LATVIJAS UNIVERSITĀTE
DATORIKAS FAKULTĀTE

TEKSTŪRAS KOORDINĀTU ĢENERĒŠANAS AGLORITMI 3D MODEĻIEM

BAKALAURA DARBS

Autors: **Pāvels Jacuks**
Studenta apliecības Nr.: 13005
Darba vadītājs: doc. Kārlis Freivalds

RĪGA 2017

ANOTĀCIJA

Darba autors skaidro tekstūras koordinātu ģenerēšanas procesus OpenGL API un izskata ar to izstrādes saistītos jautājumus. Tekstūras koordinātes izmanto, lai sasaistītu telpisku ģeometriju ar digitālo bildi.

Autora mērķis ir piedāvāt tekstūras uzklāšanas un tekstūras koordinātu ģenerēšanas risinājumus ar OpenGL API, kas darbotos uz dažādiem objektiem.

Atslēgas vārdi: OpenGL; Tekstūras koordinātu ģenerēšana; Tekstūras uzklāšana;

ABSTRACT

TEXTURE COORDINATE GENERATION OF 3D MODELS

In this thesis the author explains process of texture coordinate generation in OpenGL API and reviews questions related to it. Texture coordinates are used to link 3D geometry with digital image.

The author's goal is to offer solution for texture mapping and texture coordinate generation that works on different objects with OpenGL.

Keywords: OpenGL; texgen; texture mapping;

SATURA VADĪTĀJS

IEVADS

Bakalaura darba ievadā tiek izskaidrots, kas ir tekstūras uzklāšana un tekstūras koordinātu ģenerēšana, kā arī darba autora mērķi un motivācija darba veikšanai.

1.1. Tekstūra

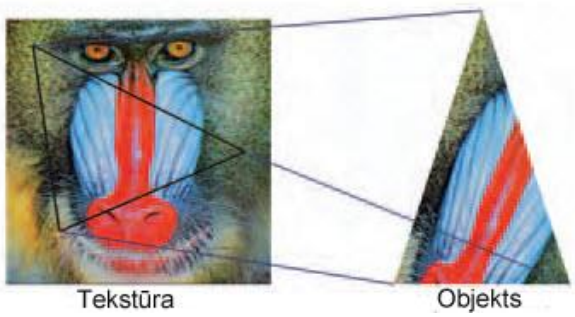
Lietojumprogrammā, kura attēlo telpiskus objektus, tekstūras izmantošana ir viena no visbūtiskākajām metodēm, lai kontrolētu objekta izskatu. Visizplātītākā tekstūras pielietošana ir nodrošināt objekta virsmas apkrāsojumu, modificējot virsmas krāsu uz katru pikseli. Šai apkrāsošanai kā pamatu izmanto digitālu bildi, skat. *att 1.1.1.*

att. 1.1.1. Kubs ar uzklātu tekstūru

lai lietojumprogrammas kods spētu pareizi uzklāt tekstūru, uzklāšanai ir nepieciešama saikne starp tekstūru un objekta virsmu. Šo saikni sauc par tekstūras koordinātēm.

1.2. Tekstūras koordinātes.

Tekstūras koordinātas, ko sauc arī par s, t, ir ciparu pāri, kuri tiek glabāti ģeometrijas virsotnēs. Šie vērtību pāri tiek izmantoti, lai pielāgotu plakanu 2D tekstūru uz telpiska objekta skat *att. 1.2.1.*

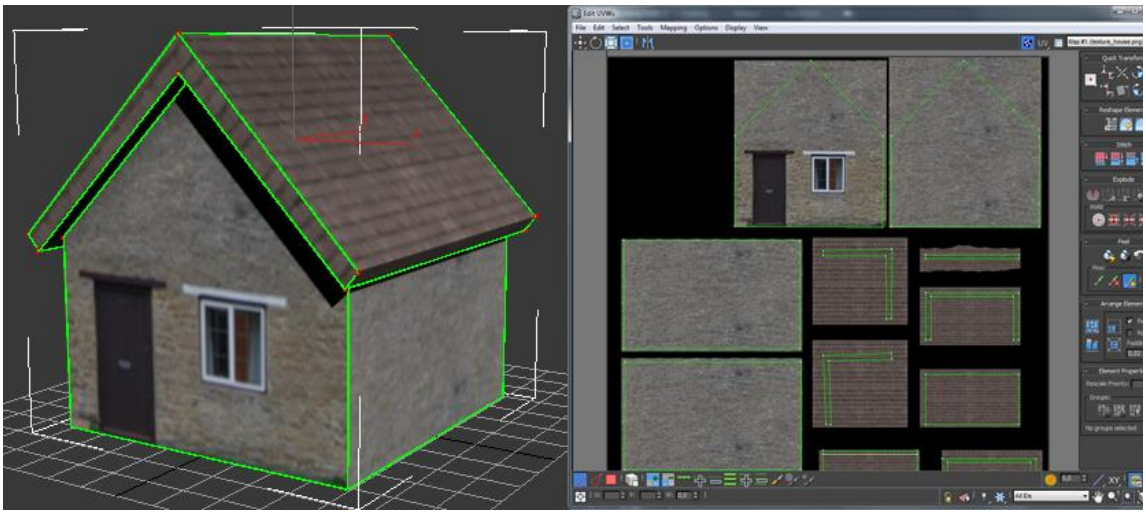


att. 1.2.1. Tekstūras koordināšu demonstrācija

Tekstūras koordinātes saista pozīcijas uz tekstūras attēla ar objekta virsotnes pozīcijām. Koordinātu noteikšana katrai objekta virsotnei nodrošina šī objekta pilnu apkrāsošanu.
Taču individuāli norādīt tekstūras koordinātes objektiem ar lielu virsotņu skaitu ir neiespējams bez tekstūras koordinātu ģenerēšanas.

1.3. Tekstūras koordināšu ģenerēšana.

Visbiežāk, lai uzstādītu tekstūras koordinātes, izmanto modeļu izstrādes programmatūru kā *3ds Max* vai *Blender skat. att. 1.3.1.* Šāda programmatūrā piedāvā ērtus rīkus koordinātu ģenerēšanai. Eksportējot objektus dažādos formātos, koordinātes tiek saglabātas.



att. 1.3.1. Tekstūras koordināšu uzstādīšana izmantojot 3ds Max.

Tomēr ir tādi gadījumi, kuros nevar izmantot modeļu izstrādes programmatūru, lai uzstādītu tekstūras koordinātes. Piemērām, gadījumos, kad objekts tiek izveidots pa taisno paštaisītajā lietojumprogrammā. Tieši par tādiem gadījumiem ir vislielākais uzsvars šajā bak. darbā.

1.4. Mērķi

Šī bak. darba mērķis ir piedāvāt tekstūras uzklāšanas un tekstūras koordinātu ģenerēšanas risinājumus, kas darbotos uz dažādiem objektiem paštaisītajā lietojumprogrammā. Šī darba uzdevumi ir ne tikai nodrošināt tekstūras koordinātes telpiskiem objektiem, bet arī nodrošināt to, lai uzklātās tekstūras tiek vizualizētas.

Mērķa īstenošanai autors izvēlējās OpenGL platformu. OpenGL (*Open Graphics Library*) ir datorikas nozares standarta lietojumprogrammas saskarne (*API*), kas paredzēta 2D un 3D grafikas attēlošanai.

1.5. Motivācija

Nākotnē, darba autors vēlas izveidot lietojumprogrammu, kas proceduāri ģenerē dažādus mošķus kā spēlē „*Spore", skat. att. 1.5.1,* un izmantot šo projektu kā maģistra darbu.

att. 1.3.1. Spēles *„Spore"* Proceduāri ģenerēti mošķi

Viens no lielākajiem soļiem, lai to īstenot, ir atrast risinājumus precīzai tekstūras uzklāšanai un automātiskai koordinātu ģenerēšanai.

TERMINI UN SVEŠVĀRDI

OpenGL	OpenGL (*Open Graphics Library*) ir datorikas nozares standarta lietojumprogrammas saskarne (*API*), kas paredzēta 2D un 3D grafikas attēlošanai.
API	Lietojumprogrammas saskarne.
pipeline	Datu apstrādes elementu kopa, kas ir savienota virknē, kurā viena elementa rezultāts ir nākamās paaudzes ievades elements.
object space	OpenGL 3D transformācijās objekta telpa ir tā koordinātes attiecībā pret tā lokālo izcelsmi.
eye space	OpenGL 3D transformācijās redzamības telpas koordinātes ir attēlotas no kameras vai skatītāja skatpunkta.
s,t,r,q	OpenGL tekstūras koordinātes. DirectX tās tiek sauktas par U, V, Q.
modelview	Objekta matricas un skata matricas konkatenācija.
enviroment mapping	OpenGL apkārtējās vides uzklāšana uz objekta virsmas.
texel	Tekselis, vērtība vienam no elementiem masīvā, kas satur digitāla attēla datus.
texgen	Tekstūras koordinātu ģenerēšana.
antialiasing	Kropļojumnovēršanas filtrs.
SOIL	*Simple OpenGL Image Library* ir bibliotēka, ar kuras palīdzību attēla faili tiek ielādēti un pārveidoti par tekstūras objektiem.
cubemap	Apvienota tekstūra, kas sastāv no sešiem atsevišķiem attēliem.

1. IEVADS OPENGL 3D GRAFIKAS APSTRĀDES PROCESSIEM

Šajā nodaļā ir aprakstītas OpenGL transformācijas telpiskai ģeometrijai un tekstūrām, kā arī sniedz ieskatu tam, kuros transformāciju posmos tiek ģenerētas tekstūras koordinātes.

1.1 Pārskats OpenGL 3D transformācijām

Pēc Tom McReynold domām [2] 19. lpp., OpenGL iekļauj sevī vienkāršu un tajā pašā laikā jaudīgu transformācijās modeli. Virsotnes satur informāciju ne tikai par savu atrašanas vietu, bet arī par normāles stāvokli, virzienu, un tekstūras koordinātēm. Šīs vērtības tiek manipulētas ar transformāciju virkni (pozīciju lineārā kombinācija, rotācija, mērogošana un nobīdes). OpenGL specifikācijā [1] 41. lpp. ir norādīts, ka šo transformāciju būtiskā reprezentācija iekš OpenGL ir 4×4 matrica. OpenGL kontrolētās transformācijas, kopā ar perspektīvas dalīšanas funkcionalitāti ir pieejama gan pozicionālai gan tekstūru koordinātu sistēmai un sniedz būtisku kontroli lietojumprogrammā.

OpenGL transformation pipeline vai OpenGL transformāciju sistēma pārveido lietojumprogrammas virsotņu datus uz ekrāna koordinātēm, kur tie ir gatavi rasterizācijai. Kā jau visas 3D grafikas sistēmas, OpenGL izmanto lineāro algebru – atpazīst virsotnes kā vektorus un pārveido tos izmantojot vektoru-matricu reizināšanu, Tom McReynold [2] 20. lpp. Transformācijas process tiek saukts par *pipeline*, jo ģeometrijas dati tiek caurlaisti starp vairākām koordinātu sistēmām pirms tie sasniedz ekrāna koordinātu sistēmu. Katra no koordinātu sistēmām kalpo vienai vai vairākām OpenGL funkcijām.

OpenGL specifikācijā [1] 41. lpp. ir norādīts, ka tiek izmantotas piecas koordinātu sistēmas jeb telpas: *object space* vai objektu telpa, kas sāk darbību ar lietojumprogrammas koordinātēm, *eye space* vai redzamā telpa, kur scēna tiek sakārtota, *clip space, NDC space* vai normalizēta telpa un ekrāna telpa, kura iestata kadra bufera pikseļus.

1.2 Objekta telpa un *modelview* matrica

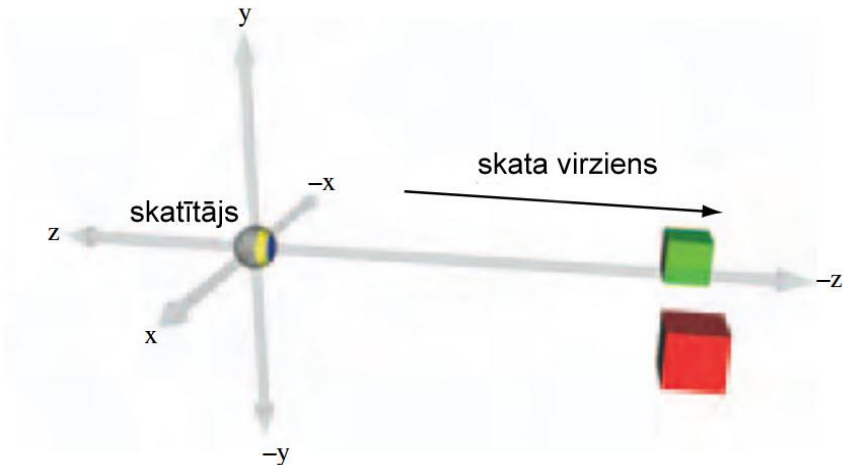
Sākotnējā stāvoklī no lietojumprogrammas tiek iesūtītas tekstūras, virsotņu un gaismas pozīcijas koordinātes, kā arī normāles vektori. Tiek uzskatīts, ka šīs netransformētās vērtības atrodas objekta telpā. Šajā telpā eksistē tikai modelis, kurā visi dati atsaucās tikai uz šī modeļa enkuru – ja uz scēnas atrodas daudz objektu, tad,

izmantojot tikai objekta telpas datus, nevarēs attēlot visus objektus vienā saskaņotā telpa. Ja lietojumprogrammā ir iestatīta, objektu tekstūras kodinātu ģenerēšanas opcija Objektu telpā, tad šajā etapā tie tiek ģenerēti no netransformētiem datiem. Tātad, lai pārveidotu objekta telpas koordinātes uz *eye space* - redzamības telpas koordinātēm, OpenGL transformē objekta telpu ar šī kadra *modelview* matricu – tas ir vajadzīgs, lai sakārtotu virkni ar dažādiem objektiem vienā saskaņotā telpā, kura attēlota no konkrēta skatupunkta. *Modelview* transformācija veic gan modeļu, gan skata transformācijas. Modeļu transformācija novieto un orientē objektus. Bieži vien katram objektam ir vajadzīga sava transformācija, lai to korekti novietot, tāpēc tā tiek pielietota objektam pēc objekta līdz to vairs nav. Taču skata transformācija atpazīst virkni ar objektiem kā kopīgu grupu, tā orientē visus objektus pret kameras atrašanas vietu scēnā. Transformācija mainīsies tikai tad, kad mainās kameras atrašanas vieta. Rezultātā, šī transformēta telpa tiek uzskatīta par redzamo telpu.

1.3 *Eye space* un projekcijas matrica.

Tekstūras koordinātu ģenerēšana un gaismas funkcionalitāte notiek redzamās telpas etapā. OpenGL redzamā telpa ir definēta šādi:

- Šī koordinātu sistēma tiek bāzēta no kameras pozīcijas skatpunkta.
- Skata virziens ir vērsts uz negatīvo z asi.
- Pozitīvais y ir uz augšu.
- Pozitīvais x ir vērsts uz labo kameras pusi.



att. 1.3.1 **redzamā telpa**

Lai vajadzīgā ģeometrija attēlotos, tai ir jāatrodas kaut kur uz redzamās telpas negatīvās z ases, jo ģeometrija aiz skatupunkta, uz pozitīvās z ases, netiek attēlota, *att 1.3.1*.

Šajā etapā *Normals* tiek izmantots gaismas rēķināšanai kopā ar kameras un gaismas avotu pozīcijām, rezultātā – modificējot esošo virsotņu krāsu. Tālāk notiek projekcijas transformācija, kas pārveido atlikušās virsotnes un tekstūru koordinātes uz *clip space*..

1.4 *Clip space* un perspektīvas dalīšana.

