KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS
PRAKTINĖS INFORMATIKOS KATEDRA

Paulius Liekis

REALAUS LAIKO VIZUALIZACINIAI METODAI DEMONSTRACINĖSE PROGRAMOSE

Magistro darbas

Vadovas
doc. dr. A. Lenkevičius

KAUNAS, 2005

KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS
PRAKTINĖS INFORMATIKOS KATEDRA

TVIRTINU
Katedros vedėjas
doc. dr. D. Rubliauskas
2005 05

REALAUS LAIKO VIZUALIZACINIAI METODAI DEMONSTRACINĖSE PROGRAMOSE

Informatikos mokslo magistro baigiamasis darbas

Kalbos konsultantė
Lietuvių kalbos katedros lektorė
dr. J. Mikelionienė
2005 05

Recenzentas
doc. dr. Vacius Jusas
2005 05

Vadovas
doc. dr. A. Lenkevičius
2005 05

Atliko
IFM 9/1 gr. stud.
P. Liekis
2005 05 10

KAUNAS, 2005

KVALIFIKACINĖ KOMISIJA

Pirmininkas:	Laimutis Telksnys, akademikas
Sekretorius:	Stasys Maciulevičius, docentas
Nariai:	Rimantas Barauskas, profesorius
	Raimundas Jasinevičius, profesorius
	Jonas Kazimieras Matickas, docentas
	Jonas Mockus, akademikas
	Rimantas Plėštys, docentas
	Henrikas Pranevičius, profesorius

SUMMARY

This paper introduces a concept of *demo* arising mainly among people who have common interest in computer graphics and multimedia. The paper shows strong relationship between realtime visualization methods and the subject of this work, so-called demo programs. These programs are the most attractive production produced by community of demo-makers and also the ones most related to conceptual, mathematic and algorithmic fields of computer graphics. Methods used in demo programs varies from simple like „phong“ illumination model, generation of objects from layers, cartoon rendering style or bump-mapping, to complex: realtime ray-tracing, rendering of correct reflections or soft shadows. This work presents six demo programs made by author together with „Nesnausk!“ demo-makers group, also analyses available methods of edge detection and visualization, and describes the method proposed by author, which has distributed calculations between central and video card processors. This method is used as a main visualization effect in „Zenit“ demo program.

TURINYS

Paveikslėlių sąrašas

Lentelių sąrašas

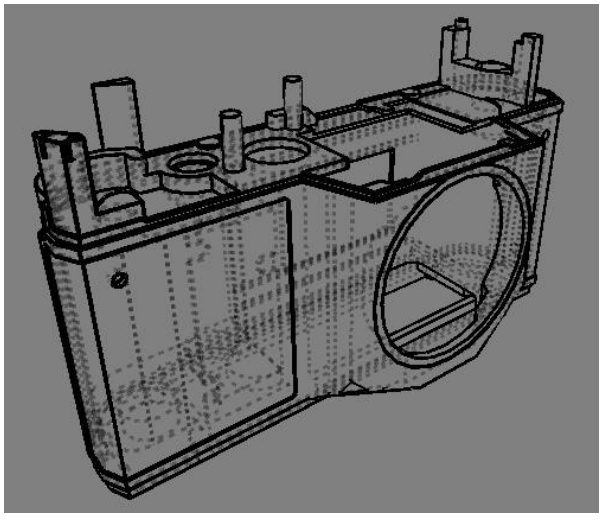
PRATARMĖ

Šis darbas supažindina su kompiuterinės grafikos „meno“ kryptimi – demonstracinių programų (angl.: *demo*) kūrimu, pagrindinai vyraujančiu tarp kompiuterinės grafikos ir multimedijos kūrėjų. Šiame darbe apžvelgiamas glaudus ryšys tarp realaus laiko vizualizacinių metodų ir demonstracinių programų. Šios programos yra vienos iš patraukliausių šios „meno“ srities (žarg.: *demoscenos*) produktų, ir taip pat labiausiai susijusios su kompiuterinės grafikos konceptualia, matematine ir algoritmine sritimis. Demonstracinėse programose naudojami įvairūs metodai, pradedant nuo paprastų, tokių kaip „phong“ apšvietimo modelis, objektų generavimas iš pjūvių, animacinio vaizdavimo stilius ar nelygaus paviršiaus simuliavimas, iki sudėtingų – realaus laiko spindulių trasavimas, taisyklingų atspindžių simuliavimas ar „minkštų“ šešėlių naudojimas. Šiame darbe apžvelgiamos šešios demonstracinės programos, kurių kūrime autorius dalyvavo, bei plačiau išanalizuoti kontūrų nustatymo bei vaizdavimo algoritmai ir aprašytas autoriaus pasiūlytas metodas, kurio skaičiavimai yra paskirstyti tarp centrinio ir vaizdo kortos procesorių. Šis metodas yra pagrindinis vizualizacinis metodas „Zenit“ demonstracinėje programoje.

ĮVADAS

Demonstracinė programa (arba tiesiog demonstracija) – tai kompiuterinė programa, kuri rodo kelių minučių ilgio kompiuterinės grafikos, garso, muzikos ir teksto mišinį. Demonstracinės programos yra šiuolaikinio kompiuterinio meno šaka. Jos supina matematiką, programavimą, grafiką, eksperimentinę muziką, dizainą ir kūrybiškumą į nepaprastą vaizdo ir garso derinį. Demonstracinės programos galutiniu rezultatu yra panašios į elektroninės muzikos vaizdo klipus, o pagrindinis skirtumas tarp jų yra tai, kad demonstracijose kiekvienas kadras yra generuojamas realiu laiku. Šios programos yra skirtos nustebinti žiūrovą, parodyti autorių kūrybišką vaizdo ir garso kūrimo kompiuteryje galimybių išnaudojimą. Kūrėjai dažniausiai yra jauni Europos programuotojai, dailininkai, dizaineriai, modeliuotojai ir elektroninės muzikos kūrėjai. Šios programos yra skirtos demonstracinių programų simpoziumams ir konkursams, kurie vyksta įvairiose Europos šalyse. Demonstracijos yra nekomerciniai produktai, juos galima parsisiųsti iš Interneto nemokamai. Viena iš demonstracijų rūšių yra pristatančioji programa, jai būdingas dydžio apribojimas. Dažniausiai pasitaikančios pristatančiųjų programų kategorijos: 64Kb, 16Kb, 4Kb.

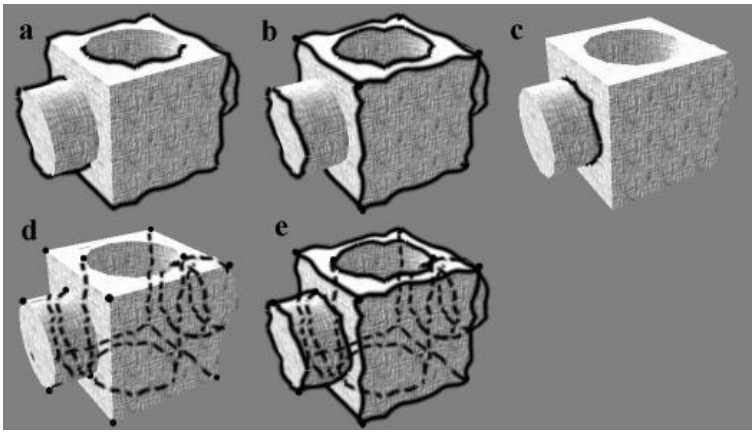
Autorius kartu su grupe „Nesnausk!“ sukūrė 6-ias demonstracines programas, kurios buvo pristatytos užsienio simpoziumuose. Vienoje iš paskutiniųjų autoriaus demonstracijų „Zenit“, kurios tikslas pavaizduoti „Zenit-E“ fotoaparato modelio išardymą ir surinkimą, buvo susidurta su problema - stilizuotų techninių vaizdų pateikimo specifika. Tokio tipo problemoms spręsti dažniai pasitelkiami nefotorealistiniai vaizdavimo metodai, šiuo atveju tam geriausiai tinka kontūrų vaizdavimas (1 pav.). Kontūrų vaizdavimas buvo pasirinktas šio tiriamojo darbo plėtotei.



1 pav. Kontūrų vaizdavimo stilius

Yra du galima grafinių vaizdų kūrimo metodai. Pirmas – tradicinis, pieštuku, anglimi it t. t. ir yra priklausomas nuo kūrėjo (menininko) gabumų. Antras – kompiuterinis vaizdų generavimo metodas. Tradiciniu būtu sukurti paveikslėliai turi meninę vertę, kurios nėra kompiuterio sugeneruotuose paveikslėliuose. Ranka pieštų vaizdų paprastumas turi keletą esminių privalumų. Menininkas gali supaprastinti paveikslėlį atmetant nereikalingas, dėmesį nukreipiančias detales ir nukreipti žiūrovo dėmesį į svarbiausias dalis. Taip pat jis gali pabrėžti tam tikrų dalių svarbumą keisdamas piešimo stilių, t. y. svarbesnes dalis nupiešdamas storomis, ryškiomis linijomis,o mažiau svarbias dalis šviesiomis, blyškiomis linijomis, t. y. gaunamas kontūrų vaizdas. Gautas paveikslas išlieka savaip realistiškas, bet skiriasi nuo fotorealistinio forma, spalva, apšvietimu ir šešėliais. Nepaisant šių skirtumų, kontūrų vaizdai dažnai sutinkami spaudoje (ypač komiksuose), filmuose ir mene.

Tobulėjant kompiuteriams bei programinei įrangai bandoma tradicinį kūrimo metodą imituoti kompiuterio pagalba. Viena iš pagrindinių šio metodo problemų yra teisingas vaizduojamo daikto kontūrų parinkimas. Sprendžiant kontūrų piešimo problemą kompiuterinėje grafikoje pirmiausia reikia nuspręsti kuriuos kontūrus norime atvaizduoti. Keletas galimų variantų pavaizduoti 2-ame paveikslėlyje. 2-o paveikslėlio a dalyje pavaizduoti išoriniai kontūrai, t. y. Tokie, kur objekto paviršius keičia matomumą: vienoje kontūro pusėje objekto paviršius iš žiūrovo pozicijos yra matomas, kitoje – nematomas. 2b ir 2c pav. vaizduoja „pažymėtus“ kontūrus. Šie kontūrai parenkami pagal tai, kaip stipriai objekto paviršius keičia kryptį. Jie yra dviejų tipų: iškilę (2b pav.) ir įdubę (2c pav.). Paveikslėlyje 2d pavaizduoti „paslėpti“ kontūrai, t. y. tokie, kuriuos užstoja objekto sienelės. Paveikslėlyje 2e pavaizduota visų šių kontūrų kombinacija.



2 pav. Kontūrų tipai: a – išoriniai, b – iškilę, c – įdubę, d - paslėptos linijos, e – visi kontūrai

Visi dabartiniai kontūrų vaizdavimo algoritmai gali būti suskirstyti į geometrinius ir paveikslėlių analizės metodus. Geometriniai nustato matomus kontūrus, juos paverčia į daugiakampius, kuriuos naudojant tekstūras atvaizduoja kaip tam tikrus piešimo stilius. Paveikslėlių analize pagrįsti metodai išanalizuoja galutinio paveikslėlio spalvų, gylio ir paviršiaus krypties informaciją tam, kad surastų taškus kurie priklauso briaunoms (pvz.: [5]). Šie abu metodai turi savų trūkumų. Greičio atžvilgiu: matomų kontūrų sąrašo sudarymas yra geometrinių, o kiekvieno taško apdorojimas yra paveikslėlių analizės algoritmų silpnoji vieta. Išgaunamo vaizdo atžvilgiu pirmieji negali pavaizduoti kontūrų tose vietose kuriose susiduria du daugiakampiai. Antrojo tipo algoritmai išsprendžia šią problemą, nes jiem nėra būtina, kad būtų geometrinės briaunos, tačiau jie negali atvaizduoti stilizuotų kontūrų (pvz.: brūkšninės ar kreivos linijos), o tik paryškinti taškus esančius ant kontūrų.

Siekiant stilizuotų kontūrų piešimo buvo pasirinkta geometrinių algoritmų grupė. Šiuos algoritmus galima skirstyti į keletą tipų pagal tai kaip jie paskirsto skaičiavimus tarp centrinio procesoriaus ir vaizdo plokštės. Dauguma senesnių algoritmų kontūrų sąrašą sudarinėdavo centriniame procesoriuje (pvz.: [8], [11]), o visą vaizdavimą perduodavo vaizdo plokštei. Šis duomenų perdavimas yra viena iš silpniausiųjų geometrinių algoritmų vietų. Dėl šios problemos ir dėl skaičiavimo vaizdo plokštėse didėjimo galimybių naujesniuose algoritmuose siūloma visus skaičiavimus perkelti į vaizdo plokštę [3], tačiau šiuolaikinės vaizdo plokštės yra orientuotos į lygiagrečius skaičiavimus, todėl tenka kai kuriuos skaičiavimus ir duomenis atkartoti kelis kartus. Šis skaičiavimų perkėlimas visiškai atlaisvina centrinį procesorių, tačiau stipriai apkrauna vaizdo plokštės atmintį ir procesorių.

Šių dviejų problemų suderinimui buvo nuspręsta sukurti geometrinį kontūrų vaizdavimo algoritmą, kuris skaičiavimus padalintų tarp centrinio procesoriaus ir vaizdo plokštės, bei apkrautų vaizdo plokštės atmintį mažiau, nei algoritmai visus skaičiavimus atliekantys vaizdo plokštėje.

Bendrojoje darbo dalyje pateikiama demonstracinių programų istorija, evoliucija ir dabartinės kryptys, taip pat aprašomos 6-ios demonstracinės programos, kurios buvo sukurtos šio darbo metu, ir jose realizuoti realaus laiko vizualizaciniai metodai. Taip pat aptariami ankstesni darbai kontūrų vaizdavimo realiuoju laiku srityje ir suformuluojamos kryptys ir reikalavimai naujai kuriamiems metodams ir algoritmams.

Tiriamojoje darbo dalyje aprašomi geometrinis realaus laiko kontūrų vaizdavimo metodas, jo realizacijos detalės, pateikiami rezultatai, aptarimas bei palyginimai su kitais egzistuojančiais metodais.

1. BENDROJI DALIS

1.1. Demonstracinės programos

Demonstracinių programų entuziastų bendruomenė, iškilusi apie 80-uosius, yra sudaryta daugiausia iš aukštųjų mokyklų ir universitetų studentų, kuriem būdingi bendri interesai kompiuterinėje grafikoje, muzikoje ir multimedijoje. Realaus laiko vizualizacinių efektų sąvoka buvo viena iš pagrindų nuo pat demonstracinių programų atsiradimo pradžios.

1.1.1. Istorija

Maždaug 1980 metais grupė jaunų aktyvių žmonių įsigijo savo pirmuosius kompiuterius. Kaip ir kiekvienas pradedantysis, pradėjo nuo žaidimų. Deja, kompiuteriniai žaidimai tuo metu buvo labai brangūs, taigi jaunieji entuziastai bandė surasti būdą gauti žaidimus nemokamai. Nukopijuoti žaidimą nebuvo labai sudėtinga, tačiau žaidimų kūrimo kompanijos įdiegdavo apsaugas nuo kopijavimo. Kai kurie iš šių žaidėjų nusprendė apeiti šias apsaugas ir sukūrė taip vadinamus „cracks“ – apsaugas panaikinančias programas. Tai, kas prasidėjo kaip paprastas pasilinksminimas, neužilgo peraugo į varžymąsi tarp skirtingų grupių, vadinamų „crackers“. Toks programinės įrangos įsigijimo būdas nebuvo legalus, todėl šių programų kūrėjai susikūrė pravardes, kad nuslėpti savo realią tapatybę.

Tikslas būdavo programą padaryti kiek įmanoma populiaresnę. Galbūt tai buvo viena iš priežasčių, kodėl būdavo ne tik sukuriama programa, bet ir įtraukiama tam tikra demonstracija – parodomos autorių pravardės ir šiek tiek grafikos ar muzikos.

Per keletą metų įvyko šiokie tokie pokyčiai - kai kurie entuziastai susidomėjo grafinių efektų kūrimu labiau nei pačiu nulaužiančių programų kūrimu. Jie atsiskyrė ir pradėjo kurti stulbinančius kompiuterinius efektus, kurie toli peržengė įprastas demonstracijas. Šis vaizdo ir garso derinys dabar ir žinomas kaip demonstracija ar demonstracinė programa. Pastaruoju metu tai tapo ypatinga ir unikali meno rūšis.

Laikui bėgant kito ir kompiuterių platformos, naudojamos demonstracinėms programoms. Pradžioje buvo naudojami Commodore 64, Sinclair/ZX-Spectrum, po to išpopuliarėjo Amiga, Atari ST ir galiausiai įsigalėjo asmeniniai kompiuteriai (PC). Dabar demonstracinių programų kūrėjai dirba visose platformose, įskaitant žaidimų konsoles (Playstation, X-box), delninius kompiuterius (PDA, GBA) ir mobiliuosius telefonus.

1.1.2. Grupės

Demonstracinės programos kūrimui reikalinga grupė žmonių – programuotojų, dailininkų, modeliuotojų, dizainerių ir muzikantų. Labai retai vienas žmogus gali gerai atlikti keletą iš šių darbų. Taigi, prie programos dirba grupė žmonių, paprastai pasivadinusi kažkokiu vardu.

