Teemu Turku

Roolipelidemo OpenGL:llä

Opinnäytetyö
Tietotekniikka / Peliohjelmointi

2019

Kaakkois-Suomen
ammattikorkeakoulu

Kaakkois-Suomen ammattikorkeakoulu

Tekijä/Tekijät	Tutkinto	Aika
Teemu Turku	Insinööri (AMK)	Syyskuu 2019
Opinnäytetyön nimi Roolipelidemo OpenGL:llä		61 sivua
Toimeksiantaja Kaakkois-Suomen ammattikorkeakoulu, 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 ja siitä, mitä lopullisessa pelissä tulisi olla. Työssä käytetään ohjelmointikielenä C++ -ohjelmointikieltä. Opinnäytetyön lukijalla oletetaan olevan ymmärrys ohjelmoinnin perusteista.

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. Työssä myös kerrotaan, kuinka luodaan yksinkertainen collision-järjestelmä, ja kuinka voidaan helposti määritellä kenttään tapahtumia.

Pelidemo sisältää yhden kentän, jossa pelaaja pystyy liikuttamaan pelaajahahmoa ja jutte-lemaan pelimaailmassa seisovan hahmon kanssa. Pelaaja pystyy myös tutkimaan kentällä näkyvää tynnyriä. Opinnäytetyössä asetetut tavoitteet pelidemolle saavutettiin.

Asiasanat

OpenGL, C++, pelimoottori, peliohjelmointi

South-Eastern Finland
University of Applied Sciences

Author (authors)	Degree	Time
Teemu Turku	Bachelor of Engineering	September 2019

Thesis title

61 pages

Role–playing game demo made in OpenGL

Commissioned by

South-Eastern Finland University of Applied Sciences, Gamelab

Supervisor

Niina Mässeli

Abstract

The objective of this thesis is to create a gameplay demo using the OpenGL–programming interface. The focus of the thesis is to create a simple game engine for a small role–playing game.

The beginning of the thesis explains which software libraries and programs were used. The planning section covers the planning of the project and tells which features the final product should have. The game engine was written in C++. The reader is presumed to have knowledge of basic programming concepts.

The implementation section of the thesis covers how the game engine was made. The section describes how to create an OpenGL-context window, how to play sounds, get user input and load custom 3d models. A camera is implemented into a 3d–world and the use of model view and projection matrices is described. The implementation of a simple collision detection system is also explained, as is the process of defining specific level events.

The demo contains one level, where the player can move the main character around and interact with the other characters in the level. The player is also able to investigate a barrel placed on the level. The level has looping music playing in the background and an event system, which makes it easy to add text events. The goals set for the demo were achieved.

Keywords

OpenGL, C++, game engine, game programming

SISÄLLYS

1 JOHDANTO 6

2 KÄYTETYT MENETELMÄT 6

2.1 Ohjelmat 7

2.2 Ohjelmakirjastot 7

2.3 Otsikkotiedostot 10

2.4 Muut 11

3 SUUNNITTELU 12

3.1 Kenttä 15

3.2 Maailmankartta 15

3.3 Taistelu 16

3.4 Päävalikko 16

3.5 Valikko 17

4 TOTEUTUS 17

4.1 Tavoitteet 18

4.2 Pelimaailman piirtäminen näytölle 18

4.2.1 Ikkunan luonti 18

4.2.2 Varjostinohjelma 21

4.2.3 Suorakulmion piirtäminen 26

4.2.4 Tekstuurit 29

4.3 Kameran määrittely 34

4.4 3D-mallien lataaminen 41

4.5 Drawable-luokka 44
4.6 Käyttäjän syöte 46
4.7 Tekstin renderöinti 46
4.8 Liikkumisen ja interaktiivisuuden lisääminen peliin 49
4.8.1 Mallin liikuttaminen 50
4.8.2 Collider-luokka 50
4.8.3 Event-luokka 51
4.8.4 EventHandler-luokka 51
4.9 Äänet 52
5 YHTEENVETO 54
LÄHTEET 56

1 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. (Zarrad 2018.) 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 vaikuttaa kentällä oleviin esineisiin. Opinnäytetyössä keskitytään tekemään toimiva ja pelattava peli Windows-alustoille. Iso osa työhön käytetystä ajasta menee alkukantaisen pelimoottorin ohjelmointiin.

Halusin syventää ohjelmoinnin osaamistani ja tutustua tarkemmin siihen, miten pelimoottorit toimivat. Valmiit pelimoottorit luovat yleensä ikkunan ja hoitavat kolmiulotteisten mallien piirtämisen automaattisesti. Pelimoottorien monet toiminnot tapahtuvat siten, ettei ohjelmoija välttämättä tiedä, miten esimerkiksi pelimoottori piirtää kappaleen ruudulle. Opinnäytetyössä perehdytään mm. näiden toimintojen toteuttamiseen.

Toimeksiantajana opinnäytetyössä on Kaakkois-Suomen ammattikorkeakoulun Gamelab. Gamelab on peliohjelmoinnin oppimisympäristö Kotkassa, ja tarjoaa laajan peliohjelmoinnin opintokokonaisuuden sekä nykyaikaiset välineet pelien ohjelmointiin. (XAMK 2019.)

2 KÄYTETYT MENETELMÄT

Vaihtoehtoja grafiikan piirtämiselle, äänien toistamiselle ja muillekin erilaisille toiminnoille löytyy runsaasti. Onkin siis tärkeää valita oikeat työkalut pelin tekoon. OpenGL valittiin grafiikkakirjastoksi siksi että siihen löytyy todella paljon dokumentaatiota sen suosion myötä. Suurin osa opinnäytetyössä käytetyistä ohjelmistokirjastoista valittiin niiden helppokäyttöisyyden vuoksi. Iso osa ohjelmistokirjastoista valittiin myös sillä perusteella, että ne toimivat hyvin OpenGL-rajapinnan kanssa.

2.1 Ohjelmat

Tässä luvussa kerrotaan opinnäytetyössä käytetyistä tietokoneohjelmista. Koodin editointia varten valittiin Visual Studio Code sen muokattavuuden vuoksi. Projektissa kuitenkin siirryttiin käyttämään Visual Studio 2019 -ohjelmaa kirjastojen linkityksen helpottamiseksi. Blender valittiin ohjelmaksi kolmiulotteisten mallien käsittelyyn. Se on ilmainen ja suosittu 3D-mallinnusohjelma, joten sen käyttöön löytyy paljon dokumentaatiota.

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.)

Visual Studio 2019

Projektin loppuvaiheessa vaihdettiin koodin editointiohjelmaksi sekä suoritettavan tiedoston luomiseen Visual Studio 2019. On helpompi saada ohjelmistokirjastot linkitettyä ja toimimaan Visual Studion kanssa kuin MinGW-työkalulla, sillä suurimmalle osalle ohjelmistokirjastoista on valmiiksi kirjoitettu CMakeLists-tiedosto Visual Studioa varten. (Visual Studio 2019.)

Blender

Blender on avoimen lähdekoodin 3D-mallinnusohjelma. Sillä pystyy mm. mallintamaan kolmiulotteisia malleja ja luomaan luut sekä animaatiot niille. 3D-mallinnuksessa luut mahdollistavat mallin liikutuksen. Blender-ohjelman kehittämiseen ja ylläpitämiseen perustettin Blender Foundation -säätiö vuonna 2002. Blender on järjestelmäriippumaton ja sen käyttöliittymä käyttää OpenGL-rajapintaa. Suurin osa opinnäytetyön malleista on tehty Blenderillä. (Blender 2019.)

2.2 Ohjelmakirjastot

Tässä luvussa kerrotaan työssä käytetyistä ohjelmakirjastoista. Ohjelmakirjastot ovat kokoelma tiedostoja, ohjelmia tai funktioita, joita pystyy omassa koo-

dissa käyttämään. Opinnäytetyössä ohjelmakirjastojen avulla toteutetaan suurin osa projektin toiminnallisuudesta. Esimerkiksi tyhjän ikkunan luominen, grafiikan piirtäminen tai äänentoisto ei olisi mahdollista ilman ohjelmakirjastoja.

OpenGL

Graafisia ohjelmistokirjastoja ovat mm. Microsoft Direct3D, Vulkan sekä OpenGL. Grafiikkaa voi piirtää myös käyttämällä multimediakirjastoja, joita ovat mm. Simple and Fast Multimedia Library ja Simple DirectMedia Layer. Projektiin valittiin OpenGL, sillä se soveltuu hyvin kolmiulotteisten pelien tekoon. OpenGL-ohjelmointirajapinnasta löytyy myös kattava dokumentaatio, joka on edistänyt sen suosiota. OpenGL on Silicon Graphics Incin 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:n versioissa lähestymistapa 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 helpokäyttöisempiä ja paremmin ymmärrettäviä, tämä lähestymistapa ei ollut kovin tehokas. (Learn OpenGL 2014a.)

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 löytyy paljon erilaisia OpenGL-latauskirjastoja, joiden tehtävänä on automaattisesti selvittää oikean funktion muistipaikka sillä hetkellä olevaan alustaan sopivaksi. Työssä päädyttiin käyttämään LearnOpenGL opetussivustolla käytettyä Multi-Language GL/GLES/EGL/GLX/WGL Loader-Generatoria (glad), sillä se vaikutti yksinkertaiselta sekä helppokäyttöiseltä. (OpenGL Wiki 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.)

OpenAL

Äänentoistoon tarkoitettuja ohjelmistokirjastoja löytyy valtava määrä. Opinnäytetyöhön valittiin OpenAL äänentoistoon ja freealut äänitiedostojen lukemiseen ja lataamiseen. Freealut on ilmainen ja päivitetty ohjelmakirjasto, joka on tehty OpenAL Utility Toolkitin pohjalta. (The OpenAL Utility Toolkit 2006). 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-ohjelmointikielellä 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 2014e.)

2.3 Otsikkotiedostot

Otsikkotiedostot ovat tiedostoja, jotka sisältävät hyödyllisiä funktioita ja muuttujia ohjelman käyttöön. Otsikkotiedostot ovat tyypillisesti ”.h”- tai ”.hpp” -päätteisiä ja helppoja sisällyttää ohjelmaan ”#include”-direktiivillä. Opinnäytetyössä käytetään otsikkotiedostoja mm. vektori- ja matriisimatematiikkaan sekä kuvatiedostojen lataamiseen.

GLM

Kolmiulotteisessa pelissä tarvitaan paljon vektori- sekä matriisimatematiikkaa. Onneksi näiden laskutoimitusten helpotukseksi löytyy OpenGL Mathematics -matematiikkakirjasto (GLM). 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.)

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 2019b.)

2.4 Muut

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.)

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++ -ohjelmointikielen koodi tietokoneen ymmärrettäväksi konekoodiksi Windows-käyttöjärjestelmillä. (Mingw 2019.)

