SVEUČILIŠTE U RIJECI
TEHNIČKI FAKULTET

Diplomski studij računalstva

QML, ubuntu phone, jolla, linux

Student: Josip Puniš,

Mentor: Jerko Škifić

Rijeka, kolovoz 2016.

SADRŽAJ

1. Uvod

Tema ovog rada je izrada aplikacije koristeći QML (*Qt Meta Language*). Osnovni uvjet za aplikaciju je taj da se može izvršavati na mobilnim uređajima koji imaju Ubuntu Touch operacijski sustav i na računalima neovisno o operacijskom sustavu. Upravo iz tog razloga (platformska nezavisnost aplikacije) korišten je QML. Osim samog QML u razvoju su korišteni OpenCV i dlib. Razlog njihovog korištenja je sama ideja aplikacije. OpenCV je besplatna biblioteka koja sadrži veliki broj funkcija u području računalnog vida dok dlib je biblioteka u kojoj su implementirani najkorišteniji algoritmi strojnog učenja.

Aplikacija koje je razvijena naziva se FaceSwap i njezina jedina funkcija je zamjena lica osobe s jedne slike na drugu sliku.

Sama logika detekcije lica napravljena je pomoću OpenCV i dlib te je sav kod napisan u C++. Taj dio aplikacije smješten je u poseban priključak (eng. plugin). QML korišten je u kreiranju sučelja aplikacije.

2. Razvojni alati

Kao šta je već u uvodu rečeno za razvoj ove aplikacije korištena su tri alata a to su: QML, OpenCV i dlib. Potrebno se svakog od njih opisati i objasniti prije nego se krene s opisom aplikacije.

2.1. QML

QML (*Qt Meta Language)* je deklarativan programski jezik koji služi za definiranje izgleda i ponašanja grafičkih sučelja. QML također je skripti jezik te nije potrebno kompajlirati kod prije izvršavanja. Unutar QML grafičko sučelje prikazano je kao stablo objekta u kojem svaki objekt ima određena svojstva. QML je pripada u Qt *framework-u*. Svi QML dokumenti ima ekstenziju .qml.

QML osim platformske nezavisnosti karakterizira ga još veoma jednostavna i čitka sintaksa i mogućnost korištenja *inline* JavaScripta-a. Osim JavaScripta QML može se još koristi u kombinaciji sa C++. QML-om definira se grafičko sučelje a sama logika pomoću C++. Također je moguće unutar C++ koda manipulirati QML objektima. U praksi se pojavljuju hibridne aplikacije QML, C++ i JavaScripta.

Unutar Qt postoji modul *Qt QML* u kojoj je opisan QML kao jezik i QML interpreter koji izvodi naredbe. Ukoliko se želi koristiti QML obavezno se mora dodati taj modul u kod. To se vrši naredbama *import QtQml 2.0* ili *#include <QtQml>* ovisno da li se koristi QML ili C++. Osim u kodu mora se dodati mora se dodati i u .pro datotekama naredba *QT +=qml* (naravno, *a*ko se koristi *qmake* a ne *cmake*) kako bi se mogla aplikacija vezati s tim modulom. Drugi bitan modul je *Qt Quick*. Unutar tog modula definirani su bazični objekti za kreiranje grafičkih sučelja.

2.1.1. QML sintaksa

QML karakterizira ga jednostavna i čitka sintaksa veoma slična JSON-u (*JavaScript Object Notation*). Na slici 1 može se vidjeti jedan jednostavan kod napisan u QML.

Svaki QML dokument na početku ima jedan ili više importa. Import se može napisati u obliku puta do datoteke (tada se koriste navodnici) ili tako da se navede ime imenika i njegova verzija (prikazano na slici 1.).

Nakon importa može se krenuti s kreiranje objekta. Objekti se kreiraju tako da se prvo navodi ime objekta te nakon toga unutar tijela, koji je omeđeno parom vitičastih zagrada ({}) , navode atributi. Svaki objekt ima poseban set atributa. Vrijednosti atributima pridodajemo tako

```
import QtQuick 2.0

 Rectangle {
     id: canvas
     width: 250
     height: 200
     color: "blue"

     Image {
         id: logo
         source: "pics/logo.png"
         anchors.centerIn: parent
         x: canvas.height / 5
     }
 }
```

Slika 1. QML sintaksa

da nakon imena atributa slijedi dvotočka (:) te na kraju vrijednosti. U istom redu možemo definirati proizvoljan broj atributa. U tom slučaju svaki atribut mora biti odvojen točkom-zarezom (;). Ukoliko se nalazi samo jedan atribut u redu možemo izostaviti točku-zarez. Osim atributa unutar objekta se može kreirati novi objekt.

Postoji nepisana konvencija kojeg se pridržavaju programeri. Na početku se definira id atribut, nakon njega idu *property* iliti varijable, signali, JavaScript funkcije, ostali atributi objekta, unutarnji objekti te na kraju definiraju se stanja i transakcije.

Postoje dvije vrste komentara i to su one standardne koje postoje i u ostalim jezicima. Jednoredne označavaju se sa // a više redne sa /* */.

2.1.2. QML tipovi podataka

Postoje 3 vrste tipova podataka. Prva vrsta su primitivni tipovi a to su cijeli brojevi (integeri), decimalni brojevi s dvostrukom preciznošću (*float*), *string*-ovi i *bool* vrijednosti. Sljedeća vrsta su JavaScript tipovi. To su tipovi koji su karakteristični za JavaScript programe. Bilo koji JavaScript tip može se koristiti unutar QML pomoću generičkog tipa *var*, primjer *property var niz: new Array();*

Posljednja vrsta podataka su QML objekti. Svaki taj objekt naslijeđen je od *QtObject* objekta. Prije korištenja bilo kojeg objekta mora se importirati odgovarajući modul. Kao šta je prije rečeno unutar *QtQuick-a* definirana je većina tih objekta.

Svaki QML objekt ima set atributa. Svaki taj atribut predstavlja određeno svojstvo tog objekta, npr. boja, širina, visina, id, … Atribute možemo podijeliti u šest kategorija.

U prvoj kategoriji nalazi se samo jedan atribut i to je id. Id predstavlja jedinstveni identifikacijski kod koji jednoznačno određuje objekt. Vrijednost tog atributa mora početi s malim slovom ili donjom crtom i može sadržavati samo alfa numeričke znakove. Vrijednost id se ne može promijeniti na drugom mjestu.

Druga kategorija su takozvani *property* atributi. Oni mogu poprimiti neku statičnu vrijednost ili dinamičke vrijednosti koje se dobije izvršavanje naredbi. Deklariraju se na sljedeći način: *property tip_podataka ime : vrijednost.* Vrijednost se može izostaviti. Vrijednost mora odgovarati tipu podataka inače dolazi do pogreške. Postoje nekoliko posebnih vrsta *property* atributa među kojima najkorišteniji su *alias* i *readonly*. *Alias* atributi spremaju referencu na neki drugi objekt a *readonly* onemogućavaju mijenjanje vrijednosti tog atributa.

Treća kategorija atributa su atributi signala i signal *headleri*. Signal je obavijest da se nešto desilo npr. klik na gumb, slika se skinula itd. Drugi objekti se obavještavaju o nastaloj promijeni pomoću signala. Da bi se definirao novi signal koristi se sljedeća sintaksa *signal ime_signala[(par1,par2,…)]*. Nakon imena signala može se navesti lista parametra koju će signal poslati. Ovdje je bitno za naglasiti da pojedini objekt može imati samo jedan signal s istim imenom tj. signali se ne mogu *override-ati.* Da bi određeni drugi objekt mogao reagirati na promjenu mora koristiti četvrtu vrstu atributa, signal *handler*. Na ovu vrstu atributa može gledati kao posebnu vrstu metoda koju poziva QML stroj kada dolazi do emitiranja signala vezanog za tu promjenu. Imena signal *hendlera* počinju s prefiksom *on* nakon kojeg slijedi ime signala.

Peta kategorija su atributi metoda. To su zapravo JavaScript funkcije. Deklaracija: *function ime_funkncije(par1,…) {/*tijelo funkcije*/}.* Funkcija može ali ne mora imati listu parametra. Kasnije se te funkcije mogu pozivati unutar ili izvan definirajučega objekta. Unutar istog objekta ne smije se pojavljivati metoda i signal istog imena.

Zadnja kategorija su dodijeljeni atributi. To su atributi koji su definirani u samom objektu. Njima možemo samo pridružiti vrijednost. Svaki objekt ima odgovarajući set tih atributa.

2.1.3. Podržane platforme

Snaga QT i time QML je platformska nezavisnost. Trenutno, u trenutku pisanja, podržane platforme su:

a) Microsoft Windows – Windows 10 (32 i 64 bita), Windows 8.1 (32 i 64 bita), Windows 10 (32 i 64 bita) i Windows 7 (32 i 64 bita)
b) Linux/X11 – OpenSuse 13.1 (64 bita) i novije verzije, Ubuntu 14.04 (64 bita) i novije verzije, Red Hat 6.6 (64 bita) i novije verzije
c) OSX – 10.8, 10.9, 10.10, 10.11
d) Linux ugradbeni sustavi i QNX
e) Mobilne platforme – Windows Phone 8.1, iOS 6 i novije verzije, Android (API Level 16 i novije)

Svi navedeni operacijski sustavi biti će podržani minimalno do 15.6.2017.

2.1.4. Qmake

Qmake je alata razvijen od strane Qt-a koji služi u procesu prevađanja (eng. *compile*), vezivanja (eng. *linking*) i izgradnje (eng. *building*) računalnih programa na različitim platformama. Automatizira proces izgradnje takozvanih *makefile*-ova koji služe za izgradnju software-a. Ekstenzija qmake datoteka je .pro i takve datoteke se nazivaju *project file*. Unutar svake te datoteke nalaze se uputstva kako sagraditi program. Također omogućuje programeru automatsko kreiranje .moc i .uic datoteka. *Meta-Object Compiler* (MOC) je program koji je zadužen za rješavanje Qt C++ datoteka. Svaki put kad prevoditelj naiđe na *Q_OBJECT* makro naredbu unutar C++ koda, MOC kreira meta-objektni kod. Meta-objektni kod je potreban zbog mehanizma signala i signal *handlera* i pronalaženja tipa podatka u stvarnom vremenu. UIC iliti *User Interface Compiler* čita XML datoteku u kojoj je opisano grafičko sučelje i generira pripadajući C++ kod.

Alternativa qmaku je cmake ali uglavnom se prilikom rada s Qt preferira qmake.

2.2. OpenCV

OpenCV (*Open Source Computer Vision*) je knjižnica programskih funkcija namijenjena ponajprije području računalnog vida u stvarnom vremenu. Osim funkcijama za baratanje slike i videa implementirani su i različiti algoritmi strojnog učenja. Trenutno ima implementirano oko 2500 različitih algoritma. Knjižnica je razvijena od strane Intela a danas je otvorenog koda pod BSD (*Berkeley Software Distribution*) licencom koja omogućava besplatno korištenje. Osim šta je besplatan OpenCV je i *cross-platform*. Može se koristiti na Windows-u, Linux-u, OS X-u te na mobilnim platformama Android, iOS i BlackBerry 10.

