Teemu Turku

Roolipelidemo OpenGL:llä

Opinnäytetyö

Tietotekniikka / Peliohjelmointi

2019



|  |  |  |
| --- | --- | --- |
| **Tekijä/Tekijät** | **Tutkinto** | **Aika** |
| Teemu Turku | Tietotekniikan insinööri | Syyskuu 2019 |
| **Opinnäytetyön** **nimi** | | |
| Roolipelidemo OpenGL:llä. | | XX sivua  X liitesivua |
| **Toimeksiantaja** | | |
| Gamelab | | |
| **Ohjaaja** | | |
| Niina Mässeli | | |
| **Tiivistelmä** | | |
| Tässä opinnäytetyössä toteutetaan roolipeli demo OpenGL-ohjelmointi rajapintaa käyttäen. Demo sisältää | | |
| **Asiasanat** | | |
| dokumentointi, malli, opinnäytetyö, raportointi | | |

|  |  |  |  |
| --- | --- | --- | --- |
| **Author** **(authors)** | **Degree** | | **Time** |
| Teemu Turku | Bachelor of Information Technology | | September 2019 |
| **Thesis title** | | | |
| Role-playing game demo made in OpenGL | | XX pages  X pages of appendices | |
| **Commissioned by** | | | |
| Gamelab | | | |
| **Supervisor** | | | |
| Niina Mässeli | | | |
| **Abstract** | | | |
| Tämä on malli opinnäytetyön englanninkielisestä tiivistelmäsivusta. Jos kirjoitat työsi englanniksi, suomenna työsi otsikko suomenkieliseen tiivistelmään. Hyödynnä suomenkielisen tiivistelmän ohjetta ja huomioi lisäksi seuraavat asiat:  Tiivistelmän alkuun merkitään työn bibliografiset tiedot lomakkeen mukaisiin kohtiin malliesimerkkien kirjoitusasua mukaillen. Otsikon ensimmäinen sana kirjoitetaan isolla alkukirjaimella, muuten otsikossa käytetään pieniä kirjaimia. Ne otsikon sanat kirjoitetaan isolla, joissa kielioppisäännöt vaativat ison kirjaimen käyttöä.  Tiivistelmä kirjoitetaan esimerkiksi kolmeksi kappaleeksi. Ensimmäisessä kappaleessa voi lyhyesti selittää taustan, johon työn tekeminen liittyy. Työn tavoite (objective) tulee mainita. Työn toimeksiantajaa (principal tai commissioner) ei tarvitse tässä toistaa, koska se tulee tunnistetiedoissa esiin, ellei asiaan ole jotain oleellista lisättävää.  Tiivistelmäteksti etenee imperfektissä: *The objective of the thesis was to – –*. Vältä persoonapronominien (*I*, *we*...) käyttöä. Käytä niiden sijasta passiivia.  Toisessa kappaleessa tutkimuksen metodi tai projektin etenemisprosessi kuvaillaan suurpiirteisesti. Päivämääriä, kyselyjen palautusprosentteja tai muita yksityiskohtaisia tietoja ei ole syytä mainita, elleivät ne ole erityisen olennaisia. Myöskään tutkimuksenteon itsestäänselvyyksiä ei kuvailla (esim. *Some background reading of relevant literature was done).* Selvittää-verbi ei koskaan abstraktissa ole *clear up*, ja erittäin harvoin se on *clarify*.  Kolmanteen kappaleeseen tulee yhteenveto tuloksista ja johtopäätöksistä. Ne ovat olennainen osa abstraktia, sillä ne kertovat lukijalle, miksi koko tutkimus tai projekti kannatti tehdä. Lukijalle tulee yksiselitteisesti tehdä ero, mitkä ovat kirjoittajan omia tuloksia ja aikaansaannoksia, mitkä taas taustatietoa ja alalla yleisesti tunnettuja faktoja. Lopuksi tulee arvioida työn onnistumista tai kuvailla hyötyä toimeksiantajalle.  Kannattaa käyttää MOT-sanakirjaston Proofing-tyylintarkistustyökalua abstractin oikolukuprosessissa. | | | |
| **Keywords** | | | |
| documentation, model, thesis, report writing | | | |

**SISÄLLYS**

[1 johdanto 6](#_Toc532839602)

[2 opinnäytteen ulkoasu 6](#_Toc532839603)

[2.1 Kansi, sisällysluettelo, reunukset, sisennys, kirjainlajit ja -koot ja sivunumerointi 6](#_Toc532839604)

[2.2 Otsikot ja sisällysluettelo 8](#_Toc532839605)

[2.3 Kuvat ja taulukot 8](#_Toc532839606)

[2.4 Tekstin tehostus 11](#_Toc532839607)

[3 OTSIKKO 11](#_Toc532839608)

[LÄHTEET 12](#_Toc532839609)

KUVALUETTELO

LIITTEET

Liite 1. Kirjoitusalustan toiminta ja suositeltavat tyylit

Liite 2. Kaavojen merkintä

Liite 3. Kuva- tai taulukkoluettelo

# johdanto

Peliteollisuus on nopeasti kasvava viihdeteollisuuden ala, joten pelien kehitykseen tarkoitetut välineet ovat kehittyneet nopeasti. Nämä pelimoottorit ovat vuosikymmenien saatossa muuttuneet yhä helppokäyttöisemmiksi ja helposti saataviksi. Opinnäytetyössä perehdytään pelimoottorin ohjelmointiin, vaikka tavoitteena ja pääpisteenä työssä on toteuttaa toimiva pelidemo.

