Osaamista
ja oivallusta
tulevaisuuden
tekemiseen

Jonathan Lakatos

Grafiikka-algoritmien implementointi

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikka

Insinöörityö

21.4.2020

Tekijä Otsikko Sivumäärä Aika	Jonathan Lakatos Grafiikka-algoritmien implementointi 35 sivua 21.4.2020
Tutkinto	Insinööri (AMK)
Tutkinto-ohjelma	Tieto- ja viestintätekniikka
Ammatillinen pääaine	Pelisovellukset
Ohjaaja	Lehtori Miikka Mäki-Uuro

Insinöörityössä perehdyttiin tietokonegrafiikan toimintaan ja pureuduttiin grafiikan piirtämisen eri vaiheisiin. Työssä selvitettiin, miten näytönohjaimia hyödynnetään ja miten grafiikan prosessointia voidaan näytönohjaimissa ohjailla varjostinohjelmien avulla. Työssä tutustuttiin myös tietokonegrafiikan historiaan ja siihen, miten se on kehittynyt vuosien varrella. Lisäksi tutustuttiin piirrettävän datan, kuten 3D-mallien ja tekstuurien rakenteeseen, ja selvitettiin, mitä asioita tulee ottaa huomioon, kun näitä halutaan käsitellä näytönohjaimen puolella.

Insinöörityön osana toteutettiin grafiikkasovellus käyttäen C++-ohjelmointikieltä ja OpenGL-rajapintaa. Sovelluksessa luotiin 3D-objekteja, jotka piirrettiin ohjelman luomaan ikkunaan. Ohjelmassa 3D-objektin data vietiin näytönohjaimelle, jossa sitä käsiteltiin verteksi- ja fragmenttivarjostimilla. Sovelluksessa myös tutustuttiin siihen, miten verteksivarjostimessa voidaan matriisien avulla muuttaa näkymän suuntaa ja perspektiiviä tai 3D-objektien kokoa, rotaatiota tai sijaintia.

Sovellukseen toteutettiin myös erilaisia tekstuurienluontifunktioita, kuten Perlinin ja Worleyn kohinatekstuurin luontifunktiot. Sovelluksessa myös hyödynnettiin fraktaaleita erilaisten kuvien luomiseen ja pohdittiin, miten niitä voisi hyödyntää erilaisen tietokonegrafiikan luomisessa. Sovellukseen luotiin myös tekstuurinkäsittelyfunktioita, kuten piirtofunktio, jolla voidaan käsitellä yksittäisiä tekstuurin pikseleitä, tai kontrastifunktio, jolla voidaan muuttaa koko tekstuurin kontrastia.

Sovelluksen kehittäminen kerrytti käytännön kokemusta siitä, miten OpenGL-rajapinnan avulla viedään 3D-objektien ja tekstuurien dataa näytönohjaimelle, sekä siitä, miten varjostimia luodaan ja mitkä ovat niiden funktiot.

Avainsanat	tietokonegrafiikka, OpenGL, varjostin, 3D, tekstuurit

Author Title Number of Pages Date	Jonathan Lakatos Implementation of Graphic Algorithms 35 pages 21 April 2020
Degree	Bachelor of Engineering
Degree Programme	Information and Communications Technology
Professional Major	Game Applications
Instructor	Miikka Mäki-Uuro, Senior Lecturer

The purpose of this thesis was to dive deep into the workings of computer graphics and the different phases of the process of drawing pictures to computer screen. The study also focuses on how graphics cards are utilized and how graphics can be manipulated in graphics cards with shader programs. Part of this project work was to research the history of computer graphics and how it has evolved over the years. Finally, this study analysed how graphical data like 3D-models and texture are constructed and what we need to consider while using these with graphics cards.

As a part of the thesis a program was developed using C++ and OpenGL API. The program creates 3D-objects and draws those to a window. The program passes the data of the 3D objects to the video card where it is used by vertex- and fragment shaders. In the program we also explore how matrices are used to change view, perspective and transformation of 3D objects.

Different types of texture generation functions were created for the program for example functions for creating Perlin and Worley noise textures. The program also uses fractals to create images and, in the thesis, we think about different types of graphics solutions that fractals could be used in. Also, some simple texture manipulation functions such as drawing, adding contrast and combining two textures into a completely new texture were also created for the program.

The program was helpful with acquiring some practical experience working with OpenGL API and how it manages graphical data and its memory allocation to graphic cards' memory.

Keywords	Computer graphics, OpenGL, Shader, 3D, textures

Sisällys

Lyhenteet

1 Johdanto 1

2 Tietokonegrafiikka 1

2.1 Historia lyhyesti 2
2.2 Grafiikan ohjelmointirajapinnat 5
2.3 Grafiikan liukuhihna 6
2.4 Näytönohjaimet 8
2.5 3D-mallit 9
2.6 Tekstuurit 11

3 Grafiikkasovelluksen toteutus 14

3.1 Tavoite ja toteutetut metodit 14
3.2 Ohjelmointirajapinnat ja kirjastot 14
3.3 3D-mallien toteutus ja piirtäminen 15
3.4 MVP-matriisit 16
3.5 Varjostimet 18
3.6 Tekstuurin käsittely ja generointi 19
3.6.1 Kohina 20
3.6.2 Fraktaalit 23
3.6.3 3D-tekstuuri 24
3.6.4 Tekstuuriin piirtäminen 25
3.6.5 Tekstuurin manipulointi 27
3.7 Projektin pohdinta 28

4 Yhteenveto 29

Lähteet 31

Lyhenteet

API	Application Programming Interface. Ohjelmointirajapinta
BIOS	Basic Input/Output System
BPP	Bits Per Pixel
CAD	Computer-Aided Design
CGA	Color Graphics Adapter
CPU	Computer Processing Unit
CSG	Constructive Solid Geometry
EGA	Enchanted Graphics Adapter
GLFW	Graphic library framework.
GLM	OpenGL Math -kirjasto
GPU	Graphic Processing Unit. Grafiikkaprosessori.
GLSL	OpenGL shading language.
HDMI	High-Definition Multimedia Interface
HLSL	High-Level Shading Language
MSL	Metal Shading Language
MVP	Model View Projection

OpenGL	Open graphics library.
PNG	Portable network graphics. Bittikarttagrafikan tallennusformaatti.
RAMDAC	Random Access Memory Digital to Analog Converter
RGBA	Red Green Blue Alpha
SPIR-V	Standard Portable Intermediate Representation
VGA	Video Graphics Array

1 Johdanto

Nykyään tietokonegrafiikka on merkittävässä roolissa jokapäiväisessä elämässä, mutta harvemmin tullaan ajatelleeksi, miten tietokonegrafiikka muodostuu tietokoneiden näytöille. Tässä insinöörityössä on tästä otettu selvää. Mielenkiinto aihetta kohtaan on syntynyt, koska se, miten tietokone grafiikkaa piirtää, on jäänyt epäselväksi, vaikka työskenteleekin paljon tietokoneiden ja erilaisten visuaalisten ohjelmien, kuten pelisovelluksien, kehityksen parissa. Myös parempi ymmärrys siitä, miten tietokonegrafiikka muodostuu, auttaa luomaan parempia visuaalisia ohjelmia ja helpottaa grafiikan muokkaamista.

Insinöörityö käsittelee pääpiirteittäin tietokonegrafiikan historian, piirtämiseen käytettävät työkalut ja sen, mistä erilainen piirrettävä data, kuten 3D-objektit ja tekstuurit, rakentuvat. Tavoitteena oli saada hyvä yleiskäsitys siitä, miten grafiikkaa piirtävä teknologia on kehittynyt ja miten se toimii nykyään. Insinöörityötä varten luotiin myös OpenGL:ää käyttävä C++-sovellus, jolla pystytään piirtämään tietokonegrafiikkaa ja luomaan erilaista piirrettävää dataa, kuten tekstuureita, joita voidaan hyödyntää esimerkiksi peligrafiikan ja erilaisten efektien luomisessa. Sovelluksen tavoitteena oli antaa käytännön kokemusta tietokonegrafiikan luomisesta ja toimia hyvänä pohjana erilaisten graafisten työkalujen luomiseen.

Luvussa 2 perehdytään yleisesti tietokonegrafiikan ja graafisen datan toimintaan ja tutustutaan grafiikkaa piirtäviin ohjelmointirajapintoihin ja näytönohjaimiin. Luvussa 3 taas käydään läpi sitä, miten työtä varten luotu sovellus toimii, ja saadaan käytännön kokemusta siitä, miten tietokonegrafiikka toimii OpenGL-rajapinnan kanssa. Tässä luvussa selvitetään myös, miten luodaan erilaisia metodeja generoimaan graafista dataa. Luvussa 4 käydään läpi, mitä saatiin tehtyä, ja pohditaan, mitä olisi voitu tehdä paremmin.

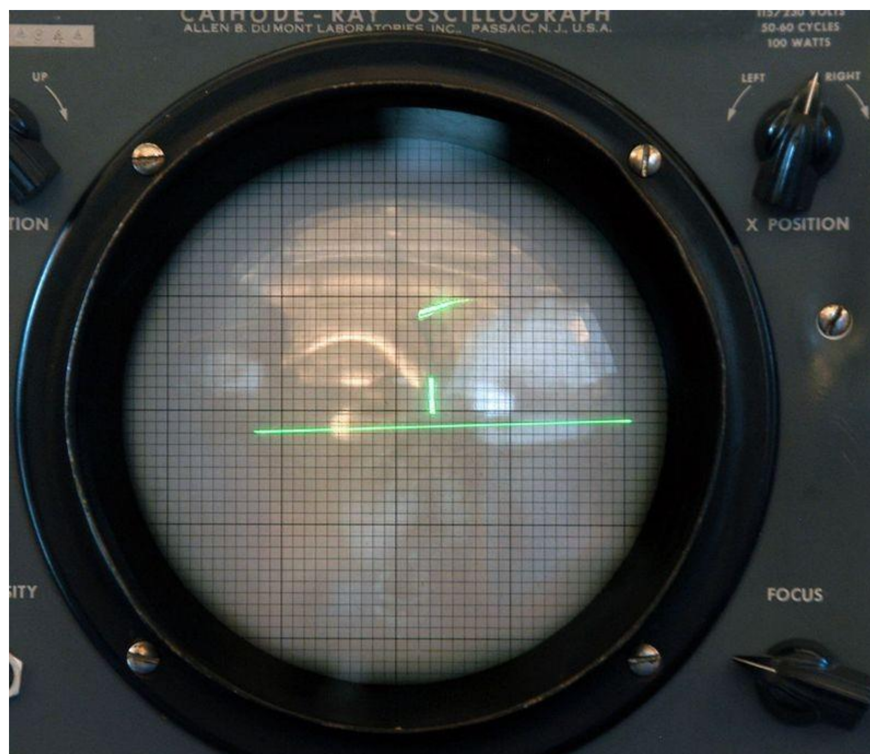
2 Tietokonegrafiikka

Tietokonegrafiikalla tarkoitetaan kaikkea tiekoneen luomaa grafiikkaa. Useimmiten tämä mielletään tietokoneen näytölle piirrettyinä kuvina. Tässä luvussa tutustutaan lyhyesti tietokonegrafiikan historiaan ja selvitetään, mistä kaikki on alkanut ja mitä eri vaiheita tämän teknologian kehitys on käynyt läpi vuosien varrella. Sitten tutustutaan lähemmin

ohjelmointirajapintoihin ja niiden rooliin kuvan piirtämisessä. Tämän jälkeen käydään läpi piirtämisen eri vaiheita ja tutustutaan tarkemmin niiden toimintaan. Sitten perehdytään myös tarkemmin piirtämiseen käytettävään laitteistoon eli näytönohjaimiin ja siihen, mistä nämä koostuvat. Lopuksi käydään vielä läpi, mistä piirrettävä data, kuten 3D-mallit ja tekstuurit, muodostuvat ja mitä tulee ottaa huomioon niitä käsitellessä.

2.1 Historia lyhyesti

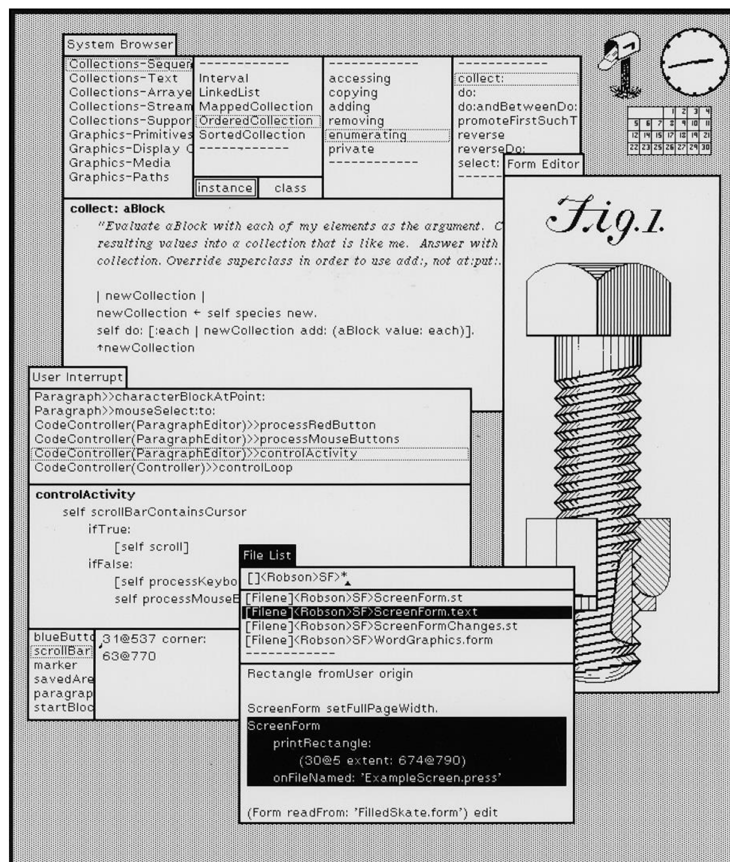
Ensimmäiset tietokoneet olivat nykyisiin tietokoneisiin verrattuna hyvin yksinkertaisia ja niiden käyttö perustui datan prosessointiin, jonka tulokset koneet tallensivat esimerkiksi magneettinauhoille tai reikäkorteille. Tällöin tietokoneissa ei vielä ollut paljoakaan tehoa, joten ne eivät pystyneet silloin vielä visualisoimaan paljoakaan käyttäjälle. Tietokoneet alkoivat piirtää kuvia näytöille vasta 1950-luvulla. Tällöin tietokonegrafiikka esiintyi esimerkiksi oskilloskoopeille piirrettyinä kuvina (kuva 1). [1.]



