

UNIVERZITA KOMENSKÉHO V BRATISLAVE  
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY

PLUGIN PRE UNITY  
IMPLEMENTUJÚCI PODPORU  
LASEROVEJ PIŠTOLE  
BAKALÁRSKA PRÁCA

2023  
MARIÁN KICA

UNIVERZITA KOMENSKÉHO V BRATISLAVE  
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY

PLUGIN PRE UNITY  
IMPLEMENTUJÚCI PODPORU  
LASEROVEJ PIŠTOLE  
BAKALÁRSKA PRÁCA

Študijný program: Aplikovaná informatika  
Študijný odbor: Informatika  
Školiace pracovisko: Katedra aplikovanej informatiky  
Školiteľ: RNDr. Andrej Lúčny, PhD.

Bratislava, 2023  
Marián Kica



Univerzita Komenského v Bratislave  
Fakulta matematiky, fyziky a informatiky

---

## ZADANIE ZÁVEREČNEJ PRÁCE

**Meno a priezvisko študenta:** Marián Kica  
**Študijný program:** aplikovaná informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)  
**Študijný odbor:** informatika  
**Typ záverečnej práce:** bakalárska  
**Jazyk záverečnej práce:** slovenský  
**Sekundárny jazyk:** anglický

**Názov:** Plugin pre Unity implementujúci podporu laserovej pištole  
*The plugin for Unity that implements support for the laser gun*

**Anotácia:** Práca má implementačný charakter. Cieľom je vytvoriť plugin pre podporu laserovej pištole, ktorou je možné mieriť na stenu, na ktorú je dátovým projektorom premietaná počítačová hra. Projekčná scéna je snímaná z kamery, ktorej objektív je prekrytý vhodne prekříženými polarizačnými filtrami. Pomocou kalibračného princípu je potom skutočný laserový lúč premenený na analogický laserový lúč vo virtuálnom prostredí. Vytvorený plugin je následne využitý na implementáciu jednoduchkej počítačovej hry

**Cieľ:** Práca má dva ciele:  
1. vyvinúť plugin implementujúci podporu laserovej pištole  
2. vyskúšať ho v jednoduchkej počítačovej hre, ktorú si bakalant sám navrhne

**Literatúra:** Hocking, Joesph: Unity in Action, Second Edition, Manning Publications, 2018  
Learning OpenCV 3, Computer Vision in C++ with the OpenCV Library By Gary Bradski, Adrian Kaehler, O'Reilly Media, 2016  
learnopencv.com  
opencv.org

**Poznámka:** Unity3D, C#, C++, OpenCV

**Kľúčové slová:** počítačová hra, OpenCV, C++, kalibrácia

**Vedúci:** RNDr. Andrej Lúčny, PhD.  
**Katedra:** FMFI.KAI - Katedra aplikovanej informatiky  
**Vedúci katedry:** prof. Ing. Igor Farkaš, Dr.  
**Dátum zadania:** 28.05.2022

**Dátum schválenia:** 19.09.2022

doc. RNDr. Damas Gruska, PhD.  
garant študijného programu

---

študent

---

vedúci práce

**Pod'akovanie:** podakovanie sem.

## Abstrakt

Slovenský abstrakt v rozsahu 100–500 slov, jeden odstavec. Abstrakt stručne sumarizuje výsledky práce. Mal by byť pochopiteľný pre bežného informatika. Nemal by teda využívať skratky, termíny alebo označenie zavedené v práci, okrem tých, ktoré sú všeobecne známe.

**Kľúčové slová:** počítačova hra, OpenCV, C++, kalibrácia

## Abstract

Abstract in the English language (translation of the abstract in the Slovak language).

**Keywords:** computer game, OpenCV, C++, calibration

# Obsah

|  |           |
|--|-----------|
| <b>Úvod</b>  | <b>1</b>  |
| <b>1 Predchádzajúce práce</b>  | <b>2</b>  |
| 1.1 Two-Player Boids Game With Laser Pointer Controllers . . . . .   | 2         |
| 1.2 Smokeless Range ©2.0 - Home Simulator . . . . .                  | 3         |
| <b>2 Východiska práce</b>  | <b>5</b>  |
| 2.1 Použité technológie . . . . .                                    | 5         |
| 2.1.1 Unity game engine . . . . .                                    | 5         |
| 2.1.2 C# . . . . .   | 5         |
| 2.2 Spracovanie obrazu a počítačové videnie . . . . .                | 6         |
| 2.2.1 Počítačové videnie nie je vôbec jednoduché . . . . .           | 6         |
| 2.2.2 Reprezentácia a analýza obrazu . . . . .                       | 7         |
| <b>3 Špecifikácia požiadaviek a návrh systému</b>                    | <b>9</b>  |
| 3.1 Špecifikácia požiadaviek . . . . .                               | 9         |
| 3.2 Návrh systému . . . . .  | 10        |
| 3.2.1 Celkový návrh . . . . .  | 10        |
| 3.2.2 Upevnenie polarizačných filtrov . . . . .                      | 11        |
| 3.2.3 Návrh pluginu . . . . .  | 15        |
| 3.2.4 Návrh aplikácie . . . . .                                      | 22        |
| <b>4 Implementácia systému a výsledky</b>                            | <b>25</b> |
| 4.1 Upevnenie polarizačných filtrov . . . . .                        | 25        |
| 4.1.1 Prototyp . . . . .   | 25        |
| 4.1.2 Produkt . . . . .  | 26        |
| 4.2 Plugin . . . . .   | 27        |
| 4.2.1 Implementácia stavového automatu . . . . .                     | 27        |
| 4.2.2 Implementácia detekcie laserového lúča v obraze z kamery . . . | 30        |
| 4.2.3 Implementácia kalibrácie a transformácia súradníc . . . . .    | 36        |
| 4.2.4 Výsledky implementácie kalibrácie a transformácia súradníc . . | 38        |

|       |  |           |
|-------|--|-----------|
| 4.2.5 | Implementácia simulovania kliknutia ľavým tlačidlom myši . . . | 49        |
| 4.2.6 | Implementácia časti Utility . . . . .                          | 52        |
| 4.3   | Aplikácia . . . . .  | 52        |
| 4.3.1 | Implementácia hlavného menu . . . . .                          | 53        |
| 4.3.2 | Implementácia hernej logiky . . . . .                          | 53        |
| 4.3.3 | Integrácia pluginu do aplikácie . . . . .                      | 54        |
| 4.4   | Obmedzenia a odporúčania k plugin-u . . . . .                  | 55        |
|       | <b>Záver</b>   | <b>57</b> |
|       | <b>A Zdrojové kódy</b>   | <b>60</b> |
|       | <b>B Návod a príručky</b>                                      | <b>61</b> |



# Zoznam obrázkov

|      |   |    |
|------|---|----|
| 2.1  | Ilustrácia lokálneho a globálneho pohľadu [1] . . . . .                         | 7  |
| 3.1  | Ilustračný obrázok polarizačných filtrov umiestnených na sebe . . . . .         | 10 |
| 3.2  | Schéma celkového návrhu . . . . .   | 11 |
| 3.3  | Polarizačné filtre . . . . .  | 12 |
| 3.4  | Obrázok, premietaný projektorom v obrázkoch 3.5 - 3.8 . . . . .                 | 12 |
| 3.5  | Obraz z kamery za denného svetla, bez použitia polarizačných filtrov . . . . .  | 13 |
| 3.6  | Obraz z kamery za denného svetla, s použitím polarizačných filtrov . . . . .    | 13 |
| 3.7  | Obraz z kamery bez denného svetla, bez použitia polarizačných filtrov . . . . . | 13 |
| 3.8  | Obraz z kamery bez denného svetla, s použitím polarizačných filtrov . . . . .   | 14 |
| 3.9  | Návrh modelu upevnenia polarizačných filtrov . . . . .                          | 14 |
| 3.10 | Popis návrhu mechanizmu upevnenia polarizačných filtrov . . . . .               | 15 |
| 3.11 | Komponent diagram pre plugin . . . . .  | 16 |
| 3.12 | Stavový automat pre main driver . . . . .                                       | 17 |
| 3.13 | Príklady detekcie lasera v obraze z kamery . . . . .                            | 18 |
| 3.14 | Netypické príklady detekcie lasera v obraze z kamery . . . . .                  | 19 |
| 3.15 | Návrh konfiguračného menu . . . . .   | 20 |
| 3.16 | Návrh kalibračného menu . . . . .   | 21 |
| 3.17 | Návrh používateľského rozhrania pre ladiace informácie . . . . .                | 22 |
| 3.18 | Návrh mechaniky hry - ukáž / skry . . . . .                                     | 23 |
| 3.19 | Návrh mechaniky hry - dobrý / zlý . . . . .                                     | 23 |
| 3.20 | Návrh hlavného menu aplikácie . . . . .   | 24 |
| 3.21 | Návrh menu výberu úrovne . . . . .  | 24 |
| 4.1  | Prototyp upevnenia polarizačných filtrov na kameru . . . . .                    | 26 |
| 4.2  | Produkt upevnenia polarizačných filtrov na kameru . . . . .                     | 26 |
| 4.3  | Štatistiky zobrazujúce výkon tretieho riešenia 4.2.2 . . . . .                  | 34 |
| 4.4  | Grafy zobrazujúce výkon tretieho riešenia 4.2.2 . . . . .                       | 35 |
| 4.5  | Štatistiky zobrazujúce výkon štvrtého riešenia 4.2.2 . . . . .                  | 35 |
| 4.6  | Grafy zobrazujúce výkon štvrtého riešenia 4.2.2 . . . . .                       | 36 |
| 4.7  | Kalibrácia pomocou 2 bodov . . . . .  | 37 |

|      |   |    |
|------|---|----|
| 4.8  | Kalibrácia pomocou 4 bodov . . . . .  | 38 |
| 4.9  | Nástroj na testovanie presnosti . . . . .   | 39 |
| 4.10 | Obraz z kamery pri prvom testovacom scenári . . . . .                                     | 39 |
| 4.11 | Výsledky prvého riešenia pri prvom testovacom scenári . . . . .                           | 40 |
| 4.12 | Výsledky druhého riešenia pri prvom testovacom scenári . . . . .                          | 41 |
| 4.13 | Výsledky tretieho riešenia s parametrom $d = 0,9$ pri prvom testovacom scenári . . . . .  | 42 |
| 4.14 | Výsledky tretieho riešenia s parametrom $d = 0,5$ pri prvom testovacom scenári . . . . .  | 43 |
| 4.15 | Obraz z kamery pri druhom testovacom scenári . . . . .                                    | 44 |
| 4.16 | Výsledky prvého riešenia pri druhom testovacom scenári . . . . .                          | 44 |
| 4.17 | Výsledky druhého riešenia pri druhom testovacom scenári . . . . .                         | 45 |
| 4.18 | Výsledky tretieho riešenia s parametrom $d = 0,9$ pri druhom testovacom scenári . . . . . | 46 |
| 4.19 | Výsledky tretieho riešenia s parametrom $d = 0,5$ pri druhom testovacom scenári . . . . . | 47 |
| 4.20 | Obraz z kamery pri treťom testovacom scenári . . . . .                                    | 48 |
| 4.21 | Obraz z kamery umiestnenou nezávislou osobou pri štvrtom testovacom scenári . . . . .     | 49 |
| 4.22 | Mechanika hry - ukáž / skry . . . . .   | 54 |
| 4.23 | Mechanika hry - dobrý / zlý . . . . .   | 54 |

# Listings

|      |  |    |
|------|--|----|
| 4.1  | Abstraktná trieda stavu . . . . .  | 27 |
| 4.2  | Štruktúry pre údaje inicializácie . . . . .                              | 27 |
| 4.3  | Štruktúra pre údaje kalibrácie . . . . .                                 | 28 |
| 4.4  | Stručný pseudokód manager-a stavového automatu . . . . .                 | 28 |
| 4.5  | Kód controller-a pre stavový automat . . . . .                           | 29 |
| 4.6  | Štruktúra BorderPoint . . . . .  | 30 |
| 4.7  | Reprezentácia obdĺžnika ohraničujúceho laserový lúč . . . . .            | 30 |
| 4.8  | Pseudokód iterácie - svietivosť pixlu . . . . .                          | 31 |
| 4.9  | Pseudokód iterácie - farba pixlu . . . . .                               | 32 |
| 4.10 | Pseudokód metódy na detekciu laserového lúča v obraze z kamery . . . . . | 32 |
| 4.11 | Zmena štruktúry obdĺžnika ohraničujúceho laserový lúč . . . . .          | 33 |
| 4.12 | Simulácia kliknutia cez Windows API . . . . .                            | 49 |
| 4.13 | Interface pre objekty podporujúce interakciu s laserom . . . . .         | 51 |
| 4.14 | Časť kódu zobrazujúca nevýhodu pri treťom riešení 4.2.5 . . . . .        | 51 |
| 4.15 | Implementácia interface-u pre interakciu objektu s plugin-om . . . . .   | 55 |

# Úvod

Sem príde úvod neskôr koncom roka.

Put your introduction here. By the way, you can cite articles [2], books [3], websites [4], manuals [5], or other resources. You can reference other chapters, sections, images, etc. using the `cleveref` package like this: chapter 1.

`L`A<sub>T</sub>E<sub>X</sub> basics tutorial: [https://www.overleaf.com/learn/latex/Learn\\_LaTeX\\_in\\_30\\_minutes](https://www.overleaf.com/learn/latex/Learn_LaTeX_in_30_minutes)

# Kapitola 1

## Predchádzajúce práce

V tejto kapitole čitateľa oboznámime s už existujúcimi alebo podobnými riešeniami, či už nekomerčnými alebo komerčnými, ku ktorým sa nám podarilo dopátrať. Čitateľ sa dozvie, čo ponúkajú dané riešenia, ako fungujú, ich výhody i nevýhody a prečo nie sú moc vhodnými riešeniami.

### 1.1 Two-Player Boids Game With Laser Pointer Controllers

Nasledujúca sekcia je spracovaná z internetového zdroja, kde bola práca zverejnená [6].

Cieľom autorov tejto práce bolo rozšíriť boid algoritmus pre dvoch hráčov, kde títo dvaja hráči budú medzi sebou súťažiť o to, kto skonzumuje najviac boid-ov, s tým, že postavy hráčov sú ovládané pomocou červeného a zeleného laserového lúča.

Princíp tejto hry je založený na boid algoritme, ktorý sa učili na predmete *Obsolete-Designing with Microcontrollers*. Pôvodná verzia obsahovala len jedného hráča *predátora* a *boid-y*, ktoré tento hráč mal konzumovať. Boid-y simulovali správanie stáda. Každý boid sa hýbal s prihliadnutím na to, akým smerom sa hýbali ostatní v jeho okolí a snažil sa vyhnúť predátorom. Boid-y navyše vedeli ako blízko sa pri nich nachádza predátor. V prípade, že boli v rádiuse nejakého predátora, tak sa snažili utiecť.

Keďže oblasť štúdia autorov tejto práce bola elektrotechnické a počítačové inžinierstvo (Electrical and Computer Engineering), tak ich práca bola viac zameraná na microcontroller-y a komunikáciu medzi nimi.

Autori vo svojej práci použili projektor na premietanie obrazu hry. USB webkameru na získavanie obrazu projektovaného na plátno. Raspberry Pi 3 B na beh programu na rozpoznávanie obrazu. PIC32 microcontroller, ktorý komunikoval s Raspberry Pi a TFT displejom cez rozhranie UART, jeho úlohou boli výpočty boid algoritmu a vykresľovanie hry na TFT displej. TFT displej pre zobrazovanie hry. Červené a zelené laserové ukazovátka.

Ich systém fungoval nasledovne, získal sa obraz z webkamery, na ktorom sa za pomoci knižnice OpenCV zistili súradnice červeného a zeleného lasera v rámci vytýčenej obdĺžnikovej oblasti. Tieto súradnice potom poslali do PIC32, kde sa zobrali do úvahy vo výpočtoch boid algoritmu.

Laserovým lúčom dávali hráči len pokyn svojim predátorom nech sa pohnú tým smerom, kam ukazujú so svojim laser pointerom, pre plynulejší pohyb (namiesto toho, aby sa predátor v momente premiestnil na súradnice). Ich predátor konzumoval boidy s ktorými prišiel do styku. Celkový počet boid-ov bol fixný <sup>1</sup>. Hra skončila, keď zostali len predátori, t. j. už nebolo čo loviť.

Nevýhody tejto práce sú, že používa príliš špecifický hardvér, nepodporuje ľubovoľnú aplikáciu, ale len tú, ktorú autori implementovali, možnosti použitia a rozširovania sú dosť obmedzené. Z hľadiska výkonu autori uvádzali, že to zvládne ledva 30 snímkov za sekundu pri 40 boid-och a to ešte so zníženým rozlíšením spracovaného obrazu na 450x600, čo ale môže byť dôsledkom slabého výkonu hardvéru. Graficky je aplikácia na úrovni ping-pongu z 80. rokov<sup>2</sup>.

Výhoda je, že podporujú dvoch hráčov, zatiaľ čo v našej práci podporujeme len jeden vstup, jeden červený laserový lúč, v prípade viacerých by sa to rátalo stále ako jeden.

## 1.2 Smokeless Range ©2.0 - Home Simulator

Podarilo sa nám nájsť aj jedno komerčné riešenie. Stojí za ním americká firma Laser Ammo Ltd. s R&D v Izraeli. Podobne ako predchádzajúca sekcia, táto je tiež spracovaná z internetového zdroja [7]. Firma sa venuje hlavne oblastiam civilnej obrany.

Autori tohto produktu ponúkajú virtuálnu strelnicu priamo u vás doma. Postavíte sa pred stenu, zapnete projektor a kameru, ktorú umiestnite do požadovanej vzdialenosti, spustíte softvér, nakalibrujete kameru a môžete trénovať.

Ponúkajú softvér spolu s hardvérom, no hodnota ich produktu sa pohybuje vysoko v trojciferných číslach. Z hardvéru ponúkajú len kameru citlivú na IR svetlo a USB kábel. K tomu si budete musieť zaobstarať počítač spĺňajúci minimálne požiadavky uvedené na ich stránke. Projektor, ktorý musí tiež spĺňať minimálne požiadavky a zariadenie s IR laserom, ktoré sa dá zaobstarať len od ich firmy, samozrejme za ďalšiu symbolickú čiastku. Je dobré podotknúť, že viditeľné lasery alebo iné IR zariadenia nepodporujú.

Softvér funguje z užívateľského pohľadu jednoducho. Je potrebné umiestniť kameru do istej vzdialenosti určenej manuálom. Po spustení softvéru je potrebná kalibrácia kamery (skoro automatická), ktorá prebieha tak, že sa na obrazovku vykreslí desiatka bielych kruhov po obvode a vo vnútri. Užívateľ má možnosť zobrazíť výsledok a prípadne

---

<sup>1</sup>autori uvádzali 40 ks

<sup>2</sup>čierno-biela

manuálne presunúť polohy detekovaných bodov. Takisto je tam hneď možnosť otestovať detekciu, čo nie je nič iné ako len, že zobrazí červenú bodku tam kde detekovalo IR laser. Na záver sa dá táto kalibrácia uložiť a znova použiť, pokiaľ zariadenia nezmenili svoju polohu. [8]

Problémy a nevýhody tohto riešenia sú podpora len konkrétneho programu s obmedzenou sadou funkcionality, vysoko nákladné, malá rozšíriteľnosť, závislosť produktu od iných (hlavne IR laser), nefunkčnosť s bežne používanými laser pointermi.

Výhodou by mohol byť IR laser, ktorý sa dá lepšie zachytiť oproti viditeľnému laserovému lúču (menej interferencie z okolitého svetla), no je to i nevýhodou, lebo si vyžaduje kameru citlivú na IR svetlo, čím bežne dostupné kamery nedisponujú.

# Kapitola 2

## Východiska práce

V tejto kapitole sa čitateľ dozvie niečo o použitých technológiách a prípadnej problematike súvisiacej s našou prácou.

### 2.1 Použité technológie

#### 2.1.1 Unity game engine

Unity game engine, známy aj ako Unity 3D je multiplatformový počítačový softvér na tvorbu softvéru s rôznorodým použitím v odvetví herného, automobilového, filmového priemyslu, architektúry, dizajnu, virtuálnej a rozšírenej reality a rôzne real-time aplikácie.

Samotné Unity je naprogramované v jazyku C++, ale v rámci Unity sa používa programovací jazyk C#.

Najviac známe príklady použitia Unity sú v oblasti herného priemyslu ako *Escape from Tarkov*, *Fall guys*, *MARVEL SNAP*, *Cult of the Lamb* a pre mobily najstaršia a kedysi nahrávanejšia hra *Subway surfers* [9]. Ale aj vo vzdelávaní, Univerzita v Miami použila XR aplikácie vytvorené v Unity na tréning chirurgov [10]. Aj v automobilovom priemysle, automobilky BMW a Volvo používajú Unity na simulácie scenárov pre vývoj autonómneho riadenia. V oblasti filmového priemyslu stojí za zmienku CGI voda vo filme *Avatar: The Way of Water*, ktorá bola vytvorená s pomocou Unity [11].

