Juha Heiskanen

DirectX-pelimoottori

Metropolia Ammattikorkeakoulu

Insinööri AMK

Tietotekniikka

Opinnäytetyö

01.05.2014

Metropolia

Tekijä(t) Otsikko Sivumäärä Aika	Juha Heiskanen DirectX-pelimoottori 40 sivua + 2 liitettä 01.05.2014
Tutkinto	Insinööri (AMK)
Koulutusohjelma	Tietotekniikka
Suuntautumisvaihtoehto	Tietoverkot
Ohjaaja(t)	Osaamisaluepäällikkö Janne Salonen

Opinnäytetyön tarkoituksena oli opiskella ja tutkia DirectX 11 -ohjelmistorajapintoja ja tavoitteena oli ohjelmoida toimiva pelimoottorin runko Windows-ympäristöön. Työssä tutustuttiin Windows-ohjelmistokehitysympäristöön ja DirectX:n eri osa-alueisiin.

Microsoftin kehittämä DirectX on kokoelma etenkin pelintekoon suunnattuja ohjelmointirajapintoja. DirectX tarjoaa yhtenäisen kehitysympäristön mm. grafiikanluontiin, äänien hallintaan ja ohjauslaitteita varten. DirectX esiintyi ensimmäisen kerran vuonna 1995, kun Windows 95 -käyttöjärjestelmä julkaistiin.

DirectX:n ohjelmointikieli on C++, ja tätä käytetään yhdessä Windows SDK:n kanssa. DirectX:n omat ohjelmointirajapinnat sallivat suoran pääsyn tietokoneeseen asennettuun laitteistoon ja näin ohjelmakoodista saadaan hyvin tehokasta. Tehokas grafiikan luonti tapahtuu HLSL-ohjelmointikielellä, joka on kehitetty näytönohjaimien grafiikkapiirien ohjelmointiin.

Työmäärä osoittautui oletettua suuremmaksi, ja vain osa ohjelman sisällöstä päätyi opinnäytetyöhön. DirectX on äärimmäisen tehokas, mutta jo pienenkin asian toteuttaminen vaatii paljon ohjelmakoodia. Samoin virheiden havaitseminen ja korjaaminen ilman erillisiä apukeinoja voi olla vaikeaa. Lopputuloksena syntyi kuitenkin toimiva ohjelmarunko, jota voi vapaasti laajentaa erilaisiin peliympäristöihin.

Avainsanat	pelimoottori, visual c++, directx, hlsl, windows sdk, microsoft, quad tree, korkeuskartta, taivas, vesi

Author(s) Title Number of Pages Date	Juha Heiskanen DirectX Game Engine 40 pages + 2 appendices 1 May 2014
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Data Networks
Instructor(s)	Janne Salonen, Head of Department

The objective of this thesis was to study and to get familiar with the DirectX 11 Application Programming Interfaces and the aim was to program a properly functioning game engine body for the Windows environment. This study examined the Windows software development environment and different parts of DirectX.

DirectX was developed by Microsoft and it is a collection of different types of Application Programming Interfaces especially aimed for game programming. DirectX provides a unified development environment for graphics, sound and input hardware. DirectX appeared the first time in 1995 when the Windows 95 operating system was released.

The main programming language for DirectX is C++ and this is used together with the Windows SDK. The programming interfaces of DirectX allow direct access to the installed hardware and this makes all written code very effective. The High-Performance graphics are created by using HLSL programming language which is developed for the GPUs of graphics cards.

The amount of work turned out to be more than had been expected and only a part of the program ended up in the thesis. DirectX is extremely powerful but even a small function requires a lot of code. Also the identifying and correcting of programming errors without additional tools can be very challenging. However, the result was a functional program body which can be freely expanded for a variety of game environments.

Keywords	game engine, visual c++, directx, hlsl, windows sdk, Microsoft, quad tree, height map terrain, sky dome, small body water

Sisällys

Lyhenteitä ja käsitteitä

AA *Anti-Aliasing*. Reunanpehmennystekniikka, jolla pyritään vähentämään graafiikan sahalaitaisuutta.

API *Application Programming Interface*. Ohjelmointirajapinta, jonka mukaan eri ohjelmat voivat keskustella keskenään.

BMP Microsoftin kehittämä häviötön kuvaformaatti.

ClearType ®
Microsoftin rekisteröimä tuotemerkki, jonka tarkoituksena on parantaa tekstin luettavuutta tietokonenäytöllä.

DDA *Data-Driven Architecture*. Tieto-ohjattu arkkitehtuuri, jossa ohjelman käyttäytymistä muutetaan sisältöä lisäämällä ja sitä poistamalla koskematta itse ohjelmarunkoon.

DDS *DirectDraw Surface*. Microsoftin kehittämä DirectX:ää varten luotu kuvaformaatti.

DLL *Dynamic-Link Library*. Dynaaminen linkkikirjasto, joka sisältää ajettavaa ohjelmakoodia.

DSP *Digital Signal Processing.* Digitaalinen signaalinkäsittely.

Force Feedback
Peliohjaimissa käytettävä tekniikka, jolla tarkkaillaan laitteen asentoa ja laitteeseen kohdistettua fyysistä voimaa.

GPGPU *General-Purpose Computing on Graphics Processing Units*. Tekniikka, joka on suunniteltu tekemään näytönohjainlaskentaa.

Instantiointi

Optimointitekniikka, jolla monistetaan jo luotuja objekteja ja näin vähennetään resurssienkäyttöä.

JPEG *Joint Photographic Experts Group*. Häviöllistä pakkausta käyttävä bittikarttagrafiikan tallennusformaatti. Tunnetaan myös nimellä JPG.

LOD *Level of Detail*. Optimointitekniikka, jolla rajoitetaan 3D-grafiikan piirtämistä objektien etäisyyden perusteella.

Mappaus Kuvien sijoittelua kohteen koordinaattien ja pinnan muotojen mukaisesti.

OBJ Wavefront Technologies -yrityksen animointia varten kehittelemä 3D-mallien tietoja sisältävä formaatti.

OpenGL *Open Graphics Library*. Silicon Graphicsin kehittämä 2D- ja 3D-grafiikan tekoon suunniteltu ohjelmointirajapinta.

OpenType ®

Microsoftin kehittämä ja rekisteröimä laitteistoriippumaton fonttiformaatti.

PNG *Portable Network Graphics*. Avoin, häviötön ja pakkaamista tukeva bittikarttaformaatti.

SDK *Software Development Kit*. Kokoelma erilaisia ohjelmistorajapintoja.

Shader Model

HLSL-ohjelmistokielen versio.

Sprite Kaksiulotteinen, yleensä monta eri kuvaa sisältävä bittikarttakuva, josta vain haluttua osaa näytetään aina kerrallaan.

STL *Standard Template Library*. Ohjelmoinnissa käyetettävä useista yleisistä luokista koostuva standardikirjasto.

Streamlining
Tietokonepelaajien kesken käytetty termi, joka tarkoittaa jonkin asian yksinkertaistamista tai pelkistämistä.

Swash-typografia
Fonttien koristeluun suunniteltu tekniikka, jolla lisätään jo ennalta luotuun fonttiin lisää grafiikkaa.

Tekstuuri
Vektorigrafiikalla toteutetun pinnan päälle tarkoitettu päällyste,

Tile Saumaton bittikarttakuva, joita voidaan yhdistellä yhtenäiseksi kokonaisuudeksi.

Texture Blending
Tekniikka, jolla sekoittamalla useita erilaisia tekstuureja muodostetaan ohjelmallisesti yksi tekstuuri.

UniCode Tietokonejärjestelmiä varten kehitetty merkistöstandardi.

WARP *Windows Advanced Rasterization Platform*. Windows-ympäristössä käytettävä ohjelmistorasteroija, jolla suoritetaan näytönohjaimelle tarkoitettua ohjelmakoodia keskussuorittimella.

Wrapperi Eng. *wrapper* on kääreluokka, jolla käännetään ohjelmistorajapinnan käyttämää ohjelmointikieltä toiselle rajapinnalle.

1 Johdanto

Mobiililaitteille ohjelmoitujen pelien määrä kasvaa päivä päivältä, ja tietokonepelaaminen on vähentynyt tuntuvasti viimeisen kymmenen vuoden aikana. Syynä tähän on pelikonsolien ja etenkin mobiililaitteiden lisääntyminen kuluttajien keskuudessa. Pelikonsoleiden ja mobiililaitteiden tekninen rakenne ovat jo nyt hyvin lähellä toisiaan, ja näin vastavalmistuneiden ohjelmoijien on helppo omaksua useampaa alustaa palveleva ohjelmointiympäristö. Tästä ilmiöstä tietenkin kärsii eniten tietokoneilla pelaavat kuluttajat, joille puhtaiden PC-pelien tarjonta käy aina vain suppeammaksi.

Nykyään on enää vähän sellaisia pelitaloja, jotka tuottavat pelimateriaalia ainoastaan tietokoneille. Suuret pelitalot ovat nähneet pelituotannon kannattavammaksi silloin, kun luodaan täysin universaali pelimoottori palvelemaan erilaisia alustoja. Tällöin ajankohtaiseksi tulevat erilaiset wrapperit, joilla parannetaan laitteistojen ja ohjelmakoodin yhteensopivuutta. Vapaasti suomennettuna wrapperi on ohjelmallisesti tehty tulkki, joka kääntää esimerkiksi OpenGL:llä toteutetun koodin sellaiseen muotoon, että DirectX sen ymmärtää. Valitettavasti näiden wrapperien käyttö tekee ohjelmakoodista kömpelöä, raskasta ja etenkin virhealtista. PC-pelaajat, jotka ovat sijoittaneet suuren määrän rahaa monen näytönohjaimen kokoonpanoihin, suureen määrään keskusmuistia ja moniydinprosessoreihin, kärsivät alituisesti suorituskykyongelmista uusien pelijulkaisuiden kautta. Laitevalmistajat painivat saman asian kanssa, koska kuluttajat vaativat tietokoneiltaan suorituskykyä. Tilannetta yritetään korjata ajuripäivityksien voimin. Näytönohjainvalmistajat ovat alkaneet tuottaa erilaisia profiileja tunnetuimpia pelejä varten, mutta monesti käy niin, että toista peliä varten luotu profiili ei suostukaan toimimaan toisessa, vaikka pelien julkaisuajankohdat olisivat hyvinkin lähellä toisiaan.

Lisäksi pelimaailmaa hämmentää etenkin PC-pelaajia koskettava ilmiö, joka on saanut nimekseen pelkistäminen. Peliharrastajien kesken kyseistä, jopa vitsaukseksikin kutsuttua ilmiötä, kutsutaan termillä *streamlining*. Koska pelikonsolien ja etenkin mobiililaitteiden suorituskyky sekä tallennuskapasiteetti ovat hyvin rajallista, pelejä on pyritty pitämään tarkoituksena pienenä ja yksinkertaisina. Tästä ilmiöstä eniten kärsii pelimaailman koko ja keinoäly, mutta tietenkin myös grafiikka. Myös roolipelien hahmonluontia on yksinkertaistettu, tai hahmonluonti on jätetty kokonaan pois. Näiden muokkauksien saattelemana suuri osa peleistä alkaa muistuttaa liian paljon toisiaan, ja pelaajien tyytymättömyys näkyy hyvin selkeästi pelikaupoissa ja pelitalojen verkkokeskustelusivuilla.

Halusin henkilökohtaisesti ottaa kantaa tietokonepelaajia vaivaavaan ongelmaan ja tunsin palavaa halua luoda täysin optimoitu pelimoottori Windows-ympäristöön. Kuitenkin ajankäyttöni tähän projektiin on rajallinen, joten keskityn pääasiallisesti 3D-grafiikan luontiin ja erilaisiin tehosteisiin. Useista vaihtoehdoista päädyin käyttämään DirectX 11 -rajapintaa, jota tukee valtaosa kuluttajilla olevista käyttöjärjestelmistä. Olen tehnyt aikaisemmin ohjelmia DirectX:n versioilla 7 ja 9, mutta ennen tätä opinnäytetyötäni en ole juurikaan ollut tekemisissä DirectX 11 -ohjelmoinnin kanssa. Samalla kun tutustun paremmin DirectX 11 -ohjelmistorajapintaan, niin haluan myös kertoa tästä aiheesta suomeksi. DirectX on tuttu termi pelaajien keskuudessa, mutta harvalla todellisuudessa on paljoakaan käsitystä siitä, mitä kaikkea se käytännössä sisältää. Suomessa tämä on suuri ongelma, koska itse materiaalia DirectX:stä on paikallisella kielellä hyvin niukasti saatavilla. Tulen käyttämään ohjelmoinnissani apuna Microsoftin virallisia dokumentteja ja esimerkkejä, koska haluan kirjoittamani koodin olevan mahdollisimman puhdasta, selkeää ja ammattitaitoista. Tulen myös testaamaan koodiani useilla erilaisilla laitteistoilla ja käyttöjärjestelmillä.

Pelimoottorini on tarkoitettu pääasiassa reaaliaikaisia strategiapelejä varten, mutta varmasti muokattuna se toimii hyvin myös toimintapelin pohjana. Pelimoottorista löytyvät useimmat vaaditut peruskomponentit, joihin sisältyvät mm. taivas, liikkuvat pilvet, vesi, Quad Tree -osioinnilla tehty korkeuskartta, itse tuotettujen 3D-mallien käyttö, instantiointi, erilaisia teksturointimenetelmiä, HLSL ja sekä näppäimistöä että hiirtä tukeva input-luokka. Tämän lisäksi teen pelimoottoriin yksinkertaisen Game State Management -rakenteen, jonka pohjalta tulen rakentamaan myös jonkinlaisen alkuvalikon.

Tämä aihe on valtavan laaja, mutta pyrin rajaamaan sitä tarpeen mukaan. Toivon suuresti, että tästä opinnäytetyöstäni tulee olemaan hyötyä sellaiselle ihmiselle, jolla on ollut vaikeuksia opetella DirectX-ohjelmointia vieraskielisen dokumentoinnin vuoksi. Eniten mielihyvää tuottaisi se, jos tämä työni alentaisi lukijan kynnystä aloittaa oma ohjelmointiura DirectX:n parissa.

2 Pelimoottoreista

Termi *pelimoottori* nousi ensimmäisen kerran esiin vuonna 1990, kun ID Software kehitti suuren suosion saaneen Doom-nimisen tietokonepelin. Doom oli rakenteeltaan ensimmäinen julkaistu peli, jossa lukuisat ohjelmistokomponentit olivat erillään ohjelman ydinrungosta. Tämä salli sen, että samaa ohjelmarunkoa pystyttiin pienillä muutoksilla käyttämään yhä uudelleen erilaisissa toteutuksissa. Tämä keksintö helpotti huomattavasti pelikehittäjien työtä, ja näin pelitalot alkoivatkin suojaamaan omia ohjelmarunkojaan lisensseillä. Uutena ilmiönä syntyivät myös modaaja-yhteisöt (*mod community*), jotka muokkasivat valmiiksi tehtyjä pelejä luomalla itse sisältöjä. Tämä johti siihen, että pelituottajat alkoivat lisäämään peleihinsä erilaisia työkalupakkeja ja skriptauskieliä, jotka mahdollistivat tehokkaan toimintaympäristön pelin muokkaamista varten. [1, s. 13.]

