Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Aleks Huč

Izdelava in uporaba senčilnikov v sodobni računalniški grafiki

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: 00022/2012
Datum: 10.04.2012

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

Kandidat: **ALEKS HUČ**

Naslov: **IZDELAVA IN UPORABA SENČILNIKOV V SODOBNI RAČUNALNIŠKI GRAFIKI**

MAKING AND USING SHADERS IN MODERN COMPUTER GRAPHICS

Vrsta naloge: Diplomsko delo univerzitetnega študija prve stopnje

Tematika naloge:

V diplomski nalogi predstavite teoretične osnove senčilnikov in primerjajte jezike za pisanje senčilnikov. V enem od jezikov implementirajte lastne senčilnike, ki pokrivajo različne dele grafičnega cevovoda, od senčilnikov oglišč, geometrije, teselacije, do senčilnikov fragmentov.

Mentor:

doc. dr. Matija Marolt

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

Dekan:

prof. dr. Nikolaj Zimic

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Aleks Huč, z vpisno številko **63090055**, sem avtor diplomskega dela z naslovom:

Izdelava in uporaba senčilnikov v sodobni računalniški grafiki

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,
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki "Dela FRI".

V Ljubljani, dne 31. avgust 2012 Podpis avtorja:

Za pomoč pri izdelavi diplomskega dela se zahvaljujem docentu dr. Matiji Maroltu, ki me je kot mentor vodil, spodbujal in usmerjal, hkrati pa se zahvaljujem vsem profesorjem in asistentom na dodiplomskem študiju za odstiranje novih razsežnosti in pogledov na računalništvo.

Kazalo

Kazalo slik

Kazalo izsekov kode

Povzetek

Na področju potrošniške računalniške grafike je s pojavom grafičnih kartic s programirljivim cevovodom prišlo do velikega preobrata pri razvoju in izrisu računalniške grafike. Kaj in kako se bo nekaj izvedlo v posamezni programirljivi stopnji cevovoda, je določeno z majhnim programom, z drugo besedo, senčilnikom (angl. shader).

Cilj diplomskega dela je predstavitev osnov sodobne računalniške grafike oziroma uporabe senčilnikov v računalniški grafiki ter njihova izdelava in implementacija v igrico ali druge podporne grafične aplikacije. Na začetku predstavimo teoretične osnove senčilnikov in primerjamo programske jezike za pisanje senčilnikov ter programske grafične vmesnike, ki predstavljajo ogrodje za izvajanje senčilnikov. V nadaljevanju izdelamo praktične senčilnike v programskem grafičnem vmesniku OpenGL in programskem jeziku za senčilnike OpenGL Shading Language. Izdelane senčilnike implementiramo v podporne grafične aplikacije in v igrico, ki smo jo že predhodno razvili pri predmetu računalniška grafika in tehnologija iger na Fakulteti za računalništvo in informatiko. Podporne grafične aplikacije razvijemo ob izdelavi senčilnikov z namenom, da senčilnike preizkusimo v kontroliranem okolju. Izdelamo in implementiramo senčilnike za krožno meglenje celotne scene, sisteme delcev, Phongovo globalno osvetljevanje scene, mapiranje senc in na koncu še teselacijski senčilnik tridimenzionalnih modelov v sceni. V zaključku predstavimo ključne ugotovitve pri izdelavi in implementaciji senčilnikov ter njihovih podpornih grafičnih aplikacij.

Ključne besede:

Računalniška grafika, grafična kartica, grafični cevovod, senčilnik, OpenGL, OpenGL Shading Language, Lightweight Java Game Library, Java, računalniške igre, teselacija, meglenje scene, sistem delcev, Phongom model osvetljevanja, mapiranje senc.

Abstract

In the area of consumer computer graphic was a major turning point in the development of graphics and it's rendering, when new graphic cards with programmable pipeline emerged. What and how something will be done in a single stage of programmable pipeline is provided with a small program called shader.

The aim of the thesis is to present the basics of modern computer graphics and use of shaders in computer graphics. We start by presenting the theoretical basis of shaders and comparing programming languages for writing shaders and software graphical interfaces that represent the framework for the implementation of shaders. After that we work on practical examples of shaders with graphical programming language OpenGL and with language for writing shaders called OpenGL Shading Language. We implement shaders in graphics applications and in a game, that we developed in the subject computer graphics and gaming technology at Faculty of Computer and Information Science. Supporting graphics applications are developed when creating shaders in order to test them in a controlled environment. We create and implement shaders for circular blurring of the whole scene, particle systems, Phong global illumination, shadow mapping and three-dimensional model tessellation. In conclusion, we present key findings in development and implementation of cars and their supporting graphic applications.

Keywords:

Computer graphics, graphics card, graphics pipeline, shader, OpenGL, OpenGL Shading Language, Lightweight Java Game Library, Java, computer games, tessellation, scene blurring, particle system, Phongom lighting, shadow mapping.

Poglavje 1

Uvod

Računalniška grafika v osebnih računalnikih in pripadajoča strojna oprema sta naredili velik napredek, odkar se je na trgu prvič pojavila cenovno dostopna grafična kartica 3Dfx Voodoo leta 1995. Čeprav je porast zmogljivosti omogočal osebnim računalnikom hitreje izrisovati grafiko, se sama kakovost grafike ni veliko spremenila. Glavna omejitev za izboljšanje realističnosti računalniške grafike je bil fiksno določen nabor funkcij, ki so jih za izrisovanje grafike ponujale takratne grafične kartice. Nabor funkcij in grafične algoritme so določili razvijalci arhitekture grafičnih kartic in s tem omejili razvijalce in oblikovalce iger na uporabo le teh.

V tem času so razvijalci in oblikovalci za kakovostno in foto-realistično grafiko uporabljali grafično orodje RenderMan, ki ga je razvil filmski studio Pixar Animation Studios. RenderMan je bil uporabljen za razvoj in oblikovanje animiranih filmov, kot sta Toy story in A bug's life, ki sta imela za tisti čas najbolj napredno foto-realistično grafiko. RenderMan-ova programirljivost mu je omogočala, da se je razvijal z nastajanjem novih izrisovalnih tehnik, ki so jih pisali sami razvijalci. Ker ni bilo postavljenih strogih omejitev pri izračunavanju, je RenderMan ponudil razvijalcem zelo veliko fleksibilnost in možnost za izražanje kreativnosti. Zaradi svoje kompleksnosti je bil RenderMan realiziran kot programsko orodje in ni bil namenjen za izrisovanje računalniške grafike v realnem času.

Okoli leta 2000 se je na trgu prvič pojavila cenovno dostopna strojna oprema, ki je podpirala osnovno programirljivost grafičnih funkcij in algoritmov na sami grafični kartici, podobno kot RenderMan, vendar v realnem času.

Najbolj uporabljana programska vmesnika, ki sta se razvijala vzporedno z razvojem cenovno dostopnih grafičnih kartic, sta DirectX in OpenGL. Istočasno s strojno opremo sta oba posodobila svoje vmesnike tudi za programiranje posameznih stopenj grafičnega cevovoda in s tem omogočila razvijalcem in oblikovalcem mnogo večjo fleksibilnost pri izrisovanju grafike, izdelavo novih operacij in postopkov ter omogočila, da si je vsak razvijalec prilagodil delovanje grafičnega cevovoda svojim potrebam[1].

V današnjem času je uporaba fiksnega grafičnega cevovoda skoraj čisto zamrla in se uporablja le za preprosto grafiko, medtem ko se senčilniki izkazali za zelo fleksibilno orodje in so postali *de facto* standard za kakršnokoli resnejšo uporabo grafike. Zahteve po močnejši, naprednejši in še bolj prilagodljivi grafiki postavljajo predvsem razvijalci računalniških iger in zabavna industrija. Pokazatelj hitrega razvoja računalniške grafike je že to, da so prvi programirljivi cevovodi ponudili le programiranje senčilnikov oglišč in senčilnikov fragmentov. Današnje grafične kartice pa omogočajo še programiranje senčilnikov geometrije in senčilnikov teselacije.

Prve grafične kartice s programirljivim cevovodom so imele določeno število procesnih enot, na katerih se je lahko paralelno izvajalo le več instanc senčilnika oglišč in določeno število drugih procesnih enot, na katerih se je lahko paralelno izvajalo le več instanc senčilnika fragmentov. S posplošenjem procesorskih enot so proizvajalci grafičnih kartic omogočili, da sta se lahko na vseh procesorskih enotah izvajali dve vrsti senčilnikov. Posplošenje procesorskih enot je pripeljalo do visokega povečanja zmogljivosti, prav tako pa je odprlo možnosti za razvoj novih vrst senčilnikov.

V zadnjem času se je opazno povečala tudi uporaba grafičnih kartic za izračun kompleksnih matematičnih problemov z visoko stopnjo paralelnosti, saj so za take probleme grafične kartice veliko bolj zmogljive od splošno

namenskih centralnih procesnih enot v računalnikih. Področje računalniške grafike se zelo hitro spreminja in vedno najdemo nove načine, kako uporabiti grafične kartice.

Cilj diplomskega dela je predstavitev osnov sodobne računalniške grafike oziroma uporabe senčilnikov v računalniški grafiki ter njihova izdelava in implementacija v igrico ali druge podporne grafične aplikacije. Na začetku pregledamo programirljive dele grafičnega cevovoda. Njihovo delovanje določajo senčilniki. Primerjamo programske jezike za pisanje senčilnikov ter programske grafične vmesnike, ki predstavljajo ogrodje za izvajanje senčilnikov. V nadaljevanju določimo principe razvoja senčilnikov in opišemo strukturo senčilnikov. Opišemo, kako v programskem grafičnem vmesniku OpenGL ustvarimo senčilnike, senčilniške programe, ki poskrbijo za uporabo senčilnika, ter kako lahko grafične aplikacije komunicirajo s senčilnikom. Sledi izdelava senčilnikov v programskem grafičnem vmesniku OpenGL in programskem jeziku za senčilnike OpenGL Shading Language. Izdelane senčilnike implementiramo v podporne grafične aplikacije in v igrico, ki smo jo že predhodno razvili pri predmetu računalniška grafika in tehnologija iger. Podporne grafične aplikacije razvijemo ob izdelavi senčilnikov z namenom, da senčilnike preizkusimo v kontroliranem okolju. Izdelamo in implementiramo senčilnike za krožno meglenje celotne scene, sisteme delcev, Phongovo globalno osvetljevanje scene, mapiranje senc in na koncu še teselacijski senčilnik tridimenzionalnih modelov v sceni. V zaključku predstavljamo ključne ugotovitve pri izdelavi in implementaciji senčilnikov ter njihovih podpornih grafičnih aplikacij.

Poglavje 2

Teorija senčilnikov

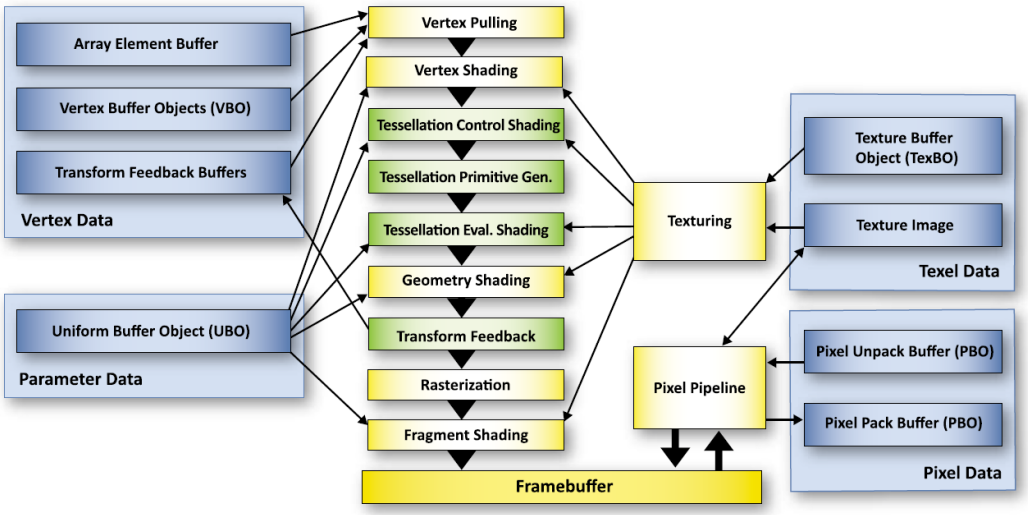
Na področju računalniške grafike je senčilnik (angl. shader) računalniški program, ki se ga uporablja za opis postopka, ki ga izvede posamezen programirljiv del grafičnega cevovoda ter omogoča visoko stopnjo prilagodljivosti. Programirljiv grafični cevovod je v večini primerov nadomestil fiksno določen grafični cevovod, ki je omogočal le osnovno transformiranje modelov in osnovne funkcije za izris posameznih pikslov (najmanjša enota na zaslonu na, katero lahko vplivamo), ki so jih določili proizvajalci grafičnih kartic[16].

2.1 Programirljiv grafični cevovod

Grafične kartice so eden pomembnejših delov računalnika, ki skrbijo za prikaz slike na zaslonu. Na grafični kartici se nahajata grafični procesor in grafični pomnilnik. Grafični procesor poskrbi za obdelavo in pripravo slike za izris na zaslonu. Temu postopku rečemo grafični cevovod. Grafični pomnilnik pa shranjuje podatke, ki nastanejo pri obdelavi slike, ali pa vanj shranimo vnaprej definirane grafične objekte in tako zmanjšamo komunikacijo med grafično kartico in procesorjem.

Grafični cevovod je postopek, ki za vhod prejme prostorsko sceno in kot izhod poda dvodimenzionalno rasterizirano sliko (Slika 2.1). Posamezne stopnje programirljivega cevovoda so opisane v spodnjih podpoglavjih (2.1.1

- 2.1.9). Podatki o sceni se podajo grafični kartici, na kateri se obdelajo v grafičnem cevovodu. Grafična kartica poda na izhod sliko, ki jo lahko prikažemo na zaslonu. Grafični cevovodi se stalno razvijajo in dopolnjujejo. V trenutni različici OpenGL je podprtih pet vrst senčilnikov, in sicer: senčilniki oglišč, nadzorni senčilnik teselacije, ocenjevalni senčilnik teselacije, senčilnik geometrije in senčilnik fragmentov. Vsak senčilnik predstavlja samostojno programirljivo enoto grafičnega cevovoda, še vedno pa grafični cevovod vsebuje nekaj fiksnih stopenj, predvsem za operacije, ki so dobro določene in pri katerih ni potrebe po prilagodljivosti.

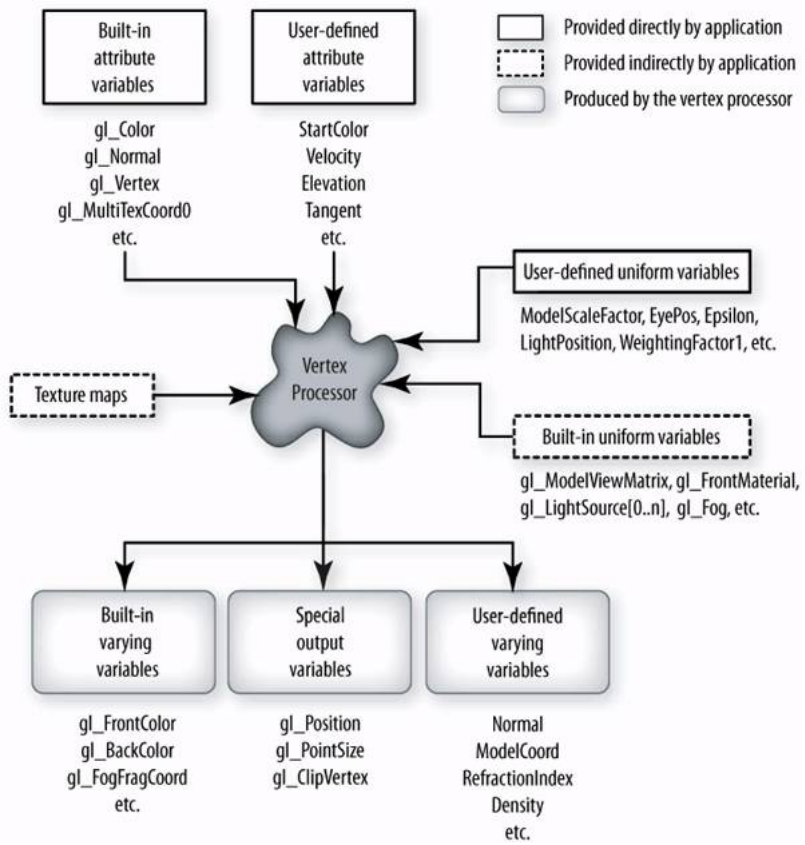


Slika 2.1: Sodobni OpenGL 4.x grafični cevovod[2].

2.1.1 Pridobivanje oglišč

Grafična kartica dobi podatke o ogliščih na različne načine: kot seznam oglišč (angl. Array Element Buffer), kot pomnilnik oglišč (angl. Vertex Buffer Object) ali pa spremenjena oglišča, ki so nastala pri prejšnjih iteracijah cevovoda (angl. Transform Feedback Buffers). V tej fazi se pripravijo oglišča in njihovi atributi za obdelavo.

2.1.2 Senčilnik oglišč



Slika 2.2: Vhodne in izhodne spremenljivke pri procesorju oglišč[3].

Procesor oglišč je programirljiva enota, ki izvede operacije nad podanimi oglišči in z njihovimi lastnostmi. Tradicionalne grafične operacije, ki jih izvaja procesor oglišč, so: transformiranje oglišč, transformiranje normal in normaliziranje, generiranje koordinat za teksture, transformiranje koordinat tekstur, osvetljevanje, barvanje ...

Ker je procesor programirljiv, ga lahko uporabimo še za mnoge druge izračune. Senčilniki oglišč so programi, ki se izvajajo na procesorju oglišč in z njimi lahko izvedemo skoraj kakršnokoli operacijo nad posameznimi oglišči in njihovimi lastnostmi. Ni možno imeti senčilnika oglišč za samo eno funkcionalnost, vendar mora nadomestiti vsaj vse tradicionalne operacije, ki se

izvedejo v tej stopnji cevovoda, kot je na primer transformacija oglišč iz koordinatnega sistema modela v koordinatni sistem projekcije kamere.

Procesor izvaja operacije za eno oglišče hkrati. Kljub temu pa omogoča paralelno izvajaje operacij nad posameznimi neodvisnimi oglišči.

Procesor oglišč dobi na vhodu vgrajene spremenljivke in tudi spremenljivke, ki jih je definiral uporabnik, ima pa tudi dostop pomnilnika tekstur. Vhodne spremenljivke so namenjene samo senčilniku oglišč (*in* spremenljivke) ali pa jih lahko uporabljamo tudi v drugih senčilnikih (*uniform* spremenljivke). Kot izhod pa lahko poleg že vgrajenih spremenljivk poda tudi uporabniško določene spremenljivke (*out* spremenljivke) [3] (Slika 2.2).

2.1.3 Nadzorni senčilnik teselacije

Teseliranje je proces ustvarjanja ravnine z uporabo ponavljajočih geometrijskih oblik brez prekrivanja in brez lukenj.

Procesor za nadzor teselacije je programirljiva enota, ki za vhod vzame oglišča in iz njih ustvari krpo (angl. Patch) z omejeno velikostjo. Krpa je sestavljena iz seznama oglišč, lastnosti posameznih oglišč in množico lastnosti za krpo kot celoto. Pod lastnosti krpe spada tudi stopnja teselacije, ki pove, kako nadrobno bo podana krpa teselirana. Program, ki se izvede na procesorju, se imenuje nadzorni senčilnik teselacije, ki pripravi vhodne podatke za uporabo v teselacijskem generatorju primitivov.

Za vsako obdelano krpo je potrebno več klicev nadzornega senčilnika teselacije, po enega za vsako oglišče izhodne krpe. Vsak klic zapiše vse atribute za pripadajoče vozlišče krpe. Nadzorni senčilnik teselacije lahko bere izhodna oglišča drugih klicev senčilnika, prav tako pa lahko bere in zapisuje atribute za celotno krpo. Za obdelavo ene krpe je torej potrebnih več klicev senčilnika, zato se po koncu vseh klicev izhodna oglišča in atributi združijo v izhodno krpo, ki je nato poslana v obdelavo naslednji stopnji grafičnega cevovoda[4, 17].