CMake

CMake on avoimen lähdekoodin alustariippumaton työkalu koodin kääntämiseen. Opinnäytetyössä käytetyt ohjelmakirjastot on käännetty käyttäen CMake-ohjelmaa. CMaken nykyinen versio on 3.16.0. (CMake 2019a.)

CMake käyttää CMakeLists-tekstitiedostoa, jossa määritellään käännettävä lähdekoodi sekä kohteeksi suoritettava kirjasto, ohjelmakirjasto tai molemmat. Opinnäytetyössä luodaan CMake-ohjelmalla käytettäville ohjelmakirjastoille Visual Studio -projekti, joka käynnistetään. Visual Studiossa rakennetaan lopulliset käytettävät kirjastot build-painikkeesta. (CMake 2019b.)

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 2019a).

Singleton

Singleton on suunnittelumalli, jolla varmistetaan, että luokasta on olemassa vain yksi olio koodissa. Singleton-mallia on moitittu, sillä se antaa pääsyn luokkaan mistä tahansa kohtaa koodissa, ja väärinkäytettynä voi aiheuttaa tuoda ohjelmistoon monenlaisia ongelmia. (Nystrom 2014, 73–75.)

Singleton-mallia käytetään opinnäytetyössä, jotta päästään helposti käsiksi joidenkin luokkien funktioihin ja toimintoihin. Esimerkiksi ääniä halutaan toistaa monessa eri tilanteessa, joten AudioManager-luokan PlaySound-funktiota halutaan kutsua vaikkapa pelaajan painaessa välilyöntiä tai kolmiulotteisen mallin törmätessä seinään. On siis helpompaa käyttää AudioManager-luokassa singleton-mallia kuin olisi viedä instanssi AudioManager-oliosta kaikkiin eri luokkiin ja funktioihin, joissa sitä tarvitaan. Esimerkki C++ -ohjelmointikielessä singleton-mallista löytyy kuvasta 1.

Kuvan 1 singleton-luokka on C++ -ohjelmointikielellä kirjoitettu esimerkki singleton-mallista. C++11 versiossa lokaali staattinen muuttuja luodaan vain kerran, joten kuvan mukainen funktio palauttaa aina saman olion luokasta. (Nystrom 2014, 75.)

```
class Singleton
{
    static Singleton& GetInstance()
    {
        static Singleton instance;
        return instance;
    };
};
```

Kuva 1. Singleton-malli

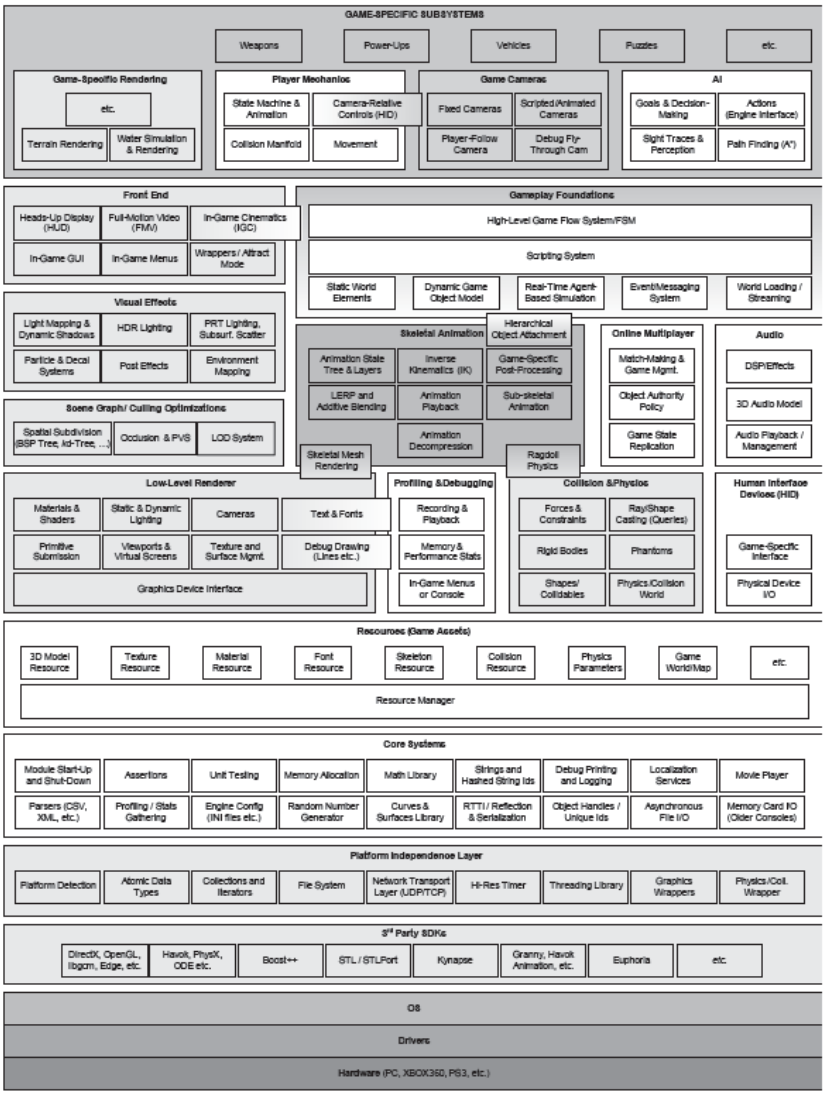
3 SUUNNITTELU

Ensimmäiset pelimoottorit syntyivät 1990-luvun puolivälissä, kun Id Softwaren suuren suosion saavuttanut peli Doom julkaistiin. Pelin eri osat, kuten äänentoisto ja grafiikanpiirto oli ohjelmoitu toimimaan erillään. Pelin pelaajien sekä

ohjelmoinnista kiinnostuneiden oli mahdollista tehdä täysin uusia pelejä käyttäen hyödyksi Doom-pelin eri osia. (Gregory 2009, 11.)

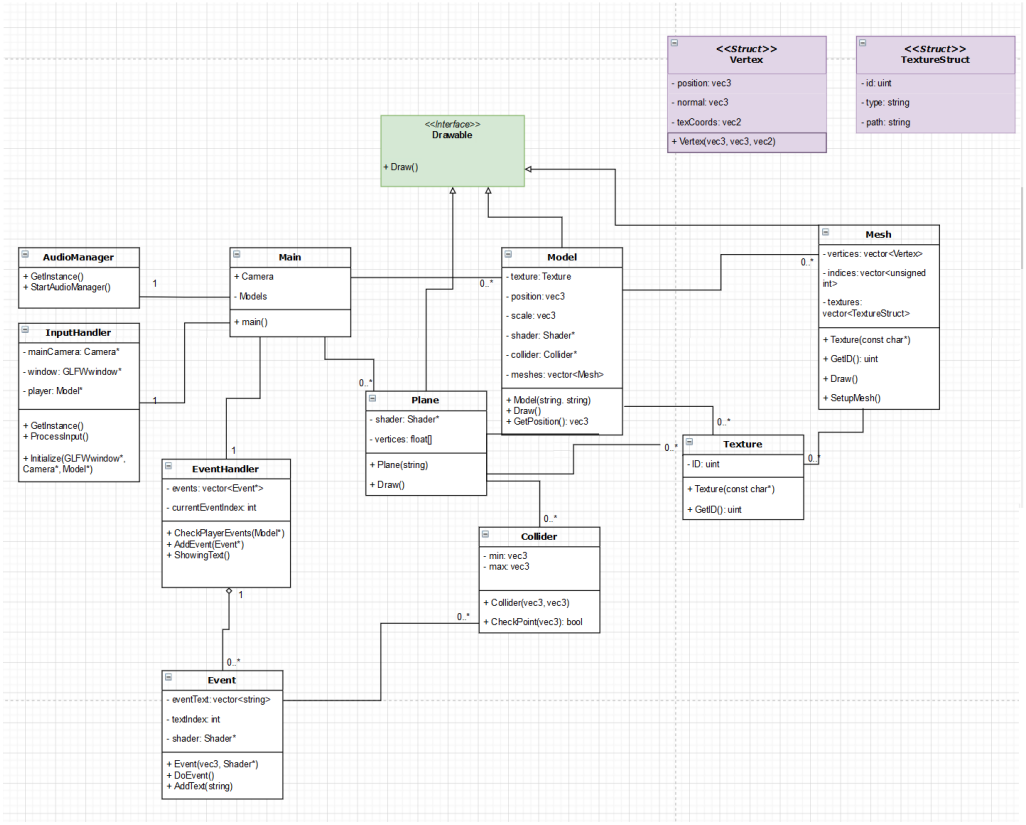
Pelimoottorilta vaaditaan yleensä erilaisia ominaisuuksia riippuen kehitettävän pelin genrestä. Yhteistä kaikille kolmiulotteisille peleille on kuitenkin genrestä riippumatta, että tarvitaan tapa saada pelaajalta syöte. Tarvitaan myös tapa näyttää tekstiä, piirtää kolmiulotteisia malleja sekä äänentoisto. (Gregory 2009, 13.)

Kuvassa 2 näkyy tyypillisen kolmiulotteisen pelimoottorin ajonaikaiset komponentit. Pelimoottorit koostuvat monesta eri osasta. Opinnäytetyössä keskitytään luomaan toimiva ja pelattava pelidemo, joten työssä toteutetaan pelimoottorille tärkeimmät komponentit, kuten mm. äänentoisto, kamera sekä pelaajan liike.



Kuva 2. Pelimoottorin ajonaikaiset komponentit (Gregory 2009, 29)

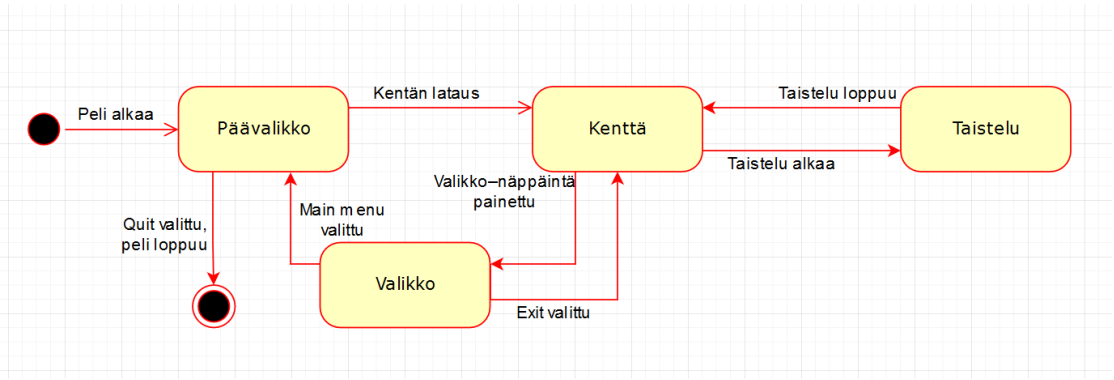
Pelin idea syntyi Playstation 1 -ajan roolipelien innoittamana. Työssä pyritään luomaan 1990-luvun loppupuolen tyylinen roolipeli hyödyntäen samankaltaisia tekniikoita kuin mitä tuohon aikaan käytettiin. Pelikentällä on kolmiulotteinen pelihahmo, jota pelaaja pystyy liikuttamaan etukäteen renderöidyllä taustalla. Opinnäytetyössä on yksi kenttä, jossa pelaajaa pystyy liikuttamaan, ja jossa pystyy juttelemaan kentällä olevan hahmon kanssa. Kuvassa 3 näkyy luokkakaavio peliin suunnitelluista luokista. Kaikki ruudulle piirrettävät luokat perivät kaaviossa näkyvän Drawable-luokan, ja sisältävät toteutuksen Draw-funktiolle. Pelidemossa pääsilmukka on Main-luokassa, joten luokat joita käytetään, luodaan Main-luokassa. AudioManager-, InputHandler- sekä EventHandler -luokat käyttävät singleton-mallia, joten niistä jokaisesta luodaan yksi olio Main-luokassa. Vextex- ja TextureStruct -rakenteet on määritelty kolmiulotteisten mallien lataamista varten. Luokkakaavion eri luokkien toiminnoista kerrotaan enemmän luvussa 4.