Kuva 1. William Higinbothamin vuonna 1958 kehittämä oskilloskoopille piirretty Tennis for two -peli [2].

Näytöt alkoivat tietokoneissa yleistyä 1960-luvulla. Ne olivat tällöin vielä pieni osa tietokoneen toimintaa. Pääosin tietokoneita ohjailtiin erillisiltä terminaaleilta, jotka näihin aikoihin olivat esimerkiksi kaukokirjoittimia. Vuonna 1963 julkaistiin Ivan Sutherlandin kehittämä Sketchpad-ohjelma, jolla pystyi valokynällä piirtämään näytölle viivoja ja muotoja. Sketchpadillä pystyttiin myös luomaan kolmiulotteisia muotoja, joita pystyi siirtelemään ja kääntämään. [3; 4.]

1970-luvulla mikroprosessorien ansiosta tietokoneista tuli pienempiä ja tehokkaampia. Parempi laskentateho johti myös siihen, että tietokoneiden näytöt vaihtuivat vektorikuvia piirtävistä näytöistä rasterinäyttöihin, joissa kuva muodostuu lukuisista pikseleistä. Vuonna 1973 julkaistiin Xerox Alto, joka oli ensimmäinen tietokone, jonka käyttöjärjestelmä tuki graafista käyttöliittymää (kuva 2). Xerox Alton kuva muodostui 606 x 808 mustista tai valkoisista pikseleistä, ja tämän graafinen käyttöliittymä sisälsi ikkunoita, kuvakkeita, valikkoja ja osoittimet, jotka ovat yhä keskeisiä moderneissa graafisissa käyttöliittymissä. [5; 6.]



Kuva 2. Xerox Alton graafinen käyttöliittymä [7].

1970-luvulla syntyivät myös ensimmäiset pelihallien videopelit, kuten Pong (1972) ja Gun Fight (1974). Nämä arcade-pelit olivat graafisesti hyvin edistyneitä, sillä toisin kuin senaikaisissa tietokoneissa, nämä pelikoneet pystyivät käyttämään enemmän laskentatehoa ja muistia pelien graafiseen ilmeeseen. Monilla pelikoneilla oli myös grafiikkaa varten omaa laitteistoa, esimerkiksi suuren suosion saavuttanut Space Invaders (1978) käytti Fujitsun MB14241-näytönkontrolleria kiihdyttämään Sprite-kuvien piirtämistä. [8; 9.]

Tietokoneet alkoivat 1980-luvulla myös käyttää grafiikkaa varten tehtyä laitteistoa. Ne keskittyivät erityisesti tehostamaan kaksiulotteisen kuvan näytölle piirtämistä ja parantamaan kuvien laatua. 1980-luvulla kehittyi erityisesti tietokoneiden kyky näyttää eri värejä näytöllä. Esimerkiksi vuonna 1981 kehitetyllä CGA-standardilla pystyttiin näyttämään 4 eri väriä 320 x 200 -resoluutiolla, mutta vuoteen 1987 mennessä kehitetyllä VGA-standardilla pystyttiin näyttämään näytöllä samaan aikaan 16 eri väriä 640 x 480 -resoluutiolla. [10; 11.]

1990-luvulla 3D-grafiikka alkoi yleistyä kotikoneissa. Tämä johtui sekä kehittyvästä ohjelmistosta että paremmasta laitteistosta. Vuonna 1992 SGI, joka tuotti graafisia työasemia, julkaisi OpenGL-ohjelmointirajapinnan. OpenGL tehosti 2D- ja 3D-kuvan tuottamista ja standardisoi grafiikkalaitteiston käsittelyä. Tämä johti siihen, ettei jokaiselle laitteistolle tarvinnut kirjoittaa uutta ohjelmistoa, vaan laitteiston ajurit toimivat yhteen OpenGL:n kanssa. [12.]

1990-luvun puolessavälissä julkaistiin viidennen sukupolven pelikonsolit, jotka erosivat aiemmista sukupolvista enimmäkseen siten, että ne tukivat 3D-grafiikkaa [13]. Samoihin aikoihin alkoi tietokoneille ilmestyä 3D-grafiikkaa tukevaa laitteistoa, kuten nvidian NV1 ja 3Dfx:n Voodoo [14]. Nämä mahdollistivat reaaliaikaisen 3D-grafiikan myös kotikoneilla. Uudet pelikonsolit ja näytönohjaimet johtivat suureen 3D-median kasvuun pelien ja elokuvien osalta.

Näytönohjaimet ja niitä ohjailevat rajapinnat ovat 2000-luvun alusta kehittyneet tehokkaammiksi ja paremmiksi. Nykyään pystytään reaaliaikaisesti piirtämään näytölle hyvin realistista grafiikkaa. Tämä kuitenkin yleensä vaatii erilaisia temppuja, joilla usein ’huijataan’ asiat näyttämään paremmilta käyttämällä vähemmän laskentatehoa. Vaikka tietokonegrafiikan kanssa on päästy hyvin pitkälle, on siinä vielä paljon kehitettävää.

2.2 Grafiikan ohjelmointirajapinnat

Ohjelmat suoritetaan tietokoneen suorittimella, mutta grafiikan suorittamiseen useimmiten halutaan käyttää näytönohjaimen suoritinta, joka on tehokkaampi käsittelemään piirrettävää dataa. Koska näytönohjaimien kanssa ei voida kommunikoida suoraan, tarvitaan avuksi ohjelmointirajapintaa. Grafiikan ohjelmointirajapintojen eli grafiikan API:en (engl. Application Programming Interface) päätehtävä on ohjelman ja näytönohjaimen välillä kommunikoiminen. Tämä tapahtuu siten, että API kommunikoi näytönohjaimen ajurin kanssa, joka kääntää API:n välittämän komennon näytönohjaimen komennoksi. Tämä vaatii sen, että näytönohjaimen ajureissa on implementoituna komennot, joita API:lla kutsutaan. [15.] Moderneja grafiikan ohjelmointirajapintoja on muutamia erilaisia, ja ne eroavat toisistaan lähinnä sen mukaan, mitkä käyttöjärjestelmät ja näytönohjaimet tukevat niitä. Esimerkiksi DirectX toimii Windows-käyttöjärjestelmällä, Metal iOS:llä ja OpenGL ja Vulkan toimivat kaikilla suurilla käyttöjärjestelmillä. [16.]

Viime aikoina graafisten API:en kehityksen suuntana on ollut vähentää abstraktiota ja päästää kehittäjä lähemmäs näytönohjaimen toimintaa. Tällaisia API:ja ovat Vulkan, Metal ja DirectX 12. Näiden API:en laiteläheisyys antaa paremman kontrollin ohjelmointiin, mikä usein myös johtaa tehokkaampiin ohjelmiin. Tällöin kuitenkin ohjelmointirajapinta tekee vähemmän tehtäviä automaattisesti, mikä hankaloittaa ohjelmointia jonkin verran. [17; 18.]

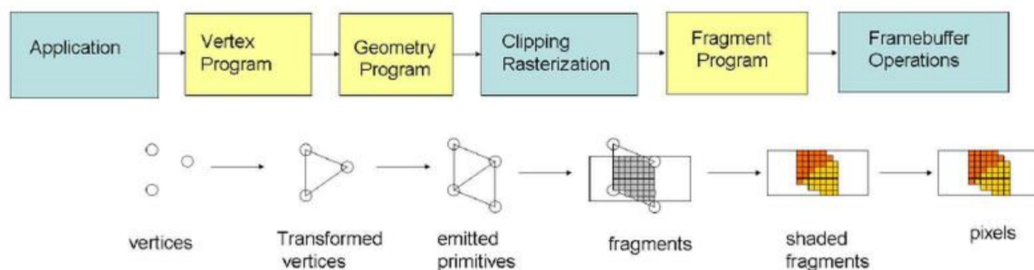
API:en tehtävänä on myös kertoa, miten näytönohjaimelle välitettyä dataa tulisi käsitellä varjostinohjelmien (engl. Shader) avulla. Varjostinohjelmia käsitteleviä ohjelmointikieliä on muutamia erilaisia ja eri grafiikan API:t tukevat eri kieliä. Esimerkiksi Metal käyttää MSL:ää, OpenGL GLSL:ää ja DirectX HLSL:ää. Vulkan puolestaan käyttää SPIR-V-nimistä tavukoodia. Se ei ole kovin luettavaa, minkä takia on luotu kääntäjä, joka kykenee kääntämään GLSL:ää ja HLSL:ää SPIR-V:ksi. [19; 20.]

Varjostinohjelmointikielet muistuttavat paljon muita ohjelmointikieliä ja pystyvät suorittamaan IF- ja FOR-lauseita ja muita yleisiä operaatioita. Suurimpana erona muihin kieliin on se, että varjostinkieliset ohjelmat suoritetaan näytönohjaimen suorittimella. Suuri osa varjostinkielien standardifunktioista käsittelee vektori- ja matriisilaskentaa ja osa funktioista on käytettävissä vain tietyissä piirtämisen vaiheissa. Varjostinohjelmilla on myös omat datatyypit esimerkiksi matriiseja ja vektoreita varten. Varjostinohjelmissa pitää

myös ottaa huomioon, että ne saavat datan tietokoneen suorittimella suoritettavalta ohjelmalta puskureiden tai yhteisten muuttujien muodossa. Puskurit ovat listoja alustamatonta dataa, ja yhteiset muuttujat ovat varjostinohjelmalle globaaleja muuttujia, jotka pysyvät samana koko piirtokutsun ajan. [21.]

2.3 Grafiikan liukuhihna

Grafiikan piirtäminen ruudulle käy läpi monta eri vaihetta liukuhihnamaisesti tehden eri toimia eri vaiheissa (kuva 3). Liukuhihnan ensimmäinen vaihe on piirrettävän datan alustaminen ja sen lähettäminen näytönohjaimelle. Piirrettävä data on useimmiten 3D-malleja ja tekstuureita, jotka voidaan joko luoda käytettävässä ohjelmassa tai tuoda esimerkiksi tietokoneen muistista. Tämä data tallennetaan näytönohjaimen muistiin käyttäen apuna ohjelmointirajapintaa, joka kommunikoi näytönohjaimen kanssa. 3D-objekteista muodostetaan puskureita, eli alueita muistia, jotka sisältävät objektin verteksien tiedot. Verteksit ovat pisteitä, joista 3D-objektit koostuvat. Verteksin dataan kuuluu aina sen sijainti, ja tämän lisäksi siihen voivat esimerkiksi kuulua verteksien normaalivektorit ja tekstuurin koordinaatit. Tekstuurit tallennetaan myös näytönohjaimen muistiin. Tekstuureita viedessä voidaan myös kertoa, miten tekstuureita tulee käsitellä, ja asettaa niille mipmap-tasot, jotka auttavat tekstuurin suodattamisessa. Ohjelmasta viedään myös varjostinohjelmat näytönohjaimelle. Niillä ohjaillaan sitä, miten lähetettyä dataa käsitellään näytönohjaimen puolella. Ohjelmassa voidaan lähettää näytönohjaimelle myös muuta piirtämistä avustavaa dataa, kuten valojen tietoja, MVP-matriiseja, jotka ohjailevat ohjelman 'kameraa', tai muuta dataa. [22.]



Kuva 3. Grafiikan liukuhihnan vaiheet järjestyksessä ohjelmalta ruudulle [23].

Varjostimet ovat ohjelmia, jotka käsittelevät piirrettävää dataa näytönohjaimessa. Objektien piirtäminen vaatii useamman varjostinohjelman, joista eri ohjelmat käsittelevät eri

Metropolia

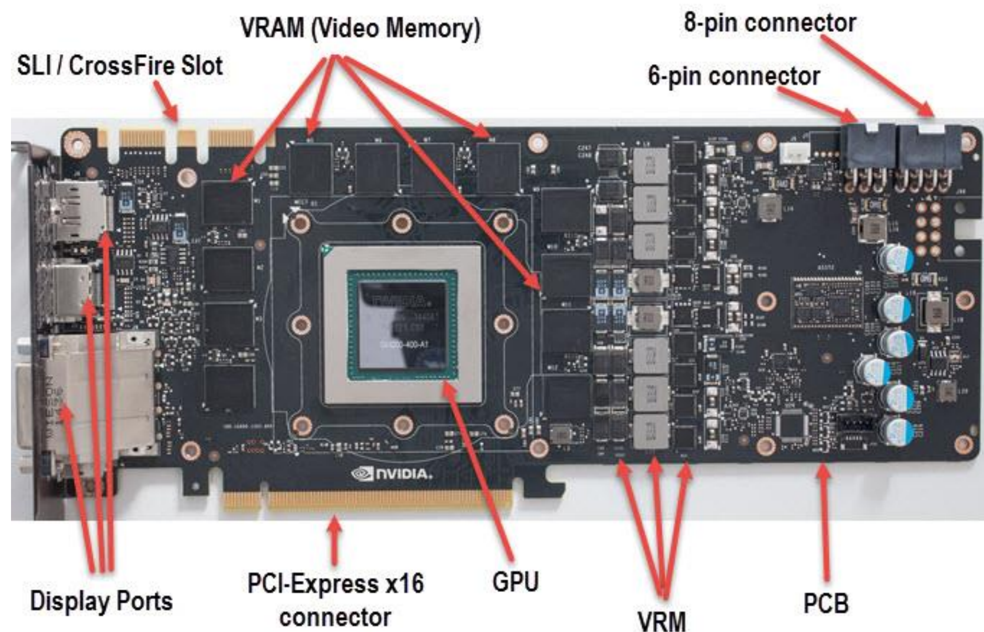
piirtämisen vaiheita. Varjostinohjelmia ovat esimerkiksi verteksi-, geometria-, tesselaatio- ja fragmenttiohjelma, joista verteksi- ja fragmenttiohjelmat ovat pakollisia. Varjostinohjelmat voivat saada useita sisääntuloarvoja ja lähettää useampia ulostuloarvoja. Varjostinohjelman ulostuloarvot toimivat liukuhihnassa seuraavan varjostinohjelman sisääntuloarvoina. [22.]