Tässä opinnäytetyössä toteutetaan kolmiulotteisen roolipelin kokeiluversio, jossa pelaaja pystyy liikuttamaan päähahmoa ja vuorovaikuttaa kentällä oleviin esineisiin. Opinnäytetyössä keskitytään tekemään toimiva ja pelattava peli Windows-alustoille. Iso osa työhön käytetystä ajasta menee alkukantaisen pelimoottorin ohjelmointiin.

# Käytetyt menetelmät

Erilaisia avoimen lähdekoodin ohjelmistokirjastoja löytyy internetistä tänä päivänä pilvin pimein. Vaihtoehtoja grafiikan piirtämiselle, äänien toistamiselle ja muillekin erilaisille toiminnoille löytyy runsaasti. Onkin siis tärkeää valita oikeat työkalut pelin tekoon.

## OpenGL

Graafisia ohjelmistokirjastoja tarjoavat mm. Microsoft DirectX, Vulkan, SFML, SDL sekä OpenGL. Projektiin valittiin OpenGL, sillä se soveltuu hyvin kolmiulotteisten pelien tekoon. OpenGL:stä löytyy myös todella kattava dokumentaatio, joka on edistänyt OpenGL:än suosiota. OpenGL on Silicon Graphics Inc:in vuonna 1992 julkaisema kieli- ja alustariippumaton ohjelmointirajapinta vektorigrafiikan piirtämiseen. Vuodesta 2006 eteenpäin OpenGL:ää on ylläpitänyt Khronos Group. (OpenGL, 2019.)

OpenGL ei tarjoa valmiita funktioita, vaan määritelmiä, joilla ohjataan OpenGL:n valtavaa tilakonetta. Nämä määritelmät kertovat ja määräävät eri funktioiden tulokset ja sen, kuinka nämä funktiot tulisi suorittaa. Näillä työkaluilla ohjelmoijien tehtäväksi jää toteuttaa nämä funktiot.

Vanhemmissa OpenGL versioissa lähestymistapa ohjelmoinnissa oli täysin erilainen kuin uudemmissa versioissa. Ennen OpenGL 3.2 versiota suurin osa toiminnallisuudesta oli piilotettu eikä ohjelmoijilla ollut mahdollisuutta säätää OpenGL:ään sisällettyjä laskutoimituksia. Vaikkakin tämä teki toiminnoista helppokäyttöisempiä ja paremmin ymmärrettäviä, tämä lähestymistapa ei ollut kovin tehokas. (Learn OpenGL, 2014.)

OpenGL 3.2 versiosta eteenpäin on ruvettu suosimaan core-profile-kehittämistapaa, jossa vanhentuneista toiminnallisuuksista hankkiuduttiin eroon. Vaikkakin tämä uusi kehitystapa voi olla hankalampi oppia, se myös pakottaa kehittäjän tutustumaan siihen, kuinka OpenGL ja sen funktiot oikeasti toimivat. Core-profile kehitystapa on myös paljon joustavampi, tehokkaampi ja tarjoaa käyttäjälle paremman ymmärryksen siitä, mitä pinnan alla tapahtuu. (emt.)

## GLFW

Ikkunan luomiseen on monia mahdollisuuksia, esimerkiksi The OpenGL Utility Toolkit (GLUT) sekä GL Frame Work (GLFW). GLUT:ia ei ole päivitetty vuosiin, mutta siitä on uudempi versio freeglut, joka saa päivityksiä vielä tänäkin päivänä. Projektissa päädyttiin käyttämään GLFW:tä, joka on ilmainen, avoimen lähdekoodin ohjelmakirjasto OpenGL-sovelluksen kehitykseen. Se tarjoaa yksinkertaisen, alustariippumattoman ohjelmointirajapinnan mm. ikkunan luomiseen, syötteen lukemiseen sekä tapahtumien hallintaan. (GLFW, 2019.)

## GLAD

Koska on olemassa paljon erilaisia ajureita erilaisiin grafiikkakortteihin, tarvitsevat OpenGL-ajurit tiedon OpenGL-funktioiden sijainnista ohjelman ajon aikana. Kehittäjän täytyy hakea funktion muistipaikka ja tallentaa se osoittimeen voidakseen käyttää tätä myöhemmin. Funktioiden määrän kasvaessa tämä voi käydä todella työlääksi. Onneksi tähän hommaan löytyy paljon erilaisia OpenGL latauskirjastoja, joiden tehtävänä on automaattisesti selvittää oikean funktion muistipaikka sillä hetkellä olevaan alustaan sopivaksi. Päädyin käyttämään LearnOpenGL-opetus-sivustolla käytettyä glad:ia, sillä se vaikutti yksinkertaiselta sekä helppokäyttöiseltä. (OpenGL Wiki, 2019.)

## GLSL

OpenGL Shading Language (GLSL) on ohjelmointikieli, jolla ohjelmoidaan varjostinohjelmia (eng. shader). Varjostinohjelmat ovat todella tärkeä osa OpenGL:än käyttöä, sillä ilman niitä ei ruudulle saa piirrettyä mitään. Ruutu on mahdollista tyhjentää ilman varjostinohjelmaa, mutta siihen se sitten jääkin. (Shreiner, Sellers, Kessenich & Licea-Kane, 2013, 34-35.)