Pelimoottori on laaja käsite, ja raja pelimoottorin ja itse pelin välillä on hyvin häilyvä. Joissakin tapauksissa pelimoottorissa on selkeästi eroteltu pelimoottori itse pelistä, kun taas toisissa peleissä ei edes ole yritetty erottaa näitä toisistaan. Luultavimmin tieto-ohjattavuus (*data-driven architecture*) on se suurin ero, mikä erottaa sen ohjelmasta, joka on peli, mutta ei pelimoottori. Meidän tulisi käyttää termiä pelimoottori silloin, kun itse ohjelma on laajennettavissa ja sitä voidaan käyttää uuden pelin runkona ilman suuria muutoksia. [1, s. 13.]

3 Käytetty ohjelmointikieli, Visual C++

Visual C++, joka tunnetaan monesti nimellä MSVC tai VC++, on ohjelmointiympäristö, joka käsittää kaksi erillään, jokseenkin hyvin toisiaan lähellä olevaa ohjelmointikieltä. Ensimmäinen näistä on ISO/IEC standardoitu C++, joka tunnetaan paremmin nimellä natiivi C++. Toinen kielistä on C++/CLI, joka on kehitetty yksinomaan Windows-ympäristöön.

3.1 ISO/IEC standardin C++

Natiivi C++ on yleisesti ottaen suosituin ohjelmointikieli sovelluksia kehittävien ammattilaisten keskuudessa. Etenkin C++ tulee esiin silloin, kun pääasiallisena kriteerinä on ohjelman suorituskyky ja kehitettävyys. C++ sisältää C-kirjaston hieman muokattuna ja al-

goritmeja sisältävän Standard Template Libraryn (STL). Tämä käsittää luokat, periytymisen, mallit (templates) ja poikkeukset. C++:n kehitti Bjarne Stroustrup 1980-luvulla. [2, s. 63 - 108.]

C++-kielellä on ollut tähän päivään mennessä neljä erilaista standardia. Standardi ISO/IEC 14882:1998 vahvistettiin vuonna 1998. Standardi ISO/IEC 14882:2003 vahvistettiin vuonna 2003. ISO/IEC TR 19768:2007 -standardi oli jo ennemminkin laajennus, joka lisäsi C++:aan muun muassa referenssipointterit ja laajennetun matematiikkakirjaston. Tämä vahvistettiin vuonna 2007. Uusin standardi on ISO/IEC 14882:2011, joka tunnetaan paremmin nimellä C++11, vahvistettiin vuonna 2011. [3.]

Matalamman tason ohjelmointikielenä C++ on laiteläheinen kieli, eikä se siksi sisällä esimerkiksi automaattista roskienkeräystä. Muistinhallinta suoritetaan manuaalisesti ohjelmankirjoittajan toimesta, ja erilaisten viitteiden ja osoitinmuuttujien käyttö on kielelle ominaista.

ISO/IEC C++ -ohjelmointikieli ei ole alustarajoitteinen, joten tätä käytetään yleisesti niin teollisuudessa, mobiililaitteissa, tietokoneissa kuin pelikonsoleissakin. C++:lla on kaikkein laajimmin käytetty ohjelmointikieli koko maailmassa. Sillä on kirjoitettu mm. käyttöjärjestelmiä, pelejä, laiteohjaimia ja tieteen käyttöön suunnattuja ohjelmia. [4.]

3.2 C++/CLI-standardi

C++/CLI (*Common Language Infrastructure*) on Microsoftin kehittämä laajennus natiivi-C++:n päälle, mikä mahdollistaa sovellusten kehittämisen C++:lla .NET-ympäristöön. Tämä laajennuksen kohteena ovat pääasiallisesti virtuaalikoneympäristöt, jotka tukevat .NET Framework -ohjelmistokomponenttikirjastoa. Näin C++ saadaan yhdistettyä muiden NET Frameworkia tukevien ohjelmointikielien, kuten BASICin ja C#:n kanssa. Yksinkertaistettuna C++/CLI on siis osa ECMA-standardia, joka määrittää kokonaisuudessaan virtuaalikoneympäristön .NET:iä varten. [5, s. 2.]

4 Windows API (Application Programming Interface)

Windows API, josta käytetään yleisesti nimikettä WinAPI, on ohjelmointirajapinta, jota käytetään yksinomaan Windows-sovellusten kehittämiseen. WinAPI tarjoaa resurssit tiedostojärjestelmien, laitteiden, prosessien, säikeiden ja virheiden käsittelyyn. Tässä opinnäytetyössäni käytän Windows 7.0A SDK:ta (Software Development Kit), joka on julkaistu Windows 7 -käyttöjärjestelmän yhteydessä.

4.1 GDI

WinAPI sisältää GDI:n (Graphics Device Interface), joka on yksi käyttöjärjestelmän ydinkomponenteista. Se on vastuussa graafisten objektien esittämisestä sekä niiden välittämisestä ulostulosignaaleille, kuten tulostimille ja näytöille. GDI:n tehtäviin kuuluvat esimerkiksi suorien ja kurvien piirtäminen, kirjasimien renderöinti ja palettien hallinta. [5, s. 848.]

Käytännössä pelkän GDI:n avulla on mahdollista toteuttaa jonkin pienimuotoinen pelin grafiikka kokonaisuudessaan, mutta tehokkaaseen grafiikanmallinnukseen siitä ei ole. GDI on myös äärimmäisen haastava käytettävä, jos sillä halutaan tehdä kehittyneempää animointia.

4.2 GDI+

Kun Windows XP julkaistiin, niin sen yhteydessä GDI:tä laajennettiin GDI+-lisäkirjastolla. GDI+ lisäsi jo olemassa olevan GDI:n päälle reunanpehmennystekniikan (Anti-Aliasing), liukulukukoordinaatit, gradienttivarjostuksen (Gradien Shading), monimutkaisemman polun hallinnan ja tuen affiiniselle kuvaukselle. GDI+ tuki myös uudempia kuvaformaatteja, kuten JPEG ja PNG aikaisemman BMP:n sijaan. [6.]

5 Microsoft DirectX

DirectX on kokoelma erilaisia multimedia-, ja etenkin peliohjelmointia varten kehiteltyjä kirjastoja. Nämä kirjastot tarjoavat valmiita työkaluja mm. grafiikanluontiin, oheislaitteiden hallintaan ja äänentoistamiseen. DirectX julkaistiin ensimmäisen kerran vuoden 1994 loppupuolella Windows 95 -käyttöjärjestelmän yhteydessä. Tämän jälkeen DirectX on ollut pääasiallinen työkalu erilaisten multimediaominaisuuksia sisältävien sovellusten kehittämiseen Windows-ympäristöön. [7]

Pelimoottorissani käyttämäni DirectX 11 on kesäkuussa 2010 Windows 7:n yhteydessä julkaistu multimediarajapinta, joka toi suuria uudistuksia yleisemmin käytettyyn DirectX 9:ään. Uudistuksiin kuuluu mm. tesselaatio ja saumaton LOD (Level of Detail), jotka ovat eräänlaisia optimointitekniikoita 3D-grafiikan toteuttamisessa.

DirectX 11 tarjoaa tehokkaan kehitysympäristön etenkin 3D-grafiikan luomiseen ja on osoittautunut hyvinkin mielekkääksi käyttää. Ainoana haittapuolena DirectX 11:ssä näen sen, että natiivi tuki 3D-mallien X-formaatille on poistettu kokonaan, joten jos omia 3D-malleja haluaa ohjelmakoodiinsa tuoda, niin on käytettävä joko kolmannen osapuolen luomia muuntimia tai luotava 3D-malleille täysin oma formaatti. Tämä on mielestäni suurin syy, miksi valtaosa Windows-pelikehittäjistä pysyvät vielä edelleenkin DirectX 9.0c:ssä.

5.1 Direct3D (D3D)

Direct3D on osa DirectX API:a, joka toimii pohjana vektorigrafiikkan luonnissa. Vaikka nimi viittaakin 3-ulotteisuuteen, niin sillä voidaan ihan yhtälailla toistaa myös 2D-grafiikkaa. Direct3D API:a käytetään sovelluksissa, joissa graafinen suorituskyky on tärkeää. Näin se soveltuukin erinomaisesti peliohjelmiin. DirectDraw:n lailla se käyttää hyödykseen erillistä grafiikkakiihdytinkorttia, jos sellainen laitteistossa on. Direct3D sallii grafiikkakiihdytyksen sekä täyden kiihdytyksen 3D-renderöintiliukuhihnassa, ja myös niin, että vain osa grafiikasta on kiihdytetty. Direct3D sisältää ominaisuuksia, kuten z-bufferoinnin (Z-Buffering), avaruudellisen reunapehmennyksen (Spatial Anti-Aliasing), alfa-blendauksen (Alpha Blending), mipmappauksen (Mipmapping), ilmastollisia tehosteita (Atmospheric Effects) ja perspektiiviä korjaavan tekstuurimappauksen (Perspective-Correct Texture Mapping). [8, Direct3D 11 Graphics.]

Direct3D on ollut osana DirectX:ää ensimmäisestä versiosta alkaen, ja sitä on päivitetty aina eteenpäin uusien versioiden yhteydessä. Vaikka Direct3D olikin alun perin tarkoitettu vain 3D-ympäristön mallintamiseen, niin DirectX 8 -versiossa DirectDraw-kehysrajapinta sulautettiin osaksi Direct3D:tä. Näin DirectX sai vastuulleen myös 2D-grafiikan toistamisen. (Wikipedia 2014.)

Direct3D tarjoaa täyden verteksi (Vertex) -ohjelmistoemuloinnin, mutta ei tue pikselipohjaista ohjelmistoemulointia niille ominaisuuksille, joita käytössä olevan näytönohjain ei tue. Direct3D sisältää referenssi rasteroija-wrapperin (Reference Rasterizer), jolla voidaan emuloida grafiikkaa ohjelmistopohjaisesti kiihdytinkortin sijaan, mutta tämä on hyvin raskas ja hidas operaatio. Näin ollen se ei sovellu peleihin tai ohjelmiin, jotka vaativat suurta suorituskykyä. Kuitenkin tätä ominaisuutta käytetään yleisesti virheiden etsimisessä ohjelmistotuotannossa. Referenssirasteroija päivittyi DirecX 10.1:n myötä uuteen reaaliaikaiseen ohjelmalliseen rasterointitekniikkaan nimeltään WARP (Windows Advanced Rasterization Platform). [8, Direct3D 11 Graphics.]

Direct3D on ensisijainen kilpailija avoimen koodin OpenGL:lle, ja molemmilla näillä on oma uskollinen harrastajajoukkonsa. Direct3D on pääasiallinen työkalu opinnäytetyössäni, ja pyrin käyttämään hyödykseni mahdollisimman paljon DirectX 11 -ominaisuuksia, joita en ole aikaisemmissa versioissa vielä koskaan kokeillut.

5.2 DXGI

DXGI (DirectX Graphics Infrastructure) on graafinen infrastruktuuri, joka ensisijaisesti tarkkailee ja hallinnoi isäntäkoneessaan olevaa laitteistoa. Usein syntyy tilanne, että jokin tietokoneen komponentti toimii muita komponentteja hitaammin, ja näin DXGI puuttuu asiaan. Se hallinnoi ja etsii toisistaan riippumattomia matalantason tehtäviä, joita se sitten jakaa laitteiston kesken.

Aikaisemmissa DirectX-versioissa tämän tyyppinen tehtävänjako sisältyi Direct3D API:iin. Kuitenkin se rajoittui luetteloimaan järjestelmäkomponenttien renderöimien kuvaruutujen määriä ulostulosignaalille ohjaamaan gammaa ja hallitsemaan kokoruutuun siirtymistä. DirectX-version 10 myötä kyseinen toiminto haluttiin eristää Direct3D:stä, ja näin syntyi DXGI. [8, DXGI Overview.]

DXGI siis sisältää itsessään aikaisemmin kehitetyt ominaisuudet, joita Direct3D:ssä jo laitteistonhallinnan osalta oli. DXGI:n tarkoitus on kommunikoida suoraan kernel-moodin ajureiden ja laitteiston kanssa, ja näin muodostaa eräänlainen rajapinta näiden ja käyttäjä-moodin ajureiden välille. DXGI on suunniteltu myös hyvin pitkälle tulevaisuuteen ja siihen on tehty myös oma erillinen osionsa tulevaisuuden grafiikkakomponenttien varalta. [8, DXGI Overview.]

Tavallisesti DXGI:tä kutsutaan Direct3D API:n kautta, joka hoitaa kommunikoinnin DXGI:n kanssa. Kuitenkin siihen voidaan tarpeen vaatiessa päästä käsiksi myös suoraan, jos halutaan vaikka ainoastaan listata erilaisia laitteita tai määrätä itse, kuinka tieto esitetään ulostulosignaaleille. [8, DXGI Overview.]

5.3 Direct2D

Direct2D on laitteistokiihdytetty, välitön, 2D-grafiikanluontiin tehty API. Sitä käytetään korkealaatuisen 2D-geometrian, bittikarttojen ja tekstin esittämiseen. Direct2D on suunniteltu toimimaan yhteistyössä niin GDI:n, GDI+:n kuin Direct3D:nkin kanssa. [8, Direct2D.]

Koska Direct2D on natiivikoodilla toteutettu API ja ohjelmoitu C++:lla, niin se on äärimmäisen suorituskykyinen. Direct2D käyttää laiteriippumatonta koordinaatiojärjestelmää, mikä sallii sen automaattisesti skaalatuvan esitetyn näyttöresoluution ja näyttölaitteen mukaan. Kaikesta yksinkertaisuudestaan huolimatta Direct2D tukee useita erilaisia kuvanparannus- ja optimointitekniikoita, kuten esimerkiksi DirectWritea tai DirectX 10:n yhteydessä esitettyä WARP-rasterointitekniikkaa. [8, Direct2D.]

5.4 DirectWrite

DirectWrite on DirectX 10:n yhteydessä julkaistu laitteistosta riippumaton tekstinalustusjärjestelmä ja symbolien renderöinti-API, joka suunniteltiin korvaamaan GDI/GDI+. Sen päällimmäisenä ominaisuutena on kyky parantaa merkkien erottuvuutta niin dokumenteissa kuin käyttöliittymissäkin. Se tukee mittayksiköitä, sub-pikseleitä, osumien tunnistusta ja useita tekstiformaatteja. Microsoftin kehittelemä ClearType-tekstinrenderöintitekniikka on myös tuettu, jota voidaan käyttää mm. GDI:n tai Direct2D:n kanssa. Di-

rectWrite on täysin UniCode-yhteensopiva ja tukee suuren määrän erimaalaisia merkistöjä. Tuki löytyy myös OpenType -kirjaisimille ja niiden erikoisominaisuuksille, kuten tekstin ulkoasua koristavalle Swash-typografialle. [8, DirectWrite.]

DirectWritea käytetään peliohjelmoinnissa useimmiten Direct2D:n kanssa, jolloin teksti voidaan renderöidä joko grafiikkakiihdytteisesti tai käyttämällä WARP ohjelmistorasteroijaa. Näin merkkeihin saadaan lisättyä erilaisia tehosteita, kuten reunanpehmennystä.

