Anssi Seppä

3D-grafiikkamoottorin toteutus

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tietotekniikka

Insinöörityö

24.4.2018

Metropolia

Tekijä(t) Otsikko Sivumäärä Aika	Anssi Seppä 3D-grafiikkamoottorin toteutus 32 sivua 24.4.2018
Tutkinto	Insinööri (AMK)
Koulutusohjelma	Tietotekniikka
Suuntautumisvaihtoehto	Ohjelmistotekniikka
Ohjaaja(t)	Lehtori Miikka Mäki-Uuro

Insinöörityön tavoitteena oli toteuttaa reaaliaikaista kolmiulotteista tietokonegrafiikkaa piirtävä grafiikkamoottori. Grafiikkamoottorin oli tuettava erilaisia kolmiulotteisuutta ja realistisuutta korostavia tekniikoita.

Työn alussa käydään läpi kolmiulotteisen tietokonegrafiikan piirtämisen perusteita ja toteutetun grafiikkamoottorin grafiikkarajapinnaksi valitun OpenGL:n toimintaa ja käyttöä. Grafiikkamoottorin toteutuksessa käytettiin OpenGL-rajapinnan modernia versiota, jossa piirtäminen tapahtuu varjostinohjelmilla. Alun jälkeen avataan tarkemmin grafiikkamoottorin käsitettä. Lopussa esitellään tehdyn grafiikkamoottorin rakennetta ja toteutettuja tekniikoita, kuten valaistusta ja teksturointia.

Insinöörityön lopputuloksena luotu grafiikkamoottori kykeni piirtämään moniosaisia kolmiulotteisia virtuaalimaailmoja. Moottoriin lisättiin tuki tekstuuri- ja normaalikuvaukselle sekä valaistukselle. Valaistukseen toteutettiin erityyppisiä valoja, joiden avulla voitiin jäljitellä reaalimaailmassa nähtäviä erilaisia valonlähteitä. Valaistuksen piirtoa tehostettiin viivästetyllä piirrolla.

Avainsanat	Grafiikkamoottori, 3D-grafiikka, OpenGL

Author(s) Title Number of Pages Date	Anssi Seppä The implementation of a 3D graphics engine 32 pages 24 April 2018
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Software Engineering
Instructor(s)	Miikka Mäki-Uuro, Senior Lecturer

The purpose of this thesis was to implement a real-time 3D graphics engine. The graphics engine needed to be able to support different techniques to make the drawn image more realistic.

The beginning of the thesis goes through the fundamentals of how 3D graphics can be drawn on screen. Next the thesis explains how OpenGL graphics API can be used for the drawing. The implemented graphics engine used modern OpenGL as its graphics API. Drawing with the modern OpenGL focuses heavily on shader programs. After the fundamentals the thesis gives an overview of what a 3D graphics engine is. Finally, the implemented graphics engine is being presented to the reader.

The implemented graphics engine was able to draw complex three dimensional scenes with texture and normal mapping and lighting. The lighting module supported various kinds of lights that could imitate different light sources found in the real world. Furthermore, the rendering was made more performant by deferred shading.

Keywords	Graphics engine, 3D graphics, OpenGL

Sisällys

Lyhenteet

API	Application Programming Interface. Ohjelmointirajapinta.
EKJ	Entiteetti-komponentti-järjestelmä. Ohjelmoinnissa käytettävä ohjelman arkkitehtuurin suunnitteluperiaate.
GLSL	OpenGL Shading Language. Ohjelmointikieli, jonka avulla voidaan luoda OpenGL-varjostinohjelmia.
IBO	Index Buffer Object. Indeksoidussa piirrossa käytettävä OpenGL-indeksipuskuriobjekti.
OpenGL	Open Graphics Library. Tietokonegrafiikan piirtämiseen tehty ohjelmointirajapinta.
VAO	Vertex Array Object. OpenGL-kärkipistetaulukko-objekti, joka hallinnoi piirrossa tarvittavia kärkipistetietoja.
VBO	Vertex Buffer Object. OpenGL-kärkipistepuskuriobjekti, johon voidaan tallentaa piirrossa tarvittavat kärkipisteiden ominaisuudet.

1 Johdanto

Insinöörityön tarkoituksena oli tutkia 3D-grafiikan piirtämistä ja toteuttaa 3D-grafiikan piirtämiseen tarkoitettu grafiikkamoottori. Insinöörityö keskittyi pelimaailman kolmiulotteiseen piirtämiseen, mutta samaa grafiikkamoottoria on mahdollista käyttää myös muihin sovelluskohteisiin. Grafiikkamoottori tähtäsi uudelleenkäytettävyyteen ja sen tuli tukea erilaisia realistisuutta parantavia tekniikoita kuten tekstuurikuvausta ja valonmallinnusta.

3D-grafiikka yleistyy jatkuvasti mobiililaitteiden tehojen kasvun ja webteknologioiden kehittymisen myötä. Tietokoneiden näytönohjaimet nopeutuvat vuosi vuodelta, joten tietokoneella tuotetusta grafiikasta saadaan jatkuvasti realistisemman näköistä. Lisäksi erilaiset lisätyn todellisuuden ja virtuaalitodellisuuden sovellukset luovat kolmiulotteisuudelle uusia näkökulmia.

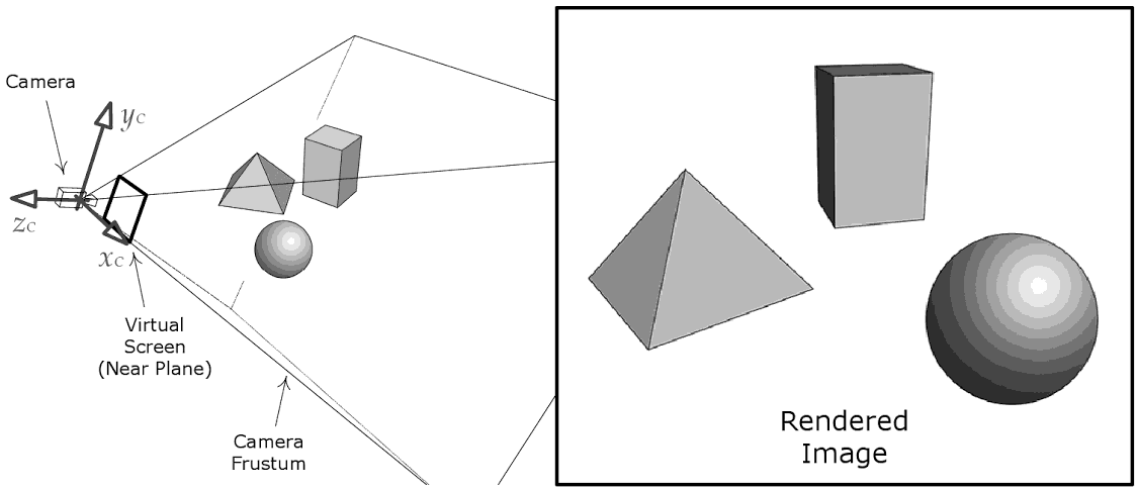
Kolmiulotteista grafiikkaa on käytetty jo pitkään apuna eri aloilla. Esimerkiksi tietokoneavusteisessa suunnittelussa jokin valmistettava tuote voidaan ensin suunnitella ja mallintaa kolmiulotteisena kuvana tietokoneen näytölle. Kolmiulotteisia visualisointeja voidaan hyödyntää eri prosessien simuloimisessa tai työntekijöiden kouluttamisessa. Rakennusalalla taloista luodaan kolmiulotteisia piirustuksia, ja arkkitehdit voivat katsella suunnittelemaansa taloa realistisessa valaistuksessa.

Insinöörityön alussa käydään yleisellä tasolla läpi 3D-grafiikan piirtämistä kappaleiden matemaattisesta määrittelystä aina ruudulle saakka. Tämän jälkeen tutustutaan grafiikan piirtämiseen tarkoitettuun OpenGL-ohjelmointirajapintaan, jonka avulla piirtäminen voidaan toteuttaa näytönohjaimella. Seuraavaksi avataan grafiikkamoottoria käsitteenä. Lopussa esitellään toteutettu grafiikkamoottori.

2 Reaaliaikainen 3D-grafiikka

Virtuaalinen kolmiulotteinen pelimaailma koostuu erilaisista matemaattisesti määritellyistä 3D-malleista. Pistettä, josta pelimaailmaa katsotaan, sanotaan kameraksi. Kameraa voidaan liikuttaa ja kääntää maailmassa. Kameran näkökentässä olevat kolmiulotteiset mallit muunnetaan kaksiulotteiselle kuvapinnalle, esimerkiksi tietokoneen näytölle. Kuva 1 havainnollistaa, kuinka kameran edessä kolmiulotteisessa maailmassa

olevat kappaleet piirretään näytölle. Maailmasta voidaan tehdä entistä todentuntuisempi erilaisilla maailmaan sijoitetuilla valonlähteillä, jotka vaikuttavat 3D-mallien pintoihin.



Kuva 1. Vasemmalla virtuaalinen maailma. Oikealla kameran läpi piirretty kuva. [1, s. 445]

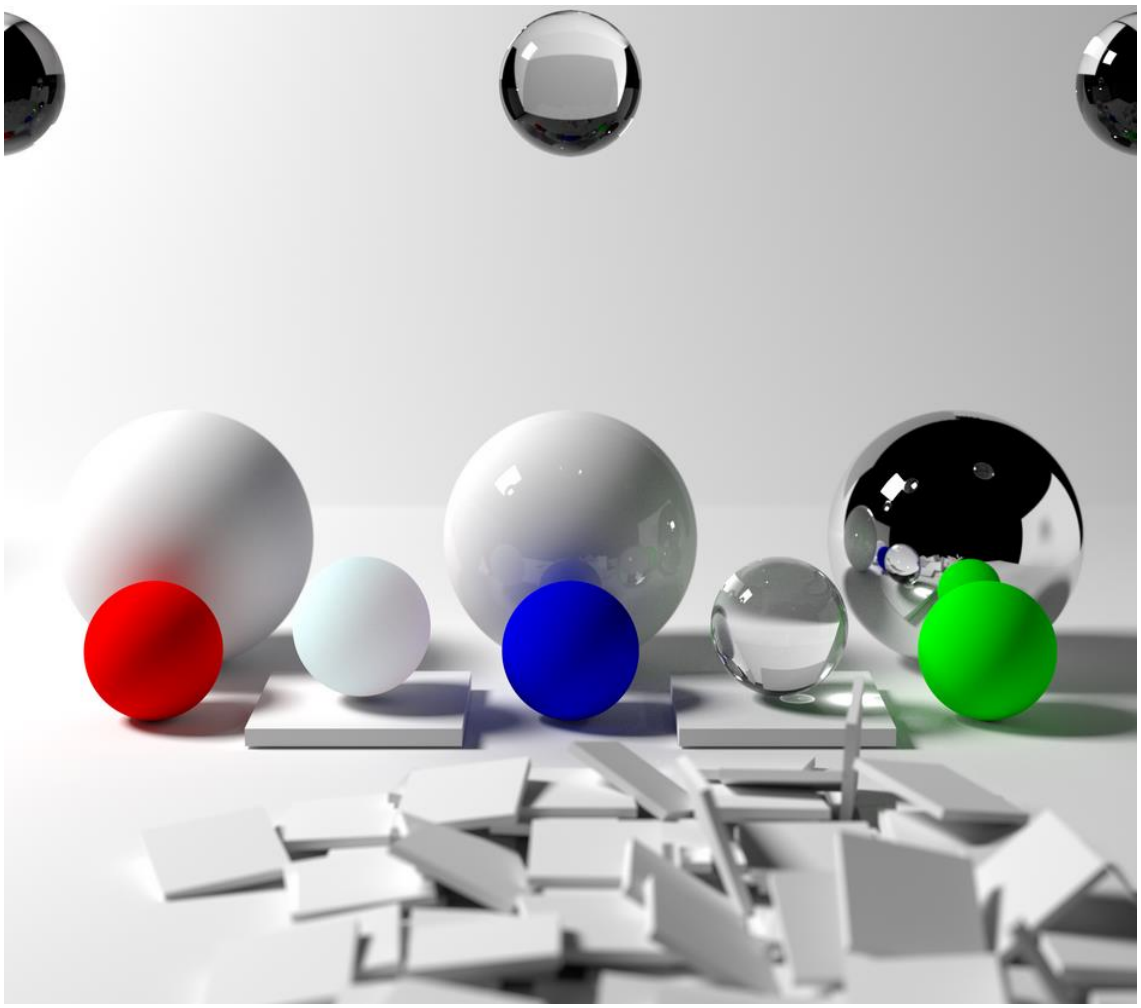
Reaaliaikaisella 3D-grafiikalla tarkoitetaan kuvan piirtämistä näytölle yhä uudestaan niin, että piirretty maailma näyttää liikkuvan reaaliajassa. Jotta pelin liike olisi riittävän sulavaa, täytyy kuvaa päivittää useita kertoja sekunnissa. Erilaisia piirtotekniikoita on paljon, mutta kaikkein realistisimman kuvan antavat tekniikat eivät sovellu reaaliaikaisuutta vaativiin ohjelmiin, kuten peleihin, koska yksittäisen kuvan piirtäminen vie liikaa aikaa. [1, s. 444-445.]

