LAHDEN AMMATTIKORKEAKOULU
Lahti University of Applied Sciences

DIRECT3D HLSL-SHADERIT

LAHDEN
AMMATTIKORKEAKOULU
Tekniikan ala
Mediatekniikka
Tekninen visualisointi
Opinnäytetyö
Kevät 2012
Vesa Viljanen

Lahden ammattikorkeakoulu
Mediatekniikka

VILJANEN, VESA: Direct3D HLSL-shaderit

Teknisen visualisoinnin opinnäytetyö, 77 sivua, 3 liitesivua

Kevät 2012

TIIVISTELMÄ

Opinnäytetyön tavoitteena oli syventyä Direct3D:n HLSL shaderien teoriaan ja oppia luomaan niiden avulla eri käyttökohteisiin soveltuvia shader-ohjelmia. Teoreettisen osuuden luonnissa oli hyödynnetty aiempaa kokemusta, internetistä löytyviä sähköisiä lähteitä ja alan kirjallisuutta (mm. ShaderX series). Alun perin työn case-osuuden tavoitteena oli luoda käyttökokemukseltaan mahdollisimman todentuntuinen kaivinkonesimulaattori ja perehtyä yleisesti simulaattorien kehittämiseen. Shaderit ottivat isoimman osan aiheesta ja simulaattorille ominainen fysikaalinen todellisuus ja realistiset yksityiskohdat saivat jäädä vähemmälle.

Internetistä löytyvien lähteiden perusteella pystyi rakentamaan kuvan näytönohjainten shader-arkkitehtuurista ja ohjelmoinnillisista käsitteistä. Painettuna lähteenä ShaderX 7 -kirja auttoi ymmärtämään enemmän nykypäivänä yleisten tietokone-efektien tekniikoista. Kelvollisen kehitysympäristön shader prototyyppien kehittämiseen antoi Quest3d-ohjelma. Prototyyppien perusteella keksittiin useita tapoja hyödyntää shadereita oikeissa sovelluksissa ja casessa. Kynnys shaderien kehittämiseen madaltui lisääntyneen tiedon myötä.

Simulaattoriosuutta varten haastatelttiin muutamia henkilöitä, joilla on usean vuoden kokemus maanrakennustöistä. Lisäksi opinnäytetyöhön kuvattiin referenssivideoita kaivinkoneen liikeradoista Porvoon ja Sipoon rajalla sijaitsevalla Anttilan sähköaseman laajennustyömaalla. Paikalla ollessa ja videoiden perusteella käsitys kaivinkoneen liikkeistä, ulkoasusta ja ns. olemuksesta tarkentui.

Simulaattorin tavoite oli saavuttaa mahdollisimman todentuntuinen ajokokemus, joten ohjainten rakentaminen tuli tarpeeseen. Ohjainten rakentamiseen kului noin viikko, jonka aikana ratti- ja poljinyhdistelmästä muokkautui kaivinkoneen ajo-ohjaimet ja ohjainlaatikko (ns. kojelauta). Samalla kaivuuohjainten rakentaminen toteutettiin lennokkisimulaattorin ohjaimen pohjalta. Ohjaimia rakentaessa oppi tuntemaan peliohjainten rakennetta paremmin. Jatkossa peliohjainten piirejä olisi mahdollista hyödyntää erilaisissa kokeiluissa fyysisenä käyttöliittymänä tietokoneeseen.

Asiasanat: HLSL, Direct3D, shader, verteksi, pikseli, renderöinti, grafiikkaohjelmointi

Lahti University of Applied Sciences
Degree Programme in Media Technology

VILJANEN, VESA: Direct3D HLSL shaders

Bachelor's Thesis in Visualisation Engineering, 77 pages, 3 pages of appendices

Spring 2012

ABSTRACT

The objective of this study was to examine Direct3D HLSL graphic shaders. The main focus was on the theory of shaders and developing new ones from scratch. The theoretical part is based on the writer's earlier experience, internet and literature (e.g. ShaderX series). At first the objective was to create an excavator simulator with realistic operating experience, which is physically as realistic as possible, but shaders took a bigger part of the topic.

Internet was the greatest source for getting an overview of shader graphics pipeline, graphics card architecture and programming concepts. One book of the ShaderX series (ShaderX 7) gave a new perspective for developing general computer generated effects. The Quest3d program provided a flexible developing environment, which gave a good base for shader prototyping. Based on the prototypes, various ways were invented to utilize shaders in real applications and the case part of this thesis. Also the threshold for developing shaders lowered, because of increased knowledge.

For the simulator a couple of people were interviewed, who had many years of experience of earthmoving. Reference videos, which showed the movements of excavator parts, were recorded on a construction site. The excursions and videos provided a better perception of the motions and layout of the excavator.

The objective of the excavator simulator was to reach as realistic as operating experience as possible. Wheel and pedal gaming controls were modified to the excavator driving controls and controlbox (i.e. dashboard). At the same time digging controls were made. Making and modifying the controls taught technical structures of gaming controls. In the future it could be possible to use game control circuits as an interface to a computer.

