Tuukka Hevosmaa

3D-NÄKYMÄN OHJELMOINTI OPENGL 4.5 -OHJELMOINTIRAJAPINNALLA

Opinnäytetyö
CENTRIA-AMMATTIKORKEAKOULU
Tietotekniikan koulutusohjelma
Syyskuu 2017

TIIVISTELMÄ OPINNÄYTETYÖSTÄ

Centria-ammattikorkeakoulu	**Aika** Syyskuu 2017	**Tekijä/tekijät** Tuukka Hevosmaa
Koulutusohjelma Tietotekniikka		
Työn nimi 3D-NÄKYMÄN OHJELMOINTI OPENGL 4.5 -OHJELMOINTIRAJAPINNALLA		
Työn ohjaaja Kauko Kolehmainen		**Sivumäärä** 31 + 1
Työelämäohjaaja		

Tämä opinnäytetyö tutki OpenGL 4.5 -ohjelmointirajapinnan toimintoja, joiden perusteella tuotettiin ja esiteltiin niitä havainnollistava tietokonesovellus. Työ oli omavalintainen ja sen tavoitteena oli kasvattaa kirjoittajan grafiikkaohjelmointitaitoja sekä tuottaa pohjasovellus mahdollisia tulevaisuuden projekteja varten.

Tietoperusta koostui pääpiirteissään OpenGL:n esittelystä, käytettyjen työkalujen esittelystä, grafiikkaohjelmoinnin olennaisten osa-alueiden selvityksestä, tuotoksen esittelystä sekä työn johtopäätöksistä. Työ suoritettiin ohjelmoimalla tietoperustan pohjalta sen käytäntöön tuova sovellus.

Työn tutkimusongelmat selvitettiin tyydyttävästi ja johtopäätöksenä OpenGL oli moneen käyttötarkoitukseen soveltuva ohjelmointirajapinta.

Asiasanat
3D-mallinnus, C++, grafiikkaohjelmointi, OpenGL, renderöinti, tietokonegrafiikka

Centria
UNIVERSITY OF APPLIED SCIENCES

ABSTRACT

Centria University of Applied Sciences	**Date** September 2017	**Author** Tuukka Hevosmaa
Degree programme Information Technology		
Name of thesis PROGRAMMING A 3D SCENE WITH OPENGL 4.5 API		
Instructor Kauko Kolehmainen		**Pages** 31 + 1
Supervisor		

This thesis researched the functionalities of the OpenGL 4.5 application programming interface (API), based on which a computer program representing those functionalities was produced and presented. The thesis was self-chosen and its goal was to improve the author's graphics programming skills as well as produce a program which would form the basis for possible future projects.

The information base consisted broadly of introducing OpenGL and the utilized tools, clarifying the essential parts of graphics programming, presenting the produced program and the conclusions of the thesis. The thesis was implemented by applying the information base into practice by programming the software based on it.

The research problems of the thesis were answered in a satisfactory manner and as a conclusion OpenGL was an API suitable for a wide variety of applications.

Key words
3D modelling, C++, computer graphics, graphics programming, OpenGL, rendering

KÄSITTEIDEN MÄÄRITTELY

Ohjelmointirajapinta	Tiettyjen ohjelmien välisen kommunikoinnin mahdollistavat määritykset
Renderöinti	Grafiikan piirron vaiheet yhdistävä operaatio
Pikseli	Näyttölaitteen ruudun pienin elementti, joka muiden pikseleiden kanssa muodostaa ruudun kuvan
Sirpale	Tietyn pikselin piirtoon tarvittava data
Vektori	Tilassa olevaa sijaintia tai suuntaa esittävä arvo
Matriisi	Suorakulmainen arvojen joukko
Kuvauskanta	Tiedostojoukon metadataa säilövä tietorakenne
Linkitys	Kirjaston suoritettavaan tiedostoon yhdistävä operaatio
Koonti	Lähdekoodin kääntö konekieliseksi
Enumeraatio	Kokoelman elementtien järjestetty listaus
Normaali	Kappaleeseen kohtisuorassa oleva objekti

1 JOHDANTO

Tutkin tässä opinnäytetyössä 3D-tietokonegrafiikan periaatteita sekä OpenGL-ohjelmointirajapinnan toimintaa ja ominaisuuksia. Ensimmäisten lukujen esittelyjen jälkeen käytän oppimaani OpenGL-järjestelmän luontiin ja esittelen tuloksena saadun kolmiulotteisen sovelluksen sekä johtopäätökseni kirjaston soveltuvuudesta erilaisiin käyttötarkoituksiin. Työn tavoitteena on kasvattaa kirjoittajan ohjelmointitaitoja ja tuntemusta tietokonegrafiikan piirron menetelmistä sekä tutustuttaa lukija samoihin menetelmiin.

Valitsin tämän aiheen siksi, koska olen aina nauttinut videopelien grafiikka-asetusten säätämisestä ja harrastuksen myötä olen tarpeesta ja kiinnostuksesta tutkinut aiheen termien merkityksiä. Sellaisten käsitteiden kuten mm. resoluution, reunanpehmennyksen ja anisotrooppisen suodatuksen merkitysten selvittäminen on olennaista suorituskyvyn optimoinnin kannalta. Tämä selvitystyö on kasvattanut uteliaisuuttani grafiikkalaitteiston matalamman tason toimintaa kohden. Syvempi perehtyminen aiheeseen on saanut minut kiinnostumaan ohjelmointirajapintojen toiminnasta ja yrittämään grafiikkalaitteiston hallintaa. Opinnäytetyön valmistumisen myötä saan toivottavasti tarpeeksi valmiuksia työtehtäviin, joissa vaaditaan tietokonegrafiikan piirtomenetelmien tuntemusta.

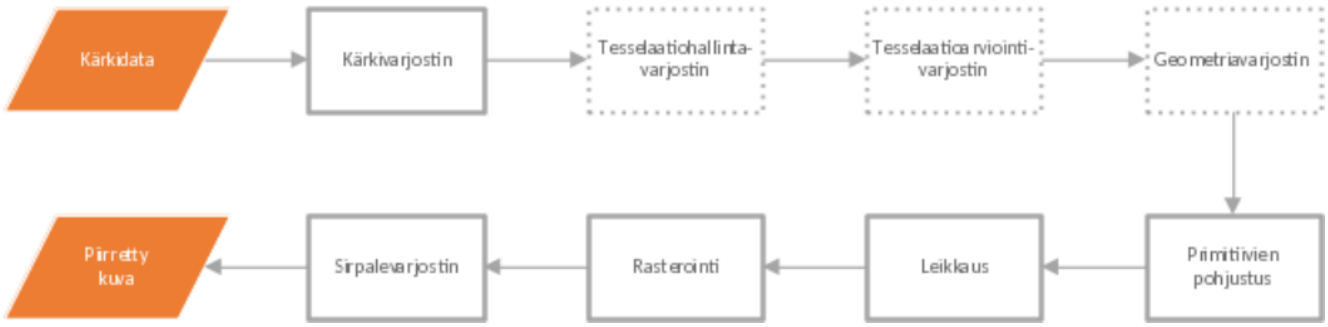
Tutkimusta aloittaessani kysyin itseltäni: mitä tietokonelaitteisto tekee perin pohjin alusta loppuun, kun siltä pyydetään jonkin sovelluksen toimesta mitä tahansa grafiikkaan sidottua toimintoa? Mikä mahdollistaa ohjelmoijan pääsyn näihin toimintoihin? Miten tietty rajapinta saadaan suorittamaan haluttuja operaatioita? Mitkä ovat ne oleelliset faktat, joita ohjelmoija käyttää hyväkseen toimivan graafisen sovelluksen luonnissa? Vastaamalla näihin tutkimuskysymyksiin saan selville kaikki ne seikat, jotka mahdollistavat teoriapohjan soveltamisen käytäntöön.

Kerron työssä aluksi perustietoa OpenGL:stä, sen toimintamenetelmistä ja tapahtumaketjusta. Tämä johtaa sovellusesimerkissä käyttämieni työkalujen, kuten kehitysympäristön ja muiden tarpeellisten kirjastojen esittelyyn. Kerron myös hieman käyttämästäni ohjelmointikielestä sekä versionhallintajärjestelmästä ja perusteluni niiden valinnoille. Seuraava pääluku kattaa yksityiskohtaisesti kaikki olennaiset 3D-malliesimerkin tekemiseen tarvitsemani tiedot sovelluksen eri tasojen toteutuksesta, kuten esimerkiksi virtuaalisen kappaleen mallin ja tekstuurin määrittämisen. Ennen työn yhteenvetoa johtopäätösten muodossa esittelen tutkimustyön tuotoksena saadun sovelluksen, joka havainnollistaa alkeellisten geometristen muotojen ilmenemistä OpenGL-kontekstissa.

2 OPENGL

OpenGL on kaksi- ja kolmiulotteisten graafisten tietokonesovellusten rakentamiseen tarkoitettu ohjelmointirajapinta, joka antaa ohjelmoijalle pääsyn tietokoneen grafiikkalaitteiston toimintoihin. Kirjaston käskyt ovat riippumattomia käyttö- ja ikkunointijärjestelmästä sekä tietokonelaitteistosta, joten sovelluksia voidaan rakentaa suurelle alustamäärälle. Grafiikan piirtäminen tapahtuu seuraavasti: muotojen data haetaan syötettyjen geometristen primitiivien, kuten esim. pisteiden, viivojen ja kolmioiden perusteella. Varjostinten suorittamat laskelmat määräävät syötteiden renderöintiominaisuudet. Rasteroinnissa primitiivien matemaattiset kuvaukset muutetaan niiden sirpaleiksi, joilla on yhteydet ruudun sijainteihin. Tuotettujen sirpaleiden lopullinen väri ja sijainti määrätään sirpalevarjostimen toimesta. (Shreiner, Sellers, Kessenich & Licea-Kane 2013, 2–3.)

Kuvan renderöinti OpenGL:llä perustuu varjostimien kärkidatalle suorittamiin operaatioihin. Saatavilla olevista varjostimista ainoastaan kärki- ja sirpalevarjostimet ovat vaadittuja, mutta tesselaatio- ja geometriavarjostimien käyttö mahdollistaa laadukkaampien mallien luonnin. (KUVIO 1.) (Shreiner ym. 2013, 11.)



