Ilkka Hyyryläinen

2D-PELIMOOTTORI TAUSTALATAUSOMINAISUUKSILLA

Opinnäytetyö
Kajaanin ammattikorkeakoulu
Luonnontieteiden ala
Tietojenkäsittelyn koulutusohjelma
Syksy 2014

KAJAANIN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

OPINNÄYTETYÖ
TIIVISTELMÄ

Koulutusala Luonnontieteiden ala	Koulutusohjelma Tietojenkäsittely
Tekijä(t) Ilkka Hyyryläinen	
Työn nimi 2D-pelimoottori taustalatausominaisuuksilla	
Vaihtoehtoiset ammattiopinnot	Toimeksiantaja
Aika Syksy 2014	Sivumäärä ja liitteet 31

Kaupallisten ja ilmaisten pelimoottoreiden määrä ja laatu on kasvanut pelialan mukana, mikä on tehnyt oman pelimoottorin tekemisestä lähes tarpeetonta. Omalla pelimoottorilla voi kuitenkin saada aikaiseksi tukevamman pohjan peliään varten, kun pelimoottori suunnitellaan juuri kyseistä peliä varten.

Opinnäytetyössä käsitellään pelimoottorin toteutusta C++-ohjelmointikielellä, SDL 2.0 -ohjelmistokirjastolla ja OpenGL-rajapinnalla. Nämä työkalut ovat melko yleisiä pelialalla, eritoten pelialan pienyrittäjien tuottamissa peleissä.

Opinnäytetyön teoriaosuus käsittelee pelimoottoreiden suunnittelua ja toteutusta hyödyntäen OpenGL-rajapintaa. Käytännön osuudessa toteutettiin kaksiulotteista grafiikkaa piirtävä pelimoottori, jossa hyödynnetään taustalatausominaisuutta pelimaailmojen käsittelyssä. Lisäksi pelimoottorissa toteutettiin visuaalisia efektejä käyttäen suuntavektorikarttoja.

Analyysiosuus käsittelee toteutetun pelimoottorin käytettävyyttä ja puutteita. Toteutetulla pelimoottorilla voi toteuttaa korttipelejä ja pulmapelejä. Vaativimpien pelien toteutus vaatii pelimoottorin jatkokehitystä, sillä törmäyksentunnistus puuttuu kokonaan.

Kieli	Suomi
Asiasanat	Pelimoottori, OpenGL, taustalataus, SDL 2.0
Säilytyspaikka	☒ Verkkokirjasto Theseus ☐ Kajaanin ammattikorkeakoulun kirjasto

KAJAANIN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

THESIS
ABSTRACT

School Natural Sciences	Degree Programme Business Information Technology
Author(s) Ilkka Hyyryläinen	
Title 2D-game engine with streaming functionality	
Optional Professional Studies	Commissioned by
Date Fall 2014	Total Number of Pages and Appendices 31

The quality and quantity of commercial and free game engines has increased as the game industry has grown. With this, the creation of your own game engine has become practically redundant. But with your own game engine it is still possible to have a much better starting point for your game, as the game engine is designed precisely for the game in question.

This thesis looks into the implementation of a game engine using C++-programming language, SDL 2.0 library, and OpenGL pipeline. These tools are quite common in the game industry, especially in indie game development.

The thesis theory section looks into game engine design and implementation using OpenGL pipeline. In the practical section a game engine that renders 2D graphics is implemented. This engine uses streaming functionality for handling game worlds. The engine also includes visual effects that are implemented using normal maps.

The analysis section talks about the possible utilities and lacking functions of the developed game engine. The engine can be used to create virtual card games and puzzle games. Any more demanding games require continued development of the engine, since it lacks proper collision detection.

Language of Thesis	Finnish
Keywords	Game engine, streaming, OpenGL, SDL 2.0
Deposited at	☒ Electronic library Theseus ☐ Library of Kajaani University of Applied Sciences

SISÄLLYS

SYMBOLILUETTELO

FBO	Frame Buffer Object eli näytönohjaimen muistiin tallennettu olio, joka sisältää dataa kuvan piirrosta.
Fragmentti	Fragmentilla tarkoitetaan vertekseillä rakennetun kolmion pinnalla sijaitsevaa pistettä, jolla on väriarvo. Yleensä yksi fragmentti vastaa yhtä pikseliä ruudulla.
Kuutiokartta	Kuutiokartta on kaksiulotteinen tekstuuri, joka sisältää kuution kuudelle eri seinämälle pintakuvion, jotka muodostavat 360-asteisen maisemakuvan kuution sisältä päin katsoessa.
Modi	Peli tai pelin lisäosa, joka hyödyntää julkaistun pelin valmista pelimoottoria ja sisältöä. Modit yleisesti vaativat alkuperäisen pelin asentamista toimiakseen.
OpenGL	OpenGL on ohjelmointirajapinta, jonka avulla ohjelmoija voi hallita grafiikan laskentalaitteistoa, esim. näytönohjainta.
Peligenre	Ryhmä, johon peli sijoitetaan sen mekanismien ja sisällön mukaan. Esimerkiksi ajopelit, toimintapelit, urheilupelit, simulaattorit.
Pelikonsoli	Laite, jonka tarkoitus on pääosin ajaa peliohjelmistoa. Useimmiten liitetään televisioon osaksi kodin viihdekeskusta. Esimerkkeinä Playstation, Xbox.
Ruutukartta	Ruutukartta on ruudukko, jonka nelimuotoiset palaset muodostavat yhdessä pelimaailman. Ruutukartan ideana on ruutujen grafiikan uudelleenkäytettävyys ja toistuvuus.
Tekstuuri	Tekstuuri on pintakuvio. Usein kaksiulotteinen värikartta, joka voidaan generoida ohjelmassa tai ladata kuvatiedostosta.
Tesselaatio	Grafiikan piirrossa algoritmi, jolla saadaan yksinkertaisemmasta kolmiulotteisesta mallista tehtyä dynaamisesti tarkempi ja yksityiskohtaisempi riippuen tilanteesta.
Suuntavektorikartta	Tekstuuri, jonka väriarvot kuvaavat suuntavektoreita. Toisin sanottuna se on kaksiulotteinen taulukko suuntavektoreista.
Säie	Ohjelman operaatioita suorittava komentojono, joka suorittaa koodin operaatioita yksi kerrallaan. Useampi säie ohjelmassa mahdollistaa operaatioiden suorittamista samanaikaisesti.
Säieturvallisuus	Säieturvallisuudella tarkoitetaan ohjelman koodin osaa, joka manipuloi säikeiden jakamaa dataa tavalla, joka ei aiheuta ongelmia ohjelman toiminnassa.

Verteksi	Piste, jolla on sijainti kolmiulotteisessa avaruudessa. Useampi verteksi liitetään yhteen muodostaen viivoja, jotka muodostavat keskenään kolmioita ja nämä kolmiot muodostavat kolmiulotteisia muotoja grafiikkaa piirtäessä.

1 JOHDANTO

Nykyään löytyy jo suuri määrä ilmaisia ja kaupallisia pelimoottoreita monenlaisia erilaisia pelejä varten. Tämän takia on hyvin yleistä hankkia lisenssi jo valmiiseen pelimoottoriin uuden ohjelmoimisen sijasta. Kuitenkin oman pelimoottorin tekeminen mahdollistaa sen räätälöimisen juuri sitä tarvitsevaa peliä varten ja mahdollistaa paremman ymmärryksen koko järjestelmän toiminnallisuudesta.

Tässä projektissa tähdättiin pelimoottorin uudelleenkäytettävyyteen monenlaisissa erilaisissa peleissä rajoittaen grafiikan toteutuksen ainoastaan kaksiulotteisiin kuviin. Suuri tekijä tämän päätöksen takana oli uteliaisuus ja mielenkiinto omanlaisen pelimoottorin toteuttamisesta hyödyntäen OpenGL:n varjostinkielen mahdollistamia visuaalisia efektejä ja pelimaailman latautumisesta taustalla, joka mahdollistaa keskeytymättömän pelikokemuksen. Lisäksi tätä pelimoottoria aiotaan käyttää tulevissa kaupallisissa peleissä.

Pelimoottori on rajoitettu kaksiulotteiseen grafiikkaan, sillä näiden tulevien pelien olisi myös tarkoitus toimia kaksiulotteisesti. Kaksiulotteisuus on myös henkilökohtainen tyylivalinta, ja sitä on paljon yksinkertaisempi toteuttaa.

Pelimoottorissa hyödynnetään paljon valmiita avoimia kirjastoja välttäen kaikkia Windows käyttöjärjestelmän funktioita ja ominaisuuksia siltä varalta, että pelimoottorilla toteutettuja pelejä haluttaisiin kääntää muillekin käyttöjärjestelmille. Pääsääntöisesti pelimoottori on kuitenkin tehty kotikäyttöisille tietokoneille, mutta konsoleille kääntäminen ei myöskään ole ylitsepääsemätön este.

Jo ennen opinnäytetyön aloittamista tiedettiin, ettei tämä pelimoottori tule olemaan täysin valmis, vaan sitä on jatkettava opintojen jälkeen. Kuitenkin tarkoituksena oli saada pelimoottorin pääominaisuudet toimimaan opinnäytetyötä varten varatun kolmen kuukauden aikana.

2 PELIMOOTTORIT

Pelimoottorin ja pelin raja on kuin veteen piirretty viiva. Joissain tapauksissa pelimoottori on helpompi erottaa pelistä, mutta toisissa tapauksissa se on lähes mahdotonta. Peli saattaa piirtojärjestelmässään tietää, miten tietty hahmo piirretään, ja toisen pelin piirtojärjestelmä tietää vain, miten hahmon kaltaisia olioita piirretään, mutta itse hahmon data tulee muualta. Voidaan väittää, että datapohjainen arkkitehtuuri erottaa pelimoottorin muusta pelin ohjelmistosta. Pelimoottori on ohjelmisto, jonka koodia voidaan uudelleen käyttää uusien pelien luomiseen. Pelimoottorin käytettävyys riippuu myös itse pelimoottorista. Jotkin pelimoottorit soveltuvat paremmin tietyntyyppisiin peleihin mutta heikommin toisiin. Pelimoottori, joka on tehty sopimaan mahdollisimman moneen eri peligenreen, ei toimi yhtä hyvin kuin pelimoottori, joka on tehty nimenomaan kyseistä genreä varten. (Gregory 2009, 11–13.)

2.1 Pelimoottorien historiaa

90-luvulla pelimoottori termi ilmaantui toimintapelien kuten suuresti menestyneen Doomin myötä. Id Softwaren kehittämä Doom oli koodiarkkitehtuuriltaan rakennettu siten, että pelin ydinohjelmisto eriytettiin pelin sisällöstä ja säännöistä. Tähän ydinohjelmistoon sisältyi kolmiulotteisen grafiikan piirto, äänten toisto ja törmäysfysiikan laskenta. Tästä erotuksesta todettiin olevan suurta hyötyä, kun pelien kehittäjät ostivat muiden pelien lisenssejä ja hyödynsivät niiden valmista koodia uusia pelejään varten, lisäämällä vain omat peliympäristöt, hahmot, äänet ja muun oleellisen sisällön. 90-luvun loppupuolella pelejä, kuten Quake III ja Unreal, valmistettiin pitäen tämä uudelleenkäytettävyys mielessä, antaen pelaajille käyttöön tarvittavat työkalut omien pelien tai modien valmistamista varten. Pelimoottorien lisenssien myynnistä tuli toinen tuote pelistudioille. (Gregory 2009, 11.)

Pelimoottorin vaatimukset vaihtelevat eri peligenrejen välillä. Lähes kaikissa pelimoottoreissa on kuitenkin tiettyjä yhtenäisiä vaatimuksia, kuten kuvan piirto, äänitiedostojen toisto ja käyttäjän antaman ohjainsyötteen tulkinta. Huolimatta Epic Gamesin kehittämän Unreal Engine pelimoottorin nojautumisesta ammuntapeleihin, sitä on onnistuneesti käytetty myös muunlaisiin peleihin hyödyntäen pelimoottorin grafiikan piirtojärjestelmää ja muita perusominaisuuksia. Otetaan esimerkiksi kolme eri peligenreä: Ensimmäisen persoonan ammuntapelit, kolmannen persoonan pelit ja kilpa-ajopelit. Ensimmäisen persoonan näkökulmasta esitetyt

ammuntapelit ovat usein pelimoottoreiltaan kaikista vaativimmat, sillä näissä peleissä pyritään mahdollisimman paljon pelaajan sisäistämiseen itse pelihahmon saappaisiin. Tämän takia pelimoottorin on pystyttävä tehokkaasti piirtämään mahdollisimman todentuntuisia, kolmiulotteisia ja laajoja ympäristöjä. Tämänkaltaiset pelit vaativat myös uskottavaa tekoälyä tietokoneen ohjaamilta hahmoilta sekä uskottavia animaatioita. Tasoloikkapelit tai pelit, joita esitetään kolmannen persoonan näkökulmasta, eivät välttämättä edes vaadi kolmiulotteisia ympäristöjä. Tällaisissa peleissä voidaan ottaa enemmän vapauksia pelin taiteellisessa tyylissä ja tehdä pelistä enemmän piirretyn animaation näköinen. Jos kyseessä on kuitenkin ammuntapeli kolmannesta näkökulmasta, ovat vaatimukset lähes samanlaiset kuin ensimmäisen persoonan ammuntapeleissä, mutta suurempi huomio kiinnittyy pelattavan hahmon animaatioon. Kilpa-ajopelit vaativat eniten graafista suorituskykyä ajoneuvoihin ja joskus myös niitä ajaviin hahmoihin. Koska näissä peleissä pysytään usein hyvin lineaarisilla teillä, joista ei päästä ulos tutkimaan ympäristöjä, voidaan kaukaisimmat objektit piirtää yksinkertaisina kaksiulotteisina kulisseina. Tällöin pelimoottori voi keskittää suurimman osan piirtoprosessistaan itse tien ja lähimmän ympäristön grafiikkaan. (Gregory 2009, 13–20.)

