Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Simon Žagar

Vizualizacija žil tilnika z OpenGL-om

diplomsko delo

univerzitetni študijski program prve stopnje
računalništvo in informatika

Mentor: doc. dr. Matija Marolt

Ljubljana, 2012

Univerza *v Ljubljani*
Fakulteta *za računalništvo in informatiko*

Tržaška 25
1001 Ljubljana, Slovenija
telefon: 01 476 84 11
01 476 83 87
faks: 01 426 46 47
01 476 87 11
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 00035/2012
Datum: 12.04.2012

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **SIMON ŽAGAR**

Naslov: **VIZUALIZACIJA ŽIL TILNIKA Z OPENGL-OM**
VISUALISATION OF NECK VEINS WITH OPENGL

Vrsta naloge: Diplomsko delo univerzitetnega študija prve stopnje

Tematika naloge:

Napišite aplikacijo, ki omogoča prikaz modela žil tilnika s pomočjo OpenGLa. Aplikacija naj omogoča odpiranje datotek s podatki o modelu, prosto premikanje kamere, izračun normal, Phongovo osvetljevanje in senčenje ter prikaz na 3D zaslonih.

Mentor:

doc. dr. Matija Marolt

Fakulteta za računalništvo in informatiko
2
UNIVERZA V LJUBLJANI

Dekan:

prof. dr. Nikolaj Zimic

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Simon Žagar, z vpisno številko **63090355**, sem avtor diplomskega dela z naslovom:

Vizualizacija žil tilnika z OpenGL-om

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom doc. dr. Matije Marolta,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela in
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki "Dela FRI".

V Ljubljani, dne 18. septembra 2012 Podpis avtorja:

svojim staršem

Kazalo

Povzetek

Napisal sem aplikacijo, ki prikazuje model žil tilnika v OpenGL-u. Naredil sem uporabniški vmesnik za odpiranje datotek s podatki o modelu, ogled modela s poljubnega položaja in kota, s prikazom pomoči ali licenc in izbiro ločljivosti. Implementiral sem vrtenje modela z miško ter premikanje in vrtenje kamere s tipkovnico in miško. Napisal sem svojo kodo, ki bere podatke o modelu, izračuna odmik od središča in model samodejno postavi v primerno začetno pozo, računa normale ploskev ter jih na uporabnikov ukaz deli. Za učinkovitejše delovanje aplikacije sem uporabil koncept *buffer object*. Poleg tega sem implementiral Phongovo osvetljevanje in senčenje z uporabo senčilnikov.

Ključne besede:
OpenGL, transformacije, kvaternioni, osvetljevanje, senčenje, senčilniki, deljenje ploskev, vizualizacija podatkov

Abstract

I wrote an application that renders a model of neck veins using OpenGL. I created a user interface for opening files containing model data, viewing the model from an arbitrary position, showing help or licencing information and choosing resolution. I implemented model rotation with a mouse as well as camera rotation and translation with a keyboard or a mouse. I wrote a parser for reading files with model's data that also calculates the distance from koordinate origin and positions the model. It also calculates the normals of the model and applies a subdivision surface algorithm when instructed to. For more efficient rendering I used buffer objects. I implemented Phong lighting and shading using shaders.

Key words:
OpenGL, transformations, quaternions, lighting, shading, shaders, subdivision surface, data visualization

Poglavje 1

Uvod

Področje vizualizacije podatkov se ukvarja s povečanjem njihove berljivosti in estetike. Pogosto nek nov način prikaza podatkov človeku zelo hitro razkrije v njih zakopane zanimive informacije ali pomaga pri razumevanju nekega sistema. Primer je prikaz statističnih podatkov. Interakcija z navideznim okoljem omogoča človeku vajo in učenje brez nevarnosti ali porabe dragocenih virov. Primer je vizualizacija toka zraka preko aerodinamičnih teles. Omogoča tudi interakcijo s človeku sicer nedostopnim okoljem. Računalniška grafika dovoljuje nemoteno raziskovanje notranjosti človeškega telesa.
V znanosti, tehniki in medicini (angl. scientific visualization) ter v poslovnih, industrijskih in raznih drugih okoljih (angl. business visualization) se pogosto proizvede množične podatke. Te je koristno preučiti. Numerične simulacije lahko proizvedejo milijone vrednosti. Trende v podatkih se lahko poišče z metodami iz statistike in umetne inteligence, lahko pa postanejo očitni z vizualizacijo. Zbirka podatkov lahko vsebuje skalarje, vektorje, tenzorje itd. Podatki so lahko razporejeni po poljubno dimenzionalnem prostoru, barvno kodirani ali izkoriščajo posebno strojno opremo. Primer slednje je zaslon, ki je sposoben stereoskopskega prikaza. [14]

Cilj moje naloge je bil narediti aplikacijo, ki prikazuje model žil tilnika za njihovo obravnavo. Model se da obračati in si ga ogledati s poljubnega položaja. Prikaz omogoča stereo sliko, deljenje ploskev ter Phongovo osvetljevanje in senčenje. Aplikacija podpira odpiranje ustreznih datotek s podatki o žilah. V poglavju 2 opisu-

jem orodja in znanja, ki sem jih uporabljal za izdelavo aplikacije. Najprej opišem, kje sem ta orodja uporabil, nato pa sledi njihov individualen opis. V poglavju 3 pa povem več o tem, kako in kaj sem naredil v okviru te aplikacije.

Poglavje 2

Orodja in znanja, uporabljena za izdelavo aplikacije

V tem poglavju sledi opis orodij in znanj, ki sem jih uporabil v izdelavi aplikacije.

- Matematični koncept geometrijskih transformacij sem uporabil za premikanje in obračanje modela žil po prostoru.
- Kvaternione sem implementiral kot Javine objekte in razred, ki hranijo informacijo o orientacijah modela žil in kamere. Vsebujejo tudi metode za operacije nad kvaternioni in uporabo teh skupaj s transformacijskimi matrikami.
- Aplikacijo sem razvil s programskim jezikom Java v razvojnem okolju Eclipse.
- Za prikaz grafike sem uporabil API OpenGL.
  - Razumevanje delovanja transformacij v OpenGLu sem potreboval za pravilno uporabo geometrijskih transformacij.
  - S knjižnjico LWJGL sem dostopal do funkcij OpenGLa.
  - Seznanjenost s fiksno zmogljivostjo cevovoda mi je pomagala pri razumevanju in uporabi jezika OpenGL Shading Language, da sem lahko polepšal osvetljevanje in senčenje modela žil. Za to sem potreboval tudi znanje o slednjih dveh konceptih.

  - Za večjo hitrost delovanja aplikacije sem informacijo za izris modela žil poslal na stran strežnika implementacije OpenGL kot entitete buffer object.
  - Za lepši prikaz grafičnega vmesnika sem uporabil zlivanje.

- Za učinkovitejšo rabo aplikacije sem implementiral 2D uporabniški vmesnik. Za slednjega sem uporabil gumbe in sorodne elemente knjižnice TWL. Z uporabo ortogonalne projekcije v OpenGL-u sem naredil HUD, ki je namenjen premikanju in vrtenju kamere.

- Podatke za izris modela žil sem bral z datotek .obj.

- Vrtenje modela žil sem implementiral s pomočjo koncepta metanja žarkov.

2.1 Geometrijske transformacije

Geometrijske transformacije so postopki, izvršeni nad geometrijsko predstavljenimi predmeti. Spreminjajo jim velikost, položaj in usmeritev v prostoru.
Nekateri paketi za delo z računalniško grafiko razlikujejo med *geometric transformations* (geometrijske transformacije) in *modeling transformations*. Slednje se sklicujejo na grajenje prizorov in bolj zapletenih predmetov. Ti pa so hierarhično zgrajeni iz bolj enostavnih predmetov. Velikosti in odnosi teh predmetov v prostoru so določeni z geometrijskimi transformacijami. [14]

2.1.1 Osnovne transformacije

Transformacije lahko izvršimo na točkah v prostoru, ki določajo položaj, obliko, velikost in orientacijo nekega predmeta. Če želimo premakniti nek trikotnik, ki je v OpenGL-u določen s tremi točkami, izvršimo transformacijo na vseh treh točkah, preden se uporabijo za izris trikotnika.

Translacija je premik točke $P = (x, y, z)$ za odmik $\mathbf{t} = (t_x, t_y, t_z)$. Dobljena točka se nahaja na koordinatah $P' = (x + t_x, y + t_y, z + t_z)$. Če zapišemo točko P v homogenih koordinatah P_h, lahko translacijo izvršimo v obliki množenja s

translacijsko matriko T.

$$P'_h = TP_h = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x + t_x \\ y + t_y \\ z + t_z \\ 1 \end{bmatrix} \tag{2.1}$$

Tudi vrtenje, povečevanje in striženje lahko izvedemo v obliki množenja z matriko. Ker lahko zaporedje transformacij zapišemo kot produkt matrik, je zato koristno translacijo zapisati kot matrično množenje za enostavnejše sestavljanje transformacij v eno samo matriko. Nato lahko vse točke enega predmeta množimo z eno tako sestavljeno transformacijsko matriko in s tem prihranimo pri računanju. Seveda morajo biti obenem tudi rotacija, povečevanje in striženje predstavljeni z matrikami v homogenih koordinatah. V 3D prostoru to pomeni 4x4 matrike.

Vrtenje v tridimenzionalnem prostoru okrog koordinatne osi z lahko izrazimo s sledečimi enačbami 2.2:

$$\begin{aligned} x' &= x\cos\theta - y\sin\theta \\ y' &= x\sin\theta + y\cos\theta \\ z' &= z \end{aligned} \tag{2.2}$$

To v matričnem množenju s homogenimi koordinatami pomeni enačbo 2.3:

$$P' = R_z(\theta) \cdot P = \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{2.3}$$

Pri pozitivnem kotu θ bo vrtenje v obratni smeri urinega kazalca, če se naš pogled nahaja na pozitivnem delu z osi in je usmerjen proti koordinatnemu izhodišču. Za vrtenje okrog osi x priredimo enačbe 2.2 tako, da zamenjamo vsak x, y in z v ciklični permutaciji 2.4.

$$x \to y \to z \to x \tag{2.4}$$

Iz tako prirejenih enačb lahko zopet sestavimo matriko, ki da enak rezultat.

Iz takih in matrik za translacijo lahko sestavimo vrtenja okrog poljubnih osi.

2.2 Kvaternioni

2.2.1 Kvaternioni na splošno

Kvaternioni so razširitev kompleksnih števil. Prvič jih je opisal irski matematik William Rowan Hamilton leta 1843. Kvaternion je definiral kot količnik dveh usmerjenih linij v tridimenzionalnemu prostoru ali kot količnik dveh vektorjev. Lahko so predstavljeni tudi kot vsota skalarja in vektorja.

Kot množica so kvaternioni $\mathbb{H}$ enaki $\mathbb{R}^4$. $\mathbb{H}$ ima operacije seštevanja, množenja s skalarjem in kvaternionsko množenje. Vsota dveh elementov $\mathbb{H}$ je definirana kot vsota dveh elementov $\mathbb{R}^4$. Produkt elementa $\mathbb{H}$ z realnim številom je definiran tako, da je enak kot produkt v $\mathbb{R}^4$.

