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 roolipelidemo OpenGL–ohjelmointirajapintaa käyttäen. Työssä keskitytään selvittämään, kuinka luodaan mahdollisimman yksinkertainen pelimoottori pienen pelin testaamiseen.  Opinnäytetyön alussa selvitetään, mitä eri ohjelmistokirjastoja ja ohjelmia työssä käytetään. Työssä kerrotaan pelin suunnittelusta…JOTAIN TÄHÄN  Opinnäytetyössä toteutetaan pelimoottoriin grafiikan piirto, käyttäjän syötteen tunnistus, äänien toistaminen sekä omien kolmiulotteisten mallien lataaminen. Työssä kerrotaan, kuinka kamera määritellään kolmiulotteiseen maailmaan, ja kuinka model–, view– ja projection matriiseja käytetään kolmiulotteisen maailman näyttämiseen.  Pelidemo sisältää yhden kentän, jossa pelaaja pystyy liikuttamaan pelaajahahmoa ja juttelemaan pelimaailmassa seisovan ei-pelaaja-hahmon kanssa. | | |
| **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](#_Toc25173965)

[2 Käytetyt menetelmät 6](#_Toc25173966)

[2.1 OpenGL 6](#_Toc25173967)

[2.2 GLFW 7](#_Toc25173968)

[2.3 GLAD 7](#_Toc25173969)

[2.4 GLSL 8](#_Toc25173970)

[2.5 GLM 8](#_Toc25173971)

[2.6 OpenAL 8](#_Toc25173972)

[2.7 FreeType 8](#_Toc25173973)

[2.8 stb\_image.h 9](#_Toc25173974)

[2.9 Assimp 9](#_Toc25173975)

[2.10 MinGW 9](#_Toc25173976)

[2.11 Visual Studio Code 9](#_Toc25173977)

[2.12 Blender 9](#_Toc25173978)

[2.13 Versionhallinta 9](#_Toc25173979)

[3 Suunnittelu 10](#_Toc25173980)

[3.1 Pelin eri tilat 10](#_Toc25173981)

[3.1.1 Kenttä 10](#_Toc25173982)

[3.1.2 Maailmankartta 11](#_Toc25173983)

[3.1.3 Taistelu 11](#_Toc25173984)

[3.1.4 Päävalikko 11](#_Toc25173985)

[3.1.5 Valikko 11](#_Toc25173986)

[3.2 Pelin kentät 12](#_Toc25173987)

[4 Toteutus 12](#_Toc25173988)

[4.1 Tavoitteet 12](#_Toc25173989)

[4.2 Pelimaailman piirtäminen näytölle 12](#_Toc25173990)

[4.2.1 Ikkunan luonti 12](#_Toc25173991)

[4.2.2 Varjostinohjelma 14](#_Toc25173992)

[4.2.3 Suorakulmion piirtäminen 18](#_Toc25173993)

[4.2.4 Tekstuurit 20](#_Toc25173994)

[4.3 Kameran määrittely 24](#_Toc25173995)

[4.4 3d–mallien lataaminen 29](#_Toc25173996)

[4.5 Renderer–luokka 32](#_Toc25173997)

[4.6 Liikkumisen ja interaktiivisuuden lisääminen peliin 33](#_Toc25173998)

[4.7 Pelin luokat 33](#_Toc25173999)

[4.7.1 Camera–luokka 33](#_Toc25174000)

[4.7.2 Shader–luokka 33](#_Toc25174001)

[4.7.3 Model–luokka 33](#_Toc25174002)

[4.7.4 Stats–luokka 33](#_Toc25174003)

[4.8 Äänet 33](#_Toc25174004)

[5 Yhteenveto 33](#_Toc25174005)

[LÄHTEET 34](#_Toc25174006)

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. (Ammattinetti, 2019.) 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

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–ohjelmointirajapinnasta löytyy myös todella kattava dokumentaatio, joka on edistänyt sen suosiota. OpenGL on Silicon Graphics Inc:in vuonna 1992 julkaisema kieli– ja alustariippumaton ohjelmointirajapinta vektorigrafiikan piirtämiseen. Vuodesta 2006 eteenpäin OpenGL–rajapintaa on ylläpitänyt Khronos Group. (OpenGL, 2019.)

OpenGL ei tarjoa valmiita funktioita, vaan määritelmiä, joilla ohjataan OpenGL–rajapinnan 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–rajapintaan 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 ilmaista avoimen lähdekoodin GLFW–ohjelmakirjastoa, joka on tarkoitettu 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. Työssä päädyttiin käyttämään LearnOpenGL–opetus–sivustolla käytettyä Multi–Language GL/GLES/EGL/GLX/WGL Loader–Generatoria (glad), 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–rajapinnan 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–kirjastossa käytetään samoja nimeämistapoja kuin GLSL–kielessä, joten jos varjostinkielen erilaiset muuttujatyypit ovat tuttuja, on GLM–kirjaston 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–rajapinnan toiminta muistuttaa hieman OpenGL–rajapintaa 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.)

