3D-PELIMOOTTORIN TOTEUTUS

CASE: Project Cactus

Kalle Jokinen

Opinnäytetyö
Toukokuu 2017
Tietojenkäsittelyn koulutusohjelma
Ohjelmistotuotanto

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tietojenkäsittelyn koulutusohjelma
Ohjelmistotuotanto

JOKINEN, KALLE:
3D-pelimoottorin toteutus
CASE: Project Cactus

Opinnäytetyö 37 sivua, joista liitteitä 1 sivu
Toukokuu 2017

Opinnäytetyön toteutustapana oli yhdistelmä case-henkisestä tapaustutkimuksesta ja kokeellisesta ohjelmistokehityksestä. Työn tilaajana toimi Tampereen ammattikorkeakoulun tietojenkäsittelyn pelituotannon koulutusohjelma.

Opinnäytetyön tarkoituksena oli luoda pelimoottori ja tätä pelimoottoria hyödyntäen asiakkaan tarvetta vastaava 3D-mallien esikatselusovellus. Työn tavoitteena oli auttaa tilaajaa ja pelituotannon opiskelijoita ymmärtämään pelimoottorien sisäistä toimintaa.

Opinnäytetyöprosessin alussa perehdyttiin olemassa oleviin pelimoottoreihin ja tutkittiin niiden käyttäytymistä. Tutkimuksien tuloksena syntyi lista halutuista ominaisuuksista, joista karsittiin pois ne ominaisuudet, joita ei ollut mahdollista toteuttaa järkevässä ajassa. Pelimoottorin varsinainen toteutus oli lähinnä kokeellista ohjelmistokehitystä, jonka tavoitteena oli imitoida esikuviaan mahdollisimman tarkasti.

Työn tuloksena syntyi tilaajan tarvetta vastaava 3D-mallien esikatselusovellus. Lisäarvoa tilaajalle tuottaa oheistuotteena syntynyt pelimoottorin sisäistä toimintaa kuvaava dokumentaatio, jota tilaaja pystyy hyödyntämään osana koulutustaan. Lisäksi tilaaja pystyy hyödyntämään pelimoottorin lähdekoodeja parhaaksi katsomallaan tavalla.

Pelimoottorin toteutus oli todella laaja projekti, mutta siitä huolimatta se onnistui hyvin. Sen kehitystyö tulee jatkumaan harrasteprojektina opinnäytetyön palautuksen jälkeen yhteistyössä tilaajan kanssa. Ensimmäisenä opinnäytetyön ulkopuolisena työnä on mahdollistaa animaatioiden käyttö pelimoottorissa. Myös äänijärjestelmän toteuttaminen on korkealla työlistalla.

Asiasanat: peli, pelimoottori, renderöinti, ohjelmointi, C++

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Degree Programme in Business Information Systems
Option of Software Development

JOKINEN, KALLE:
Implementation of a 3D Game Engine
CASE: Project Cactus

Bachelor's thesis 37 pages, appendices 1 page
May 2017

The purpose of this thesis was to create a new game engine and use that engine to build software to match the client's need for an application to preview 3D models. The objective was to help game production teachers and students to understand how a game engine works. A combination of a case-study and experimental software development was employed as the chief study method. This thesis was commissioned by the Game Production unit at Tampere University of Applied Sciences.

The process began with research of pre-existing game engines, and as a result, a list of wanted features was created. Features requiring too much time had to be removed from the list. The actual implementation process can be described as experimental software development, the purpose of which was to imitate the chosen features.

As the result of this thesis an application was created to match the client's need to preview 3D models, and documentation describing the inner workings of the game engine was created as a side product. This documentation can be used by the client as a part of the teaching process, and the client may utilize the source code freely.

The scope of the game engine implementation project was huge. Nonetheless it turned out well. Development of the engine will continue as a hobby project by the author. In the next phase, priority will be given to support for animations and a sound system.

Key words: game, game engine, rendering, programming, C++

SISÄLLYS

1 JOHDANTO ... 6
2 PELIMOOTTORIN MÄÄRITELMÄ ... 9
3 PELIMOOTTORIN SUUNNITTELU ... 10
3.1 Peliobjekti ... 10
3.2 Skene-graafi ... 10
3.3 Muunnos ... 11
3.4 Komponentti ... 12
3.4.1 Start ... 12
3.4.2 Stop ... 12
3.4.3 Update ... 13
3.4.4 FixedUpdate ... 13
3.4.5 LateUpdate ... 13
4 GRAFIIKAN ESITYSRAJAPINTA ... 15
4.1 Geometria ... 15
4.2 Varjostinohjelma (Shader Program) ... 15
4.2.1 Kärkivarjostin (Vertex Shader) ... 16
4.2.2 Geometriavarjostin (Geometry Shader) ... 16
4.2.3 Kappalevarjostin (Fragment Shader) ... 16
4.3 Uniform-muuttuja (Uniform) ... 17
4.4 Tekstuuri ... 17
5 KÄYTETTÄVÄT TEKNOLOGIAT JA TYÖKALUT ... 18
5.1 Ohjelmointikieli ... 18
5.2 Ohjelmointiympäristö ... 19
5.3 Versiohallinta ... 19
5.4 Käännösautomaatio ... 19
6 PELIMOOTTORIN TOTEUTUS ... 21
6.1 Rajapinnat ... 21
6.2 Pääsilmukka ... 21
6.3 Renderöintimoottori ... 22
6.3.1 Suorarenderöinti ... 22
6.3.2 Viivästetty renderöinti ... 23
6.3.3 Valot ... 24
6.3.4 Renderöintiketju ... 24
6.4 Varjokartat ... 26
6.5 Kerrostettu varjokartta (Cascaded Shadow Map) ... 26

6.5.1 Suuntaamaton varjokartta (Omnidirectional Shadow Map) 27
6.5.2 Suunnattu varjokartta (Shadow Map) 28
6.6 Väriavaruus ja niiden muunnokset... 29
6.6.1 RGB 29
6.6.2 YCoCg 30
6.6.3 Muunnokset.. 31
6.7 Geometriapuskurin optimointi... 32
6.8 Tekstin renderöinti.. 34
POHDINTA 35
LÄHTEET.. 36
LIITTEET 37

LYHENTEET JA TERMIT

Attenuation	Vaimentuminen. Matemaattinen kaava, jolla kuvataan esimerkiksi äänen tai valon vaimenemista.
Cube texture	Kuution muotoinen tekstuurityyppi, joka koostuu kuudesta kuution eri pinnasta.
Diffuse color	Hajontaväri. Väri, joka tulee esiin kohteesta valaistuna.
FPS-peli	First Person Shoote, Ensimmäisen persoonan ammuntapeli, Ampumiseen keskittyvä peli, jossa pelimaailma kuvataan pelihahmon silmin.
Geometriapuskuri	Geometry Buffer, G-Buffer, puskuri johon lasketaan ruudulla näkyvä geometria jälkiprosessointia varten.
IDE	Integrated Development Environment, Kehitysympäristö.
Interpolointi	Laskennallinen asetuspiste kahden pisteen välillä ajan funktiona.
Komponentti	Component, Peliobjektille lisättävä toiminnallisuus.
Mesh	3D-mallin geometriakartta tietokoneen ymmärtämässä muodossa.
Muunnos	Transformation, koostettu tieto sijainnista, kierrosta ja skaalauksesta. Sisäisesti esitetty transformaatiomatriisina.
Normaalivektori	Vektori, joka osoittaa aina käsiteltävän kolmion ulkoreunasta ulos.
OpenGL	Open Graphics Library, standardoitu ohjelmointirajapinta, joka soveltuu sekä 2D, että 3D grafiikan esitykseen.

Peliobjekti	GameObject, Pelimoottorin perusrakennuspalikka.
Render pass	Renderöintioperaation työvaihe.
Renderöijä	Renderer, ohjelmiston osa, joka on vastuussa tietokonegrafiikan rasteroinnista ruudulle.
RGB	Väriavaruus, jossa jokainen kanava vastaa värin voimakkuutta.
Signed distance field	Etumerkillinen etäisyyskartta, joka kertoo etäisyyden kiinteästä reunasta.
Skene-graafi	Scene Graph, pelin hallinnoima kokonaisuus. Esimerkiksi kenttä, joka sisältää kaikki ruudulla näkyvät esineet.
Specular color	Heijastusväri. Väri, joka heijastuu takaisin kohteesta.
YCoCg	Väriavaruus, jossa kanavien arvot vastaavat kirkkautta, oranssin värikkyysmuutosta ja vihreän värikkyysmuutosta.

