SVEUČILIŠTE U ZAGREBU

FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 1593

Programirljivo grafičko sklopovlje

Krešimir Jozić

Zagreb, srpanj 2006.

Sadržaj

1. Uvod....1
2. Povijesni pregled....2
 2.1. Razvoj grafičkog sklopovlja....2
 2.2. Programska sučelja za računalnu grafiku....3
  2.2.1. Glide....3
  2.2.2. OpenGL....4
  2.2.3. DirectX....4
3. Neprogramirljivo sklopovlje....6
 3.1. Ulaz u grafički cjevovod....7
 3.2. Transformacije vrhova....7
 3.3. Sakupljanje primitiva i rasterizacija....8
 3.4. Preslikavanje tekstura na fragmente i bojanje....10
 3.5. Rasterske operacije....10
  3.5.1. Spremnik boje (engl. color buffer)....11
  3.5.2. Spremnik dubine (engl. depth buffer, Z buffer)....11
  3.5.3. Spremnik maske (engl. stencil buffer)....12
  3.5.4. Akumulacijski spremnik (engl. accumulation buffer)....12
  3.5.5. Stereo spremnici (engl. stereo buffers)....12
  3.5.6. Dodatni spremnici (engl. auxiliary buffers)....12
4. Programirljivo grafičko sklopovlje....17
 4.1. Programirljivi grafički procesori....20
 4.2. Procesor vrhova....21
 4.3. Procesor fragmenata....30
5. Jezici visoke razine....35
 5.1. Programski jezik Cg (i HLSL)....36
  5.1.1. Komentari....38
  5.1.2. Identifikatori....38
  5.1.3. Varijable....38
  5.1.4. Konstante....40
  5.1.5. Strukture....40
  5.1.6. Polja....40
  5.1.7. Pretvorba tipa....40
  5.1.8. Izdvajanje, zamjena i maskiranje komponenti vektora....41
  5.1.9. Funkcije....42
  5.1.10. Sučelja....44
  5.1.11. Vezivanje varijabli uz registre....44
  5.1.12. Uniformni parametri....45
  5.1.13. Uzorkovanje tekstura....46
  5.1.14. Operatori....46
  5.1.15. Uvjeti....47
  5.1.16. Petlje....47
  5.1.17. Ugrađene funkcije....49
 5.2. Programski jezik GLSL....52
  5.2.1. Predprocesor....54
  5.2.2. Komentari....54
  5.2.3. Identifikatori....54
  5.2.4. Varijable....55
  5.2.5. Konstante....55
  5.2.6. Strukture....55
  5.2.7. Polja....55

5.2.8. Pretvorba tipa......55
5.2.9. Izdvajanje, zamjena i maskiranje komponenti vektora......56
5.2.10. Funkcije......56
5.2.11. Sučelja......56
5.2.12. Vezivanje varijabli uz registre......56
5.2.13. Uzorkovanje tekstura......58
5.2.14. Operatori......58
5.2.15. Uvjeti......58
5.2.16. Petlje......58
5.2.17. Ugrađene varijable, konstante i strukture......58
5.2.18. Ugrađene funkcije......62
6. Programski alati......66
6.1. Kratki opisi programskih alata......66
6.1.1. NVPerfKit......66
6.1.2. NVPerfHUD......66
6.1.3. NVShaderPerf......67
6.1.4. DXT utilities......67
6.1.5. BuGLe......67
6.1.6. GLSL Validate......67
6.1.7. GIMP normalmap plugin......68
6.1.8. FX Composer......68
6.1.9. RenderMonkey......68
6.2. Detaljni opisi nekoliko programskih alata......68
6.2.1. NVShaderPerf......69
6.2.2. FX Composer......75
6.2.3. RenderMonkey......82
7. Programski primjeri......86
7.1. Preslikavanje izbočina......87
7.1.1. Cg......87
7.1.2. GLSL......88
7.1.3. Rezultati......89
7.2. Valovita deformacija......90
7.2.1. Cg......91
7.2.2. GLSL......92
7.2.3. Rezultati......93
7.3. Izrezivanje traka......94
7.3.1. GLSL......94
7.3.2. Cg......95
7.3.3. Rezultati......96
7.4. Miješanje tekstura......97
7.4.1. GLSL......97
7.4.2. Cg......98
7.4.3. Rezultati......100
7.5. Ostali primjeri......101
8. Programski zadatak......105
8.1. Inkscape......105
8.2. Blender......106
8.3. OGRE......107
8.4. Postupak izrade programskog rješenja......114
9. Zaključak......123
10. Dodatak A – postavljanje programa za sjenčanje u asemblerskom obliku......124
11. Dodatak B – postavljanje programa za sjenčanje u programskom jeziku Cg......132

12. Dodatak C – postavljanje programa za sjenčanje u programskom jeziku GLSL............138
13. Dodatak D – SDL..145
13.1. Postavljanje uređaja za prikaz...146
13.2. Učitavanje slika...147
13.3. Reakcija na događaje i zamjena dvostrukog spremnika..................................148
13.4. Programski odsječak..148
14. Literatura...151
15. Sažetak...152

Popis tablica

Tablica 1: Nekoliko primjera raspodjela zauzeća memorije piksela......................................13
Tablica 2: Tipični parametri procesora vrhova treće generacije..22
Tablica 3: Atributni registri procesora vrhova..23
Tablica 4: Parametarski registri procesora vrhova..24
Tablica 5: Pristup parametrima materijala..24
Tablica 6: Pristup parametrima svjetla...25
Tablica 7: Pristup matricama...25
Tablica 8: Izlazni registri procesora vrhova..26
Tablica 9: Skup naredbi procesora vrhova..27
Tablica 10: Tipični parametri procesora fragmenata treće generacije...................................30
Tablica 11: Atributni registri procesora fragmenata..31
Tablica 12: Izlazni registri procesora fragmenata..31
Tablica 13: Skup naredbi procesora fragmenata..31
Tablica 14: Popis ugrađenih funkcija programskog jezika Cg...49
Tablica 15: Popis ugrađenih funkcija programskog jezika GLSL..62
Tablica 16: Specifikacije za nekoliko grafičkih kartica tvrtke NVIDIA................................86
Tablica 17: Rezultati simulacija programa za sjenčanje fragmenata programskog primjera za preslikavanje izbočina...89
Tablica 18: Rezultati simulacija programa za sjenčanje fragmenata programskog primjera za valovitu deformaciju..93
Tablica 19: Rezultati simulacija programa za sjenčanje fragmenata programskog primjera za izrezivanje traka...96
Tablica 20: Rezultati simulacija programa za sjenčanje fragmenata programskog primjera za miješanje tekstura (prvi prolaz)..100
Tablica 21: Rezultati simulacija programa za sjenčanje fragmenata programskog primjera za miješanje tekstura (drugi prolaz)..100
Tablica 22: Korištene funkcije u ARB i OpenGL 2.0 notaciji...144

Popis slika

Slika 1: Grafički protočni sustav....6
Slika 2: Faze kroz koje prolaze geometrijski elementi....10
Slika 3: Faza rasterskih operacija....15
Slika 4: Shema programirljivog sklopovlja....18
Slika 5: Shema procesora vrhova i fragmenata....19
Slika 6: Korištenje Cg-a iz grafičke aplikacije....37
Slika 7: Korištenje GLSL programa u grafičkoj aplikaciji....53
Slika 8: Osnovni raspored prozora FX Composer-a....75
Slika 9: Prozor za prikaz teksture....76
Slika 10: Prozor za prikaz materijala....76
Slika 11: Prozor za prikaz scene....76
Slika 12: Prozor za prikaz grafa scene....77
Slika 13: Prozor za poruke prevodioca....77
Slika 14: Svojstva materijala i scene....78
Slika 15: Integrirani NVShaderPerf....78
Slika 16: Osnovni izgled RenderMonkey-a....82
Slika 17: Alatna traka....83
Slika 18: Poruke prevodioca....83
Slika 19: Srednji dio prozora....83
Slika 20: Prozor s grafom scene....84
Slika 21: Sažeti graf scene – prikladan za 3D umjetnika....85
Slika 22: Prozor za podešavanje atributa materijala....85
Slika 23: Rezultati programa za preslikavanje izbočina....90
Slika 24: Rezultati programa za valovitu deformaciju....93
Slika 25: Rezultati programa za izrezivanje traka....96
Slika 26: Rezultati programa za miješanje tekstura....101
Slika 27: Programi RealPlayer i GIMP prelaze preko brida kocke (dvije susjedne radne površine)....102
Slika 28: Automatski posloženi prozori....103
Slika 29: Kruženje između otvorenih prozora....103
Slika 30: Program Totem Movie Player prikazuje film u poluprozirnom prozoru preko programa Firefox....104
Slika 31: Prikaz Lindenmayer-ovih (L) sustava u Inkscape-u....106
Slika 32: Blender s modelom scene, grafom scene i vrijednostima materijala....107
Slika 33: oFusion dodatak za 3D Studio Max koji u pozadini koristi OGRE....108
Slika 34: Sučelje OGREExport-a....113
Slika 35: Sučelje dotScene Exporter-a....114
Slika 36: Logo FER-a u vektorskom obliku sa slikom kao predloškom....115
Slika 37: Most u vektorskom obliku....116
Slika 38: Logo FER-a pretvoren u 3D objekt....116
Slika 39: Most pretvoren u 3D objekt....117
Slika 40: Uvodni prozor OGRE-a za podešavanje iscrtavanja....120
Slika 41: Scena s logom FER-a....121
Slika 42: Scena s mostom....121
Slika 43: Programski primjer koji koristi programe za sjenčanje u asemblerskom obliku....131

1. Uvod

Na početku će biti dan kratak povijesni pregled grafičkog sklopovlja od “fiksnog” (neprogramirljivog), pa sve do programirljivog. Programirljivo sklopovlje biti će opisano iz perspektive programera, dakle shemom i asemblerskim naredbama.
Zatim slijedi objašnjenje programskih jezika visoke razine koji se koriste za programiranje grafičkog sklopovlja.

Praktični dio sastoji se od pregleda gotovih programskih alata za izradu i analizu programa za sjenčanje. Pri tome će se koristiti programski alati koji su otvorenog koda (engl. open source) ili nisu otvorenog koda, ali su besplatni.

Na kraju će biti demonstriran program napisan u programskom jeziku C++ koji koristi jezike za sjenčanje Cg i GLSL.

Kako grafičko sklopovlje jako brzo napreduje, nemoguće je davati konkretne primjere vezane uz više od jedne generacije grafičkih procesora. Zbog toga će se svi konkretni primjeri vezani uz grafičko sklopovlje odnositi na grafičku karticu Gainward 2400/Ultra GS GLH, s grafičkim procesorom GeForce 6800 GT (ukoliko nije drugačije naznačeno).
.
Korišteni operacijski sustavi su:
- Microsoft Windows XP, 32 bitna verzija
- Novell openSUSE Linux 10.0, 64 bitna verzija
- Novell openSUSE Linux 10.1, 64 bitna verzija

2. Povijesni pregled

Da bi se razjasnila važnost programirljivog grafičkog sklopovlja potrebno je izložiti kratak povijesni pregled razvoja grafičkog sklopovlja.
Iako profesionalno grafičko sklopovlje postoji već preko 30 godina, ovdje će se razmatrati grafičko sklopovlje dostupno običnom korisniku, tj. ono na osobnim računalima i to isključivo ono koje podržava 3D grafiku.

2.1. Razvoj grafičkog sklopovlja

Moglo bi se reći da doba 3D grafike na osobnim računalima počinje 1995. godine. Te godine je tvrtka S3 napravila grafičku karticu po imenu ViRGE (**Vi**rtual **R**eality **G**raphics **E**ngine). Tada je to bila najbrža grafička kartica koja je podržavala VGA standard i imala mogućnosti ubrzavanja 2D i 3D grafike. No, bila je proglašena "prvim grafičkim 3D deceleratorom". Iako je elementarna podrška za 3D grafiku bila brža od izračuna na centralnom procesoru (CPU), naprednije mogućnosti kao što su bilinearno filtriranje i efekti magle bili su dosta sporiji od izračuna na centralnom procesoru..
Isto tako, nije imala ni podršku za OpenGL, pa nije bila primjenjiva u profesionalnoj grafici.

Tvrtka 3dfx 1996. godine izdaje grafičku karticu po imenu Voodoo. Ta grafička kartica se smatra za prvu pravu 3D ubrzivačku karticu na osobnim računalima. No, i ona je imala nekoliko problema. Bila je isključivo namijenjena za 3D grafiku, pa se za prikaz 2D grafike morala posebnim kabelom spojiti na dodatnu 2D grafičku karticu. Osim toga, dosta je loše filtrirala teksture i imala je problema s prikazom boja u računalnim igrama.

Sve naredne generacije grafičkih kartica do 2001. godine nastavile su trend sklopovskog implementiranja funkcija, uz iznimku složenijih poslova koji su se obavljali u pogonskim programima (engl. driver).

Prednost sklopovski implementiranih funkcija je brzina, a nedostatak je nefleksibilnost. Programirljivo sklopovlje gotovo je uvijek sporije zbog nekoliko razloga: dohvat naredbe, dekodiranje naredbe, dohvat operanada itd. No, koliko god da je neprogramirljivo sklopovlje brže, kod složenijih zadataka brže je programirljivo sklopovlje - upravo zbog fleksibilnosti. Ako, npr. imamo neku scenu koja zahtijeva 5 prolaza izračunavanja na neprogramirljivom sklopovlju, to se može napraviti u svega 2 do 3 prolaza na programirljivom sklopovlju. Čak i ako je programirljivo sklopovlje sporije 30%, ukupno će biti brže oko 20%.

Zbog toga su već 1999. godine grafičke kartice prestale koristi isključivo "čvrsto" sklopovski implementirane funkcije. Te godine sve su se funkcije vezane uz 3D grafiku u potpunosti prebacile na grafičku karticu. Zbog toga se procesor koji se nalazi na grafičkoj kartici od tada naziva GPU (**G**raphical **P**rocessing **U**nit). Mnoge faze prilikom iscrtavanja 3D scena moraju se barem malo "prilagoditi", tj. više se ne mogu koristiti sklopovski implementirane funkcije koje nisu bar malo prilagodljive (no to još ne znači da su programirljive).

2001. godine izlazi grafička kartica po imenu GeForce 3 koja podržava male programe u geometrijskoj fazi. Ti programi se nazivaju "programi za sjenčanje vrhova" (engl. vertex shader, vertex program). To su vrlo ograničeni programi. Moraju biti vrlo mali i mogu obavljati samo jednostavne aritmetičke operacije.

Već iduće godine izlaze grafičke kartice koje imaju programe za sjenčanje fragmenata (engl. pixel shader, fragment shader, fragment program). Programi za sjenčanje fragmenata ove generacije mogli su raditi ono što su mogli programi za sjenčanje vrhova prethodne generacije, uz dodatak pristupa teksturama. Programi za sjenčanje vrhova prilično su unaprijeđeni u odnosu na prethodnu fazu, no još uvijek nemaju pravu kontrolu toka, odnosno ne podržavaju naredbe skoka (koje se obično nazivaju JMP i sl.). Umjesto toga koriste uvjetno izvršavanje naredbe, tj. naredba ima sufiks koji određuje da li će se izvršiti ako je neki zahtjev ispunjen (npr. ADDNZ – zbroji ako nije nula).

Tijekom iduće dvije godine mogućnosti programa za sjenčanje vrhova i fragmenata povećavale su se. Krajem 2004. godine izlazi grafička kartica po imenu GeForce 6800 koja donosi mnogo poboljšanja.
Sada programi za sjenčanje vrhova i fragmenata podržavaju potpunu kontrolu toka, odnosno podržavaju naredbe skoka. Programi za sjenčanje vrhova sada mogu pristupati teksturama, što omogućava neke vrlo napredne stvari kao što je npr. fizikalno ispravna simulacija vode u stvarnom vremenu.

Od tada su uglavnom poboljšanja bila u vezi frekvencija GPU-a i memorije, kao i povećanju broja cjevovoda.

2.2. Programska sučelja za računalnu grafiku

Programskih sučelja za računalnu grafiku ima nekoliko, no ukratko će biti opisana samo tri: Glide, OpenGL i DirectX (točnije, Direct3D).

2.2.1. Glide

Glide je standard tvrtke 3dfx. To je isključivo vlasnički (engl. proprietary) standard.
Ime Glide je odabrano tako da podsjeća na profesionalni standard OpenGL, ali dovoljno različito da bi se izbjegli sudski sporovi.

Može ga se smatrati ograničenim podskupom OpenGL-a. Taj podskup je odabran tako da se sve funkcije mogu izvesti sklopovski, što je dovelo do mnogih ograničenja. Glavno ograničenje je 16 bitna boja, tj. ukupan broj boja koji se mogao prikazati bio je 65536.

Kako su ostali proizvođači podržavali OpenGL i DirectX i sve se više približavali brzinama Voodoo grafičkih kartica, tvrtka 3dfx je otvorila Glide standard kako bi ga

popularizirala. Taj pokušaj je izveden prekasno, što je dovelo do propasti tvrtke 3dfx.

2.2.2. OpenGL

OpenGL (**Open Graphics Library**) je skup funkcija (API) koje služe za iscrtavanje scene iz osnovnih elemenata (pristup niske razine). Naziva se još i "asembler za grafiku". Nastao je 1992. godine iz sličnog projekta tvrtke Silicon Graphics Incorporated (SGI) po imenu IrisGL.

Glavne karakteristike su mu te što je otvoren standard (Open kod OpenGL) i što radi na gotovo svim platformama (slobodno bi se moglo reći na svim, jer čak i neki modeli mobitela imaju podršku za OpenGL, tj. OpenGL za ugrađene sustave – OpenGL ES). Jedini nedostatak je taj što bez sklopovske podrške radi vrlo sporo.
Postoji i potpuno programska izvedba OpenGL-a po imenu MESA.

Razvoj OpenGL-a nadzire ARB (**A**rchitecture **R**eview **B**oard), odbor koji se sastoji od predstavnika tvrtki koje proizvode grafičko sklopovlje i stručnjaka iz područja računalne grafike.

OpenGL se najčešće koristi kao skup funkcija (varijabli, konstanti itd.) napisanih u programskom jeziku C, no moguće ga je koristiti u skoro svakom novijem programskom jeziku (pa čak i u starijim, npr. Fortran).

Kao primarni jezik za razvoj OpenGL-a odabran je C iz nekoliko razloga. Prvi razlog je mogućnost implementacije na više operacijskih sustava. OpenGL je najčešće implementiran na razini upravljačkih programa, a kako stariji sustavi (kao npr. UNIX) za razvoj jezgre i upravljačkih programa koriste C, OpenGL se vrlo lako ugradi u upravljačke programe.
Drugi razlog je nadogradnja. OpenGL standard se proširuje ekstenzijama. Ekstenzija se definira opisno, a implementacija se prepušta proizvođaču. S programske strane gledano, sve se svodi na dodavanje novih funkcija i konstanti. No, to je lakše napraviti u nekom objektno orjentiranom jeziku. Zašto se onda ne koristi neki objektno orjentirani jezik (npr. C++)? Zbog toga što je glavna prednost objektno orjentiranih jezika ujedno i njihova velika mana, a to je apstrakcija. Netko bi gledajući skup objekata lako mogao ne zapaziti novu funkciju ili dvije (zato što ih ima nekoliko stotina). Kod projekata pisanih u C-u obično se samo izlistaju nove funkcije, tako da ih je nemoguće preskočiti.

Verzija OpenGL-a se označava po principu X.Y, gdje se Y uvećava za 1 kada se skupi dovoljan broj novih ekstenzija. Trenutna verzija je 2.0, a prije nje su bile 1.0, 1.1, 1.2, 1.3, 1.4 i 1.5. Da bi proizvođač mogao reći da njegov proizvod podržava neku verziju OpenGL-a, on mora implementirati sve ekstenzije koje su određene standardom.

2.2.3. DirectX

DirectX se sastoji od nekoliko multimedijskih sučelja (X u imenu DirectX):

- DirectDraw – baratanje 2D rasterskim sadržajem (sada je dio Direct3D-a)
- Direct3D – baratanje 3D primitivima
- DirectInput – interakcija s mišem, tipkovnicom i ostalim ulaznim uređajima

- DirectPlay – podrška za igranje preko mreže
- DirectSound – snimanje i reprodukcija zvuka
- DirectMusic – reprodukcija soundtrack-ova načinjenih s DirectMedia Producer-om
- DirectShow – reprodukcija streaming audia i videa
- DirectSetup – automatska instalacija DirectX komponenti
- DirectX Media Objects – različiti objekti kao npr. koderi, dekoderi, efekti ...

Nama je zanimljiv samo onaj dio vezan uz grafiku - Direct3D.

Kako je uopće došlo do razvoja DirectX-a? Odgovor je jednostavan. OpenGL je bio dostupan samo na velikim, skupim računalima. Zbog toga je Direct3D koristio apstrakciju sklopovlja da bi omogućio izradu aplikacija koje će raditi i na osobnim računalima.

Direct3D se sastoji od 2 dijela:

- HAL (**H**ardware **A**bstraction **L**ayer)
- HEL (**H**ardware **E**mulation **L**ayer)

HAL obavlja apstrakciju određene funkcionalnosti sklopovlja (npr. baratanje spremnikom okvira) koje proizvođači različito ostvaruju.
HEL obavlja emulaciju funkcionalnosti, tj. programski izvodi ono što nije izvedeno sklopovski.

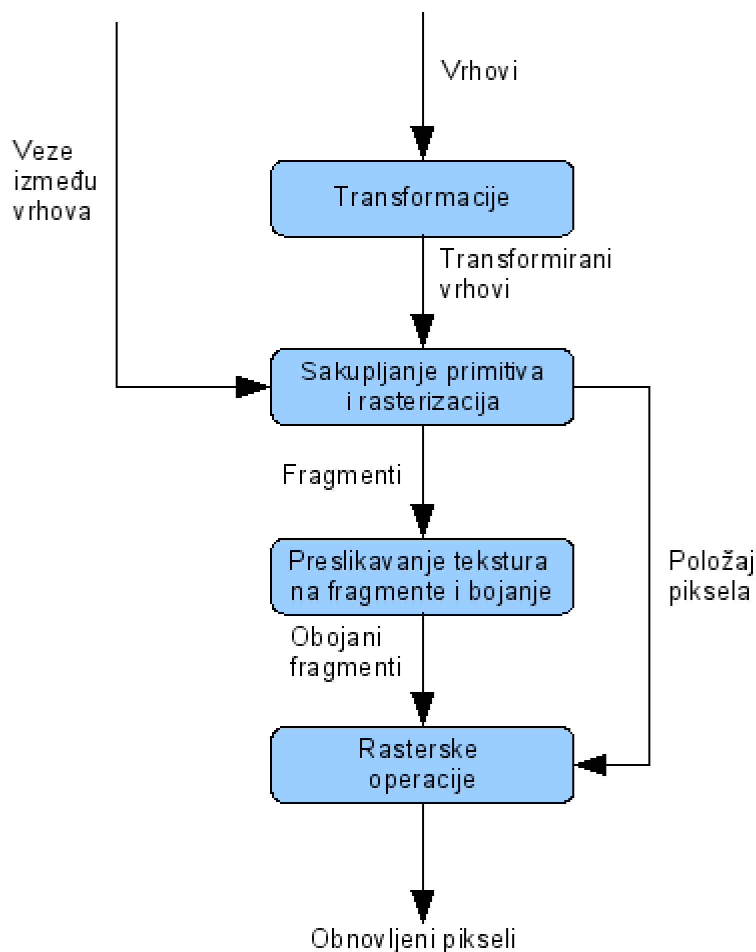
Za razliku od OpenGL-a, DirectX koristi objektno orijentirani pristup – COM (**C**omponent **O**bject **M**odel). Moguće ga je koristiti u nekoliko programskih jezika: C, C++, Visual Basic, C#. Kao što se vidi, podrška za programske jezike puno je manja nego kod OpenGL-a.

DirectX više je popularan od OpenGL-a uglavnom u računalnim igrama zbog toga što koristi apstraktniji pristup. Ima dosta gotovih objekata koji ponekad znaju dosta olakšati posao (npr. učitavanje geometrije scene iz datoteke i sl).

DirectX je definiran od strane Microsoft-a, što znači da je u potpunosti vlasnički standard. To ima nekoliko nedostataka. Nitko ne može utjecati na standard, a nove mogućnosti se pojavljuju tek s novom verzijom DirectX-a. Kod OpenGL-a nove mogućnosti obično dolaze s novom verzijom upravljačkih programa.

3. Neprogramirljivo sklopovlje

Grafički protočni sustav (cjevovod) sastoji se od niza stupnjeva koji obavljaju određene operacije po fiksnom redoslijedu.
Moderne grafičke kartice imaju po nekoliko cjevovoda u geometrijskoj fazi i nešto više u rasterizacijskoj fazi.
U geometrijskoj fazi barata se s vrhovima poligona kojih je puno manje nego fragmenata (piksela) u rasterizacijskoj fazi, pa zbog toga treba i manje cjevovoda.



Slika 1: Grafički protočni sustav.

3.1. Ulaz u grafički cjevovod

Ulaz u grafički cjevovod predstavljaju oni podaci koji dolaze iz aplikacije, a to su:

- vrhovi
- veze među njima
- položaj
- boja
- koordinate teksture itd.

Aplikacije rade na apstraktnom nivou i barataju objektima. Objekti se sastoje od osnovnih geometrijskih elemenata i/ili različitih vrsta krivulja. Elementi se sastoje od vrhova i imaju nekoliko atributa (položaj, boja ...). Osim geometrijskih elemenata imamo još i normale, tangente i sl.
Sve to se nalazi u globalnom virtualnom svijetu (memoriji računala) i mora se transformirati prije nego što se pošalje na zaslon.

3.2. Transformacije vrhova

Ovo je prva faza grafičkog protočnog sustava. Postoji niz transformacija koje se obavljaju nad svakim vrhom:

- transformacija modela – obavlja transformaciju iz prostora objekta (modela) u prostor svijeta
- transformacija pogleda – obavlja transformaciju iz prostora svijeta u prostor oka
- projektivna transformacija – obavlja transformaciju iz prostora oka u omeđeni prostor
- perspektivno dijeljenje – obavlja transformaciju iz omeđenog prostora u normalizirani prostor
- transformacija vidnog polja i dubine – obavlja projekciju iz normaliziranog prostora u prostor prozora na zaslonu

Ova faza naziva se geometrijskom fazom i vrlo je bitna. Stoga će svi dijelovi biti opisani redom.

Prostor objekta ili prostor modela je prostor kojega koristi umjetnik kada radi model objekta. Taj prostor određuje isključivo atribute modela kao što su orjentacija, skala i položaj. Prostori dva različita modela ne moraju nužno biti isti. Prostor objekta služi tomu da bi se održala konzistentnost među točkama modela. Ako npr. imamo cilindar i kuglu, možemo za njih koristiti cilindrični i sferni koordinatni prostor, a možemo ih smjestiti i u kartezijev koordinatni sustav. Koristimo takav koordinatni sustav u kakvom nam je lakše izraditi model.

Transformacija modela koristi se za smještanje objekta u prostor virtualnog svijeta. Ako imamo neki objekt i želimo ga smjestiti u virtualni svijet morat ćemo ga translatirati, skalirati ili rotirati. Ako želimo dva ista predmeta u sceni, uzet ćemo samo jedan model i napraviti transformaciju iz njegovog prostora modela u prostor svijeta dva puta. Time smo uštedili na memoriji, jer koristimo samo jedan objekt, a kao rezultat imamo dva u sceni.

Prostor svijeta koristi se da bi se utvrdio odnos među objektima. U prostoru objekta ne postoji odnos između objekata, cijeli prostor objekta odnosi se samo na jedan objekt. Odnos među objektima može biti njihov položaj, veličina i orjentacija. Da bi to mogli ostvariti prvo moramo odrediti centar svijeta (najčeće sredina scene) i "nebo" (smjer prema gore – najčešće pozitivna os y). Nakon toga objekte smještamo u svijet (scenu) transformacijama modela.

Transformacija pogleda je transformacija koja pomiče oko u prostoru svijeta u ishodište prostora oka i po potrebi ga rotira.

Prostor oka (pogleda) je prostor iz kojega se vidi scena. U ishodištu tog prostora je "oko", odnosno točka iz koje gledamo scenu. U ovom prostoru se obavlja većina proračuna vezanih uz osvjetljenje.

Projektivna transformacija definira način projekcije iz prostora oka u omeđeni prostor. Ta projekcija može biti ortogonalna ili perspektivna.

Omeđeni prostor se nalazi unutar prostora oka i omeđuje onaj dio prostora koje oko zaista može vidjeti (vidno polje). Omeđeni prostor koristi homogene koordinate (x, y, z, w).

Perspektivno dijeljenje se koristi da bi se homogene koordinate prebacile u normalizirani prostor. To se postiže dijeljenjem sa w koordinatom. Kao rezultat dobijemo (x/w, y/w, z/w, 1).

Normalizirani prostor je prostor u kojem se koristi koordinatni sustav uređaja za prikaz. Taj koordinatni sustav je dvodimenzionalan uz dodatak z koordinate koja se obično koristi za određivanje vidljivosti.

Transformacija vidnog polja i dubine koristi se da bi se točke iz normaliziranog prostora prebacile u prostor prozora na zaslonu.

Prostor prozora na zaslonu je prostor u kojemu se koriste koordinate piksela za određivanje položaja točke u prozoru koji se prikazuje na zaslonu.

3.3. Sakupljanje primitiva i rasterizacija

Vrhovi (i njihovi atributi) nakon transformiranja idu u slijedeći stupanj cjevovoda gdje se vrhovi povezuju u geometrijske elemente (sakupljanje primitiva) i nakon toga rasteriziraju.

Kao rezultat faze sakupljanja geometrijskih elemenata možemo dobiti slijedeće elemente:

- točke
- linije
- petlje linija
- trake linija
- trokute
- trake trokuta

- lepeze trokuta
- četverokute
- trake četverokuta
- poligone

Elementi ili dijelovi elemenata će možda biti odbačeni ako se ne nalaze u vidljivom dijelu 3D prostora ili na osnovu toga da li im je lice vidljivo ili ne. Licem se smatra ona strana poligona iz koje ide normala.

Preostali poligoni bivaju rasterizirani. Rasterizacija je proces kojim se određuje koji su pikseli prekriveni geometrijskim elementima. Između broja vrhova poligona i broja piksela koje on prekriva nema korelacije. Neki poligon može biti toliko sitan da ne zauzima niti jedan jedini piksel, a isto tako može zauzeti cijeli zaslon i prekriti preko milijun piksela.

U gornjem odlomku je korišten termin piksel iako je možda bolji termin fragment. Piksel je najmanji element slike koji se može prikazati na zaslonu ili ispisati na pisaču. Piksel je u tom slučaju definiran svojim položajem i bojom. Prije iscrtavanja (ispisa) piksel je zapisan u spremniku okvira (ili memorijskom spremniku, engl. framebuffer) koji, osim položaja i boje, sadrži i još neke atribute (dubina, maska ...).
Dakle, piksel je krajnji rezultat svih transformacija.

Postoji i jedan sličan pojam – fragment. Taj pojam ima nekoliko različitih tumačenja, pa često dovodi do zbrke.
Starije tumačenje (prije programirljivog sklopovlja) je to da je fragment poligon koji prolazi kroz transformacije. Kako je poligon prolazio kroz transformacije, mogao je prekriti nula, jedan ili više piksela. U tom kontekstu pojam fragment ima vrlo logično značenje – mali, nerastavljivi dio koji se transformira. Na neprogramirljivom sklopovlju to je potpuno istinito - poligon se ne može rastaviti na manje dijelove pod našim utjecajem (iako ga sklopovlje rasterizira, mi na to ne možemo utjecati).

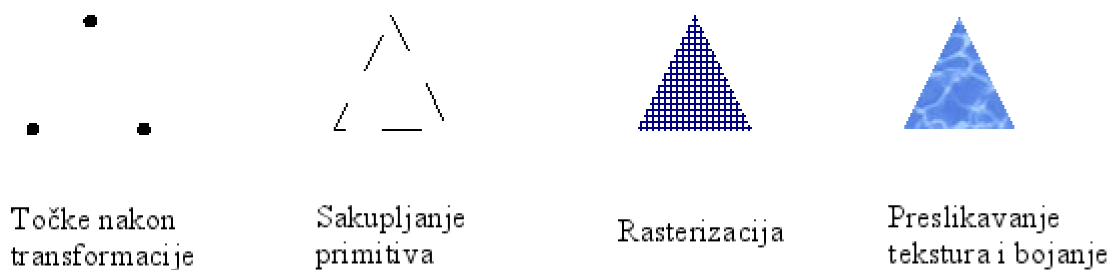
Pojavom programirljivog sklopovlja dobivamo mogućnost utjecanja nakon faze rasterizacije. Zbog toga se značenje termina fragment mijenja. Sada je fragment potencijalni piksel. Nakon faze rasterizacije dobivamo hrpu piksela (točnije fragmenata) koji prolaze dodatne testove i transformacije, te se na kraju zapisuju u spremnik okvira. U ovom slučaju programer može direktno utjecati na testove i transformacije. Zbog toga se elementi dobiveni rasterizacijom više ne zovu pikseli, nego fragmenti. Ako fragment prođe sve testove i bude zapisan u spremnik okvira, on postaje piksel.

I dalje vrijedi da fragment može prekriti nula, jedan ili više piksela. Ako, npr. koristimo antialiasing da bi izgladili rubove, onda scenu uvećamo nekoliko puta. Nakon toga grupiramo fragmente, interpoliramo ih i zapišemo u spremnik okvira. U ovom slučaju jedan piksel se sastoji od onoliko fragmenata koliko je scena uvećana (gruba aproksimacija).
Također, možemo eksplicitno odbaciti fragment. Na taj način se on nikada neće zapisati u spremnik okvira, odnosno neće nikada postati piksel.

Prethodna dva slučaja su krajnosti. U normalnoj situaciji jedan će fragment postati jedan piksel.

3.4. Preslikavanje tekstura na fragmente i bojanje

Poligon nakon rasterizacije na nula ili više fragmenata ulazi u ovu fazu. Ovdje mu se određuje boja na osnovu tekstura i različitih matematičkih transformacija kroz koje prolazi. Ako je potrebno, ovdje mu se mijenja i vrijednost dubine. Na osnovu toga, fragment može biti odbačen da bi se izbjegla obnova piksela u spremniku okvira. Dakle, za svaki ulazni fragment na izlazu dobivamo nula ili jedan fragment.



Slika 2: Faze kroz koje prolaze geometrijski elementi.

3.5. Rasterske operacije

Ovdje se izvode razni testovi i različite operacije vezane uz smještanje fragmenata u spremnik okvira. Da bi se shvatili svi dijelovi ove faze, treba prvo objasniti memorijski spremnik okvira.

Koncept spremnika okvira postoji već od kraja 60-ih godina prošlog stoljeća. Prvo se odnosio na uređaj memorijskih spremnika (engl. framebuffer device). Taj uređaj je apstrakcija grafičkog sklopovlja. Razni proizvođači grafičkog sklopovlja različito izvode neke dijelove. Npr. kod nekog starijeg sklopovlja ne može se pristupiti cjelokupnoj video memoriji odjednom, nego preko segmenata određene veličine koji se pomiču unutar video memorije. Ti segmenti mogu biti različitih veličina (najčešće 4kB i 64kB). Zatim je tu i problem boje. Za svaki kanal boje (RGB ili RGBA) može biti odvojeno od 1 do 32 bita po komponenti boje, odnosno do 128 bita ukupno.
Kako na tržištu ima puno proizvođača i još više modela, jako je teško raditi grafičke programe.
Zbog toga neki operacijski sustavi (prvenstveno Linux) imaju virtualni uređaj koji se zove framebuffer device (u konzoli fbdev, a u X sustavu prozora Xvfb) koji obavlja apstrakciju grafičkog sklopovlja. Taj pristup jako olakšava programiranje grafike, jer imamo samo jedan dobro definiran skup funkcija za rad s grafičkim uređajima.
Opisani spremnik okvira odnosi se na 2D grafiku, pa se stoga samo koristi spremnik boje.

Profesionalna grafička sučelja kao što je OpenGL koriste više spremnika:

- spremnik boje i dvostruki spremnik
- spremnik dubine
- spremnik maske
- akumulacijski spremnik
- stereo spremnici
- dodatni spremnici

3.5.1. Spremnik boje (engl. color buffer)

Namjena mu je pohrana boje piksela. Pretežno se koristi RGBA kolor model, a podržane su i neke njegove varijante (ARGB, ABGR, RGB, BGR). Ponekad se može naći podrška i za YUV kolor model. Za svaku komponentu se koristi cjelobrojna vrijednost bez predznaka.

Broj bitova po pikselu za prikaz boje jako varira. Na monokromatskim sustavima je dovoljan jedan bit po pikselu. Na prvim sustavima u boji koristilo se 2 bita po pikselu, a zatim 4 i 8. Zasloni obično mogu prikazati puno više boja od 2^8, pa se stoga koriste palete. Za svaku vrijednost indeksa palete definirana je prava boja koja se šalje na prikaz.

Tek 16 bitna boja donosi nešto veću kvalitetu. U ovom slučaju se koristi 2 bajta za prikaz svakog piksela, što daje 65536 boja ukupno. Shema koja se često koristi za određivanje RGBA vrijednosti je (u bitovima): 5:6:5:0 i 5:5:5:1.

24 bitna boja se naziva "prava" boja (engl. true color). Korištenjem 3 bajta po pikselu daje oko 16.7 milijuna boja. To je dovoljan broj boja za većinu primjena u 2D grafici, ali za 3D grafiku to još nije dovoljno. Zbog toga se u 3D grafici koristi 32 bitna boja, gdje se 24 bita koriste za prikaz boje, a 8 bita za određivanje prozirnosti.
Sve navedene kombinacije mogu se direktno prikazivati na zaslonu. U većini slučajeva je to u redu, ali ponekad imamo kompleksne scene s velikim dinamičkim rasponom svjetline – HDR (engl. **H**igh **D**ynamic **R**ange). Za prikaz takve scene je i 16.7 milijuna dovoljno, ali nije dovoljno za izračun. Zbog toga se koristi 16 ili 32 bita po komponenti, što daje ukupno 64 ili 128 bita po jednom pikselu. Tako zapisana boja koristi vrijednost zapisanu pomoću pomičnog zareza, pa se ne može direktno prikazati na zaslonu.

Prilikom pripreme slike za prikaz nužno je korištenje spremnika boje, a često i svih ostalih spremnika. Ako se koristi samo jedan spremnik boje može doći do titranja slike na zaslonu.
Da bi se to spriječilo, može se pričekati s iscrtavanjem dok se zraka CRT monitora ne ugasi i počne s vertikalnim povratkom. Tu nastaje problem sporosti osvježavanja slike, jer se mora čekati da se zraka vrati. To vrijeme se može iskorisiti za neki koristan posao. Zbog toga se koristi dodatni spremnik boje u kojeg se piše dok prvi čeka na iscrtavanje, a onda se zamjene.

3.5.2. Spremnik dubine (engl. depth buffer, Z buffer)

Ovaj spremnik koristi se za zapis Z koordinate piksela. Ta koordinata se koristi kod ispitivanja prekrivanja, da se utvrdi koji je piksel vidljiv, a koji nije i na osnovu toga se odbacuju nevidljivi. Ovaj spremnik koristi 16 ili 24 bita po pikselu.

3.5.3. Spremnik maske (engl. stencil buffer)

Pomoću ovog spremnika maskira se dio zaslona. Ako, npr. radimo simulaciju vožnje automobila ili zrakoplova, dio zaslona će uvijek biti prekriven pločom s instrumentima. Ta ploča se uglavnom ne mijenja (osim možda brzinomjera), pa je nije potrebno stalno osvježavati. Zbog toga je prekrijemo određenom maskom koju spremimo u spremnik maske. Nakon toga možemo ostatak zaslona mijenjati po želji, bez straha da će se ploča s instrumentima promijeniti. Ovo ima još jednu prednost, a to je brzina. Pošto se određeni dio zaslona neće iscrtati, nije potrebno izračunavati sve fragmente do kraja. Ako su fragmenti u maskiranom dijelu može ih se samo odbaciti. Kako su maske ustvari vrlo jednostavne, ovaj spremnik koristi do 8 bita po pikselu.

3.5.4. Akumulacijski spremnik (engl. accumulation buffer)

Koristi se uglavnom za određene specijalne učinke (engl. special effect) kao što su dubina polja (engl. depth of field) i zamućivanje objekta koji se brzo kreće (engl. motion blur). Svi ti specijalni učinci se odvijaju u više koraka. Prvo se izračuna scena i spremi u akumulacijski spremnik. Zatim se određeni objekt pomakne (sve ostalo mora ostati isto kao i prije, uključujući boju i položaj). Nakon toga se nove vrijednosti boje pribroje onima u akumulacijskom spremniku. Taj proces se ponavlja nekoliko puta. Na kraju procesa uzimaju se boje iz akumulacijskog spremika, dijele s brojem ponavljanja izračuna i smještaju u spremnik boje.
Kako se postupak ponavlja nekoliko puta, moramo koristiti više bitova za svaku komponentu boje da ne bi došlo do preljeva. Obično se koristi 16 bitova po komponenti boje, što daje ukupno 64 bita po pikselu.

3.5.5. Stereo spremnici (engl. stereo buffers)

Uređaji koji se koriste u svrhu prikaza u virtualnoj stvarnosti najčešće su stereoskopske naočale. Za svako oko se koristi poseban zaslon. Dakle, ista scena se iscrtava za svako oko posebno. Zbog toga OpenGL ima podršku za stereo spremnike koji to omogućavaju.

Već je prije spomenuto da se za ubrzavanje iscrtavanja koriste dvostruki spremnici. Kada se uz njih koriste još i stereo spremnici, broj spremnika poraste na 4. To znači da su zahtjevi za video memorijom veliki. Zbog toga je uporaba stereo spremnika jako rijetka i na većini grafičkih kartica se uopće ni ne ugrađuje.

3.5.6. Dodatni spremnici (engl. auxiliary buffers)

Dodatni spremnici su spremnici boje koji ne služe za iscrtavanje na zaslon, nego za privremenu pohranu podataka da bi se izbjeglo nekoliko prolaza algoritma. Svi dodatni spremnici moraju imati iste atribute piksela, jer koriste isti spremnik dubine.

