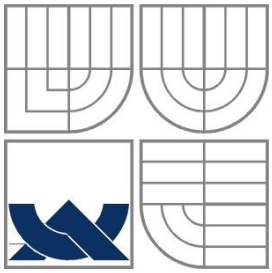
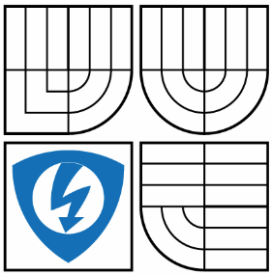


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
ÚSTAV AUTOMATIZACE A MĚŘICÍ TECHNIKY
FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION
DEPARTMENT OF CONTROL AND INSTRUMENTATION

PROJEKCE DAT DO SCÉNY
PROJECTOR CAMERA COOPERATION

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — BC. VIKTOR WALTER
AUTHOR

VEDOUCÍ PRÁCE — ING. MILOSLAV RICHTER, PH.D.
SUPERVISOR

BRNO 2016

VYSOKÉ UČENÍ
TECHNICKÉ V BRNĚ

Fakulta elektrotechniky
a komunikačních technologií

Ústav automatizace a měřicí techniky

Diplomová práce

magisterský navazující studijní obor
Kybernetika, automatizace a měření

Student: Bc. Viktor Walter
Ročník: 2

ID: 146999
Akademický rok: 2015/16

NÁZEV TÉMATU:

Projekce dat do scény

POKYNY PRO VYPRACOVÁNÍ:

Navrhněte systém pro promítání dat projektorem do scény (spolupráce snímající kamery a projektoru).
1) Seznamte se s mechanizmem pořízení dat, reprezentujících tvar scény, pomocí kamer.
2) Navrhněte mechanizmus a algoritmy pro kvalitní projekci dat do 2D a 3D scény.
3) Navrhněte vhodnou demonstrační úlohu (úlohy).
4) Realizujte navržené algoritmy a aplikujte je na navržené úlohy.
5) Zhodnoťte možnosti a výsledné parametry řešení.

DOPORUČENÁ LITERATURA:

Kraus K.: Photogrammetrie 1 und 2, Ummler / Bonn, 1996
Žára J., Beneš B., Sochor J., Felkel P.: Moderní počítačová grafika, Computer Press, 1998, ISBN 80-251-0454-0
Hlaváč V., Šonka M.: Počítačové vidění,Grada, Praha 1992, ISBN 80-85424-67-3
Faugeras O.: Three-Dimensional Computer Vision, The MIT Press 1993

Termín zadání: 8. 2. 2016 ***Termín odevzdání:*** 16.5.2016

Vedoucí práce: Ing. Miloslav Richter, Ph.D.
Konzultanti diplomové práce:

doc. Ing. Václav Jirsík, CSc.
předseda oborové rady

UPOZORNĚNÍ:

Abstrakt

Diplomová práce se zabývá spoluprací kamer a projektorů při promítání dat do scény. Popisuje prostředky a teorii potřebnou pro takovou spolupráci a navrhuje demonstrační úlohy. Součástí práce je program, který dokáže za pomoci projektoru a kamery získat potřebné parametry těchto zařízení. Kromě toho program dokáže i demonstrovat kvalitu kalibrace promítáním vzoru na objekt podle jeho aktuální polohy a natočení a rovněž i rekonstruovat tvar objektu za pomoci projekce strukturovaného světla. Rekonstrukci tvaru je možné demonstrovat promítáním vrstevnicového vzoru nebo vizualizací obtékající vody. Práce taktéž popisuje některé problémy a pozorování, ke kterým se při tvorbě a testování programu přišlo.

Kľúčové slová

Projekce, Kamery, Počítačové vidění, OpenCV, OpenGL, Strukturované světlo, 3D skenování, Kalibrace kamery, Kalibrace projektoru

Abstract

The focus of this thesis is the cooperation of cameras and projectors in projection of data into a scene. It describes the means and theory necessary to achieve such cooperation, and suggests tasks for demonstration. A part of this project is also a program capable of using a camera and a projector to obtain necessary parameters of these devices. The program can demonstrate the quality of this calibration by projecting a pattern onto an object according to its current pose, as well as reconstruct the shape of an object with structured light. The thesis also describes some challenges and observations from development and testing of the program.

Keywords

Projection, Cameras, Computer vision, OpenCV, OpenGL, Structured light, 3D scanning, Camera calibration, Projector calibration,

Bibliografická citácia:

WALTER, V. *Projekce dat do scény*. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, 2016. 92s. Vedoucí diplomové práce byl Ing. Miloslav Richter, Ph.D

Prehlásenie

„Prehlasujem, že svoju diplomovú prácu na tému projekce dat do scény som vypracoval samostatne pod vedením vedúceho diplomovej práce a s použitím odbornej literatúry a ďalších informačných zdrojov ktoré sú všetky citované v práci a uvedené v zozname literatúry na konci práce. Ako autor uvedenej diplomovej práce ďalej prehlasujem, že v súvislosti s vytvorením tejto diplomovej práce som neporušil autorské práva tretích osôb, zvlášť som nezasiahol nedovoleným spôsobom do cudzích autorských práv osobnostných a som si plne vedomý následkov porušenia ustanovenia § 11 a nasledujúcich autorského zákona č. 121/2000 Sb., vrátane možných trestnoprávnych dôsledkov vyplývajúcich z ustanovení časti druhej, hlavy VI. diel 4 Trestného zákonníka č. 40/2009 Sb.

V Brne dne: **7. května 2016** ……

podpis autora

Poďakovanie

Ďakujem vedúcemu práce Ing. Miloslavovi Richterovi, Ph.D. za účinnú metodickú, pedagogickú a odbornú pomoc a ďalšie cenné rady pri spracovávaní mojej diplomovej práce a taktiež za poskytnutie technického vybavenia potrebného pre praktickú časť práce.

V Brne dne: **7. května 2016** ……

podpis autora

Obsah

1 ÚVOD

Zadanie pre túto prácu bolo:

- Zoznámiť sa s mechanizmom snímania dát reprezentujúcich tvar scény pomocou kamier
- Navrhnúť mechanizmus a algoritmy pre kvalitnú projekciu dát do 2D a 3D scény.
- Navrhnúť vhodnú demonštračnú úlohu alebo úlohy
- Realizovať navrhnuté algoritmy a aplikovať ich na navrhnuté úlohy
- Zhodnotiť možnosti a výsledné parametre riešení

V teoretickej časti sú vysvetlené pojmy týkajúce sa kalibrácie kamery a projektora a vytknuté sú tu najdôležitejšie veličiny s ktorými sa tu pracuje. Taktiež sú tu spísané niektoré výpočty, algoritmy a postupy ktoré sú použité v praktickej časti projektu. Je tu zahrnutý aj abstraktný popis demonštračných úloh, z ktorých niektoré sú v rámci praktickej časti aj implementované.
V praktickej časti tejto práce je popísaný program ktorý bol v rámci projektu vytvorený. Je tu obsiahnutý popis používania hotového programu, ako aj popis samotného zdrojového kódu na ktorom je program založený. Program implementuje dve z navrhnutých demonštračných úloh.
Praktická časť hovorí aj o procese testovania programu počas vývoja, a z takto nadobudnutých skúseností sú vyvodené návrhy pre voľbu budúcich krokov.

2 VLASTNOSTI KAMIER A PROJEKTOROV

Ak chceme vykonať kvalitnú projekciu dát do trojrozmernej scény, je potrebné najprv poznať vlastnosti použitých kamier a projektorov, pretože všetky prepočty vyplývajú z nich.

2.1 Kamery

Digitálnu kameru je možné opísať pomocou kombinácie extrinzických a intrinzických parametrov.

Extrinzické parametre zahŕňajú informácie o polohe a natočení kamery v danej scéne, zatiaľ čo intrinzické parametre hovoria o vnútorných vlastnostiach kamery. Tieto vnútorné vlastnosti zahŕňajú ohniskovú vzdialenosť kamery, ako aj polohu jej optického centra, pomer strán pixelov, poruchu pravouhlosti pixelov a skreslenie spôsobené šošovkami.

Intrinzické parametre sa zvyčajne popisujú pomocou takzvanej intrinzickej matice (camera matrix) a sady deformačných koeficientov (distortion coefficients).

Intrinzická matica popisuje kameru bez skreslení, podľa modelu dierkovej komory. Jej tvar je zvyčajne nasledovný:

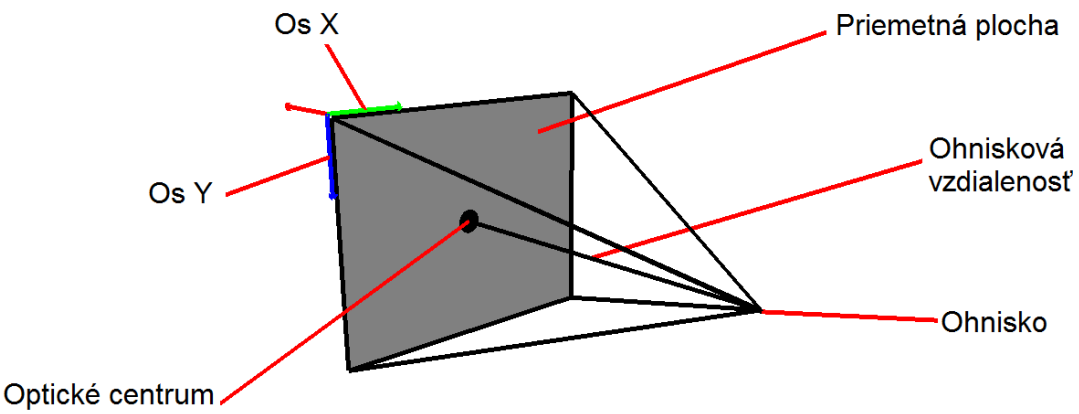
$$\begin{bmatrix} f_x & s & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{2.1}$$

Kde

- s [-] je zošikmenie
- f_x [pixel] je ohnisková vzdialenosť v šírkach pixelov po osi x
- f_y [pixel] je ohnisková vzdialenosť v šírkach pixelov po osi y
- c_x [pixel] je poloha optického centra v osi x
- c_x [pixel] je poloha optického centra v osi y

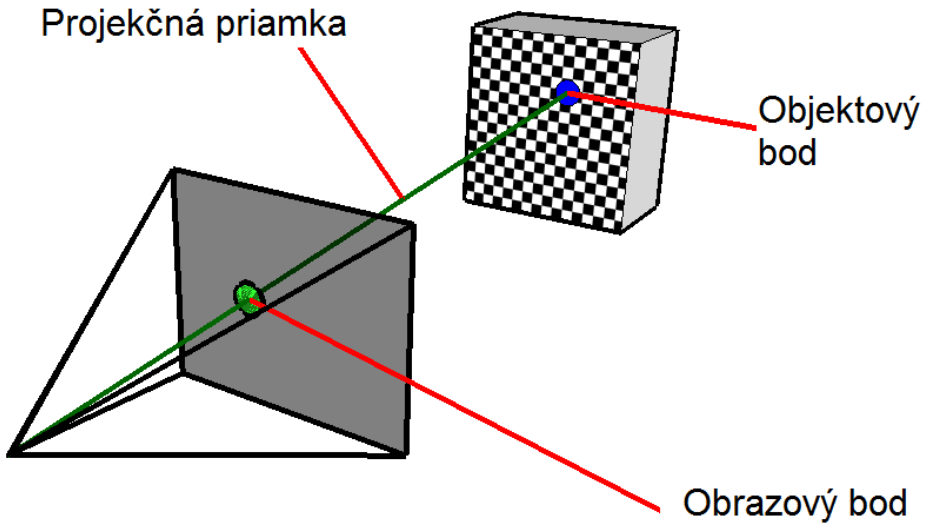
[1]

Jednotky v ktorých sú jednotlivé prvky uvádzané môžu byť aj iné, v tejto práci sú používané výlučne tieto.



Obrázok 2.1: Význam hlavných parametrov kamery modelovanej ako dierková komora

Pomocou tejto matice, v kombinácii s údajmi o polohe a natočení kamery, môžeme vypočítať polohu bodov na priemetni kamery, do ktorých sa premietnu body známej polohy v trojrozmerných koordinátach scény. Toto zanedbáva skreslenia objektívu, ktorými sa reálne kamery odlišujú od ideálnej dierkovej komory, pričom je ale možné tieto skreslenia odstrániť, ak poznáme deformačné koeficienty. [2]
Body v scéne nazývame objektové body a ich obrazy získané kamerou nazývame obrazovými bodmi.



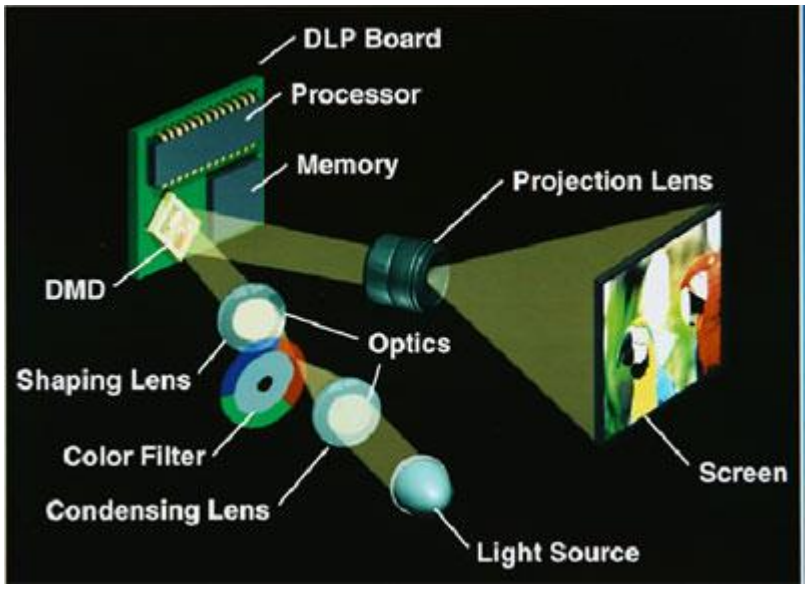
Obrázok 2.2: Vzťah objektového a obrazového bodu

2.2 Projektory

Na dátový projektor sa v tejto práci dívam ako na obrátenú kameru. Je to teda zariadenie, ktoré body z rovinného obrazu transformuje na projekčné priamky vychádzajúce z ohniska, teda lúče svetla ktoré následne dopadajú na plátno, alebo na iný povrch v scéne. Body dopadu lúčov považujem za objektové body, a body v obraze ktorého premietanie ich prináša do scény považujem za obrazové body. Podobne ako kamera, aj projektor má vlastné koeficienty skreslenia, keďže taktiež používa reálne šošovky.

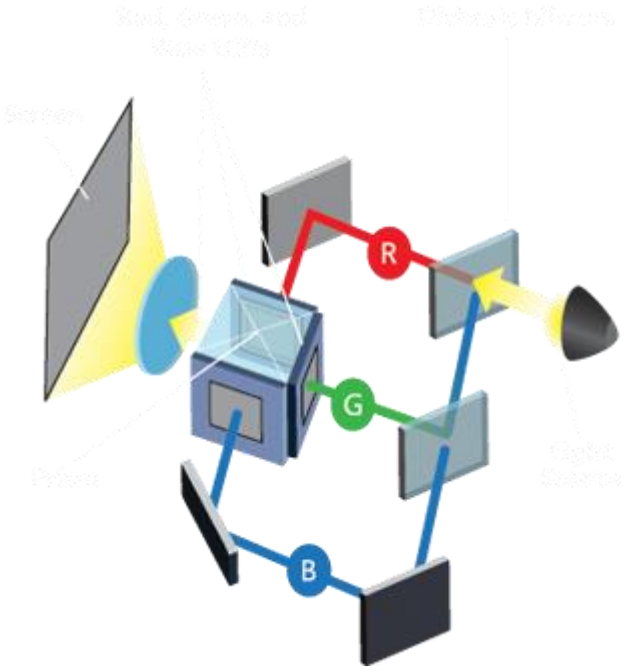
V rámci tejto práce budem využívať dva základné typy projektorov, a to DLP a 3LCD.

DLP projektor premieta farebný obraz pomocou sekvenčného striedania premietania troch a viac farebných kanálov. Kanály sa striedajú dostatočne rýchlo na to, aby sa ľudskému oku javil obraz ako farebný, hoci v skutočnosti je v každom okamihu monochromatický v červenej, modrej alebo zelenej farbe, ktorý predstavuje iba jednu zložku obrazu [3]. Toto ale neplatí pre prípad, ak je premietaný obraz snímaný kamerou, ktorá často nemá integračnú schopnosť ľudského oka, a teda zachytí v každej snímke obraz iba jedného z farebných kanálov.



Obrázok 2.3: Princíp fungovania DLP projektorov [3]

3LCD projektory spájajú lúče v troch farbách do jedného, pričom každý z lúčov predtým prechádza LCD panelom, a teda nesie obsah jedného z farebných kanálov obrazu. [4]



Obrázok 2.4: Princíp fungovania projektorov typu 3LCD [4]

Keďže sú v tomto prípade kanály prenášané paralelne, spomínaný problém pri použití kamery nenastáva.

3 TRANSFORMÁCIE

V práci budem používať viacero koordinát pre použitie pri transformáciách bodov medzi nimi. Pri všetkých systémoch súradníc budem predpokladať, že osi X, Y a Z sú na seba kolmé.

Koordináty kamery majú počiatok v ohnisku kamery a ich os Z je natočená od ohniska smerom k optickému centru na priemernej rovine kamery.

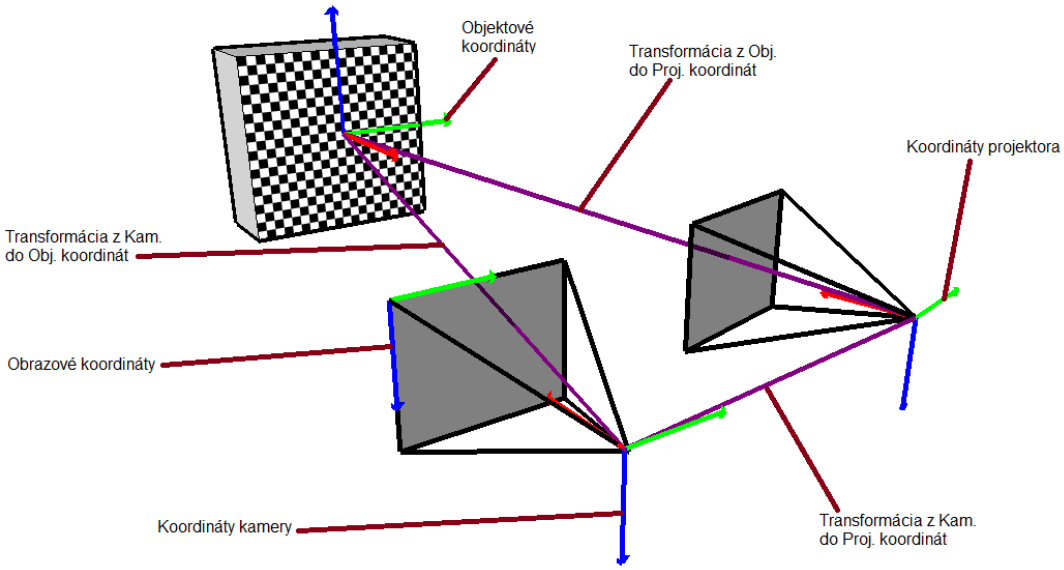
Koordináty projektora majú obdobne počiatok vo vnútornom ohnisku projektora a majú aj obdobný smer.

Objektové koordináty majú počiatok vo vybranom bode v objekte. V prípade fyzického šachovnicového vzoru budú mať počiatok v jeho strede, pričom je os Z kolmá na plochu šachovnice a osi X a Y sú rovnobežné so stranami štvorcov.

Obrazové koordináty kamery a projektora majú iba dve osi, a to x a y, pričom počiatok je v ľavom hornom rohu obrazu, a kladné smery osí sú doprava a nadol. Tieto koordináty budú použité v obraze z kamery a obraze zaslanom na premietanie do projektora.

Dôležité transformácie koordinát budú medzi kamerou a objektom, medzi objektom a projektorom a medzi kamerou a projektorom, ktoré sa dajú získať ako kombinácia prvých dvoch transformácii.

Transformácia medzi obrazovými koordinátami a koordinátami kamery spočíva v posunutí do optického centra, vynásobení osi Y pomerom f_y/f_x pre neštvorcové pixely a pridaní Z súradnice s konštantnou hodnotou ohniskovej vzdialenosti f_x. Nakoniec je potrebné všetky tri rozmery prenásobiť fyzickou šírkou pixelu.



Obrázok 3.1: Popis významných systémov koordinát a ich vzájomné transformácie

4 KALIBRÁCIA

Pre kvalitnú projekciu dát do priestorovej scény je potrebné kameru, ako aj projektor najprv skalibrovať.
Kalibrácia je činnosť, pri ktorej určujeme závislosť reálnej hodnoty meranej veličiny a hodnoty udanej prístrojom.
V tomto prípade pod kalibráciou budem rozumieť určovanie intrinzickej matice a koeficientov skreslenia pre kameru a projektor.
Fotometrická kalibrácia je činnosť taktiež niekedy nazývaná kalibráciou kamery. [5] Tá sa týka mapovania farieb zaznamenávaných kamerou. [6]
Toto pre účely tohto projektu nie je potrebné.

4.1 Kalibrácia kamery

Kameru je možné kalibrovať pomocou množiny známych obrazových bodov, ktoré zodpovedajú známym objektovým bodom v scéne. Pre jednoduchú kalibráciu kamery sa používajú ploché obrazy alebo priestorové objekty známych rozmerov, ktoré je možné jednoducho detekovať pri ich snímaní kamerou. [1]
Jednoduchým kalibračným vzorom je šachovnicový vzor so známou dĺžkou strán štvorcov. [7]
Ako objektové body je možné použiť rohové body štvorcov šachovnice, ktoré je možné pomerne jednoducho v obraze detekovať a taktiež je jednoduché popísať ich polohy v objektových koordinátach. Obrazové body budú definované ako miesta v obraze kde sú tieto rohové body zachytené kamerou.
Z týchto obrazových, a im príslušných objektových bodov môžeme pomocou známych algoritmov vyriešiť parametre kamery. [1]
Pre kvalitnú kalibráciu je potrebné zachytiť viaceré pohľady na šachovnicový vzor. Takto budú vzaté do úvahy rôzne vplyvy skreslenia, ktoré nemusia byť z jednej snímky zjavné. Jedna konkrétna projekcia objektových bodov na kameru môže byť vysvetlená rôznymi parametrami kamery, ale viaceré rôzne projekcie môžu viesť k jedinému správnemu riešeniu. [8]

4.2 Kalibrácia projektora

Podobne ako pri kalibrácii kamery použijeme dva sety bodov a šachovnicový vzor. Tento vzor bude premietaný projektorom na rovinu známej polohy a natočenia, a snímaný kamerou. Obrazové body budú v prípade projektora rohové body šachovnicového vzoru posielaného do projektora. Premietnutý šachovnicový vzor je možné detekovať podobne ako vzor na fyzickom kalibračnom vzore. Objektové body budú body na rovine do ktorých sa tieto body premietnu. [9]

Z tohto vyplýva, že musíme poznať polohu a natočenie roviny na ktorú vzor premietame voči kamere pri každej snímke. Preto je potrebné vždy zachytiť vedľa seba plochý, fyzický kalibračný vzor pripevnený na premietacie plátno a vzor premietaný na to isté plátno projektorom. Za predpokladu že plátno je ploché a parametre kamery sú už známe, môžeme z obrazu fyzického vzoru určiť polohu plátna. Potom pomocou intrinzickej matice kamery a bodov premietanej šachovnice ktoré sme na kamere taktiež zachytili, vieme určiť trojrozmerné polohy premietnutých bodov.
Výpočty podľa tohto postupu sú nasledovné:

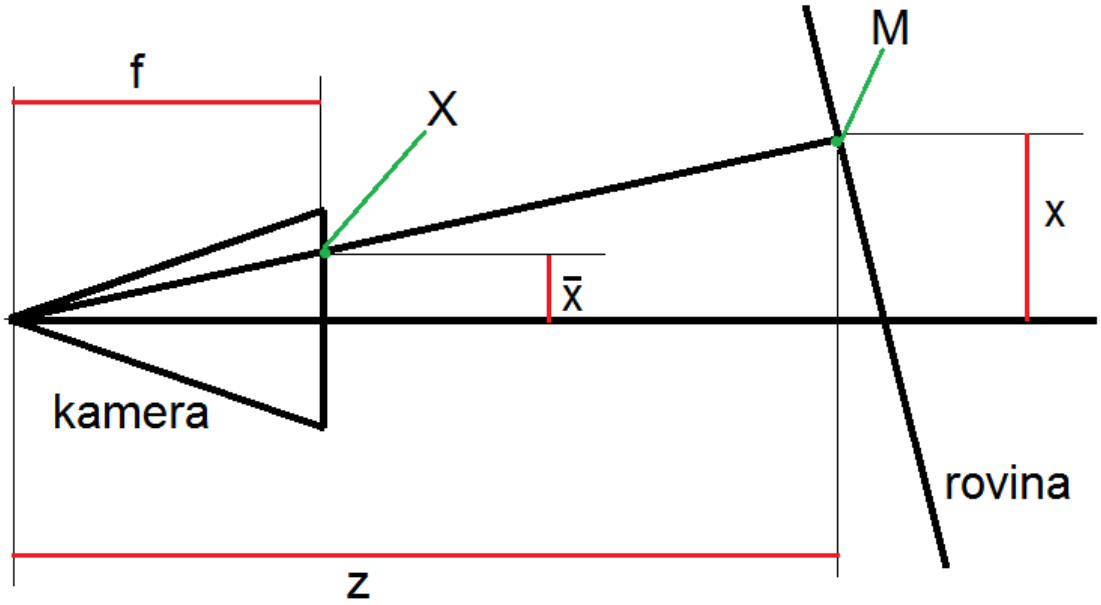
Vychádzame z intrinzickej matice. Potom:

$$f_x = \frac{f}{s_x} \; a \; f_y = \frac{f}{s_y} \tag{4.1}$$

Kde f je ohnisková vzdialenosť v ľubovoľných jednotkách a $s_x \; a \; s_y$ sú rozmery pixelov v týchto jednotkách. Odtiaľ pomer strán pixelov je:

$$\frac{s_y}{s_x} = \frac{f_x}{f_y} = r \tag{4.2}$$

Keďže z dôvodu podobnosti trojuholníkov tu nezáleží na reálnych rozmeroch, iba na ich vzájomných pomeroch, zvolím pre jednoduchosť $s_x = 1$.
Vytvorím si náčrt rozmerov ktoré budem ďalej používať:



Obrázok 4.1 Súradnice obrazu bodu na priemetnej rovine kamery a na fyzickej rovine

X je označenie bodu na priemetnej rovine kamery a M je označenie jeho ekvivalentu na fyzickej rovine. $\bar{x}$ je x-ová súradnica bodu X v obrazových koordinátach, posunutá o x-

ovú polohu optického centra. Potom x je x-ová súradnica bodu M v koordinátach kamery a z je z-ová súradnica tohto bodu. Pre os y je situácia podobná, ale tu budeme $\bar{y}$ násobiť r.
Za uvedených predpokladov sú súradnice bodu X v koordinátach kamery:

$$X = \begin{pmatrix} \bar{x} \\ \bar{y} * r \\ f \end{pmatrix} \tag{4.3}$$

Súradnice bodu M v týchto koordinátach budú označené ako:

$$M = \begin{pmatrix} x \\ x \\ z \end{pmatrix} \tag{4.4}$$

Z podobnosti trojuholníkov vyplýva, že:

$$M = X * t = \begin{pmatrix} \bar{x} * t \\ \bar{y} * r * t \\ f * t \end{pmatrix} \tag{4.5}$$

Kde t je konštanta ktorú sa snažíme získať.

Rovina je definovaná ako množina bodov $\hat{p}$ takých, že:

$$(\hat{p} - \widehat{p_0}) * \hat{n} = 0 \qquad \text{[10](4.6)}$$
$$\hat{p} * \hat{n} = \widehat{p_0} * \hat{n}$$

Kde

- $\widehat{p_0}$ je konkrétny bod na rovine
- $\hat{n}$ j normála roviny

Normálu a bod na rovine môžeme určiť pomocou známej translácie T a rotácie R fyzickej šachovnice ktorú sme zachytili, a ktorá by mala ležať na fyzickej premietacej rovine:

$$\hat{n} = \begin{pmatrix} n_x \\ n_y \\ n_z \end{pmatrix} = R * \hat{n}_I = R * \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \tag{4.7}$$
$$\widehat{p_0} = \begin{pmatrix} x_0 \\ y_0 \\ z_0 \end{pmatrix} = T * \hat{p}_I = T * \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

Do definície roviny dosadíme za $\hat{p}$ bod M.

$$n_x * x + n_y * y + n_z * z = n_x * x_0 + n_y * y_0 + n_z * z_0 \tag{4.8}$$
$$n_x * \bar{x} * t + n_y * \bar{y} * r * t + n_z * f * t = n_x * x_0 + n_y * y_0 + n_z * z_0 \tag{4.9}$$
$$n_x * x_0 + n_y * y_0 + n_z * z_0 = d \tag{4.10}$$

Pričom vieme že d je konštanta.
Potom vieme určiť konštantu t ako:

$$t = \frac{d}{n_x * \bar{x} + n_y * \bar{y} * r + n_z * f} \tag{4.11}$$

Súradnice bodu M v koordinátach kamery vieme potom určiť už zmieneným vzťahom (4.5).

Takto vypočítame polohy všetkých rohových bodov premietaného vzoru na fyzickej rovine.

Následne môžeme vyriešiť parametre projektora podobným algoritmom ako parametre kamery.

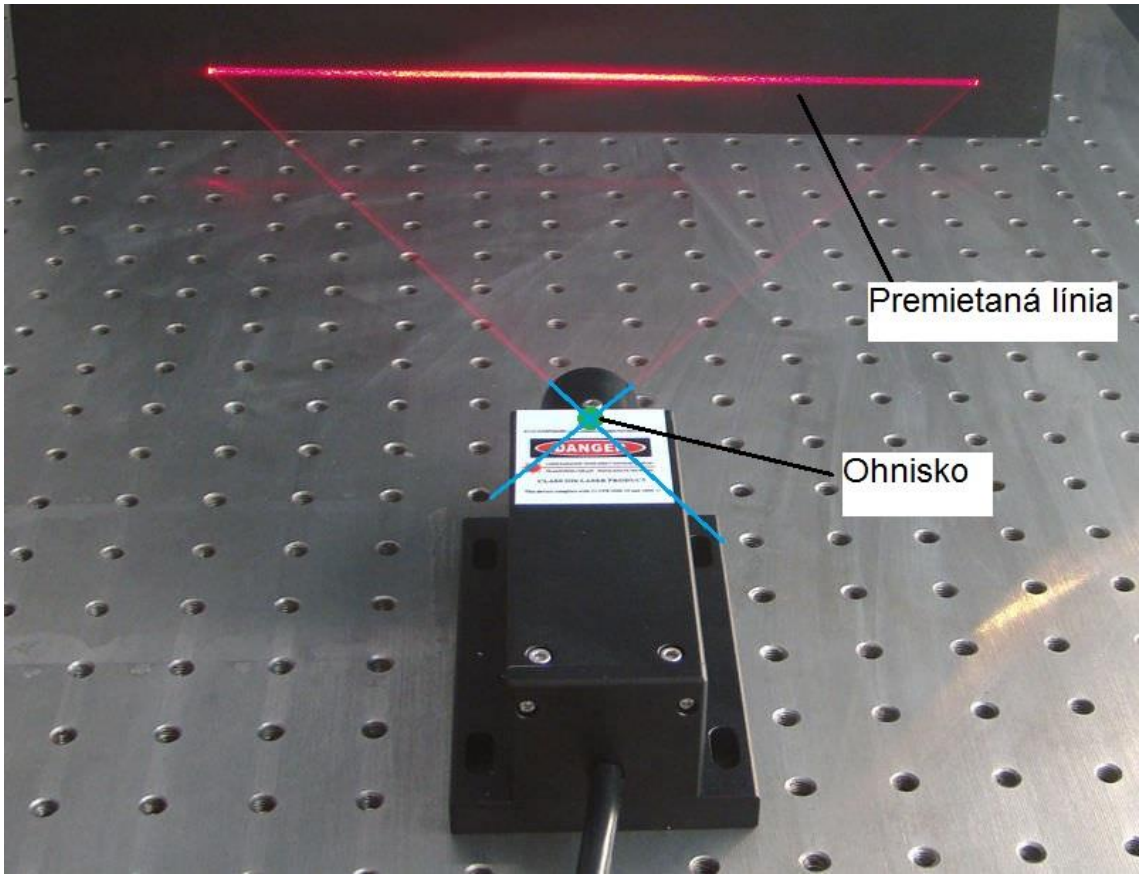
4.3 Vzájomná poloha kamery a projektora

Okrem intrinzických parametrov kamery a projektora je potrebné taktiež určiť vzájomnú polohu kamery a projektora. Toto je možné dosiahnuť podobným postupom ako pri kalibrácii projektora. Pri zachytení premietnutého, ako aj fyzického vzoru kamerou určíme najprv polohu fyzického vzoru. Toto je pri kamere so známymi parametrami dosiahnuteľné rovnako ako v predchádzajúcom kroku. Potom vypočítame trojrozmernú polohu premietnutých bodov a z nich, určíme polohu projektora. Keďže fyzické polohy premietnutých bodov sú vypočítané v koordinátach kamery, bude aj poloha a natočenie projektora v týchto koordinátach.