2.1.4 Teselacijski generator primitivov

Teselacijski generator primitivov je del fiksno določenega grafičnega cevovoda, ki razstavi vhodno krpo v množico primitivov glede na stopnjo teselacije, ki določi, kako drobno razdeljen naj bo izhod. Generator primitivov začne s trikotnikom ali štirikotnikom in razdeli vsako stranico lika približno na vrednost, ki je shranjena v seznamu zunanjih stopenj teselacije. Seznam vsebuje po eno vrednost za vsako stranico primitiva. Notranjost primitiva pa je teselirana glede na vrednost, ki je shranjena v seznamu notranjih stopenj teselacije. Ta seznam vedno vsebuje samo eno število.

Teselator deluje na tri načine: *triangles* in *quads*, v katerih razdeli trikotno ali štirikotno obliko v množico trikotnikov, ki pokrijejo vhodno krpo, in *isolines*, v kateremu razdeli štirikotno obliko v množico zaporedno prelomljenih črt, ki potekajo horizontalno skozi krpo. Vsako generirano vozlišče pridobi u in v ali u, v in w koordinate, ki predstavljajo relativno lokacijo generiranega oglišča v razdeljenemu trikotniku ali štirikotniku[4].

2.1.5 Ocenjevalni senčilnik teselacije

Procesor za ocenjevanje teselacije je programirljiva enota, ki za vsako oglišče, ki ga generira teselacijski generator primitivov, izračuna lokacijo in ostale atribute iz u in v ali u, v ali w koordinat. Program, ki se izvaja na procesorju za ocenjevanje teselacije, se imenuje ocenjevalni senčilnik teselacije. Pri izračunu končnih atributov oglišč lahko ocenjevalni senčilnik teselacije bere atribute kateregakoli oglišča v krpi, ki je bila generirana. Posamezni klici senčilnika so popolnoma neodvisni, vendar imajo vsi enake spremenljivke na vhodu.

Teselator izvaja operacije na ogliščih po tem, ko so že bila obdelana s strani senčilnika oglišč. Primitivi, ki so bili generirani v postopku teselacije, so nato podani naslednji stopnji grafičnega cevovoda[4].

2.1.6 Senčilnik geometrije

Procesor za geometrijo dobi na vhod posamezen primitiv in ima dostop do vseh lastnosti njegovih oglišč, ki jih lahko uporabi za izdelavo novih primitivov. Program, ki se izvaja na procesorjih geometrije, se imenuje senčilnik geometrije. Senčilnik geometrije ima fiksno določen tip primitivov na izhodu, to je točka, zaporedje črt ali pa zaporedje trikotnikov, ki jih definira z oglišči. Na izhod lahko pošlje več primitivov in ni potrebno, da so med seboj povezani. Primitivi, ki so nastali v tej stopnji cevovoda, se naprej obdelujejo popolnoma identično kot primitivi, ki so bili definirani že na vhodu v grafični cevovod[5].

2.1.7 Povratna informacija o transformaciji

Povratna informacija o transformaciji je stopnja cevovoda, ki izbrane primitive, njihova oglišča in lastnosti shrani v pomnilniški objekt. Torej shrani transformirane primitive, ki jih lahko nato uporabljamo v drugih stopnjah grafičnega cevovoda[6].

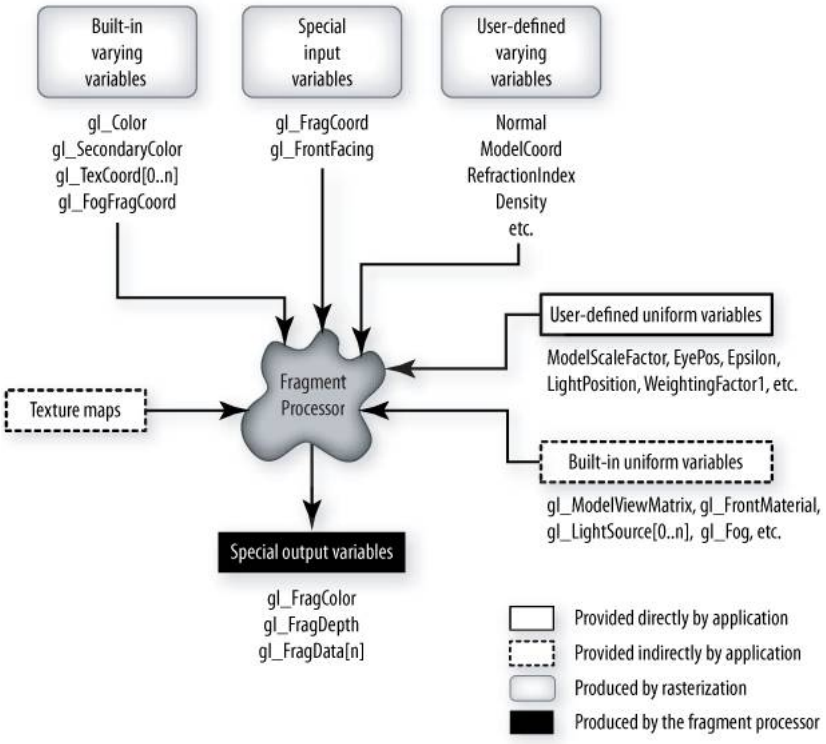
2.1.8 Rasterizacija

Geometrijski primitivi, ki se uporabljajo v grafičnem cevovodu, vsebujejo podatke za vsako njihovo oglišče. V tej stopnji cevovoda primitive razstavimo na manjše enote, ki so enako velike, kot piksli v pomnilniku, v katerega shranjujemo obdelane slike za prikaz na zaslonu. Temu postopku pravimo rasterizacija. Vsak delec, ki ga generiramo pri rasterizaciji, se imenuje fragment.

Na primer: Črto, ki je definirana z dvema ogliščema in na zaslonu pokrije pet pikslov, z rasterizacijo pretvorimo v pet fragmentov. Fragment vsebuje koordinati, ki določata mesto na zaslonu, globino, barvo, koordinate teksture ... Vrednosti lastnosti fragmentov dobimo z interpolacijo med vrednostmi oglišč primitiva. Takoj po rasterizaciji ima vsak fragment primarno in sekundarno barvo in končna barva se nato določi glede na zahtevo tako,

da ju interpoliramo ali pa uporabimo eno od teh dveh barv[3].

2.1.9 Senčilnik fragmentov



Slika 2.3: Vhodne in izhodne spremenljivke pri procesorju fragmentov[3].

Procesor fragmentov je programirljiva enota, ki izvaja operacije nad fragmenti in njihovimi lastnostmi. Tradicionalne grafične operacije, ki jih izvaja procesor fragmentov, so : operacije nad interpoliranimi vrednostmi, dostop do tekstur, uporaba tekstur, uporaba megle, seštevanje barv ...

Procesor fragmentov omogoča še množico drugih operacij, vendar lahko izvaja operacije le za en fragment hkrati. Kljub temu pa omogoča paralelno izvajaje operacij nad posameznimi neodvisnimi fragmenti.

Na procesorju fragmentov se izvajajo senčilniki fragmentov, ki izvajajo operacije nad posameznimi fragmenti, ki nastanejo med postopkom rasterizacije. Ni možno imeti senčilnika fragmentov za samo eno nalogo, ampak mora nadomestiti vse tradicionalne operacije, ki se izvedejo v tej stopnji cevovoda. Senčilnik fragmentov ne more spreminjati koordinat fragmentov, ki opisujejo, kje na zaslonu se pojavijo. Če so slike za teksture že shranjene v pomnilniku tekstur, se senčilnik fragmentov lahko uporabi za procesiranje pikslov, ki potrebujejo dostop do piksla in njegovih sosedov. Trikotnik je lahko izrisan z vključenimi teksturami, tako da senčilnik fragmentov prebere sliko iz pomnilnika tekstur in jo nanese na trikotnik, medtem ko izvaja tradicionalne operacije, kot so: povečevanje pikslov, velikost in pogled, pogled v tabele barv, konvolucija, barvne matrike ...

Procesor oglišč dobi kot vhod vgrajene spremenljivke in tudi spremenljivke, ki jih je definiral uporabnik, ima pa tudi dostop pomnilnika tekstur. Vhodne spremenljivke so lahko namenjene samo senčilniku oglišč (*in* spremenljivke) ali pa jih lahko uporabljamo tudi v drugih senčilnikih (*uniform* spremenljivke). Kot izhod pa poda v naprej določene spremenljivke in spremenljivke, ki jih je določil uporabnik (*out* spremenljivke) (Slika 2.3). Fragmenti se nato pošljejo naprej po grafičnem cevovodu, v katerem se izvede množica preverjanj, kot na primer preverjanje prekrivanja in globine. Potem se fragmenti shranijo v pomnilnik fragmentov, ki ga uporabimo za izris slike na zaslonu[3].

2.2 Programski jeziki za senčilnike

2.2.1 Časovni pregled

Rob Cook in Ken Perlin sta bila prva, ki sta razvila programski jezik za opis operacij senčilnikov. Njun razvoj se je nanašal samo na izrisovanje v nerealnem času. Perlin je definiral funkcije za šum in vpeljal konstrukte za nadzor delovanja. Cook pa je delal na senčilnih drevesih pri filmskem studiu Lucasfilm, definiral klasifikacijo senčilnikov, kot senčilnike površja, senčilnike

za osvetljevanje, senčilnike atmosfere ... Prav tako je določil možnost definiranje operacij senčilnikov kot izraz.

Njuno delo se je razvilo v razvoj programskega jezika za opis senčilnih operacij, ki ga je prevzel Pat Hanrahan in ga je leta 1988 pripeljal do izdaje prve verzije RenderMan Interface Specification pod okriljem filmskega studia Pixar. Sčasoma je RenderMan postal *de facto* standard za senčilni jezik, ki ga uporabljamo za izrisovanje v nerealnem času, predvsem v filmskih studiih in se razširjeno uporablja še danes.

Prvi programski jezik za senčilnike, ki je omogočal izrisovanje v realnem času, je bil predstavljen na University of North Carolina. Demonstracija je potekala na kompleksni paralelni arhitekturi PixelFlow, ki je bila razvita v 90-tih letih. Programski jezik je omogočal izrisovanje scen s proceduralnim senčenjem s tridesetimi slikami na sekundo in ta programski jezik za senčilnike je opisal Marc Olano leta 1998.

Po odhodu z univerze se je Olano pridružil ekipi na Silicone Graphics Inc., ki je razvijala programski jezik za senčilnike, ki teče znotraj programskega vmesnika OpenGL. Delo se je zaključilo leta 2000 z izidom prvega komercialno dostopnega in visoko nivojskega programskega jezika za pisanje senčilnikov, ki se izvaja v realnem času, z imenom OpenGL Shader. Problem OpenGL Shader je bil v tem, da je predvideval, da bo OpenGL grafični vmesnik uporabljen kot zbirnik za zaganjanje senčilnikov.

Junija leta 1999 se je v laboratoriju za računalniško grafiko na Stanfordu začel razvoj programskega jezika za senčilnike, ki se bo izvajal v realnem času in bo za poganjanje uporabljal že obstoječo tehnologijo potrošniških grafičnih kartic. Jezik se je imenoval Stanford Real-Time Shading Language in je bil predstavljen leta 2001.

Programski jeziki za senčilnike, ki so najbolj razširjeni danes, so OpenGL Shading Language (GLSL), Microsoftov High Level Shader Language (HLSL) in NVIDIA-in Cg ter so prizadevanja za definiranje ekonomsko rentabilnega, realno časovnega in visoko nivojskega programskega jezika za pisanje senčilnikov. Specifikacijo za GLSL je izdal Dave Baldwin iz podjetja 3DLabs

oktobra 2001. NVIDIA-ina specifikacija za Cg je bila izdana junija 2002, specifikacija za Microsoftov HLSL pa je bila izdana novembra 2002. Jeziki vsebujejo nekaj podobnosti, saj so podjetja, ki so jih razvila, zelo povezana[3].

2.2.2 OpenGL Shading Language (GLSL)

Zaradi uspeha standarda OpenGL, so na OpenGL ARB (Architecture Review Board) razvili programski jezik za senčilnike, ki je zelo napreden, visoko nivojski, strojno neodvisen, dovolj enostaven in dovolj močan, da bo, kar se da dolgo, kljuboval stalnemu razvoju na tem področju. S tem so drastično zmanjšali potrebo po razvijanju dodatkov k osnovni funkcionalnosti, kot je bilo potrebno pri fiksnem cevovodu.

Glavne lastnosti GLSL:

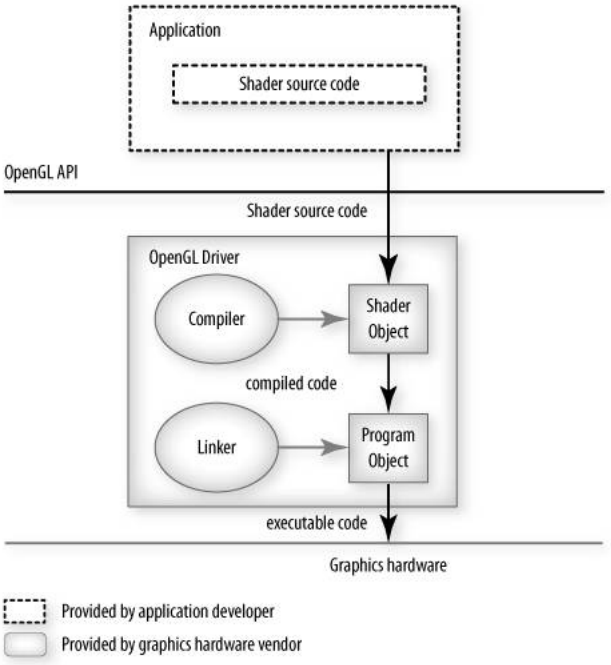
- Sintaksa jezika bazira na programskem jeziku C.
- Osnovna struktura in veliko ključnih besede je enakih kot v programskem jeziku C.
- Vektorji in matrike so vključeni kot osnovni tipi spremenljivk.
- Vsebuje tipe spremenljivk *in*, *uniform* in *out*, s katerimi upravljamo z branjem in pisanjem spremenljivk.
  - Spremenljivke tipa *in* omogočajo komunikacijo, ki zahteva dostop do pogosto spreminjajočih spremenljivk v aplikaciji za vsako.
  - Spremenljivke tipa *in* in *out* so izhod iz ene vrste senčilnika in so vhod v drugo vrsto senčilnika.
  - Spremenljivke tipa *uniform* omogočajo komunikacijo, ki zahteva dostop do počasi spreminjajočih spremenljivk v aplikaciji.
- Dodan je podatkovni tip *sampler*, ki skrbi za dostop do tekstur.
- Vgrajena imena spremenljivk se lahko uporabljajo za dostop do stanja OpenGL in za komunikacijo s fiksno funkcionalnostjo OpenGL.

- Vgrajenih je veliko funkcij za izvedbo pogostih grafičnih operacij.
- Deklaracija funkcij je potrebna, podpira tudi ponovno definiranje funkcij, glede na število in tip spremenljivk.
- Spremenljivke se lahko deklarira šele, ko jih potrebujemo.

GLSL je zasnovan za uporabo znotraj OpenGL izvršilnega okolja, zato je zelo povezan z arhitekturo OpenGL.

Izvirna koda senčilnikov se prevede šele ob izvajanju, zato ni potrebe po več prevedenih programih, ki so namenjeni različnim platformam.

Visok nivo programskega jezika omogoča neodvisnost od strojne opreme in tako omogoča izdelovalcem grafičnih kartic proste roke pri zasnovanju arhitekture (Slika 2.4).



Slika 2.4: Izvršilni model OpenGL senčilnikov[3].

Ker se visoko nivojska koda prevede znotraj OpenGL gonilnika (Slika 2.4), ki ga izda proizvajalec strojne opreme, je na proizvajalcu samem, da

zagotovi najboljšo zmogljivost. Izboljšave OpenGL prevajalnika lahko naredi proizvajalec strojne opreme z izdajo nove različice OpenGL gonilnika. Spremembe gonilnika ne vplivajo na sintakso senčilnika in zato ob novejši različici gonilnika ni potrebno spreminjati že obstoječih senčilnikov.

GLSL je del prosto dostopnega standarda, ki ga podpira veliko proizvajalcev programske opreme.

Visoko nivojski jezik se glede na to, katero vrsto senčilnika pišemo, zelo malo spreminja. Podpira modularno programiranje senčilnikov, saj ima prevajanje in povezovanje senčilnikov definirano kot dva koraka in s tem omogoči razvijalcem več svobode pri implementaciji zahtevnih algoritmov. Lahko ga definirajo kot velik in kompleksen senčilnik ali pa kot seznam senčilnikov, ki se neodvisno prevedejo in povežejo z objektom programa. Senčilniki so lahko izdelani s skupnim vmesnikom in so med seboj zamenljivi. Povezovalnik jih poveže in s tem naredi program.

GLSL in pripadajoča prevajalnik ter povezovalnik sta del OpenGL (Slika 2.4). Tako aplikacijam ni potrebno skrbeti za dodatne knjižnice za uspešno izvršitev. Prav tako so vse izboljšave izdane kot del OpenGL gonilnika[3].

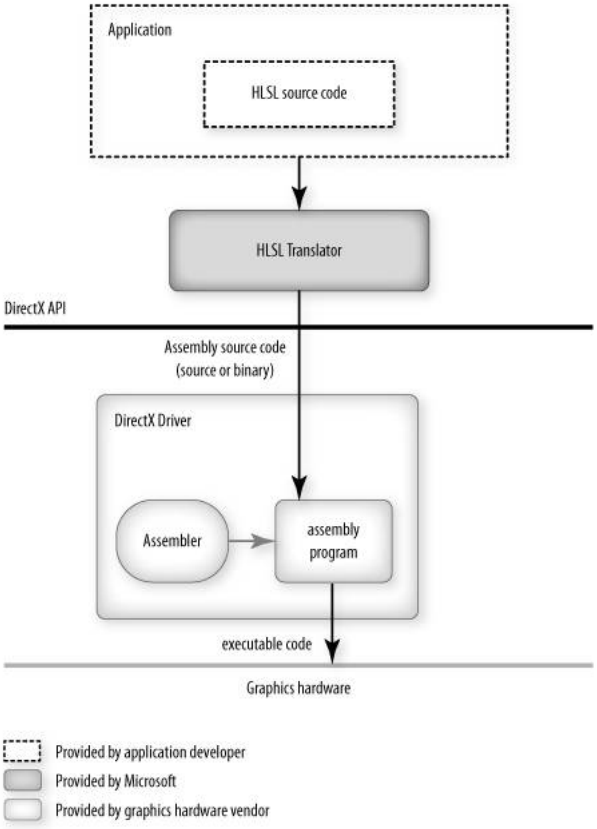
2.2.3 High Level Shader Language (HLSL)

HLSL ali High Level Shader Language je razvilo podjetje Microsoft in ga prvič predstavilo z izidom grafičnega vmesnika DirectX 9 leta 2002. V smislu sintakse in funkcionalnosti je HLSL zelo podoben OpenGL Shading Language.

Glavna razlika Med HLSL in GLSL je v izvršilnem okolju. HLSL prevajalnik se nahaja izven izvršilnega okolja grafičnega vmesnika DirectX, zato HLSL prevajalnik prevede izvirno kodo direktno v zbirnik. Različni nivoji funkcionalnosti so definirani za senčilnike na nivoju zbirnika in se razlikujejo po številki verzije, ki jo potrebujejo za izvedbo (Slika 2.5).

Prednost takega izvršilnega modela je, da se lahko senčilnik prevede v zbirnik že pred izvajanjem programa. Morda je potrebno, da se ta koda še enkrat prevede v zbirnik, ki ga uporablja računalnik, na katerem se izvaja

program. Ta prevod se opravi v času izvajanja (Slika 2.5). Pri GLSL pa je prevajalnik del OpenGL gonilnika in se zato prevede iz izvorne kode šele v času izvajanje programa. Prevajalnik za HLSL je podan s strani Microsofta, medtem ko prevajalnik za GLSL napiše vsak proizvajalec strojne opreme, kar mu omogoča večjo optimizacijo izvajanja senčilnika na uporabljeni grafični kartici.



Slika 2.5: Izvršilni model HLSL senčilnikov[3].

Microsoft zagotavlja tudi programsko opremo DirectX Effects Framework, ki omogoča razvijalcem organizacijo in izdajo senčilnikov, ki imajo enako funkcionalnost na strojni opremi z različnimi funkcionalnostmi.

Osnovni tipi so zelo podobni GLSL, z izjemo majhnih razlik v poimenovanju tipov. DirectX ima dva tipa več za števila s plavajočo vejico, in sicer s

polovično natančnostjo in dvojno natančnostjo. Ostala sintaksa in vgrajene funkcije so zelo podobne.

