SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 4313

Virtualne maske pokretane praćenjem lica

Klaudija Palle

Zagreb, lipanj 2016.

SADRŽAJ

1. Uvod

Računala su postala neizostavni dio ljudske svakodnevice. Koristimo ih u sve moguće svrhe, od obavljanja važnih poslova za osnovno funkcioniranje društva do uporaba za čistu zabavu. Kako sve više i više koristimo računala, to se više razvija i način na koji imamo interakciju s njima.

Tradicionalni način na koji bi davali upute računalu bio je putem dodatnih uređaja (poput miša i tipkovnice) koji bi kodirali naše poruke na način razumljiv računalnu. Danas se sve više teži tome da računalima komuniciramo na način koji je sličniji komunikaciji između ljudi, poput korištenja govora, gesta i dodira. Rad većine današnjih računala svodi se na izvršavanje niza unaprijed poznatih instrukcija, što se drastično razlikuje od načina funkcioniranja ljudskog mozga. Zbog toga je problem pretvaranja ljudskog govora, gesta i izraza lica u jezik razumljiv računalu izrazito složen te postoje područja posvećena isključivo proučavanju tog problema.

Računalni vid područje je koje se bavi analizom i obrađivanjem slika i izvlačenjem korisnih informacija iz istih. Računalni vid sastoji se od više specijaliziranih područja ovisno o tipu informacije koji želimo izvući, npr. prepoznavanje specifičnih objekata, prepoznavanja pokreta i događaja u videu (nizu slika), rekonstrukcija slika i scena itd. Glavna ideja računalnog vida je da računalom simuliramo ljudski vid i način percipiranja. [10]

Jedno specijalizirano područje računalnog vida prepoznavanje je ljudskog lica i njegovih karakteristika u slikama i videima. Iako postoje mnoge korisne primjene tog područja (raspoznavanje i verifikacija identiteta, detekcija umora vozača, raspoznavanje emocija [6]), u ovom radu bavimo se zabavnom uporabom. Cilj je rada stvoriti jednostavnu aplikaciju koja će u stvarnom vremenu iscrtavati virtualnu masku povrh lica u videu pomoću prepoznavanja karakterističnih točaka lica.

U 2. poglavlju navode se i opisuju alati i tehnologije korištene u implementaciji aplikacije. U 3. poglavlju ulazi se dublje u bibloteku Visage|SDK i njene funkcionalnosti koje su korištene u radu. Također se obrađuje problem stvaranja maski. Pogavlje 4. opisuje ideju i proces implementacije. U 5. poglavlju istaknuti su neki nedostatci tre-

nutne implementacije i ideje za potencijalna buduća poboljšanja. U zaključku ukratko se opisuju rezultati i ocjenjuje uspješnost implementacije.

2. Alati i tehnologije

Rad je programiran u programskom jeziku C++, a osim standardnih biblioteka jezika C++ korištene su još 4 dodatne biblioteke za implementaciju. Riječ je o bibliotekama Visage|SDK, OpenCV, GLFW te GLAD, koju koristimo za učitavanje OpenGL funkcija. U idućim odjeljcima ukratko je opisana svaka od biblioteka i njene funkcionalnosti važne za rad aplikacije.

2.1. Visage|SDK

Visage|SDK biblioteka je za računalni vid razvijena od tvrtke Visage Technologies. Biblioteka je specijalizirana za detekciju ljudskog lica u videima i slikama. Također se koristi za animaciju likova, procjenu dobi, spola i emocija osoba s ulaznih slika. [2] Više o korištenim funkcionalnostima biblioteke Visage|SDK bit će rečeno u idućem poglavlju.

2.2. OpenCV

OpenCV *(Open Source Computer Vision)* također je biblioteka računalnog vida s jako širokim područjem primjena. Biblioteku je moguće koristiti na više različitih platformi (engl. *cross-platform*), te je slobodna za uporabu pod BSD licencom. OpenCV koristi hardversko ubrzanje za postizanje optimalnih performansi. Samo neke od mnogobrojnih primjena su praćenje kretanja, segmentacija, prepoznavanje gesti, robotika, prepoznavanje dubine u slikama, itd. [1]

U ovom radu koristi se jako uzak skup svih njenih funkcionalnosti, pa se ne će previše ulaziti u opis biblioteke. Uporaba se svodi isključivo na pristup slikama kamere (engl. *camera frame*) računala te za učitavanje slika sa tvrdog diska u radnu memoriju računala. Dakle, koristi se za povezivanje ostalih komponenti aplikacije.

2.3. GLFW i OpenGL

OpenGL *(Open Graphics Library)* programsko je sučelje (engl. *application programming interface, API*) za iscrtavanje 2D i 3D grafike. OpenGL predstavlja standard funkcija za interakciju s grafičkim procesorom (engl. *graphics processing unit, GPU*) kojim se omogućuje hardversko ubrzanje, no moguće je da je dio funkcija implementiran programski. OpenGL dostupan je u više programskih jezika (engl. *cross-language*) i za više platformi. [8]

Iako OpenGL omogućuje direktan pristup funkcijama grafičkog procesora, kad ga koristimo za izradu aplikacija redovito uz njega koristimo neku drugu biblioteku koja se brine o stvaranju konteksta za OpenGL, održavanju prozora te obrađivanjem događaja ulazno-izlaznih napravi poput pritisaka tipki na tipkovnici, pomaka miša, i sl. Upravo u ove svrhe korištena je biblioteka GLFW. [5]

3. Maska i ključne točke lica

Kako bi bilo moguće početi postavljati masku na lice, prvo je potrebno odrediti koje će se karakteristične točke koristiti za definiranje maske te zatim te točke detektirati u video snimci. U prvom odjeljku bavit ćemo se pronalaženjem karakterističnih točki lica u slici i videu, a u drugom odjeljku promatrat ćemo podatke koji su nam potrebni da bi imali potpuno definiranu masku.