KUVIO 1. OpenGL:n renderöintiputki (mukaillen Shreiner ym. 2013, 10)

2.1 Varjostimet

Varjostimet ovat OpenGL-järjestelmän aliohjelmia, jotka käsittelevät kaikkea niille vietyä dataa ja vievät sitä eteenpäin muihin operaatioihin. Ne ovat suuressa osassa renderöintiputkea tarkastellessa ja ilman

niiden toimintaa ei ole mahdollista suorittaa mitään huomattavia piirto-operaatioita. Nykyisessä iteraatiossaan versiosta 3.1 lähtien varjostimet ovat pakollinen osa mitä tahansa OpenGL-ohjelmaa. (Shreiner ym. 2013, 34.)

Yksinkertaisimmillaan kärkivarjostin vie syötteen datan eteenpäin ilman sen muokkaamista, mutta halutessaan ohjelmoija voi käyttää tätä tasoa monipuoliseen datan manipulointiin, kuten esimerkiksi kärjen ruutusijainnin tai valaistuksen laskemiseen. Tesselaatiotason tehtävä on kasvattaa geometristen primitiivien määrää jakamalla kärkidatan kokoelmista saatuja paikkoja (patch). Yksittäisten primitiivien manipulointia voidaan suorittaa geometriavarjostimelle viedyllä datalla. Ennen seuraavia tasoja kärkien data järjestetään niistä muodostuviin primitiiveihin. Leikkaus taas on OpenGL:n automaattisesti suorittama vaihe, joka varmistaa primitiivien pysymisen näyttöruudulla muokkaamalla niitä tarvittaessa esim. lyhentämällä kolmion kärkeä sen ulottuessa ruudun ulkopuolelle. (Shreiner ym. 2013, 12–13.)

2.2 Puskurit

Puskurit sisältävät kaiken kuvan piirtämiseen ruudulle tarvittavan datan. Viimeinen yksittäisten sirpaleiden käsittely hoidetaan varjostinoperaatioiden jälkeen, jolloin suoritetaan testejä sirpaleen näkyvyyden osalta (Shreiner ym. 2013, 13). Kolmiulotteisessa tilassa pikseleiden näkyvyyttä rajoitetaan säilömällä jokaisen pikselin syvyysarvo syvyyspuskuriin, jolloin pienemmän arvon omaavat pikselit korvaavat suurempiarvoiset. Näkyvyyttä voidaan rajoittaa myös kaavainpuskurin avulla, joka rajoittaa piirron tiettyihin ruudun alueisiin. (Shreiner ym. 2013, 146.)

Mikäli sirpale läpäisee näiden puskureiden asettamat testit, se voidaan kirjoittaa suoraan kaikkien puskureiden muodostamaan kehyspuskuriin, jolloin sen pikselin väri ja mahdollisesti syvyysarvo päivitetään (Shreiner ym. 2013, 14). Kehyspuskuri on siis useimmiten itse kohdenäyttölaite. Syvyys- ja kaavainpuskureiden määrittämiä ominaisuuksia ei lopputuloksessa kuitenkaan nähdä suoraan, vaan niiden suorittamat tehtävät ovat hyödyllisiä taustaoperaatioita. Sen sijaan näkyvissä on ainoastaan ensisijaisen väripuskurin sisältämä informaatio. Nimensä mukaisesti väripuskurit, joita voi olla käytössä useampia samanaikaisesti, sisältävät kaikkien ruudun pikseleiden väridatan ja mahdollisesti tiedon niiden läpinäkyvyydestä. Ensisijaisesta väripuskurista poiketen kaikki muut OpenGL-järjestelmän väripuskurit ovat ruudun ulkopuolella. (Shreiner ym. 2013, 144.)

Jokaisen ruudunpäivitys- ja renderöintioperaation jälkeen kaikki aktiiviset puskurit tyhjennetään vähintään kerran. Ennen seuraavaa kirjoitusta puskureihin tulee niiden osallisuus joko hyväksyä tai estää peiteoperaation kautta. (Shreiner ym. 2013, 146–147.)

2.3 OpenGL Shading Language

Kaikki OpenGL-järjestelmän käyttämät varjostinohjelmat kirjoitetaan C-ohjelmointikieleen perustuvalla OpenGL Shading Language (GLSL) -kielellä (Rodríguez 2013, 35). OpenGL:n alkuvaiheissa grafiikan renderöinnin tapahtumaketju oli ennalta määrätty ja muuttumaton, jolloin kaikki data kävi läpi samat operaatiot ilman ohituksia tai vaihtelua. Grafiikkaprosessorin ohjelmoitavuuden myötä ohjelmoijat saivat mahdollisuuden muokata tapahtumaketjua varjostimilla, jotka korvasivat osan entisistä renderöinnin tasoista. Alustariippumaton GLSL kehitettiin vuonna 2004 helpottamaan grafiikkaohjelmoijien taakkaa, sillä ennen sen julkaisua kaikille laitteistovalmistajille täytyi kehittää omat varjostimet valmistajan käyttämällä Assembly-kielen variaatiolla. (Rodríguez 2013, 23–24.)

GLSL:ssä käytetään kaikkien C:stä tuttujen muuttujien perustyyppien lisäksi erillisiä tyyppejä mm. vektoreiden esitykseen. Vektorityyppejä on saatavilla bool-, int-, uint-, float- ja double-elementeistä koostuville vektoreille, joista kaikista on 2-, 3- ja 4-elementtiset versiot. Matriisien esitykseen käytetään ainoastaan kerta- ja tuplatarkkuuksisia liukulukuja ja matriisin koko voidaan määrittää muuttujan esittelyssä. Mikäli halutaan käyttää neliömallisia matriiseja, voidaan sarake-rivi -määrityksen sijaan käyttää numeroita 2, 3 tai 4 jotka vastaavat haluttuja mittoja. Näiden tyyppien lisäksi on tekstuureille varattu sampler-tyyppi. Muuttujien alustuksessa ja arvon asetuksessa tulee käyttää käytetyn tyypin rakentajaa. Vektorien määrityksessä rakentajia voi myös yhdistellä sillä ehdolla, että lopputulos on kohteen kokoinen: esimerkiksi 4-elementtisen vec4-tyypin voi alustaa kahdella vec2-rakentajalla. (Rodríguez 2013, 37–39.)

3 TYÖKALUT

OpenGL-sovelluksen ohjelmoijalle on avoinna laaja valikoima työkaluja, joilla kaikilla on omat etunsa ja haittansa. Näiden vaihtoehtojen arvioiminen ja sopivien ohjelmistojen sekä kirjastojen valinta on olennainen osa projektin pohjustusta. Valintani perustuu ennen kaikkea omaan perehtyneisyyteeni tai täysin uuden aihealueen tapauksessa muiden kehittäjien mielipiteisiin sekä valintahetkellä oleellisten kriteerien, kuten käyttöjärjestelmän ja haluttujen ominaisuuksien, olemassaoloon.

3.1 Kehitysympäristö

Käytän tässä projektissa Windows 7 -laitetta, joten valitsin ohjelman kehitysympäristöksi itselleni tutun Microsoftin Visual Studio 2015:n. Olen käyttänyt ensisijaisesti tätä kehitysympäristöä opinnoissani ja sen suosion sekä Microsoftin hallitsevuuden takia sille löytyy huomattava määrä tukimateriaalia, josta on ollut suuri apu projektissani. Useimmat Internetissä kohtaamani OpenGL-ohjeistukset käyttävät esimerkkinä juuri Visual Studiota ja tästä syystä ohjelman kehitys on sujunut kohtuullisen ongelmattomasti ohjelmiston suhteen.

3.2 Muut kirjastot

Graafisten sovellusten ohjelmointi laajalle alustapohjalle pelkillä OpenGL-toiminnoilla on haastavaa ja monimutkaista. Tästä syystä on suositeltavaa käyttää peruskirjaston lisäksi erillään kehiteltyjä kirjastoja, jotka helpottavat eri osa-alueiden hallintaa.

Koska kaikki OpenGL:n tukemat alustat suorittavat ikkunoiden ja grafiikan luonnin, syötön hallinnan ja matalan tason laitteistopääsyn eri tavalla, on ohjelmoijan vaikea kehittää sovelluksia usealle alustalle samanaikaisesti. Simple DirectMedia Layer (SDL) on tähän tarkoitukseen luotu kirjasto. SDL antaa pääsyn kaikkiin alustakohtaisiin ominaisuuksiin yhdenmukaisten toimintojen kautta. Vaikka ohjelman kehitys keskittyisi vain yhdelle alustalle, on SDL helppokäyttöisyytensä takia silti hyvä työkalu edellä mainittujen seikkojen hoitamiseen. (Mitchell 2013, 5-6.) Header-tiedoston sisällyttämisen ja globaalien ikkuna- sekä renderöintiosoittimien luonnin jälkeen SDL voidaan alustaa joko kaikilla tai halutuilla osa-

järjestelmillä. Näitä osajärjestelmiä ovat haptisen palautteen, videon, audion, ajastuksen ja ohjaimen järjestelmät. Lisäksi SDL alustaa oletuksena tapahtumien hallinnan, syötteen ja lähdön sekä rihmoittamisen osajärjestelmät. (Mitchell 2013, 14–16.)

Lisäksi tarpeellisia kirjastoja ovat OpenGL Extension Wrangler Library (GLEW), OpenGL Mathematics (GLM) ja Simple OpenGL Image Library (SOIL). GLEW määrittää ajonaikaisesti kohdealustan tukemat OpenGL-laajennukset ja hoitaa niihin liittyvien funktioiden osoittimet (Verkkosivut: GLEW: The OpenGL Extension Wrangler Library). GLM on C++-kielinen kirjasto, joka helpottaa ohjelmoijan matematiikan hallintaa kasvattamalla toiminnallisuutta GLSL-spesifikaatiota mukailevilla funktioilla ja luokilla (Riccio 2016, 6). SOIL taas hoitaa tekstuurien latauksen OpenGL-ohjelmaan (Verkkosivut: lonesock.net: SOIL). Näiden kirjastojen samanaikainen käyttö mahdollistaa kaikki testiohjelman tarvitsemat ominaisuudet.

Valitsin edellä esitellyt kirjastot niiden suosion, yksinkertaisuuden ja dokumentaation määrän perusteella. Varsinkin ikkunan luontiin on saatavilla useita muita kirjastoja, mutta SDL:n monimuotoisuus tekee siitä parhaan vaihtoehdon tähän työhön.