## GLM

Kolmiulotteisessa pelissä tarvitaan paljon vektori sekä matriisimatematiikkaa. Onneksi näiden laskutoimitusten helpotukseksi löytyy OpenGL Mathematics (GLM) matematiikkakirjasto. GLM on otsikkotiedosto, joka tarvitsee vain sisällyttää ohjelmaan toimiakseen. GLM:ssä käytetään samoja nimeämistapoja kuin GLSL:ssä, joten jos varjostinkielen erilaiset muuttujatyypit ovat tuttuja, on GLM:än käyttö helpompaa. GLM sisältää liudan luokkia ja funktioita, jotka helpottavat matriisien ja vektorien kanssa työskentelyä. (OpenGL Mathematics, 2019.)

## OpenAL

Äänentoistoon tarkoitettuja ohjelmistokirjastoja löytyy valtava määrä. Opinnäytetyöhön valittiin OpenAL äänentoistoon, ja OpenALUT äänitiedostojen lukemiseen ja lataamiseen. OpenAL:än toiminta muistuttaa hieman OpenGL:ää siten, että molemmat ovat hieman kuin valtavia tilakoneita, joita ohjataan vaihtelemalla tilojen arvoja. (OpenAL, 2005.)

## FreeType

FreeType on C-kielellä kirjoitettu ilmainen ohjelmakirjasto helpottamaan fonttien ja tekstin lataamista ja piirtämistä (FreeType, 2018). Teksti on mahdollista myös renderöidä tekstuurina kolmiulotteiselle pinnalle. Tämä on nopea, helppo ja vielä tänäkin päivänä suosittu tapa, mutta ei kovinkaan joustava. Jos fonttia haluaa vaihtaa, on uusi bittikarttafontti käännettävä kokonaan uudestaan ja vain yksi resoluutio on tuettu, eli zoomaaminen paljastaa pikselöidyt kulmat. (Learn OpenGL, 2014.)

## MinGW

Pelin kääntämiseen ja lopullisen suoritettavan tiedoston luomiseen käytettiin opinnäytetyössä Minimalist GNU for Windowsia eli MinGW:tä. MinGW on avoimen lähdekoodin ohjelmointityökalukokoelma, jolla on mahdollista muuttaa kirjoitettu C- tai C++-kielen koodi tietokoneen ymmärrettäväksi konekoodiksi Windows-käyttöjärjestelmillä. (Mingw, 2019.)

## Versionhallinta

Versionhallinta on järjestelmä, joka tallentaa projektista ja sen tiedostoista tiedot tapahtuneista muutoksista. Näin on mahdollista palata projektissa aiempaan versioon ja palauttaa poistetut tai vanhemmat versiot tiedostoista (Git, 2019). Versionhallinta mahdollistaa monen ihmisen samanaikaisen työskentelyn saman projektin parissa, mutta on myös hyödyllinen henkilökohtaisissa yhden hengen projekteissa. Versionhallintaohjelmaksi valittiin Git sekä Github. Git on avoimen lähdekoodin versionhallintajärjestelmä, jolla projektiin tehdyt muutokset saadaan seurantaan (emt.). Github puolestaan tarjoaa alustan, jolla on helppoa ja yksinkertaista hallita Git-projekteja (Github, 2019).

# Suunnittelu

Pelin idea syntyi Playstation 1 ajan roolipelien innoittamana. Työssä pyritään luomaan 1990-luvun loppupuolen roolipeli hyödyntäen samankaltaisia tekniikoita, kuin mitä tuohon aikaan käytettiin. Pelikentällä on kolmiulotteinen pelihahmo, jota pelaaja pystyy liikuttamaan etukäteen renderöidyllä taustalla.

## Pelin eri tilat

Peliin on tarkoitus tehdä neljä erilaista tilaa: Kenttä, Taistelu, Päävalikko sekä Valikko-tilat. Pelissä tilakone säätelee senhetkisen tilan mukaan, mitä näytölle piirretään ja mitä käyttäjän antamat syötteet tekevät pelissä.

### Kenttä

Kenttä-tilassa näytetään pelaajalle pelinäkymä, jossa pelaajahahmoa voi liikutella pelimaailmassa. Kentällä pelaaja pystyy myös vaikuttamaan kentällä oleviin esineisiin ja henkilöihin, esimerkiksi avaamaan aarrearkkuja sekä juttelemaan ei-pelaaja-hahmoille. Kentistä pääsee toisiin kenttiin liikkumalla kentällä tiettyyn kohtaan, esimerkiksi talon oviaukolle, jolloin pääsee taloon sisään. Tällöin kenttämanageriluokalle kerrotaan, että siirrytään kentästä toiseen, jolloin taustakuva vaihdetaan, poistetaan näkymästä edellisen kentän 3d-mallit sekä ladataan kentälle uuden kentän mallit. Kentässä pelaaja voi myös avata Valikon painamalla valikko-näppäintä.

### Maailmankartta

Maailmankartalla pelaaja pystyy liikkumaan kaupungista toiseen ja eri kenttien välillä. Myös valikon saa auki kartalla painamalla valikko-näppäintä. Maailmankartta toimii melko lailla samoin kuin kenttä, mutta 3d-mallit kartalla ovat pienempiä eivätkä niin yksityiskohtaisia, jotta voidaan piirtää kartalle esimerkiksi paljon puita metsiin ja muita erilaisia malleja.