OpenCV napisan je u C++ ali ima sučelje i prema drugim programskim jezicima kao što je C, Java, Python i MATHLAB.

Primjeri korištenja: prepoznavanje lica, identifikacija objekta, izrada 3D modela objekta, pronalaženje sličnih slika iz baza podataka, praćenje pogleda, klasificiranje ponašanja ljudi na videu itd.

OpenCv ima modularnu strukturu. Centralni modul naziva se *core*. Unutar njega definirani su svi bazični tipovi podataka i osnovne funkcije. Ostali moduli koriste taj modul pa *core* je neizostavni dio. Osim *core-a* bitni moduli su *imgproc* u kojoj su definirani naprednije funkcije za baratanje sa slikama kao što su linearno i nelinearno filtriranje, geometrijske transformacije, transformacije boja itd. Nakon njega slijedeći bitan modul je *objdetect* koji služi za detekciju objekta kao što su lica. Osim ta tri navedena modula u izradi ove aplikacije korišteni su moduli *photo* i *imgcodecs*. *Imgcodecs* služi za pisanje i čitanje slika za diskova ili memorije uređaja a *photo* za naprednije korištenje slika.

2.3. Dlib

Dlib je knjižnice opće namjere razvijena 2002 godine od strane Davisa Kinga. Knjižnica je napisana u C++ jeziku. Onda kao i OpenCv je *opensource*, koristi BSL(*Boost Software Licence*) licencu što je čini slobodnu za korištenje u bilo kojem pogledu. Trenutno knjižnica je platformski neovisna, može se koristiti i na Windows-ima, Linux-u i OS X-u.

Sadržaj knjižnice može se podijeliti prema vrsti algoritma. Tako imamo:

a) Algoritmi strojnog učenja – algoritmi za klasifikaciju, SVM (*Support vector machine*), k-means ,Newman clustering, …
b) Numerički algoritmi – brzi algoritmi za rad s matricama, dekompoziciju (npr eigenvektor)
c) Algoritmi za rad sa slikama – detekcija objekta, rubova, SURF,…
d) Kompresija podataka – CRC 32, MD 5, …
e) Testiranje – unit testovi, logeri, …
f) Grafičko sučelje – izrada jednostavni sučelja
g) Rad s mrežama – TCP socket API, HTTP server, …
h) Generalni alati – serijalizacija objekta, XML, kontejneri, …

3. Ciljane platforme

Aplikacija je zamišljena kao mobilna aplikacija za Ubuntu Touch operacijski sustav. Osim toga aplikacija se mora izvršavati i na drugim operacijskim sustavima kao što su Linux ili Windows. Jedina nova platforma je Ubuntu Touch pa je potrebno detaljnije opisati.

3.1. Ubuntu Touch

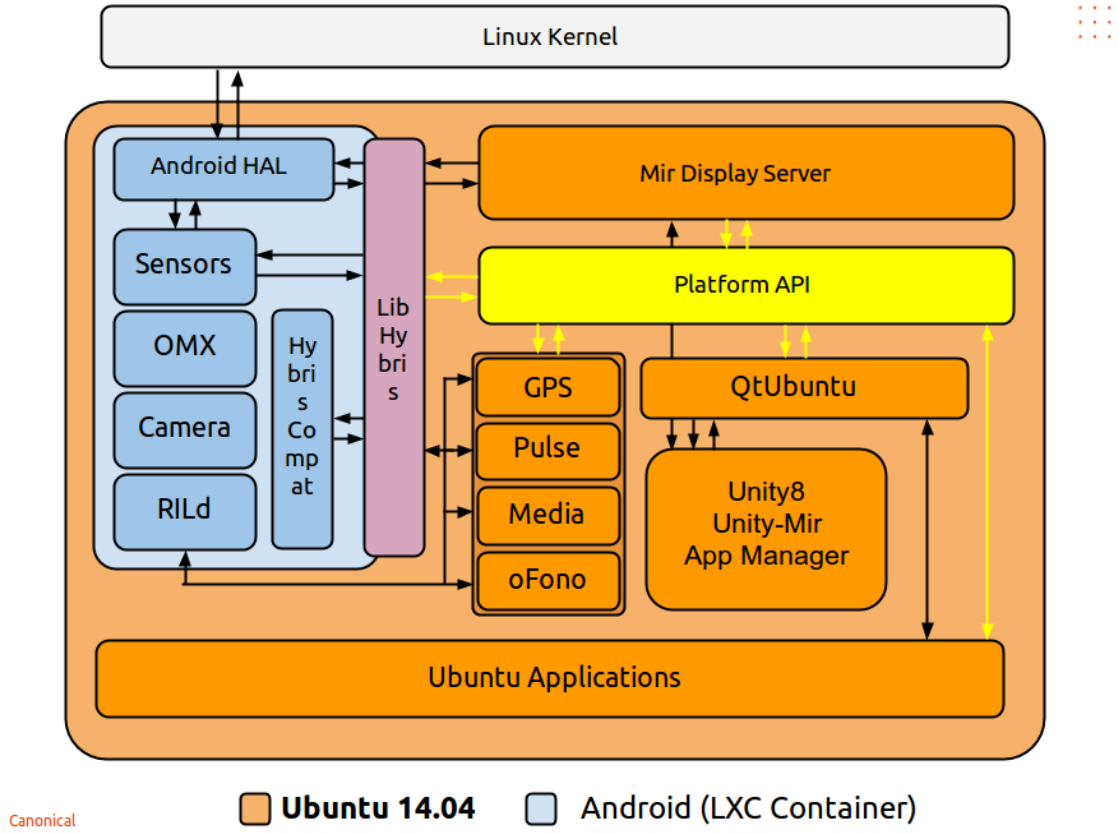
Ubuntu Touch ili Ubuntu Phone je mobilna verzija Ubuntu operacijskog sustava razvijena od strane Canonical UK Ltd. Dizajnirana je ponajprije za mobilne uređaje kao što pametni telefoni i tableti. Prva verzija izašla je 17.3.2013 a zadnja službena verzija je 15.04 koja je izašla 19.7.2016. Na slici 2. vidi se početni zaslon Ubuntu Phone.

Slika 2. Ubuntu Touch

Danas na tržištu postoje četiri mobilna uređaja koji direktno podržavaju taj operacijski sustav. To su : Meizu MX4, BQ Aquaris E5, BQ Aquaris 4.5, Meizu Pro 5. BQ Aquaris 4.5 bio je prvi uređaj koji je podržavao taj sustav. Osim tih uređaja Ubuntu Touch može se instalirati na praktički sve uređaja koji imaju instalirani Android OS.

Da bi se na uređaju mogao instalirati taj OS, uređaj mora zadovoljiti sljedeće kriterije: CPU – ARM Cortex A7 ili bolji, memorija 1 GB, i ekran na dodir mora omogućavati *multi-touch.*

Na slici 3. prikazana je arhitektura operacijskog sustava. Osnova tog operacijskog sustava je Linux karnela. Komunikaciju s hardwareom vrši Android HAL (*Hardware Abstraction Layer*) koji je preuzet iz Android OS. Također su preuzeti i određeni driveri i *libhybris (*sloj kompatibilnosti – koristi se prilikom komunikacije i korištenja drivera*)*. To je bilo moguće jer je Android *opensource*. Unity 8 napisan je nanovo za potrebe operacijskog sustava koristeći Qt 5.



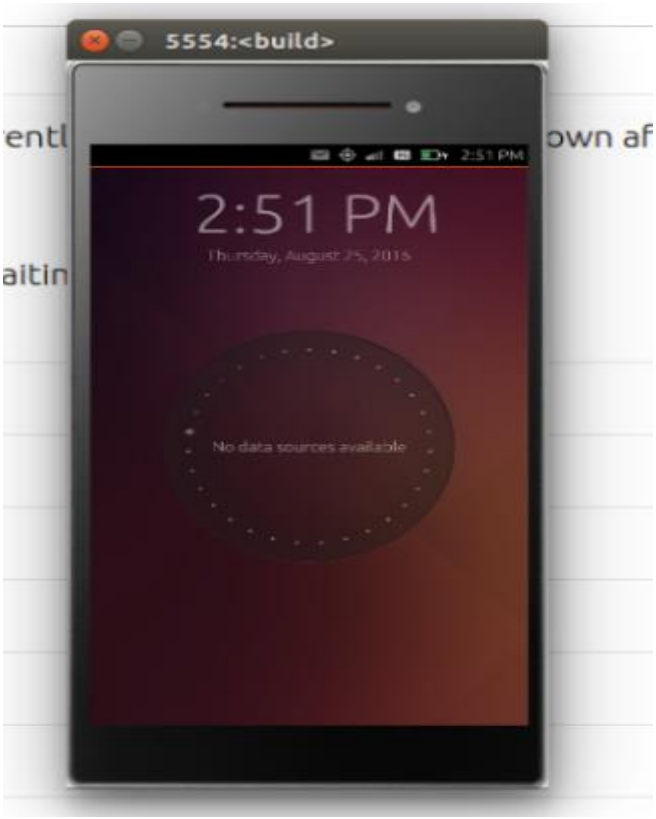
Slika 3. Arhitektura OS

Šta se tiče samog razvoja aplikacija, Ubuntu Touch daje programeru dvije solucije. Mogu se razvijati klasične aplikacije ili *scope.* Na *scope* možemo gledati kao neki posebnu vrstu web stranica koje se vrti na početnom zaslonu telefona. Na taj način korisnik dolazi brže do potrebnog sadržaja. Aplikacije se mogu razvijati u HTML5 tehnologiji i u Qt-u. Najčešće se korisničko sučelje razvija u QML. A šta se tiče same logike najčešće se koristi JavaScript i C++. Naravno mogu se koristi i drugi jezici, ponajprije Python i GO. Za razvoj aplikacija koristi se razvojna okolina Ubuntu SDK.

Aplikacije se pakiraju u *click* pakete. *Click* paketi su posebna vrsta paketa koji su nastali pojavom Ubuntu Touch. Najviše se koriste za instalaciju aplikaciju na mobilne uređaje ali se mogu koristiti i na ostalim Linux operacijskim sustavima. Ubuntu SDK automatski kreira *click* paket. Ukoliko se koristi neki drugi IDE *click* paket može se kreirati naredbom *click build.* Svaka aplikacija koja želi biti dostavljena u *click* formatu mora imati *manifest.json* datoteku. Unutar manifesta, koji je u JSON formatu, moraju se definirati sigurnosna pravila (koje resurse aplikacija može koristiti). Osim manifesta sama aplikacija mora zadovoljavati neke određene uvjete. Prvi uvjet je taj da ukupna veličina aplikacije ne smije prelaziti 3 GB. Također aplikacija mora biti besplatna za korištenje (barem do verzije 13.10). Ostali uvjeti mogu se pronaći na službenoj stranici(https://myapps.developer.ubuntu.com/dev/tos/).