HLSL se razlikuje od GLSL tudi v tem, kako podaja uporabniško določene spremenljivke med različnimi vrstami senčilnikov. Pri GLSL samo dodamo uporabniškim spremenljivkam, ki se prenesejo med senčilniki, tipa *in* in *out* oziroma jih prenesemo preko že vgrajenih spremenljivk. HLSL pa omogoča prenos uporabniških spremenljivk preko že rezerviranih spremenljivk, in sicer *POSITION*, *COLOR[i]* in *TEXCOORD[i]*.

HLSL je bil narejen za grafični vmesnik DirectX podjetja Microsoft, medtem ko je bil GLSL narejen za grafični vmesnik OpenGL. Microsoft lahko sam spreminja DirectX in teče le na na njihovem operacijskem sistemu, medtem ko je OpenGL prosto dostopen in na mnogih platformah podprt standard, ki se spreminja počasneje, vendar pa ohranja združljivost za nazaj[3].

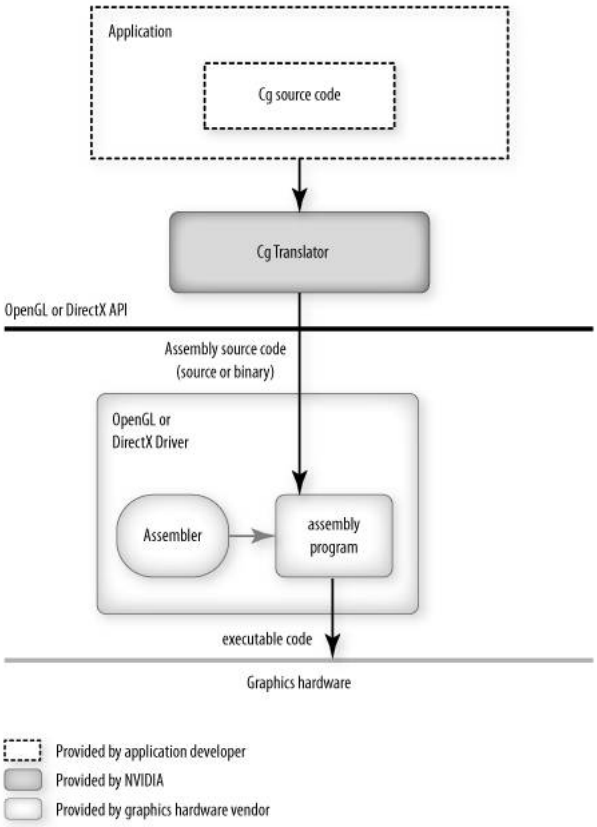
2.2.4 NVIDIA Cg

Cg je visoko nivojski programski jezik za senčilnike, ki ga je razvilo podjetje NVIDIA. Cg in HLSL ste bila skupaj razvita v podjetjih Microsoft in NVIDIA, zato sta si zelo podobna.

Velika prednost, ki jo ima Cg je, da lahko Cg prevajalnik generira ali HLSL senčilnik ali GLSL senčilnik, zato lahko senčilnike napisane v Cg jeziku uporabljamo tako z DirectX in OpenGL (Slika 2.6), vendar pa morata vmesnika imeti dostop do posebne knjižnice, ki se imenuje Cg Runtime library. Vmesni nivo, ki ga ustvari Cg, je lahko v veliko pomoč pri enostavnih senčilnikih, lahko pa je problematičen pri kompleksnejših senčilnikih, saj zakrije delovanje senčilnikov na nižjem nivoju.

V primerjavi z GLSL ima Cg enake prednosti in slabosti, kot jih ima HLSL.

NVIDIA zagotavlja tudi napredno ogrodje za razvoj senčilnikov, ki omogoča od aplikacije neodvisen razvoj senčilnikov in nam zagotavlja zagon samega senčilnika znotraj tega ogrodja.[3]



Slika 2.6: Izvršilni model Cg senčilnikov[3].

Poglavje 3

Izdelava in implementacija senčilnikov

3.1 Principi razvoja senčilnikov

Razvoj senčilnikov je razvoj programske opreme za specifično področje. Zato so tudi pri razvoju senčilnikov koristni principi in prakse, ki jih uporabljamo pri razvoju druge programske opreme.

Razvoja senčilnikov se torej lotimo po znanem zaporedju opravil. Začnemo z dizajnom, nadaljujemo z izdelavo in implementacijo, ki jo temeljito preizkusimo in zanjo napišemo dokumentacijo[3].

Osnovni princi razvoja senčilnikov[3]:

- Razumevanje problema;
- Postopno dodajanje kompleksnosti;
- Iterativni razvoj in testiranje;
- Preprostost;
- Modularnost.

3.2 Struktura senčilnika

Grafični program v večini primerov vsebuje dva senčilnika, in sicer en senčilnik oglišč ter en senčilnik fragmentov. Senčilnika za teselacijo in senčilnik za geometrijo sta neobvezna. Program lahko vsebuje več senčilnikov ene vrste, vendar lahko le eden vsebuje funkcijo *main()*.

Naslednji preprost primer izrazi temperaturo površine z barvo. Interval temperatur in barva sta podana kot parametra. Najprej se izvede senčilnik oglišč nad vsakem ogliščem posebej (Izsek kode 3.1).

```
// Spremenljivke tipa uniform spremeni aplikacija najvec enkrat
// na vsak primitiv.
uniform float CoolestTemp;
uniform float TempRange;

// Spremenljivke tipa in se ponavadi spremenijo za vsako oglisce
   .
in float VertexTemp;

// S spremenljivoe tipa out komuniciramo s sencilnikom
   fragmentov.
out float Temperature;

void main()
{
    // Izracunamo temperaturo, ki bo interpolirana za vsak
       fragment
    // posebej v intervalu [0.0, 1.0].
    Temperature = (VertexTemp - CoolestTemp) / TempRange;

        // Transformiramo polozaj oglisca iz koordinatnega
           sistema modela
        // v koordinatni sistem projekcije kamere.
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

Izsek kode 3.1: Primer senčilnika oglišč.

Za obdelavo oglišč sledi fiksen del cevovoda, ki poskrbi za sestavo primitivov ali pa se izvedeta senčilnika teselacije in/ali senčilnik geometrije. V tej stopnji se zagotovijo potrebni podatki, ki omogočijo rasterizaciji, da ustvari fragmente. Za vsak fragment se nato izvede senčilnik fragmentov (Izsek kode 3.2).

```
// Tip vec3 deklarira vektor s tremi spremenljivkami tipa float.
uniform vec3 CoolestColor;
uniform vec3 HottestColor;

// Spremenljivka Temperature sedaj vsebuje ze interpolirano
// vrednost za trenutni fragment, ki smo jo nastavili v
// sencilniku oglisc.
in float Temperature;

void main()
{
        // Barvo dobimo tako, da vzamemo srednjo vrednost med
           dvema
        // barvama z vgrajeno funkcijo mix().
    vec3 color = mix(CoolestColor, HottestColor, Temperature);

    // Naredimo vektor s stirimi komponentami tako, da dodamo
    // prosojnost.
    gl_FragColor = vec4(color, 1.0);
}
```

Izsek kode 3.2: Primer senčilnika fragmentov.

Vsak senčilnik lahko prejme uporabniško določene spremenljivke preko spremenljivk, deklariranih s predpono *uniform*. Senčilnik oglišč lahko prejme dodatne podatke za vsako oglišče posebej preko uporabniško določenih spremenljivk s predpono *in*. Senčilnik oglišč nato preko uporabniško določenih spremenljivk s predpono *out* pošlje spremenljivke naslednjemu senčilniku, ki jih prebere s spremenljivko z enakim imenom in predpono *in*.

Senčilniki lahko komunicirajo s fiksnim grafičnim cevovodom preko vgrajenih spremenljivk s predpono *gl_*. V primeru s pisanjem v spremenljivko

gl_Position povemo fiksnemu grafičnemu cevovodu, kje se nahaja transformirano oglišče. S pisanjem v spremenljivko *gl_FragColor* pa cevovodu povemo, s katero barvo naj pobarva določen fragment. Spremenljivke s predpono *gl_* so sicer v zadnjih različicah GLSL postale zastarele (angl. depricated). Zagotoviti jih mora uporabnik sam z uporabo uporabniško določenih spremenljivk.

Senčilniki se lahko izvedejo večkrat za posamezen primitiv, oglišče ali fragment. Veliko prehodov enega senčilnika se lahko izvede paralelno, vendar posamezni prehodi med seboj niso povezani oziroma so med seboj neodvisni.

GLSL ima veliko vgrajenih tipov spremenljivk, s katerimi olajša pisanje grafičnih operacij. Podpira spremenljivke v obliki skalarjev, vektorjev, matrik, seznamov, strukture in druge specifične tipe. Spremenljivkam moramo nujno podati tip, saj nimamo v naprej določenih privzetih tipov. Prav tako lahko spremenljivke definiramo, ko jih potrebujemo.

Tipi skalarjev so: *float*, ki predstavlja število, zapisano s plavajočo vejico z enojno natančnostjo, *int*, ki predstavlja cela števila, in *boolean*, ki predstavlja Boolovo število.

Vektorji lahko vsebujejo 2, 3 ali 4 komponente, katerih tip so podprti skalarji. GLSL tudi podpira posebne operacije nad vektorji in omogoča preprost dostop do posameznih komponent vektorja.

Matrike so velikosti 2x2, 3x3 ali 4x4, katerih komponente so tipa *float*. Omogočajo enostaven dostop do posameznih komponent.

Sampler je poseben tip spremenljivke, ki skrbi za uporabo tekstur. Vrednost lahko dobi podano preko glavnega programa.

GLSL omogoča programerju definiranje lastnih struktur ter združi več vgrajenih tipov spremenljivk v zaključeno celoto oziroma kot nov tip spremenljivke z več komponentami.

Seznami vgrajenih tipov spremenljivk se deklarirajo le z oglatimi oklepaji, v katerih podamo velikost seznama, saj GLSL ne podpira kazalcev na seznam. Vse spremenljivke in seznami se prenesejo po vrednosti, pripadajoči funkciji. Če deklariramo seznam brez določene velikosti, bo velikost seznama enaka najvišjemu indeksu, na katerega je bila zapisana vrednost.

Spremenljivke so lahko inicializirane ob deklaraciji. Konstante so predstavljene s predpono *const* in morajo biti inicializirane ob deklaraciji. Spremenljivke s predpono *in*, *out* in *uniform* ne smejo biti inicializirane ob deklaraciji, saj dobijo vrednost preko glavnega programa. Za deklariranje spremenljivk, ki imajo več komponent, uporabimo konstruktorje. Z njimi opravljamo tudi konverzije med tipi spremenljivk.

Pri uporabi senčilnikov se pri vsaki zahtevi vedno najprej zažene funkcija *main()*, še pred tem pa se inicilizirajo globalne spremenljivke, ki so podane izven funkcije *main()*.

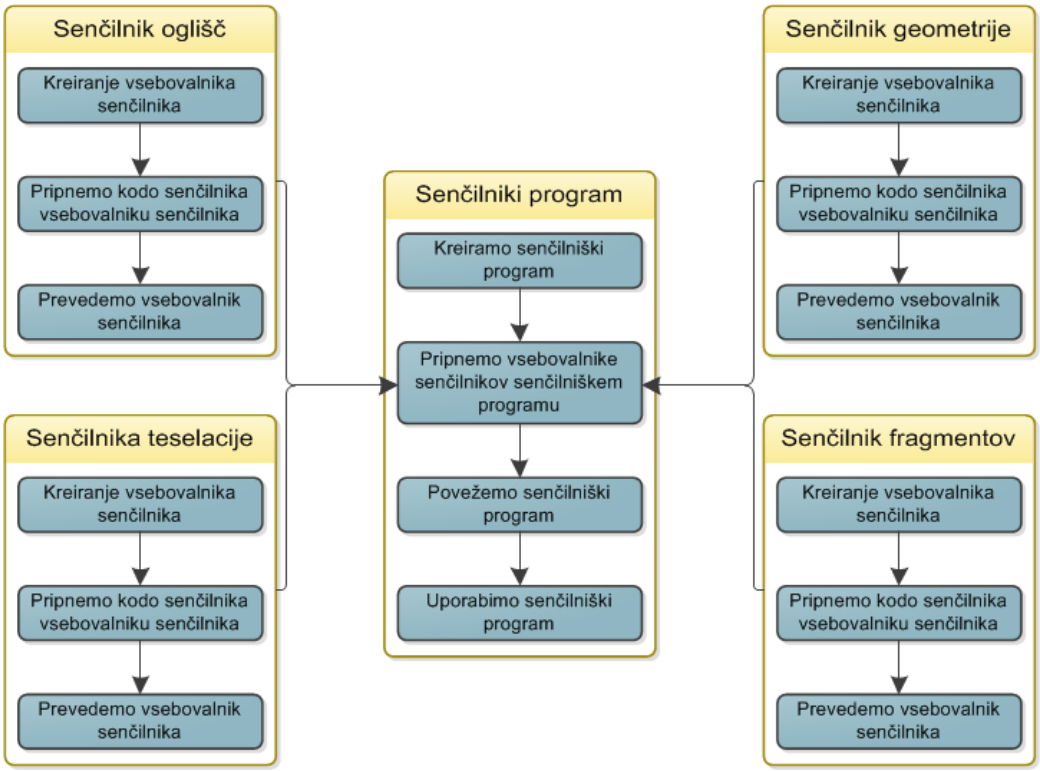
Za izvajanje zank lahko uporabimo stavek *for*, *while* in *do-while*. Za prekinitev ali preskok enega obhoda zanke uporabimo besedi *break* in *continue*. Za izbiro pa uporabimo besed *if* in *if-else*.

V senčilnikih lahko definiramo tudi lastne funkcije, ki jih lahko ponovno definiramo z drugačnim tipom parametrov ali njihovim številom. Senčilniki tudi ne podpirajo rekurzivnih klicev funkcij[3].

3.3 Uporaba senčilnika GLSL z LWJGL knjižnico

Potek pisanja programa z vmesnikom OpenGL je tak kot pisanje programa v programskem jeziku C. Senčilnik si predstavljamo kot modul v jeziku C in zato je potrebno vsakega posebej prevesti. Prevedene senčilnike moramo nato še povezati v program, ki ga potem uporabljamo med izvajanjem (Slika 3.1).

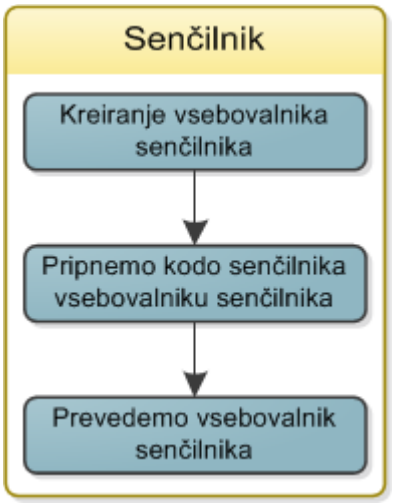
LWJGL knjižnica (angl. Lightweight Java Game Library), ki smo jo uporabljali pri implementaciji, zagotavlja grafično knjižnico OpenGL za programski jezik Java in bazira na odprtokodni licenci. Knjižnica je stabilna in tudi primerna za razvoj večjih in kompleksnih projektov[10].



Slika 3.1: Potrebni koraki za kreiranje posameznih senčilnikov in senčilniškega programa. Senčilnika teselacije in senčilnik geometrije so opcijski [7].

3.3.1 Kreiranje senčilnika

V prvem koraku (Slika 3.2) ustvarimo objekt, ki se obnaša kot vsebovalnik senčilnika. S konstanto mu povemo, za katero vrsto senčilnika gre. Izvorno kodo preberemo iz datoteke in jo shranimo v niz. V naslednjem koraku pripnemo niz, ki vsebuje izvorno kodo, v vsebovalnik senčilnika. Na koncu senčilnik še prevedemo[7, 9]. Postopek je prikazan na sliki 3.2 in programska koda je prikazana v izseku kode 3.3.



Slika 3.2: Potrebni koraki za kreiranje posameznega senčilnika[7].

```
int vertexShader = glCreateShader(GL_VERTEX_SHADER);

StringBuilder vertexShaderSource = new StringBuilder();
try{
        BufferedReader reader = new BufferedReader(new
            FileReader(vertexShaderLocation));
        String line;
        while((line = reader.readLine()) != null){
                vertexShaderSource.append(line).append('\n');
```

```
            }
            reader.close();
}
catch(IOException e){
            System.err.println("Vertex shader wasn't loaded properly
                .");
            Display.destroy();
            System.exit(1);
}

glShaderSource(vertexShader, vertexShaderSource);

glCompileShader(vertexShader);
if (glGetShader(vertexShader, GL_COMPILE_STATUS) == GL_FALSE) {
            System.err.println("Vertex shader wasn't able to be
                compiled correctly. Error log:");
            System.err.println(glGetShaderInfoLog(vertexShader,
                1024));
}
```

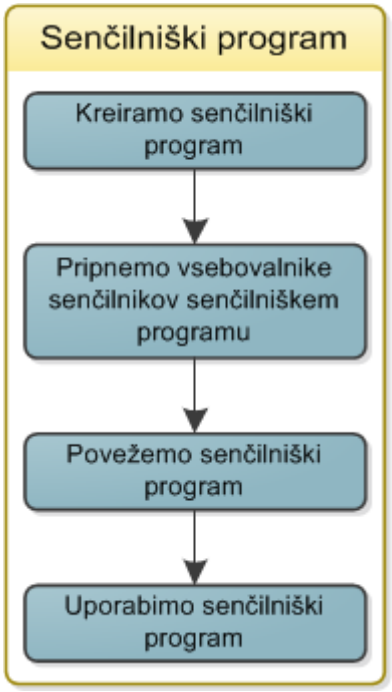
Izsek kode 3.3: Programska koda za kreiranje senčilnika.

3.3.2 Kreiranje senčilnega programa

V prvem koraku (Slika 3.3) ustvarimo objekt, ki se obnaša kot vsebovalnik senčilnega programa. Ustvarimo lahko toliko senčilnih programov, kolikor želimo. Med izrisovanjem grafike na zaslon lahko preprosto preklapljamo pod posameznimi senčilnimi programi. Preklapljamo lahko tudi znotraj izrisovanje ene slike, tako da uporabimo en senčilni program za izrisovanje določenege dela scene in drugega za drug del scene.

V naslednjem koraku pripnemo posamezne senčilnike na senčilni program. Ni potrebno, da je senčilnik preveden ali da ima že pripeto izvorno kodo.

Senčilnemu programu lahko pripnemo mnogo senčilnikov iste vrste, vendar ima lahko le eden izmed senčilnikov iste vrste funkcijo *main()*.



Slika 3.3: Potrebni koraki za kreiranje posameznega senčilniškega programa[7].

Prav tako lahko en senčilnik pripnemo več senčilnim programom in ga lahko uporabljamo v več senčilnih programih hkrati.

V zadnjem koraku moramo še povezati senčilni program. Pred izvedbo tega morajo biti vsi senčilniki, ki so pripreti na senčilni program, že prevedeni.

Ko smo povezali senčilni program, lahko spreminjamo posamezne senčilnike, vendar če želimo, da se spremembe poznajo, moramo ponovno zagnati metodo za povezovanje senčilnega programa. Po uspešnem povezovanju se nato senčilni program naloži v pomnilnik.

Vsak program ima svoj vsebovalnik. Tako imamo lahko pripravljenih in povezanih, kolikor želimo mnogo senčilnih programov. Med njimi lahko enostavno preklapljamo, vendar je naenkrat lahko aktiven samo en senčilni program[7, 9]. Postopek je prikazan na sliki 3.3 in programska koda je prikazana v izseku kode 3.4.

```
int shaderProgram = glCreateProgram();

glAttachShader(shaderProgram, vertexShader);

glLinkProgram(shaderProgram);

glValidateProgram(shaderProgram);
```

Izsek kode 3.4: Programska koda za kreiranje senčilnega programa.

3.4 Komunikacija med glavnim programom in senčilnikom

Glavni program lahko komunicira s senčilnikom preko *unifrom* spremenljivk, ki služijo kot globalne, ne pogosto se spreminjajoče spremenljivke, ki so dostopne v vsaki vrsti senčilnikov.

Da jih lahko uporabimo, moramo po njihovi lokaciji povprašati senčilniški program, potem na dano lokacijo shranimo podatke, ki jih dobi senčilnik. Uniformne spremenljivke imajo veliko tipov, prav tako lahko sami definiramo

kompleksnejše tipe. Pri podajanju spremenljivke senčilniškemu programu moramo povedati, za kakšen tip spremenljivke gre[7] (Izsek kode 3.5).