Näytönohjaimelle lähetetty data, kuten verteksipuskurit ja yhteiset muuttujat, menevät ensin verteksiohjelmalle, joka on varjostinohjelma, joka käy läpi kaikki objektin verteksit. Tässä vaiheessa esimerkiksi muutetaan verteksin sijainti paikallisesta koordinaatistosta näkymän koordinaatistoon käyttämällä MVP-matriiseja tai lasketaan verteksien valotusta. Vertekseille lasketut uudet koordinaatit ja väri-, valaistus- tai muut arvot viedään ulostulona seuraavalle ohjelmalle. Verteksivarjostimen jälkeen voidaan tarvittaessa ohjata data tesselaatiovarjostimelle, jossa voidaan jakaa tasoja pienemmiksi paloiksi, tai geometriavarjostimelle, jossa voidaan luoda objektille lisää geometriaa. Näitä varjostimia ei ole pakko olla ohjelmassa, jolloin ne voidaan jättää pois, mikäli niitä ei käytetä. Verteksikäsittelyn jälkeen siirrytään verteksien jälkikäsittelyvaiheeseen, jossa kootaan vertekseistä kolmioita ja leikataan pois kolmiot ja verteksit, joiden tiedetään jäävän piirtämättä. Tällaisia ovat esimerkiksi kolmiot, jotka ovat kuvan ulkopuolella tai jäävät kameraan nähden väärinpäin. Tämän jälkeen muodostetut kolmiot rasteroidaan eli niistä muodostetaan pikselin kokoisia fragmentteja. Koska fragmentit saavat arvoja useammalta verteksiltä, joudutaan fragmenteille interpoloimaan uudet arvot. [22.]

Fragmentit menevät fragmenttivarjostimeen, jossa pikseli saa itselleen lopullisen väriarvon. Samalla seurataan fragmenttien syvyysarvoja ja jätetään laskematta fragmentit, jotka jäävät muiden fragmenttien taakse. Tämä toimii z-puskurointitekniikalla (Z-buffering), jossa piirretyistä fragmenteista otetaan talteen sen etäisyys kamerasta. Jos piirrettävän fragmentin sijainnilla on jo z-puskurissa arvo, sitä verrataan piirrettävän fragmentin etäisyyteen kamerasta. Jos puskurissa oleva arvo on lähempänä kameraa kuin käsiteltävän fragmentin, se jätetään piirtämättä. Tällä tavalla voidaan välttää turhaa työtä ja säästää prosessointitehoa. Fragmenttien arvot tallennetaan sitten kuvapuskuriin, joka menee näytettäväksi näytölle. [22; 24.]

2.4 Näytönohjaimet

Näytönohjainten päätehtävä on käsitellä tietokoneen näytölle piirrettävää dataa. Näytönohjain voi olla joko osana emolevyä tai se voi olla siitä erillinen. Emolevyyn integroitu näytönohjain jakaa prosessointitehoa ja muistia suorittimen kanssa, mikä johtaa hitaampaan grafiikan suorittamiseen. Emolevystä erillisellä näytönohjaimella on oma muisti ja grafiikkasuoritin, jolloin säästyy tietokoneen suorittimen prosessointitehoa ja muistia muihin asioihin. Näytönohjaimet (kuva 4) koostuvat pääosin muistin ja suorittimen lisäksi jäähdytyselementistä, video-BIOS:sta ja ulostuloliitännöistä. Jäähdytyselementin idea on nimensä mukaan kiinnittää lämpöä ja estää näytönohjainta ylikuumentumasta. Video-BIOS taas hallitsee näytönohjaimen toimintoja ja asetuksia. Näytönohjaimessa voi olla monia erilaisia ulostuloliitäntöjä, esimerkiksi VGA-, HDMI- tai displayPort-liitäntä. Liitäntöjen tarkoitus on kuljettaa piirrettävä data niihin kiinnitettyjä johtoja pitkin näytölle. Joidenkin vanhempien liitäntöjen kanssa joudutaan käyttämään RAMDAC:a eli Random Access Memory Digital to Analog Converteria, joka muuttaa näytönohjaimesta ulos tulevan datan digitaalisesta analogiseksi. Tämä tarvitaan, koska monet vanhat näytöt eivät kykene käsittelemään digitaalista dataa. [25; 26.]



Kuva 4. Näytönohjain ja osat, joista se koostuu [25].

Näytönohjaimen muistiin tallennetaan kuvapuskurit, varjostinohjelmat, tekstuurit sekä muu kuvan piirtämiseen tarvittava data [26]. Kuvapuskuri pitää sisällään grafiikan liukuhihnan läpikäyneet fragmentit, jolloin niistä muodostuu valmis näytölle piirrettävä kuva. Useimmiten kuvan piirtämistä varten on varattu kaksi kuvapuskuria. Tämä mahdollistaa seuraavan kuvan laskemisen samalla, kun toista kuvaa näytetään näytöllä. Kun uusi kuva on valmis, vaihdetaan näytöllä näytettävä kuvapuskuri toiseen ja aloitetaan uuden kuvan laskeminen tälle puskurille. Tämä tekee kuvien vaihtumisesta sulavampaa, kuin mitä tulisi käyttämällä vain yhtä kuvapuskuria. [27.]

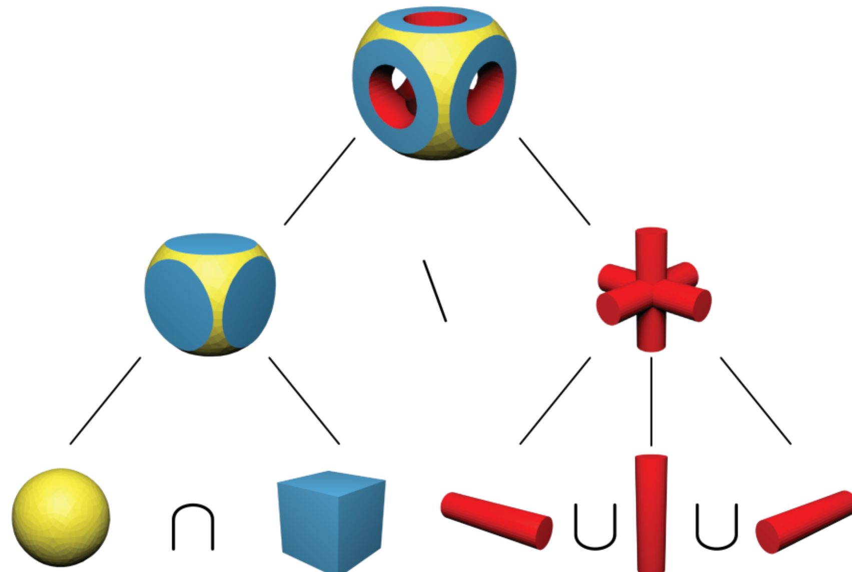
Näytönohjaimen grafiikkasuoritin eli GPU on erityisesti suunniteltu grafiikan laskemiseen, jolloin tämän toiminta eroaa CPU:n eli tietokoneen suorittimen toiminnasta. Toisin kuin CPU, joka käsittelee laajasti erilaisia tehtäviä, GPU on luotu käsittelemään suuria määriä samankaltaisia tehtäviä samanaikaisesti. Toisin sanoen GPU käsittelee tuhansia grafiikkaa laskevia säikeitä samanaikaisesti, mikä nopeuttaa kuvan laskemista huomattavasti. Tämä on mahdollista, sillä piirrettävät verteksit ja fragmentit ovat toisistaan riippumattomia. Koska GPU:t kykenevät käsittelemään tuhansia prosesseja samanaikaisesti, kiinnostaa tämä teho muutakin kuin grafiikan laskemista. [26.]

2.5 3D-mallit

3D-mallit ovat objekteja, joita voidaan tarkastella kolmiulotteisessa avaruudessa. Tämä tarkoittaa sitä, että objektilla on korkeus- ja leveysarvojen lisäksi myös syvyysarvo ja sitä pystyy tarkastelemaan eri kulmista. 3D-mallit jakaantuvat pääasiassa kahteen kategoriaan: umpinaisiin malleihin (solid models) ja pintamalleihin (surface models). [28.]

Umpinaiset 3D-mallit ovat nimensä mukaan umpinaisia ja pitävät tarkasti kirjaa objektin fyysistä ominaisuuksista. Tämä tekee umpinaisista malleista erittäin sopivia tekniseen työhön ja tietokoneavustettuun suunnitteluun (CAD). Yksi tapa muodostaa umpinaisia malleja on rakentava umpinainen geometria (CSG), jossa 3D-malli muodostuu yhdistämällä primitiivisiä muotoja Boolen-operaatioiden avulla. Primitiivit ja Boolen-operaatiot

muodostavat puumaisen rakenteen, jonka huipulle muodostuu valmis malli (kuva 5). [29.]

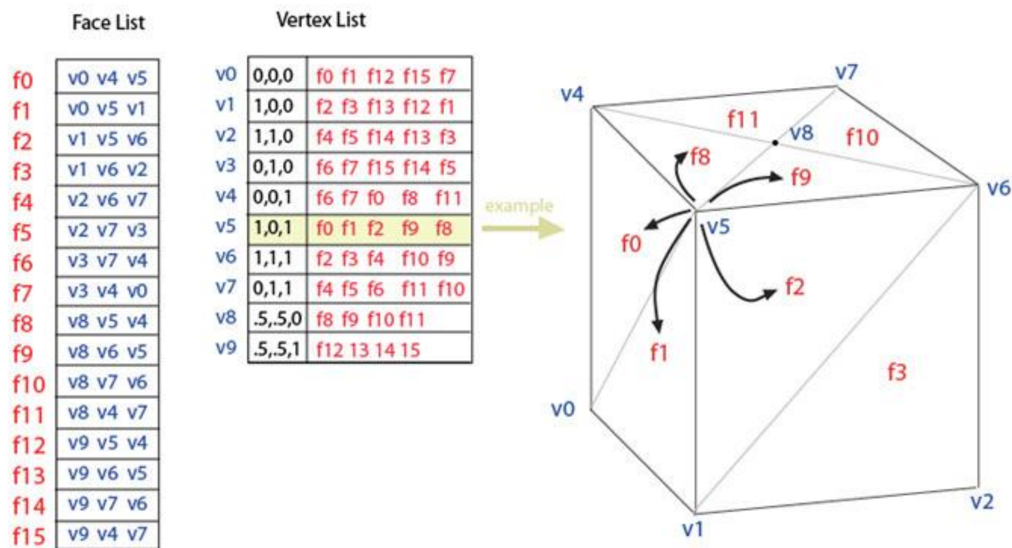


Kuva 5. Umpinainen malli, joka muodostuu yhdistelemällä objekteja käyttämällä Boolen-operaatioita [29].

Pintamallit eroavat umpinaisista malleista siten, että niissä 3D-mallista muodostetaan vain pinta, jolloin 3D-malli jää sisältä ontoksi. Mallin pinta koostuu pisteistä eli vertekseistä, jotka yhdessä muodostavat yleensä kolmioita tai neliöitä. Kolmiot sopivat mainiosti muodostamaan 3D-malleja, sillä kolmion kulmapisteet ovat aina samalla tasolla, toisin kuin esimerkiksi neliöillä. Tämä varmistaa sen, että pisteiden muodostama tahko on aina samanlainen. Koska pintamallien ei tarvitse muodostua tarkoista matemaattisista muodoista niin kuin umpinaisten mallien, voidaan muodostaa 3D-malli luonnollisemmista muodoista. Tämä johtaa myös siihen, että pintamallit ovat paljon helpommin muunneltavissa esimerkiksi animaatiokäyttöön. Nämä tekijät tekevät pintamalleista erittäin sopivia videopeli ja animaatio käyttöön.

Koska pintamallien elementit muodostuvat vertekseistä, pystytään 3D-malli piirtämään, jos tiedetään sen verteksit ja niiden yhteydet muihin vertekseihin. Tallentamalla 3D-mallista pelkästään verteksit ja niiden yhteydet saadaan 3D-malli tallennettua käyttämällä pieni määrä muistia, mutta tämä rakenne vaatii enemmän työtä, jotta se saadaan piirrettyä tietokoneen näytölle tai sitä saadaan muokattua. Helpompaa piirtämistä varten voidaan muodostaa tietorakenne, joka pitää kirjaa 3D-mallin vertekseistä ja niistä muodostuvista kolmioista (kuva 6). Kolmioiden tallentamisessa on hyvä ottaa huomioon se,

missä järjestyksessä verteksit muodostavat kolmion. Esimerkiksi 3D-mallin kolmiot voidaan muodostaa siten, että kolmioiden verteksit kulkevat vastapäivään. Tällöin kaikki kolmiot käyttäytyvät samalla tavalla, mikäli niistä lasketaan esimerkiksi kolmioiden normaalivektoreita, joiden avulla lasketaan esimerkiksi valojen vaikutusta 3D-malliin. [30; 31.]