### Taistelu

Taistelutila alkaa animaatiolla, jossa kamera lentää kentälle näkymään, jossa näkyvät kaikki taistelussa olevat 3d-mallit. Taistelu on vuoropohjainen, ja aluksi pelaaja päättää vuoro perään, mitä kukin pelihahmo tekee. Kun päätökset on tehty, komentoja toteutetaan sen mukaan, minkä hahmon nopeus on suurin.

### Päävalikko

Päävalikossa on neljä eri painiketta: New Game-, Continue-, Options-, sekä Exit Game-painikkeet. New Game aloittaa pelin tarinan alusta ja lataa ensimmäisen kentän. Continue-painikkeesta pääsee tarkastelemaan tallennuksia, sekä jatkamaan peliä tallennuksesta, jolloin peli lataa tallennuksesta oikeat tiedot hahmoille sekä oikean kentän. Options-painike piilottaa ruudulla näkyvät painikkeet ja näyttää Options-valikon, jossa pelaaja voi säätää musiikin sekä äänieffektien voimakkuutta, muuttaa pelin resoluutiota tai palata takaisin Options-valikosta Päävalikkoon. Exit Game-painike sulkee pelin, ja palauttaa pelaajan työpöydälle.

### Valikko

Valikkoon pääsee pelissä kentällä sekä maailmankartalla painamalla valikko-näppäintä. Valikossa ruudun vasemmalla puolella ovat painikkeet ja ruudun oikealla puolella näkyy aktiivisten hahmojen elämät sekä taikapisteet. Valikossa on Item-, Magic-, Status-, Options- sekä Quit -painikkeet. Item-painikkeesta pääsee tarkastelemaan saatuja esineitä ja varusteita. Magic-painike avaa listan hahmojen osaamista taioista, joita pystyy valikossa käyttämään, esimerkiksi Heal-taikaa voi käyttää valikossa saadakseen elämää hahmoille takaisin. Status-painikkeesta pääsee tutkailemaan eri hahmojen statteja, esimerkiksi kuinka paljon hyökkäys- tai puolustusvoimaa hahmoilla on. Options-painike toimii samoin kuin päävalikossa, eli pääsee säätämään pelin asetuksia. Quit-painike palauttaa pelin takaisin päävalikkoon.

## Pelin kentät

Opinnäytetyössä on yksi kenttä,

# Toteutus

## Tavoitteet

Pelidemon toteutusta varten on ensin luotava perusta, jonka päälle peli rakennetaan. Tarvitaan pelimoottori, joka pystyy piirtämään kolmiulotteista vektorigrafiikkaa. Kolmiulotteiset mallit on luotava jollakin toisella ohjelmalla, sillä olisi todella työlästä määritellä joka ikisen vektoripisteen sijainti OpenGL:lle, varsinkin kun yksinkertaisissakin malleissa voi vektoripisteitä olla satoja, tai jopa tuhansia. Tähän tarvitaan koodia, jolla saadaan ladattua mallinnusohjelmalla luotu malli. On myös kirjoitettava varjostinohjelmat, jotka määrittelevät, kuinka malli piirtyy näytölle. Varjostinohjelmat täytyy luoda ja ne täytyy linkittää OpenGL:lle. Kuvia pitää myös pystyä lataamaan, jotta niitä voidaan käyttää tekstuureissa.

## Pelimaailman piirtäminen näytölle

### Ikkunan luonti

Kuvassa 1 näkyy mahdollisimman yksinkertaisen ikkunan luonti GLFW:ää käyttäen. Aluksi GLFW alustetaan, ja sille kerrotaan, mitä OpenGL-versiota käytetään ja että käytetään core-profiilia glfwWindowHint-funktiolla. Tämän jälkeen luodaan itse ikkuna glfwCreateWindow-funktiolla, jolle annetaan parametreiksi ikkunan leveys, korkeus, ikkunan otsikkoteksti. Toiseksi viimeinen parametri kertoo, mitä ruutua käytetään jos halutaan peli koko näytölle, joten NULL tarkoittaa että ikkuna ei aukea kokonäytön tilaan. Viimeisellä parametrilla kerrotaan, halutaanko että ikkuna jakaa resursseja toisen ikkunan kanssa, NULL tässä tarkoittaa ettei resursseja jaeta. Tämän jälkeen varmistetaan, että ikkuna on luotu onnistuneesti. Jos näin ei ole, ohjelma suljetaan ja konsoliin tulostetaan virheteksti. Sitten alustetaan GLAD, joka huolehtii että OpenGL-funktiot toimivat eri näytönohjaimien ajureilla. OpenGL:lle välitetään tieto ikkunan koosta glViewport-funktiolla. Sitten päästään piirtosilmukkaan, joka pyörii kunnes käyttäjä sulkee ikkunan. Ikkunan sulkemisen jälkeen kutsutaan glfwTerminate-funktiota, joka tuhoaa vielä auki olevat ikkunat sekä vapauttaa varatun muistin. (Learn OpenGL, 2014)

    // Initialize GLFW and configure

    glfwInit();

    glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MAJOR, 4);

    glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MINOR, 6);

    glfwWindowHint(GLFW\_OPENGL\_PROFILE, GLFW\_OPENGL\_CORE\_PROFILE);

