Jarno Ruuskanen

Renderöintimoottorin toteutus

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tietotekniikka

Insinöörityö

9.5.2016

Metropolia

Tekijä(t) Otsikko Sivumäärä Aika	Jarno Ruuskanen Renderöintimoottorin toteutus 34 sivua + 2 liitettä 9.5.2016
Tutkinto	Insinööri (AMK)
Koulutusohjelma	Tietotekniikka
Suuntautumisvaihtoehto	Ohjelmistotekniikka
Ohjaaja(t)	Lehtori Miikka Mäki-Uuro

Tämä insinöörityö käsittelee pelimoottorin pohjana toimivan renderöintimoottorin luontia. Insinöörityössä käydään läpi C++-ohjelmointikielellä luodun renderöintimoottorin toteutusta sekä perehdytään OpenGL-rajapinnan toimintaan. Tekstin luvut on tarvittaessa jaoteltu teoriaosuuksiin sekä toteutusosuuksiin.

Insinöörityössä keskitytään aluksi OpenGL:n peruskäsitteisiin, puhutaan SDL2:n käyttämisestä sovelluksen ikkunointikirjastona sekä tarkastellaan suhteellisen laajasti varjostimien toimintaa ja niiden vaikutusta ohjelman suoritukseen. Lisäksi käsitellään matriisilaskennan ja matriisien merkitystä sovelluksessa, tutkitaan 3D-mallien luontia OpenGL:llä sekä käydään läpi muutamia valaisumalleja. Lopuksi kerrataan ohjelman kirjoittamisen aikana ilmenneitä ongelmia ja puhutaan jatkokehityksestä.

Työn tuloksena syntyi renderöintimoottorin prototyyppi, joka osaa ladata käyttäjän sille syöttämiä fbx-tiedostoja ruudulle näkyviin tietyin rajoituksin. Sovellus osaa ladata tiedostoista niiden kulmapistetiedot, sijainti- ja skaalatiedot, tekstuurit, tekstuurien koordinaatit, materiaalitiedot sekä normaalit. Käyttäjä voi ladata useampia tiedostoja ja kokonaisia skenaarioita yhtäaikaisesti ruudulle. Sovelluksesta löytyy myös muutamia valaisutapoja sekä kamera, jolla skenaariota voi tarkkailla eri kulmista. Tarvittaessa käyttäjä voi myös kirjoittaa ohjelmaan omia varjostimia.

Avainsanat	Renderöintimoottori, Moderni OpenGL, C++, FBX SDK

Author(s) Title Number of Pages Date	Jarno Ruuskanen Implementation of Rendering Engine 34 pages + 2 appendices 9 May 2016
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Software Engineering
Instructor(s)	Miikka Mäki-Uuro, Senior Lecturer

This engineering thesis focuses on the creation of a rendering engine, which will later work as a base for a game engine. The thesis goes through the structure of the engine, which was written in C++, and also takes a look on the concept of OpenGL interface.

First the thesis focuses on the very basics of OpenGL, talks about SDL2 windowing library, takes a wide look on shaders, how they work and how they affect the application and covers the basic matrix functionality. After that the thesis delves into the creation of 3D objects in OpenGL, discusses how a few different lighting models work, talks about the problems which occurred when writing the software and lastly takes a look on the future of the software.

The end product is a rendering engine prototype, which fulfills most of the original goals it was given. The engine can read the following information from fbx-files and show them on screen: vertex information, textures, texture coordinates, material information, size, location, rotation and normal information. The software can load multiple files and whole scenarios simultaneously on screen. The application also has two lighting models and a camera, which allows the user to view objects in different angles. The user can also write and add their own shaders to the software.

Keywords	Rendering Engine, Modern OpenGL, C++, FBX SDK

Sisällys

Liitteet

Liite 1. Esimerkki ladatusta skenaariosta

Liite 2. Esimerkki sovelluksen ongelmakohdista

Lyhenteet

OpenGL	Open Graphics Library.
SGI	Silicon Graphics Inc. Yritys, jonka toimesta OpenGL sai alkunsa.
ARB	Architectural Review Board. Ryhmä joka valvoo OpenGL:n kehitystä.
SDL2	Simple DirectMedia Layer. Kirjasto, joka tarjoaa mm. ikkunoinnin OpenGL-materiaalin esitykseen.
GLSL	OpenGL Shading Language. C:n tapainen kieli, jolla ohjelmoidaan varjostimia.
GLM	Matematiikkakirjasto, joka soveltuu hyvin matriisien laskentaan.
VBO	Vertex Buffer Object. OpenGL:n elementti, johon voidaan tallentaa graafisten objektien sisältämää dataa kuten väritietoja, kulmapistetietoja jne.
VAO	Vertex Array Object. OpenGL:n elementti, joka toimittaa datan kulmapiste-varjostimille.
FBX SDK	FBX SDK on Autodesckin luoma kirjasto FBX-tiedostojen lataukseen.

1 Johdanto

Tämän insinöörityön tarkoituksena on luoda sovellus, joka toimii pohjana tulevalle pelimoottorille. Alkuvaiheessa kyseessä on renderöintimoottori, jonka avulla käyttäjä voi ladata fbx-formaattiin tallennettuja 2D- ja 3D-malleja, jotka on luotu kolmannen osapuolen muokkausohjelmalla. Sovellus on kirjoitettu C++-kielellä, ja se käyttää avukseen modernia OpenGL-rajapintaa (3.0 ja uudempaa).

Insinöörityön jokainen luku on jaettu siten, että kustakin aiheesta käsitellään ensin mahdolliset peruskäsitteet, jonka jälkeen näytetään, miten aihealue liittyy sovellukseen ja sen toimintaan. Tekstissä käydään läpi OpenGL:n historiaa ja toiminnallisuutta, varjostimien toimintaperiaatteita ja niiden käyttöä sovelluksessa, 3D-mallien luontia OpenGL-ympäristössä ja valaistuksen perusteita. Paikoin puhutaan myös apuna käytetyistä kolmannen osapuolen kirjastoista. Lopuksi puhutaan suurimmista ongelmista, joita sovelluksen luonnin aikana ilmeni.

Työn tuloksena syntynyt renderöintimoottorin prototyyppi osaa esittää samanaikaisesti kokonaisia skenaarioita ruudulla, jotka on joko kasattu yhdestä tai useammasta tiedostosta. Jokaisesta objektista pystytään lukemaan sen sijainti, koko, kulmapistetiedot, materiaalitiedot, sisältääkö objekti tekstuureja, tekstuurien koordinaatit sekä normaalit. Sovellus sisältää myös erilaisia valonlähteitä.

Insinöörityön pohjalta lukijalle selkeytyy modernin OpenGL:n toimintaperiaate sekä myös osa renderöintimoottorin luontiin vaadittavista työvaiheista. Lukuisat teoriaosuudet myös vahvistavat lukijan tietämystä OpenGL:stä ja siihen liittyvistä konsepteista.

2 OpenGL

OpenGL (Open Graphics Library) on laitteistoriippumaton rajapinta, joka tuo tietokoneen graafisen käyttöliittymän ohjelmoijan käytettäväksi mahdollistaen tietokonegrafiikan luomisen. Ensimmäinen versio OpenGL:stä (OpenGL 1.0) julkaistiin kesäkuussa 1992. Julkaisija, Silicon Graphics Inc. (SGI) oli aloittanut OpenGL:n kehityksen vuonna 1990 huomattuaan laitteistoriippumattoman grafiikkakirjaston mukanaan tuomat hyödyt.

Ennen OpenGL:n julkaisua oli hyvin yleistä, että varsinkin kalliita graafisia tuotteita valmistavat yritykset loivat omia grafiikkakirjastojaan tuotteidensa tueksi. Halvempia tuotteita luovat kilpailijat käyttivät puolestaan grafiikkarajapintoja, jotka sopivat usein yhteen muiden kilpailijoiden tuotteiden kanssa. SGI, joka tuotti markkinoille pääasiassa kalliita graafisia työpisteitä, huomasi laitteistoriippumattoman grafiikkarajapinnan mukanaan tuomat edut ja päätti ryhtyä toimiin. SGI muokkasi luomaansa IRIS GL -grafiikkakirjastoa poistaen siitä laitteistoriippuvaiset osuudet, jonka jälkeen kirjasto laitettiin vapaaseen levitykseen [1.]

OpenGL:n yhtenä suurimmista vahvuuksista pidetään sen laitteistoriippumattomuutta, mikä tarkoittaa sitä, että kerran kirjoitettu ohjelma toimii lähes millä tahansa alustalla. Näin ohjelmistojen kehittäjille jää enemmän aikaa keskittyä itse ohjelman luontiin sen sijaan, että pitäisi keskittyä laitteistojen välisiin eroavaisuuksiin ja tästä syntyvien ongelmien ratkomiseen.

Nykyään OpenGL:ää kehittää monien yritysten jäsenistä koostuva ryhmä nimeltään OpenGL Architectural Review Board (ARB), joka on osa Khronos Grouppia. Ryhmän tarkoituksena on valvoa ja jatkaa OpenGL:n kehitystä. Ajan kuluessa ja tekniikan kehittyessä myös OpenGL on kehittynyt melkoisesti, mutta tämä on tuonut mukanaan isoja ongelmia liittyen vanhempien versioiden tukemiseen. Aiemmat käytännöt (kuten jokaisen piirtokäskyn lähetys erikseen CPU:lta näytönohjaimelle) ovat voineet vanheta ja asettavat uusien OpenGL-versioiden myötä suuria rajoitteita järjestelmän suorituskyvylle. Yhä kasvavasta ongelmasta tuli lopulta niin suuri, että vuonna 2008 ARB päätti jakaa OpenGL:n kehityksen kahteen eri haaraan, moderniin ja vanhaan. Vanha haara tukee edelleen kaikkia OpenGL-versioita aina 1.0 asti, mutta on huomattavasti pidempi. Moderni OpenGL tukee julkaisuja versiosta 3.0 eteenpäin. Tätä insinöörityötä kirjoitettaessa (kevät 2016) uusin versio OpenGL:stä on 4.5, joka julkaistiin elokuussa 2014.

3 Grafiikan luominen OpenGL:ssä

Maksimaalisen suoritustehon saavuttamiseksi työ kannattaa usein jakaa pienempiin osioihin, jotka voidaan suorittaa yksitellen ja tarvittaessa samanaikaisesti. Nykypäivän näytönohjaimista löytyy muutamasta kymmenestä tuhansiin pientä prosessoria, varjostinydintä (shader core), jotka ovat ohjelmoitavissa. Nämä ytimet suorittavat varjostimiksi (shader) kutsuttuja pieniä ohjelmia, joilla voidaan vaikuttaa grafiikan esitystapaan. Vaikka yksittäinen ydin onkin melko vaatimaton suorituskyvyiltään, pystyvät tuhannet ytimet yhteistyössä toteuttamaan erittäin suuria työmääriä ja saamaan aikaan hienoja lopputuloksia.

Modernissa OpenGL:ssä tämä grafiikkaliukuhihna (kuva 1) on erittäin keskeisessä osassa ohjelmaa ja sovelluksen tekijän tulee itse huolehtia siitä, että tieto kulkee grafiikkaliukuhihnan läpi oikeassa muodossa. Kuvasta 1 näkyy yksinkertaistettu versio liukuhihnasta, joka koostuu seuraavista varjostimista: kulmapistevarjostin (vertex shader), tessellaation kontrollointivarjostin (tessellation control shader), tessellaation evaluointivarjostin (tessellation evaluation shader), geometriavarjostin (geometry shader) sekä fragmenttivarjostin (fragment shader). Kuvan kulmikkaat laatikot ovat käyttäjän ohjelmoitavissa eikä pyöreäkulmaisten laatikoiden toimintaan ohjelmoijan tarvitse puuttua, koska ne ovat automatisoituja.

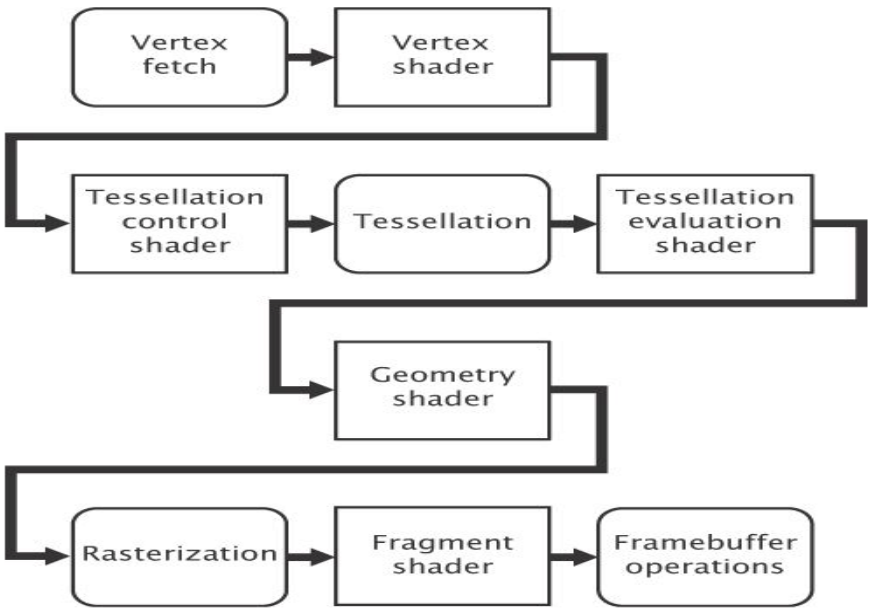


Figure 1.1: Simplified graphics pipeline

Kuva 1. Yksinkertaistettu kuva ohjelmoitavasta grafiikkaputkesta [2].