## stb\_image.h

Otsikkotiedosto stb\_image.h kuuluu yhden tiedoston avoimen lähdekoodin kirjastoihin. Tiedostosta löytyy kuvan lataus sekä avaaminen tiedostosta sekä muistista. Tämä kuvanlataaja tukee monia eri kuvatiedostomuotoja ja tekee kuvien ja tekstuurien käytöstä paljon helpompaa. (Github, 2019.)

## Assimp

Open Asset Import Library eli assimp on avoimen lähdekoodin ohjelmistokirjasto, joka mahdollistaa erilaisten 3d–mallitiedostojen lataamisen. Assimp osaa ladata mallin tiedostosta luut, materiaalit, tekstuurit ja tukee myös animaatioita. (Assimp, 2019.)

## MinGW

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

## Visual Studio Code

Visual Studio Code on kevyt koodin editointiohjelma, joka tukee monia eri ohjelmointikieliä. Codeen saa ladattua lisäosia, jotka helpottavat työskentelyä ja mahdollistavat esimerkiksi funktioiden ja luokkien luomisen valikosta. (Visual Studio Code, 2019)

## Blender

Blender on avoimen lähdekoodin 3d–mallinnusohjelma. Sillä pystyy mm. mallintamaan kolmiulotteisia malleja ja luomaan luut sekä animaatiot niille. Suurin osa opinnäytetyön malleista on tehty Blenderillä. (Blender, 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 tyylinen roolipeli hyödyntäen samankaltaisia tekniikoita, kuin mitä tuohon aikaan käytettiin. Pelikentällä on kolmiulotteinen pelihahmo, jota pelaaja pystyy liikuttamaan etukäteen renderöidyllä taustalla. Opinnäytetyössä on yksi kenttä, jossa pelaajaa pystyy liikuttamaan, ja jossa pystyy juttelemaan kentällä olevan hahmon kanssa.

## 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. Opinnäytetyössä toteutettiin pelkästään kenttä–tila.

### Kenttä

Kenttä–tilassa näytetään pelaajalle pelinäkymä, jossa pelaajahahmoa voi liikutella pelimaailmassa. Kentissä tausta on valmiiksi renderöity kuva, ja kamera on käännetty katsomaan hieman alaspäin. Näin näyttää, että kentällä olevat kolmiulotteiset hahmot kävelevät kaksiulotteiseen taustaan piirretyllä lattioilla ja alustoilla. 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. Maailmankartalla hahmoa ohjataan kolmannesta persoonasta. Kamera on pelaaja hahmon yläpuolella ja takana, ja osoittaa hahmoa päin.

### 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, jossa taustalla näkyy pieni rakennus. Kentällä on pelaajan hahmo, sekä paikallaan oleva hahmo, jolle voi jutella.

# 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–rajapinnalle, 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–rajapinnalle. Kuvia pitää myös pystyä lataamaan, jotta niitä voidaan käyttää tekstuureissa.

## Pelimaailman piirtäminen näytölle

### Ikkunan luonti

Kuvassa 1 näkyy mahdollisimman yksinkertaisen ikkunan luonti GLFW–ohjelmakirjastoa 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–rajapinnalle välitetään tieto ikkunan koosta glViewport–funktiolla. Sitten mennään 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. GLFW–Ikkunan luonti

### Varjostinohjelma

Varjostinohjelma on koodia, joka kertoo OpenGL–rajapinnan ”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 koostuu. 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 luoda 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–rajapinnan piirtoprosessi (Learn OpenGL, 2014)

GLSL on hyvin samantapaista C–kielen kanssa, ja siihen on myös otettu hieman osia C++–kielestä. 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–kielen perus–, vektori sekä matriisitietotyypit (Shreiner et al. 2013, 40)



Kuva 4. GLSL–kielen 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 5 ja 6 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–rajapinnalle tämän käyttöön. Shader–luokan konstruktorissa aluksi luetaan fs– ja vs–tiedostoissa olevat varjostimet char–tyypin muuttujaan myöhempään käyttöön. Varjostimen luonti tapahtuu kuvan 7 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 8. 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–rajapinnan sisäisesti luodusta taulukosta glUniform1i–funktion käyttöön. Kuvassa 9 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–rajapinnalla saadaan malli piirrettyä, täytyy rajapinnalle 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–rajapinnassa 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–rajapinnalle 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–rajapinnassa, 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. Jokainen piirrettävä objekti periytyy Drawable–luokasta ja omaa Draw–funktion, joka hoitelee piirtoon liittyvät prosessit. Näin on helpompi säilöä kentälle piirrettävät objektit taulukkoon, ja kutsua kaikkien piirrettävien objektien piirtofunktiota yhdessä loopissa.