Atributi piksela su poznati i kao “formati” piksela. Pojam se odnosi na broj bajtova koji svaki piksel zauzima u pojedinom spremniku od kojih se sastoji memorijski spremnik okvira. Nekoliko primjera navedeno je u tablici 1.

Uporaba dodatnih spremnika na prvi pogled nije sasvim jasna. Ako imamo jako kompleksnu scenu sa složenim osvjetljenjem, sjenama, refleksijama itd., morat ćemo napraviti nekoliko prolaza algoritma (do 5-6 na starijem sklopovlju). To se sve može napraviti u svega 2-3 koraka uporabom dodatnih spremnika. U prvom prolazu algoritma izračunavamo normale, dubinu scene za određivanje sjena, reflektirane zrake i svaku od nabrojanih komponenti spremimo u zaseban dodatni spremnik. U drugom prolazu algoritma navedene komponente kombiniramo i zapisujemo u spremnik boje koji se onda šalje na zaslon.
Vremenska ušteda je jako velika, ali je zato i zahtjev za video memorijom jako velik. Zbog toga se na puno grafičkog sklopovlja ni ne ugrađuju.

Tablica 1: Nekoliko primjera raspodjela zauzeća memorije piksela

Ukupan broj bitova boje po pikselu	32	32	16	0	64	128
Dvostruki spremnik	da	da	ne	ne	da	da
Stereo spremnik	ne	ne	ne	ne	ne	ne
Broj bitova za crvenu boju	8	8	5	0	16	32
Broj bitova za zelenu boju	8	8	6	0	16	32
Broj bitova za plavu boju	8	8	5	0	16	32
Broj bitova za alfa komponentu	8	0	0	0	16	32
Broj dodatnih spremnika	4	4	4	4	4	4
Broj bitova Z spremnika	24	24	16	24	24	24
Broj bitova spremnika maske	8	8	0	0	8	8
Broj bitova crvene boje akumulacijskog spremnika	16	16	16	16	16	16

Broj bitova zelene boje akumulacijskog spremnika	16	16	16	16	16	16
Broj bitova plave boje akumulacijskog spremnika	16	16	16	16	16	16
Broj bitova alfa komponente akumulacijskog spremnika	16	16	16	16	16	16

Gornja tablica zatijeva objašnjenje, jer bi netko mogao pomisliti da su neke vrijednosti u tablici neispravne.

Prva stvar koja se može uočiti je ta da niti jedan od formata piksela nema podršku za stereo spremnik. Razlog tomu je što je uporaba stereoskopskih naočala prilično rijetka, pa se ne isplati na sklopovlje ugrađivati podršku za stereo spremnike. Gotovo sigurno je da će se stereoskopske naočale koristiti na specijaliziranom sklopovlju za virtualnu stvarnost koje ima ugrađenu podršku za stereo spremnike.

Prvi i drugi primjer u tablici prikazuju tipične RGBA i RGB formate piksela sa 8 bita po komponenti boje, odnosno 32 i 24 bita ukupno.

Treći primjer prikazuje RGB format piksela koji ima ukupno 16 bita za prikaz boje.

Četvrti format uopće ne koristi spremnik boje (nije greška). Ovaj spremnik je dobar za izračune koji se oslanjaju na akumulacijski spremnik.

Peti i šesti format su HDR. Oni koriste 16 i 32 bitne realne brojeve po komponenti boje, što daje 64 i 128 bitova ukupno.

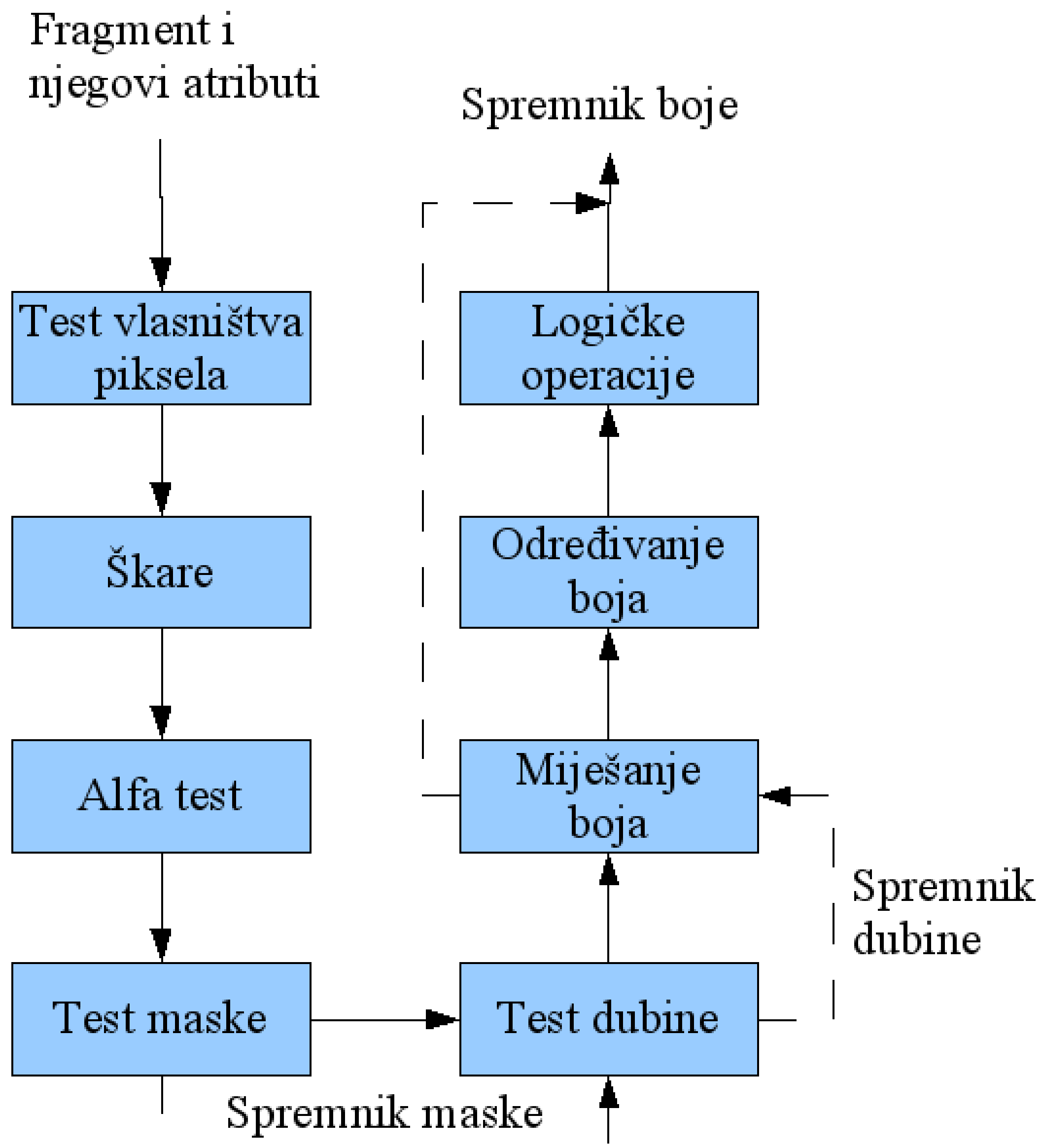
Valja uočiti neke zajedničke karakteristike svih formata piksela:

- 16 bita po komponenti boje akumulacijskog spremnika
- 8 bita spremnika maske (gdje se koristi)
- 16 ili 24 bita spremnika dubine
- 4 dodatna spremnika

Sada kada su opisani svi spremnici od kojih se sastoji spremnik okvira, moguće je opisati fazu rasterskih operacija jer se ona oslanja na te spremnike da bi osvježila vrijednost piksela u spremniku boje. Uz svaki od spremnika je vezan jedan ili više testova koji određuje da li će fragment u njega zapisati. Ako fragment ne prođe neki od testova, on se neće zapisati u spremnik boje, ali postoji mogućnost da će se zapisati u neki od ostalih spremnika.

Ulaz u fazu rasterskih operacija čine fragment i njegovi atributi, a izlaz predstavlja zapis njegove boje u spremniku boje. Osim spremnika boja, neki njegovi atributi ostaju zapisani u ostalim spremnicima.

Moglo bi se reći da je ovo računski najzahtjevnija faza. Prve grafičke kartice samo su ovu fazu izvodile sklopovski, dok se sve ostalo izračunavalo na centralnom procesoru.



Slika 3: Faza rasterskih operacija.

Test vlasništva piksela određuje da li piksel na određenoj lokaciji pripada OpenGL aplikaciji ili ne. Kako se u današnje vrijeme na osobnim računalima sve aplikacije koriste u grafičkom okruženju, potrebno je utvrditi tko je zadužen za iscrtavanje određenog piksela – menadžer prozora ili OpenGL aplikacija.

Škare su test koji se obavlja da bi se izrezao željeni pravokutni dio zaslona. Većinom se cijela scena prikazuje na zaslonu, no ponekad to ne želimo. Npr. ako imamo scenu za koju smo sigurni da neće cijela biti prikazana na zaslonu (spremnik u video memoriji u kojeg iscrtavamo scenu je veći od onog koji služi za prikaz), možemo je vrlo jednostavno ograničiti ovim testom. To se radi tako da se postavi virtualni pravokutnik oko područja koje želimo prikazati i uključimo test. Sve što je izvan virtualnog pravokutnika biva odbačeno. Na taj smo način uštedjeli na broju operacija i vremenu koji bi se potrošilo na izračun cijele scene.

Alfa test je zadužen za odbacivanje ili prihvaćanje fragmenata na osnovu alfa komponente boje. To se radi tako da se zada funkcija testiranja (manje, veće, jednako ...) i konstanta za usporedbu. Ako fragment ne prođe test, odbacuje se bez zapisivanja u bilo koji od spremnika.

Test maske obavlja odbacivanje ili prihvaćanje dijelova scene definiranih maskom. Ako fragment bude prihvaćen, zapisat će se u spremnik maske (ili neće – ovisno o tome kako je zadano) i nastaviti s daljnim testovima. Upotreba testa maske je opisana kod odlomka o spremniku maske.

Test dubine se koristi za odbacivanje nevidljivih poligona (ili u ovom slučaju fragmenata). Svi fragmenti koji su zaklonjeni nekim drugim neće se prikazati. Da li je nešto zaklonjeno ili ne definirano je time da li mu je vrijednost dubine veća ili manja od onog s čime ga uspoređujemo. No, ta definicija se može i obrnuti, pa ćemo u tom slučaju prikazivati piksele koji imaju veću vrijednost dubine (unutrašnjost objekta umjesto vanjštine).

Miješanje boje služi tome da se odredi konačna boja koja će se zapisati u spremnik boje. To se određuje tako da se uzme boja dolaznog fragmenta, boja piksela iz spremnika boje i funkcija za miješanje. Funkcija za miješanje boje ima nekoliko, a najpoznatije su: samo stara boja, samo nova boja, razlika njih dvije, veća ili manja od njih itd.

Određivanje boja (engl. dithering) gotovo se više ne koristi na modernim grafičkim karticama. Taj pojam je vezan uz grafičke kartice i zaslone koji barataju malim brojem boja. Ako grafička kartica (ili zaslon) ima malo boja, onda ona neće moći prikazati neke boje, nego se mora poslužiti trikom. Taj trik je da se nepostojeća boja prikazuje sa postojećima koje se naizmjence slažu. Npr. ako ne možemo prikazati narančastu boju, koristit ćemo crvenu i bijelu koje će se naizmjence koristiti da bi određena ploha izgledala kao da je obojena u narančasto. Taj efekt se vrlo lako uoči iz blizine, no sa veće udaljenosti ne zamjećuje se tako lako.

Logičke operacije se koriste za obavljanje određene logičke operacije nad bojom fragmenta koji dolazi s vrijednošću boje piksela zapisane u spremniku boje.
Ove operacije su definirane isto kao i logičke operacije u programskom jeziku C i gotovo nikad se ne upotrebljavaju.

4. Programirljivo grafičko sklopovlje

Kao što je već rečeno u uvodnom dijelu, razdoblje programirljivog sklopovlja počinje 2001. godine. Tadašnji programi za sjenčanje vrhova i fragmenata bili su prilično ograničeni. Npr. programi za sjenčanje fragmenata smjeli su imati svega nekoliko naredbi i to po određenom redoslijedu. Na početku se moralo pristupati teksturama (ako je to potrebno), a aritmetičke operacije morale su ići na kraj. Moglo se pristupiti maksimalno 4 teksture, a aritmetičkih operacija ukupno je smjelo biti 8.

Dvije godine kasnije počele su se pojavljivati grafičke kartice koje su bile već prilično programirljive, no još uvijek uz neka veća ograničenja (nemogućnost pristupa teksturama iz programa za sjenčanje vrhova, ograničen broj registara, nedostatak potpune kontrole toka). Te godine su programi za sjenčanje počeli dobivati na popularnosti, iako se još uvijek gotovo nigdje nisu upotrebljavali (osim možda u profesionalnim programima za modeliranje i animaciju).

Tek krajem 2004. godine dostignuta je potpuna razina programirljivosti, uz jedno logično ograničenje, a to je broj naredbi. Broj naredbi ne smije biti prevelik, jer bi se iscrtavanje puno usporilo, tako da nam specijalizirano grafičko sklopovlje ne bi bilo od koristi.

Kao što se vidi iz slike 4, princip programirljivosti vrlo je jednostavan – na postojeće sklopovlje dodaju se dva procesora: jedan za vrhove i jedan za fragmente.

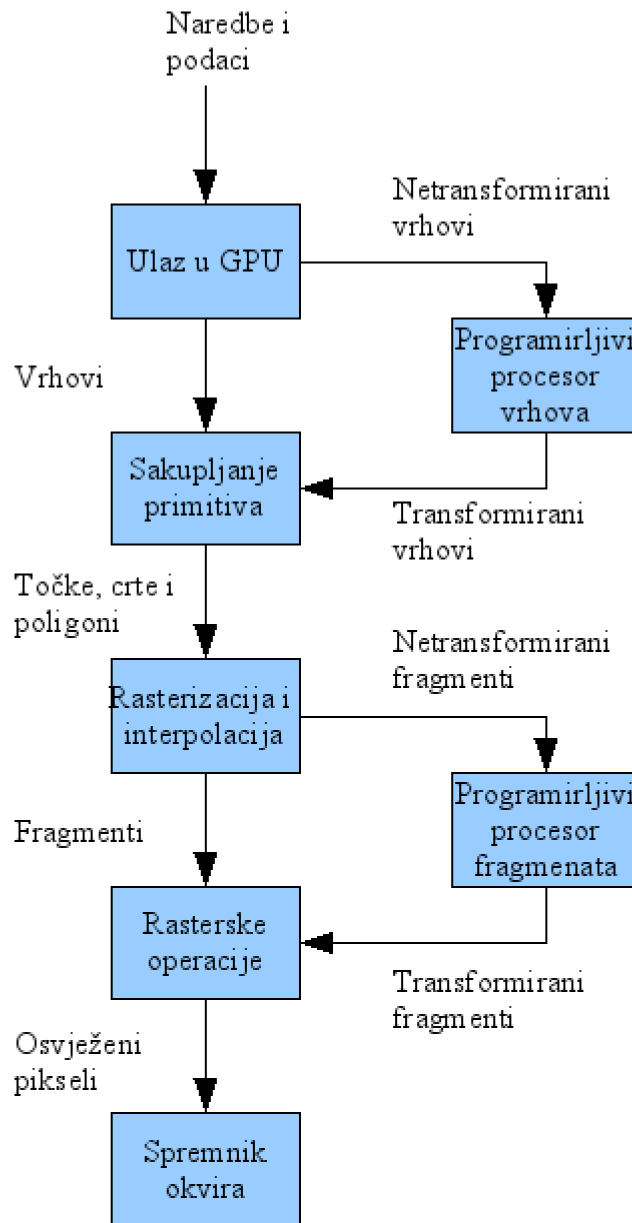
U praksi se broj vrhova u sceni tijekom jedne sekunde mjeri u stotinama milijuna, a broj piksela u milijardama. Pošto je transformacija vrhova odvojena faza od transformacije fragmenata, one se mogu odvijati nezavisno (paralelno).
Zbog toga je uobičajeno da je broj procesora za fragmente veći od broja procesora za vrhove. Tipični odnosi su 16/6 i 24/8.

Lijeva strana slike 4 već je opisana ranije u poglavlju o neprogramirljivom sklopovlju. Procesori na desnoj strani su jedini dodatak.

Princip rada je slijedeći: vrhovi i fragmenti prolaze kroz sklopovski implementirane (fiksne) funkcije ili kroz procesore koji ih transformiraju. Dakle, imamo 4 moguće kombinacije:

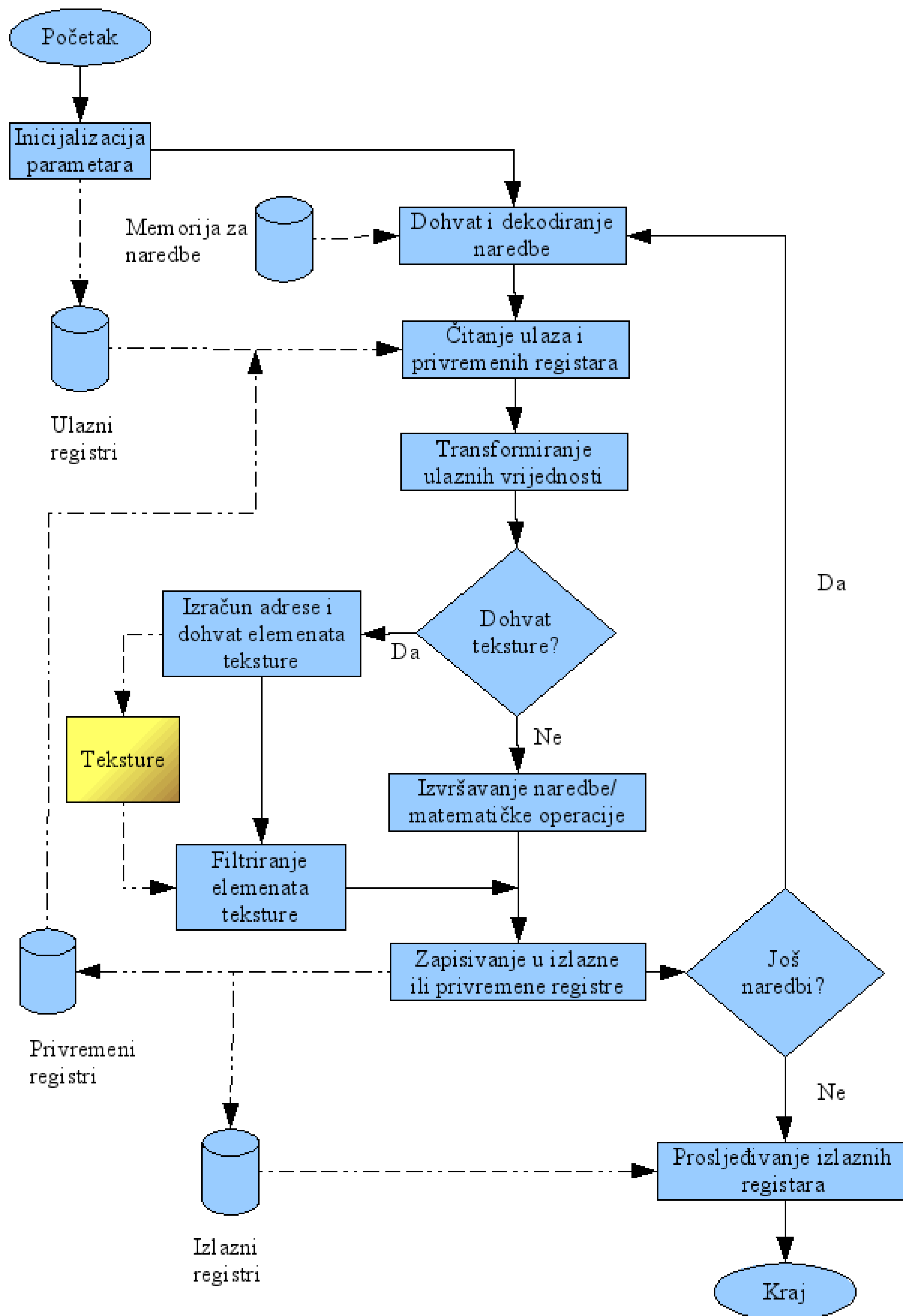
- vrhovi – fiksne funkcije, fragmenti – fiksne funkcije
- vrhovi – procesor, fragmenti – fiksne funkcije
- vrhovi – procesor, fragmenti – procesor
- vrhovi – fiksne funkcije, fragmenti – procesor

Važno je uočiti da se ne mogu istovremeno koristiti fiksne funkcije i procesor. Sve ono što bi se inače obavilo fiksnim funkcijama treba se izvesti na procesoru (ako ga se koristi).



Slika 4: Shema programirljivog sklopovlja.

Slijedeći dio opisuje procesore vrhova i fragmenata. Svi opisi će se odnositi na što novije sklopovlje (ukoliko nije drugačije naznačeno). To je opravdano s obzirom na brzinu razvoja grafičkog sklopovlja.



Slika 5: Shema procesora vrhova i fragmenata.

4.1. Programirljivi grafički procesori

Na početku treba objasniti neke elementarne stvari vezane uz arhitekturu procesora vrhova i fragmenata.

Prema tipu naredbi procesori se dijele na RISC i CISC. Grafički procesori su slični tipu RISC, iako se ne mogu strogo klasificirati ni u jedan tip.

Prema porodici procesori dijele se na: SPARC, x86, PowerPC, Alpha itd. Grafički procesori se ne dijele na porodice. Postoji samo jedna jedina porodica. Oni se dijele prema vrsti transformacija koju obavljaju: procesori za transformaciju vrhova i procesori za transformaciju fragmenata.

No, grafički procesori se dijele na generacije, isto kao i svi ostali procesori. Kako unutar x86 porodice imamo Pentium, Pentium II, Pentium III i Pentium IV, tako unutar grafičkih procesora imamo porodice 1, 2 i 3.

Generacija grafičkih procesora definira se OpenGL ekstenzijama, odnosno revizijom DirectX standarda (8.1, 9.0, 9.0a, 9.0b, 9.0c). Kako je već nekoliko puta spomenuto, opisi će se odnositi na OpenGL.
Svaka generacija definira zahtijeve koje oba grafička procesora moraju ispunjavati.

Podjela na generacije nije stroga, jer ima puno ekstenzija koje utječu na definiciju zatjeva. Najčešće se koristi termin "profil". Svaka generacija ima po nekoliko profila koji imaju manje ili više mogućnosti. U praksi se koristi najslabiji profil koji ispunjava sve zahtjeve koji su nam potrebni. U tom slučaju je veća šansa da će program raditi na slabijem sklopovlju.

Neki od profila koji se trenutno upotrebljavaju:

- DirectX
  - Fragmenti
    - dx8ps
    - ps_1_1
    - ps_1_2
    - ps_1_3
    - ps_2_0
    - ps_2_x
    - ps_3_0
    - dx9ps2
  - Vrhovi
    - dx8vs
    - dxvs2
    - vs_1_1
    - vs_2_0
    - vs_2_x
    - vs_3_0
- OpenGL

- Fragmenti
  - glslf
  - arbfp1
  - fp20
  - fp30
  - fp40
- Vrhovi
  - glslv
  - arbvp1
  - vp20
  - vp30
  - vp40

Napomena: Navedena imena profila odgovaraju konvenciji programskog jezika Cg (DirectX dio se podudara sa jezikom HLSL). Definiranje profila u asemblerskim programima je nešto drugačije.

Dosta profila je funkcionalno ekvivalentno: vp40 i vs_3_0, fp30 i ps_2_x, vp30 i vs_2_x itd.
Numeriranje DirectX profila odgovara generaciji procesora, dok OpenGL profili imaju oznaku uvećanu za jedan.

Procesor vrhova biti će objašnjen na profilu vp40, a procesor fragmenata na profilu fp40.

4.2. Procesor vrhova

Prva službena verzija procesora vrhova definirana je ekstenzijom ARB_vertex_program. Ona odgovara profilu arbvp1. Trenutno aktualna verzija grafičkih procesora (verzija 3) definirana je ekstenzijom NV_vertex_program3 koja je proširenje ekstenzije ARB_vertex_program. Odgovarajući profil je vp40.

Treba napomenuti da svi proizvođači ne podržavaju nužno funkcionalnost NV_vertex_program3 ekstenzije direktno u asemblerskom obliku. Npr. ATI podržava samo ARB_vertex_program u asemblerskom obliku, dok funcionalnost NV_vertex_program3 i sličnih ekstenzija pruža samo kroz jezike više razine (npr. GLSL).

Osnovni tipovi podataka su skalari, vektori i matrice. Podrazumijevani tip podatka je 32 bitni realni broj - **`float`** (isti kao u programskom jeziku C). Postoji još i 16 bitni realni broj – **`half`** i 12 bitni realni broj - **`fixed`**. Postoje još i tipovi podataka koji odgovaraju cjelobrojnim vrijednostima (**`int`**) i logičkim (**`bool`**). Zadnja dva tipa podataka se aproksimiraju realnim brojevima.

Najčešće se koriste 32 bitni realni brojevi, osim kada nije potrebna preciznost. U tom

slučaju se koriste 16 i 12 bitni realni brojevi. Korištenjem realnih brojeva koji su manji od 32 bita dobivamo na brzini (do 2 puta).

Upotreba ne-realnih brojeva nije uobičajena, osim u situacijama gdje se ispituju uvjeti.

Vektori se sastoje od skalara, a matrice od vektora. Tu su moguće različite kombinacije dimenzija, no uobičajene su 4 za vektore i 4x4 za matrice.

Skoro sve naredbe rade sa 4 skalara paralelno, odnosno koriste SIMD princip (engl. **S**ingle **I**nstruction **M**ultiple **D**ata).

Moguće su slijedeće kombinacije izračuna u jednom taktu:

- 4 skalara
- 2 dvokomponentna vektora
- skalar i jedan trokomponentni vektor
- četverokomponentni vektor

U slijedećoj tablici navedena su neka karakteristična svojstva ekstenzije NV_vertex_program3. Vrijednosti su prikupljene na grafičkoj kartici navedenoj u uvodu uz pomoć programa OpenGL Extensions Viewer tvrtke Realtech VR.

Tablica 2: Tipični parametri procesora vrhova treće generacije

Kapacitet memorije za naredbe	512
Ukupno izvršenih naredbi	65536
Privremenih vrijednosti	48
Parametarskih registara	256
Atributa	16
Adresnih registara	2
Broj matrica	8
Dubina stoga	1
Teksturnih jedinica	4
Uvjetnih registara	2

Memorija za naredbe je memorija u kojoj se drži operacijski kod naredbe. Kapacitet te memorije može prihvatiti najviše 512 naredbi. No, to ne ograničava ukupan broj naredbi koje se mogu izvršiti. Korištenjem petlji, skokova itd. može se izvršiti najviše 65536 naredbi za svaki vrh.

Privremene vrijednosti su one koje služe za pohranu međurezultata u složenim operacijama.

Parametarski registri su registri koje program koristi prilikom rada uglavnom za čitanje. Postoje lokalni parametarski registri i parametarski registri okruženja. Parametarski registri okruženja se postavljaju iz grafičke aplikacije, npr. projekcijska matrica, položaj izvora svjetla itd.

Atributi su položaj vrha, boja vrha, koordinate teksture itd.

Adresni registri služe za adresiranje lokalnih registara u dinamičkim uvjetima, odnosno kada se adresira neki registar čiju adresu nije moguće ustanoviti prilikom prevođenja programa.

Stog služi za pohranu adresnih registara. Kao što se vidi, za sada mu je primjena vrlo ograničena.

Teksturne jedinice su dijelovi grafičkog sklopovlja koje na osnovu teksturnih koordinata vraćaju boju koja odgovara zadanoj adresi teksture. Procesor vrhova može koristiti maksimalno 4 teksturne jedinice, što je i više nego dovoljno jer se teksture u ovoj fazi uglavnom koriste za određivanje parametara fluida i sl., a ne za određivanje boje fragmenta.

Uvjetni registri služe za stvaranje maske pomoću 4 vrijednosti: GT (greater than), EQ (equal), LT (less than), ili UN (unordered). Ta se maska onda može koristiti za upis u registar ili za određivanje skoka.

Tablica 3: Atributni registri procesora vrhova

Registar	Alternativni naziv	Upotreba
vertex.position	vertex.attrib[0]	Položaj vrha
vertex.weight	vertex.attrib[1]	Težina vrha – kod dvije modelirajuće matrice
vertex.weight[0]	vertex.attrib[1]	Težina vrha – kod dvije modelirajuće matrice
vertex.normal	vertex.attrib[2]	Normala vrha
vertex.color	vertex.attrib[3]	Boja vrha
vertex.color.primary	vertex.attrib[3]	Boja vrha
vertex.color.secondary	vertex.attrib[4]	Sekundarna boja vrha
vertex.fogcoord	vertex.attrib[5]	Koordinata u polju magle
vertex.texcoord	vertex.attrib[8]	Koordinata teksture
vertex.texcoord[0]	vertex.attrib[8]	Koordinata teksture

vertex.texcoord[1]	vertex.attrib[9]	Koordinata teksture 1
vertex.texcoord[2]	vertex.attrib[10]	Koordinata teksture 2
vertex.texcoord[3]	vertex.attrib[11]	Koordinata teksture 3
vertex.texcoord[4]	vertex.attrib[12]	Koordinata teksture 4
vertex.texcoord[5]	vertex.attrib[13]	Koordinata teksture 5
vertex.texcoord[6]	vertex.attrib[14]	Koordinata teksture 6
vertex.texcoord[7]	vertex.attrib[15]	Koordinata teksture 7
vertex.texcoord[n]	vertex.attrib[8+n]	Koordinata teksture n

Tablica 4: Parametarski registri procesora vrhova

Naziv	Opis
program.env[a]	Parametar okoliša a
program.env[a..b]	Raspon parametara okoliša od a do b
program.local[a]	Lokalni parametar a
program.local[a..b]	Raspon lokalnih parametara od a do b

Tablica 5: Pristup parametrima materijala

Naziv	Opis
state.material.ambient	Ambijentna komponenta boje prednje strane
state.material.diffuse	Difuzna komponenta boje prednje strane
state.material.specular	Zrcalna komponenta boje prednje strane
state.material.emission	Komponenta zračenja boje prednje strane
state.material.shininess	Komponenta svjetline boje prednje strane
state.material.front.ambient	Ambijentna komponenta boje prednje strane
state.material.front.diffuse	Difuzna komponenta boje prednje strane
state.material.front.specular	Zrcalna komponenta boje prednje strane
state.material.front.emission	Komponenta zračenja boje prednje strane
state.material.front.shininess	Komponenta svjetline boje prednje strane

state.material.back.ambient	Ambijentna komponenta boje stražnje strane
state.material.back.diffuse	Difuzna komponenta boje stražnje strane
state.material.back.specular	Zrcalna komponenta boje stražnje strane
state.material.back.emission	Komponenta zračenja boje stražnje strane
state.material.back.shininess	Komponenta svjetline boje stražnje strane

Tablica 6: Pristup parametrima svjetla

Naziv	Opis
state.light[n].ambient	Ambijentna komponenta svjetla n
state.light[n].diffuse	Difuzna komponenta svjetla n
state.light[n].specular	Zrcalna komponenta svjetla n
state.light[n].position	Položaj svjetla n
state.light[n].attenuation	Komponente prigušenja svjetla n
state.light[n].spot.direction	Orjentacija svjetla n

Tablica 7: Pristup matricama

Naziv	Opis
state.matrix.modelview[n]	Matrica model-pogled n
state.matrix.projection	Projekcijska matrica
state.matrix.mvp	Ukupna transformacijska matrica
state.matrix.texture[n]	Matrica za transformaciju tekstura n

Osim dosadašnjih registara i parametara procesora vrhova postoji još i skup izlaznih registara koji nije spomenut u tablici 2. Ti registri služe za izlaz vrhova iz geometrijske faze i ulazak u fazu prikupljanja primitiva i rasterizaciju.

Tablica 8: Izlazni registri procesora vrhova

Naziv	Opis
result.position	Položaj
result.color	Primarna boja prednje strane
result.color.primary	Primarna boja prednje strane
result.color.secondary	Sekundarna boja prednje strane
result.color.front	Primarna boja prednje strane
result.color.front.primary	Primarna boja prednje strane
result.color.front.secondary	Sekundarna boja prednje strane
result.color.back	Primarna boja stražnje strane
result.color.back.primary	Primarna boja stražnje strane
result.color.back.secondary	Sekundarna boja stražnje strane
result.fogcoord	Koordinata u polju magle
result.pointsize	Veličina točke
result.texcoord	Koordinata teksture
result.texcoord[n]	Koordinata teksture n

Napomena: Prethodne tablice ne sadrže sve registre i parametre. Za detalje pogledati specifikacije ARB_vertex_program*X* i NV_vertex_program*X* ekstenzija (svaka od njih je vrlo opsežna – ukupno imaju nekoliko stotina stranica).

Sada kada su poznati najvažniji registri procesora vrhova, bit će naveden skup naredbi. Kako ih ima 48, bit će nabrojane uz kratak opis.

Oznake koje su korištene u tablici:

- 'v' – vektor
- 's' – skalar
- 'a' – adresni registar
- 'c' – uvjetni registar

Tablica 9: Skup naredbi procesora vrhova

Naredba	Ulaz	Izlaz	Opis
ABS	v	v	Apsolutna vrijednost
ADD	v,v	v	Zbroj
ARA	a	a	Zbroj s adresnim registrom
ARL	s	a	Učitaj u adresni registar
ARR	v	a	Učitaj u adresni registar s zaokruživanjem
BRA	c	-	Grananje
CAL	c	-	Poziv procedure
COS	s	ssss	Kosinus
DP3	v,v	ssss	Trokomponentni skalarni umnožak
DP4	v,v	ssss	Četverokomponentni skalarni umnožak
DPH	v,v	ssss	Homogeni skalarni umnožak
DST	v,v	v	Vektor udaljenosti
EX2	s	ssss	Potencija broja 2
EXP	s	v	Potencija broja 2 - aproksimacija
FLR	v	v	Najmanji cijeli broj
FRC	v	v	Ostatak iza decimalne točke
LG2	s	ssss	Logaritam po bazi 2
LIT	v	v	Izračun koeficijenata svjetla
LOG	s	v	Logaritam po bazi 2 - aproksimacija
MAD	v,v,v	v	Umnožak pa zbroj
MAX	v,v	v	Maksimum
MIN	v,v	v	Minimum
MOV	v	v	Premjesti
MUL	v,v	v	Pomnoži
POPA	-	a	Skini sa stoga
POW	s,s	ssss	Potencija
PUSHA	a	-	Stavi na stog
RCC	s	ssss	Recipročna vrijednost s rezanjem
RCP	s	ssss	Recipročna vrijednost
RET	c	-	Povratak iz procedure
RSQ	s	ssss	Recipročni drugi korijen

SEQ	v,v	v	Postavi ako je jednak
SFL	v,v	v	Postavi ako nije istina
SGE	v,v	v	Postavi ako je veći ili jednak
SGT	v,v	v	Postavi ako je veći
SIN	s	ssss	Sinus
SLE	v,v	v	Postavi ako je manji ili jednak
SLT	v,v	v	Postavi ako je manji
SNE	v,v	v	Postavi ako nije jednak
SSG	v	v	Postavi predznak
STR	v,v	v	Postavi ako je istina
SUB	v,v	v	Oduzmi
SWZ	v	v	Promješaj
TEX	v	v	Pristupi teksturi
TXB	v	v	Pristupi teksturi s razinom detalja (LOD)
TXL	v	v	Pristupi teksturi s eksplicitnom razinom detalja (LOD)
TXP	v	v	Pristupi projektivnoj teksturi
XPD	v,v	v	Vektorski umnožak

Kao što se vidi iz tablice, uvijek se radi s registrima koji sadrže 4 komponente: 4 skalara ili četverokomponentni vektor.

Neke naredbe kao npr. ADD rade s vektorima (u ovom slučaju rezultat je zbroj), a neke sa skalarima kao npr. SIN koji izračunaju vrijednost i rezultat zapišu u registar kao 4 ista skalara.

Svaki skalar se sastoji od 4 dijela: xyzw ili rgba – po želji programera.

```
R1 = (1, 2, 3, 4);

R1.x = R1.r = 1
R1.y = R1.g = 2
R1.z = R1.b = 3
R1.w = R1.a = 4
```

Ako je odabrana jedna od vrijednosti iz skupa rgba, onda se ne smiju koristiti vrijednosti iz skupa xyzw u istom izrazu. Vrijedi i obratno.

```
R0 = (1, 2, 3, 4);
R1 = (5, 6, 7, 8);

MOV R0.xyz, R1.yzwx

R0 => (6, 7, 8, 4)
```

Sada kada je poznat način pohrane vrijednosti u registre može se objasniti izgled jednog tipičnog programa.

Na početku ide **`!!ARBvp1.0`** – oznaka koja govori da je riječ o programu namijenjenom za izvršavanju na procesoru vrhova (definiranom ekstenzijom ARB_vertex_program – osnovni profil arbfp1).

Nakon toga ide **`OPTION`** - oznaka da želimo da se uključe i dodaci definirani nekom drugom ekstenzijom.
Npr. *OPTION NV_vertex_program3;* uključuje dodatke definirane ekstenzijom NV_vertex_program3 – profil vp40.

Zatim slijedi deklaracija atributa, parametara, privremenih i izlaznih varijabli.

```
ATTRIB ulazni_polozaj = vertex.position;
...
PARAM ukupna_transformacijska_matrica[4] = { state.matrix.mvp };
...
TEMP privremena;
...
OUTPUT izlazni_polozaj = result.position;
```

Nakon toga idu asemblerske naredbe programa i na samom kraju oznaka **`END`**.

Primjer: Ulazni položaj vrha treba pomnožiti s ukupnom transformacijskom matricom (model-pogled-projekcija) da bi se dobio izlazni položaj vrha.

```
!!ARBvp1.0
OPTION NV_vertex_program3;

ATTRIB uPoz = vertex.attrib[0];
PARAM c[4] = { program.local[0..3] };
OUTPUT iPoz = result.position;

DP4   iPoz.w, uPoz, c[3];
DP4   iPoz.z, uPoz, c[2];
```

```
DP4   iPoz.y, uPoz, c[1];
DP4   iPoz.x, uPoz, c[0];
END
```

U dodatku A biti će opisan postupak učitavanja i prevođenja programa na konkretnom primjeru.

4.3. Procesor fragmenata

Prve verzije procesora fragmenata u stvari nisu uopće bili procesori, nego napredne operacije nad teksturama. Pri tome se često koristio program **nvparse** koji je naredbe jednostavnog skriptnog jezika prevodio u pozive OpenGL funkcija.

Prva službena definicija procesora fragmenata se može naći u ekstenziji ARB_fragment_program – profil arbfp1. Definicija najnovije verzije može se naći u ekstenziji NV_fragment_program2 – profil fp40.

Procesori fragmenata imaju mnogo sličnosti sa procesorima vrhova, pa će samo razlike biti opisane. Te razlike su uglavnom u ulaznim i izlaznim registrima i većem skupu naredbi, dok se u ostalom vrlo malo razlikuju.

Tablica 10: Tipični parametri procesora fragmenata treće generacije

Kapacitet memorije za naredbe	4096
Max. ALU naredbi	4096
Max. naredbi vezanih uz teksture	4096
Parametara okruženja	256
Max. izvršenih naredbi	65536
Dubina stoga za poziv podprograma	4
Ugniježđenih petlji	4
Ugniježđenih if uvjeta	48
Teksturnih jedinica	16
Privremenih varijabli	32
Atributa	16
Lokalnih parametara	512

Tablica 11: Atributni registri procesora fragmenata

Naziv	Opis
fragment.color	Primarna boja
fragment.color.primary	Primarna boja
fragment.color.secondary	Sekundarna boja
fragment.texcoord	Koordinate teksture, teksturna jednica 0
fragment.texcoord[n]	Koordinate teksture, teksturna jednica n
fragment.fogcoord	Koordinata u polju magle
fragment.position	Položaj

Tablica 12: Izlazni registri procesora fragmenata

Naziv	Opis
result.color	boja
result.depth	dubina

Tablica 13: Skup naredbi procesora fragmenata

Naredba	Ulaz	Izlaz	Opis
ABS	v	v	Apsolutna vrijednost
ADD	v,v	v	Zbroj
BRK	c	-	Prekid izvođenja petlje
CAL	c	-	Poziv procedure
CMP	v,v,v	v	Usporedba
COS	s	ssss	Kosinus s redukcijom na [-π, π]
DDX	v	v	Parcijalna derivacija po X
DDY	v	v	Parcijalna derivacija po Y
DIV	v,s	v	Podijeli komponente vektora sa skalarom
DP2	v,v	ssss	Dvokomponentni skalarni umnožak
DP2A	v,v,v	ssss	Dvokomponentni skalarni umnožak sa skalarnim zbrojem
DP3	v,v	ssss	Trokomponentni skalarni umnožak

DP4	v,v	ssss	Četverokomponentni skalarni umnožak
DPH	v,v	ssss	Homogeni skalarni umnožak
DST	v,v	v	Vektor udaljenosti
ELSE	-	-	Započni else blok if uvjeta
ENDIF	-	-	Kraj if uvjeta
ENDLOOP	-	-	Kraj loop petlje
ENDREP	-	-	Kraj repeat petlje
EX2	s	ssss	Eksponent po bazi 2
FLR	v	v	Najmanji cijeli broj
FRC	v	v	Ostatak iza decimalne točke
IF	c	-	Početak if uvjeta
KIL	v ili c	v	Uništi fragment
LG2	s	ssss	Logaritam po bazi 2
LIT	v	v	Izračunaj koeficijente svjetla
LOOP	v	-	Početak loop petlje
LRP	v,v,v	v	Linearna interpolacija
MAD	v,v,v	v	Umnožak pa zbroj
MAX	v,v	v	Maksimum
MIN	v,v	v	Minimum
MOV	v	v	Premjesti
MUL	v,v	v	Pomnoži
NRM	v	v	Normaliziraj trokomponentni vektor
PK2H	v	ssss	Zapakiraj dva 16 bitna realna broja
PK2US	v	ssss	Zapakiraj dva 16 bitna skalara bez predznaka
PK4B	v	ssss	Zapakiraj četiri 8 bitna skalara sa predznakom
PK4UB	v	ssss	Zapakiraj četiri 8 bitna skalara bez predznaka
POW	s,s	ssss	Potenciraj
RCP	s	ssss	Recipročna vrijednost
REP	v	-	Početak repeat petlje
RET	c	-	Povratak iz procedure
RFL	v	v	Reflektirani vektor
RSQ	s	ssss	Recipročni kvadratni korjen
SCS	s	ss--	Sinus/kosinus bez redukcije
SEQ	v,v	v	Postavi ako je jednak

SFL	v,v	v	Postavi ako nije istina
SGE	v,v	v	Postavi ako je veći ili jednak
SGT	v,v	v	Postavi ako je veći
SIN	s	ssss	Sinus sa redukcijom na $[-\pi, \pi]$
SLE	v,v	v	Postavi ako je manji ili jednak
SLT	v,v	v	Postavi ako je manji
SNE	v,v	v	Postavi ako nije jednak
STR	v,v	v	Postavi ako je istina
SUB	v,v	v	Oduzmi
SWZ	v	v	Promješaj
TEX	v	v	Pristupi teksturi
TXB	v	v	Pristupi teksturi s razinom detalja (LOD)
TXD	v,v,v	v	Pristupi teksturi sa maskom
TXL	v	v	Pristupi teksturi s eksplicitnom razinom detalja (LOD)
TXP	v	v	Pristupi projektivnoj teksturi
UP2H	s	v	Otpakiraj dva 16 bitna realna broja
UP2US	s	v	Otpakiraj dva 16 bitna skalara bez predznaka
UP4B	s	v	Otpakiraj četiri 8 bitna skalara sa predznakom
UP4UB	s	v	Otpakiraj četiri 8 bitna skalara bez predznaka
X2D	v,v,v	v	2D transformacija koordinata
XPD	v,v	v	Vektorski umnožak

Postupak programiranja sličan je kao i kod procesora vrhova. Na početku ide **`!!ARBfp1.0`** – oznaka da se radi o programu za transformaciju fragmenata – profil arbfp1. Zatim idu opcije – u ovom slučaju **`OPTION NV_fragment_program2;`** - profil fp40. Nakon toga idu deklaracije, a zatim asemblerske naredbe i **`END`** na kraju.