Kuva 3. Luokkakaavio pelin sisältämistä luokista

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, mutta tässä luvussa kerrotaan mitä lopulliseen peliin on tarkoitus toteuttaa. Kuvassa 4 näkyy tilakaavio pelin eri tiloista.



Kuva 4. Tilakaavio pelin eri tiloista

3.1 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.

3.2 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

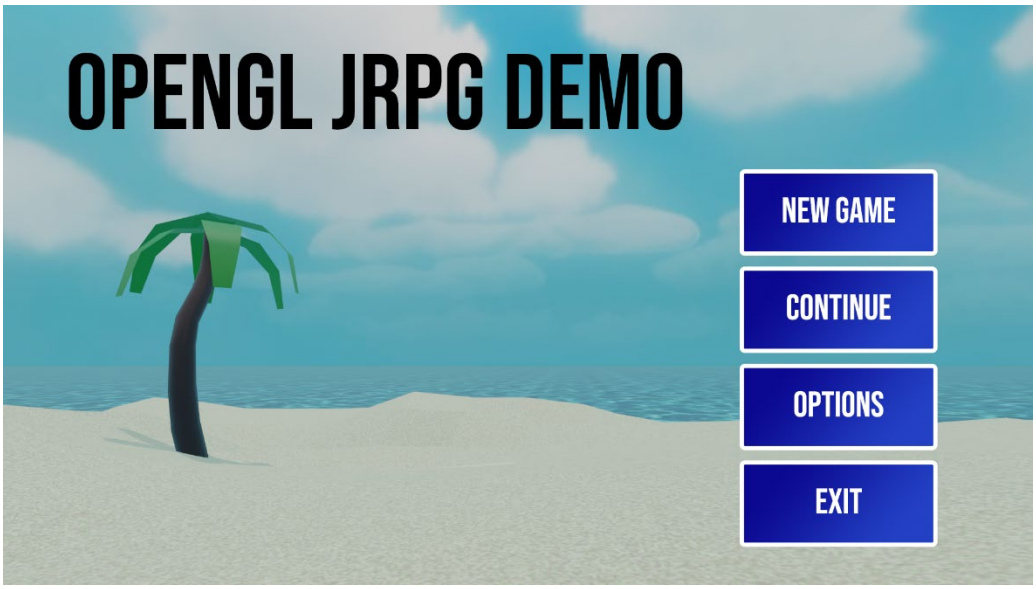
ohjataan kolmannesta persoonasta. Kamera on pelaaja hahmon yläpuolella ja takana, ja osoittaa hahmoa päin.

3.3 Taistelu

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

3.4 Päävalikko

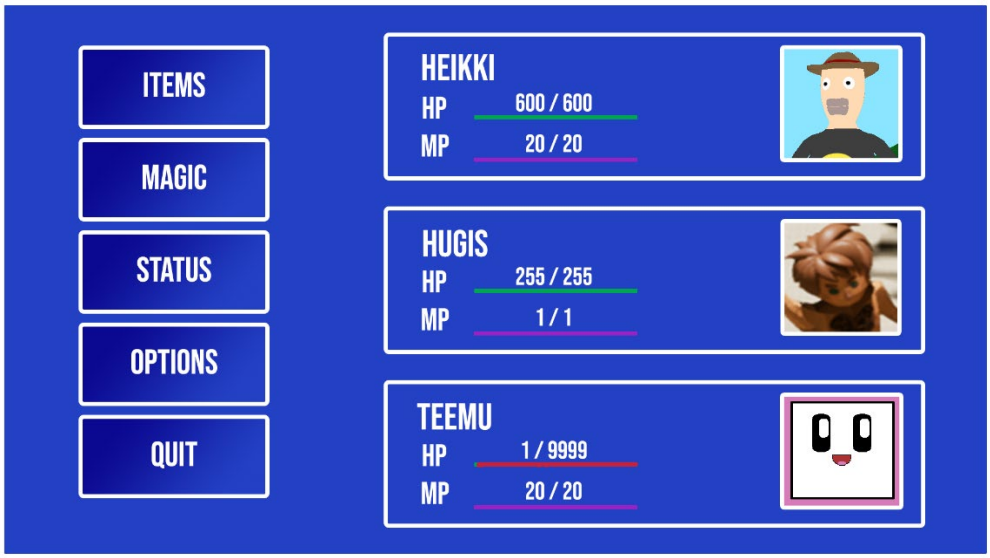
Päävalikossa on neljä eri painiketta: New Game-, Continue-, Options-, sekä Exit Game -painikkeet. New Game aloittaa pelin tarinan alusta ja lataa ensimmäisen kentän. Continue-painikkeesta pääsee tarkastelemaan tallennuksia, sekä jatkamaan peliä tallennuksesta, jolloin peli lataa tallennuksesta oikeat 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. Luonnos päävalikosta näkyy kuvassa 5



Kuva 5. Luonnos päävalikosta

3.5 Valikko

Valikkoon pääsee pelissä kentällä sekä maailmankartalla painamalla valikkonäppäintä. 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 tilastoja, 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. Luonnos valikosta näkyy kuvassa 6.



Kuva 6. Luonnos valikosta

4 TOTEUTUS

Tässä luvussa kerrotaan opinnäytetyön tavoitteista, ja siitä mitä lopullisessa pelidemossa kuuluisi olla. Kappaleessa käydään läpi vaihe vaiheelta, miten toteutetaan toiminnot yksinkertaista pelimoottoria varten. Aluksi kerrotaan, kuinka saadaan avattua OpenGL-kontekstin ikkuna. Sitten käydään läpi varjostinohjelman luominen ja piirretään näytölle kolmio. Seuraavaksi luodaan tekstuurien latausta varten luokka ja määritellään kamera kolmiulotteisten kap-

paleiden piirtoa varten. Lopuksi kerrotaan, kuinka peliin lisätään myös pelaajan liikkuminen ja äänentoisto sekä kuinka luodaan pelimaailmaan tapahtumia Event- ja EventHandler-luokkien avulla.

4.1 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 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.

4.2 Pelimaailman piirtäminen näytölle

Pelimaailman piirtäminen OpenGL-rajapinnan avulla vaatii, että määritellään OpenGL-kontekstin ikkuna sekä vähintään yksi verteksivarjostin ja pikselivarjostin. OpenGL-rajapinnalle pitää myös syöttää tieto kappaleesta, joka halutaan piirtää. Seuraavissa luvuissa kerrotaan, kuinka saadaan piirrettyä kaksi- ja kolmiulotteisia kappaleita OpenGL-rajapinnalla.

4.2.1 Ikkunan luonti

Kuvassa 7 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 2014d.)

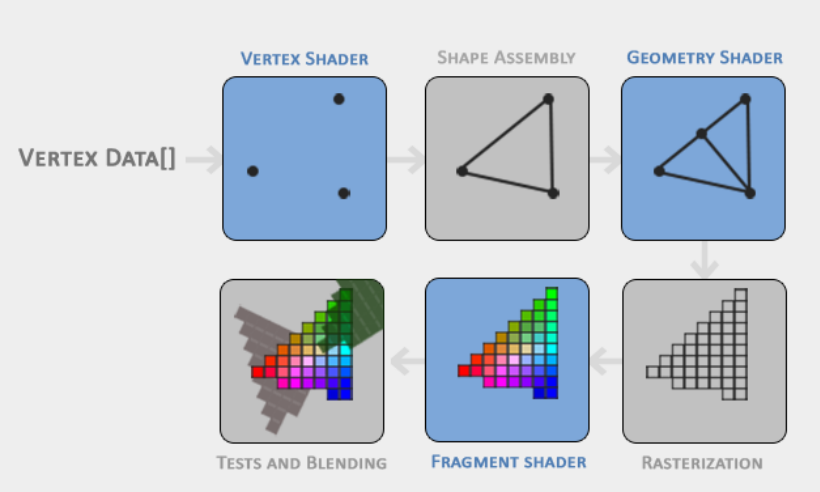
```
    // Initialize GLFW and configure
    glfwInit();
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 4);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 6);
    glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);

    // Create the window and set the width and height
    GLFWwindow* window = glfwCreateWindow(SCREENWIDTH, SCREEN-
HEIGHT, "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 de-
fines 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 7. GLFW-Ikkunan luonti

4.2.2 Varjostinohjelma

Varjostinohjelma kertoo OpenGL-rajapinnan ”grafiikkaliukuhihnalle” *(*engl. rendering pipeline*)*, mitä näytölle piirretään. Varjostinohjelmalle annetaan sarja 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), pikselivarjostin (engl. fragment shader) sekä geometriavarjostin (engl. geometry shader). Verteksivarjostin sekä pikselivarjostin ovat molemmat pakollisia OpenGL:än osia piirtoa varten, geometriavarjostin taas ei. (Learn OpenGL 2014c)

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 pikselivarjostin saa tiedon kolmion välisistä pikseleistä käyttöönsä. Ennen kuin pikselivarjostin tekee tehtävänsä, tapahtuu leikkaus, jolloin hävitetään pikselit, jotka eivät ole näkökentässä. Pikselivarjostimessa lasketaan pikseleiden lopulliset värit ja lasketaan esimerkiksi valojen vaikutus malliin. (Emt.)



Kuva 8. OpenGL-rajapinnan piirtoprosessi (Learn OpenGL 2014)

GLSL on hyvin samantapaista C-ohjelmointikielen kanssa, ja siihen on myös otettu hieman osia C++ -kielestä. GLSL-varjostimissa on mahdollista käyttää kuvan 9 mukaisia tietotyyppejä. Varjostimissa on myös mahdollista antaa muuttujille kuvan 10 mukaisia määritteitä, joilla kerrotaan esimerkiksi, onko muuttuja syöte vai tuloste.

Base Type	2D vec	3D vec	4D vec	Matrix Types		

`float`	`vec2`	`vec3`	`vec4`	`mat2`
				`mat2x2`
				`mat3x2`
				`mat4x2`
					`mat3`
					`mat2x3`
					`mat3x3`
					`mat4x3`
						`mat4`
						`mat2x4`
						`mat3x4`
						`mat4x4`

`double`	`dvec2`	`dvec3`	`dvec4`	`dmat2`
				`dmat2x2`
				`dmat3x2`
				`dmat4x2`
					`dmat3`
					`dmat2x3`
					`dmat3x3`
					`dmat4x3`
						`dmat4`
						`dmat2x4`
						`dmat3x4`
						`dmat4x4`

`int`	`ivec2`	`ivec3`	`ivec4`	–		
`uint`	`uvec2`	`uvec3`	`uvec4`	–		
`bool`	`bvec2`	`bvec3`	`bvec4`	–		

Kuva 9. GLSL-ohjelmointikielen perus-, vektori- sekä matriisitietotyypit (Shreiner et al. 2013, 40)

Type Modifier	Description
`const`	Labels a variable as a read-only. It will also be a compile-time constant if its initializer is a compile-time constant.
`in`	Specifies that the variable is an input to the shader stage.
`out`	Specifies that the variable is an output from a shader stage.
`uniform`	Specifies that the value is passed to the shader from the application and is constant across a given primitive.
`buffer`	Specifies read-write memory shared with the application. This memory is also referred to as a *shader storage buffer*.
`shared`	Specifies that the variables are shared within a local work group. This is only used in compute shaders.

Kuva 10. GLSL-ohjelmointikielen tyyppimääritteet (Shreiner et al. 2013, 46)

Opinnäytetyössä verteksivarjostinohjelmat on kirjoitettu vs-päätteisiin (vertex shader) tiedostoihin, ja pikselivarjostinohjelmat fs-päätteisiin (fragment shader) tiedostoihin. Yksinkertaisimmillaan verteksi- ja pikselivarjostimet näyttävät kuvan 11 ja 12 mukaisilta. Verteksivarjostimeen annetaan pisteet kolmiulotteisessa koordinaatistossa, ja pikselivarjostimeen annetaan väriarvo, jolla halutaan pikselit 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 11. Yksinkertainen verteksivarjostin