5.5 DirectCompute

DirectCompute on vielä hyvin uusi API, joka tukee Microsoftin kehittelemää laskentatekniikkaa nimeltään GPGPU (General-Purpose computing on Graphics Processing Units). DirectCompute hoitaa kommunikointia laitteiston ja sovellusten välillä, ja pystyy suorittamaan tehokkaasti massiivisia määriä rinnakkaisia laskutoimituksia. Peleissä DirectComputea hyödynnetään parantamaan renderöintiä, fysiikkaa, valaistusta, varjoja ja jopa tekoälyn toimintaa. [8, Direct3D 11 Graphics.]

DirectComputea ei käytetä ainoastaan peliohjelmissa, vaan sitä hyödynnetään muissakin sovelluksissa, jotka vaativat raakaa laskutehoa. Käyttökohteita löytyy niin tieteellisessä laskennassa kuin tutkimusprojekteissakin. GPGPU on myös kulkeutunut huimaa vauhtia kuluttajakäyttöön, ja DirectCompute on oivallinen vaihtoehto, jolla sellainen voidaan toteuttaa. Kuluttajien suosiossa etenkin ovat videonkäsittely ja transkoodaus.

5.6 DirectX Input

DirectX Input on kokoelma erilaisia kirjastoja, jotka liittyvät oheislaitteiden kautta tulevan datan käsittelyyn.

5.6.1 DirectInput

DiriectInput on ohjelmistorajapinta, joka kerää tietokoneen käyttäjältä tulevia signaaleja erilaisten oheislaitteiden kautta. Näihin tiedonsyöttölaitteisiin kuuluvat näppäimistöt, hiiret ja erityyppiset peliohjaimet. Samalla se sallii järjestelmässä erilaisten toimintojen

mappaukset, kuten erikoistoiminnot peliohjaimien liikeratojen ja näppäinyhdistelmien perusteella. Lisäksi se ohjaa tärinä- ja Force Feedback -toimintoja, jos oheislaite tämänlaista tekniikkaa tukee. [9, DirectX Input.]

5.6.2 XInput

Vaikka Microsoft viimeiseen asti halusi pitää kiinni ajatuksesta, että DirectInput voisi yksinään hallita kaikenlaiset mahdolliset hallintalaitteen samalla API:lla, niin erilaisten ongelmatilanteiden vuoksi tästä ajatuksesta jouduttiin luopumaan viimein vuonna 2011, kun Microsoftin itse luoma Xbox 360 -peliohjain ei suostunut toimimaan tässä ympäristössä. Näin Microsoftin jo vuonna 2005 kehitetty kyseistä laitetta ohjaava XInput API jouduttiin lisäämään DirectInput:n rinnalle. Pelikehittäjien keskuudessa tämä ratkaisu on aiheuttanut suurta hämmennystä, koska XInput on äärimmäisen rajallinen API. XInput tukee vain korkeintaan neljää eri laitetta kerrallaan, eikä se toimi kuin Microsoftin omien GamePad-tyyppisten peliohjaimien kanssa. [Wikipedia 2014.]

5.7 DirectX Audio

DirectX Audio on ohjelmistokomponentti, joka tarjoaa äänentoistoon ja äänentallennukseen liittyviä työkaluja. Se pystyy luomaan matalaviiveisen yhteyden ääniohjaimen ja sovelluksen välille, ja näin se voi käsitellä useampia äänivirtoja samanaikaisesti. Tämä alun perin DirectSound-nimisenä käytetty API, on ehkä DirectX:n eniten muokattu ohjelmistokomponentti. Sen käyttötapaa on muutettu elinkaarensa aikana puoleen jos toiseenkin.

Tämänhetkinen versio jakaantuu selvästi kahteen erilaiseen tekniikkaan, joista toinen on nimeltään XACT ja toinen XAudio2. XACT:lla voidaan toistaa erilaisia äänentoistoon tarkoitettua sisältöä ja näitä voidaan myös muokata reaaliajassa. XAudio2:lla taas voidaan toteuttaa omiin tarkoituksiin kokonainen äänen käsittelyyn suunniteltu moottori. Näillä kummallakin on oma käyttötarkoituksensa ja niitä voidaan tarvittaessa ajaa rinnakkain. [9, DirectX Audio.]

5.7.1 XACT

XACT ja siihen liittyvä API ovat korkean tason työkaluja, joita sekä suunnittelijat että ohjelmoijat käyttävät pelien äänien kehittämiseen. XACT tarjoaa puitteet ääniresurssien organisointiin ja hallitsemiseen. Tuotettuja ääniä voidaan muokata erilaisilla suodattimilla ja tiedoston toistolle voidaan määrittää aloitus- ja lopetuspisteitä. Useita äänitiedostoja voidaan yhdistellä aaltopankeiksi (Wave Bank). Myös aaltotiedostoille suunnattuja ehtoja ja asetuksia voidaan yhdistää äänipankeiksi (Sound Bank). XACT API:n tehtäviin kuuluu itse äänitiedostojen muokkaaminen, integrointi ja muutostietojen välittäminen eteenpäin. [9, DirectX Audio.]

5.7.2 XAudio2

XAudio2 on API, joka yhdisti vanhentuneen DirectSoundin ja XAudion. Se tarjoaa monenlaisia työkaluja signaalinmuunnoksiin ja äänen miksaukseen.

XAudio2 pystyy yhdistämään monta erillistä ääntä yhdeksi audio-streamiksi. Tällä pyritään siihen, että äänet toistetaan juuri silloin, kun niitä halutaan toistaa, eikä niiden väliin synny tarpeettomia viiveaikoja. Tämä mahdollistaa myös sen, että ääniasetuksia voidaan jakaa audio-stream-kohtaisesti, joten loppukäyttäjän on helppo määrittää ääniasetuksia tarpeen mukaan. Esimerkiksi jos halutaan vähentää taustamusiikin äänen määrää, niin pelimaailman äänitehosteet eivät tästä kärsi. XAudio2:ssa on myös oma DSP (digital signal processing), jolla pystytään luomaan erilaisia varjostimia ääniympäristöä ajatellen. Tällä voidaan luoda kaikuja, äänien pysähtymistä esteisiin ja äänien heijastumisia erilaisista pinnoista. [9, DirectX Audio.]

5.8 DirectX Diagnostics (DxDiag)

DirectX Diagnostics on vianmäärityökalu, jonka avulla voidaan selvittää DirectX–multimediatekniikan ongelmia. DirectX Diagnostics tunnistaa tietokoneeseen asennettuja komponentteja ja niiden ominaisuuksia. Sen avulla näkee myös helposti tietokoneeseen asennetun DirectX-version.

DirectX Diagnostics käynnistetään kirjoittamalla komentokehoitteessa *dxdiag*. Tämä avaa pienen ikkunan, joka kysyy, haluaako käyttäjä digitaalisesti allekirjoittaa tietokoneeseen asennetut laitteistoajurit. Valinnasta riippumatta ohjelma etenee diagnostiikka-ikkunaan, jossa pystytään tekemään erilaista laitteistoon ja ajureihin liittyvää tarkkailua. DirectX Diagnostics on äärimmäisen hyödyllinen työkalu silloin, kun halutaan etsiä ohjelmasta laitteistoyhteensopivuuksista aiheutuvia ongelmia. Ohjelmassa voidaan tallentaa tekstimuotoon suuri määrä tietoa laitteistossa käytettävästä käyttöjärjestelmästä ja asennetuista ajureista. Tämä tieto on korvaamattoman tärkeää sovelluskehittäjälle.

5.9 DirectSetup

DirectSetup on yksinkertainen kirjasto, joka sisältää toimintoja DirectX:n tietokoneelle asentamista varten. Sen avulla voidaan myös tarkistaa jo asennetun DirectX:n versio. Useimmiten DirectSetup-tiedostot toimitetaan sovelluksen mukana, jos ohjelma DirectX-ohjelmistorajapintaa toimiakseen tarvitsee. Uusimman version DirectSetupista voi kuitenkin ladata suoraan Microsoftin verkkosivuilta.

6 Ohjelman eteneminen ja keskeiset hallinnolliset tietorakenteet

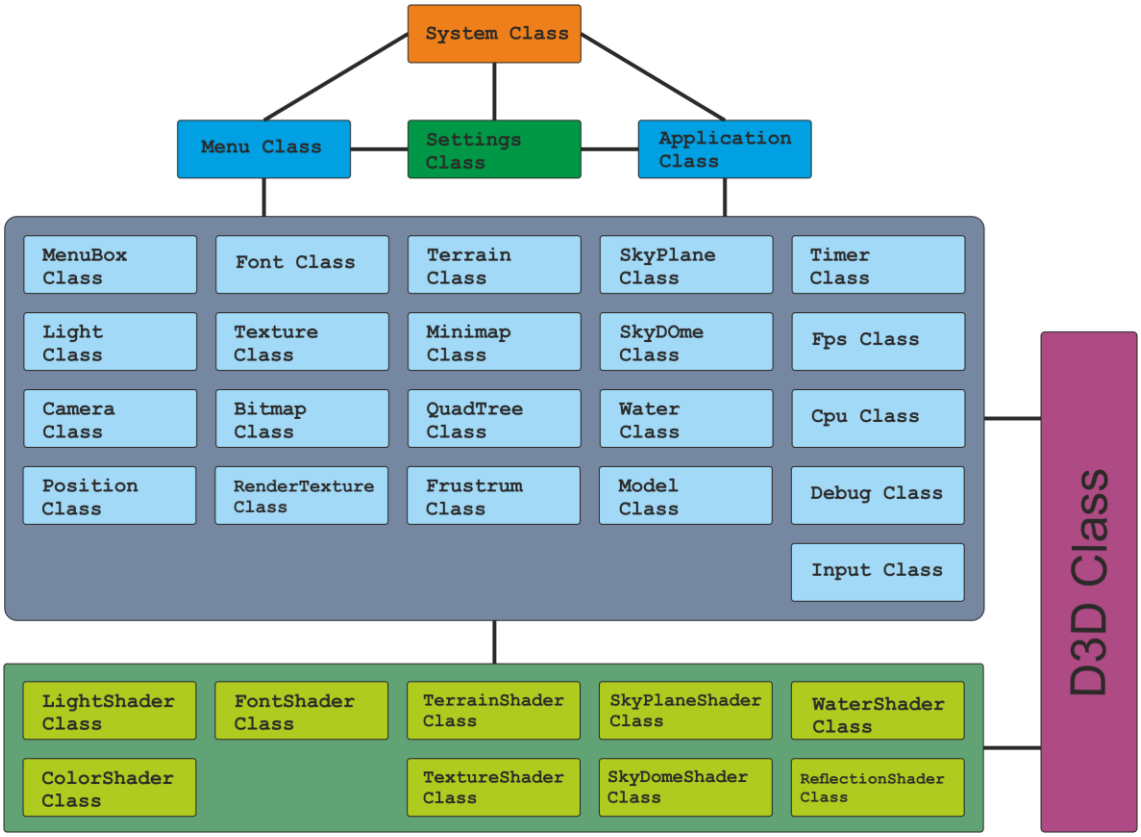
6.1 Ohjelman rakenne

Olen pyrkinyt pitämään pelimoottorini rakenteen mahdollisimman yksinkertaisena, mikä helpottaa mahdollisten virheiden havaitsemisessa. Jokaiselle toiminnalle on tehty aivan oma luokka, ja nämä luokat on sijoiteltu kaikki omiin tiedostoihinsa. Jokainen tehty luokka sisältää luokkatiedoston (.cpp) lisäksi myös otsikon (.h). Jokaisen erillinen algoritmin perään on lisätty tarkastusehto, joka palauttaessaan arvon NULL antaa siihen kohtaa kirjatun virheilmoituksen.

Sovellus alkaa pääohjelmasta nimeltään *main*. Tähän sisältyy varsinaisen peliohjelman alustus, päivitys ja lopetus. Päivityssilmukkaa toistetaan niin kauan, kunnes silmukka keskeytetään, ja sen jälkeen edetään sovelluksen sulkemiseen.

Pelin rungosta huolehtii luokka nimeltä *Systemclass*. Se luo ikkunat, huolehtii DirectX:n käytöstä ja peliohjelman etenemisjärjestyksestä. SystemClass on näin ollen luokka, joka

ajetaan ensimmäisenä ja suljetaan viimeisenä. Sen seurana on tällä hetkellä kaksi pääasiallista pelitilaluokkaa. Näiden nimet ovat *MenuClass* ja *ApplicationClass*. MenuClass sisältää pelin alkuvalikon ja ApplicationClass itse pelitilan. Nämä edellä mainitut luokat käyttävät paljon apuluokkia, joita käytetään joko yksinään tai erikseen muiden luokkien rinnalla. Näitä ovat esimerkiksi erilaiset grafiikkaa, oheislaitteita ja diagnostiikkaa hallitsevat luokat. Apuluokat taas käyttävät erilaisia varjostinluokkia, jotka on jaettu myös erikseen käytettävyyden ja selkeyden vuoksi. Kaikille apuluokille taas yhteisiä luokkia on kaksi, D3DClass ja SettingsClass. D3DClass on DirectX:n oma luokka, joka käyttää DirectX-kirjastoja. Kaikkea DirectX:ään liittyvää käytetään aina D3DClass-luokan kautta. SettingsClass sisältää taas yhteisiä asetuksia, kuten grafiikkatiloja ja näyttöresoluutiota. Ohjelman etenemistä esittää kuvassa 1 näkyvä luokkarakenne.



Kuva1. Ohjelman luokkarakenne.

6.2 Ohjelman eteneminen

Ohjelman perusrunko jakaantuu selvästi kolmeen osaan, ja samaa mallia käyttävät kaikki ohjelmassa käytettävät luokat. Ensimmäisenä tapahtuu alustus, jolla luodaan tarvittavat oliot ja annetaan niille alkuarvot. Tämän jälkeen ajetaan tietoja päivittävää silmukkaa niin kauan, kunnes saadaan keskeytyspyyntö. Keskeytyspyynnön saatua nollataan ja poistetaan kaikki oliot. Koodiesimerkissä 1 on ote main.cpp:n toiminnasta.

```
bool result;
SystemClass* System;

// Create the system object.
System = new SystemClass;
if(!System)
{
            return 0;
}

// Initialize and run the system object.
result = System->Initialize();
if(result)
{
            System->Run();
}

// Release wrapper objects
System->CleanUp();


// Release system object and exit program.
System->Shutdown();
delete System;
System = 0;

return 0;
```

Koodiesimerkki 1. System-olion luonti, pääsilmukan kierrättäminen ja olion tuhoaminen käytön jälkeen

Initialize

Initialize on vaihe, jossa luodaan ja alustetaan kaikki pelissä tarvittavat oliot. Kun peliohjelma käynnistetään, niin pääohjelma alustaa SystemClass-olion. SystemClass-luokka sisältää Initialize-funktion, joka huolehtii kaikesta graafisen käyttöympäristön alustamisesta. Se ajaa erillisen funktion Windows-ikkunan luomiseen ja alustaa ikkunassa ajettavat instanssit. Myös jokainen pelitilaluokka sisältää Initialize-funktion. Funktio alustaa

kaikki tässä kyseisen luokan käyttämät oliot, kuten kamerat, oheislaitteet, graafiset objektit, varjostimet ja tietenkin myös DirectX:n.