Grupės nariai dirba kartu, kad sukurti galutinį produktą – demonstracinę programą. Dailininkai piešia paveiksliukus, modeliuoja, tekstūruoja ir animuoja 3D modelius; muzikantai kuria muziką; programuotojai turi visa tai priversti sklandžiai dirbti realiuoju laiku, realizuoti grafikos efektus. Aišku, kad pradedančioms grupėms gerą demonstracinę programą iš karto sukurti nepavyksta. Geros demonstracinės programos kūrimas gali pareikalauti keleto mėnesių įtempto visos grupės darbo.

Įdomus visos demonstracinių programų aspektas yra tai, kad jos yra visiškai nekomercinė – demonstracinės programos nėra parduodamos, už jas negaunami jokie pinigai. Atrodo, kad šių programų kūrėjus išlaiko vien tik šios meno rūšies žavesys, techniniai ir kūrybiniai aspektai, pramoga rungtyniauti tarpusavyje bei, žinoma, draugiškumas.

1.1.3. Simpoziumai ir konkursai

Kasmet vyksta kelios dešimtys demonstracinių programų simpoziumų, kur susirenka grupės iš viso pasaulio. Simpoziumuose kuriamos ar baiginėjamos programos bei vyksta konkursai – renkamos geriausios demonstracijos ir vyksta kitokios atrakcijos. Konkursai dalinami į atskiras kategorijas, atitinkančias demonstracijų rūšis.

1.1.3.1. Demonstracinės programos

Pagrindinė kategorija – šiems produktams netaikomi ypatingi apribojimai – demonstracija turi veikti su geru „šiuolaikiniu“ kompiuteriu ar žaidimų konsole. Demonstracijos paprastai yra 2-10 minučių ilgio ir (šiuo metu – 2005 m.) nuo kelių iki keliolikos megabaitų apimties.

1.1.3.2. Pristatančios programos

Šioje kategorijoje įvedamas produkto apimties apribojimas – visa programa (programos kodas, grafika, muzika ir visa kita) turi sutilpti į nurodytą dydį. Kūrėjai čia demonstruoja techninius sugebėjimus – kiek vaizdo ir muzikos įmanoma sutalpinti į tuos keliolika ar keliasdešimt kilobaitų.

Standartiniai šių programų dydžiai 4kb, 16kb ir 64kb. Palyginimui galima pasakyti, kad 64 kilobaitus užima daugmaž 1 sekundė „nesuspausto“ garso arba viena „suspausta“ nuotrauka, o

pristatančiose programose į šį dydį sutalpinama vidutiniškai apie 5-15 minučių „filmuko“ su trimačiais modeliais, tekstūromis ir garsu.

1.1.3.3. Muzika

Muzikos varžybos dažniausiai yra padalijamos į keletą kategorijų pagal muzikos stilių (progresyvi, alternatyvi ir t. t.) ir/arba muzikos formatą (keturių kanalų, daugiakanalis ar šiais laikais – MP3/OGG).

1.1.3.4. Grafika

Grafikos varžybos paprastai suskirstomos į nupieštus, sugeneruotus modeliavimo arba spindulių trasavimo programomis.

1.1.3.5. Kita

Kitos kategorijos yra „greitosios rungtys“ – čia kuriama per trumpą laiką; „laukinės rungtys“ – čia rodoma viskas kas nepatenka į kitas kategorijas (gyvi pasirodymai, sukurti filmukai, t. t.); „linksmosios rungtys“ (kompaktinių diskų mėtymas, futbolas ir k. t.).

1.2. „Nesnausk!“ sukurtos demonstracinės programos

1.2.1. „Demo 612“

Demonstracinė programa sukurta per 2003 metų gruodį – 2004 metų sausį, pristatyta Synthesis’04 simpoziume (Prancūzija), kur užėmė 6-ą vietą. Kadangi programa buvo siųsta per internetą, tai ji taipogi dalyvavo ir demonstracijų atsiųstų per internetą kategorijoje, kurioje užėmė 1-ą vietą.

Programoje realizuoti keli standartiniai demonstracijų efektai:

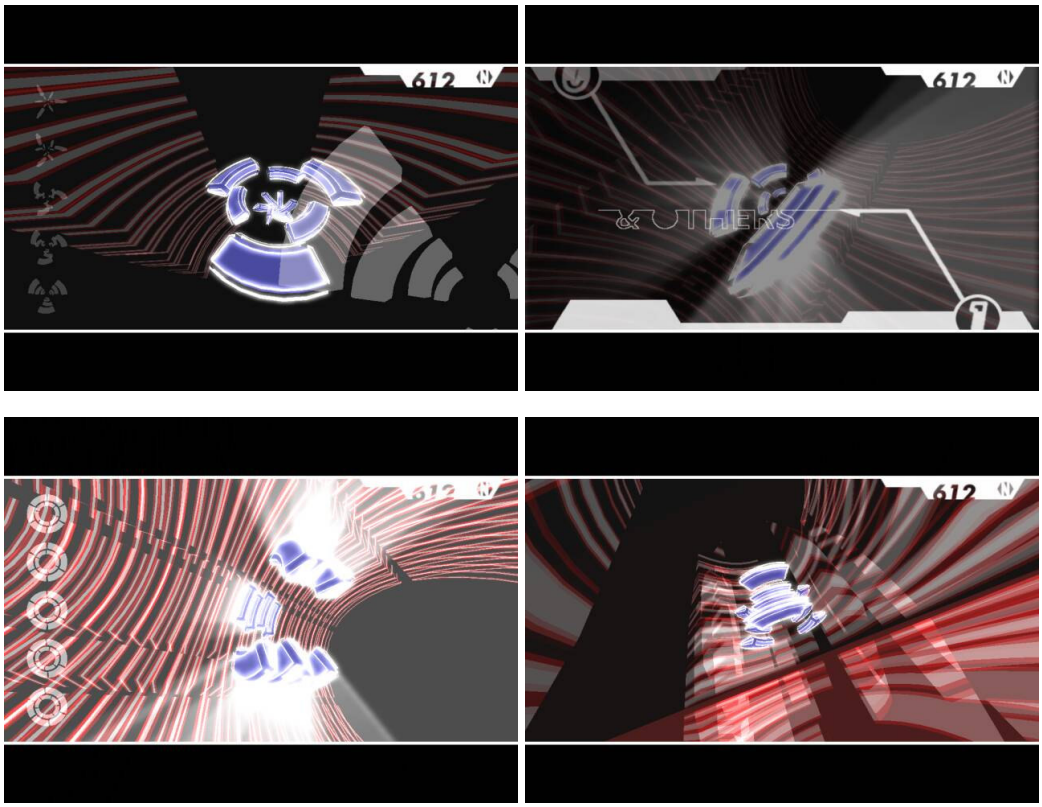
- spindėjimas/švytėjimas, supaprastintas aplinkos atspindėjimas;
- erdviniai šviesos spinduliai.

Visose grupės „Nesnausk!“ kurtose demonstracinėse programose yra naudojamas bendras „variklis“ – rinkinys programos tekstų, kurie yra nepriklausomi nuo kuriamos programos, todėl gali būti naudojamos ne vienoje programoje. Šis rinkinys yra „nusistovėjęs“, tačiau vistiek yra tobulinamas iteracijų metodų kiekvienos demonstracinės programos kūrimo metu, t. y. kuriant programą, jos dalys, kurios gali būti panaudotos vėlesnėse programose, yra perkeliamos į „variklį“ ir

patvarkomos taip, kad būtų kiek įmanoma patogiau naudoti pakartotinai. Šio „variklio“ išeities tekstų dydis šiuo metu yra 1.3Mb (apie 55000 C++ programos teksto eilučių).

Bendroji programos informacija:

- kūrimo trukmė – 2 mėnesiai;
- programos dydis – apie 6.8Mb;
- programos (filmuko) trukmė – 5min;
- pagrindinės programos tekstų dydis – 60Kb (apie 2500 eilučių C++ kodo);
- vaizdą apdorojančios programos dydis – 20Kb (apie 1000 eilučių HLSL kodo);
- demonstracinę programą galima parsisiųsti iš interneto adresu: <http://www.nesnausk.org/project.php?project=10>



3 pav. „Demo 612“ ekrano paveikslėliai

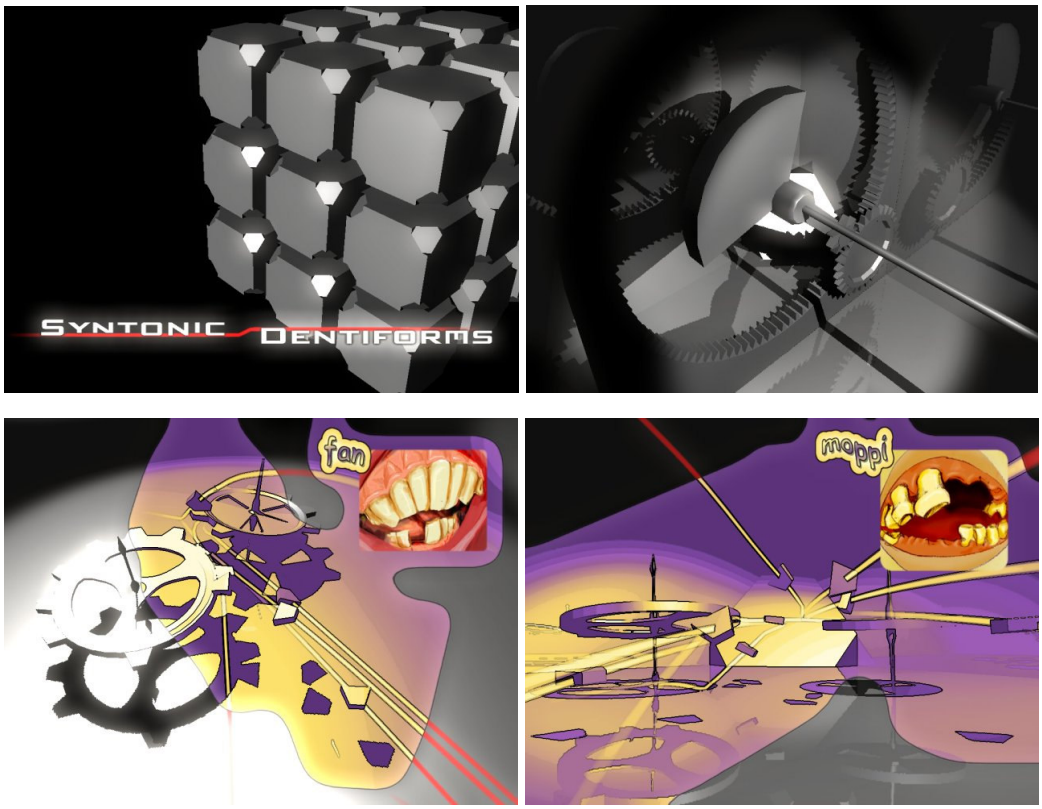
1.2.2. „Syntonic Dentiforms“

Demonstracinė programa sukurta per 2004 metų vasarį – balandį, pristatyta Breakpoint'04 simpoziume (Vokietijoje), kur užėmė 6-ą vietą (iš 22-iejų). Breakpoint – tai vienas didžiausių kasmetinių demonstracinių programų simpoziumų. Šioje parodoje (kaip ir daugumoje kitų demonstracijų parodų) autoriai privalo dalyvauti, taigi į Breakpoint'ą pristatyti savo darbo buvo

nuvykusi ir mūsų komanda. 2005 metų kovą ši demonstracinė programa buvo nominuota ir laimėjo „Scene.org“ „Breakthrought performace“ apdovanojimą.

Realizuoti grafiniai efektai:

- kiekvieno taško difuzinis ir veidrodinis apšvietimas;
- realaus laiko šešėliai;
- daugkartiniai plokšti atspindžiai;
- papildomo apdorojimo efektai (animacinis vaizdavimo stilius ir spindesys).



4 pav. „Syntonic Dentiforms“ ekrano paveikslėliai

Bendroji programos informacija:

- kūrimo trukmė – 3 mėnesiai;
- programos dydis – apie 9Mb;
- programos (filmuko) trukmė – 3,5min;
- pagrindinės programos tekstų dydis – 70Kb (apie 3000 eilučių C++ kodo);
- vaizdą apdorojančios programos dydis – 35Kb (apie 2000 eilučių HLSL kodo);
- demonstracinę programą galima parsisiųsti iš interneto adresu: <http://www.nesnausk.org/project.php?project=11>

1.2.3. „The Fly“

Demonstracinė programa sukurta per 2004 metų kovą – gegužę, dalyvavusi Microsoft korporacijos organizuojamame studentų grafikos programavimo konkurse Imagine Cup, kuriame užėmė 2-ąją vietą. Demonstracinė programa „The Fly“ meniškai perteikia idėją apie „rekursinį pasaulį“.

Realizuoti grafiniai efektai:

- nelygių paviršių imitavimas;
- kameros vaizdo fokusavimo imitavimas;
- iš anksto apskaičiuoto apšvietimo naudojimas;
- kelių sluoksnių vietovės paviršius;
- projektuoti šešėliai;
- vaivorykštiniai permatomi sparnai.

5 pav. „The Fly“ ekrano paveikslėliai

Bendroji programos informacija:

- kūrimo trukmė – 3 mėnesiai;
- programos dydis – apie 28Mb;
- programos (filmuko) trukmė – 3,5min;

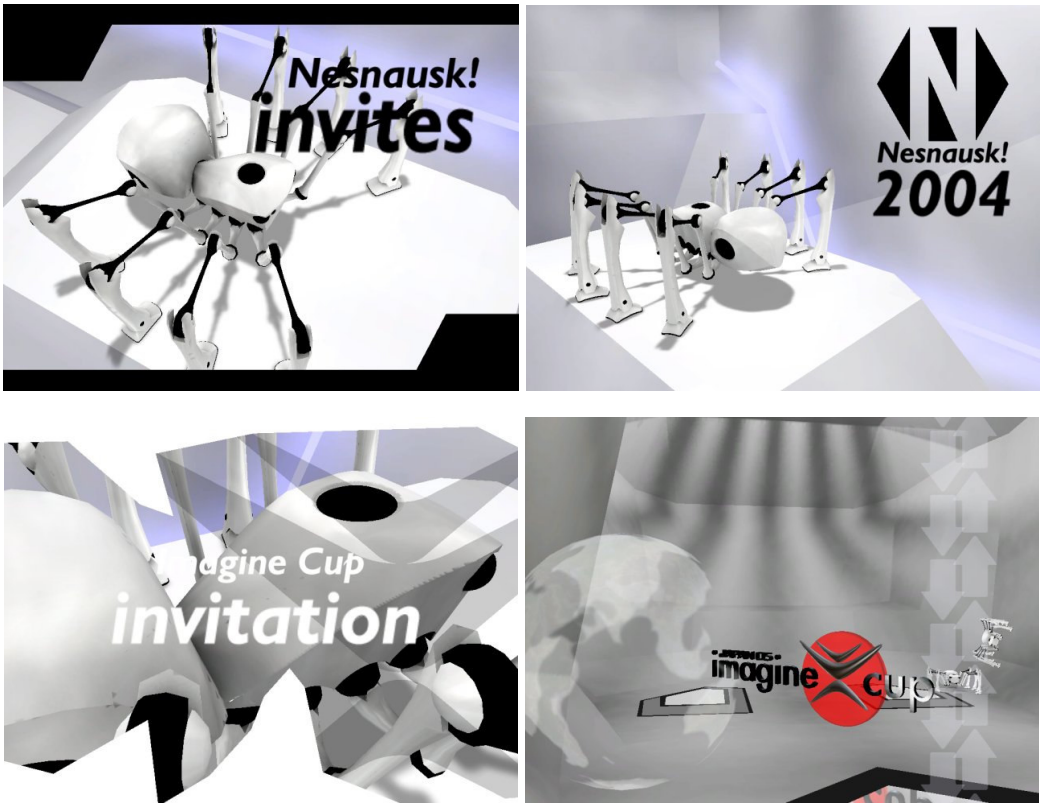
- pagrindinės programos tekstų dydis – 70Kb (apie 3000 eilučių C++ kodo);
- vaizdą apdorojančios programos dydis – 50Kb (apie 2500 eilučių HLSL kodo);
- demonstracinę programą galima parsisiųsti iš interneto adresu: <http://www.nesnausk.org/project.php?project=12>

1.2.4. „Secret government thing“

Demonstracinė programa sukurta per 2004 metų rugpjūtį, dalyvavusi Suomijoje vykusiame Assembly simpoziume. Assembly – tai vienas didžiausių kasmetinių simpoziumų. Konkurso Imagine Cup 2004 finalo metu grupė „Nesnausk!“ ekrano užsklandą, kuri laimėjo pirmąją vietą. Šios užsklandos pagrindu buvo sukurta demonstracinė programa „Secret government thing“.

Realizuoti grafiniai efektai:

- iš anksto apskaičiuoto apšvietimo naudojimas;
- realaus laiko vorų animavimas;
- plokšti atspindžiai, „minkšti“ šešėliai.