2.2 Pelimoottorin arkkitehtuuri

Pelimoottorit rakennetaan tasoista, kuten kaikki muutkin ohjelmistojärjestelmät. Ylemmät tasot ovat riippuvaisia alemmista tasoista eikä toisinpäin. Kiertävällä riippuvuudella tarkoitetaan sitä, kun alempi taso on riippuvainen ylemmästä tasosta. Tällaisia riippuvuuskierteitä on vältettävä kaikissa ohjelmistojärjestelmissä, koska se aiheuttaa epävakaata toiminnallisuutta ja haittaa koodin uudelleenkäytettävyyttä. Tämä on eritoten tärkeätä huomioida pelimoottoreiden kaltaisissa suurissa järjestelmissä. (Gregory 2009, 28.)

2.2.1 Pelimoottorin rakenteen tasot

Alimpana tasona pelimoottorijärjestelmässä on itse laitteisto. Tämä on se tietokone tai pelikonsoli, jolla peliä pelataan. Tämän jälkeen tulevat laitteiston ajurit, jotka suojaavat ylempiä tasoja laitteiston kommunikaation yksityiskohdilta.

Kolmanneksi alimpana tasona on käyttöjärjestelmä. Tietokoneissa käyttöjärjestelmä on aina päällä, mikä on huomioitava pelimoottoria tehdessä, sillä esimerkiksi Microsoftin Windows

käyttöjärjestelmä jakaa laitteiston resursseja muidenkin ohjelmien ja prosessien kesken. Pelimoottorilla ei siis voida olettaa olevan täyttä hallintaa laitteiston resursseista tietokoneella, vaan sen on jaettava kyseiset resurssit muiden ohjelmien kanssa. Konsolipeleissä tällaista jakamista ei tarvinnut pelätä, kunnes PS3 ja Xbox 360 toivat mukanaan käyttöjärjestelmiinsä pelinaikaisia ominaisuuksia, joilla pelaajalle näytetään netistä saatuja viestejä tai annetaan pelaajan keskeyttää peli päästäkseen käsiksi konsolin päävalikkoon.

Käyttöjärjestelmän yläpuolella ovat kolmannen osapuolen ohjelmistokirjastot. Näitä ovat kolmiulotteisen grafiikan piirtokirjastot, kuten OpenGL ja DirectX, tai valmiit fysiikka- ja animaatiomoottorit. Kaikki pelimoottorin peruskivinä käytetyt ohjelmistokirjastot löytyvät tältä tasolta.

Jatkaen tasoa ylemmäksi tulee vastaan alustariippuvuustaso. Koska joidenkin pelimoottoreiden on toimittava useammalla eri laitteistopohjalla, tämä ohjelmistotaso tarvitaan suojaamaan ylempien tasojen toimintaa alemmilta tasoilta, jotka ovat enemmän alustariippuvaisia. Tällä tasolla joko sidotaan tai korvataan yleisimmin käytetyt C-kirjaston standardit funktiot, käyttöjärjestelmän kutsut ja kolmannen osapuolen ohjelmistokirjastot. Tällöin varmistetaan pelimoottorin toimiminen samalla tavalla kaikilla tarkoitetuilla alustoilla.

Seuraavaa tasoa voidaan kutsua pelimoottorin ydintasoksi. Tällä tasolla ovat esimerkiksi pelimoottorin asetukset, profilointi, statistiikka, matematiikkakirjasto ja moduulien hallinta. Eli itse pelimoottorin ydinominaisuudet, jotka eivät liity suoraan itse peliin.

Resurssien hallinta on myös omana tasonaan pelimoottorin arkkitehtuurissa. Jotkin pelimoottorit sisältävät oman keskitetyn resurssien hallinnan, joka ylläpitää kaikkia sisältökutsuja. Jotkin pelimoottorit sen sijaan jättävät tämän osan ohjelmoijan vastuuksi.

Resurssien hallinnan jälkeen tulevat itse pelimoottorin erilaiset alakomponentit. Näistä komponenteista piirtomoottori on yksi suurimmista ja monimutkaisimmista pelimoottorin osista. Tämän tason toteutuksesta ei ole mitään yhtä ja ainutta hyväksyttyä tapaa. Sen voi toteuttaa monella eri tavalla. Yksi usein tehokas tapa toteuttaa tämä taso on jakaa se omiin pienempiin tasoihin. Muita alakomponentteja tällä tasolla ovat esimerkiksi äänten hallinta, fysiikan laskenta ja törmäysten tunnistus, käyttäjän syöte ja verkkopeliajuri.

Viimeinen taso sisältää itse pelin luokat ja objektit. Pelin ja pelimoottorin välinen raja voitaisiin vetää tämän ja alakomponenttitason välille. Tämän tason sisältö on hyvin paljon riippuvainen itse pelistä. (Gregory 2009, 30–49.)

2.2.2 Työkalut ja sisällönhallinta

Pelimoottorit vaativat toimiakseen paljon dataa erilaisten tiedostojen kuten asetustallenteiden ja koodien muodossa. Pelien ollessa luonnoltaan multimediaohjelmia tarvitaan myös useita erilaisia graafisia tiedostoja (kuvat, 3D-mallit) ja ääniä. Näitä tiedostoja ei yleensä luoda itse pelimoottorissa, vaan kolmannen osapuolen ohjelmissa, jotka erikoistuvat nimenomaan kyseisten tiedostojen luomiseen. Näitä ohjelmia ovat esimerkiksi Adobe Photoshop kuvien luontia varten, 3ds Max tai Maya kolmiulotteisia malleja ja animaatioita varten, ja ääniä voi luoda vaikka SoundForge-ohjelmalla. Tietenkin joitain pelimoottorin vaatimia tiedostoja ei voi luoda kolmannen osapuolen ohjelmissa, vaan niitä varten on luotava omat ohjelmat, kuten esimerkiksi pelimaailman editori. Näiden kolmannen osapuolen ohjelmien tuottamista tiedostoista on osattava erottaa ne tiedostotyypit, jotka ovat oikeasti hyödyllisiä pelimoottorissa. Esimerkiksi Maya tallentaa omissa tiedostoissaan hyvin monimutkaista dataa, josta pelimoottori tarvitsisi vain pienen osan. Tämänkaltaisten tiedostojen lukeminen pelin aikana on myös liian hidasta. Tämän takia kolmannen osapuolen ohjelmien data muutetaan sopivampaan tiedostomuotoon ja joskus vielä muokataan muutoksen jälkeenkin pelimoottorin tarpeiden täyttämiseksi. (Gregory 2009, 49–50.)

2.2.3 Pelin olioiden hallinta

Jokainen pelin käyttämä olio vaatii jonkinlaisen sille ainutlaatuisen tunnuksen, jotta se voidaan erottaa muista pelin olioista. Tämän tunnuksen avulla olio voidaan löytää pelin ajon aikana ja sitä voidaan käyttää kohteena olioiden välisessä kommunikaatiossa. Nämä tunnukset auttavat myös mahdollisten pelimoottoreiden maailmojen editoreissa löytämään ja tunnistamaan tiettyjä olioita. Pelin ajon aikana tarvitaan useita eri tapoja löytää olioita. Pelissä voidaan haluta käsitellä tiettyä oliota tunnuksen mukaan tai jonkin pelitilanteen ehdon kuten lasketun etäisyyden pelaajan pelihahmon ja itse olion mukaan. Olion löydyttyä siihen on pystyttävä viittaamaan. C++- ja C-ohjelmointikielissä kaikista suoraviivaisin tapa on tehdä viittaukset osoittimilla. Pelimoottoreissa on myös mahdollista käyttää muita ratkaisuja, kuten kahvoja tai älykkäitä osoittimia. (Gregory 2009, 750–751.)

2.2.4 Osoittimien puutteet

Osoittimet ovat tehokas ja yksinkertainen tapa tehdä viittauksia, mutta niiden mukana tulee muutama ongelma. Näitä ongelmia ovat orvot oliot, vanhentuneet osoittimet ja väärät osoittimet. Ideaalisesti jokaisella oliolla on omistaja, jonka vastuuna on olion ylläpito koko sen elinajan aikana. Tämä omistaja luo kyseisen olion ja poistaa sen, kun sitä ei enää tarvita. Osoittimissa ei ole itsessään toiminnallisuutta ylläpitämään näitä sääntöjä. (Gregory 2009, 750–751.)

2.2.5 Älykäs osoitin

Älykäs osoitin on pieni olio, jonka tarkoitus on toimia hyvin paljon samankaltaisesti kuin tavallisen osoittimenkin, mutta välttäen C- ja C++-ohjelmointikielien osoittimien yleisimpiä ongelmia. Älykäs osoitin sisältää paikallisen osoittimen datajäsenenään ja sisältää operaattoreiden ylikirjoitettuja määritelmiä, jotka saavat sen käyttäytymään osoittimena. Osoittimien *- ja ->-operaattorit voidaan siis yliajaa omilla funktioilla, varmistamaan, että oikea muistiosoite palautetaan. Älykäs osoitin voi sisältää myös metadataa ja toteuttaa muutamia lisätoimintoja, koska se on olio. Esimerkiksi älykäs osoitin voi tunnistaa, milloin sen osoittaman muistiosoitteen data on poistettu ja asettaa itsensä tyhjäksi (null). Älykkäät osoittimet voivat myös pitää yllä kirjaa muistiosoitteeseen tehdyistä viittauksista. Tätä kutsutaan viittauslaskimeksi. Kun tiettyyn muistiosoitteeseen osoittavien älykkäiden osoittimien määrä laskee nollaan, suoritetaan muistin vapautus kyseiseltä muistiosoitteelta, estäen orpojen olioiden muodostumista. Täten ohjelmoijan ei tarvitse huolehtia kaikkien osoittimien datan hallitsemisesta. Älykkäät osoittimet on helppo toteuttaa, mutta vaikea toteuttaa oikein, sillä niiden toteutuksessa on otettava huomioon monia erilaisia tapauksia. (Gregory 2009, 751.)

2.2.6 Kahva

Kahva on monella tapaa samanlainen kuin älykäs osoitin, mutta se on yksinkertaisempi toteuttaa ja sisältää vähemmän riskejä. Kahva on sarjanumero, joka osoittaa kahvataulun lohkoon. Kyseinen kahvataulu sisältää varsinaiset osoittimet olioihin, joita kahvoilla haetaan. Kahvat ovat paljon turvallisempi tapa suorittaa viittauksia olioihin kuin osoittimet. Jos kah-

vataulun osoittimen data poistetaan ja osoitin asetetaan tyhjäksi, ovat tällöin myös kaikki siihen viittaavat kahvat jo valmiiksi tyhjiä viittauksia, välttäen vanhentuneiden osoittimien syntymistä. Ongelmana ilmenee kahvaratkaisussa datan poistaminen ja myöhemmin sen korvaaminen toisella oliolla. Jos jossain päin ohjelman ajon aikana jokin toinen olio käyttää vanhaa kahvaa, ne tulevatkin viittaamaan tähän uuteen olioon luullen sitä vanhaksi poistetuksi olioksi. Tämä voidaan ratkaista lisäämällä kahvaan myös kyseisen olion tunnusluku. Täten oliot, jotka viittaavat kyseiseen kahvaan, varmistavat myös, että he hakevat tiettyä oliota. (Gregory 2009, 752–753.)

2.3 Taustalataus

Taustalatauksella tarkoitetaan prosessia, jossa dataa ladataan taustalla samalla kun itse pääohjelma on käynnissä. Englanniksi tämä termi tunnetaan nimellä streaming. Näin monissa peleissä saadaan aikaiseksi keskeytymätön pelikokemus lataamalla taustalla dataa tulevista tasoista DVD-, BluRay- tai kovalevyiltä, samaan aikaan kun peliä pelataan. Yleisimmin ääni- ja kuvatiedostot ladataan taustalla, mutta mikään ei estä lataamasta muunlaisiakin tiedostoja, kuten geometriadataa tai animaatioita. Jotta taustalataamista voitaisiin tukea pelimoottorissa, on hyödynnettävä epäsynkronista tiedoston lataamis- ja tallennuskirjastoa, joka antaa ohjelman jatkaa toimintaansa tiedostojen lataamisen aikana. Jotkin käyttöjärjestelmät tuovat mukanaan tällaisia ohjelmistokirjastoja asynkronista lataamista varten. Esimerkiksi Windows Common Language Runtime (CLR) tuo mukanaan funktioita, kuten System.IO.BeginRead() ja System.IO.BeginWrite(). Playstation 3 -pelikonsolissa on myös tähän sopiva ohjelmointirajapinta nimeltä ”fios”. Mikäli käytetyllä järjestelmällä ei ole omaa ohjelmistokirjastoa datan asynkronista lataamista ja tallentamista varten, on sellainen luotava itse. (Gregory 2009, 269.)

2.4 Taustalatauksen historiaa

Yleisimmin taustalatausta käytettiin esimerkiksi CD-levyjä lukevissa pelikonsoleissa musiikin lataamiseen. Tällä tavalla voitiin säästää muistia pelikonsoleiden rajallisella laitteistolla, kun koko kappaletta ei ladattu yhdellä kertaa, vaan ainoastaan se pieni osa, jota tarvittiin kyseisellä hetkellä musiikin toistamiseen. CD-levyasemien hitauden vuoksi pelit eivät voineet ajaa kuin yhtä taustalatausprosessia samanaikaisesti. Nopeampien DVD-levyasemien tultua peli-

konsoleihin voitiin konsolipeleissä ajaa samaan aikaan useampaa stream-prosessia. Taustalataus oli konsolipeleissä eritoten tarpeellista, koska konsolien laitevalmistajat asettivat tiukkoja rajoituksia lataustaukojen pituuksille. Taustalatauksella pysyttiin näiden rajoitusten sisällä. (Carter 2004, 4.)

2.5 Taustalatauksen toteutus

Taustalatauksen toteuttamiseksi sisällön lataus suoritetaan asynkronisesti muun pelin toimintojen kanssa, ettei peli keskeydy, kun dataa ladataan muistiin. Datan lataaminen suoritetaan eri säikeellä erillään muusta pelin toiminnallisuudesta. (Adam Lake 2010, 528.)