Kukin varjostinosio (shader stage) keskittyy tekemään läpi kulkevalle datalle tietyntyyppisiä muunnoksia, kuten sijainti-, väri- tai muodonmuutoksia. Varjostimien toimintaan ja niiden ohjelmointiin palataan tarkemmin luvussa 5.

4 SDL2-kirjasto

Koska OpenGL ei sisällä ikkunointimekanismia, tehtävää varten valittiin SDL2-kirjasto, jonka avulla pääsee tarvittaessa helposti käsiksi myös äänen, näppäimistön ja hiiren toimintoihin. SDL2 soveltuu erityisen hyvin sovellukseen, jota tämä insinöörityö käsittelee, koska tarkoituksena on tehdä laitteistoriippumaton ohjelmisto. Kyseinen kirjasto on tuettuna Windowsilla, Linuxilla, Applen OS:llä, Androidilla sekä iOS:lla.

Ennen SDL2:n käyttöä on varmistettava, että kirjasto on käytössä. Tämä suoritetaan kutsumalla SDL_Init()–funktiota, jolle annetaan parametriksi, mitä halutaan alustaa. Jos ei ole aivan varma, mitä aikoo SDL:llä tehdä, kannattaa käyttää parametria SDL_INIT_EVERYTHING, jolla saa kaikki kirjaston ominaisuudet käyttöönsä. Alustuksen jälkeen on myös mahdollista määrittää muutamien OpenGL-attribuuttien kokoja SDL_GL_SetAttribute()–funktiolla. Tämä tulee tehdä ennen ikkunan luomista. Ohjelma toimii myös ilman kyseisten väriattribuuttien asettamista, mutta manuaalisesti kirjoittamalla voidaan tarvittaessa antaa kyseisille kanaville enemmän tilaa (katso kuva 2).

```
119     if (SDL_Init(SDL_INIT_EVERYTHING) != 0)
120     {
121         logSDLError(std::cout, "SDL_Init");
122     }
123
124     //How many bits are used for each color;
125     SDL_GL_SetAttribute(SDL_GL_RED_SIZE,8);
126     SDL_GL_SetAttribute(SDL_GL_GREEN_SIZE, 8);
127     SDL_GL_SetAttribute(SDL_GL_BLUE_SIZE, 8);
128     SDL_GL_SetAttribute(SDL_GL_ALPHA_SIZE, 8);
129     //32 bits of color data.
130     SDL_GL_SetAttribute(SDL_GL_BUFFER_SIZE, 32);
131     //Double buffering
132     SDL_GL_SetAttribute(SDL_GL_DOUBLEBUFFER, 1);
```

Kuva 2. Kirjaston alustus sekä attribuuttien lisäys.

SDL2:ssa ikkunan luonti on tehty hyvin yksinkertaiseksi. Tarvitaan SDL_Window–muuttuja, joka luodaan SDL_CreateWindow()-funktiolla. Parametreiksi annetaan haluttu nimi, ikkunan sijainti ruudulla, leveys, korkeus sekä sisällön tyyppi (kuva 3).

```
134     const std::string& _title = "Model loader";
135     _window = SDL_CreateWindow(_title.c_str(),
136                                SDL_WINDOWPOS_CENTERED,
137                                SDL_WINDOWPOS_CENTERED,
138                                _screenWidth,
139                                _screenHeight,
140                                SDL_WINDOW_OPENGL);
141
142     if (_window == nullptr)
143     {
144         logSDLError(std::cout, "CreateWindow");
145         SDL_Quit();
146     }
```

Kuva 3. Ikkunan luonti sekä virhetarkistus.

Jotta OpenGL-sisältö saadaan näkymään ruudussa oikein, on se laitettava oikeaan kontekstiin. SDL_GLContext-muuttuja alustetaan SDL_GL_CreateContext() -nimisellä funktiolla, jolle annetaan parametriksi aikaisemmin luotu ikkuna. Sovelluksessa käytetään myös hyödyksi GLEW-nimistä kirjastoa, joka otetaan käyttöön glewInit()-funktiolla. Tämän lisäksi käytetään myös GLEW-kirjaston globaalia glewExperimental-muuttujaa, joka mahdollistaa testivaiheessa olevien ajurien käytön. Kyseinen muuttuja tulee asettaa GL_TRUE arvoon ennen GLEW-kirjaston alustusta (kuva 4) [3.]

```
_glContext = SDL_GL_CreateContext(_window);
if (_glContext == nullptr)
{
    logSDLError(std::cout, "SDL_GLContext");
    //cleanup(glContext);
    SDL_Quit();
}

glewExperimental = GL_TRUE;
```

```
GLenum error = glewInit();
if (error != GLEW_OK)
{
    std::cout << "Glew could not be initialized" << std::endl;
}
```

Kuva 4. Kontekstin luonti sekä GLEWin alustus.

5 Varjostimet

Sovelluksen on tarkoitus käyttää hyödykseen mahdollisimman modernia OpenGL:ää, joten sovelluksen tekijän on myös tiedettävä tiettyjä perusasioita varjostimien ohjelmoinnista. Kuten jo aikaisemmin listattiin, grafiikkaputkessa ohjelmoitavia varjostimia on viisi kappaletta: kulmapiste-, tessellaation kontrollointi -, tessellaation evaluointi-, geometria- ja fragmenttivarjostin (putki kulkee tässä järjestyksessä). Ohjelmoijan pitää huolehtia itse vähintään kulmapiste- sekä fragmenttivarjostimien kirjoittamisesta, jos ruudulta haluaa nähdä jotain. Lisäksi on olemassa laskentavarjostin (compute shader), jota käytetään muusta grafiikkaputkesta erillään suorittamaan laskennallisia toimenpiteitä. Laskentavarjostinta ei siis voi liittää samaan varjostinohjelmaan, jossa on jo liitettynä muun tyyppisiä varjostimia. Sovellus käyttää toistaiseksi pääasiassa aikaisemmin mainittuja viittä varjostinta ja jättää laskentavarjostimien käytön vähemmälle huomiolle.

Pelkkä varjostimien kirjoittaminen ei kuitenkaan vielä riitä ohjelman käynnistymiseen, vaan kyseiset varjostimet pitää myös koota ja linkittää OpenGL-sovellukseen, jotta ne toimivat oikein. Ohjelmaa varten luotiin funktion, jolle syötetään vector-muuttujassa lista varjostimien osoitteista, ja ohjelma kokoaa sekä linkittää kyseiset varjostimet automaattisesti osaksi ohjelmistoa. Tämän ansiosta sovelluksen käyttäjä pystyy helposti kirjoittamaan eri varjostimia ja päättämään, mitä niistä halutaan käyttää ohjelmassa (kuva 5).

```
//First let's choose which shaders we want to use.
//Add them to an array and send the array to the glslProgram.
//"Shaders/testShader.vert", "Shaders/testShader.frag","Shaders/testShader.tcs","Shaders/testShader.tes",
std::vector<std::string> _shaderFiles = { "Shaders/basicShader.vert","Shaders/basicShader.frag" };
_shaderProgram.compileShaders(_shaderFiles);
_shaderProgram.linkShaders();
```

Kuva 5. Varjostimien valintatapa sovelluksessa.

Kommenttiriviltä löytyy myös muita sovelluksessa käytettyjä varjostimia, joita voi lisätä vektoriin tarpeen mukaan. GLSLProgram-luokan compileShaders-metodi lukee kunkin tiedostopolun päätteen ja luo oikeanlaisen varjostimen sen avulla. Funktion vastaanottama, string-muuttujia sisältävä vektori iteroidaan läpi yksitellen. Se kokoaa jokaisen valitun varjostimen yksitellen ja tarkistaa samalla mahdolliset virheet varjostimista.

GLSLProgram-luokka käyttää melko paljon hyödyksi GLSL Cook Book -nimisen kirjan esimerkkejä (kuva 6) [4; 5].

GLSLProgram-luokka sisältää myös compileShader-funktion, jota käytetään vektoria iteroitaessa luomaan yksittäiset varjostimet. Parametreiksi funktio tarvitsee tiedoston osoitteen sekä GLuint-muuttujan, johon varjostin halutaan alustaa. Tiedoston sisältämä data luetaan talteen string-muuttujaan, jonka jälkeen muuttuja syötetään **glShaderSource-**nimiselle funktiolle. Funktio vastaanottaa parametreina varjostimen (GLuint-muuttuja), taulukon sisältämien string-muuttujien määrän, referenssin tiedoston sisältämään dataan sekä pituuden. Jos pituus on merkitty tyhjäksi (nullptr), ohjelma olettaa jokaisen stringin päättyvän null-merkinnällä '\0'. glShaderSource korvaa kutsuttaessa varjostimessa jo mahdollisesti olleen datan uudella.

Kun varjostimelle haluttu data on luettu talteen, pitää varjostin vielä koota. Tämä onnistuu **glCompileShader**–nimisellä funktiolla, jolle annetaan parametriksi varjostin, joka halutaan koota. Varjostimen luomisen jälkeen on vielä hyvä tarkistaa, ilmenikö kasauksessa ongelmia. Tämä onnistuu helpoiten **glGetShaderiv**-funktiolla.

```
void GLSLProgram::compileShader(const std::string& shaderFilePath, GLuint shaderID)
{
    //Open the shader files and read the contents
    std::ifstream shaderFile(shaderFilePath);
    if (shaderFile.fail())
        perror(shaderFilePath.c_str());

    std::string content = "";
    std::string line;

    //Read the file content each line at a time.
    while (std::getline(shaderFile, line))
        content += line + "\n";

    shaderFile.close();

    //ShaderID, number of strings, reference to data, string length
    const char* contentsPointer = content.c_str();
    glShaderSource(shaderID, 1, &contentsPointer, nullptr);

    glCompileShader(shaderID);
```

Kuva 6. Yksittäisen varjostimen luominen.

Kun jokainen varjostin on luotu, on ne vielä linkitettävä osaksi sovellusta. Tätä varten ohjelmasta löytyy **linkShaders**-funktio (kuva 7). Ennen varjostimien käyttöönottoa on niitä varten luotava varjostinohjelma. Tämä onnistuu **glCreateProgram**-funktiolla. Ohjelmaa ei ole pakollista luoda vasta varjostimien luonnin jälkeen, mutta tähän ratkaisuun

päädyttiin, koska se ratkaisi muutaman ohjelman suoritusjärjestyksestä syntyneen ongelman.

Seuraavaksi ohjelma tarkistaa, mitkä varjostimet halutaan ottaa käyttöön ja ne liitetään varjostinohjelmaan **glAttachShader**-funktiolla. Funktio ottaa vastaan varjostinohjelman sekä varjostimen, joka siihen halutaan liittää.

```
void GLSLProgram::linkShaders()
{
    //First we create the shader program where we will attach the shaders.
    _programID = glCreateProgram();

    //Next we go through the shader handle map and see which shaders have been initialized so we can
    //link them with the program.
    for (auto it : _shaderHandles)
    {
        if (it.second != 0)
        {
            glAttachShader(_programID, it.second);
            std::cout << "Shader attached: " <<it.second << std::endl;
        }
    }
    //This next part is mostly from OpenGL-wiki and has been modified
    //to suit my needs.
    //https://www.opengl.org/wiki/Shader_Compilation#Example

    //Link the program
    glLinkProgram(_programID);
```

Kuva 7. Varjostinohjelman luonti sekä varjostimien liittäminen osaksi ohjelmaa.

Kun kaikki halutut varjostimet on liitetty varjostinohjelmaan, kutsutaan **glLinkProgram**-funktiota, jolle annetaan parametriksi haluttu ohjelma. Funktio linkittää luodun varjostinohjelman osaksi OpenGL-sovellusta. Seuraavaksi tarkistetaan vielä, ilmenikö linkityksessä ongelmia. Kun varjostimet on lisätty osaksi luotua ohjelmaa, voidaan niiden käyttämä tila vapauttaa. Tämä hoituu kutsumalla ensin **glDetachShader**-funktiota, joka ottaa vastaan samat parametrit kuin glAttachShader. Irrottamisen jälkeen voidaan kutsua **glDeleteShader**-funktiota, joka poistaa varjostimen ja vapauttaa muistin (kuva 8). Ohjelman toimivuuden kannalta on hyvin tärkeää, että turhaa muistia vievät muuttujat poistetaan. Tämä ei ehkä näy suoraan suoritustehossa ohjelman ollessa vielä pienessä mittakaavassa, mutta voi kasvaa isoksi ongelmaksi myöhemmin, jos siitä ei huolehdita.

```
//Always detach shaders after a successful link.
//You can also delete the shaders so they can be reused elsewhere.
for (std::map<GLSLShader::GLSLShaderType, GLuint>::iterator ite = _shaderHandles.begin();
ite != _shaderHandles.end(); ++ite)
{
    if (ite->second != 0)
    {
        glDetachShader(_programID, ite->second);
        glDeleteShader(ite->second);
    }
}
```

Kuva 8. Varjostimien käyttämä tila voidaan vapauttaa, kun ne on liitetty osaksi ohjelmaa.