```
// Shader with color and only positions.
#version 460 core
out vec4 FragColor;

uniform vec4 myColor;

void main()
{
    FragColor = vec4(myColor);
}
```

Kuva 12. Yksinkertainen pikselivarjostin

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 13 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 pikselivarjostimen luominen tapahtuu suurimmaksi osin samalla tavalla ja samoilla funktioilla. (Learn OpenGL 2014c.)

```
    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" << in-
foLog << std::endl;
    }
```

Kuva 13. Varjostimen luonti

Kun varjostimet on luotu, täytyy vielä luoda varjostinohjelma, joka on viimeinen linkitetty versio yhdistetyistä varjostimista. Tämä prosessi näkyy kuvassa 14. 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, mikä 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" << in-
foLog << std::endl;
    }

    // Activate shader.
glUseProgram(shaderProgram);

    // Delete shader objects.
    glDeleteShader(vertexShader);
    glDeleteShader(fragmentShader);
```

Kuva 14. 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 muuttujan 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 15 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);
}
```

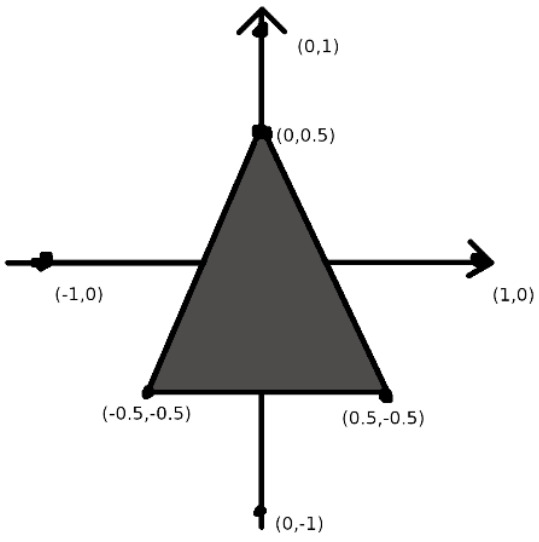
Kuva 15. Kokonaisluvun asettaminen Shader-luokassa

4.2.3 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 16 mukaisesti. Jokaisella rivillä on tieto esimerkiksi pisteen sijainnista, tekstuurikoordinaateista sekä normaalien koordinaateista. OpenGL käsittelee verteksien koordinaatteja normalisoituna laitekoordinaattivälillä, mikä 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 2014c.)

```
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 16. Kolmion verteksit



Kuva 17. Kolmio normalisoiduilla laitekoordinaateilla (Learn OpenGL 2014)

Kun mallin koordinaatit on määritelty liukulukutaulukossa, täytyy grafiikkakortista varata muistia näitä verteksitietoja varten ja tallettaa tiedot vertex buffer -objektiin (VBO). VBO 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. Puskuriobjektimuuttujaan 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.

Vertex array object (VAO) on myös pakollinen OpenGL-rajapinnassa, ja se luodaan samalla tavoin kuin VBO. VAO sisältää verteksitiedot glVertexAttribPointer-funktion kutsuista, ja sen avulla on mahdollista nopeasti vaihtaa verteksitietoja. Kun myös VAO 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 18.

```
    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);
    glVertexAt-
tribPointer(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 18. Vertex buffer -objektin luominen ja tiedon lähettäminen sekä kolmion piirtäminen

On myös olemassa element buffer object (EBO), jonka avulla saadaan piirrettyä neliö tai suorakulmio paljon tehokkaammin. Ilman element buffer -objektia suorakulmio täytyisi määritellä kahden kolmion eli kuuden verteksin avulla. EBO mahdollistaa suorakulmion piirron käyttämällä vain neljää verteksiä, sekä kokonaislukutaulukkoa, joka kertoo verteksien piirtojärjestyksen. Esimerkki EBO:n käytöstä löytyy kuvasta 19. (Learn OpenGL 2014c.)

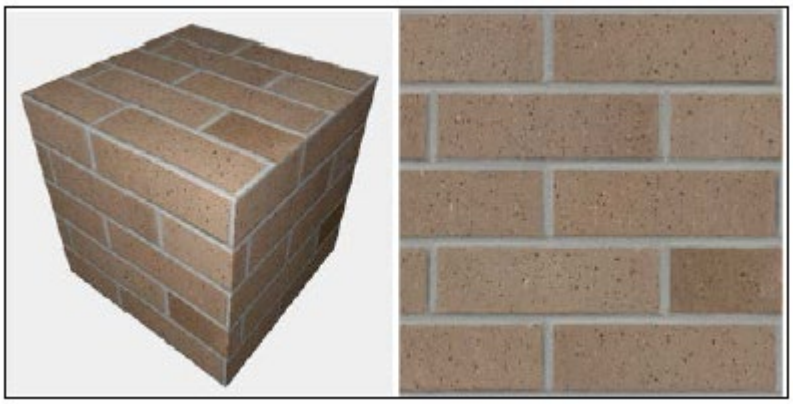
```
    float vertices[] =
    { // Positions
        0.5f,  0.5f, 0.0f,  // top right
        0.5f, -0.5f, 0.0f,  // bottom right
        -0.5f, -0.5f, 0.0f,  // bottom left
        -0.5f,  0.5f, 0.0f   // top left
    };
    unsigned int indices[] =
    {
        // The order to draw the vertices
        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), indi-
ces, GL_STATIC_DRAW);

    // Drawing
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
    glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);
```

Kuva 19. Element buffer -objektin käyttö

Opinnäytetyössä jokaisella piirrettävällä mallilla ja peliobjektilla on oma VBO, VAO, sekä EBO. 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 silmukassa. Opinnäytetyöhön on myös määritelty GameObject-luokka, joka sisältää luokat kuutiolle sekä neliölle. Luokasta saa helposti luotua piirrettäviä kuutioita varjostimien testausta varten.

4.2.4 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 20. 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 21 näkyy stb_image-otsikkotiedoston käyttö tiedon lataamiseen kuvasta. (Learn OpenGL 2014f.)

```
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 21. 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 22 mukaiselta.

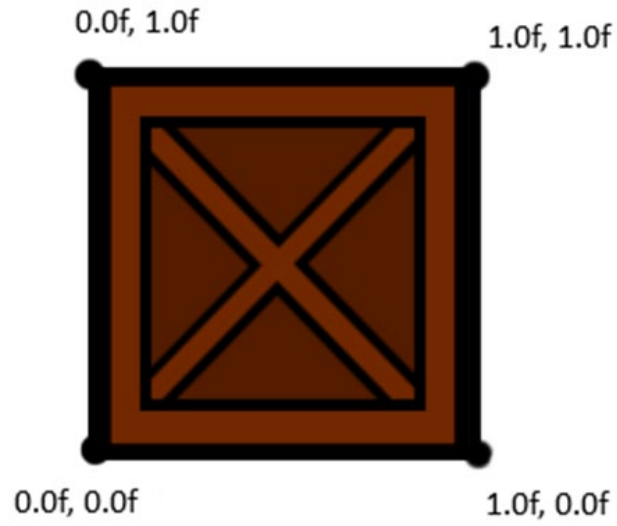
```
// Creating a texture
    unsigned int texture;
    glGenTextures(1, &texture);
    glBindTexture(GL_TEXTURE_2D, texture);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB, GL_UN-
SIGNED_BYTE, data);
    glGenerateMipmap(GL_TEXTURE_2D);
```

Kuva 22. Tekstuurin luominen

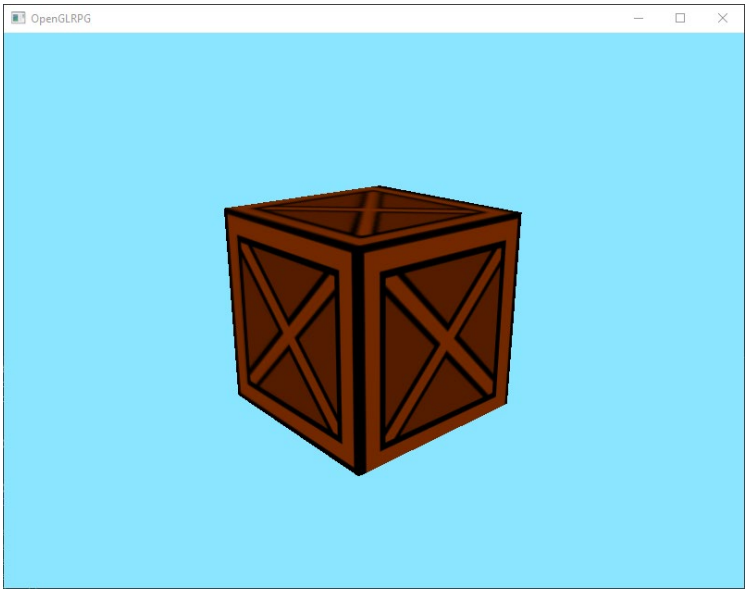
Tekstuuri on luotu, mutta tarvitaan tieto siitä, kuinka tekstuuri kääriytyy kolmiulotteisen mallin ympärille. Tämä hoituu teksturikoordinaateilla, jotka yleensä annetaan verteksien mukaan, siten että jokaisella verteksillä on oma x- ja y -akselin tekstuurikoordinaatti. Kuvassa 23 on neliön määrittely tekstuurikoordinaattien kanssa ja kuvassa 24 näkyy, kuinka kuva asettuu neliön sisälle täyttäen koko neliön näillä tekstuurikoordinaattiarvoilla. Kuvassa 25 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 23. Neliön määrittely tekstuurikoordinaattien kanssa