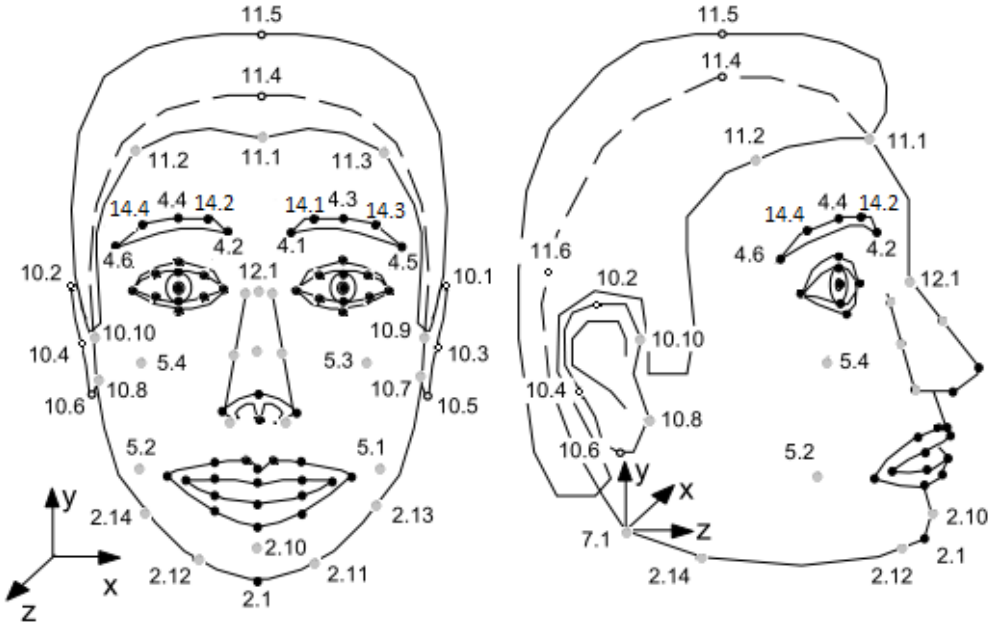
3.1. Ključne točke lica

Kao što je već rečeno, za detektiranje karakterističnih točaka lica koriste se funkcionalnosti biblioteke Visage|SDK. Cilj nam je dobiti točne 2D koordinate odgovarajućih točaka unutar slike ili videa.

Razredi `VisageTracker` i `VisageFeaturesDetector` omogućuju da im se proslijedi niz slika ili samo jedna slika za koju vraćaju strukturu podataka koja sadrži podatke o svim licima koja su otkrivena. `VisageTracker` koristi se za detektiranje lica u sekvenci slika (tj. videu), a `VisageFeaturesDetector` koristi se za detekciju u jednoj slici.

Struktura u biblioteci, koju oba razreda vraćaju kao rezultat, zove se `FaceData`. Struktura sadrži podatke o translaciji glave u odnosu na kameru, orijentaciji i nagibu glave, smjeru pogleda i mnoge druge. Nama najvažniji podaci su koordinate karakterističnih točaka lica sadržane u razredu `FDP`. U strukturi `FaceData` dane su 2D koordinate u odnosu na sliku, globalne 3D koordinate u odnosu na kameru i relativne 3D koordinate u odnosu na odabranu točku unutar lica. 2D koordinate moguće je i izračunati iz danih globalnih 3D koordinata te translacije i rotacije glave. [3]

Unutar razreda `FDP` jedna karakteristična točka definirana je indeksom grupe i indeksom unutar grupe. Visage|SDK definira brojeve grupa od 2 do 15, a broj točaka unutar grupe razlikuje se ovisno o grupi. Na slici 3.1 istaknute su najvažnije točke lica, a na slici 3.2 istaknute su karakteristične točke očiju, ustiju i nosa.



Slika 3.1: Karakteristične točke lica *(preuzeto iz [3])*

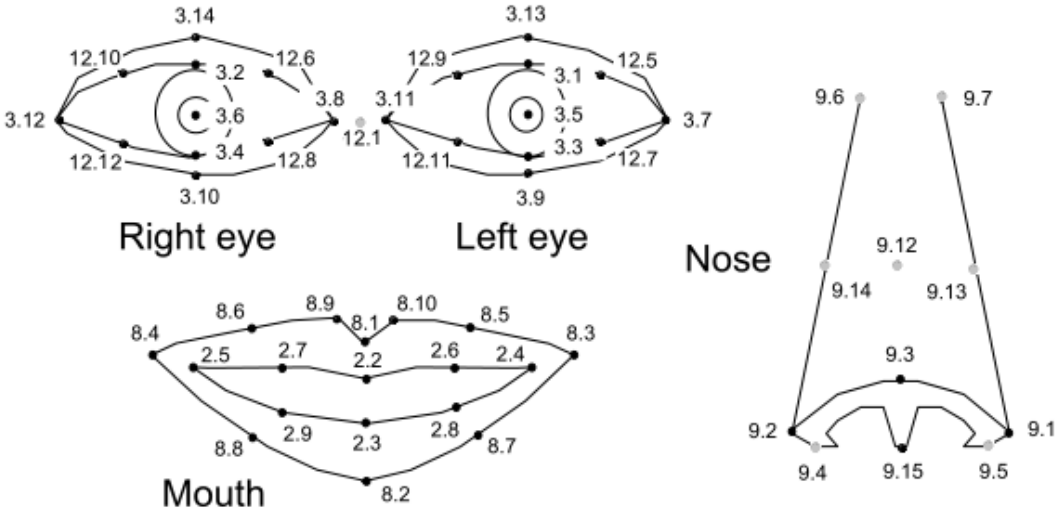
3.2. Generiranje maske

Dalje u radu koristit ćemo samo podskup dostupnih točaka za iscrtavanje maske, te ćemo iz tih probranih točaka sastaviti mrežu povezanih trokuta (engl. *triangle mesh*) koja će činiti virtualnu masku. Probrane točke (s odgovarajućim indeksom grupe i indeksom unutar grupe) i generirana mreža prikazani su na slici 3.3.