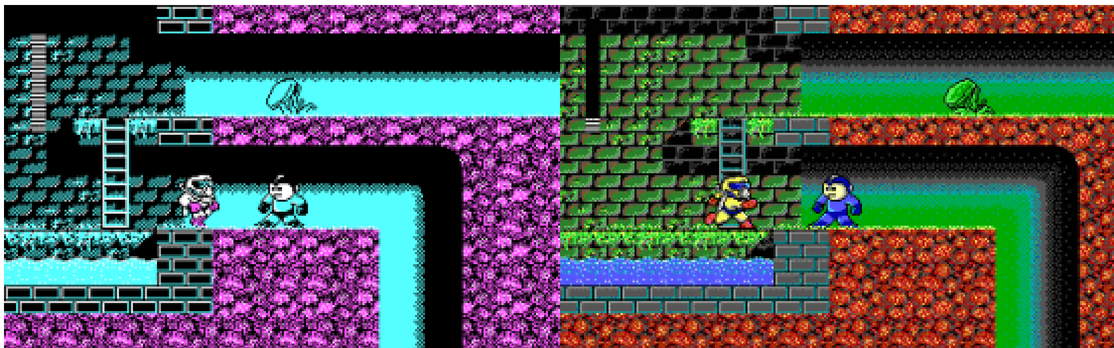
Kuva 6. Pintamalli, jossa objektin kolmioihin on tallennettu verteksit, joista ne muodostuvat ja vertekseille on tallennettu sijainnin lisäksi kaikki kolmiot, joihin se kuuluu [31].

2.6 Tekstuurit

Tekstuurit ovat kaksiulotteisia bittikarttakuvia, jotka koostuvat tekseleistä. Tekseli tulee sanoista tekstuuri ja pikseli. Niin kuin pikseli kuvastaa kuvaruudun pienimpiä pisteitä, tekseli kuvastaakin tekstuurin yksittäistä pistettä. Tekselit itsessään koostuvat yhdestä tai useammasta väriarvosta. Tekselit ovat useimmiten jaettu kolmeen väriarvoon: punaiseen, vihreään ja siniseen. Väriarvoille annetut arvot yhdessä muodostavat tekselille sen lopullisen värin. Tekselin väriin vaikuttaa myös tekstuurin värisyvyys eli, se kuinka monesta bitistä tekstuurin pikseli muodostuu. Nykyisin tekstuurien värisyvyys on yleensä 24 bittiä tai enemmän. 24 bitillä pystytään muodostamaan 2^{24} eri väriä eli yli 16 miljoonaa väriä. Tällä saadaan muodostettua kaikki värit, mitä ihmissilmä pystyy erottamaan. Tekseleissä on usein väriarvojen lisäksi neljäntenä arvona tallennettu alfa-arvo. Alfa-arvo

antaa tekstuurin pikseleille ylimääräistä dataa, jolla voidaan ohjailla pikselin arvoa varjostimessa. Useimmiten alfa-arvoa käytetään ohjailemaan tekstuurin pikseleiden läpinäkyvyyttä. [32.]

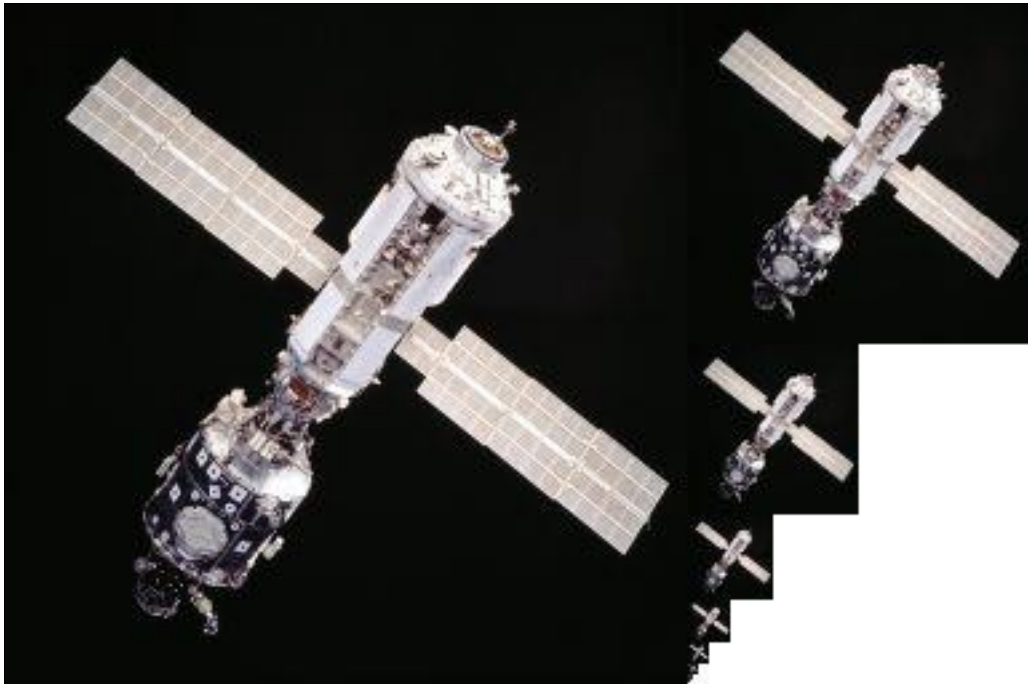
Vanhoissa tietokoneissa ei ollut käytössä yhtä suurta määrää muistia kuin nykyisissä, jolloin jouduttiin karsimaan värisyvyydestä tai resoluutiosta. Esimerkiksi 1980-luvun alussa, monissa tietokoneissa käytetyssä värigrafiikka-adapterissa (CGA) oli muistia vain 16 kilotavua. Tämä mahdollisti kaksiväristä kuvaa 640 x 200 -resoluutiolla tai sitten neliväristä kuvaa 320 x 200 -resoluutiolla. Kuvassa 7 vertaillaan CGA-standardilla piirrettyä kuvaa EGA-standardilla piirrettyyn kuvaan. EGA:lla 320 x 200 -resoluutioisessa kuvassa pystyttiin käyttämään 16 värin palettia, jolloin kuvaan saatiin myös piirrettyä enemmän yksityiskohtia. Koska CGA-standardilla 320 x 200 -resoluutioisessa kuvassa on vain 2 bittiä pikseliä kohti (bpp), siitä on vaikea muodostaa suoraan väriarvoa, minkä takia värit tallennettiin paletteihin, joista värin pystyi hakemaan pikselin arvolla. Paleteilla kyettiin säästämään muistia, sillä jokaista pikseliä varten tarvittiin vähemmän bittejä, mutta sen sijaan tämä johti siihen, että kuvassa voi olla vain paletissa olevia värejä. [33; 34.]



Kuva 7. CGA:n 4 värin moodi verrattuna EGA:n 16 värin moodiin Mega Man III -pelissä [33].

Kun tekstuuria viedään näytönohjaimelle, tulee kertoa, miten väriarvoja valitaan fragmenteille, kun fragmentti on joko suurempi tai pienempi kuin tekstuurin pikselit. Tämä tulee tehdä, koska fragmentti voi saada arvoja useammasta tekstuurinpikselistä, jolloin ei ole selvää, mikä väri tulisi valita fragmentille. Tekstuuri voidaan asettaa valitsemaan arvo joko läheisimmästä tekselistä, jolloin fragmentti saa lähimmän tekselin väriarvon, tai sitten lineaarisesti, jolloin väriarvo saadaan sekoittamalla läheisimpien tekseleiden väriarvoja keskenään. [35.]

Tekstuureiden käsittelyä varten voidaan tekstuurille luoda mipmap-tasot. Ne ovat pienennettyjä versioita tekstuurista. Tasot ovat pienennettyjä siten, että jokainen taso on aina puolet edellistä pienempi, kunnes viimeinen taso on vain yhden tekselin kokoinen (kuva 8). Tämän takia on tärkeää, että tekstuuri on kooltaan jokin luku korotettuna kahdella (1, 2, 4, 8, 16, jne.), jolloin tekstuuri puolittuu ongelmitta. Koska jokainen taso on puolet edellisestä, tekstuuri, jolla on mipmap-tasot, vie vain kolmanneksen enemmän tilaa kuin alkuperäinen tekstuuri. Mipmapeilla ratkaistaan ongelma, kun fragmentti on suurempi kuin tekstuurin pikselit. Sen sijaan, että joudutaan laskemaan fragmentin väriarvo käyttäen monia tekseleitä, voidaan vähentää tekseleiden määrää käyttämällä mipmap-tasoa, jonka koko paremmin kuvaa fragmentin kokoa suhteessa tekstuuriin. Esimerkiksi fragmentti, joka peittää koko tekstuurin, käyttäisi tekstuurin viimeistä mipmap-tasoa, joka muodostuu vain yhdestä tekselistä. Tällöin kuvasta tulee selkeämpi ja fragmenttien väriarvot kuvastavat paremmin tekstuurin antamia väriarvoja. [36.]



Kuva 8. Jokainen mipmap-taso on aina kooltaan puolet edellisestä tasosta [37].

Metropolia

3 Grafiikkasovelluksen toteutus

3.1 Tavoite ja toteutetut metodit

Insinöörityössä oli tavoitteena syventää ymmärrystä tietokonegrafiikan luomisesta ja päästä tutustumaan kaikkiin eri vaiheisiin, jotka piirrettävä kuva käy läpi ohjelmalta näytölle. Tätä varten luotiin ohjelma, joka pystyy piirtämään, käsittelemään ja luomaan 3D-objekteja ja tekstuureita. Ohjelman haluttiin olevan joustava ja helposti muokattava niin, että siihen on helppo lisätä uusia ominaisuuksia. Ohjelman oli myös tarkoitus olla hyvä pohja erilaisille työkaluille, jotka voisivat helpottaa ja tehostaa peligrafiikan luomista.

Työssä tehty sovellus suorittaa grafiikan piirtämisen eri vaiheet ja luo ikkunoita, joihin se piirtää teksturoituja 3D-objekteja. Ohjelma pystyy generoimaan yksinkertaisia 3D-objekteja, kuten tasoja ja kuutioita. Ohjelmassa keskityttiin paljon myös tekstuurin käsittelyyn ja generointiin. Tämä koettiin tärkeäksi, koska erilaisilla tekstuureilla on suuri vaikutus eri efektien luomisessa ja objektien ulkonäössä. Ohjelmassa toteutettiin erilaisten kohinatekstuurien luonti, kuten Perlinin ja Worleyn kohinan luonti. Ohjelmassa tutustuttiin myös fraktaaleihin sekä asioihin, joihin fraktaaleja voitaisiin mahdollisesti soveltaa. Sovellukseen toteutettiin myös muutamia tekstuurin käsittelyfunktioita, kuten kontrastin lisäys ja tekstuurien yhdistäminen, ja piirtämisfunktioita, kuten viivojen piirtäminen ja suoraan tekstuuriin piirtäminen.

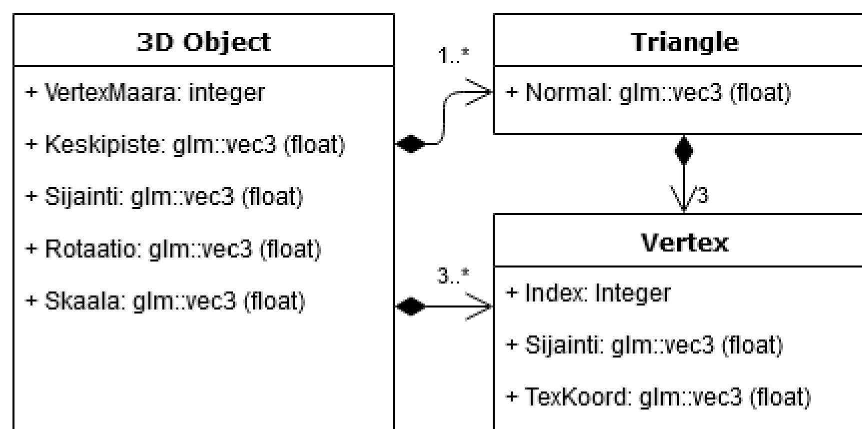
3.2 Ohjelmointirajapinnat ja kirjastot

Grafiikkatyökalun ohjelmointi toteutettiin käyttäen pääosin C++-ohjelmointikieltä. Grafiikan käsittelyyn käytettiin OpenGL-rajapintaa ja OpenGL shading languagea eli GLSL:ää varjostimien ohjelmointiin. OpenGL:n apuriksi tuodaan ohjelmaan myös OpenGL Extention Wrangler -kirjasto, joka auttaa käsittelemään OpenGL:ää. OpenGL:n luomien kuvien näyttämiseen kaivattiin ikkunaa, jolla kuvat pystyttiin näyttämään. Nämä ikkunat luotiin ohjelmassa OpenGL Frameworkillä eli GLFW:llä, joka ikkunoiden käsittelyn lisäksi seuraa käyttäjän näppäimistöllä tai hiirellä annettuja syötteitä. Ohjelmassa käytettiin myös OpenGL mathiä eli GLM:ää, joka on matematiikkakirjasto, joka imitoi GLSL:n matematiikan toimintaa. Tämä helpottaa grafiikkaan liittyvien laskujen laskemista C++-ohjelman

puolella. Ohjelmassa käytetään myös LibPNG-kirjastoa, joka auttaa kuvien käsittelyssä ja tallentamisessa png-muotoisiksi.

