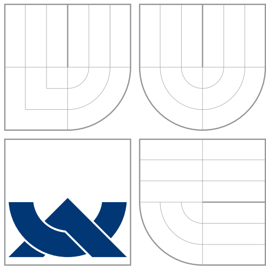
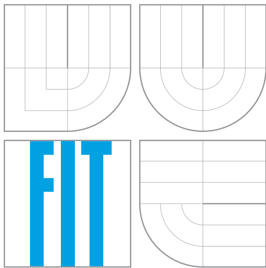


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

UNIFORM MARKER FIELDS PRO ANDROID

UNIFORM MARKER FIELDS FOR ANDROID

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — MAREK SALÁT
AUTHOR

VEDOUCÍ PRÁCE — Doc. Ing. ADAM HEROUT, Ph.D.
SUPERVISOR

BRNO 2013

Abstrakt
Tato práce se zabývá návrhem a implementací vlastního detektoru *Uniform Marker Fields* a ladící aplikace pro mobilní zařízení. Implementace detektoru přináší alternativní postupy k původnímu řešení, tak aby co nejvíce vyhovovala málo výkonným zařízením. Hlavním cílem této práce je nalézt pozici a rotaci modulu markeru, případně pozici kamery v prostoru. Výsledné řešení je možné využít jako knihovnu pro detekci modulu markeru nebo základ pro aplikace využívající rozšířenou realitu s použitím *Uniform Marker Fields*.

Abstract
This thesis deals with a desing and an implementation of a detector for *Uniform Marker Fields* and a debugging appliaction for Android mobile devices. Implementation comes with alternativ methods in considiration of original solution, suitable for low-performance devices. The main goal of detector is find a position and a rotation of marker module or alternatively a position of the camera in real space. The resulting soulution can be also used as a library for the marker module detection or as a core for augmented reality based application using *Uniform Marker Fields*.

Klíčová slova
Uniform Marker Fields, detekce, marker, Android, rozšířená realita, mobilní zařízení, kalibrace kamery

Keywords
Uniform Marker Fields, detection, marker, Android, augmented reality, mobile devices, camera calibration

Citace
Marek Salát: Uniform marker fields pro Android, bakalářská práce, Brno, FIT VUT v Brně, 2013

Uniform marker fields pro Android

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Adama Herouta. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

....
Marek Salát
12. května 2013

Poděkování

Chtěl bych poděkovat panu Adamu Heroutovi za zajímavé podněty a rady. Také bych chtěl poděkovat panu Istvánu Szentandrásimu za ochotu s vstřícné jednání.

© Marek Salát, 2013.

Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

1 Úvod **2**

2 Rozsířená realita **3**
- 2.1 Využití rozšířené reality . . . 4
- 2.2 S využitím 2D markerů . . . 4

3 Uniform Marker Fields **6**
- 3.1 Orientované okno . . . 7
- 3.2 Detekce Uniform Marker Fields . . . 8

4 Návrh ladící aplikace, detektoru a způsobu detekce **13**
- 4.1 Popis funkcí ladící aplikace . . . 13
- 4.2 Rozhraní detektoru . . . 14
- 4.3 Model ladící aplikace . . . 14
- 4.4 Nalezení edgelů – adaptivní prahování . . . 15
- 4.5 Upřesnení odhadu směru přímek . . . 16
- 4.6 Rozdělení shluků přímek do skupin . . . 17
- 4.7 Zpřesnění výpočtu dvou úběžníků – filtrace přímek . . . 18
- 4.8 Způsob nalezení vějířů a reprezentace . . . 19
- 4.9 Lokalizace pozice uvnitř markeru . . . 20

5 Implementace a optimalizace aplikace **23**
- 5.1 Android . . . 23
- 5.2 Optimalizace rychlosti – Pool, inline funkce . . . 25
- 5.3 Reprezentace markeru . . . 25
- 5.4 Výpočet úběžníků eigen-dekompozicí . . . 26
- 5.5 Kalibrace kamery . . . 27
- 5.6 3D rekonstrukce . . . 27

6 Testování a vyhodnocení výsledků **29**

7 Závěr **31**

A Obrázky **33**

Kapitola 1

Úvod

Rozšířená realita (RR) je v poslední době velmi oblíbená. Prozatím nedisponuje velkým trhem a většina dnešních aplikací je spíše ve fázi prototypu než hotovým produktem. To dělá z rozšířené reality nevyužitou mezeru na trhu, které je možno využít.

Vysoké technologické nároky (na paměť nebo CPU) jsou jedním z hlavních důvodů nízkého počtu aplikací pro rozšířenou realitu v mobilních zařízeních, která nedisponují dostatečným výkonem. *Uniform makrer fields* (*UMF*) je navrhnut pro málo výkonná zařízení, kde je snaha zjistit pozici kamery v prostoru za použití co nejméně zdrojů. *UMF* má kromě rychlé detekce i jiné výhody mezi něž patří například překrytí markeru předměty, neboť k detekci postačí, aby byla viditelná pouze jeho část. S těmito vlastnostmi je možné mít marker velkých rozměrů (stůl, koberec atd.).

Hlavním cílem této práce bylo ověřit, zda jsou dnešní mobilní zařízení připravena na RR a zda je možné na takovýchto zařízeních detekovat pozici kamery v prostoru v reálném čase. K těmto účelům bylo potřeba vyvinout aplikaci pro detekci *UMF*, která by zohledňovala nízký výkon mobilních zařízení a zároveň umožňovala přijatelně kvalitní detekci. Možné řešení nabízí článek [7], který naznačuje, jak by mohl algoritmus detekce probíhat. Tato práce vychází z tohoto článku a přináší určité alternativní postupy vzhledem k původnímu řešení. Tato práce se snaží osvětlit všechny jednotlivé kroky detekce a s nimi související problémy.

Po dohodě s vedoucím jsem se rozhodl soustředit na stabilitu a přesnost detekce pozice markeru. Cílem mé práce je tedy extrahovat modul markeru (zjistit pozici modulu uvnitř markeru s jeho rotací). Pozice kamery byla ponechána jako případné rozšíření práce.

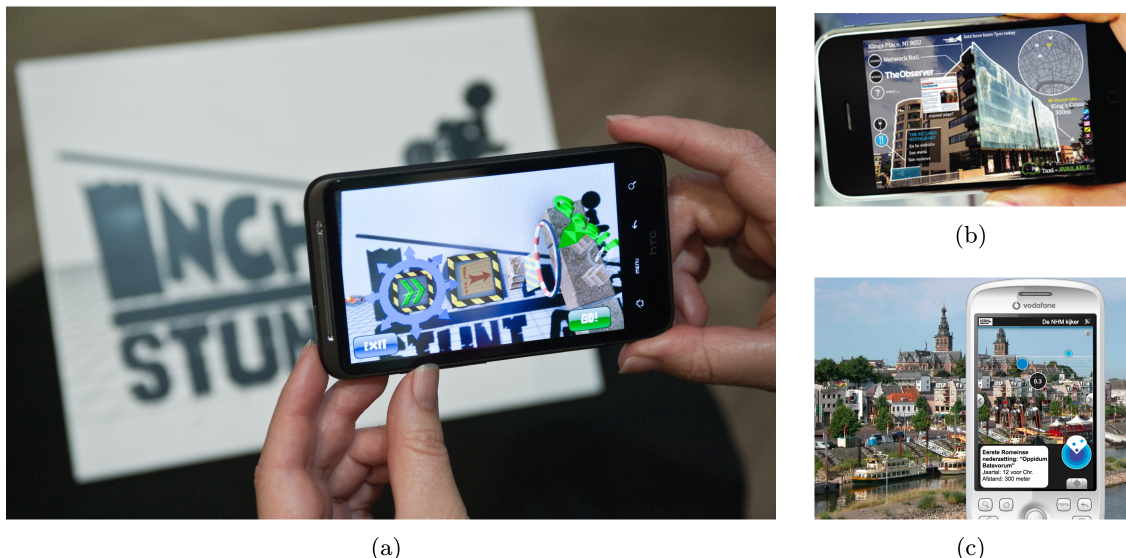
Dalším cílem bylo vyvinout ladící aplikaci, jež je schopná zobrazit všechny kroky detekce a data s nimi spojená. Aplikace je navržena tak, aby bylo možné, jednak měnit markery za běhu, jednak snadno přidávat novou funkcionalitu (různé typy markerů, modulů, klíčů a barev). Je rovněž podstatné, aby detektor nebyl závislý na cílové platformě, i když je primárně vyvinut pro Android. Aplikace poskytuje podporu pro ladění a testování celého systému.

Práce může posloužit jako jistý návod pro ty, co se rozhodnou využívat ve svých aplikacích *UMF*. Čtenář bude obeznámen s problematikou detekce *UMF* a možnými řešeními. Budou zde diskutována také alternativní řešení či doplnění, která nejsou implementována, avšak mohla by být zařazena do příští verze detektoru.

Kapitola 2

Rozsířená realita

Rozšířenou realitu je možné chápat jako přímý nebo nepřímý pohled na fyzický (reálný) svět, v němž jsou za pomoci vstupních senzorů (zvuk, video, GPS, gyroskop nebo kompas) přidány a zobrazeny objekty. Rozšířená realita na rozdíl od virtuální reality nahrazuje nebo přidává prvky do reálného světa namísto jeho vytváření a simulace [9]. Výsledkem je nový pohled na svět a vnímání reality. Tohoto je třeba dosáhnout v reálném čase, aby měl uživatel pocit, že to co vidí, může být skutečné. Prostřednictvím pokročilých technologií pro rozšířenou realitu (počítačové vidění, rozpoznávání objektů) se informace získané o okolním světě uživatele mohou stát interaktivní a digitálně manipulovatelné. Na obrázku 2.1 jsou ukázky aplikací využívající rozšířenou realitu.



(a) (b) (c)

Obrázek 2.1: Ukázky aplikací pro rozšířenou realitu. Na obrázku 2.1(a)[1]účastník soutěže *Qualcomm Augmented Reality Developer Challenge 2010*, hra *Inch High Stunt Guy* od Defiant Development Pty Ltd. umístěná na druhém místě s odměnou 50 000$. Ukázka způsobu navigace v budovách na obrázku 2.1(b)[2]. Navigace ve městech na obrázku 2.1(c)[3].

[1] http://multivu.prnewswire.com/mnr/qualcomm/48677/

[2] http://agbeat.com/wp-content/uploads/2009/12/layar.png

[3] http://static.guim.co.uk/sys-images/Guardian/Pix/pictures/2010/3/20/1269125712641/Augmented-reality-001.jpg

2.1 Využití rozšířené reality

Rozšířená realita má mnoho využití v mnoha různých aplikacích. Příklady jsou uvedeny v následujících odstavcích.

Zábava Rozšířená realita má mimo jiné uplatnění v herním průmyslu. Lze ji využít jako nový způsob ovládání a hraní her (použitím telefonu, tabletu nebo jiného speciálního zařízení). Existuje model [6, str. 5] vozidla připevněný na hydraulických pístech. Čelně před uživatelem se nachází velká obrazovka. Simulace jízdy je dosahováno nakláněním karoserie. Jedná se o kombinaci rozšířené a virtuální reality.

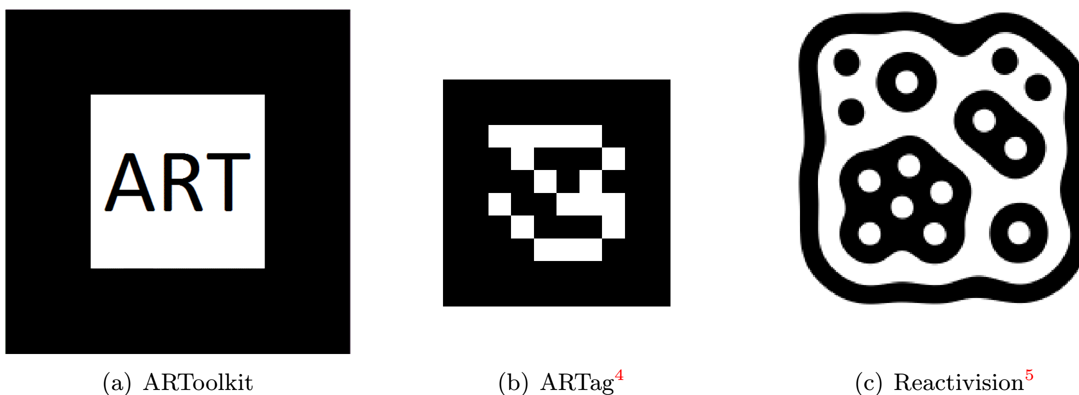
Reklama Dnes se hojně využívají QR kódy, které mohou obsahovat odkaz na firmu nebo produkt. Bylo by možné použít markery zobrazující uživateli 3D objekty (produkt, logo nebo jiný 3D objekt), jenž by byly pozorovatelné z libovolných úhlů.

Armáda Použití RR se speciálními brýlemi pomáhá některým armádám lépe vycvičit své vojáky a připravit je na různé situace. Armádní jednotky využívají tyto brýle pro rozšířenou realitu k zobrazení výšky nebo polohy [6, str. 5]. Pilotům stíhacích letadel je možné zobrazovat informace (zaměření, poloha, rotace) přímo do hledí přilby.

Navigace a turistika Uživatel si pomocí kamery může prohlížet panoráma, zatím co mu mohou být na obrazovku zobrazovány informace o významných bodech v krajině (památky, hory, jezera) nebo cestě, kterou by se mohl vydat.

2.2 S využitím 2D markerů

Aplikace pro rozšířenou realitu využívají různé druhy rovinných markerů ke zjištění pozice kamery ve scéně. Tyto markery jsou často dvoubarevné z důvodu snížení citlivosti osvětlení scény. Pro aplikace využívající rozšířenou realitu je nezbytné najít marker i ve velkých scénách (marker zabírá malou plochu scény). Na obrázku 2.2 jsou k vidění markery používané v různých aplikacích.



(a) ARToolkit (b) ARTag[4] (c) Reactivision[5]

Obrázek 2.2: Používané markery.

2.2.1 ARToolkit

ARToolkit [4] je dvoubarevný marker lemovaný širokým okrajem. Uvnitř markeru se nachází libovolný, od jiných markerů odlišný identifikátor. Detekce těchto markerů probíhá v následujících krocích [4, 3]:

- Celý vstupní obraz získaný z kamery je upraven prostřednictvím adaptivního prahování.
- Jsou nalezeny markery pomocí hrubých okrajů, kterými jsou obklopeny. Čtyři hrubé okraje určují potenciální marker a jejich rohy jsou použity k určení homografie, aby bylo možné zrekonstruovat marker zkreslený perspektivou.
- Dále se rozpozná identifikace ze zrekonstruovaného markeru. Střed je vzorkován $N \times N$ mřížkou (typicky 16×16 nebo 32×32). Vektor udávající barvy jednotlivých polí je porovnán s vektory v databázi pomocí korelace pro všechny směry (sever, jih, východ, západ).
- Čtyři body markeru jsou použity k definici homografie (pozice v prostoru).

Při použití těchto markerů na mobilních zařízeních dochází k problému s prahováním obrazu, protože je nutné zpracovat každý pixel. Tím je značně ovlivněna rychlost detekce. Rovněž při určitém osvětlení není možné marker ve scéně nalézt. Další nevýhodou je, že detekovaný vektor musí být porovnán s každým vektorem v databázi ve všech směrech. S vyšším počtem markerů v databázi roste čas jejich identifikace [3].

