Tallinna Ülikool

Informaatika Instituut

WebGL'i kasutamine interaktiivsete graafikarakenduste loomiseks veebilehitsejas: õppematerjal

Bakalaureusetöö

Autor: Raner Piibur

Juhendaja: Jaagup Kippar

Autor: ..... ” ..... ”2015

Juhendaja: ..... ” ..... ”2015

Instituudi direktor: ..... ” ..... ”2015

Tallinn 2015

Autorideklaratsioon

Deklareerin, et käesolev bakalaureusetöö on minu töö tulemus ja seda ei ole kellegi teise poolt varem kaitsmisele esitatud. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on viidatud.

.... (kuupäev) (autor)

Sisukord

Sissejuhatus

Ei ole kahtlustki, et tänapäeval toimub suurem osa tegevusest arvuti taga veebilehitsejates. Kasvanud on kompleksse interaktiivse visualisatsiooni kuvamise nõudlus veebis. Tõenäoliselt enamikul veebiprogrammeerijatel on kokkupuude HTML5'iga ja vähemalt mingi arusaam SVG'ist. HTML5 ja SVG on standardiseeritud meetodid, millega graafikat genereerida ja seega on ka laialdaselt kasutatud. Nende meetodite puudus on kiirus. Veidigi keerulisema visualisatsiooni korral jääb jõudlusest puudu ja visualisatsioon oleks vaja kiirendada.

Lisaks on algajale graafika programmeerimise huvilisele OpenGL'i õppematerjal küllaltki keeruline ja aeganõudev, sest materjaliga kaasa tulev lähtekood on olenevalt operatsioonisüsteemist vaja uuesti kompileerida. Võib juhtuda, et kasutatakse valet arenduskeskkonda ja pole installeeritud vajalikke teeke, et kood üldse kompileeruks, mistõttu annavad paljud huvilised lihtsalt alla. Tegeleda on vaja ka viitadega, mis on C keeles kohane. Õppimisele lisandub väga palju müra, mis raskendab niigi keerulise teema õppimist kordades.

Mõlemad probleemid suudab lahendada WebGL, mis võimaldab meil graafikakiirendi abil veebilehitsejas graafikat renderdada ja mille ainukesteks nõueteks on see, et kasutatakse kaasaegset veebilehitsejat ning osatakse veidikene Javascripti.

Töö teema valik tugineb autori isiklikust huvist graafika renderdamise ja graafikamootorite vastu. Osaledes kursusel „Graafika ja muusika programmeerimine" (ainekood IFI6028) tehti tutvust Three.js teegiga. Three.js on javascripti teek, mis võimaldab luua ja kuvada animeeritud 3D graafikat veebilehitseja, kasutades graafika renderdamiseks WebGL'i. Teek abstraheerib terve WebGL osa ja sobib hästi algajale, kes pole 2D/3D graafikaga üldse kokku puutunud. Autor leiab aga, et iga arendaja, kes tahab tõsisemalt tegeleda graafika programmeerimisega, peaks omama mingit ülevaadet, kuidas graafika renderdamine abstraheerimata madalatasemelise API'iga toimib. Kui otseselt ise WebGL'i või OpenGL'i ei kasutata, siis vähemalt aitab see arendajal teha tulevikus erinevate tehnoloogiate vahel valiku või siis olemasolevaid mängumootoreid ja raamistikke vastavalt vajadusele modifitseerida.

Hetkel leidub WebGL'i õpetavat materjali vähe. Paljud ingliskeelsed õppematerjalid kasutavad õpetamiseks Three.js teeki, mis eemaldab otsese kokkupuute WebGL'ga. Alternatiivina võib kasutada ka OpenGL 3.3+ ja OpenGL ES 2.0 ingliskeelseid materjale, kuid näitekood ei ole Javascripti keeles ja Javascripti ümber tõlkides ei pruugi need ka veebilehitsejas töötada. Eestikeelne materjal WebGL'i kohta puudub. Käesoleva bakalaureusetöö eesmärgiks on luua WebGL'i ja graafika

programmeerimist käsitlev õppematerjal, mis sobiks nii „Graafika ja muusika programmeerimine“ kursusel õpetamiseks kui ka iseseisvaks õppimiseks.

Mõisted

API (*Application Programming Interface*) – Hulk toimetusi, protokolle ja töövahendeid, mille abil arendatakse tarkvara.

OpenGL (*Open Graphics Library*) - Madalatasemeline API, mille abil suhelda graafikaprotsessoriga. OpenGL on kõige rohkem kasutatud 2D ja 3D graafika API vastavas valdkonnas.

OpenGL ES *(OpenGL for Embedded Systems)* – Alamhulk OpenGL API'st. Leiab kasutust mobiilsetes seadmetes.

WebGL *(Web Graphics Library)* – Baseerub OpenGL ES API'il. Jooksutatakse HTML5 Canvas elemendis. On toetatud kõikides suuremates veebilehitsejates (Internet Explorer'is alates versioonist 11).

Renderdamise järjekord *(WebGL pipeline/Rendering pipeline)* – Renderdamisprotsessi erinevad etapid graafikaprotsessoris. Tavalised etapid on näiteks: värvi ja valguse kalkuleerimine GLSL/HLSL programmide abil, perspektiivi projekteerimine, akna pügamine (aknast väljas olevate objektide või nende osade eemaldamine) ja renderdamine.

GLSL *(OpenGL Shading Language)* – Baseerub ANSI C keelel. Keelt on laiendatud vektor ja maatriks tüüpidega ning kohendatud paralleelselt jooksmiseks. GLSL'i kasutatakse renderdamise järjekorras varjundprogrammides.

Varjundaja *(Shader)* – Programm, mis jookseb renderdamise järjekorras kindlal etapil. OpenGL'il baseeruvates API'des kirjutatakse programm GLSL keeles, Direct3D puhul HLSL keeles.

Tipuvarjundaja *(Vertex Shader)* – Käsitleb tippude töötlemist. Tavaliselt toimub etapis tippude transformeerimine objektruumist projektsioonruumi e. kolmemõõtmeline tipp projekteeritakse kahemõõtmelisse ruumi (olenevalt tehnikast, mida kasutatakse, ei pruugi see muidugi alati nii olla). Lagivarjundajas toimub ka muude atribuutide eelsätestamine järgmiste etappide jaoks.

Pikslivarjundaja *(Fragment Shader)* – Käsitleb piksli/fragmendi töötlemist. Etapp on viimane programmeeritav etapp. Lihtsamal juhul toimub selles varjundajas vaid fragmendi värvi määramine, mis siis ekraanile kuvatakse. Tavaliselt toimub etapis ka valgusarvutused, varjude lisamine jms.

1. WebGL'i ülevaade

Peatüki eesmärk on saada WebGL'st ülevaade, mis võimaldaks meil mõista, mida WebGL endast täpsemalt kujutab. Toome välja WebGL'i API'iga kaasnevad lisaprobleemid, mida tavaliselt Javascriptis üldiselt ei arvestata – spetsiifilise graafikaprotsessori funktsionaalsuse puudumine või vead draiverites. Lühidalt tutvume peatükis API ajalooga, mis võimaldab paremini prognoosida tema jätkusuutlikkust ka tulevikus.

1.1. WebGL'i tutvustus

WebGL on mitmeplatvormne API loomaks 2D ja 3D graafikat veebilehitsejas. WebGL'le saab ligi kasutades HTML5 elementi *Canvas.* API baseerub OpenGL ES 2.0 spetsifikatsioonil, mis on OpenGL alamhulk ja kasutab graafikaprotsessoril jooksvate programmeeritavate etappide tarbeks OpenGL varjundamiskeelt (*OpenGL Shading Language*), mida lühidalt nimetatakse GLSL (Khronos Group, 2011). GLSL keele abil on võimalik kirjutada renderdamise järjekorras erinevatel etappidel jooksvaid programme (varjundajaid): tipuvarjundajad ja pikslivarjundajad (Khronos Group, 2009).

Tänu ANGLE (*Almost Native Graphics Layer Engine*) teegile ei ole graafikaprotsessoril OpenGL tuge vaja, vaid OpenGL ES või WebGL API kutsumused on võimalik selle teegi abil transleerida DirectX 9 või DirectX 11 API kutsumusteks. ANGLE on kasutusel Google Chrome ja Mozilla Firefox veebilehitsejates.

Kuna WebGL baseerub OpenGL ES API'il, mis on OpenGL alamhulk, toetavad seda kõik tänapäevased graafikaprotsessorid. Sarnasuste tõttu ja tänu asm.js Javascripti alamkeelele on võimalik näiteks C++ keelt ja OpenGL API'it kasutav töölauarakendus kompileerida koodiks, millest veebilehitseja aru saab. Asm.js on Javascript'i range alamhulk, mis võimaldab määrata muutujate tüübid, mida muidu selles keeles määrata ei saa, võimaldades seega staatilises keeles kirjutatud rakendus kompileerida Javascripti ja OpenGL'i kutsumused WebGL'i kutsumusteks.

WebGL API'ile saab küll ligi kasutades dünaamilist skriptimiskeelt Javascript, kuid WebGL iseenesest on madalatasemeline API. Isegi lihtsamate tegevuste tarbeks tuleb kirjutada üpriski palju koodi. Tuleb kompileerida ja ühendada varjundajaid ning sööta erinevatele varjundprogrammidele just neile vajalikke muutujaid. Objektide transformeerimiseks ja animeerimiseks on vaja teostada ka vektor ja maatriks arvutusi ning tunda on vaja ka optikat.

Khronos Group (Khronos Group, 2011) toob välja mitmeid WebGL'i eeliseid:

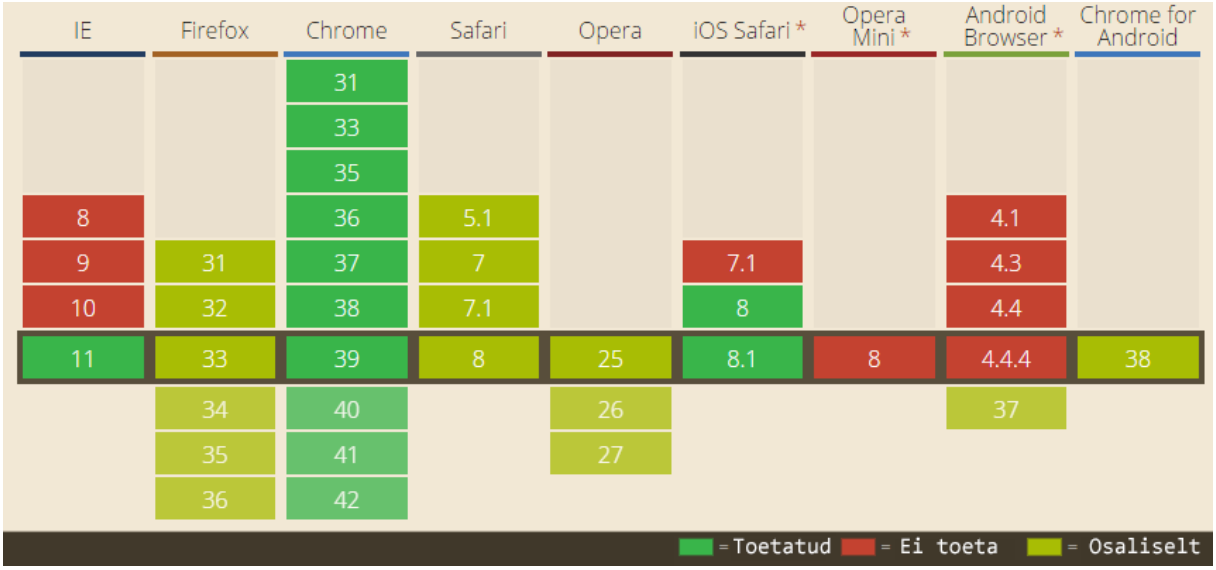
- Baseerub laialt kasutatud OpenGL API'ile.
- Toetatud kõikides populaarsemates veebilehitsejates ja jookseb kõikides peamistes operatsioonisüsteemides.
- Integreeritud HTML5 elemendiga ja seeläbi on võimalik suhelda teiste HTML elementidega. Samuti on võimalik sellele elemendile lisada kuulareid.
- Töötab pistikprogrammita.
- Toob riistvara poolt kiirendatud graafika veebilehitsejasse.
- WebGL kirjutatakse Javascript'i keeles ja seega ei ole vaja koodi kompileerida. See omadus on hea nii prototüübi loomiseks, kui ka algajatele õppimiseks, sest võimaldab kiiresti tulemust vaadata ja vigasid siluda.

WebGL sarnaselt OpenGL'ga võimaldab kasutada ka erinevaid laiendusi, mille abil on võimalik kasutada uusimaid tehnoloogiaid, mis ei pruugi kasutuses oleva versiooni spetsifikatsioonides kirjas olla.

1.2. WebGL'i toetavad veebilehitsejad ja graafikaprotsessorid

WebGL on toetatud praktiliselt kõikides uuemates populaarsetes veebilehitsejates (vt Joonis 1). Osaline toetus tuleneb suuresti sellest, et kõikidel kasutajatel ei pruugi olla ligipääs WebGL'le. Tihtilugu on probleem vanas graafikaprotsessoris, millel puudub mingi funktsionaalsus või graafikakiirendi draiverites peituvates vigades.

Google Chrome on lahendanud vanade GPU'ide probleemi litsentseeritud tarkvara *SwiftShader* abil, mis emuleerib graafikakiirendit. Kuigi tarkvara ei ole nii kiire kui riistvara, võimaldab ta siiski vana riistvaraga kasutajatel WebGL rakendust näha ja kasutada (Bauman, Salomon, 2012). See fakt on ka põhjus, miks Chrome'il on juba pikemat aega parim WebGL toetus.



Joonis 1 Toetatud veebilehitsejad (Deveria, 2014)

Google Chrome'is on võimalik saada informatsiooni GPU toetatud funktsioonide kohta kirjutades aadressireale ***chrome://gpu/***. Avanev infoleht on eriti kasulik nutitelefonides, et mugavalt teada saada, mis laiendused on mingis mobiilses GPU'is toetatud.

Kaks peamist veebilehitsejat Mozilla Firefox ja Google Chrome, mis juba pikemat aega toetavad WebGL'i on koostanud musta nimekirja protsessoritest, mis ei toeta või millel esineb probleeme WebGL sisu näitamisega. Mõned GPU'id on keelatud ka turvakaalutlustel. Autor leiab, et kuna nimekiri on küllaltki suur ja pidevalt muutuv, on mõistlikum pakkuda nimekirjade aadressid:

- Khronos Group (Khronos Group, 2014) must nimekiri (ei pruugi olla täiesti ajakohane, seega tasub olla ettevaatlik) - https://www.khronos.org/webgl/wiki/BlacklistsAndWhitelists
- Google Chromiumi (Chromiumi autorid, 2013) must nimekiri (muutub pidevalt, pakkudes ajakohast informatsiooni) - https://code.google.com/p/chromium/codesearch#chromium/src/gpu/config/software_rendering_list_json.cc
- Mozilla Firefoxi (Mozilla, 2013) hästi dokumenteeritud must nimekiri - https://wiki.mozilla.org/Blocklisting/Blocked_Graphics_Drivers

Nimekirjas on päris palju graafikaprotsessoreid, mis on iseenesest üpriski vanad ja autor leiab, et sellise riistvaraga kasutajad ei kuulu ka tõenäoliselt WebGL kasutajate hulka.

WebGL API on ligilähedane OpenGL 2.0 API'ile, mis ilmus 2004. aasta II. pooles ja teoreetiliselt peaks kõik lauaarvuti videokaardid märgitud ajastust WebGL'i toetama. Küll võib aga probleeme tekkida

draiveritega või vigadega veebilehitsejas endas. Lisaks peab mainima, et WebGL's leidub laiendusi, mis on pärit OpenGL'st 3.0 (Akeley, Segal, 2008), seega spetsiifilisi laiendusi kasutades kitseneb kindlasti toetatud arvutite hulk.

Androidi ja Apple'i seadmed, mis toetavad joonisel 1 väljatoodud WebGL toega mobiilseid veebilehitsejat, peaksid omama ka WebGL toega graafikaprotsessoreid. Küll aga näitame koheselt, et tegelikult see nii ei ole.

Androidi arendajalehelt leiab info, et alates Android Froyost (API tase 8) on olemas OpenGL ES 2.0 tugi (Google, 2014). AppBrain (AppBrain, 2014) on välja toonud statistika, mis näitab selgelt, et praktiliselt kõik turul kasutusel olevad Androidi seadmed omavad OpenGL ES 2.0 tuge ja omavad seega ka teoreetiliselt WebGL'i ja tema laienduste tuge. Kõik OpenGL ES toega tooted on välja toodud Khronos Group „Conformant products" lehelt (Khronos Group, 2014).

Florian Boesch (Boesch, 2014) on oma ajaveebi sissekandes „Some issues with Apples iOS WebGL implementation" välja toonud vead (ingl k *bugs*), mis hetkel iOS'i ja Androidi seadmetes leiduvad. Autor on ise kasutanud erinevaid WebGL'i laiendusi ja võib kinnitada, et kuigi seadmed peaksid olema võimelised üles märgitud laiendusi kasutama, ei ole nad saadaval.

Autor katsetas WebGL'i mitme erineva operatsioonisüsteemi ja riistvara peal, kasutades veebilehitsejana Google Chrome'i või vabavaralist Chromiumi. Katsetustes kasutas autor endaloodud rakendusi:

- Rakendus 1 - koosneb paarist objektist ja liikuvast valgusallikast. Ei kasuta laiendusi ja on seega kindlalt ühilduv OpenGL 2.0 ja OpenGL ES 2.0 toega GPU'ga.
- Rakendus 2 – koosneb mitmekümnes objektist ja liikuvast valgusallikast. Lisaks kasutab järgnevaid laiendusi: *WEBGL_depth_texture (OES_depth_texture)* ja *WEBGL_draw_buffers (ARB_draw_buffers).* Spetsifikatsiooni kohaselt on laiendused toetatud OpenGL ES 2.0 API'is.

Uuringust saadud tulemuste põhjal (vt Tabelist 1) selgus, et lauaarvuti GPU, mis toetas vähemalt OpenGL 3.3 versiooni oli võimeline mõlemat rakendust jooksutama. Mobiilsed GPU'd OpenGL ES 2.0 või OpenGL ES 3.0 toega ei olnud võimelised Rakendust 2 jooksutama. Kasutatud laiendused *OES_depth_texture* ja *ARB_draw_buffers* võib leida OpenGL 1.4 ja OpenGL ES 2.0 spetsifikatsioonidest ja seega ei saa väita, et GPU'd, mis kasutavad laiendusi toetavaid API versioone, oleksid võimelised ka laiendusi kasutama.

Tabel 1 Süsteemide uurimisel saadud tulemused

GPU	OS	Rakendus 1	Rakendus 2
Geforce 6200 Turbocache (OpenGL 2.1)	Windows XP	Probleeme ei esinenud. Rakendus jooksis aeglaselt.	Ei käivitunud.
Geforce GTX 770 (OpenGL 4.5)	Windows 7	Probleeme ei esinenud.	Probleeme ei esinenud.
Geforce 8600M GT (OpenGL 3.3)	Ubuntu 14.04.1	Probleeme ei esinenud.	Probleeme ei esinenud.
PowerVR SGX531 (OpenGL ES 2.0)	Android 4.4	Probleeme ei esinenud. Rakendus jooksis aeglaselt.	Ei käivitunud.
Qualcomm Adreno 330 (OpenGL ES 3.0)	Android 5.0	Probleeme ei esinenud.	Ei käivitunud.

See alapeatükk näitas, et WebGL tugi erinevates veebilehitsejates on marginaalselt kasvanud, kus isegi Microsofti uusim Internet Explorer toetab WebGL API'it, mis veel aasta tagasi näis autorile võimatuna. Tutvusime ka probleemiga, mis esineb madalatasemelise API'iga, nimelt riistvara kokkusobimatus erinevatel põhjustel. Autor loodab, et alapeatükk näitas, et riistvaralist toetust ei ole nii kerge määrata, kui lihtsalt vaadata GPU'is toetatud API versiooni, vaid testida tuleb spetsiifilist graafikaprotsessorit.

1.3. Erinevused OpenGL'i ja WebGL'i vahel

WebGL baseerub OpenGL ES 2.0 spetsifikatsioonil ja säilitab OpenGL ES 2.0 semantika, et lähtekood oleks porditav ka mobiilsetele seadmetele (Khronos Group, 2014). Sellest faktist tulenevalt võime uurida ka OpenGL ES 2.0 ja OpenGL erinevusi.

Khronos Group (Khronos Group, 2014) on välja toonud mitmeid erinevusi:

- Tekstuurid peavad olema 2. astmes – Tekstuurid, mille laius või kõrgus ei ole 2. astmes (2, 4 ... 256, 512) saavad värviks musta. Õnneks on võimalik lihtsalt kirjutada algoritm, mis valiks tekstuuri dimensioonideks lähima 2. astme.
- Puudub topelttäpsuse (*double precision*) toetus – Suurim kasutatav ujukoma arv on 32 biti suurune.

- Puudub 3D tekstuuri toetus – Võimalik on emuleerida 3D tekstuuri 2D tekstuuriga, kirjutades selleks vajaliku algoritm.
- Erinevalt OpenGL'st, kus *vertex attribute 0* omab spetsiifilist semantikat, kus on vaja see aktiveerida massiivina, siis WebGL's käituvad kõik tippude atribuudid ühte moodi.
- Varjundajates olevate funktsioonidega, mis lõppevad sõnaga „Lod" (nt *texture2DLod*) on kasutatavad vaid lagivarjundajas.
- Varjundajates on võimalik määrata ujukoma arvude täpsused piiritlustega *precision highp float* või *precision mediump float.* OpenGL's kasutatakse tavaliselt alati kõige täpsemat ujukoma arvu. Kuna WebGL soovib olla porditav ja on kasutatav ka mobiilsetel seadmetel tuleb varjundajates kontrollida *#ifdef GL_FRAGMENT_PRECISION_HIGH* abil, kas seadmes on võimalik seda täpsust kasutada või ei.

Kuna WebGL rakendus jookseb otse veebilehitsejas ja jooksutab koodi, mis asub teises arvutis, on suurt tähelepanu pööratud turvalisusele. Erinevalt OpenGL'st, kus on võimalik ligi saada graafikaprotsessori mäluosale, mis ei pruugi kuuluda käesolevale programmile, siis WebGL's ei ole see võimalik. Sellest põhjusest tulenevalt ei ole võimalik varjundajates dünaamiliselt itereerida üle massiivi nt kasutades *for-tsüklit.* Iga massiivi indeks on vaja staatiliselt defineerida, et varjundajat kompileerides oleks võimalik indeksit kontrollida.

1.4. OpenGL'i ja WebGL'i lühiajalugu

Aastal 1981 rajati firma nimega Silicon Graphics (SGI), mis spetsialiseerus 3D arvutigraafikale ning arendas valdkonna tarbeks riistvara ja tarkvara. SGI arendas teegi IRIS GL (*Integrated Raster Imaging System Graphical Library*), mille abil sai luua 2D ja 3D graafikat tööjaamade jaoks. IRIS GL oli 90. alguses *de facto* 3D graafika teek. Probleemiks sai aga selle suletus, mille tulemusena oli SGI kaotamas oma positsiooni turul. Säilitamaks oma positsiooni eemaldas SGI IRIS GL teegist kõik funktsionaalsuse, mis ei olnud seotud graafikaga ja avaldas selle 1992. aastal OpenGL nime all. (Luten, 2014)

SGI ei avaldanud lähtekoodi, vaid ainult spetsifikatsiooni, kuidas API peaks töötama. Tekkis abstraktsioon riistvara ja tarkvara vahel, ning arendajad said ise valida, kuidas nad OpenGL'i implementeerivad. See abstraktsioon on tõene ka tänapäeval. Abstraktsioonist tulenevalt oli võimalik OpenGL'i ka laiendada, kui riistvara või tarkvara arendaja leidis, et mingi funktsionaalsus spetsifikatsioonist puudus. (Luten, 2014)

1990. aastate lõpuks oli OpenGL'st saanud standard nii CAD tarkvaras kui ka arvutimängude valdkonnas. OpenGL'i kasutasid Id Software (Quake 2), Epic Games (Unreal) ja Valve (Half-Life) – tõenäoliselt ühed kõige mõjuvõimsamad firmad arvutimängude valdkonnas sellel ajahetkel ja ka tänapäeval.

Aastad 2000-2009 enne OpenGL 3.1 versiooni oli OpenGL suurtes raskustes ja oli vajunud stagnatsiooni. Kuigi OpenGL oli standard professionaalse tarkvara loomisel (CAD-tarkvara, rastergraafika redaktor), suundusid ka sellise tarkvara loojad Windows platvormil Direct3D'd kasutama. OpenGL'i iseloomustab sellel ajastul kõike paremini killustatus, kus arendaja ei saanud erinevalt Direct3D'st kunagi kindel olla, et graafikaprotsessor API'it ja tema funktsionaalsusi toetab. (Luten, 2014)

Enne WebGL'i ajaarvamist oli võimalik 3D graafikat kiirendada Java Appleti või Adoble Flash Stage3D abil. Kumbki pistikprogramm ei olnud aga standardiseeritud ja ei sobinud Web 2.0'i. Aastal 2006 alustas Mozilla töötaja Vladimir Vukicevic eksperimenteerimist Canvas 3D'ga, kasutades OpenGL ES API'it. Eksperimendist valminud API sai pakutud Khronos Groupile standardiseerimiseks. Järgnevalt lõid Mozilla koos Khronos Groupiga WebGL'i töörühma ja 2011. aasta märtsis ilmus esimene ametlik versioon 1.0. (Dorland, Besten, 2014)

Hetkel on OpenGL ES *de facto* 2D ja 3D graafika API mobiilsetel platvormidel. WebGL'i toetavad kõik populaarsemad veebilehitsejad, kaasa arvatud Microsofti – kes on olnud OpenGL'i konkurent Windows operatsioonisüsteemi aegade algusest – Internet Explorer. OpenGL API on näidanud, et ka väga rasketel aegadel jääb API siiski pinnale püsima. OpenGL API ja tema erinevad versioonid, koos WebGL'ga, ei kao tõenäoliselt kasutuselt veel niipea.

2. Alternatiivid WebGL'le

WebGL'le otsest konkurenti nagu OpenGL puhul Direct3D ei ole. Küll aga on alternatiivseid mootoreid või raamistikke. WebGL'i asemel on võimalik muidugi kasutada ka raamistikke Three.js või Babylon.js, mis abstraheerivad terve API osa ja pakuvad peale selle veel muud funktsionaalsust nagu objektide laadimine, kollisiooni tuvastamine jne. Järgnevalt toome välja alternatiivid, mis ei kasuta WebGL'i.

Autor omab käesolevas alapeatükkides välja toodud alternatiividega ka praktilist kogemust. Mõnega suuremal määral kui teisega.

2.1. Unity 3D

Unity on mängumootor nagu seda on CryEngine, Unreal Engine või Source Engine. Veebilehitsejas on võimalik Unity'ga tehtud rakendust kasutada läbi Unity Web Player (lühidalt UWP) rakenduse. UWP on veebilehitsejas plugin, mis töötab Windows ja Mac OS X operatsioonisüsteemides kõikides suuremates veebilehitsejates. Unity mängumootoriga loodud rakendust on võimalik iseenesest jooksutada kõikidel platvormidel, kus on võimalik mänge mängida.

Unity eelis WebGL'i ees on see, et tegu on siiski mängumootoriga. Kui peamine eesmärk on arvutimängude loomine erinevatele platvormidele, ei ole kahtluski, et valida tuleks Unity.

Kui aga eesmärgiks on luua rakendus, mis jookseks veebilehitsejas ilma pluginata - tuleb tunnistada, et Unity Web Player rakendust ei leia enamuste kasutajate arvutitest - langeb kaalukauss WebGL kasuks. Kuna WeGL'i näol on tegu API'iga on ta võrreldes mängumootoriga paindlikum ja võimalik on luua ükskõik millist 2D või 3D graafilist rakendust või visualisatsiooni. UWP näol on tegu pluginaga, mis ei võimalda seega suhelda teiste DOM elementidega.

2.2. Flash 3D

Flash on tõenäoliselt kõige levinum plugin veebilehitsejas. Flash on platvorm, millega on võimalik luua mänge, animatsioone, veebirakendusi jms. Flash leiab kasutamist ka video või audio voogesitajana. 3D rakendusi on võimalik luua Flash Playeris ja Adobe AIR'is oleva Stage3D API abil. Mobiilsetes seadmetes ei ole Flash toetatud.

Flashi soovitab autor kasutada juhul, kui arendaja on aastaid tegelenud Flash arendamisega ja tunneb ennast selles süsteemis koduselt ja ei ole ressursse, et WebGL'i või mõnda tema raamistikku tundma õppida. Kõikidel muudel juhtudel on WebGL autori arvates parem, milles kohe ka selgusele jõuame.

Stage3D API ei ole standardiseeritud API nagu seda on Direct3D või OpenGL. Flashis kasutatud varjundajate formaat, mis meenutab Assembly keelt on võrreldes HLSL'iga või GLSL'iga autori arvates tunduvalt halvem. Suur läbimurre varjundajates oligi just Direct3D 9.0 versiooniga toimunud muutus, kus varjundajates võeti kasutusele C sarnane keel, mille implementeeris hiljem ka OpenGL GLSL näol.

Erinevalt WebGL'st, kus varjundajad kompileeritakse veebilehitsejas, peab Flash varjundajad kompileerima enneaegselt, tehes seega reaalajas varjundajatega eksperimenteerimise võimatuks.

Stage3D on ka tunduvalt aeglasem, kui uurida Felix Turner'i blogis välja toodud demosid ja lugeda tagasisidet (Turner, 2012). Autor jooksutas demosid nii Mozilla Firefoxis, kui ka Google Chrome'is ja Stage3D renderdamiskiirus oli marginaalselt väiksem.

Flash rakenduse jooksutamiseks on vajalik plugina olemasolu ja ei ole võimalik suhelda DOM elementidega. Standardiseeritud HTML5 on sihikindlalt hakanud Flash rakendusi asendama ja maailm liigub üha enam ühisloome suunas. Flash ei ole toetatud mobiilsetes seadmetes.

2.3. Silverlight 3D

Silverlight on Microsofti raamistik, millega on võimalik luua veebirakendusi. 3D rakendusi saab luua SilverLight 3D abil. Nagu ülejäänute alternatiivide puhul vajab ka Silverlight pluginat ja ei jookse ka mobiilsetel seadmetel.

Florian Boesch (Boesch, 2014) kirjutab, et 3D rakenduse jooksutamiseks on vaja kasutajal läbida mitmed segased dialoogid, et 3D rakendus üldse käivitada ja käivitamine pidavat olema praktiliselt võimatu.

Alates versioonist 5 kasutab Silverlight XNA teeki 3D graafika loomiseks (Lal, 2011). Microsoft on tänaseks aga XNA toetamise lõpetanud ja XNA elab tänapäeval edasi veel MonoGame mitmeplatvormilises raamistikus (järgnevas versioon kaob ka XNA). Lisaks sellele on Internet Explorer 11.0 versioonis WebGL toetatud, mis on otsene Silverlight konkurent. Need faktid lubavad prognoosida, et Silverlight on tõenäoliselt tulevikuta tehnoloogia, mille kasutamine on küllaltki riskantne.

2.4. SVG

SVG (*Scalable Vector Graphics*) on XML märgenduskeel kahedimensioonilise vektorgraafika kirjeldamiseks. SVG on W3C standard ja on toetatud kõikides kaasaegsetes veebilehitsejates (W3C, 2014).

Kuna SVG element kuulub DOM elementide hulka, on võimalik elementidele lisada kuulareid ja CSS abil määrata elementidele stiil. Veebilehitseja oskab SVG elemente ise renderdada ja kasutab nende määramiseks XML märgenduskeelt. SVG sobib igaljuhul igasuguse 2D graafika loomiseks. Tema eelisteks WebGL'i ees on kindlasti lihtsus ja ühilduvus riistvaraga.

SVG probleemiks on samas just see XML märgenduskeel, juhul kui elementide arv muutub liiga suureks ja neid on vaja liigutada reaalajas. WebGL'i presenteeritakse küll põhiliselt 3D API'ina, kuid kompleksse 2D visualisatsiooni (fraktaalid, dünaamilised graafid jne.) renderdamine WebGL'i abil võimaldab meil osa CPU peal tehtavast tööst viia GPU peale, mis märgatavalt tõstab rakenduse jõudlust.

3. Olemasolevad õppematerjalid

Käesolevas peatükis analüüsime erinevaid õppematerjale, mille abil on võimalik WebGL'i õppida. Üritame katta laia spektrumit olemasolevatest materjalidest, analüüsides WebGL'i, OpenGL'i, OpenGL ES'i materjale erinevatel meediumitel, mis on lihtsalt ja laialdaselt kättesaadavad. Autor leiab, et ei piisa WebGL'i materjalide analüüsist, sest materjali leidub küllaltki vähe ja tihtilugu on autori arvates tegu suhteliselt kasina materjaliga, milles sisalduv informatsioon võib olla lünklik. Analüüsist saadud teadmised võtame kasutusele õppematerjali koostamisel.

Välja toodud materjalidega omab autor otsest kokkupuudet ja on kasutanud neid WebGL'i ja graafika programmeerimise õppimiseks. Autor üritab olla võimalikult objektiivne, tuues välja materjalide positiivsed ja negatiivsed küljed.

3.1. Learning WebGL – 3D Programming for the Web

Tüüp	E-õppematerjal, ajaveeb
Sihtrühm	Algajad
Aadress	http://learningwebgl.com
Autor(id)	Giles Thomas, Tony Parisi

Õppematerjal on kirjutatud ajaveebi stiilis. Materjali koostamist alustas Giles Thomas 2009. aastal, kui ta WebGL'i õpinguid alustas, ning praeguseks hetkeks on materjali täiustamise üle võtnud raamatu „*WebGL: Up and Running*" autor Tony Parisi. Materjali esimesed kümme õppetundi baseeruvad NeHe OpenGL õppematerjalil (GameDev.net, 2012).

Ajaveebis käsitletakse kõiki peamisi teemasid, mida algaja peaks tundma õppima, enne kui saaks edasi liikuda edasijõudnutele mõeldud materjalile. Kuna materjal on kirjutatud õppijalt õppijale on tekst kergesti loetav ja arusaadav. Materjal on eelkõige lähtepunktiks, mis annaks esimese praktilise kogemuse WebGL'ga.

Autor leiab, et materjal ei käsitle või jätab segaseks mõned 3D graafika ja WebGL API kontseptsioonid ning ei paku ka allikaid, kust oleks võimalik nende teemade kohta lisaks õppida, mis kokkuvõttes võib tekitada õppurile rohkelt küsimusi ja lünkliku informatsiooni.

3.2. Webgl Academy: Tutorial to learn WebGL

Tüüp	E-õppematerjal, demo
Sihtrühm	Algajad, edasijõudnud
Aadress	http://www.webglacademy.com/
Autor(id)	Xavier Bourry

Autor leiab, et tegu ei ole päris õppematerjaliga, vaid pigem näitekoodiga, kust saab uurida, kuidas mingi asi on implementeeritud. Teemad ei ole algaja jaoks piisavalt lahti seletatud, et ta tegelikult mõistaks, kuidas renderdamise järjekord toimib.

Materjal sobib lisainformatsiooniks, kuid selle materjali põhjal kontseptsioone ja täidlast arusaamist omandada on algajale väga raske protsess.

3.3. Tutorials for modern OpenGL(3.3+)

Tüüp	E-õppematerjal, ajaveeb
Sihtrühm	Algajad, edasijõudnud
Aadress	http://www.opengl-tutorial.org/
Autor(id)	Sam Hocevar

See õppematerjal käsitleb kaasaegset OpenGL'i alates versioonist 3.3, kust on eemaldatud fikseeritud funktsionaalsus. Materjalist leiab nii algajatele, kui ka edasijõudnutele mõeldud teemasid nagu nt osakeste efektid, varjud, teksti renderdamine jne.

Materjal on kesktee liiga detailse ja liiga pealiskaudsuse vahel. Praktiliste näidete vahel leiab kontseptsioonide seletusi ja ohtralt pildimaterjali, mis õppimist lihtsustavad. Välja on toodud ka muud allikad, mille abil lisainformatsiooni saada.

Autor leiab, et materjaliga kaasa tulev näitekood ei ole kõige parem, sellepärast et terve loogika, ka see, mis ei ole seotud otseselt peatükiga on ühes kohas. See lahendus muudab lähtekoodi loetavust tunduvalt halvemaks. Samuti on kontseptsioonide seletused peatükkide vahele peidetud, seega kui õppija tahab midagi korrata, peab ta peatükkide vahel õige koha ise üles leidma.

Materjal on OpenGL'i kohta ja ei sobi algajatele WebGL õppijatele, vaid neile, kes on WebGL'ga juba midagi loonud ja suudavad lähtekoodi ümber transleerida ning on teadlikud WebGL'i ja OpenGL'i erinevustest.

3.4. Learning Modern 3D Graphics Programming

Tüüp	E-õppematerjal, raamat
Sihtrühm	Algajad, edasijõudnud
Aadress	http://arcsynthesis.org/gltut/
Autor(id)	Jason L. McKesson

Õppematerjal käsitleb kaasaegset graafika programmeerimist. Materjal on saadaval paberkandjal ja ka elektrooniliselt ning on mõeldud nii algajatele, kui ka edasijõudnutele.

Raamat on väga põhjalik ja suur rõhk on pandud matemaatikale ja kontseptsioonide omandamisele, et õppija oskaks tulevikus erinevaid graafikaga seotud probleeme lahendada. Raamatus kasutatakse OpenGL'i, kuid ennekõike on tegu puhtalt graafika programmeerimise õpetamisega, mis õpetab selle kõrvalt ka OpenGL'i.

Autor soovitab raamatut kasutada siis, kui WebGL'ga on esmane kogemus olemas ja renderdamise järjekorrast on selge ülevaade, et asi süvitsi ette võtta ja sellest materjalist saadud arusaamad WebGL rakenduses kasutusele võtta.

3.5. Learn OpenGL ES

Tüüp	E-õppematerjal
Sihtrühm	Algajad
Aadress	http://www.learnopengles.com/
Autor(id)	Kevin Brothaler

Materjal annab sissejuhatuse OpenGL ES 2.0 kasutamiseks Androidi seadmetes. Autor kirjutab materjali paralleelselt API õppimisega, ning seetõttu on materjal mõeldud algajatele.

E-õppematerjal annab hea sissejuhatuse OpenGL ES kasutamiseks Androidi platvormil. OpenGL ES, olles sarnase süntaksiga, annab võimaluse näitekoodi ilma suurema vaevata ümber transleerida WebGL'i omaks. Algajal võib siiski probleeme tekitada erinevate massiivide, maatriksite ja

puhverobjektide loomisega, sest Javascript on erinevalt Javast dünaamiline keel, millest puuduvad muutujate tüübid.

Kuna materjal on sissejuhatuseks, mis käsitleb põhiliselt kõige algelisemat programmi, ilma objektide laadimiseta, liigutamiseta, ei ole tegu täieliku materjaliga ja jätab autori arvates suure osa informatsioonist välja, mis tuleks algajal omandada.

3.6. The Official Khronos WebGL Repository

Tüüp	Lähtekoodi hoidla
Sihtrühm	WebGL arendajad, WebGL kasutajad
Aadress	https://github.com/KhronosGroup/WebGL
Autor(id)	Khronos Group

Tegu on ametliku Khronos Groupi WebGL hoidlaga. Hoidlast leiab WebGL'i spetsifikatsioonid ja ühilduvustestid. Lisaks on hoidlas saadaval ka erinevad demod, mis võivad arendajatele huvi pakkuda. Kindlasti ei ole hoidla mõeldud kasutamiseks algajatele, vaid juba WebGL API'it kasutavatele arendajatele, kellel on vaja detailset ülevaadet API'ist.