Kun halutut varjostimet on luotu ja liitetty osaksi ohjelmaa, on ne enää otettava käyttöön. Tätä varten OpenGL:ssä on olemassa funktio nimeltään **glUseProgram**, joka ottaa parametrina vastaan halutun varjostinohjelman. Tätä varten GLSLProgram-luokkaan luotiin funktiot **use** sekä **unuse**, joissa yksinkertaisesti kutsutaan glUseProgram-funktiota. Unuse-funktiossa parametriksi annetaan numero 0, jolla poistetaan käytetty varjostinohjelma käytöstä. glUseProgram-funktiota tulee kutsua aina, kun halutaan käyttää varjostimia ohjelmassa, joten tarvittavat funktiokutsut kannattaneekin sijoittaa funktioon, joka hoitaa ruudun päivittämisen [4; 5.]

5.1 Varjostimien kirjoittaminen

Varjostimia käytettäessä prosessi alkaa aina linjaston alkupäästä kulmapistevarjostimesta. Siitä matka jatkuu tesselaation hallintavarjostimelle, tessellaation evaluointivarjostimelle, geometriavarjostimelle ja lopulta fragmenttivarjostimelle. Fragmenttivarjostinta ennen suoritetaan rasterointi, joka rikkoo primitiivit (eli siis pisteet, viivat ja kolmiot) osiin ja laskee, mitkä pikselit ovat näiden primitiivien peitossa. Linjastoa edetessä jokaista varjostinta kutsutaan ”varjostintasoksi” ja jokaisella on oma erikoistehtävänsä liukuhihnalla. Varjostimille saapuva data merkitään yleisimmin **in**-avainsanalla ja seuraavalle varjostintasolle toimitettava data merkitään **out**-avainsanalla. Nimien ja datan tyypin on oltava yhtenäistä tasolta toiselle siirryttäessä. Jos käytössä on esimerkiksi kulmapistevarjostin sekä fragmenttivarjostin ja väritiedot on toimitettu kulmapistevarjostimelle, saadaan väritiedot toimitettua fragmenttivarjostimelle esittelemällä lähtevä muuttuja **out vec4 colour** ja asettamalla saapunut tieto siihen. Fragmenttivarjostimessa kyseinen data otetaan vastaan samalla nimellä, mutta käyttäen in-avainsanaa (**in vec4 colour**).

On myös olemassa **uniform-**muuttujia, joita voidaan lähettää mille tahansa varjostintasolle (normaalisti data toimitetaan aina ensin kulmapistevarjostimelle). Tällaisia muuttujia käytetään usein esimerkiksi matriisien tai tekstuuridatan kuljettamiseen.

5.1.1 Kulmapistevarjostin

Kulmapistevarjostin on linjaston ensimmäinen varjostin, ja siinä käsitellään nimensä mukaisesti graafisten objektien kulmapisteitä. Kulmapistevarjostimelle toimitetaan kulmapistedataa **vec**-tyyppisissä muuttujissa, jotka koostuvat x-, y-, z- ja usein myös w-koordinaateista (tässä tapauksessa muuttuja olisi tyyppiä **vec4**). Kulmapistevarjostimille voi tarvittaessa toimittaa myös mitä tahansa muuta dataa, joka toimitetaan eteenpäin joko muokkaamatta tai muokattuna.

Ennen kulmapistevarjostinta suoritetaan kulmapisteiden haku (vertex fetch) ja data toimitetaan edelleen kulmapistevarjostimelle. Kulmapistevarjostimessa avainsanalla **in** merkityt muuttujat ottavat tämän datan vastaan. Jotta oikea data menee oikeaan osoitteeseen, on jokaiselle muuttujalle määriteltävä sen paikka **layout**-avainsanalla ja määrittämällä tämän jälkeen attribuutin sijainti (esimerkiksi **layout (location=0) in vec4 position**). Saapuvan datan ohjaamista kullekin muuttujalle tarkastellaan tarkemmin kappaleessa 7 (katso myös kuva 16) puhuessa vertex array -objekteista.

Kulmapistevarjostimessa on muutamia ennalta nimettyjä, ulos ohjattuja muuttujia (nämä muuttujat on siis merkitty avainsanalla **out**). Yksi näistä on **gl_Position**, johon kulmapistedata yhdistetään. Toinen melko usein käytetty muuttuja on nimeltään **gl_PointSize**, jolla määritellään kunkin kulmapisteen leveys/korkeus pikseleinä.

Kulmapistevarjostimessa on myös ennalta määriteltyjä sisääntulomuuttujia(**in**), kuten **gl_VertexID** sekä **gl_InstanceID**. Muuttuja gl_VertexID kuvastaa kulloinkin kyseessä olevan kulmapisteen indeksiä ja gl_InstanceID:ta käytetään puolestaan silloin, kun halutaan suorittaa jonkinsorttista instanssipiirtämistä (eli piirretään esimerkiksi samasta objektista useita versioita eri sijainteihin) [6; 7.] Kulmapistevarjostimen jälkeen vuorossa on tesselointi.

5.1.2 Tesselointi

Linjaston kolme seuraavaa varjostintasoa käsittelevät kaikki tesselointia: tesseloinnin kontrollointivarjostin, tesselointimoottori sekä tesseloinnin evaluointivarjostin. Tesselointi tarkoittaa prosessia, jossa useammista kulmapisteistä koostuva pinta (patch) jaetaan pienempiin osiin, primitiiveihin (pisteet, viivat, kolmiot). Kuva 9 näyttää tilanteen kahdesta kolmiosta koostuvasta neliöstä (kuvassa vasemmalla), joka on jaettu pienempiin osiin tesseloinnin avulla (kuvassa oikealla).

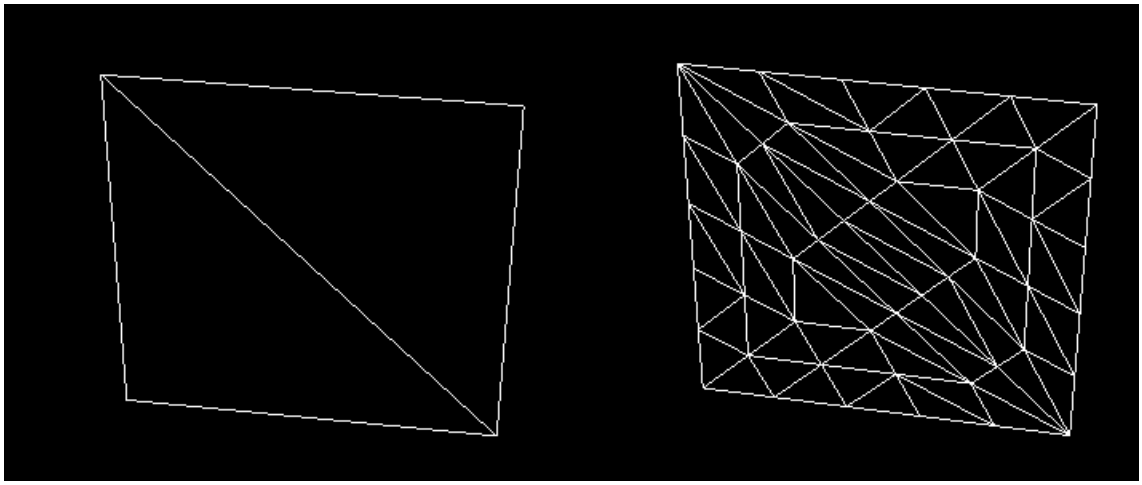
Edellä mainituista kolmesta varjostintasosta kaksi ovat käyttäjän ohjelmoitavissa, tesselointimoottori toimii itsenäisesti. Tesselointivarjostimien käyttö ei ole OpenGL:ssä pakollista, mutta tesseloinnin käytöstä voi olla erittäin paljon hyötyä. Tesselointi mahdollistaa pintojen jakamisen pienempiin osiin, joka puolestaan mahdollistaa yksityiskohtaisempien grafiikkaobjektien luomisen sekä erilaisten kartoitusten lisäämisen (bump mapping, displacement mapping) [8.]

Ensimmäisenä on vuorossa tesselaation kontrollointivarjostin, jonka tehtävänä on määritellä tesseloinnin taso sekä tarvittaessa muokata saapuvaa dataa oikeaan muotoon, jotta se voidaan syöttää tesselointimoottorille. Kontrollointivarjostin ei ole pakollinen, vaikka tesselointia haluaisikin käyttää. Mikäli kontrollointivarjostinta ei löydy, syötetään data suoraan tesselointimoottorille ja tesseloinnin taso voidaan määrittää kontrollointivarjostimen sijasta suoraan sovelluksesta funktiokutsulla **glPatchParameter***.

Tesselointimoottori jakaa sille syötetyt isommat pinnat pienempiin osiin, joita kutsutaan siis primitiiveiksi. Tesseloinnin lopputulokseen vaikuttaa kontrollointivarjostimen lisäksi myöskin evaluointivarjostimessa määritettävä: **välistys** (equal_spacing, fractional_even_spacing, fractional_odd_spacing), **primitiivin tyyppi**, johon pinnat halutaan jakaa (triangles, quads, isolines) sekä primitiivien haluttu **kiertojärjestys** (**cw**/**ccw** eli myötä tai vastapäivään).

Edellä mainittujen määritelmien lisäksi evaluointivarjostimen tehtävänä on kasata tesselointimoottorin luoma data siihen muotoon, että se voidaan toimittaa eteenpäin. Evaluointivarjostin suoritetaan jokaiselle saapuvalle kulmapisteelle, joten tesseloinnin käyttö voi rasittaa konetta hyvin paljon. Tästä syystä on aina hyvä pohtia, tarvitaanko kulloisessakin objektissa tesselointia. Toisin kuin kontrollointivarjostin, evaluointivarjostimen käyttö on pakollista, mikäli tesselointia halutaan käyttää ohjelmassa [9; 10.]

On hyvä huomata muutamia ohjelman toimintaan vaikuttavia asioita, joita tesseloinnin käyttö aiheuttaa. Kun sovelluksessa halutaan käyttää tesselointivarjostimia, tulee piirtofunktiossa käyttää GLenumia GL_PATCHES (katso kuva 18). Lisäksi kun esimerkiksi väridataa halutaan kuljettaa fragmenttivarjostimelle tesselointi- tai geometriavarjostimen kautta, on kyseisen attribuutin oltava taulukkomuodossa (**in vec4 colour [ ]**). Tämä johtuu siitä, että kyseiset varjostimet käsittelevät primitiivejä sekä patcheja.



Kuva 9. 2D-pinta ilman tesselaatiota ja tesseloituna.

5.1.3 Geometriavarjostin

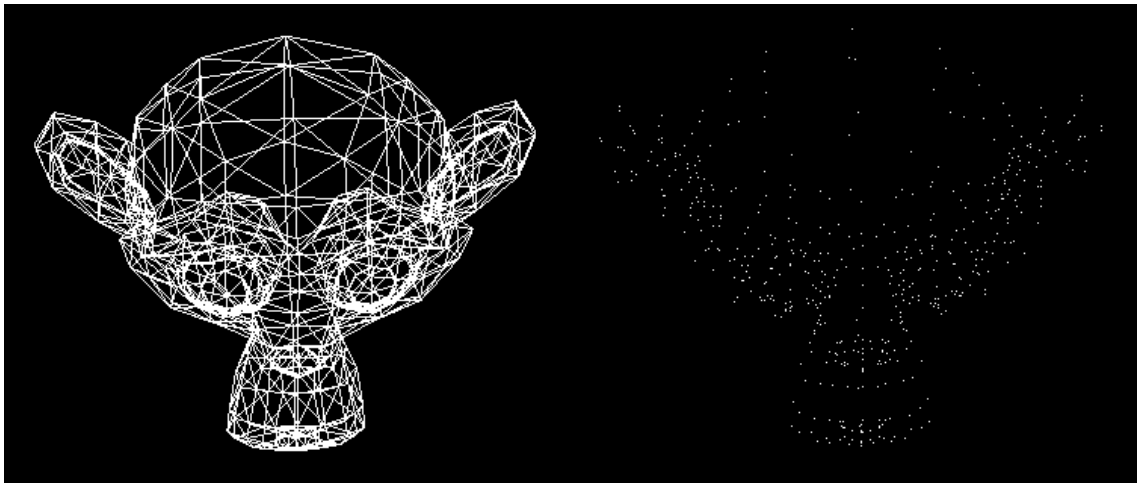
Geometriavarjostin on varjostimista ainoa, joka pystyy muuttamaan saapuvan ja lähtevän datan muotoa esimerkiksi kolmioista pisteiksi. Varjostimen on kuitenkin tiedettävä saapuvan ja lähtevän primitiivin tyyppi (eli otetaanko vastaan kolmioita, pisteitä jne). Jos esimerkiksi vastaanotetaan kolmioita ja lähetetään eteenpäin pisteitä, voidaan se ilmaista määritteillä **layout (triangles) in** sekä **layout (points,max_vertices=3) out**. Ulospäin lähtevässä primitiivissä määritellään myös, kuinka monta kulmapistettä varjostin voi korkeintaan piirtää aina kun sitä kutsutaan. Siinä missä esimerkiksi kulmapistevarjostin käsittelee yhden kulmapisteen kerrallaan, geometriavarjostin käsittelee yhden primitiivin kerrallaan, joka voi sisältää useita kulmapisteitä. Geometriavarjostin pääsee käsiksi kaikkiin näihin primitiivin kulmapistetietoihin ja voi tarvittaessa muokata, poistaa tai luoda primitiivejä.