### Tekstuurit

Tekstuurit ja kuvat tuovat peliin ja pelin 3d–malleihin lisää mielenkiintoa. Tekstuurit ovat kaksiulotteisia kuvia esimerkiksi puulaatikon seinistä tai tiiliseinästä. Näitä kuvia käytetään tuomaan elävyyttä ja mielenkiintoa pelin kolmiulotteisiin malleihin.



Kuva 14. Tekstuuri kuutiossa (Wolff, 2013, 119)

Kun uusi tekstuuri luodaan, annetaan tekstuurin konstruktorifunktiossa polku, josta tekstuuriin haluttu kuva löytyy. Ensimmäisenä tekstuurin konstruktorissa täytyy ladata kuvatiedostosta kuvan tiedot, eli kuvan leveys, korkeus sekä värikanavien määrä. Tämä onnistuu stb\_image.h otsikkotiedostosta löytyvästä funktiosta stbi\_load, jolle annetaan parametreinä kuvan polku sekä kokonaislukumuuttujat joihin leveys, korkeus ja värikanavien määrä tallennetaan. Tiedot sijoitetaan char–tyypin osoittimeen myöhempään käyttöön. Ennen tietojen lataamista kutsutaan funktiota stbi\_set\_flip\_vertically\_on\_load parametrilla true, jolloin kuvat latautuvat oikein päin. Kuvassa 15 näkyy stb\_image otsikkotiedoston käyttö tiedon lataamiseen kuvasta.

stbi\_set\_flip\_vertically\_on\_load(true);

unsigned char\* data = stbi\_load(path, &width, &height, &nrChannels, 0);

if(data)

{

        GLenum format;

        if(nrChannels == 1)

        {

            format = GL\_RED;

        }

        else if(nrChannels == 3)

        {

            format = GL\_RGB;

        }

        else if(nrChannels == 4)

        {

            format = GL\_RGBA;

        }

}

Kuva 15. Kuvatiedon lataaminen.

Kun kuvatiedot on ladattu, täytyy tekstuureille luoda OpenGL–rajapinnalle tekstuuriobjekti, joka tallennetaan kokonaislukutunnisteeseen monien muiden OpenGL–objektien tavoin. Tekstuurin luominen on melko yksinkertaista, luodaan tekstuuri kutsumalla funktiota glGenTexture. Sitten sidotaan tekstuuri OpenGL–rajapinnan käyttöön glBindTexture–funktiolla, jotta tulevat tekstuurifunktiot koskevat juuri luotua tekstuuria. Tämän jälkeen tekstuuri luodaan ladatusta kuvatiedosta funktiolla glTexImage2D. Tämä funktio ottaa ensimmäiseksi parametrikseen kohdetekstuurin tyypin, joten GL\_TEXTURE\_2D kertoo OpenGL–rajapinnalle, että halutaan kaksiulotteinen tekstuuri. Toinen parametri on tekstuurin mipmap–taso, joka halutaan yleensä jättää perustasolle, eli nollalle. Seuraava parametri kertoo, millaiseen formaattiin tekstuuri tallentuu. Neljäs ja viides parametri kertoo tekstuurin leveyden ja korkeuden. Kuudes parametri on jäännös vanhemmista versioista, ja kannattaa jättää nollaksi. Seitsemäs ja kahdeksas parametri kertovat lähdekuvan formaatin ja tietotyypin. Viimeisenä argumenttina annetaan aiemmin tallennettu char–pointteri, joka sisältää kuvan datan. Tämän jälkeen määritellään vielä formaatti ladattujen värikanavien määrän perusteella. Kokonaisuudessaan tekstuuriobjektin luominen näyttää kuvan 16 mukaiselta.

// Creating a texture

    unsigned int texture;

    glGenTextures(1, &texture);

    glBindTexture(GL\_TEXTURE\_2D, texture);

    glTexImage2D(GL\_TEXTURE\_2D, 0, GL\_RGB, width, height, 0, GL\_RGB, GL\_UNSIGNED\_BYTE, data);

    glGenerateMipmap(GL\_TEXTURE\_2D);

Kuva 16. Tekstuurin luominen

Tekstuuri on luotu, mutta tarvitaan tieto siitä, kuinka tekstuuri kääriytyy kolmiulotteisen mallin ympärille. Tämä hoituu teksturikoordinaateilla, jotka yleensä annetaan verteksien mukaan, siten että jokaisella verteksillä on oma x– ja y–akselin tekstuurikoordinaatti. Kuvassa 17 on neliön määrittely tekstuurikoordinaattien kanssa ja kuvassa 18 näkyy, kuinka kuva asettuu neliön sisälle täyttäen koko neliön näillä tekstuurikoordinaattiarvoilla. Kuvassa 19 näkyy tekstuuri käärittynä kolmiulotteisen kuution ympärille.

float vertices[] =

    {

    // positions          // texture coords

     0.5f,  0.5f, 0.0f,   1.0f, 1.0f,   // top right

     0.5f, -0.5f, 0.0f,   1.0f, 0.0f,   // bottom right

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

    -0.5f,  0.5f, 0.0f,   0.0f, 1.0f    // top left

    };