6 pav. „Secret government thing“ ekrano paveikslėliai

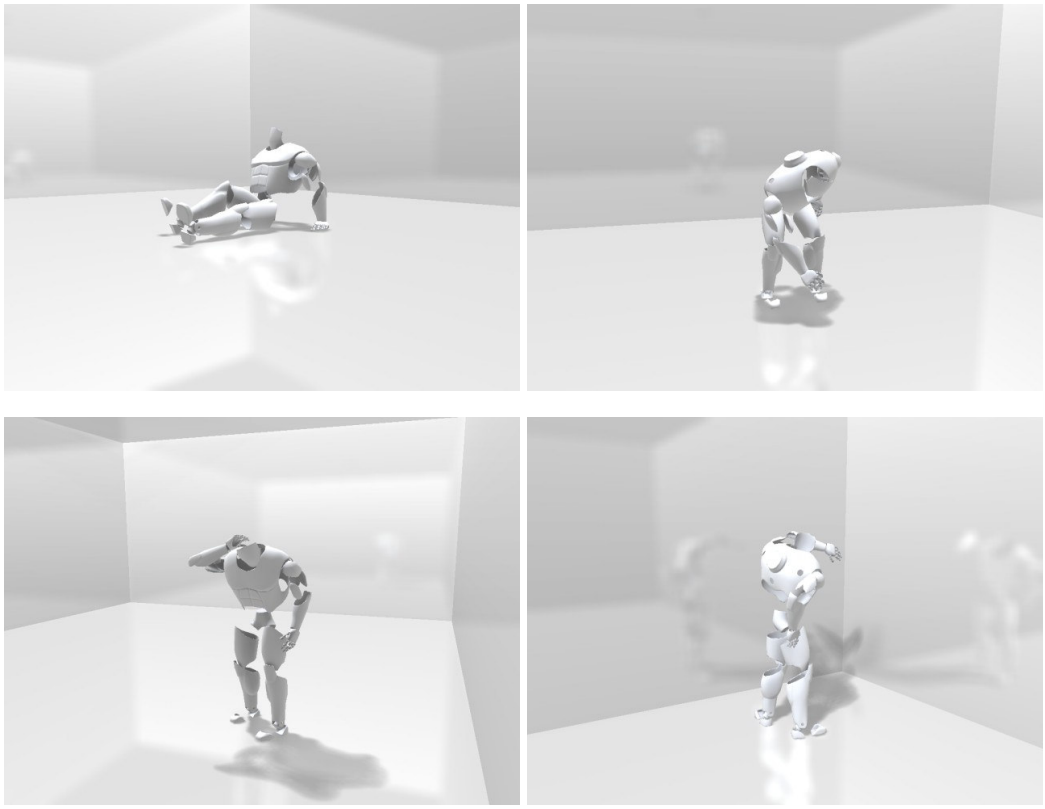
Bendroji programos informacija:

- kūrimo trukmė – 1 mėnuo;

- programos dydis – apie 9Mb;
- programos (filmuko) trukmė – 1,5min;
- pagrindinės programos tekstų dydis – 60Kb (apie 2500 eilučių C++ kodo);
- vaizdą apdorojančios programos dydis – 30Kb (apie 1500 eilučių HLSL kodo);
- demonstracinę programą galima parsisiųsti iš interneto adresu: <http://www.nesnausk.org/project.php?project=13>

1.2.5. „in.out.side: the shell“

Demonstracinė programa sukurta per 2005 metų sausį – gegužę, dalyvaujanti Microsoft korporacijos organizuojamame studentų grafikos programavimo konkurse Imagine Cup 2005.



7 pav. „in.out.side: the shell“ ekrano paveikslėliai

Realizuoti grafiniai efektai:

- objektų paviršiaus krypties saugojimas tekstūrose;
- iš anksto apskaičiuoto apšvietimo naudojimas;
- „minkšti“ šešėliai veikėjams;
- realaus laiko sulieti atspindžiai plastikinio kambario simuliavimui;
- viso ekrano apdorojimo efektai (spindėjimas, vaizdo fokusavimas);

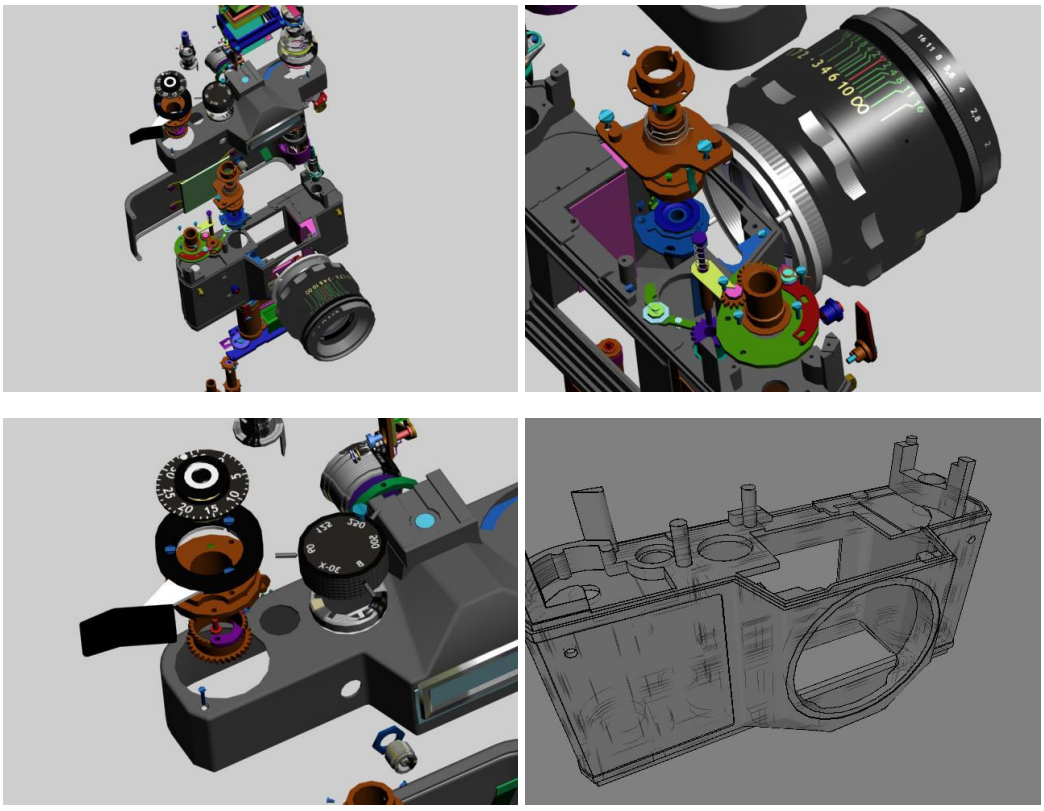
- specialūs efektai elektrai pagrindinio veikėjo viduje ir sienų griovimo „magijai“.

Bendroji programos informacija:

- kūrimo trukmė – 5 mėnesiai;
- programos dydis – apie 40Mb;
- programos (filmuko) trukmė – 5min;
- pagrindinės programos tekstų dydis – 60Kb (apie 2000 eilučių C++ kodo);
- vaizdą apdorojančios programos dydis – 50Kb (apie 3000 eilučių HLSL kodo);

1.2.6. „Zenit“

Demonstracinė programa „Zenit“ buvo pradėta kurti 2004 metais. Programos idėja buvo pavaizduoti fotoaparato „Zenit“ išsiardymą ir surinkimą taip tarsi, tai būtų daroma realiai, t. y. detalės ardomos ir sudedamos nuosekliai, taip kaip būtų ardomas ir surenkamas realus fotoaparatas. Tam buvo sukurtas tikslus fotoaparato modelis, kurį sudaro apie 500 dalių. Kai kurios modelio dalys pavaizduotos 8-ame paveikslėlyje.



8 pav. Fotoaparato „Zenit“ modelio paveikslėliai

Šiai demonstracinei programai norėjome suteikti savitą stilių. Dėl jos techniško turinio nusprendėme pasidomėti, kokie yra metodai techniniams vaizdams kurti. Tam dažniausiai

naudojami nefotorealistiniai vaizdavimo metodai, šiuo atveju mums geriausiai tiko kontūrų vaizdavimas. Pavyzdžiui kontūrų vaizdavimo stiliuje naudojant grafitinį pieštuką primenančius kontūrus ir ortogonalias projekcijas išgaunami vaizdai primena brėžinius darytus ant popieriaus pieštuku. Realiojo laiko kontūrų vaizdavimas buvo pasirinktas šiam darbui plėtoti.

1.3. Kontūrų vaizdavimas

Kontūrai vaidina svarbų vaidmenį mūsų trimatės formos suvokime. Menininkai ir projektuotojai dažnai pabrėžia kontūrus paryškindami detales prie jų arba tiesiog nupiešdami pačius kontūrus. Netgi mažiausias ranka pieštas paveikslėlis dažnai bus sudarytas iš kontūrų ir aštrių detalių. Tobulėjant kompiuteriams dauguma kontūrų vaizdavimo stilių bandoma perkelti į kompiuterinę grafiką. Viena iš analizės šakų orientuojasi į stilizuotus kontūrus, kur kontūrai keičia plotį, tekstūrą ar kažkokiu kitu būdu primena ranka pieštus paveikslėlius. Stilizavimas yra viena iš nefotorealistinio vaizdavimo stiliaus savybių. Patys stilizuoti kontūrai gali vaizduoti medžiagą, iš kurios objektas padarytas, vaizduoti smulkias detales arba tiesiog paryškinti tam tikras svarbias detales (pvz.: užstojamas dalis).

Viena iš problemų nefotorealistiniame vaizdavime – sąryšis tarp vaizdų iš kardo į kadrą, t. y. kontūrų potėpiai turi tolygiai kisti laike, tolydžiai pereidami iš kadro į kadrą keičiantis objektų pozicijoms animuotoje scenoje. Laikinas sąryšis yra ypač sudėtingas kontūrų vaizdavime, nes dažnai sudėtinga nustatyti ryšį tarp atitinkamų kontūrų dviejuose gretimuose kadruose. Be to bandymas sudaryti kontūrų tolydumą ant pačio 3D modelio prieštarauja bandymui sudaryti kontūrų tolydumą ekrano erdvėje. Paskutinis reikalavimas priversti kontūrų vaizdavimo metodus veikti realiu laiku tam, kad jie galėtų būti naudojami žaidimuose, realaus laiko bei demonstracinėse programose.

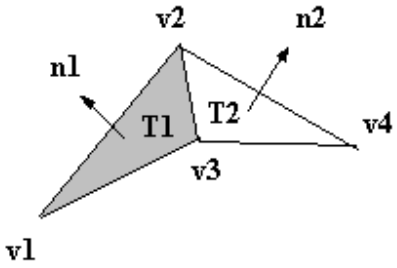
Šiuo metu visus kontūrų vaizdavimo algoritmus pagal veikimo principą grubiai galima suskirstyti į dvi rūšis: geometriniai ir paveikslėlių analize pagrįsti algoritmai.

1.3.1. Objekto kontūras

Objekto kontūras tai yra objekto briaunų rinkinys kurį mes norime paryškinti. 9-ame paveikslėlyje pavaizduota briauna tarp dviejų gretimų trikampių *T1=<v1, v3, v2>* ir *T2=<v2, v3, v3>* einanti iš *v2* į *v3*. Briaunai gretimų trikampių normalės yra lygios:

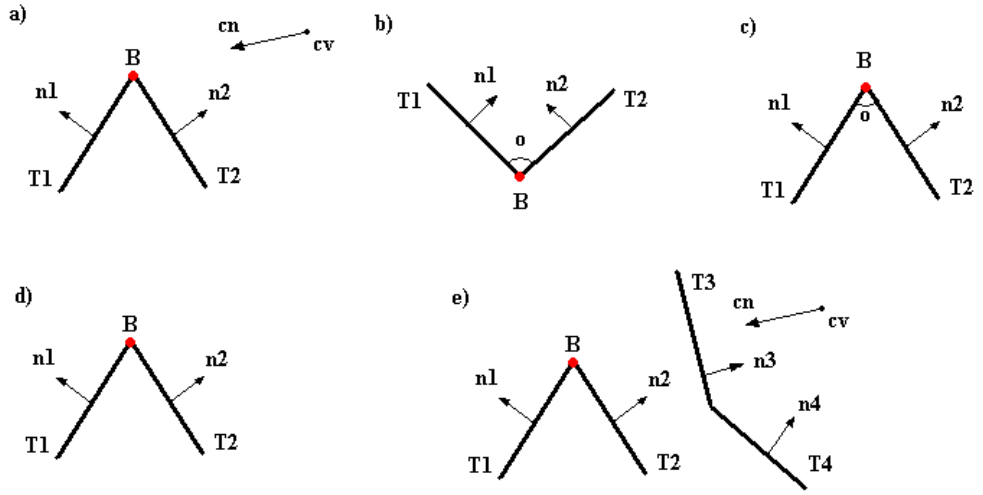
$$n1=(v3-v1)x(v2-v1) / |(v3-v1)x(v2-v1)| \quad (1)$$

$$n2=(v2-v4)x(v3-v4) / |(v2-v4)x(v3-v4)| \quad (2)$$



9 pav. Objekto briauna tarp viršūnių v2 ir v3

Briaunos gali būti: išorinės, įdubusios, iškilusios, pažymėtos ir paslėptos. Kai iš vartotojo yra matomas tik vienas iš gretimų trikampių tai ši briauna vadinama išorine briauna. Tokios briaunos pavyzdys pateiktas 10-o paveikslėlio a dalyje (briauna pažymėta raudonu tašku, ji rodoma išilgai stebėjimo krypčiai). Jei stebėtojas yra *cv* pozicijoje, o jo žiūrėjimo kryptis yra *cn*, tai išorinei briaunai turi galioti savybė: $(cv \cdot n1) \cdot (cv \cdot n1) < 0$. Briauna vadiname įdubusia (10b paveikslėlis), kai jos gretimų trikampių normalės yra atsisukusios viena į kitą, o kampas *o* yra mažesnis tam tikrą vartotojo nustatytą kampą. Briauna vadiname iškilusia (10b paveikslėlis), kai jos gretimų trikampių normalės yra nusisukusios viena nuo kitos, o kampas *o* yra mažesnis tam tikrą vartotojo nustatytą kampą. Pažymėta briauna – tai kuri yra vaizduojama visą laiką. Šias briaunas parenka modeliuotojai kuriantys modelius, tai dažniausiai būna briaunos paryškinančios svarbias detales todėl jas verta rodyti visą laiką. Paskutinioji briaunų rūšis yra „paslėptos“ briaunos, tai tokios briaunos kurios turi būti rodomos (pvz.: jos yra pažymėtos ar išorinės), tačiau jas užstoja kiti trikampiai. Tokios briaunos pavyzdys pateiktas 10-o paveikslėlio e dalyje: briauna *B* turi būti paryškinta, nes ji yra išorinė, tačiau ją užstoja trikampis *T3*. Paslėptos briaunos dažniausiai vaizduojamos naudojant kitokį potepį nei matomos briaunos.



10 pav. Briaunų tipai: išorinė, įdubusi, iškilusi, pažymėta ir „paslėpta“

1.3.2. Techninė įranga

Metodai aprašomi šiame darbe yra orientuoti į modernią programuojamą grafinę įrangą, kuri yra pasiekiama naudojantis tokiomis bibliotekomis kaip OpenGL arba DirectX. Grafinės vaizdo plokštės kanalas (*pipeline*) susideda iš keturių dalių: viršūnių procesoriaus, rasterizatoriaus, taškų procesoriaus ir didelio kiekio (kelių šimtų megabaitų) video atminties. Ši atmintis yra skirta saugoti ekrano buferiui, tekstūroms ir geometriniams objektams. Kanalas jungiantis centrinį procesorių ir vaizdo plokštę dažniausiai yra per lėtas tam, kad būtų galima perduoti tekstūras ir objektų geometriją, kurios reikia kiekvieną kadrą. Tai iškelia tokį reikalavimą: reikia stengtis kas kadrą perdavinėti kuo mažesnį informacijos kiekį, o visą kitą laikyti video atmintyje, į kurią duomenys turi būti užkraunami per paruošiamąjį etapą. Tikslas yra perkelti kiek įmanoma daugiau skaičiavimų į vaizdo plokštę tam, kad apeiti šį pralaidumo apribojimą. Tačiau ne visada šis perkėlimas yra įmanomas – dažnai tai yra problematiška, nes vaizdo plokštėje visi skaičiavimai yra išlygiagretinti. Objektų geometrija yra saugoma viršūnių buferiuose (*vertex buffers*) ir yra perduodama į viršūnių procesorių kaip viršūnių srautas. Pagal programuotojo nurodymą paduodamų viršūnių tvarka gali būti tokia pati kaip viršūnių buferyje arba atitikti tvarką nurodytą indeksų buferyje (*index buffer*). Viršūnių procesorius yra atsakingas už viršūnių transformavimą iš objekto erdvės į ekrano erdvę ir įvairių viršūnės savybių apskaičiavimą (pavyzdžiui.: spalvos, apšvietimo, tekstūrų koordinačių ir pan.). Dažniausiai viršūnių transformacija atliekama dauginant viršūnės poziciją iš taip vadinamos „pasaulio-vaizdo-projekcijos“ matricos. Ši matrica yra sudaryta iš trijų: objekto orientacijos „pasaulyje“, kameros vaizdo ir kameros projekcijos matricų.

Viršūnių procesorius turi dvi savybes, kurios apriboja galimybes, bet labai padidina skaičiavimų greitį. Pirma: kiekviena viršūnė turi būti apdorota atskirai. Jokia informacija negali būti išsaugota tarp dviejų viršūnių apskaičiavimo, taip pat viršūnių procesorius negali rašyti atgal į paduodamą buferį. Antra: negali būti sukurtos naujos ar panaikintos paduotos viršūnės.