Za definicijo produkta dveh elementov $\mathbb{H}$ potrebujemo izbiro baze v $\mathbb{R}^4$. Običajno se elementi baze označujejo z 1, i, j in k. Vsak element $\mathbb{H}$ se da enolično zapisati kot linearno kombinacijo elementov baze. Z definicijo produktov baznih elementov lahko z distribucijo definiramo kvaternionsko množenje.

$$i^2 = j^2 = k^2 = ijk = -1 \tag{2.5}$$

Iz enačbe 2.5 lahko izpeljemo rezultate vseh kombinacij množenj po dva elementa baze.

x	**1**	**i**	**j**	**k**
1	1	i	j	k
i	i	-1	k	-j
j	j	-k	-1	i
k	k	j	-i	-1

Tabela 2.1: Množenje elementov baze

Elementi v prvem stolpcu tabele 2.2.1 so na levi strani množenja. Tako velja

na primer $ik = -j$ in $ki = j$. Sedaj lahko izračunamo hamiltonov produkt.

$$\begin{aligned}(a_1 + b_1 i + c_1 j + d_1 k)(a_2 + b_2 i + c_2 j + d_2 k) = \\ a_1 a_2 + a_1 b_2 i + a_1 c_2 j + a_1 d_2 k \\ + b_1 a_2 i + b_1 b_2 ii + b_1 c_2 ij + b_1 d_2 ik \\ + c_1 a_2 j + c_1 b_2 ji + c_1 c_2 jj + c_1 d_2 jk \\ + d_1 a_2 k + d_1 b_2 ki + d_1 c_2 kj + d_1 d_2 kk = \\ a_1 a_2 - b_1 b_2 - c_1 c_2 - d_1 d_2 \\ + (a_1 b_2 + b_1 a_2 + c_1 d_2 - d_1 c_2) i \\ + (a_1 c_2 - b_1 d_2 + c_1 a_2 + d_1 b_2) j \\ + (a_1 d_2 + b_1 c_2 - c_1 b_2 + d_1 a_2) k\end{aligned} \tag{2.6}$$

Konjugacija kvaterniona $q = a + bi + cj + dk$ je kvaternion $\overline{q} = a - bi - cj - dk$. Norma kvaterniona je

$$\|q\| = \sqrt{q\overline{q}} = \sqrt{\overline{q}q} = \sqrt{a^2 + b^2 + c^2 + d^2} \tag{2.7}$$

in je enaka kot evklidska norma v $\mathbb{H}$, gledanem kot $\mathbb{R}^4$.
Enotski kvaternion je kvaternion

$$U_q = \frac{q}{\|q\|} \tag{2.8}$$

z normo 1.
Obratna vrednost kvaterniona q je kvaternion:

$$q^{-1} = \frac{\bar{q}}{\|q\|^2} \tag{2.9}$$

Kvaternione se da predstaviti tudi kot četverico, skalarni z vektorskim delom, matriko 2x2 ali matriko 4x4. [5] [4]

2.2.2 Kvaternioni v računalniški grafiki

V računalniški grafiki se kvaternioni uporabljajo za predstavitev rotacij zaradi sledečih prednosti:

- vsak kvaternion lahko predstavimo s samo štirimi vrednostmi, predstavitev z rotacijsko matriko jih potrebuje devet;

- zaradi numerične nestabilnosti lahko matrike izgubijo na ortogonalnosti. To se lahko zgodi tudi pri kvaternionih, a če kvaternion preprosto normaliziramo, je problem že rešen. Popravljanje matrik je manj enostavno;
- sestavljanje dveh vrtenj je množenje dveh rotacijskih matrik ali množenje dveh kvaternionov. Slednje je hitrejše;
- če želimo interpolirati dve vrtenji, predstavljeni z eulerjevimi koti, in se dve osi, okoli katerih vrtimo, poravnata, izgubimo stopnjo prostosti. Temu rečemo kardanska zapora, ki pri animaciji lahko povzroča nezaželeno vedenje. Primer kardanske zapore dobimo, če postavimo vrtenje z eulerjevimi koti najprej okoli osi x, nato okoli osi y in potem okoli osi z. Zatem pa zavrtimo za 90° okoli osi y. Tako se bo os z poravnala z osjo x. Pri interpolaciji vrtenj, predstavljenih s kvaternioni, sploh ne pride do kardanske zapore.

Kvaternion z normo 1 vedno predstavlja vrtenje ali orientacijo v prostoru. Vrtenje okrog enotskega vektorja $\mathbf{u}$ za kot θ lahko predstavimo s kvaternionom:

$$q = (\cos\frac{\theta}{2}, \mathbf{u}\sin\frac{\theta}{2}) \tag{2.10}$$

Točko $\mathbf{p}$ v trirazsežnem prostoru, ki jo želimo zavrteti, zapišemo kot kvaternion $\mathbf{P} = (0, \mathbf{p})$. Vrtenje izvršimo z naslednjo operacijo:

$$\mathbf{P}' = q\mathbf{P}q^{-1} \tag{2.11}$$

Točka po vrtenju $\mathbf{p}'$ se nahaja znotraj dobljenega kvaterniona $\mathbf{P}' = (0, \mathbf{p}')$. Kvaternion $q = (s, a, b, c)$ lahko prevedemo v matrično obliko $\mathbf{M_r}$:

$$\mathbf{M_r} = \begin{bmatrix} 1-2b^2-2c^2 & 2ab-2sc & 2ac+2sb \\ 2ab+2sc & 1-2a^2-2c^2 & 2bc-2sa \\ 2ac-2sb & 2bc+2sa & 1-2a^2-2b^2 \end{bmatrix} \tag{2.12}$$

[14]

2.3 Java in Eclipse

Praktični del sem naredil v programskem jeziku Java v razvojnem okolju Eclipse. Java je objektno orientiran visokonivojski programski jezik s sintakso, podobno

jeziku C++. Za Javo velja, da se njena koda ne prevede v strojno, ampak v kodo *bytecode*, ki teče na virtualnem stroju *Java Virtual Machine* (JVM). To omogoča večjo prenosljivost Javinih programov. JVM sam skrbi za brisanje podatkov, na katere ne kaže nobena referenca v programu, ki se zaganja. [3] Eclipse je razvojno okolje, ki je večinoma napisano v Javi, ki temelji na vtičnikih in lahko tako podpira več programskih jezikov. Ponuja prevajalnik, razhroščevalnik, opozarjanje na sintaktične napake in mnoga druga orodja. [2]

2.4 OpenGL (Open Graphics Library)

Standardna specifikacija OpenGL je zasnovana kot API (Application Programming Interface), vmesnik za dostopanje do grafičnih funkcij za različne platforme in programske jezike. Funkcije OpenGL-a se začnejo z "gl". Ostale besede funkcije se začnejo z veliko začetnico. Primer so funkcije **glBegin**, **glMatrixMode** ali **glClear**. Parametri funkcij so pogosto cela števila, ki dajejo navodila različnim funkcijam. Da ni potrebno programerjem vedeti, kaj katero število naredi, OpenGL poimenuje razna cela števila s konstantami. Konstante so zapisane z velikimi črkami in se začnejo z "GL". Primeri simboličnih konstant so **GL_QUADS**, **GL_MODELVIEW**, **GL_COLOR_BUFFER_BIT**.
Vsaka OpenGL implementacija vključuje tudi OpenGL Utility (GLU), ki priskrbi razne funkcije. Primer teh je pripravljanje projekcijskih matrik. Funkcije GLU-ja se začnejo z "glu". Za stvaritev grafičnih elementov je potrebno najprej ustvariti okno na zaslonu. Upravljanje oken temelji na platformi, OpenGL pa mora biti od nje neodvisen. Zato obstajajo knjižnice za upravljanje z okni, ki podpirajo OpenGL. Applovi sistemi lahko uporabljajo AppleGL (AGL), ki se začnejo z "agl" oziroma Core OpenGL (CGL). Windows uporablja WGL, katerega funkcije se začnejo z "wgl". Linuxu pa ustreza 'OpenGL Extension to the X Window System' (GLX), katerega rutine se začnejo z "glx". OpenGL Utility Toolkit (GLUT) je knjižnica funkcij za interakcijo s kakršnimkoli sistemom oken. Njene funkcije se začnejo z "glut". [14]

OpenGL API se osredotoča na risanje in včasih branje grafičnih elementov v spomin *frame buffer*. (*Buffer* je del spomina, namenjen določenim podatkom.)

Izvajanje programov poteka na način odjemalec-strežnik (angl. *client-server*). Program aplikacije je odjemalec in pošilja OpenGL-ove ukaze. Te interpretira in izvrši implementacija OpenGL-a, ki je strežnik. Odjemalec in strežnik sta lahko na različnih računalnikih. Ukazi se vedno izvajajo v vrstnem redu, v katerem jih dobi strežnik. Procesu, ko strežnik pretvori prejete podatke v nekaj, kar se prikaže na zaslonu, rečemo *rendering* (od tu dalje upodabljanje). Ta process je pospešen s strojno opremo. Del teh ukazov se izvaja na CPU-ju (Central Processing Unit ali slovensko centralno procesna enota). Strojni opremi, ki je posvečena risanju in vzdrževanju vsebine zaslona, rečemo *graphics accelerator* ali slovensko grafični pospeševalnik. Primer je grafični pospeševalnik, ki sem ga uporabljal sam, grafična kartica ATI Mobility Radeon HD 4570. Vsak viden element slike zaslona (ali piksel, angl. pixel) je predstavljen z enim ali več bajti spomina na grafičnem pospeševalniku. Ti skupaj tvorijo spomin *display memory*, ki se osveži večkrat na sekundo. Drugi del spomina grafičnih pospeševalnikov je spomin *offscreen memory*, ki se ne prikazuje na zaslonu. Za dodeljevanje delov prostora skrbijo sistemi oken in pri vsaki implementaciji so za to in sorodne stvari na voljo funkcije (WGL, AGL in GLX). Območje spomina, ki se spreminja kot posledica upodabljanja, je *frame buffer*. V sistemu oken *frame buffer* ustreza oknu, brez oken pa celotnemu prikazu. En *frame buffer* je sestavljen iz več *bufferjev*:

- *color buffer* (teh je lahko več) hrani barve;
- *depth buffer* hrani globine (oziroma oddaljenosti od pogleda). Po izrisu vsakega predmeta se njegove razdalje od kamere/pogleda za vsak piksel zapišejo v *depth buffer*. Tako se lahko za vsak piksel pri izrisu novega predmeta preveri ali ga prej narisani prekriva;
- *stencil buffer* hrani vrednosti, ki se uporabljajo kot šablone. Z izrisom predmeta lahko shranimo zapletene oblike kot vrednosti v *stencil bufferju*. Tako določimo, da se (ne) piše v *color buffer* le za piksle, ki ustrezajo dodeljenim vrednostim v *stencil bufferju*;
- *accumulation buffer* ima večjo natančnost shranjevanja barv kot *color buffer* in tako omogoča shranjevanje več slik. Tako bi lahko v *accumulation buffer* vnesli več vrednosti različnih slik. Te bi delili s številom slik in dobili effekte,

kot sta zamegljenost zunaj fokusa leče ali zamazanost barv zaradi hitro premikajočih se predmetov;

- *multisample buffer* shranjuje po več vzorcev barv in jih pri izrisu združi, da doseže *anti-aliasing* čez celotno sliko;

- *pomožni buffer* (tudi teh je lahko več) hrani poljubne podatke;

Večina strojne opreme podpira *double buffering*. En *buffer* je v ozadju in se vanj piše, drugi pa se prikazuje. Ko je upodabljanje končano, se zamenjata. *OpenGL state* je stanje OpenGL-a kot avtomata (angl. *state machine*). Shranjeno je v podatkovno strukturo *graphics context*. Slednje je ustvarjeno ali uničeno s funkcijami, vezanimi na sistem oken. Take funkcije določijo tudi to, na katere grafične kontekste in *frame bufferje* se nanašajo ukazi OpenGL-a, ki nato sledijo. [16]