    // Create the window and set the width and height

    GLFWwindow\* window = glfwCreateWindow(SCREENWIDTH, SCREENHEIGHT, "OpenGLRPG", NULL, NULL);

    if (window == NULL)

    {

        std::cout << "Failed to create GLFW window" << std::endl;

        glfwTerminate();

        return -1;

    }

    glfwMakeContextCurrent(window);

    // Initialize GLAD with OS-specific pointers so the glfwGetProcAddress defines the correct

    // function based on which OS we're compiling for.

    if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress))

    {

        std::cout << "Failed to initialize GLAD" << std::endl;

        return -1;

    }

    // Tell OpenGL the size of the rendering window

    glViewport(0, 0, SCREENWIDTH, SCREENHEIGHT);

    // Tell GLFW to call FramebufferSizeCallback on every window resize.

    glfwSetFramebufferSizeCallback(window, FramebufferSizeCallback);

    // Rendering loop

    while(!glfwWindowShouldClose(window))

    {

        glfwSwapBuffers(window);

        glfwPollEvents();

    }

    glfwTerminate();

    return 0;

Kuva 1. Ikkunan luonti GLFW:llä

### Varjostinohjelma

Varjostinohjelma on koodia, joka kertoo OpenGL:än ”grafiikkaliukuhihnalle” *(*engl. rendering pipeline*)* mitä näytölle piirretään. Varjostinohjelmalle annetaan liuta 3D koordinaatteja, jotka OpenGL prosessoi ja muuttaa väritetyiksi 2D pikseleiksi ruudulle näytettäväksi. OpenGL-varjostimet kirjoitetaan OpenGL Shading Language-kielellä (GLSL). Erilaisia GLSL-varjostinohjelmatyyppejä ovat verteksivarjostin (engl. vertex shader), sirpalevarjostin (engl. fragment shader) sekä geometriavarjostin (engl. geometry shader). Verteksivarjostin sekä sirpalevarjostin ovat molemmat pakollisia OpenGL:än osia piirtoa varten, geometriavarjostin taas ei. (Learn OpenGL, 2014)

Piirtoprosessin alussa verteksivarjostimelle annetaan lista pisteitä, joista piirrettävä kuvio muodostuu. Näiden pisteiden kuuluu määritellä kolmio, jonka väliin väriä halutaan. Kolmioita määritellään ja muodostetaan sen verran, että niistä saadaan kasattua kappale, joka halutaan piirtää. Kun verteksivarjostin on muodostanut kolmion, annetaan syöte eteenpäin geometriavarjostimelle, jossa pisteitä on vielä mahdollista muokata ja muodostaa lisää. Geometriavarjostin antaa pisteet eteenpäin rasterointivaiheeseen, jossa kolmiot määritellään ruudulle oikeisiin kohtiin, jolloin sirpalevarjostin saa tiedon sirpaleista käyttöönsä. Ennen kuin sirpalevarjostin tekee tehtävänsä, tapahtuu leikkaus, jolloin hävitetään sirpaleet, jotka eivät ole näkökentässä. Sirpalevarjostimessa lasketaan pikseleiden lopulliset värit ja lasketaan esimerkiksi valojen vaikutus malliin. (emt.)



Kuva 2. OpenGL:n piirtoprosessi (Learn OpenGL, 2014)

GLSL on hyvin samantapaista C-kielen kanssa, ja siihen on myös otettu hieman osia C++:sta. GLSL-varjostimissa on mahdollista käyttää kuvan 3 mukaisia tietotyyppejä. Varjostimissa on myös mahdollista antaa muuttujille kuvan 4 mukaisia määritteitä, joilla kerrotaan esimerkiksi, onko muuttuja syöte vai tuloste.



Kuva 3. GLSL:n perus-, vektori sekä matriisitietotyypit (Shreiner et al. 2013, 40)



Kuva 4. GLSL:n tyyppimääritteet (Shreiner et al. 2013, 46)

Opinnäytetyössä verteksivarjostinohjelmat on kirjoitettu vs-päätteisiin (vertex shader) tiedostoihin, ja sirpalevarjostinohjelmat fs-päätteisiin (fragment shader) tiedostoihin. Yksinkertaisimmillaan verteksi- ja sirpalevarjostimet näyttävät kuvan 3 ja 4 mukaisilta. Verteksivarjostimeen annetaan pisteet kolmiulotteisessa koordinaatistossa, ja sirpalevarjostimeen annetaan väriarvo, jolla halutaan sirpaleet värjätä.

// Shader with color and only positions.

#version 460 core

layout (location = 0) in vec3 aPos;

void main()

{

    gl\_Position = vec4(aPos, 1.0);

}

Kuva 5. Yksinkertainen verteksivarjostin

// Shader with color and only positions.