Primjer: Napisati program koji će u zavisnosti od R komponente ulazne boje (> 0.5) pristupiti prvoj ili drugoj teksturi i nakon toga rezultatu pribrojiti ulaznu boju.

```
!!ARBfp1.0
OPTION NV_fragment_program2;

PARAM c[1] = { { 0.5 } };
TEMP R0;
TEMP HC;
OUTPUT oCol = result.color;
TEX   R0, fragment.texcoord[0], texture[0], 2D;
SGTRC HC.x, fragment.color.primary, c[0];
TEX   R0(EQ.x), fragment.texcoord[1], texture[1], 2D;
ADDR  oCol, R0, fragment.color.primary;
END
```

5. Jezici visoke razine

Razvoj jezika visoke razine počeo je paralelno sa razvojem programirljivog grafičkog sklopovlja. To ima puno prednosti. Glavna prednost je ta da programer uopće ne mora učiti asembler za grafičke procesore kao što je to morao za procesore opće namjene prije nego što su se počeli javljati prvi jezični procesori (kompajleri).

Druga prednost je ta što je prevedeni kod učinkovit kao da je ručno napisan u asembleru, ako ne i bolji. To je obratno nego kod procesora opće namjene.

Treća prednost je korištenje SIMD arhitekture. U jezike visoke razine za grafičke procesore već postoje ugrađeni tipovi podataka (vektori) koji potpuno iskorištavaju SIMD arhitekturu.

Kod programa koji se izvode na procesorima opće namjene rijetko se upotrebljavaju prednosti SIMD arhitekture zbog toga što jezični procesori nemaju dobru podršku za vektore. Podrška za vektore može se naći kroz različite dodatke jezičnim procesorima, ali je obično dosta složena, pa se nikome ne da to učiti – iako bi dobitak na performansama bio do 300%.

Zbog navedenih prednosti, jezici visoke razine već su našli primjenu u konzolama za igre. Npr. Microsoft Xbox koristi kombinaciju DirectX/HLSL, a Sony PlayStation3 će koristiti OpenGL/Cg.

Osim toga, koriste se i za poslove koji nisu vezani uz računalnu grafiku, npr. fizikalne simulacije.

Jezici i programi za sjenčanje nisu nastali kada i programirljivo grafičko sklopovlje za osobna računala, već nekih desetljeće i pol ranije.
Njihov nastanak je vezan uz filmsku produkciju (specijalne učinke i animirane filmove). U filmskoj produkciji je potrebna vrlo visoka kvaliteta, bez obzira na brzinu izračuna. Zbog toga tadašnji programi za sjenčanje (a i neki sadašnji) ne rade izračun u stvarnom vremenu već off-line.

Prve jezike za sjenčanje napravili su Robert L. Cook i Ken Perlin sredinom 80-ih godina prošlog stoljeća.
Perlin je radio proceduralne funcije šuma koje se sada zovu po njemu, a Cook je radio na "stablima za sjenčanje" za tvrtku Lucasfilm.
Njihove ideje dovele su do nastanka jezika (ustvari cijelog sustava) Renderman kojega je napravio Pat Hanrahan za tvrtku Pixar.
Razvijeno je još nekoliko jezika za sjenčanje od kojih su poznatiji ISL tvrtke SGI i RTSL sveučilišta Stanford.

2002. godine javljaju se jezici HLSL (**H**igh **L**evel **S**hading **L**anguage), Cg (**C** for **g**raphics) i GLSL (Open**GL** **S**hading **L**anguage). To su jezici namjenjeni za rad s programirljivim grafičkim sklopovljem.

U zadnje vrijeme pojavili su se jezici Sh i BrookGPU. To su programski jezici opće namjene koji u sebi imaju ugrađen jezik za sjenčanje.

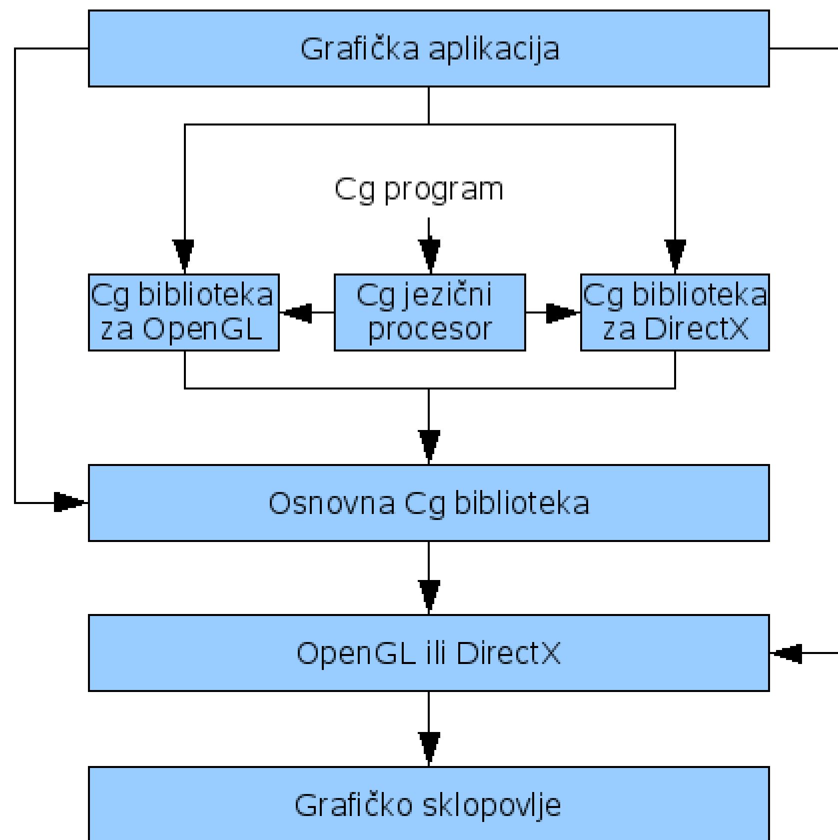
5.1. Programski jezik Cg (i HLSL)

Programski jezik Cg nastao je iz programskog jezika HLSL. HLSL je napravio Microsoft kao dio DirectX paketa. Prednost mu je ta što je vrlo sličan C-u, pa se vrlo lako uči. Nedostaci su mu što radi samo s DirectX-om i to na operacijskom sustavu Windows.
Zbog toga je NVIDIA u suradnji s Microsoft-om napravila programski jezik Cg. Cg je u potpunosti preuzeo sintaksu HLSL-a, ali radi i sa DirectX-om i sa OpenGL-om. To znači da radi i na više operacijskih sustava od HLSL-a (uglavnom Windows i Linux).
Svi daljni opisi se odnose na programski jezik Cg, iako se većina može primjeniti i na HLSL.

Postupak vezan uz pisanje programa u Cg-u i njegovo korištenje je slijedeći:

1. Napisati u datoteke programe za transformaciju vrhova i fragmenata
2. Otvoriti datoteke i učitati ih
3. Prevesti programe u asemblerske naredbe pomoću Cg jezičnog procesora (kompajlera)
4. Asemblerske naredbe predati upravljačkim programima pomoću Cg biblioteke (upravljački programi asemblerske naredbe prevode u strojni kod)
5. Povezati varijable iz programskog jezika Cg s onima iz programskog jezika opće namjene koji se koristi za izradu grafičke aplikacije
6. Uključiti procesore vrhova i fragmenata
7. Slati poligone na iscrtavanje pomoću DirectX-a/OpenGL-a
8. Isključiti procesore vrhova i fragmenata

Navedeni koraci obično se koriste za pojedinu scenu, dok se zadnja tri ponavljaju za svako iscrtavanje te scene.



Slika 6: Korištenje Cg-a iz grafičke aplikacije.

Kod programiranja u Cg-u valja zapamtiti da to nije programski jezik opće namjene, iako ima sve karakteristike jezika opće namjene.
Cg je jezik koji je namjenjen za izradu programa koji će se izvršavati na grafičkom sklopovlju (procesor vrhova i procesor fragmenata).

Njegova ograničenja su slijedeća:

- samo statička alokacija memorije (prilikom deklaracije varijabli), nema funkcije malloc ili operatora new
- ne može raditi s datotekama
- ne postoji koncept višezadaćnosti, višedretvenosti i sl.

Ta ograničenja se čine velikima, ali u biti nisu jer te poslove obavlja grafička aplikacija prije nego se počne s iscrtavanjem scene.

Treba napomenuti da se Cg ne koristi isključivo za grafiku, iako se izvršava na grafičkom sklopovlju. Grafičko sklopovlje je konstruirano tako da su mu osnovni tipovi podataka skalari, vektori i matrice. To omogućava izvršavanje programa koji inače barataju takvim podacima, a nisu nužno grafički. Npr. obrada audio signala, digitalna obrada slike, video

procesiranje, konstrukcija mikroelektroničkih sklopova, simulacija fluida i ostale fizikalne simulacije, simulacija telekomunikacijskih sustava. Teoretski bi se moglo koristiti i za složena dešifriranja zbog velike moći procesiranja, ali za sada se ne upotrebljava u te svrhe.

Na slijedećih nekoliko stranica opisana je sintaksa programskog jezika Cg.

5.1.1. Komentari

Kao i svi programski jezici i Cg ima komentare. Prilikom programiranja u Cg-u komentari su vrlo bitni – više nego u programskim jezicima opće namjene. Razlog tomu je namjena koda. Programi pisani u Cg-u uglavnom se sastoje od matematičkih izraza, pa ih treba dobro komentirati jer se inače vrlo lako može zaboraviti kako program radi.

Komentari koji se mogu koristiti su //, /* i */.

```
float tmp;     //privremena vrijednost

/*varijabla koja čuva vrijednost boje*/
float4 color;
```

5.1.2. Identifikatori

Identifikatori su imena varijabli, struktura, članova struktura, funkcija itd.
Mogu sadržavati slova brojeve i znak _. Na prvom mjestu mora biti slovo ili znak _.

Ispravni identifikatori: _1, dva, osamdeset5
Neispravni identifikatori: 2jedan, 17_

5.1.3. Varijable

Cg za razliku od programskih jezika opće namjene ima relativno malo različitih skalarnih tipova podataka, ali ima jako puno vektorskih i matričnih tipova koje programski jezici opće namjene uglavnom uopće nemaju.

- bool - Booleova vrijednost (istina ili laž)
- int - 32 bitni cijeli broj s predznakom
- unsigned int - 32 bitni cijeli broj bez predznaka
- fixed - 12 bitni realni broj
- half - 16 bitni realni broj
- float - 32 bitni realni broj
- double - za sada 32 bitni realni broj – ovisi o sklopovlju
- float2 - vektor dimenzije 2

- float3 - vektor dimenzije 3
- float4 - vektor dimenzije 4
- float2x2 - matrica dimenzije 2x2
- float2x3 - matrica dimenzije 2x3
- float2x4 - matrica dimenzije 2x4
- float3x2 - matrica dimenzije 3x2
- float3x3 - matrica dimenzije 3x3
- float3x4 - matrica dimenzije 3x4
- float4x2 - matrica dimenzije 4x2
- float4x3 - matrica dimenzije 4x3
- float4x4 - matrica dimenzije 4x4

Iz gornje liste se vidi da su osnovni tipovi podataka bool, int, unsigned int, fixed, half, float i double.
Također se vidi da postoje vektorski i matrični tipovi od kojih su navedeni samo oni tipa float. Vektorski i matrični tipovi postoje za sve osnovne tipove podataka.

Vektorski tipovi zapisani u obliku regularnog izraza:

```
(bool | int | unsigned int | fixed | half | float | double)[234]
```

Matrični tipovi zapisani u obliku regularnog izraza:

```
(bool | int | unsigned int | fixed | half | float | double)[234]x[234]
```

Vrijednosti se varijablama pridružuju kao skalarima i poljima u C-u:

```
float     a = 5.0f;
float3    b = {1.0f, 2.0f, 3.0f};
float2x2  d = {{5.0f, 6.0f},{7.0f, 8.0f}};
```

Napomena: Tipovi podataka ovise o generaciji grafičkih procesora.

U Cg-u varijable mogu biti deklarirane lokalno (unutar funkcija) ili globalno (izvan funkcija). Uobičajena je uporaba lokalnih varijabli.
Pravila za deklaracije varijabli su preuzete iz programskog jezika C++. To znači da varijable mogu biti deklarirane neposredno prije korištenja u tijelu funkcije, a ne nužno na početku kao u programskom jeziku C.

5.1.4. Konstante

Konstanta se definiraju isto kao i varijable s ključnom riječi `const` na početku.

```
const int c = 300000;
const float pi = 3.14f;
```

5.1.5. Strukture

Strukture su skupovi varijabli definiranih elementarnim tipovima podataka. Koriste se za stvaranje složenih tipova podataka, olakšavanje prenošenja parametara u funkcije itd.

```
struct str {
     int a;
     float4 b;
     bool c;
};

str test;
test.a = 5;
```

5.1.6. Polja

Polja su skupovi varijabli istog tipa. Korisne su za držanje podataka grupiranim i prenošenje parametara u funkcije.

```
float a[4] = {1.0f, 2.0f, 3.0f, 4.0f};
a[1] = 5.0f;
float3 b[4];
float3 c;
b[0] = c;
```

Veličina polja je ograničena generacijom procesora koji se koristi.

5.1.7. Pretvorba tipa

Cg koristi funkcijski oblik pretvorbe tipa (C++ stil) iako se može koristiti i C način. Pretvorba tipa može biti implicitna i eksplicitna. Preporuča se koristiti eksplicitnu pretvorbu tipa zbog mnogo vrsta vektorskih i matričnih oblika.

Eksplicitan oblik pretvorbe tipa:

```
tip_podatka(vrijednosti)      //C++ način - preporuča se
(tip_podatka)vrijednost       //C način
```

Primjeri:

```
float a;
int b = 2;

a = float(b);
```

Gornji primjer je možda prejednostavan jer će se pretvorba tipa obaviti implicitno, a da jezični procesor pri tom neće javiti ni upozorenje.

```
float3 a = {1.0f, 2.0f, 3.0f};
float4 b = float4(a, 4.0f);


float3x3 c = {{1.0f, 2.0f, 3.0f},
              {4.0f, 5.0f, 6.0f},
              {7.0f, 8.0f, 9.0f}};
float2x2 d = float2x2(c);
float2x2 e = (float2x2)c;
```

5.1.8. Izdvajanje, zamjena i maskiranje komponenti vektora

Ovi pojmovi se odnose na pristup pojedinim komponentama vektora. Svaki četverodimenzijalni vektor se sastoji od slijedećih komponenata: RGBA, XYZW ili STPQ.
Ako se koriste oznake RGBA, onda se ne smiju istovremeno miješati s oznakama XYZW ili STPQ.
Trokomponentni vektori imaju komponente RGB, XYZ ili STP, a dvokomponentni RG, XY ili ST.

Izdvajanje je metoda koja se koristi kada želimo iz vektora izdvojiti određene komponente:

```
float2 a;
float3 b;
a = b.xz;
```

Zamjena se koristi kada želimo zamijeniti poredak komponenata unutar vektora:

```
float3 a;
float3 b;
a = b.yxz;
```

Maska je postupak kojim se određene komponente vektora štite od promjene:

```
float3 a;
float4 b;
b.xyz = a; //maskirana je w komponenta vektora b
```

Navedena 3 postupka su elementarna. Moguće ih je kombinirati na različite načine.

```
float4 a;
float3 b;
float2 c;
a.xwz = b.xzy + float3(c, 1.0f).yzx;
```

5.1.9. Funkcije

Svrha funkcija ista je kao i u svim programskim jezicima – skup naredbi koje se često ponavljaju izdvojiti u poseban blok.

U programskim jezicima opće namjene postoji specifična funkcija po imenu `main` koja mora biti definirana u svakom programu, a služi za početna podešavanja.
Cg ne zahtijeva funkciju main, ali se onda u grafičkoj aplikaciji mora eksplicitno navesti koja je funkcija glavna.

Ovdje se javlja još jedan tip podatka – `void`. To je prazan tip – ne može sadržavati nikakvu vrijednost. Koristi se kao povratni tip za funkcije koje ne vraćaju rezultat.

```
void f1(float a)
{
	...
}

float4 f2(int b; bool c)
{
	...

	return float4(1, 2, 3, 4);
}
```

Parametri se u funkcije prenose preko vrijednosti, jer u Cg-u ne postoje pokazivači i reference. Tu bi se pojavio problem kod vraćanja više vrijednosti iz funkcije da nije korišten VHDL pristup. VHDL za svaki parametar funkcije zahtijeva "smjer" podatka: `in`, `out` i `inout`.
Cg ne zahtijeva navođenje smjera podataka, ali mu to može poslužiti za vraćanje više

vrijednosti iz funkcije.
Ovaj pristup se može izbjeći tako da funkcija vraća strukturu kao povratnu vrijednost, no to može biti rasipanje vrlo ograničenog memorijskog prostora.

```
struct str {
     int a;
     float b;
};

str funkcija(float g) {
     str struktura;
     struktura.a = 5;
     ...
     return struktura;
}

str funkcija2(in float a, out int b, inout float4 c)
{
     c.y = a * 15;
     b = c.w - 21;
     ...
}
```

Cg podržava preopterećivanje funkcija. To znači da može postojati više funkcija s istim imenom. U tom slučaju se funkcije moraju razlikovati po ulaznim parametrima ili ulaznim parametrima i povratnim tipovima. Ne mogu se razlikovati samo po povratnim tipovima.

```
float len(float2 x)
{
     ...
}

float len(float3 x)
{
     ...
}

float len(float4 x)
{
     ...
}
```

5.1.10. Sučelja

Sučelja su blokovi koji sadrže prototipe funkcija. Kako Cg ne podržava klase, implementacija funkcija sučelja se obavlja u strukturama. Za razliku od programskog jezika JAVA, Cg ne podržava deklaraciju varijabli unutar sučelja.

```
interface Sučelje {
     float zbroj(float x, float y);
}

struct Struktura : Sučelje {
     float zbroj(float x, float y)
     {
          return x + y;
     }
}
```

5.1.11. Vezivanje varijabli uz registre

U prethodnom poglavlju objašnjeno je da procesori vrhova i fragmenata imaju ulazne, lokalne, privremene i izlazne registre.
Da bi se varijabla pridružila registru, iza imena varijable navede se dvotočka i zatim registar. Ne može se eksplicitno specificirati ulazni ili izlazni registar već to ovisi o semantici (in ili out parametar, povratna vrijednost funkcije itd.).
Vezivanje varijabli uz registre može se koristiti u strukturama, parametrima funkcija i uz samo ime funkcije.

```
struct str {
     float4 poz : POSITION;
     float4 boja : COLOR;
}:

float4 main(str struktura) : POSITION
{
     ...
}

void main(     float4 inPos : POSITION,
               out float4 outPos : POSITION,
               float4 boja : COLOR)
{
     ...
}
```

```
str main (float4 polozaj : POSITION)
{
     ...
}

float4 main (  float4 poz : POSITION) : POSITION
{
     ...
}
```

Programi namijenjeni za procesor vrhova moraju imati povezani izlazni registar `POSITION`, a programi namjenjeni za procesor fragmenata izlazni registar `COLOR`.
To se mora ispuniti, jer inače ostatak grafičkog cjevovoda neće moći nastaviti ispravno raditi.

Neki od registara koji se mogu povezati s varijablama:

- POSITION - položaj vrha
- COLOR - boja vrha ili fragmenta
- TEXCOORDn - koordinata teksture
- NORMAL - normala
- PSIZE - veličina točke
- TEXUNITn - teksturna jednica n

Većina registara se veže uz četverokomponentne vektore, osim tekstura koje mogu imati 1, 2, 3 ili 4 dimenzionalne koordinate.

Svi ulazni parametri funkcija koji su vezani uz registre mijenjaju se zavisno o tome koji se vrh ili fragment transformira (položaj, adresa teksture itd.).

5.1.12. Uniformni parametri

Funkcije mogu primati i uniformne parametre (ključna riječ `uniform` prije deklaracije varijable). Uniformni parametri su oni koji su postavljeni iz grafičke aplikacije i koji se ne mijenjaju tijekom izvršavanja programa, bez obzira koji vrh ili fragment se transformira.
Takvi parametri obično se koriste za postavljanje projekcijskih matrica, teksturnih matrica, parametre osvjetljenja, parametre magle itd. Navedeni parametri su isti za cijelu scenu, pa prema tome nema potrebe mijenjati ih za pojedini vrh ili fragment.

```
float4 main(   float4 položaj : POSITION,
               uniform float4x4 projMatrica) : POSITION
{
      ...
}
```

5.1.13. Uzorkovanje tekstura

Jedini parametri koji moraju biti uniformni su objekti za uzorkovanje tekstura.
Objekti za uzorkovanje tekstura se mogu smatrati kao varijable čije su vrijednosti teksture. Zbog toga se objekti za uzorkovanje tekstura vežu uz teksturne jedinice.
Teksturne jedinice su dijelovi grafičkog procesora koji za zadane koordinate tekstura vraćaju boju koja se nalazi na tim koordinatama.
Do boje na zadanim koordinatama teksture se dolazi pomoću funkcije kojoj se zadaje objekt za uzorkovanje tekstura i koordinate. Neke od tih funkcija biti će opisane kasnije.

Oni mogu biti:

- sampler1D - jednodimenzionalna tekstura – potencija broja 2
- sampler2D - dvodimenzionalna tekstura – potencija broja 2
- samplerRECT - dvodimenzionalna tekstura – ne mora biti potencija broja 2
- sampler3D - prostorna tekstura
- samplerCUBE - tekstura koja se sastoji od 6 stranica – kocka

```
float4 main(   float4 položaj : POSITION,
               uniform sampler2D jedinica0 : TEXUNIT0,
               float2 koord : TEXCOORD0) : POSITION
{
      ...
}
```

Treba napomenuti da se kod starijih profila moralo eksplicitno navesti koja se teksturna jedinica koristi pomoću `TEXUNIT`*n*. Kod novijih profila to nije više potrebno, jer se povezivanje vrši na višoj razini – iz grafičke aplikacije.

5.1.14. Operatori

Za skalare su isti kao u programskom jeziku C. Za vektore se primjenjuju na svaku komponentu posebno.

Trenutno podržani operatori: () [] . ! ++ - + (cast) * / < <= > >= == != && || ? : = += -= *= /= ,

Operatori rezervirani za buduću implementaciju: -> ~ sizeof % << >> & ^ | %= &= ^= |= <<= >>=

Primjer:

```
float4 a;
```

```
float4 b;
float4 c;
float4 d;

a.x = b.x * c.x + d.x;
a.y = b.y * c.y + d.y;
a.z = b.z * c.z + d.z;
a.w = b.w * c.w + d.w;
```

Gornji odsječak daje identičan rezultat kao donji, samo što se donji 4 puta brže izračuna.

```
a = b * c + d;
```

5.1.15. Uvjeti

Uvjeti imaju dva oblika: **`?:`** i **`if-else`**. Upotreba je ista kao u programskom jeziku C.

```
rezultat = (uvjet) ? uvjet istinit : uvjet nije istinit;


if (uvjet1) {
     ...
} else if (uvjet2) {
     ...
} else if (uvjetN) {
     ...
} else {
     ...
}
```

Kod prethodnog primjera if uvjeta mogu se izostaviti else i else if blokovi.

5.1.16. Petlje

Postoje 3 vrste petlji:

- for
- while
- do - while

Kod petlji postoji ograničenje upotrebe. Starije generacije grafičkih procesora nisu podržavale dinamičku kontrolu toka. Zbog toga uvjet zaustavljanja petlje mora biti poznat u

vrijeme prevođenja programa. Isto tako su petlje morale biti kratke jer se koristila metoda odmotavanja petlji. Ta metoda radi na slijedeći način:

- Odredimo broj naredbi koje se nalaze u petlji - N
- Odredimo broj iteracija petlje - M
- Blok naredbi koje se nalaze u petlji (N) napišemo M puta – za svaku iteraciju petlje po jednom, uz parametare koji bi se koristili u određenoj iteraciji petlje

Sa ili bez dinamičke kontrole toka će se izvršiti N*M naredbi. S dinamičkom kontrolom toka program će sadržavati N naredbi, a bez dinamičke kontrole toka će sadržavati N*M naredbi.

```
for (inicijalizacija; uvjet_zaustavljanja; uvećanje)
{
     ...
}

while (uvjet) {
     ...
}

do {
     ...
} while (uvjet);
```

Primjeri:

```
for (int i = 0, sum = 0; i < 10; i++) {
     sum += i;
}

while(1) {
     fork(Agent(Smith)); //fork() ne postoji u Cg-u.
}                        //Ovdje je samo radi ilustracije.

do {
     a = b + 10;
} while (b < 200);
```

5.1.17. Ugrađene funkcije

Ugrađenih funkcija ima puno, pa će stoga biti prikazane tablično.

Napomena: Prije korištenja funkcija treba provjeriti koje funkcije podržava koja generacija grafičkih procesora.

Tablica 14: Popis ugrađenih funkcija programskog jezika Cg

Funkcija	Opis
abs(x)	Apsolutna vrijednost od x
acos(x)	Arcus kosinus od x u intervalu [0, π], x je iz intervala [-1, 1]
all(x)	Vraća **true** ako je svaka komponenta od x različita od 0
any(x)	Vraća **true** ako je bar jedna komponenta od x različita od 0
asin(x)	Arcus sinus od x u intervalu [-π/2, π/2], x je iz intervala [-1, 1]
atan(x)	Arcus tangens od x u intervalu [-π/2, π/2]
atan2(y, x)	Arcus tangens od y/x u intervalu [-π, π]
ceil(x)	Najmanji cijeli broj koji nije manji od x
clamp(x, a, b)	Smješta x u interval [a, b] na slijedeći način: • a ako je x < a • b ako je x > b • x inače
cos(x)	Kosinus od x
cosh(x)	Kosinus hiperbolni od x
cross(A, B)	Vektorski umnožak A i B (moraju biti trokomponentni vektori)
degrees(x)	Pretvorba iz radijana u stupnjeve
determinant(M)	Determinanta matrice M
dot(A, B)	Skalarni umnožak vektora A i B
exp(x)	e^x
exp2(x)	2^x
floor(x)	Najveći cijeli broj koji nije veći od x
fmod(x, y)	Ostatak dijeljenja x/y s istim predznakom od x. Ako je y nula, rezultat ovisi o implementaciji
frac(x)	Ostatak iza decimalne točke iz intervala [0, 1)
frexp(x, out exp)	Dijeli x na dva dijela: • normalizirani dio iza decimalne točke iz intervala [½, 1)

	• potenciju broja 2 koja je spremljena u `exp` Ako je x nula, oba dijela su 0
isfinite(x)	Vraća **true** ako je x konačan broj
isinf(x)	Vraća **true** ako x nije konačan broj
isnan(x)	Vraća **true** ako x nije broj
ldexp(x, n)	$x * 2^n$
lerp(a, b, f)	Linearna interpolacija: `(1 - f) * a + b * f`, gdje su a i b skalarni ili vektorski tipovi, a f skalar ili isti vektorski tip kao a i b
lit(NdorL, NdotH, m)	Izračunava ambijentnu, difuznu i zrcalnu komponentu boje
log(x)	ln(x), x mora biti veći od 0
log2(x)	ld(x), x mora biti veći od 0
log10(x)	log(x), x mora biti veći od 0
max(a, b)	Maksimum od a i b
min(a, b)	Minimum od a i b
modf(x, out ip)	Vraća cjelobrojni dio od x u `ip`, a dio iza decimalne točke preko imena funkcije
mul(M, N)	Vraća umnožak matrica M i N
mul(M, v)	Umnožak matrice M i vektora v
mul(v, M)	Umnožak vektora v i matrice M
noise(x)	Stvara vrijednost pomoću funkcije šuma. Uvijek vraća istu vrijednost za isti ulazni parametar.
pow(x, y)	x^y
radians(x)	Pretvorba iz stupnjeva u radijane
round(x)	Najbliži cijeli broj od x
rsqrt(x)	Recipročni kvadratni korijen od x, x mora biti veći od 0
saturate(x)	Smješta x u interval [0, 1]
sign(x)	Vraća predznak od x na slijedeći način: • 1 ako je x > 0 • -1 ako je x < 0 • 0 inače
sin(x)	Sinus od x
sincos(x, out s, out c)	U s se smješta sinus od x, a u c kosinus od x. Ova fukcija je učinkovita kada nam trebaju i sinus i kosinus od x, jer bi ih inače morali računati nezavisno.
sinh(x)	Sinus hiperbolni od x
smoothstep(min, max, x)	x se prvo smješta u interval [min, max], a zatim izračunava

	$-2 * ((x - min) / (max - min))^3 + 3 * ((x - min) / (max - min))^2$
step(a, x)	Funkcija vraća: • 0 ako je x < a • 1 ako je x >= a
sqrt(x)	Kvadratni korijen od x, x mora biti veći od 0
tan(x)	Tangens od x
tanh(x)	Tangens hiperbolni od x
transpose(M)	Transponira matricu M
distance(pt1, pt2)	Euklidska udaljenost između točaka pt1 i pt2
faceforward(N, I, Ng)	Funkcija vraća: • N ako je Ng • I < 0 • -N inače
length(v)	Duljina vektora v
normalize(v)	Normalizira vektor v
reflect(I, N)	Vraća vektor refleksije – mora biti trokomponentni
refract(I, N, eta)	Vraća vektor refrakcije – mora biti trokomponentni
ddx(a)	Aproksimacija parcijalne derivacije od a po x
ddy(a)	Aproksimacija parcijalne derivacije od a po y
debug(x)	Ako je uključena opcija DEBUG prevodioca, kopira vrijednost od x u izlaznu boju i prekida daljnje izvršavanje programa
tex1D(...)	Pristup 1D neprojektivnoj teksturi
tex1Dproj(...)	Pristup 1D projektivnoj teksturi
tex2D(...)	Pristup 2D neprojektivnoj teksturi
tex2Dproj(...)	Pristup 2D projektivnoj teksturi
textRECT(...)	Pristup 2D neprojektivnoj teksturi čije dimenzije ne moraju biti potencije broja 2
texRECTproj(...)	Pristup 2D projektivnoj teksturi čije dimenzije ne moraju biti potencije broja 2
tex3D(...)	Pristup 3D neprojektivnoj teksturi
tex3Dproj(...)	Pristup 3D projektivnoj teksturi
texCUBE(...)	Pristup kubnoj neprojektivnoj teksturi
texCUBEproj(...)	Pristup kubnoj projektivnoj teksturi

Zadnjih 10 funkcija namjenjene su za pristup teksturama. Ukupno postoji 24 funkcije za pristup teksturama, ali se uglavnom koristi 10.

Najčešći oblik je **`texX(samplerY tex, floatZ s)`**, gdje je X jedan od sufiksa iz gornje tablice, Y jedan od sufiksa koji su navedeni u odlomku "Uzorkovanje tekstura", a Z broj koji odgovara dimenzionalnosti vektora koji služi za određivanje adrese teksture.

Postoji još i oblik **`texX(samplerY tex, floatZ s, floatZ dsdx, floatZ dsdy)`**. X, Y i Z su već opisani, a dsdx i dsdy predstavljaju vrijednost derivacije na zadanim koordinatama teksture.

Postoje još i četiri naredbe koje ne pripadaju niti u jednu kategoriju:

- discard – trenutno prekida izvršenje programa i odbacuje fragment
- break – prekida izvršavanje petlje
- continue – preskače ostatak petlje i kreće u novu iteraciju
- return – povratak vrijednosti iz funkcije

Cjelovit programski primjer će biti izložen u dodatku B.

5.2. Programski jezik GLSL

Postoji nekoliko naziva za ovaj jezik: GLSL, OGLSL, OpenGL Shading Language i glslang. Ovaj jezik je isključivo namijenjen za uporabu s OpenGL-om, kao što je HLSL s DirectX-om.

GLSL je temeljen na ANSI C standardu, osim dijelova koji su u konfliktu s jednostavnošću implementacije i onima koji ograničavaju performanse.
Korišteni su i neki elementi programskog jezika C++: preopterećivanje funkcija i deklaracija varijabli neposredno prije korištenja.

Motivacija za nastanak i razvoj GLSL-a je jako brz razvoj grafičkog sklopovlja. Kako se sklopovlje razvija, tako raste i broj OpenGL ekstenzija kako bi podržao nove mogućnosti sklopovlja. Trenutno postoji oko 370 ekstenzija. Postojanje tako velikog broja ekstenzija otežava ili u potpunosti onemogućava prenošenje grafičke aplikacije između grafičkog sklopovlja različitih generacija (čak i ako je razlika samo u jednoj generaciji) ili proizvođača (NVIDIA u prosjeku implementira 30% više ekstenzija nego ATI).

Grafički procesori su nastali da bi se riješili takvi problemi. Programi pisani u asembleru šalju se upravljačkim programima gdje se prevode u binarni oblik ili se mogu prevesti nezavisno i poslati upravljačkim programima u binarnom obliku. Programi pisani u Cg-u se prevode u Cg biblioteci i šalju upravljačkim programima.

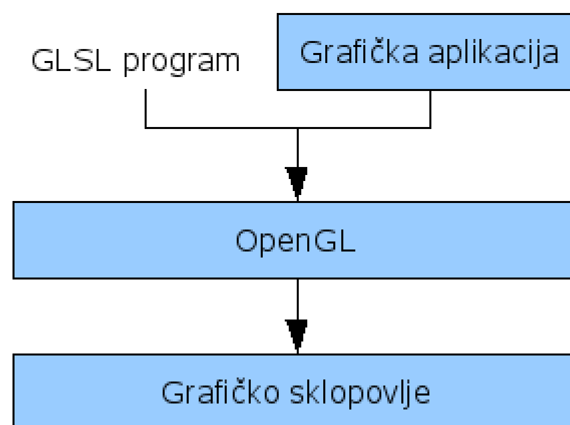
U oba slučaja moramo navesti profil procesora za koji smo namijenili program.
Problem je u tome što imamo istu situaciju kao i s brojnim ekstenzijama: ako program odnesemo na stariji grafički procesor – neće raditi.

Taj problem ne postoji kod GLSL-a. Program se napiše i pošalje upravljačkim programima preko OpenGL-a (i sam OpenGL se nalazi u upravljačkim programima). Ako upravljački programi podržavaju GLSL, program će raditi bez obzira za generaciju procesora.

Korištenje programa napisanih u GLSL-u:

- Kreiramo programski objekt
- Učinimo ga trenutno aktivnim
- Učitamo program iz datoteke
- Prevedemo program
- Koristimo program za što smo naumili
- Izbrišemo programski objekt

Grafička kartica podržava GLSL ako ima ugrađenu ekstenziju `GL_ARB_shading_language_100`. Ta ekstenzija ne radi ništa – služi samo kao indikator postojanja podrške za GLSL.
Osim ove, postoje još 3 ekstenzije koje služe za baratanje programima i programskim objektima. Sve to će biti detaljno opisano u dodatku C.



Slika 7: Korištenje GLSL programa u grafičkoj aplikaciji.

Sintaksa GLSL-a je dosta slična sintaksi Cg-a, pa će biti opisivane samo razlike.

5.2.1. Predprocesor

Predprocesor se izvršava prije prevođenja programa. Svrha mu je prilagoditi postupak prevođenja, pratiti greške, provjeriti verziju prevodioca itd.

Definirane su slijedeće predprocesorske naredbe:

- # - ignorira se
- #define - isto kao u programskom jeziku C++
- #undef - isto kao u programskom jeziku C++
- #if - isto kao u programskom jeziku C++
- #ifdef - isto kao u programskom jeziku C++
- #ifndef - isto kao u programskom jeziku C++
- #else - isto kao u programskom jeziku C++
- #elif - isto kao u programskom jeziku C++
- #endif - isto kao u programskom jeziku C++
- #error - smješta poruku greške u dnevnik (engl. log)
- #pragma - upravlja načinom rada jezičnog procesora
- #extension - određuje način baratanja ekstenzijama
- #version - programi navode verziju jezika u kojoj su napisani
- #line - određuje način brojanja linija

Postoji jedan ugrađeni operator **`defined`** i tri makro-a: **`__LINE__`**, **`__FILE__`** i **`__VERSION__`**.

Operator defined provjerava da li je određeni identifikator definiran:

- defined identifikator
- defined (identifikator)

__LINE__ - vraća n + 1, gdje je n broj znakova za novi red koji mu prethode
__FILE__ - redni broj tekstualne linije koja se procesira
__VERSION__ - verzija GLSL-a

5.2.2. Komentari

Isto kao u Cg-u.

5.2.3. Identifikatori

Isto kao u Cg-u.

5.2.4. Varijable

GLSL ima manje tipova podataka nego Cg, pa će stoga svi biti navedeni:

- void - povratni tip za funkcije koje ne vraćaju vrijednosti
- bool - Booleova vrijednost, **true** ili **false**
- int - 32 bitni cijeli broj s predznakom
- float - 32 bitni realni broj
- vec2 - vektor dimenzije 2 s realnim vrijednostima
- vec3 - vektor dimenzije 3 s realnim vrijednostima
- vec4 - vektor dimenzije 4 s realnim vrijednostima
- bvec2 - vektor dimenzije 2 s Booleovim vrijednostima
- bvec3 - vektor dimenzije 3 s Booleovim vrijednostima
- bvec4 - vektor dimenzije 4 s Booleovim vrijednostima
- ivec2 - vektor dimenzije 2 s cjelobrojnim vrijednostima
- ivec3 - vektor dimenzije 3 s cjelobrojnim vrijednostima
- ivec4 - vektor dimenzije 4 s cjelobrojnim vrijednostima
- mat2 - matrica dimenzija 2x2 s realnim vrijednostima
- mat3 - matrica dimenzija 3x3 s realnim vrijednostima
- mat4 - matrica dimenzija 4x4 s realnim vrijednostima

Kod Cg-a je nepisano pravilo da se varijable uglavnom definiraju kao lokalne. Kod GLSL-a to nije slučaj.

5.2.5. Konstante

Isto kao u Cg-u. Paziti na drugačije tipove podataka.

5.2.6. Strukture

Isto kao u Cg-u. Paziti na drugačije tipove podataka.

5.2.7. Polja

Isto kao u Cg-u. Paziti na drugačije tipove podataka.

5.2.8. Pretvorba tipa

U GLSL-u postoji samo funkcijski oblik pretvorbe tipa:

```
novi_tip(stari_tip)
```

Taj oblik je poznat i kao konstruktorski tip, jer je sličan konstruktorima u objektno

orjentiranim jezicima.

Primjeri:

```
vec3(float)     //svaka komponenta vektora ima istu
                //vrijednost
vec4(ivec4)     //pretvara vektor s cjelobrojnim
                //komponentama u vektor s realnim
mat2(float)     //radi dijagonalnu matricu
mat2(vec2, vec2)    //radi matricu od dva vektora
```

5.2.9. Izdvajanje, zamjena i maskiranje komponenti vektora

Isto kao u Cg-u. Paziti na drugačije tipove podataka.

5.2.10. Funkcije

Isto kao u Cg-u. Paziti na drugačije tipove podataka.

5.2.11. Sučelja

Ne postoje u GLSL-u.

5.2.12. Vezivanje varijabli uz registre

Vezivanje varijabli uz registre ostvaruje se nešto drugačije nego u Cg-u.
Varijable se vezuju uz registre da bi se ostvarilo prenošenje parametara između grafičke aplikacije i procesora vrhova, procesora vrhova i procesora fragmenata, te procesora fragmenata i završne faze iscrtavanja.
Kod Cg-a se to radilo tako da se navela varijabla, stavila dvotočka i zatim navelo ime registra.
Kod GLSL-a se to radi pomoću kvalifikatora. Postoje 3 vrste kvalifikatora (od kojih je uniform poznat iz Cg-a):

- **`attribute`** – prenošenje vrijednosti između grafičke aplikacije (OpenGL-a) i programa za sjenčanje vrhova (položaj, boja, koordinate teksture ...) - vrijednosti se mijenjaju od vrha do vrha
- **`uniform`** – imaju istu vrijednost za sve vrhove/fragmente (projekcijska matrica, teksturna matrica ...)
- **`varying`** – prenošenje vrijednosti između programa za sjenčanje vrhova i programa za sjenčanje fragmenata (novo izračunat položaj, transformirane koordinate tekstura ...)