2.2.2 ARTag

Na rozdíl od ARToolkit, je ARTag [1] založen na vyhledávání markerů pomocí čar v obraze. Tímto postupem je odstraněna nevýhoda prahování obrazu. Přímky jsou propojeny do segmentů tvořících čtyřúhelníky. Detekce probíhá podobně jako u ARToolkit, rozdíl je v identifikaci markeru. Vnitřní část markeru se skládá z 6×6 dvou barevných čtverců tvořících posloupnost třiceti šesti bitů. V identifikaci je na deseti bitech zakódováno ID, zbylých 26 bitů slouží k detekci chyb a pro zachování jedinečnosti ve všech čtyřech orientacích. [3]

Detekce vychází z předpokladu, že se všechny čtyři hrany markeru protnou. Výhoda detekce pomocí hran spočívá v možnosti odhadu přibližné pozice hrany, pokud jedna z hran chybí nebo není úplná. Zakódování informace o rotaci přímo do markeru nevyžaduje porovnávání identifikačního vektoru s databází markerů,. Rotace a identifikace je zjištěna dekódováním tohoto vektoru.

[4] http://www.artag.net/

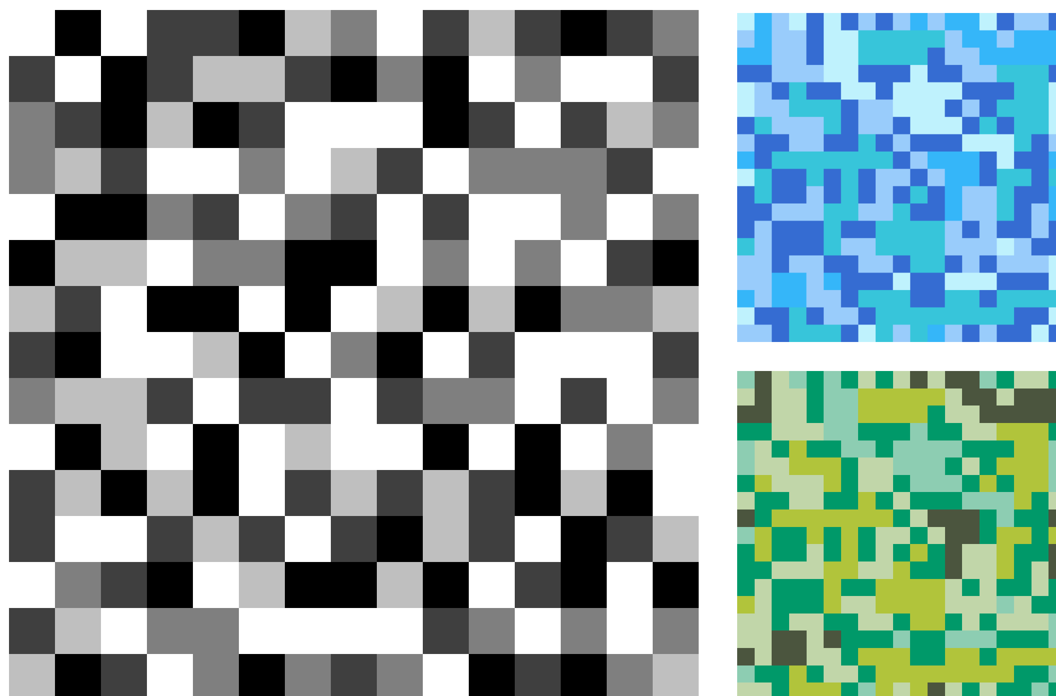
[5] http://reactivision.sourceforge.net/

Kapitola 3

Uniform Marker Fields

Některé aplikace vyžadují markery větších rozměrů, se kterými by bylo možné pokrýt větší plochu (stůl, projektor, podlahu) a současně vyžadují možnost pohybu nad celou plochou, přičemž v některých okamžicích může být viditelná pouze část markeru. Nicméně některé detektory (ARToolkit, ARTag) vyžadují, aby kamera zabírala celý marker (velkou část), tedy žádný okraj markeru nesmí být mimo zorné pole kamery (není možné se pohybovat ve scéně tak, že kamera směřuje na střed markeru a rohy markeru jsou mimo obraz). V takovém to případě se používá více malých, od sebe rozlišitelných markerů [7], které jsou nutné pro pohyb nad celou scénou.

Každý snímek z kamery musí obsahovat alespoň jeden marker dostatečně velký k poskytnutí potřebných informací, a zároveň dostatečně malý, aby se vešel do celého snímku. *Nested markers* [8] řeší tento problém rekurzivním vkládáním menších markerů do větších. Alternativou může být koncept *Uniform Marker Fields* na obrázku 3.1.



Obrázek 3.1: Markery 15×15 s velikostí okna 3×3.

Uniform Marker Fields je navrhnut pro detekci a lokalizaci pozice kamery za různých světelných podmínek (přímě osvětlení, stíny, různá intenzita světla) s velkými pozorovacími úhly. Jeho dalším úkolem je vypořádat se s překrytím markeru předměty a s rychlým pohybem způsobujícím rozmazání obrazu. Především jde o to detekovat marker, který je pozorován pouze z části [2]. Mít možnost se co nejvíce přiblížit markeru, stejně tak, jako pozorovat marker z větší vzdálenosti. Další důležitou vlastností je rychlá detekce s co nejmenším počtem navštívených pixelů.

UMF vychází z předpokladu, že se všechny hrany uvnitř markeru protnou ve dvou hlavních úběžnících scény. Mřížka tvořená čtverci markeru, může být detekována matematickými formalismy. Poloha uvnitř markeru je definována hranami mezi čtverci [2]. *Uniform* je myšleno v tom smyslu, že prvky použité pro detekci a lokalizaci pole a prvky pro identifikaci jednotlivých oken v poli, jsou rovnoměrně prolnuty napříč celou plochou *marker field* [7].

V původním článku byl použit marker s binárními (dvoubarevnými) okny. V mé práci je použit marker více barevný. V barevném markeru je možné najít větší počet čar a zároveň může být použito menší okno. Velikost okna určuje nejmenší možnou viditelnou část markeru v případě, že pro pro zjištění polohy není použit rozhodovací strom, v němž je možné detekovat pozici markeru porovnáním malého počtu sousedních čtverců (hran).

3.1 Orientované okno

Uniform Marker Fields je tvořen do sebe zapadajícími jedinečnými okny (moduly). Aby bylo možné zjistit rotaci markeru (sever, jih, východ, západ), musí být každé okno jedinečné ve všech směrech. Tedy pro libovolné okno je možné zjistit na jaké je pozici a jakou má rotaci. Pro k barev a velikost okna n je maximální počet oken (uvedeno v [7])

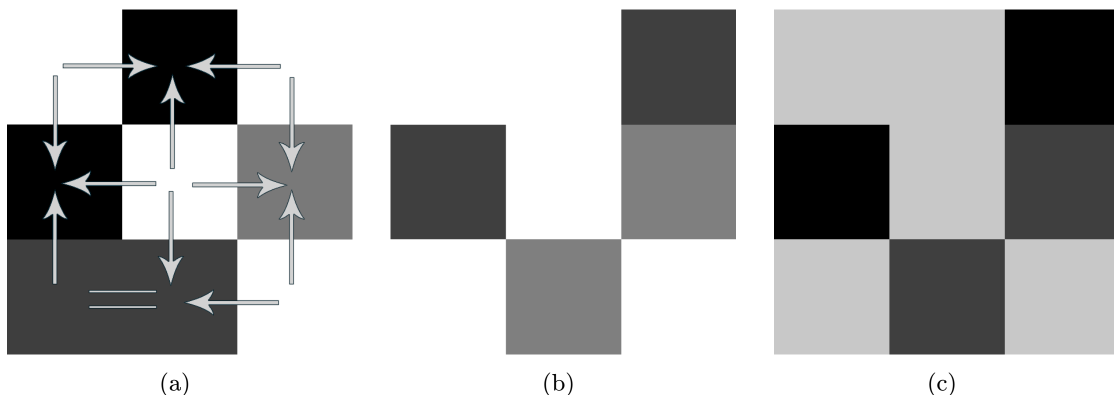
$$N \leq \frac{k^{n^2} - k \left\lfloor \frac{n^2+1}{2} \right\rfloor}{4}, \tag{3.1}$$

tedy maximální velikost markeru může být vypočtena jako

$$(h-n+1)(w-n+1) \leq \frac{k^{n^2} - k \left\lfloor \frac{n^2+1}{2} \right\rfloor}{4}, \tag{3.2}$$

kde h je výška a w šířka. k udává počet barev. Marker s pěti barvami by mohl mít $k = 5$, ale vzhledem ke způsobu detekce je $k = 3$, protože jsou zde pouze tři možnosti porovnání $(<, >, =$, porovnání bílé a šedé má stejný výsledek jako porovnání šedé a černé). Pro čtvercový marker o velikosti okna $n = 3$ je maximální velikost 72×72, ovšem vygenerovat takovýto marker je časově velmi náročné.

Zpracovávaný snímek může být pořízen pod různým osvětlením. Je velmi složité od sebe rozlišit jednotlivé odstíny barev. Z tohoto důvodu se pro lokalizaci pozice uvnitř markeru porovnává směr gradientu hrany mezi čtverci (obrázek 3.2(a)) [2]. Výsledkem porovnání je $(<, >, =)$. Toto porovnání zvyšuje nároky na generování markeru, protože i když má marker pět barev ($k = 5$), může obsahovat okna, která mají stejný klíč. Má-li okno pouze světlé barvy (neobsahuje nejtmavší), může existovat stejné okno pouze s větším odstínem. Taková to okna budou při porovnávání gradientu totožná (obrázky 3.2(b) a 3.2(c)). Pokud se klíč oken vytváří tímto způsobem, vždy bude $k = 3$, což snižuje maximální velikost markeru.



Obrázek 3.2: 3.2(a) porovnání směru gradientu hrany. Okna 3.2(b) a 3.2(c) při porovnávání směru gradientu hrany jsou totožná. Je nutné, aby se taková to okna v markeru nevyskytovala.

3.2 Detekce Uniform Marker Fields

Požívané detektory markerů jako ARTag, ARToolkit (obrázek 3.1) nejdříve detekují marker (typicky v čenobílém obrazu nalezením tlusté hrany, která marker obklopuje) ve scéně, poté co jsou markery identifikovány, je možné najít pozici kamery. Algoritmus představený I. Szentandrási [7] nerozlišuje prvky pro lokalizaci a prvky pro identifikaci.

Detekce markeru je postupně vykonávána v těchto krocích:

1. **Získání edgelů** (tj. hrana nebo-li pixel na hraně; termín vypůjčen od Martin Hirzer [3]) – Edgel je popsán bodem v obraze a orientací (vektor, přímka, koncové body).

2. **Výpočet dvou dominantních úběžníků** mezi edgely. Oba úběžníky definují horizont (přímka procházející oběma úběžníky).

3. **Nalezení mřížky (grid) tvořící hrany markeru** jako dvě skupiny pravidelně se opakujících přímek procházejících úběžníkem. Tyto dvě skupiny budou dále uváděny jako vějíře.

4. **Zjištění pozice v markeru** nalezením orientovaného okna (modulu) v detekované mřížce.

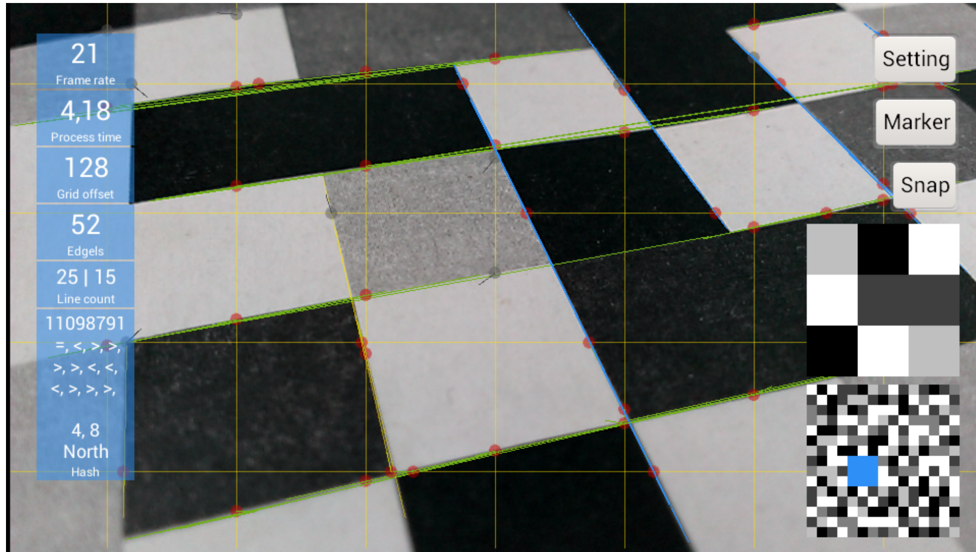
Kroky tohoto algoritmu jsou popsány v dalších odstavcích. Rovnice uvedené v této kapitole jsou převzaty z původního článku [7]. Implementace a optimalizace je uvedena v kapitole 5.

3.2.1 Získání hran – edgelů

Hledání přímek je možné provést pomocí řídce rozmístěných horizontálních a vertikálních čar tvořících mřížku (*scanlines*). Tento postup byl ukázán v článku Gridding Hough Transform [11]. Některé detektory markerů [1] hledají přímky na horizontálních a vertikálních čárách. Výhoda tohoto přístupu spočívá v tom, že při hledání přímek je zpracováno velmi malé množství pixelů.

K nalezení hrany $\mathbf{p}_0$ algoritmus používá adaptivní prahování založené na plovoucím průměru (podrobněji popsáno v kapitole 4.4) o velikosti okna k. Sobelův operátor aplikovaný

na každou detekovanou hranu (rovnice (4.1)), odhadne směr gradientu hrany $\mathbf{n}_0$. Normálový vektor směru gradientu $\mathbf{s}_0$ představuje hrubý odhad směru přímky.



Obrázek 3.3: *Žluté* – horizontální a vertikální přímky představují část obrazu, na níž jsou hledány hrany. *Červené* body – přínosné hrany (*šedé* jsou nepřínosné pro další zpracování). *Modré* a *zelené* (jsou zde roztříděny dalším zpracováním) přímky byly detekovány z počátečního červeného bodu. Krátké úsečky vycházející z edgelu představují původní odhad směru přímky.

Sklon přímky je upřesňován hledáním dalších hran ve směru odhadu. Nechť w je krok (podrobněji popsáno v části 4.5). Každý bod $\mathbf{p}_i$ je upřesňován přibližnými odhady bodu $\widetilde{\mathbf{p}}_i$:

$$\widetilde{\mathbf{p}}_{i+1} = \mathbf{p}_0 + iw\hat{\mathbf{s}}_i \, , \tag{3.3}$$

kde $\hat{\mathbf{s}}_i$ představuje normalizovaný vektor $\mathbf{s}_i$. Pokud je hrana nalezena, sklon přímky je aktualizován pomocí nově nalezeného bodu.

