Nefotorealistični prikaz 3D modela korištenjem cel-shadera

Udovičić, Stjepan

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU

FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA

Sveučilišni studij

NEFOTOREALISTIČNI PRIKAZ 3D MODELA KORIŠTENJEM CEL-SHADERA

Diplomski rad

Stjepan Udovičić

Osijek, 2016.

Sadržaj

1. Uvod

Razvojem novih tehnologija i uređaja, postoji kontinuirana potreba za napretkom računalne grafike, kako u mogućnostima tako i performansama. Iako je primarni fokus ovoga napretka bio foto realističan prikaz, ponekada je potreban drugačiji pristup iz praktičnih ili estetskih razloga. Kako bi grafička sučelja omogućila slobodu obrade slike, a u isto vrijeme i kontrolu nad performansama, koriste se programska sučelja - API [1]. Jedno od najpoznatijih grafičkih aplikacijski sučelja je OpenGL [2].

Za prikaz 3D modela u realnom vremenu, potrebno je izvršiti niz zahtjevnih matematičkih operacija kako bi se odredio položaj i boja svake točke [3] na ekranu. Takve operacije najčešće nisu dovoljno brze na procesoru samog računala, pa se za te potrebe koristi specijalizirano sklopovlje grafičkih kartica. Pristup tom sklopovlju omogućen nam je preko OpenGL-a koji omogućava izvršavanje vlastitih programa izravno na grafičkim karticama, pri čemu se ostvaruje velika brzina rada zbog paralelnog načina obrade podataka.

U ovom radu opisano je kako se koristi moderni OpenGL za prikaz modela. Opisuje se način učitavanja podataka i njihov prijenos u memoriju grafičke kartice. Izradom vlastitih programa za sjenčanje [4], prikazano je kako se vrši programiranje grafičkih kartica za obradu slike s ciljem postizanja specijalnih efekata.

1.1. Zadatak diplomskog rada

Cilj je razviti prilagođeni sustav sjenčanja za OpenGL koji će 3D modele prikazivati nefotorealistično u svrhu stvaranja dojma kako su ručno nacrtani. Potrebno je razviti aplikaciju koja omogućava učitavanje 3D modela iz datoteke te ga prikazuje korištenjem prilagođenog sustava sjenčanja. Funkcionalnost razvijene aplikacije treba testirati prikazivanjem nekoliko 3D modela.

[1] engl. Application programming interface - set definicija, protokola i alata za izradu programa

[2] engl. Open Graphics Library

[3] engl. pixel - najmanja jedinica u računalnoj slici

[4] engl. shader

2. OpenGL aplikacijsko sučelje

OpenGL razvio se kao nasljednik IrisGL-a [5]. Njegov glavni nedostatak bio je što su mogućnosti samog sučelja ovisile o mogućnostima sklopovlja, te se nije mogao primijeniti na različitim uređajima. Zbog potrebe za izradom standarda, početkom 1990-ih godina tvrtka Silicon Graphics Inc. (SCI) započela je sa izradom OpenGL specifikacije kako bi formalno definirala aplikacijsko programsko sučelje (API) prema grafičkim karticama. Godine 1992., prva verzija OpenGl-a je objavljena. Od 2016.g. ne-profitna grupa Khronos zadužena je za razvoj OpenGL-a [1].

Primarni zadatak ovog API-a je prikaz 2D i 3D vektorske grafike. On omogućuje komunikaciju sa grafičkim procesorom (GPU) s ciljem sklopovskog ubrzanja grafičkog prikaza, neovisno o programskom jeziku i operativnom sustavu.

Zbog svoje proširenosti i upotrebe, postao je industrijski standard. Podržan je na velikom broju uređaja, od računala do pametnih telefona. Ima široku primjenu u industriji (CAD [6], GIS [7], simulacijama i vizualizacijama) te u izradi računalnih igara.

2.1. OpenGL grafički cjevovod

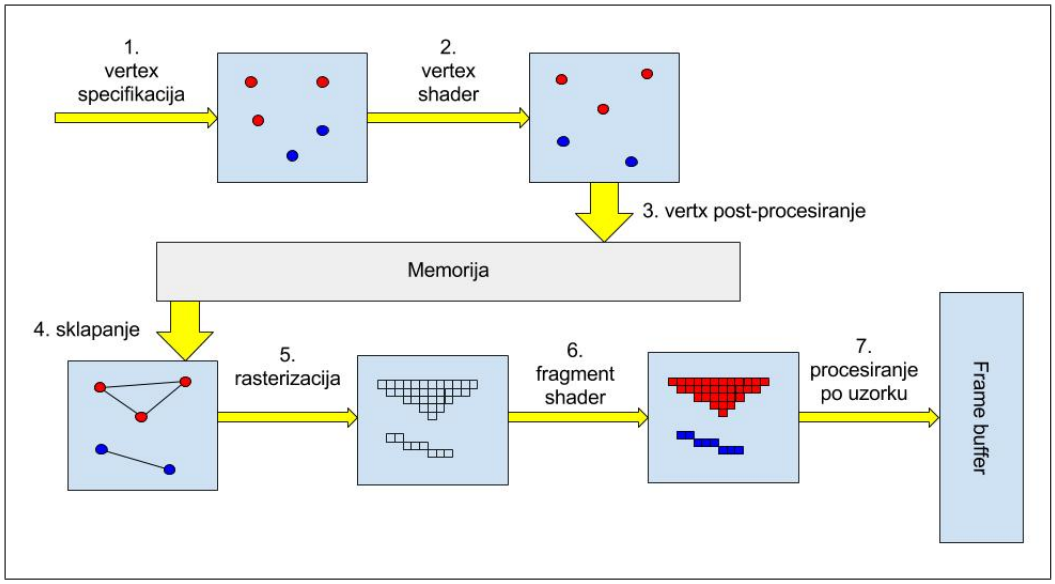
U svojim početcima, prilikom rada s računalnom grafikom nije bilo puno mjesta za manipulaciju sa slikom. Grafičko sklopovlje nije omogućavalo puno manipulacije sa bojom i pozicijom objekata na ekranu. Računalo je *slalo* opise vektore i teksturu koju oni stvaraju, dok su grafičke kartice bila zadužene za stvaranje slike iz ta dva dobivena podatka.

Takav način rada podrazumijevao je da se sva potreban kalkulacija (tranformacija koordinata - pomak) i sjenčanje prethodno odrade na samom računalu, prije nego što se pošalju grafičkoj kartici za prikaz. Glavni nedostatak tog pristupa je što svu kalkulaciju morala obrađivati centralna procesorska jedinica (CPU) koja osim što nije bila dizajnirana za takve operacije, je morala obrađivati i niz drugih podataka istovremeno.

Iz tog razloga, pojavila se potreba za programabilnim *grafičkim cjevovodom* [8] koji bi omogućio

[5] engl. Integrated Raster Imaging System Graphics Library
[6] engl. Computer-Aided Design
[7] engl. Geographic Information System
[8] engl. Graphival pipline

manipulaciju podacima na grafičkoj kartici. OpenGL cjevovod sastoji se od 7. komponenti[2], prikazanih na slici 2.1.