Šī koordināšu sistēma nosaka ģeometrijas attēlošanas loku. Citiem vārdiem – tie objekti, vai to daļas, kas atrodas ārpus *clip space* noteiktās zonas, netiek zīmēti.

$$-wclip \leq xclip \leq wclip$$
$$-wclip \leq yclip \leq wclip$$
$$-wclip \leq zclip \leq wclip$$

Perspektīvas dalīšanas process pēc būtības samazina tālāko ģeometriju un palielina to, kas atrodas tuvāk kameras pozīcijai.

1.5 *NDC space*.

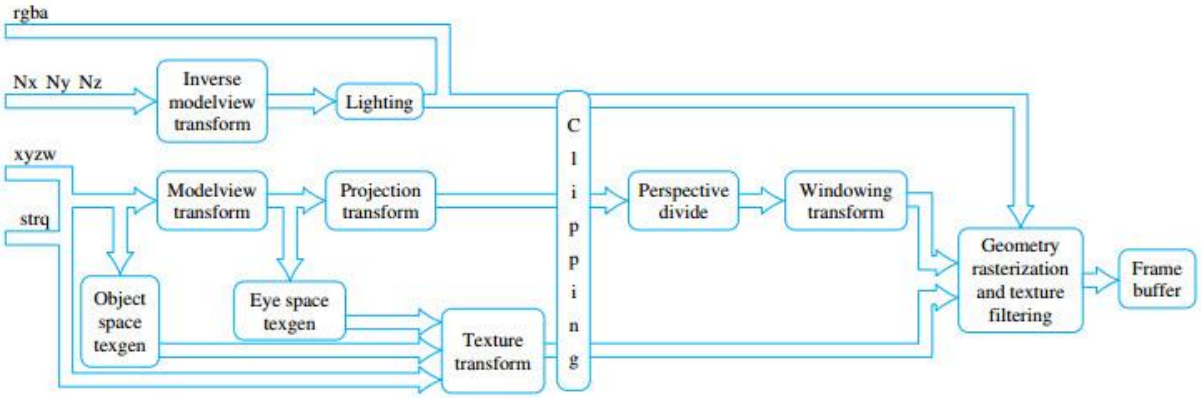
Normalized Device Coordinate space. Šajā koordinātu sistēmā virsotņu dati tiek normalizēti – virsotņu x, y un z vērtības tiek pārvērstas 0.0 - 1.0 diapazonā. OpenGL neizpilda papildus funkcijas šajā normalizētajā telpā. Tā ir koordinātu sistēma, kas pastāv starp perspektīvas dalīšanu un ekrāna telpu.

1.6 Ekrāna telpa.

Šī telpa uzstāda ģeometriju uz pikseļu pozīcijām kadru buferī, lai to varētu attēlot uz ekrāna. Ekrāna koordinātu x y sākums atrodas ekrāna apakšējā kreisajā stūrī. Koordinātu z vērtība izvēršas dziļuma buferī un katra z koordināte ir mērogota tā, lai tā būtu 0 (viss tuvāk skatītājam) līdz 1(tālākais no skatītāja) diapazonā.

1.7 TEKSTŪRAS KOORDINĀTU TRANSFORMĀCIJAS *PIPELINE*

Tekstūras koordinātēm ir sava transformāciju sistēma skat. *att 2.1.*



att.2.1. **Tekstūru koordinātu transformācijas sistēma**

Visbiežāk 3D modelim, kas tiek sūtīts zīmēšanai, ir jau uzstādītas tekstūras koordinātes, piemēram, eksportējot no 3ds Max. Protams, ir gadījumi, kad to nav un tiek ģenerēti automātiski ar OpenGL, bet abos gadījumos tekstūras koordinātes ir transformētas ar 4×4 tekstūru transformācijas matricu. Līdzīgi kā virsotņu koordinātēm, tekstūras koordinātēm vienmēr ir četras komponentes – s, t, r, q, kur s, t, r ir ekvivalentas x, y, z. Komponentes r un q parasti netiek izmantotas, piemēram, komponente r ir priekš 3D tekstūrām, tādos gadījumos šo komponentu vietā tiek rakstīta noklusējuma vērtība 0 priekš r un 1 priekš q.

1.8 Tekstūras Transformācijas Matrica

No sākuma, šī darba autoram rādījās nesaprašana par tekstūras matricas lomu; Kāpēc tā ir vajadzīga? Bet atklājās, ka tāda tekstūras matrica eksistē, jo ne visi pielieto standarta tekstūras uz standarta virsmas [4]. Tekstūras matricu izmanto, lai nobīdītu tekstūras koordinātes, kas rezultātā pārbīda tekstūru pāri modelim. To var izmantot arī priekš tekstūras rotācijas, neaizskarot to ģeometriju, pie kuras tekstūra ir piestiprināta. Tekstūras matrica eksistē, jo tā ir noderīga (bet ne tik ļoti bieži), lai pārveidotu tekstūras koordinātes, pirms tās piekļūst pie tekstūras.

Pēc tekstūras matricas transformācijas, koordinātes tiek pārveidotas ar interpolāciju un perspektīvas dalīšanu. Tam ir būtiska nozīme, kad tekstūrēta ģeometrija tiek attēlota 3D vidē, jo bez šiem procesiem, tā tiek attēlota nepareizi [5], skat. *att. 2.1.1.*

att. 2.1.1. Perspektīvas Korekcija [5]

Šī 4 × 4 transformēta matrica ar perspektīvas dalīšanu tiek izmantota kā pamats vairākiem paņēmieniem vai tehnikām kā projicētās tekstūras un *volume texturing.*

1.9 Tekstūras koordinātu ģenerēšanas pārskats

Tekstūras koordinātu *pipeline* ir arī iespēja ģenerēt tekstūras koordinātes objektiem, kuriem to nav. Šī funkcionalitāte, kas saucās tekstūras koordinātu ģenerēšana vai *texgen,* ļauj izveidot attiecību starp ģeometriju un pašu tekstūru. Koordinātu ģenerēšanas avots (x, y, z, w) vērtības var būt netransformētas virsotnes (*objekta telpā*), vai virsotnes kas ir transformētas ar *modelview* matricu (*eye space*).

OpenGL ir pāris *texgen* kategorijas: divas versijas ar lineāro kārtošanu, Tom McReynolds 25. lpp. Primā versija ir balstās uz virsotņu normālēm un otrā uz atstarošanas vektoriem, kuri tiek izmantoti priekš *enviroment mapping.* Lielāka atšķirība ir tāda – kurā no 3D transformācijas etapiem tiek ģenerētas koordinātes. Ja koordinātes tiek ģenerētas objekta telpā ar netransformētām virsotnēm, tad tekstūra attēlosies uz objekta virsmas tradicionālajā veidā, piemēram, ja mūsu objekts ir vienkāršs kvadrāts bez tekstūru koordinātēm un ir pieejama kastes tekstūra/bilde, tad, ģenerējot koordinātes objekta telpā un pielietojot kastes tekstūru, rezultātā tiks attēlota 3D kaste. Savukārt, ja *texgen* process notiek *eye space* etapā, pēc *modelview* transformācijas, tad ir pieejami daudz vairāk datu specifiskām darbībām, piemēram, ja objekts ir tējkanna, tam var izveidot dinamisku tekstūru, kas atspoguļo apkārtējo vidi uz savas virsmas. Tā ir dinamiska tekstūra, jo, pārvietojot kameru vai objektu, virsmas krāsojums mainīsies.

2. TEKSTŪRAS UZKLĀŠANAS RISINĀJUMS

Texture mapping vai tekstūras uzklāšana ir viena no visbūtiskākajām metodēm, lai kontrolētu objekta izskatu. Visizplātītākais tekstūras uzklāšanas pielietošana ir nodrošināt virsmas apkrāsojumu ģeometrijai, modificējot virsmas krāsu uz katru pikseli. Šai apkrāsošanai kā pamatu izmanto digitālu bildi. Proti, ar tekstūras uzklāšanu var sasniegt daudz vairāk. Šī nodaļa pārskata OpenGL tekstūras uzklāšanas iespējas, liekot uzsvaru uz funkcijām, kas ir svarīgas sarežģītākiem renderēšanas paņēmieniem.

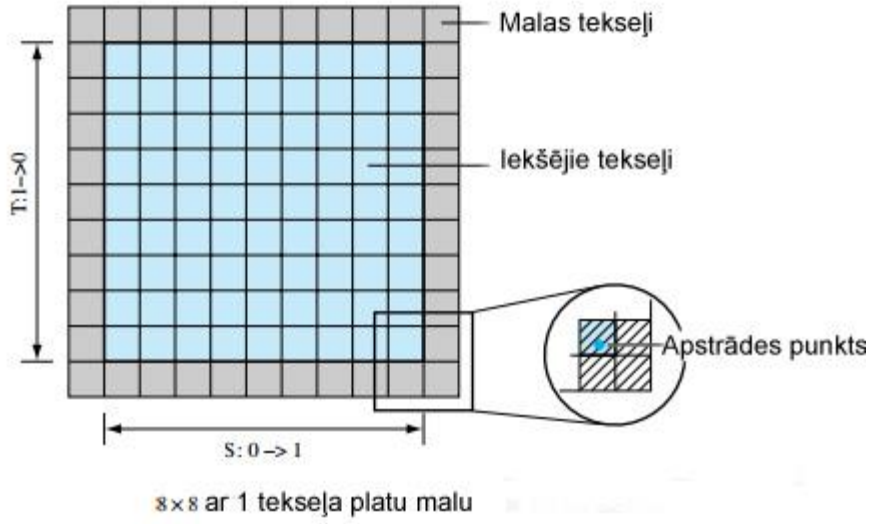
2.1. Tekstūras attēlu ielāde

Pēc Tom McReynolds vārdiem [2] 73. lpp., OpenGL reprezentē digitālo bildi kā n-dimensionālo masīvu ar krāsu vērtībām. Katrs šī masīva individuāls elements tiek saukts par *texel* vai tekselis. OpenGL ir nepieciešams, ka tekstūras attēla dimensijas ir 2^n, citiem vārdiem, attēls ar 264 × 264 dimensijām derēs, bet 207 × 207 nederēs, jo dalot ar divi nebūs vesels skaitlis. Galvenais iemesls, kāpēc tas nepieciešams, ir lai vienkāršotu aprēķinu veikšanu. Šai vienkāršošanai ir trūkumi – attēliem, kuriem dimensijas nav 2^n, ir nepieciešama apgriešana, lai to varētu dēvēt par tekstūru. Ir OpenGL paplašinājumi, kas atceļ šo ierobežojumu, bet tajā pašā laikā ierobežo funkcionalitāti, ko var pielietot tekstūrai.

`glTexImage1D`, `glTexImage2D` un `glTexImage3D` komandas ielādēs visu attēlu, kas ievieto attēla datus sistēmas atmiņā. OpenGL pikseļu krātuves *unpack* statuss norāda kā attēla dati ir sakārtoti sistēmas atmiņa. OpenGL pikseļu pārsūtīšanas *pipeline* apstrādā tekstūras attēla datus. Šajā posmā var veikt tādas darbības kā, piemēram, krāsu spektra pārbīde. Kā jau zināms, tekstūras attēli ir saistīti ar tekstūras koordinātēm. Tekstūras koordinātes, neatkarīgi no attēla izšķirtspējas, ir diapazonā no 0 līdz 1. Individuāli tekseļi ir sasniedzami, mērogojot koordinātes vērtības ar tekstūras dimensijām. Ir gadījumi, kad tekstūras koordinātes ir ārpus [0, 1] diapazona. Tādā gadījumā OpenGL var uzstādīt, lai tās daļas tiek aplauztas vai lai tās tiek morfētas vai iespiestas līdz robežām.

2.2. Tekstūras robežas

Tekstūras malas izmanto vairāki specifiski tekstūras *wrapping* vai iesaiņošanas režīmi, lai izrēķinātu tekseli ar lineāro filtrāciju. Pēc definīcijas, tekstūras robeža ir ārpus tekstūras koordinātu diapazona [0,1], Advanced Graphics Programming Using OpenGL [2] 75.lpp. Tekstūras malas tiek izmantotas, kad tekstūra tiek apstrādāta blakus koordinātu robežām [0,1] un tekstūru aplauzšanas režīms ir uzstādīts uz `GL_CLAMP` opciju. Savukārt, izmantojot `GL_NEAREST` filtrāciju, apstrādē neizmanto tekstūras malas, jo šajā filtrācijas režīmā apstrāde izmanto tuvāko tekseli, kas vienmēr ir [0,1] diapazonā. Pati tekstūras mala, *skat. att 2.2.1,* ir apvilkta ap tekstūras robežu un ir vienu tekseli plata, bet neatkarīgi no šī piemēra, mala var būt arī vairāku tekseļu plata.



att. 2.2.1. **tekstūras robeža**

Ar glTexImage funkciju var mainīt parametru, kurš nosaka vai attēlam ir vai nav malas. Ja nav vajadzīga tekstūras mala un tās vietā jābūt vienkāršai krāsai, tad caur glTexParameter var uzstādīt `GL_TEXTURE_BORDER_COLOR` atribūtu. Tomēr tekstūras robežas ir ļoti noderīgas gadījumos, kad vairākas tekstūras tiek izmantotas kopā, lai veidotu lielāku tekstūru jeb *tiling*. Ja nav robežu, tad, izmantojot `GL_LINEAR` filtrāciju, krāsa tekstūras malās būs nepareizi apstrādātas, rezultātā veidojot vizuālos artefaktus.

2.3. Tekstūru formāti

Izvēloties attiecīgo tekstūras formātu, lietojumprogrammā var radīt kompromisus starp tekseļa izšķirtspēju, tekstūras lielumu un tekstūru ielādes noslogojumu. Piemēram, uzstādot GL_LUMINACE formātu GL_RGB standarta formāta vietā, tas samazina tekstūras atmiņu

lietojumu līdz vienai trešdaļai. OpenGL Specifikācijā [1] 154. lpp. pastāv arī izmēra-specifiski vai *size-specific* tekstūru formāti kā GL_RGBA8 vai GL _RGBA, kas virza OpenGL implementāciju, lai saglabātu tekstūru ar norādīto izšķirtspēju. Vispārīgi tekstūru formāti, piemēram, GL_RGBA, pēc idejas, ir ērtāki, jo lietojumprogramma izvēlās vispiemērotāko formātu priekš piedāvāta uzdevuma. Tomēr, ja ir svarīgi uzturēt noteiktu formāta izšķirtspējas līmeni, tad ir jālieto izmēra-specifisku tekstūras formātu. Tādā gadījumā, izvēloties zemākas izšķirtspējas krāsu formātus, attēla kvalitātes samazināšana ir neizbēgama. Izvēlēties pareizo kompromisu starp krāsu izšķirtspēju un lielumu nav vienkāršs jautājums, bet tā kā tekstūras ir piemērotas uz specifiskām virsmām, ir pieejama pielāgojamība starp kompromisiem nekā, piemēram, kadra bufera izšķirtspējas maiņa. Piemēram, tekstūra, kas atrodas uz objekta virsmas, kas vienmēr atrodas tālu no skatītāja, tiek maz ietekmēta no šāda kompromisa [6].

tabula 2.3.1. **Iekšējie tekstūru formāti**

Sized Internal Format	Base Internal Format	R bits	G bits	B bits	A bits	L bits	I bits
ALPHA4	ALPHA				4		
ALPHA8	ALPHA				8		
ALPHA12	ALPHA				12		
ALPHA16	ALPHA				16		
LUMINANCE4	LUMINANCE					4	
LUMINANCE8	LUMINANCE					8	
LUMINANCE12	LUMINANCE					12	
LUMINANCE16	LUMINANCE					16	
LUMINANCE4_ALPHA4	LUMINANCE_ALPHA				4	4	
LUMINANCE6_ALPHA2	LUMINANCE_ALPHA				2	6	
LUMINANCE8_ALPHA8	LUMINANCE_ALPHA				8	8	
LUMINANCE12_ALPHA4	LUMINANCE_ALPHA				12	4	
LUMINANCE16_ALPHA16	LUMINANCE_ALPHA				16	16	
INTENSITY4	INTENSITY						4
INTENSITY8	INTENSITY						8
INTENSITY12	INTENSITY						12
INTENSITY16	INTENSITY						16
R3_G3_B2	RGB	3	3	2			
RGB4	RGB	4	4	4			
RGB5	RGB	5	5	5			
RGB8	RGB	8	8	8			
RGB10	RGB	10	10	10			
RGB12	RGB	12	12	12			
RGB16	RGB	16	16	16			
RGBA2	RGBA	2	2	2	2		
RGBA4	RGBA	4	4	4	4		
RGB5_A1	RGBA	5	5	5	1		
RGBA8	RGBA	8	8	8	8		
RGB10_A2	RGBA	10	10	10	2		
RGBA12	RGBA	12	12	12	12		
RGBA16	RGBA	16	16	16	16		

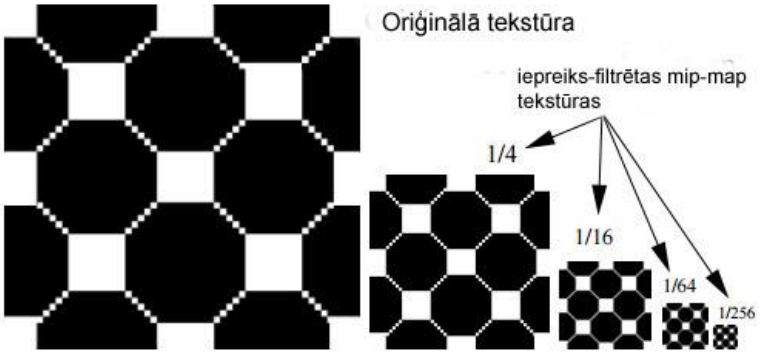
Tabulā 2.3.1. ir attēloti Iekšējie tekstūru formāti. Katra formātam ir atbilstošs bāzes iekšējais formāts un komponentu izšķirtspējas. Pēc Tom McReynolds [2] domām, vislabākā

pieeja, izmantojot specifiskus iekšējo tekstūru formātus, ir rūpīgi analizēt tekstūras lomu aplikācijā, lai izmantotu tekseļa izšķirtspēju tur, kur tam ir vislielākā ietekme un mazāki trūkumi.

