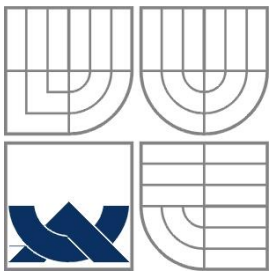
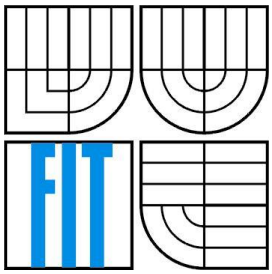


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

VYUŽITÍ VR HEADSETU PRO VÝUKU HRY NA KLAVÍR/KLÁVESY

VR HEADSET ASSISTANCE FOR KEYBOARD PLAYING

BAKALÁŘSKÁ PRÁCE
BACHELOR‘S THESIS

AUTOR PRÁCE
AUTHOR
MICHAL CHOMO

VEDOUCÍ PRÁCE
SUPERVISOR
ING. MAREK ŠOLONY

BRNO 2016

Abstrakt

Tahle práce se zabývá návrhem a implementací aplikace pro chytrý telefon s operačním systémem Android, která zpracuje obraz snímaný kamerou telefonu, detekuje v něm klávesy a pak na obrazovce telefonu zobrazí reální obraz z kamery doplněn o akordy nebo stupnice přímo na klávesách. Displej telefonu je rozdělen na dvě polovice, tedy pro každé oko uživatele je samostatný obraz a telefon je vložen do headsetu Google Cardboard. Aplikace je implementována v jazyce Java a s použitím Native Development Kitu také v C++, na zpracování obrazu je použita knihovna OpenCV.

Abstract

This thesis presents a design and implementation of an application for a smartphone with Android operating system, which processes image recorded with phone camera, detects keyboard in it and then displays real image from camera on phone screen with chords or scales added directly on keyboards. Phone display is split in two, so there is individual image for each eye and the phone is inserted in the Google Cardboard headset. Application is implemented in Java and using Native Development Kit also in C++, for image processing, OpenCV library is used.

Klíčová slova

rozšířená realita, zpracování obrazu, chytrý telefon, Android, Google Cardboard, OpenCV

Keywords

augmented reality, image processing, smartphone, Android, Google Cardboard, OpenCV

Citace

CHOMO, Michal. *Využití VR headsetu pro výuku hry na klavír/klávesy*. Brno, 2016. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Šolony Marek.

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Marka Šolonyho. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Michal Chomo
16. května 2016

Poděkování

Chtěl bych poděkovat vedoucímu mojí bakalářské práce Ing. Markovi Šolonymu za čas a úsilí, které věnoval konzultacím se mnou a taky za cenné rady.

Obsah

1 Úvod

Rozšírená realita je stále populárnejšia a nachádza využitie v mnohých odvetviach ako napr. medicína, vzdelávanie, armáda, zábavný priemysel atď. Je ale potrebné odlíšiť tento pojem od pojmu virtuálna realita, pretože sú často zamieňané. Virtuálna realita je umelo vytvorený svet, ktorý sa skladá z obrazov a zvukov vytvorených počítačom a ktorý je ovplyvnený konaním osoby nachádzajúcej sa v ňom. [1] Rozšírená realita je upravená verzia reality vytvorená pomocou technológie tak, že prekrýva obraz snímaný nejakým zariadením (napr. kamerou smartfónu) pridanými informáciami. [1] Táto práca sa bude venovať rozšírenej realite a jej použitiu na vzdelávanie v hre na klávesy. Cieľom je vytvoriť aplikáciu pre smartfón s operačným systémom Android, ktorá pomocou headsetu Google Cardboard umožní užívateľovi naučiť sa základy hry na klávesoch intuitívnym spôsobom. Aplikácia má názov *CarboardKeyboard*. Veľká výhoda spočíva v dostupnosti potrebného vybavenia, pretože smartfón v dnešnej dobe vlastní alebo si môže dovoliť skoro každý a Google Cardboard je veľmi lacný headset, nakoľko je vyrobený z kartónového papiera.

Technológie použité na vývoj aplikácie sú programovacie jazyky Java a C++ a open source knižnica OpenCV. Jazyk Java je použitý preto, že je základným jazykom na tvorbu aplikácií pre systém Android a najlepšou možnosťou, čo sa týka jednoduchosti vývoja a prenositeľnosti aplikácie na rôzne hardvérové platformy. Jazyk C++ zase poskytuje lepší výkon pri náročných úkonoch, čo spracovanie obrazu v reálnom čase určite je, ale na druhú stranu je potrebné použiť Native Development Kit a vývoj sa týmto komplikuje. Knižnica OpenCV je zameraná na spracovanie obrazu v reálnom čase a medzi jej výhody patrí široká škála funkcií, početná užívateľská komunita a trvalé aktualizácie v intervaloch 3 až 4 mesiace. Na detekciu klávesov v obraze sú použité ArUco markery, ktoré majú v OpenCV funkcionalitu vo vlastnom module.

V kapitole 2 je podrobnejší popis rozšírenej reality a rôzne pohľady na tento pojem. Kapitola 3 sa zaoberá problematikou planárnej homografie, vysvetľuje jej význam a spôsob počítania. V kapitole 4 sú popísané ArUco markery, spôsob ich vytvárania a proces detekcie. Kapitola 5 prezentuje návrh a implementáciu aplikácie s ukážkami a vysvetlením kódu. V kapitole 6 sú komentované výsledky testovania aplikácie.

2 Rozšírená realita

V tejto kapitole sa čitateľ zoznámi s pojmom rozšírená realita a pochopí základné fungovanie a princípy. Informácie sú primárne čerpané z [2]. Rozšírená realita sa dá chápať ako prostredie, kde koexistujú reálne a virtuálne objekty. Virtuálne objekty použité v rozšírenej realite môžu zahŕňať texty, obrázky, videá, zvuky, 3D modely a animácie.
[2] definuje rozšírenú realitu ako systém s nasledujúcimi charakteristikami:

- Kombinuje reálne a virtuálne objekty
- Je interaktívny v reálnom čase
- Je zaznamenaný v troch dimenziách

Rozšírená realita sa dá aplikovať na všetky zmysly, nie len na zrak. Zatiaľ je kladený najväčší dôraz na zmiešavanie reálnych a virtuálnych obrazov a grafiky, avšak je možné zahrnúť do rozšírenej reality aj zvuk alebo hmat pomocou slúchadiel resp. špeciálnych rukavíc.

2.1 Technológie tvorby rozšírenej reality

Základným rozhodnutím pri tvorbe rozšírenej reality je, akým spôsobom sa bude kombinovať reálne s virtuálnym. Sú dve možnosti: optické a video technológie. Každá má svoje výhody a nevýhody, ktoré budú popísané nižšie. Zariadenie, ktoré sa používa na zobrazenie rozšírenej reality, sa nazýva head-mounted display (HMD), vo voľnom preklade displej založený na hlave. HMD môžu byť uzavreté alebo priehľadné, pričom priehľadnosť môže byť buď na optickej báze alebo pomocou kamery snímajúcej reálny svet.

2.1.1 Optické HMD

Optické HMD fungujú tak, že na polopriehľadný displej pred užívateľovými očami zobrazujú virtuálne objekty, takže užívateľ vidí aj reálny svet aj virtuálne objekty. Tento prístup je podobný ako pri head-up displejoch (HUD) používaných v armádnych lietadlách alebo moderných automobiloch, s tým rozdielom, že HMD je pripevnený na hlave užívateľa. HMD väčšinou redukujú prichádzajúce svetlo,

Obrázok 2.1: Head-up displej v automobile

aby užívateľ vnímal aj svetlo vychádzajúce z displeja, teda pri vypnutom displeji je efekt trochu podobný slnečným okuliarom.
Typickým príkladom optického HMD sú okuliare Google Glass zobrazené na Obrázok 2.2. Je možné ich ovládať hlasovými inštrukciami a touchpadom na pravej strane.

Obrázok 2.2: Google Glass

Medzi výhody optických HMD patrí:

- Žiadna latencia obrazu reálneho sveta
- Rozlíšenie nie je limitované rozlíšením kamery
- Keď zariadenie stratí energiu, užívateľ stále vidí reálny svet
- Žiadna odchýlka pohľadu spôsobená odlišným umiestnením kamery a očí

Nevýhody optických HMD sú napríklad:

- Virtuálne objekty nemôžu úplne zakryť reálne objekty
- Oneskorenie v zobrazení virtuálnych objektov
- Úzke zorné pole

2.1.2 HMD s kamerou

HMD s kamerou sú v podstate uzavreté, ale pomocou jednej alebo dvoch kamier zabezpečujú obraz reálneho sveta, ktorý je digitálne spracovaný a sú doňho pridané virtuálne objekty. Tento obraz sa užívateľovi premieta na displej, prípadne dva displeje pred očami.
Výhody HMD s kamerou sú:

- Široké zorné pole
- Obraz z kamery pomáha určiť polohu hlavy užívateľa
- Adaptácia svetlosti reálnych a virtuálnych objektov
- Žiadny rozdiel latencie medzi reálnymi a virtuálnymi objektmi

Medzi nevýhody HMD s kamerou patria:

- Odchýlka pohľadu spôsobená odlišným umiestnením kamery a očí
- Rozlíšenie je limitované rozlíšením kamery a displeja
- Latencia obrazu

2.1.3 Google Cardboard

Google Cardboard je platforma virtuálnej a rozšírenej reality vyvinutá spoločnosťou Google. Headset Google Cardboard sa skladá z jednoduchých a lacných komponentov, takže si ho užívatelia môžu vyrobiť aj sami. Oficiálny výrobca neexistuje, Google iba zverejnil špecifikáciu komponentov a výroby a mnoho výrobcov ponúka set potrebných častí a návodu na zloženie.
Cardboard sa skladá z:

- Kus kartónu presne špecifikovaného tvaru
- Šošovky s ohniskovou vzdialenosťou 45mm
- Magnety alebo kapacitná páska
- Suchý zips
- Gumička na udržanie telefónu v stabilnej polohe

Po zložení headsetu sa do jeho zadnej časti umiestni smartfón, na ktorom je nainštalovaná aplikácia Cardboard alebo iná aplikácia kompatibilná s touto platformou. Aplikácia rozdelí displej smartfónu na dve časti a aplikuje sférické skreslenie, aby vyrovnala asférické skreslenie zo šošoviek. Výsledkom je stereoskopický obraz so širokým zorným poľom. Ovládanie aplikácie sa vykonáva pomocou posunu magnetu na boku headsetu. Magnetometer v smartfóne zaznamená tento pohyb a interpretuje ho ako stlačenie tlačidla. Výber z menu teda prebieha tak, že užívateľ otáča hlavou a keď sa v strede obrazovky zvýrazní položka, ktorú chce zvoliť, posunie magnet. Cardboard sa dá vnímať ako HMD s kamerou, keď je na snímanie okolia použitá kamera smartfónu vloženého v Cardboard-e. Týmto spôsobom je realizovaná aj aplikácia pre výuku hry na klávesy.

Obrázok 2.3: Google Cardboard

2.2 Využitie rozšírenej reality

Rozšírená realita nachádza využitie v mnohých odvetviach ľudskej činnosti. Táto podkapitola na niekoľkých príkladoch opisuje, ako sa dá rozšírená realita využiť na rôzne prospešné účely.

- **Medicína**
  Rozšírená realita môže poskytnúť chirurgovi užitočné informácie ako napr. srdcovú frekvenciu, krvný tlak, stav pacientových orgánov atď. Taktiež sa dá pomocou tomografie, ultrazvuku a magnetickej rezonancie skonštruovať model pacienta a doktor potom pomocou rozšírenej reality v reálnom čase vidí do vnútra pacienta.
- **Armáda**
  Pre vojakov s HMD rozšírená realita slúži ako prepojený komunikačný systém poskytujúci informácie o bojovej situácii. Vojak tak vidí ľudí a rôzne objekty označené špeciálnymi indikátormi, ktoré ho môžu varovať pred potenciálnym nebezpečenstvom. Taktiež sa mu zobrazujú virtuálne mapy prostredia pre lepšiu navigáciu.

- **Vzdelávanie**
  Rozšírená realita vhodne dopĺňa štandardné prostriedky vo vzdelávaní. V učebniciach sa nachádzajú špeciálne značky, ktoré obsahujú grafické alebo zvukové informácie, ktoré sa dajú zobraziť pomocou smartfónu alebo tabletu. Taktiež sa používajú interaktívne 3D modely chemických molekúl alebo ľudského tela, vďaka ktorým študenti lepšie pochopia danú problematiku.