Slika 2.1: Dijagram toka OpenGL cjevovoda

1. **Vertex specifikacija:** U ovom koraku aplikacija prenosi podatke grafičkoj kartici - opise vertex-a [9]. Način na koji će se vertex tumačiti/iscrtavati se kasnije obrađuje.

2. **Vertex shader:** Izvršavanje vertex shader korisničkog programa. Cilj ovog koraka je transformirati ulaznu poziciju vektora u njegov krajnji oblik. Ovdje se najčešće vrše transformacije s ciljem postizanja pogleda iz perspektive, rotacije i uvećanja. U ovoj fazi pokreće se i geometrijski shader, koji radi kao i vertex shader, samo na razini ploha.

3. **Vertx post-procesiranje:** Rezultati prošlog koraka spremaju se za na to predviđene memorijske lokacije.

4. **Sklapanje ploha:** U ovome koraku, grafički procesor sklapa ploha od unaprijed obrađenih vertexa.

5. **Rasterizacija:** Sklopljene plohe razdvajaju se na fragment koji se prosljeđuju dalje kako bi im se odredila boja, odnosno tekstura.

6. **Fragment shader:** Svaki fragment se prosljeđuje korisničkom programu fragment shaderu) čiji izlaz predstavlja boju danog fragmenta.

[9]Točka u trodimenzionalnom prostoru. Osim pozicije, vertex može sadržavati i druge informacije, poput boje

7. **Procesiranje po uzorku:** Ovaj korak služi za izvršavanje raznih testova koji mogu utjecati na krajnji rezultat, primjerice test dubine (ukoliko se fragment nalazi iza nekog drugog, vjerojatno neće biti prikazan). Također ovdje se vrše operacije i odbacivanja fragmenata koji nisu na vidljivom djelu ekrana, stapanje boja i sl.

Kod ovoga koraka bitno je napomenuti kako su procesi optimizirani na način da se što više operacija može izvršavati paralelno. Današnje grafičke kartice imaju i po nekoliko stotina jezgri koje mogu paralelno raditi, što omogućava paralelno izvršavanje nekoliko shadera istovremeno, što u konačnici rezultira velikom brzinom obrade podataka, nešto što nije moguće na centralnoj procesorskoj jedinici,

2.2. Shaderi i GLSL

Kao što je opisano u prethodnom poglavlju, shaderi su najbitnije komponente progamabilnog grafičkog cjevovoda. Oni su zapravo male korisničke aplikacije koje se izvode paralelno, i obrađuju manji set informacija od jednom.

Korisničke aplikacije za vertex i fragment shader (opcinoalno i geometrijski) se dostavljaju grafičkoj kartici, i zatim komapjliraju u jedan *program* koji se izvodi u sklopu cjevovoda. Bitno je napomenuti da je moguće prirediti više od jednog programa, te ih mijenjati tokom izvođenja.

Shaderi se pišu programskom jeziku koji je posebno dizajniran za ovu primjenu - GLSL [10]. GLSL programski jezik baziran je na C programskom jezu, i dio je OpenGL specifikacije. Razlikuje se od C-a u nekoliko ključnih stvari: nadograđen je da podržava matrične operacije i tipove podataka. Za razliku od C-a, ne podržava preopterećenje funkcija na osnovu ulaznih parametara niti rekurzivne funkcije.

Svaki shader rasolaže s nekoliko ulaznih varijabli:

- **Ulazne varijable:** Svaki shader imaju svoju glavnu ulaznu varijablu. Za vertex shader to je *position*, vektor s četiri vrijednosti koji opisuje početnu poziciju objekta u 3D prostoru. Četvrta vrijednost (w) označava orijentaciju. U slučaju fragment shadera, stvar je malo složenija, jer se samom fragmentu ne mora nužno dodijeliti samo boja, već se može dodjeliti

[10] engl. OpenGL Shading Language

i komad teksture. Naziv samih varijabli ovisi o načinu učitavanja modela, odnosno ime se proizvoljno dodjeljuje kod učitavanja modela.

- **Uniform varijable:** Ovo su varijable koje su dostupne svim shaderima. Njihovu vrijednost postavlja glavni program prije početka rendering procesa, te ona ostaje ista tokom cijelog izvođenja. Ove varijable se najčešće koriste za prijenos MVP[11] matrica, koje su obrađene u sljedećem poglavlju.

- **Varying varijable:** Ovo su varijable koje su dostupne svim shaderima, no njihovu vrijednost postavlja jedan shader, kako bi određenu informaciju mogao proslijediti dalje niz cjevovod. Primjerice, normala vertexa ulazni je parametar vertex shadera. No ta vrijednost često je potrebna fragmen shaderu kako bi pravilno odredio sjenčanje objekta. Iz toga razloga, često se definira *varying* varijabla, čija se vrijednost postavlja unutar vertex shadera za daljnje korištenje.

Izlaz iz vertex shader-a je varijabla *gl_Position* [12], dok je izlaz fragment shader-a varijabla *gl_FragColor* [13]. Iako su ovi izlazi nužni za rad, pojedini shaderi su često zaduženi za kalkulaciju i niz drugih *varying* varijabli koje će se koristiti dalje u grafičkom cjevovodu.

2.3. Transformacije objekta

Kako je ranije spomenuto, ideja programabilnog grafičkog cjevovoda je pomaknuti grafičke kalkulacije sa centralnog procesora na grafički procesor. To prvenstveno znači rekalkulaciju pozicije i oblika objekta uslijed pomaka samog objekta ili kamere. To se najčešće vrši množenjem kordinata pojedine točke objekta sa takozvanom MVP [14] matricom. MVP matrica se sastoji iz tri dijela [3]:

1. **Matrica modela:** Matrica koja opisuje osnovne transformacije modela: translaciju (2–1), rotaciju (2–2)-(2–4) i skaliranje (2–5). Dobiva se množenjem istoimenih matrica.

[11] engl. Model View Projection

[12] Interna OpenGL varijabla koja opisuje položaj trenutnog vertexa u 3D prostoru

[13] Interna OpenGL varijabla koja opisuje boju treutnog fragmenta

[14] engl. Model View Projection

$$T = \begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2–1}$$

$$R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos(\alpha) & -sin(\alpha) & 0 \\ 0 & sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{2–2}$$

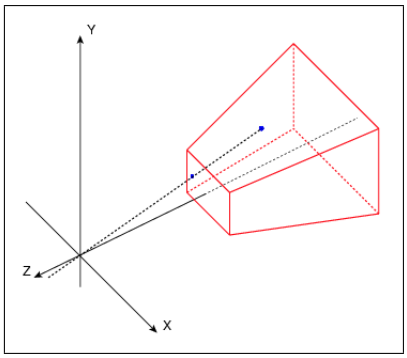
$$R_y = \begin{bmatrix} cos(\alpha) & 0 & sin(\alpha) & 0 \\ 0 & 1 & 0 & 0 \\ -sin(\alpha) & 0 & \cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{2–3}$$

$$R_z = \begin{bmatrix} cos(\alpha) & -sin(\alpha) & 0 & 0 \\ sin(\alpha) & \cos(\alpha) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2–4}$$

$$S = \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2–5}$$

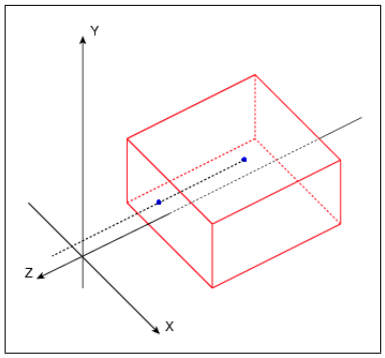
2. **Matrica scene (view):** Budući da se u računalnoj grafici kamera ne pomiče, već se pomiče svijet oko nje - ova matrica služi kako bi ispravno pozicionirala objekt u odnosu na kameru.