3.7. Web technology for developers – WebGL

Tüüp	E-õppematerjal, infoleht
Sihtrühm	Algajad
Aadress	https://developer.mozilla.org/en-US/docs/Web/WebGL
Autor(id)	Mozilla Developer Network

See lehekülg sisaldab endas erinevaid elementaarseid WebGL'i näiteid ja häid tavasid. Lehelt leiab ka informatsiooni WebGL'i laienduste ühilduvuse kohta erinevates veebilehitsejates (laienduste ühilduvuse jaoks on soovitatav siiski kasutada http://webglstats.com/ lehte). Materjalist leiab ka erinevaid viiteid teistele WebGL'i materjalidele ja teekidele, mis API kasutamist lihtsustavad.

3.8. OGLdev – Modern OpenGL Tutorials

Tüüp	E-õppematerjal
Sihtrühm	Algajad ja edasijõudnud
Aadress	http://ogldev.atspace.co.uk/
Autor(id)	Etay Meiri

See õppematerjal käsitleb kaasaegset OpenGL'i alates versioonist 3.3, kust on eemaldatud fikseeritud funktsionaalsus. Materjalist leiab nii algajatele, kui ka edasijõudnutele mõeldud teemasid. Autori arvates on tegu materjaliga, mis kõikidest teistest e-õppematerjalidest käsitleb paljusid vajalikke teemasid koos praktiliste näidetega.

Materjali iga peatükiga käib kaasas lähtekood. Peatükid on hästi struktureeritud, kus alguses tuuakse välja taustinfo ning pärast seda käiakse läbi lähtekood, mis seletatakse põhjalikult tükkideks. Autor leiab, et osad, mis käsitlevad matemaatikat, võiksid siiski olla eraldi teema all.

Materjalis kasutatakse C++'i ja OpenGL'i, mistõttu on seda algajale WebGL'i õppijal suhteliselt keeruline kui mitte võimatu kasutada. Materjalis kasutatakse ka ohtralt erinevaid objekte ning autori endaloodud klasse, mis on vältimatu, kuid teeb näitekoodis orienteerumise keeruliseks.

4. SAM-mudel

Õppematerjali koostamiseks kasutame SAM-mudelit (ingl k *Successive Approximation Model*). Autor kasutas seminaritöö „Libgdx raamistik ja 2D arvutigraafika õppematerjal“ käigus ADDIE-mudelit, mille suureks puuduseks autori arvates oli selle lineaarne protsess. Pideva tagasiside puudulikkus tulevastelt õppematerjali kasutajatelt, ei võimaldanud arendamise käigus korralikult hinnata loodava õppematerjali kvaliteeti ja kasutajasõbralikkust, mille SAM-mudel loodetavasti lahendab.

SAM-mudel pakub lihtsat, kuid kindlat teed eesmärgi saavutamiseks alternatiivina ADDIE-mudelile. Mudel on kindlalt defineeritud, kuid ei kaota ära, ja pigem innustab loomingulisust ning eksperimenteerimist. Mudel täidab neli kriteeriumit: iteratiivne, koostööl põhinev, tõhus ja efektiivne ning juhitav. (Allen, 2012)

SAM-mudelil on kaks taset: SAM1 ja SAM2. SAM1 on mõeldud kasutamiseks väikesele meeskonnale või üksikisikule, kellel ei ole vaja kõrvalist abi. SAM2 on SAM1 edasiarendus, mis on mõeldud kasutamiseks suuremate projektide tarbeks. (Allen, 2012)

4.1. SAM1

SAM1 sobib väikeste projektide jaoks, kus töötatakse üksi või väikeses grupis ning vaja ei lähe väljastpoolt tulevat abi. Mudelis kasutatakse iteratiivset tööprotsessi, mis kordub vähemalt kolm korda ja mis jaguneb kolmeks osaks: analüüs, kavandamine ja arendamine. SAM1’i abil valmib kasutusvõimeline toode paari iteratsiooniga, sellepärast et toote loomist alustatakse juba protsessi alguses. (Allen, 2012)

Esimese iteratsiooni analüüsi osas toimub hetke situatsiooni, vajaduste ja eesmärkide hindamine. Kavandamise osas toimub ilmselgete eesmärkide seadmine, meediumi valimine jms. Arenduse osas valmistatakse esimene algeline prototüüp, mis annaks esimese kujutluse loodavast materjalist.

Teise iteratsiooni analüüsi osas hinnatakse esimeses iteratsioonis tehtut. Vajadusel kogutakse lisainformatsiooni ja hinnatakse eesmärgid ümber. Kavandamise osas luuakse uusi alternatiive eelmistele kavanditele või täiustatakse eelnevat ideed, olenevalt selles, mis selgus analüüsi käigus. Arenduse osas valmivad prototüübid, mis kujutavad põhjalikumalt lõplikku versiooni.

Kolmas iteratsioon on sarnane teisele iteratsioonile, kuid keskendub rohkem arendamisele. Selles iteratsioonis keskendutakse probleemide uuesti läbivaatamisele ja vigade likvideerimisele.

4.2. SAM2

Mudel on mõeldud suuremahulistele projektidele, kus toote arendamist ei saa integreerida kavandamisega ja projektid valmivad paljude inimeste koostööna. Mudel on jagatud kolmeks etapiks: ettevalmistus, iteratiivne kavandamine ja iteratiivne arendamine. (Allen, 2012)

Ettevalmistuse etapis toimub taustinfo kogumine, mille käigus tehakse selgeks eesseisvad probleemid. Samuti toimub etapis eesmärgi seadistamine ja mõningate valikute likvideerimine. Etapis toimub ka *Savvy Start,* mille käigus vaatab meeskond üle kogutud informatsiooni ja loob esimesed kavandid (Allen, 2012).

Iteratiivses kavandamise etapis kasutatakse SAM1'is käsitletud iteratiivset tööprotsessi, mille käigus valmib toote kavand. Etapist toimub lisaks projekti planeerimine, kus analüüsitakse projekti arendusdetaile, aega ja kulu ning toimub ka täiendav kavandamine, mille käigus korraldatakse *Savy Start* istungeid.

Iteratiivses arendamise etapis rakendatakse samuti SAM1'is käsitletud tööprotsessi. Arenduse etapi alguses koostatakse *design proof*, millega alustatakse toote arendamist. Toode luuakse kolmes versioonis: alfa, beeta ja kuld. Kuldne versioon võetakse kasutusele.

5. Õppematerjali koostamine

Õppematerjali koostamisel lähtume SAM1-mudelis välja toodud tööprotsessist. Tulenevalt sellest, et autor ei tunne seda valdkonda spetsialisti tasemel, võtame kasutusele SAM2-mudelis välja toodud ettevalmistuse etapi, mille käigus kogume vajalikku taustinfot, et õppematerjali koostada.

Käesolev peatükk koosneb kolmest osast: ettevalmistus, iteratiivne hindamine ja kavandamine ning arendamine. Ettevalmistuse etapis selgitame välja vajaduse ja kogume taustinfot. Hindamise ja kavandamise osas käsitletakse loodava õppematerjali eesmärki, õpiväljundeid, struktuuri ja meediumit. Arendamise etapis toimub õppematerjali koostamine. Selles osas toome välja erinevates iteratsioonides tehtud muudatused, testimisest saadud tulemused ja järeldused.

5.1. Ettevalmistus

Selles alapeatükis sõnastame, millest tuleneb WebGL'le õppematerjali koostamise vajadus ja kogume informatsiooni materjali loomiseks. Etapi läbides peaks autoril olema piisavalt teadmisi, et õppematerjali kavandamist ja arendamist alustada.

5.1.1. WebGL'i õppematerjali vajadus

Hetkel puudub eestikeelne materjal, mille abil oleks loengutes võimalik WebGL'i õpetada või seda iseseisvalt õppida. Autorile teadaolevalt puudub eesti keeles ka materjal, mis õpetaks graafika programmeerimist läbi OpenGL ja OpenGL ES API'ide. Tallinna Ülikooli aines „Graafika ja muusika programmeerimine" (ainekood IFI6028) sai õpitud Three.js teeki, mis on autori arvates sissejuhatuseks väga hea, kuid tõsisemale WebGL'st ja graafika programmeerimisest huvitujale jääb sellest materjalist väheks.

Autor leiab, et teekide õppimine võib tekitada illusiooni teadmistest, mida ei pruugi isikul olla. Madalatasemelise API õppimine annab õppijale laialdasemad teadmised, võimaldades mõelda suuremalt ning näha probleeme, mida teekide kasutajad ei pruugi näha, sest otseselt suheldakse graafika riistvaraga. Graafikarakendusi arendades on vaja rakendus tihti muuta omanäoliseks. Olenevalt rakendusest, võib juhtuda, et vastavalt rakenduse iseloomust on vaja rakendus tugevalt optimiseerida. Omamata teadmiseid GLSL keelega kirjutatud tipu- ja pikslivarjundajatest ning arusaama renderdamise järjekorrast ja matemaatikast, luuakse lihtsalt üldine rakendus, mis ei ole optimiseeritud ega omanäoline. Omades teadmisi API kohta, on õppijad võimelised nägema

probleemide lahendamiseks uusi võimalusi ja ei ole piiratud teekide poolt pakutud funktsionaalsusega.

5.1.2. Informatsiooni kogumine

Informatsiooni kogumiseks õppis autor teemat mitmete internetis olevate materjalide põhjal. Materjalide analüüsi käsitles autor kolmandas peatükis "Olemasolevad õppematerjalid".

Teadmiste täiustamiseks luges autor järgmiseid raamatuid: „OpenGL SuperBible Sixth Edition" (Haemel, Sellers, Wright, 2013), „3D Math Primer for Graphics and Game Development (2nd Edition)" (Dunn, Parberry, 2011) ja „Game Engine Architecture" (Gregory, 2009). Raamatud andsid autorile detailsema ülevaate kaasaegsest renderdamise järjekorrast ja ka üldisemaid teadmisi renderdamismootorist ja sellega seonduvast matemaatikast.

Praktiliselt sai saadud teadmisi kasutatud WebGL rakenduste loomisel, kus selgusid WebGL eripärad ja erinevused. Kogutud informatsioonist selgusid järgmised teemad, mida autori arvates peab materjalis kindlasti käsitlema:

- Vektorid ja maatriksid
- Erinevad koordinaatsüsteemid
- Valgustusmudel
- Koordinaatruumides ja –süsteemides liikumine
- Varjundajad
- Renderdamise järjekord
- Tekstuurid
- Renderdamiseks kasutatavate andmete kuju
- Objektide liikumine
- Rakenduse interaktiivseks muutmine

Informatsiooni kogumise käigus selgus, et matemaatika ja füüsika on 3D API'ide (sealhulgas WebGL'i) väga tähtis osa, mistõttu on vaja tutvustada kasutatavat matemaatikat ja füüsikat. Autor jõudis järeldusele, et väga tähtis on õppijal aru saada, kuidas koordinaatsüsteemides ja -ruumides liikuda, kuidas toimib renderdamise järjekord ja mis kujul salvestada andmeid. Autor arvab, et põhjalikumalt vajab käsitlemist ka tekstuur, mida algajatele mõeldud materjalid peavad vähem tähtsaks.

Kindlasti on tähtis tippude välja heitmine (*ingl k vertex culling*), objektide laadimine, teksti renderdamine jms, kuid kindlasti ei ole need esmatähtsad. Neid teemasid käsitleme vaid siis, kui selleks aega jääb.

Autor leidis, et raamatu kujul olev materjal oli paremini struktureeritud, kuid raamatuga ei olnud kaasas näitekoodi või oli kood raskesti käivitatav. E-õppematerjalid sisaldasid ohtralt praktilisi näiteid, koos lähtekoodiga, kuid puudu jäi mõningal määral teooriast. Nendest asjaoludest lähtuvalt oli vaja pidevalt uurida paralleelselt kahte-kolme materjali, et vajalikud teadmised kätte saada. Seda vaatlust saab kasutada õppematerjali koostamisel, võttes kasutusele loetud raamatute ja e-õppematerjali positiivsed küljed ning üritades vältida negatiivseid külgi.

5.2. Terminite eestikeelsete vastete leidmine

Autor on saanud osa termineid Blenderi kasutusjuhendist (Blender, 2012). Sõnad, millele autor eestikeelset vastet ei leidnud või ei pidanud eestikeelset terminit sobilikuks, toob autor välja alljärgnevalt. Iga termini kõrval on termini lühiseletus, mis kontekstis seda kasutada. Autor üritas terminid tõlkida võimalikult ligilähedaselt ingliskeelsetele terminitele, et õppuril oleks pärast lihtsam ingliskeelset materjali lugeda.

- kaadripuhver (ingl k *frame buffer*) – Igas kaadris väljuvad pikslid pikslivarjundajast puhvrisse.
- kaameramaatriks (ingl k *camera matrix*) – Ruumi erinevatest nurkadest vaatamiseks on meil vaja kaamerat, mille saame defineerida kaameramaatriksiga.
- kaameraruum (ingl k *camera space*) – Iga kaameramaatriks defineerib enda koordinaatide süsteemi ehk kaameraruumi, kuhu me saame teised objektid vajadusel viia.
- koordinaatruum (ingl k *coordinate space*) – Meil on võimalik maatriksite abil defineerida lõpmata hulk koordinaatsüsteeme, kuhu me saame meie poolt defineeritud objekte viia. Iga maatriks defineerib mingit koordinaatruumi.
- maailmaruum (ingl k *world space*) – Kõik objektid ja ka kaamerad paiknevad ühtses ruumis. Selleks ruumiks on maailmaruum.
- mudelmaatriks (ingl k *model matrix*) – Kõik objektid/mudelid paiknevad objektruumis. Objektruumist maailmaruumi viimiseks kasutame mudelmaatriksit.
- normaliseeritud seadme ruum (ingl k *normalized device coordinates*) – Kõik tipud, mis paiknevad selles ruumis on vahemikus [-1, 1]. Sellest ruumist on lihtne koordinaadid viia vaateakna ruumi. Seda ruumi tunnevad kõik seadmed (GPU'id).
- objektruum (ingl k *object space*) – Ruum, kus paiknevad mingit objekti defineerivad tipud.

- projektsioonmaatriks (ingl k *projection matrix või clip matrix*) – Maatriks, mida kasutatakse pügamisruumi defineerimiseks ja tippude ettevalmistamiseks projekteerimiseks. Muutes maatriksit muutub ka projektsioon.
- pügamisruum (ingl k *clip space*) – Pügamisruumi defineerib projektsioonmaatriks. Kõik tipud, mis on pügamisruumist väljas, pügatakse ehk visatakse välja ja ekraanile ei renderdata.
- vaateakna ruum (ingl k *screen space*) – Vaateakna ruum on aken, kuhu renderdame. Täisekraani puhul terve ekraan. Me projekteerime 3D ruumi 2D pinnale, mistõttu me vaataks justkui aknast välja.

5.3. Hindamine ja kavandamine

Selles osas toimub eesmärgi ja õpiväljundite sõnastamine, üldise struktuuri ja sisu kavandamine, õpiteede määramine ning meediumi valimine. Õppematerjali koostamiseks kasutame iteratiivset tööprotsessi, mistõttu võib plaan iga iteratsiooniga muutuda. Killustatuse vältimiseks on käesolevas osas kajastatud ainult lõplik kavand.

5.3.1. Eesmärgi sõnastamine ja õpiväljundid

Eesmärgiks on luua õppematerjal, mille abil oleks võimalik omandada baasteadmised WebGL'st, saada sissejuhatus graafika programmeerimisest ja õppida, kuidas luua WebGL'i abil reaalajas jooksev interaktiivne graafikarakendus. Materjali koostamisel lähtub autor kogemustest, mis on saadud WebGL'i, OpenGL'i õppimisega ja WebGL'i rakenduste loomisega. Eesmärgiks on koostada materjal viisil, mida oleks hea kasutada erineva tasemega ja õppimisviisiga õppijatel.

Õppematerjal on eeskätt suunatud WebGL'st ja graafika programmeerimisest huvitujatele, kes oskavad mõnda programmeerimiskeelt vähemalt kesktasemel ning omavad võimet lähtekoodi lugeda. Materjali omandamist lihtsustab kindlasti hea matemaatiline ja loogiline mõtlemine.

Õppematerjali läbinu:

- Omab ülevaadet WebGL'st ja renderdamise järjekorrast.
- Teab, mis eesmärgil kasutatakse vektoreid ja maatrikseid.
- Saab aru, kuidas luua tsükkel, et pilt liikuvaks teha.
- Mõistab, kuidas rakendus interaktiivseks muuta.
- Suudab iseseisvalt edasi õppida.
- Oskab saadud teadmisi rakendada, et WebGL'i abil lihtne interaktiivne rakendus luua.

5.3.2. Struktuur

Õppematerjali ülesehitus tugineb autori kogemustel, mis on saadud graafika programmerimist õppides ja rakendusi luues. Ehitusel toetub autor ka samalaadsetel õppematerjalidel.

Autor jagab õppematerjali kaheks osaks, millest esimene on põhiliselt teoreetiline ja teine põhiliselt praktiline. Autor leiab, et niiviisi on võimalik õppuril ühest kindlast kohast leida kontseptsioonide seletusi, mis muidu oleks paisatud segamini koodinäidetega, muutes materjali vähem efektiivsemaks.

Teoreetilises osas tutvustame WebGL'i, seletame lahti praktilises osas korduvad WebGL'i meetodid ja hädavajaliku matemaatika. Iga teoreetilise osa lõpust leiab õppur ülesandeid või kordamisküsimusi. Teoorias käsitletud teemad peaksid andma raamistiku, millele teadmisi ehitada. See osa on mõeldud õppurile ka pidevaks kordamiseks, kui praktilises osas peaks mõni ebaselgus esile kerkima. Graafika programmeerimist ei ole võimalik katta ühe ainsa materjaliga ning seetõttu on teoreetilises osas lisaks erinevad viited lisamaterjalidele, mis asja detailsemalt või teise nurga alt seletavad.

Praktilises osas loome praktilisi näited, mis komplementeeriks teoreetilist osa ja näitaks praktiliselt, kuidas graafika renderdamine ja erinevate probleemide lahendamine reaalselt käib. Praktilisest osast leiab ka seletusi, mis teoreetilisse ossa ei sobi. Iga praktilise osa lõpust leiab õppur ülesanded või kordamisküsimused, et teadmiste omandamises veenduda. Praktilise osa läbides peaks õppur olema võimeline WebGL'i iseseisvalt kasutama ja koos teoreetilise osaga omama baasteadmiseid graafika programmeerimisest.

5.3.3. Õpiteede määramine

Materjal on mõeldud kasutamiseks samas keskkonnas, kus WebGL – veebilehitsejas. Materjali kasutamiseks on vajalik internet ühendus, sest autor innustab õppureid uurima lisaallikaid, mis teemat teisest perspektiivist või detailsemalt seletavad ja kindlasti omandamist lihtsustavad.

Autor leiab, et võimalikke viise, kuidas õppija materjali kasutab on mitmeid:

- Õppur alustab õppimist teooria lugemisega, pärast mida suundub praktilisi ülesandeid lahendama
- Õppur uurib ja muudab näitekoodi ja loeb paralleelselt teooriat
- Õppur leiab, et teooria, mis puudutab matemaatikat on raske ja liigub praktilisi ülesandeid lahendama. Parema ülevaate saades uurib matemaatika osa
- Õppur kasutab näitekoodi, et luua enda rakendus. Võibolla uurib ka teooriat

Praktikaosa on kindlalt mõeldud lahendamiseks lineaarselt, kus iga järgneva õppetüki arusaamiseks on vaja läbida eelnev õppetükk. Kui vaja viidatakse praktika osas teooriale, mida on õppuril võimalik lugeda, et asjast parem arusaam saada.

Teoreetilist osa on soovitatav pidevalt korrata ja uurida ka seal sisalduvaid lisamaterjale. Teoreetilist osa on võimalik läbida nii lineaarselt kui ka vastavalt sellele, mis teemat praktikas käsitletakse. Parim moodus on lugeda teooriat, pärast mida liikuda praktika juurde ja olenevalt peatükist korrata jällegi teooria osa.

Teooria osa, mis sisaldab WebGL funktsioone on mõeldud kasutamiseks koos praktikaga, kui õppur tunneb, et mõni meetod on tema jaoks seletatud ebamääraselt või tahab meetodi kohta rohkem teada saada. „WebGL funktsioonid" osas leiduvat näidiskoodi saab kasutada ka viitematerjalina, kui õppur hakkab looma iseseisvalt rakendusi.

Peatükk „Alternatiivne materjal ja õppimise jätkamine" on mõeldud õppurile, kes leidis, et õppematerjal ei ole neile sobilik või sooviks õppimist jätkata.

5.3.4. Meedium

Materjali meediumiks valis autor GithHub Wiki. Tegu on väga kaasaegse meediumiga, mida kasutavad laialdaselt erinevad avatud lähtekoodiga projektid dokumentatsioonide ja juhendite kirjutamiseks. Wiki kirjutamisel saab kasutada mitmeid erinevaid keeli, mille abil on võimalik määrata pealkirju, näitekoodi blokke, pilte, tsitaate jne. Selle õppematerjali loomisel kasutame Markdown keelt, mis pärast transleeritakse HTML koodiks. GitHub Wiki eelised traditsioonilise meediumi ees on mitmeid:

- Hüperlinkide abil on võimalik lugeja kiiresti suunata õigesse kohta.
- Kõige uuem versioon on alati saadaval ühest kohast.
- Sarnaselt lähtekoodiga on võimalik näha muutmise ajalugu.
- Võimalik on kaasata õppijad materjali arendamisse.
- Materjaliga kaasaskäiv lähtekood kasutab sama süsteemi.
- Materjali võivad muuta kõik isikud, kellel on selleks õigus.

Meediumi valikust tulenevalt ei ole mõistlik tervet näitekoodi peatükki lisada, vaid õppuril on mõistlikum näitekood avada veebilehitseja teises aknas või arenduseks kasutatavas tekstiredaktoris/programmeerimiskeskkonnas. Küll aga tuuakse peatükkides välja näitekoodi osasid.

Korduvad mõisted või kasutatud WebGL meetodid, mis võivad õppurile segased olla, sisaldavad endas hüperlinke vastavate seletuste juurde. Näitekoodi osa eraldamiseks kasutame Markdown keele *Code* blokki, mis eraldab selle halli blokiga (vt joonist 2).

```
var shaderProgram = shaderProgramLoader.getProgram(<tipuvarjundaja asukoht>, <lagiva
```

Joonis 2 Näitekoodi blokk

Teadmised, mida on vaja kindlasti meeles pidada ja mida ei pruugi õppur teada, lisatakse märkustena või juhistena materjali sisse, kasutades Markdown *Blockquotes* elementi, mis seda osa rõhutab (vt joonist 3).

> Maailmaruumist kaameraruumi saame kaameramaatriksiga.

Joonis 3 Märkuse blokk

5.4. Arendamine

Selles osas toimub õppematerjali arendamine ja testimine. Iteratsioonides I ja II toimub iteratsiooni käigus loodud õppematerjali testimine ja järelduste tegemine. Olenevalt saadud järeldusest on võimalik õppematerjali ümber hinnata ning sisusse või struktuuri sisse viia muudatusi. Igat järgnevat iteratsiooni täiustatakse vastavalt sellele, mis järeldustele jõuti iteratsiooni lõpus ja mis muudatused viidi sisse kavandis. Iteratsioonis III toimub õppematerjali viimistlus, kus keskendutakse arendamisele ning üritatakse vältida ümberhindamist.

5.4.1. Iteratsioon I

Esimese iteratsioonis on ainult õppematerjali teoreetiline osa. Andes testijatele seda osa lugeda, on võimalik määrata, kui raskeks võivad osutuda erinevad praktilised näited ja ülesanded. Sõltuvalt tulemustest on võimalik praktika osa lihtsamaks või raskemaks muuta.

Prototüüp koosneb järgnevatest teoreetilistest peatükkidest:

- **Vektorid** – Ülevaade vektoritest oskajatele kordamiseks ja oskamatutele õppimiseks.
- **Maatriksid** – Seletus, kuidas geomeetriliselt maatriksit kujutatakse ja kasutatakse.
- **Polaarne ja sfääriline koordinaatsüsteem** – Ülevaade polaarsest ja sfäärilisest koordinaatsüsteemist ning kuidas süsteeme ristkoordinaatsüsteemi teisendada.

- **Valgus ja valgustusmudel** – Lihtsaima valgustusmudeli tutvustus, mida kasutatakse valguse simuleerimiseks.
- **Koordinaatruumid** – Kiire ülevaade tavalistest koordinaatruumidest, mida kasutatakse renderdamise järjekorras.
- **Renderdamise järjekord** – Seletus, mis sammudest koosneb ekraanile renderdamine.
- **Varjundajad** – Tipu- ja pikslivarjundaja tutvustus ja seletus.

Testimine

Versiooni testisid viis inimest, kellest kaks on programmeerijad ja üks programmeerimisest huvituja. Kõik testijad on keskkooli programmi kohaselt kokku puutunud vähemalt vektoritega ja polaarse koordinaatsüsteemiga. Mõned peaksid olema läbinud ka maatriksite osa. Optikaga peaksid testijad omama kokkupuudet põhikoolist ja keskkoolist. Testijatel lastakse materjal läbi lugeda ja jälgitakse nende reaktsioone ning muljeid.

Selgus, et kuigi kõik testijad peaksid omama kokkupuudet nende teemadega, olid matemaatikat ja füüsikat puudutavad osad mõne testijate jaoks keerulisemad, kui autor oleks osanud oodata. Kergemad või isegi ebavajalikud olid teemad testijale, kes on praktiliselt neid ka rakendanud. Positiivselt reageeris testija, kes oli koolis neid teemasid hiljuti käsitlenud.

Järeldus

Varieeruv tase annab alust arvata, et matemaatika ja WebGL'i teemad tuleks hoida üksteisest lahus, mida autor oli algusest peale kavandanud. Hoides teemasid lahus, on võimalik olenevalt teemast viidata vastavale matemaatika osale, mida teemaga kursis olev õpilane ei pea lugema. Selline lahendus parandab kindlasti materjali loetavust ja võimaldab õppuril paremini järge pidada.

5.4.2. Iteratsioon II

Teises iteratsioonis lisandub õppematerjali praktiline osa. Täiustatakse ka teoreetilist osa, mis komplementeeriks praktilist osa. Iteratsioonis arendatakse iga praktilise osa kohta lähtekood, mida oleks õpilasel võimalik käivitada ning koheselt ka muudatusi sisse viia. Lähtekood luuakse viisil, et seda oleks võimalikult lihtne muuta ja peamine osa, mis on käesolevas tunnis kõige tähtsam, oleks muust mürast eraldatud.

Materjali teoreetilisse osasse lisame peatüki „WebGL funktsioonid", kust leiab õppur kasutatud meetodite seletuse. Pannes funktsioonid eraldi peatükki on õppuril alati võimalus kindlast kohast

leida informatsiooni küsimusi tekitava meetodi kohta. Selline lahendus võimaldab meil praktikas püsida kergemalt ettenähtud kursil.

Õppematerjali lisati järgmised teoreetilised osad:

- **Tekstuurid ja tekstuuride kasutamine** – Tutvustus, kuidas ja milleks kasutatakse WebGL's tekstuure.
- **WebGL funktsioonid** – Praktilises osas kasutatud WebGL funktsioonide sügavam seletus ja näitekood.

Õppematerjali lisati praktiline osa, mis koosneb järgnevatest osadest:

- **Ettevalmistus** – Juhend, kuidas lähtekoodi kasutada ja NodeJS serverit kasutada.
- **Konteksti loomine ja varjundajate laadimine –** Seletus, kuidas konteksti luua ja varjundajaid laadida.
- **Esimene kolmnurk** – Õpilase esimene kokkupuude WebGL'ga ja praktiline näide, kuidas renderdamise järjekord töötab.
- **Tipu värv –** Kujundile värvi lisamine. Näidata, kuidas API ja varjundajad omavahel suhtlevad.
- **Indeksite kasutamine –** Lineaarse massiivi asemel indeksite kasutamine. Näidata, et andmeid on võimalik mitut moodi defineerida ja arendaja on see, kes otsustab, mida andmed kujutavad.
- **Maatriksid –** Kuidas maatrikseid renderdamisel kasutada. Näidata, mida saab maatriksitega saavutada. Näide, mida kujutavad praktiliselt endas koordinaatruumid.
- **Liikumine –** Kuidas muuta staatiline pilt muuta dünaamiliseks. Põgus näide, kuidas kasutada vektoreid ja polaarset koordinaatsüsteemi objekti positsiooni muutmiseks.
- **Tekstuur ja 3D objekt –** Milleks kasutada tekstuure ja kuidas luua 3D objekt. Näidata, et objekt on lihtsalt kolmnurkadest loodud tasapind, mille peale on projekteeritud mingisugune tekstuur.
- **Hiir –** Veebilehitseja kuularite kasutamine, et rakendus interaktiivseks muuta. Näide, kuidas kasutada sfäärilist koordinaatsüsteemi. Tutvustada Euleri nurkade probleeme.
- **Renderdamine tekstuurile –** Näide, kuidas kasutada tekstuuri sellele renderdamiseks. Eesmärk on ümber lükata arusaam, et tekstuur on ainult staatiline pilt. Näidata ka, mis on tegelikult kaadripuhver.
- **Valgus –** Lihtsa valgustusmudeli loomine. Näidata, milles peitub tegelikult varjundajate võimsus.

- **Lisaülesanded –** Ülesanded õppurile, kes tahaks raskemaid probleeme lahendada või ei suuda pärast materjali läbimist otsustada, mida järgmiseks ette võtta.

Testimine

Versiooni testisid kolm inimest. Testimine toimus kahe testijaga personaalselt, kus vajadusel juhendati, kuid põhiliselt jälgiti, kuidas testija materjali kasutab ja mis probleemid võivad esineda struktuuris, sõnastuses ja esitluses. Praktilist ja teoreetilist osa testisid kaks inimest ning üks inimene testis ainult teoreetilist osa, sest ei omanud piisavat programmeerimise kogemust.

Testimise käigus selgus, et inimesed lähenevad materjalile erinevalt. Üks testijatest luges põhjalikult teooria osa, teine alustas kohe praktika. Praktilist osa lugesid testijad soravalt. Probleeme tekkis ainult siis, kui vaja oli leida teooria osast teema, mis seletaks praktika osas loodud koodi. Juhendamisel leidsid testijad teemad ülesse ja järgnevatest peatükkides oskasid materjali ise leida. Selgus, et eelneva testi käigus tehtud otsus materjali erinevad osad lahus hoida oli õige.

Testijatel tekkis kohati probleeme, kuidas materjalis orienteeruda, mis näidiskood avada ja millist koodiosa lugeda, mistõttu oli vaja testijatel autorilt instruktsioone.

Testimisel selgusid materjalis leiduvate lausete ebaselge sõnastus. Üks testijatest tõi välja, et peatükkide lõpus võiks olla ka pildid, mis annaks talle võimaluse tulemust näha ilma, et peaks näidiskoodi käivitama.

Lahendades ülesandeid leidis üks testija, et peatükkides võiks olla ka keerulisemaid ülesandeid, mille lahendamiseks peab rohkem vaeva nägema.

Näitekoodiga suutsid testijad suhteliselt rahuldavalt ümber käia, kui võtame arvesse fakti, et WebGL'i kasutades tuleb kirjutada palju koodi, et midagi saavutada. Testijad viisid näitekoodi sisse ka muudatusi ja suhtusid sellesse väga positiivselt.

Järeldus

Õppijad lähenevad õppematerjalile erinevalt ja leiavad erinevaid puudujääke. Enesekindlamad testijad asuvad pigem näitekoodi kallale, kui loevad põhjalikult materjali läbi. Hea on materjaliga kaasa anda peatükkides käsitletud näitekood, mis on mõistlik kommenteerida selliselt, et õppija sinna ära ei eksiks. Probleeme võib tekkida, kui materjali liigselt tükeldada, mis võib viia olukorrani, kus õppija ei suuda materjalis enam orienteeruda. Mõistlik on õppematerjaliga kaasa anda õpijuhend.

5.4.3. Iteratsioon III

Materjalis paremini orienteerumiseks lisati peatükk „Kuidas õppida?”, mis tooks välja ühe võimalikust õpiteest, kuidas õppuril oleks võimalik materjali õppida.

Materjali teoreetilisse osasse sai lisatud peatükk „Terminite ingliskeelsed vasted ja lühiseletused“, mis võimaldab õppuril terminite tähendused kiiresti üle vaadata.

Praktilisesse osasse lisati iga peatükki lõppu pilt, mis võimaldaks tulemust näha ilma, et peaks lähtekoodi käivitama. Lisaks sai täiustatud ülesandeid.

Täiustatud sai näitekoodi selliselt, et peatükis käsitletud teema oleks selgesti eraldatud ja kiiresti leitav.

Materjali lisati peatükk „Alternatiivne materjal ja õppimise jätkamine“, mis annaks õppijale lisavalikuid ja uusi õpiväljundeid.

Kokkuvõte

Suurem osa tegevusest arvuti taga toimub tänapäeval veebilehitsejas. Kasvanud on nõudlus keerulisemate graafikarakenduste järele otse veebilehitsejas, mis vajavad suurt jõudlust. Lisaks on graafika programmeerimise huvilisele, kes alles alustab teema õppimist, OpenGL'i õppimine suhteliselt raske, eriti kui puudub C ja C++ keelega kogemus. Mõlemad probleemid suudab lahendada WebGL. Siiani puudub teema kohta eestikeelne õppematerjal.

Bakalaureusetöö eesmärgiks oli koostada õppematerjal, mille abil omandada baasteadmised WebGL'st, näidata, kuidas WebGL'i koos Javascriptiga kasutada, et luua interaktiivne graafikarakendus veebilehitsejas ja saada sissejuhatus graafika programmeerimisse. Eesmärk oli koostada õppematerjal viisil, et seda oleks hea kasutada erineva tasemega ja õppimisstiiliga õppijatel.

Käesolevas töös tutvustati WebGL'i ja selle API ajalugu. Toodi välja veebilehitsejate ja graafikaprotsessorite toetusega seonduvad probleemid. Anti ülevaade alternatiividest, mida WebGL'i asemel on võimalik kasutada. Analüüsiti erinevaid olemasolevaid õppematerjale ning toodi välja erinevused WebGL'i ja OpenGL'i vahel. Viimases peatükis käsitleti materjali koostamisprotsessi.

Materjal koostati lähtudes SAM-mudelist (*Successive Approximation Model*), mis võimaldas õppematerjali koostada iteratiivselt, kaasates tööprotsessi ka õpilased, kes pidevalt tagasisidet andsid.

Testimise käigus selgus, et õppijate tase varieerub, mistõttu jagati materjal teoreetiliseks ja praktiliseks, et parandada õppematerjali voolavust ja teemal püsimist. Loodud lahendus muutis õppematerjali komplekssemaks ja vaja oli luua õpijuhend. Väga positiivseks osutus õppematerjaliga kaasas olev näitekood, mida oli võimalik lihtsalt käivitada ja muuta.

Bakalaureusetöö käigus valmis õppematerjal, mille abil on autori arvates võimalik õppijal saada baasteadmised, kuidas kasutada WebGL'i interaktiivsete graafikarakenduste loomiseks. Autor leiab, et õppematerjal on heaks hüppelauaks ka graafika programmeerimisse, et järgnevalt raskemad materjalid ette võtta. Tulevikus soovib autor jätkata materjali täiustamist ja parandamist.

Õppematerjali koostamine andis autorile edasi paremad teadmised WebGL'st ja graafika programmeerimise valdkonnast. Autorile andis bakalaureusetöö juurde uusi teadmisi ka õppematerjali koostamise protsessist, mis tulevikus loodavate materjalide kvaliteeti kindlasti parandab.

Summary

This Bachelor Thesis is titled "Using WebGL to Create Interactive Graphics Application in Web Browser: Study Material".

Nowadays majority of time in front of computer is spent using a web browser. Demand of complex graphics applications with great performance needs in web browser is steadily increasing. Additionally beginners who start to learn graphics programming using OpenGL have a great deal of difficulty, especially when they have no experience with C++ and OpenGL. Using WebGL solves both of those problems. Right now there is no material for WebGL in Estonian.

Purpose of this Bachelor Thesis was to create study material that would help to learn the basics of WebGL and graphics programming, which would enable learners to create interactive graphics application in web browser.

Study material was developed using SAM model (Successive Approximation Model). The model allows to develop products using iterative process, which enables rapid development and constant feedback from the learners.

During development it was found that learners' levels vary greatly. To make the study material more readable for different types of learners, author divided it into theoretical and practical. Dividation made the material more complex and study guide was created to solve the problem. Including source code with every practical chapter was positively received.

The first chapter of the thesis provides an overview of WebGL. Second chapter introduces and analyzes alternatives to WebGL. Third chapter analyzes readily available study materials, which learners can use at this moment to learn WebGL and OpenGL. Fourth chapter provides overview of SAM model. Fifth and final chapter describes development of the study material in iterations.

As a result of this Bachelor Thesis author developed study material, which he finds is suitable for learning the basics of how to make interactive graphics application in web browser using WebGL. Additionally author finds the study material to be a good starting point to learn about graphics programming.

Kasutatud kirjandus

Khronos Group. (13. aprill 2014). *WebGL and OpenGL Differences.* Kasutamise kuupäev: 17. november 2014.a., allikas https://www.khronos.org/webgl/wiki/WebGL_and_OpenGL_Differences

Khronos Group. *OpenGL Overview.* Kasutamise kuupäev: 17. november 2014.a., allikas https://www.opengl.org/about/

Khronos Group. (10. aprill 2011). *Getting Started.* Kasutamise kuupäev 19. november 2014.a., allikas https://www.khronos.org/webgl/wiki/Getting_Started

Dorland, J., Besten, W. (2014). *De opkomst van WebGL.* Kasutamise kuupäev 19. november 2014.a., allikas http://www.students.science.uu.nl/~3685632/content/history.html

Deveria, A. *WebGL – 3D Canvas graphics.* Kasutamise kuupäev 20. november 2014.a., allikas http://caniuse.com/#search=webgl

Akeley, K., Segal, M. (11. august 2008). *The OpenGL Graphics System: A Specification (Version 3.0 - August 11, 2008).* Kasutamise kuupäev 20. november 2014.a., allikas https://www.opengl.org/registry/doc/glspec30.20080811.pdf

Khronos Group. (1. oktoober 2014). *BlacklistsAndWhiteLists.* Kasutamise kuupäev 20. november 2014.a., allikas https://www.khronos.org/webgl/wiki/BlacklistsAndWhitelists

Chromiumi autorid. (2013). *software_rendering_list_json.cc*. Kasutamise kuupäev 23. november 2014.a., allikas https://code.google.com/p/chromium/codesearch#chromium/src/gpu/config/software_rendering_list_json.cc

Mozilla. (5. august 2013). *Blocklisting/Blocked Graphics Drivers.* Kasutamise kuupäev 23. november 2014.a., allikas https://code.google.com/p/chromium/codesearch#chromium/src/gpu/config/software_rendering_list_json.cc

Google. (9. veebruar 2012). *GPU accelerating 2D Canvas and enabling 3D content for older GPUs.* Kasutamise kuupäev 23. november 2014.a., allikas http://blog.chromium.org/2012/02/gpu-accelerating-2d-canvas-and-enabling.html

Boesch, F. (8. juuni 2014). *Some issues with Apples iOS WebGL implementation.* Kasutamise kuupäev 25. november 2014.a., allikas http://codeflow.org/entries/2014/jun/08/some-issues-with-apples-ios-webgl-implementation/

Khronos Group. *Conformant Products.* Kasutamise kuupäev 25. november 2014.a., allikas https://www.khronos.org/conformance/adopters/conformant-products#opengles

AppBrain. *Top Android SDK versions. Kasutamise kuupäev 25. november 2014.a., allikas* http://www.appbrain.com/stats/top-android-sdk-versions

Google. *OpenGL ES.* Kasutamise kuupäev 25. november 2014.a., allikas http://developer.android.com/guide/topics/graphics/opengl.html

Boesch, F. (2. veebruar 2013). *Why you should use WebGL.* Kasutamise kuupäev 29. november 2014.a., allikas http://codeflow.org/entries/2013/feb/02/why-you-should-use-webgl/

Luten, E. (24. mai 2014). *What is OpenGL?* Kasutamise kuupäev 01. detsember 2014.a., allikas http://openglbook.com/chapter-0-preface-what-is-opengl.html

Turner, F. (28. oktoober 2011). *Stage3D vs WebGL Performance.* Kasutamise kuupäev 03. detsember 2014.a., allikas http://www.airtightinteractive.com/2011/10/stage3d-vs-webgl-performance/

Lal, R. (november 2011). *Developing 3D Objects in Silverlight.* Kasutamise kuupäev 03. detsember 2014.a., allikas http://msdn.microsoft.com/en-us/magazine/hh547098.aspx

W3C. Scalable Vector Graphics (SVG). Kasutamise kuupäev 04. detsember 2014.a., allikas http://www.w3.org/Graphics/SVG/

Learning WebGL. *Learning WebGL - 3D Programming for the Web.* Kasutamise kuupäev 05. detsember 2014.a., allikas http://learningwebgl.com

GameDev.net. (2012). *Legacy Tutorials.* Kasutamise kuupäev 12. detsember 2014.a., allikas http://nehe.gamedev.net/

Blender. (10. veebruar 2012). *Manual.* Kasutamise kuupäev 20. jaanuar 2015.a., allikas http://wiki.blender.org/index.php/Doc:ET/2.6/Manual

Khronos Group. (12. mai 2009). *The OpenGL ES Shading Language 1.0 Specification*. Kasutamise kuupäev 25. jaanuar 2015.a., allikas https://www.khronos.org/files/opengles_shading_language.pdf

Dunn, F., Parberry, I. (2011). *3D Math Primer for Graphics and Game Development (2nd Edition).* Boca Raton: CRC Press

Sellers, G., Wright, S. R., Haemel, N. (2013). *OpenGL SuperBible Sixth Edition*. Crawfordsville: RR Donnelley

Gregory, J. (2009). *Game Engine Architecture.* Boca Raton: CRC Press

Allen, W. M. (2012). *Leaving Addie for Sam: An Agile Model for Developing the Best Learning Experiences*. Alexandria: ASTD Press

Lisad

Lisad 1. Õppematerjal

Bakalaureusetöö käigus koostatud õppematerjal on kättesaadav veebiaadressitelt:

- Õppematerjal – https://github.com/ranerp/webgl-sissejuhatus/wiki
- Lähtekood - https://github.com/ranerp/webgl-sissejuhatus/

Lisad 2. Õppematerjal paberkandjal

Tallinna Ülikool

Informaatika Instituut

WebGL’i kasutamine interaktiivsete graafikarakenduste loomiseks veebilehitsejas.