#### 2.1.2 C#

C# je moderný, vysoko úrovňový, typový, objektovo orientovaný programovací jazyk vytvorený Microsoftom. Spočiatku closed-source, neskôr open-source. Syntaxou je podobný jazykom C a C++. Používa sa v .NET aplikáciach, Windows Forms a Universal Windows Platform<sup>1</sup> [12].

---

<sup>1</sup>známe aj pod skratkou UWP



## 2.2 Spracovanie obrazu a počítačové videnie

S rapídny rastom technológií v oblasti autonómneho systému riadenia, autonómnych strojov, biomedicíny a monitorovacích systémov rastie aj objem dát a potreba ich spracovávať. Spracovávať tieto dáta bez pomoci počítačov je prakticky nemožné. Analýzu obrazu preto chceme vykonávať za pomoci počítačov. So spracovaním obrazu alebo počítačovým videním sa čitateľ už stretol napríklad v mobilných aplikáciach rôzne filtre pri fotografovaní alebo detekcia vozidiel na dopravných kamerách. Počítačové videnie sa snaží o podobný efekt ľudského videnia a porozumenia obrazu v elektronickej podobe v počítači.

Jednými zo základných charakteristík porozumenia obrazu a problémov počítačového videnia sú zachytávanie obrazu, segmentácia obrazu, prispôbenie modelu a predikcia pohybu.

### 2.2.1 Počítačové videnie nie je vôbec jednoduché

**Strata informácií pri prechode z 3D do 2D** nie je nič prekvapujúce. Vyskytuje sa napríklad pri typických záznamových zariadeniach, ako sú kamery alebo ľudské oko. Kamera zachytáva obraz z 3D priestoru a ukladá ho už len v 2D podobe. Ako príklad si zoberme malý objekt v blízkosti kamery, ktorý je vidno rovnako ako veľký objekt vzdialený od kamery.

**Interpretácia obrazu** je problém, ktorý ľudia riešia bez uvedomenia si každý deň. Keď sa človek snaží pochopiť obraz, jeho pochopenie sa stavia na základe predchádzajúcich poznatkov a skúseností. K tomuto sa používa umelá inteligencia, aby obohatila počítače schopnosťou porozumieť pozorovaniam. Pod interpretáciou obrazu môžeme chápať funkciu

$$\textit{interpretácia: dáta obrazu} \longrightarrow \textit{model}$$

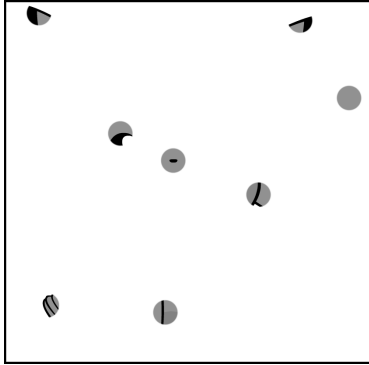
Model je nejaká špecifická doména v ktorej pozorované objekty majú zmysel. Ako príklady môžu byť rieky v satelitnej snímke alebo súčiastky v nejakej výrobní fabrike pri kontrole kvality. Navyše, interpretácií toho istého obrazu môže byť viac než len jedna. Interpretácia obrazu v počítačovom videní môže byť chápaná ako inštancia sémantiky. V praxi to znamená to, že ak algoritmus, ktorý sa snaží porozumieť obrazu vie do akej domény patrí pozorovaný objekt, tak automatická analýza môže byť použitá pri riešení komplikovanejších problémov.

**Šum** je prítomný v každom meraní reálneho sveta. Existencia šumu si žiada matematické nástroje, ktoré sú schopné vysporiadať sa z neistotou, ako príklad nám poslúži teória pravdepodobnosti.

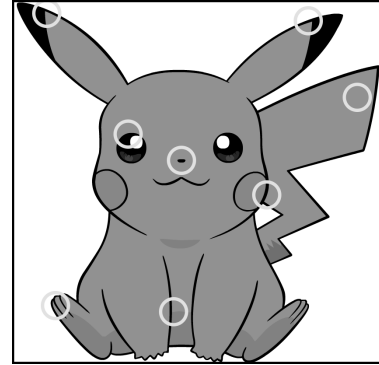
**Priveľa dát.** Obrázky zo záznamových zariadení sú veľké. Síce technologické pokroky v procesoroch, grafických procesoroch a pamäti umožňujú spracovávať čoraz

viacej informácií za čoraz kratší čas, efektívnosť je dôležitá a žiaľ ešte stále veľa aplikácií zaostáva v real-time výkonoch.

**Lokálny pohľad oproti globálnemu pohľadu.** Bežné algoritmy analýzy obrazu pracujú nad konkrétnym prvkom (napríklad jeden pixel v obrázku) a jeho susedmi. Počítač vidí obraz cez kľúčové dierky, čo v dôsledku robí porozumenie obrazu na globálnej úrovni veľmi ťažkým.



(a) Ilustrácia pohľadu cez kľúčové dierky. Iba z tejto informácie je veľmi ťažké uhádnuť kontext obrázka.



(b) Pre človeka veľmi jednoduché porozumieť čo sa nachádza na obrázku, keď má k dispozícii globálny pohľad

Obr. 2.1: Ilustrácia lokálneho a globálneho pohľadu [1]

## 2.2.2 Reprezentácia a analýza obrazu

Porozumenie obrazu počítačom môžeme chápať ako pokus o nájdenie relácie vstupného obrazu s predtým zavedenými modelmi pozorovaného sveta. Transformácia vstupného obrazu do modelu danej domény redukuje informáciu obsiahnutú v obraze na relevantnú informáciu pre danú doménu. Tento proces je rozdelený do niekoľkých krokov a úrovní reprezentujúcich daný obraz. Na najspodnejšej vrstve máme surové dáta, kde vyššie úrovne interpretujú tieto dáta. Hierarchia reprezentácie obrazu je často kategorizovaná na low-level spracovanie obrazu a high-level porozumenie obrazu.

Metódy low-level spracovania zvyčajne vedú o obsahu spracovávaného obrazu veľmi málo. V týchto prípadoch je informácia o obsahu poskytnutá high-level algoritmom alebo priamo človekom, ktorý rozumie danej doméne. Low-level metódy zahŕňajú kompresiu, pedspracovanie pre filtrovanie šumu, extrakciu hrán a ostrenie obrazu. Low-level metódy sa v značnej miere prekrývajú s oblasťou digitálneho spracovania obrazu.

High-level spracovanie je založené na vedomostiach, cieľoch a plánoch ako dosiahnuť tieto ciele. Často sa dajú a aj používajú metódy pomocou umelej inteligencie. High-level počítačové videnie sa snaží napodobniť ľudské poznanie a schopnosť robiť rozhodnutia podľa informácie obsiahnutej v obraze. High-level počítačové videnie začína nejakým formálnym modelom domény reálneho sveta a potom vnímaná realita vo forme

digitalizovaného obrazu je porovnávaná s týmto modelom. Hľadá sa zhoda s týmto modelom, prípadne čiastočná zhoda. Počítač používa low-level spracovanie obrazu na nájdenie informácie, aby aktualizoval svoj model. Tento proces je opakovaný iteratívne. Porozumením obrazu sa stane spolupráca medzi týmito procesmi.

Obraz spracovávaný počítačom je najskôr digitalizovaný, kde môže byť reprezentovaný maticou o rozmerov  $m \times n$  kde  $m, n \in \mathcal{N} - 0$ . Jednotlivé prvky matice sú väčšinou vektory troch hodnôt (červenej, zelenej a modrej) z intervalu  $\langle 0, 255 \rangle$ . V prípade videí sú dané dáta asociované s rýchlosťou snímkovania. V závere sú tieto dáta na low-level úrovni reprezentované množinou alebo postupnosťou matíc.

Taký bežný scenár spracovania obrazu zachytáva zachytávanie obrazu pomocou senzoru (napríklad kamerou), ktorý je digitalizovaný. Potom sa zredukuje šum pomocou pred spracovania obrazu. Následne sa môžu použiť metódy na zvýraznenie niektorých črt ako napríklad extrakcia hrán.

Ďalej nasleduje **segmentácia obrazu**, kde sa počítač snaží oddeliť objekty od pozadia. Segmentácia môže byť úplná alebo čiastočná. Úplná segmentácia je možná iba pre jednoduché úlohy ako rozpoznávanie tmavých objektov od bieleho pozadia. Tento prístup sa používa pri optical character recognition, OCR. V komplikovanejších prípadoch, čo je zvyčajne každý bežný prípad, sa low-level metódy starajú o čiastočnú segmentáciu. Hľadanie hraníc objektov je typickým príkladom low-level čiastočnej segmentácie.

Low-level spracovanie a high-level spracovanie obrazu sa líšia v reprezentácií dát. Ako sme uvádzali, low-level dáta sa skladajú z matíc, kde prvkami sú napríklad vektory reprezentujúce farbu pixlu. High-level dáta sú zvyčajne vyjadrené v symbolickej forme. Sú extrahované z obrazu za pomoci low-level metód je ich výrazne menej v porovnaní s low-level dátami. High-level dáta reprezentujú vedomosti o obsahu obraze napríklad veľkosť objektu, tvar objektu, vzájomnú reláciu objektu a obrazu.

Úlohou Sekcie 2.2 je oboznámenie sa z danou problematikou, avšak problematika počítačového videnia a spracovania obrazu siaha oveľa hlbšie než si môžeme momentálne dovoliť. V prípade hlbšieho záujmu čitateľa, odporúčame siahnuť po knihe o spracovaní obrazu [13], z ktorej sme spracovali len maličkú časť.

# Kapitola 3

## Špecifikácia požiadaviek a návrh systému

V tejto kapitole uvedieme špecifikáciu požiadaviek a postupne oboznámime čitateľa s celkovým návrhom, návrhom upevnenia polarizačných filtrov, návrhom plugin-u a aplikácie, do ktorej bude tento plugin neskôr integrovaný.

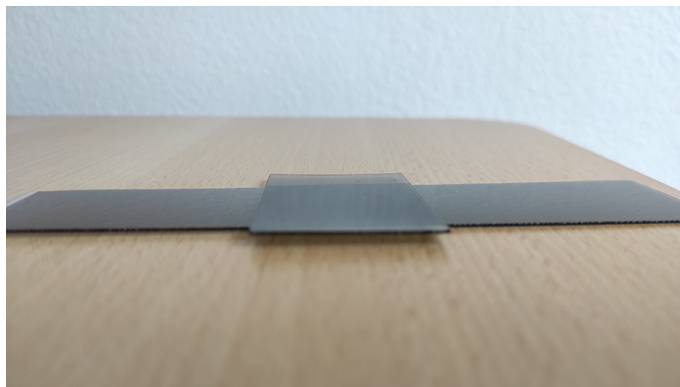
### 3.1 Špecifikácia požiadaviek

Plugin bude pracovať na platforme Unity. Plugin bude spracovávať obraz z kamery. V spracovávanom obraze bude zisťovať polohu laserového lúča. Na zistenej polohe laserového lúča bude simulovať kliknutie ľavým tlačidlom myši v aplikácii, v ktorej bude použitý.

Na použitie tohto plugin-u v prípade vývoja aplikácie je nutné mať nainštalovaný program Unity<sup>1</sup>. Ďalej je potrebné stiahnuť samotný plugin z GitHub repozitára a následne importovať do projektu. Z hľadiska hardvéru, použitie tohto plugin-u si vyžaduje počítač s operačným systémom Windows, Mac OS alebo Linux na ktorom je nainštalovaný program Unity. Počítač by mal mať dostatočný výpočtový výkon. K počítaču musí byť zabezpečené pripojenie projektora a webkamery alebo kamery. Na webkamere/kamere musí byť upevnená dvojica vhodne prekřížených polarizačných filtrov. Medzi týmito dvoma filterami nesme byť žiadna medzera (t. j. musia byť na sebe ako je uvedené na obrázku 3.1).

---

<sup>1</sup>verzia 2021.3.10f1 a vyššie



Obr. 3.1: Ilustračný obrázok polarizačných filtrov umiestnených na sebe

Plugin bude schopný pracovať s ľubovoľnou<sup>2</sup> webkamerou a ľubovoľným projektorom. Plugin bude fungovať nezávisle od projektu. Jeho použitie bude čo najjednoduchšie, aby programátorovi stačilo len tento plugin importovať do projektu a nastaviť zopár hodnôt. Je odporúčané používať systém v miestnosti s nie príliš silným osvetlením, hlavne pri obraze premietanom projektorom na stenu či plátno.

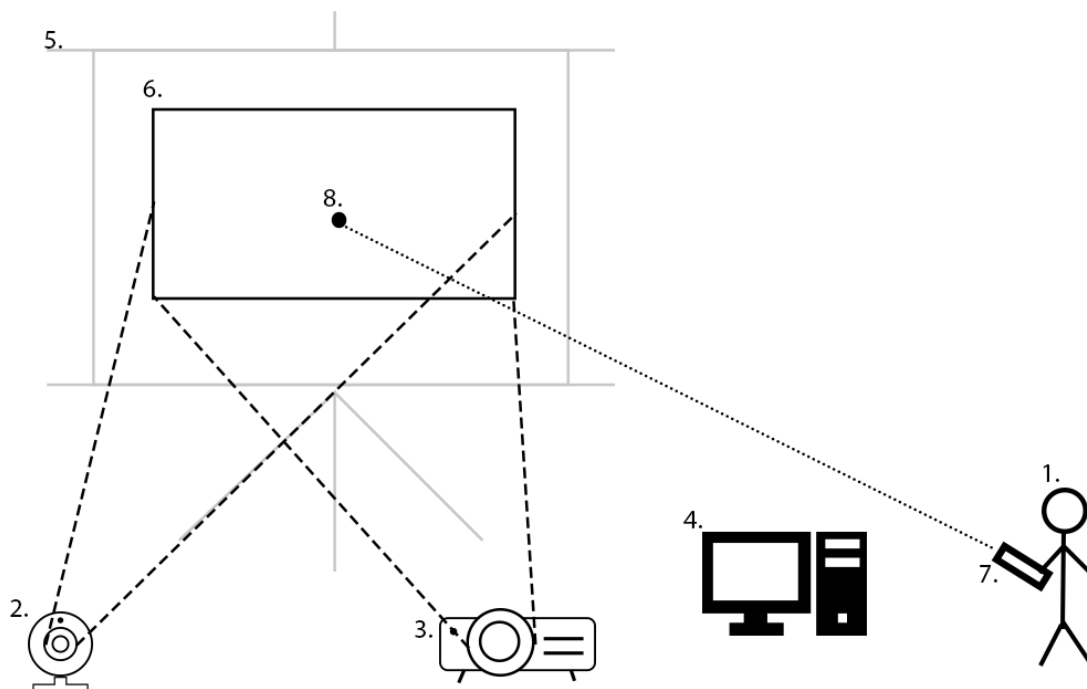
## 3.2 Návrh systému

### 3.2.1 Celkový návrh

Ako to celé bude fungovať čitateľovi popíšeme za pomoci Obrázka 3.2. V celom systéme vystupujú traja hlavný aktéri. Projektor (3.), ktorý premieta obraz z počítača (4.) na premietaciu plochu (5.). Kamera (2.), ktorá sníma obraz premietaný z projektora (6.) a osoba (1.), ktorá za pomoci zariadenia tvoriaceho laserový lúč (7.) ovláda niektoré časti aplikácie, ktorá je spustená na počítači a premietaná z projektora. (8.) znázorňuje polohu laserového lúča.

---

<sup>2</sup>podporovanou Unity



Obr. 3.2: Schéma celkového návrhu

### 3.2.2 Upevnenie polarizačných filtrov

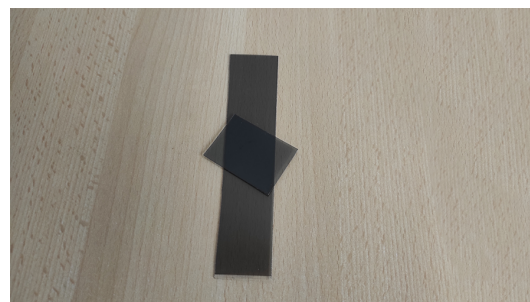
V obraze, ktorý budeme spracovávať nás zaujíma len jedna vec, červený laserový lúč. Zisťovať polohu laserového lúča v obraze nie je až také jednoduché, pretože v obraze sa nachádza veľa svetelného šumu<sup>3</sup>, ktoré nám toto komplikuje. Zároveň každá kamera sníma obraz/svetlo inak. Potrebujeme teda spôsob, ako sa zbaviť zbytočného svetla, čo nám zisťovanie pozície laserového lúča v obraze uľahčí. Na uľahčenie tohto problému sme sa rozhodli použiť dvojicu vhodne<sup>4</sup> prekrížených polarizačných filtrov, zobrazených na Obrázku 3.3b.

<sup>3</sup>nežiadúce okolité svetlo

<sup>4</sup>pod pojmom vhodne prekrížené, máme na mysli taký uhol prekríženia polarizačných filtrov, ktorý zredukuje nadbytočné svetlo do takej miery, že laserový lúč je stále dostatočne a zreteľne viditeľný



(a) 2ks polarizačných filtrov



(b) príklad vhodného prekríženia

Obr. 3.3: Polarizačné filtre

Ako bolo spomenuté v sekcii 3.1, tieto filtre musia byť priamo na sebe, bez žiadnej medzery medzi nimi. V opačnom prípade by sa to správalo, ako keby tam bol len jeden. Dvojica vhodne prekrížených filtrov zredukuje svetlo, ktoré sa cez nich dostane a keďže svetlo z laserového lúča je dosť silné svetlo, tak v tomto filtrovanom obraze bude dobre viditeľné. Porovnanie rozdielu pri použití a nepoužití polarizačných filtrov za rôznych svetelných podmienok pri premietaní Obrázku 3.4 cez projektor si čitateľ môže pozrieť na Obrázkoch 3.5 až 3.8.