Toto je veľmi výhodné, pretože takto poznáme priamo vzájomnú polohu kamery a projektora bez nutnosti skladať transformácie.

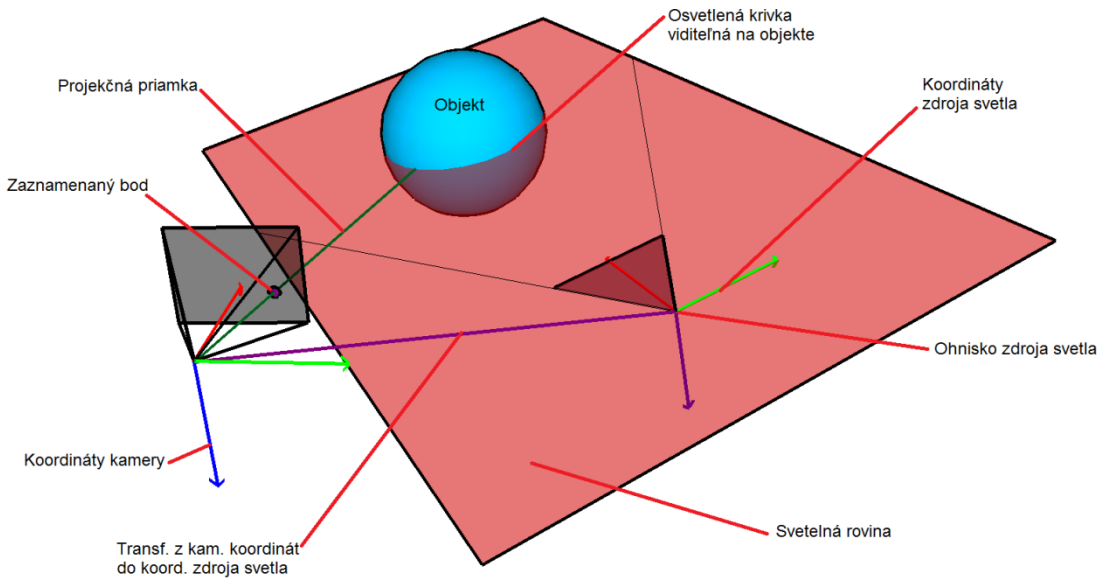
5 REKONŠTRUKCIA TVARU OBJEKTU ŠTRUKTÚROVANÝM SVETLOM

Premietnutím čiary svetla na priestorový objekt vznikne na povrchu tohto objektu osvetlená krivka, ktorá popisuje hranice rezu tohto objektu rovinou. Táto rovina je tá, na ktorej leží ohnisko zdroja, z ktorého všetky lúče vychádzajú, a na ktorej tiež leží priamka ktorá vznikne premietnutím tohto svetla na rovný povrch. Tieto sú ilustrované na Obrázok 5.1. Túto rovinu budem pre účely mojej práce nazývať svetelnou rovinou.



Obrázok 5.1: Laserová plocha s vyznačeným ohniskom a líniou na rovnej doske [11]

Ak čiaru premietnutú na objekt sledujeme kamerou so známymi intrinzickými parametrami, a so známou polohou voči zmienenému zdroju svetla, môžeme z jej obrazu určiť polohu povrchových bodov objektu, ktoré ležia pozdĺž svetelnej čiary.
K tomuto možno pristupovať tak, že vypočítame priesečníky svetelnej roviny s projekčnými priamkami zodpovedajúcimi polohám obrazových bodov zakrivenej línie na kamere. Polohu svetelnej roviny určíme z polohy kamery voči zdroju svetla a projekčné priamky určíme z obrazových koordinát bodov krivky na snímke a z intrinzických parametrov kamery. Priesečník roviny a priamky vypočítavame rovnako ako je popísané v časti 4.2.



Obrázok 5.2: Popis vzťahu svetelnej roviny k obrazu osvetlenia objektu na kamere

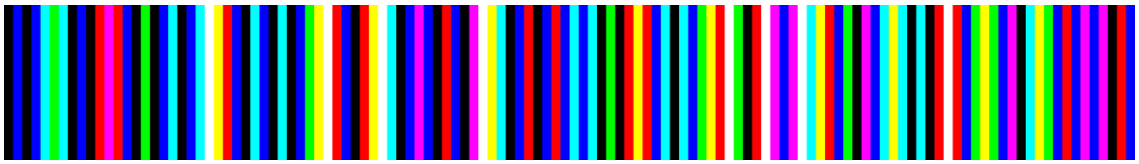
Takto získame sériu trojrozmerných bodov povrchu snímaného objektu.
Ak ale chceme získať približný tvar plochy povrchu telesa, nepostačí nám iba jedna čiara.
Používaný postup je napríklad taký, že zdroj svetelnej roviny posúvame v priestore [12] alebo je možné napríklad otáčať skenovaný objekt [13].

Tieto postupy ale majú zásadnú nevýhodu v tom, že je potrebné získať viacero snímok v čase za sebou, a vyžadujú mechanický pohyb.
Aby sme získali trojrozmerný tvar z jednej snímky, potrebujeme viacero individuálne rozlíšiteľných svetelných rovín. Na toto jednofarebný laser – ako na Obrázok 5.1 – nepostačuje. Viacero rovnakých línii na obrázku nevieme jednoznačne individuálne odlíšiť, takže by sme nevedeli s ktorou svetelnou rovinou počítať priesečník. Riešením je využitie farebného dátového projektora.

5.1 Identifikácia pásov s použitím De Bruijnovych vzorov

Existuje viacero metód ako pomocou premietaného farebného vzoru rekonštruovať tvar objektu [14] [15] [16]. V tejto práci som implementoval metódu, odvodenú z práce „Rapid Shape Acquisition Using Color Structured Light and Multi-pass Dynamic Programming“ ktorú napísali Li Zhang, Brian Curless a Seven M. Seitz [17]. Tento dokument popisuje algoritmus, s ktorým je možné v určitých prípadoch identifikovať hranice premietaných farebných pásov v obraze ako prvky s určeným poradím v danej sekvencii.

Dátový projektor premieta v troch farebných kanáloch R, G a B. V týchto kanáloch zvyčajne aj snímajú kamery. Keďže potrebujeme aby jednotlivé farebné pásy boli čo najlepšie rozlíšiteľné, budeme ich voliť vo farbách, ktoré majú jednotlivé kanály buď v maximálnej, alebo v nulovej intenzite. Budeme teda používať pásy vo farbách čierna, biela, modrá, zelená, červená, žltá, zelenomodrá a magenta. Farebné prechody na rozhraniach týchto pásov môžu teda byť siedmych typov, z hľadiska zmeny jednotlivých kanálov. Môžeme na rozhraní zmeniť napríklad iba kanál R, iba G, kanály G aj B a podobne. Zo sekvencie boli vylúčené dve možnosti kde sa zmení červený aj zelený kanál zároveň, pretože podľa Zhang a kol. tieto prechody trpia zvýšeným tzv. crosstalkom, teda jeden kanál nevhodne zasahuje detekciu druhého.
Aby sme dosiahli vysokú pravdepodobnosť že daný prechod správne identifikujeme v obraze kde rovnakých môže byť viac, musíme zabezpečiť, že prechody budú zoradené tak aby celý set obsahoval každú n-ticu možných prechodov za sebou len jeden krát, čím sa každá n-tica prechodov v obraze stane za ideálnych podmienok jedinečnou.
Takúto vlastnosť majú sekvencie, ktoré nazývame De Bruijnove [18].
Keďže máme päť prechodov, budem používať rovnako ako Zhang a kol. De Bruijnovu sekvenciu s jedinečnými trojicami prechodov za sebou. Táto sekvencia má dĺžku 125 prvkov hodnôt 1 až 5, ktoré reprezentujú 5 povolených farebných prechodov. Táto sekvencia je známa, a získal som ju pomocou algoritmu na stránkach Håkana Kjellerstranda [19].
Farebné pásy z nej odvodím tak, že si zvolím počiatočnú farbu, napríklad čiernu, pre prvý pás a každý ďalší pás bude mať farbu, ktorú získam zmenou farby predchádzajúceho pásu podľa príslušného prvku De Bruijnovej sekvencie. Touto zmenou sa myslí, že ak napríklad príslušný prvok sekvencie je 1, ktorému prísluší zmena kanálu B a žiadna zmena pre G a R, potom ak B kanál predchádzajúceho pásu je 0, zmením ho na maximálnu intenzitu a ak je v maximálnej intenzite, zmením ho na 0. Zvyšné dva kanály nemením.

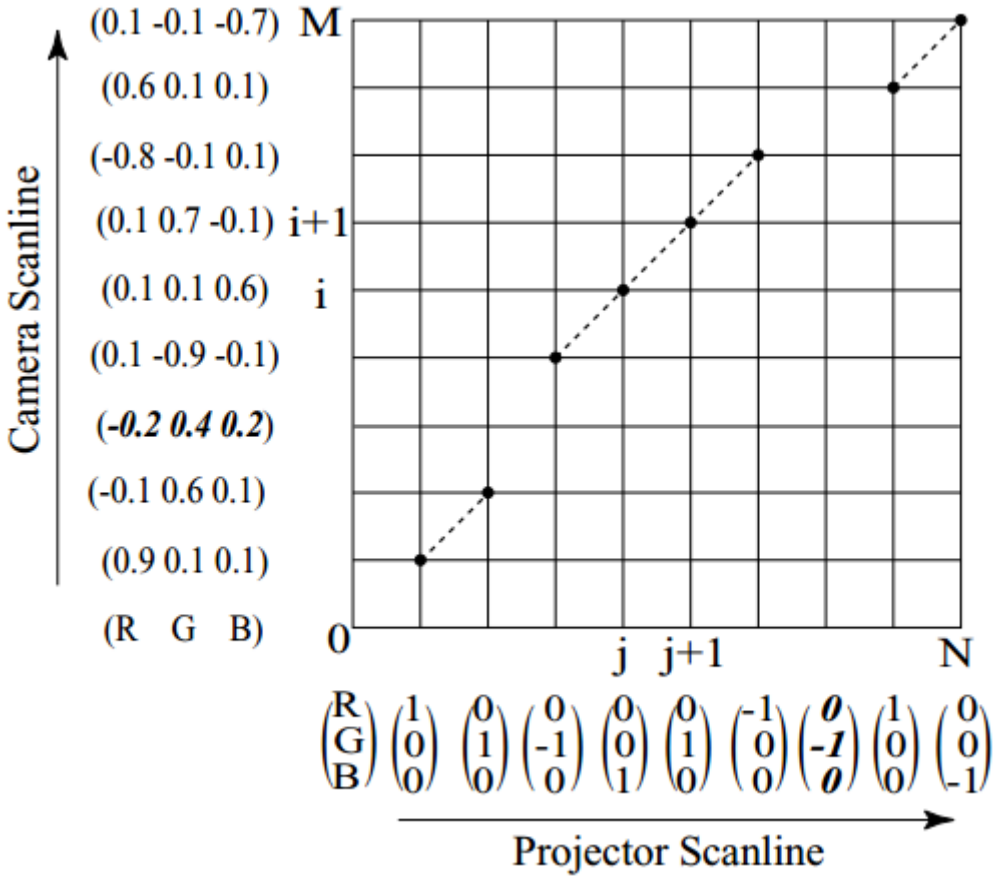


Obrázok 5.3: Farebné pásy získané pomocou de Bruijnovej sekvencie

Takto si vypočítam celú sekvenciu farieb. Túto premietnem projektorom na objekt a snímam ju kamerou.
Na obraze potom pozdĺž skenovacej línie približne kolmej na farebné pásy – v mojom prípade som zvolil vodorovné riadky obrazu – hľadám hranice pásov ako lokálne maximá obrazu súčtu štvorcov zmien jednotlivých kanálov medzi dvoma susednými pixelmi. Po skenovacej línii dostanem takto M hraničných bodov.
Pre ďalší krok budem predpokladať monotónnosť prechodov v obraze – nemusia byť viditeľné všetky prechody a môžu byť zachytené aj falošné hranice, no nemali by tu byť

skupiny prechodov, ktoré podľa premietaného poradia nepatria medzi prechody ktoré ich obklopujú. Takáto chyba môže nastať, ak je napríklad pred snímaným objektom ešte jeden osvetlený objekt, ktorého pásy sa kvôli paralaxe v obraze objavia mimo očakávanej sekvencie.

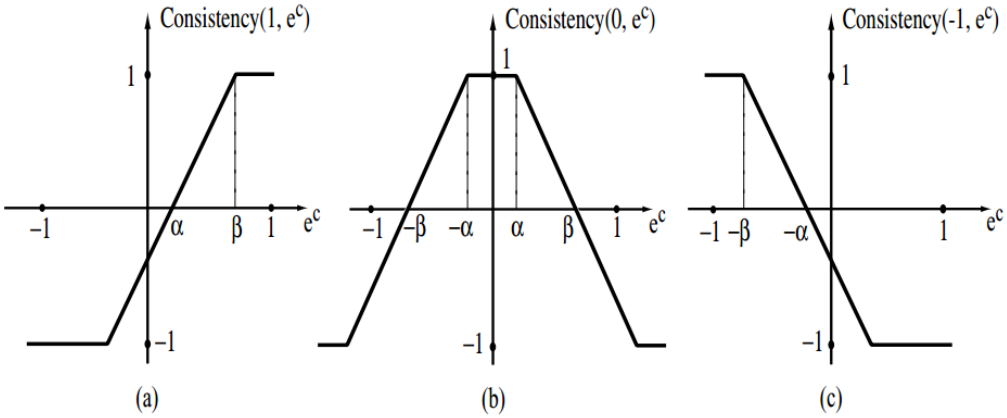
Vytvorím si mriežku, kde na jednej osi bude sekvencia N prechodov ako ich premietam a na druhej bude sekvencia M hraničných bodov ako ich snímam. Potom možno za predpokladu monotónnosti očakávať, že cesta spájajúca premietané a zachytené prechody ktoré si zodpovedajú bude monotónne stúpajúca, pravdepodobne prerušovaná cesta. Prerušenia nastávajú ak je na obraze prechod vynechaný, alebo je detekovaný neexistujúci, ako je ilustrované na Obrázok 5.4.



Obrázok 5.4: Mriežka na ktorej hľadáme optimálnu monotónnu cestu [17]

Túto cestu môžem v mriežke hľadať s využitím známych a očakávaných farebných prechodov. Každému bodu v mriežke priradím hodnotu, určujúcu kvalitu zhody. Túto hodnotu vypočítam ako najmenšiu z troch hodnôt funkcie konzistencie pre jednotlivé kanály. Funkcia konzistencie je popísaná jedným z troch grafov na Obrázok 5.5, pričom e^C je zmena hodnoty jedného farebného kanálu na nasnímanom prechode. Obe vstupné hodnoty funkcie konzistencie sú normalizované tak, aby maximálne možné zmeny intenzity boli -1 a 1, pričom 0 znamená že na prechode sa kanál nezmenil. Funkcia má

dva parametre – α a β – ktorými môžeme regulovať striktnosť požiadaviek na zhody. Širší rozstup týchto hodnôt umožňuje nájsť sekvenciu aj pri menej kvalitnom obraze, ale prináša možnosť nálezu neexistujúcich zhôd.



Obrázok 5.5: Funkcia konzistencie pre tri možné hodnoty očakávanej zmeny jasu

Cez takto vyplnenú mriežku hľadám od posledného bodu pre zaznamenané a premietané prechody – M a N na obr Obrázok 5.4 – cestu najvyššej celkovej konzistencie. V každom bode sa môžem pohybovať tromi spôsobmi (podľa obrázka):

- Diagonálne smerom k nižším prechodom – prechody si zodpovedajú
- Nadol – bol detekovaný neexistujúci prechod
- Vľavo – premietaný prechod v obraze chýba

Optimálnu cestu hľadáme rekurzívne, výpočtom hodnôt celkového skóre každého bodu [i,j] v mriežke. Počnúc bodom [M, N] vyberám ako skóre bodu vždy najvyššiu z troch hodnôt:

- Skóre bodu [i-1,j-1] + konzistencia bodu [i,j] (prípad zhody)
- Skóre bodu [i-1,j]
- Skóre bodu [i,j-1]

Pre i=0 alebo j=0 je skóre rovné 0.

Ako je zjavné, skóre pre [i,j] je funkcia skóre bodov [i-1,j-1] , [i-1,j] a [i,j-1]. Tento rekurzívny algoritmus som oproti Zhang a kol. optimalizoval dvoma spôsobmi:

- Hodnoty konzistencie som dopredu zadal do mriežky MxN, aby ich nebolo vždy nutné vypočítavať
- Ak už bolo pre určitý bod skóre vypočítané, potom ak je znova požadovaná hodnota tohto skóre, nebudem ju vypočítavať, ale ju načítam zo samostatnej mriežky pre skóre. V opačnom prípade by totiž boli volania funkcie skóre výrazne početnejšie. Napríklad pre oba body [5,5] a [4,5] je potrebné určiť skóre bodu [4,4].

V takto prehľadanej mriežke potom považujem za príslušnosť (korešpondenciu) prechodov i a j tie páry, pri ktorých bolo skóre určené prípadom zhody (vyššie).
Tieto korešpondencie znamenajú, že nájdený prechod i považujem za obraz premietaného prechodu j, a teda prechod i leží fyzicky na svetelnej rovine patriacej prechodu j.

Zhang a spol. popisujú aj úpravu algoritmu tak, aby mohla byť porušená monotónnosť. Tento postup je založený na opakovanom hľadaní optimálnej cesty vyššie zmienenou mriežkou, pričom sú riadky i a stĺpce j prislúchajúce nájdenej korešpondencii vypustené z mriežky pre ďalšiu iteráciu. Takto sa algoritmus ale stáva pomalším a zložitejším. Pre moje účely toto nepotrebujem, pretože budem skenovať len jednoduché predmety.
Taktiež sa v dokumente popisuje možnosť zlepšiť kvalitu rekonštrukcie s použitím viacerých snímok, na ktorých sú obrazy objektu s premietanými pásmi ktoré sú postupne posúvané. Toto by taktiež algoritmus spomalilo a zvýšilo jeho komplikovanosť. V oboch prípadoch pomalý algoritmus taktiež zabraňuje použitiu v dynamickom skenovaní (časť 6.2.2), pri použití viacerých snímok s časovým posunom vzoru je porušená aj podmienka, že tvar musí byť získaný z jednej snímky.

Jeden pôsob ako by sa dal potlačiť vplyv farebnosti snímaného povrchu na zachytené farby pásov, je najprv osvetliť objekt bielym svetlom, a následne z takejto snímky vyvodiť farebnú korekciu obrazu s De Bruijnovymi pásmi. Toto by potlačilo aj vplyv slabšieho osvetlenia častí objektu, ale taktiež to popiera podmienku rekonštrukcie z jednej snímky.

5.2 Rekonštrukcia tvaru zo získaných bodov.

Ak sa nám podarilo určiť ktorému z premietaných prechodov nájdený prechodový bod prislúcha, potom vieme určiť správnu svetelnú rovinu na ktorej leží. Hľadaním priesečníkov projekčných priamok prechádzajúcich obrazmi zaznamenaných bodov so svetelnou rovinou ktorá im náleží dokážeme jednoznačne určiť polohu bodu na povrchu objektu v priestore. Toto musíme vykonať pre každý prechodový bod pre ktorý sme príslušný prechod našli, a to v každom skenovacom riadku obrazu ktorý sme použili.
V mojom programe nepoužívam každý riadok obrazu z kamery, ale vynechávam vždy počet riadkov rovný polovici šírky premietaného pásu v pixeloch. Dôvod je zrýchlenie programu. Takto získavam ako výstup model, ktorý nemá v jednom rozmere výrazne vyššiu mieru detailnosti ako v druhom.

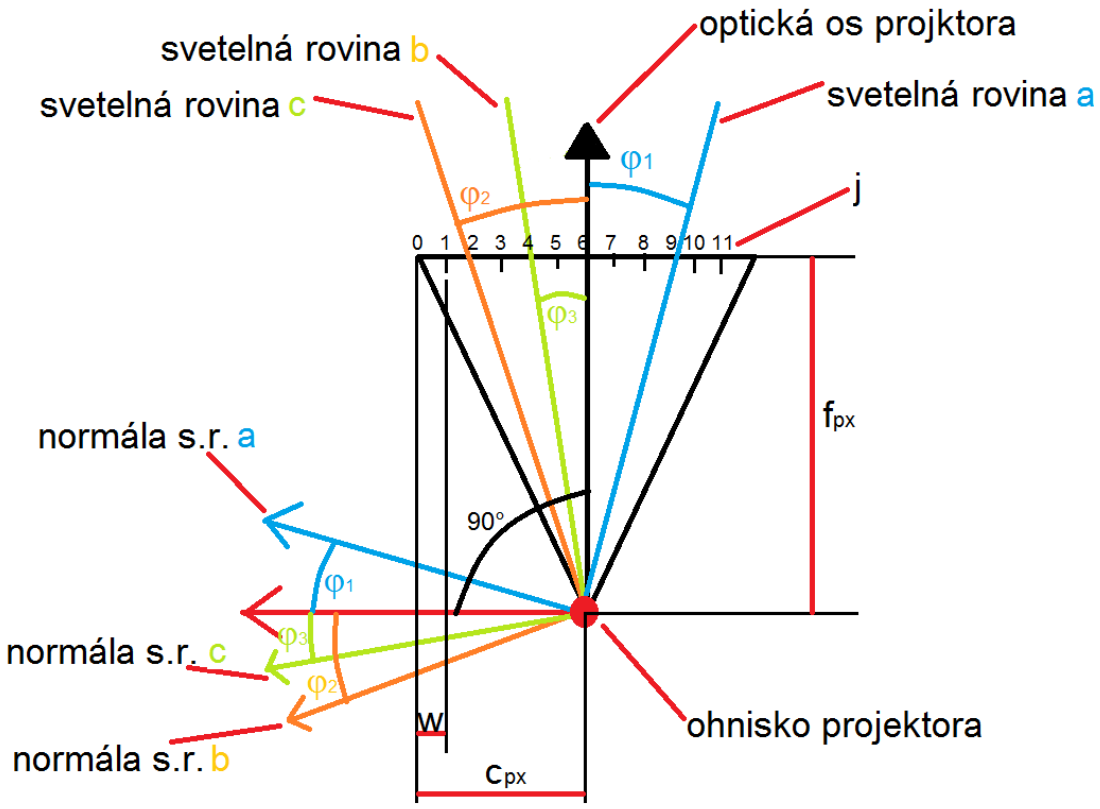
Polohu a natočenie jednotlivých svetelných rovín definujem pre funkciu výpočtu priesečníkov pomocou bodu ktorý na nich leží a ich normály. Bod ktorý na svetelnej rovine určite leží je ohnisko projektora, ktorého polohu voči kamere viem určiť ako posuv projektora voči kamere. Smer normály určíme tak, že jednotkový vektor natočený v smere optickej osi projektora – tento smer poznáme ako natočenie projektora voči

kamere – pootočíme o 90° okolo jeho zvislej osi, a potom ho pootočíme ešte o uhol φ, ktorý získam pomocou vzorca:

$$\varphi = \arctan\left(\frac{j * w - c_{PX}}{f_{PX}}\right) \tag{5.1}$$

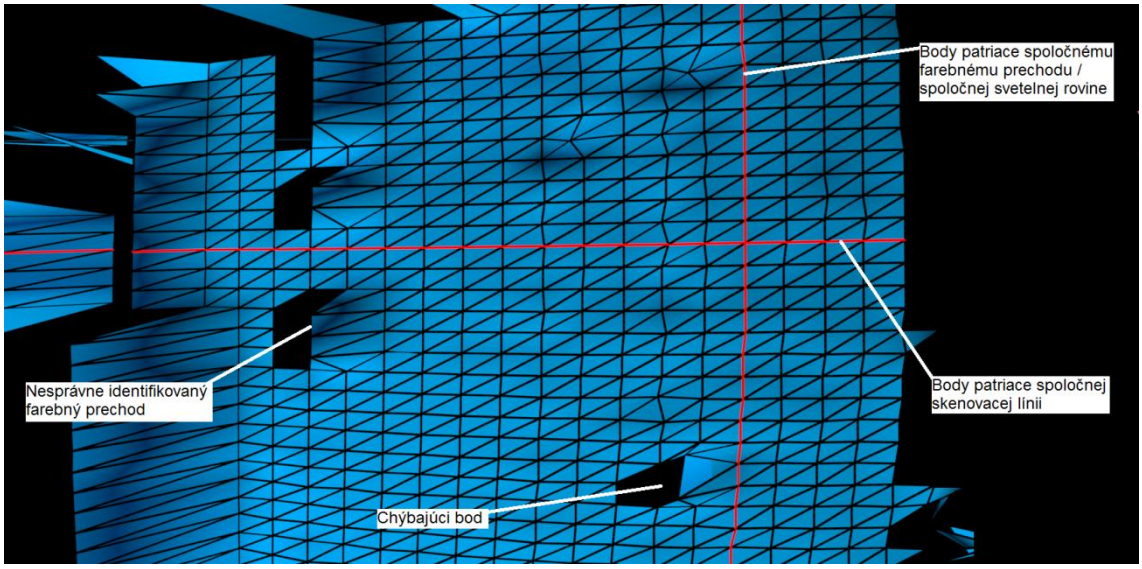
Kde

- j [-] je poradové číslo identifikovaného farebného prechodu
- w [pixel] je šírka jedného premietaného farebného pásu
- c_{px} [pixel] je vodorovná súradnica optického centra projektora
- f_{px} [pixel] je ohnisková vzdialenosť projektora v šírkach pixelov po osi x



Obrázok 5.6: Popis určovania normály svetelnej roviny

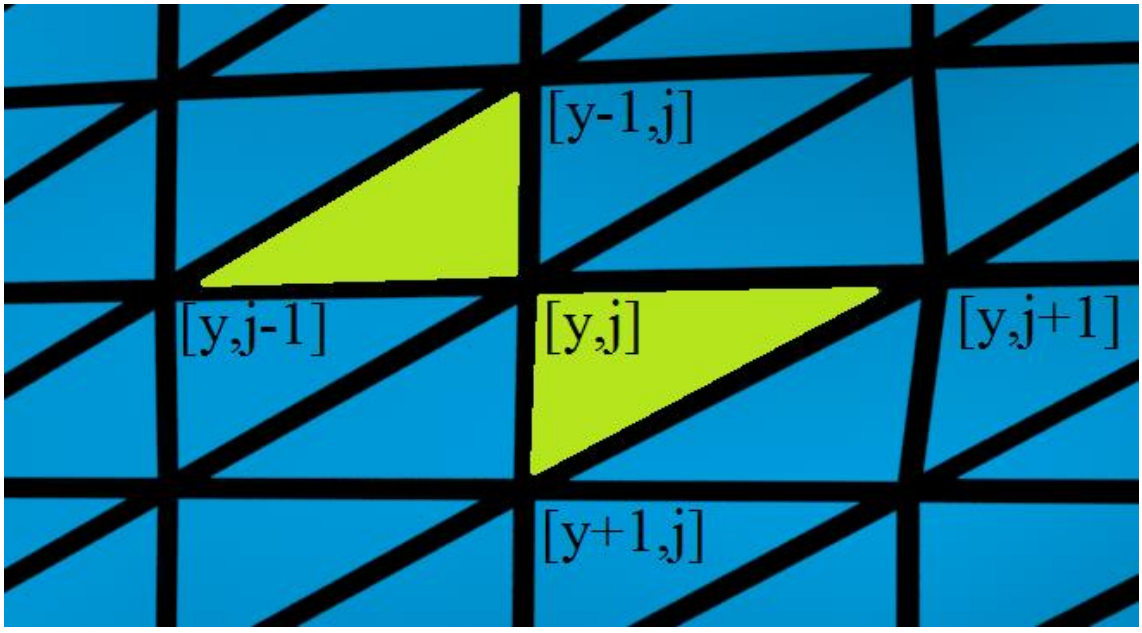
Po lokalizovaní bodov na povrchu objektu vo viacerých líniách môžem zostaviť model povrchu.
Pre jednoduchosť a rýchlosť algoritmu je model zostavený ako deformovaná štvorcová mriežka. V tejto mriežke sú na spoločných stĺpcoch body ktoré boli identifikované na spoločnom farebnom prechode – svetelnej ploche – a v spoločných riadkoch sú body nájdené na spoločnej skenovacej línii – viď Obrázok 5.7.



Obrázok 5.7: Mriežka rekonštruujúca skenovaný objekt na Obrázok 7.24

Táto mriežka tvorí model zložený v skutočnosti z trojuholníkov, pričom pre každý nájdený bod [y,j] – patrí skenovacej línii y a svetelnej rovine j – sa snažím zostaviť dva trojuholníky:

- trojuholník ktorého vrcholy sú aktuálny bod, a ak boli v obraze detekované, body [y-1,j] a [y,j-1]
- trojuholník ktorého vrcholy sú aktuálny bod, a ak boli v obraze detekované, body [y+1,j] a [y,j+1]



Obrázok 5.8: Tvorba trojuholníkov v mriežke rekonštrukcie

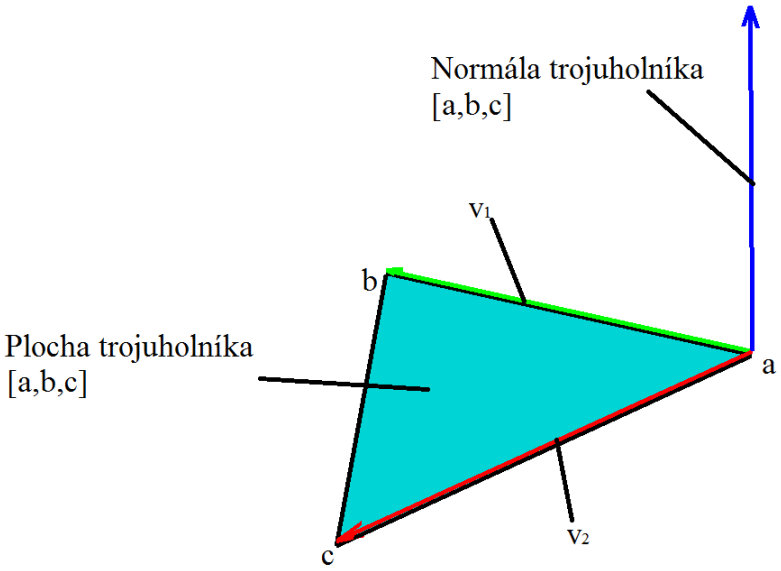
Napríklad ak sú nájdené štyri body s [y,j] = [1,1], [1,2], [2,1] a [2,2], zostavia štvoruholník, zložený z dvoch trojuholníkov, rozdelených diagonálou medzi bodmi a [1,2], [2,1].
Keďže nevytváram záplaty priamo zo štvoruholníkov medzi [y,j], [y+1,j], [y,j+1] a [y+1,j+1], nespôsobí strata jedného z bodov [y,j] a [y+1,j+1] stratu celého štvoruholníka, ale len jedného z dvoch trojuholníkov z ktorých sa tu skladá.

Výhodou takejto jednoduchej mriežky je, že umožňuje jednoduchú vizuálnu analýzu chýb detekcie, ako na Obrázok 5.7. Okrem toho je tento postup rekonštrukcie povrchu zo známych bodov výpočtovo rýchlejší než tvorba siete na báze hľadania najbližších susedov a je odolnejší outlierom než napríklad generovanie konvexnej obálky nájdených bodov.

Pre mnohé spôsoby vykresľovania v 3D grafike je potrebná znalosť normálových vektorov povrchu. Jednou z často používaných aplikácii je napríklad v tieňovaní, kde vzájomný uhol normály, kamery a zdroja osvetlenia určí, ako bude daný bod povrchu z pohľad kamery osvetlený [20].
V mojom programe sa ale priamo tieňovaním nezaoberám, pretože fyzický objekt na ktorý premietneme textúru má vlastné tieňovanie vyplývajúce z fyzického sveta, a pridávanie ďalších takýchto efektov by neprinieslo závažné zlepšenie využiteľnosti ani dojmu, zvlášť keďže ak sa objekt nachádza v premietacom poli projektora, budú aj oblasti na ktoré premietame „čiernu“ farbu mierne osvetlené.
Normálové vektory ale využívam pri aplikácii kombinácie textúr v závislosti od sklonu povrchu, ako to popisujem v častiach 6.2.1 a 7.4.6.
Normály vypočítavam pre jednotlivé vrcholy. Vychádzam pritom z predpokladu, že normála jedného samostatného trojuholníka má smer rovný vektorovému súčinu vektorov v_1 a v_2, ktoré vzniknú z rozdielov dvoch dvojíc vrcholov a,b,c:

$$v_1 = b - a; \quad v_2 = c - a \tag{5.2}$$

$$n' = v_1 \times v_2 \tag{5.3}$$



Obrázok 5.9: Popis určovania normály trojuholníka

Bežne je ďalším krokom normalizovať vektor n’ na jednotkovú veľkosť. Keďže ale v sieti trojuholníkov často jednému vrcholu náleží viac ako jeden trojuholník, budem jeho normálu počítať ako vážený priemer normál týchto trojuholníkov. Predpokladám tu, že väčšie trojuholníky majú väčší vplyv na smer normály než menšie, takže môžem veľkosť vektoru n’ – závislú na veľkosti trojuholníka – použiť ako túto váhu. Potom získam normálu vrcholu ako normalizáciu priemeru jednotlivých vektorov n’ od každého z okolitých trojuholníkov na jednotkovú veľkosť [21].