1 JOHDANTO

Videopelien luominen on nykyään helpompaa kuin koskaan ennen kiitos laadukkaiden ja edullisten pelimoottorien. Näiden pelimoottorien tarjoamat työkalut ovat niin helppokäyttöisiä, että lähes kuka tahansa pystyy niitä käyttäen luomaan omia pelejä. (Unity Learn 2017.)

Myös monet suurista pelitaloista hyödyntävät valmiita pelimoottoreita. Usein on taloudellisesti kannattavampaa hyödyntää valmista pelimoottoria, kuin kirjoittaa oma jokaiselle pelille. Laadukkaita pelimoottoreita on mistä valita. Esimerkkeinä mainittakoon Unreal Engine ja Unity.

Valitettavasti tämä suuntaus johtaa siihen, ettei moni enää tiedä mitä kaikkea pelimoottori tekee pelikehittäjän puolesta. Ymmärrys pelimoottorin toiminnasta auttaisi pelikehittäjiä kirjoittamaan suorituskykyisempää koodia ja hyödyntämään pelimoottorien ominaisuuksia laajemmin.

Opinnäytetyön tilaaja on yksi suurimmista alalle kouluttavista tahoista, joka hyödyntää valmiita pelimoottoreita opetuksessaan. Tilaajalla oli tarve sovellukselle, jolla pystytään aiempaa monipuolisemmin esikatselemaan 3D-malleja. Tämän lisäksi tilaajalle lisäarvoa tuo selvitys pelimoottorin sisäisestä toiminnasta.

Opinnäytetyön tarkoituksena oli kirjoittaa uusi pelimoottori ja sitä pelimoottoria käyttäen luoda tilaajan tarpeita vastaava 3D-mallien esikatselusovellus. Oheistuotteena työlle syntyi pelimoottorin toimintaa kuvaava dokumentaatio, jota tilaaja voi hyödyntää koulutuksessa.

2 PELIMOOTTORIN MÄÄRITELMÄ

Pelimoottorin käsite syntyi 1990-luvun puolivälissä, kun ohjelmoijat alkoivat eriyttää pelien sisältöä niiden toimintalogiikasta. Tämä mahdollisti sen, että samaa koodikantaa oli mahdollista käyttää myös seuraavissa projekteissa. Tällöin käsite ei ollut vielä vakiintunut, mutta ohjelmoijat luoneet pelimoottorin. (Gregory 2009, 11.)

Sisällön eriyttäminen mahdollisti uusien, samankaltaisten pelien luomisen nopeasti. Oli tarpeen vain luoda uutta pelisisältöä (tarina, grafiikat, äänet, pelitasot yms.) sen sijaan, että peli olisi pitänyt ohjelmoida alusta alkaen. Tästä alkoi ajanjakso, jolloin kehittäjät alkoivat lisensoida omia pelimoottoreitaan ulkopuolisille tahoille. (Gregory 2009, 11.)

Pelimoottorien syntyyn vaikutti vahvasti id Softwaren vuonna 1993 julkaisema peli *Doom*. Doom oli ensimmäisen persoonan ammuntapeli (FPS-peli) ja se saavutti suuren suosion. Doomin suosion myötä id Software kehitti ja julkaisi myös pelin nimeltä *Quake*. Myös Quake menestyi erinomaisesti ja sen vaikutukset ovat yhä nähtävillä. Esimerkiksi Valve Corporationin tunnettu pelimoottori Source Engine pohjautuu kaukaisesti Quaken pelimoottoriin. (Gregory 2009, 25.)

3 PELIMOOTTORIN SUUNNITTELU

Pelimoottorin suunnitteluvaiheessa on hyvä tiedostaa, ettei yksi pelimoottori yksinkertaisesti sovellu välttämättä kaikille peligenreille. Tieto siitä millaisia pelejä pelimoottorilla tulisi pystyä toteuttamaan ohjaa suunnitteluprosessia hyvin vahvasti. Tämä tulisi myös pitää mielessä koko prosessin ajan ja suunnitella moottoria siten, että sitä olisi mahdollista laajentaa myöhemmin sen sijaan, että yrittäisi saada kaikkia toiminnollisuuksia pakattua moottoriin itseensä.

Esimerkkitoteutuksen suunnittelussa suurimmat yksittäiset vaikuttajat olivat tilaajan nykykäytännöt ja Unity-pelimoottorilla tapahtuva opetus. Tästä syystä pelimoottorin tuli olla mahdollisimman helposti omaksuttava Unityä ennestään käyttäneille. Tämän vuoksi pelimoottori vastaa useilta toiminnoiltaan Unityä.

3.1 Peliobjekti

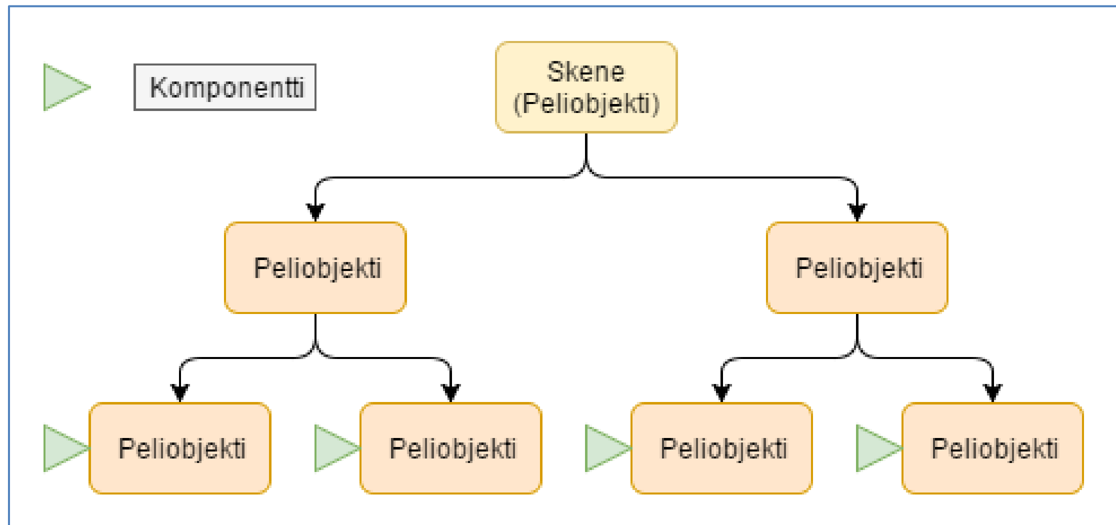
Peliobjekti (*GameObject*) on pelimoottorin pienin ja samalla tärkein osa. Kaikki ruudulla näkyvät objektit ovat tyypiltään peliobjekteja. Peliobjekti ei itsessään tuota ruudulle vielä mitään näkyvää, vaan se toimii säiliönä muille peliobjekteille ja komponenteille.

Peliobjekti sisältää vähintään tiedon omasta sijainnistaan 3D-avaruudessa (muunnos) ja sen lisäksi mahdollisesti lapsiobjekteja ja komponentteja ennalta määrittelemättömän määrän. Peliobjektit ja niiden väliset suhteet muodostavat rungon skene-graafille.

3.2 Skene-graafi

Skene-graafi (*Scene Graph*) periytyy peliobjektista. Se kuvastaa ruudulla näkyvän peliympäristön juuriobjektia ja pitää kirjaa muun muassa valoista. Skene-graafilla yleisesti esitetään senhetkistä kenttää (map/level), mutta sitä voidaan myös hyödyntää esimerkiksi pelivalikoissa.

Skene-graafien välillä on mahdollista siirtyä ohjelmallisesti, mutta käytännön syistä niitä ei ole suositeltavaa pitää useampaa yhtäaikaisesti ladattuna muistissa. Peliobjektit voivat olla osa vain yhtä skene-graafia kerrallaan. Skene-graafin yksinkertaistettu rakenne esitetty kaaviossa 1.



KAAVIO 1. Yksinkertainen skene-graafi

3.3 Muunnos

Jokaisella peliobjektilla on oma muunnoksensa (*Transformation*), joka kuvastaa kyseisen objektin sijaintia ja asentoa 3D-avaruudessa. Muunnos sisältää peliobjektin sijaintitiedon vektorina (vector), kierron kvaterniona (quaternion) ja skaalauksen vektorina.