2.4.1 Transformacije v OpenGL-u

OpenGL nudi funkcije za osnovne transformacije. Za translacijo podamo odmike po koordinatnih oseh ukazu **glTranslate*(tx, ty, tz)**. Za rotacijo podamo kot in izbiro osi ukazu **glRotate*(theta, vx, vy, vz)**. Za skaliranje podamo faktorje po vsaki osi ukazu **glScale*(sx, sy, sz)**. Vsaka transformacija se prevede v 4x4 matriko in se uporabi na vseh točkah, podanih po njenem klicu. Za dostop do matrike, ki se uporablja za transformiranje predmetov, uporabimo ukaz **glMatrixMode(GL_MODELVIEW)**. Tako je izbrana matrika *modelview matrix* z desne pomnožena z matrikami, ki predstavljajo zgoraj omenjene transformacije. Enotsko matriko naložimo z ukazom **glLoadIdentity()**. Poljubno matriko naložimo z ukazom **glLoadMatrix*(matrika4x4)**. V OpenGL-u morajo biti elementi podani po stolpcih. Po vrsti so najprej podani elementi prvega stolpca, nato drugega itd. Če želimo trenutno matriko pomnožiti s poljubno matriko, uporabimo **glMultMatrix*(matrika4x4)**. Iz zgoraj napisanega sledi, da je nazadnje podana transformacija izvršena prva. OpenGL za vsako vrsto matrike (*modelview*, *projection*, *texture*, *color*) shranjuje sklad matrik. Na začetku ima vsak sklad le enotsko matriko. Zgornja matrika na skladu je trenutno uporabljena (angl. *current matrix*). OpenGL podpira shranjevanje vsaj 32-ih matrik na skladu za *modelview mode* in vsaj 2 matriki za ostale načine. Trenutno matriko skopiramo in dodamo

na vrh sklada s klicem **glPushMatrix()**. Zgornjo matriko lahko uničimo s klicem **glPopMatrix()**. [14]

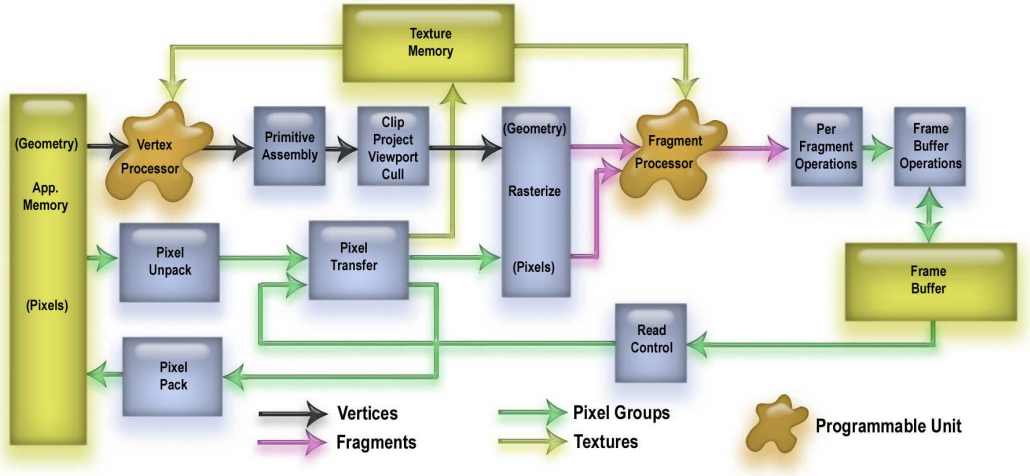
2.4.2 LWJGL

LWJGL (Lightweight Java Game Library) je knjižnica, ki daje programerju Jave dostop do funkcij OpenGL-a, OpenCL-a (Open Computing Language), OpenAL-a (Open Audio Library) ter do krmilnikov, kot so igralni ploščki, volani in palice. Knjižnjica LWJGL je bila zasnovana s pričakovanjem, da se bo razvila v temelje kakšnega programerju/umetniku bolj prijaznega igralnega pogona (angl. *game engine*). [6] Metode, ki so zgolj namenjene hitrejšemu programiranju v C-ju (na primer **glColor3fv**) za Javo niso implementirane. Knjižnica javi izjemo (angl. *exception*), če na Windowsu strojno pospeševanje ni na voljo. Ker so snovalci LWJGL-a želeli, da bi knjižnica nekoč tekla na manjših napravah, je njena binarna verzija manjša od enega megabajta. Redkeje uporabljene funkcije niso implementirali. (LWJGL podpira le eno okno in delovanje v oknu sploh ni nujno podprto.) Še več funkcij so avtorji izpustili, ker so bili mnenja, da jih programerji video iger ne bodo potrebovali. Risanje z le enim *bufferjem* ni podprto. (Uporablja se *Double buffering.*) Funkcijam programer zaradi hitrosti podaja *bufferje* in podajanje polj ni na voljo. Nekatere Javine izjeme dedujejo od razreda *RuntimeException.* To pomeni, da so vrste *unchecked exception.* To so izjeme, po katerih se od programa ne pričakuje, da bo okreval, ampak naj do takih izjem sploh ne bi prišlo. Izjeme so tako narejene tudi zato, ker je sicer njihovo lovljenje (angl. *catch*) povzročilo nepreglednost kode. [7]

2.4.3 Fiksna zmogljivost cevovoda (angl. Fixed Functionality Pipeline)

Ker se morajo operacije OpenGL-a izvesti v omejenem vrstnem redu, ga lahko vidimo kot *graphics programming pipeline* ali po slovensko grafični cevovod. Na sliki 2.1 je prikazano okvirno delovanje OpenGL-a po specifikaciji 2.0. OpenGL ima približno tako delovanje že od samega začetka. Od avgusta 2012 je zunaj že specifikacija OpenGL 4.3, ki grafičnim karticam narekuje še večjo zmogljivost. Vsaka implemetacija OpenGL-a mora dati rezultate, ki so v skladu z njegovo specifikacijo.

Seveda ima lahko implementacija OpenGLa povsem drugačen diagram delovanja, ampak glede na rezultate mora izgledati, kot da sledi specifikaciji. [16]



Slika 2.1: OpenGL cevovod [9], sorodna slika se nahaja tudi v [16].

Sledi opis delov cevovoda za upodabljanje trirazsežnih predmetov. Ostali deli cevovoda na sliki 2.1 se uporabljajo za procesiranje slikovnih podatkov.

- (Geometry) Podatki o geometriji pridejo z aplikacije oziroma se lahko nahajajo že na grafični kartici pred izrisom. Aplikacija povzroči začetek risanja.
- (Vertex Processor) Nato se točke v prostoru transformirajo v skladu z matrikama *modelview* in projekcijsko matriko. Od tu naprej bom uporabljal izraz ogljišča, da ločim *vertex* (ogljišče) od *point* (točka). Normale se transformirajo z inverzno in transponirano matriko *modelview*. Tu se izračuna tudi osvetljevanje (angl. lighting), zato se tej fazi reče tudi *transformation and lighting* ali *T&L*.
- (Primitive Assembly) Tu so skupine ogljišč zvezane v posamezne primitive. Točka (angl. *point*) je sestavljena iz enega ogljišča (angl. *vertex*), črta iz dveh ogljišč, trikotnik iz treh itd. Tako je zato, ker so operacije naslednje faze odvisne od vrste primitivov, ki se jih dobi.
- (Clip, Project, Viewport, Cull) Rezanje (angl. *clipping*) se znebi vseh primitivov, ki so zunaj pogleda in odščipnih delov, ki štrlijo izven njega. Sledi

projekcija (angl. projection). Če je perspektivna, se po njej x, y in z koordinate delijo s homogeno koordinato w. Nato pa se ogljišča transformirajo v koordinate okna, določenega z **glViewport**. Izločijo (angl. *culling*) se poligoni, vidni le z ene napačne strani, ki niso ustrezno orientirani.

- (Rasterization) Točke, črte in poligoni so tu razgrajeni na enote tako, da vsaka ustreza enemu pikslu v ciljnem *frame bufferju*. Tem enotam rečemo fragmenti (angl. *fragments*). Fragment je sestavljen iz koordinat v oknu, globine, barve, koordinate v teksturi itd. Slednje vrednosti so določene z interpolacijo ustreznih vrednosti pri ogljiščih primitiva. Glede na nastavitev z ukazom **glShadeModel** so barve fragmentov lahko interpolirane (parameter **GL_SMOOTH**) ali pa enake zadnjemu ogljišču (**GL_FLAT**).

- (Fragment Processor) Ena operacija te faze je *texture mapping* ali lepljenje teksture. Tu so koordinate fragmenta uporabljene za dostop do dela grafičnega spomina, ki se mu reče *texture memory*. Dostop in to, kako so pridobljene vrednosti uporabljene na fragmentu, se določi med mnogimi izbirami, ki jih omogoča OpenGL.

- (Per Fragment Operations) Te operacije vključujejo razne teste, ki preverjajo, ali naj se barva fragmenta uporabi: *Pixel ownership test*, *scissor test*, *alpha test*, *stencil test*, *depth test*. Sem spadajo tudi sledeče operacije: zlivanje (angl. *blending*), *dithering*, *logical operations* ali slovensko logične operacije.

- (Frame Buffer Operations) Sem spada vse, kar vpliva na celoten *frame buffer*. OpenGL podpira prikaz stereo slik in *double buffering*. Za cilj upodabljanja (*rendering*) je zato veliko izbire. Območje *frame bufferja* je *buffer* in jih imamo več: prednji (angl. front), zadnji (angl. back), levi (angl. left), desni (angl. right), prednji levi, prednji desni, zadnji levi, zadnji desni, prednji in zadnji, aux0, aux1 itd. Za cilj upodabljanja izberemo *buffer* z ukazom **glDrawBuffer** in več *bufferjev* z ukazom **glDrawBuffers**. Ali se sme pisati znotraj regij ciljnih *bufferjev*, se določa z ukazi **glColorMask** (ali se sme pisati v rdeče, zelene, modre ali alfa komponente), **glDepthMask** (ali se sme pisati v komponento za globino), **glStencilMask** (v katere bite *stencil* komponente se sme pisati). Vrednosti v *frame bufferju* se inicializira

z ukazom **glClear**. Vrednosti, ki naj se pri tem uporabijo, se poda z ukazi **glClearColor**, **glClearDepth**, **glClearStencil** in **glClearAccum**.

