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 prekríženými 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 jednoduchej počítačovej hry

Cieľ: Práca má dva ciele:
1. vyvinúť plugin implementujúci podporu laserovej pištole
2. vyskúšať ho v jednoduchej 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

Poďakovanie: Chcel by som veľmi pekne poďakovať vedúcemu práce RNDr. Andrejovi Lúčnemu, PhD. za poskytnutú tému i požičanie potrebného vybavenia. Najviac vďačím za neoceniteľné rady pri tvorbe písomnej časti bakalárskej práce i praktickej časti a hlavne za ochotu a venovaný čas.

Abstrakt

KICA, Marián: Plugin pre Unity implementujúci podporu laserovej pištole [Bakalárska práca]. Univerzita Komenského v Bratislave. Fakulta matematiky, fyziky a informatiky; Katedra aplikovanej informatiky. Vedúci bakalárskej práce: RNDr. Andrej Lúčny, PhD. Bratislava: FMFI UK, 2023, 73 strán, 45 obrázkov, 15 výstrižkov kódu, 1 tabuľka.

Naša bakalárska práca sa zaoberá implementáciou plugin-u pre Unity implementujúceho podporu laserovej pištole a implementáciou počítačovej hry, do ktorej bude tento plugin integrovaný. V našej práci sa čitateľ dozvie všetko od návrhu po rôzne metódy implementácie, ktoré sme odskúšali, ich výsledky, výhody i nevýhody a pri relevantných implementačných riešeniach aj ich porovnania. Výsledkom práce je plugin pre Unity, ktorý je dostupný na GitHub repozitári uvedenom v Dodatku A a k tomu manuál na použitie plugin-u a jednoduchá počítačová hra, do ktorej bol tento plugin implementovaný a odskúšaný.

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

Abstract

KICA, Marián: Plugin for Unity implementing laser gun support [Bachelor thesis]. Comenius University in Bratislava. Faculty of Mathematics, Physics and Informatics; Department of Applied Informatics . Bachelor thesis supervisor: RNDr. Andrej Lúčny, PhD. Bratislava: FMFI UK, 2023. 73 pages, 45 figures, 15 listings, 1 table.

Our bachelor thesis deals with implementing a plugin for Unity that implements laser gun support and a computer game into which we integrate it. In our thesis, the reader will learn everything from the design to the different implementation methods we have tried, their results, advantages and disadvantages, and their comparisons. The result of this bachelor's thesis is a plugin for Unity, available on the GitHub repository listed in Appendix A, plus a manual on how to use the plugin and a simple computer game in which we tested it.

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

Obsah

Zoznam obrázkov

Zoznam kódov

Úvod

Interakcia s počítačom je v dnešnej dobe, aspoň v rozvinutejších krajinách, pomerne bežnou záležitosťou. Aplikáciu, ktorá beží na počítači ovládame pomocou klávesnice a myši. Keď k počítaču zapojíme projektor, vieme prezentovať obrazovku počítača, väčšinou ide o slajd-y cez prezentačný softvér. Prezentáciu môžeme ovládať myšou, klávesnicou, resp. špeciálnym zariadením, ktoré pripojíme k počítaču pomocou bezdrôtového USB dongle. Avšak naše ovládanie aplikácie je často limitované na posúvanie slajd-ov. Čo keby sme mohli viac? Vedeli by sme spustenú aplikáciu zobrazovanú projektorom ovládať omnoho interaktívnejšie, celkový zážitok by bol lepší. Nemuseli by sme nutne stáť vedľa počítača v dosahu klávesnice, myši a vedeli by sme nie len ovládať prezentácie, ale ľubovoľné[2] aplikácie z rôznej pozície[3]. Dnes si už asi každý odskúšal virtuálnu realitu. Virtuálna realita ponúka interakciu ako zo Sci-Fi, síce ešte len jednoduchú a viac menej audio-vizuálnu a pre jednu osobu. Virtuálna realita nie je moc rozšírená a viacej ľudí má k dispozícii skôr projektor ako headset pre virtuálnu realitu. Taktiež, nosiť displej na hlave nie je vždy a pre dlhšie používanie moc komfortné. Za pomoci dodatočného hardvéru (kamery) a spracovania obrazu vieme poskytnúť vylepšenú interakciu s počítačom pri využití projektora. "Môžeme klikať" na objekty pomocou laserového lúča a tak ovládať aplikáciu[1], skoro ako keby sme používali myš [1].

Tému pre túto bakalársku prácu sme si zvolili na základe svojich osobných preferencií a smerom, ktorým sa chceme v budúcnosti uberať. Navyše, pre nás táto práca bola veľmi zaujímavá, nie len pre svoju interaktívnosť, ale i preto, že sa v nej využívajú technológie, ktorým by sme sa chceli viacej venovať. Podobných riešení venujúcich sa tejto téme je len málo, aspoň z tých verejne známych a dostupných.

Táto bakalárska práca má skôr implementačný charakter. Jej cieľom je vytvoriť plugin pre podporu laserovej pištole, ktorou je možné mieriť na stenu, na ktorú je dátovým projektorom premietaná počítačová hra. Potom sa skutočný laserový lúč premení na analogický laserový lúč vo virtuálnom prostredí. Vytvorený plugin je následne využitý na implementáciu jednoduchej počítačovej hry. Dôraz kladieme na plugin, preto sa len okrajovo venujeme hernej mechanike pilotnej aplikácii.

Prácu sme rozčlenili do štyroch hlavných kapitol. V prvej kapitole "Predchádzajúce práce"

[2] vytvorené/ú v Unity

[3] v okolí, kde je zobrazovaný obraz z počítača cez projektor

oboznámime čitateľa s riešeniami podobnými našej práci, ku ktorým sa nám podarilo dopátrať. Ku každému riešeniu máme pripravený stručný opis o čom je, ktorý je ďalej sprevádzaný opisom ako funguje dané riešenie a na záver sa zaoberáme výhodami a nevýhodami daného riešenia.

V druhej kapitole "Východiská práce" uvedieme použité technológie v našej práci. Spravíme veľmi ľahký a stručný úvod do problematiky spracovania obrazu a počítačového videnia kde uvedieme, čo všetko zahŕňa spracovanie obrazu a ako principiálne funguje.

V tretej kapitole "Špecifikácia požiadaviek a návrh systému" predstavíme špecifikáciu plugin-u aj aplikácie. Po špecifikácii nasleduje detailný návrh plugin-u a aplikácie, v ktorom sa čitateľ dozvie celkový návrh, návrh potrebnej mechanickej výbavy ako je upevnenie polarizačných filtrov, návrh plugin-u aj jeho používateľského rozhrania. Záver kapitoly ukončíme návrhom aplikácie (jednoduchej počítačovej hry) a jej používateľského rozhrania.

V štvrtej kapitole "Implementácia systému a výsledky" máme pre čitateľa pripravenú stručnú implementáciu upevnenia polarizačných filtrov, ktorá začne prototypom a skončí robustnejším produktom. Ďalej bude nasledovať detailná implementačná časť plugin-u, v ktorej sa zameriame na hlavné časti implementácie sprevádzané stručnými výstrižkami kódov alebo pseudo-kódov. Spomenieme výhody i nevýhody odskúšaných riešení a relevantné riešenia porovnáme z hľadiska výkonu či presnosti. Nezabudneme taktiež spomenúť implementáciu aplikácie a integráciu plugin-u do vytvorenej aplikácie. V závere štvrtej kapitoly sme si pripravili obmedzenia a odporúčania k plugin-u.

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

Táto sekcia je spracovaná z internetového zdroja, kde bola práca zverejnená [2].

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 *ObsoleteDesigning 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 ukazovátko.

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 svojím predátorom nech sa pohnú tým smerom, kam ukazujú so svojim laserovým lúčom, 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 [1]. 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[2].

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 [3]. 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 zobraziť výsledok a prípadne 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

[1] autori uvádzali 40 ks

[2] čierno-biela

kalibrácia uložiť a znova použiť, pokiaľ zariadenia nezmenili svoju polohu. [4]

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 požívanými 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

Východiská 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 [5]. Ale aj vo vzdelávaní, Univerzita v Miami použila XR aplikácie vytvorené v Unity na trénovanie chirurgov [6]. 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 [7].

2.1.2 Programovací jazyk C#

C# je moderný, vysoko úrovňový, typový, objektovo orientovaný programovací jazyk vytvorený Microsoftom. Spočiatku closed-soruce, neskôr open-source. Syntaxou je podobný jazykom C++ a Java Používa sa v .NET aplikáciach, Windows Forms a Universal Windows Platform[1] [8].

[1] známe aj pod skratkou UWP

2.2 Spracovanie obrazu a počítačové videnie

S rapídnym 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. Počítačove 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, prispôsobenie modelu a predikcia pohybu.

2.2.1 Problémy pri spracovaní obrazu

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 stavia na základe predchádzajúcich poznatkov a skúseností. K tomuto sa používa umelá inteligencia, aby obohatila počítače schopnosťou porozumieť pozorovaniam. Pod interpretáciou obrazu môžme 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 výrobnej 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, 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.

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 [9]. 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 vedia 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, predspracovanie 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 použiť a aj používajú metódy pomocou umelej inteligencie. High-level počítačove videnie sa snaží napodobniť ľudské poznanie a schopnosť robiť rozhodnutia podľa informácie obsiahnutej v obraze. 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 $m,n \in \mathcal{N}$. Jednotlivé prvky matice sú väčšinou vektory troch hodnôt (červenej, zelenej a modrej farby) z intervalu <0, 255>. 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 predspracovania obrazu. Následne sa môžu použiť metódy na zvýraznenie niektorých čŕt 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 a 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.

Hoci z počítačového videnia a spracovania obrazu [10] v našej práci využívame len zlomok, museli sme sa v priebehu práce oboznámiť aj s metódami, ktoré sme nakoniec nepoužili, ale použitie ktorých sme zvažovali.

Kapitola 3

Špecifikácia požiadaviek a návrh systému

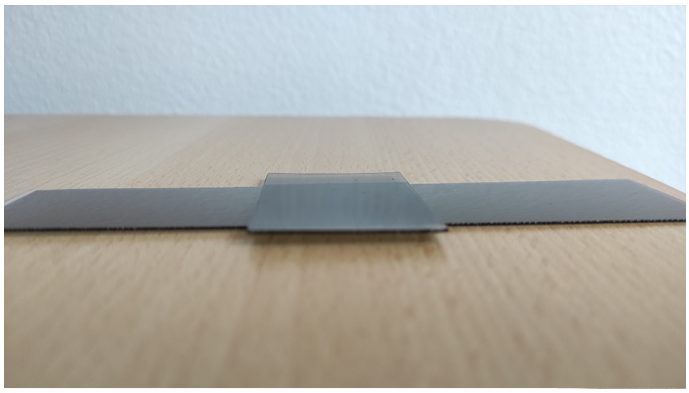
V tejto kapitole uvedieme špecifikáciu požiadaviek a postupne oboznámime čitateľa s celkovým návrhom, návrhom upevnenia polarizačných filtrov, návrhom plugin-u a aplikácie, do ktorej bude tento plugin neskôr integrovaný.

3.1 Špecifikácia požiadaviek

Plugin bude pracovať na platforme Unity. Plugin bude spracovávať obraz z kamery. V spracovávanom obraze bude zistovať 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[1]. 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, napríklad bežný herný notebook. K počítaču musí byť zabezpečené pripojenie projektora a webkamery alebo kamery. Na webkamere/kamere musí byť upevnená dvojica vhodne prekrížených polarizačných filtrov. Medzi týmito dvoma filtrami nesme byť žiadna medzera (t. j. musia byť na sebe ako je uvedené na Obrázku 3.1).

[1] verzia 2021.3.10f1 a vyššie



Obr. 3.1: Ilustračný obrázok polarizačných filtrov umiestnených na sebe

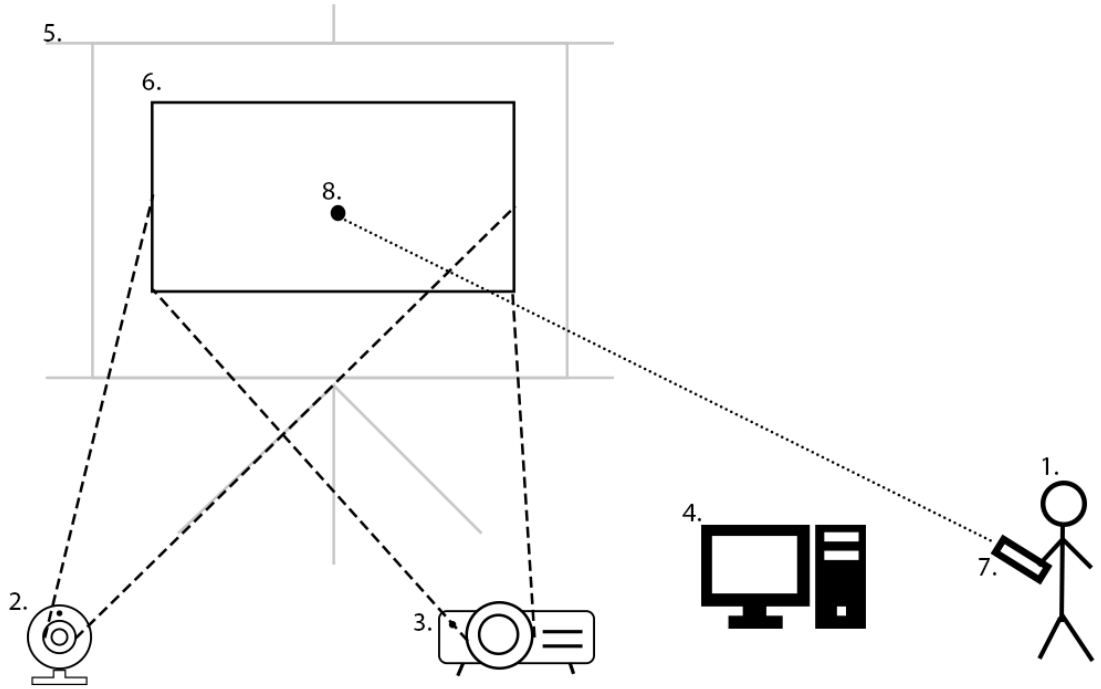
Plugin bude schopný pracovať s ľubovoľnou[2] webkamerou a ľubovoľným projektorom. Plugin bude fungovať nezávisle od projektu. Jeho použitie bude čo najjednoduchšie, aby programátorovi stačilo len tento plugin importovať do projektu a nastaviť zopár hodnôt. Je odporúčané používať systém v miestnosti s 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 hlavní 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.

[2]podporovanou Unity



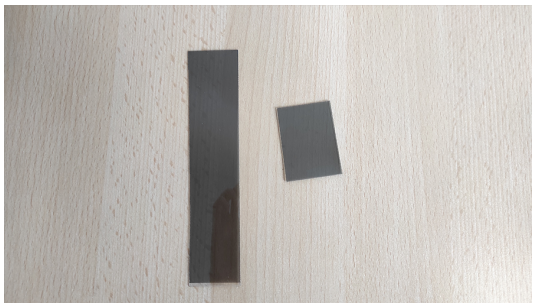
Obr. 3.2: Schéma celkového návrhu

3.2.2 Upevnenie polarizačných filtrov

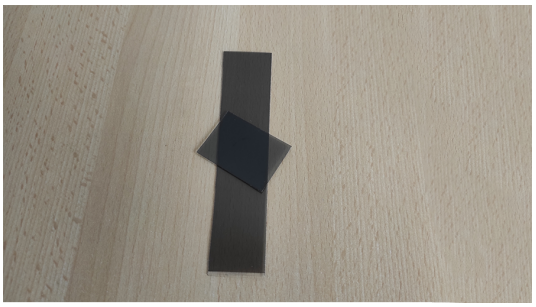
V obraze, ktorý budeme spracovávať nás zaujíma len jedna vec, červený laserový lúč. Zisťovať polohu laserového lúča v obraze nie je až také jednoduché, pretože v obraze sa nachádza veľa svetelného šumu[3] ktoré nám zisťovanie polohy značne komplikuje. Zároveň každá kamera sníma obraz/svetlo inak. Potrebujeme 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[4] prekrížených polarizačných filtrov, zobrazených na Obrázku 3.3b.