6 NAVRHOVANÉ DEMONŠTRAČNÉ ÚLOHY

6.1 Sledovanie premietacej dosky

Jednoduchá úloha, ktorou je možné demonštrovať spoluprácu kamery a projektora, a taktiež experimentálne overiť správnosť kalibrácie, je sledovanie pohyblivej premietacej dosky. Táto doska by mala mať identifikovateľné body, ktorých vzájomné polohy sú známe. Pomocou nich je možné odhadnúť polohu a natočenie tejto plochy voči kamere. So znalosťou vzájomnej polohy kamery a projektora vieme potom určiť, kde sa táto premietacia plocha nachádza v koordinátach projektora.

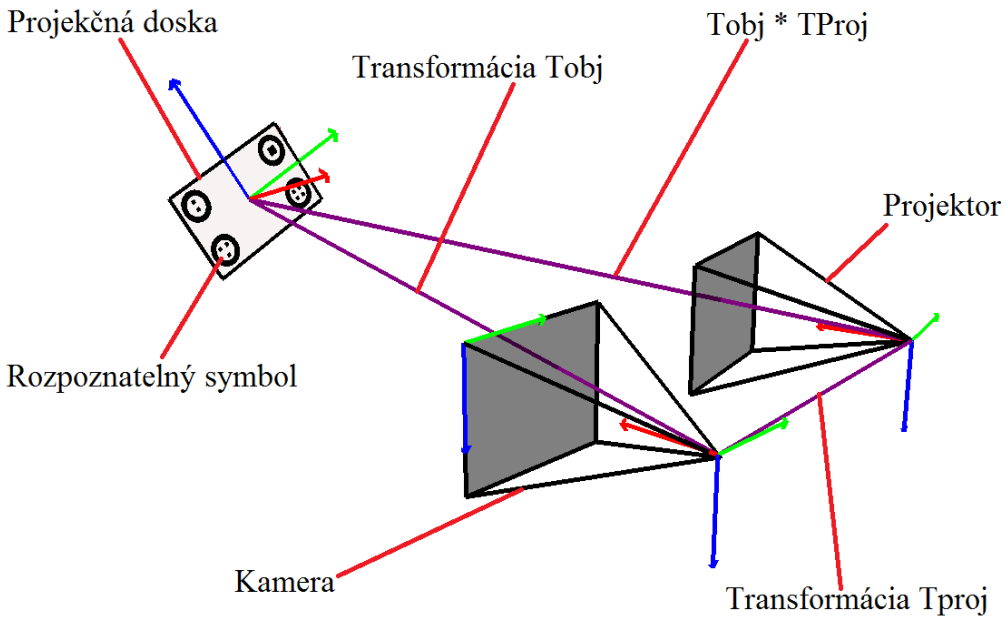
Potom môžem vykresliť v týchto koordinátach a v rovnakej polohe voči virtuánej kamere plochú textúru, napríklad pomocou OpenGL. Virtuálna kamera musí mať rovnaké, alebo dostatočne podobné parametre ako projektor. Obraz s takto vykreslenou textúrou potom môžem zaslať do projektora. Výsledkom je, že projektor bude premietať danú textúru na plochu pohyblivej dosky tak, ako by bola pevne nakreslená na doske.

Doska sa takto bude javiť ako virtuálna obrazovka.

V takejto aplikácii je nutné vziať do úvahy, že doska musí byť viditeľná z pohľadu kamery a zároveň sa musí nachádzať v premietacom poli projektora. Taktiež je potrebné, aby normála dosky nebola príliš odklonená od kamery, čo by zhoršilo schopnosť kamery detekovať jej polohu, a nesmie byť ani príliš odklonená od projektora, pretože v dôsledku konečného rozlíšenia projektora by mala premietaná textúra na doske v niektorých smeroch malé rozlíšenie.

Tieto problémy je teoreticky možné riešiť použitím viacerých kamier a projektorov ktoré by sa striedali, alebo dopĺňali. Toto je ale náročné ako finančne, tak aj z hľadiska programovania, a v tejto práci sa touto možnosťou nezaoberám.

Pri použití jednej kamery a jedného projektora je výhodné umiestniť ich blízko seba a natočiť ich do podobného smeru. Takto je jednoduchšie udržať plochu v dosahu ako kamery, tak aj projektora.



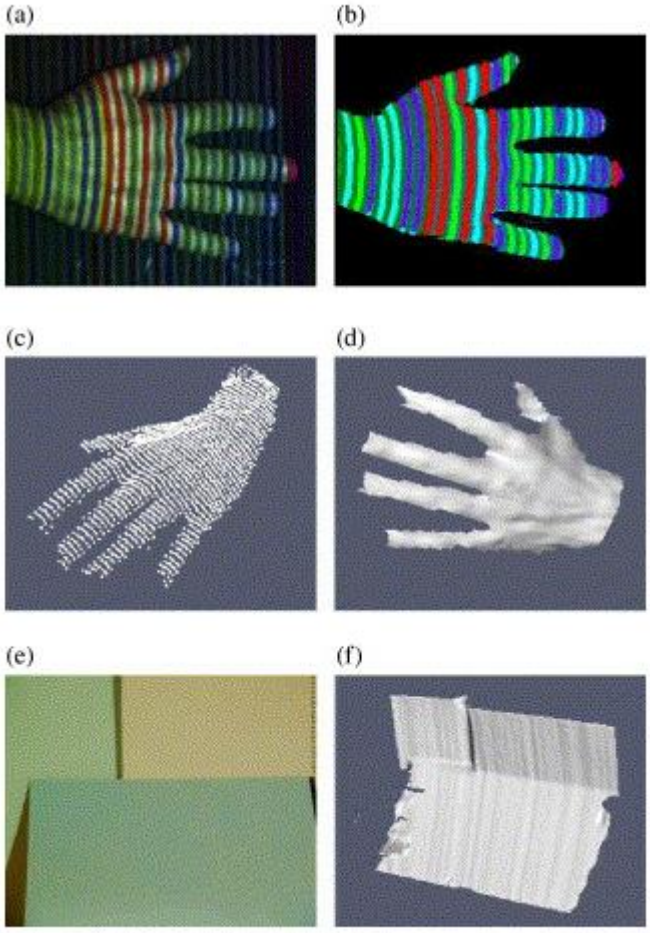
Obrázok 6.1: Ilustrácia princípu premietanej textúry na pohyblivú dosku

6.1.1 Virtuálna maska

Počas práce na tomto projekte som zvažoval aj ďalšiu možnú demonštračnou úlohou, principiálne podobnú predchádzajúcej, založenú na premietaní virtuálnej masky na tvár osoby stojacej pred projektorom.
Keďže ľudská hlava nemá veľmi rôznorodé veľkosti, mohlo by byť možné odhadnúť na základe polohy tváre na kamere polohu hlavy v priestore podobne ako pri premietacej doske. Následne by bol umiestnený jednoduchý trojrozmerný model tváre s aplikovanou textúrou vo virtuálnom priestore do pozície zodpovedajúcej snímanej tvári osoby. Túto masku by sme potom vykreslili a premietli projektorom ako dosku v predchádzajúcom prípade.
Od tejto úlohy som upustil, a to z dôvodu nenájdenia dostatočne presného a rýchleho algoritmu nachádzania polohy a natočenia tváre s ľubovoľným natočením v obraze. Ďalší problém je tu náročnosť testovania a potenciálne ohrozenie zraku užívateľa jasným svetlom projektora voči ktorému by musel stáť čelom.

6.2 Textúra nerovného povrchu

Mierne zložitejšou aplikáciou je využitie projektora na získanie trojrozmerného tvaru objektov v scéne, s následnou aplikáciou textúry na objekt.
Na tento účel je možné použiť projekciu štruktúrovaného svetla ktoré bude snímané kamerou, a takto dokážeme zrekonštruovať tvar objektu v jednej snímke. Spôsob ako k tomuto pristupujem je popísaný v časti 5.



Obrázok 6.2: Modelovanie scény pomocou De Brujinovych vzorov [14]

6.2.1 Generovanie textúry

Po získaní modelu objektu je možné na tento model namapovať textúru. Táto textúra by mala demonštrovať získanú informáciu o tvare objektu. Pre jednoduchosť je možné zvoliť výškovú mapu. Výška zobrazovaná mapou môže zobrazovať vzdialenosť od projektora, vzdialenosť od kamery, výšku v zvislom smere, alebo vzdialenosť po ľubovoľnej priamke. Úlohu výškovej mapy môžu plniť aj vrstevnice.
Takéto mapovanie je najjednoduchšie dosiahnuť ako planárnu alebo lineárnu projekciu textúry, zobrazujúcej súvislé, rovnobežné pásy alebo čiary zodpovedajúce jednotlivým výškam.

Ako textúrovací gradient som zvolil stochastickú postupnosť vygenerovanú v programe Adobe Photoshop. Na rozdiel od plynulého farebného prechodu je na takomto gradiente dobre vidieť oblasti rovnakej vzdialenosti nezávisle na vzdialenosti objektu. Pri použití plynulého prechodu by napríklad v prípade, že prechod je mapovaný na priveľký rozsah

vzdialeností, mohol byť celý objekt pokrytý zdanlivo rovnakou farbou, kde by vzdialenosť bola rozlíšená len malou zmenou odtieňa.
Ak by boli použité úzke pásy s väčšími rozstupmi, ako to býva v niektorých vrstevnicových zobrazeniach, potom by pásy z dôvodu rozlíšenia nemuseli byť dobre viditeľné.

Obrázok 6.3: Stochastický gradient používaný pri premietaní vrstevníc

Zložitejšie, ale pre demonštračné účely pôsobivejšie, je generovať textúru ktorá určitým spôsobom zodpovedá zakriveniu povrchu. Príkladom ktorý v tejto práci používam, je animácia tečúcej vody, ktorá simuluje obtekanie nerovného povrchu.
Pritom mám na mysli obraz vodopádu, kde takéto javy v prírode pozorujeme.

Obrázok 6.4: Fotografia vodopádu s vyznačenými stavmi vody ktoré napodobujem

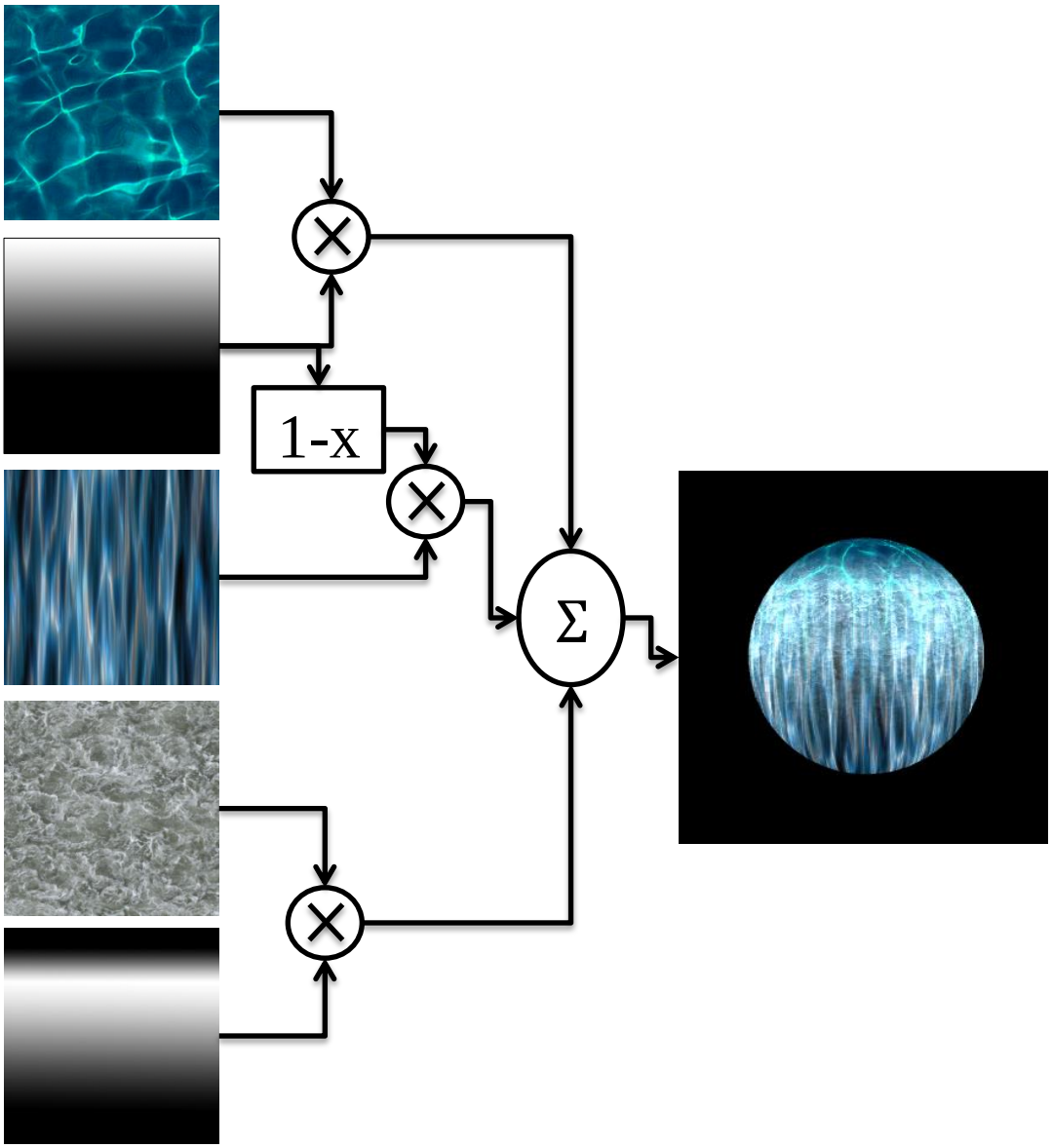
Keďže voda tečie rýchlejšie po rovinách s väčším sklonom, potrebujem textúru povrchu prispôsobiť jeho normálam.
Budem tu používať tri textúry zachycujúce vodu v troch stavoch:

- Pokojná hladina
- Rýchlo tečúci priamy prúd
- Pena

Okrem toho potrebujem dve maskovacie mapy, pomocou ktorých ich zlúčim:

- Maska prechodu vodorovnej hladiny k zvislej až prevrátenej nadol
- Maska okraja spádu, na ktorom sa tvorí svetlejšia pena.

Spôsob ako tieto textúry spájam do výsledného efektu je popísaný na Obrázok 6.5, kde je zahrnuté aj moje výsledné testovacie vykreslenie na guli. Čo nie je na obrázku viditeľné je, že celková textúra sa navyše mení v čase, a to posuvom textúr rýchlosťou zodpovedajúcou danému stavu vody. Tento postup je implementovaný vo fragmentovom shaderi (7.4.6).
V prípade úspešného zdokonalenia programu by malo byť možné napríklad premietnuť takúto textúru na sochu, a vytvoriť z nej tak v noci virtuálnu fontánu.



Obrázok 6.5: Kompozícia textúr použitá v programe, spolu s ukážkovým vykreslením

6.2.2 Rekonštrukcia pohyblivých scén

Zvažoval som aj úpravu tohto skenovania tak, aby bolo možné objekt opätovne modelovať v čase tak, aby sa prispôsobovala premietaná textúra premenlivej scéne. Pôvodné smerovanie k tejto možnosti je dôvodom, prečo som zvolil metódu skenovania z jednej snímky.

Moja koncepcia je taká, že pri stálom pozorovaní scény kamerou je na objekty premietaná textúra. Každých niekoľko snímok – napríklad jedna z 25 snímok za sekundu – by ale miesto textúry bol premietnutý vzor štruktúrovaného svetla miesto textúry, z čoho by bol získaný záber pre tvar objektu v danej sekunde. Takto by pre ľudské oko mohla vzniknúť ilúzia nepretržitej znalosti o tvare objektu. Alternatívou pre pravidelné vzorkovanie je spustenie ďalšieho skenovania vždy pri zachytení výraznejšej zmeny v scéne.

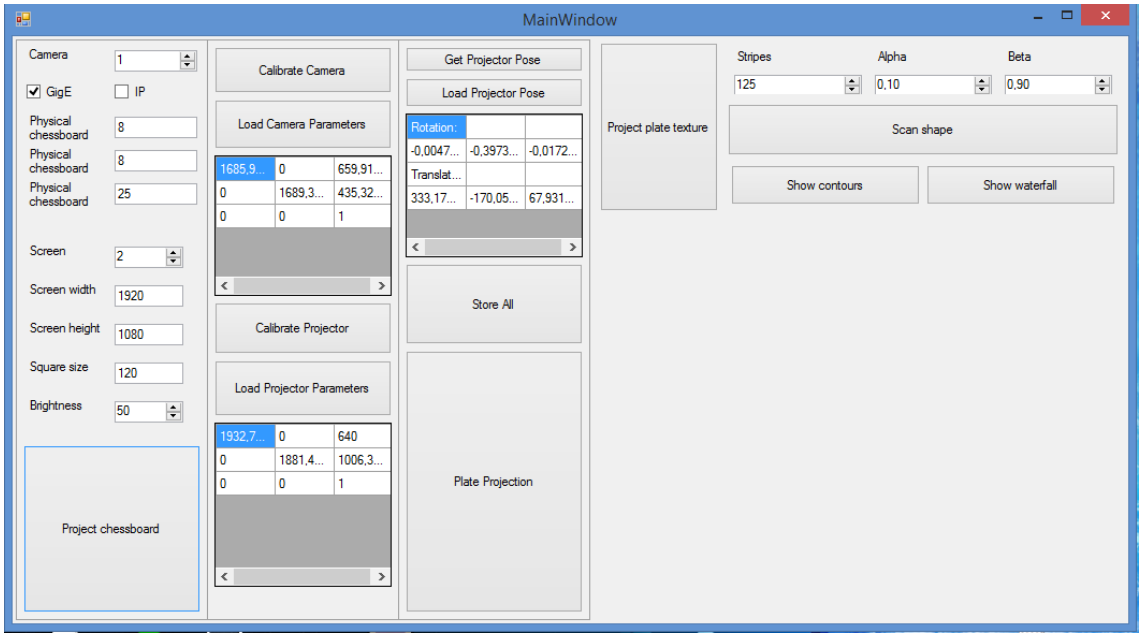
Túto úpravu sa mi nepodarilo implementovať. Rekonštrukcia tvaru objektu je príliš výpočtovo náročná na to aby ju bolo možné vykonať rýchlosťou blízkou reálnemu času. Toto je teoreticky možné vyriešiť použitím rýchlejšej výpočtovej techniky než akú mám k dispozícii, v kombinácii s optimálnejšími algoritmami. Výhodou by bolo skenovanie v nižšom rozlíšení z hľadiska pixelov alebo polygónov, čo by ale zhoršilo kvalitu rekonštrukcie.

Potenciálne problémy tu vidím ešte napríklad v synchronizácii snímok kde sa premieta štruktúrované svetlo so spustením algoritmu rekonštrukcie, ako aj v dôsledkoch striktných požiadaviek na osvetlenie scény.

7 PROGRAM

Základ demonštračných aplikácii som naprogramoval v jazyku C++ v prostredí Microsoft Visual Studio 2013.
Program má jednoduché grafické užívateľské prostredie a obsahuje aj textovú konzolu pre jednoduchšie zobrazenie hodnôt.
V programe využívam knižnice OpenCV na operácie počítačového videnia.
Pre pokročilé vizualizácie používam knižnicu OpenGL [22].
Ovládacie prvky programu, ako aj celý zdrojový kód s komentármi, som napísal v anglickom jazyku. Dôvodom je jednoduchšia orientácia pre prípadných zahraničných užívateľov a vývojárov.

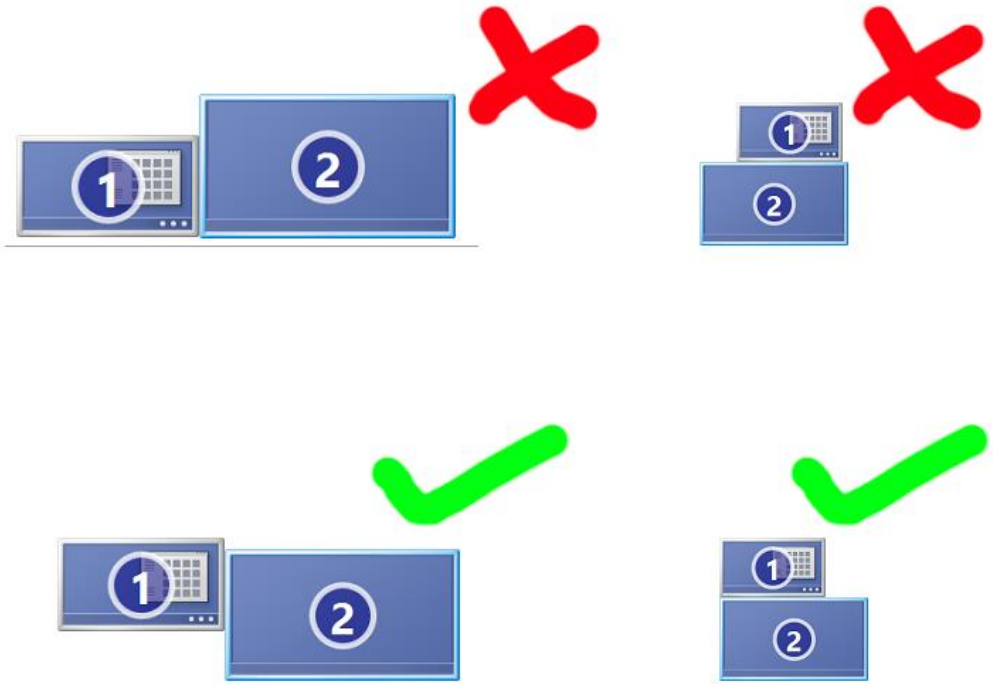
7.1 Užívateľské prostredie



Obrázok 7.1: Hlavné okno programu

Užívateľské prostredie je vo forme štandardného okna.
Umožňuje užívateľovi zvoliť si kameru ktorou bude scéna snímaná. Toto môže byť buď webkamera pripojená napríklad cez rozhranie USB, alebo kamera pripojená cez Gigabitovú sieť. Je možné aj pripojiť webkameru ktorá posiela obraz ako webový stream. Túto možnosť som využíval pri testovaní, kde som ako kameru používal inteligentný telefón s aplikáciou ktorá umožnila zdielať obraz z kamery cez sieť.
Užívateľ si môže zvoliť počet štvorcov na fyzickom kalibračnom vzore pre kameru, ako aj veľkosť štvorcov v milimetroch. Tento kalibračný vzor by mal byť podľa možnosti čiernobiely šachovnicový vzor. Pri testovaní som ako tento vzor používal štandardnú šachovnicu 8x8, so štvorcami so stranou dĺžky d = 25 mm.

Rovnako je možné zvoliť aj ktorý monitor bude použitý na premietanie, teda ktoré z pripojených zobrazovacích zariadení je projektor. Tu treba vziať na vedomie, že pre správne fungovanie programu musia byť zobrazovacie zariadenia v móde „Rozšírenia monitora“. Pritom je potrebné zabezpečiť, aby ľavý horný roh zvoleného monitora nebol vo virtuálnom usporiadaní ani vyššie, ako horný okraj hlavného monitora, ani na ľavo od ľavého okraja hlavného monitora. V opačnom prípade program nedokáže vykresľovacie okno umiestniť na celú plochu projektora.



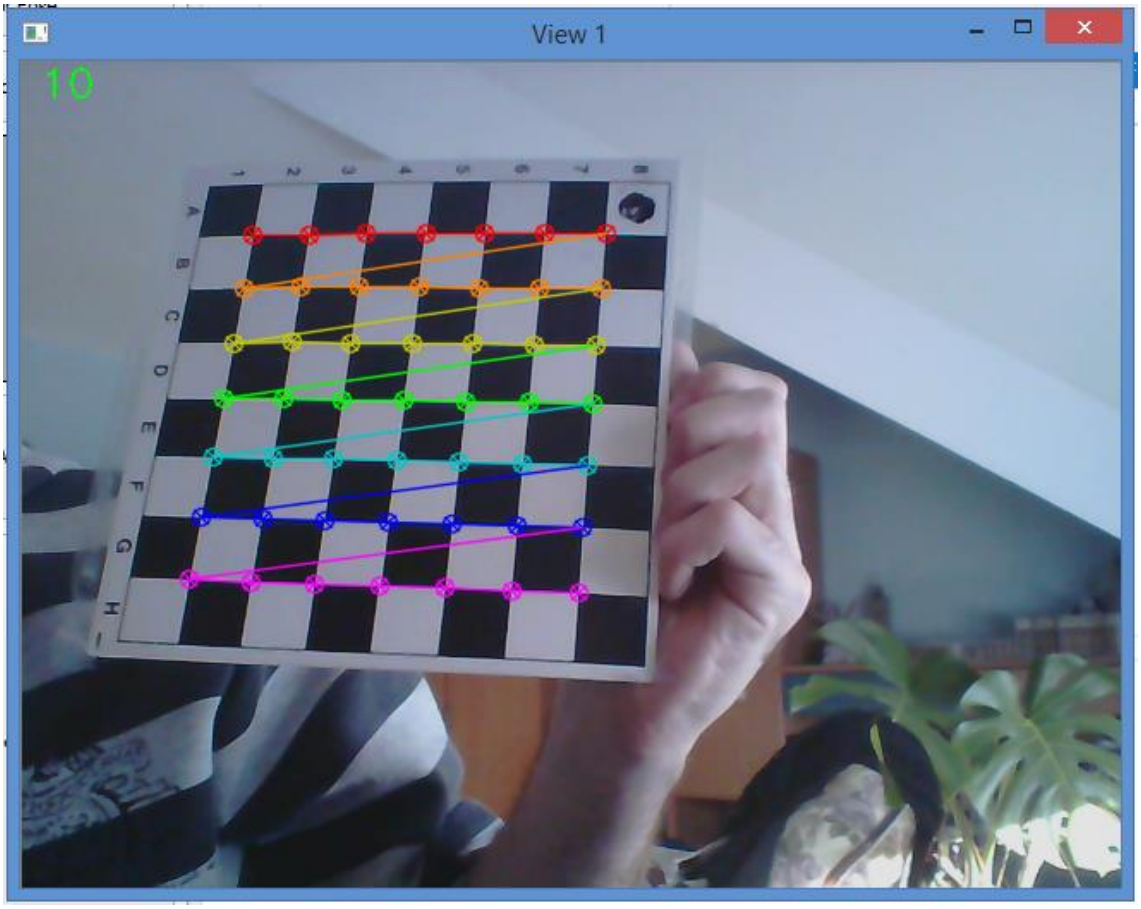
Obrázok 7.2: Prijateľné a nevhodné konfigurácie rozšírenej obazovky

Rozlíšenie daného monitora program rozpozná automaticky, ale je možné zmeniť rozmery premietaného kalibračného vzoru v pixeloch, ako aj veľkosť jeho štvorcov. Dôležitá je tiež možnosť nastaviť jas tohto vzoru. Pri priveľkom jase sa na kamere vzor javí nejasný, svetlé štvorce pretekajú do oblastí tmavých a je ich tak ťažšie detekovať. Vhodné nastavenie jasu je závislé aj od vzdialenosti od premietacieho plátna – čím je plátno bližšie, tým sa obraz javí jasnejší, čo vyplýva zo zákona inverzných štvorcov vzdailenosti. Zadaný jas udáva hodnotu od 0 po 255 pre všetky tri RGB kanály bielych štvorcov.

Tlačidlo „Project chessboard“ zobrazí okno bez okrajov, ktoré je automaticky umiestnené na plochu zvoleného zobrazovacieho zariadenia a ktoré je vyplnené kalibračným šachovnicovým vzorom.

Ak je zvolená kamera, je možné spustiť jej kalibráciu tlačidlom „Calibrate Camera“.

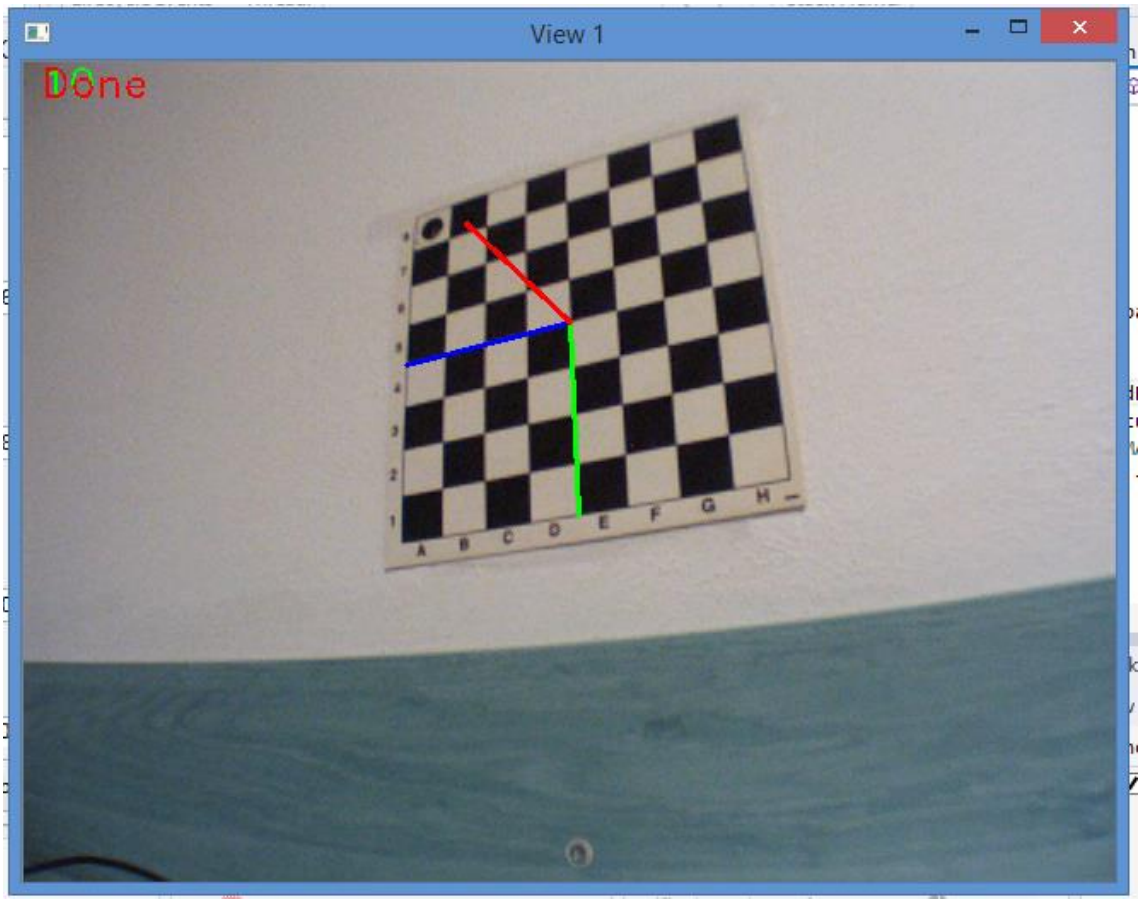
Po jeho stlačení sa zobrazí okno s obrazom zo zvolenej kamery. V prípade detekcie šachovnice zvolených rozmerov je tento vzor označený farebnými čiarami a kružnicami okolo rohových bodov.