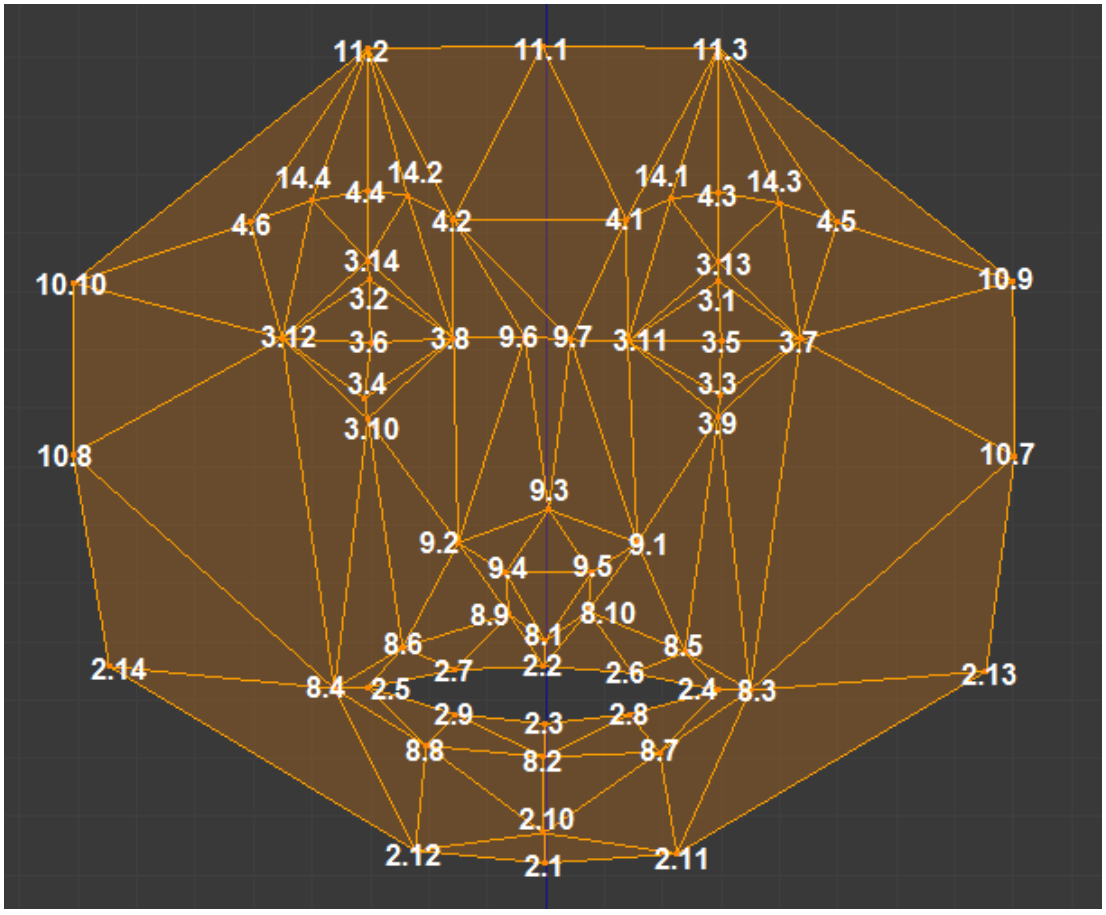
Da maska bude potpuno definirana, potrebno je imati teksturu maske i koordinate probranih karakterističnih točaka unutar teksture.

Ako za masku želimo koristiti lice druge osobe, unutar biblioteke VisagelSDK nalazi se razred `VisageDetector` koji nam ovo olakšava. Moguće je učitati sliku iz datoteke i danu sliku proslijediti detektoru koji će nam onda vratiti strukturu podataka sa svim karakterističnim točkama. Učitanu sliku naravno potrebno je poslati na grafički procesor u obliku teksture.

Kada želim kao masku koristiti sliku nečega poput životinje ili animiranog lika nastaje problem. Naime, za takva lica još ne postoji softver koji bi nam detektirao odgovarajuće karakteristične točke. Iz tog razloga, za maske životinja potrebno je ručno odrediti tražene karakteristične točke. Kad je broj odabranih točaka malen, to je još izvedivo, ali ako u budućnosti budemo željeli preciznije maske, ne će biti moguće stvoriti velik broj maski u kratkom roku. Također, životinjska lica nemaju uvijek prikladan oblik lica i glave na kojem bi se sa sigurnošću mogle odrediti neke karakteristične točke.



Slika 3.2: Karakteristične točke očiju, ustiju i nosa *(preuzeto iz [3])*



Slika 3.3: Mreža trokuta maske

4. Implementacija

Osnovna ideja rada je detektirati unaprijed odabrane točke lica (navedene u prethodnom poglavlju), iz njih sastaviti mrežu trokuta i zatim preko slike videa iscrtavati trokute teksturirane maskom. U prvom odjeljku opisana je osnovna implementacija iscrtavanja pomoću OpenGL-a, a u drugom i trećem odjeljku opisane su dorade osnovnog programa provjerom dubine i ublažavanjem rubova maske.

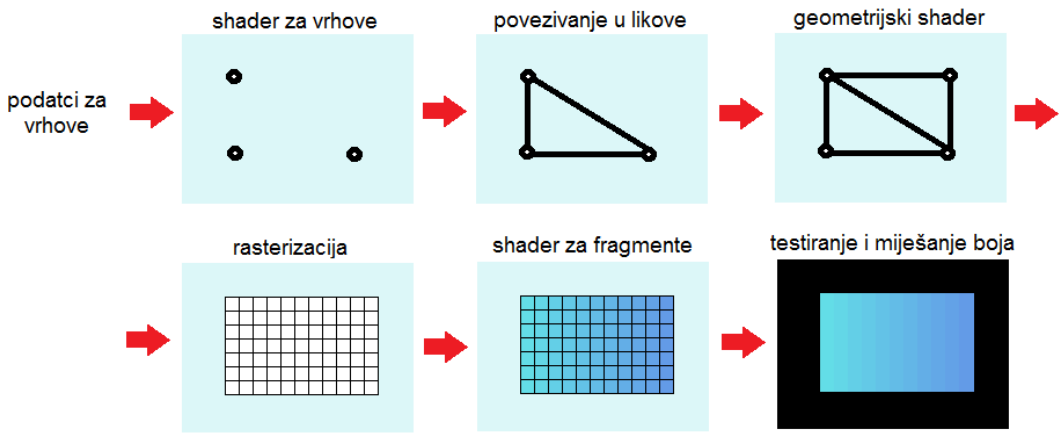
4.1. Crtanje teksturiranih trokuta

Kako radimo s OpenGL verzijom 3.3 naviše, moramo se pobliže upoznati s protočnom strukturom grafičkog sustava. Shader je jednostavan program koji se, u pravilu, izvodi na grafičkom procesori, i različiti shaderi koriste se u različitim dijelovima protočne strukture. Jezik koji koristimo za shadere u aplikaciji je GLSL *(OpenGL Shading Language)*.

U okvirnim crtama, grafički sustav prvo prima podatke o vrhovima, koje zatim obrađuje shader za vrhove (engl. *vertex shader*), zatim se iz vrhova stvaraju primitivni oblici (najčešće trokuti). Ove oblike moguće je, opcionalno, dodatno modificirati pomoću geometrijskog shadera (engl. *geometry shader*). Dobiveni likovi zatim se rasteriziraju. Dio rasteriziranih fragmenata odmah se odbacuje ako je izvan vidnog polja, a preostali rasterizirani fragmenti (slikovni elementi (engl. *pixel*)) se obrađuju u shaderu za fragmente (engl. *fragment shader*). U konačnom koraku radi se provjera dubine i miješanje boja ovisno o prozirnosti. Pojednostavljena struktura grafičkog protočnog sustava prikazana je na slici 4.1. [4]

Prvi korak koji moramo obaviti jest slanje podataka o vrhovima na GPU. U našem slučaju podatci o svakom vrhu sastojat će se od koordinata karakterističnih točaka u slici videa, te koordinata odgovarajućih točaka na teksturi maske.