3.3 Ohjelmointikieli

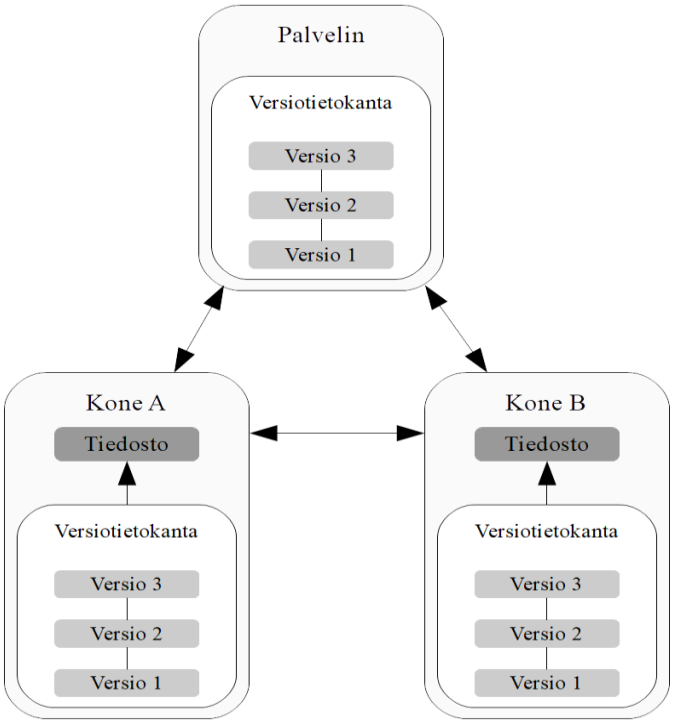
Koska OpenGL on C-kielen kirjasto ja GLSL vastaa pitkälti C:n asettamia konventioita, päätin valita projektin ohjelmointikieleksi C-kieleen pohjautuvan C++:n. Tämä Bell Laboratoriesin Bjarne Stroustrupin kehittämä kieli lisää C:hen hyödyllisiä ominaisuuksia joista kenties merkittävin on olio-ohjelmoinnin mahdollisuus. Kielen kehitys aloitettiin C:n odottamattoman suosion seurauksena ja sen laajan käyttöönoton takia seuraajan tuli olla täysin uuden kielen sijaan edeltäjää laajentava ja parantava spesifikaatio. (Deitel & Deitel 2016, 582.)

Turhan ohjelmoinnin välttämiseksi C++:lla on laaja vakiokirjasto, jonka luokat ja funktiot jouduttavat ohjelmistokehitystä. Näihin erillisiin header-tiedostoihin jaettuihin ominaisuuksiin kuuluu mm. syötteen sekä lähdön, matematiikan, ajan ja merkkirivistöjen käsittelyn hoitavia funktioita. Ohjelmoija voi myös luoda omia headereitä. Header-tiedostot tulee sisällyttää ohjelmakoodin alussa #include-esikääntäjädirektiivin avulla. (Deitel & Deitel 2016, 586–588.)

C++ on oliosuuntautunut ohjelmointikieli, eli ohjelmoija pääasiassa jäsentää koodinsa itsemääritellyistä luokista luotujen olioiden kautta. Luokka sisältää kaikki jonkin objektin määrittävät ominaisuudet ja käytökset. Luodut oliot kommunikoivat rajapintojen kautta ja voivat vaikuttaa toistensa käyttäytymiseen. Tämä käytäntö liittää koodin vahvasti todellisiin esineisiin ja konsepteihin, tehden ohjelmistosuunnittelusta luontevaa. Luokat voivat myös periytyä muista luokista, jolloin ne säilyttävät vanhemman luokan ominaisuuksia mutta lisäävät omia piirteitään. (Deitel & Deitel 2016, 605–606.)

3.4 Versionhallinta

Projektin eri vaiheista voidaan pitää kirjaa versionhallintajärjestelmällä, joka mahdollistaa mm. tiedostojen palautuksen aikaisempaan versioon sekä muutosten vertailun. Tiedostojen muutosten historiaa voidaan säilyttää paikallisessa tietokannassa, mutta turvallisempi tapa on sijoittaa tietokanta myös palvelimelle, jolta se kopioidaan käyttölaitteelle aina sitä käytettäessä. Tämä mahdollistaa useiden laitteiden pääsyn lisäksi jatkuvan varmuuskopion ylläpidon, sillä vikatilanteessa tietokanta voidaan kopioida takaisin palvelimelle ilman tietojen häviämistä. Tällaista järjestelmää kutsutaan hajautetuksi versionhallintajärjestelmäksi (eng. Distributed Version Control System, DVCS). (KUVIO 2.) (Chacon & Straub 2014, 1–4.)



KUVIO 2. Hajautettu versionhallintajärjestelmä (Mukaillen Chacon & Straub 2014, 4)

Git on vuonna 2005 julkaistu DVCS, jonka kehitys sai vaikutteita Linux-kehitystiimin aiemmin käyttämästä BitKeeperistä. Tavoitteina uudelle järjestelmälle oli toteuttaa nopea ja yksinkertainen versionhallinta, joka olisi täysin hajautettu ja kykenevä hallitsemaan erittäin suuria projekteja. Git eroaa muista versionhallintajärjestelmistä tallentamalla jokaisen tiedoston muutosten sijaan tilannevedoksen koko projektista. Muuttumattoman tiedoston tapauksessa se linkitetään aikaisempaan, identtiseen tiedostoon. Koska tilannevedos säilytetään palvelimen lisäksi paikallisesti, ovat operaatiot nopeita ja paikalliseen tietokantaan voi ilman verkkoyhteyttä vapaasti kirjoittaa muutoksia, jotka kopioidaan palvelimelle yhteyden muodostuttua. Järjestelmän alaiset muokatut tiedostot siirtyvät työskentelyhakemistosta järjestysalueelle, josta niiden tilannevedos sitoutetaan pysyvästi Git-kuvauskantaan. (Chacon & Straub 2014, 5–8.) Git-kuvauskantojen suurin isännöitsijä on GitHub (Chacon & Straub 2014, 131).

4 3D-NÄKYMÄN LUONTI

Kolmiulotteisen grafiikkasovelluksen luonti OpenGL:n avulla on koko projektin kannalta monivaiheinen prosessi, joka alkaa käytännössä kehitysympäristön valmistelulla ja jatkuu kärkidatan perusteella luotujen objektien syvyyden ja valaistuksen käsittelyyn.

4.1 Pohjustus

Visual Studio -projektin luonnin jälkeen on syytä linkittää linkitystä vaativat kirjastot ja sisällyttää ne koodiin. Valitsemistani kirjastoista ainoastaan GLM ei vaadi linkitystä, joten sen tarpeelliset headerit voidaan sisällyttää ilman lisäoperaatioita. Muissa tapauksissa suoritetaan linkitys projektin asetuksista, joissa määritetään polut lisäriippuvuuksille sekä sisällytettäville headereille. Kirjastot tuodaan projektiin C++:n #include-esikääntäjädirektiivillä. Koska käytössä on GLEW:n staattinen versio, lisäksi määritetään sen tyyppi ennen headerin sisällyttämistä ohjelmasuorituksessa (GLEW: The OpenGL Extension Wrangler Library). Kyseessä olevien kirjastojen tiedostot ovat eri lisenssien alaisesti vapaasti saatavilla kehittäjien verkkosivustoilta. Alla oleva listaus sisältää kaikkien projektin vaatimien kirjastojen sisällytyksen. Sisällytykset tulee mainita ensimmäisenä ohjelmasuorituksessa, jotta koodi kykenee viittaamaan niihin.

```
#define GLEW_STATIC
#include <GL/glew.h>
#include <SDL.h>
#include <SDL_opengl.h>
#include <SOIL/SOIL.h>
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/type_ptr.hpp>
#include <stdio.h>
#include <iostream>
```

4.1.1 Alustus ja konteksti

Ensimmäinen vaihe SDL-pohjaisen ohjelman kehityksessä on kontekstin luonti, joka määrittää sen toimintaympäristön ominaisuudet. SDL:n alustuksen jälkeen konteksti asetetaan estämään vanhentuneiden funktioiden käyttö, joka onnistuu valitsemalla OpenGL:n ydinprofiilin. Seuraavaksi määritetään haluttu OpenGL-versio ja kaavainpuskurin minimibittimäärä. Näiden määritysten jälkeen voidaan luoda ikkuna, jonka perusteella SDL-konteksti luodaan. (Overvoorde 2017, 10–11.)

```
SDL_Init(SDL_INIT_VIDEO);
SDL_GL_SetAttribute(SDL_GL_CONTEXT_PROFILE_MASK,
SDL_GL_CONTEXT_PROFILE_CORE);
SDL_GL_SetAttribute(SDL_GL_CONTEXT_MAJOR_VERSION, 4);
SDL_GL_SetAttribute(SDL_GL_CONTEXT_MINOR_VERSION, 5);
SDL_GL_SetAttribute(SDL_GL_STENCIL_SIZE, 8);

SDL_Window* window = SDL_CreateWindow("OpenGL", 100, 100,
1024, 768, SDL_WINDOW_OPENGL);
SDL_GLContext context = SDL_GL_CreateContext(window);
```

Tämän esimerkkikoodin seurauksena alustetaan OpenGL 4.5 -spesifikaation ohjelma, jonka kaavainpuskurin minimibittimäärä on 8 ja jonka valittu profiili hylkää vanhentuneet funktiot. 1024 pikseliä korkea ja 768 pikseliä leveä OpenGL:n käytettävissä oleva ikkuna luodaan näytön koordinaatteihin (100, 100).

Ohjelman lopussa täytyy aina muistaa tuhota käytetty konteksti sekä sulkea SDL. Tämä vapauttaa kontekstille varatut resurssit. SDL:n voi sulkea myös alijärjestelmä kerrallaan, mutta joka tapauksessa on parasta kutsua kaiken kattava SDL_Quit()-funktio viimeisenä. (Overvoorde 2017, 11.)

```
SDL_GL_DeleteContext(context);
SDL_Quit();
```

Nyt voidaan kirjoittaa silmukka, jonka sisällä tapahtumia käsitellään.

```
SDL_Event windowEvent;
while (true) {
```

```
        if (SDL_PollEvent(&windowEvent)) {
                if (windowEvent.type == SDL_QUIT) break;
        }

        SDL_GL_SwapWindow(window);
}
```

Yllä olevassa koodissa ikkunatapahtumamuuttujan luonnin jälkeen avataan päättymätön silmukka, jossa mahdollisia tapahtumia jatkuvasti etsitään ja ohjelman tässä vaiheessa toimitaan ainoastaan käyttäjän painaessa ikkunan sulkevaa ruksia. Tällöin silmukasta poistutaan ja ohjelma jatkaa kontekstin tuhoamiseen. Silmukan lopussa kutsutaan funktiota, joka esimerkin kaltaisessa tuplapuskuroidussa ohjelmassa vaihtaa etu- ja takapuskurin paikkaa. (Overvoorde 2017, 11–12.)