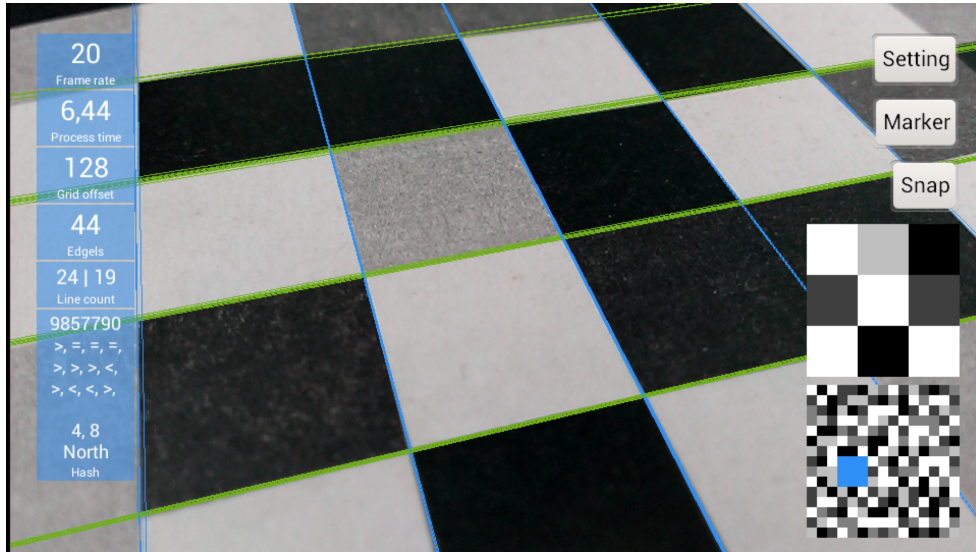
$$\mathbf{s}_{i+1} = \mathbf{p}_{i+1} - \mathbf{p}_0 \tag{3.4}$$

Stejný postup je aplikován na rostoucí $i \in \{0, 1, 2 \ldots\}$ a také klesající $i \in \{-1, -2 \ldots\}$.

Pouze přínosné edgely jsou uloženy pro další zpracování. Pokud bod $\widetilde{\mathbf{p}}_i$ neleží na téže přímce s určitou tolerancí nebo pokud není nalezen dostatečný počet upřesnění $\widetilde{\mathbf{p}}_i$, edgel je zahozen. Szentandrási uvádí [7], že přibližně sto upřesněných přímek dostačuje pro další zpracování. Na obrázku 3.3 je možno vidět výstup tohoto kroku algoritmu.

3.2.2 Určení dvou hlavních úběžníků

Jsou-li nalezeny edgely, algoritmus roztřídí přímky do dvou hlavních skupin podle sklonu. Tyto skupiny by měly odpovídat dvěma hlavním směrům hran markeru. V této práci je použito třídění přímek pomocí histogramu úhlů (podrobněji popsáno v kapitole 4.6). Ve více *zašuměných* datech je vhodné použít metody podobné RANSAC (použito v původní implementaci I. Szentandrási [7]). Dvě skupiny přímek jsou ukázány na obrázku 3.4.



Obrázek 3.4: *Zelené, modré* – dvě skupiny přímek rozšířené do nekonečna. Pro každou skupinu je vypočten použitím eigen-dekompozice úběžník. *Žluté* přímky jsou nevhodné pro další zpracování (neodpovídají úhlem nebo neprochází úběžníkem).

Použitím skupiny přímek může být úběžník vypočten velmi přesně. Za pomoci homogenních souřadnic pro úběžník $\mathbf{v}$ a skupinou přímek $\mathbf{l}_i = (a, b, c)$ tedy

$$\mathbf{l}_i \cdot (x, y, 1) = 0 \quad \Rightarrow \quad ax + by + c = 0.$$

Všechny přímky musí procházet úběžníkem. Musí platit

$$\forall i \colon \mathbf{v} \cdot \mathbf{l}_i = 0. \tag{3.5}$$

Samotný výpočet úběžníků se provádí nasazením nadroviny přes všechny pozorované přímky (propsáno v původním článku [7]). Normála nadroviny je vypočtena eigen-dekompozicí použitím korelační matice.

$$C = (\mathbf{l}_0, \dots \mathbf{l}_N)(\mathbf{l}_0, \dots \mathbf{l}_N)^T \tag{3.6}$$

Tato matice je 3×3 a symetrická, to znamená, že řešení může být nalezeno velmi přesně a efektivně. Výsledkem je vlastní vektor v homogenních souřadnicích udávající pozici úběžníku.

3.2.3 Určení dvou vějířů mřížky

Jsou-li nalezeny úběžníky $\mathbf{v}_1, \mathbf{v}_2$ pro každou skupinu edgelů, je horizont vypočten (tj. přímka procházející oběma úběžníky) jako $\mathbf{h} = \mathbf{v}_1 \times \mathbf{v}_1$. Libovolná přímka korespondující s hranou markeru může být vypočtena pomocí horizontu dle následující rovnice

$$b\mathbf{l}_i = \hat{\mathbf{l}}_{base} + (ki + q)\hat{\mathbf{h}}, \tag{3.7}$$

kde $\hat{\mathbf{l}}_{base}$ je libovolná (různá od horizontu) přímka procházející úběžníkem. Zvolil jsem přímku procházející středem obrazu. Při výpočtu $ki + q$ není známa délka vektoru přímky, proto je zde násobena b. Hodnota $ki + q$ se vypočítá pro každou přímku. Hodnoty jsou shlukovány. Každému shluku je přiřazeno i, které je proloženo přímkou lineární regresí [7]. Výstup tohoto kroku je na obrázku 3.4. Podrobněji rozebráno v kapitole 4.7.

3.2.4 Zjištění pozice uvnitř markeru

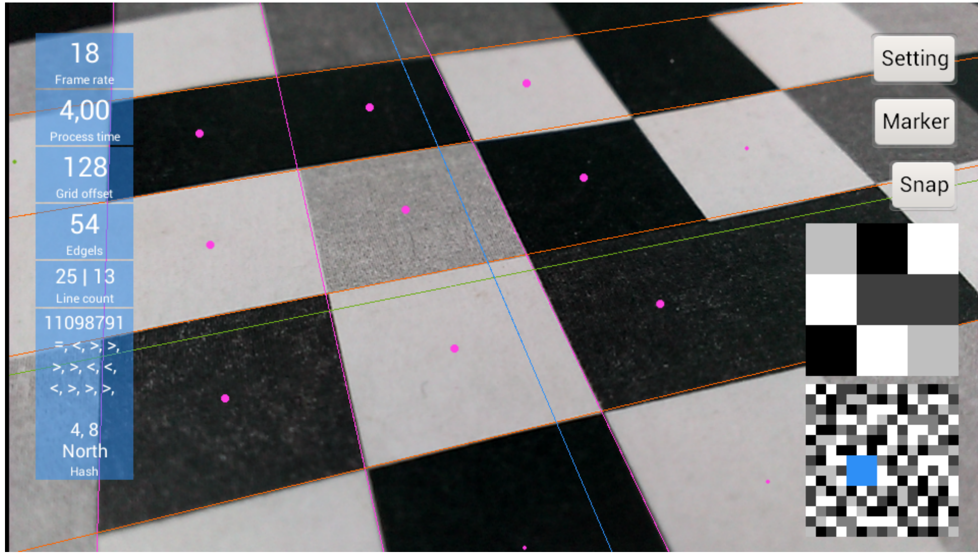
Jsou-li nalezeny přímky, hrany markeru mohou být nalezeny následujícím způsobem. Přímky obou vějířů se liší pouze v $\hat{\mathbf{l}}_{base}$ a každý vějíř má vlastní dvojici k a q.

$$\mathbf{l}_i^{(1)} = \hat{\mathbf{l}}_{base}^{(1)} + (k^{(1)}i + q^{(1)})\hat{\mathbf{h}} \tag{3.8}$$

$$\mathbf{l}_i^{(2)} = \hat{\mathbf{l}}_{base}^{(2)} + (k^{(2)}i + q^{(2)})\hat{\mathbf{h}} \tag{3.9}$$

Tudíž středy čtverců detekovaného markeru mohou být vypočteny jako

$$\mathbf{x}_{ij} = \mathbf{l}_{(i+1/2)}^{(1)} \times \mathbf{l}_{(j+1/2)}^{(2)}, \forall i, j \in \mathbb{N}. \tag{3.10}$$



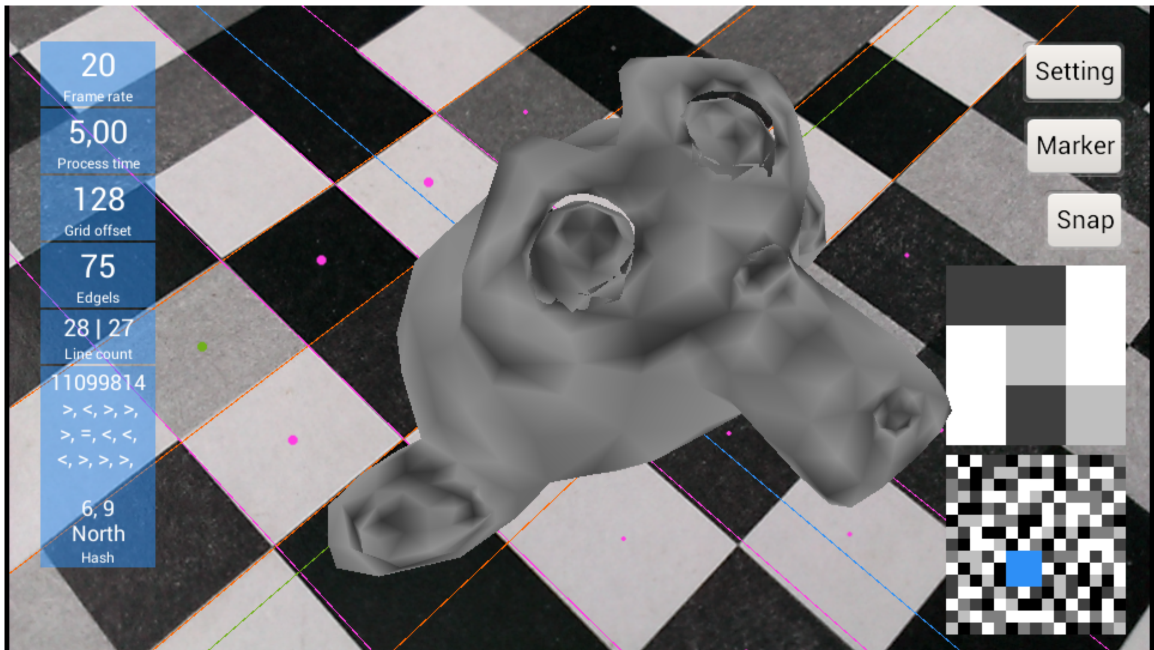
Obrázek 3.5: Vějíře a nalezení mřížky. Body představují $\mathbf{x}_{ij}$.

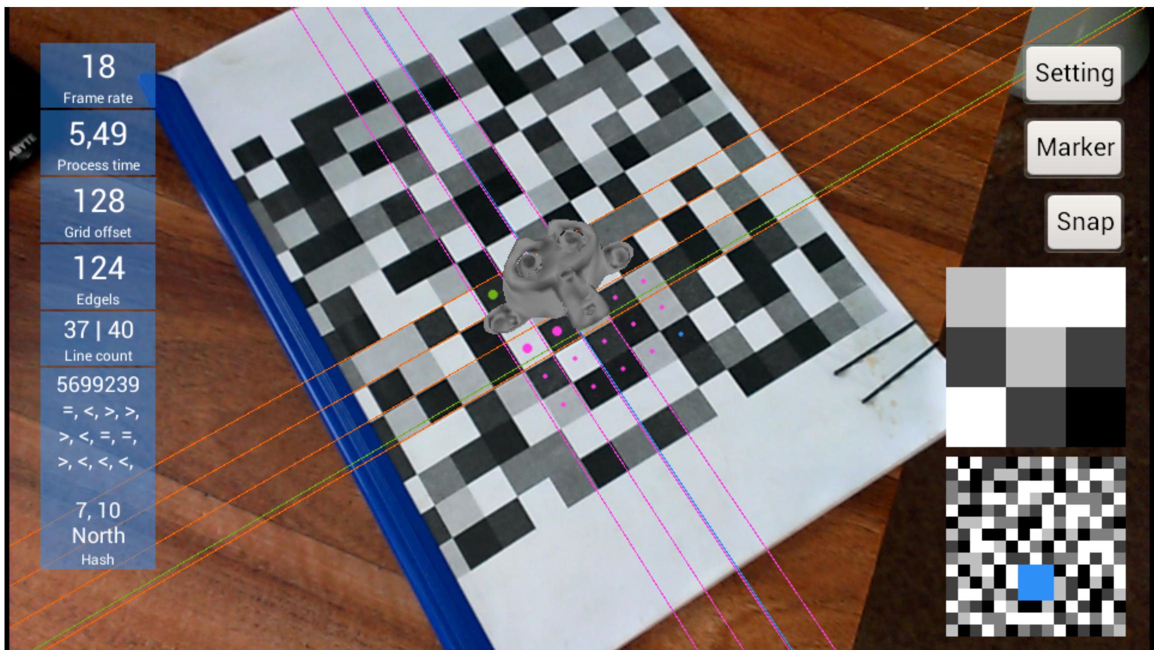
Následně je extrahována barva z každého středu a hledá se $n \times n$ okno uvnitř této mřížky. Extrahované body $\mathbf{x}_{ij}$ jsou zobrazeny na obrázku 3.5.

3.2.5 Pozice kamery v prostoru

Pozice kamery v prostoru je vypočtena korespondencí detekovaných středů markeru a modelových souřadnic (pro každý střed je známa pozice v markeru i se správnou rotací) a zjištěním homografie mezi body. Knihovny jako *OpenCV* nebo *FastCV* jsou schopny řešit tento problém, kdy ze středů a jejich modelových souřadnic vypočítají rotační matici a translační vektor. Kalibrace kamery je popsána v části 5.5.

Výsledek celého algoritmu je na obrázku 3.6.





Obrázek 3.6: Ukázka pozice kamery v prostoru.

Kapitola 4

Návrh ladící aplikace, detektoru a způsobu detekce

Článek popisující algoritmus detekce markeru [7, 2] je popsán velmi zevrubně. Jde spíše o návrh jak by mohla detekce probíhat, než o podrobný popis jednotlivých kroků. Má práce spočívala v návrhu a implementaci těchto detailů tak, aby co nejlépe vyhovovala mobilním zařízením. Jednotlivé kroky detekce jsou popsány v této kapitole, protože se nejedná o konkrétní implementační řešení, ale o postupy, které jsem navrhl a nejsou závislé na programovacím jazyce. V následujících odstavcích jsou podrobně popisovány všechny kroky detekce. Implementační detaily (např. způsob přístupu ke kameře, optimalizace rychlosti, uložení markeru) jsou popsány v kapitole 5. Dále je zde zaznamenán celý návrh aplikace, uživatelské rozhraní a také rozhraní knihovny pro detekci markeru.

V částech 4.2 a 4.3 je nastíněn význam nejdůležitějších prvků, jejich navrhovaná komunikace a závislosti. Diagramy tříd nebo rozhraní modulů je možné nalézt v příloze A.3.

4.1 Popis funkcí ladící aplikace

Uživatelské rozhraní je k vidění na obrázku 3.6. Následuje popis jednotlivých prvků a jejich funkcí.

Levý sloupec obsahuje informace získané z aktuálního snímku (počet snímku za sekundu[1], čas zpracování, rozestup přímek mřížky, počet nalezených edgelů, počet přímek ve skupinách, klíč okna, pozice uvnitř markeru).

Gesto *pinch to zoom* (*štípnutí*) pohybem dvou prstů po obrazovce, k sobě nebo od sebe, je možné měnit velikost offsetu mřížky.

Pravý sloupec dole zobrazuje aktuální detekovaný modul markeru a celý náhled markeru s vyznačeným modulem.