Kiekviena viršūnė turi daug laukų, kurie yra žinomi kaip atributai. Dažniausiai naudojami atributai yra 3D pozicija, paviršiaus krypties normalė, keletas tekstūrų koordinačių, spalva. Dažnai programuotojas naudoja šiuos atributus kitiems tikslams pavyzdžiui animacijoms saugoti. Nors viršūnių procesorius negali keisti paduoto buferio, jis gali pakeisti viršūnės atributus prieš perduodant tolesniems skaičiavimams.

Rasterizatorius gauna transformuotų viršūnių informaciją iš viršūnių procesoriaus, paverčia juos į trikampius (ar kito tipo objektus), kurie yra nukarpomi pagal ribojančias plokštumas ir tada suskaido juos į taškus, kurie yra paduodami taškų procesoriui. Rasterizatorius nėra programuojamas, jam galima nurodyti tikai objektų tipą (trikampiai, linijos, keturkampiai ir pan.). Norint padidinti spartą, objektus galima sujungti į juostas, kur kiekvieni du gretimi objektai turi bendras viršūnes, taip yra sutaupoma apdorojamų viršūnių kiekis, o tuo pačiu padidinamas bendras apdorojimo greitis. Reikia atkreipti dėmesį, kad tiktai rasterizatorius dirba su objektais, t. y. viršūnių procesorius dirba su viršūnėmis, o taškų procesorius su taškais ir nė vienas nežino kokie objektai yra apdorojami.

Taškų procesorius apdoroja jam paduotus taškus. Jis negali keisti taško pozicijos, o tiktai paskaičiuoti apšvietimą, parinkti spalvą iš tekstūrų ar kažkokiu kitu būdu apskaičiuoti taško spalvą ir permatomumą.

Šiame darbe aprašomas algoritmas kuriame viršūnių pozicijos informacija yra patalpinta video atmintį paruošiamajame žingsnyje, o viršūnių matomumo informacija paskaičiuojama centriniame procesoriuje ir perduodama į vaizdo plokštę kas kadrą. Vaizdo plokštėje šie du viršūnių srautai sujungiami į vieną ir perduodami į viršūnių procesorių. Viršūnių procesorius naudojant paduotą matomumo informaciją apskaičiuoja viršūnės transformacijos vektorių; naudojant šį vektorių bei „pasaulio-vaizdo-projekcijos“ matricą transformuoja viršūnės poziciją ir apskaičiuoja tekstūrų koordinates. Šios transformuotos viršūnės yra perduodamos į rasterizatorių kuriame 6-ios viršūnės (arba du trikampiai) atitinka vieną briauną. Iš rasterizatoriaus informacija patenka į taškų procesorių kuriame taškam parenkamos spalvos ir permatomumas naudojant gautas tekstūrų koordinates. Apskaičiavus taškų spalvas kontūrai galiausiai supaišomi į ekraną.

1.3.3. Ankstesni darbai

Card [2] ir Gooch [1] pasiūlė metodą kaip briaunai gretimų trikampių normales sutalpinti į viršūnės tekstūros koordinates. Tokiu būdu kiekviena briauna yra vaizduojama kaip keturkampis, kurio plotis yra sumažinamas iki 0, jei nustatoma, kad briauna neturi būti paišoma. McGuire ir Hughes [3] praplėtė šią idėją patalpindami keturias dviejų gretimų trikampių viršūnes vietoj tų trikampių normalių, tai leidžia paskaičiuoti normales animuojamiems objektams. Jie taip pat pasiūlė tekstūrų koordinačių parametrizaciją, tam kad kontūrams būtų galima suteikti potepius. Jų metode buvo įvesti papildomi daugiakampiai, kurie užpildo kampus tarp briaunų keturkampių, tačiau dėl jų naudojamo skaičiavimo metodo kartais užpildomos neteisingos dalys. Didžiausias jų pasiūlyto metodo trūkumas – vaizdo plokštės procesoriaus ir atminties apkrovimas. Kaip buvo minėta 1.3.2 skyriuje norint visus skaičiavimus atlikti naudojant vien tik vaizdo plokštės procesorių reikia duplikuoti daug duomenų ir skaičiavimų. Dėl šio duplikavimo McGuire ir Hughes metodas naudoja apie 9 kartus daugiau atminties ir veikia apie 15 kartų lėčiau nei standartinis objekto nupiešimas (t. y. objekto nupiešimas su viena tekstūra ir apšviesto vienu šviesos šaltiniu). Remiantis McGuire ir Hughes gautais rezultatais, buvo nuspręsta perskirstyti skaičiavimus tarp centrinio procesoriaus ir vaizdo plokštės, tokiu būdu buvo sumažintas vaizdo plokštės procesoriaus ir atminties apkrovimas, tačiau padidintas duomenų magistralės bei centrinio procesoriaus apkrovimas. Taip pat šiame darbe siūlomas naujas metodas briaunų keturkampių tarpams užpildyti, kuris išsprendžia problemas, su kuriomis susidūrė McGuire ir Hughes.

Kontūrų paišymo algoritmus galima suskirstyti į dvi kategorijas: tuos kurie sukuria kontūrų geometriją ir tuos kurie analizuoja paveikslėlius. Egzistuoja dar viena metodų grupė panašių į geometrinius algoritmus. Vieną iš tokių aprašo Raskar [10]. Šis metodas sukuria padidintą daugiakampį už kiekvieno trikampio, tokiu būdu iš stebėtojo pozicijos matomas kontūras. Kiek anksčiau pasiūlyto metodo idėja buvo padidinti nuo stebėtojo nusisukusius trikampius ir „apsukti“ juos į stebėtoją (šį metodą jau gan seniai naudoja kompiuterinės grafikos menininkai kurdami animacinio stiliaus veikėjus). Šių metodų realizacija buvo orientuota į centrinį procesorių, tačiau skaičiavimus nesunku pilnai perkelti į vaizdo plokštės procesorių. Raskar metodas yra gan ribotas – jis negali paišyti storų ar stilizuotų kontūrų, o tik plonas linijas, taip pat naudojant šį metodą yra supaišoma daug trikampių, kurie yra uždengti, o tai sąlygoja mažą efektyvumą. Dėl šių algoritmų prasto našumo ir vaizdo kokybės santykio šiuo metu jie nėra aktyviai tobulinami.

1.3.4. Paveikslėlių analize pagrįsti kontūrų algoritmai

Paveikslėlių analize pagrįsti metodai išanalizuoja galutinį vaizdą tam, kad nustatytų kuriose vietose yra kontūrai. Vieną iš tokių metodų aprašė Saito ir Takahashi [5]. ATI programuotojai parodė [6] kaip naudojant šiuolaikines vaizdo kortas visus skaičiavimus galima atlikti nenaudojant pagrindinio procesoriaus, t. y. visus kontūrų nustatymo skaičiavimus atlikti naudojant tik vaizdo plokštės procesorių, tai yra labai didelis privalumas algoritmui, nes daugelyje programų centrinis procesorius būna apkrautas žymiai labiau nei vaizdo plokštės procesorius. Šio tipo algoritmų pagrindiniai žingsniai:

1) visų objektų supaišymas į ekrano buferį;
2) kontūrų nustatymas atliekant ekrano buferio analizę;
3) rastų kontūrų supaišymas.

Iš 1-o žingsnio galime matyti, kad šio tipo algoritmams kontūrų nustatymo trukmė yra nepriklausoma nuo objektų kiekio, jų dydžio ir išsidėstymo ekrane. Tai yra didelis privalumas kai yra daugybė objektų ir jie vienas kitą dengia – kontūrai nustatinėjami tik matomoms dalims, tačiau tai tampa trūkumu, kai objektų yra mažai ir jie dengia tik mažą ekrano dalį, nes tokiu atveju vistiek yra analizuojamas visas ekrano buferis.

Dažniausiai yra sudaromi keli skirtingų tipų ekrano buferiai, nes standartinis ekrano buferis laiko tik objekto spalvos (ir apšvietimo) informaciją. Analizuojant tokį buferį įmanoma nustatyti tiktai kontūrus tarp skirtingų spalvų. Taigi norint nustatyti daugiau tipų kontūrų paišant objektus yra pildomi keli ekrano buferiai. Ne visada įmanoma užpildyti visus reikiamus ekrano buferius per vieną kartą, todėl kartais tenka visus objektus supaišinėti keltą kartų pildant vis kitus ekrano buferius. Keletas svarbiausių ekrano buferio tipų:

- spalvos – galima nustatyti kontūrus tarp spalvų;
- objekto identifikacijos – objekto siluetams nustatyti;
- apšvietimo – kontūrams tarp skirtingų apšvietimų;
- paviršiaus krypties – įdubusiems ir iškilusiems kontūrams.

Kontūrai nustatomi ten kur yra ekrano buferio netolydumai (skirtumas tarp dviejų gretimų reikšmių peržengia tam tikrą nustatytą ribą) arba reikšmė viršija tam tikrą slenkstinę ribą. Šių metodų privalumas yra toks, kad jie gali rasti kontūrus ten kur jie nebūtinai yra objekte, pvz.: parametrizuotose kreivėse arba ten kur susiduria dvi plokštumos. Šiame darbe analizuojamas metodas to padaryti negali.

Didžiausias šio tipo algoritmų trūkumas – jie negali atvaizduoti stilizuotų kontūrų (arba tos galimybės yra labai ribotos). Dažniausiai šio tipo algoritmuose kontūrai yra tiesiog paryškinami, tačiau jiems negali būti suteikta tam tikra tekstūra. Keletą primityvių tokių kontūrų stilizavimo metodų aprašo Gooch [8], Dietrich [9] ir Everitt [7].

1.3.5. Geometriniai kontūrų nustatymo algoritmai

Ši, antroji algoritmų grupė, yra pagrįsta objekto geometrijos informacijos analize stebėtojo atžvilgiu. Iš nustatytų matomų kontūrų konstruojami daugiakampiai, kurie yra perduodami vaizdavimui. Didžiausias šių algoritmų privalumas yra tai, kad jie įgalina atvaizduoti stilizuotus kontūrus (pvz.: kontūrus tarsi jie būtų piešti pieštuku ar anglimi ir pan.). Priešingai nei ekrano buferius analizuojantys algoritmai, šių algoritmų veikimo greitis yra tiesiogiai priklausomas nuo objekto detalumo, nes norint nustatyti matomas briaunas naudojant „grubios jėgos“ algoritmą reikia patikrinti visų briaunų matomumą, t. y. atliekamo darbo sudėtingumas O(N), kur N briaunų skaičius. Dalis tyrimų orientuojasi į šio sudėtingumo sumažinimą. Hall [11] siūlo mažint sudėtingumą įsimenant paskutinio kadro briaunų matomumą, tačiau jis efektyvūs tik tuo atveju, kai pokyčiai tarp kadrų yra nedideli. Gooch [8] suranda kontūrus trikampių normales traktuodamas kaip sferos taškus, o briaunas kaip arkas tarp šių vektorių, tokiu atveju reikalingas briaunas galime rasti turimą sferą kertant plokštuma orientuota į stebėtoją.

Antra problema susijusi su objektų detalumu yra tai, kad mažo detalumo objektams yra būdingas kontūrų pozicijos netolydumas, t. y. galima pastebėti kada pasikeičia išorinis kontūras: viena briauna tampa nematoma, kita pasidaro matoma. Ši problema sprendžiama analizuojant rastų matomų kontūrų aibę iš jos sukonstruojant naują aibę, bei palaikant tolydumą tarp šių sukonstruotų aibių gretimuose kadruose. Kalnins aprašo metodą [12] kuriame išgaunamas beveik idealus iš kadro į kadrą tolydumas. Deja šis metodas negali būti realizuotas vaizdo plokštės procesoriuje, todėl šiame darbe tenka naudoti žymiai primityvesnius metodus. Šiame darbe realizuota tekstūrų parametrizacija yra žymiai prastesnės kokybės, tačiau gali būti skaičiuojama kelioms viršūnėms lygiagrečiai, o tai lemia gerokai didesnį našumą.

Geometrinio kontūrų vaizdavimo algoritmų pagrindiniai žingsniai:

1) objekto nupaišymas;
2) matomų briaunų nustatymas;
3) matomų briaunų nupaišymas.

Daugumoje šių algoritmų didžiausia bėda – kontūrų sąrašo perdavimas į vaizdo kortą, todėl naujesniuose metoduose siūloma kuo didesnę dalį skaičiavimų perkelti į vaizdo kortą.

1.3.6. Kontūrų vaizdavimo analizės išvados

1. Dauguma ankstesnių metodų realizacijų buvo orientuota į centrinį procesorių, todėl kontūrų matomumo nustatymo ir vaizdavimo vaizdo plokštėje galimybės nėra plačiai išanalizuotos.
2. McGuire ir Hughes [3] pasiūlytas metodas visus skaičiavimus atlieka vaizdo plokštėje, tačiau naudoja labai daug atminties.
3. Vien tik vaizdo plokštėje skaičiavimus atliekantys metodai susiduria su labai dideliais apribojimais, nes visus skaičiavimus reikia atlikti lygiagrečiai, todėl kartais nukenčia vaizdo kokybė, pvz.: paišant storus kontūrus matosi aštrūs kampai tarp briaunų.
4. Šiame darbe buvo nuspręsta realizuoti kontūrų matomumo nustatymo ir vaizdavimo algoritmą, kuris briaunų matomumo skaičiavimus atliktų centriniame procesoriuje, o briaunų transformavimo ir vaizdavimo dalį – vaizdo plokštėje.
5. Taip pat buvo nuspręsta pabandyti išspęsti „aštrių briaunų galų“ problemą, sujungiant gretimas briaunas arba nupaišant briaunų galuose apskritimus.

2. TIRIAMOJI DALIS

2.1. Algoritmas

Algoritmo pagrindinė idėja yra viršūnių sąraše perduoti informaciją ne apie objekto viršūnes ir trikampius, o apie trikampius, kurie atitinka objekto briaunas. Aprašomame algoritme galima išskirti tris pagrindinius žingsnius:

1) Paruošiamasis, kurio metu iš paduoto objekto struktūros yra suskaičiuojama viršūnių, briaunų ir trikampių topologija. Taip pat į vaizdo kortos atmintį perduodama dalis briaunų viršūnių informacijos.
2) Briaunų matomumo apskaičiavimas ir matomumo informacijos perdavimas į vaizdo kortą. Šis žingsnis atliekamas prieš supaišant kiekvieną kadrą.
3) Briaunų transformavimas ir tekstūrų koordinačių paskaičiavimas.

Atlikus šiuos žingsnius viršūnės yra perduodamos rasterizatoriui, po to taškų procesoriui, kol galiausiai briaunos yra supaišomos į ekraną, tačiau mūsų šie žingsniai nedomina, nes jie yra už aprašomo algoritmo ribų.

2.1.1. Briaunų struktūra

Algoritmo suformuotame viršūnių sąraše kiekviena viršūnė saugo informaciją apie briauną (kiekviena briauna yra sudaryta iš keturių iš eilės einančių viršūnių). Tam kad galėtume atlikti briaunos transformaciją apdorojant po vieną viršūnę ir neturint informacijos apie kitas viršūnes, mes sutalpiname visą reikalingą informaciją į kiekvieną briaunos viršūnę. Tokiu atveju mūsų briaunos viršūnė turės tokius atributus: <*v, e*>, kur v yra briaunos galo pozicija, e –briaunos krypties vektorius. v sąrašas yra paduodamas į vaizdo kortą, o e sąrašas - paskaičiuojamas ir perduodamas kas kadrą. Paruošiamajame žingsnyje kiekvienai objekto briaunai sukuriamas atitinkamas keturkampis, kuris yra sudarytas iš keturių iš eilės einančių viršūnių. Jei briaunos indeksas yra *j*, tai jai bus sukurtos keturios atitinkančios viršūnės su indeksais: *4j*, *4j+1*, *4j+2*, *4j+3*, kurioms galios šios savybės: *v[4j] = v[4j+1]*, *v[4j+2] = v[4j+3]*, *e[4j] = -e[4j+1]* ir *e[4j+2] = -e[4j+3]*.

2.1.2. Paruošiamasis žingsnis

Paruošiamojo žingsnio metu yra optimizuojamas paduotas objektas, kurio braunas reikės rodyti, bei paskaičiuojama viršūnių, briaunų ir trikampių topologija, taip pat sudaromas sąrašas briaunų, kurios yra vaizduojamos visą laiką.

Optimizavimo žingsnis mūsų atveju yra paprastas: yra sujungiamos viršūnės, tarp kurių atstumas yra mažesnis už tam tikrą programuotojo nustatytą. Nors šis žingsnis yra labai paprastas, tačiau jis yra be galo svarbus, nes dažnai modeliavimo programomis sukurtiems modeliams yra būdinga, tai kad viena viršūnė priklausanti keliems trikampiams yra saugoma kaip kelios skirtingos viršūnės, t. y. tarp šių vektorių atstumas būna lygus 0, tačiau programa jas traktuoja kaip skirtingas viršūnes. Nesujungus tokių viršūnių nebūtų įmanoma paskaičiuoti topologijos informacijos.

