dosť náročná úloha, ktoré nemusia staršie zariadenia úplne zvládať. Preto som sa snažil zvoliť verziu, ktorá nie je až tak stará, pretože výkonnosť zariadení sa výrazne zlepšuje z roka na rok.

Na základe týchto kritérií a rozloženia verzií systému som sa rozhodol zvoliť verziu 7.0 ako minimálne podporovanú verziu, pretože nejedná sa o veľmi starú verziu (v auguste bude 5 rokov stará) a podľa kapitoly 2.1 má 73.7% zariadení verziu väčšiu alebo rovnakú verziu systému ako 7.0. Musíme brať do úvahy, že toto percento bude reálne väčšie, pretože údaje o rozložení sú z minulého roku a rozloženie sa výrazne posúva k vyšším verziám každý rok.

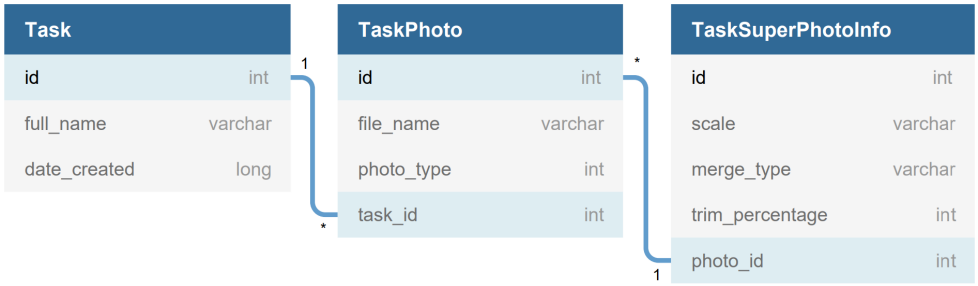
### 6.2.2 Architektúra aplikácie

Pri návrhu architektúry aplikácie som vychádzal z kapitoly 2.7. Aplikácia využíva *MVVM* architektúru, ktorá využíva komponenty *Model*, *Repository*, *View-Model* a *View*. Do rovnakých balíčkov sú podelené aj zdrojové kódy aplikácie, ktoré naviac obsahujú ešte balíčky *Utils* a *Worker*.

#### Model

V tejto časti sú objekty, ktoré zaručujú perzistentné dáta aplikácie. Jedná sa teda o databázu aplikácie. Na implementáciu databáze som použil knižnicu *Room*[4], vďaka ktorej je možné celý návrh databázy vykonať objektovo v jazyku `Java`. Navrhnutú databázu môžeme vidieť na ER diagrame na obrázku 6.1.

[3] https://play.google.com/store/apps/details?id=uk.tensorzoom
[4] https://developer.android.com/training/data-storage/room




Obr. 6.1: Schéma navrhnutej databázy pomocou ER diagramu.

Ako som uviedol v úvode kapitoly, aplikácia bude mať na začiatku zoznam úloh, ktoré po otvorení zobrazia konkrétnu úlohu, ktorá obsahuje všetky fotky. K výsledným fotografiám sú okrem iného ukladané aj informácie o vzniku fotografie. Po prevode týchto častí na entity dostávame 3 entity:

- **Task** – entita, ktorá obsahuje informácie o úlohe – jej názov a dátum vytvorenia.
- **TaskPhoto** – entita, ktorá obsahuje všetky fotografie patriace úlohe. Obsahuje cestu k súboru na úložisku, identifikátor, ktorý rozlišuje o aký typ fotografie sa jedná (fotografia z kamery, zarovnaná fotografia alebo výsledná fotografia) a cudzí kľúč, ktorý odkazuje na úlohu, do ktorej daná fotografia patrí.
- **TaskSuperPhotoInfo** – entita, ktorá obsahuje informácie o vzniku výslednej kvalitnej fotografie. Obsahuje informácie o type spojenia, zväčšenia a percenta orezania extrémnych hodnôt. tiež obsahuje cudzí kľúč, ktorý odkazuje na fotografiu, ku ktorej tieto údaje patria.

### Repository

V tomto balíčku sú triedy, ktoré pracujú s entitami databázy. Tieto triedy majú výhodu v tom, že vytvárajú jeden vstupný bod na prácu s dátami a v prípade zmeny stačí vykonať zmenu len v týchto triedach a nie v každom *View-Model*. Tieto triedy zaručujú vykonávanie úkonov nad dátami neblokujúcim spôsobom v pozadí aplikácie.

### View-Model

V tomto balíčku sú uložené samotné *View-Model* triedy, ktoré slúžia na prepojenie komunikácie *View* a *Repository* a tiež na ukladanie dát, ktoré ostávajú nemenné pri zmene rozloženia alebo rotácii zariadenia. Medzi takéto dáta patrí napríklad identifikátor aktuálne prehliadanej úlohy alebo obrázku.

### View

V tomto balíčku sú triedy, ktoré obsahujú samotné fragmenty a aktivity. Rozhodol som sa využiť prístup nazývaný *single-activity*, ktorý v aplikácii využíva len jednu základnú aktivitu, v ktorej sú zobrazované a menené fragmenty jednotlivých obrazoviek. Výhodu to má napríklad v jednoduchosti animovania týchto fragmentov alebo možnosti zdieľania dát naprieč fragmentami. Aby som si prácu s fragmentami ešte zjednodušil, využil som

knižnicu *Navigation Component*[5], ktorá je odporúčaná pri *single-activity* prístupe. Vďaka nej je jednoduché navigovať medzi jednotlivými fragmentami a predávať im vstupné dáta.

Na ďalšie zjednodušenie práce s jednotlivými prvkami fragmentov som využil knižnicu *View-Binding*[6], ktorá za programátora automaticky generuje triedy, ktoré uľahčujú prístup k jednotlivým prvkom fragmentov.

#### Utils

Balíček, v ktorom sú uložené triedy, ktoré pomáhajú pri programovaní ako napríklad formátovanie dátumu, prevod reťazcov na čísla.

#### Workers

V tomto balíčku sú obsiahnuté všetky triedy, ktoré zabezpečujú proces spracovania obrázkov v pozadí. Na zaručenie kompatibility som použil knižnicu *WorkManager*[7], ktorá za programátora rieši problém vykonávania úloh na pozadí naprieč rôznymi verziami systému.

## 6.3 Proces vytvárania výslednej fotky

V tejto kapitole popíšem návrh využitia procesu navrhnutého v kapitole 5 do mobilnej aplikácie na systém Android. Prvým krokom je navrhnutie a otestovanie spôsobu na vytváranie fotiek v bezstratovom formáte a následne z týchto fotografií vytvoriť výsledok pomocou už spomenutého procesu spracovania.

### 6.3.1 Vytváranie bezstratových fotiek

V tejto časti sa budem snažiť kompletne popísať návrh na vytváranie fotiek v bezstratovom formáte. Budem vychádzať z kapitoly 2.9. Najjednoduchšie a najviac kompatibilné medzi všetkými riešeniami by bolo využiť knižnicu *CameraX*, no bohužiaľ táto knižnica nepodporuje vytváranie fotiek v iných formátoch ako *JPEG*. Ostáva teda už len priamo *Camera2*, no keďže cieľom aplikácie nie je vytvoriť aplikáciu na fotografovanie, ktorá funguje na každom telefóne, bolo by vhodné nájsť riešenie, ktoré pracuje s *Camera2* a zároveň umožňuje vytvárať fotky v rôznych formátoch, prípadne by tam bolo možné túto funkcionalitu dopísať.

Po hľadaní som sa rozhodol použiť knižnicu **CameraView**[8] na základe toho, že je stále aktualizovaná, pomerne populárna a umožňuje vykonávať fotky vo formáte *JPEG* alebo *DNG*. Táto knižnica za nás okrem kompatibility rieši aj spravovanie povolení k prístupu ku kamere. Najjednoduchším riešením by bolo vytvárať fotky v bezstratovom formáte *DNG* a tie následne skonvertovať do *PNG*, no ako som spomínal v kapitole 2.9, nie každý výrobca telefónov plne implementoval *Camera2 API* a preto niektoré telefóny podporu vytvárania *DNG* nemajú. Nejedná sa len o staré telefóny. Napríklad v mojom telefóne *Xiaomi Mi A1* s Android 9 podpora *RAW* fotiek tiež chýba.