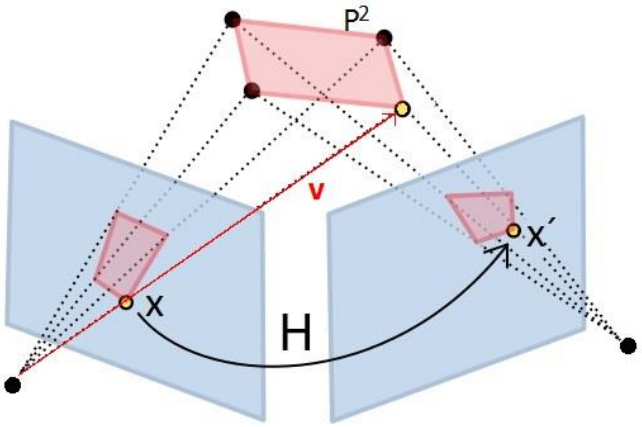
- **Zábavný priemysel**
  Do športových prenosov sa pridávajú virtuálne informácie napr. v podobe národnej vlajky v dráhe pretekára alebo čiary indikujúcej aktuálny najlepší výkon pri plaveckých pretekoch. Aj vizualizácia za moderátorom počasia je forma rozšírenej reality. Tiež vzniká množstvo hier pre smartfóny, ktoré využívajú rozšírenú realitu a poskytujú tak užívateľom intenzívny zážitok z hry.

3 Planárna homografia

V tejto kapitole sa čitateľ dozvie o problematike planárnej homografie. Kapitola je zameraná na vysvetlenie princípu planárnej homografie a dôvodu, prečo sa používa v počítačovej grafike aj konkrétne v tejto bakalárskej práci. Informácie sú čerpané z [3] a [4].

3.1 Čo to je homografia

Dvojrozmerný bod so súradnicami (x, y) v obraze môže byť reprezentovaný trojrozmerným vektorom $\mathbf{v} = (x_1, x_2, x_3)$, kde $x = \frac{x_1}{x_3}$ a $y = \frac{x_2}{x_3}$. Toto sa nazýva homogénna reprezentácia bodu, ktorý leží v projektívnej rovine P^2. Homografia je bijektívne zobrazenie bodov a priamok v projektívnej rovine P^2. Synonymické pojmy pre toto zobrazenie zahŕňajú kolineáciu, projektivitu a planárnu projektívnu transformáciu. Algebraická definícia znie takto: Bijektívne zobrazenie z roviny P^2 do nej samotnej je projektivitou vtedy a len vtedy, ak existuje matica H s rozmermi $3x3$, ktorá nie je jednotková a pre každý bod v rovine P^2 reprezentovaný vektorom $\mathbf{v}$ existuje ním zobrazovaný bod, ktorý sa rovná $H * \mathbf{v}$. To znamená, že ak chceme vypočítať homografiu, ktorá zobrazuje každý bod x do jeho príslušného bodu x', je potrebné vypočítať maticu H. Maticu H je možné násobiť ľubovoľnou nenulovou konštantou bez zmeny projektívnej transformácie, teda matica H je homogénna matica a obsahuje 9 elementov, z ktorých je 8 neznámych.



Obrázok 3.1: Ilustrácia homografie

3.2 Použitie homografie v počítačovej grafike

V počítačovej grafike sa vyskytuje mnoho situácií, pri ktorých je potrebné vypočítať homografiu. Táto podkapitola prezentuje niektoré z týchto situácií a ukazuje príklady použitia homografie v praxi.

3.2.1 Kalibrácia kamery

Kalibrácia kamery je proces určovania vnútorných a vonkajších parametrov kamery. Vnútorné parametre sú špecifické pre konkrétnu kameru, sú to napr. ohnisková vzdialenosť, bod stredu snímky a skreslenie objektívu. Vonkajšie parametre predstavujú pozíciu a orientáciu kamery v priestore.

Kalibrácia je často hlavným krokom mnohých grafických aplikácií, pretože dovoľuje určiť vzťah medzi bodmi na obraze a bodmi v skutočnom priestore. Pre početné základné operácie spracovania obrazu, napr. odstránenie radiálneho skreslenia, je potrebné poznať maticu kamery, označovanú *K*. Vnútorné a vonkajšie parametre kamery je možné určiť vypočítaním homografie z viacerých obrazov rovnakého rovinného útvaru snímaných z rôznej perspektívy. Používa sa rovnica (*3.1*)[3]:

$$H = KRt, \tag{3.1}$$

kde *H* je matica homografie, *K* je matica vnútorných parametrov kamery, *R* je matica rotácie a **t** je vektor posunu.

3.2.2 3D rekonštrukcia

Cieľom 3D rekonštrukcie v počítačovej grafike je rekonštruovať objekty a pozíciu kamery v scéne z obrazov scény. Je to veľmi užitočné v medicínskom zobrazovaní, kde z viacerých obrazov niektorej časti tela, napr. mozgu, je možné vytvoriť 3D model. Vypočítanie homografie je kľúčovým bodom pri 3D rekonštrukcii, pretože je potrebné získať zobrazenia medzi bodmi v jednotlivých obrazoch.

3.2.3 Vizuálna metrológia

Vizuálna metrológia sa používa na výpočet veľkosti objektov a vzdialeností medzi nimi z obrazov daných objektov. Metrológia znamená štúdium merania a algoritmy vizuálnej metrológie automatizujú tento proces. Tento problém je veľmi významný, pretože často sú potrebné dôležité merania a bolo by veľmi drahé alebo zložité urobiť ich manuálne. Homografia dovoľuje vytvoriť z viacerých obrazov plochy jeden obraz, na ktorom sa dajú urobiť merania.

3.2.4 Stereo zobrazovanie

Stereo zobrazovanie je proces snímania hĺbky z viacerých obrazov scény. Tieto obrazy sú snímané z rôznych pozícií kamery, ale môžu byť snímané aj z jedného miesta s použitím fotometrického sterea. Analýzou vzdialenosti medzi dvoma obrazmi toho istého bodu v reálnom svete sa dá vypočítať disparita, ktorá je nepriamo súvisiaca s hĺbkou bodu. Stereo je značne skúmaný problém v počítačovej grafike a veľa algoritmov bolo navrhnutých na vypočítanie stereo vlastností scény reprezentovanej obrazmi. Hlavným bodom väčšiny algoritmov je nájdenie súhlasnosti medzi bodmi v obrazoch. S použitím epipolárnej geometrie môže byť hľadanie súhlasného bodu zjednodušené z hľadania v celom obraze na hľadanie na jednej priamke, ktorá sa nazýva epipolárna priamka.

3.3 Algoritmy na výpočet homografie

Táto podkapitola prezentuje niektoré z najpoužívanejších algoritmov na výpočet homografie. Tieto algoritmy sa používajú aj v knižnici OpenCV.

3.3.1 Základný DLT (Direct Linear Transform) algoritmus

DLT je jednoduchý algoritmus používaný na výpočet matice homografie zo zadanej množiny bodov. Keďže pracujeme s homogénnymi súradnicami, vzťah dvoch súhlasných bodov **x** a **x´** je definovaný takto:

$$c\begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = H\begin{pmatrix} x \\ y \\ 1 \end{pmatrix}, \tag{3.2}$$

kde *c* je nenulová konštanta, $(u \quad v \quad 1)^T$ reprezentuje $\mathbf{x}'$, $(x \quad y \quad 1)^T$ reprezentuje $\mathbf{x}$ a $H = \begin{bmatrix} h_1 & h_2 & h_3 \\ h_4 & h_5 & h_6 \\ h_7 & h_8 & h_9 \end{bmatrix}$.

Vydelením prvého riadku rovnice (*3.2*) tretím riadkom a druhého riadku tretím riadkom dostaneme tieto dve rovnice:

$$-h_1 x - h_2 y - h_3 + (h_7 x + h_8 y + h_9) u = 0 \tag{3.3}$$

$$-h_4 x - h_5 y - h_6 + (h_7 x + h_8 y + h_9) u = 0 \tag{3.4}$$

Rovnice (*3.3*) a (*3.4*) môžu byť zapísané v maticovej forme takto:

$$A_i \boldsymbol{h} = 0 \tag{3.5}$$

kde $A_i = \begin{pmatrix} -x & -y & -1 & 0 & 0 & 0 & ux & uy & u \\ 0 & 0 & 0 & -x & -y & -1 & vx & vy & v \end{pmatrix}$ a $\boldsymbol{h} = (h_1\ h_2\ h_3\ h_4\ h_5\ h_6\ h_7\ h_8\ h_9)^T$.

Keďže každý vzťah dvoch bodov poskytuje dve rovnice, štyri vzťahy dostačujú na vyriešenie ôsmich neznámych matice *H*, pričom žiadne tri body nesmú byť kolineárne. Štyri 2x9 A_i matice (jedna pre každý vzťah) sa spoja do jednej 8x9 matice *A*. V mnohých prípadoch je možné použiť viac ako štyri vzťahy, aby bolo zaistené robustnejšie riešenie. Koľkokoľvek vzťahov sa použije, ak sú všetky presné, tak *A* bude mať stále hodnosť 8 a bude existovať jedno homogénne riešenie. V praxi to však je tak, že vzťahy sú nepresné a tak ani riešenie nie je úplne presné.

3.3.2 RANSAC (Random Sample Consensus)

RANSAC je najviac používaný robustný algoritmus na výpočet homografie. Princíp algoritmu je, že pre niekoľko iterácií sa náhodne určí vzorka štyroch vzťahov a matica homografie *H* sa vypočíta z týchto vzťahov. Každý ďalší vzťah je potom zaradený medzi „inliery“ alebo „outliery“ (inliery sú prvky z množiny, ktorá má určité parametre a outliery sú prvky, ktoré nepatria do tejto množiny) podľa zhodnosti s *H*. Robustnosť algoritmu spočíva práve v tom, že výsledok nie je ovplyvnený prítomnosťou outlierov. Po dokončení všetkých iterácií sa vyberie iterácia s najväčším počtom inlierov. *H* je potom prepočítaná zo všetkých vzťahov, ktoré patria medzi inliery v danej iterácii. Dôležitý problém pri aplikácii algoritmu RANSAC je rozhodnutie, ako klasifikovať vzťahy ako inliery a outliery. Cieľom je určiť prah vzdialenosti (napr. medzi $\mathbf{x}'$ a $H\mathbf{x}$) *t* taký, že vzťah je inlier s pravdepodobnosťou α. Ďalší problém je určenie počtu iterácií algoritmu. Je takmer nerealizovateľné skúšať každú kombináciu 4 vzťahov, teda je potrebné určiť počet iterácií *N*, ktorý s pravdepodobnosťou *p* zaistí, že aspoň jeden z náhodných vzorkov bude bez outlierov. Je dokázané, že $N = \frac{\log(1-p)}{\log(1-(1-\epsilon)^S)}$, kde ε je pravdepodobnosť, že vzťah je outlier a S je počet vzťahov použitých v každej iterácii.

3.3.3 PROSAC (Progressive Sample Consensus)

Štruktúra algoritmu PROSAC je podobná algoritmu RANSAC. Najprv sa náhodne vygenerujú vzorky, ktoré, na rozdiel od RANSAC-u, nie sú vytvárané zo všetkých dát, ale z podmnožiny dát s najvyššou

kvalitou. Veľkosť množiny vzoriek sa stále zvyšuje, takže vzorky s väčšou pravdepodobnosťou byť inliermy sú preskúmané najskôr. V podstate PROSAC vytvára vzorky rovnaké ako RANSAC, ale v inom poradí. Vďaka tejto zmene poradia dokáže PROSAC ušetriť výpočty a dosahovať rýchlosti 10 až 100-krát väčšie ako RANSAC. Algoritmus je ukončený, keď pravdepodobnosť existencie riešenia lepšieho ako doterajšie najlepšie riešenie klesne pod 5%. Tento algoritmus je použitý na výpočet homografie v aplikácii *CardboardKeyboard*, pretože je to najrýchlejší algoritmus na výpočet homografie, ktorý najnovšia verzia knižnice OpenCV poskytuje.