Key words: HLSL, Direct3D, shader, vertex, pixel, rendering, graphics programming

SISÄLLYS

LYHENNE- JA TERMILUETTELO

CPU, Central Processing Unit	Keskusyksikön prosessori
D3D	Direct3D-rajapinta (API)
DX 9	DirectX 9 -rajapinta (API)
DOF	Depth of Field -efekti (syväterävyys)
GLSL	OpenGL Shader Language
GPGPU	General Purposes on GPU
GPU, Graphics Processing Unit	Näytönohjaimen prosessori
Grafiikka-pipeline	3d-grafiikan vaiheittainen laskentaprosessi
HLSL	High Level Shader Language
Mesh-malli	Pintarakenne koostuu kolmioista tai neliöistä
MRT	Multiple Render Targets
Shader	3d-grafiikan renderöinnin määrittävä ohjelma
SM X.0	ShaderModel X.0
T&L	Transform & Lighting
Verteksi	Tietoa sisältävä piste 3d-avaruudessa
VS, PS, GS	Vertex-, Pixel-, Geometry Shader
VTF	Vertex Texture Fetch

Digitaalisten materiaalien termejä

Diffuse	Materiaalin väri
Environment map	Heijastuvaa ympäristöä jäljittelevä kartta
Normal(map)	Normaalivektori(kartta)
Reflection	Heijastus
Specular	Kiilto

Muita lyhenteitä ja termejä

A/D-piiri	Analog-to-Digital (sähkökomponentti)

1 JOHDANTO

Opinnäytetyö käsittelee Direct3D shadereitä teorian tasolla ja ulkoasullisista, sekä teknisistä hyödynnettävyys näkökulmista. Shadereiden teoreettista puolta tutkitaan esimerkein, havainnollistavien kaavioiden ja kuvien avulla. Lisäksi ohjelmoinnillisia elementtejä ja termejä on pyritty nostamaan esiin ja menemään aiheen pintaa syvemmälle. Keskeisenä osana on Direct3D 9 ShaderModel 3.0 shaderit.

Tekstissä on otettu huomioon Direct3D 9:n vanhanaikaisuus, mutta kuitenkin sen tehokkuus nykypäivänä. Suurin syy Direct3D 9:n käyttöön lienee se, että se kattaa toiminnallisuudellaan suurimman osan olemassa olevista PC-järjestelmistä.

Case-osuudessa käydään läpi kaivinkonesimulaattorin luontiin liittyviä vaiheita ja yksityiskohtia. Koko case-osuuden sisältö ei liity suoraan shadereiden toteutukseen, mutta pyrkii antamaan laajempaa kuvaa käyttökohteessaan. Case osuus on toteutettu Quest3d-ohjelman 4-versiota hyödyntäen. Quest3d 4 tarjoaa tuen Direct3D 9:lle, joten siinä voi toteuttaa ainoastaan HLSL ShaderModel 1.0 – 3.0 shadereitä. Kuvassa 1 on ruutukaappaus caseen luodusta kaivinkonesimulaattorista.

Useimmat tekstin kohdat ovat hyvin käytännön läheisiä, joten lukijalla olisi hyvä olla aiempaa tietämystä ohjelmoinnista, 3d-mallintamisesta ja kuvankäsittelystä ymmärtääkseen shadereiden toiminta- ja tuottamisprosesseja.

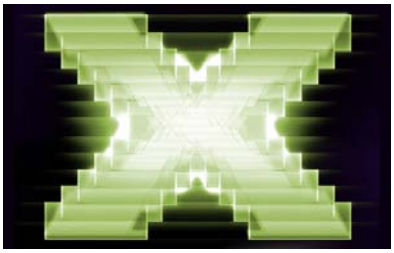
KUVA 1. Ruutukaappaus caseen luodusta simulaattorista (Viljanen 2012)

2 GRAFIIKKAOHJELMOINTI SHADEREILLÄ

2.1 Reaaliaikaiset GPU shaderit käsitteenä

Shaderit ovat ohjelmia, joilla ohjataan pääasiassa 3d-grafiikan, mutta myös 2d-grafiikan piirtoa ruudulla nähtäväksi kuvaksi. Tällaista tietokoneen tuottamaa piirtoa tai kuvan generointia kutsutaan renderöinniksi. Keskusyksikön prosessorilla toteutettua renderöintiä kutsutaan CPU-renderöinniksi ja näytönohjaimen prosessorilla GPU-renderöinniksi. Aiemmin GPU-renderöinti liitettiin suoraan reealiaikaiseen (real-time) kuvan laskentaan, mutta nykyään se voi merkitä still-kuvien tai animaatioiden ns. offline renderöintiäkin GPU:lla kiihdytettynä. Opinnäytetyö käsittelee reaaliaikaista GPU-renderöintiä Direct3D 9 shadereiden kautta. (Greenheck 2011.)