Kuva 24. Neliön tekstuurikoordinaatit



Kuva 25. Teksturoitu kuutio

Varjostimeen täytyy määritellä sampler2D-tyyppinen muuttuja, joka on GLSL-kieleen sisäänrakennettu tietotyyppi tekstuuriobjekteille. Jotta saadaan piirrettyä luotu tekstuuri värin sijaan, pitää pikselivarjostimessa kutsua funktiota texture, jolla annetaan tekstuuriobjekti sekä tekstuurikoordinaatit. Esimerkki yksinkertaisesta verteksi- ja pikselivarjostimesta tekstuurien kanssa näkyy kuvissa 26 ja 27.

```
#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 26. Verteksivarjostin tekstuureilla

```
#version 460 core
out vec4 FragColor;

in vec2 TexCoord;

uniform sampler2D ourTexture;

void main()
{
    FragColor = texture(ourTexture, TexCoord);
}
```

Kuva 27. Pikselivarjostin tekstuureilla

Varjostinta käytettäessä täytyy vain ennen piirtokutsua muistaa kiinnittää oikea tekstuuri OpenGL-rajapintaan glBindTexture-funktion kutsulla, jolloin pikselivarjostin 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.

4.3 Kameran määrittely

Kamera käyttäytyy pelin tilasta riippuen eri tavoin. Kentällä ollessa pelaaja ei pysty itse vaikuttamaan kameraan, ja kamera automaattisesti liikkuu hieman pelaajan mukana tietyissä 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 2013, 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.)

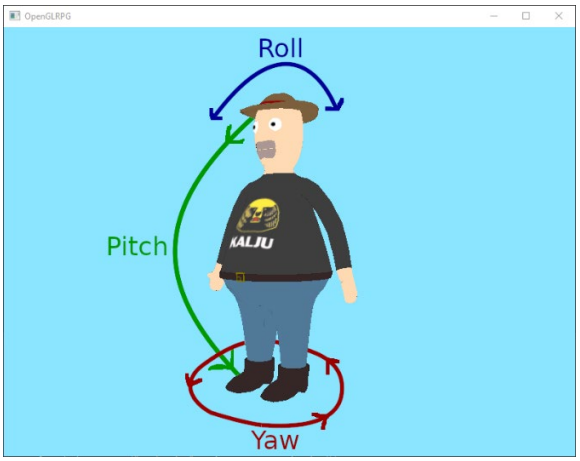
View-matriisi saadaan luotua käyttämällä GLM-ohjelmakirjaston funktiota lookAt, jolle annetaan parametrina aiemmin määritellyt kameran koordinaatiston vektorit, eli kamerasta katsottuna eteen, ylös ja oikealle osoittavat vektorit. Funktio palauttaa 4x4-matriisin, joka sijoitetaan kamera-luokan matriisimuuttujaan 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 28. (Learn OpenG 2014a.)

```
// Camera definitions
    // Positive z-axis goes towards the screen and to-
wards 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 prod-
uct 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 func-
tion the camera position,
    // target position and world vector up.
    view = glm::lookAt(cameraPos, cameraPos + cameraFront, cameraUp);

    projection = glm::perspective(glm::radians(45.0f), (float)screen-
Width / (float)screenHeight, 0.1f, 10000.0f);
```

Kuva 28. Kameran konstruktorifunktio

Opinnäytetyössä kameraa haluttiin myös kääntää, jotta on helpompi tarkastella ja testata kolmiulotteisia malleja. Eulerin kulmilla tarkoitetaan matemaattista esitysmuotoa kappaleen asennolle kolmen kulman avulla (Goldstein 1980, 143–148). Kameran kääntämistä varten täytyy määritellä nämä Eulerin 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 29 näyttää, kuinka Eulerin kulmat vaikuttavat kappaleen rotaatioon.



Kuva 29. Roll-, pitch-, ja yaw -rotaatiot

Kun kameraa liikutetaan, täytyy kameran view-matriisi laskea uudestaan. Kameran liikuttaminen tapahtuu opinnäytetyössä funktiolla MoveCamera, jolle annetaan parametrinä suunta. MoveCamera-funktio näkyy kuvassa 30.

```
void Camera::MoveCamera(Direction dir)
{
    switch (dir)
    {
    case Direction::Forward:
        cameraPos += cameraSpeed * cameraFront;
        break;

    case Direction::Right:
        cameraPos += glm::normalize(glm::cross(cameraFront, cameraUp)) * camer-
aSpeed;
        break;

    case Direction::Back:
        cameraPos -= cameraSpeed * cameraFront;
        break;

    case Direction::Left:
        cameraPos -= glm::normalize(glm::cross(cameraFront, cameraUp)) * camer-
aSpeed;
        break;

    default:
        break;
    }
    view = glm::lookAt(cameraPos, cameraPos + cameraFront, cameraUp);
}
```

Kuva 30. 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 31.

```
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, be-
cause their length gets closer to 0 the more you look up or down which re-
sults in slower movement.
    cameraUp = glm::normalize(glm::cross(cameraRight, cameraFront));
    view = glm::lookAt(cameraPos, cameraPos + cameraFront, cameraUp);
}
```

Kuva 31. 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 32.

```
    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 32. Model-matriisin käyttö

Jotta model-, view- sekä projection-matriiseja voidaan käyttää pelissä, pitää kirjoittaa varjostin, joka käyttää näitä matriiseja. Pikselivarjostimeen ei tarvitse tehdä Model View Projection -mallia varten muutoksia. Verteksivarjostimeen luodaan muuttujat matriiseja varten, ja varjostimelle syötetyt verteksisijainnit kerrotaan niillä. Esimerkki verteksivarjostimesta, joka käyttää näitä matriiseja hyödyksi on kuvassa 33.

```
// 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 33. Verteksivarjostin Model View Projection -matriisien kanssa.

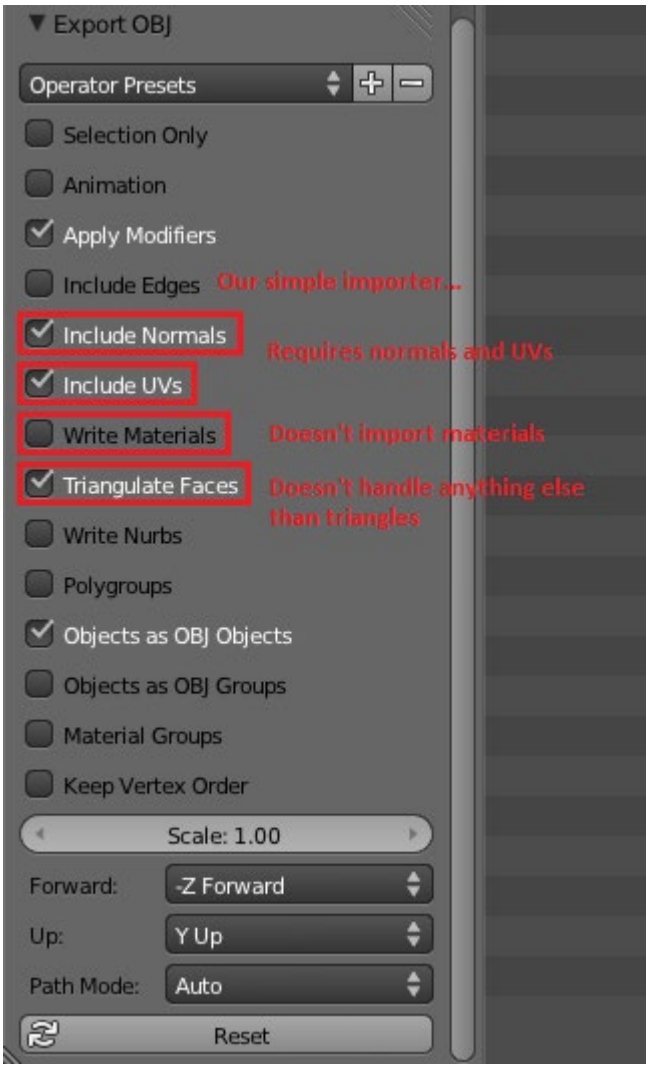
Pääohjelmassa varjostimelle annetaan matriisit kutsumalla Shader-luokan funktiota SetMat4, jolle annetaan parametrina muuttujan nimi varjostimessa sekä itse matriisi. Shader-luokan SetMat4-funktio kutsuu glad-ohjelmistokirjaston funktiota glUniformMatrix4fv, joka vie tiedon matriisista varjostimelle. Esimerkki matriisien viemisestä varjostimelle on kuvassa 34.

```
        colorNShader.Use();
        colorNShader.SetMat4("view", mainCamera.GetView());
        colorNShader.SetMat4("projection", mainCamera.GetProjection());
```

Kuva 34. Matriisien vieminen varjostimelle

4.4 3D-mallien lataaminen

Kolmiulotteisten mallien verteksien määrittely käsin on työlästä. Tästä syystä mallit luodaan opinnäytetyössä 3D-mallinnusohjelmassa, Blenderissä. Mallit viedään kuvan 35 asetusten mukaisesti obj-tiedostoon, jotta ne saadaan ladattua oikein opinnäytetyössä.



Kuva 35. 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 36. 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 silmukassa kutsutaan rekursiivisesti ProcessNodea, jolle annetaan parametrina tämänhetkisen noden lapsinodet. Noden meshit käydään läpi ensimmäisessä silmukassa, 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 37.

```
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 37. ProcessNode-funktio

Kun ProcessNode on päässyt rekursiivisesti loppuun, ja kaikki nodet on käyty läpi, voidaan malli piirtää. Mallin piirto tapahtuu mallin Draw-funktiossa, jossa käydään mesh-lista läpi silmukassa ja kutsutaan jokaiselle yksittäiselle meshille Mesh-luokasta löytyvää Draw-funktiota. Mesh-luokan Draw-funktiossa on määritelty itse meshin piirtokoodi. Kuvassa 38 näkyy obj-malli, joka on ladattu käyttäen assimp-ohjelmakirjastoa.