3.3.4 LMEDS (Least Median of Squares Regression)

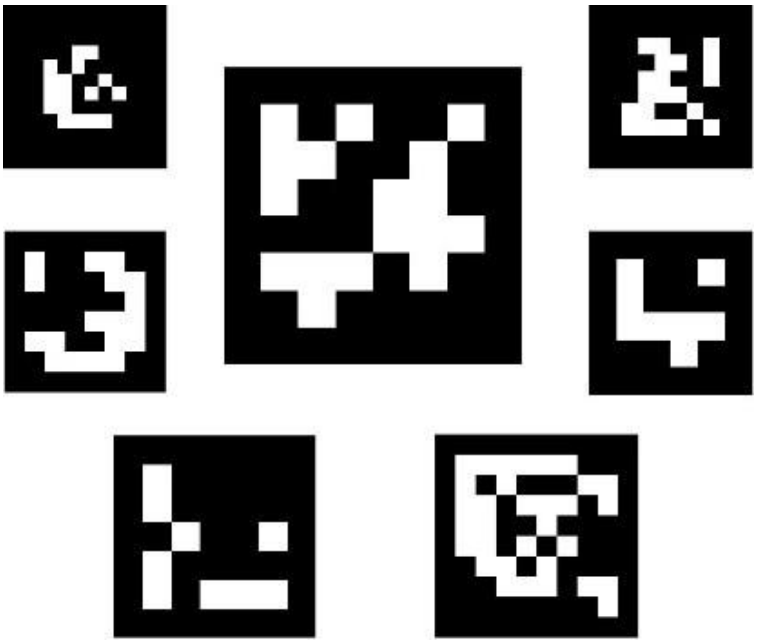
Ako bolo uvedené vyššie, algoritmus RANSAC robí rozhodnutia na základe počtu vzťahov spadajúcich do istého intervalu určeného prahom vzdialenosti. Toto je jeden zo spôsobov riešenia faktu, že algoritmy ako verzia DLT s algebraickou vzdialenosťou nie sú veľmi robustné, s ohľadom na outliery. Na tému vylepšenia robustnosti regresných metód sa robí veľa výskumov. Jedným príkladom je nahradiť umocnenú vzdialenosť absolútnou hodnotou vzdialenosti. Toto zlepší robustnosť, pretože outliery nie sú penalizované tak prísne ako keď sú umocnené. Populárna metóda pre výpočet homografie je LMEDS – najmenší medián štvorcov. Táto metóda nahrádza súčet mocnín vzdialeností mediánom. LMEDS pracuje veľmi dobre, ak počet outlierov je menší ako 50% a jej výhoda oproti RANSAC-u je, že nevyžaduje žiadne určovanie prahu alebo znalosť očakávaného počtu chýb. Hlavná nevýhoda LMEDS spočíva v tom, že nefunguje pri počte outlierov väčšom ako polovica.

4 ArUco markery

Táto kapitola čitateľovi priblíži ArUco markery, ktoré sú použité na detekciu kláves v obraze. Informácie sú čerpané z [5]. ArUco markery boli vyvinuté v Ústave výpočetnej a numerickej analýzy na Univerzite v Cordobe, Názov ArUco je akronym skratiek AR - Augmented reality a UCO – University of Cordoba.

Určovanie pozície kamery je veľmi dôležité v mnohých aplikáciách, kde sa využíva počítačové videnie, napr. rozšírená realita, navigácia robotov atď. Tento proces je založený na vyhľadaní vzťahov medzi bodmi v reálnom priestore a ich 2D projekciou. Toto je zvyčajne náročné a preto sa používajú umelo vytvorené markery, aby sa proces uľahčil a zrýchlil. Jeden z najpopulárnejších prístupov je použitie štvorcových markerov s binárnym kódom vo vnútri. Hlavná výhoda takýchto markerov je, že už jeden marker poskytuje dostatok vzťahov (jeho štyri rohy) na určenie pozície kamery. Vnútorný binárny kód im taktiež umožňuje aplikovať detekciu a korekciu chýb.

ArUco marker je štvorcový marker zložený z čierneho rámu a vnútornej binárnej matice určujúcej jeho identifikátor. Čierny rám pomáha pri rýchlej detekcii v obraze a binárna kodifikácia umožňuje identifikáciu markeru a aplikáciu techník na detekciu a korekciu chýb. Veľkosť markeru určuje veľkosť vnútornej matice, napr. marker veľkosti 4x4 má 16-bitovú maticu.



Obrázok 4.1: Príklady ArUco markerov

Marker sa môže v prostredí nachádzať akokoľvek otočený a proces detekcie musí byť schopný určiť jeho originálnu rotáciu tak, že každý roh je jednoznačne identifikovaný. Tomuto tiež napomáha binárna kodifikácia.

ArUco markery sú združované to tzv. adresárov, ktoré obsahujú binárne kodifikácie markerov rovnakej veľkosti s identifikátormi od 1 po veľkosť adresára. Teda hlavnými vlastnosťami adresára je jeho veľkosť a veľkosť markerov, ktoré obsahuje. ArUco modul v OpenCV poskytuje niekoľko preddefinovaných adresárov rôznych veľkostí.

Identifikátor markeru nie je, ako by sa mohlo zdať, číslo získané skonvertovaním binárnej kodifikácie na číslo v desiatkovej sústave. Dôvod je ten, že takéto číslo by pre markery s väčšou veľkosťou bolo

obrovské a tým pádom nepraktické na ukladanie a použitie v programe. Identifikátor markeru je namiesto toho jeho index v rámci adresára, ku ktorému patrí.

4.1 Generovanie adresárov

Pri navrhovaní a generovaní adresárov markerov je dôležité brať do úvahy najmä množstvo chýb prvého a druhého druhu (chyby v testovacom procese, kde niečo, čo má byť prijaté, je vylúčené a naopak), medzimarkerovú zámenu a počet validných markerov. Chyby prvého a druhého druhu a medzimarkerová zámena sa obvykle riešia bitmi pre detekciu a korekciu chýb, ktoré na druhú stranu redukujú počet validných markerov v adresári. Počet validných markerov závisí na Hammingovej vzdialenosti markerov. Ak je príliš malá, niekoľko chybných bitov môže viesť ku detekcii iného markeru z adresára a táto chyba nebude rozpoznaná. Ďalšia požadovaná vlastnosť markerov je veľký počet prechodov medzi bitmi, aby marker nebol považovaný za obyčajný objekt v priestore a naopak. Napr. binárny kód pozostávajúci len z núl tvorí čierny štvorec, ktorý by sa veľmi ľahko dal pomýliť s nejakým objektom.

4.2 Rozmiestnenie markerov na klávesoch

Obrázok 4.2: Rozmiestnenie markerov na klávesoch

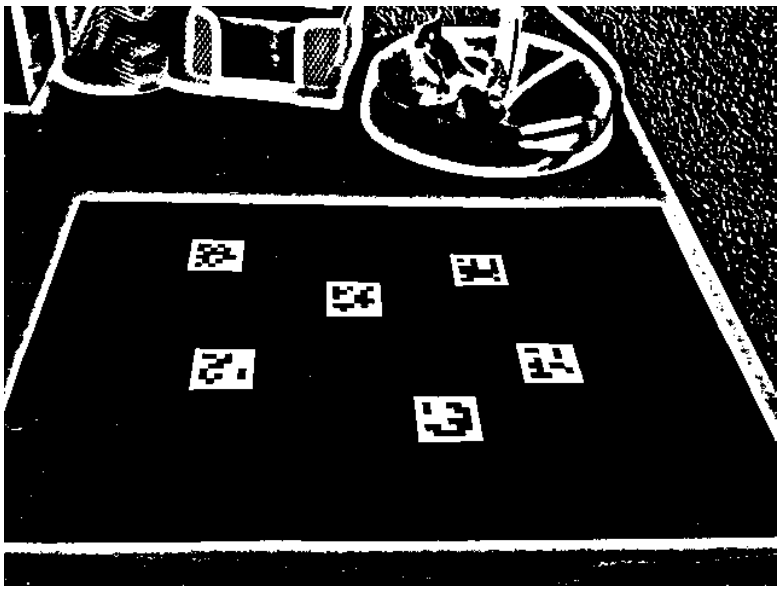
Ako už bolo spomenuté vyššie, ArUco modul pre knižnicu OpenCV poskytuje preddefinované adresáre, ktoré pokrývajú niekoľko veľkostí adresárov aj markerov, Čím väčší je adresár, tým je nižšia vzdialenosť medzi markermi a vyššia pravdepodobnosť chyby. Taktiež veľkosť markerov má vplyv na vzdialenosť, tu ale platí priama úmernosť, t.j. čím sú markery väčšie, tým je väčšia vzdialenosť medzi nimi. V aplikácii *CardboardKeyboard* je použitý adresár veľkosti 50 s veľkosťou markerov 4x4. Tento adresár bohato postačuje aj na klávesy s 88 klávesmi, pretože počet potrebných markerov sa dá vypočítať ako *2*n + 2*, kde *n* je počet oktáv, teda pre sedem oktáv 88-klávesových klávesov postačuje 16 markerov. Adresár veľkosti 50 je použitý preto, že je najmenší z poskytovaných preddefinovaných adresárov a teda jeho markery majú medzi sebou veľkú vzdialenosť. Veľkosť markerov 4x4 je určená z dôvodu jednoduchšej a tým pádom aj rýchlejšej detekcie, keďže čím viac bitov marker obsahuje, tým viac ich treba analyzovať a spomaľuje sa proces detekcie. Veľkosť markerov 4x4 je najmenšia podporovaná veľkosť. Markery treba na klávesy rozmiestniť tak, že všetky markery s nepárnym identifikátorom sú nad klávesami C a markery s párnym identifikátorom sú v spodnej časti kláves C.

Takto je zaručené, že štyri markery ohraničujú oktávu a potom je možné vypočítať homografiu a aplikovať perspektívnu projekciu na virtuálne objekty. Začína sa vždy markerom s identifikátorom jeden na najnižšej možnej klávese C, t.j. tá, ktorá je na klávesoch najviac vľavo. Rozmiestnenie markerov je možné vidieť na Obrázok 4.2.

4.3 Detekcia markerov

V tejto podkapitole sú vysvetlené kroky potrebné pre automatickú detekciu markerov v obraze. Tento proces sa skladá z viacerých krokov a jeho cieľom je nájsť v obraze markery a extrahovať z nich binárny kód.

1. **Segmentácia obrazu:** Najprv sa extrahujú najvýraznejšie kontúry z čiernobieleho obrazu. Používa sa adaptívne prahovanie (viď Obrázok 4.3), ktoré je robustné pri rôznych svetelných podmienkach.

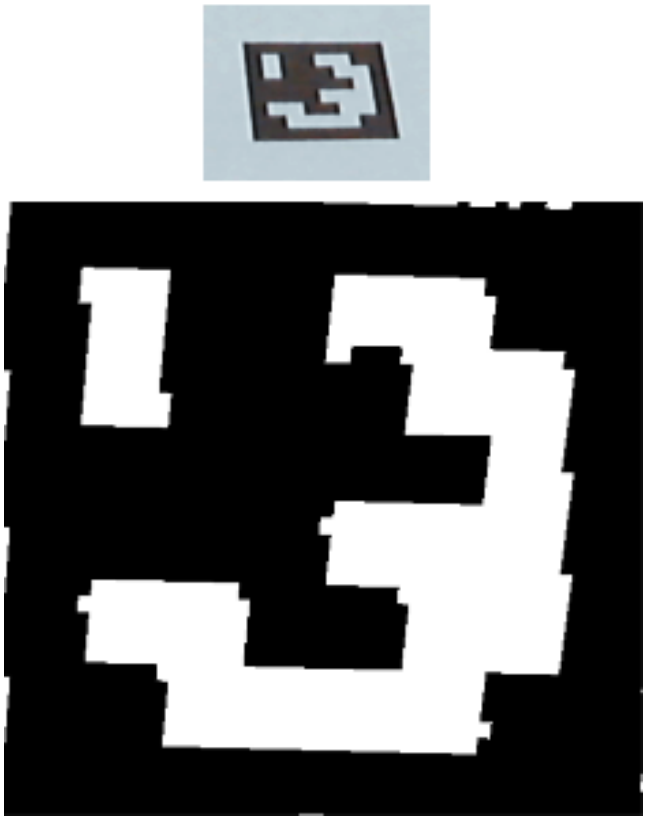


Obrázok 4.3: Adaptívne prahovanie

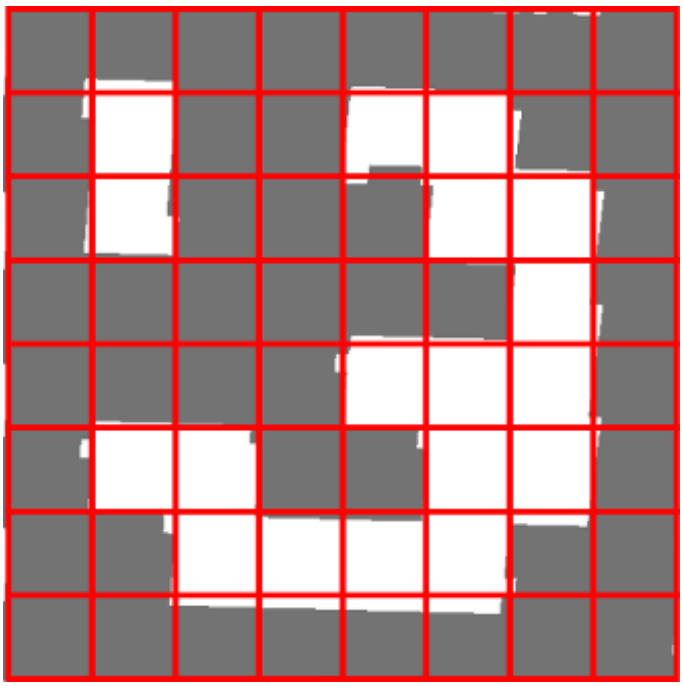
2. **Extrakcia kontúr a filtrácia:** Potom, ako sa na obraz aplikuje prahovanie, vykoná sa extrakcia kontúr, ktorá vytvorí množinu kontúr, z ktorých väčšina je irelevantná. Následne sa vykoná polygonálna aproximácia pomocou algoritmu Douglas-Peucker[6]. Keďže markery sú uzavreté v obdĺžnikových kontúrach, všetky kontúry, ktoré nie sú podobné štvor-vrcholovým polygónom, sú vyradené. Nakoniec sa zjednodušia blízke kontúry tak, že sa ponechajú len okrajové.