2.4.4 OpenGL Shading Language (GLSL)

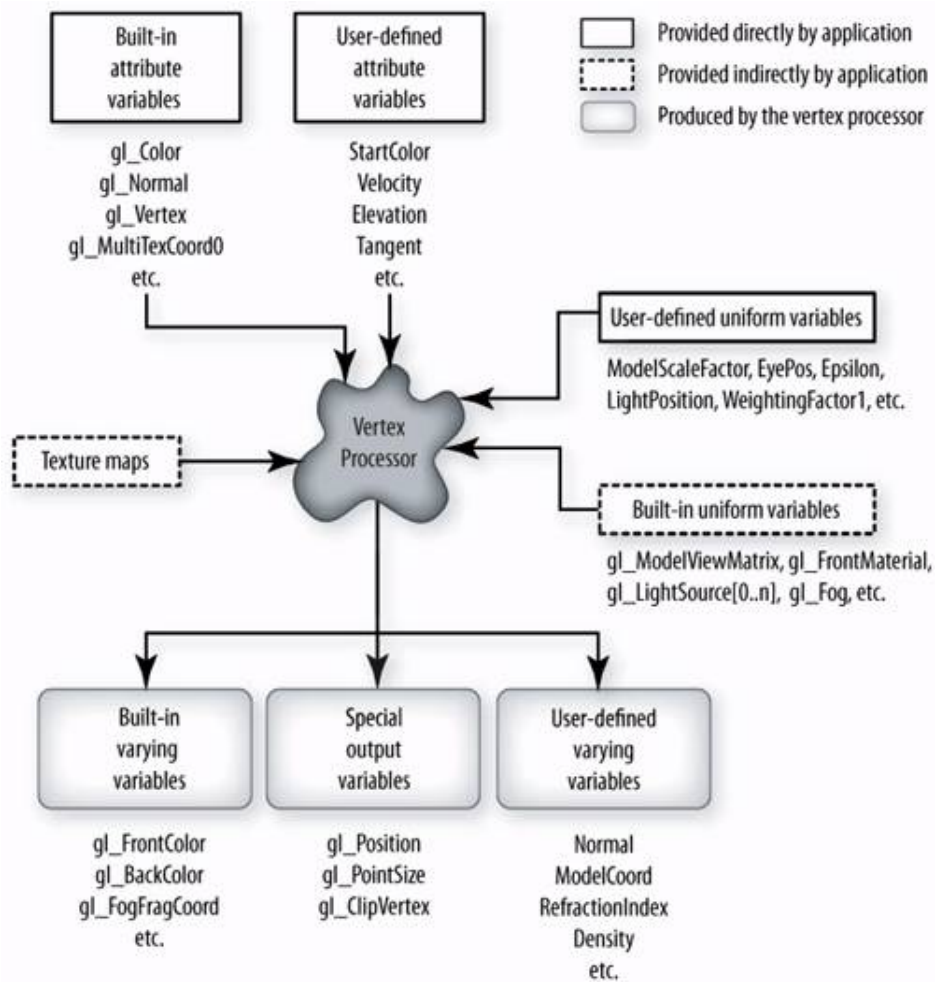
OpenGL Shading Language je jezik za programiranje določenih delov grafičnega cevovoda. Pred verzijo 2.0 OpenGL-a programabilnost grafične strojne opreme grobo rečeno ni bila izpostavljena. Zaradi optimizacijskih razlogov naj ne bi operacije znotraj strojne opreme potekale poljubno, ampak so jih proizvajalci določili po lastni presoji za podporo grafičnih funkcij, katere naj bi njihov izdelek podpiral. Specifikacija OpenGL 2.0 pa je prisilila proizvajalce k podpiranju programabilnosti delov cevovoda in sicer delov *transformation and lighting* ter *fragment Processing*. OpenGL 4.3 podpira tudi *Tesselation Control Shader*, *Tesselation Evaluation Shader* in *Compute Shader*. Torej je programabilnih še več delov cevovoda. Ampak v tej nalogi se omejujem na verzijo OpenGL-a 2.0.

Kodi v jeziku OpenGL Shading Language, ki določa dogajanje v eni od teh dveh faz cevovoda, rečemo *shader* ali senčilnik. Imamo *vertex shader* za senčilnike transformacij ogljišč in osvetljevanje ter *fragment shader* za processiranje fragmentov. Z besedno zvezo *OpenGL shader* poudarimo, da je senčilnik napisan v OpenGL-ovem jeziku za senčenje in ne v katerem drugem. OpenGL prispeva mehanizem za prevajanje in povezovanje kode v program. Jezik je podoben C-ju. Podpira matrične in vektorske operacije. Na shemi 2.1 se vidi, katera dva dela cevovoda lahko programiramo.

Senčilnik ogljišč Tak senčilnik zamenja naslednje operacije:

- transformacije ogljišč,
- transformiranje in normaliziranje normal,
- tvorjenje teksturnih koordinat,
- osvetljevanje,
- uporaba barvnega materiala.

Senčilnik, ki zamenja eno od teh operacij, mora poskrbeti tudi za ostale, sicer se ne bodo izvedle. Spodaj je slika 2.2, ki prikazuje vhode in izhode senčilnika za ogljišča. Senčilnik poda algoritem, ki se izvede na vhodnih podatkih in proizvede izhodne.



Slika 2.2: Vhod in izhod za procesor senčilnika za ogljišča [10], sorodna slika se nahaja tudi v [16].

Spremenljivke vrste *attribute variables* predstavljajo vrednosti, ki jih aplikacija podaja senčilniku ogljišč in niso na voljo senčilniku fragmentov. Vrednosti teh spremenljivk so lahko podane znotraj ukazov **glBegin** ali **glEnd** ali znotraj polj in so lahko tako podane za vsako ogljišče. Te spremenljivke se delijo na že vgrajene in na take, ki jih definira uporabnik. Do že vgrajenih lahko dostopamo z določenimi imeni spremenljivk: **gl_Color**, **gl_Normal**, **gl_Vertex**.

Spremenljivke vrste *uniform variables* predstavljajo vrednosti, ki se redkeje spreminjajo. Uporabljajo se lahko za oba senčilnika. Ne moremo jim dajati vred-

nosti znotraj ukazov **glBegin** ali **glEnd**. Tudi te lahko definira programer ali pa uporablja že vgrajene, ki imajo rezervirano predpono "gl_". Z njimi lahko senčilnik dostopa do OpenGL-ovega grafičnega konteksta. Z ukazom **glGetUniformLocation** pridobimo lokacijo uniformnih spremenljivk, ki jih je definiral programer senčilnika. V to lokacijo lahko shranimo vrednost z ukazom **glUniform**.

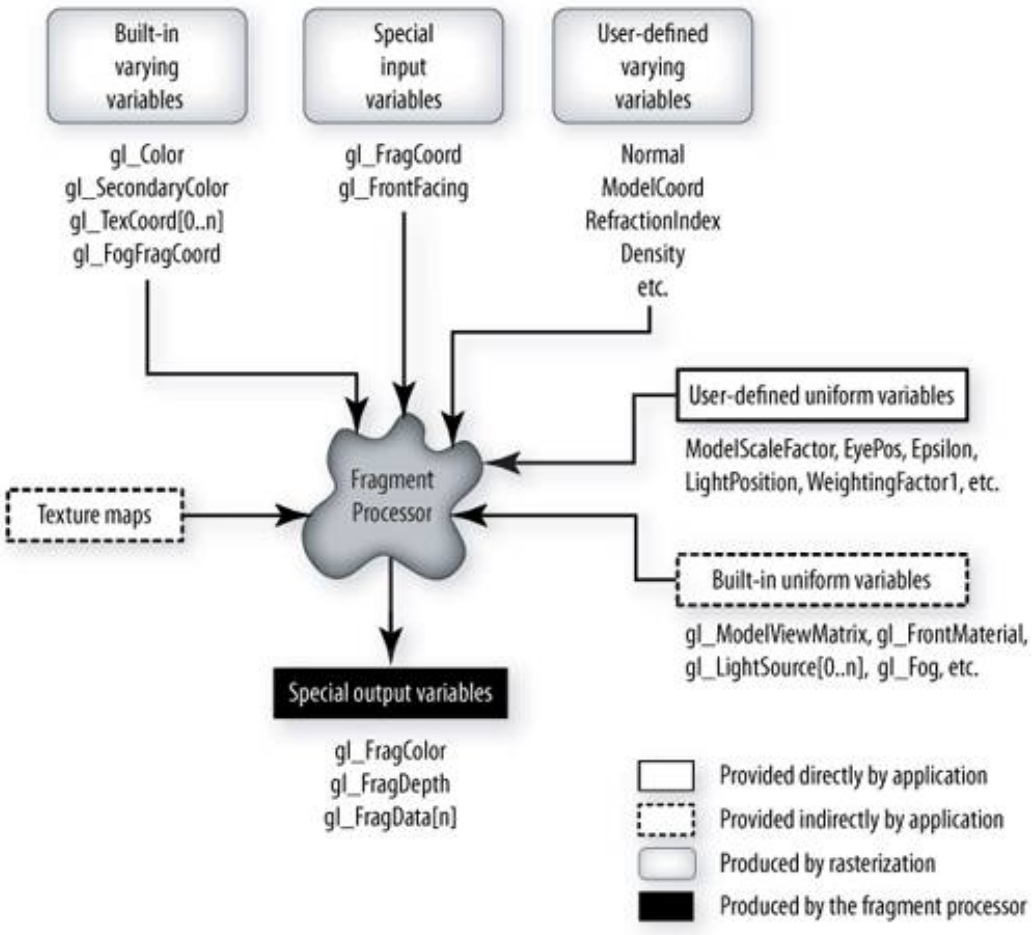
Spremenljivke vrste **varying variables** so podane naprej senčilnikom fragmentov. Za vsak fragment so interpolirane med ogljišči. Primer spremenljivk, ki jih je lahko koristno interpolirati, so normale, barve, koordinate tekstur itd. Procesor ogljišč podatke pošlje naprej po cevovodu.

Senčilnik ogljišč se zažene enkrat na vsako podano ogljišče. Implementacija OpenGL-a lahko proces paralelizira. Senčilnik ogljišč izračunane podatke poda naprej po posebnih spremenljivkah. Lokacija v prostoru se shrani v **gl_Position**. Po rasterizaciji pride na vrsto senčilnik fragmentov.

Senčilnik fragmentov Tak senčilnik zamenja naslednje operacije:

- operacije na interpoliranih vrednostih,
- dostopanje do tekstur,
- uporaba tekstur,
- megla,
- seštevek barv.

Senčilnik, ki zamenja eno od teh operacij, mora poskrbeti tudi za ostale, sicer se ne bodo izvedle. Spodaj je slika 2.3, ki prikazuje vhode in izhode senčilnika za fragmente. Senčilnik poda algoritem, ki se izvede na vhodnih podatkih in proizvede izhodne.



Slika 2.3: Vhod in izhod za procesor senčilnika za fragmente [10], sorodna slika se nahaja tudi v [16].

Senčilnik fragmentov ne more spreminjati x ali y lokacije fragmentov. Senčilnik fragmentov se zažene za vsak fragment posebej in nima dostopa do ostalih. To omogoča implementaciji OpenGL-a paralelizirati izvajanje senčilnikov nad fragmenti. Če so slikovni podatki naloženi v spomin za teksture, lahko za vsak piksel izvajamo operacije, ki zahtevajo dostop do sosednjih. Glavni vhod senčilnika fragmentov so interpolirane spremenljivke **varying variables** kot rezultat rasterizacije. Spremenljivke **varying variables**, ki jih definira programer, se morajo ujemati v senčilniku ogljišč in senčilniku fragmentov. Dostopne so razne spremenljivke, ki so rezultat fiksne zmogljivosti. Primer sta spremenljivki **gl_fragCoord**

in **gl_FrontFacing**. Prav tako je preko vgrajenih uniformnih spremenljivk dostopen grafični kontekst. Procesor fragmentov lahko večkrat dostopa do tekstur in uporabi te vrednosti, kakor želi. Tudi senčilnik fragmentov poda izračunano vrednost za fragment s spreminjanjem posebnih spremenljivk: **gl_FragColor**, **gl_FragDepth** in **gl_FragData**.

Združeni senčilniki *Unified Shader Model* uporablja konsistenčen nabor ukazov preko vseh senčilnikov. Kljub temu so še vedno majhne razlike med senčilnikom ogljišč in fragmentov. Ko grafična strojna oprema podpira *Unified Shader Model*, je smiselno uporabiti *Unified Shading Architecture*. To je arhitektura, pri kateri so enote za eno vrsto senčilnikov lahko uporabljene pri drugih vrstah senčilnikov. Tako ni potrebno uravnovesiti dela za procesorje fragmentov in procesorje ogljišč ter je lažje v večji meri izkoristiti zmogljivost strojne opreme. [13]

2.4.5 Buffer object

Če ogljišča podamo takoj pred izrisom, rečemo temu *immediate mode*. Lahko jih podamo posamezno z ukazom **glVertex** znotraj ukazov **glBegin** in **glEnd**. Z ukazom **glBegin** povemo ali rišemo trikotnike, štirikotnike, pahljače itd. Lastnosti ogljišč podamo z ukazom **glColor** za barvo, **glNormal** za normalo, **glTexCoord** za koordinato na teksturi itd. Da se izognemo klicanju funkcij za vsako ogljišče, lahko uporabimo polja ogljišč (angl. *vertex arrays*). Tako pošljemo po več ogljišč in njihovih lastnosti hkrati. Za izris vseh pošljemo le en ukaz. Primer je ukaz **glDrawElements**. Implementacija OpenGL-a lahko podatke, organizirane v polja, uporabi z višjo hitrostjo. Barve lahko podamo z **glColorPointer**, normale z **glNormalPointer** in ogljišča z **glVertexPointer**.