Kuva 38. Malli assimp-kirjastolla ladattuna

4.5 Drawable-luokka

Drawable-luokka on abstrakti luokka, joka sisältää vain yhden virtuaalifunktion Draw. Kaikki opinnäytetyön luokat, jotka halutaan piirtää ruudulle, perivät Drawable-luokan ja sisältävät toteutuksen Draw-funktiolle, jossa piirtokoodi on määritelty. Pääohjelmassa määritellään listamuuttuja, johon kaikki Drawable-luokkaa käyttävät oliot lisätään. Näin voidaan pääsilmukassa helposti kutsua kaikkien piirrettävien olioiden Draw-funktiota omassa silmukassa. Kuvassa 39 näkyy Drawable-luokan toteutus ja kuvassa 40 näkyy Drawable-listan käyttö pääohjelmassa.

```
class Drawable
{
public:
    virtual void Draw() = 0;
};
```

Kuva 39. Drawable-luokan määrittely

```
    Drawable* player = new Model((pathToRoot + "/Game/Mod-
els/hugi/Hugis.obj").c_str(), (pathToRoot + "/Game/Models/Cube/Col-
ors.png").c_str(),
    unlitShader, glm::vec3(4.0, -1.0, 1.0), glm::vec3(0.005), false);


    Drawable* barrel = new Model((pathToRoot + "/Game/Models/Barrel/bar-
rel.obj").c_str(), (pathToRoot + "/Game/Models/Barrel/barrel.png").c_str(),
    unlitShader, glm::vec3(8.0, -0.5, 3.0), glm::vec3(0.5), false);


    // List of drawables
    std::vector<Drawable*> drawables;
    drawables.push_back(player);
    drawables.push_back(barrel);


    // Main loop
    while(!glfwWindowShouldClose(window))
    {
        unlitShader.Use();
        unlitShader.SetMat4("view", mainCamera.GetView());
        unlitShader.SetMat4("projection", mainCamera.GetProjection());


        // Loop for drawables
        for(auto &it : drawables)
        {
            (it)->Draw();
        }


        // Check and call events and swap buffers
        glfwSwapBuffers(window);
        glfwPollEvents();
    }
```

Kuva 40. Drawable-lista pääohjelmassa

4.6 Käyttäjän syöte

Opinnäytetyössä InputHandler-luokka hallitsee käyttäjän syötteen. Luokka hyödyntää singleton-mallia, sillä halutaan, että luokasta on vain yksi instanssi, johon päästään helposti käsiksi. InputHandler-luokka tarvitsee tiedon ikkunasta, kamerasta, sekä pelaajasta, jotta näitä voidaan ohjata näppäimistöllä. Nämä tiedot annetaan luokalle ennen pääsilmukkaan menemistä pääohjelmassa Initialize-funktiokutsulla, jolla tiedot viedään parametreina. Pääsilmukan alussa kutsutaan funktiota ProcessInput, johon määritellään, mitä minkäkin näppäimen painalluksen halutaan tekevän. ProcessInput-funktiossa tarkistetaan, mitä näppäintä on painettu ehtolauseilla. Esimerkki ruudun sulkemisesta Esc-näppäintä painamalla näkyy kuvassa 41.

```
if (glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
{
    glfwSetWindowShouldClose(window, true);
}
```

Kuva 41. Syötteen tarkistus

GLFW-ohjelmistokirjaston glfwGetKey-funktiolle annetaan parametreinä tämänhetkinen ikkuna sekä näppäimistön näppäin, jonka tila halutaan tietää. Ehtolauseessa katsotaan, onko näppäimen tila GLFW_PRESS, joka tarkoittaa, että näppäintä on painettu. Ruutu suljetaan funktiolla glfwSetWindowShouldClose, jos näppäintä on painettu.

4.7 Tekstin renderöinti

OpenGL-rajapinta ei tarjoa tekstin näyttämiseen työkaluja, joten ohjelmoijan on hoidettava tämä itse. On mahdollista tehdä suuri kuva, jossa jokainen kirjain on järjestelty peräkkäin, josta kirjaimia sitten näytettäisiin kuvina. Tekstin näyttämiseen ja fonttien lataukseen on nykyisin helpompia sekä tehokkaampia tapoja. Opinnäytetyössä käytetään FreeType-ohjelmakirjastoa, joka mahdollistaa TrueType fonts (ttf) -tyyppisten fonttitiedostojen latauksen koodissa, sekä niiden renderöinnin bittikarttoihin. (Learn OpenGL 2014e.)

Fontin lataamiseksi alustetaan FT_Library-tyypin muuttuja kutsumalla funktiota FT_Init_Freetype. Funktio palauttaa kokonaisluvun, ja jos tämä kokonaisluku on nolla, alustuksessa on tapahtunut virhe. FreeType-fontti ladataan

muuttujaan, joka on tyyppiä FT_Face. Tämä tapahtuu funktiolla FT_New_Face, joka myös palauttaa nollan virheen tapahtuessa. Tämän jälkeen määritellään, minkä kokoisena fontit ladataan tästä FT_Face-muuttujasta funktiolla FT_Set_Pixel_Sizes. (Emt.)

FT_Face-muuttuja koostuu erikokoisista kirjoitusmerkeistä. Kirjoitusmerkit on hyvä ladata aluksi muuttujaan, sillä olisi raskasta tehdä lataus joka ruudunpäivityksellä. Kirjoitusmerkin säilytystä varten määritellään koodissa Character-rakenne sekä taulukko, johon kaikki kirjaimet tallennetaan Character-muuttujina. Character-rakenne näkyy kuvassa 42.

```
struct Character
{
    GLuint     TextureID;  // ID handle of the glyph texture
    glm::ivec2 Size;       // Size of glyph
    glm::ivec2 Bearing;    // Offset from baseline to left/top of glyph
    GLuint     Advance;    // Offset to advance to next glyph
};
```

Kuva 42. Character-rakenne

Ttf-tiedostosta ladataan kaikki kirjaimet taulukkoon Font-luokan LoadCharacterSet-funktiolla. Font-luokka hyödyntää singleton-mallia, jotta siihen päästään helposti käsiksi mm. pääohjelmasta sekä Event-luokasta. LoadCharaterSet-funktiossa mennään silmukkaan, joka pyörii 128 kertaa, ladaten ensimmäiset 128 kirjainta ttf-tiedostosta. Silmukan alussa kutsutaan FT_Load_Char-funktiota, joka lataa kirjoitusmerkin muuttujaan. Kirjoitusmerkille luodaan tekstuuri ja kirjoitusmerkki säilötään taulukkoon myöhempää käyttöä varten. Kun silmukka on käyty läpi, luodaan tekstin näyttämistä varten vielä verteksipuskuriobjekti sekä verteksitaulukko-objekti ja viedään tieto tarvittavista vertekseistä OpenGL-rajapinnalle glBufferData-funktion kutsulla. Funktiolla viedään kuusi verteksiä, joista jokainen pitää sisällään tiedon sijainneista sekä tekstuurikoordinaateista. Koska teksti on kaksiulotteista, tulee lopullinen koko tiedolle joka viedään, olemaan sizeof(GLfloat)*6*4, sillä luvut ovat liukulukuja. (emt.)

Kirjoitusmerkit on ladattu muistiin, mutta täytyy vielä tehdä funktio, joka renderöi halutut merkit näytölle sekä varjostin tekstiä varten. Verteksivarjostimelle viedään neljä liukulukua, joista kaksi ensimmäistä kertoo sijainnin, ja kaksi vii-

meistä kertoo tekstuurikoordinaatit. Varjostimelle viedään myös vain projektiomatriisi, sillä halutaan että teksti näkyy aina ruudulla, eli perspektiiviä ei tarvita. Pikselivarjostimelle viedään kuva kirjaimesta sekä väri, jolla kirjain halutaan näyttää. Pikselivarjostimessa muutetaan kirjaimen kuvasta tausta läpinäkyväksi käyttämällä tekstuurin punaista värikomponenttia läpinäkyvyytenä. Jotta nämä tekstivarjostimet toimivat oikein, täytyy vielä pääohjelmassa laittaa värien sekoitus päälle kuvan 43 mukaisilla funktiokutsuilla. Fonttien renderöintiä varten kirjoitetut varjostimet näkyvät kuvissa 44 ja 45.

```
    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

Kuva 43. Värien sekoitus päälle

```
#version 460 core
layout (location = 0) in vec4 vertex; // vec2 pos, vec2 tex
out vec2 texCoord;

uniform mat4 projection;

void main()
{
    gl_Position = projection * vec4(vertex.xy, 0.0, 1.0);
    texCoord = vertex.zw;
}
```

Kuva 44. Font-verteksivarjostin

```
#version 460 core
out vec4 FragColor;
in vec2 texCoord;

uniform sampler2D text;
uniform vec3 textColor;

void main()
{
    vec4 sampled = vec4(1.0, 1.0, 1.0, texture(text, texCoord).r);
    FragColor = vec4(textColor, 1.0) * sampled;
}
```

Kuva 45. Font-pikselivarjostin

Tekstin renderöintiä varten määritellään funktio RenderText, jolle annetaan parametreinä näytettävä teksti, tekstin sijainti näytöllä, tekstin koko sekä väri. Funktio vie aluksi tiedon väristä varjostimelle. Tämän jälkeen käydään silmukalla läpi teksti kirjain kerrallaan, lasketaan jokaiselle kirjaimelle oikea sijainti ja viedään kirjaimen tekstuuri sekä sijainti ja tekstuurikoordinaatit OpenGL-rajapinnan käyttöön. Nyt voidaan tekstiä näyttää pelissä kuvan 46 mukaisella funktiokutsulla. Kuvassa 47 näkyy teksti peliruudulla.

```
Font::GetInstance().RenderText(*textShader, ”Barrel is too heavy!”, 20.0f,
20.0f, 0.5f, glm::vec3(1.0f, 1.0f, 1.0f));
```

Kuva 46. RenderText-funktio

Kuva 47. Teksti peliruudulla

4.8 Liikkumisen ja interaktiivisuuden lisääminen peliin

Jotta peliä voi pelata, on peliin lisättävä pelaajahahmon liikutus sekä mahdollisuus vaikuttaa kentällä oleviin esineisiin. Mallin liikutus onnistuu Model-matriisin avulla. Interaktiivisuutta varten täytyy määritellä pelikentälle alue, jossa ta-

pahtuman halutaan tapahtuvan. Tämä tehdään Collider-luokan avulla. Tapahtumat määritellään Event-luokassa ja EventHandler-luokka hallitsee kaikkia tapahtumia.

4.8.1 Mallin liikuttaminen

Pelaajamallin liikuttaminen on todella helppoa Model-luokkaan määriteltyä Model-matriisia sekä position-muuttujaa käyttäen. Mallia liikutetaan Model-luokan funktiolla Move, joka ottaa parametreikseen suuntavektorin sekä nopeuden, joka kertoo, kuinka paljon mallia liikutetaan. Suuntavektori normalisoidaan ja kerrotaan nopeudella, ja tämän laskun tulos lisätään position-muuttujaan. Draw-funktiossa mallin liikutus tapahtuu GLM-ohjelmakirjaston funktiolla ”translate”, jolle annetaan parametreiksi Model-matriisi ja position-muuttuja. Funktio palauttaa translaatiomatriisin, joka sijoitetaan Model-matriisiin ja viedään varjostimelle. Kuvassa 48 näkyy Move-funktio ja kuvassa 49 InputHandler-luokassa Move-funktion kutsuminen, kun pelaaja on painanut W-näppäintä näppäimistöllä.