Oman sijaintinsa lisäksi muunnoksella on tieto ylemmän tason muunnoksesta, joka tulee huomioida lopullisessa matriisissa. Kun näistä lähteistä lasketaan matriisit ja ne kerrotaan ylemmän tason muunnoksella, saadaan laskettua lopullinen transformaatiomatriisi, joka sisältää kaiken edellä mainitun informaation yhdessä 4x4 kokoisessa matriisissa. Tämä matriisi on vastuussa objektin lopullisesta sijainnista ruudulla.

3.4 Komponentti

Komponentit (Component) ovat peliobjektien rinnalla pelimoottorin tärkeimpiä osia. Pelin kehittäjät saavat lisättyä peliobjekteille toiminnollisuuksia kirjoittamalla itse omia komponentteja, jotka tekevät juuri sitä, mitä kehittäjä haluaa niiden tekevän. Komponentit voivat kysyä omilta isäntä-peliobjekteiltaan näiden muunnoksia viittauksia muihin komponentteihin. Näin komponentit voivat vaikuttaa peliobjektin sijaintiin ruudulla ja ne voivat kommunikoida keskenään.

Tämä tarkoittaa myös sitä, että komponenttien vastuualueet on helppo määrittää, eikä niillä ole liikaa sivuvaikutuksia. Komponenttien sisäinen toiminta perustuu siihen, että kehittäjät ohittavat haluamansa virtuaaliset metodit ja kirjoittavat haluamansa toiminnallisuuden niiden avulla. Ohitettavia metodeita on muutamia erilaisia, joista osa on tarkoitettu puhtaasti pelimoottorin sisäiseen käyttöön. (Unity Manual 2016.)

3.4.1 Start

Kun pelimoottori käynnistää sen hetkisen skene-graafin, kutsutaan kaikkien sen alaisten peliobjektien ja komponenttien start-metodia. Start-metodissa yleisesti luetaan välimuistiin viittaukset muista objekti ja komponentti ilmentymistä. (Unity Manual 2016.)

Välimuistiin lataaminen on tärkeää, sillä se vähentää huomattavasti pelimoottorin kuormaa verrattaessa siihen, että jokaisella objektin tai komponentin päivitysmetodikutsulla etsittäisiin haluttu ilmentymä. Ilmentymän etsintä on laskennallisesti raskasta ja se hukkaa kallisarvoista prosessoriaikaa.

3.4.2 Stop

Pelimoottorin vapauttaessa skene-graafia, kutsutaan kaikkien sen alaisten peliobjektien ja komponenttien stop-metodia. Samoin kuin Start-metodia, tätä kutsutaan vain kerran. (Unity Manual 2016.)

Stop-metodissa yleisesti vapautetaan välimuistiin ladatut resurssit ja suljetaan mahdolliset tiedostokahvat. Mikäli varattuja resursseja ei ole, voidaan Stop-metodi jättää toteuttamatta.

3.4.3 Update

Ennen aktiivisen ruudun renderöintiä kutsutaan kaikkien skene-graafin alaisten objektien ja komponenttien update-metodia. Update-metodia käytettäessä on syytä suorittaa vain hyvin nopeita toimenpiteitä, sillä tätä metodia kutsutaan yhtä monta kertaa sekunnissa kuin ruutu päivittyy. Tämä päivitysnopeus ei ole vakio. (Unity Manual 2016.)

Ruudunpäivitysnopeuteen ja sen myötä update-metodiin vaikuttaa huomattavasti renderöitävän skenen monimutkaisuus. Mikäli update-metodia halutaan käyttää peliobjektien liikuttamiseen tai muuten aktiiviseen laskentaan, tulee siinä hyödyntää ruudun piirtämiseen kulunutta delta-aikaa interpoloimaan oletettavaa sijaintia.

3.4.4 FixedUpdate

FixedUpdate-metodi on lähes vastaavanlainen, kuin Update-metodi. Erona näillä on kuitenkin se, että FixedUpdatea kutsutaan kiinteällä aikavälillä, kun taas Update-metodia kutsutaan ennen jokaista ruudun piirtämistä. (Unity Manual 2016.)

FixedUpdaten päivitysnopeudeksi vakiintui pelimoottorin suunnitteluvaiheessa 60 kertaan sekunnissa. FixedUpdate soveltuukin erinomaisesti esimerkiksi pelin fysiikoiden mallintamiseen sen käytöksen ennustettavuuden ansiosta.

3.4.5 LateUpdate

LateUpdate-metodia kutsutaan yhtä usein kuin Update-metodia. Ero näillä metodeilla on se, että LateUpdate-metodia kutsutaan, kun ruudun renderöinti on valmis. Tähän pistee-

seen saavuttaessa kaikki käyttäjäsyötteet ja sisäisten tilojen päivitykset ovat jo tapahtuneet, joten tämä soveltuu erityisen hyvin esimerkiksi pelaajaa seuraavan kameran toteutukselle. (Unity Manual 2016.)

4 GRAFIIKAN ESITYSRAJAPINTA

Tulevissa kappaleissa esiintyy 3D-grafiikan esitysrajapinnoissa käytettyjä termejä. Termien täyttä ymmärrystä ei lukijalta vaadita, mutta niitä on avattu seuraavien osalta: geometria, varjostinohjelma, kärkivarjostin, geometriavarjostin, kappalevarjostin, uniform-muuttuja ja tekstuuri.

4.1 Geometria

Näytönohjaimella grafiikkaa piirtäessä tulee piirrettävän objektin olla näytönohjaimen ymmärtämässä muodossa. Näytönohjain pystyy käsittelemään pisteitä, viivoja, kolmioita (triangles) ja nelikulmioita (quads).

Tämä ei tarkoita sitä, että näytönohjain pystyisi käsittelemään kaikkia muotoja yhtä tehokkaasti kuin kolmioita, sillä sisäisesti näytönohjaimet joutuvat kuitenkin muuntamaan nelikulmiot kolmioiksi. Tästä syystä monimutkaisia 3D-malleja piirrettäessä tulisi käyttää lähes poikkeuksetta kolmioita. (OpenGL Wiki 2017.)

4.2 Varjostinohjelma (Shader Program)

Varjostinohjelma on sovellus, jota ajetaan useimmiten näytönohjaimella. Yksi varjostinohjelma voi sisältää useita varjostimia. Vaatimuksena kuitenkin on, että jokainen varjostin vastaa liukuhihnan eri vaiheesta.

Mikäli varjostinohjelman varjostimet sisältävät Uniform-muuttujia, ovat ne kaikkien eri liukuhihnan vaiheiden käytössä tasavertaisesti. Yleisimmät varjostintyypit ovat kärkivarjostin, geometriavarjostin ja kappalevarjostin. (OpenGL Wiki 2017.)

4.2.1 Kärkivarjostin (Vertex Shader)

Yleisin varjostintyyppi on kärkivarjostin, joka on samalla liukuhihnan ensimmäinen vaihe. Tässä vaiheessa varjostin saa piirrettävän objektin geometrian näytönohjaimelta ja käsittelee sitä kolmion kärkinä. Kolmion kärjet ovat tässä vaiheessa liukuhihnaa vielä mallin-avaruudessa (model space).

Useimmissa tapauksissa kärkivarjostin siirtää mallin omasta avaruudestaan lopulliseen maailman-avaruuteen (world space) käyttäen muunnosmatriisia. Muunnosmatriisi annetaan yleisesti varjostimelle Uniform-muuttujana. (OpenGL Wiki 2017.)

4.2.2 Geometriavarjostin (Geometry Shader)

Geometriavarjostin on harvemmin käytetty varjostintyyppi. Mikäli sellainen on käytössä, istuu se liukuhihnalla heti kärkivarjostimen perässä. Geometriavarjostin saa sisääntulevan datan suoraa kärkivarjostimelta.

Geometriavarjostimen vastuulla on generoida sisään tulevasta datasta uutta geometriaa seuraavaa varjostinvaihetta varten. Käytännössä tämä voi tarkoittaa esimerkiksi sitä, että kärkivarjostin syöttää geometriavarjostimelle kuution. Sen pohjalta generoidaan pallo, joka syötetään seuraavalle vaiheelle. (OpenGL Wiki 2017.)

4.2.3 Kappalevarjostin (Fragment Shader)

Kappalevarjostin on varjostintyypeistä viimeinen ja samalla raskain, sillä sen käsittelemä datamäärä on huomattavasti suurempi, kuin aiempien varjostinvaiheiden. Kappalevarjostin vastaa jokaisen yksittäisen pikselin piirtämisestä ruudulle.