#version 460 core

out vec4 FragColor;

uniform vec4 myColor;

void main()

{

    FragColor = vec4(myColor);

}

Kuva 6. Yksinkertainen sirpalevarjostin

Varjostimet täytyy myös kääntää ja luoda varjostinohjelma, johon nämä varjostimet kiinnitetään ja joka linkitetään OpenGL:lle tämän käyttöön. Shader-luokan muodostimessa aluksi luetaan fs- ja vs-tiedostoissa olevat varjostimet char-tyypin muuttujaan myöhempään käyttöön. Varjostimen luonti tapahtuu kuvan 5 esittämällä tavalla. Varjostimet luodaan glCreateShader-funktiolla, jonka palauttama tunnistinnumero tallennetaan kokonaislukuun. Tämän jälkeen aiemmin luettu varjostinkoodi kiinnitetään itse varjostimeen glShaderSource-funktiolla. Sitten varjostin vielä käännetään glCompileShader-funktiolla. Tämän jälkeen on viisasta tarkistaa, että kääntäminen onnistui ja ilmoittaa, jos kääntämisen aikana on tapahtunut virhe. Verteksivarjostimen sekä sirpalevarjostimen luominen tapahtuu suurimmaksi osin samalla tavalla ja samoilla funktioilla. (Learn OpenGL, 2014)

unsigned int vertexShader;

vertexShader = glCreateShader(GL\_VERTEX\_SHADER);

glShaderSource(vertexShader, 1, &vertexShaderSource, NULL);

glCompileShader(vertexShader);

glGetShaderiv(vertexShader, GL\_COMPILE\_STATUS, &success);

if(!success)

{

     glGetShaderInfoLog(vertex, 512, NULL, infoLog);

std::cerr << "ERROR!! Vertex shader compiltion failed!\n" << infoLog << std::endl;

}

Kuva 7. Varjostimen luonti

Kun varjostimet on luotu, täytyy vielä luoda varjostinohjelma, joka on viimeinen linkitetty versio yhdistetyistä varjostimista. Tämä prosessi näkyy kuvassa 6. Varjostinohjelma luodaan funktiolla glCreateProgram, joka palauttaa tunnistinnumeron. Aiemmin luodut varjostimet kiinnitetään ohjelmaan funktiolla glAttachShader, ja linkitetään funktiolla glLinkProgram. On hyvä tarkistaa, tapahtuiko linkityksessä virheitä. Tämän jälkeen varjostinohjelma on valmis käytettäväksi, joka tapahtuu funktiolla glUseProgram. Kun varjostimet on linkitetty ja varjostinohjelma on valmiina käytettäväksi, voidaan itse varjostinobjektit poistaa funktiolla glDeleteShader. (emt.)

// Create shader program and assign the ID to the integer.

unsigned int shaderProgram;

shaderProgram = glCreateProgram();

// Attach and link shaders.

glAttachShader(shaderProgram, vertexShader);

glAttachShader(shaderProgram, fragmentShader);

glLinkProgram(shaderProgram);

    // Check errors.

    glGetProgramiv(shaderProgram, GL\_LINK\_STATUS, &success);

    if(!success) {

        glGetProgramInfoLog(shaderProgram, 512, NULL, infoLog);

        std::cerr << "ERROR!! Shader program linking failed!\n" << infoLog << std::endl;

    }

    // Activate shader.

glUseProgram(shaderProgram);

    // Delete shader objects.

    glDeleteShader(vertexShader);

    glDeleteShader(fragmentShader);

Kuva 8. Varjostinohjelman luonti

Varjostinohjelman luokka sisältää myös paljon erilaisia asettajafunktioita, kuten esimerkiksi SetBool(const std::string &name, bool value) sekä SetInt(const std::string &name, int value), joille annetaan parametreiksi muutujan nimi varjostimessa, sekä uusi arvo jonka koodi asettaa glad-ohjelmistokirjaston funktiolla glUniform1i. Funktiossa nimi annetaan glGetUniformLocation-funktioon sen hetkisen varjostimen tunnistinluvun kanssa, jolloin tämä funktio palauttaa oikean indeksin OpenGL:n sisäisesti luodusta taulukosta glUniform1i-funktion käyttöön. Kuvassa 7 näkyy esimerkki kokonaisluvun asettamisesta varjostimeen. Glad sisältää funktiot eri tietotyyppien asettamiseen, esimerkiksi liukuluvun asettaminen tapahtuu funktiolla glUniform1f ja 4x4-matriisi asetetaan funktiolla glUniformMatrix4fv. (Shreiner et al. 2013, 46-48.)

void Shader::SetInt(const std::string &name, int value) const

{

    glUniform1i(glGetUniformLocation(ID, name.c\_str()), value);

}

Kuva 9. Kokonaisluvun asettaminen Shader-luokassa

### Suorakulmion piirtäminen

Jotta OpenGL:llä saadaan malli piirrettyä, täytyy OpenGL:lle antaa tieto pisteiden sijainneista syötteenä. Tämä tieto annetaan liukulukutaulukkona, jossa tieto pisteistä on yleensä kuvan 10 mukaisesti. Jokaisella rivillä on tieto esimerkiksi pisteen sijainnista, tekstuurikoordinaateista sekä normaalien koordinaateista. OpenGL käsittelee verteksien koordinaatteja normalisoituna laitekoordinaattivälillä, joka tarkoittaa, että piste (-1.0f, -1.0f, 0.0f) sijaitsee ruudun vasemmassa alareunassa ja piste (1.0f, 1.0f, 0.0f) taas ruudun oikeassa yläreunassa. (Learn OpenGL, 2013)

float vertices[] = {

    //  x,     y,    z,    Positions

    -0.5f, -0.5f, 0.0f, // Vertex 1

     0.5f, -0.5f, 0.0f, // Vertex 2

     0.0f,  0.5f, 0.0f  // Vertex 3

};

Kuva 10. Kolmion verteksit



Kuva 11. Kolmio normalisoiduilla laitekoordinaateilla. (Learn OpenGL, 2013)

Kun mallin koordinaatit ovat määritelty liukulukutaulukossa, täytyy grafiikkakortista varata muistia näitä verteksitietoja varten ja tallettaa tiedot verteksipuskuriobjectiin (engl. Vertex buffer object). Verteksipuskuriobjekti on tehokas, sillä se pystyy säilömään suuren määrän verteksejä, ja sen avulla on mahdollista lähettää grafiikkakortille suuri määrä verteksejä yhdellä kertaa. Tiedon lähettäminen grafiikkakortille prosessorilta on melko hidasta, joten on suositeltavaa koittaa saada mahdollisimman suuri määrä dataa siirrettyä kerralla. (emt.)