Koordinate točaka u videu dobijemo pomoću `VisageTrackera` koji vraća `FaceData` strukturu za svaku sličicu videa. Koristit ćemo dane 2D koordinate, tj. na GPU šaljemo samo x i y koordinatu. Za koordinate tekstura koristit ćemo također 2D koordinate, no



Slika 4.1: Protočna struktura grafičkog sustava

one dobivene iz `FaceData` strukture za masku. S obzirom na to da se maska ne će mijenjati, njene koordinate možemo poslati samo jednom na početku izvođenja, dok ćemo koordinate točaka u videu morati slati za svaku sličicu.

Iz vrhova ćemo graditi trokute tako da je potrebno poslati podatke za tri vrha za svaki trokut. S obzirom da trokuti međusobno dijele vrhove, obično se ne šalju podaci koji se ponavljaju. Umjesto toga šaljemo podatke za sve različite vrhove, a zatim pošaljemo polje indeksa gdje svaka tri člana pokazuju na vrhove iz prvog polja koje treba upotrijebiti za stvaranje trokuta.

Potrebno je napisati i grafičkom procesoru proslijediti shader koji će obrađivati podatke za svaki vrh. U shaderu za vrhove potrebno postaviti vektor pozicije koji će GPU dalje koristiti. Također možemo proslijediti neke druge podatke kroz shader koje će onda neki daljnji shader koristiti.

U našem slučaju, koordinate karakterističnih točaka u teksturi samo ćemo proslijediti dalje. Koordinate karakterističnih točaka u videu iskoristit ćemo za definiranje pozicije. Iscrtavamo u 2D grafici tako da ćemo postaviti samo x i y koordinatu dok ćemo ostale za sada postaviti na 1.0. Vidljivo polje unutar OpenGLa je u rasponu $[-1, 1]$ za x i y koordinatu, a VisageTracker vraća koordinate u rasponu $[0, 1]$ pa je prije potrebno primijeniti afinu transformaciju. Kod konačnog shadera za vrhove napisan je niže.

```
#version 330 core
layout (location = 0) in vec2 position;
layout (location = 1) in vec2 maskPosition;
out vec2 MaskPosition;
void main()
```

```
{
    gl_Position = vec4(position.x * 2.0 - 1.0, position.y *
        2.0 - 1.0, 1.0, 1.0);
    MaskPosition = maskPosition;
}
```

Nakon shadera za vrhove vrhovi se povezuju u trokute kako smo specificirali poslanim poljem indeksa. Ti trokuti zatim se rasteriziraju i stvaraju se fragmenti. Potrebno je napisati shader za fragmente čija je glavna svrha odrediti boju fragmenta, tj. boju budućeg slikovnog elementa na ekranu.

U shaderu za vrhove proslijedili smo samo koordinate teksture za vrhove, ali ne i za svaki pojedini fragment trokuta. OpenGL se pobrine da se izlazni parametri shadera za vrhove linearno interpoliraju po trokutu te se izračuna njihova vrijednost za svaki pojedini fragment. Pomoću interpoliranih podataka i ugrađene funkcije unutar shadera za fragmente možemo dohvatiti boju aktivne teksture na željenoj poziciji. Tu boju proslijedit ćemo kao izlaznu boju. Konačni shader za fragmente prikazan je u nastavku.

```
#version 330 core
in vec2 MaskPosition;
out vec4 color;
uniform sampler2D Texture;
void main()
{
    color = texture(Texture, MaskPosition);
}
```

Važno je primijetiti da je prije crtanja potrebno stvoriti i za OpenGL kontekst vezati shader program. Isto je potrebno napraviti za teksturu. Vezanjem teksture za kontekst automatski se postavlja pretpostavljena uniformna varijabla `Texture`. Uniform u GLSL jeziku predstavlja globalnu varijablu koja je zajednička za sve vrhove ili fragmente.

Konačno, možemo se pozabaviti načinom na koji se prenose podatci na GPU i načinom na koji će se ti podatci interpretirati. Za ovo će nam biti potrebna tri tipa objekata koje OpenGL specificira. To su *vertex buffer object (VBO)*, *vertex array object (VAO)* i *element buffer object (EBO)*.

VBO je spremnik koji sprema podatke o atributima vrhova. EBO je spremnik koji sadrži redoslijed vrhova koje koristimo, odnosno indekse vrhova za trokute. VAO je objekt uz koji povezujemo odgovarajuće VBO i EBO objekte te pomoću kojeg defini-

ramo način na koji će se interpretirati podatci u VBO (pomoću naredbe `glVertexAttribPointer`).

U OpenGL-u svaki puta kada želimo obavljati operacije nad objektom, radimo to tako da prvo postavimo taj objekt kao aktivni objekt, tj. vežemo ga za OpenGL kontekst. Sve naredbe koje potom slijede obavljaju se nad aktivnim objektom, sve dok se ne postavi neki drugi aktivni objekt, ili oslobodimo trenutno vezani objekt. Slijedi kod u kojem se stvaraju ovi objekti i postavljaju atributi.

```
// stvaranje objekata
glGenVertexArrays(1, &vertexArrayObject);
glGenBuffers(1, &vertexBufferObjectPosition);
glGenBuffers(1, &vertexBufferObjectMaskPosition);
glGenBuffers(1, &elementBufferObject);

// vezemo VAO za kontekst
glBindVertexArray(vertexArrayObject);

// vezemo EBO za VAO (i kontekst)
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, elementBufferObject);

// vezemo VBO objekte i opisujemo kako interpretirati podatke
   u poljima
glBindBuffer(GL_ARRAY_BUFFER, vertexBufferObjectPosition);
glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 2 *
   sizeof(GLfloat), (GLvoid*) 0);
glEnableVertexAttribArray(0);
glBindBuffer(GL_ARRAY_BUFFER, vertexBufferObjectMaskPosition);
glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, 2 *
   sizeof(GLfloat), (GLvoid*) 0);
glEnableVertexAttribArray(1);

// oslobadja aktivni VAO
glBindVertexArray(0);
```