Obr. 3.4: Obrázok, premietaný projektorom v obrázkoch 3.5 - 3.8

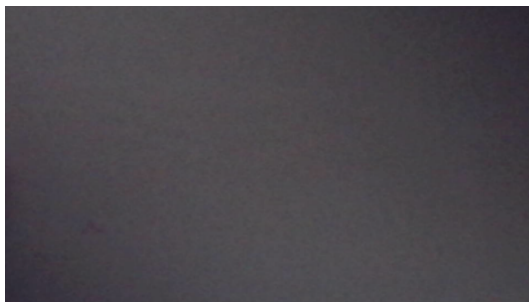


(a) bez lasera



(b) s laserom

Obr. 3.5: Obráz z kamery za denného svetla, bez použitia polarizačných filtrov



(a) bez lasera



(b) s laserom

Obr. 3.6: Obráz z kamery za denného svetla, s použitím polarizačných filtrov



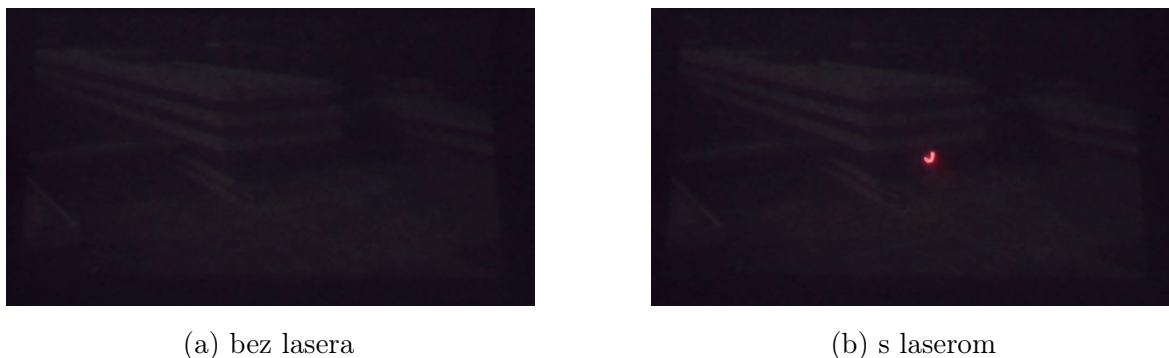
(a) bez lasera



(b) s laserom

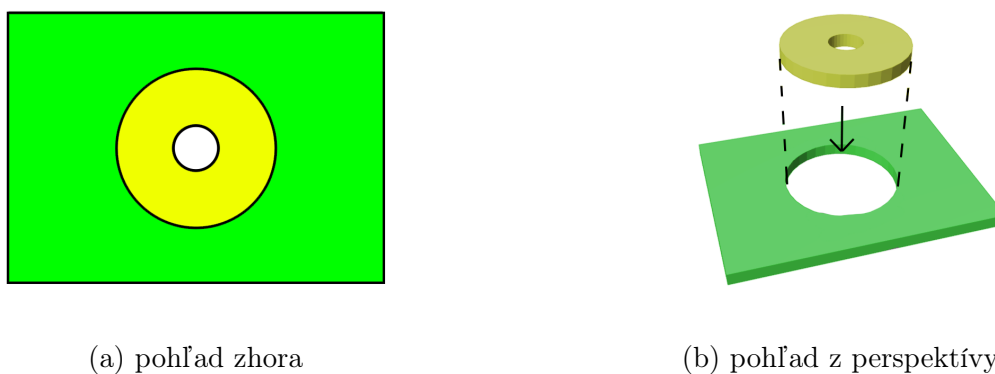
Obr. 3.7: Obráz z kamery bez denného svetla, bez použitia polarizačných filtrov





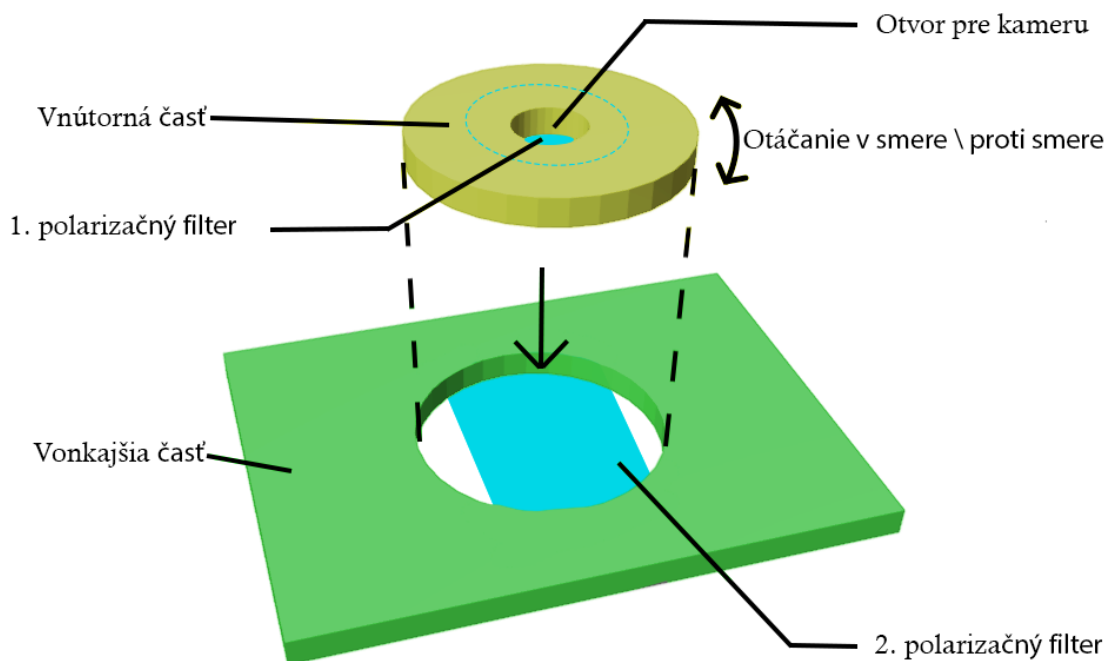
Obr. 3.8: Obráz z kamery bez denného svetla, s použitím polarizačných filtrov

Pred samotným návrhom plugin-u a aplikácie je potrebné vymyslieť a zhotoviť návrh upevnenia dvojice polarizačných filtrov na kameru. Kamery sú rôznych druhov a rôznych veľkostí, preto návrh upevnenia polarizačných filtrov je pre našu prácu veľmi špecifický a nebude ho možné uplatniť pri použití inej kamery, ako používame v našej práci, samotný plugin to však nijako neovplyvní. Keďže nevieme dopredu, aké bude vhodné<sup>5</sup> prekríženie polarizačných filtrov pre každý scenár, tak potrebujeme mechanizmus, ktorý nám jednoducho a počas prevádzky umožní ľubovoľne meniť uhol prekríženia polarizačných filtrov, t. j. otáčať jeden z nich. Vymysleli sme následovný mechanizmus. Máme jeden obdĺžnikový útvar. Tento útvar je ďalej rozdelený na 2 kusy tým že sa doň vyreže kruh (viď Obrázok 3.9a). Tieto dve časti do seba pasujú, ako je naznačené na Obrázku 3.9b. Vnútorňa časť je kruhového tvaru, obsahujúca dieru pre kameru, ktorá sa vloží do vonkajšej časti a umožňuje voľne sa otáčať vo vonkajšej časti. Týmto mechanizmom sme zabezpečili, že polarizačné filtre sú umiestnené priamo na sebe, ako sme špecifikovali v sekcii 3.1 a zároveň nám poskytujú možnosť ľubovoľne meniť uhol prekríženia polarizačných filtrov počas prevádzky. Tento model sa následne už len upevní na kameru. Na Obrázku 3.10 je pre čitateľa zobrazený celkový návrh mechanizmu v 3D aj s popisom.



Obr. 3.9: Návrh modelu upevnenia polarizačných filtrov

<sup>5</sup>rôzne prostredia si vyžadujú rôzny uhol prekríženia polarizačných filtrov



Obr. 3.10: Popis návrhu mechanizmu upevnenia polarizačných filtrov

### 3.2.3 Návrh pluginu

Plugin chceme navrhnuť čo najviac modulárne, aby sa dal v prípade potreby jednoducho rozširovať resp. modifikovať. Plugin bude pozostávať z dvoch základných častí, z jadra a z utilít.

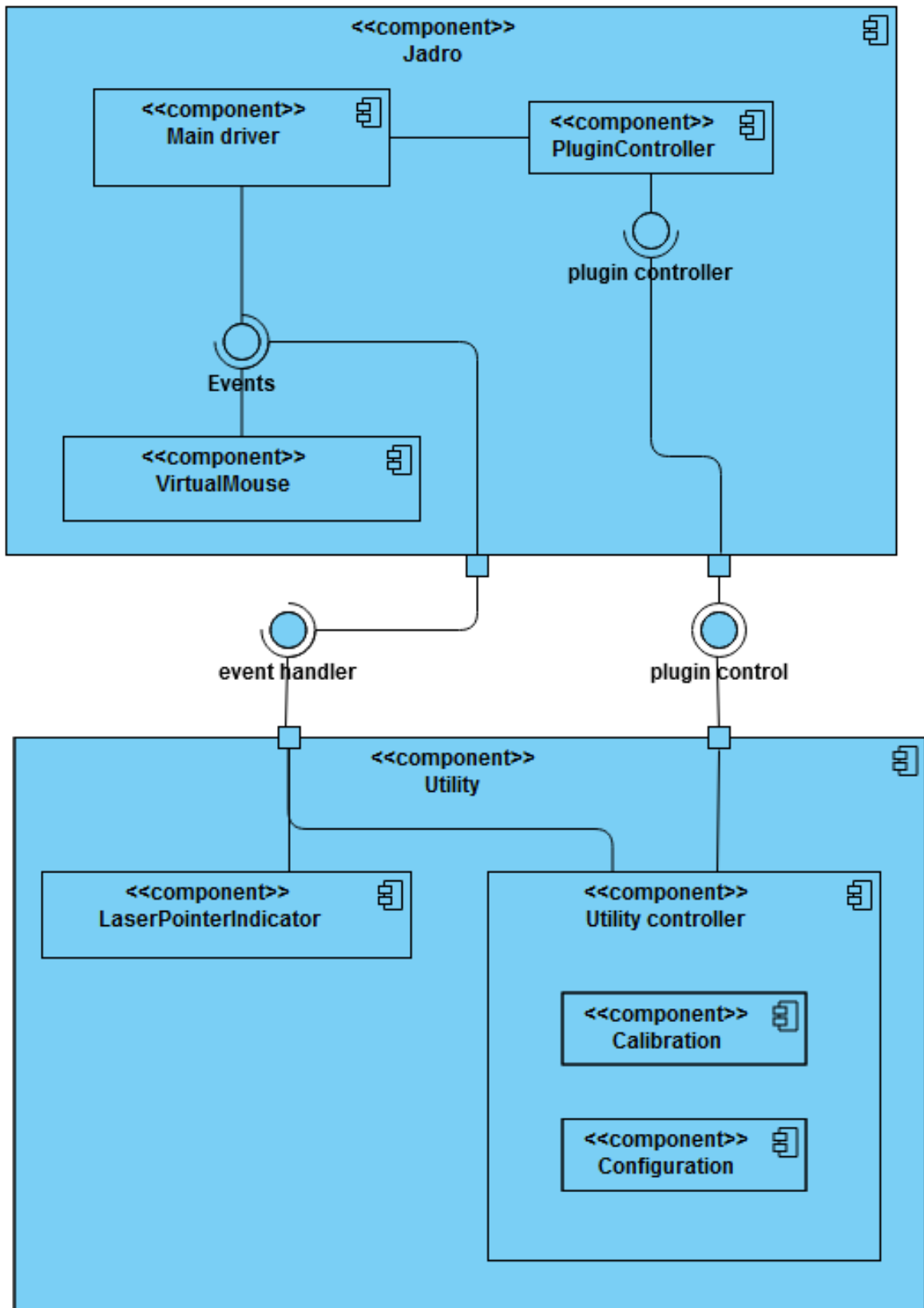
Jadro bude zastrešovať tri ďalšie komponenty. Prvý z týchto troch komponentov, ďalej len **main driver**, bude fungovať ako automat. Jednou z jeho hlavných úloh bude detekcia laserového lúča v obraze, ktorý mu bude poskytovaný z kamery a s tým spojená kalibrácia a transformácia súradníc, kde bol detegovaný laserový lúč v spracovávanom obraze do súradníc obrazovky. T.j. preklad súradníc z jedného sveta (obraz poskytovaný kamerou vo svojom rozlíšení) do druhého sveta (obrazovka, na ktorej je spustená aplikácia vo svojom rozlíšení). **Main driver** bude pri významných bodoch vyvolávať udalosti, na ktoré môžu reagovať ostatné komponenty alebo systémy.

Druhý zo spomínaných komponentov, ďalej len **virtual mouse** bude reagovať na udalosti z **main driver-a** a bude mať na starosti simuláciu kliknutia ľavým tlačidlom myši.

Nakoniec, tretí komponent bude slúžiť ako ovládač, ďalej ako **plugin controller**, ktorý bude môcť používať iný systém alebo programátor na ovládanie plugin-u cez poskytnuté verejné metódy. Popísaný návrh komponentov je znázornený na Obrázku 3.11.

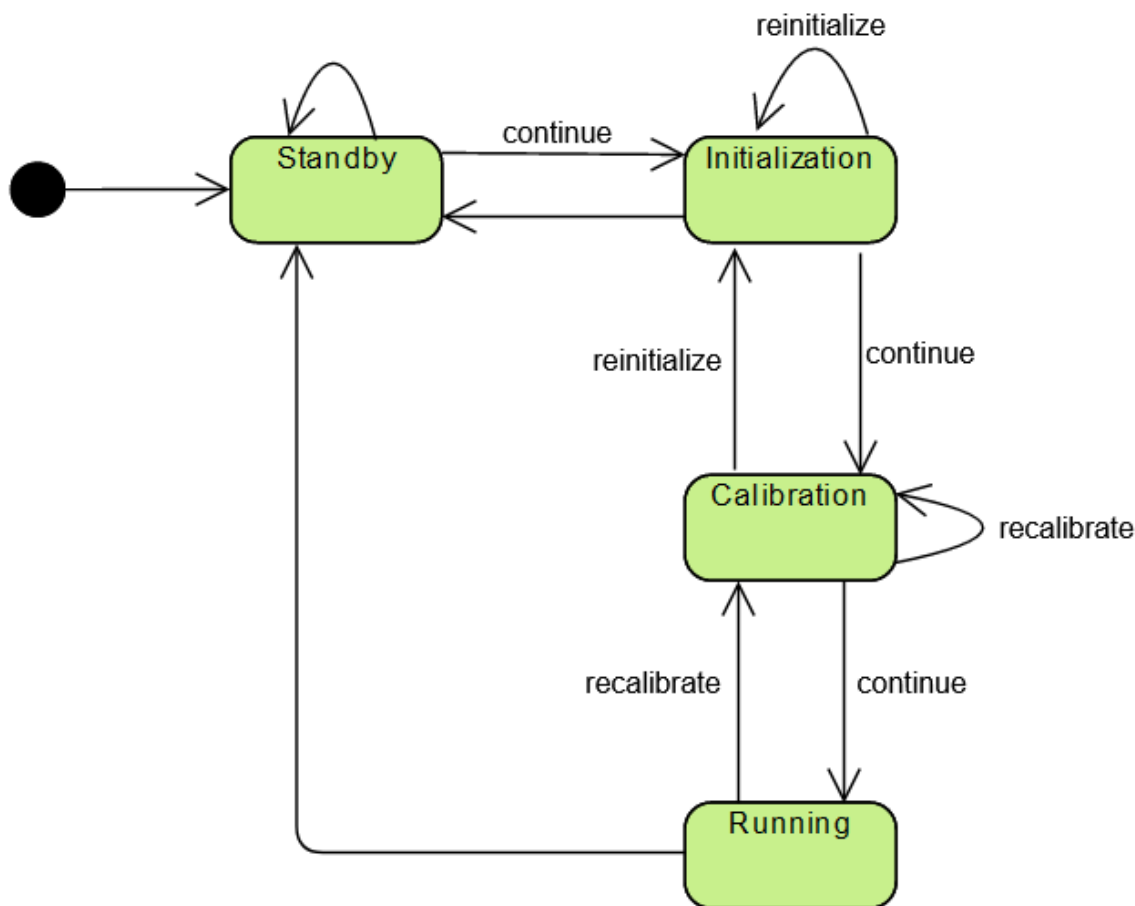
V nasledujúcom odseku čitateľovi bližšie popíšeme návrh automatu pre komponent

**main driver** aj s jeho vlastnosťami a prechodmi.



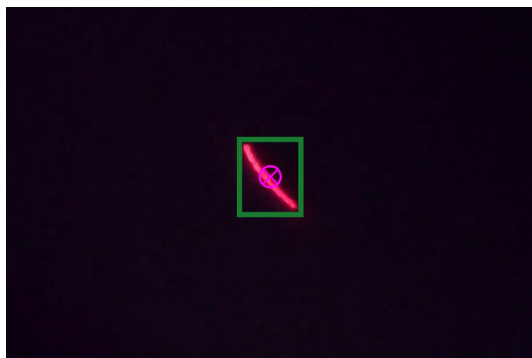
Obr. 3.11: Komponent diagram pre plugin

**Main driver** obsahuje celkovo štyri stavy *Standby*, *Initialization*, *Calibration*, *Running*. **Main driver** začína v stave *Standby*, ktorý je zároveň počiatočný. V tomto stave sa v skutočnosti nič nedeje, ako indikuje meno stavu, tento stav slúži len ako taka čakáreň. Zo stavu *Standby* sú dovolené len prechody do stavov *Standby* a *Initialization*. Po prechode zo stavu *Standby* do stavu *Initialization* sa nastaví počiatočné parametre pre správne fungovanie plugin-u. Zo stavu *Initialization* sú dovolené len prechody do stavov *Initialization* pre opätovnú inicializáciu, *Standby* pri zrušení a *Calibration* v prípade pokračovania. Ďalší stav v poradí, *Calibration*, slúži na kalibráciu plugin-u. V tomto stave sa začne proces kalibrácie, vypočítajú sa hodnoty pomocou ktorých sa neskôr budú transformovať súradnice. Zo stavu *Calibration* sú povolené len prechody do stavov *Calibration* pre prípad opätovnej kalibrácie a *Running* v prípade pokračovania. Nakoniec zostáva stav *Running*, v ktorom sa bude nachádzať automat väčšinu času prevádzky. Bude zabezpečovať hlavne detekciu laserového lúča v obraze z kamery. Zo stavu *Running* sú dovolené len prechody do stavov *Calibration* v prípade opätovnej kalibrácie a *Standby*. Navrhnutý automat, ktorý sme čitateľovi práve popísali aj s prechodmi je znázornený na Obrázku 3.12.

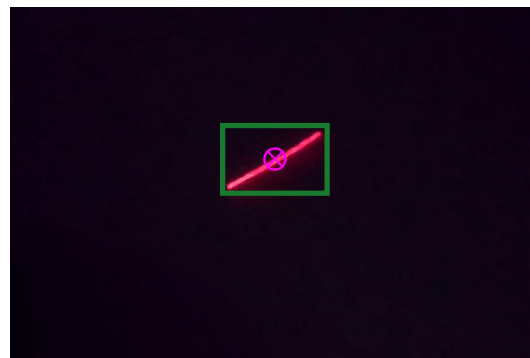


Obr. 3.12: Stavový automat pre main driver

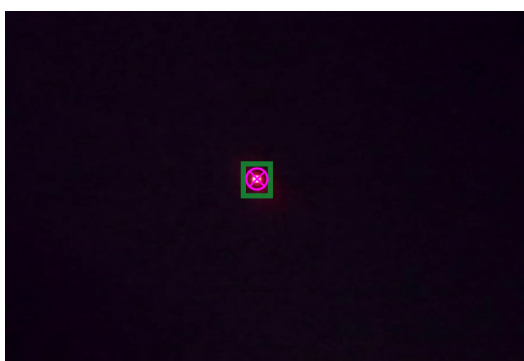
Detekciu laserového lúča sme sa rozhodli, že budeme riešiť tak, že detegovaný laserový lúč enkapsulujeme do obdĺžnika, ako je vidieť na Obrázkoch 3.13. Ideálny prípad by bol, keby detegovaný laserový lúč bol kruhového tvaru ako je znázorňené na Obrázku 3.13c, no museli sme sa zamyslieť aj nad prípadmi, ktoré nezdieľajú takéto ideálne črty. Pre takéto prípady sme sa rozhodli použiť rovnaký prístup ako pri ideálnych, t. j. enkapsulácia do obdĺžnika (viď Obrázok 3.13a a 3.13b).



(a) príklad detekcie 1



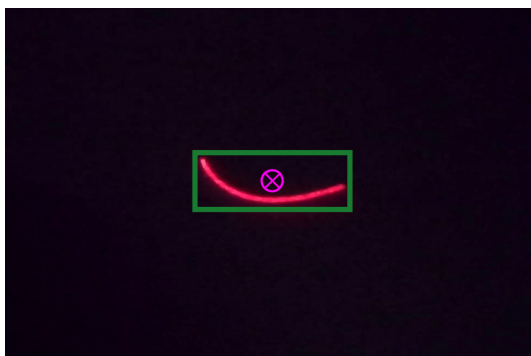
(b) príklad detekcie 2



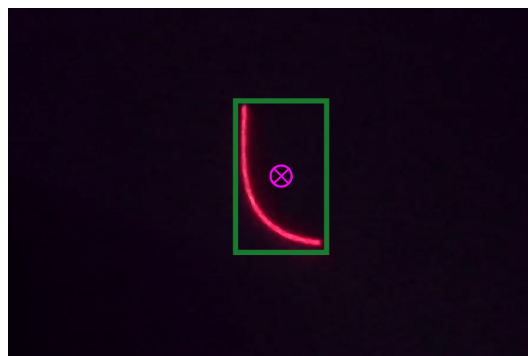
(c) príklad ideálnej situácie

Obr. 3.13: Príklady detekcie lasera v obraze z kamery

Avšak toto so sebou prináša aj isté anomálie, ktoré sú znázornené na Obrázkoch 3.14, kde by výsledok detekcie nedával moc zmysel. Pravdepodobnosť, že by takáto situácia nastala je nízka, lebo mieriť presne s laserovým zariadením do diaľky (aspoň pre ľudí) je náročnejšie, ako sa na prvý pohľad môže zdať. Preto osoba s laserovým zariadením sa bude snažiť skôr o presnosť ako rýchly a nezmyselný pohyb vyúsťujúci do spomínaných anomálii. Napríklad v situáciách zobrazených na Obrázkoch 3.14b, 3.14c, 3.14d, 3.14e, 3.14f, 3.14g a 3.14h by sme bežne uvažovali, že detekcia by bola niekde na trase laserového lúča. Na obrázkoch 3.13 a 3.14 zelený obdĺžnik reprezentuje enkapsuláciu laserového lúča. Fialový kruh s krížom reprezentuje stred, respektíve výsledné súradnice detekcie laserového lúča v jednom snímku obrazu z kamery.