```
int uniform = glGetUniformLocation(shaderProgram, "
    NameOfTheUniform");

glUniform1i(uniform, 5);
```

Izsek kode 3.5: Pridobitev lokacije *uniform* spremenljivke in shranjevanje vrednosti vanjo.

Na podoben način lahko uporabljamo tudi *in* spremenljivke oziroma atribute, ki se lahko spreminjajo dosti bolj pogosto, vendar do njih lahko odstopamo samo v senčilniku oglišč. Najprej povprašamo senčilniški program po njihovi lokaciji, ki jo potem uporabimo za pošiljanje podatkov senčilniku (Izsek kode 3.6). Atributi imajo veliko tipov, prav tako lahko sami definiramo kompleksnejše tipe. Pri podajanju atributa senčilniškemu programu moramo povedati, za kakšen tip spremenljivke gre[7].

```
int attribute = glGetAttribLocation(shaderProgram, "
    NameOfTheAttribute");

glVertexAttrib3f(attribute, 1.0f, 1.0f, 1.0f);
```

Izsek kode 3.6: Pridobitev lokacije *in* spremenljivke in shranjevanje vrednosti vanjo.

Pri uporabi pomnilnika oglišč uporabimo kazalec na seznam vrednosti, preko katerega samodejno iterira grafični cevovod. Pri definiranju pomnilniška oglišč določimo kazalcu številko, ki jo pri izrisu omogočimo, ter za vse ostalo poskrbi grafični cevovod. Pomnilniku oglišč moramo podati položaje oglišč in povedati, katera oglišča tvorijo posamezen primitiv[7] (Izsek kode 3.7).

```
vao = glGenVertexArrays();
glBindVertexArray(vao);

vbo = glGenBuffers();
glBindBuffer(GL_ARRAY_BUFFER, vbo);
glBufferData(GL_ARRAY_BUFFER, verticesBuffer, GL_STATIC_DRAW);
```

```
glVertexAttribPointer(0, 3, GL_FLOAT, false, 0, 0);

glBindBuffer(GL_ARRAY_BUFFER, 0);

glBindVertexArray(0);

vboi = glGenBuffers();
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, vboi);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, indicesBuffer,
    GL_STATIC_DRAW);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);

glBindVertexArray(vao);

glEnableVertexAttribArray(0);

glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, vboi);
glDrawElements(GL_TRIANGLES, indices.length, GL_UNSIGNED_BYTE,
    0);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);

glDisableVertexAttribArray(0);
glBindVertexArray(0);
```

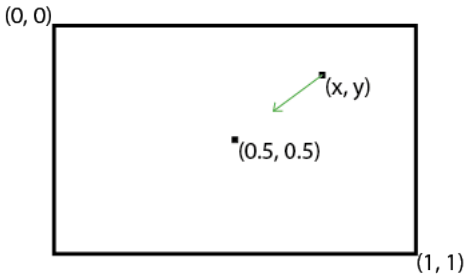
Izsek kode 3.7: Ustvarimo seznam pomnilnikov oglišč ter mu podamo oglišča in povemo katera oglišča sestavljajo primitive. Na koncu še določimo na kateri številki kazalca se nahajaj pomnilnik oglišč.

3.5 Senčilnik za krožno zameglitev celotne scene

Senčilnik za krožno zameglitev celotne scene spada v skupino tako imenovanih *post-processing* senčilnikov, saj se izvaja na že izrisani sceni. Posledica tega je, da porabimo za izris ene slike na zaslon dva prehoda skozi grafični cevovod. V prvem prehodu izrišemo celotno sceno in jo shranimo v teksturo. V drugem prehodu izrišemo kvadrat skozi cel zaslon in nanj pripnemo teksturo,

ki smo jo dobili pri prvem prehodu in nad njo izvedli krožno zameglitev.[11]

Senčilnik zamegli sliko v smeri proti središču zaslona (Slika 3.5). V našem primeru uporabljamo koordinate za nanos teksture na kvadrat oziroma *uv-koordinate*. Nanos teksture začnemo na koordinatah (0, 0) in končamo na koordinatah (1, 1) (Slika 3.4). Ko izrisujemo vsak fragment posebej, poznamo njegove *uv-koordinate*, ki nam jih poda senčilnik oglišč, ki opravi interpolacijo *uv-koordinat* med *uv-koordinatam* oglišč, med katerimi se nahaja fragment. Trenutne *uv-koordinate* fragmenta odštejemo od središča zaslona, ki se nahaja na *uv-koordinatah* (0.5, 0.5), in tako dobimo vektor od fragmenta do središča zaslona (Slika 3.4). Dolžina izračunanega vektorja nam pove oddaljenost fragmenta od središča in jo uporabimo kot utež za meglenje teksture; bolj kot je fragment oddaljen od središča, bolj bo zameglen.[11]



Slika 3.4: Prikaz *uv-koordinat* teksture na zaslonu in vektorja za poljubno točko, ki kaže proti središču zaslona.

Najprej moramo ustvariti teksturo, v katero bomo izrisali sceno brez zameglitve. Ustvarimo teksturo velikosti našega zaslona in nastavimo, da shrani barvo za vsak fragment posebej s konstanto GL_RGB[15]. Nato nastavimo funkciji, ki poskrbita za vrednost posameznega fragmenta, ali teksturo nanašamo na ploskev, večjo od teksture (GL_TEXTURE_MIN_FILTER)[15], ali jo nanašamo na ploskev, manjšo od teksture (GL_TEXTURE_MAG_FILTER)[15], da izbereta teksturni fragment, ki je najbližje centru fragmenta, za katerega vrednost iščemo (GL_NEAREST)[15]. Da ne pride do napak pri izrisovanju robnih elementov teksture, nastavimo še parametra za ovitje teksture v interval znotraj $[\frac{1}{2N}, 1 - \frac{1}{2N}]$, kjer je N velikost teksture po

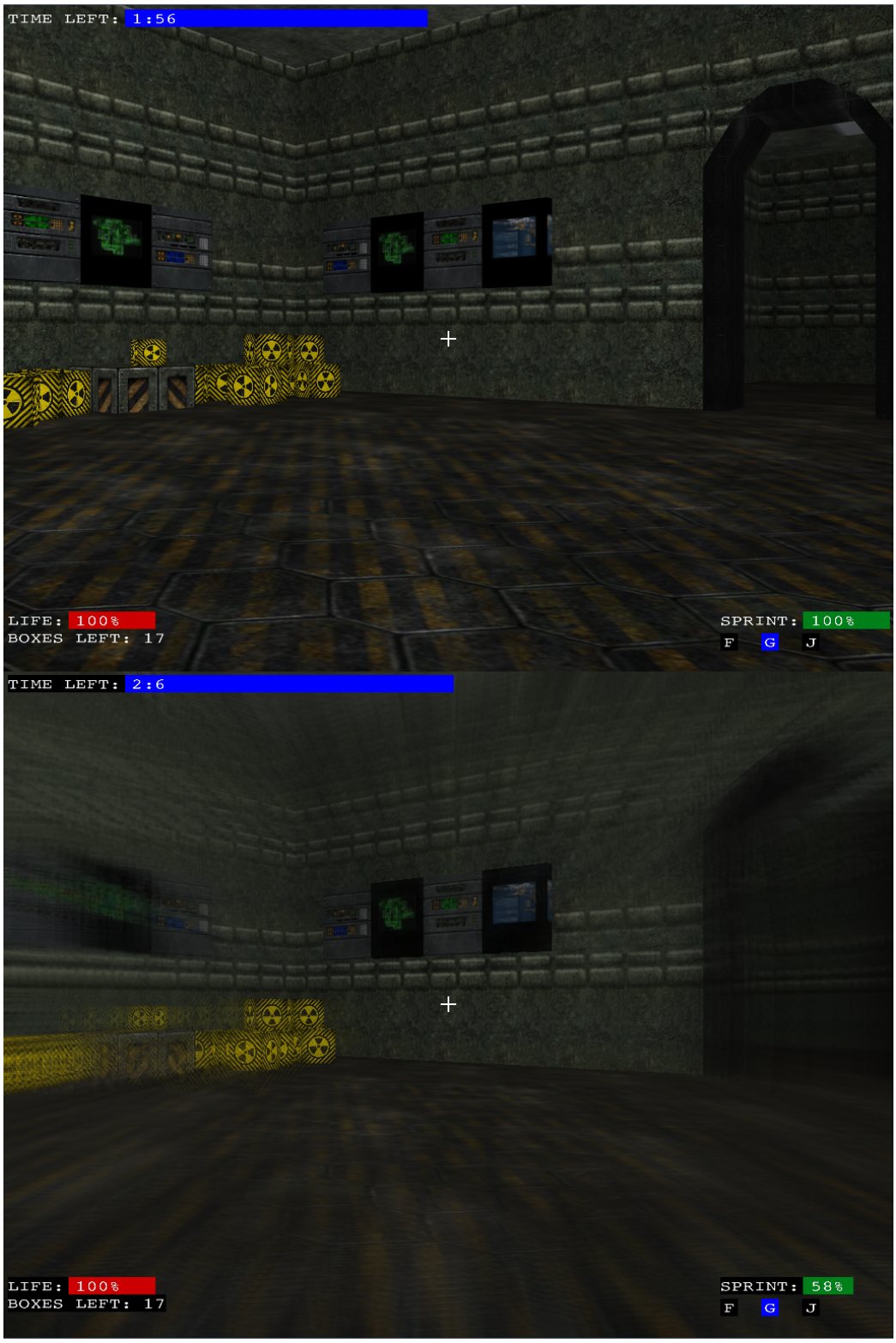
posamezni osi (GL_CLAMP_TO_EDGE)[15]. Nastavimo ju za vsako koordinato posebej (GL_TEXTURE_WRAP_S, GL_TEXTURE_WRAP_T)[15, 12] (Izsek kode 3.8).

```
blurTextureID = glGenTextures();
glBindTexture(GL_TEXTURE_2D, blurTextureID);

glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, 1024, 768, 0, GL_RGB,
    GL_UNSIGNED_BYTE, (ByteBuffer)null);

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST
    );
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST
    );
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
    GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
    GL_CLAMP_TO_EDGE);

glBindTexture(GL_TEXTURE_2D, 0);
```

Izsek kode 3.8: Ustvarimo teksturo za shranjevanje celotne scene.

Teksturo nato pripnemo na pomnilnik slike (GL_FRAMEBUFFER)[15], ki poskrbi da se scena izriše in shrani v teksturo. Pri definiciji pomnilnika slike povemo, kateri del izrisane scene naj se shrani v teksturo. V našem primeru shranimo barvo posameznih fragmentov teksture (GL_COLOR_ATTACHMENT0)[15].

Za pravilni izris scene v teksturo moramo ustvariti še izrisovalni pomnilnik (GL_RENDERBUFFER), ki ga nastavimo, da vsebuje informacijo o globini scene (GL_DEPTH_COMPONENT)[15] in nam tako omogoči test globine posameznih primitivov. Na tak način se scena pravilno izriše in shrani v teksturo[12] (Izsek kode 3.9).



Slika 3.5: Scena brez meglenja zgoraj in s krožnim meglenjem spodaj.

```
blurFBOID = glGenFramebuffers();
blurRBOID = glGenRenderbuffers();

glBindFramebuffer(GL_FRAMEBUFFER, blurFBOID);
glBindRenderbuffer(GL_RENDERBUFFER, blurRBOID);

glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT, 1024,
    768);

glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
    GL_TEXTURE_2D, blurTextureID, 0);
glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
    GL_RENDERBUFFER, blurRBOID);

glDrawBuffer(GL_COLOR_ATTACHMENT0);

glBindRenderbuffer(GL_RENDERBUFFER, 0);
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

Izsek kode 3.9: Ustvarimo pomnilnik slike za shranjevanje scene v teksturo.

V prvem prehodu grafičnega cevovoda shranimo izris scene preprosto tako, da pred normalnim izrisom scene vključimo inicializiran pomnilnik slik in ga po izrisu scene izključimo (Izsek kode 3.10).

```
glBindFramebuffer(GL_FRAMEBUFFER, blurFBOID);

// Izris celotne scene.

glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

Izsek kode 3.10: Uporaba pomnilnika slike za shranjevanje scene v teksturo.

V drugem prehodu grafičnega cevovoda pa aktiviramo naš senčilnik, mu podamo teksturo, v katero smo v prejšnjem prehodu shranili sceno, in izrišemo celozaslonski kvadrat, na kateremu uporabimo našo zamegleno teksturo (Izsek kode 3.11).

```
glMatrixMode(GL_MODELVIEW);
glPushMatrix();
glLoadIdentity();
glMatrixMode(GL_PROJECTION);
glPushMatrix();
glLoadIdentity();

glUseProgram(blurShaderProgram);

glActiveTexture(GL_TEXTURE5);
glBindTexture(GL_TEXTURE_2D, blurTextureID);
glUniform1i(blurTextureUniform, 5);

glBegin(GL_QUADS);
        glNormal3f( 0.0f, 0.0f, 1.0f);
        glVertex2f(-1.0f, -1.0f);
        glVertex2f( 1.0f, -1.0f);
        glVertex2f( 1.0f, 1.0f);
        glVertex2f(-1.0f, 1.0f);
glEnd();

glBindTexture(GL_TEXTURE_2D, 0);

glUseProgram(0);

glPopMatrix();
glMatrixMode(GL_MODELVIEW);
glPopMatrix();
```

Izsek kode 3.11: Koda za izris krožno zameglene teksture scene čez cel zaslon.

Senčilnik oglišč spremeni vhodne koordinate oglišč, tako da lahko posamezna koordinata zavzame vrednosti -1, 0 ali 1 in se zapišejo v vgrajeno izhodno spremenljivko za položaj oglišč *gl_Position*. Senčilnik nato iz vhodnih koordinat izračuna *uv-koordinate* za našo zamegleno teksturo, in sicer tako, da so *uv-koordinate* oglišč kvadrata (0, 0), (1, 0), (0, 1) in (1, 1) (Slika 3.4) ter jih pošlje naprej senčilniku fragmentov[11] (Izsek kode 3.12).

```
out vec2  uv;

void main(void)
{
   gl_Position = vec4( gl_Vertex.xy, 0.0, 1.0 );
   gl_Position = sign( gl_Position );

   uv = (vec2( gl_Position.x, gl_Position.y ) + vec2( 1.0 ) ) /
      vec2( 2.0 );
}
```

Izsek kode 3.12: Senčilnik oglišč za krožno meglenje celotne scene.

V senčilniku fragmentov definiramo seznam koeficientov s katerimi izberemo deset vzorcev v majhni okolici *uv-koordinat*, ki jih prejmemo iz senčilnika fragmentov. Nato izračunamo vektor od *uv-koordinat* do središča zaslona, ki se nahaja na koordinatah (0.5, 0.5), izračunamo njegovo dolžino in ga normaliziramo. V naslednjem koraku uporabimo vgrajeno funkcijo *texture2D()*, ki nam poišče vrednost teksture na podanih koordinatah in s tem dobimo vrednost teksture za trenutni *uv-koordinati*. Nato v zanki ponovno uporabimo funkcijo *texture2D()*, še na *uv-koordinatah* prestavljenih s koeficienti iz seznama, ki smo ga definirali na začetku in vse vrednosti seštejemo. Prištejemo še vrednost na dejanskih *uv-koordinatah*. Vsoto delimo s številom 11, ker smo dejansko vzeli 11 vzorcev. Otežimo oddaljenost *uv-koordinat* od središča in jo omejimo na intervalu med 0 in 1. S tem dosežemo, da so *uv-koordinate*, ki so bolj oddaljene od središča, bolj zameglene. Končno barvo fragmenta dobimo tako, da uporabim vgrajeno funkcijo *mix()*, ki nam interpolira dve barvi med seboj s podano utežjo. Definirani imamo še dve konstanti, s katerima lahko prilagajamo, koliko so vzorci oddaljeni od *uv-koordinat* in koliko otežimo oddaljenost *uv-koordinat* od središča[11] (Izsek kode 3.13).

```
uniform sampler2D tex;

in vec2 uv;
```

```
const float sampleDist = 1.0;
const float sampleStrength = 2.2;

void main(void)
{
        float samples[10] = float[](
                -0.08, -0.05, -0.03,
                -0.02, -0.01,  0.01,
                 0.02,  0.03,  0.05,
                 0.08
                );

    vec2 dir = 0.5 - uv;

    float dist = sqrt(dir.x*dir.x + dir.y*dir.y);

    dir = dir/dist;

    vec4 color = texture2D(tex,uv);

    vec4 sum = color;

    for (int i = 0; i < 10; i++)
    {
      sum += texture2D( tex, uv + dir * samples[i] * sampleDist
          );
    }

    sum *= 1.0/11.0;

    float t = dist * sampleStrength;
    t = clamp( t ,0.0,1.0);

    gl_FragColor = mix( color, sum, t );
}
```

Izsek kode 3.13: Senčilnik fragmentov za krožno meglenje celotne scene.

Pri implementaciji je izziv predstavljala predvsem nastavitev teksture in

pomnilnika slik, saj je bil v literaturi na temo meglenja scene izris scene v teksturo samo omenjen, ni pa bilo razloženo, kako to narediti. Postopek, kako izrisati sceno v teksturo, smo morali najti drugje. Najprej smo se morali seznaniti z delovanjem pomnilnika slik, kako lahko vanj shranimo sceno kot teksturo. Podobni primeri so bili v celoti napisani v drugih jezikih in je bilo potrebno preučiti uporabo enakih funkcij z LWJGL knjižnico. Najprej smo definirali teksturo in pomnilnik slik in vanj izrisali sceno, vendar je bila scena narobe shranjena v teksturo. Potrebno je bilo še vključiti globinsko informacijo v sceni za pravilen izris, zato smo pomnilniku slike pripeli še izrisovalni pomnilnik, ki je poskrbel za globinsko informacijo. Tako se je scena v teksturo shranila pravilno. Nato smo napisali senčilnika in ju vključili v program. Na koncu smo še prilagajali parametra (sampleDist in sampleStrength), da smo dobili želeni rezultat.

3.6 Senčilnik sistema delcev

Sistem delcev se razlikuje od tradicionalnega izrisovanja ploskev na treh pomembnih področjih:

- Objekt ni definiran s poligoni ali krivimi ploskvami, temveč je predstavljen z množico delcev, ki so ponavadi geometrični primitivi, ki definirajo volumen objekta.

- Sistem delcev je dinamičen in ne statičen, kot je to pri objektih predstavljenih s ploskvami. Delci, ki sestavljajo objekt, najprej nastanejo, se razvijajo in nato umrejo. Med svojim življenjem delci lahko spreminjajo svoj položaj in obliko.

- Objekti, ki so definirani s sistemom delcev, niso v celoti specificirani. Za sistem delcev so postavljeni začetni pogoji, pravila za rojstvo, razvoj in smrt. Stohastični procesi pa vplivajo na vse tri stopnje življenj delcev, zato sta oblika in izgled objekta nedeterministična.

Da poenostavimo izrisovanje in računsko zahtevnost sistema delcev, predpostavimo, da med delci na prihaja do trkov, da delci ne odbijajo svetlobe, temveč jo oddajajo in da delci ne mečejo sence na druge delce v sistemu.

Posamezni delci imajo atribute, kot so položaj, barva, presojnost, hitrost, velikost, oblika in življenjska doba. Pri izrisu sistema delcev se uporabijo atributi vsakega delca in globalne spremenljivke v določeni funkciji, ki nam pove spremembo položaja in izgleda vsakega delca za vsak posameznem izris.

Prednost sistema delcev je, da lahko ustvarimo kompleksne, prilagodljive in dinamične efekte zelo preprosto. Prav tako opravimo del računanja na sami grafični kartici in ne na centralni procesni enoti, kar nam lahko prinese velike pridobitve v hitrosti izrisovanja.

Za naš primer smo naredili senčilnik, ki poskrbi za nastanek velikega števila majhnih delcev različnih barv ter jih izstreli stran od izvora v določeni smeri. Vsi delci ne nastanejo naenkrat, vendar nastajajo enakomerno, dokler niso vsi nastali. Začetne hitrosti so naključne znotraj intervala, zato imajo delci približno določeno smer navzgor in stran od izvora. Ko so delci nastali, nanje začne vplivati gravitacija, ki jo simuliramo z uporabo funkcije[3] (Slika 3.6).

Za vsak delec definiramo začetni položaj, naključno izbrano barvo, naključno izbrano hitrost znotraj omejitev in naključno izbran začetni čas. Ustvarimo dvodimenzionalno mrežo delcev, ki ji definiramo dolžino in širino ter s tem določimo število delcev. Določimo tudi položaj mreže v sceni ter začetne hitrosti v vse tri smeri. Začetni položaji delcev imajo enako z koordinato, medtem ko sta x in y koordinati določeni naključno znotraj intervala. Barve izberemo v intervalu med 0.5 in 1.0; s tem dosežemo žive barve. Vektorjem hitrosti določimo naključne vrednosti, s čimer omogočimo delcem, da potujejo navzgor. Vektor hitrosti v smeri y pomnožimo z 10, vektorjema hitrosti v smeri x in z pa prištejemo 3 in s tem omogočimo, da delci potujejo stran od izvora. Na koncu še naključno izberemo začetni čas posameznega delca na intervalu med 1 in 10[3] (Izsek kode 3.14).

Slika 3.6: Prikaz sistema delcev v različnih časih (t(zgoraj levo) = 0.1 s, t(zgoraj desno) = 0.5 s, t(spodaj levo) = 1.0 s, t(spodaj desno) = 1.5 s).

Posamezne atribute shranimo v pomnilnik oglišč, ki jih nato povežemo v pomnilniški sezname pomnilnikov oglišč[10] (Izsek kode 3.14).