Õppematerjal

Autor: Raner Piibur

Tallinn 2015

Sisukord

Sissejuhatus

Tere tulemast, WebGL'i ja graafika programmeerimise huviline!

Käesolevalt leheküljelt leiab õppematerjali algajale, kes soovib omandada baasteadmised WebGL'st ja saada sissejuhatuse graafika programmeerimisse. Materjali läbides peaks õppijal tekkima parem ettekujutus, kuidas tänapäeval läbi API (*Application Programming Interface*) GPU'iga suhtlemine käib ja kuidas näeb välja renderdamise järjekord. WebGL'ga seotud teema on väga suur ja lai ning me ei saa käsitleda absoluutselt kõike, mistõttu on koostatud materjal küllaltki kokkuvõtlik, kuid siiski piisavalt informatiivne.

WebGL'i kasutatakse selleks, et kuvada 3D maailm 2D tasandile. WebGL ei ole erinevalt *legacy* OpenGL'st fikseeritud funktsionaalsusega, mis meie eest igasugused maatriksite ja vektoritega seotud toimetused ära teeks, vaid kõik on vaja programmeerija poolt ise defineerida. Sellest asjaolust tulenevalt on vaja omandada teadmisi graafika programmeerimisest ja sellega seonduvast matemaatikas ning füüsikast. Kogu kompott on alguses kindlasti väga raskesti seeditav, kuid sihikindlus viib võidule. Ei tasu pead norgu lasta!

Meie materjal on ennekõike pühendatud WebGL'le, mistõttu käsitleme maatrikseid, trigonomeetriat jms väga kokkuvõtvalt, et lugejal tekiks üldine pilt, mida nad endast kujutavad. Soovitatav on kindlasti pärast materjali läbimist ette võtta mõni korralik lineaaralgebra materjal (uuri näiteks *khanacademy.com* või *coursera.org*).

Kui lugeja eesmärgiks on tulevikus WebGL'i ja graafika programmeerimisega tõsisemalt tegelema hakata, on soovitatav iga peatükk koos lisamaterjaliga korralikult läbi seedida. Autori enda kogemustest võib öelda, et seda tegemata tekib suuri raskusi uute algoritmide/tehnikate implementeerimisega ja probleemide esinemisel *debugimisega*.

Arvan, et pärast materjali läbimist, tunnete ennast kindlasti kodusemalt mõne mahukama ja keerulisema materjali lugemisel.

Mõisted

API – Hulk toimetusi, protokolle ja töövahendeid, mille abil arendatakse tarkvara.

OpenGL (Open Graphics Library) - Madalatasemeline API, mille abil suhelda graafikaprotsessoriga. OpenGL on kõige rohkem kasutatud 2D ja 3D graafika API selles tööstusharus.

OpenGL ES (OpenGL for Embedded Systems) – Alamhulk OpenGL API'st. Leiab kasutust mobiilsetes seadmetes.

WebGL (Web Graphics Library) – Baseerub OpenGL ES API'i. Jooksutatakse HTML5 Canvas elemendis. On toetatud kõikides suuremates veebilehitsejates (Internet Explorer alates versioonist 11).

Graafika järjekord – Renderdamisprotsessi erinevad etapid graafikaprotsessoris. Tavalised etapid on näiteks: värvi ja valguse kalkuleerimine GLSL/HLSL programmide abil, perspektiivi projekteerimine, akna pügamine (aknast väljas olevate objektide või nende osade eemaldamine) ja renderdamine.

GLSL (OpenGL Shading Language) – Baseerub ANSI C keelel. Keelt on laiendatud vektor ja maatriks tüüpidega ning kohendatud paralleelselt jooksmiseks. GLSL'i kasutatakse graafika järjekorras varjundprogrammides.

Varjundaja (Shader) – Programm, mis jookseb graafika järjekorras kindlal etapil. OpenGL'i baseeruvates API'ides kirjutatakse programm GLSL keeles, Direct3D puhul HLSL keeles.

Tipuvarjundaja (Vertex Shader) – Käsitleb tippude töötlemist. Tavaliselt toimub selles etapis tippude transformeerimine objektruumist projektsioonruumi e. kolmemõõtmeline tipp projekteeritakse kahemõõtmelisse ruumi (olenevalt tehnikast, mida kasutatakse, ei pruugi see muidugi alati nii olla). Tipuvarjundajas toimub ka muude atribuutide eelsätestamine järgmiste etappide jaoks.

Pikslivarjundaja (Fragment Shader) – Käsitleb piksli/fragmendi töötlemist. See etapp on viimane programmeeritav etapp. Lihtsamal juhul toimub varjundajas vaid fragmendi värvi määramine, mis siis ekraanile kuvatakse. Tavaliselt toimub etapis ka valgusarvutused, varjude lisamine jms.

Terminite ingliskeelsed vasted ja lühiseletused

Terminid on alfabeetilises järjekorras.

emiteeriv - emissive: Valguskomponent, mida kiirgab objekt.

hajus - diffuse: Valguskomponent, mis on ühtlaselt hajutatud üle terve pinna.

homogeenne jagamine - homogeneous division: Jagamine, mis teostatakse, et 3D punkt 2D ruumi viia.

kaadripuhver - frame buffer: Puhver, kuhu renderdatakse. Kaadripuhver koosneb omakorda teistest puhvritest.

kaameramaatriks - camera matrix, view matrix: Maatriks, mida kasutame kaameraruumi defineerimiseks.

kaameraruum - camera space: Koordinaatsüsteem, kus kaamera paikneb.

koordinaatruum - coordinate space: Meie käsitleme kui kolme vektorit, mis moodustavad koordinaatsüsteemi.

küllastunud - ambient: Valguskomponent, mis täidab terve ruumi.

läikiv - specular: Valguskomponent, mis peegeldub valgusallikalt vaataja silma.

maailmaruum - world space: Süsteem, kus kõik objektid ja kaamerad paiknevad.

mudelmaatriks - model matrix, object matrix: Maatriks, millega viia objekt või mudel maailmaruumi.

normaliseeritud seadme ruum - normalized device coordinates: Ruum, mida tunnevad kõik seadmed ja koordinaadid on vahemikus [-1, 1].

objektruum - object space: Süsteem, kus objekti tipud suhtes üksteisega paiknevad.

perspektiivi projekteerimine - perspective projection: 3D ruumis paikneva tipu 2D ruumi viimine.

pikslivarjundaja - fragment shader (OpenGL), pixel shader (Direct3D): Varjundaja, kus teostatakse arvutusi kolmnurgas paiknevate pikslite/fragmentide peal.

puhver - buffer: GPU'is paiknev mäluhulk, mingite andmetega.

projektsioonmaatriks - projection matrix, clip matrix: Maatriks, millega tipud pügamisruumi viia ja neid pügamiseks ette valmistada.

pügamisruum - clip space: Pügamisruumi defineerib tüvipüramiid või risttahukas ja selle abil on võimalik määrata, mis objektid paiknevad meie poolt vaadeldud ruumis.

renderdamise järjekord - graphics pipeline, WebGL pipeline, OpenGL pipeline: Samm etappe, mis renderdamiseks läbitakse.

tipp - vertex: kolme- või kahemõõtmeline punkt mingis ruumis.

tipuvarjundaja - vertex shader: Varjundaja, kus teostatakse arvutusi tippude peal.

tüvipüramiid - frustum: Defineerib pügamisruumi.

vaateakna ruum - screen space: Aken (*canvas*), kuhu renderdatakse.

varjundaja - shader: Programmeeritav ala renderdamise järjekorras.

Kuidas õppida?

Kindlasti võib ette tulla mitmeid asju, millest aru ei saa või vajaks detailsemat seletust ja julgelt tasub lugeda lisamaterjali, mis teooria osas on välja toodud.

Võid alustada õppimist nii teooria lugemisega kui ka praktika tegemisega. Iga praktika osa alguses on viidatud teooria osale, mida peaks siiski enne lugema. Iga praktilise osaga käib kaasas lähtekood, mida on võimalik uurida ja puurida päikeseloojanguni. Alljärgnevalt leiad ühe võimalikust õpiteest, kuidas materjali kasutada.

Lähtekood

Lähtekood on meie põhiline õppevahend. Lähtekood on kommenteeritud detailideni, nii et õppur, kes on oskuslik lähtekoodi lugeja võib saada päris palju informatsiooni, lugedes lihtsalt lähtekoodi.

Lähtekood asub WIKI'st eelneval URL'il: https://github.com/ranerp/webgl-sissejuhatus.

Igal tunnil on oma lähtekood. Apache kasutajatel on see **builds/lesson00/js/bundle.js** ja NodeJS kasutajatel on see **lessons/lesson00/main.js**

Igas tunnis on meil ka erinevad varjundajad, mis asuvad vastava tunni kaustas.

- Tipuvarjundaja: **vertex.shader**
- Pikslivarjundaja: **fragment.shader**

Lähtekoodi struktuur on järgmine:

```
//////////////////Osa, mis loob konteksti ja laeb meie varjundajad serverist//////////////

See osa tegeleb meile tüütu osaga ehk varjundajate laadimisega ja konteksti loomisega.
See osa ei muutu peatükkide vältel. Muutuvad vaid failide asukohtade väärtused.

////////////////////////// LESSON00 - ESIMENE KOLMNURK ////////////////////////////////////

Tegu on põhiosaga, mis peatükkide käigus areneb ja muutub. Siin on võimalik koodi katsetada
ja muuta.

/////////////////////////////////////////LÕPP/////////////////////////////////////////////
```

Samm 1

Käesoleva materjali omandamiseks on soovitavad alustuseks lugeda teooriast järgnevad peatükid:

- Renderdamise järjekord
- Varjundajad

Need peatükid peaksid andma ettekujutuse, kuidas WebGL töötab. Neid teemasid tasub niikaua uurida, kuni tekib selge ülevaade, kuidas renderdamise järjekord töötab.

Samm 2

Alustada praktikaga.

Tõmmata või kloonida (olenevalt GitHub kogemusest) lähtekood aadressilt: https://github.com/ranerp/webgl-sissejuhatus.

Kasutada võib NodeJS serverit või Apache serverit nagu WAMP(Windows) või LAMP(Linux). Algajatel, kellel ei ole Javascript'is kogemusi on kindlasti parem kasutada Apache serverit.

Kasutades **Apache't** tuleb kopeerida **builds** kaust oma **Apache** serveri kausta.

```
Apache lähtekood -> <peakaust>\builds\

Kopeerida kausta, mida Apache serveerib -> C:\wamp\www\builds\
```

Selles kaustas asuvad vastavate tundide lähtekood.

Kui kasutate NodeJS serverit, siis räägime pikemalt osas „Ettevalmistus“. Sel juhul asuvad tundide lähtekoodid järgnevas kohas:

```
<peakaust>\lessons\
```

Samm 3

Tööta kõik praktika peatükid korralikult läbi. Paralleelselt uuri teooriat ja ära karda asju ka mujalt uurida. Kõike ei pruugi kohe mõista ja mõistmine võib saabuda hilinemisega.

Teooria osas on ära toodud WebGL’i meetodid ja näitekood, kuidas meetodit kasutada. Kui ei ole täpselt selge, mida see meetod teeb, siis uuri seda sealt.

Uuri näitekoodi. Muuda näitekoodi. Kui on välja toodud ülesanded, lahenda need. Ürita olla loominguline ja ise midagi teha. Võib selguda, et mingi asi ei ole teostatav või ei ole piisavalt teadmisi. Võib juhtuda, et järsku hakkad mõistma, miks on asju just sellisel viisil lahendatud.

Samm 4 või 5

Hakka lahendama praktika viimases peatükis välja toodud lisaülesandeid. Kui need ülesanded ei sobi mõtle ise midagi välja.

Samm 5 või 4

Loe välja toodud alternatiivseid või jätkamiseks mõeldud materjale.

1. Teooria

Teooria osas räägime kokkuvõtvalt meile vajalikust matemaatikast: Vektorid, maatriksid, koordinaatsüsteemid, valgustusmudel ja koordinaatruumid. Teooria osas räägime ka täpsemalt, mis on renderdamise järjekord, varjundajad, tekstuurid ja tipuandmed.

Selle osa juurde tasub alati tagasi pöörduda, kui praktika osas leiad, et ei mõista asja täielikult. Teooria, mis puudutab graafika programmeerimist on tegelikult väga lai ja suur teema, seega on selles osas käsitletud teemad väga kokkuvõtvad ja jätavad nii mõnegi detaili välja, seega tulevikus tasub kindlasti lugeda materjali, mis neid teemasid põhjalikult seletavad.

1.1. Matemaatika

Selles alapeatükis käsitleme matemaatikat, ilma milleta ei ole WebGL'iga võimalik praktiliselt mitte midagi saavutada. Valgustusmudeli osas puudutame ka optikat, mis kuulub pigem füüsika valdkonda, kuid otsustasime panna siiski matemaatika osasse.

Kindlasti kasuta õppimiseks ka paberit ja pliiatsit. Maatriksite paremaks õppimiseks konstrueeri näiteks kahedimensiooniline maatriks, mis defineeriks koordinaatsüsteemi ja ürita mõni objekt sinna süsteemi viia.

Kui sa suudad teemasid seostada enda kogemustega või objektidega enda elust oled tõenäoliselt asjadest aru saanud.

1.1.1. Vektorid

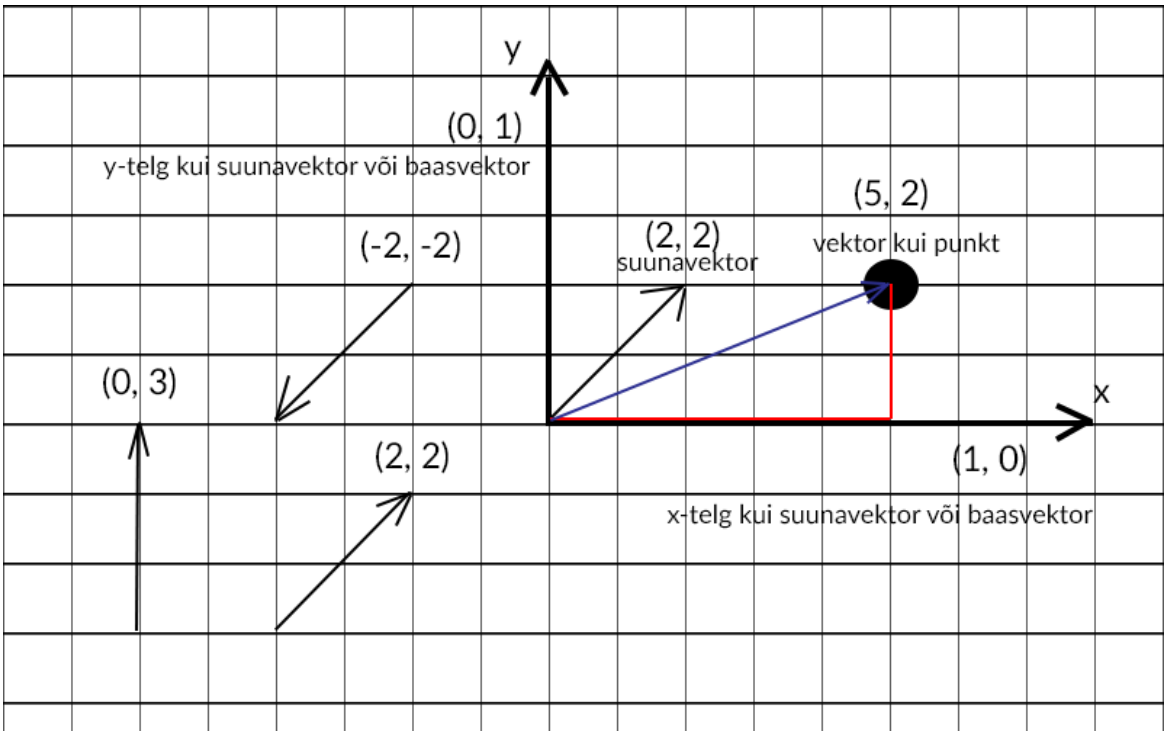
Vektor on massiiv numbritega. Vektoris olevate numbrite arv tähistab vektori dimensiooni. Geomeetriliselt võib vektorit vaadata kui suunatud sirglõiku, millel on pikkus ja suund. (Dunn, Parberry, 2011)

- Vektori *pikkus* näitab selle lõigu pikkust. Pikkus on skalaar.
- Vektori *suund* näitab, mis suunas vektor ruumis näitab.

Skalaar on tavaline arv. Näiteks kiirus 1000 km/h või kaal 67 kg on skalaarväärtused.

Vektor tähistab mitut asja ja see on meie enda teha, mida me täpselt mingi vektoriga defineerime (vt joonist 3). Oletame, et meil on vektor **(0, 1)**. Seda vektorit võime tõlgendada mitut moodi:

- Vektor võib olla suunavektor. Näiteks võib osutada, mis suunas tuul puhub. Joonistades vektori näiteks kaardi peale, millel on x- ja y-telg, näeme, et vektor osutab põhja ja seega tuul puhub lõunast põhja ehk tegu on lõunatuulega.
- Sama vektor võib näidata ka nihutamist 1 ühiku võrra mööda y-telge. Olles punktis (2, 2) ja liikudes 1 ühiku võrra mööda y-telge, satume koordinaatidele (2, 3).
- Vektor võib olla ka üks baasvektoritest ristkoordinaatide süsteemis ehk defineerida ühte koordinaatsüsteemi telge.
- Vektor võib tähistada punkti süsteemis. Vektorit võib vaadelda ka kui nihutamist ristkoordinaatide süsteemi (0, 0) punktist. Kasutades eelpool defineeritud vektorit (0, 1), liigume kõigepealt 0 ühiku võrra mööda x-telge ja 1 ühiku võrra mööda y-telge, sattudes koordinaatidele (0, 1).



Joonis 1 2D vektorid

Tehted vektoritega

- **Korrutamine skalaariga**

```
k = 2
V = (0, 2)
kV = (2* 0, 2*2) => (0, 4)
```

- **Vastandvektor**

```
V = (1, 2)
-V = (-1, -2)
```

- **Vektorite liitmine**

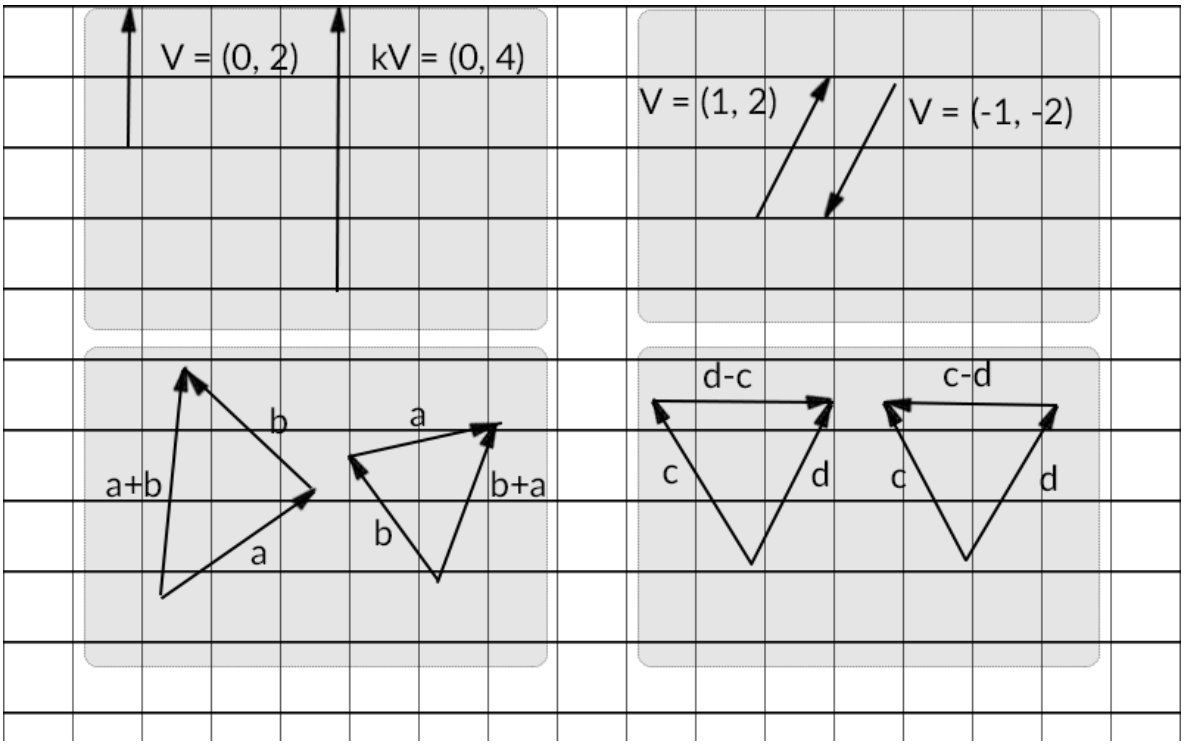
  Liitmine on kommutatiivne.

```
A = (2, 3)
B = (5, 7)
A + B = B + A = (2 + 5, 3 + 7) => (7, 10)
```

- **Vektorite lahutamine**

  Lahutamine on antikommutatiivne. vektor A - B osutab vastasuunas vektorile B - A. Tehte abil on võimalik leida kaugus või suund kahe vektori vahel.

```
A = (3, 1)
B = (4, 2)
A - B = (-1, -1)
B - A = (1, 1)
```



Joonis 2 Tehted vektoritega

- Vektorite skalaarkorrutis

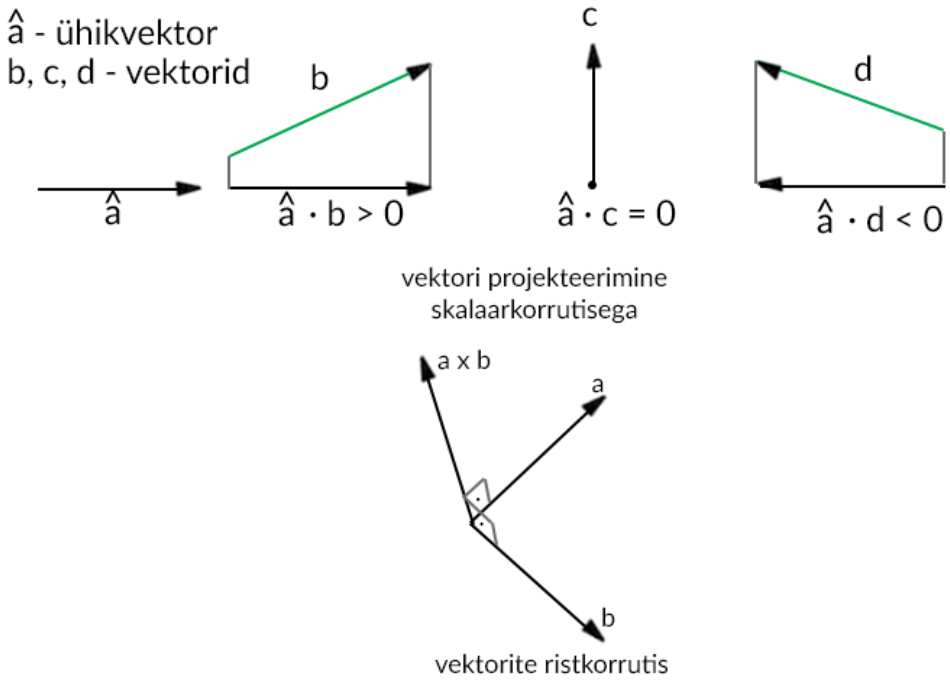
  Skalaarkorrutise abil on võimalik üks vektor projekteerida teisele vektorile (vt joonist 3). Ühikvektorite skalaarkorrutamisel on võimalik leida ka nurk nende vahel. Teemast räägime pikemalt **Valgus ja valgustusmudel** peatükis.

```
A = (2, 1)
B = (3, 1)
A · B = 2*3 + 1*1 = 7
```

- Vektorite ristkorrutis

  Vektorite ristkorrutise abil on võimalik leida vektor, mis on mõlema vektoriga risti (vt joonist 3).

```
x1     x2     y1z2 - z1y2
y1  x  y2  =  z1x2 - x1z2
z1     z2     x1y2 - y1x2

[2  3  4] x [1  5  -6] =
[(3)*(-6) - (4)*(5)
(4)*(1)  - (2)*(-6)
(2)*(5)  - (3)*(1)] =

= [-38  16  7]
```



Joonis 3 Vektorite skalaar ja ristkorrutis

Ühikvektor

Ühikvektori pikkus on 1. Suunavektorid on tavaliselt ühikvektorid. Ühikvektori hea omadus on see, et näiteks skalaarkorrutades seda vektorit teise vektoriga, ei lisandu uuele vektorile magnituudi ja saadakse teise vektori projekteeritud pikkus ühikvektorile (vt joonist 3).

Lisamaterjal

- http://chortle.ccsu.edu/VectorLessons/vectorIndex.html
- http://www.mathsisfun.com/algebra/vectors.html

1.1.2. Maatriksid

Maatriksid on graafika programmeerimise juures kõige tähtsamad objektid. Nende abil saame teiste objektidega manipuleerida. Neid liigutada, pöörata, viia 3D ruumist 2D ruumi jne. Lineaaralgebras defineeritakse maatriksid, kui ristkülikukujulist ruudustikku numbritest, millest moodustuvad read ja veerud (Dunn, Parberry, 2011). Kui vektor on ühe-dimensiooniline massiiv, mis sisaldab endas skalaare, siis maatriksit võib geomeetrias kujutada kui massiivi, mis sisaldab endas vektoreid.

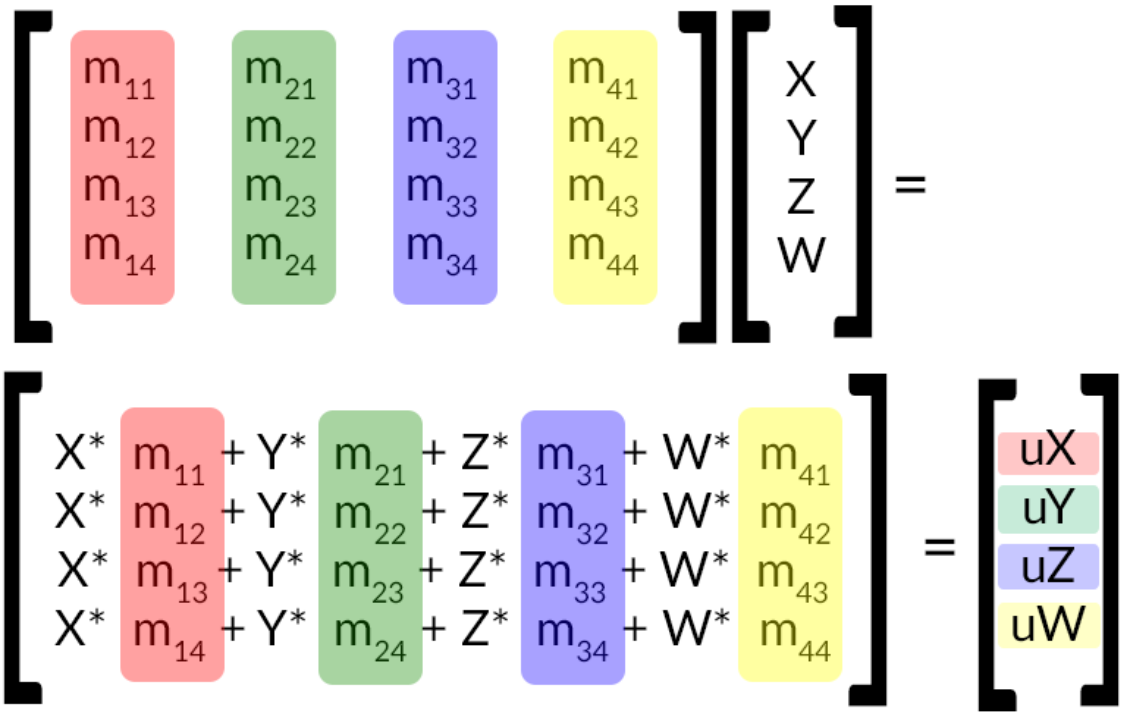
Nagu teada on meie maailm kolmemõõtmeline ja seega on mõistlik arvutis kujutada objekte kolmemõõtmelisena, millel on x, y, z kooordinaadid. Tegelikult aga jääb kolmest mõõtmest väheks. Nimelt kui meil on vaja objekti pöörata või tema mõõtkava suurendada/vähendada, piisab meile 3x3 maatriksist, kuid transleerimist ei saa 3x3 maatriksi korrutamisel sooritada. Seetõttu tuleb meil maatriksid ja vektorid viia homogeensesse ruumi (*homogeneous space*), kus on neli mõõdet.

Võib tunduda, et see ruum on väga keeruline, kuid lähemalt vaadates on see tegelikult väga lihtne. Viimast skalaari w saab kasutada selleks, et ladustada sinna translatsioon, mis võimaldab meil defineerida translatsiooni maatriksid. Skalaari w kasutatakse ka selleks, et punkt projekteerida mingisse ruumi jagades komponendid (x/w, y/w, z/w) skalaariga läbi.

w = 0.0 | vec4(x, y, z, 0.0) - tegemist on suunavektori, mitte punktiga.

w = 1.0 | vec4(x, y, z, 1.0) - tegemist on punktiga, mis asub ruumis.

Võttes vektori $\mathbf{x = (x_1; x_2; x_3)}$ ja korrutades selle maatriksi reaga (baasvektorite x koordinaadid) $\mathbf{b = (b_1; b_2; b_3)}$ saame lineaarkombinatsiooni $x_1b_1 + x_2b_2 + x_3b_3$. Olles kolmemõõtmelises ruumis ei ole meil maatriksi teguritesse võimalik salvestada translatsiooni ilma, et see objekti x-telje suhtes ei väänaks ega keeraks. Seetõttu kasutame 4x4 maatriksid, et viimasesse w veergu salvestada translatsioon.

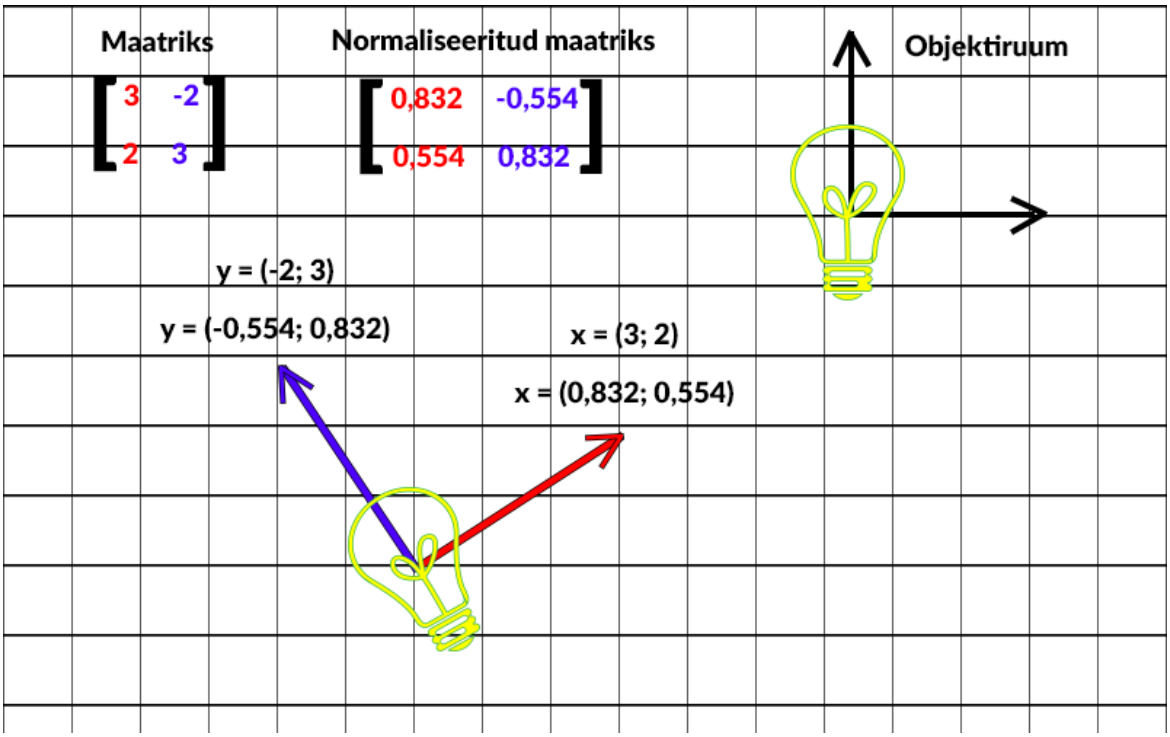


Joonis 1 Maatriksi ja vektori korrutamine

OpenGL ja WebGL kasutavad veerumaatrikseid. Kindlasti on koolis õpitud ristkoordinaatide süsteemi, mida kasutatakse ka graafika programmeerimises. Mõistagi on süsteemis kolm telge: x, y ja z. Maatriksis võime veerge interpreteerida telgedena (vt joonist 1). Maatriksi korrutamine mingi vektoriga või teise maatriksiga viib nad selle maatriksi koordinaatsüsteemi (vt joonist 2). Korrutamisel toimub ka objekti skaleerimine ning seetõttu on koordinaatsüsteemide baasvektorid normaliseeritud kujul (ühikvektorid). Mõõtkavaga manipuleerimiseks kirjutame skalaarid maatriksi diagonaali, mida saab ka vajadusel kombineerida teise maatriksiga:

```
[X  0  0  0
 0  Y  0  0
 0  0  Z  0
 0  0  0  1]
```

Vektor on normaliseeritud, kui tema pikkus on 1. Normaliseerimiseks jagame kõik vektori komponendid läbi tema pikkusega. Pikkuse saame teada tuntud Pythagorase teoreemi abil.



Joonis 2 Maatriksi abil transformeerimine

WebGL's kasutatakse veerumaatrikseid. Maatriksite korrutamine on antikommutatiivne ja seega tuleb maatriks korrutada vektoriga ja mitte vastupidi. Kui meil on vaja vektor viia objektruumist maailmaruumi ja sealt edasi kaameraruumi tuleb see sooritada järgmises järjekorras: kaameramaatrix * maailmamaatrix * vektor (objektruumis).

Maatriksis olevad baasvektorite koordinaadid ei ole midagi muud, kui tegurid lineaarkombinatsioonis, mille abil iga vektori koordinaat projekteerida vastavale teljele. Kõik projekteeritud koordinaadid kokku moodustavad uue vektori, mis paikneb transformeerimiseks kasutatud maatriksi koordinaatsüsteemis.

Arvutusnäide

Arvutamiseks kasutame maatriksit, mille koordinaatsüsteemi teljed on järgmised:

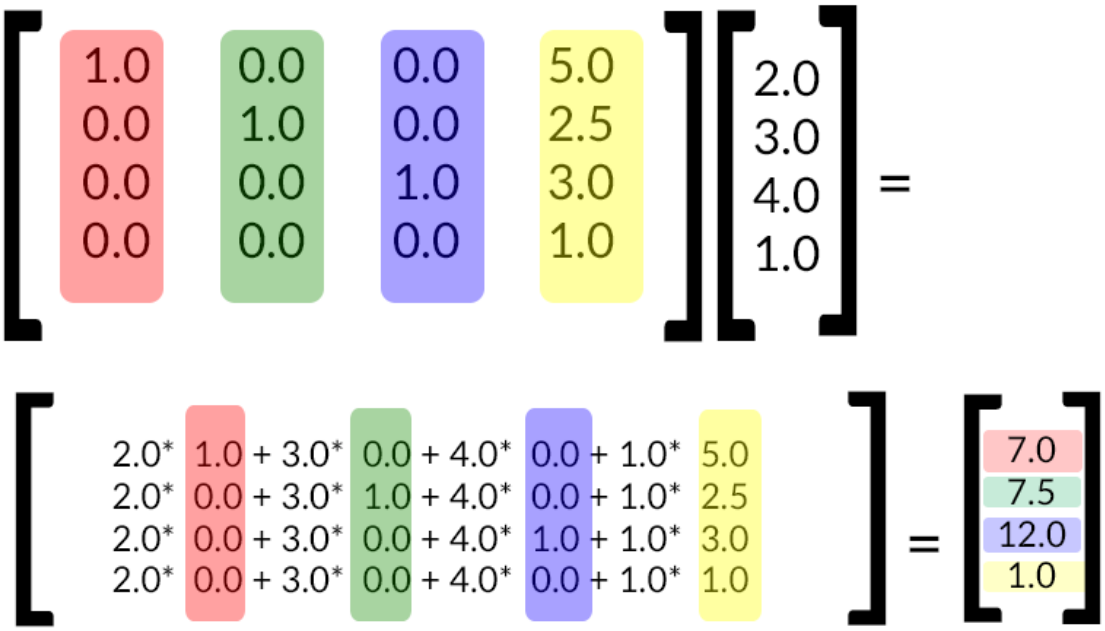
- x = (1; 0; 0; 0)
- y = (0; 1; 0; 0)
- z = (0; 0; 1; 0)

Maatriksi vektorisse **w = (5; 2.5; 3; 1)** on salvestatud ka translatsiooni osa, mis vektorit transleerib. See maatriks liigutab vektorit x-teljel 5 ühikut, y-teljel 2,5 ühikut, z-teljel 3,0 ühikut.

Vektor, mida tahame projekteerida ja transleerida on **v = (2; 3; 4; 1)**. Kuna viimane **w** komponent on 1.0 on tegu punktiga ruumis, mistõttu saab teda ka transleerida.