Ukaze lahko shranimo za poznejše izvajanje v podatkovni strukturi *display list*, ki je shranjena na strani strežnika. Vsi ukazi med ukazoma **glNewList** in **glEndList** so shranjeni in jih lahko pokličemo z **glCallList**. Nekateri ukazi OpenGL-a niso dovoljeni za shranjevanje v *display list*. Implementacija OpenGL-a lahko tako shranjene ukaze optimizira. Dodatna prednost je, da se lahko *display liste* shranjuje na grafičnem pospeševalniku. Ta način shranjevanja ukazov je uporaben, če jih želimo večkrat izvršiti.

Namesto shranjevanja ukazov lahko na stran strežnika pošljemo le podatke o

ogljiščih. Tudi tako so lahko potrebni podatki shranjeni na grafičnem pospeševalniku in ne obremenjujejo vhodno-izhodnega sistema računalnika ob vsaki njihovi uporabi. Ukaz **glBindBuffer** ustvari ali pa uporabi nek *buffer object*. Veže ga na neko tarčo. Primer podajam s konstanto **GL_ARRAY_BUFFER**. Vsak *buffer object* ima svoje ime, ki je pozitivno celo število, razen 0. Če vežemo neko tarčo na ime 0, ukazi, ki kažejo na ustrezne podatke (**glVertexPointer**, **glNormalPointer**, **glColorPointer** itd.), pokažejo na podatkovne strukture na strani odjemalca. Če uporabimo ime za nek *buffer object*, pa enaki ukazi kažejo na odmik znotraj trenutno vezanih *buffer object*ov. [16] [8]

2.4.6 Zlivanje (angl. Blending)

Večina grafičnih paketov vsebuje metode za razne učinke mešanja barv (angl. *color-blending functions* ali *image-compositing functions*). V OpenGL-u se dve barvi zlije tako, da se najprej naloži prvi predmet v *frame buffer* in se ga nato združi z barvo drugega predmeta v *frame bufferju*. Trenutni barvi v *frame bufferju* se reče *Opengl destination color*. Barvi drugega predmeta se reče *OpenGL source color*. Zlivanja se lahko izvaja samo v RGB ali RGBA načinu. Če zlivanje prej ne aktiviramo s klicem **glEnable(GL_BLEND)** bo nov predmet v ustreznem mestu v *frame bufferju* preprosto zamenjal vrednost. Za določanje načina zlivanja sta na voljo dve množici faktorjev zlivanja (angl. *blending factors*). Ena množica je za trenutne vrednosti v *frame bufferju*, druga pa za barve, ki jih bomo dodali. Nova barva je izračunana po sledeči formuli:

$$(S_r R_s + D_r R_d, \quad S_g G_s + D_g G_d, \quad S_b B_s + D_b B_d, \quad S_a A_s + D_a A_d). \tag{2.13}$$

RGBA barvne komponente vira (angl. *source*) so tu (R_s, G_s, B_s, A_s), barvne komponente za ponor (angl. *destination*) pa (R_d, G_d, B_d, A_d). Faktorji zlivanja vira so (S_r, S_g, S_b, S_a), ponora pa (D_r, D_g, D_b, D_a). Faktorje zlivanja določimo s klicem funkcije **glBlendFunc(sFactor, dFactor)**. Parametroma *sFactor* in *dFactor* so določene OpenGL konstante. Konstanta **GL_ZERO** da faktorje zlivanja (0, 0, 0, 0), **GL_ONE** pa (1, 1, 1, 1). Faktorje zlivanja lahko postavimo na alfa vrednost vira ali ponora s konstanto **GL_SRC_ALPHA** ali **GL_DST_ALPHA**. Če želimo faktorje zlivanja ena minus alfa ponora ali ena minus alfa vira, uporabimo

GL_ONE_MINUS_SRC_ALPHA ali **GL_ONE_MINUS_DST_ALPHA**. Obstajajo še druge konstante. [14]

2.4.7 Osvetljevanje

Izgled predmetov glede na svetlobo, ki pada nanje in način, kako predmeti to svetlobo odbijajo, približno izračunamo z modeli osvetljevanja. Simuliranje obnašanja svetlobe kot v realnem svetu bi bilo za računalnik veliko preveč računsko intenzivno, sploh v realnem času. Modeli osvetljevanja izrabijo razne informacije za izračun barve predmetov. Te informacije so lahko: kako so predmeti obrnjeni glede na vire svetlobe, kje se nahaja opazovalec, kakšne barve sevajo viri svetlobe in kakšne barve seva ali odbija predmet itd. Nekateri modeli osvetljevanja so manj računsko zahtevni, drugi dajo lepši rezultat, eni potrebujejo manj informacij itd. OpenGL privzeto podpira Blinnov-Phongov model osvetljevanja. Ta loči računanje barve na različne dele in sešteje njihove prispevke.

$$\begin{aligned} Barva_{končna} &= Material_{ambient} * Scena_{ambient} \\ +Material_{difuzniDel} &* Svetloba_{difuzniDel} * Intenziteta_{difuzniDel} \\ +Material_{odblesk} &* Svetloba_{odblesk} * Intenziteta_{odblesk} \end{aligned} \tag{2.14}$$

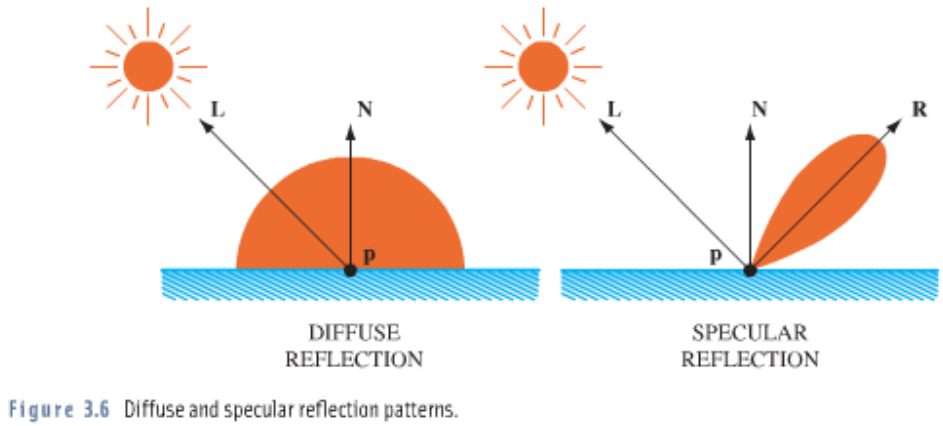


Figure 3.6 Diffuse and specular reflection patterns.

Slika 2.4: Odboj svetlobe, levo difuzni in desno odblesk [15].

Seveda je lahko virov svetlobe več in tudi površina lahko oddaja lastno svetlobo. Tega v zgornji enačbi 2.14 nisem upošteval.

ambient Površina je zelo redko brez kakršnekoli osvetlitve. Do nje skoraj vedno pride nekaj svetlobe, ki se je tako ali drugače odbijala do tja. A te majhne količine razpršene svetlobe se ne splača računati. Zato preprosto podamo majhno osvetlitev vsakemu delu površine. $Material_{ambient}$ je barva površine zavoljo tega dela osvetljevanja in $Scena_{ambient}$ je svetloba okolja.

difuzniDel Ta del osvetljevanja upošteva, pod katerim kotom pade svetloba na material, a odseva to svetlobo v vse smeri enako. Zato je ta osvetlitev neodvisna od kota, pod katerim površino gledamo. $Material_{difuzniDel}$ je barva površine. $Svetloba_{difuzniDel}$ je barva svetlobe enega vira. Naj bo $\mathbf{L}$ enotski vektor, ki kaže od točke na površini proti viru svetlobe. $Intenziteta_{difuzniDel}$ je enaka skalarnemu produktu normale in vektorja $\mathbf{L}$. Če je skalarni produkt negativen, se uporabi vrednost 0.

odblesk Ta del osvetlitve je odvisen od kota, pod katerim gledamo površino. Barva površine je $Material_{odblesk}$. $Svetloba_{odblesk}$ je barva svetlobe. Naj bo $\mathbf{V}$ enotski vektor, ki kaže od točke na površini proti lokaciji opazovalca. Naj bo $\mathbf{R_l}$ enotski vektor, ki kaže v smeri odboja svetlobnega žarka od površine. $Intenziteta_{odblesk}$ se po Phongovemu modelu osvetljevanja izračuna kot skalarni produkt med vektorjem $\mathbf{V}$ in vektorjem $\mathbf{R_l}$ na neko potenco n, ki odraža, kako gladka je površina. Ker je $\mathbf{R_l}$ malenkost bolj računsko zahtevno izračunati, se uporablja Blinnov-Phongov model, ki za podoben rezultat skalarno zmnoži nek srednji enotski vektor med $\mathbf{V}$ in $\mathbf{L}$ z normalo $\mathbf{N}$:

$$Intenziteta_{odblesk} = (\frac{\mathbf{V}+\mathbf{L}}{\|\mathbf{V}+\mathbf{L}\|} \odot \mathbf{N})^n \tag{2.15}$$

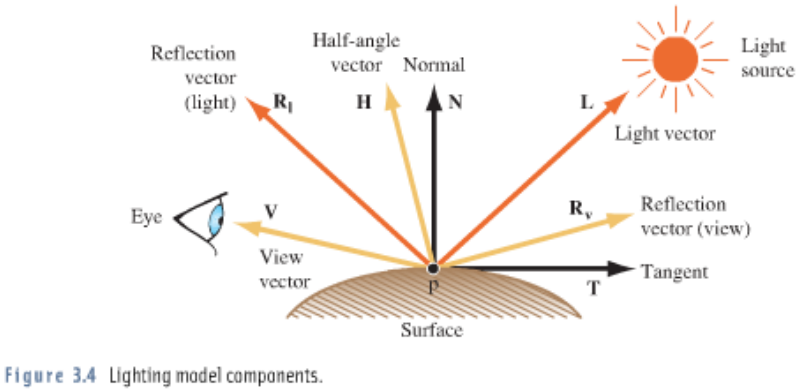


Figure 3.4 Lighting model components.

Slika 2.5: Komponente pri osvetljevanju [15]

[15]

2.4.8 Senčenje (angl. shading)

Senčenje je računanje barve za vsak fragment. Tu ne gre za senčilnike, ki so programi za dele OpenGL-ovega cevovoda. En primer senčenja je, da za vsak fragment preprosto uporabimo zadnje ogljišče primitiva. V angleščini je to *constant shading*. V OpenGL-u to izberemo s konstanto **GL_FLAT** kot parameter ukaza **glShadeModel**.

Drug primer senčenja je interpolacija barv ogljišč. Temu rečemo tudi Gourandovo senčenje. V angleščini je to *smooth shading*. V OpenGL-u to možnost izberemo s konstanto **GL_SMOOTH**.

Barve fragmentov lahko določimo tudi s teksturo.

Phongovo senčenje pa pomeni, da sploh ne uporabimo barve, izračunane na ogljiščih. Namesto tega interpoliramo normalo in izračunamo barvo za vsak fragment posebej. [15]

2.5 2D Uporabniški vmesnik

Interakcija v 3D aplikacijah je lahko izvedena na mnogo načinov. A bolje je izkoristiti načine za delo z računalnikom, s katerimi so uporabniki že seznanjeni. Zato se še vedno uporablja metafore, ki so se skozi čas izkazale za učinkovite.