Obrázok 7.3: Detekované rohy na šachovnici

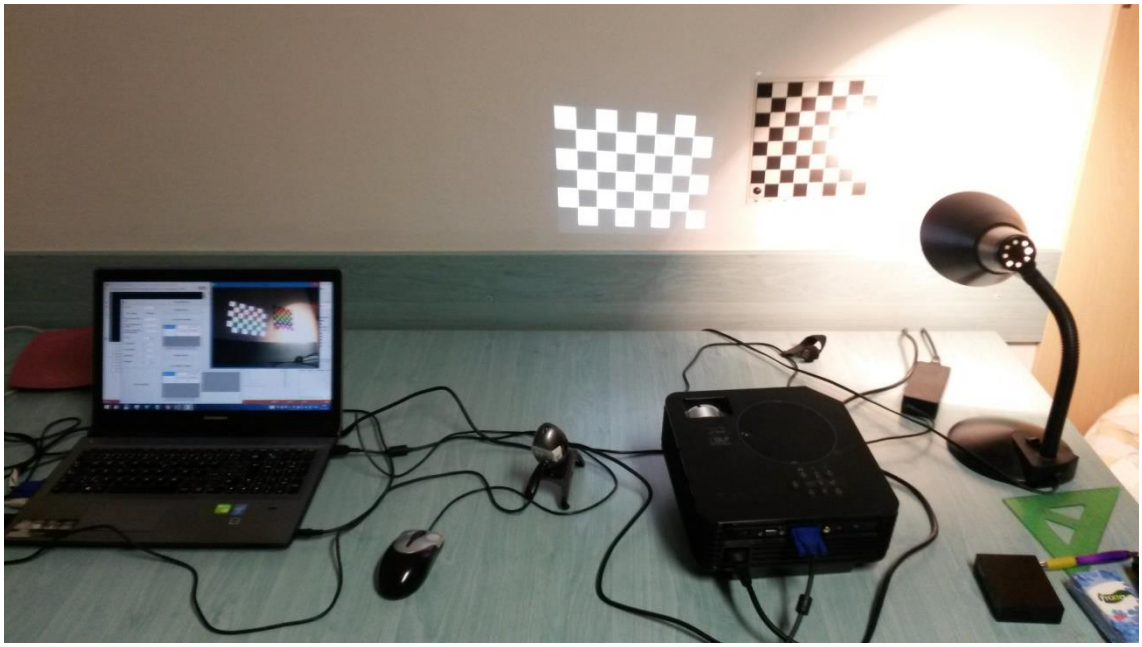
Stlačením tlačidla Enter sa polohy rohových bodov v danej snímke – ak sú nejaké detekované – uložia a číslo v ľavom hornom rohu sa dekrementuje. Toto číslo indikuje požadovaný počet ďalších snímok na kalibráciu, a jeho úvodná hodnota je v kóde nastavená na 10. Tento počet sa pri testoch ukázal byť ako dostatočný pre získanie kvalitnej kalibrácie, v prípade že jednotlivé použité snímky obsahujú dostatočne rôznorodé uhly pohľadu na vzor a jeho pozície v obraze.
Po uložení desiatich vzorov sa vypočítajú intrinzické parametre kamery a zobrazovacie okno sa zavrie. V prípade, že užívateľ chce predčasne ukončiť kalibráciu kamery, je možné tak učiniť stlačením tlačidla Esc.
Po kalibrácii sa parametre uložia do textového súboru CamParams.txt v aktuálnej zložke programu. Tento súbor je vhodné pre budúce použitie kopírovať do vlastnej zložky. Okrem toho sa zobrazia hodnoty intrinzickej matice kamery v poli typu DataGridView. Parametre je možné aj načítať z predchádzajúcich súborov, a to tlačidlom „Load Camera Parameters“.
Ak sú už parametre získané – kalibráciou alebo načítaním už získaných hodnôt – tak tlačidlo „Calibrate Camera“ spustí inú funkciu. Táto umožňuje testovať kvalitu kalibrácie umiestnením troch úsečiek do obrazu. Každá úsečka je vo virtuálnom priestore dlhá 100mm. Tieto úsečky reprezentujú tri rozmery, a pri správnej kalibrácii budú umiestnené do stredu snímaného šachovnicového vzoru s dvoma úsečkami

rovnobežnými so stranami šachovnice a treťou úsečkou simulujúcou smer kolmý na šachovnicu.



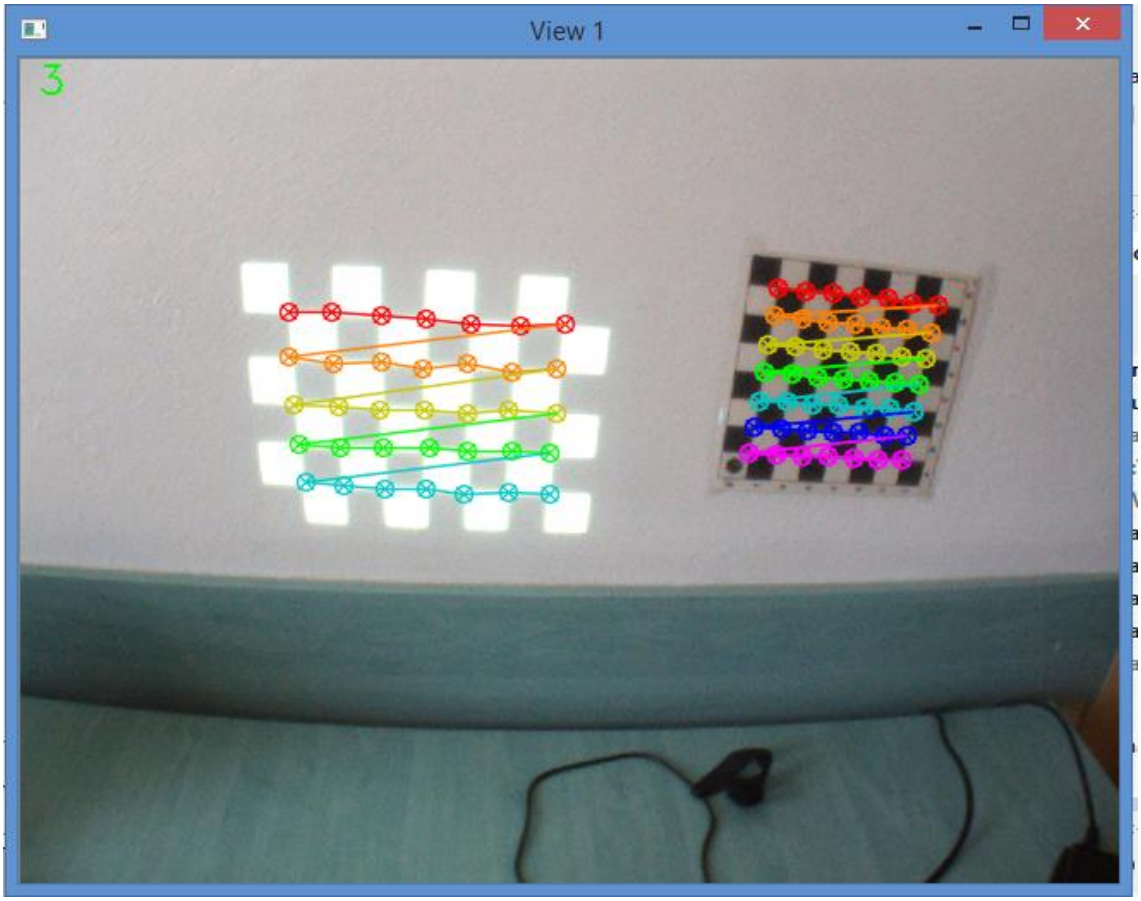
Obrázok 7.4: Demonštrácia kvality kalibrácie kamery

Ak je kamera skalibrovaná, a zároveň je aktuálne premietaný kalibračný vzor, potom je možné spustiť tlačidlom „Calibrate Projector“ kalibráciu projektora.
Pre tento účel je potrebné umiestniť fyzický kalibračný vzor na tú istú fyzickú rovinu, na ktorú je vzor premietaný, ako je to napríklad na Obrázok 7.5.



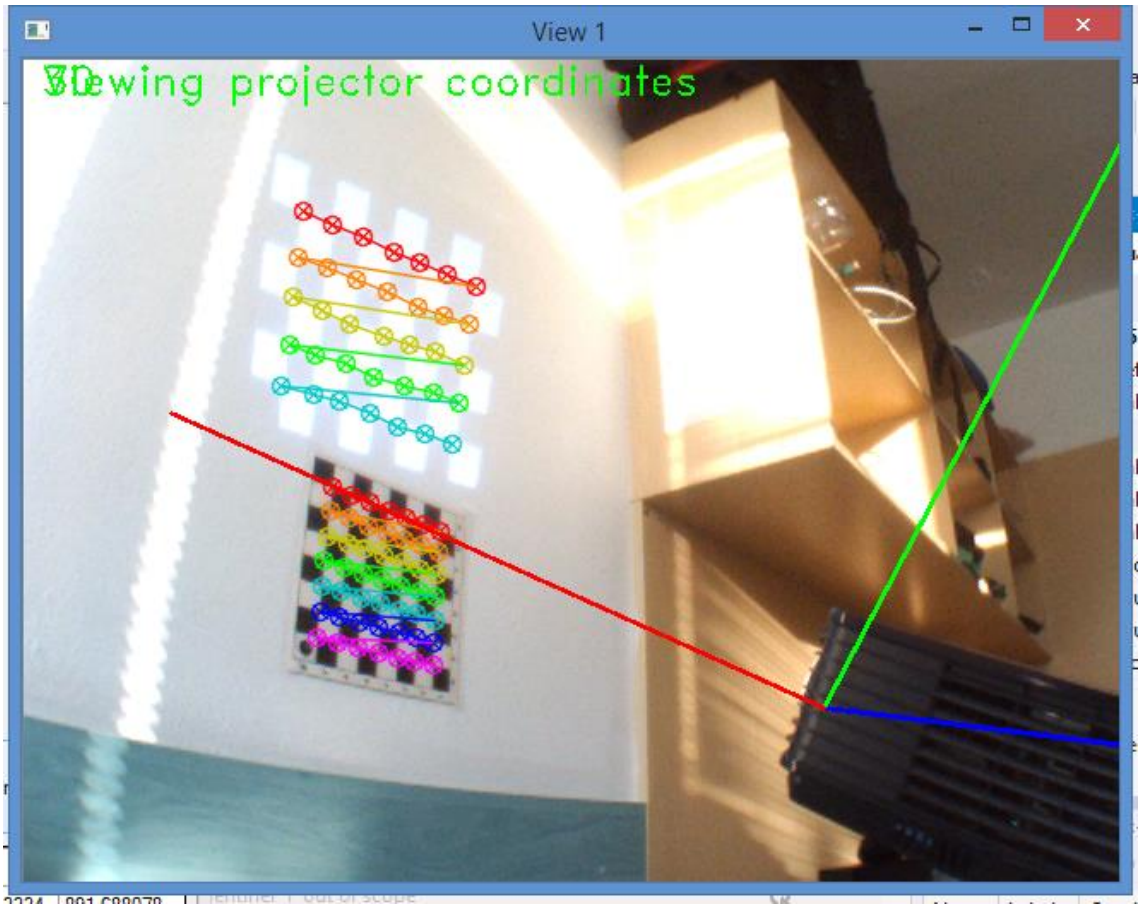
Obrázok 7.5: Zostava pre vývoj a testovanie programu

Kamera musí byť smerovaná tak, aby oba vzory – fyzický aj premietaný – boli zachytené. Ak sú vzory detekované, budú zvýraznené rovnakým spôsobom ako pri kalibrácii kamery.



Obrázok 7.6: Detekcia fyzického a premietnutého vzoru

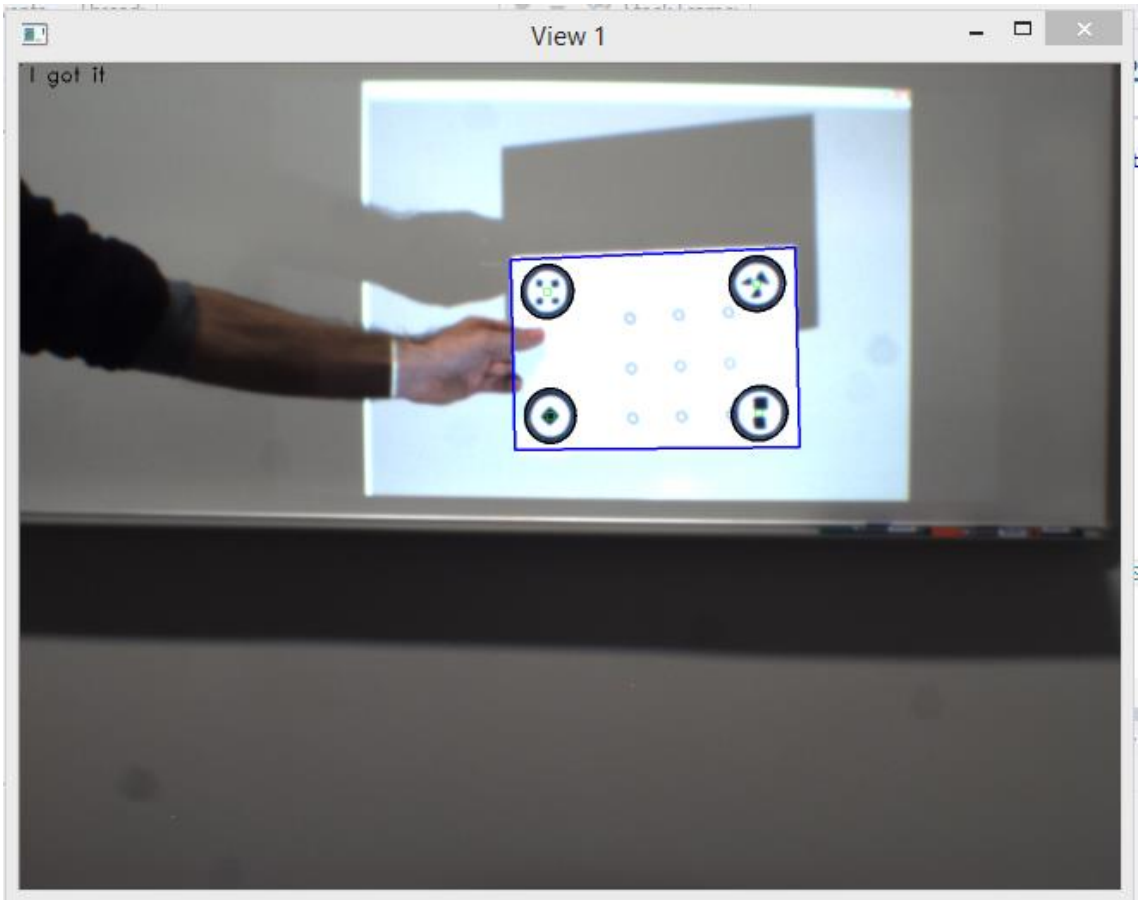
V prípade že sú oba vzory v obraze, ale jeden z nich nie je zvýraznený, je potrebné upraviť svetelné podmienky. Toto je možné dosiahnuť pridaním, alebo zmenou nastavenia zdrojov svetla. Druhou možnosťou je pozmeniť jas premietaného vzoru. Toto sa dá dosiahnuť tak, že klikneme na okno s premietaným vzorom myšou, čím sa zavrie, potom pozmeníme v hlavnom okne hodnotu jasu vzoru a znova toto okno zobrazíme tlačidlom „Project chessboard“
Ak sú zvýraznené na kamere oba vzory, uložíme rozloženie do pamäti tlačilom Enter. Toto je potrebné opakovať 30 krát, s čo najrôznejšími uhlami pohľadu kamery a zároveň rôznymi uhlami premietania projektora na rovinu.
Po uložení tridsiatich záznamov sa vypočítajú parametre projektora a okno s obrazom z kamery sa zavrie. Parametre sa zobrazia a uložia rovnako ako pri kalibrácii kamery a je ich aj rovnako možné načítať.
Ak už program získal parametre projektora, potom tlačidlo „Calibrate Projector“ spustí funkciu na overovanie kalibrácie, kde umiestňuje do obrazu podobnú trojicu úsečiek ako po kalibrácii kamery. Tento krát je trojica umiestnená tak, aby odhadovala polohu a smer projektora na základe zaznamenaných kalibračných vzorov a parametrov kamery a projektora.



Obrázok 7.7: Demonštrácia kvality kalibrácie projektora – červená čiara aproximuje optickú os projektora

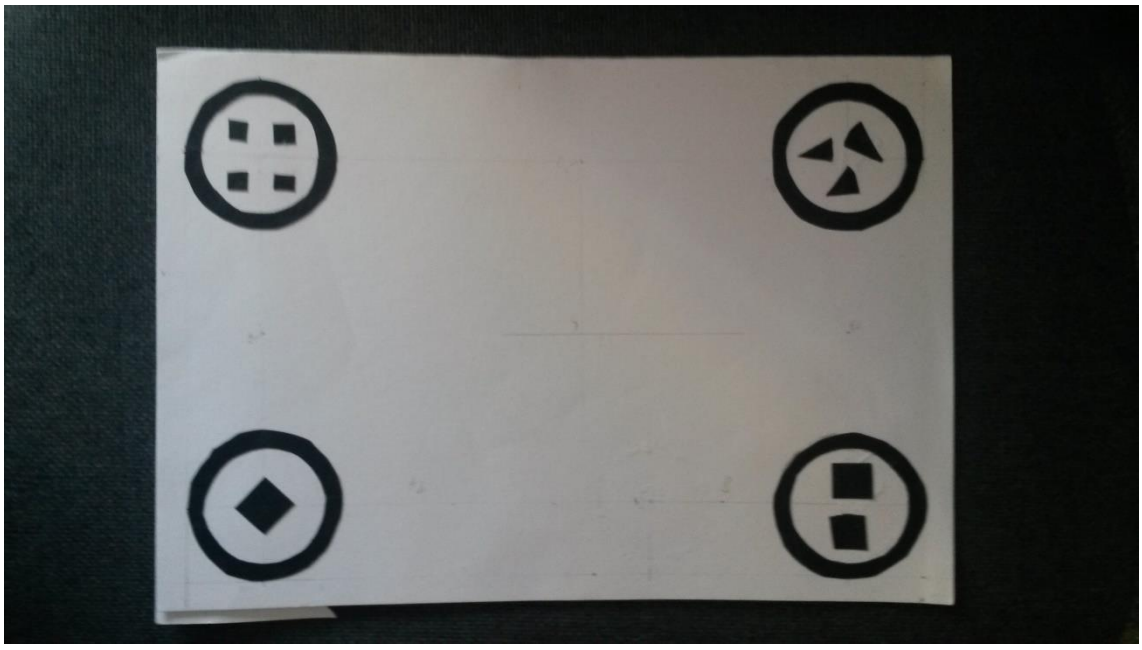
Na Obrázok 7.7 si môšete všimnúť, že projektor premieta vzor smerom nahor od optickej osi tak, že táto jeho obraz ani nepretína. Takto sú projektory konštruované pomerne často.
Ak už poznáme parametre projektora aj kamery, potom môžeme určiť ich vzájomnú polohu. Toto sa robí v samostatnej fáze pomocou tlačidla „Get Projector Pose“. Zobrazí sa okno s obrazom kamery, v ktorom sa podobne ako pri kalibrácii projektora zvýrazňujú dva kalibračné vzory. Ak sú oba viditeľné na kamere, potom tlačidlom Enter vypočítame polohu a zavrieme okno. Vzájomný posuv a natočenie sa zobrazia uložia rovnako ako parametre v predchádzajúcich bodoch, a rovnako je ich aj možné načítať.
Cesty k všetkým trom použitým súborom – parametre kamery, parametre projektora a vzájomná poloha – je možné uložiť tlačidlom „Store All“. Takto sa vytvorí súbor ktorým je možné načítať pri spustení programu všetky hodnoty.

Po získaní Parametrov kamery aj projektora a aj ich vzájomnej polohy, môžeme zahájiť prvú demonštračnú úlohu tlačidlom „Plate Projection“
Toto tlačidlo zaháji cyklus, v ktorom kamera vyhľadáva dosku so známou štvoricou symbolov, s pomocou ktorých vie určiť natočenie a polohu dosky.



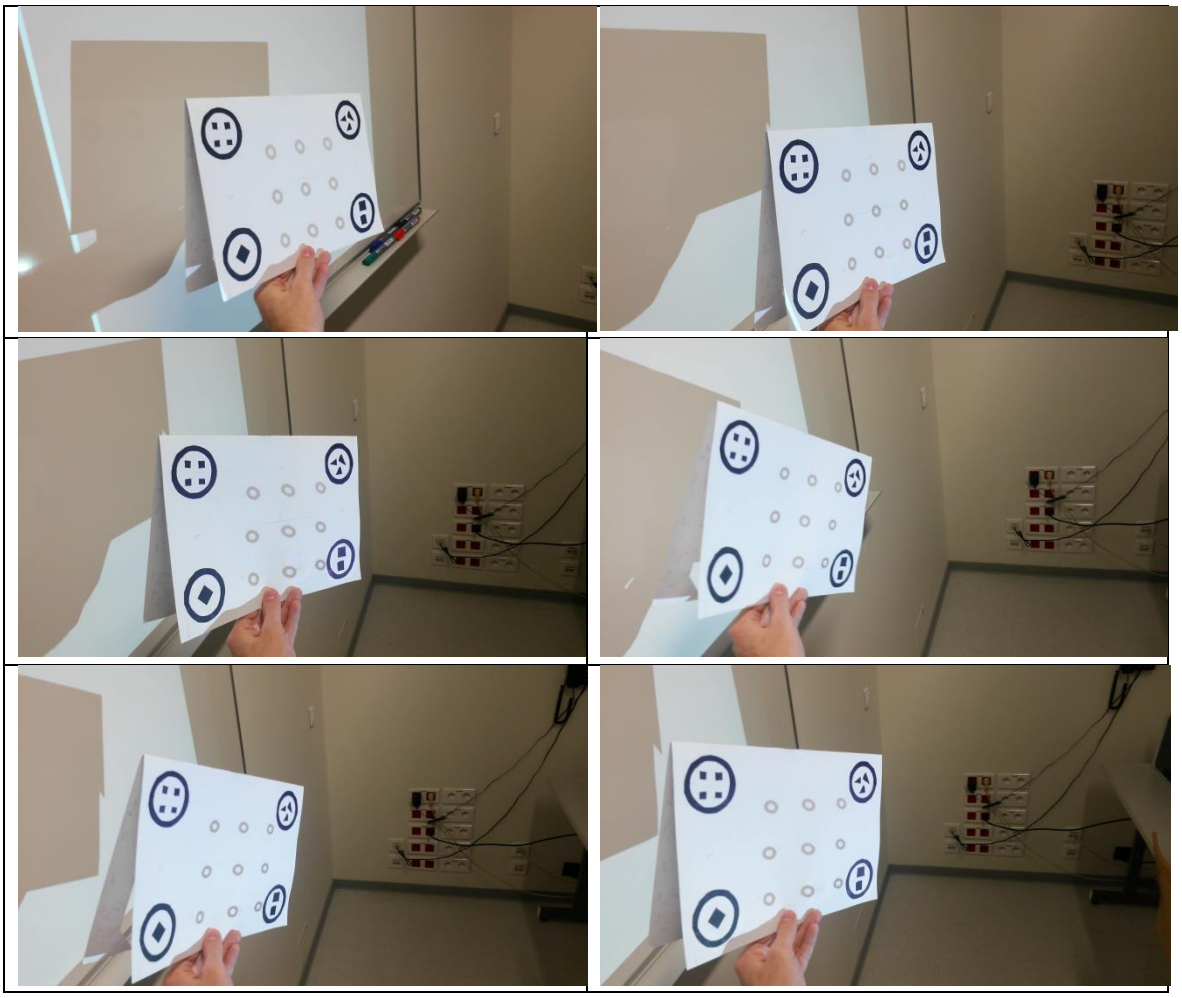
Obrázok 7.8: Detekcia polohy a natočenia premietacej dosky

Tieto symboly sú vystrihnuté s matného čierneho papiera, ktorý dobre absorbuje jasné svetlo z projektora a okolitého osvetlenia. Jednotlivé symboly sú individuálne odlíšiteľné pre jednoznačné určenie polohy. Stredy symbolov majú definované rozstupy, a to 220 mm pozdĺž dlhšej strany a 130 mm pozdĺž kratšej.



Obrázok 7.9: Použitá premietacia doska s detekovateľným vzormi

Ak sa doska nachádza v dosahu projektora a zároveň je viditeľná na kamere, bude na ňu premietaná skupina deviatich bodov, ktoré sa projektor snaží umiestniť na dosku s ohľadom na jej natočenie a polohu.
Projektor tu premieta biele pozadie, ktorého jas je určený rovnakým nastavením ako jas kalibračných vzorov v predchádzajúcich krokoch. Biele pozadie zároveň poskytuje osvetlenie na pohyblivú premietaciu dosku, ktoré zlepšuje viditeľnosť pre kameru.



Obrázok 7.10: Premietnutý vzor zodpovedá polohe a natočeniu dosky

V prípade že je skupina bodov neustále posunutá voči doske, je možné že bola nesprávne odhadnutá vzájomná poloha projektora a kamery. V takomto prípade je možné buď znova túto polohu detekovať, alebo ručne prepísať vzájomný posuv v súbore kde bol uložený a znova ho načítať.
Pozornosť je potrebné v tomto ohľade venovať aj nastaviteľnému smeru lúčov projektora. Niektoré modely sú totiž vybavené prvkami pre úpravu smeru lúčov z projektora, ktoré zmenia polohu optického centra projektora. Pri testovaní sa ukázalo, že ak boli tieto parametre nedopatrením pozmenené, je možné práve pomocou funkcie programu pre premietanie bodov na plochu opraviť nastavenie. Toto sa vykonáva manuálnym centrovaním premietaných bodov na premietaciu dosku. Po takejto korekcii bude premietanie správne, za podmienky že nebola zmenená ohnisková vzdialenosť projektora.

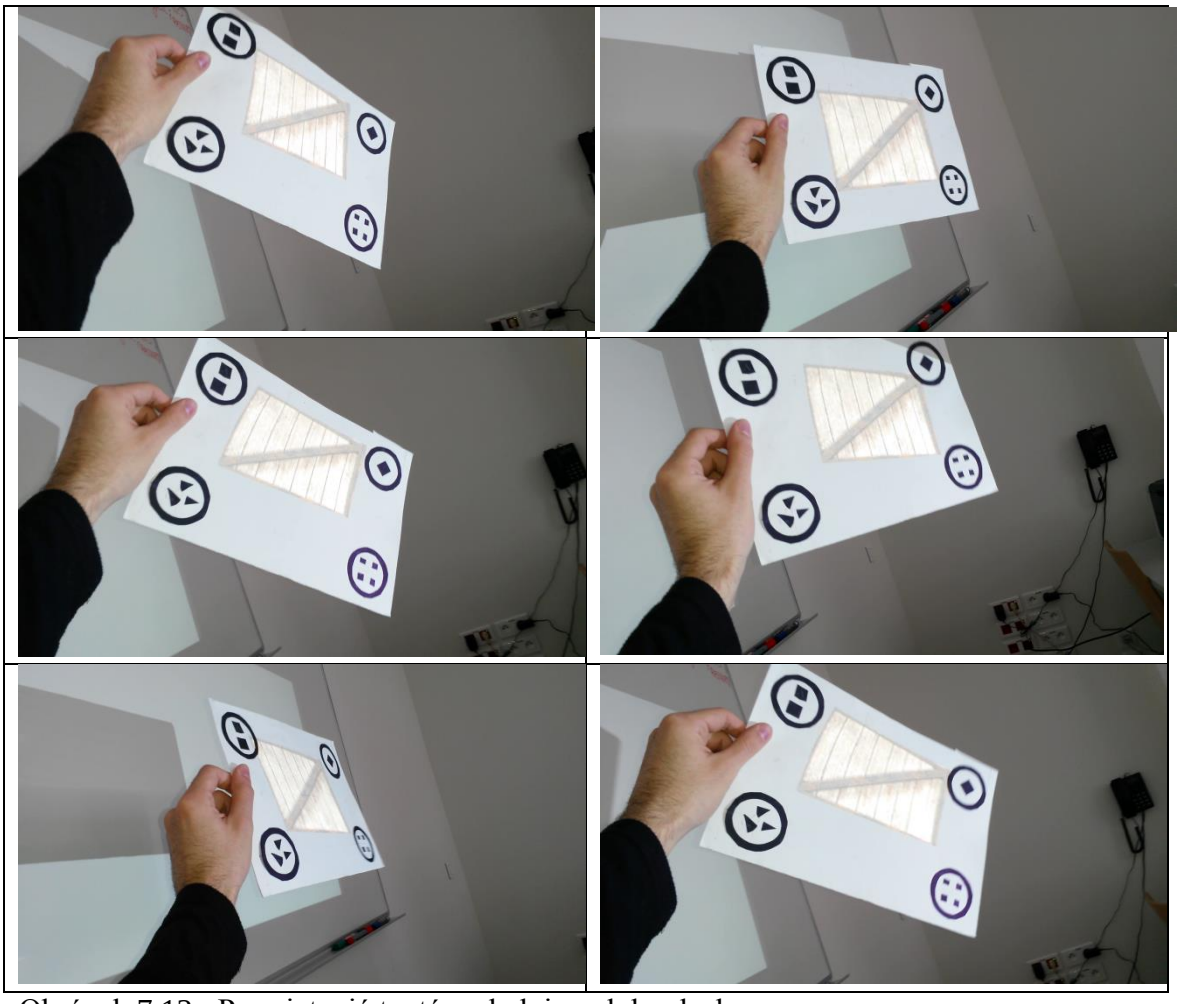
Obrázok 7.11: Ovládacie prvky na projektore EPSON EMP-TW620 [23]

V prípade že ohnisková vzdialenosť zmenená bola, s použitím prvkov nastavenia veľkosti premietaného obrazu, môže byť potrebná opätovná kalibrácia projektora. Keďže táto úloha je časovo náročná a vyžaduje časté premiestňovanie kamery a projektora, je vhodné pracovať s jednou s krajných polôh ovládania ohniskovej vzdialenosti. Takto je možné jednoducho opraviť zmenu nastavením vybranej krajnej polohy.

Projektor je často možné zaostriť na vybranú vzdialenosť. Pri používaní tohto programu je výhodné zvoliť si strednú vzdialenosť od projektora v ktorej chceme pracovať a zaostriť projektor na túto vzdialenosť. Zmena vzdialenosti na ktorú projektor zaostrujeme pri testoch nemala pozorovateľný vplyv na presnosť projekcie. V modeli projektora vo forme dierkovej komory, ktorý je tu používaný pri výpočtoch, totiž takéto ostrenie nemení ohniskovú vzdialenosť.

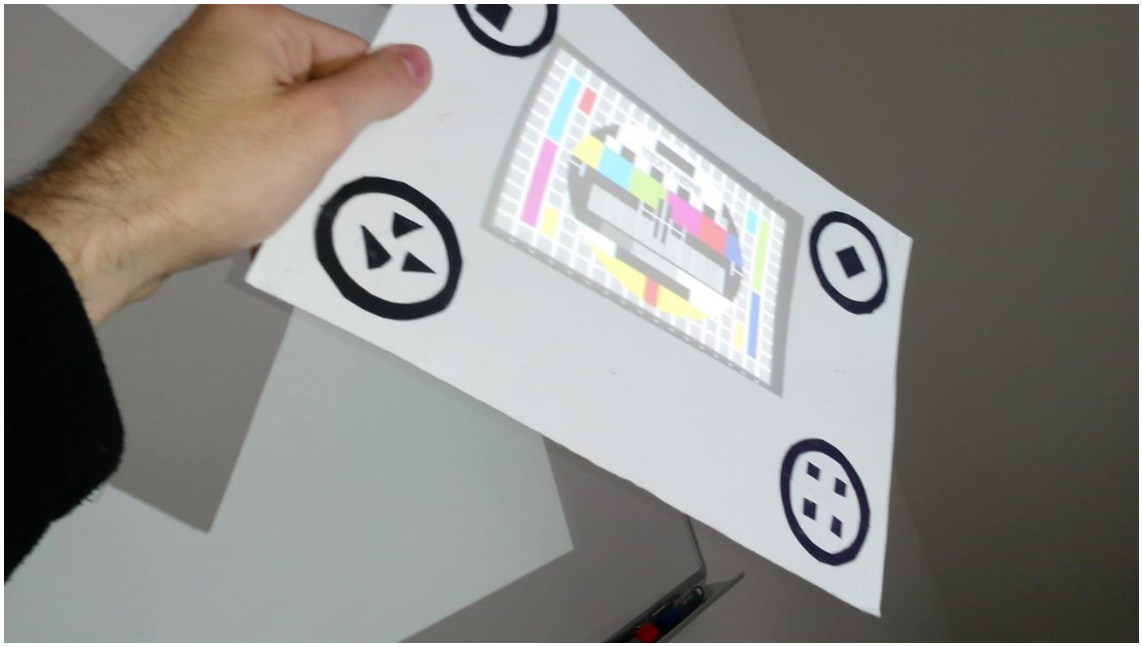
Pri projektoroch je ale nutné vziať na vedomie, že je väčšinou možné digitálne alebo mechanicky zmeniť aj lichobežníkovú korekciu. Pre naše účely je potrebné lichobežníkovú korekciu podľa možnosti vypnúť, pretože by skreslila vypočítané parametre projektora.

Na pravej strane hlavného okna je možné spustiť dve demonštračné aplikácie s využitím OpenGL. Pre použitie oboch je najprv potrebné skalibrovať kameru a projektor a určiť ich vzájomnú polohu.
Tlačidlo „Project plate texture“ spustí premietanie textúry na rovnakú projekčnú dosku ako pri premietaní bodov po stlačení tlačidla „Plate Projection“.
Táto textúra bude vždy premietaná tak, aby sa z pohľadu kolmo na dosku javila ako plochý obraz na nej. Toto demonštruje výmenu dát medzi počítačovým videním a rozhraním OpenGL. Je tu vidieť, že boli správne nastavené extrinzické aj intrinzické parametre projektora do projekčnej matice v OpenGL, ako aj správne určenie polohy dosky v priestore.