2.4. Tekstūru filtrēšana

Tekstūras filtrēšana vai izlīdzināšana ir metode, ko izmanto, lai noteiktu krāsu tekstūras pikselim, izmantojot tuvumā esošo tekseļa krāsu. Ir divas galvenās tekstūras filtrēšanas kategorijas, palielināšanas filtrēšana un mazināšanas filtrēšana [6]. Atkarībā no stāvokļa tekstūras filtrēšana ir vai nu atjaunošanas filtrs, kur izkliedēti dati tiek interpolēti, lai aizpildītu spraugas (palielināšana), vai arī kropļojumnovērses vai *antialiasing* tips, kur tekstūras sampli eksistē daudz vairāk, nekā tas ir vajadzīgs priekš tekstūras aizpildes. Citiem vārdiem, filtrēšana nosaka to, kā tekstūra izskatīsies dažādās virsmās, izmēros, leņķos un mērogos. Atkarībā no izvēlētā filtra algoritma, rezultāti atšķirās ar dažādu izplūšanas līmeni un detalizāciju.

OpenGL piedāvā vairākas filtrēšanas metodes, lai izskaitļotu tekseļa vērtību. Vienkāršākais filtrs ir *point sampling* vai punktveida nolasītajiem, kurā tiek atlasīta tekseļa vērtība, kas atrodas vistuvāk tekstūras koordinātēm. Punktu nolase reti sniedz apmierinošus rezultātus, tāpēc lietojumprogrammā biežāk izvēlās filtra veidu, kas interpolē [6]. Palielināšanai – OpenGL atbalsta tikai lineāro interpolāciju starp četrām tekseļu vērtībām. Attiecībā uz mazināšanu OpenGL atbalsta dažādu veidu mipkartēšanu vai *mip-mapping,* un vislietderīgākais, un tajā paša laikā skaitļojamākais, ir trilineārā mipkartēšana. Izmantojot mip-kartēšanu, tekstūru veido vairāki detalizācijas līmeņi (*LOD*). Katrs mipkartēšanas līmenis ir noteikts tekstūras attēls. Pamata mipkartēšanas līmenim ir augstākā izšķirtspēja un tas tiek saukts par nulto mipkartēšanas līmeni. Katrs nākamais līmenis ir degradēts uz pusi no tā izmēra (augstums, platums un dziļums). Lai mipkartēšanas filtrēšana strādātu pareizi, katrs mipkartēšanas līmenis ir iepriekšējā līmeņa versija ar samazinātu izšķirtspēju. 2.4.1. attēlā redzams, kā tekstūras mipkartēšanas līmeņi nodrošina vairāku detalizācijas līmeņus tekstūras attēlam.



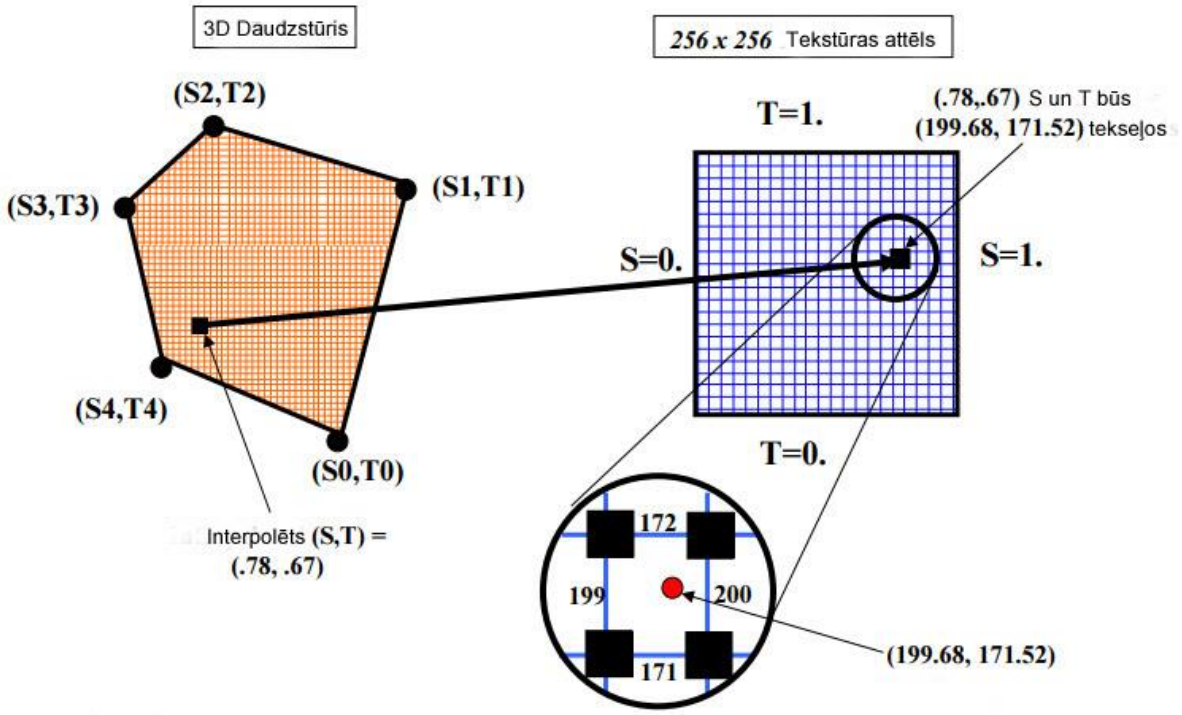
att. 2.4.1. **Detalizācijas līmeņi**

OpenGL nav iebūvētu komandu, kas paredzētas, lai ģenerētu mipkartes, bet GLU nodrošina dažādas vienkāršas rutīnas (gluBuild2DMipmaps) mipkaršu ģenerēšanai.

3. TEKSTŪRAS UZKLĀŠANAS RISINĀJUMA PIEMĒRI

3.1. Tekstūras uzklāšana

Attēlā *att. 3.1.1.* ir redzams, kā darbojās tekstūras uzklāšana starp telpisku objektu un tā tekstūras koordinātēm S un T.



***att. 3.1.1.* Tekstūras uzklāšana**

Tiek norādīts (s, t) pāris pie katras virsotnes, kopā ar virsotnes koordinātu. Tajā pašā laikā, kad OpenGL interpolē koordinātas, krāsas utt. iekš daudzstūra, tiek arī interpolētas (s, t) koordinātes. Kā iepriekšējās nodāļās ir aprakstīts, rezultāta interpolētas (s, t) koordinātes būs [0, 1] diapazonā, kuras izmanto, lai atrastu krāsas vērtību dotajam pikselim. Tas ir redzams šajā *att. 3.1.1* piemēra bāzē. Attēls, kas izmantots kā objekta tekstūra, ir ar 256 × 256 izšķirtspēju, un apstrādātās (s, t) koordinātes ir ar 0.78 un 0.67 vērtībām. Lai piemeklētu atbilstošu krāsas vērtību uz telpiskā objekta virsmas, interpolētas koordinātes (s, t) tiek piereizinātas ar atbilstošām tekstūras attēlu dimensijām.

$$0{,}78 * 256 = 199{,}68$$
$$0{,}67 * 256 = 171{,}52$$

3.2. Tekstūras uzklāšanas risinājums

3.2.1. Tekstūras attēla izveide lietojumprogrammā

Viens no veidiem ir izveidot pašam attēlu. Ir dažādi veidi, kā glabāt krāsu vērtības informāciju atkarībā no tā, vai glabājiet vienu, divus, trīs vai četras vērtības uz vienu tekseli. Trīs vērtību (*RGB*) tekseli var izveidot šādi:

```
unsigned char Texture[][3] =
{
        {     R0,    G0,    B0     },
        {     R1,    G1,    B1     },
        {     R2,    G3,    B2     },
       •••
};
```

[7]

tabula 3.2.1.1. RGB Krāsu vērtības

`R0, G0, B0`	Pirmās rindas RGB vērtības atbilst pirmajam bildes tekselim un atrodas 0 – 255 vērtību diapazonā.
`R1, G1, B1`	Šīs vērtības atbilst otrajam bildes tekselim.

Četru vērtību (RGBA) tekseli var izveidot šādi:

```
unsigned char Texture[][4] =
{
        {     R0,    G0,    B0,    A0     },
        {     R1,    G1,    B1,    A1     },
        {     R2,    G3,    B2,    A2     },
       •••
};
```

tabula 3.2.1.2. RGBA Krāsu vērtības

`R0, G0, B0, A0`	Pirmās rindas RGBA vērtības atbilst pirmajam bildes tekselim un atrodas 0 – 255 vērtību diapazonā.
`R1, G1, B1, A0`	Šīs vērtības atbilst otrajam bildes tekselim.

Šādu tekseļa krāsu definēšanas veidu izmanto diezgan reti, taču šis piemērs labi atspoguļo to, kādus datus satur tekselis.

Visbiežāk lietojumprogramma importē gatavu attēlu. Šīm attēlam ir nepieciešamas izmaiņas, lai tas kļūtu par tekstūras objektu. SOIL vai *Simple OpenGL Image Library* ir maza un viegli lietojama bibliotēka, ar kuras palīdzību attēla faili tiek ielādēti un pārveidoti par tekstūras objektiem.

```
int width, height;
unsigned char* image = SOIL_load_image("img.png", &width, &height, 0, SOIL_LOAD_RGB);

glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB,
             GL_UNSIGNED_BYTE, image);
```

[7]

tabula 3.2.1.3. glTexImage2D parametri

GL_TEXTURE_2D	Norāda uz to, ka šī ir parasta 2D tekstūra. Citi varianti būtu 1D un 3D tekstūras.
0 (*LOD*)	Otrais parametrs norāda uz detalizācijas līmeni, kas pašlaik ir 0 un tas nozīmē, ka tiek izmantots bāzes attēls.
GL_RGB	Norāda iekšējo tekstūras formātu, un šajā gadījumā ir GL_RGB formāts.
width	Norāda tekstūras platumu.
height	Norāda tekstūras augstumu.
0 (*border*)	Norāda uz to, cik plata būs tekstūras mala.
GL_UNSIGNED_BYTE	Attēla dati tiek ielādēti masīvā, un GL_UNSIGNED_BYTE ir šī masīva formāts, kas tiks sūtīts uz videokarti.
image	Masīvs vai tekstūras objekts, kas satur attēla datus.

Citas bibliotēkas, kas atbalsta dažādus faila tipus, kā SOIL ir *DevIL* un *FreeImage*.

3.2.2. Sagatavošana zīmēšanai

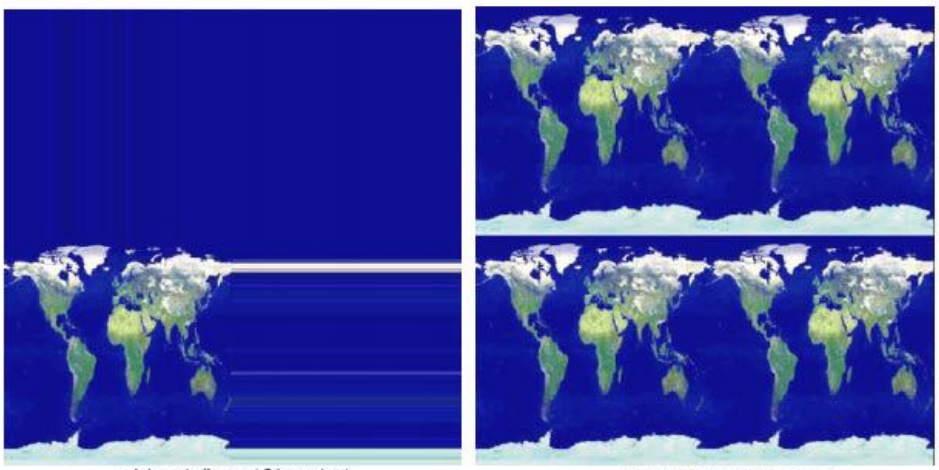
Vajag definēt tekstūras aplaušanas parametrus. Kā jau iepriekšējās nodaļās bija aprakstīts, šis parametrs maina tekstūru, ja tekstūras koordinātes ir ārpus 1.0 un 0.0 robežām.

```
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, wrap );
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, wrap );
```
[7]

Kur *wrap* var būt:

tabula 3.2.2.1. wrap parametrs

GL_CLAMP	Norāda, ka tekstūra atkārtosies, ja transformētas tekstūras koordinātes ir ārpus [0,1]. Skat *att 3.2.2.1.* ***a***.
GL_REPEAT	Norāda uz to ka tā tekstūras daļa ārpus tekstūras koordinātēm [0,1] būs iespiesta uz 0 un 1 robežām. Skat *att 3.2.2.1.* ***b***.