2.1 Säteenseuranta

Säteenseuranta on yksi vanhimmista piirtotekniikoista. Sitä käytetään yleisesti esimerkiksi animaatioelokuvissa. Tekniikan peruslähtökohtana on seurata säteitä kamerasta kuvapinnan yksittäisten pikseleiden läpi kolmiulotteiseen maailmaan. Jos säde osuu johonkin maailmassa sijaitsevaan kohteeseen, lasketaan säteen lähtöpikselille väri kohteen materiaalin ja maailmassa vaikuttavien valonlähteiden mukaan. [2, s. 70.]

Säteenseurantatekniikasta on erilaisia variaatioita, joista esimerkiksi polunseurantamenetelmällä epäsuoran valon vaikutusta voidaan mallintaa erittäin tarkasti, jolloin piirretystä kuvasta saadaan todella fotorealistinen [2, s. 429] (ks. kuva 2). Säteiden suuren lukumäärän ja niihin liittyvien laskuoperaatioiden takia

säteenseurantatekniikoita ei vielä kuitenkaan ole yksinään kyetty käyttämään pelien reaaliaikaiseen piirtämiseen kotikoneilla [2, s. 106].

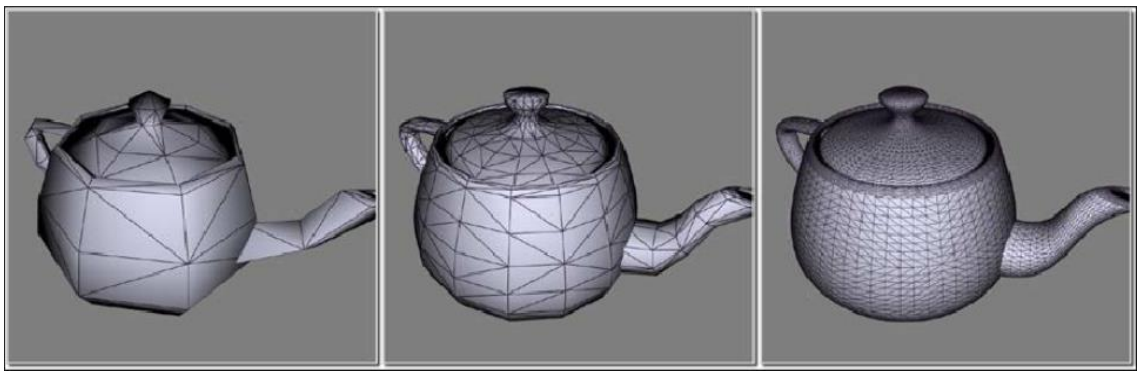


Kuva 2. Polunseurantamenetelmällä luotu kuva. [14.]

2.2 Rasterointi

3D-grafiikan rasteroinnilla tarkoitetaan prosessia, jossa kolmioista muodostuva kolmiulotteinen maailma piirretään kolmio kerrallaan näytölle [2, s. 10]. Rasterointi on nopeutensa ansiosta tällä hetkellä käytetyin reaaliaikaisen 3D-grafiikan piirtotekniikka. Ennen kuin näytönohjaimet nopeutuivat ja yleistyivät, rasterointi toteutettiin ohjelmallisesti tietokoneen prosessorilla. Prosessori on tarkoitettu yleiseen laskentaan, kun taas näytönohjaimet on suunniteltu varta vasten kuvien rasterointiin. Nykyään reaaliaikaisen 3D-grafiikan piirto tehdäänkin lähes poikkeuksetta näytönohjaimella. [2, s. 114-115; 1, s.494.]

Yksittäisen mallin pinta koostuu monikulmioverkosta (engl. mesh). Reaaliaikaisessa piirtämisessä monikulmiona käytetään kolmiota muun muassa siksi, koska sen kärkipisteet (engl. vertex) sijaitsevat aina samalla tasolla. Kärkipiste esitetään kolmialkioisena vektorina. Kärkipisteiden yhdistämää muotoa sanotaan usein myös primitiiviksi. Kolmioiden lisäksi muita piirrettäviä primitiivejä ovat esimerkiksi pisteet ja viivat. Kuten kuvasta 3 voidaan havaita, 3D-mallin pinta on sitä tarkempi mitä enemmän kolmioita siihen sisältyy, mikä taas toisaalta tarkoittaa sitä, että käsiteltävän tiedon määrä kasvaa. [3, s. 1-2; 1, s. 447-448.] Yksinkertaistettuna rasterointiprosessissa kärkipisteet muunnetaan kolmiulotteisesta maailmasta kaksiulotteiselle kuvapinnalle ja kärkipisteiden muodostamat kolmiot väritetään [2, s.10].



Kuva 3. 3D-mallin pinnan tarkkuus kolmioiden määrän kasvaessa [6, s. 254].

Muuntaminen tapahtuu vaiheittain koordinaatistoavaruudesta toiseen. Muunnosprosessi on esitetty kuvassa 4. Muunnos kahden koordinaatiston välillä voidaan esittää 4x4-matriisina. Yhteen matriisiin voidaan tallentaa mikä tahansa paikka ja orientaatio, ja jos mallin kaikki kärkipisteet kerrotaan tällä matriisilla, voidaan mallia kääntää ja sijoittaa se uuteen paikkaan. Lisäksi kaksi matriisia voidaan kertoa yhteen, jolloin tuloksena on matriisi, joka ilmaisee molempien matriisien paikan ja orientaation.

Kuva 4. Koordinaatistomuunnokset.

Aluksi 3D-malli sijaitsee sen paikallisessa koordinaatistossa, jossa sen kärkipisteet ovat sijoittuneet suhteessa mallin paikalliseen origoon. Seuraava koordinaatistoavaruus on itse pelimaailman avaruus, jossa kaikki mallit sijaitsevat suhteessa pelimaailman origoon. Tässä vaiheessa mallia voidaan kääntää, suurentaa tai pienentää, sekä siirtää.

Maailman koordinaatistoavaruuden jälkeen kärkipisteet siirretään kameran koordinaatistoon näkymämatriisin avulla, jolloin mallit sijaitsevat suhteessa kameraan, josta pelimaailmaa katsotaan. Koordinaatiston x- ja y- akselit ovat poikittain kuvapintaan ja katsojaan nähden, ja z-akseli kulkee kohti suoraan kuvapintaan päin. Näkymämatriisia muuttamalla pelimaailmaa voidaan katsella eri kohdista.

Viimeinen matriisimuunnos tapahtuu kameran koordinaatistosta kuvapinnan avaruuteen. Projektiomatriisi määrittää näkymälle raja-alueen, jonka ulkopuolelle jäävät kärkipisteet leikataan pois. Projektiokoordinaatistomuunnoksen jälkeen kärkipisteen sanotaankin olevan leikkauskoordinaatistossa.

Perspektiiviprojektiossa kauempana olevat mallit näyttävät pienemmiltä kuin lähempänä sijaitsevat mallit, jolloin piirrettyyn kuvaan saadaan syvyys. Perspektiiviprojektiota

käytetään kolmiulotteisten kuvien piirtämisessä. Näkymäalue on huipusta katkaistun pyramidin muotoinen.

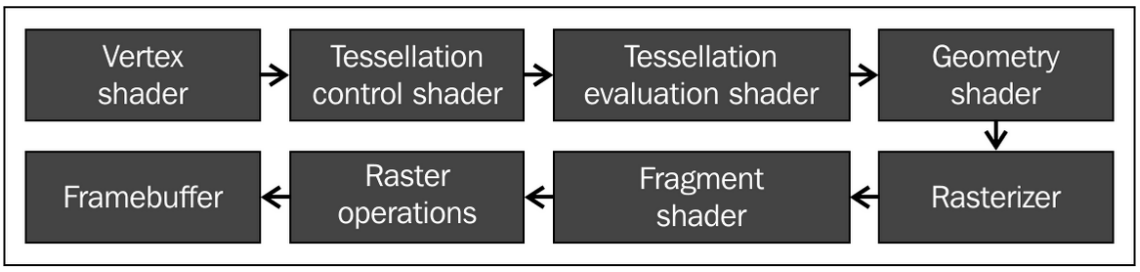
Yhdensuuntaisprojektiossa etäisyys kamerasta ei vaikuta mallin kokoon, joten sitä käytetäänkin enimmäkseen kaksiulotteisen grafiikan piirtämiseen. Yhdensuuntaisprojektion näkymäalue on suorakulmion muotoinen. [4, s. 58-81.]

3 OpenGL-rajapinta

OpenGL on alustariippumaton rasterointiin perustuva ohjelmointirajapinta. Silicon Graphics julkaisi ensimmäisen OpenGL-version vuonna 1992. Nykyisin OpenGL:n kehityksestä vastaa Khronos Group, joka on eri yritysten yhteenliittymä. [4, s. 6-7.]

3.1 Liukuhihna

OpenGL-rajapinnan avulla laitteiston näytönohjainta pystytään ohjaamaan ilman sen sisäisen toiminnan tuntemista. Piirtoprosessi etenee liukuhihnamaisesti (ks. kuva 5) erilaisia välivaiheita pitkin. [4, s. 3-6.]



Kuva 5. Grafiikan ohjelmoitava liukuhihna [5, s. 40].

Liukuhihnan välivaiheiden muokkaus oli rajattu määrättyihin operaatioihin ennen OpenGL-versiota 2.0. Esimerkiksi valaistus oli toteutettu täysin OpenGL-rajapinnan sisällä. Versio 2.0 toi mukanaan varjostinohjelmat (engl. shader), joilla näytönohjaimen toimintaa voidaan ohjelmoida GLSL-ohjelmointikielellä. Nykyaikaiset näytönohjaimet sisältävät monia varjostinytimiä, jotka suorittavat varjostimia samanaikaisesti. [6, s. 8.]

Liukuhihna alkaa kärkipisteiden käsittelyvaiheesta. Kärkipistevarjostin (engl. vertex shader) ajetaan näytönohjaimella kärkipiste kerrallaan. Varjostimella voidaan muokata

erilaisia kärkipisteen ominaisuuksia. Kärkipistevarjostimelle tulevat ominaisuudet saadaan OpenGL-rajapintaa käyttävältä ohjelmalta. [6, s. 25.] Keskeisin varjostimen tehtävä on kertoa kärkipisteen paikkavektori muunnosmatriiseilla [1, s. 496].