Mikäli ladattavat tiedostot ovat vielä pakattuna, on hyvä idea luoda datan purkamista varten myös oma säie. (Think Services, 2006.)

Tiedostoja ladatessa on myös priorisoitava datan tarpeellisuutta. Joitakin tiedostoja tarvitaan enemmän kuin toisia pelin toiminnallisuuden varmistamiseksi. Ensimmäisenä on ladattava välttämättömimmät tiedostot, jotka ovat pakollisia pelin toiminnan kannalta. Mikäli näitä tiedostoja ei ole ladattuna, kun niitä tulisi käyttää, peli keskeytetään lataustauon ajaksi. Seuraavaksi ladataan tiedostot, jotka ovat tärkeitä mutta eivät täysin välttämättömiä pelin etenemiseen. Esimerkiksi verkkopelin pelaajien luomat pelihahmot voivat olla lataamatta samalla kun pelaajien annetaan keskustella keskenään pelin chat-ikkunassa. Kolmanneksi tärkeimmät tiedostot sisältävät koristeellista dataa, kuten taustan grafiikkaa tai hienoja visuaalisia efektejä. Tämä data vaikuttaa vain pelin ulkonäköön, ei sen toimintaan. Viimeiseksi ladataan tiedostot, joita ei vielä tarvita pelissä, mutta ennustetaan tarvittavan pian. Nämä latausprioriteetit ovat itsestään selviä, mutta haaste onkin luoda looginen järjestelmä, joka osaa jakaa kaiken datan oikeisiin prioriteetteihin. Tämän prioriteettijaon ei tarvitse olla täysin automaattinen, vaikka sellaisesta olisikin paljon hyötyä. Prioriteettien jakamisen voi jättää suunnittelijan vastuuksi. (Ansari 2011, 23–25.)

Pelimoottorin maailman lataamisen järjestelmällä on kaksi päävastuuta. Toinen on tiedostojen lataamisjärjestelmä, joka lataa maailman palaset ja niiden kaikki materiaalit levyltä välimuistiin, ja toinen on muistin varaaminen ja vapauttaminen näitä resursseja varten. Pelimoottorin on myös pystyttävä luomaan ja poistamaan pelin objekteja samalla, kun niitä tarvitaan tai ne poistuvat käytöstä. Kaikkein suoraviivaisin tapa ladata pelimaailmoja muistiin on aikaisimmissa peleissä käytetty tapa, jossa muistiin ladataan vain yksi taso kerrallaan. Muistin

hallinta tällaisessa ratkaisussa on melko suoraviivaista. Pelin käynnistyessä kaikki yleisimmin käytetyt tiedostot ladataan muistiin valmiiksi. Tämän jälkeen loput muistista on käytössä pelimaailmoja varten. Muistiin ladataan vain yksi taso kerrallaan, ja pelaajan päästäessä kentän läpi se poistetaan muistista ja uusi taso ladataan tilalle. Tässä ratkaisussa tietenkin pelaaja joutuu odottamaan, kunnes seuraava taso on ladattu muistiin.

Tämän lataustauon poistamiseksi yksinkertainen tapa on jakaa muisti kahteen osaan. Ensimmäisellä osalla on se taso, jota pelaaja pelaa. Samalla kun pelaaja etenee tasoa läpi, toiseen muistiosaan ladataan seuraavaa tasoa. Haittapuolena tämä rajoittaa jokaisen tason kokoa puoleen. Ilmalukkoratkaisussa muistissa pidetään pienempi osa valmiina yksinkertaisempaa ilmalukkotasoa varten. Tällä ilmalukkotasolla on jonkinlainen portti tai muu ominaisuus, joka estää pelaajaa näkemästä ilmalukon ulkopuolelle. Pelaajaa estetään pääsemästä takaisin edelliseen tasoon, joka poistetaan muistista. Samalla kun pelaaja pelaa ilmalukkotasossa jonkinlaista tehtävää, joka voi vaihdella yksinkertaisesta läpijuoksusta taisteluun vihollislaumaa vastaan, peli lataa seuraavan täyden tason muistiin. Halo-pelissä käytettiin tämänkaltaista tapaa. Suuremmat alueet olivat yleensä yhdistettynä toisiinsa pienemmillä suljetuilla tiloilla, jotka estivät pelaajaa palaamasta takaisin. (Gregory 2009, 741–743.)

Joissakin peleissä kuitenkin halutaan saada pelaaja tuntemaan olevansa suuressa maailmassa, jota ei ole erotettu pienillä ilmalukoilla. Näissä peleissä olisi parasta, että maailma avautuisi pelaajan edessä mahdollisimman luonnollisesti ja uskottavasti. Nykyiset pelimoottorit tukevat tällaisia pelimaailmoja käyttäen taustalatausta. Taustalatauksen voi toteuttaa usealla eri tavalla, mutta päätavoitteena on kuitenkin ladata dataa samalla kun pelaaja pelaa peliä ja hallita muistia tätä dataa varten. Nykyisillä tietokoneilla ja pelikonsoleilla on tarpeeksi välimuistia ylläpitämään useampia kentän palasia ladattuna kerralla. Esimerkiksi muistiin voitaisiin ladata ensimmäiset kolme palasta: A, B ja C. Kun pelaaja etenee A-palasta B-palaan ja on ehtinyt B-palalla tarpeeksi kauas A:sta, ettei se enää näy, poistetaan A muistista ja aloitetaan palan D lataaminen muistiin. Tätä operaatiota toistetaan, eikä pelaaja tule koskaan huomaamaan, että pelimaailma onkin pienemmissä palasissa, vaan peli tuntuu yhdeltä suurelta tasolta. Tällaisessa ratkaisussa on kuitenkin hyvin tärkeätä, että jokaisen palasen koko on lähes sama. Niiden on oltava tarpeeksi suuria täyttämään niille varattu tila muistista mutta ei koskaan sen suurempia. Tämän kiertämiseksi voidaan yrittää hienompaa muistin jakamista. Isojen maailman palasten taustalataamisen sijasta kaikki pelin sisältö, kuten tekstuurit ja animaatiot, jaetaan samankokoisiksi pienemmiksi palasiksi. (Gregory 2009, 743–744.)

3 GLSL-VARJOSTIMET

GLSL (OpenGL Shading Language) on OpenGL:n käyttämä ohjelmointikieli, joka on olennainen osa OpenGL:n ohjelmointirajapintaa. Tulevaisuudessa jokainen ohjelma, joka käyttää OpenGL-rajapintaa, hyödyntää yhtä tai useampaa GLSL-ohjelmaa. Näitä pienohjelmia kutsutaan myös varjostinohjelmiksi tai varjostimiksi. Varjostin on pieni ohjelma, jonka suorittamiseen käytetään näytönohjaimen grafiikkasuoritinta. Varjostimen nimellä viitataan näiden ohjelmien yleiseen käyttöön valojen ja varjojen laskemisessa kolmiulotteisissa kuvissa. Varjostimien käyttö ei kuitenkaan rajoitu vain valon laskemiseen, vaan niillä voidaan myös saada aikaan paljon enemmän, kuten animaatiota ja tesselaatiota. Varjostinohjelmat suunnitellaan toimimaan grafiikkasuorittimella usein samanaikaisesti keskenään hyödyntäen suorittimien määrää, tehden varjostinohjelmista erittäin tehokkaita. (Wolff 2011, 6.)

3.1 Varjostimien lyhyt historia

Ennen varjostimia OpenGL-rajapinnassa käytettiin kiinteää funktioputkea, joka sisälsi ennalta määritetyn valo ja varjostusalgoritmin. Ohjelmoijien täytyi käyttää erilaisia keinoja koodissaan, jotta tämä putki saatiin toimimaan heidän haluamallaan tavalla. Vain näin saatiin aikaiseksi realistisia efektejä. GLSL-kieli kehitettiin korvaamaan tämä funktioputki ja antamaan ohjelmoijille vapaus luoda omia algoritmeja toteuttamaan sitä, mitä vanha metodi teki ohjelmoijan puolesta. (Wolff 2011, 6–7.)

3.2 Varjostimien käyttö

Windows-puolella ohjelmoitaessa on huomioitava, että Windows-käyttöjärjestelmän mukana tulevat OpenGL-ohjelmointirajapinnan koodikirjastot ovat vanhentuneita eikä Microsoft aio niitä päivittää. Tämän takia Windowsilla ohjelmoitaessa ei voida suoraan kutsua uusia OpenGL-funktioita, vaan ne on linkitettävä käyttäen muita koodikirjastoja, joista voidaan hakea osoittimet uusiin funktioihin ajon aikana. Yksi tällaisista ohjelmistokirjastoista tunnetaan nimellä GLEW (OpenGL Extension Wrangler). GLEW-ohjelmistokirjaston voi ladata ilmaiseksi osoitteesta http://glew.sourceforge.net. Tämä ohjelmistokirjasto sisältää OpenGL-funktio-osoittimien lisäksi muutamia omia funktioita, jotka ovat melko hyödyllisiä.

Funktio ”visualinfo()” luo tekstitiedoston, johon listataan kaikki käytössä olevat OpenGL, WGL- ja GLU-lisäosat. Funktio ”glewinfo()” listaa käyttäjän ajureiden tukemat funktiot. GLEW-ohjelmistokirjaston avulla voi myös tarkistaa lisäosien käytettävyyttä tarkastelemalla sen määrittämiä globaaleja muuttujia. Lisäksi GLEW:n kanssa voi käyttää GLM (OpenGL Mathematics)-ohjelmistokirjastoa. OpenGL 4.0 -versiossa poistettiin vanhat matriisipinot, kuten GL_MODELVIEW ja GL_PROJECTION. GLM-ohjelmistokirjasto tuo mukanaan GLSL-ohjelmoijille hyödyllisiä matemaattisia funktioita ja matriiseja. (Wolff 2011, 7–10.)

GLSL-ohjelmointikieli on syntaksisesti samankaltainen C-kieleen verrattuna. OpenGL 4.0 -versio sisältää viisi varjostintasoa: verteksi, geometria, tesselaation hallinta, tesselaation arviointi ja fragmentti. Verteksi-varjostin suoritetaan kerran jokaista syötettyä verteksiä kohden, mahdollisesti samanaikaisesti. Verteksin sijainti lasketaan ohjelmassa ja asetetaan gl_Position-muuttujaan ennen ohjelman päättymistä. Verteksiohjelma voi myös lähettää muuta dataa alemmille tasoille käyttäen ulostulomuuttujia. (Wolff 2011, 48.)

Verteksiohjelma korvaa entisen funktioputken tuomat operaatiot, esimerkiksi verteksien transformaation, normalisoinnin, valaistuksen. Nämä toiminnallisuudet toteutetaan itse ohjelmoimalla verteksiohjelmassa. (GameDev.net, LLC.)

Fragmenttiohjelmassa rasteroinnin tuottamat fragmentit muutetaan värillisiksi pikseleiksi ja syvyysarvoiksi. (Movania, 2013, 16.)

Fragmenttiohjelmassa ohjelmoijalla on täysi hallinta jokaisen fragmentin prosessoinnissa. Fragmenttiohjelma korvaa vanhasta funktioputkesta sumun ja tekstuurien laskennan. (GameDev.net, LLC.)

GLSL-ohjelmointikieli sisältää neljä perusdatatyyppiä. Ensimmäisinä ovat int, float ja bool (kokonaisluku, liukuluku ja Boolean arvo). Näille kolmelle datatyypille ovat olemassa myös vektorimuuttujat. Ne sisältävät vähintään kaksi ja enintään neljä muuttujaa. Float-datatyypille on vielä olemassa matriisimuuttujat. Niiden koko on aina symmetrinen, pienimmillään 2x2 ja suurimmillaan 4x4. Neljäs datatyyppi on sampler. Näitä käytetään tekstuurien käsittelyssä. Samplereihin voi tallentaa yksi-, kaksi- ja kolmiulotteisia tekstuureja, kuutiokarttoja ja syvyyskomponenttitekstuureita. Sampleri-datatyypin muuttujien on aina oltava uniformeja. Uniformit muuttujat eivät muutu kuvan piirron aikana. Näitä voidaan käyttää verteksi- ja fragmenttiohjelmissa. Attribuutit ovat muuttujia, joita voidaan käyttää vain verteksiohjelmissa. Ne ovat syötemuuttujia, joiden arvot muuttuvat jokaisella verteksillä. Varying-muuttujat ovat

tarpeen verteksiohjelmien ja fragmenttiohjelmien välisessä datan siirrossa. Verteksiohjelmat voivat muuttaa niiden arvoja, joita fragmenttiohjelmat lukevat mutta eivät voi enää muokata. Kaikki muuttujat, jotka ovat tyyppiä uniform, attribute tai varying, täytyy määrittää globaaleina muuttujina. Niitä ei saa määrittää funktioissa. (GameDev.net, LLC.)

3.3 Varjostimien koodi

GLSL-ohjelmointikielessä on muutamia toimintoja ja rajoituksia. Esimerkiksi se on täysin tyyppiturvallinen, eli tietyntyyppistä dataa ei voi asettaa vääränlaiseen muuttujaan. Esimerkiksi float-muuttujaan ei saa asettaa kokonaislukua esimerkin 1 tapaan. Jos float-muuttujan on oltava kokonaisluku, on se määriteltävä esimerkin 2 tapaan.

- Esimerkki 1: float scale = 1; //väärin

- Esimerkki 2: float scale = 1.0; //oikein

Jokaisessa varjostinohjelmassa täytyy olla main()-niminen void-funktio määriteltynä. Tätä void-funktiota kutsutaan, kun ohjelma käynnistyy. Lisäksi jos erityyppisiä muuttujia halutaan muuttaa toisen tyyppisiksi, on kutsuttava kyseisen tyypin rakentajafunktiota.

- Esimerkki 3: vec3 v3 = vec3(v2, 1.0);

(GameDev.net, LLC.)