***att. 3.2.2.1. wrap* parametrs**

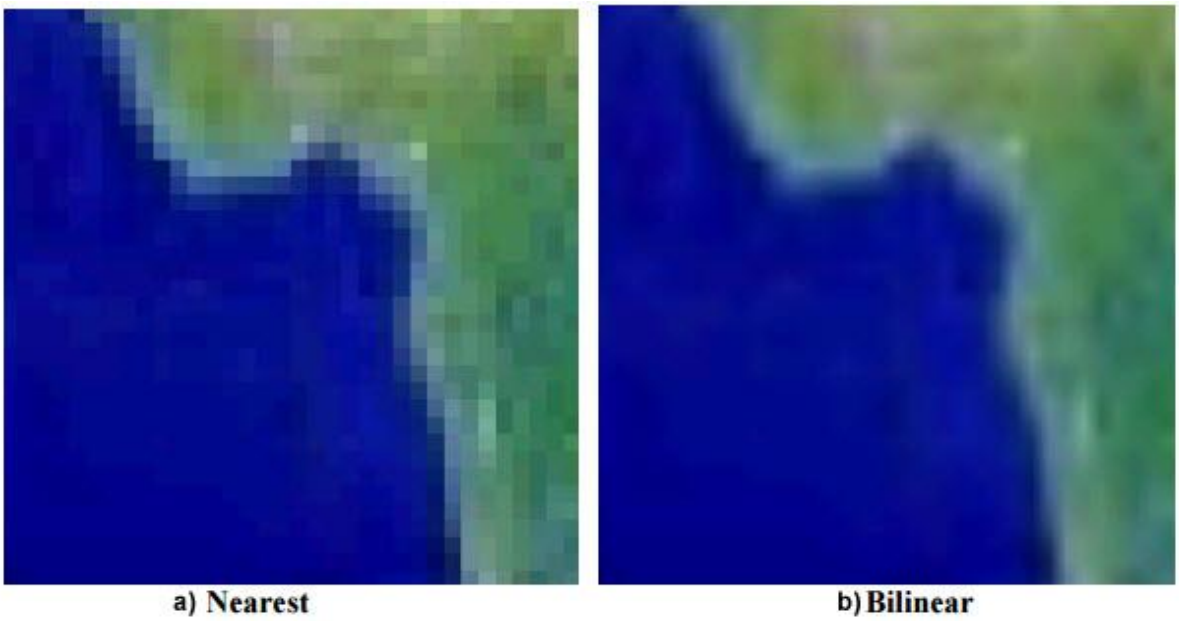
Otrais solis ir definēt tekstūras filtra parametrus. Tas kontrolē tekstūras izskatu, ja tā tiek palielināta vai samazināta.

```
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, filter );
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, filter );
```
[7]

Kur *filter* var būt:

tabula 3.2.2.2. wrap parametrs

GL_NEAREST	Šī opcija apstrādei izmanto tuvāko tekseli, kas vienmēr būs [0,1] diapazonā. Skat *att 3.2.2.2.* ***a***.
GL_LINEAR	Šī opcija apstrādei izmanto četrus tuvākos tekseļus. Šai filtrēšanai vel ir nepieciešama tekstūras mala. Skat *att 3.2.2.2.* ***b***.



***att. 3.2.2.1. wrap* parametrs**

3.2.3. Telpiska kuba izveide un renderēšana

Šie koda fragmenti tiek rakstīti *Display()* funkcijā.

Pirmais solis ir aktivizēt tekstūras uzklāšanu.

```
glEnable( GL_TEXTURE_2D );
```

Jānorāda kuba virsotnes, kā arī s un t koordinātes katrai virsotnei:

```
glBegin( GL_POLYGON );
      glTexCoord2f( s0, t0 );
      glNormal3f( nx0, ny0, nz0 );
      glVertex3f( x0, y0, z0 );

      glTexCoord2f( s1, t1 );
      glNormal3f( nx1, ny1, nz1 );
      glVertex3f( x1, y1, z1 );

      . . .
glEnd();
```

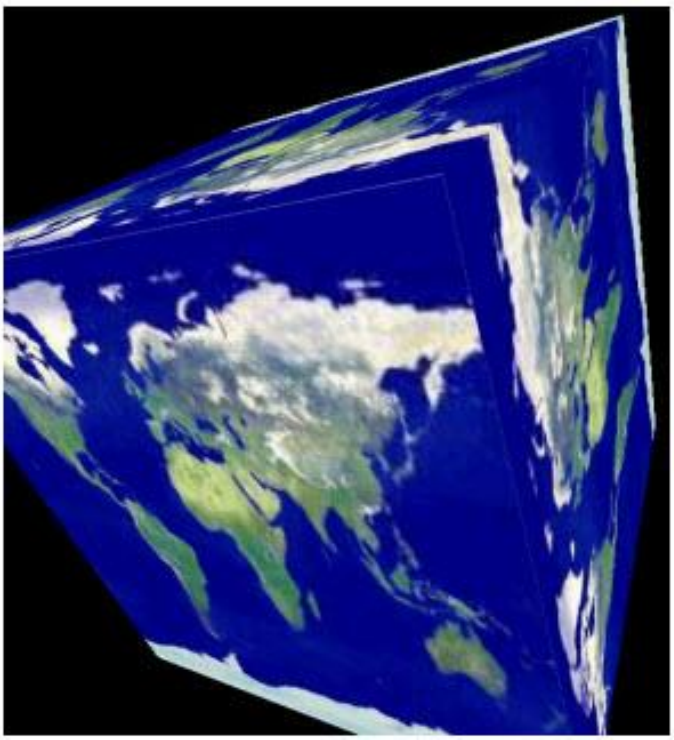
[7]

Šīs funkcijas izveidos atbilstošus virsotņu, normāļu un tekstūras koordinātu sarakstus, kas rezultātā tiks attēlots uz ekrāna.

Pēc šīm koda rindām ir jāatslēdz tekstūras uzklāšanas opcija.

```
glDisable( GL_TEXTURE_2D );
```

Bildē *att 3.2.3.2.* ir redzams, kā izskatīsies šis tekstūrēts kubs.



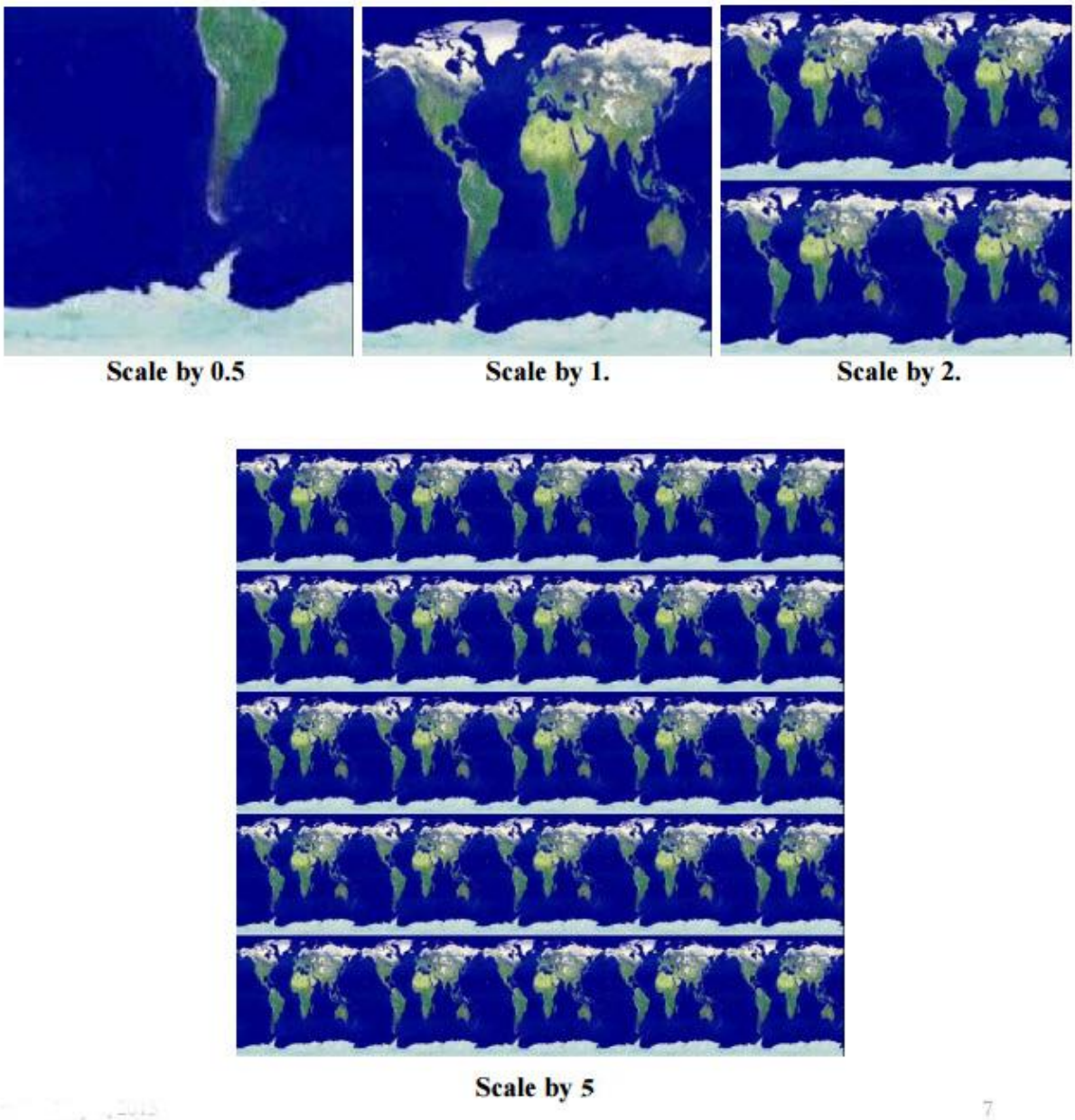
***att. 3.2.3.2.* tekstūrēts kubs**

3.2.4. Tekstūras koordinātu pārveidošana

Papildus projekcijas un *modelview* matricām OpenGL uztur arī tekstūras koordināšu transformācijas S un T koordinātēm. Tiek izmantotas visas tās pašas transformācijas rutīnas: `glRotatef(); glScalef(); glTranslatef(),` kas attiecās arī uz objekta matricām, bet, lai pārveidotu koordinātes, ir nepieciešams norādīt to, ka transformēsiet tieši tekstūras koordinātu matricu.

```
glMatrixMode( GL_TEXTURE );
```

Šajā gadījumā ir jāatceras, ka tiek mainītas tekstūras koordinātes, nevis tekstūras attēls. Tekstūras attēla maiņa uz priekšu ir tā pati, kas pārveido tekstūras koordinātes atpakaļ.



att. 3.2.4.1. **dažādas transformācijas tekstūras koordinātu matricai.**

4. VIDES UZKLĀŠANAS METODES

Šajā nodaļā tiks aprakstīti OpenGL zināmie risinājumi vides uzklāšanai. Var likties, ka tie nav cieši saistīti ar šī darba risinājumiem, tomēr daļa no risinājumiem balstās uz paņēmieniem, kas ir aprakstīti šajā nodaļā.

4.1. Kas ir vides uzklāšana?

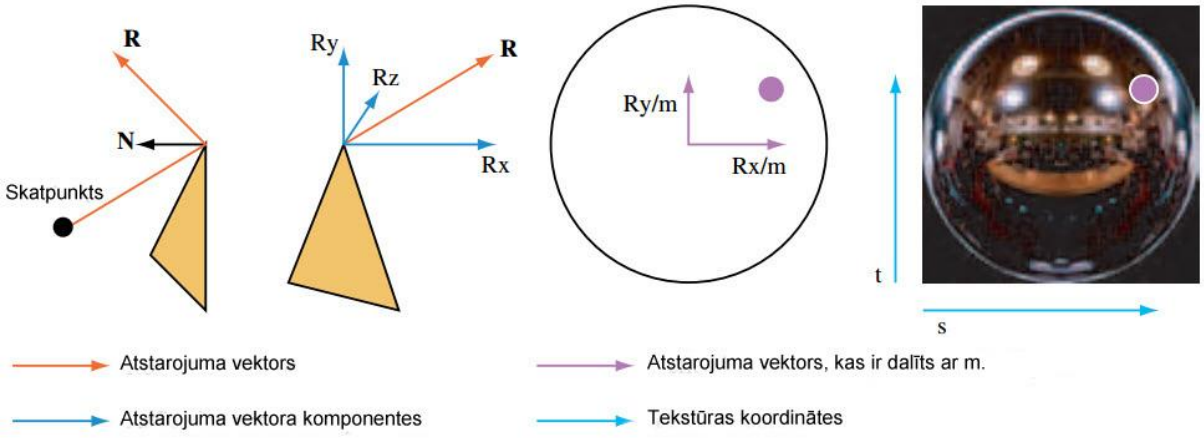
Ainas reālismu var uzlabot ar *enviroment mapping* vai vides uzklāšanu, modelējot apgaismojuma efektus, kas radušies no apkārtējas vides. OpenGL nodrošina *ambient light* vai apkārtējo apgaismojumu, bet tās gaismas vienādojums ir tikai rupjš tuvinājums reālai apkārtējai videi. Izmantojot OpenGL tekstūras uzklāšanas funkcionalitāti, ir pieejama daudz labāka pieeja apkārtējas vides attēlošanai uz objekta virsmas. Termina „vides uzklāšana" apraksta teksturēšanas metodi, ko izmanto, lai modelētu dažādas apkārtējās vides ietekmes uz objekta izskatu. Vides uzklāšana, kā parasta tekstūras uzklāšana, maina objekta izskatu, taču vides uzklāšanas tekstūrā tiek ņemts vērā apkārtējās vides skats. Ja objekta virsmai ir uzstādīta augsta spoguļgluduma vai *specular* vērtība, tad tekstūrā uzrādīsies apkārtējo objektu atspoguļojums. Vides uzklāšanas tekstūra, kas ir izveidota, ir pareizi jāuzklāj objekta virsmai. Tā kā ar vides tekstūrā tiek imitēts gaismas efekts, tekseļi tiek atlasīti kā funkcija no normāles-vektora vai atstarošanas vektora katram punktam uz objekta virsmas. Šie vektori tiek pārveidoti tekstūras koordinātēs uz katras virsotnes, tad katrs virsmas punkts tiek interpolēts, līdzīgi kā tas tiek veikts standarta virsmas tekstūrā.

4.2. Vides tekstūru koordinātu ģenerēšanas pārskats.

OpenGL vides uzklāšanas funkcionalitāte ir sadalītas divās daļās: kopa ar tekstūras koordinātu ģenerēšanas funkcijām un papildus tekstūras tips ar nosaukumu *cubemap.* Lai palielinātu funkcionalitāti, abas grupas ir ortogonālas; tekstūras izveides funkcijas var izmantot ar jebkura veida tekstūras tipiem un *cubemap* tekstūras var indeksēt ar trīs tekstūras koordinātēm „Advanced Graphics Programming Using OpenGL" [2] 81.lpp. Vides uzklāšanai ir paredzētas trīs tekstūras koordinātu ģenerēšanas funkcijas: *normal mapping, reflection vector mapping* un *sphere mapping.* Funkciju var atlasīt, iestatot atbilstošo parametru glTexGen komandā: `GL_NORMAL_MAP`, `GL_REFLECTION_MAP` vai `GL_SPHERE MAP`, OpenGL Specifikācija [1] 50.lpp.

GL_NORMAL_MAP - *Normal-vector* tekstūras ģenerēšana nodrošina, ka tekstūras objektu ir iespējams lietot uz virsmas, pamatojoties uz virsmas-normāles virzienu. Tā izmanto trīs komponentu normāles kā tekstūras koordinātes, uzklājot Nx, Ny un Nz atbilstoši s, t un r vērtībām. Ir pieņemts, ka normāles vektori ir vienas vienības garumā, tāpēc ģenerētas tekstūras koordinātes ir diapazonā no -1 līdz 1. Pēc Tom McReynolds domām [2], tieši šī metode ir noderīga, veicot objekta izkliedētās gaismas atstarošanu, virsmas krāsa kļūst par virsmas izvietojuma un orientācijas funkciju attiecībā pret tas apkārtnes gaismas avotiem.

GL_REFLECTION_MAP - Atstarošanas tekstūras ģenerēšana indeksē virsmu, kas balstās uz atstarojuma vektora komponentēm. Atstarojuma vektoru aprēķina, izmantojot virsotnes normāles un acs skata vektoru (*eye vector*). Svarīgi pieminēt, ka acs skata vektoru rēķina, ņemot garumu starp acs pozīciju un virsotnes atrašanas vērtībām. Acu vektors U un atstarojuma vektors R tiek rēķināti redzamajā telpā vai *eye space*. Atstarošanas vektoru iegūst, izmantojot vienādojumu R = U − 2NT(U · N), kur N ir virsotnes normāle, kas ir jau transformēta uz redzamo telpu. Izmantotais atstarojuma vienādojums ir atstarojuma vektora skaitļošanas standarts. Kad atstarojuma vektors ir aprēķināts, tā sastāvdaļas tiek pārveidotas par tekstūras koordinātēm, uzklājot Rx, Ry un Rz atbilstoši s, t un r vērtībām. Tā kā N un U ir normalizēti, R ir arī normalizēts, tāpēc tekstūras koordinātes būs no -1 līdz 1 diapazonā. Šī funkcija ir noderīga, lai modelētu spoguļgludus objektus, kuru apgaismojums ir atkarīgs gan no objekta, gan skatītāja pozīcijas.