Update

Update-funktio päivittää ohjelmaa. Pääohjelma päivittää SystemClass-oliota, SystemClass-luokka päivittää pelitilaluokkia ja pelitilaluokat päivittävät apuluokkia ja varjostimia. Käytännössä koko pääohjelmassa oleva Update-funktio onkin jono peräkkäisiä päivitysfunktiota.

Release

Release-funktio kirjaimellisesti vapauttaa käytetyt oliot. Niille annetaan ensin NULL-arvo, ja tämän jälkeen olio poistetaan muistista. Kun pääohjelma suorittaa Release-funktion, niin se ajaa myös SystemClass-olion Release-funktion. Jälkimmäinen Release-funktio taas sisältää kaikki pelitilaluokkien Release-funktiot.

6.3 Game-State Management (GSM)

Game-State Management on peliohjelman oma hallintamanageri, joka ohjaa peliohjelman kulkua. Se voidaan toteuttaa monella eri tavalla, mutta tässäkin tilanteessa suosin yksinkertaista vaihtoehtoa. Peliohjelmallani on 5 eri tilaa. Siinä on tilat *Splash*, *Menu*, *Game*, *Options* ja *Exit*. *Splash* on tila, joka ladataan aina ohjelman alussa. Ensimmäisen näistä on tarkoitus olla eräänlainen aloitusruutu, jossa voi olla joko pieni videon pätkä tai peliä esittävä kuva. Peliohjelma etenee automaattisesti määrätyn ajan kuluessa Menu-tilaan.

Menu-tilassa peliohjelma käynnistää peli-ikkunaan alkuvalikon. Alkuvalikosta voidaan käynnistää peli, siirtyä asetuksiin tai sulkea ohjelma. Menussa voi liikkua toistaiseksi vain näppäimistön avulla, mutta tulevaisuudessa haluan mahdollistaa myös hiirellä liikkumisen. Kun valikosta valitsee asetukset, niin pelitila siirtyy kohtaan Options. Valikosta valitsemalla itse pelin siirrytään pelitilaan nimeltä Game. Options-tilassa on tarkoitus muokata pelin graafisia asetuksia, kuten näyttötarkkuutta, kontrastia ja reunanpehmennystä. Tästä tilasta päästään takaisin Menu-tilaan. Game-tilassa käynnistetään varsinainen

peli. Sitä toistetaan niin kauan, kunnes keskeytyspyyntö annetaan. Peli palautuu tämän jälkeen takaisin Menu-tilaan.

7 Graafisen käyttöympäristön hallinta

Windows-ympäristöön tarkoitettu ohjelma aloitetaan aina ensin ikkunoiden luonnilla. Peliohjelmoinnissa yleensä käytetään vaan yhtä interaktiivista ikkunaa, johon tuodaan erilaisia ohjelmallisia toteutuksia. Näitä toteutuksia kutsutaan instansseiksi. Koodiesimerkissä 2 on esitelty Windows-ikkunan luontifunktio parametreineen.

```
void SystemClass::InitializeWindows(int& screenWidth, int& screenHeight)
{
WNDCLASSEX wc;
DEVMODE dmScreenSettings;
int posX, posY;

// Get an external pointer to this object.
ApplicationHandle = this;

// Get the instance of this application.
m_hinstance = GetModuleHandle(NULL);

// Give the application a name.
m_applicationName = L"iEngine";

// Setup the windows class with default settings.
wc.style         = CS_HREDRAW | CS_VREDRAW | CS_OWNDC;
wc.lpfnWndProc   = WndProc;
wc.cbClsExtra    = 0;
wc.cbWndExtra    = 0;
wc.hInstance     = m_hinstance;
wc.hIcon         = LoadIcon(NULL, IDI_WINLOGO);
wc.hIconSm       = wc.hIcon;
wc.hCursor       = LoadCursor(NULL, IDC_ARROW);
wc.hbrBackground = (HBRUSH)GetStockObject(BLACK_BRUSH);
wc.lpszMenuName  = NULL;
wc.lpszClassName = m_applicationName;
wc.cbSize        = sizeof(WNDCLASSEX);

// Register the window class.
RegisterClassEx(&wc);

// Determine the resolution of the clients desktop screen.
screenWidth  = GetSystemMetrics(SM_CXSCREEN);
screenHeight = GetSystemMetrics(SM_CYSCREEN);

// Setup the screen settings depending on whether it is running in full screen or
in windowed mode.
if(FULL_SCREEN)
{
// If full screen set the screen to maximum size of the users desktop and 32bit.
memset(&dmScreenSettings, 0, sizeof(dmScreenSettings));
dmScreenSettings.dmSize       = sizeof(dmScreenSettings);
```

```
dmScreenSettings.dmPelsWidth  = (unsigned long)screenWidth;
dmScreenSettings.dmPelsHeight = (unsigned long)screenHeight;
dmScreenSettings.dmBitsPerPel = 32;
dmScreenSettings.dmFields     = DM_BITSPERPEL | DM_PELSWIDTH | DM_PELSHEIGHT;

// Change the display settings to full screen.
ChangeDisplaySettings(&dmScreenSettings, CDS_FULLSCREEN);
// Set the position of the window to the top left corner.
posX = posY = 0;
}
else
{

// If windowed then set it to 800x600 resolution.
screenWidth  = 800;
screenHeight = 600;

// Place the window in the middle of the screen.
posX = (GetSystemMetrics(SM_CXSCREEN) - screenWidth)  / 2;
posY = (GetSystemMetrics(SM_CYSCREEN) - screenHeight) / 2;
}

// Create the window with the screen settings and get the handle to it.
m_hwnd = CreateWindowEx(WS_EX_APPWINDOW, m_applicationName, m_applicationName,
WS_CLIPSIBLINGS | WS_CLIPCHILDREN | WS_POPUP,
posX, posY, screenWidth, screenHeight, NULL, NULL, m_hinstance, NULL);

// Bring the window up on the screen and set it as main focus.
ShowWindow(m_hwnd, SW_SHOW);
SetForegroundWindow(m_hwnd);
SetFocus(m_hwnd);

// Hide the mouse cursor.
ShowCursor(false);

return;
}
```

Koodiesimerkki 2. Windows-ikkunan luonti parametreineen.

Peli-ikkunassa toistetaan erilaisia instansseja, joista pienimpiä ja kevyimpiä pidetään muistissa ja kutsutaan aina tarvittaessa. Esimerkiksi usein toistetut latausikkunat ja valikot ovat hyvä pitää muistissa, koska näin niiden kutsuminen on nopeaa. Kuitenkin kaikkein raskaimmat ja eniten muistia vievät toteutukset on syytä ladata aina erikseen ja sulkea käytön jälkeen muistin vapauttamiseksi. Koodiesimerkissä 3 esitellään menu-instanssi ja määritetään se ajettavaksi juuri luodussa ikkunassa.

```
// Create the menu wrapper object.
m_Menu = new MenuClass;

// Initialize the menu wrapper object.
result = m_Menu->Initialize(m_hinstance, m_hwnd, m_screenWidth, m_screenHeight);
```

Koodiesimerkki 3. Menu-olion luonti. Esimerkistä on selkeyden vuoksi jätetty pois virheiden tarkistukseen tarkoitetut komennot.

7.1 Kameraluokka ja D3DXMATRIX

3D-ympäristöä luodessa ja hallittaessa avuksi tulevat matriisit. D3DXMATRIX on Direct3D:n dynaaminen linkkikirjasto (DLL) matriisien käsittelyä varten. Sillä voidaan tietenkin luoda, muokata tai poistaa matriisitaulukkoja, mutta myös tehdä matriiseilla erilaisia laskutoimituksia ja muunnoksia. D3DMATRIX sisältää valmiit työkalut, joilla matriiseja voi skaalata, kääntää ja monistaa. Peliohjelmoinnissa yleisimmin käytetyt matriisit ovat 2x2-, 3x3- tai 4x4-matriiseja, koska ne perustuvat DirectX:n omiin vektorimalleihin (D3DXVECTOR2, D3DXVECTOR3, D3DXVECTOR4). 3D-ympäristöissä olen nähnyt käytettävän vain 4x4-matriiseja ja näitä olen itsekin tottunut enimmäkseen käyttämään. D3DMATRIX ei sinällään ole rajoitettu matriisien koon suhteen, kunhan laitteistossa vain riittää tehoa näiden laskemiseen. DirectX:ssä käytetään oletuksena vasenkätistä koordinaatistoa. [10, s. 30-35.]

Kameraluokka on ehdoton työkalu, kun käsitellään liikkuvaa ympäristöä ja etenkin kolmiulotteista maailmaa. Kamera-olio toimii käytännössä samalla tavalla kuin oikeakin kamera. Sillä on parametreina sijainti, kuvaussuunta ja linssiominaisuudet. Kolmiulotteista kameran tuottamaa näkymää mallinnetaan matriisien avulla. Kuvaruudulla näkyvä maailma ei oikeasti ole kolmiulotteinen. Kameran tehtävä on vain mallintaa kolmiulotteinen näkymä kaksiulotteiselle pinnalle, eli näytölle. Tämä toteutetaan luomalla keinotekoinen syvyysvaikutelma, jonka ihmissilmä ymmärtää kolmiulotteisena. Koodiesimerkissä 4 on malli kamera-olion luonnista ja alustuksesta.

```
CameraClass* m_Camera;

m_Camera = new CameraClass;

m_Camera->SetPosition(0.0f, 0.0f, -1.0f);
m_Camera->SetRotation(0.0f, 0.0f, 0.0f);
m_Camera->GenerateBaseViewMatrix();
```

Koodiesimerkki 4. Kamera-olion luonti ja sen alustus.

Maailmamatriisi (World Matrix)

World Matrix on koko luodun keinomaailman perusmatriisi. Se määrittää, missä koossa, asennossa ja paikassa luotu maailma on. Sen avulla ympäristöstä voidaan poimia erilaisia pisteitä. Tämä helpottaa mm. uusien objektien lisäämisessä ja törmäyksien tarkistuksessa. Käytän ohjelmakoodissani näistä molempia tekniikoita.

Projektio-matriisi (Projektion Matrix)

Projektiomatriisilla luodaan erilaisia keinotekoisia tehosteita, kuten heijastuksia, varjoja ja vääristymiä. Varjot ja valaistukset ovat laskennallisia tekstuuripintoja ja projektiomatriisia käytetään näiden paikkojen ja muotojen laskemiseen.

Näkymämatriisi (View Matrix)

View Matrix eli näkymämatriisi määrittää pisteen, josta maailmaa katsotaan. Monesti tämä voi olla vaikka pelaajan silmät tai jokin piste pelaajan takana. Näkymämatriisin avulla luodaan keinotekoinen syvyysvaikutelma, joka mallinnetaan kaksiulotteiseen muotoon.

Ortogonaalinen-matriisi (Ortho Matrix)

Ortho Matrix käsittelee ortogonaalista ympäristöä. Tätä tarvitaan silloin, kun halutaan jonkin pisteen seuraavan maailman mukana. Eli käytännössä tämä piste säilyttää määrätyn välimatkan ja asennon suhteutettuna koko avaruuteen. Käytän tätä matriisia, kun käsittelen kaksiulotteisia pintoja kuten tekstiä ja bittikarttakuvia.

7.2 High-Level Shader Language (HLSL)

HLSL on Microsoftin kehittämä näytönohjaimille suunnattu ohjelmointikieli, jota käytetään yhdessä Direct3D API:n kanssa. Sillä voidaan luoda varjostimia Direct3D-putkea (*Direct3D pipeline*) varten ja näitä voidaan käyttää DirectX 11 -versiossa neljällä eri ta-

solla. Nämä tasot ovat kärkipiste-varjostin, pikseli-varjostin, geometria-varjostin ja laskenta-varjostin. HLSL on C-kielen tyylinen korkean tason kieli, johon on lisätty käskyjä ja toimintoja grafiikkasuoritinta ajatellen. HLSL esiintyi ensimmäisen kerran DirectX:n versiossa 9.0, joka käytti Shader Model 1.0:aa. Käyttämäni DirectX 11 on suunniteltu käyttämään Model Shader 5.0:aa, mutta se on alaspäin yhteensopiva ja näin tukee myös kaikkia aikaisempia versioita.

Kärkipistevarjostin (Vertex Shader)

Kärkipiste-varjostin suorittaa per-vertex-käsittelyä kuten muutoksia, pintojen päällystämistä (skinning), vertex-siirtymiä, ja laskennallisia per-vertex-materiaalin ominaisuuksia. Vaihtoehtoisesti vertex-varjostin voi myös esittää muun muassa tekstuurikoordinaatteja, Vertex-väriarvoja, sekä vertex:n valaistusta (*lightning*) ja sumuisuutta (*fog factor*) esittäviä kertoimia.

Geometriavarjostin (Geometry Shader)

Geometria-varjostin suorittaa per-primitiivisiä menetelmiä, kuten materiaalin valintaa ja siluettireunan havaitsemista. Sillä tehdään geometrisiä muutoksia jo aikaisemmin luodulle mallille. Esimerkkeinä mainittakoon vaikka partikkeleiden, karvapintojen tai evien luominen. Myös varjon määrää voidaan säädellä, ja yhdellä suorituskerralla voidaan renderöidä useita eri pintoja (faces), kunhan vain käytetään cube-mallista tekstuuria.

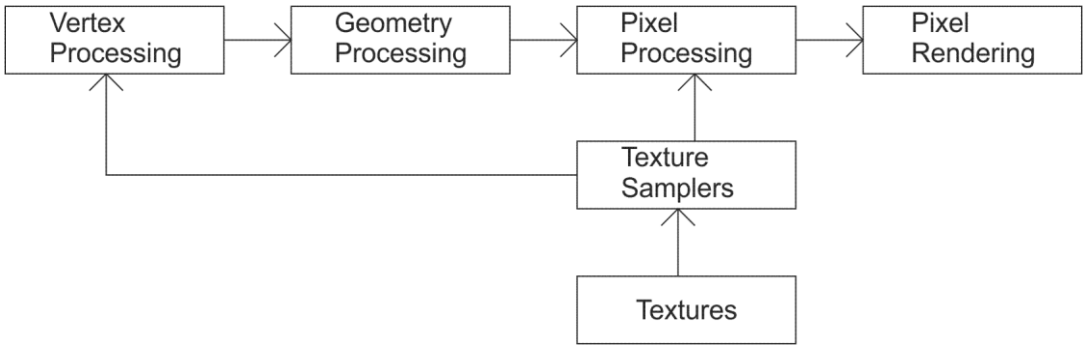
Pikselivarjostin (Pixel Shader)

Pikseli-varjostin suorittaa per-pikseli-käsittelyä kuten tekstuurin sekoittamista (texture blending), valaistusmallin laskentaa ja per-pikseli-normaalin ja / tai ympäristön kartoitusta. Pikselivarjostimet toimivat yhteistyössä kärkipiste- ja geometria-varjostimen kanssa. Vertex-varjostimen luoma ja geometria-varjostimen muokkaama tuote tarjoaa lähtötiedot pixel-varjostimen tuottamalle tuotokselle.