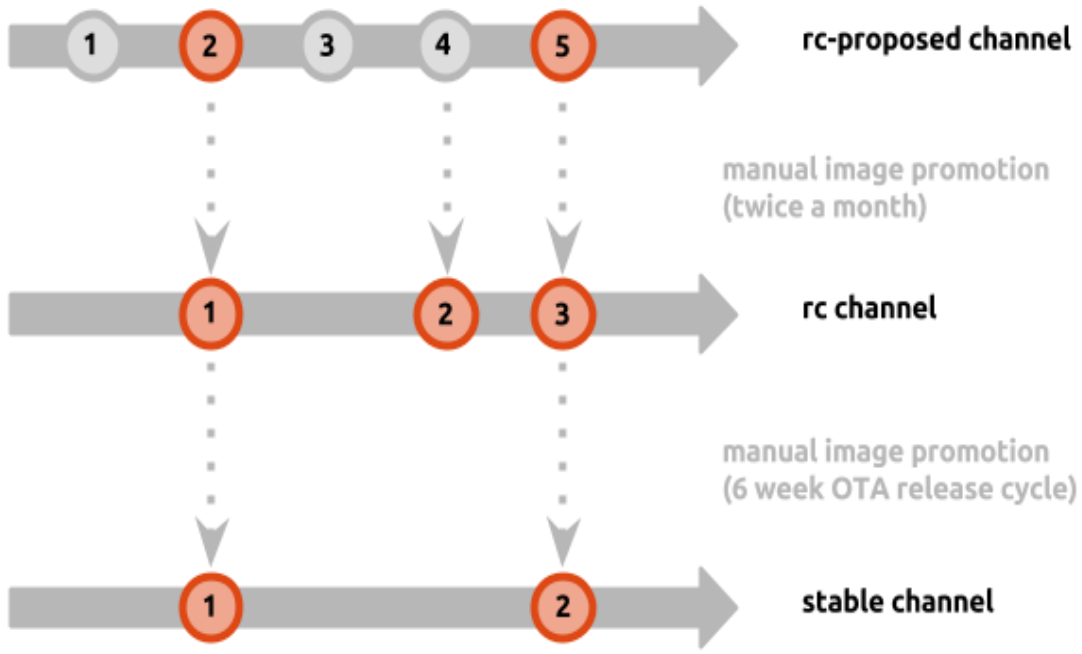
3.1.1. Ubuntu Touch emulator

Kao bi se mogle razvijati aplikacije za mobilne uređaje bez potrebe da programer kupi uređaj moguće je testirati aplikacije unutar emuliranog uređaja. No za razliku od Android emulatora ili ostalih virtualnih mašina ovaj emulator je veoma loš. Pola verzija operacijskog sustava koje su dostupne ne rade. Također emulator je nestabilan te se puno puta ruši. Ogroman je problem i taj da se emulator mora ponovo pokretati nekoliko puta da bi se opće sustav podigao. Na slici 4. može se vidjeti pokrenuti emulator na računalu.



Slika 4. Emulator

Prva stvar da bi se emulator pokrenuo računalo koje ga pokreće mora imati minimalno 512 MB radne memorije, 4GB slobodnog prostora i OpenGL grafički driver. Ako su uvjeti zadovoljeni može se krenuti s kreiranjem nove instance. Prva stvar koju je potrebno odabrati je arhitektura. Može se odabrati armhf i i386. i386 predstavlja x86 arhitektura procesora dok armhf predstavlja ARM arhitektura (RISC procesor) koju koriste svi mobilni uređaji i tableti. Dobro je napomenuti da je i386 arhitektura znatno brža od armhf i preporuča se njeno korištenje. Nakon arhitektura bira se „kanal“ (eng.*channel*). Slike OS objavljuju se na serveru na nekoliko različitih kanala. Kanal nije ništa drugo nego *timeline* objava. Ukoliko se ne specificira posebno koristi se *devel* kanal. Osim njega postoje drugi kanali. Svaki kanal služi određenoj svrsi i predstavljaju životni ciklus slike OS. Taj ciklus može se vidjeti na slici 5.



Slika 5. Životni ciklus slike

Svakodnevno se objavljuju slike na *proposed* kanalu. Nakon šta prođe testiranja i osigura određenu kvalitetu slika prelazi na *rc* ili *devel* kanal. Nakon dodatnih testiranja slika prelazi na *stable* kanal gdje se slika dalje ne testira niti se ne razvija. Opet valja napomenuti da se mora paziti prilikom preuzimanja slika OS jer postoji velika mogućnost da emulator neće raditi. Prilikom razvoja ove aplikacija testirane su sve slike sa svih kanala ali radila je jedino slika preuzeta s *devel* kanala. Sliku sa *stable* kanala iz nepoznatog razloga nije bilo moguće preuzeti.

Na kraju preostaje odabrati *kit. Kit* specificira okolinu za razvoj aplikacije, npr. qt verziju, C++ prevoditelj, *root,* ...Preporučuje se da IDE automatski kreira novi *kit* za emulator kako bi se smanjila mogućnost pogreške.

4. Princip zamjene lica

Proces zamjene lica s jedne slike na drugu slike kompliciraniji je nego šta se na prvi pogled čini. Sav taj proces zahtjeva odlično poznavanje matematike i algoritama strojnog učenja. Pisati cijeli kod od početka bez korištenja tuđih knjižnica Sizifov je posao. Zato su korištene knjižnice OpenCV i dlib. Proces ćemo podijeliti u nekoliko koraka.

Prvi korak je pronaći točke koje obilježavaju lica on obje slike. Te točke su tipične točke koje se nalaze na nosu, očima, ustima i na okviru lica (pogledati sliku 6.)

Slika 6. Točke lica

Za zamjenu lica nisu nam potrebne sve točke nego samo one na vanjskim rubovima. Iako OpenCV je već 2001 godine imao implementirani algoritam za detekciju točaka na licu za izradu ove aplikacije korištena je dlib-ov funkcija *frontal_face_dector*. Dlib-ova verzija je jednostavnija za korištenje, lakše se implementira i daje bolje rezultate. Da bi detektirao lice ova funkcija koristi HOG (*Histogram of oriented gradients*). Ova tehnika koristi se u različitim

segmentima računalnom vida i obrade slike a funkcionira tako broji pojavljivanja orijentacije gradijenta na lokaliziranim područjima slike. Iako kao klasifikator u HOG-u mogu se koristiti neuralne mreže dlib-ova verzija koristi SVM (*Support Vector Machine*). Osim same implementacije algoritma dlib dostavlja i testne podatke za učenje i trening samog algoritma.

Nakon šta su pronađene potrebne točke potrebno je pronaći konveksni omotač (eng. *convex hull*). Konveksni omotač po definiciji je najmanji konveksni poligon (onaj mnogokut kod kojeg skup točaka između bilo koje dvije točke mnogokuta također pripada mnogokutu) koji obuhvaća se točke. Za pronaći konveksni omotač koristi se funkcija iz OpenCv *convexHull*. Toj funkciji šalju se pronađe točke lica sa slika.

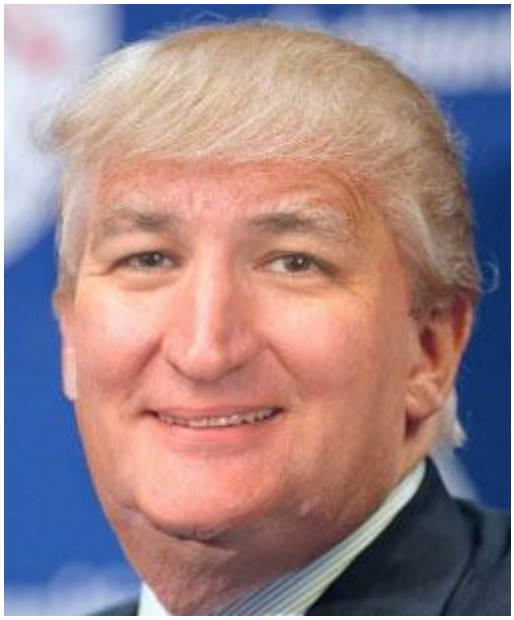
Sljedeći korak je Delaunayeva triangulacija nad konveksnim omotačem. Triangulacija u geometriju označava proces podijele ravnine u trokute. Nad jednom ravninom postoje različite kombinacije triangulacija. Delaunayeva triangulacija je stroži oblik triangulacije u kojoj se ravnina i dalje dijeli na trokute ali se ti trokuti biraju tako da niti jedan vrh trokuta se ne nalazi unutar opisane kružnice drugog trokuta. Takva triangulacija minimalizira zbroj svih kutova svih trokuta u ravnini. Na slici 7. vidi se Delaunayeva triangulacija nad točkama pronađenih na slici 6.

Slika 7. Delaunayeva triangulacija

Nakon šta pronađemo trokute na jednoj i na drugoj slici preostaje nam „stopiti srodne“ trokute. Na taj način lice jedne osobe zalijepimo na lice druge osobe. To se radi tako da odaberemo jedan trokut iz prve slike. Nakon toga odaberemo srodni trokut iz druge slike i izračunamo tako zvanu afini transformaciju (eng. *affine transformation*). Afini transformacija je funkcija definirana unutar afini prostora (poopćenje euklidskoga prostora u algebarskoj geometriji pri kojemu koordinate točaka nisu nužno realni ili kompleksni brojevi, već elementi zadanoga polja). Afini transformacija ne garantira održavanje istih kutova i duljina linija ali garantira održavanje omjera duljina i paralelnosti linija. Neke od promjena koje nastaju primjenom te transformacije su rotacija, skaliranje, refleksija, promjena oblika, … OpenCv ima implementiranu funkciju za izračun transformacije, *getAffineTransform*. Nakon šta smo izračunali transformaciju moramo je primijeniti na nad trokutima da bi dobili transformiranu sliku. OpenCV ima definiranu funkciju koja upravo radi taj posao a ona se zove *wrapAffine*. Pošto *wrapAffine* radi nad slikama a ne nad trokutima prije samog pozivanja te funkcije mora se napraviti mali trik. Trik je taj da se prvo izračuna okvir za trokut nakon toga se izradi transformacija nad svim pikselima i na kraju se maskiraju pikseli izvan trokuta. Za izračunavanje maska za trokute koristi se funkcija *fillConvexPoly*.

Sad imamo lice jedne osobe zalijepljeno, bolje rečeno smješteno nad licem druge osobe. Da bi uredili tu sliku da bude realističnija postoji funkcija u OpenCv koja se zove *seamlessClone* koja primjenom različitih filtera, transformacija i promjena boje spaja dvije slike. Rezultati funkcije mogu se vidjeti pogledom na slike 8 i 9. Slika 8 prikazuje kako bi izgledala slika bez primjene funkcije a slika 9 sa primjenom te funkcija. Razlika je očigledna.

Slika 8. Bez seamlessClone



Slika 9. Nakon primjene seamlessClone

Prilikom korištenja te funkcije može se odabrati vrsta klonirajuće metode. Postoje 3 metode a to su normalna, miješana i izmjena svojstva. Da se ne ulazi u previše detalja najbolja opcija za izmjenu lica je normalna metoda. Ta metoda se najviše koristi prilikom kloniranja objekta sa složenim vanjskim linijama.

5. Opis aplikacije

Aplikacija je izrađena u Ubuntu SDK razvojnoj okolini. Korišten je template *QML App with C++ plugin (qmake)*. Ovaj template je odabran iz razloga šta je potrebno napraviti poseban dodatak u C++ koji će obavljati zamjenu lica sa slika. Jednostavno nije moguće to napraviti koristeći samo QML i JavaScript. Odabirom ovog template-a IDE automatski generira sve potrebne .pro datoteke, manifest i potrebne .cpp datoteke za plugin. Naravno, mogli smo koristiti i Qt template ali onda se ne bi automatski kreirale neke datoteke, kao što je manifest.

