УНИВЕРЗИТЕТ У БЕОГРАДУ
МАТЕМАТИЧКИ ФАКУЛТЕТ

Лука Живановић

РАЗВОЈ АПЛИКАЦИЈЕ ЗА УРЕЂИВАЊЕ И ПРИКАЗИВАЊЕ СЛИКА У ПРОШИРЕНОЈ СТВАРНОСТИ ЗА ОПЕРАТИВНИ СИСТЕМ ANDROID УПОТРЕБОМ РАЗВОЈНОГ ОКРУЖЕЊА UNITY

мастер рад

Београд, 2022.

Ментор:

др Иван Чукић, доцент
Универзитет у Београду, Математички факултет

Чланови комисије:

др Ивана Томашевић, доцент
Универзитет у Београду, Математички факултет

др Мирјана Маљковић, доцент
Универзитет у Београду, Математички факултет

Датум одбране: 30.09.2022.

Наслов мастер рада: Развој апликације за уређивање и приказивање слика у проширеној стварности за оперативни систем Android употребом развојног окружења Unity

Резиме: Апликација описана у раду демонстрира како се у развојном окружењу *Unity* може решити проблем учитавања корисничке слике са мобилног уређаја, њеног уређивања и позиционирања у проширеној стварности (енг. *Augmenter Reality - AR*). Слика се учитава помоћу функционалности оперативног система *Android*. У раду су приказани основни алгоритми за модификовање слике (осветљеност, контраст, итд.), као и напреднији алгоритми (проналажење ивица, уље на платну). Такође је описана процедура динамичког креирања објекта од корисничке слике и њено позиционирање на зид просторије у проширеној стварности помоћу камере *Android* уређаја.

Кључне речи: Проширена стварност, развојно окружење *Unity*, алгоритми за уређивање слика, оперативни систем *Android*

Садржај

Глава 1

Увод

Unity је развојно окружење за креирање видео игара (енг. *game engine*) за различите платформаме (конзоле за играње видео игара, рачунаре, мобилне телефоне, итд.). Иако је иницијално замишљено и направљено за развој тродимензионалних игара, у овом раду биће приказан развој апликације која користи проширену стварност (енг. *Augmented reality - AR*) како би помогла кориснику да направи, модификује и види слику на зиду просторије користећи камеру уређаја заснованог на оперативном систему *Android*.

Главна тематика овог рада је упознавање са развојним окружењем и проширеном стварношћу на примеру поменуте апликације.

У другом поглављу је описана проширена стварност, односно основни концепти и историја, као и хардвер, софтвер и примене ове технологије. Ова апликација је развијена за *Android* уређаје чије се камере и сензори користе како би се слика позиционирала у простору и везала за локацију на зиду. Такође су описани и други уређаји који помажу у конзумирању проширене стварности као и разне примене у којима се ова технологија показала корисном, попут индустрије видео игара, ауто-мото индустрије, едукације и других.

У трећем поглављу су објашњени основни концепти развојног окружења *Unity*, чије је разумевање потребно да би се испратио развој апликације описан у петом поглављу.

Четврто погавље пролази кроз све функционалности које апликација пружа кориснику, док је у петом поглављу описана сама имплементација у којој ће бити обрађени алгоритми за уређивање слике, детекција објеката методом емитовања зрака (енг. *raycasting*), оптимизација кода за *Android* уређаје и

других.

Четврто погавље пролази кроз све функционалности које апликација пружа кориснику, док је у петом поглављу описана сама имплементација у којој су обрађене методе коришћене у раду попут алгоритама за уређивање слике, детекција објеката методом емитовања зрака (енг. *raycasting*), оптимизација кода за *Android* уређаје итд.

У последњем поглављу је изведен закључак у ком је описан значај апликације и потенцијал рада у овом развојном окружењу и проширеној стварности.

Глава 2

Проширена стварност

Да би дефиниција проширене стварности била у потпуности разумљива, дефинишимо прво технологију на основу које је проширена стварност настала - виртуелна стварност (енг. *Virtual reality - VR*).

Виртуелна стварност је симулирано тродимензионално окружење са којим корисник може да интерагује слично као са стварношћу. Симулација стварности је направљена рачунарским хардвером и софтвером [1]. За разлику од уобичајног конзумирања рачунарског софтвера, за доживљај виртуелне стварности корисник би требало да носи уређаје попут кацига или наочара како би интераговао са генерисаном околином. Што је корисник изолованији од реалног окружења, то је доживљај виртуелне стварности уверљивији.

Проширена стварност је варијанта виртуелне стварности. За разлику од виртуелне стварности где се корисник налази у потпуно синтетичком визуелном окружењу, у проширеној стварности корисник интерагује са реаланим светом око себе који је обогаћен синтетички генерисаним објектима.

Коначно, проширену стварност можемо дефинисати као интерактивно искуство реалног окружења у реалном времену где су објекти реалног света измењени рачунарски генерисаним информацијама које утичу на различита чула [2] - најчешће чуло вида, али и слуха и додира.

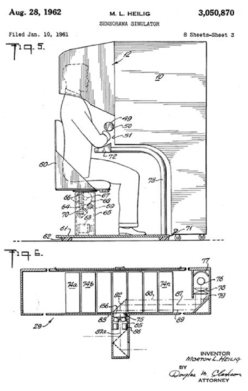
2.1 Историја

Идеја о проширеној стварности је први пут документована 1901. године у наслову „Главни кључ" (енг. *The Master Key*) Л. Френка Баума [3] (слика 2.1). У поменутој новели дечак добија мноштво поклона од демона, међу

којима је и обележивач људи (енг. *Character Marker*) који је представљен као пар наочара које на челима људи носиоцу приказују слова која описују карактерне особине појединца.

Слика 2.1: Главни кључ

1955. године излази рад под називом „Биоскоп будућности" (енг. *The Cinema of the Future*) у коме аутор Мортон Хајлинг описује уређај који има интегрисан екран у боји, стерео звучнике, емитере мириса, покретну столицу и вентилаторе за доживљај ветра. 1962. године је имплементиран овај прототип под називом *Sensorama* [4], помоћу кога је корисник могао да доживи вожњу мотоцикла кроз горе поменуте сензације.



Виртуелна стварност наставља да се развија 60-их, 70-их и 80-их година прошлог века. Током тог периода настаје први екран причвршћен за главу (енг. $head - mounted\ display$), што је претеча данашњих VR наочара и први преносни рачунар.

Године 1986. у оквиру компаније IBM Рон Фаигенблат објављује рад у ком описује први сценарио проширене стварности у ком је могуће мање екране померати по великој површини ради лакше организације [5].

Термин проширена стварност (AR) први пут се појављује у раду Томаса П. Каудела 1990. године, а први пут је под тим називом имплементиран за потребе авијације Сједињених Америчких Држава 1992. године [7]. Након тога проширена стварност је убрзо дошла и у комерцијалне сврхе, где су примери уређаја *Sony EyeToy* [6] и $Microsoft\ HoloLens$ [8].

2.2 Хардвер

Проширена стварност обухвата велики спектар примена, што ћемо поменути у наредном поглављу. У зависности од примене варира и хардвер који се користи.

Паметни телефони и таблети

Данас је најучесталији уређај који се користи за конзумирање садржаја у проширеној стварности паметан телефон. Конкретно код телефона или таблета користимо камеру али и сензоре микроелектромеханичког система (енг. *microelectromechanical systems* - $MEMS$), као што су сензор за убрзање (енг. *accelerometer*), систем глобалног позиционирања (GPS) и компас. Поменуте сензоре користимо како бисмо одржали положај објеката, рецимо у сценарију када напусте видно поље камере, или да одржимо релативан угао објекта у односу на уређај.

Наочаре и монитори фиксирани на глави

Монитори фиксирани на глави (енг. $head-mounted\ displays$ - HMD) су категорија уређаја коју дефинишу један или више екрана позиционирана непосредно испред лица корисника у затвореном простору (слика 2.2). Иако се овакви системи најчешће користе за виртуелну стварност, уз помоћ камере корисник може имати и сензације реалног света. За разлику од HMD, наочаре за проширену стварност имају провидна или полупровидна стакла на која се емитују синтетички објекти који се помоћу камере и сензора интегришу у реалан свет (слика 2.3).

Слика 2.2: HMD - Oculus Quest 2

Слика 2.3: Наочаре за проширену стварност - Hololens

Екрани испред лица

Екрани испред лица (енг. $Heads-up\ display$ - HUD) су провидни екрани који покривају део корисничког видног поља. Ови уређаји приказују релевантне информације тако да корисник не мора да скреће поглед са главне радње. Ова примена се неће сложити са неким дефиницијама проширене стварности. Међутим како ови уређаји прате информације спољашњег света и враћају кориснику информацију у реалном времену, сматрамо их хардвером за проширену стварност. Користе се у војне, авио и ауто-мото сврхе (слика 2.4).

Слика 2.4: Екран испред лица у аутомобилу

Контактна сочива

Екрани у контактним сочивима су патентирани 1999. године за потребе пројекта који није успешно имплементиран. Следећа верзија, која није била јавно доступна, настала је 2011. године и развила ју је војска Сједињених Америчких Држава. Први прототип приказан широј публици је направила компанија под називом *Mojo Vision* и приказан је на конференцији *CES* 2020. године [9].

Уређаји за праћење

Претходно смо поменули сензоре телефона и њихову улогу у проширеној стварности. Такође постоји велики број сензора развијених искључиво за ове потребе који могу да измере разне параметре везане за субјекат који се прати. Поменути сензори (слика 2.5) пружају релативне информације у односу на суседне сензоре, чиме се постиже већа прецизност мапирања реалног света у синтетички и обрнуто. Они често раде са таласима ван видљивог спектра (нпр. у инфрацрвеном), тако да систем може имати више информација него људско око, што је неопходно за бољу интеграцију реалног и синтетичког система у превише тамним или превише светлим сценама. Продор у комерцијално тржиште је направила компанија *Apple*, тиме што су имплементирали *Lidar* сензор у своје мобилне уређаје 2020. године, који служи за прецизну детекцију рељефа објекта у који је уперен [10].

Слика 2.5: Студио за проширену стварност

2.3 Софтвер

Кључни посао алгоритама за рад са проширеном стварношћу је разумевање реалног света и интеграција са њим. Да би се успешно интегрисао са реалним светом, алгоритму су потребне информације реалног света што добија од горе поменутих сензора.

Основни сензор којим ови алгоритми баратају је камера, те се сами алгоритми ослањају на рачунарски вид (енг. *computer vision*) [11]. Рачунарски вид је наука у успону и њени алгоритми се успешно користе за проналажење углова, ивица и осталих карактеристика добијених из дигиталне слике или секвенци слика.

По наласку поменутих информација алгоритам покушава да мапира те специфичности простора у свој координатни систем. Овом процесу могу помоћи и маркирани објекти за бољу апркосимацију дальина у простору (слика 2.6).

Слика 2.6: Маркери за калибрацију проширене стварности

За коришћење проширене стварности често се користе готови додаци (енг. *plugin*) који пружају разне сценарије за рад у проширеној стварности. Представник таквог додатка је *ARCore XR*, који је искоришћен у овој апликацији. Овај додатак може да детектује равни и мрежу објекта, има подршку за емитовање зрака (енг. *raycast*), проналажење задате слике (маркера) у простору, препознавање карактеристика људског тела и лица, итд.

2.4 Примене

Проширена стварност се примењује у многим пољима, а неке од њих су:

- авијација,
- археологија,
- архитектура,
- ауто-мото индустрија,
- видео игре,
- војска,
- друштвене мреже,
- едукација,
- колаборација,
- комерцијална употреба,
- рекламирање,
- уметност.

У наставку поглавље ћемо детаљније објаснити неке од наведених примена.

Видео игре

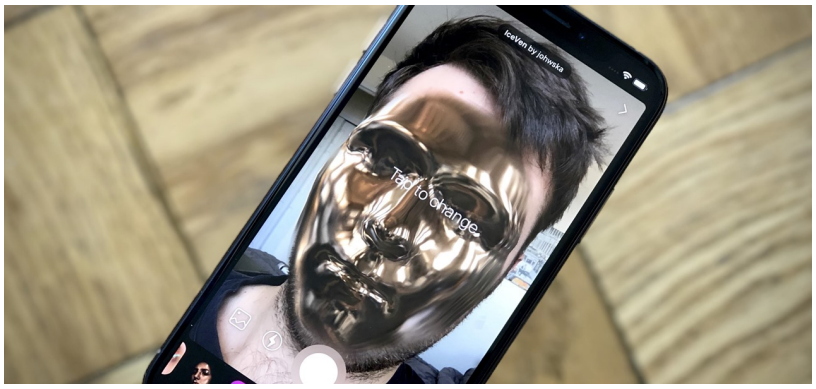
Игре су можда и најпопуларнији вид коришћења проширене стварности. Популарне су друштвене игре где играчи могу играти популарне наслове на свом столу путем интернета. Играчи користе генерисану таблу за играње да би урадили различите задатке и сл (слика 2.7). Предност играња у проширеној стварности, осим превазилажења проблема физичке дистанце, је смањивање материјала потребног за израду игара, смањивање цене игре, као и специјални ефекти и анимације које се могу имплементирати у овим системима, за разлику од физичких игара. Други тип видео игара које користе проширену стварност ослањају се на природу и околину, па у зависности од физичке локације корисинка игра нуди различита искуства. Најпопуларнија игра која се у потпуности ослања на проширену стварност је *PokemonGo* (слика 3.1) [12] која је имала и до 150 милиона активних играча у току једног месеца, а почетком 2019. године је имала више од милијарду преузимања.