4 TAVOITTEET

Opinnäytetyön projekti on kaksiulotteista grafiikkaa piirtävän pelimoottorin valmistaminen C++-ohjelmointikielellä. Tällä pelimoottorilla on tarkoitus valmistumisen jälkeen luoda kaupallinen peli PC-laitteistolle. Pelimoottorin pääominaisuus on taustalataus järjestelmä, joka lataa pelin ajon aikana taustalla pelin käyttämää dataa keskeyttämättä peliä, mahdollistaen mahdollisimman sulavan ja keskeytymättömän pelikokemuksen. Pelimoottori sisältää vielä muutamia valaistustehosteita, joissa hyödynnetään suuntavektorikarttoja. Opinnäytteen onnistuminen edellyttää siis käyttökelpoisen pelimoottorin valmistamista kolmen kuukauden aikana toimivalla taustalatauslogiikalla ja valoefekteillä piirretyllä kaksiulotteisella grafiikalla.

Pelimoottorissa käytetään SDL 2.0 -ohjelmointikirjastoa esimerkiksi ikkunan luomiseen ja muuhun hyödylliseen. OpenGL-ohjelmointirajapintaa käytetään grafiikan piirtämiseen. Opinnäytetyön päätteeksi teen myös demopelin, joka esittelee pelin eri ominaisuudet.

Tämä pelimoottori ei tietenkään ole täysin valmis tämän opinnäytetyön projektin päätyttyä. Opinnäytetyön päätyttyä pelimoottorin kehittämistä jatketaan optimoiden sen koodia, lisäämällä toiminnallisuuksia ja luomalla mukaan scene-editorin. Täten saadaan aikaan käytettävä pohja pelien valmistusta varten.

5 SUUNNITTELU

Pelimoottorin suunnittelu toteutettiin jaksoittaisesti, koska tietyt asiat olivat jo ennestään tuttuja peliohjelmoinnista mutta jotkin asiat olivat täysin uusia. Pelimoottorissa tahdottiin saada tutummat perusasiat ensimmäisenä valmiiksi, jonka jälkeen voitiin miettiä uusia asioita kun ne alkoivat olla lähempänä toteutuksen tarpeellisuutta. Ensimmäisenä aloitettiin perusrakenteen suunnittelusta, johon sisältyi ikkunan luominen SDL 2.0-ohjelmistokirjastolla ja toistorakenteen toteutus. Tämän jälkeen suunniteltiin tarkemmin pelimoottorin käyttämiä perusluokkia, jotka toteutettiin nopeasti. Lopulta projektin edettyä siihen vaiheeseen, jossa taustalatausta tulisi suunnitella, alkoivat suurimmat haasteet tulla vastaan.

5.1 Luokkarakenne

Pelimoottorin luokkarakenne on suunniteltu välttämään ylimääräisen syötedatan tarpeellisuutta. Jos tiettyä dataa tarvitaan aina jonkin funktion toteuttamiseen ja kyseistä funktiota ei tarvita kuin yhteen tietynlaiseen operaatioon, jossa sen paluuarvolla ei ole väliä, ei kyseistä dataa pitäisi tarvita edes syöttää. Esimerkiksi jos Sprite-luokka aina tarvitsee piirtoonsa näytön kokosuhteen verrattuna pelin alkuperäiseen kokoon, pitäisi Sprite-luokan osata hakea se tieto suoraan siltä luokalta, joka hallitsee pelimoottorin ikkunaa. Tämän takia muilta luokilta ja olioilta dataa tarvitsevat luokat saavat osoittimen Game-luokkaan. Tämä Game-luokka hallitsee lähes kaikkia olioita ja luokkia pelimoottorissa, joten sen kautta voi päästä käsiksi mihin tahansa olioon ja dataan. Näin esimerkiksi ContentDrawManager-luokka, joka hallitsee pelin sisällön lataamista, voi lisätä Game-olioon lataamansa Scene-oliot.

```
manager->scenes.push_back(loadedScene);
```

Tämä siis tarkoittaa, että oliot voivat viitata Game-olioon, mutta Game-olio voi myös viitata näihin olioihin itseensä. C++-kielessä ei saa linkittää header-tiedostoja, jotka myös linkittävät takaisin niiden linkittäjään. Tämä voidaan kiertää esittämällä käytettävät ulkopuoliset luokat prototyyppeinä jokaisen luokan header (.h)-tiedostossa ennen omaa luokkamääritystä.

```
class Game;

class Sprite
{
public:
        Sprite();
        Game* manager;
};
```

Tällöin linkitys ulkopuolisten luokkien header-tiedostoihin tehdään lähdetiedostoissa (.cpp). Tämän toteutuksen ansiosta oliolle voidaan syöttää osoitin Game-luokan olioon samalla kun Game-oliolla on osoitin kyseiseen olioon itseensä.

5.2 Taustalataus

Pelimoottorissa ei ole tarkoitus toteuttaa tiedostojen osittaista lataamista, vaan kaikki tiedostot ladataan kokonaisina. Pelin maailma jaetaan pienempiin tasoihin, joita nimitetään sceneiksi. Nämä scenet tallennetaan tekstitiedostoihin.

Nämä scenet sisältävät datan niiden tarvitsemista kuvatiedostoista, äänitiedostoista ja kaikista pelin olioista. Kun pelaaja pelaa yhtä sceneä, ladataan taustalla sen viereiset scenet. Nämä scenet liitetään toisiinsa ankkuriolioilla, jotka kertovat pelimoottorille missä scene-olioiden sisältävät oliot sijaitsevat suhteessa toisiinsa kaksiulotteisessa koordinaatistossa. Näiden ankkurien ansiosta pelimaailman scenet voidaan asettaa vieretysten näyttämään yhtä kokonaista maailmaa. Täten peli voi jatkua ilman keskeyttäviä lataustaukoja.

Suurin ongelma taustalatauksen toteutuksessa on kuvatiedostojen käyttöönotto, sillä OpenGL-konteksti ei ole säieturvallinen. (Apple Inc.) Taustalatauksen on tapahduttava toisella säikeellä erillään pelin piirto- ja päivitys-operaatioista. OpenGL-kontekstia tulisi täten asettaa vuorotellen eri säikeiden käyttöön ja keskeyttää toisen säikeen toiminta jaksoittaisesti. Tämä ongelma voidaan kiertää lataamalla ensin kuvien data välimuistiin taustalataussäikeellä, jonka jälkeen peliä ajava säie määrätään ottamaan kuvat käyttöön. Kuvien kopioiminen kontekstiin tapahtuu lataamiseen verrattuna paljon nopeammin, joten tämä ei vaikuta merkittävästi pelin suorituskykyyn.

Vasta toteutuksen jälkeen opinnäytetyön loppuvaiheilla tuli ilmi, että SDL 2.0 -ohjelmistokirjasto sisältää funktion kontekstin omistajuuden vaihtamista varten säikeiden

välillä. Tämä olisi yksinkertaisempi ratkaisu ja säästäisi muistia, sillä kaikkia kuvatiedostoja ei tarvitsisi ladata kerralla välimuistiin ennen kontekstiin siirtämistä. Tätä ratkaisua aiotaan hyödyntää opinnäytetyön päätyttyä jatkokehityksessä. Tämä funktio on SDL_GL_MakeCurrent().

6 TOTEUTUS

Toteutus tapahtui jaksoittaisesti suunnittelun kanssa. Aloittaen ensin yksinkertaisemmista perusasioista kuten ikkunan luomisesta, pelin toistoajon toteutuksesta ja pelin käyttämistä olioluokista. Eli koko projektia ei suunniteltu kerralla, koska alussa ei voitu olettaa kaiken toimivan odotetusti. Uusien vaiheiden suunnittelussa otettiin huomioon edellisten toteutusvaiheiden tulokset. Toisin sanottuna projekti toteutettiin Scrumin kaltaisella ketterällä ohjelmistokehitysmallilla. Pelimoottorissa käytettiin SDL 2.0- ja tinyxml-ohjelmistokirjastoja. Koska Windowsin mukana tulevat OpenGL-päätetiedostot ovat vanhentuneita, piti ladata GLEW-ohjelmistokirjasto, joka lataa osoittimet uusimpiin OpenGL-funktioihin. (Wolff 2011, 8.)

6.1 Game-luokka

Game-luokka oli ensimmäinen pelimoottorissa toteutettu luokka. Pelimoottorin käynnistyessä luodaan Game-luokan tyyppinen olio ja kutsutaan sen run()-funktiota. Tämä funktio jatkaa toimintaansa while()-toistokierteessä niin kauan, kunnes ikkuna suljetaan, eli SDL_PollEvent() havaitsee SDL_QUIT-tapahtuman. Seuraavassa on koodiesimerkki SDL_QUIT-tapahtuman tarkistamisesta.

```
SDL_Event e;

while(!close)
{
	while(SDL_PollEvent(&e) != 0)
	{
		if(e.type == SDL_QUIT)
		{
			close = true;
		}
	}
}
```

Game-oliota luotaessa sen rakentajafunktio luo RenderEnvironment-olion, joka luo OpenGL-kontekstin ja pelin ikkunan. Kun ikkuna on luotu ja run()-funktiota kutsuttu, peli alkaa kutsumalla initGame()-funktiota. Tässä funktiossa on tarkoitus tehdä pelin aloitusta vaativat lataukset ja operaatiot. InitGame()-funktiossa kutsutaan myös ContentDrawManager-olion giveInitialScene()-funktiota, jolla asetetaan ensimmäisen Scene-tiedoston nimi.

Kun initGame()-funktio on ohi, run()-funktio luo toisen säikeen, joka aloittaa ensimmäisen kentän ja sitä seuraavien kenttien lataamisen taustalla. Sitten varsinainen toistoajo eli pelin operaatio alkaa, kunnes peli sammutetaan. Jokaisella toistoajolla kutsutaan funktioita update() ja draw(), jotka hallitsevat pelin grafiikan piirtoa ja olioiden päivittämistä.

6.2 Vector2D

Vector2D on kaksiulotteisen vektorin olioluokka. Se sisältää vain muuttujat X ja Y, jotka tallennetaan float-tyyppisinä muuttujina. Lisäksi se sisältää useita erilaisia funktioita, joiden avulla voidaan laskea erilaisia vektoriyhtälöitä. Vektorista voi saada funktiokutsulla magnitude() sen pituuden, ja angle()-funktio antaa sen osoittaman kulman radiaaneissa. Lisäksi vektoreita voi laskea yhteen, vähentää toisistaan, kertoa tai jakaa skalaariluvulla sekä laskea kahden vektorin välisen pistetulon funktiolla dot(). Vektorille löytyy myös rotaatiofunktio rotate(float radians) ja matriisilaskentafunktio multiplyByMatrix(float* matrix_2x2). Matriisilaskennassa vektori hyväksyy vain 2x2-kokoisia matriiseja, jotka on tallennettu float-taulukkoon (float[4]). Vektoriluokka ei tarvitse dataa itsensä ulkopuolelta muutoin kuin syötteenä, joten sillä ei ole osoitinta peliluokkaan.

6.3 Bound

Bound-olioluokka toimii kantaluokkana erilaisten muotojen päällekkäisyyksiä laskeville olioluokille. Alun perin sen perivät alaluokat luotiin törmäystunnistusta varten, mutta lopulta todettiin valmiin fysiikkamoottorin Box2D:n olevan parempi ratkaisu. Bound-luokan periviä luokkia ovat Circle, Triangle ja Rect, jotka kuvaavat ympyrää, kolmiota ja suorakulmaista nelikulmiota. Kaikki alaluokat osaavat käsitellä päällekkäisyystarkastuksen toisiinsa ja itsensä kaltaisiin olioihin. Bound-luokka ja sen alaluokat eivät tarvitse dataa ulkopuoleltaan muuten kuin syötteenä, joten niillä ei ole osoitinta peliluokkaan.

Päällekkäisyyttä tarkastellessa olio ensin tarkistaa kohdeolion tyypin, eli onko kyseessä nelikulmio Rect, kolmio Triangle vai ympyrä Circle. Nelikulmiot tarkistavat päällekkäisyyden yksinkertaisesti vertaamalla sijainti- ja kokoarvojaan. Tämä tarkastus monimutkaistuu, kun edes toinen näistä olioista sisältää rotaatiota. Tällöin verrataan ensin suuremmalla nelikulmiolla näiden kahden olion läheisyyttä. Täten jos nelikulmiot ovat tarpeeksi kaukana toisistaan,

voi olettaa, etteivät ne voi olla päällekkäin. Sen sijaan jos ne ovat tarpeeksi lähellä toisiaan, tarkistetaan, ovatko kummankaan nelikulmion kulmat toisen sisällä. Tämä laskenta on melko kevyt ja auttaa päättelemään, jos pienempi nelikulmio on kokonaan suuremman sisällä. Jos tämä tarkastus kertoo, että kummankaan nelikulmion kulmat eivät ole toisen sisällä, tarkistetaan vielä lopuksi, että leikkaavatko reunat toisiaan. Nämä laskennat toteutetaan tässä järjestyksessä niiden nopeuden mukaan ja varmistetaan mahdollisimman nopea toiminnallisuus.

Mikäli kyseessä on nelikulmion ja ympyrän välinen päällekkäisyys, tarkastetaan, onko ympyrän keskipiste nelikulmion sisällä. Tämän jälkeen jos ympyrän keskipiste ei ollut nelikulmion sisällä, tarkastetaan ympyrän keskipisteen etäisyys jokaisesta nelikulmion reunasta. Jos jokin reuna on lähempänä ympyrää kuin sen säteen mitta, tällöin ympyrä törmää nelikulmioon.

Kahden ympyrän välinen törmäystarkistus toteutetaan yksinkertaisesti laskemalla ympyröiden keskipisteiden etäisyys ja vertaamalla niiden säteiden summaan. Jos keskipisteet ovat lähempänä kuin niiden säteiden summa, on tapahtunut kahden ympyrän välinen päällekkäisyys.

Kolmioiden tapauksissa edetään lähes identtisesti nelikulmioiden tapaan, tarkistaen kulmien sisäkkäisyydet ja sitten reunojen leikkaukset. Kolmioiden ja ympyröiden välinen törmäys on myös erittäin samanlainen kuin ympyröitä verratessa nelikulmioihin.