4.1.2 Varjostinohjelmien luonti, koonti ja käyttö

Varjostinohjelmat voidaan kirjoittaa joko erillisinä, pääohjelmaan ladattavina tiedostoina tai merkkirivistönä suoraan koodiin. Varjostinohjelmassa ensimmäisenä tulee kertoa käytettävä GLSL:n versio. Tämä versiointi on OpenGL 3.3 -spesifikaatiosta lähtien ollut sidottuna vastaavaan OpenGL-versioon, joten on loogista käyttää myös GLSL:n versiota 4.5. (Shreiner ym. 2013, 23.)

```
#version 450 core
...
void main() {
        ...
}
```

Kuten OpenGL-profiilin valinnassa, versiomäärityksen ”core” -pääte estää vanhentuneiden funktioiden käytön. Varjostimen vastaanottamat tulomuuttujat sekä edelleen vietävät lähtömuuttujat voidaan määrittää tämän jälkeen ja itse operaatiot käsitellään C-ohjelman tapaisesti main-rutiinissa. (Shreiner ym. 2013, 23-24.) Jokaiselle varjostimelle luodaan oma varjostinobjekti, joka mahdollistaa niiden helpon uudelleenkäytön. Varjostinobjektia luodessa määritetään sen tyyppi, jonka jälkeen siihen voidaan yhdistää ylläolevan koodin kaltainen varjostinlähde. Kun varjostin on koottu, on syytä tarkistaa, onnistuiko koonti

ilman virheitä ja tarvittaessa poistua ohjelmasta. Nämä askeleet suoritetaan jokaisen käytettävän varjostimen osalta. (Shreiner ym. 2013, 72–73.)

```
GLuint vertexShader = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vertexShader, 1, &vertexSource, NULL);
glCompileShader(vertexShader);

GLint status;
glGetShaderiv(vertexShader, GL_COMPILE_STATUS, &status);
if (status == GL_TRUE) {
        std::cout << "Vertex compiled!" << std::endl;
}
```

Tässä luodaan kärkivarjostimelle objekti glCreateShader-funktiolla ja sen tyyppiargumentilla GL_VERTEX_SHADER. Varjostinlähteen valinnassa käytetään argumentteina luotua kohdeobjektia sekä lähteen formatoinnin kannalta tarpeellista tietoa. Esimerkkinä myös hyvin yksinkertainen tapa tarkistaa varjostimen koonti, jonka seurauksena ohjelma ilmoittaa, jos koonti on onnistunut. glGetShaderiv palauttaa varjostimen tilan koonnin jälkeen, jota voidaan käyttää halutulla tavalla. (Shreiner ym. 2013, 72–73.)

Jos kaikki tarvitut varjostinobjektit kootaan ilman virheitä, voi ohjelma jatkaa varjostinohjelman määritykseen. Varjostinohjelma on suoritettava ohjelma, joka sisältää kaikki liitosoperaation jälkeen yhdistetyt varjostinobjektit. Varjostinohjelman nimeämisen ja luonnin jälkeen siihen assosioidaan halutut varjostimet kiinnitysfunktiolla. Lopulta varjostimet käsitellään yhdeksi ohjelmaksi ja otetaan käyttöön. (Shreiner ym. 2013, 73–75.)

```
GLuint shaderProgram = glCreateProgram();
glAttachShader(shaderProgram, vertexShader);
glAttachShader(shaderProgram, fragmentShader);

glLinkProgram(shaderProgram);
glUseProgram(shaderProgram);
```

Varjostimien lähdekoodin haun ja koonnin jälkeen loput operaatiot ovat suoraviivaisia. Tämä koodi esittää varjostinohjelman käyttöönoton.

4.2 Mallit

Sanalla malli viitataan tässä yhteydessä kärki- ja tekstuuridatan yhdessä luomaan virtuaaliseen esineeseen. Kärjet ovat minkä tahansa geometrian nurkkapisteitä ja ne määrittävät esineen muodon. Tekstuuri taas on kärkien muodostaman geometrian päälle sijoitettava kuvadata. Kolmiulotteisen näkymän kaikki geometria koostuu mallien kärkien, tekstuurien ja syvyysvaikutelman määrittelystä.

4.2.1 Monikulmioiden määritys ja piirto

Minkä tahansa muodon piirto-operaation suorittamiseksi OpenGL tarvitsee ensisijaisesti tietoa halutuista kärkien ominaisuuksista. Tämä tieto syötetään kärkitaulukko-objektiin (eng. vertex array object, VAO), jonka alustuksen ja puskuroinnin jälkeen ohjelma voi käyttää sitä syötedatana. VAO:n alustus suoritetaan allokoimalla haluttu määrä vapaana olevia objektinimiä määritettyyn taulukkoon. Nimien generoinnin jälkeen voidaan ohjelmaa käskeä luomaan VAO, jolloin se saa käytettävissä olevan nimen, sille allokoidaan sen tarvitsema muisti ja siitä tehdään aktiivinen. Tämä sitominen tulee suorittaa aina kun datalle halutaan suorittaa operaatioita, mikäli objekti ei ole sillä hetkellä sidottuna. (Shreiner ym. 2013, 17–18.)

VAO:n sisältämä data säilötään kärkipuskuriobjektiin (eng. vertex buffer object, VBO), joka on OpenGL:n käytettävissä olevaa muistia. Samoin kuin VAO:n alustuksessa VBO tarvitsee ensisijaisesti nimen, jonka vastaavan generoinnin jälkeen se luodaan puskureille määritetyn sitomisfunktion avulla. Näitä puskureita on monenlaisia, joten toisin kuin VAO:n tapauksessa määritetään tässä vaiheessa minkälaista tietoa puskuriin säilötään. Seuraava vaihe on siirtää kärkidata puskuriobjektiin. (Shreiner ym. 2013, 19–21.)

```
GLuint vao;
glGenVertexArrays(1, &vao);
glBindVertexArray(vao);

float vertices[] = {
        ...
```

```
};

GLuint vbo;
glGenBuffers(1, &vbo);
glBindBuffer(GL_ARRAY_BUFFER, vbo);
glBufferData(GL_ARRAY_BUFFER,  sizeof(vertices),  vertices,
GL_STATIC_DRAW);
```

Tässä koodissa alustetaan ensin VAO:lle muuttuja, jota seuraavilla riveillä hyödynnetään generointi- ja sitomisfunktioissa. Generointifunktion ensimmäisellä argumentilla kerrotaan haluttujen nimien määrä. Puskurointioperaatiot tarvitsevat kärkidatan taulukkomuodossa, joten vertices-taulukko alustetaan tässä vaiheessa. Tämä taulukko sisältää ensisijaisesti informaation monikulmion kärkien sijainneista ja väreistä. Ominaisuuksien esittely määritetään myöhemmin ohjelmasuorituksessa. VBO:n luonnissa ainoa eroavaisuus aikaisempaan on kärkipuskurin täsmennys GL_ARRAY_BUFFER-enumeraatiolla. Lopulta glBufferData-funktio täyttää puskurin sille vietyjen argumenttien perusteella, joihin kuuluu viittaus aktiiviseen puskuriin, kärkitaulukon koosta laskettu tarvitun muistin määrä, itse kärkidata ja datan kirjoituksen ja lukemisen määrittävä käyttöenumeraatio (Shreiner ym. 2013, 21–22).

Jotta varjostimet saisivat pääsyn muun ohjelman dataan, tulee puskurissa oleva data assosioida kärkivarjostimen muuttujiin. Varjostimen syötemuuttujat liitetään kärkiominaisuustaulukkoon, jolloin OpenGL voi hakea ne muistista. Taulukon päälle kytkemisen jälkeen ohjelma on valmis piirtoon. (Shreiner ym. 2013, 25–28.)

```
GLint colAttrib = glGetAttribLocation(shaderProgram,
"color");
glEnableVertexAttribArray(colAttrib);
glVertexAttribPointer(colAttrib, 3, GL_FLOAT, GL_FALSE, 7 *
sizeof(float), (void*)(2*sizeof(float)));
```

Varjostinohjelman muuttujan sijainnin haussa käytetään esimerkiksi glGetAttribLocation-funktiota, joka sijoittaa ohjelman tunnistimen ja halutun syötemuuttujan perusteella tiedon sen sijainnista muuttujaan. Tässä esimerkissä kärkivarjostimen color-muuttujan sijainti kirjoitetaan pääohjelman colAttrib-muuttujaan. (Shreiner ym. 2013, 130.) Tätä muuttujaa hyödyntämällä voidaan seuraavalla rivillä ottaa kär-

kiominaisuustaulukko käyttöön. Lopuksi varjostinohjelmalle kerrotaan mistä data löytyy. Koska kyseessä on kärkien väriominaisuudet, jotka on määritelty muiden kärkiominaisuuksien ohella esimerkiksi edellä käytetyssä vertices-taulukossa, käytetään glVertexAttribPointer-funktion argumentteina tarpeellisia arvoja. Sijaintimuuttujan jälkeen funktiolle kerrotaan arvojen määrä, tässä tapauksessa kolme värin R-, G-, ja B-arvojen esittämiseksi. Arvojen tyypin ja normalisoinnin käytön täsmennyksen jälkeen kaksi viimeistä argumenttia kertovat, miten data on aseteltu kärkitaulukossa. Ensin syötetään taulukon kärjet erottava stride-arvo. Koska käytössä on taulukko, jonka float-tyyppiset tiedot ovat asetettu seitsemän arvon ryhmiin, annetaan funktiolle vastaava arvo eli seitsemän kertaa float-luvun koko. Lopulta arvojen mahdollinen ohitus täsmennetään OpenGL:n BUFFER_OFFSET-makron avulla. Esimerkin taulukko sisältää kärkien RGB-väriarvot jokaisen seitsemän arvon jonon kolmannessa, neljännessä ja viidennessä sijainnissa, joten kaksi ensimmäistä arvoa ohitetaan. (Shreiner ym. 2013, 26–27.)