Arvutuskäik mõttes võib olla järgmine (vt joonist 3):

1. Kirjutan vektori transponeeritud kujul *[2.0 3.0 4.0 1.0]* maatriksi ülesse.
2. Maatriksi iga reaga ja vektoriga teostan skalaarkorrutise.
3. Iga skalaarkorrutisest saadud skalaar on uue vektori üks komponent.



Joonis 3 Maatriksi ja vektori korrutamine

Uuri kindlasti: http://www.realtimerendering.com/udacity/transforms.html

Ühikmaatriks (*identity matrix*)

On maatriks, mille peadiagonaalis olevad elemendid on 1 ja ülejäänud 0. Korrutades vektorit ühikmaatriksiga ei muutu vektor.

Pöördmaatriks (*inverse matrix*)

Kui maatriksil eksisteerib pöördmaatriksit saame seda kasutada, et eelnev transformatsioon nullida.

Lisamaterjal

- http://www.realtimerendering.com/udacity/transforms.html
- http://ocw.mit.edu/courses/mathematics/18-06-linear-algebra-spring-2010/video-lectures/
- http://www.arcsynthesis.org/gltut/Positioning/Tutorial%2006.html
- http://www.in.tum.de/fileadmin/user_upload/Lehrstuehle/Lehrstuhl_XV/Teaching/SS07/Praktikum/MatricesTips.pdf
- http://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/
- 3D Math Primer for Graphics and Game Development (2nd Edition) Chapter 4 - Chapter 6.

Ülesanded

1. On maatriks:

```
1  0  0  10
0  1  0  5
0  0  1  0
0  0  0  0
```

   Mida on võimalik selle maatriksiga teha?

2. Rakendades objektile maatriksit:

```
2  0  0  0
0  1  0  0
0  0  2  0
0  0  0  0
```

   Mis juhtub objektiga?

3. Kombineeri ülesannetes 1 ja 2 olevad maatriksid. Milline on kombineeritud maatriks?

4. Kas maatriksis olevad vektorid on normaliseeritud kujul?

```
1.1  0    0    0
0    1.5  0    0
0    0    3.0  0
0    0    0    0
```

5. Loe lisamaterjali!

Märkused

- Teekides, mis sooritavad tehteid maatriksiga salvestavad arve järjest. Seega baaskvektorid võib massiivis vaadelda kui ridadena nt x = (m[0], m[1], m[2], m[3]). Siiski arvutamine toimub samamoodi nagu sai seletatud siin materjalis.

1.1.3. Polaarne ja sfääriline koordinaatsüsteem

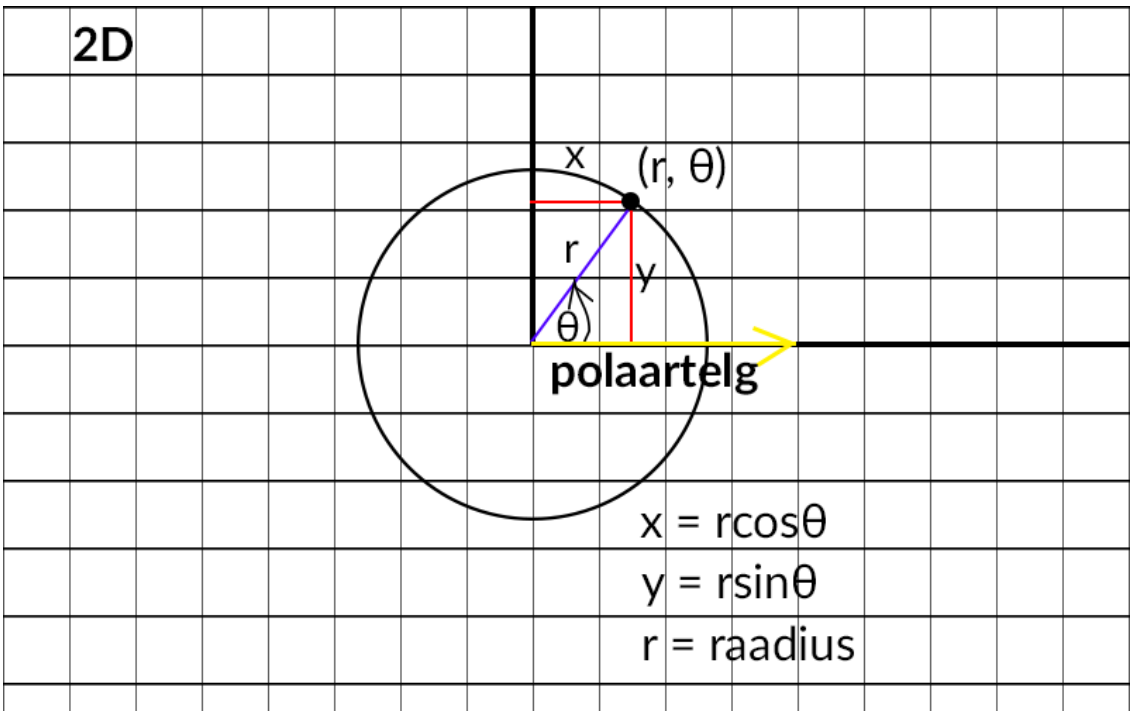
Objektide defineerimiseks ruumis kasutame Cartesian'i koordinaatsüsteemi ehk ristkoordinaatide süsteemi, mille alternatiiviks oleks polaarne või sfääriline koordinaatsüsteem. Nende süsteemidega on võimalik implementeerida näiteks erinevaid kaameraid.

Polaarne koordinaatsüsteem

Polaarses koordinaatsüsteemis defineeritakse punkti kahe muutujaga, raadius **r** ja nurk **θ**, mis moodustavad paari **(r, θ)**. See süsteem on kahemõõtmeline ja süsteemil on üks telg, mida nimetatakse polaarteljeks (Dunn, Parberry, 2011). Telg osutab tavaliselt paremale (vt joonist 1), kuid olenevalt vajadusele võib telge defineerida selliselt, et see osutaks objekti vaatamise suunda (3D mängumootorites).

Punkti määramiseks kahemõõtmelises ruumis võib seda ette kujutada järgmiselt:

- Vaata polaartelje osutamise suunas, ise asudes lähtepunktis.
- Pööra ennast defineeritud nurga võrra.
- Liigu edasi raadiuse võrra.
- EUREKA! Oled kohal.



Joonis 1 Polaarne koordinaatsüsteem

Polaarses koordinaatsüsteemis eksisteerib selline fenomen nagu *alias* ehk igat punkti on võimalik defineerida lõpmatul arvul viisil. Punkt (1, 0) on sama, mis (1, 360) või (-1, 180). Fakti tuleb meeles pidada, kui polaarkoordinaadid on vaja viia ristkoordinaatide süsteemi või vastupidi. Tavaliselt kasutatakse paari defineerimiseks kanoonilist kuju, mis rangete reeglitega polaarkoordinaadi sobivasse kujusse viiks (praktika osas toome välja, kuidas polaarkoordinaadid kanoonilisse kujusse viia). Lisaks konverteerides polaarkordinaate ristkoordinaatide süsteemi, tuleb kasutada trigonomeetrilistel arvutustel mitte **atan** funktsiooni, vaid **atan2** meetodit (punkt võib asuda ükskõik, millises veerandis ja arvutuse **y/x** sooritamisel läheb osa informatsioonist kaduma, kui selleks kasutada **atan** meetodit).

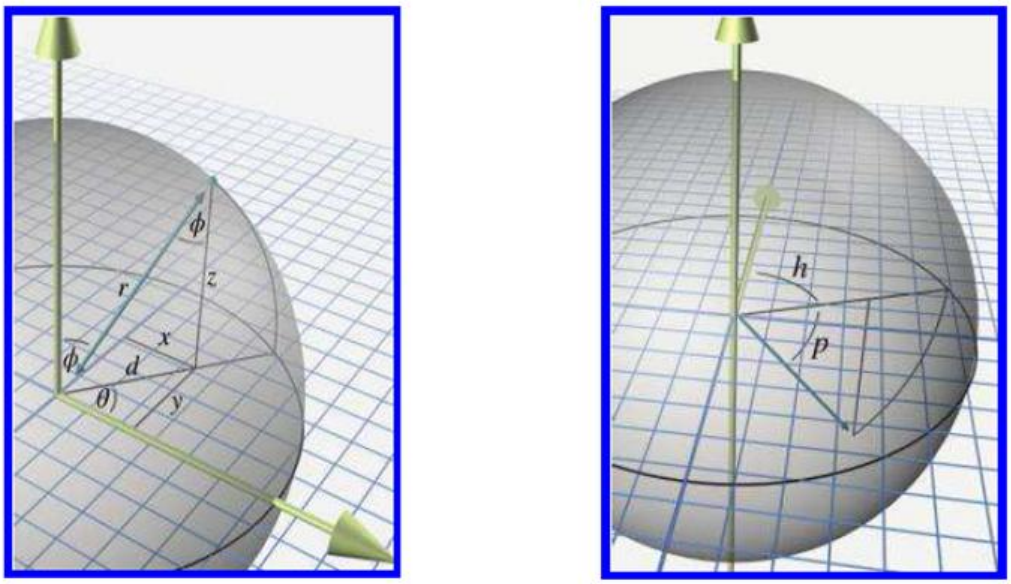
Sfääriline koordinaatsüsteem

Sfäärilises koordinaatsüsteemis defineerime punkti kolme muutujaga raadius **r**, horisontaalnurk **θ** ja vertikaalnurk **φ**, mis moodustavad kolmiku **(r, θ, φ)**. Selles süsteemis on kaks telge: horisontaalne polaartelg, mis osutab ristkoordinaatide süsteemis positiivse x-telje suunas ja vertikaalne polaartelg, mis osutab positiivse y-telje suunas. (Dunn, Parberry, 2011)

Punkti määramiseks selles ruumis tuleb läbida järgmised sammud:

- Seisa lähtepunktis ja vaata suunas, kuhu näitab horisontaalne polaartelg. Vertikaaltelg osutab jalgadest peani.
- Siruta käsi välja samas suunas, kuhu näitab vertikaaltelg ehk otse ülesse.
- Liigu vastupäeva horisontaalnurga võrra.
- Liiguta kätt alla vertikaalnurga võrra.
- Liigu raadiuse võrra käega osutatud suunas.
- EUREKA! Oled kohal.

Sarnaselt polaarse koordinaatsüsteemiga on vaja ka sfäärilised koordinaadid viia kanoonilisse kujusse (implementeerimist vaata lähtekoodist), sest ka selles süsteemis esineb *alias*. Tuleb kasutada ka **atan2** meetodit.



Joonis 2 Sfääriline koordinaatsüsteem. (Dunn, Parberry 2011)

Ristkoordinaadistikku teisendamine

Jooniselt 2.1 näeme, kuidas matemaatikas tavaliselt polaarkoordinaate tähistatakse ja neile vastavad ristkoordinaatide süsteemi koordinaadid leitakse.

Jooniselt 2.2 näeme, kuidas 3d graafikas tavaliselt polaarnurki defineeritakse. Vertikaalnurk tähistab tõusunurka ja horisontaalnurk, mis tähistab nurgamuutust näoga osutatud suunast.

Leiame ristkoordinaadid joonise 2.1 järgi:

```
d = rsin(ϕ)

x = dcos(θ) = rsin(ϕ)cos(θ)
y = dsin(θ) = rsin(ϕ)sin(θ)
z = rcos(ϕ)
```

Leiame ristkoordinaadid joonise 2.2 järgi:

```
d = rcos(h)

x = dsin(p) = rcos(h)sin(p)
y = sin(p)
```

```
z = dcos(p) = rcos(h)cos(p)
```

Lisamaterjal

- https://www.khanacademy.org/math/precalculus/parametric_equations/polar_coor/v/polar-coordinates-1
- http://www.mathsisfun.com/polar-cartesian-coordinates.html
- http://mathinsight.org/spherical_coordinates

Ülesanded

1. Polaarses koordinaatsüsteemis on defineeritud paar **(2.0, 60°)**. Arvuta vastavad koordinaadid ristkoordinaatide süsteemis.
2. Ristkoordinaatide süsteemis on defineeritud koordinaadid **(0, 5.0)**. Arvuta vastavad koordinaadid polaarses koordinaatsüsteemis.
3. Sfäärilises koordinaatsüsteemis on defineeritud kolmik **(5.0, 45°, 90°)**. Arvuta vastavad koordinaadid ristkoordinaatide süsteemis.
4. Ristkoordinaatide süsteemis on defineeritud koordinaadid **(5.0, 4.0, 3.0)**. Arvuta vastavad koordinaadid sfäärilises koordinaatsüsteemis.

1.1.4. Valgus ja valgustusmudel

Mis on valgus? Valgust defineeritakse kui elektromagneetilist radiatsiooni kindlal lainepikkusel, mida inimene suudab tajuda. Olenevalt materjalist võib mingi osa lainest materjalis neelduda. Seetõttu olenevalt, mis lainepikkust valgusallikas kiirgab ja materjal neelab, peegeldub inimsilma tagasi kindel lainepikkus, mida me tajume ja defineerime kui värvust.

Värvusi ei ole mõistlik arvutigraafikas defineerida lainepikkustega, vaid parem on kasutada RGB-mudelit. Seda mudelit kasutavad monitorid, telekad, videokaamerad jne. Mudelis on defineeritud kolm komponenti: R (punane), G (roheline), B (sinine), mida saab varjundajates defineerida järgmiselt:

```
vec4(1.0, 1.0, 1.0, 1.0)
```

Neljas komponent defineerib läbipaistmatust.

Kui tavaliselt inimesed on harjunud, et värvid on vahemikus 0-255, siis varjundajates on see normaliseeritud ehk vahemikus 0-1.

Valgustusmudel

Reaalselt peegeldub valgus mitme objekti pealt, enne kui ta jõuab inimese silma. Kui me arvestaks renderdamisel igat valguskiirt ja tema teekonda, muutuksid kalkulatsioonid väga keerukaks ja ressurssinõudlikuks, mida tänapäeval rahuldava kaadrisagedusega ei ole reaalajas jooksvas rakenduses võimalik saavutada. Aastal 1975 tutvustas Bui Tuong Phong valgustusmudelit, mis on kiire, piisavalt reaalne ja lihtne. Mudelis on neli komponenti: emiteeriv (*emmissive*), peegelduv (*specular*), hajus (*diffuse*) ja küllastunud (*ambient*) (Dunn, Parberry, 2011).

- **Emiteeriv** - Radiatsiooni kogus, mida objekt kiirgab tasapinnast defineeritud suunas. Ei sõltu valguskiire langemise suunast.
- **Peegelduv** - Valgus, mis tuleb otse valgusallikast ja peegeldub tasapinnast nagu täiesti sile peegel vaataja silma. Annab objektile läikiva välimuse.
- **Hajus** - Valgus, mis tuleb allikast mingi nurga all ja hajutub ühtlaselt üle pinna.
- **Küllastunud** - Valgus, mis täidab tervet ruumi. Komponent imiteerib mitteotsest valgust, mis ruumis mitmeid kordi põrkub.

Joonis 1 Valguse komponendid. (Dunn, Parberry 2011)

Mudelis omistatakse igale objektile materjal, mis defineerib objekti omadusi. Materjali atribuutideks on eelpool mainitud neli komponenti ja lisaks ka näiteks läige (mida läikivam on materjal, seda suurem on peegelduv punkt).

Valguse arvutamine

Mudelis ei võta me arvesse mitteotsest valgust ja arvestame vaid kiirtega, mis peegelduvad objektilt meie silma. Esiteks on meil vaja teada valguse ja materjali värvuseid. Vaja on teada ka:

- valgusallika kiirgamise suunda
- punkti koordinaate
- punkti tasapinna normaalvektorit
- vaataja koordinaate

Arvutuseks kasutatav valem on järgmine: **lõppvalgus = küllastunud + hajus + emiteeriv + peegelduv**

· tähistab vektorite skalaarkorrutist

küllastunud = (küllastunud$_{ruum}$ + küllastunud$_{materjal}$)max(n·l, 0)

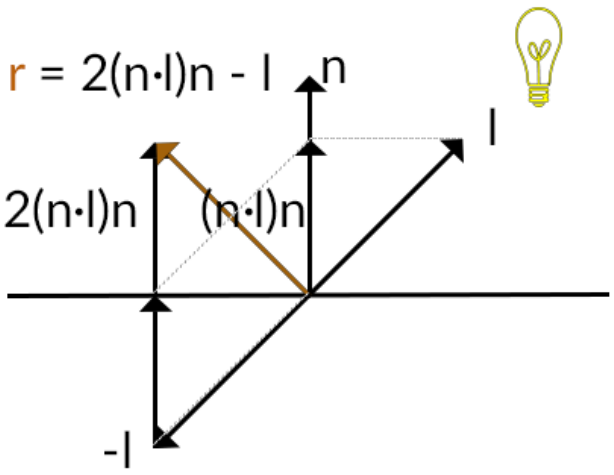
hajus = hajus$_{valgusallikas}$ + hajus$_{materjal}$

emiteeriv = emiteeriv$_{materjal}$

peegelduv = (peegelduv$_{valgusallikas}$ + peegelduv$_{materjal}$)max(v·r, 0)$^{m}_{läige}$

Kuidas arvutada kiire peegeldus?

Kasutades vektoreid ja nende seaduseid on suhteliselt lihtsalt võimalik välja arvutada objektilt peegeldunud kiir (vt joonist 2). Varjundajates on võimalik kasutada *reflect* meetodit, kuid see arvutuskäik on vaja endale selgeks teha, sest näitab mitmeid printsiipe, kuidas tehted vektoritega käivad ja miks on meil üldse normaalvektoreid vaja.



l = suunavektor punktist valgusesse

r = reflekteeritud suunavektor

(n·l)n = punkti normaalvektorile projekteeritud l

Joonis 2 Valguse peegeldumine

Arvutuskäik:

- **(n·l)n**: Teades suunavektorit punktist valgusallikani on meil võimalik see projekteerida punkti normaalvektorile. Vektorite skalaarkorrutise **(n·l)** abil on meil võimalik leida vektori **l** projekteeritud pikkus kõikidele vektoritele, mis on paralleelsed vektoriga n. Korrutades pikkuse normaalvektoriga **n** saame me normaalvektorile projekteeritud vektori **(n·l)n**.

- **-l**: Kuna täiuslikult peegeldunud vektori projekteeritud pikkus normaalvektoriga risti olevale tasapinnale on vastassuunaline vektorile **l**, leiame **-l** vahetades vektori **l** märgid.
- **2(n·l)**: Kui projekteerida **-l** normaalvektorile on näha, et liikusime mööda normaalvektorit **-2(n·l)** pikkust alla, kuid meil oleks vaja liikuda vastasuunas, milleks arvutame 2(n·l)n.
- **r**: Vektorite kolmnurga reeglist teame, et kahe vektori liitmisel tekib vektor, mille alguspunkt asub esimese vektori algpunktis ja lõpp-punkt teise vektori lõpp-punktis. Arvutame **-l + 2(n·l)n => 2(n·l)n – l**

Hajus- ja peegeldusfaktorite arvutamine

Kui meil on teada reflekteeritud vektor on meil võimalik arvutada hajus- ja peegeldusfaktorid, mis näitavad, kui suur panus on nendel komponentidel koguvalgusele.

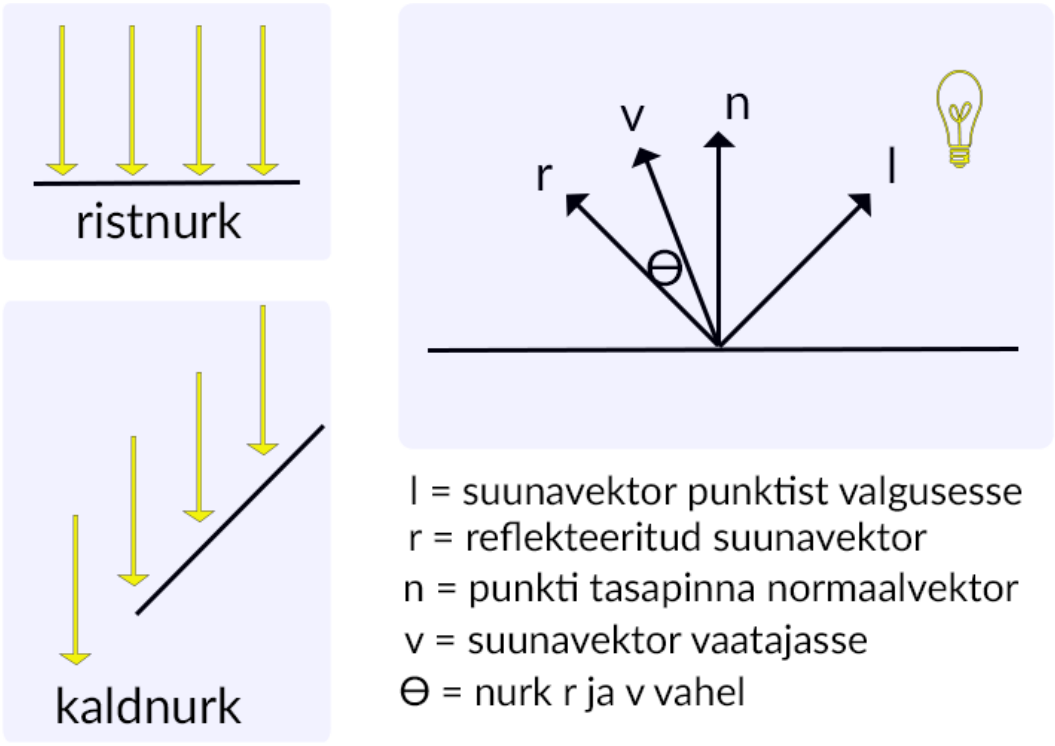
Meil on teada fakt, et tasapind, mida kiired tabavad ristnurga all, saab rohkem valgust, kui tasapind, mida kiired tabavad kaldnurga all (vt joonist 3). Kui me reaalselt vaatame, kuidas valgus peegeldub mingilt läikivalt materjalilt, siis näeme, et mida väiksem on nurk peegeldunud vektori ja meie silma suunatud vektori vahel, seda läikivam on see punkt.

Vektorite skalaarkorrutisega saame ühe vektori projekteeritud pikkuse teisele vektorile. Kui tegu on ühikvektoritega (mis vastab tavaliselt tõele), siis skalaarkorrutis on vahemikus **-1 < (a·n) < 1**. Suhtelise nurga saab määrata lihtsalt:

- **(a·n) = 0 ehk Θ = 90º**: vektorid on risti.
- **(a·n) < 1 ehk 90º < Θ ≤ 180º**: vektorid osutavad laias laastus vastassuunas.
- **(a·n) > 0 ehk 0º ≤ Θ < 90º**: vektorid osutavad laias laastus samassuunas.

Läikivuse arvutamiseks arvutame nurga vektorite **v** ja **r** vahel (vt joonist 3). Saadud tulemuse saame astendada ka materjali läikega või läike eksponendiga, mille abil on võimalik läikeomadusi hästi kontrollida. Seega peegelduskomponenti saame kontrollida korrutades selle läikefaktoriga: $\mathbf{max(v \cdot r, 0)^{m_{läige}}}$.

Hajuvuskomponendi kontrollimiseks on meil hajuvusfaktor. Mida väiksem on nurk **n** ja **l** vahel, seda "eredam" on objekt. Hajuvusfaktor on lihtne: **max(n·l, 0)**



Joonis 3 Peegeldus

Lisamaterjal

- http://http.developer.nvidia.com/CgTutorial/cg_tutorial_chapter05.html
- 3D Math Primer for Graphics and Game Development (2nd Edition) Chapter 10.6 - 10.7.

Ülesanded

1. Kui reflekteeritud vektori ja vaataja suunalise vektori vaheline nurk on täisnurk, siis mis on läikefaktor?
2. Punkti normaalvektori ja valgusallika suunalise vektori vaheline nurk on nürinurk, siis mis on hajuvusfaktor?
3. Ruumis on kaks valgusallikat, mis simuleerivad päikest ja mille kiired langevad erinevast suunast. Meil on teada punkti normaalvektor n ja valgusallikate l_1 ja $l_2$suunavektorid punktist. Kumma valgusallika läikefaktor on suurem?
    - n = (0.0, 1.0, 0.0)
    - l_1 = (0.707, -0.707, 0.0)
    - l_1 = (-0.707, -0.707, 0.0)

1.1.5. Koordinaatruumid

Renderdamise järjekorras konverteeritakse tipud mitme ruumi vahel enne, kui nad jõuavad kaadripuhvrisse.

Kõige parem seletus on tõenäoliselt illustreeriv näide reaalsest maailmast. Oletame, et meil on tool, mis paikneb ruumis. Tooli **objektruum** võib olla näiteks tema tagumine vasak jalg, mille koordinaadid on (0, 0, 0). Kui me liigume mööda jalga üles muutub meie y koordinaat. Jõudes seljatoeni, võime paikneda näitlikult koordinaatidel (0, 40cm, 0).

Tool paikneb muidugi toas ja meie tuba on **maailmaruum**. Oletame, et meie toa alumine vasak nurk põrandal tähistab koordinaate (0, 0, 0). Tool asub nurgast ühe meetri kaugusel paremal ja kuna ühik, mida me kasutame on sentimeeter on tooli koordinaadid maailmaruumis (100cm, 0, 0). Tooli seljatoe koordinaadid maailmaruumis on seega (100cm, 40cm, 0). Tooli enda ruumist maailmaruumi viimiseks kasutame **mudelmaatriksit**.

Järgmiseks tahame viia toa ja seal paikneva tooli vaatesse, mida näeb meie silm ehk **kaameraruumi**. Asume me näiteks akna taga ja vaatame ruumi. Koordinaadid (0, 0, 0) tähistavad seda, kus paikneb meie silm. Meie silm asub ülevalpool tooli maailmaruumi koordinaatidel (200cm, 100cm, 0) ja meie vaatest on tooli seljatoe koordinaadid (-100cm, -60cm). Maailmaruumist silmavaatesse viimiseks kasutame **kaameramaatriksit**.

Kui me tahaks kõike seda, mida me näeme joonistada, on meil vaja kasutada **projektsioonmaatriksit**.

Järgmiste ruumide ja konverteerimistega tegeleb *API* ja neid me iseseisvalt muuta ei saa.

Vaade

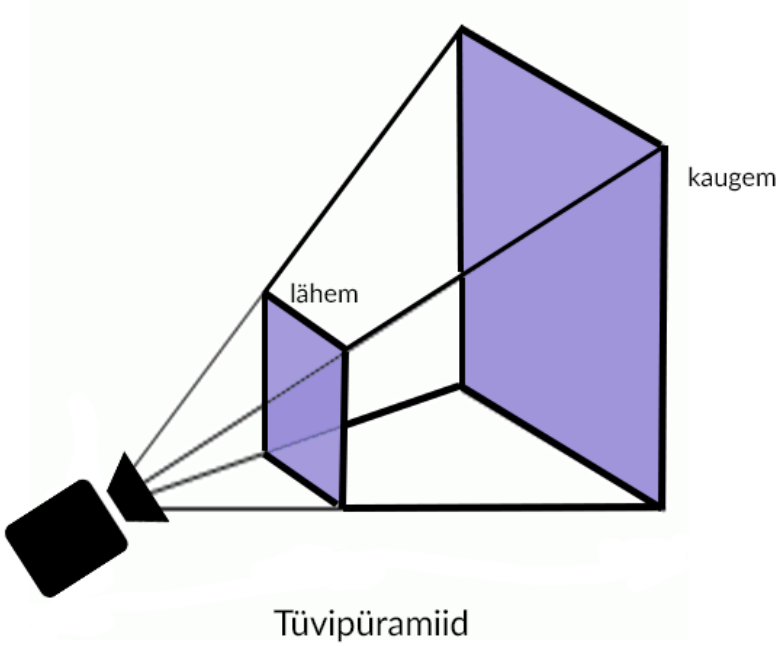
Vaade defineerib, mis tipud on kaamerale nähtavad. Vaade, kus arvestatakse ka perspektiivi on kujult ära lõigatud tipuga püramiid ehk tüvipüramiid (vt joonist 1).

Ilma perspektiivita vaade on ortograafiline vaade, mis on kujult risttahukas. Vaate tippude koordinaat **w** ei muutu. Perspektiivi ei arvestata ja seega objekti kaugusest ei olene objekti mõõtmed. Kasutatakse 2D maailma loomiseks.

Tüvipüramiidil on 6 tasandit, mida nimetatakse ka pügamistasanditeks:

- Lähem (*near*) - Kaamerale lähim tasand z-teljelt vaadatuna.
- Kaugem (*far*) - Kaamerale kaugeim tasand z-teljelt vaadatuna.

- Vasak (*left*) - Kaamerast vasakule jääv tasand x-teljelt vaadatuna.
- Parem (*right*) - Kaamerast paremale jääv tasand x-teljelt vaadatuna.
- Ülemine (*up*) - Kaamerast ülesse jääv tasand y-teljelt vaadatuna.
- Alumine (*down*) - Kaamerast alla jääv tasand y_teljelt vaadatuna.



Joonis 1 Perspektiivi projekteerimiseks kasutatav tüvipüramiid

Tüvipüramiidi tasandid defineeritakse lähema ja kaugema tasandi koordinaatidega z-teljel ja kaamera vertikaalse vaatenurgaga.

Objektruum

Objekti kohalik ruum. Valmistades CAD- või 3D modelleerimistarkvaras mõne objekti, asub ta objektiruumis.

Maailmaruum

Ruum või stseen, kus kõik objektid paiknevad. Tipu koordinaadid asuvad maailmas.

Objektiruumist maailmaruumi saame me mudelmaatriksiga.

Kaameraruum

Kaameraruum on vaade maailmale läbi kaamerasilma. Tipud paiknevad koordinaatidel vaadates neid läbi kaamerasilma.

Maailmaruumist kaameraruumi saame kaameramaatriksiga.

Pügamisruum

Pügamisruumi konverteeritakse tipud joonisel 1 näidatud tüvipüramiidi abil (ortograafilise projektsiooni korral risttahukas), mis defineeritakse projektsioonmaatriksi abil. Kõik tipud, mis ei asu püramiidis pügatakse ja nad ei jõua pikslivarjundajasse. Tipuvarjundaja väljastab tippe pügamisruumi.

Enne pügamisruumi oli tipu **w** koordinaat 1.0. Pügamisruumis paiknevates tippudes on **w** koordinaadis vajalik info, et tipp 2D ruumi projekteerida. Pügamisruumis olevatele tippudele on teostatud kaamera suum (*field of view*). Pügamisruumis on tipud viidud ülilihtsale kujule, mille abil on võimalik triviaalselt määrata, kas tipud paiknevad defineeritud maailmas või mitte.

Vajalik info tipu koordinaati **w** saadakse projektsioonmaatriksi korrutamisel, kus toimub tehe **w** = **z**.
Tipp on nähtaval, kui täidetud on järgmised tingimused:

```
(vasak tasand) -w  ≤  x  ≤  w (parem tasand)
(alumine tasand) -w  ≤  y  ≤  w (ülemine tasand)
(lähem tasand) -w  ≤  z  ≤  w (kaugem tasand)
```

Kui kõik tipu koordinaadid rahuldavad eelpool toodud tingimused, paikneb tipp ruumis ja pügamist ei toimu.

Kaameraruumist pügamisruumi saame projektsioonmaatriksiga.

Normaliseeritud seadme koordinaadid (*NDC*)

3D ruumist on tipud viidud 2D ruumi. *NDC* ruumi viiakse tipud teostades nende peal homogeenne jagamine ehk x/w, y/w/ ja z/w.

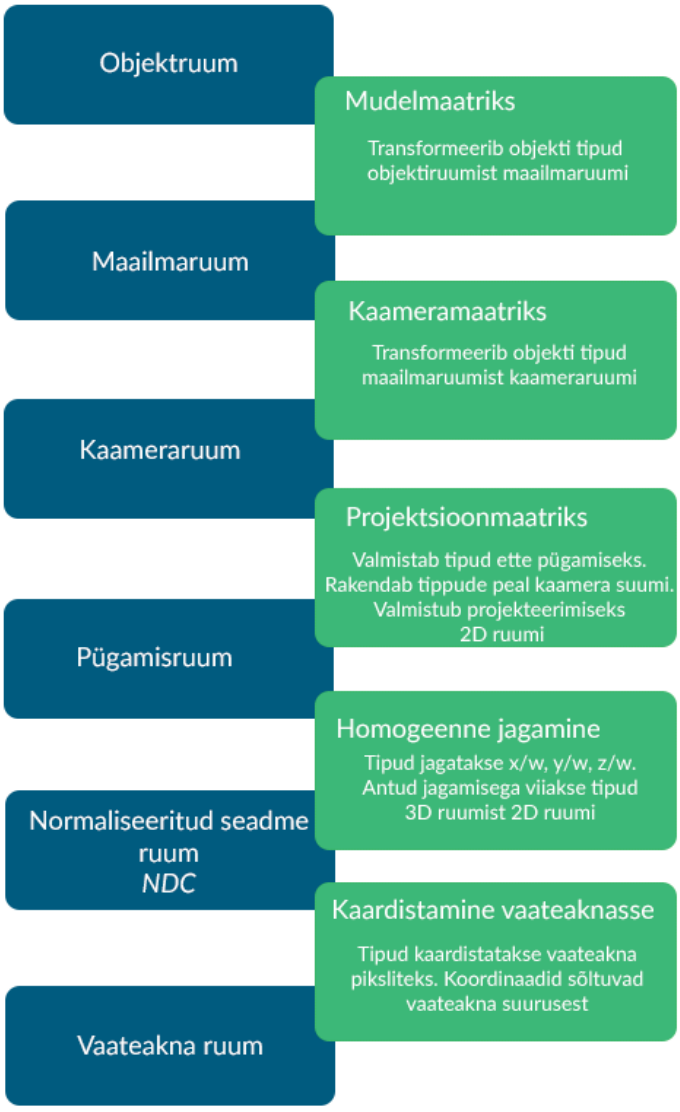
Homogeenne jagamine on 2D ruumi projekteerimine.

Vaateakna ruum

Kui pügamisruumis olevad tipud on pügatud (vaatest väljas olevad tipud on ära visatud), siis kaardistatakse tipud vaateakna ruumi koordinaatidele, mis vastavad pikslile kaadripuhvris.

Selle ruumi koordinaadid on ekraanil olevate pikslite koordinaadid.

Ruumis on koordinaat (0, 0) vasak ülemine äär. Kui siiamaani on y-telje positiivne suund olnud ülesse, siis aknaruumis on alla.



Joonis 2 Tippude konverteerimine renderdamise järjekorras

Uuri kindlasti: http://www.realtimerendering.com/udacity/transforms.html

Lisamaterjal

- http://www.realtimerendering.com/udacity/transforms.html
- http://www.arcsynthesis.org/gltut/Positioning/Tut04%20Perspective%20Projection.html
- http://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/
- http://http.developer.nvidia.com/CgTutorial/cg_tutorial_chapter04.html
- http://www.arcsynthesis.org/gltut/Positioning/Tutorial%2006.html#d0e5963
- http://www.arcsynthesis.org/gltut/Positioning/Tutorial%2007.html#d0e6879
- 3D Math Primer for Graphics and Game Development Chapter 10.2 - 10.3

1.2. WebGL

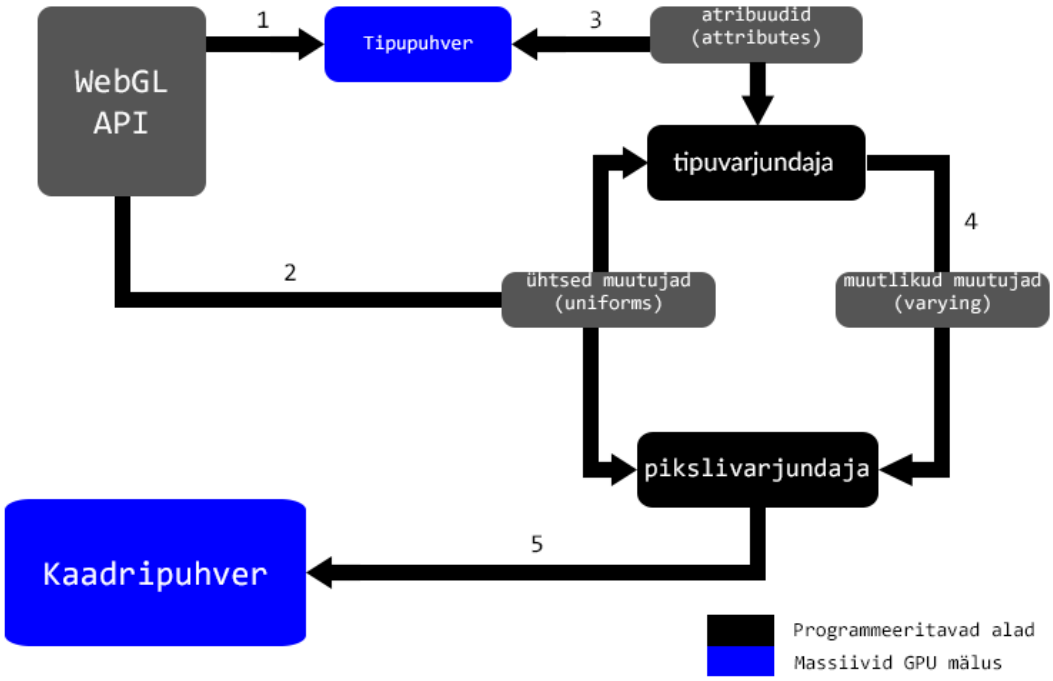
Selles peatükis käsitleme renderdamise järjekorda, varjundajaid, tipuandmeid, tekstuure ning toome välja ka detailsemad näited, kuidas ja kus kasutada erinevaid WebGL funktsioone.

Huvitujatel tasub uurida ka WebGL'i spetsifikatsiooni veebiaadressilt: https://www.khronos.org/registry/webgl/specs/1.0/

1.2.1. Renderdamise järjekord

Rendering pipeline või renderdamise järjekord on rida samme, mis tuleb läbida, et ekraanile mingi objekt renderdada (Khronos Group, 2015). Siin anname lihtsustatud ülevaate sammudest, mida tuleb teada, et materjalis kasutatavast koodist aru saada.

Pea meeles! WebGL tegeleb renderdamise järjekorras vaid osaga, mida oleks käsitsi mõttetu teha. Võrreldes vana OpenGL API'iga ei oma ta mingit kontseptsiooni ruumist, valgusest, objektidest, vaid renderdab seda, mis talle ette söödetakse. Seetõttu on meil olemas ka varjundajad, kus me saame vahepeal andmetega manipuleerida.



Joonis 1 Renderdamise järjekord

1. WebGL API abil laetakse tipupuhvrisse (VBO - *Vertex Buffer Object*) tipuandmeid (positsioon, normaalvektor, värv).
2. WebGL API abil defineeritakse ühtsed muutujad, mida on võimalik kasutada kõikides varjundajates. Ühtsed muutujad jäävad kõikide sammude vältel samaks.
3. Tipuvarjundajas kasutatavad atribuudid (positsioon, normaalvektor, värv) viitavad tipupuhvris asuvatele andmetele.
4. Tipuvarjundajas teostatud arvutusi on võimalik muutlikke muutujate abil pikslivarjundajasse edasi anda.

5. Pikslivarjundajast väljub nt sügavus (depth) ja värvus. Väljuv väärtus kirjutatakse potentsiaalselt kaadripuhvrisse (FBO - *Frame Buffer Object*).

Järjekord algab sellega, et üks või mitu tipupuhvrit täidetakse tipuatribuutidega. Tipuatribuutideks võivad olla positsioon, positsiooni või kolmnurga normaalvektor, tipu värvus või värvused. Tipupuhvrid, mis varustavad järjekorda andmetega nimetatakse tipumassiiviks (*vertex array*). Lisaks tipumassiivile antakse järjekorda kaasa elementide massiiv (*element array*). See massiiv sisaldab endas informatsiooni (indekseid) selle kohta, millised tipumassiivi tipud tuleks töö käigus ette võtta ja millised tipud moodustavad omavahel kolmnurga.