Setting je nabídka, která umožňuje nastavení zobrazení jednotlivých kroků detekce nebo zobrazení dalších ladících informací (mřížka, edgely, odhady směru, nalezené přímky, přímky

[1] Jedná se o počet snímku za sekundu, které dodává kamera, nikoli o teoretický odhad počtu možných zpracovávaných snímků.

procházející úběžníky, detekované vějíře, detekovaný grid, 3D objekty – Susanne, translační vektor, rotace $\mathbf{I}_{base}$).

Marker poskytuje výběr z dostupných typů markerů. Aplikace předpokládá, že je možné měnit markery za běhu aplikace. Markery mohou být různých rozměrů, barev, velikostí oken a dále se mohou lišit způsobem vytváření klíče.

Snap uloží aktuální snímek v `*.yuv` formátu a `png` aktuálního náhledu. Dlouhým stiskem tlačítka je možné nahrát video (pouze `yuv` formát), které může být použito v testech. Nahrávání je přerušeno opětovným dlouhým stiskem. Všechny informace jsou ukládány do adresáře `/sdcard/umf/snaps/`.

4.2 Rozhraní detektoru

Processor Detekce mřížky markeru je navrhnuta pro jazyk C, aby byl přeložitelný do nativního kódu procesoru. Výstupem procesoru jsou pouze středy markeru a jejich barvy. `Processor` neví nic o markeru (velikost, typ okna atd.), pouze zpracuje celý snímek. Je zde možné nastavit offset mřížky, fov_x kamery, vypočítat rotační matici a translační vektor. `Processor` obsahuje strukturu `PreviewFrame`, která drží všechny informace důležité pro detekci a hlavně vykonává jednotlivé kroky detekce. Jazyk C není objektový, z tohoto důvodu struktury obsahují ukazatele na funkce, aby bylo možné snadno měnit funkcionalitu jednotlivých kroků nezávisle na jiných.

Detektor využívá `Processor` pouze pro nalezení gridu, sám poté z gridu extrahuje modul markeru a zjišťuje jeho modelové souřadnice. Pokud jsou modelové souřadnice identifikovatelné, detektor požádá `Processor` o rotační matici a translační vektor. Rozhraní detektoru je navrhnuto pro jazyk *Java*, vzhledem k modelu aplikace (musí znát marker).

4.3 Model ladící aplikace

FourOrientableWindow – Modul markeru je nezávislý na markeru, způsobu vytváření nebo velikosti klíče. Obsahuje pouze pozici v markeru a klíče všech směrů (sever, jih, východ západ). Po zadání klíče vrátí okno rotaci pouze v případě, že klíč odpovídá jednomu ze směrů.

Hash představuje část modulu markeru. Je dán klíčem a rotací. Umí se porovnat s ostatními (nutné pro řazení ve stromu). Hash může být různých typů, velikostí. Každá implementace Hash je zodpovědná za uložení klíče, musí být implementována rotace vlevo, vpravo a porovnání.

HashFactory Protože každý hash může být vytvářen jiným způsobem (např. barevný hash je tvořen třemi hashy), je použit návrhový vzor továrna, který generuje instance hashů. Tato továrna umí kopírovat a klonovat hashe. Také je odpovědná za vytvoření hashe z barev (např. pro hash 3×3 obdrží uložiště `Storage` devíti barev a vytvoří instanci hashe).

Storage slouží jako uložiště barev. Pro jednu pozici čtverce je možné mít více barev (obrázek 4.6). Tato třída obstarává uložení a porovnání barev (např. střed obsahuje pouze jednu barvu – lze použít `StorageCenter`, pro střed obsahující pět barev – lze použít `StorageFive`). K porovnání konkrétních barev využívá komparátoru barev. Tím je dosaženo nezávislosti na barevném modelu.

Comparator porovnává barvy (např. porovnání pouze červené složky RGB modelu obstarává `ComparatorRed`).

Marker umí přidat modul a vyhledat jej podle zadaného klíče. Zná barvy jednotlivých čtverců. Marker se vytváří pomocí `Loader`, který jej inicializuje, jakmile je to žádané (metoda `load()`). Marker také disponuje `HashFactory`. Klíč markeru se vytváří vždy pomocí této továrny. Marker je inicializován na vyžádání přes `Loader`, protože se uvažuje několik (předem neznámých) způsobů načítání markeru. Každý marker obsahuje `HashFactory`, která je zodpovědná za vytvoření hashe. Tímto způsobem je zajištěna variabilita systému. Jednoduchý marker zajišťuje třída `SimpleMarker`, jež je schopná vytvořit libovolnou kombinaci markeru (např. CSV, okno 3×3, barvy čtverců na středu, velikost 15×15).

Loader je zodpovědný za inicializaci markeru. Jelikož jde vždy o podobný problém (procházením oknem a postupné vytváření oken) lišící se pouze způsobem uložení markeru (SVG, CSV), je zde `AbstractLoader`, který vyžaduje po potomkovi pouze barvu čtverce na určité pozici.

MarkerManager je databáze dostupných markerů.

DetectorPreviewCallback je volán vždy, když kamera obdrží snímek. Každý snímek je zpracováván detektorem. Protože není předem znám přesný počet klientů majících zájem o výsledek detekce (`PreviewFrame`, edgely), jedná se *Observer*, do něhož se klienti registrují. Po každé detekci jsou upozorněni všichni klienti.

4.4 Nalezení edgelů – adaptivní prahování

Zpracovávaný snímek je často velmi zašuměný, barva kolísá a ne každý pixel má barvu podobnou barvě pixelu předcházejícího (následujícího). Snímek je také rozostřen, což způsobuje rychlý pohyb kamery. Hledané hrany tedy nejsou *ostré*, ale *rozmazané* (gradient hrany je velmi široký). Detektor hran uvažuje zašuměný rozostřený snímek. Aby bylo možné detekovat hranu v takovém to snímku, hledá se *vrchol* (maximum) gradientu pomocí adaptivního prahování.

Navrhl jsem adaptivní prahování využívající dvě okna s plovoucím průměrem. Okna jsou umístěna za sebou a pohybují se současně po mřížce (*žluté* horizontální, vertikální přímky na obrázku 3.3). Do prvního (na obrázku 4.1 do plovoucího průměru vpravo) je vkládán aktuální pixel, do druhého poslední pixel z okna předcházejícího. Sleduje se absolutní hodnota rozdílu oken. Jakmile rozdíl obou oken přesáhne práh $T = 25$, hledá se největší absolutní hodnota, dokud rozdíl neklesne pod práh. Jsou-li obě okna ve stejné barvě, absolutní hodnota rozdílu oken musí být pod prahem. Detekce hran je na obrázku 4.1.

Obrázek 4.1: Hledání hran pomocí dvou oken plovoucího průměru.

V původní implementaci je použito jedno velké okno s plovoucím průměrem, kde se porovnává aktuální pixel s hodnotou průměru okna.

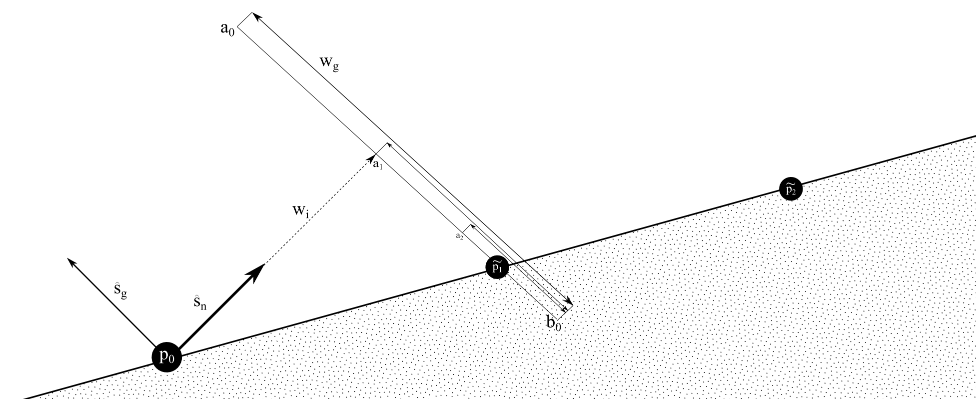
4.5 Upřesnení odhadu směru přímek

Pro každý nalezený edgel je hledána celá hrana (přímka), které odpovídá. Nejdříve se odhadne směr gradientu $\mathbf{s}_{gi}$, pomocí Sobelova operátoru 4.1 (tj. první derivaci v bodě). Normála směru gradientu $\mathbf{s}_i$ udává hledaný sklon přímky. Sobelův operátor odhadne směr velmi nepřesně, z toho důvodu je sklon dále upřesňován. $\mathbf{s}_{gi}$ i $\mathbf{s}_i$ jsou vždy normalizovány.

Při upřesňování se poskočí ve směru odhadu o w rovnice (3.3). Následně jsou vypočteny dva body

$$\mathbf{a}_i = \mathbf{p}_i + w_g \hat{\mathbf{s}}_{gi} \quad \mathbf{b}_i = \mathbf{p}_i - w_g \hat{\mathbf{s}}_{gi},$$

(kde w_g je polovina velikosti úsečky), jenž tvoří úsečku rovnoběžnou s odhadem gradientu. Půlením intervalu je nalezena hrana (vrchol gradientu). Pokud je úsečka příliš dlouhá a není možné úsečku dále dělit (rozdíl barev $\mathbf{a}$ a $\mathbf{b}$ je menší než práh) nebo nalezený bod $\widetilde{\mathbf{p}}_i$ překračuje povolené odchýlení od původního směru, je tento odhad zahozen. Pokud je $\widetilde{\mathbf{p}}_i$ přijat, je upraven sklon $\widetilde{\mathbf{s}}_i$. Další skok se provádí z nově upraveného sklonu (v původní implementaci se vždy skáče z prvotního odhadu).



Obrázek 4.2: Upřesnění směru přímky.

Může nastat situace, kdy bod $\widetilde{\mathbf{p}}_i$ leží v úhlové toleranci, ale patří jiné přímce. Je tedy vhodné znovu vypočítat odhad směru gradientu pro bod $\widetilde{\mathbf{p}}_i$ a porovnat jej s původním odhadem směru. Pokud odhad neodpovídá směru nebo je nulový, odhad $\widetilde{\mathbf{p}}_i$ je zahozen.

Pro další zpracování jsou použity přímky delší než 64 pixelů. Jelikož upřesnění přímek vyžaduje mnoho přístupů do paměti, zvolil jsem maximální možnou délku přímky 512 pixelů.

Protože původní odhad Sobelova operátoru je velmi nepřesný, je vhodné v prvních krocích upřesňovat blíže k edgelu. Zvolil jsem přístup, kdy se velikost kroku w mění s každým odhadem $\widetilde{\mathbf{p}}_i$, následujícím způsobem:

$$w_{i+1} = w_i + (10i + 5)$$
$$w_0 = 0.$$

Nejdříve je přímka upřesňována ve směru odhadu, poté je směr odhadu a směr gradientu otočen o 180°. Tímto způsobem je docíleno upřesnění přímky v obou směrech.

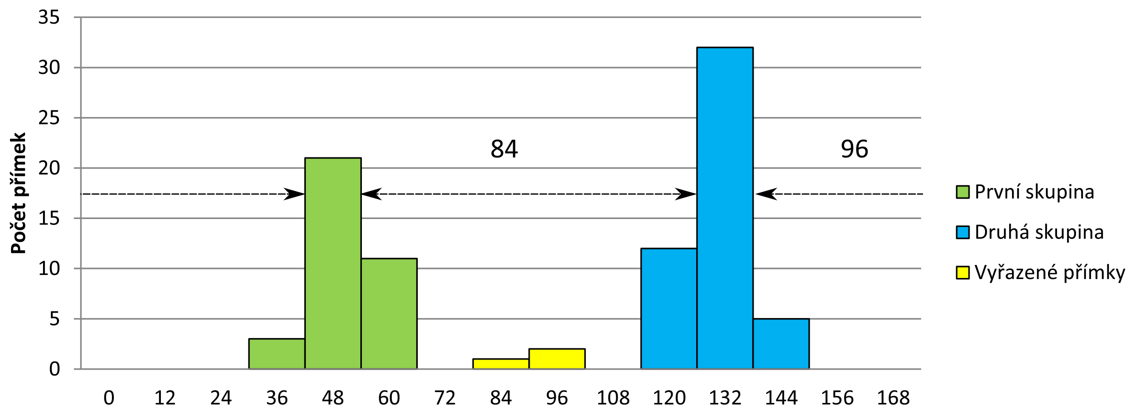
V první fázi implementace byl při porovnání barev brán pro každý pixel průměr okolí, aby bylo možné upřesňovat i v zašuměných snímcích. Dalším testováním jsem zjistil, že upřesňování přistupuje velmi často do paměti (pro každý pixel). Tyto přístupy velmi zpomalovaly detekci a způsobovaly velké kolísání času detekce. Z toho důvodu jsem se rozhodl brát v úvahu pouze jeden pixel. Tím se výrazně snížil čas detekce, zmírnilo se kolísání času a detekce byla srovnatelně kvalitní při porovnání na testovacích snímcích.

Sobelův operátor Použil jsem následující Sobelův operátor, kde h odhad ve směru x a v ve směru y. Odhad je dostatečně přesný a nevyžaduje velké množství informací (z obrazu potřebuje pouze čtyři body).

$$h = \begin{pmatrix} -1 & 0 & 1 \end{pmatrix} \quad v = \begin{pmatrix} -1 \\ 0 \\ 1 \end{pmatrix} \tag{4.1}$$

4.6 Rozdělení shluků přímek do skupin

Před nalezením úběžníků je vhodné nejdříve přímky rozdělit do dvou hlavních skupin podle významných směrů. Jako řešení jsem navrhl histogram úhlů, kde jsou přímky nejdříve roztříděny podle jejich sklonu a následně se vyhledávají dva dominantní shluky určující hlavní směr.



Obrázek 4.3: Histogram úhlů – *zelená, modrá* hlavní skupiny. *Žlutá* – zahozené přímky a nepočítá se s nimi v dalším zpracování. Histogram je rozdělen na 15 hodnot po 12°.

Detektor nejprve vloží přímku do histogramu podle sklonu. Následně v histogramu vyhledá první maximum, a to označí jako shluk A. Poté se hledá další maximum, které je vzdáleno od prvního alespoň o 30°, toto maximum je označeno jako B. S histogramem se pracuje jako s kruhovým bufferem. Přímky do 10° okolo maxima jsou přiřazeny do skupiny, ostatní přímky jsou zahozeny. Histogram se skupinami je na obrázku 4.3. Výstup tohoto kroku je na obrázku 3.3.