Näin assosioidut varjostimien syötemuuttujat esitellään asettamalla niiden edelle in-avainsana ja vastaavasti varjostimesta edelleen lähtevät muuttujat tarkennetaan out-avainsanalla. Kärkivarjostimen ulostulo eli kärkien sijaintidata sijoitetaan ennalta määritettyyn vec4-tyyppiseen gl_Position-muuttujaan. Koska sijainnit ovat määritetty kolmiulotteisesti XYZ-koordinaatistossa, ei muuttujan viimeinen arvo ole tarpeellinen ja se voidaan tässä tapauksessa asettaa luvuksi yksi. (De Vries 2017, 37–39.) Myös sirpalevarjostimella on yksi pakollinen ulostulo, joka määrää kärkidatasta laskettujen sirpaleiden värit. Kuten sijainnin tilanteessa on tämä muuttuja 4-elementtinen vektori, mutta koordinaattien sijaan sen elementit vastaavat haluttuja sirpaleen värin R-, G- ja B arvoja sekä läpinäkyvyyttä. (Overvoorde 2017, 23–24.)

Kun määritettyjen kärkien muodostama muoto halutaan piirtää, on yksinkertaisinta käyttää OpenGL:n taulukkopiirtomenetelmää. Tämä menetelmä lukee käytössä olevien kärkitaulukoiden datan järjestyksessä ensimmäisestä viimeiseen elementtiin ja piirtää niiden perusteella ruudulle primitiivin. (Shreiner ym. 2013, 115.)

```
glDrawArrays(GL_TRIANGLES, 0, 3);
```

Taulukkopiirtofunktion ensimmäisellä argumentilla kerrotaan sen toimintatila, joka on tässä tapauksessa GL_TRIANGLES eli kolmioprimitiivien piirtoon soveltuva tila. Toinen argumentti on taulukon alusta ohitettavien elementtien määrä ja viimeinen kärkien määrä, kolmion tapauksessa kolme. Tämä menetelmä ei kuitenkaan tue kärkien uudelleenkäyttöä ja on joustamaton. Indeksoidulla piirrolla voidaan taulukon elementtejä käyttää vapaasti halutussa järjestyksessä ja myös hyödyntää samaa kärkeä useita ker-

toja yhdessä piirto-operaatiossa. glDrawElements-funktio käyttää elementtipuskurin sisältämää indeksidataa. (Shreiner ym. 2013, 115-116.) Kuten muidenkin puskureiden tapauksessa, luodaan puskurille ensin objekti, joka täytetään taulukkomuodossa olevalla indeksidatalla.

```
GLuint elements[] = {
        ...
};

GLuint ebo;
glGenBuffers(1, &ebo);

glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ebo);
glBufferData(GL_ELEMENT_ARRAY_BUFFER,      sizeof(elements),
elements, GL_STATIC_DRAW);
```

Ainoa muutos aiemmin esiteltyihin puskurioperaatioihin on kohdepuskurin määritys, joka tässä tapauksessa saavutetaan käyttämällä GL_ELEMENT_ARRAY_BUFFER-argumenttia puskurointifunktiossa. Itse piirtofunktio eroaa hieman aiemmasta ja se tarvitsee tiedon halutusta toimintatilasta, indeksitaulukon elementtien määrästä, niiden tyypistä sekä mahdollisesta ohitusarvosta. (Overvoorde 2017, 33.)

```
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);
```

Koska elementtien määrä on asetettu luvuksi kuusi, voidaan tällä esimerkillä piirtää helposti kahden kolmion muodostamia muotoja ilman ylimääräisiä piirto-operaatioita. GL_UNSIGNED_INT-argumentilla kerrotaan taulukon elementtien olevan etumerkittömiä kokonaislukuja ja viimeinen argumentti osoittaa, ettei taulukon alusta ohiteta yhtään elementtiä (Shreiner ym. 2013, 116).

4.2.2 Tekstuurit

Objektien pintojen peittämiseen käytetään tekstuureja, jotka ovat suuria tekseleistä koostuvia kuvadatan kimpaleita. Tekstuurikartoitus tarkoittaa arvojen, kuten esimerkiksi värin hakemista sirpalevarjostimen erikoistuneesta taulukosta. Yksi-, kaksi-, ja kolmiulotteisten tekstuurikarttojen lisäksi OpenGL tukee

kuutiokartta-, puskuri-, ja taulukkotekstuureja ja ne ovat useimmiten valokuvia tai muita valmiita kuvatiedostoja. Tekstuurien käyttö vaatii niiden sitomista tiettyihin OpenGL-kontekstin sitomispisteisiin ja varjostinten tekstuurin ulotteisuutta vastaavan näytteenotinmuuttujan hyödyntämistä. (Shreiner ym. 2013, 260–262.)

Samoin kuin esimerkiksi kärkipuskureiden tapauksessa tekstuuri otetaan käyttöön varaamalla ensin haluttu määrä nimiä tekstuuriobjekteille, jotka sijoitetaan määritettyyn taulukkoon. Objekti luodaan sitomalla se aktivoituun tekstuuriyksikköön. (Shreiner ym. 2013, 263-265.) Tekstuurit tarvitsevat kärkikohtaiset koordinaatit, joiden perusteella sirpaleelle haetaan väri. Siinä tapauksessa, että koordinaatit asettuvat kärkisijaintien ulkopuolelle, tulee tekstuuria muokata. Tätä muokkausta voidaan hallita asettamalla käyttäytymisen määräävät parametrit jokaiselle tekstuurin akselille. (Shreiner ym. 2013, 298–301.) Kuvan lataus tiedostosta OpenGL:ään suoritetaan SOIL-kirjastolla.

Koska tekstuurien koordinaatit sijaitsevat kärkitaulukossa ja varjostimet tarvitsevat pääsyn uuteen dataan, aloitetaan tekstuurien alustus noutamalla uuden varjostinmuuttujan sijainti ja assosioimalla taulukon arvot niihin (Overvoorde 2017, 41).

```
GLint texAttrib = glGetAttribLocation(shaderProgram,
"texcoord");
glEnableVertexAttribArray(texAttrib);
glVertexAttribPointer(texAttrib, 2, GL_FLOAT, GL_FALSE, 7 *
sizeof(float), (void*)(5 * sizeof(float)));
```

Tässä siis lisätään viittaus kärkitaulukon rakenteen mukaan sen jokaisen indeksin kahteen viimeiseen arvoon. Seuraavaksi suoritetaan tekstuurien käsittelyyn vaaditut alustukset. (Overvoorde 2017, 41.)

```
GLuint textures[1];
glGenTextures(1, textures);
```

Yllä olevassa esimerkissä generoidaan ja sijoitetaan textures-taulukkoon tekstuureille varattu nimi. Nyt voidaan siirtyä tekstuurien määritykseen.

```
int width, height;
unsigned char* image;
```

```
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, textures[0]);
image = SOIL_load_image("test.png", &width, &height, 0,
SOIL_LOAD_RGB);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0,
GL_RGB, GL_UNSIGNED_BYTE, image);
SOIL_free_image_data(image);
```

Koska OpenGL-konteksti tukee useiden tekstuuriyksiköiden samanaikaista käyttöä, voidaan aktiivinen yksikkö määrittää ennen tekstuuriobjektin luomista glActiveTexture-funktiolla. Tekstuuri sidotaan aktiiviseen yksikköön argumentilla, joka määrittää sen ulotteisuuden, tässä tapauksessa luotu tekstuuri määritetään kaksiulotteiseksi. Luoduille tekstuuriobjekteille määritetään paikka muistista sen ulotteisuutta vastaavan varastointifunktion avulla. Tässä esimerkissä käytetään tekstuurin uudelleenmäärityksen mahdollistavaa funktiota, jonka parametrit määrittävät mm. tekselidatan formaatin ja tyypin. (Shreiner ym. 2013, 264-269.) SOIL:in SOIL_load_image -metodi lataa muistista kuvan etumerkittömään char-muuttujaan. Tämän metodin argumentteina käytetään halutun tiedoston polkua, varattujen korkeus- ja leveysmuuttujien osoitteita sekä tietoa kuvan kanavista. SOIL_LOAD_RGB-parametri pakottaa kyseisen tiedoston kanavat RGB-formaattiin. Lopulta kuvadatan muisti vapautetaan. (Overvoorde 2017, 39-40.) Sirpalevarjostimen puolella tehdään tässä vaiheessa pääsyn tekseleihin mahdollistava lisäys (Shreiner ym. 2013, 262).

```
uniform sampler2D tex;
```

Muuttujan sampler2D-tyyppi vastaa jälleen kyseisen tekstuurin ulotteisuutta. Uniform on OpenGL:n määrite, joka täsmentää muuttujan arvon muuttumattomaksi käsittelyssä olevan primitiivin kannalta ja myös sovelluksen määritettäväksi ennen varjostimen suoritusta (Shreiner ym. 2013, 46).

Tekseleiden lukumenetelmää hallitaan asettamalla aktiivisen kohteen ominaisuuksille parametreja (Shreiner ym. 2013, 295). Mikäli tekstuurin koordinaatit päätyvät määritellyn 0.0 – 1.0 -alueen ulkopuolelle, tulee ohjelman muuttaa koordinaatteja asianmukaisesti. Tekstuurin S-, R- ja T -akseleiden suuntaista käyttäytymistä määräävät parametrit, jotka asettavat sen joko toistumaan säännöllisesti tai kiinnittymään eri tavoin tekstuurin reunaan. (Shreiner ym. 2013, 300.)

```
glTexParameteri(GL_TEXTURE_2D,          GL_TEXTURE_WRAP_S,
GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D,          GL_TEXTURE_WRAP_T,
GL_REPEAT);
```

Tekselit harvoin vastaavat täydellisesti lopullisen kuvan pikseleitä. Tästä syystä yhden pikselin alueella voi olla joko suuri joukko tekseleitä tai yhden tekselin pieni osa. Näissä tapauksissa tekstuuri suodatetaan joko bilineaarisella menetelmällä tai sitä ei suodateta lainkaan hakemalla uusi arvo lähimmän tekselin perusteella. Bilineaarinen suodatus tarkoittaa uudelleennäytteenottoa, jossa tekstuurikoordinaatin kahden lähimmän näytteen laskettujen painotusten perusteella saadaan niiden keskiarvo. Menetelmä suoritetaan vuoron perään kaikille tarpeellisille akseleille, kaksiulotteisen tekstuurin tapauksessa s- ja r-akseleille. Suurennukselle ja pienennykselle on OpenGL:ssä omat parametrit, jotka voidaan asettaa halutusti. (Shreiner ym. 2013, 329–332.)

```
glTexParameteri(GL_TEXTURE_2D,        GL_TEXTURE_MIN_FILTER,
GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D,        GL_TEXTURE_MAG_FILTER,
GL_LINEAR);
```

Näin asetettujen parametrien perusteella tekselit tarvittaessa toistuvat tekstuurin reunaan asti ja niiden sisältämä data suodatetaan lineaarisesti.