Слика 2.7: Видео игра за мобилни телефон $Kingfall$

Друштвене мреже

Друштвене мреже су сигурно апликације које имају највише корисника који користе вештачку стварност. Тренутно се велика већина филтера на друштвеним мрежама (слика 2.8) заснива на проширеној стварности, а број налога на друштвеним мрежама је 2021. године прешао три милијарде.

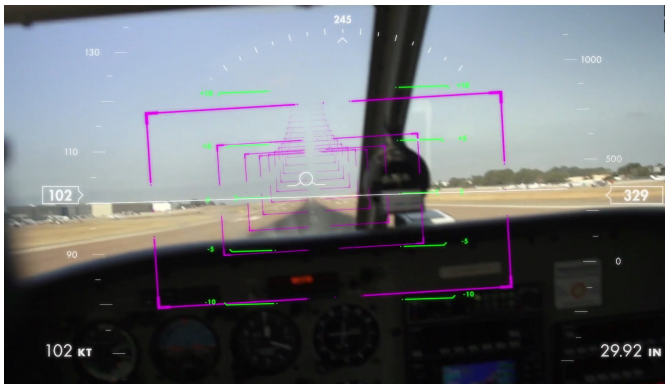
Слика 2.8: Проширена стварност на друштвеним мрежама



Војска и авијација

Док су се претходне две имплементације заснивале на употреби мобилних уређаја, војска и авијација користе специјализовани хардвер за проширену стварност, као што су наочаре, монитори фиксирани на глави, екрани испред лица и сл. На овим уређајима војници и пилоти могу видети релевантне информације у току задатка, користити их за тренинг и комуницирати са колегама (слика 2.9)

Слика 2.9: Проширена стварност у авијацији



Глава 3

Развојно окружење Unity

Развојно окружење *Unity* пружа кориснику читав спектар алата за ефикасно прављење 3D игара. Иако је то примарна примена развојног окружења, богатство алата дозвољава кориснику и много више од тога.

3.1 Историја

Unity је развојно окружење које је развила компанија *Unity Technologies* 2007. године. Од настанка до данашњег дана, *Unity* је прошао кроз 5 значајних верзија које су донеле бројне погодности корисницима. Године 2007. додата је подршка за развој игара за оперативни систем *iOS*, који користе телефони компаније *Apple*. Убрзо и *Anrdoid* платформа добија подршку, и само окружење добија нове могућности. Због наведених разлога у мају 2012. године бива проглашен као најбоље развојно окружење за развој мобилних игара у познатом часопису под називом *Game Developer*.

Године 2012. *Facebook* имплементира комплет за развој софтвера (енг. *software development kit - SDK*). До дана данашњег листа партнера са *Unity*-ем расте, тако да данас корисник може развијати апликације и видео игре користећи интерфејс за програмирање апликација (енг. *application programming interface - API*) *Vulkan*, систем за физику *Microsoft Havok*, и системе за управљање проширеном стварношћу *ARKit* и *ARCore XR* оперативних система *Android* и *iOS*, као и многе друге производе осталих партнера.

Тренутно је могуће креирати апликације за све популарне платформе, попут:

- Windows, MacOS, Linux,
- Android, iOS,
- Xbox, Playstation, Nintendo switch,
- WebGL, Unity Web Player.

Данас *Unity* користи преко 230 хиљада програмера, што је 93% већи број него 2021. године [13]. Успешност *Unity*-а доказују и успеси наслова направљених у овом окружењу, међу којима је и игра *Among Us*, која је бележила преко 400 хиљада играча у једном дану, *PokemonGo* чији је успех поменут у претходном поглављу, као и *Beat Saber*, најпопуларнију видео игру у виртуелној стварности (слика 3.1).

Слика 3.1: Популарни наслови направљени у развојном окружењу *Unity*: *PokemonGo*, *Among Us*, *Beat Saber*

3.2 Општи елементи развојног окружења

Активе

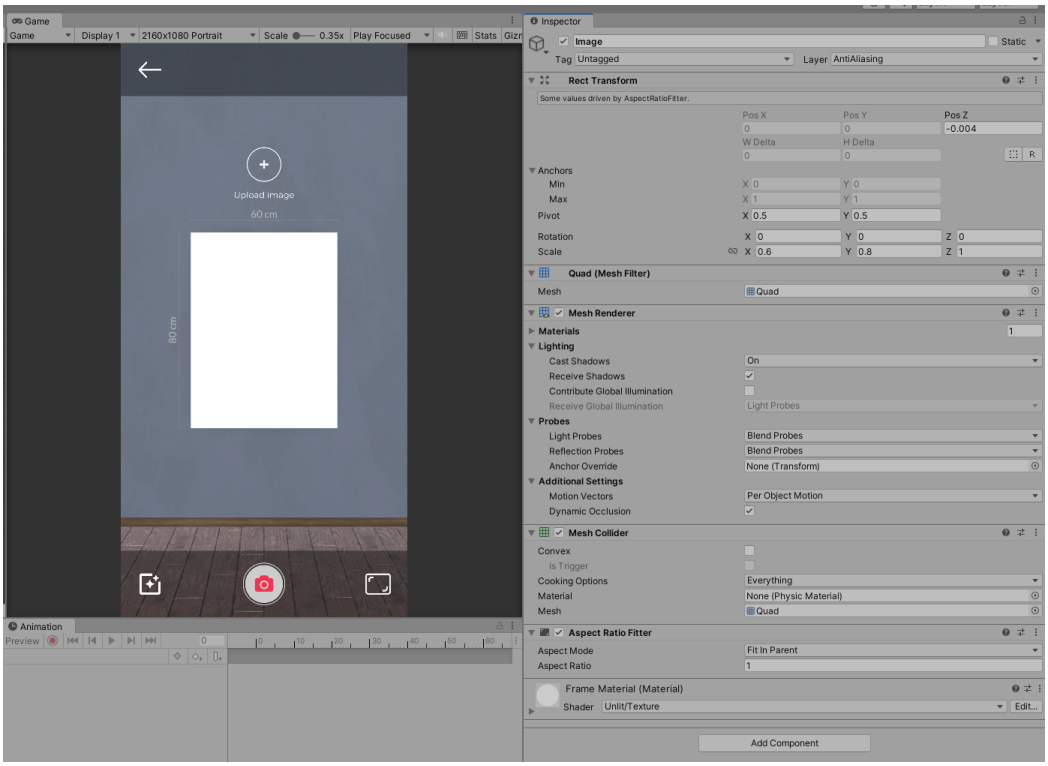
Актива (енг. *Asset*) је основни елемент развојног окружења. Актива може бити слика, 3D модел, звук, анимација, итд. *Unity* има интегрисан алат за рад са већином ових елемената, тако да корисник може направити 3D модел, текстуру за њега и анимацију у оквиру окружења. Такође, *Unity* има продавницу актива (енг. *Asset store*), на коју корисници могу постављати бесплатне или плаћене активе како би их други корисници преузимали или куповали.

Објекти игре и компоненте

Како се већина датотека у *Unity* пројекту сматра активом, тако се и сви елементи у самој апликацији сматрају објектом игре (енг. *GameObject*). Обје-

кат игре може имплементирати разна својства помоћу компоненти, као што су 3D мреже, модел, звук, скрипте и сл. На слици 3.2 је приказан објекат као платно и са десне стране су побројане све његове компоненте. Компонента *Mesh Renderer* служи за само исцртавање објекта, *Mesh Colider* детектује колизије овог и других објеката или зракова (енг. *ray*), компонента *Aspect Ratio Fitter* служи за одржавање висине и ширине у односу на родитеља.

Слика 3.2: Компоненте



Такође је могуће наслеђивати објекте. То је изузетно корисно уколико желимо да групишемо објекте, пошто ће се исти налазити у координатном систему родитеља. Приметимо на слици 3.2 да су X и Y координате постављене на 0. То је случај зато што се платно у хијерархијском стаблу налази испод слике, која је позиционирана где треба на зиду. Платно је у односу на њу позиционирано на почетку локалног координатног система (координатног система слике) и померено је од камере за 0,004 јединице.

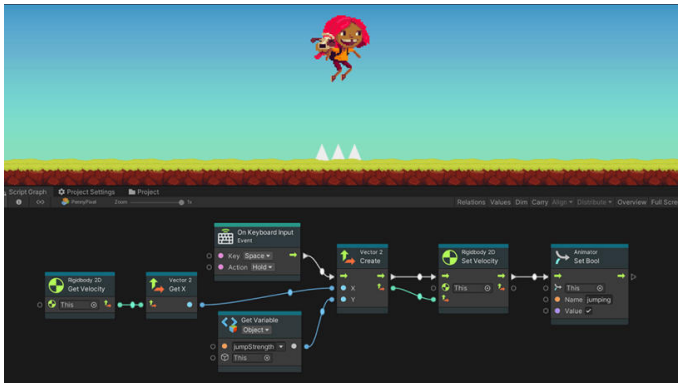
Функционалност објеката ћемо дефинисати скриптама. Да би се функционалност учитала у саму сцену апликације она мора бити везана за објекат,

тако да често правимо објекте који немају физичка својства али имају скрипте као компоненте које дају функционалност њима и другим објектима.

Скрипте

Предефинисане компоненте имају разне функционалности али не могу покрити сваки сценарио који је програмеру потребан. За то користимо скрипте које нам допуштају да сами дефинишемо функционалност апликације. Поменули смо како је скрипта компонента објекта и као таква може контролисати објекат по жељи програмера. Скрипте се пишу у програмском језику C#, а од 2021. године програмерима је доступан алат *Unity Visual Scripting* који допушта кориснику да помоћу визуелних дијаграма описује фунцкионалности апликације (слика 3.3).

Слика 3.3: *Unity Visual Scripting*



Скрипте се такође користе као компоненте објеката који немају физичку репрезентацију на сцени и воде рачуна о току игре или апликације (памте тренутна стања, праве и бришу објекте, итд.)

По креирању скрипте, направиће се C# фајл који ће имплементирати класу са називом те датотеке. Класа наслеђује *Unity* предефинисану класу под називом *MonoBehaviour*, која има разне функционалности. Главне су две методе: покрени (енг. *Start*) и освежи (енг. *Update*). Метода покрени се позива по прављењу објекта, док се метода освежи зове сваки пут када се исцрта нова слика на екрану. Учесталост исцртавања слике зависи од подешавања, комплексности апликације и јачине хардвера који је покреће.

Сцене

Сви објекти и стања су везани за сцену. Сцена представља логичку целину апликације и садржи иницијалну листу објеката. У прављењу видео игара сцене се користе за нивое, мени, одјавне шпице и сл. Сцене су погодне како би растеретиле хардвер и поделиле апликацију на целине.

Шаблони

Шаблони представљају објекте који су сачувани као активе. Шаблон не мора иницијално бити везан за сцену, већ се он често користи како би се направиле његове инстанце са могућим модификацијама у току извршавања апликације. Пример шаблона који се често користи у видео играма је пројектил (метак), кога треба инстанцирати много пута у сцени и то по потреби.

3.3 Својства и процедуре развојног окружења

Простор и координатни систем

Основа развојног окружења *Unity* је еуклидски тродимензионални простор. Координате простора су обележене са X, Y и Z, где X представља ширину, Y висину, а Z дубину. Сваки објекат може имати произвољан број компоненти, али обавезно мора имплементирати $transform$ компоненту коју дефинишу три параметра: позиција у простору, ротација и величина. Свака од ове три компоненте је описана са по три вредности које се везују за горе поменуте координате.

Unity такође пружа подршку рада са дводимензионалним објектима, као што су менији у играма. Ипак, у позадини *Unity* користи тродимензионални простор како би описао ове објекте, тако да је дводимензиони простор привидно створен и зависи од подешавања камере.