Obrázok 7.12 : Premietaniá textúra sleduje polohu dosky

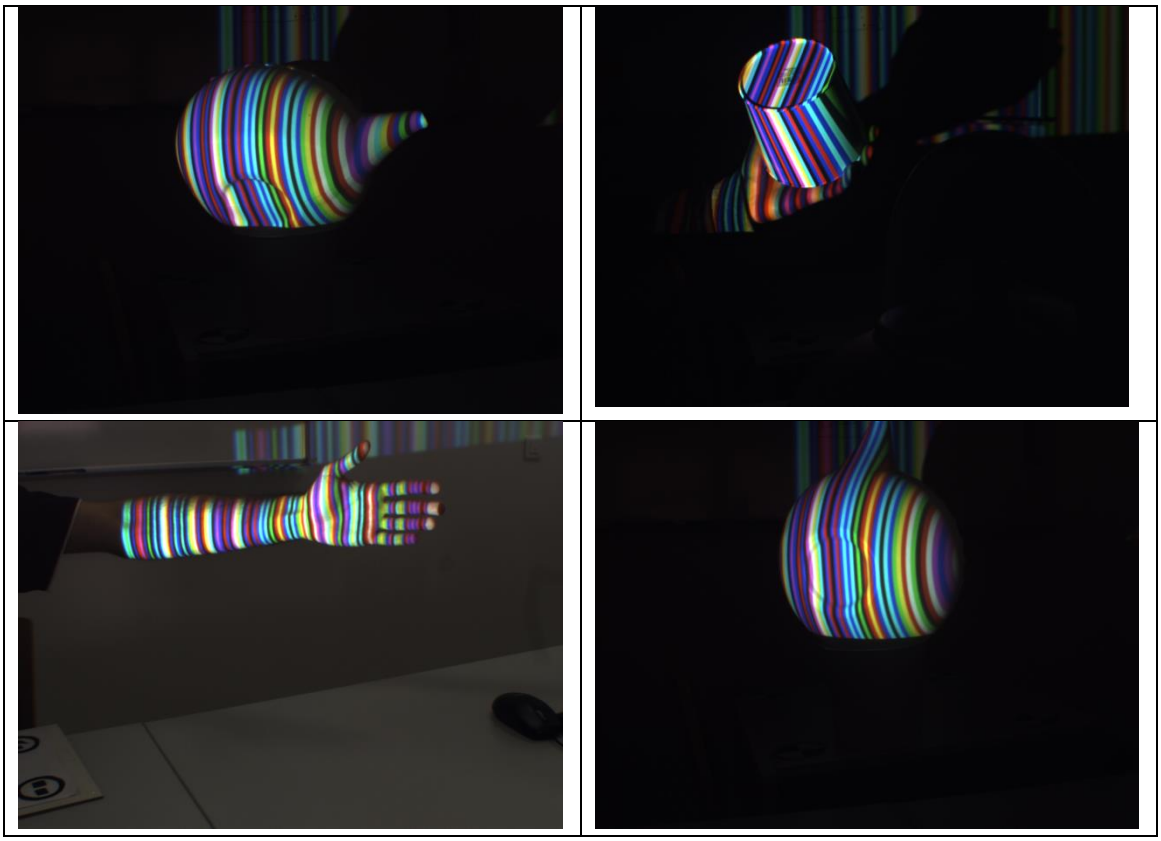
Textúra je zvolená staticky v kóde, keďže pre demonštračné účely nemá iná textúra zvláštny význam, no úpravou v kóde je možné zmeniť textúru na obrázok ľubovoľného obsahu. Tento by pre zaistenie kompatibility mal mať šírku v pixeloch rovnú celočíselnej mocnine dvoch a nie väčšiu ako 512, a mal by byť pokiaľ možno vo formáte BMP.



Obrázok 7.13: Použitie alternatívnej textúry

Poslednou funkciou programu je textúrovanie povrchu fyzických objektov, v závislosti od ich tvaru. Táto funkcia sa skladá z dvoch fáz. Prvou je rekonštrukcia tvaru objektu. Pre tento účel je tu využité štruktúrované svetlo. To má podobu farebných pásov v tzv. De Bruijnovej sekvencii – viď časť 5.1. Počet premietaných pásov zvolíme v numerickom poli „Stripes“. Pásov môže byť najviac 125. Jas premietaného vzoru nastavíme rovnako ako pri pri premietanom kalibračnom vzore, teda v poli „Brightness“ v ľavej časti hlavného okna. Tlačidlom „Scan shape“ premietneme zvoleným projektorom štruktúrované svetlo, a zobrazíme výstup zo zvolenej kamery. Funkcia je výrazne citlivá ako na svetelné podmienky, tak aj na zaostrenie kamery a projektora a na tvar a farbu snímaného objektu.

Pre optimálne výsledky je potrebné zabezpečiť čo najmenšiu mieru vonkajšieho osvetlenia. Kameru je s pomocou clony a prípadne zmenou expozície potrebné nastaviť tak, aby sa snímaný obraz skladal z čierneho okolia, a zakrivených farebných pásov v čo najvýraznejších základných farbách bez väčších gradientov jasu. Treba teda zamedziť vzniku výrazných odleskov, ale zároveň zabezpečiť dostatočný jas pásov v obraze tak, aby neboli v určitých oblastiach objektu tmavšie ako inde. Toto nastavenie je možné dolaďovať aj zmenou jasu premietaného obrazu. Príklad požadovaného zobrazenia je na fotografiách z Obrázok 7.14.



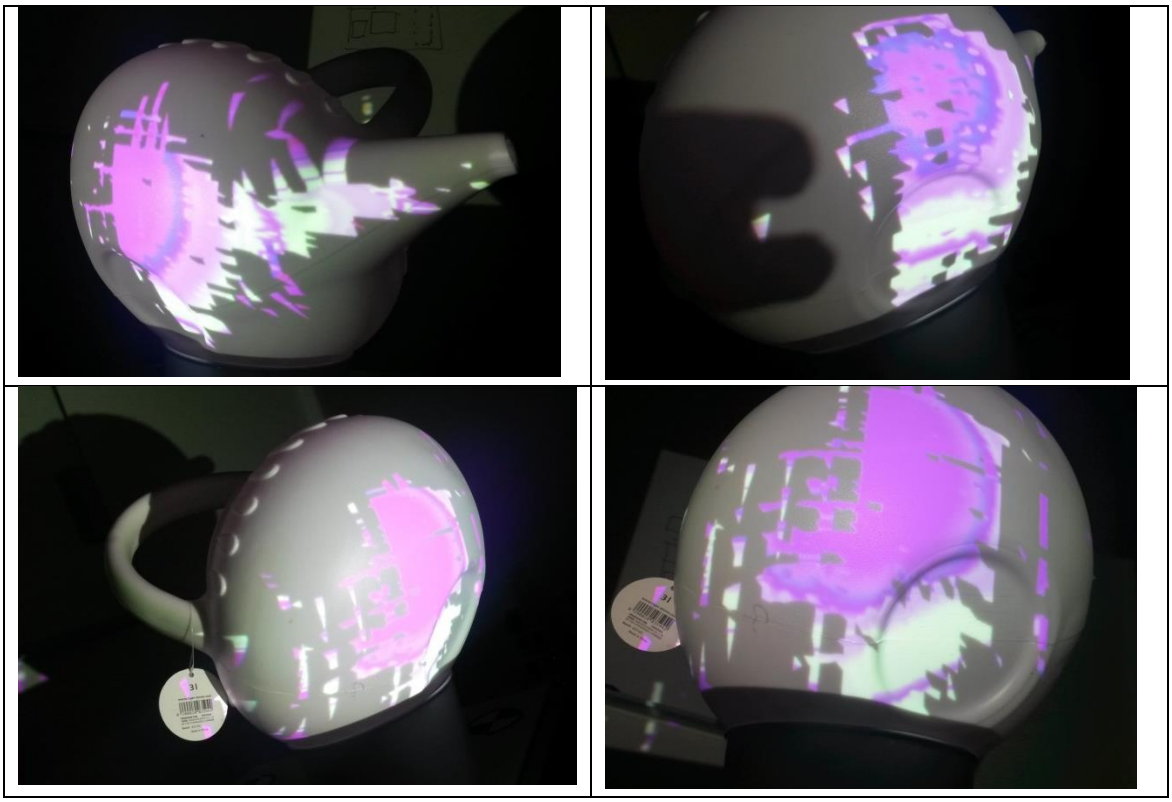
Obrázok 7.14: Vhodný obraz objektov osvetlených štruktúrovaným svetlom

Okrem svetelných podmienok je kvalita rekonštrukcie závislá na tvare a na vlastnostiach povrchu snímaného objektu. Objekt by mal zaberať čo najväčšiu časť snímaného obrazu. Keďže projektory zväčša nie je možné zaostriť na veľmi malú vzdialenosť, táto podmienka znamená že snímaný objekt nesmie byť príliš malý a mal by byť súvisle pokrytý čo najväčším počtom pásov. Objekt by nemal byť príliš členitý, najmä v smere pozdĺž premietaných pásov. Plochy objektu ktoré chceme modelovať musia byť viditeľné v obraze z kamery a ich sklon voči osi kamery nesmie byť tak veľký, aby sa na ich obraze nedali spoľahlivo odlíšiť jednotlivé farebné pásy.
Objekt by nemal byť ani príliš tmavý, aby pásy lepšie vynikli, a nemal by byť výrazne farebný.
Pred snímaním je tiež možné meniť v numerických poliach „Alpha“ a „Beta“ parametre funkcie konzistencie, popísanej v časti 5.1. Hodnota Alpha musí byť menšia až rovná hodnote Beta, pričom vyšší rozstup umožní detekciu aj pri obraze horšej kvality, ale prináša riziko nesprávnej identifikácie farebných prechodov v sekvencii. Opakovaným testovaním som dospel k záveru, že pre väčšinu podmienok sú vhodné hodnoty v okolí Alpha = 0,05 a Beta = 0,95.

Po splnení týchto podmienok tlačidlom Enter nasníma program osvetlený objekt. Táto operácia môže trvať značnú dobu. Táto závisí na viacerých faktoroch, ako je rozlíšenie kamery, výkon používaného procesora a počet pásov premietaného vzoru.

Menší počet pásov môže výrazne zrýchliť výpočet, ale model objektu má potom menšie polygónové rozlíšenie a nesprávne identifikovaný bod ovplyvní väčšiu plochu textúry premietanej v druhej fáze. Príklad výsledku takéhoto modelovania s nižším rozlíšením je na Obrázok 7.15 v pravom hornom rohu.

Po nasnímaní objektu je možné jedným z tlačidiel „Show contours“ a „Show waterfall“ premietať textúru na snímaný objekt. Textúra bude pokrývať len tie časti povrchu, ktoré boli úspešne lokalizované v priestore.
Tlačidlo „Show contours“ premietne na objekt gradient, ktorý zobrazí na objekte vrstevnice, z hľadiska vzdialenosti od projektora.
Na obrázkoch Obrázok 7.15 je vidieť, že farebné prechody textúry sú zhodné s kontúrami rovnobežných plochých rezov objektom, kolmých na optickú os projektora.

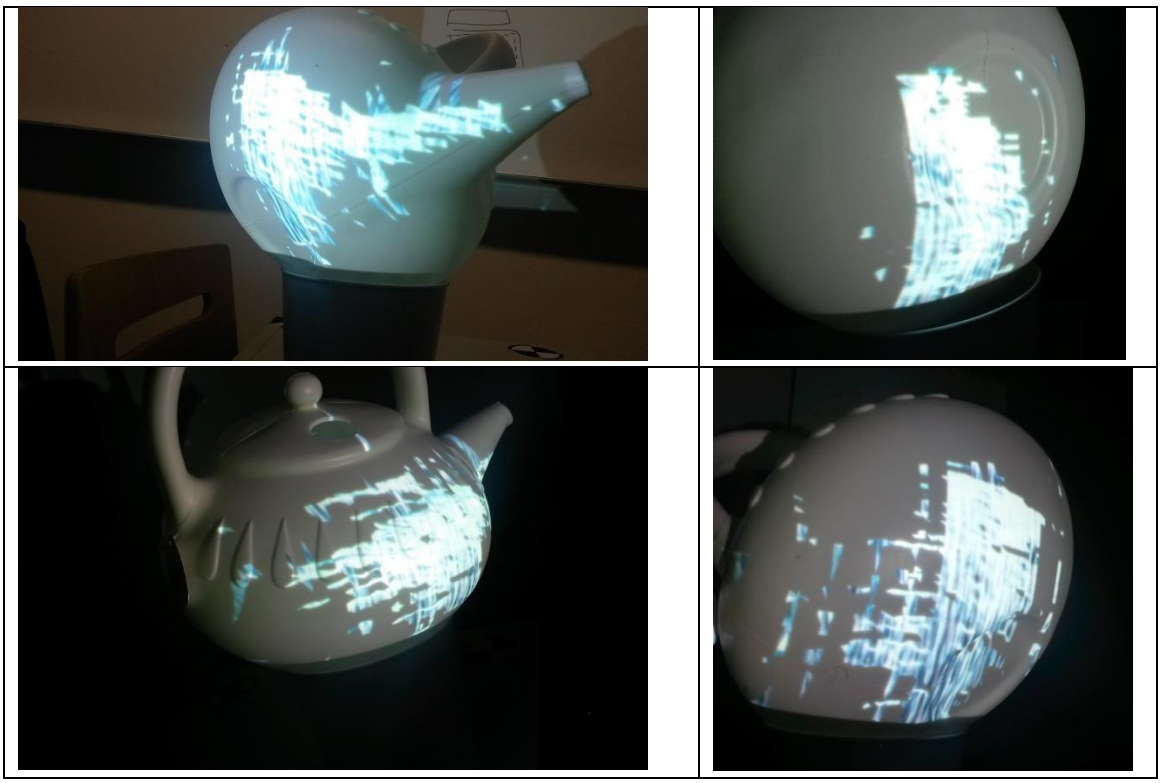


Obrázok 7.15: Premietanie vrstevnicového gradientu na objekt

Textúrovaný 3D objekt je pre testovacie účely možné posúvať v projekcii. Toto sa vykonáva posúvaním myši na obraze projektora, za stáleho držania jej ľavého tlačidla. Kliknutím pravým tlačidlom myši na obraz projektora sa projekcia ukončí.

Pri spustení premietania takto textúrovaného objektu je jeho 3D model pre účely testovania alebo archivácie automaticky exportovaný do súboru „export.obj“ . Tento súbor je vo formáte OBJ spoločnosti Wavefront [24].

Tlačidlo „Show waterfall“ spustí premietanie vizuálne zaujímavejšieho efektu. Funkcia premietne na objekt dynamickú textúru stekajúcej vody, závislú na sklone plôch objektu. Povrchy ktoré sú blízke vodorovnej polohe sú pokryté pomaly sa meniacim obrazom hladiny vody, povrchy ktoré sú zvislé až naklonené nadol sú pokryté obrazom zvislo tečúcej vody, a na rozhraní týchto dvoch oblastí je premietaná rozvírená pena. Všetky tri oblasti nemusia byť viditeľné pri každom objekte.
Táto aplikácia demonštruje znalosť programu o normálach povrchu objektu, ako aj možnosti použitia pokročilejších efektov pri spolupráci projektora a kamery.
Na snímkach na Obrázok 7.16 je vidieť, že zvislé plochy sú pokryté svetlou textúrou peny, ztiaľ čo plochy naklonené smerom nadol zobrazujú textúru tečúcej vody. Na plochách naklonených nahor je tiež vidieť názak textúry stojacej vody.



Obrázok 7.16: Premietanie obtekajúcej vody na objekt

7.2 OpenCV

Funkcie počítačového videnia boli zväčša odvodené od knižnice OpenCV, verzie 2.4.9. Táto knižnica je šírená pod licenciou BSD. [25]
Najdôležitejšie použité funkcie sa týkali:

- detekcie šachovnicového vzoru a jeho zobrazenia:
    - `cv::findChessboardCorners` detekuje čierne štvorce šachovice a ukladá pozície ich rohov na základe známych rozmerov. Treba zmieniť, že keďže táto funkcia hľadá tmavé štvorce, je výhodné a zväčša aj nutné pri premietanom vzore používať inverzný obraz, pretože z fyzikálneho princípu vyplýva, že tmavé štvorce šachovnice sú na obraze z kamery vo farbe premietacej plochy ktorá býva relatívne svetlá, zatiaľ čo svetlé štvorce sú na obraze veľmi jasné až biele. [7]
    - `cv::cornerSubPix` spresní polohy už detekovaných rohových bodov [26]
    - `cv::drawChessboardCorners` vykreslí tieto rohy spolu s úsečkami značiacimi ich poradie [7]
- výpočtov v trojrozmernom priestore a transformácii bodov
    - `cv::calibrateCamera` získa kalibračné parametre kamery a v našom prípade aj projektora na základe zoznamu priestorových bodov a im zodpovedajúcich obrazov na kamere [7]
    - `cv::solvePnP` určí rotáciu a posuv nulového bodu objektových súradníc voči kamere, pri znalosti kalibračných parametrov tejto kamery [7]
    - `cv::projectPoints` generuje polohy obrazov zadaných trojrozmerných bodov na kamere zadaných parametrov [7]
    - `cv::composeRT` Kombinuje dva páry transformačných a rotačných vektorov do jednej takejto dvojice [7]
    - `cv::Rodrigues` umožňuje konvertovať rotačný vektor skladajúci sa z troch súradníc na rotačnú maticu a naopak [7]
    - `cv::undistortPoints` upraví zoznam súradníc bodov na kamere tak, aby z nich bol odstránený vplyv známeho skreslenia objektívu [27]

7.3 OpenGL

Pre zobrazovanie pokročilejšej 3D grafiky bolo potrebné použiť grafické rozhranie. Zvolil som si OpenGL API, pretože nezávislý programátori sú pri ňom oslobodení od licenčných povinností [22].
Aby som udržal kompatibilitu s mojim procesorom, používal som vždy OpenGL verzie 4.2. Aby som mohol v programe efektívne túto grafickú knižnicu používať, využil som tri knižnice na prácu s ňou.
Prvou je FreeGLUT, open-source alternatíva pre OpenGL Utility Toolkit (GLUT), ktorá zjednodušuje prácu s OpenGL vytváraním okien, inicializáciou kontextov a podobne [28]. Oproti GLUT je knižnica FreeGLUT upravená tak, že ju je možné používať mimo hlavného vlákna programu [29]. Toto potrebujem nie len pre zlepšenie výkonu paralelizmom, ale aj pre to, že v mojom programe musí okno s 3D grafikou existovať na pozadí, umožňujúc takto súčasný beh počítačového videnia v reálnom čase a užívateľský zásah.
Druhou z týchto knižníc je GLEW, alebo The OpenGL Extension Wrangler Library. Táto open-source knižnica mi poskytuje prístup k funkciám jadra a rozšírení OpenGL [30].
Poslednou knižnicou ktorú som s OpenGL používal je SOIL, alebo Simple OpenGL Image Library [31]. Táto knižnica mi výrazne zjednodušila načítavanie obrázkových súborov do OpenGL pre účey textúrovania.
Štruktúra grafického enginu ktorý je v programe implementovaný je inšpirovaná postupom popísaným na stránkach in2gpu.com [32].

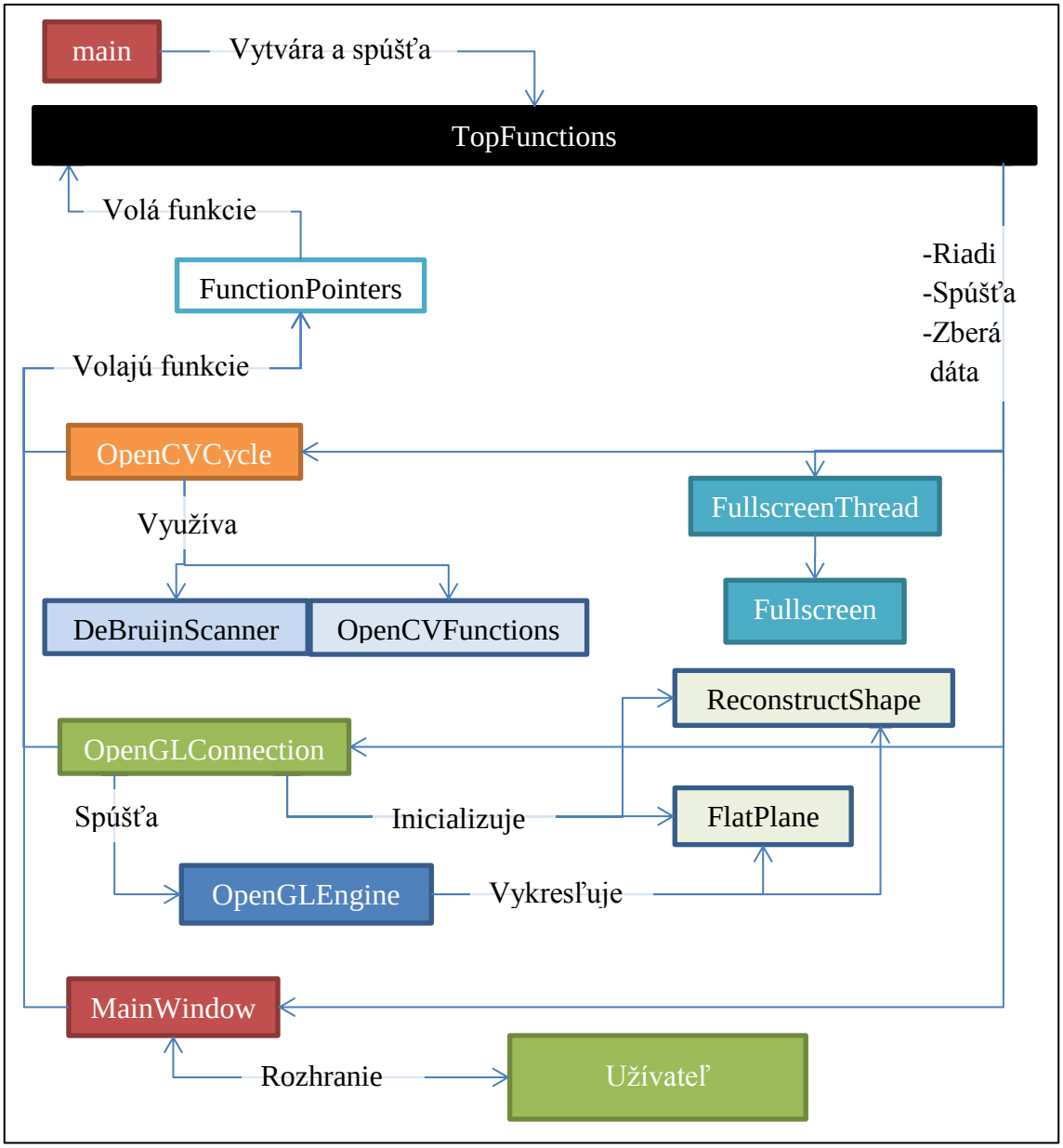
7.4 Zdrojový kód

V tejto časti popisujem niektoré časti zdrojového kódu, a pre vybrané zaujímavé funkcie vkladám celý ich zdokumentovaný text. Z praktických dôvodov tu nie je obsiahnutý celý zdrojový kód programu. Ten je priložený na sprievodnom disku.
Zdrojový kód je rozdelený na viacero súborov:

- Hlavné:
    - main.cpp – obsahuje vytvorenie inštancie TopFunctions a spustenie hlavného okna.
    - TopFunctions.h a TopFunctions.cpp – obsahujú globálne dostupné funkcie pre všetky významné časti kódu.
    - FunctionPointers.h – obsahuje triedu FunctionPointers, ktorá obsahuje ukazovatele na funkcie, ktoré smerujú na dôležité funkcie v TopFunctions. Pomocou týchto ukazovateľov k hlavným funkciám pristupujú ostatné časti kódu.
    - MeshPoints.h – obsahuje štruktúru meshPoint, ktorá reprezentuje jeden vrchol v sieti ktorá reprezentuje nasnímaný 3D objekt.
    - MainWindow.h a MainWindow.resx - obsahuje grafický návrh hlavného okna programu, spolu s metódami volanými udalosťami v grafickom prostredí a pri štarte.
    - FullScreen.h a FullScreen.resx - obsahuje grafický návrh okna pre zobrazenie obrazu na celú plochu vybranej obrazovky, spolu s metódami volanými udalosťami v grafickom prostredí a pri štarte.
    - FullscreenThread.h – obsahuje defníciu vlákna na ktorom beží zobrazovanie okna definovaného v súboroch FullScreen.
- Položky OpenCV:
    - OpenCVFunctions.cpp a OpenCVFunctions.h – obsahujú špecializované funkcie pre počítačové videnie, zväčša využívajúce funkcie OpenCV.
    - OpenCVCycle.cpp a OpenCVCycle.h – obsahujú základný cyklus počítačového videnia pre všetky fázy kalibrácie a zobrazovania.
    - DeBruijnScanner.cpp a DeBruijnScanner.h – obsahujú funkcie, ktoré sú potrebné na generovanie vzorov pre 3D rekonštrukciu objektu, ako aj funkcie s ktorých pomocou tento objekt rekonštruujeme zo snímky.
- Položky OpenGL:
    - OpenGLEngine – pripojený projekt, ktorý zaobaľuje grafický engine [33] ktorý je v hlavnom projekte používaný pre prípravu modelov a zobrazovanie. Patria pod neho tieto súbory:
        - Engine.cpp a Engine.h – spúšťajú FreeGLUT a GLEW, ako aj správcov scény, shaderov a modelov.
        - Scene_Manager.cpp a Scene_Manager.h – obsahuje správcu scény. Je tu implementovaná tvorba intrinzickej matice projektora pre OpenGL.

- Shader_Manager.cpp a Shader_Manager.h – správca shaderov, ktorý načítava a spúšťa zvolený vrcholový a fragmentový shader.
- Models_Manager.cpp a Models_Manager.h – správca modelov, ktorý pridáva modely do scény a spúšťa ich vykresľovanie.
- Model.cpp a Model.h – abstraktná trieda z ktorej sú odvodené použité 3D modely DlatPlane a ReconstructShape (nižšie)
- ISceneObject.h – abstraktná trieda objektov z ktorej je odvodená trieda Model. Pracuje s ňou správca modelov.
- VertexFormat.h – obsahuje štruktúru, ktorá nesie informácie o 3D vrchole, vrátane pozície, normály, farby a textúrových koordinát.
- Init_GLUT.cpp a Init_GLUT.h – trieda pre inicializáciu GLUT, ktorá tiež spúšťa GLEW
- Init_GLEW.cpp a Init_GLEW.h – trieda pre inicializáciu GLEW
- IListener.h – trieda ktorá spúšťa jednotlivé časovo závislé funkcie keď dostane signál od OpenGL. Je od nej odvodený správca scény. Takáto funkcia je napríklad zobrazenie aktuálnej snímky.
- WindowInfo.h – štruktúra ktorá popisuje vlastnosti okna do ktorého vykresľujeme.
- ContextInfo.h – štruktúra ktorá udáva používanú verziu OpenGL.
- FrameBufferInfo.h – štruktúra ktorou nastavujeme vlastnosti vykresľovania, ako je používanie dvojitého bufferu, farebná hĺbka alebo antialiasing

- OpenGLConnection.cpp a OpenGLConnection.h – obsahuje funkcie, ktoré spúšťajú vlákno vykresľovania v prostredí OpenGL pre jednotlivé úlohy, a umožňujú komunikáciu s týmto vláknom
- FlatPlane.cpp a FlatPlane.h – obsahujú triedu, popisujúcu textúrovaný obdĺžnik ktorý je umiestňovaný na pohyblivú premietaciu dosku.
- ReconstructShape.cpp a ReconstructShape.h – obsahujú triedu, ktorá zo série bodov v priestore zostaví sieť textúrovaných trojuholníkov, ktorá aproximuje nasnímaný objekt.



Obrázok 7.17: Zjednodušená schéma súhry modulov programu

7.4.1 TopFunctions

Tieto súbory obsahujú užitočné funkcie ku ktorým môže byť výhodné mať prístup zo všetkých ostatných súčastí programu. Hlavičkový súbor zapuzdruje tieto funkcie ako metódy triedy `TopFunctionsUnmanaged`. Inštancia tejto triedy je obsiahnutá ako statický člen „managed“ triedy `TopFunctions`. Dôvodom je, že chcem mať stály prístup k premennej, reprezentujúcej hlavné okno. Toto okno je štandardný formulár systému Windows, a je ako taký typu „ref class“, alebo „CLR class“, nedá sa teda zaradiť ako členská premenná štandardnej C++ triedy. CLR triedy sú „managed“, čo pre moje účely znamená napríklad to, že ukazovatele na členov inštancie triedy sa

neodkazujú stále na rovnaké miesto v pamäti. Keďže tu s ukazovateľmi budem pracovať, tomuto zamedzím tak, že všetky funkcie ktoré potrebujem inde používať, sú statické metódy statického člena triedy TopFunctions.
Tieto funkcie zdieľam s ostatnými časťami kódu cez objekt typu `FunctionPointers`, ktorý obsahuje ukazovatele na funkcie. Tie sú v konštruktore `TopFunctionsUnmanaged` cez metódu `SetUpFunctionPointers()` naplnené adresami dôležitých funkcii. V hlavičke FunctionPointers.h, kde je trieda pre zdieľanie funkcii popísaná, nie je zahrnutých mnoho hlavičkových knižníc, čo odstraňuje nutnosť mnohonásobného vkladania veľkých knižníc pri každom výskyte v kóde. Sú tu zahrnuté štandardné knižnice „std::string“ a „std::vector“, ako aj „MeshPoints.h“ popísaný vyššie, a jadro OpenCV „opencv2\core\core.hpp“. Toto jadro je potrebné, pretože niektoré z funkcii používajú typ „`cv::Mat`“ ako parametre. Ostatné použité knižnice sú zahrnuté v súbore TopFunctions.h a TopFunctions.cpp.
`TopFunctionsUnmanaged` nesie aj stavové premenné a iné hodnoty používané naprieč programom. Tieto premenné sú napríklad spoločný objekt pre vykonávanie cyklu počítačového videnia, objekty pre spúšťanie vláken 3D vykresľovania, rozlíšenie obrazoviek, získané parametre kamery a projektora, ako aj indikátory hovoriace o splnení určitých podmienok. Tieto premenné sú privátne pre `TopFunctionsUnmanaged`.
Funkcie tu obsiahnuté sa týkajú spúšťania jednotlivých fáz kalibrácie a počítačového videnia, zobrazovania obrázkov, ukladania a načítavania súborov so získanými intrinzickými a extrinzickými parametrami kamery a projektora, spúšťania 3D vykresľovania a komunikácie so samostatnými vláknami ako aj nastavovanie a čítanie privátnych hodnôt.

7.4.2 OpenCVFunctions

Tieto súbory obsahujú funkcie využívajúce OpenCV, ktoré sú využívané najmä v hlavnom cykle počítačového videnia. Všetky sú zapuzdrené do menného priestoru OCVFunctions. Medzi tieto funkcie patrí načítavanie maticových dát zo súborov a ich ukladanie do súborov. Taktiež sú tu funkcie pre generovanie obrazu šachovnice a zoznamov rohových bodov šachovníc so známymi rozmermi, a to v dvojrozmernom aj trojrozmernom priestore, ktoré sa využívajú pri kalibrácii.
Sú tu taktiež funkcie využívajúce knižnicu „PvApi.h“ ktorá umožňuje získavať obraz z priemyselnej kamery Allied Prosilica GC1290C pripojenej cez Ethernetový kábel, ktorá bola pri testovaní použitá. [33]
Taktiež zahrnuté sú aj funkcie pre skladanie priestorových transformácii a výpočet spätnej transformácie z doprednej ako aj funkcia pre zobrazenie trojrozmerných osí do obrazu podľa známeho natočenia, posuvu a parametrov kamery.
Zmieniť a rozviesť treba funkciu „`OCVFunctions::CalculateIntersection`“ pre výpočet polohy objektového bodu premietnutého vzoru v priestore. Táto funkcia

vypočítava priesečníky projekčných priamok a jednotlivými zvolenými bodmi premietnutého vzoru zaznamenanými na kamere, s rovinou na ktorej ležia . Funkcia prepočítava vždy jeden bod, takže je potrebné ju volať cyklicky pre prepočet všetkých detekovaných bodov v obraze. Polohu a natočenie tejto roviny voči kamere je možné získať pomocou detekcie fyzického šachovnicového vzoru na tejto rovine, ktorý v obraze z kamery taktiež musí byť viditeľný. Telo funkcie je nasledovné:

```
cv::Point3f OCVFunctions::CalculateIntersection(
            cv::Point2f CamCoord,
            cv::Mat Rotation,
            cv::Mat Translation,
            cv::Mat Im)
{
      cv::Point3f Output;

      double x = CamCoord.x - Im.at<double>(0, 2);
            //x in camera coordinates with respect to origin point
      double y = CamCoord.y - Im.at<double>(1, 2);
            //y in camera coordinates with respect to origin point
      double r = Im.at<double>(0, 0) / Im.at<double>(1, 1);
            //ratio of sy/sx
      double f = Im.at<double>(0, 0);
            //fx

      cv::Mat n; //rotated normal
      cv::Mat p; //translated origin
      cv::Mat ni = cv::Mat(3, 1, CV_64F);
            //unrotated normal - facing camera
      cv::Mat pi = cv::Mat(3, 1, CV_64F);
            //unrotated origin - in camera center

      ni.at<double>(0, 0) = 0.0;
      ni.at<double>(1, 0) = 0.0;
      ni.at<double>(2, 0) = 1.0;

      pi.at<double>(0, 0) = 0.0;
      pi.at<double>(1, 0) = 0.0;
      pi.at<double>(2, 0) = 0.0;

      cv::Mat RotMat;
      cv::Rodrigues(Rotation, RotMat);

      n = RotMat * ni;
      p = pi + Translation;

      double d =   n.at<double>(0)*p.at<double>(0) +
                   n.at<double>(1)*p.at<double>(1) +
                   n.at<double>(2)*p.at<double>(2);
      double t =
            d /
            (n.at<double>(0)*x + n.at<double>(1)*y*r + n.at<double>(2)*f);
      //parameter
      Output.x = t*x;
      Output.y = t*y*r;
      Output.z = t*f;
      return Output;
}
```

Zdrojový kód 7.1

Táto funkcia implementuje vzťah popísaný v časti 4.2. Parameter `CamCoord` je dvojrozmerný rohový bod na obraze z kamery, ktorého poloha by mala byť korigovaná odstránením skreslenia objektívu pomocou funkcie `cv::undistortPoints`. Matice `Rotation` a `Translation` hovoria o polohe a natočení premietacej roviny voči kamere. Matica `Im` je intrinzická matica kamery po odstránení skreslenia. Na začiatku sa vypočítajú premenné `x` a `y` – poloha rohového bodu voči počiatku, `r` – pomer výšky pixelov kamery k ich šírke a `f` – ohnisková vzdialenosť kamery v šírkach pixelu. V ďalšom kroku sú generované premenné `ni` a `pi`. Tieto predstavujú normálu a jeden bod roviny prechádzajúcej počiatkom súradnicového systému a kolmej na os z. `RotMat` je rotačná matica vypočítaná z rotačného vektoru `Rotation`. Normálu premietacej roviny `n` potom získame vynásobením tejto matice normálou `ni` a bod ktorým prechádza získame jednoduchým sčítaním súradníc `pi` s posuvom roviny voči kamere. Nasledujúci krok je výpočet parametra `t` ktorý vyjadruje pomer vzdialenosti obrazu bodu na projekčnej rovine kamery od ohniska voči vzdialenosti reálneho bodu v priestore od ohniska. Potom už môžeme určiť trojrozmerné súradnice reálneho bodu v ležiaceho na premietacej ploche vynásobením polohy jeho obrazu a ohniskovej vzdialenosti parametrom `t`. Tieto súradnice sú zapísané do premennej `Output` a vrátené volajúcej funkcii.