[3] nežiadúce okolité svetlo

[4] 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

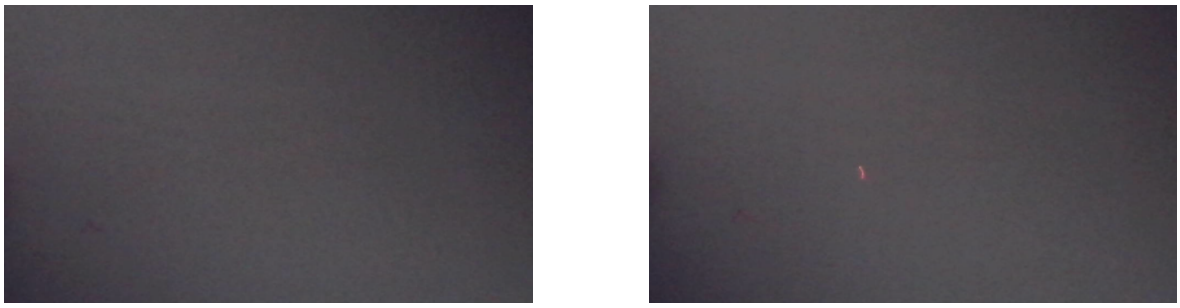
Ako bolo spomenuté v Sekcii 3.1, tieto filtre musia byť priamo na sebe, bez žiadnej medzery medzi nimi. V opačnom prípade by sa to správalo, ako keby tam bol len jeden. Dvojica vhodne prekrížených filtrov zredukuje svetlo, ktoré sa cez nich dostane a keďže svetlo z laserového lúča je dosť silné svetlo, 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[5], premietaný projektorom v obrázkoch 3.5 - 3.8

[5]obrázok FMFI UK, https://navyske.sk/data/images/754.jpg, pristupované 19.3.2023

(a) bez lasera (b) s laserom

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

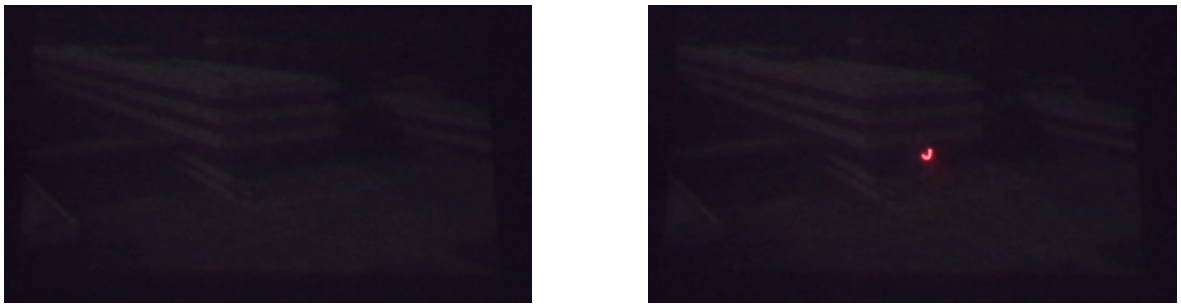


(a) bez lasera (b) s laserom

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

(a) bez lasera (b) s laserom

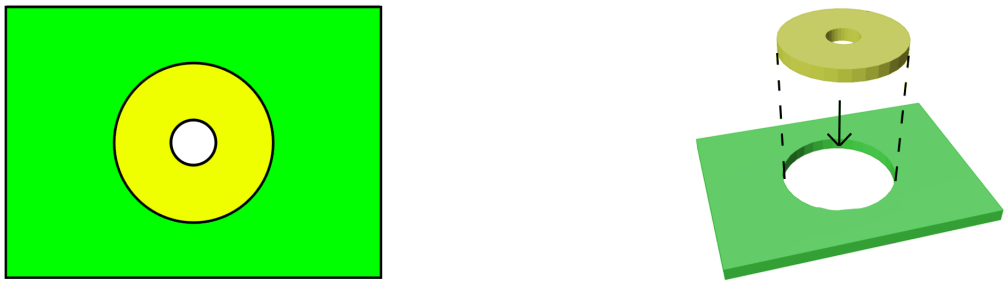
Obr. 3.7: Obraz z kamery bez denného svetla, bez použitia polarizačných filtrov



(a) bez lasera (b) s laserom

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

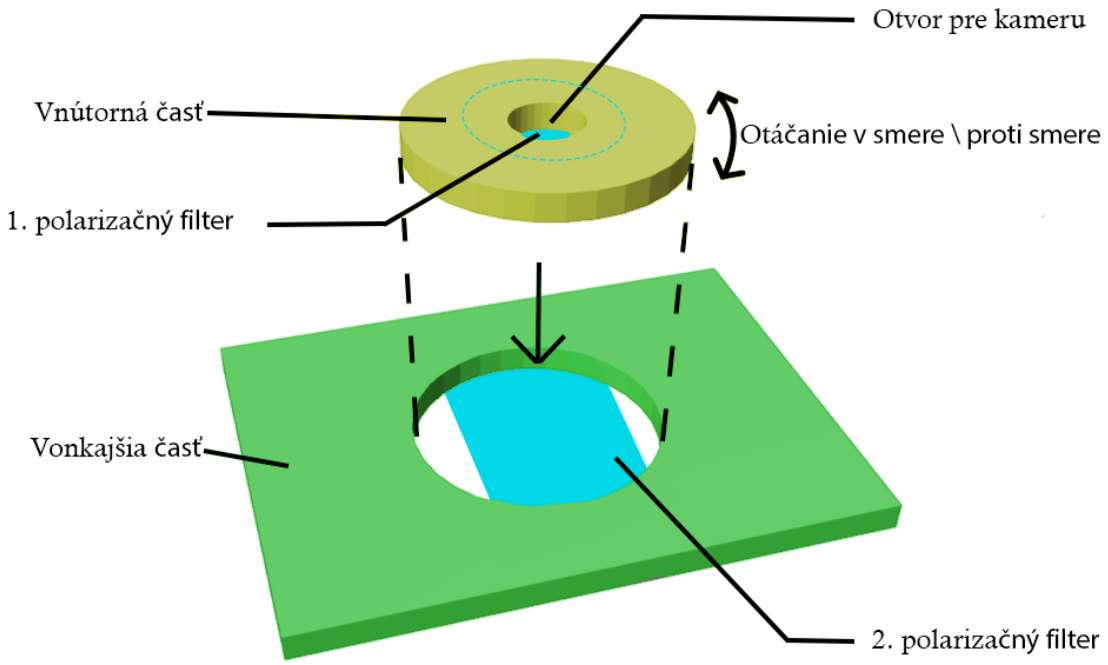
Pred samotným návrhom plugin-u a aplikácie je potrebné vymyslieť a zhotoviť návrh upevnenia dvojice polarizačných filtrov na kameru. Kamery sú rôznych druhov a rôznych veľkostí, preto návrh upevnenia polarizačných filtrov je pre našu prácu veľmi špecifický a nebude ho možné uplatniť pri použití inej kamery, ako používame v našej práci, samotný plugin to však nijako neovplyvní. Keďže nevieme dopredu, aké bude vhodné[6] 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. Vnútorná 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.



(a) pohľad zhora (b) pohľad z perspektívy

Obr. 3.9: Návrh modelu upevnenia polarizačných filtrov

[6] rôzne prostredia si vyžadujú rôzny uhol prekríženia polarizačných filtrov



Obr. 3.10: Popis návrhu mechanizmu upevnenia polarizačných filtrov

3.2.3 Návrh pluginu

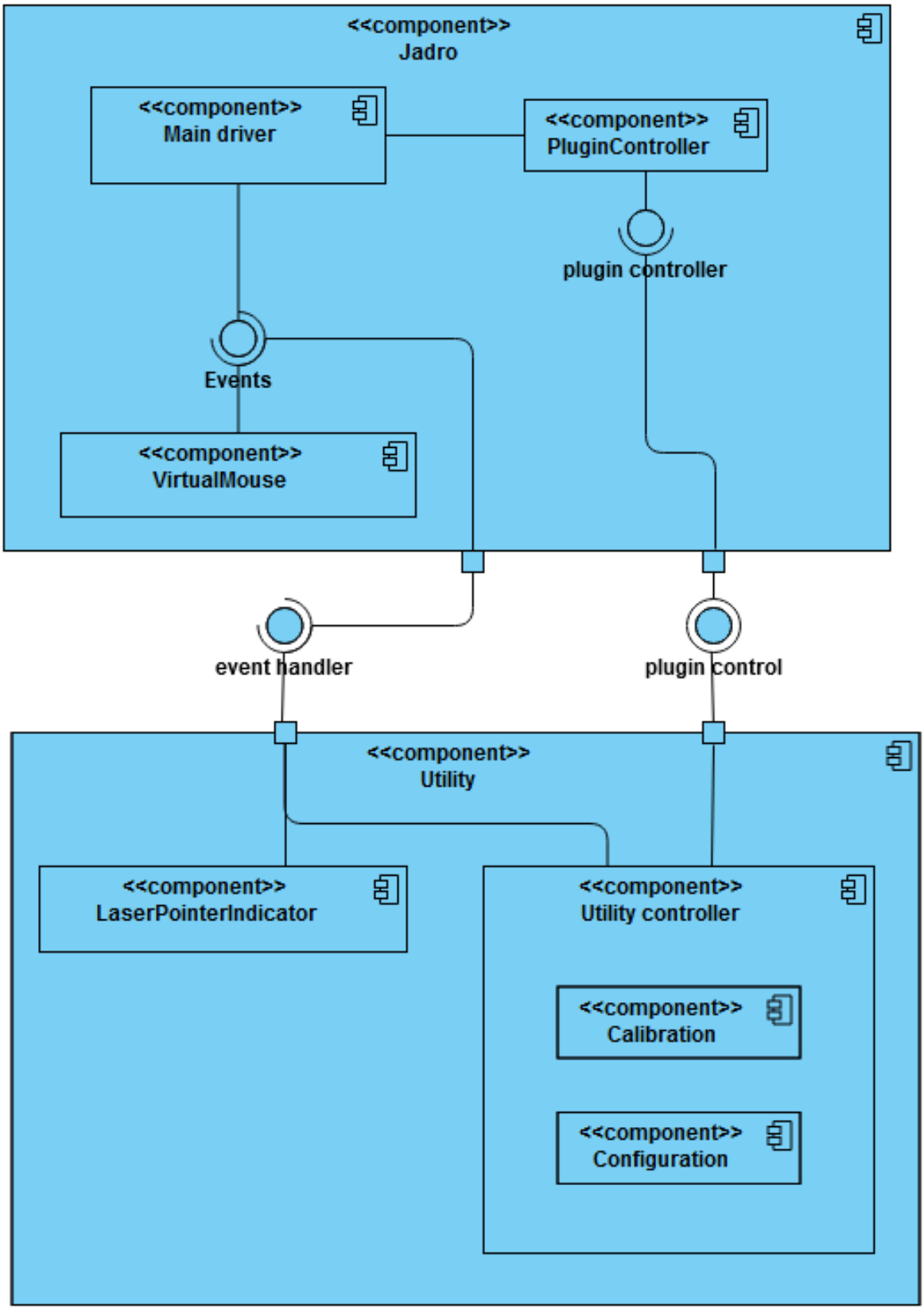
Plugin chceme navrhnúť čo najviac modulárne, aby sa dal v prípade potreby jednoducho rozširovať, respektíve modifikovať. Plugin bude pozostávať z dvoch základných častí, z jadra a z utilít.

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.

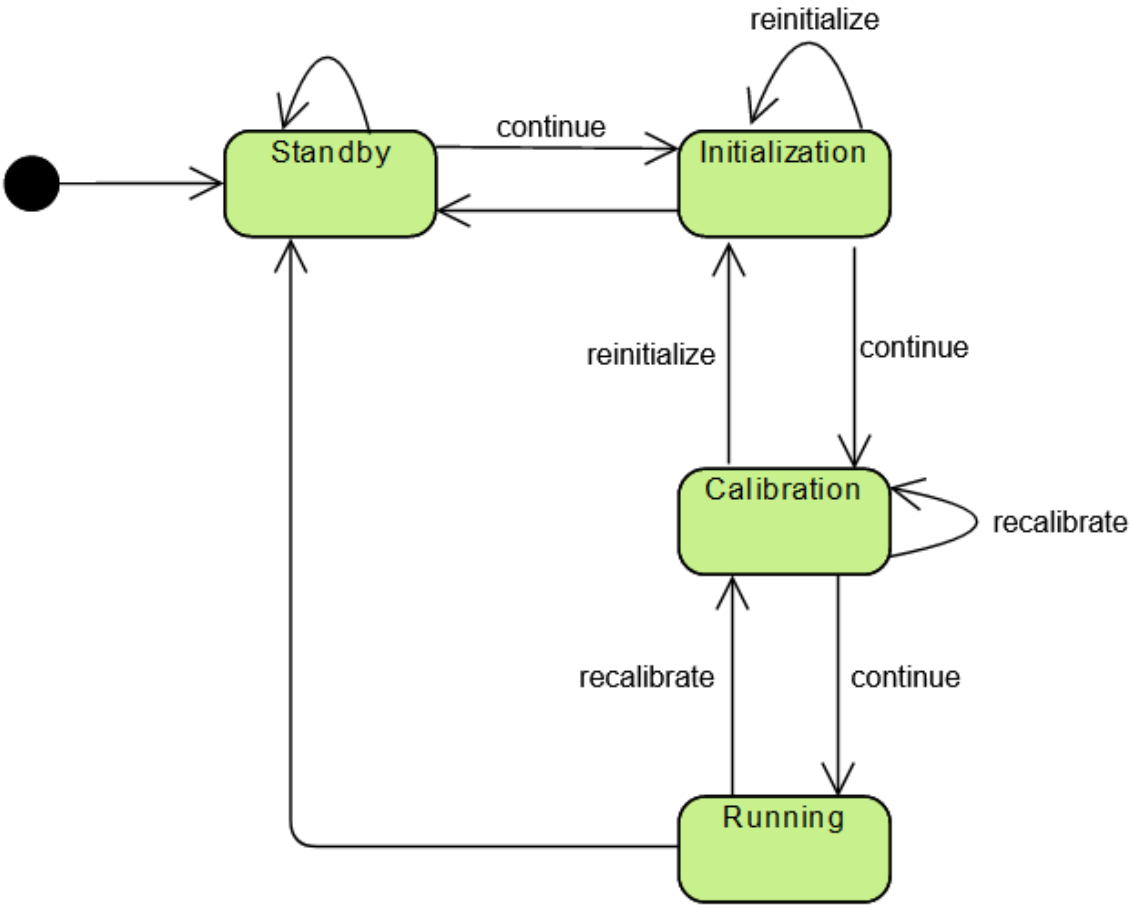
V nasledujúcom odseku čitateľovi bližšie popíšeme návrh automatu pre komponent **main driver** aj s jeho vlastnosťami a prechodmi.



Obr. 3.11: Diagram komponentov 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 sku-

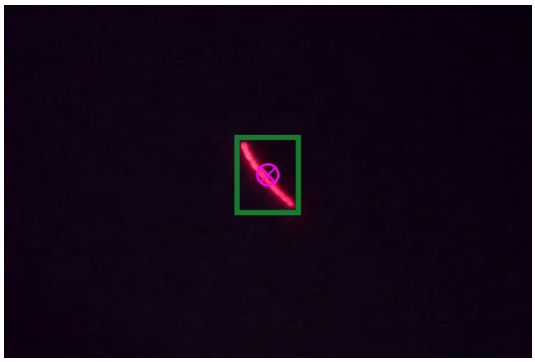
točnosti nič nedeje, ako indikuje meno stavu. Tento stav slúži len ako taka čakáreň. Zo stavu *Standby* sú dovolené len prechody do stavov *Standby* a *Initialization*. Po prechode zo stavu *Standby* do stavu *Initialization* sa nastavia počiatočné parametre pre správne fungovanie plugin-u. Zo stavu *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.