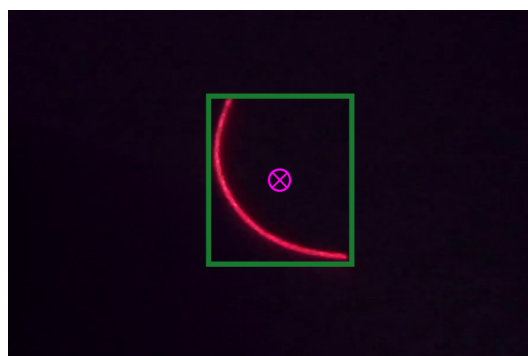
(a) laser tvaru konvexnej paraboly 1



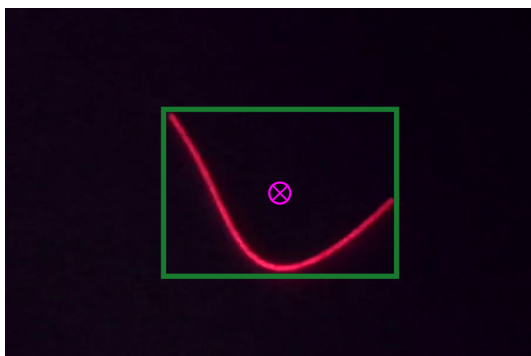
(b) laser tvaru konvexnej paraboly 2



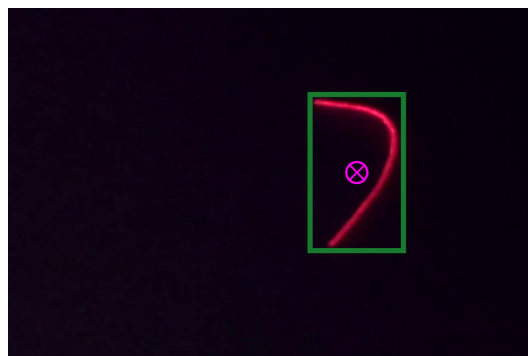
(c) laser tvaru konvexnej paraboly 3



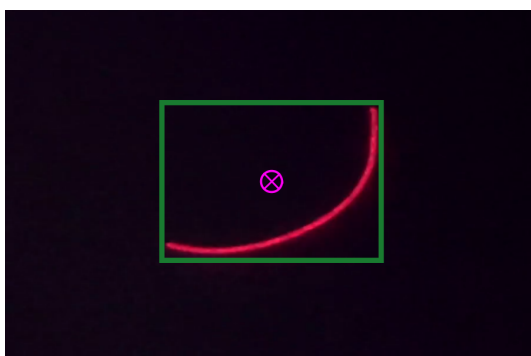
(d) laser tvaru konvexnej paraboly 4



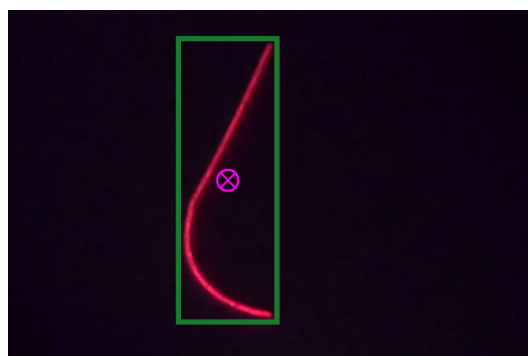
(e) laser tvaru konvexnej paraboly 5



(f) laser tvaru horizontálne preklopeného L



(g) laser tvaru konvexnej paraboly 6



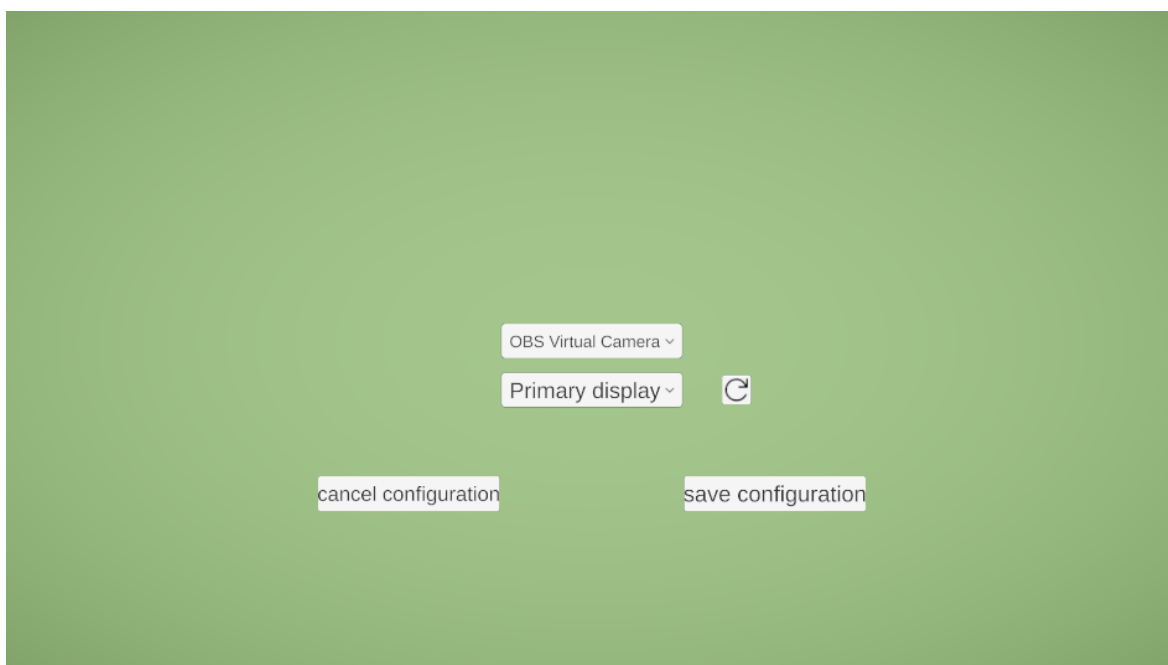
(h) laser tvaru L

Obr. 3.14: Netypické príklady detekcie lasera v obraze z kamery

Druhá časť utility obsahuje doplnujúce prostriedky pre plugin. Zastrešuje komponenty pre jednoduché konfiguračné menu, jednoduché kalibračné menu, jednoduché používateľské rozhranie pre ladiace informácie a indikátor miesta detekcie laserového lúča.

### Utility - konfiguračné menu

Konfiguračné menu pozostáva z dvoch tlačidiel typu dropdown. Prvý dropdown obsahuje záznamové zariadenie<sup>6</sup>. Druhý dropdown obsahuje obrazovky v počítači<sup>7</sup>. Ďalej obsahuje tlačidlá *save configuration*, ktorým sa uložia zvolené hodnoty a zavrie sa konfiguračné menu a *cancel configuration*, ktoré ukončí konfiguračné menu bez akýchkoľvek zmien. Nakoniec ešte tlačidlo pre manuálnu obnovu údajov.



Obr. 3.15: Návrh konfiguračného menu

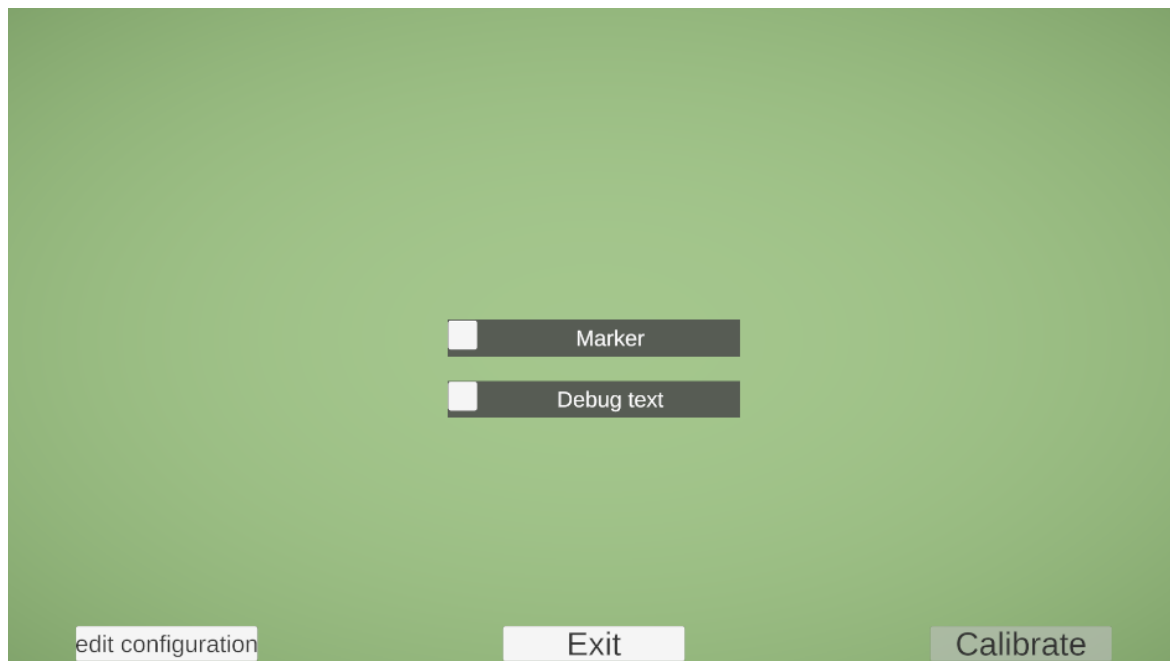
### Utility - kalibračné menu

Kalibračné menu pozostáva z troch tlačidiel *exit*, *edit configuration*, *calibrate*. Tlačidlo *exit* zavrie kalibračné menu. Tlačidlo *edit configuration* bude vždy dostupné a zabezpečí otvorenie konfiguračného menu. Tlačidlo *calibrate* bude dostupné len vtedy, ak sa nakonfiguroval plugin. To znamená, že kým sa nenakonfiguruje plugin cez konfiguračné menu, tlačidlo *calibrate* je nedostupné. Ďalej obsahuje dve tlačidlá typu checkbox. Prvý checkbox *Marker* povoľuje zobrazovanie indikátora laserového lúča na obrazovke.

<sup>6</sup>kameru alebo webkameru

<sup>7</sup>hlavná a externá obrazovka, prípadne ďalšie

Druhý checkbox *Debug text* povoľuje zobrazovanie používateľského rozhrania pre ladiace informácie.



Obr. 3.16: Návrh kalibračného menu

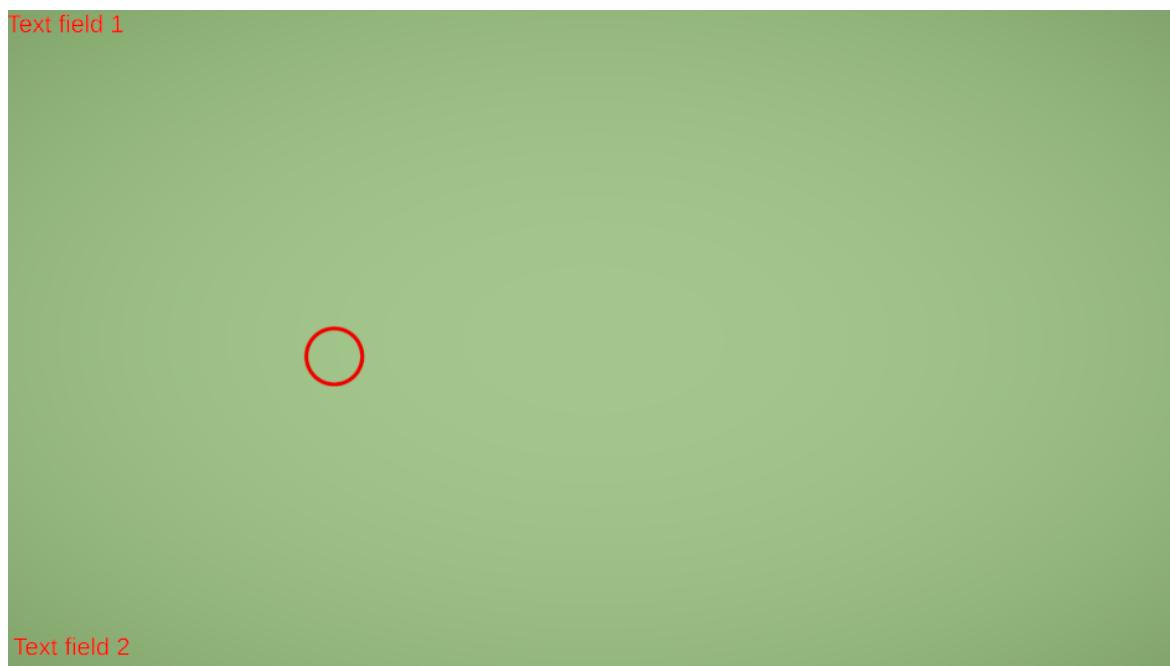
### Utility - jednoduché používateľské rozhranie pre ladiace informácie

Používateľské rozhranie pre ladiace informácie obsahuje dve textové polia. Prvé textové pole je zakotvené v ľavom hornom rohu. Obsahuje informácie o rozlíšení kamery, rozlíšení obrazovky na ktorej je spustená aplikácia a počet snímkov<sup>8</sup> za sekundu. Druhé textové pole zakotvené v ľavom dolnom rohu obsahuje informáciu o simulovanom kliknutí na interpretovaných súradniciach. Tieto súradnice sa budú zobrazovať v druhom textovom poli. Ďalej obsahuje kruhový indikátor laserového lúča, ktorý sa zobrazí na pozícii, kde bol detegovaný laserový lúč.

---

<sup>8</sup>počet snímkov za sekundu záznamového zariadenia, t. j. kamery





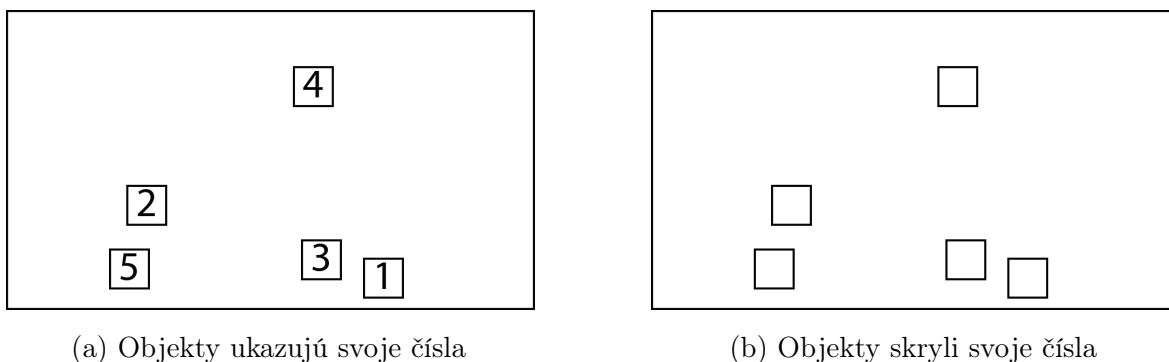
Obr. 3.17: Návrh používateľského rozhrania pre ladiace informácie

Návrhy vyššie spomenutých používateľských rozhraní sme zhotovili priamo v Unity Editore.

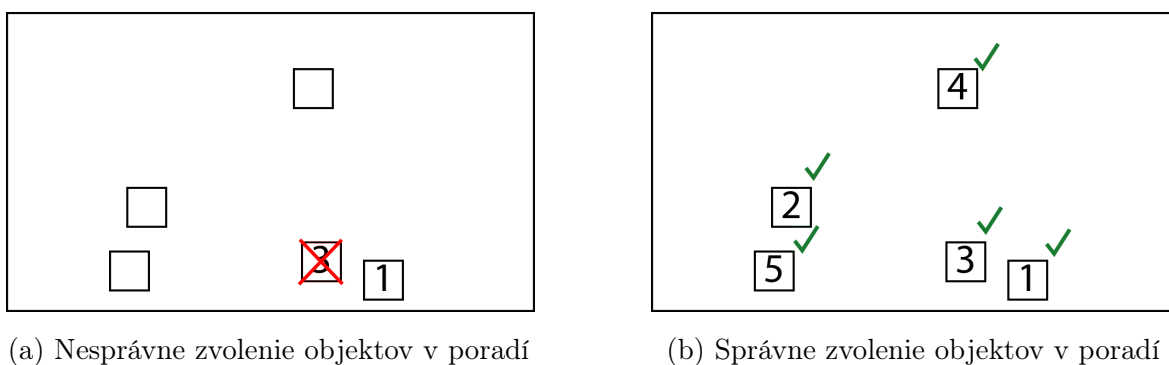
### 3.2.4 Návrh aplikácie

Aplikáciu môžeme navrhnuť úplne nezávislú od plugin-u, skoro bez myšlienky existencie plugin-u. Lebo musíme splniť jeden predpoklad a to, že aplikácia musí byť klikacia. Inak by plugin pre túto aplikáciu moc zmysel nedával. S týmto predpokladom na pamäti sme sa rozhodli vytvoriť aplikáciu v podobe hry.

Princíp hry je nasledovný. Hra pozostáva z viacerých úrovní, začína sa na úrovni 1 s tým, že v menu si bude môcť hráč zvoliť štartovaciu úroveň (viď Obrázok 3.21). Každá úroveň pozostáva z  $N$  objektov. Úroveň  $M$  obsahuje teda  $M$  objektov, kde  $M$  je nenulové prirodzené číslo. Pri každej úrovni sa na obrazovke náhodne umiestni týchto  $M$  objektov. Každý objekt má svoje číslo. Ak máme  $N$  objektov, tak na obrazovke budeme vidieť objekty s číslami 1, 2, 3, ...,  $N$ . Na začiatku úrovne každý objekt ukáže svoje číslo (viď Obrázok 3.18a) a potom po uplynutí časového intervalu ho prestane zobrazovať (viď Obrázok 3.18b). Tento časový interval počas ktorého objekty ukazujú svoje čísla sa bude so stúpajúcou sa úrovňou mierne zväčšovať. Po tom ako všetky objekty skryjú svoje čísla sa začína hrať. Úlohou hráča bude zvoliť, resp. kliknúť na objekty presne v poradí od 1 po  $N$ . Ak sa to hráčovi podarí (viď Obrázok 3.19b) postupuje do ďalšej úrovne kde je o  $+1$  objekt navyše. Ak sa mu to nepodarí (viď Obrázok 3.19a) úroveň sa opakuje s tým, že pozície objektov zostávajú zachované (viď Obrázok 3.18a). Objekty opäť ukážu svoje číslo a obdobne po uplynutí časového intervalu ho skryjú.



Obr. 3.18: Návrh mechaniky hry - ukáž / skry



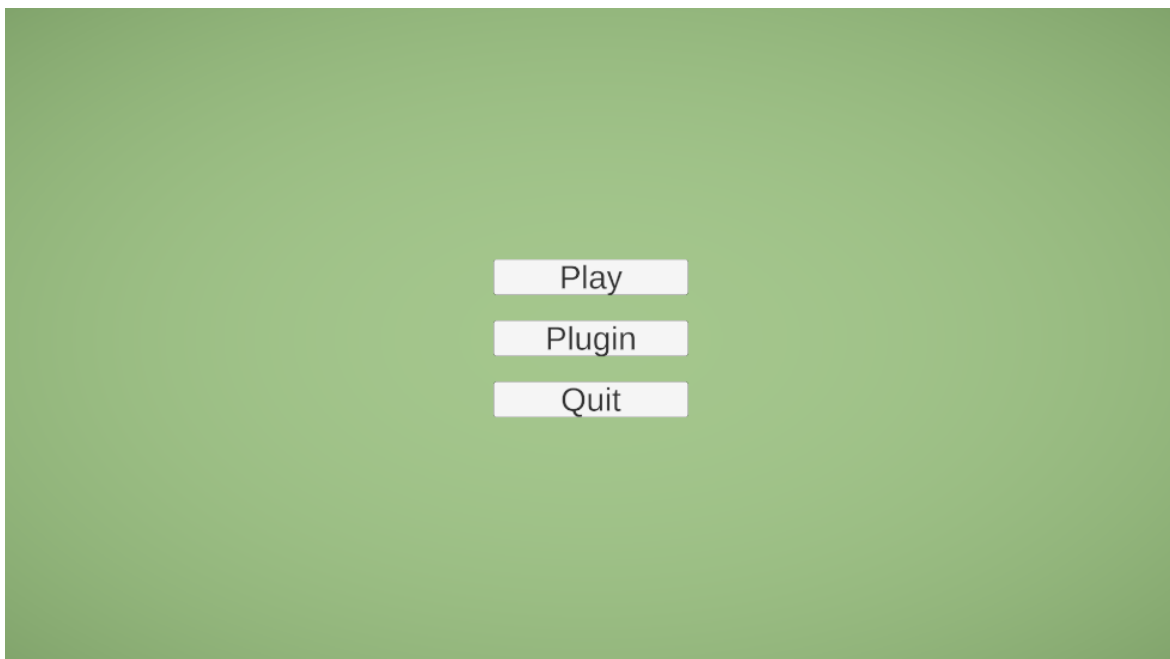
Obr. 3.19: Návrh mechaniky hry - dobrý / zlý

Takúto hru sme si vymysleli, pretože je interaktívna a vyzerá na prijateľného kandidáta pre integráciu s plugin-om. Navyše má aj priaznivé účinky, ako povzbudzovanie hráča k mozgovej aktivite.

Nasledujúce dva návrhy sú vytvorené priamo v Unity Editor-e.