Nakon stvaranja objekata, potrebno je prenijeti podatke na GPU. To možemo učiniti odmah na početku programa (npr. podatci za masku), ili za svaki sličicu videa (podatci koordinata videa). Kod za oba slučaja prikazan je niže. Glavnu razliku čini posljednji argument funkcije `glBufferData`, koji OpenGL-u poručuje koliko često da očekuje promjenu podataka u VBO-u.

```
// popunjavamo VBO polje podatcima iz FaceData strutkure za
   masku
for (int i = 0; i < numOfVertices; i++) {
   verticesDataMaskPostion[2 * i] =
      maskFaceData.featurePoints2D->
      getFPPos(fdpGroupAndIndex[i][0],
         fdpGroupAndIndex[i][1])[0];
   verticesDataMaskPostion[2 * i + 1] =
      maskFaceData.featurePoints2D->
      getFPPos(fdpGroupAndIndex[i][0],
         fdpGroupAndIndex[i][1])[1];
}

// prenosimo polja na GPU
glBindBuffer(GL_ARRAY_BUFFER, vertexBufferObjectMaskPosition);
glBufferData(GL_ARRAY_BUFFER, sizeof(GLfloat) * 2 *
   numOfVertices,
      verticesDataMaskPostion, GL_STATIC_DRAW);
```

```
// popunjavamo VBO polje podatcima iz FaceData strutkure za
   video
for (int i = 0; i < numOfVertices; i++) {
   verticesDataPosition[2 * i] =
      cameraFaceData.featurePoints2D->
      getFPPos(fdpGroupAndIndex[i][0],
         fdpGroupAndIndex[i][1])[0];
   verticesDataPosition[2 * i + 1] =
      cameraFaceData.featurePoints2D->
      getFPPos(fdpGroupAndIndex[i][0],
         fdpGroupAndIndex[i][1])[1];
}
// popunjavamo EBO polje podatcima o indekisma vrhova trokuta
for (int i = 0; i < numOfTriangles; i++) {
   elementDrawingData[3 * i] = elementData[3 * i];
   elementDrawingData[3 * i + 1] = elementData[3 * i + ];
   elementDrawingData[3 * i + 2] = elementData[3 * i + ];
}
```

```
// prenosimo podatke iz VBO-a i EBO-a na GPU
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, elementBufferObject);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(GLuint) * 3 *
   numOfTriangles,
      elementDrawingData, GL_STREAM_DRAW);
glBindBuffer(GL_ARRAY_BUFFER, vertexBufferObjectPosition);
glBufferData(GL_ARRAY_BUFFER, sizeof(GLfloat) * 2 *
   numOfVertices,
      verticesDataPostion, GL_STREAM_DRAW);
```

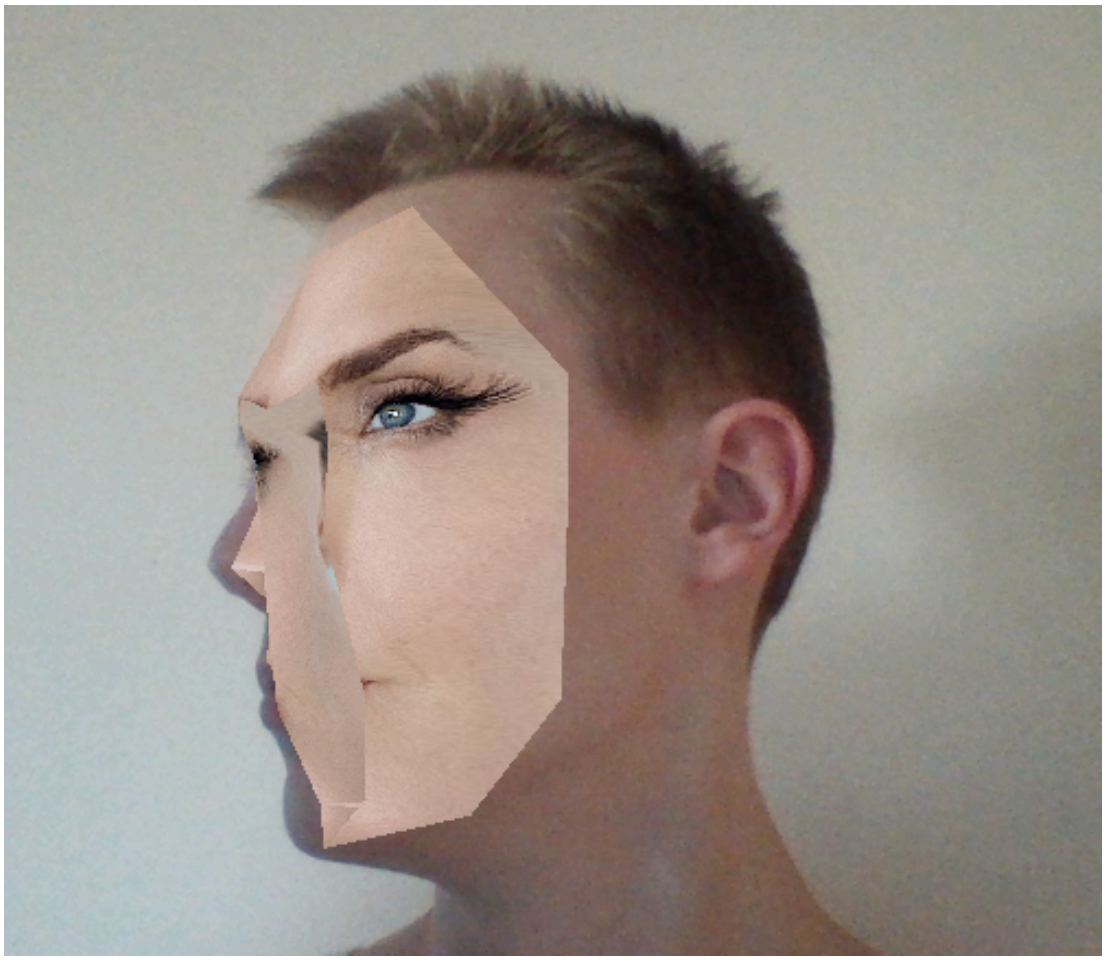
Konačni rezultat crtanja maske možemo vidjeti na slici 4.2.

Slika 4.2: Rezultat crtanja teksturiranih trokuta

4.2. Provjera dubine

S trenutnom implementacijom maska izgleda normalno kada se gleda sprijeda, no čim okrenemo glavu u stranu ili neki slični položaj u kojem dolazi do preklapanja trokuta koje crtamo, vidimo da nastaju problemi. Ponekad se iscrtavaju trokuti koji bi se trebali

nalaziti sa stražnje strane lica i ne bi trebali biti vidljivi. Nepoželjni efekt ilustriran je na slici 4.3.



Slika 4.3: Krivo iscrtavanje stražnjih trokuta

Visage|SDK osim 2D koordinata daje i 3D koordinate točaka u prostoru u odnosu na kameru. Ovo nam omogućava da u OpenGL-u radimo provjeru dubine i time eliminiramo slikovne elemente koji bi trebali biti sakriveni, tj. one slikovne elemente koji su udaljeniji od kamere.