Jediným riešením ostáva už len takúto funkcionalitu do danej knižnice dopísať. Vďaka *Camera2* máme prístup priamo k pixelom zo senzoru vo formáte *YUV*. Tieto pixely môžeme pomocou výpočtov spomínaných v kapitole 3.1 skonvertovať na *RGB* pixely, ktoré následne už len uložíme do *PNG* súboru na úložisku. Po uložení na úložisko je ešte dôležité uložiť

[5] https://developer.android.com/guide/navigation/navigation-migrate
[6] https://developer.android.com/topic/libraries/view-binding
[7] https://developer.android.com/topic/libraries/architecture/workmanager
[8] https://github.com/natario1/CameraView

*EXIF* metadáta o rotácii telefónu do snímku, pretože bez tejto informácie by sa snímok vždy uložil v rovnakom smere nezávisle na rotácii telefónu.

Po otestovaní tejto funkcionality je realita trochu horšia. Aj po tom, čo som celý prepočet prepísal do natívneho `C++` kódu (bližšie popísané v kapitole 7.2.1), trvá vytváranie takýchto fotiek dlho. Môj telefón *Xiamo Mi A1* vytvára fotografie v rozlíšení $1440 \times 1080$ a prepočet z *YUV* do *RGB* vrátane zakódovania do *PNG* formátu trvá približne 5 sekúnd. Tento čas postupne rastie na telefónoch, ktoré dokážu vytvárať fotky ešte vo vyšších rozlíšeniach. Túto skutočnosť som skúšal overiť ešte na telefónoch *Xiaomi Mi A3* a *Samsung Galaxy S9+*, ktoré vytvárali fotky v rozlíšení $4000 \times 3000$ a prepočet a zakódovanie trvali približne 12 sekúnd.

Na základe tohto zistenia som sa rozhodol pridať do nastavení aplikácie možnosť, ktorá dovolí užívateľovi vybrať, či chce vytvárať fotografie v bezstratovom formáte *PNG*, ale za cenu dlhšieho vytvárania fotky alebo vo formáte *JPEG*.

### 6.3.2 Spracovanie fotografií

Ako som uviedol na začiatku kapitoly, tak pri návrhu budem využívať proces navrhnutý v kapitole 5. Prvým dôležitým krokom je zvoliť vhodnú knižnicu na implementáciu procesu do aplikácie na systém Android. Google ponúka vo svojom *Mobile Vision API*[9] mnoho funkcionalít, ako napríklad detekcia objektov, no žiadna z týchto funkcií nie je aplikovateľná na náš problém. Populárna knižnica `OpenCV`, ktorú som využíval aj na prvotný návrh a testovanie, je tiež dostupná aj pre systém Android. Výhodou knižnice je, že je napísaná v jazyku `C++` čo znamená, že je oveľa rýchlejšia ako hociktorá `Java` knižnica. Je aj ľahko využiteľná, pretože k `C++` kódu poskytuje aj *Java Native Interface*, vďaka ktorému môžeme jednoducho volať akcelerované metódy priamo v jazyku `Java`. `OpenCV` je tiež veľmi dobre zdokumentovaná a existuje k nej veľa príkladov. Jej nevýhodou ale je, že aplikácia má po skompilovaní veľkú veľkosť a je potrebné ju kompilovať konkrétne pre rôzne architektúry procesorov. Túto knižnicu som sa rozhodol vybrať aj preto, lebo som v nej už mal otestovaný celý proces spracovania v jazyku `Python`.

Pri rozhodovaní ktoré algoritmy na zarovnanie a spájanie fotografií v aplikácii použiť, som sa rozhodol ponechať používateľom čo najväčší výber a kontrolu nad algoritmami. To znamená, že užívateľ pred spustením zarovnávania obrázkov si bude môcť zvoliť, aký algoritmus sa má použiť na detekciu významných bodov (*SIFT* alebo *ORB*), na nájdenie spoločných významných bodov (*BFMatcher* alebo *FLANN*) a tiež aj možnosť zvoliť percento, ktoré určí, koľko najlepších nájdených zhodných významných bodov sa ponechá na výpočet homografie. Homografia bude počítaná pomocou *RANSAC* algoritmu. Ak užívateľ nechce nad takýmito vecami rozmýšľať, tak sú v aplikácii prednastavené predvolené hodnoty: detektor *SIFT* a *BFMatcher* na dosiahnutie najlepšej kvality zarovnania a ponechanie 70% najlepších párov významných bodov.

V prípade spájania fotografií som tiež ponechal užívateľovi voľnú ruku, kde výslednú fotografiu môže vytvoriť obyčajným spriemerovaním hodnôt alebo môže využiť kvalitnejší orezaný priemer, pri ktorom si môže zvoliť aj percento orezaných hodnôt z oboch strán extrémov. Podobne môže zvoliť aj to, či chce zarovnané obrázky zväčšiť. Na výber má *bilineárne* alebo *bikubické* zväčšenie $2\times$ alebo $3\times$ v každej osi. Toto vytváranie výslednej fotografie bolo potrebné vykonať v jazyku `C++`, pretože spracovanie v jazyku `Java` by bolo veľmi pomalé.

[9] https://developers.google.com/vision

Pri testovaní tohto procesu na telefóne *Xiaomi Mi A1* sa mi potvrdili závery z testovaní ako v kapitole 4.4 a 5.1.1: *ORB* je niekoľko násobne rýchlejší zatiaľ čo *SIFT* omnoho kvalitnejší. Pri testovaní zarovnávania som sa stretol s problémom, ktorý sa pri testovaní na počítači nevyskytol. Problémom bolo dochádzanie operačnej pamäte pri detekcii významných bodov pomocou *SIFT* pri fotografiách, ktoré mali väčšie rozlíšenie. Aplikácia sa v tom prípade bez nejakého upozornenia len vypla. Tuto skutočnosť som skúsil otestovať na oveľa výkonnejšom telefóne *Samsung Galaxy S9+*, na ktorom táto detekcia identických fotografií prebehla bez problémov. Dôvodom je hlavne väčšia operačná pamäť zariadenia. Na vyriešenie problému som sa rozhodol pridať užívateľovi pri voľbe algoritmov zarovnania možnosť zvoliť, či majú byť vstupné obrázky pred vstupom do algoritmu na detekciu zmenšené na polovičnú veľkosť, aby sa predišlo vyčerpaniu operačnej pamäti. Vypočítaná homografia je ale aplikovaná na fotografie s originálnym rozlíšením, pretože zmena rozlíšenia nemá vplyv na detekciu významných bodov.

## 6.4 Užívateľské rozhranie

Veľmi dôležitou časťou aplikácie je samotné užívateľské rozhranie. Myšlienka je taká, že užívateľ po spustení aplikácie vytvorí novú úlohu, do ktorej nafotí objekt, ktorý chce spracovať a následne spustí samotné zarovnanie a spojenie fotiek. Na základe tejto myšlienky môžeme aplikáciu rozvrhnúť do viacerých častí: zoznam úloh, konkrétna úloha obsahujúca vstupné, zarovnané a výsledné fotky, aktivita na vytváranie fotiek v konkrétnej úlohe, aktivity na spustenie úloh spracovania, aktivita na podrobnejšie prezeranie fotografií a notifikácie, ktoré informujú o procese spracovania.

Pri návrhu užívateľského rozhrania som sa rozhodol nenačrtávať žiadne *wireframe*, ale rovno vytvoriť prvé verzie UI v `XML` za použitia editora rozloženia UI vo vývojovom prostredí *Android Studio*, ktoré som následne naplnil ukážkovými dátami a využíval na testovanie na užívateľoch. Tieto prvotné funkčné verzie potom bolo jednoduchšie prerobiť na základe výsledkov testovania, ako načrtnuté *wireframe* nanovo prerábať a následne prepisovať do `XML`.