Dôležitá funkcia je aj „`OCVFunctions::GetMarkerLocations`“ ktorá hľadá štyri značky (fiducial markery), na doske z Obrázok 7.9. Jej telo je nasledovné:

```
std::vector<cv::Point2f> OCVFunctions::GetMarkerLocations(cv::Mat& input)
{
	int n = 0; //count of detected fiducial markers
	cv::Point2f pts[4]; //positions of found marker centers
	bool ptfound[4]; //state which of the four markers has been found
	for (int i = 0; i < 4; i++)
		ptfound[i] = false;	//initialize all markers as not found

	std::vector<cv::Point2f> output; //variable for the output vector

	cv::Mat image_gray;
	cv::Mat image_corners;
	cv::cvtColor(input, image_gray, CV_BGR2GRAY);
	//to get greyscale image
	cv::blur(image_gray, image_gray, cv::Size(3, 3));
	//blur to remove noise
	cv::threshold(image_gray, image_corners, 100, 255,
		cv::THRESH_BINARY | cv::THRESH_OTSU);
	//binarization by threshold gained with Otsu's method

	cv::vector<cv::vector<cv::Point> > contours;
	//holds all the contours from image
	cv::vector<cv::Vec4i> hierarchy; //holds hierarchy of contours
	cv::findContours(image_corners, contours, hierarchy,
		CV_RETR_TREE, CV_CHAIN_APPROX_SIMPLE, cv::Point(0, 0));

	std::vector<cv::RotatedRect> minRect(contours.size());
	// smallest rectangle for contour to fit in
```

```
std::vector<cv::RotatedRect> minEllipse(contours.size());
// smallest ellipse for contour to fit in
for (unsigned int i = 0; i < contours.size(); i++)
{
	double area_real; //true area of contour
	double area_estimate;
	//area of ellipse approximating the contour

	minRect[i] = cv::minAreaRect(cv::Mat(contours[i]));
	//fit the smallest possible rectangle
	area_real = abs(cv::contourArea(contours[i]));
	if (area_real >
	(input.size().height * input.size().width / 25))
		continue; // discard contours that are too big
	area_estimate = CV_PI * minRect[i].size.width *
		minRect[i].size.height / 4;
	// area of an ellipse

	double error = abs(area_real - area_estimate);
	// contour should resemble an ellipse
	if ( error < area_real*0.7)
		if ((contours[i].size() > 5) &&
		 (hierarchy[i][2] != -1) &&
		(hierarchy[hierarchy[i][2]][2] != -1))
		//the contour should be bigger than 5 pixels, and have
		at least one other contour inside of it,
		//that also has at least one contour inside of itself -
		this is true of the used markers
		{
			int j = 0;
			int hri = hierarchy[hierarchy[i][2]][2];
			//first member of the third level contours
			bool b = true;
			while (b)
			//count all contours of the same sublevel
			belonging to contour i
			{
				if (hri != -1)
				{
					j++;
					hri = hierarchy[hri][0];
				}
				else
					b = false;
			}

			minEllipse[i] =
cv::fitEllipse(cv::Mat(contours[i]));
		// fit ellipse to the accepted contour for displaying

			if (!ptfound[j - 1])
			//if marker with this number of third level
			subcontours has not been found yet
			//(should be only one of each)
			{
				pts[j - 1] = minEllipse[i].center;
				//store center of marker
				ptfound[j - 1] = true; //mark it as found
				n++; //increase number of found markers
			}

		}
```

```
	}

	for (unsigned int i = 0; i< contours.size(); i++)
	{
		cv::Scalar color = cv::Scalar(0,0,0);
		ellipse(input, minEllipse[i], color, 2, 8);
		//draw ellipse on image
	}

	if (n == 4) //return any points only if all four have been found
		for (int k = 0; k < 4; k++)
		{
		output.push_back(pts[k]);
		}

	return output; // return 4 points all an empty vector
}
```

Zdrojový kód 7.2

Matica `input` je vstupný obraz. Tento je najprv prevedený na čiernobiely funkciou `cv::cvtColor`. Následne je mierne rozostrený funkciou `cv::blur` pre potlačenie šumu. Z takto predspracovaného obrazu je získaný binárny obraz prahovaním cez funkciu `cv::threshold`. Z binarizovaného obrazu získame funkciou `cv::findContours` oddelené kontúry. Tieto postupne prechádzame a určujeme im najmenší opísaný obdĺžnik. Z jeho rozmerov určíme plochu akú by mala najväčšia elipsa do tohto obdĺžnika vpísaná a porovnávame ich s reálnou plochou kontúry. Ak je absolútny rozdiel týchto plôch menší ako 0,7 násobok reálnej plochy – hodnota získaná empiricky vykonáme ďalšie kontroly.

Tieto pozostávajú zo sledovania hierarchie. [34]

Je to teda overenie či má daná kontúra vo vnútri práve jednu ďalšiu kontúru – biely kruh vo vnútri čierneho – a či táto má práve jednu až štyri ďalšie kontúry vo svojom vnútri – malé čierne obrazce v kruhoch. Ak už obrazec s daným počtom kontúr tretej úrovne bol náhodou nájdený tak pokračujeme na ďalšiu kontúru, inak zapíšeme nález tejto kontúry do poľa `pts` a inkrementujeme premennú n. Po nájdení všetkých štyroch bodov vykreslíme odhadnuté elipsy okolo symbolov a vrátime volajúcej funkcii vektor s polohami symbolov.

7.4.3 DeBruijnScanner

V týchto súboroch sú funkcie, ktorými implementujem hľadanie hraníc premietaných farebných pásov, ako je to popísané v časti 5.1. Tieto funkcie sú zapuzdrené ako metódy triedy `ScannerObject`. Najdôležitejšou verejne prístupnou funkciou tu je `ScannerObject::GetMeshFromImage`. Tá s pomocou nasnímaného obrazu, intrinzických matíc projektora a kamery a ich vzájomnej polohy vypočíta polohy bodov v sieti na povrchu snímaného objektu, a vráti ich ako zoznam bodov s indexmi priľahlých bodov pozdĺž spoločnej svetelnej plochy a spoločnej skenovacej línie. Telo funkcie je nasledovné:

```
std::vector<meshPoint> ScannerObject::GetMeshFromImage(
	cv::Mat input, int DBSequenceLength, cv::Mat ICam, cv::Mat DCam,
cv::Mat IProj, cv::Mat DProj, cv::Mat TProj, cv::Mat RProj, double alpha,
double beta)
{
	clampAlpha = alpha;
	clampBeta = beta;  //store parameters of consistency function

	GenerateDeBruijnGradients(DBSequenceLength);
	//prepares the list expected unit transitions to be projected
	stripeWidth = input.size().width / DBSequenceLength;
	//width of one projected stripe
	cv::Mat imageGradient = cv::Mat(input.size().height,
	input.size().width, CV_16SC3);
	//stores the image of color gradients between two adjacent pixels
	along the horizontal axis
	cv::Mat imageSquareGrad =
		GenerateGradientImage(input, imageGradient);
	//generates an image of sums of squares of gradients for each of the
	 RGB channels
	//also populates the imageGradient
	cv::Mat imageBorders =
		GetLocalMaxima(imageSquareGrad, DBGradientThreshold);
	//finds the stripe transitions as the local maxima of
	imageSquareGrad along the horizontal axis

	cv::imwrite("borders.bmp", imageBorders); // for analysis

	std::vector<int> lineBorders;
	//holds the x coordinates of border pixels, along the current
	 scanline
	std::vector<cv::Vec3d> lineBorderGradients;
	//holds color gradients for each of the border pixels
	std::vector<cv::Point3f> discoveredPoints;
	//holds the x and y position of analyzed border points,
	//and also the j - index of the identified projected transition

	cv::Mat coloredBorders = cv::Mat(imageBorders.size(), CV_8UC3);
	//for analysis

	std::vector<cv::Vec3b> colors =
		GenerateDeBruijnColors(DBSequenceLength);
	//generates a list of colors from the De Bruijn sequence

	LineJump = stripeWidth/2; // for performance purposes

	for (int y = 0; y < imageBorders.size().height; y++)
	//for every line of image
	{
		if ((y % LineJump) != 0)
			continue; //scanning every line takes too much time

		std::cout << y << std::endl; //current line

		lineBorders.clear();
		lineBorderGradients.clear();

		for (int x = 0; x < input.size().width - 1; x++)
		{
			if (imageBorders.at<unsigned char>(y, x) > 0)
			{
				lineBorders.push_back(x);
```

```
                        lineBorderGradients.push_back(
                ((cv::Vec3d)imageGradient.at<cv::Vec3s>(y, x)) / 255.0);
                        //populate the lists
                    }
                }
                if (lineBorders.size() == 0)
                    continue; //nothing to work with

                scoreGrid =
                    cv::Mat(lineBorders.size(), DBSequenceLength, CV_64F);
                //grid for optimal path search
                populateScoreGrid(lineBorderGradients, false);
                //fill the grid with consistency values based on projected (j)
                 and obtained (i) transitions
                std::vector<correspondence> foundCorrespondences =
                    Match(scoreGrid);
                //fill foundCorrespondences with i-s and j-s of corresponding
                 transitions
                //The Match function finds the optimal path through the
                scoreGrid

                for (int i = 0; i < foundCorrespondences.size(); i++)
                {
                cv::Scalar color = cv::Scalar(
                            colors.at(foundCorrespondences.at(i).j).val[0],
                            colors.at(foundCorrespondences.at(i).j).val[1],
                            colors.at(foundCorrespondences.at(i).j).val[2]);
                    cv::Rect rec =
cv::Rect(lineBorders.at(foundCorrespondences.at(i).i) - 1, y - 1, 3, 3);
                    cv::rectangle(coloredBorders, rec, color, CV_FILLED);
                    //draw square of the color of the supposedly preceding
                    stripe

                    discoveredPoints.push_back(
                        cv::Point3f(
                        lineBorders.at(foundCorrespondences.at(i).i),
                        y,
                        foundCorrespondences.at(i).j));
                    //add the coordinates and transition indices j to the
                     vector
                }
                foundCorrespondences.clear();
            }
            std::cout << "Done." << std::endl;

            std::vector<cv::Point3f> recPoints =
                reconstructPoints(discoveredPoints, ICam,IProj,TProj,RProj );
            //estimate 3D positions of the points through line-plane
             intersections

            cv::imwrite("ColoredBorders.bmp", coloredBorders);
            //for analysis of flaws in reconstruction
            std::cout << "Scanning done." << std::endl;

            return FindAdjacencies(discoveredPoints,recPoints);
            //returns points with indices of next and previous points
            //along common transition and common scanline, if they are adjacent
            in sequence,
            //skipped transitions and scanlines mean missing adjacency
}
```

Zdrojový kód 7.3

Táto funkcia implementuje postupy popísané v častiach 5 a 5.1, a jej výstupný vektor obsahuje nájdené 3D body aj s ich priľahlými bodmi, čo je príprava na rekonštrukciu povrchu trojuholníkovou sieťou, ako je popísaná v 5.2. Funkcia je volaná nepriamo z OpenCVCycle (7.4.4) a využíva viaceré niektoré funkcie z OpenCVFunctions (7.4.2) a viaceré z DeBruijnScanner.

Najdôležitejšou z týchto je funkcia `ScannerObject::Match` ktorá spustí rekurzívne hľadanie optimálnej cesty po `scoreGrid`, čím priradí časti nájdených prechodov z obrazu index premietaného prechodu ktorý by mu mal zodpovedať. Toto vykonáva spustením ďalšej funkcie – `ScannerObject::optPathCost` – s počiatočnou pozíciou v mriežke na indexoch [M,N], teda posledný nájdený a premietnutý prechod. Funkcia `ScannerObject::optPathCost` opakovane volá sama seba a obsahuje moju optimalizáciu popísaná v časti 5.1.

Toto je telo funkcie:

```
double ScannerObject::optPathCost(int i, int j)
{
	double competingScores[3];
	//scores from each of the three possible nex steps in path
	double *optimalScore;
	//holds best score the possible steps

	if ((i == 0) || (j == 0))
	{
		return 0.0;
		//we stop when we run out of found or projected transitions
	}
	else
	{
		if (exploredIndices.at<unsigned char>(i, j) != 0)
		{
			return costGrid.at<double>(i, j);
			//my optimization - exploredIndices is 0 for positions
			in grid
			//which were not evaluated yet.
			//cost grid stores the previosly calculated values
		}
		else
		{
			competingScores[0] =
				optPathCost(i - 1, j - 1) +
				scoreGrid.at<double>(i, j);
			competingScores[1] = optPathCost(i, j - 1);
			competingScores[2] = optPathCost(i - 1, j);
			//we recursively evaluate paths in three possible
			directions
			//each of them branches to three directions again and
			so on,
			//until we reach state of (i == 0) || (j == 0)

			optimalScore =
			std::max_element(competingScores, competingScores + 3);
			//we pick best of the possible scores
			costGrid.at<double>(i, j) = *optimalScore;
			//we store current score, so that we will not have to
			iterate from this point again
```

```
                    exploredIndices.at<unsigned char>(i, j) =
                          (optimalScore - competingScores + 1);
                    //we store the directon towards the best path to this
                     point.
                    //1 - diagonal, 2 - towards lower j, 3 -towards lower i
                    //after the recursion stops, we trace optimal path from
                    [M,N] along these directions

                    return costGrid.at<double>(i, j);
               }
          }
}
```

Zdrojový kód 7.4

Po tom ako sa vráti výsledok prvého volania funkcie, môžeme nájsť optimálnu cestu mriežkou podľa hodnôt v `exploredIndices`, pričom tam kde je hodnota 1 sme našli zhodu medzi nasnímaným a premietaným prechodom.

To, že `scoreGrid` a `exploredIndices` sú matice OpenCV prináša výhodu pri testovaní a hľadaní chýb, vďaka možnosti jednoduchého zobrazenia a uloženia do súboru, ako je to popísané v časti 0.

7.4.4 OpenCVCycle

Tieto súbory implementujú cykly načítavania obrazu z kamery a jeho zobrazovanie a spracovanie ideálne v reálnom čase.

Všetky typy a funkcie sú zapuzdrené do menného priestoru „`CVCycle`“. Tento priestor obsahuje vymenovací typ `calib_phase` ktorý hovorí o aktuálnej fáze kalibrácii alebo premietania. Je tu tiež definovaná štruktúra `CaptureParams` ktorá sprostredkuje komunikáciu medzi hlavným cyklom a vláknom záznamu z kamery. Najdôležitejšou položkou je trieda `CVCycleObject` ktorá obsahuje dátové štruktúry a premenné s ktorými sa pracuje v hlavnom cykle a metódy popisujúce spracovanie obrazu v jednotlivých fázach používania programu, ktoré sú cyklicky volané metódou `cv_cycle`. Táto funkcia najprv vykoná potrebné inicializácie premenných a kamery podľa užívateľských nastavení. Je možné používať štandardnú webkameru, ako aj priemyselnú kameru pripojenú cez gigabitový Ethernetový kábel v prípade že je na danom počítači nainštalovaný príslušný driver. Je tu aj možnosť snímať z webového streamu určeného pomocou adresy URL. Táto možnosť bola využitá pre testovacie účely spôsobom popísaným v časti 0.

Po inicializácii je spustené vlákno ktoré voľne beží a čerpá obrazy z vybranej kamery. Toto zabraňuje hromadeniu obrazov vo vyrovnávacej pamäti, ktoré inak nastáva pri pomalom spracovaní obrazu, a ktoré má za následok postupne rastúce zaostávanie spracovaného obrazu za realitou. Pre tento účel sú definované dve vlákna. Jedno je určené pre prácu z obrazom z webkamery alebo z webového streamu, druhé pre prácu s priemyselnou kamerou Allied pomocou funkcii z OpenCVFunctions využívajúcich knižnicu „PvApi.h“. Vlákno komunikuje s hlavným cyklom pomocou spoločnej štruktúry typu `CaptureParams` ktorá obsahuje ukazovateľ na kameru, zachytený obraz a flag ktorý hovorí o tom, či je žiaduce aby vlákno pokračovalo. Aby sa predišlo

chybám spôsobeným spoločným prístupom k týmto dátam, je prístup ošetrený mutexom `mtx` ktorý musí byť pre cyklus alebo vlákno uvoľnený predchádzajúcim.
Po získaní aktuálnej snímky, ktorá je zapísaná do premennej `image_original` je táto snímka spracovaná jednou z špecializovaných metód, podľa hodnoty vstupnej premennej typu `calib_phase`. Po spracovaní je pomocou funkcie `cv::imshow`, ktorú poskytuje OpenCV modul HighUI, zobrazený obraz z premennej `image_processed` ktorý vznikol úpravou `image_original`. Po stlačení tlačidla Esc je cyklus ukončený, potom ako je zmenou flagu ukončené snímacie vlákno.
Špecializované metódy spracovania využívajú funkcie OpenCV zmienené v 0, ako aj rozšírené funkcie zo súborov OpenCVFunctions. Každá z metód vykonáva príslušnú prácu s obrazom, ale taktiež upravuje obraz `image_processed` ktorý je zobrazovaný užívateľovi a takto podáva spätnú väzbu, pomocou ktorej môže nastavovať smer a polohu kamery a projektora, ako aj svetelné podmienky.
Tieto metódy sú:
`CVCycleObject::CameraCalibCycle` – Zaznamenáva rohové body šachovnice a po získaní dostatočného počtu zoznamov bodov, určeného konštantou `MaxCamCalibFrames`, vykoná kalibráciu kamery s využitím známych rozmerov šachovnice. Pridanie aktuálne pozorovaného vzoru sa vykoná vždy po stlačení tlačidla Enter.
`CVCycleObject::ProjectorCalibCycle` – Detekuje fyzický šachovnicový vzor, ktorým určuje polohu premietacej roviny, a zároveň detekuje premietnutý vzor na tejto rovine. Detekcia premietaného vzoru vyžaduje inverziu obrazu. Pomocou známych parametrov kamery a polohy roviny prepočíta rohové body premietnutého vzoru do priestoru, pričom sú popísané v súradnicovom systéme kamery. Pridanie aktuálne pozorovaného vzoru sa vykoná vždy po stlačení tlačidla Enter. Po získaní dostatočného počtu zoznamov takýchto prepočítaných bodov, určeného konštantou `MaxProjCalibFrames`, vykoná kalibráciu projektora, s využitím známeho rozlíšenia monitora predstavujúceho projektor a veľkosti štvorcov v pixeloch. Výpočet parametrov projektora vyžaduje úvodný odhad, pretože priestorové body nie sú dokonale rovine rovnobežnej s dvoma osami ich objektového súradnicového systému. Odhad intrinzickej matrice ktorý používam je optické centrum v strede premietaného obrazu, štvorcové pixely a ohnisková vzdialenosť rovná 2,5 násobku výšky obrazu v pixeloch, čo som odhadol pohľadom na premietajúci projektor najčastejšie používaný pri testoch.
`CVCycleObject::ProjectorPoseCycle` – Určí vzájomnú polohu kamery a projektora. Vyžaduje aby podobne ako v predchádzajúcej metóde boli viditeľné oba šachovnicové vzory a aby bola kamera, ako aj projektor skalibrované.
`CVCycleObject::DetectPlateCycle` – ak sú kamera aj projektor skalibrované a poznáme aj ich vzájomnú polohu, potom táto metóda zaháji jednu z dvoch ukážkových aplikácii založených na sledovaní priestorovej polohy špeciálnej platne popísanej v časti 7.1, viditeľnú na Obrázok 7.9.

Ak je vstupný parameter `bool useOpenGL` rovný `false`, potom sa premietne na platňu deväť bodov, ktoré sú usporiadané do štvorca so stranami rovnobežnými so stranami dosky a majú na doske stále rozstupy veľkosti 50mm. Využíva pritom funkcia `GetMarkerLocations` z OpenCVFunctions popísaná v 7.4.2 na detekciu symbolov na doske. Polohy symbolov tvoria premietnutý obdĺžnik, ktorého reálne rozmery poznáme, a teda môžeme funkciou `cv::solvePnP` a kalibračnými parametrami kamery určiť natočenie a posuv dosky v koordinátach kamery. So znalosťou vzájomnej polohy projektora a kamery je prepočítaná poloha tejto dosky do koordinát projektora. Funkciou `cv::projectPoints` s parametrami projektora potom vypočítame polohy zmienených deviatich bodov v jeho obraze, pričom sú body pôvodne v rovine posunuté a natočené voči projektoru rovnako ako doska. Na tieto polohy v obraze vykreslíme kružnice, a vzniknutý obraz zobrazíme projektorom.
V prípade že je vstupný parameter `bool useOpenGL` rovný `true`, potom sa na dosku premietne obdĺžnik s textúrou. Tento obdĺžnik má rozmery 150 mm x 100 mm, a program ho umiestňuje vždy tak, aby sa z pohľadu kolmo na dosku javil na rovnakom mieste a neskreslený perspektívnym zobrazením., teda strany tohto obdĺžnika zostanú rovnobežné so stranami dosky pri zmene jej polohy a natočenia a nezmení sa ani pomer strán. Funkcia takto využíva OpenGL na úlohu popísanú v častiach 6.1 a 7.1.
`CVCycleObject::ScanShapeCycle` – ak sú kamera aj projektor skalibrované a poznáme aj ich vzájomnú polohu, potom táto metóda spustí skenovanie tvaru objektu ako je popísané v 5.1, 5.2 a 7.4.3. Na ploche projektora sa premietne De Bruijnova sekvencia farieb zvoleného počtu pásov a spustí sa zobrazovanie obrazu z kamery. Ak sa nám obraz osvetleného objektu javí vhodný pre skenovanie, stlačením tlačidla Enter sa z aktuálnej snímky prepočítajú polohy povrchových bodov a uložia sa do premennej `std::vector<meshPoint> Mpts` v `TopFunctionsUnmanaged` pre ďalšie využitie.

7.4.5 OpenGLConnection a 3D modely

Súbory OpenGLConnection umožňujú spustiť samostatné vlákno pre jednu z úloh na báze OpenGL. Pred spustením vlákna sa naplní štruktúra `GlShapeThreadParams` alebo `GlPlaneThreadParams` intrinzickými parametrami projektora, ako aj ukazovateľom na inicializovanú inštanciu vybraného typu 3D objektu a v prípade vlákna pre 3D rekonštrukciu objektov aj vektorom `std::vector<meshPoint> Mpts` získaný z `CVCycleObject::ScanShapeCycle` (7.4.4). Ukazovateľ je na štruktúru s parametrami je podaný vláknu, ktoré je potom spustené. V týchto vláknach sa inicializuje grafický engine s použitím intrinzických parametrov projektora, vkladajú sa do neho textúrové súbory následne je spustený. Vlákno má s hlavným vláknom zdieľaný mutex `gl_mtx`. Ten zabezpečuje, že ak chceme vykonať nejakú zmenu v používanom 3D objekte, nezasiahneme do prebiehajúcich procesov vykonávaných enginom.

Toto je využité vo funkcii SetPose, ktorá nastaví danému 3D objektu zvolenú pozíciu vo formáte matice 4x4, ktorá je vytvorená z vektorov pozície a natočenia ako ich udáva OpenGL. Táto konverzia sa vykonáva vo funkcii pre rekonštruovaný tvar `TopFunctionsUnmanaged::SetShapeRT` ,respektíve pre textúrovaný obdĺžnik v `TopFunctionsUnmanaged::SetPlaneRT`.

Súbory FlatPlane popisujú správanie 3D textúrovaného obdĺžnika, ktorý v jednej z demonštračných funkcii umiestňujeme a natáčame podľa fyzickej platne (6.1). Toto je zapuzdrené v triede `FlatPlane` odvodenej z triedy `Model` (0). Metóda `FlatPlane::Create` vytvorí 3D model obdĺžnika. Metóda `FlatPlane::Draw` je volaná správcom modelov vždy, keď OpenGL upovedomí správcu scény. Táto metóda vykreslí obdĺžnik. Pri tom sú používané tri transformačné matice, ktoré aplikuje vrcholový shader:

- `view_matrix` – určuje polohu virtuálnej kamery (projektora) voči nulovému bodu scény. Pre moje účely ju nechávam ako jednotkovú maticu, ktorú nič nemení
- `projection_matrix` – túto maticu skladám pri „Reshape“ signáli z OpenGL spôsobom uvedeným v časti 0.
- `pose_matrix` – táto matica je deklarovaná v predkovi triedy, teda v triede `Model`. Funkciou `SetPose` v OpenGLConnection ju môžem meniť, a tak polohovať obdĺžnik na požiadanie.

Súbory ReconstructShape sú podobne odvodené z triedy `Model` a popisujú rekonštruovaný povrch skenovaného objektu. Štruktúra `ReconstructShape` sa ale líši tým, že obsahuje metódy na export tvaru do OBJ súboru [24], funkciu na výpočet normály trojuholníka metódou popísanou v časti 5.2, a tým že v metóde `ReconstructShape::Create` vytvára trojuholníkovú sieť spôsobom ktorý je popísaný v tej istej časti, vrátane jej vrcholových normál. Vstupom do funkcie je vektor povrchových bodov získaný pri skenovaní objektu. Telo tejto funkcie vyzerá takto:

```
void ReconstructShape::Create(std::vector<meshPoint> mpts)
{
	mouseX = 0;
	mouseY = 0;
	//mouse rotation or translation - for testing purposes

	std::vector<VertexFormat> vertices;
	//list of all used vertices
	std::vector<unsigned int> indices;
	//list of indices in order in wchich they form consecutive triangles

	for (int i = 0; i < mpts.size(); i++)
	//for each of the identified surface points
	{
		vertices.push_back(VertexFormat(
			glm::vec3(
				mpts.at(i).Current.x*0.001,
```

```
                        mpts.at(i).Current.y*(-0.001),
                        mpts.at(i).Current.z*(0.001)),
                  glm::vec2(mpts.at(i).Current.z*0.001, 0)));
      //insert vertex position in milimeters, as well as the z axis
      as texture coordinate for testing. Y-axis is flipped in OpenGL

      if ((mpts.at(i).prevX != -1) && (mpts.at(i).prevY != -1) )
      {
            indices.push_back(mpts[i].prevX);
            indices.push_back(mpts[i].prevY);
            indices.push_back(i);
            //if possible, create upper-left triangle
      }

      if ((mpts.at(i).nextX != -1) && (mpts.at(i).nextY != -1))
      {
            indices.push_back(mpts[i].nextX);
            indices.push_back(mpts[i].nextY);
            indices.push_back(i);
            //if possible, create lowerr-right triangle
      }
}

std::vector<glm::vec3> currIndexNormals;
//holds unnormalized normals for all triangles connected to current
vertex
std::vector<std::vector<glm::vec3>> normalsPerIndex;
//holds the above for each of the vertices

for (int j = 0; j < vertices.size(); j++)
{
      currIndexNormals.clear();
      for (int i = 0; i < indices.size(); i++)
      {
            if (indices[i] == j)
            {
                  int triangle = i / 3; //index of current triangle

                  glm::vec3 v1
                   vertices[indices[triangle * 3]].position;
                  glm::vec3 v2 =
                  vertices[indices[triangle * 3 + 1]].position;
                  glm::vec3 v3 =
                  vertices[indices[triangle * 3 + 2]].position;
                  //vertices of current triangle

                  glm::vec3 normal = GetNormalFromTri(v1, v2, v3);
                  //calculates unnormalized normal of the triangle
                  //returns [0,0,0] for degenerate triangles, where
                   all vertices are in line

                  if (normal != glm::vec3(0, 0, 0))
                  {
                        currIndexNormals.push_back(normal);
                        //if triangle was not degenerate, store the
                        normal
                  }
            }
      }
      normalsPerIndex.push_back(currIndexNormals);
      //store each adjacent triangle normal
}
```

```
	glm::vec3 sum;
	for (int i = 0; i < normalsPerIndex.size(); i++)
	{
		sum = glm::vec3(0, 0, 0);
		for (int j = 0; j < normalsPerIndex[i].size(); j++)
		{
			sum = sum + normalsPerIndex[i][j];
			//summarize all the adjacent unnormalized normals
		}
		vertices[i].normal = glm::normalize(sum /
(float)normalsPerIndex[i].size());
		//now we divide by their count and normalize the vertex normal
	}

	indexCount = indices.size();
	//for use in Draw function

	std::cout << "Total triangles: " << indices.size() / 3 << std::endl;

	OutputToFile(vertices, indices);
	//export to Wavefront OBJ file.

	GLuint vao, vbo, ibo;
	//vertex array object, vertex buffer object and index buffer object

	glGenVertexArrays(1, &vao);
	glBindVertexArray(vao);
	//bind the vertex array object

	glGenBuffers(1, &vbo);
	glBindBuffer(GL_ARRAY_BUFFER, vbo);
	//bind the vertex buffer object
	glBufferData(GL_ARRAY_BUFFER, vertices.size() *
		sizeof(VertexFormat), &vertices[0], GL_STATIC_DRAW);
	//store the vertices in the vertex buffer

	glGenBuffers(1, &ibo);
	glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ibo);
	//bind the index buffer object
	glBufferData(GL_ELEMENT_ARRAY_BUFFER, indices.size() *
		 sizeof(unsigned int), &indices[0], GL_STATIC_DRAW);
	//store the indices in the index buffer

	glEnableVertexAttribArray(0);
	glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE,
		sizeof(VertexFormat), (void*)0);
	glEnableVertexAttribArray(1);
	glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE,
		sizeof(VertexFormat),
		(void*)(offsetof(VertexFormat, VertexFormat::texture)));
	glEnableVertexAttribArray(2);
	glVertexAttribPointer(2, 3, GL_FLOAT, GL_FALSE,
		sizeof(VertexFormat),
		(void*)(offsetof(VertexFormat, VertexFormat::normal)));
	//Create separate attribute pointers for properties of the vertices
	- the position, the texture coordnates and the normals

	glBindVertexArray(0);

	this->vao = vao;
	this->vbos.push_back(vbo);
```

```
        this->vbos.push_back(ibo);
        //store the pointers for future use

        glutMouseFunc(ShapeMouseAction);
        //bind function for mouse clicking
        glutMotionFunc(MouseMotion);
        //bind function for mouse draging

        startTime = std::chrono::high_resolution_clock::now();
        //set up reference time for dynamic texture
}
```

Zdrojový kód 7.5

Je tu vidieť, že funkcia v prvej časti vytvára 3D geometriu rekonštruovaného tvaru vrátane vrcholových normál, a v druhej túto geometriu sprístupňuje grafickému procesoru s využitím GLEW funkcii (0).