3. **Extrakcia kódu markerov:** Ďalší krok spočíva v analýze vnútornej oblasti kontúr s cieľom extrahovania vnútorného kódu. Najskôr sa odstráni perspektívna projekcia vypočítaním matice homografie (viď Obrázok 4.4). Na výsledný obraz sa aplikuje Otsuho prahovanie, aby sa rozdelili biele a čierne pixely. Potom sa obraz rozdelí na mriežku s rovnakým počtom buniek ako počet bitov v markeri (viď Obrázok 4.5). V každej bunke sa spočítajú biele a čierne pixely a podľa toho sa určí hodnota bitu priradeného bunke. Ak má viac čiernych pixelov, nadobúda

hodnotu 0, ak má viac bielych pixelov, nadobúda hodnotu 1. Prvý vyraďovací test kontroluje, či marker obsahuje čierny okraj. Ak všetky okrajové bity majú hodnotu 0, tak sa vnútorná mriežka analyzuje metódou popísanou nižšie.



Obrázok 4.4: Odstránenie perspektívnej projekcie



Obrázok 4.5: Marker rozdelený na mriežku

4. **Identifikácia markerov a korekcia chýb:** V tomto kroku je nutné určiť, ktoré kontúry sú skutočne markery a patria do adresára a ktoré sú len časťami prostredia. Keď je kód kontúry extrahovaný, získajú sa štyri rôzne identifikátory (jeden pre každú možnú rotáciu). Ak sa aspoň jeden z nich nachádza v adresári, kontúra je považovaná za validný marker. Kvôli zrýchleniu tohto procesu sú prvky adresára uložené vo vyváženom binárnom strome a markery sú reprezentované celočíselnou hodnotou získanou sčítaním všetkých ich bitov. Tento proces ma logaritmickú zložitosť $O(4 \log_2(|D|))$, kde D je adresár a faktor 4 indikuje, že je potrebné jedno hľadanie pre každú rotáciu. Ak sa nenájde žiadna zhoda, môže sa použiť korekčná metóda. Vzhľadom na to, že minimálna vzdialenosť medzi dvomi markermi v D je ε, je možné detegovať a opraviť chybu najviac $[(\varepsilon - 1) / 2]$ bitov. Teda metóda detekcie a korekcie chýb spočíva vo vypočítaní vzdialenosti chybnej kontúry voči všetkým markerom v D. Ak je vzdialenosť rovnaká alebo menšia ako $[(\varepsilon - 1) / 2]$, najbližšia rotácia kontúry je považovaná za validný marker. Tento proces má lineárnu zložitosť $O(4(|D|))$, pretože každá rotácia kontúry sa musí porovnať s celým adresárom. Každopádne je možné tento proces paralelizovať a na súčasných počítačoch sa dá veľmi efektívne implementovať.

5 Návrh a implementácia

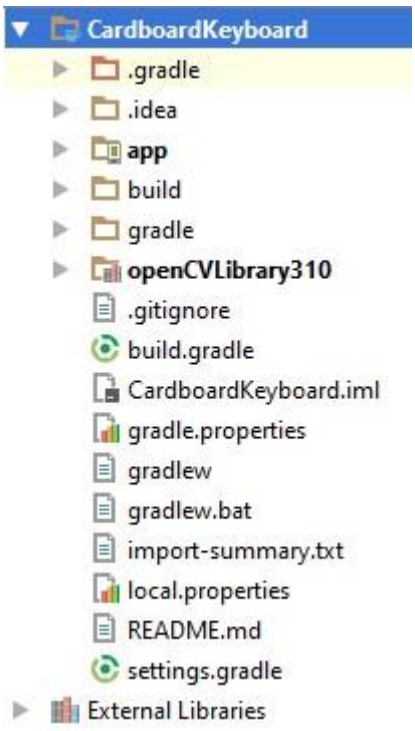
Táto kapitola popisuje štruktúru aplikácie, použité technológie, postupy, algoritmy a spôsoby detekcie a vykresľovania virtuálnych objektov do obrazu na plochu kláves. Aplikácia *CardboardKeyboard* je aplikácia pre smartfóny s operačným systémom Android verzie 4.4 a vyššej. Na vývoj aplikácie bolo použité vývojové prostredie Android Studio 2.1. Android Studio používa na zostavenie aplikácie zostavovací systém Gradle verzie 2.10. Časť aplikácie, ktorá získava dáta z kamery, je implementovaná v programovacom jazyku Java a časť vykonávajúca detekciu a vykresľovanie je implementovaná v jazyku C++. V obidvoch častiach je použitá knižnica OpenCV verzie 3.1.0. OpenCV je open source knižnica určená na prácu s počítačovou grafikou v reálnom čase. Je implementovaná v C++ a podporuje viacero platforiem, napr. Windows, Linux, OS X a taktiež mobilné platformy Android, iOS a.i.

5.1 Android Native Development Kit

V tejto podkapitole je vysvetlený pojem Android NDK, informácie sú čerpané z [7]. Android NDK je sada nástrojov umožňujúca implementovať časti aplikácie s použitím kompilovaných programovacích jazykov C alebo C++. Typicky sa používa v aplikáciách, ktoré vyžadujú vysoký výpočetný výkon, teda práca s obrazom, spracovanie signálov atď., alebo keď je potrebné použiť existujúcu knižnicu implementovanú v C alebo C++. NDK kompiluje kód jazykov C a C++ do binárneho kódu, z ktorého potom generuje dynamické knižnice. Funkcie z týchto knižníc začínajúce prefixom `Java_` sa pomocou Java Native Interface (rozhranie zabezpečujúce komunikáciu komponentov Javy a C alebo C++) namapujú na príslušné metódy tried v kóde Javy. V Jave je potrebné metódy deklarovať pomocou kľúčového slova `native`, aby bolo jasné, že implementácia metódy pochádza z dynamickej knižnice. Dynamické knižnice sú v Jave načítavané pomocou funkcie `System.loadLibrary`, táto sa typicky vkladá do statického bloku triedy, ktorá obsahuje metódy z dynamických knižníc. Aby dynamické knižnice správne fungovali, musia byť umiestnené v špecifickom adresári *jniLibs*. O generovanie dynamických knižníc zo zdrojového kódu kompilovaných jazykov sa stará skript ndk-build, ktorý je spúšťaný zostavovacím skriptom build.gradle. ndk-build potrebuje súbor Android.mk, kde je definované meno knižnice, mená zdrojových súborov, príznaky pre kompilátor a linkované knižnice. Taktiež je možné vytvoriť nepovinný súbor Application.mk, ktorý popisuje ABI (Application Binary Interface), verziu Androidu a štandardné knižnice používané celou aplikáciou. ABI určuje, na akú konkrétnu procesorovú architektúru sa generuje strojový kód. Štandardne je ABI nastavené na armeabi, teda pre ARM procesory. Aplikácia *CardboardKeyboard* má však nastavené armeabi-v7a, tiež pre ARM procesory, ale s podporou hardvérových operácií s plávajúcou desatinnou čiarkou, vďaka čomu je dosiahnutá vyššia rýchlosť.

5.2 Súborová štruktúra aplikácie

Táto podkapitola popisuje logické členenie aplikácie na rôzne adresáre, ktoré obsahujú potrebné súbory, od zdrojových súborov aplikácie cez zdrojové súbory knižnice OpenCV a zdieľané knižnice až



Obrázok 5.1: Koreňový adresár CardboardKeyboard

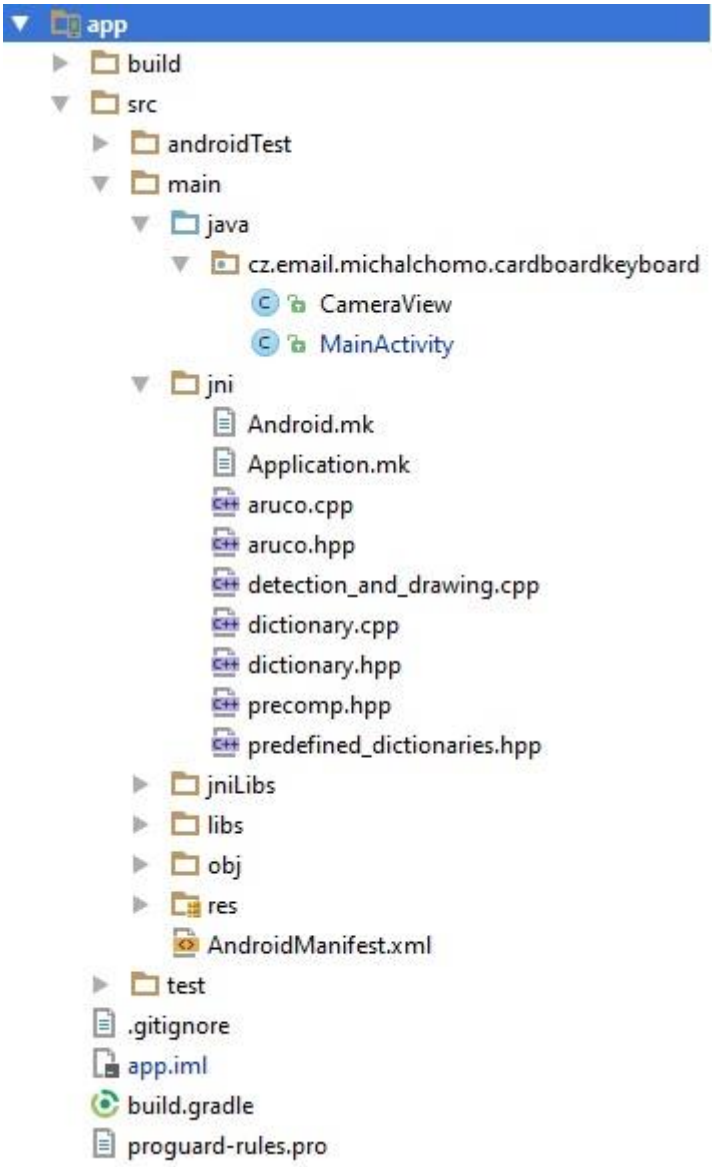
po zostavovacie skripty. Koreňový adresár je zobrazený na Obrázok 5.1. Účel jednotlivých adresárov a súborov je nasledujúci:

- **.gradle:** Cache a binárne súbory generované zostavovacím systémom Gradle.
- **.idea:** Konfiguračné súbory IntelliJ IDEA (Android Studio je postavené na tomto IDE).
- **app:** Hlavný adresár aplikácie obsahujúci zdrojové kódy Java aj C++ časti.
- **build:** Výstupné súbory procesu make pre všetky moduly.
- **gradle:** Gradle wrapper – zariadi inštaláciu správnej verzie Gradle-u podľa konfiguračného súboru gradle-wrapper.properties.
- **openCVLibrary310:** Adresár obsahujúci Java zdrojové kódy knižnice OpenCV.
- **build.gradle:** Konfiguračný súbor zostavovacieho systému Gradle pre všetky moduly.
- **gradle.properties:** Globálne nastavenia Gradle-u.
- **gradlew:** Spúšťací skript Gradle-u pre UNIX.
- **gradle:** Spúšťací skript Gradle-u pre Windows.
- **local.properties:** Súbor špecifikujúci cestu ku SDK a NDK na konkrétnom PC.
- **settings.gradle:** Súbor určujúci, ktoré moduly sa majú zostaviť.