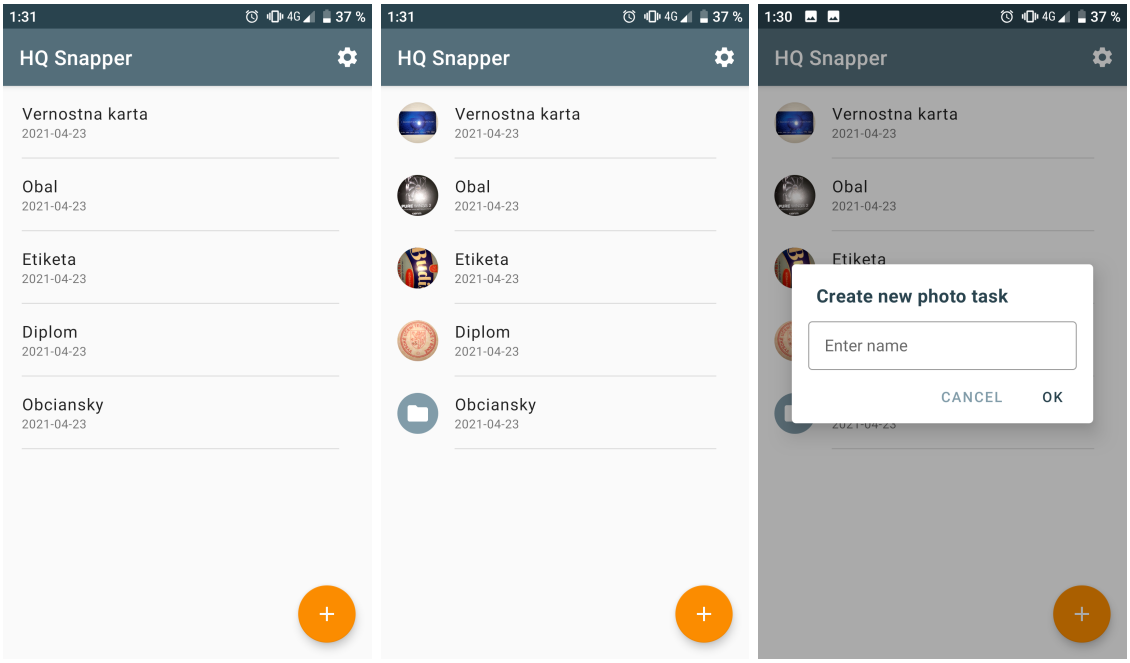
Na dizajn užívateľského rozhrania som sa rozhodol nevymýšľať nič nové, pretože by to v konečnom dôsledku nevyzeralo tak dobre ako už navrhnuté dizajny. Preto som sa rozhodol použiť Androidom odporúčaný **Material Design**, ktorý v svojej dokumentácii[10] obsahuje rôzne ukážky a pravidlá, aby sme vo výsledku dostali kvalitný dizajn.

### 6.4.1 Zoznam úloh

Dôvodom na vytvorenie separátnej úlohy pre každú sadu fotiek je hlavne to, že každá úloha bude obsahovať množstvo fotiek, ktoré musia byť nejak kategorizované, pretože by v nich pri veľkom počte úloh bol zmätok. Pre vytvorenie úlohy je od užívateľa potrebné zadať iba názov úlohy. Ku každej novo vytvorenej úlohe sa následne priradí dátum jej vytvorenia. Prvotný návrh zoznamu úloh a dialógu je na obrázku 6.2. Jedná sa o jednoduchý zoznam, kde každý prvok zoznamu obsahuje názov a čas vytvorenia úlohy. V pravom dolnom rohu je jasne zvýraznené tlačítko, ktorým sa vytvára dialóg na vytvorenie novej úlohy.

Pri užívateľskom testovaní užívatelia nemali problém pochopiť systém úloh a bez problémov dokázali vytvoriť novú úlohu. Problém s použitím užívatelia začínali mať v prípade, kedy v tomto zozname bolo veľa úloh a už sa v nich nedokázali jednoznačne orientovať

[10] https://material.io/develop/android




Obr. 6.2: Aktivita zobrazujúca zoznam vytvorených úloh. Zľava: prvotná verzia zoznamu pred testovaním bez miniatúr, zoznam po pridaní miniatúr, dialóg na vytvorenie novej úlohy.

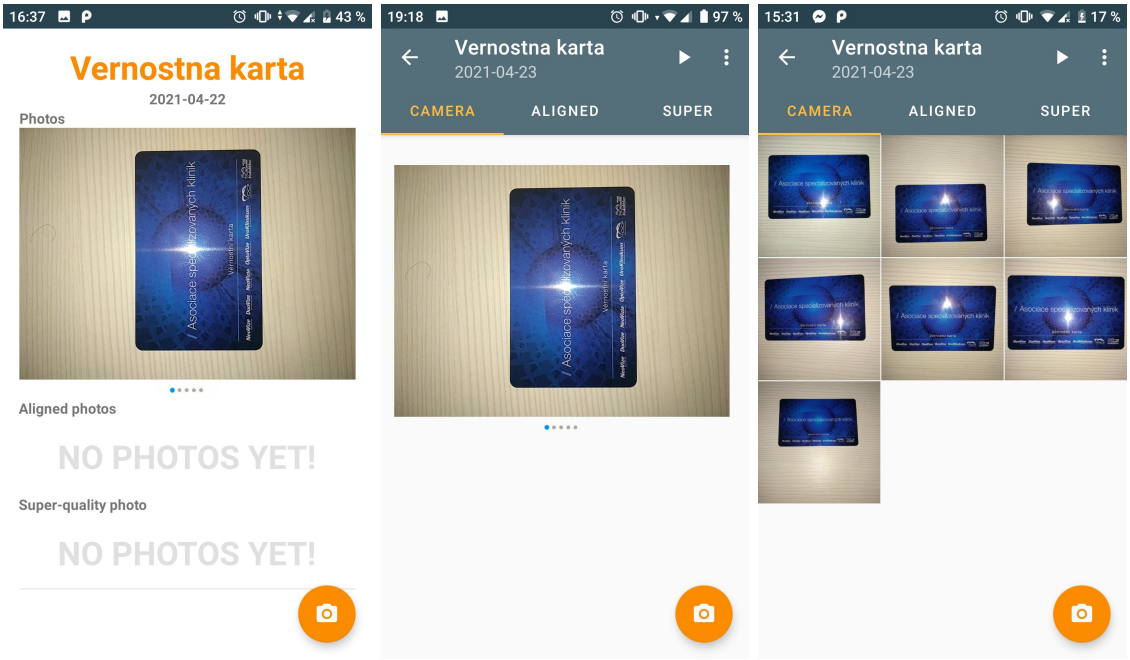
len podľa názvu a dátumu vytvorenia. Preto som sa rozhodol ku každej úlohe pridať aj miniatúru, ktorá bude vytvorená z fotiek, ktoré daná úloha obsahuje. V prípade, že úloha neobsahuje žiadne fotky, tak sa zobrazí len predvolený obrázok (*placeholder*).

### 6.4.2 Konkrétna úloha

V tejto aktivite by mali byť zobrazené jednotlivé fotky daného objektu podelené do kategórii, aby sme dokázali rozlíšiť, ktoré fotky sú priamo z kamery a ktoré sú zarovnané alebo výstupné. Prvý návrh aktivity je vidieť na obrázku 6.3 vľavo. Aktivita bol jedno veľké rolovacie zobrazenie, kde sa pod sebou nachádzali jednotlivé typy fotiek, medzi ktorými sa dalo rolovať horizontálne. Každá táto fotografia sa dala dvoma prstami priblížiť. Toto rozloženie čerpalo inspiráciu z aplikácie *Instagram*, kde príspevky obsahujúce viacero fotiek využívajú rovnaké horizontálne rolovanie a priblíženie dvoma prstami. Do dolného rohu som pridal tlačítko na vytvorenie nových fotografií do úlohy. V hornej časti som pridal tlačítko na spustenie zarovnania alebo spájania a následne možnosti na premenovanie alebo zmazanie úlohy a všetkých jej fotografií.