Shadereita voi kirjoittaa korkeamman tason ohjelmointikielillä tai konekielellä eli Assemblyllä. Korkeamman tason GPU shader -kieliä ovat HLSL (High Level Shader Language), GLSL (OpenGL Shader Language) ja Cg (C for graphics). Shader-ohjelmat eivät ole itsenäisiä vaan vaativat aina rajapinnan, jonka avulla ne suoritetaan. Varteenotettavia näytönohjaimelle suunniteltuja rajapintoja (API) on olemassa kaksi, jotka ovat Direct3D ja OpenGL. Direct3D API on osa DirectX API-kokoelmaa, johon kuuluu useita erilaisia multimediakirjastoja. Direct3D ja OpenGL ovat ns. HAL (Hardware Abstaraction Layer) -periaatteella toimivia, jotka hyödyntävät tarkoitukseensa kohdistetun laitteen resursseja. DirectX:n voi tunnistaa esimerkiksi kuvassa 2 esiintyvästä vihreästä X-logosta, joka kulkee usein D3D-rajapintaa käyttävien sovellusten ja näytönohjainten mukana. (Conitec Datasystems 2011; Microsoft 2012a.)



KUVA 2. DirectX -logo (ExtremeTech 2011)

Termi ”shader” on peräisin Pixarin vuonna 1989 alulle laittamasta RenderMan-ohjelmasta. RenderMan julkaistiin 1995 julkiseen käyttöön, jonka yhteydessä shader-termi vasta vakiintui. Nykyisin termi on vakiintunut ja laajentunut jonkin verran varjostus (shading) -merkityksen ulkopuolelle koskien laaja-alaisempaa laskentaa GPU:lla. Lisäksi shader-termi esiintyy esimerkiksi MentalRay-renderöintiohjelman yhteydessä. (Wikipedia 2012e.)

2.2 GPU-renderöinti ennen shadereita

Ohjelmoitavat shaderit tulivat näytönohjaimiin marraskuussa vuonna 2000 Direct3D 8 julkaisun yhteydessä. Ennen Direct3D 8:aa näytönohjaimet olivat kiinteästi toimivia lisälaitteita, joiden tehtävä oli ainoastaan kaavamaisesti renderöidä 3d-grafiikkaa reaaliajassa.

Aluksi GPU-renderöinnissä oli yleisempää, että tietokoneen prosessorilla suoritettiin verteksi shaderiä vastaavat operaatiot ennen näytönohjainta eli ns. Software T&L. Näytönohjain suoritti sitten grafiikan rasteroinnin ja yhdisti prosessoituun dataan mahdolliset tekstuurit. Muita hienouksia, kuten sumu oli näytönohjaimelta kiinteästi lisättävissä. Myöhemmin mukaan tulivat epätasaisuuksia määrittelevät bump map-kartat. Näytönohjainten kehittyessä suosiota alkoi saada rautalaskentainen (GPU) muunnos ja valaistus (Hardware Transform & Lighting, T&L), joka on nykyään verteksi shaderin myötä lähes itsestäänselvyys. (Wikipedia 2012f.)

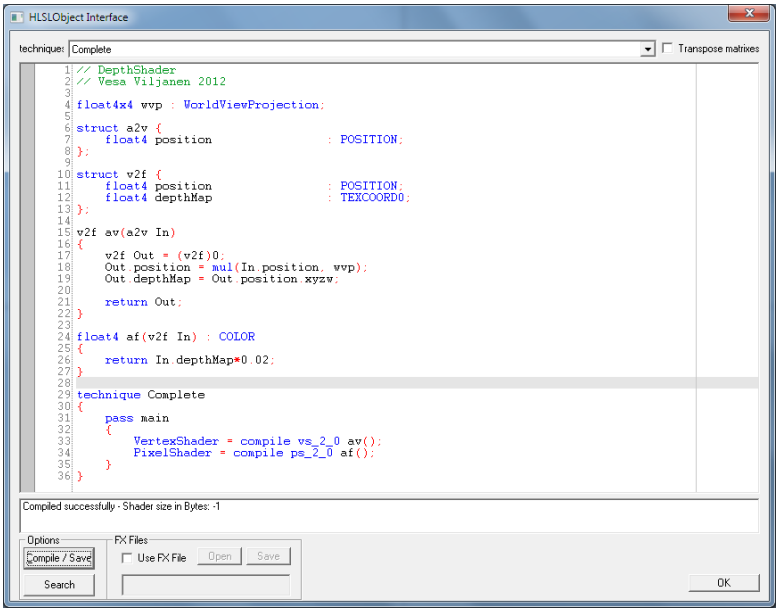
2.3 Ohjelmointikielet

Ohjelmointikielistä HLSL kuuluu Microsoftin Direct3D:hen, GLSL Khronosin OpenGL:ään ja Cg on Nvidian ja Microsoftin yhteistyössä kehittämä kieli. Cg:tä on kehitetty kieleksi, joka kääntyisi Direct3D:lle ja OpenGL:lle. (Wikipedia 2012e.)