Puskuriobjektin luominen tapahtuu OpenGL:ssä funktiolla glGenBuffers, joka luo sille annetun kokonaisluvun verran puskureita käyttöön. Puskuriobjektiin kiinnitetään luotu puskuriobjekti funktiolla glBindBuffer, jonka jälkeen lähetetään aiemmin määrittelemän kolmion verteksit puskuriin funktiolla glBufferData. OpenGL:lle pitää vielä kertoa, kuinka verteksidataa käytetään funktiolla glVertexAttribPointer jokaista eri tietoa kohden. Eli jos kolmion liukulukutaulukossa on määritelty verteksien sijainti sekä tekstuurikoordinaatti, pitää glVertexAttribPointer-funktiota kutsua kahdesti oikeilla arvoilla ja tiedoilla. Funktiolle annetaan ensimmäisenä parametrina verteksivarjostimessa määritelty sijainti. Toisena parametrina kerrotaan verteksitiedon koko, eli kun ollaan lähettämässä kolmiulotteisen pisteen sijaintia, on tämä luku kolme. Seuraavana kerrotaan, mitä tietotyyppiä verteksitieto on ja halutaanko tieto normalisoida. Toiseksi viimeinen parametri kertoo, kuinka suuri väli on verteksitietojen välillä. Viimeinen parametri kertoo kuinka suuri poikkeama on tiedon ja puskurin ensimmäisen muistipaikan välillä.

Verteksitaulukko-objekti on myös pakollinen OpenGL:ssä, ja se luodaan samalla tavoin kuin verteksipuskuriobjekti. Verteksitaulukko-objekti sisältää verteksitiedot glVertexAttribPointer-funktion kutsuista, ja sen avulla on mahdollista nopeasti vaihtaa verteksitietoja. Kun myös verteksitaulukko-objekti on valmis, voidaan kolmio piirtää ruudulle funktiolla glDrawArrays. Funktiolle kerrotaan, että halutaan piirtää kolmio antamalla parametri GL\_TRIANGLES, ja ilmoitetaan, että aloittava indeksi on nolla ja verteksejä on yhteensä kolme. Koko prosessi näkyy kuvassa 12.

    unsigned int VAO;

    glGenVertexArrays(1, &VAO);

    unsigned int VBO;

    glGenBuffers(1, &VBO);

    glBindVertexArray(VAO);

    glBindBuffer(GL\_ARRAY\_BUFFER, VBO);

glBufferData(GL\_ARRAY\_BUFFER, sizeof(vertices), vertices, GL\_STATIC\_DRAW);

glVertexAttribPointer(0, 3, GL\_FLOAT, GL\_FALSE, 3 \* sizeof(float), (void\*)0);

glEnableVertexAttribArray(0);

// Unbind buffer and VAO.

    glBindBuffer(GL\_ARRAY\_BUFFER, 0);

glBindVertexArray(0);

// Drawing

    glUseProgram(shaderProgram);

    glBindVertexArray(VAO);

    glDrawArrays(GL\_TRIANGLES, 0, 3);

Kuva 12. Verteksipuskuriobjektin luominen ja tiedon lähettäminen sekä kolmion piirtäminen.

On myös olemassa elementtipuskuriobjekti, jonka avulla saadaan piirrettyä neliö tai suorakulmio paljon tehokkaammin. Ilman elementtipuskuriobjektia suorakulmio täytyisi määritellä kahden kolmion eli kuuden verteksin avulla. Elementtipuskuriobjekti mahdollistaa suorakulmion piirron käyttämällä vain neljää verteksiä, sekä kokonaislukutaulukkoa joka kertoo verteksien piirtojärjestyksen. Esimerkki elementtipuskuriobjektin käytöstä löytyy kuvasta 13. (Learn OpenGL, 2013)

float vertices[] =

    { // Positions

        0.5f,  0.5f, 0.0f,  // top right

        0.5f, -0.5f, 0.0f,  // bottom right

        -0.5f, -0.5f, 0.0f,  // bottom left

        -0.5f,  0.5f, 0.0f   // top left

    };

    unsigned int indices[] =

    {

        // The order to draw the vertices

        0, 1, 3,   // first triangle

        1, 2, 3    // second triangle

    };

    // Creating the element buffer object

    unsigned int EBO;

    glGenBuffers(1, &EBO);

    glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, EBO);

    glBufferData(GL\_ELEMENT\_ARRAY\_BUFFER, sizeof(indices), indices, GL\_STATIC\_DRAW);

    // Drawing

    glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, EBO);

    glDrawElements(GL\_TRIANGLES, 6, GL\_UNSIGNED\_INT, 0);

Kuva 13. Elementtipuskuriobjektin käyttö

Opinnäytetyössä jokaisella piirrettävällä mallilla ja peliobjektilla on oma verteksipuskuriobjekti, verteksitaulukko-objekti, sekä elementtipuskuriobjekti. Jokaisella piirrettävällä objektilla on myös Draw-funktio, joka hoitelee piirtoon liittyvät prosessit. Näin on helpompi säilöä kentälle piirrettävät objektit taulukkoon, ja kutsua kaikkien piirrettävien objektien piirtofunktiota yhdessä loopissa.