Po prvom teste sa však ukázalo, že táto jedna rolovacia aktivita je veľmi preplnená a neprehľadná. Užívatelia sa v nej nevedeli orientovať. Preto som sa rozhodol jednotlivé kategórie fotografií rozdeliť do kartového rozloženia, ako vidíme na obrázku 6.3 v strede. V každej karte bol potom zachovaný princíp zobrazenia fotiek z prvotného návrhu.

Po otestovaní tejto verzie sa užívateľom jednoduchšie orientovalo v danej úlohe a medzi fotografiami. Problém ale bol v prípade, keď je v úlohe vytvorených viacero fotiek, tak na to, aby sa užívateľ dokázal pozrieť na poslednú z nich, musí rolovať cez všetky ostatné. V prípade veľkého množstva fotiek toto riešenie nie je ideálne. Možno toto je jeden z dôvodov prečo aj *Instagram* limituje počet fotiek v jednom príspevku na 10 kusov.

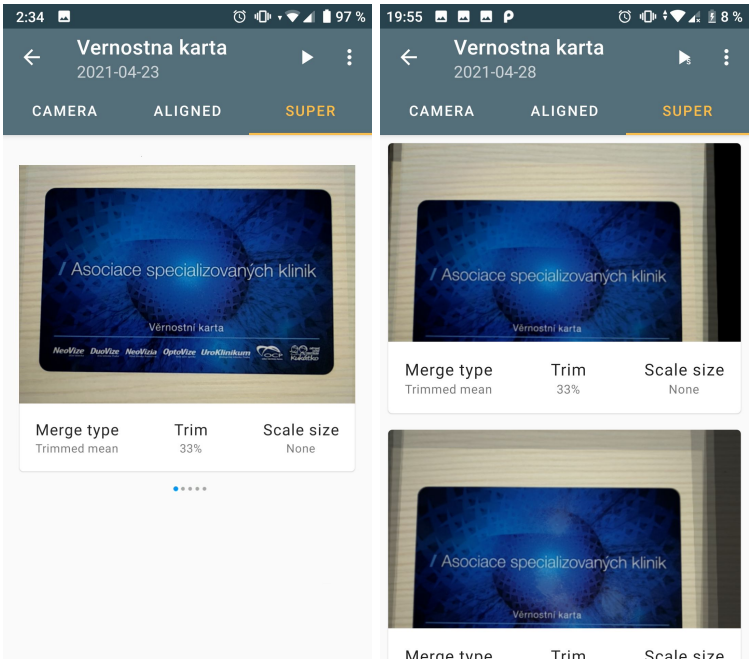



Obr. 6.3: Jednotlivé iterácie vývoja užívateľského rozhrania konkrétnej úlohy od prvotnej verzie (vľavo) až po finálnu verziu (vpravo).

Po tomto teste sa tiež objavil problém, že na to, aby sa dalo horizontálne rolovať medzi fotkami, muselo byť vypnuté horizontálne rolovanie medzi jednotlivými kartami, čo viacerým užívateľom prišlo nevhodné, keďže z mnohých aplikácii sú intuitívne navyknutí na takýto pohyb medzi kartami.

V poslednom a finálnom návrhu som sa teda rozhodol zobrať si inšpiráciu z aplikácie galérie. Navrhnuté rozloženie môžeme vidieť na obrázku 6.3 vpravo. Aktivita obsahuje mriežku s tromi stĺpcami, kde jednotlivé fotografie sú orezané na štvorcový formát. V tomto prípade bolo treba doriešiť ešte bližšie prezeranie fotiek napríklad zväčšením. V tomto som si tak isto zobral inšpiráciu z aplikácie galérie, kde po kliknutí na konkrétny štvorec fotky sa daná fotografia zobrazí vo väčšom náhľade.

Toto rozloženie bolo možné aplikovať na karty, ktoré obsahovali fotografie z kamery a zarovnané fotografie. Pri výsledných kvalitných fotografiách je potrebné uložiť informáciu ako vznikla daná fotka (aký typ spojenia, *upscaling*...). V tomto prípade nebolo vhodné použiť toto galériové rozloženie, pretože tam nie je priestor na zobrazenie týchto informácii. Tiež vo výsledných fotografiách nebude také množstvo fotografií, ako ich je vo vstupných a zarovnaných fotografiách. Ako prvé ma teda napadlo znovu použiť horizontálne rolovanie fotiek ako v prvotných iteráciách, pri ktorých by sa pri každom rolovaní zmenil popis. Obrázok tejto aktivity môžeme vidieť na obrázku 6.4 vľavo. Tu sa tiež však vyskytol ten istý problém ako v prvom návrhu, kde na horizontálne rolovanie medzi fotkami bolo potrebné vypnúť horizontálne rolovanie medzi jednotlivými kartami, ktoré prišlo užívateľom neintuitívne. Z tohto dôvodu som sa rozhodol toto rozloženie zmeniť na jednoduchý zoznam náhľadov, pod ktorými je popis metódy vzniku fotografie. Toto rozloženie môžeme vidieť na obrázku 6.4 vpravo.




Obr. 6.4: Prvotný návrh rozloženia karty s výslednými fotkami (vľavo) a finálny návrh (vpravo).

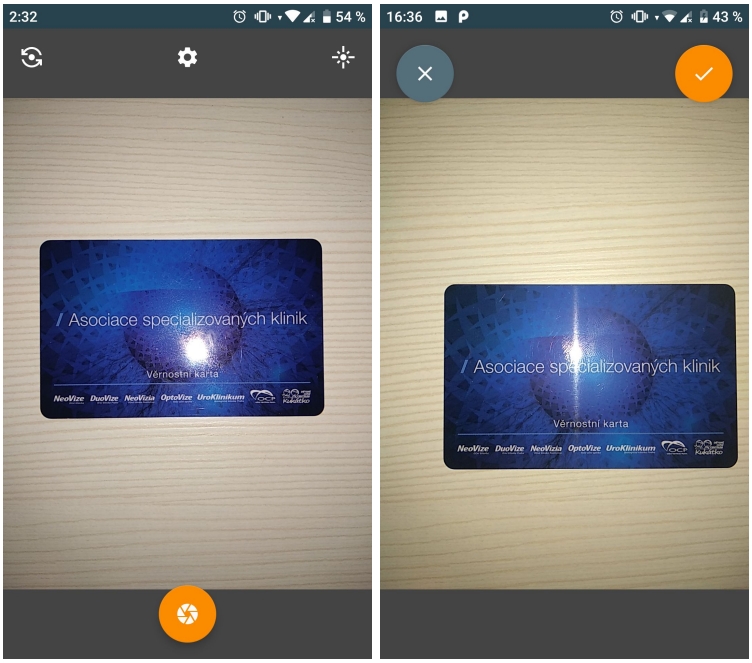
### 6.4.3 Vytváranie fotografií

UI vytvárania fotografií som sa snažil urobiť čo najjednoduchšie. Snažil som sa rozhranie zachovať čo najpodobnejšie natívnym aplikáciam na vytváranie fotiek a to tak, že tlačítko na zachytávanie fotiek som dal do stredu dole a do hornej časti rohov som dal tlačítka na zmenu kamery (predná, zadná), nastavenia kamery a nastavenie blesku.

Po vytvorení fotky sa prejde do aktivity, ktorá zobrazuje novo vytvorenú fotografiu. Prvotná myšlienka bola taká, že po vytvorení fotografie užívateľ danú fotku prezrie a potvrdí, že sa má daná fotografia uložiť alebo zahodiť. Po uložení alebo zahodení sa prejde naspäť na aktivitu vytvárania fotografie. Aktivity môžeme vidieť na obrázku 6.5.