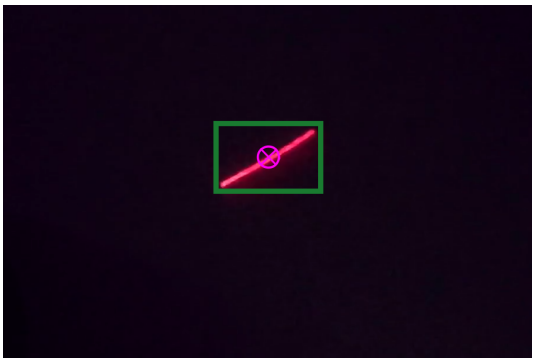
Obr. 3.12: Stavový automat pre main driver

Detekciu laserového lúča sme sa rozhodli, že budeme riešiť tak, že detegovaný laserový lúč ohraničíme obdĺžnikom, ako je vidieť na Obrázkoch 3.13. Ideálny prípad by bol, keby detegovaný laserový lúč bol kruhového tvaru, ako je znázornené na Obrázku 3.13c, no museli

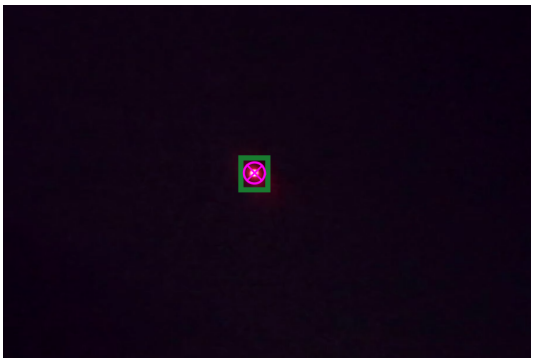
sme sa zamyslieť aj nad prípadmi, ktoré nezdieľajú takéto ideálne črty. Pre takéto prípady sme sa rozhodli použiť rovnaký prístup ako pri ideálnych, t. j. enkapsulácia do obdĺžnika (viď Obrázok 3.13a a 3.13b).



(a) príklad detekcie 1



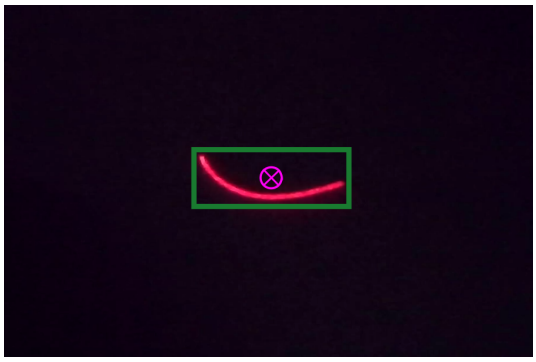
(b) príklad detekcie 2



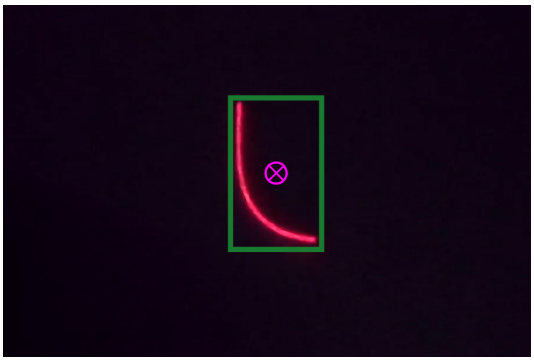
(c) príklad ideálnej situácie

Obr. 3.13: Príklady detekcie lasera v obraze z kamery

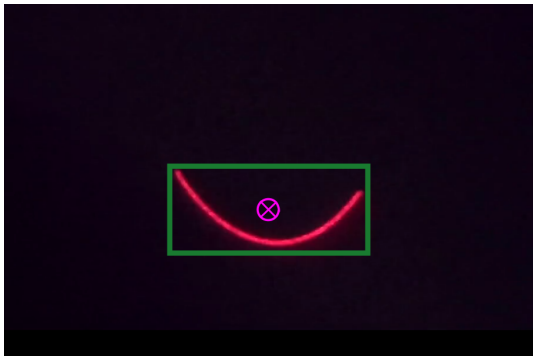
Avšak toto so sebou prináša aj isté anomálie, ktoré sú znázornené na Obrázkoch 3.14, kde by výsledok detekcie nedával moc zmysel. Pravdepodobnosť, že by takáto situácia nastala je však nízka, lebo mieriť presne s laserovým zariadením do diaľky (aspoň pre ľudí) je náročnejšie, ako sa na prvý pohľad môže zdať. Preto osoba s laserovým zariadením sa bude snažiť skôr o presnosť ako rýchly a nezmyselný pohyb vyúsťujúc do spomínaných anomálii. Napríklad v situáciach zobrazených na Obrázkoch 3.14b až 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.



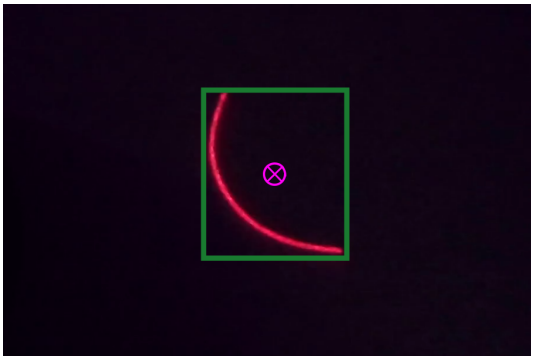
(a) laser tvaru konvexnej paraboly 1



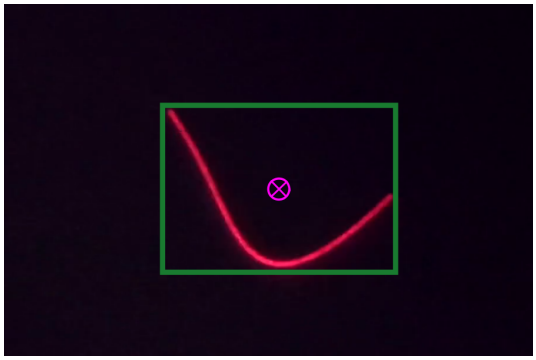
(b) laser tvaru konvexnej paraboly 2



(c) laser tvaru konvexnej paraboly 3

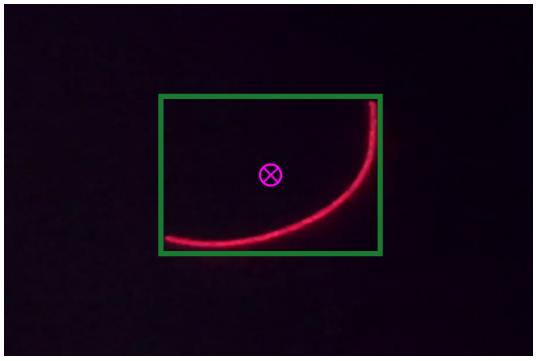


(d) laser tvaru konvexnej paraboly 4

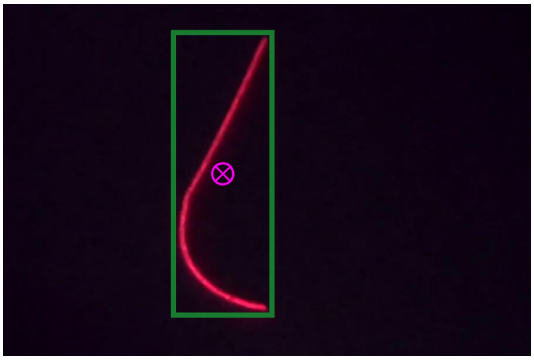


(e) laser tvaru konvexnej paraboly 5

(f) laser tvaru horizontálne preklopeného L



(g) laser tvaru konvexnej paraboly 6



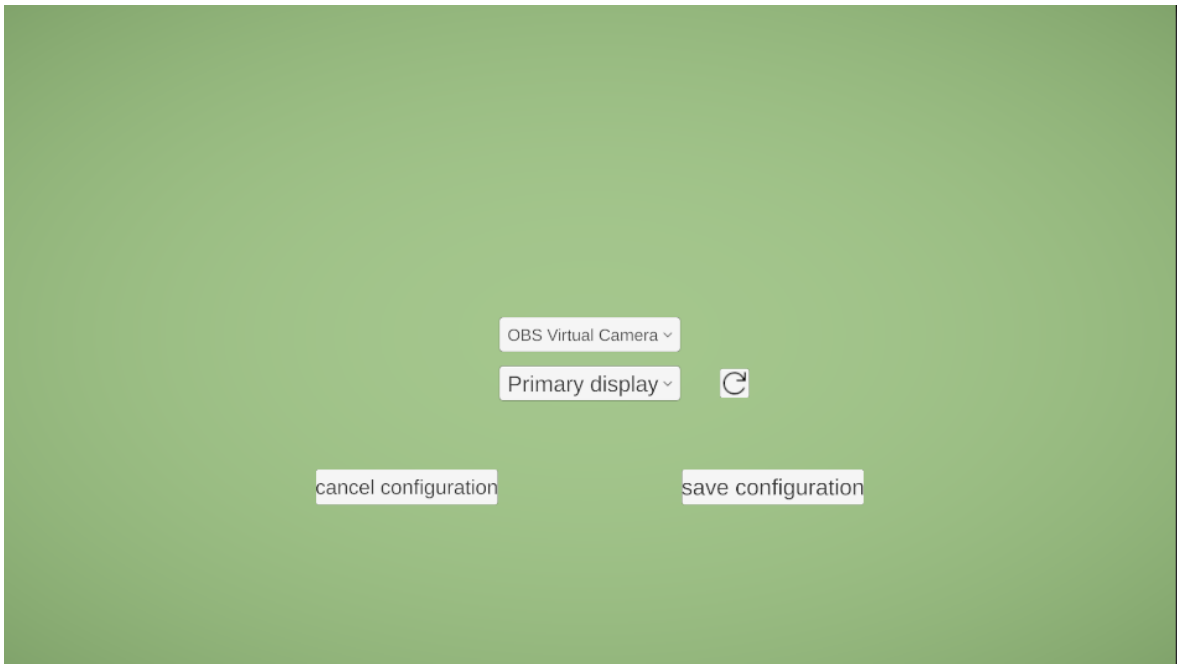
(h) laser tvaru L

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

Druhá časť utility obsahuje doplňujúce 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 bude pozostávať z dvoch tlačidiel typu dropdown. Prvý dropdown obsahuje záznamové zariadenie[7]. Druhý dropdown obsahuje obrazovky v počítači[8]. 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 sme pridali aj tlačidlo pre manuálnu obnovu údajov.



Obr. 3.15: Návrh konfiguračného menu

Utility - kalibračné menu

Kalibračné menu bude pozostávať z troch tlačidiel *exit*, *edit configuration*, *calibrate*. Pomocou tlačidla *exit* budeme schopný zavrieť 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* zostáva nedostupné. Ďalej obsahuje dve tlačidlá typu checkbox. Prvý checkbox *Marker* povoľuje zobrazovanie indikátora laserového lúča na obrazovke. Druhý checkbox *Debug text* povoľuje zobrazovanie používateľského rozhrania pre ladiace informácie.

[7] kameru alebo webkameru

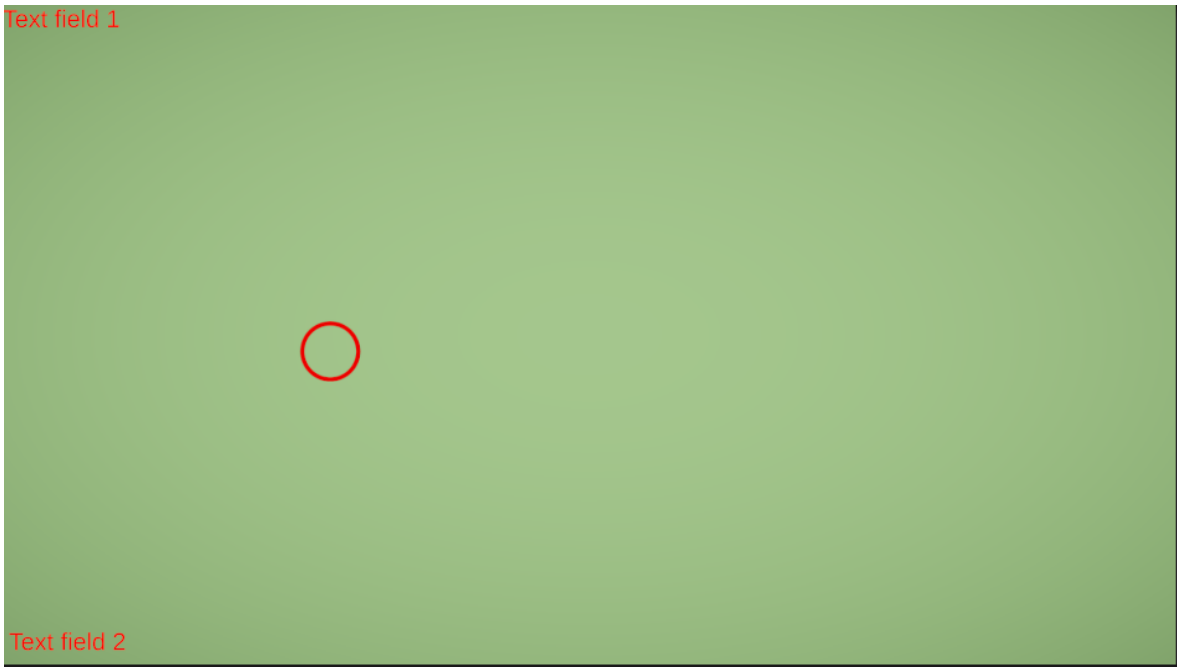
[8] hlavná a externá obrazovka, prípadne ďalšie

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[9] za sekundu. Druhé textové pole zakotvené v ľavom dolnom rohu obsahuje informáciu o simulovanom kliknutí na interpretovaných súradniciach. Ďalej obsahuje kruhový indikátor laserového lúča, ktorý sa zobrazí na pozícii, kde bol detegovaný laserový lúč.

[9] počet snímkov za sekundu záznamového zariadenia, t. j. kamery



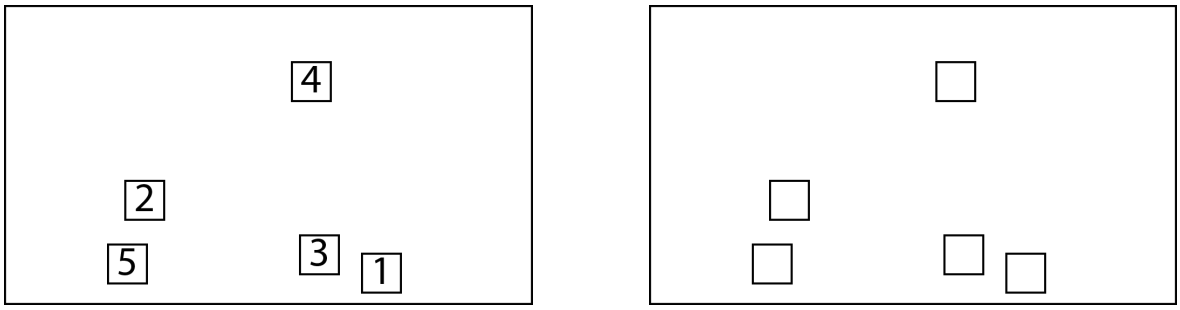
Obr. 3.17: Návrh používateľského rozhrania pre ladiace informácie

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

3.2.4 Návrh aplikácie

Aplikáciu môžeme navrhnúť úplne nezávislú od plugin-u, skoro bez pomyslenia na jeho existenciu, 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.