Aplikaciju se može podijeliti na dva dijela. Prvi je dio C++ plugin za zamjenu lica a drugi je QML grafičko sučelje. Kompletna aplikacija nalazi se na github-u (link: https://github.com/josip31/PlatNezProgramiranje).

Naziv aplikacije je FaceSwap. Izrađena je u *framework*-u 15.04.

5.1. C++ plugin

Prije nego se krene s opisom ovog konkretnog plugina valja opisati kako se kreiraju i koriste u QML. QML *Engine* zadužen je za učitavanje C++ *plugin*-a unutar QML. Dovoljno je samo unutar QML importirati *plugin* preko imena i verzije. Da bi se kreirao novi *plugin* treba se kreirati pod klasa klase *QQmlExtensionPlugin*. Unutar te klase potrebno je koristeći makro naredbu *Q_PLUGIN_METADATA* registrirati *plugin* sa Qt meta sistemom. Također mora se *override*-ati metoda *registerTypes* i pozvati metoda *qmlRegisterTypes* kako bi se *plugin* mogao eksportirati. Osim kreiranja pod klase potrebno je kreirati i dvije nove datoteke. Prva je *project* datoteka a druga *qmldir* datoteka. Unutar *qmldir* datoteke definira se URI (*Uniform Resource Identifier*), tip plugina, …

5.1.1. Plugin project datoteka

Slika 10. prikazuje samo jedan dio *project* datoteke koju će kasnije qmake koristiti za kreiranje *makefile-a*.

```
TEMPLATE = lib
TARGET = FaceSwapbackend
QT += qml quick
CONFIG += qt plugin
CONFIG +=c++11

#load(ubuntu-click)

TARGET = $$qtLibraryTarget($$TARGET)

# Input
SOURCES += \
    backend.cpp \
    mytype.cpp

HEADERS += \
    backend.h \
    mytype.h

OTHER_FILES = qmldir \
              shape_predictor_68_face_landmarks.dat \

!equals(_PRO_FILE_PWD_, $$OUT_PWD) {
    copy_qmldir.target = $$OUT_PWD/qmldir
    copy_qmldir.depends = $$_PRO_FILE_PWD_/qmldir
    copy_qmldir.commands = $(COPY_FILE) \"$$replace(copy_qmldir.depends, /, $$QMAKE_DIR_SEP)\" \"$$replace(copy_qmldir.target, /, $$QMAKE_D
    QMAKE_EXTRA_TARGETS += copy_qmldir
    PRE_TARGETDEPS += $$copy_qmldir.target
}

qmldir.files = qmldir
installPath = $${UBUNTU_CLICK_PLUGIN_PATH}/FaceSwap
qmldir.path = $$installPath
target.path = $$installPath
```

Slika 10. Dio plugin project datoteke

Prva bitna stvar koju valja odmah primijetiti je prva varijabla *TEMPLATE*. Varijablom se specificira vrsta *templata* koja će se kreirati. Moguće vrijednosti su: *APP, LIB, SUBDIR, AUX, VCAPP,VCLIB. APP* označava da će se kreirati *Makefile-ovi* za kreiranje aplikacije, *LIB* za kreiranje biblioteke, *SUBDIR* da se nalaze drugi direktoriji a *AUX* sugerira da će se kreirati datoteke koje ništa ne *build*-aju. Kod kreiranja *plugin-a* koristi se *LIB template*. Na kraju kreirati će se nova dinamička knjižnica (.so datoteka).

Sljedeća važna varijabla je *TARGET* koja označava ime ciljne datoteke. Ime te varijable mora odgovarati imenu navedenom u *qmldir* datoteci, varijabla *plugin*.

Varijabla *QT* specificira module koje projekt koristi. Početne definirane vrijednosti su *gui* i *qt*. Kad se izrađuje *plugin* potrebno je dodati module *qml* i *plugin.*

Varijabla *CONFIG* specificira dodatne opcije i opcije za prevoditelja. Neke od najznačajnih opcija su:

a) *Release* – projekt će biti izgrađen u *release* modalitetu
b) *Debug* - projekt će biti izgrađen u *debug* modalitetu
c) *C++11* – prevoditelj će koristiti C++11 standard
d) *Thread* – omogućuje se korištenje niti
e) *Exception_off* -prekida se podrška javljanja izuzetaka
f) *Qt* -označava da se radi o Qt aplikaciji
g) *Windows* -označava da se radi o aplikaciji za Windowse
h) *Static* – radi se o statičnoj knjižnici
i) *Shared* – radi se o dinamičkoj knjižnici
j) *Plugin* – označava o *plugin-u*. Može se koristiti jedino ako je *TEMPLATE=lib*

Od ovih svih navedenih opcija očito je da su se morale koristiti opcije *qt* i *plugin.* Ono šta nije tako očito je zašto se koristila opcija *C++11*. Ta opcija se koristila kako bi se zadovoljila dlib knjižnica. Ukoliko bi se isključila onda bi prevoditelj javio pogrešku i aplikacija se ne bi mogla prevesti.

Varijable *SOURCES* i *HEADERS* označavaju izvorišne datoteke i datoteke zaglavlja. Bitno je naglasiti da qmake čitaju datoteke zaglavlja automatski detektira da li je potrebno uključiti *MOC. Plugin* se sastoji od dvije klase. Prva klasa, *BackendPlugin,* je definira zaglavljen *backend.h* a njena implementacija nalazi se u datoteci *backend.cpp*. Druga klasa, *MyType*, definira zaglavljen *mytype.h* a njena implementacija nalazi se u datoteci *mytype.cpp*.

OTHER_FILES označava ostale datoteke koje se ne mogu okarakterizirati. Ovdje se dodaje *qmldir* datoteka. Još jedna bitna datoteka koja se ovdje dodala je datoteka s podacima za učenje dlib-ovog algoritma za detekciju lica.

Još jedna jako bitna stvar je vezivanje aplikacije s potrebnim knjižnicama. Pošto ova aplikacija koristi OpenCV i dlib potrebno je pravilno povezati aplikaciju s knjižnicama. Na slici 11 vidi se jedan primjer povezivanja. Varijabla *LIBS* označava knjižnice koje su potrebne aplikaciji i koje *linker* mora povezati. Oznakom -L označava se put gdje se nalazi knjižnica a oznakom -l ime te knjižnice. Ova aplikacija zahtijeva 6 dodatnih knjižnica a to su: libdlib.so, libopencv_core.so, libopencv_imgcodecs.so, libopencv_imgproc.so, libopencv_objdetec.so i libopencv_photo.so.

```
win32:CONFIG(release, debug|release): LIBS += -L$$PWD/../../../../../Downloads/dlib/build/dlib/release/ -ldlib
else:win32:CONFIG(debug, debug|release): LIBS += -L$$PWD/../../../../../Downloads/dlib/build/dlib/debug/ -ldlib
else:unix: LIBS += -L$$PWD/../../../../../Downloads/dlib/build/dlib/ -ldlib
```

Slika 11. Povezivanje s knjižnicom

5.1.2. Qmldir

Qmldir datoteka za ovaj *plugin* je veoma jednostavna. Sastoji se od samo dvije linije koda (slika 12).

```
module FaceSwap
plugin FaceSwapbackend
```

Slika 12.Qmldir

Naredbom *module* definiramo ime (URI). Kasnije pomoću tog imena importiramo plugin u QML datoteke. Naredbom *plugin* deklarira se koji će *plugin* omogućiti u modulu. Dobra je praksa dodati i treću naredbu, *typeinfo*. Tom se naredbom označava ime datoteke koja opisuje naš *plugin*. Takve datoteke imaju nastavak *.qmltypes.* Za izradu tih datoteka koristi se alat *qmlplugindump.* Naravno, moguće je i ručno napisati tu datoteku ali preporuča se korištenje dok alata. Za kreiranje datoteke koristi se naredba: *qmlplugindump ime_plugina verzija put_do_plugina > put_do_plugina/ime_plugina.qmltypes*

5.1.3. Backend klasa

Ranije je napomenuto da prilikom izrade plugina potrebno je kreirati pod klasu klase *QQmlExtesionPlugin.* Upravo klasa *BackendPlugin* služi tome. Na slici 13 prikaza ne datoteka zaglavlja a na slici 14 samo implementacija.

Koriste se dvije makro naredbe. Prva je *Q_OBJECT* kako bi se kreirao *moc* kod a druga je *Q_PLUGIN_METADATA.* Ova druga makronaredba služi da bi se dalo do znanja da se

koriste meta podaci dio *plugin-a*. Naredba prima jedan argument a to je IID od sučelja kojeg implementira.

Šta se tiče metoda potrebno je definirati samo dvije. Prva metoda je *registerType* koju smo već prije opisali a druga je *initializeEngine.* Metoda *initializeEngine* virtualna metoda koja inicijalizira QML *engine.* Metoda *registerTypes* je ovdje najbitnija metoda. Ona registrira QML tip sa određenim URI-em. Metoda prima samo jedan argument i to je upravo taj URI. Unutar te metode mora se pozvati metoda *qmlRegisterType.* Ova metoda registrira C++ tip unutar QML sistema. Metoda prima 4 argumenta: URI, verziju, pod verziju i ime novog QML tipa. Metoda vraća ID registracije (cijeli broj). Vidi se u implementaciji da je metoda generička i mora joj se proslijediti tip podataka. Taj tip je tip objekta koji se registrira, u ovom slučaju to je *MyType.*

```
class BackendPlugin : public QQmlExtensionPlugin
{
    Q_OBJECT
    Q_PLUGIN_METADATA(IID "org.qt-project.Qt.QQmlExtensionInterface")

public:
    void registerTypes(const char *uri);
    void initializeEngine(QQmlEngine *engine, const char *uri);
};
#endif // BACKEND_PLUGIN_H
```

Slika 13. BackendPlugin header

```
#include <QtQml>
#include <QtQml/QQmlContext>
#include "backend.h"
#include "mytype.h"

void BackendPlugin::registerTypes(const char *uri)
{
    Q_ASSERT(uri == QLatin1String("FaceSwap"));

    qmlRegisterType<MyType>(uri, 1, 0, "FaceSwapLogic");
}

void BackendPlugin::initializeEngine(QQmlEngine *engine, const char *uri)
{
    QQmlExtensionPlugin::initializeEngine(engine, uri);
}
```

Slika 14. Implementacija BackendPlugin klase

5.1.4. MyType klasa

Unutar te klase implementirana je logika zamjene lica koja je opisana u poglavlju 4. Ova klasa nasljeđuje klasu *QObject.* Ukoliko naš *plugin* želi nešto nacrtati na ekranu onda umjesto

QObject klase nasljeđuje se od *QQuickItem* klase. U našem slučaju ništa se neće direktno crtati po ekranu pa je dovoljno naslijediti *QObjeckt.* Ne bi bilo pogrešno da smo koristili drugu klasu.