2.5.1 TWL

TWL (Themable Widget Library) je knjižnica za grajenje grafičnih uporabniških vmesnikov za Javo, ki uporablja OpenGL. Omogoča grajenje gumbov, label, list itd. Za stil prikaza teh elementov uporablja slike in datoteke xml, ki iz slik izčrpajo potrebne podatke. [11]

2.5.2 HUD

HUD (Heads-Up Display) je grafična vsebina, prikazana na vrhu vseh ostalih elementov. Služi za prikaz raznih informacij in interakcijo z aplikacijo. Uporablja se predvsem v video igrah. Primer je prikaz časa, raznih točk, stanja likov, zemljevida itd.

2.6 Datoteke vrste .obj

Podatke o modelih, ki jih 3D aplikacija prikazuje, se lahko shranjuje na razne načine. V svoji nalogi sem zgradil aplikacijo, ki je odpirala datoteko vrste .obj. Vendar je podroben opis teh datotek tu neprimeren, ker sem dobival malce poenostavljeno in popačeno verzijo teh datotek. V datotekah, ki sem jih dobil, so bile koordinate ogljišč podane standardno. Vsaka vrstica, ki je predstavljala ogljišče, se je začela s črko v, ki so ji sledila tri realna števila, ki predstavljajo položaj v koordinatnem sistemu. Toda indeksi so bili v obratnem vrstnem redu in oznake g, ki označujejo skupine, so bile uporabljene nepravilno.

2.7 Metanje žarkov (angl. Ray Casting)

Metanje žarkov je metoda, po kateri za vsak piksel posebej izračunamo potovanje žarka skozi ta piksel, kot da bi potoval s kamere proti predmetu namesto obratno. Ko ugotovimo, kateri predmet je žarek zadel, izračunamo osvetlitev glede na barve virov svetlobe, barvo materiala predmeta, ter njihovo postavitev in orientacijo. Torej lahko uporabimo kar Blinnov-Phongov model osvetljevanja za vsako zadeto točko. V svoji nalogi nisem uporabljal omenjene metode za računanje barv, ampak za obračanje modela z miško.

Poglavje 3

Implementacija

Cilj aplikacije je prikazati žile nekega pacienta za njihovo obravnavo. Model žil naj se da obračati in si ga ogledati s poljubnega položaja. Prikaz naj omogoča stereo sliko, deljenje ploskev ter Phongovo osvetljevanje in senčenje. Aplikacija naj podpira odpiranje ustreznih datotek s podatki o žilah. Moja naloga se začne, ko je bil pacient že pregledan in je bila iz dobljenih signalov zgrajena datoteka, ki vsebuje lokacije ogljišč v prostoru in informacijo o topologiji. Lastnik podatkov žil, ki sem jih dobil kot vhod, je Laboratorij za slikovne tehnologije, Fakulteta za elektrotehniko, Univerza v Ljubljani. V prejšnjem poglavju je opisan del orodij in znanja, potrebnega za implementacijo vizualizacije žil. Sledi opis implementacije. Izdelal sem aplikacijo, ki vizualizira model žil. Deluje tako, da prebere datoteko s podatki modela, izračuna normale, zgradi *buffer object* in model prikaže na zaslonu. Uporabnik lahko uporablja različne ločljivosti in barvne globine, stereo sliko (vrste *interlaced*). Model lahko obrača ali pa prosto premika kamero v prostoru na več načinov. Model je izrisan z uporabo senčilnikov, ki uporabljajo Phongov model osvetljevanja in tudi senčenja. Uporabnik lahko uporabi *subdivision surface*, ki model zgladi, a za ceno več trikotnikov.

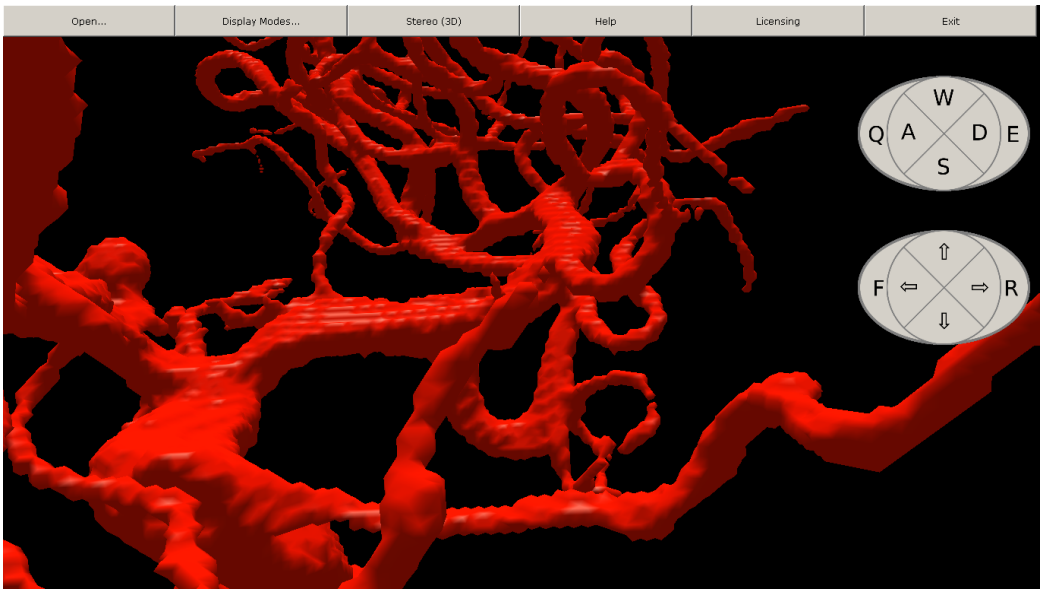
3.1 Okno aplikacije

Knjižnica LWJGL poenostavi delo z okni. Zato mi za operacijski sistem Windows mi ni bilo potrebno uporabljati WGL-ja. Razred **Display** ustvari novo okno in za okolje, v kateremu se program izvaja, poda možne nastavitve za okno ali

celozaslonski način.

3.2 OpenGL Shading Language in osvetljevanje

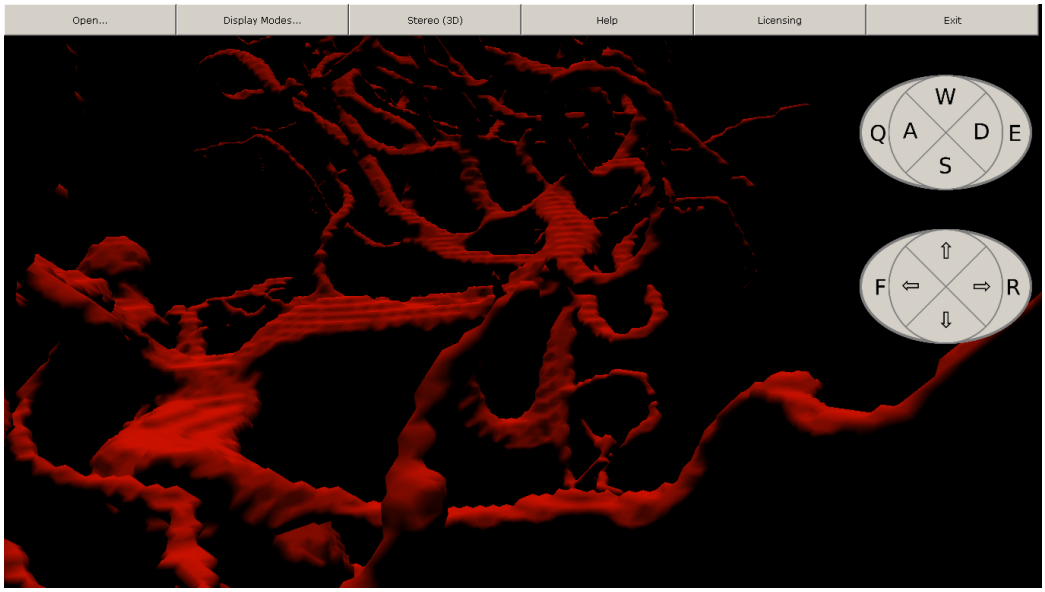
Tipke 1-5 izbirajo med različnimi senčilniki. To sem naredil zato, da sem lahko primerjal manjše spremembe v kodi. Tipka 5 da končno verzijo senčilnikov. Tipka 0 pa pokaže, kako izgleda model, če ga riše fiksna zmogljivost OpenGL-a.



Slika 3.1: Rezultat pri fiksni zmogljivosti.

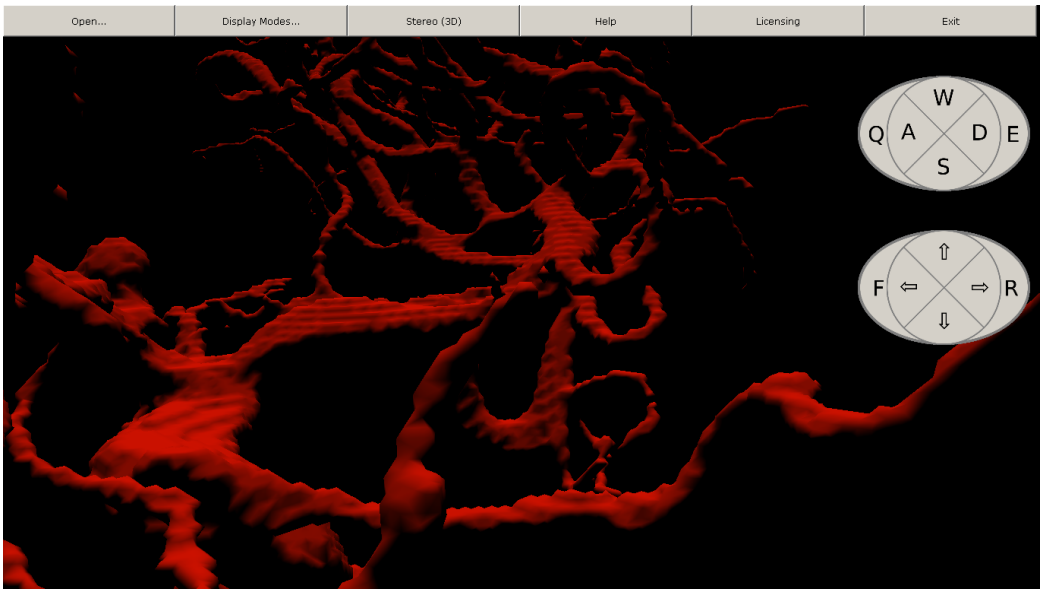
Sledi opis senčilnikov po vrstnem redu, po katerem so bili razviti. V enakem vrstnem redu so vezani na tipke. Vsak naslednji senčilnik je enak prejšnjemu s prištevkom spremembe v njegovem opisu. Pod opise podajam tudi slike, ki prikazujejo, kako izgleda prikaz modela po dodanih spremembah.

- Tipka 1: Normale vsakega ogljišča je potrebno transformirati z matriko **gl_NormalMatrix**. Model ni nikoli skaliran in zato dobljene normale ni potrebno normalizirati. Senčilnik ogljišč definira spremenljivko vrste *varying*, ki poda intenziteto difuzne osvetlitve. Senčilnik fragmentov preprosto pomnoži dobljeno (interpolirano) intenziteto z barvo krvi.