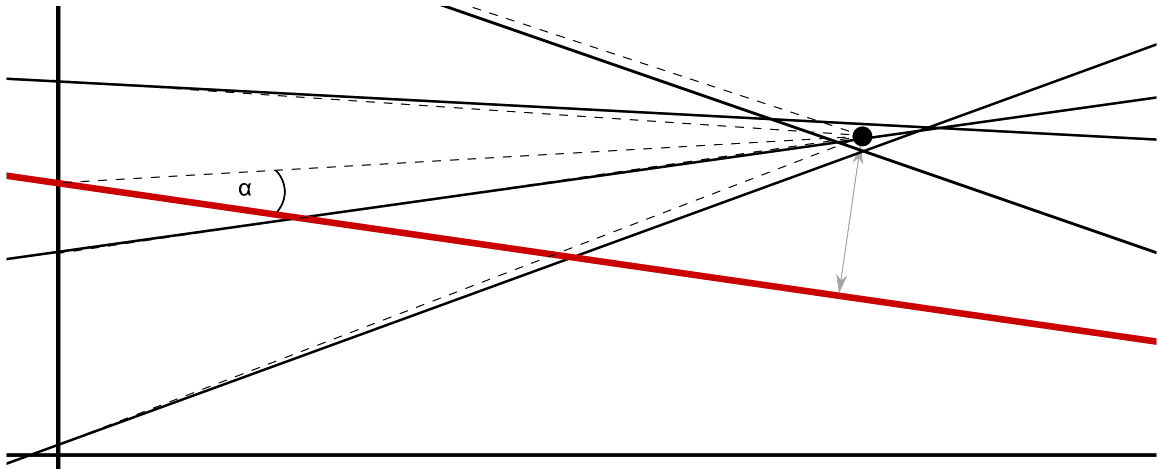
Úhel přímek je vypočten jako $\alpha = \arctan(\frac{-a}{b})$, a poté je převeden do intervalu $\langle 0°, 180° \rangle$ stupňů, přičtením k zápornému úhlu 180° (tj. pootočení vektoru o 180°). Je nutné brát úhel přímky α, aby rozlišení pro všechny úhly bylo stejné. Nestačí vzít x-ovou souřadnici směru přímky $\hat{\mathbf{s}}_i$, i když představuje kosinus úhlu, protože by do některých hodnot histogram spadalo více úhlů.

Požadavek dalšího zpracování je, aby detekované středy mřížky byly uspořádány tak, aby $\mathbf{l}_{i_0} \times \mathbf{l}_{j_0}$ byl vlevo nahoře, průsečík $\mathbf{l}_{i_n} \times \mathbf{l}_{j_n}$ vpravo dole (navíc pří vykreslování nedocházelo k přeblikávání skupin přímek). Jako prvotní odhad postačí shluk blíže 90° označit jako první skupinu. U toho postupu je problém při pootočení telefonu vzhledem k markeru o 45°, řešení je uvedeno v kapitole 4.9.

4.7 Zpřesnění výpočtu dvou úběžníků – filtrace přímek

Výpočet úběžníků pomocí eigendekompozice vyžaduje, aby přímky procházely úběžníkem, nebo alespoň s co nejmenší chybou. Protože marker může být překryt předměty, které mohou přidávat přímky neprocházející jedním z úběžníků, je vhodné tyto přímky detekovat a odfiltrovat. Základní odfiltrování provede histogram úhlů (kapitola 4.6 obrázek 4.3) a přímky příliš se odklánějící od hlavního směru jsou zahozeny. Existují případy, kdy si detektor s takto jednoduchým filtrováním nevystačí.

Může nastat situace, že přímka odpovídá hlavnímu směru, ale vzhledem k její pozici, neprochází úběžníkem. Takováto přímka může značně ovlivnit polohu vypočteného úběžníku eigendekompozicí a měla by být zahozena. Tato přímka je znázorněna na obrázku 4.4.



Obrázek 4.4: *Plná čára* – $\hat{\mathbf{l}}_i$, *přerušovaná* – $\hat{\mathbf{l}}'_i$. *Červená čára* – úhlem odpovídá skupině, ale neprochází úběžníkem. Taková to přímka zkresluje pozici vypočteného úběžníku.

Řešením tohoto problému je odhad přibližné pozice hledaného úběžníku. Nejdříve je vypočteno několik průsečíků náhodným výběrem dvojic přímek a to tak, aby nebyla vybrána

jedna přímka vícekrát a zároveň, aby přímky neležely na stejné detekované hraně markeru (pro tyto přímky platí $\hat{\mathbf{l}}_i - \hat{\mathbf{l}}_j \approx (0,0,0) \quad i,j \in \mathbb{N}$).

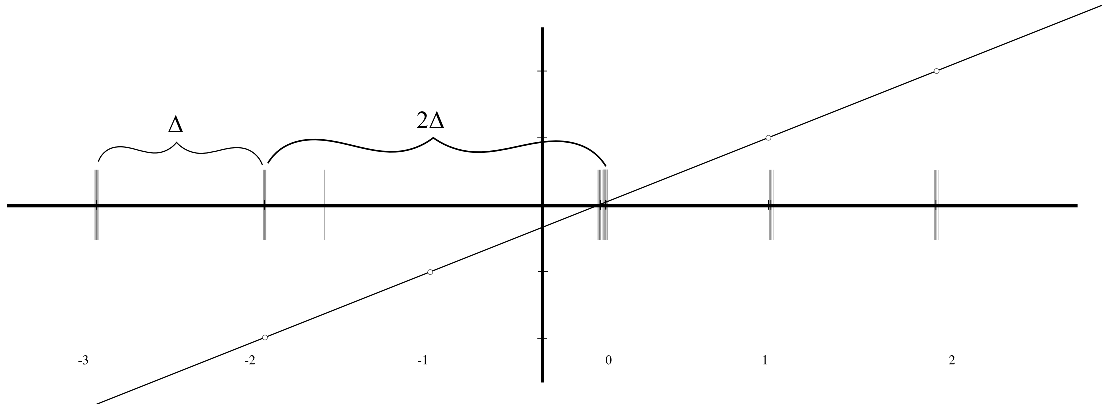
Po výpočtu průsečíků je hledán střed významného shluku. Je zvolen radius r a spočítáno kolik bodů a s jakou chybou (druhá mocnina vzdálenosti mezi body) padlo do tohoto okolí. Tento postup je opakován pro každý bod (lze i náhodně). Bod s největším počtem bodů a zároveň s nejmenší chybou je hledaný odhad úběžníku $\mathbf{v}'_i$. Rádius r je volen procentuálně podle nejvzdálenějšího bodu (čím je úběžník dále, tím je větší chyba a rozptyl průsečíků).

Je-li vypočten odhad úběžníku, dojde k odfiltrování přímek následujícím způsobem. Měření vzdálenosti $\mathbf{v}'_i$ od přímky není vhodné, neboť se vzdáleností úběžníku roste chyba, a tedy i vzdálenost odhadu úběžníku od přímky. Pro každý odhad by se musela vypočítat nová přípustná odchylka od úběžníku. Jako dostačující řešení jsem zvolil úhlovou toleranci přímky a úběžníku. Je zvolen bod na přímce $\mathbf{a}_i$ (pro všechny přímky stejný, například pro $x = 0$) a vypočten úhel mezi přímkou $\mathbf{l}_i$ a přímkou $\mathbf{l}'_i = \mathbf{a}_i \times \mathbf{v}'_i$. Pokud tento úhel přesáhne toleranci, je tato přímka zahozena. Způsob měření je znázorněn na obrázku 4.4.

Dále algoritmus vypočte úběžníky eigen dekompozicí a aktualizuje směr přímek tak, aby procházely úběžníkem. Tato část nebyla implementována, protože problémy s rotační maticí *FastCV* způsobily, že tento krok musel být vynechán z časových důvodů a ponechán, jako možné řešení do další verze detektoru.

4.8 Způsob nalezení vějířů a reprezentace

V tomto kroku jsou známy úběžníky. Přímky jsou vypočteny tak, aby procházely bodem, kde byla detekována hrana, a nalezeným úběžníkem. Aby bylo možné zrekonstruovat mřížku (dále může být uvedeno jako *grid*) tvořící hrany markeru, je třeba najít funkci $f(i) = ki + q$ představující $ki + q$ pro i-tou přímku, i může být i desetinné číslo (například $f(0,5)$, které zobrazuje přímku procházející středy čtverců). Tato funkce je na obrázku 4.5. Následně může být libovolná hrana markeru vypočtena dle rovnice (3.7), kde $b = 1$.



Obrázek 4.5: *Svislé čáry* – $ki + q$ pro jednotlivé přímky. *Přímka* – výsledná funkce pro $f(i) = ki + q$ nalezená lineární regresí. Δ hledaný krok.

Aby bylo možné najít funkci $f(i)$, je třeba znát krok (vzdálenost mezi shluky). Níže je uveden navrhovaný postup nalezení kroku Δ.

1. **Výpočet** $ki + q$ pro každou přímku, podle rovnice (3.7), známé proměnné jsou $\mathbf{I}_{base}, \mathbf{h}_{base}$, neznámé jsou b (není známa velikost vektoru přímky) a $ki + q$. Jde tedy

o tři rovnice o dvou neznámých. Pokud jsou přímky rovnoběžné, je úběžník v nekonečnu a zároveň musí být $a = 0$, nebo $b = 0$ přímky $\mathbf{l}$, proto rovnice, která má nejmenší $\mathbf{l}$ je zahozena.

2. **Seřazení** přímek podle $ki + q$. Implementováno řadícím algoritmem *QuickSort*.

3. **Určení středu shluků, ohodnocení** podle počtu přímek, které obsahují. Přímky jsou procházeny zleva doprava. Během procházení přímek je aktualizován střed shluku. Je-li následující $ki + q$ větší ε, je shluk ukončen, vynulován střed, pokračuje se hledáním dalšího shluku.

4. **Nalezení kroku** Δ mezi shluky. Zde se předpokládá, že některé shluky chybí, jiné mohou být mimo krok. Hledá se nejčastější vzdálenost Δ. Projdou se všechny shluky a je měřena vzdálenost k následujícímu shluku. Tato vzdálenost se uloží, buď již k existující vzdálenosti, pokud jejich rozdíl nepřesáhne ε, anebo je vytvořena nová. Každá vzdálenost je ohodnocena počtem čar, které ji tvoří. Při vložení nové vzdálenosti k již existující, je upravena vzdálenost váženým aritmetickým průměrem podle rovnice (4.2)

$$\Delta_{new} = \frac{\Delta r + \Delta_{new} r_{new}}{r + r_{new}}. \tag{4.2}$$

   Postupným upravováním vzdálenosti může nastat situace, kdy stejné (rozdíl menší ε) vzdálenosti jsou uloženy vícekrát. Posledním průchodem přes vzdálenosti je kontrolována jejich podobnost, podobné Δ jsou spojeny opět podle rovnice (4.2).

   Vzdálenost s největším ohodnocením je prohlášena za hledaný krok Δ.

5. **Přiřazení** i každému shluku, indexováno od 0. Po lineární regresi je výsledná přímka $((a, b, c).(x, y, 1) = 0)$ posunuta

$$c' = a \left\lfloor \frac{c}{\Delta} \right\rfloor$$

   tak, aby přímka s indexem 0 byla co nejblíže vpravo od $\mathbf{I}_{base}$.

   Chybí-li některý shluk, musí mu být přiřazen správný index. Index následujícího shluku je vypočten jako

$$i_{n+1} = i_n + \left\lfloor \frac{(ki + q)_{n+1} - (ki + q)_n}{\Delta} \right\rfloor$$

   .

6. **Lineární regrese** – proložení přímkou.

4.9 Lokalizace pozice uvnitř markeru

Detektor má jediný cíl, a to, co nejpřesněji detekovat středy čtverců markeru v obraze, předat jejich souřadnice s barvou. Výpočet středů čtverců je popsán v rovnici (3.10). Detektor nerozlišuje mezi zamýšlenou velikostí okna, uložením oken nebo způsobem identifikace oken (pomocí hashe), z tohoto důvodu vrací více středů tvořící mříž. Aplikace poté sama rozhodne, jak tyto středy zpracovat.

4.9.1 Výpočet středů ve směru čtení

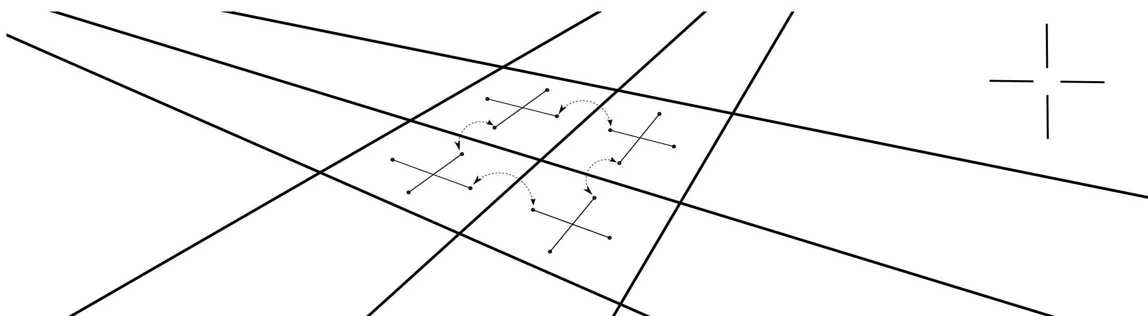
Pro další zpracování je nutné, aby detekované středy byly uloženy zleva doprava (aby $\mathbf{l}_{i_0} \times \mathbf{l}_{j_0}$ byl vlevo nahoře, $\mathbf{l}_{i_n} \times \mathbf{l}_{j_n}$ vpravo dole). Výpočet průsečíků v jiném směru může být způsoben přiřazením opačných indexů při hledání kroku 4.7. Směr čtení je detekován ještě před samotným výpočtem středů čtverců, a to vektorovým součinem přímek

$$\mathbf{a} = \mathbf{l}_{base_i} \times \mathbf{l}_{base_j}.$$

Je-li vektor **a** směrem do obrazu (souřadnice $c < 0$), čte se z levého dolního rohu do pravého horního. Správného čtení je docíleno překlopením přímky $f(i) = (-1)ki + q$. Následně jsou středy vypočteny podle rovnice (3.10).

4.9.2 Barva čtverce – rozložení barev

Někdy je vhodné znát i jiné barvy detekovaného čtverce, nejen pouze středovou. Pokud je snímek pořízen za špatných světelných podmínek, mohou se barvy mezi středem a hranou výrazně lišit. Z tohoto důvodu je vhodné porovnávat barvy, nacházejí se, co nejblíže u sebe (blíž k hraně, ke které patří, obrázek 4.6).



Obrázek 4.6: Způsob výpočtu hashe $2x2$ typu *cross* pomocí více barev z různých pozic.

4.9.3 Zpracování detekovaných středů čtverců markeru

Jsou-li detekovány středy, ladící aplikace zjistí, na jakou část markeru se kamera dívá, nalezením orientovaného okna v mřížce. Níže jsou popsány způsoby nalezení okna.

Procházení Rozhodl jsem se pro jednoduché procházení gridu, kde se hledá první platné okno.

Kruhový pohyb Alternativou je procházení gridu kruhovým pohybem od středu ke kraji.

Největší absolutní hodnota Může nastat situace, že čtverce budou mít různou hodnotu, ale pod osvětlením tyto barvy splývají. Okna s méně se lišícími barvami jsou náchylná pro detekci. Z toho důvodu je vhodné vybrat taková okna, která mají největší absolutní hodnotu rozdílu hran.

Rozhodovací strom V článku *Five Shades of Grey for Fast and Realiable Camera Pose Estatimation* [2] je použit postup vyhledávání pozice v markeru prostřednictvím rozhodovacího stromu. Výhodou tohoto postupu je, že stačí porovnat malý počet hran pro získání pozice a jsou tolerovány chyby (například u těžko rozlišitelných čtverců). Nevýhodou je vysoká paměťová náročnost.

Kapitola 5

Implementace a optimalizace aplikace

V této kapitole jsou popsány implementační detaily. Také jsou zde uvedeny optimalizační techniky, použité nástroje a způsob nastavení. Celá aplikace, s výjimkou detektoru, je implementována v jazyce *Java*, protože Android *SDK* (*Software Development Kit*) je primárně určen pro tento jazyk. Alternativní možností je využití *NDK* (*Native Development Kit*) určen pro jazyk *C/C++*, zde však není taková podpora, jakou nabízí *SDK*. Detektor byl napsán v jazyce *C*. Hlavním důvodem byla rychlost detektoru, jelikož *Java* není kompilovaný, ale interpretovaný jazyk. Překlad *C/C++* je představen v části 5.1.3. Rozhraní *JNI* sloužící pro komunikaci mezi nativním kódem a interpretovaným kódem, je prezentováno v části JNI 5.1.2.