Laskentavarjostin (Compute Shader)

Laskenta-varjostin on ohjelmoitava varjostintaso, joka yltää grafiikkaohjelmoinnin ulkopuolelle. Vaikka sitä käytetäänkin HLSL:n kanssa, niin sillä ei ole mitään tekemistä muun kolmen varjostintason kanssa. Laskenta-varjostin toimii täysin itsenäisesti, ja se tarjoaa mahdollisuuden käyttää näytönohjaimia matemaattisissa toimeksiannoissa. Tämä tekniikka kantaa myös nimeä DirectCompute, joka esiteltiin ensimmäisen kerran DirectX 11:n yhteydessä vuonna 2008. Tämän jälkeen myös DirectX 10:een on lisätty myös vastaava ominaisuus, ja se kantaa nimeä DirectCompute 4.0, kun DirectX 11 versiossa se on DirectCompute 5.0. Tätä varjostinta en opinnäytetyössäni käytä, mutta näin sen tarpeelliseksi mainita, koska se on yhtenä osana HLSL:ää.

Kuva 2 esittää HLSL:n toimintaa, josta olen tarkoituksellisesti jättänyt tesselaation osan pois, koska en tässä opinnäytetyössä aihetta käsittele.



Kuva 2. HLSL:n rakenne

8 Käytetyt 2D-rakenteet

Kaksiulotteiset kuvat ovat yksi oleellisimmista asioista peliohjelmoinnissa. Niillä toteutetaan muun muassa taustakuvia, valikoita, spritejä, tekstuureita ja tekstiä. DirectX 11:ssä kaksiulotteiset kuvat renderöidään mappaamalla tekstuureita polygoneihin. Kuitenkin, kun halutaan bittikarttakuvan toimivan esimerkiksi latauskuvana tai käyttöympäristöä kuvaavana kehikkona, niin joudumme ottamaan Z-syvyyspuskurin pois päältä aina ennen 2D-kuvien piirtämistä. Tämä mahdollistaa sen, etteivät 2D-kuvat jäisi 3D-mallien alle. Tämän lisäksi 3D-maailman ulkopuoliset kuvat näytetään aina erikseen ortogonaalisella

projektiomatriisilla, jotta nämä kuvat säilyttävät paikkansa ja kokonsa, vaikka pelimaailmassa liikuttaisiinkin.

Kaikkia 2D-malleja varten olen luonut oman tekstuuriluokan, jota käytetään yhteisesti kaikkien kaksiulotteisten kuvien tulostamiseen. Koodiesimerkissä 5 on tämän yksinkertaisen luokan rakenne kokonaisuudessaan.

```
#include <d3d11.h>
#include <d3dx11tex.h>

class TextureClass
{
public:
            TextureClass();
            TextureClass(const TextureClass&);
            ~TextureClass();

            bool Initialize(ID3D11Device*, WCHAR*);
            void Shutdown();

            ID3D11ShaderResourceView* GetTexture();

private:
            ID3D11ShaderResourceView* m_texture;
};
```

Koodiesimerkki 5. Tekstuuriluokka kaikkien kaksiulotteisten kuvien lataamiseen.

8.1 Bittikarttatekstuurit

DirectX 11:ssä bittikarttakuvat ovat kaksiulotteisia polygoneja, jotka on päällystetty haluamallamme tekstuuripinnalla. Tämä polygoni muodostuu kolmioista ja paikkavektorin lisäksi sillä on ainoastaan kaksiulotteisia koordinaattiarvoja.

Pelimoottorissani aina bittikarttoja käsitellessä otan huomioon näyttötilan resoluution, jota käytän hyväkseni kuvien sijoittelussa ja skaalaamisessa. Näin pidän huolen siitä, että suurissa resoluutioissa kaksiulotteiset kuvat eivät muutu kohtuuttoman pieniksi, eivätkä ne myöskään joudu kuvan näyttötilan ulkopuolelle.

Bittikarttoja käyttäviä luokkia on pelimoottorissani tällä hetkellä 7. Rakenteellisesti näissä kaikissa on käytännössä muuttujina ainoastaan paikkavektori ja itse tekstuuri. Kuitenkin näiden muuttujien lisäksi on erilaisia luokkakohtaisia muuttujia, jotka tarjoavat lisätietoa

sen hetkisestä bittikarttakuvan tilasta. Näitä lisätietoja käytän muun muassa hiirenkursorin hallinnassa, joka on myös pelimoottorissani bittikarttakuva.

8.2 Graafinen teksti ja fonttiluokka

Tekstin tulostaminen kuvaruudulle on yleensä hyvin tärkeää minkä tahansa ohjelman toiminnan kannalta. DirectX-ikkunassa emme kuitenkaan voi käyttää Windowsin omaa fontti-moottoria vaan joudumme luomaan tätä käyttöä varten aivan oman. Tämä koostuu itse fonttiluokasta ja varjostinluokasta.

Ensiksi tarvitaan kaksiulotteinen kuva, joka sisältää kaikki tarvittavat kirjaisinmerkit. Käytän tähän tarkoitukseen kuvassa 3 näkyvää 1024x16 kokoista DDS-tekstuuria.

Kuva 3. Fonttitekstuuri.

Kuten kuvasta huomataan, niin kaikki kirjasinmerkit ovat samassa kuvatiedostossa. Tämän perusteella voimme yksinkertaisen fonttimoottorin, joka piirtää yksittäisiä merkkejä ruudulle haluamallamme tavalla. Tämä toimii käytännössä niin, että luomme neliön muotoisen kuvan, joka koostuu kahdesta kolmiosta. Sen jälkeen tulostamme fonttitiedoston avulla halutun tekstuuripinnan tähän luotuun neliöön. Kyseinen teksturointi vaatii kuitenkin sen, että DirectX:n täytyy tietää, mikä merkki kuvatiedostosta milloinkin piirretään. Tämän vuoksi joudumme luomaan vielä eräänlaisen indeksointi-tiedoston, jonka avulla voidaan poimia halutun alueen fontti-tiedostosta teksturointia varten. Koodiesimerkissä 6 on pieni osa tästä tiedostosta, mutta siitä nähdään, kuinka tämä indeksointi tiedoston sisällä on tehty.

```
32   0.0         0.0          0
33 ! 0.0         0.000976563  1
34 " 0.00195313  0.00488281   3
35 # 0.00585938  0.0136719    8
36 $ 0.0146484   0.0195313    5
37 % 0.0205078   0.0302734    10
38 & 0.03125     0.0390625    8
```

```
39 ' 0.0400391  0.0410156   1
40 ( 0.0419922  0.0449219   3
```

Koodiesimerkki 6. Esimerkki indeksointitiedostosta.

Koodiesimerkissä 6 olevat arvot järjestyksessä tarkoittavat seuraavaa: merkin ASCII-arvo, merkin symboli, tekstuurin vasen koordinaatti, tekstuurin oikea koordinaatti ja merkin leveys pikseleinä. Näiden tiedostojen perusteella voimme luoda rakenteen, joka ottaa vastaan haluttavan tekstuurin koordinaatit ja koon. Koodiesimerkissä 7 tämä tietorakenne löytyy nimellä FontType.

Seuraavaksi luomme VertexType-tietorakenteen, jonka avulla voimme luoda neliön muotoisen alueen merkin renderöintiä varten. Tämä yhden neliön piirtämiseen tarvitaan kaksi kolmiota, ja nämä kolmiot tarvitsevat tiedoikseen vain paikkakoordinaatit ja käytettävän tekstuurin. Tämä on esitetty myös Koodiesimerkissä 7.

Tämän jälkeen luomme funktion, joka käsittelee kolmioista muodostuvan vertex-listan rakentamista palauttaen samalla arvoja merkkien tulostamista varten. Tämä funktio saa nimekseen BuildVertexArray.

Viimeiseksi lisäämme funktiot resurssien lataamista ja vapauttamista varten, jotta voimme käyttää luokkaa halutullamme tavalla.

```
class FontClass
{
private:
            struct FontType
            {
                        float left, right;
                        int size;
            };

            struct VertexType
            {
                        D3DXVECTOR3 position;
                        D3DXVECTOR2 texture;
            };

public:
            FontClass();
            FontClass(const FontClass&);
            ~FontClass();

            bool Initialize(ID3D11Device*, char*, WCHAR*);
            void Shutdown();
```

```
            ID3D11ShaderResourceView* GetTexture();

            void BuildVertexArray(void*, char*, float, float);

private:
            bool LoadFontData(char*);
            void ReleaseFontData();
            bool LoadTexture(ID3D11Device*, WCHAR*);
            void ReleaseTexture();

private:
            FontType* m_Font;
            TextureClass* m_Texture;
};
```

Koodiesimerkki 7. Fonttiluokan määritys.

Ohjelmassani pelkästään tätä fonttiluokkaa käsittelevää ohjelmakoodia varjostimineen on yli 2000 riviä, joten en näe sitä asialliseksi listata kokonaisuudessaan tähän. Tarkoituksenani on pysyä ainoastaan yhdessä fonttiluokassa, jota on mahdollista käyttää erilaisissa tilanteissa, ja näin sen täytyy sallia ottamaan vastaan monia erilaisia parametreja.

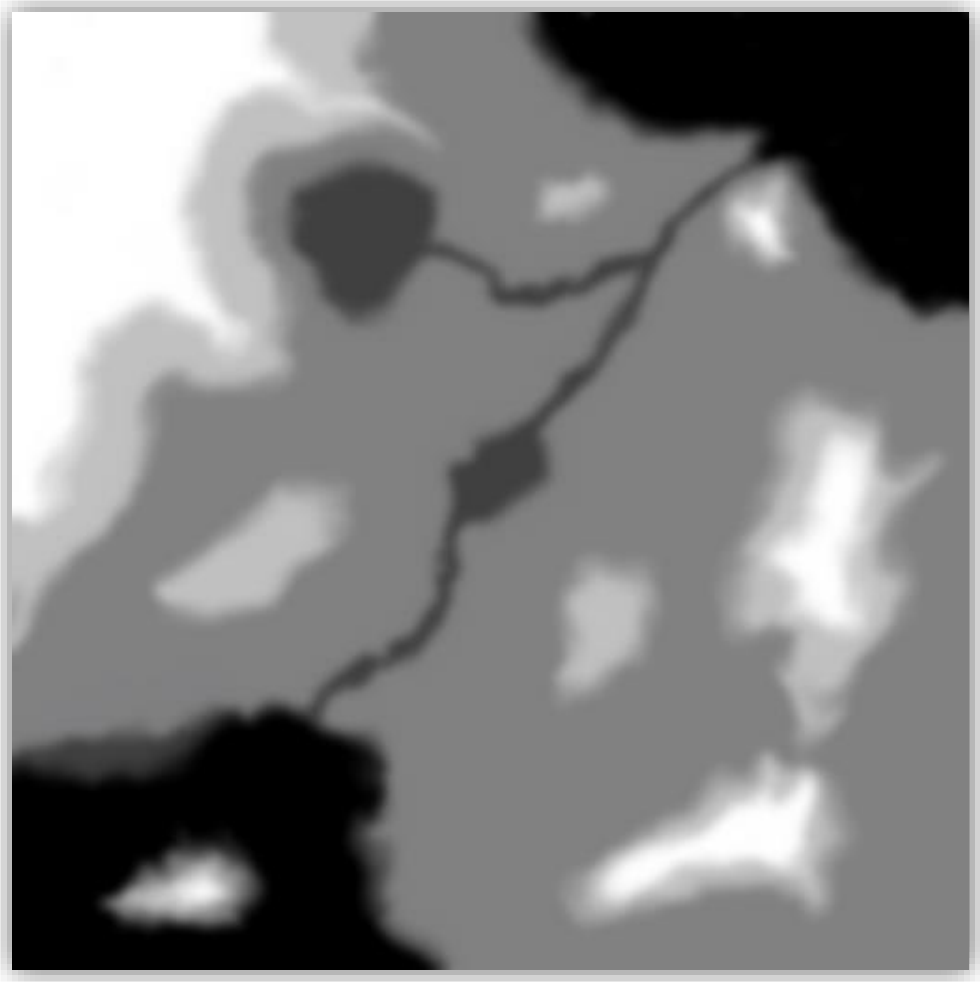
9 Käytetyt 3D-rakenteet

9.1 Korkeuskartta (Heigth Map Terrain)

Korkeuskartan avulla luotu maasto on tekniikka, jolla tallennetaan korkeuspisteitä jossakin formaatissa olevaan tiedostoon. Yleisimpiä metodeja tämänlaisen datan tallentamiseen ovat bittikartta-, raw-, teksti- tai binääritiedostot. Näissä käytetään arvoja väliltä 0-255, jossa matalin kohta maastossa on 0 ja korkein 255.

9.1.1 Korkeuskartta bittikarttakuvasta

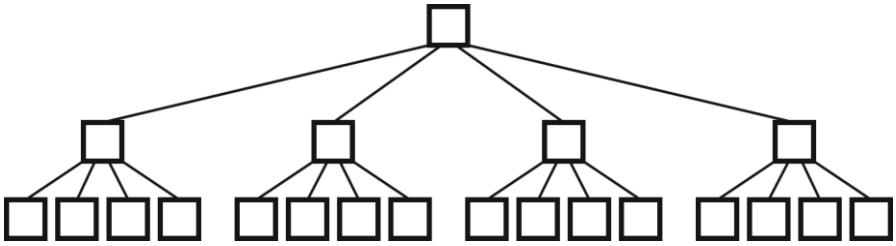
Omassa ohjelmaesimerkissäni päätin käyttää korkeuskartan luomiseen bittikarttatiedostoa. Käytän harmaasävyistä 8-bittistä 1024x1024 -kokoista BMP-kuvatiedostoa, jossa musta väri on kartassa kaikkein matalin kohta ja valkoinen taas kaikkein korkein. Käytin bittikartta-kuvan luomiseen Corel PhotoPaint X6 -ohjelmaa. Kuvassa 4 on yksi korkeuskarttaa varten tekemistäni kuvatiedostosta.



Kuva 4. Korkeuskarttaa varten luotu BMP-kuva.