Камера

Камера је један од најбитнијих објеката за развој апликације зато што она у реалном времену претвара стања на сцени у слику коју корисник може

видети у јединици времена. Осим поменуте $transform$ компоненте, камера такође има камера (енг. *camera*) компоненту помоћу које се могу контролисати разни параметри слични параметрима реалне камере попут перспективе и видног поља (енг. $Field\ of\ View$ - FOV). Иако је камера један од основних објеката који се аутоматски прави по прављењу нове сцене, она може бити изостављена и замењена сличним објектима, што ће бити демонстрирано у даљем тексту када ће нам бити потребна физичка камера мобилног уређаја.

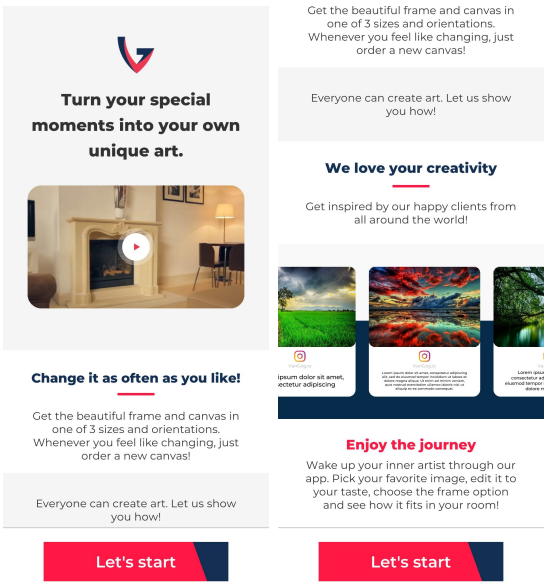
Глава 4

Опис апликације

Апликација развијена као део овог рада демонстрира коришћење проширене стварности у развојном окружењу *Unity* и имплементира функционалности учитавања и модификовања слике, као и њено постављање и померање по зиду у проширеној стварности користећи уређаје засноване на оперативном систему *Android*. Апликација се састоји из три сцене.

Прва сцена (слика 4.1) описује функционалност апликације подстичући корисника да је користи како би направио и видео на зиду просторије своју креацију.

Слика 4.1: Почетна сцена

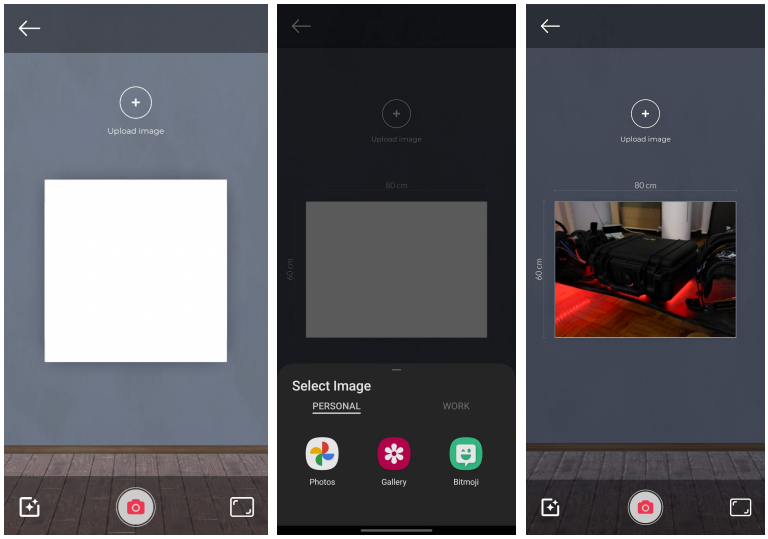


Ова сцена се састоји од промотивног видеа, текста и низа слика са дру-

штвених мрежа. Корисник се може кретати по страници вертикално, повлачењем прста (енг. *scroll*), као и хоризонтално по сликама са друштвених мрежа. Притиском на дугме које је позиционирано на дну екрана корисник прелази на другу сцену.

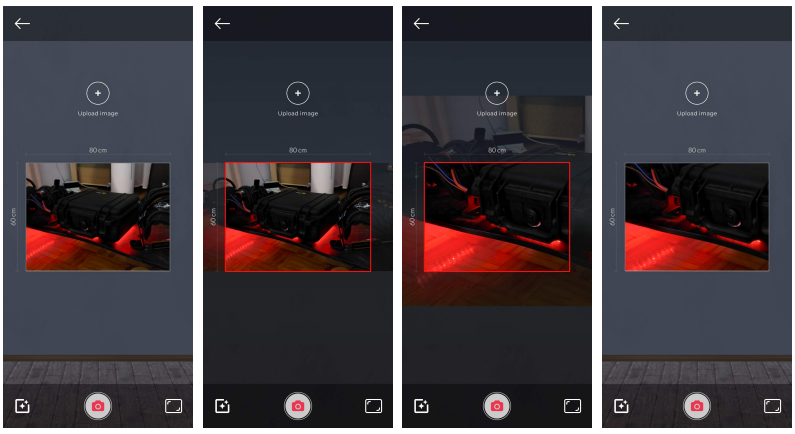
Друга сцена почиње анимацијом која симулира корисника који са леве стране прилази платну и стаје испред њега. Платно је иницијално празно и притиском на њега или дугме за додавање фотографије пружа кориснику могућност да исту учита са свог мобилног уређаја (слика 4.2).

Слика 4.2: Учитавање



Притиском на учитану слику корисник је може померити и исећи како би жељени део слике остао у раму (слика 4.3).

Слика 4.3: Позиционирање



Корисник такође може искористити функционалности доњег менија (тамни правоугаоник на дну слике 4.3), а то су:

- улазак у мени за модификовање слике, оквира и рама,
- улазак у сцену у којој може видети своју слику у проширеној стварности,
- промену димензије слике.

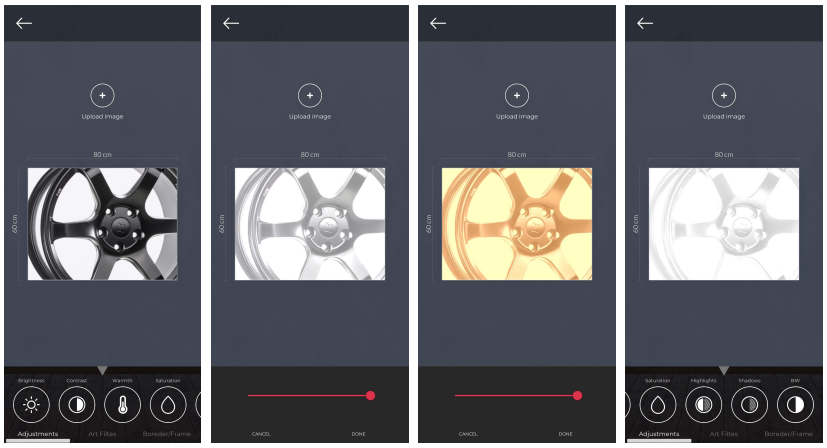
Притиском на прву иконицу менија, корисник може изабрати примену модификација, уметничких филтера и измене рама и оквира.

Имплементиране су све основне модификације:

- осветљење,
- контраст,
- топлота,
- сатурација,
- светли детаљи (енг. *highlights*),
- сенке,
- црно-бело.

Модификације се могу применити притиском на жељену иконицу која приказује клизач (енг. *slider*). Померањем клизача се мења вредност параметра изабраног филтера и резултат се приказује кориснику у реалном времену.

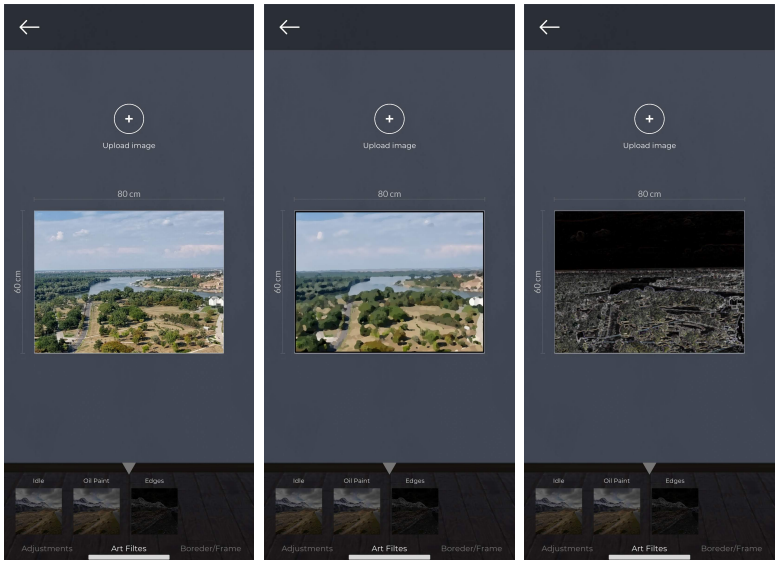
Слика 4.4: Филтери



Пример на слици 4.4 показује примену осветљења, топлоте и црно-белог филтера.

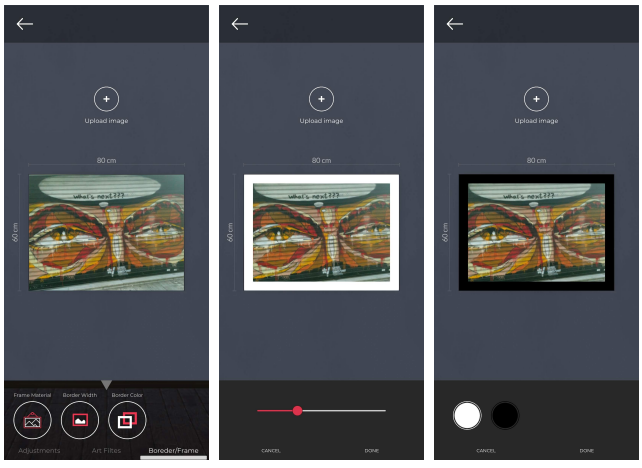
Друга опција у менију су уметнички филтери (слика 4.5), који се примењују само притиском на дугме и ту разликујемо идентитет - који не мења слику, уље на платну и детекцију ивица, чији се примери могу видети на слици.

Слика 4.5: Уметнички филтери



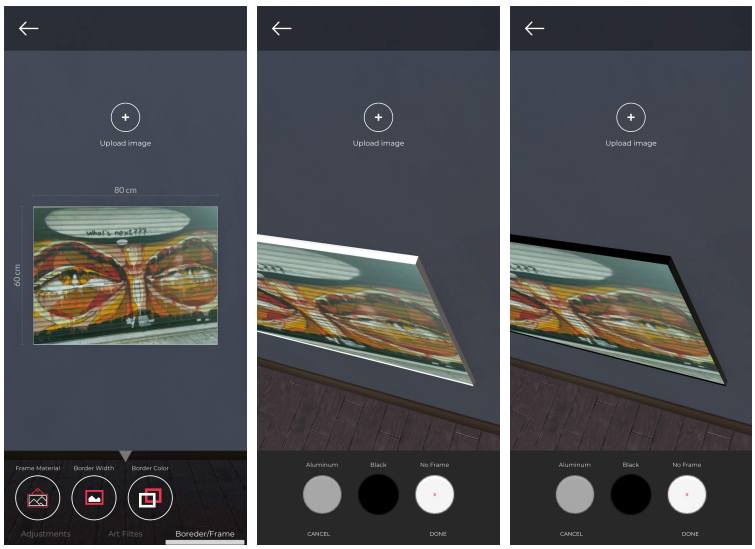
Коначно, корисник може мењати својства оквира и рама. Може се контролисати боја и ширина оквира као што је приказано на слици 4.6.

Слика 4.6: Промена боје и ширине оквира



Рам се из тренутне перспективе камере види као линија, тако да је за измену карактеристика рама потребно да се промени угао гледања, слично реалном свету. Притиском на дугме за промену материјала рама камера прилази ивици слике како би корисник могао да изабере материјал (слика 4.7).