Svi kvalifikatori se navode prilikom deklaracije varijable, npr.:

```
attribute vec3 abc;
uniform vec2 def;
varying vec4 ghi;
```

Varijable s kvalifikatorima se deklariraju kao globalne. Vrijednosti varijabli koje imaju kvalifikatore **`attribute`** i **`uniform`** postavljaju se iz grafičke aplikacije.
Izlazne vrijednosti iz programa za sjenčanje fragmenata prenose se završnoj fazi iscrtavanja pomoću ugrađenih varijabli koje ne treba eksplicitno definirati – radi toga nema kvalifikatora vezanog za izlaz programa za sjenčanje fragmenata (izlazna boja fragmenta - `gl_FragColor`).

Postoje i u programima za sjenčanje vrhova varijable s implicitnim kvalifikatorima koje ne treba deklarirati (ukupna transformacijska matrica - `gl_ModelViewProjectionMatrix` – uniform, ulazni položaj vrha - `gl_Vertex` – attribute, izlazni položaj vrha – `gl_Position` – varying).
To su samo neke od ugrađenih varijabli. Sve će biti navedene u tablici pred kraj poglavlja.

Primjer:

Program za sjenčanje vrhova:

```
varying vec3 color;

void main()
{
    color = vec3(1.0, 1.0, 1.0);

    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

Program za sjenčanje fragmenata:

```
varying vec3 color;

void main()
{
    gl_FragColor = vec4(color, 1.0);
}
```

5.2.13. Uzorkovanje tekstura

Postoje slijedeći objekti za uzorkovanje tekstura:

- sampler1D - pristup 1D teksturi
- sampler1DShadow - pristup 1D teksturi s usporedbom dubine
- sampler2D - pristup 2D teksturi
- sampler2DShadow - pristup 2D teksturi s usporedbom dubine
- sampler3D - pristup 3D teksturi
- samplerCube - pristup kubnoj teksturi

Objekti za uzorkovanje tekstura moraju biti deklarirani s kvalifikatorom uniform.

Primjer:

```
uniform sampler2D tex;

void main() {
     ...
}
```

5.2.14. Operatori

Trenutno podržani operatori: () [] . ++ -- + - ! * / < > <= >= == != && ^^ || ?: = += -= *= /= ,

Rezervirani za buduću implementaciju: ~ % << >> & ^ | %= <<= >>= &= ^= |=

5.2.15. Uvjeti

Isto kao u Cg-u.

5.2.16. Petlje

Isto kao u Cg-u.

5.2.17. Ugrađene varijable, konstante i strukture

U odlomku 4.2.12. “Vezivanje varijabli uz registre” navedeno je da programi za sjenčanje vrhova i fragmenata imaju unaprijed definirane varijable. Ovdje će sve biti navedene, kao i konstante i strukture.

U OpenGL-u postoji pravilo da sva imena varijabli, konstanti i funkcija počinju s “gl”. To pravilo vrijedi i ovdje.

Osnovne varijable programa za sjenčanje vrhova:

- `vec4 gl_Position` - izlazni položaj vrha
- `float gl_PointSize` - veličina točke u pikselima
- `vec4 gl_ClipVertex` - koordinate u prostoru omeđenom ravninama koje odsijecaju sve što je iza njih, tzv. “omeđeni prostor”

Osnovne varijable programa za sjenčanje fragmenata:

- `vec4 gl_FragCoord` - ulazne koordinate fragmenta
- `bool gl_FrontFacing` - sadrži **true** ako fragment pripada poligonu koji je pozitivno orjentiran
- `vec4 gl_FragColor` - izlazna boja fragmenta
- `vec4 gl_FragData[]` - dodatni podaci o fragmentu
- `float gl_FragDepth` - izlazna dubina fragmenta

Varijable programa za sjenčanje vrhova s kvalifikatorom attribute:

- `attribute vec4 gl_Color;`
- `attribute vec4 gl_SecondaryColor;`
- `attribute vec3 gl_Normal;`
- `attribute vec4 gl_Vertex;`
- `attribute vec4 gl_MultiTexCoord0;`
- `attribute vec4 gl_MultiTexCoord1;`
- `attribute vec4 gl_MultiTexCoord2;`
- `attribute vec4 gl_MultiTexCoord3;`
- `attribute vec4 gl_MultiTexCoord4;`
- `attribute vec4 gl_MultiTexCoord5;`
- `attribute vec4 gl_MultiTexCoord6;`
- `attribute vec4 gl_MultiTexCoord7;`
- `attribute float gl_FogCoord;`

Ugrađene konstante:

- `const int gl_MaxLights = 8;`
- `const int gl_MaxClipPlanes = 6;`
- `const int gl_MaxTextureUnits = 2;`
- `const int gl_MaxTextureCoords = 2;`
- `const int gl_MaxVertexAttribs = 16;`
- `const int gl_MaxVertexUniformComponents = 512;`
- `const int gl_MaxVaryingFloats = 32;`

```
• const int gl_MaxVertexTextureImageUnits = 0;
• const int gl_MaxCombinedTextureImageUnits = 2;
• const int gl_MaxTextureImageUnits = 2;
• const int gl_MaxFragmentUniformComponents = 64;
• const int gl_MaxDrawBuffers = 1;
```

Navedene vrijednosti su minimalne preporučene standardom - ovise o implementaciji.

Ugrađene varijable s kvalifikatorom uniform:

```
• uniform mat4 gl_ModelViewMatrix;
• uniform mat4 gl_ProjectionMatrix;
• uniform mat4 gl_ModelViewProjectionMatrix;
• uniform mat4 gl_TextureMatrix[gl_MaxTextureCoords];
• uniform mat3 gl_NormalMatrix;
• uniform mat4 gl_ModelViewMatrixInverse;
• uniform mat4 gl_ProjectionMatrixInverse;
• uniform mat4 gl_ModelViewProjectionMatrixInverse;
• uniform mat4 gl_TextureMatrixInverse[gl_MaxTextureCoords];
• uniform mat4 gl_ModelViewMatrixTranspose;
• uniform mat4 gl_ProjectionMatrixTranspose;
• uniform mat4 gl_ModelViewProjectionMatrixTranspose;
• uniform mat4 gl_TextureMatrixTranspose[gl_MaxTextureCoords];
• uniform mat4 gl_ModelViewMatrixInverseTranspose;
• uniform mat4 gl_ProjectionMatrixInverseTranspose;
• uniform mat4 gl_ModelViewProjectionMatrixInverseTranspose;
• uniform mat4
  gl_TextureMatrixInverseTranspose[gl_MaxTextureCoords];
• uniform float gl_NormalScale;
• struct gl_DepthRangeParameters {
     float near;
     float far;
     float diff;
  };
  uniform gl_DepthRangeParameters gl_DepthRange;
• uniform vec4 gl_ClipPlane[gl_MaxClipPlanes];
• struct gl_PointParameters {
     float size;
     float sizeMin;
     float sizeMax;
     float fadeTresholdSize;
     float distanceConstantAttenuation;
     float distanceLinearAttenuation;
     distanceQuadraticAttenuation;
  };
  uniform gl_PointParameters gl_Point;
• struct gl_MaterialParameters {
     vec4 emission;
     vec4 ambient;
     vec4 diffuse;
     vec4 specular;
     float shininess;
  };
  uniform gl_MaterialParameters gl_FrontMaterial;
```

```
   uniform gl_MaterialParameters gl_BackMaterial;
•  struct gl_LightSourceParameters {
      vec4 ambient;
      vec4 diffuse;
      vec4 specular;
      vec4 position;
      vec4 halfVector;
      vec3 spotDirection;
      float spotExponent;
      float spotCutoff;
      float spotCosCutoff;
      float constantAttenuation;
      float linearAttenuation;
      float quadraticAttenuation;
   };
   uniform gl_LightSourceParameters gl_LightSource[gl_MaxLights];
•  struct gl_LightModelParameters {
      vec4 ambient;
   };
   uniform gl_LightModelParameters gl_LightModel;
•  struct gl_LightModelProducts {
      vec4 sceneColor;
   };
   uniform gl_LightModelProducts gl_FrontLightModelProduct;
   uniform gl_LightModelProducts gl_BackLightModelProduct;
•  struct gl_LightProducts {
      vec4 ambient;
      vec4 diffuse;
      vec4 specular;
   };
   uniform gl_LightProducts gl_FrontLightProduct[gl_MaxLights];
   uniform gl_LightProducts gl_BackLightProduct[gl_MaxLights];
•  uniform vec4 gl_TextureEnvColor[gl_MaxTextureImageUnits];
•  uniform vec4 gl_EyePlaneS[gl_MaxTextureCoords];
•  uniform vec4 gl_EyePlaneT[gl_MaxTextureCoords];
•  uniform vec4 gl_EyePlaneR[gl_MaxTextureCoords];
•  uniform vec4 gl_EyePlaneQ[gl_MaxTextureCoords];
•  uniform vec4 gl_ObjectPlaneS[gl_MaxTextureCoords];
•  uniform vec4 gl_ObjectPlaneT[gl_MaxTextureCoords];
•  uniform vec4 gl_ObjectPlaneR[gl_MaxTextureCoords];
•  uniform vec4 gl_ObjectPlaneQ[gl_MaxTextureCoords];
•  struct gl_FogParameters {
      vec4 color;
      float density;
      float start;
      float end;
      float scale;
   };
   uniform gl_FogParameters gl_Fog;
```

Varijable (samo za pisanje) programa za sjenčanje vrhova s kvalifikatorom varying:

- `varying vec4 gl_FrontColor;`
- `varying vec4 gl_BackColor;`

- `varying vec4 gl_FrontSecondaryColor;`
- `varying vec4 gl_BackSecondaryColor;`
- `varying vec4 gl_TexCoord[];`
- `varying float gl_FogFragColor;`

Varijable (samo za čitanje) programa za sjenčanje fragmenata s kvalifikatorom varying:

- `varying vec4 gl_Color;`
- `varying vec4 gl_SecondaryColor;`
- `varying vec4 gl_TexCoord[];`
- `varying float gl_FogFragCoord;`

5.2.18. Ugrađene funkcije

Tablica 15: Popis ugrađenih funkcija programskog jezika GLSL

Funkcija	Opis
radians(x)	Pretvorba iz stupnjeva u radijane
degrees(x)	Pretvorba iz radijana u stupnjeve
sin(x)	Sinux od x
cos(x)	Kosinus od x
tan(x)	Tangens od x
asin(x)	Arcus sinus od x u intervalu [-π/2, π/2], x je iz intervala [-1, 1]
acos(x)	Arcus kosinus od x u intervalu [0, π]
atan(y, x)	Arcus tangens od y/x u intervalu [-π, π]
atan(x)	Arcus tangens od x u intervalu [-π/2, π/2]
pow(x, y)	x^y
exp(x)	e^X
log(x)	ln(x), x mora biti veći od 0
exp2(x)	2^X
log2(x)	ld(x), x mora biti veći od 0
sqrt(x)	Kvadratni korijen od x, x mora biti veći od 0
inversesqrt(x)	Recipročni kvadratni korijen od x, x mora biti veći od 0
abs(x)	Apsolutna vrijednost od x
sign(x)	Vraća predznak od x na slijedeći način: • 1 ako je x > 0

	• -1 ako je x < 0 • 0 inače
floor(x)	Najveći cijeli broj koji nije veći od x
ceil(x)	Najmanji cijeli broj koji nije manji od x
fract(x)	Ostatak iza decimalne točke iz intervala [0, 1)
mod(x, y)	Modul, vraća: **`x – y * floor(x / y)`**
min(x, y)	Minimum od x i y
max(x, y)	Minimum od x i y
clamp(x, a, b)	Smješta x u interval [a, b] na slijedeći način: • a ako je x < a • b ako je x > b • x inače
mix(a, b, f)	Linearna interpolacija: **`(1 – f) * a + b * f`**
step(a, x)	Funkcija vraća: • 0 ako je x < a • 1 ako je x >= a
smoothstep(min, max, x)	x se prvo smješta u interval [min, max], a zatim izračunava $-2 * ((x - min) / (max - min))^3 + 3 * ((x - min) / (max - min))^2$
length(v)	Duljina vektora v
distance(x, y)	Euklidska udaljenost između točaka x i y
dot(u, v)	Skalarni umnožak vektora u i v
cross(u, v)	Vektorski umnožak u i v (moraju biti trokomponentni vektori)
normalize(v)	Normalizira vektor v
ftransform()	Funkcija se može koristiti samo kod transformiranja vrhova. Izračunava izlazni položaj vrha na način kako bi se to radilo da se koriste sklopovski implementirane funkcije.
faceforward(N, I, Nref)	Funkcija vraća: • N ako je Nref • I < 0 • -N inače
reflect(I, N)	Vraća vektor refleksije, ulazni vektori moraju biti normalizirani
refract(I, N, eta)	Vraća vektor refrakcije, ulazni vektori moraju biti normalizirani
matrixCompMult(A, B)	Množi matrice A i B komponentu po komponentu. Za standardni umnožak matrica upotrijebiti operator *.
lessThan(x, y)	Vraća vektor čije se komponente sastoje od rezultata usporedbe x < y.
lessThanEqual(x, y)	Vraća vektor čije se komponente sastoje od rezultata usporedbe x <= y.
greaterThan(x, y)	Vraća vektor čije se komponente sastoje od rezultata usporedbe x

	> y.
greaterThanEqual(x, y)	Vraća vektor čije se komponente sastoje od rezultata usporedbe x >= y.
equal(x, y)	Vraća vektor čije se komponente sastoje od rezultata usporedbe x == y.
notEqual(x, y)	Vraća vektor čije se komponente sastoje od rezultata usporedbe x != y.
any(x)	Vraća istinu ako barem jedna komponenta od x ima vrijednost **`true`**
all(x)	Vraća istinu ako sve komponente od x imaju vrijednost **`true`**
not(x)	Vraća vektor čije se komponete sastoje od logičkih negacija komponenata vektora x
texture1D(...)	Pristup 1D teksturi
texture1DProj(...)	Pristup 1D teksturi s projekcijom
texture1DLod(...)	Pristup 1D teksturi sa zadanom razinom detalja
texture1DProjLod(...)	Pristup 1D teksturi s projekcijom i sa zadanom razinom detalja
texture2D(...)	Pristup 2D teksturi
texture2DProj(...)	Pristup 2D teksturi s projekcijom
texture2DLod(...)	Pristup 2D teksturi sa zadanom razinom detalja
texture2DProjLod(...)	Pristup 2D teksturi s projekcijom i sa zadanom razinom detalja
texture3D(...)	Pristup 3D teksturi
texture3DProj(...)	Pristup 3D teksturi s projekcijom
texture3DLod(...)	Pristup 3D teksturi sa zadanom razinom detalja
texture3DProjLod(...)	Pristup 3D teksturi s projekcijom i sa zadanom razinom detalja
textureCube(...)	Pristup kubnoj teksturi
textureCubeLod(...)	Pristup kubnoj teksturi sa zadanom razinom detalja
shadow1D(...)	Pristup 1D teksturi s usporedbom dubine
shadow2D(...)	Pristup 2D teksturi s usporedbom dubine
shadow1DProj(...)	Pristup 1D teksturi s usporedbom dubine i projekcijom
shadow2DProj(...)	Pristup 2D teksturi s usporedbom dubine i projekcijom
shadow1DLod(...)	Pristup 1D teksturi s usporedbom dubine i odabranom razinom detalja
shadow2DLod(...)	Pristup 2D teksturi s usporedbom dubine i odabranom razinom detalja
shadow1DProjLod(...)	Pristup 1D teksturi s usporedbom dubine, projekcijom i odabranom razinom detalja

shadow2DProjLod(...)	Pristup 2D teksturi s usporedbom dubine, projekcijom i odabranom razinom detalja
dFdx(a)	Aproksimacija parcijalne derivacije od a po x
dFdy(a)	Aproksimacija parcijalne derivacije od a po y
fwidth(a)	Vraća: **`abs(dFdx(a)) + abs(dFdy(a))`**
noise1(x)	Vraća 1D vrijednost šuma na osnovu ulazne vrijednosti x
noise2(x)	Vraća 2D vrijednost šuma na osnovu ulazne vrijednosti x
noise3(x)	Vraća 3D vrijednost šuma na osnovu ulazne vrijednosti x
noise4(x)	Vraća 4D vrijednost šuma na osnovu ulazne vrijednosti x

Sve funkcije za dohvat tekstura koje imaju sufiks “Lod” mogu se koristiti samo na procesorima vrhova.
Funkcije za dohvat teksture koje imaju sufiks “Proj” obavljaju dijeljenje svih komponenata sa zadnjom, odnosno obavljaju projekciju (sjene ili teksture).

Postoje još i 4 naredbe koje ne pripadaju niti jednoj gore navedenoj kategoriji: discard, break, continue i return. Njihovo značenje isto je kao i u Cg-u

6. Programski alati

U ovom poglavlju biti će opisano nekoliko programskih alata koji služe za testiranje performansi grafičkih aplikacija, izradu i testiranje programa za sjenčanje, provjeru sintakse programa za sjenčanje itd.
Programskih alata ima puno, pa će na početku biti dan kratak pregled više njih, a kasnije će nekoliko programskih alata biti detaljno objašnjeno.
Svi programski alati su besplatni (a neki i otvorenog koda), tako da ih svatko slobodno može skinuti s Interneta i koristiti bez ikakvih ograničenja.

Većina programskih alata radi pod operacijskim sustavom Microsoft Windows XP. Svega nekoliko njih je namijenjeno za operacijski sustav Linux (UNIX), ali se neki od onih namijenjenih za Windows XP može koristiti pod Linux-om uz pomoć emulatora Wine (i sličnih).

Programski alati za izradu i testiranje programa za sjenčanje uglavnom koriste platformu .NET. Kako postoji implementacija otvorenog koda .NET platforme po imenu Mono koju sponzorira tvrtka Novell (proizvodi NetWare, SUSE Linux), može se očekivati da će ti programski alati uskoro raditi i na Linux-u.

Uz svaki od programskih alata biti će naznačeno za koji je operacijski sustav namijenjen, te koje grafičko sučelje podržava (ovdje se misli na 3D grafičko sučelje, ukoliko nije drugačije naznačeno).

6.1. Kratki opisi programskih alata

6.1.1. NVPerfKit

Operacijski sustav: Windows
Grafičko sučelje: DirectX, OpenGL
Namjena: Grafičkim aplikacijama pruža pristup registrima za mjerenje performansi na grafičkom sklopovlju.

6.1.2. NVPerfHUD

Operacijski sustav: Windows
Grafičko sučelje: DirectX
Namjena: Detaljno ispitivanje grafičkog cjevovoda. Može se podešavati gotovo svaki dio cjevovoda i na osnovu toga vidjeti razlike u performansama grafičke aplikacije. Rezultati koji se mogu vidjeti su:

- broj okvira u sekundi (engl. **F**rames **P**er **S**econd – FPS)

- broj trokuta po okviru
- zauzeće video memorije
- zauzeće AGP memorije
- opterećenje CPU-a
- opterećenje GPU-a
- broj pristupa teksturama
- vrijeme koje koriste pogonski programi
- vrijeme koje je potrebno da bi se izračunao jedan okvir

Na osnovu svega toga moguće je vrlo precizno ustanoviti koji dio aplikacije uzrokuje pad performansi, odnosno na koji način se to može ispraviti.

6.1.3. NVShaderPerf

Operacijski sustav: Windows
Grafičko sučelje: nijedno - konzolni program
Namjena: Testiranje performansi programa za sjenčanje. Podržava programe za sjenčanje vrhova i fragmenata, te programske jezike HLSL, Cg, GLSL i asembler. Rezultat je izvještaj koji sadrži broj korištenih registara, broj taktova koliko je trajalo izvršavanje programa i broj transformiranih piksela.

6.1.4. DXT utilities

Operacijski sustav: Windows
Grafičko sučelje: nijedno – konzolni program
Namjena: Skup alata za baratanje teksturama: stvaranje i manipulacija MIP i komprimiranim teksturama, baratanje vrlo velikim teksturama, spajanje puno malih tekstura u jednu veliku radi poboljšanja performansi itd.

6.1.5. BuGLe

Operacijski sustav: Linux
Grafičko sučelje: OpenGL
Namjena: Praćenje performansi i otkrivanje grešaka, iscrtavanje scene kao žičani okvir, snima animaciju u video datoteku ili kao statičnu sliku, te omogućava postavljanje prekidnih točaka na OpenGL funkcije (engl. debugger).

6.1.6. GLSL Validate

Operacijski sustav: Windows, Linux s emulatorom Wine
Grafičko sučelje: interno koristi OpenGL biblioteku
Namjena: Ispitivanje ispravnosti sintakse programa za sjenčanje napisanih u jeziku GLSL.

6.1.7. GIMP normalmap plugin

Operacijski sustav: Linux, Windows
Grafičko sučelje: interno koristi OpenGL
Namjena: Dodatak za program GIMP (**G**NU **I**mage **M**anipulation **T**ool) – program za digitalnu obradu slike. Sličan dodatak postoji i za Photoshop, samo što je ovaj složeniji.
Mogućnosti koje pruža:

- filteri, opći 3x3, 5x5, 7x7, 9x9, te 3x3 i 5x5 Sobel i Prewitt za detekciju ruba
- skaliranje normale poslije filtriranja
- omatanje tekstura – kada se ista tekstura nastavlja na samu sebe i tako se stvara ponavljajući uzorak, npr. cigla, kamen, drvo itd.
- teksture s normalama – vrijednost boje ili alfa komponente teksture se koristi za pohranu normala
- zapis visine u alfa komponetu teksture i još nekoliko funkcija za rad s alfa komponentom boje – korisno kad se visina terena čita iz alfa komponente, a boja iz RGB komponente teksture (engl. height mapping)
- 2D i 3D pregled rezultata

6.1.8. FX Composer

Operacijski sustav: Windows
Grafičko sučelje: DirectX
Namjena: Izrada, otklanjanje grešaka i optimizacija programa za sjenčanje napisanih u jeziku HLSL/Cg. Koristi se FX datoteka u kojoj su zapisani programi za sjenčanje i dodatni atributi. Može se koristiti i više od jednog prolaza za izračun scene. U sebi ima integriran NVShaderPerf (opisan ranije u ovom poglavlju).

6.1.9. RenderMonkey

Operacijski sustav: Windows
Grafičko sučelje: DirectX, OpenGL
Namjena: Isto što i FX Composer, samo što ima podršku i za GLSL. RenderMonkey je nešto složeniji alat. U njemu se može bolje organizirati scena i njeni atributi. Osim za programiranje, dobar je i za poslove koje obavlja umjetnik za 3D grafiku.

6.2. Detaljni opisi nekoliko programskih alata

Ovdje će biti opisani programski alati NVShaderPerf, FX Composer i RenderMonkey.
Kako je prvi konzolna aplikacija, bit će prikazan njegov ispis. Ostala dva biti će prikazana uglavnom preko slika s kratkim komentarima.

Iako je NVShaderPerf integriran u FX Composer, bit će opisan zasebno da bi se vidjele sve njegove mogućnosti.

6.2.1. NVShaderPerf

Iz imena programa može se zaključiti da je proizvođač tvrtka NVIDIA. Zbog toga se mogu izračunavati performanse programa samo za grafičke kartice te tvrtke.

Uporaba NVShaderPerf-a je vrlo jednostavna: `nvshaderperf ime_programa`

Ako je ulazni program za sjenčanje vrhova, kao izlaz se dobije asemblerski oblik i broj naredbi. Ako je ulazni program za sjenčanje fragmenata, kao rezultat se dobije ispis programa prevedenog u asemblerski oblik i ispis prilagođen za određenu grafičku karticu. Za oba ispisa dobije se broj naredbi, broj korištenih registara, trajanje izvršavanja programa (u ciklusima) i propusnost pristupa teksturama u milijunima piksela po sekundi.

Da bi se precizno odredila grafička kartica potrebno je navesti parametar `-a` i kraticu grafičke kartice.

```
nvshaderperf -a graficka_kartica ime_programa
```

Kratice su slijedeće:

- G70 - GeForce 7800 GTX
- NV44 - GeForce 6200
- NV43-GT - GeForce 6600 GT
- NV40-12 - GeForce 6800
- NV40-GT - GeForce 6800 GT
- NV40 - GeForce 6800 Ultra
- NV38 - GeForceFX 5950 Ultra
- NV36 - GeForceFX 5700 Ultra
- NV35 - GeForceFX 5900 Ultra
- NV34 - GeForceFX 5200 Ultra
- NV31 - GeForceFX 5600 Ultra
- NV30 - GeForceFX 5800 Ultra

Kao što se vidi, uz NV stoje dva broja: prvi označava oznaku arhitektura, a drugi samu grafičku karticu. Oznaka arhitekture 3 znači da ta arhitektura podržava grafičke procesore druge generacije. Analogno tome, oznaka arhitekture 4 znači da ta arhitektura podržava grafičke procesore treće generacije. Oznaka G70 isto označava arhitekturu 4.

Oznaka grafičke kartice određuje specifični model grafičke kartice određene arhitekture. Grafičke kartice iste arhitekture razlikuju se po količini memorije i frekvencijama na kojima rade grafički procesor i memorija, te broju cjevovoda.

Ako NVShaderPerf ne uspije sam prepoznati tip programa, to mu se može eksplicitno navesti preko `-type` parametra. Moguće vrijednosti su: glsl, glsl_vs, glsl_ps, hlsl, hlsl_vs, hlsl_ps, cg, cg_vs, cg_ps, bin, ps, vs, sm.

Bit će prikazan primjer programa za sjenčanje fragmenata za grafičke kartice GeForceFX 5200 Ultra i GeForce 6800 Ultra.

Testni program za sjenčanje fragmenata:

```
half diffuse(half4 l) { return l.y; }
half specular(half4 l) { return l.z; }

// Main shader.

half4 main(float3 Peye        : TEXCOORD0,
           half3 Neye         : TEXCOORD1,
           half2 uv           : TEXCOORD2,
           half3 Kd           : COLOR0,
           half3 Ks           : COLOR1,
           uniform sampler2D diffuseMap,
           uniform float3 Plight,
           uniform half3 lightColor,
           uniform half3 shininess) : COLOR
{
    // Normalize surface normal, vector to light source, and vector
    // to the viewer
    half3 N = normalize(Neye);
    half3 L = normalize(Plight - Peye);
    half3 V = normalize(-Peye);

    // Compute half-angle vector for specular lighting
    half3 H = normalize(L + V);

    // Compute lighting values.  lit() returns the diffuse coefficient
    // in y (or zero, if NdotL < 0), and the specular coefficient in z
    // (or zero, also if NdotL < 0).
    half NdotL = dot(N, L), NdotH = dot(N, H);
    half4 lighting = lit(NdotL, NdotH, shininess);

    // Compute overall color for the fragment.  Scale sum of diffuse
    // and specular contributions together and by the light color.
    half3 C = lightColor *
       (diffuse(lighting) * Kd * (half3)tex2D(diffuseMap, uv).xyz +
        specular(lighting) * Ks);

    // Always set the alpha value to 1.
    return half4(C, 1);
}
```

Rezultati za GeForceFX 5200 Ultra:

```
----------------------------------------------------------------------------
Running performance on file demo_frag.cg
```

```
Compiled using internal compiler.
FP Assembly Code:
!!FP1.0
# cgc version 1.3.0001, build date Nov 17 2004 14:57:08
# command line args: -quiet -profile fp30
# source file: demo_frag.cg
#vendor NVIDIA Corporation
#version 1.0.02
#profile fp30
#program main
#semantic main.diffuseMap
#semantic main.Plight
#semantic main.lightColor
#semantic main.shininess
#var float3 Peye : $vin.TEXCOORD0 : TEX0 : 0 : 1
#var float3 Neye : $vin.TEXCOORD1 : TEX1 : 1 : 1
#var float2 uv : $vin.TEXCOORD2 : TEX2 : 2 : 1
#var float3 Kd : $vin.COLOR0 : COL0 : 3 : 1
#var float3 Ks : $vin.COLOR1 : COL1 : 4 : 1
#var sampler2D diffuseMap :  : texunit 0 : 5 : 1
#var float3 Plight :  :  : 6 : 1
#var half3 lightColor :  :  : 7 : 1
#var half3 shininess :  :  : 8 : 1
#var float4 main : $vout.COLOR : COL : -1 : 1
DECLARE Plight;
DECLARE shininess;
DECLARE lightColor;
ADDR  R0.xyz, -f[TEX0], Plight;
DP3R  R0.w, R0, R0;
RSQR  R0.w, R0.w;
MULR  H1.xyz, R0.w, R0;
DP3R  R0.x, -f[TEX0], -f[TEX0];
RSQR  R0.x, R0.x;
MULR  H0.xyz, R0.x, -f[TEX0];
ADDH  H0.xyz, H1, H0;
DP3H  H0.w, H0, H0;
RSQH  H0.w, H0.w;
MULH  H2.xyz, H0.w, H0;
DP3H  H0.x, f[TEX1], f[TEX1];
RSQH  H0.x, H0.x;
MULH  H0.xyz, H0.x, f[TEX1];
DP3H  H1.x, H0, H1;
DP3H  H1.y, H0, H2;
MOVH  H1.z, shininess.x;
LITH  H1.yz, H1.xyzz;
MULH  H0.xyz, H1.y, f[COL0];
MULH  H1.xyz, H1.z, f[COL1];
TEX   R0, f[TEX2], TEX0, 2D;
MADH  H0.xyz, H0, R0, H1;
MULH  o[COLR].xyz, H0, lightColor;
MOVH  o[COLR].w, {1}.x;
END
# 24 instructions, 1 R-regs, 3 H-regs
-------------------- NV34 --------------------
Target: GeForceFX 5200 Ultra (NV34) :: Unified Compiler: v77.72
Cycles: 31 :: # R Registers: 2
Pixel throughput (assuming 1 cycle texture lookup) 25.81 MP/s
Compiled using cg.dll.
FP Assembly Code:
!!ARBfp1.0
# cgc version 1.4.0000, build date Jun  9 2005 12:09:02
# command line args: -q -profile arbfp1 -entry main -quiet -profile fp30
```

```
# source file: demo_frag.cg
#vendor NVIDIA Corporation
#version 1.0.02
#profile arbfp1
#program main
#semantic main.diffuseMap
#semantic main.Plight
#semantic main.lightColor
#semantic main.shininess
#var float3 Peye : $vin.TEXCOORD0 : TEX0 : 0 : 1
#var float3 Neye : $vin.TEXCOORD1 : TEX1 : 1 : 1
#var float2 uv : $vin.TEXCOORD2 : TEX2 : 2 : 1
#var float3 Kd : $vin.COLOR0 : COL0 : 3 : 1
#var float3 Ks : $vin.COLOR1 : COL1 : 4 : 1
#var sampler2D diffuseMap :  : texunit 0 : 5 : 1
#var float3 Plight :  : c[0] : 6 : 1
#var half3 lightColor :  : c[2] : 7 : 1
#var half3 shininess :  : c[1] : 8 : 1
#var float4 main : $vout.COLOR : COL : -1 : 1
#const c[3] = 1
PARAM c[4] = { program.local[0..2],
             { 1 } };
TEMP R0;
TEMP R1;
TEMP R2;
ADD R1.xyz, -fragment.texcoord[0], c[0];
DP3 R0.y, R1, R1;
RSQ R0.w, R0.y;
MUL R1.xyz, R0.w, R1;
DP3 R0.x, -fragment.texcoord[0], -fragment.texcoord[0];
RSQ R0.x, R0.x;
MUL R0.xyz, R0.x, -fragment.texcoord[0];
ADD R0.xyz, R1, R0;
DP3 R1.w, R0, R0;
DP3 R0.w, fragment.texcoord[1], fragment.texcoord[1];
RSQ R1.w, R1.w;
MUL R2.xyz, R1.w, R0;
RSQ R0.w, R0.w;
MUL R0.xyz, R0.w, fragment.texcoord[1];
DP3 R2.y, R0, R2;
DP3 R2.x, R0, R1;
MOV R2.z, c[1].x;
LIT R0.yz, R2.xyzz;
MUL R1.xyz, R0.y, fragment.color.primary;
MUL R2.xyz, R0.z, fragment.color.secondary;
TEX R0.xyz, fragment.texcoord[2], texture[0], 2D;
MAD R0.xyz, R1, R0, R2;
MUL result.color.xyz, R0, c[2];
MOV result.color.w, c[3].x;
END
# 24 instructions, 3 R-regs
-------------------- NV34 --------------------
Target: GeForceFX 5200 Ultra (NV34) :: Unified Compiler: v77.72
Cycles: 32 :: # R Registers: 4
Pixel throughput (assuming 1 cycle texture lookup) 25.00 MP/s
```

<u>Rezultati za GeForce 6800 Ultra:</u>