3. **Projekcijska matrica:** Služi za definiranje transformacije s ciljem postizanja željene projekcije. To je najčešće perspektivna projekcija (prikazana na slici 2.2 te opisana u 2–6), no može biti ortogonalna (prikazana na slici 2.3 te opisana u 2–7) prikazana na slici i bilo kako drugo proizvoljno definirana.



Slika 2.2: Perspektivna projekcija[7]

$$P_p = \begin{bmatrix} \frac{2 * blizu}{desno - ljevo} & 0 & \frac{desno + ljevo}{desno - ljevo} & 0 \\ 0 & \frac{2 * blizu}{gore - dolje} & \frac{gore + dolje}{gore - dolje} & 0 \\ 0 & 0 & -\frac{daleko + blizu}{daleko - blizu} & -\frac{2 * daleko * blizu}{daleko - blizu} \\ 0 & 0 & -1 & 0 \end{bmatrix} \quad (2\text{–}6)$$



Slika 2.3: Ortogonalna projekcija[7]

$$P_o = \begin{bmatrix} \frac{2}{desno - ljevo} & 0 & 0 & -\frac{desno + ljevo}{desno - ljevo} \\ 0 & \frac{2}{gore - dolje} & 0 & -\frac{gore + dolje}{gore - dolje} \\ 0 & 0 & -\frac{2}{daleko - blizu} & -\frac{daleko + blizu}{daleko - blizu} \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (2\text{–}7)$$

Sam izračun MVP matrice, obično se vršri na centralnoj procesorskoj jedinici i kao takav se dostavlja grafičkoj kartici. Ona je zatim zadužena da svaki pojedini vertex pozicionira na pravo

mjesto, uzimajući MVP matricu u obzir - pomnoži koordinate vertxa sa MVP matricom da dobije konačnu poziciju u prostoru.

Na ovaj način, smanjuje se količina posla koju obavlja centralna procesorska jedinica. Ona samo računa način tranformacije, dok se stvarni pomak modela odvija na samoj grafičkoj kartici, koja je u mogućnosti to obaviti neusporedivo brže, za svaki vertex posebno.

2.4. WebGL

WebGL [15] je JavaScript grafički API koji je nastao na temelju OpenGL ES 2.0 standarda, te služi za hardwersko ubrzanje prilikom prikazivanja 3D grafike na internet preglednicima. Razvoj standarda vrši Khronos WebGL grupa, a sam standard podržan je od strane svi vodećih internet preglednika koji podržavaju HTML 5 canvas element.

Iako se zapravo ne radi o istom standardu kao i OpenGL, WebGL zadržava što je moguće više sličnosti sa OpenGL-om radi lakog prijenosa aplikacija s jednog API-a na drugi.

Kao i OpenGL, WebGL također ima programabilan grafički cjevovod, uz ograničenje da ne podržava geometrijski shader. Ostatak funkcionalnosti je isti, i ponaša se na identičan način.

2.5. Modeli u Three.js JSON formatu

Kao i OpenGL, WebGL propisuje funkcije za pristupanje grafičkom sučelju na vrlo niskoj razini, što za korisnika znači da mora sam voditi računa o alokaciji memorije i prijenosu podataka. Iz tog razloga, postoji mnogo frameworka koji se zaduženi za olakšavanje rada s nekim osnovnim stvarima.

Za potrebe ovoga rada, korišten je three.js [16], WebGL framework koji olakšava poslove poput prijenosa modela u memoriju grafičke kartice, kao i postupak kompajliranja korisničkih sahadera.

Kao format spremanja 3D modela, korišten je three.js JSON format, budući da je s JSON formatom najlakše raditi u *JavaScript*-u. Unutar datoteke, model je opisan kroz nekoliko ključnih elemenata [4]:

[15] engl. Web Graphics Library
[16] http://threejs.org/ verzija 73

- **vertices:** Popis vertex, pri čemu je svaki opisan sa tri koordinate: x,y i z. Ovo polje more biti proizvoljne duljina, ali je bitno da je višekratnik broja 3.
- **normals:** Kao i vertex-i, normale su opisane sa tri koordinate. Njihov broj pak ovisi o vrsti ravnina koju vertex-i opsiuju: trokuti, četverokuti, ...
- **faces:** Ovo polje opisuje koji vertexi i normale (proizvoljno i teksture i boje) čine jednu plohu. Prvi broj u nizu definira masku pod kojom se učitavaju ostali podaci. Evo nekoliko primjera:
  - **0, 0,1,2** : Vodeća 0 označava da slijede tri vertexa koji međusobno tvore trokut. To su vertexi 0,1 i 2, odnosno prvih 9 koordinata koje su prosljeđene u polju *vertices*.
  - **1, 0,1,2,3** : Vodeći 1 označava da slijede četiri vertexa koji međusobno tvore četverokut. To su vertexi 0,1,2 i 3.
- **metadata:** Ovo polje sadrži opisne podatke poput verzije samog formata, broj vertexa, normala i ploha, informacije o programu s kojim je model kreiran i sl.
- **scale:** Pomoću ovog parametra omogućeno je jednostavno skaliranje modela.
- **name:** Ovo polje sadrži ime modela.

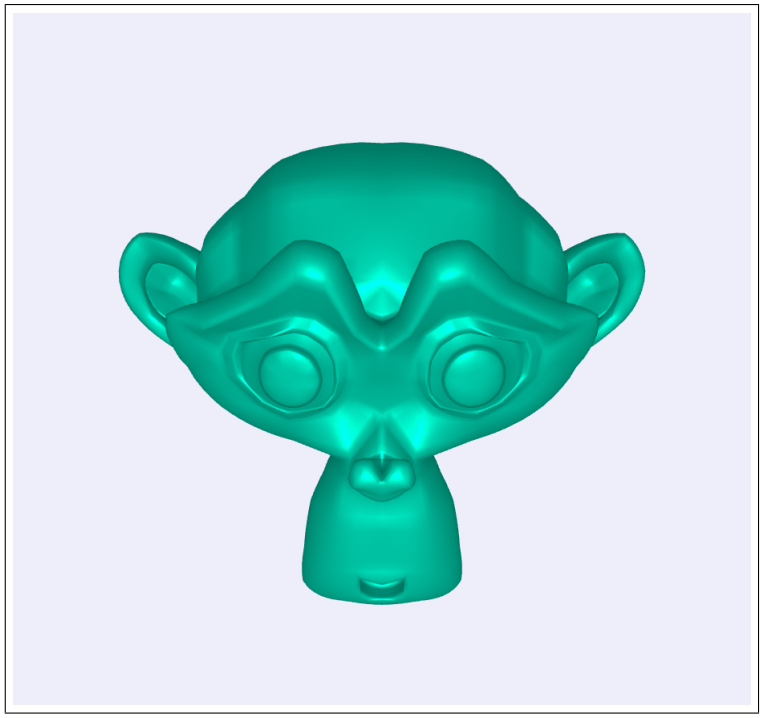
Učitavanje modela vrši se preko *JSONLoader* objekta koji je sastavni dio *three.js* frameworka. Njemu se izravno pruža *JSON* polje u tekstualnom obliku. Sam objekt zadužen je za alociranje memorije i prijenos na grafičku karticu, uključujući i potrebno skaliranje, ako je tako određeno u samome modelu.