***att. 4.2.1.* sfēriskās uzklāšanas tekstūras koordinātu ģenerēšana.**

GL_SPHERE_MAP – sfēriskās uzklāšanas tekstūras ģenerēšana tika atbalstīta jau no OpenGL versijas 1.0. Lai gan tekstūras ģenerēšanas veidi veido trīs koordinātes, sfēras uzklāšana veido tikai divas: s un t. Tas notiek, ģenerējot atstarošanas vektoru, kas definēts iepriekš, tad mērogojot Rx un Ry komponentes ar modificētu atstarojuma vektora garumu M. M garumu aprēķina kā divi R2x + R2y +(Rz +1)2. Rx un Ry dalīšana ar šo M garumu,

projicē divas komponentes vienā vektorā. Kad apstrādāti Rx un Ry vektori ir mērogoti, tie būs [0, 1] robežās, un tos var izmantot kā s un t koordinātes. Tātad, kaut arī pārējie tekstūras ģenerēšanas režīmi tiek veidoti no trīs koordinātēm, tāpēc ir nepieciešama papildus tekstūra, kas tās ver indeksēt (parasti tā ir *cubemap*), sfēriskās uzklāšanas tekstūras koordinātu ģenerēšanas funkcija izmanto normālu 2D tekstūru skat. *att 4.2.1*.

4.3. Kubiskā uzklāšana vai *Cube Mapping*

Cubemap tekstūra sastāv no sešām 2D tekstūrām, tās var iztēloties kā saplacinātu kubu ar izlīdzinātām skaldnēm. To izmanto kā vides avotu – objekti ar augstu spoguļgluduma vērtību atstaros *cubmap* tekstūras saturu. S, t un r tekstūras koordinātes veido normalizēta vektora komponentes, kas rodas no kuba centra. Vektora galvenā ass ir ar lielāko izmēra komponenti un tiek izmantota, lai izvēlētos tekstūru prieks kuba skaldnes (*cube face*). Atlikušās divas komponentes indeksē tekseļus filtrēšanai. Tā kā komponentu diapazons ir no -1 līdz 1, filtrēšanas etaps mērogo un nobīda vērtības normālajā [0, 1] diapazonā, lai tās varētu izmantot, lai indeksēt kuba skaldni 2D tekstūrā. *Cubemap* izmantošanai ir nepieciešams, lai tā tekstūras ir ielādētas, konfigurētas un iespējotas. Katrai virsotnei ir jābūt iestatītām vai ģenerētām tekstūras koordinātēm. Cubemap tekstūru var ielādēt ar parastām, iepriekš definētām, OpenGL komandām kā glTexImage2D, bet pirmajam komandas parametram, kas apraksta ielādes mērķi (piemēram, kā iepriekš aprakstīts - parastai 2D tekstūrai ir GL_TEXTURE_2D), ir jānorāda uz vienu no kuba skaldnēm, skat. *tabula 4.2.1*.

***tabula 4.3.1. cubemap* parametrs**

Tekstūras identifikācija	Orientācija
GL_TEXTURE_CUBE_MAP_POSITIVE_X	Pa labi
GL_TEXTURE_CUBE_MAP_NEGATIVE_X	Pa kreisi
GL_TEXTURE_CUBE_MAP_POSITIVE_X	Augša
GL_TEXTURE_CUBE_MAP_NEGATIVE_X	Leja
GL_TEXTURE_CUBE_MAP_POSITIVE_X	Mugurpuse
GL_TEXTURE_CUBE_MAP_NEGATIVE_X	Priekša

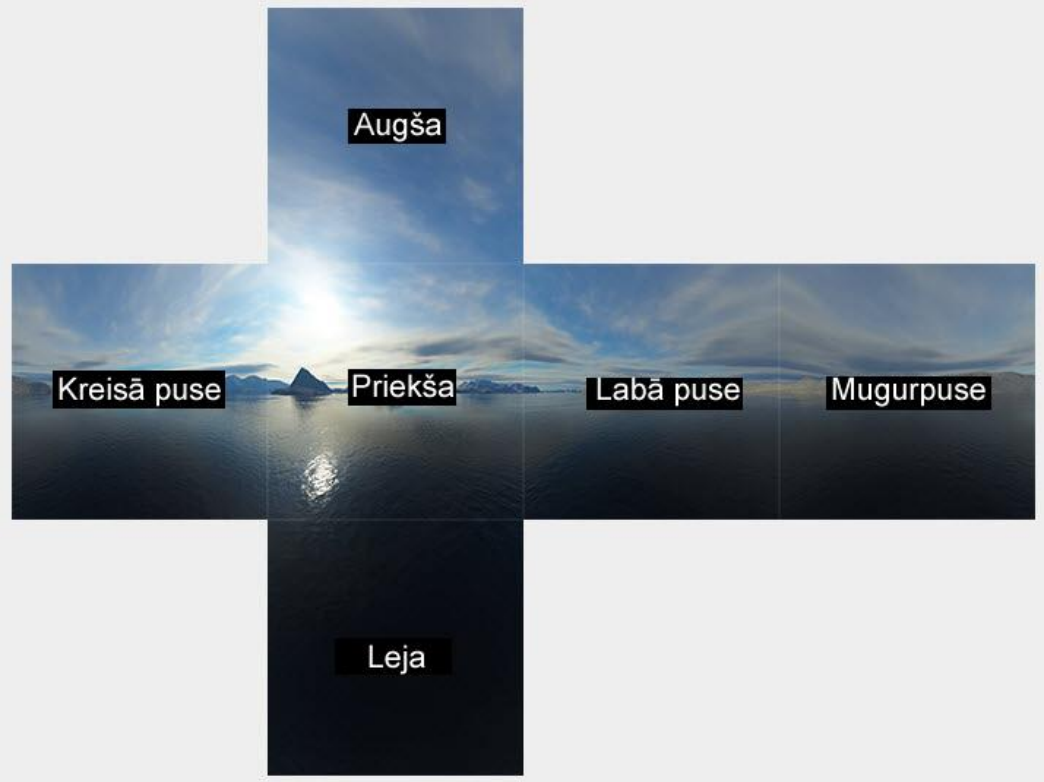
Rezultātā, lai ielādētu tekstūru visām kuba skaldnēm, ir nepieciešams izsaukt sešas glTexImage2D komandas.

Cubemap teksturēšana ir iespējota, izmantojot glEnable komandu ar GL_TEXTURE_CUBE_MAP argumentu. Katra kuba skaldne var būt 2D tekstūra vai

mipkarte. Tam var piemērot parastas procedūras un rutīnas, vienīgā atšķirība ir glTexImage2D tekstūras mērķa parametrā – atbilstīgais tekstūras mērķis, kas parādīts *tabulā 4.2.1.*, ir jāizmanto GL_TEXURE_2D vietā.

4.3.2. Debeskaste vai *Skybox*

Debes-kaste ir liels kubs, kas ietver visu ainu un satur sešus attēlus ar apkārtējo vidi. Tā liek spēlētājam domāt, ka vide, kurā viņš atrodas, ir daudz lielāka, nekā tā patiesībā ir. Dažu videospēļu *skybox* piemēri ir kalni, mākoņi vai zvaigžņota nakts debess. Debeskartes lieliski iederas *cubemap* funkcionalitātē: ir kubs, kam ir sešas skaldnes, un visām skaldnēm jābūt tekstūrētām. Parasti ir pietiekami daudz resursu, lai varētu atrast šādas debeskastes un to attēliem ir līdzīga shēma kā *att. 4.2.2.1.*

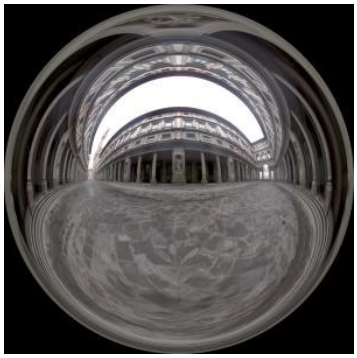


***att. 4.3.1. debes-kastes* tekstūru shēma**

Ja šīs sešas puses tiktu salocītas, tad iegūtu teksturētu kubu, kas imitē plašu ainavu. Daži resursi nodrošina *cubemap* tekstūru tādā formātā, kas ir attēlots *att 4.3.1.*, tādā gadījumā ir manuāli jāizvelk seši skaldnes attēli. Tomēr lielākā daļā gadījumu tie tiek piedāvāti kā seši dažādi tekstūras attēli.

4.2. Sfēriskā uzklāšana vai *Sphere Mapping*

Sphere map tekstūra ir parasta 2D tekstūra, bet ar iepriekš speciāli deformētām īpašībām skat *att. 4.3.1.* Gatavas tekstūras var atrast internetā, bet nesagādā grūtības izveidot tās pašam izmantojot, piemēram, *Adobe Photoshop* ar *Spherify* filtru. To var iztēloties kā hroma sfēru, kas atspoguļo apkārtni.



***att. 4.3.1. sphere map* tekstūra**

Pareizi normalizēti atstarošanas vektori ir vienmēr garantēti ietilpt sfēras aplī [2] 82.lpp, tekseļi, kas ir ārpus sfēras, nekad netiks izmantoti. *Sphere map* tekstūras attēls ir veidots tā, lai sniegtu tekseļus tam koordinātēm, kas iegūtas no atstarojuma vektoriem uz katru virsotni. Tā kā sfēriskai uzklāšanai ir nepieciešama tikai viena tekstūra, OpenGL konfigurēšana ir pietiekami vienkārša. Tā kā *sphere map* tekstūra ir 2D, ir jāģenerē tikai s un t koordinātes.

```
glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);

glEnable(GL_TEXTURE_GEN_S);
glEnable(GL_TEXTURE_GEN_T);
```

Tāpat kā jebkura cita vides uzklāšanas metode, *sphere map* tekstūras koordinātu ģenerēšanu var uzskatīt par divdimensionālo apstrādes funkciju, kas pārvērš atstarojuma vektora virzienus uz krāsas vērtībām.

5. TEKSTŪRAS KOORDINĀŠU ĢENERĒŠANAS RISINĀJUMI

Šajā sadaļā ir aprakstīti risinājumi tekstūras koordinātu ģenerēšanai, ko autors izmantoja savā darbā.

5.1. Plaknes vienādojums

Tā vietā, lai nevajadzētu atsevišķi noteikt tekstūras koordinātes katrai virsotnei, var izmantot fiksētas tekstūras koordinātu ģenerēšanas (texgen) funkcijas, lai OpenGL automātiski pats aprēķina tekstūras koordinātes. Tas izmanto funkciju glTexGen un glEnable režīmus: GL_TEXTURE_GEN_S un GL_TEXTURE_GEN_T.

Texgen režīmi GL_OBJECT_PLANE un GL_EYE_PLANE veido tekstūras koordinātes, pamatojoties uz virsotņu attālumu no plaknes. Plakni 3D dimensiju telpā var noteikt ar šādu vienādojumu:

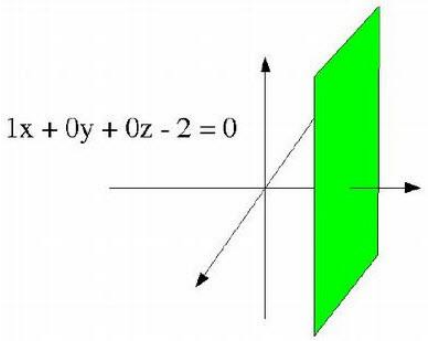
$$Ax + By + Cz + D = 0$$

Šī formula definē plakni kā punktu kopu, kas atbilst vienādojumam. Citiem vārdiem sakot, tas kalpo par pārbaudi, vai konkrētais punkts [x, y, z] ir uz plaknes vai nav. Cita interpretācija ir tādā, ka Ax+By+Cz+D rezultāts ir proporcionāls punkta [x,y,z] attālumam no plaknes.

5.1.1. Koeficentu nozīme

Attēls *att. 5.1.1.1.* ir šī piemēra bāze. Vektors [A, B, C] ir normāle, kas ir perpendikulāra plaknei. Tāpēc, tas nosaka plaknes orientāciju.
Ar šādiem koeficientiem:

$$A = 1; B = 0; C = 0;$$



***att. 5.1.1.1. Koeficientu* A B C D piemērs.**

Definējot plakni ar šādiem koeficientiem, tā būs perpendikulāra X asij, vai paralēla Y un Z asu plaknei. Koeficients D kontrolē plaknes attālumu no izcelsmes. Ja [A, B,C] ir vienības vektors, tad D ir vienāds ar attālumu no plaknes līdz izcelsmei.

Jāievēro, ka, ja uz doto [x, y, z] lieto Ax + By + Cz + D, vai 2Ax + 2By + 2Cz + 2D vai kopumā:

$$NAx + NBy + NCz + ND = 0$$

Citiem vārdiem sakot, ja visus četrus koeficientus reizina ar konstanti N, tad tas joprojām nosaka to pašu plakni. Tomēr tas pārsniedz rezultātu par N reizēm. Tas ir noderīgi tekstūras koordinātu ģenerēšanā.

5.2. *Texgen* režīmi

Tekstūras koordinātu ģenerēšanas režīms var būt viens no [8]:

- GL_OBJECT_LINEAR
- GL_EYE_LINEAR
- GL_SPHERE_MAP

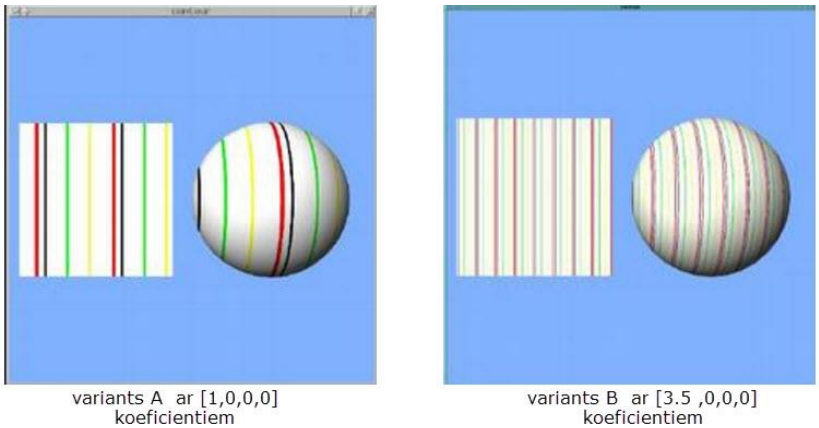
5.2.1. *GL_OBJECT_LINEAR*

Kad koordinātu ģenerēšanas režīms ir uzstādīts uz GL_OBJECT_LINEAR, tekstūras koordinātes aprēķina, izmantojot plaknes vienādojuma koeficientus, kas ir padoti ar funkciju glTexGenfv(GL_S, GL_OBJECT_PLANE, plaknes koificenti).

Koordināti (šajā gadījumā S) aprēķina šādi:

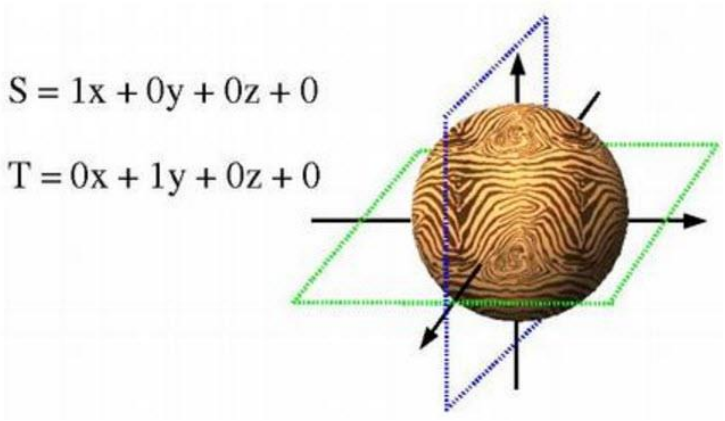
$$S = Ax + By + Cz + D$$

Vērtības [x, y, z] ir pozīcijas koordinātes katrai virsotnei. Ja [A, B, C] ir vienības vektors, tas nozīmē, ka tekstūras koordināte S ir vienāda ar virsotnes attālumu no plaknes. Skat. att 5.2.1.1. kā mainās objekta izskats mainot koeficientus.