- HLSL — High Level Shading Language (Shader Model)
- GLSL — OpenGL Shader Language
- Cg — C for graphics

Korkeamman tason kielet kääntyvät konekieleksi näytönohjaimen muistiin, josta niitä suoritetaan näytönohjaimen useassa ytimessä samanaikaisesti. Nykyiset näytönohjaimet kykenevät suorittamaan useita shadereitä tehokkaasti, koska niiden arkkitehtuuri perustuu rinnakkaislaskentaan. Nykyisissä peleissä ja muissa kehittyneemmissä PC:n tai pelikonsolien 3d-sovelluksissa piirto tapahtuu lähes poikkeuksetta shaderien avulla. (Samyn 2011.)

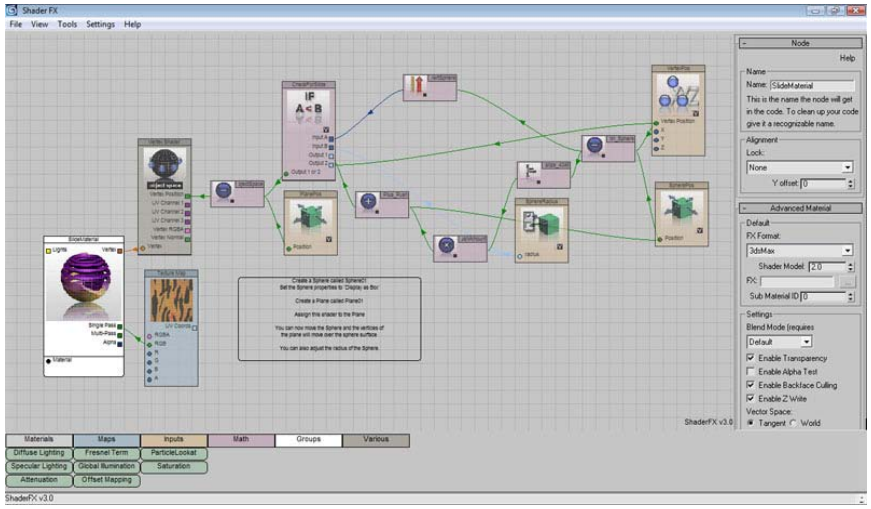
Esimerkiksi shaderohjelman kääntäminen Quest3d:ssä suoritetaan kuvassa 3 näkyvässä ikkunassa. Compile/Save painiketta painaessa tulee joko ilmoitus virheestä tai ilmoitus onnistuneesta käännöstä ja shaderin viemästä muistin määrästä.



KUVA 3. Shaderohjelman kääntäminen Quest3d:ssä (Viljanen 2012)

2.4 Node-pohjaiset editorit

Ohjelmoinnin lisäksi käytetään jonkin verran editoreita, joiden käyttöliittymä on graafinen ja ns. node-pohjainen. Kuvassa 4 on esitettynä tällainen käyttöliittymä ja siinä näkyvillä ”naruilla” tehdään kytköksiä eri operaatioiden välillä. Editori voi tuottaa halutun ohjelmointikielen koodia visuaalisten kytkentöjen perusteella. Tämänkaltaisia editoreja on mm. ShaderFX plugin ja Unreal Enginestä löytyvä oma materiaali editori. (Lumonix 2012.)



KUVA 4. ShaderFX-käyttöliittymä. (Lumonix 2012)

2.5 HLSL Shader Model -formaatin versiot

Lisäksi Direct3D:n HLSL shaderit on jaettu eri formaatteihin. Formaatteja kutsutaan ShaderModel-termillä, jonka perään on liitetty version numero. Kuvassa 5 on lueteltuna Shader Model -formaatit ja shader-profiilit (Shader Profile). (Microsoft 2012b.)

Shader Model	Shader Profiles
Shader Model 1	vs_1_1
Shader Model 2	ps_2_0, ps_2_x, vs_2_0, vs_2_x
Shader Model 3	ps_3_0, vs_3_0
Shader Model 4	gs_4_0, ps_4_0, vs_4_0, gs_4_1, ps_4_1, vs_4_1
Shader Model 5	cs_4_0, cs_4_1, cs_5_0, ds_5_0, gs_4_0*, gs_4_1*, gs_5_0, hs_5_0, ps_4_0*, vs_4_0*, ps_4_1*, ps_5_0, vs_4_1*, vs_5_0

(* - gs_4_0, gs_4_1, ps_4_0, ps_4_1, vs_4_0 ja vs_4_1 oli esiteltynä shader model 4.0:ssa, kuitenkin, shader model 5 lisää tuen rakenteellisille puskureille (structured buffers) ja tavu osoitteiden puskureille (byte address buffers).)

KUVA 5. Shader Model vs Shader Profiles (Microsoft 2012g)

Direct3D 8 – 11 jakautuminen ShaderModel versioihin:

- Direct3D 8: SM 1.0
- Direct3D 9: SM 1.0 – 3.0
- Direct3D 10: SM 4.0
- Direct3D 10.1 SM 4.1
- Direct3D 11: SM 5.0
- Direct3D 11.1: Tulossa 2012 syksyllä (Windows 8 julkaisu).