3. Sjenčanje (shading)

Sjenčanje je jedan od najbitnijih dijelova u 3D grafici. U ovome procesu određuje se boja i osvjetljenje pojedinoga fragmenta, te u konačnici i cijeloga modela. Iako ona ne mora nužno ovisiti o nekom izvoru svjetlosti, najčešće to nije slučaj.

Ovaj proces obrađuje se u fragment shader programu, te je u potpunoj kontroli korisnika. Ponekada, sjenčanje fragmenta se mora ponoviti više puta kako bi se postigao željeni slučaj. Takav način rada zove se iscrtavanje u više iteracija [17]. Taj princip je korišten u postizanju toon shading efekta kako je objašnjeno dalje u ovome poglavlju.

Jedan od najjednostavnijih primjera sjenčanja je *difuzno sjenčanje*, koji daje rezultat vrlo sličan onome kako ljudsko oko percipira predmete oko sobe. Jedan takav primjer prikazan je na slici 3.4.



Slika 3.4: Model osjenčan difuznom rasvjetom

Difuzno sjenčanje radi na sljedećem principu: intenzitet osnovna boja modela se pojačava, odnosno smanjuje u ovisnosti o kutu koju zatvara zraka svjetlosti i normale plohe na koju ta zraka pada. Kako je prikazano jednadžbom (3–8), množenjem matrice normala plohe i matrice pozicije

[17] engl. Multipass rendering

izvora svjetlosti, dobiva se faktor pojačanja. Za konačni rezultat, potrebno je faktor pojačanja pomnožiti sa osnovnom bojom modela.

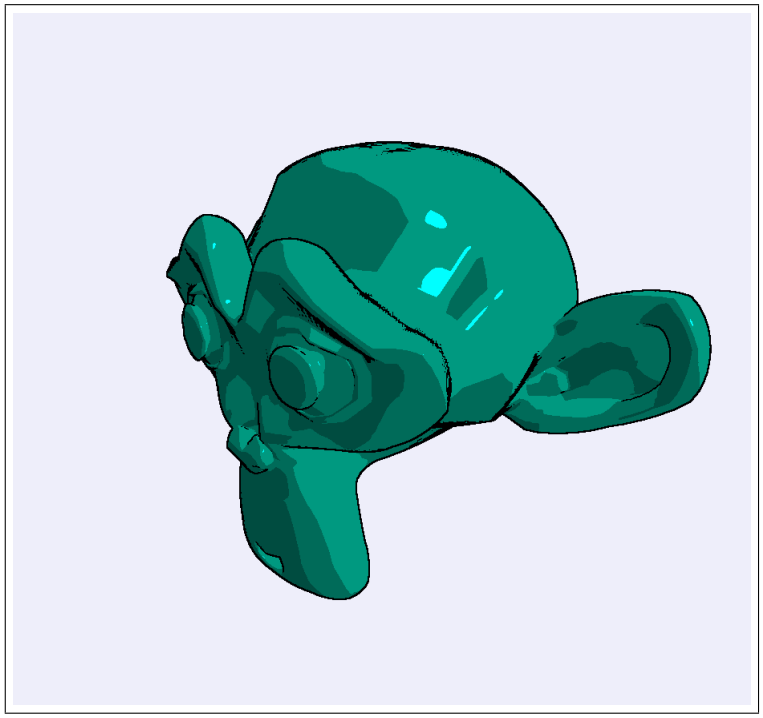
$$C = (N \cdot L) * B \tag{3–8}$$

Gdje je N matrica normale plohe, L matrica pozicije svijetla te B matrica osnovne boje modela.

3.1. Toon-shading princip

Toon-shading naziv dolazi od engleske riječki *cartoon*, što označava crtani film. Vrsta sjenčanja dobila je naziv zbog toga što se isti princip koristio u crtanim filmova kako bi se na jednostavan način prikazao 3D prostor.

Efekt se postiže na način da se modelu ocrtaju rubovi, te da se osjenča pojednostavljenim difuznim sjenčanjem - umjesto računanja faktora pojačanja za svaku pojedinu plohu, iste se dijele na zone čiji kut svijetla i normale pripada određenoj toleranciji. Na taj način na modelu se razlikuju samo tri do četiri tona osnovne boje modela. Primjer *toon shading* načina sjenčanja prikazan je na slici 3.5.



Slika 3.5: Model sjenčan toon shaderom

Kako bi bilo moguće postići efekt, potrebno je napraviti sljedeće koraka:

1. Ekstrakcija rubova
2. Pojednostavljeno difuzno sjenčanje
3. Stapanje slojeva u jednu sliku

Budući da je *fragment shader* zadužen za obradu samo jednoga *fragmenta*, on nije svjestan svoje okoline. Točnije, prilikom obrade *fragmenta* u *fragment shaderu*, još nije određeno koji će biti njegovi susjedni *fragmenti*. Kao posljedica toga dolazi činjenica da ekstrakciju rubova nije moguće odraditi u jednoj iteraciji. Iz tog razloga, *toon shader* zahtjeva sjenčanje u više iteracija, točnije u njih četiri.

Kao posljedica toga dolaze performanse *toon shadera*, koji je (u ovoj implementaciji) četiri puta sporiji od nekih jednostavnijih *shadera*, poput difuznoga sjenčanje. Iako za današnje grafičke procesore ovo nije problem, uvijek je potrebno smanjiti kompleksnost gdje je to moguće. S tim ciljem, implementacija ovog algoritma biti će na način da se druga faza ekstracije rubova odvija u isto vrijeme kada i spajanje slojeva kako bi ukupan broj iteracija smanjili za jedan i ubrzali algoritam.

Sljedeća poglavlja opisuju svaki korak zasebno, te finalno spajanje slojeva u konačnu sliku koja se prikazuje korisniku.

3.2. Iscrtavanje obrisa i rubova

Prvi korak u *toon shading*-u je ekstrakcija rubova. Iako postoji mnogo načina da se ovaj korak uradi, najjednostavniji je korištenjem *Canny*-evog algoritma detekcije rubova. Budući da se ta metoda zasniva na pronalaženju razlike u kontrastu susjednih *pixel*-a slike, potrebno je posebno prirediti sliku na kojoj će se detektor izvršiti.