***att. 5.2.1.1. Koeficientu* A B C D piemērs.**

Šajos piemēros tiek ģenerēta tikai viena koordināte S un izmantota viendimensionālā tekstūra (glTexture1D). Īstenībā, Gan S, gan arī T koordinātes var izmantot automātisku ģenerēšanu, lai lietotu 2D tekstūru uz automātiski ģenerētām koordinātēm. Tas tiek darīts, izveidojot līdzīgas glTexGen funkcijas izpildi uz T koordinātu, bet izmantojot plakni, kas ir orientēta uz y asi, līdzīgi kā *att. 5.2.1.2.*



att. 5.2.1.2. S un T divdimensionālā koordinātu ģenerēšana.

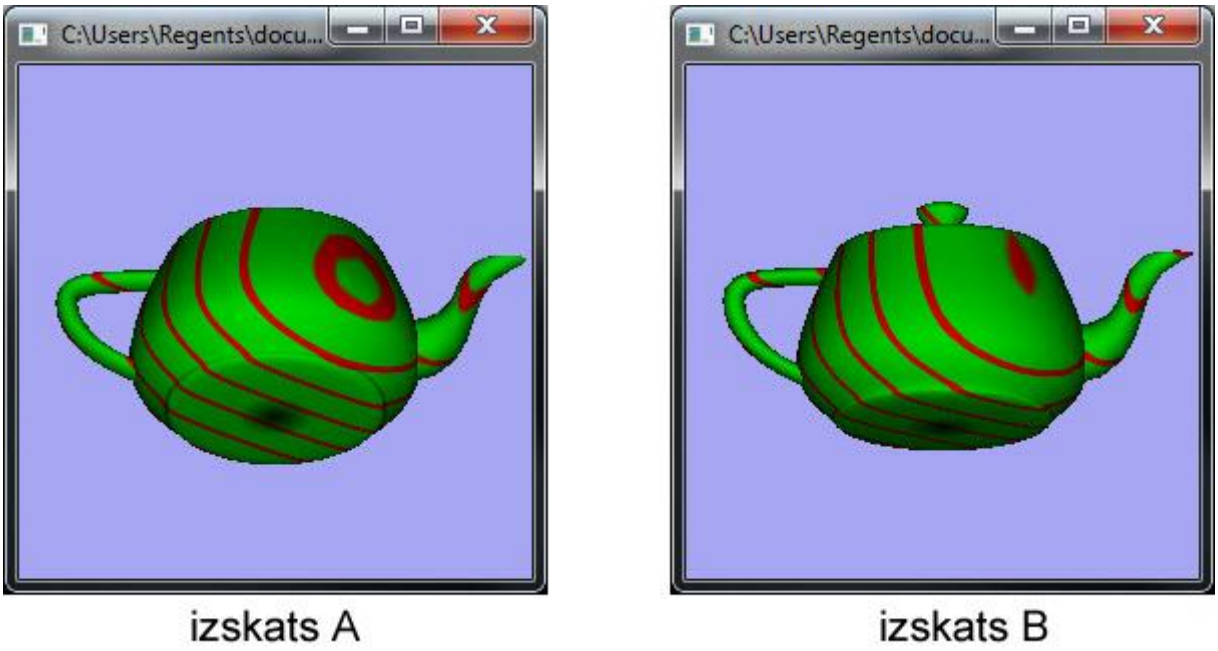
```
SplaneCoefficients = [ 1, 0, 0, 0 ]
glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_OBJECT_LINEAR)
glTexGenfv(GL_S, GL_EYE_PLANE, SplaneCoefficients)
glTexGenfv(GL_S, GL_OBJECT_PLANE, SplaneCoefficients)

TplaneCoefficients = [ 0, 1, 0, 0 ]
glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_OBJECT_LINEAR)
glTexGenfv(GL_T, GL_EYE_PLANE, TplaneCoefficients)
glTexGenfv(GL_T, GL_OBJECT_PLANE, TplaneCoefficients)
```

Šīs process darbībā uz OpenGL ir redzams attēlā *5.2.1.3.* Tā ir viena no darba autora rakstītām lietojumprogrammām.

***att. 5.2.1.3. S un T* divdimensionālā koordinātu ģenerēšana.**

5.2.2. *GL_EYE_LINEAR*

Režīms GL_EYE_LINEAR darbojās līdzīgi kā iepriekšējais, ar koeficientiem kas tiek padoti uz funkciju glTexGenfv(GL_S, GL_EYE_PLANE, koeficienti). Taču lielākā atšķirība ir tāda, ka GL_OBJECT_LINEAR darbojās objekta koordinātēs, savukārt GL_EYE_LINEAR darbojās skata koordinātēs. Tas nozīmē, ka tekstūras koordinātes, kas tiek izskaitļotas ar GL_EYE_LINEAR, izmanto virsotnes pēc visām *modelview* un kameras skata transformācijām. Rezultātā, ja objekts pārbīdās, tad tekstūra uz objekta arī tiks nobīdīta attiecīgi pret skatītāju. Savukārt GL_OBJECT_LINEAR ir atkarīgs tikai no tām vērtībām, kas ir padotas pirms visām transformācijām, un nemainās, ja objekts (vai kamera) pārvietojas. Šīs process darbībā uz OpenGL ir redzams attēlā *5.2.2.1.* Tā ir viena no darba autora rakstītām lietojumprogrammām.



***att. 5.2.1.3. GL_EYE_LINEAR* tekstūras koordinātu ģenerēšanas demonstrācija.**

Kā redzams attēlos, kad objekts tiek nedaudz rotēts, uzklātā tekstūra nobīdās uz objekta virsmas. Tādā veidā tekstūra būs vienmēr orientēta pret skatītāja virzienu.

5.2.3. *GL_SPHERE_MAP*

Sfēriskā uzklāšana, pazīstama arī kā vides uzklāšana, ir *texgen* režīms, kurā tiek imitēta atstarojošā virsma. Tā ir iespējota, iestatot *texgen* režīmu uz GL_SPHERE_MAP.

```
glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP)
glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP)
```

Sfēriskās tekstūras koordinātes aprēķina, ņemot skatpunkta vektoru un atspoguļojot to pret virsmas normāli. Tad iegūtais virziens tiek izmantots, lai atrastu atbilstošo tekseli. Tiek pieņemts, ka tekstūras attēls ir deformēts tā, lai tiktu atspoguļots 360 grādu skats uz apkārtējo vidi. Detalizētāks apraksts ir dots nodaļā *4.2.* Šīs process darbībā uz OpenGL ir redzams attēlā *5.2.2.1.* Tā ir viena no darba autora rakstītām lietojumprogrammām.

***att. 5.2.2.1. GL_SPHERE_MAP* tekstūras koordinātu ģenerēšanas demonstrācija.**

5.3. *Kubiskās uzklāšanas* risinājums

Salīdzinot ar iepriekšējo risinājumu, kubiskā uzklāšana ir daudz efektīvāks paņēmiens, jo tas piedāvā daudz vairāk kontroles pāri objekta izskatam. Ar kubisku uzklāšanu var definēt gan mugurpuses, gan priekšas gan citu pušu izskatu. Grāmatās un interneta resursos norādīte risinājumi ir derīgi tikai kubiem vai debeskastēm[2][11][12], savukārt autora personīgais risinājums ir derīgs dažādiem objektiem ar dažādu virsmu.

Iemesls, kāpēc citu resursu risinājumi strādā tikai ar kubiem un debeskastēm, kas ir iebūvēti kodā, ir tāds, ka tie objekti tiek iekodēti tā, ka vienmēr centrēti ap telpas sākumpunktu (x=0,y=0,z=0)[11]. Darbojoties ar dažādiem modeļiem, modeļa sākumpunkts nesakrīt ar telpas sākumpunktu. Tāpēc mans risinājums iekļauj modeļa sākumpunkta korekciju, kuru pa taisno var rediģēt lietojumprogrammas darbības laikā. Lai sasniegtu tādu rezultātu, no sākuma tika ielādēta *cubemap* tekstūra.

5.3.1. Kubiskā tekstūra

Cubemap ir tekstūra, tāpat kā jebkura cita, tāpēc, lai tā strādātu, mums ir jāsasaista to ar atbilstošu tekstūras identifikāciju vai mērķi `GL_TEXTURE_CUBE_MAP` (*texture target*), pirms mēs veicam jebkādas turpmākas darbības.

```
GLuint textureID;
glGenTextures(1, &textureID);
glBindTexture(GL_TEXTURE_CUBE_MAP, textureID);
```

Tāpēc, ka *cubemap* sastāv no sešām tekstūrām, viena uz katru skaldni, mums jāizsauc glTexImage2D sešas reizes ar atbilstošiem parametriem. Taču šoreiz tekstūras identifikācijas parametrā ir jāuzstāda uz konkrētu kuba skaldni, galvenokārt, lai paskaidrotu OpenGL, kurai kuba skaldnei tiek sūtīta tekstūra. Šīs tekstūras mērķa parametri ir attēloti 4.2. nodaļā *tabula 4.2.1.*

Nākamajās rindās ir cikls, kas iesūta tekstūras uz katru tekstūras mērķi. Sākot ar GL_TEXTURE_CUBE_MAP_POSITIVE_X, kas ar katru iterāciju rezultātā efektīvi izciklo starp visiem tekstūras mērķiem.

```
int width, height;
unsigned char* image;
for (GLuint i = 0; i < textures_faces.size(); i++)
{
    image = SOIL_load_image(textures_faces[i], &width, &height, 0, SOIL_LOAD_RGB);
    glTexImage2D(
        GL_TEXTURE_CUBE_MAP_POSITIVE_X + i,
        0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, image
        );
}
```

Tā kā *cubemap* ir tekstūra, tāpat kā jebkura cita, ir jānosaka arī tā aplaušanas un filtrēšanas metodes.

```
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_R, GL_CLAMP_TO_EDGE);
```

Ir iestatīta aplaušanas metode `GL_CLAMP_TO_EDGE`, jo tekstūras koordinātēm, kas precīzi atrodas starp divām virsmām, iespējams, neatlasīs precīzu vērtību (dažu ierobežojumu dēļ), tāpēc, izmantojot GL_CLAMP_TO_EDGE, OpenGL vienmēr atgriezīs pareizu malas vērtību ikreiz, kad tiek apstrādāti dati starp virsmām.

5.3.2. Kubiskās tekstūras ielāde

Lai ielādētu tekstūru, tika uzrakstīta šādā funkcija, kas akceptē vektoru ar sešām tekstūras adresēm.

```
GLuint loadCubemap(vector<const GLchar*> faces)
{
    GLuint textureID;
    glGenTextures(1, &textureID);
    glActiveTexture(GL_TEXTURE0);

    int width, height;
    unsigned char* image;

    glBindTexture(GL_TEXTURE_CUBE_MAP, textureID);
    for (GLuint i = 0; i < faces.size(); i++)
    {
        image = SOIL_load_image(faces[i], &width, &height, 0, SOIL_LOAD_RGB);
        glTexImage2D(
            GL_TEXTURE_CUBE_MAP_POSITIVE_X + i, 0,
            GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, image
            );
    }
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_R, GL_CLAMP_TO_EDGE);
    glBindTexture(GL_TEXTURE_CUBE_MAP, 0);

    return textureID;
```

Būtībā gandrīz viss *cubemap* kods tika aprakstīts iepriekšējā sadaļā, bet tagad tas ir apvienots vienā pielietojamā funkcijā. Tādā gadījumā, pirms šī funkcija tiek izsaukta, vektoram, kas satur tekstūras, ir jādefinē atbilstoši tekstūru ceļi.

```
vector<const GLchar*> faces;
faces.push_back("right.jpg");
faces.push_back("left.jpg");
faces.push_back("top.jpg");
faces.push_back("bottom.jpg");
faces.push_back("back.jpg");
faces.push_back("front.jpg");
GLuint cubemapTexture = loadCubemap(faces);
```

Tagad ir ielādēta kubiskā tekstūra kā *cubemaps* un `cubemapTexture` ir tās ID. Tagad tā ir gatava uzklāšanai uz objekta.

5.3.3. Objekta attēlošana ar uzklātu kubisku tekstūru

Tika izveidots 3D vektors, kas apzīmēs nobīdi no centra:

```
glm::vec3 originOffset = glm::vec3(0.0f, 0.0f, 0.0f)
```

Tā kā šis vektors tiek izmantots sākumpunkta korekcijai, ir vēlams šo vektoru padot virsotņu ēnotājam. Ēnotāji vai shaders ir maza programma, ko parasti iekļauj lietojumprogrammas kodā kā atsevišķu failu, OpenGL ēnotāji tiek rakstīti GLSL valodā.

Lai sazinātos ar virsotņu un fragmentu ēnotāju, tiek izmantoti „*uniform"* vai vienoti mainīgie. Tie tiek deklarēti pašā ēnotāja kodā, lai to vērtības varētu mainīt no C++ koda.

```
uniform vec3 c;
```

Turpmāk, mainīgajā c tiks padots sākumpunkta nobīdes vektors `originOffset`. Vienoti mainīgie ēnotajā ir tikai lasāmi vai „read-only", tiem ir vienāda vērtība starp visiem apstrādes etapiem. Tos var mainīt tikai savā C++ programmā. Lai ar C++ kodu mainīt vienotu mainīgo, kas atrodas ēnotājā, tiek izmantota `glGetUniformLocation` komanda. Tā atgriež vienotā mainīgā adresi, ja mainīgais nav atrasts, tiek atgriezts -1. Ar `glUniform` var iestatīt šī vienotā mainīgā vērtību.

```
GLint loc = glGetUniformLocation(ProgramObject, "c");
if (loc != -1)
{
    glUniform1f(loc, originOffset);
}
```

Tādā veidā ēnotāja tiek padots sākumpunkta korekcijas vektors. Pienācis laiks to izmantot.
Ja objekta atrašanas vieta ir centrā, koordinātēs (0,0,0), katrs tā pozicionālais vektors ir arī virziena vektors no sākumpunkta. Šīs virziena vektors ir tas, kas vajadzīgs, lai iegūtu atbilstošo tekseļa krāsas vērtību. Šī iemesla dēļ tekstūras koordināšu vērtībās tiek definētas kā pozīcijas un korekcijas vektoru starpība.