Hlavný adresár aplikácie je zobrazený na Obrázok 5.2. Adresáre a súbory sú popísané nasledovne:

- **build:** Výstupné súbory procesu make pre modul app.
- **src:** Adresár obsahujúci všetky zdrojové súbory aplikácie s výnimkou zdrojových súborov knižnice OpenCV.
    - **main/java/.../CameraView.java:** Trieda umožňujúca manipulovať s nastaveniami kamery, v aplikácii *CardboardKeyboard* je použitá len na nastavovanie rozlíšenia.
    - **main/java/.../MainActivity.java:** Hlavná trieda Java časti aplikácie. Predstavuje hlavnú Androidovú aktivitu aplikácie, načítava dynamické knižnice, získava obrazové

dáta z kamery, volá funkcie z dynamickej C++ knižnice a rozdeľuje výsledný obraz na dve totožné polovice.



Obrázok 5.2: Hlavný adresár aplikácie

- **main/jni:** Adresár združujúci C++ zdrojové súbory a vyššie popísané súbory Android.mk a Application.mk.
  - **main/jni/detection_and_drawing.cpp:** Najdôležitejší súbor aplikácie, obsahuje detekciu markerov, vykresľovanie mien tónov a vizualizáciu akordov.
  - **main/jni/ostatné .cpp a .hpp súbory:** Zdrojové kódy ArUco modulu OpenCV knižnice.
- **main/jniLibs:** Prekompilované dynamické a statické časti knižnice OpenCV.
- **main/libs:** Dynamická knižnica s názvom imageproc vygenerovaná zo súborov v main/jni.
- **main/res:** Adresár združujúci rôzne zdroje pre aplikáciu, napr. ikony, obrázky, rozloženie UI elementov v pohľade, textové reťazce a pod. V aplikácii

CardboardKeyboard nenájde veľké uplatnenie, keďže pohľad je definovaný len jeden a o všetko vykresľovanie a manipuláciu s obrazom sa stará OpenCV a dynamická knižnica imageproc.

- **main/AndroidManifest.xml:** Súbor popisujúci aplikáciu, obsahuje informácie potrebné pre systém, aby mohol aplikáciu spúšťať. Určuje minimálnu verziu SDK, oprávnenia pre aplikáciu, vlastnosti jednotlivých aktivít atď.

- **build.gradle:** Konfiguračný súbor pre zostavovací systém, obsahuje definície cieľového a minimálneho API, potrebné závislosti a spúšťa skript ndk-build pre preklad a zostavenie dynamickej knižnice z C++ súborov v aplikácii.

5.3 Inicializácia a získavanie dát z kamery

O získavanie dát z kamery smartfónu sa stará časť aplikácie implementovaná v jazyku Java. Táto časť pozostáva z dvoch tried `MainActivity` a `CameraView`. `MainActivity` je trieda, ktorá dedí zo vstavanej Androidovej triedy `Activity`. Aktivita je v Androide jeden zo základných blokov aplikácie a reprezentuje jednu obrazovku s užívateľským rozhraním. Typicky je v aplikácii viacero aktivít, ktoré ponúkajú užívateľovi rôznu funkčnosť aplikácie. V aplikácii *CardboardKeyboard* však nie je potrebné prepínať medzi aktivitami, teda `MainActivity` je hlavnou a jedinou aktivitou aplikácie. Trieda `MainActivity` implementuje rozhranie `CvCameraViewListener2` z knižnice OpenCV. Rozhranie `CvCameraViewListener2` obsahuje tri metódy `onCameraViewStarted`, `onCameraViewStopped` a `onCameraFrame`, ktoré sú implementované triedou `MainActivity` a budú popísané nižšie. `CameraView` je trieda pre nastavovanie rozlíšenia, prípadne FPS kamery. Dedí od triedy `JavaCameraView` a redefinuje niektoré metódy tejto triedy.

Na začiatku triedy `MainActivity` sú definované členské konštanty a deklarované členské premenné, ako vidno v Kód 5.1. Konštanta `TAG` slúži na identifikáciu triedy pri chybových a logovacích výpisoch, konštanty `ResolutionX` a `ResolutionY` určujú rozlíšenie obrazov z kamery. Členská premenná `mCameraView` predstavuje Androidový pohľad, základný blok pri vytváraní užívateľského rozhrania. Pohľad je možné si predstaviť ako obdĺžnikovú oblasť, kde sú združené nejaké objekty užívateľského rozhrania, pričom pohľad môže reagovať na zmeny vo svojom vnútri. Aplikácia má väčšinou v jednej aktivite viac pohľadov, ale v aplikácii *CardboardKeyboard* to nie je potrebné.

```
private static final String TAG = "MainActivity";
private static final int ResolutionX = 800;
private static final int ResolutionY = 480;

private CameraView mCameraView;

private Mat mRgba;
private Mat mGray;
private Mat mHalf;
```

Kód 5.1:Deklarácia členských konštánt a premenných

Členské premenné `mRgba`, `mGray` a `mHalf` sú matice knižnice OpenCV pre farebný obraz, čiernobiely obraz a rozpolený farebný obraz z kamery. Trieda `MainActivity` taktiež obsahuje načítavanie knižnice OpenCV a dynamickej knižnice imageproc, ktoré je možné vidieť v Kód 5.2.

```
static {
    try {
        System.loadLibrary("imageproc");
        System.loadLibrary("opencv_java3");
    } catch (Exception e) {
        Log.e(TAG, e.getMessage());
    }
}
```

Kód 5.2: Načítavanie knižníc

OpenCV sa v moderných verziách väčšinou nevkladá priamo do aplikácie, ale načítava sa pomocou pomocnej triedy `OpenCVLoader`. V takom prípade však treba nainštalovať na smartfón aplikáciu OpenCV Manager. V prípade načítania pomocou `System.loadLibrary` sa užívateľ nemusí zaoberať inštalovaním OpenCV Manageru, pretože OpenCV je zahrnutá v aplikácii. Jedna nevýhoda tohto prístupu je o niečo väčšie APK.

`MainActivity` implementuje klasické metódy pre Androidovú aktivitu, ktorými sú `onCreate`, `onDestroy`, `onResume` a `onPause`. Vo všetkých týchto metódach sa najskôr volá implementácia z rodičovskej triedy. V metóde `onCreate` dochádza ku inicializácii aktivity, nastavuje sa tu rozloženie obsahu (vyberá sa zo zdrojov), pohľadu `mCameraView` sa priradí pohľad z tohto rozloženia, aktivita sa nastaví ako prijímač obrazov z kamery a povolí sa spojenie s kamerou. Metóda `onDestroy` sa volá predtým, ako sa aktivita končí, v tejto metóde sa odpája spojenie s kamerou. V metóde `onPause`, ktorá je volaná, keď užívateľ opustí aktivitu, ale aktivita stále beží v pozadí, sa tiež odpája spojenie s kamerou.

Nastavenie rozlíšenia kamery sa vykonáva v metóde `onCameraViewStarted` z rozhrania `CvCameraViewListener2`.

```
public void onCameraViewStarted(int width, int height) {
    List<Camera.Size> resList = mCameraView.getResolutionList();

    ListIterator<Camera.Size> resolutionItr = resList.listIterator();
    while(resolutionItr.hasNext()) {
        Camera.Size element = resolutionItr.next();
        if(element.width == ResolutionX && element.height == ResolutionY) {
            mCameraView.setResolution(element);
            break;
        }
    }
}
```

Kód 5.3: Nastavenie rozlíšenia

Ako je vidieť v Kód 5.3, rozlíšenie sa nastavuje výberom z dostupných rozlíšení, ktoré sa získavajú metódou `getResolutionList` triedy `CameraView`. Dostupné rozlíšenia vo forme zoradenej kolekcie sa uložia do premennej `resList`, ktorou sa potom iteruje pomocou iterátora a ak sa nájde rozlíšenie zhodujúce sa s konštantami `ResolutionX` a `ResolutionY`, toto rozlíšenie sa nastaví. Najdôležitejšou metódou `MainActivity` je metóda `onCameraFrame`, ktorá sa volá pri každom získanom obraze z kamery a vracia obraz, ktorý sa zobrazí na displeji. Najskôr sa do členských matíc

`mRgba` a `mGray` vloží farebný a čiernobiely obraz z kamery a potom sa zavolá metóda `detectMarkersAndDraw` z dynamickej knižnice imageproc. Tejto metóde sú predané obidve matice `mRgba` a `mGray` pomocou metódy `getNativeObjAddress`, ktorá jej predá adresy matíc. Vykresľovanie virtuálnych objektov v C++ časti bude popísané nižšie. Po spracovaní nasleduje proces horizontálneho rozdelenia obrazu na dve totožné polovice. Začína skopírovaním matice `mRgba` do matice `mHalf`. Pokračuje zmenou veľkosti `mHalf` na polovičný počet stĺpcov. Následne sa `mHalf` nakopíruje do submatíc ľavej a pravej horizontálnej polovice `mRgb`. Nakoniec sa uvoľní pamäť alokovaná pre maticu `mHalf` a vráti sa matica `mRgba`, ktorá už obsahuje virtuálne objekty a je rozdelená na dve horizontálne polovice. Funkcia `onCameraFrame` je zobrazená v Kód 5.4.

```
public Mat onCameraFrame(CameraBridgeViewBase.CvCameraViewFrame inputFrame) {
    mRgba = inputFrame.rgba();
    mGray = inputFrame.gray();

    detectMarkersAndDraw(mGray.getNativeObjAddr(), mRgba.getNativeObjAddr());

    mHalf = mRgba.clone();

    try {
        Imgproc.resize(mHalf, mHalf, new Size(mHalf.cols() / 2, mHalf.rows()), 0,
            0, Imgproc.INTER_LINEAR);
        mHalf.copyTo(mRgba.submat(new Rect(0, 0, mHalf.cols(), mHalf.rows())));
        mHalf.copyTo(mRgba.submat(new Rect(mHalf.cols(), 0, mHalf.cols(),
            mHalf.rows())));
    } catch (Exception e) {
        Log.e(TAG, e.getMessage());
    }

    mHalf.release();

    return mRgba;
}
```

Kód 5.4: Funkcia onCameraFrame

5.4 Detekcia markerov

Detekcia markerov a vykresľovanie virtuálnych objektov do obrazu sú implementované v jazyku C++. Hlavnou funkciou tejto časti je funkcia `detectMarkersAndDraw`, ktorá je ako jediná volaná z Java kódu. Ostatné funkcie v C++ časti nie sú deklarované tak, aby sa mohli volať z Java kódu, teda sú buď volané z funkcie `detectMarkersAndDraw` alebo medzi sebou. Hlavička funkcie `detectMarkersAndDraw` je zobrazená v Kód 5.5. `JNIEXPORT` je makro, ktoré znamená, že funkcia sa má zapísať do dynamickej tabuľky vygenerovanej dynamickej knižnice, aby ju JNI mohlo lokalizovať v Jave. `JNICALL` je makro obsahujúce príznaky prekladača. Prefix Java_ bol spomínaný vyššie, po ňom nasleduje meno balíku, meno aktivity a názov funkcie. Parameter `*env` je ukazovateľ na virtuálny stroj Javy, parametre `mAddrGr` a `mAddrRgba` sú ukazovatele na matice s čiernobielym a farebným obrazom z kamery.

```
JNIEXPORT void JNICALL
Java_cz_email_michalchomo_cardboardkeyboard_MainActivity_detectMarkersAndDraw
(JNIEnv *env, jlong matAddrGr, jlong matAddrRgba)
```

Kód 5.5: Hlavička funkcie detectMarkersAndDraw

Funkcia `detectMarkersAndDraw` začína prevzatím matíc z adries predaných parametrami. V Kód 5.6 je možné vidieť, že obidve adresy matíc sú najskôr pretypované na ukazovateľ na typ `Mat` a následne dereferencované a uložené ako referencie na matice predané parametrami do `mGr` a `mRgb`.

```
Mat &mGr = *(Mat *) matAddrGr;
Mat &mRgb = *(Mat *) matAddrRgba;
```

Kód 5.6: Prevzatie matíc s obrazmi z kamery

Nasledujú deklarácie vektorov `markerIds` a `markerCorners`. `markerIds` je vektor celých čísel, do ktorého sa ukladajú identifikátory detegovaných markerov v poradí v akom boli detegované. Do vektoru `markerCorners` sa ukladajú vektory štyroch bodov typu `Point2f` reprezentujúcich pozície štyroch rohov daného markeru. `Point2f` je typ knižnice OpenCV, ktorý obsahuje x-ovú a y-ovú súradnicu bodu v obraze typu `float`. Prvý bod je vždy ľavý horný roh a ďalej ide číslovanie v smere hodinových ručičiek. Ďalej je definovaný inteligentný ukazovateľ `dictionary` na adresár markerov, ktorý je získaný funkciou `getPredefinedDictionary`. Adresár obsahuje 50 markerov veľkosti 4x4. Následne sa volá funkcia `detectMarkers` z ArUco modulu knižnice OpenCV. Táto funkcia je obalená try-catch blokom a prijíma tieto vstupné parametre:

- Matica obrazu, v ktorom sa detegujú markery
- Adresár, ktorého markery sa budú porovnávať s detegovanými
- Vektor pre uloženie vektorov bodov s pozíciami rohov markerov
- Vektor pre uloženie identifikátorov markerov

Funkcii `detectMarkers` sa teda predávajú parametre `mGr`, `dictionary`, `markerCorners` a `markerIds`. Na konci funkcie `detectMarkersAndDraw` je blok s podmienkou, ktorý vidieť v Kód 5.7.

```
if(markerIds.size() > 3) {
    try {
        draw(mRgb, markerCorners, getSortedIds(markerIds));
    } catch (cv::Exception& e) {
        __android_log_print(ANDROID_LOG_VERBOSE, APPNAME, "%s", e.what());
    }
}
```

Kód 5.7: Volanie funkcie pre vykresľovanie virtuálnych objektov

V podmienke sa testuje, či celkový počet detegovaných markerov presiahol tri. Ak áno, zavolá sa funkcia `draw` pre vykresľovanie. Dôvod testovania voči trom je, že virtuálne objekty sa vykresľujú vždy do jednej oktávy, ktorá je ohraničená štyrmi markermi a z bodov týchto markerov sa počíta homografia, aby sa mohla aplikovať perspektívna projekcia na virtuálne objekty. Funkcia `draw` prijíma ako parametre maticu s farebným obrazom z kamery, vektor vektorov s bodmi rohov markerov a vektor indexov markerov zoradených podľa identifikátorov získaný funkciou `getSortedIds`.