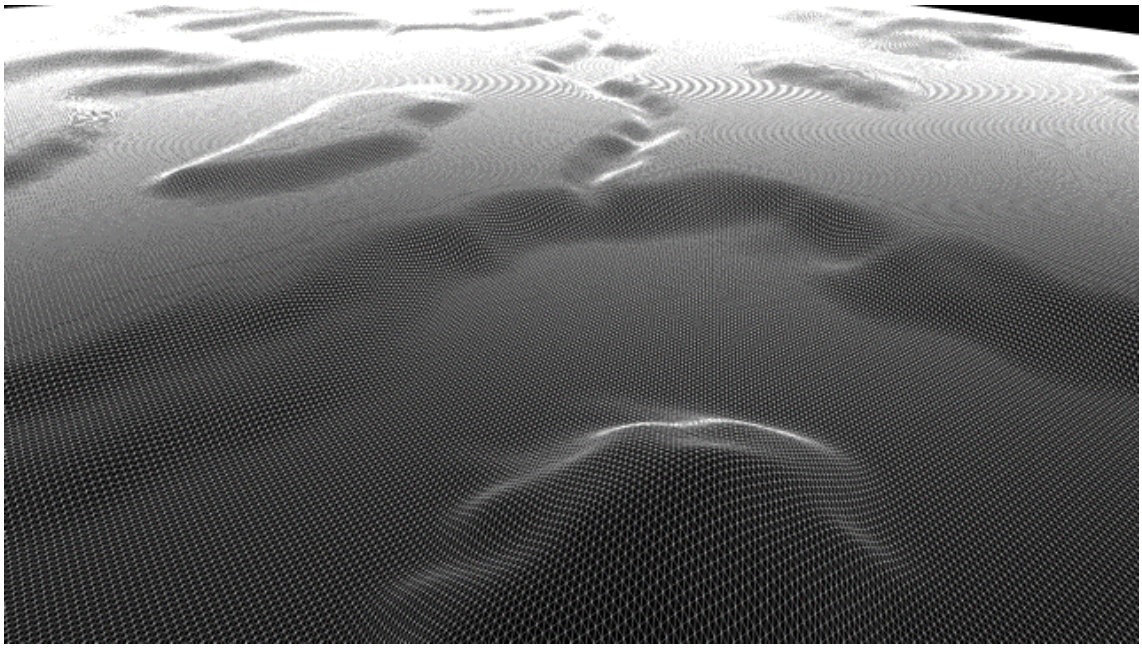
9.1.2 Quad Tree -osiointi (Quad Tree Partition)

Korkeuskarttaa tehdessäni päätin käyttää QTP-algoritmia, jolla pystytään luomaan hyvinkin suuria korkeuskarttoja. Quad Tree lisää toiminnallisuutta niin optimoinnin kuin käytännöllisyydenkin suhteen. Quad Tree on nimensä mukaisesti eräänlainen neliömallia hyödyntävä rakennepuu, jossa käsiteltävä alue jaetaan pienempiin osiin (*node*), ja näitä voidaan käsitellä itsenäisesti. Tämä on oivallinen keino optimoida suorituskykyä esimerkiksi silloin, kun halutaan, että kaukana olevat alueet pysyvät piirtoalueen ulkopuolella. Myös kolmioiden poimintaan (*triangle picking*) tätä tekniikkaa on hyvä käyttää suurissa alueissa, koska näin voidaan rajata pienempiä alueita, joissa milläkin hetkellä törmäyksiä ja rajojen ylityksiä tunnistetaan. Tämä keventää huomattavasti laskennallista työtä, ja näin ollen jäljellä olevia resursseja voidaan hyödyntää toisaalla. Kuva 5 esittää Quad Treen toimintaa käytännössä.



KUVA 5. Quad Tree:ta kuvaava rakennepuu.

Opinnäytetyössä käyttämässäni korkeuskartassa rajoitin yhden alueen (*node*) maksimikooksi 20 000 kolmiota. Kun alueella on kolmioita enemmän kuin tämä arvo määrittää, niin kyseinen alue jaetaan neljään osaan. Jos jokin näistä osista vielä ylittää vielä tämän määritetyn arvon, niin jaetaan sekin vielä neljään osaan jne. Koko korkeuskartassani on vähän yli 500 000 kolmiota, ja näin kartta on jaettu 64 pienempään osaan. Kuvassa 6 voi nähdä Quad Tree -muunnoksen jälkeen toteutetun mallin korkeuskartasta.

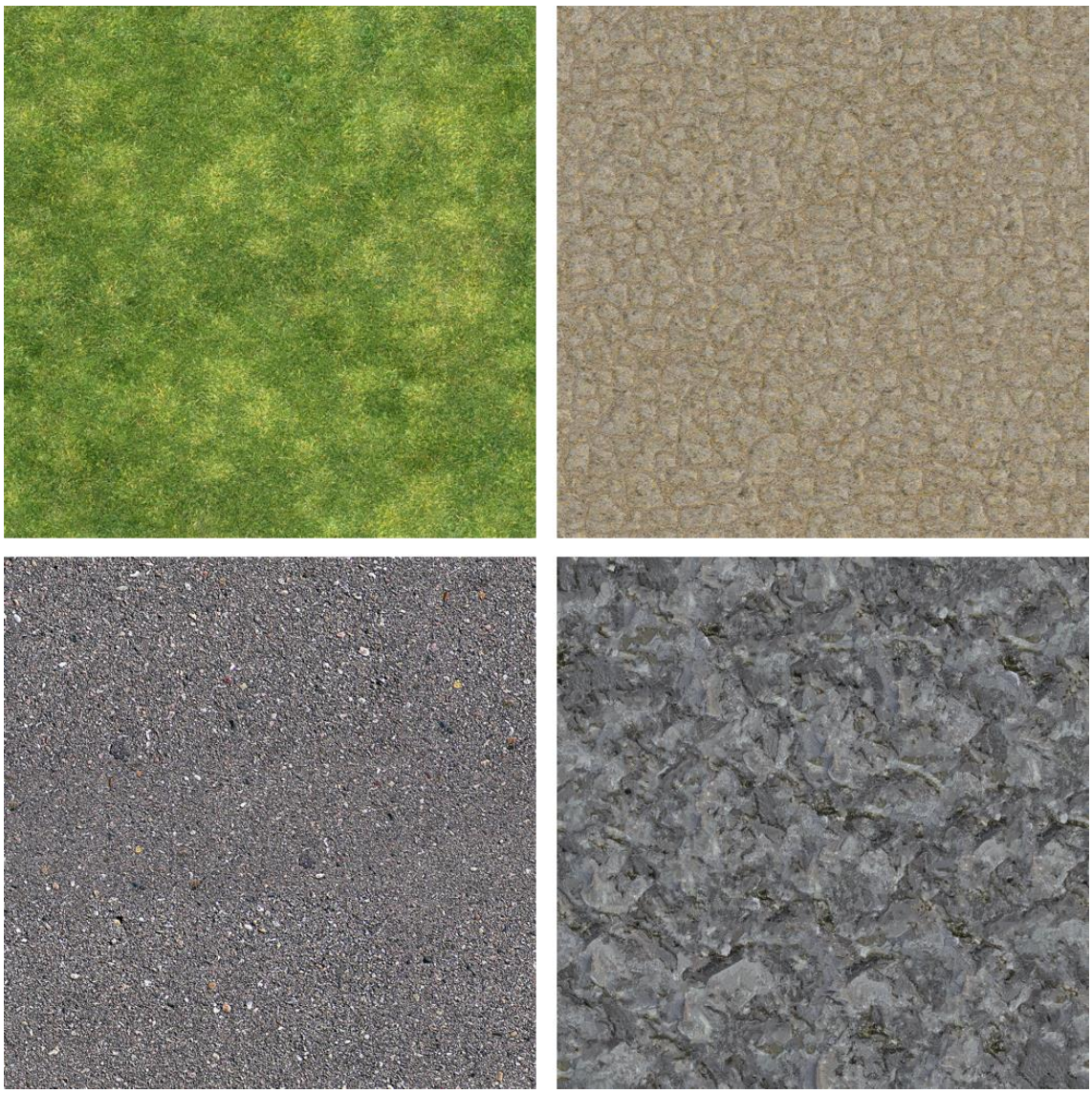


Kuva 6. Wire-Frame-näkymä generoidusta korkeuskartasta.

9.1.3 Korkeuskartan teksturointi

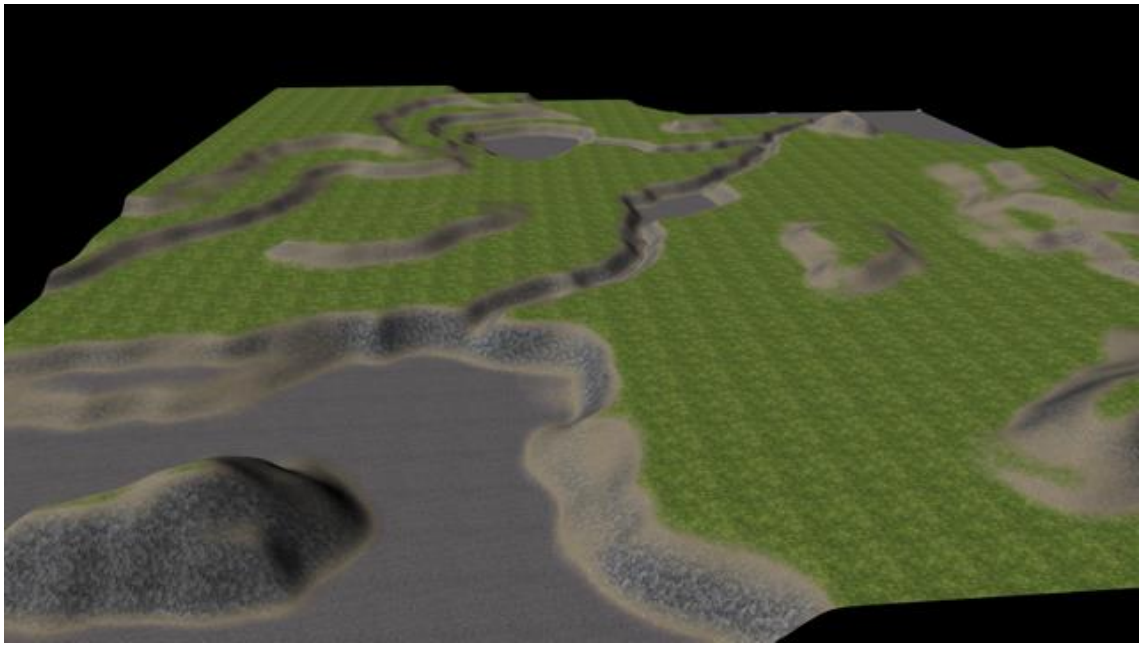
Päätin käyttää esimerkin vuoksi kahta erilaista teksturointimenetelmää yhdistettynä, jotta saisin lisättyä enemmän yksityiskohtia maaston muotoja ajatellen. Ensisijaisena teksturointimenetelmänä käytän tekniikkaa nimeltä *Slope Based Terrain Texturing*, eli teksturointi tapahtuu seuraamalla maaston kaltevuutta. Näin erilaisille jyrkkyysasteille voidaan

laittaa erilaiset tekstuurit. Toisena tekniikkana edellisen rinnalla käytän tekniikkaa nimeltä *Height Based Terrain Texturing*, eli teksturointi tapahtuu maaston korkeuden mukaan. Lopputuloksena on yhdistelmä, joka tarkkailee yhtä aikaa sekä maaston kaltevuutta että korkeusarvoja, ja tämän perusteella luo teksturoinnin koko korkeuskartan alueelle. Toteutin tämän kaiken käyttämällä HLSL:ää, ja neljää erilaista tekstuuria. Kuvassa 7 ovat kaikki nämä neljä tekstuuria eriteltynä. Tekstuurien koko on 2048x2048 pikseliä.



Kuva 7. Korkeuskartan pinnan tekstuurit.

Valmista teksturoitua korkeuskarttaa esittää kuva 8.



Kuva 8. Teksturoitu korkeuskartta.

9.1.4 Height Based Movement

Jotta kamera-olioni ei joutuisi missään vaiheessa kartan alapuolelle, käytän Quad Treen rinnalla yhtä ominaisuutta, jolla pystyn tarkkailemaan kamera-olion sijainnin perusteella tarkastamaan sen hetkisen maaston korkeuden. Tätä samaa ominaisuutta voin toki käyttää vaikka jonkin yksittäisen 3D-mallin hallintaan, jos haluan rajoittaa sen Y-suuntaisen paikkavektorin maaston mukaan.

9.2 Valotehosteet

Pelimaailmaan saadaan lisää realistisuutta, kun siihen lisätään erilaisia valonlähteitä. Loin yhden valoluokan, jossa on yhdistettynä kaksi erilaista valorakennetta. Ensimmäinen näistä on aurinkoa kuvastava kohdevalon (*directional diffuse color*) tyylinen, ja toinen näistä valottaa (ambient color) käsiteltävää objektia. Kummallakin valotyypillä on kolme erillistä väriarvoa. Nämä ovat punainen, vihreä ja sininen. Nämä arvot määritellään liukuluvulla, jonka arvo on 0.0-1.0f. Mitä pienempiä luvut ovat, sitä heikompi on myös valaistuksen teho. Puhdas valkoinen valo on näin ollen arvoltaan 1.0, 1.0, 1.0. Valoluokkani käyttää yhtä HLSL:llä tehtyä varjostinta.

Ambient Color

Ambient Color on tekniikka, jolla korostetaan halutun objektin perusväriä. Tällä tavoin voidaan esimerkiksi jokin yksittäinen kohde saada hehkumaan kirkkaammin tai erisävyisenä kuin muut ympärillä olevat objektit. Ambien Color ei itsessään loista valoa ympäristöön; se ainoastaan vaikuttaa kohteen omaan väriin.

Direct Diffuse Color

Direct Diffuse Color on kohdevalo, jolla on annettuna parametreina valovoima ja sen suunta. Tämä on näin ollen oivallinen tämä mallintaa vaikka taivaalta tulevaa auringon valoa. Myös tätä tekniikkaa hyödyntämällä voidaan toteuttaa varjoja. Olen laajentanut valoluokkaani myös varjojen mallintamista varten, mutta aihe on suhteellisen suuri, eikä enää mahdu tämän opinnäytetyön sisältöön.

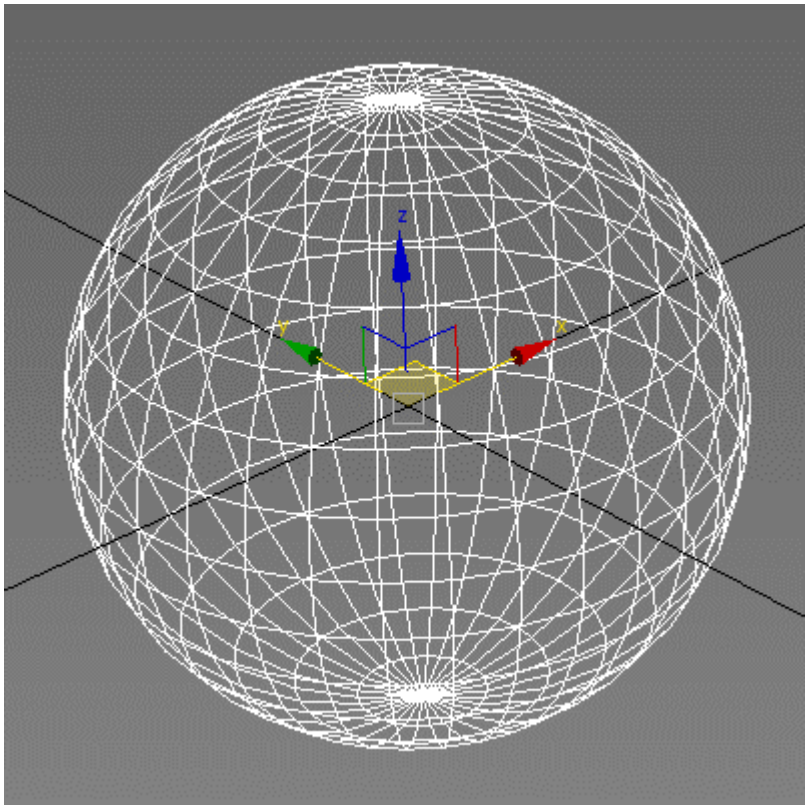
9.3 Pelimaailman taivas (Sky)

Taivaan luominen tuo pelikentälle paljon tunnelmaa, ja näin halusin erityisesti kiinnittää huomiota siihen, miten tämän haluan toteuttaa. Olen aikaisemmin tehnyt taivaita monenlaisilla tekniikoilla, joita ovat olleet *Sky Sphere*, *Sky Cube* ja *Half Sphere*. Tähän projektiin halusin kuitenkin kokeilla jotakin uutta, joten päätin luoda taivaan kahdessa eri osassa. Toinen näistä kuvaa itse taivaan väritystä ja toinen lisää siihen erilaisia graafisia kerroksia. Jotta taivaasta saisi vielä elävämmän näköisen, lisäsin yhden tehosteen, joka mallintaa pilvien muodostumista ja liikettä taivaalla.