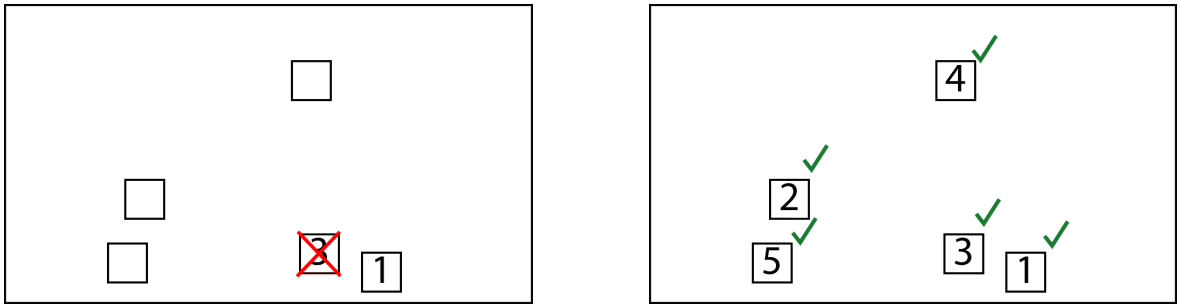
Princíp hry je nasledovný. Hra pozostáva z viacerých úrovní, začína sa na úrovni 1 s tým, že v menu si bude môcť hráč zvoliť štartovaciu úroveň (viď Obrázok 3.21). Každá úroveň pozostáva z N objektov. Úroveň M obsahuje teda M objektov, kde M je nenulové prirodzené číslo. Pri každej úrovni sa na obrazovke náhodne umiestní 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, respektíve 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.



(a) Objekty ukazujú svoje čísla

(b) Objekty skryli svoje čísla

Obr. 3.18: Návrh mechaniky hry - ukáž / skry



(a) Nesprávne zvolenie objektov v poradí

(b) Správne zvolenie objektov v poradí

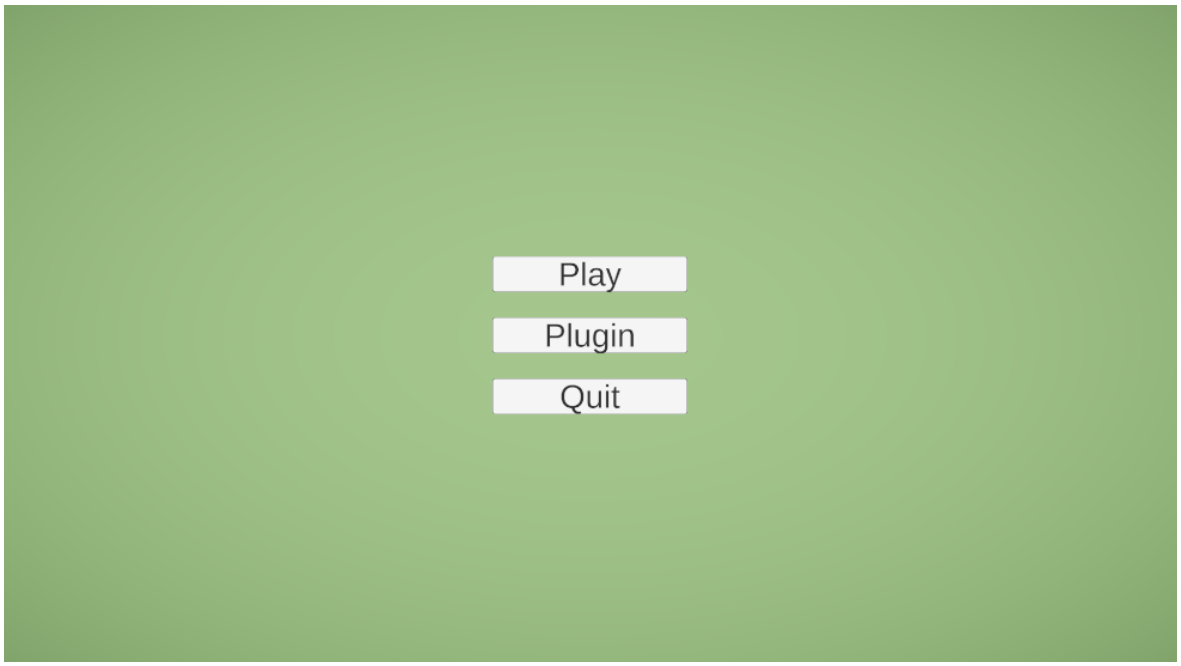
Obr. 3.19: Návrh mechaniky hry - dobrý / zlý

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 dve tlačidlá, *Play* pre začatie hry, respektíve presun do menu výberu úrovne a *Quit* pre ukončenie aplikácie. Nakoniec sme pridali aj tlačilo *Plugin*, ktoré spustí menu plugin-u, keďže chceme integrovať náš plugin do tejto aplikácie, aby sme mali možnosť ho spustiť.



Obr. 3.20: Návrh hlavného menu aplikácie

Návrh menu výberu úrovne

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

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 popíšeme stručne. 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[1], 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 prototyp zobrazený na Obrázku 4.1. Prototyp bol vyrobený z papierovej krabičky z tenkého a tvrdšieho papiera. Obsahoval dva do seba vnorené kruhy s otvorom, ktoré obsahovali polarizačný filter. Vonkajší kruh bol statický. Vnútorný kruh sa otáčal vnútri vonkajšieho kruhu pre prispôsobenie uhla prekríženia polarizačných filtrov. Vo vnútri krabičky 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 používanie prototypu stalo ošemetným a aj materiál bol opotrebovanejší.

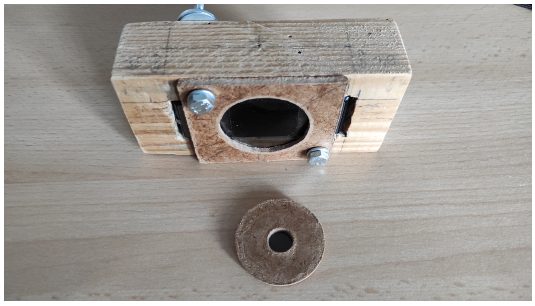
[1] version control system - napr. git

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 funkcionalita polarizačných filtrov bola zachovaná a kamera zostala upevnená na jednom mieste po celú dobu používania. Tento produkt už pokrýval nevýhody prototypu a veľmi ľahko sa s ním pracovalo.

Jediná nevýhoda bola, že sme nemysleli na teplo a kamera by sa mohla prehriať z nedostatku cirkulácie vzduchu, ale tento problém sme počas celej doby nezaznamenali.



(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** [11] sme implementovali stavový automat pre jadro plugin-u. Vytvorili sme si abstraktnú triedu *LPIPBaseState* (viď Kód 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(LPIPCoreManager lpipCoreManager);
4     public abstract void UpdateState();
5     public abstract void ExitState();
6 }
```

Kód 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* (viď Kód 4.2). Navyše vypočíta a uloží aj pomer rozlíšenia, v ktorom je spustená aplikácia ku rozlíšeniu, ktoré dostaneme z kamery. Tento pomer 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 }
```

Kód 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* (viď Kód 4.3).

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

Kód 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 vyvtorených v stavoch *InitializationState* a *ManualCalibrationState*. Počas behu stavu sa vyvoláva udalosť, že sa detegoval laserový lúč a má sa simulovať kliknutie na jeho súradniciach. Podrobnejší popis je rozpísaný v Sekciách 4.2.2 a 4.2.3.

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

Ďalej sme vytvorili triedu *LPIPCoreManager* (viď Kód 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 automatu.

```
1  public class LPIPCoreManager : MonoBehaviour{
2      public LPIPBaseState InitializationState { get; private set; }
3      public LPIPBaseState ManualCalibrationState {get; private set;}
4      public LPIPBaseState RunningState { get; private set; }
5      public LPIPBaseState StandbyState { get; private set; }
6
7      private LPIPBaseState _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();
```

```
        }
    }
}
```

Kód 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* (viď Kód 4.5) pomocou návrhového vzoru **Singleton** [11], ktorá slúži pre programátora alebo iný systém na ovládanie pluginu a nastavenie konfiguračných premenných cez metódy *SetWebCamTexture(...)* a *SetProjectorDisplayId(...)*.

```
public class LPIPCoreController : MonoBehaviour
{
  [SerializeField] private LPIPCoreManager _lpipCoreManager;

  public static LPIPCoreController Instance;
  private bool isWebCamTextureConfigured = false;
  private bool isProjectorIdConfigured = false;

  private void Awake()
  {
      if (Instance == null){
          Instance = this;
      }
  }
  public void InitializeLPIP(){
      if (isWebCamTextureConfigured && isProjectorIdConfigured){
        _lpipCoreManager.StartLPIP();
      }
      else{
          if (!(isWebCamTextureConfigured && isProjectorIdConfigured)){
              Debug.LogError("WebCamTexture and ProjectorId is not
    configured!");
          }
      }
  }
  public void ResetLPIP(){
     _lpipCoreManager.ResetLPIP();
  }
  public void ReCalibrateLPIP(){
      _lpipCoreManager.ReCalibrateLPIP();
  }
  public void SetWebCamTexture(WebCamTexture webCamTexture){
      if (webCamTexture == null){
          Debug.LogError("webCamTexture == NULL");
          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  }
```

Kód 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* (viď Kód 4.6).

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

Kód 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 (viď Kód 4.7).

```
1      private BorderPoint top;
2      private BorderPoint bottom;
3      private BorderPoint left;
4      private BorderPoint right;
```

Kód 4.7: Reprezentácia obdĺžnika ohraničujúceho laserový lúč

Samotná detekcia laserového lúča v obraze pozostávala z jednoduchej rutiny. Z premennej *webCamTexture*, ktorá obsahovala referenciu na objekt kamery sme si získali pole pixlov obrazu z kamery volaním vstavanej Unity metódy (*webCamTexture.GetPixels32()*), ktoré

sme si uložili do premennej *webCamPixels* typu *Color32*. Následne sme pole *webCamPixels* prechádzali cyklom.

Prvé riešenie - intenzita pixlu

V rámci iterácie pixlov sme pre každý pixel vypočítali jeho intenzitu pomocou vzorca [12] a aj jeho 2D súradnice, keďže sa nachádzal v 1D poli a aktualizovali hraničné hodnoty nášho ohraničujúceho obdĺžnika (viď Kód 4.8).

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

Kód 4.8: Pseudokód cyklu - intenzita pixlu

Toto riešenie malo isté nedostatky. Keď do záberu kamery umiestníme objekt, napríklad 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 intenzity 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 snímkov, ktoré obsahovali laserový lúč s aplikovanými polarizačnými filtrami na kamere. Podobne ako v prvom riešení sme si tiež vyrátali 2D súradnice a aktualizovali hraničné hodnoty (viď Kód 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)
```

```
    {
        updateDetectionBorders();
    }

}
```

Kód 4.9: Pseudokód cyklu - 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 tmu, pixle neboli len čierne, respektíve 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. Ľuďom sa to zleje do jednej tmavej farby. Tento problém by sa dal riešiť tak, že obraz by sme predspracovali a vyhli sa tak tomuto efektu. Alebo sa môžme pokúsiť, tak ako v prvom riešení, obmedziť pixle cez ktoré sa cyklí.

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 viď Kód 4.10.

```
private void DetectLaserInFrame(){
    laserDetected = false;
    //Array of pixels
    webCamPixels = webCamTexture.GetPixels32();

    for (int i = 0; i < webCamPixels.Length; i++)
    {
        var pixelLuminance = R_VALUE * webCamPixels[i].r+ G_VALUE *
    webCamPixels[i].g + B_VALUE * webCamPixels[i].b;
        int currentX = i % CAMERA_WIDTH;
        int currentY = i / CAMERA_WIDTH;
        if (pixelLuminance > MIN_PIXEL_LUMINANCE)
        {
            if(webCamPixels[i].r > 120 && webCamPixels[i].g < 189  &&
    webCamPixels[i].b < 170)
            {
                // update laser encapsulating rectangle borders
                updateDetectionBorders();
                laserDetected = true;
            }
        }
    }
```

```
21
22      if(laserDetected){
23          //invoke left mouse button click simulation with detected
        coordinations
24           SimulateLeftClick();
25      }
26  }
```

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

Všetky tieto tri riešenia zdieľali spoločný problém. Bežali výhradne na procesore, čo malo negatívny dopad 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 mohli odľahčiť prácu CPU a zapojiť do práce GPU. Zmenili sme dátovú štruktúru reprezentujúcu obdĺžnik ohraničujúci laserový lúč (viď Kód 4.11). Tentoraz nám táto štruktúra stačila na reprezentáciu celého obdĺžnika 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  };
```

Kód 4.11: Zmena štruktúry obdĺžnika ohraničujúceho laserový lúč

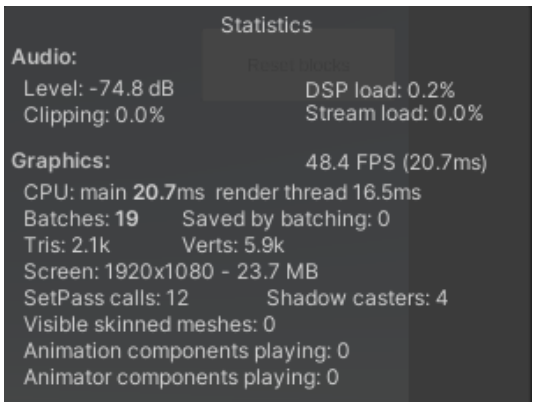
Teraz výpočet na GPU prebiehal tak, že pre každý pixel sa zistilo, či je v rámci povolených hodnôt R, G a B. Intenzitu pixlu sme nevyužili, ani nepoužili. Potom sme skontrolovali, či sme nevyskočili za hranice textúry, keďže čítanie mimo rozsah pri textúrach je povolené, ale vracia východziu hodnotu, čo by nám skreslilo výsledok a aktualizovali hranice obdĺžnika pomocou metód *InterlockedMin(...)* a *InterlockedMax(...)*[2]. Potom už stačilo prevziať si výsledok výpočtu na CPU a zavolať metódu na simuláciu kliknutia, v prípade ak sa laserový lúč podarilo detegovať.

Porovnanie výkonov

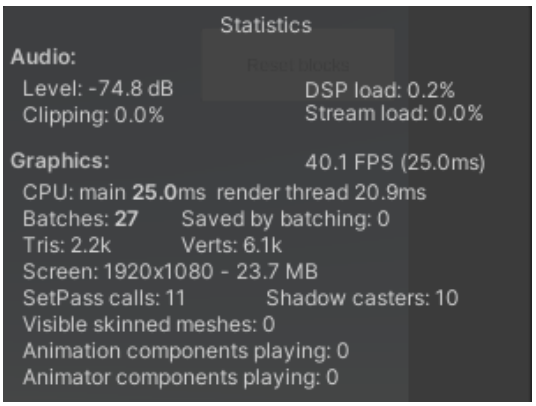
Keďže v prvých troch riešeniach sa všetko robilo výhradne cez CPU, celkový výkon bol výrazne ovplyvnený a tým pádom prázdna aplikácia čisto s plugin-om bežala priemerne na 45 fps s frametime-om okolo 24 ms. Keď sme spravili štvrté riešenie s pomocou GPU, tak výkon prázdnej aplikácie čisto s plugin-om bol priemerne 100 fps s frametime-om 9 ms. Na

[2] špeciálne funkcie, pretože môžu nastať race conditions

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). Kamera poskytovala obraz v rozlíšení 1920 x 1080, v tomto rozlíšení sa spracovával aj obraz. Výkon v Unity editore sa môže líšiť od výkonu skompilovanej verzie. Spomínané grafy a štatistky sú z Unity editor-a. Beh aplikácie bol uskutočnený v Unity edtior-e.