Geometriavarjostimen avulla voidaan myöskin toteuttaa kerroksittaista piirtämistä (layered rendering). Tämä tarkoittaa sitä, että yhtä primitiiviä voidaan käyttää hyväksi

useammassa kuvassa sen sijaan, että jokaiselle kuvalle pitäisi luoda/sitoa oma primitiivinsä. [11; 12]

Geometriavarjostin on tesselointivarjostimien tavoin vapaavalintainen varjostin, jota ei ole pakko käyttää. Kuva 10 näyttää tilanteen, jossa geometriavarjostin ottaa vastaan kolmioita ja lähettää eteenpäin näiden kolmioiden kulmapisteet, jolloin saadaan kuvan oikeassa laidassa näkyvä lopputulos. Pisteiden kokoa voi tarvittaessa muokata esimerkiksi suoraan sovelluksesta kutsulla **glPointSize(float size)** tai varjostimesta määrittelemällä koko muuttujalle **gl_PointSize**.

Kuten muillakin varjostimilla, myös geometriavarjostimella on joitain valmiiksi varattuja muuttujia. Edellä mainittu gl_PointSize kuuluu geometriavarjostimelle saapuvaan **gl_PerVertex**-blokkiin, joka sisältää myös muuttujat **vec4 gl_Position** sekä **float gl_ClipDistance[ ].** Näiden muuttujien tiedot on pääasiassa täytetty jo linjaston aikaisemmissa vaiheissa, mutta geometriavarjostimella on olemassa myös muutama oma, primitiiveihin perustuva muuttujansa. Toinen näistä on **int gl_PrimitiveIDIn**, joka kuvastaa kulloisenkin primitiivin numerotunnistetta (tämä lasketaan tähän asti piirrettyjen primitiivien avulla). Jokaisella instanssilla on myös oma **int**-tyyppinen numerotunnisteensa, joka löytyy muuttujasta **gl_InvocationID**.



Kuva 10. Vasemmalla objekti ilman geometriavarjostinta, oikealla varjostimen kanssa

5.1.4 Rasterointi

Rasterointi on linjaston seuraava osuus ennen fragmenttivarjostinta. Tämä linjaston osa ei ole käyttäjän ohjelmoitavissa. Vaikka sovelluksessa olisikin kolmiulotteinen objekti, näytetään se lopulta kuitenkin kaksiulotteisella pinnalla. Rasteroija kokoaa tiedot piirrettävistä pikseleistä (väri, syvyys jne.) fragmentteihin ja lähettää ne edelleen fragmenttivarjostimelle [13; 14.]

5.1.5 Fragmenttivarjostin

Fragmenttivarjostin on grafiikkalinjaston viimeinen ohjelmoitava varjostin ja käytännössä katsoen myös kulmapistevarjostimen tavoin pakollinen, mikäli ruudulta tahtoo nähdä jotain. Ohjelma lähtee käyntiin ilman fragmenttivarjostintakin, mutta pikseleiltä puuttuu tällöin väri eikä ruudulle mahdollisesti piirrettäviä kuvioita pysty näkemään.

Fragmenttivarjostin vastaanottaa rasteroijan tuottamia fragmentteja, joiden avulla se antaa kullekin pikselille oikean värin ja syvyyden. Fragmenttivarjostimissa toteutetaan usein myös osa valaistukseen liittyvistä laskutoimituksista (esimerkkinä Phong-valaistus), joskaan tämä ei ole pakollista (Gouraud-valaistus, jossa suurin osa valoon liittyvistä laskutoimituksista on kulmapistevarjostimessa) [15.]

Mikäli fragmenttivarjostimelta halutaan lähettää ulos väritietoja (tämä ei ole pakollista), liitetään ne usein vec4-muuttujaan, joka sisältää kunkin värin normalisoidun voimakkuuden (punainen, vihreä, sininen, alpha). Fragmenttivarjostimessa käsitellään myös tekstuureihin liittyvä toiminnallisuus (Sampler-muuttujat). Fragmenttivarjostimen jälkeen on jäljellä kuvapuskuri, joka tuo linjastolla luodun datan käyttäjän nähtäväksi [16.]

5.1.6 Kuvapuskuri

Kuvapuskuri (framebuffer) on linjaston viimeisin vaihe ja käytännössä se osa, joka näkyy SDL2:n luomassa ikkunassa. Kuvapuskuri sisältää tiedon siitä, mitä pitää piirtää ja mihin. Ennen kuin fragmentteja lähetetään kuvapuskurille piirrettäväksi, on niille mahdollista tehdä erilaisia testejä. Yksi näistä testeistä on nimeltään saksitesti (scissor test), jonka avulla ruudulta rajataan alue, jonka ulkopuolelle ei piirretä mitään. Testin saa tarvittaessa päälle funktiokutsulla **glScissor**, jolle annetaan parametreiksi piirrettävän alueen x- ja y-

koordinaatit sekä alueen haluttu leveys ja korkeus. Saksitestillä varmistetaan, ettei halutun alueen (esimerkiksi sovelluksen ikkunan) ulkopuolelle piirretä turhaan mitään [17.]

Sapluunatestaus (stencil test) vertaa jokaisen fragmentin sapluuna-arvoa (tyyppiä **unsigned int**) sapluunapuskurissa sijaitsevaan arvoon ja poistaa fragmentin, mikäli se ei täytä testin vaatimuksia. Sapluunatestaus vaatii aina, että sovelluksessa on käytössä sapluunapuskuri. Testin saa tarvittaessa päälle sovelluksesta funktiokutsulla **glEnable(GL_STENCIL_TEST)** [18.]

Fragmenteille on mahdollista tehdä myös syvyystestaus (depth test), joka vertaa fragmenttien z-koordinaatteja syvyyspuskurin (depth buffer) koordinaatteihin ja määrittelee, mikä pikseli on lähimpänä katsojaa ja mikä on kauimpana (nämä arvot ovat normalisoituja väliltä 0 ja 1, missä 0 on lähimpänä ja 1 kauimpana). Näin määritetään, mitä ruudulla kulloinkin näkyy ja mikä osa objektista on piilossa.

Kun kaikki halutut/tarvittavat testit on käyty läpi, siirretään data kuvapuskurille piirrettäväksi. Grafiikkaa esittävissä sovelluksissa on tapana käyttää tuplapuskurointia (double buffering), jolloin vältytään ruudun välkkymiseltä. Tuplapuskuroinnissa sovelluksella on käytössään kaksi kuvaa, joista toiseen piirretään muutokset ja toinen näytetään katsojalle vuorotellen. Toisen kuvan ollessa piilossa siihen tehdään tarvittavat muutokset, jonka jälkeen se siirretään näytettäväksi. SDL2 hoitaa tämän toiminnallisuuden funktiokutsulla **SDL_GL_SwapWindow**, jolle annetaan parametriksi ohjelman alussa luotu ikkuna. Funktiota tulee kutsua aina ruutua päivitettäessä.

5.1.7 Sovelluksen käyttämät varjostimet

Tällä hetkellä sovelluksesta löytyy useita erilaisia varjostimia, joita käyttäjä voi tarpeen tullen käyttää. Ohjelmasta löytyy valaistukseen keskittyviä varjostimia, yksinkertaisia tesselaatio- sekä geometriavarjostimia, tekstuureja ja materiaaleja käsitteleviä fragmenttivarjostimia sekä erilaisia kulmapistevarjostimia. Sovellukseen on myös helppo kirjoittaa lisää omia varjostimia: sovellus kääntää minkä tahansa varjostimen ja linkittää sen osaksi ohjelmaa.

Sovellus lataa pistetiedot fbx-tiedostoista ja lähettää ne kulmapistevarjostimelle vec4-muuttujassa. Tekstuurikoordinaatit lähetetään vec2-muuttujassa ja mahdolliset matriisit,

joilla objektia liikutellaan, lähetetään mat4-tyypin uniformeissa. Väridata toimitetaan myöskin vec4-muuttujissa (alpha-kanava on oletusarvona 1).

5.2 Matriisilaskenta

Sovelluksessa pitää pystyä tarvittaessa tarkastelemaan sovelluksen ruudulle lataamia objekteja eri kuvakulmista. OpenGL:ssä esineiden ja asioiden liikuttaminen ja kääntäminen hoidetaan matriisilaskennan avulla. OpenGL ei tunne kameraa käsitteenä, mutta sen toimintaa voidaan silti jäljitellä uskottavasti. Siinä missä helposti voisi kuvitella kameran liikkuvan, kun esimerkiksi pelaaja liikkuu pelikentällä, oikeasti maailmaa liikutetaan vastakkaiseen suuntaan, pelaajan kamera pysyy paikallaan. Tämä luo illuusion siitä, että pelaaja liikkuu, vaikka todellisuudessa maailma pelaajan ympärillä liikkuu. Tätä toiminnallisuutta varten sovellukseen kirjoitettiin Camera-niminen luokka, jolla voidaan alustaa kamera sekä tehdä pientä matriisilaskentaa. Laskennassa käytetään hyväksi matematiikkakirjastoa nimeltään **GLM**, joka soveltuu tehtävään erityisen hyvin, koska se käyttää samaa kirjoitusasua kuin OpenGL:n varjostimet.

Kameran alustusta varten tarvitaan haluttu kameran sijainti kolmiulotteisessa tilassa, näkökenttä asteina, kuvasuhde (käytännössä ruudun leveys jaettuna ruudun korkeudella) sekä lähin ja pisin matka, johon halutaan nähdä (kuva 11).

```
//Position, field of view, aspect, closes we can see, furthest we can see.
Camera(const glm::vec3 pos, float fov, float aspect, float zNear, float zFar)
{
    _perspective = glm::perspective(fov, aspect, zNear, zFar);
    _position = pos;
    _forward = glm::vec3(0, 0, 1);
    _up = glm::vec3(0, 1, 0);
}
```

Kuva 11. Kameran alustus

GLM-kirjastosta löytyy funktio, jonka avulla kamera pystytään ”kääntämään” haluttuun pisteeseen. Kyseinen **lookAt**-funktio luo matriisin vastaanottamistaan: positiosta, josta katsotaan (esim. kameran sijainti), pisteestä, johon katsotaan, sekä ylöspäin osoittavan vektorin suunnasta (ohjelma haluaa siis tietää, mikä xyz-akseleista osoittaa ylöspäin, yleisimmin tämä on y-akseli) [19.]

Matriisilaskentaa tehdessä laskujärjestykseen on syytä kiinnittää erityistä huomiota. Jos halutaan siirtää objekti tiettyyn pisteeseen ja kääntää sitä esimerkiksi 90 astetta, on objekti ensin käännettävä ja sen jälkeen siirrettävä haluttuun pisteeseen. Jos kertolaskun tekee toisin päin, objekti siirretään aluksi haluttuun pisteeseen ja sen jälkeen sitä käännetään 90 astetta **origoon nähden**, jolloin lopputulos on aivan erilainen kuin mitä toivottiin. Laskutoimitus on melko yksinkertaista saada oikeaan järjestykseen: objektia laskutoimituksessa lähinnä olevat matriisit suoritetaan ensin. Jos siis halutaan esimerkiksi kääntää objektia ja sen jälkeen liikuttaa sitä, on laskutoimitus: liikkeen matriisi * kääntömatriisi * objektin sijainti. Matriisit voidaan tietenkin kertoa keskenään jo aikaisemmassa vaiheessa, järjestys säilyy silti samana [20; 21; 22.]

Matriisien laskennassa käytetään x-, y- ja z–koordinaattien lisäksi avuksi myös w-koordinaattia. Yksinkertaistettuna w-koordinaatti kertoo, onko kyseessä sijainti (jolloin arvo on 1) vaiko suunta (arvo 0) avaruudessa. Rotaation ja skaalauksen pystyy laskemaan 3D-koordinaateilla, mutta objektien liikuttelu vaatii 4D-koordinaattien käyttöä, sillä matriisin pystyrivien määrän on vastattava kerrottavan pisteen osien lukumäärää (x, y, z, w). Sovellus asettaa oletuksena w-koordinaatit arvoon 1.