Za te potrebe, u prvoj iteraciji iscrtana je slika modela u kojoj je boja svakoga *pixel*-a predstavljena kao njegova udaljenost od kamere. U ovome slučaju, kamera je pozicionirana ispred samoga modela, tako da je udaljenost jednaka z koordinati *pixel*-a u koordinatnom sustavu. Rezultat ove iteracije prikazan je na slici 3.6.

Slika 3.6: Model s iscrtanom *dubinom* pixel-a

Kako je prethodno napomenuto, samu detekciju rubova spojili smo u istoj iteraciji sa samim spajanjem slojeva. No za potrebe analize, ovaj korak biti će objašnjen kao zasebna cjelina.

Canny-ev algoritam detekcije rubova[5] zasniva se na principu traženja razlike u kontrastu susjednih *pixel*-a. U koliko ta razlika prelazi određenu vrijednost, smatra se da trenutni *pixel* predstavlja rub na slici.

Kontrast između susjednih *pixel*-a. određuje se tako da se susjedstvo piksela pomnoži sa *maskom*. Zbroj svih elemenata dobivene matrice predstavlja vrijednost trenutno *pixel*-a. Budući da *Canny*-eva *maska* radi samo u jednome smjeru, detekciju je potrebno izvršiti dva puta: jednom u smjeru x te jednom u smjeru y kooridnatnog sustava slike. *Maske* korištene za ovu potrebu predstavljene su matricama C_x i C_y prikazanim u (3–9).

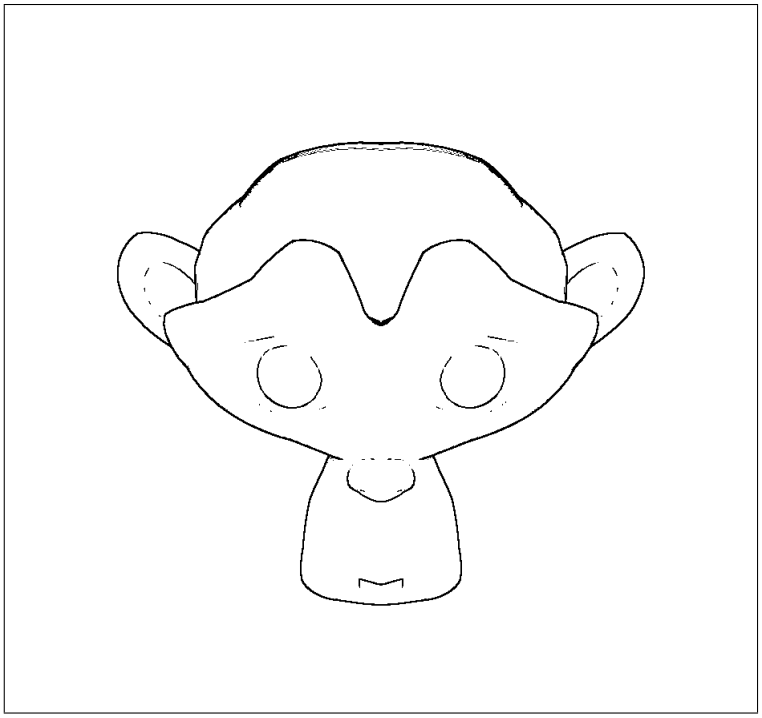
$$C_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}, C_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \tag{3–9}$$

U koliko sumu svih elemenata matrice C_x predstavimo kao s_x, a sumu svih elemenata matrice C_y predstavimo kao s_y, tada se vrijednost trenutnoga *pixel*-a t računa prema 3–10.

$$t = \sqrt{{s_x}^2 + {s_y}^2} \tag{3–10}$$

Za svaki *pixel* čija vrijednost t prelazi zadanu granicu smatra se kao rub modela. Za potrebe algoritma, rubovi su iscrtani crnom bojom, dok su svi ostali *pixel*-i ostavljeni bez boje, kako bi ih kasnije mogli lakše stopiti sa osjenčanom slikom.

Sama granična vrijednost nije unaprijed definira *Canny*-evim algoritmom, već se proizvoljno određuje. Za potrebe ovog *shader*-a, određeno je da će ta vrijednost iznositi 0.02. Konačni rezultat prikazan je na slici 3.7.



Slika 3.7: Model s iscrtanim obrisima

Kao što je vidljivo na slici 3.7, potrebno je napomenuti da ova metoda ne detektira sve rubove ispravno. No budući da u slučaju *toon shader*-a rubovi služe samo za naglašavanje oblina, visoka preciznost nije potrebna, te je ovaj rezultat dovoljno dobar.

3.3. Sjenčanje

Za potrebe sjenčanja modela u *toon shader*-u koristi se pojednostavljeno difuzno sjenčanje. Umjesto da se faktor pojačanja svjetlosti računa na osnovi svakoga fragmenta posebno, on je predodređen

za dane intervale. Na taj način dobiva se model sa unaprijed predodređenim brojem tonova. Za potrebe ovog *toon shader*-a određena su četiri tona osnovne boje.

Prvo je potrebno odrediti faktor pojačanja svjetlosti t na isti način kao što se radi prilikom difuznog sjenčanja: Matricu N koja predstavlja normale fragmenta potrebno je pomnožiti sa matricom L koja predstavlja poziciju svjetla u prostoru kako je prikazano jednadžbom (3–11).

$$t = N \cdot L \tag{3–11}$$

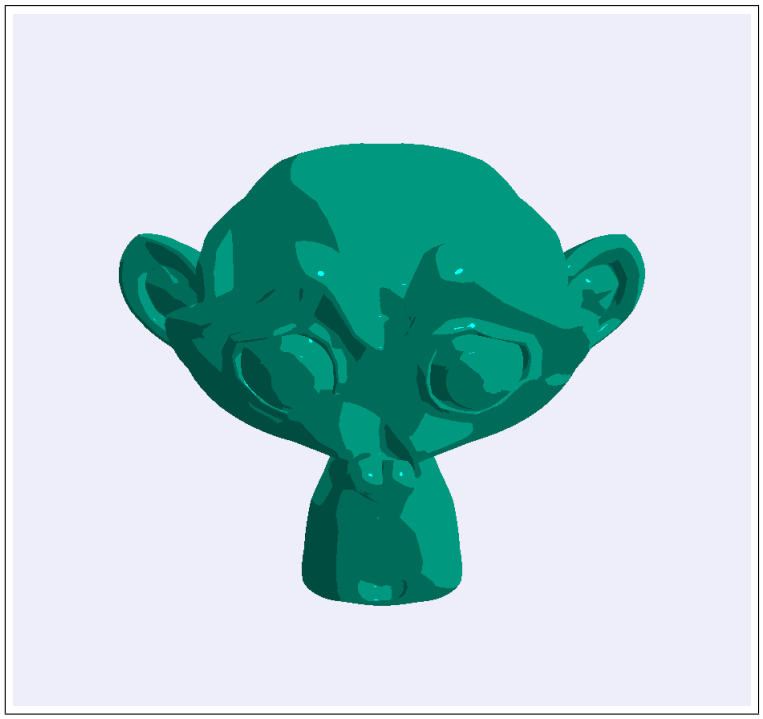
No za razliku od (3–8), gdje je faktor pomnožen izravno sa osnovnom bojom modela, ovdje ćemo konačni faktor pojačanja svjetlosti t_f odrediti iz unaprijed određenih intervala na osnovu t prema funkciji$f(t)$ opisanoj u (3–12).

$$f(t) = \begin{cases} 0.5, & t \leq -0.6 \\ 0.7, & -0.6 < t \leq 0.1 \\ 1, & 0.1 < t \leq 0.97 \\ 3, & t > 0.97 \end{cases} \tag{3–12}$$

Konačni boju fragmenta C dobivamo iz osnovne boje modela B i konačnog faktora pojačanja svjetlosti t_f prema (3–13).

$$C = t_f * B \tag{3–13}$$

Primjenom navedenoga u korisničkom *fragment shader*-u dobivamo rezultat prikazan na slici 3.8, koji predstavlja posljednji korak prije stapanja slojeva u jednu sliku.