```
public void createPoints(int w, int h, float x , float y, float
    z, float speedX, float speedY, float speedZ){
        float[] verts = new float[h * w * 3];
        float[] colors = new float[h * w * 3];
        float[] velocities = new float[h * w * 3];
        float[] startTimes = new float[h * w];

        int vertsP = 0;
        int colorsP = 0;
        int velocitiesP = 0;
        int startTimesP = 0;

        arrayHeight = h;
        arrayWidth = w;

        for(float i = 0.5f / w - 0.5f; i < 0.5f; i = i + 1.0f /
            w){
                for(float j = 0.5f / h - 0.5f; j < 0.5f; j = j +
                    1.0f / h){
                        verts[vertsP] = i + x;
                        verts[vertsP + 1] = j + y;
                        verts[vertsP + 2] = 0.0f + z;
                        vertsP += 3;

                        colors[colorsP] = (float)Math.random() *
                            0.5f + 0.5f;
                        colors[colorsP + 1] = (float)Math.random
                            () * 0.5f + 0.5f;
                        colors[colorsP + 2] = (float)Math.random
                            () * 0.5f + 0.5f;
                        colorsP += 3;

                        velocities[velocitiesP] = (float)Math.
                            random() + speedX;
```

```
                    velocities[velocitiesP + 1] = (float)
                        Math.random() * speedY;
                    velocities[velocitiesP + 2] = (float)
                        Math.random() + speedZ;
                    velocitiesP += 3;

                    startTimes[startTimesP] = (float)Math.
                        random() * 10.0f;
                    startTimesP += 1;
                }
            }

            FloatBuffer verticesBuffer = asFloatBuffer(verts);
            FloatBuffer colorsBuffer = asFloatBuffer(colors);
            FloatBuffer velocitiesBuffer = asFloatBuffer(velocities)
                ;
            FloatBuffer startTimesBuffer = asFloatBuffer(startTimes)
                ;

            vao = glGenVertexArrays();
            glBindVertexArray(vao);

            vbo = glGenBuffers();
            glBindBuffer(GL_ARRAY_BUFFER, vbo);
            glBufferData(GL_ARRAY_BUFFER, verticesBuffer,
                GL_STATIC_DRAW);
            glVertexAttribPointer(0, 3, GL_FLOAT, false, 0, 0);
            glBindBuffer(GL_ARRAY_BUFFER, 0);

            vboc = glGenBuffers();
            glBindBuffer(GL_ARRAY_BUFFER, vboc);
            glBufferData(GL_ARRAY_BUFFER, colorsBuffer,
                GL_STATIC_DRAW);
            glVertexAttribPointer(1, 3, GL_FLOAT, false, 0, 0);
            glBindBuffer(GL_ARRAY_BUFFER, 0);

            vbov = glGenBuffers();
            glBindBuffer(GL_ARRAY_BUFFER, vbov);
```

```
        glBufferData(GL_ARRAY_BUFFER, velocitiesBuffer,
            GL_STATIC_DRAW);
        glVertexAttribPointer(2, 3, GL_FLOAT, false, 0, 0);
        glBindBuffer(GL_ARRAY_BUFFER, 0);

        vbos = glGenBuffers();
        glBindBuffer(GL_ARRAY_BUFFER, vbos);
        glBufferData(GL_ARRAY_BUFFER, startTimesBuffer,
            GL_STATIC_DRAW);
        glVertexAttribPointer(3, 1, GL_FLOAT, false, 0, 0);
        glBindBuffer(GL_ARRAY_BUFFER, 0);

        glBindVertexArray(0);
}
```

Izsek kode 3.14: Za generiranje sistema delcev ustvarimo pomnilnike oglišč ter jih povežemo v seznam pomnilnikov.

Pred izrisom omogočimo uporabo transparentnosti. S tem dosežemo, da so delci, ki so še v začetnem stanju, nevidni. Za izris seznama pomnilnikov oglišč le tega najprej omogočimo in nato omogočimo še številke atributov, na katere smo pripeli posamezne pomnilnike oglišč[10]. Nato izrišemo sistem delcev kot točke[3] (GL_POINTS) (Izsek kode 3.15).

```
public void drawPoints(){

        glEnable(GL_BLEND);
        glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

        glBindVertexArray(vao);
        glEnableVertexAttribArray(0);
        glEnableVertexAttribArray(1);
        glEnableVertexAttribArray(2);
        glEnableVertexAttribArray(3);

        glDrawArrays(GL_POINTS, 0, arrayHeight * arrayWidth);

        glDisableVertexAttribArray(0);
```

```
        glDisableVertexAttribArray(1);
        glDisableVertexAttribArray(2);
        glDisableVertexAttribArray(3);
        glBindVertexArray(0);
}
```

Izsek kode 3.15: Izris sistema delcev.

Pri izrisu sistema delcev moramo najprej omogočiti senčilniški program za sistem delcev. Senčilniku moramo zagotoviti trenutni čas, v katerem se nahaja sistem delcev, matriko za preslikavo oglišč iz koordinatnega sistema modela v koordinatni sistem pogleda in matriko za preslikavo oglišč iz koordinatnega sistema pogleda v koordinatni sistem projekcije kamere. Nato še posodobimo čas in izrišemo sistem delcev v trenutnem času[10, 3] (Izsek kode 3.16).

```
glUseProgram(particleShaderProgram);
glUniform1f(particleTimeUniform, particleSystemTime);
glUniformMatrix4(particleModelViewMatrixUniform, false,
    matrix4fToFB(modelViewMatrix));
glUniformMatrix4(particleProjectionMatrixUniform, false,
    matrix4fToFB(projectionMatrix));

particleSystemTime += 0.005f;
if(particleSystemTime > 10.0f){
        particleSystemTime = 0.0f;
}

particleSystem.drawPoints();

glUseProgram(0);
```

Izsek kode 3.16: Nastavitev globalnih spremenljivk za izris sistema delcev.

Ključ pri izrisovanju sistema delcev je senčilnik oglišč, ki namesto transformacije vhodnih oglišč izračuna trenutno lokacijo posameznega oglišča s funkcijo, ki za parametra vzame začetni položaj, ostale atribute oglišča in globalno spremenljivko, ki predstavlja čas, v katerem se nahaja sistem del-

cev. Transformira se šele na novo izračunan položaj delca (Izsek kode 3.17).

Senčilnik oglišč najprej izračuna starost delca. Če je ta vrednost manjša od 0, potem delec še ni nastal. V tem primeru delec ostane na začetnem položaju in dobi prozorno barvo. Če pa je vrednost večja od 0, uporabimo naslednjo enačbo za izračun trenutnega položaja delca:

$$P = P_i + vt + \frac{1}{2}at^2$$

V tej enačbi P_i predstavlja začetni položaj delca , v predstavlja začetno hitrost, t predstavlja pretečen čas, a predstavlja pospešek in P predstavlja izračunan položaj delca. Za pospešek smo vzeli gravitacijski pospešek na zemeljskem površju, ki znaša $9.8\frac{m}{s^2}$. V našem poenostavljenem primeru pospešek vpliva le na y koordinato. Na koncu transformiramo novo lokacijo delca[3] (Izsek kode 3.17).

```
uniform float Time;
uniform mat4 ProjectionMatrix;
uniform mat4 ModelViewMatrix;

layout (location = 0) in vec3 MCVertex;
layout (location = 1) in vec3 MColor;
layout (location = 2) in vec3 Velocity;
layout (location = 3) in float StartTime;

out vec4 Color;

void main(){
        vec4 vert;
        float t = Time - StartTime;

        if(t >= 0.0 && t <= 3.0){
                vert = vec4(MCVertex, 1.0) + vec4(Velocity * t,
                    0.0);
                vert.y -= 5.9 * t * t;
                Color = vec4(MColor, 1.0);
        }
        else{
```

```
                vert = vec4(MCVertex, 1.0);
                Color = vec4(0.0, 0.0, 0.0, 0.0);
        }

        gl_Position = ProjectionMatrix * ModelViewMatrix * vert;
}
```

Izsek kode 3.17: Senčilnik oglišč sistema delcev.

Senčilnik fragmentov je zelo enostaven, saj le shrani barvo posameznega delca, ki jo dobi iz senčilnika oglišč (Izsek kode 3.18).

```
in vec4 Color;

layout (location = 0) out vec4 FragColor;

void main(){
        FragColor = Color;
}
```

Izsek kode 3.18: Senčilnik fragmentov sistema delcev.

Izzivi pri implementaciji so se pojavili pri izrisovanju sistema delcev s pomnilnikom oglišč, saj se z LWJGL knjižnico pomnilniki oglišč uporabljajo drugače kot pri drugih jezikih. Ugotoviti je bilo potrebno, kako podati matrike senčilniku oglišč, saj se jih v LWJGL knjižnici podaja kot pomnilnik števil in potrebno je bilo pravilno pretvoriti matrike v pomnilnik števil. Nato smo izdelali senčilnike za sistem delcev in dodali, da so delci v začetnem stanju prosojni. Spremenili smo kreiranje sistema delcev, tako da nam omogoča postavitev sistema delcev na različne lokacije. S spreminjanjem parametrov sistema delcev smo lahko dosegli zelo različne učinke. Ugotovili smo, da nam že preprost sistem delcev omogoča veliko fleksibilnost.

3.7 Senčilnik za Phongov model globalnega osvetljevanja

Z vpeljavo osvetljevanja pridobimo veliko na realističnosti grafike, brez da bi zelo povečali grafično zahtevnost aplikacije. Eden izmed najbolj pogosto uporabljenih algoritmov je Phongov model osvetljevanja, ki je sestavljen iz treh približkov odbojev svetlobe. To so ambientni, razpršeni in zrcalni odboj. Phongov model osvetljevanja da zelo dober rezultat ob seštetju vseh treh komponent in je računsko nezahteven. Na sliki 3.8 vidimo primerjavo med enakomernim in Phongovim osvetljevanjem. Na sliki 3.7 pa vidimo Phongovo osvetljevanje brez tekstur.

Skupno intenziteto odbite svetlobe (I), izračunamo tako, da preprosto seštejemo posamezne intenzitete za zrcalni (L_s), razpršeni (L_d) in ambientni (L_a) odboj.

$$I = L_a + L_d + L_s$$

Če hočemo, da je odbita svetloba odvisna tudi od površine, od katere se odbije, dodamo koeficiente površine k vsaki komponenti odbite svetlobe.

$$I = L_a K_a + L_d K_d + L_s K_s$$

Ambientna svetloba predstavlja približek svetlobe, ki se nahaja povsod v sceni, saj se le ta odbija od sten in ostalih predmetov. Povsod dodamo konstantno osvetlitev in tako nimamo več popolnoma temnih delov.

Ambientna svetloba (I_a) je enaka produktu ambientne komponente luči (L_a) in ambiente komponente površine (K_a).

$$I_a = L_a K_a$$

Razpršena svetloba predstavlja približek svetlobe, ki se odbija enakomerno na vse strani od površine, na katero sveti luč. Ni pomembno, pod kakšnim kotom gledamo površino, vedno vidimo enako odbito svetlobo. S tem dosežemo motno, nesvetlečo površino, ki je drobno hrapava.

Razpršena svetloba je odvisna samo od kota med normalo površine, na katero sveti luč, in položajem same luči. Približek kota med normalo površine v točki odboja (n) in vektorjem smeri od točke odboja do luči (s) dobimo s skalarnim produktom teh dveh vektorjev. Vektor smeri od točke odboja do luči dobimo tako, da od položaja luči odštejemo položaj točke odboja in rezultat normaliziramo. Paziti moramo, da so vse točke, s katerimi računamo, v enakem koordinatnem sistemu.

Razpršena svetloba (I_d) je enaka produktu razpršilne komponente luči (L_d), razpršilne komponente površine (K_d) in skalarnega produkta normale površine v točki odboja (n) in vektorja smeri od točke odboja do luči (s).

$$I_d = L_d K_d(\vec{n} \cdot \vec{s})$$

Zrcalna svetloba predstavlja približek svetlobe, ki se odbije pod približno istim kotom, kot je le ta padla na površino. Tako dobimo učinek sijaja, ki predstavlja zelo gladko površino. Dobimo zabrisan odsev vira svetlobe. Prav tako pa je zrcalni odboj odvisen tudi od smeri, iz katere gledamo površino.

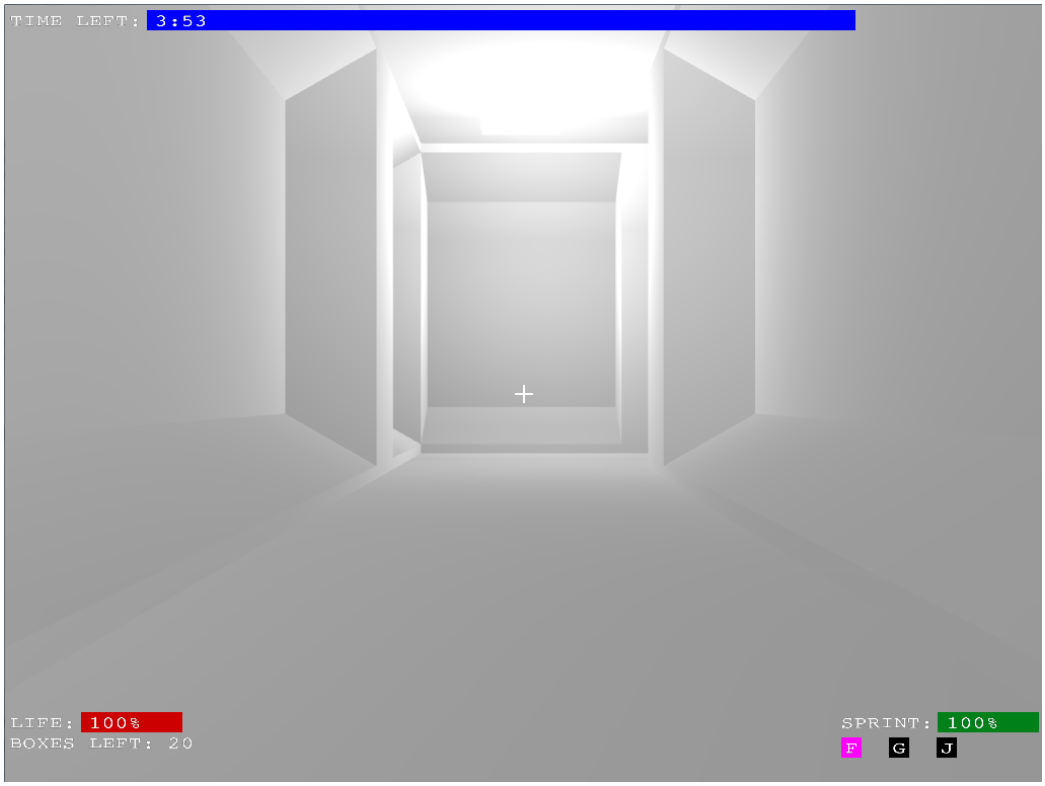
Najprej moramo dobiti vektor svetlobe, ki se je odbila pod približno istim kotom, kot je le ta padla na površino (r). To naredimo z vektorskim produktom vektorja smeri od točke odboja do luči (s) in normale površine v točki odboja (n). Produkt nato z desne strani pomnožimo z normalo površine v točki odboja (n) in vse skupaj pomnožimo z dve ter odštejemo vektor smeri od točke odboja do luči(s).

$$r = 2(\vec{s} \cdot \vec{n})\vec{n} - \vec{s}$$

Vektor smeri od točke odboja do položaja kamere (v) izračunamo tako, da odštejemo položaj točke odboja od položaja kamere in rezultat normaliziramo.

Zrcalna svetloba (I_s) je enaka produktu zrcalne komponente luči (L_s), zrcalne komponente površine (K_s) in potence, katere eksponent je stopnja zrcaljenja (Ns), skalarnega produkta vektorja odbite svetlobe pod istim kotom, kot je ta padla na površino (r), in vektorja smeri od točke odboja do

položaja kamere(v).

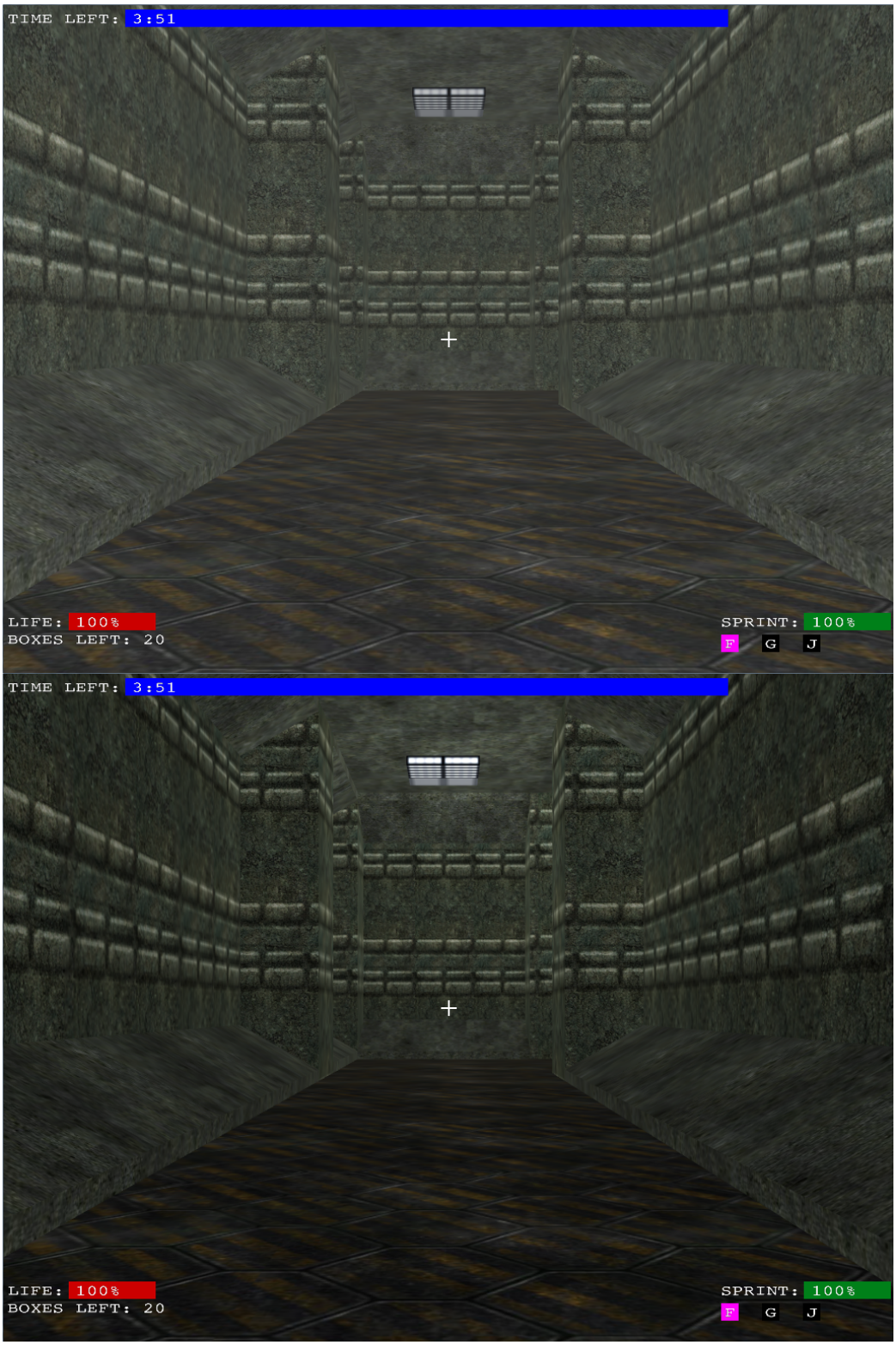


Slika 3.7: Prikaz scene s Phongovim osvetljevanja brez tekstur.

Za prilagajanje zrcalnega odboja potenciramo skalarni produkt s stopnjo zrcaljenja (Ns), in sicer: večje je število, manjši in bolj direkten bo odboj[8, 18, 19].

$$I_s = L_s K_s (\vec{r} \cdot \vec{v})^{Ns}$$

V senčilniku oglišč zgolj pripravimo potrebne spremenljivke, medtem ko osvetljevanje izračunamo v senčilniku fragmentov. Če izračunamo barvo za vsak fragment, dobimo dosti boljše rezultate, kot če bi izračunali barvo za vsako oglišče in to barvo potem interpolirali za vsak fragment[19]. V našem primeru smo dodali še teksture in upoštevali več luči v sceni.



Slika 3.8: Prikaz scene z enakomernim osvetljevanjem zgoraj in s Phongovim osvetljevanjem spodaj.

V senčilniku oglišč najprej transformiramo položaj oglišča iz koordinatnega sistema modela v koordinatni sistem pogleda. Nato transformiramo normalo tega oglišča iz koordinatnega sistema modela v koordinatni sistem pogleda. Matrika, ki nam transformira normalo, je enaka matriki, ki nam transformira oglišče, le da ne vsebuje zadnjega stolpca in zadnje vrstice. Matrika za preslikavo normal je drugačna le v primeru, če neenakomerno povečamo objekt, vendar se to zgodi zelo redko. V naslednjem koraku zapišemo v spremenljivko še *uv-koordinati* teksture. Do sedaj izračunane vrednosti pošljemo naprej senčilniku fragmentov. Na koncu še transformiramo oglišče v koordinatni sistem projekcije kamere[8, 18, 19] (Izsek kode 3.19).