3.3 3D-mallien toteutus ja piirtäminen

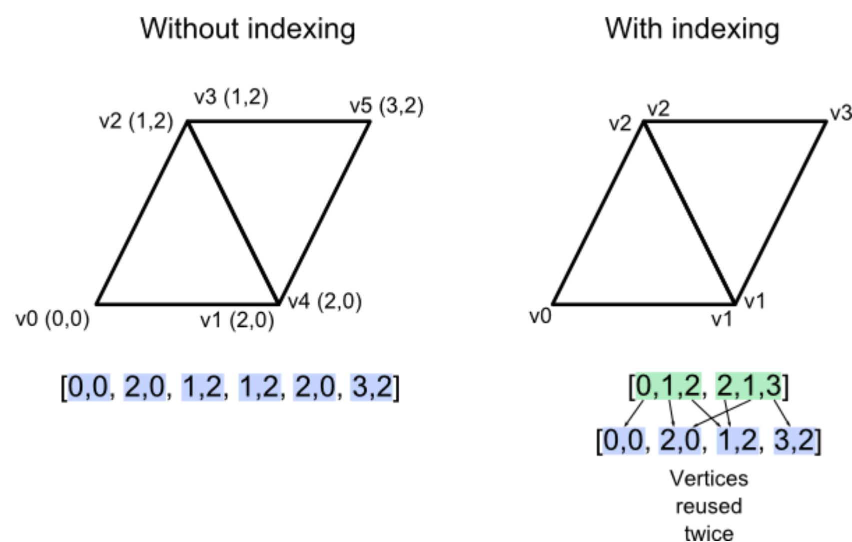
Koska OpenGL käsittelee pääasiassa 3D-grafiikkaa, ohjelmalle on luotu luokat 3D-objekteista, kolmiosta ja vertekseistä (kuva 9). Verteksit koostuvat paikkavektorista, tekstuurikoordinaatista ja indeksiarvosta. Kolmio-luokat taas muodostuvat kolmesta verteksistä, ja ne auttavat pitämään kirjaa siitä, mitkä verteksit muodostavat mitkäkin kolmiot. 3D-objektit taas pitävät sisällään listan vertekseistään ja kolmioistaan sekä muutosvektorit, jotka vaikuttavat objektin rotaatioon, kokoon ja paikkaan.



Kuva 9. 3D-objektit muodostuvat useista kolmio- ja verteksiolioista.

Objektin piirtämistä varten luodaan verteksin datasta puskureita. Puskurit ovat muistin alueita, joihin tallennetaan listoja dataa. Verteksiin liittyvää dataa on esimerkiksi sen sijainti, tekstuurin koordinaatti, normaali vektori, väri tai muu arvo, joka halutaan välittää verteksille. Koska puskureissa oleva data ei ole tarkkaan määriteltyä, luodaan vertex array -objekti pitämään kirjaa datasta. Puskurit kiinnitetään tähän objektiin, ja sille kerrotaan, montako arvoa jokainen verteksi käyttää ja minkämuotoista data on. Esimerkiksi verteksin sijainti voisi olla kolme liukulukuarvoa, joista jokainen arvo kuvastaa sijaintia XYZ-akseleilla.

Objekteja voidaan piirtää ruudulle käyttämällä joko glDrawArrays- tai glDrawElements-komentoa. Komennoilla voidaan piirtää OpenGL:n primitiivejä, jotka ovat pisteitä, viivoja tai kolmioita. Tämän ohjelman tapauksessa halutaan piirtää kolmioita. GlDrawElements- ja glDrawArrays-komennot eroavat toisistaan siinä, millaista dataa ne käsittelevät. GlDrawArray-komennon kanssa käytettävässä verteksidatassa on muodostettu jokaiselle primitiiville omat verteksit, joita ei jaeta muiden primitiivien kanssa. Koska primitiivit eivät voi jakaa verteksejä, joudutaan luomaan ylimääräisiä verteksejä paikkoihin, joissa primitiivit käyttäisivät samaa verteksiä. GlDrawElements-komento taas muodostaa kolmiot käyttämällä verteksien indeksiarvoja, jolloin ei tarvitse luoda samaa verteksiä uudestaan, vaan voidaan indeksiarvolla osoittaa uudestaan samaan verteksiin (kuva 10). Verteksien indeksoiminen säästää siten myös hieman muistia.



Kuva 10. Kulmapisteet indeksoimalla voidaan osoittaa samaan verteksiin useita kertoja [38].

3.4 MVP-matriisit

Oletuksena OpenGL piirtää kuvan pisteestä (0, 0, 0) katsoen, ja näkymän kulmat sijaitsevat koordinaatiston XY-akselilla pisteissä (-1, -1), (-1, 1), (1, -1), (1, 1). Tämä ei yleensä tuota haluttua kuvaa, sillä tämä johtaa siihen, että objektit venyvät ikkunan koon mukaan, mikä vaatii sen, että 3D-objektit on luotu ottaen huomioon näkymän koordinaatisto. Tämä ei tosin ole erityisen tehokasta, joten sen sijaan voidaan kertoa objektien verteksien paikkavektorit MVP-matriiseilla. Tämä muuttaa verteksien koordinaatistoa, mikä muuttaa sitä, miten objektit näkymään piirretään. MVP-matriisit koostuvat kolmesta 4 x 4 -kokoisesta matriisista: malli- (model), näkymä- (view) ja projektiomatriisista (projection).

Mallimatriisi käsittelee objektin rotaatiota, kokoa ja sijaintia. Se saadaan muodostettua kertomalla keskenään rotaatio-, skaala- ja translaatiomatriisit (kuva 11). Matriisia muodostaessa on hyvä ottaa huomioon, että objektin skaalaaminen ja kääntäminen tehdään suhteessa origoon. Mikäli objekti halutaan kääntää tai skaalata jonkin muun pisteen kuin origon suhteen, tulee mallimatriisia luodessa kertoa translaatiomatriisilla, joka korjaa origon suhteen muokkaamisen aiheuttaman väärän sijainnin. Ohjelmassa objektia muokkaavat arvot tallennetaan 3D-objektiluokkaan vektoreihin, jotka antavat arvot mallimatriisin luomista varten.

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \begin{pmatrix} 1 & 0 & 0 & tx \\ 0 & 1 & 0 & ty \\ 0 & 0 & 1 & tz \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \begin{pmatrix} sx & 0 & 0 & 0 \\ 0 & sy & 0 & 0 \\ 0 & 0 & sz & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Identity Matrix glTranslatef(tx,ty,tz) glScalef(sx,sy,sz)

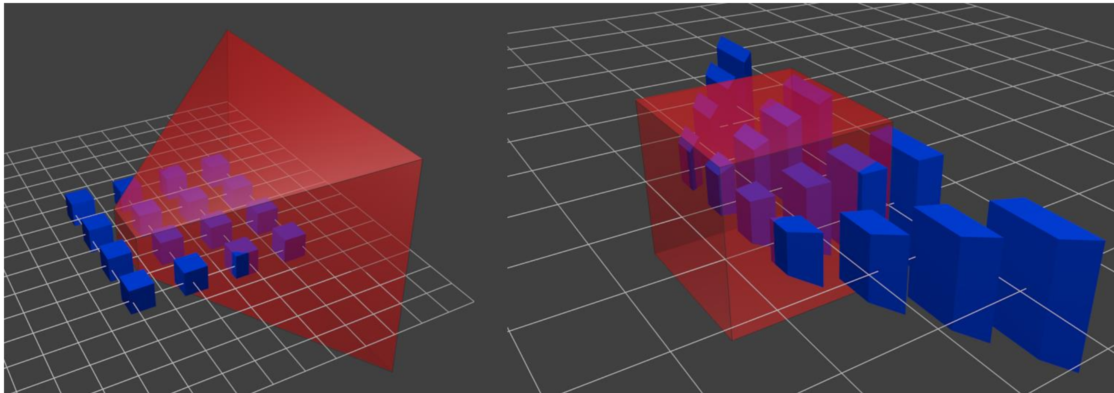
$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(d) & -\sin(d) & 0 \\ 0 & \sin(d) & \cos(d) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \begin{pmatrix} \cos(d) & 0 & \sin(d) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(d) & 0 & \cos(d) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \begin{pmatrix} \cos(d) & -\sin(d) & 0 & 0 \\ \sin(d) & \cos(d) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

glRotatef(d,1,0,0) glRotatef(d,0,1,0) glRotatef(d,0,0,1)

Kuva 11. Mallimatriisi muodostuu kertomalla keskenään objektin translaatio-, skaala- ja rotaatiomatriisit [39].

Näkymämatriisin tehtävä on määrittää 'kameran' sijainti ja suunta. Ohjelmassa tämä matriisi on luotu käyttämällä glm-kirjaston lookAt-funktiota, jolle annetaan parametrina kameran sijainti, piste, jota katsotaan, ja vektori, joka osoittaa, mikä suunta on ylöspäin. Projektiomatriisin tehtävä on ottaa huomioon näkymän mittasuhteet ja se, miten käsitellään objektien etäisyyttä kamerasta. Tämä varmistaa, että ikkunan muoto ei vaikuta myös objektien muotoon. Tällä luodaan myös perspektiivi näkymään (kuva 12), jolloin objektit, jotka ovat kaukana kamerasta, myös näyttävät pienemmiltä kuin objektit lähempänä kameraa. Ilman tätä objektit ovat aina samankokoisia riippumatta siitä, kuinka kaukana objektit ovat. Ohjelmassa projektiomatriisi saadaan muodostettua käyttämällä glm-kirjaston perspective-funktioita. Tälle annetaan parametreina kulma, joka kuvastaa sitä, missä suhteessa objektin koko muuttuu etäisyyden perusteella; näkymän kuvasuhde ja

näkymän minimi- ja maksimietäisyydet, minkä perusteella jätetään pois objektit, jotka ovat joko liian lähellä tai kaukana kamerasta. [40.]



Kuva 12. Verteksien kertominen projektiomatriisilla luo näkymään perspektiivin vaikutelman [41].

3.5 Varjostimet

Ohjelmassa todellinen piirtäminen tapahtuu näytönohjaimen puolella, jossa piirrettävän datan ohjaamista varten tarvitaan varjostinohjelmat. Varjostinohjelmien käyttämistä varten tarvitsee luoda ohjelma, johon varjostinohjelmat kiinnitetään. Tähän ohjelmaan voidaan myös liittää piirrettävälle datalle yhteisiä parametreja varjostimien käytettäväksi. Tällaisia parametreja voisivat olla esimerkiksi MVP-matriisit tai valojen tiedot. Ohjelma tarvitsee vähintään verteksi- ja fragmenttivarjostimen objektien piirtämistä varten. Verteksien varjostin saa käyttöönsä verteksipuskureihin tallennetut tiedot ja varjostinohjelmiin liitetyt parametrit. Verteksipuskureihin tallennettu data on jokaiselle verteksille oma, ja liitetyt parametrit ovat kaikille vertekseille yhteiset. Verteksivarjostimessa voidaan esimerkiksi laskea vertekseille uudet koordinaatit MVP-matriiseiden avulla, laskea valojen arvoja tai käsitellä muuta dataa muita varjostimia varten.

Verteksikäsittelyn jälkeen objektin kolmiot fragmentoidaan, jolloin syntyy pisteitä, joita voidaan käsitellä fragmenttivarjostimella. Fragmentit saavat itselleen verteksivarjostimen välittämät arvot ja OpenGL:lle kiinnitettyjen tekstuurien arvot. Verteksien välittämät arvot ovat interpoloituja sen mukaan, kuinka lähellä fragmentti on kolmion verteksejä. Tämä mahdollistaa sen, että fragmentit saavat omat arvot, ilman että jokaiselle fragmentille tarvitsee erikseen asettaa niitä. Fragmenttivarjostimessa lasketaan pisteelle väriarvo.

Väriarvo voidaan joko asettaa suoraan joksikin väriksi tai napata se tekstuurista. Tekstuurista väriarvo saadaan käyttämällä tekstuurikoordinaattiarvoja. Tämä toimii siten, että vertekseille on asetettu kaksiulotteinen koordinaatti, jonka arvot ovat nollan ja ykkösen välillä. Tällä koordinaatilla saadaan sitten haettua väriarvo tekstuurista. Tätä tapaa asettaa objektille tekstuuri kutsutaan UV-kartoitukseksi. Fragmenteille voidaan varjostimessa myös laskea valon vaikutusta ja muita efektejä. Fragmenttikäsittelyn jälkeen pisteille on laskettu väriarvot ja ohjelman ikkunaan saadaan piirrettyä kuva.

3.6 Tekstuurin käsittely ja generointi

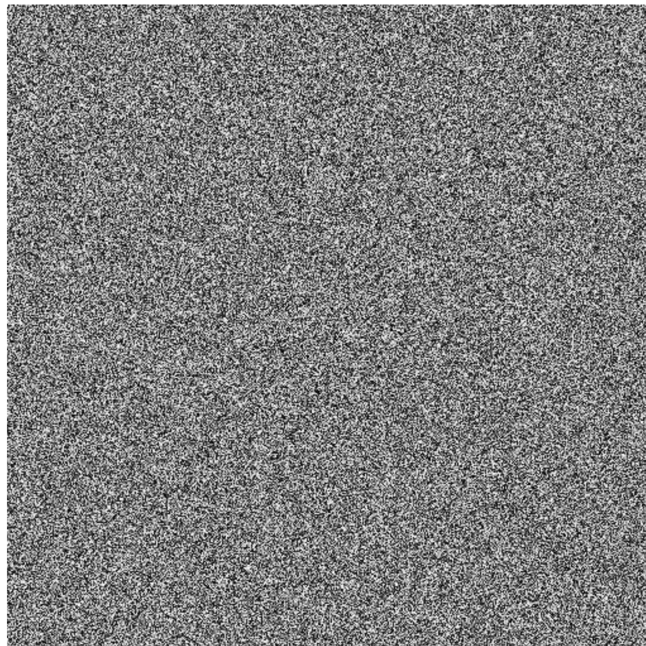
Ohjelmassa tekstuurin käsittelyä varten luodaan tekstuuriluokka, joka sisältää tekstuurin pituuden ja leveyden tekseleinä, ja tekstuurin väriarvot muodostava liukulukulista. Väriarvot ovat RGBA-formaatissa tallennettuja, eli yksi väriarvo koostuu neljästä liukuluvusta. Tekstuureiden käyttöä varten tulee OpenGL:lle luoda tekstuurilista. Siihen luodaan käytettävät tekstuurit kertomalla sille tekstuurin formaatti, datatyyppi, koko ja tekstuurin väriarvolista. Tekstuureille voidaan myös kertoa, miten OpenGL käsittelee objektien teksturointia esimerkiksi tilanteissa, joissa tekstuuriin osoittavat koordinaatit ovat tekstuurin ulkopuolella, tai miten väriarvo tulisi valita pisteisiin, joissa tekstuuria joudutaan suurentamaan tai pienentämään.