5.1 Android

Tato část kapitoly je zaměřená na drobné implementační detaily týkající se mobilní platformy Android.

Minimální verzi SDK, podporovanou zařízením, jsem stanovil na verzi 8 (Android 2.2.x *Froyo*), cílovou verzi jsem zvolil 14 (Android 4.0, 4.0.1, 4.0.2 *Ice cream Sandwitch*).

5.1.1 Vykreslování informací

Vykreslování informací probíhá ve třech vzájemně se překrývajících nezávislých `View`, které vyplňují celou plochu obrazovky. Textové informace jsou zobrazeny pomocí `TextView`.

První je `CameraView` obstarávající vykreslování náhledu snímků kamery. Toto vykreslování zajišťuje sama kamera, po zaregistrování `SurfaceHolder` z `CameraPreview`. `View`, jakmile je to možné, obdrží zprávu vyžadující překreslení. Zde dochází k vykreslení ladících informací (pokud jsou k dispozici).

Jako druhé je zde `GLSurfaceView`, kde dochází k vykreslování 3D objektů pomocí *OpenGL ES*. U tohoto `View` nedochází k obnově kontextu. Po přesunu aplikace do pozadí, je kontext vždy zrušen a opětovně vytvořen, jakmile je aplikace viditelná (`onResume()`). Rovněž jsou také zahozeny veškeré informace a nastavení (textury, projekční matice, frustrum a další).

Poslední `MarkerSurfaceView` slouží k vykreslení nalezené pozice v markeru a detailu nalezeného okna.

Canvas

Vykreslování všech informací probíhá na `Canvas` v metodě `onDraw (Canvas c)` zodpovědného `View`. Nevýhodou tohoto vykreslení je, že není hardwarově akcelerované. To způsobuje, že pokud je třeba vykreslit příliš mnoho informací, dochází k pozdnímu vykreslení. Protože se jedná o ladící informace, zpoždění je akceptováno. Pro reálné aplikace, by bylo vhodnější všechny informace vykreslovat pomocí *OpenGL ES*.

OpenGL ES

Android disponuje *OpenGL ES*, což představuje standard odvozený od *OpenGL*. Má několik rozhraní odvozených od `GL`, jako `GL10` pro použití *OpenGL ES 1.0*, `GL11` pro verzi 1.1, `GL10Ext`, rozšířená verze 1.0, `GL20` pro nejnovější verzi 2.0 (tato verze není podporována staršími telefony) [5]. Pro účely ladící aplikace jsem zvolil *OpenGL ES* ve verzi 1.0.

Aby bylo možné vykreslovat předem připravené objekty, použil jsem knihovnu na načtení `*.obj` souborů od Balazs Karsay[1]. `ObjLoader` nebyl původně napsán pro Android, proto jsem knihovnu upravil tak, aby nebyla závislá na cestě k souboru a 3D objekt byl načítán z `InputStream`.

5.1.2 Java Native Interface

Java Native Interface je prostředek, jak propojit nativní kód přeložený pro danou platformu a bytecode. Je potřeba mít přeloženou knihovnu v adresáři *lib/*, pojmenovaná prefixem *lib* (například *libumf*). Knihovna je načtena statickým inicializátorem třídy. Funkce knihovny jsou v třídě uvedeny klíčovým slovem `native`. Funkce exportované z *C* musí mít prefix jména balíku oddělený podtržítkem. Například pro třídu `NativeProcessor`, funkci `free()` v balíku `fit.bp.umf.jni` je funkce v *C* pojmenována

```
Java_fit_bp_umf_jni_NativeProcessor_free ().
```

5.1.3 NDK – překlad do nativního kódu

Překlad zdrojových kódu na mobilní procesory a základní *API* poskytuje *NDK*. Překlad zajišťuje `ndk-build`. Je potřeba specifikovat `Andoird.mk`, který má podobnou syntax, jako `Makefile` a `Apliaction.mk`, kde jsou uvedeny platformy, pro něž bude knihovna přeložena. Zvolil jsem `armeabi` dostačující na většinu telefonů a `armeabi-v7a` určené procesorům disponujícím hardwarovou *FPU*. Je možné aplikaci přeložit i na `x86`, ovšem detektor je závislý na *FastCV*, které není přeloženo pro tuto platformu.

5.1.4 Kamera

Aby bylo možné získávat data z kamery, aplikace musí mít povolena práva v Manifest souboru. Je důležité, aby aplikace správně uvolňovala alokované zdroje, tedy při přechodu aplikace do pozadí, je nutné uvolnit, odregistrovat všechny `callback` funkce, pozastavit prohlížení, zastavit kameru.

Získání náhledu kamery je provedeno zaregistrováním `camera.addPreviewCallback()`. Kamera volá tento `Callback`, jakmile obdrží snímek. S každým snímkem dochází k alokaci

[1] Dostupné na `http://sourceforge.net/projects/objloaderforand/` pod *GNU General Public License version 3.0 (GPLv3)* licencí.

nového prostoru pro další náhled. Toto chování značně narušuje plynulý běh aplikace, obzvláště na méně výkonných zařízeních. Z tohoto důvodu jsem použil `camera.addCallbackWithBuffer ( previewCallback )`, poté byly přidány tři buffery `camera.addCallbackBuffer(new byte[dataBufferSize])`. Buffery kamery jsou uloženy jako fronta, je tedy nutné po zpracování snímku (v `PreviewCallback`), snímek opět vložit do fronty bufferů.

Ve výchozím nastavení mohou být snímky velmi neostré. Některá zařízení mají zabudovanou podporu automatického ostření. Režim ostření jsem zvolil

```
parameters.setFocusMode(Camera.Parameters.FOCUS_MODE_CONTINUOUS_VIDEO ),
```

který plynule provádí ostření během pohybu.

5.1.5 Formát obrazu YUV

Výchozím formátem náhledu kamery je NV21 (jde o YUV420p). Nejdříve jsou na jednom `byte` uloženy Y složky (jasové), následně U, poté V. Pro každé čtyři pixely je zde jedna U a jedna V složka na čtyřech bitech [10]. Při detekci gridu se využívá pouze Y složka, s obrazem se tedy pracuje, jako s černobílým.

5.2 Optimalizace rychlosti – Pool, inline funkce

V této části jsou popsány dvě nejdůležitější optimalizace. Přizpůsobení některých kroků detekce byla popsána v návrhu aplikace (kapitola 4).

Edgely – alokace paměti Rychlost zpracování může značně ovlivnit alokace paměti. Není vhodné, pro každý nalezený edgel alokovat nový prostor. Z tohoto důvodu se využívá již, předem naalokované pole edgelů (výchozí velikost pole je 256), které se inicializuje před začátkem zpracování každého snímku (ukazatel na poslední prvek je nastaven na začátek pole). Přidání edgelu probíhá změnou všech hodnot posledního edgelu v poli a zároveň je posunut ukazatel konce na další prvek. Pokud má ukazatel posledního prvku stejnou hodnotu, jako velikost naalokovaného pole, pole je zvětšeno na dvojnásobek své velikosti. V celé aplikaci, kde je to možné (detektor, ladící aplikace, vykreslování), se recyklují naalokovaná data ze snímku předcházejícího (nejsou využívány hodnoty, ale pouze prostor).

Inline funkce Pri překladu je funkce rozvinuta (instrukce funkce jsou přímo vloženy na místo, kde dochází k volání). Při běhu programu tak nedochází k volání funkce (několik desítek taktů procesoru), ale přímo k vykonání instrukcí. Kód využívající `inline` funkce může značně ovlivnit velikost přeložené knihovny. Pokud je některá funkce volána příliš často (např. přidání pixelu do plovoucího průměru), je vhodné tuto funkci přepsat na `inline`, a tím ušetřit mnoho taktů procesoru. `Inline` funkce nemohou být rekurzivní a neměly by přesáhnout maximální počet instrukcí. Ten je možné nastavit pomocí parametru překladu

```
--param max-inline-insns-single = 4096.
```

5.3 Reprezentace markeru

Okna markeru jsou uložena v `TreeMap`, jde o červenomodrý binární strom. Klíčem je vypočítaný hash okna. Každé okno je vloženo do stromu se všemi rotacemi (sever, jih, východ,

západ). Při vyhledávání na základě klíče, dochází k navrácení okna, z něhož je následně možné zjistit rotaci a pozici modulu v markeru.

Uložení markeru je nezávislé na velikosti okna nebo způsobu vytváření hashe. Při vytváření markeru se používá továrna (`Factory`), která sama vytvoří hash podle barev.

5.3.1 Podporované formáty markerů – SVG, CSV

Aplikace podporuje dva typy formátů marker. Markery uložené v SVG formátu jsou načítány pomocí knihovny SVG-Android[2]. Dalším možným způsobem načtení markeru je uložení v CSV, kde buňky obsahují index barev. Výhodou uložení v CSV je menší paměťová náročnost, neboť u SVG je třeba načíst celou bitmapu. Jinou výhodou může být snadná možnost přebarvení markeru prostřednictvím třídy `Palette`. Nevýhodou SVG-Android je, že neimplementuje celý standard SVG, z tohoto důvodu musejí být markery upravovány tak, aby je mohla knihovna správně zpracovat.

5.3.2 Okno – Reprezentace hash

Jako výchozí jsem zvolil velikost okna 3×3, dostačující pro většinu markerů, které jsem měl k dispozici. Klíč pro šedé markery je možno uložit na 32b (12 hran po dvou bitech). Hash okna vznikne porovnáním vnitřních hran. Okno se prochází zleva doprava, přičemž dochází k porovnávání mezi aktuálním čtvercem a čtvercem nad ním (pokud se nejedná o první řádek), následně je aktuální čtverec porovnáván s čtvercem vpravo (pokud aktuální není poslední ve sloupci). V současnosti jsou podporovány pouze černobílé markery. Rotace hashe vpravo je na obrázku A.2.

Hash pro barevné markery obsahuje tři hashe, každý pro RGB barevnou složku. Výsledný hash je konkatenace R, G, B hashe. K dokončení podpory pro barevné markery, by bylo zapotřebí přepracování detektoru, aby středy čtverců vracely barvu ve formátu RGB. Tento krok však vyžaduje dopracování komparátorů pro GB barevnou složku.

5.4 Výpočet úběžníků eigen-dekompozicí

Z vektoru přímek je vytvořena matice, která je násobena transponovanou maticí (rovnice (3.6)). Výsledkem je vlastní matice a vlastní vektor. Následně se ve vlastním vektoru hledá minimální odchylka. Index nejmenšího čísla je řádkem v matici hledaného úběžníku v homogenních souřadnicích. Pozice úběžníku je nalezena po vydělení vektoru homogenní souřadnicí. Zdrojový kód výpočtu eigen-dekompozice vznikl úpravou kódu z knihovny *„Java Matrix – JAMA“*[3]

[2] SVG-Android od John Watkinson dostupné na `http://code.google.com/p/svg-android/` pod licencí *Apache License 2.0*

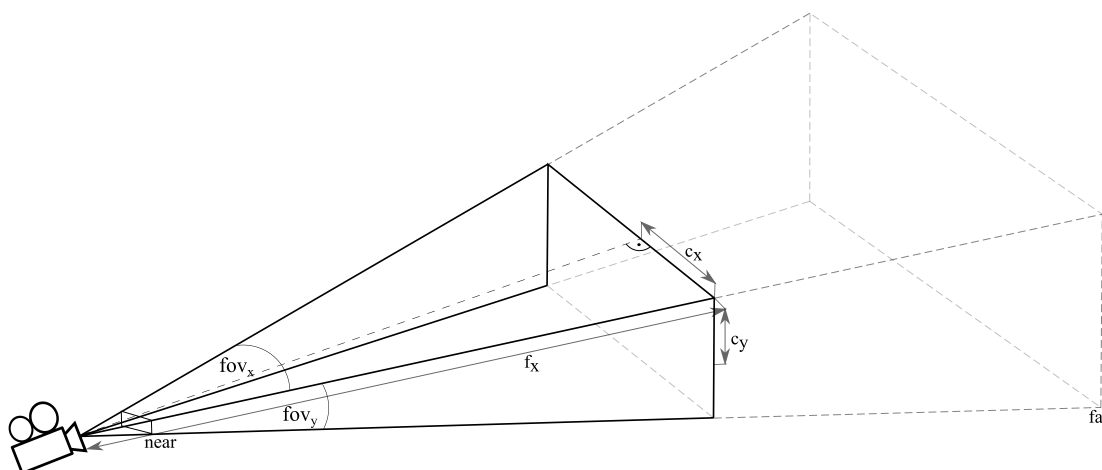
[3] Dostupné na `http://math.nist.gov/javanumerics/jama/` pod veřejnou licencí.

5.5 Kalibrace kamery

Aplikace vykresluje 3D objekty dle zjištěné pozice kamery. Frustrum *OpenGL ES* musí být nastaveno tak, aby odpovídalo kameře. Frustrum je vypočteno

$$\begin{aligned} left &= -h_s \\ right &= h_s \\ top &= -h_s/ar \\ bottom &= h_s/ar\,, \end{aligned}$$

kde $h_s = near\,|\tan(fov_x/2)|$, ar je poměr stran (*aspect ratio*).



Obrázek 5.1: Frustrum near-plane, far-plane. Zorný úhel kamery fov_x a fov_y. Střed obrazovky c_x, c_y.

Každý bod gridu musí být převeden do souřadného systému kamery

$$g_i^{'} = \left(\frac{g_{i_x} - c_x}{f_x} \quad , \quad \frac{h - c_y - g_{i_y}}{f_y} \right),$$

kde g_i je i-tý bod gridu, c_x, c_y je střed obrazovky, h je výška snímku. Snímek získaný z kamery má osu y rostoucí směrem dolů a $(0,0)$ se nachází v levém horním rohu. *OpenGL* má y rostoucí směrem nahoru a $(0,0)$ se nachází ve středu obrazovky. Z těchto důvodů bylo nutné body převést, tak aby odpovídaly *OpenGl* $(g_{i_x} - c_x,\ h - c_y - g_{i_y})$. Je-li znám zorný úhel kamery fov_x, tak $fov_y = fov_x$, tedy

$$f_x = \left| \frac{c_x}{\tan(fov_x/2)} \right|.$$

5.6 3D rekonstrukce

V tomto kroku je již grid detekován a jsou známy veškeré modelové (3D) souřadnice všech bodů gridu. Pro výpočet translačního vektoru a rotační matice prostřednictvím korespondencí (2D $\longrightarrow$ 3D) jsem použil funkci `fcvGeom3PointPoseEstimatef32` z knihovny *FastCV*[4]. Funkce očekává korespondence v následujícím tvaru

[4]Licence dostupná na https://developer.qualcomm.com/mobile-development/mobile-technologies/computer-vision-fastcv/fastcv-license-agreement

- `from` čtveřici 3D souřadnic
- `to` čtveřici 2D souřadnic
- `indices` určuje pořadí bodů

Pro čtyři body by měla funkce vrátit pole 4×3 (ve skutečnosti vrátí dvě velmi podobné možnosti) obsahující translační vektor a rotační matici. *FastCV* vrací translační vektor pro pozitivní osu z do obrazu, *OpenGL* ale směřuje negativní z do obrazu. Z tohoto důvodu bylo nutné při přípravě *ModelView* matice, otočit osy následujícím způsobem

```
gl.glScalef(1f, 1f, -1f);
    gl.glTranslatef(t[0], t[1], -t[2]);
    gl.glMultMatrixf(r, 0);
gl.glScalef(1f, 1f, -1f);.
```

Nepodařilo se mi vyřešit problém s rotací. Rotační matice ve většině případů neodpovídá očekávanému výsledku. Matice je někdy špatně rotována podle osy y, jindy podle osy x. Osa z přibližně odpovídá. Tento krok jsem řešil nejdéle, nicméně neúspěšně. Vzhledem k tomu, že nebyl cílem zadání práce, rozhodl jsem se od řešení upustit.