Liukuhihnan alkupäässä data oli kolmion kärkipisteinä, mutta tässä vaiheessa liukuhihnaa näytönohjain on käsitellyt kärkipisteet ja data on nyt kolmion sisältäminä pikseleinä. Tässä varjostinvaiheessa tapahtuu myös valojen ja varjojen käsittely, jotka vaikuttavat piirrettävän pikselin väriin. (OpenGL Wiki 2017.)

4.3 Uniform-muuttuja (Uniform)

Uniform-muuttuja on varjostinohjelmassa määritelty muuttuja, johon voidaan ohjelmallisesti syöttää tietoa prosessorilla ajettavasta ohjelmasta. Tällainen muuttuja on perimmäiseltä olemukseltaan yksisuuntainen, eikä siitä ole mahdollista lukea tietoa kuin ainoastaan varjostinohjelman sisällä.

Sisäisesti varjostinohjelmassa Uniform-muuttuja voi olla tyypiltään lähes millainen vain. Vastuu datan oikeamuotoisuudesta on Uniform-muuttujaan kirjoittavalla prosessilla. Oikeaoppinen data voi olla esimerkiksi kokonaisluku (int), liukuluku (float), vektori (x, y, z), tai 4x4 matriisi (float[16]). (OpenGL Wiki 2017.)

4.4 Tekstuuri

Tekstuuri on esitysrajapintakohtainen olio, joka sisältää yhden tai useamman näytönohjaimen muistiin ladattavan kuvan. Näitä kuvia voidaan käyttää ruudulla näkyvien objektien päällystämiseen, eli teksturointiin.

Tekstuuri voi myös olla renderöinnin kohteena, jolloin piirto-operaatiot eivät muuta suoraan näkyvää kuvaa. Tällaisessa skenaariossa piirto kohdistuu tekstuuriin, jolloin myöhemmässä vaiheessa tätä kohdetekstuuria voidaan käyttää samoin kuin mitä tahansa muuta tekstuuria. (OpenGL Wiki 2017.)

5 KÄYTETTÄVÄT TEKNOLOGIAT JA TYÖKALUT

Pelimoottorin toteutusta edelsi käytettävien teknologioiden ja työkalujen valinta. Teknologioiden tulisi olla laajalti levinneitä ja vakiintuneita. Työkalujen puolestaan tulisi olla helposti saatavilla ja niiden tulisi olla mahdollisimman helppokäyttöisiä.

5.1 Ohjelmointikieli

Anthony A. määrittelee ohjelmointikielen tietokoneohjelmien kirjoitusasuna kirjassaan ”Introduction to programming languages”. Ohjelmointikieliä on olemassa useita kymmeniä, mutta ne voidaan pääsääntöisesti jakaa tulkittaviin ja käännettäviin kieliin.

Tulkittavia kieliä suoritettaessa, tarvitaan kielikohtainen tulkki. Tulkin vastuulla on kääntää sovellus ajon aikana prosessorin ymmärtämään muotoon. Käännettävissä kielissä käännös suoritetaan luontivaiheen yhteydessä, minkä jälkeen sovellus on suoraan prosessorin ymmärtämässä muodossa.

Toteutuksen ohjelmointikieltä valittaessa vaihtoehtoina oli Java, C# ja C++, joista jokainen olisi soveltunut pelimoottorin toteutukseen. Näistä tulkittavia kieliä edustaa Java ja käännettäviä kieliä C# ja C++.

Valintaprosessia ohjasi ajatus mahdollisimman monen kohdeympäristön tukemisesta. Mahdollisesti tuettuihin ympäristöihin lukeutuivat pelikonsolit (Xbox One, PlayStation 4) ja tietokoneet (Windows- ja Linux-käyttöjärjestelmillä). Ohjelmointikieleksi valittiin lopulta C++ yksinkertaisen eliminointiprosessin tuloksena. Java ei ole tuettuna kummallakaan pelikonsoleista. C# on heikosti tuettu Linux-käyttöjärjestelmässä ja ei ollenkaan PlayStation 4 -konsolilla. Jäljelle jäi ainoastaan C++, jota tuetaan kaikissa ympäristöissä.

5.2 Ohjelmointiympäristö

Projektissa oli käytössä ”Eclipse IDE” -niminen ohjelmointiympäristö. Eclipse valittiin sen laajennettavuuden ja helppokäyttöisyyden vuoksi. Lisäosien avulla Eclipseen saatiin tuki esimerkiksi varjostinohjelmoinnissa käytetyn GLSL-ohjelmointikielen syntaksiväritykselle ja autotäydennykselle.

Lisäksi Eclipse on saatavilla sekä Windows-, että Linux-käyttöjärjestelmille. Tuki molemmille käyttöjärjestelmille oli suuri etu, sillä kehitystyötä tapahtui molemmilla käyttöjärjestelmillä. Tämän ansiosta siirtymät ympäristöjen välillä toiseen tuntuivat luontevilta.

5.3 Versiohallinta

Versiohallinta on keino hallita työn eri versioita. Yksinkertaisimmillaan tämä tarkoittaa alkuperäisen tiedoston kopioimista ulkoiselle medialle aina tarvittaessa. Tämän kokoluokan projektiin sopi paremmin oikea versionhallintasovellus. Yleisesti käytettyjä versionhallintasovelluksia ovat esimerkiksi SVN, Git ja Mercurial.

Versiohallintasovellukset voidaan jakaa kahteen ryhmään, keskitettyihin ja hajautettuihin. Keskitetyissä versiohallinnoissa koko versioarkisto (repository) on vain etäpalvelimella. Hajautetuissa puolestaan kopio versioarkistosta löytyy jokaiselta asiakkaalta. Edellä mainituista ratkaisuista SVN käyttää keskitettyä versioarkistoa, Git ja Mercurial puolestaan käyttävät hajautettua versioarkistoa. (O’Sullivan B. 2009.)

Opinnäytetyössä työkaluksi valittiin Git. Perusteina Git:n valinnalle oli sen hajautettu toteutus ja helppokäyttöisyys. Lisäksi tarjolla oli useita palveluntarjoajia, joiden hallinnoimia Git-palvelimia sai käyttää maksutta.

5.4 Käännösautomaatio

Sovelluksen käännösprosessin yksinkertaistamiseksi projektille valittiin käännösautomaatiotyökaluksi CMake-niminen ohjelmisto. Vaihtoehtoina olisi ollut käyttää perinteisempää Makefile-käännösympäristöä.

CMaken avulla käännösympäristön konfiguroiminen oli suoraviivaisempaa, kuin muissa käännösautomaatiotyökaluissa. Se tukee monia eri kääntäjiä ja on saatavilla kaikille yleisimmille käyttöjärjestelmille. CMaken avulla projekti oli helppo paloitella loogisiin käännösyksiköihin ja koostaa niistä lopullinen pelimoottori. CMaken tukee myös käännöstä lähdekansion ulkopuolella (out-of-source build).

6 PELIMOOTTORIN TOTEUTUS

6.1 Rajapinnat

Pelimoottorin suunnitteluvaiheessa tehtiin päätös siitä, ettei toteutusta tultaisi sitomaan yksittäiseen grafiikan esitysrajapintaan tai käyttöjärjestelmäriippuvaisiin rutiineihin. Tästä syystä pelimoottoria toteuttaessa pyrittiin abstraktoimaan kaikki laitteisto- ja käyttöjärjestelmäriippuvainen koodi rajapintojen taakse.

Tästä hyvänä esimerkkinä on renderöijä, jonka rajapinta määritellään virtuaalisessa IRenderer -rajapintaluokassa. Pelimoottorilla ei ole mitään käsitystä siitä, mitä grafiikan esitysrajapintaa se käyttää (OpenGL, DirectX, Vulkan tms.), vaan se käyttää ainoastaan rajapinnan tarjoamia kutsuja. Tätä rajapintaa vastaan on luotu konkreettinen toteutus OpenGLRenderer, joka tulkitsee pelimoottorin käskyt OpenGL:n ymmärtämään muotoon.

IRenderer-rajapinnan toteuttava luokka tarjoaa myös keinon luoda esitysrajapintakohtaiset toteutukset *IShader-*, *ITexture-, IFramebuffer- ja IUniform* -rajapinnoista. Nimistä voi päätellä niiden käyttötarkoitukset. Näissä tapauksissa perustelut rajapintojen käytölle ovat samat kuin ylemmän tason IRendererin yhteydessä, eli tarkoitus on piilottaa toteutustekniset yksityiskohdat pois käyttäjältä.

6.2 Pääsilmukka

Pelimoottorin toiminnan kannalta olisi tärkeää, että moottorin toiminta olisi mahdollisimman ennakoitavissa. Ennakoitavuus mahdollistaa huomattavasti helpommat laskennat esimerkiksi törmäystarkistuksissa ja fysiikassa, kun arvoja ei tarvitse välttämättä interpoloida.