Topologijos informacija susideda iš kelių gretutinumo sąrašų, kurie yra reikalingi skaičiavimams kiekviename kadre. Pirma – norint žinoti ar briauna yra išorinė, reikia žinoti jai gretimus trikampius, bei jų matomumą. Antra – tam, kad greitai suskaičiuoti kiek paryškinamų briaunų priklauso kiekvienai viršūnei, mes pasiruošiame kiekvienai briaunai gretimų viršūnių sąrašą (t. y. kiekvienai briaunai po dvi jos galų viršūnes).

Trečioji paruošiamojo žingsnio dalis yra visada paryškinamų briaunų sąrašo sudarymas. Į šį sąrašą įeina įdubusios, iškilusios ir pažymėtos briaunos (jos aprašomos 1.3.1 skyrelyje). Taigi kas kadrą mums reikia nustatyti kurios yra išorinės briaunos, visos kitų paryškinimo požymis gali būti apskaičiuotas paruošiamojo žingsnio metu, nes jis nepriklauso nuo objekto ir stebėtojo orientacijos ir pozicijos vienas kito atžvilgiu.

Paskutinė paruošiamojo žingsnio dalis yra statinio briaunų viršūnių sąrašo sudarymas. Šios dalies metu mes sudarome atributų *v* sąrašą, kuris yra aprašytas 2.1.1 skyrelyje. Taip pat užpildome indeksų buferį, kuris kas kadrą naudojamas briaunoms vaizduoti.

2.1.3. Briaunų matomumo apskaičiavimas

Šio žingsnio metu yra apskaičiuojamas išorinių briaunų matomumas, kuris sujungiamas su iš anksto apskaičiuotų briaunų matomumu ir galiausiai naudojant šią matomumo informaciją yra sudaromas *e* atributų sąrašas (aprašytas 2.1.1 skyrelyje), kuris perduodamas į vaizdo kortą. Išorinių briaunų matomumas apskaičiuojamas keliais žingsniais:

1) Paskaičiuojama visų trikampių matomumas. Trikampis yra matomas tada, kai jo normalės kryptis yra priešinga stebėtojo krypčiai, t. y. $n \cdot cn < 0$, kur n yra trikampio normalė, o cn stebėto kryptis.
2) Naudojant pirmame žingsnyje paskaičiuotą trikampių matomumo informaciją bei paruošiamojo žingsnio metu sudarytą briaunai gretimų trikampių informaciją yra surašoma briaunų matomumo informacija. Briauna yra paryškinama tada, kai jai galioja sąlyga: *(t1 and not t2) or (not t1 and t2)*, kur *t1* ir *t2* briaunai gretimų trikampių matomumo požymiai.

Turint briaunų matomumo informaciją yra sudaromas *e* atributų sąrašas. Kiekvienai matomai briaunai yra perduodami keturi *e* atributai: *+eb, -eb, +eb, -eb*, kur *eb* yra briaunos krypties vektorius, o nematomoms briaunoms perduodami keturi nuliniai vektoriai: *e0, e0, e0, e0*, kur *e0=(0,0,0)*.

Šiame žingsnyje taip pat atliekamos kelios papildomos vektoriaus *eb* modifikacijos, tačiau šios modifikacijos atliekamos norint išgauti papildomus vizualinius efektus, o ne pakeisti briaunų matomumą, to dėl jų aprašymas pateiktas tolesniuose skyriuose.

2.1.4. Briaunų transformavimas

Briaunų transformavimas atliekamas transformuojant briaunos viršūnes po vieną vaizdo kortoje, viršūnių procesoriuje. Į viršūnių procesorių ateina jau sujungti *v* ir *e* srautai, taigi jis kiekvienai viršūnei gauna du atributus: <*v, e*>. Kiekvienos viršūnės transformavimas atliekamas per kelis žingsnius (formulių dėmenys pavaizduoti 11-ame paveikslėlyje):

1) Paskaičiuojame vektorių einantį iš stebėtojo į duotą viršūnę (*cp* stebėtojo pozicija):

$$dir = v - cp$$

(3)

2) Paskaičiuojame viršūnės transformacijos vektorių, kuris yra statmenas vektoriui *dir* ir briaunos krypties vektoriui *e*:

$$n = \frac{e \times dir}{|e \times dir|},$$

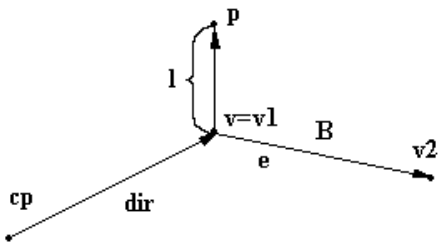
(4)

jei vektoriaus *e* ilgis yra lygus nuliui (t. y. briauna turi būti nematoma), tai vektoriui *n* priskiriamas nulinis vektorius, t. y. *n=(0, 0, 0).* Tokiu būdu gaunamos briaunos plotis lygos 0, taigi jos nesimato. Yra ir kitas briaunų paslėpimo būdas: vektoriui *n* priskirti *–dir* reikšmę, tokiu būdu vektorius *p* perstumiamas už stebėtojo „nugaros“. Tokios viršūnės būtų atmestos nes jos patektų už ribojančios plokštumos. Šis metodas padeda įveikti viršūnių procesoriaus nesugebėjimą pašalinti viršūnių. Atmetimas pagal ribojančias plokštumas vykdomas prieš viršūnėms patenkant į rasterizatorių, todėl nematomos briaunos nedaug apkrauna vaizdo plokštę.

3) Galiausiai *n* vektorius pridedamas prie *v* ir šis dauginamas iš „vaizdo-projekcijos“ matricos *VP* (*l* - briaunos pločio faktorius):

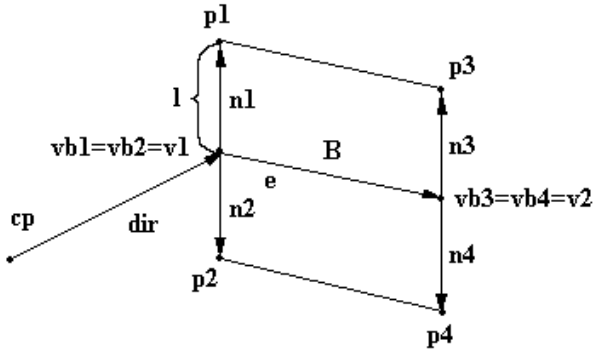
$$p = (v + n * l) * VP,$$

(5)



11 pav. Briaunos viršūnės transformavimas

Visi viršūnės transformavimo skaičiavimų dėmenys pavaizduoti 11-ame paveikslėlyje. Pateiktame paveikslėlyje yra transformuojama briaunos *B* einančios iš *v1* į *v2* viršūnė *v1*, kuri turi du atributus <*v, e*> (*v*=*v1*, *e*=*v2*-*v1*). *cp* yra stebėtojo pozicija, *dir* – vektorius einantis iš stebėtojo į duotą viršūnę, *l* – briaunos pločio faktorius (bendras visoms objekto briaunoms), o *p* – galutinė viršūnės pozicija.



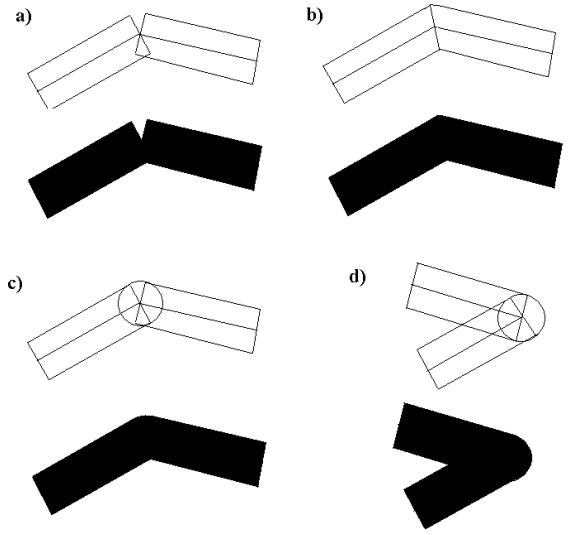
12 pav. Visų briaunos viršūnių transformavimas

Visų briaunos viršūnių transformacija pateikta 12-ame paveikslėlyje. Briauna *B* einanti iš viršūnės *v1* į *v2* viršūnių procesoriui yra pateikiama kaip keturios atributų poros: <*vb1, eb1*>, <*vb2, eb2*>, <*vb3, eb3*> *ir* <*vb4, eb4*> (įrašius reikšmes gautume: <*v1, +e*>, <*v1, -e*>, <*v2, +e*> *ir* <*v2, -e*>). Dviems viršūnėms esančioms tame pačiame briaunos gale (pvz.: 1-ai ir 2-ai viršūnėms) vektorius *e* įrašomas su skirtingais ženklais, todėl gaunami vektoriai n1 ir n2 yra priešingų ženklų. Pridėjus šiuos vektorius prie vektorių *vb1* ir *vb2*, gauname briauną, kurios plotis – *2l*. Jei paduoti vektorių *eb* ilgiai yra lygūs nuliui (t. y. briauna turi būti nematoma), tai visi vektoriai n irgi turės ilgį 0, taigi briaunos plotis bus 0. Patransformavus visas briaunos viršūnes gauname briauną atitinkantį keturkampį *[p1, p2, p4, p3]*.

Viršūnių procesoriuje apskaičiavus viršūnės pozicija taip pat yra apskaičiuojama jos tekstūros koordinatės. Šie skaičiavimai pateikiami tolesniuose skyriuose.

2.1.5. Storos briaunos

Ankstesniuose skyriuose aprašyti skaičiavimai puikiai tinka plonoms briaunoms (tokioms kurių plotis yra keli taškai), tačiau platesnėms briaunoms išryškėja aštrūs briaunų galai ir tarpai tarp gretimų briaunų (žiūrėti 13-o paveikslėlio a dalį).



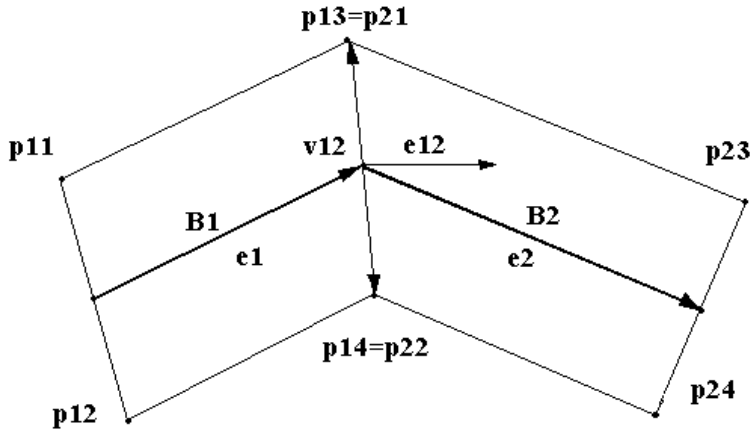
13 pav. a - aštrūs briaunų kampai ir tarpai tarp jų; b, c, d - šių problemų sprendimo būdai

Paveikslėlyje 13b pateiktas vienas iš tokių problemų sprendimų būdų: gretimų briaunų galus sutapatinant su pusiaukampine tarp briaunų. Šis sprendimas tinka tik tokioms situacijoms kur sueina dvi briaunos, o pakankamai dažnai pasitaiko, kad į vieną tašką sueina daugiau nei dvi briaunos, ypač kai yra įdubusių ar iškilusių briaunų (pvz.: kubo kampas). Antrasis sprendimas pateiktas 13-o paveikslėlio c ir d dalyse. Jo idėja nupiešti apskritimą visur kur sueina kelios briaunos. Jis puikiai veikia netgi tuose atvejuose, kai kampas tarp briaunų yra smailas (13d paveikslėlis), tačiau šio metodo trūkumas, kad jo neįmanoma realizuot nenusiunčiant į vaizdo kortą daugiau viršūnių, taigi padidėja atminties ir kanalo tarp vaizdo plokštės ir centrinio procesoriaus apkrovimas.

2.1.5.1. Storų briaunų realizacija

Šiame darbe buvo nuspręsta 1-ą problemos spendimo būdą pateiktą 2.1.5 skyrelyje (13b paveikslėlis) ten, kur susiduria dvi briaunos, nes jam lengviau pritaikoma tekstūrų parametrizacija, bei realizuoti 2-ą sprendimo būdą (13 c ir d paveikslėliai) visais kitais atvejais. Norint realizuoti

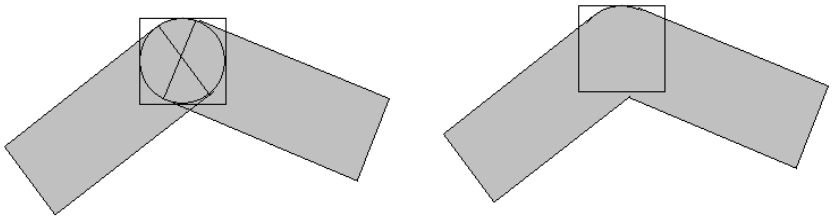
storų briaunų pirmąjį atvejį, tereikia kiek kitaip suformuoti briaunų viršūnių *e* atributus. Tokios realizacijos brėžinys pateiktas 14-ame paveikslėlyje.



14 pav. Storų briaunų realizacija

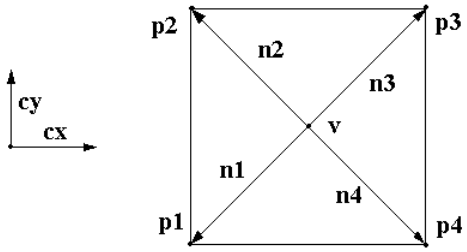
Standartiniu atveju briaunos B1 3-ios ir 4-os viršūnių atributai būtų <*v12, +e1*> ir <*v12, -e1*>, briaunos B2 1-os ir 2-os - <*v12, +e2*> ir <*v12, -e2*>. Norint padaryt taip, kad p13 ir p21, bei p14 ir p22 sutaptų reikia pakeisti atributų poras taip: B1 3 - <*v12, +e12*>, 4 - <*v12, -e12*>; B2 1 - <*v12, -e12*>, 3 - <*v12, +e12*>. Visa kita realizacija išlieka tokia pati.

Antras sprendimo atvejis kiek komplikuotesnis – neužtenka vien tik modifikuoti esamus duomenis, o reikia papildomų viršūnių iš, kurių būtų naudojamos apskritimui supaišyti. Be to ne visada reikia tuos apskritimus paišyti, o tik tada kai yra 1, 3, arba daugiau matomų briaunų sueinančių į šį tašką. Dėl šių priežasčių buvo pasirinktas identiškas sprendimas kaip ir briaunų paišymo atveju, t. y. dalį informacijos laikyti vaizdo kortoje, o matomumo informaciją paskaičiavus centriniame procesoriuje perduoti kas kadrą. Apskritimui nupaišyti galimi keli sprendimo būdai: suformuoti visą apskritimą iš viršūnių, tačiau tam, kad suformuot apvalų (su mažom paklaidom) apskritimą reikia daug viršūnių, arba galima paišyti daugiakampį su permatoma tekstūra, kurioje yra pavaizduotas apskritimas. Šiame darbe naudojamas antras sprendimo būdas apskritimą paišant ant kvadrato (15-as pav.).



15 pav. Tarpo užpildymas naudojant kvadratą su apskritimo tekstūra

Šio keturkampio kiekviena viršūnė turi tris atributus <*v, tx, ty*>, kur *v* kvadrato centras, o *tx* ir *ty* naudojamas tekstūrų koordinatėms ir matomumo informacijai perduoti. Atributų *v* sąrašas į vaizdo kortą perduodamas per paruošiamąjį žingsnį, o <*tx, ty*> suformuojama kas kadrą pagal briaunų matomumą. Visoms keturioms keturkampio viršūnėms yra perduodamas tas pats *v* – keturkampio centras, o <*tx, ty*> suformuojamos tokios reikšmės: <*0, 0*>, <*0, 1*>, <*1, 1*> ir <*1, 0*>, jei keturkampis turi būti matomas, arba visoms viršūnėms perduodama <*0, 0*>, jei turi būti nematomas.



16 pav. Kvadrato viršūnių transformavimas

Viršūnių procesoriuje atliekami tokie skaičiavimo žingsniai (brėžinys pateiktas 16-ame paveikslėlyje):

1) Paskaičiuojame viršūnės transformacijos vektorių, kuris yra statmenas vektoriui *dir* ir briaunos krypties vektoriui *e*:

$$n = (tx - 0.5) * cx + (ty - 0.5) * cy ,$$

(6)

čia cx – stebėtojo vektorius į dešinę, o cy – į viršų. Jei visų viršūnių tx ir ty sutaps (t. y. keturkampis turi būti nematomas), tai sutaps ir n vektoriai, taigi gausime keturkampį kurio dydis 0. Paslepiant šiuos keturkampius galima taikyti tą patį sprendimą kaip ir briaunų atveju – perstumti viršūnes už stebėtojo „nugaros“.

2) Galiausiai *n* vektorius pridedamas prie *v* ir šis dauginamas iš „vaizdo-projekcijos“ matricos *VP* (*r* – apskritimo spindulys):

$$p = (v + n * r) * VP ,$$

(7)

Viršūnių procesoriuje taip pat užpildomos tekstūrų koordinatės, tačiau siuo atveju jų skaičiuoti nereikia, o jos paimamos tiesiai iš *tx* ir *ty* atributų.