Kapitola 6

Testování a vyhodnocení výsledků

Pro testovací účely bylo ladící aplikací nahráno několik videí v rozlišení 960×544. Videa byla nahrána v YUV formátu ukládáním každého náhledu kamery do souboru. Tyto videa je možné najít na přiloženém disku. Videa byla rozdělena do několika kategorií, podle jejich hlavního pohybu (podobně jako v článku *Uniform Marker Fields* [7]). Byl použit šedotónový marker 15×15 vytištěný na standardní kancelářský papír velikosti A4. Marker byl již značně opotřebován, mírně zvlněný (možné vidět na obrázku 3.5), a tisk nebyl příliš kvalitní (bílá místa a šmouhy). Videa byla nahrána při statické pozici telefonu (maximální vzdálenost přibližně 30 cm, minimální přibližně 2 cm), kde se pohybovalo markerem, aby se zabránilo zkreslení informací způsobenému třesem ruky. Testováno na *HTC one S* (Android 4.0.1 i 4.1) s 1.7GHz *Snapdragon S3* procesorem. Rychlost zpracování byla měřena *C* funkcí `gettimeofday`.

Tabulka 6.1 zobrazuje úspěšnost extrakce modulu z gridu, zjištění jeho pozice a rotace. Jelikož je možné detekovat modul, který se nachází na jiné pozici nebo rotaci než by se očekávalo, je u všech videí předem známa rotace markeru. Ta se porovnává s rotací získaného modulu, pro ověření správné detekce rotace.

Nízká úspěšnost *Zoom* byla způsobena nezaostřením snímků (přibližně v polovině videa), snímek byl rozmazán a nebylo možné správně nalézt grid. Chybějící shluky přímek způsobily, že se šířka gridu zdvojnásobila (zobrazeno na grafu A.6 v příloze). Do testů

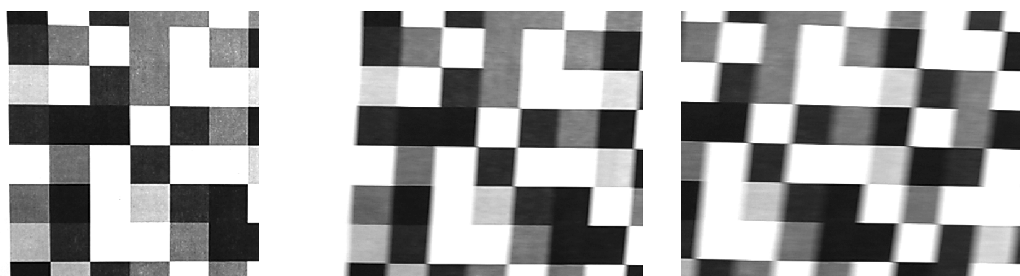
Typ videa	Úspěšnost	Použito bodů	Čas [ms]
Vertikální	98,9 %	3,33 %	2,41
Vertikální*	98,4 %	3,29 %	2,36
Horizontální*	91,1 %	3,14 %	2,64
Perspektiva	99,4 %	3,49 %	2,70
Perspektiva*	96,7 %	3,19 %	2,41
Zoom	95,2 %	3,28 %	2,46
Třepání	95,3 %	3,35 %	2,70
Pohyb**	95,7 %	3,24 %	2,38
Celkem	96,3 %	3,27 %	2,50

Tabulka 6.1: Úspěšnost detekce pozice markeru na použitých videích. Celkem 1 301 snímků. Offset mřížky 128. * rychlý pohyb, ** (*zig-zag*) pohyb.

Krok	Čas [ms]	%
Hrany	1,124	45,0 %
Přímky	0,988	39,6 %
Třídění	0,037	1,5 %
Úběžníky	0,021	0,8 %
Vějíře	0,036	1,4 %
Grid	0,016	0,6 %
Modul	0,262	10,5 %
Pozice	0,011	0,4 %
Celkem	2,497	100 %

Tabulka 6.2: Časy jednotlivých kroků detekce.

byl také zařazen horizontální posun, kde při rychlém pohybu dochází k deformaci snímku (obrázek 6.1), to způsobilo nízkou úspěšnost detekce modulu markeru.



Obrázek 6.1: Ukázka deformace snímku při rychlém horizontálním posunu.

Při použití offsetu mřížky 64 pixelů, úspěšnost detekce vzroste na 98,3 %, počet zpracovaných pixelů 6,64 %, čas 4,326 ms. Nutno dodat, že časy zpracování snímku se v reálné aplikaci dosti liší. Testy se pouze snaží detekovat marker a zpracovat výsledek. V reálné aplikaci ještě dochází k vykreslování informací (není HW akcelerováno), renderování snímku a dalších nezbytných operací, které zpomalují běh aplikace. Časy více kolísají, což nejspíše způsobuje *Garbage Collector*, nebo přepínání kontextu mezi vlákny. Průměrné zpracování snímku může být až dvojnásobné.

Úspěšnost detekce také značně ovlivňuje způsob zjištění modulu markeru, který by se měl vyrovnat s různým osvětlením a také se špatnou kvalitou markeru. Při extrakci modulu z gridu je vhodné upřednostňovat moduly s větší absolutní hodnotou, tedy moduly, které obsahují nejméně shodných čtverců (nejméně =).

V tabulce 6.2 jsou rozebrány časy jednotlivých kroků detekce. Čas značně ovlivňuje přístup do pamětí. Najde-li se příliš mnoho dlouhých přímek, čas detekce se výrazně prodlouží, což je dáno častým přístupem do paměti při upřesňování přímek. Hledání hran u vertikálního průchodu, kde je pro každý další pixel (řádek) nutné přeskočit velké množství dat, způsobí nevyužití *Cache* procesoru a zpoždění detekce.

Nejslabším místem detekce je nalezení úběžníků. Pokud se nenaleznou korektně, celá detekce je poté špatně. Pokud se pozice úběžníků příliš mění, mění se i sklon $\mathbf{I}_{base}$, ten způsobí naklonění gridu. Tedy třesou-li se úběžníky, bude se třást i detekovaná pozice kamery.

Pro měření třesu jsem použil šířku a výšku v pixelech detekovaného gridu. U videa s vertikálním posunem, je předpoklad konstantní velikosti, vzhledem ke kolmému pozorování markeru. Graf šířky je k vidění v příloze A.5, A.6 a A.7. Hladký průběh znamená kvalitnější detekci. Směrodatná odchylka šířky i výšky pro A.6 byla 0,77.

Na grafu A.7 je zobrazena velikost gridu pro perspektivní video. Marker byl nakláněn z kolmé pozice, až do přibližně čtyřiceti pěti stupňů. Výrazné skoky ve výšce jsou způsobeny posunem celého gridu, na jinou pozici markeru ($\mathbf{I}_{base}$ odpovídá hraně markeru). Skoky při přiblížení vznikly nezaostřením snímku (vzdálenost přibližně 2 cm), to způsobilo posun přímek a nenalezení správné velikosti gridu.

Kapitola 7

Závěr

Cílem práce bylo navrhnout a implementovat aplikaci, která detekuje hrany markeru, extrahovat modul (okno) a zjistit jeho umístění a rotaci v markeru. Byly popsány způsoby detekce dnes používaných markerů a také způsob detekce *UMF*. Seznámil jsem se s technologiemi pro vývoj na OS Android, také s *OpenGL ES* nebo způsobem propojení programovacího jazyku *Java* a nativního kódu.

Detektor gridu byl navržen s ohledem na slabý výkon mobilních zařízení, přičemž bylo dosáhnuto rychlosti detekce 2,497 ms na testovací sadě, s úspěšností detekce 96,3%. Detektor přináší jisté změny oproti původnímu řešení od I. Szentandrási [7], které bylo určeno pro PC. Některé postupy (adaptivní prahování) již I. Szentandrási použil ve své implementaci.

Byla také naimplementována ladící aplikace zobrazující jednotlivé kroky detekce. Dále poskytuje různé typy markeru, které je možné měnit za běhu programu. Dále umožňuje pořídit snímek s výsledkem detekce nebo natočit krátké videa použitelné pro testovací účely. Byl také částečně implementován způsob zjištění pozice kamery v prostoru. Zde se mi nepodařilo dohodnout s *FastCV*. Z tohoto důvodu při určitém naklonění telefonu dochází ke špatné rotaci (i translaci) objektu.

V testovací sadě byla zahrnuta videa s velmi rychlým pohybem, přesto bylo dosáhnuto úspěšnosti extrakce modulu 98,3% s offsetem mřížky 64 px. Detekce gridu na zaostřených snímcích je stabilní a nedochází k velkému třesu či zkreslení. Tyto výsledky řadím mezi hlavní úspěchy a považuji je za identifikátor rychlého a kvalitního řešení detektoru.

Slabé místo detekce vidím v určení úběžníku. Pokud dochází k jejich nesprávné lokalizaci nebo dochází k velkým skokům (třes, šum) mezi snímky, potom další kroky detekce vychází ze špatných údajů a výsledek bude velmi nekvalitní. Překvapilo mě, velké kolísání času ladící aplikace, které se neprojevovalo v testech a také velký rozdíl časů detekce mezi ladící aplikací a testy, což bylo nejspíše způsobeno přepínáním kontextu mezi vlákny. Detekci rovněž zpomaluje příliš častý přístup do paměti (obrazu).

Do další verze detektoru bych zahrnul jiný způsob extrakce modulu z gridu (založen na rozhodovacím stromu), a také jiný způsob zjištění pozice kamery (dosáhnout nezávislosti na *FastCV*). Také bych rád vytvořil použitelnou aplikaci (jednoduché demo nebo hru) s RR využívající všech výhod *UMF*. Plakát i video jsou umístěny na přiloženém disku.

Literatura

[1] Fiala M.: Fiducial marker system using digital techniques. In *IEEE Computer Conference on Computer Vision and Pattern Recognition*, ročník 2, Washington, DC, USA, 2005, s. 590–596.

[2] Herout A.; Szentandrási I.; Zachariáš M.; aj.: Five shades of grey for fast and reliable camera pose estimation, 2013.

[3] Hirzer M.: Marker detection for augmented reality application. Technická zpráva, Computer Graphics and Vision, Graz University of Technology, Austria, 2008.

[4] Kato I. P. H.; Billingurst M.: ARToolkit user manual, version 2.33. 2000, Human interface technologie lab, University of Washington.

[5] Smithwick M.; Verma M.: *Pro OpenGL ES*. Apress, 2012, ISBN-13 (pbk): 978-1-4302-4002-0, ISBN-13 (electronic): 978-1-4302-4003-7.

[6] Sood R.: *Pro Android Augmented Reality*. Apress, 2012, ISBN-13 (pbk): 978-1-4302-3945-1, ISBN-13 (electronic): 978-1-4302-3946-8.

[7] Szentandrási I.; Zachariáš M.; Havel J.; aj.: Uniform Marker Fields, 2013.

[8] Tateno K.; Kitahara I.; Ohta Y.: A nested marker for augmented reality. In *ACM SIGGRAPH 2006 Sketches*, SIGGRAPH '06, New York, NY, USA: ACM, 2006, ISBN 1-59593-364-6, doi:10.1145/1179849.1180039.

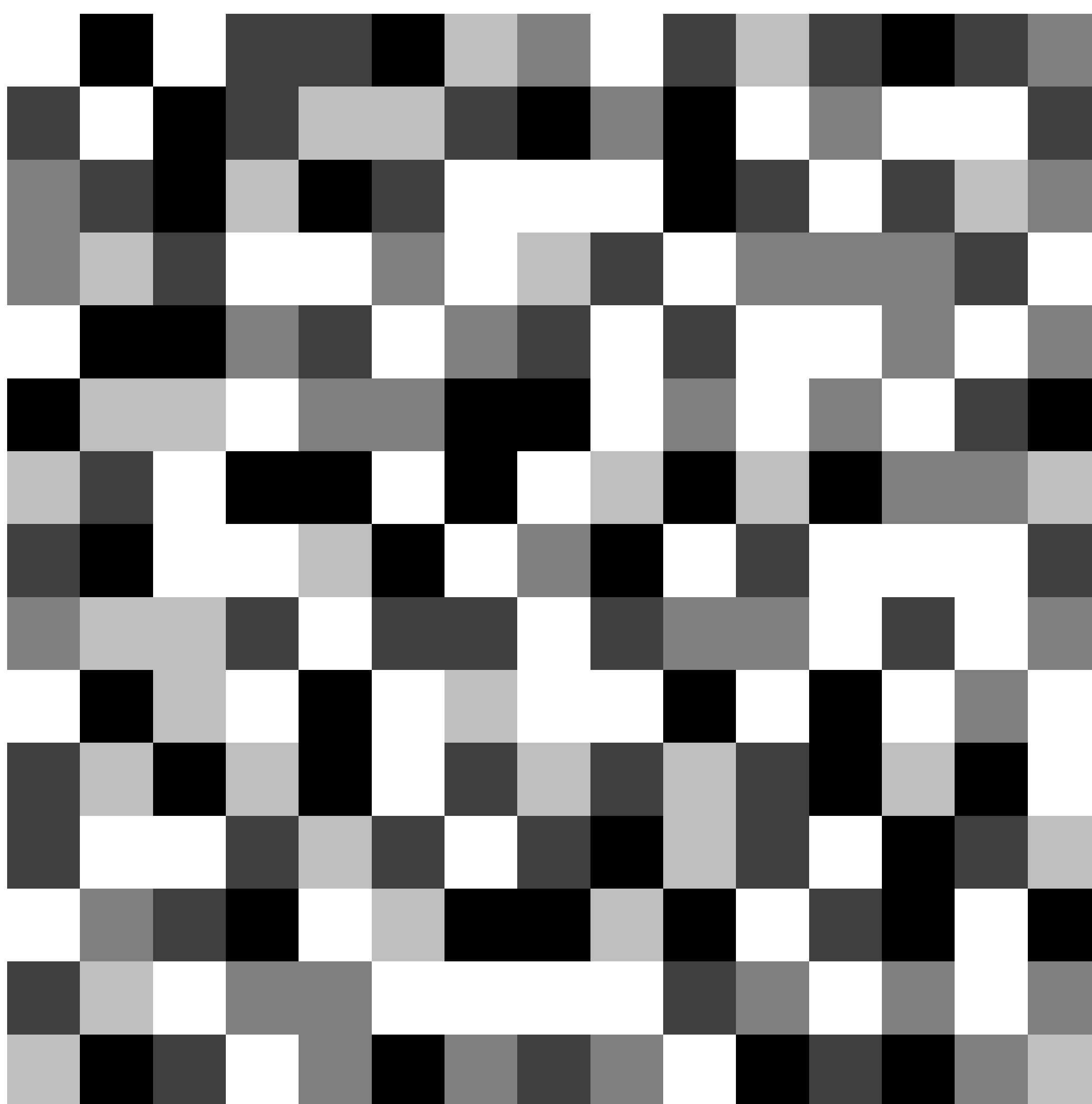
[9] Wikipedia: Augmented reality — Wikipedia, The Free Encyclopedia. 2013, [Online; poslední modifikace 22. 4. 2013].
URL http://en.wikipedia.org/wiki/Augmented_reality