Kuva 17. Neliön määrittely tekstuurikoordinaattien kanssa



Kuva 18. Neliön tekstuurikoordinaatit



Kuva 19. Teksturoitu kuutio

Varjostimeen täytyy määritellä sampler2D tyyppinen muuttuja, joka on GLSL–kieleen sisäänrakennettu tietotyyppi tekstuuriobjekteille. Jotta saadaan piirrettyä luotu tekstuuri värin sijaan, pitää sirpalevarjostimessa kutsua funktiota texture, jolla annetaan tekstuuriobjekti sekä tekstuurikoordinaatit. Esimerkki yksinkertaisesta verteksi ja sirpalevarjostimesta tekstuurien kanssa näkyy kuvissa 20 ja 21.

#version 460 core

layout (location = 0) in vec3 aPos;

layout (location = 1) in vec2 aTexCoord;

out vec2 TexCoord;

void main()

{

    gl\_Position = vec4(aPos, 1.0);

    TexCoord = aTexCoord;

}

Kuva 20. Verteksivarjostin tekstuureilla

#version 460 core

out vec4 FragColor;

in vec2 TexCoord;

uniform sampler2D ourTexture;

void main()

{

    FragColor = texture(ourTexture, TexCoord);

}

Kuva 21. Sirpalevarjostin tekstuureilla

Varjostinta käytettäessä täytyy vain ennen piirtokutsua muistaa kiinnittää oikea tekstuuri OpenGL–rajapintaan glBindTexture–funktion kutsulla, jolloin sirpalevarjostin saa automaattisesti tietoonsa oikean tekstuurin.

Opinnäytetyössä suurin osa tekstuureista luodaan Model– sekä GameObject–luokissa kolmiulotteisen mallin luonnin yhteydessä. Pelissä tekstuureita tarvitaan ainoastaan näissä luokissa, joten on loogista luoda ne samalla, kuin itse mallikin. Jos mallille halutaan luonnin yhteydessä antaa tekstuuri, annetaan mallille tekstuurin polku char–pointterina parametrina mallin konstruktorifunktiolle. Konstruktorissa uusi tekstuuri luodaan tekemällä uusi tekstuuriobjekti, joka tapahtuu kutsumalla tekstuuriluokan konstruktoria ja tallettamalla se malliluokassa sijaitsevaan tekstuurityypin muuttujaan.

## Kameran määrittely

Kamera käyttäytyy pelin tilasta riippuen eri tavoin. Kentällä ollessa pelaaja ei pysty itse vaikuttamaan kameraan, ja kamera automaattisesti liikkuu hieman pelaajan mukana joissain kentissä. Taistelukamera on hieman monimutkaisempi. Taisteluun mentäessä kameraa liikutetaan, jotta näyttää että kamera lentää taistelukentälle, ja kuvaa kaikkia taistelussa olevia hahmoja. Maailmankartalla ollessa pelaaja pystyy kääntelemään kameraa, jolloin kamera pyörii pelaajan ympäri. Päävalikossa kamera muutetaan ortografiseen tilaan, sillä perspektiiviä ei tarvita. Valikossa kaikki piirrettävät elementit ovat kaksiulotteisia.

Opinnäytetyössä toteutettiin pelkästään kenttätilan kamera. Kamera–luokan konstruktorissa määritellään kameran käyttöön tarvittavat muuttujat ja vektorit. Jotta kameraa ja kuvakulmaa voidaan säädellä, tarvitaan tieto kameran sijainnista, suuntavektori kamerasta suoraan eteenpäin, vektori kamerasta suoraan oikealle sekä vektori kamerasta suoraan ylöspäin. Eli kameralle määritellään kolmiulotteinen suorakulmainen koordinaatisto, jossa kamera on origossa. Näillä vektoreilla saadaan mm. pistetulon avulla selville, osoittaako kolmiulotteisen kappaleen seinämä kameraan. Vektorien ristitulolla on mahdollista määrittää esimerkiksi kolmiulotteisen kappaleen seinämän normaalivektori sekä suunta kappaleeseen. (McShaffry ja Graham, 448–455)

Kameran konstruktorissa määritellään aluksi kameran sijainti maailmassa, joka on pelin alkaessa pisteessä (0.0, 0.0, 3.0). Tämä piste on valittu alkupisteeksi siksi, että z-akselin ajatellaan menevän pelaajan ruutua kohti, joten kameran halutaan alkutilanteessa olevan tässä pisteessä kameran koordinaatiston määrittelyä varten. Kamera-luokan worldUp–muuttuja alustetaan vektorilla (0.0, 1.0, 0.0), joka osoittaa x– ja z–akseleiden tason normaalin suuntaan. Alkutilanteessa määritellään kameran katsesuunnaksi vektori (0.0, 0.0, -1.0), jolloin kamera osoittaa maailman origoa kohti. Kameran koordinaatiston oikea vektori saadaan laskemalla normaalivektori worldUp–vektorin ja kameran katsesuunnan vektorin ristitulosta. Kamerasta ylöspäin osoittava vektori saadaan laskettua kameran katsesuunnan vektorin sekä kameran oikean vektorin ristitulosta.