Slika 3.8: Osjenčani model

3.4. Spajanje slojeva

Rezultate prethodnih međukoraka potrebno je smjestiti u međuspremink, odnosno kao teksturu koja se prosljeđuje *shader* programu u trećoj iteraciji. Te teksture prikazane su slikama 3.6 i 3.8. Ovdje je potrebno raditi sa teksturama, jer jedino na taj način *shader* programu možemo omogućiti rad nad susjedstvom *pixel*-a.

Posljednja iteracija *fragment shader*-a tada prvo provjerava dali se trenutni fragment treba tretirati kao rub. U koliko da, dodjeljuje mu se crna boja. U koliko ne, dodjeljuje mu se odgovarajuća boja sa prethodno osjenčanoga modela.

Na taj način postiže se efekt da je slika 3.7 stavljena ispred 3.8, odnosno krajnji rezultat *toon shading* algoritma.Taj rezultat prikazan je na slici 3.9.

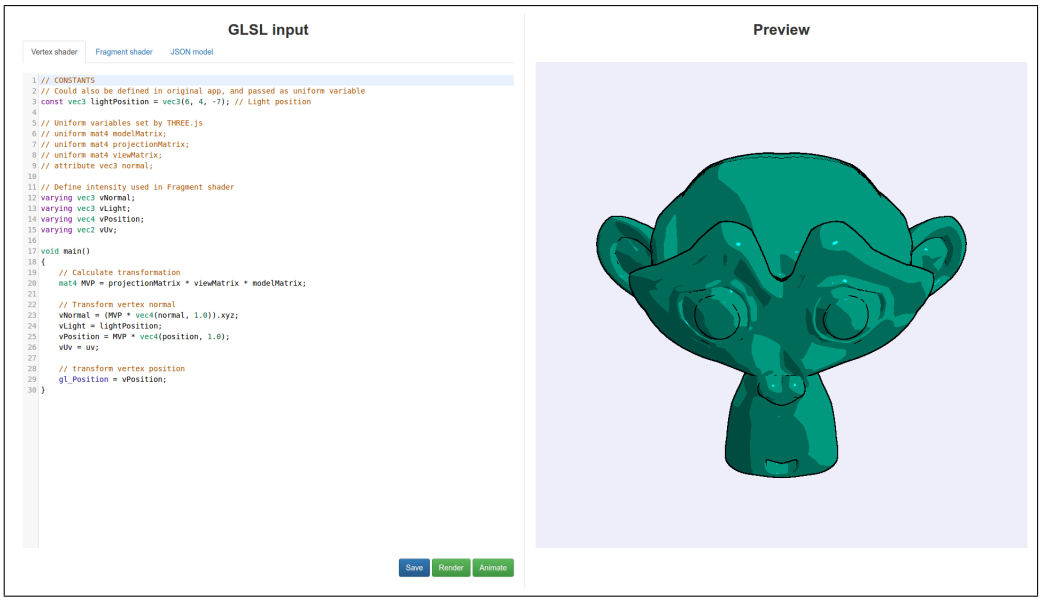
Slika 3.9: Rezultat dobiven spajanjem slojeva

4. Implementacija

Prethodna poglavlja opisivala su metodu *toon shading*-a te način implementacija koji se svodi na izradu *shader* programa. No sam *shader* program, izvodi se na grafičkoj kartici. U ovom poglavlju opisuje se sučelje koje je izrađeno za potrebe rada, koje krajnjem korisniku omogućava izmjenu modela i *shader* programa. Opisuje se način implemntacije, te interakcija sa grafičkom karticom korištenjem *three.js* biblioteke za rad s WebGL-om.

Aplikacija je izrađena pomoću HTML5 tehnologije i *JavaScrip* programskog jezika. Na taj način, omogućeno je izvršavanje neovisno o platformi na većini modernih internet preglednika, te uređajima koji podržavaju HTML5 *canvas* element. Budući da mogućnost izvršavanja aplikacije ovisi o grafičkom procesoru i karakteristikama samoga uređaja, korištena je biblioteka *modernizer*[18] za određivanje sklopovske podrške samoga uređaja na kojemu se aplikacija izvršava.

Aplikacija podržava statični prikaz modela te animaciju rotacijom modela oko svoje osi, kako bi se mogućnosti *shader* korisničkoga programa mogle proučiti iz više kutova pod različitom rasvjetom. Izrađen je na način da se prilagođava veličini ekrana, kako bi se mogla izvršavati na uređajima raznih veličina. Korisničko sučelje prikazano je na slici 4.10.



Slika 4.10: Korisničko sučelje aplikacije

[18]https://modernizr.com/

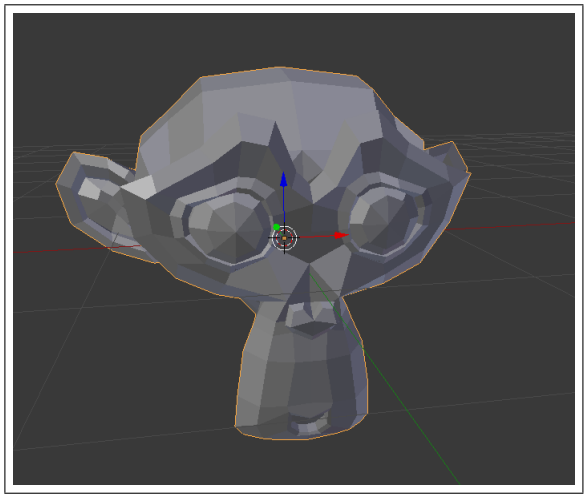
4.1. Korištenje *three.js*-a

Three.js [6] biblioteka napisana je u JavaScript programskom jeziku i služi za kreiranje i prikaz 3D sadržaja u internet preglednicima. Sama biblioteka omogućava krajnjem korisniku brži pristup grafičko kartici, te olakšava procese učitavanja i prijenosa modela. Također omogućava lakši rad sa korisničkim *shader* programima.

Prva javno dostupna verzija biblioteke pojavila se početkom 2010.g. Od onda se razvija kao projekt otvorenoga koda pod *MIT licencom*. Svoju popularnost počela je postizati godinu dana kasnije, kada je internet preglednik *Firefox* razvio podršku za WebGL. Trenutno broji preko 600 suradnika koji su doprinijeli izvornom kodu.