9.3.1 Sky Dome

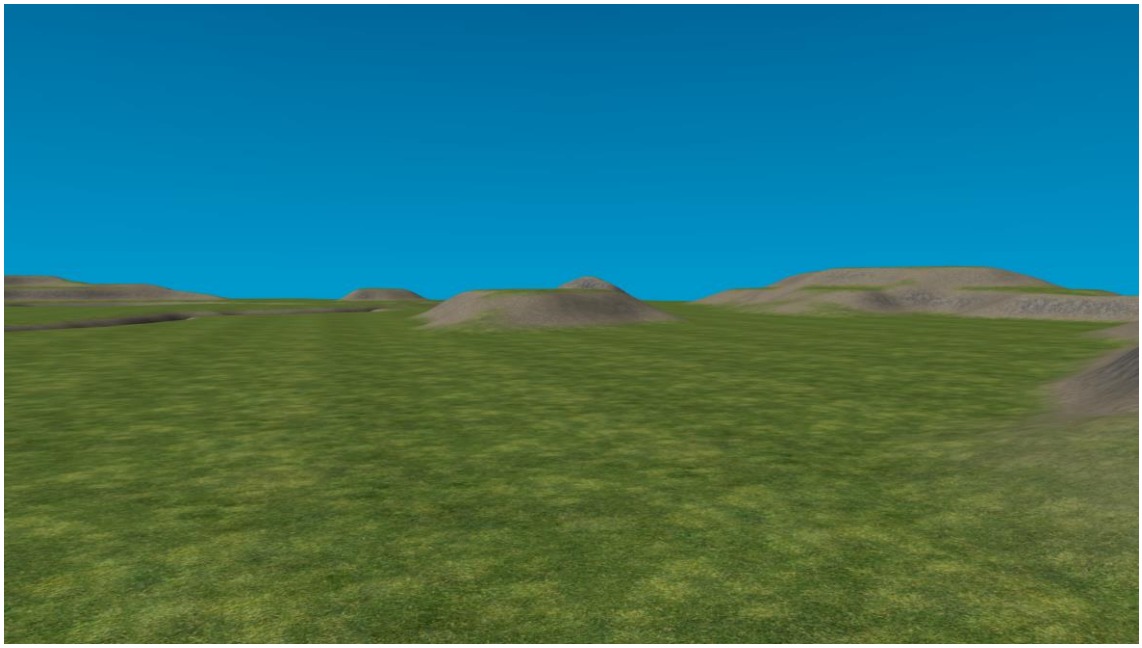
Sky Dome on yleinen nimitys 3D-mallille, joka kuvaa taivaan rakennetta. Sillä ei ole tarkkaan määriteltyä muotoa. Se voi olla yhtälailla täysin pyöreän muotoinen pallo tai vaikka pallon puolikas, johon on lisätty erilliset helmat horisontin kuvaamiseksi. Perusajatus on kuitenkin se, että Sky Domen teksturointi on sen sisäpuolella ja kappaletta myös katsotaan sisältäpäin. Sky Dome liikkuu aina kameran mukana, joten välimatka Sky Domen reunoihin pysyy muuttumattomana. Tietenkin tämä sallii sen, että voidaan määrittää se piste, mistä kohtaa Sky Domea ylipäätään katsotaan.

Käytin tällä kertaa Sky Domena pallon muotoista 3D-mallia, jonka mallinsin 3D Studio Maxilla. Tallensin tämän OBJ-formaattiin ja muunsin tämän tekstimuotoon eräällä aikaisemmin kirjoittamallani ohjelmalla. Koska tämä versio DirectX:stä ei tue enää vanhentunutta X-formaattia, niin tämä ratkaisu helpotti suuresti työn etenemistä. Ajan käyttö ei enää sallinut sitä, että olisin luonut kokonaan täysin oman formaatin pelimoottoriani varten. Kuitenkin tulevaisuudessa tämä on todennäköisesti välttämätöntä. Kuva 9 esittää Sky Domen 3D-mallia.



Kuva 9. Sky Dome. 3DSMax ei näytä 3D-mallissa olevia kolmioita. Todellisuudessa kuitenkin jokainen kuvassa näkyvä nelikulmainen alue on jaettu kahteen kolmioon.

Aikaisemmin kun olen aina lisännyt bittikarttatekstuurit suoraan Sky Domen pintaan, niin halusin tällä kertaa ainoastaan tehdä tähän taustavärin. Taivaan väriä esittävä liukuva väritys on toteutettu kahdella erisävyisellä värillä. Pallon pohjoisnapa on kaikkein tummimmalla värillä värjätty. Keskeltä ja siitä alaspäin pallo on väriltään vaalein. Tällä tavoin sain tehtyä yksinkertaisen havainnollistamisen horisontille. Kuvassa 10 on näkymä tästä liukuvärityksellä tehdystä horisontista.



Kuva 10. Sky Dome.

9.3.2 Sky Plane

Sky Plane on taivasta varten luotu pinta, johon piirretään erilaisia tehosteita. Se voi olla joko staattinen eli muuttumaton tai dynaaminen, jossa tapahtuu jonkinlaisia muutoksia. Staattisena esimerkkinä voi toimia vaikka pilviä kuvaava bittikarttakuva, jota vieritetään Sky Planea pitkin. Dynaaminen Sky Plane taas tarkoittaa sitä, että siinä tapahtuu jonkinlaisia teksturointiin vaikuttavia muutoksia kuten pilvien muodostumista.

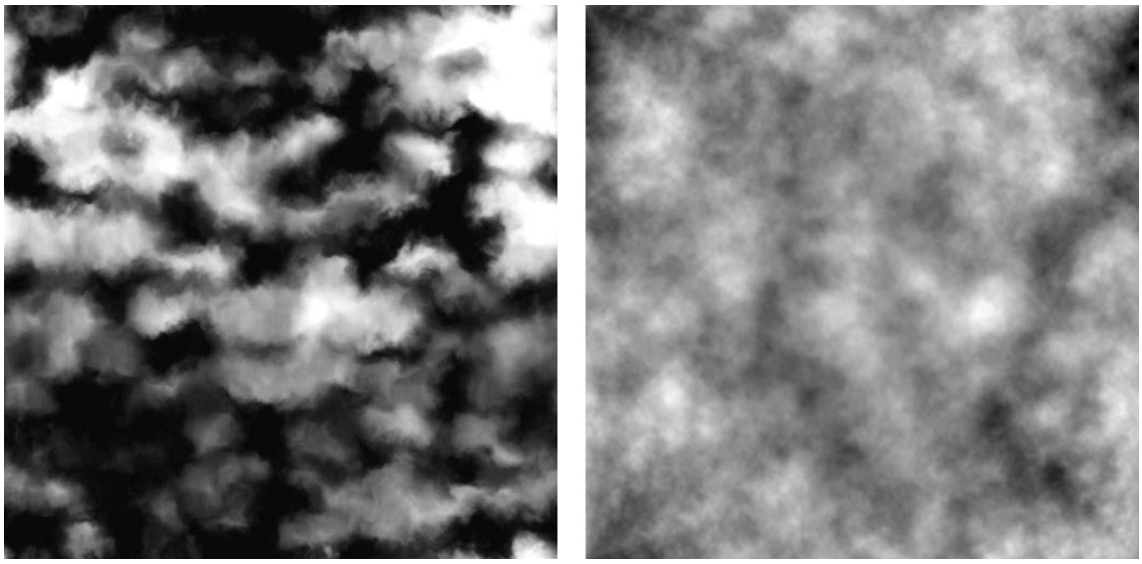
Pelimoottorissani on laskennallisesti tehty Sky Plane, joka on neliömallinen sataan segmenttiin jaettu pinta. Siinä on näin ollen 200 kolmiota. Tämä pinta on tarkoituksellisesti hieman kaareva, joten se myötäilee osittain Sky Domen muotoa. Tämä tapahtuu niin, että Sky Planen keskikohta on samassa pisteessä Sky Domen pohjoisen kärkipisteen kanssa. Kuitenkin Sky Planen kulmapisteitä voin ohjelmallisesti muuttaa Y-akselin suuntaisesti, ja näin pystyn määrittämään haluamani kaarevuuden varsinaiselle tekstuurikerrokselle. Sky Planen sivun pituus on tällä hetkellä kymmenkertainen suhteessa Sky Domen säteeseen.

Bittikartta-pilvet (Bitmap Clouds)

Jotta taivaalle saataisiin myös jotakin näkyvää kuten pilviä, Sky Plane täytyy teksturoida. Tämä voidaan tehdä joko yhdellä tekstuurilla tai useita tekstuureja yhdistelmällä, jolla saadaan havainnollistettua erilaisia pilvikerroksia. Tietenkin tekstuureita voi myös sekoittaa (*alpha blending*), joilla useista erilaisista tekstuureista generoidaan vain yksi uniikki tekstuuri. Tässä opinnäytetyössäni käytän vain yhtä pohjatekstuuria, jota käytän pohjana taivaan animaatiolle.

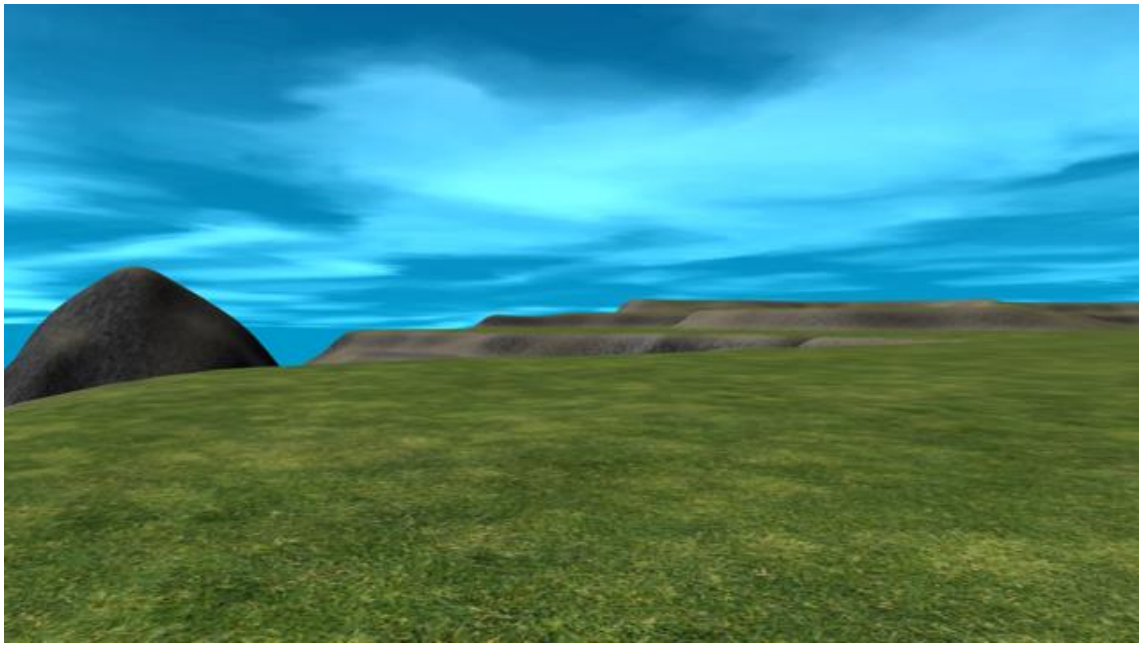
Häiriöllä muuttuvat pilvet (Preturbed Clouds)

Preturbed Clouds on hyvä esimerkki dynaamisesta teksturoinnista. Sillä voidaan pienellä vaivalla luoda näyttäviäkin tehosteita Sky Planen pinnalle. Idea perustuu siihen, että taivaalle valituille tekstuureille tehdään reaaliaikaisia muunnoksia käyttämällä jotakin ulkopuolista häiriötä. Nämä muutokset voidaan tehdä joko täysin laskennallisesti tai käyttämällä erilaisia valmiiksi luotuja tietolähteitä, kuten muita tekstuureita. Valitsin tähän työhön erillisen kohinakartan, jolla pehmennän ja muokkaan pilvien reunoja. Näin lopputuloksena pilvet muuttuvat reaaliajassa erimuotoisiksi, samalla kun tekstuureita vieritetään Sky Planen pinnalla. Kuvassa 11 vasemmalla puolella oleva bittikarttakuva esittää pilvikerroksessa käyttämääni tekstuuria, ja oikealla puolella on taas esillä kohinakartta. Molemmat bittikarttakuvat ovat kooltaan 1024x1024 pikseliä.



Kuva 11. Sky Planen teksturoinnissa käytetyt bittikarttakuvat.

Kuvasta 12 voidaan nähdä valmis pilvikerros, joka syntyi kahden bittikarttakuvan yhdistelmästä.



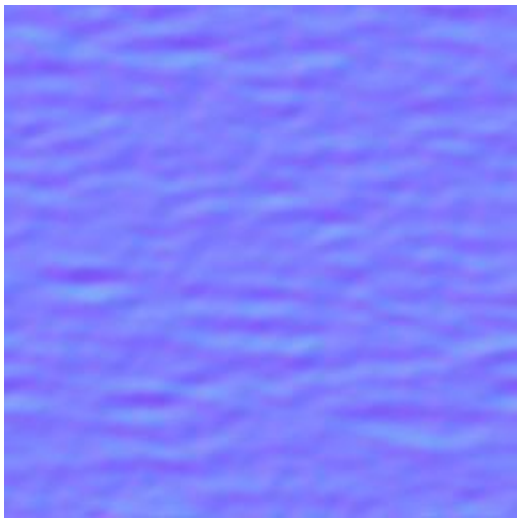
Kuva 12. Erillisellä häiriökartalla toteutettu pilvikerros.

9.4 Pelialueen vesi

Veden luominen on yksi asia, jossa joudun aina painimaan suorituskyvyn ja näyttävyyden kesken. Vesi on useimmiten resurssien osalta raskain peliohjelman komponentti, koska se vaatii useita renderöintikertoja antaakseen toivotun lopputuloksen. Veden voi toki toteuttaa monella tavalla riippuen etenkin siitä, minkälaiseen ympäristöön vettä olemme tekemässä. Merivesi luonnollisesti suurine aaltoineen tulee näyttää erilaiselta kuin pienen puron virtaava vesi tai liki paikallaan pysyvä altaan vesi. Myös erilaiset nesteiden käyttäytymistä mallintavat moottorit ovat vielä erillään tästä kaikesta.