6 3D-mallien lataus

Sovellus käyttää hyväkseen Autodeskin kehittämää, ilmaista fbx sdk:ta, jolla pystyy lataamaan fbx-formaattiin tallennettuja malleja. Fbx-formaatti tukee myös esimerkiksi animaatioita, joita sovellus tulee tukemaan tulevaisuudessa. [23; 24.] Kaikki objektin tiedot talletetaan Vertex-struktuuriin, joka sisältää tällä hetkellä sijainti-, väri-, normaali-, tekstuurikoordinaattitietoja sekä lisäksi kunkin objektin tekstuurin osoitteen.

Sovellus on rakennettu niin, että tarvittaessa käyttäjä voi ladata useamman fbx-tiedoston samanaikaisesti ruudulle. Ohjelma osaa myös ladata kokonaisen skenaarion, jossa on useita objekteja yhtäaikaisesti. Blenderiä käytettäessä on jokaisen objektin sijainti, rotaatio sekä skaalaus asetettava erikseen. Fbx sdk:ta käytettäessä käyttäjän tulee ensin luoda skenaario, josta tietoja voidaan kerätä tarpeen mukaan. Sovellukseen on luotu tätä varten funktion nimeltä **loadFBXScene** (katso kuva 12), joka ottaa vastaan halutun tiedoston osoitteen ja luo siitä uuden **FbxScene**-osoittimen, jonka se palauttaa sitä kutsuneelle taholle. Fbx sdk:ssa kaikki alkaa **FbxManagerin** luomisesta kutsulla FbxManager::Create(). Managerin luonnin jälkeen pitää myöskin luoda **FbxIOSettings**-tyypin

muuttuja, jonka avulla esitetään käytettävissä olevat tuonti-/vientivaihtoehdot. Seuraavaksi luodaan **FbxImporter**-muuttuja, jota käytetään myöhemmin itse skenaarion tietojen hakemiseen. Kun halutut tiedot on saatu kerätyksi talteen, on hyvä muistaa poistaa turhat muuttujat, jotta muistia ei kulu turhaan.

```
325   //Loads and returns the information of a fbx file.
326   FbxScene* GraphicsManager::loadFBXScene(const char* filePath)
327   {
328       //First we initialize the fbx sdk manager.
329       FbxManager* _sdkManager = FbxManager::Create();
330
331       //We also need an InputOutput settings object.
332       FbxIOSettings *_ios = FbxIOSettings::Create(_sdkManager, IOSROOT);
333
334       //Importer creation
335       FbxImporter* _importer = FbxImporter::Create(_sdkManager, "");
336
337       if (!_importer->Initialize(filePath, -1, _sdkManager->GetIOSettings()))
338       {
339           logError(_importer->GetStatus().GetErrorString());
340       }
341
342       //New scene
343       FbxScene* _scene = FbxScene::Create(_sdkManager, "SceneName");
344
345       //File contents are imported to the scene
346       _importer->Import(_scene);
347       //The importer is no longer required so it can be destroyed.
348       _importer->Destroy();
349
350       return _scene;
351   }
```

Kuva 12. Fbx-skenaarion luonti ja palautus.

Kun skenaario on luotu onnistuneesti, on aika kerätä siitä haluttuja tietoja. Jokainen fbx-skenaario koostuu solmuista, joiden alta löytyy lisää alisolmuja ja lopulta jokaisen skenaariossa esiintyvän objektin tiedot. Aluksi aloitetaan hakemalla skenaarion ensimmäinen solmu **GetRoodNode**-funktiolla. Kun on tarkistettu, että kyseinen solmu löytyy, voidaan siitä hakea kaikki alisolmut **GetChild**-funktiolla. Jokaisella alisolmulla on olemassa FbxMesh-muuttuja, josta saadaan jokaisen objektin data selville. Sovellus käy jokaisen alisolmun jokaisen silmukan lävitse ja kerää tarvittavat tiedot talteen aloittaen objektin pistetietojen keräyksestä. Sovellukseen luotiin **getMeshInformation**-niminen funktio, joka käy lävitse kaikki halutut tiedot yksitellen ja lisää ne aikaisemmin mainittuun Vertex-struktuuriin. Ensiksi vuorossa on **addMeshVertices**-funktio, jolle annetaan parametrina

solmun pointteri, josta tietoa haetaan, sekä Vertex-muuttujan pointteri, johon tiedot halutaan tallentaa. Funktio sisältää yhden for-silmukan, jossa jokainen w-koordinaatti asetetaan vakioarvoon 1.0f (kuva 13).

```
93    void GraphicsManager::addMeshVertices(FbxMesh* _mesh, Vertex* _meshObject)
94    {
95        int _verticeAmount = _mesh->GetControlPointsCount();
96        for (int i = 0; i < _verticeAmount; i++)
97        {
98            FbxVector4 _vector = _mesh->GetControlPointAt(i);
99
100           _meshObject[i].position.x = (GLfloat)_vector.mData[0];
101           _meshObject[i].position.y = (GLfloat)_vector.mData[1];
102           _meshObject[i].position.z = (GLfloat)_vector.mData[2];
103           _meshObject[i].position.w = 1.0f;
104       }
105   }
```

Kuva 13. Pistetietojen lisäys Vertex-struktuuriin.

Sovelluksesta löytyy myös funktio **addMeshTextures**, joka etsii objektissa mahdollisesti esiintyvän tekstuurin ja tallentaa sen osoitteen, jotta sen avulla voidaan myöhemmin luoda OpenGL-tekstuuri. Kyseinen funktio osaa myöskin tarkistaa, onko objektiin mahdollisesti liitetty materiaaleja, joissa esiintyy esimerkiksi väri-informaatiota. Tällaisia tilanteita tulee esiin esimerkiksi ladatessa low polygon -malleja.

Jokainen sovelluksen **addMesh***-funktioista (* Textures,Normals ja Vertices) toimii samalla periaatteella: parametreiksi lähetetään solmun osoitin, josta fbx-dataa halutaan etsiä sekä osoitin Vertex-muuttujaan, johon data halutaan lisätä. Sisällä addMeshTexture-funktiossa luodaan ensin **FbxGeometryElementMaterial**-muuttuja, johon ladataan solmusta mahdollisesti löytynyt materiaalidata **GetElementMaterial**-funktiolla. Seuraavaksi pitää luoda **FbxSurfaceMaterial**-osoitin, josta saadaan selville sekä materiaali- että tekstuuritietoja.

Mikäli grafiikkaobjektilla on värimateriaali, on se tallennettuna **FbxSurfaceLambert**-tyypin muuttujaan, joka saadaan suoraan muuttamalla aikaisemmin luotu FbxSurfaceMaterial tähän muotoon. SurfaceLambert sisältää sovelluksen tarvitsemat väritiedot, jotka on tallennettu Diffuse-ominaisuuteen. Kun objektin väri on saatu selville, asetetaan kyseinen väri Vertex-muuttujaan for-silmukassa (kuva 14). Kuvassa 14 näkyy myös FbxProperty-muuttuja **_prop**, jota käytetään funktiossa myöhemmin, kun etsitään grafiikkaobjektiin mahdollisesti liitettyä tekstuuritiedostoa.

Kuva 21 (liite 1) näyttää skenaarion, jossa on ladattuna useampia objekteja väritietoineen. Kyseisessä skenaariossa ei ole valaistusta lainkaan, joten tästä syystä pintojen muotoja on vaikeampi hahmottaa. Toistaiseksi sovelluksessa yksittäisellä objektilla voi olla ainoastaan yksi väri, mutta tämä ei ole lopullinen ratkaisu.

```
252     //First we try to get the element material out of the mesh.
253     FbxGeometryElementMaterial* _materialElement = _mesh->GetElementMaterial(0);
254     //If found:
255     if (_materialElement)
256     {
257         int _materialID = _materialElement->GetIndexArray().GetAt(0);
258         FbxSurfaceMaterial* _surfaceMaterial = _mesh->GetNode()->GetMaterial(_materialID);
259         if (_materialElement >= 0)
260         {
261             FbxProperty _prop = _surfaceMaterial->FindProperty(FbxSurfaceMaterial::sDiffuse);
262             //Let's check if the surface material has a colour.
263             if (_surfaceMaterial->GetClassId().Is(FbxSurfaceLambert::ClassId))
264             {
265                 FbxSurfaceLambert *_lambert = (FbxSurfaceLambert*)_surfaceMaterial;
266                 FbxPropertyT<FbxDouble3> _colourProperty = _lambert->Diffuse;
267                 FbxDouble3 _colour = _colourProperty.Get();
268                 int _verticeAmount = _mesh->GetControlPointsCount();
269                 for (int j = 0; j < _verticeAmount; j++)
270                 {
271                     _meshObject[j].colour.r = (GLfloat)_colour[0];
272                     _meshObject[j].colour.g = (GLfloat)_colour[1];
273                     _meshObject[j].colour.b = (GLfloat)_colour[2];
274                     _meshObject[j].colour.a = 1.0f;
275                 }
276             }
```

Kuva 14. Materiaalin väritietojen tallennus Vertex-muuttujaan.

Kuten aikaisemmin mainittiin, grafiikkaobjektiin voidaan myös liittää tekstuuri. Toistaiseksi sovellus tukee pelkästään yhtä tekstuuria jokaista grafiikkaobjektia kohden, mutta tulevaisuudessa yhdellä objektilla voi mahdollisesti olla useampia tekstuureja. Aluksi tarkistetaan kerroksittaisten tekstuurien lukumäärä, ja jos tämä lukumäärä on 0, haetaan _prop-muuttujasta **FbxTexture**-tyyppistä tekstuuria. Tekstuurin löydyttyä se lisätään _meshObjectin _**texturePath**-muuttujaan (joka on tyyppiä string) myöhempää käyttöä varten (kuva 15).

```
278                 //Let's check if the material has layered textures.
279                 GLuint _layeredTextureCount = _prop.GetSrcObjectCount<FbxLayeredTexture>();
280                 if (_layeredTextureCount > 0)
281                 {
282                     std::cout << "layer count: " << _layeredTextureCount << std::endl;
283                 }
284                 else
285                 {
286                     GLuint _textureCount = _prop.GetSrcObjectCount<FbxTexture>();
287                     std::cout << "Texture count: " << _textureCount << std::endl;
288                     for (int i = 0; i < _textureCount; i++)
289                     {
290                         FbxTexture* _texture = FbxCast<FbxTexture>(_prop.GetSrcObject<FbxTexture>(i));
291                         FbxFileTexture* _fileTexture = FbxCast<FbxFileTexture>(_texture);
292                         _meshObject->_texturePath = _fileTexture->GetFileName();
293                     }
294                 }
```

Kuva 15. Tekstuurin lisäys _meshObjectiin.

Tekstuureja varten tarvitaan myöskin tekstuurikoordinaatit, joiden avulla OpenGL tietää, mihin kohtaan ja miten tekstuuri tulee objektiin kiinnittää. Tätä varten sovelluksesta löytyy **addUVData**-niminen funktio. Toiminnallisuus ei ole aivan täysin valmis, mutta tekstuurin pystyy lisäämään osittain objektiin (liite 2).

Kun sovelluksessa halutaan käyttää erilaisia valoja, on objekteille tallennettava myös niiden pintojen normaalit. Tämä tehdään sovelluksesta löytyvällä **addMeshNormals**-funktiolla, joka käy objektin kulmapisteet läpi ja tallentaa kaikki niiden normaalit talteen. On hyvä huomata, että yhdellä kulmapisteellä voi olla useampi normaali riippuen siitä, mitä pintaa vasten kyseistä kulmapistettä verrataan [25]. Tämä toiminnallisuus on tekstuurikoordinaattien tavoin vielä kesken (aiheesta lisää luvussa 9).

7 3D-mallin luominen OpenGL:llä

Kun kaikki halutut tiedot on saatu kerättyä talteen fbx-tiedostosta, voidaan siirtyä OpenGL:n puolelle. Sovellus sisältää GraphicsObject-nimisen luokan, joka sisältää kaiken tarvittavan OpenGL-mallien luomiseen ja piirtämiseen. Piirtometodi tullaan ehkä tulevaisuudessa siirtämään pois luokan sisältä, mutta toistaiseksi kyseinen funktio löytyy vielä luokasta, sillä se helpottaa koodin kirjoittamista. Käyttäjän valitsemat tiedostot toimitetaan GraphicsManager-luokalle, joka alustaa jokaisesta tiedostosta löytyvät objektit yksitellen ja tallentaa ne sisältämäänsä GraphicsObject-tyyppiseen tauluun myöhempää käyttöä varten.