5.5 Zoradenie markerov podľa identifikátorov

Funkciu `getSortedIds` je možné vidieť v Kód 5.8. Na začiatku funkcie je definovaný vektor celých čísel `sortedIds`, ktorý má fixnú veľkosť určenú makrom `SORTED_IDS_SIZE` s inicializačnými hodnotami -1 pre každý element. Do tohto vektoru sa neskôr ukladajú zoradené identifikátory. Ďalej je definované počítadlo `i` začínajúce od nuly, ktoré reprezentuje originálne poradie markeru. Posledná definovaná premenná je `minId`, kde sa neskôr uloží najmenší identifikátor spomedzi detegovaných markerov a tento sa potom vloží na koniec vektoru `sortedIds`, aby bol jednoducho prístupný. Na začiatku sa do premennej `minId` vloží počet markerov vo vektore `sortedIds`, teda akoby identifikátor s maximálnou hodnotou.

```
vector<int> getSortedIds(vector<int> &markerIds) {
    vector<int> sortedIds(SORTED_IDS_SIZE, -1);
    int i = 0;
    int minId = SORTED_IDS_SIZE;
    for(vector<int>::iterator it = markerIds.begin();it != markerIds.end();it++) {
        if(*it < SORTED_IDS_SIZE){
            sortedIds.at(*it) = i;
        } else {
            break;
        }
        ++i;
        if(*it < minId) minId = *it;
    }
    sortedIds.erase(remove(sortedIds.begin(), sortedIds.end(), -1),
       sortedIds.end());
    sortedIds.push_back(minId);
    return sortedIds;
}
```

Kód 5.8: Funkcia getSortedIds

Vo `for` cykle sa iteruje cez pôvodný vektor identifikátorov zoradených podľa poradia, ako boli detegované. Do vektoru `sortedIds` sa vloží hodnota počítadla `i` na index identifikátoru markeru. Vďaka tomuto je potom možné iterovať cez stúpajúce hodnoty identifikátorov a pritom sa dostať ku správnemu vektoru s bodmi rohov markeru. Ďalej sa inkrementuje počítadlo `i` a hodnota aktuálneho markeru sa porovnáva s najmenším identifikátorom. Ak je hodnota aktuálneho markeru menšia, uloží sa do `minId`. Po cykle sa vymažú z vektoru `sortedIds` všetky „prázdne“ elementy, teda tie s hodnotou -1. Na odstránenie je použitý erase-remove idióm, kde funkcia `remove` presunie všetky elementy s hodnotou -1 za zvyšné elementy a vráti iterátor na prvý element s hodnotou -1. Funkcia `erase` potom vymaže všetky elementy od tohto iterátoru až po koniec vektoru. Vymaže sa aj nultý element vektoru, teda celé číslovanie identifikátorov markerov sa posunie o jeden dozadu. Na konci funkcie sa ešte na koniec vektoru `sortedIds` vloží najmenší identifikátor a funkcia `getSortedIds` vráti vektor `sortedIds`.

5.6 Vykresľovanie virtuálnych objektov

Telo funkcie `draw` začína deklaráciami potrebných matíc a vektorov. Prvá je deklarovaná matica `overlay`, ktorej je nastavená rovnaká veľkosť a typ ako matici obrazu z kamery `mRgb`. Do tejto matice

sa budú vykresľovať virtuálne objekty a potom sa pomocou perspektívnej projekcie premietne do oblasti oktávy v obraze z kamery. Následne sú deklarované pomocné matice, ktoré budú vysvetlené nižšie a matica homografie `H`. Potom sú deklarované vektory bodov `Point2f octaveCorners` a `overlayCorners`. `octaveCorners` reprezentujú 4 body ohraničujúce oktávu v obraze z kamery. Tieto body sa získavajú z bodov rohov detegovaných markerov. `overlayCorners` sú body rohov matice `overlay`. Pomocou bodov týchto vektorov sa potom vypočíta homografia. Na konci deklaračného bloku je definovaná premenná `octaveNumber` určujúca číslo oktávy získané funkciou `getOctaveNumber`, ktorá prijíma parametre `id` a `keysCount`, kde `id` je najmenší identifikátor a `keysCount` počet kláves na klávesoch. Vo funkcii `getOctaveNumber` je jednoduchý konečný automat, ktorý podľa parametrov `id` a `keysCount` na klávesoch vráti správne číslo oktávy. Aby sa ušetrilo písanie ďalšieho konečného automatu konvertujúceho identifikátor na číslo oktávy, bola použitá polygonálna interpolácia dvojíc identifikátor a číslo oktávy na určenie vzorca (*5.1*):

$$\check{c}\acute{i}slo\ oktávy = \frac{identifikátor + 3}{2} \tag{5.1}$$

Pre klávesy s počtom kláves 88 je vzorec trochu pozmenený. Funkciu `getOctaveNumber` je možné vidieť v Kód 5.9.

```
int getOctaveNumber(int id, int keysCount) {
    switch(keysCount) {
        case 49:
        case 61:
        case 76:
            return (id + 3) / 2;
        case 88:
            return (id + 1) / 2;
        default: return (id + 3) / 2;
    }
}
```

Kód 5.9: Funkcia getOctaveNumber

Ďalej vo funkcii `draw` je naplnenie vektoru `overlayCorners` bodmi rohov matice `overlay` a po ňom nasleduje `for` cyklus (viď Kód 5.13), v ktorom sa iteruje cez identifikátory s tým, že počítadlo sa inkrementuje vždy o dva, teda sa vlastne iteruje akoby cez oktávy. Na začiatku cyklu je volaná funkcia `drawNoteNames`, ktorá prijíma parametre `&overlay` a `octaveNumber`, t.j. referenciu na maticu `overlay` a číslo oktávy. Táto funkcia je zobrazená v Kód 5.10. Na začiatku funkcie sú definované vlastnosti písma `fontFace`, `fontScale` a `thickness`, teda typ, veľkosť a hrúbka písma. Ďalej sú definované horizontálna a vertikálna osmina typu `float`. Horizontálna preto, že v oktáve je osem bielych kláves a takto sa dá medzi nimi navigovať a vertikálna preto, že tiež je pomocou nej možné správne umiestniť mená tónov a čísla oktáv. Potom je definovaný bod `notePosition` typu `Point2f` určujúci pozíciu textu s menom tónu a číslom oktávy. Začiatočná pozícia je na klávese najviac vľavo, teda tóne C. Posledný je definovaný reťazec `notes` s menami tónov jednej oktávy, začínajúcej tónom C. Na konci funkcie je cyklus `for`, v ktorom sa iteruje cez mená tónov a pre každý tón sa pomocou funkcie `putText` z knižnice OpenCV vypíše jeho názov a číslo oktávy na príslušný kláves. Pre získanie farby textu je použitá funkcia `getColor`, ktorá prijíma ako parameter hodnotu enumerátoru `Color` a vracia premennú typu `Scalar` z knižnice OpenCV reprezentujúcu farbu v RGB. Funkcia `getColor` obsahuje statické pole farieb typu `Scalar`. Bod `notePosition` sa v každej iterácii posunie o horizontálnu osminu, aby sa názov nasledujúceho tónu a číslo oktávy zobrazili na správny kláves.

```
void drawNoteNames(Mat &overlay, int octaveNumber) {
    int fontFace = FONT_HERSHEY_SIMPLEX;
    float fontScale = 2.0;
    int thickness = 3;
    float horizontalEighth = overlay.cols / 8;
    float verticalEighth = overlay.rows / 8;
    Point2f notePosition = Point2f((horizontalEighth / 8), (overlay.rows -
        verticalEighth));
    string notes("CDEFGAHC");

    for(auto c : notes) {
        putText(overlay, c + intToString(octaveNumber), notePosition, fontFace,
            fontScale, getColor(COLOR_TEXT), thickness);
        notePosition.x += horizontalEighth;
    }
}
```

Kód 5.10: Funkcia drawNoteNames

Po zavolaní funkcie `drawNoteNames` sa inkrementuje číslo oktávy a potom sa volá funkcia `drawChords`. Táto funkcia prijíma parametre `&overlay` a `&wholeScreen`, t.j. referenciu na maticu `overlay` a referenciu na maticu obrazu z kamery `mRgb`. Štruktúra deklarácií je veľmi podobná funkcii `drawNoteNames`, avšak je pridaná premenná `lineThickness` určujúca hrúbku úsečiek a premenná `linesPoints` typu `ChordLinesPoints`. `ChordLinesPoints` je štruktúra obsahujúca dve polia typu `Point2f`, jedno pre začiatočné body úsečiek akordu a jedno pre koncové body. Nasleduje cyklus `for` (viď Kód 5.11), v ktorom sa iteruje cez mená akordov pomocou počítadla.

```
for(unsigned int i = 0; i < chordNames.size(); ++i) {
    putText(wholeScreen, chordNames.substr(i, 1), namePosition, fontFace,
        fontScale, getColor(static_cast<Color>(i)), textThickness);
    namePosition.x += horizontalEighth;

    linesPoints = getChordLinePoints(static_cast<OctaveNote>(i), horizontalEighth,
        verticalEighth);
    for(unsigned int j = 0; j < 3; ++j) {
        line(overlay, linesPoints.lineStarts[j], linesPoints.lineEnds[j],
            getColor(static_cast<Color>(i)), lineThickness);
        if(j == 0) {
            Point2f point = Point2f((linesPoints.lineStarts[j].x +
                linesPoints.lineEnds[j].x) / 2, linesPoints.lineStarts[j].y);
            circle(overlay, point, 5, Scalar(255, 255, 255), -1);
        }
    }
}
```

Kód 5.11: For cyklus funkcie drawChords

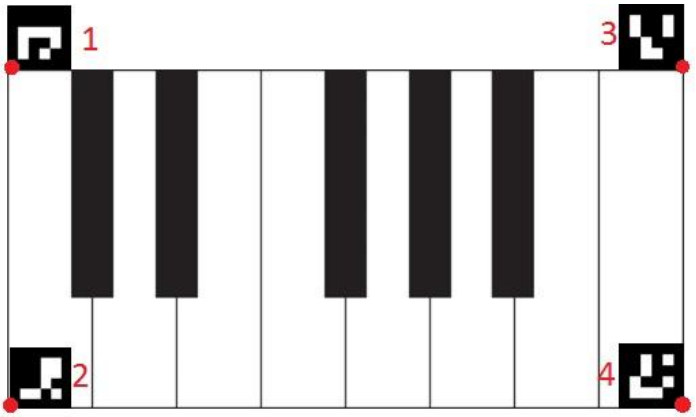
Na začiatku cyklu sa na hornú časť obrazu z kamery vypíše meno akordu v príslušnej farbe. Farba sa získa funkciou `getColor`, ktorej sa predá ako parameter počítadlo pretypované na enumerátor `Color`. Nasleduje posunutie bodu `namePosition` a získanie štruktúry `ChordLinesPoints` pre daný akord funkciou `getChordLinesPoints`. Funkcia `getChordLinesPoints` prijíma hodnotu enumerátoru `OctaveNote` určujúcu, pre ktorý akord sa vráti štruktúra `ChordLinesPoints` a horizontálnu

a vertikálnu osminu matice `overlay`, aby sa body štruktúry dali určiť. Obsahuje konečný automat, ktorý podľa predanej hodnoty enumerátoru `OctaveNote` vypočíta a vráti body úsečiek v štruktúre `ChordLinesPoints`. Príklad jedného stavu automatu je v Kód 5.12. Funkcia `getChordLinesPoints` obsahuje dve lokálne premenné `linesPoints` a `point`. `linesPoints` je štruktúra `ChordLinesPoints`, ktorá bude vrátená a `point` je bod typu `Point2f` reprezentujúci začiatočný alebo koncový bod úsečky. V každom stave automatu sa na začiatku určí vertikálna súradnica bodu `point` vynásobením vertikálnej osminy konštantou. Vertikálna súradnica je pre všetky body úsečiek akordu rovnaká. Konštanty pre násobenie vertikálnej osminy boli určené experimentálne.

```
case C:
    point.y = verticalEighth * 5.5;
    point.x = getXCoordOfNote(C, horizontalEighth);
    linesPoints.lineStarts[0] = point;
    point.x += horizontalEighth;
    linesPoints.lineEnds[0] = point;

    point.x = getXCoordOfNote(E, horizontalEighth);
    linesPoints.lineStarts[1] = point;
    point.x += horizontalEighth;
    linesPoints.lineEnds[1] = point;

    point.x = getXCoordOfNote(G, horizontalEighth);
    linesPoints.lineStarts[2] = point;
    point.x += horizontalEighth;
    linesPoints.lineEnds[2] = point;
    break;
```