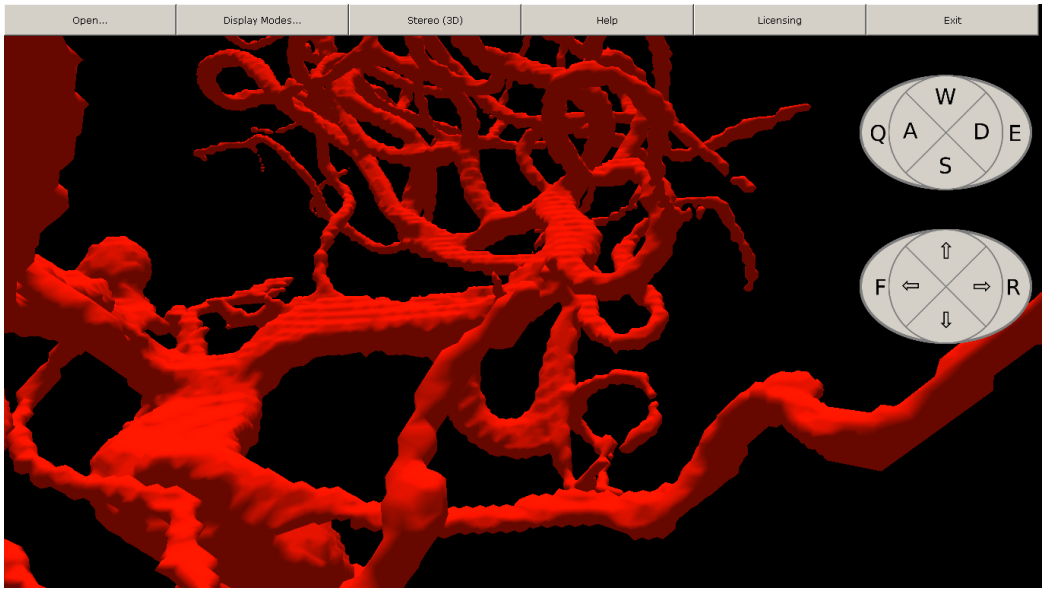
Slika 3.2: Rezultat pri senčilniku, vezanem na tipko 1.

- Tipka 2: Tu senčilnik ogljišč le poda normalo kot spremenljivko vrste *varying* (po njeni transformaciji). Barvo računa senčilnik fragmentov po normalizaciji normale.



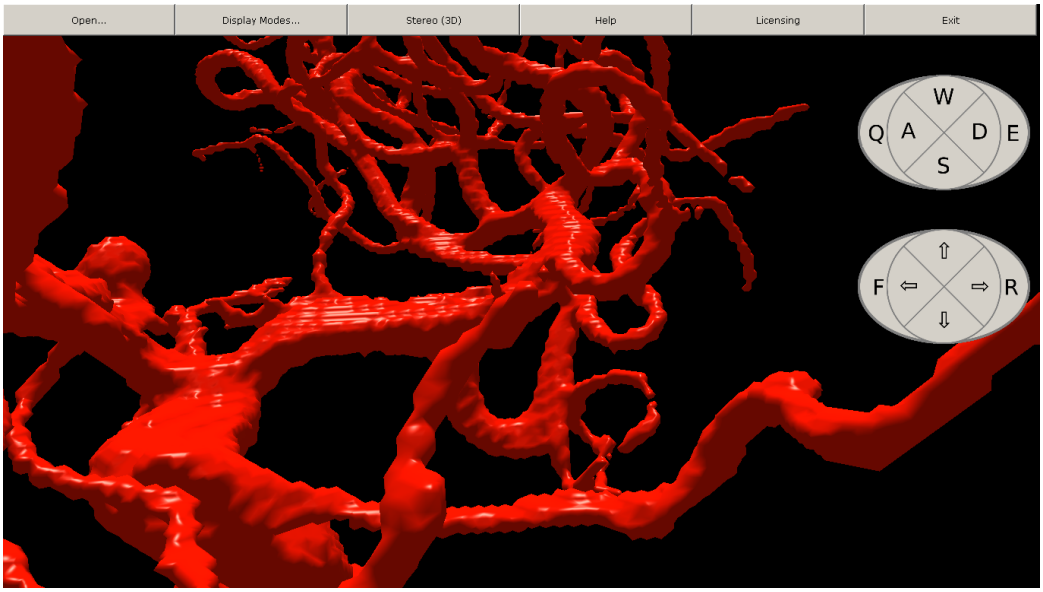
Slika 3.3: Rezultat pri senčilniku, vezanem na tipko 2, razlika s prejšnjim je tu še komaj opazna.

- Tipka 3: Tu senčilnik ogljišč doda barvo površine, ki nastane zaradi ambientne svetlobe.



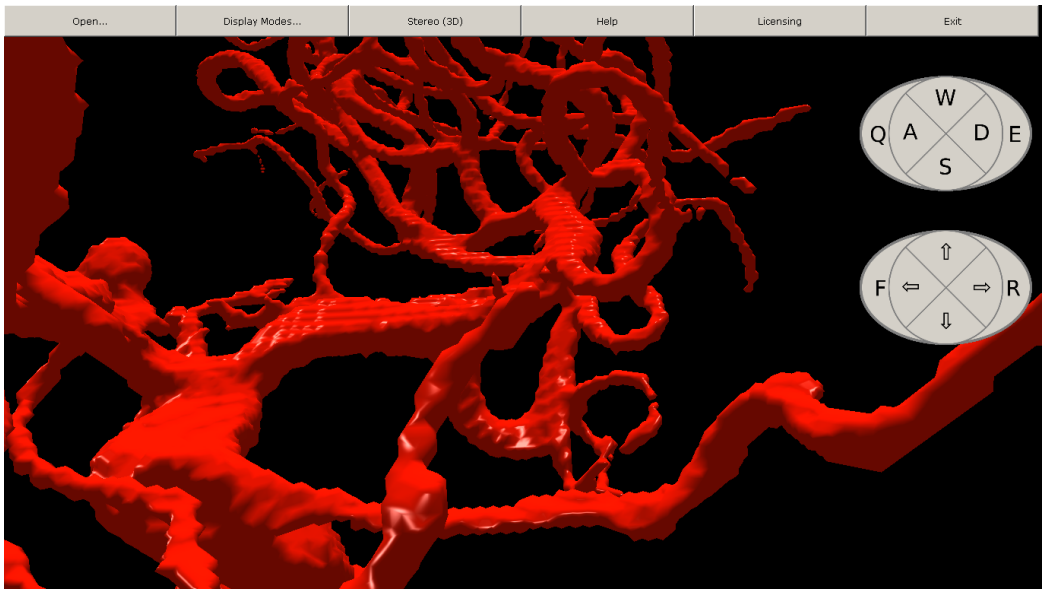
Slika 3.4: Rezultat pri senčilniku, vezanem na tipko 3.

- Tipka 4: Senčilnik ogljišč je tu enak zgornjemu. Senčilnik fragmentov pa doda še odblesk po Blinnovem-Phongovem modelu. Rezultat je sedaj že boljši od fiksne zmogljivosti cevovoda, ker barva ni izračunana za vsako ogljišče in interpolirana po fragmentih, ampak je izračunana za vsak fragment.



Slika 3.5: Rezultat pri senčilniku, vezanem na tipko 4. Tu senčilnik uporabi Blinnov-Phongov model, tako kot ga uporabi fiksna zmogljivost, a barva ni interpolirana, pač pa so normale, kar se tu končno bolj pozna.

- Tipka 5: Tu senčilnik ogljišč definira spremenljivko vrste *varying*, ki označuje lokacijo ogljišča. Interpolirano lokacijo ogljišča senčilnik fragmentov izkoristi, da izračuna odblesk po Phongovem modelu.



Slika 3.6: Rezultat pri končni verziji senčilnika. Vezan je na tipko 5.

[12]

3.3 Branje datoteke vrste .obj

Preden lahko pošljemo geometrijo modela skozi cevovod OpenGL-a, jo moramo nekako podati implementaciji OpenGL-a. Podatke moramo imeti za ta namen prirejene v glavnem pomnilniku ali že v spominu grafičnega pospeševalnika. Podatke modela lahko preberemo z drugih medijev, na primer diska. Program, ki sem ga napisal za branje .obj datotek, je prirejen za delo z .obj datotekami, ki sem jih dobil kot primer. Prebere lokacije vseh ogljišč in njihovih indeksov za vsak trikotnik. Trikotnike loči v skupine, če to prebere v datoteki. Sproti računa odmik od koordinatnega izhodišča, zato, da lahko model po končani konstrukciji postavi v koordinatno izhodišče. Poleg tega uporabi izračunane informacije, zato, da ustrezno premakne kamero stran od modela za razdaljo d_{kamera}, da se ga vidi celega,

a obenem ni predaleč. Naj bo d_{radij} radij očrtane krogle kvadra, ki vsebuje model. Naj bo θ manjši od dveh zornih kotov. Navadno je vertikalni manjši. Razdalja d_{kamera} je izračunana po naslednji formuli:

$$d_{kamera} = \frac{d_{radij}}{\tan \frac{\theta}{2}} \tag{3.1}$$

Po končanem branju datoteke, ki predstavlja model, program zažene grajenje *buffer objectov*. Izračuna normale ter jih skupaj z lokacijami ogljišč pošlje na stran OpenGL-ovega strežnika kot en VBO. Pošlje tudi indekse ogljišč vseh trikotnikov kot še en *buffer object*.
Algoritem izračuna normale po naslednjem postopku

- Za vsak trikotnik ABC izračuna vektorja $\overrightarrow{AB}$ in $\overrightarrow{AC}$.
- Izračuna vektorski produkt $\mathbf{n} = \overrightarrow{AB} \times \overrightarrow{AC}$.
- Za vsako ogljišče primitiva prišteje **n** ustrezni normali.
- Normalizira dobljene normale na vsakem ogljišču.

Po pošiljanju podatkov na stran strežnika bo model upodobljen s frekvenco zaslona. To je večinoma šestdesetkrat na sekundo. Pri vsakem izrisu je potrebno le povedati implementaciji OpenGL-a kateri podatki so ogljišča in kateri indeksi ter katere vrste primitivov rišemo. Podatki so že na strani strežnika in jih ni potrebno pošiljati ter s tem obremenjevati vhodno-izhodnega sistema in centralno procesne enote.

3.4 Deljenje ploskev (angl. subdivision surface)

Podatke za izris modela v moji aplikaciji hranijo objekti razreda **Mesh**. Hranijo tudi informacijo o modelu na strani odjemalca, zato, da lahko to informacijo izkoristijo za grajenje zglajenega modela, sestavljenega iz več trikotnikov. Program, ki sem ga napisal, izračuna nova ogljišča in nove indekse za nove trikotnike. Pridela 6-krat več geometrije in jo pošlje na stran strežnika. Tako ni potrebno za vsak izris pošiljati podatkov o geometriji in topologiji, ampak le povedati, katere podatke

na strani strežnika naj uporabi za upodabljanje. Algoritem za izračun gladkejšega modela je sledeč:

- za vsak trikotnik izračunaj novo ogljišče **F**, ki bo povprečje treh ogljišč trikotnika;
- za vsako ogljišče ugotovi, katerih trikotnikov se dotika in katera ogljišča so mu sosednja;
- za vsako ogljišče poglej za vsako njegovo sosednje ogljišče, če za to ogljšče še nisi izvedel sledečega koraka, sicer ga izvedi;
- med vsakima dvema povezanima ogljiščema izračunaj novo ogljišče **E**, ki je povprečje teh dveh ogljišč in dveh prej izračunanih ogljišč **F** dveh trikotnikov, ki si delita ti dve ogljišči;
- vsako ustvarjeno ogljišče **E** dodaj podatkovni strukturi, ki pomni, katero od treh ogljišč za vsak trikotnik je to;
- ugotovi, ali ima ogljišče **P** pogoje za izvajanje tega koraka (število robov n je enako številu ploskev n). Prestavi vsako originalno ogljišče **P** po formuli $\frac{\mathbf{F_{neighbours}}+2\mathbf{E_{neighbours}}+(n-3)\mathbf{P}}{n}$. $E_{neighbours}$ je povprečje ogljišč **E** med **P** in sosednjimi ogljišči. $\mathbf{F_{neighbours}}$ je povprečje ogljišč **F** na trikotnikih, ki se dotikajo **P**. n je število robov in število ploskev, ki se povprečijo;
- izračunaj nove indekse za vsak trikotnik.