Järgnevalt rakendatakse iga tipu peal tipuvarjundaja programmi. Programmi on võimalik programmeerijal ise defineerida. Tipuvarjundajast väljuvad tipud (muutuja gl_Position) ühendatakse ja nendest moodustuvad kolmnurgad või jooned (kolmnurgad moodustatakse vastavalt elementide massiivis defineeritud järjekorrale). Seda etappi kutsutakse primitiivide montaažiks (*primitive assembly*).

Iga kolmnurk, mis väljub montaažist läbib pügamise (*clipping*). Pügamise käigus kontrollitakse, kas mingi osa kolmnurgast asetseb vaateaknast väljaspool või mitte. Juhul kui mingi osa kolmnurgast asetseb aknast väljaspool, jagatakse kolmnurk väiksemateks tükkideks. Täiesti väljaspool asetsev kolmnurk visatakse ära.

Aknas asuvad kolmnurgad saadetakse edasi rasteriseerimisele. Rasteriseerimise käigus tehakse kindlaks fragmendid (ühele pikslile võib vastate mitu fragmenti, mille põhjal määratakse lõpuks piksli värvus), mis paiknevad kolmnurga sees. Fragmendid saadetakse pikslivarjundajasse.

Pikslivarjundaja programm defineeritakse programmeerija poolt ja tema eesmärk on määrata iga fragmendi värvus, mis saadetakse potentsiaalselt kaadripuhvrisse.

Lisamaterjal

- http://en.wikibooks.org/wiki/GLSL_Programming/OpenGL_ES_2.0_Pipeline
- https://www.opengl.org/wiki/Rendering_Pipeline_Overview
- OpenGL SuperBible Sixth Edition. *Chapter 3 - Following the Pipeline*.

Märkused

- WebGL järjekord erineb OpenGL'i omast selle poolest, et puudub *Tessellation* etapp ja geomeetriavarjundaja.
- Fragmenti võib vaadelda, kui andmeid, mida on vaja, et moodustada piksel. Fragment sisaldab endas lisaks muud informatsiooni, mille põhjal on võimalik teha järelduse, kas kirjutada uus piksel kaadripuhvrisse või mitte.

1.2.2. Varjundajad

Renderdamise järjekorras on kaks varjundajat, mida on võimalik programmeerida: tipuvarjundaja ja pikslivarjundaja. Varjundajad on programmid, mis jooksevad otse graafikaprotsessoris. Programmide jooksutamiseks on vaja mõlemat tüüpi varjundajad kompileerida, siduda Programm Objekt'iga ja seejärel see objekt aktiveerida.

Uuri, mis etappidel varjundajad käivitatakse Renderdamise järjekord peatükist.

Varjundajates on võimalik ligi pääseda tekstuuridele, ühtsetele muutujatele (*uniforms*), ühtsete muutujate blokkidele (*uniform blocks*). Tuleb meeles pidada, et ressursside kasutamisel on piir ja tavaliselt on kindlalt ära määratud iga tüübi maksimaalne arv.

Varjundajaid kirjutatakse *OpenGL Shading Language* (GLSL) abil, mis baseerub ANSI C keelel. WebGL *pipeline* kasutab *OpenGL ES Shading Language 1.0* versiooni. GLSL on küll sarnane, kuid on mõeldud siiski paralleelselt jooksutamiseks ja teheteks maatriksite ja vektoritega. (Khronos Group, 2009)

Kindlasti uurida ja kasutada WebGL juhendkaarti. Huvi korral võib põhjalikumalt uurida ka GLSL spetsifikatsiooni.

Tipuvarjundaja

Tipuvarjundaja tegeleb iga tipu (punkt/koordinaat) töötlemisega, mis saadakse tipupuhvrist. Tipupuhvris töödeldakse tippe, millele elementide massiiv viitab. Tipuvarjundaja saab sisendist ühe tipu ja väljastab tipu ka väljundisse (Khronos Group, 2015). Tipuvarjundajasse on võimalik saata ka muid andmeid, mis on seotud tipuga, mida on siis võimalik kasutada programmis tipu töötlemiseks või edasi saatmiseks pikslivarjundajasse.

Tipuvarjundaja on tavaliselt muutumatu sisendi suhtes. See tähendab, et samad sisendväärtused väljastavad alati samad väljundväärtused.

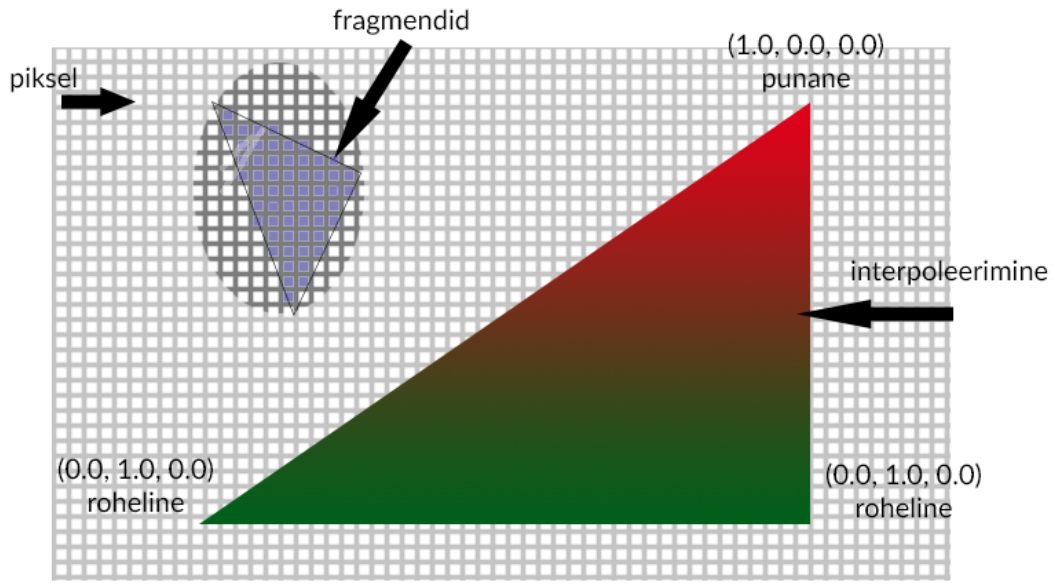
```
precision highp float;
attribute vec3 a_Position;

void main() {
    gl_Position = vec4(a_Position, 1.0);
}
```

Pikslivarjundaja

Pikslivarjundaja e fragmendivarjundaja tegeleb rasteriseerimisest saadud fragmendi töötlemisega (vt joonist 1). Pikslivarjundajasse saadetakse tipuvarjundajast saadud interpoleeritud väärtused (vt joonist 1).

Üks tipp on punane (1.0, 0.0, 0.0) ja teine tipp on roheline (0.0, 1.0, 0.0). Kui fragment asub täpselt kahe tipu vahel, siis interpoleeritud värvus on kollane (0.5, 0.5, 0.0).
Pikslivarjundajast väljastatakse hulk värve, sügavusväärtus ja ka teisi parameetreid, mida me hetkel ei käsitle. Väljastamise kuju on järgmine:

```
gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
```



Joonis 1 Interpoleerimine ja fragmendid

```
precision mediump float;

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

Lisamaterjal

- WebGL juhendkaart - https://www.khronos.org/files/webgl/webgl-reference-card-1_0.pdf
- The OpenGL ES Shading Language 1.0 Specification - https://www.khronos.org/files/opengles_shading_language.pdf

1.2.3. Tipuandmed

Objektide renderdamiseks on vaja WebGL'le anda andmeid, mida saaks renderdada. Vajalik on defineerida protsess, kuidas andmeid kasutada ja defineerida, kuidas andmeid tõlgendada. WebGL API'it kasutava programmeerija on see, kes defineerib ja omastab andmetele mõtte. Seega tuleb rakendust arendades otsustada, mida täpselt on vaja lahendada ja millist struktuuri kasutada.

Pea meeles! WebGL tegeleb renderdamise järjekorras vaid osaga, mida oleks käsitsi mõttetu teha. Võrreldes vana OpenGL API'iga ei oma ta mingit kontseptsiooni ruumist, valgusest, objektidest, vaid renderdab vaid seda, mis talle ette söödetakse. Seetõttu on meil olemas ka varjundajad, kus me saame vahepeal andmetega manipuleerida.

Primitiivid

Massiivis olevaid andmeid on võimalik tõlgendada mitut moodi. Tippude loend võib defineerida järgmiseid primitiive (Sellers *et al.,* 2013):

- **Kolmnurkade riba** (*GL_TRIANGLE_STRIP*) - Kolm järjestikku tippu moodustavad kolmnurga. Riba vaatab selles suunas, mis suunas on keeratud esimene kolmnurk.
- **Iseseisvad kolmnurgad** (*GL_TRIANGLES*) - Tipud 0, 1, 2 moodustavad kolmnurga. Tipud 3, 4, 5 moodustavad kolmnurga jne.
- **Kolmnurkade lehvik** (*GL_TRIANGLE_FAN*) - Esimene tipp on fikseeritud. Järgmised kahe tipu grupid moodustavad kolmnurga. Kolmnurgad on nt. (0, 1, 2), (0, 3, 4) jne.
- **Iseseisvad jooned** (*GL_LINES*) - Tipud 0,1 moodustavad joone. Tipud 2, 3 moodustavad joone jne.
- **Joonte riba** (*GL_LINE_STRIP*) - Järjestikus olevad tipud moodustavad jooned. N tipu moodustavad N - 1 joont.
- **Joonte tsükkel** (*GL_LINE_LOOP*) - Toimub täpselt samamoodi nagu joonte riba, kuid esimene ja viimane joon on ühendatud.

Andmestruktuur

Oletame, et meil on massiiv tipuandmetega, kus 3 järjestikust tippu moodustavad ühe kolmnurga:

```
{ {1, 0, 0}, {1, 1, 0}, {0, 0, 1}, {1, 1, 1} }
```

Kui kasutada neid andmeid, võetakse tipud ette vasakult-paremale. Tavaliselt kasutatakse samal koordinaatidel asuvaid punkte mitmete kolmnurkade poolt ja andmed sellisel kujul võivad korduda. Andmemahu vähendamiseks saame kasutusele võtta elementide massiivi, mis viitab tippudele, millest moodustub üks kolmnurk.

```
{ 1, 2, 3, 2, 2, 3}
```

Võttes kasutusele elementide massiivi liigub mööda *pipeline*'i järgmised andmed:

```
{ {1, 1, 0}, {0, 0, 1}, {1, 1, 1}, {0, 0, 1}, {0, 0, 0}, {1, 1, 1} }
```

Kui meil on sadu tuhandeid tippe, siis seda massiivi kasutades vähendame andmemahtu tunduvalt. Kui tavaliselt üks tipuatribuut on 32 baiti, siis elementide massiivis olev indeks on 2-4 baiti suur (Khronos Group, 2015). Tavaliselt tipuatribuut koosneb järgnevatest osadest:

```
{ {1, 1, 0}, {1, 0, 0 }, {0.5, 0.5} }
```

1. **Tipu koordinaadid** - Iga koordinaat on 32-bitine arv ehk 4 baiti. Kokku 12 baiti.
2. **Normaalvektor** (vektor, mis on tasapinnaga risti) - Samuti 12 baiti.
3. **Tekstuuri koordinaadid** - 8 baiti.

1.2.4. Tekstuurid ja tekstuuride kasutamine

Tekstuur on struktuur, kuhu saab mingit informatsiooni ladustada. Tihti salvestatakse sinna värve, millest moodustub pilt. Nimetus "tekstuur" on suhteliselt eksitav. Tekstuurile mõeldes tuleb kindlasti mõttesse mingi 2-dimensiooniline pilt, kuid tekstuur võib olla ka defineeritud kuubina, millel on kuus külge ja koosneb kuuest erinevast tekstuurist.

Tekstuur on lihtsalt n-dimensiooniline massiiv, mida on võimalik varjundajates kasutada.

Milleks kasutada?

Kuna tekstuur on lihtsalt ladustamiseks mõeldud struktuur, siis on tekstuuri võimalik salvestada peale RGB või RGBA informatsiooni ka muud kasulikku. Kõike, mida on võimalik arvudesse kodeerida, on võimalik nende ladustamiseks kasutada ka tekstuure. Graafikaprogrammeerimises on mitmeid tehnikaid, mis tekstuuri muul eesmärgil kasutavad: *environment mapping*_, specular mapping_, *normal mapping*, *height mapping*,*depth mapping*, *ambient occulsion* jne.

Ettevaatlik tuleb olla, mis arvutüüpi tekstuuris kasutada. Näiteks *UNSIGNED_BYTE*(vahemikus [0, 255]) korral võib osa informatsioonist kaduma minna. Kui värvide salvestamiseks kasutada *FLOAT* või *UNSIGNED_SHORT* tüüpi, siis on see lihtsalt ruumi raiskamine ja kannatada saab ka renderdamise kiirus.

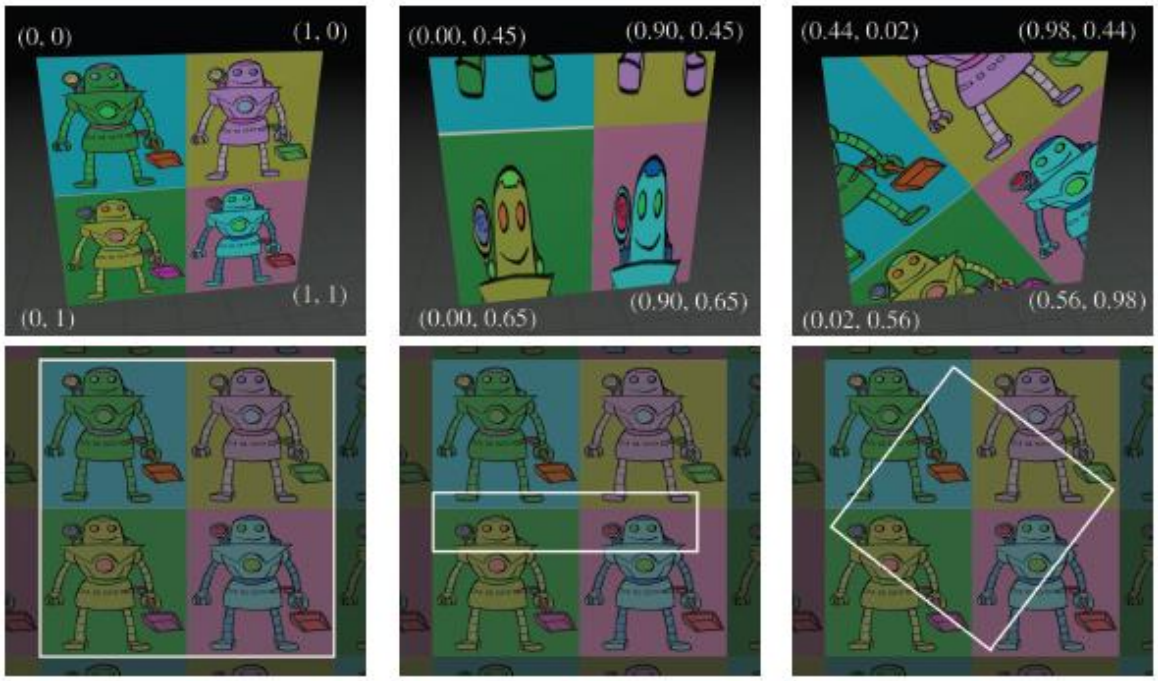
Tekstuuri kaardistamine

Objekt ruumis koosneb tippudest. Tipud moodustavad kolmnurga. Kolmnurgad moodustavad mingi tasapinna. Kui me kasutame ainult tipuatribuudi värvi, siis pikslid, mis kolmnurgas asuvad on interpoleeritud värvusega. Niiviisi ei ole meil võimalik luua objekti, mis meenutaks meile midagi reaalsest maailmast.

Seetõttu kasutame tekstuuri kaardistamist, kus pikslivarjundajas küsime tekstuuri koordinaate ja tekstuurist saadud värvuse võime värvipuhvrisse saata või tavalisemalt väärtust valguse kalkuleerimiseks kasutada.

Tekstuuri võib kasutada ka muu informatsiooni salvestamiseks kui ainult värv!

Tekstuuri koordinaadid on vahemikus [0, 1]. See aga ei tähenda, et me ei võiks vahemikust välja minna. Kui meil on koordinaadid vahemikust väljas ja kasutame funktsiooni *REPEAT* võime luua mustri objekti pinnal (vt joonist 1).



Joonist 1 Tekstuuri kaardistamine. (Dunn, Parberry 2011)

Kui olete lugenud varjundajate osa, siis peaks teadma, et tipuvarjundajast pikslivarjundajasse minevad väärtused interpoleeritakse. Kui meil on üks tipp tekstuuri koordinaatidega **(0, 0)** ja teine tipp tekstuuri koordinaatidega **(1, 1)**, siis kahe tipu keskel asuva piksli tekstuuri koordinaadid on **(0.5, 0.5)**.

Tekstuuri filtrid ja funktsioonid

Tekstuuri filtrid saame WebGL's määrata taoliselt:

```
gl.textureParameterf(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S. gl.REPEAT);
```

Filtrite abil on esiteks võimalik määrata, kuidas tekstuur käitub, kui ta asub mingil kaugusel.

```
GL.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.NEAREST);
GL.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST);
```

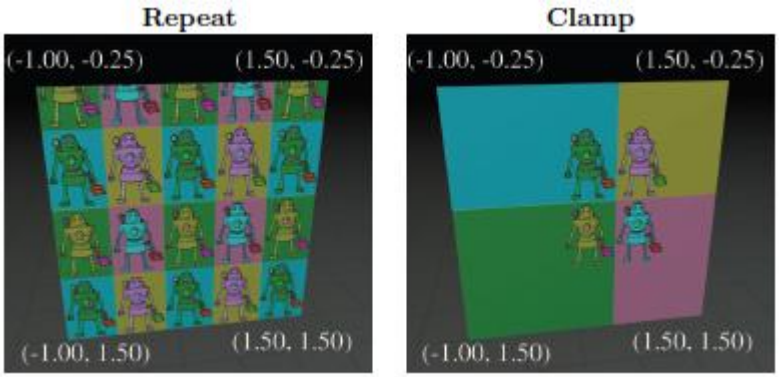
- **TEXTURE_MAG_FILTER** - Tekstuuri suurendamist nimetatakse *MAG - magnification*. Selle konstandiga on võimalik määrata, mis funktsiooni kasutada, kui tasapind, kus tekstuur asub on suurem, kui tegelikult tekstuur ise.
- **TEXTURE_MIN_FILTER** - Tekstuuri vähendamist nimetatakse *MIN - minification*. Selle konstandiga on võimalik määrata, mis funktsiooni kasutada, kui tasapind on väiksem, kui kasutatud tekstuur.

- **LINEAR** - Kui anname mingid tekstuuri koordinaadid, siis värv saadakse funktsioonist nii, et võetakse kaalutud keskmine kõikide ümbritsevate pikslite vahel (interpoleeritakse).
- **NEAREST** - Kõige kiirem ja kõige halvem kvaliteet. Tekstuurilt võetakse värv, mis on kõige lähemal meie poolt defineeritud koordinaatidele.

Samuti on võimalik määrata, mida teeb tekstuur, kui ta mingi objekti tasapinnale viia.

```
gl.texParameterf(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S, gl.REPEAT);
gl.texParameterf(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T, gl.REPEAT);
```

- **TEXTURE_WRAP_S** - *S* või *U* tähistavad tekstuuri x-koordinaati. Võimalik määrata, mis funktsiooni kasutada tekstuuri x-koordinaatide jaoks, kui nad on ületanud vahemiku [0, 1].
- **TEXTURE_WRAP_T** - *T* või *V* tähistavad tekstuuri y-koordinaati. Võimalik määrata, mis funktsiooni kasutada tekstuuri y-koordinaatide jaoks, kui nad on ületanud vahemiku [0, 1].
- **REPEAT** - Tekstuur kordub, kui on ületanud vahemiku [0, 1]. Mõistlik kasutada, kui meil on mingi korduva mustriga objekt.
- **CLAMP** - Viimast koordinaati, mis on veel piirides kasutatakse uuesti kõikide järgmiste väärtuste puhul, mis on ületanud vahemiku [0, 1].



Joonis 2 REPEAT ja CLAMP. (Dunn, Parberry 2011)

Mipmapping

Võimalik on kasutada tehnikat *mipmapping*, mis loob meie peamisest tekstuurist väiksemad koopiad. See tehnika parandab renderdamiskiirust ja vigu, mis võib juhtuda, kui objekt on kaugel ja tasapind, kus tekstuur asub, on väike. Kui meil on saadaval väiksemad versioonid tekstuurist, on meil võimalik kasutada järgnevaid filtreid:

- **NEAREST_MIPMAP_NEAREST** - Võtab lähima *mip* taseme ja sellelt lähima naaberpiksli.
- **NEAREST_MIPMAP_LINEAR** - Teostab *mip* tasemete vahel interpoleerimise ja võtab tulemusest lähima naaberpiksli.
- **LINEAR_MIPMAP_NEAREST** - Võtab lähima *mip* taseme ja sealt interpoleeritud värvuse.
- **LINEAR_MIPMAP_LINEAR** - Teostab *mip* tasemete vahel interpoleerimise ja teostab interpoleerimise ka värvi valikul. Kutsutakse *TRILINEAR FILTERING*.

Lisamaterjal

- https://msdn.microsoft.com/en-us/library/ie/dn302436(v=vs.85).aspx
- http://http.developer.nvidia.com/CgTutorial/cg_tutorial_chapter03.html
- http://www.arcsynthesis.org/gltut/Texturing/Texturing.html

1.2.5. WebGL funktsioonid

Peatükis seletame materjalis kasutatud WebGL funktsioonid lahti ja toome ka koodinäite, kuidas ja kus funktsiooni kasutada.

1.2.5.1. activeTexture

Aktiveerib defineeritud tekstuuri. Aktiveeritud tekstuuri on võimalik varjundajas kasutada.

WebGL versioonis 1.0 on võimalik spetsifikatsiooni kohaselt aktiveerida 32 erinevat tekstuuri. Loomulikult on see implementeerimise küsimus ja kõik graafikaprotsessorid ei pruugi 32. tekstuuri toetada. (Khronos Group, 2012)

```
void activeTexture(GLenum texture);
```

Näide

```
//Aktiveerime tekstuuri
gl.activeTexture(gl.TEXTURE0);

//Määrame tekstuuri, mida varjundajas kasutada
gl.uniform1i(u_SamplerLocation, 0);
```

Lisamaterjal

- https://msdn.microsoft.com/en-us/library/ie/dn302363(v=vs.85).aspx
- https://www.khronos.org/opengles/sdk/docs/man/xhtml/glActiveTexture.xml

1.2.5.2. attachShader

Lisab defineeritud varjundaja WebGLProgram objekti külge.

```
void attachShader(program, shader);
```

Näide

```
//Loome WebGLProgram objekti
var program = gl.createProgram();

//Seome tipuvarjundaja loodud programmiga
gl.attachShader(program, vertexShader);
```

```
//Seome pikslivarjundaja loodud programmiga
gl.attachShader(program, fragmentShader);
```

Lisamaterjal

- https://msdn.microsoft.com/en-us/library/ie/dn302364(v=vs.85).aspx
- https://www.khronos.org/opengles/sdk/docs/man/xhtml/glAttachShader.xml

1.2.5.3. bindBuffer

Seob defineeritud puhvri WebGL kontekstiga. Eelmine puhver, mis oli kontekstiga seotud, seotakse lahti. Puhver sisaldab arvväärtuseid näiteks kolmnurga tippe või värve.

```
void bindBuffer(target, buffer);
```

Näide

```
//Loome uue puhvri kuhu andmeid salvestada
var myBuffer = gl.createBuffer()

var myVerticesData = [
   0.0,  1.0, 0.0
  -1.0, -1.0, 0.0,
   1.0, -1.0, 0.0
];

//Seome kontekstiga puhvri, mis sisaldab tippe, mida ekraanile joonistada
gl.bindBuffer(gl.ARRAY_BUFFER. myBuffer);

//Täidame eelpool seotud puhvri andmetega, mis tulevad üleval defineeritud massiivist
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(myVerticesData), gl.STATIC_DRAW);

...

//Renderdame tipud. Viimane parameeter 3 näitab mitu kolmnurka tahame renderdada.
gl.drawArrays(gl.TRIANGLES, 0, 3)
```

Lisamaterjal

- https://msdn.microsoft.com/en-us/library/ie/dn302365(v=vs.85).aspx
- https://www.khronos.org/opengles/sdk/docs/man/xhtml/glBindBuffer.xml

1.2.5.4. bindTexture

Seob defineeritud tekstuuri WebGL kontekstiga. Sidudes tekstuuri kontekstiga on meil võimalik muuta tekstuuri parameetreid ehk kuidas tekstuur käituma peaks. Kui me tahame mingit pilti sellesse tekstuuri laadida peame selle kindlasti enne siduma.

```
void bindTexture(target texture object);
```

Näide

```
//Loome tekstuuri, mis mingisugust pilti hoidma hakkab
var texture = gl.createTexture();

//Seome tekstuuri kontekstiga
gl.bindTexture(gl.TEXTURE_2D, texture);

//Laeme pildi tekstuuri.
gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGB, gl.RGB, gl.UNSIGNED_BYTE, imageObject);

//Seadistame pildi parameetrid
gl.texParameterf(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S, gl.REPEAT);
gl.texParameterf(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T, gl.REPEAT);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.NEAREST);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST);
```

Lisamaterjal

- https://www.khronos.org/opengles/sdk/docs/man/xhtml/glBindTexture.xml
- https://msdn.microsoft.com/en-us/library/ie/dn302368(v=vs.85).aspx
- https://msdn.microsoft.com/en-us/library/ie/dn302436(v=vs.85).aspx

1.2.5.5. bufferData

Loob mälus puhvri ja saadab sinna andmed, parameetriks defineeritud massiivist. Kui massiivi ei ole defineeritud on puhvris olevad andmed 0.

```
void bufferData(target, size or data, usage);
```

Viimane 3 parameeter defineerib, kuidas antud andmeid hakatakse kasutama (Microsoft):

- *STATIC_DRAW* - Andmed, ladustatakse ühe korra ja kasutatakse mitmeid kordi renderdamiseks.
- *DYNAMIC_DRAW* - Andmeid ladustatakse mitmeid kordi ja kasutatakse mitmeid kordi renderdamiseks.
- *STREAM_DRAW* - Andmeid ladustatakse ühe korra ja kasutatakse aega-ajalt.

Näide

```
//Loome uue puhvri kuhu andmeid salvestada
var myBuffer = gl.createBuffer()

var myVerticesData = [
   0.0,  1.0, 0.0
  -1.0, -1.0, 0.0,
   1.0, -1.0, 0.0
];

//Seome kontekstiga puhvri, mis sisaldab tippe, mida ekraanile joonistada
gl.bindBuffer(gl.ARRAY_BUFFER. myBuffer);

//Täidame eelpool seotud puhvri andmetega, mis tulevad üleval defineeritud massiivist
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(myVerticesData), gl.STATIC_DRAW);

...

//Renderdame antud tipud. Viimane parameeter 3 näitab mitu tipu on massiivis.
gl.drawArrays(gl.TRIANGLES, 0, 3)
```

Lisamaterjal

- https://www.khronos.org/opengles/sdk/docs/man/xhtml/glBufferData.xml
- https://msdn.microsoft.com/en-us/library/ie/dn302373(v=vs.85).aspx

1.2.5.6. clearColor

Võimaldab määrata puhvri puhastusvärvus. Meetodi abil on võimalik defineerida värv, millega iga kaadri lõpus meetodiga **clear()** puhver täidetakse.

```
void gl.clearColor(0.0, 0.0, 0.0, 1.0);
```

Näide

```
//Määran puhastusvärvuseks musta
gl.clearColor(0.0, 0.0, 0.0, 1.0);

//Puhastan sügavus- ja värvipuhvri
gl.clear(gl.DEPTH_BUFFER_BIT | gl.COLOR_BUFFER_BIT);

...
//Renderdame antud tipud. Viimane parameeter 3 näitab mitu kolmnurka tahame renderdada.
gl.drawArrays(gl.TRIANGLES, 0, 3)
```

Lisamaterjal

- https://www.khronos.org/opengles/sdk/docs/man/xhtml/glClearColor.xml
- https://msdn.microsoft.com/en-us/library/ie/dn302377(v=vs.85).aspx

1.2.5.7. clear

Funktsiooni väljakutsumisel puhastatakse puhver

```
void gl.clear(gl.DEPTH_BUFFER_BIT | gl.COLOR_BUFFER_BIT))
```

Antud meetod kutsutakse välja tavaliselt pärast meetodit **clearColor()**. Meetodi abil on võimalik defineerida, mis puhvrid kaadripuhvrist puhastatakse.

Kaadripuhver ise sisaldab endas mitut puhvrit, mis säilitavad erinevaid andmeid: värv, sügavus jne.

- *DEPTH_BUFFER_BIT* - puhastab sügavuspuhvrist.
- *COLOR_BUFFER_BIT* - puhastab värvipuhvri.

Näide

```
//Määran puhastusvärvuseks musta
gl.clearColor(0.0, 0.0, 0.0, 1.0);

//Puhastan sügavus- ja värvipuhvri
gl.clear(gl.DEPTH_BUFFER_BIT | gl.COLOR_BUFFER_BIT);

...
//Renderdame antud tipud. Viimane parameeter 3 näitab mitu kolmnurka tahame renderdada.
gl.drawArrays(gl.TRIANGLES, 0, 3)
```

Lisamaterjal

- https://www.khronos.org/opengles/sdk/docs/man/xhtml/glClear.xml
- https://msdn.microsoft.com/en-us/library/ie/dn302376(v=vs.85).aspx

1.2.5.8. colorMask

Funktsiooni abil saab määrata, mis värvid kirjutatakse kaadripuhvrisse. Algselt on kõik väärtused tõesed.

```
void colorMask(boolean red, boolean green, boolean blue, boolean alpha);
```

Lisamaterjal

- https://msdn.microsoft.com/en-us/library/ie/dn434080(v=vs.85).aspx
- https://www.khronos.org/opengles/sdk/docs/man/xhtml/glColorMask.xml

1.2.5.9. compileShader

Kompileerib *string*'i kujul oleva varjundaja binaarsesse vormi, mida saab WebGLProgram objekt kasutada.

```
void compileShader(shader);
```

Näide

```
//String, mis sisaldab endas koodi, mis käivitatakse tipuvarjundajas
var vertexShaderString = "
     precision mediump;
     attribute vec3 a_Position;

     void main() {
         gl_Position = vec4(a_Position, 1.0);
     }
   ";

//Loon tipuvarjundaja
var vertexShader = gl.createShader(gl.VERTEX_SHADER);

//Annan varjundajale andmed
gl.shaderSource(shader, vertexShaderString);

//Kompileerin varjundaja binaarsesse vormi
gl.compileShader(shader);

...

//Edasi seotakse binaarsesse vormi viidud varjundaja WebGLProgram objektiga
```

Vaata ka linkProgram näide

Lisamaterjal

- https://msdn.microsoft.com/en-us/library/ie/dn302379(v=vs.85).aspx
- https://www.khronos.org/opengles/sdk/docs/man/xhtml/glCompileShader.xml

1.2.5.10. createShader

Loob WebGLShader objekti, mis on tühi. Pärast antud meetodi kasutamist tuleb välja kutsuda meetodid, mis täidavad ja kompileerivad antud varjundaja lähetkoodi.

```
WebGLShader createShader(type);
```

Näide

```
//String, mis sisaldab endas koodi, mis käivitatakse tipuvarjundajas
var vertexShaderString = "
     precision mediump;
     attribute vec3 a_Position;

     void main() {
         gl_Position = vec4(a_Position, 1.0);
     }
   ";

//Loome WebGLShader objekti
var shader = gl.createShader(gl.VERTEX_SHADER);

//Määrame antud varjundaja lähtekoodi
gl.shaderSource(shader, vertexShaderString);

//Kompileerime lähtekoodi binaarsesse vormi
gl.compileShader(shader);

...

//Varjundaja lisatakse WebGLProgram objekti, mida saab renderdamise järjekorras kasutada.
```

Vaata ka linkProgram näide

Lisamaterjal

- https://msdn.microsoft.com/en-us/library/ie/dn302386(v=vs.85).aspx
- https://www.khronos.org/opengles/sdk/docs/man/xhtml/glShaderSource.xml

1.2.5.11. createProgram

Loob tühja WebGLProgram objekti, millega seotakse tipu ja lagivarjundaja. Renderdamist ei toimu, kui pole ühtegi programmi kontekstiga seotud.

```
WebGLProgram createProgram();
```

Näide

```
var program = gl.createProgram();

...

//Loome ja kompileerime varjundajad

...

//Seome varjundajad antud programmiga
gl.attachShader(program, vertexShader);
gl.attachShader(program, fragmentShader;

//Seome programmi kontekstiga, et seda renderdamisel kasutada
gl.useProgram(program);
```

Lisamaterjal

- https://msdn.microsoft.com/en-us/library/ie/dn302384(v=vs.85).aspx
- https://www.khronos.org/opengles/sdk/docs/man/xhtml/glCreateProgram.xml

1.2.5.12. enableVertexAttribArray

Lülitab sisse tipuatribuudi(positsioon, normaalvektor, värv) massiivi tipupuhvris (vt Renderdamise järjekord).

```
void enableVertexAttribArray(index);
```

Näide

```
//Loome WebGLProgram objekti, kuhu saab kompileeritud varjundajad külge pookida
var shaderProgram = gl.createProgram();

...

//Määrame programmi, mida hetkel kasutada renderdamiseks
gl.useProgram(shaderProgram);

//Saame indeksi, mis näitab kus asub meie programmis kasutatavas tipuvarjundajas
//olev tipuatribuut nimega a_VertexPosition.
```

```
var a_VertexPositionLocation = gl.getAttribLocation(shaderProgram, "a_VertexPosition")

...

//Massiiv, mis sisaldab tippe, mida tahame renderdada. 3 tipu ehk 1 kolmnurka (kui kasutame
gl.TRIANGLES)
var myVerticesData = [
   0.0,  1.0, 0.0
  -1.0, -1.0, 0.0,
   1.0, -1.0, 0.0
];

//Loome puhvri, kuhu salvestada tipuandmed
var vertexBuffer = gl.createBuffer();

//Seome antud puhvri kontekstiga, et sellega operatsioone teha
gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);

//Saadame andmed puhvrisse
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(myVerticesData), gl.STATIC_DRAW);

//Määrame, kus antud tipuatribuut meie poolt kontekstiga seotud puhvris asub.
gl.vertexAttribPointer(a_VertexPositionLocation, 3, gl.FLOAT, false, 0, 0)

//Aktiveerime eelpool küsitud atribuudi
gl.enableVertexAttribArray(a_VertexPositionLocation);

//Renderdame tipud
gl.drawArrays(gl.TRIANGLES, 0, 3)
```

Lisamaterjal

- https://msdn.microsoft.com/en-us/library/ie/dn302400(v=vs.85).aspx
- https://www.khronos.org/opengles/sdk/docs/man/xhtml/glEnableVertexAttribArray.xml

1.2.5.13. enable

Meetodi abil on võimalik funktsionaalsust sisse lülitada

```
void gl.enable(capability);
```

Kuna WebGL'i võib vaadelda kui lõplikku automaati on antud meetodi abil võimalik sisse või välja lülitada erinevat funktsionaalsust. Mõningad funktsionaalsuse on järgmised:

- *GL_BLEND* - Pikslivarjundajas arvutatud värvus segatakse kaadripuhvri värvipuhvris oleva väärtusega.
- *GL_CULL_FACE* - Olenevalt, mis järjekorras on hulktahukad (*polygon*) keritud, heidetakse nad kaadrist välja.
- *GL_DEPTH_TEST* - Kontrollitakse fragmendi/piksli sügavusväärtus, et otsustada, kas kirjutada antud väärtus kaadripuhvrisse või mitte.

Näide

```
...

//Lülitab sisse pikslite sügavuse võrdlemise, mistõttu sügavusväärtuseid kirjutatakse kaadripuhvri sügavuspuhvrisse.
gl.enable(gl.DEPTH_TEST);

//Määrab sügavustesti funktsiooniks vähima ehk uus sügavusväärtus kirjutatakse juhul, kui seal olev väärtus on suurem.
gl.depthFunc(gl.LESS);

...

//Toimub renderdamine, kus renderdamise järjekorras täidetakse kaadripuhvri värvi- ja sügavuspuhvrid.
```

Lisamaterjal

- https://msdn.microsoft.com/en-us/library/ie/dn302399(v=vs.85).aspx
- https://www.khronos.org/opengles/sdk/docs/man/xhtml/glEnable.xml
- https://msdn.microsoft.com/en-us/library/ie/dn302390(v=vs.85).aspx

1.2.5.14. disable

Meetodiga on võimalik funktsionaalsust välja lülitada. Töötab vastupidiselt meetodile *enable*. Vaata antud meetodit enable meetod

```
void gl.disable(capability);
```

Lisamaterjal

- https://www.khronos.org/opengles/sdk/docs/man/xhtml/glDisable.xml
- https://msdn.microsoft.com/en-us/library/ie/dn302393(v=vs.85).aspx

1.2.5.15. drawArrays

Meetod, mis renderdab eelpool defineeritud ja seotud WebGLProgram objekti ja tipuandmete põhjal geomeetrilised primitiivid(kolmnurgad, jooned või hoopiski kolmnurkade lehvik...).

```
void drawArrays(mode/primitive type, first element in array, number of primitives to render);
```

Näide

```
//Loome uue puhvri kuhu andmeid salvestada
var myBuffer = gl.createBuffer()

var myVerticesData = [
   0.0,  1.0, 0.0
  -1.0, -1.0, 0.0,
   1.0, -1.0, 0.0
];

//Seome kontekstiga puhvri, mis sisaldab tippe, mida ekraanile joonistada
gl.bindBuffer(gl.ARRAY_BUFFER. myBuffer);

//Täidame eelpool seotud puhvri andmetega, mis tulevad üleval defineeritud massiivist
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(myVerticesData), gl.STATIC_DRAW);

...

//Renderdame antud tipud. Viimane parameeter 3 näitab mitu tipu on massiivis.
gl.drawArrays(gl.TRIANGLES, 0, 3)
```

Lisamaterjal

- https://www.khronos.org/opengles/sdk/docs/man/xhtml/glDrawArrays.xml
- https://msdn.microsoft.com/en-us/library/ie/dn302395(v=vs.85).aspx

1.2.5.16. drawElements

Meetodi abil renderdatakse tipupuhvris olevad tipud kui geomeetrilised primitiivid(kolmnurgad, jooned või hoopiski kolmnurkade lehvik...). Renderdamiseks kasutatakse tippe, mis on defineeritud elementide massiivi poolt, mis sisaldab tippude asukoha indekseid tipupuhvris.

```
void drawElements(mode/primitive type, number of indexes/elements in array, type, offset);
```

Näide

```
//Loome uue puhvri kuhu tipuandmeid salvestada
var myVertexBuffer = gl.createBuffer()

//Tippude andmed. Sisaldavad ainult positsiooni
var myVerticesData = [
   -1.0, -1.0,  1.0,
    1.0, -1.0,  1.0,
    1.0,  1.0,  1.0,
   -1.0,  1.0,  1.0,
];

//Seome kontekstiga puhvri, mis sisaldab tippe, mida ekraanile joonistada
gl.bindBuffer(gl.ARRAY_BUFFER. myVertexBuffer);

//Täidame eelpool seotud puhvri andmetega, mis tulevad üleval defineeritud massiivist
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(myVerticesData), gl.STATIC_DRAW);

//Loome puhvri, kuhu tippude indekseid salvestada
var myIndexBuffer = gl.createBuffer();

//Tippude indeksid
var myIndicesData = [
   0,  1,  2,
   0,  2,  3
];

//Määrame elementide(indeksite) arvu antud massiivis, mida tuleb teada allpool olevas meetodis drawElements
myIndexBuffer.numItems = 6;

//Seome kontekstiga puhvri, mis sisaldab tippude indekseid
```

```
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, myIndexBuffer);

//Täidame eelpool seotud puhvri andmetega, mis tulevad üleval defineeritud indeksite massiivist
gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new Uint16Array(myIndicesData), gl.STATIC_DRAW);

...