### Návrh hlavného menu

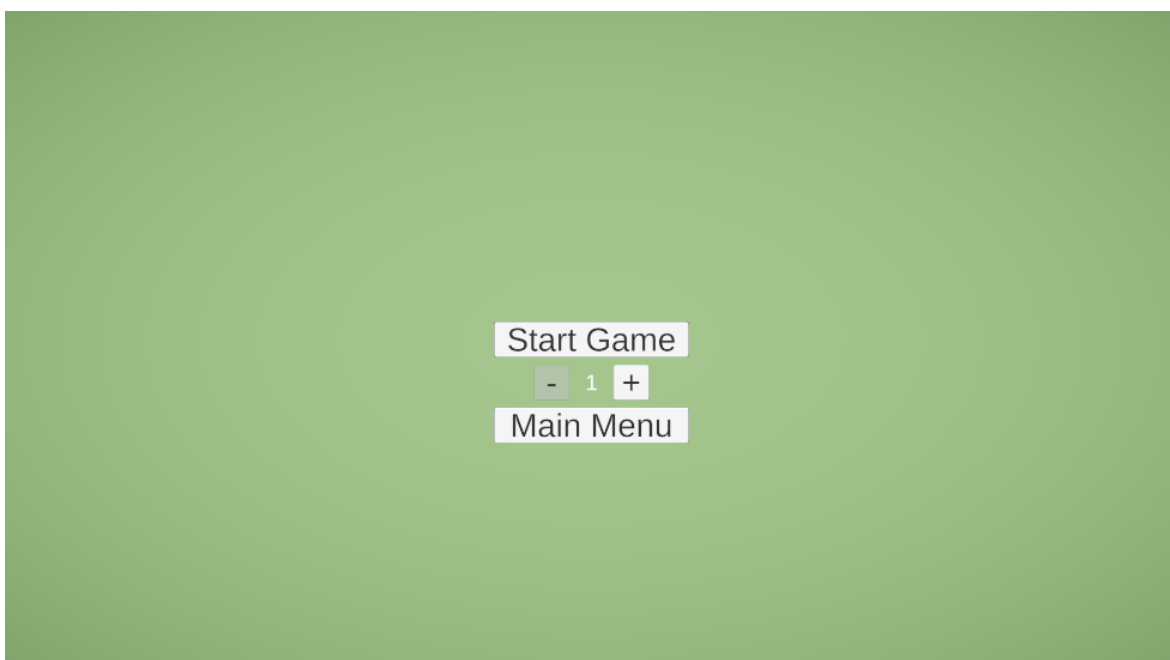
Jednoduché hlavné menu aplikácie (viď Obrázok 3.20) v ktorom sa nachádzajú tri tlačidlá. Tlačidlo *Play* pre začatie hry, respektíve presun do menu výberu úrovne. Tlačidlo *Quit* pre ukončenie aplikácie. Nakoniec sme pridali aj tlačilo *Plugin*, ktoré spustí menu plugin-u, keďže chceme integrovať náš plugin do tejto aplikácie, tak aby sme ho mali ako spustiť.



Obr. 3.20: Návrh hlavného menu aplikácie

### Návrh menu výberu úrovne

Menu výberu úrovne (viď Obrázok 3.21) obsahuje dve tlačidlá. Tlačidlo *Start Game* pre začatie hry na úrovni, ktorú si používateľ zvolil. Tlačidlo *Main Menu* pre návrat do hlavného menu. A tlačidlá *+* a *-* pre zväčšenie alebo zmenšenie hodnoty aktuálnej úrovne, ktorá je zobrazená medzi týmito dvoma tlačidlami.



Obr. 3.21: Návrh menu výberu úrovne

# Kapitola 4

## Implementácia systému a výsledky

V tejto kapitole čitateľovi popíšeme implementácie jednotlivých častí pluginu a aplikácie. Viac sa zameriame na dôležité časti implementácie, menej dôležité časti čitateľovi stručne popíšeme. Uvedieme postupy, ktoré sme skúsili, ale rozhodli sa ich nepoužiť. Oboznámime čitateľa s výhodami i nevýhodami daných riešení. Relevantné riešenia porovnáme z hľadiska presnosti alebo výkonu. Dôležité časti doplníme aj ukážkami kódu alebo pseudo kódom.

Pri implementácii sme postupovali podľa nášho návrhu spracovanom v kapitole 3. Postupovali sme iteratívne a vytvárali si prototypy. Z týchto prototypov vznikali ďalšie rafinovanejšie verzie. Počas celej implementácie sme používali VCS<sup>1</sup>, vďaka čomu sa nám jednoducho spravovali pracovné verzie, ľahko prepínalo medzi rôznymi verziami implementácie a v prípade chýb alebo problémov sme ľahko zistili kde nastala chyba.

Kompletný zdrojový kód je pre čitateľa k dispozícii na GitHub-e, odkaz sa nachádza v Dodatku A.

### 4.1 Upevnenie polarizačných filtrov

#### 4.1.1 Prototyp

Prvá verzia, s ktorou sme už pracovali pri vývoji softvérovej časti bol tento prototyp (viď Obrázok 4.1). Prototyp bol vyrobený z tenkej papierovej krabice od čaju. Prototyp obsahoval dva do seba vnorené kruhy s otvorom, ktoré obsahovali polarizačný filter. Vonkajší kruh bol statický. Vnútorň kruh sa otáčal vnútri vonkajšieho kruhu pre prispôsobenie prepusteného svetla. Vo vnútri sa nachádzala kamera, ktorá bola upevnená dvomi jedno centimetrovými podpornými papierikmi.

Výhodou bola rýchla konštrukcia a jednoduchosť. Nevýhodou bola krehkosť a často upevnená kamera vo vnútri povolila a spadla. Pre začiatok nám to stačilo no časom sa

---

<sup>1</sup>version control system - napr. git

požívanie prototypu stalo ošemetným a aj materiál bol opotrebovanejší.



Obr. 4.1: Prototyp upevnenia polarizačných filtrov na kameru

#### 4.1.2 Produkt

Časom sa prototyp stával menej a menej použiteľnejší, tak sme sa rozhodli spraviť robustnejšiu verziu. Dizajn sa moc nezmenil a väčšina prvkov zostala podobných (viď Obrázok 4.2). Kamera je zapustená do dreva s otvorom na kábel. Otočná funkcionálnosť polarizačných filtrov bola zachovaná a kamera zostala upevnená na jednom mieste počas celej doby používania. Tento produkt už pokrýval nevýhody prototypu a veľmi ľahko sa s ním pracovalo.

Jediná nevýhoda by bola, že sme nemysleli na teplo a kamera by sa mohla prehriať z nedostatku cirkulácie vzduchu, ale zatiaľ sa nám to nestalo.



(a) pohľad A



(b) pohľad B

Obr. 4.2: Produkt upevnenia polarizačných filtrov na kameru

## 4.2 Plugin

### 4.2.1 Implementácia stavového automatu

Pomocou návrhového vzoru **State pattern** sme implementovali stavový automat pre jadro pluginu. Vytvorili sme si abstraktnú triedu *LPIPBaseState* (Listing 4.1), z ktorej sme potom derivovali konkrétne stavy *InitializationState*, *ManualCalibrationState*, *RunningState* a *StandbyState*. Jediná metóda s parametrom v hlavičke, *EnterState(...)*, obsahuje parameter, ktorý slúži na odovzdanie referencie na manager-a, ktorú si uloží do svojej lokálnej premennej.

```

1 public abstract class LPIPBaseState
2 {
3     public abstract void EnterState(LIPICoreManager lpipCoreManager);
4     public abstract void UpdateState();
5     public abstract void ExitState();
6 }

```

Listing 4.1: Abstraktná trieda stavu

Konkrétne stavy riešili už konkrétne úlohy, z toho vychádzajú aj ich názvy.

Stav *InitializationState* rieši inicializáciu premenných pluginu, t. j. zistí si údaje o rozlíšení kamery a rozlíšení obrazovky v ktorej je spustená aplikácia. Tieto údaje si uloží do štruktúr *WindowData* a *CameraData* (Listing 4.2). Navyše vypočíta a uloží aj pomer rozlíšenia v ktorom je spustená aplikácia ku rozlíšeniu, ktoré dostaneme z kamery, ktorý bude potrebný neskôr pri kalibrácii.

```

1 public struct CameraData
2 {
3     public int CAMERA_WIDTH;
4     public int CAMERA_HEIGHT;
5 }
6 public struct WindowData
7 {
8     public int GAME_WINDOW_WIDTH;
9     public int GAME_WINDOW_HEIGHT;
10    public float GAME_WINDOW_FACTORX;
11    public float GAME_WINDOW_FACTORY;
12 }

```

Listing 4.2: Štruktúry pre údaje inicializácie

Stav *ManualCalibrationState* rieši manuálnu kalibráciu. Pomocou ľavého tlačidla myši sa zvolia na obrazovke štyri body v poradí ľavý dolný, pravý dolný, pravý horný a ľavý horný. Tu sa zoberie do úvahy škálovanie uložené v štruktúre *WindowData*, pretože rozlíšenie aplikácie a rozlíšenie kamery nemusí byť totožné, teda je potrebné súradnice

vynásobiť škálovaním. Pri spustení kalibrácie automat vyvolá udalosť začatia kalibrácie a pri ukončení vyvolá udalosť skončenia kalibrácie. Pri ukončení kalibrácie sa uložia údaje o zvolených bodoch do štruktúry *LPIPCalibrationData* (Listing 4.3).

```

1 public struct LPIPCalibrationData
2 {
3     public Vector2[] real;
4     public Vector2[] ideal;
5 }

```

Listing 4.3: Štruktúra pre údaje kalibrácie

*Ideal* reprezentuje pole bodov (so súradnicami x,y) okna aplikácie a *real* reprezentuje pole bodov, ktoré boli zvolené pri kalibrácii.

Stav *RunningState* vykonáva hlavnú prácu, detekciu laserového lúča v obraze za pomoci poskytnutých údajov z *InitializationState* a *ManualCalibrationState*. Počas behu stavu sa vyvoláva udalosť, že sa detegoval laserový lúč a má sa simulovať klik na jeho súradniciach. Podrobnejší popis je rozpísaný v sekciiach 4.2.2 a 4.2.3.

Stav *StandbyState* nič nerobí, v tomto stave sa čaká.

Ďalej sme vytvorili triedu *LIPICoreManager* (Listing 4.4), ktorej úlohou je ovládať náš automat, respektíve prepínať stavy pomocou metódy *SwitchState(...)* a zabezpečiť ich vykonávanie metódou *\_currentState.UpdateState()* v *Update()*, ktorý je vykonávaný každý frame. V premennej *\_currentState* si udržiavame aktuálny stav.

```

1 public class LIPICoreManager : MonoBehaviour{
2     public LIPIBaseState InitializationState { get; private set; }
3     public LIPIBaseState ManualCalibrationState {get; private set;}
4     public LIPIBaseState RunningState { get; private set; }
5     public LIPIBaseState StandbyState { get; private set; }
6
7     private LIPIBaseState _currentState;
8
9     private void Awake(){
10        InitializationState = new LPIPInitializationState();
11        ManualCalibrationState = new LPIPManualCalibrationState();
12        RunningState = new LPIPRunningState();
13        StandbyState = new LPIPStandbyState();
14    }
15    private void Start(){
16        SwitchState(StandbyState);
17        _currentState.EnterState(this);
18    }
19    private void Update(){
20        _currentState.UpdateState();
21    }
22    ...

```

```

23     public void SwitchState(LPIPBaseState state){
24         if (TransitionToStateIsAllowed(state)){
25             _currentState?.ExitState();
26             _currentState = state;
27             _currentState?.EnterState(this);
28         }
29         else{
30             printErrorMessage();
31         }
32     }
33 }

```

Listing 4.4: Stručný pseudokód manager-a stavového automatu

V triede *LPIPCoreManager* sme pridali ešte pomocné metódy na spustenie (*StartLPIP()*), reštartovanie na začiatok (*ResetLPIP()*) a kalibráciu (*ReCalibrateLPIP()*). Implementovali sme aj triedu *LPIPCoreController* (Listing 4.5) pomocou návrhového vzoru **Singleton**, ktorá slúži pre programátora alebo iný systém na ovládanie pluginu a nastavenie konfiguračných premenných cez metódy *SetWebCamTexture(...)* a *SetProjectorDisplayId(...)*.

```

1 public class LPIPCoreController : MonoBehaviour
2 {
3     [SerializeField] private LPIPCoreManager _lpipCoreManager;
4
5     public static LPIPCoreController Instance;
6     private bool isWebCamTextureConfigured = false;
7     private bool isProjectorIdConfigured = false;
8
9     private void Awake()
10    {
11        if (Instance == null){
12            Instance = this;
13        }
14    }
15    public void InitializeLPIP(){
16        if (isWebCamTextureConfigured && isProjectorIdConfigured){
17            _lpipCoreManager.StartLPIP();
18        }
19        else{
20            if (!(isWebCamTextureConfigured && isProjectorIdConfigured)){
21                Debug.LogError("WebCamTexture and ProjectorId is not
22                configured!");
23            }
24        }
25    }
26    public void ResetLPIP(){

```



```

26     _lpipCoreManager.ResetLPIP();
27 }
28 public void ReCalibrateLPIP(){
29     _lpipCoreManager.ReCalibrateLPIP();
30 }
31 public void SetWebCamTexture(WebCamTexture webCamTexture){
32     if (webCamTexture == null){
33         Debug.LogError("webCamTexture == NULL");
34         return;
35     }
36     _lpipCoreManager.WebCamTexture = webCamTexture;
37     isWebCamTextureConfigured = true;
38 }
39 }
40 public void SetProjectorDisplayId(int projectorDisplayId)
41 {
42     if (projectorDisplayId < 0){
43         Debug.LogError("projectorDisplayId must be non negative!");
44         return;
45     }
46     _lpipCoreManager.PROJECTOR_DISPLAY_ID = projectorDisplayId;
47     isProjectorIdConfigured = true;
48 }
49 }

```

Listing 4.5: Kód controller-a pre stavový automat

## 4.2.2 Implementácia detekcie laserového lúča v obraze z kamery

Detekciu laserového lúča v obraze sme implementovali tak, ako sme popísali v kapitole 3, sekcii 3.2.3. Vytvorili sme si štruktúru reprezentujúcu hraničný bod nazvaný *BorderPoint* (Listing 4.6).

```

1 public struct BorderPoint
2 {
3     public double luminance;
4     public int value;
5 }

```

Listing 4.6: Štruktúra BorderPoint

Náš obdĺžnik, ktorý ohraničoval detegovaný laserový lúč je reprezentovaný štyrmi hraničnými hodnotami. Horným - *top*, dolným - *bottom*, ľavým - *left* a pravým - *right* ohraničením (Listing 4.7).

```

1     private BorderPoint top;
2     private BorderPoint bottom;

```

```

3     private BorderPoint left ;
4     private BorderPoint right ;

```

Listing 4.7: Reprezentácia obdĺžnika ohraničujúceho laserový lúč

Samotná detekcia laserového lúča v obraze pozostávala z jednoduchej rutiny. Z premennej *webCamTexture*, ktorá referencovala kameru sme si získali pole aktuálnych pixlov z kamery volaním vstavanej Unity metódy (*webCamTexture.GetPixels32()*), ktoré sme si uložili do premennej *webCamPixels* typu *Color32*. Potom sme cez pole *webCamPixels* iterovali.

### Prvé riešenie - svietivosť pixlu

V rámci iterácie pixlov sme pre každý pixel vypočítali jeho svietivosť pomocou vzorca [15] a aj jeho 2D súradnice, keďže sa nachádzal v 1D poli a aktualizovali hraničné hodnoty nášho ohraničujúceho obdĺžnika (Listing 4.8).

```

1 for (int i = 0; i < webCamPixels.Length; i++)
2 {
3     var pixelLuminance = R_VALUE * webCamPixels[i].r + G_VALUE *
4     webCamPixels[i].g + B_VALUE * webCamPixels[i].b;
5     int currentX = i % CAMERA_WIDTH;
6     int currentY = i / CAMERA_WIDTH;
7
8     if (pixelLuminance > MIN_PIXEL_LUMINANCE)
9     {
10        updateDetectionBorders();
11    }
12 }

```

Listing 4.8: Pseudokód iterácie - svietivosť pixlu

Toto riešenie malo isté nedostatky. Keď do záberu kamery umiestníme objekt napr. sklenený pohár, tak z pohára sa môže odrážať svetlo a môže to prispieť do algoritmu detekcie, čo nám pokazí detekciu. Toto sa dá jednoducho vyriešiť tak, že prinútime algoritmus iterovať len cez také pixle, ktoré sú v oblasti, ktorá sa vyznačila pri kalibrácii. Avšak, keby bol pohár umiestnený v rámci tejto oblasti, už by sme to neobišli.

### Druhé riešenie - farba pixlu

Tentoraz sme sa namiesto svietivosti zamerali na farby pixlov. Zisťovali sme, či sa farba pixlu nachádza v intervale, ktorý sme si pevne určili. Interval pre červenú, zelenú a modrú sme určili po preskúmaní pár snímok, ktoré obsahovali laserový lúč s aplikovanými polarizačnými filtermi na kamere. Podobne ako v prvom riešení, sme si tiež vyrátali 2D súradnice a aktualizovali hraničné hodnoty (Listing 4.9).

```

1 for (int i = 0; i < webCamPixels.Length; i++)
2 {
3     int currentX = i % CAMERA_WIDTH;
4     int currentY = i / CAMERA_WIDTH;
5
6     if(webCamPixels[i].r > 120 && webCamPixels[i].g < 189 &&
webCamPixels[i].b < 170)
7     {
8         updateDetectionBorders();
9     }
10
11 }

```

Listing 4.9: Pseudokód iterácie - farba pixlu

Tiež sa druhé riešenie nezaobíde bez drobných problémov. Síce ľudia vnímajú tmavý obraz ako čiernu, respektíve odtieň tmavej farby, kamery farby vnímajú inak a navyše sa k tomu pridáva šum. Pri druhom riešení sa stávalo, že keď kamera snímala tmú, pixle neboli len čierne, resp. tmavé, ale kamera si obraz “zkrášľovala” a teda farby pixlov poskakovali, čím sa často stávalo, že algoritmus detegoval pixel s farbou v rámci intervalu, i keď nám sa to nemuselo zdať. Ľudia si tento detail bežne nevšimnú, keďže nevidia každý pixel, ľudom sa to zleje do jednej tmavej farby. Tento problém by sa dal riešiť tak, že obraz by sme pred-spracovali a vyhli sa tak tomuto efektu. Alebo sa môžeme pokúsiť, tak ako v prvom riešení, obmedziť pixle cez ktoré sa iteruje.

### Tretie riešenie - kombinácia prvého a druhého riešenia

Vyskúšali sme použiť aj kombináciu prvého a druhého riešenia, čím sa nám podarilo tieto problémy znížiť aspoň tak, aby to neovplyvňovalo bežný chod. Výsledný pseudokód detekcie laserového lúča v obraze je uvedený v Listing 4.10.

```

1 private void DetectLaserInFrame() {
2     laserDetected = false;
3     //Array of pixels
4     webCamPixels = webCamTexture.GetPixels32();
5
6     for (int i = 0; i < webCamPixels.Length; i++)
7     {
8         var pixelLuminance = R_VALUE * webCamPixels[i].r + G_VALUE *
webCamPixels[i].g + B_VALUE * webCamPixels[i].b;
9         int currentX = i % CAMERA_WIDTH;
10        int currentY = i / CAMERA_WIDTH;
11        if (pixelLuminance > MIN_PIXEL_LUMINANCE)
12        {

```

```

13         if(webCamPixels[i].r > 120 && webCamPixels[i].g < 189 &&
webCamPixels[i].b < 170)
14             {
15                 // update laser encapsulating rectangle borders
16                 updateDetectionBorders();
17                 laserDetected = true;
18             }
19         }
20     }
21
22     if(laserDetected){
23         //invoke left mouse button click simulation with detected
coordinates
24         SimulateLeftClick();
25     }
26 }

```

Listing 4.10: Pseudokód metódy na detekciu laserového lúča v obraze z kamery

Všetky tieto tri riešenia zdieľali spoločný problém. Všetky tieto riešenia bežali výhradne na procesore, čo negatívne ovplyvňovalo výkon. Preto sme vyskúšali implementovať aj štvrté riešenie.

### Štvrté riešenie - compute shader

V štvrtom riešení sme presunuli časť, ktorá zisťovala či sa v obrázku nachádza laserový lúč do compute shader-u, aby sme to mohli vykonať na GPU. Zmenili sme dátovú štruktúru reprezentujúcu obdĺžnik ohraničujúci laserový lúč (Listing 4.11), tentoraz nám táto štruktúra stačila na reprezentáciu celého obdĺžnik ohraničujúceho laserový lúč.

```

1 public struct Bound {
2     public int minX;
3     public int minY;
4     public int maxX;
5     public int maxY;
6     public int detected;
7 };

```

Listing 4.11: Zmena štruktúry obdĺžnika ohraničujúceho laserový lúč

Teraz výpočet na GPU prebiehal tak, že pre každý pixel sa zistilo, či je v rámci povolených hodnôt R, G a B. Svietivosť pixlu sme nevyužili, ani nepoužili. Potom sme skontrolovali, či sme nevyskočili za hranice textúry, keďže čítanie mimo rozsah pri textúrach je povolené, ale vracia východziu hodnotu, čo by nám skreslilo výsledok a aktualizovali hranice obdĺžnika pomocou metód *InterlockedMin(...)* a *InterlockedMax(...)*.