Kun ruudulle halutaan piirtyvän jotain, on näytönohjaimelle annettava piirtokäsky. Ennen modernia OpenGL:ää oli tapana, että tietokoneen prosessori lähetti jokaisen piirtokäskyn erikseen näytönohjaimelle – tästä seurasi melko suuria ongelmia myöhemmin erilaisten pullonkaulojen muodossa. Moderni OpenGL käyttää **VBO** (vertex buffer object) -nimisiä objekteja, joilla on mahdollista siirtää isoja määriä dataa näytönohjaimen piirrettäväksi kerralla. Tämä toimenpide vähentää tietokoneen prosessorin kuormaa huomattavasti ja sallii paremmin näytönohjaimen potentiaalin hyödyntämisen. Modernissa OpenGL:ssä on käytössä myös **VAO** (vertex array object), joka huolehtii tiedon siirtämisestä sovellukselta kulmapistevarjostimelle oikeassa muodossa.

Uusi grafiikkaobjekti alustetaan GraphicsObject-luokasta löytyvällä **initializeObject**-funktiolla (kuva 16), joka ottaa vastaan Vertex-muuttujan pointterin (joka siis sisältää piirrettävän objektin tiedot), pisteiden lukumäärän, indeksit sekä indeksien lukumäärän. Koska sovelluksessa käytetään osaa näistä arvoista myöhemmin, talletetaan ne muistiin funktion alussa. Seuraavaksi funktiossa luodaan uusi vertex array -objekti kutsumalla **glGenVertexArrays**-nimistä funktiota, jolle annetaan parametriksi luotavien objektien määrä sekä referenssi GLuint-tyypin muuttujaan, johon VAO halutaan alustaa. Kun objekti on luotu, pitää se vielä ottaa käyttöön **glBindVertexArray**-funktiolla, jolle annetaan parametriksi juuri alustettu GLuint-muuttuja.

Tämän jälkeen onkin aika luoda VBO, johon piirrettävän grafiikkaobjektin data siirretään. VBO luodaan hyvin samaan tapaan kuin VAO kutsumalla **glGenBuffers**-nimistä funktiota. Sovellus käyttää tässä yhteydessä GLuint-muuttujia sisältävää taulua, koska sovelluksessa tarvitaan useampia puskuriobjekteja, jotta ohjelma toimii oikein. Puskuriobjektin luonnin jälkeen vbo otetaan käyttöön **glBindBuffer**-funktiolla, jolle annetaan parametriksi puskurin tyyppi sekä juuri alustettu puskuriobjekti. Itse data siirretään näytönohjaimelle **glBufferData**-nimisellä funktiolla, joka ottaa parametreinaan vastaan puskurin tyypin, koon, datan sekä piirtotyylin.

Tämän operaation jälkeen funktio kertoo VAO:lle, missä formaatissa tieto pitää syöttää kulmapistevarjostimelle. Tämä onnistuu **glVertexAttribPointer**-funktiolla, jolle annetaan parametriksi attribuutin indeksi kulmapistevarjostimessa, yksittäisen kulmapisteen komponenttien määrä, datan tyyppi, GLenum sen mukaan onko data normalisoitu vai ei, kulmapisteiden välinen ero sekä pointteri siihen kohtaan, josta data alkaa. Seuraavaksi sovelluksessa kutsutaan **glEnableVertexAttribArray**-funktiota, joka ottaa kyseisen attribuutin käyttöön.

Sovellus käyttää hyödykseen myös indeksejä, joita varten pitää luoda puskuri samalla tavalla kuin kulmapistepuskuri luotiin. Indeksien käyttö vähentää tallennettavien pisteiden määrää, sillä jokainen kulmapiste pitää tallentaa muistiin vain kerran, ja siihen voidaan viitata indeksillä tarvittaessa. Tämä nopeuttaa isompien mallien piirtämistä, jotka saattavat sisältää erittäin suuren määrän kulmapisteitä [26; 27.]

```
135  void GraphicsObject::initializeObject(Vertex *_objectInfo, int _verticeAmount,GLuint* _indices, GLuint _indiceCount)
136  {
137
138      this->_indices = _indices;
139      this->_indiceCount = _indiceCount;
140      this->_texturePath = _objectInfo->_texturePath;
141
142      //First we create the vertex array object.
143      glGenVertexArrays(1, &_vao);
144      glBindVertexArray(_vao);
145      //Then we create the vertex buffer.
146      glGenBuffers(1, &_buffer[0]);
147      glBindBuffer(GL_ARRAY_BUFFER, _buffer[0]);
148      glBufferData(GL_ARRAY_BUFFER, sizeof(Vertex)*_verticeAmount, _objectInfo, GL_STATIC_DRAW);
149
150      //Point to the position data in Vertex.
151      glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)offsetof(Vertex,position));
152      glEnableVertexAttribArray(0);
153
154      //Lastly we create the index data and buffer it.
155      glGenBuffers(1, &_buffer[1]);
156      glBindBuffer(GL_ELEMENT_ARRAY_BUFFER,_buffer[1]);
157      glBufferData(GL_ELEMENT_ARRAY_BUFFER, _indiceCount*sizeof(GLuint), _indices, GL_STATIC_DRAW);
158
159  }
```

Kuva 16. Grafiikkaobjektin alustus ja sen kulmapistetietojen puskurointi näytönohjaimelle.

Mikäli objektista löytyi aikaisemmin tekstuuri, myös se tulee alustaa. Tätä varten ohjelmasta löytyy **Texture**-niminen luokka, joka osaa lukea png-tyyppisiä kuvia ja luoda niistä tekstuurielementtejä OpenGL:lle. Luokka käyttää hyväkseen virallista png-tiedostojen lukemiseen käytettävää kirjastoa nimeltään **libpng**, joka pystyy lukemaan kuvasta kaikki OpenGL:n tarvitsemat tiedot talteen. Tavoitteena on tulevaisuudessa laajentaa luokka tukemaan myös kuvia, jotka on tallennettu eri formaatteihin (jpeg, bmp). Toistaiseksi kuitenkin Blenderillä tehdyissä objekteissa tulee olla png-tyypin tekstuuri, jos tekstuurien halutaan näkyvän sovelluksessa.

Sovellus ottaa yhteyttä Texture-luokan **loadTexture**-funktioon, joka luo tekstuurin ja palauttaa sen käyttäjälle. Kirjaston pitkähkön alustamisen jälkeen sovellukseni lukee kuvan tärkeimmät tiedot talteen **_imageData** sekä **_rowPointers** nimisiin muuttujiin, joiden avulla tekstuuri pystytään alustamaan. Tekstuuri luodaan hyvin samalla tavalla, kuin aikaisemmin luodut puskuriobjektit. **glCreateTextures**-funktiolle syötetään parametreina tekstuurin tyyppi, luotavien tekstuurien määrä sekä muuttuja, johon tekstuuri halutaan alustaa. Seuraavaksi sovelluksen täytyy tietää, kuinka paljon kyseinen tekstuuri tulee

viemään tilaa. Tähän tarkoitukseen löytyy funktio nimeltä glTextureStorage2D, jolle annetaan parametreiksi haluttu tekstuuriobjekti, mipmappien lukumäärä, datan tyyppi sekä tekselien lukumäärä (käytännössä kuvan korkeus kertaa leveys). Mipmapeilla tarkoitetaan saman kuvan eriresoluutioisia versioita. Niitä käytetään moneen tarkoitukseen, kuten esimerkiksi nopeuttamaan piirtoaikoja, parantamaan kuvanlaatua sekä vähentämään näytönohjaimen rasitusta. [29]

Kun tilatiedot on täytetty, on tekstuuri otettava käyttöön **glBindTexture**-funktiolla. Parametreikseen funktio tarvitsee tekstuurin tyypin sekä tekstuuriobjektin. Seuraavaksi data on toimitettava tekstuurille **glTextureSubImage2D**-funktiolla, jotta sitä voidaan käyttää sovelluksessa. Parametriksi funktio tarvitsee halutun tekstuuriobjektin, tekstuuritasojen lukumäärän, x- sekä y-koordinaattien tekselioffsetit tekstuuritaulussa, tekstuurin korkeuden sekä leveyden, pikselidatan formaatin, tyypin sekä osoittimen muistissa olevaan kuvadataan. Datan lisäyksen jälkeen luodut taulut voidaan poistaa, koska tietoja ei enää tarvita. Tämän jälkeen funktio palauttaa luodun tekstuurin takaisin sitä kutsuneelle funktiolle (kuva 17).

```
136     GLuint _texture;
137     //First we create a new 2D texture object
138     glCreateTextures(GL_TEXTURE_2D, 1, &_texture);
139
140     //Then we need to tell how much storage space the texture uses. More info in OpenGL Superbible's page 153
141     glTextureStorage2D(_texture, 1, GL_RGBA32F, _width, _height);
142     //Then we bind the texture
143     glBindTexture(GL_TEXTURE_2D, _texture);
144
145     //Next we update the texture data.
146     //Parameters: The texture object, level, offsets (2),size in texels (2),
147     //four-channel data, floating point data and the pointer to data.
148     glTextureSubImage2D(_texture, 0, 0, 0, _width, _height, GL_RGBA, GL_UNSIGNED_BYTE, (GLvoid*)_imageData);
149     GLenum error = glGetError();
150     if (error != 0)
151     {
152         std::cout << "Texture error: " << error << std::endl;
153     }
154     //Clean up memory and close everything after use
155     png_destroy_read_struct(&_pngPtr, &_infoPtr, &_endInfo);
156     delete[] _imageData;
157     delete[] _rowPointers;
158     fclose(fp);
159
160     return _texture;
161 }
```

Kuva 17. Tekstuurin luonti Texture-luokan loadTexture-funktiossa.

Kun objekti on alustettu ja kaikki tarpeellinen data on liitetty siihen, voidaan se piirtää GraphicsObject-luokasta löytyvällä **drawObject**-piirtofunktiolla. Ruudunpäivityksestä vastaava funktio ottaa yhteyttä GraphicsManager-luokkaan ja käskee tätä piirtämään koko alustetun skenaarion. Jokainen alustettu objekti on tallennettu vektoriin, joka käydään piirrettäessä läpi.

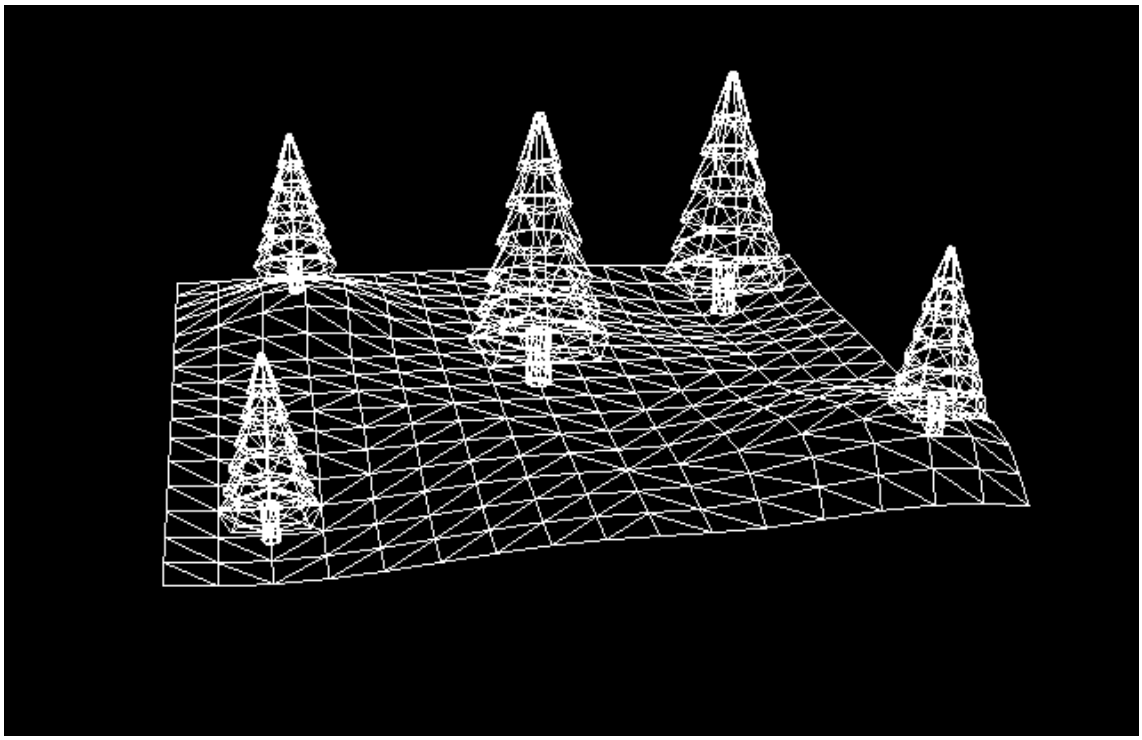
Objektien piirtäminen on varsin yksinkertaista: ennen jokaista piirtokutsua pitää aikaisemmin alustettu kulmapistetaulukko-objekti (vao) ottaa käyttöön glBindVertexArray-komennolla, jolle annetaan kyseinen vao parametrina. Seuraavaksi kutsutaan **glDrawElements**-funktiota, joka suorittaa itse objektin piirtämisen. Funktio tarvitsee tiedon siitä, millaisia kuvioita käyttäjä haluaa piirtää, piirrettävien elementtien lukumäärän, piirrettävän datan tyypin sekä indeksien osoitteen muistissa. Piirtokutsun jälkeen vao vapautetaan kutsumalla uudestaan glBindVertexArray-funktiota parametrilla 0 [28.]

```
173  void GraphicsObject::drawObject(float _currentTime)
174  {
175      //Please note that when using Tessellation you must use GL_PATCHES instead of GL_TRIANGLES
176      glBindVertexArray(_vao);
177
178      glDrawElements(GL_TRIANGLES,this->_indiceCount,GL_UNSIGNED_INT,(void*)0);
179
180      glBindVertexArray(0);
181  }
182
```

Kuva 18. Objektin piirtofunktio.