//Seome kontekstiga tippude puhvri ja määrame veretxAttribPointer abil tipuatribuudi
//(positsiooni) asukoha antud massiivis

...

//Renderdame elementide massiivi abil defineeritud tipud, mis tulevad tipupuhvrist
gl.drawElements(gl.TRIANGLES, myIndexBuffer.numItems, gl.UNSIGNED_SHORT, 0);
```

Lisamaterjal

- https://msdn.microsoft.com/en-us/library/ie/dn302396(v=vs.85).aspx
- https://www.khronos.org/opengles/sdk/docs/man/xhtml/glDrawElements.xml

1.2.5.17. getContext

Antud meetodi abil luuakse WebGL kontekst.

```
gl = canvas.getContext("webgl") || canvas.getContext("webgl-experimental");
```

HTML5's on võimalik **canvas.getContext(contextType, contextAttributes)** meetodi abil võimalik luua objekt, mille abil antud elementi joonistada. Kasutades parameetriks**"webgl"** või **webgl-experimental** luuakse **WebGLRenderingContext**. Antud kontekstile on võimalik ligipääseda ainult veebilehitsejaga, milles on implementeeritud **WebGL versioon 1 (OpenGL ES 2.0)**. (Mozilla Developer Network)

Andes **getContext** meetodisse parameetriks *"webgl2"* on võimalik luua ka versioon 2. kontekst, kuid hetkel on see toetatud vaid Mozilla Firefoxis.

Näide

HTML

```
<!DOCTYPE html>
<html>
```

```
<head>
</head>
<body>
  <canvas id="webGLCanvas" width="800" height="600"></canvas>
</body>
</html>
```

Javascript

```
//Saame Canvas elemendi viite, kuhu hakkame renderdama
var canvas = document.getElementById("webGLCanvas");

//Loome WebGL konteksti
gl = canvas.getContext("webgl") || canvas.getContext("experimental-webgl");
```

Lisamaterjal

- https://developer.mozilla.org/en-US/docs/Web/WebGL/Getting_started_with_WebGL

1.2.5.18. getAttribLocation

Meetodi abil on võimalik WebGLProgram objektist saada tipuatribuudi asukoha/indeksi.

```
Number getAttribLocation(shaderProgram, name);
```

Näide

```
//Tipuvarjundaja
var vertexShader = "
  attribute vec3 a_Position;

  void main() {
    gl_Position = vec4(a_Position, 1.0);
  }
";

//Massiiv, mis sisaldab tippe, mida tahame renderdada. 3 tipu ehk 1 kolmnurka (kui kasutame
gl.TRIANGLES)
var myVerticesData = [
   0.0,  1.0, 0.0
  -1.0, -1.0, 0.0,
   1.0, -1.0, 0.0
];
```

```
//Loome WebGLProgram objekti, mida renderdamisel kasutame
var shaderProgram = gl.createProgram();

...

Loome tipupuhvri ja täidame selle andmetega.
Kompileerime ja lisame varjundajad eelpool defineeritud programmi.

...

//Saame tipuvarjundajas defineeritud atribuudi indeksi/asukoha.
var a_ValueLocation = gl.getAttribLocation(shaderProgram, "u_Position");

//Määrame, kus antud tipuatribuut meie poolt kontekstiga seotud puhvris asub.
gl.vertexAttribPointer(a_VertexPositionLocation, 3, gl.FLOAT, false, 0, 0)

//Aktiveerime eelpool küsitud atribuudi, et seda saaks renderdamise järjekorras kasutada
gl.enableVertexAttribArray(a_VertexPositionLocation);

//Renderdame kaadripuhvrisse. Tipuvarjundaja muutujasse a_Position on võimalik nüüd antud
andmed kätte saada.
gl.drawArrays(gl.TRIANGLES, 0, 3);
```

Lisamaterjal

- https://msdn.microsoft.com/en-us/library/ie/dn302408(v=vs.85).aspx
- https://www.khronos.org/opengles/sdk/docs/man/xhtml/glGetAttribLocation.xml

1.2.5.19. getUniformLocation

Antud meetod tagastab WebGLProgram objekti varjundajas oleva ühtse muutuja (*uniform*) asukoha/indeksi.

```
WebGLUniformLocation getUniformLocation(shaderProgram, name);
```

Näide

```
//Tipuvarjundaja
var vertexShader = "
  uniform float u_Value;
```

```
  void main() {
    gl_Position = vec4(1.0, 1.0, 1.0, 1.0);
  }
";

//Loome WebGLProgram objekti, mida renderdamisel kasutame
var shaderProgram = gl.createProgram();

...

Kompileerime ja lisame varjundajad eelpool defineeritud programmi

...

//Saame tipuvarjundajas defineeritud ühtse muutuja asukoha.
var u_ValueLocation = gl.getUniformLocation(shaderProgram, "u_Value");

...

//Määrame mingi juhusliku väärtuse
var value = 9000;

//Määrame hetkel aktiveeritud programmi ühtsesse muutujasse u_Value väärtuse 9000
gl.uniform1f(u_ValueLocation, value);

...

//Renderdame mingid andmed. Varjundajates on nüüd ühtses muutujas u_Value väärtus 9000
gl.drawArrays(gl.TRIANGLES, 0, 3);
```

Lisamaterjal

- https://msdn.microsoft.com/en-us/library/ie/dn302424(v=vs.85).aspx
- https://www.khronos.org/opengles/sdk/docs/man/xhtml/glGetUniformLocation.xml

1.2.5.20. linkProgram

Seob lisatud varjundajad (lisatud *attachShader* meetodi abil) WebGLProgram objektiga, et edasi saaks neid kasutada renderdamiseks GPU poolt.

```
void linkProgram(shaderProgram);
```

Näide

```
//String, mis sisaldab endas koodi, mis käivitatakse tipuvarjundajas
var vertexShaderString = "
     precision mediump;
     attribute vec3 a_Position;

     void main() {
         gl_Position = vec4(a_Position, 1.0);
     }
   ";

var fragmentShaderString = "
    precision mediump;

    void main(){
       gl_FragColor = vec(0.0, 1.0, 0.0, 1.0);
    }
  ";

//Loome varjundajad
var fragmentShader = gl.createShader(gl.VERTEX_SHADER);
var vertexShader = gl.createShader(gl.FRAGMENT_SHADER);

//Määrame varjundajate lähtekoodi
gl.shaderSource(fragmentShader, fragmentShaderString);
gl.shaderSource(vertexShader, vertexShaderString);

/Kompileerime varjundajad
gl.compile(fragmentShader);
gl.compile(vertexShader);

//Loome WebGLProgram objekti
var program = gl.createProgram();

//Seome varjundajad loodud programmiga
gl.attachShader(program, vertexShader);
gl.attachShader(program, fragmentShader);

gl.linkProgram(program);

...
```

```
//Määrame renderdamiseks kasutatava programmi
gl.useProgram(program);

Kasutame renderdamiseks antud programmi
```

Lisamaterjal

- https://www.khronos.org/opengles/sdk/docs/man/xhtml/glLinkProgram.xml
- https://msdn.microsoft.com/en-us/library/ie/dn302428(v=vs.85).aspx

1.2.5.21. shaderSource

Määrab varjundajas kasutatava lähtekoodi

```
void shaderSource(shader, sourceCode);
```

Näide

```
//Loome tipuvarjundaja
var vertexShader = gl.createShader(gl.vertexShader);

//Määrame varjundaja lähtekoodi
gl.shaderSource(vertexShader, vertexShaderSource);

//Kompileerime varjundaja
gl.compileShader(vertexShader);

...

//Lisame varjundaja programmi ja nüüd saame seda kasutada renderdamise järjekorras
```

Lisamaterjal

- https://msdn.microsoft.com/en-us/library/ie/dn302434(v=vs.85).aspx
- https://www.khronos.org/opengles/sdk/docs/man/xhtml/glShaderSource.xml

1.2.5.22. vertexAttribPointer

Meetod määrab andmete formaadi ja andmete asukoha massiivis.

Näide

```
//Tipuvarjundaja
var vertexShader = "
  attribute vec3 a_Position;

  void main() {
    gl_Position = vec4(a_Position, 1.0);
  }
";

//Massiiv, mis sisaldab tippe, mida tahame renderdada. 3 tipu ehk 1 kolmnurka (kui kasutame
gl.TRIANGLES)
var myVerticesData = [
   0.0,  1.0, 0.0
  -1.0, -1.0, 0.0,
   1.0, -1.0, 0.0
];

//Loome WebGLProgram objekti, mida renderdamisel kasutame
var shaderProgram = gl.createProgram();

...

Loome tipupuhvri ja täidame selle andmetega.
Kompileerime ja lisame varjundajad eelpool defineeritud programmi.

...

//Saame tipuvarjundajas defineeritud atribuudi indeksi/asukoha.
var a_ValueLocation = gl.getAttribLocation(shaderProgram, "u_Position");

//Määrame, kus antud tipuatribuut meie poolt kontekstiga seotud puhvris asub.
gl.vertexAttribPointer(a_VertexPositionLocation, 3, gl.FLOAT, false, 0, 0)

//Aktiveerime eelpool küsitud atribuudi, et seda saaks renderdamise järjekorras kasutada
gl.enableVertexAttribArray(a_VertexPositionLocation);
```

```
//Renderdame kaadripuhvrisse. Tipuvarjundaja muutujasse a_Position on võimalik nüüd antud andmed kätte saada.
gl.drawArrays(gl.TRIANGLES, 0, 3);
```

Lisamaterjal

- https://msdn.microsoft.com/en-us/library/ie/dn302460(v=vs.85).aspx
- https://www.khronos.org/opengles/sdk/docs/man/xhtml/glVertexAttribPointer.xml

1.2.5.23. viewport

Antud meetodi abil spetsifitseeritakse vaateakna suuruse, millest sõltub ka kaadripuhvri suurus. Antud meetod tuleb välja kutsuda enne mõnda joonistamiskutsumust (*drawArrays*,*drawElements*).

```
void viewport(int x, int y, long width, long height);
```

Lisamaterjal

- https://msdn.microsoft.com/en-us/library/ie/dn302461(v=vs.85).aspx
- https://www.khronos.org/opengles/sdk/docs/man/xhtml/glViewport.xml

1.2.5.24. texImage2D

Meetod määrab/laeb aktiveeritud tekstuuri (*bindTexture*) pildi/kujutise.

```
void texImage2D(target, level, internalformat, width, height, border, format, type, pixels);
void texImage2D(target, level internalformat, format, type, pixels/image/canvas/video);

void texImage2D(gl.TEXTURE_2D, 0, gl.RGB, gl.RGB, gl.UNSIGNED_BYTE, image);
```

Tekstuuri peatükis seletame pikemalt

Näide

```
//Loome tekstuuri
var texture = gl.createTexture();

//Loome pildi
var image = new Image();
```

```
//Määrame kutsumuse, kui pilt on serverist laetud
image.onload = function() {
  handleLoadedPicture(texture, image);
};

//Määrame pildi relatiivse asukoha serveris
image.src= "./img/picture.png";

//Määrame tekstuurile pildi
function handleLoadedPicture(texture, image) {

  //Aktiveerime tekstuuri sihtmärki TEXTURE_2D
  gl.bindTexture(gl.TEXTURE_2D, texture);

  //Laeme pildi tekstuuri
  gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGB, gl.RGB, gl.UNSIGNED_BYTE, image);

  //Seadistame tekstuuri parameetrid
  gl.texParameterf(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S, gl.REPEAT);
  gl.texParameterf(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T, gl.REPEAT);
  gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.NEAREST);
  gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST);

  //Seome tekstuuri kontekstist lahti
  gl.bindTexture(gl.TEXTURE_2D, null);
};
```

Lisamaterjal

- https://www.khronos.org/opengles/sdk/docs/man/xhtml/glTexImage2D.xml
- https://msdn.microsoft.com/en-us/library/ie/dn302435(v=vs.85).aspx

1.2.5.25. texParameter

Määrab tekstuuri parameetrid ehk kuidas antud tekstuur käitub.

Loe Tekstuurid ja tekstuuride kasutamine.

```
//Täisarve
void texParameteri(target, parameter name, parameter)
```

```
//Ujukoma arvud
void texParameteri(target, parameter name, parameter)
```

Näide

```
//Loome tekstuuri
var texture = gl.createTexture();

//Loome pildi
var image = new Image();

//Määrame kutsumuse, kui pilt on serverist laetud
image.onload = function() {
  handleLoadedPicture(texture, image);
};

//Määrame pildi relatiivse asukoha serveris
image.src= "./img/picture.png";

//Määrame tekstuurile pildi
function handleLoadedPicture(texture, image) {

  //Aktiveerime tekstuuri sihtmärki TEXTURE_2D
  gl.bindTexture(gl.TEXTURE_2D, texture);

  //Laeme pildi tekstuuri
  gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGB, gl.RGB, gl.UNSIGNED_BYTE, image);

  //Seadistame tekstuuri parameetrid
  gl.texParameterf(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S, gl.REPEAT);
  gl.texParameterf(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T, gl.REPEAT);
  gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.NEAREST);
  gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST);

  //Seome tekstuuri kontekstist lahti
  gl.bindTexture(gl.TEXTURE_2D, null);
};
```

Lisamaterjal

- https://www.khronos.org/opengles/sdk/docs/man/xhtml/glTexParameter.xml
- https://msdn.microsoft.com/en-us/library/ie/dn302436(v=vs.85).aspx
- https://msdn.microsoft.com/en-us/library/ie/dn302437(v=vs.85).aspx

1.2.5.26. uniform

Antud meetodite abil on võimalik määrata ühtse muutuja väärtus WebGLProgram objekti varjundajates. Meetodeid on mitmeid, mille lõpus olevad sümbolid **1f**, **1fv**, **1i**, **1iv**...**4f**, **4fv**,**4i**, **4iv** tähistavad ühtsete muutujate andmete tüüpe. Sümbolid tähendavad järgmist:

- numbrid **1 - 4**: mitu arvu kaas anda
- **f või i** - kas tegu on täisarvu või ujukoma arvuga
- **v** - kas tegu on vektoriga

Maatriksite jaoks on eraldi meetodid:

- **uniformMatrix2fv(location, 2x2 matrix)**
- **uniformMatrix3fv(location, 3x3 matrix)**
- **uniformMatrix4fv(location, 4x4 matrix)**

Mõningad meetodid on järgmised:

- void uniform1f(location, number);
- void uniform1fv(location, vector(1.0));
- uniform3f(location, float x, float y, float z);
- uniform4fv(location, vector(1.0, 1.0, 1.0, 1.0));
- uniformMatrix2Fv(location, boolean transpose, 2x2 matrix);
- uniformMatrix4Fv(location, boolean transpose, 4x4 matrix);

Näide

Määrame ühe ujukoma arvu.

```
//Tipuvarjundaja
var vertexShader = "
  uniform float u_Value;

  void main() {
    gl_Position = vec4(1.0, 1.0, 1.0, 1.0);
  }
";

//Loome WebGLProgram objekti, mida renderdamisel kasutame
```

```
var shaderProgram = gl.createProgram();

...

Kompileerime ja lisame varjundajad eelpool defineeritud programmi

...

//Saame tipuvarjundajas defineeritud ühtse muutuja asukoha.
var u_ValueLocation = gl.getUniformLocation(shaderProgram, "u_Value");

...

//Määrame mingi juhusliku väärtuse
var value = 9000;

//Määrame hetkel aktiveeritud programmi ühtsesse muutujasse u_Value väärtuse 9000
gl.uniform1f(u_ValueLocation, value);

...

//Renderdame mingid andmed. Varjundajates on nüüd ühtses muutujas u_Value väärtus 9000
gl.drawArrays(gl.TRIANGLES, 0, 3);
```

Lisamaterjal

- https://www.khronos.org/opengles/sdk/docs/man/xhtml/glUniform.xml

1.2.5.27. useProgram

Meetodi abil määrame WebGLProgram objekti, mida renderdamisel kasutada.

```
void useProgram(shaderProgram);
```

Näide

```
var shaderProgram = gl.createProgram();

...

Kompileerime ja seome varjundajad loodud programmiga.
```

```
...

Puhvrite loomine ja neile andmete andmine.
...

//Määrame programmi, mida renderdamisel kasutada
gl.useProgram(shaderProgram);

//Renderdame mingit tipud. Renderdamise järjekorras kasutame eelpool defineeritud
//programmi ja temaga seotud varjundajaid
gl.drawArrays(gl.TRIANGLES, 0, 3);
```

Vaata ka linkProgram näide

Lisamaterjal

- https://www.khronos.org/opengles/sdk/docs/man/xhtml/glUseProgram.xml
- https://msdn.microsoft.com/en-us/library/ie/dn302459(v=vs.85).aspx

2. Praktika

Praktika osas saame rakendada teooria osas saadud teadmisi. Kindlasti tasub enne tööle asumist lugeda õppetükis välja toodud teooriaosa.

Praktika osa läbimiseks on vaja alla laadida õppematerjaliga tulev lähtekood. Lähtekood asub aadressil:

- https://github.com/ranerp/webgl-sissejuhatus

2.1. Ettevalmistus

Veenduge, et teie veebilehitseja toetab WebGL'i - http://webglreport.com/

Selles materjalis on võimalik lähtekoodiga tegelemiseks kasutada Apache serverit või NodeJS serverit.

Näidete jooksutamiseks Apache serveris kopeerida kausta **builds** sisu oma **www** kataloogi. **Kui otsustate Apache kasuks võite järgmise osa juurde liikuda.**

Kasutame Node.js'i ja tema järgnevaid mooduleid:

- **Node.js** http://nodejs.org/ - Node.js abil on võimalik luua sündmuspõhist, kerget ja efektiivset serverirakendust.
- **Browserify** http://browserify.org/ - Võimaldab meil kasutada **require()** meetodit, millega erinevad moodulid sisse laadida.
- **Watchify** https://github.com/substack/watchify - Lähtekoodi muudatuste korral kompileerib koodi uuesti.

Node.js installeerimine Windowsis

1. Laadige alla ja installeerige Node.js.
2. Vaja on seadistada teekond, kus Node.js täideviiv programm asub:
    - *Control Panel\System and Security\System\Advanced System Settings\Environment Variables*
    - Leida sealt *System variables\Path*
    - Sellesse muutujasse lisada Node.js installeerimise asukoht (nt *C:\Program Files (x86)\nodejs\;*)

Node.js installeerimine Linuxis

Uurige aadressilt https://github.com/joyent/node/wiki/installing-node.js-via-package-manager

Command Prompt vajalikke käskude teadmine

- *cd* - Failisüsteemis liikumine. Näited:

```
F: - vahetame ketast
cd.. - liigume ühe kausta võrra tagasi
cd "Program Files (x86)/nodejs" - liigume nodejs kausta
```

- *dir* - Kaustasisu vaatamine

Node.js moodulite installeerimine

Kui on alla tõmmatud lähtekood ja installeeritud Node.js tuleb liikuda kausta, kus lähtekood asub.

cd C:/prog/webglstudy

Meie projektis asub **package.json** fail, mis sisaldab informatsiooni erinevate moodulite kohta, mida projekt vajab. Installeerimiseks anname käsurealt järgmise käsu:

```
npm install
```

Teine võimalus on installeerida moodulit ka globaalselt, et neid oleks võimalik kasutada kõikides projektides:

```
npm install -g browserify
npm install -g watchify
```

Kui kasutad mooduleid Javascript failis *require('express')*, siis installeerida lokaalselt. Kui plaan on kasutada mooduleid käsurealt, siis installeerida globaalselt.

Node.js õppetunni käivitamine

Node.js'i kasutame käsurealt. Olles projekti juurkataloogis on meil võimalik lihtsate skriptidega server ja moodulid käivitada.

```
//Käivitab esimese õppetunni.
npm run lesson00

npm run lesson01

...
```

Neid skripte kasutades on võimalik **lessons** kataloogis muudatusi sisse viia ja veebilehitsejat värskendades kohe ka muudatusi märgata.

Lõpetamiseks kasutada **CTRL** + **C** klahvikombinatsiooni.

Kui juhtub, et **Watchify** lõpetab töötamise, siis tuleb õppetund taaskäivitada.

2.2. Konteksti loomine ja varjundajate laadimine

Informatsioon

Avada: *lesson00/main.js* ja *utils/shaderprogramloader.js*.

Konteksti loomine

Enne kui saame WebGL'i kasutada on vaja luua kontekst, mis võimaldab meil seda kasutada. Konteksti küsime HTML5 **Canvas** elemendilt **getContext()** meetodi abil. Juhul kui veebilehitseja toetab WebGL'i on meil võimalik kasutada WebGL API'it.

Kui kontekst luuakse on mõistlik koheselt seadistada meetodiga **viewport()** vaateakna suurus. Kui vaateakna mõõtmed muutuvad on vaja meetod uuesti välja kutsuda.

Varjundajate laadimine

Kuna renderdamise järjekorras on meil võimalik defineerida programmid (vt Varjundajad), mis GPU'is jooksevad, on meil vaja need kuidagi defineerida ja kontekstiga siduda. Selle jaoks luuakse *WebGLProgram* objekt, millega on võimalik siduda tipu- ja pikslivarjundaja. Varjundajad on vaja aga kirjutada inimesele mõistlikumas keeles, kui seda on binaarkood ja seega on need vaja enne GPU'isse saatmist kompileerida. Varjundajate lähtekoodi on mõistlik hoida serveris, kuskil eraldi kaustas ja eraldi failides. Ennekõike seetõttu, et lõpuks võib varjundajaid olla päris palju ja lihtsalt *HTML*'i kirjutada ei ole mõistlik.

Varjundajaid laeme asünkroonselt (vaata failist **shaderprogramloader.js**). Materjalis kasutame laadimiseks **jQuery.ajax()** meetodit ja *callback* funktsiooni, mis kutsutakse välja siis, kui programmis olevad varjundajad on kompileeritud ja programmiga seotud. Renderdamise järjekorras on tähtis, et enne joonistamist oleks kontekstiga seotud töötav *WebGLProgram*.

Kuna laadimiseks kasutatav kood võib jääda suhteliselt segaseks, seletame, mis meetodit on vaja kutsuda, et programm saaks laetud:

1. Esiteks loome programmi **createProgram()** meetodi abil.
2. Kui lähtekood on laetud on vaja luua uus varjundaja meetodi **createShader()** abil.
3. Järgmiseks on vaja varjundajas määrata lähtekood **shaderSource()** meetodiga.
4. Kui lähtekood on määratud, tuleb kood ka kompileerida kasutades meetodit **compileShader()**.

5. Kompileeritud varjundajad seome programmiga **attachShader()** meetodi abil.
6. Kui kõik sammud on tehtud on võimalik renderdamisel seda programmi kasutada, sidudes selle kontekstiga **useProgram()**

2.3. 00 Esimene kolmnurk

Informatsioon

Apache: *builds/lesson00*

Lähtekoodi asukoht: *lessons/lesson00*

Varjundajate asukoht: *shaders/lesson00*

NodeJS käivituskäsk:

```
npm run lesson00
```

Esimene kolmnurk

Selles tunnis joonistame ekraanile lihtsa ühevärvilise kolmnurga. Alguses on vaja kirjutada päris palju koodi ja kõik tundub suhteliselt tülikas, kuid see annab väga suure vabaduse luua meile spetsiifiline rakendus. Paljud tegevuses saab korraliku rakenduse loomise käigus abstraheerida (nt Three.js).

Nagu igas järgmises tunnis, tuleb esiteks luua WebGL kontekst, laadida varjundajate lähtekood, luua programm ja kompileeritud varjundajad siduda programmiga. Ühes rakenduses võib kasutada niipalju programme, kui just parasjagu vaja on.

Vajaliku programmi laeme meetodiga:

```
var shaderProgram = shaderProgramLoader.getProgram(<tipuvarjundaja asukoht>, <lagivarjundaja asukoht>, <meetod, mis kutsutakse välja, kui programm on laetud>);
```

Selle meetodi viimane parameeter on *callback* funktsioon, mis käivitatakse siis, kui programm on täielikult laetud ja kasutuskõlblik. Kutsutakse välja meetod **render()**.

Tuleks kindlasti uurida renderdamise järjekorda.
Kui tahame midagi ekraanile joonistada on meil vaja andmeid, mida üldse joonistada. Andmed on tavaliselt lihtsalt koordinaadid objektruumis. Meie defineerime kolm tipu, mis moodustavad kolmnurga. Kõik kompleksed objektid, mis 3D modelleerimise tarkvaraga luuakse, koosnevad kolmnurkadest.

```
//Tippude andmed, mis moodustavad ühe kolmnurga
var myVerticesData = [
    0.0,   1.0,  0.0,   // Tipp 1
```

```
   -1.0,  -1.0,  0.0,   // Tipp 2
   1.0,  -1.0,  0.0    // Tipp 3
];
```

Ei ole ka eriti mõistlik, et võimas GPU, peaks koguaeg andmeid CPU käest küsima, mistõttu tuleb luua puhver GPU'is, kuhu kõik need andmed salvestada. GPU'is toimub paralleelne arvutamine ja seega on võimalik tohutu kiirusega sajad tuhanded tipud läbi töödelda ja kaadripuhvrisse saata.

```
//Loome puhvri, kuhu tipuandmed viia. Seome ka puhvri kontekstiga, et temale käske edasi anda
var vertexBuffer = GL.createBuffer();
GL.bindBuffer(GL.ARRAY_BUFFER, vertexBuffer);

//Anname loodud puhvrile andmed
GL.bufferData(GL.ARRAY_BUFFER, new Float32Array(myVerticesData), GL.STATIC_DRAW);
```

Kui tipud on puhvris olemas valmistume renderdamiseks. Selleks seome kontekstiga programmi, mida kasutada. Kui kontekstiga pole seotud ühtegi programmi ei ole meil ka varjundajaid ja renderdamise järjekord on katki.

```
//Määrame programmi, mida me renderdamisel kasutada tahame
GL.useProgram(shaderProgram);
```

Meie tipuvarjundaja näeb välja järgmine:

```
precision mediump float;

attribute vec3 a_Position;

void main(void) {
  gl_Position = vec4(a_Position, 1.0);
}
```

Tal on üks tipuatribuut, mida kasutame selleks, et määrata koordinaat pügamisruumis. Esiteks peame me teadma tema asukohta varjundajas, et positsioonile muutujat saata.

```
//Saame indeksi, mis näitab kus asub meie programmis kasutatavas tipuvarjundajas
//olev tipuatribuut nimega a_VertexPosition
var a_Position = GL.getAttribLocation(shaderProgram, "a_Position");
```

Kui oleme positsiooni saanud on meil võimalik määrata viit, mis viitab meie tipupuhvri olevatele andmetele.

```
//Määrame, kus tipuatribuut meie poolt kontekstiga seotud puhvris asub.
GL.vertexAttribPointer(a_Position, 3, GL.FLOAT, false, 0, 0);
```

Kuna meie atribuut ei pruugi olla kasutusel, siis on vaja see ka aktiveerida.

```
//Aktiveerime eelpool küsitud atribuudi
GL.enableVertexAttribArray(a_Position);
```

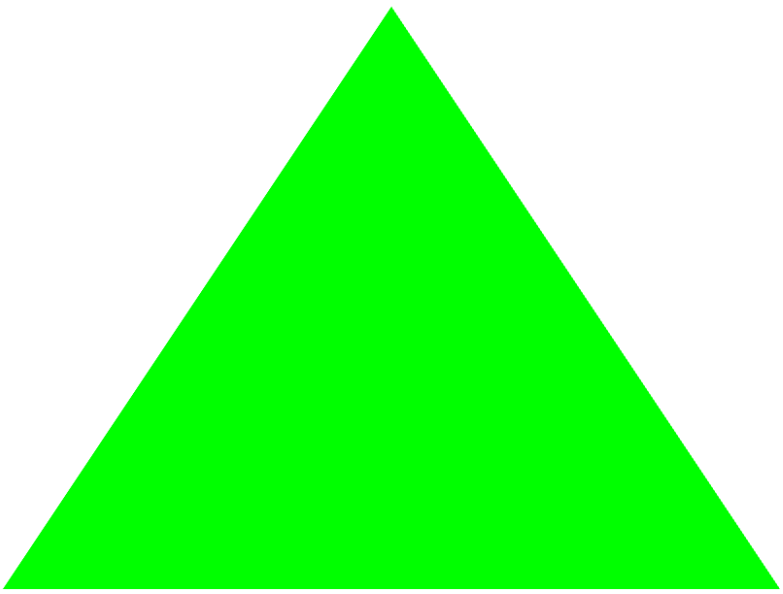
Lõpuks pärast suurt higi ja pisaraid oleme niikaugel, et kutsuda välja meetod, mis lõpuks kaadripuhvrisse ja meie Canvas elemendile pildi renderdaks. Meie tipud võetakse järjest ette ja saadetakse tipuvarjundajasse ning lõpuks kirjutatakse nad kaadripuhvrisse.

```
//Renderdame Kolmnurgad
GL.drawArrays(GL.TRIANGLES, 0, 3);
```

Tulemus

Kui käivitada ja vaadata, mis ekraanile ilmub, siis on näha, et roheline kolmnurk täidab praktiliselt terve elemendi (vt joonist 1). Põhjus tuleneb sellest, et meil ei ole maatrikseid. Ei toimu homogeenset jagamist, mistõttu puudu perspektiiv. Tipuandmed on lihtsalt normaliseeritud seadme koordinaadid:

```
-1 < x < 1
-1 < y < 1

Alumine vasak nurk: (-1, -1)
Ülemine vasak nurk: (-1,  1)
Alumine parem nurk: ( 1, -1)
Ülemine parem nurk: ( 1,  1)
```



Joonis 1 Roheline kolmnurk

Ülesanded

1. Muuda pikslivarjundajas (*fragment.shader*) kolmnurga värvus punaseks.
2. Muuda tipuandmeid nii, et tekiks nelinurk. Nelinurk koosneb kuuest tipust, kus kolmnurkade kaks tipu on samadel koordinaatidel.
3. Uuri, mis on *Normalized device coordinates*.

2.4. 01 Tipu värv

Informatsioon

Apache: *builds/lesson01*

Lähtekoodi asukoht: lessons/lesson01

Varjundajate asukoht: shaders/lesson01

NodeJS käivituskäsk:

```
npm run lesson01
```

Tipu värv

Selles tunnis lisame tippudele ka värvused. Värvused viime tipuvarjundajasse samamoodi nagu positsiooni ehk tipuatribuudina.

Tipuvarjundaja

```
precision mediump float;

attribute vec3 a_Position;
attribute vec3 a_Color;

varying vec3 v_Color;

void main(void) {
  gl_Position = vec4(a_Position, 1.0);

  v_Color = a_Color;
}
```

Pikslivarjundaja

```
precision mediump float;

varying vec3 v_Color;

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

Kuna atribuute saab kasutada vaid tipuvarjundaja, siis on meil vaja kasutada muutlikku muutujat (*varying*), mille abil on meil võimalik andmeid pikslivarjundajasse saata. Pikslivarjundajasse tulev väärtus on interpoleeritud (vt Varjundajad).

Meil on olemas tipuandmete massiiv. Nüüd loome värvide jaoks värvide massiiv. Värvid on RGB kujul.

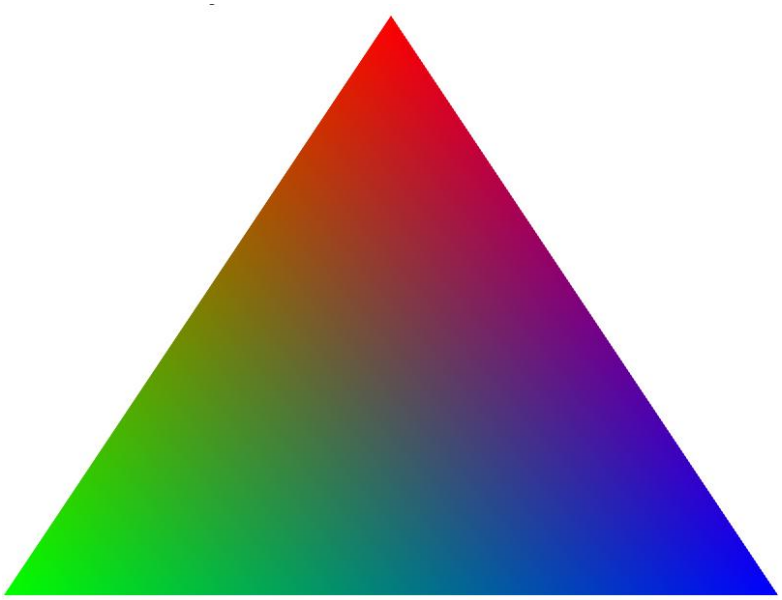
```
//Tippude värvid
   var myVerticesColor = [
       1.0,  0.0,  0.0,   // Tipp 1 punane
       0.0,  1.0,  0.0,   // Tipp 2 roheline
       0.0,  0.0,  1.0    // Tipp 3 sinine
   ];
```

Sarnaselt eelmise tunniga on vaja luua uus puhver ja see seadistada ning õige viit määrata. Tuleb mainida, et praktikas hoitakse kõiki tipuandmeid ühes puhvris. Tavaliselt hoitakse mitmete kui mitte kõikide objektide andmeid ühes puhvris.

Tähtis on aru saada, et renderdamiseks vajalikud puhvrid oleks kontekstiga seotud. Iga kord, kui puhvriga on vaja tegevusi teha on see vaja kontekstiga siduda.

Tulemus

Käivitades rakenduse ilmub ekraanile kena värviline kolmnurk (vt joonist 1), milles olevad pikslid on interpoleeritud.



Joonis 1 Värviline kolmnurk

Ülesanded

1. Uuri sügavamalt, mida iga meetod teeb. **WebGL funktsioonid**
2. Vaheta värve ja jälgi muudatusi.
3. Muuda pikslivarjundajas viimast läbipaistvuse parameetrit ja vaata, mis juhtub.
4. Muuda kolmnurk nelinurgaks

2.5. 02 Indeksite kasutamine

Informatsioon

Apache: *builds/lesson02*

Lähtekoodi asukoht: lessons/lesson02

Varjundajate asukoht: shaders/lesson02

NodeJS käivituskäsk:

```
npm run lesson02
```

Indeksite kasutamine

Stseen, mis sisaldab kompleksseid objekte koosneb sadadest tuhandetest tippudest. Ühte tippu saab tavaliselt kasutada mitme kolmnurga defineerimiseks. Mõistlikum oleks kasutada indekseid, mis määravad, millised tipud moodustavad kolmnurga (vt Tipuandmed). Niiviisi on meil võimalik terves stseenis olevad tipud ladustada ühte puhvrisse ja indeksite abil määrata, millised tipud moodustavad mingi objekti.

Esiteks on meil vaja defineerida elementide massiiv, mis sisaldab endas indekseid, luua ka seekord puhver ja see andmetega täita.

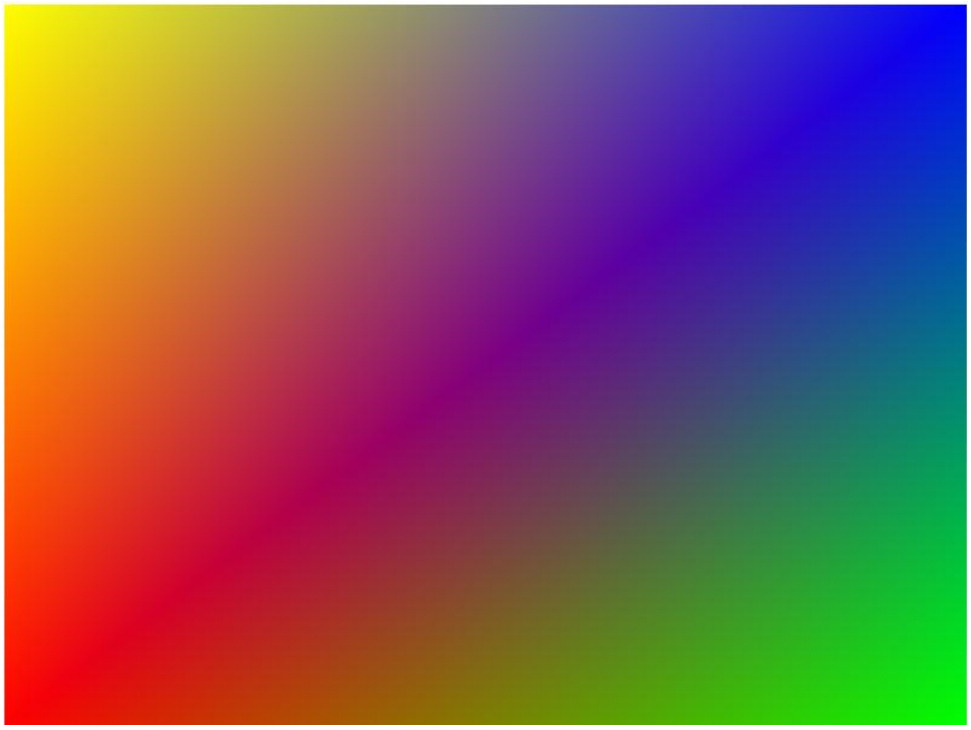
```
//Tippude indeksid
var myIndicesData = [
    0,  1,  2,
    0,  2,  3
];

//Loome puhvri, kuhu indeksid viia. Seome ka puhvri kontekstiga, et temale käske edasi anda
var indexBuffer = GL.createBuffer();
indexBuffer.numberOfIndexes = 6;
GL.bindBuffer(GL.ELEMENT_ARRAY_BUFFER, indexBuffer);

//Anname loodud puhvrile andmed
 GL.bufferData(GL.ELEMENT_ARRAY_BUFFER, new Uint16Array(myIndicesData), GL.STATIC_DRAW);
```

Kui enne kasutasime meetodit **drawArrays()**, siis seekord kasutame hoopiski meetodit **drawElements()**. Tuleb meeles pidada, et kui tahame puhvrit kasutada peame selle ka kontekstiga siduma.

```
//Renderdame kolmnurgad indeksite järgi
GL.bindBuffer(GL.ELEMENT_ARRAY_BUFFER, indexBuffer);
GL.drawElements(GL.TRIANGLES, indexBuffer.numberOfIndexes, GL.UNSIGNED_SHORT, 0);
```

Tulemus

Seekord määrame indeksite abil kahest kolmnurgast koosneva nelinurga, mistõttu näeme ekraanil värvilist nelinurka.



Joonis 1 Värviline nelinurk

Ülesanded

1. Miks on kompleksete stseenide renderdamisel parem kasutada indekseid?
2. Joonista elemendid, mitte *GL.TRIANGLES* moodust kasutades, vaid *GL.LINE_LOOP* abil. Mitut kolmnurka näed?
3. Muuda rakendust nii, et joontest tekiks ümbrik.

2.6. 03 Maatriksid

Informatsioon

Apache: *builds/lesson03*

Lähtekoodi asukoht: lessons/lesson03

Varjundajate asukoht: shaders/lesson03

NodeJS käivituskäsk:

```
npm run lesson03
```

Maatriksid

"Unfortunately, no one can be told what the matrix is. You have to see it for yourself." - Morpheus (The Matrix, 1999)

Kätte on jõudnud aeg kasutada maatrikseid. Ei tasu muretseda. Maatriksite jaoks kasutame **glMatrix** teeki, mille abil on võimalik kõik kalkulatsioonid kenasti ära teha. Enne kui edasi minna, tasuks kindlasti vaadata koordinaatruumide osa ja üritada aru saada vektoritest ja maatriksitest. Võin garanteerida, et alguses ei saa te mitte kui midagi aru. Soovitatav on seega rahulikuks jääda ja praktikaga edasi minna. Alguses piisab arusaamisest, mida mingit tüüpi maatriks teeb.