```
-----------------------------------------------------------------------------
Running performance on file demo_frag.cg
Compiled using internal compiler.
FP Assembly Code:
!!FP1.0
# cgc version 1.3.0001, build date Nov 17 2004 14:57:08
# command line args: -quiet -profile fp30
# source file: demo_frag.cg
#vendor NVIDIA Corporation
#version 1.0.02
#profile fp30
#program main
#semantic main.diffuseMap
#semantic main.Plight
#semantic main.lightColor
#semantic main.shininess
#var float3 Peye : $vin.TEXCOORD0 : TEX0 : 0 : 1
#var float3 Neye : $vin.TEXCOORD1 : TEX1 : 1 : 1
#var float2 uv : $vin.TEXCOORD2 : TEX2 : 2 : 1
#var float3 Kd : $vin.COLOR0 : COL0 : 3 : 1
#var float3 Ks : $vin.COLOR1 : COL1 : 4 : 1
#var sampler2D diffuseMap :  : texunit 0 : 5 : 1
#var float3 Plight :  :  : 6 : 1
#var half3 lightColor :  :  : 7 : 1
#var half3 shininess :  :  : 8 : 1
#var float4 main : $vout.COLOR : COL : -1 : 1
DECLARE Plight;
DECLARE shininess;
DECLARE lightColor;
ADDR  R0.xyz, -f[TEX0], Plight;
DP3R  R0.w, R0, R0;
RSQR  R0.w, R0.w;
MULR  H1.xyz, R0.w, R0;
DP3R  R0.x, -f[TEX0], -f[TEX0];
RSQR  R0.x, R0.x;
MULR  H0.xyz, R0.x, -f[TEX0];
ADDH  H0.xyz, H1, H0;
DP3H  H0.w, H0, H0;
RSQH  H0.w, H0.w;
MULH  H2.xyz, H0.w, H0;
DP3H  H0.x, f[TEX1], f[TEX1];
RSQH  H0.x, H0.x;
MULH  H0.xyz, H0.x, f[TEX1];
DP3H  H1.x, H0, H1;
DP3H  H1.y, H0, H2;
MOVH  H1.z, shininess.x;
LITH  H1.yz, H1.xyzz;
MULH  H0.xyz, H1.y, f[COL0];
MULH  H1.xyz, H1.z, f[COL1];
TEX   R0, f[TEX2], TEX0, 2D;
MADH  H0.xyz, H0, R0, H1;
MULH  o[COLR].xyz, H0, lightColor;
MOVH  o[COLR].w, {1}.x;
END
# 24 instructions, 1 R-regs, 3 H-regs
-------------------- NV40 --------------------
Target: GeForce 6800 Ultra (NV40) :: Unified Compiler: v77.72
Cycles: 14.00 :: R Regs Used: 2 :: R Regs Max Index (0 based): 1
Pixel throughput (assuming 1 cycle texture lookup) 457.14 MP/s
Compiled using cg.dll.
```

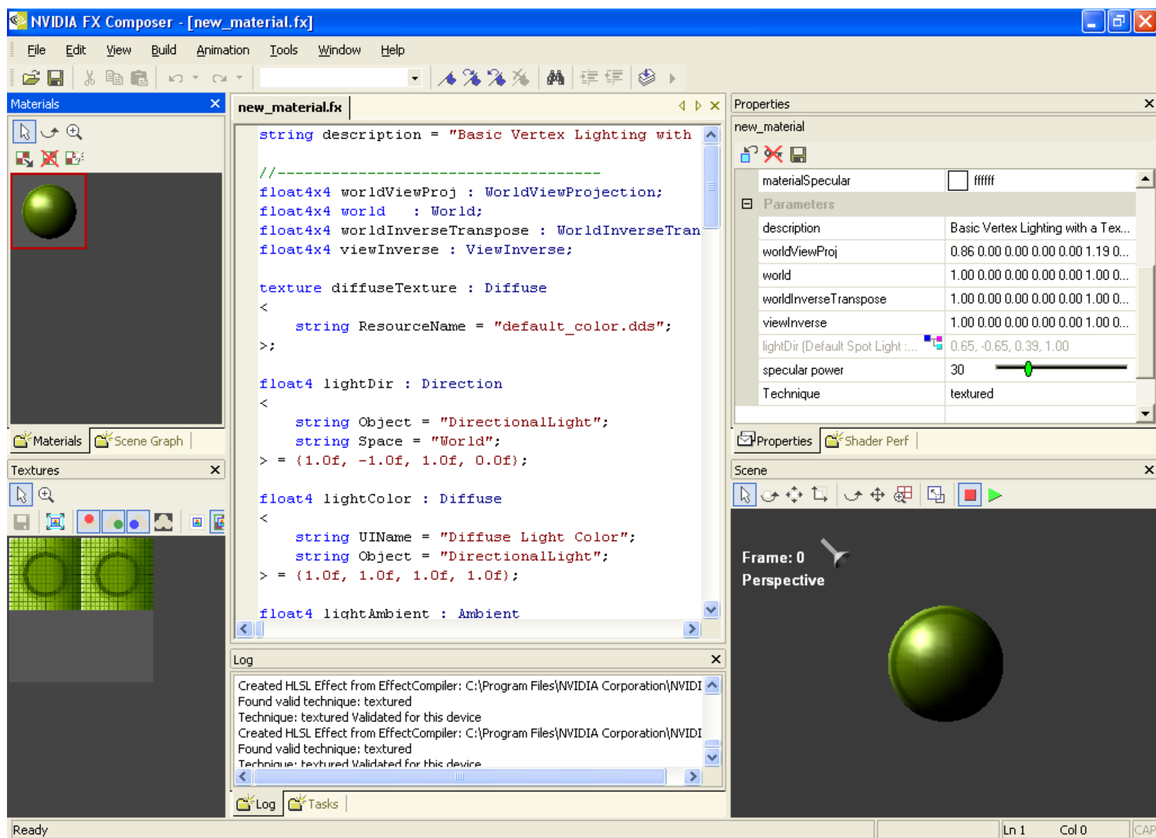
```
FP Assembly Code:
!!ARBfp1.0
# cgc version 1.4.0000, build date Jun  9 2005 12:09:02
# command line args: -q -profile arbfp1 -entry main -quiet -profile fp30
# source file: demo_frag.cg
#vendor NVIDIA Corporation
#version 1.0.02
#profile arbfp1
#program main
#semantic main.diffuseMap
#semantic main.Plight
#semantic main.lightColor
#semantic main.shininess
#var float3 Peye : $vin.TEXCOORD0 : TEX0 : 0 : 1
#var float3 Neye : $vin.TEXCOORD1 : TEX1 : 1 : 1
#var float2 uv : $vin.TEXCOORD2 : TEX2 : 2 : 1
#var float3 Kd : $vin.COLOR0 : COL0 : 3 : 1
#var float3 Ks : $vin.COLOR1 : COL1 : 4 : 1
#var sampler2D diffuseMap :  : texunit 0 : 5 : 1
#var float3 Plight :  : c[0] : 6 : 1
#var half3 lightColor :  : c[2] : 7 : 1
#var half3 shininess :  : c[1] : 8 : 1
#var float4 main : $vout.COLOR : COL : -1 : 1
#const c[3] = 1
PARAM c[4] = { program.local[0..2],
             { 1 } };
TEMP R0;
TEMP R1;
TEMP R2;
ADD R1.xyz, -fragment.texcoord[0], c[0];
DP3 R0.y, R1, R1;
RSQ R0.w, R0.y;
MUL R1.xyz, R0.w, R1;
DP3 R0.x, -fragment.texcoord[0], -fragment.texcoord[0];
RSQ R0.x, R0.x;
MUL R0.xyz, R0.x, -fragment.texcoord[0];
ADD R0.xyz, R1, R0;
DP3 R1.w, R0, R0;
DP3 R0.w, fragment.texcoord[1], fragment.texcoord[1];
RSQ R1.w, R1.w;
MUL R2.xyz, R1.w, R0;
RSQ R0.w, R0.w;
MUL R0.xyz, R0.w, fragment.texcoord[1];
DP3 R2.y, R0, R2;
DP3 R2.x, R0, R1;
MOV R2.z, c[1].x;
LIT R0.yz, R2.xyzz;
MUL R1.xyz, R0.y, fragment.color.primary;
MUL R2.xyz, R0.z, fragment.color.secondary;
TEX R0.xyz, fragment.texcoord[2], texture[0], 2D;
MAD R0.xyz, R1, R0, R2;
MUL result.color.xyz, R0, c[2];
MOV result.color.w, c[3].x;
END
# 24 instructions, 3 R-regs
-------------------- NV40 --------------------
Target: GeForce 6800 Ultra (NV40) :: Unified Compiler: v77.72
Cycles: 15.00 :: R Regs Used: 3 :: R Regs Max Index (0 based): 2
Pixel throughput (assuming 1 cycle texture lookup) 426.67 MP/s
```

6.2.2. FX Composer

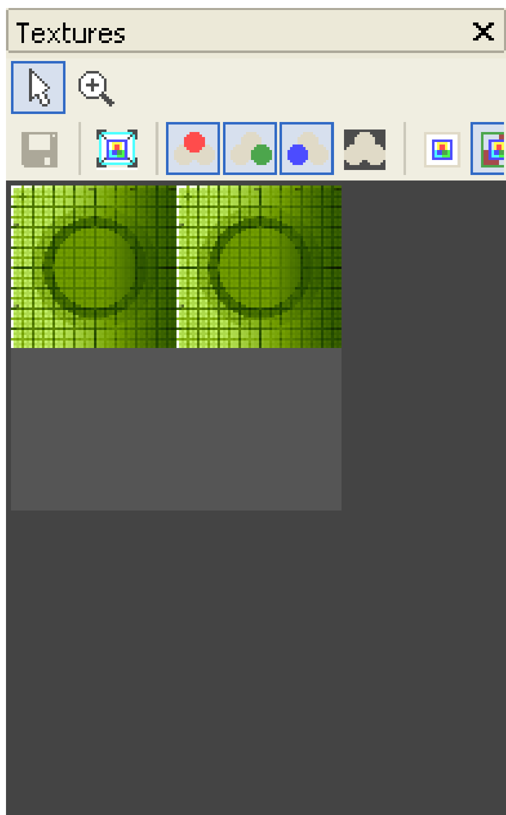
FX Composer je integrirano razvojno okruženje (IDE) za razvoj programa za sjenčanje. Dosta je jednostavan, tako da se programer vrlo brzo može priviknuti na korištenje. Za 3D umjetnika nije baš dobar jer su neke stvari prilično tehničke.

Sučelje se sastoji od tri dijela:

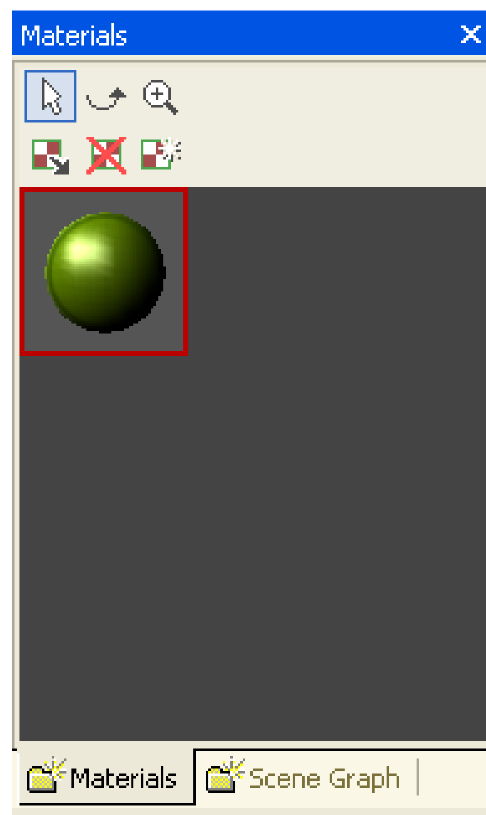
- lijevi dio je namijenjen za rad s grafom scene i materijalima
- u srednji dio se unose programi i ispisuju poruke prilikom prevođenja programa
- desni dio je namijenjen za podešavanja svojstava scene, prikaz izračunate scene i testiranje performansi programa za sjenčanje (NVShaderPerf)



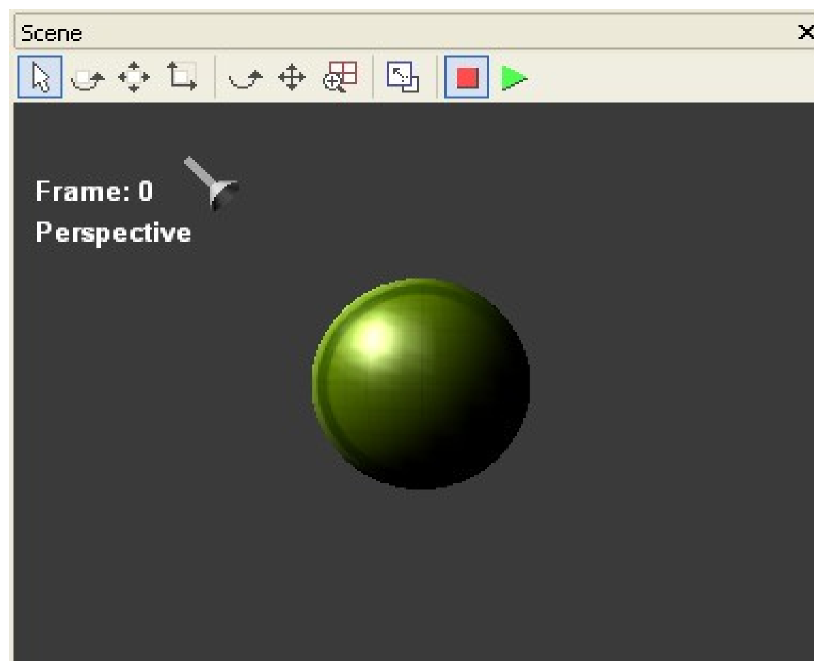
Slika 8: Osnovni raspored prozora FX Composer-a.



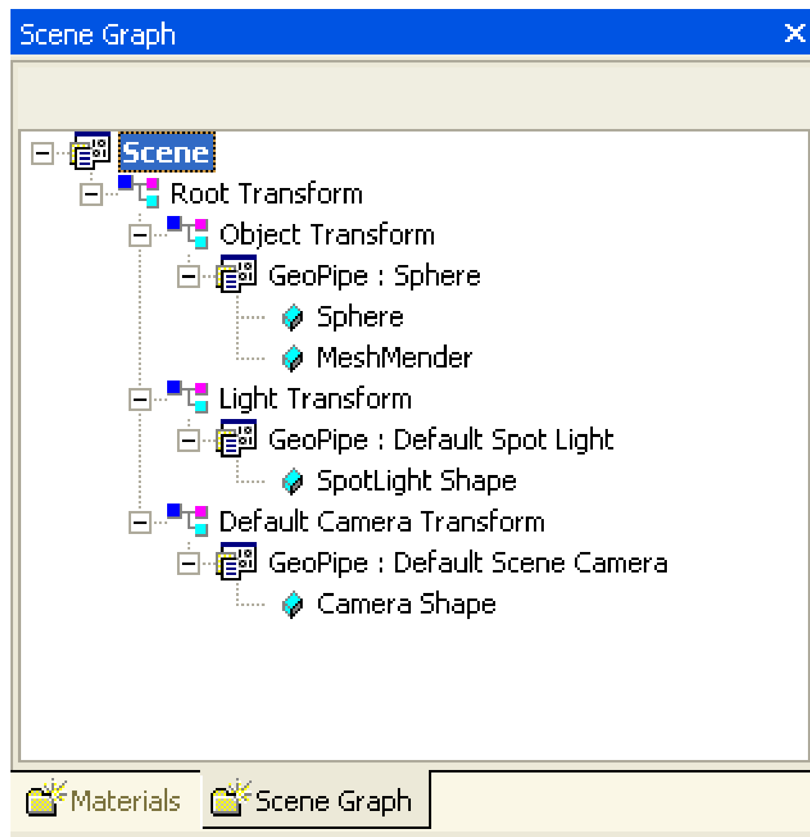
Slika 9: Prozor za prikaz teksture.



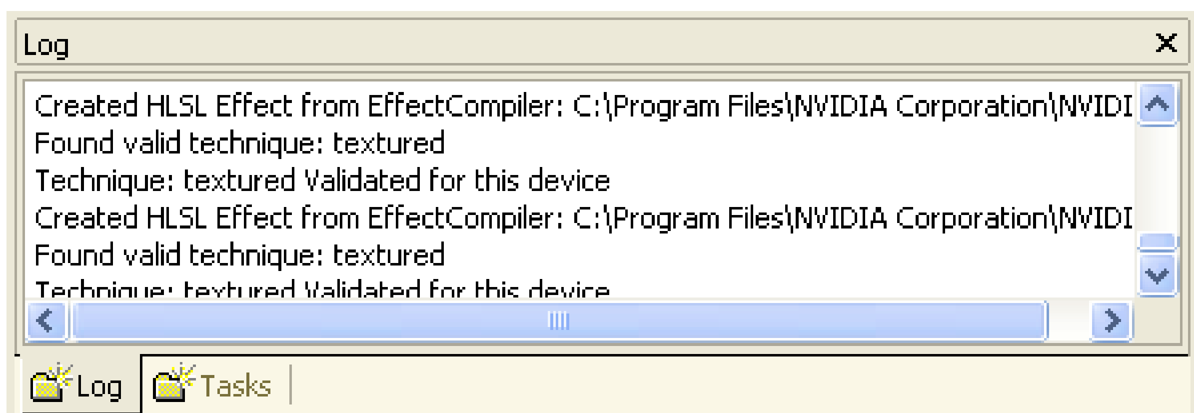
Slika 10: Prozor za prikaz materijala.



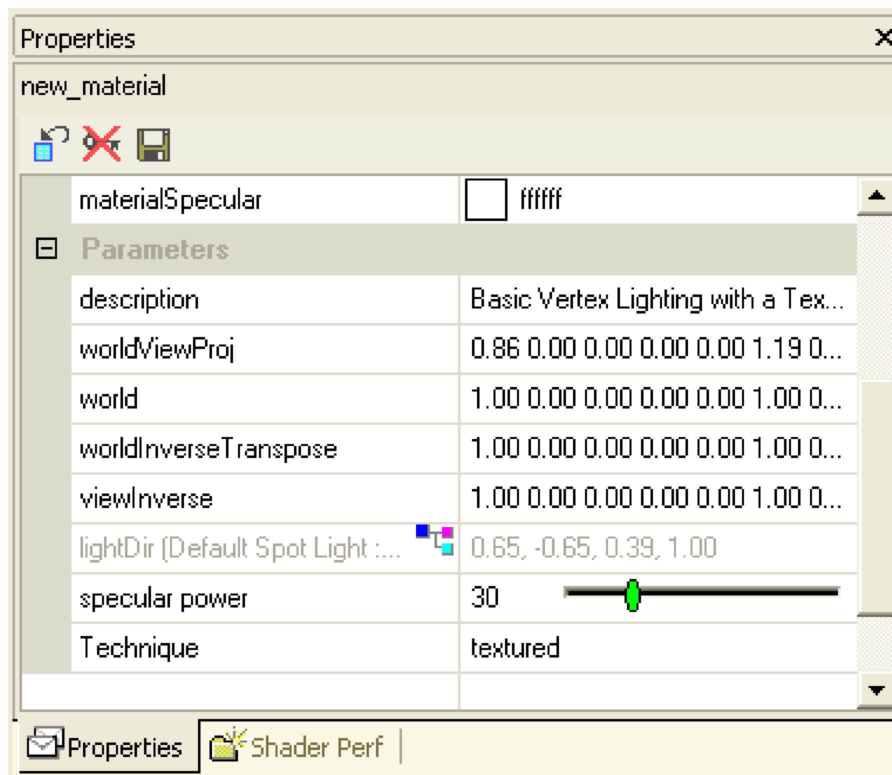
Slika 11: Prozor za prikaz scene.



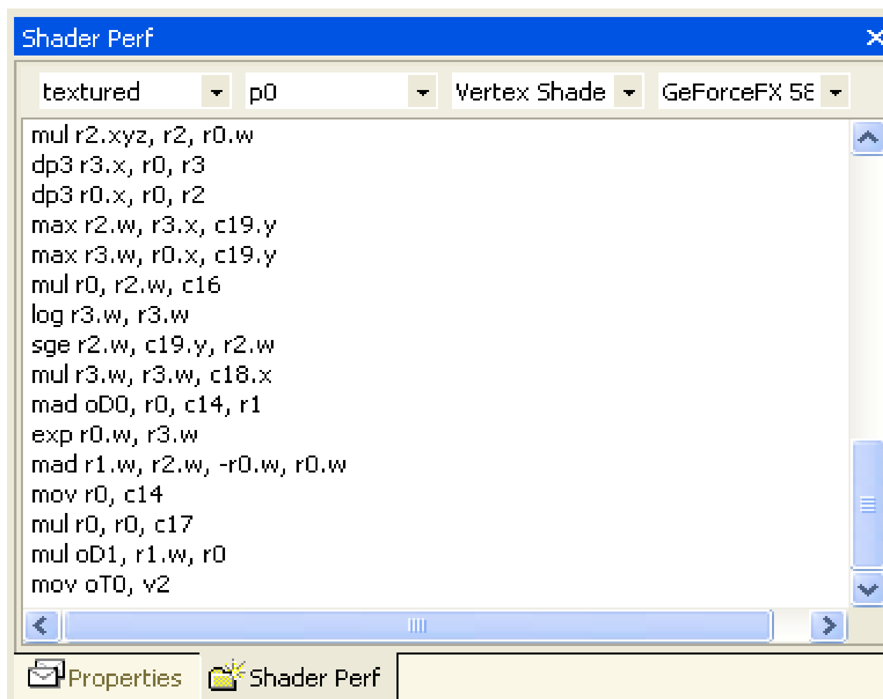
Slika 12: Prozor za prikaz grafa scene.



Slika 13: Prozor za poruke prevodioca.



Slika 14: Svojstva materijala i scene.



Slika 15: Integrirani NVShaderPerf.

Na slici 9 vidi se umanjena tekstura. Tekstura se može skalirati po želji, kao i automatski skalirati na prirodnu veličinu. Mogu se birati pojedine komponente boje (RGBA) za prikaz.

Graf scene prikazan je na slici 12. Tipovi datoteka iz kojih se može učitati graf scene su: NVB, PLY, OBJ, X. Formati OBJ i X su dosta česti u računalnoj grafici, pa stoga postoji dosta alata za modeliranje koji ih podržavaju.

Slika 14 prikazuje svojstva materija i scene. Ta svojstva su definirana u istoj datoteci kao i programi za sjenčanje Datoteka ima nastavak **fx** - odatle FX u imenu FX Composer-a (fx je česta kratica za effect).
Bitna karakteristika ove datoteke je da može sadržavati nekoliko postupaka iscrtavanja (engl. technique) od kojih svaki može sadržavati nekoliko prolaza izračuna scene (engl. pass).

Na slici 15 vidi se izlaz programa NVShaderPerf koji je opisan ranije u ovom poglavlju. U ovom konkretnom slučaju prikazan je izlaz za program za sjenčanje vrhova za grafičku karticu GeForceFX 5800 Ultra.

Primjer FX datoteke (s komentarima):

```
//Na početku ide kratak opis namjene datoteke
string description = "Basic Vertex Lighting with a Texture";

//Zatim slijedi definicija matrica za transformacije, projekcije itd.
float4x4 worldViewProj : WorldViewProjection;
float4x4 world   : World;
float4x4 worldInverseTranspose : WorldInverseTranspose;
float4x4 viewInverse : ViewInverse;

//Nakon toga se navode datoteke iz kojih se učitavaju teksture
texture diffuseTexture : Diffuse
<
      string ResourceName = "default_color.dds";
>;

//Slijedi definicija materijala (tu pripadaju i svjetla)
float4 lightDir : Direction
<
      string Object = "DirectionalLight";
    string Space = "World";
> = {1.0f, -1.0f, 1.0f, 0.0f};

float4 lightColor : Diffuse
<
    string UIName = "Diffuse Light Color";
    string Object = "DirectionalLight";
> = {1.0f, 1.0f, 1.0f, 1.0f};

float4 lightAmbient : Ambient
<
    string UIWidget = "Ambient Light Color";
    string Space = "material";
> = {0.0f, 0.0f, 0.0f, 1.0f};

float4 materialDiffuse : Diffuse
```

```
<
    string UIWidget = "Surface Color";
    string Space = "material";
> = {1.0f, 1.0f, 1.0f, 1.0f};

float4 materialSpecular : Specular
<
      string UIWidget = "Surface Specular";
      string Space = "material";
> = {1.0f, 1.0f, 1.0f, 1.0f};

float shininess : SpecularPower
<
    string UIWidget = "slider";
    float UIMin = 1.0;
    float UIMax = 128.0;
    float UIStep = 1.0;
    string UIName = "specular power";
> = 30.0;


//Definicija struktura za prenošenje parametara u i iz programa za sjenčanje vrhova
struct vertexInput {
    float3 position             : POSITION;
    float3 normal               : NORMAL;
    float4 texCoordDiffuse      : TEXCOORD0;
};

struct vertexOutput {
    float4 hPosition            : POSITION;
    float4 texCoordDiffuse      : TEXCOORD0;
    float4 diffAmbColor         : COLOR0;
    float4 specCol              : COLOR1;
};


//Program za sjenčanje vrhova
vertexOutput VS_TransformAndTexture(vertexInput IN)
{
    vertexOutput OUT;
    OUT.hPosition = mul( float4(IN.position.xyz , 1.0) , worldViewProj);
    OUT.texCoordDiffuse = IN.texCoordDiffuse;

      //calculate our vectors N, E, L, and H
      float3 worldEyePos = viewInverse[3].xyz;
    float3 worldVertPos = mul(IN.position, world).xyz;
      float4 N = mul(IN.normal, worldInverseTranspose); //normal vector
    float3 E = normalize(worldEyePos - worldVertPos); //eye vector
    float3 L = normalize( -lightDir.xyz); //light vector
    float3 H = normalize(E + L); //half angle vector

      //calculate the diffuse and specular contributions
    float  diff = max(0 , dot(N,L));
    float  spec = pow( max(0 , dot(N,H) ) , shininess );
    if( diff <= 0 )
    {
        spec = 0;
    }

      //output diffuse
    float4 ambColor = materialDiffuse * lightAmbient;
    float4 diffColor = materialDiffuse * diff * lightColor ;
```

```
    OUT.diffAmbColor = diffColor + ambColor;

       //output specular
    float4 specColor = materialSpecular * lightColor * spec;
    OUT.specCol = specColor;

    return OUT;
}


//Podešavanje filtriranja teksture
sampler TextureSampler = sampler_state
{
    texture = <diffuseTexture>;
    AddressU  = CLAMP;
    AddressV  = CLAMP;
    AddressW  = CLAMP;
    MIPFILTER = LINEAR;
    MINFILTER = LINEAR;
    MAGFILTER = LINEAR;
};


//Program za sjenčanje fragmenata
float4 PS_Textured( vertexOutput IN): COLOR
{
  float4 diffuseTexture = tex2D( TextureSampler, IN.texCoordDiffuse );
  return IN.diffAmbColor * diffuseTexture + IN.specCol;
}


//Ova datoteka ima samo jedan postupak iscrtavanja sa samo jednim prolazom
technique textured
{
    pass p0
    {
             VertexShader = compile vs_1_1 VS_TransformAndTexture();
             PixelShader  = compile ps_1_1 PS_Textured();
    }
}
```

Korištenje više postupaka iscrtavanja može se učiniti pomalo čudnim, ali postoje dva dobra razloga za to. Prvi razlog je taj da se isti postupak iscrtavanja može napraviti za različite generacije grafičkih procesora i promatrati da li daju isti rezultat (izgled scene).
Drugi razlog je taj da možemo imati samo jednu datoteku sa svim programima za sjenčanje koji se koriste za izračun scene, npr. program za preslikavanje izbočina prilikom iscrtavanja zida i program koji izračunava refleksiju na mramornom podu.

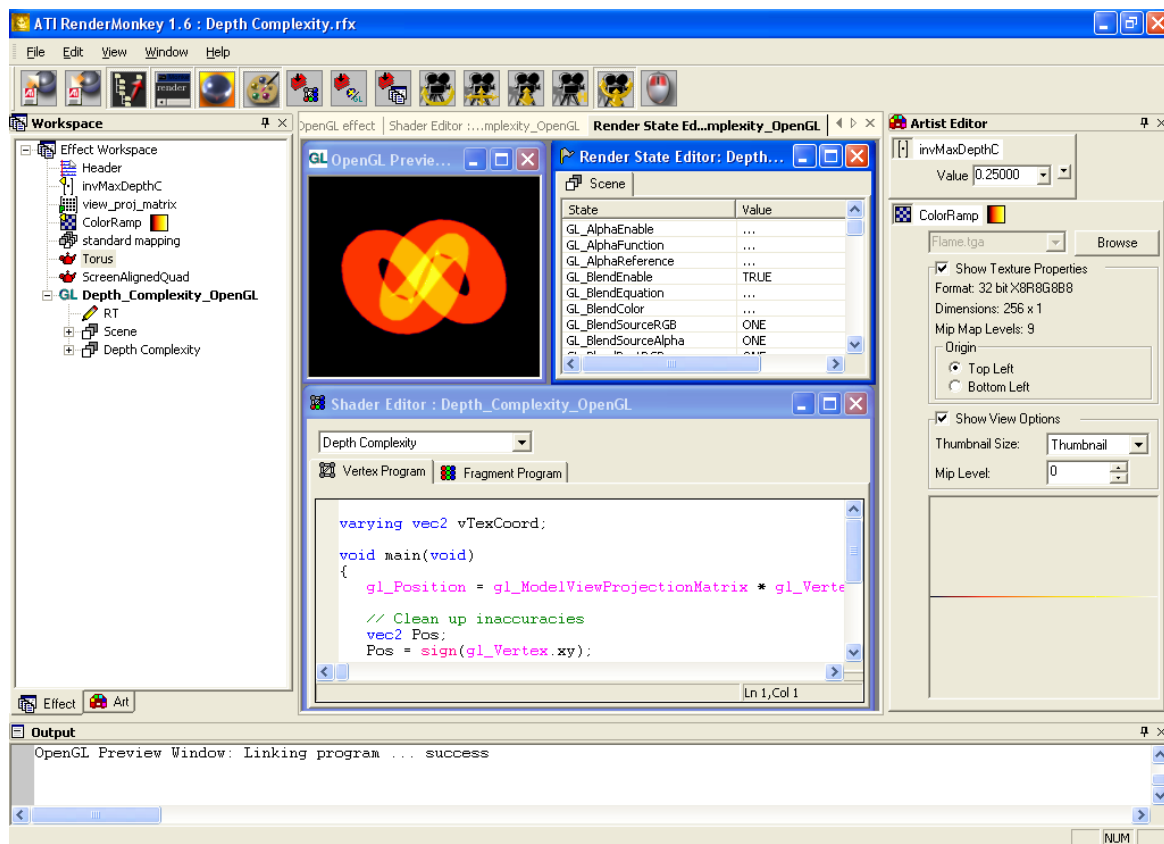
6.2.3. RenderMonkey

RenderMonkey je integrirano razvojno okruženje namijenjeno za izradu programa za sjenčanje i sve ostale poslove koje obavlja 3D umjetnik. Nešto je kompliciraniji od FX Composer-a.

Za zapis programa za sjenčanje, materijala i ostalih atributa koristi **rfx** datoteku. Ta datoteka ima sličan format kao i fx datoteka od FX Composera, samo što je zapisana kao XML.

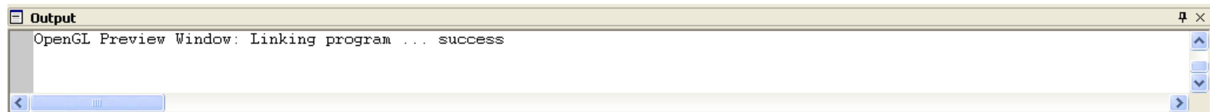
Sučelje se sastoji od slijedećih dijelova:

- dno prozora – poruke prevodioca
- alatna traka – manipuliranje scenom, prevođenje programa za sjenčanje, odabir vidljivih prozora
- sredina prozora – ovdje se otvaraju pomoćni prozori za pregled scene, pisanje programa za sjenčanje, podešavanje OpenGL varijabli stanja
- lijevi dio – graf scene i smanjeni dio grafa scene koji je dovoljan za 3D umjetnika
- desni dio – podešavanje atributa materijala koje koristi 3D umjetnik

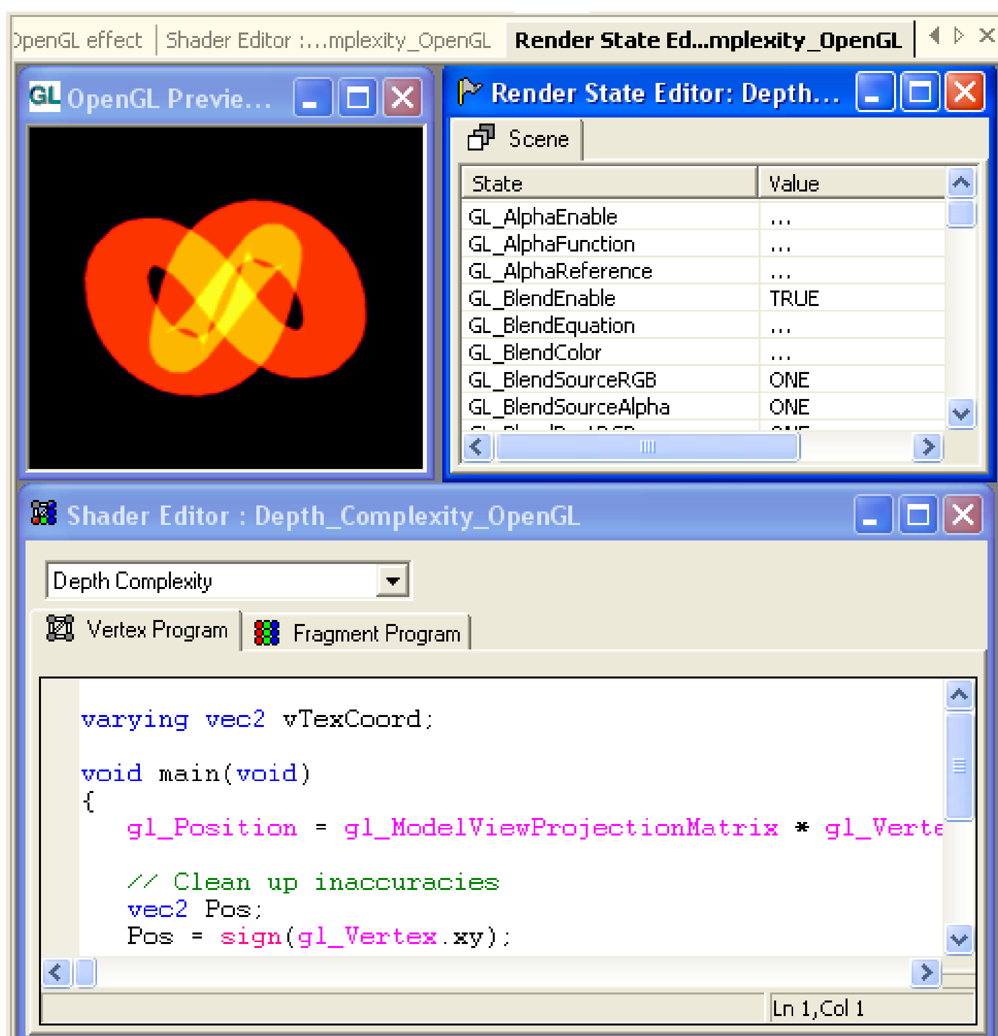


Slika 16: Osnovni izgled RenderMonkey-a.

Slika 17: Alatna traka.



Slika 18: Poruke prevodioca.

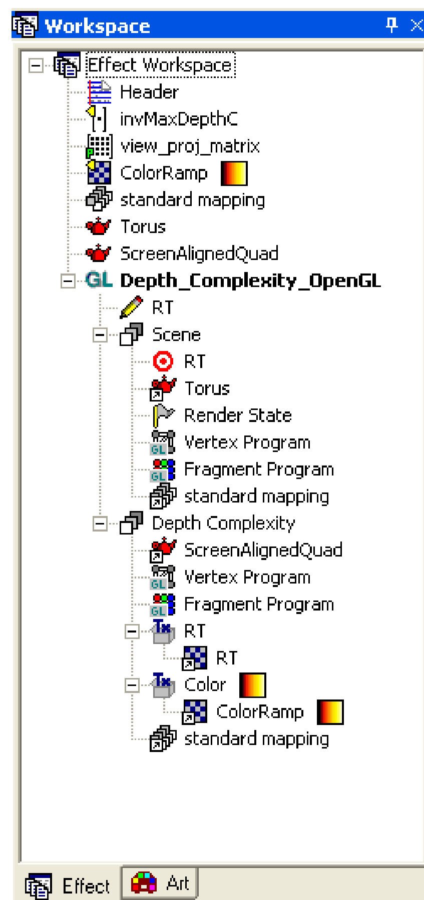


Slika 19: Srednji dio prozora.

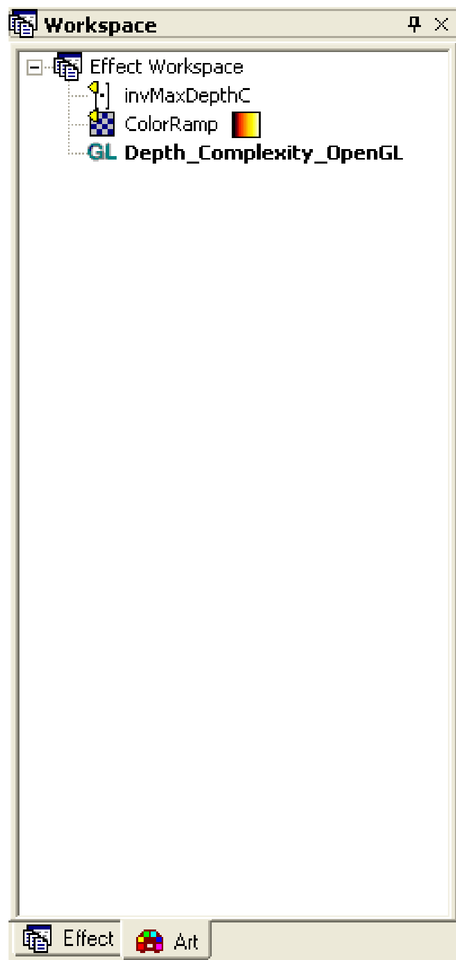
Na gornjoj slici je prikazan srednji dio prozora u kojem su otvorena tri prozora. Gornji lijevi prozor je gotova scena. Gornji desni prozor sadrži varijable stanja OpenGL-a.

Osim manipuliranja varijablama stanja, imamo mogućnost odabira prostora za pohranu rezultata izračuna scene (engl render target). Taj prostor može biti prostor video memorije iz kojega se direktno iscrtava na zaslon ili to može biti tekstura. Pohrana rezultata izračuna u teksturu koristi se u najsloženijim višeprolaznim algoritmima. Npr. ako imamo scenu sa zrcalom, prvo izračunamo cijelu scenu i pohranimo je u teksturu. Nakon toga tu teksturu zalijepimo na zrcalo rotiranu oko y osi. Kao rezultat dobijemo cjelokupnu scenu sa zrcalom koje prikazuje reflektirani dio scene.

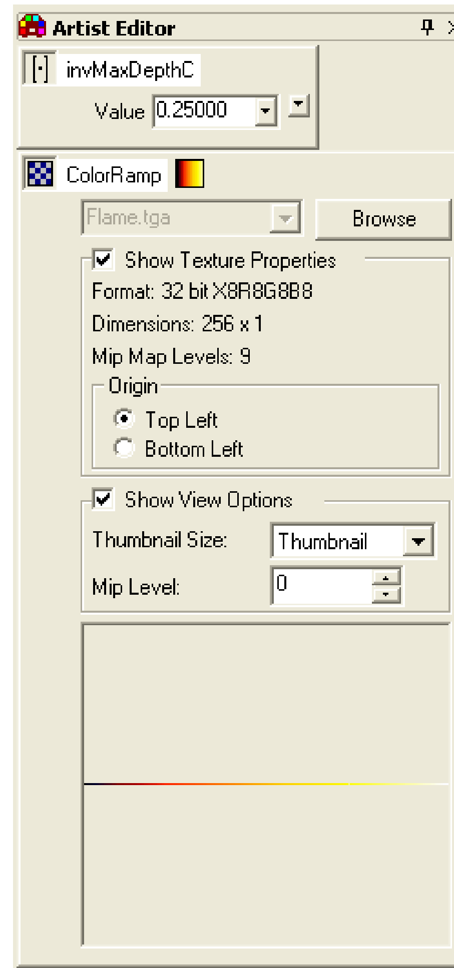
Donji srednji dio sadrži prozor koji služi za uređivanje programa za sjenčanje. Može se primijetiti da je program obojan u više boja, zavisno o sintaksnim elementima – varijable, funkcije, ugrađene varijable, komentari itd.



Slika 20: Prozor s grafom scene.



Slika 21: Sažeti graf scene – prikladan za 3D umjetnika



Slika 22: Prozor za podešavanje atributa materijala

7. Programski primjeri

Svi programski primjeri izrađeni su pomoću programskog alata RenderMonkey. Ima ih ukupno četiri:

- preslikavanje izbočina
- valovita deformacija
- izrezivanje traka
- miješanje tekstura

Svi programski primjeri su napravljeni u Cg-u i GLSL-u da bi se omogućila usporedba između ta dva jezika.

Ovdje će primjeri biti samo kratko opisani, izlistan kod programa za sjenčanje i prikazane slike za svaki od njih. Za svaki primjer bit će napravljeno nekoliko testova performansi.

Cjeloviti projekti (datoteke s nastavkom .rfx) mogu se naći na priloženom CD-u. Korišteni su materijali koji idu uz RenderMonkey da bi se omogućila prenosivost. Jedino je za primjer izrezivanja traka korištena dodatna slika "interlace.bmp" koja je priložena na CD-u, u istom direktoriju kao i projekti.

Za testiranje performansi korišten je programski alat NVShaderPerf. Testirane su performanse samo programa za sjenčanje fragmenata napisanih u programskoj jeziku GLSL. NVShaderPerf prije testiranja performansi prevodi program u asemblerski oblik.

NVShaderPerf na osnovu broja naredbi asemblerskog programa izračunava broj ciklusa potreban za njegovo izvođenje. Na osnovu broja ciklusa i specifikacija tvrtke NVIDIA za određenu arhitekturu/generaciju grafičkih kartica, izračunava se broj iscrtanih piksela u sekundi. Točan broj iscrtanih piksela u sekundi ovisi o frekvencijama GPU-a i RAM-a, kao i o širini memorijske riječi.

Tablica 16: Specifikacije za nekoliko grafičkih kartica tvrtke NVIDIA

Oznaka arhitekture	Naziv grafičke kartice	Vrhova u sekundi [x10^6]	Piksela u sekundi [x10^9]	Frekvencija GPU-a [MHz]	Frekvencija RAM-a [MHz]
NV34	GeForce FX 5200	80	1.1	250	450
NV30	GeForce FX 5800	200	4	400	800
NV40-GT	GeForce 6800 GT	600	6.4	400	1050
G70	GeForce 7800 GT	1000	9.6	450	1200

7.1. Preslikavanje izbočina

Ovaj programski primjer prikazuje najprimitivniji oblik preslikavanja izbočina. Preslikavanje izbočina je postupak kojim se glatka ploha učini da izgleda kao da je hrapava, iako ustvari nije. To se postiže tako da se u teksturu pohrane lažne normale koje se onda koriste kod izračuna osvjetljenja umjesto pravih normala objekta.

Kako su normale normalizirane, njihove komponente su u rasponu [-1, 1]. Teksture koriste vrijednosti u rasponu [0, 1] pa je zbog toga potrebno normale smjestiti u taj raspon na slijedeći način:

```
nova_vrijednost = 0.5  * stara_vrijednost + 0.5
```

Za obrnuti postupak vrijedi formula:

```
stara_vrijednost = 2 * (nova_vrijednost - 0.5)
```

Valja zapamtiti da je ovo najprimitivniji oblik preslikavanja izbočina koji dobro radi samo s okomitim ravnim plohama.

Normalizacija se u ovom primjeru ne obavlja matematičkim putem, nego pristupom kubnoj teksturi čije komponente boje su u biti normalizirani koeficijenti vektora.
Na taj način je izvršena vremenska ušteda od nekoliko ciklusa prilikom svake normalizacije. Jedini nedostatak ovog postupka je manja točnost izračuna, što ionako nije bitno jer je i samo preslikavanje izbočina aproksimativan postupak.

7.1.1. Cg

Program za sjenčanje vrhova:

```
uniform float3 lightPos;
uniform float4x4 matViewProjection;

struct VS_INPUT
{
   float3 position : POSITION0;
   float2 texcoord : TEXCOORD0;
};

struct VS_OUTPUT
{
   float4 position : POSITION0;
   float2 texcoord : TEXCOORD0;
   float3 lightDir : TEXCOORD1;
};

VS_OUTPUT vs_main(VS_INPUT Input)
{
```

```
   VS_OUTPUT Output;

   Output.position = mul(matViewProjection, float4(Input.position, 1));
   Output.texcoord = Input.texcoord;
   Output.lightDir = lightPos - Input.position.xyz;

   return Output;
}
```

Program za sjenčanje fragmenata:

```
uniform sampler2D tex0;
uniform samplerCUBE texNormal;

float3 expand(float3 v)
{
   return (v - 0.5) * 2;
}

float4 ps_main(float2 texcoord : TEXCOORD0,
               float3 lightDir : TEXCOORD1
) : COLOR0
{
   float3 lightTex = texCUBE(texNormal, lightDir).xyz;
   float3 light = expand(lightTex);
   float3 normalTex = tex2D(tex0, texcoord);
   float3 normal = expand(normalTex);

   return dot(normal, light);
}
```

7.1.2. GLSL

Program za sjenčanje vrhova:

```
uniform vec3 lightPos;
varying vec2 texcoord;
varying vec3 lightDir;

void main(void)
{
   gl_Position = gl_ModelViewProjectionMatrix * vec4(gl_Vertex.xyz,
                 1.0);
   texcoord = gl_MultiTexCoord0.xy;
   lightDir = lightPos - gl_Vertex.xyz;
}
```

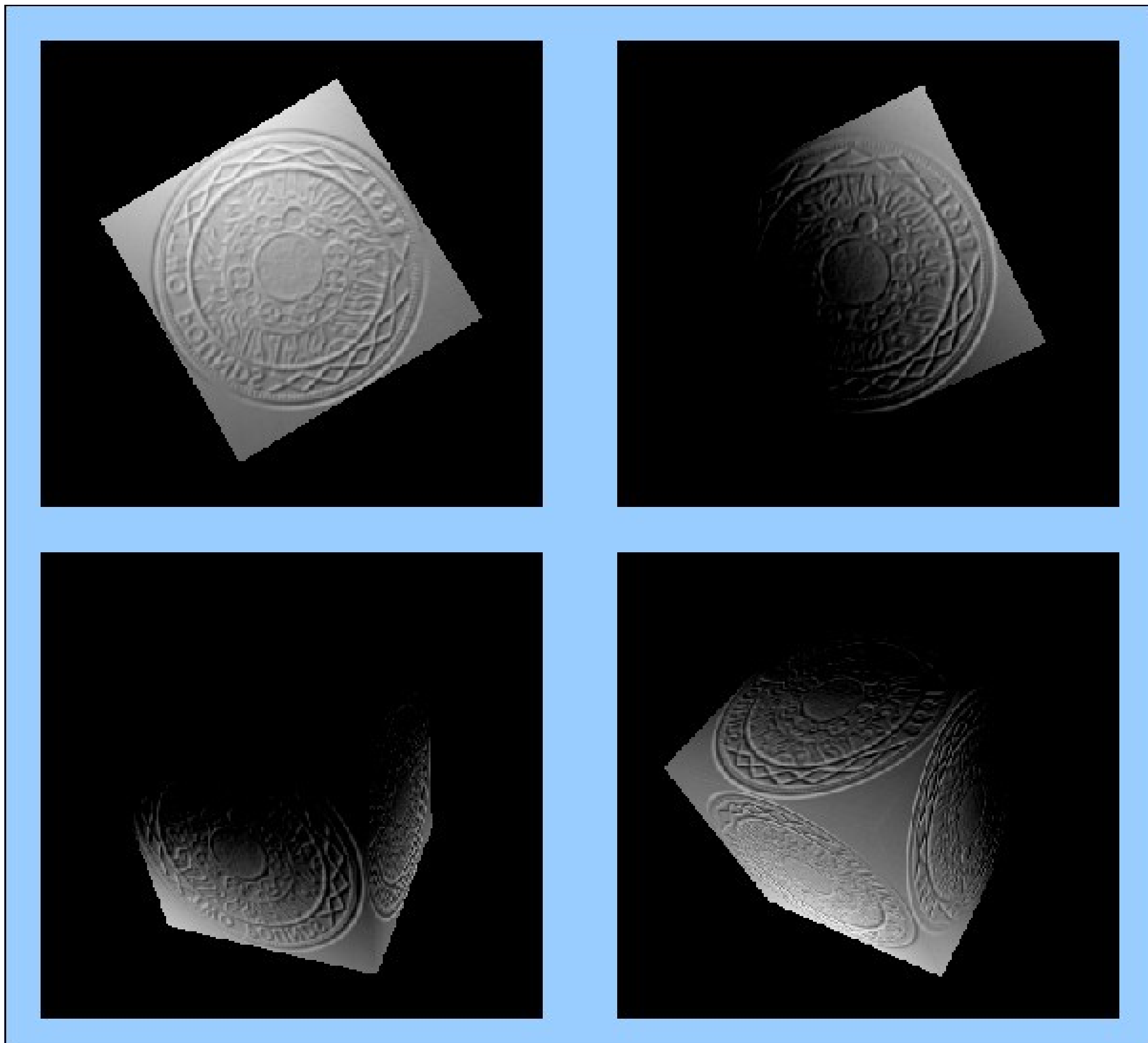
Program za sjenčanje fragmenata:

```
uniform sampler2D tex0;
uniform samplerCube texNormal;
varying vec2 texcoord;
varying vec3 lightDir;

vec3 expand(vec3 v)
{
   return (v - 0.5) * 2.0;
}

void main(void)
{
   vec3 lightTex = textureCube(texNormal, lightDir).xyz;
   vec3 light = expand(lightTex);
   vec3 normalTex = texture2D(tex0, texcoord).xyz;
   vec3 normal = expand(normalTex);

   gl_FragColor = vec4(dot(normal, light));
}
```

7.1.3. Rezultati

Tablica 17: Rezultati simulacija programa za sjenčanje fragmenata programskog primjera za preslikavanje izbočina

Broj instrukcija	7			
Broj korištenih registara	2			
Arhitektura	NV34	NV30	NV40-GT	G70
Broj ciklusa	4	4	3	3
Broj piksela u sekundi [x10^6]	200	500	1870	3200



Slika 23: Rezultati programa za preslikavanje izbočina.

7.2. Valovita deformacija

Valovita deformacija je vrlo jednostavan postupak kojim se objekt učini tako da izgleda kao da mu je površina načinjena od valova.