Naredbama `glEnable(GL_DEPTH_TEST);` i `glDepthFunc(GL_LEQUAL);` omogućimo provjeravanje dubine i odaberemo funkciju koja će odbacivati fragmente. Odabrana funkcija ne će odbacivati slikovne elemente sa z-koordinatom jednakom ili manjom od slikovnog elementa na istom položaju. Kako bi provjeravanje dubine funkcioniralo potrebno je u shaderu za vrhove postaviti z-koordinatu, te ćemo zbog toga shaderu za vrhove prosljeđivati tri koordinate umjesto dvije. Prve dvije koordinate će i dalje biti 2D koordinate, a treća će biti z-koordinata koja ima značenje udaljenosti od kamere. Raspon z-koordinate koju prihvaća OpenGL je od -1 do 1. Raspon ko-

ordinata koje `VisageTracker` vraća nije unaprijed poznat. Zato ćemo, prije nego pošaljemo sve koordinate na GPU, proći po njima te odrediti najmanju i najveću.

Najmanju i najveću z-koordinatu proslijedit ćemo shaderu za vrhove kao uniforme kako bi on mogao skalirati dobivene koordinate na raspon $[-0.9, 0.9]$ te ih tako proslijediti kao dubinu. Sličici dobivenoj iz kamere dubinu ćemo postaviti na 1.0 kako bi se uvijek nalazila iza maske. Promjene u shaderu za vrhove prikazane su niže. Rezultati su prikazani na slici 4.3.

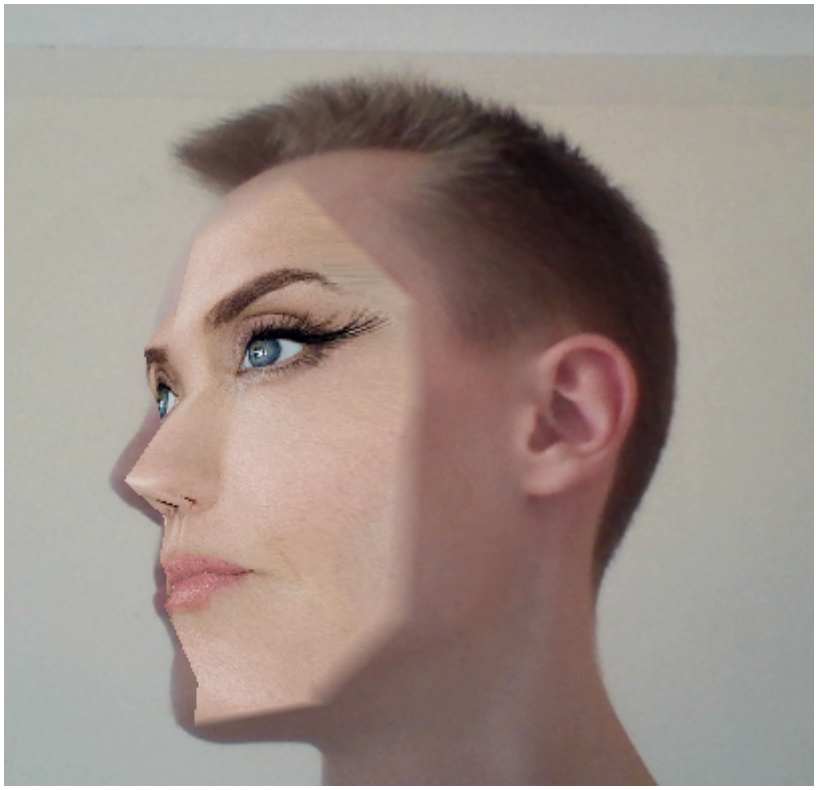
```
#version 330 core
layout (location = 0) in vec3 position;
layout (location = 1) in vec2 maskPosition;
uniform float Near;
uniform float Far;
out vec2 MaskPosition;
void main()
{
   float Depth = (position.z - Near) / (Far - Near) * 1.8 -
      0.9;
   gl_Position = vec4(position.x * 2.0 - 1.0, position.y *
      2.0 - 1.0, Depth, 1.0);
   MaskPosition = maskPosition;
}
```

4.3. Ublažavanje rubova

Konačno, želimo da nam maska izgleda kao da je naslikana na licu. Kako bi ostvarili ovaj učinak koristit ćemo činjenicu da slikovni element koji crtamo može imati svojstvo prozirnosti. Do sada su svi slikovni elementi koje smo crtali bili potpuno neprozirni.

Odlučujemo da će karakteristične točke na rubu lica biti više prozirne od točaka unutar samog lica. Zato je na GPU potebno poslati još jedno polje podataka koje će za svaku točku lica definirati treba li biti potpuno ili djelomično prozirna. U shaderu za vrhove ovaj podatak samo ćemo proslijediti daljnjim koracima na obrađivanje. Kod je prikazan niže

```
#version 330 core
layout (location = 0) in vec3 position;
layout (location = 1) in vec2 maskPosition;
```

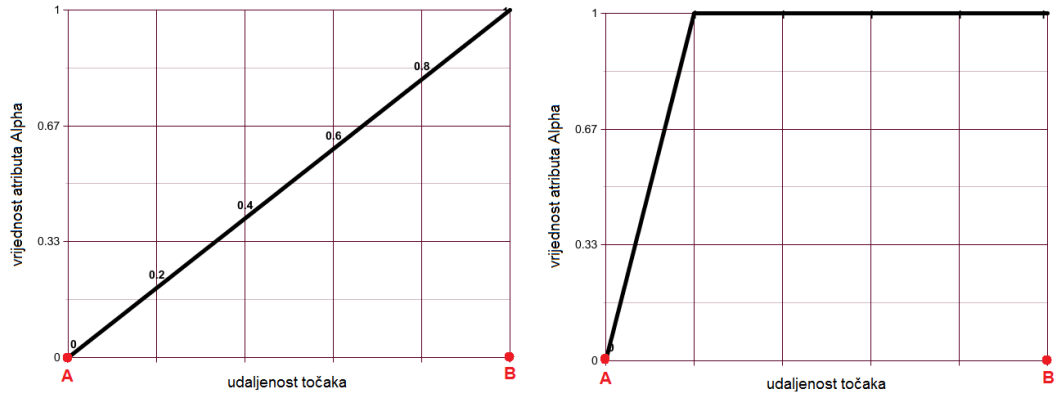


Slika 4.4: Rezultati crtanja s provjerom dubine

```
layout (location = 2) in float alpha;
uniform float Near;
uniform float Far;
out vec2 MaskPosition;
out float Alpha;
void main()
{
   float Depth = (position.z - Near) / (Far - Near) * 1.8 -
      0.9;
   gl_Position = vec4(position.x * 2.0 - 1.0, position.y *
      2.0 - 1.0, Depth, 1.0);
   MaskPosition = maskPosition;
   Alpha = alpha;
}
```

Za rubne točke lica prema dogovoru slat ćemo za vrijednost atributa `alpha` 0.0, a za središnje točke vrijednost 1.0. Već je rečeno da će OpenGL vrijednost tog atributa linearno interpolirati po fragmentima, no mi ne želimo da u rubnim trokutima prozirnost bude linearno raspoređena, već želimo da prozirnost naglo pada tek uz sam

rub lica. Ako pretpostavimo da se točka A nalazi na samom rubu lica, a točka B na središnjem dijelu, slika 4.5 prikazuje usporedbu interpolacije koju će OpenGL učiniti, i učinka koij mi želimo.