2.1.6. Paslėpti kontūrai

Naudojant šį algoritmą mes taip pat galima pavaizduoti paslėptus kontūrus naudojant kitokį potepį (techninėse iliustracijose tokie kontūrai vaizduojami brūkšninėmis arba taškinėmis linijomis). Tai atliekama vaizdo kortoje keliais žingsniais (angl.: *passes*):

1) į ekraną supaišomi visi objektai, o į gylio buferį surašomi atstumai nuo stebėtojo iki paišomo objekto taško;
2) paišomi visi kontūrai tikrinant gylio informaciją gylio buferyje, jei tam tikra kontūro dalis yra toliau nuo stebėtojo nei gylio buferyje įrašytos reikšmės, tai ta kontūro dalis nepaišoma;
3) gylio vertinimą padarome priešingą nei 2-ame žingsnyje ir vėl paišomi visi kontūrai tikrinant gylio informaciją gylio buferyje. Vykdomas toks gylio tikrinimas: paišomos tik tos kontūro dalys kurios yra toliau nuo stebėtojo nei gylio buferyje įrašytos reikšmės.

Pirmojo žingsnio metu bus supaišyti visi objektai, antrojo – matomi kontūrai, o trečiojo – paslėpti kontūrai. Šie žingsniai realizuoja Appel pasiūlytą idėją [18]. Paslėptų kontūrų pavyzdžius galite rasti 1-ame, 2-ame ir 8d paveikslėliuose.

2.1.7. Tekstūrų parametrizacija

Norint išgaut stilizuotus kontūrus (pvz.: brūkšnines ar vingiuotas linijas) jiem yra suteikiamos tekstūros (tekstūruoto ir netekstūruotų kontūrų pavyzdžiai pateikti 17 pav.). Šios tekstūros yra supaišytos taip, kad x koordinatė kinta išilgai kontūro, o y kinta nuo 0 iki 1 nuo išorinio iki vidinio briaunos krašto. Šios tekstūros padarytos taip, kad sudėjus horizontaliai dvi viena šalia kitos, tarp jų nesimato sujungimo, t. y. vienos galas sutampa su kitos pradžia. Reikalinga tam tikra parametrizacija, kuri kiekvienai kontūro viršūnei priskiria atitinkamą tekstūros koordinačių porą taip, kad tekstūra eitų išilgai kontūro. Gera parametrizacija maksimaliai sumažina tekstūros netolydumus erdvėje ir laike (gretimuose kadruose). Ankstesni metodai galėjo patenkinti šiuos reikalavimus sujungdami gretimas briaunas į vieną ištisinį kontūrą, taip išgaunant tolydumą erdvėje, bei išanalizuojant praeito kadro kontūrus, tai išgaunant tolydumą laike. Tai buvo įmanoma, nes buvo atliekama centriniame procesoriuje. Naudojant esamas vaizdo kortas to padaryti neįmanoma, todėl tenka naudoti gerokai primityvesnius metodus.

Šiame darbe naudojamas kontūrų tekstūrų parametrizacija ekrano erdvėje. Ši parametrizacija vykdoma ekrano erdvėje ir gali būti pilnai atliekama viršūnių procesoriuje. Mus domina tik x

tekstūros koordinatė, nes *y* koordinates naudojame pastovias: 0 ir 1 kontūro vidui ir išorei. Tekstūros *x* koordinatei yra priskiriama viršūnės *x* ekrano erdvės koordinatė, kai briaunos kryptis ekrane yra labiau horizontali nei vertikali, arba priskiriama viršūnės *y* ekrano erdvės koordinatė priešingu atveju. Viršūnių procesoriui yra paduodama atributų pora <*v, e*>, kur *v* – viršūnės pozicija, o *e* – briaunos kryptis, tai viršūnės poziciją ekrano erdvėje *vs* ir briaunos kryptį ekrano erdvėje *es* galime gauti *v* ir *e* daugindami iš „vaizdo-projekcijos“ matricos VP:

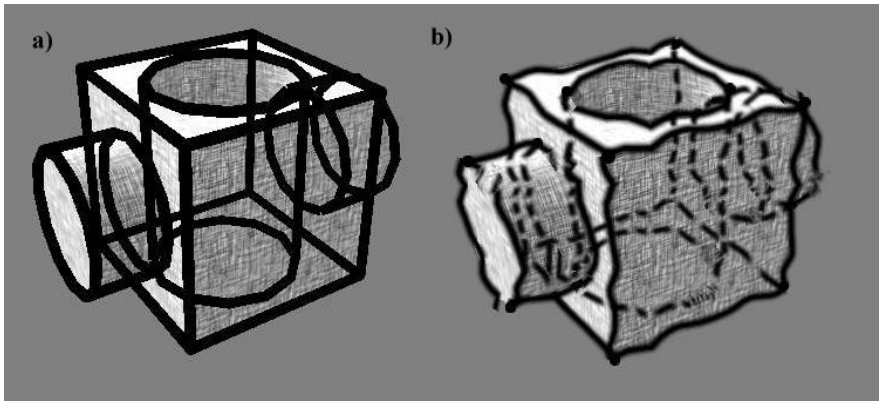
$$vs = v \cdot VP$$

(8)

$$ve = e \cdot VP$$

(9)

Briauna yra labiau horizontali nei vertikali, kai *ve.x* > *ve.y*. Šioje parametrizacijo netolydumai yra ten kur susiduria horizontalios ir vertikalios parametrizacijos briaunos.



17 pav. Kontūrų tekstūravimas: a – netekstūruotas, b – tekstūruotas kontūras

2.1.8. Animuoti objektai

Visi šie skaičiavimai yra pritaikomi ir matricomis animuotiems objektams, tam kiekvienam objektui yra suteikiama „pasaulio“ (angl.: *world*) matrica *W*, kuri apibrėžia objekto poziciją ir orientaciją erdvėje. Norint gauti atributų porą <*v, e*> „pasaulio“ erdvėje, atliekamos dvi transformacijos:

$$vp = v \cdot W$$

(10)

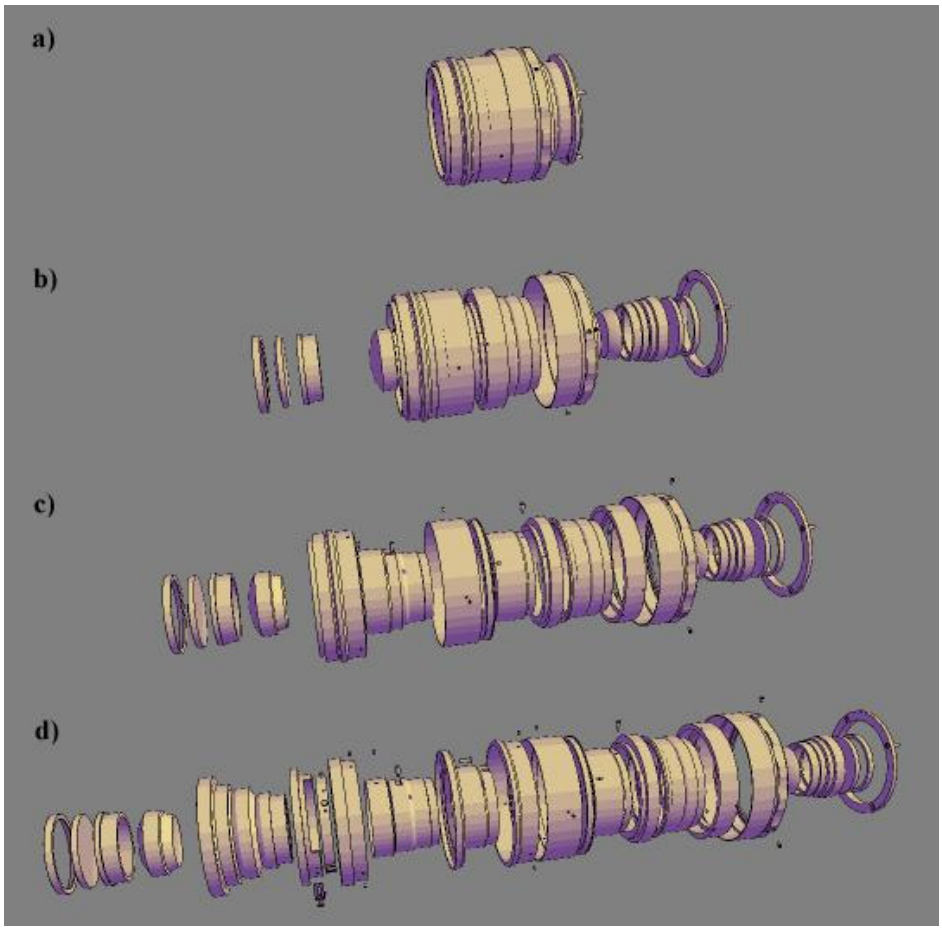
$$ep = e \cdot W$$

(11)

Gauta <*vp, ve*> yra atributų <*v, e*> pora transformuota į pasaulio koordinates. Dažnai pasitaiko, kad atributą reikia transformuoti į „pasaulio“, o po to dar į ekrano erdvę. Tokiu atveju šie

du skaičiavimai sujungiami į vieną prieš paišant objektą pasiruošiant „pasaulio-vaizdo-projekcijos“ matricą *MVP*, tada transformacija atliekama ne dauginant atributą iš dviejų matricų atskirai, o tik iš šios vienos matricos.

18 pav. paveikslėlyje pateiktas animuotų objektų pavyzdys iš demonstracinės programos „Zenit“. Čia pavaizduota fotoaparato „Zenit-E“ objektyvo modelis, kurio detalės yra sujungtos į hierarchinę struktūrą.



18 pav. Fotoaparato objektyvo išardymo animacija

2.2. Atminties naudojimas

Vaizduojant plonas briaunas kiekvienai briaunos viršūnei reikia 24 baitų, iš kurių 12-a perduodami iš centrinio procesoriaus. Vaizduojant storas briaunas ir norint paslėpti aštrius kraštus naudojant viršūnes dengiančius kvadratus, reikia 20-ies baitų kiekvienai kvadrato viršūnei, iš kurių 8-i yra perduodami iš centrinio procesoriaus, taigi sumoje gauname 44-is baitus, iš kurių 20 yra perduodami iš centrinio procesoriaus. Palyginimui objekto vaizdavimui, kuris turi spalvą ir 2D tekstūrų koordinačių poras, t. y. atributų sąrašas <*v, c, n, t0, t1*>, reikia 52 baitų. Objektas turintis *V*

viršūnių turės apie tris kart daugiau briaunų ($E=V*3$). Kiekvienas briaunos stačiakampis ir viršūnės kvadratas turės po 4-ias viršūnes, taigi vaizduojant vien tik briaunas mums reikės apie $24*4*3*V=288*V$ baitų, tai yra apie 5-is kartus daugiau atminties, nei vaizduojant standartinį objektą. Norint paslėpti aštrius kampus reikės papildomų $20*4*V=80*V$ baitų, taigi maksimaliai gali reikėt $368*V$ baitų, t. y. apie 7-is kartus daugiau atminties, nei vaizduojant standartinį objektą.

2.3. Našumas

Šiame darbe pasiūlytas plonų briaunų vaizdavimo algoritmas veikia apie 10-imt kartų lėčiau, nei paišant tą patį objektą su vienu šviesos šaltiniu ir viena tekstūra. Tam yra keletas priežasčių:

1) apdorojamų viršūnių kiekis yra apie 12 kartų didesnis;
2) be briaunų vaizdavimo taip pat atliekamas jų matomumo skaičiavimas;
3) viršūnes apdorojanti programa nėra optimizuota konkrečiam atvejui, o pritaikyta įvairių tipų stilizuotų kontūrų demonstravimui.

Paišant storas briaunas ir paslepiant aštrius kraštus, šis santykis dar suprastėja iki maždaug 12-15. Norint padidinti našumą reiktų optimizuoti viršūnes apdorojančią programą: paišant plonas juodas briaunas vietoj to, kad naudoti juodą tekstūrą reiktų neskaičiuoti tekstūrų koordinačių ir briaunas paišyti vientisa juoda spalva. Norint nepaišyti „paslėptų“ briaunų vietoj to, kad naudoti permatomas tekstūras reiktų pašalinti tą programą kuri atlieka paišymą. Taip pat reiktų naudoti optimizuotus matomų briaunų suradimo metodus.

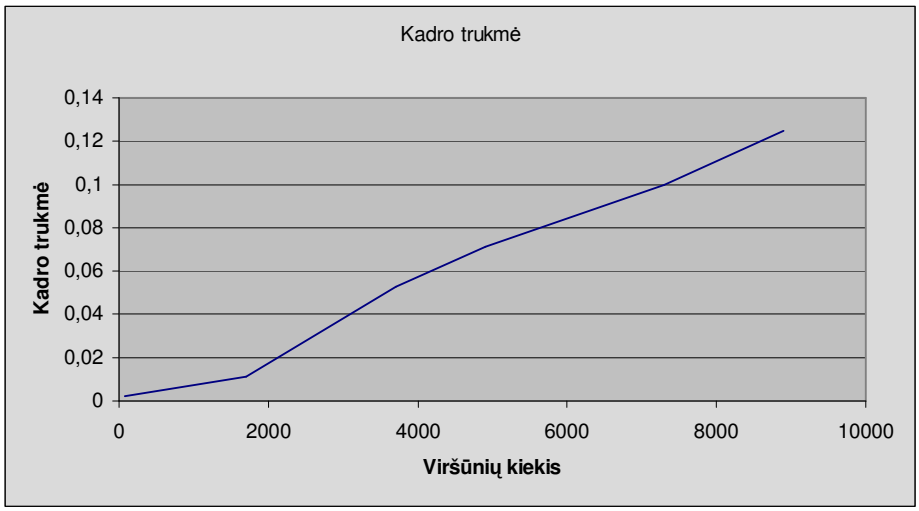
Šio darbo tikslas buvo sumažinti atminties apkrovimą, bei pabandyt padidinti našumą perskirstant skaičiavimus tarp vaizdo plokštės ir centrinio procesoriaus. Atliekant šį perskirtymą buvo atsižvelgta, kuriuos skaičiavimus galima atlikti lygiagrečiai, ir kuriems yra didelis tarpusavio priklausomumas. Lygiagrečius skaičiavimus (t. y. briaunų transformavimą) galima paspartint atliekant kuo daugiau skaičiavimų vienu metu, t. y. didinant viršūnių procesorių kiekį. Kitą dalį galima paspartint naudojat optimizavimo algoritmus. Žinoma šią dalį, kurią mes atliekame centriniame procesoriuje, galima būtų išlygiagretinti, tačiau tokiu atveju padidėtų atminties apkrovimas ir nebūtų galima taikyti optimizavimo algoritmų.

Bandomieji skaičiavimai buvo atlikti naudojant kompiuterį su AMD Athlon 1,33 GHz centriniu procesoriumi ir GeForce 6800 vaizdo plokšte. Rezultatai pateikti 1-oje lentelėje ir 18-oliktame paveikslėlyje.

1 lentelė

Skaičiavimų trukmės priklausomybė nuo viršūnių kiekio

Viršūnių kiekis	80	1700	3700	4900	7300	8900
Kadrų kiekis per sekundę	500	90	19	14	10	8
Kadro trukmė (s)	0,002	0,011111	0,052632	0,071429	0,1	0,125



19 pav. Kadro trukmės priklausomybė nuo viršūnių kiekio

2.4. Galimi patobulinimai

Manoma, kad dabartiniai vaizdo plokščių apribojimai bus panaikinti arba bent jau sušvelninti per tam tikrą laiką. Šiuo metu jau yra vaizdo kortos, kurios leidžia skaityti iš tekstūros viršūnių procesoriuje. Pasinaudojant šia savybe bus galima sutaupyti atminties objekto informaciją surašant į tekstūrą. Taip pat šiuo metu vaizdo plokštės atlieka operacijas tik su slankaus kablelio skaičiais ir neturi operacijų skirtų su bitais sveikuose skaičiuose. Šias operacijas galima pakeisti slankaus kablelio aritmetinėmis operacijomis, tačiau jos veikia lėčiau yra neleistinos paklaidos. Turint tokių operacijų rinkinį būtų galima briaunų matomumo informaciją talpinti po briauną į vieną bitą ir perduoti kaip vaizdo parametrus. Tokiu atveju kiekvienoje viršūnėje būtų įrašyta, kuris bitas nusako tos viršūnės matomumą. Toks matomumo informacijos talpinimas leistų gerokai sumažinti magistralės tarp centrinio procesoriaus ir vaizdo plokštės apkrovimą.

Numatoma, kad vaizdo plokštės 2006 metais leis sukurti ir sunaikinti viršūnes viršūnių procesoriuje ir taip pat leis viršūnių procesoriui rašyti į atmintį. Turint tokias galimybes bus galima sukurti žymiai sudėtingesnius algoritmus, kurie veiks dalinai arba pilnai vaizdo plokštėje.