Tesselaatio- ja geometriavarjostimet ovat piirtämisen kannalta valinnaisia. Tesselaatiovarjostimilla voidaan muuttaa 3D-mallin pinnan tarkkuutta lisäämällä tai vähentämällä primitiivien määrää. Tesselaatiota voidaan käyttää esimerkiksi piirtämällä lähempänä näkyvät kohteet tarkempina ja kauempana näkyvät kohteet huonommalla tarkkuudella. Geometriavarjostin suoritetaan kerran jokaista primitiivimuotoa kohden, ja sillä voidaan luoda uusia, ja jopa erityyppisiä, primitiivejä. [6, s. 215-219.]

Geometriavarjostimen jälkeen kärkipisteiden paikat on muunnettu haluttuun koordinaatistoon ja yksittäiset kärkipisteet on koostettu kokonaisiksi primitiiveiksi, esimerkiksi kolmioiksi. Ennen pikselivarjostinta (engl. fragment shader) OpenGL suorittaa muutaman ei-ohjelmoitavan vaiheen. Näkyvän alueen ulkopuolelle jäävät kärkipisteet leikataan pois ja loput muunnetaan näytön koordinaatistoon. Tämän jälkeen voidaan vielä leikata kokonaisia kolmioita pois lopullisesta piirrettävästä pinnasta. Usein 3D-mallien etusivun takana piilossa oleva takasivu halutaan jättää piirtovaiheesta pois. Tähän vaiheeseen voidaan vaikuttaa ohjelmassa ennen piirtokomennon suorittamista määrittämällä kolmioiden etu- tai takasivu leikattavaksi (engl. face culling). Kolmion etu- tai takasivu määräytyy kolmion kärkipisteiden järjestyksen perusteella. Jäljelle jääneet kolmiot päätyvät rasterointi vaiheeseen. Rasteroinnissa määritetään näytön pikselit, jotka jäävät kolmioiden sisälle.

Viimeinen ohjelmoitava vaihe liukuhihnalla on pikselivarjostin. Varjostin suoritetaan kerran jokaista rasteroinnin tuloksena syntynyttä pikseliä kohden. Varjostimen lopputuloksena saadaan yksittäisen kuvapuskurin pikselin väri.

Ennen kuvapuskuria pikseli viedään vielä erilaisten testien läpi. Pikselitestien toimintaan voidaan vaikuttaa asettamalla niille erilaisia tiloja. Syvyystestissä uuden pikselin z-koordinaattia verrataan kuvapuskurin syvyyspuskurissa olevan pikselin z-koordinaattiin. Testi voidaan asettaa kokonaan pois päältä tai sen vertailufunktiota voidaan muuttaa. Testien lisäksi voidaan pikselin väri sekoittaa kuvapuskurissa olevan pikselin väriin. Tätä käytetään mm. läpinäkyvien mallien piirtämisessä.

Liukuhihna päättyy kuvapuskuriin. Oletuskuvapuskurin lisäksi voidaan myös luoda erillisiä kuvapuskureita ja käyttää niitä piirtämisen apuna. Oletuskuvapuskuri luodaan OpenGL-rajapinnan alkuvalmistelujen yhteydessä. [4, s. 38-47.]

3.2 OpenGL käytännössä

OpenGL-rajapinnan toimintaperiaatetta voidaan kuvata eräänlaisena tilakoneena. Rajapinnan käyttäminen alkaa OpenGL-kontekstin luomisesta. Konteksti pitää sisällään koko rajapinnan senhetkisen tilan, jota voidaan muuttaa erilaisilla funktiokutsuilla. Piirtokomennon yhteydessä kontekstin tila konkretisoituu kuvana kuvapuskuriin.

3.2.1 Puskuriobjektit

Erilaiset OpenGL-objektit ovat merkittävässä osassa, kun kontekstin tilaa halutaan muuttaa. Objektit pitävät sisällään jonkin laajemman kontekstin tilakokonaisuuden. Objekteja voidaan luoda glGen-alkuisilla OpenGL-rajapinnan funktiokutsuilla. Funktiokutsut palauttavat kokonaisluvun, jolla objektiin voidaan myöhemmin viitata. Funktio ei koskaan palauta arvoa nolla. Vastaavasti glDelete-alkuiset kutsut poistavat objektin.

Objektin sisältämää tilaa voidaan muuttaa asettamalla se ensin aktiiviseksi kontekstiin glBind-alkuisella funktiokutsulla, jolle annetaan parametrina objektin luonnissa saatu kokonaislukuviite sekä aktivoitavan objektin kohde kontekstissa. Tämän jälkeen kohteen tyyppiä vasten tehdyt rajapinnankutsut vaikuttavat aktivoituun objektiin. Kuitenkin useimmille kohteille kokonaisluvun arvo nolla tarkoittaa, että kyseiseen kohteeseen ei ole sidottu objektia. OpenGL-objektit voidaan jakaa tavallisiin objekteihin, jotka pitävät sisällään tietoa ja säiliöobjekteihin, jotka pitävät sisällään viittauksia tavallisiin objekteihin. [7.]

Puskuriobjektit ovat OpenGL-objekteja, joihin voidaan tallentaa monenlaista tietoa. Puskurin käyttö määräytyy sen mukaan, mihin OpenGL-kontekstin kohteeseen se asetetaan aktiiviseksi. Esimerkiksi jos puskurin tietoa halutaan käyttää kärkipistevarjostimen syötteenä, täytyy puskuri aktivoida GL_ARRAY_BUFFER-määreellä. Kuvassa 6 on esitetty kärkipistepuskurin luonti, kontekstiin aktivointi ja puskurin muistin alustus.

```
unsigned int vbo;
glGenBuffers(1, &vbo);
glBindBuffer(GL_ARRAY_BUFFER, vbo);

const float vertices[] =
{
    -0.5f, -0.5f, 0.0f, 1.0f, 0.0f, 0.0f,
    -0.5f, +0.5f, 0.0f, 0.0f, 1.0f, 0.0f,
    +0.5f, +0.5f, 0.0f, 0.0f, 0.0f, 1.0f,
    +0.5f, -0.5f, 0.0f, 0.0f, 1.0f, 1.0f
};

glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
```

Kuva 6. Kärkipistepuskurin luonti ja alustus.

Puskuri alustetaan glBufferData-funktiolla. Funktiolle annetaan viite liukulukutaulukkoon, joka pitää sisällään kahden kolmion muodostaman neliön kärkipisteiden paikat ja jokaiselle kärkipisteelle oma väri. Viimeinen glBufferData-funktion parametri määrittää, miten puskuria käytetään, eli luetaanko vai kirjoitetaanko puskuriin ja kuinka usein puskurin sisältöä muutetaan. GL_STATIC_DRAW soveltuu yksittäisten kappaleiden piirtämiseen, sillä kärkipistetieto kirjoitetaan puskuriin vain kerran.

Ennen kuin puskuria voidaan käyttää piirtämiseen, täytyy OpenGL:lle kertoa, missä kohdassa puskuria kärkipisteiden eri ominaisuudet (engl. attributes) sijaitsevat. Tämä tehdään kuvassa 7.

```
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), 0);

glEnableVertexAttribArray(1);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float),
    (void*)(3 * sizeof(float)));
```

Kuva 7. Ominaisuuksien paikkojen määrittäminen.

Ensimmäisen rivin glEnableVertexAttribArray-funktiokutsu sallii parametrina annetun ominaisuuden, jolloin OpenGL voi käyttää puskurin ominaisuusdataa kärkipistevarjostimella tätä viittausta vasten. Itse ominaisuuksien sijaintien määritys tapahtuu glVertexAttribPointer-kutsulla. Funktion ensimmäinen parametri on paikka varjostimella, mihin ominaisuusdata viedään. Toinen parametri kertoo, kuinka monta komponenttia yksi ominaisuuden arvo pitää sisällään. Kolmas parametri määrittää

arvojen tyypin. Neljännen parametrin epätosi arvo tarkoittaa, että puskurin dataa ei normalisoida. Viides parametri kertoo tavujen määrän kärkipisteiden välillä. Viimeinen parametri määrittää, kuinka monen tavun päästä kyseinen ominaisuus alkaa. [4, s. 92-102.]

OpenGL:n tarjoamat piirtokomennot voidaan jakaa indeksoituihin ja ei-indeksoituihin komentoihin. Ei-indeksoiduissa komennoissa piirrettävän kohteen kaikkien kolmioiden kärkipisteet määritetään puskuriin, ja piirtokomentoa suoritettaessa kaikki puskurin kärkipisteet käydään järjestyksessä läpi kärkipistevarjostimella. Indeksoiduissa piirtokomennoissa kärkipistepuskurin lisäksi luodaan erillinen indeksipuskuri (ks. kuva 8), joka sisältää viitaukset yksittäisiin kärkipistepuskurin kärkipisteisiin.

```
unsigned int ibo;
glGenBuffers(1, &ibo);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ibo);

const unsigned int indices[] =
{
    0, 1, 2,
    2, 3, 0
};

glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices,
    GL_STATIC_DRAW);
```

Kuva 8. Indeksointipuskurin luonti ja alustus.

Kuvan 8 alussa luodaan OpenGL-puskuriobjekti aivan kuten kuvan 7 kärkipistepuskurin luonnin yhteydessä. Tällä kertaa puskuri kuitenkin asetetaan aktiiviseksi GL_ELEMENT_ARRAY_BUFFER-kohteeseen, jolloin puskuria käytetään indeksipuskurina. Tämän jälkeen määritetään taulukollinen kokonaislukuja, jotka kertovat piirtovaiheessa kärkipistepuskurin järjestyksen.

Aikaisemmin luotuun kärkipistepuskuriin määriteltiin neliön neljä kärkipistettä. Tämä ei vielä riitä määrittelemään piirrettäviä kolmioita, koska OpenGL ei tiedä, mitkä kärkipisteet muodostavat kolmiot. Jos kärkipistepuskuriin olisi määritelty neliön sisältämien kolmioiden kaikki kuusi kärkipistettä järjestyksessä, ei indeksointipuskurille olisi ollut tarvetta, ja neliö voitaisiin piirtää käyttämällä glDrawArrays-muotoista piirtokomentoa. Indeksipuskurin etuna on se, ettei kaikkia piirrettävän kappaleen kolmioiden yhteisiä

kärkipisteitä tarvitse määritellä kärkipistepuskuriin montaa kertaa, jolloin puskurin koko pysyy pienenä.

Kuvan 8 indeksitaulukko sisältää kahden kolmion viittaukset kärkipistepuskurin taulukon alkioihin. Indeksipuskuri alustetaan samalla glBufferData-funktiolla, kuten kärkipistepuskurinkin tapauksessa. Indeksipuskuria käytettäessä täytyy piirtokomennon olla muotoa glDrawElements. [4, s. 231-234.]

Kärkipistetaulukko-objekti (engl. vertex array object) on OpenGL-säiliöobjekti, joka pitää sisällään viittaukset esimerkiksi yhden kappaleen tarvitsemiin puskureihin. Kärkipistetaulukon luonti ja aktiiviseksi asettaminen tapahtuu samankaltaisesti kuin puskuriobjektien tapauksessa, kuten kuvasta 9 voidaan havaita.

```
unsigned int vao;
glGenVertexArrays(1, &vao);
glBindVertexArray(vao);
```

Kuva 9. Kärkipistetaulukon luominen.

Kärkipistetaulukko pitää asettaa aktiiviseksi ennen kärkipistepuskurin ominaisuuksien määrittelyä ja indeksipuskurin sitomista kontekstiin, jotta taulukko tallentaisi viittaukset näihin. Kärkipistetaulukko helpottaa puskureiden käsittelyä ennen piirtokomentoa. [6, s. 30.]