Potom už stačilo prevziať si výsledok výpočtu na CPU a zavolať metódu na simuláciu kliknutia, ak sa laserový lúč podarilo detegovať.

### Porovnanie výkonov

Keďže v prvých troch riešeniach sa všetko robilo výhradne cez CPU, celkový výkon bol výrazne ovplyvnený a tým pádom prázdna aplikácia, čisto s plugin-om bežala priemerne na 45 fps s frametime-om okolo 24 ms. Keď sme spravili štvrté riešenie, s pomocou GPU, tak výkon prázdnej aplikácie čisto s plugin-om bol priemerne 100 fps s frametime-om 9 ms. Na Obrázkoch 4.3 až 4.6 sú zobrazené grafy a štatistiky výkonu tretieho a štvrtého riešenia pri konfiguráciách notebooku (A) Ryzen 7 5800H, iGPU AMD Radeon Graphics, (on battery) a (B) Ryzen 7 5800H, dGPU Nvidia RTX 3060 Mobile, 130W (on power). Zaznamenaný nárast výkonu štvrtého riešenia oproti tretiemu je priemerne 33% pri konfigurácií (A) a 150% pri konfigurácií (B). Výkon v Unity editore sa môže líšiť od výkonu skompilovanej verzie. Spomínané grafy a štatistiky sú z Unity editor-a, beh aplikácie bol takisto uskutočnený v Unity editor-e.

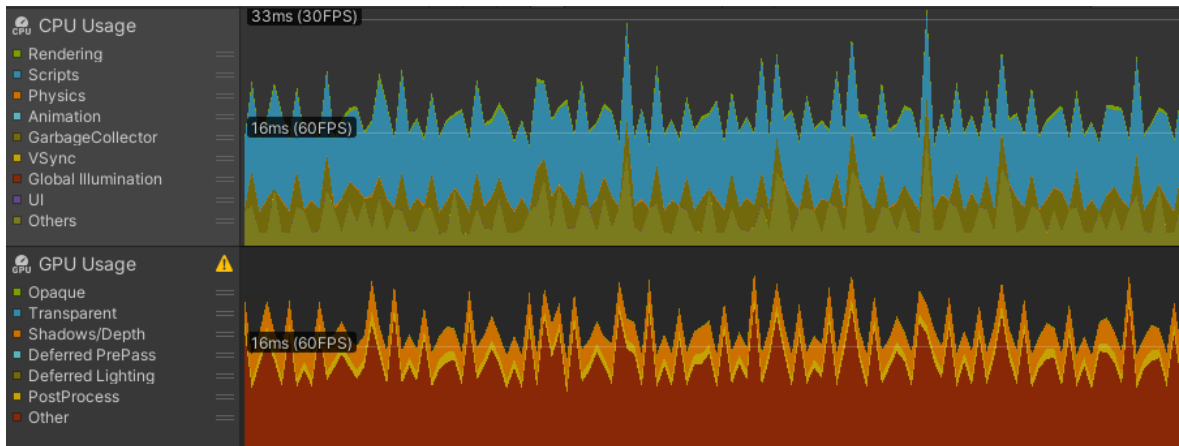
| Statistics                            |                      |
|---------------------------------------|----------------------|
| <b>Audio:</b>                         |                      |
| Level: -74.8 dB                       | DSP load: 0.2%       |
| Clipping: 0.0%                        | Stream load: 0.0%    |
| <b>Graphics:</b> 48.4 FPS (20.7ms)    |                      |
| CPU: main 20.7ms render thread 16.5ms |                      |
| Batches: 19                           | Saved by batching: 0 |
| Tris: 2.1k                            | Verts: 5.9k          |
| Screen: 1920x1080 - 23.7 MB           |                      |
| SetPass calls: 12                     | Shadow casters: 4    |
| Visible skinned meshes: 0             |                      |
| Animation components playing: 0       |                      |
| Animator components playing: 0        |                      |

(a) Konfigurácia (A)

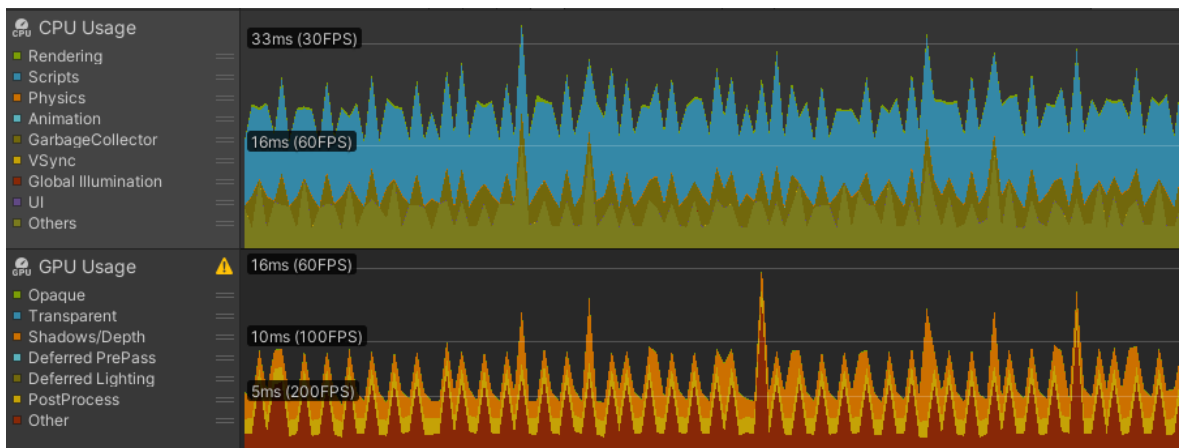
| Statistics                            |                      |
|---------------------------------------|----------------------|
| <b>Audio:</b>                         |                      |
| Level: -74.8 dB                       | DSP load: 0.2%       |
| Clipping: 0.0%                        | Stream load: 0.0%    |
| <b>Graphics:</b> 40.1 FPS (25.0ms)    |                      |
| CPU: main 25.0ms render thread 20.9ms |                      |
| Batches: 27                           | Saved by batching: 0 |
| Tris: 2.2k                            | Verts: 6.1k          |
| Screen: 1920x1080 - 23.7 MB           |                      |
| SetPass calls: 11                     | Shadow casters: 10   |
| Visible skinned meshes: 0             |                      |
| Animation components playing: 0       |                      |
| Animator components playing: 0        |                      |

(b) Konfigurácia (B)

Obr. 4.3: Štatistiky zobrazujúce výkon tretieho riešenia 4.2.2

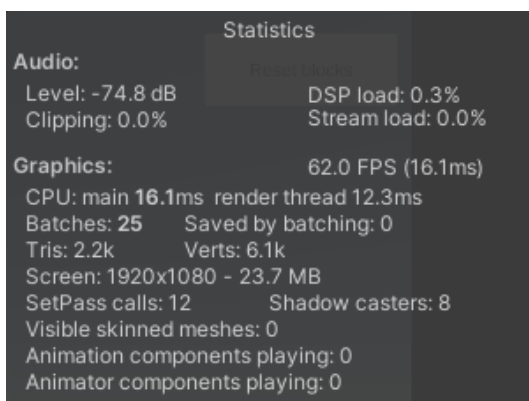


(a) Konfigurácia (A)

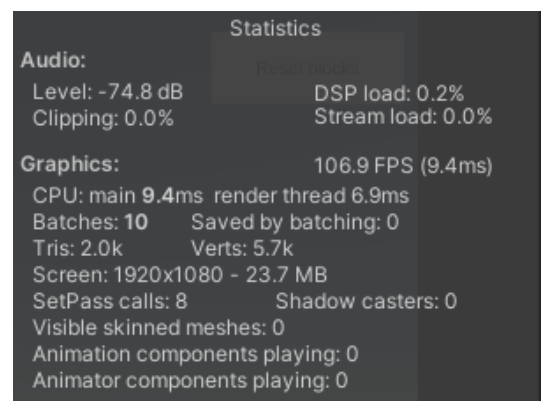


(b) Konfigurácia (B)

Obr. 4.4: Grafy zobrazujúce výkon tretieho riešenia 4.2.2

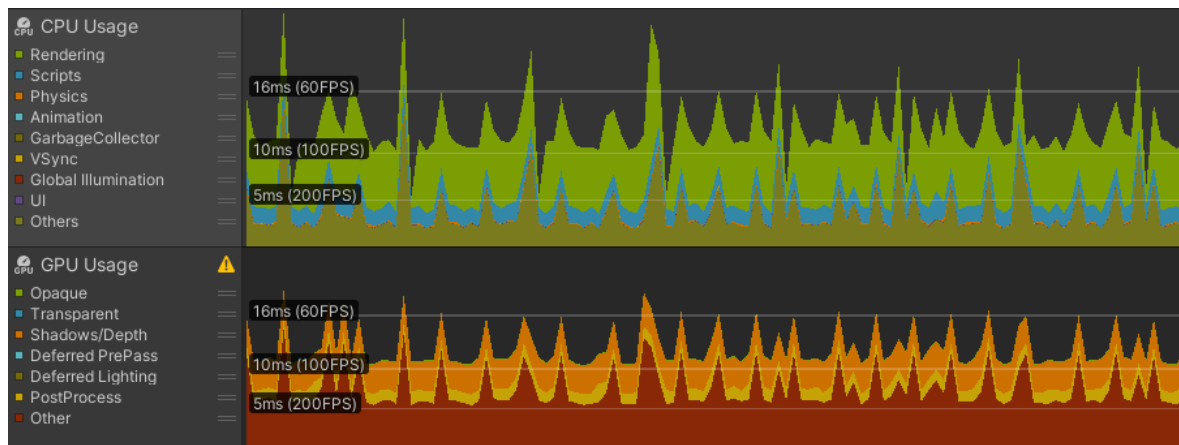


(a) Konfigurácia (A)

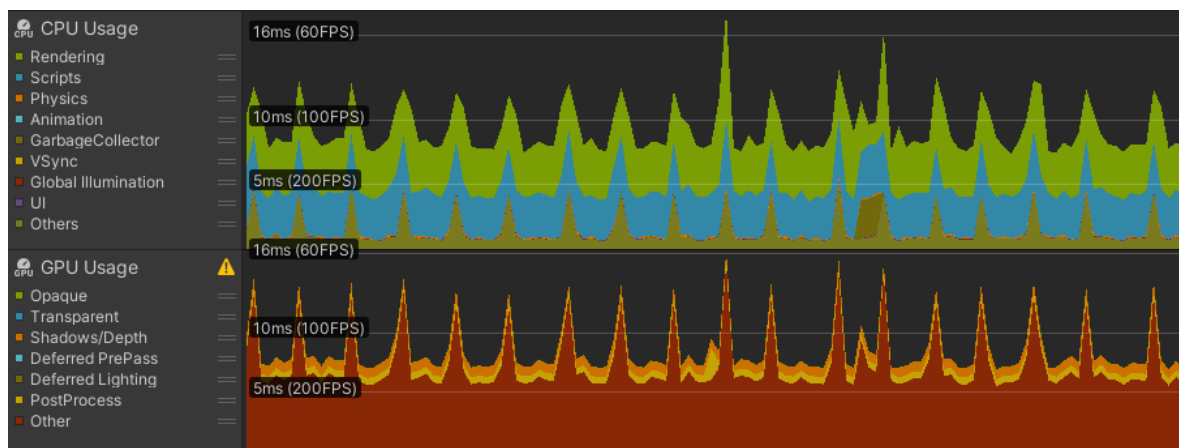


(b) Konfigurácia (B)

Obr. 4.5: Štatistiky zobrazujúce výkon štvrtého riešenia 4.2.2



(a) Konfigurácia (A)



(b) Konfigurácia (B)

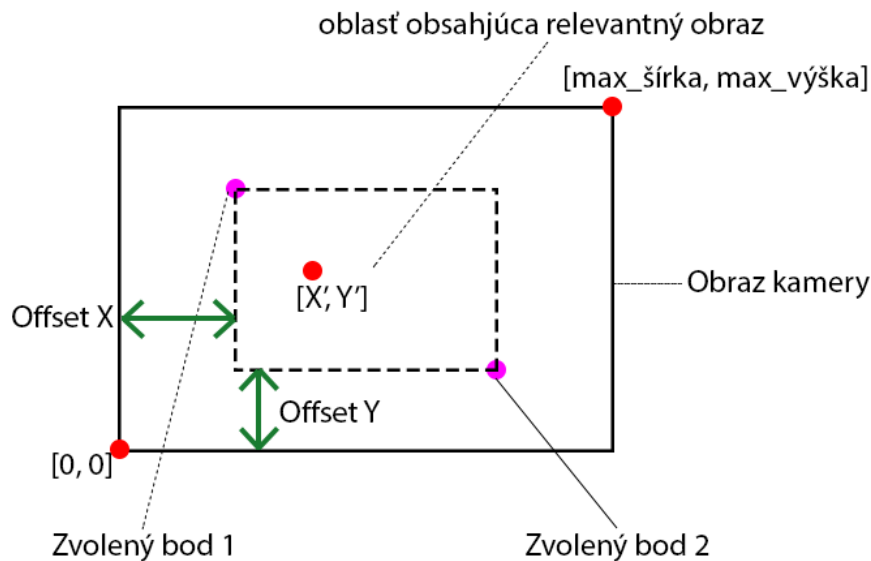
Obr. 4.6: Grafy zobrazujúce výkon štvrtého riešenia 4.2.2

### 4.2.3 Implementácia kalibrácie a transformácia súradníc

Podobne, aj pri riešení tejto časti implementácie sme skúsili rôzne prístupy.

#### Prvé riešenie - kalibrácia cez 2 body

Ako prvé riešenie, pomerne naivné, bola kalibrácia pomocou dvoch bodov. Na obrazovke sa zobrazoval obraz z kamery a pomocou ľavého tlačidla myši sa zvolili dva body, v poradí ľavý horný a pravý dolný. Pomocou týchto dvoch bodov sme si vytvorili obdĺžnikovú oblasť, ktorá reprezentovala obrazovku aplikácie v obraze z kamery, ako naznačuje Obrázok 4.7. Keďže rozlíšenie kamery nemusí byť totožné ako rozlíšenie aplikácie, ktorá zobrazuje tento obraz z kamery. Zvolené body sme vynásobili škálovaním, ako sme spomínali v Sekcii 4.2.1, odsek o *InitializationState*.



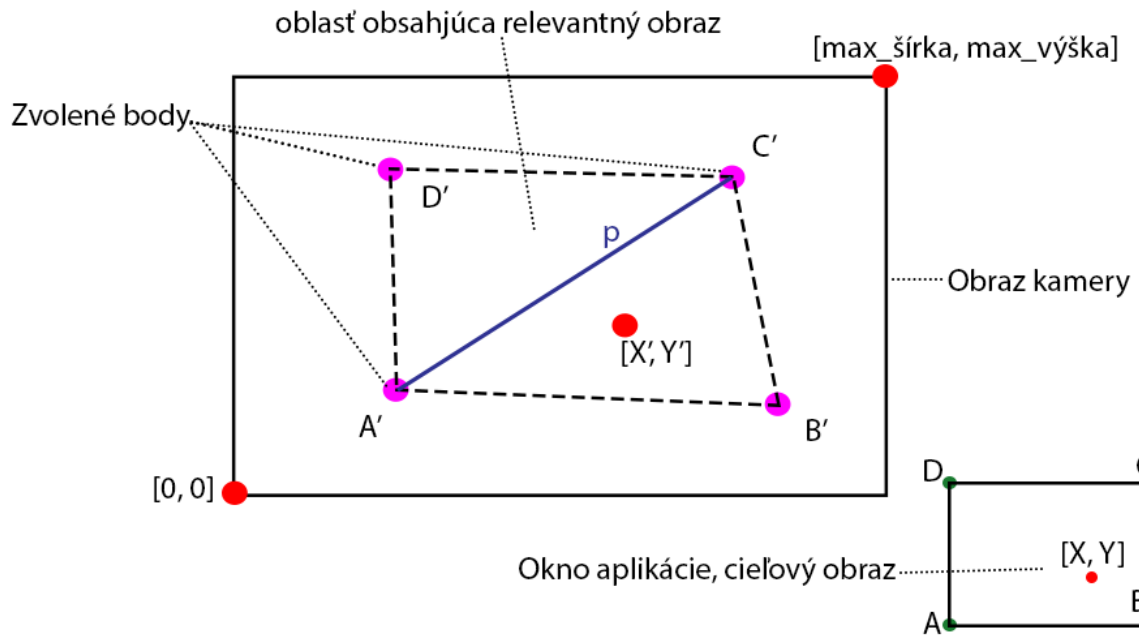
Obr. 4.7: Kalibrácia pomocou 2 bodov

Transformácia súradníc vyzerala potom nasledovne. Bod  $[X', Y']$ , stred kde sa detegoval laserový lúč v obraze z kamery. Oblasť obsahujúca obraz premietaný z projektora (relevantný obraz) bola posunutá z bodu  $[0, 0]$  o  $Offset X$  a  $Offset Y$ . Výsledné transformované súradnice bodu  $[X, Y]$  sú  $[X, Y] = [(X' + offsetX) * factorX, (Y' + offsetY) * factorY]$ , kde  $factorX$  bol pomer šírky obrazu z kamery ku šírke oblasti obsahujúcej relevantný obraz,  $factorY$  vyzeral obdobne.

### Druhé riešenie - kalibrácia cez 4 body a trojuholníky

Druhé riešenie, bola kalibrácia pomocou štyroch bodov. Podobne ako pri prvom riešení, na obrazovke sa zobrazoval obraz z kamery a pomocou ľavého tlačidla myši sa zvolili štyri body. Zvolené body sme takisto museli vynásobiť škálovaním, z podobného dôvodu. Tieto štyri body tvorili štvoruholník, oblasť v ktorej sa nachádzal obraz premietaný z projektora (relevantný obraz). Pre čitateľa máme k dispozícii Obrázok 4.8 k vyššie uvedenému opisu.





Obr. 4.8: Kalibrácia pomocou 4 bodov

Transformácia súradníc v druhom riešení prebiehala pomocou algoritmu [16], ktorý čitateľovi popíšeme s pomocou Obrázka 4.8. Štvoruholník  $ABCD$  predstavuje okno aplikácie, respektíve jej rozlíšenie. Štvoruholník  $A'B'C'D'$  predstavuje relevantný obraz, čiže okno aplikácie premietanej cez projektor. V prvom kroku si algoritmus vypočíta smernicový tvar priamky  $p$  z bodov  $A'$  a  $B'$ . V druhom kroku zistí, či sa bod  $[X', Y']$  nachádza v trojuholníku  $A'B'C'$  alebo  $A'C'D'$ . V poslednom, treťom kroku, vypočíta skutočnú polohu bodu  $[X, Y]$  k nemu prislúchajúcom trojuholníku  $ABC$  za pomoci trojuholníka  $A'B'C'$  kde sa nachádza bod  $[X', Y']$ .

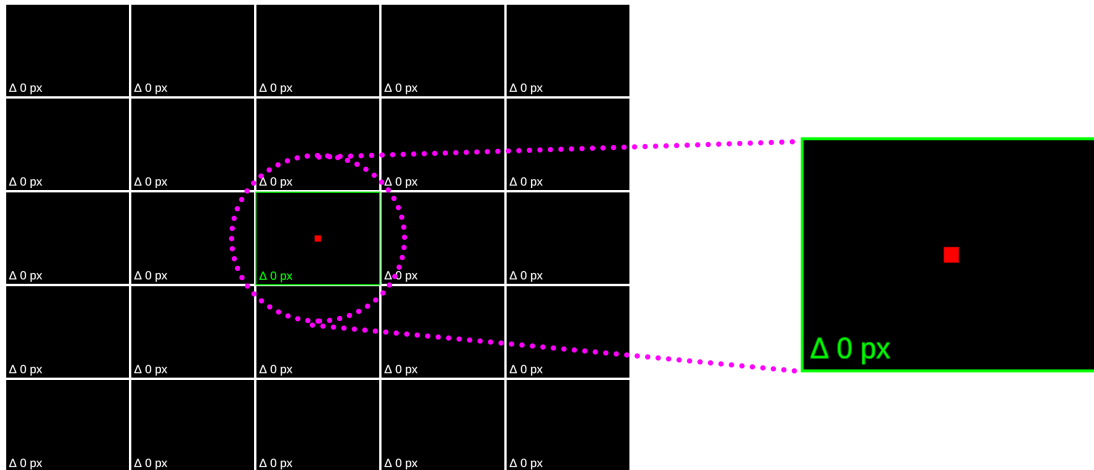
### Tretie riešenie - kalibrácia cez 4 body s inou transformáciou súradníc

V treťom riešení kalibrácia prebiehala úplne rovnako ako v druhom riešení. Jedine sme zmenili algoritmus transformácie súradníc. V treťom riešení sme použili algoritmus [17].