```
void Move(glm::vec3 direction, float speed = 0.05f)
{
    position += glm::normalize(direction) * speed;
}
```

Kuva 48. Model –luokan Move –funktio

```
if (glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS)
{
    player->Move(glm::vec3(1.0, 0.0, 0.0));
}
```

Kuva 49. Pelaajan Move-funktion kutsuminen InputHandler-luokassa

4.8.2 Collider-luokka

Pelissä halutaan, että kentällä oleville hahmoille pystyy juttelemaan ja pelimaailmassa olevia esineitä pystyy tutkimaan. Tätä varten täytyy määritellä alue, jonka sisällä ollessa pelaaja voi painaa näppäintä tehdäkseen toiminnon. Opinnäytetyössä toteutetaan yksinkertainen Collider-luokka, joka määrittelee kahden pisteen avulla alueen, jonka sisällä voidaan tehdä toiminto. Collider-luokan konstruktorifunktiolle annetaan minimi- ja maksimipisteet, jotka rajaavat halutun alueen. Collider-luokka sisältää funktion CheckPoint, jolle annetaan parametrinä vektori. Funktio vertaa vektorin x-, y- ja z-arvoja minimi- ja

maksimipisteiden x-, y- ja z-arvoihin, ja palauttaa arvon true, jos parametrinä annetun pisteen arvot ovat suurempia kuin minimipisteen arvot ja pienempiä kuin maksimipisteen arvot. Muussa tapauksessa funktio palauttaa arvon false.

4.8.3 Event-luokka

Event-luokka määrittelee kentällä tapahtuman. Event-luokassa on muuttujina Collider, joka määrittelee alueen, jossa tapahtuma tapahtuu sekä lista teksteistä, jotka halutaan piirtää ruudulle, kun pelaaja painaa välilyöntiä tapahtuman alueella. Luokassa on myös textIndex kokonaislukumuuttuja, jonka alkuarvona on -1. Tämä muuttuja määrittelee, mikä teksti näytetään tekstilistasta. Event-luokan konstruktorille annetaan parametreinä sijainti, jossa tapahtuman halutaan tapahtuvan, varjostin tekstin piirtämiseen sekä näytettävä teksti, joka lisätään listaan tapahtuman teksteistä. Konstruktorissa luodaan uusi Collider tapahtumalle, jonka minimi- ja maksimipisteet määritellään sijaintiparametrin avulla vähentämällä ja lisäämällä sijaintiin luvun 1.

Event-luokassa on AddText-funktio, jolle annetaan parametrinä tekstiä. Tämä teksti lisätään funktiossa listaan tapahtuman teksteistä. Luokassa on myös RenderEventText-funktio, joka renderöi halutun tekstin ruudulle kutsumalla Font-luokan RenderText-funktiota kuvan 50 mukaisesti.

```
void RenderEventText()
{
    Font::GetInstance().RenderText(*textShader, eventText[textIndex], 20.0f, 20.0f, 0.5f, glm::vec3(1.0f, 1.0f, 1.0f));
}
```

Kuva 50. RenderEventText-funktio

Event-luokassa on myös määritelty DoEvent-funktio, joka nostaa textIndex-muuttujan arvoa yhdellä, jos muuttujan arvo ei ole sama kuin eventText-listan viimeinen indeksi. Näin voidaan DoEvent-funktiolla muuttaa tekstiä, ja siirtyä tekstilistassa seuraavaan tekstiin.

4.8.4 EventHandler-luokka

EventHandler-luokka on vastuussa kaikista kentän tapahtumista. Luokassa on lista kaikista Event-luokan olioista, kokonaisluku joka kertoo tämänhetkisen ta-

pahtuman indeksin sekä kokonaisluku, jota käytetään ajan laskemiseen. Luokassa on AddEvent-funktio, jolla Event-olio lisätään listaan tapahtumista. Luokassa määriteltyä CheckPlayerEvents-funktiota kutsutaan, kun pelaaja painaa välilyöntiä. Funktio käy silmukassa läpi kaikki eventit ja tarkistaa, onko pelaaja yhdenkään tapahtuman colliderin sisällä. Jos pelaaja on tapahtumapaikan sisällä, kutsutaan kyseisen Event-olion DoEvent-funktiota, asetetaan currentEventIndex-muuttuja vastaamaan tämän tapahtuman indeksiä ja asetetaan textTimer-ajastimeen arvo 250. Funktiossa ShowingText tapahtuu tapahtuman tekstin piirtäminen. Tätä funktiota kutsutaan pääsilmukassa. Funktiossa on ehto, joka toteutuu, jos textTimer-muuttujan arvo on enemmän kuin nolla. Muuttujan arvoa vähennetään joka kutsulla ruudunpiirtoon kuluneen ajan verran, ja samalla kutsutaan tämänhetkisen tapahtuman RenderEventText-funktiota, joka piirtää tapahtuman tekstin näytölle. Kun ajastin saavuttaa arvon nolla, ei ehto toteudu, ja teksti katoaa. ShowingText-funktio näkyy kuvassa 51.

```
void ShowingText()
{
    if (textTimer > 0)
    {
        events[currentEventIndex]->RenderEventText();
        textTimer -= deltaTime;
    }
}
```

Kuva 51. ShowingText-funktio

4.9 Äänet

Opinnäytetyössä on toteutettu äänentoisto OpenAL-rajapintaa sekä freealut-ohjelmakirjastoa hyödyntäen. AudioManager-luokka hallitsee pelin ääniä, ja käynnistää taustamusiikin pelin alussa StartAudioManager-funktion kutsulla. AudioManager-luokka käyttää singleton-mallia, joten tämän funktiokutsun yhteydessä luodaan instanssi luokasta, joka pysyy ohjelman ajon aikana elossa. StartAudioManager-funktion alussa määritellään ALuint-tietotyyppiä olevat buffer- ja source-nimiset muuttujat sekä ALint-tyypin state-muuttuja. (The OpenAL Utility Toolkit 2006.)

Freealut-kirjastosta kutsutaan funktiota alutInit, joka alustaa ALUT-ohjelmakirjaston, luo OpenAL-kontekstin käytössä olevaan äänentoistolaitteeseen ja tekee tästä senhetkisen OpenAL-kontekstin. Alustuksessa tapahtuneet virheet tarkastetaan Audiomanager-luokan funktiolla CheckForErrors. Tämä funktio sijoittaa alGetError-funktion palauttaman virhekoodin muuttujaan, ja virheen tapahtuessa tulostaa ruudulle tiedon siitä. Kuvassa 52 näkyy CheckForErrors-funktion toiminta. (Emt.)

```
    ALCenum error;
    error = alGetError();
    if (error != AL_NO_ERROR)
    {
        std::cerr << "Error! Here is what happened: " << error << std::endl;
}
```

Kuva 52. CheckForErrors-funktio

Kun virheentarkistus on tehty, voidaan wav-päätteinen musiikkitiedosto ladata ja sijoittaa buffer-muuttujaan funktiolla alutCreateBufferFromFile. Tälle funktiolle annetaan parametriksi musiikkitiedoston polku, ja funktio yrittää tiedoston nimen ja sisällön perusteella ladata äänidatan ja vie sen OpenAL-puskurille. Kun tieto on puskuroitu OpenAL-rajapinnalle, täytyy vielä luoda äänilähde, jotta äänen voi toistaa. Tämä onnistuu funktiolla alGenSources, jolle annetaan parametreinä äänilähteiden määrä sekä muuttuja, johon äänilähde tallentuu. Äänilähteeseen puskuroidaan aiemmin ladattu äänidata funktiolla alSourcei, jolle annetaan parametriksi äänilähde, AL_BUFFER joka kertoo, että ollaan puskuroimassa äänidata, sekä itse puskuri. Tämän jälkeen säädetään alSourcei-funktiolla äänilähde toistumaan loputtomiin, sillä halutaan, että taustamusiikki pysyy soimassa. Äänilähde käynnistetään funktiolla alSourcePlay. StartAudioManager-funktio näkyy kuvassa 53. (Emt.)

```
    ALuint buffer, source;
    ALint state;

    // Initialize the environment
    alutInit(0, NULL);

    // Capture errors
    CheckForErrors();
    // Load pcm data into buffer
    buffer = alutCreateBufferFromFile("Sounds/Music/alenarag.wav");

    // Create sound source (use buffer to fill source)
    alGenSources(1, &source);
    alSourcei(source, AL_BUFFER, buffer);
    // Set looping true
    alSourcei(source, AL_LOOPING, 1);