Järjestelmä on suunniteltu siten, että se kutsuu kiintein aikavälein peliobjektien FixedUpdate -metodeja, joissa suoritetaan esimerkiksi fysiikan laskentaa tai muuta aikakriittistä toimintaa. Tämä aikaväli on erikseen määriteltävissä, mutta oletusarvoisesti se tapahtuu

60 kertaa sekunnissa. Kiinteän aikavälin päivitysten lisäksi pelimoottori kutsuu muuttuvin aikavälein Update-metodia. Tämä tapahtuu yhtä usein kuin ruutuja piirretään.

6.3 Renderöintimoottori

Pelkkä *IRenderer*-yhteensopiva esitysrajapinta ei vielä riitä pelimaailman piirtämiseen, sillä se vasta mahdollistaa pelimoottorin ja esitysrajapinnan välisen kommunikoinnin. Renderöintimoottorin vastuulle jäävät aktiivisen ruudun valmistelu ja lähetys esitysrajapinnalle. Työ kuulostaa yksinkertaiselta, mutta sitä se ei ole, sillä renderöintimoottori on suurin yksittäinen kokonaisuus koko pelimoottorissa.

Renderöintimoottoria toteuttaessa oli päätettävä mitä renderöintitekniikkaa aiotaan käyttää. Jokaisella tekniikalla on omat heikkoutensa ja vahvuutensa, joten renderöintimoottorin tulevat käyttökohteet vaikuttivat tässä tapauksessa renderöintitekniikan valintaan.

Esimerkkitoteutuksessa oli käytössä aina työn loppusuoralle asti suorarenderöinti (eng. Forward rendering), joka oli helppo ymmärtää ja toteuttaa. Se ei ollut tarpeeksi suorituskykyinen useiden eri valonlähteiden kanssa, joten se korvattiin viivästetyllä renderöinnillä (eng. Deferred rendering).

6.3.1 Suorarenderöinti

Suorarenderöinti on suoraviivainen ja yksinkertainen lähestymistapa geometrian ja valojen esitykselle, jossa jokainen ruudulle piirrettävä objekti käsitellään jokaisen valon osalta erikseen. Se ei vaadi minkäänlaisia erityisominaisuuksia käytettävältä laitteistolta ja on lisäksi helppo toteuttaa. Erityisen hyvin tekniikka soveltuu tilanteisiin, joissa on vähän dynaamisia valonlähteitä. (Unity Manual 2016.)

Suorarenderöinti toimi moitteitta esimerkkisovelluksessa siihen asti, kunnes ruudulla oli useita aktiivisia valoja samanaikaisesti. Vasta tällöin ruudunpäivitysnopeus laski huomattavasti. Tekniikan periaate esitetty kuvassa 1.

```
for (valo : listaValoista) {
    for (objekti : listaObjekteista) {
        objekti.renderöi(valo);
    }
}
```

KUVA 1. Esimerkki suorarenderöinnistä pseudokoodina

6.3.2 Viivästetty renderöinti

Viivästetyssä renderöinnissä useiden valonlähteiden aiheuttamaa ongelmaa yritetään ratkaista siten, että ensimmäisellä renderöintikierroksella (render pass) ruudun geometria piirretään geometriapuskuriin (G-Buffer). Kun geometriapuskuri on luotu, voidaan sitä käyttää seuraavilla renderöintikierroksilla ilman, että geometriaa tarvitsisi piirtää joka kerta uudestaan. (Unity Manual 2016.)

Geometriapuskuri muodostuu ruutupuskurista (eng. Framebuffer), jossa on vähintään kaksi kohdetekstuuria. Puskuriin säilötään vähintään kohteen geometria ja syvyys.

Esimerkkitoteutuksessa geometriapuskurissa kohdetekstuureita on kolme. Data myös pakataan esimerkkitoteutuksessa ennen puskuriin kirjoittamista. Esimerkkitoteutuksen geometriapuskurin sisältö on kuvattu taulukossa 1.

TAULUKKO 1. Geometriapuskurin sisältö

R	G	B	A
Kohdeväri Y	Kohdeväri Co/Cg (Lomitettu)	Heijastusväri Y	Heijastusväri Co/Cg (Lomitettu)
Normaalivektori 1/2 (Pakattu)	Normaalivektori 2/2 (Pakattu)	Kiiltävyys	Heijastuskerroin
Syvyys X	Syvyys Y	Syvyys Z	Vapaa

Viivästetyssä renderöinnissä valoja käsitellessä valojen vaikutus lasketaan ainoastaan pikseleihin, jotka ovat valon vaikutusalueen sisällä. Käytännössä tämä tapahtuu siten, että valon paikalle sijoitetaan kunkin valotyypin muotoinen geometria, minkä jälkeen näytönohjain käyttää varjostimia (shader) vain geometrian peittämän alueen pikseleihin.

Eri valotyypeillä on erilaiset geometriat. Pistevaloilla on yksinkertaisesti pallo, jonka säde on sama kuin valon vaikutusalue (range). Spottivaloilla on kartio, jonka skaalaukseen vaikuttavat spottivalon vaikutusalue ja kulma. Yksinkertaisin kaikista näistä on suuntavalo, jonka vaikutusalueena on aina koko ruutu, ilman poikkeuksia. Valojen käsittely viivästetyssä renderöinnissä on esitetty kuvassa 2.

```
for (objekti : listaObjekteista) {
    geometriaPuskuri.renderöiObjecti(objecti);
}

for (valo : listaValoista) {
    geometriaPuskuri.renderöiValo(valo);
}
```

KUVA 2. Esimerkki viivästetystä renderöinnistä pseudokoodina

6.3.3 Valot

Pelimoottori tarjoaa kolme erilaista valotyyppiä, joita ovat suuntavalo, pistevalo ja spottivalo. Jokainen valaisin käyttäytyy muista poikkeavasti.

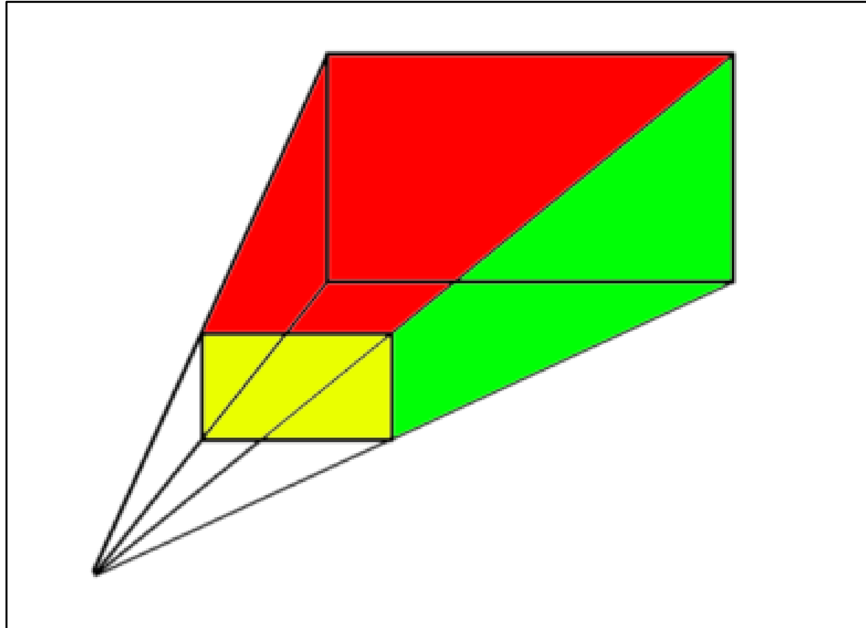
Suuntavalo valaisee kuten aurinko. Valo tulee tietystä kulmasta, eikä siihen vaikuta perspektiivi. Valoteho ei heikkene, vaikka etäisyys muuttuisikin. Pistevalo on kuin hehkulamppu. Se valaisee tasaisesti joka suuntaan, kunnes etäisyyden kasvaessa valo heikkenee ja lopulta häviää kokonaan. Spottivalo on nimensä mukaisesti kuten spotti. Sillä on tiedot valokeilansa kulmasta ja suunnasta, ja se valaisee vain yhteen suuntaan. Kuten pistevalo, myös spottivalo himmenee etäisyyden kasvaessa, kunnes se häviää kokonaan.