3.2.2 Varjostimet

OpenGL-varjostimet ovat jossakin näytönohjaimen piirtoliukuhihnan vaiheessa suoritettavia ohjelmia. Varjostinohjelmat kirjoitetaan C-ohjelmointikieltä muistuttavalla OpenGL-varjostinkielellä (engl. OpenGL Shading Language). GLSL-kielellä kirjoitetut varjostimet voidaan kääntää OpenGL-rajapintaan määritetyllä kääntäjällä, jolloin lopputuloksena saadaan varjostinobjekti. Kuvassa 10 on esitetty kärkipistevarjostinobjektin luonti ja kääntäminen.

```
unsigned int vertexShader = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vertexShader, 1, vertexShaderSource, 0);
glCompileShader(vertexShader);
```

Kuva 10. Kärkipistevarjostimen luominen.

Kuvan 10 rivillä yksi luodaan tyhjä varjostinobjekti glCreateShader-funktiolla, jolle annetaan parametrina varjostimen tyyppi, joka esimerkin tapauksessa on kärkipistevarjostin. Esimerkiksi pikselivarjostin voidaan luoda GL_FRAGMENT_SHADER-tyypillä. Seuraavalla rivillä varjostimeen liitetään varjostinohjelman lähdekoodi merkkijonona. Viimeinen rivi tekee itse kääntämisen.

Kuvassa 11 luodaan ja linkataan OpenGL-varjostinohjelmaobjekti. Ohjelmaobjektiin liitetään halutut varjostinobjektit glAttachShader-funktiolla. Viimeisellä rivillä varjostinobjektit yhdistetään näytönohjaimella ajettavaksi kokonaisuudeksi. Tässä vaiheessa varjostinobjektit voidaan tuhota glDeleteShader-funktiolla. Varjostinohjelma voidaan tuhota glDeleteProgram-funktiolla, kun sille ei enää ole tarvetta.

```
unsigned int shaderProgram = glCreateProgram();
glAttachShader(shaderProgram, vertexShader);
glAttachShader(shaderProgram, fragmentShader);
glLinkProgram(shaderProgram);
```

Kuva 11. Varjostinohjelmaobjektin luominen.

Kärkipistepuskuriin määritellyn neliön piirtämiseen näytölle tarvitaan vähintään sekä kärkipiste- että pikselivarjostin [4, s. 16-21]. Kuten kuvasta 5 nähdään, liukuhihna alkaa kärkipistevarjostimelta. Kuvassa 12 on esitetty yksinkertainen kärkipistevarjostin. Ensimmäisellä rivillä määritetään GLSL:n versio ja profiili. Core-profiili tarkoittaa, että varjostin ei käytä vanhentuneita GLSL:n ominaisuuksia. [4, s. 16.]

```
#version 440 core

layout(location = 0) in vec3 position;
layout(location = 1) in vec3 color;

out vec4 vs_color;

void main()
{
    vs_color = vec4(color, 1.0);
    gl_Position = vec4(position, 1.0);
};
```

Kuva 12. GLSL-kärkipistevarjostin.

Kuvan 12 seuraavilla kahdella layout-alkuisella rivillä on määritetty kärkipistevarjostimelle tulevat väri- ja paikkaominaisuudet. Layout-määreellä ominaisuuksille annetaan sijainti-indeksinumero. Paikkaominaisuudelle annetaan sijainti-indeksi nolla ja värille yksi. Tätä indeksiä käytettiin kärkipistepuskurin ominaisuuksien sijaintien määrittelyssä kuvassa 7. Ominaisuudet määritetään kärkipistevarjostimelle tulevaksi syötteeksi in-määreellä. In- ja out-avainsanoilla voidaan yleisesti määrittää varjostimelle tulevaa dataa ja seuraavalle liukuhihnan vaiheelle lähtevää dataa. [4, s. 28-31.]

Paikka- ja väriominaisuudet on esitetty kolme komponenttisena vektoria, jonka GLSL-tietotyyppi on vec3. Erikokoisten vektoreiden lisäksi muita GLSL:n tukemia perustietotyyppejä ovat muun muassa liukuluvut (float), kokonaisluvut (int) ja erikokoiset matriisit (mat*). [4, s. 188.]

Varjostimen suoritus tapahtuu main-funktiossa. Kuvan 12 kärkipistevarjostimen main-funktiossa väriominaisuudenarvo annetaan out-määreellä merkitylle vec4-tyyppiselle muuttujalle. Väriominaisuus määritettiin kärkipistepuskuriin rgb-värimallin mukaisesti kolme komponenttisena vektorina. Ominaisuus muutetaan vec4-tyyppiseksi ennen sen viemistä eteenpäin pikselivarjostimelle, jolloin siihen lisätään läpinäkyvyyttä säätelevä alfakanava.

Paikkaominaisuus tuotiin xyz-vektorina, joista z-muuttujan, eli syvyyden, arvot määriteltiin nollaksi. GLSL määrittelee erilaisia sisäänrakennettuja gl-alkuisia muuttujia. Kuvan 12 esimerkissä kärkipisteiden paikkavektorin arvo asetettiin gl_Position-muuttujaan, joka vie kärkipisteiden paikat leikkauskoordinaatistoon. [4, s. 17.]

Myös väri siirrettiin vs_color-muuttujassa eteenpäin liukuhihnalla. Kuvan 13 pikselivarjostimessa väri saadaan saman nimiseen in-muuttujaan. Kärkipisteeltä tulevan värimuuttujan arvo interpoloidaan kolmion sisällä kyseiselle pikselille. Main-funktiossa pikselin väri siirretään color-nimisessä out-muuttujassa eteenpäin ja lopulta väri päätyy näytölle. [6, s. 27.]

```
#version 440 core

in vec4 vs_color;

out vec4 color;

void main()
{
    color = vs_color;
};
```

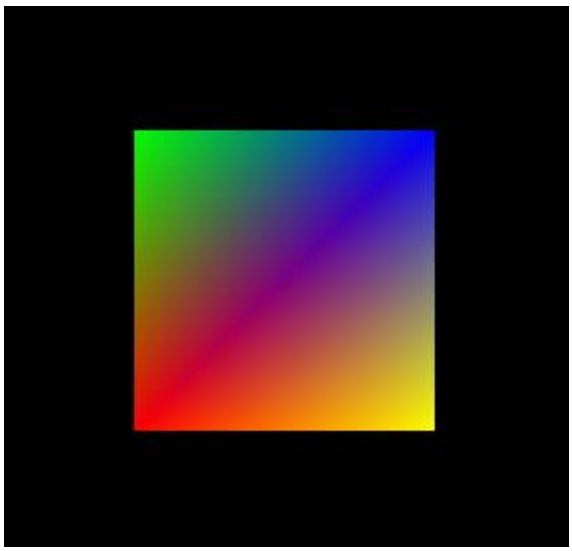
Kuva 13. GLSL-pikselivarjostin.