Näiden luokkien poistoa harkittiin pelimoottoria toteuttaessa, kun niiden todettiin olevan lähes hyödyttömiä törmäyksen tunnistamisessa. Näistä olioista on kuitenkin ollut hyötyä optimoimaan piirtoprosessia ja määrittelemään etäisyyksiä halutuista pisteistä tiettyihin kaksiulotteisiin muotoihin, joten ne on jätetty osaksi pelimoottoria.

6.4 ContentDrawManager

ContentDrawManager-luokka hallitsee pelin sisällön lataamista, säilömistä ja piirtämistä. ContentDrawManagerin kautta ladataan kaikki pelimoottorin DrawObject-oliot, ja niiden olemassaolosta ja piirtojärjestyksestä pidetään kirjaa ContentDrawManagerin listoissa. Jokaisella DrawObject-oliolla on järjestysnumero, joka päättää sen piirtojärjestyksen. Kun Sprite-olio luodaan ContentDrawManagerissa, sille syötetään sen käyttämän kuvatiedoston nimi ja järjestysnumero getSprite()-funktioon. Emitter-oliolle ei ole omaa get-funktiota, sen luominen on hiukan monimutkaisempaa. Kuvatiedostoa ladatessa ContentDrawManager tarkistaa,

onko kyseistä kuvatiedostoa ladattu jo ennalta. Jos kuvaa ei ole ladattuna ContentDrawManagerin muistissa, se lataa sen. Muussa tapauksessa palautetaan jo ladatun kuvatiedoston tekstuuritunnus. Kun kaikki Drawobject-oliot, jotka käyttävät kyseistä kuvaa, poistetaan funktiolla removeDrawObject(), poistetaan myös kyseinen kuvatiedosto muistista. ContentDrawManager sisältää myös oman SDL_Mutex-muuttujan tekstuurien lataamista varten, varmistaen, etteivät käyttäjän tekemien Sprite-olioiden latauskutsut koskaan aiheuta ongelmia taustalatauksen kanssa.

6.5 DrawObject

DrawObject on kantaluokka piirrettäville olioluokille. Sillä ei ole omaa rakentajafunktiota, ja sen muut funktiot eivät tee mitään. Nämä funktiot sen sijaan määritellään uudelleen sen perivissä funktioissa. Muuttujina DrawObjectilla on bool active, joka kertoo ContentDrawManagerille, että tuleeko olio piirtää automaattisesti, ja float order, jolla määrätään automaattisen piirron järjestys kyseiselle oliolle. Lisäksi DrawObjectilla on indeksiluku sen käyttämään kuvatiedostoon ja suuntavektorikarttaan. DrawObject-luokan perimät alaluokat suorittavat omissa draw()-funktioissaan piirtonsa, koska ne sisältävät kyseistä piirtoa varten eniten dataa, tarviten vain muutaman muuttujan RenderEnvironment- ja ContentDrawManager-olioilta.

6.6 Sprite

Sprite on DrawObjectin perivä olioluokka, joka sisältää yhden kuvan piirtoon tarvittavan datan. Sprite sisältää kuvatiedostonsa tekstuuritunnuksen, jolla se viittaa kontekstissa tallennettuun dataan. Lisäksi Sprite sisältää kuvan tarkoitetun sijainnin, tarpeellisen datan kuvan kiertoa varten, kuvan koon, kuvan skaalausarvon ja tekstuurin koordinaatit. Sprite-luokka vaatii piirtofunktiossaan dataa RenderEnvironment- ja ContentDrawManager-olioilta, joten sille on syötettävä Game-luokan osoitin rakentajafunktiossa, jonka kautta se pääsee käsiksi näihin olioihin. Sprite-oliot, joiden boolean-muuttuja active on yhtä kuin true, piirretään automaattisesti ContentDrawManagerin toimesta. Muutoin ne on piirrettävä manuaalisesti kutsumalla niiden piirtofunktiota.

6.7 Emitter

Emitter on DrawObject-luokan perivä olioluokka, joka toimii ParticleSystem-luokan kanssa. Emitter on toisin sanottuna partikkeleiden luoja. Emitter kutsuu sille syötetyn väliajan mukaan ParticleSystem-luokan startParticle()-funktiota, johon se antaa osoittimen itseensä syötteeksi. Tämä funktio käynnistää yhden partikkelin liikkeen ja toiminnan Emitterin tietojen mukaisesti. Emitter piirretään lähes samalla tavalla kuin Sprite-olio.

6.8 ParticleSystem

ParticleSystem-luokka ajaa pelimoottorin partikkeleiden toimintaa. Sen rakentajafunktiossa luodaan valmiiksi syötetty määrä partikkeleita, ja näiden partikkelien määrää ei koskaan muuteta pelin ajon aikana. Sen sijaan partikkeleita otetaan käyttöön tai poistetaan käytöstä pelin aikana. Jos järjestelmältä loppuu partikkelit kesken, vanhin käynnistetty partikkeli määritellään uudelleen uutena partikkelina. Tällä tavalla ei synny muistivuotoja ja peli etenee tasatahtia, kun olioita ei tarvitse ladata tai poistaa muistista.

6.9 RenderEnvironment

RenderEnvironment-luokkaa ei luotu tai suunniteltu muiden perusluokkien kanssa. Sen tarpeellisuus tuli ilmeisemmäksi Game-luokan paisuessa erilaisilla ikkunan hallinnan funktioilla ja muuttujilla, joten pelimoottorin hierarkian selventämisen vuoksi RenderEnvironment-luokka luotiin ylläpitämään näitä asioita. RenderEnvironment-luokan rakentajafunktio luo pelissä käytetyn ikkunan ja OpenGL-kontekstin. RenderEnvironment-luokka pitää tallessa ikkunan koon dataa, sen kuvasuhdetta ja taustavaloarvoa. RenderEnvironment-luokan tärkein funktio on setVideoMode()-funktio, jota käytetään ikkunan luomiseen pelin alussa. Tämä sama funktio voi myös luoda pelin ikkunan uudelleen ja määrittää sille uuden koon tai näyttötilan ilman, että peliä tarvitsee käynnistää uudelleen. Kaikissa peleissä kuuluisi olla mahdollisuus vaihtaa ikkunan näyttötilaa ja kokoa ilman uudelleenkäynnistyksen pakkoa. Tämän toteutus ei vaadi muuta kuin pientä vaivaa ohjelmoijalta. Ikkunan uudelleenluonti SDL 2.0 -ohjelmistokirjastossa ei vaadi muuta kuin vanhan ikkunan tuhoamisen SDL_DestroyWindow()-funktiolla ja sen uudelleen luomista SDL_CreateWindow()-

funktiolla. Uusi ikkuna asetetaan käytettäväksi ikkunaksi SDL_GL_MakeCurrent()-funktiolla. OpenGL viewport on myös muistettava päivittää uuteen ikkunakokoon glViewport()-funktiolla. RenderEnvironmentin setVideoMode()-funktio osaa tehdä nämä asiat huolimatta siitä, onko vanhaa ikkunaa jo ennestään luotu vai ei. Kontekstia ei hävitetä, joten peli voi jatkaa toimintaansa samalla tavalla kuin ennenkin. Kaikki tarpeellinen data näytön koosta ja sen kuvasuhteesta päivitetään samalla, jotta DrawObject-oliot piirtyvät oikein. Lisäksi RenderEnvironment pitää tallessa ja päivittää pelimaailman taustavalaistuksen väriarvoa ja siltä voi hakea erilaisia varjostintehosteita varten kopion piirron aikana luodusta kuvasta pelin ikkunalta.

6.10 Scene

Tämä olioluokka sisältää yhden pelimaailman osan datan. Scene-olio on siis yksi pala suuremmasta kokonaisuudesta, josta koko pelin maailma muodostuu. Scene sisältää listat sen omistamista Sprite-, Bound- ja GameObject-olioista. Scene-olio sisältää myös oman taustavalaistuksen väriarvonsa, jota RenderEnvironment käyttää määrittelemään pelissä käytettävän taustavalaistuksen. Kun pelaaja etenee yhdestä Scene-oliosta toiseen, tämä väriarvo muuttuu sulavasti. Scenet tietävät omien naapuri-Scene-olioiden nimet ja sisältävät myös listan ladatuista naapureista. Scene hallitsee myös peliolioiden päivittämistä.

6.11 GameObject

GameObject eli peliolio on olioluokka, joka sisältää yhden peliolion sijaintidatan, sarjanumeron ja sen ladanneen Scene-olion nimen. GameObject-oliolle syötetään sen luomisen jälkeen Game-luokan initGameObjects()-funktiossa BaseObject-luokan perivän olion osoitin. Tämä osoittimen osoittama olio on itsessään se varsinainen peliolio, jota pelissä käytetään. Tämä toteutus mahdollistaa peliohjelmoijan luomien olioluokkien toiminnallisuuden taustalatauksen kanssa ilman peliohjelmoijan tarvetta muokata itse pelimoottoria. Peliolion sisältämän Scene-olion nimen ja sarjanumeron avulla estetään saman olion lataaminen uudelleen sellaisissa erikoistapauksissa, joissa olio on siirtynyt ulos sen lataaman Scene-olion alueelta. Esimerkiksi pelaaja tapaa fantasiaroolipelissä palkkasoturin majatalosta ja palkkaa tämän seuraamaan pelaajaa hänen retkellään pelimaailmassa muihin Scene-olioiden alueille. Pelaajan

palatessa kyseiseen majatalon sisältävään Scene-olioon pelimoottori huomaa, että kyseinen palkkasoturi on jo ladattuna pelin muistissa, joten kun majatalo ladataan uudelleen, kyseinen palkkasoturi jätetään lataamatta. Täten ei ilmaannu outoja klooneja pelihahmoista.

6.12 BaseObject

BaseObject on kantaluokka pelin käyttämille, peliohjelmoijan luomille olioluokille. BaseObject sisältää osoittimen sen omistamaan GameObject-olioon ja uudelleenmääriteltävän update()-funktion.

6.13 Light

Light on olioluokka, johon tallennetaan yhden valonlähteen data, kuten sijainti, suunta, väri ja vaikutusetäisyys. Light-luokalla ei ole muita funktioita kuin sen rakentajafunktiot. Light-olion ei tarvitse hallita omaa dataansa toimiakseen kunnolla, joten sen voisi myös korvata Struct-muuttujalla. Light-luokan rakentajafunktio vaatii syötteenä ContentDrawManager-olion, voidakseen lisätä ja poistaa itsensä ContentDrawManagerista automaattisesti. Jos Light-oliota ei aseteta ContentDrawManagerin listoihin, sitä ei käytetä, sillä DrawObject-oliot löytävät käyttämänsä Light-oliot sieltä.

6.14 Camera

Camera-luokka hallitsee pelin katselualueen sijaintia ja sen päivittämistä. Tälle luokalle on annettava ennen pelin alkua osoitin sijaintivektoriin ja siirtymänopeus. Camera pyrkii aina sille syötetyn sijaintivektorin luokse siirtymänopeutensa mukaan. Jos siirtymänopeus on nolla, kamera asetetaan suoraan sille syötetylle sijaintivektorille. Esimerkiksi tasoloikkapelissä kohteena olisi pelihahmon sijaintivektori. Tällöin Camera pyrkii aina keskittämään itsensä pelihahmon sijaintiin.

6.15 Sound

Sound-olioluokka toimii pelimoottorissa äänenlähteenä. Eli Sound-oliolla on jokin äänitiedosto ladattuna, jota se toistaa, kun sitä niin komennetaan. Sound laskee myös äänen tasapainon stereotoistossa ja äänen volyymin. Tähän laskuun käytetään äänen vaikutusetäisyysarvoa, joka kuvaa, kuinka kauas ääni kantaa ja sitä verrataan Camera-olion sijaintiin. Esimerkiksi jos Camera on kauempana kuin äänen kantavuus, ääntä ei edes toisteta. Jos äänen lähde on Camera-olioon verrattuna sen oikealla tai vasemmalla puolella, toistetaan tämänpuoleista stereota kovemmalla volyymillä kuin vastakkaista puolta.

6.16 Taustalatauksen toteutus

Ennen uuden säikeen luontia ContentDrawManagerille annetaan aloitus-Scenen nimi giveInitialScene()-funktiolla ja Scenen vaihtumisen sääntöjä hallinnoivat arvot giveStreamRuleData()-funktiolla. Nämä arvot ovat osoitin Vector2D-olioon ja etäisyyttä kuvaava liukuluku. Syötetty Vector2D-olio voi olla mikä tahansa vektori, mutta alkuperäisenä tarkoituksena on asettaa siihen joko pelin kameran tai pelihahmon sijaintia kuvaava Vector2D-olio. Näin ContentDrawManager tietää pelitilanteen sijainnin, ja kun kyseisen Vector2D-olion koordinaatit lähestyvät Scene-olion rajaa, tarkistetaan etäisyysarvolla, onko pelaaja jo tarpeeksi lähellä uutta Sceneä. Jos on, varmistetaan, että se on ladattu. Jos seuraava Scene-olio ei ole ehtinyt latautua, kun pelaaja pääsee tarpeeksi lähelle sitä, peli keskeytetään, kunnes Scene on latautunut. Kun nämä säännöt on asetettu, käynnistetään uusi säie, joka kutsuu threadOperation()-funktiota Game-luokassa. Tälle funktiolle syötetään ContentDrawManager-olio, joka jatkaa taustalatausta streamOperation()-funktiossa.

ContentDrawManager pitää listaa lataamattomista mutta tiedetyistä Scene-olioista, eli nykyisten Scene-olioiden naapureista. ContentDrawManager jatkaa niiden lataamista niin kauan, kun etäisyyslaskuri on suurempi tai yhtä suuri kuin nolla. Etäisyyslaskurin numero siis kuvaa, kuinka pitkältä etäisyydeltä Scenejä ladataan. Jos arvo on yksi, niin ladataan ainoastaan nykyisen Scene-olion naapurit. Jos arvo onkin kaksi, niin ladataan myös näiden naapureiden naapurit. Kyseinen etäisyyslaskuri asetetaan uudelleen, kun Sceneä vaihdetaan, eli sääntövektori siirtyy toisen Scene-olion pelialueelle. Tämä etäisyyslaskurin arvo riippuu pelattavasta Scene-oliosta, eli se voi vaihdella eri Scene-olioiden välillä.