    // Play
alSourcePlay(source);
```

Kuva 53. StartAudioManager-funktio

5 YHTEENVETO

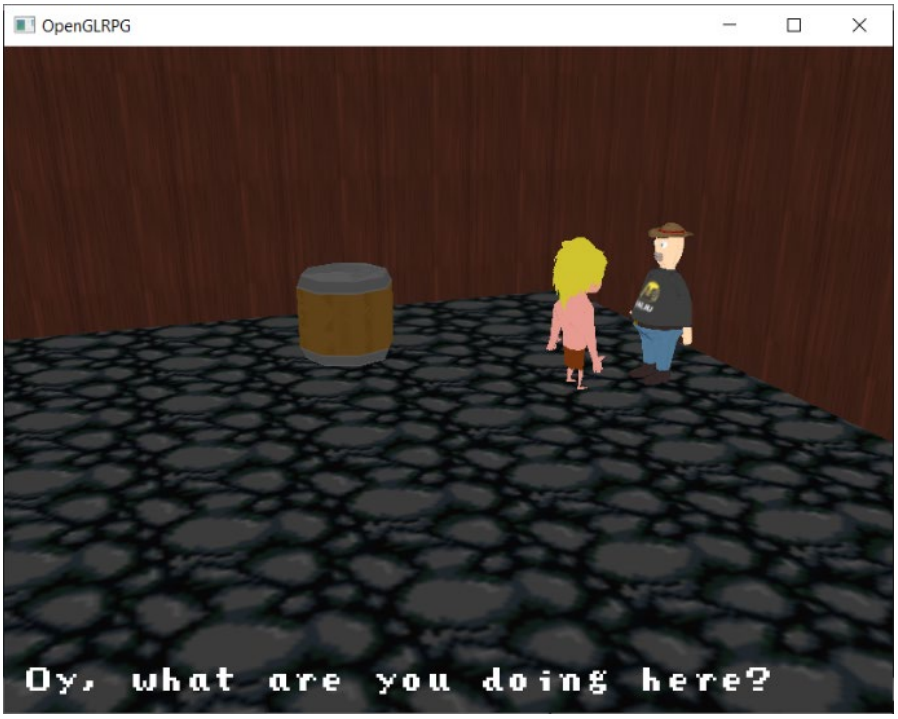
Suurin osa pelidemolle asetetuista tavoitteista saavutettiin. Pelidemoon ei ajan puitteissa keretty tekemään luokkaa, joka vastaisi kokonaan piirtämisestä, joten piirtäminen tapahtuu vielä pääohjelmassa suoraan. Peliin toteutettiin pelimoottorille tärkeimmät ominaisuudet, ja pelimoottoria on mahdollista muokata muihinkin peligenreihin sopivaksi.

Peliä on mahdollista jatkokehittää toteuttamalla Renderer-luokka, joka vastaa kokonaan kaiken piirtämisestä. Peliin olisi myös hyvä tehdä jonkinlainen kenttien hallinta, jotta voidaan näyttää eri malleja ja hahmoja eri kentillä.

Pelimoottorin tekeminen voi tuntua aluksi hankalalta ja haastavalta. On kuitenkin palkitsevaa saada toimintoja toimimaan, kuten ikkunan avaaminen tai kolmion piirto näytölle. Vaikka nämä toiminnot kuulostavat helpoilta, voi näiden toimintojen toteuttaminen tuntua haastavalta pelkällä grafiikkarajapinnalla.

Pelin ohjelmointi käyttäen OpenGL-rajapintaa sekä erilaisia ohjelmistokirjastoja hyödyntäen on ollut todella opettavainen kokemus ja herättänyt entisestään mielenkiintoa pelimoottorin ohjelmointiin. Pelimoottorin ohjelmointi on myös hidasta, sillä kaikki toiminnot pitää toteuttaa itse.

Uuden peli-idean tai pelidemon testausta varten on parempi tehdä testiversio valmiilla pelimoottorilla kuin tehdä koko pelimoottori alusta lähtien, ellei pelidemo vaadi jotakin, mitä ei valmiilla pelimoottorilla ole mahdollista toteuttaa. Peliohjelmoijan on kuitenkin hyvä tietää, miten pelimoottorit toimivat, joten oman pelimoottorin ohjelmointi on kannattavaa oppimisen kannalta.



Kuva 54. Lopullinen pelidemo

LÄHTEET

Assimp. 2018. The Open-Asset-Importer-Lib. WWW-dokumentti. Saatavissa: http://www.assimp.org/ [viitattu 13.11.2019].

Blender. 2019. About. WWW-dokumentti. Saatavissa: https://www.blender.org/about/ [viitattu 20.11.2019].

Cmake. 2019a. WWW-dokumentti. Saatavissa: https://cmake.org/ [viitattu 05.12.2019].

Cmake. 2019b. CMake Tutorial. WWW-dokumentti. Saatavissa: https://cmake.org/cmake/help/latest/guide/tutorial/index.html [viitattu 05.12.2019].

FileFormat. 2019. Wavefront OBJ File Format Summary. WWW-dokumentti. Saatavissa: https://www.fileformat.info/format/wavefrontobj/egff.htm [viitattu 20.11.2019].

FreeType. 2018. What is FreeType? WWW–dokumentti. Saatavissa: https://www.freetype.org/freetype2/docs/index.html [viitattu 28.10.2019].

Git. 2019. Git – About Version Control. WWW-dokumentti. Saatavissa: https://git–scm.com/book/en/v2/Getting–Started–About–Version–Control [viitattu 22.10.2019].

Github. 2019a. Hello World · GitHub Guides. WWW-dokumentti. Saatavissa: https://guides.github.com/activities/hello-world/ [viitattu 22.10.2019].

Github. 2019b. Stb. WWW-dokumentti. Saatavissa: https://github.com/nothings/stb [viitattu 06.11.2019].

GLFW. 2019. WWW-dokumentti. Saatavissa: https://www.glfw.org/ [viitattu 24.10.2019].

GLM. 2019. OpenGL Mathematics. WWW-dokumentti. Saatavissa: https://glm.g-truc.net/0.9.9/index.html [viitattu 17.10.2019].

Goldstein, H. 1980. Classical Mechanics. 2. painos. Massachusetts: Addison-Wesley.

Gregory, J. 2009. Game Engine Architecture. 1. painos. Wellesley: A K Peters/CRC Press.

Learn OpenGL. 2014a. Camera. WWW-dokumentti. Saatavissa: https://learnopengl.com/Getting-started/Camera [viitattu 23.11.2019].

Learn OpenGL. 2014b. Core-profile vs Immediate mode. WWW-dokumentti. Saatavissa: https://learnopengl.com/Getting-started/OpenGL [viitattu 11.10.2019].

Learn OpenGL. 2014c. Hello Triangle. WWW-dokumentti. Saatavissa: https://learnopengl.com/Getting-started/Hello-Triangle [viitattu 31.10.2019].

Learn OpenGL. 2014d. Hello Window. WWW-dokumentti. Saatavissa: https://learnopengl.com/Getting-started/Hello-Window [viitattu 28.10.2019].

Learn OpenGL. 2014e. Text Rendering. WWW-dokumentti. Saatavissa: https://learnopengl.com/In–Practice/Text–Rendering [viitattu 31.10.2019].

Learn OpenGL. 2014f. Textures. WWW-dokumentti. Saatavissa: https://learnopengl.com/Getting-started/Textures [viitattu 25.11.2019].

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

Mingw. 2019. Minimalist GNU for Windows. WWW-dokumentti. Saatavissa: http://www.mingw.org/ [viitattu 18.10.2019].

Nystrom, R. 2014. Game Programming Patterns. 1. painos. Genever Benning.

OpenAL. 2005. OpenAL 1.1 Specification and Reference. OpenAL-dokumentaatio. Saatavissa: https://openal.org/documentation/openal–1.1–specification.pdf [viitattu 22.10.2019].

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 WWW-dokumentti. Saatavissa: https://www.opengl.org/about/ [viitattu 28.10.2019].

OpenGL Wiki. 2019. OpenGL Loading Library WWW-dokumentti. Saatavissa: https://www.khronos.org/opengl/wiki/OpenGL_Loading_Library [viitattu 24.10.2019].

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

The OpenAL Utility Toolkit. 2006. OpenAL Utility Toolkit Reference Manual. WWW-dokumentti. Saatavissa: http://distro.ibiblio.org/rootlinux/rootlinux-ports/more/freealut/freealut-1.1.0/doc/alut.html [viitattu 25.11.2019].

Visual Studio. 2019. WWW-dokumentti. Saatavissa: https://visualstudio.microsoft.com/vs/ [viitattu 24.11.2019].

Visual Studio Code. 2019. Getting Started. WWW-dokumentti. Saatavissa: https://code.visualstudio.com/docs [viitattu 20.11.2019].

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

XAMK. 2019. Insinööri (AMK), Peliohjelmointi. WWW-dokumentti. Saatavissa: https://www.xamk.fi/koulutukset/insinoori-amk-peliohjelmointi/ [viitattu 26.11.2019].

Zarrad, A. 2018. Game Engine Solutions. E-kirja. London: IntechOpen. Saatavissa: https://www.intechopen.com/books/simulation-and-gaming/game-engine-solutions [viitattu 04.12.2019].

KUVALUETTELO

Kuva 1. Singleton-malli 12

Kuva 2. Pelimoottorin ajonaikaiset komponentit (Gregory 2009, 29) 13

Kuva 3. Luokkakaavio pelin sisältämistä luokista 14

Kuva 4. Tilakaavio pelin eri tiloista 15

Kuva 5. Luonnos päävalikosta 16

Kuva 6. Luonnos valikosta 17

Kuva 7. GLFW-Ikkunan luonti 20

Kuva 8. OpenGL-rajapinnan piirtoprosessi (Learn OpenGL 2014) 21

Kuva 9. GLSL-ohjelmointikielen perus-, vektori- sekä matriisitietotyypit (Shreiner et al. 2013, 40) 22

Kuva 10. GLSL-ohjelmointikielen tyyppimääritteet (Shreiner et al. 2013, 46). 22

Kuva 11. Yksinkertainen verteksivarjostin 23

Kuva 12. Yksinkertainen pikselivarjostin 23

Kuva 13. Varjostimen luonti 24

Kuva 14. Varjostinohjelman luonti 25

Kuva 15. Kokonaisluvun asettaminen Shader-luokassa 26

Kuva 16. Kolmion verteksit 26

Kuva 17. Kolmio normalisoiduilla laitekoordinaateilla (Learn OpenGL 2014) . 26

Kuva 18. Vertex buffer -objektin luominen ja tiedon lähettäminen sekä kolmion piirtäminen 28

Kuva 19. Element buffer -objektin käyttö 29

Kuva 20. Tekstuuri kuutiossa (Wolff 2013, 119) 30

Kuva 21. Kuvatiedon lataaminen 31

Kuva 22. Tekstuurin luominen 32

Kuva 23. Neliön määrittely tekstuurikoordinaattien kanssa 32

Kuva 24. Neliön tekstuurikoordinaatit 32

Kuva 25. Teksturoitu kuutio 33

Kuva 26. Verteksivarjostin tekstuureilla 33

Kuva 27. Pikselivarjostin tekstuureilla 34

Kuva 28. Kameran konstruktorifunktio 36

Kuva 29. Roll-, pitch-, ja yaw -rotaatiot 37

Kuva 30. MoveCamera-funktio 38

Kuva 31. UpdateVectors-funktio 39

Kuva 32. Model-matriisin käyttö 40

Kuva 33. Verteksivarjostin Model View Projection -matriisien kanssa. 40

Kuva 34. Matriisien vieminen varjostimelle 41

Kuva 35. Mallin vienti Blenderistä obj-tiedostoon (OpenGL-tutorial 2017) 41

Kuva 36. Kolmiulotteinen malli omalla obj-tiedoston lataajalla 42

Kuva 37. ProcessNode-funktio 43

Kuva 38. Malli assimp-kirjastolla ladattuna 44

Kuva 39. Drawable-luokan määrittely 45

Kuva 40. Drawable-lista pääohjelmassa 45

Kuva 41. Syötteen tarkistus 46

Kuva 42. Character-rakenne 47

Kuva 43. Värien sekoitus päälle 48

Kuva 44. Font-verteksivarjostin 48

Kuva 45. Font-pikselivarjostin....48

Kuva 46. RenderText-funktio....49

Kuva 47. Teksti peliruudulla 49

Kuva 48. Model –luokan Move –funktio....50

Kuva 49. Pelaajan Move-funktion kutsuminen InputHandler-luokassa....50

Kuva 50. RenderEventText-funktio....51

Kuva 51. ShowingText-funktio....52

Kuva 52. CheckForErrors-funktio 53

Kuva 53. StartAudioManager-funktio....54

Kuva 54. Lopullinen pelidemo 55