4.2. Učitavanje modela

Aplikacija učitava modele u *three.js JSON* formatu. Većina današnjih alata za 3D modeliranje podržava ovaj format nativno ili kroz dodatke koji se naknadno instaliraju na aplikaciju. Za potrebe ovoga rada korišten je programski alat *Blender*[19], pomoću kojega je u *three.js JSON* formatu izvezena prilagođena verzija modela *monkey* koji dolaz sa alatom, prikazan na slici 4.11.



Slika 4.11: Osnovni *Monkey* model iz alata *Blender*

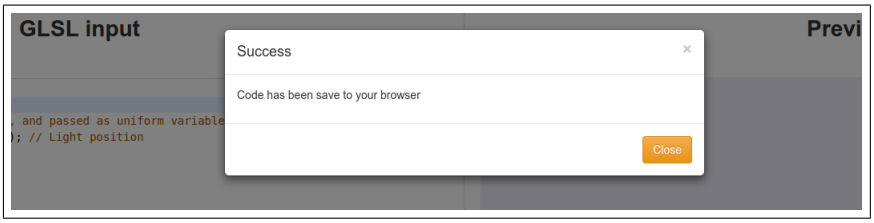
Unutar same aplikacije korišten je *three.js* osnovni objekt *THREE.JSONLoader* [20] koji je zadužen

[19] https://www.blender.org/

[20] http://threejs.org/docs/api/loaders/JSONLoader.html

za kreiranje samoga modela na grafičkoj kartici preko ulazne datoteke. *THREE.JSONLoader* objektu se prosljeđuje sadržaj same datoteke u tekstualnom formatu.

Korisničko sučelje aplikacije učitava sadržaj datoteke na način da korisnik sadržaj mora unijeti u za to predviđeno mjesto (na kartici *JSON model*) te potvrditi svoj unos pritiskom na tipku *Save*. U tome trenutku aplikacija će korisnički unos spremiti u lokalno spremište internet preglednika, kako korisnik ne bi morao ponavljati ovu radnju prilikom sljedećega korištenja aplikacije. Prilikom uspješnoga spremanja izmjena korisnik dobiva obavijest kao što je prikazano na slici 4.12.



Slika 4.12: Obavijest krajnjem korisniku nakon spremanja izmjena

Učitani model potom se prosljeđuje objektu *THREE.Mesh* [21], koji služi za daljnju interakciju modela i aplikacije. Budući da se sam model nalazi na grafičkoj kartici, a ne u radnoj memoriji računala, *THREE.Mesh* kao povratnu informaciju aplikaciji vraća identifikacijsku oznaku modela, koja ga jednoznačno označava u memoriji grafičke kartice. Na taj način vrši se komunikacija i interakcija između modela i ostatka aplikacije.

4.3. Učitavanje korisničkoga *shader* programa

Slično kao i učitavanja samoga modela, vrši se i učitavanje *shader* korisničkoga programa. Na karticama *Vertex shader* i *Fragment shader* predviđen je unos korisničkih programa. Aplikacija podržava naglašavanje sintakse, kako bi omogućila krajnjem korisniku olakšani rad. Sam unos vrši se na način da korisnik mora unijeti izvorni kod *shader* korisničkoga programa u za to predviđena mjesta. Kao i prilikom unosa modela, nakon unosa programskog koda, korisnik mora potvrditi svoj unos pritiskom na tipku *Save*, kako bi se programski kod spremio u lokalno spremište internet preglednika za naknadnu upotrebu. Prilikom uspješnoga spremanja izmjena korisnik dobiva obavijest kao što je prikazano na slici 4.12.

[21] http://threejs.org/docs/#Reference/Objects/Mesh

Korištenjem *three.js* osnovnog objekta *THREE.ShaderMaterial* [22] kreira se *materijal* na kojemu se vrši sjenčanje korisničkim programom. Prilikom stvaranja materijala, objekt prima nekoliko ulaznih parametara:

- **vertexShader:** Izvorni kod *vertex shader* korisničkog programa koji je prethodno unesen
- **fragmentShader:** Izvorni kod *fragmentShader* korisničkog programa koji je prethodno unesen
- **uniforms:** *Uniform* varijable koje će biti proslijeđene *shader* korisničkom programu prilikom izvršavanja.

Za potrebe ove aplikacije korišteno je nekoliko *uniform* varijabli, od kojih je najbitnija *pass* koja označava indeks sloja koji se trenutno obrađuje, kako je opisano u poglavlju 3. ovoga rada. Izuzev *pass* varijable, *shader* programu potrebno je proslijediti i reference na dvije teksture koje su mu potrebne za rad, kao što je kasnije pojašnjeno u poglavlju 4.4., koje programu omogućuju izvršavanje u više iteracija.

Sam objekt *THREE.ShaderMaterial* zadužen je prijenos izvornoga koda na grafičku karticu, te njegovo kompajliranje. Uslijed uspješnog prijenosa, objekt *THREE.ShaderMaterial* nam vraća jednoznačnu referencu na izvršni program u memoriji grafičke kartice koji se dalje koristi za interakciju s ostatkom aplikacije. No u koliko prijenos nije bio uspješan, odnosno došlo je do pogreške (najčešće uslijed pogreške u izvornom kodu korisičkog *shader*-a), *three.js* biblioteka nas o tome obavještava putem konzole internet preglednik.

Budući da konzolni prozor internet preglednika nije odmah dostupno, poželjno ga je otvoriti prilikom rada s aplikacijom, kako bi pravovremeno uočili pogreške. Također, aplikacija izrađena za potrebe ovoga rada i sama otkriva određene pogreške, te o tome obavještava krajnjega korisnika kako bi se pogreška uočila i prije otvaranja konzolnog prozora. Jedan takav primjer prikazan je na slici 4.13. gdje je došlo do pogreške prilikom učitavanja modela.

[22] http://threejs.org/docs/api/materials/ShaderMaterial.html

Slika 4.13: Obavijest o pogrešci prilikom izvođenja aplikacije

4.4. Korištenje više slojeva

Kao što je prethodno opisano u poglavlju 3., ova implementacija *toon shader* načina sjenčanja zahtjeva više iteracija za postizanje željenoga efekta. Budući da grafički cjevovod, kako je opisano u poglavlju 2.1., radi na razini *pixel*-a i fragmenata, pojedina iteracija nema pristup cjelovitoj slici, niti je moguće rezultat u konačnici vratiti na ulaz cjevovoda. Iz toga razlika, korištene su dvije teksture kako je prethodno opisano.

Teksture u *OpenGL*-u su slike koje se mogu *nalijepiti* na materijal pojedinog objekta na sceni. Sam *fragment shader* zadužen je za mapiranje teksture na objekt. Iz toga razloga, teksture su naprijed učitane u memoriju grafičke kartice, te su u potpunosti dostupne svakom fragmentu u *fragment shader* korisničkom programu. Dostupnost cijele slike omogućava pretraživanje susjedstva *pixel*-a opisanoga u poglavlju 3.2., za svaki fragment pojedinačno.