Kun Scene ladataan, otetaan talteen sen käyttämien tiedostojen nimet ja polut, jotka ovat tallennettuina Scene-tiedostossa. Samalla Scenen erilaisia olioita luodaan tiedoston antaman datan mukaan. Kun kaikki oliot on luotu, aloitetaan tiedostojen lataaminen. Tiedostojen lataamisen ja käyttöönoton jälkeen pelin oliot asetetaan pelille käyttöön.

6.16.1 Kuvatiedostot

Scene-tiedostoja ladatessa otetaan listaan talteen niiden käyttämien kuvatiedostojen nimet. Samalla luodaan näitä kuvatiedostoja käyttävät Sprite-oliot kyseisen Scene-olion listaan. Kun oliot on rakennettu, aloitetaan kuvatiedostojen lataaminen. Tämä lataaminen, kuten muukin aiemmin tapahtunut Scene-tiedoston käsittely, tapahtuu edelleen toisessa säikeessä erillään pelin logiikkaa suorittavasta säikeestä. Datan siirtäminen kovalevyltä välimuistiin on hidasta, sillä kovalevyjä ei yleisesti tarvita nopeaan datan käsittelyyn, vaan sen säilömiseen. Kun kaikki kuvatiedostot on ladattu välimuistiin, ne otetaan käyttöön OpenGL-kontekstissa. Tätä käyttöönottoa ei suoriteta lataussäikeellä, sillä OpenGL-konteksti ei ole säieturvallinen. Tämä voitaisiin ratkaista asettamalla konteksti vuorotellen toisen säikeen käyttöön samalla kun toinen säie odottaa kontekstin vapautumista, mutta tämä ei ollut toteutusvaiheessa selvää. (Apple Inc.). Täten toinen säie asettaa itsensä odotustilaan ja antaa pääsäikeelle signaalin kuvatiedostojen valmiudesta. Pääsäie asettaa kuvatiedostot kontekstiin yksi Sprite-olio kerrallaan yhtä läpiajoa eli suoritettua ruutupäivitystä kohden. Koska dataa siirretään tässä vaiheessa välimuistista näytönohjaimen muistiin, on prosessi paljon nopeampi kuin kovalevyltä ladatessa, joten tämä siirto ei vaikuta pelin etenemiseen lähes lainkaan. Kun kuvatiedostot on asetettu OpenGL-kontekstiin, pääsäie antaa signaalin toiselle säikeelle, joka jatkaa toimintaansa poistamalla välimuistista tarpeettoman kuvadatan.

6.16.2 Pelioliot

Pelioliot eli GameObject-oliot ja BaseObject-luokan perivät oliot, joita pelin on osattava ladata taustalla huolimatta siitä, ovatko ne pelimoottorin omia vai peliohjelmoijan lisäämiä luokkia, vaativat hiukan erikoisen käsittelytavan. Scene-tiedostoihin tallennetaan GameObject-olioluokalle tarkoitettua metadataa, johon voi sisällyttää minkä tahansa peliohjelmoijan luoman olion data merkkijonossa. GameObject-oliota ladatessa tämä metadata ote-

taan talteen listaan ja luodaan GameObject-olio, joka sisältää muun tarpeellisen sijainti ja grafiikkadatan. Tämä olio ja sen metadata lähetetään käsiteltäväksi Game-olion initGameObjects()-funktiolle. Tämän funktion on tarkoitus olla uudelleenkirjoitettavissa peliohjelmoijan toimesta. Tämä funktio ajetaan kerran jokaista uutta GameObject-oliota kohden. Funktiossa tarkistetaan GameObject-olion tyyppinimi, joka sille annettiin Scene tiedostoa ladatessa. Jos esimerkiksi kyseisen olion olisi tarkoitus olla peliohjelmoijan luoma Treasure-olio, jota pelaaja keräisi saadakseen pisteitä, olisi kyseisen GameObject-olion tyyppinimi treasure. Kun funktiossa todetaan, että olio on tyyppiä treasure, luodaan uusi Treasure-olio. Tämän olion täytyy periä BaseObject-luokka toimiakseen oikein. BaseObject on olioluokka, joka sisältää osoittimen GameObject-olioon. Tälle GameObject-oliolle on myös annettava osoitin kyseiseen BaseObject-olioon Näin pelin Scene-olio, jolla on lista kaikista GameObject-olioista, voi päivittää peliohjelmoijan luomia BaseObject-olioita. Täten pelimoottori pystyy taustalatauksessaan käsittelemään myös peliohjelmoijien omia olioluokkia. Seuraavassa on koodiesimerkki initGameObjects()-funktion toteutuksesta.

```
void Game::initGameObjects(GameObject* object, std::vector<std::string> value-
Names, std::vector<std::string> valueData)
{
	//here you can transform game objects into your own classes (that inherit
	BaseObject class)

	if(object->type == "default")
	{
		BaseObject* base = new BaseObject(object);
	}
	else if(object->type == "treasure")
	{
		Treasure* base = new Treasure(object);
	}

}
```

6.17 Piirtäminen

ContentDawManager-olio suorittaa jokaisella läpiajolla luotujen DrawObject-olioiden piirron niihin asetetun piirtojärjestyksen mukaan. Jos DrawObjectia ei ole asetettu aktiiviseksi, sitä ei piirretä ContentDrawManagerin toimesta. Näiden DrawObject-olioiden piirtokutsun voi tehdä manuaalisesti, mutta niissä on otettava huomioon piirtojärjestys. Jos manuaalinen kutsu tehdään ennen ContentDrawManagerin piirtokutsua, ne saattavat jäädä muiden olioi-

den taakse ruudulla. Järkevin tapa käyttää manuaalista piirtoa olisi pelin valikoiden ja muiden ruutuelementtien piirtäminen ContentDrawManagerin piirron jälkeen.

6.17.1 Sprite

Sprite-olion piirto-operaation alussa tarkistetaan Sprite-olion sijainti pelin kameraan verrattuna. Jos Sprite sijaitsee edes hiukan kameran esittämän alueen sisäpuolella, se piirretään. Jokaisella Sprite-oliolla on niiden käyttämän varjostinohjelman tunnus tallessa. Oletuksena Sprite-oliot käyttävät yksinkertaisella valaistuksella toteutettua varjostinta, jossa ei huomioida suuntavektorikarttaa.

Sprite aloittaa piirtonsa asettamalla oikean tekstuurin käyttöön kontekstissa käyttäen sen tunnuslukua. Jos Sprite-oliolla on suuntavektorikartta, sekin asetetaan käyttöön sen omalle sampler-oliolle varjostinohjelmassa. Sprite-olio luo verteksidatan piirtoa varten. Kyseiseen verteksidataan lasketaan myös mukaan ikkunan kuvasuhde ja kokosuhde, jotka Sprite hakee RenderEnvironment-oliolta. Sijaintidata asetetaan uniform-tyyppiseen vektoriin varjostinohjelmassa. Verteksidataan ei lasketa sijaintia etukäteen, koska sitä ennen varjostinohjelman on kerrottava verteksidata rotaatiomatriisilla. Jos verteksidataan on jo valmiiksi laskettu sijainti ennen rotaatiomatriisin kertolaskua, kuva kiertyy väärän origon mukaan. Verteksidataan laitetaan vielä mukaan tekstuurikoordinaatit. Lopuksi haetaan valaistusta varten tarvittava data, jonka jälkeen Sprite-olio piirretään kutsumalla glDrawElements()-funktiota.

6.17.2 Emitter

Emitter-olioiden piirto toteutuu hyvin samankaltaisesti kuin Sprite-olioilla. Suurimpina eroina ovat kamera tarkistuksen puute ja verteksidatan rakentaminen. Emitter-oliot eivät pysty tehokkaasti päättelemään jokaisen partikkelinsa sijaintia suhteessa pelin kameraan, joten Emitter-olioiden piirtokutsu tapahtuu jokaisella läpiajolla. Emitterien verteksidata rakennetaan niiden luomien partikkeleiden mukaan. Jokaista partikkelia kohden määritellään nelikulmio, johon Emitter-olion tekstuuri piirretään. Emitterit voivat käyttää samoja varjostinohjelmia kuin Sprite-oliot. Emitterien valonlähteet haetaan niiden sijainnin mukaan. Tämä tietenkin tarkoittaa, että valaistuksessa voi ilmaantua puutteita, partikkelien edetessä pitkiä matkoja.

6.18 Valaistuksen toteutus

Pelimoottorin piirron valaistusefektin toteutus on melko yksinkertainen, kunnes mukaan lasketaan suuntavektorikartat ja suuntavalot. Ensimmäisenä haasteena on käytettävien valojen valitseminen. Pelimaailmassa saattaa olla kymmeniä valoja, mutta yksittäinen DrawObject-olio ei usein tarvitse kuin muutaman lähimmän. Lisäksi kaikkien pelimaailman valojen mukaan ottaminen jokaiseen olion piirtolaskentaan olisi hyvin raskasta.

6.18.1 Valojen valinta

Sprite-oliot valitsevat käyttämänsä valot piirtofunktionsa toteutuksessa. Pelimoottorin varjostinohjelmat tukevat neljää valonlähdettä ja yhtä taustavaloarvoa. Pelimaailmassa itsessään voi tietenkin olla useampia valonlähteitä kuin vain neljä. Täten oikeat valot on osattava valita oikein jokaista DrawObject-oliota kohden. Taustavalaistus, joka saadaan RenderEnvironment-oliolta, on yksinkertaisesti oletusväri, jolla kaikki tekstuurien väriarvot kerrotaan, eli toisin sanottuna taustavalo on pimeiden alueiden väri. Varsinaiset valonlähteet valitaan niiden tärkeysarvojen ja etäisyyksien mukaan. Ideana on antaa peliohjelmoijan päättää, mitkä valot ovat piirrossa kaikista tärkeimpiä, mutta silti antaa pelimoottorin päätellä käytettävät valot. Esimerkiksi jos pelihahmoa seuraa kirkas soihdun valo, tulisi tämän valonlähteen olla aina käytössä valaisemassa ympäristöä. Täten sen tärkeysarvo on suurempi kuin muiden valojen.

Sitten oletetaan pelimaailman ympäristössä ilmenevien valojen olevan tasa-arvoisia, jolloin niiden valintakriteerinä toimii niiden etäisyys kyseisestä piirrettävästä Sprite-oliosta. Täten pelihahmo, joka kulkee maailman läpi soihdun kanssa, aina valaistaan soihdun valolla, ja tilaa jää ympäristön muulle valaistukselle. Ympäristössäkin varmistetaan, ettei pelaajan ohi kulkiessa soihdun läsnäolo oudosti korvaa ympäristön valoja, kun ympäristön Sprite-oliot varaavat aina paikan tälle tärkeälle valolle.

6.18.2 Valaistuksen laskeminen

Valaistuksen tason laskeminen tapahtuu GLSL-ohjelmointikielellä toteutetussa varjostinohjelmassa, joka tunnetaan nimellä fragmenttiohjelma. Fragmenttiohjelma ajetaan kerran jo-

kaista polygonin fragmenttia tai piirrettävää pikseliä kohden. Tämä on siis se lopullinen vaihe, jossa väriarvo lasketaan. Pikselien väriarvon pohjana käytetään syötetyn kuvatiedoston väriarvoa, joka kerrotaan valaistuksen väriarvolla. Valojen väriarvo lasketaan yhteen neljästä valosta. Yhden valon väriarvo on valon väri kertaa valon vaikutusetäisyyden ja valon etäisyyden fragmenttiin erotuksella, joka jaetaan valon vaikutusetäisyydellä. Eli jos C = valon väri, D = valon vaikutusetäisyys, L = valon ja fragmentin välinen etäisyys, yhtälö on C * ((D – L) / D). Sulkujen sisäinen erotuslasku saattaa olla negatiivinen, joten käytämme GLSL:n max()-funktiota rajaamaan erotuksen pienimmän tuloksen nollaan, joka tarkoittaisi pimeätä arvoa. Tämä lasku toteutetaan neljä kertaa jokaista valoa kohden, lisäten kaikkien neljän laskennan tulokset yhteen. Seuraavassa on koodiesimerkki yhden valonlähteen arvon laskemisesta yhtä pikseliä kohden.

```
//Yhden valon väriarvon laskenta
vec4 diffuse = lightColor[0] * max(0, ((lightDist[0] - distance(lightPos[0], pos)) /
lightDist[0]));
```

Lopullinen valon arvo kerrotaan vielä itsellään eli potenssiin kaksi, jotta valon vaikutuskäyrä ei olisi täysin lineaarinen. Näin saadaan aikaan paljon luonnollisemman oloinen valaistus. Tähän lisätään vielä taustavalaistuksen arvo, jota käytetään tekstuurin väriarvon kertoimena.

6.18.3 Suuntavalot

Suuntavalot toimivat lähes täysin samalla tavalla kuin aiemmassa luvussa, mutta ne paistavat valoa vain rajoitettuun kulmaan asti. Tämä rajoitus toteutetaan laskemalla valon tulosuunnan ja valonlähteen osoittaman suunnan välinen pistetulo. Tästä pistetulosta sitten vähennetään syötetty rajoitinluku. Jos rajoitinluku on 0, suuntavalon säteen kulma on 180 astetta ja luvun ollessa 1 kulma on 0. Arvo jaetaan vielä numeron yksi ja rajoitinluvun erotuksella, sitten se rajataan max()-funktiolla estäen negatiiviset arvot. Valon lähteestä laskettu arvo kerrotaan tällä pistetulolla. Seuraavassa on koodiesimerkki pistetulon laskemisesta valaistusta varten ja sen käytöstä valon kertoimena.

```
//spotDir = valon lähteen osoittama suunta
//dir = valon tulosuunta
//coneLimit = rajoitinluku
//lightValue = valon väriarvo etäisyyden ja vaikutusetäisyyden mukaan

float dotProduct = max(0, (dot(spotDir, dir) – coneLimit) / (1 – coneLimit));
lightValue = lightValue * dotProduct;
```

6.19 Suuntavektorikartat

Suuntavektorikartta on tekstuuri, jonka väriarvoilla kuvataan suuntia. Punainen väri kuvaa X-akselin arvoa ja vihreä Y-akselia. GLSL-kielessä värejä käsitellään skaalassa 0–1. Suuntavektorit voivat kuitenkin osoittaa negatiiviseen suuntaan, joten tätä arvoa ei voi käyttää sellaisenaan. Väriarvot käännetään suuntavektoreiksi muuttamalla skaalaa seuraavan listan mukaan.