```
out vec4 Position;
out vec4 Normal;
out vec4 Light;
out vec2 TexCoord;

void main(){
        Position = gl_ModelViewMatrix * gl_Vertex;
        Normal = vec4(gl_NormalMatrix * gl_Normal, 1.0);
        TexCoord = gl_MultiTexCoord0.st;
        gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

Izsek kode 3.19: Senčilnik oglišč za Phongov model osvetljevanja.

V senčilniku fragmentov najprej dobimo vrednost teksture v danem fragmentu. Nato se sprehodimo po vseh lučeh v sceni in za vsako posebej izračunamo Phongov model osvetljevanja, po enačbah, ki smo jih opisali (I), le da smo tukaj dodali funkcijo, ki poskrbi za upadanje intenzitete luči z oddaljenostjo fragmenta od luči. Koeficient upadanje intenzitete luči (f) izračunamo tako, da 1 delimo s vsoto 1, razdalje od fragmenta do luči (d), ulomljeno z 150 in razdalje od fragmenta do luči (d), ulomljeno s 150 na drugo potenco[18].

$$\frac{1}{1 + \frac{d}{150} + (\frac{d}{150})^2}$$

Na koncu seštejemo skupno ambientno in skupno razpršilno komponento ter ju pomnožimo z vrednostjo teksture ter prištejemo skupno zrcalno komponento. S tem dosežemo, da tekstura vpliva samo na ambientno in razpršilno komponento (Izsek kode 3.20).

```
uniform sampler2D Texture;

const vec4 Ka = vec4(0.1, 0.1, 0.1, 1.0);
const vec4 Kd = vec4(0.7, 0.7, 0.7, 1.0);
const vec4 Ks = vec4(1.0, 1.0, 1.0, 1.0);
const float Ns = 128;

in vec4 Normal;
in vec4 Position;
in vec2 TexCoord;

vec4 ads(){
        vec4 totalD = vec4(0.0, 0.0, 0.0, 1.0);
        vec4 totalS = vec4(0.0, 0.0, 0.0, 1.0);
        vec4 totalA = vec4(0.0, 0.0, 0.0, 1.0);
        vec4 texColor = texture(Texture, TexCoord);
        for(int i = 0; i < 8; i++){
                vec4 n = normalize(Normal);
                vec4 s = normalize(gl_LightSource[i].position -
                    Position);
                vec4 v = normalize(-Position);
                vec4 r = reflect(-s, n);
                vec4 texColor = texture(Texture, TexCoord);
                float d = length(gl_LightSource[i].position -
                    Position);
                float f = 1 / (1 + (d/150) + pow((d/150), 2));
                vec4 Id = gl_LightSource[i].diffuse * Kd * max(
                    dot(s, n), 0.0) + vec4(1.0, 1.0, 1.0, 1.0);
```

```
                vec4 Is = gl_LightSource[i].specular * Ks * pow(
                    max(dot(r, v), 0.0), Ns);
                vec4 Ia = gl_LightSource[i].ambient * Ka;
                totalA = totalA + Ia;
                totalD = totalD + Id * f;
                totalS = totalS + Is * f;
        }
        return (totalA + totalD) * texColor + totalS;
}

void main(){
        gl_FragColor = ads();
}
```

Izsek kode 3.20: Senčilnik fragmentov za Phongov model osvetljevanja.

Pri izrisu scene moramo na začetku vedno nastaviti luči, da se le te pravilno transformirajo s trenutno matriko na skladu matrik, nato omogočimo senčilniški program, podamo teksturo senčilniku in izrišemo sceno (Izsek kode 3.21).

```
GL11.glEnable(GL11.GL_LIGHTING);

float[] light_position0 = new float[] {0.0f, 100.0f, 0.0f, 1.0f
    };
float[] light_position1 = new float[] {430.0f, 100.0f, -750.0f,
    1.0f};
float[] light_position2 = new float[] {1200.0f, 100.0f, -750.0f,
     1.0f};
float[] light_position3 = new float[] {2100.0f, 100.0f, -750.0f,
     1.0f};
float[] light_position4 = new float[] {2700.0f, 100.0f, 50.0f,
    1.0f};
float[] light_position5 = new float[] {1800.0f, 100.0f, 50.0f,
    1.0f};
float[] light_position6 = new float[] {1000.0f, 100.0f, 500.0f,
    1.0f};
float[] light_position7 = new float[] {100.0f, 100.0f, 750.0f,
    1.0f};
```

```
glLight(GL_LIGHT0, GL_POSITION, allocFloats(light_position0));
glLight(GL_LIGHT1, GL_POSITION, allocFloats(light_position1));
glLight(GL_LIGHT2, GL_POSITION, allocFloats(light_position2));
glLight(GL_LIGHT3, GL_POSITION, allocFloats(light_position3));
glLight(GL_LIGHT4, GL_POSITION, allocFloats(light_position4));
glLight(GL_LIGHT5, GL_POSITION, allocFloats(light_position5));
glLight(GL_LIGHT6, GL_POSITION, allocFloats(light_position6));
glLight(GL_LIGHT7, GL_POSITION, allocFloats(light_position7));

glEnable(GL_LIGHT0);
glEnable(GL_LIGHT1);
glEnable(GL_LIGHT2);
glEnable(GL_LIGHT3);
glEnable(GL_LIGHT4);
glEnable(GL_LIGHT5);
glEnable(GL_LIGHT6);
glEnable(GL_LIGHT7);

glUseProgram(lightingShaderProgram);

glUniform1i(lightingTextureUniform, 0);

// Izrisemo celotno sceno.

glUseProgram(0);
```

Izsek kode 3.21: Nastavitve luči in teksture za izris scene z Phongovim osvetljevanjem.

Phongov model osvetljevanja smo napisali z zastarelim GLSL jezikom, saj uporablja spremenljivke s predpono *gl_*, ki so zastarele v novejših različicah jezika. V takem jeziku je napisan zaradi predstavitve zastarelega jezika in zaradi tega, ker igrica, v katero smo implementirali ta senčilnik uporablja, zastarelo različico OpenGL in prav tako GLSL.

Pri dodajanju teksture smo morali realizirati spreminjanje teksture za posamezne primitive in vrednost teksture pravilno dodati v Phongov model

osvetljevanja.

Izzivi so se pojavili pri uporabi luči, saj so se le te premikale s kamero. To smo odpravili tako, da smo njihove lokacije pred vsakim izrisom scene ponovno transformirali. Prav tako smo se morali seznaniti z uporabo že vgrajenih spremenljivk za posamezno luč.

Ker se v sceni seštevajo svetlobe posameznih luči, smo tako dobili popolnoma svetlo sceno. Da to preprečimo, smo morali implementirati funkcijo za vpadanje jakosti posamezne luči z njeno oddaljenostjo. Za to funkcijo je bilo potrebno veliko testiranja, preden smo našli vrednost, ki daje želene rezultate pri osvetljevanju.

3.8 Senčilnik za mapiranje senc

Osnovni algoritem mapiranja senc je sestavljen iz dveh korakov. V prvem koraku izrišemo sceno, kot jo vidi luč, zavržemo informacijo o barvi ter shranimo le podatke o globini. V drugem koraku izrišemo sceno, kot jo vidi kamera. Transformiramo vsako točko v koordinatni sistem luči in ugotovimo, ali je točka v vidnem polju luči. Če je točka v vidnem polju luči, potem jo izrišemo normalno. Če točka ni v vidnem polju luči potem jo izrišemo v senci (Slika 3.10).

Mapiranje senc je sistemsko nezahteven algoritem. Njegova kompleksnost narašča linearno s kompleksnostjo scene. Algoritem zelo dobro izkorišča trenutno grafično strojno oprem in z njim dosežemo zelo realistični efekt. Sence lahko zameglimo, s tem dosežemo mehke sence.

Slabe lastnosti algoritma so, da navadno uporablja nizko resolucijske teksture, ki so projicirane pod kotom in tako nastanejo napake pri izrisu. Tudi z implementacijo številnih naprednih tehnik za zmanjšanje napak pri izrisu se še vedno težko znebimo vseh napak. Algoritem je zahteven za implementacijo in ga je težko razhroščevati.

Slika 3.9: Prikaz globinske teksture s položaja luči.

V prvem koraku ustvarimo in nastavimo teksturo, v katero bomo shranili globinsko informacijo vidnega polja luči. Če imamo v sceni več luči, moramo za vsako luč narediti svojo globinsko sliko njihovega vidnega polja. V našem primeru bomo imeli samo eno luč. Ustvarimo teksturo, ki shrani samo globinsko informacijo za vsak fragment GL_DEPTH_COMPONENT[15]. Velikost teksture vpliva na sistemsko zahtevnost algoritma, zato izberemo najnižjo resolucijo teksture, ki nam še omogoči želen rezultat. V našem primeru je velikost teksture enako našemu zaslonu.

Nato nastavimo funkciji, ki poskrbita za vrednost posameznega fragmenta, ali teksturo nanašamo na ploskev, večjo od teksture (GL_TEXTURE_MIN_FILTER)[15], ali jo nanašamo na ploskev, manjšo od teksture (GL_TEXTURE_MAG_FILTER)[15]. S tem izberemo teksturni fragment, ki je najbližje centru fragmenta, za katerega vrednost iščemo (GL_NEAREST)[15].

Nastavimo še parametra za ovitje teksture, da ne pride do napak pri izrisovanju robnih elementov teksture, v interval znotraj $[\frac{1}{2N}, 1-\frac{1}{2N}]$, kjer je N velikost teksture v posamezni smeri (GL_CLAMP_TO_EDGE)[15]. Nastavimo ju za vsako *uv-koordinato* posebej (GL_TEXTURE_WRAP_S, GL_TEXTURE-

_WRAP_T)[15]. Ta nastavitev nam tudi prepreči nastajanje robov senc pri dodajanju senc iz večih globinskih tekstur.

Za kasnejše izboljševanje mapiranja senc nastavimo mejo teksture na črno barvo. S tem tudi dosežemo nenamerno ponavljanje teksture izven vidnega polja luči.

Zadnja stvar, ki jo naredimo pri nastavljanju teksture, je, da izklopimo primerjalno funkcijo teksture, saj primerjanje opravimo sami in lahko povzroči probleme[19] (Izsek kode 3.22).

```
shadowMappingTexture = glGenTextures();
glBindTexture(GL_TEXTURE_2D, shadowMappingTexture);
glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT, SCREEN_WIDTH,
    SCREEN_HEIGHT, 0, GL_DEPTH_COMPONENT, GL_UNSIGNED_BYTE, (
    ByteBuffer)null);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST
    );
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST
    );
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
    GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
    GL_CLAMP_TO_EDGE);
glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR,
    asFloatBuffer(new float[]{0.0f, 0.0f, 0.0f, 0.0f}));
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_MODE, GL_NONE)
    ;
glBindTexture(GL_TEXTURE_2D, 0);
```

Izsek kode 3.22: Ustvarimo teksturo za shranjevanje globinske informacije.

Nato moramo globinsko teksturo pripeti na pomnilnik slik, če imamo več globinskih tekstur vsako pripnemo drugemu pomnilniku slik. Najprej ustvarimo pomnilnik slik, mu pripnemo globinsko teksturo ter mu onemogočimo izris barv (Izsek kode 3.23).

```
shadowMappingFBO = glGenFramebuffers();
glBindFramebuffer(GL_FRAMEBUFFER, shadowMappingFBO);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
```

```
    GL_TEXTURE_2D, shadowMappingTexture, 0);
glDrawBuffer(GL_NONE);
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

Izsek kode 3.23: Ustvarimo pomnilnik slike za shranjevanje globinske informacije scene v teksturo.

Globinsko teksturo scene (Slika 3.9) dobimo tako, da najprej omogočimo pomnilnik slik, na katerega je pripeta globinska tekstura. Nato izbrišemo trenutni pomnilnik barv in globine ter omogočimo, da se pri izrisovanju odstranijo poligoni, ki so v vidnem polju kamere. S tem izboljšamo globinsko teksturo in zmanjšamo število napak pri sencah. Omogočimo senčilniški program, ki vsebuje enostavna prehodna senčilnika ter mu zagotovimo potrebne matrike in izrišemo sceno. Senčilniku podamo matriko za transformacijo iz koordinatnega sistema modela v koordinatni sistem sveta, matriko za transformacijo iz koordinatnega sveta v koordinatni sistem pogleda luči in matriko za transformacijo iz koordinatnega sveta pogleda luči v koordinatni sistem projekcije luči[13, 19] (Izsek kode 3.24).

```
private static void renderShadowScene() {
        glBindFramebuffer(GL_FRAMEBUFFER, shadowMappingFBO);

        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        glCullFace(GL_FRONT);

        glUseProgram(shadowMappingShaderProgram);

        glUniformMatrix4(shadowViewMatrixUniform, false,
            matrix4fToFB(lightViewMatrix));
        glUniformMatrix4(shadowProjectionMatrixUniform, false,
            matrix4fToFB(projectionMatrix));

        renderScene(shadowModelMatrixUniform);

        glUseProgram(0);

        glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

```
}

private static void renderScene(int uniform) {
        for(int i = 0; i < cubes.size(); i++){
                glUniformMatrix4(uniform, false, matrix4fToFB(
                    cubes.get(i).getModelMatrix()));
                cubes.get(i).draw();
        }
}
```

Izsek kode 3.24: Izris scene in shranjevanje globinske informacije v podano teksturo.

Senčilnik oglišč, ki ga uporabimo pri izrisu globinske teksture, preprosto transformira oglišča v projekcijo luči[19] (Izsek kode 3.25).

```
uniform mat4 ModelMatrix;
uniform mat4 ViewMatrix;
uniform mat4 ProjectionMatrix;

layout (location = 0) in vec3 VertexPosition;

void main(){
        gl_Position = ProjectionMatrix * ViewMatrix *
            ModelMatrix * vec4(VertexPosition, 1.0);
}
```

Izsek kode 3.25: Senčilnik oglišč za transformacijo oglišč na prave položaje.

Senčilnik fragmentov pa ne naredi ničesar, saj se globinska informacija samodejno shrani v pomnilnik[19] (Izsek kode 3.26).

```
void main(){
        // Do nothing.
}
```

Izsek kode 3.26: Senčilnik fragmentov ne naredi ničesar, saj se globinska informacija zapiše avtomatsko.

Za izris scene s sencami najprej izbrišemo trenutni pomnilnik barv in globine ter nastavimo, da pri izrisovanje odstranimo tiste poligone, ki so zakriti

v vidnem polju kamere. Nato aktiviramo poljubno teksturo in ji pripnemo našo globinsko teksturo. Omogočimo senčilniški program in mu podamo matriko za transformacijo iz koordinatnega sistema modela v koordinatni sistem sveta (matrika modela), matriko za transformacijo iz koordinatnega sveta v koordinatni sistem pogleda kamere (matrika pogleda kamere), matriko za transformacijo iz koordinatnega sveta v koordinatni sistem projekcije kamere (projekcijska matrika kamere) in matriko za transformacijo iz koordinatnega sveta v koordinatni sistem pogleda luči (matrika pogleda luči). Podamo mu še globinsko teksturo, položaj luči in barvo, nato izrišemo sceno[14, 19] (Izsek kode 3.27).

```
private static void renderCombinedScene() {
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

        glCullFace(GL_BACK);

        glActiveTexture(GL_TEXTURE0);
        glBindTexture(GL_TEXTURE_2D, shadowMappingTexture);

        glUseProgram(shaderProgram);

        glUniformMatrix4(viewMatrixUniform, false, matrix4fToFB(
            viewMatrix));
        glUniformMatrix4(projectionMatrixUniform, false,
            matrix4fToFB(projectionMatrix));
        glUniformMatrix4(lightViewMatrixUniform, false,
            matrix4fToFB(lightViewMatrix));
        glUniform1i(shadowMapUniform, 0);
        glUniform3f(lightColorUniform, LIGHT_COLOR.x,
            LIGHT_COLOR.y, LIGHT_COLOR.z);
        glUniform3f(lightPositionUniform, LIGHT_POSITION.x,
            LIGHT_POSITION.y, LIGHT_POSITION.z);

        renderScene(modelMatrixUniform);

        glUseProgram(0);
        glBindTexture(GL_TEXTURE_2D, 0);
```

```
}

private static void renderScene(int uniform) {
        for(int i = 0; i < cubes.size(); i++){
                glUniformMatrix4(uniform, false, matrix4fToFB(
                    cubes.get(i).getModelMatrix()));
                cubes.get(i).draw();
        }
}
```

Izsek kode 3.27: Izris končne scene s sencami.

Senčilnik oglišč mapiranja senc opravlja še Phongovo globalno osvetljevanje, ki je opisan v prejšnjem poglavju. Najprej moramo izračunati matriko, ki preslika oglišče v koordinatni sistem projekcije luči. V našem primeru ima projekcija luči enako projekcijsko matriko in matriko modela ter različno matriko pogleda. Torej pomnožimo matriko modela z matriko pogleda luči in nato še z matriko projekcije. Rezultat pomnožimo še s posebno matriko (B), ki smo definirali kot konstanto v našem senčilniku, s katero dosežemo, da so vse vrednosti rezultata na intervalu med 0 in 1[19].

$$B = \begin{bmatrix} 0.5 & 0.0 & 0.0 & 0.5 \\ 0.0 & 0.5 & 0.0 & 0.5 \\ 0.0 & 0.0 & 0.5 & 0.5 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix}$$

Tako dobimo matriko, ki transformira oglišče v projekcijo luči, z vrednostmi na intervalu med 0 in 1. Da so koordinate na intervalu med 0 in 1 želimo, ker smo globinsko sliko scene shranili v teksturo, ki ima koordinate tudi na intervalu med 0 in 1. Nato še pomnožimo oglišče z izračunano matriko in te koordinate pošljemo senčilniku fragmentov[19] (Izsek kode 3.28).

```
uniform mat4 ModelMatrix;
uniform mat4 ViewMatrix;
uniform mat4 ProjectionMatrix;
uniform mat4 LightViewMatrix;
```

```
layout (location = 0) in vec3 VertexPosition;
layout (location = 1) in vec3 VertexNormal;
layout (location = 2) in vec3 VertexColor;

out vec3 Color;
out vec3 Normal;
out vec3 PositionV;
out vec4 ShadowCoord;

const mat4 bias = mat4(0.5, 0.0, 0.0, 0.0,
                       0.0, 0.5, 0.0, 0.0,
                       0.0, 0.0, 0.5, 0.0,
                       0.5, 0.5, 0.5, 1.0);

void main(){
        Color = VertexColor;
        Normal = VertexNormal;

        //Shadow mapping
        mat4 ShadowMatrix = bias * ProjectionMatrix *
            LightViewMatrix * ModelMatrix;
        ShadowCoord = ShadowMatrix * vec4(VertexPosition, 1.0);

        PositionV = (ViewMatrix * ModelMatrix * vec4(
            VertexPosition, 1.0)).xyz;
        gl_Position = ProjectionMatrix * ViewMatrix *
            ModelMatrix * vec4(VertexPosition, 1.0);
}
```

Izsek kode 3.28: Senčilnik oglišč za izris končne scene s sencami.