Kuten kuvan 16 piirtometodista käy ilmi, sovelluksessa esiintyvät mallit rakentuvat kolmioista. Tästä syystä mahdollisessa kolmannen osapuolen sovelluksessa pitää varmistaa, että objektit koostuvat myöskin kolmioista. Blenderissä tämän voi tehdä valitsemalla objektin, vaihtamalla muokkaustilaan (Edit Mode) ja valitsemalla objektin asetuksista (Ctrl+F) Triangulate Faces.

Kuvassa 19 näkyy skenaario, jossa jokaisesta objektista on alustettu ainoastaan kulmapisteet, joiden avulla skenaario on piirretty. Koska sovelluksesta löytyy myös aiemmin luotu ”kamera”, pystyy käyttäjä tarvittaessa tarkastelemaan skenaariota eri kuvakulmista, vaikkakin toistaiseksi ominaisuus on kovakoodattu ja kamera kulkee samaa rataa pitkin koko ohjelman suorituksen ajan. Sovellukseen on tulossa ominaisuus, joka mahdollistaa myös skenaarion tarkastelemisen näppäimistön ja hiiren avulla.



Kuva 19. Kuvassa skenaario, jossa käytetty piirtämiseen vain objektien kulmapisteitä.

Sovellus osaa myös lukea skenaarion objekteista mahdollisesti löytyneiden materiaalien tietoja. Liite 1 näyttää kuvan skenaariosta, jossa ei ole valoa, mutta jokaisella objektilla on oma väri, joka on määritelty Blenderillä.

8 Valon käyttäminen

Kuten kuvasta 21 (liite 1) käy ilmi, skenaario ilman valoa näyttää melko tylsältä, ja pintojen yksityiskohdat jäävät suurimmaksi osaksi katselijalta piiloon. Sovelluksessa ei toistaiseksi ole täysin toimivaa valaistusta johtuen pintojen normaaleissa ilmenneestä ongelmasta, mutta erilaisista valaistustavoista on silti hyvä olla tietoinen. Valaisutapoja on olemassa monia, joista seuraavaksi käsitellään perusasioita ympäröivästä valaistuksesta (ambient lighting) sekä kohdevalaistuksesta.

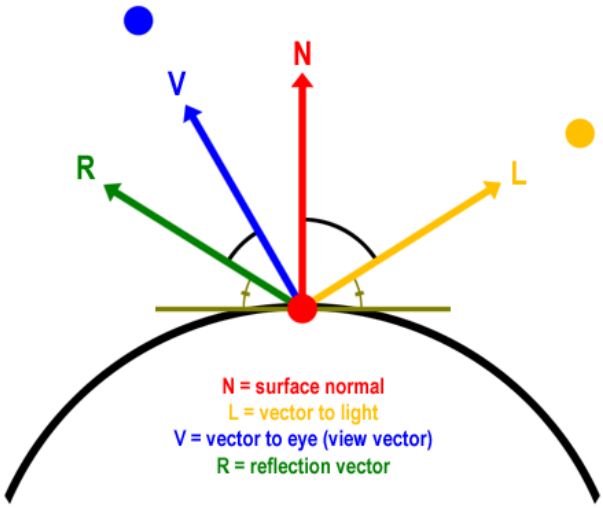
8.1 Ympäröivä valaistus

Yksi valaistustapa on käyttää ympäröivää valaistusta, jossa valolla ei ole yhtä selkeää lähdettä. Tällaisessa valaistuksessa valo on kimpoillut ympäristössään niin paljon, että

se on menettänyt suunnan ja valaistus on yhtä voimakasta kaikkialla. Valon heijastuminen pinnoilta on melko yksinkertaista laskea: kukin värikomponentti kerrotaan ympäröivällä valaistuksella (eli siis esimerkiksi vec3 color * vec3 ambient). Ympäröivä valaistus on valaistuksista kaikista helpoin toteuttaa.

8.2 Kohdevalaistus

Kohdevalaistuksessa valolla on selkeä lähde, ja se vaikuttaa skenaariossa näkyvien objektien pintojen väreihin sen mukaan, missä kulmassa ja millä voimakkuudella se pintoihin osuu. Yksi tapa matkia kohdevalaistusta on käyttää niin sanottua Phong-valaistusta (kutsutaan myös nimellä Phong-heijastusmalli, Phong reflection model). Kuvassa 20 näkyy Phong-valaistukseen tarvittavat vektorit, joiden avulla jokaisen pinnan väri voidaan laskea. R kuvastaa valon heijastusta pinnasta, N pinnan normaalia, L valonlähdettä sekä V katselijaa (kamera).



Kuva 20. Phong-valaistuksessa käytettävät vektorit.

Phong-heijastus perustuu siihen, että jokaisella pinnalla ja valonlähteellä on olemassa kolme eri heijastusarvoa: ympäröivä-, diffuusi- sekä spekulaarinen (ambient, diffuse, specular) arvo. Lopullinen pinnan heijastus on näiden komponenttien summa. Komponentit lasketaan seuraavasti:

- Ympäröivä heijastus on yleensä vakio, ja se koostuu valonlähteen sekä grafiikkaobjektin pinnan ambient-komponenttien tulosta.

- Diffuusi heijastus (hajaheijastus) lasketaan laskemalla aluksi valonlähteen sekä pinnan normaalin pistetulo ja valitsemalla saaduista kahdesta arvosta suurempi (GLSL-kielestä löytyy tätä varten **max-**funktio). Saatu tulos kuvastaa vektorien välisen kulman kosinia. Tämä tulos kerrotaan vielä sekä valonlähteen että grafiikkaobjektin pinnan diffuusiarvoilla, jolloin saadaan pinnan heijastama diffuusi valoarvo. Tällä laskutoimituksella on mahdollista saada myös negatiivisia tuloksia, jolloin tulos pitää muuttaa arvoon 0, koska valon arvo ei voi olla negatiivinen.

- Spekulaarinen heijastus lasketaan selvittämällä ensin valon heijastumissuunta R (GLSL:ssä on tätä varten funktio **reflect**, jolle annetaan parametreiksi negatiivinen valon L suuntavektori sekä kameran normaalivektori V). Tämän jälkeen lasketaan kameran normaalivektorin V sekä heijastusvektorin pistetulo ja valitaan suurempi kahdesta saadusta arvosta. Seuraavaksi tämä arvo nostetaan heijastuvuuden potenssiin (heijastuvuus on ohjelmoijan valitsema vakioarvo, yleensä kuitenkin maksimissaan 128.0). Tästä saatu arvo kerrotaan vielä materiaalin sekä valolähteen spekulaarisilla arvoilla.

Koko laskutoimitus:

$$I_p = k_a i_a + k_d(\vec{L} \cdot \vec{N}) i_d + k_s(\vec{R} \cdot \vec{V})^{\alpha} i_s, \qquad (1)$$

jossa k_a on materiaalin ja i_a valonlähteen ambienttikomponentti, k_d on materiaalin ja i_d valonlähteen diffuusikomponentti ja k_s materiaalin ja i_s spekulaarikomponentti eksponentin $\propto$ ollessa haluttu heijastuvuus [29.]

Sovelluksessa on olemassa Phong- sekä Gouraud-valaistus, mutta valaistustoiminnallisuus ei ole virheellisistä normaaleista johtuen kunnossa (katso nykyinen tilanne liite 2).

9 Suurimmat ongelmakohdat

9.1 Tekstuurikoordinaatit

Sovellus ei toimi täysin tekstuurikoordinaattien suhteen: tekstuurit eivät jostain syystä näy kappaleessa oikein, vaan ainoastaan yksi sivu on teksturoitu (liite 2) grafiikkaobjektista riippumatta. Tarkoituksena olisi, että ohjelma osaisi automaattisesti tarkistaa onko objektiin lisätty tekstuuria ja jos on niin ohjelma luo tekstuurin objektille sovelluksessa. Ongelma lienee siinä, että objektin pistetiedot ovat indeksoituja, mutta tekstuurikoordinaatteja ei lisätä oikein jokaiselle pisteelle.

9.2 Normaalit

Sovelluksessa ei toistaiseksi tallenneta normaaleja oikein ja tästä syystä myöskään valo ei heijastu pinnoilta oikein (liite 2). FBX SDK palauttaa tietyn pisteen normaalien keskiarvon (yhdellä pisteellä voi siis olla useampia normaaleja riippuen siitä, millä pinnalla kyseinen kulmapiste sijaitsee), kun sen sijaan tarvittaisiin yhdelle kulmapisteelle sen jokainen normaali. Ongelmasta johtuen myöskin sovelluksen valaistus ei näy oikein, vaikka valaistus muuten toimiikin. Tarvittaessa normaalit voidaan laskea kulmapisteiden avulla, joten ongelmaa ei ole kovin vaikea ratkaista.

9.3 Materiaalit ja niiden sisältämä väritieto

Luettaessa 2D- ja 3D-mallien materiaaleja törmättiin myös ongelmaan väritietojen lukemisen kanssa. Tällä hetkellä jokainen Blenderillä luotu malli näyttäisi sisältävän ainoastaan yhden materiaalivärin, joka tuottaa hiukan ongelmia piirrettäessä objekteja (kuten esimerkiksi puita), joissa yleensä luonnollisesti on monta eri väriä. Ongelma on kuitenkin ratkaistavissa.

9.4 Kolmannen osapuolen mallinnusohjelmat

Luotaessa 2D- ja 3D-malleja kolmannen osapuolen sovelluksella on hyvä varautua muutamiin asioihin. Ensin kannattaa selvittää, minkälaista koordinaatistoa ohjelma käyttää.

Blenderiä käytettäessä ongelmakohtia oli muutamia. Jokaisen objektin rotaatio, skaalaus ja sijaintitiedot pitää erikseen lisätä objektiin ennen niiden tuomista OpenGL-sovellukseen, tai muuten ne eivät näy ruudulla oikein.

Rotaatio-ongelma johtuu siitä, että Blender käyttää xzy-koordinaatteja yleisemmin käytetyn xyz-koordinaatiston sijaan. Tästä johtuen jokainen objekti näkyi aluksi 90 asteen kulmassa sovelluksessa. Lisäksi jos skenaario sisälsi useamman kuin yhden objektin, asetettiin kaikki objektit jostain syystä origoon. Ongelma ratkesi lisäämällä jokaiselle objektille erikseen sijainnin, rotaation sekä skaalan.

10 Yhteenveto

Tällä hetkellä sovellus on normaalien ja tekstuurien osalta hiukan kesken. Molemmat tiedot saadaan kuitenkin haettua objektilta, mutta niitä ei ilmeisesti tallenneta oikein. Sovelluksesta löytyy tästä huolimatta suurin osa ominaisuuksista, joita siltä haluttiin löytyvän. Sovellus osaa ladata, tietyin rajoituksin, kolmannen osapuolen sovelluksissa tehtyjä fbx-tiedostoja ja kokonaisia skenaarioita ruudulle, käyttäjä voi kirjoittaa ja käyttää erilaisia varjostimia halunsa mukaan ja skenaariota voi tarkastella kameran avulla. Sovelluksessa on myös muutamia erilaisia valaistusvaihtoehtoja, mutta näiden toiminta varmistuu vasta, kun grafiikkaobjektien normaalien ongelma on saatu selvitettyä.

Tämän insinöörityön tekovaiheessa Khronos Group julkaisi OpenGL:n seuraavan kehitysasteen, jota kutsutaan nimellä Vulkan [30]. Vulkan perustuu AMD:n Mantle APIin ja se tarjoaa alemman tason grafiikkarajapinnan, jolla pääsee entistä paremmin käsiksi suoraan näytönohjaimen toiminnallisuuteen. Vulkanin lukuisten etujen vuoksi aion päivittää sovellukseni tukemaan Vulkania.

OpenGL:n käyttö on suhteellisen yksinkertaista oppia, joskin toisinaan se voi olla hiukan sekavaa johtuen informaation suuresta määrästä sekä siitä, että saman asian voi OpenGL:ssä yleensä tehdä monella eri tavalla. Internetistä löytyy loputon määrä opetusmateriaalia, josta voi olla hankalaa erotella uusimmat ja parhaimmat käytännöt. Lisäksi internetistä löytyvässä informaatiossa isona ongelmana oli myös se, että usein foorumeilla liikkuu virheellisiä toteutuksia, joita sitten kopioidaan sivulta toiselle. Osittain tästä syystä hankin itselleni kirjan nimeltä OpenGL Super Bible, joka keskittyy OpenGL 4.5-versioon ja sen käyttöön [31]. Kirjasta oli sovellusta tehdessä paljon apua, sillä siinä asiat

on selitetty yksityiskohtaisesti ja usein myös esimerkkejä käyttäen. Grafiikkaohjelmointia opiskelevan on mielestäni helpompi oppia ensin modernin OpenGL:n toimintaa ennen (mahdollisesti) Vulkaniin siirtymistä, sillä Vulkan on alemman tason rajapinta, jossa käyttäjän pitää ottaa huomioon monia asioita, joita OpenGL hoitaa automaattisesti.