4.2.3 Syvyys ja näkökulma

Sekä katselukulman että kappaleiden sijainnin, kierron ja skaalauksen manipulointi suoritetaan OpenGL-kontekstissa matriisimuunnoksilla (Shreiner ym. 2013, 214). GLM helpottaa tätä työtä funktioilla, jotka muokkaavat ns. malli-, katselu- ja projektiomatriiseja. Yhdistämällä nämä matriisit sijaintidataan voidaan saavuttaa kaikki tarvittavat muunnokset. (De Vries 2017, 91.)

Mikäli objektin kärkiä ei ole määritetty tilakoordinaateilla, suorittaa mallimatriisi muunnoksen mallin sijainneista tilan sijainteihin. Jos muunnosta ei tarvita, voi tämän matriisin asettaa yksikkömatriisiksi eli

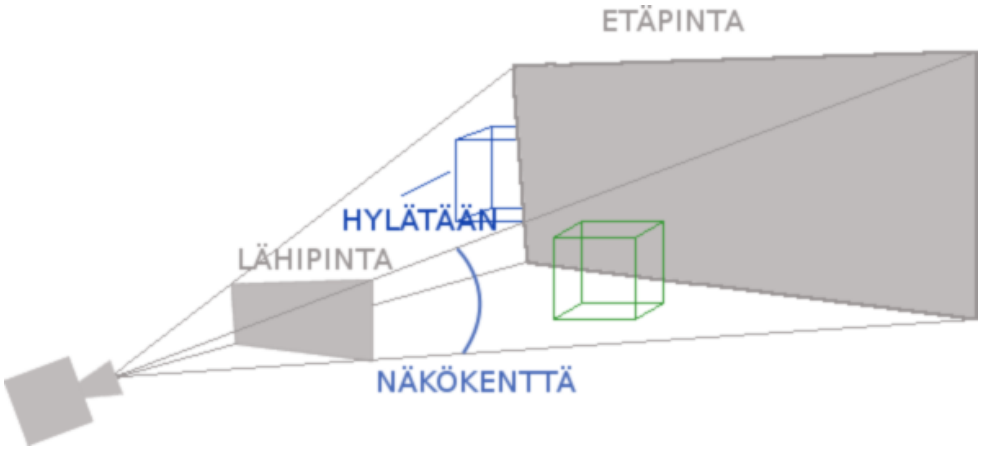
matriisiksi, jolla vektoreita tai matriiseja kerrottaessa saadaan tulokseksi alkuperäiset arvot. Mallin sijainnin manipulointi voidaan suorittaa tässä vaiheessa esim. GLM:n rotate-funktiolla. (De Vries 2017, 91.)

```
glm::mat4 model;
model = glm::rotate(model, glm::radians(60.0f),
glm::vec3(0.0f, 0.0f, 1.0f));
```

Koska luotu matriisi on oletuksena yksikkömatriisi, voidaan se suoraan sijoittaa jatko-operaatioiden funktioihin, kuten yllä on esitetty (De Vries 2017, 81). Esimerkin seurauksena 4x4 -suuruista model-matriisia kierretään 60° z-akselin ympäri. Seuraavaksi määritetään katselukulma katselumatriisin avulla. GLM auttaa tässä vaiheessa lookAt-funktiolla, joka simuloi oikean maailman liikkuvaa kameraa. (De Vries 2017, 97–98.)

```
glm::mat4 view = glm::lookAt(
        glm::vec3(1.5f, 1.8f, 1.5f),
        glm::vec3(0.0f, 0.0f, 0.0f),
        glm::vec3(0.0f, 1.0f, 0.0f)
);
```

Funktion parametrit määrittävät kameran sijainnin, ruudulle keskitettävän pisteen ja ”ylös”-akselin (De Vries 2017, 98). Lopulta projektiomatriisi selvittää leikkauskoordinaatit, joiden perusteella ruudun ulkopuolelle jääviä kärkiä ei piirretä. GLM:n perspective-funktio määrää halutun näkökentän, ruudun kuvasuhteen sekä lähi- ja etäpinnat. Näkökenttä esitetään pituussuuntaisena asteena ja viittaa kulmaan, joka muodostuu kameran ja ikkunan väliin. Lähi- ja etäpinnat taas ovat kamerasta laskettuja etäisyyksiä, joiden perusteella joko liian lähellä tai liian kaukana olevia kärkiä ei piirretä. (KUVIO 3.) (De Vries 2017, 89–90.)



KUVIO 3. Näkökulman katkaistu pyramidi (Mukaillen De Vries 2017, 90)

```
glm::mat4 proj = glm::perspective(glm::radians(80.0f),
1024.0f / 768.0f, 1.0f, 10.0f);
```

Tässä tapauksessa näkökenttä on 80°, kuvasuhde lasketaan resoluution kautta ja lähi- sekä etäpinnat ovat määritetty yhden ja kymmenen yksikön etäisyyksille. Koska kaikkia luotuja matriiseja tarvitaan varjostimissa, liitetään ne varjostimien vastaaviin uniformeihin.

```
GLint uniModel = glGetUniformLocation(shaderProgram,
"model");
glUniformMatrix4fv(uniModel, 1, GL_FALSE,
glm::value_ptr(model));
GLint uniView = glGetUniformLocation(shaderProgram,
...
GLint uniProj = glGetUniformLocation(shaderProgram,
...
```

Kärkivarjostimeen lisätään tarvittavat uniformit, ja sen sijaintimuuttuja päivitetään vastaamaan tehtyjä muutoksia (De Vries 2017, 92–93.)

```
uniform mat4 model;
uniform mat4 view;
uniform mat4 proj;

gl_Position = proj * view * model * vec4(position, 1.0);
```

Tässä tilanteessa syvyyspuskurin tulee seurata kappaleen etäisyyttä katselupisteestä ja ohjeistaa kontekstia piirtämään ainoastaan näkyvissä olevat pikselit. Oletuksena syvyyspuskurin tyhjennysarvo on katselualueen kauimmainen arvo, joten kaikkien objektien syvyydet ovat tuota arvoa lähempänä. Syvyyden testaus otetaan käyttöön viemällä tarvittava arvo glEnable-funktiolle ja tyhjentämällä puskuri jokaisella ruudunpäivityksellä. (Shreiner ym. 2013, 163.)

```
glEnable(GL_DEPTH_TEST);
...
while (true) {
        ...
        glClear(GL_COLOR_BUFFER_BIT |GL_DEPTH_BUFFER_BIT);
        ...
}
```

4.3 Valaistus

Tietokonegrafiikan valaistuksen klassinen malli sisältää kolme valaistuksen tasoa, jotka muodostavat kohtuullisen realistisen approksimaation oikean maailman valaistuksesta. Klassisen mallin ensimmäinen taso käsittää ympäröivän valon, jolla ei ole suuntaa ja joka kattaa vakiona koko näkymän. Ympäröivän valaistuksen laskeminen ei vaadi valonlähteiden tai katsojan silmän suunnan määrittelyä. Ympäröivän valon päälle lasketaan hajotettu valo, joka on pinnoille tasaisesti jakaantunut jonkin valonlähteen säteilemä valo. Heijastustehoste asetetaan muun valaistuksen päälle ja se on peilimäinen kiilto, josta ilmenee pinnan näennäinen kiiltävyys. Tämä tehoste on kulmasidonnainen ja sen laskemiseen tarvitaan tiedot pintanormaalista sekä valonlähteen ja silmän suunnasta. (Shreiner ym. 2013, 360–362.) Valonlähteen sijainnin havainnollistamiseksi voidaan sille luoda valo-objekti, joka voi olla mikä tahansa muoto. Jotta muiden objektien muutokset eivät vaikuttaisi vakiona säilyvään valo-objektiin, luodaan sille oma kärkitaulukko-objekti ja kärki- sekä sirpalevarjostimet, jotka määrittävät sen värin, joka yleensä jätetään valkoiseksi RGB-arvoilla 1, 1 ja 1. Objekti siirretään tilakoordinaateilla esitettyyn valonlähteen sijaintiin, jolloin niiden sijainnit vastaavat toisiaan. (De Vries 2017, 110–112.)

Luonnossa kappaleen värin määrittää siitä heijastuvien valon aallonpituuksien määrä. Valonlähteestä säteilevä valo koostuu useista väreistä, jotka esineen kohdatessaan joko imeytyvät esineeseen tai heijastuvat ja päätyvät silmään. Tietokonegrafiikassa tätä ilmiötä voidaan simuloida asettamalla valon värille RGB-tyyppinen vektori ja kertomalla yhteen tämä väri sekä kappaleen oma väri. (De Vries 2017, 109–110.) Yksinkertainen mallinnus ympäröivästä valosta suoritetaan asettamalla valolle vahvuus, jolla valon väri kerrotaan ennen sen lisäämistä kappaleen väriin. (De Vries 2017, 115.)

```
glm::vec3 lightColor(1.0f, 1.0f, 1.0f);
GLint uniLightColor = glGetUniformLocation(shaderProgram,
"lightColor");
glUniform3fv(uniLightColor, 1, glm::value_ptr(lightColor));

glm::vec3 lightPos(1.0f, 1.0f, 1.5f);
GLint uniLightPos = glGetUniformLocation(shaderProgram,
"lightPos");
glUniform3fv(uniLightPos, 1, glm::value_ptr(lightPos));
```

Valon väri ja myöhemmin tarpeellinen valonlähteen sijainti ovat tässä esimerkissä globaaleja muuttujia ja ne liitetään varjostimiin aiemmin selostetulla tavalla. Samalla nämä muuttujat lisätään varjostimiin. Itse laskutoimitukset suoritetaan sirpalevarjostimessa, jolloin ympäröivän valon vahvuus ja valon väri kerrotaan kappaleen värillä ja lopulta syötetään ulos lähtevään arvoon outColor.