Kód 5.12: Jeden stav konečného automatu funkcie getChordLinesPoints

Potom sa určia horizontálne súradnice pre každú úsečku tak, že začiatočná horizontálna súradnica tónu sa získa funkciou `getXCoordOfNote`. Táto funkcia prijíma ako parametre hodnotu enumerátoru `OctaveNote` a horizontálnu osminu a pomocou konečného automatu vráti začiatočnú horizontálnu súradnicu daného tónu/klávesu. Táto súradnica sa priradí bodu `point`, ktorý sa uloží do poľa začiatočných bodov úsečiek v štruktúre `linesPoints`. Potom sa k horizontálnej súradnici začiatočného bodu pripočíta horizontálna osmina, priradí sa bodu `point` a ten sa uloží do poľa koncových bodov úsečiek v štruktúre `linesPoints`. Tento proces sa vykoná pre každý tón akordu, pričom prvý bod v poli začiatočných aj koncových bodov patrí vždy úsečke základného tónu akordu.



Obrázok 5.3: Body ukladané do vektoru octaveCorners

Po volaní funkcie `drawChords` sa naplní vektor `octaveCorners` bodmi z rohov príslušných markerov ohraničujúcich oktávu. Pre lepšie pochopenie, ktoré body sa ukladajú do `octaveCorners`, viď Obrázok 5.3.

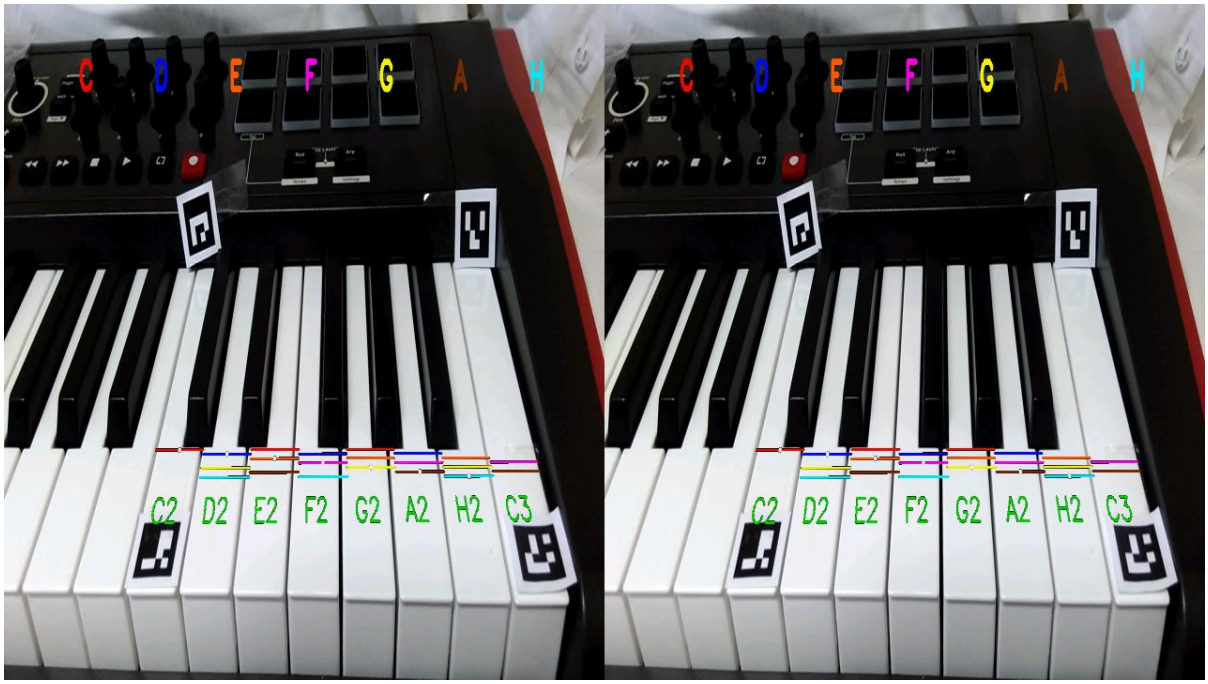
```
for(unsigned int i = 0; i < (sortedIds.size() - 4); i += 2)
{
    drawNoteNames(overlay, octaveNumber);
    ++octaveNumber;
    drawChords(overlay, mRgb);

    octaveCorners.push_back(markerCorners[sortedIds[i]][BOTTOM_LEFT]);
    octaveCorners.push_back(markerCorners[sortedIds[i+1]][BOTTOM_LEFT]);
    octaveCorners.push_back(markerCorners[sortedIds[i+2]][BOTTOM_RIGHT]);
    octaveCorners.push_back(markerCorners[sortedIds[i+3]][BOTTOM_RIGHT]);

    H = findHomography(overlayCorners, octaveCorners, RHO);
    octaveCorners.clear();

    if(H.empty()) {
        __android_log_print(ANDROID_LOG_VERBOSE, APPNAME, "ERR: findHomography
            returned empty matrix");
        return;
    }
    warpPerspective(overlay, overlayWarped, H, mRgb.size());
    cvtColor(overlayWarped, mask, CV_BGR2GRAY);
    threshold(mask, mask, 0, 255, CV_THRESH_BINARY);
    bitwise_not(mask, maskInv);
    mRgb.copyTo(result1, maskInv);
    overlayWarped.copyTo(result2, mask);
    mRgb = result1 + result2;
}
```

Kód 5.13: For cyklus funkcie draw

5.7 Perspektívna projekcia a spojenie virtuálneho obrazu s reálnym

Do matice `H` sa uloží matica homografie vypočítaná funkciou `findHomography` zo vzťahov bodov v `overlayCorners` a `octaveCorners`. Posledný parameter funkcie `findHomography` `RHO` znamená, že sa používa algoritmus PROSAC. Obsah vektoru `octaveCorners` sa vymaže, pretože v ďalšej iterácii sa vykresľuje do ďalšej oktávy, ak bolo detegovaných dosť markerov. Potom sa kontroluje, či matica `H` nie je prázdna a po tejto kontrole začína aplikovanie perspektívnej projekcie a spájanie obrazu s virtuálnymi objektmi s obrazom z kamery. Perspektívna projekcia sa aplikuje pomocou funkcie `warpPerspective` z knižnice OpenCV. Táto funkcia podľa matice homografie `H` perspektívne premietne obraz z matice `overlay` do matice `overlayWarped`, ktorá má veľkosť matice `mRgb`, teda obrazu z kamery. V tomto momente sú v matici `overlayWarped` „sploštené“ virtuálne objekty na čiernom pozadí a je potrebné ich dostať do obrazu z kamery, t.j. do matice `mRgb`. Spojenie prebieha tak, že sa z matice `overlayWarped` vytvorí čiernobiely obraz, na ktorý sa aplikuje binárne

prahovanie a tak vzniká matica `mask` - maska matice `overlayWarped`. V tejto maske sú všetky virtuálne objekty úplne biele a pozadie úplne čierne. Ku tejto maske sa urobí inverzná maska `maskInv` pomocou bitovej negácie. Následne sa matica `mRgb` maskovaná maticou `maskInv` nakopíruje do pomocnej matice `result1` a matica `overlayWarped` maskovaná maticou `mask` sa nakopíruje do pomocnej matice `result2`. Výsledný obraz (viď Obrázok 5.4) sa získa sčítaním matíc `result1` a `result2`.



Obrázok 5.4: Výsledný obraz s virtuálnymi objektmi

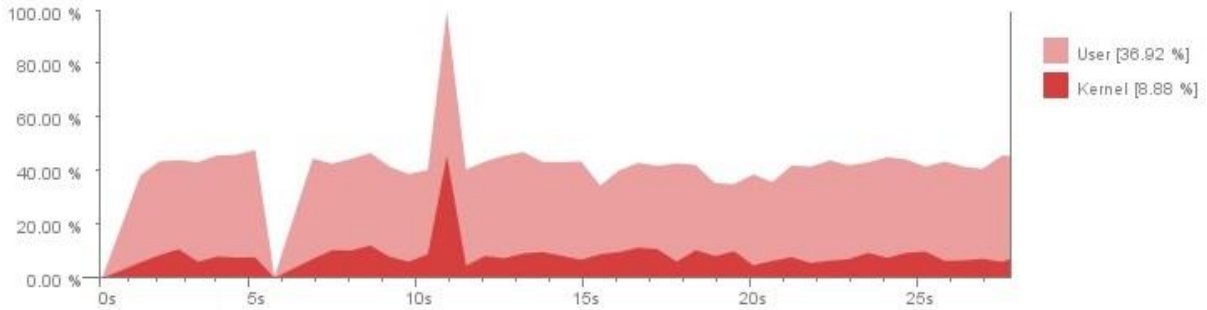
6 Výsledky a testovanie

V tejto kapitole sú prezentované a komentované výsledky testovania aplikácie. Aplikácia bola testovaná na smartfóne Xiaomi Redmi 2 s operačným systémom Android verzie 4.4 KitKat. Parametre zariadenie sú nasledujúce:

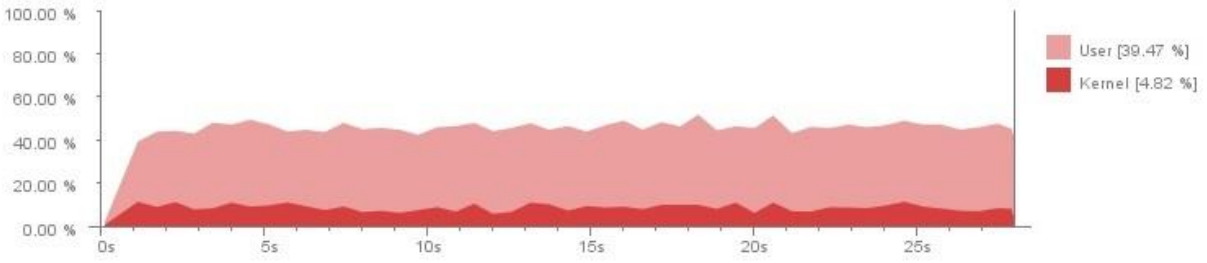
- **CPU:** Qualcomm Snapdragon 410 (MSM8916) - Quad Core 1.2GHz (64 bit)
- **Grafické jadro:** Adreno 306
- **RAM:** 2 GB (LPDDR3)
- **Veľkosť displeja:** 4.7"
- **Rozlíšenie displeja:** 1280x720 HD
- **Počet farieb:** 16 miliónov
- **Fotoaparát:** 8 MP, f/2.2, 28mm, automatické zaostrovanie
- **Video:** 1080p, 30fps

6.1 Hardvérové nároky aplikácie

Boli odmerané procesorové a pamäťové nároky bežiacej aplikácie pri snímaní okolia kamerou bez detegovania markerov vykresľovania virtuálnych objektov a potom aj s detegovaním a vykresľovaním. Pre tieto merania bolo zvolené rozlíšenie 800x480 pixelov, ako optimálny pomer medzi viditeľnosťou a počtom snímok za sekundu a štandardné rozlíšenie pre aplikáciu *CardboardKeyboard*.
Na merania bol použitý Android Monitor z IDE Android Studio.