```
pomak = faktor * sin(položaj.y * freq * vrijeme)
novi_polozaj = položaj + normala * pomak
```

Dovoljno je napraviti deformaciju po samo jednoj osi (u ovom slučaju y). Osim deformacija mora se napraviti bar difuzno osvjetljenje, jer scena bez difuznog osvjetljenja izgleda kao da je dvodimenzionalna.

7.2.1. Cg

Program za sjenčanje vrhova:

```
uniform float4x4 matViewProjection;
uniform float3 polozaj_svjetla;
uniform float vrijeme;
uniform float freq;
uniform float faktor;
uniform float4 boja_svjetla;
uniform float Kd;

struct VS_INPUT
{
   float3 Position : POSITION0;
   float3 Normal   : NORMAL0;
};

struct VS_OUTPUT
{
   float4 Position : POSITION0;
   float4 Color    : COLOR0;
};

VS_OUTPUT vs_main(VS_INPUT Input)
{
   VS_OUTPUT Output;

   float pomak = faktor * sin(Input.Position.y * freq * vrijeme);
   float4 smjer = float4(Input.Normal, 0.0f);
   float4 novi_polozaj = float4(Input.Position, 1.0) + smjer * pomak;

   Output.Position = mul(matViewProjection, novi_polozaj);

   float3 L = normalize(polozaj_svjetla - novi_polozaj.xyz);
   float dif = max(dot(Input.Normal, L), 0);
   float3 difuzna = boja_svjetla.xyz * dif * Kd;

   Output.Color = float4(difuzna, 1.0f);

   return Output;
}
```

Program za sjenčanje fragmenata:

```
float4 ps_main(float4 boja : COLOR0) : COLOR0
{
   return boja;
}
```

7.2.2. GLSL

Program za sjenčanje vrhova:

```
uniform vec3 polozaj_svjetla;
uniform float vrijeme;
uniform float freq;
uniform float faktor;
uniform vec4 boja_svjetla;
uniform float Kd;

void main(void)
{

   float pomak = faktor * sin(gl_Vertex.y * freq * vrijeme);
   vec4 smjer = vec4(gl_Normal, 0.0);
   vec4 novi_polozaj = vec4(gl_Vertex.xyz, 1.0) + smjer * pomak;

   gl_Position = gl_ModelViewProjectionMatrix * novi_polozaj;

   //Oprez: razlika izmedu OpenGL i DirectX koordinata
   vec3 novi_polozaj_svjetla = polozaj_svjetla * vec3(-1.0, -1.0, 1.0);

   vec3 L = normalize(novi_polozaj_svjetla - novi_polozaj.xyz);
   float dif = max(dot(gl_Normal, L), 0.0);
   vec3 difuzna = boja_svjetla.xyz * dif * Kd;

   gl_FrontColor = vec4(difuzna, 1.0);
}
```

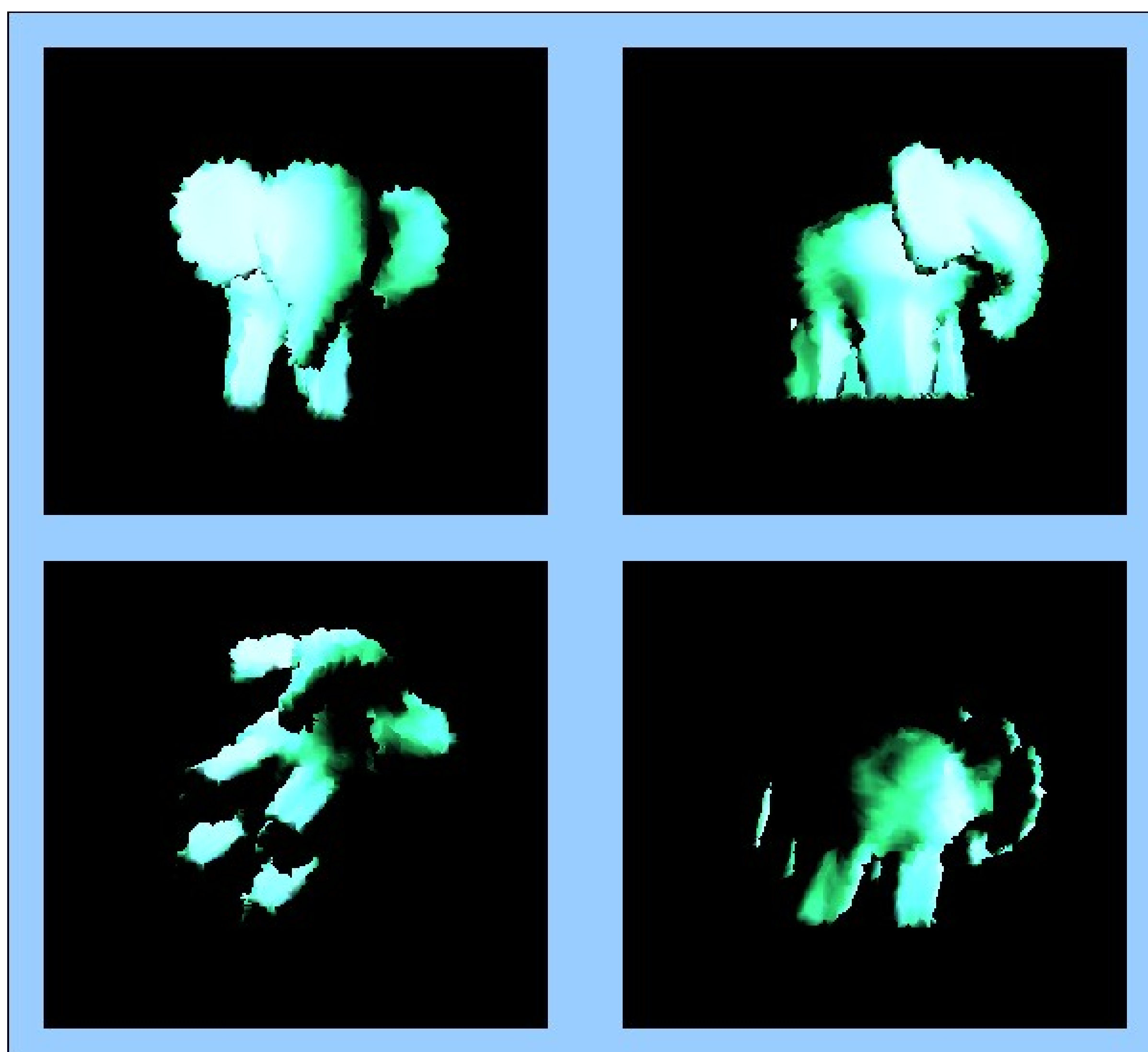
Program za sjenčanje fragmenata:

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

7.2.3. Rezultati

Tablica 18: Rezultati simulacija programa za sjenčanje fragmenata programskog primjera za valovitu deformaciju

Broj instrukcija	1			
Broj korištenih registara	0			
Arhitektura	NV34	NV30	NV40-GT	G70
Broj ciklusa	2	2	1	1
Broj piksela u sekundi [x10^6]	400	1000	5600	9600



Slika 24: Rezultati programa za valovitu deformaciju.

7.3. Izrezivanje traka

Ovaj programski primjer načinjen je tako da podsjeća na način rada televizora sa katodnom cijevi. Takvi televizori naizmjence prikazuju parne i neparne polu-slike (engl. interlaced effect). Svaka polu-slika se sastoji od traka (parnih ili neparnih).

Ovaj primjer samo podsjeća na taj efekt, jer se sastoji samo od jednog skupa traka koje još uz to ne zauzimaju ni parne ni neparne redove, nego su pokretne.

7.3.1. GLSL

Program za sjenčanje vrhova:

```
varying vec2 texCoord0;
varying vec2 texCoord1;

uniform float vrijeme;
uniform float brzina0;
uniform float brzina1;

void main(void)
{
   gl_Position = gl_ModelViewProjectionMatrix * vec4(gl_Vertex.xyz,
                  1.0);
   texCoord0 = gl_MultiTexCoord0.xy;
   texCoord1 = texCoord0;
   texCoord0.y += vrijeme * brzina0;
   texCoord1.x += vrijeme * brzina1;
}
```

Program za sjenčanje fragmenata:

```
uniform sampler2D tex0;
uniform sampler2D tex1;

varying vec2 texCoord0;
varying vec2 texCoord1;

void main(void)
{
   vec4 uzorak = texture2D(tex0, texCoord0);

   if (uzorak.x == 0.0)
    discard;

   gl_FragColor = texture2D(tex1, texCoord1);
}
```

7.3.2. Cg

Program za sjenčanje vrhova:

```
uniform float4x4 matViewProjection;
uniform float vrijeme;
uniform float brzina0;
uniform float brzina1;

struct VS_INPUT
{
   float3 Position : POSITION0;
   float2 texCoord : TEXCOORD0;
};

struct VS_OUTPUT
{
   float4 Position  : POSITION0;
   float2 texCoord0 : TEXCOORD0;
   float2 texCoord1 : TEXCOORD1;
};

VS_OUTPUT vs_main( VS_INPUT Input )
{
   VS_OUTPUT Output;

   Output.Position = mul(matViewProjection,
                        float4(Input.Position, 1.0));

   Output.texCoord0 = Input.texCoord;
   Output.texCoord1 = Input.texCoord;
   Output.texCoord0.y += vrijeme * brzina0;
   Output.texCoord1.x += vrijeme * brzina1;

   return Output;
}
```

Program za sjenčanje fragmenata:

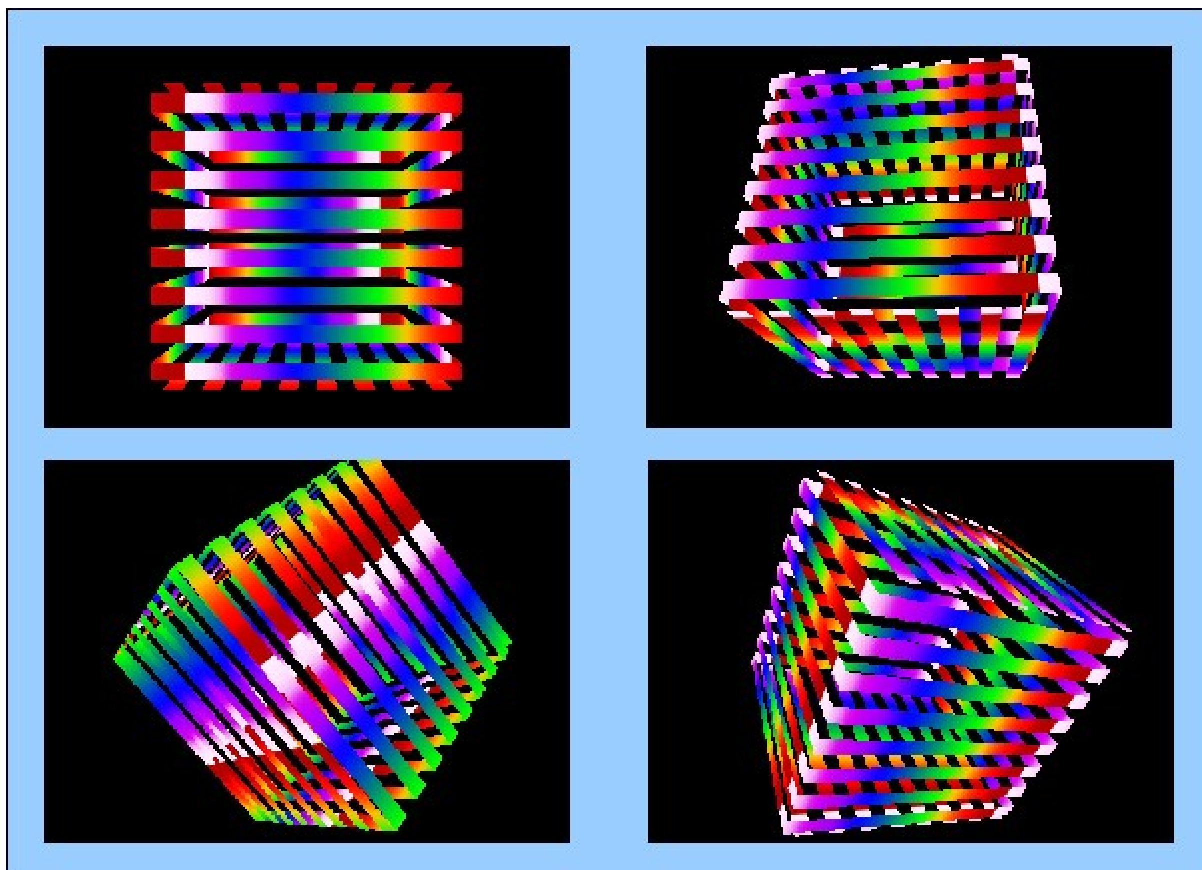
```
uniform sampler2D tex0;
uniform sampler2D tex1;

struct VS_INPUT
{
   float4 Position  : POSITION0;
   float2 texCoord0 : TEXCOORD0;
   float2 texCoord1 : TEXCOORD1;
};

float4 ps_main(VS_INPUT Input) : COLOR0
{
   float4 uzorak = tex2D(tex0, Input.texCoord0);

   if (uzorak.x == 0.0)
```

```
    discard;

   return tex2D(tex1, Input.texCoord1);
}
```

7.3.3. Rezultati

Tablica 19: Rezultati simulacija programa za sjenčanje fragmenata programskog primjera za izrezivanje traka

Broj instrukcija	6			
Broj korištenih registara	1			
Arhitektura	NV34	NV30	NV40-GT	G70
Broj ciklusa	6	6	3	3
Broj piksela u sekundi [x10^6]	133.33	333.33	1870	3200



Slika 25: Rezultati programa za izrezivanje traka.

7.4. Miješanje tekstura

Ovaj primjer koristi dva prolaza za izračun scene. Prvi prolaz izračunava scenu koja sadrži objekt koji nalikuje isprepletenim torusima i rezultat sprema u teksturu. U drugom prolazu ta se tekstura miješa sa teksturom koja prikazuje sjedište tvrtke ATI. Može se mijenjati parametar interpolacije te dvije teksture koji se nalazi u intervalu [0, 1]. Također je moguće mijenjati pomak obje teksture po X i po Y osi u intervalima od [0, 1].

7.4.1. GLSL

<u>Program za sjenčanje vrhova, prvi prolaz:</u>

```
uniform float vrijeme;
uniform float brzina;

void main(void)
{
   mat4 rotacijska;
   float sinus =  sin( brzina * vrijeme);
   float kosinus =  cos( brzina * vrijeme );

   rotacijska[0] = vec4(1.0, 0.0, 0.0, 0.0);
   rotacijska[1] = vec4(0, kosinus, -sinus, 0.0);
   rotacijska[2] = vec4(0.0, sinus, kosinus, 0.0);
   rotacijska[3] = vec4(0.0, 0.0, 0.0, 1.0);

   vec4 polozaj = rotacijska * vec4(gl_Vertex.xyz, 1.0);

   gl_Position = gl_ModelViewProjectionMatrix * polozaj;
}
```

<u>Program za sjenčanje fragmenata, prvi prolaz:</u>

```
void main(void)
{
   gl_FragColor = vec4( 0.4, 0.0, 0.9, 1.0 );
}
```

<u>Program za sjenčanje vrhova, drugi prolaz:</u>

```
varying vec2 texCoord0;
varying vec2 texCoord1;
uniform vec2 offset0;
uniform vec2 offset1;
```

```
void main(void)
{
   gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
   texCoord0 = gl_MultiTexCoord0.xy + offset0;
   texCoord1 = gl_MultiTexCoord0.xy + offset1;
}
```

Program za sjenčanje fragmenata, drugi prolaz:

```
uniform float interpolacija;
varying vec2 texCoord0;
varying vec2 texCoord1;
uniform sampler2D tex0;
uniform sampler2D tex1;

void main(void)
{
   vec3 boja0 = texture2D(tex0, texCoord0).xyz;
   vec3 boja1 = texture2D(tex1, texCoord1).xyz;
   gl_FragColor = vec4(mix(boja0, boja1, interpolacija), 1.0);
}
```

7.4.2. Cg

Program za sjenčanje vrhova, prvi prolaz:

```
uniform float4x4 matViewProjection;
uniform float vrijeme;
uniform float brzina;

struct VS_INPUT
{
   float4 Position : POSITION0;
};

struct VS_OUTPUT
{
   float4 Position : POSITION0;
};

VS_OUTPUT vs_main( VS_INPUT Input )
{
   VS_OUTPUT Output;

   float4x4 rotacijska;
   float sinus =  sin( brzina * vrijeme);
   float kosinus =  cos( brzina * vrijeme );

   rotacijska[0] = float4(1.0, 0.0, 0.0, 0.0);
   rotacijska[1] = float4(0, kosinus, -sinus, 0.0);
   rotacijska[2] = float4(0.0, sinus, kosinus, 0.0);
```

```
   rotacijska[3] = float4(0.0, 0.0, 0.0, 1.0);

   float4 polozaj = mul(rotacijska, float4(Input.Position.xyz, 1.0));

   Output.Position = mul( matViewProjection, polozaj );

   return Output;
}
```

Program za sjenčanje fragmenata, prvi prolaz:

```
float4 ps_main() : COLOR0
{
   return( float4( 0.4, 0.0, 0.9, 1.0 ) );
}
```

Program za sjenčanje vrhova, drugi prolaz:

```
uniform float4x4 matViewProjection;
uniform float2 offset0;
uniform float2 offset1;

struct VS_INPUT
{
   float4 Position : POSITION0;
   float2 texCoord : TEXCOORD0;
};

struct VS_OUTPUT
{
   float4 Position  : POSITION0;
   float2 texCoord0 : TEXCOORD0;
   float2 texCoord1 : TEXCOORD1;
};

VS_OUTPUT vs_main(VS_INPUT Input)
{
   VS_OUTPUT Output;

   Output.Position = mul( matViewProjection, Input.Position );

   Output.texCoord0 = Input.texCoord.xy + offset0;
   Output.texCoord1 = Input.texCoord.xy + offset1;

   return Output;
}
```

Program za sjenčanje fragmenata, drugi prolaz:

```
uniform float interpolacija;
uniform sampler2D tex0;
uniform sampler2D tex1;

struct VS_INPUT
{
   float4 Position  : POSITION0;
   float2 texCoord0 : TEXCOORD0;
   float2 texCoord1 : TEXCOORD1;
};

float4 ps_main(VS_INPUT Input) : COLOR0
{
   float3 boja0 = tex2D(tex0, Input.texCoord0).xyz;
   float3 boja1 = tex2D(tex1, Input.texCoord1).xyz;

   return float4(lerp(boja0, boja1, interpolacija), 1.0);
}
```

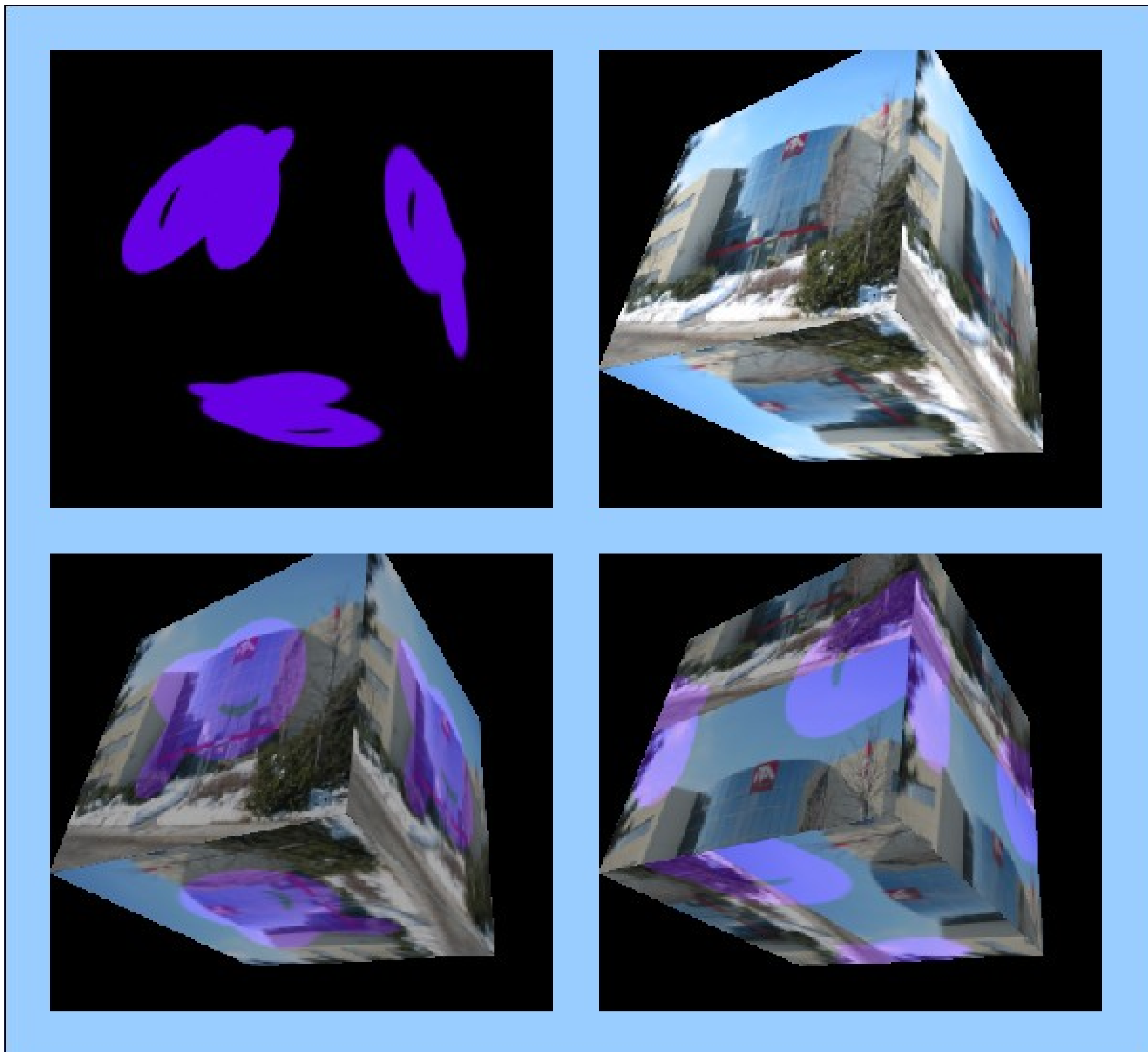
7.4.3. Rezultati

Tablica 20: Rezultati simulacija programa za sjenčanje fragmenata programskog primjera za miješanje tekstura (prvi prolaz)

Broj instrukcija	1			
Broj korištenih registara	0			
Arhitektura	NV34	NV30	NV40-GT	G70
Broj ciklusa	1	1	1	1
Broj piksela u sekundi [x10^6]	800	2000	5600	9600

Tablica 21: Rezultati simulacija programa za sjenčanje fragmenata programskog primjera za miješanje tekstura (drugi prolaz)

Broj instrukcija	5			
Broj korištenih registara	2			
Arhitektura	NV34	NV30	NV40-GT	G70
Broj ciklusa	4	4	3	3
Broj piksela u sekundi [x10^6]	200	500	1870	3200



Slika 26: Rezultati programa za miješanje tekstura.

7.5. Ostali primjeri

Pristup sličan miješanju tekstura koristi novi X poslužitelj prozora za UNIX operacijske sustave po imenu Xgl tvrtke Novell.

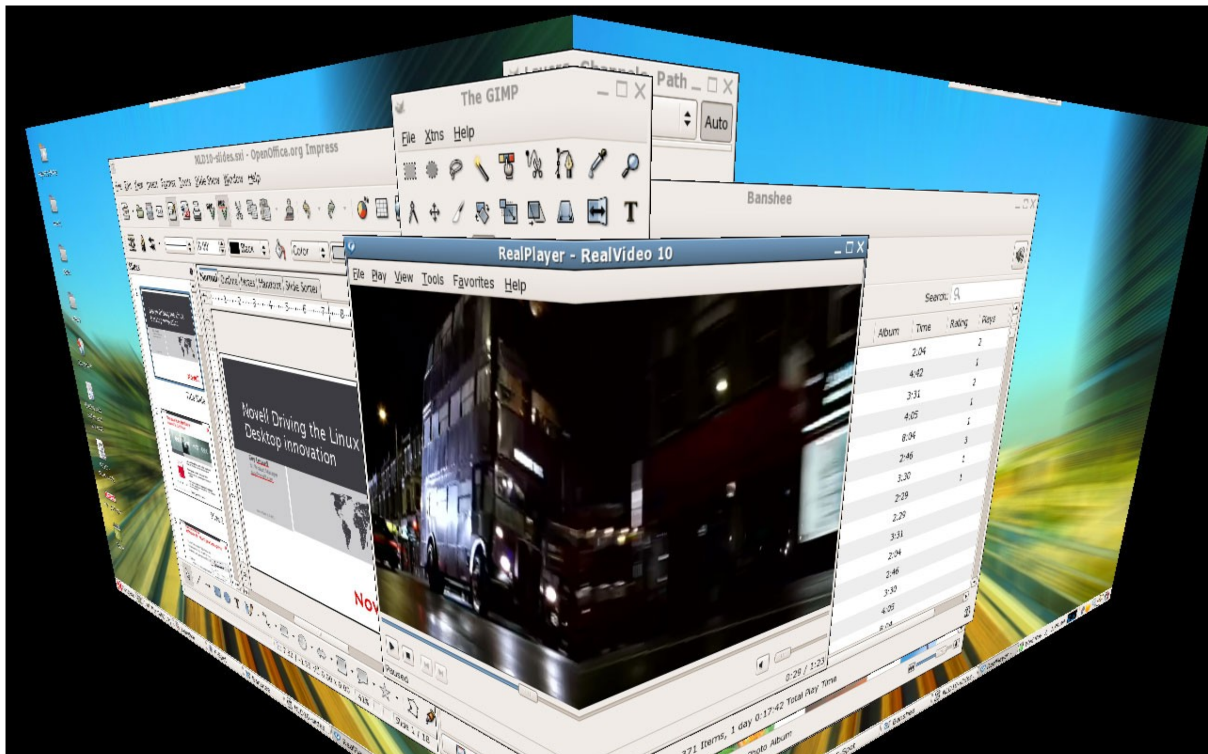
Sličnost prethodnog programskog primjera s Xgl-om je namjerna. Cilj je dokazati da će novi X poslužitelj prozora raditi i na starijem sklopovlju, recimo na grafičkoj kartici GeForce 4 sa samo 32 MB RAM-a (iako je GeForce 4 standardno dolazila sa 64 MB RAM-a).

Prednost je ta što se svaka radna površina iscrta u memoriju grafičke kartice, a zatim se

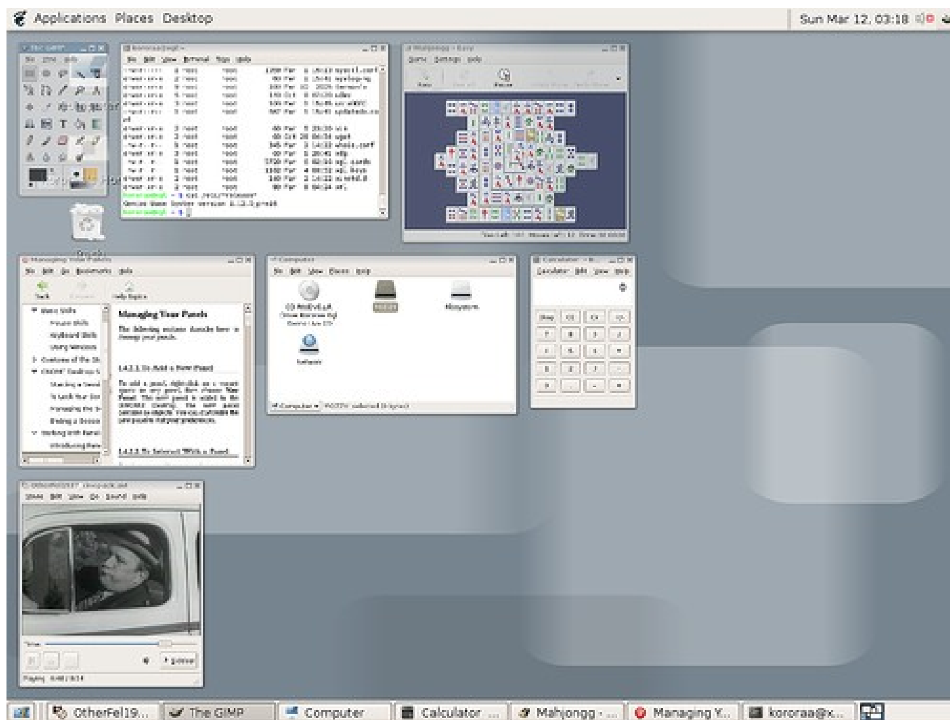
sve radne površine nalijepe na kocku. Kako su sve teksture u video memoriji, može im se pristupiti jako brzo. Osim toga, mogu se izvoditi razni efekti poput valova ili povećanja slike za osobe sa slabijim vidom.

Treba napomenuti da će Windows Vista imati 3D sučelje po imenu Aero. To sučelje će iscrtavati prozore u 3 dimenzije. Taj pristup uglavnom ima samo nedostatke: teža navigacija i daleko brže i skuplje sklopovlje koje je potrebno za iscrtavanje.

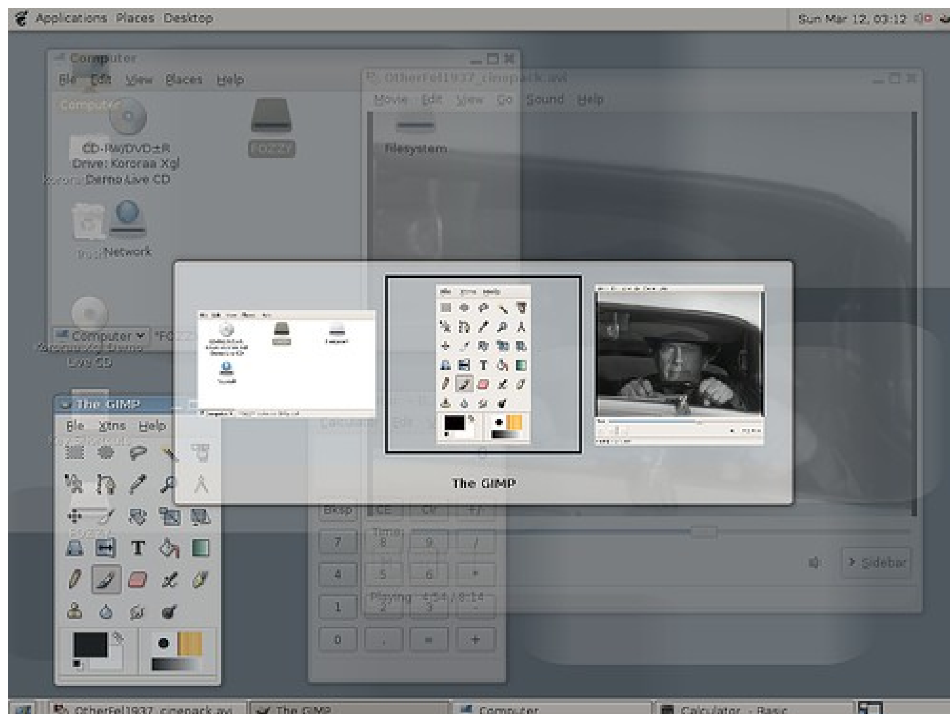
Slijedeće slike prikazuju Xgl poslužitelj prozora s menadžerom prozora GNOME. Sučelje Aero neće biti prikazano zato što je pod vlasničkom licencom (engl. propriatery software), dok je Xgl pod licencom otvorenog koda (engl. open source software).



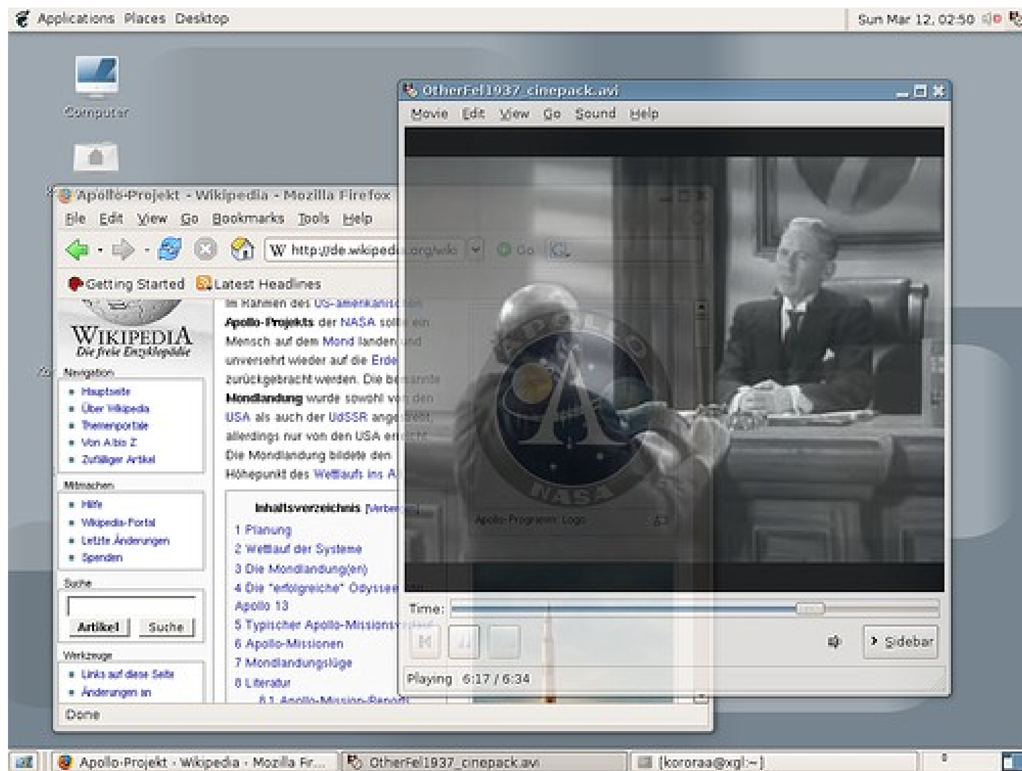
Slika 27: Programi RealPlayer i GIMP prelaze preko brida kocke (dvije susjedne radne površine).



Slika 28: Automatski posloženi prozori.



Slika 29: Kruženje između otvorenih prozora.



Slika 30: Program Totem Movie Player prikazuje film u poluprozirnom prozoru preko programa Firefox.

8. Programski zadatak

U ovom poglavlju biti će objašnjeno kako se može napraviti cjelovito programsko rješenje uz vrlo malo truda. To je moguće zahvaljujući raznim alatima za izradu skica, modela i grafa scene.

Svi programski alati koji će ovdje biti opisani su isključivo otvorenog koda. To nam daje mogućnost da ih prilagođavamo na razne načine kako bi si što više olakšali posao. Treba napomenuti da su programski alati otvorenog koda i inače konstruirani tako da direktno podržavaju ostale popularne programske alate otvorenog koda, što kod programskih alata zatvorenog koda nije slučaj.

Korišteni programski alati:

- Inkscape – alat za vektorsku grafiku (http://www.inkscape.org/)
- Blender – alat za 3D modeliranje (http://www.blender.org)
- OGRE – biblioteka namjenjena za iscrtavanje 3D scene (http://www.ogre3d.org)

Svaki programski alat biti će ukratko opisan.

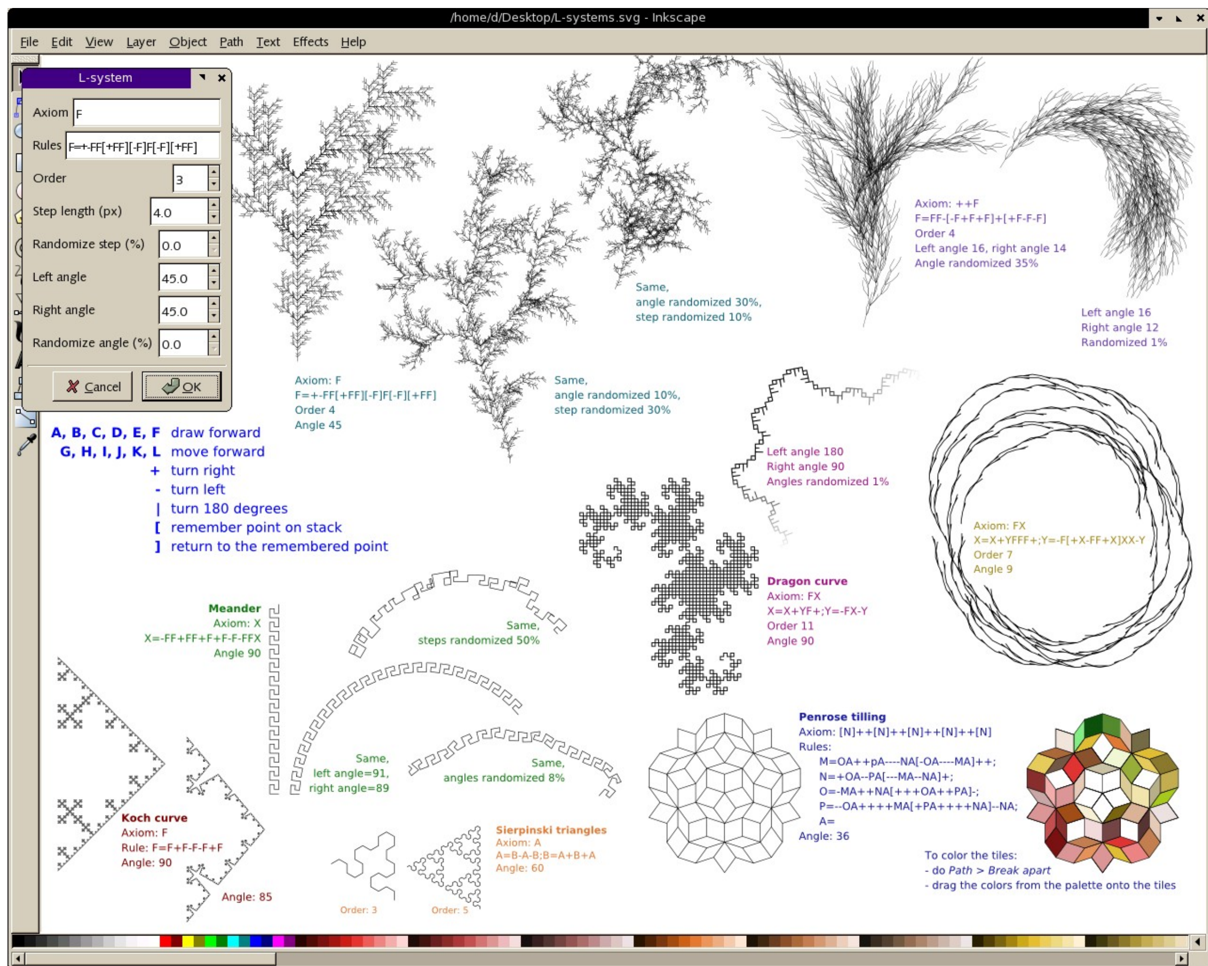
8.1. Inkscape

Inkscape je programski alat namijenjen za izradu 2D crteža u vektorskom obliku. Njegove mogućnosti slične su onima koje imaju mnogo popularniji programski alati zatvorenog koda kao npr. Illustrator, FreeHand, CorelDraw i Xara X.

Osnovni format zapisa crteža je SVG (**S**calabe **V**ector **G**raphics). To je format koji za spremanje podataka koristi XML, što ga za razliku od raznih binarnih formata, čini prenosivim na gotovo sve platforme.

Inkscape je vrlo napredan alat koji podržava slojeve (engl. layer), rasterske slike kao predložak za crteže, krivulje, uređivanje crteža preko ručnog unosa XML-a itd.

Napomena: Inspiracija za ovo poglavlje je članak „Modelling Seminaarinmäki buildings with Inkscape and Blender“ s web stranice http://www.arthis.jyu.fi/bridge/inkscape.php, autora Saana Tammisto i Ari Häyrinen. Odnosi se na projekt sveučilišta Jyväskylä (Finska) u svezi očuvanja kulturne baštine uz pomoć visokih tehnologija.

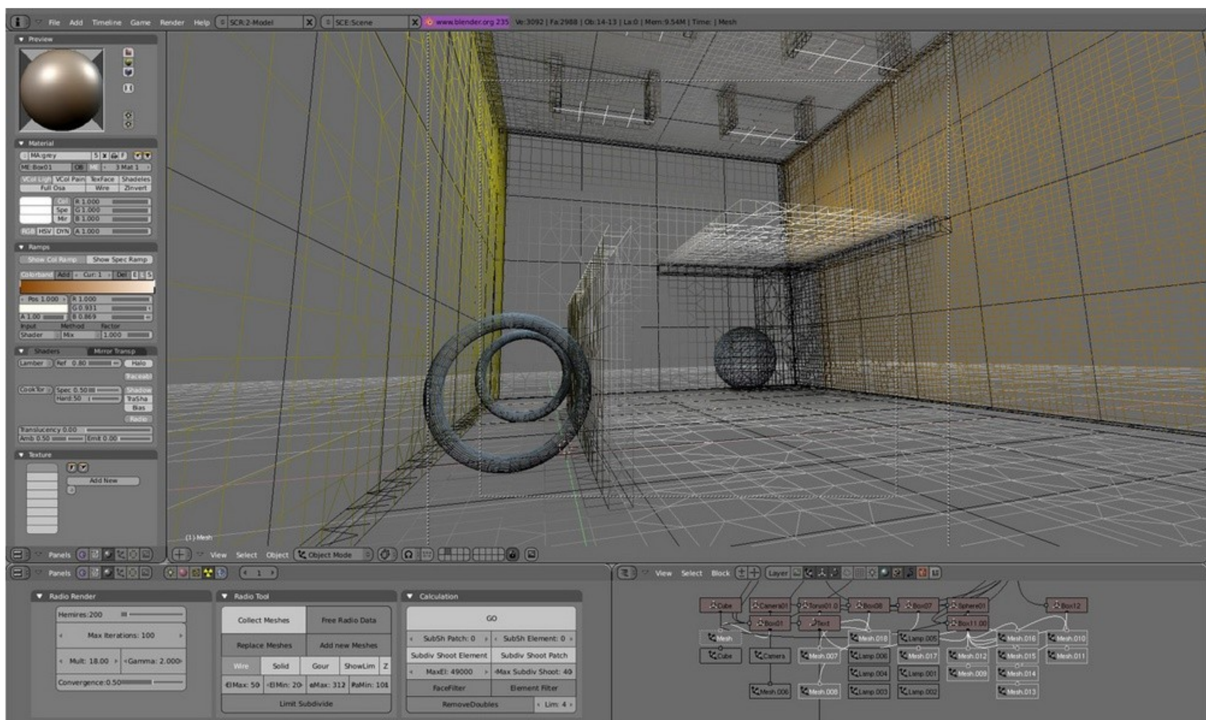


Slika 31: Prikaz Lindenmayer-ovih (L) sustava u Inkscape-u.

8.2. Blender

Blender je programski alat za 3D modeliranje, animaciju, iscrtavanje, post-produkciju, interaktivno manipuliranje scenom i reprodukciju 3D materijala.
Nastao je 1995. godine kao komercijalni proizvod tvrtke za 3D animaciju po imenu NeoGeo (Nizozemska). Blender je od 2002. godine program otvorenog koda (to je postignuto zahvaljujući neprofitnoj organizaciji posvećenoj Blender-u koja je prikupila 100.000 € za otkup licence).

Svi profesionalni alati za 3D modeliranje imaju svoj skriptni jezik za proširivanje njihovih mogućnosti i rade na svega jednom ili dva operacijska sustava. Blender je tu mnogo napredniji. Kao skriptni jezik koristi Python – vrlo popularan objektno orijentirani skriptni jezik. Cjelokupno sučelje Blender-a iscrtava se u OpenGL-u što ga čini neovisnim o sustavu prozora operacijskog sustava. Prednost toga je da radi na svim operacijskim sustavima koji podržavaju OpenGL.



Slika 32: Blender s modelom scene, grafom scene i vrijednostima materijala

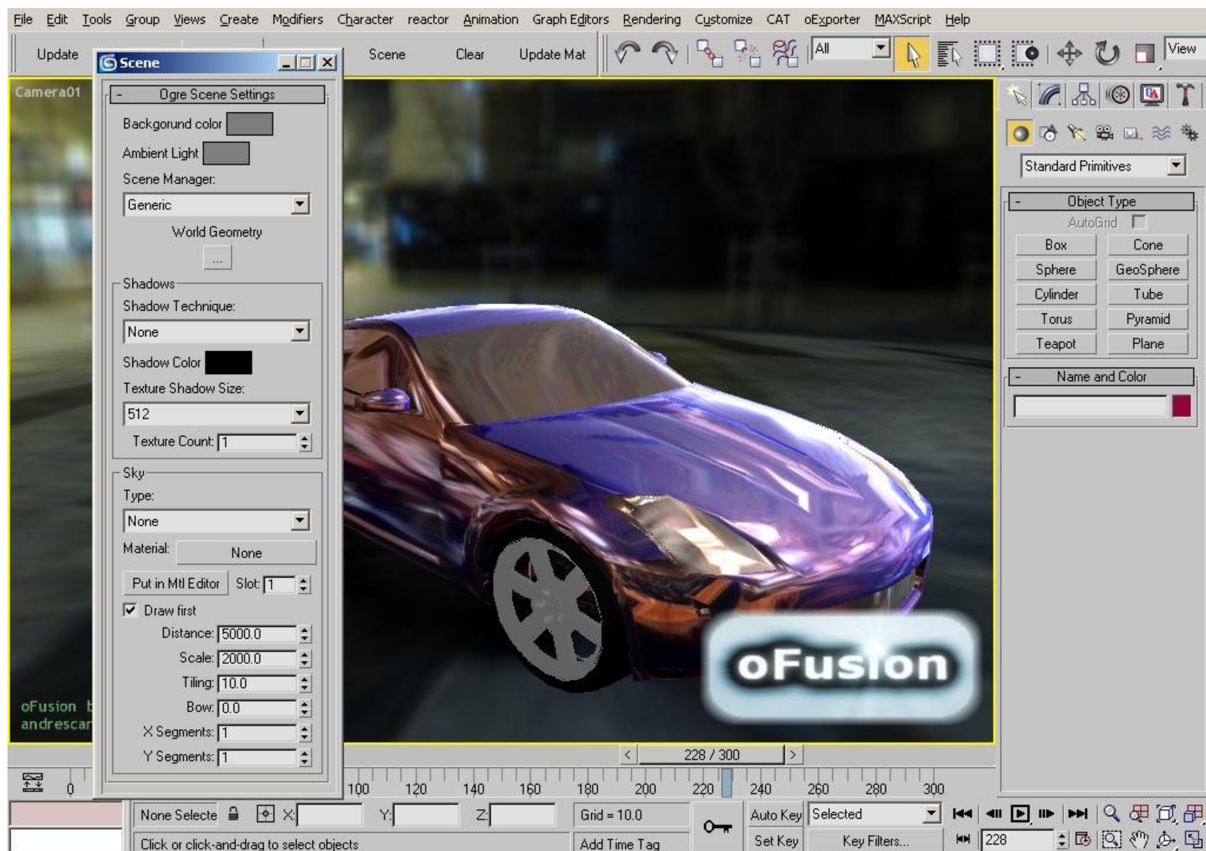
8.3. OGRE

OGRE je biblioteka za iscrtavanje 3D scena. Podržava OpenGL i DirectX. Od operacijskih sustava podržava Windows, Linux i Mac OS X. Prepoznaje mnogobrojne formate slika (zahvaljujući ostalim bibliotekama otvorenog koda, kao npr. DevIL i SDL).

Temeljni dio OGRE-a je graf scene. Čvorovi grafa scene su objekti. Objekti su zapisani u XML datoteci čija sintaksa jako podsjeća na format OBJ koji se često koristi u 3D grafici. XML oblik se koristi prilikom modeliranja i prenošenja objekata, dok se za učitavanje u krajnji program koristi ekvivalentni binarni oblik (isporučuje se alat za pretvorbu XML-a u binarni oblik i obratno).
Uz svaki objekt je zapisano i koji materijal mu pripada. Taj materijal mogu biti osnovne komponente osvjetljenja, ali i programi za sjenčanje (HLSL, Cg, GLSL). Materijali podržavaju višeprolazno izračunavanje scene.

Ostale mogućnosti neće biti nabrojene, jer bi to zahtijevalo još nekoliko stranica teksta.

OGRE dolazi s mnogo primjera, kao i predloškom programskog koda. Prednost predloška je ta da se pomoću njega može napraviti jednostavna scena sa dodanih svega nekoliko linija koda. Nedostatak je taj da se za korištenje dobivenog programa mora koristiti OGRE struktura direktorija, jer su neke putanje fiksno određene.



Slika 33: oFusion dodatak za 3D Studio Max koji u pozadini koristi OGRE.

<u>Primjer kocke zapisane u XML formatu:</u>