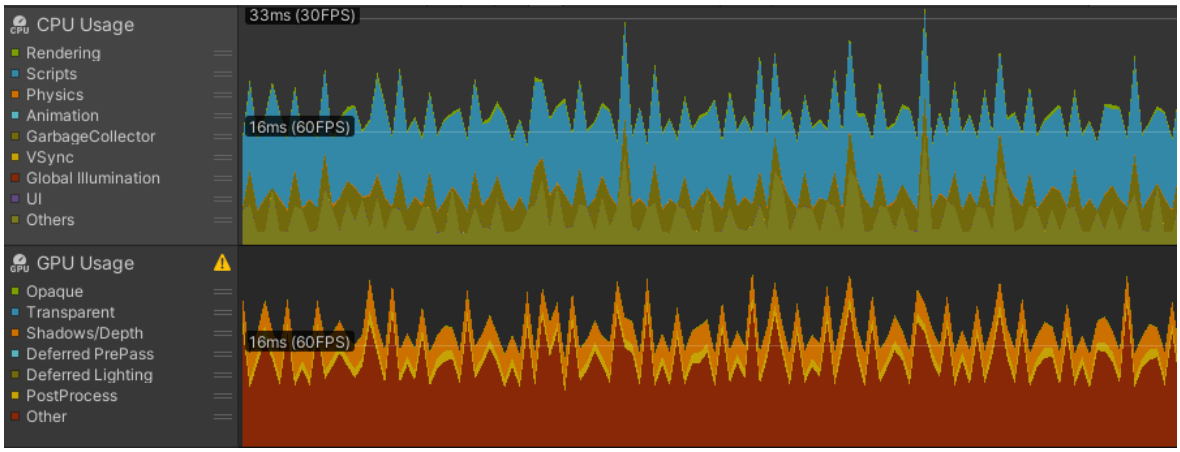


(a) Konfigurácia (A)

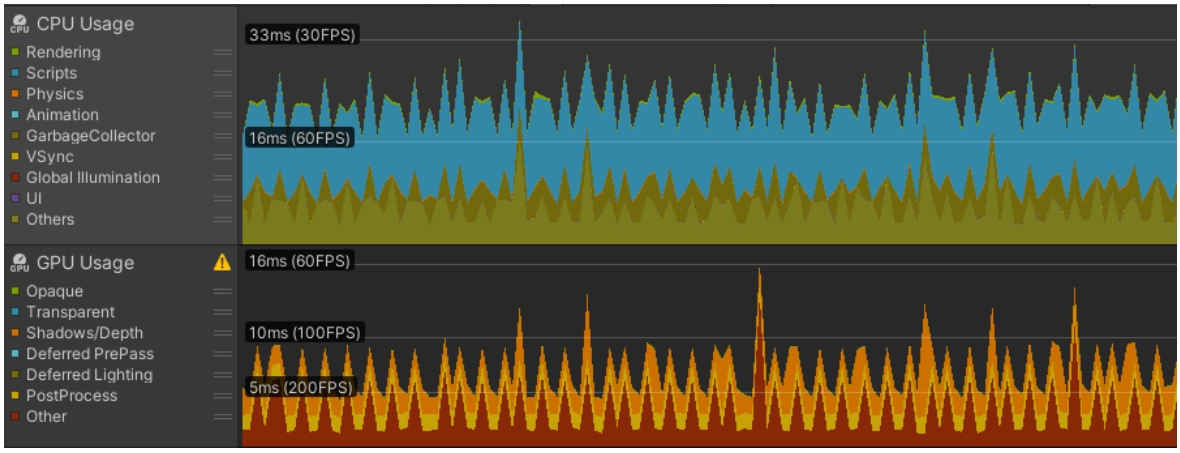


(b) Konfigurácia (B)

Obr. 4.3: Štatistiky zobrazujúce výkon tretieho riešenia 4.2.2

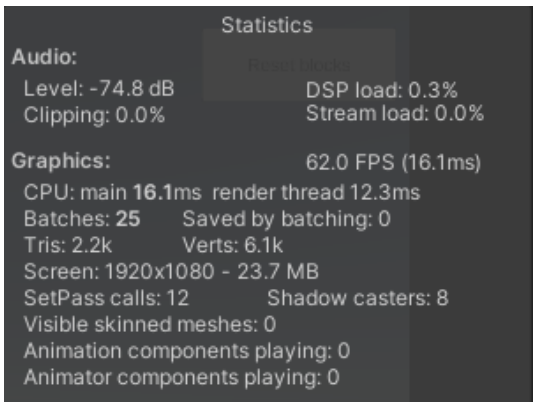


(a) Konfigurácia (A)

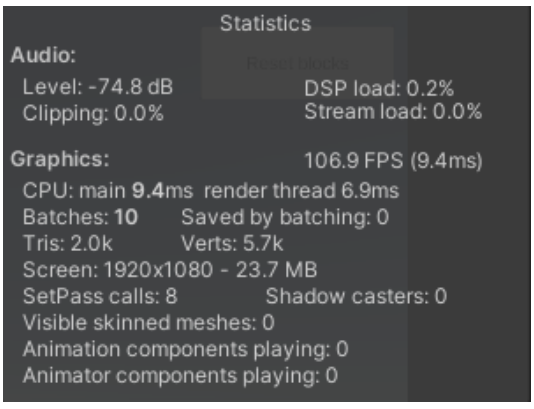


(b) Konfigurácia (B)

Obr. 4.4: Grafy zobrazujúce výkon tretieho riešenia 4.2.2

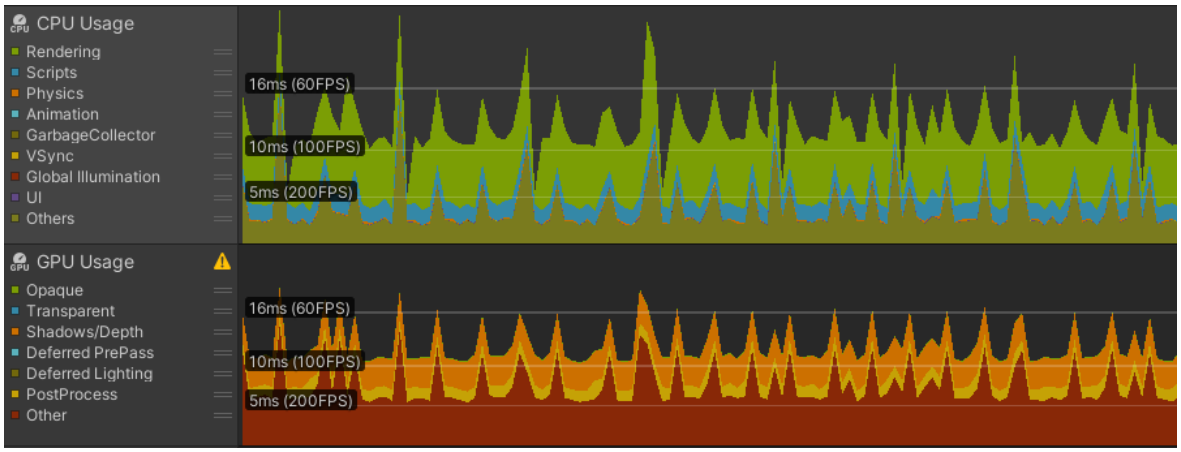


(a) Konfigurácia (A)

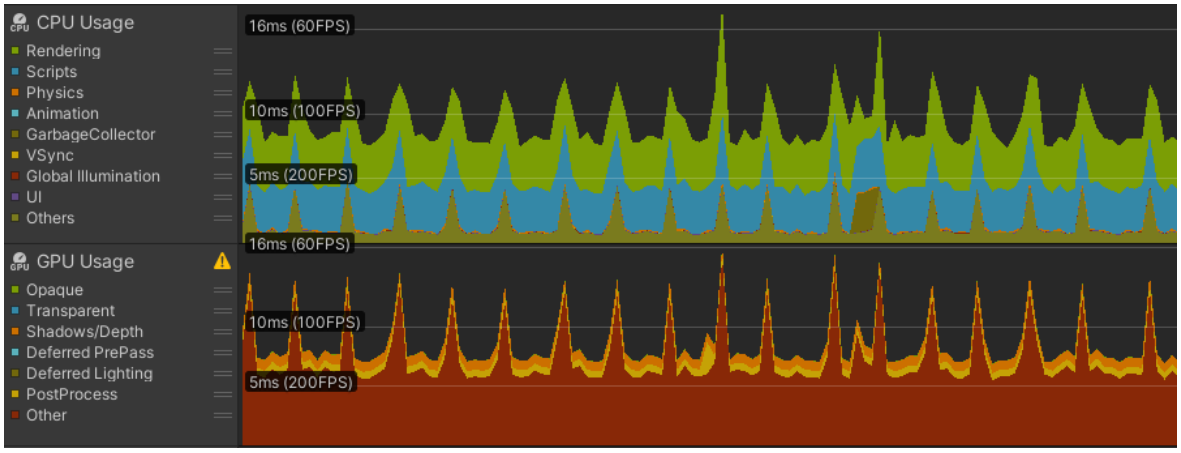


(b) Konfigurácia (B)

Obr. 4.5: Štatistiky zobrazujúce výkon štvrtého riešenia 4.2.2



(a) Konfigurácia (A)



(b) Konfigurácia (B)

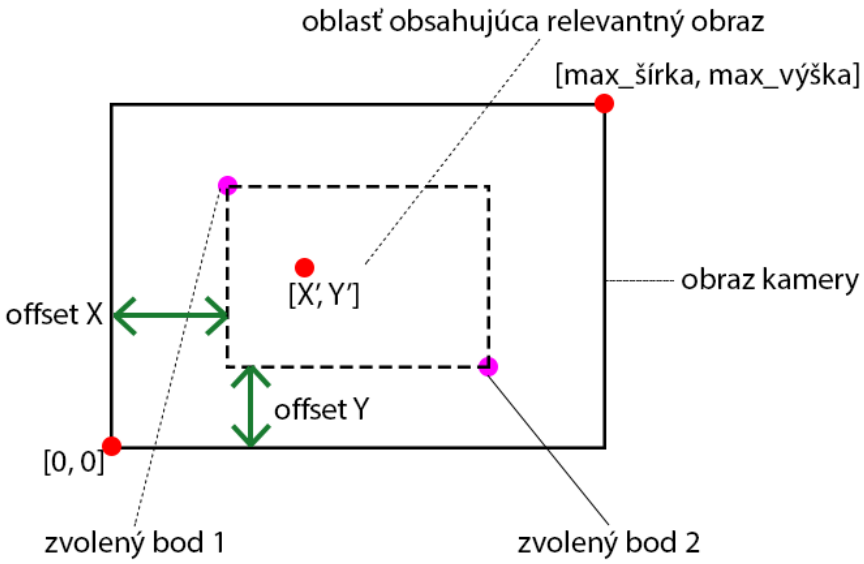
Obr. 4.6: Grafy zobrazujúce výkon štvrtého riešenia 4.2.2

4.2.3 Implementácia kalibrácie a transformácie súradníc

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 Sekcií 4.2.1, v odseku o *InitializationState*.

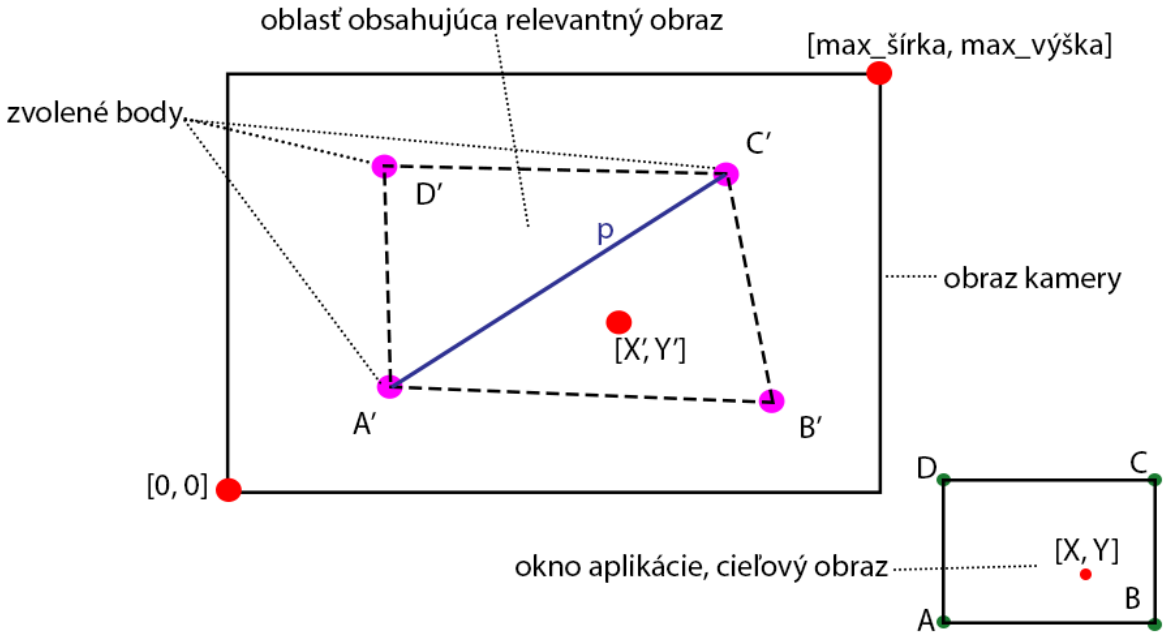


Obr. 4.7: Kalibrácia pomocou 2 bodov

Transformácia súradníc vyzerala potom nasledovne. Bod $[X', Y']$ reprezentuje 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 sú $[X, Y] = [(X' + offsetX) * factorX, (Y' + offsetY) * factorY]$, kde $factorX$ bol pomer šírky obrazu z kamery ku šírke oblasti obsahujúcej relevantný obraz, $factorY$ obdobne.

Druhé riešenie - kalibrácia cez 4 body a trojuholníky

Druhé riešenie bola kalibrácia pomocou štyroch bodov. Podobne ako pri prvom riešení sa na obrazovke 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 [13], 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 prislúchajúcemu trojuholníku ABC za pomoci trojuholníka $A'B'C'$, kde sa nachádza bod $[X',Y']$.

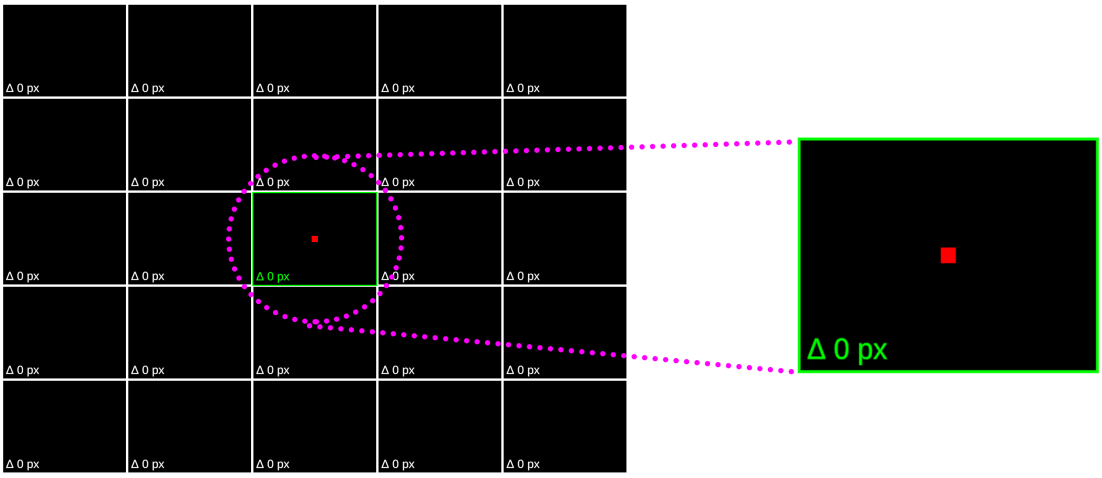
Tretie riešenie - kalibrácia cez 4 body s nelineárnou transformáciou súradníc

V tretom riešení kalibrácia prebiehala úplne rovnako ako v druhom riešení. Jedine sme zmenili algoritmus transformácie súradníc. V tretom riešení sme použili algoritmus nelineárnej transformácie súradníc [14].