Koska ohjelman haluttiin toimivan työkaluna erilaisen tekstuurin generoimiseen pelikäyttöön, luotiin monia eri funktioita käsittelemään ja luomaan tekstuuria. Tekstuurin käsittely on kohtuullisen yksinkertaista. Koska tekstuuri koostuu liukulukulistasta, voidaan tekstuurin pikselin väriarvoa muokata muokkaamalla siihen viittaavien liukulukujen arvoja listassa. Listassa tekstuurin pistettä vastaa neljä liukulukuarvoa, jotka ovat RGBA-formaatissa. Tämä tarkoittaa sitä, että näistä neljästä arvosta ensimmäinen kuvastaa tekselin punaisuutta, toinen vihreyttä, kolmas sinisyyttä ja neljäs arvo on alfa-arvo, jota usein käytetään kuvastamaan läpinäkyvyyttä. Koska listan arvot ovat liukulukuja, niiden arvot ovat nollan ja ykkösen välillä, jossa nolla asettaa väriarvon painotuksen minimiin ja ykkönen maksimiin. Värit sekoittuvat yhteen ja muodostavat yhdessä halutun värin. Esimerkiksi väriarvo, joka muodostuu arvoista (1, 0, 1, 1), ei saa painotusta vihreästä, mutta saa täyden painotuksen sinisestä ja punaisesta, jolloin väriarvoksi muodostuu liila. Yksinkertaisimmillaan tekstuurin generointifunktio on funktio, joka käy tekstuurin väriarvolistan läpi neljä liukulukua kerrallaan ja asettaa niille uuden väriarvon. Ohjelmaan luotiin

funktio, joka sitten osaa generoituja tekstuureja tallentaa png-muotoisiksi kuviksi käyttäen apunaan libPNG-kirjaston funktioita.

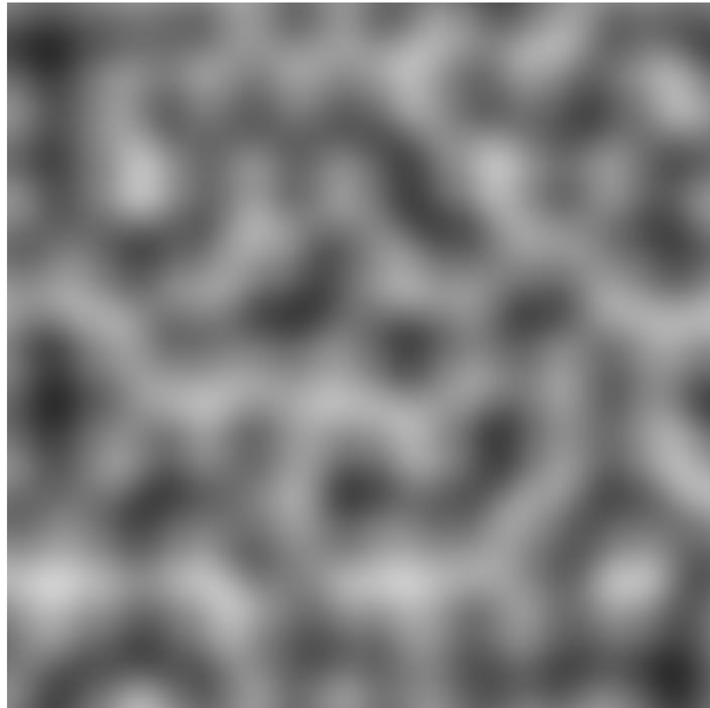
3.6.1 Kohina

Kohinalla tarkoitetaan satunnaissignaalia. Tietokonegrafiikassa kohinaa voidaan käyttää esimerkiksi proseduraaliseen generointiin tai auttamaan luomaan erilaisia graafisia efektejä. Yleisimmillään visuaalinen kohina on kuva, jossa jokaiselle tekstuurin pikselille on annettu satunnainen arvo mustan ja valkoisen väliltä. Tällainen kohina tunnetaan myös valkoisena kohinana (engl. white noise) (kuva 13). Usein kuitenkaan ei haluta käyttää täysin satunnaista kohinaa, vaan halutaan tietynlaista satunnaista kuvaa avustamaan eri efektien luomisessa. Eräänlaista ohjailtua kohinaa ovat esimerkiksi Perlinin ja Worleyn kohinat.



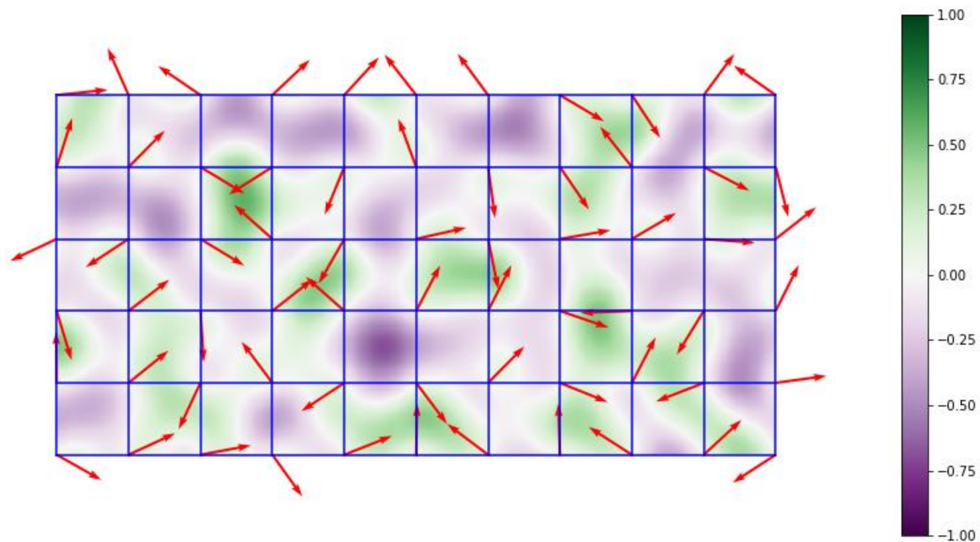
Kuva 13. Kohinatekstuuri, jossa jokainen väriarvo on valittu satunnaisesti 0:n ja 1:n väliltä.

Perlinin kohina ei ole niin satunnaista kuin valkoinen kohina, sillä tekstuurin pikselien väriarvossa otetaan huomioon pikselin ympäristö. Perlinin kohinassa väriarvo vaihtuu toiseen vähitellen, muodostaen tekstuuriin erimuotoisia alueita (kuva 14). Perlinin kohinaa käytetään usein esimerkiksi generoimaan luonnollisen näköistä maastoa tai muita pinnanmuotoja.



Kuva 14. Perlinin kohinassa väriarvo vaihtuu pehmeästi väristä toiseen.

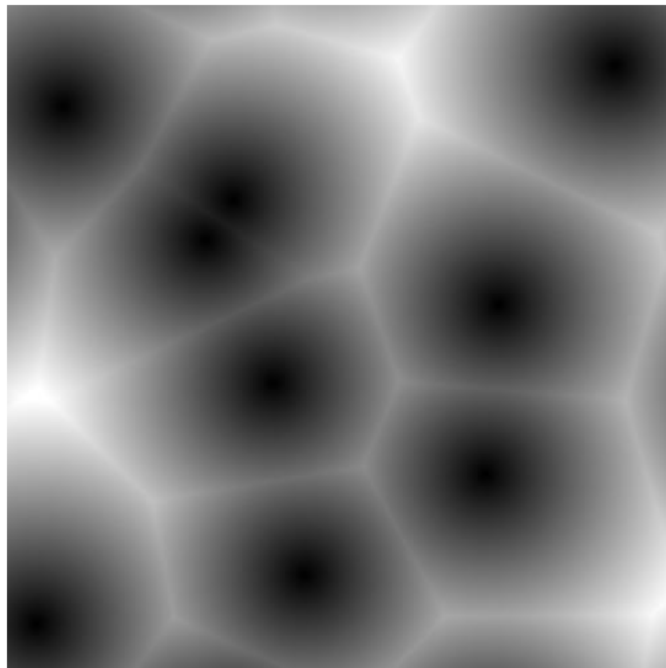
Perlinin kohinan generoiminen perustuu siihen, että tekstuuri jaetaan ruudukkoon ja jokaiseen ruudun kulmaan luodaan gradienttivektori. Tällöin jokaiselle ruudulle muodostuu neljä gradienttivektoria, joiden avulla lasketaan ruudun sisällä olevien pikselien arvoja. Ruudut, jotka jakavat kulmia, jakavat myös kulmassa olevan gradienttivektorin, jolloin tekstuuri jatkuu sulavasti myös ruutujen välillä. Muodostetun ruudukon ruutujen koot määrittävät, minkäkokoista kuviota kohinaan luodaan. Gradienttivektori on yksikkövektori, joka osoittaa satunnaiseen suuntaan. Sitä käytetään antamaan pikselille väriarvo (kuva 15). Tämä väriarvo saadaan, kun muodostetaan etäisyysvektori, jonka alkupiste on samassa pisteessä kuin gradienttivektorin alkupiste, ja loppupiste on pikselin pisteessä, ja lasketaan sen ja gradienttivektorin pistetulo. Pistetulo kertoo tämän gradienttivektorin antaman väriarvon pikselille. Laskettu väriarvo on -1:n ja 1:n välinen arvo, joka kertoo, kuinka kaukana pikseli on gradienttivektorista ja kuinka samansuuntainen gradienttivektori on etäisyysvektorin kanssa.



Kuva 15. Tekstuurin pikselit, joihin gradienttivektorit osoittavat, saavat kuvassa vihreän värin, kun taas pikselit, jotka ovat gradienttivektoreihin nähden vastakkaisessa suunnassa saavat liilan värin [42].

Pikselille lasketaan tämä pistetulo käyttäen kaikkia oman ruudun neljää gradienttivektoria, jolloin pikseli saa neljä eri väriarvoa. Tämän jälkeen interpoloidaan saatujen väriarvojen väliltä, jolloin saadaan laskettua painotettu väriarvo. Väriarvo asetetaan pikselille, ja tämä väriarvon laskeminen toistetaan jokaiselle tekstuurin pikselille, mikä sitten muodostaa Perlinin kohinatekstuurin. [43.]

Worleyn kohina (kuva 16) on solumaista kohinaa, eli tekstuurille muodostuu solumaisesti jakautunut rakenne. Worleyn kohina perustuu Perlinin kohinan tapaan myös siihen, että tekstuuri on jaettu ruutuihin. Mutta toisin kuin Perlinin kohinassa, Worleyn kohinassa ei luoda gradienttivektoreita, vaan jokaiseen ruutuun luodaan satunnaiseen paikkaan piste. Näitä pisteitä käytetään määrittelemään pikseleiden väriarvoja. Tämä tapahtuu siten, että pikselille selvitetään, mikä on sen etäisyys lähimmälle ruuduille asetetulle pisteelle. Tämä piste voi olla joko samassa ruudussa, kuin missä pikseli sijaitsee, tai se voi sijaita jossain ruutua ympäröivistä kahdeksasta ruudusta. Sitten, kun lähimmän pisteen etäisyys on selvitetty, käytetään tätä etäisyyttä laskemaan tekstuurin pikselin väriarvo. Koska pikselin väriarvo määräytyy lähimmän pisteen arvosta, muodostuu pisteiden välille selkeä raja kohtiin, joissa pisteet ovat yhtä lähellä pikseliä. [44.]

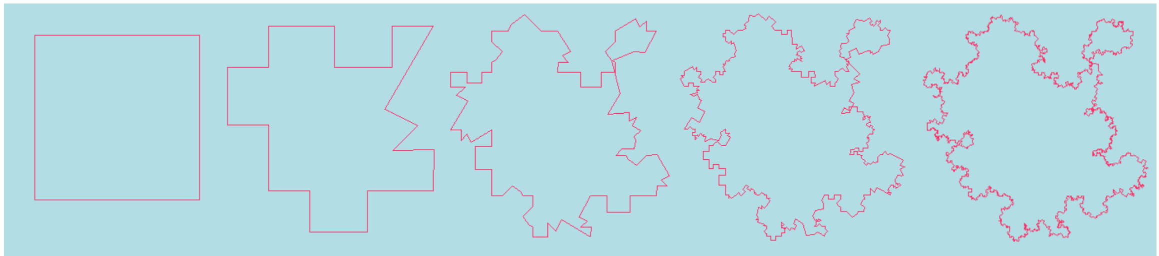


Kuva 16. Worleyn kohinan tekstuurin pikselit saavat väriarvonsa ruudukossa olevien pisteiden mukaan.

3.6.2 Fraktaalit

Fraktaaleina yleisesti tunnetaan muodot ja kuviot, jotka toistavat itseään äärettömiin. Tämä ei kuitenkaan määrittele fraktaaleja, ja sanotaan, että fraktaalien idea perustuu enemmän siihen, että kuvio säilyttää karkeutensa, vaikka kuviota tarkasteltaisiin lähempää. [45.] Vaikka fraktaalien määritelmä on hieman epäselvä, voidaan kuitenkin hyödyntää fraktaaleiden ideaa itsensä toistosta ja karkeuden säilyttämisestä. Ohjelmaan luotiin

funktio, joka piirtää tekstuuriin kuvia, jotka muistuttavat karkeita maiden rajoja (kuva 17). Tämä toteutettiin siten, että luotiin luokka fraktaaliviivalle, joka voi koostua useammasta fraktaaliviivaluokasta. Fraktaaliviiva koostuu siis alku- ja loppupisteestä sekä listasta fraktaaliviivoja. Luokkaan luotiin myös funktio, joka korvaa suoran viivan, kuviolla, joka koostuu useammasta suorasta viivasta. Kuviosta, joka korvaa alkuperäisen viivan, luotiin ohjelmaan muutama eri vaihtoehto tuomaan vaihtelua kuvaan. Näistä kuvion viivoista muodostetaan uusia fraktaaliviivoja alkuperäisen fraktaaliviivan fraktaaliviivalistaan. Jos funktiota kutsutaan uudestaan fraktaaliviivalle, fraktaaliviiva ei luo enää itselleen uusia viivoja, vaan kutsuu tätä samaa funktiota listassa olevilla fraktaaliviivoilla, jolloin ne luovat omiin listoihinsa uusia viivoja. Sitten kun fraktaalikuva halutaan piirtää, kutsutaan ohjelmaan luotua viivanpiirtofunktiota fraktaaliviivoille, joiden fraktaaliviivalista on tyhjä. Näiden löytämiseksi käydään läpi alkuperäistä fraktaaliviivaoliota ja katsotaan, onko sen fraktaaliviivalista tyhjä. Jos ei ole, toistetaan tämä kaikille listan viivoille, kunnes kaikki viivat on käyty läpi, jolloin viimeiset viivat on piirretty tekstuuriin muodostamaan fraktaalikuva.