2.6 Direct3D 9

Direct3D 9 ensimmäinen versio julkaistiin vuoden 2002 joulukuussa Windows XP:n yhteydessä. Sen jälkeen sitä on päivitetty kolmeen kertaan versioin, jotka ovat 9.0a (maaliskuu 2003), 9.0b (elokuu 2003) ja 9.0c (elokuu 2004). Vaikka D3D 9 ensimmäisen version julkaisusta tulee tänä vuonna 10 vuotta, se on edelleen suhteellisen nykyaikainen, tehokas ja varteenotettava vaihtoehto visuaalisesta vaatimustasosta riippuen. D3D 9 verteksi- ja pikseli-shaderit eivät eroa ohjelmoinnillisesti juurikaan uudempien versioiden vastaaviin. Mahdollisesti haittaavin rajoitus on se, että Direct3D 9 asettaa tiukempia rajoituksia näytönohjaimen muistin varaukseen kuin sitä uudemmat versiot. Lisäksi Direct3D 9 -sovellukset kattavat toimivuudellaan suuremman osan PC-järjestelmistä kuin nykyiset Direct3D 10, 10.1 ja 11. Direct3D 9:llä shaderien tuottaminen on suhteellisen yksinkertaista ja useimmat internetistä löytyvät esimerkit on toteutettu D3D 9 aikaan. Suurimmat eroavaisuudet ja rajoitteet D3D 9:n ja 11:n välillä ovat ns. geometria shader ja tesselaatio. Niiden hierarkkisen sijainnin pipeline-rakenteessa voi tulkita kuvasta 6. (Computer Hope 2012.)

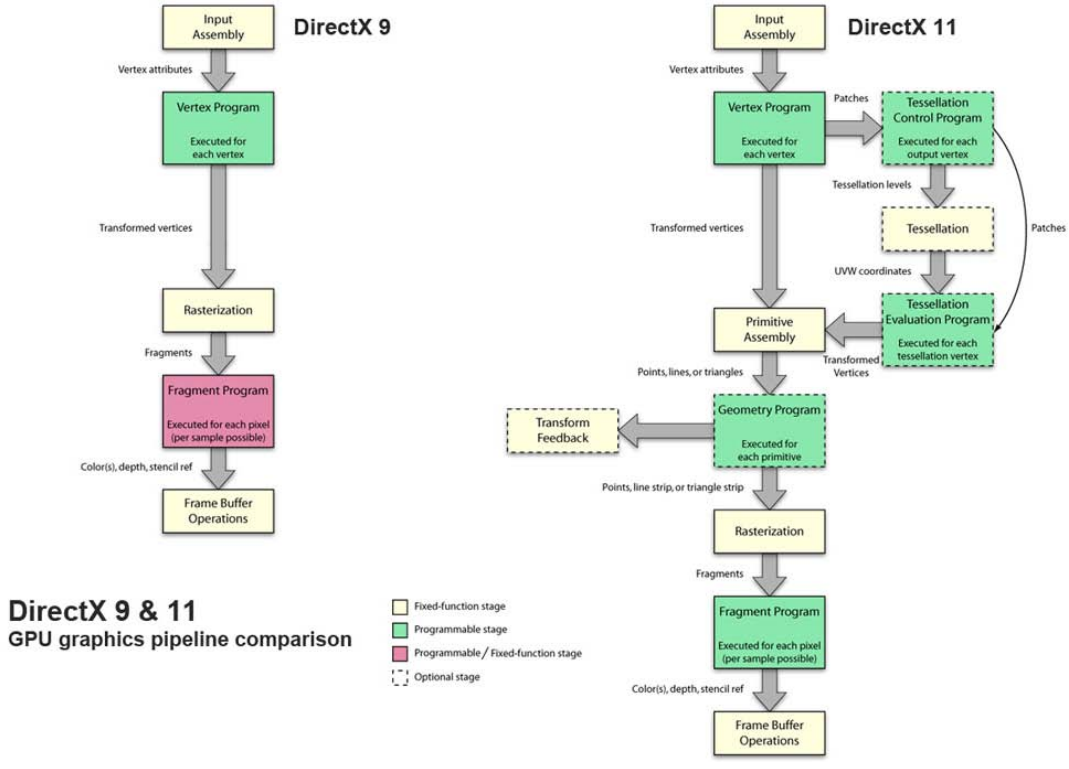
2.7 Direct3D 10, 10.1 ja 11

Direct3D 10 -näytönohjaimia on ollut saatavana Windows Vistan julkaisusta lähtien eli vuodesta 2006. D3D 10 toi mukanaan uuden ja merkittävän ominaisuuden, joka on geometria shader. Geometria shaderistä on kerrottu lisää luvun 3.2 alla. D3D 10 on lisäksi päivittynyt versioon 10.1 Windows Vistan Service Pack 1 päivityksen yhteydessä, jonka muutokset olivat pääasiassa suorituskykyyn liittyviä.