7.4.6 Shadery

Každá z demonštračných úloh využívajúcich OpenGL má vlastné shaderové súbory.
Tieto sú tvorené párom „vertex shader“ a „fragment shader“. Vertex shader, alebo vrcholový shader, prepočítava polohy a prípadne aj iné parametre vrcholov do zobrazovacej roviny. Toto sa v programe deje jednoduchým násobením polôh vrcholov tromi maticami popísanými v časti 7.4.5. Takto prepočítané pozície sú potom predané fragmentovému shaderu. Okrem nich je možné odovzdať parametre ako sú normály povrchu, textúrovacie koordináty alebo netransformované polohy vrcholov.
Vo fragmentovom shaderi potom tieto parametre interpolované naprieč povrchmi fragmentov, čím získame plynulejšie prechody farieb alebo polohu pixelu v textúre ktorého farbu priradíme danému pixelu vo vykreslenom obraze. Ukážkou toho je fragmentový shader pre efekt tečúcej vody, implementujúci postup popísaný v časti 6.2.1. Jeho obsah je nasledovný:

```
#version 420 core

layout(location = 0) out vec4 out_color;
      //the outputted calculated color

uniform sampler2D alpha1ChanTex;
      //mask dividing calm, horizontal water from sloped flow
uniform sampler2D alpha2ChanTex;
      //mask for foam on the transition

uniform sampler2D specialTex1;
      //horizontal water surface texture
uniform sampler2D specialTex2;
      //fast flowing waterfall texture
uniform sampler2D specialTex3;
      //foam texture hiding the transition


uniform float     Timer;
      //timer updated outside of the shader
```

```
in vec2 texcoord; //texture coordinate
in vec3 normal;    //vertex normal
in vec4 position; //vertex position

void main() {
    float offset = Timer * 0.0001;
            //scale the time change down

    vec4 firstColour  = texture (specialTex1,   vec2(position.x*3,
position.z*3 + offset*3));
            //the horizontal water texture is mapped from the top of the
model, and slowly moved for illusion of flow
    vec4 secondColour = texture (specialTex2,   vec2(position.x*3,
position.y*3 + offset*100));
            //the vertical water texture is mapped from the front of the
model, and rapidly moved for illusion of fall
      vec4 thirdColour  = texture (specialTex3,   vec2(position.x*3,
position.y*3 + offset*30));
            //the foam texture is mapped from the front of the model, and
moved for illusion of rolling

    vec4 aChan1Colour  = texture (alpha1ChanTex, vec2(1,
((normal.y+1.0)*0.5) ));
            //transition from slow to fast water is mapped by the surface
inclination
    vec4 aChan2Colour  = texture (alpha2ChanTex, vec2(1,
(normal.y+1.0)*0.5 ));
            //the foam mask is mapped in the same way

    out_color = firstColour * aChan1Colour.r + secondColour * (1 -
aChan1Colour.r) + thirdColour * aChan2Colour.r ;
            //all the textures are put together
}
```

Zdrojový kód 7.6

Je tu vidieť ako sú použité normály povrchu pre simuláciu vody ktorá tečie rýchlejšie po povrchu s väčším sklonom.

7.5 Testovanie a vývoj

Po dokončení aktuálneho stavu programu, ale aj počas tvorby, som vykonával testy pre overenie funkčnosti aplikácie, pre voľbu vhodných postupov a pre vyvodenie dôsledkov do budúcnosti. Zhŕňam tu pozorovania, problémy a moje metódy riešenia z týchto testov.

Pri testovaní v domácich podmienkach som mal k dispozícii starší model projektora HP–BENQ sRGB, a dve webové kamery, z ktorých jedna je integrovaná v mojom notebooku a druhá – Creative Webcam Live!Ultra – bola zapožičaná od vedúceho práce. Pre niektoré účely ale rozlíšenie a kvalita obrazu tejto kamery neboli dostatočné, takže som sa rozhodol využiť efektívnejšie univerzálne vybavenie ktoré mám k dispozícii, konkrétne relatívne kvalitnú kameru stavanú v mojom smartfóne.
Tento cieľ som dosiahol tak, že zo smartfónu Samsung Galaxy S4 Active bol pomocou aplikácie IP Webcam [35] zasielaný aktuálny obraz z jeho kamery na WiFi sieť v podobe webvého streamu. Pre väčšiu rýchlosť prenosu som použil nahradenie WiFi káblom pomocou USB tetheringu. Toto mi sprístupnilo kvalitnú kameru s vysokým rozlíšením pre domáce testy. Záznam podľa adresy URL nebol funkčný v OpenCV verzie 3.0.0, čo je hlavný dôvod prečo som neprešiel v projekte na túto novšiu verziu.

Detekcia šachovnicového vzoru bola pomerne pomalý proces, a to nie len z dôvodu výpočtovej sily, ale aj z dôvodu nutnosti prispôsobovať svetelné podmienky kamere. Detekcia dvoch šachovnicových vzorov zároveň – ako to potrebné v mojej metóde kalibrácie projektora – bola výrazne pomalšia než detekcia iba jedného vzoru, a taktiež bola viditeľne pomalšia detekcia väčšieho vzoru – s viacerými štvorcami – voči vzoru menšiemu. Pri vývoji mi táto detekcia zaberala o to viac času, že som nepočítal s jednou dôležitou vlastnosťou funkcie `cv::findChessboardCorners` (0). Predpokladal som, že funkcia hľadá hrany medzi bielymi a čiernymi štvorcami šachovnice na základe vysokých lokálnych gradientov, a prispôsobuje hranicu jasu ktorou odlišuje čierne štvorce od bielych celkovému osvetleniu scény. V skutočnosti ale funkcia hľadá len čierne, alebo výrazne tmavé štvorce. Pokým som na tento omyl našiel, zachytávanie šachovnicového vzoru spolu s fyzickou šachovnicou v jednom obraze bolo veľmi zložité. Bolo totiž potrebné prispôsobovať osvetlenie každej z dvoch šachovníc, ako aj celej scény tak, aby sa na oboch javili tmavé štvorce približne v rovnakom odtieni, čo som ale bez znalosti tohto princípu videl len ako náhodne dosiahnutý stav kedy sú oba vzory detekované. Po odhalení mojej chyby som jednoducho pre detekciu premietanej šachovnice obraz prepočítal na inverzný (negatív). Pri jednotných svetelných podmienkach pre fyzický a premietnutý vzor, a pri bielej premietacej rovine, je premietnutý vzor vždy ten jasnejší. Z hľadiska skutočných hodnôt jasu sú „čierne“ štvorce premietanej šachovnice rovnako svetlé, alebo dokonca svetlejšie, ako biele štvorce fyzickej šachovnice.

Na inverznom obraze sa potom biele premietané štvorce, ktoré sú na kamere výrazné, zmenia na výrazne čierne štvorce ktoré funkcia vie nájsť.

Po zhromaždení potrebného počtu vzorov bola výpočtová náročnosť určenia parametrov kamery aj projektora tak veľká, že program zostal stáť niekedy až na 5 až 10 sekúnd.
Toto ale nie je samostatne problematické, pretože kalibrácie je teoreticky potrebné vykonávať iba raz. Zhromažďovanie vzorov pre kalibráciu projektora náročnejšie aj z toho dôvodu, že bolo potrebné nie len snímať vzory z viacerých uhlov, ale aj premietať z rôznych uhloch pre čo najlepšiu kalibráciu. Tento proces bol celkovo zdĺhavý. Pri používaní staršieho projektora značky HP dokonca nastával problém, kedy bol niekedy v dôsledku častého premiestňovania projektora rozpájaný elektrický kontakt v zariadení, čo spôsobovalo okamžité vypnutie zariadenia. Zariadenie sa dalo znova spustiť natočením do vhodnej polohy, no bolo vždy potrebné čakať kým sa inicializuje.

Na proces kalibrácie nemal žiaden pozorovateľný vplyv typ použitého projektora – DLP alebo 3LCD – pretože premietaný vzor je na kamere väčšinou dostatočne jasný na to, aby sa pri zachytení ktorejkoľvek z farebných fáz DLP projektora javila premietaná biela farba ako približne biela. Pred detekciou je vždy prevádzaný na čiernobiely.

V tejto fáze projektu som narazil aj na problém, kedy sa zdalo že zobrazovaný obraz pri dlhodobejšom snímaní postupne stále viac zaostával za realitou. Príčinu som odhalil v časovej náročnosti výpočtov ktoré prebiehali pri detekcií vzorov. Keďže nasledujúca snímka z kamery nebola odobratá v dostatočne krátkom čase od zachytenia, táto zostala v zásobníku do ktorého medzitým prišla ďalšia. Takto sa snímky postupne hromadili a obraz postupne spomaľoval, až kým bol program ukončený alebo sa zrútil. Problém som vyriešil zavedením samostatného vlákna, ktoré z kamery odoberá snímky hneď ako je to možné, zatiaľ čo si z neho vždy aktuálnu snímku vezme hlavné vlákno programu keď ho potrebuje. Detailnejšie je toto popísanú v časti 7.4.4. Obraz potom síce môže byť trhaný – zobrazovanie je vykonávané v hlavnom vlákne – ale nespomaľuje voči skutočnosti.

Po kalibrácii boli v ďalšom kroku určené vzájomné pozície kamery a projektora. Tieto hodnoty, najmä vzájomný posuv, mali v niektorých prípadoch viditeľné odchýlky od pozorovanej reality, v ráde centimetrov. Bolo ich možné ručne opraviť editáciou súboru s polohami. Tieto chyby mohli byť spôsobené nepresnosťou detekcie vzorov v kroku určovania polôh, ale nie je možné vylúčiť aj možnosť že nastali v dôsledku nepresnosti v kalibrácii, alebo nechcenou zmenou optickej osi projektora, ako je to popísané v časti 7.1.

V kroku demonštračného premietania bodov na pohyblivú dosku som začal používať projektor EPSON EMP-TW620, ktorý nebol kazový ako môj testovací projektor značky

HP. Okrem toho je typu 3LCD, na rozdiel od môjho projektora typu DLP, čo je principiálne vhodnejšie pre skenovanie farebným štruktúrovaným svetlom v neskoršej úlohe. Okrem toho som pre snímanie dosky so symbolmi začal používať priemyselnú kameru Allied Prosilica GC1290C, ktorú bolo možné pripevniť na statív a upravovať jej expozíciu a iné nastavenia proprietárnym softvérom. Obe zariadenia mi dal k dispozícii vedúci práce.

V tomto kroku sa ukázalo, že detekcia značiek, ako ak prepočet polôh bodov do priestoru sú rýchle a pomerne spoľahlivé, a v konečnom dôsledku dokazovali, že získané kalibračné parametre sa blížia realite.
Osvetlenie dosky v tomto prípade spoľahlivo zabezpečoval samotný projektor, pretože body mali biele pozadie a značky na doske sú matne čierne.

Vznikal tu ale problém pri rýchlom pohybe dosky. Keďže objektív PENTAX H612A(KP) ktorý som mal k dispozícii nebolo možné zaostriť na dostatočnú vzdialenosť od kamery, bolo potrebné na ňom nastaviť veľmi úzku clonu. Takto spôsobené stmavenie a zvýšenie zrnitosti obrazu bolo možné kompenzovať jedine dlhou expozíciou. Potom bol ale pri rýchlom pohybe obraz rozmazaný, a nebolo možné detekovať značky, čo sa navonok prejavuje ako zasekávanie sa premietaného obrazu.
Tento problém bol vyriešený po tom, ako mi vedúci práce zabezpečil úpravu závitu na kamere. Po tom sa dal obraz zaostriť tak, že objektív nebol zaskrutkovaný do kamery až po doraz, ale len do úrovne kde bol obraz kamery ostrý.

Grafický engine pre prácu s OpenGL som z počiatku programoval ako samostatný projekt v Mirosoft Visual Studio.

Dôvod k tomu bol oddelenie riešených úloh od zvyšku programu, a takto zamedzenie počiatočným chybám ktoré vyplývajú z interakcie medzi časťami programu.
Z dôvodu malých skúseností s prácou s OpenGL vo väčších projektoch, som čerpal inšpiráciu z internetových zdrojov vo forme tutoriálov a odporúčaní, pričom najvýznamnejšou pomocou pri stavaní flexibilnej, ale pritom jednoduchej štruktúry enginu mi boli stránky in2gpu.com [32]. V tomto samostatnom projekte som taktiež testoval moje shadery, ako je efekt tečúcej vody s ktorým som tu vytvoril aj ukážku na ilustrácii Obrázok 6.5.

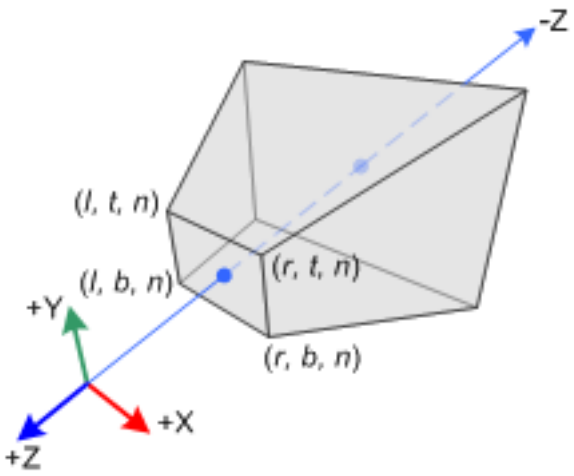
Po integrácii tohto enginu do zvyšku projektu vyšiel najavo problém sériovosti. Ak som totiž spustil vykresľovanie obrazu z inej časti programu, vykreslovacie okno prebralo kontrolu nad celým programom. Toto znamenalo, že okno počítačového videnia zastalo, a ovládacie prvky hlavného okna sa nedali požívať. Musel som preto vykonať úpravu, aby sa procesy vykresľovania, ako aj jeho okno, dostali na pozadie. Toto som dosiahol tak, že engine spúšťam na samostatnom vlákne, a komunikáciu s ním ošetrujem mutexom (7.4.5).

Pri prepočte intrinzických parametrov a transformácii z formátu používaného v OpenCV do formátu OpenGL som spravil dôležité pozorovania. V kamerovom priestore je kladný smer osi Y vždy nadol, teda v smere rastúceho indexu riadku obrazu. Naopak v OpenGL osa Y vždy smeruje nadol, pričom ale ostatné dve osi zostávajú nezmenené. Z programátorského hľadiska toto znamenalo, že nielen polohy, ale aj priestorové transformácie musím prevrátiť po tejto osi. S výhodou som tu využil fakt, že OpenGL elementárne transformácie tuhých telies vyjadruje v dvoch vektoroch R a T.
T je tu translácia, zložená zo zložiek posuvu v osiach X, Y a Z. Pre konverziu tu postačuje zmeniť znamienko zložky Y.
R je rotácia, vyjadrená v troch zložkách podľa Rodriguesovho vzorca [36]. Z princípu potom opäť postačilo zmeniť znamienko druhej zložky. Rotačnú maticu R_R veľkosti 3x3 som získal konverziou upraveného vektora R funkciou `cv::Rodrigues`. Výsledná maticu homogénnej transformácie RT_{MAT}, akú OpenGL požaduje, zložená z týchto dvoch prvkov sa dá potom vypočítať nasledovne:

$$RT_{MAT} = \begin{bmatrix} 0 & 0 & 0 & T_X \\ 0 & 0 & 0 & T_Y \\ 0 & 0 & 0 & T_Z \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} & R_R & & 0 \\ & & & 0 \\ & & & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{7.1}$$

Treba tiež podotknúť, že tieto matice sa pre OpenGL zadávajú v transponovanom tvare.

V OpenGL modelujem projektor ako ideálnu perspektívnu projekciu dierkovou komorou.
Priestor záberu takejto virtuálnej kamery je na Obrázok 7.18.



Obrázok 7.18: Oblasť záberu jednoduchej OpenGL kamery [37]

V tomto obrázku je šesť charakteristických parametrov, ktoré znamenajú:

- [l, r] – súradnice ľavého a pravého okraja projekčnej roviny
- [t, b] – súradnice horného a dolného okraja projekčnej roviny
- [n, f] – súradnice najnižšej a najvyššej vzdialenosti, medzi ktorými budú objekty vo virtuálnom priestore vykreslené.

Podľa [37] z týchto parametrov v konečnom dôsledku zostavíme projekčnú maticu v tvare:

$$\begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{r-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix} \tag{7.2}$$

Za predpokladu, že:

$$\begin{gathered} r-l=W;\ \ t-b=H \\ r=\ -c_x;\ \ l=W-c_x \\ t=\ -c_y;\ \ b=H-c_y \end{gathered} \tag{7.3}$$

Kde:

- W [pixel] je šírka obrazu projektora
- H [pixel] je výška obrazu projektora
- c_x [pixel] je x-ová súradnica optického centra obrazu projektora
- c_y [pixel] je y-ová súradnica optického centra obrazu projektora

Môžem túto maticu upraviť na tvar:

$$\begin{bmatrix} \frac{2n}{W} & 0 & 1-\frac{2c_x}{W} & 0 \\ 0 & \frac{2n}{H} & 1-\frac{2c_y}{W} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix} \tag{7.4}$$

Pre zachovanie správnych pomerov nahradím v prvom riadku n s f_x a v druhom f_y, čo sú ohniskové vzdialenosti projektora v dĺžkach strán x a y pixelov. V treťom riadku ich meniť nemusím, pretože tento riadok nemá vplyv na polohu zobrazených bodov v obraze. Pomocou neho sú len lineárne mapované výšky bodov do z-buffera za účelom rozoznania poradia prekrývajúcich sa polygónov.
Hodnoty v treťom stĺpci taktiež vynásobím hodnotou -1, pretože v porovnaní so schémou na Obrázok 7.18, ja považujem za kladný smer osi Z smer od kamery dopredu. Z rovnakého dôvodu zmením aj znamienko posledného prvku v treťom riadku. Samotnému c_y taktiež zmením znamienko, pretože ko bolo zmienené vyššie, kladný smer osi Y je v OpenGL opačný ako v OpenCV kde bolo c_y určené:

$$\begin{bmatrix} \frac{2f_x}{W} & 0 & \frac{2c_x}{W}-1 & 0 \\ 0 & \frac{2f_y}{H} & \frac{-2c_y}{H}-1 & 0 \\ 0 & 0 & \frac{f+n}{f-n} & \frac{2fn}{f-n} \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{7.5}$$

Za premenné f a n si dosadím hodnoty 5000 a 0.0001, čo je veľký rozsah vďaka ktorému budú vykreslené aj body ktoré boli v dôsledku chyby nájdené veľmi blízko, alebo veľmi ďaleko od projektora. Takto môžem lepšie určiť chybovosť použitého algoritmu.
Po použití tejto matice na premietanie textúry na pohyblivú dosku som ale zistil, že premietaná textúra sa vždy javí mať dvakrát menšiu dĺžku strán než som zadefinoval, a taktiež sa dvakrát pomalšie pohybuje v rovine XY. Z tohto som vyvodil, že sa v skutočnosti počíta s dvakrát menšou ohniskovou vzdialenosťou, než akú projektor v skutočnosti má. Tento vplyv neviem zdôvodniť, ale viem ho korigovať jednoducho zdvojnásobením vložených f_x a f_y:

$$\begin{bmatrix} \frac{4f_x}{W} & 0 & \frac{2c_x}{W} - 1 & 0 \\ 0 & \frac{4f_y}{H} & \frac{-2c_y}{H} - 1 & 0 \\ 0 & 0 & \frac{f+n}{f-n} & \frac{2fn}{f-n} \\ 0 & 0 & 1 & 0 \end{bmatrix} \quad (7.6)$$

Po takejto úprave sa už premietaný vzor správal podľa očakávaní, teda držal sa na pozícií pohyblivej fyzickej dosky, a mal na nej rozmery aké boli zadané.
Keďže intrinzická matica ktorú som s OpenCV získal ešte obsahuje s, teda zošikmenie pixelov, rozhodol som sa tento vplyv taktiež zohľadniť v projekčnej matici pre OpenGL. V mojich testoch bol tento parameter vždy nulový, ale iné použité vybavenie by tomuto nemuselo zodpovedať. Viem, že parameter s ovplyvňuje obrazovú súradnicu X premietnutého bodu, a násobí sa súradnicou Y v priestore. Aby bol zachovaný pomer tohto vplyvu voči f_x, pridám do prvého riadka prvok $4s/W$. Matica ktorú takto nakoniec používam, je táto:

$$\begin{bmatrix} \frac{4f_x}{W} & \frac{4s}{W} & \frac{2c_x}{W} - 1 & 0 \\ 0 & \frac{4f_y}{H} & \frac{-2c_y}{H} - 1 & 0 \\ 0 & 0 & \frac{f+n}{f-n} & \frac{2fn}{f-n} \\ 0 & 0 & 1 & 0 \end{bmatrix} \quad (7.7)$$

Počas návrhu som sa taktiež pokúšal prekonať vplyv skreslenia objektívu projektora, čo sú ďalšie intrinzické parametre projektora.

Toto skreslenie OpenCV vyjadruje v 4 až v 8 prvkoch. V najjednoduchšom prípade sú to štyri prvky k_1, k_2, p_1, p_2, kde:

- k_1 a k_2 sú koeficienty radiálneho skreslenia
- p_1 a p_2 sú koeficienty tangenciálneho skreslenia

Skreslený bod [x,y] na obraze reálnej kamery môžeme korigovať na [x_{corr},y_{corr}] nasledujúcimi rovnicami [38]:

$$x_{corr} = x + x * (k_1 r^2 + k_2 r^4) + (2p_1 xy + p_2(r^2 + 2x^2)) \quad (7.8)$$

$$y_{corr} = y + y * (k_1 r^2 + k_2 r^4) + (2p_2 xy + p_1(r^2 + 2y^2)) \quad (7.9)$$

Ja ale v prípade projektora – reverznej kamery – musím riešiť reverznú korekciu:

$$r^2 = (x - c_x)^2 + (y - c_y)^2 \tag{7.10}$$

$$x + x * \left(k_1\left((x - c_x)^2 + \left(y - c_y\right)^2\right) + k_2\left((x - c_x)^2 + \left(y - c_y\right)^2\right)^2\right) + \left(2p_1xy + p_2\left(\left((x - c_x)^2 + \left(y - c_y\right)^2\right) + 2x^2\right)\right) - x_{corr} = 0 \tag{7.11}$$

$$y + y * \left(k_1\left((x - c_x)^2 + \left(y - c_y\right)^2\right) + k_2\left((x - c_x)^2 + \left(y - c_y\right)^2\right)^2\right) + \left(2p_2xy + p_1\left(\left((x - c_x)^2 + \left(y - c_y\right)^2\right) + 2y^2\right)\right) - y_{corr} = 0 \tag{7.12}$$

Tu musím s pomocou známych koordinát [x_{corr},y_{corr}] získať hodnoty [x,y]. Analyticky neviem vyriešiť sústavu dvoch polynómov štvrtého stupňa o dvoch neznámych, na ktoré by sa toto dalo rozrátať. Zložitosť sa ešte zvýši, ak koeficienty skreslenia nebudú len štyri. Teoreticky môže existovať dostatočne presná aproximácia jednoduchším výpočtom.

Keďže premietaný obraz sa nejaví voči fyzickým objektom viditeľne skreslený (na pohyblivej doske boli strany premietnutej textúry vždy priame), usudzujem že takéto skreslenie objektívu je minimálne v mojom prípade zanedbateľné, takže korekciu nevykonávam.

Kým som k tomuto záveru prišiel, pokúšal som sa korekciu riešiť vo fragmentovom shaderi spôsobom, ktorý bol popísaný v [39]. V tom čase som si ale neuvedomil, že toto riešenie implementovalo priamu korekciu v kamere, a nie reverznú korekciu pre projektor ako ju tu popisujem. Priviedol som na fragmentový shader polohy optického centra projektora, jeho rozlíšenie a koeficienty oboch typov skreslení:

```
#version 420 core

layout(location = 0) out vec4 out_color;

uniform sampler2D texture1;

in vec2 texcoord;

// Camera calibration information

uniform vec2 imageSize; // Used to re-project pixel space to uv-space
uniform vec2 opticalCenter; // Optical center in pixel space
uniform vec2 focalLength; // Focal length in pixel space
uniform vec2 radialDistortion; // Coefficients
uniform vec2 tangentialDistortion; // Coefficients

void main()
{
```

```
   vec2 opticalCenterUV = opticalCenter / imageSize;
    vec2 focalLengthUV = focalLength / imageSize;
    vec2 lensCoordinates = (texcoord - opticalCenterUV) / focalLengthUV;

    float radiusSquared = dot(lensCoordinates, lensCoordinates);
    float radiusQuadrupled = radiusSquared * radiusSquared;

    float radialCoeff = radialDistortion.x * radiusSquared +
radialDistortion.y * radiusQuadrupled;

    float dx =
      tangentialDistortion.x * 2.0 * lensCoordinates.x * lensCoordinates.y
      + tangentialDistortion.y * (radiusSquared + 2.0 * lensCoordinates.x
      * lensCoordinates.x);
    float dy =
      tangentialDistortion.x * (radiusSquared + 2.0*lensCoordinates.x *
      lensCoordinates.x) + tangentialDistortion.y * 2.0*lensCoordinates.x
      * lensCoordinates.y;

      vec2 tangentialCoeff = vec2(dx, dy);

    vec2 distortedUV =
      ((lensCoordinates + lensCoordinates * radialCoeff + tangentialCoeff)
      * focalLengthUV) + opticalCenterUV;

  vec4 color = texture(texture1, distortedUV);
  out_color = color;
}
```

Zdrojový kód 7.7

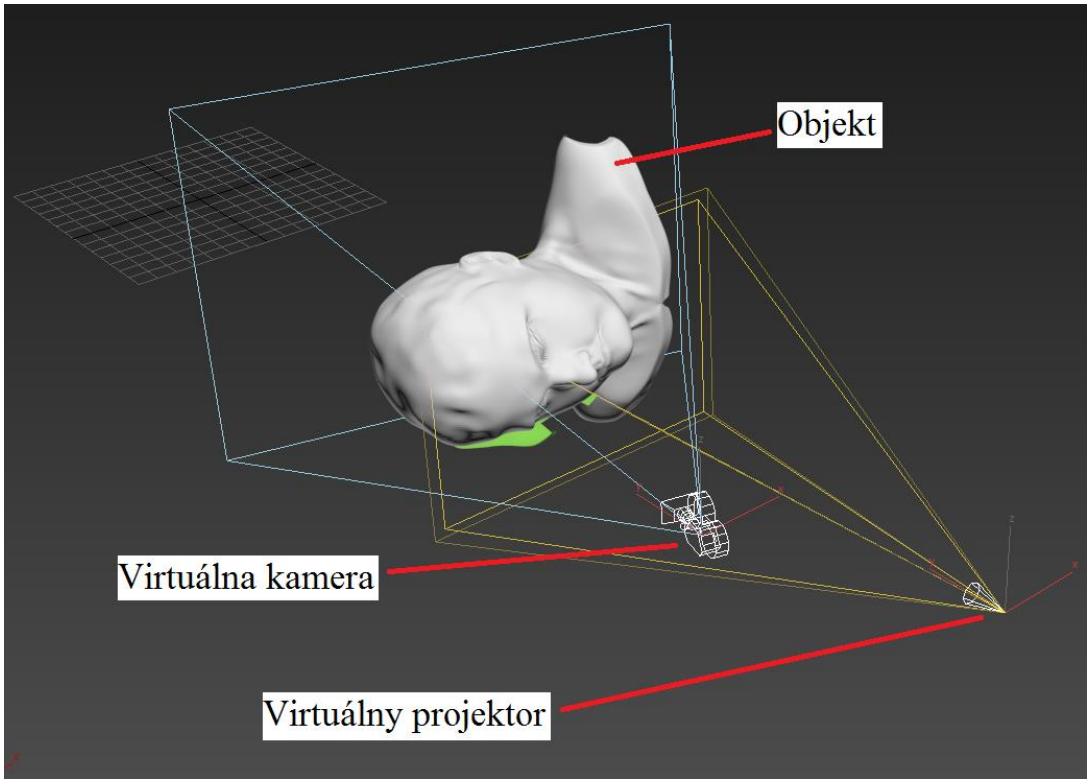
Textúrové koordináty sa teda zmenia tak, aby potlačili vplyv skreslenia v obraze, nie v priestore.

Dôsledok použitia tohto postupu bol, že sa dovtedy neviditeľné skreslenie v rámci textúry zvýšilo na viditeľné, ako je vidieť na Obrázok 7.19. Ľavý obrázok premietnutý na dosku sa javil byť neskreslený, pravý obrázok, pri ktorom je aplikovaný nesprávna korekcia bol viditeľne deformovaný. Deformácia sa odrazí len na textúre, nie na polohe vrcholov.

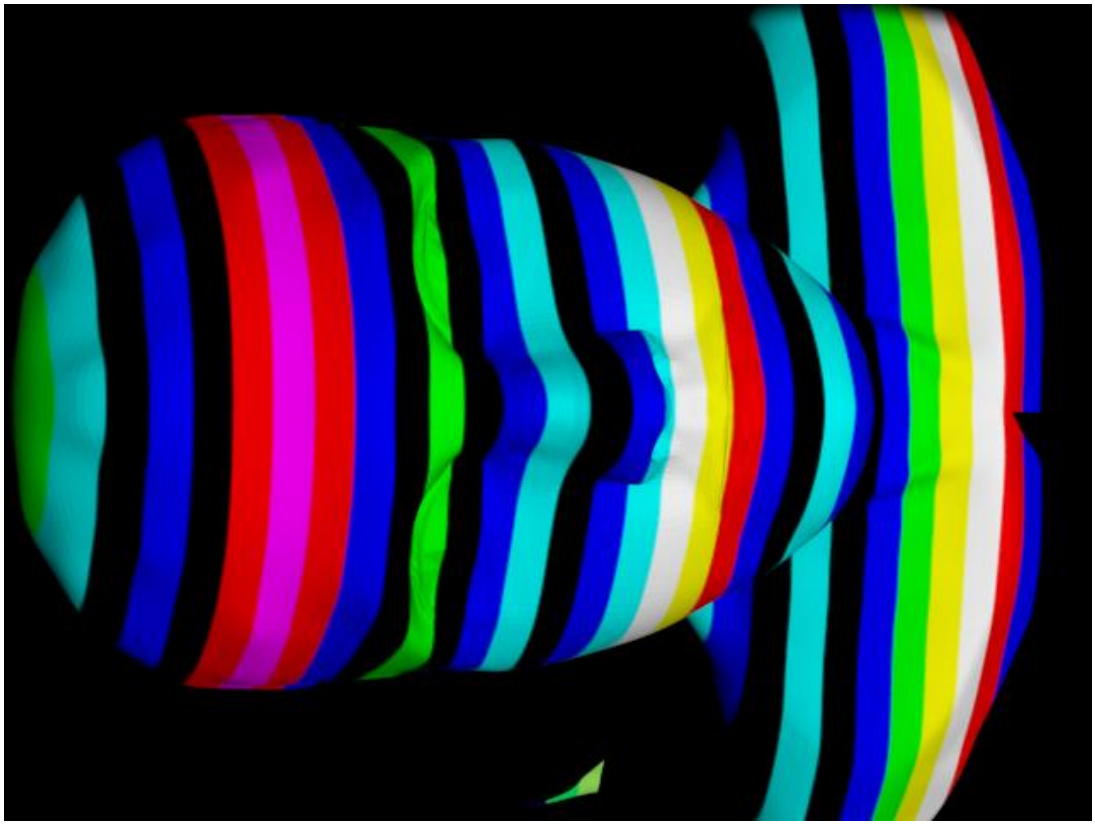
Dôležitým obmedzením, s ktorým som sa pri používaní tejto demonštračnej aplikácie stretol, bol maximálny ohol odklonu od kamery. Keď normála povrchu pohyblivej dosky prekročila odklon od optickej osi kamery približne 45°, identifikačné symboly prestanú byť rozpoznateľné. Toto obmedzenie je striktnejšie než požiadavky na maximálny odklon od projektora, pretože pri uhle 45° mal obraz textúry na doske stále primerané rozlíšenie aj v smere, v ktorom bol reprezentovaný najmenším počtom pixelov. Toto obmedzenie by mohlo byť prekonateľné s použitím iných identifikačných značiek, alebo použitím viacerých kamier (6.1).