Sirpalevarjostin:

```
...
uniform vec3 lightColor;
uniform vec3 lightPos;
void main() {
        float ambientStrength = 0.1;
        vec3 ambient = ambientStrength * lightColor;
        vec3 result = ambient * Color;
        ...
        outColor = vec4(result, 1.0);
}
```

Hajotettu valo tekee kappaleesta sitä kirkkaamman, mitä lähempänä sen sirpaleet ovat valonsäteiden suuntaumaa. Sirpaleen pintanormaalin ja valonsäteen vektorin välisen kulman suuruus kertoo valon paikallisen intensiteetin. Kulman pienentyessä vektoreiden pistetulo lähenee lukua yksi, jolloin intensiteetti kasvaa. Vastaavasti 90 asteen kulmassa pistetulo on nolla eli valoa ei heijastu lainkaan. Kuution kaltaisen yksinkertaisen muodon pintanormaalit voidaan sijoittaa suoraan olemassa olevaan kärkitaulukkoon. Tämä vaatii uuden kärkiominaisuuden määrittämistä ja aiempien päivittämistä uuteen kärkitaulukkoformaattiin. (De Vries 2017, 115–117.)

```
GLfloat vertices[] = {
//            Position            Color             Texcoord.   Normal
             -0.5f, -0.5f, -0.5f, 1.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, -1.0f,
              0.5f, -0.5f, -0.5f, 1.0f, 0.0f, 0.0f, 1.0f, 0.0f, 0.0f, 0.0f, -1.0f,
              0.5f,  0.5f, -0.5f, 1.0f, 0.0f, 0.0f, 1.0f, 1.0f, 0.0f, 0.0f, -1.0f,
              0.5f,  0.5f, -0.5f, 1.0f, 0.0f, 0.0f, 1.0f, 1.0f, 0.0f, 0.0f, -1.0f,
             -0.5f,  0.5f, -0.5f, 1.0f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f, 0.0f, -1.0f,
             -0.5f, -0.5f, -0.5f, 1.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, -1.0f,
             ...
}
```

Kärkitaulukon kolme viimeistä arvoa vastaavat nyt pintanormaalien suuntaa, joten taulukon rivin pituus on yhteensä 11 yksikköä. Uuden kärkiominaisuuden lisäksi muiden ominaisuuksien stride-arvot vaihdetaan vastaamaan tätä muutosta. Yllä oleva ote kärkitaulukosta esittää kuution ensimmäisen tahkon ominaisuudet, jotka ovat järjestyksessä vasemmalta sijainti, väri, tekstuurikoordinaatit sekä pintanormaalit.

```
GLint normalAttrib = glGetAttribLocation(shaderProgram,
"normal");
glEnableVertexAttribArray(normalAttrib);
glVertexAttribPointer(normalAttrib, 3, GL_FLOAT, GL_FALSE,
11 * sizeof(float), (void*)(8 * sizeof(float)));
```

Sirpaleiden sijainnin selvittämiseksi voidaan kärkidatan sijaintiominaisuudet kertoa kärkivarjostimessa mallimatriisilla, jolloin ne muuntuvat tilakoordinaateiksi. Tätä tietoa käytetään valonsäteen suuntavektorin laskemisessa vähentämällä valonlähteen sijaintivektorista sirpaleen sijaintivektori. Seuraavien laskutoimitusten yksinkertaistamiseksi ja koska ainoa aiheellinen tieto vektoreista on niiden suunta, normalisoidaan kaikki vektoridata, jotta vektorit päätyvät yksikkövektoreiksi. Normaalien ja valon suuntavektorin pistetulon arvo asetetaan max-funktiolla vähintään luvuksi nolla, sillä värien negatiiviset arvot eivät ole toivottuja. Tämä hajaantumiskomponentti lisätään ympäristökomponenttiin ja kerrotaan kappaleen värillä, jolloin lopputulos voidaan viedä värin lähtömuuttujaan. (De Vries 2017, 117–118.)

Kärkivarjostin:

```
...
in vec3 normal;
out vec3 Normal;
out vec3 FragPos;
void main() {
        Normal = normal;
        FragPos = vec3(model * vec4(position, 1.0));
}
```

Sirpalevarjostin:

```
...
in vec3 Normal;
in vec3 FragPos;
void main() {
        vec3 norm = normalize(Normal);
        vec3 lightDir = normalize(lightPos - FragPos);
        float diff = max(dot(norm, lightDir), 0.0);
        vec3 diffuse = diff * lightColor;
        vec3 result = (ambient + diffuse) * Color;
        outColor = vec4(result, 1.0);
}
```

Viimeisenä osana klassista valaistusmallia lasketaan heijastustehoste. Tehoste on voimakkaimmillaan siinä pisteessä, missä valon heijastus havaitaan. Pintanormaalin ympäri heijastuvan valon suuntavektorin ja katselusuunnan kulman pienentyessä tehosteen vaikutus kasvaa. Siinä pisteessä, missä valonlähde heijastuu kohteesta takaisin, esiintyy kirkas korostus. (De Vries 2017, 120–121.) Näkymän katsojan sijainti tilakoordinaateissa vastaa lookAt-funktion ensimmäistä parametria, joten samaa vektoria voidaan käyttää seuraavissa sirpalevarjostimen laskutoimituksissa kyseisen muuttujan varjostimille viennin jälkeen.

```
glm::vec3 viewPos(-1.0f, 0.8f, 1.5f);
```

```
GLint uniViewPos = glGetUniformLocation(shaderProgram,
"viewPos");
glUniform3fv(uniViewPos, 1, glm::value_ptr(viewPos));
```

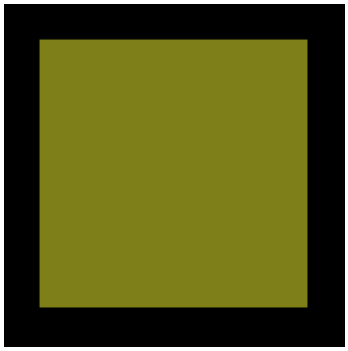
Sirpalevarjostin:

```
...
uniform vec3 viewPos;
void main() {
        float specularStrength = 1.5;
        vec3 viewDir = normalize(normalize(viewPos) - Frag-
        Pos);
        vec3 reflectDir = reflect(-lightDir, norm);
        float spec = pow(max(dot(viewDir, reflectDir), 0.0),
        8);
        vec3 specular = specularStrength * spec *
        lightColor;
        vec3 result = (ambient + diffuse + specular) * tex-
        ture(tex, Texcoord*6).rgb;
        ...
        outColor = vec4(result, 1.0);
}
```

Heijastuskomponentin voimakkuus määritellään muuttujassa, jonka jälkeen lasketaan katsojan suuntavektori sekä heijastusvektori. Kaikki vektorit normalisoidaan tarvittaessa. Heijastusvektorin kaavassa valon suuntavektori muutetaan käänteiseksi siitä syystä, että reflect-funktio odottaa vektorin kulkevan valonlähteestä kohti sirpaletta, mutta edeltävien laskutoimitusten takia vektori kulkee päinvastaiseen suuntaan. Kun nämä vektorit on selvitetty, voidaan heijastuskomponentti laskea niiden pistetulosta. Pistetulo nostetaan halutun kiiltävyyden perusteella valittuun potenssiin, jonka kasvaessa heijastusvaikutus keskittyy voimakkaammin. Heijastus lisätään tulokseen muun valaistuksen rinnalle. (De Vries 2017, 121–122.) Tähän mennessä valaistusarvot on lisätty kappaleelle määritettyyn väriin, mutta sama pätee tekstuureihin, kuten yllä on esitetty.

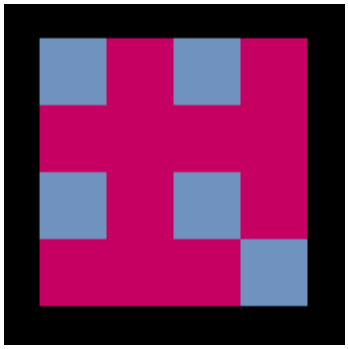
5 ESIMERKKISOVELLUS

Opinnäytetyön teoriapohjaa laatiessani kehitin kirjoitustyön rinnalla tietokonesovelluksen, jonka tarkoituksena oli soveltaa oppimaani käytäntöön ja syventää ymmärrystäni aiheesta. Käytän ensisijaisesti menetelmiä ja järjestystä, jotka perustuvat edellisessä pääluvussa esiintyvään koodipohjaan. Sovelluksesta otetut kuvakaappaukset käsittävät vaiheittain kuutio-objektin luonnin ja tarpeellisten tilan ominaisuuksien määrityksen. Kappaleen rakennus voidaan aloittaa määrittämällä kuution ensimmäisen tahkon ominaisuudet. Yksivärisen neliön kärkidataan tulee sisällyttää sijaintikoordinaatit sekä RGB-muotoinen värikoodi, jotka esimerkissä johtavat kuvan 1 esittämään muotoon.

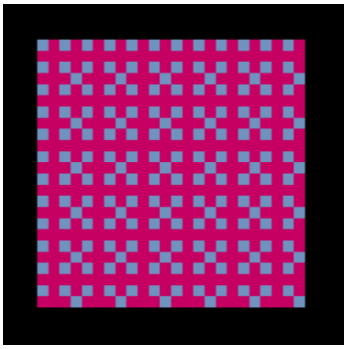


KUVA 1. Yksivärinen neliö

Tekstuurikoordinaattien lisäyksen jälkeen tuodaan SOIL-kirjastolla koodiin yksinkertainen koetekstuuri, joka korvaa edellisen väridatan. Kertomalla tekstuurin koordinaatit kokonaisluvulla saadaan aikaiseksi vaikutelma toistuvasta kuviosta, joka käyttäytyy asetettujen parametrien mukaisesti. Kuvassa 2 tekstuuri kattaa aluksi koko muodon, sitten toistuu molempien akseleiden suuntaisesti ilman suodatusta kuvassa 3 ja lopulta kiinnittyy r-akselin suuntaisesti reunaan bilineaarisella suodatuksella kuvassa 4. Seuraavia vaiheita varten palataan kuvan 3 suodattamattomaan versioon.



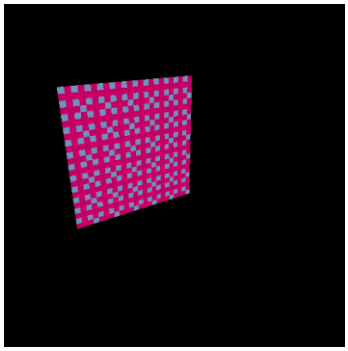
KUVA 2. Neliö tekstuurilla



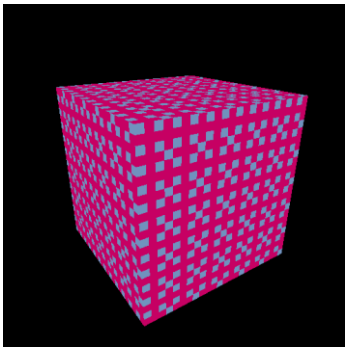
KUVA 3. Toistuva tekstuuri

KUVA 4. Parametrien muutos

Syvyysvaikutelman saavuttamiseksi näkymää manipuloidaan matriisimuunnoksilla. Itse mallia ei tässä tapauksessa ole tarpeellista siirtää, mutta halutessa mallimatriisille voidaan suorittaa mm. skaalaus-, kierto- tai siirto-operaatioita. Näkymän virtuaalista kameraa siirretään taaksepäin ja käännetään y-akselin ympäri. Projektiolle määritellään 80°:n näkökenttä ja ikkunan resoluution mukainen kuvasuhde sekä sopivat lähi- ja etäpintojen arvot. (KUVA 5.) Kuution loput tahkot muodostavat kärjet lisätään tilakoordinaatteina kärkitaulukkoon. Tekstuurin koordinaattidata myös kopioidaan muille tahkoille, jolloin se kattaa koko kappaleen. (KUVA 6.)