Direct3D 11 -näytönohjaimet julkaistiin Windows 7:n yhteydessä. Direct3D 11 verrattuna Direct3D 10:een suurin parannus on ollut tessalaatio. Tessalaatio kykenee tarkentamaan lennossa kolmioista koostuvaa 3d-mallin pintaa, pilkkoen kolmioiden reunat ja luoden uusia kolmioita kolmioiden sisään. Tessalaation lisänä voidaan käyttää displacement mappausta uusien muotojen esille tuomiseen. Tarkkuutta määrittelee tessalation level -arvo. Näin saadaan lowpoly-mallin kulmikkuutta vähennettyä reaaliajassa. Se on iso parannus verrattunna aiempiin Direct3D-versioihin, joissa ainoa mahdollisuus on tehdä geometriasta riittävän tarkka jo mallinnusohjelmassa. (Engel 2009, 46; Computer Hope 2012.)

2.8 DirectX 11.1

Nykyisten ennusteiden mukaan DirectX 11.1 julkaistaan syksyllä 2012 Windows 8:n yhteydessä. Tulevia parannuksia aiempiin nähden on WDDM 1.2, kehittyneempi integraatio Direct2D:lle, Direct3D:lle (11.1) ja DirectCompute ominaisuuksille. Lisäksi se sisältää osia XNA-kirjastosta, kuten DirectXMath:n, XAudio2:n ja XInput:n. Varsinaisen Direct3D 11.1 -rajapinnan uudistuksia ovat mm. isommat puskurimuistit ja kehittyneemmät sekoitustilat. (Wikipedia 2012b.)

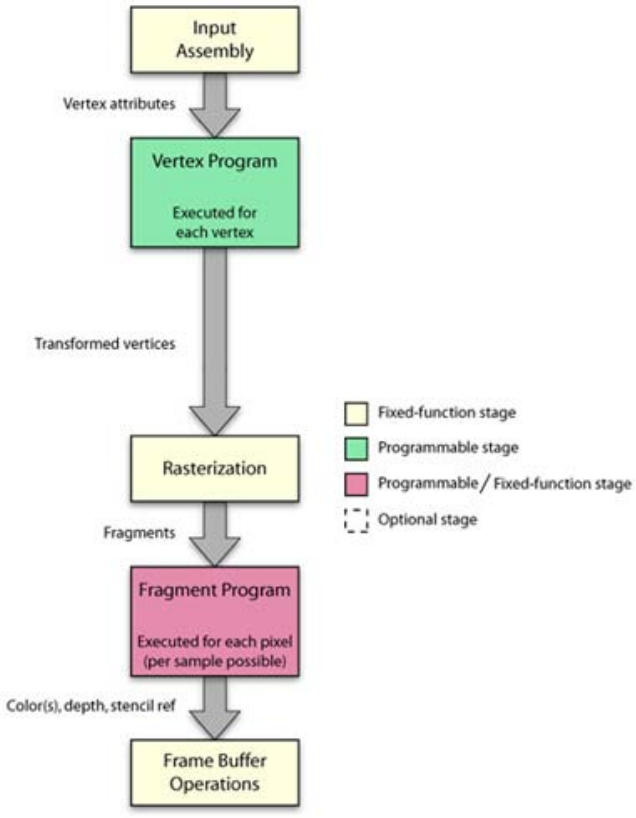


KUVA 6. DirectX 9 ja 11 vertailu (Wikipedia 2012; Viljanen 2012)

3 GPU SHADER PIPELINE

3.1 Renderöinti pipeline -käsitteenä

Pipeline-käsite on tärkeä, kun puhutaan näytönohjaimella renderöinnistä. Suomennettuna pipeline tarkoittaa putkistoa tai putkilinjaa, mutta se kuvastaa erityisen hyvin tapaa, kuinka shaderit suoritetaan GPU:lla. Pipeline-tyyppinen suoritustapa ja rajoittuneisuus mahdollistavat näytönohjaimelle ominaisen rinnakkaislaskennan nopeuden. Direct3D 9:n pipeline-rakenne on havainnollistettu kuvassa 7. (Microsoft 2012c; Microsoft 2012f.)



KUVA 7. DirectX 9 pipeline (Wikipedia 2012; Viljanen 2012)

Tietojen koostaminen (Input Assembler)

Tietokoneen keskusmuistissa sijaitseva Direct3D-sovellus lähtee käyntiin ja näytönohjaimella suoritettava data pitää koostaa prosessointia varten. Shaderohjelma suoritetaan näytönohjaimella, minkä jälkeen tietojen koostamisesta vastaa kuviossa 7 ensimmäisenä esitetty kohta Input Assembly. Siinä näytönohjain lukee käyt-

täjän määrittämiin puskureihin tallennettuja tietoja, esimerkiksi verteksipuskuria, josta saadaan 3d-mallin geometria näytönohjaimen muistiin. Muuta informaatiota voivat olla tekstuurit, muuttujat, vektorit ja matriisit. (Microsoft 2012d)

Kiinteätoiminen shader pipeline D3D9 (Fixed Function Pipeline, FFP)