Lähteet

1. Graham Sellers, Richard S. Wright, Jr, Nicholas Haemel. 31.7.2015. The Origins and Evolution of OpenGL. OpenGL Super Bible 7. painos, s. 7.

2. Graham Sellers, Richard S. Wright, Jr, Nicholas Haemel. 31.7.2015. Simplified graphics pipeline. OpenGL Super Bible 7. painos, s. 6.

3. Lesson 2: Don't put everything in main. 15.8. 2013. Verkkodokumentti. Will Usher. <http://www.willusher.io/pages/sdl2/>. Päivitetty 2.1.2016. Luettu 8.2.2016.

4. David Wolff. July 2011. Compiling a shader. OpenGL 4.0 Shading Language Cookbook, s. 15.

5. David Wolff. July 2011. Linking a shader program. OpenGL 4.0 Shading Language Cookbook, s. 18.

6. VertexShader. 8.9.2009. Verkkodokumentti. OpenGL Wiki <https://www.opengl.org/wiki/Vertex_Shader>. Päivitetty 4.7.2015. Luettu 15.2.2016.

7. Graham Sellers, Richard S. Wright, Jr, Nicholas Haemel. 31.7.2015. Passing data to the vertex shader. OpenGL Super Bible 7. painos, s. 28.

8. Tessellation. 4.10.2012. Verkkodokumentti. OpenGL Wiki <https://www.opengl.org/wiki/Tessellation>. Päivitetty 4.8.2015. Luettu 28.2.2016.

9. Graham Sellers, Richard S. Wright, Jr, Nicholas Haemel. 31.7.2015. Tessellation. OpenGL Super Bible 7. painos, s. 33.

10. Graham Sellers, Richard S. Wright, Jr, Nicholas Haemel. 31.7.2015. Tessellation. OpenGL Super Bible 7. painos, s. 306.

11. Geometry Shader. 18.9.2012. Verkkodokumentti. OpenGL Wiki <https://www.opengl.org/wiki/Geometry_Shader>. Päivitetty 14.12.2015. Luettu 1.3.2016.

12. Graham Sellers, Richard S. Wright, Jr, Nicholas Haemel. 31.7.2015. Geometry Shaders. OpenGL Super Bible 7. painos, s. 37.

13. 3 Rasterization. 29.3.1997. Verkkodokumentti. David Blythe. <https://www.opengl.org/documentation/specs/version1.1/glspec1.1/node41.html>. Luettu 4.4.2016.

14. Graham Sellers, Richard S. Wright, Jr, Nicholas Haemel. 31.7.2015. Primitive Assembly, Clipping and Rasterization. OpenGL Super Bible 7. painos, s. 39.

15. Fragment Shader. 8.9.2012. Verkkodokumentti. OpenGL Wiki <https://www.opengl.org/wiki/Fragment_Shader>. Päivitetty 21.10.2015. Luettu 15.2.2016.

16. Graham Sellers, Richard S. Wright, Jr, Nicholas Haemel. 31.7.2015. Fragment Shaders. OpenGL Super Bible 7. painos, s. 43.

17. Graham Sellers, Richard S. Wright, Jr, Nicholas Haemel. 31.7.2015. Frame-buffer Operations. OpenGL Super Bible 7. painos, s. 47.

18. Stencil testing. Verkkodokumentti. LearnOpenGL. <http://www.learno-pengl.com/#!Advanced-OpenGL/Stencil-testing>. Luettu 5.3.2016.

19. Camera. Verkkodokumentti. Learn OpenGL. <http://learnopengl.com/#!Getting-started/Camerahttps://www.opengl.org/wiki/Shader_Compilation>. Luettu 20.2.2016.

20. Tutorial 3: Matrices. Verkkodokumentti. OpenGL-Tutorial. <http://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/> Luettu 15.2.2016.

21. Modern OpenGL 03 – Matrices, Depth Buffering, Animation. 16.12.2012. Verk-kodokumentti. Tom Dalling. <http://www.tomdalling.com/blog/modern-opengl/03-matrices-depth-buffering-animation/>. Luettu 15.2.2016.

22. Graham Sellers, Richard S. Wright, Jr, Nicholas Haemel. 31.7.2015. Math for 3D Graphics. OpenGL Super Bible 7. painos, s. 55.

23. Class List. Verkkodokumentti. Autodesk Fbx 2016. <http://help.auto-desk.com/view/FBX/2016/ENU/?guid=__cpp_ref_annotated_html>. Luettu 25.2.2016.

24. Your First FBX SDK Program. Verkkodokumentti. Autodesk FBX 2016. <http://help.autodesk.com/view/FBX/2016/ENU/?guid=__fi-les_GUID_29C09995_47A9_4B49_9535_2F6BDC5C4107_htm>. Luettu 25.2.2016.

25. Graham Sellers, Richard S. Wright, Jr, Nicholas Haemel. 31.7.2015. Normal Mapping. OpenGL Super Bible 7. painos, s. 582-586.

26. Graham Sellers, Richard S. Wright, Jr, Nicholas Haemel. 31.7.2015. Buffers. OpenGL Super Bible 7. painos, s. 100.

27. Vertex Specification. 3.10.2012. Verkkodokumentti. OpenGL Wiki. <https://www.opengl.org/wiki/Vertex_Specification>. Päivitetty 16.12.2015. Luettu 18.3.2016.

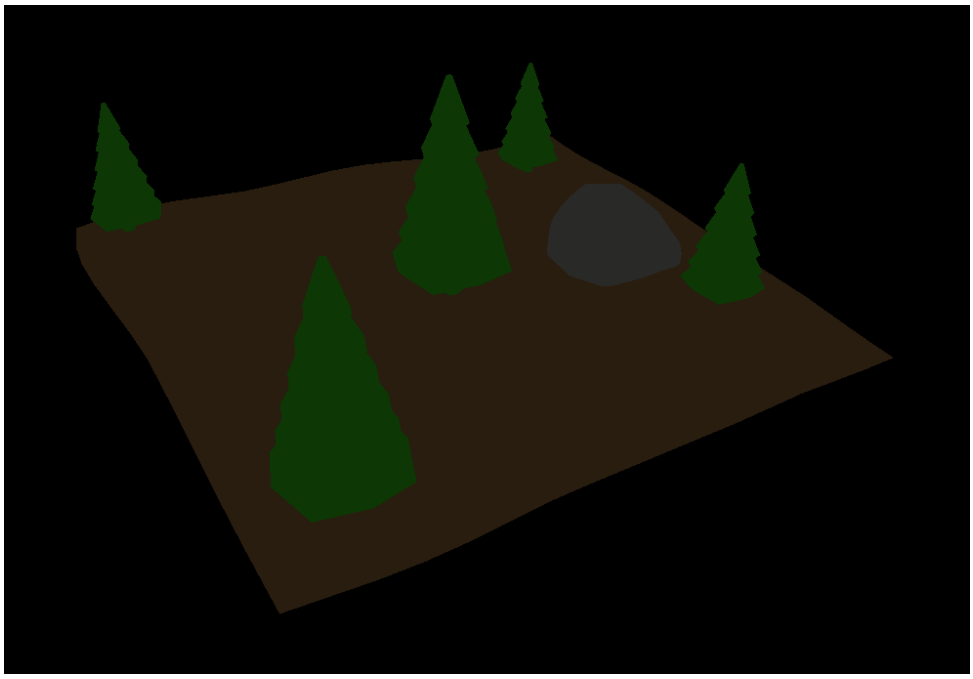
28. Vertex Rendering. 5.9.2012. Verkkodokumentti. OpenGL Wiki. <https://www.opengl.org/wiki/Vertex_Rendering>. Päivitetty 24.1.2016. Luettu 15.2.2016.

29. Graham Sellers, Richard S. Wright, Jr, Nicholas Haemel. 31.7.2015. Lighting Models. OpenGL Super Bible 7. painos, s. 568.

30. Vulkan. 16.2.2016. Verkkodokumentti. Khronos Group. < https://www.khronos.org/vulkan/>. Luettu 1.3.2016.

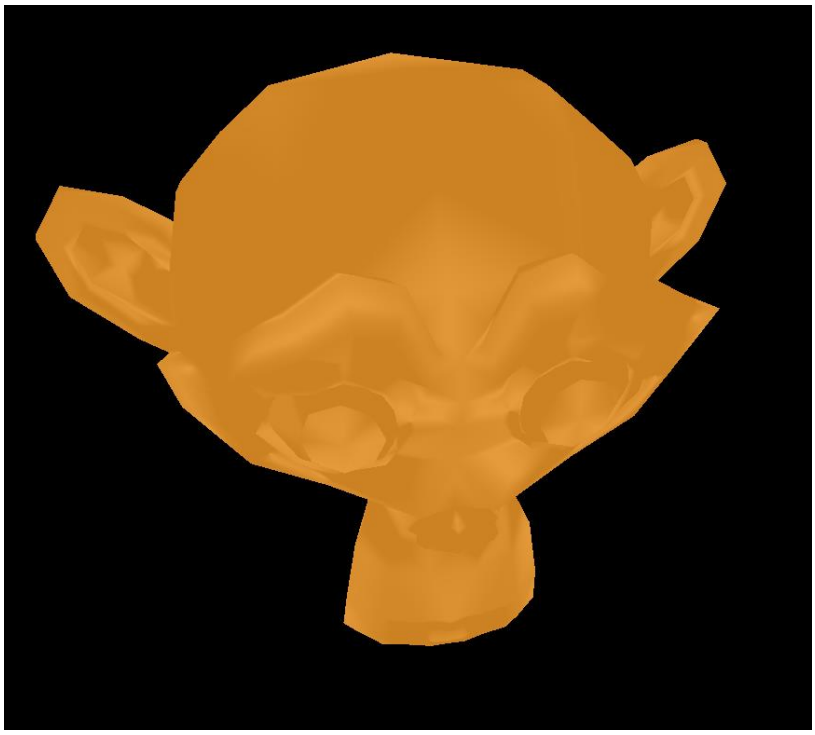
31. Graham Sellers, Richard S. Wright, Jr, Nicholas Haemel. 31.7.2015. Introduction. OpenGL Super Bible 7. painos, s. 3.

Esimerkki ladatusta skenaariosta

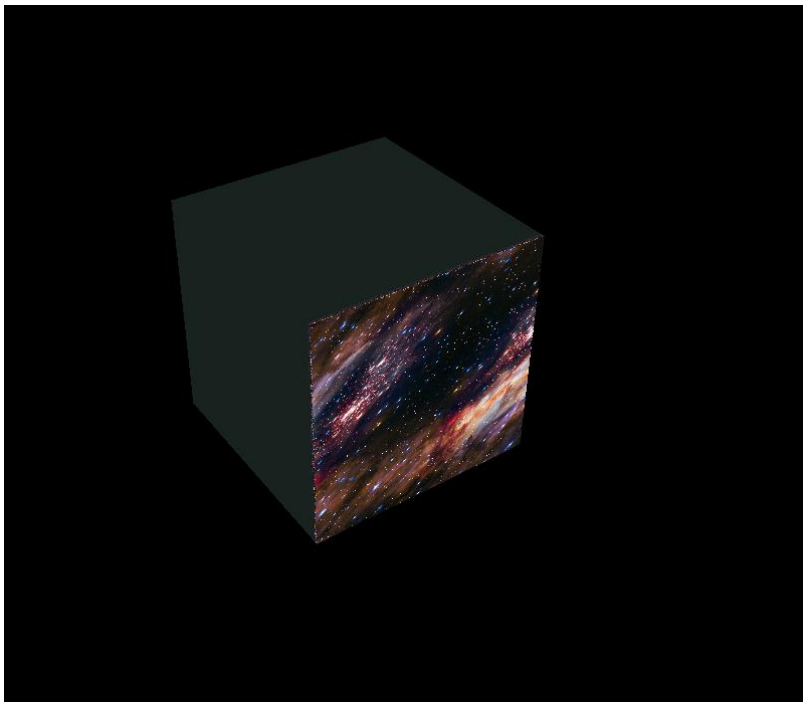


Kuva 21. Kuvassa skenaario low polygon –objekteja. Skenaariossa ei ole erillistä valonlähdettä.

Esimerkkejä sovelluksen ongelmakohdista



Kuva 22. Phong-valaistus ei toimi oikein ilmeisesti virheellisten normaalien takia.



Kuva 23. Kuva teksturoinnin ongelmasta, jossa tekstuuri piirtyy ainoastaan yhdelle pinnalle.