Opinnäytetyössä käytetään Model View Projection–mallia, joka tarkoittaa, että koodissa määritellään kolme matriisia. Model–matriisiin tallennetaan tieto kolmiulotteisen kappaleen sijainnista ja orientaatiosta. View–matriisiin tarvitaan tieto kameran sijainnista ja orientaatiosta pelimaailmassa, ja Projection–matriisiin tallennetaan tieto näytön koosta ja näkökentästä. (McShaffry ja Graham, 478–479)

View–matriisi saadaan luotua käyttämällä GLM–ohjelmakirjaston funktiota lookAt, jolle annetaan parametrina aiemmin määritellyt kameran koordinaatiston vektorit, eli kamerasta katsottuna eteen, ylös ja oikealle osoittavat vektorit. Funktio palauttaa 4x4 matriisin, joka sijoitetaan kamera–luokan matriisimuuttujaan view. Projection–matriisin laskemiseen käytetään GLM–ohjelmakirjaston perspective–funktiota. Funktiolle annetaan parametreiksi näkökentän suuruus asteina, ruudun kuvasuhde sekä arvot sille, kuinka läheltä ja kaukaa kamera leikkaa piirrettävän alueen. Kameran luontiprosessi näkyy kuvassa 22. (Learn OpenGL, 2013)

// Camera definitions

    // Positive z-axis goes towards the screen and towards you, so we want to move the camera that way.

    cameraPos = glm::vec3(0.0f, 0.0f, 3.0f);

    // World up vector

    worldUp = glm::vec3(0.0f, 1.0f, 0.0f);

    // Camera front

    cameraFront = glm::vec3(0.0, 0.0, -1.0f);

    // Get camera right vector by getting the cross product of worldUp and cameraDirection vectors

    cameraRight = glm::normalize(glm::cross(worldUp, cameraFront));

    // Get camera up vector

    cameraUp = glm::cross(cameraFront, cameraRight);

    // Use glm::lookAt to define the view matrix by providing the function the camera position,

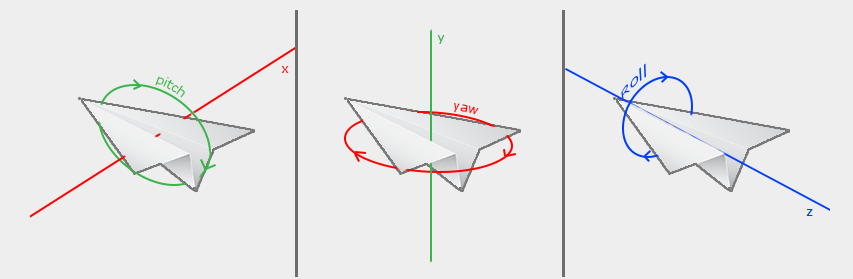
    // target position and world vector up.

    view = glm::lookAt(cameraPos, cameraPos + cameraFront, cameraUp);

    projection = glm::perspective(glm::radians(45.0f), (float)screenWidth / (float)screenHeight, 0.1f, 10000.0f);

Kuva 22. Kameran konstruktorifunktio

Opinnäytetyössä kameraa haluttiin myös kääntää, jotta on helpompi tarkastella ja testata kolmiulotteisia malleja. Kameran kääntämistä varten täytyy määritellä Eulerin kaavan mukaiset kulmat: roll, yaw ja pitch, joiden avulla voidaan esittää mikä tahansa rotaatio kolmiulotteisessa maailmassa. Pitch–kulma kertoo, kuinka paljon katsotaan ylös tai alas. Yaw–kulma kertoo käännön suuruuden sivuttaissuunnassa. Roll kertoo pyörähdyksen määrän. Kuva 23 näyttää, kuinka Eulerin kulmat vaikuttavat kappaleen rotaatioon.



Kuva 23. Roll–, pitch–, ja yaw–rotaatiot (Learn OpenGL, 2013)

Kun kameraa liikutetaan, täytyy kameran view–matriisi laskea uudestaan. Kameran liikuttaminen tapahtuu opinnäytetyössä funktiolla MoveCamera, jolle annetaan parametrinä suunta. MoveCamera–funktio näkyy kuvassa 24.

void Camera::MoveCamera(Direction dir)

{

    switch (dir)

    {

    case Direction::Forward:

        cameraPos += cameraSpeed \* cameraFront;

        break;

    case Direction::Right:

        cameraPos += glm::normalize(glm::cross(cameraFront, cameraUp)) \* cameraSpeed;

        break;

    case Direction::Back:

        cameraPos -= cameraSpeed \* cameraFront;

        break;

    case Direction::Left:

        cameraPos -= glm::normalize(glm::cross(cameraFront, cameraUp)) \* cameraSpeed;

        break;

    default:

        break;