Kao u prethodnoj klasi mora se dodati *Q_OBJECT* makro. Novost u ovoj klasi je *Q_PROPERTY* naredba. Ovom naredbom kreiraju se atributi koji će biti dostupni iz QML. QML direktno ne može pristupati varijablama deklariranim u klasi. Također ne može koristiti ni njene metode. Da bi se definiralo novo svojstva prva stvar koja se prosljeđuje naredbi je tip i ime. Ukoliko se želi pročitati vrijednost tog svojstva mora se definirati metoda za čitanje. To je najobičnija *get* metoda. Unutar *Q_PROPERTY*-a se mora naznačiti koja metoda je zadužena za čitanje. To se radi tako da se nakon ključne riječi *READ* navodi ime metode. Slično je ako želimo izmjenjivati vrijednost svojstva. Tada se definira nova *set* metoda a ona se označava unutar *Q_PROPERTY*-a sa ključnom riječi *WRITE* i imenom metode. Također može se definirati i signale koji se emitiraju ukoliko dođe do promjene. To se radi tako da se definira metoda i onda se ona registrira unutar *Q_PROPERTY*-a sa ključnom riječi *NOTIFY*. Ovog puta te metode moraju biti smještene unutar *Q*_SIGNAL bloka (vidi sliku 16) .Na slici 15. vide se sva nova svojstva definira za potrebe ove klase. Prvo svojstvo koristilo je samo prilikom testiranja. Preko svojstva *image1source* i *image2source* klasa dobiva informacije (*path* za čitanje slika) o slikama koje se sudjelovati u zamjeni. Zadnje svojstvo je URL novo nastale slike.

```
Q_PROPERTY(QString message READ message WRITE setMessage NOTIFY messageSetSignal )
Q_PROPERTY(QString image1source READ image1source WRITE setImage1source NOTIFY image1sourceChanged)
Q_PROPERTY(QString image2source READ image2source WRITE setImage2source NOTIFY image1sourceChanged)
Q_PROPERTY(QUrl imageswapped READ imageswapped WRITE setImageswapped NOTIFY imageSwappedChanged)
```

Slika 15. Q_Property

```
Q_SIGNALS:
    void messageSetSignal();
    void image1sourceChanged();
    void image2sourceChanged();
    void imageSwappedChanged();
```

Slika 16. Signali

Na slici 17 vide se implementacija metoda za *READ* i metoda za *WRITE* za svojstva *image1source* i *image2source*. Ponovimo još jednom, to su najobičnije *get* i *set* metode koje čitaju ili mijenjaju varijable klase.

```
QString image1source(){return img1_src;}
void setImage1source (QString src){img1_src=src;}

QString image2source(){return img2_src;}
void setImage2source (QString src){img2_src=src;}
```

Slika 17.READ i WRITE metode

Konstruktor i destruktor klase prikazani su na slici 18. Vidi se da destruktor ne radi ništa posebno. Šta se tiče konstruktora on najprije kreira novi objekt tipa *get_frontal_face_dector* koji je definiran u dlib knjižnici. Nakon toga deserijaliziraju se testni podatci i kreira objekt *shape_predictor* koji je također definiran u istoj knjižnici. Na kraju inicijalizira se varijabla *IMAGE_SAVE_PATH.* Ta varijabla predstavlja mjesto gdje će se spremati slike.

```
MyType::MyType(QObject *parent): QObject(parent)
{
    this->detector = dlib::get_frontal_face_detector();
    dlib::deserialize("../FaceSwap/backend/FaceSwap/shape_predictor_68_face_landmarks.dat") >> sp;
    QStringList dirList=QStandardPaths::standardLocations(QStandardPaths::PicturesLocation);
    IMAGE_SAVE_PATH=dirList.at(0).toLocal8Bit().constData();
   }

MyType::~MyType() {

}
```

Slika 18. Konstruktor i destruktor

Najbitnija funkcija cijele klase je *doTheSwap* koja obavlja cijeli proces opisan u poglavlju 4. Bitno je za naglasiti da je funkciju poziva metoda *createSwap* (slika 19) koja spada u grupu *Q_SLOTS.* To je napravljeno iz razloga da bi se funkcija mogla pozivati iz QML.

```
public Q_SLOTS:
    void createSwap(){doTheSwap();}
```

Slika 19. createSwap()

Prva stvar koju metoda *doTheSwap* radi je čitanje varijabli *img1_src* i *img2_src* u kojoj je spremljena putanja do slika. Potrebna je pretvorba iz *QStringa* u normalni *string*. Nakon što su putanje poznate učitavaju se slike pomoću metode *imread* definirane u OpenCv. Učitane slike spremaju se u *Mat* tip podataka koji nije ništa drugo nego n-dimenzionalna matrica. Nakon učitavanja slika slijedi pronalaženje tipičnih točaka lica. Tu ulogu obavlja metoda *imageToPoints*. Njena implementacija prikaza je na slici 20.

```
std::vector<cv::Point2f> MyType::imageToPoints(std::string path){
      std::vector<Point2f> points;
      try
        {
           dlib::array2d<dlib::rgb_pixel> img;
           dlib::load_image(img, path);

           std::vector<dlib::rectangle> dets = detector(img);

           dlib::full_object_detection shape = sp(img, dets[0]);
         for (int i=0;i< 68;i++){

          int xx=shape.part(i).x();
          int yy=shape.part(i).y();
          cv::Point2f temp(xx,yy);
         points.push_back(temp);
        }

        }
        catch (std::exception& e)
        {
          //  std::cout << "\nexception thrown!" << std::endl;
           // std::cout << e.what() << std::endl;
        }


        return points;
 }
```

Slika 20. imageToPoints

Funkciji se ne prosljeđuje učitana slika nego put do slike pa se slika ponovno učitava. Ovdje se radi dupli posao ali nije bilo drugačijeg rješenja. Problem je u tome šta OpenCV vraća učitanu sliku u *Mat* formatu a za rad s dlib-om potrebna nam je slika u drugačijem formatu. Taj format je vektor *dlib::rgb_pixel*. Sad kad imamo sliku u pravom formatu može se krenuti sa detekcijom točaka. Prvi korak je pronaći sva lica na slici. Kako bi se pronašla lica koristi se već spomenuti dlib-ov *object detector*. Taj objekt detektira sva lica na slici ali nama je potrebno samo jedno pa zato kod traženja točka lica šaljemo samo prvi pronađeni objekt, tj. prvo pronađeno lice. Za pronalaženje točka lica koristi se objekt *sp* koji je tipa *dlib::shape_predictor*. Taj objekt pronalazi točno 68 karakterističnih točka. Kako bi mogli te točke koristiti s OpenCv radimo pretvorbu u tip *vector<Point2f>*.

Sljedeći korak je pronalaženje konveksne ravnine šta je trivijalno jer OpenCv ima gotovu funkciju koja radi taj posao. Funkciji *convexHull* moramo proslijediti samo točke lica, mjesto gdje će spremiti rezultate, orijentaciju (u našem slučaju je *false*, obrnuto od smjera kazaljke na satu) i zastavicu da li funkcija vraća točke ravnine ili indekse ravnine (stavljajući *false* funkcija vraća indekse). Ti indeksi označavaju koji član matrice slike ulazi u ravninu. Zato moramo jednom *for* petljom proći kroz sve pronađene indekse i spremiti odgovarajuće točke unutar nizova *hull1 i hull2*.

Sljedeći korak je izrada Delaunayeve triangulacije. Implementacija je prikazana na slici 21.

```
void MyType::calculateDelaunayTriangles(Rect rect, std::vector<Point2f> &points, std::vector< std::vector<int> > &delaunayTriangle){

    Subdiv2D subdiv(rect);


    for( std::vector<Point2f>::iterator it = points.begin(); it != points.end(); it++)
        subdiv.insert(*it);

    std::vector<Vec6f> triangleList;
    subdiv.getTriangleList(triangleList);
    std::vector<Point2f> pt(3);
    std::vector<int> ind(3);

    for( size_t i = 0; i < triangleList.size(); i++ )
    {
        Vec6f t = triangleList[i];
        pt[0] = Point2f(t[0], t[1]);
        pt[1] = Point2f(t[2], t[3]);
        pt[2] = Point2f(t[4], t[5 ]);

        if ( rect.contains(pt[0]) && rect.contains(pt[1]) && rect.contains(pt[2])){
            for(int j = 0; j < 3; j++)
                for(size_t k = 0; k < points.size(); k++)
                    if(abs(pt[j].x - points[k].x) < 1.0 && abs(pt[j].y - points[k].y) < 1)
                        ind[j] = k;

            delaunayTriangle.push_back(ind);
        }
    }

}
```

Slika 21. calculateDelaunayTriangles

Nema se ništa posebno za kazati o ovoj funkciji. Ona jednostavno slijedi algoritam za pronalaženje triangulacije. Zatim slijedi sama triangulacija koja je implementirana funkcijom *triangulate*. Unutar te funkcije poziva se i druga funkcija koja je zadužena za afine transformaciju. Ta funkcija se naziva *warpAffineTrans*. Obje implementacije mogu se vidjeti na slici 22 i na slici 23. Obje procedure već su opisane pa ne treba trošiti puno riječi. Samo valja napomenuti da su sve korištene funkcije definirane unutar OpenCv. Valja samo napomenuti novo korištene funkcije. Funkcija *boundingRect* vraća minimalni pravokutnih koji obuhvaća točke i *fillConvexPoly* koja računa konveksni mnogokut. Koristi se i nova OpenCV klasa SubDiv2 koja predstavlja podjelu ravnine na nekoliko dijelova tako da se ti dijelovi međusobno ne isprepliču

```
void MyType::triangulate(Mat &img1, Mat &img2, std::vector<Point2f> &pts1, std::vector<Point2f> &pts2)
{

    Rect rect1 = boundingRect(pts1);
    Rect rect2 = boundingRect(pts2);

   std::vector<Point2f> t1Rect, t2Rect;
   std::vector<Point> t2RectInt;
    for(int i = 0; i < 3; i++)
    {

        t1Rect.push_back( Point2f( pts1[i].x - rect1.x, pts1[i].y -  rect1.y) );
        t2Rect.push_back( Point2f( pts2[i].x - rect2.x, pts2[i].y - rect2.y) );
        t2RectInt.push_back( Point(pts2[i].x - rect2.x, pts2[i].y - rect2.y) );

    }

    Mat mask = Mat::zeros(rect2.height, rect2.width, CV_32FC3);
    fillConvexPoly(mask, t2RectInt, Scalar(1.0, 1.0, 1.0), 16, 0);

    Mat img1Rect;
    img1(rect1).copyTo(img1Rect);

    Mat img2Rect = Mat::zeros(rect2.height, rect2.width, img1Rect.type());

    warpAffineTrans(img2Rect, img1Rect, t1Rect, t2Rect);

    multiply(img2Rect,mask, img2Rect);
    multiply(img2(rect2), Scalar(1.0,1.0,1.0) - mask, img2(rect2));
    img2(rect2) = img2(rect2) + img2Rect;


}
```

Slika 22. triangulate

```
void MyType::warpAffineTrans(Mat &warpImage, Mat &src, std::vector<Point2f> &srcVec, std::vector<Point2f> &dstVec)
{
    Mat warpMat = getAffineTransform( srcVec, dstVec );
    warpAffine( src, warpImage, warpMat, warpImage.size(), INTER_LINEAR, BORDER_REFLECT_101);
}
```

Slika 23. warpAffineTran

Na kraju jedino šta prestaje je zalijepiti lice s prve slike na drugu i pokrenuti *seamlessClone* kako bi nova slika izgleda bolje. Novu sliku spremamo na predefinirano mjesto i na kraju emitiramo signal da je došlo do kraja. Kod se može vidjeti na slici 23.

```
Mat mask = Mat::zeros(img2.rows, img2.cols, img2.depth());
fillConvexPoly(mask,&hull8U[0], hull8U.size(), Scalar(255,255,255));

Rect r = boundingRect(hull2);
Point center = (r.tl() + r.br()) / 2;
Mat output;
img1Warped.convertTo(img1Warped, CV_8UC3);
seamlessClone(img1Warped,img2, mask, center, output, NORMAL_CLONE);
setImageswapped(QUrl(QString::fromStdString(IMAGE_SAVE_PATH+"/face_swap.jpg")));
imwrite(IMAGE_SAVE_PATH+"/face_swap.jpg",output);
emit imageSwappedChanged();
```

Slika 24. Kod za lijepljenje

5.2. QML grafičko sučelje

Cijelo grafičko sučelje realizirano je samo koristeći QML i JavaScript. Sučelje je dizajnirano da bude što jednostavnije korištenje aplikacije na mobilnim platformama.