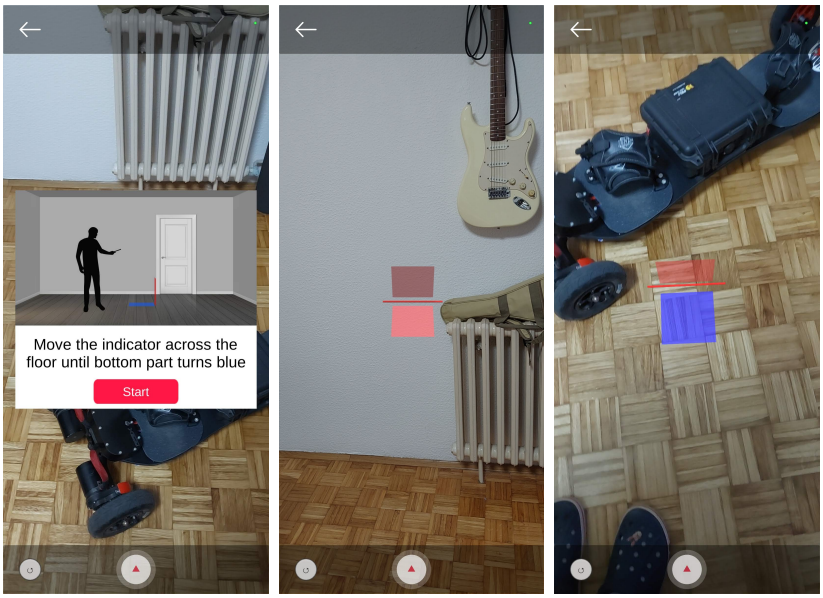
Слика 4.7: Промена материјала рама



Притиском дугмета за промену димензије слике, корисник ће у зависности од тренутне димензије добити следећу из низа: 60цм x 80цм, 60цм x 60цм, 80цм x 80цм и 80цм x 60цм.

Притиском на дугме обележено камером корисник улази у сцену са проширеном стварношћу. По уласку у сцену приказује се камера мобилног уређаја и упутство које наводи корисника да упери камеру у под просторије и помера је док доњи део индикатора не постане плав (слика 4.8).

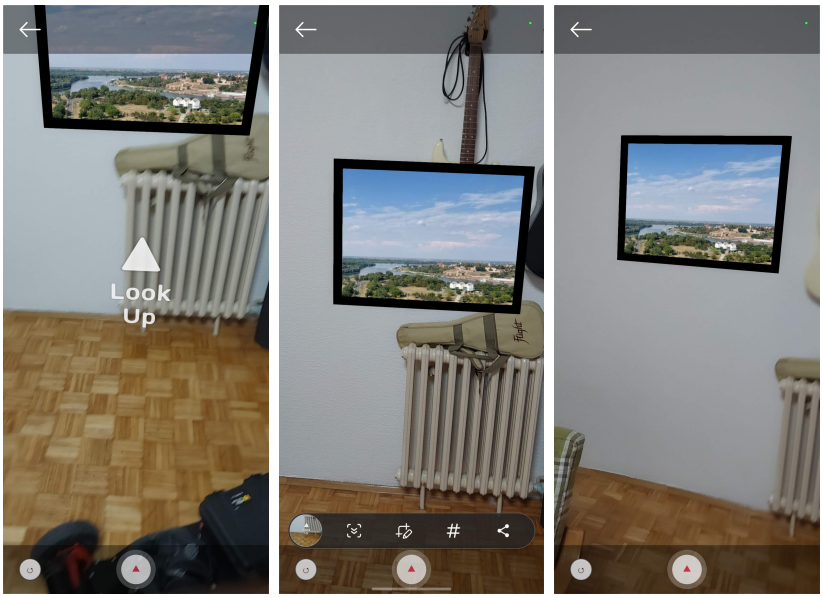
Слика 4.8: Калибрација индикатора у проширеној стварности



Тиме апликација обавештава корисника да је детектовала раван пода. Потом слична анимација обавештава корисника да индикатор приближи зиду просторије и притисне дугме за постављање слике на зид. Слика ће бити постављена у висини камере, а корисник ће бити обавештен где се слика налази.

Притиском екрана и померањем прста корисник може позиционирати слику на зиду (слика 4.9).

Слика 4.9: Постављање и померање слике по зиду



Слика ће остати фиксирана на зиду невезано за позицију корисника, тако да се корисник може померати по просторији и гледати слику из различитих углова (слика 4.10).

Слика 4.10: Посматрање постављене слике из другог угла

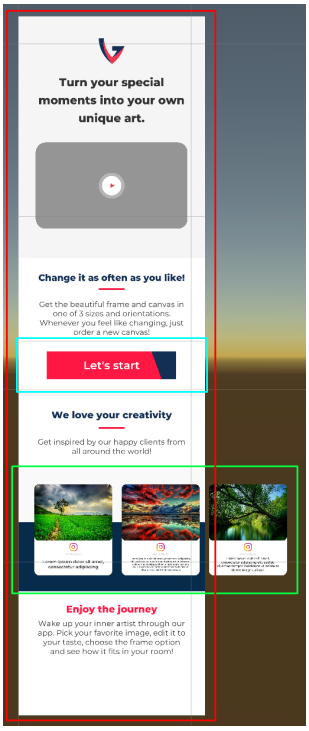
Глава 5

Имплементација

Свака сцена у апликацији представља различите имплементационе изазове. Прву сцену карактерише кориснички интерфејс, другу разне манипулације над текстурама и објектима, као и оптимизације, док трећу сцену карактеришу решења проблема проширене стварности.

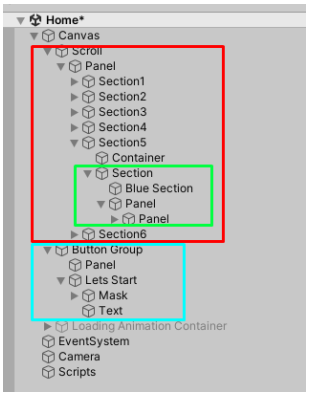
5.1 Почетна сцена

Слика 5.1: Почетна страница



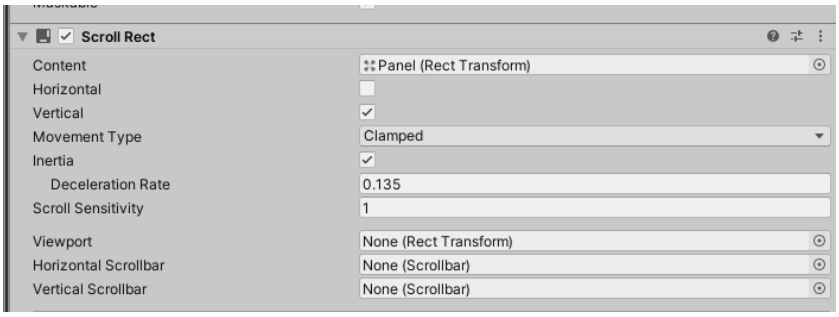
Кориснички интерфејс приказан на слици 5.1 се састоји од три велике секције уоквирене црвеном, плавом и зеленом бојом.

Слика 5.2: Стабло елемената



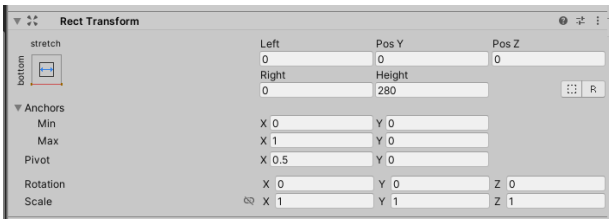
Сви елементи су хијерархијски поређани у стаблу елемената (слика 5.2). Доминантна је црвена секција, која се може вертикално померати (енг. *scroll*). Овај ефекат се постиже постављањем *Scroll Rect* својства на *Unity* објекат (слика 5.3). Додатно, постављањем атрибута *Movement Type* на *Clamped*, добија се леп ефекат где корисник може да превуче садржај и ван граница, али се по склањању прста са екрана лепом анимацијом садржај странице враћа у границе екрана.

Слика 5.3: *React transform* својство



Сличан ефекат је постигнут на зеленом елементу, који се може померати хоризонтално. Црвени елемент има апсолутан положај у простору. У развојном окружењу *Unity* то се постиже постављањем сидра (енг. *anchor*) на *React Transform* својству објекта, који лепи објекат за доњу, леву и десну ивицу екрана (слика 5.4). То такође омогућава уређајима различитих резолуција да приказују дугме исправно.

Слика 5.4: *Scroll Rect* својство



5.2 Галерија

Сцена која учитава и уређује слике је приказана на слици 4.2. Прва акција која се очекује од корисника је учитавање слике, које се може урадити притиском на платно или *Upload* дугме. Притисак на *Upload* дугме је једноставно имплементиран помоћу компоненти развојног окружења *Unity*. Слика на платну нема компоненте дугмета па је потребно одрадити пројекцију притиска екрана на координатни систем апликације и проверити да ли је платно додирнуто. Сваки пут када се екран освежи проверава се додир екрана следећом функцијом:

```
void handleDefaultMode () {
    if (Input.touchCount == 1) {
        var touch = Input.GetTouch (0);

        if (isObjectTouched (image, touch)) {
            if (touch.phase == TouchPhase.Began
                || (touch.phase == TouchPhase.Stationary
                    && this.defaultIsTapped == true)) {
                this.defaultIsTapped = true;
            } else if (touch.phase == TouchPhase.Ended) {
                if (this.defaultIsTapped == true) {
                    if (StaticImages.initImageTexture) {
                        changeMode (Mode.Edit);
                    } else {
                        PickImage ();
                    }
                }
                this.defaultIsTapped = false;
            } else {
                this.defaultIsTapped = false;
            }
        }
    }
}
```

Почетно стање (енг. *default mode*) означава да се слика тренутно не уређује, о чему ће бити више речи у наставку. Уколико је то тачно и окружење региструје додир екрана, проверава се да ли је притиснута слика позивом функције *isObjectTouched*.

```
bool isObjectTouched (GameObject obj, Touch touch) {
    Ray ray = Camera.main.ScreenPointToRay (touch);
    RaycastHit hit;
    if (Physics.Raycast (ray, out hit)) {
        if (hit.transform.gameObject == obj) {
            return true;
        }
    }
    return false;
}
```

Како слику на екрану видимо пројекцијом тродимензионалног простора на дводимензионални екран коришћењем камере, исту ту камеру можемо искористити да закључимо да ли је објекат у тродимензионалном простору додирнут. То се ради тако што емитујемо зрак (енг. *ray*) из камере (која је у развојном окружењу апроксимирана једном тачком) до равни на коју се пројектују објекти које камера види и где је регистрован додир. Потом се проверава да ли је тај зрак погодио објекат који смо проследили функцији (у нашем случају платно) и ако јесте, враћамо информацију да је објекат додирнут. Када региструјемо додир на слику, проверава се стање додира. За потребе ове апликације, желимо да региструјемо да је корисник додирнуо и пустио екран, да не би дошло до случајног регистровања додира. Уколико је то тачно и слика је учитана, мењамо мод рада са почетног на уређивање слике. У нашем тренутном стању немамо учитану слику, тако да зовемо функцију *PickImage()*.

```
private void PickImage (int maxSize = -1) {
    if (!Permission.HasUserAuthorizedPermission (Permission.ExternalStorageRead)) {
        Permission.RequestUserPermission (Permission.ExternalStorageRead);
    }

    NativeGallery.Permission permission = NativeGallery.GetImageFromGallery ((path) => {
        if (path != null) {
            var texture = NativeGallery.LoadImageAtPath (path, maxSize, false);
            StaticImages.initImageTexture = texture;
            if (texture == null) {
                return;
```

```
            } else {
                initResizeImagePreview ();
                imagePreview.GetComponent<Renderer> ().material.mainTexture = texture;
                changeMode (Mode.Default);
            }

        }
    }
}
```

Android уређај не дозвољава апликацијама да приступају фотографијама уколико не затраже и добију одговарајуће дозволе. У овом случају је потребна дозвола за читање спољних података (енг. *external storage read permission*). По добијању дозволе, учитавамо спољну слику и уписујемо је у статичку променљиву *StaticImages.initImageTexture*. Напокон, учитану текстуру слике примењујемо на објекат на сцени *imagePreview*. Ово није слика која се налази на платну, већ иза платна, док ће се на платну налазити њена копија исечена на димензије платна.

Учитана слика може бити произвољне величине и односа висине и ширине. Уколико је однос висине и ширине слике мањи од односа висине и ширине рама, односно слика је хоризонтално издуженија од рама, желимо да је хоризонтално центрирамо у односу на рам, а вертикално да има исту висину као рам и да стаје у њега (слика 4.3). У наредној функцији је приказано то скалирање.

```
private void initResizeImagePreview () {
        var initWidth = StaticImages.initImageTexture.width / 1000f;
        var initHeight = StaticImages.initImageTexture.height / 1000f;
        var imagePreviewAspect = initWidth / initHeight;

        var imageWidth = image.transform.localScale.x;
        var imageHeight = image.transform.localScale.y;
        var imageAspect = imageWidth / imageHeight;

        float scale = 0;

        if (imagePreviewAspect > imageAspect) {
            scale = imageHeight / initHeight;
        } else {
            scale = imageWidth / initWidth;
        }

        imagePreview.transform.localScale = new Vector3(
            initWidth * scale, initHeight * scale, 1);
    }
```

Величина слике се дели са 1000 због неконзистентих мерних јединица.

Позиционирање слике

Након иницијалног позиционирања, корисник може да увеличава и помера слику како би фокусирао жељени субјекат ни слици или исекао нежељене.