[10] Wikipedia: YUV — Wikipedia, The Free Encyclopedia. 2013, [Online; poslední modifikace 14. 4. 2013].
URL http://en.wikipedia.org/wiki/YUV

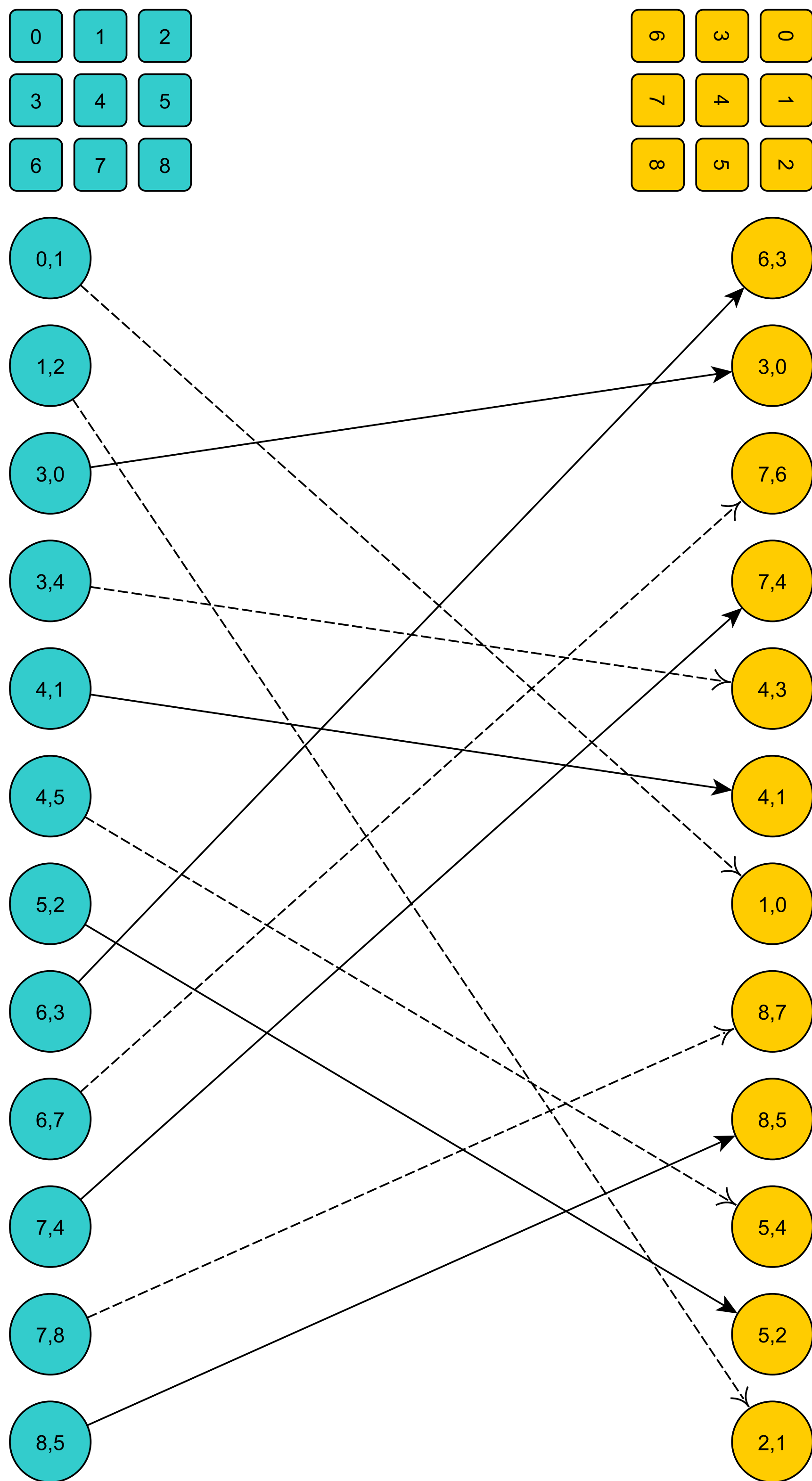
[11] Yu X.; Lai H. C.; Liu S. X. F; aj.: A gridding Hough transform for detecting the straight lines in sports video. In *IEEE International Conference of Multimedia and Expo*, 2005, ICME.

Příloha A

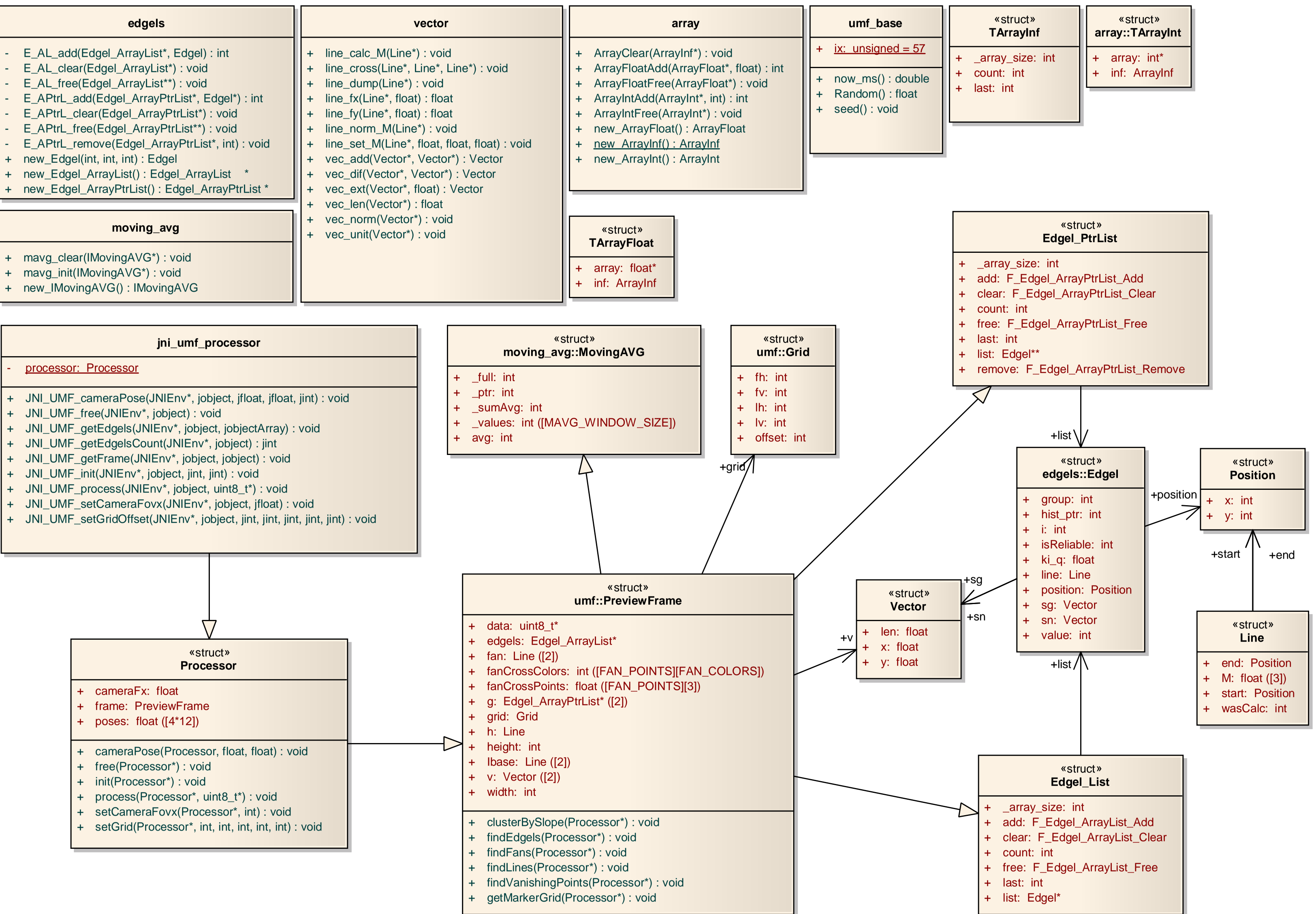
Obrázky



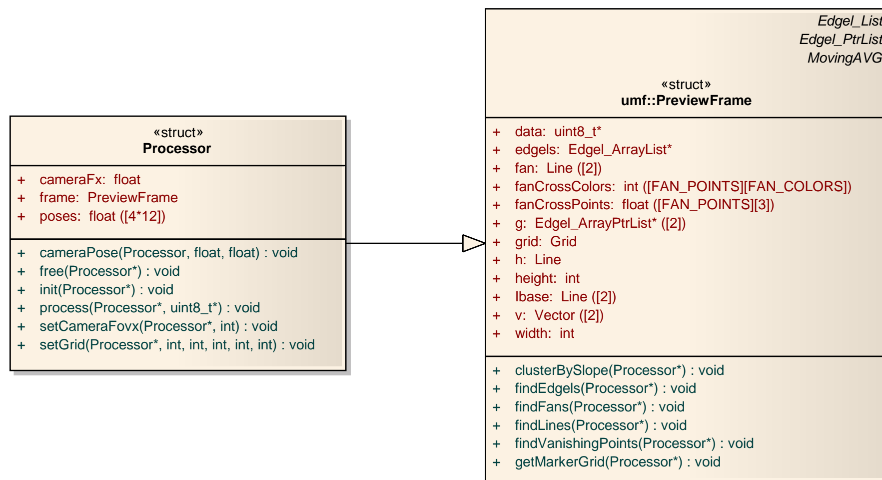
Obrázek A.1: Marker 15×15, velikost okna 3.



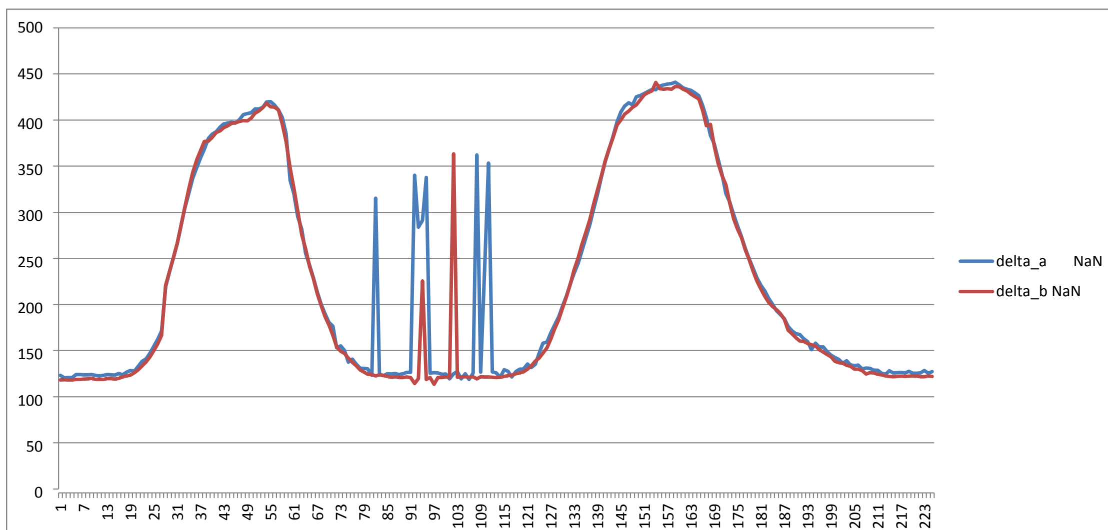
Obrázek A.2: Rotace hashe vpravo. *Přerušovaná čára* – značí negaci porovnání.



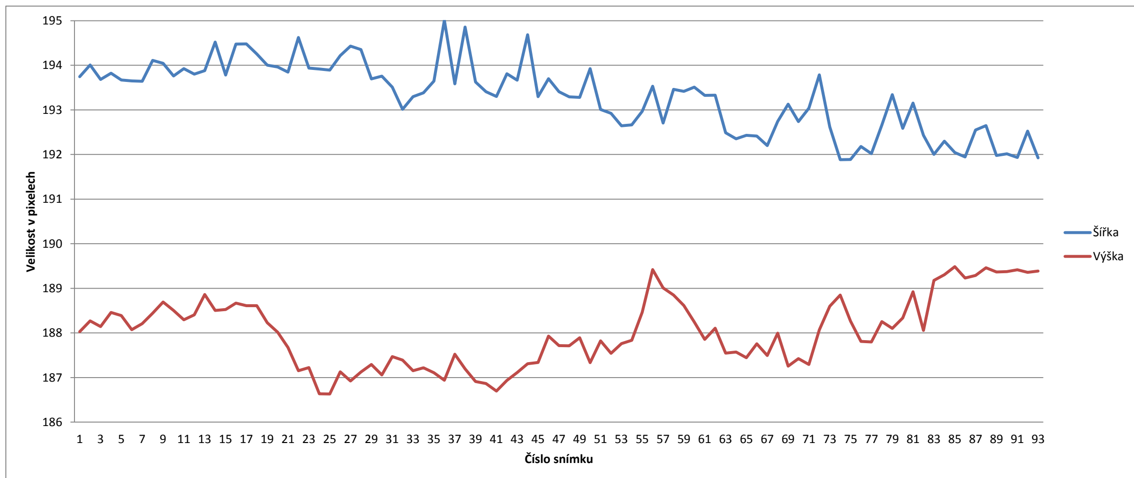
Obrázek A.3: Úplný diagram processoru.



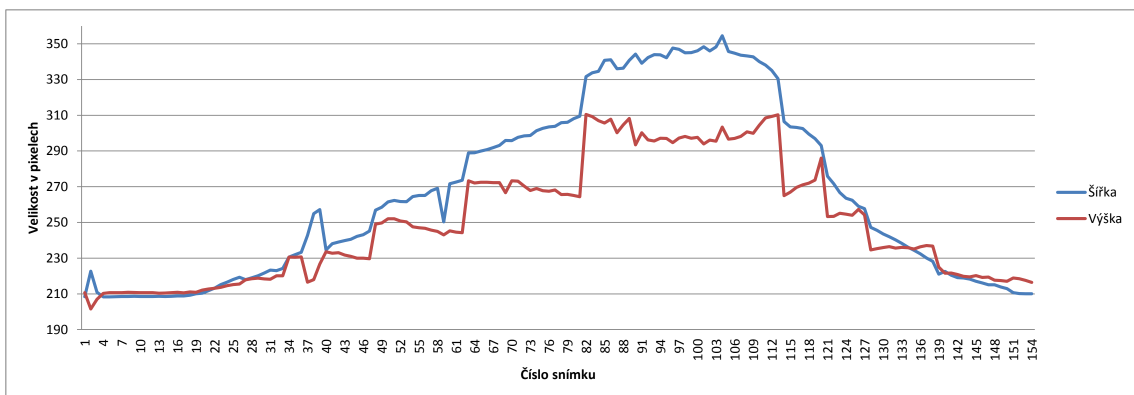
Obrázek A.4: Zjednodušený diagram processoru.



Obrázek A.5: Šířka gridu testovacího videa kategorie *Zoom*. Nezaostřený snímek způsobil, že bylo nalezeno málo přímek, mezery mezi shluky byly příliš velké. Výsledkem je špatně nalezený grid (dvojnásobná velikost).



Obrázek A.6: Šířka a výška gridu pro vertikální video.



Obrázek A.7: Šířka a výška gridu. Perspektivní video z $0° \rightarrow 45° \rightarrow 0°$. Skoky ve výšce jsou způsobeny skokem gridu na jinou pozici.