5.2.1. Project datoteka za sučelje

Ova datoteka je veoma *project* datoteci plugina ali ima male razlike. Prva razlika je ta šta je *TEMPLATE=app.* Može se koristi i *aux* ali ako se *deploy*-a aplikacija onda mora biti template *app.* Sljedeća stvar je razlika u *QT* varijabli. Ovdje se koristi module *core, multimedia* i *quick.* Imamo i dalje *SOURCE* ali sad ima samo jednu vrijednost i to je datoteka gdje je smještena *main* funkcija. Susrećemo neke nove varijable. Prva je *QML_FILES* i onda predstavlja sve datoteke koje izrađuju sučelje. To su sve QML *i* JavaScript datoteke. Možemo ih sve nabrojati ali možemo koristiti i pametniji način. Jednostavno koristimo naredbu *$$files(.qml,true).* Ta naredba pronaći će sve datoteke u direktoriju i vezati je za tu varijablu. Slično možemo napraviti za bilo koji drugi format samo umjesto *.qml* stavljamo odgovarajuću ekstenziju. Nakon toga imamo varijablu *DISTFILES.* Ovo je jedna on nejasnih varijabli ali pošto ju je automatski kreira IDE ostavljena je. U dokumentaciji piše da varijabla označava datoteke koje će se uključiti u „*dist target*“. U nju spremamo samo *.qml* datoteke. *CONF_FILE* označava konfiguracijske datoteke. Tu je smještena slika ikone i *.apparmor* datoteka. Nova jako bitna varijabla je *RESOURCE.* Ona definira ime *.qrc* datoteke. Posljednja nova varijabla je *QML2_IMPORT_PATH* gdje se definiraju novi direktoriji koje će prevoditelj pretraživati prilikom importa *QML* datoteka.

5.2.2. Apparmor datoteka

Apparmor datoteke su Linux sigurnosti moduli koji omogućavaju sistem administratoru i programeru da definiraju prava pristupa aplikacije resursima računala. *Apparmor* za našu aplikaciju napisan je u JSON obliku i može se vidjeti na slici 25.

```
{
    "policy_groups": [
        "networking",
        "camera",
        "picture_files"
    ],
    "policy_version": 1.3
}
```

Slika 25. Apparmor

Potrebna prava pristupa našoj aplikaciji su pravo pristupa kameri, mreži i čitanju slika sa memorije uređaja.

5.2.3. Desktop datoteka

Datoteka s ekstenzijom .desktop služi za pokretanje aplikacije na računalu. U njoj je definirano ime aplikacije, vrsta i kako se pokreće aplikaciju. U našem primjeru aplikaciju se pokreće unutar *qmlscene*. Potrebno je još prilikom pokretanja aplikacije definirati početno sučelje. Cijela datoteka prikazana je na slici 26.

```
[Desktop Entry]
_Name=FaceSwap
Exec=qmlscene %U FaceSwap/Main.qml
Icon=FaceSwap/FaceSwap.png
Terminal=false
Type=Application
X-Ubuntu-Touch=true
```

Slika 26. Desktop datoteka

5.2.4. Početni zaslon

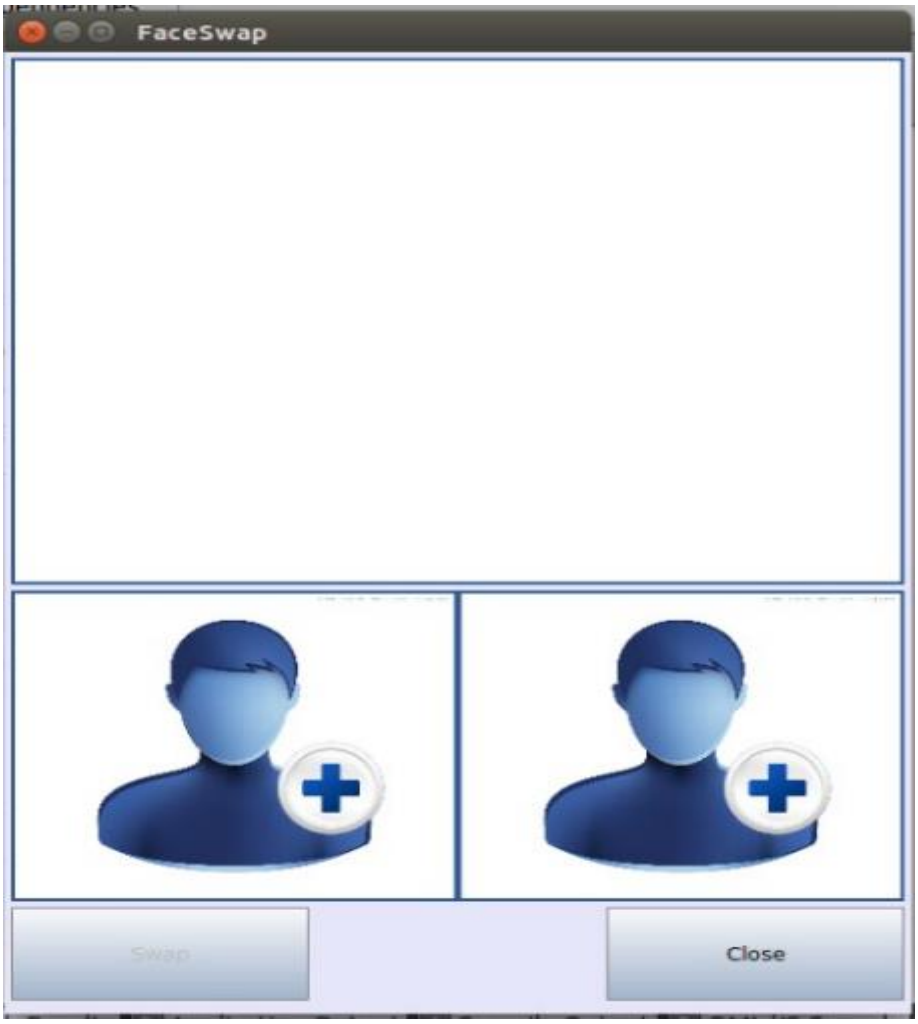
Najkompliciraniji dio grafičkog sučelja je početni zaslon. Prvi dio QML dokumenta je dio importa. Importa za početni zaslon je prikaz na slici 27. Početni zaslon spremljen je u datoteci *Main.qml.*

```
import QtQuick 2.5
import QtQuick.Controls 1.4
import QtQuick.Controls.Styles 1.4
//import Ubuntu.Components 1.3
import QtQuick.Window 2.2
import FaceSwap 1.0
```

Slika 27. Import početnog zaslona

Korišteni su samo *QtQuick* komponente. Importom *QtQuick-a* ne uključuju se direktno sve njegove komponente pa se mora pojedinačne module posebno importirati. Osim *QtQuick-a* ovdje se importirao prije kreirani modul (*import FaceSwap 1.0*).

Glavni objekt je tipa *Window*. Njega karakterizira mogućnost da unutar njega se prikazuju drugi objekti istog tipa ili objekti kao što su *Item* i ostali. *Window* definira automatski gumbe za minimalizirati, maksimizirati i zatvoriti aplikaciju. Na slici 28. prikazan je početni ekran.



Slika 28. Početni zaslon

Unutar objekta *Window* definiran je *StackView*. Ovaj element omogućuje promjenu trenutnog prikazanog elementa. Na njega možemo gledati kao neku vrstu stoga. Element na vrhu stoga se trenutno prikazuje. Kad se dodaje novi prozor on dođe na vrh stoga te prikazuje. Kad mičemo element prikazati će se onaj koji bio prije na vrhu stoga. Prvi element se postavlja tako da se atributu *initialItem* dodaje *ID* vrijednost željenog elementa. U ovom slučaju prvi element je naš element *Item* sa ID *mainWindow*. Novi elementi se dodaju metodom *push* a micanje elementa metodom *pop*.

Nakon definiranja *StackView-a* definiran je element tipa *Item*. Ovaj objekt je osnovni objekt za sve objekt definirane unutar *Qt quick-a* koji imaju vizualni prikaz. Taj objekt biti će naš početni zaslon. Unutar tog objekta definirana su novi atributi (slika 29.). Kreirana su dva atributa tipa *alias* radi jednostavnijeg referenciranja objekta. Svaki taj *alias* referencira element na kojem se prikazuje jedna od slika potrebnih za zamjenu lica. Treći novi atribut je varijabla koja predstavlja trenutno označenu sliku. Koristi se prilikom korištenja menija. Zadnja dva atributa su dvije *boolean* varijable koje predstavljaju da li je slika učitana ili ne.

```
property alias image1: addFirstPicture
property alias image2: addSecondPicture
property var selectedImage
property bool image1Ready:false
property bool image2Ready:false
```

Slika 29. Novi atributi

Početni zaslon dijeli se na tri reda. Prvi red je najveći u njenu se prikazuje novonastala slika. U drugom redu nalaze se slike koje sudjeluju u zamijeni. Zadnji red nalaze se samo dva gumba. Jedan za izlaz iz aplikacije drugi za pokrenuti zamjenu. Gumb za zamjenu je onemogućen sve dok se ne učitaju dvije slike.

Prvi red je napravljen pomoću elementa *Item*. Slika 30. prikazuje dio koda drugog reda.

```
Item{
    id:firstRow
    width: parent.width
    height: parent.height / 31 * 17
    anchors.top:parent.top
    anchors.left: parent.left
    anchors.right: parent.right
    anchors.margins: {left:4/*units.gu(0.5)*/; right:4/*units.gu(0.5)*/ ; top:4/*units.gu(0.5)*/}
```

Slika 30. Prvi red

Pozicioniranje elementa vrši se pomoću atributa *anchors*. Na *anchors* možemo gledati kako neke kukice koje pričvršćuju na odgovarajuću poziciju u relaciji s drugim elementom.