4.2.4 Výsledky implementácie kalibrácie a transformácie 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 $m,n \in \mathcal{N}$. 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 Δ <*číslo*> *px*. Pre testovanie presnosti sme spustili 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

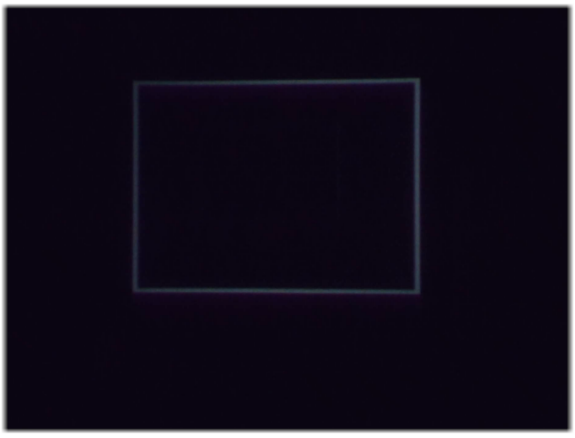
zvolenej bunke sme laserovým lúčom mierili priamo na stred červeného štvorčeka v danej bunke. Následne sme vypočítali rozdiel medzi transformovanými súradnicami a súradnicami stredu malého červeného štvorčeka zvolenej bunky a zapísali hodnotu do prislúchajúceho textového poľa. Na záver, keď sme prešli všetkými bunkami sme vypočítali výberový priemer $\bar{y}$ a rozptyl výberového priemeru s^2. Merania sme vykonali pri štyroch rôznych scenároch s použitím riešení zo Sekcie 4.2.3.



Obr. 4.9: Nástroj na testovanie presnosti

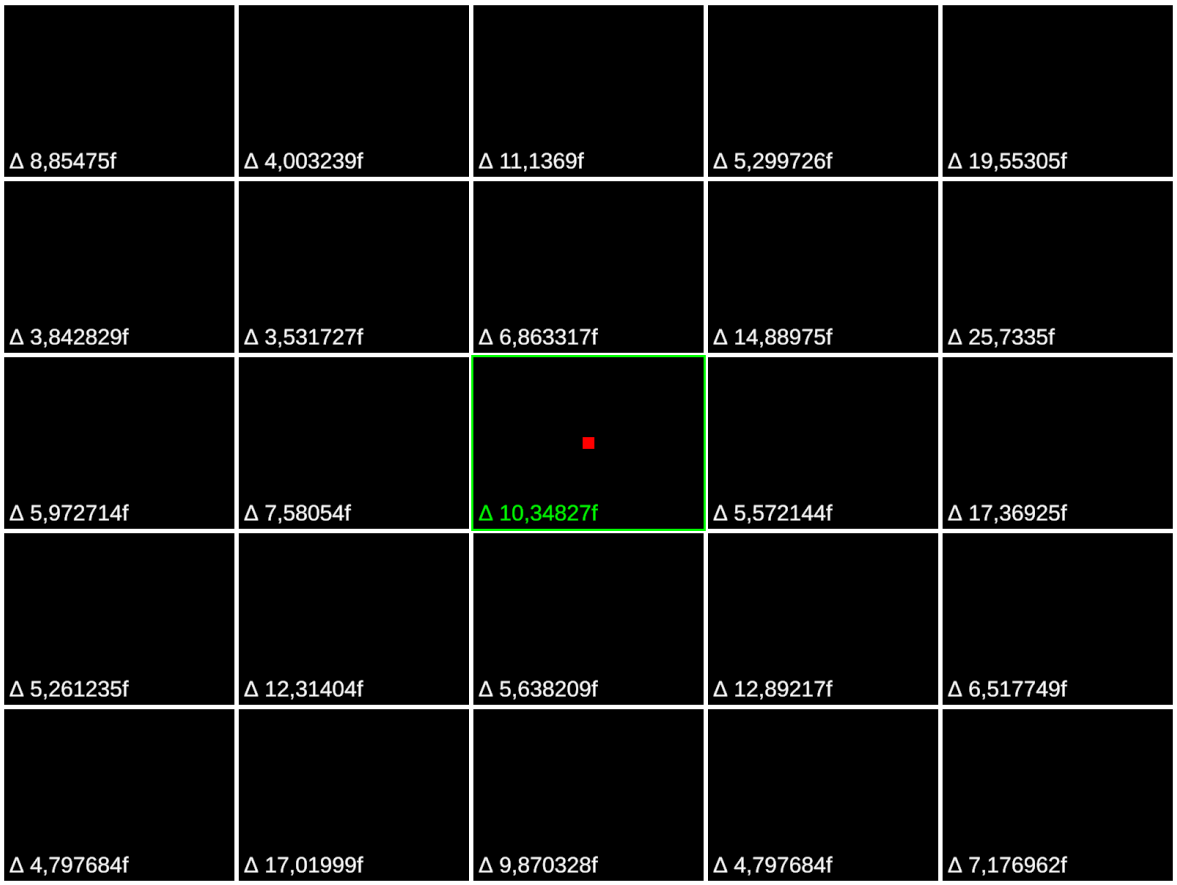
Prvý testovací scenár

V prvom testovacom scenári sme umiestnili kameru rovnobežne so stenou. Obraz, ktorý snímala kamera bol zarovno 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

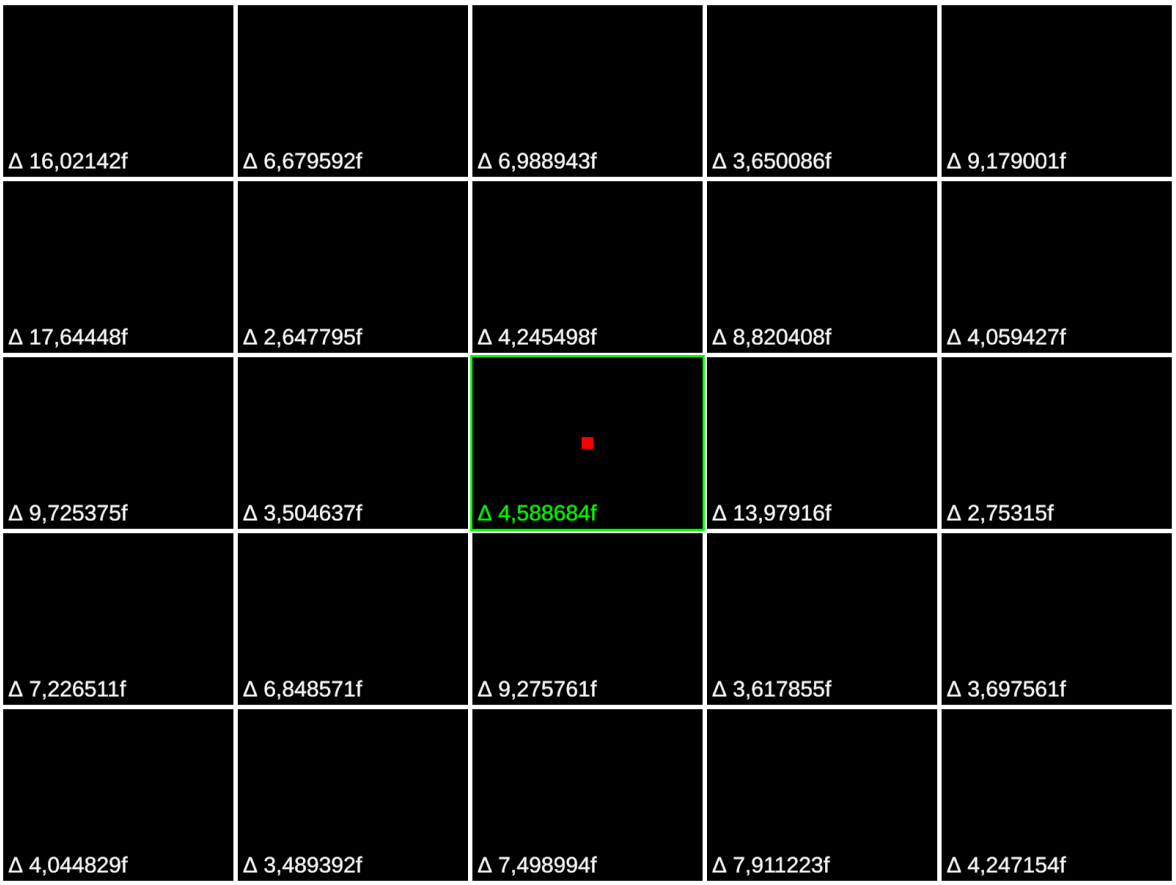
Pri **prvom riešení** sme namerali výberový priemer $\bar{y} = 9,4735$ a rozptyl výberového priemeru $s^2 = 32,5357$. Výsledky prvého riešenia pri prvom testovacom scenári sú na Obrázku 4.11.



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.

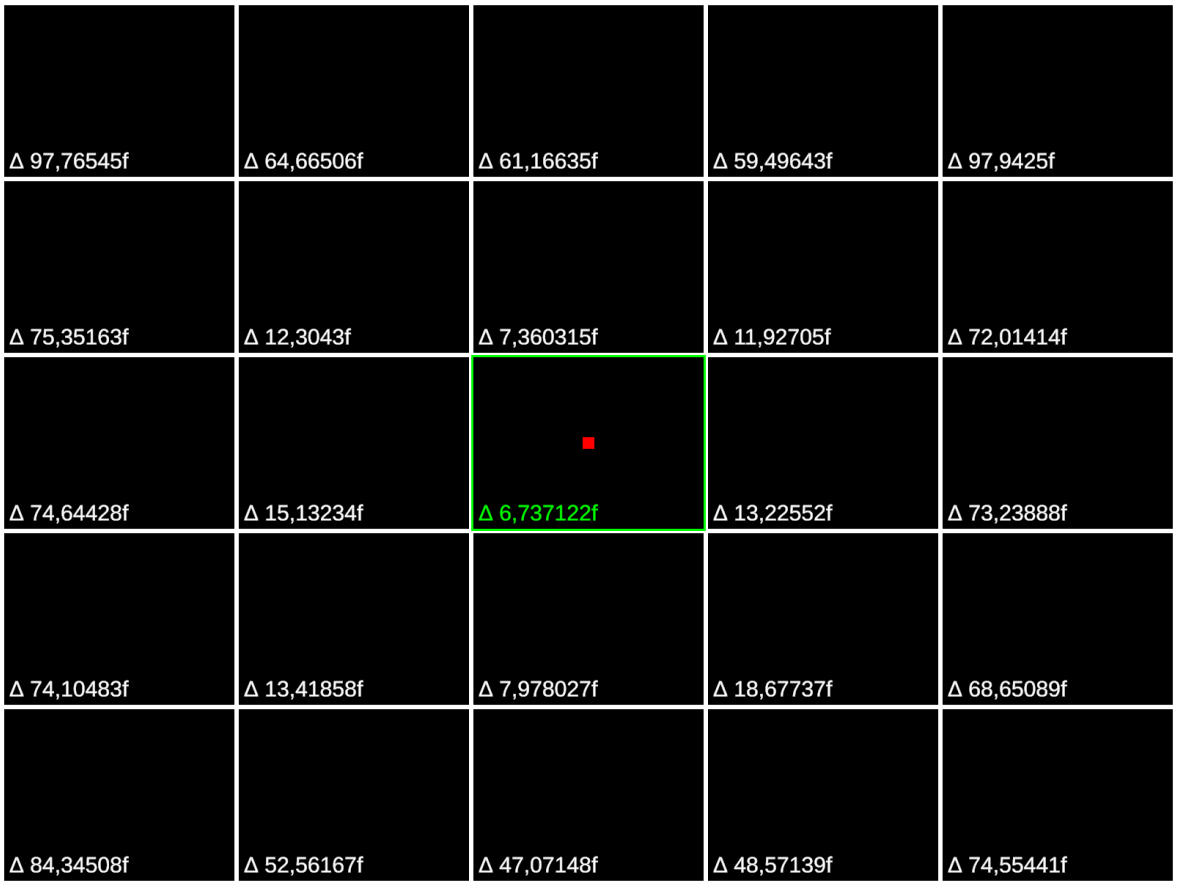
Pri **druhom riešení** sme namerali výberový priemer $\bar{y} = 6,8938$ a rozptyl výberového priemeru $s^2 = 16,6037$. Výsledky druhého riešenia pri prvom testovacom scenári sú na Obrázku 4.12.



Obr. 4.12: Výsledky druhého riešenia pri prvom testovacom scenári

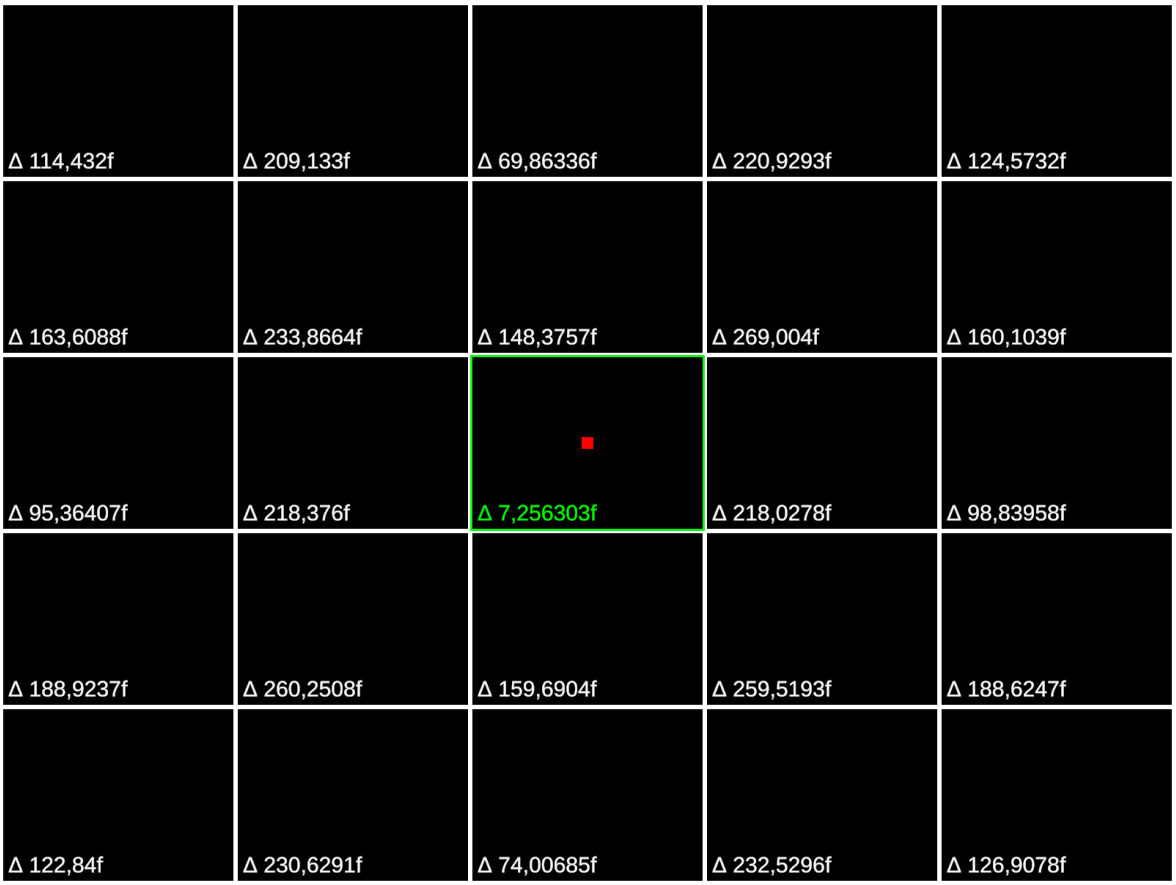
Presnosť **druhého riešenia** je podobne dosť presná a na každej bunke rovnomerná. Dokonca, v tomto testovacom scenári je druhé riešenie presnejšie ako prvé riešenie.

Pri **tret'om 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.



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 vzďaľovaním 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.