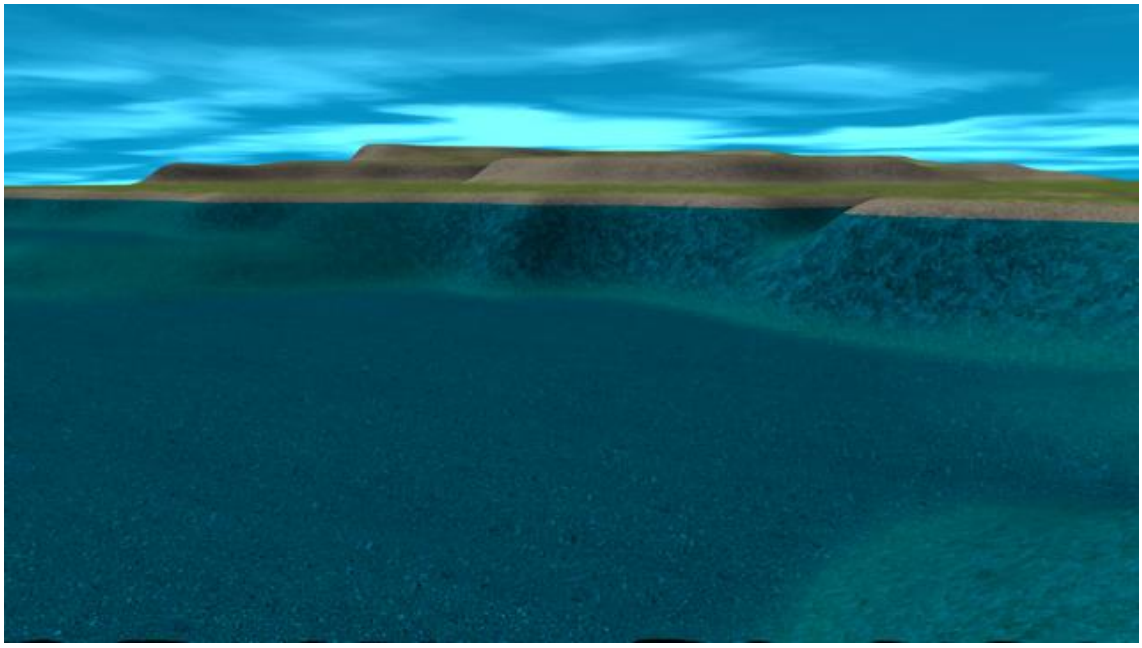
Päätin tässä opinnäytetyössäni luoda yksinkertaisen vesirungon, joka palvelee kaikkia kartan vesialueita. Käytän hyväkseni jo aikaisemmin luotuja graafisia malleja, joilla korostan vedelle ominaisia piirteitä kuten heijastumista, valon taittumista ja läpinäkyvyyttä. Käytän tässä toteutuksessa vain yhtä varjostinta, mutta käytän sitä kaksi kertaa hieman toisistaan poikkeavalla tavalla.

Ensimmäisenä loin suuren koko maastoalueen kokoisen neliönmuotoisen pinnan, joka koostuu useasta pienemmästä neliöstä (*tile*). Seuraavaksi renderöin koko luomani maiseman käyttäen heijastusvarjostinta (*reflection shader*). Tässä tapauksessa käytän kuitenkin käänteistä leikepintaa (*inverted clip plane*), koska haluan heijastumista varten renderöidä ainoastaan sen alueen, joka rajoittuu veden alapuolelle. Tämä tehdään ainoastaan sitä varten, että voin luoda veden pintaa varten projektiivisen tekstuurin kameran näkymän perusteella. Tämän lisäksi häiritsen tätä luotua tekstuuria vielä normaalikartalla (*normal map*), joka lisää vielä tekstuurin pintaan eräänlaisen vettä kuvaavan aaltotehosteen. Kuvassa 13 on käyttämäni normaalikartta.



Kuva 13. Normaalikartta (*normal map*).

Juuri luodulla tekstuurilla voimme päällystää vesialueen usealla eri tesktuurikerroksella. Esimerkiksi ensin voimme päällystää koko vesialueen 100:lla rinnakkain olevalla tekstuurilla, mutta sen lisäksi lisäämme vielä toisen kerroksen, jossa näitä rinnakkain olevia tekstuureita onkin vain 25. Näin saadaan tehtyä veden pinta, joka havainnollistaa kaksi erilaista aaltokokoa, ja näitä tekstuureja vierittämällä pystymme kuvaamaan liikkuvia laineita. Sävytin tätä normaalikartalla tehtyä kaksoistekstuuria vielä hieman sinisellä värillä, jolla on tarkoitus erottaa vesi paremmin muusta maisemasta. Kuva 14 esittää tällä tavoin luotua veden pintaa.



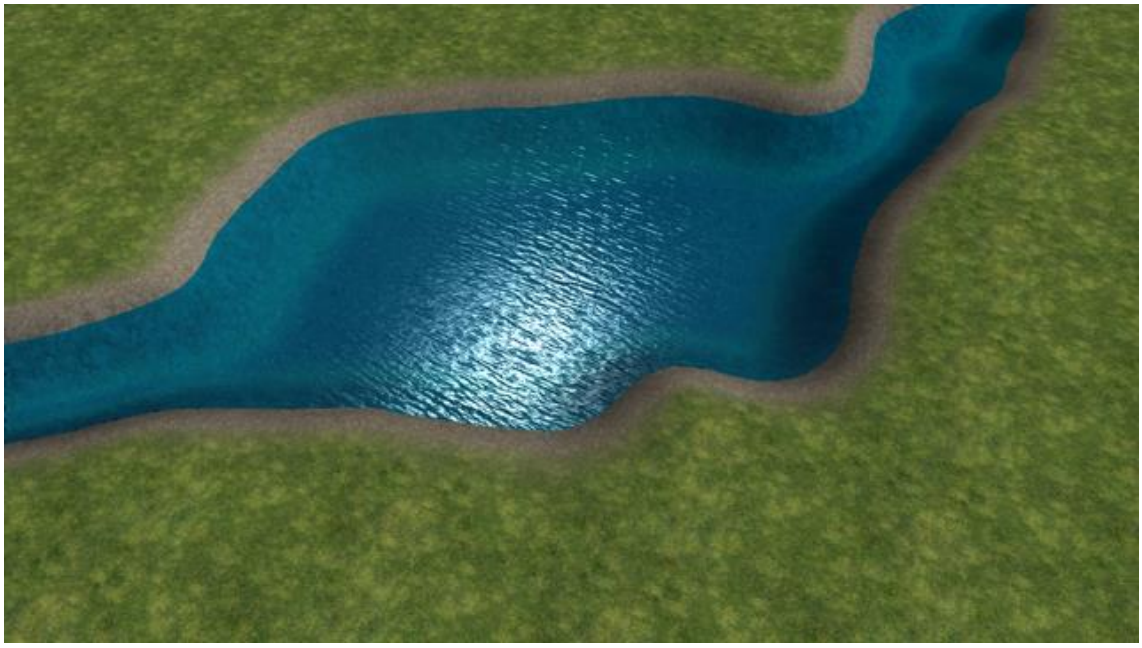
Kuva 14. Taittumistekstuuri (*refraction map*).

Monessa tapauksessa pelimoottoreissa veden toteutus rajoittuu vain siihen, mitä tässä esitin. Syynä tähän on se, että veden mallintaminen vaatii useita eri lähes koko maailman käsittäviä renderöintikertoja, ja näin ohjelmasta tulee helposti kohtuuttomankin raskas. Tämäkin aikaisemmin toteutettu veden mallinnus vaatii kaksi erillistä renderöintikertaa, ja näin ollen laskee kuvaruudun päivitysnopeuden noin puoleen aikaisemmasta. Oma luonteeni ei kuitenkaan näissä tilanteissa anna anteeksi sitä, jos jätän työn tekemisen vain puolitiehen. Siksi lisään veteen vielä kaksi haluamaani tehostetta.

Seuraavaksi teen heijastusta kuvaavan tekstuurin veden pinnalle. Tämä on hyvin samanlainen operaatio kuin aikaisemmin kertomani tekstuurin luonti normaalikartan kanssa. Tässä tilanteessa käytän myös käänteistä leikepintaa (*inverted clip plane*), ja sillä rajoitan tekstuuria varten käytettävän alueen ainoastaan veden pinnan yläpuolelle. Viimeistelen tekstuurin taas normaalikartan avulla, jolla saadaan veden pintaan aaltoja mallintavaa väreilyä. Kuvassa 15 voidaan nähdä, miltä veden pinta näyttää ainoastaan heijastustekstuuria käyttämällä.

Kuva 15. Heijastustekstuuri (*reflection map*).

Viimeistelen kokonaisuuden lisäämällä veden pintaan suoraan paistavan auringon valoa esittävän valotehosteen. Tämä antaa vedelle peilimäisen pinnan (specular reflection) ja korostaa aikaisemmin luotujen aaltojen erottuvuutta. Käytän tässäkin tapauksessa kaksinkertaista normaalikarttaa, joka esittää kahdesta erilaisesta aaltokoosta syntyvää väreilyä. Tämä tekniikka lisäsi taas yhden renderöintikerran ohjelman runkoon, joten kuvanruudun päivitysnopeus on nyt noin kolmasosan siitä, mitä se oli ennen veden luomista pelimoottoriin. Kuvasta 16 voidaan nähdä viimeistelty veden pinta.



Kuva 16. Veden pinta, kun kaikki mainitut tekniikat ovat yhdistettynä.

10 Lopuksi

Opinnäytetyön tavoite saavutettiin, mutta vain vähän yli puolet mahtui koko pelimoottorin toiminnasta tähän opinnäytetyöhön. Henkilökohtaisesti olisin halunnut vielä käsitellä varjojen muodostamista, 3D-mallien instantiointia, karttojen luontia, hiukkasia (particle systems) ja törmäyksiä. Myös erilaisia tekstuurien sekoituskeinoja ja reunanpehmennyksiä ei opinnäytetyössä käsitellä ollenkaan.

Työmäärä etenkin ohjelmakoodin osalta oli valtava. Opinnäytetyön valmistumishetkellä itse ohjelmalistauksien määrä on yli 20 000 riviä. Kirjallisuutta koskien DirectX 11 -rajapintaa on saatavilla hyvin rajoitetusti, ja hyvin paljon materiaalia jouduin käyttämään DirectX:n omasta dokumentoinnista. Tämä johtuu enimmäkseen siitä, että lokakuussa 2012 julkaistun Windows 8 -käyttöjärjelmän myötä Microsoft sisällytti DirectX:n suoraan Windowsin oman SDK:n sisään. Vaikkakin uusimmat versiot DirectX:stä ovat tällä hetkellä 11.1 (Windows 8) ja 11.2 (Windows 8.1), niin kirjailijat edelleen käyttävät uusissa kirjoissaan DirectX 11 -nimeä. Tämä mielestäni virheellinen tapa kuvata aihetta käsitteleviä kirjoja ja näin se hämää tuotteen loppukäyttäjää. Itse käyttämäni DirectX 11.0 on kuitenkin se viimeisin DirectX niille ohjelmoijille, jotka eivät halua myydä omia tuotteitaan Microsoftin omassa verkkokaupassa.

Opinnäytetyö teki välttämättömäksi sen, että DirectX:ää käsittelevää materiaalia täytyi opiskella aikaisempaa syvällisemmin. Sain paljon vastauksia mieltä vaivanneisiin kysymyksiin, ja näin tämä kehitti minua eteenpäin ohjelmoijana. Jatkossa aion kehittää tekemääni pelimoottoria eteenpäin, ja päämääränäni olisi tuottaa sen avulla vielä pelejä. Tämä on se päällimmäinen syy miksi lähdekoodi ei ole avoin, ja siksi se ei myöskään tule opinnäytetyön liitteeksi.

Lähteet

1. Jason Gregory, Jeff Lander & Matt Whiting. 2009. Game Engine Architecture. Boca Raton: Taylor and Francis Group.

2. Bjarne Stroustrup. 1994. The Design and Evolution of C++. New York: Addison-Wesley Professional.

3. 35.060: Languages used in information technology. 2014. Verkkodokumentti. International Organization for Standardization. http://www.iso.org/iso/home/store/catalogue_ics/catalogue_ics_browse.htm?ICS1=35&ICS2=060. Luettu 23.4.2014.

4. Welcome Back to C++ (Modern C++). 2014. Verkkodokumentti. Microsoft. http://msdn.microsoft.com/en-us/library/hh279654.aspx. Luettu 23.4.2014.

5. Ivor Horton. 2008. Ivor Horton's Beginning Visual C++. Indianapolis: Wiley Publishing, Inc.

6. GDI+. 2012. Verkkodokumentti. Microsoft. http://msdn.microsoft.com/en-us/library/windows/desktop/ms533798%28v=vs.85%29.aspx. Luettu 23.4.2014.

7. DirectX. 2014. Verkkodokumentti. Computer Hope. http://www.computerhope.com/jargon/d/directx.htm. Päivitetty 28.3.2014. Luettu 23.4.2024.

8. Windows DirectX Graphics. 2010. DirectX SDK (June 2010). Microsoft.

9. DirectX Software Development Kit. 2010. DirectX SDK (June 2010). Microsoft.

10. Frank D. Luna. 2008. Introduction to 3D Game Programming with-DirectX 10. Sudbury: Wordware Publishing, Inc.

Laitteistolista alustoista, joilla ohjelmakoodia on testattu

Suoritin	Muistinmäärä	Grafiikkaohjain	Käyttöjärjestelmä
Intel i7 4970X	Team Group 16GB (2600Mhz C10)	3x EVGA GTX680 2GB (Tri-SLI)	Windows 7 Pro 64bit
Intel i7 2600K	Team Group 16GB (2600Mhz C10)	3x EVGA GTX680 2GB (Tri-SLI)	Windows 8 Pro 64bit
Intel i5 3570K	G.Skill 8GB (2400MHz C11)	EVGA GTX 680 2GB	Windows 8.1 Pro 64bit
Intel i7 930	Corsair 6GB (1600MHz C8)	Sapphire Radeon 6850 1GB	Windows 7 Pro 64bit

Laitteistovaatimukset

Vähimmäisvaatimus

- Käyttöjärjestelmä: Windows 7, Windows 8, Windows 8.1, Windows RT tai Windows Server 2008 SP2, Windows Server 2012 R2
- Suoritin: 2 GHz moniydinsuoritin
- Muisti: 2 Gt
- Vapaata kiintolevytilaa: 200Mt
- Näytönohjain: DirectX 11.0 yhteensopiva, muistia 1024Mt
- Hiiri ja näppäimistö

Suositus

- Käyttöjärjestelmä: Windows 7, Windows 8, Windows 8.1, Windows RT tai Windows Server 2008 SP2, Windows Server 2012 R2
- Suoritin: Core 2 Duo 2.4 GHz, Athlon X2 2.7 GHz tai suurempi
- Muisti: 4Gt
- Vapaata kiintolevytilaa: 200Mt
- Näytönohjain: DirectX 11.0 yhteensopiva, muistia 2048Mt
- Hiiri ja näppäimistö