    }

    view = glm::lookAt(cameraPos, cameraPos + cameraFront, cameraUp);

}

Kuva 24. MoveCamera-funktio

Kun kameraa käännetään ja roll–, pitch– tai yaw–arvoja muutetaan, täytyy kameran koordinaatiston vektorit laskea uudestaan. Tämä tehdään funktiossa UpdateVectors. UpdateVectors–funktiossa aluksi lasketaan kameran eteenpäin osoittava vektori uudestaan käyttäen muuttuneita pitch– ja yaw–arvoja. Roll–muuttujan arvoa ei oteta laskuihin mukaan, sillä kameran ei haluta pyörivän z–akselilla. Kun eteenpäin osoittava vektori on saatu laskettua, käytetään sitä uuden oikealle sekä ylöspäin osoittavien vektorien laskentaan. Sitten vielä määritellään view–matriisi uudelleen näillä vektoreilla. UpdateVectors–funktio näkyy kuvassa 25.

void Camera::UpdateVectors()

{

    // Calculate the new Front vector

    glm::vec3 front;

    front.x = cos(glm::radians(yaw)) \* cos(glm::radians(pitch));

    front.y = sin(glm::radians(pitch));

    front.z = sin(glm::radians(yaw)) \* cos(glm::radians(pitch));

    cameraFront = glm::normalize(front);

    // Also re-calculate the Right and Up vector

    cameraRight = glm::normalize(glm::cross(cameraFront, worldUp));  // Normalize the vectors, because their length gets closer to 0 the more you look up or down which results in slower movement.

    cameraUp = glm::normalize(glm::cross(cameraRight, cameraFront));

    view = glm::lookAt(cameraPos, cameraPos + cameraFront, cameraUp);

}

Kuva 25. UpdateVectors–funktio

View– ja projection–matriisien lisäksi täytyy määritellä model–matriisi, joka tallennetaan mallin luokkaan muuttujaan. Model–matriisi pitää sisällään tiedon kolmiulotteisen mallin sijainnista maailmassa, mallin rotaatiosta sekä mallin koosta. Kun mallia halutaan siirtää, kutsutaan GLM–ohjelmistokirjastosta translate-funktiota, jolle annetaan vektori, johon malli halutaan siirtää. Funktio palauttaa model–matriisin, joka sisältää tiedon mallin sijainnista. Mallin kasvattamiseksi kutsutaan GLM–ohjelmakirjaston funktiota scale, jolle annetaan vektorina haluttu mallin koko. Funktion palauttama matriisi sijoitetaan myös model–muuttujaan. Mallin pyörittämiseen käytetään funktiota rotate, jolle annetaan alkuperäinen matriisi, astemäärä, sekä akseli, jonka ympäri mallia pyöritetään. Rotate–funktio palauttaa myös matriisi–tyypin muuttujan, joka sijoitetaan model–muuttujaan. Esimerkki model–matriisin käytöstä Model–luokan piirtofunktiossa näkyy kuvassa 26.

model = glm::mat4(1.0f);

    model = glm::translate(model, position);

    model = glm::scale(model, scale);

    if(needToRotate)

    {

        model = glm::rotate(model, desiredRot, glm::vec3(0.0f, 1.0f, 0.0f));

    }

    shader->SetMat4("model", model);

Kuva 26. Model–matriisin käyttö

Jotta model–, view– sekä projection–matriiseja voidaan käyttää pelissä, pitää kirjoittaa varjostin, joka käyttää näitä matriiseja. Sirpalevarjostimeen ei tarvitse tehdä ModelViewProjection–mallia varten muutoksia. Verteksivarjostimeen luodaan muuttujat matriiseja varten, ja varjostimelle syötetyt verteksisijainnit kerrotaan niillä. Esimerkki verteksivarjostimesta, joka käyttää näitä matriiseja hyödyksi on kuvassa 27.

// Shader with color and positions.

#version 460 core

layout (location = 0) in vec3 aPos;

uniform mat4 model;

uniform mat4 view;

uniform mat4 projection;

void main()

{

    gl\_Position = projection \* view \* model \* vec4(aPos, 1.0);

}

Kuva 27. Verteksivarjostin ModelViewProjection–matriisien kanssa.