Pri užívateľskom testovaní som sa stretol s niekoľkými zaujímavými vecami. Prvou z nich bolo to, že užívateľom sa nezdal intuitívny spôsob potvrdzovania alebo zahodenia fotky po jej vytvorení. Väčšina užívateľov predpokladala, že keď sa im zobrazila vytvorená fotografia, tak už bola automaticky uložená v úložisku a nemuseli nič potvrdzovať. Na základe tejto spätnej väzby som sa rozhodol pozrieť, ako danú vec majú navrhnuté natívne aplikácie na fotoaparát. Týmto sa potvrdila užívateľská spätná väzba, pretože po vytvorení fotky natívnou aplikáciou sa fotografia automaticky uloží a následne ju môže užívateľ zdieľať alebo zmazať. Na základe tohto som sa rozhodol pôvodný návrh zmeniť tak, že po vytvorení sa fotografia automaticky uloží a zobrazí sa v prehliadači fotiek.

Druhou dôležitou spätnou väzbou od užívateľov bolo to, že po návrate z prezerania vytvorenej fotografie, sa nastavenia blesku stále vracali do predvoleného nastavenia, čo dosť sťažovalo spôsob vytvárania fotografií, keďže pri vytvorení každej fotky vždy museli najprv nastaviť blesk do vybraného nastavenia a až následne mohli vytvoriť fotografiu. Ukladanie som upravil a nastavenie blesku sa už ukladá do nastavení aplikácie a pretrváva aj naprieč jednotlivými reštartami aplikácie.




Obr. 6.5: Aktivita na vytvorenie fotografie (vľavo), aktivita na potvrdenie alebo zahodenie vytvorenej fotky pred užívateľským testovaním (vpravo).

### 6.4.4 Prehliadanie fotiek

Po rozdelení konkrétnej úlohy do mriežkového rozloženia, ako v aplikácii galérie, bolo potrebné pridať aktivitu, v ktorej by si užívateľ dokázal podrobnejšie prezrieť vybranú fotografiu a prípadne informácie o nej. Toto isté platí aj pri vytvorení fotografie, kde si užívateľ chce vytvorenú fotku prezrieť.

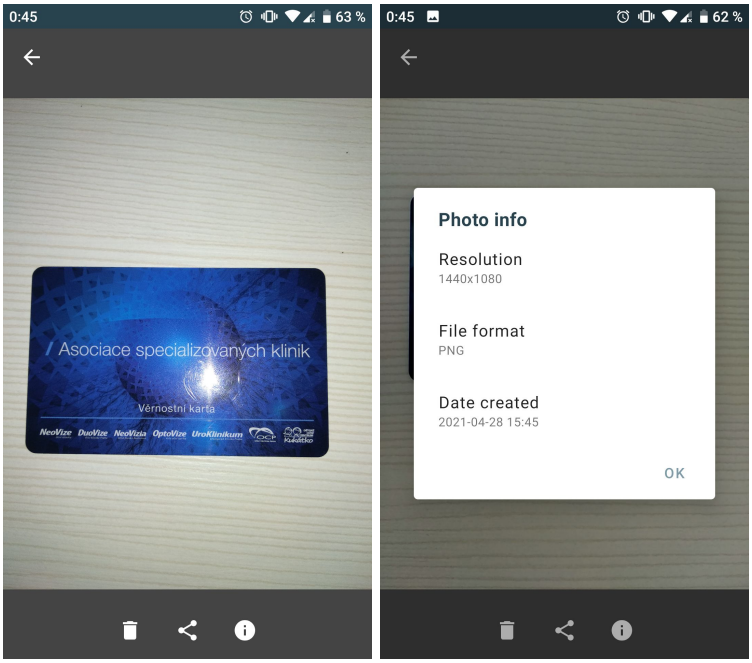
Inšpirácia pochádzala tiež z aplikácie galérie, kde po otvorení sa dá vybraná fotografia zdielať, zmazať alebo zobraziť jej informácie. Túto aktivitu môžeme vidieť na obrázku 6.6. Bolo dôležité, aby sa vybraná fotografia dala priblížiť. Na túto funkcionalitu sa mi podarilo nájsť knižnicu *PhotoView*[11].

Prvotná verzia pred testovaním po otvorení zobrazila len aktuálnu fotografiu, v ktorej sa nedalo pohybovať medzi viacerými fotografiami v danej kategórii potiahnutím prstom vľavo alebo vpravo. Keďže táto aktivita je využívaná aj pri prezeraní zarovnaných fotografií a môže slúžiť na porovnanie správnosti zarovnania jednotlivých snímkov, rozhodol som sa pridať pohyb medzi fotografiami pohybmi prstov. Aby užívatelia dokázali využiť tento pohyb na porovnanie zarovnania, je dôležité zvoliť správnu prechodovú animáciu, pretože napríklad pri využití animácie posunutia nedokážeme nič porovnať. Preto som na prechody zvolil *fade* animáciu, kde predchádzajúci snímok postupne slabne a nový snímok postupne zosiluje a tým môže užívateľ zachytiť prípadné nedokonalosti v zarovnaní.

V tejto aktivite užívateľ môže tiež získať základné informácie o fotografií ako jej formát, rozlíšenie a dátum vytvorenia. Tieto informácie sú získavané z *EXIF* metadát fotografie a sú zobrazené ako dialógové okno po kliknutí na informačnú ikonu.

Užívatelia pri teste nemali problém orientovať sa v prehliadaní fotiek, keďže takéto ovládanie je využívané v aplikáciach galérii telefónov už mnoho rokov a je intuitívne.

[11] https://github.com/Baseflow/PhotoView




Obr. 6.6: Aktivita na prehliadanie fotografií v danej kategórii. Obrázok vpravo zobrazuje dialógové okno, ktoré zobrazuje informácie o fotografií.

### 6.4.5 Spustenie zarovnania a spojenia fotiek

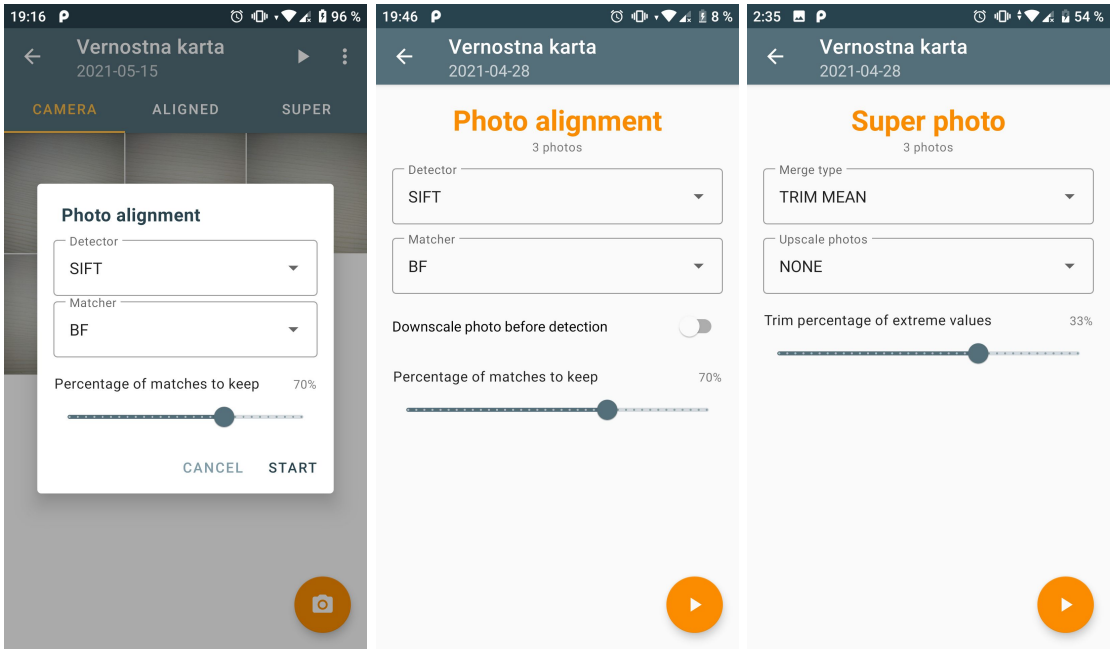
Jedná sa o jednoduché aktivity, ktoré sú otvorené po spustení zarovnania alebo vytvorenia výslednej fotky z konkrétnej úlohy. Otvorené dokážu byť až po tom, čo je dodržaný minimálny počet vytvorených snímok, v inom prípade užívateľ tieto aktivity ani nedokáže otvoriť. Ako môžeme vidieť na obrázku 6.7, aktivita obsahuje nadpis vykonávanej akcie, počet fotiek na ktorých bude akcia vykonaná a nastavenia akcie pomocou *dropdown*, prepínačov a posúvačov.