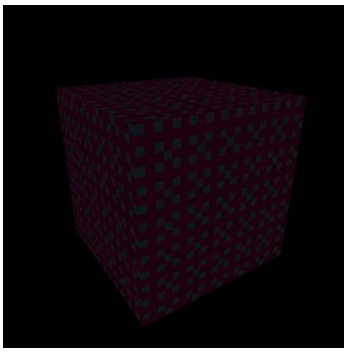


KUVA 5. Matriisimuunnokset

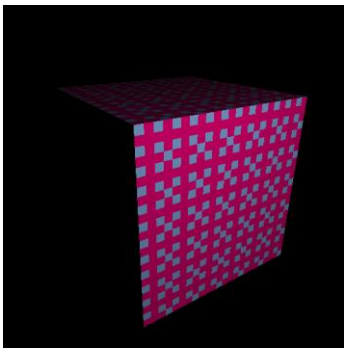


KUVA 6. Kuutio

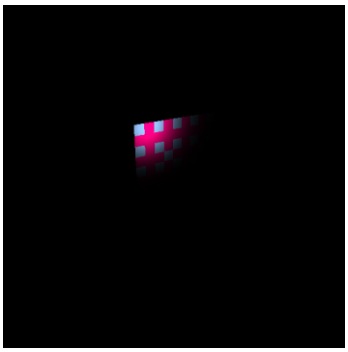
Näkymälle asetetaan heikko ympäröivä valaistus, joka kattaa tasaisesti koko kappaleen. (KUVA 7.) Kuvassa 8 lasketaan erikseen suunnattu hajaantunut valaistus ja kuvassa 9 katsojan sekä valonlähteen suuntien perusteella määritetty heijastustehoste. Valonlähde sijaitsee suhteessa kappaleeseen sen katsojalle lähimmän tahkon puolella, hieman kohotettuna y-akselilla ja kierrettynä kuution oikealle puolelle. Alla olevan kuvasarjan valaistustasot on laskettu erillään toisistaan ja kukin on kerrottu tekseleiden väridatalla.



KUVA 7. Ympäröivä valo

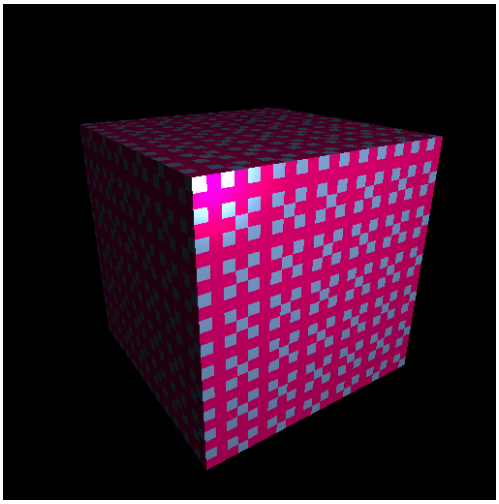


KUVA 8. Hajaantunut valo



KUVA 9. Heijastustehoste

Yhdistämällä valaistuskomponenttien summa piirrettyyn kuutioon saadaan aikaiseksi kuvassa 10 esiintyvä kappale, joka vaikuttaa sijaitsevan kolmiulotteisessa tilassa. Tilassa on myös virtuaalinen kappaleen valaiseva valonlähde. Vaikutelma on realistinen, vaikkakin kuutio esiintyy tyhjässä mustassa tilassa. Kaikki teoriapohjassa käsitellyt alueet on tuotu tässä viimeisessä piirron vaiheessa käytäntöön ja niitä voidaan soveltaa minkä tahansa muun muotoisen tai tekstuurisen kappaleen piirtoon sekä valaistuksen määritykseen. Sovelluksen ohjelmoijalle tämä edustaa mahdollista lähtöpistettä, josta grafiikkaa vaativaa toiminnallisuutta voidaan lähteä kehittämään.



KUVA 10. Lopullinen kappale

6 JOHTOPÄÄTÖKSET

Pyrin opinnäytetyössä saamaan tarpeeksi tietoa OpenGL:n toiminnoista pystyäkseni ohjelmoimaan yksinkertaisen grafiikkasovelluksen ja soveltamaan hankittua tietämystä mahdollisissa tulevaisuuden työtehtävissä. Suoritin tutkimusta lukemalla kirjallisia lähteitä sekä etsimällä tietoa Internetistä ja samanaikaisesti kirjoittamalla koodia testiohjelmaa varten. Toimiva, vaikkakin hyvin yksinkertainen esimerkkisovellus oli opinnäytetyön tuotos, jonka rakentamisen vaiheet esittelin teoriapohjaa apuna käyttäen. Sain lähteistäni odottamaani vähemmän tietoa grafiikkalaitteiston toiminnasta, joka liittyi yhteen tutkimuskysymyksistäni, mutta itse ohjelmointirajapinnan operaatiot tulivat opinnäytetyön edetessä hyvin selville. Opin hyödyntämään kaikkia sovelluksen ohjelmointiin tarvittavia työkaluja ja minulle tuli selväksi OpenGL-järjestelmän rakentamisen työnkulku sekä sen mutkikkuudet.

OpenGL soveltuu kokemukseni perusteella erittäin hyvin moneen käyttötarkoitukseen. Ohjelmointirajapinnan monipuolisuuden takia sitä voidaan hyödyntää lähes missä tahansa sovelluksessa, jossa grafiikan piirto on tarpeellista. Vaikka ohjelmoija joutuu tekemään käytännössä paljon enemmän töitä mallinnusohjelmiin tai pelimoottoreihin verrattuna, on avoimuus ja vapaus suuri hyöty. Videopelien lisäksi käyttökohteita voi olla esimerkiksi lääketieteen, rakennustekniikan, automaation tai metallitekniikan alueilla.

Mielestäni käyttämäni menetelmät ovat tällaisessa projektissa toimivia, mutta aiheesta ja omista olosuhteistani johtuen tapauksessani työn valmistuminen viivästyi pahasti. Lähteestä toiseen siirtyminen saman osa-alueen kohdalla ja tiedon ristiin vertailu oli hyvin rasittavaa, varsinkin tapauksissa, joissa saman päämäärän saavuttamiseen on useita mahdollisuuksia. Itselleni toimivan toimintatavan valitseminen määräytyi pitkälti ohjelmakoodin kautta, sillä ohjelman suorituksen ongelmanselvityksessä jouduin usein etsimään vaihtoehtoisia menetelmiä. Huomasin työn kulun hidastuvan voimakkaasti näissä tapauksissa ja kamppailin usein motivaation kanssa päästäkseni eteenpäin ongelmien ilmetessä. Näistä vaikeuksista huolimatta pidän lopputulosta hyvänä edustuksena grafiikkaohjelmoinnin kokonaisprosessista. Aihealue oli yllättävän haastavaa ymmärrettävää varsinkin siksi, koska ennen opinnäytetyön aloitusta minulla ei ollut juuri lainkaan tietoa työssä käsitellyistä asioista, mutta varmaankin tästä näkökulmasta johtuen tuotettu teksti on melko helposti ymmärrettävää sekä perinpohjaista.

Opin opinnäytetyötä tehdessäni OpenGL-järjestelmän perusteita, kuten tietoa varjostinohjelmista, 3D-mallinnuksesta sekä virtuaalisesta valaistuksesta ja näkökulmasta. Itse aihealueen tietämyksen lisäksi

pääsin käyttämään mm. versionhallinnan kaltaisia menetelmiä, joista on hyötyä ohjelmointitöissä yleisesti. Oppimani tulee mielestäni esille tekstissäni hyvin ja työ on jäsennetty loogisesti, mutta koodiesimerkeistä riippuvuus varmaankin hieman heikentää lukukokemusta. Tästä huolimatta pidän koodin sisällytyksiä tarpeellisina, sillä niistä selviää konkreettisesti tekstin esittämät asiat ja ne toimivat käytännön esimerkkeinä.

Kokonaisuutena opinnäytetyö sujui prosessin osalta kankeasti, mutta olen tyytyväinen lopputuloksiin. Opitun hyödyllisyys on kiistanalaista, sillä helpompia vaihtoehtoja grafiikan piirtoon on olemassa, mutta pidän tietämystä grafiikkalaitteiston matalan toiminnasta ohjauksesta ohjelmointirajapinnalla kuitenkin tarpeellisena alan perustietona. Tutkimustyön suoritus oli lopulta palkitsevaa ja tämä välittyy tekstissä mielestäni kohtuullisen hyvin.

LÄHTEET

Shreiner, D., Sellers, G., Kessenich, J. & Licea-Kane, B. 2013. OpenGL programming guide: the official guide to learning OpenGL, version 4.3. 8. painos. New Jersey: Pearson Education, Inc.

Rodríguez, J. 2013. GLSL Essentials. 1. painos. Birmingham: Packt Publishing Ltd.

Mitchell, S. 2013. SDL Game Development. 1. painos. Birmingham: Packt Publishing Ltd.

GLEW: The OpenGL Extension Wrangler Library. Verkkosivusto. Viitattu 16.1.2017. http://glew.sourceforge.net/index.html.

Riccio, C. 2016. GLM Manual. Saatavissa: http://glm.g-truc.net/0.9.8/glm-0.9.8.pdf. Viitattu 16.1.2017.

lonesock.net: SOIL. Verkkosivu. Viitattu 16.1.2017. http://www.lonesock.net/soil.html.

Deitel, P., Deitel H. 2016. C How to Program. 8. painos. Essex: Pearson Education Limited.

Overvoorde, A. 2017. Modern OpenGL Guide. Saatavissa: https://raw.githubusercontent.com/Overv/Open.GL/master/ebook/Modern%20OpenGL%20Guide.pdf. Viitattu 21.8.2017.

De Vries, J. 2017. Learn OpenGL. Saatavissa: https://learnopengl.com/book/learnopengl_book.pdf. Viitattu 3.9.2017.

Chacon, S., Straub, B. 2014. Pro Git. 2. painos. New York: Apress Media, LLC.