Obrázok 7.19: Vplyv nesprávnej korekcie skreslenia

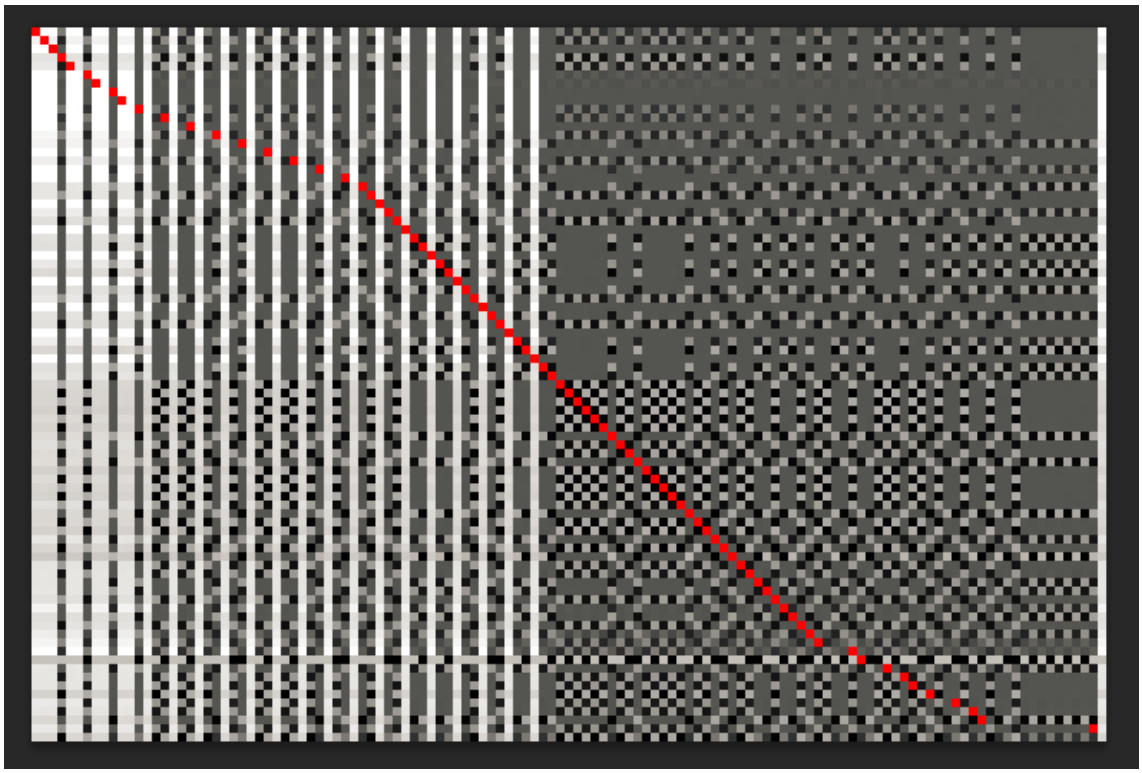
Pri programovaní implementácie algoritmov rekonštrukcie tvaru objektu štruktúrovaným svetlom, som z počiatku testoval funkčnosť programu najprv v ideálnom virtuálnom prostredí, s použitím programu Autodesk 3ds Max. V tom som si vytvoril scénu na Obrázok 7.20, kde sa nachádza ideálna virtuálna kamera a projektor, ktoré sú namierené na model ľudskej hlavy. Projektor na model premietal pásy podľa De Bruijnovej sekvencie, a takto osvetlený objekt vykresľujem z pohľadu virtuálnej kamery do obrazu ako je ten na Obrázok 7.21. Výhoda tohto oproti priamemu testovaniu v realite je, že tu poznám presné hodnoty všetkých intrinzických aj extrinzických parametrov všetkých troch objektov, a tieto môžem dosadzovať do výpočtov bez obáv o skreslenie výstupov zle nameranými parametrami. Okrem toho mi tento postup umožnil najprv programovať modul 3D skenovania povrchu oddelene od zvyšku programu – podobne ako pri grafickom engine – pretože tu nepotrebujem komunikovať s reálnou kamerou ani s projektorom.



Obrázok 7.20: Virtuálna scéna pre testovanie 3D skenovania povrchu


Obrázok 7.21: obraz z virtuálnej kamery na Obrázok 7.20

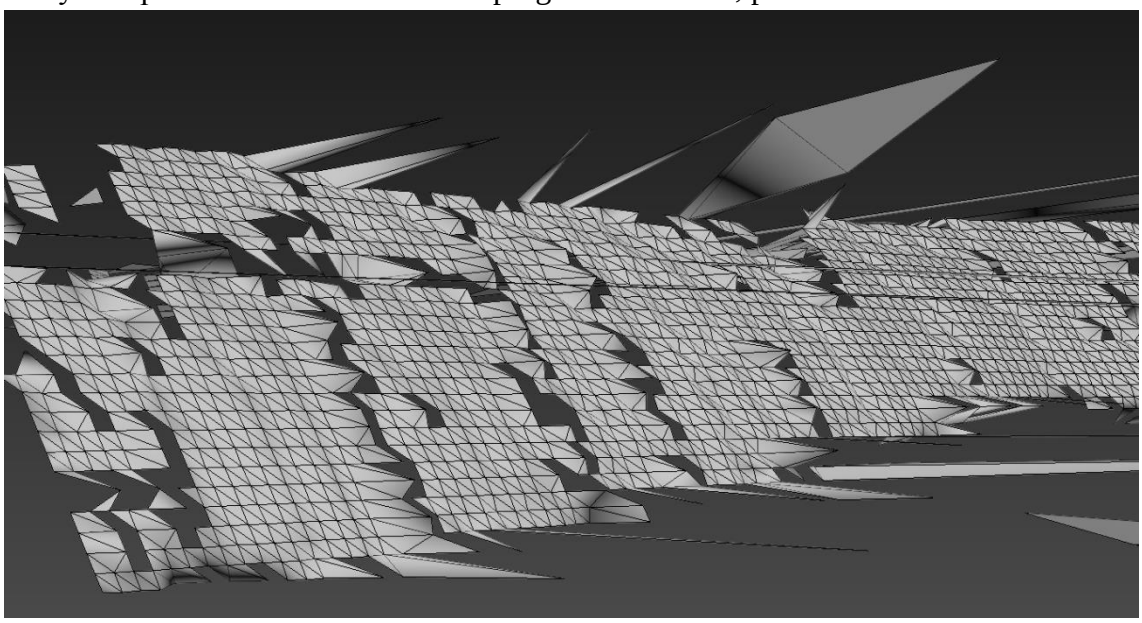
Pri tomto testovaní som taktiež využil fakt, že mriežky konzistencie a optimálne cesty cez tieto mriežky (5.1) som ukladal ako matice knižnice OpenCV. Tieto som si pre jednotlivé riadky mohol ukladať do obrázkových súborov, a takto ich analyzovať mimo programu. Na Obrázok 7.22 je ukázané, ako som prekryl odhadnutú optimálnu cestu (červená) cez mriežku konzistencie. Všimnite si, že mriežka je voči Obrázok 5.4 vertikálne prevrátená, pretože stúpajúci index nájdeného prechodu sa prejavil na obrázku ako stúpajúca hodnota osi y, ktorá smeruje nadol.



Obrázok 7.22: Vizuálna analýza optimálnej cesty cez mriežku

Z obrázku som vydedukoval čo bolo podstatou veľmi nesprávnej identifikácie prechodov. Zvislé pásy jasných farieb na tomto obrázku znamenajú veľmi vysoké hodnoty vypočítanej konzistencie s každým z nájdených prechodov. Takáto situácia by nemala nastávať, pretože prechody sú rôznorodé. Pásy sú stĺpce pre tie premietané prechody, pri ktorých boli zmeny prvých dvoch farebných kanálov, R a G, nulové. Pri každej konzistencii sa do mriežky volila najnižšia z troch hodnôt pre jednotlivé farebné kanály, a tu boli do tohto výberu brané len tieto dve hodnoty. Zmena kanála B by znížila zvolenú konzistenciu. Táto chyba vznikla pre moje nesprávne použitie štandardnej funkcie `std::min_element` ktorou som najmenší z troch prvkov vyberal. Ak ňou hľadám najnižší prvok v poli, musím ako druhý vstupný parameter vložiť počiatočnú adresu poľa zvýšenú o počet prvkov ktoré chcem prehľadávať, nie zvýšenú o počet o jeden menší ako som usudzoval [40]. Po oprave tohto omylu sa už tieto pásy neobjavili, a nájdená optimálna cesta riešila problém výrazne lepšie.

Po integrácii algoritmu do zvyšku programu som začal testovať skenovanie objektov v reálnom prostredí. Prvý závažný problém na ktorý som narazil, bolo vzájomné umiestnenie kamery a projektora. Aby bolo zakrivenie farebných rozhraní na objekte dobre viditeľné, bolo potrebné aby kamera na objekt mierila pod uhlom pootočeým približne 20° – 60°okolo zvislej osi voči uhlu ktorým na objekt mieri projektor. Ak bol tento uhol menší, program stále dokázal farebné prechody rozoznať, mal ale tendenciu detekovať nerovné povrchy ako ploché – na Obrázok 7.23 je rekonštrukcia môjho predlaktia, teda zaobleného objektu zjavne plochá. Tento obrázok som získal pri analýze exportovaného OBJ súboru v programe 3ds Max, podobne ako Obrázok 5.7.



Obrázok 7.23: Rekonštrukcia predlaktia sa javí byť plochá

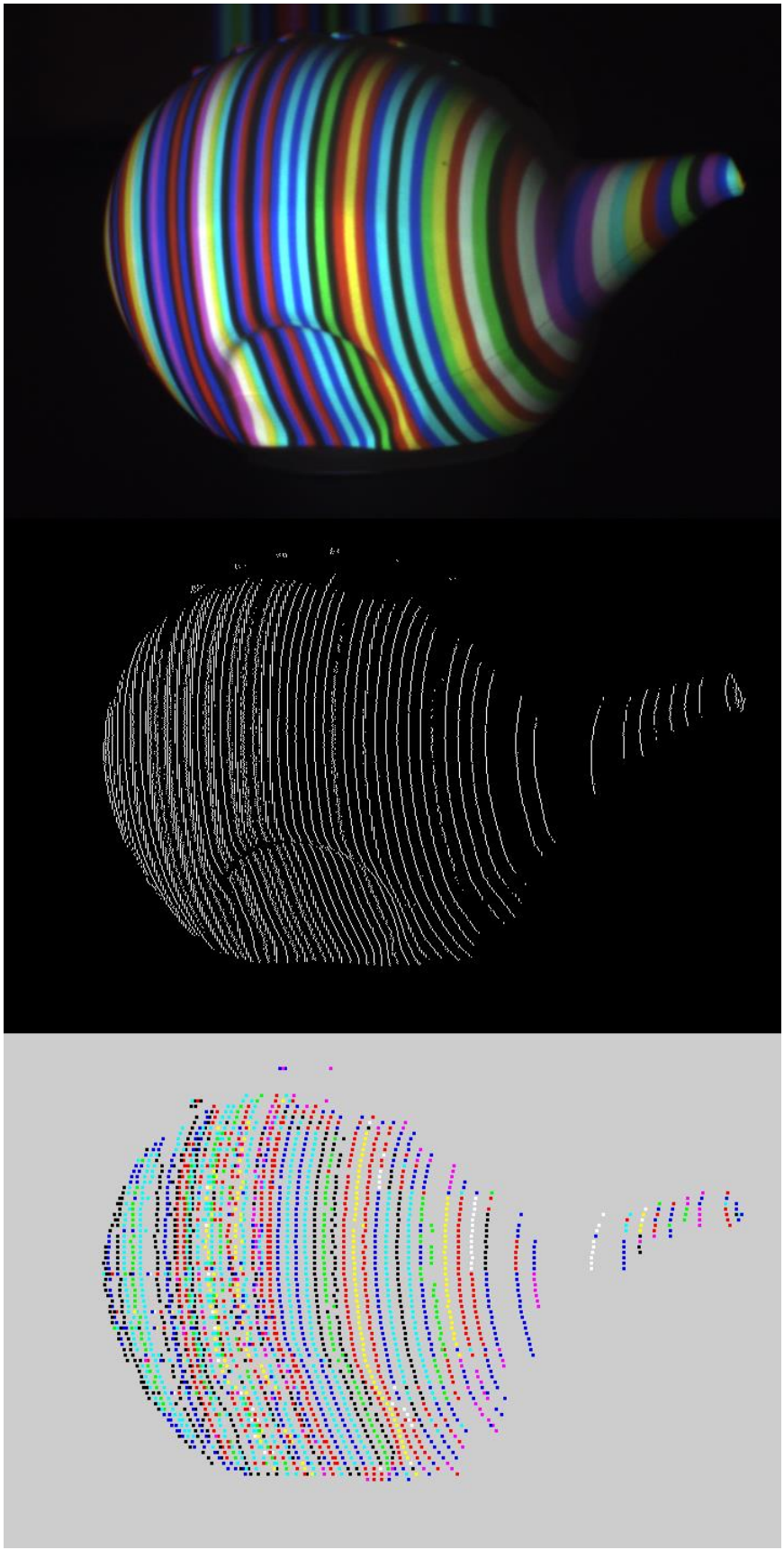
Pri natočenej kamere, ktorá musí byť navyše dostatočne blízko objektu pre lepšiu detekciu, vzrastá obtiažnosť určenia presnej vzájomnej polohy projektora a kamery. Pri predchádzajúcich úlohách, kde mierili obe zariadenia približne rovnakým smerom, som mohol určiť polohu pripevnením fyzickej šachovnice na tabuľu, alebo náprotivnú stenu a premietnutím šachovnice na túto istú plochu vedľa nej. Pri nerovnobežnom projektore voči kamere žiadnu spoločnú vzdialenú rovinu nevidím. Musel som teda pripevniť fyzickú šachovnicu na pevný plát, ktorý som potom umiestnil do malej oblasti v poli projektora a zároveň kamery. Následne som podľa obrazu z kamery posúval a natáčal dosku tak, aby premietnutý vzor bol viditeľný zároveň s fyzickým, a aby bol viditeľný celý.

Lepším riešením by mohlo byť umiestniť kameru a projektor do známej, pevne zviazanej polohy v určitej konštrukcii. Ak by sme potrebovali riešenie kde nebude projektor voči kamere vždy rovnako natočený, možno zvážiť pripevnenie kamery do otočných alebo posuvných prvkov ovládaných servomotormi. Takto by sme miesto vypočítavania vzájomnej polohy mohli túto priamo určiť.

Po úspešnom nastavení polohy som mohol začať testovať skenovanie.

Ako hlavný testovací objekt som si zvolil polievaciu konvicu, ktorú môžete vidieť na Obrázok 7.24, ako aj na Obrázok 7.15 a na Obrázok 7.16. Dôvodom bolo, že tento objekt bol primerane veľký, primerane členitý aby demonštroval obmedzenia použitej metódy, bol matnej bielej farby na ktorej je najlepšie vidieť premietnuté pásy. Predmet bol navyše ľahko prenosný, lacný, pripomína „čajník z Utahu“ používaný na testovanie 3D grafiky [41] a má aj istý tématický vzťah k efektu tečúcej vody.

Okrem posúdenia textúry premietnutej po rekonštrukcii objektu „od oka“, som mohol aj sledovať na dodatočne zobrazených obrázkoch to, ako identifikoval algoritmus jednotlivé body na rozhraniach farebných pásov. Na týchto obrázkoch (príklad na Obrázok 7.24 dole) som prechodové body vykreslil vždy vo farbe pásu, ktorý by mu mal v premietanej sekvencii predchádzať v prípade, že bola identifikácia správna. Správnu identifikáciu rozoznám jednoducho tak, že sledujem ako sa zhoduje vykreslená farba bodov s farbou pásu na ľavo od tohto bodu na pôvodnej snímke. Na obrázku je vidieť, že v pravej časti je oblasť kvalitnej identifikácie kde farebné body tvoria súvislé pásy rovnakej farby ako sú na pôvodnej snímke. V tejto oblasti potom bolo možné po premietnutí vrstevnicovej textúry súviské pokrytie (Obrázok 7.15), zatiaľ čo mimo nej chýbali trojuholníky pretože po nesprávnej identifikácii roviny na ktorej sa body nachádzajú, boli tieto body lokalizované veľmi ďaleko od pozorovanej oblasti alebo vylúčené ako outliery. Takýto Horší stav môžeme pozorovať na Obrázok 7.24 dole na ľavej časti zobrazenia. Farebné body tu majú chaotický charakter a nezodpovedajú očakávanej farbe. Dôvodom je, že ako je vidieť na Obrázok 7.24 hore, v tejto časti bol povrch objektu príliš odklonený od kamery a teda sa rozhrania farebných pásov dali horšie odlíšiť od šumu. Navyše sú tu pásy tak úzke, že neostrosť obrazu ich farby mieša naprieč viac ako jednou šírkou pásu. Inak slovami sa to dá popísať ako neprimerané potlačenie užitočných vysokých frekvencii v obraze. Dôsledkom tohto je, že sú tu rozhrania vôbec zle nachádzané, a ich obrazy (Obrázok 7.24 v strede) majú roztrúsený charakter, čím sporadicky pridávajú alebo odoberajú rozhrania zo sekvencie.
Naopak na pravej strane môžeme vidieť, že v dôsledku horšieho osvetlenia časti obrazu farebným vzorom tu rozhrania často neboli objavené. Nastaviť vhodnú úroveň osvetlenia bolo niekedy problematické, retož ebol potrebný kompromis medzi viditeľnosťou a zamedzením vzniku odleskov.
Z obrázkov je taktiež vidieť, že najlepšia identifikácia bola v strede výšky objektu, kde je v jednom riadnu mnoho rozhraní. Čím viac zoradených rozhraní po skenovacej líni bolo, tým viac bodov bolo identifikovaných správne, pretože tieto zvyšujú skóre pre naozaj správnu cestu mriežkou konzistencii.
Okrem osvetlenia a natočenia zaridení treba venovať pozornosť ostrosti. Kamera aj projektor musia byť čo najpresnejšie zaostrené na objekt. Pri projektore tu vzniká problém, pretože čím je objekt bližšie, tým na menšom rozsahu vzdialeností je obraz na ňom ostrý. V prípade modelu EPSON EMP-TW620 som nemohol zaostriť na menej ako cca 1 meter, čo vyžadovalo pridávanie podstavcov objektu mimo pracovného stola.
Výsledná reknštrukcia zo snímky na Obrázok 7.15 vľavo hore.



Obrázok 7.24: Príklad identifikácie premietaných rozhraní na ktorých body ležia

8 ZHRNUTIE

8.1 Kalibrácie

Získanie intrinzických parametrov kamery je možné dosiahnuť pomocou algoritmov obsiahnutých v knižnici OpenCV. Ich rozšírením je taktiež možné dosiahnuť kalibráciu projektora s využitím podobných princípov.
Pri kalibrácii má zásadný význam osvetlenie scény, ako aj dostatočne rôznorodé vzorky z ktorých parametre odhadujeme.
Proces kalibrácie, zvlášť pre projektor je zdĺhavý, ale v ideálom prípade nie je nutné ho často opakovať.

8.2 Demonštračné úlohy

V tejto práci som implementoval dve hlavné demonštračné úlohy pre spoluprácu kamery a projektora.
Prvá je založená na sledovaní dosky s rozpoznateľnými symbolmi textúrovaným obdĺžnikom podľa aktuálneho natočenia a umiestnenia. Táto úloha demonštruje spoluprácu zariadení v smere od kamery k projektoru, teda kamera nesleduje čo projektor premieta, ale len fyzický objekt. Projektor na neho premieta obraz ktorý potom vidí užívateľ. Čiastočne tu existuje aj primitívna spolupráca v opačnom smere, keďže projektor poskytuje objektu osvetlenie.
Druhá úloha demonštruje úlohu kde je spolupráca obojsmerná. Projektor poskytuje kamere zdroj štruktúrovaného svetla, na základe ktorého môžeme z jeho obrazu na kamere získať tvar osvetleného objektu. Následne s touto informáciou opäť opracuje projektor, a to premietnurím textúry odvodenej z tvaru objektu ktorý vidí užívateľ.
Táto úloha bola značne zložitejšia, a prinášala viaceré výzvy ktoré v práci popisujem.
Obe tieto úlohy som implementoval v priloženom programe, ktorý okrem nich obsahuje aj funkcie pre kalibráciu a lokalizáciu zariadenia.

9 ZÁVER

Cieľom tejto práce bolo zoznámiť sa s mechanizmami pre určovanie tvaru scény pomocou kamier, ako aj navrhnúť mechanizmy pre projekciu dát do scény a návrh s implementáciou demonštračných úloh pre tieto mechanizmy.

V Kalitolách 2 až 4 sú popísané teoretické vlastnosti kamier a projektorov, najdôležitejšie priestorové transformácie ktoré sa v texte práce objavujú a princípy kalibrácie projektora a kamery ktoré používam.

V piatej kapitole popisujem zvolený prístup, z ktorého sa odrážam pri návrhu aplikácie rekonštrujkcie tvaru objektu projekciou štruktúrovaného svetla. Popisujem tu tiež konkrétnu metódu idntifikácie svetelných rozhraní od autorov Zhang a spol., ktorú som v upravenej forme využil.

V šiestej kapitole navrhujem demonštračné úlohy pre spoluprácu kamery a projektora, ktoré som sa v praktickej časti práce snažil implementovať.

V siedmej kapitole, ktorá tvorí jadro praktickej časti práce, popisujem program ktorý som v rámci tohto projektu vytvoril. Tvorba tohto programu predstavuje najväčšiu zložku práce na projekte obsahovo aj časovo. Program dokáže získať geometrické parametre kamery a projektora s využitím šachovnicových kalibračných vzorov a dokáže následne aj určiť vzájomnú polohu projektora a kamery.
Program obsahuje aj demonštračné úlohy, ktoré dvoma rôznymi spôsobmi ukazujú ako je možné dosiahnuť spoluprácu kamery s projektorom
V kapitole som popísal aj problémy a zaujímavé javy s ktorými som sa pri tvorbe a používaní programu stretol, a tiež neštandardné postupy ktoré som používal. V tejto kapitole sú zahrnuté aj postupy a odporúčania k práci s programom.

Softvér vyvinutý pre tento projekt môže slúžiť ako základ pre ďalší rozvoj. Demonštračné úlohy sú čisto vizuálne, čo naznačuje potenciál pre využitie v oblastiach ako dizajn a reklama alebo zábavný priemysel. Princípy a algoritmy ktoré v nich ale používam je možné aplikovať aj inými spôsobmi. Trojrozmerné skenovanie objektov sa využíva v digitálnej archivácii artefaktov. Premietanie vrstevníc je tiež možné použiť napríklad pri kontrole rovnosti povrchov.
V práci som naznačil niekoľko možných vylepšení systému, ktoré som z časových alebo materiálnych dôvodov implementovať počas tejto práce nemohol, no sú výzvou do budúcnosti.
Taktiež verím, že pozorovania a postupy ktoré popisujem môžu poslúžiť ďalším tvorcom v podobných oblastiach.

10 BIBLIOGRAFIA

1. **Zhang, Zhengyou.** *A Flexible New Technique for Camera Calibration.* [PDF dokument] Washington : Microsoft Research, 2000. 22(11):1330–1334.

2. —. *Flexible Camera Calibration By Viewing a Plane From Unknown Orientations.* [PDF dokument] Washington : Microsoft Research, 1999. 0-7695-0164-8/99 .

3. **Wilson, Tracy V. a Johnson, Ryan.** How DLP Sets Work. *HowStuffWorks.* [Online] HowStuffWorks, a division of InfoSpace LLC, © 1998-2015. [Dátum: 5. 1 2016.] http://electronics.howstuffworks.com/dlp2.htm.

4. **3LCD.com Business Center.** Overview. *3LCD.com.* [Online] 3LCD.com Business Center, © 2016. [Dátum: 5. 1 2016.] http://www.3lcd.com/explore/default.aspx.

5. **Wikipedia contributors.** Camera resectioning. *Wikipedia, The Free Encyclopedia.* [Online] Wikipedia, The Free Encyclopedia., 10. 12 2015. [Dátum: 5. 1 2016.] https://en.wikipedia.org/w/index.php?title=Camera_resectioning&oldid=694589695.

6. —. Color mapping. *Wikipedia, The Free Encyclopedia.* [Online] Wikipedia, The Free Encyclopedia., 25. 9 2015. [Dátum: 5. 1 2016.] https://en.wikipedia.org/w/index.php?title=Color_mapping&oldid=682693297.

7. **opencv dev team.** Camera Calibration and 3D Reconstruction. *OpenCV 2.4.12.0 documentation.* [Online] 5. 1 2016. [Dátum: 5. 1 2016.] http://docs.opencv.org/2.4/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html.

8. **Kyrki, Ville.** *Machine Perception - Camera Calibration.* [PDF dokument - prednáška] Helsinki : Aalto University, 2014.

9. **Falcao, Gabriel, Hurtos, Natalia a Massich, Joan.** *Plane-based calibration of a projector-camera system.* [PDF dokument] s.l. : VIBOT Master, 2008.

10. **Wikipedia contributors.** Line–plane intersection. *Wikipedia, The Free Encyclopedia.* [Online] Wikipedia, The Free Encyclopedia., 30. 12 2015. [Dátum: 6. 1 2016.] https://en.wikipedia.org/w/index.php?title=Line%E2%80%93plane_intersection&oldid=697480228.

11. CNI - Diode laser module / red / line. *Direct Industry.* [Online] Direct Industry, © 2016. [Dátum: 8. Máj 2016.] http://www.directindustry.com/prod/changchun-new-industries-optoelectronics-techco/product-36090-1424757.html.

12. **Barry, Andrew J.** MakerScanner Software. *andrew j. barry, ph.d.* . [Online] MakerBot Industries, 30. Január 2012. [Dátum: 8. Máj 2016.] http://abarry.org/makerscanner/3-makerscanner-software.html.

13. **Saelig Company, Inc.** Saelig Announces MakerBot Digitizer Desktop 3D Scanner. *Pitchengine.* [Online] PitchEngine, Inc., © 2015. [Dátum: 8. Máj 2016.] http://www.pitchengine.com/pitches/5f3fc7f0-753a-4281-a8a9-dd74cb3dfb5f.

14. **Pagés, Jordi, a iní, a iní.** Optimised De Bruijn patterns for one-shot shape acquisition. *ScienceDirect.* [Online] 1. August 2005. [Dátum: 6. 1 2016.] http://www.sciencedirect.com/science/article/pii/S0262885605000508.

15. **Chen, S. Y., Li, Y. F. a Zhang, Jianwei.** Vision Processing for Realtime 3-D Data acquisition Based on Coded Structured Light. *IEEE Xplore.* [Online] Február 2008. [Dátum: 8. Máj 2016.] http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4429304&tag=1.
16. **Rocchini, C, a iní, a iní.** A low cost 3D scanner based on structured light. *Claudio Rocchini's Research.* [Online] 2001. [Dátum: 8. Máj 2016.] http://www.rockini.name/research/papers/vcgscanner.pdf.
17. **Zhang, Li, Curless, Brian a Seitz, Steven M.** *Rapid Shape Acquisition Using Color Structured Light.* [PDF dokument] Washington : IEEE; University of Washington, 2002. 0-7695-1521-4.
18. **Wikipedia contributors.** De Bruijn sequence. *Wikipedia, The Free Encyclopedia.* [Online] Wikipedia, The Free Encyclopedia, 30. Apríl 2016. [Dátum: 8. Máj 2016.] https://en.wikipedia.org/w/index.php?title=De_Bruijn_sequence&oldid=717928470. 717928470.
19. **Kjellerstrand, Håkan.** de Bruijn sequence. *hakank's Home Page.* [Online] [Dátum: 8. Máj 2016.] http://www.hakank.org/comb/debruijn.cgi.
20. **Wikipedia contributors.** Gouraud shading. *Wikipedia, The Free Encyclopedia.* [Online] Wikipedia, The Free Encyclopedia, 3. August 2015. [Dátum: 8. Máj 2016.] https://en.wikipedia.org/w/index.php?title=Gouraud_shading&oldid=674394846. 674394846.
21. **Draxinger, Wolfgang.** Calculating normals in a triangle mesh. *Stack Overflow.* [Online] Stack Exchange Inc, 28. Apríl 2013. [Dátum: 8. Máj 2016.] http://stackoverflow.com/questions/6656358/calculating-normals-in-a-triangle-mesh/6661242#6661242.
22. **OpenGL.org.** About. *OpenGL.* [Online] The Khronos Group, © 2016. [Dátum: 6. 1 2016.] https://www.opengl.org/about/.
23. EPSON EMP-TW620. *EPSON.* [Online] EPSON, © 2016. [Dátum: 8. Máj 2016.] https://www.epson.de/en/products/projectors/home-cinema/epson-emp-tw620.
24. **Wikipedia contributors.** Wavefront .obj file. *Wikipedia, The Free Encyclopedia.* [Online] Wikipedia, The Free Encyclopedia, 25. Apríl 2016. [Dátum: 8. Máj 2016.] https://en.wikipedia.org/w/index.php?title=Wavefront_.obj_file&oldid=717037921. 717037921.
25. OpenCV license. *OpenCV.org.* [Online] Itseez, © 2016. [Dátum: 6. 1 2016.] http://opencv.org/license.html.
26. **opencv dev team.** Feature Detection. *OpenCV 2.4.12.0 documentation.* [Online] opencv dev team, 6. 1 2016. [Dátum: 6. 1 2016.] http://docs.opencv.org/2.4/modules/imgproc/doc/feature_detection.html.
27. —. Geometric Image Transformations. *OpenCV 2.4.12.0 documentation.* [Online] opencv dev team, 6. 1 2016. [Dátum: 6. 1 2016.] Geometric Image Transformations.
28. freeglut, The Free OpenGL Utility Toolkit. *sourceforge.* [Online] Slashdot Media. [Dátum: 8. Máj 2016.] http://freeglut.sourceforge.net/.

29. **Baker, Stephen J.** Hacking GLUT to Eliminate the glutMainLoop() Problem. *sjbaker.org.* [Online] [Dátum: 8. Máj 2016.] https://www.sjbaker.org/steve/software/glut_hack.html.
30. The OpenGL Extension Wrangler Library. *sourceforge.* [Online] Slashdot Media, 31. Január 2016. [Dátum: 8. Máj 2016.] http://glew.sourceforge.net/.
31. Simple OpenGL Image Library. *www.lonesock.net.* [Online] 7. Júl 2008. [Dátum: 8. Máj 2016.] http://www.lonesock.net/soil.html.
32. **in2gpu.** MODERN OPENGL - OpenGL From Zero To Hero. *in2GPU.* [Online] in2GPU, 19. Marec 2016. [Dátum: 8. Máj 2016.] http://in2gpu.com/opengl-3/.
33. **The Linux Information Project.** Graphic Engine Definition. *The Linux Information Project.* [Online] The Linux Information Project, 2015. November 2005. [Dátum: 11. Máj 2016.] http://www.linfo.org/graphic_engine.html.
34. **Allied Vision Technologies GmbH.** *AVT PvAPI - Programmer's Reference Manual.* [PDF dokument] Stadtroda : Allied Vision Technologies GmbH, 2015.
35. Contours Hierarchy. *OpenCV 3.1.0-dev.* [Online] OpenCV, 6. 1 2016. [Dátum: 6. 1 2016.] http://docs.opencv.org/master/d9/d8b/tutorial_py_contours_hierarchy.html.
36. **Khlebovich, Pavel.** IP Webcam. *Google Play.* [Online] Google, 29. Marec 2016. [Dátum: 9. Máj 2016.] https://play.google.com/store/apps/details?id=com.pas.webcam&hl=sk.
37. **contributors, Wikipedia.** Rodrigues' rotation formula. *Wikipedia, The Free Encyclopedia.* [Online] Wikipedia, The Free Encyclopedia, 4. Máj 2016. [Dátum: 9. Máj 2016.] https://en.wikipedia.org/w/index.php?title=Rodrigues%27_rotation_formula&oldid=718643880. 718643880.
38. **Song Ho, Ahn.** OpenGL Projection Matrix. *songho.ca.* [Online] 2012. [Dátum: 9. Máj 2016.] http://www.songho.ca/opengl/gl_projectionmatrix.html.
39. **opencv dev team.** Camera calibration With OpenCV. *OpenCV 2.4.13.0 documentation.* [Online] opencv dev team, © 2014. [Dátum: 9. Máj 2016.] http://docs.opencv.org/2.4/doc/tutorials/calib3d/camera_calibration/camera_calibration.html.
40. **pixelminer.** Rendering webcam images in the Rift using OpenCV and OpenGL. *forums.oculus.com.* [Online] Oculus VR, LLC, 20. Október 2016. [Dátum: 9. Máj 2016.] https://forums.oculus.com/developer/discussion/15906/rendering-webcam-images-in-the-rift-using-opencv-and-opengl.
41. min_element. *cplusplus.com.* [Online] min_element, © 2016. [Dátum: 9. Máj 2016.] http://www.cplusplus.com/reference/algorithm/min_element/.
42. **Wikipedia contributors.** Utah teapot. *Wikipedia, The Free Encyclopedia.* [Online] Wikipedia, The Free Encyclopedia, 3. Apríl 2016. [Dátum: 9. Máj 2016.] https://en.wikipedia.org/w/index.php?title=Utah_teapot&oldid=713358430. 713358430.
43. About OpenCV. *Opencv.* [Online] © 2013. http://opencv.org/about.html.

Zoznam príloh

Obrázky v texte práce

Vzorky zdrojového kódu

Prílohy na priloženom CD

- Zdrojový kód programu
- Elektronická verzia textu práce
- Dodatočné fotografie a videá z práce s programom
- Súbory s nameranými parametrami použitých kameier a projektorov

Použité technologické prostriedky

Software

- Microsoft Visual Studio 2013 – vývoj programu
- Knižnica OpenCV 2.4.9 – funkcie počítačového videnia
- Knižnica Allied Vision PvApi – využitie záznamu z priemyselnej kamery Allied
- Allied Vision SampleViewer – nastavenie priemyselnej kamery Allied
- Microsoft Word 2010 – Tvorba tejto dokumentácie
- Google SketchUp 2016 – Tvorba grafických popisov
- Autodesk 3ds Max – analýza rekonštrukcie, testy, grafické popisy
- IP Webcam – Zdieľanie kamerového záznamu zo smartfónu
- Microsoft Paint 6.3 – Úprava obrázkov
- Adobe Photoshop CS6 – Úprava ilustrácii, úprava a tvorba textúr

Hardware

- Priemyselná kamera AlliedProsilicaGC1290C
- Objektív PENTAX H612A(KP)
- Webkamera Creative Webcam Live!Ultra
- Smartfón Samsung Galaxy S4 Active
- Projektor HP – BENQ sRGB
- Projektor EPSON EMP-TW620
- Notebook Lenovo M5400