Nyt kun piirrettävä kappale on tallennettu puskureihin ja varjostimet on määritetty, voidaan siirtyä itse piirtovaiheeseen. Ennen piirtoa asetetaan halutun kappaleen kärkipistetaulukko aktiiviseksi, sekä otetaan jokin varjostinohjelma käyttöön. Piirtovaihe on esitetty kuvassa 14. [6, s. 31.]

```
glUseProgram(shaderProgram);
glBindVertexArray(vao);

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

Kuva 14. Piirtokomento

Lopputuloksena saadaan kuvan 15 mukainen värikäs neliö. Esimerkissä esitettiin vain tarvittavat OpenGL-komennot, mutta toimivan piirto-ohjelman luomiseen tarvitaan myös OpenGL-kontekstin ja ikkunan luonti ja Windows-alustalla myös OpenGL-funktioiden lataus. Näihin toimenpiteisiin on esitetty muutama kirjasto luvussa 5. [6, s. 9-10.]



Kuva 15. OpenGL-rajapinnalla piirretty neliö.

Kuten kuvasta 15 voidaan havaita, pikselivarjostimelle tulevat väriarvot ovat kärkipisteiden läheisyydessä samat kuin kärkipistepuskurin arvot, mutta sekoittuvat toisiinsa pikselin ollessa kärkipisteiden välisellä alueella. Toinen tapa, jolla tietoa voidaan viedä varjostimille, on käyttää uniform-muuttujia. Uniform-muuttujien arvot pysyvät samana varjostinohjelmien suorituksen aikana, joten esimerkiksi projektio, näkymä- ja mallimuunnosmatriisit ovat niille hyvä käyttökohde. Lisäksi uniform-muuttujien avulla tietoa voidaan viedä suoraan mihin tahansa varjostimeen. Uniform-muuttuja voidaan määrittää varjostimessa uniform-avainsanalla. [6, s. 37-38.]

Uniform-muuttujien arvot tuodaan varjostimille OpenGL-rajapintaa käyttävältä ohjelmalta. Tähän tarvitaan muutettavan uniform-muuttujan sijainti, joka voidaan joko määritellä käyttämällä layout-määrettä, tai sijaintia voidaan kysyä varjostinohjelmalta glGetUniformLocation-funktiolla. Ennen piirtokomentoa muuttujan arvoa voidaan muuttaa glUniform-alkuisilla funktioilla riippuen muuttujan tietotyypistä. [4, s. 103-108.]

4 Grafiikkamoottorin käsite

Pelimoottori on ohjelmakirjasto, jonka avulla voidaan luoda erilaisia pelejä. Se tähtää uudelleenkäytettävyyteen tarjoamalla peleissä usein käytetyt osat, kuten esimerkiksi grafiikan piirron, fysiikat ja äänet. Eri osat pyritään usein eristämään toisistaan itsenäisiksi kokonaisuuksiksi. Moottorin avulla uuden pelin kehittämistä ei tarvitse aloittaa kokonaan alusta. [1, s.11.]

Grafiikkamoottorilla tarkoitetaan sitä pelimoottorin osaa, joka on vastuussa grafiikan piirtämisestä näytölle. Yksi yleinen tapa rakentaa grafiikkamoottorin arkkitehtuuri on tehdä se kerroksittain, jolloin piirtodata siirtyy ylemmiltä kerroksilta kohti alinta kerrosta. Kerrokset voidaan jakaa edelleen pienempiin komponentteihin.

Alimman kerroksen tehtävä on piirtää sille syötetyt geometriat mahdollisimman nopeasti. Yksi sen komponenteista pitää sisällään jonkin grafiikkarajapinnan, esimerkiksi Direct3D tai OpenGL, työlään hallinnoimisen, jolloin grafiikkamoottorin loppukäyttäjän ei välttämättä tarvitse tietää alhaisen tason grafiikkarajapinnasta mitään.

Alin piirtokerros sisältää myös piirron kannalta välttämättömät projektio- ja näkymämatriisit. Alimman kerroksen muut komponentit yhdessä hallinnoivat näytönohjaimen ja varjostimien tiloja. Samassa piirtopaketissa geometrian kanssa kerrokselle syötetään myös tieto siitä, miten geometria tulisi piirtää. Jokaiseen geometriaan on yhdistetty materiaali ja jokaiseen geometriaan vaikuttaa tietty määrä valon lähteitä. Materiaalit sisältävät tiedon piirtämisessä käytettävistä tekstuureista ja varjostimista sekä grafiikkarajapinnan tilasta.

Alin kerros huolehtii vain ja ainoastaan sille syötettyjen erilaisten geometrioiden piirtämisestä näytölle. Koko virtuaalimaailma on kuitenkin vain harvoin kerralla näkyvissä. Tämän takia tarvitaan ylemmän tason näkyvyydenhallintakerros, joka rajoittaa objektien syöttämistä alemmalle piirtokerrokselle sen mukaan, ovatko objektit näkyvissä vai eivät. Erilaisia näkyvyydenhallintamenettelyitä on monia. Yksinkertaisimmillaan objektit voidaan jättää piirtämättä, jos ne eivät sijaitse näkymäalueen sisällä. Laajoihin ja yksityiskohtaisiin maailmoihin tarvitaan kuitenkin usein monia muitakin tekniikoita.

Pelkän 3D-mallien piirron lisäksi nykyaikaiset grafiikkamoottorit tukevat laajasti erilaisia visuaalisia tehosteita. Tällaisia tehosteita ovat muun muassa savun ja tulen tuottamiseen käytetyt hiukkastehosteet (engl. particle systems), eri pintoihin ilmestyvät jäljet (engl. decal systems), liikkuvien objektien varjot ja erilaiset jälkitehosteet, kuten hehkutehoste (engl. bloom).

Jotkin näistä tehosteista luodaan erillisessä komponentissa ja syötetään alimmalle piirtokerrokselle. Toiset tehosteet voidaan toteuttaa itse piirtokerroksessa. Jälkitehosteet nimensä mukaisesti luodaan piirretyn kuvapuskurikuvan päälle.

Kolmiulotteisen grafiikan lisäksi tarvitaan usein myös kaksiulotteista grafiikkaa antamaan käyttäjälle lisäinformaatiota. Tällä tarkoitetaan esimerkiksi kolmiulotteisen maailman päälle piirrettyä käyttöliittymävalikkoa. Lisäksi erilaiset kaksiulotteiset ikkunat ja valikot ovat tärkeitä kehitysvaiheen työkaluja, joiden avulla grafiikkamoottorin toimintaa voidaan ohjata ja seurata. [1, s. 40-44.]

5 Grafiikkamoottorin toteutus

Grafiikkamoottori toteutettiin C++-ohjelmointikielellä ja OpenGL-rajapinnalla. Jotta itse grafiikan luomiseen päästiin käsiksi mahdollisimman nopeasti, käytettiin moottorin apuna muutamia alustariippumattomia kirjastoja.

Moottoria kehitettiin Windows-ympäristössä, joten uudempien OpenGL-laajennusten tuominen ohjelmaan vaatii niiden erillisen latauksen. Lataus toteutettiin GLEW-kirjaston [9] avulla. OpenGL-rajapinnalla ei pystytä luomaan OpenGL-kontekstia tai hallinnoimaan ikkunointia, joten tähän tarkoitukseen käytettiin GLFW-kirjasto [9]. GLFW hoiti myös käyttäjäsyötteiden kuuntelemisen ja ajan mittaamisen. Matemaattisiin laskuihin käytettiin GLM (OpenGL Mathematics) -kirjastoa [10], ja 3D-mallitiedostojen lukemisen hoiti Assimp (Open Asset Import Library) -kirjasto [11]. Assimp kykenee lukemaan monia erilaisia 3D-mallitiedostoja, joten se on kätevä työkalu varsinkin kehitysvaiheessa.

5.1 Arkkitehtuuri

Grafiikkamoottoria varten kehitettiin yksinkertainen pelimoottorin runko. Runko toteutettiin entiteetti-komponentti-järjestelmä (engl. entity-component-system, ECS) arkkitehtuuria mukaillen. Tämän rinnalle kehitettiin myös erilaisia hallinnoimisluokkia esimerkiksi ikkunoinnille ja resurssien lataamiselle.

5.1.1 Entiteetti-komponentti-järjestelmä

Entiteetti-komponentti-järjestelmän perusajatuksena on poistaa erilaisten pelimaailmaan luotavien toimijoiden pitkiä ja monimutkaisia periytymisketjuja suosimalla koostamista. Jokainen pelimaailman toimija, entiteetti, koostetaan erilaisista komponenteista. Komponentti määrittää entiteetille erilaisia ominaisuuksia, joita olio-ohjelmoinnissa

periytettäisiin yliluokista. Koostamalla uusien entiteettien luominen voidaan toteuttaa hyvin joustavasti, jopa ohjelman ajon aikana.

Järjestelmät käsittelevät entiteettejä, jotka pitävät hallussaan järjestelmän toiminnallisuuden mukaisia komponentteja. Esimerkiksi piirtojärjestelmä käsittelee vain niitä entiteettejä, joilla on piirtämisen kannalta olennaiset komponentit. [12.]

Grafiikkamoottorin taustalle toteutetun pelimoottorin ydinluokka pitää sisällään kaikki pelimoottorin järjestelmät. Pelimoottori voidaan alustaa luomalla sen entiteettien hallinnoimisluokalle pelimaailman objektit. Kun halutut objektit on luotu, voidaan moottori käynnistää ydinluokan pääsilmukkaa ajavalla funktiolla. Pääsilmukka päivittää joka iteraatiolla kaikki ydinluokkaan alustetut järjestelmät.

Kaikki pelimaailman objektit ovat entiteettejä, joille on lisätty erilaisia toiminnallisuuksia määritteleviä komponentteja. Entiteetti on luokka, jolla on lista komponenteista. Komponentteja voidaan luoda periyttämällä tehty luokka Component-luokasta.

Esimerkiksi pelimaailman kamera voidaan määrittää entiteettinä, jolla on kamera- ja muunnoskomponentti. Muunnoskomponentti määrittää entiteeteille paikan, orientaation ja koon. Kamerakomponenttia tarvitaan määrittämään maailman katsomispiste. Kamerakomponentti pitää hallussaan näkymä- ja projektiomatriisit. Lisäksi komponentissa on vektorit, jotka määrittävät kameralle suunnat ylös ja eteenpäin. Näiden vektoreiden ja muunnoskomponentin paikkavektorin avulla voidaan laskea näkymämatriisi.

Kameralle toteutettiin kamerajärjestelmä, jonka avulla kameran eteenpäin-vektorin suuntaa voitiin muuttaa hiiren avulla. Katsantosuunnan lisäksi järjestelmällä pystyttiin muutamaan kameraentiteetin paikkavektoria näppäimistön avulla, jolloin kameraa pystyttiin liikuttamaan pelimaailmassa. Kamerajärjestelmä osoittautui tärkeäksi testaustyökaluksi.

Kolmas tärkeä komponentti oli kärkipisteverkkokomponentti. Kärkipisteverkkokomponentti piti sisällään kärkipistetaulukon ja listan aliverkoista. Jokainen aliverkko koostui sen pinnan ulkonäön määrittävästä materiaalista ja piirtokomennossa käytettävistä indeksointitiedoista. Materiaali sisälsi tekstuurit ja pinnan valaistuksen laskennassa käytetyt arvot.