V senčilniku fragmentov najprej inicializiramo faktor sence in ga postavimo na vrednost 1, kar pomeni, da fragment ni v senci. Nato inicializiramo in definiramo majhno napako, ki nam prepreči nastanek napak. Opravimo še perspektivno deljenje naših koordinat iz senčilnika oglišč. S tem spravimo koordinate na interval med -1 in 1 in dosežemo, da oddaljeni x in y koordinati prestavimo bližje izhodišču koordinatnega sistema. S temu dosežemo,

da bolj oddaljeni predmeti izgledajo manjši.

V prvem preverjanju preprečimo, da se pojavijo inverzne in narobe obrnjene sence. Z naslednjima dvema preverjanjema preprečimo, da se pojavijo sence oglišč izven vidnega polja. Končno lahko varno uporabimo x in y koordinati, s katerima opravimo poizvedbo po globinski teksturi. Ta nam vrne vektor s štirimi enakimi vrednostmi in vzamemo prvo od njih. Nato preverimo, ali je globinska vrednost v teksturi manjša od globine oglišča na enakih koordinatah v projekcijskem prostoru luči. Če je vrednost manjša, potem se mora nekaj nahajati med globinsko vrednostjo teksture in lučjo, iz česar sledi, da je ta fragment v senci in faktor senčenja postavimo na 0.

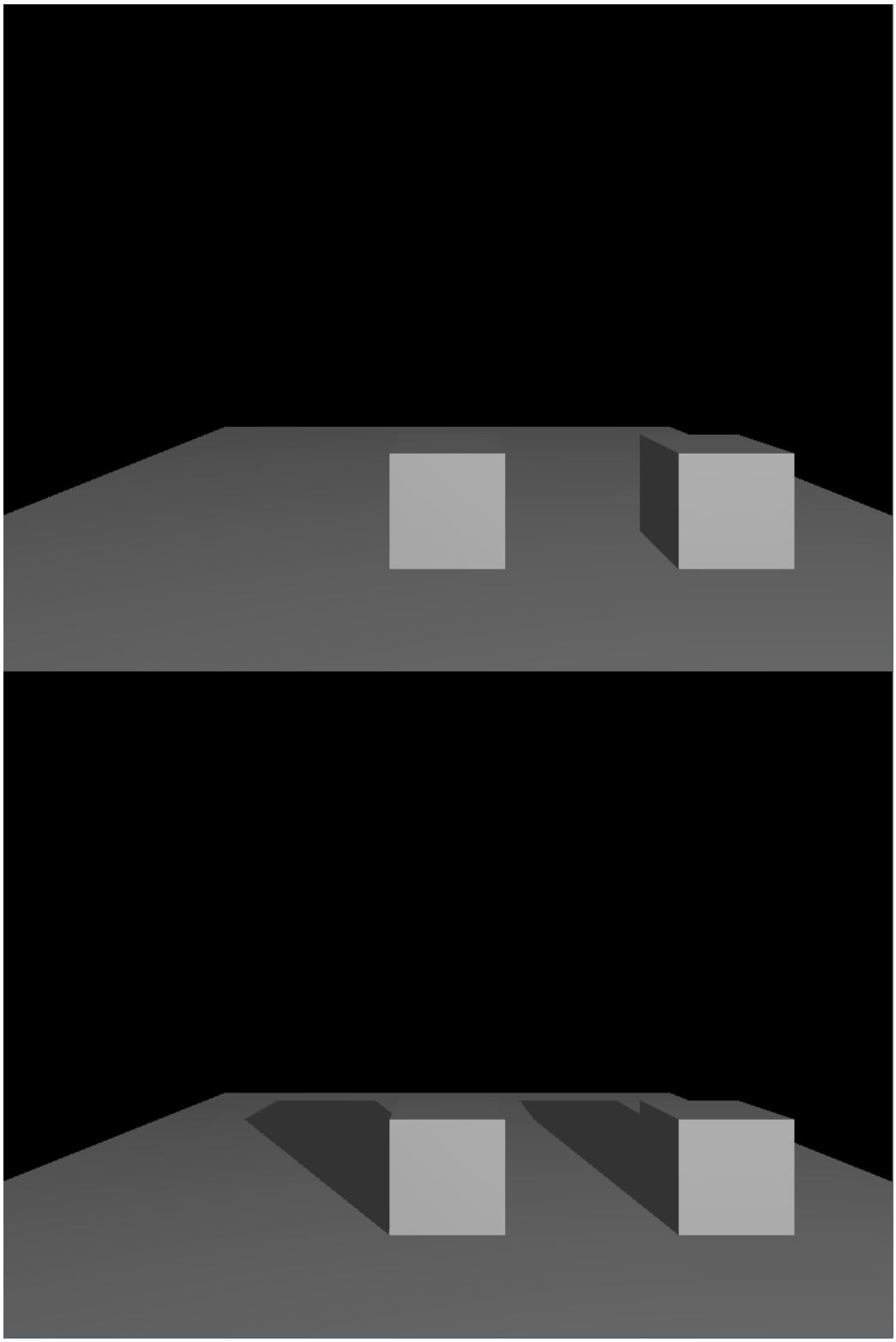
Končno barvo dobimo preprosto tako, da vsoto zrcalne komponente in razpršilne komponente barve pomnožimo z našim faktorjem sence in prištejemo še ambientno komponento barve[8, 19]. S tem dobijo fragmenti v senci barvo enako ambientni komponenti barve (Izsek kode 3.29).

```
uniform sampler2D ShadowMap;
uniform vec3 LightPosition;
uniform vec3 LightColor;

in vec3 Color;
in vec3 Normal;
in vec3 PositionV;
in vec4 ShadowCoord;

out vec4 FragColor;

vec3 ads(){
        //Phong global shading
}

void main(){
        float shadowFactor = 1.0;
        float epsilon = 0.0005;
        vec4 shadCoordsPD = ShadowCoord / ShadowCoord.w;
        if(shadCoordsPD.w <= 0.0f){
                shadowFactor = 1;
```

```
        }
        else if(shadCoordsPD.x < 0 || shadCoordsPD.y < 0){
                shadowFactor = 1;
        }
        else if(shadCoordsPD.x >= 1 || shadCoordsPD.y >= 1){
                shadowFactor = 1;
        }
        else{
                float shadow = texture2D(ShadowMap, shadCoordsPD
                    .xy).x;
                if(shadow + epsilon < shadCoordsPD.z){
                shadowFactor = 0.0;
            }
        }
        FragColor = vec4(Id + Is, 1.0) * shadowFactor + Ia;
}
```

Izsek kode 3.29: Senčilnik fragmentov za izris končne scene s sencami.

Velik izziv je predstavljala nastavitev teksture, v katero smo izrisali globinsko informacijo scene, saj je bila le ta v literaturi napačna [8]. Po daljšem raziskovanju smo našli rešitev v literaturi [19], v kateri je ta napaka tudi omenjena. Naslednji izziv je prestavljalo ohranjanje pravih matrik za transformiranje oglišč v projekcijo luči, da smo lahko z njo pravilno izrisali sence na naši končni sceni. Prav tako so bili vsi primeri napisani v drugem programskem jeziku in smo morali ugotoviti uporabe podanih funkcij z LWJGL knjižnico. Potem smo napisali senčilnika in na sceni ni bilo senc. Ugotovili smo, da smo narobe kreirali potrebne matrike za transformacijo. Osnovni algoritem mapiranja senc naredi sence z ostrim robom, ki jih lahko odpravimo z več metodami, na primer sence lahko zameglimo. Mapiranje senc se pogosto uporablja predvsem zato, ker je algoritem računsko manj zahteven od alternativnih algoritmov, vendar pa potrebuje nekaj izboljšav za prikaz realističnega videza.

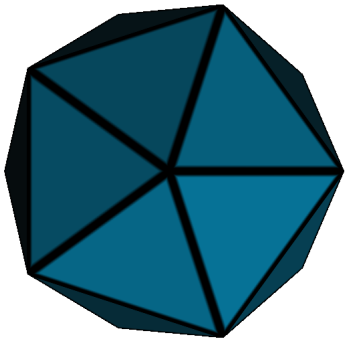


Slika 3.10: Prikaz scene brez senc zgoraj in s sencami spodaj.

3.9 Senčilnik za teselacijo trikotnikov

V našem primeru smo z uporabo teselacijskih senčilnikov iz ikozaedra (Slika 3.11) dobili približek krogle. Dodali smo še rotacijo in razpršilno osvetljevanje za bolj nazorno predstavitev.

Ikozaeder pripravimo za izris tako, da zapišemo koordinate oglišč, v sezname, ki jih shranimo v pomnilnik oglišč, ki ga nato pripnemo še seznamu pomnilnikov oglišč. Podobno naredimo s seznamom, ki pove, katera oglišča tvorijo posamezen trikotnik (Izsek kode 3.30).



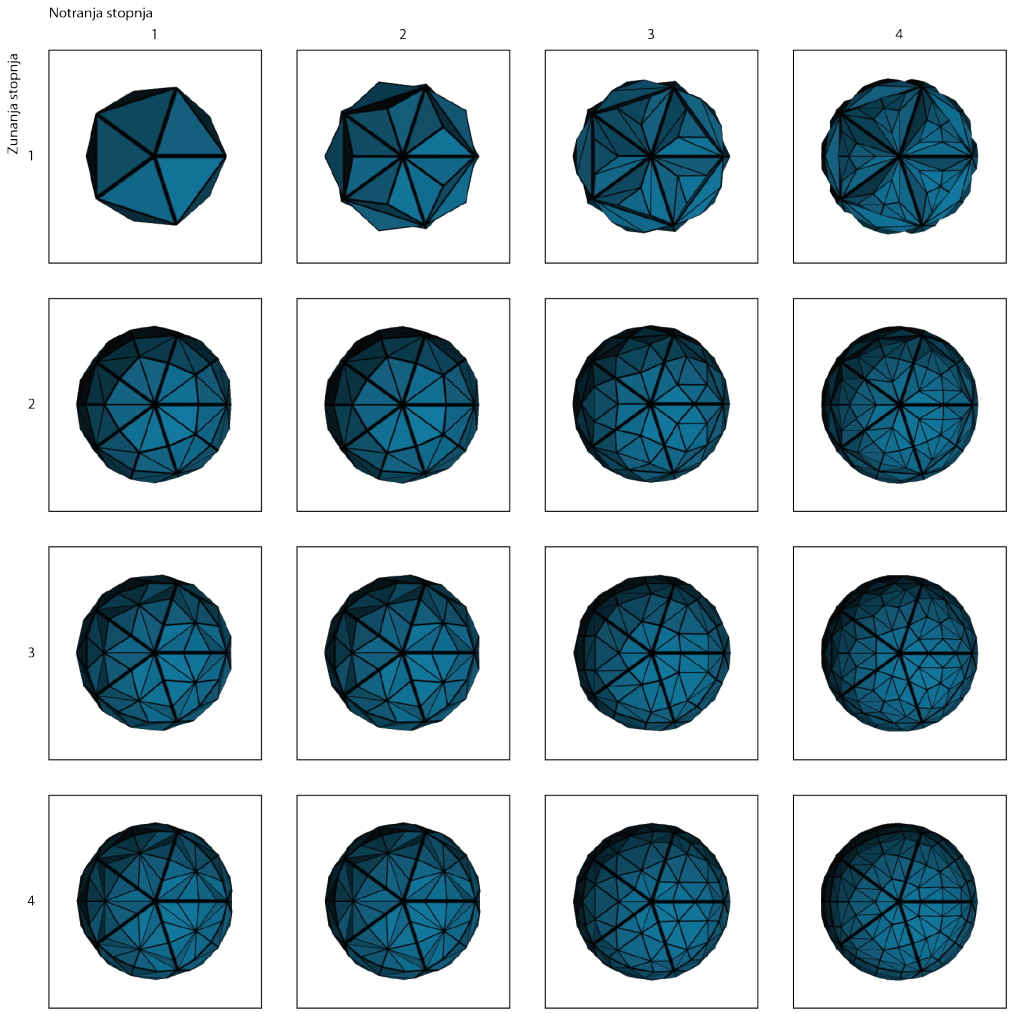
Slika 3.11: Ikozaeder.

```
float[] vertices = {
         0.000f,  0.000f,  1.000f,
         0.894f,  0.000f,  0.447f,
         0.276f,  0.851f,  0.447f,
        -0.724f,  0.526f,  0.447f,
        -0.724f, -0.526f,  0.447f,
         0.276f, -0.851f,  0.447f,
         0.724f,  0.526f, -0.447f,
        -0.276f,  0.851f, -0.447f,
        -0.894f,  0.000f, -0.447f,
        -0.276f, -0.851f, -0.447f,
         0.724f, -0.526f, -0.447f,
         0.000f,  0.000f, -1.000f
```

```
};

byte[] indices = {
        2, 1, 0,
        3, 2, 0,
        4, 3, 0,
        5, 4, 0,
        1, 5, 0,
        11, 6,  7,
        11, 7,  8,
        11, 8,  9,
        11, 9,  10,
        11, 10, 6,
        1, 2, 6,
        2, 3, 7,
        3, 4, 8,
        4, 5, 9,
        5, 1, 10,
        2,  7, 6,
        3,  8, 7,
        4,  9, 8,
        5, 10, 9,
        1, 6, 10
};

indicesCountI = indices.length;

FloatBuffer verticesBuffer = asFloatBuffer(vertices);
ByteBuffer indicesBuffer = asByteBuffer(indices);

vaoIID = glGenVertexArrays();
glBindVertexArray(vaoIID);

vboIID = glGenBuffers();
glBindBuffer(GL_ARRAY_BUFFER, vboIID);
glBufferData(GL_ARRAY_BUFFER, verticesBuffer, GL_STATIC_DRAW);
glVertexAttribPointer(0, 3, GL_FLOAT, false, 0, 0);
glBindBuffer(GL_ARRAY_BUFFER, 0);
```

```
vboIIDi = glGenBuffers();
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, vboIIDi);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, indicesBuffer,
    GL_STATIC_DRAW);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
```

Izsek kode 3.30: Kreiranje ikozaedra s pomnilnikom oglišč.



Slika 3.12: Ikozaeder z različnimi notranjimi in zunanjimi stopnjami teselacije.

Pri izrisu teseliranega ikozaedra najprej izbrišemo trenutni pomnilnik

barve in globine ter omogočimo senčilniški program za teselacijo. Dodamo še rotacijo za nazornejši prikaz. Pridobimo matriko projekcije in matriko za transformacijo iz koordinatnega sistema modela v koordinatni sistem pogleda. Iz druge matrike, poleg tega da jo podamo senčilniku, naredimo še eno novo matriko, in sicer tako, da odstranimo zadnji stolpec in zadnjo vrstico in tako dobimo matriko za preslikavo normal.

Senčilniškemu programu podamo notranjo in zunanjo stopnjo teselacije. Notranja stopnja nam pove, koliko trikotnikov bo znotraj vsakega trikotnika, ki ga definiramo na začetku. Zunanja stopnja teselacija pa pove, na koliko enakovrednih delov bomo razdelili posamezno stranico vsakega trikotnika definiranega pred teselacijo (Slika 3.12). Podamo mu še projekcijsko matriko, matriko za preslikavo normal in matriko za preslikavo iz koordinatnega sistema modela v koordinatni sistem pogleda. Na koncu podamo še potrebne parametre za osnovno razpršeno osvetljevanje[20].

Pred izrisom moramo definirati še, koliko oglišč bo senčilnik teselacije vzel za eno krpo, torej nastavimo parameter GL_PATCH_VERTICES na 3. Tako dosežemo, da bo senčilnik teselacije za eno krpo vzel en trikotnik.

Na koncu izrišemo ikozaeder, vendar moramo pri metodi za izris nastaviti vrsto primitivov, s katerimi se izriše objekt na GL_PATCHES in s tem omogočimo teselacijo na trenutnem objektu[20] (Izsek kode 3.31).

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

glUseProgram(tessellation2ShaderProgram);

float HalfWidth = 0.6f;
float HalfHeight = HalfWidth * SCREEN_HEIGHT / SCREEN_WIDTH;
glFrustum(-HalfWidth, +HalfWidth, -HalfHeight, +HalfHeight, 5,
    150);
GLU.gluLookAt(0.0f, 0.0f, -15.0f, 0.0f, 0.0f, 0.0f, 0.0f, 1.0f,
    0.0f);

float radiansPerMicrosecond = 0.05f;
float theta = elapsedMicroseconds * radiansPerMicrosecond;
```

```
glRotatef(theta, 0, 1, 0);

FloatBuffer modelViewMatrix = asFloatBuffer(new float[16]);
glGetFloat(GL_MODELVIEW_MATRIX, modelViewMatrix);

FloatBuffer projectionMatrix = asFloatBuffer(new float[16]);
glGetFloat(GL_PROJECTION_MATRIX, projectionMatrix);

Matrix4f modelViewMatrix4 = new Matrix4f();
modelViewMatrix4.load(modelViewMatrix);
modelViewMatrix.flip();

Matrix3f normalMatrix = new Matrix3f();
normalMatrix.m00 = modelViewMatrix4.m00;
normalMatrix.m01 = modelViewMatrix4.m01;
normalMatrix.m10 = modelViewMatrix4.m10;
normalMatrix.m11 = modelViewMatrix4.m11;
normalMatrix.m02 = modelViewMatrix4.m02;
normalMatrix.m12 = modelViewMatrix4.m12;
normalMatrix.m22 = modelViewMatrix4.m22;
normalMatrix.m20 = modelViewMatrix4.m20;
normalMatrix.m21 = modelViewMatrix4.m21;

FloatBuffer buffer = BufferUtils.createFloatBuffer(9);
normalMatrix.store(buffer);
buffer.flip();

glUniform1f(tess2LevelInnerUniform, tessLevelInner);
glUniform1f(tess2LevelOuterUniform, tessLevelOuter);

glUniformMatrix4(tess2ProjectionMatUniform, false,
    projectionMatrix);
glUniformMatrix4(tess2ModelViewMatUniform, false,
    modelViewMatrix);
glUniformMatrix3(tess2NormalMatUniform, false, buffer);

glUniform3f(tess2LightPosUniform, 0.25f, 0.25f, 1.0f);
glUniform3f(tess2AmbientMatUniform, 0.04f, 0.04f, 0.04f);
```

```
glUniform3f(tess2DiffuseMatUniform, 0.0f, 0.75f, 0.75f);

glPatchParameteri(GL_PATCH_VERTICES, 3);

glBindVertexArray(vaoIID);
glEnableVertexAttribArray(0);

glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, vboIIDi);
glDrawElements(GL_PATCHES, indicesCountI, GL_UNSIGNED_BYTE, 0);

glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
glDisableVertexAttribArray(0);
glBindVertexArray(0);
glUseProgram(0);

elapsedMicroseconds++;
```

Izsek kode 3.31: Priprava spremenljivk za izris teseliranega ikozaedra.

Iz senčilnika oglišč pošljemo koordinate posameznega oglišča nadzornemu senčilniku teselacije[20] (Izsek kode 3.32).