V prvom návrhu pred testovaním neboli tieto akcie samostatné aktivity, ale iba obyčajné dialógové okná. Po otestovaní užívateľom sa zdali okná príliš preplnené, preto som sa rozhodol ich dať do samostatnej aktivity.

### 6.4.6 Notifikácie aplikácie

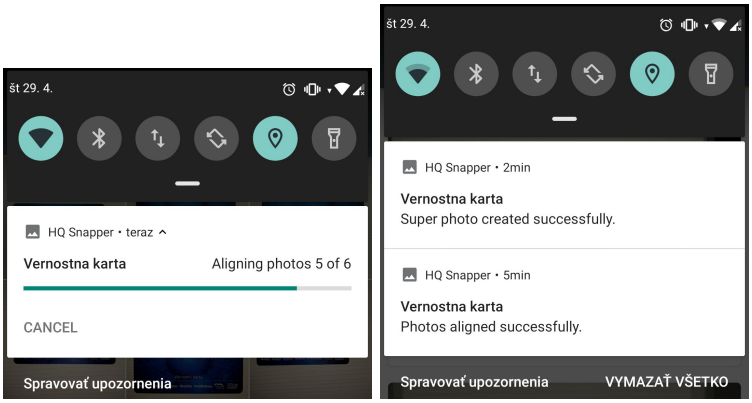
Keďže aplikácia spracováva obrázky v pozadí, je dôležité nejakým spôsobom informovať o priebehu a výsledku spracovania. Na toto vedia veľmi jednoducho poslúžiť notifikácie. Návrh notifikácií je, aby sa pri zarovnaní sa v notifikácii zobrazí aj aktuálny stav spracovania s možnosťou zrušenia. Pri spájaní zarovnaných fotografií sa zobrazí len notifikácia o spracovaní. Po dokončení spracovania sa zobrazí výsledná notifikácia, ktorá informuje o výsledku spracovania. Notifikácie sú zobrazené na obrázku 6.8.

Po prvotnom návrhu a jeho otestovaní som od niektorých užívateľov dostal návrh, aby im po kliknutí na notifikáciu o výsledku operácie (obrázok 6.8 vpravo) otvorilo aplikáciu s aktivitou, ktorá zobrazuje spracovávanú úlohu. Tento návrh dával zmysel, pretože rôzne aplikácie, ako napríklad aplikácie na posielanie správ, zobrazia notifikáciu o novej správe a následne po kliknutí na tuto notifikáciu sa zobrazí konkrétne četové okno. Túto funkciona-




Obr. 6.7: Zľava: Prvotný návrh dialógového okna na spustenie zarovnania, finálne verzie aktivít na spustenie zarovnania a vytvorenia výstupnej kvalitnej fotky.

litu som pridal a teda po kliknutí na notifikáciu o výsledku operácie sa otvorí aktivita so spracovávanou úlohou.




Obr. 6.8: Notifikácia zobrazujúca proces spracovania (vľavo) a výsledok spracovania (vpravo).

# Kapitola 7

# Implementácia aplikácie

V tejto kapitole predstavím implementačné detaily mobilnej aplikácie. Pri implementácii aplikácie som používal návrh a poznatky z predchádzajúcej kapitoly. V každej podkapitole sa budem venovať konkrétnej časti aplikácie.

## 7.1 Využité technológie pri vývoji

Na vývoj aplikácie som sa rozhodol použiť jazyk `Java` namiesto jazyku `Kotlin` hlavne kvôli mojej predchádzajúcej skúsenosti vo vývoji v tomto jazyku. Na písanie natívneho kódu som použil jazyk `C++`. Ako vývojové prostredie som používal *Android Studio* a na verzovanie projektu `git`.

## 7.2 Spracovanie fotografií

V tejto časti opíšem samotný proces implementácie spracovania vstupných fotografií a získanie výslednej kvalitnej fotografie. Prvým krokom bolo implementovať získavanie fotografií v bezstratovom formáte a následne ich zarovnanie a spojenie.

### 7.2.1 Vytváranie fotografií vo formáte PNG

Ako som uviedol v návrhu, rozhodol som sa na implementáciu kamery použiť knižnicu **CameraView**, do ktorej bolo nutné túto funkcionalitu implementovať. V zdrojom kóde tejto knižnice bolo pomerne jednoduché zorientovať sa, pretože je kvalitne zdokumentovaná.

Najprv som musel formát PNG pridať do *enumov* s hodnotou 2 a tiež pridať ho do premennej v metóde, ktorá overovala kompatibilitu jednotlivých formátov na danom zariadení. Formát PNG by mal byť kompatibilný na každom zariadení, pretože z kapitoly 2.9.1 vieme, že výstup YUV z kamery je podporovaný všade. Následne v každom súbore, kde sa nastavovali hodnoty na základe týchto *enumov*, som musel pridať vlastné nastavenia.

V metóde, ktorá je volaná pri inicializácii kamery, je potrebné nastaviť parametre senzoru na základe výstupného formátu fotografie. Na základe návrhu som nastavil ako výstupný formát YUV pixely. Výpis metódy a pridaná inicializácia je vo výpise 7.1 na riadku 10.

```
1  protected Task<CameraOptions> onStartEngine() {
2      // ...
3      LOG.i("onStartEngine:", "Opened camera device.");
4      mCameraCharacteristics = mManager.getCameraCharacteristics(mCameraId);
```

```
5       boolean flip = getAngles().flip(Reference.SENSOR, Reference.VIEW);
6       int format;
7       switch (mPictureFormat) {
8           case JPEG: format = ImageFormat.JPEG; break;
9           case DNG: format = ImageFormat.RAW_SENSOR; break;
10          case PNG: format = ImageFormat.YUV_420_888; break;
11          default: throw new IllegalArgumentException("Unknown format:" + mPictureFormat);
12      }
13      mCameraOptions = new Camera2Options(mManager, mCameraId, flip, format);
14      // ...
```

Výpis 7.1: Časť metódy `onStartEngine()` v triede `Camera2Engine` s pridanou PNG inicializáciou.

Metóda, ktorá je volaná po vyvolaní snímku a spracováva dáta v zadanom formáte zo senzoru, zavolá metódu na spracovanie PNG snímku. Môžeme ju vidieť vo výpise 7.2. Metóda volaná v riadku 4 skonvertuje YUV pixely na RGB. Táto metóda je implementovaná v jazyku `C++`, aby sa spracovanie nespomaľovalo.

Ako som spomínal v návrhu, pri testovaní snímkov dochádza k dlhému spracovaniu. Problémom je volanie metódy na riadku 24, ktorá vytvára výsledný PNG snímok. Zo začiatku som si myslel, že táto metóda je písaná v jazyku `Java` a preto je taká pomalá. Po nahliadnutí do zdrojových kódov som ale zistil, že táto metóda je implementovaná natívne a teda zrýchliť nepôjde. Po naštudovaní rôznych zdrojov som dospel k názoru, že tvorba PNG snímkov jednoducho tak dlho trvá a tento fakt je ešte znásobený menším výkonom zariadení v porovnaní s počítačmi.

```
1   private void readPngImage(@NonNull Image image) {
2       int[] result = new int[image.getWidth() * image.getHeight()];
3       byte[][] yuvBytes = fillBytes(image.getPlanes());
4       ImageUtils.convertYUV420ToARGB8888(
5               yuvBytes[0],
6               yuvBytes[1],
7               yuvBytes[2],
8               image.getWidth(),
9               image.getHeight(),
10              image.getPlanes()[0].getRowStride(),
11              image.getPlanes()[1].getRowStride(),
12              image.getPlanes()[1].getPixelStride(),
13              result
14      );
15      Bitmap bitmap = Bitmap.createBitmap(
16              image.getWidth(),
17              image.getHeight(),
18              Bitmap.Config.ARGB_8888
19      );
20      bitmap.setPixels(result, 0, image.getWidth(), 0, 0,
21          image.getWidth(), image.getHeight()
22      );
23      ByteArrayOutputStream stream = new ByteArrayOutputStream();
24      bitmap.compress(Bitmap.CompressFormat.PNG, 100, stream);
25      mResult.data = stream.toByteArray();
26      bitmap.recycle();
27  }
```

Výpis 7.2: Metóda, ktorá spracováva dáta zo senzoru v formáte YUV, prevedie ich do RGB a uloží do PNG.