Direct3D 9 sisältää jäänteitä aikaisemmista versioistaan, kuten mahdollisuuden käyttää edelleen kiinteäksi määriteltyä shader-rakennetta eli FFP:tä. FFP perustuu yleensä pelkkään verteksi shaderin ja tekstuureiden käyttöön. Esimerkiksi valaistus on aina verteksikohtainen (Gouraud varjostus) ja ei niin laadukas kuin se olisi toteutettu pikselikohtaisena (Phong varjostus) pikseli shaderillä. FFP:ssä pikseliohjelmaa ei ohjelmoida ollenkaan. Huomioitavaa on, että D3D 9 tukee tätä ainoastaan SM 1.0 – 2.0 versioissa. Ohjelmoitaessa on otettava huomioon etenkin verteksiohjelman muuttujien semantiikka, jotta rasterointi menee kiinteässä pikseli shaderissä oikein. Ohjelmoitavassa pikseli shaderissä jokaisen muuttujan semantiikan täsmällisyys ei ole niin tärkeää, mutta suotavaa. (Engel 2004a, 151.)

Pikseli shaderin muuttujat ja technique-määritelmä:

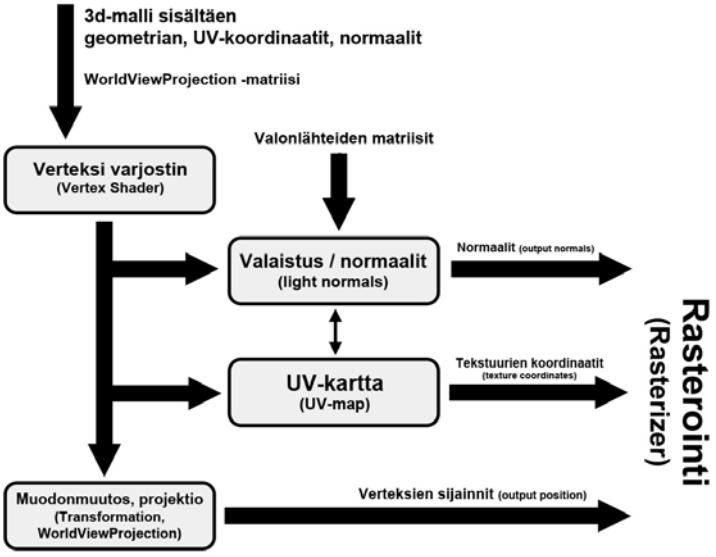
```
struct vertexToPixel {
        float4 Position             : POSITION;
        float3 Lighting             : COLOR0;
        float2 UV                   : TEXCOORD0;
};

technique FFP_shaderi
{
        pass P0
        {
                CullMode = None;
                VertexShader = compile vs_2_0 VShader();
        }
}
```

3.2 Verteksi shader (Vertex Shader)

Verteksi shader on pipeline-prosessin ensimmäinen ohjelmoitava vaihe. Sillä käsitellään geometrian jokainen verteksi. Verteksit ovat 3d-mallin kolmioista koostuvien pintojen kulmapisteitä. Verteksin kolmiulotteisen sijainnin lisäksi se voi sisältää informaatiota mm. normaaleista ja UV-karttojen koordinaateista. Verteksi shader suoritetaan kerran jokaista verteksiä kohden. Vaikka verteksi shaderillä käsitellään geometrista dataa, se ei mahdollista uuden geometrian luontia. Sitä varten on olemassa erillinen geometria shader (Direct3D 10.0 →). Kuva 8 havainnollistaa verteksi shaderin toiminnallisuutta osana pipeline kokonaisuutta. (Microsoft 2012h)

Yleisimmät verteksi shaderissä tehdyt operaatiot liittyvät ns. muodonmuutokseen (transformation). Muodonmuutoksen laskennassa käytetään usein erilaisia matriiseja, kuten sijaintiin tai projisointiin liittyviä. Koska informaatio on tietokoneella kolmiulotteista ja näytön pinta kaksiulotteinen, muodostetaan projisoitu vaikutelma kamerasta ja sen perspektiivistä verteksi shaderissä. Pääsääntöisesti 3d-mallin verteksit muunnetaan perspektiiviin ns. WorldViewProjection-matriisin avulla näytön kaksiulotteiselle pinnalle.

Tämän jälkeen muokatut verteksit siirtyvät eteenpäin, joka on rasterointi tai edellä mainittu geometria shader.