Акције позиционирања се обрађују када је апликација у моду за позиционирање, што се дешава притиском на платно када је слика учитана.

```
if (StaticImages.initImageTexture) {
    changeMode (Mode.Edit);
}
```

```
async void changeMode (Mode newMode) {
    switch (newMode) {
        case Mode.Default:
            cover.SetActive (false);
            imagePreview.SetActive (false);
            floor.SetActive (true);
            frame.SetActive (true);

            var texture = await TextureTools.processFinalImageAsync
                (imagePreview, image, true, TextureTools.processImageType.crop);

            _startedEditing = false;
            _mode = Mode.Default;
            break;
        case Mode.Edit:
            cover.SetActive (true);
            imagePreview.SetActive (true);
            floor.SetActive (false);
            frame.SetActive (false);

            _startedEditing = false; // True when user begins to zoom or drag
            imagePreview.transform.position = _imageStartPosition;
            _mode = Mode.Edit;
            break;
    }
}
```

По уласку у стање за позиционирање слике сакривамо рам слике, под и зид и приказујемо црну позадину иза које ће бити затамњен остатак слике и црвени индикатор који показује ивице које одсецају финалну слику. Сама промена величине и позиције слике ће бити обрађена у наставку. По изласку

из стања за позиционирање слике, враћају се објекти и процесуира слика у раму функцијом *processFinalImageAsync*. Приказани код функције у наставку није комплетан, зато што имплементација садржи логику за примену филтера и кеширање слике што ће бити објашњено у следећем поглављу.

```
public static async Task<Texture2D> processFinalImageAsync(
    GameObject imagePreview,
    GameObject image,
    bool updateObjects = true,
    processImageType type = processImageType.crop)
    {
        if(StaticImages.initImageTexture == null)
            return null;

        Texture2D tex = null;

        tex = ResampleAndCrop(imagePreview, image);
        tex = DecreaseResolutionHalfing(tex);

        if (updateObjects)
        {
            image.GetComponent<Renderer>().material.mainTexture = tex;
            StaticImages.finalImageTexture = tex;
        }

        return tex;
    }
```

Следећа функција прави копију оригиналне текстуре и сече је на величину рама на местима које смо изабрали у стању за позиционирање слике, оптимизује величину слике и примењује је на само платно.

```
public static Texture2D ResampleAndCrop(GameObject imagePreview, GameObject image)
    {
        Vector3 originalPosition = imagePreview.transform.position;
        Vector3 croppedPosition = image.transform.position;

        Vector3 originalScale = imagePreview.transform.localScale;
        Vector3 croppedScale = image.transform.localScale;

        // Calculates offsets in [0-1]
        var top = ((originalPosition.y + originalScale.y / 2)
                    - (croppedPosition.y + croppedScale.y / 2))
                  / originalScale.y;
        var bot = ((croppedPosition.y - croppedScale.y / 2)
                    - (originalPosition.y - originalScale.y / 2))
                  / originalScale.y;
        var left = ((croppedPosition.x - croppedScale.x / 2)
```

```
                - (originalPosition.x - originalScale.x / 2))
            / originalScale.x;
        var right = ((originalPosition.x + originalScale.x / 2)
                - (croppedPosition.x + croppedScale.x / 2))
            / originalScale.x;

        // Scales it with original texture
        int sourceWidth = source.width;
        int sourceHeight = source.height;

        // Calculates offsets
        top *= sourceHeight;
        bot *= sourceHeight;
        left *= sourceWidth;
        right *= sourceWidth;

        int targetWidth = sourceWidth - (int)left - (int)right;
        int targetHeight = sourceHeight - (int)top - (int)bot;
        var tex = new Texture2D(targetWidth, targetHeight);

        try
        {
            var pixels = source.GetPixels((int)left, (int)bot, targetWidth, targetHeight);
            tex.SetPixels(pixels);
        }
        catch (UnityException e)
        {
            Debug.LogError(e.Message);
        }

        tex.Apply(true);
        return tex;
    }
```

Функција *ResampleAndCrop* рачуна које пикселе треба исећи са оригиналне слике да би стала у рам на изабран начин. Горња ивица коју треба исећи се добија тако што се одузме y координата горње ивице оригиналне слике и горње ивице рама, односно исечене слике.

```
(originalPosition.y + originalScale.y / 2)
- (croppedPosition.y + croppedScale.y / 2)
```

Та разлика представља висину слике у тродимензионалном координатном систему, па ју је потребно нормализовати дељењем са висином слике. На сличан начин се добијају преостале три метрике (доња, лева и десна ивица).

Нормализоване вредности се множе висином, односно ширином оригиналне текстуре. Потом се прави нова текстура која ће бити примењена на

платну и њој се додељују пиксели из оригиналне слике искључујући ивице које су одсечене.

Како ову апликацију покрећу мобилни уређаји, примена алгоритама на слике велике резолуције би дуго трајала и оптерећивала уређај. Поред тога платно заузима одређени део екрана, па се слика веће резолуције од резолуције тог процента екрана не би ништа боље видела. Због набројаних разлога смањујемо резолуцију једноставним алгоритмом половљења.

```
public static Texture2D DecreaseResolutionHalfing(Texture2D tex, int size = 300)
{
    if (!tex)
        return null;

    var pixels = tex.GetPixels();

    var sourceHeight = tex.height;
    var sourceWidth = tex.width;

    var targetHeight = sourceHeight / 2;
    var targetWidth = sourceWidth / 2;

    var biggerSize = Mathf.Max(sourceWidth, sourceHeight);
    for (var currSize = biggerSize;  currSize / 2 > size; currSize /= 2)
    {
        for (int y = 0; y < targetHeight; y++)
        {
            int y2 = 2 * y;
            for (int x = 0; x < targetWidth; x++)
            {
                int x2 = 2 * x;

                Color p = (pixels[y2 * sourceWidth + x2]
                           + pixels[y2 * sourceWidth + x2 + 1])
                          / 2;
                Color q = (pixels[(y2 + 1) * sourceWidth + x2]
                           + pixels[(y2 + 1) * sourceWidth + x2 + 1])
                          / 2;
                pixels[y * targetWidth + x] = (p + q) / 2;
            }
        }

        sourceHeight = targetHeight;
        sourceWidth = targetWidth;

        targetHeight/=2;
        targetWidth/=2;
    }

    tex.Reinitialize(sourceWidth, sourceHeight);
    tex.SetPixels(0, 0, sourceWidth, sourceHeight, pixels);
```

```
    tex.Apply();

    return tex;
}
```

Алгоритам итерира кроз матрицу пиксела текстуре по X и Y оси. У свакој итерацији изолује блок од 4 пиксела и рачуна њихову аритметичку средину за сваки канал (црвени, зелени и плави). Тај просек ставља редом назад на текстуру кроз коју итерирамо. Како се никада не враћамо назад кроз ове петље, можемо слободно користити исти простор за нову, смањену слику. Овај процес се понавља док већа димензија слике није мања од 300 пиксела.

На крају функције се смањује величина текстуре и добијена матрица пиксела се поставља на текстуру.

Поменули смо у прошлој секцији да при сваком освежавању екрана проверавамо да ли је учитана слика и у односу на то мењамо стања. Такође смо описали шта се дешава након што корисник позиционира своју слику и врати се у почетно стање. Осим обрађивања притиска платна у моду позиционирања обрађујемо и померање и увеличавање слике.

```
void Update () {
    switch (_mode) {
        case Mode.Edit:
            if (_editMode != EditMode.Idle)
                Debug.Log (_editMode);
            handleIdle ();
            handleTouch ();
            handleMoving ();
            handleZooming ();
            break;
        case Mode.Default:
            handleDefaultMode ();
            break;
    }
}
```

Функција *handleIdle* обрађује догађај када екран није додирнут.

```
private bool handleTouch () {
    if (Input.touchCount != 1 || _editMode == EditMode.Zooming
        || _editMode == EditMode.Moving)
        return false;

    var touch = Input.GetTouch (0);
```

```
    if (isObjectTouched (imagePreview, touch)) {
        if (touch.phase == TouchPhase.Moved) {
            _editMode = EditMode.Moving;
            _touchStartPosition = touch.position;
            _imageStartPosition = imagePreview.transform.position;
        } else if (touch.phase == TouchPhase.Stationary
                && _editMode != EditMode.Moving) {
            _editMode = EditMode.Touched;
        }
    } else if (touch.phase == TouchPhase.Began && !isObjectTouched (image, touch)) {
        changeMode (Mode.Default);
    }

    return true;
}
```

Функција *handleTouch* прави разлику између притиска на објекат и померања објекта. Уколико је објекат притиснут и корисник помера прст, прелази се у мод . Уколико је прст стационаран и није иницран мод за померање, прелази се у мод *притиснут*. Уколико је притиснуто поред слике прелази се на *почетно стање*, што смо обрадили.

```
private bool handleMoving () {
    if (Input.touchCount != 1 || _editMode != EditMode.Moving)
        return false;

    var touch = Input.GetTouch (0);
    _startedEditing = true;

    Vector3 touchPosition = touch.position;
    movePreviewImage (touchPosition, _touchStartPosition);

    return true;
}
```

Уколико смо у функцији *handleTouch* активирали мод померања, ова функција позива *movePreviewImage* са тренутном позицијом и иницијалном позицијом када је померање започело.

```
private void movePreviewImage (
    Vector3 currentPosition, Vector3 startPosition) {
    currentPosition.z = 1.5f;
    startPosition.z = 1.5f;

    var move = Camera.main.ScreenToWorldPoint (currentPosition) -
```

```
        Camera.main.ScreenToWorldPoint (startPosition);
    move.z = 0;

    var newPosition = _imageStartPosition + move;

    imagePreview.transform.position =
        bindImageWhileMoving (newPosition);

    if (Input.touchCount != 1
        || Input.GetTouch (0).phase == TouchPhase.Ended) {
        _editMode = EditMode.Idle;
        _imageStartPosition = imagePreview.transform.position;
    }
}
```

Функција *movePreviewImage* рачуна померај и за ту вредност мења позицију слике. Z оса је вештачки постављена у овом случају, како радимо са објектима који леже на паралелној равни са x и y осом, али служи да мења брзину кретања слике у односу на померање прста по екрану. Уколико корисник покуша да помери слику тако да излази из граница рама, то неће дозволити функција *bindImageWhileMoving*.

```
private Vector3 bindImageWhileMoving (Vector3 newPosition) {
    var leftImageEdge = image.transform.position.x
                    - image.transform.localScale.x / 2;
    var rightImageEdge = image.transform.position.x
                    + image.transform.localScale.x / 2;
    var topImageEdge = image.transform.position.y
                    + image.transform.localScale.y / 2;
    var botImageEdge = image.transform.position.y
                    - image.transform.localScale.y / 2;

    var leftPreviewEdge = newPosition.x
                    - imagePreview.transform.localScale.x / 2;
    var rightPreviewEdge = newPosition.x
                    + imagePreview.transform.localScale.x / 2;
    var topPreviewEdge = newPosition.y
                    + imagePreview.transform.localScale.y / 2;
    var botPreviewEdge = newPosition.y
                    - imagePreview.transform.localScale.y / 2;

    if (leftPreviewEdge > leftImageEdge) {
        newPosition.x += leftImageEdge - leftPreviewEdge;
    } else if (rightPreviewEdge < rightImageEdge) {
        newPosition.x -= rightPreviewEdge - rightImageEdge;
    }

    if (botPreviewEdge > botImageEdge) {
        newPosition.y += botImageEdge - botPreviewEdge;
```

```
    } else if (topPreviewEdge < topImageEdge) {
        newPosition.y -= topPreviewEdge - topImageEdge;
    }

    return newPosition;
}
```

На исти начин као и функција која сече слику, ова функција рачуна ивице платна и оригиналне слике. Уколико ивице оригиналне слике прелазе ивице платна, тако да слика излази из рама, функција враћа слику тако да остане у раму.