Kui me tahame objektiga manipuleerida, näiteks seda liiguta, on meil vaja maatrikseid, nimelt **mudelmaatriksit**. Meil oleks vaja 2D ekraanil jätta ka mulje, et me vaatame läbi akna 3D ruumi, milleks kasutame **projektsioonmaatriksit**. Me tahame maailma näha ka erinevast vaatevinklist, mistõttu kasutame **kaameramaatriksit**.

Maatrikseid on võimalik korrutamise abil kombineerida, viies tipud ühest ruumist teise. **Mudelmaatriksi** abil saame tipud viia maailmaruumi ehk kohta, kus kõik teised objektid asuvad. Igal objektil on oma mudelmaatriks, mis neid erinevale positsioonile viiks. Mudelmaatriks sisaldab ka objekti pööramist ja mõõtkava suurendamist, kui see on vajalik.

Tahame 3D maailmas ringi liikuda ja stseeni erinevatest külgedest vaadata. Tihti on vajadus, et meil oleks implementeeritud erinevad kaamerad, kus näiteks üks tiirleb ümber objekti ja teine võimaldab meil vabalt liikuda. Seetõttu on vajalik **kaameramaatriks**.

Me tahame jätta mulje, et pilt mida me vaatame on ruumiline, kuigi tegelikult on ta 2D ekraanil, mistõttu vajame perspektiivi projekteerimist ja kasutame **projektsioonimaatriksit**.

Kasutades projektsioonmaatriksiks ortograafilist projektsiooni ei toimu perspektiivi jagamist. Kui perspektiivi projekteerimiseks kasutame tüvipüramiidi, siis ortograafilise projektsiooni jaoks risttahukat. **mat4.ortho()**.
Asume koodi kallale.

Kuna meil on vaja igale tipule, mis saadetakse tipuvarjundajasse rakendata maatrikseid, muudame tipuvarjundajat. Esiteks on vaja defineerida ühtsed muutujad (*uniform*). Nendele muutujatele on võimalik ligipääseda nii tipu- kui ka lagivarjundajast.

```
uniform mat4 u_ModelMatrix;
uniform mat4 u_ViewMatrix;
uniform mat4 u_ProjectionMatrix;
```

Hetkel on meil vaja neid ainult tipuvarjundajas, kus rakendame neid sissetulnud tipu peal.

```
void main(void) {
  gl_Position = u_ProjectionMatrix * u_ViewMatrix * u_ModelMatrix * vec4(a_Position, 1.0);

  v_Color = a_Color;
}
```

Maatriksite korrutamine WebGL toimub tagurpidi vastavalt selle, kuidas meie mõtleme!
Järgnevalt on vaja defineerida meie maatriksid.

Initsialiseerides maatriksid **mat4.create()** meetodi abil on nad identiteedi maatriksid. Maatriksid, millega vektorit (tippu) korrutades selle koordinaadid jäävad samaks.

Mudelmaatriks

```
//Mudelmaatriks, millega objektiruumist maailmaruumi saada
var modelMatrix = mat4.create();

//Punkt, kus objekt hetkel asub
var objectAt = [0.0, 0.0, -5.0];
//Kasutades translatsiooni, saame mudelmaatriksiga objekti liigutada
mat4.translate(modelMatrix, modelMatrix, objectAt);
```

Mudelmaatriksit defineerides transleerime ta ka (0, 0, 0) koordinaatidelt veidi ekraani sisse. **-z** on paremakäe süsteemis ekraani sisse. Kasutame selleks **mat4.translate()**meetodit.

Suru parem rusikas kokku. Tõsta nimetissõrm püsti (y-telg). Liiguta pöial kõrvale (x-telg). Liiguta keskmine sõrm nii, et see näitaks sinu suunas (z-telg). Nüüd peaks nimetissõrm näitama otse lakke, pöial sinust paremale ja keskmine sõrm osutama sinu suunas. Sõrme suund näitab telje positiivset suunda.

Kaameramaatriks

```
//Kaameramaatriks, millega maailmaruumist kaameraruumi saada
var viewMatrix = mat4.create();

//Defineerime vektorid, mille abil on võimalik kaameraruumi baasvektorid arvutada
var cameraAt = [0, 0, 5];            //Asub maailmaruumis nendel koordinaatidel
var lookAt = [0, 0, -1];                        //Mis punkti kaamera vaatab. Paremakäe koordinaatsüsteemis on -z ekraani sisse
var up = [0, 1, 0];                    //Vektor, mis näitab, kus on kaamera ülesse suunda näitav vektor

//Kalkuleerime koordinaatide järgi kaameramaatriksi
mat4.lookAt(viewMatrix, cameraAt, lookAt, up);
```

Järgnevalt defineerime koordinaadid, millega saame **mat4.lookAt()** meetodi abil **kaameramaatriksi** kalkuleerida:

- **cameraAt** - Kaamera asukoht maailmas. 5 ühikut meie suunas.
- **lookAt** - Punkt, mida kaamera vaatab. Vaatab ekraani sisse punkti -1.
- **up** - Kus asub kaamera ülesvektor. Meil on see suunatud otse taevasse.

Projektsioonmaatriks

```
//Projektsioonmaatriks, et pügamisruumi saada. Kasutades glMatrix teeki genereerime ka püramiidi, kuhu sisse objektid lähevad.
var projectionMatrix = mat4.create();
 mat4.perspective(projectionMatrix, 45.0, GL.viewportWidth / GL.viewportHeight, 1.0, 1000.0);
```

Projektsioonmaatriksi abil on võimalik tipud ette valmistada projekteerimiseks 2D ruumi. See maatriks iseloomustab meie tüvipüramiidi, mis defineerib meie ruumi, mida me näeme.

Hetkel on vaatevälja nurk 45 kraadi, mis näeb suhteliselt sobilik välja. Maatriksi loomiseks on vaja määrata ka tüvipüramiidi lähim ja kaugem tasand, mis on **1.0** ja **1000.0**.

Tasub olla ettevaatlik liiga suurte suhetega lähima ja kaugema tasandi vahel. Liiga suure suhte korral võib sügavusväärtuse asuda väga väga väikeses vahemikus - 0.999999 < z < 1.0.

Täiesti uued meetodid

Seekord nagu kord ja kohus kutsume välja meetodid, mida tuleb kutsuda igakord, kui läheb uue kaadri renderdamiseks.

```
GL.clearColor(0.0, 0.0, 0.0, 1.0);
GL.clear(GL.COLOR_BUFFER_BIT | GL.DEPTH_BUFFER_BIT);
```

Esimene meetod määrab puhastusvärvuse ja teine puhastab meie värvus- ja sügavuspuhvri. Hetkel see väga tähtis ei ole, sest me renderdame vaid ühe korra. Meil ei ole mingit tsüklit, kuid tsüklis renderdamisel on vaja puhvrid alati puhastada.

Ühtsed muutujad

Esiteks on meil vaja nagu ikka saada muutujate asukohad meie poolt aktiveeritud programmis.

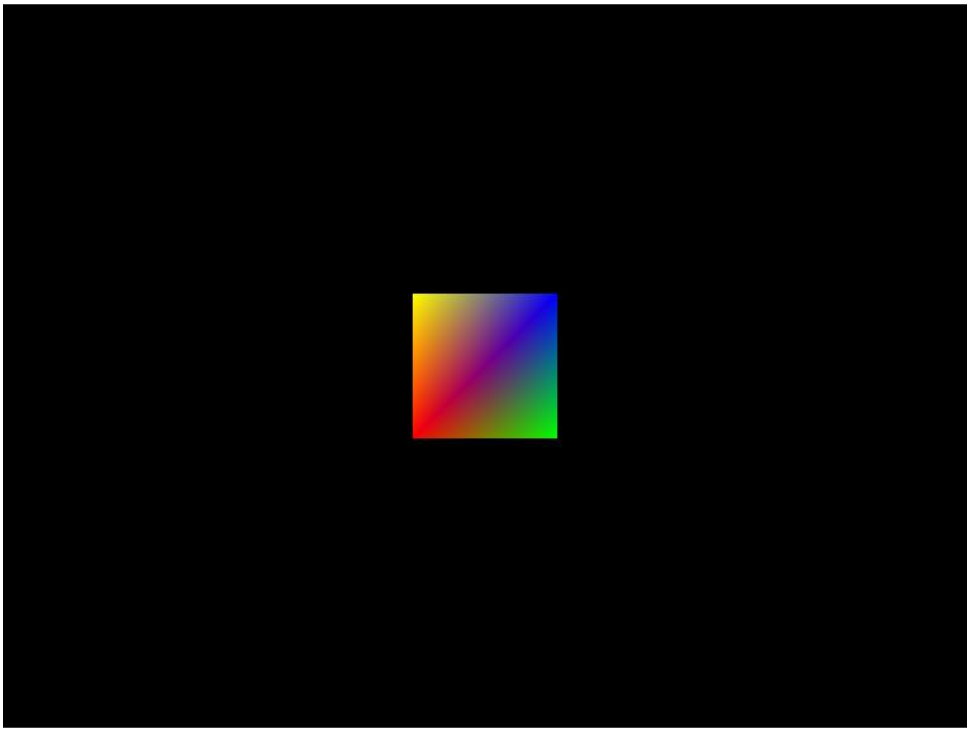
```
//Saame ühtsete muutujate asukohad
var u_ModelMatrix = GL.getUniformLocation(shaderProgram, "u_ModelMatrix");
var u_ViewMatrix = GL.getUniformLocation(shaderProgram, "u_ViewMatrix");
var u_ProjectionMatrix = GL.getUniformLocation(shaderProgram, "u_ProjectionMatrix");
```

Järgnevalt on vaja meie maatriksid programmi ka saata.

```
//Saadame meie maatriksid ka varjundajasse
GL.uniformMatrix4fv(u_ModelMatrix, false, modelMatrix);
GL.uniformMatrix4fv(u_ViewMatrix, false, viewMatrix);
GL.uniformMatrix4fv(u_ProjectionMatrix, false, projectionMatrix);
```

Tulemus

Maatriksite abil määrasime objektile uue asukoha maailmas. Määrasime ka asukoha, kust me objekti vaatama. Perspektiivi ja vaatenurga tõttu on objekt justkui kaugemal ja ei täida enam tervet ekraani.



Joonis 1 Maatriksite kasutamise lõpptulemus

Ülesanded

1. Enne meetodit **mat4.translate()** kutsumist mudelmaatriksi peal, kasuta ka**mat4.rotateZ()** ja pööra objekti 45 kraadi. Ei kasutata kraade, vaid radiaane. Pi ehk ~3.141 on 180 kraadi.
2. Muuda objekti või kaamera asukohta ja täida terve ekraan objektiga.
3. Renderda nelinurga kõrvale kolmnurk.

Lisamaterjal

- **glMatrix** dokumentatsioon - http://glmatrix.net/docs/2.2.0/

2.7. 04 Liikumine

Informatsioon

Apache: *builds/lesson04*

Lähtekoodi asukoht: lessons/lesson04

Varjundajate asukoht: shaders/lesson04

NodeJS käivituskäsk:

```
npm run lesson04
```

Avada ka: utils/looper.js

Liikumine

Selles tunnis vaatame, kuidas liigutada kaamerat ümber objekti ja ka objekti ümber oma x-telje.

Kui me tahame, et meil toimuks objektide liikumine on meil vaja tsüklit. Seetõttu loome *Looper* klassi, mis tsüklis meie **render()** meetodit välja kutsuks. Hetkel on aga probleem, nimelt toimub meil **render()** meetodis ka kõike muud, kui ainult renderdamine. Kood vajab korrastamist. Seletame seega lahti, kuidas peaks rakendus välja nägema, kui meil on tegu tsükliga.

Looper

Esiteks räägime, mida teeb *Looper*. Veebilehitsejates on meetod **requestAnimationFrame()**. Esimeseks parameetriks tahab funktsioon *callback* meetodit, mis kutsutakse välja iga tsükli alguses. Kutsudes välja meetodit **looper.loop()** hakatakse meie poolt defineeritud funktsiooni tsüklis jooksutama. Selles funktsioonis peaks toimuma objektide uuendamine ja renderdamine.

Animatsioonide puhul on väga tähtis aeg, mis läheb ühe tsükli läbimiseks. Kui meil on võimas protsessor ja graafikakaart, siis läbitakse üks tsükkel kiiremini, kui nõrgema masina peal. Kui meil toimub tsüklis objektide liigutamine, siis ühes sekundis võib võimsamas masinas objektid liikuda mitmeid kordi rohkem, kui aeglasema arvutiga. Probleemi

lahendamiseks on meil vaja **praeguse ja eelmise aja vahet** ehk **deltaaega**. Seetõttu on meil loodud ka meetod **calculateDeltaTime()**:

```
calculateDeltaTime: function() {
     var timeNow = new Date().getTime();

     if(this.lastTime != 0)
         this.deltaTime = (timeNow - this.lastTime) / 16;

     this.lastTime = timeNow;
},
```

Deltaaeg on võimsamates masinates väiksem ja korrutades selle objekti kiirusega, liigub objekt väiksema vahemaa.

Meetod setup()

Meetod kutsutakse välja, kui varjundajad on laetud. Siin toimub puhvrite loomine ja andmetega täitmine ning maatriksite initsialiseerimine. Ühesõnaga kõik, mis tsüklis ei muutu.

Ühtsete muutujate ja atribuutide asukohad määrame ka just selles funktsioonis. Uued asukohad tuleb küsida ainult juhul, kui me võtame kasutusel uue programmi.

Meetod *render()*

Meetod kutsutakse välja iga tsükkel. Selleks hetkeks on kõik vajalikud andmed puhvris ja initsialiseeritud. Enne renderdamist on vaja ainult puhvrid kontekstiga siduda, viidata õigetele tipuatribuudi andmetele, määrata ühtsed muutujad ja lõpetuks kogu kompott renderdada.

Objektide ja kaamera liigutamine

Enne kaadri renderdamist ekraanile, tuleb meil objektide liigutamiseks maatrikseid uuendada.

```
//Kutsutakse välja Looper objektis iga kaader
function loop(deltaTime) {
    update(deltaTime);

    render();
}

//Uuendab andmeid, et oleks võimalik stseen liikuma panna
```

```
function update(deltaTime) {
    APP.time += deltaTime / 100;

    updateCamera();
    updateObject();
}
```

Kaamera liigutamine - meetod *updateCamera()*

Tahame, et kaamera pöörleks ümber objekti. Kaamera liigutamiseks on vaja uuendada ka **kaameramaatriksit**. Tuleb tegeleda matemaatikaga (*Oh noes!*).

Kaamera liigutamiseks defineerisime muutuja **time**, mille abil on võimalik polaarses koordinaatsüsteemis seda liigutada. Aja suurendamiseks kasutame *Looper*'st saadud deltaaega. Mõistlik on see jagata ka mingi arvuga, et see liiga kiiresti ümber objekti ei liiguks.

Me tahame, et kaamera pöörleks ümber objekti mingil kindal kaugusel, milleks on vaja raadiust. Kuna me liigume polaarses koordinaatsüsteemis, ehk 2D ruumis võime võtta kasutusel ka muutuja **cameraHeight**, et kaamerasobivale kõrgusele viia.

Kaameramaatriksi loomiseks kasutame meetodit **mat4.lookAt()**. Meil on vaja teada:

- Kaamera asukohta
- Punkti, mida kaamera vaatab
- Kaamera ülesse suunda näitavat vektorit

Kaamera kauguse objektist saame kasutades lihtsat trigonomeetriat. Toome koordinaadid ristkoordinaatsüsteemi, kus **x = raadius * Math.cos(nurk)** ja **y = raadius * Math.sin(nurk)** (meie puhul z = y).

```
//Leiame kaamera positsiooni, mis ajas liigub polaarses koordinaatsüsteemis ja mille teisendame ristkoordinaatsüsteemi
APP.cameraAt = [APP.objectAt[0] + radius * Math.cos(APP.time),       // X
                cameraHeight + APP.objectAt[1],                      // Y
                APP.objectAt[2] + radius * Math.sin(APP.time)];      // Z
```

Kaamera asukoht on leitud. Järgnevalt tuleb leida, mis vektor kaamerast objektini. Selleks kasutame vektorite lahutamist.

```
 //Leiame vektori, kaamerast objektini
var lookDirection = [APP.objectAt[0] - APP.cameraAt[0],               // X
                     APP.objectAt[1] - APP.cameraAt[1],               // Y
```

```
                    APP.objectAt[2] - APP.cameraAt[2]];              // Z
```

Kuna meil on vaja punkti, mida kaamera vaatab saame kaks vektorit liita ja asumegi punktis, mida kaamera vaatab.

```
//Leiame punkti, mida kaamera vaatab
vec3.add(APP.lookAt, APP.cameraAt, lookDirection);
```

Lõpuks saame täiesti uue kaameramaatriksi, mis sisaldab uusi koordinaate.

```
//Uuendame kaameramaatriksit
mat4.lookAt(APP.viewMatrix, APP.cameraAt, APP.lookAt, APP.up);
```

Objekti liigutamine - meetod *updateObject()*

Objekti liigutame lihtsalt ümber oma x-telje mingi juhusliku radiaani võrra.

```
mat4.rotateX(APP.modelMatrix, APP.modelMatrix, 0.005);
```

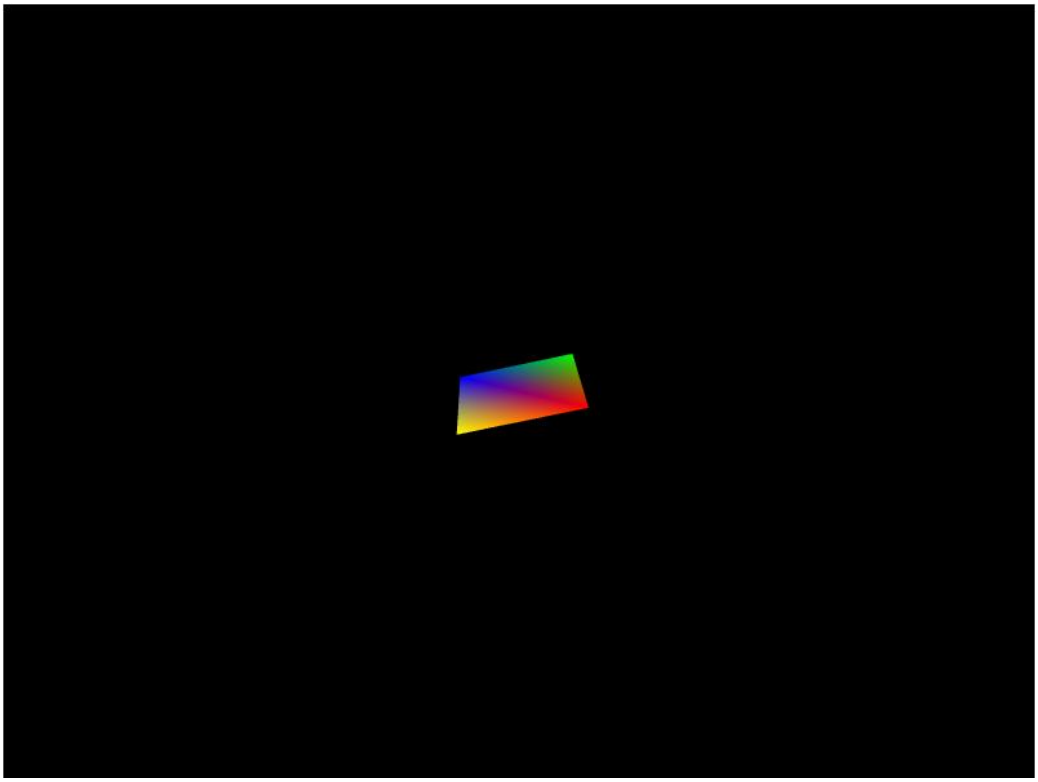
Kokkuvõte

Võtame kõik etapid kokku:

1. Programmi laadimine - *getProgram()*
2. Puhvrite, maatriksite sätestamine - *setup()*
3. Tsükkel - *loop()*
    - Objektide uuendamine *update()*
        - Kaamera liigutamine *updateCamera()*
        - Objekti liigutamine *updateObject()*
    - Renderdamine *render()*

Tulemus

Rakendust käivitades liigub objekt ümber oma x-telje ja kaamera ümber objekti. Üks kaader liikumist võib olla selline nagu näidatud jooniselt 1.



Joonis 1 Kaader liikumisest

Ülesanded

1. Muuda kaamera liikumissuunda.
2. Muuda objekti liikumist nii, et iga kaader pööraks objekt umbes 180 kraadi.
3. Renderda eelmises tunnis enda tehtud kolmnurk ja pane ta liikuma nelinurga kõrvale.

2.8. 05 Tekstuur ja 3D objekt

Informatsioon

Apache: *builds/lesson05*

Lähtekoodi asukoht: lessons/lesson05

Varjundajate asukoht: shaders/lesson05

NodeJS käivituskäsk:

```
npm run lesson05
```

Aeg on objekt kaunistada tekstuuri, sest kuidas siis muidu. Selles tunnis käsitleme ka, kuidas luua 3D objekt.

3D objekt

Praeguseks peaks juba arusaadav olema, et objektid koosnevad kolmnurkadest, mis koosnevad tippudest. Meil on vaja seega defineerida tipud selliselt, et nad moodustaksid kolmnurgad, mis omavahel moodustaksid mingi objekti.

Eelmisest koodist ei muutu väga palju. Meil on vaja defineerida lihtsalt rohkem tippe ja rohkem indekseid, et nad moodustaksid kuubi. Kindlasti küsite, mis on need kaks viimast koordinaati - iga asi omal ajal.

```
//Tippude andmed. Tipu koordinaadid x, y, z ja tekstuuri koordinaadid u, v
APP.myVerticesData = [
    //Esimene külg
    -1.0, -1.0,  1.0,  0.0, 1.0,            //ALUMINE VASAK NURK
     1.0, -1.0,  1.0,  1.0, 1.0,            //ALUMINE PAREM NURK
     1.0,  1.0,  1.0,  1.0, 0.0,            //ÜLEMINE PAREM NURK
    -1.0,  1.0,  1.0,  0.0, 0.0,            //ÜLEMINE VASAK NURK

...

//Tippude indeksid
APP.myIndicesData = [
    0, 1, 2,      0, 2, 3,    // Esimene külg
    4, 5, 6,      4, 6, 7,    // Tagumine külg
    8, 9, 10,     8, 10, 11,  // Ülemine külg
```

```
    12, 13, 14,   12, 14, 15, // Alumine külg
    16, 17, 18,   16, 18, 19, // Parem külg
    20, 21, 22,   20, 22, 23  // Vasak külg
];

//Defineerime indeksite arvu
APP.indexBuffer.numberOfIndexes = 36;

//Joonistame 36 indeksi abil kolmnurgad
GL.drawElements(GL.TRIANGLES, APP.indexBuffer.numberOfIndexes, GL.UNSIGNED_SHORT, 0);
```

Käsitsi muidugi ei maksa niiviisi objekte ehitama hakata. Väga nüri töö. Sellepärast ongi modelleerimise tarkvara, mis meie eest töö ära teeks. Ainuke asi, mis tuleb teha on kirjutada kood, mis mingi tarkvaraga loodud formaati lugeda oskaks või seal olevad andmed meile vajalikku vormi viiks.

Tekstuur

Viimased kaks koordinaati meie **myVerticesData** massiivis on tekstuuri koordinaadid, mida saame saata tipuvarjundajasse ja sealt interpoleeritud väärtuse pikslivarjundajasse. Nagu hetkel näha kasutame koordinaate vahemikus [0, 1] ja seega terve tekstuur on näha. Kuna meie tekstuur on ruudukujuline ja pind on ruudukujuline, siis pilt ei ole deformeerunud. Kui me kasutaks samasuguseid koordinaate ristküliku peal, oleks tekstuur venitatud.

Tekstuurid peavad WebGL's olema 2. astmed: 2, 4, 8, 16, 32, 64, 128, 256...

```
//Loome uue tekstuuri ja koos sellega 1x1 pikslise pildi, mis kuvatakse senikaua, kuni
tekstuur saab laetud
 APP.texture = GL.createTexture();
GL.bindTexture(GL.TEXTURE_2D, APP.texture);
GL.texImage2D(GL.TEXTURE_2D,  0,  GL.RGBA,  1,  1,  0,  GL.RGBA,    GL.UNSIGNED_BYTE,  new
Uint8Array([1, 1, 1, 1]));
GL.texParameterf(GL.TEXTURE_2D, GL.TEXTURE_WRAP_S, GL.CLAMP_TO_EDGE);
GL.texParameterf(GL.TEXTURE_2D, GL.TEXTURE_WRAP_T, GL.CLAMP_TO_EDGE);
GL.texParameteri(GL.TEXTURE_2D, GL.TEXTURE_MAG_FILTER, GL.NEAREST);
GL.texParameteri(GL.TEXTURE_2D, GL.TEXTURE_MIN_FILTER, GL.NEAREST);

var image = new Image();

image.onload = function() {
    GL.bindTexture(GL.TEXTURE_2D, APP.texture);
    GL.texImage2D(GL.TEXTURE_2D, 0, GL.RGB, GL.RGB,  GL.UNSIGNED_BYTE, image);
    GL.bindTexture(GL.TEXTURE_2D, null);
```

```
};
image.src = TEXTURE_PATH;
```

Laadimine

Tekstuuri laadimine toimub meetodis **setupAndLoadTexture()**.

Kuidas siis laadimine käib? Kui me oleme puhvri loonud, siis on võimalik kasutada meetodit **texImage2D()**, kuhu saame kaasa anda andmed, mis tekstuuri salvestatakse. Meil on võimalik sinna laadida pildifaile, *canvas* elementi, *video* elementi või lihtsalt massiivi, mis sisaldab värve. Saates sinna lihtsalt massiivi on vaja defineerida ka pildi suurus pikslites.

Kui me tahame laadida pilti, mis asub serveris, tuleb luua *Image* objekt ja määrata tema asukoht (*src*). Laadimine toimub muidugi asünkroonselt ja on vaja määrata **onload** *callback* meetod, mis käivitatakse, kui objekt on laetud. Seal laeme **texImage2D** abil laetud pildi puhvrisse.

Probleem on selles, et tekstuuride laadimine käib asünkroonselt ja renderdamine võib toimuda enne, kui meil saab pilt laetud ja puhvrisse pandud. Seetõttu määrame alguses tekstuuriks 1x1 pikslit suure pildi.

Varjundajasse saatmine

Tekstuuri saadame ühtse muutujana. Tekstuuri koordinaadid saadame atribuudina. Enne on vaja kindlasti küsida, kus on muutujate asukohad.

```
//Saame värviatribuudi asukoha
APP.a_TextureCoord = GL.getAttribLocation(shaderProgram, "a_TextureCoord");

APP.u_Texture = GL.getUniformLocation(shaderProgram, "u_Texture");
```

Järgnevalt suundume **render()** meetodisse. Meil on atribuudid nüüd samas puhvris (defineerisime positsioonid ja tekstuuri koordinaadid samas massiivis). Meetodi **vertexAttribPointer** parameetriteks on:

- Atribuudi asukoht
- Mitmest arvust atribuut koosneb
- Mis tüüpi on arv
- Kas arv on normaliseeritud ehk pikkus = 1.0
- Samm järgmise elemendini. Samm on baitides.

- Nihe 0 elemendist sinna, kuhu tahame viidata.

```
//Seome tipupuhvri ja määrame, kus tipuatribuut massiivis asub.
GL.bindBuffer(GL.ARRAY_BUFFER, APP.vertexBuffer);
GL.vertexAttribPointer(APP.a_Position, 3, GL.FLOAT, false, APP.vertexSize * 4, 0);
GL.vertexAttribPointer(APP.a_TextureCoord, 2, GL.FLOAT, false, APP.vertexSize * 4,
APP.vertexSize * 4 - 2 * 4);
```

Uus tipuatribuut on vaja ka aktiveerida.

```
 GL.enableVertexAttribArray(APP.a_TextureCoord);
```

Järgnevalt saadame tekstuuri ühtse muutujana. Meil tuleb see ka aktiveerida.

```
//Aktiveerime ja määrame tekstuuri
GL.activeTexture(GL.TEXTURE0);
GL.bindTexture(GL.TEXTURE_2D, APP.texture);
GL.uniform1i(APP.u_Texture, 0);
```

Varjundajad

Tipuvarjundajas on meil vaja atribuut (tekstuuri koordinaat) lisada ja edasi lihtsalt pikslivarjundajasse saata.

```
precision mediump float;

attribute vec3 a_Position;
attribute vec2 a_TextureCoord;

uniform mat4 u_ModelMatrix;
uniform mat4 u_ViewMatrix;
uniform mat4 u_ProjectionMatrix;

varying vec2 v_TextureCoord;

void main(void) {
  gl_Position = u_ProjectionMatrix * u_ViewMatrix * u_ModelMatrix * vec4(a_Position, 1.0);

  v_TextureCoord = a_TextureCoord;
}
```

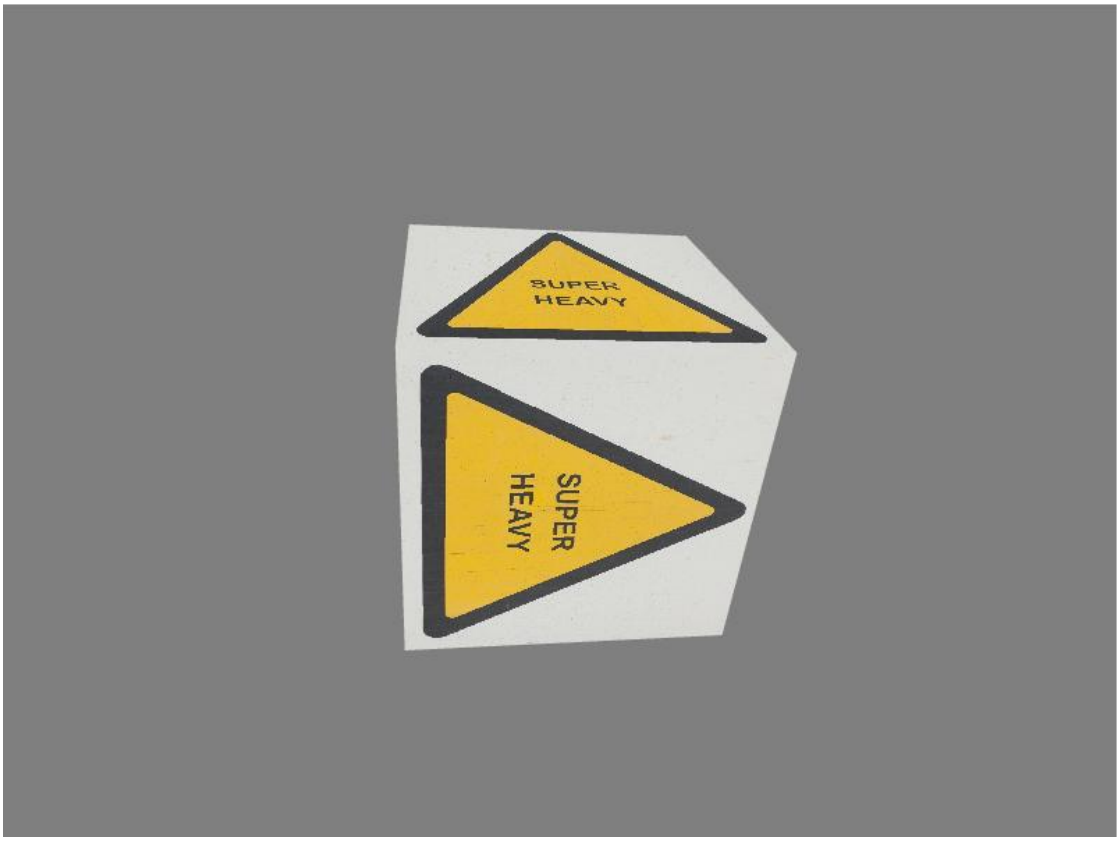
Pikslivarjundajas saame interpoleeritud koordinaadid. Muidugi on meil vaja määrata ka ühtne muutuja, et oleks võimalik meie poolt aktiveeritud tekstuurist mingi värv kätte saada.

Tekstuur tuleb meil tüübina **sampler2D**. Värvi kättesaamiseks kasutame **texture2D** meetodit, kuhu anname parameetriks tekstuuri ja koordinaadid, mida tahame saada. Tagastatakse meile värvus.

```
precision mediump float;

varying vec2 v_TextureCoord;

uniform sampler2D u_Texture;

void main(void) {
  vec4 color = texture2D(u_Texture, vec2(v_TextureCoord.s, v_TextureCoord.t));
  gl_FragColor = color;
}
```

Tulemus

Me defineerisime 3D objekti, mis tekitab illusiooni tahkest objektist, mis tegelikult on aga seest tühi. Objekt koosneb kolmnurkadest, millele on "keebina" peale pandud tekstuur.



Joonis 1 Tekstuur ja 3D objekt

Ülesanded

1. Muuda tekstuuri koordinaate ja vaata, mis juhtub.
2. Muuda tippude koordinaate ja vaata, mis juhtub.
3. Kasuta endale sobivat tekstuuri.
4. Loo kuubi kõrvale püramiid ja pane see mingis suunas liikuma. Kasuta püramiidi peal teistsugust tekstuuri.
5. Kasuta ühe objekti peal mitut erinevat tekstuuri.

2.9. 06 Hiir

Informatsioon

Apache: *builds/lesson06*

Lähtekoodi asukoht: lessons/lesson06

Varjundajate asukoht: shaders/lesson06

NodeJS Käivituskäsk:

```
npm run lesson06
```

Kaamera liigutamine hiirega

Järgnevalt paneme kaamera hiire abil liikuma. Meil on võimalik kasutada veebilehitseja kuulareid. Kui kasutaja vajutab hiire klahvi alla ja mingis suunas seda liigutab on tal võimalik kaamerat juhtida. Kaamera pöörleb ka seekord ümber objekti. Me tahaks ka, et tal oleks võimalik objekti vaadata kaugemalt ja lähemalt. Selle jaoks kasutame hiire rullikut, millega saame muuta orbiidi raadiust. Seekord tuleb meil kasutada sfäärilist koordinaatsüsteemi.

Selles osas on jällegi päris palju matemaatikat. Kaamerat implementeerides me sellest ei pääse. Tähtis on teada, et implementeerimise mooduseid on mitmeid ja igal moodusel on oma head ja vead. Tuleb vastavalt olukorrale kaaluda, mis on parim.

Kuularite lisamine

Esiteks on meil vaja lisada kuularid, mis kuulavad hiireklahvi vajutust ja selle lahtilaskmist. Peaks salvestama ka *boolean* väärtuse, et kas hiir on sellel hetkel alla vajutatud või mitte. Niiviisi on meil võimalik kuulari *callback* meetodiks kasutada ühte funktsiooni.

```
//Kas hiireklahv on all või mitte
APP.isMouseDown = false;

//Kui hiir vajutatakse alla
document.addEventListener("mousedown", mouseClickHandler, false);

//Kui hiir lastakse lahti
document.addEventListener("mouseup", mouseClickHandler, false);
```

Järgnevalt on meil vaja see meetod ka luua.

```
function mouseClickHandler() {
    APP.isMouseDown = !APP.isMouseDown;

    if(APP.isMouseDown)
        document.addEventListener("mousemove", mouseMove, false);
    else
        document.removeEventListener("mousemove", mouseMove, false);
}
```

Meetod on lihtne. Iga kord kui hiir vajutatakse alla või lastakse lahti, käivitatakse meetod **mouseClickHandler**. Kui hiir on parasjagu all, siis lisatakse kuular **mouseMove()**, mis kuulab meie hiire liigutusi. Kui hiir lastakse lahti eemaldatakse kuular.

Järgmiseks on meil vaja kuulata hiire liikumist.

```
//Hiire allhoidmisel ja liigutamisel käivitub funktsioon
function mouseMove(e) {
    var x = e.webkitMovementX || e.mozMovementX;
    var y = e.webkitMovementY || e.mozMovementY;

    if(typeof x === "undefined")
        x = 0;
    if(typeof y === "undefined")
        y = 0;



    APP.cameraX += x * 0.002;
    APP.cameraY += y * 0.002;

    restrictCameraY();
    toCanonical();

    updateCamera();
}
```

Meetodi **webkitMovementX** abil on meil võimalik saada informatsioon, palju hiir selle liigutusega oma algsest positsioonist liikus. Hetkel korrutame saadud tulemuse ka juhusliku arvuga läbi, et hiir liiguks mingi soovitud kiirusega.

On vaja kuulata ka kiire rullikut. Kuna erinevates veebilehitsejates on see kuular erineva nime all kuulame neid kõiki.

```
document.addEventListener("mousewheel", mouseScrollHandler, false);
document.addEventListener("DOMMouseScroll", mouseScrollHandler, false);
document.addEventListener("onmousewheel", mouseScrollHandler, false);
```

Meetod, mis kuulab meie hiirerullikut on järgmine.

```
function mouseScrollHandler(e) {
    var delta = 0;

    if(!e)
        e = window.event;

    if(e.wheelDelta)                    /** Internet Explorer/Opera/Google Chrome **/
        delta = e.wheelDelta / 120;

    else if(e.detail)                   /** Mozilla Firefox **/
        delta = -e.detail/3;

    if(delta) {
        if(delta > 0 && APP.radius > APP.MIN_RADIUS)
            APP.radius -= APP.ZOOM_VALUE;
        else if(delta < 0)
            APP.radius += APP.ZOOM_VALUE;
    }

        if(e.preventDefault)
            e.preventDefault();
        e.returnValue = false;

    toCanonical();
    updateCamera();
}
```

Tänu Mozilla Firefoxile, mille sündmuses ei kasutata **wheelDelta** muutujat, vaid **detail** on meil vaja käsitleda mõlemat juhtumit. Lisaks tagastab Firefox muutuja ka vastasmärgiliselt. Kui meil on rullikult saadud väärtus viidud vahemikku [-1, 1] saame vastavalt väärtusele raadiust suurendada või vähendada. Seejärel võime**kaameramaatriksit** uuendada.

Kaamera liigutamine objekti orbiidil

Kuna me kasutame Euleri nurkasid, siis meil on vaja see viia ka kanoonilisele kujule. Selleks kasutame meetodit **toCanonical()** (algoritm raamatust "3D Math Primer for Graphics and

Game Development"). Kanooniline vorm tähendab seda, et koordinaadid sfäärilises süsteemis on ainulaadsed. Meil on täidetud järgnevad tingimused:

```
r ≥ 0
-180 kraadi < cameraX ≤ 180 kraadi
-90 kraadi < cameraY ≤ 90 kraadi
raadius = 0 => cameraX = cameraY = 0
cameraY = ligikaudu 90 kraadi => cameraX = 0
```

Kaamera liigutamiseks saab kasutada ka maatrikseid või siis kvaternioone, mis on tegelikult tunduvalt paremad. Kvaternioonide implementeerimine on raskem ja koodi tuleks korralikult juurde kirjutada. Euleri nurgad on meile intuitiivsemad, mistõttu kasutame Euleri nurkasid. Euleri nurkade probleemiks on *Gimbal lock*.
Kaamera arvutamiseks kasutame meetodit **updateCamera()**. Arvutamine toimub järgmiselt:

1. Leiame kaamera asukoha objektist (Kasutame seda valemit ja vektorite liitmist).
2. Leiame suunavektori kaamerast punkti: **sihtpunkt - asukoht**.
3. Liidame kaamera asukoha ja suunavektori ning saame punkti, mida kaamera vaatab.
4. Leiame x-telje vektori kasutades polaarkoordinaatsüsteemi.
5. Järgnevalt saame vektorite ristkorrutisega leida vektori, mis defineerib kaamera üles vaatamise suuna.