Postoje nekoliko vrsta tih kukica a svaka označava neku poziciju. Primjeri tih kukica su: *top, left, right, center, fill*. Da bi se pozicionirao element moramo specificirati i element na koji ćemo zakačiti kukicu. Najbolje je element specificirati korištenjem *ID*. Na slici 30 vidi još jedan način specifikacije elementa. Da bi odabrali element u kojem je ugnježden drugi element može koristi svojstvo *parent*. Visina i širina elementa definiraju se atributima *height* i *width*. Ti atributi primaju kao vrijednosti cijeli broj ali mogu se koristi i ekspresije ukoliko rezultat je broj. Posebna vrsta *anchors* su margine. Kod margina mogu se koristiti samo *top, left, right* i *bottom* kukice. Na slici 30 kod definicije margina u komentarima se vidi korištenje Ubuntu mjerne jedinice *gu*. Vrijednosti *gu* mijenjaju se ovisno o platformi na kojoj se aplikacija vrti. Ukoliko se aplikacija vrti na retina zaslonu *1 gu =16 px,* ako se aplikacija vrti na pametnom telefonu *1 gu = 18 px,* ako se aplikacija vrti na normalnom ekranu onda je *1 gu = 8 px*. Na početku se koristio taj princip ali kasnije zbog problema s *deploy*-om aplikacije se odustalo. Potrebno je dodati novi Ubuntu modul koji nije dio Qt standarda.

Nakon šta se definirao „okvir“ može se kreirati element za prikaz slike koji će prikazivati sliku zamjene lica. Za prikaz slike koristi se tip *Image (*slika 31*).*

```
Image{
    id:swapedImage
    fillMode: Image.Stretch
    asynchronous:true
    cache:false
    smooth: true
    anchors.margins: {left:4/*units.gu(0.5)*/; right:4/*units.gu(0.5)*/ ; top:4/*units.gu(0.5)*/;}
    anchors.fill: parent

}
```

Slika 31. Image za novo kreiranu sliku

Kako bi element popunio cijeli red koristi se *anchors.fill*. Atributom *source* prosljeđuje se URL slike koje se prikazuje. Pošto još ta slika još nije kreirana nema potrebe ovdje definirati taj atribut. *fillMode* specificira šta će se dogoditi ako je slika veća od samog elementa. Tu se koristi modalitet *Stretch*, slika se skalira da stane u element. Postavljanjem atributa *asynchronous* na *true* omogućava se učitavanje slike u drugoj dretvi šta omogućuje da GUI se ne blokira dok traje učitavanje. *Cache* je postavljen na *false* jer nema potrebe spremati sliku u tu vrstu memorije. Zadnji postavljeni atribut je *smooth* koji ako je postavljen na *true* omogućuje bolje filtriranje slike šta se rezultira boljom kvalitetom slika. Nuspojava je ta šta aplikacija postaje malo sporija.

U ovom redu inicijaliziran je i element iz novokreiranog *plugin-a*. Slika 32 prikazuje taj element.

```
FaceSwapLogic {
    id:swapImageLogic
    message: "test"
    onImageswappedChanged: {
        swapedImage.source=imageswapped
    }
}
```

Slika 32.FaceSwapLogic

Ovdje je jedino važno primijetiti kreiranje *slot-a, onImageswappedChanged,* za hvatanje signala. Kad proces zamjene lica završi emitira se signal kojim se obavještava o nastaloj promijeni. Ovdje se taj signal hvata. Kad se uhvati, dodijeli se vrijednost svojstva *imageswapped* atributi *source* prethodno kreiranog elementa *Image*. Kad se *source* atribut promijeni automatski se počinje učitavati slika.

Drugi red je također napravljen elementom *Item* samo ovog puta je red podijeljen na dva stupca. Unutar svakog stupca nalazi se po jedan *Image* element. Ti elementi prikazuju slike koje će ulaziti u zamjenu lica. Na početku, prije bilo koje akcije korisnika, postavljena je jedna pomoćna slika. Ta slika služi kao neka pomoć korisniku. Slika sugerira da se mora kliknuti na to područje. Jedina novost naspram prošlo kreiranog *Image* elementa je registriranje *slot-a* kad dođe do promijene *source* atributa. Kod je prikaz na slici 33. Kad dođe do promijene *source* atributa kontrolira se da li mu je vrijednost različita od početne vrijednosti. Ako je znači da je učitana neka nova slika i postavlja se vrijednost *image1Ready* na *true.* Taj *slot* je registriran na obje slike. To nam je važno jer ukoliko je *image1Ready* i *image2Ready* postavljeni na *true* onda se gumb za zamjenu lica odblokira.

```
onSourceChanged: {
    if(source != "add-user-icon.jpg")
        mainwindow.image1Ready=true;
    else
        mainwindow.image1Ready=false;
}
```

Slika 33. onSourceChanged

Preko oba *Image* elementa je postavljena *MouseArea (*slika 34*). MouseArea* je nevidljivi element koji omogućuje hvatanje signala koji su kreirani mišom elementima koji nemaju direktnu podršku.

```
MouseArea{
    id:firstPictureArea
    anchors.fill:addFirstPicture

    onClicked: {
        mainwindow.selectedImage=addFirstPicture
        addPictureDialog.popup()}
}
```

Slika 34.MouseArea

Kad se klikne na sliku otvara se meni koji je definiran u datoteci *OpenPictureDialog.qml*. Osim otvaranja menia postavlja se varijabla *selectedImage* na trenutno odabrani *Image* element.

Treći red, kao i prethodna dva, realiziran je elementom *Item*. Unutar reda definirana su dva gumba. Jedan je za pokretanje zamjene lica a drugi je za izlaz iz aplikacije. Oba gumba realizirana su tipom *Button.* Na slici 35. prikaz je kod za gumb zamjene lica.

```
Button{
     id:swapButton
     //color: "#2F5997"
     anchors.left:parent.left
     text: "Swap"
     height: parent.height
     width: parent.width/3
     enabled: mainwindow.image1Ready && mainwindow.image2Ready
     onClicked:{
         //progressBar.visible=true
         swapedImage.source=""
         swapImageLogic.image2source=addSecondPicture.source
         swapImageLogic.image1source=addFirstPicture.source
         swapImageLogic.createSwap()

     }
     style : ButtonStyle {
         background: Rectangle{
             border.width: control.activeFocus ? 2 : 1
             border.color: "#888"
             gradient: Gradient {
                 GradientStop { position: 0 ; color: control.pressed ? "#a2b5cd" : "#fcfcfc" }
                 GradientStop { position: 1 ; color: control.pressed ? "#fcfcfc" : "#a2b5cd" }
             }

         }
      }
 }
```

Slika 35. Swap button

Button tip kreira najobičniji gumb. Posebni atributi koji su prvi put pojavljuju su *text, enabled* i *style*. Tekst koji se prikazuje na gumbu definira se atributom *text*. *Enabled* direktno svojstva *Button-a* nego je naslijeđeno od *Item* tipa. *Enabled* označava da li element može prihvaćati evente kreirane od stane tipkovnice ili miša. Na početku je to onemogućeno. Kad se učitaju obje slike onda gumb se odblokira. Registriran je *slot onClicked* koji hvata *cliked* signal. Kad dođe do emitiranja tog signala prva stvar koja se radi je postavljanje *source* atributa *swappedImage* na praznu vrijednost. To je potrebno kad se radi više od jedne zamjene lica. Sve novo kreirane slike imaju isto ime. Da bi se označilo QML stroju da je došlo do promijene i da je potrebno ponovno učitati sliku prvo se postavlja *source* na prazan *string* pa kasnije kad se kreira nova slika ponovno postavlja na pravu vrijednost. Za zamjenu lica potrebno je postaviti atribute *image1source* i *image2source.* Ti atributi su definirani u *plugin-u* i tipa *QUrl*. Oni omogućavaju da se iz C++ koda očitaju slike. Na kraju se poziva metoda za zamjenu lica.

Korištenjem *style* atributa izmjenjuje se vanjski izgled gumba. Atribut prihvaća element tipa *ButtonStyle*. Ovdje valja napomenuti da je *style* definiran unutar *Item* elementa te da svakom elementu može definirati novi stil. Zato osim *ButtonStyle* postoje primjerice *SliderStyle, CheckBoxStyle, MenuStyle, ...*

5.2.5. Meni

Prilikom klika na jednog od dva *Image* elementa definiranih u drugom redu početnog zaslona otvara se meni koji korisniku nudi tri akcije. Prva se učitavanje slike iz galerije, druga je slikanje nove slike a treća je brisanje slike. Element za izradu menija je *Menu* koji je definiran unutar *QtQuick.Controls* paketa te se taj paket mora importirati. Da bi se dodao novi element u meniju može se koristiti metoda *addItem* ili se jednostavno unutar *Menu* elementa kreiraju *MenuItem* elementi. Korištena je ova druga tehnika. Da bi meni bio dostupan unutar početnog zaslona mora se kreirati element tog tipa. Kreiranje je prikazano na slici 36.

```
OpenPictureDialog{
    id:addPictureDialog
}
```

Slika 36. OpenPictureDialog

Prvi *MenuItem (*slika 37*)*ne služi ničemu nego služi kao naslov meniju. *Menu* ima definirani *title* atribut koji služi toj svrsi ali iz nepoznatog razloga to nije funkcioniralo. Vjerojatno *title* prikazuje samo naslov kad je meni definiran u *MenuBar-u.* Da bi se imao osjećaj

da se radi o naslovu dodala se ikona pomoću atributa *iconSource* te su se sve akcije nad njime blokirale tako da su se atributi *enabled* i *checkable* postavili na *false.*

```
MenuItem{
    text:"Import image"
    checkable: false
    enabled: false
    iconName: "FaceSwap"
    iconSource: "FaceSwap.png"

}
```

Slika 37. Naslov menija

Prvi pravi element menija kreiran je nakon „naslova“ i služi za slikanje nove slike. Njegova implementacija vidljiva je na slici 38.

```
MenuItem{
    text:"Camera"
    onTriggered: {
       stack.push(takePicture)
    }
}
```

Slika 38. Camera

MenuItem ima samo jedan signal i to je *triggered* koji se emitira prilikom selekcioniranja tog elementa. Hvatanje se vrši *slot-om onTriggered*. Kad dođe do hvatanja tog signala na vrh stoga (*StackView* definiran u početnom zaslonu) stavlja se element tipa *TakePicture* koji je definiran u posebnoj QML datoteci što će rezultirati promjenom zaslona. Naravno prethodno je bilo potrebno kreirati taj element.

Druga akcija je učitavanje slike iz uređaja. Prikaz koda je na slici 39. Kad korisnik odabere ovu akciju otvara se novi prozor za odabir slike. Taj je prozor definiran u posebnoj QML datoteci. Kao bi se učitala ova datoteka prilikom odabira koristi se posebna tehnika. U hvatanju signala napravljena je jednostavna JavaScript skripta koja pokreće QML funkciju *createQMLObjekt.* Funkcija prima kao argument *string* i iz tog *stringa* kreira novi *QML* objekt.

```
MenuItem{
    text:"Gallery"
    onTriggered:{
       var loader=Qt.createQmlObject('import QtQuick 2.4; Loader{source:"ChoosePictureFromGalleryView.qml"}',mainwindow,"ChoosePictureFromGa
    }

}
```

Slika 39. Galerija

Kreirani objekt je tipa *Loader*. Ta vrsta objekta služi sa dinamičko kreiranje *QML* komponenti. Osim kreiranja mogu se dinamičko učitavati i QML datoteke. Tako je u ovom slučaju učitana datoteka *ChoosePictureFromGalleryView.qml* u kojoj je definiran *pop-up* prozor. Nakon šta se učila datoteka prikazuje se taj prozor.