Последња функционалност је увеличавање оригиналне слике:

```
private bool handleZooming () {
    if (Input.touchCount != 2)
        return false;

    _editMode = EditMode.Zooming;
    _startedEditing = true;

    var touch1 = Input.GetTouch (0);
    var touch2 = Input.GetTouch (1);

    var touch1PrevPos = touch1.position - touch1.deltaPosition;
    var touch2PrevPos = touch2.position - touch2.deltaPosition;

    float prevTouchDeltaMag =
        (touch1PrevPos - touch2PrevPos).magnitude;
    float touchDeltaMag =
        (touch1.position - touch2.position).magnitude;

    float aspect = imagePreview.transform.localScale.x
            / imagePreview.transform.localScale.y;

    float deltaMagDiff = touchDeltaMag - prevTouchDeltaMag;

    float deltaX = deltaMagDiff * aspect * .01f;
    float deltaY = deltaMagDiff * .01f;

    if (imagePreview.transform.localScale.x + deltaX > 0
        && imagePreview.transform.localScale.y + deltaY > 0) {
        imagePreview.transform.localScale = new Vector3 (
            imagePreview.transform.localScale.x + deltaX,
            imagePreview.transform.localScale.y + deltaY,
            imagePreview.transform.localScale.z
        );
    }

    return true;
```

```
}
```

Уколико не постоје тачно два улаза, функција неће ништа урадити. Уколико постоје, *Unity* препознаје мале помераје између два догађаја (делта позиција). Рачунамо која је разлика између иницијалног додира екрана и помераја и добијамо за колико треба да повећамо слику. Како слику морамо повећати по обе осе, чувамо њихов однос и множимо претходно израчунатим фактором увећања. Додатно множимо са 0,1 да би се слика спорије повећавала (константа добијена практичним тестирањем).

Када престане акција зумирања и улази не буду били регистровани, позваће се горе поменута функција *handleIdle*:

```
private bool handleIdle () {
    if (Input.touchCount != 0)
        return false;

    // Idle is the only state application can go from zooming
    bindImageWhileZooming ();

    if (_editMode == EditMode.Touched && _startedEditing)
        changeMode (Mode.Default);

    _editMode = EditMode.Idle;
    return true;
}
```

Како приликом увеличавања нисмо ограничили величину слике, на првом следећем освежавању екрана, позива се горе описана функција *handleIdle*, која позива *bindImageWhileZooming*.

```
private void bindImageWhileZooming () {
    var horizontalScale = image.transform.localScale.x
                / imagePreview.transform.localScale.x;
    var verticalScale = image.transform.localScale.y
                / imagePreview.transform.localScale.y;

    var scale = Mathf.Max (horizontalScale, verticalScale);
    if (scale > 1) {
        imagePreview.transform.localScale = new Vector3 (
            imagePreview.transform.localScale.x * scale,
            imagePreview.transform.localScale.y * scale,
            imagePreview.transform.localScale.z
        );
    }
```

```
    imagePreview.transform.position =
        bindImageWhileMoving(imagePreview.transform.position);
}
```

Слично поменутој *bindImageWhileMoving* функцији, ова функција повећава слику тако што рачуна колико је пута рам већи од новонастале слике (свака оса појединачно). Потом узимамо већу од две вредности и за толико повећавамо слику.

Коначно, позивамо опет функцију *bindImageWhileMoving* како би цела слика остала у раму.

Алгоритми за модификовање слике

Генерисање дугмади за филтере, приказаних на слици 4.4, извршава се динамички коришћењем *Unity* шаблона. За сваку инстанцу објекта тог шаблона потребно је проследити одређене параметре филтера, што се може видети у следећој функцији:

```
void initFilters()
{
    if (StaticImages.adjustmentsFiltersCount == 0)
    {
        StaticImages.adjustmentsFilters = new AdjustmentsFilter[7];
        AddFilter("Brightness", "brightness", (tex, val) =>
         (FilterFunctions.Brightness(tex, (int)val)), -130, 130);
        AddFilter("Contrast", "contrast", (tex, val) =>
         (FilterFunctions.Contrast(tex, (int)val)), -30, 30);
        AddFilter("Warmth", "warmth", (tex, val) =>
         (FilterFunctions.Warmth(tex, (int)val)), -60, 60);
        AddFilter("Saturation", "saturation", (tex, val) =>
         (FilterFunctions.Saturation(tex, val)), .4f, 1.6f, 1);
        AddFilter("Highlights", "highlights", (tex, val) =>
         (FilterFunctions.Highlights(tex, val)), -1, 1);
        AddFilter("Shadows", "shadows", (tex, val) =>
         (FilterFunctions.Shadows(tex, val)), -1, 1);
        AddFilter("BW", "bw", (tex, val) =>
         (FilterFunctions.BW(tex, val)), 0, 1, 0, true);
    }

    for (int i = 0; i < StaticImages.adjustmentsFiltersCount; i++)
    {
        AdjustmentsFilter currentFilter =
            StaticImages.adjustmentsFilters[i];

        currentFilter.addFilterToGallery();
```

```
        currentFilter.filterImage.onClick.AddListener(() =>
        {
            bindAdjustmentsSlider(currentFilter);
            Gallery.state = Constants.States.SingleAdjustment;
        });
    }
}
```

У овој петљи сваки филтер се додаје на сцену функцијом *addFilterToGallery*:

```
public void addFilterToGallery()
    {
        filterImage = UnityEngine.Object.Instantiate(
            filterImagePrefab, Vector3.zero, Quaternion.identity);
        filterImage.transform.SetParent(container.transform);

        string path = "Images/" + this.filterFile;
        Texture2D imgTex = Resources.Load<Texture2D>(path);

        Sprite sprite = Sprite.Create(
            imgTex,
            new Rect(0, 0, imgTex.width, imgTex.height),
            new Vector2(0.5f, 0.5f));

        filterImage.GetComponent<Image>().sprite = sprite;

        filterImage.transform.Find("Text")
        .GetComponent<Text>().text =
            this.filterName;
    }
```

Unity инстанцира шаблон објекта, додељује му родитеља, текст и иконицу. Потом у претходној функцији се региструје нова функција која се покреће кликом на дугме:

```
currentFilter.filterImage.onClick.AddListener(() =>
    {
        bindAdjustmentsSlider(currentFilter);
        Gallery.state = Constants.States.SingleAdjustment;
    });
```

Функција поставља стање галерије на подешавање филтера и везује вредности клизача (енг. *slider*) на тренутне вредности филтера (тренутна, минимална, максимална и подазумевана вредност, као и информација да ли се користе континуалне или дискретне вредности). Клизач има само једну

инстанцу и мења вредност сваки пут кад се учита нови филтер:

```
private void bindAdjustmentsSlider(AdjustmentsFilter filter)
{
    activeSlider = Gallery_UIHandler.SliderFilter.adjustments;

    currentFilter = filter;

    var filterValue = currentFilter.value;

    slider.minValue = currentFilter.minValue;
    slider.maxValue = currentFilter.maxValue;

    slider.value = filterValue;
    oldSliderValue = filterValue;

    slider.wholeNumbers = currentFilter.wholeNumbers;

    UI_Static.showUIPanel(sliderPanel);
}
```

Везивање клизача за филтер коначно приказује клизач на дну екрана са стањем тренутног филтера. На свако померање клизача бројач креће да одбројава 15 стотинки. Када то време истекне филтер ће се применити на слику, позивом функције поменуте раније:

```
TextureTools.processFinalImageAsync(
    imagePreview, image, true,
    TextureTools.processImageType.adjustments);
```

Овог пута ово је цела функција:

```
 public static async Task<Texture2D> processFinalImageAsync(
    GameObject imagePreview,
    GameObject image,
    bool updateObjects = true,
    processImageType type = processImageType.crop)
{
    if(StaticImages.initImageTexture == null)
        return null;

    Texture2D tex = null;

    switch (type)
    {
        case processImageType.crop:
            tex = null;
```

```
            break;
        case processImageType.artFilters:
            tex = savedCropTexture;
            break;
        case processImageType.adjustments:
            tex = savedArtTexture;
            break;
    }

    if (type != processImageType.artFilters
        && type != processImageType.adjustments) {
        tex = ResampleAndCrop(imagePreview, image);
        tex = DecreaseResolutionHalfing(tex);
        savedCropTexture = tex;
    }

    if (type != processImageType.adjustments) {
        if (HandleArtFilters.selectedFilter != null)
        {
            tex = await
            HandleArtFilters.selectedFilter.ApplyFilter(tex);
        }
        savedArtTexture = tex;
    }

    tex = await Gallery_UIHandler.ApplyFilters(tex);

    if (updateObjects)
    {
        image.GetComponent<Renderer>().material.mainTexture = tex;
        StaticImages.finalImageTexture = tex;
    }

    return tex;
}
```

Финална слика је производ померања и сечења слике, основних подешавања слике и уметничких филтера (више речи о њима касније). Код сечења и померања слике не можемо кеширати ништа, зато што филтере не извршавамо на оригиналној слици, већ на исеченој слици смањене резолуције. Ипак, кад примењујемо филтере можемо искористити исечену слику, коју чувамо у променљивој *savedCropTexture*. Такође, уметнички филтер можемо сачувати тако да се он не извршава при свакој промени подешавања. То је изузетно битно пошто је очекивање да на промену клизача апликација брзо одреагује, а уметничким филтерима треба више десетина пута дуже да се изврше.

Вратимо се на примену филтера. Када истекне бројач, позваће се функција *processFinalImageAsync* која позива *Gallery_ UIHandler.ApplyFilters*:

```
public static async Task<Texture2D> ApplyFilters(Texture2D tex)
{
    var local = FilterFunctions.TextureToLocal(tex);
    for (int i = 0; i < StaticImages.adjustmentsFiltersCount; i++)
    {
        AdjustmentsFilter filter =
            StaticImages.adjustmentsFilters[i];
        local = await
        Task.Run(() => filter.ApplyFilterLocaly(local, filter.value));
    }
    return FilterFunctions.LocalToTexture(local);
}
```

Ова функција се покреће асинхроно, како не би блокирала апликацију. Функција пролази редом кроз могуће филтере и примењује их на слику.

Свака од наведених функција манипулише сваким пикселом засебно (независно од околине).

Узмимо пример контраста. Нека C буде вредност контраста, а T задат параметар - праг (енг. *treshold*). Формула за израчунавања контраста је:
$C = ((100.0 + T)/100.0)^2$

Помоћу те променљиве можемо променити црвени, зелени и плави канал сваког пиксела користећи следећу формулу:
$R = ((((R'/255) - 0.5) * C) + 0.5) * 255$
$G = ((((G'/255) - 0.5) * C) + 0.5) * 255$
$B = ((((B'/255) - 0.5) * C) + 0.5) * 255,$
где су RGB нове вредности канала пиксела, а $R'G'B'$ старе.

У коду то изгледа овако:

```
public static LocalTexture Contrast(LocalTexture inputImage,
                                    int threshold)
{
    int imageWidth = inputImage.width;
    int imageHeight = inputImage.height;

    var pixelBuffer = inputImage.pixels;

    Color32[] resultBuffer = new Color32[pixelBuffer.Length];

    var contrast = Math.Pow((100.0 + threshold) / 100.0, 2);

    for (int y = 0; y < imageHeight; y++)
    {
        for (int x = 0; x < imageWidth; x++)
        {
            var pixelOffset = y * imageWidth + x;
```

```
            var oldColor = pixelBuffer[pixelOffset];

            var red = ((((oldColor.r / 255.0) - 0.5)
                        * contrast) + 0.5) * 255.0;
            var green = ((((oldColor.g / 255.0) - 0.5)
                        * contrast) + 0.5) * 255.0;
            var blue = ((((oldColor.b / 255.0) - 0.5)
                        * contrast) + 0.5) * 255.0;

            resultBuffer[pixelOffset].r = ClipByte(red);
            resultBuffer[pixelOffset].g = ClipByte(green);
            resultBuffer[pixelOffset].b = ClipByte(blue);
        }
    }

    inputImage.pixels = resultBuffer;

    return inputImage;
}
```

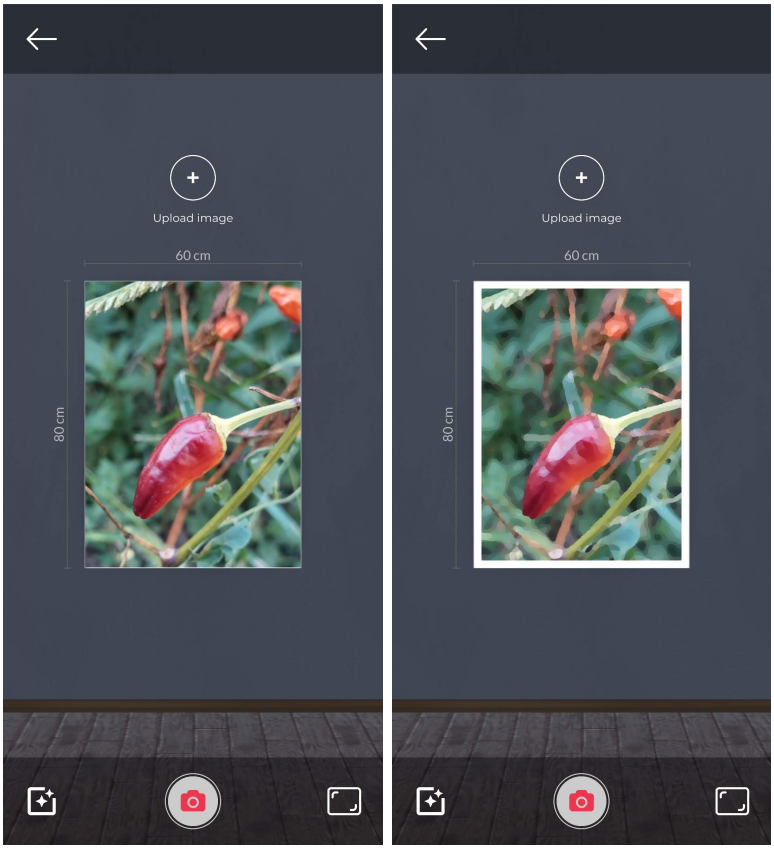
Функција *ClipByte* има улогу да вредност пиксела буде у опсегу [0,255]. На сличан начин су имплементиране и остале функције.