Kako bi omogućili *shader* korisničkom programu dostupnost rezultata iz prethodnih iteracija, rezultat prve dvije iteracija sprema se izravno u odgovarajuću teksturu u memoriji grafičke kartice, umjesto da se prikazuje korisniku na ekran. Na posljednjoj iteraciji, *shader* korisnički program ima pristup objema teksturama (iscrtanoj dubini modela prikazanoj na slici 3.6 i osjenčanoj verziji modela prikazanoj na slici 3.8), te je taj korak zadužen za stapanje rezultata i konačni prikaz na korisnikov ekran, odnosno *HTML5 canvas* element na grafičkom sučelju krajnjega korisnika.

4.5. Pokretanje aplikacije

Nakon što je korisnik unio izvorni kod *shader* programa te nakon što je unio 3D model, potrebno je odabrati način prikaza modela:

- **Statični prikaz:** Pritiskom na tipku *Redner* pokreće se statični prikaz osjenčanoga modela gdje je model okrenut izravno prema kameri. Dobiva se rezultat kao što je prikazan na slici 3.9.

- **Animirani prikaz:** Pritiskom na tipku *Animate* pokreće sa animacija modela, na način da se model okreće oko svoje osi. Na taj način omogućava se pregled iz raznih kuteva, kao što je prikazano na slici 3.5.

Korisničko sučelje omogućava kontinuirane izmjene na *shader* korisničkom programu i 3D modelu. Nakon napravljenih izmjena, potrebno je ponovno odabrati način rada, te će izmjene biti odmah vidljive. Ovdje je potrebno napomenuti kako *three.js*[23] ne podržava izmjene svih komponenti na sceni. Iz tog razloga, za veće izmjene u radu *shader* korisničkoga programa, potrebno je spremiti izmjene i osvježiti prozor internet preglednika, kako bi se aplikacija nanovo inicijalizirala, te kako bi se svi resursi nanovo učitali u memoriju grafičke kratice.

[23] Verzija 73, koja je korištena za potrebe ovoga rada

5. Zaključak

U ovom radu je prikazan, objašnjen i programski ostvaren *toon shader* način sjenčanja, kao i popratna aplikacija koja omogućava izvršavanje *shader* korisničkoga programa. U teorijskom dijelu opisane su korištene tehnologije, način rada modernih grafičkih kartica kao i osnove tranformacije objekata u 3D prostoru, kao i način rada *toon-shading* algoritma. Također je opisan način rada i korištenja korisničke aplikacije koja se koristi za pregled rezultata i unos podataka.

Na temelju izrađenoga programskog rješenja prikazan je način korištenja *OpenGL*-a, odnosno *WebGL*-a pomoću *three.js* biblioteke. Za potrebe rada samoga algoritma opisan je način detekcije rubova na računalnoj slici, te difuzni način sjenčanja, kao i njegova pojednostavljena varijanta. Prikazano je na koji način se implementiraju algoritmi sjenčanje koji zahtjeva rad u više iteracija, te kako se problem međuspremnik rješava korištenjem *OpenGL* teksturama.

Izradom korisničkog sučelja, prikazan je način implementacije koji omogućava izvršavanje aplikacije neovisno o uređaju i operativnom sustavu na kojem se aplikacija izvršava, pri čemu su korištene *HTML 5* tehnologije i *JavaScript* programski jezik

Prilikom izrade aplikacije, uočeno je ograničenje *OpenGL* implementacije, koje onemogućava implementaciju algoritma u jednoj iteraciji. Iako je stapanje određenih koraka u jednu iteraciju smanjilo kompleksnost algoritma, on i dalje zahtjeva više od jednoga koraka. Kao posljedica ovoga, krajnja aplikacija je tri puta sporija od jednostavnijih *shader* korisničkih programa.

Literatura

[1] https://www.opengl.org/wiki/History_of_OpenGL, posjećeno 1. lipnja 2016.g.

[2] https://www.opengl.org/wiki/Rendering_Pipeline_Overview, posjećeno 1, lipnja 2016.g.

[3] D. Ginsburg, B. Purnomo: OpenGL ES 3.0 Programming Gide, Second Edition, 2014.g.

[4] https://github.com/mrdoob/three.js/wiki/JSON-Model-format-3, posjećeno 3. lipnja 2016.g.

[5] http://docs.opencv.org/2.4/doc/tutorials/imgproc/imgtrans/canny_detector/canny_detector.html, posjećeno 10. lipnja 2016.g.

[6] https://github.com/mrdoob/three.js, posjećeno 15. lipnja 2016.g.

[7] http://blender.stackexchange.com/questions/648/what-are-the-differences-between-orthographic-and-perspective-views, posjećeno 6. Rujna 2016.g.

Sažetak

U ovom diplomskom radu je opisan i programski ostvaren algoritam za *toon shading* način sjenčanja 3D modela, kao i popratna korisnička aplikacija koja krajnjem korisniku omogućava izvršavanje *shader* korisničkoga programa na grafičkoj kartici računala. Prikazan je način rada algoritma, kao i način implementacije korištenjem *JavaScript* programskog jezika i *JavaScript* biblioteke. Neke od prednosti ovakvoga pristupa su mogućnost izvršavanja programskoga koda neovisno o platformi i operativnom sustavu, te jednostavnost implementacije. Uočeni nedostaci algoritma su nemogućnost implmentacije u jednoj iteraciji, što u konačni negativno utječe na performanse aplikacije.

Ključne riječi sjenčanje 3D modela, detekcija rubova, OpenGL, WebGl, Toon shader, sjenčanje u više iteracija

Abstract

This thesis describes and implements algorithm for *toon shading* of 3D models, as well as a supporting application which enables end user to execute shader program on computers graphic card. It also demonstrates how underlying algorithm works, as well as implementation of it using *JavaScript* programming language and *JavaScript* library. Some of the advantages of this approach are cross platform and operating system independent execution of application, as well as ease of implementation. Observed disadvantage of this approach is forced use of multipass rendering, which has negative impact on applications performance.

Keywords shading of 3D models, edge detection, OpenGL, WebGL, Toon shading, multipass rendering

Životopis

Stjepan Udovičić rođen je 15. svibnja 1988 u Osijeku u Republici Hrvatskoj. 2003. godine upisuje se u III. gimnaziju u Osijeku, koju je završio 2007. godine. Tijekom tog razdoblja sudjelovao je natjecanjima iz kemije i informatike. 2007.godine upisuje se na sveučilišni preddiplomski studij računarstva na Elektrotehničkom fakultetu u Osijeku, koji dovršava 2010.g, te nastavlja obrazovanje na sveučilišnom diplomskom studiju procesnog računarstva. Od 2014. godine, zaposlen je u tvrtci Inchoo d.o.o. kao web developer te se bavi izradom web trgovina na Magento platformi.

Prilozi

Prilozi na CD-u:

- **Prilog 1.** Programski kod aplikacije
- **Prilog 2.** udovicic_stjepan_diplomski.pdf

Nakon objave rada, programski kod aplikacije te rad u izvornom i PDF formatu moguće je preuzeti na lokaciji:

- https://github.com/udovicic/WebGL-Test-suite