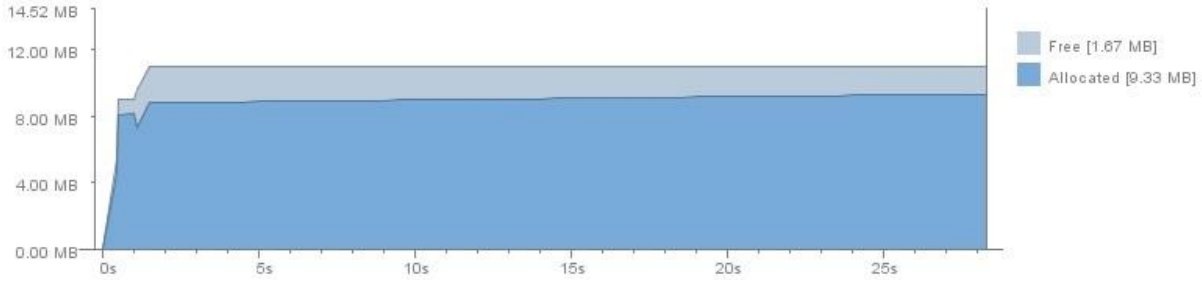
Obrázok 6.1: Využitie procesora bez detekcie a vykresľovania



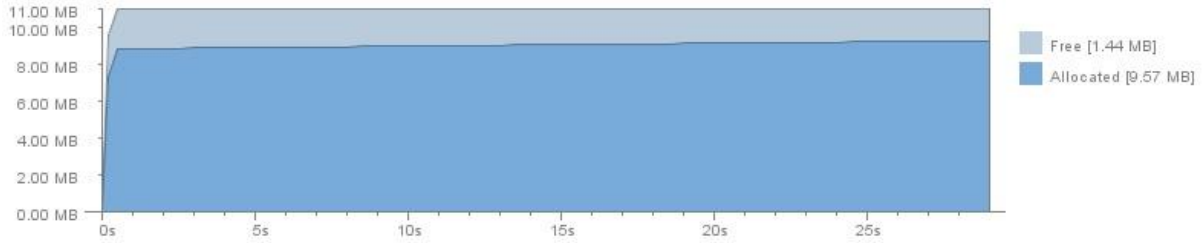
Obrázok 6.2: Využitie procesora s detekciou a vykresľovaním

Využitie procesora je zobrazené pre User a Kernel mód a zahŕňa všetky jadrá procesoru. V User móde musí kód používať systémové API, aby mohol pristupovať ku pamäti a hardvéru, a má prístup len ku adresám do virtuálnej pamäti. Vďaka tomu je možné zotavenie systému po páde aplikácie. User mód je relevantný pre meranie, keďže v ňom beží aplikácia *CardboardKeyboard*. V Obrázok 6.1 vidíme, že

využitie procesora pri snímaní okolia kamerou bez detekcie markerov a vykresľovania virtuálnych objektov sa pohybuje s miernymi výchylkami okolo 40-45%. Dôvod, prečo nie je procesor využitý napr. na 90% je, že na testovacom zariadení sa nachádza štvorjadrový procesor a aplikácia CardboardKeyboard nie je implementovaná ako viacvláknová. 45% predstavuje použitie dvoch jadier, teda sa používa viac jadier procesora naraz, ale toto je zariadené samotným procesorom, ktorý vie do istej miery rozdeliť jednovláknový kód na viacvláknový. Pri aktívnej detekcii a vykresľovaní (viď Obrázok 6.2) nie je vidieť takmer žiadny nárast využitia. Toto je pravdepodobne spôsobené tým, že už samotné snímanie je pre procesor náročné, a keď sa pridá detekcia a vykresľovanie, nestúpne využitie procesora, ale klesne počet snímok za sekundu.



Obrázok 6.3: Využitie pamäti bez detekcie a vykresľovania



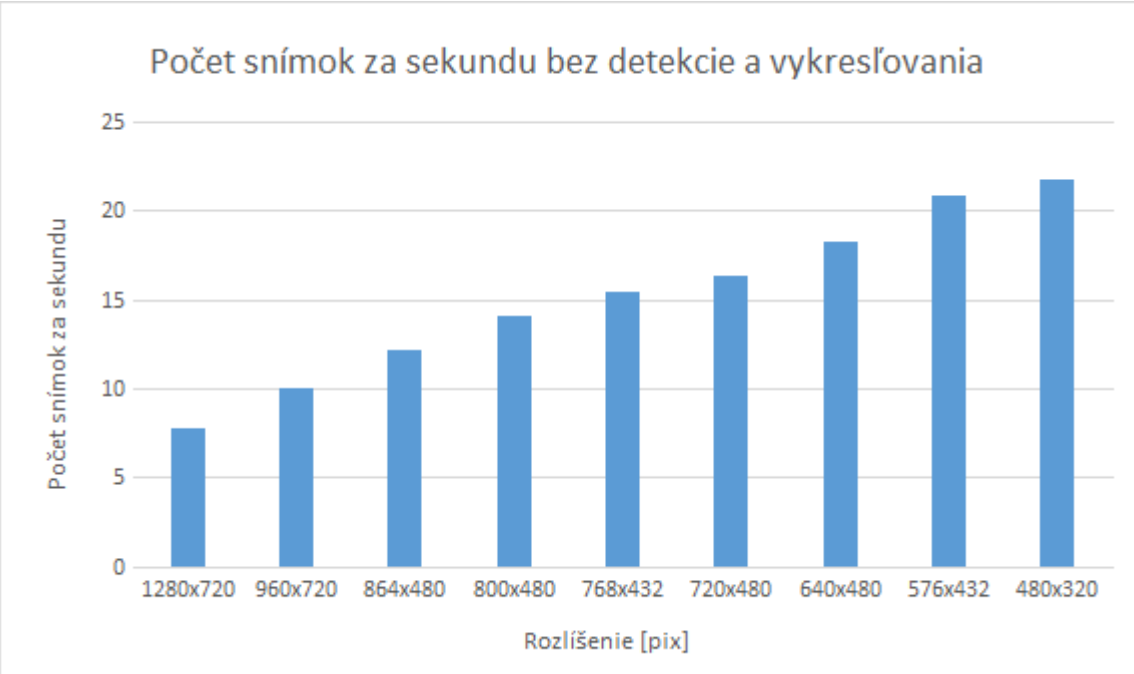
Obrázok 6.4: Využitie pamäti s detekciou a vykresľovaním

Pri využití pamäti je situácia podobná ako pri procesorovom využití. Na Obrázok 6.3 vidno, že aplikácia pri snímaní okolia bez detekcie a vykresľovania má k dispozícii cca 11MB pamäti, pričom používa okolo 9MB. Keď je aktívna detekcia a vykresľovanie (viď Obrázok 6.4), používa sa v podstate rovnaké, resp. mierne zvýšené množstvo pamäti. Veľmi podobné využitie pamäti pravdepodobne vyplýva z toho, že väčšina pamäti sa používa na dočasné uskladnenie obrazových dát z kamery a detekcia a vykresľovanie len manipulujú s týmito alokovanými dátami.

6.2 Počet snímok za sekundu

Ďalšia testovaná veličina bol počet snímok zobrazených na displeji za jednu sekundu. Táto veličina je veľmi dôležitá, pretože malý počet snímok za sekundu pri aplikáciách rozšírenej reality môže spôsobiť nevoľnosť a bolesť hlavy. Ideálny počet snímok za sekundu sa určuje ťažko, napr. pri klasických PC hrách sa odporúča počet snímok za sekundu aspoň 30 a moderné headsety pre virtuálnu realitu dosahujú 90 snímok za sekundu. 90 snímok za sekundu je samozrejme na smartfóne absolútne nereálne číslo pri aplikácii, ktorá spracováva obraz v reálnom čase, ale počet snímok okolo 30 by mohol byť dostačujúci. Testovanie prebiehalo tak, že aplikácia bola vždy zostavená s iným rozlíšením a po zostavení a spustení sa nejaký čas snímalo len okolie a potom klávesy s markermi. Je teda možné sledovať rozdiel v počte

snímok za sekundu bez detekcie a vykresľovania (viď Obrázok 6.5) a v počte snímok s detekciou a vykresľovaním (viď Obrázok 6.6).



Obrázok 6.5: Graf počtu snímok za sekundu pre rôzne rozlíšenia bez detekcie a vykresľovania

Obrázok 6.6: Graf počtu snímok za sekundu pre rôzne rozlíšenia s detekciou a vykresľovaním

Z grafov je jasne vidieť, že hodnoty počtu snímok za sekundu sú hlboko pod vhodnými hodnotami. Pre štandardné rozlíšenie testovaného zariadenia (1280x720) je počet snímok bez detekcie a vykresľovania blízky hodnote 7, čo je nedostatočné aj pre klasické nahrávanie videa. Pri aplikovaní detekcie a vykresľovania objektov hodnoty pochopiteľne klesnú, pretože je potrebné urobiť viac výpočtov, ale tento rozdiel nie je až taký markantný. Je teda potrebné zamyslieť sa nad optimalizáciou získavania dát

z kamery telefónu, napr. použiť novšie API pre kameru android.hardware.camera2. Toto API ale zatiaľ nie je podporované knižnicou OpenCV, takže treba buď implementovať vlastný spôsob prenášania dát z nového API do OpenCV alebo počkať, kým ho začne OpenCV podporovať.

7 Záver

Cieľom tejto bakalárskej práce bolo vytvoriť aplikáciu pre smartfón so systémom Android, ktorá pomocou rozšírenej reality a cenovo dostupného headsetu priblíži užívateľovi základné akordy a stupnice na piáne resp. klávesoch. Bolo potrebné zoznámiť sa s problematikou rozšírenej reality, preštudovať princíp homografie a tiež spôsob vývoja pre operačný systém Android.

Výsledkom práce je aplikácia *CardboardKeyboard* využívajúca ArUco markery, knižnicu OpenCV a headset Google Cardboard pre poskytnutie modernej formy výuky na klávesy. Na klávesy je potrebné umiestniť markery spôsobom popísaným v práci. Aplikácia hneď po spustení začína snímať okolie a keď užívateľ namieri kameru smartfónu na klávesy s rozmiestnenými markermi, na displej sa vykreslia mená tónov a akordy v stupnici C dur.

Práca je jedinečná svojím konceptom a taktiež kombináciou použitých technológií a má veľký potenciál pre budúce rozširovanie. Rozšírenia by určite mali zahŕňať pridanie viacerých akordov a stupníc, inteligentné riešenie prechodu medzi oktávami, riešenie vykresľovania v prípade, že užívateľ rukou zakryje marker a mohli by sa pridať aj nastavenia, aby užívateľ sám mohol zadávať počet kláves a požadované rozlíšenie, prípadne iné prispôsobenia. Taktiež, ako vyplynulo z testovania aplikácie, by bolo potrebné vyriešiť nízke hodnoty počtu snímok za sekundu vhodnou optimalizáciou alebo zmenou API pre získavanie dát z kamery.

Literatura

[1] Merriam-Webster [online]. ©1996-2015 [cit. 2015-12-25]. Dostupné z: http://www.merriam-webster.com/

[2] AZUMA, R. T. A survey of augmented reality. *Presence: Teleoperators* [online]. 1997, **6**(4): 355-385 [cit. 2015-12-26]. ISSN 10547460. Dostupné z: http://www.cs.unc.edu/~azuma/ARpresence.pdf

[3] DUBROFSKY, E. Homography estimation. Vancouver, 2009 [cit. 2016-04-17]. PhD Thesis. University Of British Columbia, The Faculty of Graduate Studies. Dostupné z: https://www.cs.ubc.ca/grads/resources/thesis/May09/Dubrofsky_Elan.pdf

[4] CHUM, O., J. MATAS. Matching with PROSAC-progressive sample consensus. Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on. IEEE, 2005. p. 220-226 [cit. 2016-04-23]. Dostupné z: http://cmp.felk.cvut.cz/~matas/papers/chum-prosac-cvpr05.pdf

[5] GARRIDO-JURADO, S., R. MUÑOZ-SALINAS, F.J. MADRID-CUEVAS a M.J. MARÍN-JIMÉNEZ. Automatic generation and detection of highly reliable fiducial markers under occlusion. Pattern Recognition [online]. 2014, 47(6), 2280-2292 [cit. 2016-04-24]. DOI: 10.1016/j.patcog.2014.01.005. ISSN 00313203. Dostupné z: http://linkinghub.elsevier.com/retrieve/pii/S0031320314000235

[6] DOUGLAS, D.H., T.K. PEUCKER. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Cartographica: The International Journal for Geographic Information and Geovisualization*. 1973, 112-122.

[7] *Android NDK* [online]. Mountain View (California): Google, 2016 [cit. 2016-05-05]. Dostupné z: http://developer.android.com/ndk/index.html