#### 4.2.4 Výsledky implementácie kalibrácie a transformácia súradníc

Zostrojili sme nástroj pomocou ktorého sme zistili a porovnali presnosť jednotlivých riešení za rôznych podmienok. Nástroj sme zostrojili tiež v prostredí Unity. Náš nástroj pozostával z mriežky (viď Obrázok 4.9) s rozmermi  $m \times n$  kde  $m, n \in \mathcal{N} - 0$ . Po kliknutí do niektorej bunky sa daná bunka zvýraznila zelenou farbou a v strede sa vykreslil malý červený štvorček. Každá bunka mala priradené svoje textové pole v ktorom bol vypísaný text vo formáte  $\Delta <číslo> px$ . Spôsob testovania presnosti bol, spustili sme náš nástroj

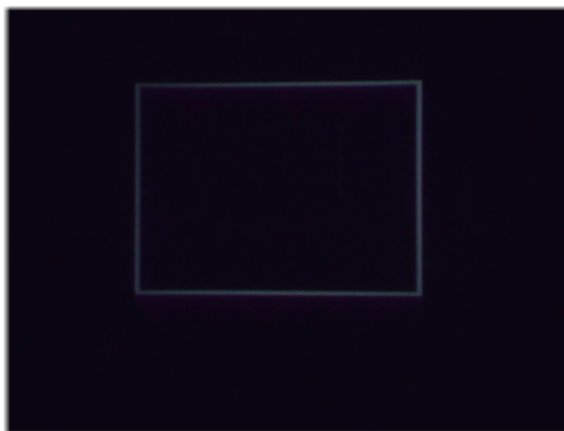
a premietali ho cez projektor na stenu (t.j. na stene sme videli mriežku). Postupne sme si zvolili po jednom každú bunku. Pri každej zvolenej bunke sme laserovým lúčom mierili priamo na stred červeného štvorčeka v danej bunke. Následne sme vypočítali rozdiel medzi transformovanými súradnicami a súradnicami stredu malého červeného štvorčeka zvolenej bunky a zapísali hodnotu do prislúchajúceho textového poľa. Na záver, keď sme prešli všetkými bunkami sme vypočítali výberový priemer  $\bar{y}$  a rozptyl výberového priemeru  $s^2$ . Merania sme vykonali pri štyroch scenároch jednotlivo s použitím riešení zo Sekcie 4.2.3.



Obr. 4.9: Nástroj na testovanie presnosti

### Prvý testovací scenár

V prvom testovacom scenári sme umiestnili kameru rovnobežne so stenou. Obraz, ktorý snímala kamera bol za rovno kamery (t. j. nebol pod žiadnou rotáciou). Obraz z pohľadu kamery je zobrazený na Obrázku 4.10.



Obr. 4.10: Obraz z kamery pri prvom testovacom scenári

Pri **prvom riešení** sme namerali výberový priemer  $\bar{y} = 9,4735$  a rozptyl výberového

priemeru  $s^2 = 32,5357$ . Výsledky prvého riešenia pri prvom testovacom scenári sú na Obrázku 4.11.

|                    |                    |                    |                    |                    |
|--------------------|--------------------|--------------------|--------------------|--------------------|
| $\Delta 8,85475f$  | $\Delta 4,003239f$ | $\Delta 11,1369f$  | $\Delta 5,299726f$ | $\Delta 19,55305f$ |
| $\Delta 3,842829f$ | $\Delta 3,531727f$ | $\Delta 6,863317f$ | $\Delta 14,88975f$ | $\Delta 25,7335f$  |
| $\Delta 5,972714f$ | $\Delta 7,58054f$  | $\Delta 10,34827f$ | $\Delta 5,572144f$ | $\Delta 17,36925f$ |
| $\Delta 5,261235f$ | $\Delta 12,31404f$ | $\Delta 5,638209f$ | $\Delta 12,89217f$ | $\Delta 6,517749f$ |
| $\Delta 4,797684f$ | $\Delta 17,01999f$ | $\Delta 9,870328f$ | $\Delta 4,797684f$ | $\Delta 7,176962f$ |

Obr. 4.11: Výsledky prvého riešenia pri prvom testovacom scenári

Presnosť pri **prvom riešení**, keď vezmeme do úvahy aj našu nepresnosť pri mierení na malý červený štvorček, je relatívne presná a skoro na každej bunke rovnomerná.

Pri **druhom riešení** sme namerali výberový priemer  $\bar{y} = 6,8938$  a rozptyl výberového priemeru  $s^2 = 16,6037$ . Výsledky druhého riešenia pri prvom testovacom scenári sú na Obrázku 4.12.

|                    |                    |                    |                    |                    |
|--------------------|--------------------|--------------------|--------------------|--------------------|
| $\Delta 16,02142f$ | $\Delta 6,679592f$ | $\Delta 6,988943f$ | $\Delta 3,650086f$ | $\Delta 9,179001f$ |
| $\Delta 17,64448f$ | $\Delta 2,647795f$ | $\Delta 4,245498f$ | $\Delta 8,820408f$ | $\Delta 4,059427f$ |
| $\Delta 9,725375f$ | $\Delta 3,504637f$ | $\Delta 4,588684f$ | $\Delta 13,97916f$ | $\Delta 2,75315f$  |
| $\Delta 7,226511f$ | $\Delta 6,848571f$ | $\Delta 9,275761f$ | $\Delta 3,617855f$ | $\Delta 3,697561f$ |
| $\Delta 4,044829f$ | $\Delta 3,489392f$ | $\Delta 7,498994f$ | $\Delta 7,911223f$ | $\Delta 4,247154f$ |

Obr. 4.12: Výsledky druhého riešenia pri prvom testovacom scenári

Presnosť **druhého riešenia** je podobne dosť presná, na každej bunke rovnomerná. Dokonca druhé riešenie je presnejšie ako prvé riešenie v tomto testovacom scenári.

Pri **treťom riešení** s parametrom  $d = 0.9$  sme namerali výberový priemer  $\bar{y} = 49,3162$  a rozptyl výberového priemeru  $s^2 = 967,2986$ . Výsledky tretieho riešenia s parametrom  $d = 0.9$  pri prvom testovacom scenári sú na Obrázku 4.13.

|                    |                    |                    |                    |                    |
|--------------------|--------------------|--------------------|--------------------|--------------------|
| $\Delta 97,76545f$ | $\Delta 64,66506f$ | $\Delta 61,16635f$ | $\Delta 59,49643f$ | $\Delta 97,9425f$  |
| $\Delta 75,35163f$ | $\Delta 12,3043f$  | $\Delta 7,360315f$ | $\Delta 11,92705f$ | $\Delta 72,01414f$ |
| $\Delta 74,64428f$ | $\Delta 15,13234f$ | $\Delta 6,737122f$ | $\Delta 13,22552f$ | $\Delta 73,23888f$ |
| $\Delta 74,10483f$ | $\Delta 13,41858f$ | $\Delta 7,978027f$ | $\Delta 18,67737f$ | $\Delta 68,65089f$ |
| $\Delta 84,34508f$ | $\Delta 52,56167f$ | $\Delta 47,07148f$ | $\Delta 48,57139f$ | $\Delta 74,55441f$ |

Obr. 4.13: Výsledky tretieho riešenia s parametrom  $d = 0,9$  pri prvom testovacom scenári

Ako si čitateľ iste všimol pri pohľade na Obrázok 4.13. Presnosť pri **treťom riešení** sa so vzdáľovaním od stredu zhoršuje. Pri zmene parametra  $d = 0.5$  sme namerali výberový priemer  $\bar{y} = 167,8270$  a rozptyl výberového priemeru  $s^2 = 4720,4770$ . Tretie riešenie s parametrom  $d = 0.5$  pri prvom testovacom scenári je znázornené na Obrázku 4.14.

|                    |                    |                    |                    |                    |
|--------------------|--------------------|--------------------|--------------------|--------------------|
| $\Delta 114,432f$  | $\Delta 209,133f$  | $\Delta 69,86336f$ | $\Delta 220,9293f$ | $\Delta 124,5732f$ |
| $\Delta 163,6088f$ | $\Delta 233,8664f$ | $\Delta 148,3757f$ | $\Delta 269,004f$  | $\Delta 160,1039f$ |
| $\Delta 95,36407f$ | $\Delta 218,376f$  | $\Delta 7,256303f$ | $\Delta 218,0278f$ | $\Delta 98,83958f$ |
| $\Delta 188,9237f$ | $\Delta 260,2508f$ | $\Delta 159,6904f$ | $\Delta 259,5193f$ | $\Delta 188,6247f$ |
| $\Delta 122,84f$   | $\Delta 230,6291f$ | $\Delta 74,00685f$ | $\Delta 232,5296f$ | $\Delta 126,9078f$ |

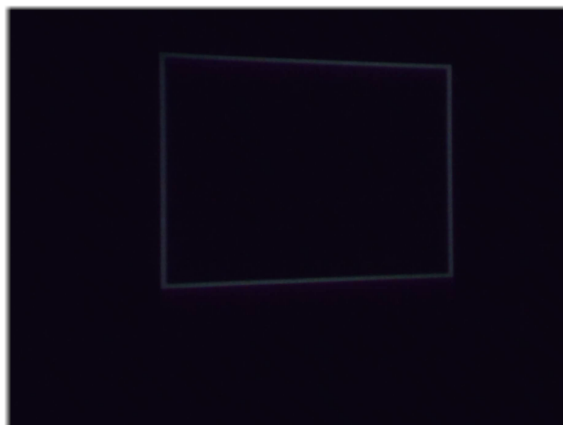
Obr. 4.14: Výsledky tretieho riešenia s parametrom  $d = 0,5$  pri prvom testovacom scenári

Všimnime si, že presnosť sa výrazne zhoršila. čím je parameter  $d$  bližšie k číslu 1 je tretie riešenie presnejšie, ale len v strede. V rohoch je dosť nepresné.

V **prvom scenári** najlepšie obstálo **druhé riešenie**, ktoré bolo najpresnejšie z týchto troch riešení.

### Druhý testovací scenár

Pri druhom testovacom scenári sme kameru umiestnili zľava, tak aby kamera snímala obraz z pohľadu zľava (viď Obrázok 4.15).



Obr. 4.15: Obráz z kamery pri druhom testovacom scenári

V **prvom riešení** sme namerali výberový priemer  $\bar{y} = 51,4699$  a rozptyl výberového priemeru  $s^2 = 437,4304$ . Výsledky prvého riešenia pri druhom testovacom scenári zobrazuje Obrázok 4.16.

|                    |                    |                    |                    |                    |
|--------------------|--------------------|--------------------|--------------------|--------------------|
| $\Delta 31,76799f$ | $\Delta 60,33149f$ | $\Delta 74,41157f$ | $\Delta 108,9974f$ | $\Delta 81,30209f$ |
| $\Delta 25,58825f$ | $\Delta 63,12215f$ | $\Delta 74,74128f$ | $\Delta 42,13389f$ | $\Delta 45,9468f$  |
| $\Delta 40,83815f$ | $\Delta 73,91881f$ | $\Delta 54,96941f$ | $\Delta 52,82735f$ | $\Delta 38,66138f$ |
| $\Delta 35,90052f$ | $\Delta 49,79339f$ | $\Delta 57,35736f$ | $\Delta 43,20942f$ | $\Delta 16,27426f$ |
| $\Delta 24,23305f$ | $\Delta 45,17727f$ | $\Delta 67,49707f$ | $\Delta 46,06869f$ | $\Delta 31,68058f$ |

Obr. 4.16: Výsledky prvého riešenia pri druhom testovacom scenári

Ako sme mohli očakávať, presnosť **prvého riešenia** dosť klesla, v dôsledku toho, že prvé riešenie neráta so zmenou perspektívy.

Pri **druhom riešení** sme namerali výberový priemer  $\bar{y} = 33,8349$  a rozptyl výberového priemeru  $s^2 = 326,7840$ . Výsledky druhého riešenia pri druhom testovacom scenári zobrazuje Obrázok 4.17.

|                    |                    |                    |                    |                    |
|--------------------|--------------------|--------------------|--------------------|--------------------|
| $\Delta 14,05424f$ | $\Delta 47,34341f$ | $\Delta 57,39388f$ | $\Delta 32,96421f$ | $\Delta 16,82016f$ |
| $\Delta 19,11367f$ | $\Delta 45,33245f$ | $\Delta 51,88244f$ | $\Delta 52,72345f$ | $\Delta 14,62681f$ |
| $\Delta 18,73365f$ | $\Delta 68,11796f$ | $\Delta 54,83965f$ | $\Delta 46,43681f$ | $\Delta 12,4828f$  |
| $\Delta 25,42296f$ | $\Delta 48,4006f$  | $\Delta 31,47509f$ | $\Delta 47,55056f$ | $\Delta 2,794037f$ |
| $\Delta 22,34269f$ | $\Delta 18,72179f$ | $\Delta 42,14218f$ | $\Delta 44,4452f$  | $\Delta 9,711857f$ |

Obr. 4.17: Výsledky druhého riešenia pri druhom testovacom scenári

Presnosť **druhého riešenia** nie je až taká dobrá ako v prvom testovacom scenári, ale je lepšia ako presnosť prvého riešenia v druhom testovacom scenári. Druhé riešenie sa lepšie vysporiada so zmenou perspektívy.

Pri **tretom riešení** s parametrom  $d = 0.9$  sme namerali výberový priemer  $\bar{y} = 60,1152$  a rozptyl výberového priemeru  $s^2 = 706,1392$ . Výsledky tretieho riešenia s parametrom  $d = 0.9$  pri druhom testovacom scenári zobrazuje Obrázok 4.18.



|                    |                    |                    |                    |                    |
|--------------------|--------------------|--------------------|--------------------|--------------------|
| $\Delta 72,93784f$ | $\Delta 53,20934f$ | $\Delta 62,76188f$ | $\Delta 80,04524f$ | $\Delta 115,0306f$ |
| $\Delta 84,85839f$ | $\Delta 12,81505f$ | $\Delta 34,72285f$ | $\Delta 28,30923f$ | $\Delta 71,41756f$ |
| $\Delta 81,27098f$ | $\Delta 23,68282f$ | $\Delta 52,62006f$ | $\Delta 35,36974f$ | $\Delta 63,63027f$ |
| $\Delta 89,40706f$ | $\Delta 19,66829f$ | $\Delta 47,37268f$ | $\Delta 39,31598f$ | $\Delta 61,18959f$ |
| $\Delta 80,80874f$ | $\Delta 46,05437f$ | $\Delta 59,81746f$ | $\Delta 80,83823f$ | $\Delta 105,7259f$ |

Obr. 4.18: Výsledky tretieho riešenia s parametrom  $d = 0,9$  pri druhom testovacom scenári

Môžeme pozorovať, že presnosť pri **treťom riešení** sa so vzdáľovaním od stredu takisto zhoršuje aj v druhom scenári. Pri zmene parametra  $d = 0.5$  sme namerali výberový priemer  $\bar{y} = 173,3343$  a rozptyl výberového priemeru  $s^2 = 3753,2810$ . Tretie riešenie s parametrom  $d = 0.5$  pri druhom testovacom scenári zobrazuje Obrázok 4.19.

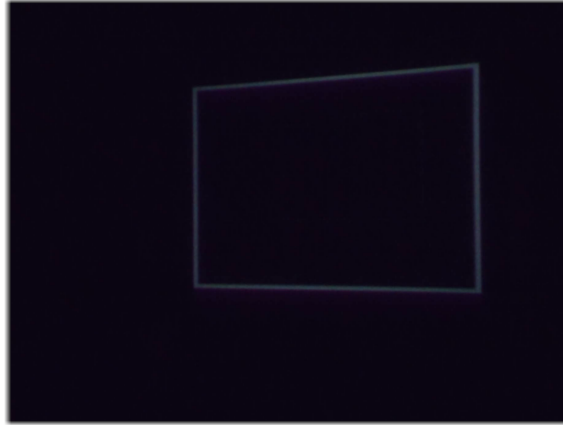
|                    |                    |                    |                    |                    |
|--------------------|--------------------|--------------------|--------------------|--------------------|
| $\Delta 128,735f$  | $\Delta 213,0701f$ | $\Delta 67,01623f$ | $\Delta 216,1893f$ | $\Delta 104,278f$  |
| $\Delta 197,1153f$ | $\Delta 244,253f$  | $\Delta 167,0285f$ | $\Delta 251,9255f$ | $\Delta 154,7426f$ |
| $\Delta 91,40408f$ | $\Delta 167,5199f$ | $\Delta 107,3271f$ | $\Delta 258,4819f$ | $\Delta 111,5834f$ |
| $\Delta 205,8345f$ | $\Delta 232,995f$  | $\Delta 191,069f$  | $\Delta 277,8812f$ | $\Delta 180,8801f$ |
| $\Delta 131,1103f$ | $\Delta 208,0728f$ | $\Delta 83,13929f$ | $\Delta 230,9524f$ | $\Delta 110,7526f$ |

Obr. 4.19: Výsledky tretieho riešenia s parametrom  $d = 0,5$  pri druhom testovacom scenári

V **druhom scenári** najlepšie obstálo opäť **druhé riešenie**, ktoré bolo najpresnejšie z týchto troch riešení.

### Tretí testovací scenár

Pri treťom testovacom scenári sme tentoraz kameru umiestnili sprava, tak aby kamera snímala obraz z pohľadu sprava (viď Obrázok 4.20).



Obr. 4.20: Obráz z kamery pri treťom testovacom scenári

V **prvom riešení** sme namerali výberový priemer  $\bar{y} = 51,8861$  a rozptyl výberového priemeru  $s^2 = 323,4306$ . Obdobne ako pri druhom scenári presnosť **prvého riešenia** klesla i v treťom scenári z rovnakých príčin.

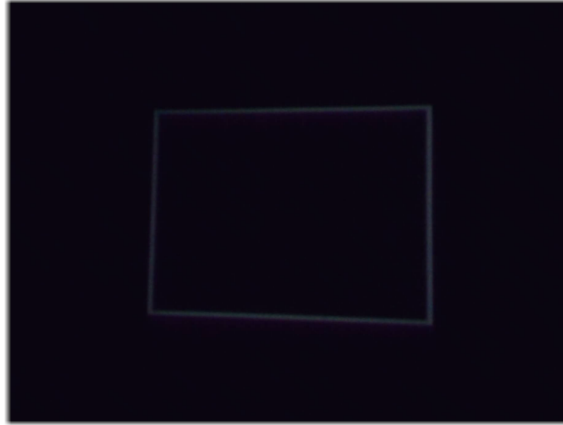
Pri **druhom riešení** sme namerali výberový priemer  $\bar{y} = 37,9524$  a rozptyl výberového priemeru  $s^2 = 311,3026$ .

Pri **treťom riešení** s parametrom  $d = 0.9$  sme namerali výberový priemer  $\bar{y} = 62,5213$  a rozptyl výberového priemeru  $s^2 = 594,8466$ . Presnosť pri **treťom riešení** sa so vzdáľovaním od stredu sa zhoršuje aj v treťom scenári. Pri zmene parametra  $d = 0.5$  sme namerali výberový priemer  $\bar{y} = 172,6442$  a rozptyl výberového priemeru  $s^2 = 4466,6510$ .

V **treťom scenári** najlepšie obstálo opäť **druhé riešenie**, ktoré bolo najpresnejšie z týchto troch riešení i keď sa nám pri treťom scenári mohli zdať prvé a tretie riešenie skoro približne presné.

#### Štvrtý testovací scenár

Pri štvrtom testovacom scenári sme si pozvali nezávislú osobu, ktorá si umiestnila kameru podľa seba, tak ako jej vyhovovalo (viď Obrázok 4.21).



Obr. 4.21: Obráz z kamery umiestnenou nezávislou osobou pri štvrtom testovacom scenári

V **prvom riešení** sme namerali výberový priemer  $\bar{y} = 31,0753$  a rozptyl výberového priemeru  $s^2 = 266,8666$ .

Pri **druhom riešení** sme namerali výberový priemer  $\bar{y} = 13,5799$  a rozptyl výberového priemeru  $s^2 = 42,2760$ .

Pri **treťom riešení** s parametrom  $d = 0.9$  sme namerali výberový priemer  $\bar{y} = 53,46875$  a rozptyl výberového priemeru  $s^2 = 892,8430$ . Pri zmene parametra  $d = 0.5$  sme namerali výberový priemer  $\bar{y} = 170,4061$  a rozptyl výberového priemeru  $s^2 = 4127,8180$ .

Vo **štvrtom scenári** najlepšie obstálo **druhé riešenie**, ktoré bolo výrazne presnejšie ako ostatné dve riešenia.

Zo všetkých štyroch scenárov dopadlo najlepšie **tretie riešenie**, ktoré vykazovalo dostatočne dobrú presnosť za rôznych podmienok.

#### 4.2.5 Implementácia simulovania kliknutia ľavým tlačidlom myši

K implementácii simulovania kliknutia ľavým tlačidlom myši sa dalo pristupovať viacerými spôsobmi. Pre nás zaujímavé spôsoby sme si odskúšali. I keď niektoré prístupy fungovali dobre, pre nás boli nevyhovujúce, ale spomenieme ich, aj s dôvodmi, prečo sa nám nehodili.