Уметнички филтери

За разлику од претходних филтера, уметнички филтери немају клизач због њихове комплексности и времена извршавања, већ су им параметри предефинисани и примењују се кликом на дугме (слика 4.5). Такође, за разлику од претходних филтера који функционишу у композицији, можемо активирати тачно један уметнички филтер. Притиском на било које од три дугмета бира се жељени филтер и позива већ позната функција *processFinalImageAsync*, која позива један од поменутих алгоритама, што ћемо детаљније описати у наставку.

Уље на платну

Слика 5.5: Уметнички филтери



Вредност сваког пиксела применом овог алгоритма не зависи само од параметара функције и иницијалне вредности пиксела, већ и његове околине. За израчунавање вредности произвољног пиксела (X, Y), потребно је анализирати све пикселе у распону $(X - Radius, Y - Radius)$ до $(X + Radius, Y + Radius)$, где променљива $Radius$ представља пречник околине и учитава се као параметар функције. Сваком пикселу у овој околини се додељује интензитет, који се рачуна на следећи начин: $((R + G + B) * levels)/255$, где променљива $levels$ одређује колико ће различитих група интензитета бити. За сваку групу интензитета меримо колико пиксела из околине јој припада и бирамо групу са највећим бројем пиксела. Осим што сумирамо број пиксела по групи, такође сумирамо и вредности црених, зелених и плавих канала. Следећи код се извршава у петљи која итерира све суседе пиксела на задатом максималном растојању (извршава се за сваког суседа):

```
currentIntensity = (int)Math.Round(((double)
            (pixelBuffer[calcOffset].r +
            pixelBuffer[calcOffset].g +
            pixelBuffer[calcOffset].b) / 3.0 *
            (levels)) / 255.0);


intensityBin[currentIntensity] += 1;
redBin[currentIntensity] += pixelBuffer[calcOffset].r;
greenBin[currentIntensity] += pixelBuffer[calcOffset].g;
blueBin[currentIntensity] += pixelBuffer[calcOffset].b;


if (intensityBin[currentIntensity] > maxIntensity)
{
    maxIntensity = intensityBin[currentIntensity];
}
```

Коначна вредност пиксела је количник сваког канала (црвеног, зеленог и плавог) и интензитета у групи која има највише представника [14]:

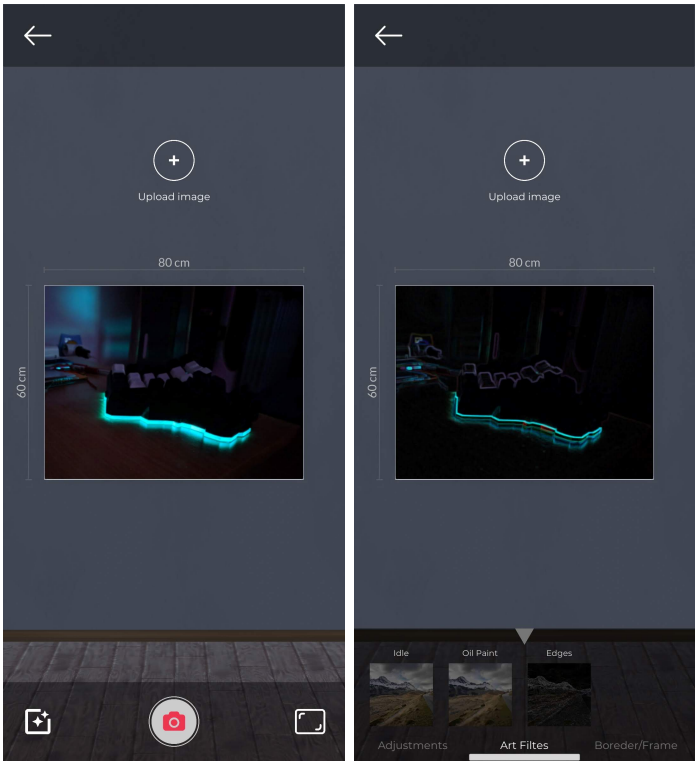
```
blue = blueBin[maxIndex] / maxIntensity;
green = greenBin[maxIndex] / maxIntensity;
red = redBin[maxIndex] / maxIntensity;

resultBuffer[byteOffset].r = ClipByte(red);
resultBuffer[byteOffset].g = ClipByte(green);
resultBuffer[byteOffset].b = ClipByte(blue);
```

Комплексност овог алгоритма је у томе што је неопходно за сваки пиксел обрађивати његове суседе, па у односу на пречник окружења добијамо различита времена извршавања. За потребе ове апликације посматрамо окружење од 9 пиксела, те је овај алгоритам око 81 пут комплекснији од горе поменутих филтера.

Детекција ивица

Слика 5.6: Уметнички филтери



Овај алгоритам ради по сличном принципу као и претходни. Итерира се кроз све пикселе, потом кроз њихова окружења у задатом пречнику. За свако окружење се налази најсветлија и најтамнија вредност свих канала (црвеног, зеленог и плавог) и вредност пиксела постаје њихова разлика. Како смо у претходним корацима смањили резолуцију слике до 300 пиксела по оси, за потребе овог алгоритма користимо пречник од 3 пиксела, што нам даје жељене резултате.

Следећи код се извршава у петљи која пролази све суседе пиксела на задатом максималном растојању:

```
minBlue = Math.Min(pixelBuffer[calcOffset].b, minBlue);
maxBlue = Math.Max(pixelBuffer[calcOffset].b, maxBlue);

minGreen = Math.Min(pixelBuffer[calcOffset].g, minGreen);
maxGreen = Math.Max(pixelBuffer[calcOffset].g, maxGreen);

minRed = Math.Min(pixelBuffer[calcOffset].r, minRed);
maxRed = Math.Max(pixelBuffer[calcOffset].r, maxRed);
```

Након израчунавања оптимума, њихове разлике постају коначна вредност пиксела [15].

```
resultBuffer[byteOffset].r = (byte)(maxRed - minRed);
resultBuffer[byteOffset].g = (byte)(maxGreen - minGreen);
resultBuffer[byteOffset].b = (byte)(maxBlue - minBlue);
```

Рам и оквир

Рам и оквир имају нешто другачију имплементацију зато што се не односе на само платно, већ су засебни *Unity* објекти.

Оквир

Оквир је објекат који у хијерархијском стаблу испод себе има 4 објекта за сваку ивицу.

Слично као са филтерима, иницијална, минимална, максимална и тренутна ширина се прослеђују клизачу. На промену вредности се покреће функција:

```
public static void changeBorderSize(
    Vector2 dimensions, ref GameObject border, float borderWidth)
{
    float x;
    float y;

    var borderTop = border.transform.Find("BorderTop");
    var borderBot = border.transform.Find("BorderBot");
    var borderLeft = border.transform.Find("BorderLeft");
    var borderRight = border.transform.Find("BorderRight");

    x = dimensions.x;
    borderTop.transform.localScale = new Vector3(
        x, borderWidth, borderTop.transform.localScale.z);
    borderBot.transform.localScale = new Vector3(
        x, borderWidth, borderBot.transform.localScale.z);

    y = dimensions.y;
    borderLeft.transform.localScale = new Vector3(
        borderWidth, y, borderLeft.transform.localScale.z);
    borderRight.transform.localScale = new Vector3(
        borderWidth, y, borderRight.transform.localScale.z);

    y = (dimensions.y - borderTop.transform.localScale.y) / 2;
    borderTop.transform.position = new Vector3(
```

```
            borderTop.transform.position.x,
            y,
            borderTop.transform.position.z);

        y = -y;
        borderBot.transform.position = new Vector3(
            borderBot.transform.position.x,
            y,
            borderBot.transform.position.z);

        x = (dimensions.x - borderLeft.transform.localScale.x) / 2;
        borderLeft.transform.position = new Vector3(
            x,
            borderLeft.transform.position.y,
            borderLeft.transform.position.z);

        x = -x;
        borderRight.transform.position = new Vector3(
            x,
            borderRight.transform.position.y,
            borderRight.transform.position.z);
}
```

Аргументи функције су димензије слике, оквир који садржи горњу, доњу леву и десну страницу, као и ширина оквира која је једнака вредности клизача. Повећавање величине сваког дела оквира је једноставно и постиже се само прослеђивањем ширине оквира X или Y параметру у зависности од тога да ли је хоризонталан или вертикалан део.

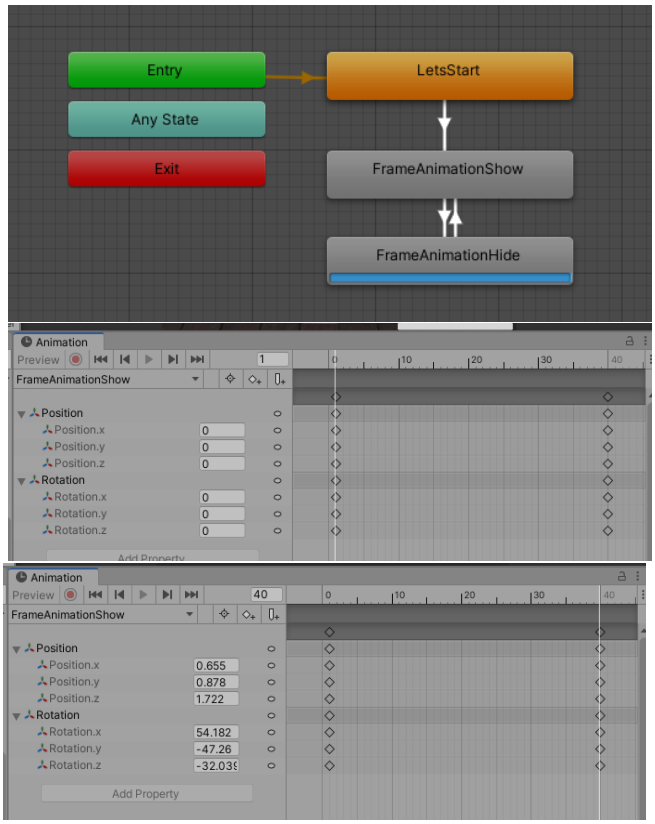
Само то би повећало оквир ван рама, тако да је потребно и померити делове ка средини слике за половину ширине, како би се ивице оквира поклапале са ивицом рама.

Рам

Рам се бира на исти начин као и уметнички филтери, притиском на жељено дугме. Оно што је занимљиво је то да улазак у опцију за промену рама мења перспективу камере да бисмо могли боље да видимо карактеристике слике.

Развојно окружење нам пружа подршку за анимирање објеката. У овом случају је то искоришћено да се анимира позиција и ротација камере.

Слика 5.7: Анимација



Алат за управљање анимацијама је једноставан за употребу. На првој слици (слика 5.7) је приказана шема промена стања која се касније иницира кроз код. Овим се обезбеђује да уколико прекинемо анимацију и вратимо на претходно стање, анимација заиста бива прекинута у том тренутку и враћање у претходно стање почиње из те тачке. Друге две слике приказују подешавања камере у кључним тачкама. Њих може бити произвољно много, али за потребе ове анимације дефинисане су прва и последња кључна тачка.

Следеће три функције контролишу анимацију позивањем одговарајуће методе *frameAnimationController* објекта, који је дефинисан у развојном окружењу *Unity*. Осим тога, потребно је приказати односно склонити секцију за одабир рама и мере величине рама са екрана. Све то је приказано у наредном коду:

```
private void showFramePanel() {
        Gallery.state = Constants.States.FrameMaterial;
        UI_Static.showUIPanel(frameMaterialPanel);
        frameAnimationController.SetBool("frameZoom", true);
        galleryScript.hideLabels();
```

```
    }

 public void handleFrameMaterialCancel()
{
    changeFrameMaterial(oldFrameMaterial);
    UI_Static.hideUIPanel(frameMaterialPanel);
    frameAnimationController.SetBool("frameZoom", false);
    Invoke("showLabels", .5f);
}

private void handleFrameMaterialDone()
{
    oldFrameMaterial = currentFrameMaterial;
    UI_Static.hideUIPanel(frameMaterialPanel);
    frameAnimationController.SetBool("frameZoom", false);
    Invoke("showLabels", .5f);
}
```

5.3 Проширена стварност

Сцена проширене стварности се покреће кликом на иконицу камере у галерији. Ова сцена обрађује два сценарија, када је слика постављена на зид и када није:

```
void Update () {
    if (!_isSceneLoaded)
        return;

    if (!_isImagePlaced) {
        UpdateIndicatorPlacement ();
        UpdateIndicator ();
    } else {
        handleMoving ();
        HideIndicator ();
    }
}
```

Тренутно обрађујемо понашање индикатора, тако да нас занима сценарио када слика и даље није постављена. У том случају, сваки пут када је освежена слика зову се две методе: *UpdateIndicatorPlacement* и *UpdateIndicator*.