Slika 4.5: Usporedba linearne i nelinearne interpolacije atributa

S obzirom da znamo da će vrijednost atributa `Alpha` u shaderu za fragmente uvijek biti interpolirana između vrijednosti 0.0 i 1.0, lako možemo unutar shadera za fragmente konstruirati funkciju koja će napraviti korekciju. Kod shadera za fragmente prikazan je niže, a konačan rezultat prikazan je na slici 4.6.

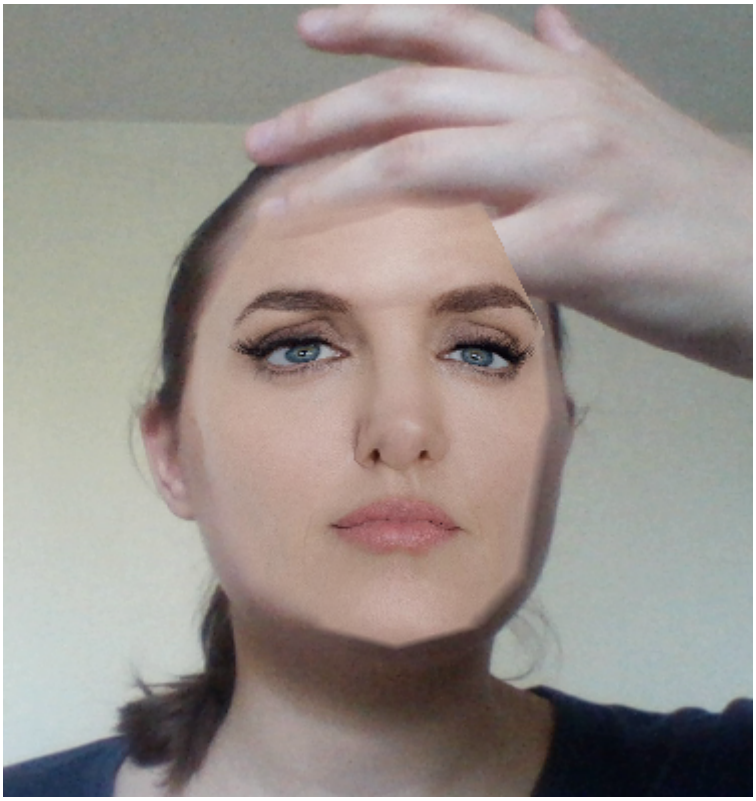
```
#version 330 core
in vec2 MaskPosition;
in float Alpha;
out vec4 color;
uniform sampler2D Texture;
void main()
{
    float newAlpha;
    if (newAlpha >= 0.2) {
        newAlpha = 0.8;
    } else {
        newAlpha = Alpha * 4 ;
    }
    color = vec4(texture(Texture, MaskPosition).rgb, newAlpha);
}
```

Slika 4.6: Prikaz djelomično prozirne maske

5. Problemi i potencijalna poboljšanja

Iako smo uspjeli ostvariti iscrtavanje maske preko lica, još uvijek postoje problemi koje se mogu uočiti i koje bi u budućnosti bilo dobro ispraviti.

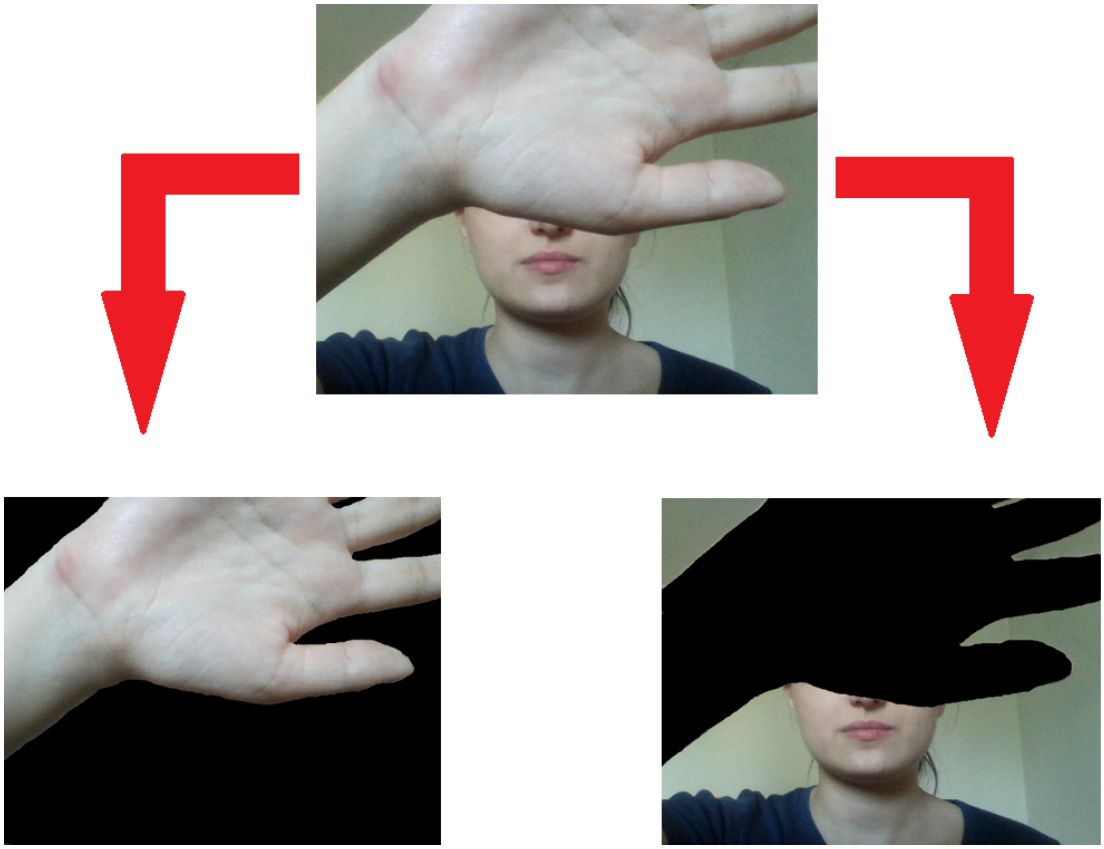
Jedan od glavnih problema s crtanjem maske javlja se kada se preko lica nalazi neki drugi objekt poput ruke ili kose. U idealnom slučaju, u pozadini slike nalazilo bi se lice i prostor iza osobe, zatim bi se ispred toga nalazila maska, i preko toga bi se nalazila nacrtana ruka. U ovakvoj implementaciji to nije slučaj, već se maska uvijek nalazi ispred svega ostaloga. Ilustrirani problem vidimo na slici 5.1.