Piirtojärjestelmä muodosti grafiikkamoottoriosuuden ytimen. Koko piirto osuus rakentui valaistuksen ympärille. Jokaisella pääsilmukan kierroksella piirtojärjestelmä käy läpi listan piirrettävistä pelimaailman entiteeteistä, muuttaa OpenGL:n tilaa ja tekee piirtokomennon. Piirtoon vaikuttavat kärkipistekomponenttien ja muunnoskomponenttien lisäksi kamera- ja valokomponentit. Valokomponentit kuvaavat erilaisia pelimaailman valonlähteitä. Piirtojärjestelmän piirtoa käydään tarkemmin läpi luvussa 5.2.5.

Moottoriin tehtiin myös yksinkertainen käyttöliittymäjärjestelmä. Käyttöliittymäjärjestelmän avulla kyettiin piirtämään yksinkertaisia käyttöliittymiä, joita käytettiin moottorin kehityksen aikana testidatan esittämiseen näytöllä. Esimerkiksi ruudunpäivitysnopeuden piirtämisellä näytölle oli suurta hyötyä piirron optimoinnissa. Käyttöliittymän piirto toteutettiin avoimen lähdekoodin IMGUI-kirjastolla.

5.1.2 Hallintaluokat

Toinen iso kokonaisuus oli erilaiset hallinnoimisluokat (engl. manager). Näillä tarkoitetaan luokkia, jotka hallitsevat jotain tiettyä ohjelman kokonaisuutta. Hallinnoimisluokat toteutettiin Singleton-sunnittelumallin mukaisesti.

Singleton-suunnittelumallilla tarkoitetaan luokkaa, josta voidaan luoda vain yksi olio. Tämä voidaan C++-ohjelmointikielellä toteuttaa niin, että piilotetaan luokan alustus ja kopiointifunktiot sekä luodaan staattinen funktio, joka palauttaa luokasta staattisen olion. Hallinnoimisluokkia tarvitaan ohjelmassa vain yksi jokaista, ja niitä saatetaan käyttää monissa eri järjestelmissä.

Kaikki pelimaailman objektit tallennettiin entiteetteinä entiteettejä hallinnoivaan luokkaan. Piirrettävät objektit ja niihin vaikuttavat valot tallennettiin kahteen eri listaan. Lisäksi luokassa on erillään kamera- ja taivaskomponentin sisältävät entiteetit.

Ikkunointi ja OpenGL-kontekstin luominen toteutettiin GLFW-kirjastolla. GLFW-kirjaston käyttö käärittiin ikkunoinnin hallintaluokkaan, joka tarjosi rajapinnan ikkunan koon ja kuvasuhteen saamiseen ja ikkunan päivittämiseen. Ikkuna ja OpenGL-konteksti luotiin luokan alustusfunktiossa.

Ikkunointiluokan päivitysfunktiota kutsuttiin pääsilmukan lopuksi. Päivitysfunktio vaihtaa etu- ja takapuskureiden paikkaa, jolloin takapuskuriin piirretty kuva näytetään ikkunassa.

Puskureiden vaihdon lisäksi päivitysfunktio käy läpi ikkunaan tehdyt tapahtumakutsut, esimerkiksi näppäinpainallukset.

Käyttäjäsyötteen kuunteleminen eriytettiin omaan luokkaansa. Käyttäjäsyötteen hallinnoimisluokka antaa tiedon hiirenliikkeistä ja hiiren näppäinten painalluksista, sekä näppäimistön syötteestä. Luokan alustusfunktiossa asetetaan takaisinkutsut GLFW:n funktioille glfwSetKeyCallback ja glfwSetCursorCallback, joita kutsutaan näppäinpainallusten ja hiirenliikkeen yhteydessä. Takaisinkutsufunktiot asettavat hiiren kursorin paikan ja tietyn näppäimen painetuksi hallinnoimisluokan luokkamuuttujiin, jolloin niitä voidaan kysyä muualta ohjelmasta isKeyPressed- ja getMousePosition-funktioilla.

Kaikki piirtämiseen käytetyt varjostimet luotiin ja tallennettiin sekä tuhottiin omassa luokassaan. Luokkaa käytettiin piirtojärjestelmässä palauttamaan oikea varjostin riippuen piirrettävän objektin materiaaliarvoista.

Myös 3D-mallidatan lataamiseen ja tallentamiseen luotiin oma luokka. 3D-mallitiedostot ladattiin avoimen lähdekoodin Assimp-kirjastolla. Assimp lukee 3D-mallitiedoston omaan tietorakenteeseensa, josta itselle merkitykselliset osat voidaan poimia.

5.1.3 OpenGL-luokat

OpenGL-rajapinnan kutsut ja käsitteet pyrittiin paketoimaan omiksi kokonaisuuksikseen. Paketointi tehtiin grafiikkarajapinnan eristämisen ja käytön helpottamisen takia. Kaikista OpenGL-objekteista tehtiin omat luokkansa. Objektien luonti ja tuhoaminen tapahtui luokkien konstruktoreissa ja destruktoreissa. Objektin luonnin yhteydessä saatu kokonaislukuviite tallennettiin luokkamuuttujaan. Lisäksi kaikilla luokilla oli funktiot objektin sitomiseksi OpenGL-kontekstiin ja siitä pois.

Kärkipistepuskuri ja indeksointipuskuri erotettiin omiin luokkiinsa. Nämä puskurit voitiin tallentaa kärkipistetaulukon luokkamuuttujiin. Eri kärkipisteiden ominaisuuksille voitiin luoda omat puskurit, joten kärkipistepuskureille oli kärkipistetaulukossa määritetty taulukko. Puskureiden tuhoaminen tehtiin kärkipistetaulukkoluokan destruktorissa.

Myös varjostimille tehtiin oma luokka. Varjostinohjelmaobjektin hallinnoimisen lisäksi luokka piti sisällään uniform-muuttujien päivittämisen. Ohjelmaobjektin luomisen jälkeen

konstruktori haki kaikkien aktiivisten uniform-muuttujien sijainnit varjostimissa ja niiden nimet. Haku toteutettiin glGetProgram-alkuisilla OpenGL-funktioilla. Muuttujien nimet ja sijainnit tallennettiin C++-standardikirjaston map-tietorakenteeseen. Ennen piirtokomentoa varjostimien uniform-muuttujat asetettiin erilaisten setUniform-funktioiden kautta antamalla funktiolle parametrina uniform-muuttujan nimi ja muutettava arvo.

5.2 Piirtotekniikat

Erilaisilla piirtotekniikoilla lopputuloksena syntyvästä kuvasta voidaan saada entistä todentuntuisempi. Lisäksi piirron järjestykseen vaikuttamalla piirtoa voidaan tietyissä tapauksissa nopeuttaa huomattavasti.

5.2.1 Valaistus

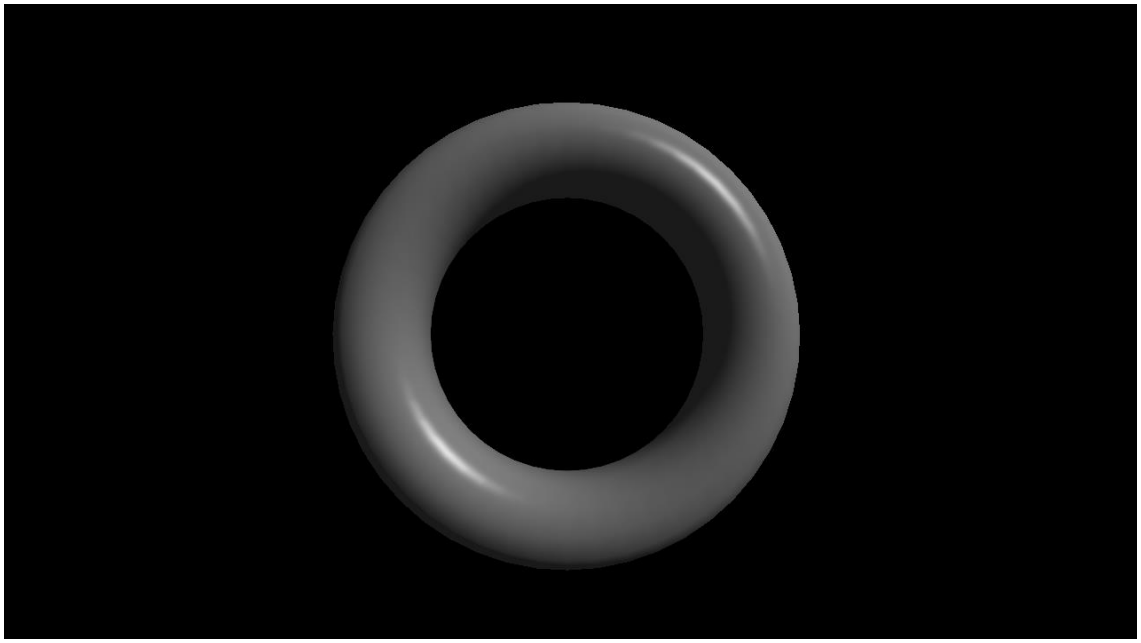
Valaistuksella on tärkeä rooli todentuntuisen maailman piirtämisessä. Yksi grafiikkamoottorin tärkeimmistä ominaisuuksista on mallintaa valon käyttäytymistä piirrettävässä maailmassa. Valonmallinnukseen on kehitetty erilaisia matemaattisia malleja.

Realistisin tulos saadaan globaaleilla valaistusmalleilla, joissa huomioidaan suoraan valonlähteestä tulevan valon lisäksi epäsuoran valon vaikutus maailmassa. Epäsuoralla valolla tarkoitetaan muista pinnoista kimmonnutta valoa. Epäsuoran valon laskeminen on erittäin raskasta, joten reaaliaikaisissa sovelluksissa se lasketaan usein etukäteen piirrettävän maailman liikkumattomille osille.

Liikkuvissa maailman osissa käytetään usein paikallisia valaistusmalleja, joissa otetaan huomioon vain valonlähteestä suoraan tuleva valo, jolloin vierekkäiset pinnat eivät vaikuta toistensa ulkonäköön. [1, s. 468-471; 13.]

Yksi yleisimmin käytetyistä paikallisista valaistusmalleista on Phong-valaistusmalli. Se määrittää kappaleen pinnasta heijastuneen valon ympäröivän valon (engl. ambient), diffuusin valon (engl. diffuse) ja spekulaarin valon (engl. specular) summana. Nämä kolme termiä määritetään kolme komponenttisina väreinä kappaleen materiaalille ja valolle.

Ympäröivällä valolla jäljitellään epäsuoraa valoa. Se valaisee kappaleen tasaisesti joka kohdasta. Ilman ympäröivää valoa varjokohdat jäisivät täysin mustiksi. Diffuusilla valolla tarkoitetaan valoa, joka heijastuu joka suuntaan kappaleen pinnasta. Toisin kuin ympäröivä valo diffuusin valon laskemisessa otetaan huomioon valon tulosuunta, jolloin kappaleen valosta poispäin olevat sivut jäävät varjoon. Spekulaarivalo muodostaa kirkkaita kohtia kiiltäväpintaisten kappaleiden pintaan. Kirkkaat kohdat muodostuvat, kun katselukulma on lähellä valon kimpoamiskulmaa. [1, s. 471-474; 4, s. 504-508.] Kuvassa 16 voidaan nähdä spekulaarin valon muodostamat valkoiset kiiltävät alueet kappaleen pinnalla, sekä diffuusin valon muodostamat varjostukset kappaleen sisäpinnalla.



Kuva 16. Phong-valaistusmallilla valaistu kappale.

Toteutetussa grafiikkamoottorissa käytettiin phong-valaistusmallia. Moottoriin voidaan määrittää yksi koko pelimaailman kattava ympäröivän valon arvo, joka otetaan huomioon, kun kappaleiden valaistusta lasketaan. Arvolla voidaan säätää pelimaailman kirkkautta.

Toinen grafiikkamoottoriin toteutettu valotyyppi oli suuntavalo (engl. Directional Light). Suuntavalon voidaan ajatella jäljittelevän aurinkoa. Sillä ei ole pelimaailmassa tarkkaa paikkaa, ja sen valonsäteet osuvat kaikkiin pelimaailman pintoihin samasta suunnasta.

Pistevalolla (engl. Point Light) tarkoitetaan valonlähdettä, jolla on tarkka paikka ja jonka valonsäteet lähtevät valon keskeltä joka suuntaan. Pistevalon teho heikkenee, mitä

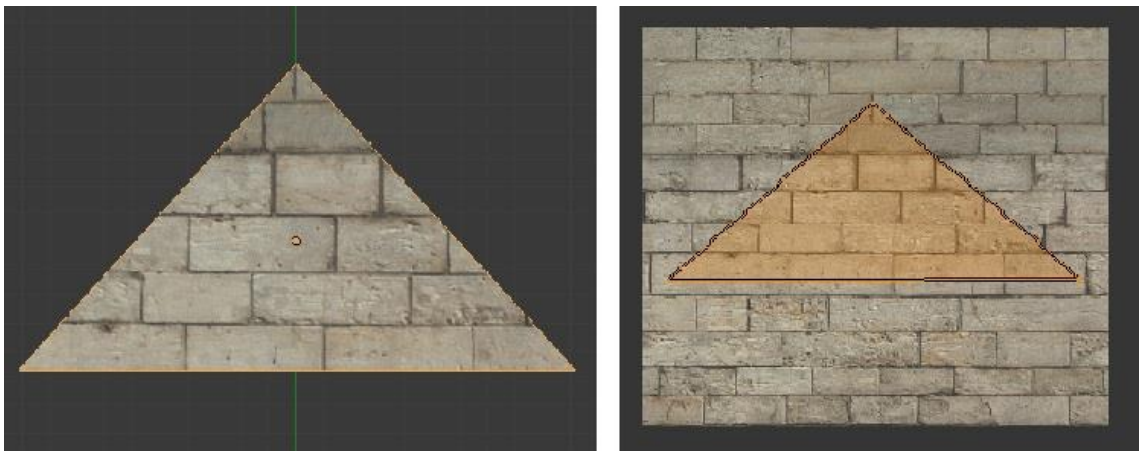
kauemmas valon keskipisteestä edetään. Pistevaloa voidaan ajatella ympäröivän pallomainen alue, jonka sisällä olevat pinnat valaistaan.

Viimeinen grafiikkamoottoriin mallinnettu valotyyppi oli kohdevalo. Kohdevalolla on sekä paikka että suunta. Kohdevalon teho heikkenee pistevalon tapaan, mutta sen vaikutusalue on kartion muotoinen. Kohdevalolla voitaisiin mallintaa esimerkiksi katuvalo.

5.2.2 Tekstuurikuvaus

Tekstuureita (engl. texture) käytetään reaaliaikaisessa piirtämisessä monenlaisiin eri tarkoituksiin. Yleisin käyttökohde on kuitenkin tekstuurikuvaus (engl. texture mapping), jossa kaksiulotteisen kuvan värit siirretään 3D-mallin pinnalle. [6, s. 117-118.] Tekstuurikuvaus on tehokas tapa lisätä pikselitason yksityiskohtia piirrettyihin pintoihin sen sijaan, että pinnat muodostettaisiin valtavasta määrästä värjättyjä kolmioita. [2, s.218.]