### Prevod YUV do RGB

Ako som uviedol v predchádzajúcej časti, metódu na prevod pixelov je potrebné implementovať natívne pre čo najefektívnejšie riešenie. Všetky potrebné výpočty boli spomenuté v kapitole 3.1. Implementáciu takéhoto prevodu v jazyku `C` sa mi podarilo nájsť v zdrojovom kóde[1] knižnice *Tensorflow*, ktorá je *open-source*. Túto implementáciu som použil v mojom riešení, čo mi uľahčilo implementáciu a testovanie, keďže knižnica ako *Tensorflow* má svoj kód riadne otestovaný. Tento natívny kód už len stačilo napojiť na *Java Native Interface*. Toto napojenie bolo tiež k dispozícii v danom zdrojovom kóde[2].

## 7.3 Zarovnávanie fotografií

Implementácia časti, ktorá zarovnáva vytvorené fotografie, nebola ničím špeciálna, pretože všetky potrebné funkcie na zarovnanie obsahovala knižnica *OpenCV*, ktorá tieto metódy vykonáva natívne. Tento proces sa vykonáva na pozadí pomocou navrhnutej knižnice *WorkManager*.

## 7.4 Spájanie fotografií

Pri spájaní jednotlivých fotografií bolo potrebné napísať vlastné riešenie, pretože takúto funkcionalitu neposkytovala žiadna knižnica. Na dosiahnutie najlepšej výkonnosti je potrebné túto časť napísať natívne, pretože algoritmus spájania funguje tak, že iteruje cez každý pixel každého vstupného obrázku a následne ich spriemeruje. Toto iterovanie je v jazyku `Java` pomalé.

Pri spájaní sa najprv z každého vstupného obrázku na danej pozícii vytvorí pole pixelov pre každý farebný kanál. Tieto polia sa v prípade orezaného spriemerovania najprv zoradia a vymaže sa z nich zadané percento hodnôt z oboch smerov poľa. Následne sa vytvorí priemer týchto polí. Pri priemere bolo dôležité dodržať aby nedošlo k pretečeniu premennej v prípade väčšieho množstva vstupných obrázkov, preto sa priemer počíta iteratívne. Výpis kódu na počítanie iteratívneho priemeru je na výpise 7.3.

```
uchar vectorMean(const std::vector<uchar> &vector) {
    double avg = 0;
    int t = 1;
    for (uchar x : vector) {
        avg += (x - avg) / t;
        ++t;
    }
    return round(avg);
}
```

Výpis 7.3: Funkcia na iteratívny výpočet priemeru vstupných hodnôt.

[1] https://github.com/tensorflow/tensorflow/blob/master/tensorflow/tools/android/test/jni/yuv2rgb.cc

[2] https://github.com/tensorflow/tensorflow/blob/master/tensorflow/tools/android/test/src/org/tensorflow/demo/env/ImageUtils.java

## 7.5 Databáza aplikácie

Ako je uvedené v návrhu, na implementáciu lokálnej databázy aplikácie je využitá knižnica *Room*. Implementácia databázy pomocou tejto knižnice sa delí na definíciu tabuliek a definíciu objektov na prístup k dátam.

Jednotlivé tabuľky databázy sú definované podľa navrhnutého ER diagramu v kapitole 6.2.2. Vďaka tejto knižnici sa jednotlivé tabuľky definujú pomocou tried, kde jednotlivé stĺpce tabuľky predstavujú triedne premenné.

Pre prácu s týmito tabuľkami je nutné pre každú z nich vytvoriť objekt prístupu k údajom (*data access object – DAO*). Každá metóda v takejto triede predstavuje nejakú operáciu vykonanú nad konkrétnou tabuľku databázy. Ukážka takejto metódy je vo výpise 7.4.

```
@Dao
public interface TaskDao {
    // ...
    @Query("SELECT * FROM task WHERE id = :id")
    LiveData<Task> get(int id);
    // ..
}
```

Výpis 7.4: Ukážka definície *DAO* metódy na prístup k dátam tabuľky.

Pri prvotnom vytvorení aplikácie sa v triede `Application`, ktorej metóda `onCreate` je volaná raz pri prvom otvorení aplikácie, vytvorí statický prístup k databáze, vďaka ktorému môžeme pristupovať k jednotlivým *DAO* programu a tak vykonávať prístupy k jednotlivým dátam v tabuľke.

## 7.6 Užívateľské rozhranie

Pri implementácii užívateľského rozhrania som využíval navrhnuté a otestované rozhranie. Keďže som využíval *MVVM* architektúru aplikácie, tak každý *fragment* mal svoj príslušný *View-Model*. Na dosiahnutie reaktívnosti aplikácie som na predávanie dát medzi *Model*, *View-Model* a *View* využíval pozorovateľné (*observable*) **LiveData**, čo znamená, že pri každej zmene dát v databáze sa zmeny automaticky prejavia v užívateľskom rozhraní. Ukážka pozorovateľného *LiveData* je možné vidieť vo výpise 7.5.

```
@Override
protected void populateWithContent() {
    // ...
    viewModel.getTask().observe(getViewLifecycleOwner(), task -> {
        viewBinding.topAppBar.setTitle(task.getName());
        viewBinding.topAppBar.setSubtitle(Utils.formatDate(task.getDateCreated()));
    });
    // ...
}
```

Výpis 7.5: Časť metódy, ktorá využíva pozorovateľné *LiveData* na nastavenie údajov v užívateľskom rozhraní.

Implementácia užívateľského rozhrania bola podľa tohto prístupu a za použitia návrhu priamočiara. Pri implementácii fragmentu na prehliadanie fotografií bolo potrebné implementovať funkcionalitu zdieľania fotografie. Na toto zdieľanie som použil *Content Provider* (opísaný v kapitole 2.3), vďaka ktorému môžu ostatné aplikácie čítať vopred určené dáta (v tomto prípade fotografiu) a naša aplikácia nevyžaduje žiadne povolenia naviac na prácu s úložiskom.

Vo fragmente na prehliadanie fotiek bolo potrebné na zobrazenie informácii o fotografiách pridať spôsob na získavanie týchto informácii z *EXIF* dát vytvorených snímkov. Na to som sa rozhodol použiť knižnicu *metadata-extractor*[3], ktorá tieto dáta získava za nás. Problémom ale bolo, že rôzny formát obrázkov má tieto dáta uložené pod inými zložkami a rôznymi názvami a preto som k tejto knižnici musel vytvoriť ešte pomocnú triedu, ktorá dokáže čítať tieto dáta nezávisle na formáte obrázku.

[3] https://github.com/drewnoakes/metadata-extractor

# Kapitola 8

# Záver

Cieľom tejto práce bolo vytvoriť aplikáciu pre systém Android, ktorá dokáže vytvárať vysoko kvalitné fotografie rovinných predmetov. Na jej vytvorenie bolo potrebné najprv naštudovať proces tvorby aplikácii pre tento systém a spôsob ako sa na ňom dajú vytvárať fotografie.

Na to, aby som dokázal navrhnúť proces spracovania obrázkov, som si musel naštudovať problematiku spracovania obrázkov, pri čom som sa zameral hlavne na ich zarovnávanie, ktoré sa skladá z detekcie významných bodov a ich deskriptorov, hľadanie zhodných významných bodov a následne výpočet homografie z týchto zhodných bodov. Zameral som sa hlavne na tri metódy: *SIFT*, *SURF* a *ORB*.