- 0 = -1
- 0,5 = 0
- 1 = 1

Yhtälö tähän muutokseen on X - 0,5 * 2.

6.19.1 Suuntavektorikartat valaistuksessa

Suuntavektorikartoilla voidaan luoda illuusio kolmiulotteisuudesta käyttämällä niitä valon laskennassa. Eli aiempiin valon laskentayhtälöihin lisätään kertoimeksi pistetulo näiden suuntavektoreiden ja valon vaikutussuunnan mukaan. GLSL-ohjelmointikielessä otetaan fragmentohjelmassa suuntavektorikartan väriarvo talteen texture()-funktiolla, kuten seuraavassa esimerkissä.

```
vec4 n = texture(normSampler, TextureCoord);
```

Väriarvo muutetaan käytettävän vektorin muotoon. Olion rotaatio on otettava huomioon, joten arvo käännetään ensin kaksiulotteiseksi vektoriksi, joka kerrotaan rotaatiomatriisilla. Lopuksi tarvitaan kolmiulotteinen vektori pistetulolaskentaa varten, sillä kaksiulotteinen vektori ei riitä saamaan aikaan haluttua tulosta. Z-arvon on aina oltava 1, jotta pistetulolaskentaan ei vahingossa syötettäisi nollavektoria. Seuraavassa on koodiesimerkki väriarvon muuttamisesta kolmiulotteiseksi vektoriksi.

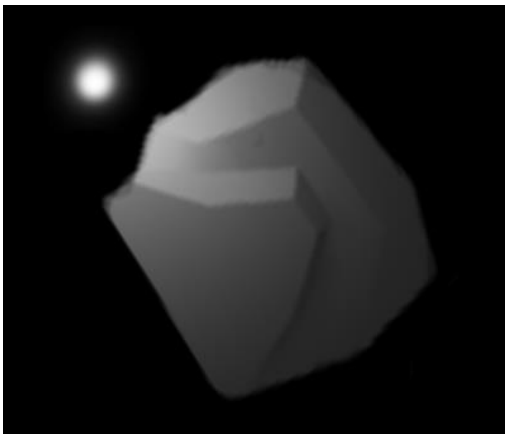
vec3 normalDir = vec3(vec2((n.x-0.5)*2, (n.y-0.5)*2) * rot, 1);

Ennen valaistuksen laskemista tarvitaan myös valon vaikutussuunnan vektori. Tämän vektorin kolmas arvo (Z) on myös aina 1. Tällä tavalla kun molemmat vektorit osoittavat Z-suunnassa aina suuntaan 1, varmistetaan, että tasaiset pinnat valaistaan positiivisella kertoimella. Seuraavassa on koodiesimerkki valon tulosuuntaa kuvaavan vektorin laskemisesta.

vec3 lightDir = vec3(lightPos[0] – pos, 1);

Lopulta näiden vektoreiden pistetuloa käytetään valon kirkkauden kertoimena. Seuraavalla koodiyhtälöllä saadaan aikaan kuvan 1 näköinen tulos.

```
//Yhden valon väriarvon laskenta
vec4 diffuse = lightColor[0] * max(0, ((lightDist[0] - distance(lightPos[0], pos)) /
lightDist[0])) * dot(lightDir, normalDir);
```



Kuva 1. -Kuva piirrettynä käyttäen suuntavektorikarttaa valaistuksessa

6.19.2 Valon taittuminen

Suuntavektorikartoilla voimme myös luoda hienoja efektejä, jotka simuloivat valon taittumista erimuotoisten lasien tai muiden valoa taittavien läpinäkyvien materiaalien läpi. Tähän tarvitaan ennen kyseistä DrawObject-oliota piirrettyjen DrawObject-olioiden tulos. Eli varjostin tarvitsee syötteeksi kuvan tähän asti piirretystä pelitilanteesta. Tämä kuva saadaan kopioimalla FBO-olion data. Kopiointi tapahtuu piirtämällä kyseisen FBO-olion data toisen FBO-olion tekstuuriin, joka syötetään varjostinohjelmalle. Valon taittamista laskevassa varjostimessa kaikki toiminnallisuus tapahtuu kuten ennenkin, paitsi lisäksi laskemme uuden taustavärin arvon hyödyntäen syötettyä kopiotekstuuria. Otamme jälleen suuntavektorikartalta

suuntavektorin talteen kuten valaistusta laskiessa. Taustaväri haetaan syötetystä kopiotekstuurista, syöttämällä texture()-funktiolle koordinaatiksi piirrettävän olion verteksisijainnin miinus suuntavektorikartan antama vektori. Täten kaksiulotteisella pinnalla taustaväriksi saadaan piirtotuloksesta ne värit, joihin valo taittuisi mennessään läpinäkyvän olion läpi. Taustaväri kerrotaan Spriten omalla värillä vielä lopuksi.

Kuva 2. -Läpinäkyvä kuva piirrettynä sokoittaen taustan värejä suuntavektorikartan mukaan

6.20 Varjostinohjelmat

Pelimoottorissa käytetään useampaa varjostinohjelmaa toteuttamaan erilaisia efektejä ja nopeuttamaan piirron etenemistä. Esimerkiksi kaikkien DrawObject-olioiden ei tarvitse laskea valaistusta suuntavektorikarttojen kanssa, joten tätä varten on nopeampi varjostinohjelma, jossa suuntavektorikarttoja ei käytetä. ContentDrawManager luo pelin käynnistyessään kaikki varjostimet ja luo niistä listan. ContentDrawManagerilta voi hakea varjostimia getShader()-funktiolla, joka ottaa syötteenä varjostinohjelman nimen.

Pelimoottorin varjostinohjelma ”default” toteuttaa valaistun kuvanpiirron neljällä valonlähteellä laskien mukaan suuntavektorikartan pistetulon. Tässä varjostinohjelmassa ovat siis valaistuksen kaikki ominaisuudet käytössä.

Yksinkertaisempaa valaistusta toteuttamaan on ”default_SimpleLight”, joka laskee valaistuksen samalla tavalla kuin ”default”, mutta suuntavektorikarttaa ei ole mukana laskennassa. Jos

DrawObject-oliolle ei ole määritelty erikseen käytettävää varjostinta, se käyttää tätä varjostinta.

Kun valaistusta ei tarvita tai haluta käyttää lainkaan, on olemassa ”default_NoLight”-varjostin, joka yksinkertaisesti piirtää kuvan sellaisenaan ruudulle. Tätä voi hyödyntää vaikka joissain partikkelitehosteissa, joissa varjojen ilmaantuminen Spriteissä olisi outoa. Esimerkiksi tulen liekeissä ei pitäisi olla varjoja.

Suuntavektorikarttoja lisää hyödyntäen on ”default_NormTransparent”-varjostin, joka käyttää syötettyä suuntavektorikarttaa laskemaan vääristyksiä aiemmin piirretystä kuvasta. Tämä varjostin vaatii toimiakseen kopion FBO:n piirtämästä pelitilanteesta.

Viimeisenä valikoiden ja muiden ruutuelementtien piirtämistä varten on ”default_HUD”, joka toimii lähes täysin samalla tavalla kuin ”default_NoLight”, mutta verteksiohjelmassa ei lasketa kameran sijaintia mukaan. Tällä varjostimella piirrettyjen Sprite-olioiden sijainti lasketaan siis ikkunan koordinaattien mukaan.

Nämä varjostimet sisältävät etuliitteen ”default_”, koska pelimoottorille voi syöttää omia varjostinohjelmia käyttöön. Jos syötetyn varjostimen nimeksi asetetaan jo aiemmin käytetyn varjostimen nimi, tämä varjostin korvataan. Täten ”default_” etuliitteellä estetään vahingollinen korvaaminen.

7 TESTAUS

Testausta tehtiin usein projektin eri ominaisuuksien valmistuessa, varmistaen uusien ominaisuuksien toiminnallisuutta. Vähän väliä projektissa toteutettiin suorituskyvyn stressitestejä ja muita erityisiä asetelmia varmistamaan, ettei muisti vuoda ja kaikki pelimoottorin osaset toimivat oikein. Pelimoottorin toteutuksen loppuvaiheilla toteutettiin myös testauksia erilaisilla kokoonpanoilla ja kahdella eri Windows-käyttöjärjestelmällä.

7.1 Muistivuotojen löytäminen

Muistivuotoja pelimoottorissa ilmeni kahdesti. Molemmilla kerroilla vuoto oli melkoisen suuri ja tapahtui Sprite-olioiden piirron sisällä. Muistivuodot havaittiin Windows käyttöjärjestelmän tehtävänhallinnan avulla. Tehtävienhallinnan prosessit-ikkunalla, katsomalla pelimoottorin prosessin varaaman muistin määrän muutoksia, voi vuotojen sattuessa havaita jatkuvaa muistinvarauksen nousua.

Muistivuotoja etsittiin kommentoimalla aina tietty osa pelimoottorin ajosta pois ja tarkastelemalla prosessin muistinvarausta sen ajon aikana. Jos muistivuoto lakkasi kommentoinnin jälkeen, voitiin olettaa muistivuodon tapahtuvan jossain kyseisen kommentoidun koodiosan sisällä. Tällöin kyseisestä alueesta kommentoitiin pienempiä osia, kunnes muistivuoto löytyi.

Molemmissa vuotojen tapauksissa kyseessä oli muistin varaaminen Sprite-olion piirtofunktiossa, jolloin muistia ei muistettu vapauttaa piirtämisprosessin päätteeksi. Toisessa tapauksessa asiaa ei voinut korjata vain muistin vapautuksella, sillä tämä aiheutti muistin korruptoitumista. Tällöin asia korjattiin luomalla kyseiselle tarvittavalle datalle muuttuja Sprite-olion sisään, johon se voitiin ottaa talteen myöhempää poistamista varten. Kyseinen data käsitteli Sprite-olion kattamaa aluetta ruudulla. Sitä käytettiin määrittelemään Sprite-olion sijaintia Camera-olioon verrattuna, jonka avulla voitiin arvioida piirtokutsun tarpeellisuutta.

7.2 Piirron suorituskyvyn testaus

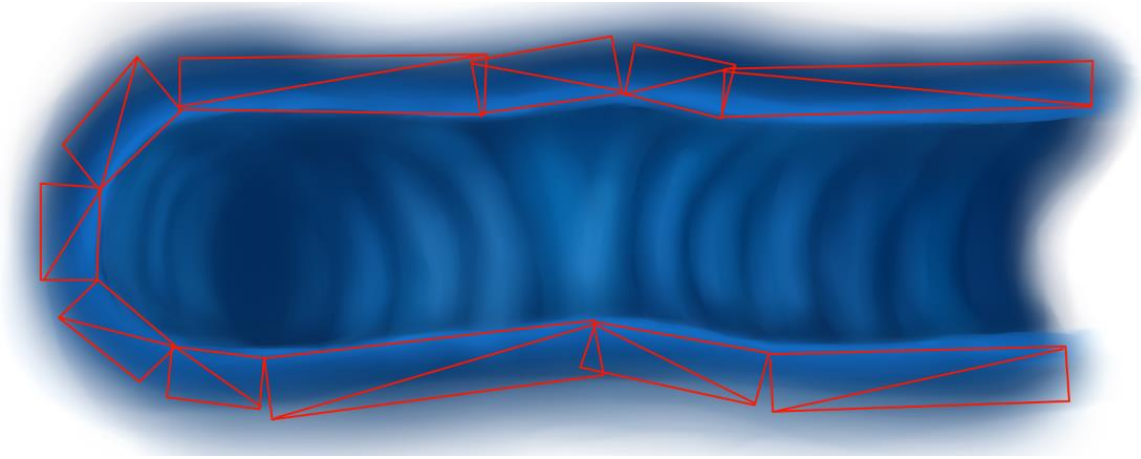
Pelimoottorin piirtoprosessin suorituskyky oli varmistettava, sillä pelimoottorilta odotetaan kykyä piirtää useita Sprite-olioita ruudulle nopeasti. Testauksessani minulla oli neljä erilaista Sprite-oliota, jotka yhdessä täyttivät ikkunan tilasta noin puolet. Tämä ei riittänyt hidastamaan pelimoottorin toimintaa, joten piirto asetettiin toteutumaan useamman kerran saman ruutupäivityksen aikana. Hidastumista alkoi ilmetä, kun piirtokertoja oli 150 kappaletta ruudun päivitystä kohden. Pelaamista haittaavaa hidastumista ilmeni 200 piirtokerran rajalla. Tähän on vielä otettava huomioon mahdollisen pelin operaatioiden vaatimat tehot, joten olisi hyvä pitää raja jossain 80 piirtokerran läheisyydessä. Täten pelimoottori pystyisi piirtämään ruudulle noin 320 Sprite-oliota yhden ruutupäivityksen aikana (4 Spriteä x 80 piirtokertaa). Jos pelimoottoriin lisätään ruutukartat, olisi hyvä suunnitella tapoja yhdistää useampia ruutuja yhdeksi olioksi, täten vähentäen piirtokutsuja. Tässä testissä ei testattu partikkelien piirtoa. Partikkelit kuitenkin piirretään yhtenäisinä olioina, joten ne eivät vaadi yhtä paljon prosessorin tehoa, kuten useamman Sprite-olion piirtäminen.