[1]

3.5 Premikanje pogleda

Aplikacija premika kamero z uporabo transformacij pred upodabljanjem modela. Premik kamere desno je dosežen s premikom modela levo. Pogled navzdol je obračanje modela navzgor okrog lokacije kamere. To sem dosegel tako, da sem hranil podatke o nahajanju kamere. Pred vsakim upodabljanjem naložim enotsko matriko v **GL_MODELVIEW** matriko. Nato je prva transformacija, ki jo

podam, množenje slednje matrike z matriko, izpeljano z inverzne vrednosti kvaterniona, ki opisuje orientacijo kamere. Po tem podam še translacijo negativne vrednosti lokacije kamere.
Podatke o lokaciji in orientaciji kamere se spreminja preko tipkovnice in preko uporabniškega vmesnika. Recimo, da želimo obrniti kamero desno za 30° relativno na njeno trenutno lego (za -30° po y osi glede na lokalni koordinatni sistem kamere). Potem z desne pomnožimo kvaternion trenutne orientacije kamere s kvaternionom za vrtenje. Recimo, da želimo premakniti kamero naprej za 1 (nedoločeno OpenGL-ovo enoto), torej za -1 po z osi glede na lokalni koordinatni sistem kamere. Potem vektor premika, ki ga dodamo koordinatam nahajanja kamere, najprej zavrtimo s pomočjo kvaterniona trenutne orientacije kamere. Poleg upravljanja s tipkovnico sem implementiral še uporabniški vmesnik z dvema elipsama. Ko uporabnik klikne na krog, ki je na sredi elipse za orientacijo, bo pridodana orientacija izračunana glede na oddaljenost od središča v 2D koordinatnem sistemu zaslona. Tako združuje po dve rotaciji: okoli osi x in okoli osi y lokalnega koordinatnega sistema kamere. Magnituda za vsako od teh dveh je izračunana s trigonometričnimi funkcijami. Del elipse, ki ga ne pokriva krog, se uporablja kot dva gumba za vrtenje glede na os, ki je krog ne pokrije. Ali je uporabnik kliknil na elipso, se izračuna po razdaljah klika od gorišč elipse.
Efekt, ki kaže magnitudo in orientacijo premika ali obračanja kamere glede na klik na omenjeni krog, je dosežen z zlivanjem teksture na štirikotniku, ki se primerno obrača.

3.6 Obračanje modela žil

Za obračanje modela žil sem si predstavljal, da ga obdaja neka krogla, ki jo lahko primeš s kurzorjem in zavrtiš. Napisal sem kodo, ki ugotovi, kje to kroglo prebada žarek, ki gre iz vira kamere in prebada zaslon, kjer se nahaja kurzor. Program si zapomni lokacijo preboda na namišljeni krogli. Ko uporabnik premakne kurzor in še vedno "drži kroglo" (torej miškin gumb), bo program izračunal nov prebod. Obrnil bo kroglo proti lokaciji novega preboda.

3.7 Stereo slika

Stereo slika pomeni izris dveh slik, ene za levo oko in druge za desno. Vsakemu očesu je nato potrebno podati ustrezno sliko. Za to obstaja več načinov. V svoji aplikaciji sem naredil stereo izris za poseben monitor, ki oddaja v sodih vrsticah sliko za eno oko in v lihih vrsticah za drugo. Z ustreznimi očali se poskrbi, da vsako oko dobi le eno od slik. Prikaz stereo slike sem vezal na gumb "Stereo 3D" grafičnega uporabniškega vmesnika. Ko ga uporabnik aktivira, se spodaj prikaže drsnik. Ta drsnik spreminja očesno razdaljo dveh slik. Lahko gre tudi v negativno razdaljo. Zato aplikaciji ni potrebno vedeti, za katero oko so lihe vrstice in za katero sode.

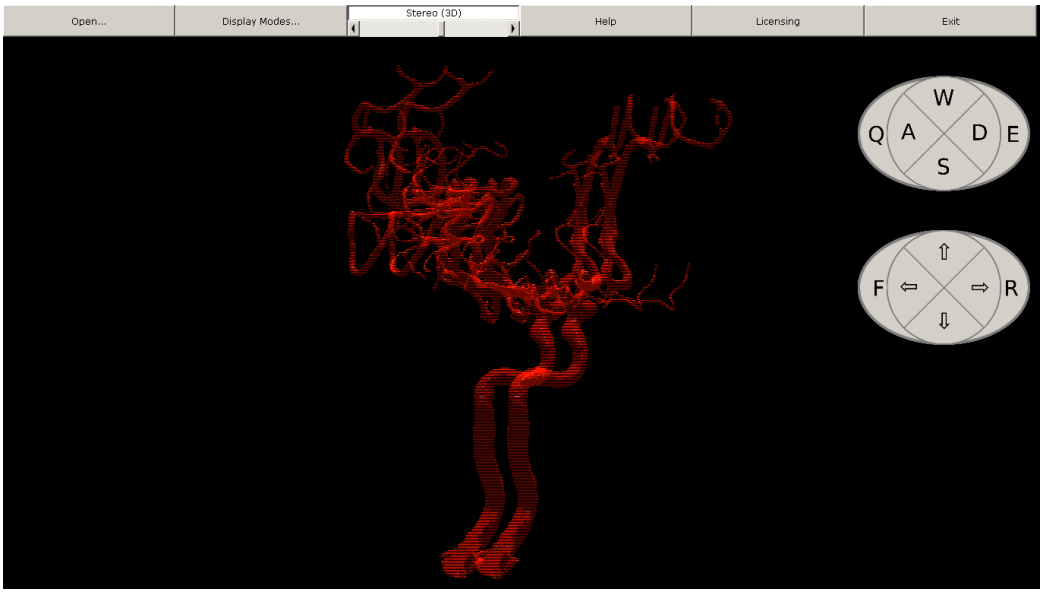
Ta prikaz sem naredil tako, da sem uporabil *stencil buffer*. Nastavil sem grafični kontekst na način, da se barva ni zapisala v *color buffer*. Namesto tega se na mestu v *stencil bufferju*, ki ustreza pikslu, na katerega se riše, spremeni vrednost na 1. Projekcijo sem nastavil na ortogonalno in narisal črto na vsako drugo vrstico. Nato sem model upodobil dvakrat z zamikom levo in desno za vsako oko. Enkrat tako, da je fragment prestal *stencil test*, kjer ima *stencil buffer* vrednost 0, drugič pa, da je prestal *stencil test*, kjer ima *stencil buffer* vrednost 1.

3.8 Shranjevanje nastavitev

Za shranjevanje nastavitev sem uporabil *interface* **java.io.Serializable**. S pomočjo **FileOutputStream**, **FileInputStream**, **ObjectInputStream**, **ObjectOutputStream** je pisanje objektov v datoteko in njihovo branje enostavno. V razred, katerega objekti vsebujejo shranjene nastavitve, lahko sproti vnešem nove spremenljivke za shranjevanje novih zmogljivosti aplikacije.

3.9 Pomoč in licence

Knjižnice, ki sem jih uporabljal, so bile na voljo pod BSD licencami. Zato morajo biti slednje nekako priložene programu. Vključil sem jih v sam program. Sestavil sem tudi kratko pomoč z navodili za uporabo programa, ki sem jo prav tako vključil v program.



Slika 3.7: Stereo slika, kot je vidna na navadnem zaslonu brez 3D očal.

Poglavje 4

Zaključek in sklepne ugotovitve

Za učinkovito programiranje 3D aplikacij v OpenGL-u je potrebno ogromno znanja iz matematike, programiranja, delovanja računalnika, delovanja OpenGL-a, pa predvsem volje in časa (ter diamantnih živcev). Poleg tega rezultat računalniškega upodabljanja pogosto ne pride od simulacije fizičnega dogajanja v resničnem svetu. Namesto tega mora čim bolj dobro lagati, da je v ozadju fizični svet. Potrebno je še znanje o modeliranju obnašanja slednjega. Kljub temu lahko uporabnik končni rezultat oceni v trenutkih. Ker čas vedno beži, je v vsaki iteraciji smiselno najprej popraviti dele aplikacije, ki najbolj motijo. V svoji nalogi sem se omejil le na eno vrsto površine in na bolj ali manj vnaprej podano obliko. Kljub temu je bila naloga časovno potratna in mislim, da bi, če bi delal na njej še leta, gotovo našel še in še prostora za izboljšavo ali optimizacijo.

V prvem in obsežnejšem delu diplome sem opisal razna znanja, ki so mi koristila pri pisanju aplikacije. V drugem delu pa sem besedilo bolj približal njenemu pisanju.

Nadaljnje delo bi lahko vsebovalo razne optimizacije, podporo za več različnih 3D (stereo) načinov prikaza, zaznavanje strukture žil, barvanje žil glede na njihovo funkcijo. Menim, da bi zdravnikom precej prav prišlo tudi ročno označevanje modelov in podpora njihovemu komentiranju ter kakšna podatkovna baza, da jim ne bi bilo potrebno samim organizirati datotek s podatki žil.

Literatura

[1] Catmull–clark deljenje ploskev na wikipediji. Spletna stran, 2012. http://en.wikipedia.org/wiki/Catmull%E2%80%93Clark_subdivision_surface.

[2] Eclipse na wikipediji. Spletna stran, 2012. http://en.wikipedia.org/wiki/Eclipse_%28software%29.

[3] Java na wikipediji. Spletna stran, 2012. http://en.wikipedia.org/wiki/Java_%28programming_language%29.

[4] Kvaternion na wikipediji. Spletna stran, 2012. http://sl.wikipedia.org/wiki/Kvaternion.

[5] Quaternion na wikipediji. Spletna stran, 2012. http://en.wikipedia.org/wiki/Quaternion.

[6] Spletna stran lwjgl-a. Spletna stran, 2012. http://lwjgl.org/.

[7] Spletna stran o lwjgl-u. Spletna stran, 2012. http://lwjgl.org/wiki/index.php?title=About_LWJGL.

[8] Spletna stran opengl-a za ukaz **glBindBuffer**. Spletna stran, 2012. http://www.opengl.org/sdk/docs/man/xhtml/glBindBuffer.xml.

[9] Spletna stran s sliko cevovoda opengl-a. Spletna stran, 2012. http://goanna.cs.rmit.edu.au/~gl/teaching/rtr&3dgp/notes/pipeline.html.

[10] Spletna stran s sliko procesorja za senčilnika opengl-a 2.0. Spletna stran, 2012. http://www.yaldex.com/open-gl/ch02lev1sec3.html.

[11] Spletna stran twl-ja. Spletna stran, 2012. http://twl.l33tlabs.org/.

[12] Spletna stran vaje za opengl shading language. Spletna stran, 2012. `http://www.lighthouse3d.com/tutorials/glsl-tutorial/?pipeline`.

[13] Združeni senčilniki na wikipediji. Spletna stran, 2012. `http://en.wikipedia.org/wiki/Unified_shader_model`.

[14] Donald Hearn and M. Pauline Baker. *Computer graphics with OpenGL.* Upper Saddle River: Pearson Prentice Hall, cop. 2004, Upper Saddle River, NJ 07458, 2004.

[15] Tom McReynolds and David Blythe. *Advanced graphics programming using openGL.* Morgan Kaufmann Publishers, 500 Sansome Street, Suite 400, San Francisco, CA 94111, 2005.

[16] Randi J. Rost, John M. Kessenich, Barthold Lichtenbelt, Hugh Malan, and Mike Weiblen. *OpenGL Shading Language, Second Edition.* Addison Wesley Professional (2006), Upper Saddle River, 2006.