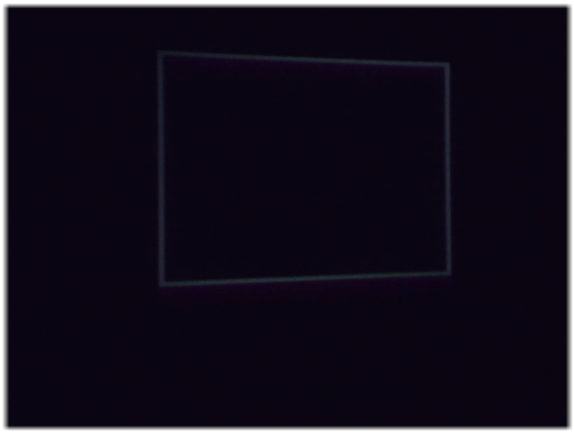
Obr. 4.14: Výsledky tretieho riešenia s parametrom $d = 0,5$ pri prvom testovacom scenári

Všimnime si, že presnosť sa výrazne zhoršila. Čím je parameter d bližšie k číslu 1, tým je tretie riešenie presnejšie, ale len v blízkosti stredu. V rohoch je dosť nepresné.

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

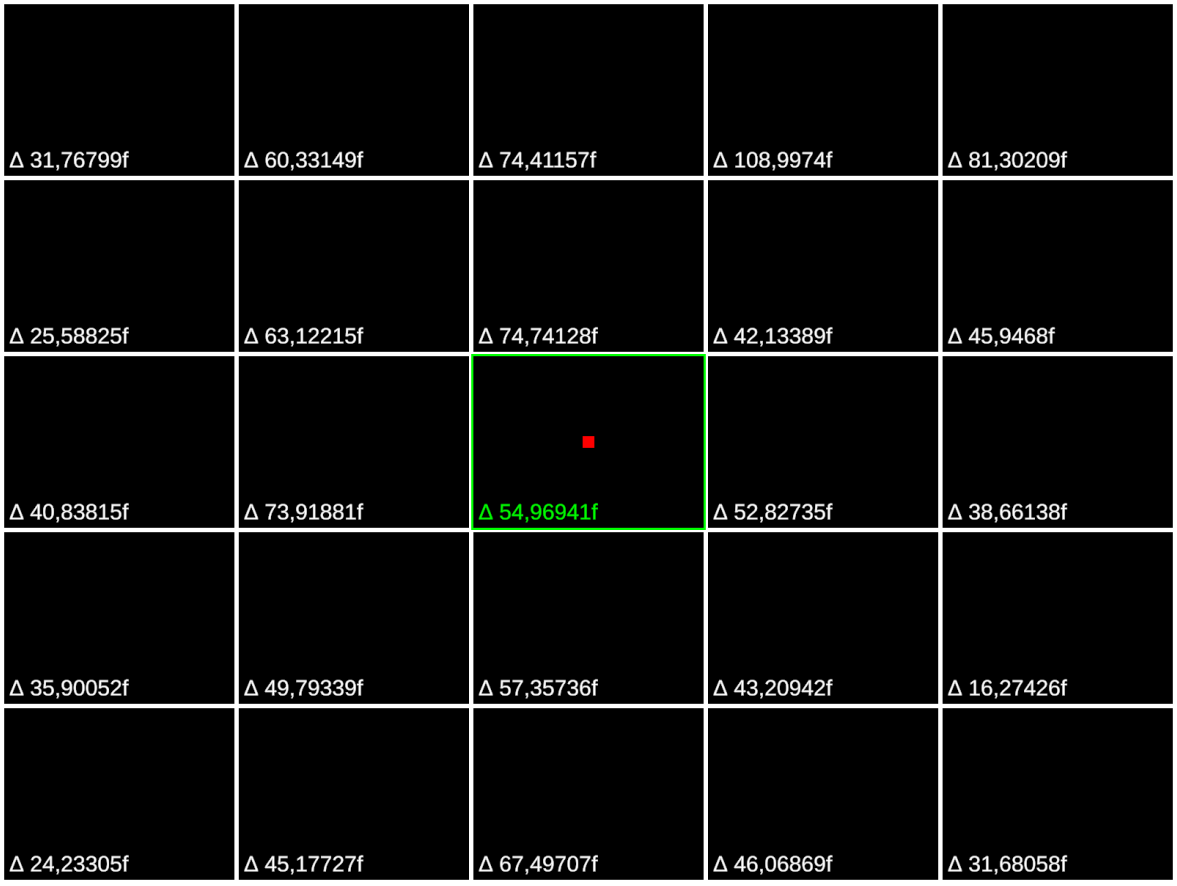
Druhý testovací scenár

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



Obr. 4.15: Obraz 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.

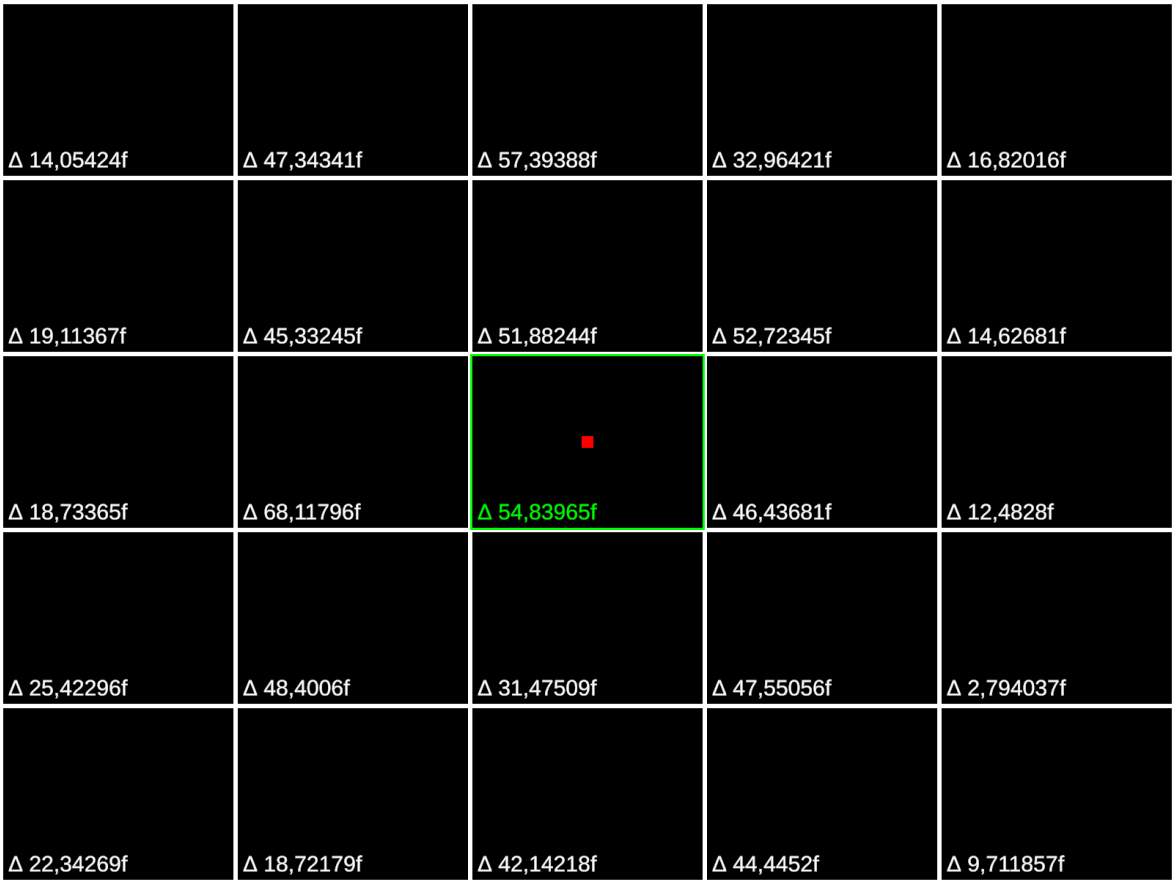


Obr. 4.16: Výsledky prvého riešenia pri druhom testovacom scenári

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

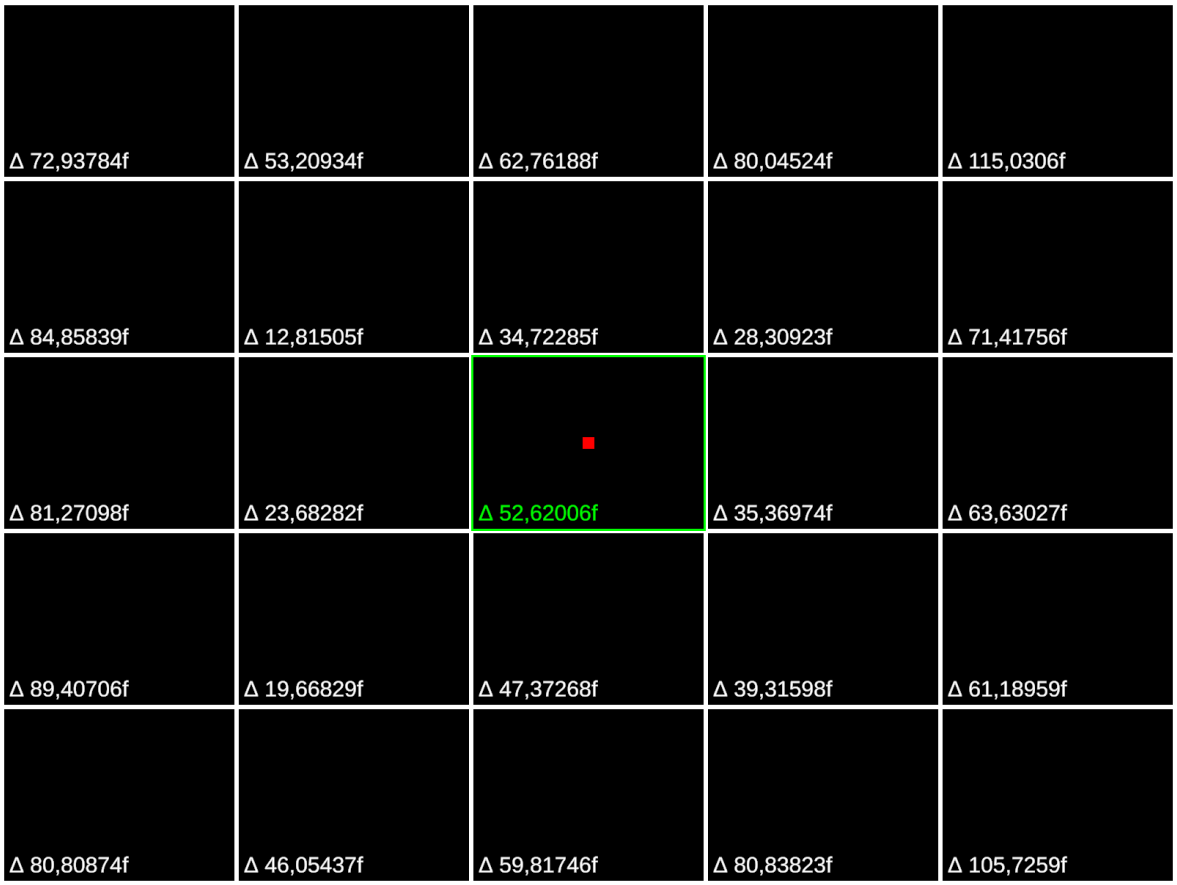
priemeru $s^2 = 326,7840$. Výsledky druhého riešenia pri druhom testovacom scenári zobrazuje Obrázok 4.17.



Obr. 4.17: Výsledky druhého riešenia pri druhom testovacom scenári

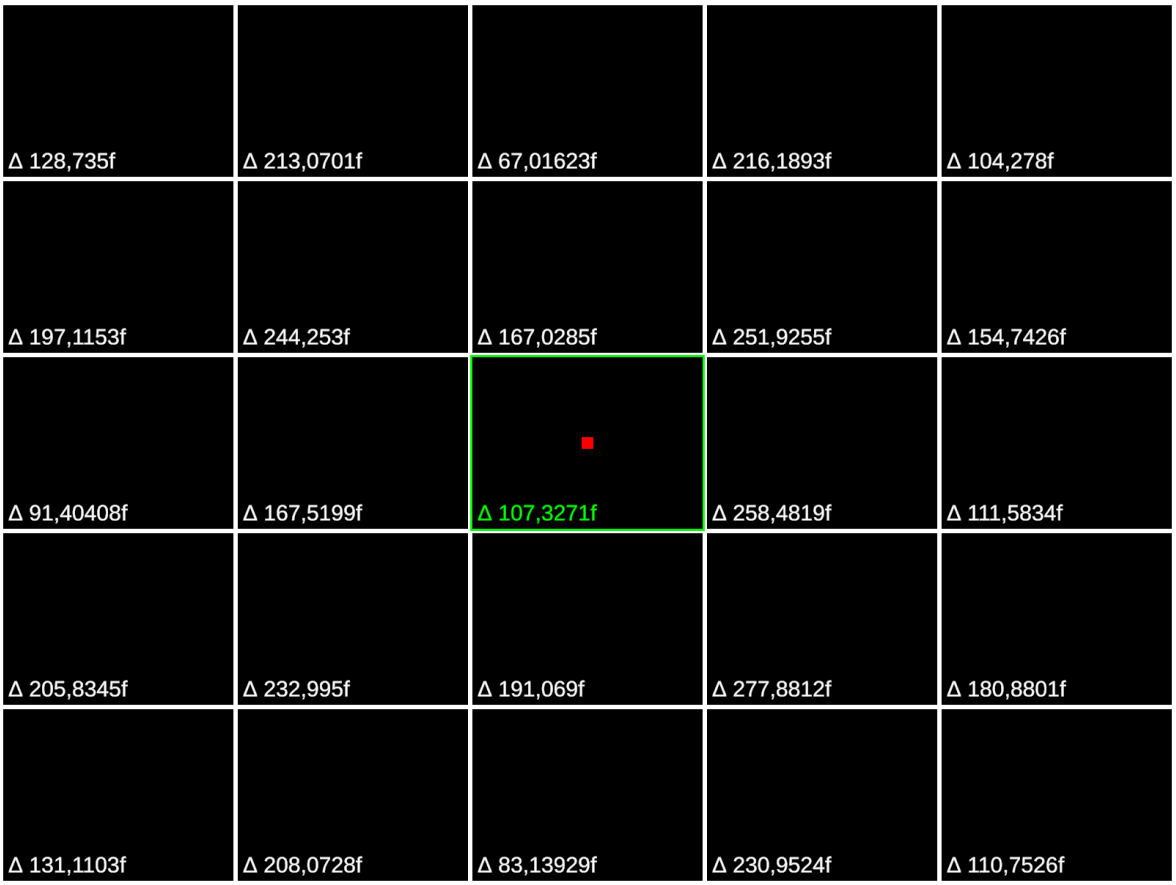
Presnosť **druhého riešenia** nie je až taká dobrá ako v prvom testovacom scenári, ale je lepšia ako presnosť prvého riešenia v druhom testovacom scenári. Druhé riešenie sa lepšie vysporiada so zmenou perspektívy.

Pri **treťom 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.



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

Môžme pozorovať, že presnosť pri **treťom riešení** sa so vzďaľovaním 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.

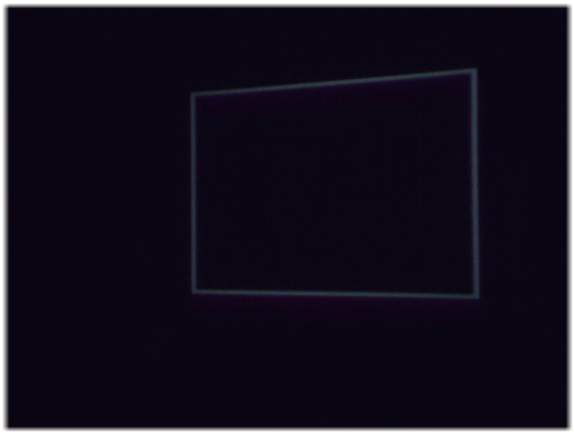


Obr. 4.19: Výsledky tretieho riešenia s parametrom $d = 0,5$ pri druhom testovacom scenári

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 z pravej strany (viď Obrázok 4.20).