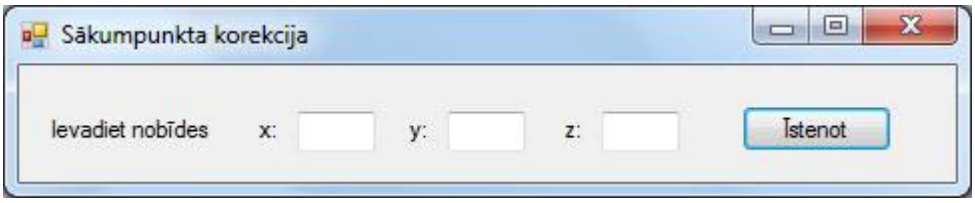
```
layout(location = 0) in vec3 position;
out vec3 TexCoords;

uniform mat4 projection;
uniform mat4 view;
uniform vec3 c;

void main()
{
    gl_Position = projection * view * vec4(position, 1.0);
    TexCoords = position - c;
}
```

Jāņem vērā to, ka virsotņu ēnotajā ienākošie pozicionāli vektori tiek ievietoti fragmentu ēnotāja kā tekstūras koordinātes. Fragmentu ēnotājs ņem šos datus kā ievadi.

```
in vec3 TexCoords;
out vec4 color;

uniform samplerCube furTexture;

void main()
{
    color = texture(furTexture, TexCoords);
}
```

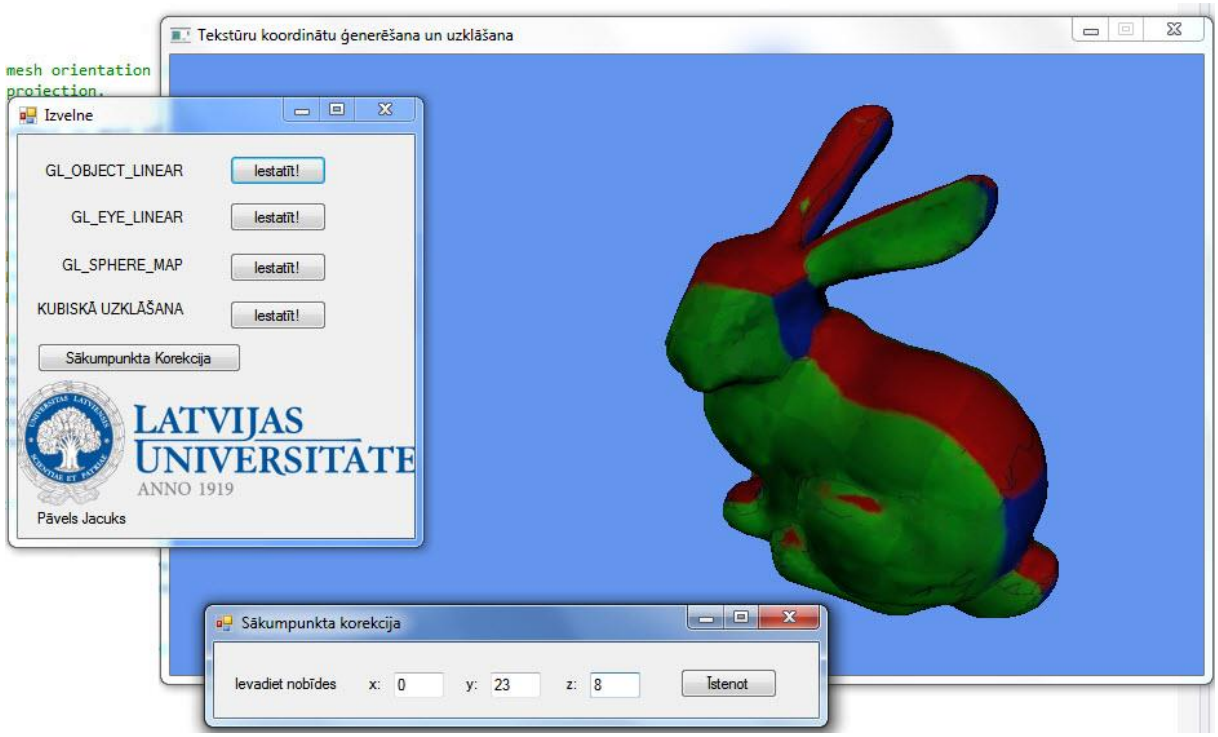
Fragmentu ēnotājs ir samērā vienkāršs. Virsotnes atribūta pozīcijas un korekcijas vektoru starpības tiek pieņemtas par tekstūras virziena vektoriem, un tās izmanto, lai iegūtu tekstūras vērtības no *cubemap*.

Tā kā lietotājam ir pašam jāievada sākumpunkta korekcijas vērtība, tika izveidota grafiskā saskarne skat. *att. 5.3.1.*



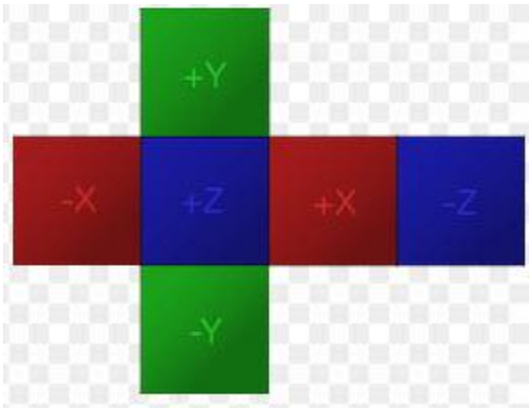
***att. 5.3.1.* Sākumpunkta korekcijas lietotāja saskarne.**

Rezultātā, pielietojot lietotāja ievadīto korekcijas vektoru, var attēlot objektu ar uzklātu tekstūru, kā redzams *att. 5.3.2.*



***att. 5.3.2.* Kubiskās uzklāšanas demonstrācija.**

Tika izmantotas sešas tekstūras, lai attēlotu virmas virzienu ar krāsu. Piemēram, virsma, kas vērsta uz augšu un leju, ir sarkanajā krāsā. Kubiskās tekstūras kompozīcija ir redzama *att. 5.3.3.*



att. 5.3.3. **Kubisko tekstūru struktūra.**

6. REZULTĀTI

Darba projektēšana un programmēšana tika uzsākta 2017. gada aprīlī un pie tā ir strādājis tikai šī bakalaura darba autors. Projektēšanas laikā autors ir iepazinies ar pieejamo literatūru, un galvenokārt balstījās uz OpenGL oficiālo dokumentāciju. Pašreiz realizēts vairāk no ieplānotas lietojumprogrammas funkcionalitātes, jo risinājumā ir iekļauta kubiskā uzklāšana. Tā ļauj apkrāsot modeli no visām pusēm ar dažādām tekstūrām, skat. ekrānuzņēmumus pielikumā.. Lietojumprogrammas paplašināšana un modificēšana turpinās.

Lietojumprogrammas iespēju saraksts:

- Tekstūras uzklāšana.
- Tekstūras koordinātu ģenerēšana izmantojot GL_OBJET_LINEAR.
- Tekstūras koordinātu ģenerēšana izmantojot GL_EYE_LINEAR.
- Tekstūras koordinātu ģenerēšana izmantojot GL_SPHERE_MAP.
- Kubiskās tekstūras uzklāšana un koordinātu ģenerēšana.
- Objekta sākumpunkta korekcija kubiskai uzklāšanai.

7. DARBA KLASES

Tabulā 7.1 ir uzskaitītas un aprakstītas visas darba klases.

tabula 7.1 **Klašu apraksti**

Klase	**Apraksts**
Form1	Lietotāja saskarnes klase tekstūras ģenerēšanas metodes izvēlnei.
Form2	Lietotāja saskarnes klase sākumpunkta korekcijai
Glute	Galvenais izpildes fails.

8. SECINĀJUMI

Darba autoram bija liels izaicinājums. OpenGL API apgūšana, pēc autora domām, ir pietiekami sarežģīta, kad tiek uzstādīti laika termiņi. Tomēr pozitīvie rezultāti un jauniegūtā padziļināta OpenGL saprašana virza autoru turpināt nodarboties ar ievadā nosaukto projektu „Mošķu proceduārā ģenerēšana".

OpenGL ir brīvi pieejama oficiālā dokumentācija. Tās saturs atbilst sarežģītības līmenim, kāds aplūkots darbā, taču konkrētā API lietošana nepieredzējušam programmētajam var radīt grūtības darba uzsākšanai.

Savukārt objektu zīmēšanas process uz OpenGL ir ļoti pielāgojams programmētāja vēlmēm.

IZMANTOTĀ LITERATŪRA

1. Grāmatas

[1] Mark Segal „The OpenGL Graphics System: A Specification v2.01" Kurt Akeley 2006

[2] Tom McReynolds „Advanced Graphics Programming Using OpenGL" David Blythe 2005 Elsevier Inc

[3] Paul Martz „OpenGL Distilled" 2006

2. Konferenču tēzes

[5] A. Sheffer, E. de Sturler „Non-Distorted Texture Mapping Using Angle Based Flattening"

3. Elektroniskie informāciju avoti

[4] „Tex Gen, the Texture Matrix, and Projected Textures"

https://www.usbid.com/datasheets/usbid/2001/2001-q1/gdc2k_texgenntexmatrix.pdf

[6] „Texture Mapping"

http://www.inf.pucrs.br/flash/tcg/aulas/texture/texmap.pdf

[7] „OpenGL Texture-Mapping Made Simpler"

http://web.engr.oregonstate.edu/~mjb/cs553/Handouts/Texture/texture.pdf

[8] „Texture Coordinate Generation"

http://resumbrae.com/ub/dms424_s05/10/print.html

[9] „TexGen funkcijas specifikācija"

https://www.khronos.org/registry/OpenGL-Refpages/gl2.1/xhtml/glTexGen.xml

[10] „OpenGL® 4.5 Reference Pages"

https://www.khronos.org/registry/OpenGL-Refpages/gl4/

[11] „OpenGL Cube Map Texturing"

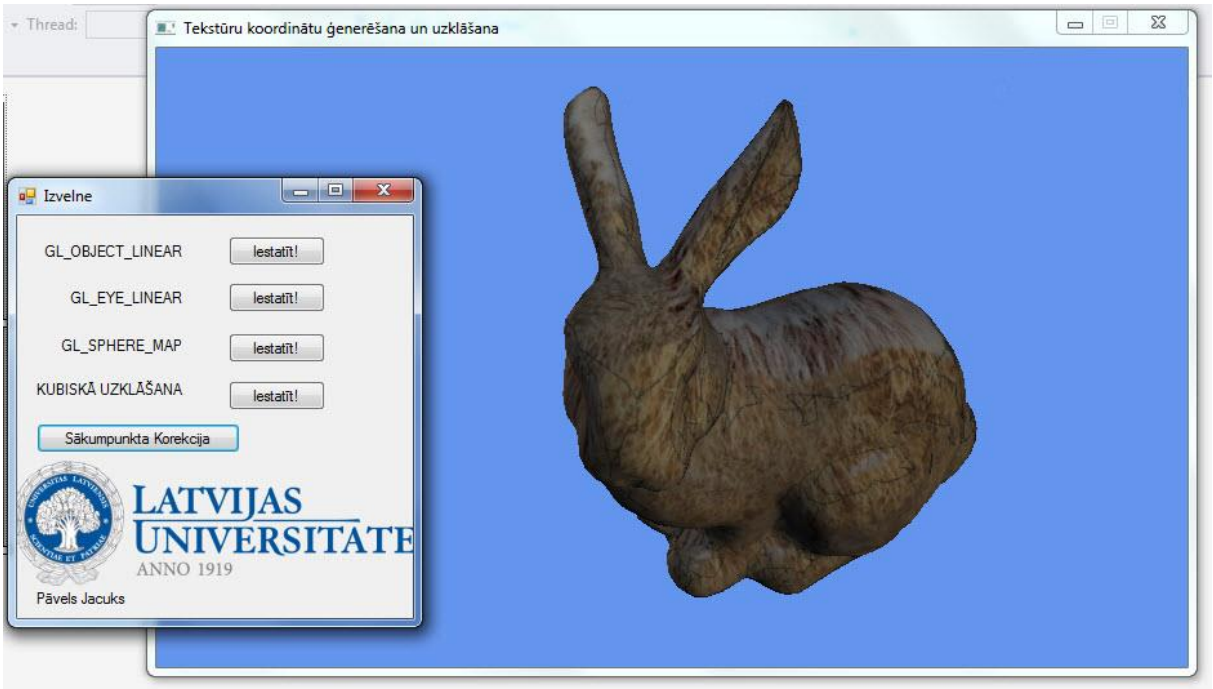
http://www.nvidia.com/object/cube_map_ogl_tutorial.html

[12] „Cube Map: Skybox and Enviroment mapping”