IŠVADOS

1. Šio darbo metu buvo sukurtos 5-ios demonstracinės programos leidusios susipažinti su metodais ir grafiniais efektais vyraujančiais tarp demonstracinių programų.
2. Kuriamai demonstracinei programai „Zenit“ buvo realizuotas pagrindas – matomų kontūrų nustatymo ir paišymo algoritmas.
3. Pristatyto algoritmo skaičiavimai yra paskirstyti tarp centrinio ir vaizdo plokštės procesorių.
4. Anksčiau pristatytuose metoduose, kurie naudojo tik vaizdo plokštės procesorių, matydavosi aštrūs kampai tarp briaunų. Šią problemą galima išspręsti sujungiant briaunų galus arba paišant apskritimus briaunų susidūrimo vietose, kas ir buvo realizuota kuriant storų briaunų paišymo algoritmą.
5. Plonų briaunų paišymo algoritmas naudoja apie 2 kartus mažiau atminties, nei ankstesniuose tokio tipo metoduose, bei turi panašų našumą, tačiau žymiai labiau apkrauna magistralę tarp centrinio ir vaizdo plokštės procesorių.
6. Joks kontūrų vaizdavimo algoritmas negali būti pripažintas absoliučiai geriausiu, kadangi kiekvieno algoritmo našumas yra optimalus esant tam tikram centrinio ir vaizdo plokštės procesorių ir kanalo tarp jų apkrovimui, todėl realizuojamoje programoje renkantis kontūrų vaizdavimo algoritmą būtina atsižvelgti į numatomus šių komponenčių apkrovimus.
7. Pristatytas algoritmas yra optimaliausias esant apylygiam vaizdo plokštės ir centrinio procesoriaus, bei mažam kanalo tarp jų apkrovimui.

LITERATŪRA

1. GOOCH, B. in Theory and Practice of Non-Photorealistic Graphics: Algorithms, Methods, and Production Systems. SIGGRAPH: tarptautinės konferencijos pranešimų medžiaga, 2003.
2. CARD, D., MITCHELL, J. L. Non-Photorealistic Rendering with Pixel and Vertex Shaders. Iš ShaderX: Vertex andPixel Shaders Tips and Tricks. - Wolfgang Engel, Wordware, 2002.
3. McGUIRE, M., HUGHES, F. J. Hardware-Determined Feature Edges. Non-photorealistic animation and rendering: tarptautinės konferencijos pranešimų medžiaga. Prancūzija, Annency, 2004.
4. LIEKIS, P. Apie DemoSCENE. Iš DemoSCENE.lt [interaktyvus]. 2004 [žiūrėta 2005-04-10]. Prieiga per Internetą: <http://demo.scene.lt/>.
5. SAITO, T., TAKAHASHI, T. Comprehensible rendering of 3d shapes. SIGGRAPH: tarptautinės konferencijos pranešimų medžiaga, 1990, p. 197-206.
6. MITCHELL, J., BRENNAN, C., CARD, D. Real-Time Image-Space Outlining for Non-Photorealistic Rendering. SIGGRAPH: tarptautinės konferencijos pranešimų medžiaga, 2002. Prieiga per Internetą: <http://www.ati.com/developer/SIGGRAPH02/SIGGRAPH2002_Sketch-Mitchell.pdf>.
7. EVERITT, C. One-Pass Silhouette Rendering with GeForce and GeForce2. NVIDIA Corporation, 2000. Prieiga per Internetą: <http://developer.nvidia.com/object/1Pass_Silhouette_Rendering%20.html>.
8. GOOCH, B., SLOAN, P.J., GOOCH, A., ir k. t. Interactive technical illustration. ACM Symposium on Interactive 3D Graphics: tarptautinės konferencijos pranešimų medžiaga, 1999.
9. DIETRICH, S. Cartoon Rendering and Advanced Texture Features of the GeForce 256 Texture Matrix, Projective Textures, Cube Maps, Texture Coordinate Generation and DOTPRODUCT3 Texture Blending. NVIDIA Corporation. Prieiga per Internetą: <http://developer.nvidia.com/object/Cartoon_Rendering_GeForce_256.html>
10. RASKAR, R., COHEN, M.. Image precision silhouette edges. ACM Symposium on Interactive 3D Graphics: tarptautinės konferencijos pranešimų medžiaga, 1999.
11. HALL T., Silhouette Tracking, 2003. Prieiga per Internetą: <http://www.geocities.com/tom_j_hall>.
12. KALNINS, R. D., DAVIDSON, P. L., MARKOSIAN, L., FINKELSTEIN, A. Coherent Stylized Silhouettes. SIGGRAPH: tarptautinės konferencijos pranešimų medžiaga, 2003, p. 856-861.

13. NIENHAUS, M., DOELLNER, J. Edge-Enhancement – An Algorithm for Real-Time Non-Photorealistic Rendering. Žurnalas WSCG. – Čekija, Plenz, 2003, Nr. 1.
14. Homepage of „Nesnausk!“ team [interaktyvus]. 2004 [žiūrėta 2005-04-20]. Prieiga per Internetą: <http://www.nesnausk.org/>.
15. ISENBERG, T., HALPER, N., STROTHOTTE, T. Stylizing Silhouettes at Interactive Rates: From Silhouette Edges to Silhouette Strokes. Eurographics: tarptautinės konferencijos pranešimų medžiaga, 2002.
16. RASKAR, R.. Hardware Support for Non-photorealistic Rendering. SIGGRAPH: tarptautinės konferencijos pranešimų medžiaga, 2001.
17. ISENBERG, T., FREUDENBERG, B., HALPER, ir k. t. A Developer’s Guide to Silhouette Algorithms for Polygonal Models. IEEE Computer Graphics and Applications, 2001.
18. APPEL, A., ROHLF, F., STEIN, A., The Haloed Line Effect for Hidden Line Elimination. SIGGRAPH: tarptautinės konferencijos pranešimų medžiaga, 1979.

TERMINŲ IR SANTRUMPŲ ŽODYNAS

1. Animacinis vaizdavimo stilius – tai kompiuterinėje grafikoje naudojamas vaizdavimo stilius, kuriame vaizdą stengiamasi padaryti panašų į animacinį filmą.
2. DirectX – grafinė biblioteka 3D trimatei grafikai vaizduoti.
3. Fotorealistinis vaizdavimo stilius – vaizdavimo stilius, kai stengiamasi vaizdą padaryti kuo realistiškesnį ir panašesnį į tikrovę (proporcijos, apšvietimas, atspindžiai, šešėliai ir t. t.).
4. Gylio buferis (z-buffer) – buferis kuris naudojamas atstumui nuo objekto iki stebėtojo saugoti.
5. „Minkšti“ šešėliai – šešėliai, kurių kraštai pasibaigia ne staiga, o kinta nuo tamsaus iki šviesaus arba kitaip sakant tai šešėliai, kurių šviesos šaltinis yra netaškinis.
6. Nefotorealistinis vaizdavimo stilius – priešingybė fotorealistiniam vaizdavimo stiliui. Vaizduojant stengiamasi perteikti pagrindines objektų savybes, tačiau gaunamas vaizdas neturi idealiai atitikti tikrojo vaizdo – gali būti pakeistos objektų formos, šešėliai, spalvos ir pan. Pavyzdžiui: animaciniai filmai, techninės iliustracijos, įvairios meno šakos ir k. t.
7. Objekto „pasaulio“ matrica – matrica, kuri nusako objekto poziciją ir orientaciją 3D erdvėje.
8. OpenGL – grafinė biblioteka 3D trimatei grafikai vaizduoti.
9. Pasaulio-vaizdo-projekcijos matrica – matrica gauta iš objekto „pasaulio“, vaizdo ir projekcijos matricų sandaugos.
10. Potepiai – tam tikros tekstūros priskiriamos kontūrams norint išgauti tam tikrą vaizdavimo stilių (pvz.: kreivas, brūkšnines ar anglimi pieštas linijas).
11. Trikampio normalė – normalinis vektorius statmenas trikampio plokštumai.
12. Vaizdo-projekcijos matrica – matrica gauta iš vaizdo (*view*) ir projekcijos (*projection*) matricų sandaugos.
13. 2D – dvimatis (masyvas, vektorius ir pan.).
14. 3D – trimatis (masyvas, vektorius ir pan.).
15. HLSL (High-Level Shader Language) – vaizdą apdorojančių programų kalba.

1 PRIEDAS. Pagrindinės programos teksto fragmentai

Failas „ContouredMesh.h“:

```
#ifndef __CONTOURED_MESH_H
#define __CONTOURED_MESH_H

#include "../MeshEntity.h"
#include "MeshTopology.h"
#include "HalfDynamicMesh.h"
#include "ContourVBFillers.h"
#include "Settings.h"

class CContouredMesh
{
public:
	CContouredMesh( const CSettings& settings );
	~CContouredMesh();

	void render( bool direct = false );

	void directRender( int pass );

	SMatrix4x4& getWorldMatrix() { return mWorld; }

	const CMeshTopology& getMeshTopology() const { return *mMeshTopology; }

private:
	CMeshEntity* mMesh;
	CHalfDynamicMesh<CContourVertexFiller::TVertex1, CContourVertexFiller::TVertex2> mMesh1;
	CHalfDynamicMesh<CContourEdgeFiller::TVertex1, CContourEdgeFiller::TVertex2> mMesh2;
	CMeshTopology* mMeshTopology;
	CContourVertexFiller* mContourVertexFiller;
	CContourEdgeFiller* mContourEdgeFiller;

	SMatrix4x4 mWorld;
};

#endif
```

Failas „ContouredMesh.cpp“:

```
#include "stdafx.h"
#include "ContouredMesh.h"

using namespace dingus;

CContouredMesh::CContouredMesh( const CSettings& settings )
{
	assert( !settings.mAnimated );
	std::string name = settings.mObjectName;

	mWorld.identify();

	mMesh = new CMeshEntity( name );
	mMesh->getRenderMesh( RM_OBJECT )->getParams().addMatrix4x4Ref( "mWorld", mWorld );
	mMesh->getRenderMesh( RM_OBJECT )->getParams().setEffect(
*CEffectBundle::getInstance().getResourceById( settings.mEffect ) );
	if( settings.mObjectTexture.length() > 0 ) mMesh->getRenderMesh( RM_OBJECT )-
>getParams().addTexture( "tBase", *CTextureBundle::getInstance().getResourceById(
settings.mObjectTexture ) );
	mMesh->getRenderMesh( RM_OBJECT )->getParams().addVector4( "color", settings.mColor );


	CMesh* mesh = CMeshBundle::getInstance().getResourceById( name );

	mMeshTopology = new CMeshTopology( *mesh, mWorld, settings );
	mContourVertexFiller = new CContourVertexFiller( *mMeshTopology );
	mContourEdgeFiller = new CContourEdgeFiller( *mMeshTopology );

	CRenderable& r1 = mMesh1.getRenderable();
```

```
        r1.getParams().addMatrix4x4Ref( "mWorld", mWorld );
        r1.getParams().setEffect( *CEffectBundle::getInstance().getResourceById( "vertices" ) );
        r1.getParams().addVector3Ref( "vCamX", G_RENDERCTX-
>getCamera().getCameraRotMatrix().getAxisX() );
        r1.getParams().addVector3Ref( "vCamY", G_RENDERCTX-
>getCamera().getCameraRotMatrix().getAxisY() );
        r1.getParams().addFloat( "fCamFactor", settings.mThickness );
        r1.getParams().addTexture( "tBase", *CTextureBundle::getInstance().getResourceById(
settings.mVertexTexture ) );


        CRenderable& r2 = mMesh2.getRenderable();
        r2.getParams().addMatrix4x4Ref( "mWorld", mWorld );
        r2.getParams().setEffect( *CEffectBundle::getInstance().getResourceById( "edges" ) );
        r2.getParams().addFloat( "fCamFactor", settings.mThickness / 2 );
        r2.getParams().addTexture( "tBase", *CTextureBundle::getInstance().getResourceById(
settings.mEdgeTexture ) );
        r2.getParams().addTexture( "tBase2", *CTextureBundle::getInstance().getResourceById(
settings.mHiddenEdgeTexture ) );

        mMesh1.setVBFiller( *mContourVertexFiller );
        mMesh2.setVBFiller( *mContourEdgeFiller );
}

CContouredMesh::~CContouredMesh()
{
        delete mMesh;
        delete mMeshTopology;
        delete mContourVertexFiller;
        delete mContourEdgeFiller;
}

void CContouredMesh::render( bool direct )
{
        float t = CSystemTimer::getInstance().getTimeS();
        //D3DXMatrixRotationYawPitchRoll( &mWorld, t / 3, 0, 0 );

        SVector3 eyePos = G_RENDERCTX->getCamera().getEye3();
        mMeshTopology->recalculateFlags( eyePos );

        mMesh->render( RM_OBJECT, direct );

        if( direct ) {
                G_RENDERCTX->directRender( mMesh1.getRenderable() );
                G_RENDERCTX->directRender( mMesh2.getRenderable() );
        } else {
                G_RENDERCTX->attach( mMesh1.getRenderable() );
                G_RENDERCTX->attach( mMesh2.getRenderable() );
        }
}

void CContouredMesh::directRender( int pass )
{
        assert( pass >= 1 && pass <= 3 );

        if( pass == 1 ) {
                float t = CSystemTimer::getInstance().getTimeS();
                SVector3 eyePos = G_RENDERCTX->getCamera().getEye3();
                mMeshTopology->recalculateFlags( eyePos );

                mMesh->render( RM_OBJECT, true );
        } else if( pass == 2 ) G_RENDERCTX->directRender( mMesh1.getRenderable() );
        else if( pass == 3 ) G_RENDERCTX->directRender( mMesh2.getRenderable() );
}
```

Failas „ContourVBFillers.h“:

```
#ifndef __CONTOUR_VBFILLERS_H
#define __CONTOUR_VBFILLERS_H

#include <dingus/gfx/Vertices.h>
#include <dingus/math/Vector2.h>
#include "MeshTopology.h"
#include "HalfDynamicMesh.h"
```

```
struct SVertexUV {
	float tu, tv;
};


class CContourVertexFiller : public IVBFiller<SVertexXyz, SVertexUV>
{
public:
	CContourVertexFiller( CMeshTopology& meshTopology );

	// IVBFiller
	virtual int getQuadCount() const;
	virtual CVertexDesc getVertexDescription() const;
	virtual void fill1( TVertex1* vb ) const;
	virtual void fill2( TVertex2* vb ) const;

private:
	CMeshTopology& mMeshTopology;
};


struct SVertexUV2 {
	SVector2 uv1;
	SVector3 uv2;
};

class CContourEdgeFiller : public IVBFiller<SVertexXyz, SVertexUV2>
{
public:
	CContourEdgeFiller( CMeshTopology& meshTopology );

	// IVBFiller
	virtual int getQuadCount() const;
	virtual CVertexDesc getVertexDescription() const;
	virtual void fill1( TVertex1* vb ) const;
	virtual void fill2( TVertex2* vb ) const;

private:
	CMeshTopology& mMeshTopology;
	SVector3 edges[1000];
};

#endif
```

Failas „ContourVBFillers.cpp“:

```
#include "stdafx.h"
#include "ContourVBFillers.h"

using namespace dingus;

CContourVertexFiller::CContourVertexFiller( CMeshTopology& meshTopology )
:	mMeshTopology( meshTopology )
{
}

int CContourVertexFiller::getQuadCount() const
{
	return mMeshTopology.getVertexCount();
}

CVertexDesc CContourVertexFiller::getVertexDescription() const
{
	CVertexDesc vd;
	vd.getStreams().push_back( CVertexStreamDesc( CVertexFormat( CVertexFormat.V_POSITION ) ) );
	vd.getStreams().push_back( CVertexStreamDesc( CVertexFormat( CVertexFormat.V_UV0_2D ) ) );

	return vd;
}

void CContourVertexFiller::fill1( TVertex1* vb ) const
{
```

```
	const SVector3* vertices = mMeshTopology.getVertices();

	for( int i = 0; i < mMeshTopology.getVertexCount(); i++ ) {
		vb->p = vertices[i]; ++vb;
		vb->p = vertices[i]; ++vb;
		vb->p = vertices[i]; ++vb;
		vb->p = vertices[i]; ++vb;
	}
}

void CContourVertexFiller::fill2( TVertex2* vb ) const
{
	int* flags = mMeshTopology.getVertexFlags();

	for( int i = 0; i < mMeshTopology.getVertexCount(); i++ ) {
		int f = flags[i] > 2 ? 1 : 0;

		vb->tu = 0; vb->tv = 0; ++vb;
		vb->tu = f; vb->tv = 0; ++vb;
		vb->tu = f; vb->tv = f; ++vb;
		vb->tu = 0; vb->tv = f; ++vb;
	}
}


CContourEdgeFiller::CContourEdgeFiller( CMeshTopology& meshTopology )
:	mMeshTopology( meshTopology )
{
}

int CContourEdgeFiller::getQuadCount() const
{
	return mMeshTopology.getEdgeCount();
}

CVertexDesc CContourEdgeFiller::getVertexDescription() const
{
	CVertexDesc vd;
	vd.getStreams().push_back( CVertexStreamDesc( CVertexFormat( CVertexFormat.V_POSITION ) ) );
	vd.getStreams().push_back( CVertexStreamDesc( CVertexFormat(
		CVertexFormat.V_UV0_2D | int( CVertexFormat.UV_3D << int( CVertexFormat.UV_BITS +
CVertexFormat.UV_SHIFT ) )
	) ) );

	return vd;
}

void CContourEdgeFiller::fill1( TVertex1* vb ) const
{
	const SVector3* vertices = mMeshTopology.getVertices();
	const CMeshTopology::SIndex2* edgeVertices = mMeshTopology.getEdgeVertices();

	for( int i = 0; i < mMeshTopology.getEdgeCount(); i++ ) {
		SVector3 v1 = vertices[edgeVertices[i].i1];
		SVector3 v2 = vertices[edgeVertices[i].i2];

		vb->p = v1; ++vb;
		vb->p = v1; ++vb;
		vb->p = v2; ++vb;
		vb->p = v2; ++vb;
	}
}

void CContourEdgeFiller::fill2( TVertex2* vb ) const
{
	SVector3 pos = G_RENDERCTX->getCamera().getEye3();

	const SVector3* vertices = mMeshTopology.getVertices();
	const SVector3* vertexEdge = mMeshTopology.getVertexEdges();
	const CMeshTopology::SIndex2* edgeVertices = mMeshTopology.getEdgeVertices();

	bool* flags = mMeshTopology.getEdgeFlags();
	int* vertexFlags = mMeshTopology.getVertexFlags();
```

```
        for( int i = 0; i < mMeshTopology.getEdgeCount(); i++ ) {
                SVector3 v1 = vertices[edgeVertices[i].i1];
                SVector3 v2 = vertices[edgeVertices[i].i2];
                SVector3 edge = v2 - v1;
                SVector3 edge1 = +edge;
                SVector3 edge2 = -edge1;

                //edges[i] = edge;

                int y1 = 0;
                int y2 = 0;

                if( flags[i] == 0 ) edge1 = edge2 = SVector3( 0, 0, 0 );
                else {
                        if( vertexFlags[edgeVertices[i].i1] == 2 ) {
                                edge1 = vertexEdge[edgeVertices[i].i1];
                                if( edge1.dot( edge ) < 0 ) {
                                        edge1 = -edge1;
                                        y1 = 1 - y1;
                                }
                        }

                        if( vertexFlags[edgeVertices[i].i2] == 2 ) {
                                edge2 = vertexEdge[edgeVertices[i].i2];
                                if( edge2.dot( edge ) > 0 ) {
                                        edge2 = -edge2;

                                } else y2 = 1 - y2;
                        }
                }

                vb->uv2 = +edge1; vb->uv1.x = 0; vb->uv1.y = y1; ++vb;
                vb->uv2 = -edge1; vb->uv1.x = 0; vb->uv1.y = 1 - y1; ++vb;
                vb->uv2 = +edge2; vb->uv1.x = 1; vb->uv1.y = 1 - y2; ++vb;
                vb->uv2 = -edge2; vb->uv1.x = 1; vb->uv1.y = y2; ++vb;
        }
}
```