Potom som zo získaných znalostí dokázal navrhnúť a implementovať samotný proces vytvorenia kvalitných fotografií z viacerých vstupných fotografií rovinného predmetu. Tento proces som aj testoval na mnou vytvorenej sade fotografií rôznych predmetov. Moje testy potvrdili výsledky testov z článku [23], kde *SIFT* dosahoval najkvalitnejšie výsledky zarovnania a *ORB* bol zase najrýchlejší.

Tento proces som implementoval do mobilnej aplikácie modernými *Jetpack* knižnicami ako napríklad *Room* pre prácu s databázou, *WorkManager* na vykonávanie plánovaných úloh na pozadí. Na vývoj aplikácie som tiež použil odporúčanú *MVVM* architektúru.

Posledným krokom bolo vytvoriť kvalitné užívateľské rozhranie. Najprv som vytvoril prvotné verzie rozhrania, ktoré som testoval na ľuďoch a získané znalosti z testovania a spätnú väzbu od ľudí som využíval na zdokonalovanie užívateľského rozhrania. Na uľahčenie vývoja rozhrania som využíval *Jetpack* knižnice ako *Navigation component* a *View Binding*.

Výsledná implementovaná aplikácia umožňuje užívateľovi vytvárať viaceré fotografie rovinných predmetov v bezstratovom formáte *PNG*, ktoré v prvom kroku podľa užívateľom preferovaných metód aplikácia zarovná. Tieto zarovnané fotografie môže užívateľ skontrolovať a prípadne odstrániť nevhodné fotografie. Následne sa z týchto zarovnaných fotografií vytvorí jedna výsledná, kvalitná fotografia, pri ktorej užívateľ tiež môže zvoliť preferovaný spôsob tvorby. Všetky fotografie vytvorené v aplikácii je možné zdieľať alebo posielať do externých aplikácii.

Do budúcna by sa aplikácia dala vylepšiť spôsobom, že by spracovanie fotografií prebiehalo v *cloude*, čo by vyriešilo problémy ako dochádzanie operačnej pamäte a nedostatočného výkonu zariadení. Tento prístup by uľahčil prípadnú expanziu aplikácie na viaceré platformy, keďže celá logika na spracovanie by bola centralizovaná a vývoj aplikácie na rôzne platformy by nebol tak komplikovaný. Negatívom tohto prístupu je, že by vyžadoval pomerne kvalitné pripojenie na internet, pretože posielanie väčšieho množstva fotografií v bezstratovom formáte je pomerne náročne na šírku pásma.

Ako možné pokračovanie aplikácie je aj vytvorenie varianty pre systém *iOS*. Vývoj na túto platformu by podľa môjho názoru bol jednoduchší, pretože nemá také veľké množstvo zariadení a tým by bola práca s kamerou jednoduchšia. Taktiež sú tieto zariadenia oveľa výkonnejšie ako väčšina zariadení so systémom *Android*.

# Literatúra


[1] Callaham, J. The history of Android: The evolution of the biggest mobile OS in the world. november 2020. https://www.androidauthority.com/history-android-os-name-789433/.

[2] *Distribution data for Android* [online]. Dostupné z: https://androiddistribution.io/.

[3] *Platform Architecture* [online]. 2020. Dostupné z: https://developer.android.com/guide/platform.

[4] *Application Fundamentals* [online]. 2019. Dostupné z: https://developer.android.com/guide/components/fundamentals.

[5] *Understand the Activity Lifecycle* [online]. 2020. Dostupné z: https://developer.android.com/guide/components/activities/activity-lifecycle.

[6] *Activity lifecycle and state* [online]. 2020. Dostupné z: https://developer.android.com/codelabs/android-training-activity-lifecycle-and-state.

[7] *Permissions on Android* [online]. 2020. Dostupné z: https://developer.android.com/guide/topics/permissions/overview.

[8] *Guide to app architecture* [online]. 2021. Dostupné z: https://developer.android.com/jetpack/guide.

[9] *LiveData Overview* [online]. 2021. Dostupné z: https://developer.android.com/topic/libraries/architecture/livedata.

[10] *Meet Android Studio* [online]. 2020. Dostupné z: https://developer.android.com/studio/intro.

[11] *CameraX overview* [online]. 2021. Dostupné z: https://developer.android.com/training/camerax.

[12] *Android.hardware.camera2* [online]. 2021. Dostupné z: https://developer.android.com/reference/android/hardware/camera2/package-summary.

[13] Ibraheem, N. A., Hasan, M. M., Khan, R. Z. a Mishra, P. K. Understanding color models: a review. *ARPN Journal of science and technology*. Citeseer. 2012, zv. 2, č. 3, s. 265–275. https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.679.8051.

[14] Jack, K. *Video demystified: a handbook for the digital engineer*. Elsevier, 2007. ISBN 978-0-7506-8395-1. https://www.sciencedirect.com/book/9780750683951/video-demystified.

[15] John, N., Viswanath, A., Vishvanathan, S. a Kp, S. Analysis of Various Color Space Models on Effective Single Image Super Resolution. August 2016, zv. 384, s. 529–540. https://link.springer.com/chapter/10.1007%2F978-3-319-23036-8_46.

[16] Chen, H., Sun, M. a Steinbach, E. Compression of Bayer-Pattern Video Sequences Using Adjusted Chroma Subsampling. *IEEE Transactions on Circuits and Systems for Video Technology*. 2009, zv. 19, č. 12, s. 1891–1896. DOI: 10.1109/TCSVT.2009.2031370. https://ieeexplore.ieee.org/document/5229234.

[17] Ansari, A., Mohammadi, M. a Parvez, M. A Comparative Study of Recent Steganography Techniques for Multiple Image Formats. *International Journal of Computer Network and Information Security*. Január 2019, zv. 11, s. 11–25. DOI: 10.5815/ijcnis.2019.01.02. http://www.mecs-press.org/ijcnis/ijcnis-v11-n1/v11n1-2.html.

[18] Hassaballah, M., Ali, A. a Alshazly, H. Image Features Detection, Description and Matching. In:. Február 2016, sv. 630. ISBN 978-3-319-28852-9. https://link.springer.com/chapter/10.1007%2F978-3-319-28854-3_2.

[19] Harris, C. G., Stephens, M. et al. A combined corner and edge detector. In: Citeseer. *Alvey Vision Conference*. 1988, sv. 15, č. 50, s. 10–5244. https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.231.1604.

[20] Lowe, D. G. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*. Springer. 2004, zv. 60, č. 2, s. 91–110. https://link.springer.com/article/10.1023/B:VISI.0000029664.99615.94.

[21] Bay, H., Ess, A., Tuytelaars, T. a Van Gool, L. Speeded-Up Robust Features (SURF). *Computer Vision and Image Understanding*. 2008, zv. 110, č. 3, s. 346–359. ISSN 1077-3142. https://www.sciencedirect.com/science/article/pii/S1077314207001555.

[22] Rublee, E., Rabaud, V., Konolige, K. a Bradski, G. ORB: An efficient alternative to SIFT or SURF. In: Ieee. *2011 International conference on computer vision*. 2011, s. 2564–2571. https://ieeexplore.ieee.org/document/6126544.

[23] Karami, E., Prasad, S. a Shehata, M. Image matching using SIFT, SURF, BRIEF and ORB: performance comparison for distorted images. *ArXiv preprint arXiv:1710.02726*. 2017. https://arxiv.org/abs/1710.02726.

[24] *Feature Matching* [online]. Dostupné z: https://docs.opencv.org/master/dc/dc3/tutorial_py_matcher.html.

[25] Bradski, G. a Kaehler, A. *Learning OpenCV: Computer Vision with the OpenCV Library*. O'Reilly Media, 2008. ISBN 9780596554040.

[26] Fischler, M. A. a Bolles, R. C. Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography. *Commun. ACM*. New York, NY, USA: Association for Computing Machinery. jún 1981, zv. 24, č. 6, s. 381–395. DOI: 10.1145/358669.358692. ISSN 0001-0782. https://dl.acm.org/doi/abs/10.1145/358669.358692.