Kuva 17. Fraktaalikuva, joka lisää itseensä yksityiskohtia kerroksittain, muodostaen mielenkiintoisen uuden kuvan.

Fraktaalikuvia voitaisiin mahdollisesti hyödyntää esimerkiksi imaginääristen karttojen luomisessa, ja samaa konseptia voisi hyödyntää esimerkiksi puiden tai maastojen generoinnissa. Maastojen kanssa konsepti voisi toimia siten, että luodaan kolmiosta suurpiirteinen maaston muoto ja jaetaan kolmiota useammiksi kolmioiksi muuttaen samalla hiukan alkuperäisen kolmion muotoa. Tätä tarpeeksi toistettua saataisiin maastoon luotua luonnollista karkeutta.

3.6.3 3D-tekstuuri

Projektissa tutustuttiin myös 3D-tekstuurin ideaan. Tavallisesta tekstuurista se ei eroa muuten kuin siten, että sillä on kolmas ulottuvuus eli syvyysarvo. Tämä käytännössä luo

siis monikerroksisen tekstuurin. OpenGL osaa käsitellä 3D-tekstuureita, ja 3D-objektit voidaan UV-kartoittaa käyttämään myös tekstuurin syvyysarvoa. Näin voidaan esimerkiksi animoida objektin tekstuuria vaihtamalla käytettävää tekstuurin syvyysarvoa kuvien välissä. Tämä ei kuitenkaan ole paras tapa animoida tekstuureja ja vie paljon muistia, riippuen siitä, kuinka monta kerrosta tekstuurissa on. 3D-tekstuurilla on myös hankalaa tehdä efektejä, joissa voitaisiin ottaa huomioon objektin sisällä oleva tekstuuri. Tämä johtuu siitä, että 3D-objekteista piirretään vain objektin pinnat, jolloin objektin sisällä ei ole mitään piirrettävää, jolloin eivät myöskään 3D-tekstuurit näy. Koska tällainen efekti vaatii paljon työtä toimiakseen ja vie paljon muistia ja laskentatehoa, ei 3D-tekstuureja käytetä paljon varjostimien kanssa.

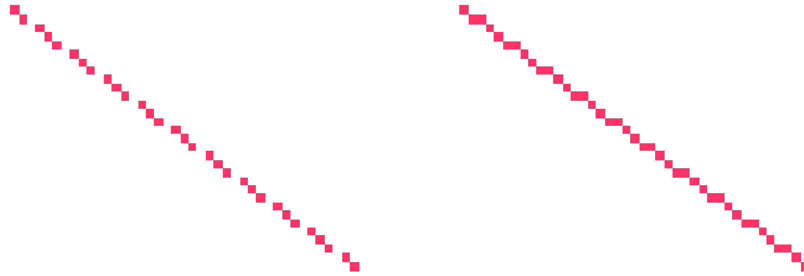
Vaikka 3D-tekstuurit eivät vaikuta erityisen hyödyllisiltä piirtämisessä, kuitenkin 3D-tekstuurin tyyppinen data voi olla erittäin hyödyllistä. Esimerkiksi jonkinlaisesta muunnellusta Perlinin kohinasta voidaan tehdä 3D-tekstuurin tyyppistä dataa, jota voitaisiin käyttää kolmiulotteisten luolamaisten rakenteiden luomiseen.

3.6.4 Tekstuuriin piirtäminen

Sovellukseen haluttiin luoda myös suoraa tekstuurin pikseleiden käsittelyä piirtämisen muodossa, joten sovellukseen luotiin piirtämis- ja viivafunktiot. Piirtämisfunktioita varten luotiin funktio, joka pystyy muuttamaan ikkunan koordinaatin tekstuurin koordinaatiksi. Tällä tekstuurin koordinaatilla pystyttiin etsimään tekstuurin väriarvolistasta arvo, jota halutaan käsitellä. Piirtämisfunktio yksinkertaisesti vain vaihtaa tällä funktiolla löydetyn arvon värin halutuksi väriksi.

Viivanpiirtofunktio piirtää viivan annetusta alkupisteestä annettuun loppupisteeseen. Funktio ottaa siis parametreikseen näiden pisteiden koordinaatit. Näiden koordinaattien avulla funktio muodostaa suoralle yhtälön käyttäen suoran yhtälöä $y = ax + b$. Yhtälössä a kuvastaa suoran kulmakerrointa ja b kuvastaa suoran ja y-akselin leikkauskohtaa. Kulmakerroin ja y-akselin leikkauskohta lasketaan siis käyttäen alku- ja loppupisteen arvoja. Suora käydään läpi pikseli pikseliltä toisen akselin suhteen ja lasketaan suoran piste käyttäen suorayhtälöä. Koska suorasta ollaan piirtämässä vain osa, aloitetaan suoran piirtäminen alkupisteestä ja lopetetaan sen käsittely loppupisteeseen. Viivan piirtämisessä tulee ottaa huomioon, kumman akselin mukaan pikseleitä käydään läpi. Koska suoraa piirretään vain yksi tekseli jokaisessa askeleessa, väärän akselin kautta käsittely

saattaa jättää pisteitä piirtämättä, jolloin viivasta tulee katkonainen (kuva 18). Akseli saadaan valittua kulmakertoimen mukaan oikeaksi. Jos kulmakerroin on välillä -1 ja 1, tiedetään, että suora on jyrkästi nouseva tai laskeva, jolloin käsitellään suoraa y-akselin mukaisesti. Muulloin käsitellään suoraa x-akselin mukaisesti. Tässä tulee myös ottaa huomioon erikoistapaukset, joissa suora on jommankumman akselin suuntainen.



Kuva 18. Väärän akselin mukaan käsittely luo katkonaisen viivan.

Ohjelmaan luotiin myös alueen täyttöfunktio, joka pystyi muuttamaan rajatun alueen toisenväriseksi. Funktio toimi siten, että se muutti valitun tekselin halutunväriseksi ja vertasi, ovatko sen viereiset tekselit samanvärisiä kuin käsiteltävä tekseli oli ennen värin vaihtamista. Jos tekseli oli samanvärinen kuin käsiteltävä tekseli oli, kutsuttiin samaa funktiota sillä tekselillä. Tämä toistui, kunnes löytyi tekseleitä, jotka eivät olleet alkuperäisen tekselin värisiä ja alueen tekselit oli käyty läpi. Tämä toimi hyvin muuttamaan pieniä alueita toisenvärisiksi, mutta koska funktio kutsui itse itseään toistuvasti, tämä johti hyvin nopeasti kutsupinon täyttymiseen ja siten ohjelman kaatumiseen suuremmilla alueilla. Koska funktiota ei koettu erityisen tärkeäksi, asia jätettiin tähän eikä funktiolle etsitty korjauksia.

Piirtofunktiot olivat hyvä tapa tutustua tekstuurin toimintaan ja selvittää, miten eri piirto-ohjelmien funktiot mahdollisesti toimivat. Kuitenkin tekstuurinkoordinaatin hakeminen käyttämällä suoraan ikkunan koordinaatteja ei kuitenkaan toimi halutusti, elleivät tekstuurin minimi- ja maksimikoordinaatit ole samassa kohtaa kuin ikkunan minimi- ja maksimikoordinaatit. Toisin sanoen 3D-objektille klikkaaminen ei palauta 3D-objektin tekstuurin koordinaattia vaan tekstuurin koordinaatin suhteessa ikkunan koordinaattiin. 3D-objektiin piirtäminen vaatisi sitä, että tiedetään, mitä kolmiota on klikattu, ja sen verteksien sijainnilla ja tekstuurinkoordinaateilla selvitettäisiin, mitä tekstuurin pikseliä halutaan käsitellä.

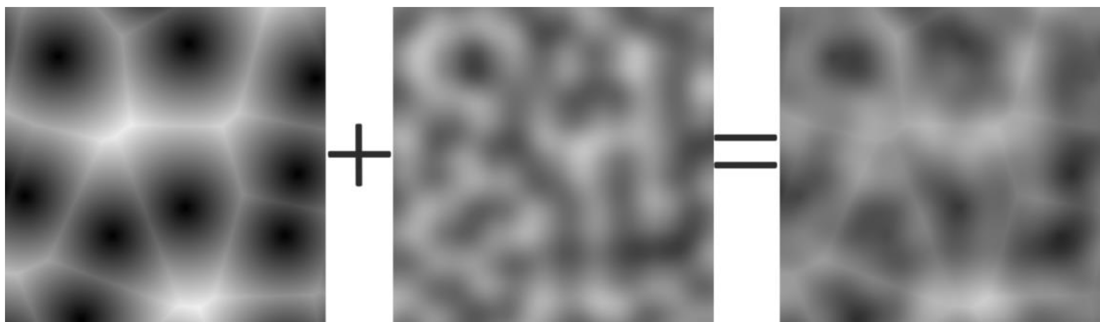
3.6.5 Tekstuurin käsittely

Sovellukseen haluttiin luoda myös funktioita, jotka vaikuttavat kaikkiin tekstuurin väriarvoihin. Esimerkiksi haluttiin muuttaa kuvan kontrastia tai yhdistellä tekstuureita. Nämä operaatiot voitaisiin hoitaa fragmenttivarjostimen puolella, jos haluttaisiin vain näyttää kuvaa. Mutta koska halutaan mahdollisesti käyttää luotuja tekstuureita muuhun grafiikan käsittelyyn, halutaan tallentaa tekstuurit muutettuina.

Sovellukseen luotiin tekstuurien yhdistämisfunktio, joka ottaa kaksi tekstuuria ja yhdistää painotetusti niiden väriarvot yhteen luoden uuden tekstuurin. Funktio käy tekstuureiden jokaisen väriarvolistan arvon läpi ja laskee niille uuden arvon käyttäen yhtälöä 1.

$$U = V1 + S * (V2 - V1) \tag{1}$$

Yhtälössä U kuvastaa uutta arvoa ja V1 ja V2 kuvastavat yhdistettävien tekstuureiden vanhoja arvoja. Yhtälössä S kuvastaa sitä, miten arvoja painotetaan, ja sen arvon tulee olla välillä 0 ja 1. Mitä suurempi S on, sitä enemmän se painottaa V2-arvoa. Esimerkiksi, jos S on 0,5, painotetaan molempia vanhoja arvoja saman verran. Kun kaikille arvoille on laskettu uusi arvo, syntyy uusi tekstuuri, joka on yhdistelmä funktiossa käytettyjä tekstuureita (kuva 19). Funktion käytössä tulee ottaa huomioon, että se käsittelee vain samankokoisia tekstuureita, jolloin tekstuurit tulee muuttaa samankokoisiksi, ennen kuin funktiota käytetään.



Kuva 19. Funktio painotetusti yhdistää tekstuurit ja muodostaa niistä uuden tekstuurin.

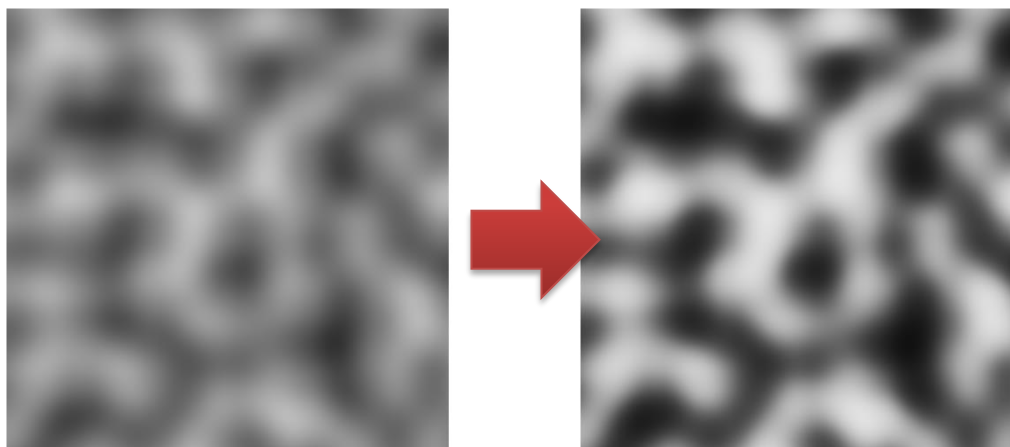
Ohjelmaan luotiin myös kontrastifunktio, joka muuttaa tekstuurin väriarvojen kontrastia. Tämä tapahtuu koodiesimerkin 1 mukaisesti.

```
void Texture::AddContrast(float str) {
	if (str == 0) {
		return;
	}
	float min = atanf(2 * 3.141 * str * (-0.5));
	float max = atanf(2 * 3.141 * str * (0.5));
	float value;

	for (int i = 0; i < texelCount * 4; ++i)
	{
		value = atanf(2 * 3.141 * str * (pic[i] - 0.5));
		pic[i] = ChangeScale(min, max, 0, 1, value);
	}
}
```

Esimerkkikoodi 1. Kontrastinlisäysfunktio.