6.3.4 Renderöintiketju

Ennen jokaisen ruudun piirtoa renderöintimoottori tarkistaa, mitkä kaikki peliobjektit tulisi piirtää ruudulle ja karsii näin näkökentän ulkopuolella olevat objektit pois. Karsinta perustuu näkökentän (eng. View Frustum) ja kohdeobjektin törmäystarkistukseen. Mikäli

objekti on joko kokonaan tai edes osittain näkökentässä, se piirretään. Mikäli se on kuitenkin näkökentän ulkopuolella, ei sitä käsitellä laisinkaan.

Näkökenttä muodostuu kuudesta eri pinnasta, joita ovat ylä-, ala-, vasen-, oikea-, lähi- ja kaukopinnat. Törmäystarkistuksessa testataan, millä puolella kutakin pintaa kohdeobjekti on, ja mikäli se on kokonaan väärällä puolella yhdestäkään pinnasta, voidaan luotettavasti todeta, ettei se kuulu näkökenttään. Näkökenttä visualisoituna kuvassa 3.



KUVA 3. Näkökentän visualisointi

Renderöinti aloitetaan piirtämällä kaikki kameran näkemät peliobjektit geometriapuskuriin. Geometriapuskuri on pyritty pitämään mahdollisimman kevyenä, joten kaikki pakattavissa oleva piirrettävä data pakataan ennen piirtoa.

Ainoa pakkaamaton data puskurissa on syvyys, jonka avulla myöhemmissä renderöintivaiheissa saadaan uudelleenrakennettua kappaleiden sijainnit 3D-avaruudessa.
Kun geometriapuskuri on käsitelty, lasketaan valojen vaikutus piirrettävään ruutuun. Mikäli käsiteltävä valo on määritelty heittämään varjo, suoritetaan vastaavanlainen peliobjektien karsintaprosessi. Tässä prosessissa ideana on karsia pois kaikki ne peliobjektit, jotka eivät ole valon vaikutusalueella. Kun karsinta on suoritettu, piirretään jäljelle jäävät peliobjektit käyttäen aktiivista valoa virtuaalisena kamerana. Tästä syntyy varjokartta, jota tullaan käyttämään seuraavassa renderöintivaiheessa.

Lopuksi valon vaikutus lasketaan geometriapuskurin tietojen perusteella, mistä lopputuloksena on lopullinen piirrettävä ruutu. Valojen ja varjojen laskenta on yksi raskaimmista työvaiheista, joten vain tärkeimmille valoille tulisi laskea varjot.

6.4 Varjokartat

Varjokartta on referenssitoteutuksessa joko yksittäinen kohdetekstuuri, tai kokoelma kohdetekstuureja, jossa on kuvattuna sen hetkinen näkymä syvyyskarttana. Syvyyskartalle piirtyy lähimmän geometrian etäisyys valonlähteestä.

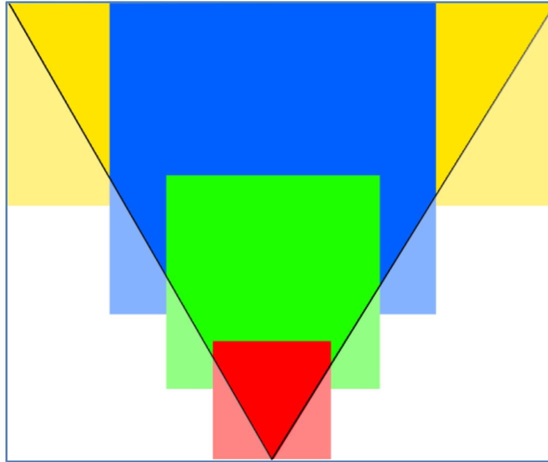
Tätä syvyyskarttaa tulkitsemalla voidaan päätellä, onko valon vaikutusalueella oleva pikseli valaistu, vai varjossa. Mikäli käsiteltävä kappale on kauempana valosta, kuin varjokartan lähin piste, on kappaleen oltava varjossa.

Vaikka sama perusajatus pätee jokaiseen varjokarttatyyppiin, on jokaista valotyyppiä kohden silti oltava oman tyyppisensä varjokartta.

6.5 Kerrostettu varjokartta (*Cascaded Shadow Map*)

Kerrostetut varjokartat ovat käytössä suuntavaloille, joilla ei ole näennäistä sijaintia pelimaailmassa, vaan ne valaisevat tasaisesti riippumatta valon etäisyydestä aina samassa kulmassa.

Kerrostettu varjokartta on nimensä mukaisesti useampikerroksinen ja referenssipelimoottorissa niitä on neljä kappaletta. Kerrokset on aseteltu hieman lomittain siten, että kameraa lähin kerros varjostaa pienimmän alueen ja on laadultaan paras. Viimeinen kerros kattaa laajimman alan, mutta on laadultaan heikoin. Kerrostetun varjokartan periaate nähtävissä kuvassa 4.



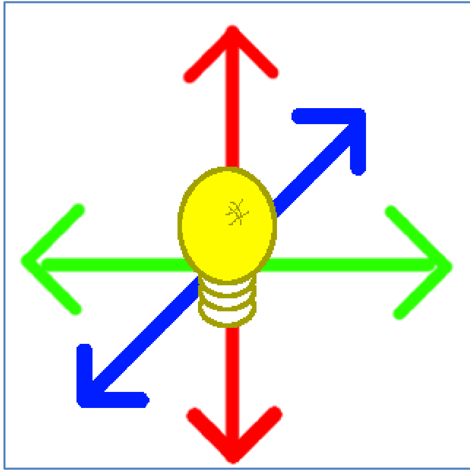
KUVA 4. Varjojen kerrosrakenne, punainen lähin, keltainen kaukaisin

Tämä lähestymistapa mahdollistaa koko näkyvän maailman varjostuksen, ilman että yksittäisen kohdetekstuurin koon tarvitsi olla epäkäytännöllisen suuri. Tässä tekniikassa hyödynnetään sitä faktaa, että kauemmat varjot eivät kuitenkaan näy terävinä, tai ole huomion keskipisteinä, joten niiden laadusta voidaan karsia (Dimitrov, R. 2007, 3).

6.5.1 Suuntaamaton varjokartta (*Omnidirectional Shadow Map*)

Suuntaamaton varjokartta on käytössä ympärisäteileville pistevaloille, jotka valaisevat ympäristöään jokaiseen suuntaan. Koska valo valaisee kaikkiin suuntiin, tulee myös varjokartan kattaa koko valaistu alue.

Suuntaamaton varjokartta tuotetaan asettamalla virtuaalinen kamera valon keskipisteeseen. Tätä kameraa käyttämällä ympäröivä maailma renderöidään kuudesta eri ennalta määritetystä kuvakulmasta. Tästä tuloksena syntyy yksi kuutiotekstuuri (*Cube Texture*), joka kattaa koko näkymän. Kuvakulmien suunnat esitetty kuvassa 5.



KUVA 5. Virtuaalisen kameran kuvakulmien suunnat

6.5.2 Suunnattu varjokartta (*Shadow Map*)

Suunnattu varjokartta on käytössä spottivaloilla, joilla on tietty suunta, johon niiden valokeila osoittaa. Tämä on kaikista varjokartoista yksinkertaisin ja helpoiten ymmärrettävä.

Suunnattu varjokartta sisältää vain yhden näkymän. Näkymä riippuu spottivalon valokeilan kulmasta ja valon heikkenemisestä (attenuation).

6.6 Väriavaruus ja niiden muunnokset

Väriavaruudet RGB, YCbCr ja muunnokset niiden välillä ovat tärkeitä piirrettävien ruutujen esityksessä. Ymmärtääkseen pelimoottorin varjostimien (shader) toimintaa ja niiden suorittamia pakkaus- ja purkuoperaatioita, tulee näistä väriavaruuksista olla vähintään perustason ymmärrys.

6.6.1 RGB

RGB on värien esitysmalli, joka koostuu kolmesta kanavasta. Jokaista kanavaa vastaa yksi väri. Kanavien värit järjestyksessä ovat punainen, vihreä ja sininen. (Eskelinen M. 2002.)

Väritila on helppo visualisoida kuvittelemalla alkuperäisen kuvan päälle värillinen kalvo, joka päästää vain tietyn värin lävitseen. Esimerkkikuva purettu RGB:n käyttämiin kanaviin kuvassa 6.