Posljednja akcija služi za brisanje slike (slika 40). Slika se zapravo ne briše nego se postavlja ponovno početna slika.

```
MenuItem{
    text:"Remove"
    onTriggered: {
        if (mainwindow.selectedImage==mainwindow.image1){
            mainwindow.image1.source="add-user-icon.jpg"
        }
        else{
            mainwindow.image2.source="add-user-icon.jpg"
        }
    }
}
```

Slika 40. Brisanje slika

5.2.6. Slikanje slike

Izgled ove komponente (slika 41) se može podijeliti na dva dijela. Prvi dio je gornji dio koji prikazuje video koja kamera snima a drugi dio je kontrolni u kojem su smještana dva gumba. Prvi služi za slikanje a drugi za povratak. Da bi se uhvatila slika koristi se metoda *imageCapture* koja je definirana u *CameraCapture* elementu. Taj je element pod klasa klase *Camera*. Povratak natrag se vrši metodom *pop* nad *StackView-om.*

Slika 41. TakePicture

Da bi se spremila slika potrebno je koristiti 3 različita elementa. Prvi element je *Camera (*slika 42*).* Kao šta i sam naziv govori element koristi se kako bi se ušlo u interakciju sa fizičkom kamerom ukoliko ona postoji na uređaju.

```
Camera {
    id: camera
    imageProcessing.whiteBalanceMode: CameraImageProcessing.WhiteBalanceFlash
    exposure {
        exposureCompensation: -1.0
        exposureMode: Camera.ExposurePortrait
    }
    flash.mode: Camera.FlashRedEyeReduction

    imageCapture {

        onImageCaptured: {
               stack.push(imagePreviewView)
               imagePreviewView.imagePreviewImage.source=preview
               if (!imagePreviewView.isGood) {
                    imageCapture.cancelCapture()
               }

        }
    }
}
```

Slika 42. Camera

Unutar same definicije korištena su tri dodatna elementa. Jedan je tipa *CameraExposure* a drugi *CameraCapture* a treći *CameraFlash*. *CameraExposure* služi za mijenjanje svojstva kamere dok *CameraCapture* omoguće samo hvatanje slike. *CameraFlash* služi za postavljanje opcija bljeskalice. Kad se pozove metoda *capture* emitiraju se dva signala *imagCapture* i *imageSaved*. Prvi se emitira kad je slika uhvaćena ali još nije spremljena u memoriju uređaja a drugi kad se spremi slika. Ovdje se hvata prvi signal. Kad dođe do hvatanja signala otvara se novi pogled definiran u datoteci *ImagePreview.qml*. Potrebno je poslati tom pogledu *URL* uhvaćene slike. *URL* je *preview.*

VideoOutput omoguće prikazivanje trenutne slike koju kamera snima na zaslonu. Kod je prikazan na slici 43. Jako bitna stvar ovdje je korištenje atributa *source.* Ovdje, kao izvor, je korišten prethodno definira kamera ali mogu se koristiti i objekti *MediaPlayer*. Isto kao i kod korištenja slike *fillMode* je postavljen na *stretch.*

```
VideoOutput {
    source: camera
    anchors.fill:parent
    fillMode: VideoOutput.Stretch
    focus : visible

}
```

Slika 43. VideoOutput

5.2.7. Pregled slike

Pregled slike definiran je u *ImagePreview.qml* datoteci. Sastoji se od *Image* elementa gdje se prikazuje slika te od dva gumba. Jedan za povratak a jedan za prihvaćanje slike. Kod *Image* elementa nema se ništa novo za reći.

Ovdje valja samo napomenuti kako je definirano spremanje slike. Kad se klikne na gumb emitira se *clicked* signal a kod za hvatanje prikazan je na slici 44.

```
onClicked: {
    isGood:true
    if (mainwindow.selectedImage==mainwindow.image1){
        mainwindow.image1.source=camera.imageCapture.capturedImagePath

    }
    else{
        mainwindow.image2.source=camera.imageCapture.capturedImagePath
    }
    stack.pop()
    stack.pop()

}
```

Slika 44. Spremanje slike

Kad se prihvati slika onda se temeljen varijable *selectedImage* određuje da li se radi o prvoj ili o drugoj slici te se postavlja *source* na odgovarajuću sliku. Nakon toga vraća se na početni zaslon tako da se pozove metoda *pop* dva puta.

5.2.8. Glavna metoda

QML dokumente učitava i izvršava *QML runtime* zato je potrebno kreirati taj objekt na početku izvršavanja. Uglavnom se to radi tako da se napiše *main* metodu i u njoj pokrene *QQmlEngine*. To se može napraviti na dva načina. Prvi način je direktno tako da se ručno kreira taj objekt a drugi način je preko *QQuickView*. *QQuickView* pod klasa klase *QQuickWindow* koja automatski pokreće *QML scene* i učitava proslijeđeni dokument. Upravo taj drugi način je

korišten. Ovdje valja napomenuti da se *QQuickView* mora poslati početni zaslon. Kod je prikaz na slici 45.

```
#include <QGuiApplication>
#include <QQuickView>

int main(int argc, char *argv[])
{
    QGuiApplication app(argc, argv);

    QQuickView view;
    view.setSource(QUrl::fromLocalFile("Main.qml"));
    view.show();

    return app.exec();
}
```

Slika 45. Main

5.2.9. *Resource* datoteka

Qt omogućuje spremanja resursa kao šta su slike, zvukovi i same QML datoteke u binarnom obliku unutar samog koda aplikacije. Taj sustav je platformski nezavisan i naziva se *Qt resource system*. Takav način osigurava da se resurse neće nestati. Sve potrebne resurse specificiraju se unutar .qrc datoteke. Ta datoteka ima XML format.

6. Prikaz rada aplikacije

Na kraju preostaje još samo prikazati kako aplikacija radi. Nažalost, aplikaciju se nije uspjelo testirati na mobilnoj platformi jer nije bilo moguće instalirati aplikaciju na emulator. Do pogreške je dolazilo zbog OpenCV. Svaki put prilikom *deploy-a* dolazilo je do pogreške prilikom vezivanja aplikacije sa OpenCv knjižnicama. Format knjižnica nije bio podržan od strane emulatora. Pokušalo se OpenCv ponovno prevesti ali nije ništa pomoglo. Prevelo se ga je i u ARM format ali pogreška je ostala ista. Tako je aplikacija testirana samo na računalu.

Na slikama 46. i 47. prikazana su slike koje ulaze u transformaciju. Osobi sa slike 46 zalijepiti ćemo lice osobe sa slike 47. Rezultat je vidljiv na slici 48.

Na slici 49. vidi se kako to sve izgleda unutar aplikacije

Slika 46.Bill Gates

Slika 47. Mark Zuckerberg

Slika 48. FaceSwap

Slika 49. Izgled u aplikaciji

7. Zaključak

QML je moćan jezik za brzi razvoj grafičkog sučelja i u kombinaciji sa C++ pruža odličan alat za razvoj kompletnih aplikacija. QML je veoma jednostavan za učenje jer ima jednostavnu sintaksu i veoma je intuitivan. Još jedna bitna stvar kod njega je ta šta omogućuje razvoj platformski nezavisnih aplikacija.

Kod razvoja aplikacije najveći problem je manjak literature. Knjiga koje bi opisale razvoj aplikacija uopće nema. Jedini izvor informacija je službena stranica Qt.

Šta se tiče Ubuntu Touch OS nisam imao prilike ga bolje upoznati tako da ne mogu komentirati. Jedini doticaj je bio preko emulatora koji su jako jako loši. Ne može se Android emulatorom. Upravo je loš emulator bio razlog nemogućnosti realiziranja projekta do kraja.

8. Popis slika

9. Literatura

Internet izvori:

http://blog.dlib.net/2014/02/dlib-186-released-make-your-own-object.html

http://dlib.net/intro.html

http://dlib.net/

http://doc.qt.io/qt-5/qtqml-index.html

http://doc.qt.io/qt-5/qmlapplications.html

http://doc.qt.io/qt-5/qtquick-index.html

http://doc.qt.io/qt-5/topics-app-development.html

http://doc.qt.io/qt-5/moc.html

http://doc.qt.io/qt-5/uic.html

http://doc.qt.io/qt-5/qtqml-modules-qmldir.html

http://doc.qt.io/qt-5/qmake-variable-reference.html

http://doc.qt.io/qt-5/qtqml-modules-cppplugins.html

http://doc.qt.io/qt-5/qtplugin.html#Q_PLUGIN_METADATA

http://doc.qt.io/qt-5/properties.html

http://doc.qt.io/qt-5/qml-qtquick-window-window.html

http://doc.qt.io/qt-5/qml-qtquick-controls-stackview.html

http://doc.qt.io/qt-5/qml-qtquick-item.html

http://doc.qt.io/qt-5/qml-qtquick-mousearea.html#details

http://doc.qt.io/qt-5/qml-qtquick-controls-button.html

http://doc.qt.io/qt-5/qml-qtquick-controls-styles-buttonstyle.html

http://doc.qt.io/qt-5/qml-qtquick-controls-menu.html

http://doc.qt.io/qt-5/qml-qtmultimedia-camera.html

http://doc.qt.io/qt-5/qml-qtmultimedia-cameracapture.html#capture-method

http://doc.qt.io/qt-5/qml-qtmultimedia-cameraexposure.html#exposureCompensation-prop

http://doc.qt.io/qt-5/qml-qtmultimedia-videooutput.html

http://doc.qt.io/qt-5/qquickview.html#details

http://doc.qt.io/qt-5/qtquick-deployment.html

http://docs.opencv.org/2.4/modules/core/doc/basic_structures.html#mat

http://docs.opencv.org/3.0-beta/modules/photo/doc/cloning.html

https://developer.ubuntu.com/en/phone/apps/sdk/tutorials/using-the-ubuntu-emulator/

http://developer.ubuntu.com/en/phone/apps/qml/

https://developer.ubuntu.com/en/phone/apps/qml/tutorials/building-your-first-qml-app/
https://en.wikipedia.org/wiki/AppArmor
https://en.wikipedia.org/wiki/Delaunay_triangulation
https://en.wikipedia.org/wiki/Dlib
https://en.wikipedia.org/wiki/Histogram_of_oriented_gradients
https://en.wikipedia.org/wiki/OpenCV
https://en.wikipedia.org/wiki/QML
https://en.wikipedia.org/wiki/Ubuntu_Touch
http://events.linuxfoundation.org/sites/events/files/slides/Ubuntu%20Touch%20Internals_1.pdf
http://www.learnopencv.com/delaunay-triangulation-and-voronoi-diagram-using-opencv-c-python/
http://www.learnopencv.com/face-morph-using-opencv-cpp-python/
http://opencv.org/about.html
http://struna.ihjj.hr/naziv/afini-prostor/32967/
http://www.ubuntu.com/phone/devices
https://wiki.ubuntu.com/Touch
https://wiki.ubuntu.com/Touch/Emulator