Funktio saa parametreinaan kontrastin voimakkuuden, joka kertoo, kuinka paljon kontrasti vaikuttaa tekstuurin väriarvoihin. Funktiossa ensin lasketaan minimi- ja maksimiarvot, jotka kertovat, millä välillä lasketut arvot ovat. Väriarvojen uudet arvot saadaan laskemalla arkustangentit käyttäen väriarvojen arvoja. Koska lasketut arvot eivät ole tässä kohtaa 0:n ja 1:n välillä, käytetään ChangeScale-funktiota, joka suhteuttaa uuden arvon 0:n ja 1:n välille käyttäen aiemmin laskettuja minimi- ja maksimiarvoja. Näin saadaan kuvaan



Kuva 20. Kontrastin lisääminen tekstuuriin korostaa tekstuurin väriarvoja.

3.7 Projektin pohdinta

Sovelluksen toteutus antoi paljon käytännön kokemusta siitä, mitä kaikkea tietokonegrafiikan piirtäminen vaatii. Sovelluksessa saatiin aikaiseksi sujuva tietokonegrafiikan piirtäminen ja monta toimivaa grafiikkaa luovaa ja muokkaavaa metodia. Suurin osa näistä

metodeista toimii sujuvasti eikä vie turhauttavaa määrää aikaa tai laskentatehoja. Kuitenkin ohjelmassa olisi vielä paljon kehitettävää. Suurin kehityksen kohde olisi todennäköisesti useampien objektien samaan aikaan piirtämisessä. Tätä varten ohjelman piirrettävien objektien luokkia pitäisi parantaa pitämään paremmin kirjaa käytettävistä varjostimista, tekstuureista ja mallimatriiseista. Tällöin olisi hyvä myös luoda uusi luokka pitämään kirjaa piirrettävistä objekteista. Tämä synnyttäisi myös uusia ongelmia, joihin ei ohjemaa luodessa ehtinyt törmätä, kuten se, miten varjostimien yhteisiä muuttujia tulisi päivittää piirtokutsujen välissä, jos eri objektit käyttävät eri varjostimia, joilla annetaan aivan erilaiset yhteiset parametrit.

Ohjelmaan olisi myös hyvä luoda paremmat mahdollisuudet tekstuurien ja 3D-mallien käyttämiseen suoraan tietokoneen muistista, jolloin ohjelman ei aina tarvitsisi luoda uusia ja niitä ei tarvitsisi turhaan pitää ohjelman muistissa. Tämä olisi erityisen hyödyllistä tekstuureiden kanssa, koska jopa kohtuullisen pienet tekstuurit vievät suuren määrän muistia. Tietenkin tässä pitää ottaa huomioon, että tekstuurin muokkausta varten sen täytyy olla ohjelman muistissa. Mutta olisi kuitenkin hyvä, jos tekstuurin voisi vapauttaa muistista silloin, kun sitä ei kaivata.

Sovellukseen voitaisiin myös luoda enemmän erilaisia tekstuureja generoivia ja muokkaavia funktioita. Ohjelmaan voitaisiin luoda myös parempia varjostimia, jotka pystyisivät paremmin käyttämään varjostimille annettuja parametreja ja ohjelman generoimia tekstuureita.

4 Yhteenveto

Insinöörityössä pyrittiin syventämään käsitystä siitä, miten tietokone muodostaa kuvia näytölle. Saatiin parempi ymmärrys siitä, miten tietokonegrafiikka on kehittynyt vuosien varrella ja miten tietokonegrafiikka toimii nykyään. Työssä tutustuttiin tarkemmin grafiikan piirtämisen eri vaiheisiin, kuten siihen, miten tietokoneohjelma vie piirrettävää dataa näytönohjaimelle ja miten näytönohjaimille vietyä dataa pystytään käsittelemään varjostinohjelmien, kuten verteksivarjostimen ja fragmenttivarjostimen, kanssa.

Työtä varten toteutettu graafinen sovellus antoi käytännön kokemusta siitä, miten OpenGL käsittelee piirrettävää dataa ja miten tähän dataan päästään käsiksi varjostinohjelmien puolella. Työssä oli tärkeää selvittää, mitä kaikkea tarvitaan grafiikan piirtämiseen, koska kun tämä ymmärretään, on helpompi tehdä piirtämiseen sopivia 3D-objekteja ja tekstuureita.

Sovelluksesta muodostui työn aikana hyvä työkalu erilaisten tekstuurien luomiseen, kuten erilaisten kohinatekstuurien generointiin. Sovellus on myös hyvä pohja uusien tekstuurin generointifunktioiden luontiin. Monet sovelluksen tekstuureja käsittelevät funktiot toimivat hyvin, mutta osa niistä tuotti ongelmia ja pisti pohtimaan uusia ratkaisuja.

Työssä jäi kuitenkin vielä epäselväksi, miten useampien objektien piirtäminen ja useampien eri varjostimien käyttäminen vaikuttaa ohjelman toimintaan ja mitä siinä tulisi ottaa huomioon. Tämän tutkiminen ja toteutus voisi olla hyvä suunta jatkaa sovelluksen kehittämistä. Sovellukseen voisi tulevaisuudessa lisätä myös mahdollisuuden ladata 3D-objekteja ja tekstuureita tietokoneen muistista. Ohjelmaan voisi lisätä myös 3D-objektien muokkausmahdollisuuksia, kuten 3D-objektin yksittäisten verteksien siirtelyn ohjelman suorittamisen aikana.

Lähteet

1 Cathode. Verkkoaineisto. History-Computer. < https://history-computer.com/ModernComputer/Basis/cathode.html>. Luettu 20.1.2020.

2 1958. Verkkoaineisto. Computer History Museum. < https://www.computerhistory.org/timeline/1958/>. Luettu 20.1.2020.

3 The very beginning of the digital representation – Ivan Sutherland Sketchpad. 2018. Verkkoaineisto. BIM A+. < https://bimaplus.org/news/the-very-beginning-of-the-digital-representation-ivan-sutherland-sketchpad/>. Luettu 24.01.2020.

4 Scetchpad. 2020. Verkkoaineisto. Wikipedia. < https://en.wikipedia.org/wiki/Sketchpad>. Luettu 24.1.2020.

5 Petzold, Charles. 2000. Code: The Hidden Language of Computer Hardware and Software. E-kirja. Microsoft Press.

6 Landley, Rob & Raymond Eric. 2004. The Art of Unix Usability. Verkkoaineisto. Catb. < http://www.catb.org/~esr/writings/taouu/html/ch02s05.html>. Luettu 26.1.2020.

7 Xerox Alto. Verkkoaineisto. The Interface Experience. < http://interface-experience.org/objects/xerox-alto/>. Luettu 24.1.2020.

8 Arcade Game. 2020. Verkkoaineisto. Wikipedia. < https://en.wikipedia.org/wiki/Arcade_game>. Luettu 24.1.2020.

9 Space Invaders. Verkkoaineisto. Giantbomb. < https://www.giantbomb.com/space-invaders/3030-5099/>. Luettu 26.1.2020.

10 Smith, Ernie. 2017. Life In Four Colors. Verkkoaineisto. Tedium. < https://tedium.co/2017/06/15/ibm-pc-cga-graphics-cards-legacy/>. Luettu 4.2.2020.

11 Video Graphics Array. 2020. Verkkoaineisto. Wikipedia. < https://en.wikipedia.org/wiki/Video_Graphics_Array>. Luettu 4.2.2020.

12 Luten, Eddy. Preface: What is OpenGL? Verkkoaineisto. OpenGLBook. < https://openglbook.com/chapter-0-preface-what-is-opengl.html>. Luettu 25.1.2020.

13 Fifth generation of video game consoles. 2020. Verkkoaineisto. Wikipedia. < https://en.wikipedia.org/wiki/Fifth_generation_of_video_game_consoles>. Luettu 28.1.2020.

14 Singer, Graham. 2013. History of the Modern Graphics Processor. Verkkoaineisto. Techspot. < https://www.techspot.com/article/653-history-of-the-gpu-part-2/>. Luettu 24.2.2020.

15 FAQ. Verkkoaineisto. Khronos. < https://www.khronos.org/opengl/wiki/FAQ> Luettu 24.2.2020

16 List of 3D graphics libraries. 2020. Verkkoaineisto. Wikipedia. < https://en.wikipedia.org/wiki/List_of_3D_graphics_libraries> Luettu 24.2.2020.

17 Macy, Seth. 2016. What is Vulkan and what does it mean for the future of gaming? Verkkoaineisto. Techradar. < https://www.techradar.com/how-to/gaming/what-is-amd-vulkan-and-what-does-it-mean-for-the-future-of-gaming-1323469/2> Luettu 1.4.2020.

18 Yusov, Egor. 2017. Designing a Modern Cross-Platform Low-Level Graphics Library. Verkkoaineisto. Gamasutra. < https://www.gamasutra.com/blogs/EgorYusov/20171130/310274/Designing_a_Modern_CrossPlatform_LowLevel_Graphics_Library.php> Luettu 1.4.2020.

19 Shader modules. Verkkoaineisto. Vulkan Tutorial. < https://vulkan-tutorial.com/Drawing_a_triangle/Graphics_pipeline_basics/Shader_modules> Luettu 1.4.2020.

20 Galvan, Alain. 2017. A Review of Shader Languages. Verkkoaineisto. < https://alain.xyz/blog/a-review-of-shader-languages> Luettu 1.4.2020.

21 OpenGL Shading Language. 2019. Verkkoaineisto. Khronos. < https://www.khronos.org/opengl/wiki/OpenGL_Shading_Language> Luettu 14.2.2020.

22 Rendering Pipeline Overview. 2019. Verkkoaineisto. Khronos. < https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview> Luettu 1.6.2019.

23 Guetter, ChristophOpenGL. 2007. Non-rigid multi-modal registration on the GPU Verkkoaineisto. ResearchGate. < https://www.researchgate.net/figure/The-graphics-pipeline-in-OpenGL-consists-of-these-5-steps-in-the-new-generation-of-cards_fig1_235696712> Luettu 12.2.2020.

24 Z-buffering. 2017. Verkkoaineisto. Computer Hope. < https://www.computerhope.com/jargon/z/zbuffering.htm> Luettu 1.4.2020.

25 Verma, Akshat. 2020. Graphics Card Components & Connectors Explained. Verkkoaineisto. Graphics Card Hub. < https://graphicscardhub.com/graphics-card-component-connectors/> Luettu 1.4.2020.

26 Video Card. 2020. Verkkoaineisto. Wikipedia. < https://en.wikipedia.org/wiki/Video_card > Luettu 1.4.2020.

27 Framebuffer. 2020. Verkkoaineisto. Wikipedia. < https://en.wikipedia.org/wiki/Framebuffer#Page_flipping > Luettu 1.4.2020.

28 Ortiz, Leon. 2020. Types of 3D Modeling: Which Is Best for Your Needs? Verkkoaineisto. All3DP. < https://all3dp.com/2/types-of-3d-modeling/> Luettu 1.4.2020.

29 Zhou Qingnan. 2018. Mesh Boolean. Verkkoaineisto. < https://pymesh.readthedocs.io/en/latest/mesh_boolean.html> Luettu 1.4.2020.

30 Introduction to Polygon Meshes. Verkkoaineisto. Scratchpixel. < https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-polygon-mesh> Luettu 1.4.2020.

31 Polygon Mesh. 2020. Verkkoaineisto. Wikipedia. < https://en.wikipedia.org/wiki/Polygon_mesh> Luettu 1.4.2020.

32 Color depth. 2020. Verkkoaineisto. Wikipedia. < https://en.wikipedia.org/wiki/Color_depth> Luettu 1.4.2020.

33 Dasenden. 2016. CGA/EGA/VGA. Verkkoaineisto. T^{3} Interactive. < https://tcubedinteractive.wordpress.com/2016/08/20/cgaegavga/> Luettu 1.4.2020.

34 Indexed color. 2020. Verkkoaineisto. Wikipedia. < https://en.wikipedia.org/wiki/Indexed_color> Luettu 1.4.2020.

35 Texture filtering. 2020. Verkkoaineisto. Wikipedia. https://en.wikipedia.org/wiki/Texture_filtering> Luettu 1.4.2020.

36 Texture. 2019. Verkkoaineisto. Khronos. < https://www.khronos.org/opengl/wiki/Texture> Luettu 1.4.2020.

37 Mipmap. 2020. Verkkoaineisto. Wikipedia. < https://en.wikipedia.org/wiki/Mipmap> Luettu 1.4.2020.

38 VBO Indexing. Verkkoaineisto. opengl-tutorial. < http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-9-vbo-indexing/> Luettu 1.4.2020.

39 Eck, David. 2018. Some Linear Algebra. Verkkoaineisto. HWS. < http://math.hws.edu/graphicsbook/c3/s5.html> Luettu 1.4.2020.

40 Vries, Joey. 2014. Coordinate Systems. Verkkoaineisto. Learn OpenGL. < https://learnopengl.com/Getting-started/Coordinate-Systems> Luettu 1.4.2020.

41 Tutorial 3: Matrices. Verkkoaineisto. opengl-tutorial. < http://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/> Luettu 1.4.2020.

42 Perlin noise. 2020. Verkkoaineisto. Wikipedia. < https://en.wikipedia.org/wiki/Perlin_noise> Luettu 1.4.2020.

43 Flafla2. 2014. Understanding Perlin Noise. Verkkoaineisto. < https://flafla2.github.io/2014/08/09/perlinnoise.html> Luettu 1.4.2020.

44 Lowe, Jen & Vivo, Patricio. 2015. Cellular Noise. Verkkoaineisto. < https://thebookofshaders.com/12/> Luettu 1.4.2020.

45 Sanderson, Grant. 2017. Fractals are typically not self-similar. Verkkoaineisto. 3blue1brown. <https://www.3blue1brown.com/videos-blog/2017/5/26/fractals-are-typically-not-self-similar> Luettu 1.4.2020.