### Tekstuurit

Tekstuurii

## Kameran määrittely

Äänentoistoon tarkoitettuja ohjelmistokirjastoja löytyy valtava määrä.

## 3d-mallien lataaminen

Äänentoistoon tarkoitettuja ohjelmistokirjastoja löytyy valtava määrä.

## Liikkumisen ja interaktiivisuuden lisääminen peliin

Äänentoistoon tarkoitettuja ohjelmistokirjastoja löytyy valtava määrä.

## Pelin luokat

### Camera-luokka

Äänentoistoon tarkoitettuja ohjelmistokirjastoja löytyy valtava määrä.

### Shader-luokka

Äänentoistoon tarkoitettuja ohjelmistokirjastoja löytyy valtava määrä.

### Model-luokka

Äänentoistoon tarkoitettuja ohjelmistokirjastoja löytyy valtava määrä.

### Stats-luokka

Äänentoistoon tarkoitettuja ohjelmistokirjastoja löytyy valtava määrä.

## Äänet

Äänentoistoon tarkoitettuja ohjelmistokirjastoja löytyy valtava määrä.

# Yhteenveto

Koska tieteelliselle tekstille on tyypillistä, että lukuja on vähintään kolme, on tässä mallipohjassakin lukuja kolme. Otsikon 3 alle kirjoitetaan tekstikappaleita tai tehdään alaluku, jonka alle kirjoitetaan tekstikappaleita.

LÄHTEET

FreeType (2018). *What is FreeType?* FreeTypen www-sivusto. Saatavissa: <https://www.freetype.org/freetype2/docs/index.html> [viitattu 28.10.2019].

Git. (2019). *Git - About Version Control*. Gitin www-sivusto. Saatavissa: <https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control> [viitattu 22.10.2019].

Github. (2019). *Hello World · GitHub Guides*. Githubin www-sivusto. Saatavissa: https://guides.github.com/activities/hello-world/ [viitattu 22.10.2019]

GLFW. (2019). GLFW:n www-sivusto. Saatavissa: https://www.glfw.org/ [viitattu 24.10.2019]

GLM. (2019). *OpenGL Mathematics.* OpenGL Mathematicsin www-sivusto. Saatavissa: https://glm.g-truc.net/0.9.9/index.html [viitattu 17.10.2019]

OpenGL. (2019). *OpenGL Overview* OpenGL:n www-sivusto. Saatavissa: https://www.opengl.org/about/ [viitattu 28.10.2019]

OpenGL Wiki. (2019). *OpenGL Loading Library* OpenGL-wikin www-sivusto. Saatavissa: https://www.khronos.org/opengl/wiki/OpenGL\_Loading\_Library [viitattu 24.10.2019]

Learn OpenGL (2014). *Core-profile vs Immediate mode.* Learn OpenGL:än www-sivusto. Saatavissa: https://learnopengl.com/Getting-started/OpenGL [viitattu 11.10.2019]

Learn OpenGL (2014). *Hello Triangle.* Learn OpenGL:än www-sivusto. Saatavissa: https://learnopengl.com/Getting-started/Hello-Triangle [viitattu 31.10.2019]

Learn OpenGL (2014). *Hello Window.* Learn OpenGL:än www-sivusto. Saatavissa: https://learnopengl.com/Getting-started/Hello-Window [viitattu 28.10.2019]

Learn OpenGL (2014). *Text Rendering.* Learn OpenGL:än www-sivusto. Saatavissa: <https://learnopengl.com/In-Practice/Text-Rendering> [viitattu 31.10.2019]

Mingw. (2019). *Minimalist GNU for Windows*. MinGW:n www-sivusto. Saatavissa: http://www.mingw.org/ [viitattu 18.10.2019]

OpenAL. (2005). *OpenAL 1.1 Specification and Reference.* OpenAL-dokumentaatio. Saatavissa: <https://openal.org/documentation/openal-1.1-specification.pdf> [viitattu 22.10.2019]

Shreiner, D., Sellers, G., Kessenich, J. ja Licea-Kane, B. (2013). *OpenGL Programming Guide*. 8. painos. Boston: Addison-Wesley.

Kiero-Hiiri, P. 2019. Tutkimustyö ja etikka. 2., uudistettu painos. Helsinki: Wanabii Sanomatalo.

Norrbacka, S. 2016. ”Kyllä ne kultatuolissa meitä vielä kantavat ja samppakaljojakin peda-joryssä ruiskutellaan” – Amk-raportointiohjeiden uhmakas voittokulku 2008–2016. Jyväskylän yliopisto. Yhteisöhistorian laitos. Väitöskirja.

Norrbacka, S. 2019. Amk-raportointiohjeet ovat aina taakkanas. Kotka: Armi Airlines.

Pohjamuta, L.-A. & Äkämä-Haara, U. 2017. Polkuopiskelu ja -anturat. *Anatomia ja minä* 5, 13–15.