http://antongerdelan.net/opengl/cubemaps.html

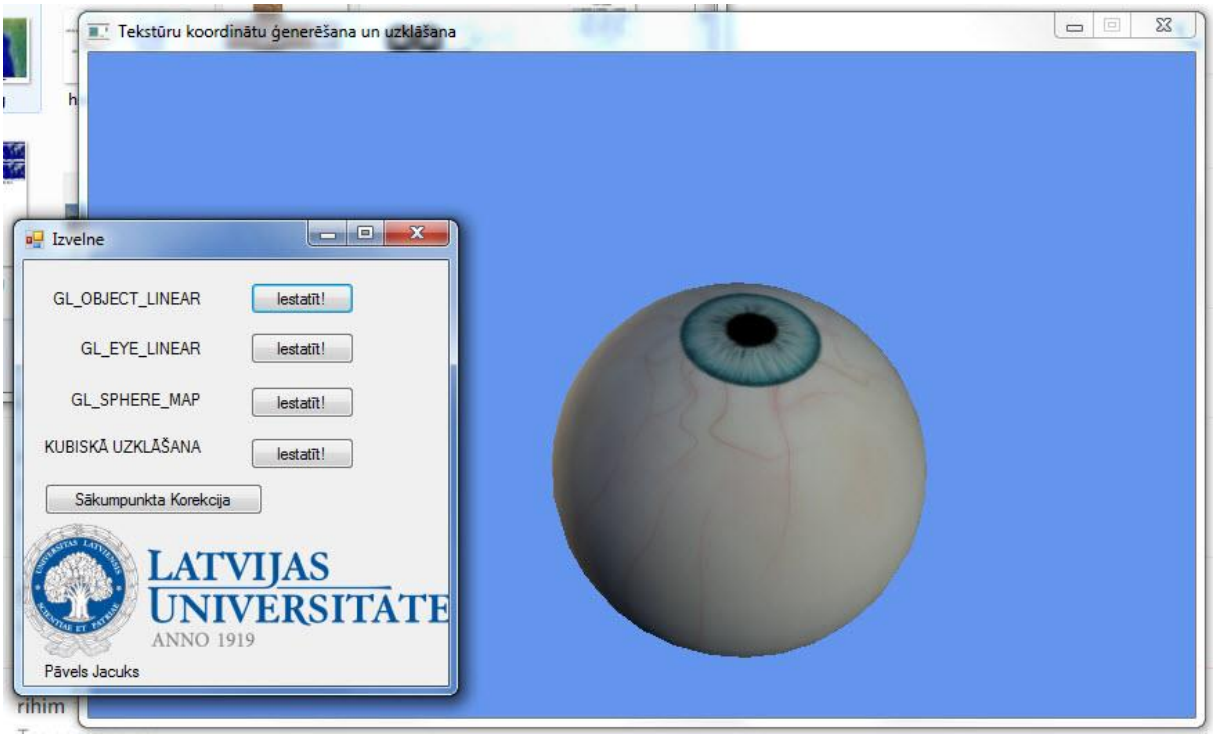
PIELIKUMI

1. Kubiskā uzklāšana zaķim.

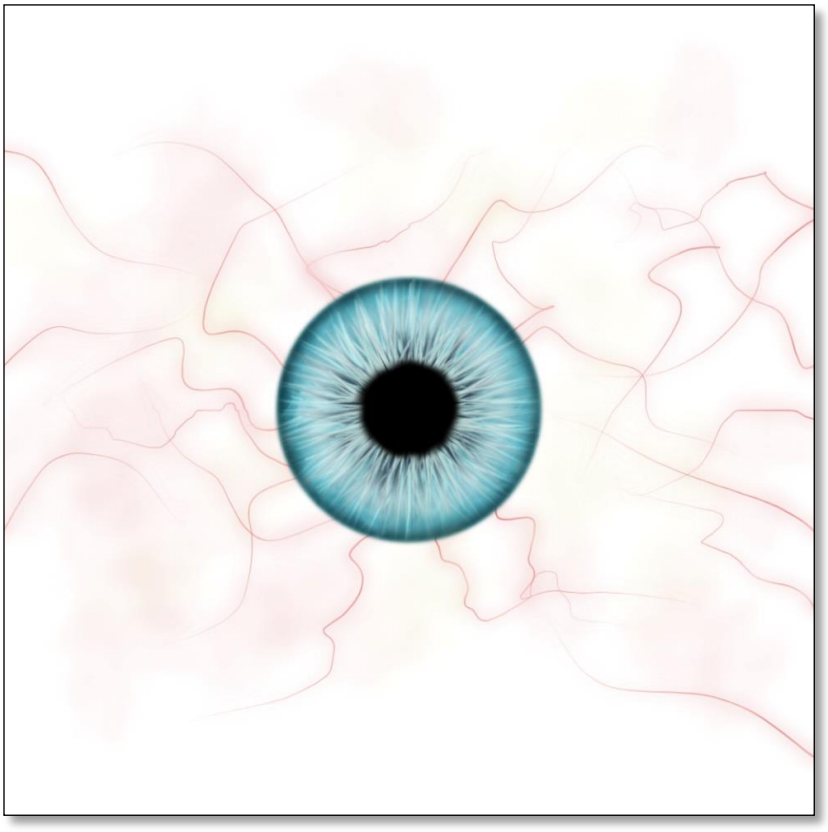


2. Izmantotā kubiskās tekstūras struktūra.

3. GL_OBJECT_LINEAR



4. Izmantotā tekstūra.



1. **Pirmkods**

```
2.        #include <iostream>
3.        #include <string>
4.        #include <vectors>
5.        #include <algorithm>
6.        using namespace std;
7.        #include <GL/glut.h>
8.        //#include <GL/glew.h>
9.        #include <GLFW/glfw3.h>
10.
11.       #include "Shader.h"
12.       #include "Camera.h"
13.       #include "Model.h"
14.
15.       #include <SOIL.h>
16.       #include <stdlib.h>
17.       #include <stdio.h>
18.       #include <assimp/Importer.hpp>
19.       #include <assimp/scene.h>
20.       #include <assimp/postprocess.h>
21.       #define      stripeImageWidth 32
22.       GLubyte stripeImage[4 * stripeImageWidth];
23.       int width, height;
24.       #ifdef GL_VERSION_1_1
25.       static GLuint texName;
26.       GLuint tex_2d;
27.       unsigned char* image;
28.       Form1 GUIselect;
29.       Form2 GUIset;
30.       #endif
31.
32.       GLuint loadCubemap(vector<const GLchar*> faces);
33.
34.
35.
36.
37.       void makeStripeImage(void)
38.       {
39.            int j;
40.
41.            for (j = 0; j < stripeImageWidth; j++) {
42.                  stripeImage[4 * j] = (GLubyte)((j <= 4) ? 255 : 0);
43.                  stripeImage[4 * j + 1] = (GLubyte)((j>4) ? 255 : 0);
44.                  stripeImage[4 * j + 2] = (GLubyte)0;
45.                  stripeImage[4 * j + 3] = (GLubyte)255;
46.            }
47.       }
48.
49.       float angle;
50.       /*  planes for texture coordinate generation  */
51.       static GLfloat xequalzero[] = { 0.8, 0.0, 0.0, 1.0 };
52.       static GLfloat yequalzero[] = { 0.0, 0.8, 0.0, 1.0 };
53.       static GLfloat slanted[] = { 1.0, 1.0, 1.0, 0.0 };
54.       static GLfloat *currentCoeff;
55.       static GLenum currentPlane;
56.       static GLint currentGenMode;
57.
58.
59.       GLuint loadCubemap(vector<const GLchar*> faces)
60.       {
61.            GLuint textureID;
62.            glGenTextures(1, &textureID);
```

```
63.
64.           int width, height;
65.           unsigned char* image;
66.
67.           glBindTexture(GL_TEXTURE_CUBE_MAP, textureID);
68.           for (GLuint i = 0; i < faces.size(); i++)
69.           {
70.                 image = SOIL_load_image(faces[i], &width, &height, 0,
      SOIL_LOAD_RGB);
71.                 glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X + i, 0, GL_RGB, width,
      height, 0, GL_RGB, GL_UNSIGNED_BYTE, image);
72.                 SOIL_free_image_data(image);
73.           }
74.           glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
75.           glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
76.           glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S,
      GL_CLAMP_TO_EDGE);
77.           glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T,
      GL_CLAMP_TO_EDGE);
78.           glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_R,
      GL_CLAMP_TO_EDGE);
79.           glBindTexture(GL_TEXTURE_CUBE_MAP, 1);
80.
81.           return textureID;
82.     }
83.
84.
85.     void init(void)
86.     {
87.
88.           glClearColor(0.65, 0.65, 0.95, 0);
89.           glEnable(GL_DEPTH_TEST);
90.           glShadeModel(GL_SMOOTH);
91.           angle = 45;
92.           makeStripeImage();
93.
94.           image = SOIL_load_image("C:\\Users\\Regents\\Documents\\Visual Studio
      2013\\Projects\\OpenGL\\Debug\\texture.bmp", &width, &height, 0,
      SOIL_LOAD_RGB);
95.           std::cout << "null: " << !image << std::endl;
96.           std::cout << "Max size: " << GL_MAX_TEXTURE_SIZE << std::endl;
97.           std::cout << "Width: " << width << std::endl;
98.           std::cout << "Height: " << height << std::endl;
99.           std::cout << "Obj: " << SOIL_last_result() << std::endl;
100.          std::cout << "Obj: " << tex_2d << std::endl;
101.          SOIL_last_result();
102.
103.
104.
105.    #ifdef GL_VERSION_1_1
106.          //glGenTextures(1, &texName);
107.          //glBindTexture(GL_TEXTURE_1D, texName);
108.    //    glGenTextures(1, &tex_2d);
109.    //    glBindTexture(GL_TEXTURE_2D, tex_2d);
110.    #endif
111.          //glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
112.          //glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
113.          //glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
114.    #ifdef GL_VERSION_1_1
115.          //glTexImage1D(GL_TEXTURE_1D, 0, GL_RGBA, stripeImageWidth, 0,
116.                //GL_RGBA, GL_UNSIGNED_BYTE, stripeImage);
117.          //glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, 450, 450, 0,
118.                //GL_RGBA, GL_UNSIGNED_BYTE, tex_2d);
119.    #else
```

```
120.      //    glTexImage1D(GL_TEXTURE_1D, 0, 4, stripeImageWidth, 0,
121.            //     GL_RGBA, GL_UNSIGNED_BYTE, stripeImage);
122.            //glTexImage1D(GL_TEXTURE_1D, 0, 4, stripeImageWidth, 0,
123.            //     GL_RGBA, GL_UNSIGNED_BYTE, stripeImage);
124.      #endif
125.
126.            glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
127.            currentCoeff = xequalzero;
128.            currentGenMode = GL_OBJECT_LINEAR;
129.            currentPlane = GL_OBJECT_PLANE;
130.            glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, currentGenMode);
131.            glTexGenfv(GL_S, currentPlane, currentCoeff);
132.            glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, currentGenMode);
133.            glTexGenfv(GL_T, currentPlane, yequalzero);
134.            glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_OBJECT_LINEAR);
135.            glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_OBJECT_LINEAR);
136.            glEnable(GL_TEXTURE_2D);
137.            glEnable(GL_COLOR_MATERIAL);
138.            glEnable(GL_TEXTURE_GEN_S);
139.            glEnable(GL_TEXTURE_GEN_T);
140.            //glEnable(GL_TEXTURE_1D);
141.            glEnable(GL_CULL_FACE);
142.            glEnable(GL_LIGHTING);
143.            glEnable(GL_LIGHT0);
144.            glEnable(GL_AUTO_NORMAL);
145.            glEnable(GL_NORMALIZE);
146.            glFrontFace(GL_CW);
147.            glCullFace(GL_BACK);
148.            glDeleteTextures(1, &tex_2d);
149.            glMaterialf(GL_FRONT, GL_SHININESS, 64.0);
150.      }
151.
152.      void display(void)
153.      {
154.            glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
155.            glEnable(GL_TEXTURE_2D);
156.            glEnable(GL_TEXTURE_GEN_S);
157.            glEnable(GL_TEXTURE_GEN_T);
158.            glColor3d(1, 1, 1);
159.            glPushMatrix();
160.            glRotatef(angle, 1.0, 0.0, 0.0);
161.            int i = GUI1.texgenMode;
162.      #ifdef GL_VERSION_1_1
163.            //glBindTexture(GL_TEXTURE_1D, texName);
164.      //    glBindTexture(GL_TEXTURE_2D, tex_2d);
165.      #endif
166.      //    glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_OBJECT_LINEAR);
167.      //    glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_OBJECT_LINEAR);
168.
169.            if (i = 1){
170.                  setFixed(1);
171.            }
172.            else if (i = 2)
173.            {
174.                  setFixed(2);
175.            }
176.            else if (i = 3)
177.            {
178.                  setFixed(3);
179.            }
180.            else if (i = 4)
181.            {
182.                  setCubemapping();
183.            }
```

```
184.            CAssimpModel.Render();
185.            glPopMatrix();
186.            glFlush();
187.      }
188.
189.      void reshape(int w, int h)
190.      {
191.            glViewport(0, 0, (GLsizei)w, (GLsizei)h);
192.            glMatrixMode(GL_PROJECTION);
193.            glLoadIdentity();
194.            if (w <= h)
195.                  glOrtho(-3.5, 3.5, -3.5*(GLfloat)h / (GLfloat)w,
196.                  3.5*(GLfloat)h / (GLfloat)w, -3.5, 3.5);
197.            else
198.                  glOrtho(-3.5*(GLfloat)w / (GLfloat)h,
199.                  3.5*(GLfloat)w / (GLfloat)h, -3.5, 3.5, -3.5, 3.5);
200.            glMatrixMode(GL_MODELVIEW);
201.            glLoadIdentity();
202.      }
203.
204.      void keyboard(unsigned char key, int x, int y)
205.      {
206.            switch (key) {
207.            case 'e':
208.            case 'E':
209.                  currentGenMode = GL_EYE_LINEAR;
210.                  currentPlane = GL_EYE_PLANE;
211.                  glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, currentGenMode);
212.                  glTexGenfv(GL_S, currentPlane, currentCoeff);
213.                  glutPostRedisplay();
214.                  break;
215.            case 'd':
216.            case 'D':
217.                  angle += 1;
218.                  glutPostRedisplay();
219.                  break;
220.            case 'a':
221.            case 'A':
222.                  angle -= 1;
223.                  glutPostRedisplay();
224.                  break;
225.            case 'o':
226.            case 'O':
227.                  currentGenMode = GL_OBJECT_LINEAR;
228.                  currentPlane = GL_OBJECT_PLANE;
229.                  glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, currentGenMode);
230.                  glTexGenfv(GL_S, currentPlane, currentCoeff);
231.                  glutPostRedisplay();
232.                  break;
233.            case 's':
234.            case 'S':
235.                  currentCoeff = slanted;
236.                  glTexGenfv(GL_S, currentPlane, currentCoeff);
237.                  glutPostRedisplay();
238.                  break;
239.            case 'x':
240.            case 'X':
241.                  currentCoeff = xequalzero;
242.                  glTexGenfv(GL_S, currentPlane, currentCoeff);
243.                  glutPostRedisplay();
244.                  break;
245.            case 'z':
246.            case 'Z':
247.                  currentGenMode = GL_SPHERE_MAP;
```

```
248.                  currentPlane = GL_OBJECT_PLANE;
249.                  glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, currentGenMode);
250.                  glTexGenfv(GL_S, currentPlane, currentCoeff);
251.                  glutPostRedisplay();
252.                  break;
253.           case 'c':
254.           case 'C':
255.                  currentGenMode = GL_OBJECT_LINEAR;
256.                  currentPlane = GL_OBJECT_PLANE;
257.                  glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, currentGenMode);
258.                  glTexGenfv(GL_S, currentPlane, xequalzero);
259.                  glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, currentGenMode);
260.                  glTexGenfv(GL_T, currentPlane, yequalzero);
261.                  glutPostRedisplay();
262.
263.                  break;
264.           case 27:
265.                  exit(0);
266.                  break;
267.           default:
268.                  break;
269.
270.                  //GL_SPHERE_MAP
271.           }
272.     }
273.
274.     void setCubemapping()
275.     {
276.           furShader.Use();
277.           glm::mat4 view = glm::mat4(camera.GetViewMatrix());
278.           glm::mat4 projection = glm::perspective(camera.Zoom, (float)screenWidth /
      (float)screenHeight, 0.1f, 100.0f);
279.           glUniformMatrix4fv(glGetUniformLocation(furShader.Program, "view"), 1,
      GL_FALSE, glm::value_ptr(view));
280.           glUniformMatrix4fv(glGetUniformLocation(furShader.Program, "projection"),
      1, GL_FALSE, glm::value_ptr(projection));
281.           glUniform2fv(glGetUniformLocation(furShader.Program, "c"),
      GUI2.originOffset)
282.           glActiveTexture(GL_TEXTURE1);
283.     }
284.
285.     void setFixed(int genType)
286.     {
287.           glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
288.           glEnable(GL_TEXTURE_2D);
289.
290.           glGenTextures(1, &tex_2d);
291.           glBindTexture(GL_TEXTURE_2D, tex_2d);
292.
293.           glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB,
294.                  GL_UNSIGNED_BYTE, image);
295.           //SOIL_free_image_data(image);
296.           glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
297.           glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
298.           glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
299.           glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
300.
301.           if (i = 1){
302.                  currentGenMode = GL_OBJECT_LINEAR;
303.                  currentPlane = GL_OBJECT_PLANE;
304.                  glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, currentGenMode);
305.                  glTexGenfv(GL_S, currentPlane, xequalzero);
306.                  glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, currentGenMode);
307.                  glTexGenfv(GL_T, currentPlane, yequalzero);
```

```
                glutPostRedisplay();
        }
        else if (i = 2)
        {
                currentGenMode = GL_EYE_LINEAR;
                currentPlane = GL_EYE_PLANE;
                glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, currentGenMode);
                glTexGenfv(GL_S, currentPlane, currentCoeff);
                glutPostRedisplay();
        }
        else if (i = 3)
        {
                currentGenMode = GL_SPHERE_MAP;
                currentPlane = GL_OBJECT_PLANE;
                glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, currentGenMode);
                glTexGenfv(GL_S, currentPlane, currentCoeff);
                glutPostRedisplay()
        }
}

int main(int argc, char** argv)
{



        Shader shader("shaders/default.vs", "shaders/default.frag");
        Shader furShader("shaders/cubemapping.vs", "shaders/cubemapping.frag");
        CAssimpModel.LoadModelFromFile("bunny.obj")
        glutInit(&argc, argv);
        Form1 GUI1;
        Form2 GUI2;

        vector<const GLchar*> faces;
        faces.push_back("fur/right.jpg");
        faces.push_back("fur/left.jpg");
        faces.push_back("fur/top.jpg");
        faces.push_back("fur/bottom.jpg");
        faces.push_back("fur/back.jpg");
        faces.push_back("fur/front.jpg");
        GLuint cubemapTexture = loadCubemap(faces);



        glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);
        glutInitWindowSize(256, 256);
        glutInitWindowPosition(100, 100);
        glutCreateWindow(argv[0]);
        init();
        glutDisplayFunc(display);

        glutReshapeFunc(reshape);
        glutKeyboardFunc(keyboard);
        glutMainLoop();

        return 0;
}
```

Bakalaura darbs „Tekstūras koordinātu ģenerēšanas algoritmi 3D modeļiem” izstrādāts LU Datorikas fakultātē.

Ar savu parakstu apliecinu, ka pētījums veikts patstāvīgi, izmantoti tikai tajā norādītie informācijas avoti un iesniegtā darba elektroniskā kopija atbilst izdrukai.

Autors: Pāvels Jacuks

Rekomendēju/nerekomendēju darbu aizstāvēšanai

Vadītājs: docents Kārlis Freivalds 25.05.2017.

Recenzents: profesors Juris Vīksna

Darbs iesniegts Datorikas fakultātē 29.05.2017.

Dekāna pilnvarotā persona: metodiķe Ārija Sproģe

Darbs aizstāvēts bakalaura gala pārbaudījuma komisijas sēdē

Komisijas sekretāre:

LATVIJAS UNIVERSITĀTE

BAKALAURA DARBS

RĪGA 2017