Slika 5.1: Problem objekta koji se trebao nalaziti ispred maske

Potencijalno buduće rješenje bilo bi da se sličica videa segmentira i podijeli na slikovne elemente koji se nalaze ispred i iza lica. Tada bi bilo moguće prvo iscrtati po-

zadinu, zatim masku, i zatim objekte koji prekrivaju virtualnu masku. Primjer moguće segmentacije dan je na slici 5.2.



Slika 5.2: Segmentacija slike temeljem dubine

Izazov se javlja u tome kako da računalno odluči koji slikovni elementi se nalaze ispred a koji iza maske. Ovo je još jedno složeno pitanje kojim se bavi računalni vid, i nažalost, ulaziti u taj problem bilo bi preopsežno u svrhe ovoga rada.

Idući problem posljedica je nedovoljnog broja točaka u mreži trokuta maske. Zbog ovoga rubovi maske čine se neprirodnima jer su opisani ravnim i izlomljenim linijama trokuta umjesto prirodnom zaobljenom linijom lica. Problem bi bio lako rješiv povećanjem broja korištenih točaka u mreži. Ovo bi, doduše, dodatno zakompliciralo stvaranje životinjskih maski [3.2], a i značajno je ograničeno brojem točaka koje pruža korištena biblioteka za detekciju točaka. Drugo potencijalno riešenje bilo bi automatsko generiranje dodatnih točaka iz postojećih točaka.

Konačno, iako nije uvijek vidljivo, računalno određivanje karakterističnih točaka još uvijek nije potpuno precizno te u nekim situacijama dolazi do većeg izražaja. Primjer se nalazi na slici 4.4 gdje linija maske ne prati točno liniju lica u području nosa.

6. Zaključak

Zadatak je rada bio ostvariti aplikaciju koja automatski postavlja i iscrtava masku povrh lica u videu. Zadatak je inspiriran sve većom popularnošću video komunikacije putem mobilnih uređaja. Promatramo trend sve većeg broja aplikacija kojima je isključivi obogatiti i učiniti zabavnijom komunikaciju između korisnika. Samo jedan primjer sličnih aplikacija su Face Swap Live [7], koja detektira dva lica u videu i zamijeni ih, te aplikacija MSQRD [9], koja crta maske preko lica. Primjera ima još mnogo.

Za ostvarenje aplikacije potrebno je odabrati prikladnu biblioteku računalnog vida. Možemo se odlučiti za biblioteku sa većom ili manjom sposobnosti generalizacije. Naravno, veća sposobnost generalizacije također znači i manju preciznost. Primjer biblioteke s većom generalizacijom čine one često korištene u aplikacijama za zamjenu dvaju lica. Često vidimo na internetu slike gdje se ljudsko lice zabunom zamijeni s utičnicom na zidu i sl. Primjer biblioteke sa većom preciznošću i detektiranju specifično ljudskoga lica je Visage|SDK korištena u ovom radu.

Uz odgovarajuću biblioteku računalnog vida, problem postaje značajno jednostavniji, te vidimo da možemo s lakoćom postići rezultate. Problemi se više javljaju kad nam cilj također postane postići uvjerljive rezultate, koji izgledaju zaista kao da je riječ o stvarnoj masci nacrtanoj na licu. U radu su napravljena neka postignuća u tom smjeru, no ima još prostora za poboljšanje. Naravno, treba uzeti u obzir želimo li postići efekt maske koja je naslikana na koži, ili fizički odvojene maske. Sve u svemu, zadatak rada je ostvaren i otvara mogućnosti za daljnju razradu.

LITERATURA

[1] OpenCV. URL https://en.wikipedia.org/wiki/OpenCV. [datum pristupa: 31.5.2016.].

[2] Visage Technologies AB. Visage|SDK. URL https://visagetechnologies.com/products-and-services/visagesdk/. [datum pristupa: 31.5.2016.].

[3] Visage Technologies AB. *Visage|SDK documentation*. Visage Technologies, 2016. [verzija biblioteke: 8.1 za Windows].

[4] Joey de Vries. Learn OpenGL. URL http://learnopengl.com. [datum pristupa: 15.5.2016.].

[5] GLFW. GLFW - An OpenGL Library. URL http://www.glfw.org/. [datum pristupa: 31.5.2016.].

[6] HOTLab. Teme za studentske radove. URL http://hotlab.fer.hr/HOTlab/students_hr/teme. [datum pristupa: 30.5.2016.].

[7] Laan Labs. Face Swap Live. URL http://faceswaplive.com/. [datum pristupa: 13.6.2016.].

[8] OpenGL.org. OpenGL Overview. URL https://www.opengl.org/about/. [datum pristupa: 31.5.2016.].

[9] Masquerade Technologies. MSQRD. URL http://msqrd.me. [datum pristupa: 16.3.2016.].

[10] Wikipedia. Computer vision. URL https://en.wikipedia.org/wiki/Computer_vision. [datum pristupa: 30.5.2016.].

Virtualne maske pokretane praćenjem lica

Sažetak

Računalni vid područje je računarstva koje teži tome da razvije algoritme koji računalima omogućuju procesuiranje slike na način sličan ljudima. U ovom radu računalni vid koristi se za iscrtavanje maske povrh ljudskog lica u video slici. Upotrebljava se biblioteka računalnog vida Visage|SDK koja omogućuje detekciju lica i njegovih karakterističnih točaka. Iz tih karakterističnih točaka stvara se mreža povezanih trokuta koji sačinjavaju lice. Odgovarajuće karakteristične točke lica (ujedno i točke mreže) potrebno je detektirati unutar videa, povrh kojeg se želi iscrtati maska, i slike maske. Pomoću tih točaka zatim se preslikava tekstura maske na odgovarajući trokut u video slici.

Ključne riječi: Računalni vid, Visage|SDK, OpenGL, OpenCV, detekcija lica, karakteristične točke lica, mreža trokuta.

Virtual masks driven by face tracking

Abstract

Computer vision is a field of computing whose purpose is inventing algorithms that enable computers to process images in a way similar to humans. In this paper computer vision is used for drawing masks over the human face in a video sequence. Visage|SDK, a library of computer vision, is used to detect faces and their feature points. From those feature points a triangle mesh is made that makes up the face. The facial feature points (the vertices of the mesh) have to be detected in the video, over which we want to draw the mask, as well as in the image of the mask. By using these points we can draw the texture from the mask over the corresponding triangle in the video frame.

Keywords: Computer vision, Visage|SDK, OpenGL, OpenCV, face detection, facial feature points, triangle mesh.