KUVA 6. RGB purettuna kanaviin (Alkuperäisen kuvan lähde: Wikipedia)

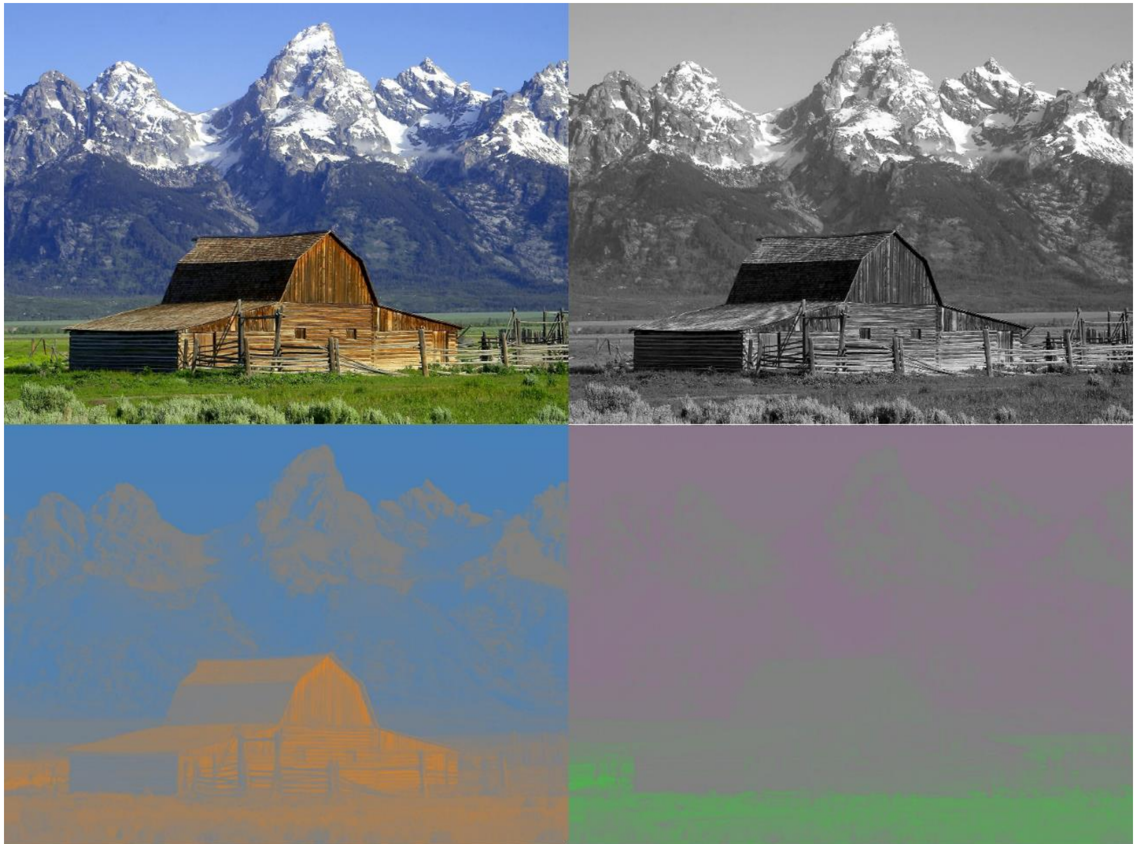
6.6.2 YCoCg

YCoCg edustaa hieman vähemmän tunnettua väritilaa, joka RGB:n lailla hyödyntää kolmea erillistä kanavaa väritiedon esitykseen. YCoCg on kilpailijoitaan paremmin pakkautuvaa. Lisäksi muunnokset YCoCg-väriavaruuteen ja takaisin ovat laskennallisesti keviytä. (Malvar H. & Sullivan G. 2008.).

Tässä väriavaruudessa kanavakohtainen informaatio on seuraavassa muodossa:

1. Valon voimakkuus (*Luminance*)
2. Oranssin värikkyysmuutos (*Orange-difference chrominance*)
3. Vihreän värikkyysmuutos (*Green-difference chrominance*)

Suurin saavutettava etu YCoCg-väriavaruuden käytössä on sen pakattavuus, ja sen tätä ominaisuutta hyödynnetään myös referenssipelimoottorissa. Esimerkkikuva purettu YCoCg:n käyttämiin kanaviin kuvassa 7.



KUVA 7. YCoCg purettuna kanaviin (Alkuperäisen kuvan lähde: Wikipedia)

6.6.3 Muunnokset

Muunnoksella tarkoitetaan siirtymää yhdestä väriavaruudesta toiseen. Muunnos voi olla joko häviöllinen tai häviötön. Häviöllisessä muunnoksessa osa alkuperäisestä informaatiosta menetetään, kun taas häviöttömässä kaikki tieto on yhä saatavilla.

RGB ja YCoCg väriavaruuksien väliset muunnokset kuuluvat häviöttömään kategoriaan ja ne voidaan suorittaa muutamalla laskutoimituksella. Esimerkit kuvissa 8 ja 9. (Malvar H. & Sullivan G. 2008.).

```
Co  = R – B;
tmp = B + Co/2;
Cg  = G - tmp;
Y   = tmp + Cg/2;
```

KUVA 8. Muunnos RGB väriavaruudesta YCoCg väriavaruuteen pseudokoodina

```
tmp = Y – Cg/2;
G   = Cg + tmp;
B   = tmp - Co/2;
R   = B + Co;
```

KUVA 9. Muunnos YCoCg väriavaruudesta RGB väriavaruuteen pseudokoodina

6.7 Geometriapuskurin optimointi

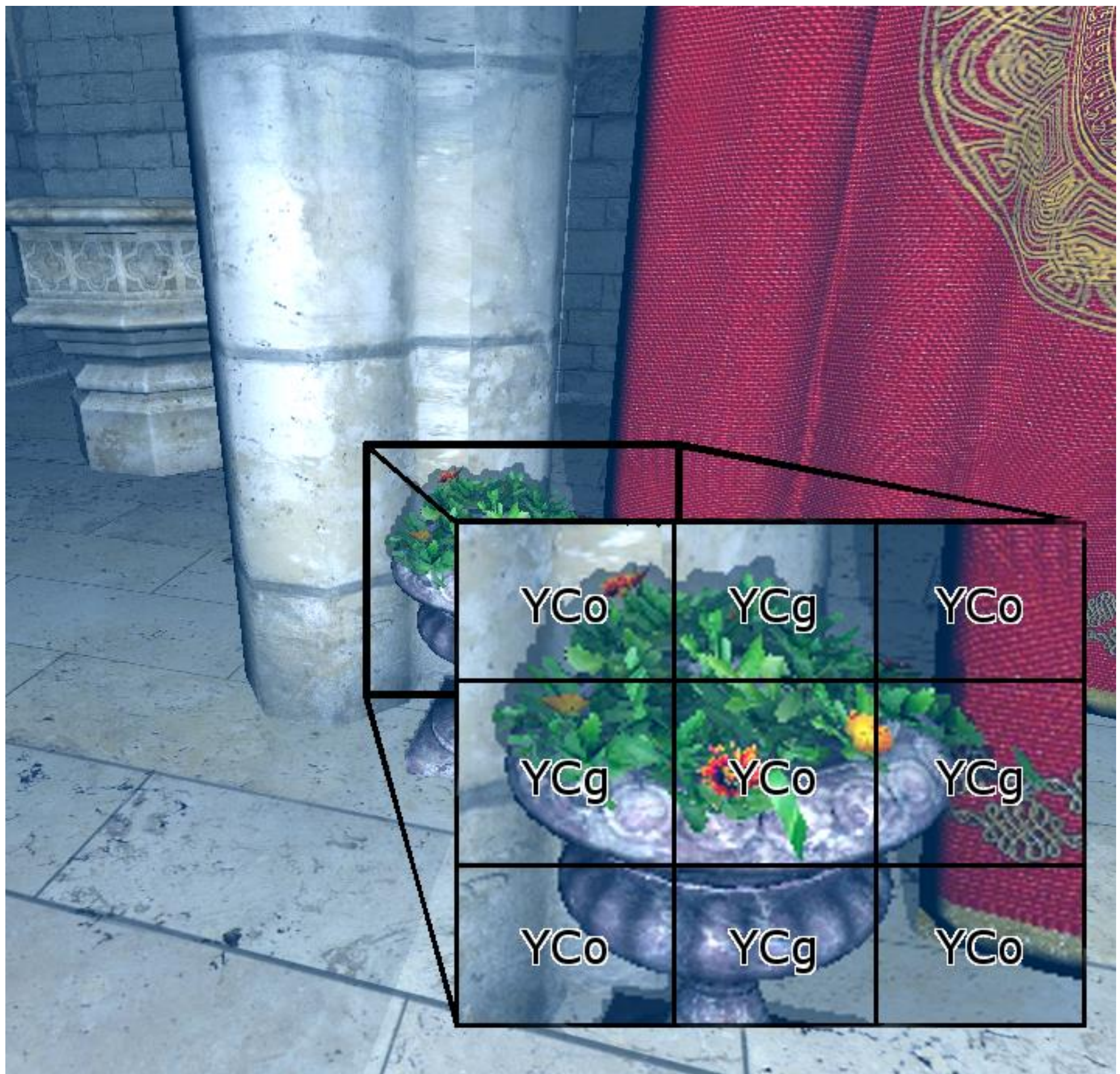
Geometriapuskurin yhteydessä käytetään useita renderöintikohteita. Kohteiden kirjoitus on helppoa ja nopeaa, mutta ongelmat kasaantuvat, kun puskurin tekstuureita näytteistetään valaistuksen yhteydessä useita kertoa. Koska viivästetty renderöinti on erityisen raskasta näytönohjaimen muistikaistalle, on kaikesta muistikaistan käyttöä alentavista toimenpiteistä huomattava hyöty.