7.3 Päällekkäisyyksien tunnistamisen testaus

Päällekkäisyyksien testin tulokset oli paljon helpompi varmistaa kuin kahdessa aiemmassa. Yksinkertaisesti pelimoottoriin asetettiin kaksi erilaista Bound-oliota, joista toisen sijaintia pystyin hallitsemaan hiirellä. Näin voitiin siirtää toista oliota miten haluttiin, testaten erilaisia tilanteita. Päällekkäisyyksien tunnistuksen toimivuutta testattiin kaikilla eri Bound-olion alaluokilla ja visualisoin ne yksinkertaisella grafiikalla. Päällekkäisyyksien sattuessa konsoliin tulostui teksti ilmoittamaan siitä. Ensimmäisenä tässä testissä huomattiin, että Bound-olioiden ja niitä kuvaavia Sprite-olioiden pyöriessä piirtofunktion positiivisen pyörintäsuunta oli vastapäivään. Tämä asia korjautui syöttämällä radiaanit varjostimiin negatiivisina. Nyt sekä Bound-oliot että niitä kuvaavat Sprite-oliot käyttäytyivät samalla tavalla ja oli mahdollista varmistaa päällekkäisyyksien tunnistuksen toimivuus tarkasti.

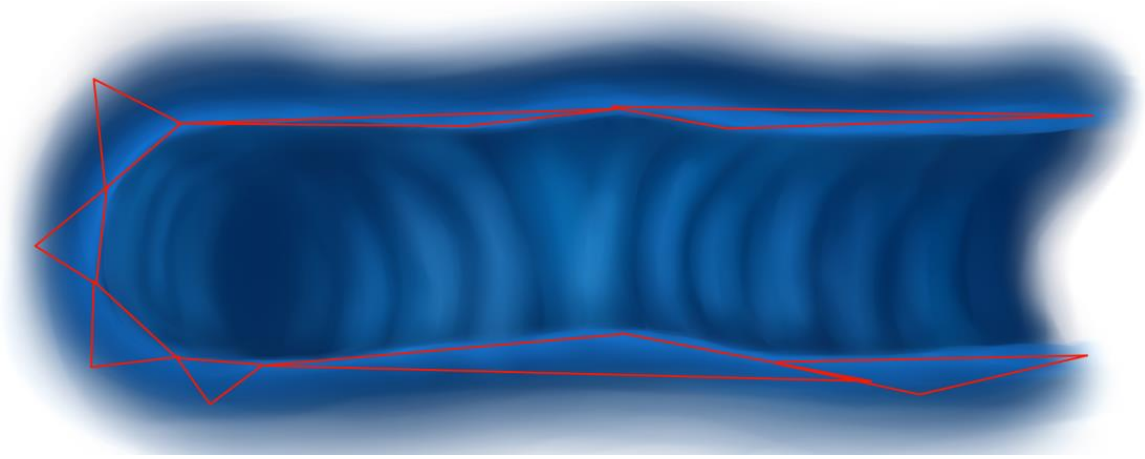
7.4 Bound-luokkien puutteet

Pelimoottorin demon grafiikkaa piirrettäessä tuli ilmi pelimaailmojen seinämien ylläpitämisen mahdollinen tehovaatimus. Demon grafiikan toteutuksen vaiheessa pelimoottorista löytyi vain luokat nelikulmioiden ja ympyröiden päällekkäisyyksille. Myöhemmin oli tarkoitus myös toteuttaa törmäyslogiikka. Tämä testaus tapahtui ennen kuin Bound-luokkien todettiin olevan kelvottomia törmäysten tunnistusta varten, ja niitä aiottiin vielä käyttää törmäyslogiikkaa varten. Pelimoottorin demon grafiikkaan päälle piirrettiin testausta varten nelikulmioista muodostuvat seinämät, jolloin voitiin nähdä, kuinka monta Rect-oliota tämän pelimaailman seinämät vaatisivat. Kuvasta 3 näkee, että tämä määrä ei ollut mahdottoman suuri.



Kuva 3. -Demossa käytetyn tunnelin seinät suorakulmioilla esitettynä

Nelikulmioiden toiminnallisuuteen kuuluu tarkistaa kaikki niiden neljä reunaa, eli myös ne seinämät, joihin pelihahmo ei edes pääsisi koskaan käsiksi. Kulmien välistä sisäkkäisyyttä tarkastellessa näissä nelikulmioissa on myös yksi kolmio liikaa. Testausta jatkaen piirrettiin suunnitelma seinämistä kolmioina, kuten kuvassa 4.



Kuva 4. -Demossa käytetyn tunnelin seinämät kolmioilla esitettynä

Testistä ilmeni, että kolmioilla saa aikaiseksi samat seinämät, mutta samalla säästetään tarkistettavien olioiden ja niiden reunojen määrässä. Tämän testauksen tuloksena todettiin, että kolmannen Bound-luokan perivän olioluokan Triangle-olion luominen olisi tarpeen pelimoottorissa, mikäli Bound-olioilla aiotaan toteuttaa törmäyslogiikkaa. Bound-luokkien tullessa tarpeettomiksi törmäysten tunnistamisessa tätä Triangle-oliota ei kuitenkaan enää tarvita.

7.5 Toimivuus eri kokoonpanoilla

Pelimoottorin demoa lähetettiin muutamille tuttavista koostuville testaajille kokeiltavaksi, jotta voitaisiin arvioida pelimoottorin toimivuutta erilaisilla kokoonpanoilla. Testissä vain kaksi testaajaa neljästä pystyi käyttämään pelimoottoria ilman ongelmia. Tarkkoja laitetietoja ei ollut saatavilla testissä, ja jotkin ongelmat toistuivat jopa kahdella täysin erilaisella kokoonpanolla. Yhtenä ongelmana ilmeni varjostimien kaatuminen, jopa silloin, kun testaaja käänsi oman demon suoraan lähdekoodista ilman virheilmoituksia. Muita ongelmia olivat musta ruutu, ilman mitään virheilmoituksia konsolissa, pelimoottorin jumittuminen latausikkunaan ja kaatuminen käynnistäessä. Myöhemmin testauksen jatkuessa tuli ilmi, että Windows-versiolla oli myös väliä pelimoottorin toiminnan kannalta. Kokoonpano, joka jäi latausikkunaan jumiin, toimi normaalisti vaihtaessa Windows 7 -käyttöjärjestelmään Windows 8:n sijasta. Pelimoottoria testattiin myös käyttäen vain yhtä prosessorin ydintä. Vaikka testatussa koneessa ydinten kellonopeus oli 4,4 ghz, pelimoottorin toiminnassa ilmeni suurta hidastumista. Vähintään kahden ytimen käyttö mahdollisti sulavan toiminnallisuuden.

8 ANALYYSI

Pelimoottorin tärkeimpien ominaisuuksien valmistuttua ja niitä esittävän demon toteutuksessa tuli ilmi monenlaisia puutteita ja rajoitteita, jotka vaatisivat vielä lisäkehitystä. Opinnäytetyön päättyessä pelimoottorilla pystyy luomaan melko yksinkertaisia kaksiulotteisia pelejä.

8.1 Törmäysten käsittely

Pelimoottorin demon myöhemmissä vaiheissa oli tarkoitus alkaa miettimään parasta tapaa käsitellä törmäyksiä, sillä tässä vaiheessa oli jonkinlainen pelikin, jossa niitä voisi testata. Kuitenkin on vaikea luoda tehokasta törmäyslogiikkaa, joka toteuttaisi mahdollisimman uskottavia ja virheettömiä tuloksia. Internetosoitteessa http://www.wildbunny.co.uk/blog/2011/03/25/speculative-contacts-an-continuous-collision-engine-approach-part-1/ kerrotaan näistä haasteista ja niiden ratkaisuista lisää. Pelimoottori pystyy omilla luokillaan havaitsemaan erilaisten muotojen päällekkäisyyksiä, mutta oikeaa törmäyslogiikkaa varten olisi parasta liittää pelimoottoriin suosittu Box2D-fysiikkamoottori.

8.2 Pelimoottorin käytettävyys

Opinnäytetyön päätyttyä pelimoottori soveltuu kaksiulotteisen grafiikan piirtämiseen monenlaisilla valaistusefekteillä. Pelimoottorin kyvyttömyys suorittaa törmäyksen logiikkaa vaatii peliohjelmoijaa soveltamaan sellaisen itse tai liittämään pelimoottoriin valmiin fysiikkamoottorin. Nykyisellä pelimoottorilla voi ilman peliohjelmoijan lisäyksiä luoda korttipelejä, pulmanratkontapelejä ja jonkinlaisia avaruusaluspelejä, joissa keskitytään enemmän tutkimusmatkailuun planeetoilla ja asteroideilla, jotka ovat taustakoristeena ennemmin kuin fyysisinä esteinä.

8.3 Efektien monipuolisuus

Pelimoottorin erilaiset varjostinohjelmat mahdollistavat hyvin monipuolisten efektien toteuttamista pelimoottorissa. Se pystyy laskemaan pistevaloja ja suuntavaloja jopa neljällä valonlähteellä jokaista DrawObject-oliota kohti. Lineaarisen valaistuksen sijasta voidaan simuloida kolmiulotteisten pintojen valaisemista suuntavektorikartoilla, ja valon taittumista voidaan imitoida läpinäkyvissä kuvissa. Kuvien piirtämisen sijasta voidaan toteuttaa myös partikkeliefektejä, joiden käyttäytymiseen voidaan vaikuttaa useilla eri arvoilla ja asetuksilla. Näitä efektejä yhdistelemällä oikein voi saada paljon aikaiseksi. Näiden efektien pitäisi riittää erinomaisesti mihin tahansa kaksiulotteiseen peliin.

8.4 Työn eteneminen

Pelimoottorin toteutus tapahtui paljon ripeämmin kuin alun perin oli suunniteltu. Pelimoottorin toteutuksen eri vaiheet oli suoritettu muutamissa päivissä verrattuna alun perin arvioituihin viikkoihin. Tämän ansiosta oli mahdollista toteuttaa useampia varjostinohjelmia pelimoottoriin ja saada aikaan näyttävämpi demo.

8.5 Jatkokehitystä

Pelimoottorissa on vielä useita puutteita aiemmin mainittujen törmäystunnistusten lisäksi. Pelimoottorin tarvitsisi tukea useampia tiedostotyyppejä. Tällä hetkellä pelimoottori osaa käyttää vain png-kuvatiedostoja tekstuuridataa varten ja ainoastaan wav-tiedostoja ääniä varten. Pelimoottorin demon tiedostokoko on eritoten musiikkitiedoston takia paljon suurempi kuin mitä sen tarvitsisi olla. Tämän takia mp3-tiedostojen tukeminen olisi hyvin tärkeää. On myös hyvin mahdollista, ettei pelimoottorissa tarvita muita kuvatiedostotyyppejä kuin png, kiitos sen alphakanavien, mutta ei useampien tiedostotyyppien tuesta ole haittaa. Lisäksi pelimoottori kaipaa mahdollisuutta ladata erilaisia asetuksia tiedostoista muokkaamaan sen toimintaa, ja funktioita kyseisten asetusten muuttamiseen ja tallentamiseen.

9 YHTEENVETO

Tämä opinnäytetyöprojekti oli hyvin mielenkiintoinen kokonaisuus. En ole aiemmin tehnyt pelimoottoreita vain pelimoottoreina, vaan pohjana ajankohtaista peliprojektia varten, jolloin nämä pelimoottorit eivät soveltuneet mihinkään muuhun peliprojektiin. Uudelleenkäytettävän pelimoottorin luominen oli omanlaisensa haaste eikä työ ole vielä edes täysin valmis. Aion jatkaa pelimoottorin kehittämistä tämän opinnäytetyön jälkeenkin kunnes pääsen siihen vaiheeseen, jolloin voin käyttää sitä haluamassani peliprojektissa. Pelimoottorin lähdekoodi on jo nyt ladattavissa githubista (https://github.com/Ilettaja/OpariMoottori) ja aion päivittää sinne myös tulevat versiot.

Suuri osa tämän pelimoottorin pääominaisuuksista toimisi myös kolmiulotteisessa pelimoottorissa vain muutamien viilausten jälkeen. Esimerkiksi pelimoottorin Scene-olioiden taustalataaminen pelin aikana. Pelimoottoreista oli myös paljon opittavaa teoriaosuutta kirjoittaessa. Pelimoottoreita tehdessä on useita satoja eri tapoja parantaa pelin toiminnallisuutta ja koodin luotettavuutta, kuten kahvat tai viisaat osoittimet. Tässä pelimoottorissani ei hyödynnetty tällaisia olioviittausluokkia. Lisäksi jatkokehitystä varten olisi hyvä alkaa erottamaan pelikoodia pelimoottorin toiminnasta. Tällä hetkellä käyttäjän olisi tarkoitus asettaa pelikoodinsa pelimoottorin funktioiden sisään, mikä ei ole aivan ihanteellinen tapa hoitaa asioita. Olen erittäin tyytyväinen aikaansaamaani taustalataustoiminnallisuuteen ja varjostinohjelmien visuaalisiin efekteihin. Jo ennen tätä projektia minua oli suuresti kiinnostanut suuntavektorikarttoja hyödyntävien efektien toteutus. Tässä projektissa sain ne viimein toteutettua.

LÄHTEET

Ansari, M. 2011. Game Developement Tools. 1. Painos.

Apple Inc. OpenGl Programming Guide for Mac – Concurrency and OpenGL. https://developer.apple.com/library/mac/documentation/graphicsimaging/conceptual/opengl-macprogguide/opengl_threading/opengl_threading.html (Luettu: 7.10.2014)

Carter, B. 2004. Game Asset Pipeline. 1. Painos.

GameDev.net, LLC. GLSL: An Introduction http://nehe.gamedev.net/article/glsl_an_introduction/25007/ (Luettu: 5.8.2014)

Gregory, J. 2009. Game Engine Architecture. 1. Painos

Lake, A. 2010. Game Engine Gems. 1. Painos.

Movania, MM. 2013. OpenGL Development Cookbook. 1. Painos.

Think Services, 2006. Streaming for Next Generation Games http://gamedeveloper.com/view/feature/130186/streaming_for_next_generation_games.php?print=1 (Luettu: 21.7.2014)

Wilston, K. 2005. A Streaming Bestiary. http://gamearchitect.net/Articles/StreamingBestiary.html (Luettu 22.7.2014)

Wolff, D. 2011. OpenGL 4.0 Shading Language Cookbook. 1. Painos.