Pääohjelmassa varjostimelle annetaan matriisit kutsumalla Shader–luokan funktiota SetMat4, jolle annetaan parametrina muuttujan nimi varjostimessa sekä itse matriisi. Shader–luokan SetMat4–funktio kutsuu glad–ohjelmistokirjaston funktiota glUniformMatrix4fv, joka vie tiedon matriisista varjostimelle. Esimerkki matriisien viemisestä varjostimelle on kuvassa 28.

        colorNShader.Use();

        colorNShader.SetMat4("view", mainCamera.GetView());

        colorNShader.SetMat4("projection", mainCamera.GetProjection());

Kuva 28. Matriisien vieminen varjostimelle

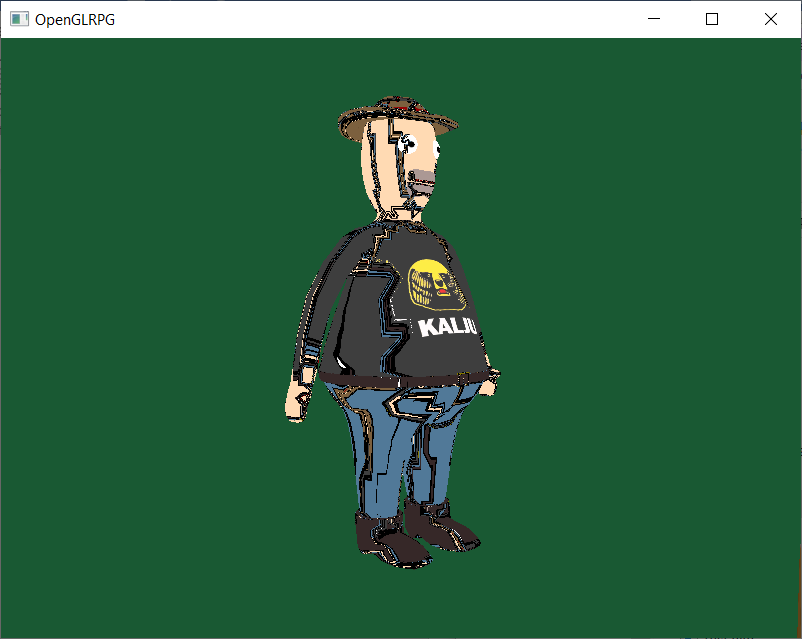
## 3d–mallien lataaminen

Kolmiulotteisten mallien verteksien määrittely käsin on todella työlästä. Tästä syystä mallit luodaan opinnäytetyössä 3d–mallinnusohjelmassa, Blenderissä. Mallit viedään kuvan 29 asetusten mukaisesti obj–tiedostoon, jotta ne saadaan ladattua oikein opinnäytetyössä.

Kuva 29. Mallin vienti Blenderistä obj–tiedostoon (OpenGL–tutorial, 2017)

Obj–tiedostomuoto valittiin malleille siksi, että niissä tieto vertekseistä on selkeimmin saatavilla ja luettavissa. Obj–tiedostossa jokaisen rivin alussa on kirjain, joka kertoo mitä tietoa rivillä on. Jos rivi alkaa kirjaimella v, rivi kertoo verteksin sijainnin. Kirjaimet vt rivin alussa kertovat, että rivi antaa tiedon verteksin tekstuurikoordinaateista. Rivi, joka alkaa kirjaimilla vn, kertoo verteksin normaalin. F–kirjain määrittelee verteksien järjestyksen ja mallin sivut. (FileFormat, 2019)

Obj–tiedosto luetaan Model–luokan konstruktorissa rivi kerrallaan läpi, ja rivin alkukirjainten perusteella tallennetaan rivin luettu arvo oikeaan taulukkoon. Kun arvot on luettu loppuun, luodaan mallille VBO, VAO sekä EBO, ja puskuroidaan obj–tiedostosta luettu data OpenGL–rajapinnan käyttöön. Uusi tekstuuri luodaan mallille käyttöön konstruktorin lopussa ja asetetaan muuttujaan texture. Mallille myös annetaan parametrina Shader–tyypin muuttuja joka asetetaan mallin shader–nimiseen muuttujaan, jotta mallilla on tieto sen käyttämästä varjostimesta, ja on helppo piirtää malli Draw–funktiossa käyttämällä tätä varjostinta. Omalla obj–tiedoston lataajalla mallin verteksit ja tekstuurit toimivat oikein, mutta jättävät joihinkin kohtiin mallia saumat. Mallien lataukseen kannattaa käyttää assimp–ohjelmakirjastoa, sillä se tukee monia eri tiedostomuotoja ja osaa ladata myös materiaalit.