##### Prvé riešenie - Windows cursor

Prvý prístup bolo použitie externej metódy `void mouse_event(...)` z windows-u cez jeho API importovaním knižnice `user32.dll` [18] (Listing 4.12).

```
1 [DllImport("user32.dll", CharSet=CharSet.Auto, CallingConvention=
   CallingConvention.StdCall)]
```

```
2 public static extern void mouse_event(uint dwFlags, uint dx, uint dy,
   uint cButtons, uint dwExtraInfo);
3 private const int MOUSEEVENTF_LEFTDOWN = 0x02;
4 private const int MOUSEEVENTF_LEFTUP = 0x04;
5
6 public static void PerformLeftClick(Vector2 pos)
7 {
8     mouse_event(MOUSEEVENTF_LEFTDOWN | MOUSEEVENTF_LEFTUP, pos.X, pos.Y
9     , 0, 0);
10 }
```

Listing 4.12: Simulácia kliknutia cez Windows API

Vyskytli by sa tu aj drobné komplikácie, ako potreba prepočítavať súradnice ak je aktívnych viacej obrazoviek. Na ilustráciu majme 2 obrazovky s rozlíšením 1920x1080, aplikácia beží na druhej obrazovke a hlavná obrazovka je prvá. X-ové súradnice kurzora myši na druhej obrazovke by boli v intervale  $\langle 1920, 3840 \rangle$  a Y-ové súradnice by v tomto prípade zostali v poriadku, interval  $\langle 0, 1080 \rangle$ .

Najväčší problém s týmto prístupom bol, že sme ovládali kurzor, čo nie je vždy vhodný prístup, lebo niekto by mohol chcieť používať kurzor myši v inom okne aj počas behu aplikácie. Navyše, tento prístup funguje len pre platformu Windows. Tento prístup je asi najkomplikovanejší z tých, čo uvedieme.

Toto riešenie sme teda nemohli použiť, lebo by sme limitovali aplikácie kompatibilné s inými platformami, ak by chceli integrovať náš plugin.

### Druhé riešenie - standalone input system

Druhý prístup bol použiť vstavanú Unity triedu *StandaloneInputModule* [19], respektíve odvodiť si vlastnú triedu a dediť z *StandaloneInputModule* a kedykoľvek chceme vyvolať kliknutie ľavým tlačidlom myši, zavoláme metódu *ProcessTouchPress(pointerData, true, true)*. Kde *pointerData* obsahuje X-ovú a Y-ovú súradnicu kliknutia.

Problém s druhým riešením je, že ak sa v aplikácií používa už nejaký iný *StandaloneInputModule* alebo vlastný modul dediaci *StandaloneInputModule*, tak by to bol problém, lebo v Unity v jednej scéne by sa mal nachádzať najviac jeden takýto modul. Čiže by sme programátorov ukrátili o ich prípadný vlastný *StandaloneInputModule*. Podobne, ani tento prístup nie je vhodný pre naše potreby.

### Tretie riešenie - raycast z kamery

Tretí prístup je trochu zložitejší oproti druhému, ale jednoduchší ako prvý. Použili sme raycast z kamery. Tu rozlišujeme 2 druhy raycast-ov. Prvý raycast je pre UI objekty, odohráva sa v screen-space. Druhý raycast je pre *GameObject-y*, t. j. objekty v scéne,

odohráva sa v world-space. Dohodli sme sa, že ak detegujeme UI objekt prvým raycast-om, už druhý raycast nerobíme, pričom vždy sa robí v poradí prvý raycast pre UI a potom druhý raycast pre objekty v scéne. UI má teda prioritu. Raycast UI objektov vykonávame pomocou statickej metódy v Unity, *EventSystem.current.RaycastAll(pointerEventData, guiRaycastResults)*, ktorá vyžaduje dva parametre, *pointerEventData* čo sú súradnice kliknutia a *guiRaycastResult* do ktorého sa nám vložia výsledky raycast-u, t. j. všetky objekty, ktoré detegoval raycast v poradí postupne ako sa detegovali. V ďalšom kroku iterujeme cez tieto výsledky a snažíme sa získať UI objekt typu *Button*, *RadioButton*, *Toggle*, *DropDown*, atď., a aktivovať ho ak sa dá. Ak sme nenarazili na žiadne UI objekty takýchto typov, tak spravíme druhý raycast. Najprv nám treba ale ešte pomocnú triedu, interface. Vytvorili sme si interface *LPIPIInteractable* (Listing 4.13).

```

1 public interface LPIPIInteractable
2 {
3     public void LPIPOnLaserHit ();
4 }

```

Listing 4.13: Interface pre objekty podporujúce interakciu s laserom

Tento interface musia implementovať všetky *GameObject-y*, t. j. objekty v scéne, ktoré chcú fungovať s laserovým lúčom. Druhý raycast vykonáme pomocou statickej Unity metódy *Camera.main.ScreenPointToRay(clickPosition)*, ktorá vyšle raycast zo súradníc *clickPosition* rovnobežne s osou Z a keď narazí na objekt, tak ho vráti. My sa následne budeme snažiť na tento objekt zavolať metódu *LPIPOnLaserHit()* pokiaľ sa bude dať, respektíve pokiaľ bude daný objekt implementovať interface *LPIPIInteractable*.

Možno sa vám bude zdať, že potreba implementovať náš interface je obmedzujúca, no ukážeme vám, že to tak len vyzerá. Totiž, ak chceme aby objekt v scéne reagoval na kliknutie ľavým tlačidlom myši, tak musíme preťažiť z *MonoBehaviour* metódu *OnMouseDown()*. Takže je to v podstate to isté. Tento spôsob navyše umožňuje rôzne reakcie pri kliknutí myšou a pri simulovanom kliknutí laserovým lúčom. Keby sme však nepožadovali takýto prístup cez náš interface, my by sme museli zisťovať, ktorý z komponentov (skriptov) objektu preťažuje metódu *OnMouseDown()*, čo ak je ich viac? Zavolali by sme jeden alebo všetky? Takýmto prístupom máme všetko relevantné na jednom mieste a ani tým nezvyšujeme prácu programátorovi, ani nám.

Jedna nevýhoda tretieho riešenia je, že musíme pre každý typ UI objektu pridať kód na jeho interakciu (Listing 4.14), čo nepredstavuje až taký problém, lebo najčastejšie UI objekty sú button-y, dropdown-y a checkbox-y.

```

1 foreach (var r in _guiRaycastResults){
2     //try button
3     r.gameObject.TryGetComponent(out Button button);
4     if (button != null){
5         if (button.interactable){

```

```

6         button.onClick?.Invoke();
7         _isUIDetected = true;
8         break;
9     }
10 }
11 //try toggle
12 r.gameObject.TryGetComponent(out Toggle toggle);
13 if (toggle != null){
14     if (toggle.interactable){
15         toggle.isOn = !toggle.isOn;
16         _isUIDetected = true;
17         break;
18     }
19 }
20 ... // other UI objects
21 }

```

Listing 4.14: Časť kódu zobrazujúca nevýhodu pri treťom riešení 4.2.5

## 4.2.6 Implementácia časti Utility

V rámci implementácie časti utility pre plugin čitateľa stručne oboznámime s jej implementáciou.

Utility obsahovali triedu *LPIPUtilityController*, ktorý ovládal všetky moduly v časti Utility, t. j. konfiguračné/kalibračné menu, debugovacie UI a indikátor miesta detekcie laserového lúča. Všetky tieto spomenuté moduly fungovali samostatne komunikovali s ovládačom *LPIPUtilityController*. Modul konfiguračné menu spravoval *LPIPConfigurationMenuController*, modul kalibračné menu spravoval *LPIPCalibrationMenuController*, modul debugovacie UI spravoval *DebugTextController* a modul indikátor miesta detekcie laserového lúča spravoval *LPIPMarkerUIController*.

Pre programátora alebo iný systém sme vytvorili triedu *LPIPUtilityPortal* podľa návrhového vzoru **Singleton**, ktorý zviditeľňoval a poskytoval nami vybrané metódy ako napríklad otvorenie a zatvorenie menu utilít, zobrazenie obrazu z kamery a podobne.

Výslednú implementáciu plugin-u sme zbalili do jedného prefab-u v Unity, aby programátorovi stačilo použiť metódu *Drag&Drop* do scény a niečo<sup>2</sup> do-programovať.

## 4.3 Aplikácia

Na rozdiel od implementácie plugin-u, implementácia aplikácie je o čosi jednoduchšia.

<sup>2</sup>interface *LPIPIInteractable* pre interakciu objektov s pluginom

### 4.3.1 Implementácia hlavného menu

Hlavné menu aplikácie sme implementovali podľa návrhu. O funkcionality hlavného menu sa stará komponent<sup>3</sup> *UIController*. *UIController* komunikuje so sub-komponentami *GamePlayUIController* a *LevelMenuController*.

*GamePlayUIController* sa využíva počas trvania hry. Poskytuje funkcionality pre návrat do menu z hracieho prostredia.

*LevelMenuController* slúži na ovládanie menu výberu úrovne, zabezpečuje funkcionality tlačidlám +, - a riadi logiku spustenia hry so zvolenou úrovňou.

*UIController* okrem komunikácie so sub-komponentami komunikuje aj s plugin-om s *LPIPUtilityPortal*. *UIController* zabezpečuje spustenie plugin-u cez *LPIPUtilityPortal.Instance.OpenUtilityMenu()* a reaguje na udalosť, keď sa zavrie utility menu, aby zobrazil hlavné menu aplikácie.

### 4.3.2 Implementácia hernej logiky

Ako prvé sme implementovali komponent *CubeManager*. *CubeManager* má na starosti správu objektov v scéne, s ktorými hráč interaguje. *CubeManager* vytvára potrebný počet objektov pomocou veľmi jednoduchého **object pooling** systému, vydáva príkazy objektom obnoviť sa do východzieho stavu a znižuje objekty v prípade, ak by sa už nezmestili. Z pochopiteľných dôvodov, keďže nemôžeme znižovať do nekonečna, tak aby objekty boli stále viditeľné, tak sme to obmedzili na nejakú minimálnu veľkosť pod ktorú sa už nedá ísť.

Každý objekt v scéne má na sebe komponent *CubeController*. *CubeController* ovláda konkrétny objekt, vie svoje poradové číslo, obnovuje objekt do východzieho stavu a reaguje na kliknutie ľavým tlačidlom myši cez metódu *OnMouseDown()*. Teda keď niekto klikne na daný objekt, tak ten konkrétny objekt upozorní *GameController* a pribalí k tomu informáciu so svojím id<sup>4</sup>.

Na záver, najdôležitejší komponent *GameController*, ktorý ovláda logiku hry. *GameController* štartuje úroveň, prikazuje objektom obnoviť sa do východzieho stavu cez *CubeManager*, prikazuje objektom zobraziť, respektíve skryť svoje číslo po uplynutí časového intervalu (viď Obrázok 4.22), vyhodnocuje či hráč klikol na objekt so správnym poradovým číslom, t. j. či kliká na objekty v správnom poradí (viď Obrázok 4.23). Pri kliknutí na ľubovoľný objekt, daný objekt zobrazí svoje číslo a ak nastal prípad, že hráč klikol na objekt s nesprávnym poradovým číslom, ktoré malo nasledovať, bude upozornený zmenou farby objektu na červenú (viď Obrázok 4.23a) a *GameController* po chvíľke aktuálnu úroveň reštartuje. V prípade kliknutia na objekt so správnym poradovým číslom, objekt dá vedieť hráčovi zmenením farby na zeleno (viď Obrázok 4.23b) a

<sup>3</sup>komponentami sa v Unity nazývajú jednotlivé skripty, ktoré sa priradia objektom v scéne

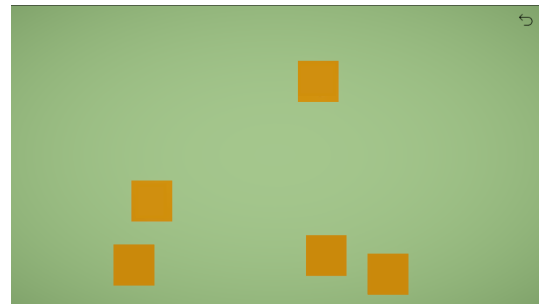
<sup>4</sup>v našom prípade je id rovné poradovému číslu



ak to náhodou bol posledný objekt s posledným poradovým číslom, tak *GameController* spustí novú úroveň s o jedno viac objektami.

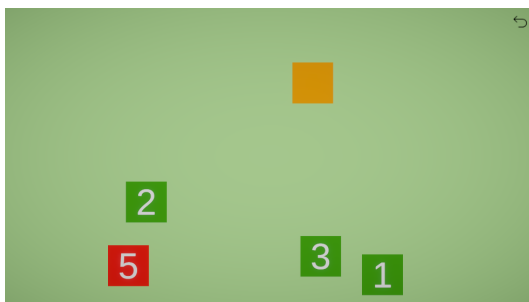


(a) Objekty ukazujú svoje poradové číslo



(b) Objekty skryli svoje poradové číslo

Obr. 4.22: Mechanika hry - ukáž / skry



(a) Nesprávne zvolenie objektov v poradí



(b) Správne zvolenie objektov v poradí

Obr. 4.23: Mechanika hry - dobrý / zlý

*GameController* sme nechceli zaťažovať zbytočne, tak on sám o sebe nemá referencie na jednotlivé objekty v scéne. Implementácia komunikácie s objektami v scéne je podobná návrhovému vzoru **Observer**, ale funguje na princípe udalostí. Keď hráč klikne na objekt, ten objekt vytvorí udalosť so svojím id. *GameController* na túto udalosť zareaguje, vyhodnotí výsledok a potom on sám vygeneruje udalosť s id, komu je udalosť určená, t. j. ostatný ju ignorujú a pribalí k tomu aj informáciu o výsledku, t. j. či bola odpoveď správna alebo nesprávna, aby objekt s daným id vedel príslušne zareagovať. Každý objekt v scéne je teda prihlásený na odber udalostí generovaných *GameController-om* a na túto informáciu reagujú komponenty týchto objektov pomocou metódy *AnswerEvent(...)*.

### 4.3.3 Integrácia pluginu do aplikácie

Integrácia plugin-u do hry prebiehala veľmi jednoducho a pohodlne. Prvý krok bol, vložiť do scény *LPIP\_Plugin* prefab plugin-u.

Druhý krok, ak sme chceli aby nejaký objekt/objekty interagovali s plugin-om museli sme implementovať interface nášho plugin-u *LPIPIInteractable*. Teda stačilo v komponente *CubeController* zabezpečiť aby sa zavolała metóda *OnMouseDown()*, t. j. aby *CubeController* implementoval interface *LPIPIInteractable* (viď Listing 4.15).

```

1 public class CubeController : MonoBehaviour, LPIPIInteractable
2 {
3     ...
4     public void LPIPOnLaserHit ()
5     {
6         OnMouseDown ();
7     }
8     ...
9 }

```

Listing 4.15: Implementácia interface-u pre interakciu objektu s plugin-om

Tretí krok, potrebujeme naštartovať plugin. Na to nám stačí pridať funkcionality do komponentu *UIController* pre tlačidlo *Plugin* na spustenie menu utilít plugin-u. K tomu použijeme na to určený komponent *LPIPUtilityPortal* a jeho metódu *LPIPUtilityPortal.Instance.OpenUtilityMenu()*. Ďalej zabezpečíme skrytie nášho menu a jeho odokrytie pri ukončení menu utilít pomocou prihlásenia sa na udalosti *LPIPUtilityController.OnUtilityMenuDisabled*, ktoré generuje *LPIPUtilityController*.

## 4.4 Obmedzenia a odporúčania k plugin-u

Podporované kamery sú ekvivalentné s podporovanými kamerami platformy Unity. Plugin funguje len pre jeden vstup (t. j. jeden laserový lúč). Plugin rozpoznáva len laserové lúče viditeľného svetla červenej farby. V prípade, že sa viaceré osoby snažia ovládať aplikáciu viacerými laserovými lúčmi červenej farby, bude to považované ako jeden, ktorý v jednom okamihu sa nachádza na viacerých miestach a bude sa postupovať postupom uvedeným v implementácii. Plugin simuluje kliknutie ľavým tlačidlom myši každý frame, je na programátorovi si to obmedziť.

Odporúča sa použiť kvalitnejšia kamera, nie nutne s vysokým rozlíšením. Pred kamerou by sa nemali nachádzať iné objekty, ktoré sú schopné odrazov svetla, okrem projekčného plátna. Odporúča sa umiestniť kameru čo najlepšie ako sa dá (pre dosiahnutie lepšej presnosti).

Odporúča sa využívať v miestnosti s čo najmenším počtom okien z východnej, južnej alebo západnej časti (pre minimalizáciu vplyvu slnečného svetla). Miestnosť v ktorej sa používa aplikácia s plugin-om by mala byť rovnomerne osvetlená a nie príliš alebo vôbec.

Oblasť ktorá je snímaná kamerou nesmie obsahovať priame slnečné svetlo (polarizačné filtre by to ako tak zvládli odfiltrovať, ale laserový lúč by už nebol dostatočne viditeľný).

# Záver

Sem príde záver na konci roka.

# Literatúra

- [1] obrázok pikachu. <https://cdn.shoplightspeed.com/shops/633327/files/29258343/156x230x1/ftla-art-kit-painting-acrylic-pikachu.jpg>. Pristupované 1.5.2023.
- [2] C. Moore and J. M. Robson. Hard tiling problems with simple tiles. *Discrete & Computational Geometry*, 26(4):573–590, January 2001.
- [3] S. W. Golomb. *Polyominoes: Puzzles, patterns, problems, and packings*. Princeton University Press, 2nd edition, 1994.
- [4] A. Biere. CaDiCaL Simplified Satisfiability Solver. <https://github.com/arminbiere/cadical>. Retrieved May 5, 2022.
- [5] The Sage Developers. *SageMath, the Sage Mathematics Software System (Version 9.4)*, 2021. <https://www.sagemath.org>.
- [6] Claire Caplan, Alec Wyatt, and Shreyas Renganathan. Two-player boids game with laser pointer controllers. [https://people.ece.cornell.edu/land/courses/ece4760/FinalProjects/f2021/crc235\\_acw254\\_sr2322/crc235\\_acw254\\_sr2322/index.html](https://people.ece.cornell.edu/land/courses/ece4760/FinalProjects/f2021/crc235_acw254_sr2322/crc235_acw254_sr2322/index.html), 2021. Pristupované 4.3.2023.
- [7] Laser Ammo Ltd. Smokeless range © 2.0 - home simulator. <https://www.laserammo.eu/smokeless-range-at-home-range>, 2015. Pristupované 4.3.2023.
- [8] Laser Ammo Ltd. Smokeless range © 2.0 - starting guide video. <https://www.youtube.com/watch?v=sC8pXNrb1Y>, 2023. Pristupované 5.3.2023.
- [9] Unity. Made with unity. <https://unity.com/made-with-unity>, 2023. Pristupované 18.3.2023.
- [10] Unity. Students get in-demand job skills with unity. <https://unity.com/case-study/university-miami-uaa-case-study>, 2023. Pristupované 18.3.2023.
- [11] Alexey Stomakhin and Alice Gardner. The water technology behind avatar: The way of water. <https://blog.unity.com/industry/technology-behind-avatar-the-way-of-water>, 2023. Pristupované 18.3.2023.

- [12] Brad Jones. What is c#? <https://www.developer.com/guides/what-is-c/>, 2001. Pristupované 5.3.2023.
- [13] Milan Sonka, Václav Hlaváč, and Roger Boyle. *Image processing, Analysis, and Machine Vision*. Cengage Learning, 4th edition, 2015.
- [14] obrázok FMFI UK. <https://navyske.sk/data/images/754.jpg>. Pristupované 19.3.2023.
- [15] ITU. *Parameter values for the HDTV standards for production and international programme exchange*, page 4. ITU, 6nd edition, 2015.
- [16] Andrej Lúčný. Point tranformation algorithm. <https://github.com/andyLucny/calib4points/blob/main/calib2.py>. Pristupované 24.2.2023.
- [17] Andrej Lúčný. Point tranformation algorithm. <https://github.com/andyLucny/calib4points/blob/main/calibqkv.py>. Pristupované 15.3.2023.
- [18] Microsoft. Documentation (winuser.h). [https://learn.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-mouse\\_event](https://learn.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-mouse_event). Pristupované 22.4.2023.
- [19] Unity. Documentation (standaloneinputmodule). <https://docs.unity3d.com/560/Documentation/ScriptReference/EventSystems.StandaloneInputModule.html>. Pristupované 22.4.2023.

# Dodatok A

## Zdrojové kódy

Kompletný zdrojový kód je k dispozícii na Github repozitáry na odkaze: <https://github.com/CapitanMikon/LaserPointerInputPlugin>

# Dodatok B

## Návody a příručky

Návody how-to a příručky what-does-what ...