Aiemmin luvussa 3.2 piirretylle neliölle määritettiin värit viemällä yksittäisiä väriarvoja kärkipistevarjostimelle, josta ne jatkoivat eteenpäin pikselivarjostimelle. Tekstuurikuvauksessa väriarvot haetaan jonkin kuvan pikseleistä pikselivarjostimella. Väriarvopuskurin sijaan määritetään tekstuurikoordinaattipuskuri, jonka arvot viedään kärkipistevarjostimelle ja eteenpäin pikselivarjostimelle. Tekstuurin koordinaatit voidaan määrittää esimerkiksi 3D-mallinnusohjelmassa. Kuvasta 17 voidaan nähdä, kuinka kolmion kärkipisteille määritetään tekstuurikoordinaatit Blenderissä.



Kuva 17. Tekstuurikoordinaattien asettaminen Blender 3D-mallinnusohjelmassa.

Kuvassa 17 vasemmalla puolella on itse kolmion geometria ja oikealla puolella kolmion kärkipisteet asetettuna tekstuurin päälle. Kärkipisteille määritetään tekstuurin koordinaatistosta koordinaatit, joiden arvot vaihtelevat nollan ja yhden välillä. Koordinaatisto on normalisoitu nollan ja yhden välille, koska tekstuurien leveys ja korkeus voivat vaihdella.

Pikselivarjostimelle määritetään sampler2D-tyyppinen uniform-muuttuja, johon tekstuuriobjekti tuodaan. Pikselivarjostimen main-funktiossa tekstuurista haetaan väriarvo tekstuurikoordinaattien kohdalta GLSL:n texture-funktiolla. Funktiolle annetaan parametrina sampler2D-muuttuja ja kärkipistevarjostimelta saadut koordinaatit. [1, s. 463; 4, s. 147.]

Grafiikkamoottorin Texture-luokka piti sisällään OpenGL:n tekstuuriobjektin lataamisen, käytön ja poistamisen. Tekstuuriobjektin luominen tapahtuu samaan tapaan kuin muidenkin objektien luominen. Ensin tekstuurille luodaan viite glGenTextures-funktiolla ja viitteen avulla se asetetaan aktiiviseksi kontekstiin glBindTexture-funktiolla. Tämän jälkeen tekstuurille voidaan antaa erilaisia asetuksia glTexParameter-alkuisilla funktioilla. Näillä asetuksilla voidaan esimerkiksi määrittää, mitä tapahtuu, jos tekstuurikoordinaattien arvot ovat normalisoidun alueen ulkopuolella. Tekstuuria voidaan tällöin esimerkiksi toistaa tai peilata kappaleen pinnalla.

Kuvatiedoston väridatan muistiin lataamiseen käytettiin stb_image-nimistä kirjastoa. Kirjaston stbi_load-funktiolle annetaan parametrina kuvatiedoston nimi, muistiosoitteet kokonaislukumuuttujiin kuvan korkeudelle, leveydelle ja kanaville. Funktio palauttaa tiedostosta luetut pikselit taulukossa ja antaa kokonaislukumuuttujille vastaavat arvot.

Tekstuuriobjekti voidaan alustaa muistiin ladatun pikselidatan avulla. Tätä varten on glTexImage2D-funktio. Funktiolle annetaan parametreina tekstuurin kohde, yhden pikselin kanavien määrä, kuvan korkeus ja leveys, pikselidatan tyyppi ja viimeisenä itse pikselidata. Kun pikselidata on annettu tekstuuriobjektille, voidaan alkuperäinen pikselidata vapauttaa muistista stbi_image_free-funktiolla.

Tämän jälkeen tekstuurista voidaan luoda kooltaan pienempiä variaatioita glGenerateMipmap-funktiolla. Pienempi resoluutioisia tekstuureja käytetään kauempana kamerasta piirretyissä kappaleissa. Mipkuvauksen (engl. mipmapping) avulla teksturoinnin suorituskyky paranee, ja se myös parantaa piirretyn kuvan laatua poistamalla tekstuurin pikselitiheydestä johtuvia häiriöitä. [4, s. 153-154.]

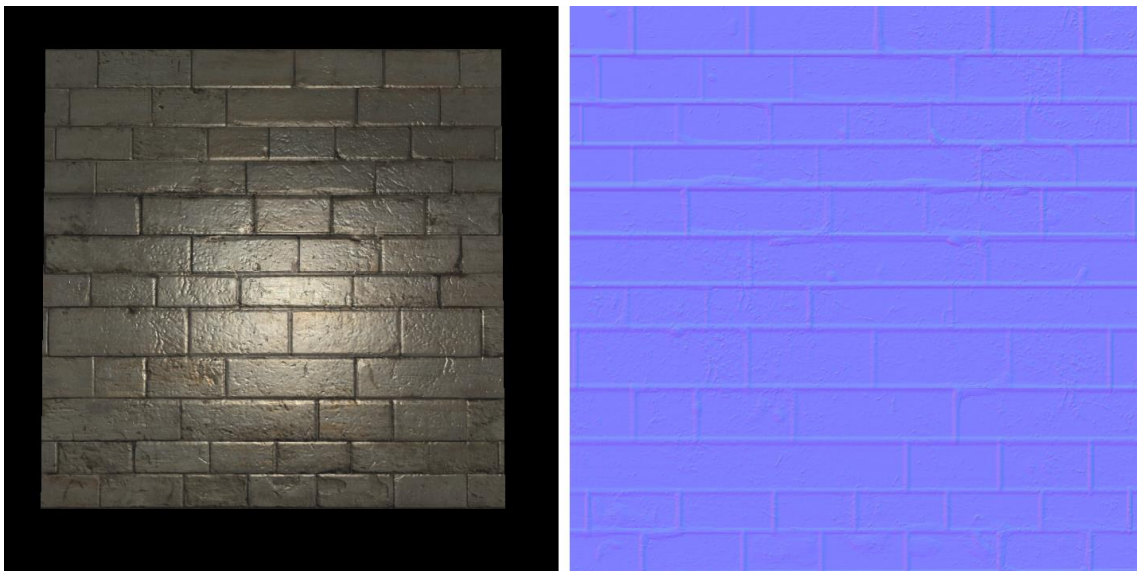
Ennen piirtokomentoa tekstuuriobjekti täytyy ottaa käyttöön määrittelemällä ensin tekstuurin yksikkö (engl. texture unit) glActiveTexture-funktiolla ja asettamalla tekstuuri aktiiviseksi GL_TEXTURE_2D-kohteeseen glBindTexture-funktiolla. Lisäksi varjostimen sampler2D uniform -muuttujalle täytyy kertoa tekstuurin yksikkö glUniform1i-funktiolla. Tekstuurin yksikkö tarvitaan, koska varjostimella voidaan käyttää monia eri tekstuureja samalla kertaa. Esimerkiksi normaalikuvaksessa varjostimelle määritetään väritekstuuri sekä pinnan normaaleja sisältävä normaalitekstuuri. [4, s. 150-151.] Grafiikkamoottorilla piirrettyjä teksturoituja 3D-malleja voidaan nähdä seuraavien lukujen kuvissa.

5.2.3 Normaalikuvaus

Normaalikuvauksessa valaistuslaskuissa käytettävä pinnan normaali sisällytetään tekstuuriin sen sijaan, että se tuotaisiin kärkipistevarjostimelle yhtenä kärkipisteenominaisuutena, ja interpoloitaisiin kolmion pikseleille. Pinnan normaali otetaan pikselivarjostimella normaalitekstuurista ja käytetään valaistuslaskuissa. Tällä tavalla pintaan saadaan pikselitason yksityiskohtia ilman, että kolmioiden määrää tarvitsee lisätä.

Normaalitekstuuri luodaan niin, että sen yksittäisten pikseleiden rgb-arvoihin tallennetaan pinnan kolmikomponenttinen suuntavektori. Kuten kuvan 18 oikeanpuoleisesta osasta voidaan havaita, normaalitekstuuri on enimmäkseen sininen, koska pinnan normaali suuntautuu tekstuurista poispäin, jolloin pikselin rgb-arvon viimeinen komponentti on suurin. [3, s. 178-179.]

Kuvan 18 vasemmalla puolella on kahdesta kolmiosta muodostuva neliö, jonka pinnan materiaaliin on määritetty kiviseinältä näyttävä väritekstuuri ja kuvan oikealla puolella oleva normaalitekstuuri. Valaistussa pinnassa voidaan havaita normaalitekstuurista johtuvia pieniä kolhuja, vaikka pinnan geometria on täysin sileä.



Kuva 18. Grafiikkamoottorilla valaistu normaalikuvattu pinta.

5.2.4 Kuutiotekstuuri

Kaksiulotteisten tekstuurien lisäksi OpenGL tukee myös kuutiotekstuureja (engl. cube map). Kuutiotekstuuri koostuu kuudesta erillisestä kaksiulotteisesta kuvasta, jotka muodostavat kuution kuusi sivua. Kuutioteksturointia käytetään usein, kun halutaan piirtää kaukana näkyvä ympäristö tai taivas.

Kuutiotekstuuri luodaan samaan tapaan kuin kaksiulotteinenkin tekstuuri, mutta OpenGL-kontekstin kiinnityskohteena käytetään GL_TEXTURE_CUBE_MAP-määrettä, ja glTexImage2D-funktiokutsuja tehdään kuusi, yksi jokaiselle sivulle. Myös kuutiotekstuurin tekstuurikoordinaatteja on kolme kahden sijaan. Tekstuurikoordinaattien voidaan ajatella olevan kolmikomponenttisia suuntavektoreita, jotka lähtevät kuution keskeltä kohti kuution sivuja.

Grafiikkamoottoriin toteutettiin taivaan piirto kuutioteksturoinnin avulla (engl. skybox). Tämä on esitetty kuvassa 19, jossa etualalla on piirretty 3D-malleja ja taustalla näkyvä taivas. Taivas on piirretty kuutionmuotoisen 3D-mallin sisäpinnalle niin, että tekstuurikoordinaatteina on käytetty kuution kärkipisteiden koordinaatteja. Kuutio piirretään muun maailman jälkeen kaiken muun taakse. [6, 136-142.]

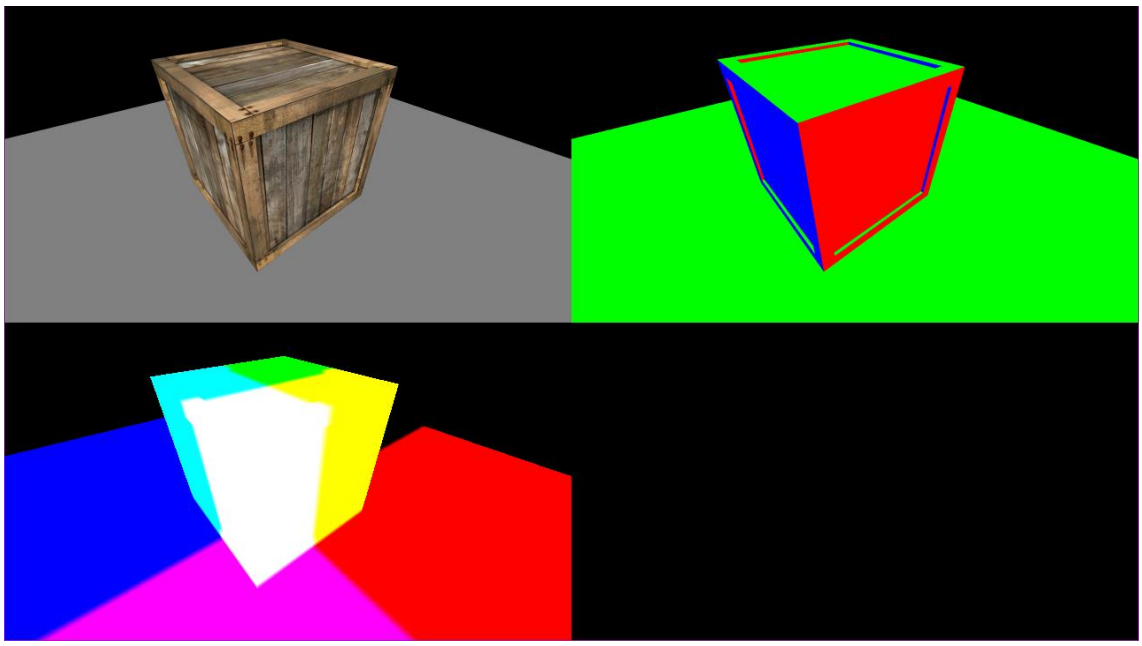
Kuva 19. Grafiikkamoottorilla piirretty kuva, jossa taustalla näkyy taivas.

5.2.5 Piirtopolut

Suoraviivaisin tapa piirtää kappaleita näytölle on yksittäisen kappaleen piirtokomennon yhteydessä käydä koko liukuhihna kerralla läpi kärkipistevarjostimesta kuvapuskuriin saakka niin, että mahdollisten valojen vaikutus lasketaan lopuksi pikselivarjostimella (engl. forward rendering). Tämä tapa on yksinkertainen toteuttaa, mutta myös usein tehoton. Valojen vaikutus pikseliin saatetaan laskea monesti turhaan, sillä vain etummaisin pikseli jää näkyviin. Lisäksi huonoimmassa tapauksessa kappaleisiin vaikuttavien valojen vaikutus lasketaan kerran jokaista valoa kohden.

Paljon valoja sisältävään maailmaan onkin hyvä käyttää erilaista piirtopolkua. Yksi yleisesti käytetty polku on viivästetty piirto (engl. deferred rendering). Viivästetyssä piirrossa virtuaalimaailman kappaleet viedään piirtolinjaston läpi kahteen kertaan. Ensimmäisellä kerralla kärkipisteiden ominaisuudet syötetään erilliseen kuvapuskuriin. Tätä kuvapuskuria kutsutaan usein G-puskuriksi (engl. g-buffer), jossa G-kirjain tulee sanasta geometria. Tässä vaiheessa oletuskuvapuskuriin ei vielä piirretä mitään.

G-puskuriin viedään niitä kärkipisteiden ominaisuuksia, joita tarvitaan myöhemmin valaistuksen laskemisessa. Jokaista ominaisuutta kohden puskuriin luodaan erillinen tekstuuri. Puskuriin voidaan viedä esimerkiksi piirrettävän maailman värit, pintojen normaalit ja paikat. Tämä on havainnollistettu kuvassa 20.



Kuva 20. G-puskurin kolme tekstuuria.

Toisella piirtolinjaston läpikäynnillä ominaisuustekstuureista muodostetaan viimeinen oletuskuvapuskuriin piirrettävä kuva (ks. kuva 21). [4, s. 548.]

Kuva 21. Viivästetyn piirron lopullinen kuva.

Grafiikkamoottorin toteutus aloitettiin edellä mainitulla yksinkertaisemmalla piirtopolulla. Näin voitiin keskittyä vain phong-valaistusmallin toteuttamiseen. Myöhemmin rinnalle tuotiin viivästetty piirto. Molemmille poluille luotiin omat piirtojärjestelmänsä.

5.3 Grafiikkamoottorin käyttö

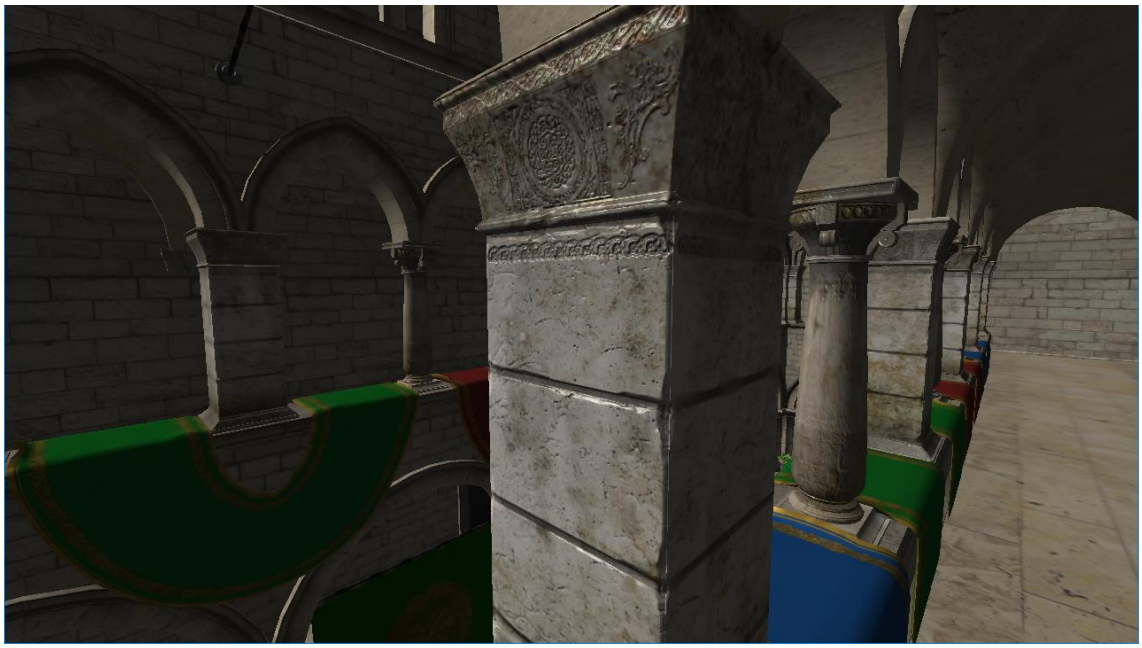
Kuvassa 22 luodaan grafiikkamoottorilla piirrettävä pelimaailma. Maailmaan luodaan kamera entiteetti, jolle määritetään ohjauskomponentti. Ohjaimen avulla kameralla voidaan lentää pelimaailmassa.

Kaikki luodut entiteetit tallennetaan entityManager-instanssiin, jolloin piirtojärjestelmä voi käyttää niihin tallennettua tietoa piirtämiseen. Kolmioverkkojen lataamiseen käytetään resourceManager-oliota. Lopuksi moottori käynnistetään run-funktiokutsulla.