See lahendus sobib hästi FPS (läbi silmade vaadates) kaameraks. Meil on aga probleem. Kui me satume põhja- või lõunapoolusele läheb kaamera katki. Lahendada probleem, kus kaamera ei lähe hulluks on suhteliselt lihtne. Määrame, mis piiridest ei tohi meie vertikaalnurk välja minna. Lahendus sobiks ka kasutajale, sest jääb alles taju, kus asub objekti alumine ja ülemine osa.

```
//Maksimaalne vertikaalnurk
APP.MAX_VERTICAL = APP.PIOVERTWO - APP.PIOVERTWO / 8;

function restrictCameraY() {
    if(Math.abs(APP.cameraY) > APP.MAX_VERTICAL) {
        if(APP.cameraY < 0)
            APP.cameraY = -APP.MAX_VERTICAL;
        else
            APP.cameraY = APP.MAX_VERTICAL;
    }
}
```

Esiteks tuleb määrata piirnurk, millest üle ei saa liikuda. Võtame mingi juhusliku nurga ~80 kraadi. Kui nüüd nurk on läinud üle piirnurga määrame talle meie poolt defineeritud maksimaalse nurga.

VOILA! Kaamerat on võimalik juhtida.

Tulemus

Kaameraga saame vaadata objekti igast küljest. Kui kasutada rullikut ja objektile lähedale minna, on näha, et osa objektist pügatakse ära, sest kolmnurgad ei asetse tüvipüramiidis.

Joonis 1 Pügatud kolmnurk

Ülesanne

1. Eemalda piirid, milles kaamera võib liikuda ja vaata, mis juhtub.
2. Võimalda kasutajal objekti suurust määrata, vajutades HTML nuppu või mõnda muud elementi.
3. Muuda kaamerat nii, et saaks maailmas vabalt ringi liikuda.
4. Uuri, mis on *Gimbal lock* ja mis juhtus Apollo 11'ga!

2.10. 07 Renderdamine tekstuurile

Informatsioon

Apache: *builds/lesson07*

Lähtekoodi asukoht: lessons/lesson07

Varjundajate asukoht: shaders/lesson07

NodeJS käivituskäsk:

```
npm run lesson07
```

Renderdamine tekstuurile

Kasutame *WEBGL_depth_texture* laiendust. Kui leiad, et veebilehitseja ei toeta seda laiendust, ürita kasutada Chrome veebilehitsejat. Võimalik on sisse lülitada ka kõik eksperimentaalsed WebGL laiendused. Liigu aadressile *chrome://flags/#enable-webgl-draft-extensions* ja aktiveeri laiendused.
Loome rakenduse, kus kasutame tekstuuri renderdamiseks sama stseeni, kuid erinevaid maatrikseid. Peastseenis on võimalik kaameraga liikuda. Stseenis, mis renderdatakse tekstuurile on kaamera paigal ja liigub objekt ümber oma x-telje.

Järgnevalt räägime, kuidas renderdada tekstuurile. Meil on selge, et pikslivarjundajast kirjutatakse väärtused kaadripuhvrisse. Kaadripuhver sisaldab erinevaid puhvreid, mis hoiavad näiteks piksli värvust ja piksli sügavust. Kui me ise kontekstiga mingit kaadripuhvrit ei seo, siis renderdamise järjekorras kirjutatakse väärtused vaikimisi määratud kaadripuhvrisse. Seda puhvrit kasutades toimubki *Canvas* elemendile renderdamine.

Valmis rakenduse **tsükkel** toimub järgmiselt:

1. Uuendame mudelmaatriksit, mida kasutame renderdamisel tekstuurile
2. Kui liigutatakse hiirt uuendame kaameramaatriksit, mida kasutatakse stseeni renderdamiseks *canvas* elemendile.
3. Renderdame tekstuurile.
4. Renderdame stseeni *canvas* elemendile, kus kasutame objekti tekstuurina eelpool renderdatud tekstuuri.

Meil on võimalus luua ka mingi muu kaadripuhver, kuhu me saaksime renderdada. Esiteks kirjutame kaadripuhvrisse värvuse, mistõttu on meil vaja kaadripuhvriga siduda tekstuur. Ainult tekstuurist ei piisa, sest otsus, kas värvus kirjutatakse puhvrisse või mitte, sõltub sellest, kas piksli sügavus on väiksem kui eelmise piksli oma puhvris. Vaja on luua sügavuspuhver, kuhu värvi sügavus paigutada. Kui värvus- ja sügavuspuhver on kaadripuhvriga seotud on meil võimalik seda renderdamisel kasutada. Enne renderdamist tuleb muidugi kaadripuhver kontekstiga siduda.

Kaadripuhvri initsialiseerimine

Kaadripuhvri loomine toimub meetodis **setupFrameBuffer()**.

Esiteks tuleb meil kaadripuhver luua. Me määrame ära ka kaadripuhvri suuruse. Enne renderdamist kutsume välja **viewport()** meetodi, et renderdamisel õiged mõõtmed oleks.

```
//Loome kaadripuhvri, kuhu saame renderdamise järjekorras stseeni renderdada.
APP.frameBuffer = GL.createFramebuffer();
GL.bindFramebuffer(GL.FRAMEBUFFER, APP.frameBuffer);
APP.frameBuffer.width = 512;
APP.frameBuffer.height = 512;
```

Järgmiseks loome tekstuuri, kuhu salvestame värvid. Kui me sellele tekstuurile renderdame, on meil pärast võimalik seda kasutada mingi muu objekti peal. Niiviisi on objekti peal liikuv pilt (ekraan). Seekord kasutame tekstuuril *mipmapping tehnikat*, mis parandab kvaliteeti.

```
//Loome värvuspuhvri, mis hoiab piksleid
APP.FBColorTexture = GL.createTexture();
GL.bindTexture(GL.TEXTURE_2D, APP.FBColorTexture);
GL.texImage2D(GL.TEXTURE_2D, 0, GL.RGBA, APP.frameBuffer.width, APP.frameBuffer.height, 0,
GL.RGBA, GL.UNSIGNED_BYTE, null);
GL.texParameteri(GL.TEXTURE_2D, GL.TEXTURE_MAG_FILTER, GL.LINEAR);
GL.texParameteri(GL.TEXTURE_2D, GL.TEXTURE_MIN_FILTER, GL.LINEAR_MIPMAP_NEAREST);
GL.generateMipmap(GL.TEXTURE_2D);
```

Vaja on luua ka sügavuspuhver, milleks kasutame renderdamispuhvrit. Seda puhvrit on võimalik siduda vaid kaadripuhvriga. Sinna ei ole võimalik tavalisel moel andmeid saata. Sügavuspuhvris olevad arvud peavad olema võimelised säilitima rohkem informatsiooni, mistõttu kasutame *GL_DEPTH_COMPONENT16* tüüpi.

```
//Loome sügavuspuhvri, mis hoiab pikslite sügavusi
APP.FBDepthBuffer = GL.createRenderbuffer();
GL.bindRenderbuffer(GL.RENDERBUFFER, APP.FBDepthBuffer);
```

```
GL.renderbufferStorage(GL.RENDERBUFFER,     GL.DEPTH_COMPONENT16,     APP.frameBuffer.width,
APP.frameBuffer.height);
```

Lõpetuseks seome loodud hoidlad kaadripuhvriga. Hea tava on ka kõik kasutatud puhvrid kontekstist lahtu siduda, kui nendega on asjatoimetused tehtud.

```
//Seome värvi- ja sügavuspuhvri kaadripuhvriga
GL.framebufferTexture2D(GL.FRAMEBUFFER,         GL.COLOR_ATTACHMENT0,         GL.TEXTURE_2D,
APP.FBColorTexture, 0);
GL.framebufferRenderbuffer(GL.FRAMEBUFFER,       GL.DEPTH_ATTACHMENT,       GL.RENDERBUFFER,
APP.FBDepthBuffer);

GL.bindTexture(GL.TEXTURE_2D, null);
GL.bindRenderbuffer(GL.RENDERBUFFER, null);
GL.bindFramebuffer(GL.FRAMEBUFFER, null);
```

Uued maatriksid

Tekstuuri renderdamiseks kasutame uusi maatrikseid. Vaja on samamoodi defineerida kolm maatriksit. **Mudelmaatriksit** transformeerime igal tsükli sammul. **Kaameramaatriks** ja **projektsioonmaatriks** ei muutu ja nad jäävad staatiliseks.

```
//Mudelmaatriks, mida kasutame tekstuurile renderdamiseks
APP.textureModelMatrix = mat4.create();

//Kaameramaatriks, mida kasutame tekstuurile renderdamiseks
APP.textureViewMatrix = mat4.create();
mat4.lookAt(APP.textureViewMatrix, [0, 0, 0], [0, 0, -5], [0, 1, 0]);

//Projektsioonimaatriks, mida kasutame tekstuurile renderdamiseks
APP.textureProjectionMatrix = mat4.create();
 mat4.perspective(APP.textureProjectionMatrix, 45.0, 1, 0.1, 100.0);
```

Kuidas näeb välja tsükkel?

Tsükkel toimub nagu ikka **loop()** meetodis. Alguses renderdame tekstuurile ja seejärel *canvas* elemendile.

```
//Kutsutakse välja Looper objektis iga kaader
function loop(deltaTime) {
    update(deltaTime);

    //Määrame kaadripuhvriks meie enda loodud kaadripuhvri
```

```
    GL.bindFramebuffer(GL.FRAMEBUFFER, APP.frameBuffer);

    //Renderdame stseeni tekstuurile
    renderToTexture();

    //Seome lahti eelmise kaadripuhvri. Pärast seda on kasutusel tavaline puhver, mida kasutatakse canvas elemendi jaoks.
    GL.bindFramebuffer(GL.FRAMEBUFFER, null);

    render();
}
```

Tekstuuril kasutatava mudelmaatriksi uuendamine

Tahame, et igal tsükli sammul keerleks meie objekt tekstuuril ning seetõttu on meil vaja muuta **updateObject()** meetodit.

```
//uuendame objekti
function updateObject() {
    mat4.rotateX(APP.textureModelMatrix, APP.textureModelMatrix, 0.005);
}
```

Tekstuuri renderdamine

Tekstuuri renderdame meetodis **renderToTexture()**. See meetod on väga sarnane eelmise tunni omale. Esiteks on meil vaja igakord enne renderdamist määrata õige vaateakna suurus.

```
//Määrame õige vaateakna suuruse
GL.viewport(0, 0, APP.frameBuffer.width, APP.frameBuffer.height);
```

Tekstuurile renderdamiseks kasutame teisi maatrikseid, mis tuleb ühtsete muutujatena varjundajatesse saata.

```
//Saadame meie tekstuuri maatriksid ka varjundajasse
GL.uniformMatrix4fv(APP.u_ModelMatrix, false, APP.textureModelMatrix);
GL.uniformMatrix4fv(APP.u_ViewMatrix, false, APP.textureViewMatrix);
GL.uniformMatrix4fv(APP.u_ProjectionMatrix, false, APP.textureProjectionMatrix);
```

Kõige suurem muutus on kindlasti see, et pärast renderdamist seome tekstuuri, kuhu renderdasime, kontekstiga ja loome sellest tekstuurist väiksemad versioonid.

```
GL.bindTexture(GL.TEXTURE_2D, APP.FBColorTexture);
```

```
GL.generateMipmap(GL.TEXTURE_2D);
GL.bindTexture(GL.TEXTURE_2D, null);
```

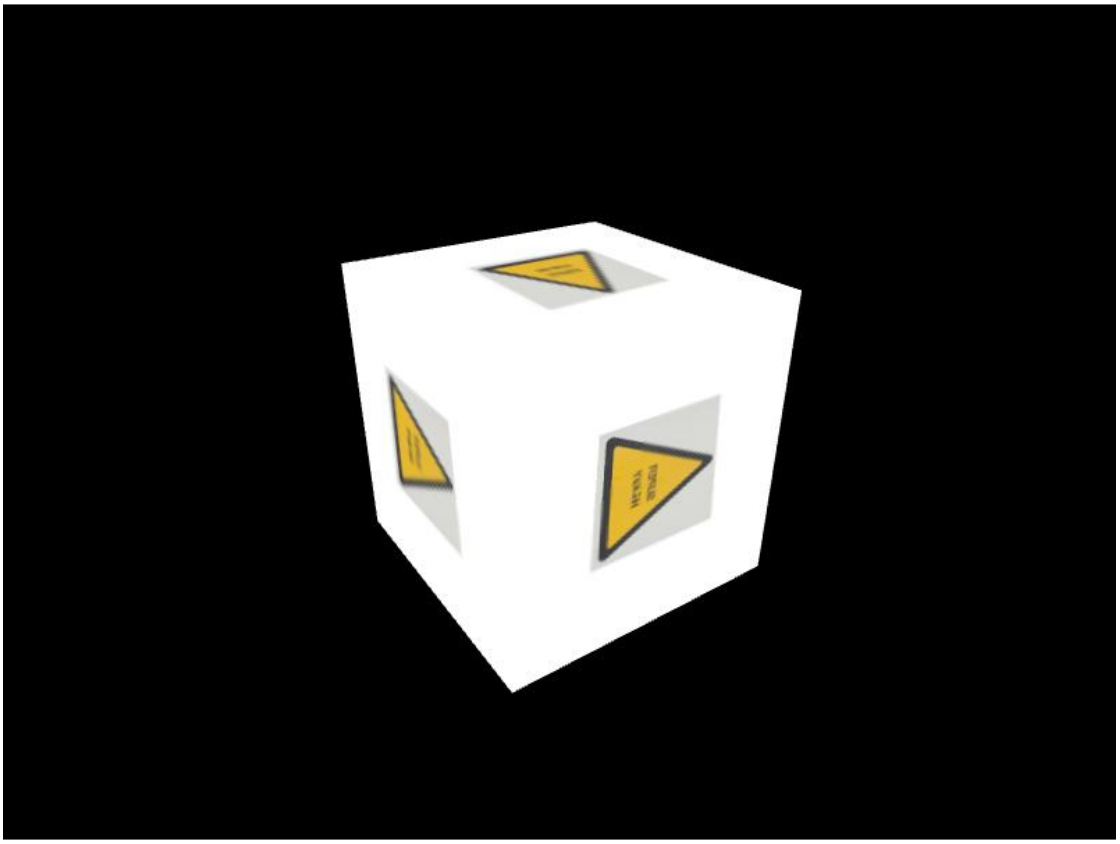
Stseeni renderdamine

Stseeni renderdamine toimub loomulikult **render()** meetodis. Ainuke asi, mis on muutunud eelmise tunniga on see, et objekti peal ei kasuta pilti, vaid tekstuuri, mille eelpool renderdasime. Seega tuleb vahetada vaid muutuja.

```
//Aktiveerime ja määrame tekstuuri
GL.activeTexture(GL.TEXTURE0);
GL.bindTexture(GL.TEXTURE_2D, APP.FBColorTexture);
GL.uniform1i(APP.u_Texture, 0);
```

Tulemus

Renderdades ühe stseeni tekstuurile saame seda tekstuuri kasutada teise objekti peal. Me kasutasime tekstuuri, et sinna stseen renderdada, kuid tekstuuri võib kasutada ükskõik, mis andmete salvestamiseks.



Joonis 1 Renderdamine tekstuurile

Ülesanded

1. Kasuta tekstuuri renderdamisel enda loodud püramiidi.
2. Renderda tekstuurile hoopis kuup ja peastseenis kasutada püramiidi.
3. Loo uued objektid ja uus stseen. Renderda see stseen tunnis olevale kuubile.

2.11. 08 Valgus

Selles osas lisame rakendusele valguse. Alustuseks uuri valgust ja valgustusmudelit. Mudel, mida kasutame on nö baasmudel. Kuidas me mudelit implementeerime see on meie enda teha. Emiteerivat komponenti me ei kasuta ja olgu tekstuur see, mis värvi "emiteerib".

Kasutame kahte valgust: esimest kasutame stseeni renderdamisel, teist tekstuuri renderdamiseks. Mõlemas renderdamisetapis kasutame sama materjali.

Valgusallikate ja materjali loomine

Valgusallikad ja materjali loome nagu kord ja kohus meetodis **setup()**.

```
//Valgusallikas, mida kasutame stseeni renderdamisel
APP.directionalLight = {
    "color": new Float32Array([1.0, 1.0, 1.0]),
    "direction": new Float32Array([-1.0, -1.0, 1.0])
};

//Valgusallikas, mida kasutame tekstuuri renderdamisel
APP.textureDirectionalLight = {
    "color": new Float32Array([1.0, 1.0, 0.0]),
    "direction": new Float32Array([1.0, 0.0, 0.0])
};

//Materjal, mida kasutame mõlemas renderdamisetapis
APP.material = {
    "ambientColor": new Float32Array([0.3, 0.3, 0.3]),
    "diffuseColor": new Float32Array([0.5, 0.5, 0.5]),
    "specularColor": new Float32Array([0.7, 0.7, 0.7]),
    "shininess": 128.0
};
```

Valgusallikana kasutame suunatud valgust (*directional light*). Seda valgust on lihtne implementeerida. Samadel põhimõtetel on võimalik pärast implementeerida punktvalgus (*point light*) või rambivalgus (*spotlight*).

Materjal

Materjali jaoks on meil vaja teada erinevate komponentide värvuseid ja läikefaktorit, millega saame muuta läikeraadiust. Suurendades faktorit läikeraadius väheneb, vähendades suureneb.

Suunatud valgus

Suunatud valgust kasutades on meil vaja kindlasti teada suunda, kuhu kiired suubuvad. Suunatud valgust on hea ette kujutada päikesena, mis asub meist nii kaugel, et kõiki kiiri võib vaadelda niiviisi, et need langevad objektile paralleelselt. Määrame ka valguse värvuse, mida saame varjundajates kasutada.

Ühtsed muutujad

Valgusallikas ja materjal on vaja saata ka ühtsete muutujatena varjundajatesse, et seal siis vajalikke arvutusi teha. Seetõttu peame samamoodi küsima nende asukohad varjundajates.

```
//Valgusallika ühtsete muutujate asukohad
APP.u_DirectionalLight = {
    "color": GL.getUniformLocation(shaderProgram, "u_DirectionalLightColor"),
    "direction": GL.getUniformLocation(shaderProgram, "u_DirectionalLightDirection")
};

//Materjali ühtsete muutujate asukohad
APP.u_Material = {
    "ambientColor": GL.getUniformLocation(shaderProgram, "u_MaterialAmbientColor"),
    "diffuseColor": GL.getUniformLocation(shaderProgram, "u_MaterialDiffuseColor"),
    "specularColor": GL.getUniformLocation(shaderProgram, "u_MaterialSpecularColor"),
    "shininess": GL.getUniformLocation(shaderProgram, "u_MaterialShininess")
};
```

Saatmine programmi

Nii tekstuuri renderdamisel kui ka stseeni renderdamisel kasutame sama materjali. Võime kirjutada eraldi meetodi, mis mõlemas renderdamisetapis annaks programmi sama materjali.

```
//Määrame valgusarvutuste jaoks ühtsed muutujad
function setMaterialUniforms() {

    //Objekti materjali muutujad
    GL.uniform3fv(APP.u_Material.ambientColor, APP.material.ambientColor);
    GL.uniform3fv(APP.u_Material.diffuseColor, APP.material.diffuseColor);
    GL.uniform3fv(APP.u_Material.specularColor, APP.material.specularColor);
    GL.uniform1f(APP.u_Material.shininess, APP.material.shininess);
}
```

Meil on defineeritud kaks erinevat valgust. Ühte kasutame tekstuuri renderdamisel ja teist stseeni renderdamisel. Seetõttu on meil vaja vastavates renderdamismeetodites saata programmi erinevad valgused.

Tekstuuri renderdamisel meetodis renderToTexture()

```
//Valgusallika muutujad
GL.uniform3fv(APP.u_DirectionalLight.color, APP.textureDirectionalLight.color);
GL.uniform3fv(APP.u_DirectionalLight.direction, APP.textureDirectionalLight.direction);
```

Stseenile renderdamisel meetodis render()

```
//Valgusallika muutujad
GL.uniform3fv(APP.u_DirectionalLight.color, APP.directionalLight.color);
GL.uniform3fv(APP.u_DirectionalLight.direction, APP.directionalLight.direction);
```

Valguse arvutamine

Praeguseks hetkeks on meil kõik vajalikud muutujad saadetud. Võime liikuda varjundajate juurde. Valgusarvutusi teostame pikslivarjundajas. Arvutamine toimub kaameraruumis, kus silma asukoht on koordinaatidel **(0, 0, 0)**.

Meie arvutamine toimub kaameraruumis ja seega peame tipuvarjundajas viima tipu, normaalvektori kaameraruumi.

Valgusarvutusi on võimalik teha ka ükskõik, mis juhuslikus ruumis. Võimalik on edukalt teostada arvutusi ka maailmaruumis. Meeles peab pidama, et arvutused toimuksid samas ruumis.

Normaalvektorid tuleb viia kaameraruumi tegelikult spetsiaalse maatriksiga, kuid hetkel me seda ei kasuta. Vaata http://www.lighthouse3d.com/tutorials/glsl-tutorial/the-normal-matrix/.

Tipuvarjundaja

Tipuvarjundajas tuleb meil tipp ja normaalvektor viia õigesse ruumi. Seda teeme samamoodi maatriksitega korrutades.

```
...

varying vec3 v_PositionCameraSpace;
varying vec3 v_NormalCameraSpace;
```

```
void main(void){

 ...

    //Viime normaalvektor kaameraruumi
    v_NormalCameraSpace = (u_ViewMatrix * u_ModelMatrix * vec4(a_Normal, 0.0)).xyz;

    //Viime positsiooni kaameraruumi
    v_PositionCameraSpace = (u_ViewMatrix * u_ModelMatrix * vec4(a_Position, 1.0)).xyz;
}
```

Pikslivarjundaja

Pikslivarjundajas toimub teoorias väljatoodud valgusmudeli implementeerimine. Esiteks on meil vaja leida valguse suund kaameraruumis ja nagu ikka teostame maatriksite korrutamise.

```
//Leiame valguse suuna kaameraruumis, sest meil peavad kõik toimuma samas ruumis.
vec3 lightDirectionCameraSpace = (u_ViewMatrix * vec4(u_DirectionalLightDirection, 0.0)).xyz;
```

Pea meeles! Enne kui suunavektorit kasutama hakkame, on see alati vaja normaliseerida.

```
//Suunavektor alati normaliseerida
vec3 lightDirectionNormal = normalize(lightDirectionCameraSpace);
```

Hajusfaktori leidimine

Hajusfaktori abil saame määrata hajusvärvuse (*diffuseColor*). Seda faktorit kasutades on tasapind, kuhu kiired langevad tervama nurga all eredam.

Hajusfaktori (*diffuseFactor*) leiame järgnevalt:

```
//Leiame nurga kiire ja normaalvektori vahel, et teada palju see punkt valgust saab.
float diffuseFactor = clamp(dot(v_NormalCameraSpace, -lightDirectionNormal), 0.0, 1.0);
```

- **dot** - skalaarkorrutis, et leida nurk normaalvektori ja valguse suuna vahel
- **clamp()** - võimaldab skalaarkorrutisest saadud tulemuse viia vahemikku [0, 1]

Läikefaktori leidmine

Läikefaktori leidmiseks on vaja aru saada, kus ruumis me paikneme. Me oleme kaameraruumis. Läikefaktori leidmiseks on meil vaja suunavektorit silma. Kaameraruumis asudes on silma koordinaadid (0, 0, 0), seega on meie vektor järgmine:

```
//Leiame suunavektori silma. Kuna kõik punktid on kaameraruumis, siis on see lihtsalt vastasuunaline punkti vektorile
vec3 vertexToEye = normalize(-v_PositionCameraSpace);
```

Järgnevalt on meil vaja arvutada kiire peegeldus. Selleks kasutame juba olemasolevat meetodit **reflect()**.

```
//Leiame peegelduse
vec3 lightReflection = normalize(reflect(lightDirectionNormal, v_NormalCameraSpace));
```

Läikefaktori arvutamiseks on meil vaja järgnevalt leida nurk peegeldunud kiire ja silma suunavektori vahel. Tulemus on vaja materjali läikivusega astendada.

```
//Aeg on arvutada nurk kiire peegelduse ja meie silma vektori vahel. Clamp meetodiga viime ta piiridesse.
float clampedVertexDotReflection = clamp(dot(vertexToEye, lightReflection), 0.0, 1.0);

//Leiame peegeldusfaktori astendades selle läikega
float specularFactor = pow(clampedVertexDotReflection, u_MaterialShininess);
```

Lõppvärvus ja komponentide arvutamine

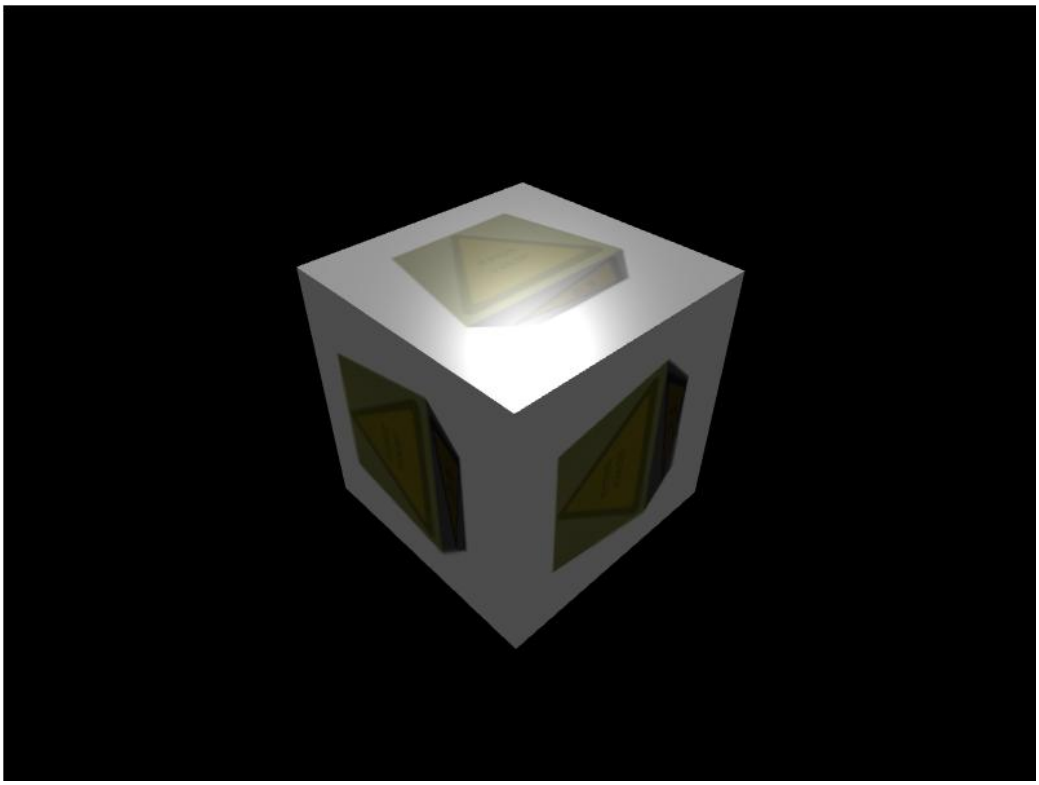
Pärast faktorite arvutamist on väga lihtne arvutada erinevad komponendid ja need kombineerides saada piksli värvus.

```
//Meie küllastunud valgus. Tekstuurist saadud värv koos materjali omadusega
vec4 ambientColor = color * vec4(u_MaterialAmbientColor, 1.0);

//Määrame hajusvalguse. Lisame juurde ka valguse värvuse, et ägedam oleks.
diffuseColor = vec4(u_DirectionalLightColor, 0.0) * vec4(u_MaterialDiffuseColor, 0.0) * diffuseFactor;

//Arvutame peegeldusvärvuse. Lisame ka valguse värvi, et ägedam oleks.
specularColor =  vec4(u_DirectionalLightColor, 0.0) * vec4(u_MaterialSpecularColor, 0.0) * specularFactor;

//Värv, mis kirjutatakse kaadripuhvrisse.
gl_FragColor = ambientColor + diffuseColor + specularColor;
```

Tulemus

Valguse simuleerimiseks teostame arvutusi varjundajates. Arvutused tipuvarundajas on mõistagi jõudlust kokkuhoidvad. Kui me renderdaks selle stseeni tekstuurile, oleks meil võimalik samale tekstuurile veel lisaks igasuguseid efekte lisada (*post processing*).

Kui meil on teada, et stseenis on liikumatud objektid ja valgusallikad, on võimalik jõudlust kokku hoida sellega, et arvutused tekstuurile salvestada (*light baking*).



Joonis 1 Valgus

Ülesanded

1. Vaheta valguse värve ja materjali omadusi ning jälgi muudatusi.
2. Lisa kasutajal võimalus materjali omadusi muuta.
3. Lisa kasutajal võimalus mõlema valguse omadusi muuta.
4. Mis ruumis toimub valguse arvutamine meie lähtekoodis?
5. Kui on teada, et arvutil ei ole jõudlust palju, siis mis varjundajas oleks parem arvutusi teostada?
6. Implementeeri punktvalgus (*point light*). Punktvalgust võib vaadelda lambina, mis kiirgab igas suunas.
7. Muuda valgusarvutust nii, et objekti kaugenedes valguse tugevus nõrgeneb.

2.12. Lisaülesanded

WebGL'i ja graafikaprogrammeerimise õppimiseks ei ole paremat moodust, kui iseseisvalt rakendust arendada. Nüüdseks peaks olema arusaamine, kuidas WebGL ja renderdamise järjekord töötab ning kuidas seda interaktiivseks muuta. Järgnevalt pakume ülesandeid, mida oleks hea lahendada, et saadud teadmised kindlustada ja uusi teadmisi selle käigus omandada.

Ülesannete lahendamisel võib alustada täiesti nullist või täiustada eelmiste praktikate käigus loodud koodi. Ei tasu meelte heita, kui midagi tööle ei saa. Kõik on selle läbi teinud. Ülesanded võivad olla ka väga aeganõudvad.

Arendada tasub väga väikeste iteratsioonide kaupa.

1. Implementeeri kaamera, mis liigub mööda defineeritud rada.
2. Võimalda rakendusel sisse laadida modelleerimistarkvaraga tehtud objekt või objekte. Soovitav kasutada näiteks fbx-conv konverterit, mis võimaldab meil andmed *JSON* formaati viia, mida oleks võimalik uurida. *Vihje: fbx-conv-win32.exe -f mudel.fbx konverteeritud_mudel.g3dj*.
3. Kui sa ei ole veel seda teinud, siis implementeerida üks kahest valgusallikas:*pointlight*, *spotlight*. *Vihje: http://www.lighthouse3d.com/tutorials/glsl-core-tutorial/point-lights/*
4. Võimalda kasutajal kuubikuid stseeni lisada ja neid liigutada. *Vihje: tipuandmed jäävad täpselt samaks nagu praktikas tehtud kuubil. Igal kuubikul on oma mudelmaatriks, mida saab liigutada. Tehnika raypicking kasutamine on eriti hea*.
5. Loo rakendus, kus kasutajal on võimalik saata pilt rakendusse. Pilti näidatakse ekraanil läbi mingisuguse filtri. *Vihje: loo filter pikslivarjundajas. Implementeeri näiteks Gaussian Blur Filter Shader*

Alternatiivne materjal ja õppimise jätkamine

Meie käsitlesime teemasid väga pinnapealselt. Kui teema väga huvitab ja on soov edasi areneda või meie loodud materjal ei olnud sobilik, toome välja materjali, mida peaks kindlasti uurima. Materjal on kõik inglise keeles.

Mõned väga paljudest raamatutest, mis tasub kätte võtta:

- Dunn, F., and Parberry, I. (2011). *3D Math Primer for Graphics and Game Development, 2nd Edition*. A.K. Peters / CRC Press.
- Sellers, G., Wright, S. R., Haemel, N. (2013). *OpenGL SuperBible Sixth Edition*. Crawfordsville: RR Donnelley
- Akenine-Moller, T., Haines, E., and Hoffman, N. (2008). *Real-Time Rendering, 3rd Edition*. A.K. Peters.
- Engel, W. (ed.) (2006). *Shader X 4: Advanced Rendering Techniques*. Charles River Media.

Loomulikult on ka internetis saadaval väga palju head materjal:

- Algajale - http://learningwebgl.com/
- Algajale (süvitsi) - http://www.arcsynthesis.org/gltut/
- Algajale - http://http.developer.nvidia.com/CgTutorial/cg_tutorial_chapter01.html
- Algajale ja edasijõudnule - http://www.learnopengl.com
- Algajale ja edasijõudnule - http://www.opengl-tutorial.org/
- Algajale ja edasijõudnule - http://ogldev.atspace.co.uk/index.html
- Edasijõudnutele - http://developer.nvidia.com/GPUGems/gpugems_part01.html
- Abimaterjal - http://www.lighthouse3d.com/

Ülesannete vastused

Maatriks

1. Selle maatriksi abil on võimalik objekt liigutada/transleerida uutele koordinaatidele. Maatriks liigutab objekti 10 ühiku võrra mööda positiivset x-telge ja 5 ühiku võrra mööda positiivset y-telge.
2. Selle maatriksi abil on võimalik objekti mõõtkava suurendada. Maatriks suurendab objekti kaks korda mööda x- ja y-telge.
3. Kombineeritud maatriks on järgmine:

```
2  0  0  10
0  1  0  5
0  0  2  0
0  0  0  0
```

8. Kui vastasid jah, siis palun loe materjal uuesti üle. Loe ka lisamaterjali!
9. Liigu järgnevates suundades:
    - http://www.arcsynthesis.org/gltut/Positioning/Tutorial%2006.html
    - http://www.in.tum.de/fileadmin/user_upload/Lehrstuehle/Lehrstuhl_XV/Teaching/SS07/Praktikum/MatricesTips.pdf
    - http://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/
    - 3D Math Primer for Graphics and Game Development (2nd Edition) Chapter 4 - Chapter 6.

Polaarne ja sfääriline koordinaatsüsteem

1. x = 1.0; y = 1.732
2. (5, 90°)
3. x = 3.535; y = 3.535; z = 0
4. (7.071, 38.659°, 64.895°)

Valgus ja valgustusmudel

1. Läikefaktor on 0.
2. Hajuvusfaktor on 0, sellepärast et kiired ei lange tasapinnal asuva punkti pinnale.
3. Mõlema läikefaktor on sama. Kiired on 180° pööratud ja langevad erinevast suunast, kuid normaalvektori suhtes langevad nad sama nurga all.

00 - Esimene kolmnurk

1. Pikslivarjundaja:

```
precision mediump float;

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

2. Tipuandmed ja renderdamiskäsk:

```
  //Tippude andmed, mis moodustavad ühe kolmnurga
  var myVerticesData = [
      -1.0,  -1.0,  0.0,
      1.0,  -1.0,  0.0,
      1.0,  1.0,  0.0,
      -1.0, -1.0, 0.0,
      -1.0,  1.0,  0.0,
      1.0,  1.0,  0.0
  ];

  GL.drawArrays(GL.TRIANGLES, 0, 6);
```

See vastus on üks võimalikest lahendustest.

02 - Indeksite kasutamine

1. Andmed on kompaktsemad.
2. Kaks kolmnurka.

03 - Maatriksid

1. mat4.rotateZ(modelMatrix, modelMatrix, Math.PI/4);
2. var cameraAt = [0, 0, -5];

04 - Liikumine

1. Vihje: Leia koht, kus toimub sisendite kuulamine või muuda kaameramaatriksi genereerimist.
2. Vihje: Meetod updateObject().
3. Vihje: Pead kasutama uut mudelmaatriksit. Renderdad nagu me teeme seda render() meetodis. Sul on vaja liikumiseks ka mudelmaatriksit uuendada.

05 - Tekstuur ja 3D objekt

1. Muuda ja vaata!
2. Muuda ja vaata!
3. Ei tööta? Tekstuuri mõõtmed peavad olema 2. astmed.
4. Vaata, kuidas kuup on loodud.
5. Vaja on sisse laadida mitu tekstuuri. Kõik on vaja aktiveerida ja ühtsete muutujatena varjundajatesse saata. Üks võimalus on luua uus tipuatribuut, mille väärtuse abil on võimalik määrata kolmnurgas kasutatav tekstuur. Ole leidlik!

06 - Hiir

1. Muuda ja vaata!
2. Vaja on mingile elemendile, näiteks *button* määrata kuular ja seda kuulata. Vajutades muutub mudelmaatriks. Vaata ka glMatrix dokumentatsiooni.
3. Kaamera ei sõltu enam objektist ja raadiusest.
4. Uuri kindlasti!

07 - Renderdamine tekstuurile

1. Loo püramiid ja vaata lähtekoodi. Jõudu!
2. Vaata lähtekoodi. Jõudu!
3. Vaata lähtekoodi. Jõudu!

08 - Valgus

1. Muuda ja vaata!
2. Kasuta lihtsalt elementi *input* või tee hoopiski kerimisriba. Elemendile on vaja lisada kuularid. Vaja on muuta muutujat *APP.material*.
3. Vaja on muuta muutujaid *APP.directionalLight* ja *APP.textureDirectionalLight*.
4. Kaameraruumis, sest seal ruumis asetseb silm koordinaatidel (0, 0, 0). Keegi ei keela ka mujal ruumis arvutusi teostada.
5. Tipuvarjundajas, sest iga kolmnurga kohta on ainult kolm tippu.
6. Punktvalgusel on defineeritud maailmaruumis olevad koordinaadid. Koordinaadid on vaja viia kaameraruumi. Kaameraruumis on vaja leida suunavektor valguse ja tipu vahel. Pärast seda toimub arvutamine nagu suunatud valguse puhul.
7. Võib kasutada näiteks järgmiseid valemeid:

$$\text{Tugevus} = \frac{1}{\text{Kaugus}^2}$$

või

$$\text{Tugevus} = \frac{1}{\text{Konstant faktor} * \text{Lineaarne faktor} * \text{Eksponentsiaalne faktor} * \text{Kaugus}^2}$$

gl_FragColor = Värvus * Tugevus

Kasutatud kirjandus

Khronos Group. (1. jaanuar 2015). *Rendering Pipeline Overview*. Kasutamise kuupäev 22. jaanuar 2015.a., allikas https://www.opengl.org/wiki/Rendering_Pipeline_Overview

Joe Groff. (5. aprill 2010). *An intro to modern OpenGL. Chapter 1: The Graphics Pipeline*. Kasutamise kuupäev 22. jaanuar 2015.a., allikas http://duriansoftware.com/joe/An-intro-to-modern-OpenGL.-Chapter-1:-The-Graphics-Pipeline.html

Khronos Group. (2011). *WebGL 1.0 API Quick Reference Card*. Kasutamise kuupäev 25. jaanuar 2015.a., allikas https://www.khronos.org/files/webgl/webgl-reference-card-1_0.pdf

Khronos Group. (12. mai 2009). *The OpenGL ES Shading Language 1.0 Specification*. Kasutamise kuupäev 25. jaanuar 2015.a., allikas https://www.khronos.org/files/opengles_shading_language.pdf

Khronos Group. (16. jaanuar 2015). *Vertex Shader*. Kasutamise kuupäev 25. jaanuar 2015.a., allikas https://www.opengl.org/wiki/Vertex_Shader

Khronos Group. (12. jaanuar 2015). *Vertex Specification*. Kasutamise kuupäev 25. jaanuar 2015.a., allika s https://www.opengl.org/wiki/Vertex_Specification

Khronos Group. (7. august 2012). *Reference:activeTexture*. Kasutamise kuupäev 4. veebruar 2015.a., allikas https://www.khronos.org/webgl/wiki/Reference:activeTexture

Microsoft. *bufferData method*. Kasutamise kuupäev 4. veebruar 2015.a., allikashttps://msdn.microsoft.com/en-us/library/ie/dn302373(v=vs.85).aspx

Mozilla Developer Network. *HTMLCanvasElement.getContext()*. Kasutamise kuupäev 1. veebruar 2015.a., allikas https://developer.mozilla.org/en-US/docs/Web/API/HTMLCanvasElement.getContext

Mozilla Developer Network. *Getting started with WebGL*. Kasutamise kuupäev 6. veebruar 2015.a., allikas https://developer.mozilla.org/en-US/docs/Web/WebGL/Getting_started_with_WebGL

Sellers, G., Wright, S. R., Haemel, N. (2013) *OpenGL SuperBible Sixth Edition*. Crawfordsville: RR Donnelley

Dunn, F., Parberry, I. (2011). *3D Math Primer for Graphics and Game Development (2nd Edition)*. Boca Raton: CRC Press