```
<mesh>
    <submeshes>
          <submesh material="Cube" usesharedvertices="false"
           use32bitindexes="false" operationtype="triangle_list">
                <faces count="12">
                      <face v1="0" v2="1" v3="2"/>
                      <face v1="2" v2="3" v3="0"/>
                      <face v1="4" v2="5" v3="7"/>
                      <face v1="7" v2="5" v3="6"/>
                      <face v1="8" v2="9" v3="11"/>
                      <face v1="11" v2="9" v3="10"/>
                      <face v1="12" v2="13" v3="15"/>
                      <face v1="15" v2="13" v3="14"/>
                      <face v1="16" v2="17" v3="18"/>
                      <face v1="18" v2="19" v3="16"/>
                      <face v1="20" v2="21" v3="23"/>
                      <face v1="23" v2="21" v3="22"/>
                </faces>
                <geometry vertexcount="24">
                      <vertexbuffer positions="true" normals="true">
                   <vertex>
                       <position x="1.000000" y="1.000000" z="-1.000000"/>
```

```
    <normal x="0.000000" y="0.000000" z="-1.000000"/>
</vertex>
<vertex>
    <position x="1.000000" y="-1.000000" z="-1.000000"/>
    <normal x="0.000000" y="0.000000" z="-1.000000"/>
</vertex>
<vertex>
    <position x="-1.000000" y="-1.000000" z="-1.000000"/>
    <normal x="0.000000" y="0.000000" z="-1.000000"/>
</vertex>
<vertex>
    <position x="-1.000000" y="1.000000" z="-1.000000"/>
    <normal x="0.000000" y="0.000000" z="-1.000000"/>
</vertex>
<vertex>
    <position x="1.000000" y="0.999999" z="1.000000"/>
    <normal x="0.000000" y="0.000000" z="1.000000"/>
</vertex>
<vertex>
    <position x="-1.000000" y="1.000000" z="1.000000"/>
    <normal x="0.000000" y="0.000000" z="1.000000"/>
</vertex>
<vertex>
    <position x="-1.000000" y="-1.000000" z="1.000000"/>
    <normal x="0.000000" y="0.000000" z="1.000000"/>
</vertex>
<vertex>
    <position x="0.999999" y="-1.000001" z="1.000000"/>
    <normal x="0.000000" y="0.000000" z="1.000000"/>
</vertex>
<vertex>
    <position x="1.000000" y="1.000000" z="-1.000000"/>
    <normal x="1.000000" y="-0.000001" z="-0.000000"/>
</vertex>
<vertex>
    <position x="1.000000" y="0.999999" z="1.000000"/>
    <normal x="1.000000" y="-0.000001" z="-0.000000"/>
</vertex>
<vertex>
    <position x="0.999999" y="-1.000001" z="1.000000"/>
    <normal x="1.000000" y="-0.000001" z="-0.000000"/>
</vertex>
<vertex>
    <position x="1.000000" y="-1.000000" z="-1.000000"/>
    <normal x="1.000000" y="-0.000001" z="-0.000000"/>
</vertex>
<vertex>
    <position x="1.000000" y="-1.000000" z="-1.000000"/>
    <normal x="-0.000000" y="-1.000000" z="-0.000000"/>
</vertex>
<vertex>
    <position x="0.999999" y="-1.000001" z="1.000000"/>
    <normal x="-0.000000" y="-1.000000" z="-0.000000"/>
</vertex>
<vertex>
    <position x="-1.000000" y="-1.000000" z="1.000000"/>
    <normal x="-0.000000" y="-1.000000" z="-0.000000"/>
</vertex>
<vertex>
    <position x="-1.000000" y="-1.000000" z="-1.000000"/>
    <normal x="-0.000000" y="-1.000000" z="-0.000000"/>
</vertex>
```

```
                    <vertex>
                        <position x="-1.000000" y="-1.000000" z="-1.000000"/>
                        <normal x="-1.000000" y="0.000000" z="-0.000000"/>
                    </vertex>
                    <vertex>
                        <position x="-1.000000" y="-1.000000" z="1.000000"/>
                        <normal x="-1.000000" y="0.000000" z="-0.000000"/>
                    </vertex>
                    <vertex>
                        <position x="-1.000000" y="1.000000" z="1.000000"/>
                        <normal x="-1.000000" y="0.000000" z="-0.000000"/>
                    </vertex>
                    <vertex>
                        <position x="-1.000000" y="1.000000" z="-1.000000"/>
                        <normal x="-1.000000" y="0.000000" z="-0.000000"/>
                    </vertex>
                    <vertex>
                        <position x="1.000000" y="0.999999" z="1.000000"/>
                        <normal x="0.000000" y="1.000000" z="0.000000"/>
                    </vertex>
                    <vertex>
                        <position x="1.000000" y="1.000000" z="-1.000000"/>
                        <normal x="0.000000" y="1.000000" z="0.000000"/>
                    </vertex>
                    <vertex>
                        <position x="-1.000000" y="1.000000" z="-1.000000"/>
                        <normal x="0.000000" y="1.000000" z="0.000000"/>
                    </vertex>
                    <vertex>
                        <position x="-1.000000" y="1.000000" z="1.000000"/>
                        <normal x="0.000000" y="1.000000" z="0.000000"/>
                    </vertex>
                        </vertexbuffer>
                </geometry>
            </submesh>
    </submeshes>
</mesh>
```


Primjer pripadajućeg materijala:

```
material Cube
{
    receive_shadows on
    technique
    {
        pass
        {
            ambient 0.500000 0.500000 0.500000 1.000000
            diffuse 0.640000 0.640000 0.640000 1.000000
            specular 0.500000 0.500000 0.500000 1.000000 12.500000
            emissive 0.000000 0.000000 0.000000 1.000000
        }
    }
}
```

Gore navedeni primjer kocke izvezen je iz Blender-a pomoću jedne od skripti koje dolaze uz OGRE.

Taj primjer se iz XML formata (obično ima nastavak mesh.xml) može pretvoriti u binarni oblik (obično ima nastavak mesh) pomoću programa OgreXMLConverter.

`OgreXMLConverter` *`ime`*`.mesh.xml` – pretvara iz XML oblika u binarni (*ime*.mesh)
`OgreXMLConverter` *`ime`*`.mesh` – pretvara iz binarnog oblika u XML (*ime*.mesh.xml)

Predložak programskog koda:

```
#include "ExampleApplication.h"

// Declare a subclass of the ExampleFrameListener class
class MyListener : public ExampleFrameListener
{
public:
  MyListener(RenderWindow* win, Camera* cam) : ExampleFrameListener(win,
             cam)
  {}

  bool frameStarted(const FrameEvent& evt)
  {
    return ExampleFrameListener::frameStarted(evt);
  }

  bool frameEnded(const FrameEvent& evt)
  {
    return ExampleFrameListener::frameEnded(evt);
  }
};

// Declare a subclass of the ExampleApplication class
class SampleApp : public ExampleApplication
{
public:
  SampleApp()
  {}

protected:
  // Define what is in the scene
  void createScene(void)
  {
  }

  // Create new frame listener
  void createFrameListener(void)
  {
    mFrameListener = new MyListener(mWindow, mCamera);
```

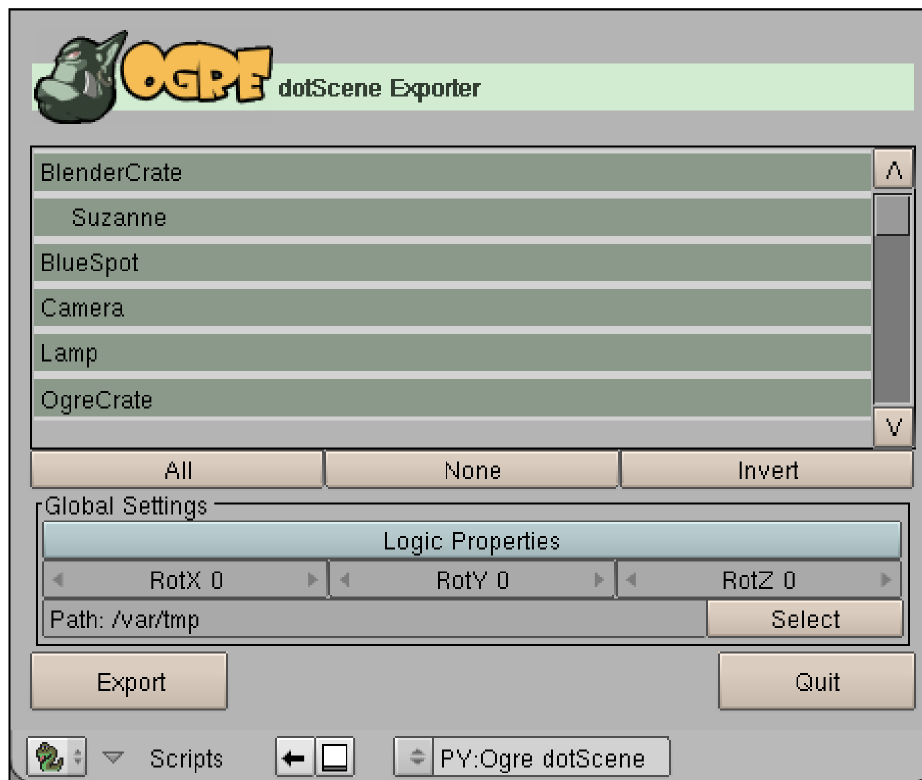
```
    mRoot->addFrameListener(mFrameListener);
  }
};

#ifdef __cplusplus
extern "C"
{
#endif

#if OGRE_PLATFORM == OGRE_PLATFORM_WIN32
#define WIN32_LEAN_AND_MEAN
#include "windows.h"
  INT WINAPI WinMain(HINSTANCE hInst, HINSTANCE, LPSTR strCmdLine, INT)
#else
  int main(int argc, char **argv)
#endif
  {
    // Instantiate our subclass
    SampleApp myApp;

    try
    {
      // ExampleApplication provides a go method, which starts the
     // rendering.
      myApp.go();
    }
    catch (Ogre::Exception& e)
    {
#if OGRE_PLATFORM == OGRE_PLATFORM_WIN32
       MessageBoxA(NULL, e.getFullDescription().c_str(), "An exception
               has occured!", MB_OK | MB_ICONERROR | MB_TASKMODAL);
#else
      std::cerr << "Exception:\n";
      std::cerr << e.getFullDescription().c_str() << "\n";
#endif
      return 1;
    }

    return 0;
  }

#ifdef __cplusplus
}
#endif
```

U funkciji SampleApp::createScene() se učitavaju objekti u scenu. Klasa SampleApp naslijeđuje klasu ExampleApplication koja obavlja sva početna podešavanja. Programeru ostaje jedino da učita objekte u scenu i napiše kod za interakciju s korisnikom.

Dodaci Blender-u koji dolaze uz OGRE su:

- BlenderImport - učitavanje objekta iz mesh.xml formata
- OGREExport - spremanje objekta u mesh.xml format
- dotScene Exporter - spremanje cijele scene

Slika 34: Sučelje OGREExport-a.



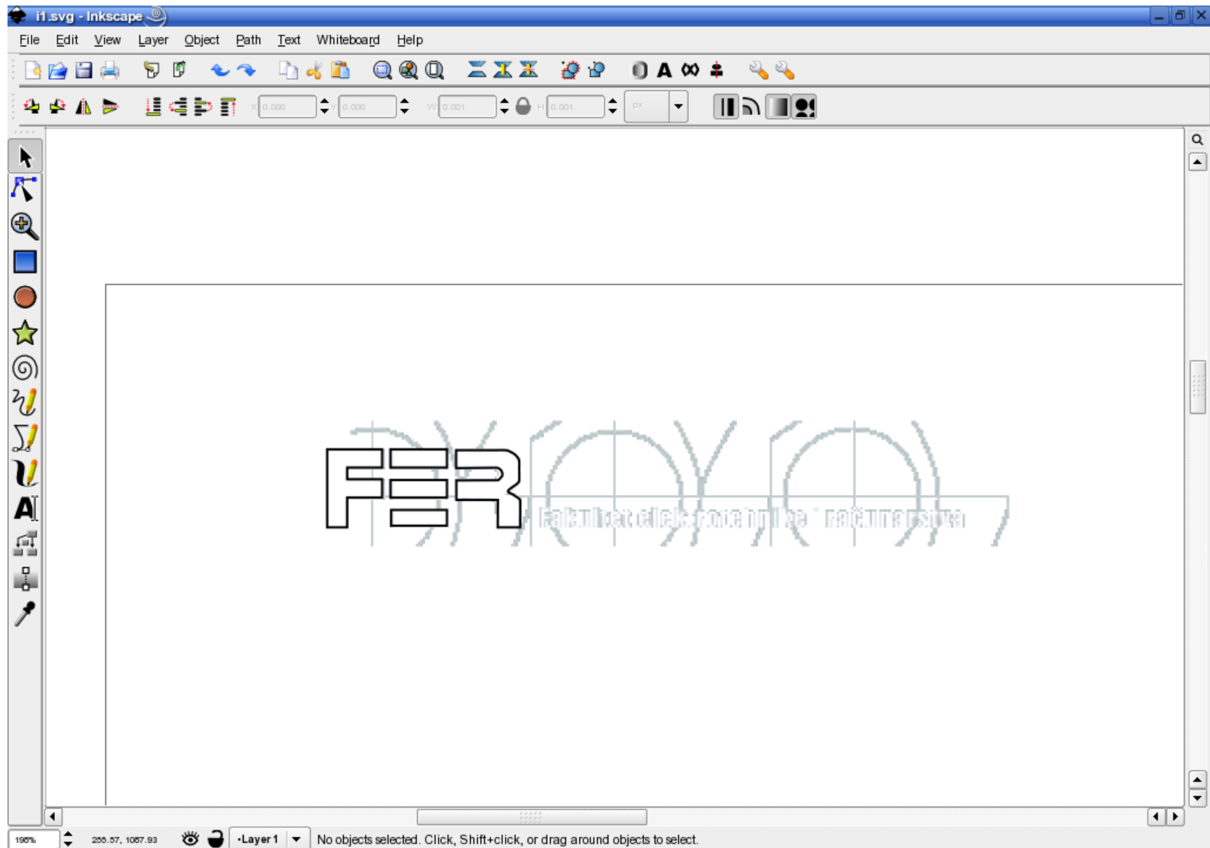
Slika 35: Sučelje dotScene Exporter-a.

8.4. Postupak izrade programskog rješenja

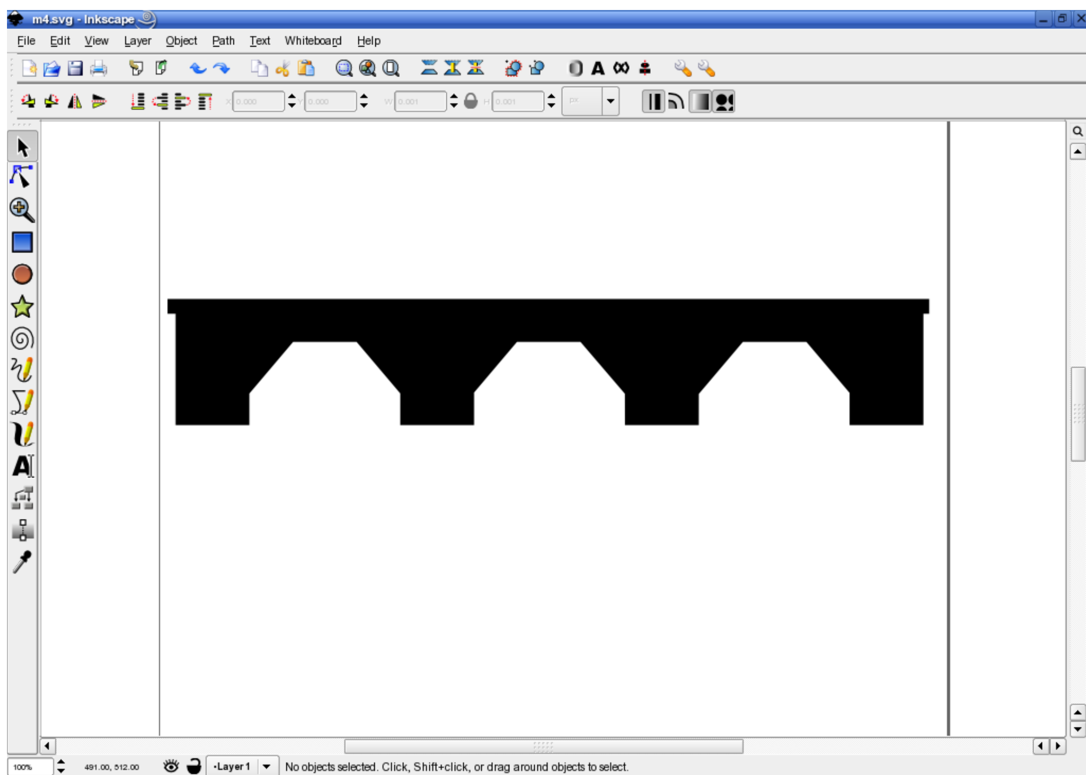
Sa adrese http://www.fer.hr preuzet je logo Fakulteta elektrotehnike i računarstva. Taj logo je ustvari slika spremljena u rasterskom formatu. Slika je zatim učitana u Inkscape (prvi sloj), gdje je korištena kao predložak za izradu loga u vektorskom obliku (drugi sloj). Nakon dovršetka vektorskog oblika, prvi sloj je izbrisan, tako da je ostao samo drugi sloj se logom u vektorskom obliku. Obje datoteke (sa i bez slike) su priložene na CD-u.

Logo u vektorskom obliku je učitan u Blender i pretvoren u niz poligona. Poligoni su zatim izvučeni (engl. extrude) tako da se dobije 3D objekt. Taj objekt je zatim sačuvan pomoću skripte ogreExport u mesh.xml format. Pripadni materijal je odbačen i napravljen novi koji koristi programe za sjenčanje. Iz demonstrativnih razloga, sve datoteke su priložene na CD-u.

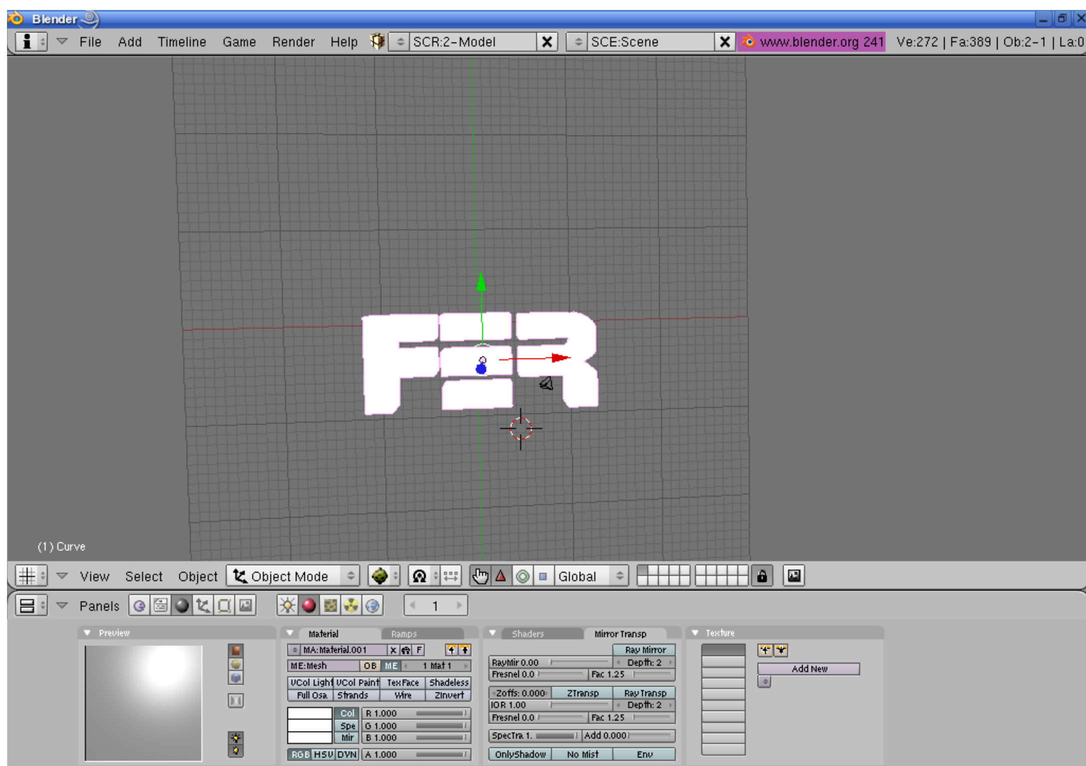
Isti postupak je ponovljen za model mosta, samo što za most nije korištena slika kao predložak, nego je oblik improviziran.



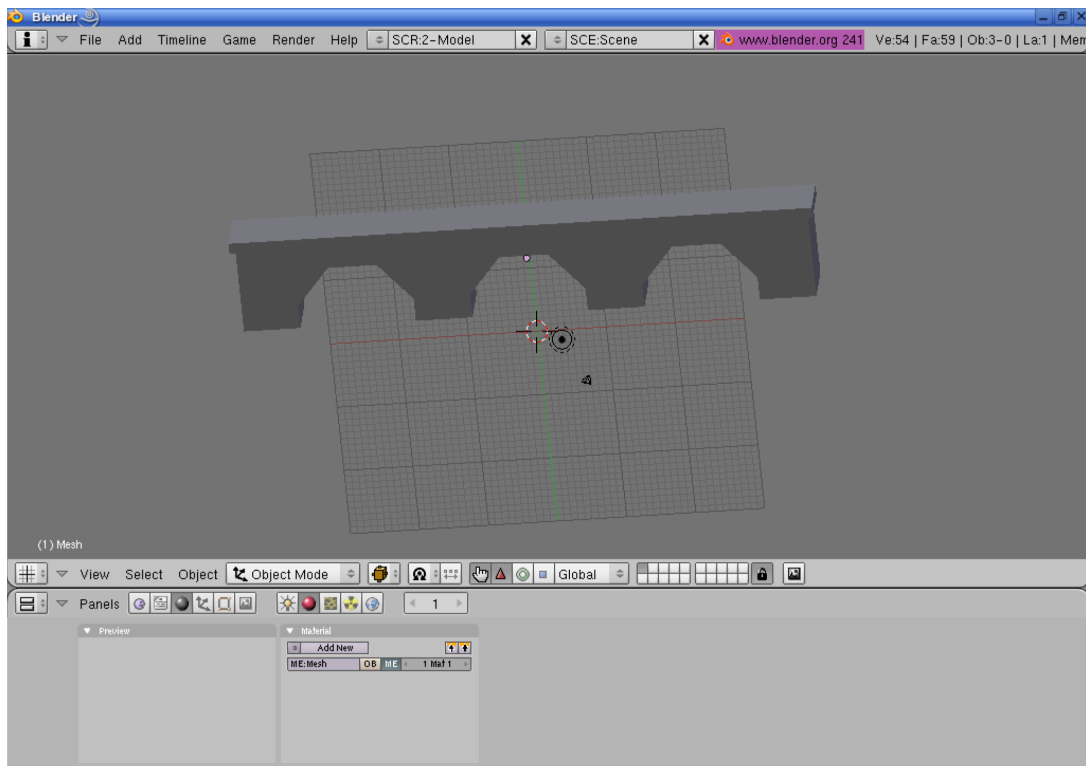
Slika 36: Logo FER-a u vektorskom obliku sa slikom kao predloškom.



Slika 37: Most u vektorskom obliku.



Slika 38: Logo FER-a pretvoren u 3D objekt.



Slika 39: Most pretvoren u 3D objekt.

<u>Materijal korišten za FER-ov logo u Blenderu izvezen u format koji podržava OGRE:</u>

```
material FER
{
	receive_shadows on
	technique
	{
		pass
		{
			ambient 0.500000 0.500000 0.500000 1.000000
			diffuse 0.800000 0.800000 0.800000 1.000000
			specular 0.500000 0.500000 0.500000 1.000000 12.500000
			emissive 0.000000 0.000000 0.000000 1.000000
		}
	}
}
```

Novi materijal za FER-ov logo koji koristi programe za sjenčanje (FER.material):

```
vertex_program FER_vs cg
{
   source FER_vs.cg
   profiles arbvp1
   entry_point main
}

fragment_program FER_fs cg
{
   source FER_fs.cg
   profiles arbfp1
   entry_point main
}


material FER
{
    technique
    {
        pass
        {
            vertex_program_ref FER_vs
            {
                param_named_auto worldViewProj worldviewproj_matrix
            }

            fragment_program_ref FER_fs
            {
            }
        }
    }
}
```

FER_vs.cg

```
float4 main (float4 polozaj : POSITION,
             uniform float4x4 worldViewProj) :POSITION
{
      return mul(worldViewProj, polozaj);
}
```

FER_fs.cg

```
float4 main(void) : COLOR
{
      return float4(1.0f, 1.0f, 1.0f, 1.0f);
}
```

Dodatak funkciji SampleApp::createScene():

```
Entity *ent = mSceneMgr->createEntity( "FER", "FER.mesh" );
SceneNode *node = mSceneMgr->getRootSceneNode()->
                  createChildSceneNode( "FERNode" );
node->pitch( Degree( -90 ) );
node->attachObject( ent );
```

Materijal korišten za most (Most.material):

```
vertex_program Most_vs glsl
{
    source Most.vert
}

fragment_program Most_fs glsl
{
    source Most.frag
}

material Most
{
    technique
    {
        pass
        {
            vertex_program_ref Most_vs
            {
            }

            fragment_program_ref Most_fs
            {
            }
        }
    }
}
```

Most.vert

```
void main(void)
{
      gl_Position = ftransform();
}
```

Most.frag

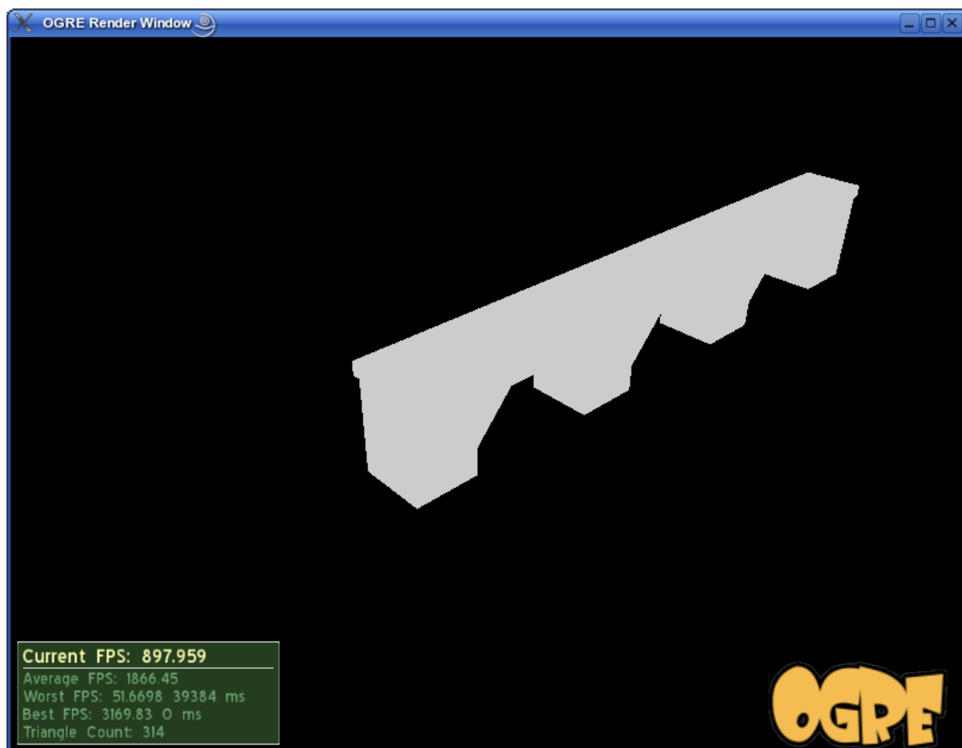
```
void main(void)
{
      gl_FragColor = vec4(0.8, 0.8, 0.8, 1.0);
}
```

Dodatak funkciji SampleApp::createScene():

```
Entity *ent = mSceneMgr->createEntity( "Most", "Most.mesh" );
SceneNode *node = mSceneMgr->getRootSceneNode()->
                  createChildSceneNode( "MostNode");
node->translate( Vector3( 0, 0, -500 ) );
node->pitch( Degree( -90 ) );
node->attachObject( ent );
```

Slika 40: Uvodni prozor OGRE-a za podešavanje iscrtavanja.

Slika 41: Scena s logom FER-a.



Slika 42: Scena s mostom.

Napomena: Namjena ovog programskog zadatka je da se na brz i jednostavan način pokaže kako se lako može izraditi 3D aplikacija uz pomoć gotovih programskih alata. Zbog toga su scene iscrtavane bez osvjetljenja, a objekti samo s difuznom komponentom boje, što za posljedicu ima nepotpun dojam trodimenzionalnosti.

Kako je korišten predložak programskog koda, program i materijali se moraju smjestiti na točno određenu lokaciju unutar OGRE strukture direktorija. Recimo da je OGRE raspakiran u direktorij OGRE_DIR bilo gdje na disku računala.

Izvršni programi idu u direktorij: OGRE_DIR/Samples/Common/bin
Materijali idu u direktorij: OGRE_DIR/Samples/Media/materials/scripts
Programi za sjenčanje idu u direktorij: OGRE_DIR/Samples/Media/materials/programs

9. Zaključak

U ovom radu izložen je pregled programirljivog grafičkog sklopovlja, kao i jezika niske i visoke razine.

Opisano je nekoliko gotovih programskih alata koji služe za razvoj i testiranje programa koji se izvode na grafičkom sklopovlju.

Dano je nekoliko primjera programa za sjenčanje koji se izvršavaju na grafičkom sklopovlju, kao i nekoliko cjelovitih programskih primjera (od postavljanja OpenGL-a, do programa za sjenčanje).

Kako grafičko sklopovlje napreduje, ima sve više mogućnosti i sve ga je teže programirati koristeći samo OpenGL i/ili DirectX funkcije. Mnogo lakše je pisati programe u jeziku visoke razine koji se onda izvršava na grafičkom procesoru. Krajnji rezultat je isti (jer se koristi isto sklopovlje), ali je postupak programiranja puno lakši.

Nekoliko godina nakon nastanka programirljivog grafičkog sklopovlja, njegove programirljive mogućnosti uglavnom su se upotrebljavale u računalnim igrama i profesionalim alatima za modeliranje. Te mogućnosti se u najnovije vrijeme koriste i za iscrtavanje radne površine operacijskih sustava, kao i u CAD sustavima. Sve češća primjena se pronalazi i u poslovima koji nisu vezani isključivo za računalnu grafiku (fizikalne simulacije, modeliranje sklopova u mikroelektronici itd.).

Buduća primjena programirljivog sklopovlja može se prepustiti samo mašti. Kako je programirljivo grafičko sklopovlje prilagođeno za rad s vektorima i matricama i ima puno cjevovoda koji rade paralelno, postiže do deset i više puta veći broj računskih operacija u sekundi od centralnog procesora.
Zbog toga je idealno za poslove kao što su šifriranje, dešifriranje, obrada audio signala za telekomunikacijske svrhe, izdvajanje značajki slike u svrhe raspoznavanja uzoraka itd.
Drugim riječima, programirljivo grafičko sklopovlje je idealno za poslove koji se daju paralelizirati, a koje inače slijedno izvršava centralni procesor računala ili grozdovi računala.
Jedna malo bolja programirljiva grafička kartica košta kao jedno prosječno računalo i ima 6 do 8 procesora vrhova, odnosno 16 do 24 procesora fragmenata.
Takva grafička kartica s 24 procesora fragmenata otprilike je ekvivalentna grozdu od 24 računala, samo što je krajnja cijena 24 puta manja ako se algoritam uspije prilagoditi za rad na grafičkoj kartici umjesto na grozdu.

Iz svega navedenog može se zaključiti da je prednost programirljivog grafičkog sklopovlja višestruka – uz manju konačnu cijenu možemo brže napraviti veći posao.

10. Dodatak A – postavljanje programa za sjenčanje u asemblerskom obliku

Prilikom programiranja u OpenGL-u uobičajeno se koristi biblioteka pomoćnih funkcija po imenu GLUT (Open**GL** **U**tility **L**ibrary). GLUT je besplatna biblioteka, ali nije otvorenog koda. Zbog toga je u primjerima korištena biblioteka funkcija po imenu SDL (**S**imple **D**irectmedia **L**ayer) koja je otvorenog koda.

OpenGL i SDL služe za početno postavljanje OpenGL-a, kao i ulaznih uređaja. Osim toga SDL može učitavati slike, ima podršku za zvuk, radi direktno s 2D grafikom i ima podršku za višedretveno programiranje. Očigledno je SDL bolji izbor. Zbog toga će SDL biti objašnjen u dodatku D.

U programskom primjeru neće biti ispitivano postojanje ekstenzija ARB_vertex_program i ARB_fragment_program. Funkcije koje pružaju te ekstenzije biti će direktno učitane. Nepostojanje funkcija indicira nepostojanje ekstenzije.

Program main.c:

```
#include <SDL/SDL.h>
#include <SDL/SDL_opengl.h>
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glext.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>


#if defined(WIN32)
#include <windows.h>
#endif


GLuint lista;           //identifikator liste za prikaz
GLuint vert;            //identifikator programa za sjenčanje vrhova
GLuint frag;            //identifikator programa za sjenčanje fragmenata
GLuint tex;             //identifikator teksture


//Deklaracija pokazivača na OpenGL funkcije. Funkcije su definirane u
//ekstenzijama ARB_vertex_program i ARB_fragment_program.
PFNGLGENPROGRAMSARBPROC glGenProgramsARB;
PFNGLBINDPROGRAMARBPROC glBindProgramARB;
PFNGLPROGRAMSTRINGARBPROC glProgramStringARB;
PFNGLDELETEPROGRAMSARBPROC glDeleteProgramsARB;


//Ova funkcija obavlja osnovna podešavanja OpenGL-a
void opengl_setup (void)
{
  SDL_Surface *surface;
  int mode;
```

```
//Pridruživanje adresa OpenGL funkcija pokazivačima
  glGenProgramsARB =
    (PFNGLGENPROGRAMSARBPROC) SDL_GL_GetProcAddress ("glGenProgramsARB");
  glBindProgramARB =
    (PFNGLBINDPROGRAMARBPROC) SDL_GL_GetProcAddress ("glBindProgramARB");
  glProgramStringARB =
    (PFNGLPROGRAMSTRINGARBPROC) SDL_GL_GetProcAddress ("glProgramStringARB");
  glDeleteProgramsARB =
    (PFNGLDELETEPROGRAMSARBPROC)
    SDL_GL_GetProcAddress ("glDeleteProgramsARB");


//Provjera da li su uspješno pronađene sve funkcije
  if (glGenProgramsARB == NULL || glBindProgramARB == NULL ||
      glProgramStringARB == NULL || glDeleteProgramsARB == NULL)
    {
      puts ("Nisu podrzane sve ekstenzije.\n");
      exit (EXIT_SUCCESS);
    }


//Učitavanje slike „Mail.bmp“
  surface = SDL_LoadBMP ("Mail.bmp");


//Provjera da li je slika uspješno učitana, odnosno da li ima ispravan format
  if (surface == NULL)
    {
      puts ("Ne mogu ucitati sliku.\n");
      exit (EXIT_FAILURE);
    }

  if (surface->format->BytesPerPixel == 3)
    {
      mode = GL_RGB;
    }
  else if (surface->format->BytesPerPixel == 4)
    {
      mode = GL_RGBA;
    }
  else
    {
      SDL_FreeSurface (surface);
      puts ("Nepoznat format teksture.\n");
      exit (EXIT_FAILURE);
    }