```
layout (location = 0) in vec3 VertexPosition;

out vec3 vPosition;

void main(){
   vPosition = VertexPosition;
}
```

Izsek kode 3.32: Senčilnik oglišč za teselacijo.

Nadzorni senčilnik teselacije nam omogoča nadziranje in prilagajanje teselatorja, ki je fiksni del grafičnega cevovoda. Najprej definiramo, koliko oglišč bo znotraj ene krpe. S tem dosežemo, da se senčilnik z enim klicem izvede trikrat. Z oglatimi oklepaji pri vhodnih in izhodnih spremenljivkam dosežemo, da lahko prenašamo več spremenljivk naenkrat. V glavni metodi najprej pošljemo koordinate oglišča ocenjevalnemu senčilniku teselacije. Ker se nadzorni senčilnik teselacije izvede trikrat za en klic, potem s spremen-

ljivko *gl_InvocationID* ugotovimo, v katerem prehodu enega klica se trenutno nahajamo, in tako pošljemo naprej koordinate treh oglišč brez uporabe zanke.

Na koncu zapišemo podano notranjo stopnjo teselacijo na prvo mesto v seznam notranjih stopenj teselacije in zapišemo podano zunanjo stopnjo teselacije na prva tri mesta seznama zunanjih stopenj teselacije. Seznam notranjih stopenj teselacije ima vedno samo eno vrednost, medtem ko vrednosti v seznamu zunanjih stopenj predstavljajo zunanjo stopnjo teselacijo za vsako stranico krpe posebej. V našem primeru so tri, ker imamo za krpo trikotnik. Vrednosti v seznamu notranjih in zunanjih vrednosti zapišemo samo enkrat za krpo[19, 20](Izsek kode 3.33).

```
layout(vertices = 3) out;

in vec3 vPosition[];
out vec3 tcPosition[];
uniform float TessLevelInner;
uniform float TessLevelOuter;

void main(){
    tcPosition[gl_InvocationID] = vPosition[gl_InvocationID];

    if (gl_InvocationID == 0) {
        gl_TessLevelInner[0] = TessLevelInner;
        gl_TessLevelOuter[0] = TessLevelOuter;
        gl_TessLevelOuter[1] = TessLevelOuter;
        gl_TessLevelOuter[2] = TessLevelOuter;
    }
}
```

Izsek kode 3.33: Nadzorni senčilnik teselacije.

Glavna naloga ocenjevalnega senčilnika teselacije je, da premakne generirane točke na želene položaje in se zažene za vsako generirano točko posebej. Določimo, da generiramo trikotnike, da imajo generirane točke enako razdaljo med seboj in da se točke sestavijo v trikotnik, orientirane v nasprotni smeri urinega kazalca. Vgrajena spremenljivka *gl_TessCoord* nam predstavlja trenutno krpo. Če je trenutna krpa trikotnik, nam ta spremenljivka poda tri-

kotnik v težiščnem koordinatnem sistemu, v katerem je lega posamezne točke določena kot masno središče mas, ki se nahajajo v ogliščih trikotnika[21].

Izračunamo nove koordinate generiranih točk, jih seštejemo in normaliziramo. To naredimo zaradi tega, da nove točke porinemo na površino ikozaedra. Senčilniku geometrije pošljemo točke krpe, ki sedaj vsebuje stare in novo generirane točke in koordinate krpe v težiščnem koordinatnem sistemu[19, 20] (Izsek kode 3.34).

```
layout(triangles, equal_spacing, cw) in;
in vec3 tcPosition[];
out vec3 tePosition;
out vec3 tePatchDistance;
uniform mat4 Projection;
uniform mat4 Modelview;

void main()
{
    vec3 p0 = gl_TessCoord.x * tcPosition[0];
    vec3 p1 = gl_TessCoord.y * tcPosition[1];
    vec3 p2 = gl_TessCoord.z * tcPosition[2];
    tePatchDistance = gl_TessCoord;
    tePosition = normalize(p0 + p1 + p2);
    gl_Position = Projection * Modelview * vec4(tePosition, 1);
}
```

Izsek kode 3.34: Ocenjevalni senčilnik teselacije.

Senčilnik geometrije v našem primeru uporabljamo zaradi nazornejše predstavitve teselacije, zato senčilnik izračuna po eno normalo za vsak trikotnik in poudari robove ikozaedra tako, da so robovi osnovnih trikotnikov bolj poudarjeni kot robovi generiranih trikotnikov. Najprej izračunamo normalo za primitiv. Za izris poudarjenih robov senčilnik pošlje cevovodu za vsako ogljišče poseben vektor s tremi komponentami, ki predstavlja razdaljo od roba. Te razdalje se avtomatsko interpolirajo pri rasterizaciji in omogočijo senčilniku fragmentov ugotoviti, koliko je določen fragment oddaljen od najbližjega roba[19, 20] (Izsek kode 3.35).

```
uniform mat4 Modelview;
uniform mat3 NormalMatrix;
layout(triangles) in;
layout(triangle_strip, max_vertices = 3) out;
in vec3 tePosition[3];
in vec3 tePatchDistance[3];
out vec3 gFacetNormal;
out vec3 gPatchDistance;
out vec3 gTriDistance;

void main()
{
    vec3 A = tePosition[2] - tePosition[0];
    vec3 B = tePosition[1] - tePosition[0];
    gFacetNormal = NormalMatrix * normalize(cross(A, B));

    gPatchDistance = tePatchDistance[0];
    gTriDistance = vec3(1, 0, 0);
    gl_Position = gl_in[0].gl_Position; EmitVertex();

    gPatchDistance = tePatchDistance[1];
    gTriDistance = vec3(0, 1, 0);
    gl_Position = gl_in[1].gl_Position; EmitVertex();

    gPatchDistance = tePatchDistance[2];
    gTriDistance = vec3(0, 0, 1);
    gl_Position = gl_in[2].gl_Position; EmitVertex();

    EndPrimitive();
}
```

Izsek kode 3.35: Senčilnik geometrije za teselacijo.

V senčilniku fragmentov izračunamo preprosto razpršeno osvetljevanje, oddaljenost do najbližjega rob generiranega trikotnika in oddaljenost do najbližjega roba osnovnega trikotnika. Končna barva je določena kot zmnožek med utežjo najbližjega roba generiranega trikotnika, utežjo najbližjega roba osnovnega trikotnika in barve, ki se izračuna pri osvetljevanju (Izsek kode

3.36). Če je fragment zelo blizu roba in je znotraj meje, da predstavlja rob, potem dobi utež vrednost 0 in tako postane fragment črne barve[20].

```
out vec4 FragColor;
in vec3 gFacetNormal;
in vec3 gTriDistance;
in vec3 gPatchDistance;
in float gPrimitive;
uniform vec3 LightPosition;
uniform vec3 DiffuseMaterial;
uniform vec3 AmbientMaterial;

float amplify(float d, float scale, float offset)
{
    d = scale * d + offset;
    d = clamp(d, 0, 1);
    d = 1 - exp2(-2*d*d);
    return d;
}

void main()
{
    vec3 N = normalize(gFacetNormal);
    vec3 L = LightPosition;
    float df = abs(dot(N, L));
    vec3 color = AmbientMaterial + df * DiffuseMaterial;

    float d1 = min(min(gTriDistance.x, gTriDistance.y),
        gTriDistance.z);
    float d2 = min(min(gPatchDistance.x, gPatchDistance.y),
        gPatchDistance.z);
    color = amplify(d1, 40, -0.5) * amplify(d2, 60, -0.5) *
        color;

    FragColor = vec4(color, 1.0);
}
```

Izsek kode 3.36: Senčilnik fragmentov za teselacijo.

Senčilniki teselacije predstavljajo velik izziv, saj so zadnja inovacija na področji grafike. Literature, s katero bi si lahko pomagali pri razumevanju teselacije, ni veliko. Uporabili smo nove vrste senčilnikov: nadzorni senčilnik teselacije, ocenjevalni senčilnik teselacije in senčilnik geometrije. Za vse nove senčilnike smo morali spoznati njihovo specifično uporabo. Vsi primeri na temo teselacije so bili napisani v drugih programskih jezikih, zato je bilo potrebno ugotoviti uporabo funkcij z LWJGL knjižnico.

Poglavje 4

Zaključek

V diplomski nalogi smo prikazali, da nam programirljivi grafični cevovod ponuja veliko več možnosti kot pa fiksno določeni cevovod. Na ta način so proizvajalci dosegli, da jim ni potrebno vedno dodajati novih funkcionalnosti v svojo arhitekturo, ampak naredijo arhitekturo splošno, ki si jo uporabniki in razvijalci sprogramirajo glede na svoje potrebe. S takim pristopom imajo razvijalci možnost razvoja vedno novih in naprednejših tehnik izrisovanja ter s tem skrbijo za stalno izboljševanje izrisa grafike.

Velik razvoj je doživel tudi programski grafični vmesnik OpenGL, saj je skozi čas opustil veliko vgrajenih funkcionalnosti in s tem ponudil razvijalcu možnost za upravljanje in nadzor le teh. Na primer, sklad transformacijskih matrik je bila vgrajena funkcija vmesnika in sedaj lahko razvijalec sam poskrbi za transformacijske scene. Večina vgrajenih funkcionalnosti se obravnava kot zastarele, a se jih še vedno da uporabljati kljub novejši različici. Ta sprememba je vplivala tudi na pisanje senčilnikov, saj so se tudi pri njih začele opuščati vgrajene funkcionalnosti, kot so dostopi do matrik na matričnem skladu transformacij, dostop do informacijo o lučeh in dostop do koordinat teksture. Te podatke mora sedaj zagotoviti uporabnik sam.

Trend je takšen, da se opuščajo funkcionalnosti, ki so se izvajale samodejno. Uporabnik dobiva čedalje več nadzora nad izrisom grafike. Slaba stran opuščanja take funkcionalnosti je ta, da je razumevanje grafike za začetnika

postalo težje, vendar pa pomaga uporabnikom, ki imajo nekoliko več znanja in izkušenj. Pridobitev opuščanja funkcionalnosti je tudi lažje razhroščevanje in odkrivanje težav, saj se funkcionalnosti ne izvajajo same nekje v ozadju, ampak jih mora implementirati razvijalec sam in tako mora poznati bistveno večji del programa.

Pri izdelavi primerov smo uporabili programski grafični vmesnik OpenGL, ker je podprt na veliko platformah in je prosto dostopen, kar je idealno za začetnike pri razvoju grafičnih aplikacij. LWJGL knjižnica, ki smo jo uporabljali pri naši implementaciji, zagotavlja grafično knjižnico OpenGL za programski jezik Java in bazira na odprtokodni licenci. Knjižnica je zelo stabilna in primerna za razvoj večjih in kompleksnih projektov. Edina večja slaba lastnost knjižnice je pomanjkanje praktičnih primerov in slaba dokumentacija. S tem je začetnikom otežena implementacija grafičnih programov. Velika večina primerov in vaj je napisanih v drugih jezikih. Funkcije so si sicer dokaj podobne, vendar ima LWJGL nekaj specifičnih principov uporabe, s katerimi se mora razvijalec seznaniti. Tudi pri izdelavi naših primerov smo se srečevali z enakimi problemi.

Za vključitev izdelanih senčilnikov v glavno grafično aplikacijo smo morali izdelati metodo, ki ustvari senčilnike, jim pripne kodo, jih prevede in nato pripne senčilnemu programu, ki smo ga morali pred tem že ustvariti.

Izdelali smo pet različnih primerov senčilnikov, in sicer za krožno meglenje, sisteme delcev, Phongovo osvetljevanje, mapiranje senc in teselacijo tridimenzionalnih objektov.

Pri izdelavi krožne zameglitve scene smo izdelali posebno vrsto senčilnika (Poglavje 3.5), ki se izvede na že izrisani sceni. V prvem prehodu skozi grafični cevovod smo shranili sceno v teksturo, ki smo jo nato krožno zameglili in izrisali v drugem prehodu grafičnega cevovoda. Glavni izziv je predstavljala pravilna nastavitev teksture, v katero smo shranili sceno. Ugotoviti je bilo potrebno, kako deluje pomnilnik slike, na katerega smo pripeli našo teksturo, in kako ga uporabiti, da se vanj izriše scena. Ko smo prvič shranili sceno v teksturo, je bila le ta napačno shranjena. Vzrok za to je bilo

neupoštevanje globinske informacije, ki jo omogočimo preko posebnega izrisovalnega pomnilnika. Senčilniku smo določili potrebne spremenljivke in dodali še globalno spremenljivko, da se je izvedlo meglenje ob pritisku določene tipke.

Sistemi delcev (Poglavje 3.6) so lep primer, kako del matematičnih kalkulacij prestaviti s centralne procesne enote na grafično kartico. Ker grafične kartice omogočajo paralelno računanje, se te kalkulacije izvršijo mnogo hitreje. Z našim primerom smo postavili ogrodje, s katerim z uporabo kompleksnejše funkcije in spremembo parametrov simuliramo vodo, ogenj in dim. S tem dosežemo mnogo bolj realistične dinamične efekte pri izrisu grafike. Sistem delcev smo realizirali s pomnilnikom oglišč, s katerim smo poenostavili metodo za izris, vendar smo pri tem morali ugotoviti pravilno uporabo pomnilnikov oglišč. Ključno je bilo pravilno posredovanje matrike senčilniku oglišč, saj se jih v LWJGL knjižnici podaja kot pomnilnik števil.

Phongov model osvetljevanja (Poglavje 3.7) je zelo pogosto uporabljen model osvetljevanja, ki je zelo dober približek realnosti. Pri izdelavi senčilnika moramo biti pozorni, da pravilno opravimo zapleten izračun osvetljevanja. Izziv pri implementaciji so predstavljale luči, saj so se le te premikale s kamero, zato smo morali njihov položaj pri vsakem izrisu ponovno transformirati v trenutno projekcijo kamere. Pri uporabi več luči je potrebno izdelati funkcijo, ki poskrbi za upad jakosti svetlobe posamezne luči z njeno oddaljenostjo, da ne dobimo popolnoma svetle scene. Osvetljevanje lahko dopolnimo s sencami ter tako še izboljšamo realistično podobo.

Mapiranje senc (Poglavje 3.8) predstavlja računsko zelo učinkovit algoritem za dodajanje senc v sceni, vendar pa proizvede ostre sence, ki jih lahko z dodatnimi izboljšavami odpravimo. Izdelava senčilnika in implementacija v aplikacijo se je izkazala za zelo zapleteno. Izziv so predstavljali nastavitev teksture, ohranjanje pravih matrik za transformiranje oglišč v projekcijo luči in uporaba funkcij z LWJGL knjižnico.

Senčilniki za teselacijo (Poglavje 3.9) so zelo nova tehnologija, ki bo počasi prešla v večjo uporabo, je pa zahtevnejša od predhodno predstavljenih (nove

vrste senčilnikov). V našem primeru smo z uporabo teselacijskih senčilnikov iz ikozaedra naredili približek krogle. Ročno smo spreminjali stopnji teselacije. V današnjih grafičnih pogonih sta stopnji teselacije pogojeni z oddaljenostjo kamere od objekta, saj s tem dosežemo povečevanje detajlov tako, da se objektu približujemo. Sam objekt je lahko dosti bolj enostaven in ga naredimo bolj detajlnega s teselacijo, ko se približamo. Tako zmanjšamo število oglišč na zaslonu, pohitrimo izrisovanje grafike in uporabimo manj zapletene tridimenzionalne modele. Glavni izziv je bila izdelava senčilnikov drugih vrst, saj se je bilo potrebno spoznati z njihovo specifično izdelavo.

Pri izdelavi primerov senčilnikov smo bili postavljeni pred izzive, kot je vključevanje senčilnikov v igrico in druge podporne grafične aplikacije. Izdelanim snečilnikom je bilo potrebno zagotoviti pravilne spremenljivke, teksture in druge kompleksne strukture. Velika slabost grafičnih aplikacij razvitih v OpenGL je njihovo razhroščevanje, saj za njih ni podanega nekega splošno namenskega urejevalnika, kot je to pri DirectX. Stanje se je izboljšalo z opuščanjem samodejnih funkcionalnosti, vendar je prostora za izboljšave še veliko, na primer z nastankom splošnega urejevalnika. Nekaj poizkusov v tej smeri je bilo, vendar so vsi zamrli in danes obstajajo le za stare različice OpenGL in GLSL, ki uporabljata samo senčilnike oglišč in senčilnike fragmentov.

Vsi izdelani senčilniki v diplomskem delu so implementirani v grafičnih aplikacijah, ki smo jih izdelali sami in proizvajajo želene efekte.

Senčilnik za krožno meglenje, sistema delcev in Phongovega osvetljevanja smo implementirali v igrico, ki smo jo razvili pri predmetu Računalniška grafika in tehnologija iger. Senčilnik mapiranja senc in teselacije smo vsakega posebej implementirali v specifično aplikacijo. Podobni senčilniki se uporabljajo v različnih grafičnih pogonih, grafičnih aplikacijah in igricah, kar je seveda *know-how* izdelovalcev grafičnih aplikacij in niso prosto dostopni.

Danes skoraj ni več uporabe fiksnega cevovoda, saj ga je pri razvoju kompleksnih grafičnih aplikacijah zamenjal bolj zmogljiv programirljiv cevovod. Vidimo, da je v zadnjih letih realističnost računalniške grafike zelo napredo-

vala, kljub temu da se funkcionalnost grafičnih kartic ni veliko spremenila, saj senčilniki omogočajo razvijalcem, da vodijo razvoj računalniške grafike. Vsak dan se pojavljajo novi in boljši načini uporabe senčilnikov. Prav tako tudi niso omejeni na področje grafike, ampak se lahko uporabljajo za reševanje kompleksnih matematičnih problemov, ki omogočajo visoko stopnjo paralelnosti. Razvoj samega grafičnega cevovoda še ni zamrl, saj smo bili leta 2010 priča podpori za teselacijo. Z razvojem računalništva se bodo odprle nove možnosti za razvoj novih izrisovalnih tehnik, novih načinov uporabe že poznanih izrisovalnih tehnik in grafičnih pogonov, ki bodo zagotavljali še bolj realističen izris grafike.

Literatura

[1] Wolfgang F. Engel, Direct3D ShaderX Vertex and Pixel Shader Tips and Tricks, Wordware Publishing Inc. Plano, Texas, Združene države Amerike, 2002.

[2] Miller Mattson, OpenGL 4.2 API Reference Card, Khronos Group, 2011. Dostopno na:
http://www.khronos.org/files/opengl42-quick-reference-card.pdf

[3] Randi J. Rost, Bill Licea-Kane, OpenGL Shading Language Third Edition, Addison-Wesley, Združene države Amerike, 2012.

[4] Pat Brown, ARB_tessellation_shader, 2010. Dostopno na:
http://www.opengl.org/registry/specs/ARB/tessellation_shader.txt

[5] Pat Brown, Barthold Lichtenbelt, ARB_geometry_shader4, 2008. Dostopno na:
http://www.opengl.org/registry/specs/ARB/geometry_shader4.txt

[6] Pat Brown, ARB_transform_feedback3, 2010. Dostopno na:
http://www.opengl.org/registry/specs/ARB/transform_feedback3.txt

[7] Lighthouse 3D, GLSL Core Tutorial, 2012. Dostopno na:
http://www.lighthouse3d.com/tutorials/glsl-core-tutorial/

[8] David Wolff, OpenGL 4.0 Shading Language Cookbook, Packt Publishing, 2011.

[9] The Coding Universe, Oskar Veerhoek, 2012. Dostopno na: http://thecodinguniverse.com/

[10] LWJGL - Lightweight Java Game Library, 2012. Dostopno na: http://lwjgl.org/

[11] Radial Blur Filter, 2008. Dostopno na: http://www.gamerendering.com/2008/12/20/radial-blur-filter/

[12] OpenGL 3.3 Tutorials, 2012. Dostopno na: http://www.opengl-tutorial.org/

[13] ShadowMapping with GLSL, Fabien Sanglard, 2012. Dostopno na: http://fabiensanglard.net/shadowmapping/index.php/

[14] Graphics for Games, Newcastle University, 2012. Dostopno na: http://research.ncl.ac.uk/game/mastersdegree/graphicsforgames/

[15] The OpenGL Reference Pages, Silicon Graphics International Corp., 2012. Dostopno na: http://www.opengl.org/sdk/docs/

[16] Shader, Wikipedia, 2012. Dostopno na: http://en.wikipedia.org/wiki/Shader

[17] Tesselation Shaders, Mike Baily, Oregon State University, 2012. Dostopno na: http://web.engr.oregonstate.edu/ mjb/cs519/Handouts/tessellation.6pp.pdf

[18] doc. dr. Matija Marolt, Računalniška grafika in tehnologija iger - prosojnice, Univerza v Ljubljani, Fakulteta za računalništvo in informatiko, 2012.

[19] Anton Gerdelan, OpenGL 4 Mini-Tutorials, 2012. Dostopno na: http://antongerdelan.net/opengl/index.php?title=Main_Page

[20] Philip Rideout, Triangle Tessellation with OpenGL 4.0, 2010. Dostopno na:
http://prideout.net/blog/?p=48#levels

[21] Težiščni koordinatni sistem, Wikipedia, 2012. Dostopno na:
http://sl.wikipedia.org/wiki/Težiščni_koordinatni_sistem