Geometriapuskuriin kirjoitetaan kaksi eri väriarvoa per pikseli, hajontaväri (diffuse color) ja heijastusväri (specular color). Jokainen talletettava väri tarvitsee kolme tavua näytönohjainmuistia ja koska värejä tarvitaan kaksi, on tarve jo kuudelle tavulle per pikseli. Tämä kuulostaa vähältä, mutta suurilla resoluutioilla puskurien koko kasvaa huomattavasti. Täysteräväpiirtoresoluutiolla (1920x1080) ajettaessa pelkästään värit tarvitsevat lähes 12 megatavua näytönohjainmuistia.

Referenssitoteutuksessa väritieto lomitetaan puskuriin. Lomituksen jälkeen materiaali vastaa monissa videokäsittelysovelluksissa käytettyä 4:2:2 -koodekkia ja värien pakkaussuhteeksi muodostuu 3:2. Lomituksen jälkeen puskuri tarvitsee enää n. kahdeksan megatavua muistia. Puskurin voi kuvitella shakkiruutuina, kuten kuvassa 10.

RGB-väriarvojen lomittaminen ei ole käytännöllistä, joten tässä vaiheessa muunnos RGB-väriavaruudesta YCoCg-väriavaruuteen tulee tarpeeseen. Vaikka itse värimuunnos on häviötön, tulee siitä lomituksen jälkeen häviöllinen. Lomitus YCoCg:llä on huomattavasti parempilaatuista, sillä ihmissilmä on herkempi havaitsemaan valon voimakkuuden muutosta, kuin valon värimuutosta.

Lomitus ei kuitenkaan ole täydellinen ratkaisu, vaan siinä tehdään kompromissi muistin käytön, muistin kaistanleveyden ja laskentanopeuden kesken. Lomitettua tietoa lukiessa, tulee puuttuvat tiedot saada tavalla tai toisella takaisin. Toteutuksessa puuttuvat arvot luetaan takaisin käyttämällä reunasuuntaista suodatusta. Käytännössä pikseliä näytteistettäessä, näytteistetään myös sen vasemmalla, oikealla, yläpuolella ja alapuolella olevat pikselit. Näistä lasketaan painotettu keskiarvo, joka korvaa puuttuvan arvon (Mavridis, P. & Papaioannou G. 2012, 23).



KUVA 10. Värien lomituksen visualisointi

6.8 Tekstin renderöinti

Pelimoottorin pitää pystyä myös tuottamaan tekstiä ruudulle. Ilman tekstejä on mahdollista luoda pelejä, mutta käytännössä jokaisesta pelistä löytyy tekstiä, ainakin valikoista. Referenssipelimoottorissa tekstin renderöintiin käytetään fonttikarttaa, joka luodaan käynnistyksen yhteydessä. Yksi fonttikartta sisältää yhden fontin merkit. Useampia fontteja käytettäessä tulee luoda useampi fonttikartta. Jokainen fonttikartta ladataan tekstuurina näytönohjaimelle ja tätä karttaa käytetään yksittäisten merkkien piirtoon.

Luotava fonttikartta ei ole puhtaassa bittikarttaformaatissa, vaan sen esitysmuotona on etumerkillinen etäisyyskartta (Signed distance field). Etäisyyskartassa kuva näyttää sumealta, koska jokainen pikseli kuvassa kertoo etäisyyden merkin reunasta. Etäisyyskartan merkkejä ei piirretä sellaisenaan. Varjostinsovellus näytteistää kartan pikselit ja tekee päätöksen siitä, onko pikseli piirrettävän merkin sisä- vai ulkopuolella. (Green, C. 2007.)

Merkkijonojen renderöinti tapahtuu pilkkomalla ne yksittäisiin merkkeihin, minkä jälkeen ne käsitellään merkki kerrallaan. Ruudulla jokaista merkkiä vastaa yksi renderöitävä teksturoitu neliö. Esimerkki etumerkillistä etäisyyskarttaa käyttävästä fonttikartasta on kuvassa 11.

KUVA 11. Esimerkki fonttikartasta

POHDINTA

Opinnäytetyön tuloksena syntyi sovellus, jota tilaaja pystyy hyödyntämään heti sellaisenaan. Sen lisäksi asiakkaalle jäävät kopiot sovelluksen ja näin ollen myös pelimoottorin lähdekoodeista. Lähdekoodeja tutkimalla ja muokkaamalla tilaaja pystyy jatkokehittämään sovellusta muuttuvien tarpeidensa mukaisesti. Pelkkien muokkausten lisäksi pelimoottorin lähdekoodeja käyttämällä on mahdollista luoda myös uusia sovelluksia tai jopa pelejä.

Pelimoottori itsessään oli erittäin laaja. Opinnäytetyön työmäärä ylitti kaikki asettamani arviot ja jälkikäteen ajatellen, sitä olisi pitänyt rajata huomattavasti tarkemmin. Suuresta työmäärästä huolimatta, koin että opinnäytetyö onnistui yli odotusten ja se kasvatti omaa osaamistani merkittävästi.

En kuitenkaan suosittelisi vastaavaa lähestymistapaa pienille peliprojekteille, sillä aikaa itse pelille ja pelimekaniikalle ei mielestäni jäisi tarpeeksi.

LÄHTEET

Gregory, J. 2009. Game Engine Architecture. Natick: A K Peters.

Unity Learn. 2017. Unity Technologies. Luettu 15.9.2016.
https://unity3d.com/learn/tutorials/

Unity Manual. 2016. Unity Technologies. Luettu 21.9.2016.
https://docs.unity3d.com/Manual/index.html

Dimitrov, R. 2007. Cascaded Shadow Maps. NVIDIA Corporation. Luettu 15.4.2017
http://developer.download.nvidia.com/SDK/10/opengl/src/cascaded_shadow_maps/doc/cascaded_shadow_maps.pdf

Green, C. 2007. Improved Alpha-Tested Magnification for Vector Textures and Special Effects. Valve Corporation. Luettu 24.9.2016.
http://www.valvesoftware.com/publications/2007/SIGGRAPH2007_AlphaTestedMagnification.pdf

Mavridis, P. & Papaioannou G. The Compact YCoCg Frame Buffer. 2012. Journal of Computer Graphics Techniques. Luettu 17.2.1017.
http://jcgt.org/published/0001/01/02/paper.pdf

OpenGL Wiki. 2015. Khronos Group. Luettu 8.4.2017.
https://www.khronos.org/opengl/wiki/

Malvar H. & Sullivan G. YCoCg-R: A Color Space with RGB Reversibility and Low Dynamic Range. 2008. Microsoft Corporation. Luettu 19.2.2017
https://www.microsoft.com/en-us/research/wp-content/uploads/2016/06/Malvar_Sullivan_YCoCg-R_JVT-I014r3-2.pdf

Eskelinen M. Värien teoria ja värimallit. 2002. Luettu 25.5.2017
http://users.jyu.fi/~tro/gtksem02/prujut/matti/varimallit.pdf

Anthony A. Introduction to Programming Languages. 2004. Luettu 25.5.2017
http://www.emu.edu.tr/aelci/Courses/D-318/D-318-Files/plbook/

O'Sullivan B. Mercurial: The Definitive Guide. 2009. Luettu 25.5.2017
http://hgbook.red-bean.com/read/

LIITTEET

Liite 1. Lähdekoodit.

Projektin lähdekoodit ovat saatavilla BitBucket -palvelusta, osoitteesta:
https://bitbucket.org/JokinenK/projectcactus