Obr. 4.20: Obraz 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.

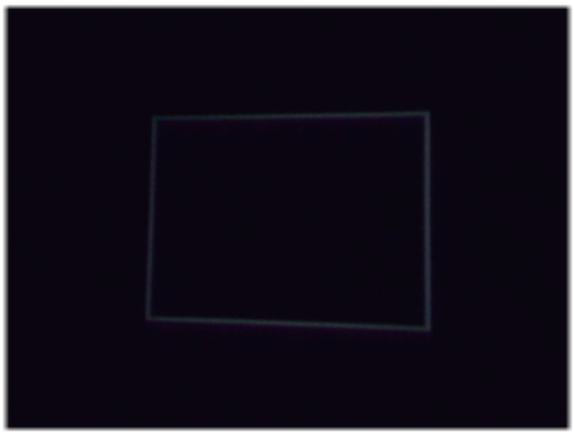
Pri **druhom riešení** sme namerali výberový priemer $\bar{y} = 37,9524$ a rozptyl výberového priemeru $s^2 = 311,3026$.

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 vzďaľovaním od stredu 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 podobne 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: Obraz 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.

Záver

Zo všetkých štyroch scenárov dopadlo najlepšie **tretie riešenie** (viď Tabuľku 4.1), ktoré vykazovalo dostatočne dobrú presnosť za rôznych podmienok.

	riešenie č.	výberový priemer (px)	rozptyl výberového priemeru (px)
Scenár 1	1	9,4735	32,5357
	2	6,8938	16,6037
	3 (d = 0.9)	49,3162	967,2986
	3 (d = 0,5)	167,8270	4720,4770
Scenár 2	1	51,4699	437,4304
	2	33,8349	326,7840
	3 (d = 0.9)	60,1152	706,1392
	3 (d = 0,5)	173,3343	3753,2810
Scenár 3	1	51,8861	323,4306
	2	37,9524	311,3026
	3 (d = 0.9)	62,5213	594,8466
	3 (d = 0,5)	172,6442	4466,6510
Scenár 4	1	31,0753	266,8666
	2	13,5799	42,2760
	3 (d = 0.9)	53,46875	892,8430
	3 (d = 0,5)	170,4061	4127,8180

Tabuľka 4.1: Tabuľka výsledkov presností riešení 1 až 3.

4.2.5 Implementácia simulovania kliknutia ľavým tlačidlom myši

K implementácií simulovania kliknutia ľavým tlačidlom myši sa dalo pristupovať viacerými spôsobmi. Pre nás zaujímavé spôsoby sme si odskúšali. Síce niektoré prístupy fungovali dobre, ale pre nás neboli vyhovujúce. Tieto prístupy spomenieme spolu 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* [15] (viď Kód 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
      , 0, 0);
9     }
```

Kód 4.12: Simulácia kliknutia cez Windows API

Vyskytli 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 <1920, 3840> a Y-ové súradnice by v tomto prípade boli z intervalu <0, 1080>.

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* [16], 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 trošku 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 iterjeme 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* (viď Kód 4.13).

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

Kód 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 súradnic *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[3] 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 (viď Kód 4.14), čo nepredstavuje až taký problém, lebo najčastejšie UI objekty sú button-y, dropdown-y a chekbox-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;
```

[3]komponentami sa v Unity nazývajú jednotlivé skripty, ktoré sa priradia objektom v scéne

```
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  }
```

Kód 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 a 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*. [17]

Pre programátora alebo iný systém sme vytvorili triedu *LPIPUtilityPortal* podľa návrhového vzoru **Singleton** [11], 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[4] doprogramovať.

4.3 Aplikácia

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

[4] inerface *LPIPIInteractable* pre interakciu objektov s pluginom

4.3.1 Implementácia hlavného menu

Hlavné menu aplikácie sme implementovali podľa návrhu. O funkcionalitu hlavného menu sa stará komponent *UIController*. *UIController* komunikuje so sub-komponentami *GamePlayUIController* a *LevelMenuController*.

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

LevelMenuController slúži na ovládanie menu výberu úrovne, zabezpečuje funkcionalitu 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. [17]

4.3.2 Implementácia hernej logiky

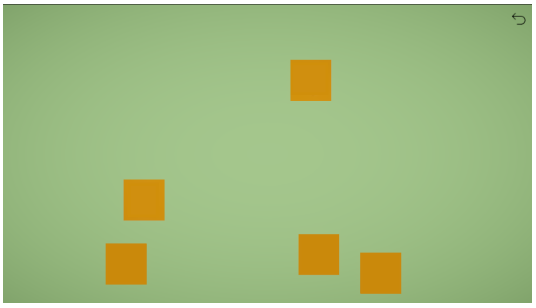
Ako prvé sme implementovali komponent *CubeManager*. *CubeManager* má na starosti správu objektov v scéne, s ktorými hráč interaguje. *CubeManager* vytvára potrebný počet objektov pomocou veľmi jednoduchého **object pooling** systému, vydáva príkazy objektom obnoviť sa do východzieho stavu a zmenšuje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

Každý objekt v scéne má na sebe komponent *CubeController* [18]. *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[5].

Na záver, najdôležitejší komponent *GameController*, ktorý ovláda logiku hry. *GameController* štartuje úrovne, 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 ľubovolný 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 ak to náhodou bol posledný objekt s posledným poradovým číslom, tak *GameController* spustí novú úroveň s o jedno viac objektami.

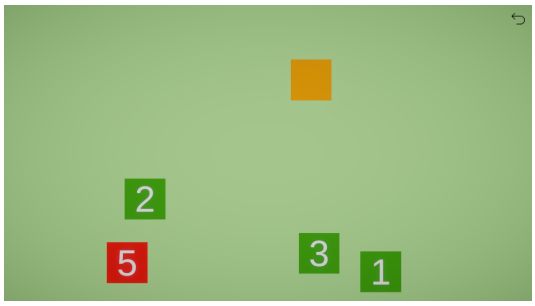
[5] v našom prípade je id rovné poradovému číslu

(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 **Oberver** [11], 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, importovať zdrojové súbory plugin-u do projektu. Potom vložiť do scény *LPIP_Plugin* prefab plugin-u.

Ak sme chceli aby nejaký objekt(y) interagovali s plugin-om museli sme implementovať interface nášho plugin-u *LPIPIInteractable*. Teda stačilo v komponente *CubeController* zabezpečiť, aby sa zavolala metóda *OnMouseDown()*, t. j. aby *CubeController* implementoval interface *LPIPIInteractable* (viď Kód 4.15).

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

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

Na záver, potrebujeme naštartovať plugin. Na to nám stačí pridať funkcionalitu 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, programátorovi je umožnené si toto prispôsobiť jeho potrebe.

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 čo najlepšej presnosti. Pri zmene polohy kamery, úmyselnom i neúmyselnom, je nutná opätovná kalibrácia.

Odporúča sa využívať v miestnosti s čo najmenším počtom okien, pre minimalizáciu vplyvu slnečného svetla. Miestnosť, v ktorej sa používa aplikácia s plugin-om by mala byť rovnomerne osvetlená a nie príliš, respektíve 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, t.j. nebol by detegovaný.

Záver

Prvým cieľom bakalárskej práce bolo vyvinúť plugin implementujúci podporu laserovej pištole. Druhým cieľom bolo vyskúšať ho v jednoduchej počítačovej hre, ktorú si navrhneme. Ciele sme si rozdelili na niekoľko podúloh, ktorým sme sa postupne venovali.

Ako prvé sme potrebovali vymyslieť upevnenie polarizačných filtrov na kameru pretože sa ukázalo, že použitie polarizačných filtrov nám vo väčšine prípadov zjednoduší hľadanie laserového lúča v obraze, čo nepredstavovalo príliš veľkú prekážku. V priebehu začiatku prác sme si zhotovili prvý funkčný prototyp uchytenia polarizačných filtrov na kameru z krabičky z tvrdého papiera, ktorý bol v tom čase dostatočne robustný a dalo sa s ním jednoducho pracovať. Časom sa však ukázalo, že tento prototyp je síce dobrý, ale neustálou manipuláciou jeho použiteľnosť a výdrž degradovala. Zhotovili sme teda rafinovanejšiu a robustnejší konštrukciu z dreva, ktorá funkcionalitou bola rovnaká ako náš prvý prototyp s drobnými zmenami. Keďže nedisponujeme 3D tlačiarňou a samotný 3D návrh modelu by spotreboval pomerne veľa času rozhodli sme použiť konštrukciu z dreva. Hlavná nevýhoda, ktorú by sa zišlo spomenúť je, že toto upevnenie polarizačných filtrov na kameru je dosť špecifické a nedá sa použiť na iné druhy kamier. Z dôvodu rôznorodosti tvarov kamier sme ani neriešili otázku všeobecného modelu upevnenie polarizačných filtrov na kameru.

Jednou z pomerne nie veľmi jednoduchých úloh bolo vymyslieť dobrú architektúru, vďaka ktorej bude plugin v prípade potreby jednoducho rozšíriteľný a ľahko použiteľný. Finálnu architektúru sa nám nepodarilo vytvoriť hneď na prvý pokus, ale stálo za tým niekoľko iterácii, ktoré si čitateľ v prípade záujmu môže pozrieť na GitHub repozitári (v Dodatku A) ako históriu commit-ov a branch-ov.

Implementácia obsahovala tiež niekoľko podúloh. Ako prvú podúlohu sme museli vyriešiť ako bude pracovať plugin, čo sme vyriešili tak, že plugin funguje ako stavový automat, ktorý sa skladá zo štyroch stavov, kde každý mal jasne definovanú svoju úlohu.

Druhá podúloha riešená v implementácii bola detekcia laserového lúča v obraze z kamery, kde sme sa pozreli na rôzne prístupy. Skúsili sme sa pozerať na intenzitu pixlov, farby pixlov a neskôr kombináciu týchto dvoch prístupov. Neskôr, keďže všetky predošlé prístupy fungovali výhradne na CPU, sme sa rozhodli toto riešenie zefektívniť a to tak, že sme prepísali kód s využitím compute shader-u a jazyka HLSL[6], kde sa do spracovania obrazu zapojilo už aj

[6]High-Level Shader Language

GPU. Pri zefektívnení sme potom dosahovali aj dvojnásobný výkon pri FullHD rozlíšení spracovávaného obrazu.

Tretia podúloha implementácie sa týkala kalibrácie a transformácie súradníc, ktorá pracovala s výsledkami druhej podúlohy. Vyskúšali sme kalibráciu len pomocou dvoch bodov, čo sa ukázalo teoreticky dobré, ale v praxi ťažko použiteľné. Použili sme teda kalibráciu pomocou až štyroch bodov, kde sme odskúšali dva principiálne rozličné algoritmy na transformáciou súradníc z jednej súradnicovej sústavy do druhej. Vytvorili sme si vlastný nástroj na meranie presnosti a odmerali presnosti všetkých troch riešení za rôznych podmienok umiestnenia kamery. Zistili sme, tak ako sme predpokladali, že kalibrácia pomocou dvoch bodov nám v praxi stačiť nebude. Pri kalibrácií pomocou štyroch bodov naše zistenia boli také, že jeden z týchto dvoch algoritmov nie je veľmi použiteľný, síce bol presný v blízkosti stredu, ale bol výrazne nepresný na krajoch, kde druhý algoritmus sa správal prijateľne. Druhý algoritmus obstál najlepšie vo všetkých testovaných scenároch a to zo stále prijateľnou presnosťou, respektíve nepresnosťou.

Štvrtá podúloha sa týkala už len realizácie simulovania kliknutia ľavým tlačidlom myši na súradniciach, ktoré sme dostali ako výsledok riešenia tretej podúlohy. Tu sme odskúšali tri prístupy. Prvý sa týkal komunikácie so systémom a hýbaním kurzora pomocou API systému. Problémom bolo, že každý operačný systém by si vyžadoval vlastnú implementáciu, spôsob akým sú vo Windowse umiestnené virtuálne obrazovky a nie vždy je žiadúce hýbať kurzorom systému. Druhý sa týkal jedného input sub-systému v Unity, ktorý sa nedal použiť čisto z praktických dôvodov. Tretí prístup sa týkal vlastného interface-u, ktorý museli implementovať všetky objekty, ktoré chcú interagovať s plugin-om. Problémom tretieho riešenia bola potreba implementovať interakciu s UI elementami.

Piata podúloha sa týkala vytvorenia doplnkových utilít na zjednodušenie používania plugin-u.

Šiesta podúloha sa týkala implementácie aplikácie, do ktorej sme následne implementovali náš plugin a odskúšali aplikáciu ovládať laserovým lúčom.

Výsledkom našej práce je plugin pre Unity implementujúci podporu laserovej pištole (všeobecne laserového lúča) a aplikácia, do ktorej sme implementovali plugin a odskúšali si ju zahrať.

Literatúra

[1] Andrej Lúčny. *Nové možnosti fyzickej interakcie hráča s počítačovou hrou. PREMENY VIZUÁLNYCH EFEKTOV*. AVFX, VŠMU, Bratislava, 2019. Dostupné na https://www.avfx.sk/konferencia-vfx-2019-pomocne-videa. Pristupované 14.5.2023.

[2] 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, 2021. Pristupované 4.3.2023.

[3] Laser Ammo Ltd. Smokeless range © 2.0 - home simulator. https://www.laserammo.eu/smokeless-range-at-home-range, 2015. Pristupované 4.3.2023.

[4] Laser Ammo Ltd. Smokeless range © 2.0 - starting guide video. https://www.youtube.com/watch?v=sC8pXNnrblY, 2023. Pristupované 5.3.2023.

[5] Unity. Made with unity. https://unity.com/made-with-unity, 2023. Pristupované 18.3.2023.

[6] Unity. Students get in-demand job skills with unity. https://unity.com/case-study/university-miami-uaa-case-study, 2023. Pristupované 18.3.2023.

[7] 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.

[8] Brad Jones. What is c#? https://www.developer.com/guides/what-is-c/, 2001. Pristupované 5.3.2023.

[9] David Marr. *Vision: A Computational Investigation into the Human Representation and Processing of Visual Information*. The MIT Press, 2010.

[10] Milan Sonka, Václav Hlavác, and Roger Boyle. *Image processing, Analysis, and Machine Vision*. Cengage Learning, 4th edition, 2015.

[11] Tony Bevis. *C# Design Pattern Essentials*. Ability First Limited, 1 edition, 2012.

[12] ITU. *Parameter values for the HDTV standards for production and international programme exchange*, page 4. ITU, 6nd edition, 2015.

[13] Andrej Lúčny. Point tranformation algorithm. `https://github.com/andylucny/calib4points/blob/main/calib2.py`. Pristupované 24.2.2023.

[14] Andrej Lúčny. Point tranformation algorithm. `https://github.com/andylucny/calib4points/blob/main/calibqkv.py`. Pristupované 15.3.2023.

[15] Microsoft. Documentation (winuser.h). `https://learn.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-mouse_event`. Pristupované 22.4.2023.

[16] Unity. Documentation (standaloneinputmodule). `https://docs.unity3d.com/560/Documentation/ScriptReference/EventSystems.StandaloneInputModule.html`. Pristupované 22.4.2023.

[17] Simon Jackson. *Unity 3D UI Essentials*. Packt Publishing, 2015.

[18] Jeff W. Murray. *C# Game Programming Cookbook for Unity 3D*. A K Peters/CRC Press, 2014.

Dodatok A

Zdrojové kódy

Kompletný zdrojový kód je k dispozícií na Github repozitári na odkaze: `https://github.com/CapitanMikon/LaserPointerInputPlugin`, na uvedenom odkaze bude umiestnené aj prezentačné video.

Dodatok B

Návody a príručky

Spracovali sme aj návod, ktorý je dostupný na GitHub repozitári ako samostatný PDF dokument (viď Dodatok A) v anglickom jazyku, ilustrovanom na našej vytvorenej aplikácii.