Kuva 30. Kolmiulotteinen malli omalla obj–tiedoston lataajalla

Assimp–ohjelmakirjasto lataa kolmiulotteisen mallin tiedot omaan tietorakenteeseensa. Assimp–kirjastolla mallin konstruktorissa ensin määritellään aiScene tyyppiä oleva scene muuttuja, johon sijoitetaan assimp–lataajan lukema mallitiedosto. ReadFile–funktio, joka lukee mallitiedoston, palauttaa scene–muuttujaan kaiken tiedon mallista, kuten verteksien sijainnit, tekstuurikoordinaatit sekä materiaalit. Assimp lataa mallin nodeihin, jotka voivat sisältää lapsinodeja. Nämä nodet käsitellään rekursiivisesti ProcessNode–funktiolla, jolle annetaan parametrina ensimmäinen node, sekä scene–muuttuja. Funktion toisessa loopissa kutsutaan rekursiivisesti ProcessNodea, jolle annetaan parametrina tämänhetkisen noden lapsinodet. Noden meshit käydään läpi ensimmäisessä loopissa, jonka sisällä ensin sijoitetaan assimp–ohjelmakirjaston aiMesh tietotyypin muuttujaan tieto noden meshistä. Tämän jälkeen annetaan aiMesh–muuttuja ProcessMesh–funktiolle käsiteltäväksi, joka muuntaa aiMesh–muuttujan itse määritellyksi Mesh–tyypin muuttujaksi, joka tallennetaan meshes–listaan. Oma Mesh–luokka sisältää listan vertekseistä, piirtojärjestyslistan sekä listan käytetyistä tekstuureista. ProcessMesh–funktiossa käydään läpi sille annetun aiMesh muuttujan verteksit, tekstuurikoordinaatit sekä piirtojärjestysmuuttujat, ja tallennetaan ne omiin listoihin. Luodaan myös tekstuuri, jota malli käyttää. Funktiosta palautetaan oma Mesh–tyypin muuttuja, jolle on annettu prosessoidut aiMesh–muuttujan listat. Tämä Mesh–muuttuja pitää sisällää tiedon vertekseistä, piirtojärjestyksestä sekä tekstuureista. Mesh–luokan konstruktorissa määritellään meshille VBO, VAO sekä EBO, ja tieto vertekseistä puskuroidaan OpenGL–rajapinnan tietoon. ProcessNode–funktio näkyy kuvassa 31.

void ProcessNode(aiNode\* node, const aiScene\* scene)

{

// process all the node's meshes (if any)

for (unsigned int i = 0; i < node->mNumMeshes; i++)

{

aiMesh\* mesh = scene->mMeshes[node->mMeshes[i]];

meshes.push\_back(ProcessMesh(mesh, scene));

}

// then do the same for each of its children

for (unsigned int i = 0; i < node->mNumChildren; i++)

{

ProcessNode(node->mChildren[i], scene);

}

}

Kuva 31. ProcessNode–funktio

Kun ProcessNode on päässyt rekursiivisesti loppuun, ja kaikki nodet on käyty läpi, voidaan malli piirtää. Mallin piirto tapahtuu mallin Draw–funktiossa, jossa loopataan meshes–listan läpi ja kutsutaan jokaiselle yksittäiselle meshille Mesh–luokasta löytyvää Draw–funktiota. Mesh–luokan Draw–funktiossa on määritelty itse meshin piirtokoodi.

## Renderer–luokka

Kerro työssä kirjoitetusta renderer luokasta sekä singletonin hyödynnys

class Renderer

{

static Renderer& GetInstance()

{

static Renderer instance;

return instance;

};

};

## Liikkumisen ja interaktiivisuuden lisääminen peliin

Pelaajan liikutuksesta tekstiä.

## Pelin luokat

### Camera–luokka

Kamera

### Shader–luokka

Varjostimet.

### Model–luokka

Model.

### Stats–luokka

Stats.

## Äänet

Äänien lisäys peliin

# Yhteenveto

Yhteenveto.

LÄHTEET

Ammattinetti (2019). *Peliteollisuus.* Ammattinetin www–sivusto. Saatavissa: http://www.ammattinetti.fi/ammattialat/detail/1a2d07830a65344600a33079becd6e88 [viitattu 18.11.2019]

Assimp (2018). *The Open–Asset–Importer–Lib*. Assimpin www–sivusto. Saatavissa: http://www.assimp.org/ [viitattu 13.11.2019]

Blender (2019). *About* Blenderin www–sivusto. Saatavissa: https://www.blender.org/about/ [viitattu 20.11.2019]

McShaffry, M. ja Graham, D. (2013). Game Coding Complete. 4. painos. Boston: Course Technology PTR.

FileFormat (2019). *Wavefront OBJ File Format Summary* FileFormatin www–sivusto. Saatavissa: https://www.fileformat.info/format/wavefrontobj/egff.htm [viitattu 20.11.2019]

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](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]

Github. (2019). *Stb.* Githubin www–sivusto. Saatavissa: <https://github.com/nothings/stb> [viitattu 06.11.2019]

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–tutorial. (2017). *Creating an OBJ file in Blender* Kuva. Saatavissa: http://www.opengl-tutorial.org/beginners-tutorials/tutorial-7-model-loading/ [viitattu 20.11.2019]

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](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](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.

Visual Studio Code. (2019). *Getting Started* Visual Studio Code:n www–sivusto. Saatavissa: https://code.visualstudio.com/docs [viitattu 20.11.2019]

Wolff, D. (2013). *OpenGL 4 Shading Language Cookbook*. 2. Painos. Birmingham: Packt Publishing.