У функцији *UpdateIndicatorPlacement* се први пут сусрећемо са библиотеком за рад са проширеном стварношћу:

```
private void UpdateIndicatorPlacement () {
```

```
    var screenCenter = arCamera.ViewportToScreenPoint
        (new Vector3 (0.5f, 0.5f));
    var hits = new List<ARRaycastHit> ();
    _arRaycastManager.Raycast(
        screenCenter,
        hits,
        UnityEngine.XR.ARSubsystems.TrackableType.Planes);

    _placementPoseIsValid = hits.Count > 0;
    if (_placementPoseIsValid) {
        _placementPose = hits[0].pose;

        var cameraForward = arCamera.transform.forward;
        var cameraBearing = new Vector3(
            cameraForward.x, 0, cameraForward.z).normalized;
        _placementPose.rotation =
            Quaternion.LookRotation (cameraBearing);
    }
}
```

Објекат $arRaycastManager$ нам слично као у галерији где смо проверавали да ли је притиснута слика, пружа опцију да видимо да ли зрак из камере има пресек са равни у реалном окружењу које детектује камера. Библиотека је направљена тако да детектује хоризонталне равни, зато што је то много чешћи случај од вертикалних. Нама је потребна детекција зида, а не пода, али тај проблем смо решили у наставку.

Након детектовања пода, индикатор ће се померити на положај пресека и ротирати према камери, задржавајући z осу управно на раван пода.

Метода $UpdateIndicator$ контролише позицију самог индикатора:

```
private void UpdateIndicator () {
    if (_placementPoseIsValid) {
        setHorizontalIndicatorValid ();

        var cameraRotation =
            arCamera.transform.rotation.eulerAngles;
        var indicatorRotation =
            placementIndicator.transform.rotation.eulerAngles;
        placementIndicator.transform.SetPositionAndRotation (
            _placementPose.position, _placementPose.rotation);
    } else {
        setIndicatorInvalid ();

        var indicatorRotation = arCamera.transform.rotation;

        placementIndicator.transform.SetPositionAndRotation (
            arCamera.transform.position +
```

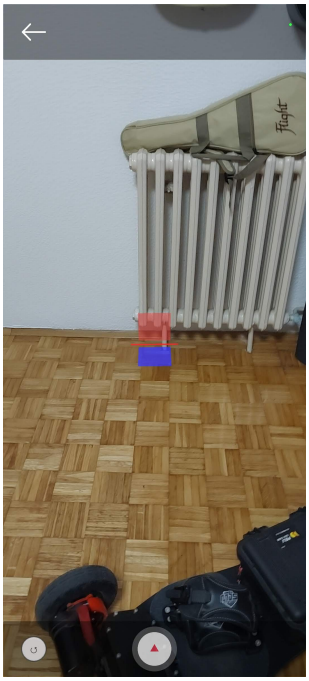
```
                arCamera.transform.forward * 1.5f,
            indicatorRotation);
        placementIndicator.transform.Rotate(
            transform.right, -45f, Space.Self);

    }
}
```

Док под није детектован, индикатор ће бити ротиран за 45 степени према камери, како би се видео и хоризонталан и вертикалан део (слика 4.8).

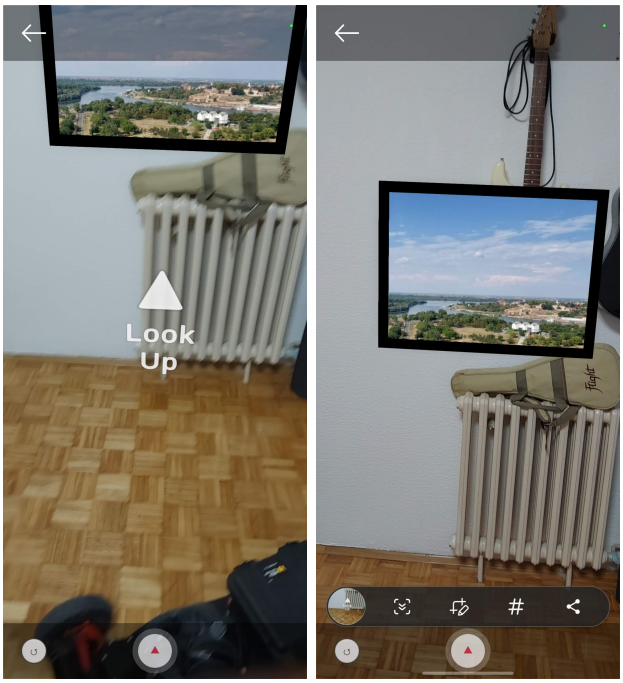
Како бисмо детектовали зид и поставили слику, морамо се ослонити на корисника. Пошто библиотека нема директну подршку за детектовање вертикалних равни, а већ смо детектовали под, довољно је да индикатор ставимо до ивице зида и притиснемо дугме на дну екрана (слика 5.8).

Слика 5.8: Позиционирање индикатора на ивицу између пода и зида



Притиском дугмета појавиће се слика на зиду и стајаће у висини камере. Како је камера фокусирана на ивицу пода и зида, такође ће се појавити текст који обавештава корисника о локацији слике (слика 5.9).

Слика 5.9: Постављање слике на зид



Слика се поставља покретањем следеће функције:

```
private void PlaceAnImage () {
    if (_isImagePlaced || !_placementPoseIsValid)
        return;

    var newPosition = _placementPose.position;
    newPosition.y = arCamera.transform.position.y;

    _image = Instantiate(
        objectToPlace, newPosition, _placementPose.rotation);
    _imageStartPosition = _image.transform.position;

    lookUp.transform.position = _placementPose.position;
    lookUp.transform.rotation = _placementPose.rotation;
    lookUp.SetActive (true);
    StartCoroutine (HideLookupAfterSeconds ());

    _isImagePlaced = true;

    _wall = new Plane(
        _image.transform.forward, _image.transform.position);
}
```

Функција инстанцира текст који показује где се слика налази и корутину која сакрива тај текст након неколико секунди. Сама слика се позиционира

на X і Y координату индикатора на поду, и на Z координату камере (односно мобилног телефона).

Након позиционирања слике при сваком освежавању екрана покрећемо функцију за померање слике:

```
private bool handleMoving () {
    if (Input.touchCount != 1)
        return false;

    var touch = Input.GetTouch (0);

    if (IsPointerOverUIObject ())
        return false;

    moveToPoint (touch.position);

    return true;
}
```

Ова функција ће притиском на екран померити слику на жељену локацију.

```
private void moveToPoint (Vector3 touchPosition) {
    Ray ray = arCamera.ScreenPointToRay (touchPosition);
    float enter = 0.0f;

    if (_wall.Raycast (ray, out enter)) {
        Vector3 hitPoint = ray.GetPoint (enter);
        _image.transform.position = hitPoint;
    }
}
```

Глава 6

Закључак

У овом раду приказана је апликација која демонстрира учитавање слике са *Android* уређаја, њено уређивање и постављање у проширеној стварности.

Приказана апликација осим што решава реалан проблем, показује пример на основу ког се може направити мноштво апликација које нису нужно игре у развојном окружењу *Unity*. Кроз овај рад обрађени су многи сценарији и методе прављења *Unity* апликација као што су рад са сценама, објектима, анимацијама, шаблонима и свакако проширеном стварношћу. Такође су приказана ограничења израде апликација за преносиве уређаје и методе оптимизације за превазилажење истих.

Проширена стварност као област је до скоро већински сматрана видом забаве, али доступношћу технологије и унапређивањем алата, почиње да се пробија и у едукативне и комерцијалне сфере. Током пандемије, док смо имали ограничено кретање, људи широм света су радили и студирали од куће и у томе им је доста помогао софтвер за комуникацију. То дешавање је променило многе индустрије, тако да је много људи наставило да ради и студира са различитих локација. Сви ови догађаји су направили добру базу за развијање апликација базираних на проширеној стварности, а најава *metaverse*-а [16] и имплементација *Lidar* сензора у нове телефоне који прецизно рачунају рељеф објеката потврђују да време виртуелне и проширене стварности тек долази.

Апликација даље може бити унапређена додавањем нових уметничких филтера, прилагођавањем *iOS* оперативном систему, додавањем подршке за рад са произвољним димензијама и произвољним бројем слика.

Литература

[1] Definition of virtual reality (датум приступа: 13.8.2022.). https://www.dictionary.com/browse/virtual-reality.

[2] Hegde, Naveen (11 June 2021). „What is Augmented Reality". ANT Developers.

[3] Johnson, Joel (1901). „The Master Key". L. Frank Baum envisions augmented reality glasses in Mote Beam 10 September 2012.

[4] Martirosov, Sergo Kopecek, Pavel (2017). Virtual Reality and its Influence on Training and Education - Literature Review, Proceedings of the 28th DAAAM International Symposium, pp.0708- 0717, B. Katalinic (Ed.), Published by DAAAM International, ISBN 978-3-902734-11-2, ISSN 1726-9679, Vienna, Austria DOI: 10.2507/28th.daaam.proceedings.100.

[5] IBM Technical Disclosure Bulletin (1 March 1987). „Absolute Display Window Mouse/Mice" (context abstract only).

[6] PlayStation.Blog (8 April 2011). „From EyeToy to NGP: PlayStation's Augmented Reality Legacy".

[7] Louis B. Rosenberg (1992). „The Use of Virtual Fixtures As Perceptual Overlays to Enhance Operator Performance in Remote Environments." Technical Report AL-TR-0089, USAF Armstrong Laboratory (AFRL), Wright-Patterson AFB OH.

[8] Digital Trends (26 July 2016). „HoloLens concept lets you control your smart home via augmented reality".

[9] „Mojo Vision's AR contact lenses are very cool, but many questions remain" (датум приступа: 21.8.2022.). TechCrunch.

[10] Luetzenburg, G., Kroon, A. Bjørk, A.A (2021). Evaluation of the Apple iPhone 12 Pro LiDAR for an Application in Geosciences. Sci Rep 11, 22221 https://doi.org/10.1038/s41598-021-01763-9.

[11] Dana H. Ballard; Christopher M. Brown (1982). Computer Vision. Prentice Hall. ISBN 978-0-13-165316-0.

[12] Denham, Jess (July 12, 2016). „Pokémon Go has won the praise of gender fluid gamers".

[13] Unity Gaming Report (2022). https://create.unity.com/gaming-report-2022.

[14] Code Project, Oil Paint Effect: Implementation of Oil Painting Effect on an Image (датум приступа: 20.8.2022.). https://www.codeproject.com/Articles/471994/OilPaintEffect.

[15] SoftwareByDefault, C How to: Min/Max Edge Detection (датум приступа: 20.8.2022.). https://softwarebydefault.com/2015/08/09/minmax-edge-detection/.

[16] Lik-Hang Lee, Tristan Braud, Pengyuan Zhou, Lin Wang, Dianlei Xu, Zijun Lin, Abhishek Kumar, Carlos Bermejo, Pan Hui (2021). All One Needs to Know about Metaverse: A Complete Survey on Technological Singularity, Virtual Ecosystem, and Research Agenda arXiv:2110.05352v3 [cs.CY] https://doi.org/10.48550/arXiv.2110.05352.

Биографија аутора

Лука Живановић (*Београд, 1. јул 1995*) завршио је основну и средњу школу у Београду. 2014. године уписује информатички смер на Математичком факултету у Београду, који завршава 2017. године, када уписује мастер студије на смеру информатика. За време студија учествује у програмерским такмичењима, међу којима је значајније *Beyond Hackathon*, одржано марта 2017. године у Атини, на ком са тимом осваја друго место у конкуренцији од 38 тимова. У октобру 2018. се запошљава као сарадник у настави на матичном факултету на коме ради два семестра. Од фебруара до августа 2019. године такође ради као предавач веб програмирања у школи „*Code by Comtrade*". Након тога се априла 2020. године запошљава као софтверски инжињер у Мајкрософт развојном центру у Београду, где и даље ради.