```
Engine engine;
EntityManager &entityManager = EntityManager::instance();
ResourceManager &resourceManager = ResourceManager::instance();
GLFWWindow &window = GLFWWindow::instance();

Entity *camera = new Entity();
camera->addComponent(new Camera(glm::radians<float>(70.f),
    window.getAspect(), 1.0f, 1000.f));
camera->addComponent(new Transform());
camera->addComponent(new Controller());
entityManager.addCamera(camera);

Entity *sponza = new Entity();
sponza->addComponent(resourceManager.
    loadStaticMesh("res/models/sponza/sponza.obj"));
sponza->addComponent(new Transform(glm::vec3(0.0f, 0.0f, 0.0f),
    glm::vec3(0.2f, 0.2f, 0.2f)));
entityManager.addEntity(sponza);

Entity *skybox = new Entity();
skybox->addComponent(resourceManager.
    loadStaticMesh("res/models/cubemap.obj"));
entityManager.addSkybox(skybox);

Entity *directionalLightEntity = new Entity();
DirectionalLight *directionalLight = new DirectionalLight();
directionalLight->color = glm::vec3(1.f, 1.f, 1.f);
directionalLight->direction = glm::vec3(1.f, 1.f, 1.f);
directionalLight->diffuseIntensity = 0.8f;
directionalLightEntity->addComponent(directionalLight);
entityManager.addLight(directionalLightEntity);

engine.run();
```

Kuva 22. Yksinkertaisen maailman luonti moottorin avulla.

Piirron lopputulos on nähtävissä kuvasta 23. Piirretty 3D-malli on yleisesti tietokoneella tuotetun grafiikan testaamisessa käytetty Sponza-palatsi, joka koostuu monesta erillisestä osasta. Kuvasta voidaan nähdä Phong-valaistusmallin varjostukset sekä pylvään pinnassa olevan normaalitekstuurin vaikutus.



Kuva 23. Toteutetulla grafiikkamoottorilla piirretty kuva.

6 Yhteenveto

Insinöörityö alkoi eri kolmiulotteisen grafiikan piirtotapojen tutkimisella. Vaikka erilaisilla säteenseurantatekniikoilla voidaan piirtää hyvin realistista kuvaa, ne eivät vielä hitautensa vuoksi täytä reaaliaikaisuuden määritelmää. Kolmioiden rasterointi sen sijaan on nopea tapa piirtää kolmiulotteista grafiikkaa, mutta piirrettävien kappaleiden pintojen yksityiskohtia varten joudutaan käyttämään lisätekniikoita.

Nykyaikana grafiikan piirto tapahtuu lähes poikkeuksetta näytönohjaimella. Näytönohjaimen käskytykseen on olemassa eri rajapintoja. Toteutetun grafiikkamoottorin grafiikkarajapinnaksi valikoitui OpenGL sen laajan laitteisto- ja käyttöjärjestelmätuen takia.

Insinöörityön lopputuloksena syntynyt grafiikkamoottori kykeni piirtämään erilaisia varsin monimutkaisia maailmoja erilaisten valojen kanssa. Valojen lisäksi grafiikkamoottori tuki

myös muita pintojen pikselitason yksityiskohtien väritystekniikoita kuten tekstuuri- ja normaalikuvausta. Eri valonlähteiden kasvaessa myös kuvan piirtäminen vie kauemmin aikaa. Tätä varten moottoriin toteutettiin viivästetyn piirron piirtopolku, jolla valaistuksen laskemista voidaan nopeuttaa.

Työn tavoitteet realistisen piirron osalta saavutettiin. Myös grafiikkamoottorin arkkitehtuuri oli monilta osin varsin onnistunut. OpenGL-rajapinnan ja muiden yksittäisten käsitteiden paketoiminen omiin luokkiinsa nopeutti ohjelman kehitystä.

Grafiikkamoottorin alhaisen tason piirtorajapinta oli kuitenkin mielestäni liian riippuvainen ylemmän tason luokista. 3D-mallien piirron ja sitä käyttävän ohjelman väliin pitäisi tehdä erillinen rajapinta, jonka kautta vain piirron kannalta tärkeät osat syötettäisiin piirtokerrokselle.

Grafiikkamoottori käsitteenä on hyvin laaja, joten jatkokehitysmahdollisuuksia arkkitehtuurin parantamisen lisäksi ei ole vaikea löytää. Yksi jatkokehitysalue on piirretyn kuvan realistisuuden lisääminen. Erilaisia rasteroinnin realistisuutta lisääviä tekniikoita löytyy valtavasti. Tällä hetkellä ehkä selkein grafiikkamoottorista puuttuva tekniikka on varjokuvaus. Varjot kuuluvat olennaisena osana valon mallinnukseen ja luovat piirrettyyn kuvaan syvyyden tuntua.

Suorituskyvyn parantamiseksi grafiikkamoottoriin olisi hyvä luoda näkyvyyden hallintakerros, jossa kuva-alueen ulkopuolelle jäävät osat karsittaisiin lopullisesta piirrosta. Lisäksi piirrettävien kappaleiden piirtojärjestystä tulisi muuttaa. Samaa materiaalia käyttävät kappaleet pitäisi järjestää piirrossa vierekkäin, jolloin OpenGL-tilamuutosten määrä pienenisi.

Lähteet

1 Gregory, Jason. 2014. Game Engine Architecture, Second Edition. Boca Raton: CRC Press.

2 Sherrod, Allen. 2008. Game Graphics Programming. Boston: Course Technology PTR.

3 Lengyel, Eric. 2012. Mathematics for 3D Game Programming and Graphics, Third Edition. Boston: Course Technology PTR.

4 Sellers, Graham. S. Wright Jr, Richard. Haemel Nicholas. 2013. OpenGL SuperBible, Sixth Edition. Indiana: Addison-Wesley.

5 Mobeen Movania, Muhammed. 2013. OpenGL Development Cookbook. Birmingham, UK: Packt Publishing Ltd.

6 Wolff, David. 2013. OpenGL 4 Shading Language Cookbook, Second Edition. Birmingham, UK: Packt Publisihing Ltd.

7 OpenGL Object. Verkkodokumentti. <https://www.khronos.org/opengl/wiki/OpenGL_Object>. Luettu 12.2.2018.

8 GLEW. Verkkodokumentti. <http://glew.sourceforge.net>. Luettu 2.9.2017.

9 GLFW. Verkkodokumentti. <www.glfw.org>. Luettu 2.9.2017.

10 GLM. Verkkodokumentti. <https://glm.g-truc.net>. Luettu 2.9.2017.

11 Assimp. Verkkodokumentti. <www.assimp.org>. Luettu 13.4.2018.

12 Entity-component-system. Verkkodokumentti. <https://en.wikipedia.org/wiki/Entity-component-system>. Luettu 9.3.2017.

13 Global Illumination. Verkkodokumentti. <https://docs.unity3d.com/Manual/GIIntro.html>. Luettu 31.2.2017.

14 Path tracing. Kuva. <https://upload.wikimedia.org/wikipedia/commons/e/e0/Path_tracing_001.png>. Katsottu 4.1.2018.