Failas „MeshTopology.h“:

```
#ifndef __MESH_TOPOLOGY_H
#define __MESH_TOPOLOGY_H

#include <dingus/math/Vector3.h>
#include "Settings.h"

namespace dingus {
        struct SVertexXyzNormal;
        class CMesh;
};

#define VERTEX_COUNT_PER_INT 10

class CMeshTopology {
public:
        struct SIndex3 {
                int i1, i2, i3;
        };

        struct SIndex2 {
                int i1, i2;
        };

public:
        explicit CMeshTopology( CMesh& mesh, SMatrix4x4& world, const CSettings& settings );
        ~CMeshTopology();

        int getEdgeCount() const { return mEdgeCount; }
        const SIndex2* getEdgeVertices() const { return mEdgeVertices; }
        bool* getEdgeFlags() { return mEdgeFlags;  }

        int getVertexCount() const { return mVertexCount; }
        const SVector3* getVertices() { return mVertices; }
        const SVector3* getVertexEdges() { return mVertexEdges; }
```

```
        int* getVertexFlags() { return mVertexFlags; }

        void recalculateFlags( const SVector3& pos );

private:
        bool hasChanged( const SVector3& pos ) const;

        SVertexXyzNormal getVertex( int index, const void* vb, int vertexStride );
        void addEdge( int triIndex, int vi1, int vi2 );
        void replaceVertex( SIndex3* triangleVertices, int visrc, int vidst );



private:
        CSettings mSettings;

        SMatrix4x4& mWorld;
        SVector3 mLastPosition;

        SVector3* mVertices;
        SVector3* mVertexEdges;
        SVector3* mNormals;
        SVector3* mNormalPositions;

        bool* mNormalFlags;
        bool* mEdgeFlags;
        char* mMarkedEdges;
        int* mVertexFlags;

        SIndex2* mEdgeNormals;
        SIndex2* mEdgeVertices;


        int mTriangleCount;
        int mEdgeCount;
        int mVertexCount;
};

#endif
```

Failas „MeshTopology.cpp“:

```
#include "stdafx.h"
#include "MeshTopology.h"

#include <dingus/gfx/Vertices.h>
#include <dingus/gfx/Mesh.h>



CMeshTopology::CMeshTopology( CMesh& mesh, SMatrix4x4& world, const CSettings& settings )
:       mLastPosition( 1e10, 0, 0 ), mWorld( world ), mSettings( settings )
{
        int i, j;

        const void* vb = mesh.lockVBRead();
        const void* ib = mesh.lockIBRead();

        int vertexStride = mesh.getVertexStride();

        mTriangleCount = mesh.getIndexCount() / 3;
        SIndex3* triangleVertices = new SIndex3[mTriangleCount];
        mNormals = new SVector3[mTriangleCount];
        mNormalPositions = new SVector3[mTriangleCount];
        mNormalFlags = new bool[mTriangleCount];

        for( i = 0; i < mTriangleCount; i++ ) {
                mesh.getTriIndices( ib, i, triangleVertices[i].i1, triangleVertices[i].i2,
triangleVertices[i].i3 );

                SVertexXyzNormal v1 = getVertex( triangleVertices[i].i1, vb, vertexStride );
                SVertexXyzNormal v2 = getVertex( triangleVertices[i].i2, vb, vertexStride );
                SVertexXyzNormal v3 = getVertex( triangleVertices[i].i3, vb, vertexStride );
```

```
                mNormals[i] = v1.n + v2.n + v3.n;
                mNormals[i].normalize();

                mNormalPositions[i] = ( v1.p + v2.p + v3.p ) / 3;
        }

        mVertexCount = mesh.getVertexCount();
        mVertices = new SVector3[mVertexCount];
        mVertexEdges = new SVector3[mVertexCount];
        for( i = 0; i < mVertexCount; i++ )
                mVertices[i] = getVertex( i, vb, vertexStride ).p;

        const float MIN_DIST = 1e-6f;

        for( i = 0; i < mVertexCount; i++ ) {
                for( j = i + 1; j < mVertexCount; j++ )
                        if( SVector3( mVertices[i] - mVertices[j] ).length() < MIN_DIST ) {
                                replaceVertex( triangleVertices, j, i );
                                replaceVertex( triangleVertices, mVertexCount - 1, j );
                                mVertices[j] = mVertices[mVertexCount - 1];
                                --mVertexCount;
                                j--;
                        }
        }

        mVertexFlags = new int[mVertexCount];

        mEdgeVertices = new SIndex2[mTriangleCount * 3];
        mEdgeNormals = new SIndex2[mTriangleCount * 3];
        for( i = 0; i < mTriangleCount * 3; i++ )
                mEdgeNormals[i].i1 = mEdgeNormals[i].i2 = -1;

        mEdgeCount = 0;

        for( i = 0; i < mTriangleCount; i++ ) {
                addEdge( i, triangleVertices[i].i1, triangleVertices[i].i2 );
                addEdge( i, triangleVertices[i].i2, triangleVertices[i].i3 );
                addEdge( i, triangleVertices[i].i3, triangleVertices[i].i1 );
        }

        mEdgeFlags = new bool[mEdgeCount];
        mMarkedEdges = new char[mEdgeCount];

        for( i = 0; i < mEdgeCount; i++ ) {
                assert( mEdgeNormals[i].i1 >= 0 && mEdgeNormals[i].i2 >= 0 );

                mMarkedEdges[i] = 0;

                float d = mNormals[mEdgeNormals[i].i1].dot( mNormals[mEdgeNormals[i].i2] );
                //mMarkedEdges[i] = d > -0.5f && d < 0.5f;

                SVector3 v = ( mNormalPositions[mEdgeNormals[i].i1] +
mNormalPositions[mEdgeNormals[i].i2] ) / 2;
                SVector3 n = ( mNormals[mEdgeNormals[i].i1] + mNormals[mEdgeNormals[i].i2] ) / 2;
                SVector3 n2 = v - ( mVertices[mEdgeVertices[i].i1] + mVertices[mEdgeVertices[i].i2] )
/ 2;

                n.normalize();
                n2.normalize();

                float d2 = n.dot( n2 );

                if( d2 < -0.001f ) { // ridge
                        if( d < 0.8f && mSettings.mShowRidges ) mMarkedEdges[i] = 1;
                } else if( d2 > 0.001f ) { // walley
                        if( d < 0.8f && mSettings.mShowWalleys ) mMarkedEdges[i] = 2;
                }


        }


        delete[] triangleVertices;
```

```
	mesh.unlockIBRead();
	mesh.unlockVBRead();
}

CMeshTopology::~CMeshTopology()
{
	delete[] mNormals;
	delete[] mNormalPositions;
	delete[] mVertices;
	delete[] mVertexEdges;

	delete[] mNormalFlags;
	delete[] mEdgeFlags;
	delete[] mMarkedEdges;
	delete[] mVertexFlags;

	delete[] mEdgeVertices;
	delete[] mEdgeNormals;
}

SVertexXyzNormal CMeshTopology::getVertex( int index, const void* vb, int vertexStride ) {
	return *(const SVertexXyzNormal*)( (const char*)vb + vertexStride * index );
}

void CMeshTopology::addEdge( int triIndex, int vi1, int vi2 ) {
	if( vi1 > vi2 ) std::swap( vi1, vi2 );

	for( int i = 0; i < mEdgeCount; i++ )
		if( vi1 == mEdgeVertices[i].i1 && vi2 == mEdgeVertices[i].i2 ) break;

	if( i == mEdgeCount ) {
		mEdgeVertices[i].i1 = vi1;
		mEdgeVertices[i].i2 = vi2;
		mEdgeCount++;
	}

	if( mEdgeNormals[i].i1 < 0 ) mEdgeNormals[i].i1 = triIndex;
	else {
		assert( mEdgeNormals[i].i2 < 0 );
		mEdgeNormals[i].i2 = triIndex;
	}
}

void CMeshTopology::replaceVertex( SIndex3* triangleVertices, int visrc, int vidst )
{
	for( int i = 0; i < mTriangleCount; i++ ) {
		if( triangleVertices[i].i1 == visrc ) triangleVertices[i].i1 = vidst;
		if( triangleVertices[i].i2 == visrc ) triangleVertices[i].i2 = vidst;
		if( triangleVertices[i].i3 == visrc ) triangleVertices[i].i3 = vidst;
	}
}

bool CMeshTopology::hasChanged( const SVector3& pos ) const
{
	return SVector3( mLastPosition - pos ).length() > 1e-4;
}

void CMeshTopology::recalculateFlags( const SVector3& worldPos ) {
	SVector3 pos;
	SMatrix4x4 inverseMatrix;
	D3DXMatrixInverse( &inverseMatrix, &pos.x, &mWorld );
	D3DXVec3TransformCoord( &pos, &worldPos, &inverseMatrix );

	if( hasChanged( pos ) ) {
		mLastPosition = pos;

		int i;

		for( i = 0; i < mTriangleCount; i++ ) {
			SVector3 dir = mNormalPositions[i] - pos;

			mNormalFlags[i] = mNormals[i].dot( dir ) < 0;
		}
```

```
                memset( mVertexEdges, 0, sizeof( SVector3 ) * mVertexCount );
                memset( mVertexFlags, 0, sizeof( int ) * mVertexCount );
                memset( mEdgeFlags, false, sizeof( bool ) * mEdgeCount );

                for( i = 0; i < mEdgeCount; i++ )
                        if( mMarkedEdges[i] > 0 || ( mSettings.mShowContours &&
                                ( mNormalFlags[mEdgeNormals[i].i1] &&
!mNormalFlags[mEdgeNormals[i].i2] ||
                                  !mNormalFlags[mEdgeNormals[i].i1] &&
mNormalFlags[mEdgeNormals[i].i2] ) ) )
                        {
                                mEdgeFlags[i] = true;

                                SVector3 edge = mVertices[mEdgeVertices[i].i1] -
mVertices[mEdgeVertices[i].i2];
                                edge.normalize();

                                if( mVertexFlags[mEdgeVertices[i].i1] == 0 )
                                        mVertexEdges[mEdgeVertices[i].i1] -= edge;
                                else mVertexEdges[mEdgeVertices[i].i1] += edge;

                                if( mVertexFlags[mEdgeVertices[i].i2] == 0 )
                                        mVertexEdges[mEdgeVertices[i].i2] += edge;
                                else mVertexEdges[mEdgeVertices[i].i2] -= edge;

                                mVertexFlags[mEdgeVertices[i].i1]++;
                                mVertexFlags[mEdgeVertices[i].i2]++;
                        }
        }
}
```

2 PRIEDAS. Vaizdo informaciją apdorojančių programų teksto fragmentai

Failas „Edges.fx “:

```
#include "lib/shared.fx"
#include "lib/structs.fx"

float4x4 mWorld;

float fCamFactor;

struct SPosTexN {
	float4 pos	: POSITION;
	float2 uv1	: TEXCOORD0;
	float3 uv2	: TEXCOORD1;
};

texture tBase;
texture tBase2;

SPosTex vsMain( SPosTexN i ) {
	float4 pos = mul( i.pos, mWorld );

	SPosTex o;

	float3 v = mul( i.uv2, mWorld );
	float3 dir = pos - vEye;
	float3 n = cross( v, dir );
	n = normalize( n );

	o.pos = pos + float4( n * fCamFactor, 0 );

	dir = normalize( dir );
	o.pos.xyz = o.pos.xyz - dir * fCamFactor * 2;

	o.pos = mul( o.pos, mViewProj );

	float3 ns = mul( n, (float3x3)mView );
	ns = mul( float4( ns, 1 ), mProjection );

	float3 spos = mul( i.pos, mViewProj );

	if( abs( ns.x ) > abs( ns.y ) ) {
		o.uv.x = spos.y * 0.5;
	} else {
		o.uv.x = spos.x * 0.5;
	}

	if( ns.x > ns.y ) {
		o.uv.y = 1;
	} else {
		o.uv.y = 0;
	}

	return o;
}

technique tecFFP {
	pass PAlphaBack {
		VertexShader = compile vs_1_1 vsMain();
		PixelShader = NULL;

		ZWriteEnable = False;
		ZFunc = Greater;

		CullMode = None;
		AlphaBlendEnable = True;
		SrcBlend = SrcAlpha;
		DestBlend = InvSrcAlpha;

		Texture[0] = <tBase2>;
```

```
                ColorOp[0]   = SelectArg1;
                ColorArg1[0] = Texture;
                AlphaOp[0]   = SelectArg1;
                AlphaArg1[0] = Texture;

                ColorOp[1]   = Disable;
                AlphaOp[1]   = Disable;
        }

        pass PFront {
                VertexShader = compile vs_1_1 vsMain();
                PixelShader = NULL;

                AlphaTestEnable = True;
                AlphaFunc = Greater;
                AlphaRef = 220;

                ZWriteEnable = True;
                ZFunc = LessEqual;

                Texture[0] = <tBase>;
        }

        pass PAlpha {
                VertexShader = compile vs_1_1 vsMain();
                PixelShader = NULL;

                AlphaTestEnable = False;
                ZWriteEnable = False;
        }

        pass PLast {
                AlphaTestEnable = False;
                AlphaBlendEnable = False;
                ZWriteEnable = True;
                ZFunc = LessEqual;
        }
}
```

Failas „Vertices.fx “:

```
#include "lib/shared.fx"
#include "lib/structs.fx"

float3 vCamX;
float3 vCamY;

float4x4 mWorld;

float fCamFactor;

texture tBase;

SPosTex vsMain( SPosTex i ) {
        float4 pos = mul( i.pos, mWorld );

        SPosTex o;

        o.pos = (
                ( i.uv.x - 0.5 ) * float4( vCamX, 0 ) +
                ( i.uv.y - 0.5 ) * float4( vCamY, 0 )
        ) * fCamFactor;
        o.pos = pos + o.pos;

        float3 dir = normalize( pos - vEye );
        o.pos.xyz = o.pos.xyz - dir * fCamFactor * 2;

        o.pos = mul( o.pos, mViewProj );

        o.uv = i.uv;

        return o;
}
```

```
technique tecFFP {
	pass POpaque {
		VertexShader = compile vs_1_1 vsMain();
		PixelShader = NULL;

		CullMode = None;

		AlphaTestEnable = True;
		AlphaFunc = Greater;
		AlphaRef = 220;

		Texture[0] = <tBase>;

		ColorOp[0]      = SelectArg1;
		ColorArg1[0] = Texture;
		AlphaOp[0]      = SelectArg1;
		AlphaArg1[0] = Texture;

		ColorOp[1]      = Disable;
		AlphaOp[1]      = Disable;
	}

	pass PAlpha {
		VertexShader = compile vs_1_1 vsMain();
		PixelShader = NULL;

		AlphaBlendEnable = True;
		SrcBlend = SrcAlpha;
		DestBlend = InvSrcAlpha;

		AlphaTestEnable = False;
		ZWriteEnable = False;
	}

	pass PLast {
		AlphaBlendEnable = False;
		ZWriteEnable = True;
	}
}
```