//Stvaranje objekta teksture, učitavanje slike u objekt i postavljanje filtera
  glGenTextures (1, &tex);
  glBindTexture (GL_TEXTURE_2D, tex);
  glTexImage2D (GL_TEXTURE_2D, 0, mode, surface->w, surface->h, 0, mode,
          GL_UNSIGNED_BYTE, surface->pixels);
  glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
  glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
  SDL_FreeSurface (surface);


  glViewport (0, 0, 640, 480);
```

```
  glMatrixMode (GL_PROJECTION);
  glLoadIdentity ();
  gluPerspective (60.0, 1.33, 0.1, 100.0);

  glMatrixMode (GL_MODELVIEW);
  glLoadIdentity ();

  glClearColor (0.0, 0.0, 0.0, 1.0);
  glClearDepth (1.0f);

  glDepthFunc (GL_LEQUAL);
  glEnable (GL_DEPTH_TEST);

  glHint (GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);
  glHint (GL_POLYGON_SMOOTH_HINT, GL_NICEST);
  glShadeModel (GL_SMOOTH);


//Stvaranje objekta liste za prikaz
  lista = glGenLists (1);


//Ovdje treba primjetiti da je slika zalijepljena za poligon tako da je okrenuta
//oko osi X i Y. To je zbog toga što BMP format sprema podatke na taj način.
  glNewList (lista, GL_COMPILE);
      glColor3f (1.0f, 0.75f, 0.5f);

      glBegin (GL_QUADS);
             glTexCoord2f (1.0f, 1.0f);
             glVertex3f (-1.0f, -1.0f, 0.0f);

             glTexCoord2f (0.0f, 1.0f);
             glVertex3f (1.0f, -1.0f, 0.0f);

             glTexCoord2f (0.0f, 0.0f);
             glVertex3f (1.0f, 1.0f, 0.0f);

             glTexCoord2f (1.0f, 0.0f);
             glVertex3f (-1.0f, 1.0f, 0.0f);
      glEnd ();
  glEndList ();


//Omogućavanje teksture
  glEnable (GL_TEXTURE_2D);
}


//Ova funkcija briše sve liste u trenutku zatvaranja programa
void cleanup (void)
{
  glDisable(GL_VERTEX_PROGRAM_ARB);
  glDisable(GL_FRAGMENT_PROGRAM_ARB);

  if (lista != 0)
    glDeleteLists (lista, 1);

  if (vert != 0)
    glDeleteProgramsARB (1, &vert);

  if (frag != 0)
    glDeleteProgramsARB (1, &frag);
```

```
  if (tex != 0)
    glDeleteTextures (1, &tex);

  SDL_Quit ();
}


//Ova funkcija služi za provjeru ulaza s tipkovnice (ESC za izlaz)
//i iscrtavanje scene
void draw_scene (void)
{
  SDL_Event event;

  while (1)
    {
      while (SDL_PollEvent (&event))
    {
      switch (event.type)
        {
        case SDL_KEYDOWN:
          switch (event.key.keysym.sym)
           {
           case SDLK_ESCAPE:
             exit (0);
           }
          break;
        case SDL_QUIT:
          exit (0);
        }
    }

      glClear (GL_DEPTH_BUFFER_BIT | GL_COLOR_BUFFER_BIT);
      glLoadIdentity ();

      gluLookAt (1.0, 1.0, -2.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);

      glCallList (lista);

      SDL_GL_SwapBuffers ();
    }
}


//Ova funkcija služi za učitavanje programa za sjenčanje
int load_file (const char *name, char **buf)
{
  FILE *fp;
  long len;

  if (*buf != NULL)
    free (*buf);

  fp = fopen (name, "rb");

  if (fp == NULL)
    return 1;


  fseek (fp, 0L, SEEK_END);
  len = ftell (fp);
```

```
  *buf = (char *) malloc (len + 1);

  if (*buf == NULL)
    {
      fclose (fp);
      return 1;
    }

  memset (*buf, 0, len + 1);

  fseek (fp, 0L, SEEK_SET);
  fread (*buf, len, 1, fp);
  fclose (fp);

  return 0;
}


//Ova funkcija služi za stvaranje objekata programa za sjenčanje, pridruživanje
//programskog koda objektima, te omogućivanje programa za sjenčanje
void load_programs (void)
{
  char *buf = NULL;
  int error;

  glGenProgramsARB (1, &vert);
  glGenProgramsARB (1, &frag);

  if (load_file ("prog.vp", &buf) != 0)
    {
      puts ("Ne mogu ucitati program za sjencanje vrhova.\n");
      exit (EXIT_FAILURE);
    }

  glBindProgramARB (GL_VERTEX_PROGRAM_ARB, vert);
  glProgramStringARB (GL_VERTEX_PROGRAM_ARB, GL_PROGRAM_FORMAT_ASCII_ARB,
                 strlen (buf), buf);
  glGetIntegerv (GL_PROGRAM_ERROR_POSITION_ARB, &error);

  if (error != -1)
    {
      puts ("Greska u programu za sjencanje vrhova.\n");
      free (buf);
      exit (EXIT_FAILURE);
    }

  if (load_file ("prog.fp", &buf) != 0)
    {
      puts ("Ne mogu ucitati program za sjencanje fragmenata.\n");
      exit (EXIT_FAILURE);
    }

  glBindProgramARB (GL_FRAGMENT_PROGRAM_ARB, frag);
  glProgramStringARB (GL_FRAGMENT_PROGRAM_ARB, GL_PROGRAM_FORMAT_ASCII_ARB,
                 strlen (buf), buf);
  glGetIntegerv (GL_PROGRAM_ERROR_POSITION_ARB, &error);

  if (error != -1)
    {
      puts ("Greska u programu za sjencanje fragmenata.\n");
      free (buf);
      exit (EXIT_FAILURE);
```

```
    }

  free (buf);

  glEnable (GL_VERTEX_PROGRAM_ARB);
  glEnable (GL_FRAGMENT_PROGRAM_ARB);
}


//Ovo je glavna funkcija u programu
int main (int argc, char *argv[])
{
  const SDL_VideoInfo *sdl;

  if (SDL_Init (SDL_INIT_VIDEO) < 0)
    {
      puts ("Ne mogu pokrenuti SDL.\n");
      exit (1);
    }

//Postavljanje funkcije koja će se pozvati u trenutku zatvaranja programa
  atexit (cleanup);

  sdl = SDL_GetVideoInfo ();

  if (sdl == NULL)
    {
      puts ("Ne mogu dohvatiti informacije o video sklopovlju.\n");
      exit (1);
    }

  SDL_GL_SetAttribute (SDL_GL_RED_SIZE, 8);
  SDL_GL_SetAttribute (SDL_GL_GREEN_SIZE, 8);
  SDL_GL_SetAttribute (SDL_GL_BLUE_SIZE, 8);
  SDL_GL_SetAttribute (SDL_GL_ALPHA_SIZE, 8);
  SDL_GL_SetAttribute (SDL_GL_DEPTH_SIZE, 24);
  SDL_GL_SetAttribute (SDL_GL_DOUBLEBUFFER, 1);

  if (SDL_SetVideoMode (640, 480, sdl->vfmt->BitsPerPixel, SDL_OPENGL) == 0)
    {
      puts ("Ne mogu postaviti video mod.\n");
      exit (1);
    }

  opengl_setup ();
  load_programs ();
  draw_scene ();

  return EXIT_SUCCESS;
}
```

Program prog.vp:

```
!!ARBvp1.0
PARAM mvp[4] = { state.matrix.mvp };

ATTRIB pos = vertex.position;
```

```
#Množenje ukupne transformacijske matrice sa ulaznim položajem vrha
#kako bi se dobio izlazni položaj vrha
DP4 result.position.x, mvp[0], pos;
DP4 result.position.y, mvp[1], pos;
DP4 result.position.z, mvp[2], pos;
DP4 result.position.w, mvp[3], pos;

#Ulazna boja vrha se samo proslijedi na izlaz
MOV result.color, vertex.color;

#Ulazne koordinate teksture se samo proslijede na izlaz.
#Treba primjetiti da nema množenja s teksturnom matricom. To je zbog toga
#što u glavnom programu nisu obavljene nikakve transformacije nad teksturom.
MOV result.texcoord[0], vertex.texcoord;

END
```

Program prog.fp:

```
!!ARBfp1.0
TEMP texColor;

#Pristup boji teksture na zadanim koordinatama
TEX texColor, fragment.texcoord[0], texture[0], 2D;

#Umnožak difuzne komponente boje poligona sa bojom teksture
MUL result.color, fragment.color, texColor;

END
```

Prevođenje programa: `gcc main.c -lGLU -lGL -lSDL -o program`

Objašnjenje parametara prevodioca gcc:

- -lGLU -povezivanje s bibliotekom GLU
- -lGL -povezivanje s bibliotekom GL
- -lSDL -povezivanje s bibliotekom SDL
- -o program -ime izlaznog programa

Program je preveden pod operacijskim sustavom Linux. Prevođenje pod operacijskim sustavom Windows je analogno. Jedina razlika je u imenu biblioteka.

Slika 43: Programski primjer koji koristi programe za sjenčanje u asemblerskom obliku.

Napomena: dodatak B i dodatak C koristit će istu scenu. Stoga u tim dodacima neće biti slike, nego samo programski kod.

11. Dodatak B – postavljanje programa za sjenčanje u programskom jeziku Cg

Prilikom korištenja programskog jezika Cg nije potrebno eksplicitno navoditi nikakve OpenGL ekstenzije. Cg ima svoje dinamičke biblioteke funkcija (dll pod Window-ima, so pod Linux-om) čije se funkcije koriste u glavnom programu. Biblioteke funkcija se mogu podijeliti na dva tipa: neovisan o grafičkom sučelju i ovisan o grafičkom sučelju (OpenGL, DirectX).

Neovisan dio koristi se za učitavanje programa za sjenčanje iz datoteka i podešavanja raznih parametara, dok se ovisni dio koristi uglavnom za sve ostalo (prevođenje i vezivanje programa, te povezivanje varijabli programa za sjenčanje s glavnim programom).

Iz priloženog koda može se vidjeti da Cg dinamička biblioteka funkcija puno olakšava rad. Automatski određuje najbolji profil programa za sjenčanje, učitava programe iz datoteka i još puno toga.

Jedini nedostatak je taj što programski jezik Cg ima malo ugrađenih varijabli i konstanti i što se varijable moraju eksplicitno vezivati uz ulazne i izlazne registre. Npr. programi za sjenčanje u asemblerskom obliku i oni napisani u GLSL-u imaju ugrađenu ukupnu transformacijsku matricu, dok se ona Cg-u mora eksplicitno navesti.

Program main.c:

```
#include <SDL/SDL.h>
#include <SDL/SDL_opengl.h>
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glext.h>
#include <Cg/cg.h>
#include <Cg/cgGL.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>


#if defined(WIN32)
#include <windows.h>
#endif


GLuint lista;
GLuint tex;
CGcontext context;
CGprogram vert;
CGprogram frag;
CGparameter MVPParam;
CGparameter texParam;
CGprofile VertProf;
CGprofile FragProf;
```

```
void opengl_setup (void)
{
  SDL_Surface *surface;
  int mode;

  context = cgCreateContext ();

  if (context == NULL)
    {
      puts ("Ne mogu napraviti Cg kontekst.\n");
      exit (EXIT_FAILURE);
    }

  VertProf = cgGLGetLatestProfile (CG_GL_VERTEX);
  if (VertProf == CG_PROFILE_UNKNOWN)
    {
      puts ("Nepoznat profil programa za sjencanje vrhova.\n");
      exit (EXIT_FAILURE);
    }
  cgGLSetOptimalOptions (VertProf);

  vert =
    cgCreateProgramFromFile (context, CG_SOURCE, "vert.cg", VertProf, "main",
                             NULL);

  if (vert == NULL)
    {
      puts ("Greška u programu za sjenčanje vrhova.\n");
      exit (EXIT_FAILURE);
    }

  cgGLLoadProgram (vert);
  MVPParam = cgGetNamedParameter (vert, "ModelViewProj");


  FragProf = cgGLGetLatestProfile (CG_GL_FRAGMENT);
  if (FragProf == CG_PROFILE_UNKNOWN)
    {
      puts ("Nepoznat profil programa za sjencanje fragmenata.\n");
      exit (EXIT_FAILURE);
    }
  cgGLSetOptimalOptions (FragProf);

  frag =
    cgCreateProgramFromFile (context, CG_SOURCE, "frag.cg", FragProf, "main",
                             NULL);

  if (frag == NULL)
    {
      puts ("Greška u programu za sjenčanje fragmenata.\n");
      exit (EXIT_FAILURE);
    }

  cgGLLoadProgram (frag);

  texParam = cgGetNamedParameter (frag, "tex");

  cgGLBindProgram (vert);
  cgGLEnableProfile (VertProf);
  cgGLBindProgram (frag);
```

```
cgGLEnableProfile (FragProf);
cgGLEnableTextureParameter (texParam);

surface = SDL_LoadBMP ("Mail.bmp");

if (surface == NULL)
  {
    puts ("Ne mogu ucitati sliku.\n");
    exit (EXIT_FAILURE);
  }

if (surface->format->BytesPerPixel == 3)
  {
    mode = GL_RGB;
  }
else if (surface->format->BytesPerPixel == 4)
  {
    mode = GL_RGBA;
  }
else
  {
    SDL_FreeSurface (surface);
    puts ("Nepoznat format teksture.\n");
    exit (EXIT_FAILURE);
  }

glGenTextures (1, &tex);
glBindTexture (GL_TEXTURE_2D, tex);
glTexImage2D (GL_TEXTURE_2D, 0, mode, surface->w, surface->h, 0, mode,
        GL_UNSIGNED_BYTE, surface->pixels);
glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
SDL_FreeSurface (surface);

glViewport (0, 0, 640, 480);

glMatrixMode (GL_PROJECTION);
glLoadIdentity ();
gluPerspective (60.0, 1.33, 0.1, 100.0);

glMatrixMode (GL_MODELVIEW);
glLoadIdentity ();

glClearColor (0.0, 0.0, 0.0, 1.0);
glClearDepth (1.0f);

glDepthFunc (GL_LEQUAL);
glEnable (GL_DEPTH_TEST);

glHint (GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);
glHint (GL_POLYGON_SMOOTH_HINT, GL_NICEST);
glShadeModel (GL_SMOOTH);

lista = glGenLists (1);

glNewList (lista, GL_COMPILE);
glColor3f (1.0f, 0.75f, 0.5f);
glBegin (GL_QUADS);
glTexCoord2f (1.0f, 1.0f);
glVertex3f (-1.0f, -1.0f, 0.0f);

glTexCoord2f (0.0f, 1.0f);
```

```
  glVertex3f (1.0f, -1.0f, 0.0f);

  glTexCoord2f (0.0f, 0.0f);
  glVertex3f (1.0f, 1.0f, 0.0f);

  glTexCoord2f (1.0f, 0.0f);
  glVertex3f (-1.0f, 1.0f, 0.0f);
  glEnd ();
  glEndList ();

  glEnable (GL_TEXTURE_2D);
}


void cleanup (void)
{
  if (lista != 0)
    glDeleteLists (lista, 1);

  if (vert != 0)
    cgDestroyProgram (vert);

  if (frag != 0)
    cgDestroyProgram (frag);

  if (tex != 0)
    glDeleteTextures (1, &tex);

  if (context != 0)
    cgDestroyContext (context);

  SDL_Quit ();
}


void draw_scene (void)
{
  SDL_Event event;

  while (1)
    {
      while (SDL_PollEvent (&event))
    {
      switch (event.type)
        {
        case SDL_KEYDOWN:
          switch (event.key.keysym.sym)
           {
           case SDLK_ESCAPE:
             exit (0);
           }
          break;
        case SDL_QUIT:
          exit (0);
        }
    }

      glClear (GL_DEPTH_BUFFER_BIT | GL_COLOR_BUFFER_BIT);
      glLoadIdentity ();

      gluLookAt (1.0, 1.0, -2.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
```

```
      cgGLSetStateMatrixParameter (MVPParam,
                          CG_GL_MODELVIEW_PROJECTION_MATRIX,
                          CG_GL_MATRIX_IDENTITY);
      cgGLSetTextureParameter (texParam, tex);



      glCallList (lista);

      SDL_GL_SwapBuffers ();
    }
}


int main (int argc, char *argv[])
{
  const SDL_VideoInfo *sdl;

  if (SDL_Init(SDL_INIT_VIDEO) < 0)
    {
      puts ("Ne mogu pokrenuti SDL.\n");
      exit (1);
    }

  atexit (cleanup);

  sdl = SDL_GetVideoInfo ();

  if (sdl == NULL)
    {
      puts ("Ne mogu dohvatiti informacije o video sklopovlju.\n");
      exit (1);
    }

  SDL_GL_SetAttribute (SDL_GL_RED_SIZE, 8);
  SDL_GL_SetAttribute (SDL_GL_GREEN_SIZE, 8);
  SDL_GL_SetAttribute (SDL_GL_BLUE_SIZE, 8);
  SDL_GL_SetAttribute (SDL_GL_ALPHA_SIZE, 8);
  SDL_GL_SetAttribute (SDL_GL_DEPTH_SIZE, 24);
  SDL_GL_SetAttribute (SDL_GL_DOUBLEBUFFER, 1);

  if (SDL_SetVideoMode (640, 480, sdl->vfmt->BitsPerPixel, SDL_OPENGL) == 0)
    {
      puts ("Ne mogu postaviti video mod.\n");
      exit (1);
    }

  opengl_setup ();
  draw_scene ();

  return EXIT_SUCCESS;
}
```

Program vert.cg:

```
struct vertex
{
    float3 position     : POSITION;
           float4 color    : COLOR0;
```

```
    float2 texcoord : TEXCOORD0;
};

struct fragment
{
    float4 position     : POSITION;
    float4 color    : COLOR0;
    float2 texcoord : TEXCOORD0;
};

fragment main(vertex IN, uniform float4x4 ModelViewProj)
{
    fragment OUT;

    float4 v = float4(IN.position.xyz, 1.0f );

    OUT.position = mul(ModelViewProj, v);
    OUT.color    = IN.color;
    OUT.texcoord = IN.texcoord;

    return OUT;
}
```

Program frag.cg:

```
struct fragment
{
    float4 position     : POSITION;
    float4 color    : COLOR0;
    float2 texcoord : TEXCOORD0;
};


float4 main(fragment IN, uniform sampler2D tex) : COLOR
{
    return tex2D(tex, IN.texcoord) * IN.color;
}
```

Prevođenje programa: `gcc main.c -lGLU -lGL -lSDL -lCg -lCgGL -o program`

12. Dodatak C – postavljanje programa za sjenčanje u programskom jeziku GLSL

Slijedeće ekstenzije definiraju podršku za GLSL:

- GL_ARB_shading_language_100
- GL_ARB_shader_object
- GL_ARB_vertex_shader
- GL_ARB_fragment_shader

Za učitavanje ekstenzija korišten je pristup iz dodatka A (učitavaju se samo funkcije bez provjere postojanja ekstenzija).

Program main.c:

```
#include <SDL/SDL.h>
#include <SDL/SDL_opengl.h>
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glext.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>


#if defined(WIN32)
#include <windows.h>
#endif


GLuint lista;
GLuint tex;
GLhandleARB prog;
GLhandleARB vert;
GLhandleARB frag;


PFNGLCREATESHADEROBJECTARBPROC glCreateShaderObjectARB;
PFNGLSHADERSOURCEARBPROC glShaderSourceARB;
PFNGLCOMPILESHADERARBPROC glCompileShaderARB;
PFNGLCREATEPROGRAMOBJECTARBPROC glCreateProgramObjectARB;
PFNGLATTACHOBJECTARBPROC glAttachObjectARB;
PFNGLLINKPROGRAMARBPROC glLinkProgramARB;
PFNGLUSEPROGRAMOBJECTARBPROC glUseProgramObjectARB;
PFNGLDETACHOBJECTARBPROC glDetachObjectARB;
PFNGLDELETEOBJECTARBPROC glDeleteObjectARB;
PFNGLGETOBJECTPARAMETERIVARBPROC glGetObjectParameterivARB;


void opengl_setup (void)
{
  SDL_Surface *surface;
  int mode;
```

```
glCreateShaderObjectARB =
  (PFNGLCREATESHADEROBJECTARBPROC)
  SDL_GL_GetProcAddress ("glCreateShaderObjectARB");
glShaderSourceARB =
  (PFNGLSHADERSOURCEARBPROC) SDL_GL_GetProcAddress ("glShaderSourceARB");
glCompileShaderARB =
  (PFNGLCOMPILESHADERARBPROC) SDL_GL_GetProcAddress ("glCompileShaderARB");
glCreateProgramObjectARB =
  (PFNGLCREATEPROGRAMOBJECTARBPROC)
  SDL_GL_GetProcAddress ("glCreateProgramObjectARB");
glAttachObjectARB =
  (PFNGLATTACHOBJECTARBPROC) SDL_GL_GetProcAddress ("glAttachObjectARB");
glLinkProgramARB =
  (PFNGLLINKPROGRAMARBPROC) SDL_GL_GetProcAddress ("glLinkProgramARB");
glUseProgramObjectARB =
  (PFNGLUSEPROGRAMOBJECTARBPROC)
  SDL_GL_GetProcAddress ("glUseProgramObjectARB");
glDetachObjectARB =
  (PFNGLDETACHOBJECTARBPROC) SDL_GL_GetProcAddress ("glDetachObjectARB");
glDeleteObjectARB =
  (PFNGLDELETEOBJECTARBPROC) SDL_GL_GetProcAddress ("glDeleteObjectARB");
glGetObjectParameterivARB =
  (PFNGLGETOBJECTPARAMETERIVARBPROC)SDL_GL_GetProcAddress("glGetObjectParamete
   rivARB");

if (glCreateShaderObjectARB == NULL || glShaderSourceARB == NULL ||
    glCompileShaderARB == NULL || glCreateProgramObjectARB == NULL ||
    glAttachObjectARB == NULL || glLinkProgramARB == NULL ||
    glUseProgramObjectARB == NULL || glDetachObjectARB == NULL ||
    glDeleteObjectARB == NULL || glGetObjectParameterivARB == NULL)
  {
    puts ("Nisu podrzane sve ekstenzije.\n");
    exit (EXIT_SUCCESS);
  }

surface = SDL_LoadBMP ("Mail.bmp");

if (surface == NULL)
  {
    puts ("Ne mogu ucitati sliku.\n");
    exit (EXIT_FAILURE);
  }

if (surface->format->BytesPerPixel == 3)
  {
    mode = GL_RGB;
  }
else if (surface->format->BytesPerPixel == 4)
  {
    mode = GL_RGBA;
  }
else
  {
    SDL_FreeSurface (surface);
    puts ("Nepoznat format teksture.\n");
    exit (EXIT_FAILURE);
  }

glGenTextures (1, &tex);
glBindTexture (GL_TEXTURE_2D, tex);
glTexImage2D (GL_TEXTURE_2D, 0, mode, surface->w, surface->h, 0, mode,
```

```
          GL_UNSIGNED_BYTE, surface->pixels);
  glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
  glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
  SDL_FreeSurface (surface);

  glViewport (0, 0, 640, 480);

  glMatrixMode (GL_PROJECTION);
  glLoadIdentity ();
  gluPerspective (60.0, 1.33, 0.1, 100.0);

  glMatrixMode (GL_MODELVIEW);
  glLoadIdentity ();

  glClearColor (0.0, 0.0, 0.0, 1.0);
  glClearDepth (1.0f);

  glDepthFunc (GL_LEQUAL);
  glEnable (GL_DEPTH_TEST);

  glHint (GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);
  glHint (GL_POLYGON_SMOOTH_HINT, GL_NICEST);
  glShadeModel (GL_SMOOTH);

  lista = glGenLists (1);

  glNewList (lista, GL_COMPILE);
  glColor3f (1.0f, 0.75f, 0.5f);
  glBegin (GL_QUADS);
  glTexCoord2f (1.0f, 1.0f);
  glVertex3f (-1.0f, -1.0f, 0.0f);

  glTexCoord2f (0.0f, 1.0f);
  glVertex3f (1.0f, -1.0f, 0.0f);

  glTexCoord2f (0.0f, 0.0f);
  glVertex3f (1.0f, 1.0f, 0.0f);

  glTexCoord2f (1.0f, 0.0f);
  glVertex3f (-1.0f, 1.0f, 0.0f);
  glEnd ();
  glEndList ();

  glEnable (GL_TEXTURE_2D);
}


void cleanup (void)
{
  glUseProgramObjectARB (0);

  if (lista != 0)
    glDeleteLists (lista, 1);

  if (vert != 0)
    {
      glDetachObjectARB (prog, vert);
      glDeleteObjectARB (vert);
    }


  if (frag != 0)
```

```
    {
      glDetachObjectARB (prog, frag);
      glDeleteObjectARB (frag);
    }

  if (tex != 0)
    glDeleteTextures (1, &tex);

  if (prog != 0)
    glDeleteObjectARB (prog);

  SDL_Quit ();
}


void draw_scene (void)
{
  SDL_Event event;

  while (1)
    {
      while (SDL_PollEvent (&event))
    {
      switch (event.type)
        {
        case SDL_KEYDOWN:
          switch (event.key.keysym.sym)
           {
           case SDLK_ESCAPE:
             exit (0);
           }
          break;
        case SDL_QUIT:
          exit (0);
        }
    }

      glClear (GL_DEPTH_BUFFER_BIT | GL_COLOR_BUFFER_BIT);
      glLoadIdentity ();

      gluLookAt (1.0, 1.0, -2.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);

      glCallList (lista);

      SDL_GL_SwapBuffers ();
    }
}


int load_file (const char *name, char **buf)
{
  FILE *fp;
  long len;

  if (*buf != NULL)
    free (*buf);

  fp = fopen (name, "rb");

  if (fp == NULL)
    return 1;
```

```
  fseek (fp, 0L, SEEK_END);
  len = ftell (fp);

  *buf = (char *) malloc (len + 1);

  if (*buf == NULL)
    {
      fclose (fp);
      return 1;
    }

  memset (*buf, 0, len + 1);

  fseek (fp, 0L, SEEK_SET);
  fread (*buf, len, 1, fp);
  fclose (fp);

  return 0;
}


void load_programs (void)
{
  char *buf = NULL;
  const char *tmp;
  int error;

  vert = glCreateShaderObjectARB (GL_VERTEX_SHADER_ARB);
  frag = glCreateShaderObjectARB (GL_FRAGMENT_SHADER_ARB);
  prog = glCreateProgramObjectARB ();

  if (load_file ("prog.vert", &buf) != 0)
    {
      puts ("Ne mogu ucitati program za sjencanje vrhova.\n");
      exit (EXIT_FAILURE);
    }

  tmp = (const char *) buf;
  glShaderSourceARB (vert, 1, &tmp, NULL);

  if (load_file ("prog.frag", &buf) != 0)
    {
      puts ("Ne mogu ucitati program za sjencanje fragmenata.\n");
      exit (EXIT_FAILURE);
    }

  tmp = (const char *) buf;
  glShaderSourceARB (frag, 1, &tmp, NULL);

  free (buf);

  glCompileShaderARB (vert);
  glGetObjectParameterivARB(vert, GL_OBJECT_COMPILE_STATUS_ARB, &error);

  if (error == 0)
    {
    puts("Greska u programu za sjencanje vrhova.\n");
    exit(EXIT_FAILURE);
    }

  glCompileShaderARB (frag);
```

```
  glGetObjectParameterivARB(frag, GL_OBJECT_COMPILE_STATUS_ARB, &error);

  if (error == 0)
    {
    puts("Greska u programu za sjencanje fragmenata.\n");
    exit(EXIT_FAILURE);
    }

  glAttachObjectARB (prog, vert);
  glAttachObjectARB (prog, frag);

  glLinkProgramARB (prog);
  glUseProgramObjectARB (prog);
}


int main (int argc, char *argv[])
{
  const SDL_VideoInfo *sdl;

  if (SDL_Init (SDL_INIT_VIDEO) < 0)
    {
      puts ("Ne mogu pokrenuti SDL.\n");
      exit (1);
    }

  atexit (cleanup);

  sdl = SDL_GetVideoInfo ();

  if (sdl == NULL)
    {
      puts ("Ne mogu dohvatiti informacije o video sklopovlju.\n");
      exit (1);
    }

  SDL_GL_SetAttribute (SDL_GL_RED_SIZE, 8);
  SDL_GL_SetAttribute (SDL_GL_GREEN_SIZE, 8);
  SDL_GL_SetAttribute (SDL_GL_BLUE_SIZE, 8);
  SDL_GL_SetAttribute (SDL_GL_ALPHA_SIZE, 8);
  SDL_GL_SetAttribute (SDL_GL_DEPTH_SIZE, 24);
  SDL_GL_SetAttribute (SDL_GL_DOUBLEBUFFER, 1);

  if (SDL_SetVideoMode (640, 480, sdl->vfmt->BitsPerPixel, SDL_OPENGL) == 0)
    {
      puts ("Ne mogu postaviti video mod.\n");
      exit (1);
    }

  opengl_setup ();
  load_programs ();
  draw_scene ();

  return EXIT_SUCCESS;
}
```

Program prog.vert:

```
void main()
{
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
    //gl_Position = ftransform(); //isto kao i gore

    gl_FrontColor = gl_Color;

    gl_TexCoord[0] = gl_MultiTexCoord0;
}
```

Program prog.frag:

```
uniform sampler2D tex;

void main()
{
    vec4 color = texture2D(tex, gl_TexCoord[0].st);
    gl_FragColor = gl_Color * color;
}
```

U glavnom programu korištena je tzv. „ARB notacija“ prilikom korištenja ekstenzija (svaka funkcija ima sufiks ARB).
Postoji i tzv. „OpenGL 2.0 notacija“ - funkcije nemaju sufiks ARB i imaju malo drugačije nazive.

Tablica 22: Korištene funkcije u ARB i OpenGL 2.0 notaciji

ARB notacija	OpenGL 2.0 notacija
glCreateShaderObjectARB	glCreateShader
glShaderSourceARB	glShaderSource
glCompileShaderARB	glCompileShader
glCreateProgramObjectARB	glCreateProgram
glAttachObjectARB	glAttachShader
glLinkProgramARB	glLinkProgram
glUseProgramObjectARB	glUseProgram
glDetachObjectARB	glDetachShader
glDeleteObjectARB	glDeleteShader i glDeleteProgram
glGetObjectParameterivARB	glGetProgram

Prevođenje programa: `gcc main.c -lGLU -lGL -lSDL -o program`

13. Dodatak D – SDL

SDL (**S**imple **D**irectmedia **L**ayer) već je kratko objašnjen u dodatku A. Tamo je korišten za početno postavljanje OpenGL-a, ali može i još puno više od toga.

SDL pruža pristup niske razine slijedećim uređajima:

- audio sklopovlju
- tipkovnici
- mišu
- upravljačkoj palici (engl. joystick)
- 3D sklopovlju (preko OpenGL-a)
- 2D spremniku okvira

Upotrebljava se za gledanje MPEG materijala, kao grafička podrška u nekoliko emulatora i računalnim igrama.

Postoji aktivna podrška za slijedeće operacijske sustave:

- Linux
- Windows
- Windows CE
- BeOS
- MacOS
- Mac OS X
- FreeBSD
- NetBSD
- OpenBSD
- DSD/OS
- Solaris
- IRIX
- QNX

Podržani su i slijedeći operacijski sustavi (iako ne službeno):

- AmigaOS
- Dreamcast
- Atari
- AIX
- OSF/Tru64
- RISC OS
- SymbianOS
- OS/2

SDL je napisan u programskom jeziku C, ali podržava i slijedeće:

- C++
- Ada
- C#

- Eiffel
- Erlang
- Euphoria
- Guile
- Haskell
- Java
- Lisp
- Lua
- ML
- Objective C
- Pascal
- Perl
- PHP
- Pike
- Pliant
- Python
- Ruby
- Smalltalk

Dodaci A, B i C sadrže identične odsječke koda u kojima se koristi SDL. Ti odsječci koda biti će ovdje objašnjeni jer se za inicijalno podešavanje OpenGL-a obično koristi GLUT, a SDL je potpuno drugačiji od GLUT-a.

13.1. Postavljanje uređaja za prikaz

Za korištenje SDL-a moraju se prvo uključiti zaglavne datoteke ***SDL.h*** i ***SDL_opengl.h***. SDL.h sadrži definicije konstanti i funkcija zajedničkih za sve podsustave SDL-a, dok SDL_opengl.h sadrži samo dijelove vezane uz OpenGL.
Uobičajeno je uključivanje i zaglavne datoteke ***stdlib.h*** radi funkcije ***atexit***. Ta funkcija se poziva u trenutku kada se gasi program.

```
#include <SDL/SDL.h>
#include <SDL/SDL_opengl.h>
#include <stdlib.h>
```

Nakon toga se u glavnom programu deklarira pokazivač na strukturu tipa ***SDL_VideoInfo*** koja će sadržavati podatke o grafičkom podsustavu računala.

```
const SDL_VideoInfo *sdl;
```

Slijedi inicijalizacija SDL-a funkcijom ***SDL_Init*** (u ovom slučaju samo grafičkog sustava).

```
SDL_Init(SDL_INIT_VIDEO);
```

Zatim se registrira funkcija koja se poziva prilikom gašenja programa. Registracija se obavlja funkcijom atexit kojoj se kao parametar predaje funkcija koja obavlja čišćenje OpenGL stanja i sve ostalo što se treba obaviti kada se program gasi. Ako se ništa osim SDL-a ne treba očistiti može se navesti kao parametar funkcija ***SDL_Quit***. Ta funkcija oslobađa sve sustave i sve ostalo što je SDL koristio.

```
atexit (cleanup);
```

Gore definiranom pokazivaču se pridružuje struktura koja sadrži podatke o grafičkom podsustavu računala pomoću funkcije ***SDL_GetVideoInfo***.

```
sdl = SDL_GetVideoInfo ();
```

Nakon toga treba definirati korištenje OpenGL spremnika pomoću funkcije ***SDL_GL_SetAttribute***.

```
//8 bita za svaku komponentu boje
SDL_GL_SetAttribute (SDL_GL_RED_SIZE, 8);
SDL_GL_SetAttribute (SDL_GL_GREEN_SIZE, 8);
SDL_GL_SetAttribute (SDL_GL_BLUE_SIZE, 8);
SDL_GL_SetAttribute (SDL_GL_ALPHA_SIZE, 8);

//24 bita za spremanje dubine
SDL_GL_SetAttribute (SDL_GL_DEPTH_SIZE, 24);

//koristi se dvostruki spremnik
SDL_GL_SetAttribute (SDL_GL_DOUBLEBUFFER, 1);
```

Na kraju preostaje samo postaviti željenu rezoluciju, broj bitova po pikselu i naznačiti da se za iscrtavanje treba koristiti OpenGL. To se obavlja pomoću funkcije ***SDL_SetVideoMode***.

```
SDL_SetVideoMode (640, 480, sdl->vfmt->BitsPerPixel,
                    SDL_OPENGL);
```

13.2. Učitavanje slika

SDL ima direktnu podršku za slike spremljene u formatu BMP. Podržava još mnogo formata slika, ali se onda prilikom prevođenja mora korisiti dodatna biblioteka funkcija po imenu ***SDL_image***. Radi jednostavnosti biti će objašnjen samo postupak učitavanja slika u formatu BMP.

Na početku se deklarira pokazivač na strukturu tipa ***SDL_Surface***.

```
SDL_Surface *surface;
```

Nakon toga se učita slika iz datoteke pomoću funkcije ***SDL_LoadBMP***.

```
surface = SDL_LoadBMP ("Mail.bmp");
```

Preko pokazivača se može doći do dimenzija slike (širina - **w**, visina - **h**) kao i do samih vrijednosti boja (**pixels**). Te informacije se dalje koriste u OpenGL funkcijama za rad s teksturama.

Na kraju se oslobodi memorija koju slika zauzima pomoću funkcije ***SDL_FreeSurface***.

```
SDL_FreeSurface (surface);
```

13.3. Reakcija na događaje i zamjena dvostrukog spremnika

Ova dva pojma su namjerno spomenuta zajedno zato što se koriste u funkciji za iscrtavanje scene.

Prvo se definira struktura tipa ***SDL_Event*** koja sadrži sve podatke o tome koji je događaj nastupio.

```
SDL_Event event;
```

Aktivni događaji pohranjuju se u navedenu strukturu pomoću funkcije ***SDL_PollEvent***.

```
SDL_PollEvent (&event);
```

Slijedi petlja koja ispituje događaje i obavlja određene radnje kao reakciju na te događaje, najčešće transformacije scene. Transformirana scena se tada iscrta.
Da bi se scena prikazana na zaslonu treba zamijeniti spremnik boje (koji je udvostručen da ne bi bilo titranja slike prilikom iscrtavanja). To se radi funkcijom SDL_GL_SwapBuffers ().

```
SDL_GL_SwapBuffers ();
```

13.4. Programski odsječak

```
#include <SDL/SDL.h>
#include <SDL/SDL_opengl.h>
#include <stdlib.h>
#include <stdio.h>

    ...
```

```
void opengl_setup (void)
{
  SDL_Surface *surface;
  int mode;

    ...

  surface = SDL_LoadBMP ("Mail.bmp");

  if (surface == NULL)
    {
      puts ("Ne mogu ucitati sliku.\n");
      exit (EXIT_FAILURE);
    }

  if (surface->format->BytesPerPixel == 3)
    {
      mode = GL_RGB;
    }
  else if (surface->format->BytesPerPixel == 4)
    {
      mode = GL_RGBA;
    }
  else
    {
      SDL_FreeSurface (surface);
      puts ("Nepoznat format teksture.\n");
      exit (EXIT_FAILURE);
    }

    ...

  SDL_FreeSurface (surface);

    ...

}


void cleanup (void)
{

    ...

  SDL_Quit ();
}


void draw_scene (void)
{
  SDL_Event event;

  while (1)
    {
      while (SDL_PollEvent (&event))
    {
      switch (event.type)
        {
        case SDL_KEYDOWN:
          switch (event.key.keysym.sym)
           {
           case SDLK_ESCAPE:
```

```
              exit (0);
            }
          break;
        case SDL_QUIT:
          exit (0);
        }
    }

    ...

     SDL_GL_SwapBuffers ();
    }
}


int main (int argc, char *argv[])
{
  const SDL_VideoInfo *sdl;

  if (SDL_Init(SDL_INIT_VIDEO) < 0)
    {
      puts ("Ne mogu pokrenuti SDL.\n");
      exit (1);
    }

  atexit (cleanup);

  sdl = SDL_GetVideoInfo ();

  if (sdl == NULL)
    {
      puts ("Ne mogu dohvatiti informacije o video sklopovlju.\n");
      exit (1);
    }

  SDL_GL_SetAttribute (SDL_GL_RED_SIZE, 8);
  SDL_GL_SetAttribute (SDL_GL_GREEN_SIZE, 8);
  SDL_GL_SetAttribute (SDL_GL_BLUE_SIZE, 8);
  SDL_GL_SetAttribute (SDL_GL_ALPHA_SIZE, 8);
  SDL_GL_SetAttribute (SDL_GL_DEPTH_SIZE, 24);
  SDL_GL_SetAttribute (SDL_GL_DOUBLEBUFFER, 1);

  if (SDL_SetVideoMode (640, 480, sdl->vfmt->BitsPerPixel, SDL_OPENGL) == 0)
    {
      puts ("Ne mogu postaviti video mod.\n");
      exit (1);
    }

  opengl_setup ();
  draw_scene ();

  return EXIT_SUCCESS;
}
```

14. Literatura

1. Fernando, Randima; Kilgard, Mark J.: „The Cg Tutorial - The Definitive Guide to Programmable Real-Time Graphics“, Addison-Wesley, Boston, travanj 2003.
2. Kessenich, John; Baldwin, Dave; Rost, Randi: „The OpenGL Shading Language“, revizija 1.10.59, 3Dlabs Inc. Ltd., 30. travanj 2004.
3. D'Angelo, Alex: „HELLO, Cg!“, s Interneta, http://developer.nvidia.com/object/hello_cg_tutorial.html

15. Sažetak

Tema ovog diplomskog rada je programirljivo grafičko sklopovlje. Iako programirljivo grafičko sklopovlje u profesionalnoj primjeni postoji već nekoliko desetljeća, tek je prije nekoliko godina postalo dostupno na osobnim računalima.

Sve do 2001. godine grafičke kartice na osobnim računalima koristile su sklopovski implementirane funkcije koje su služile za ubrzavanje grafičkih operacija. Tako izvedeno sklopovlje je ponekad dosta teško programirati, upravo zbog toga što ima funkcije koje se ne mogu mijenjati, nego im se samo mogu prosljeđivati parametri.

Od 2001. godine mnogo toga se promijenilo. Sada su sve grafičke kartice programirljive, odnosno imaju ugrađen grafički procesor (GPU) koji se može programirati isto kao i centralni procesor računala (CPU). Neki proizvođači više ne izvode sklopovski implementirane funkcije, nego te funkcije izvode kao kratke programe koji se izvode na grafičkom procesoru.

Na tržištu postoji mnogo programa koji olakšavaju izradu programa za grafičke procesore. Takvi programi imaju predloške gotovih programa, provjeravaju sintaksu programa, mjere performanse itd.
Takvi programi su obično besplatni, a mnogi su i otvorenog koda, što korisnicima omogućava da ih prepravljaju i poboljšavaju kako god im to odgovara.

Valja napomenuti kako grafički procesori ne moraju isključivo izvršavati programe koji su namijenjeni za računalnu grafiku, nego i sve ostale programe koji intenzivno koriste vektore i matrice.

Summary

Theme of this diploma thesis is programmable graphics hardware. Although programmable graphics hardware exists in professional graphics for decades, it became available for personal computers a few years ago.

Until year 2001 graphic cards on personal computers used hardware implemented functions which where used for acceleration of graphics operations. Hardware constructed in that way is sometimes hard to program, because it consists of functions which can not be changed – it is only possible to pass parameters to them.

Since year 2001 many things have changed. Now all graphic cards are programmable, that is - they have graphics processor (GPU) which can be programmed just as computer central processor (CPU). Some manufacturers do not implement hardware functions any more, they rather implement them as short programs which are being executed on graphics processor.

Plenty of programs exist on market that make it easy for their users to create programs for graphics processors. These programs have templates of final programs, they can check syntax, measure performance etc.
They are usually free, and some of them are even open source, which makes it possible for the users to modify and improve them in order to meet their needs.

It should be mentioned that graphics processors don't exclusively have to execute programs intended for computer graphics, but for all other programs which intensively use vectors and matrices.