KUVA 8. Verteksi varjostin (Penfold 2002; Viljanen 2012)

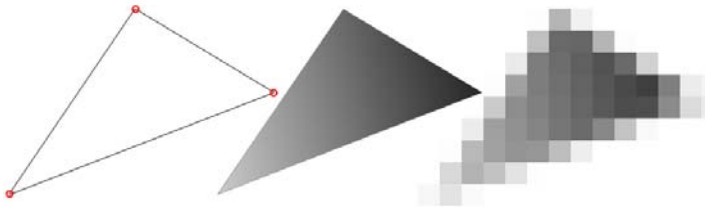
Geometria shader (Geometry shader, Direct3D 10.0 →)

Uudemmissa Direct3D 10.0:n jälkeen tulleissa versioissa on mahdollisuus kirjoittaa geometria shadereitä. Geometria shader antaa mahdollisuuden luoda verteksi shaderiltä saatuun geometriaan uutta tai tuhota vanhaa. Geometria shader sijoittuu kuvan 6 pipeline kaaviossa erikoisesti verteksi shaderin jälkeen, josta uusi geometria palautetaan takaisin verteksi shaderille ennen rasterointia. Yleisimpiä käyttökohteita lienee luoda ruohoa, karvaa tai dynaamisia 3d-objekteja. Muita yleisiä käyttökohteita on simuloida tilojen volumetrisia efektejä, kuten savuja, sumuja, pölyjä ja niiden kanssa reagoivaa valaistusta tai varjoja. (Wikipedia 2012a.)

3.3 Rasterointi (Rasterizer)

Rasterointi muuntaa verteksien väleihin muodostetut kolmiopinnat pikseleiden kaltaisiksi pisteinformaatioiksi (per-pixel data). Toisin sanoen rasteroinnissa on kyse verteksien välille jäävien kolmioiden täyttämisestä. Täytössä pikselit tai fragmentit määräytyvät interpoloimalla verteksien välit niihin kytköksissä olevilla tiedoilla. Useimmiten näitä tietoja ovat sijainnit, normaalit ja uv-koordinaatit. Rasteroijan tehtäviä on lisäksi siivota näkymättömät pinnat (culling ja clipping) ja asettaa syvyys- tai stensiilipuskuri (depth buffer, stencil buffer). Culling operaatioita ovat mm. kamerasta poispäin suuntautuvat pinnat ja erittäin pienet pinnat ja clipping operaatioita erittäin kaukana olevat pinnat ja kuva-alueen ulkopuolelle jäävät pinnat. Syvyys- ja stensiilipuskureita käytetään ns. maalarin algoritmin ratkonnassa eli mikä pinta on minkäkin pinnan edessä. (Myllymaa 2003.)

Ohjelmakoodin Technique määritelmässä voidaan vaikuttaa mm. culling ominaisuuksiin, josta tarkemmin kohdassa 5.5. Tämän jälkeen pikselit tai fragmentit siirtyvät pikseli shaderille. Rasterointi on kuvan 9 kaltainen operaatio.



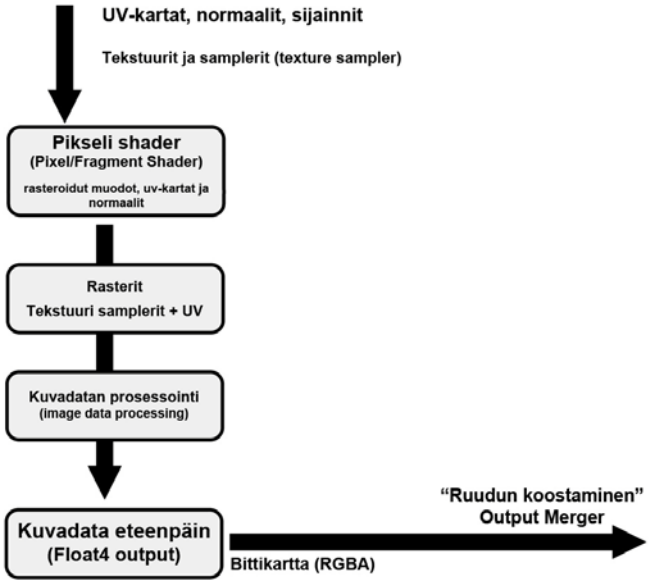
KUVA 9. Rasteroitu vektorikolmio (Viljanen 2012)

3.4 Pikseli shader (Pixel/Fragment Shader)

Pikseli shaderin pohjaksi saadaan kaksiulotteisiksi pikseleiksi rasteroidut 3d-objektit eli niistä muodostuneet siluettimaiset alueet. Siluetit eivät ole suoraan kelvollisia näkymään kolmiulotteisena grafiikkana, vaan usein pintoihin luodaan vaikutelma käyttämällä varjostavia tekniikoita. Varjostaminen onnistuu rasteroiduin normaalein, jota käytetään pikseli shaderissä. Pikseli shaderin ajatus on verrattavissa kuvankäsittelyyn.

Piksel shaderissä on paljon mahdollisuuksia toteuttaa luovia lopputuloksia ja sisällyttää todellisuutta jäljitteleviä efektejä. Näistä yleisimpinä mm. pikselikohtaiset normaalit (pinnan epätasaisuudet), specular map (kiillot) ja reflection map (heijastus). Usein kehitettävyyden kannalta on parempi sisäistää näiden logiikka kuin kopioida jo valmista. Se antaa paremman käsityksen siitä, miten tekniikkaa voi hyödyntää muissakin tilanteissa. (Microsoft 2012h.)

Lopputulos renderöidään, josta se päätyy viimeiseen ns. ruudun kokoamiseen (Output Merger). Kuva 10 havainnollistaa vaiheiden kulkua pikseli shaderin sisällä.

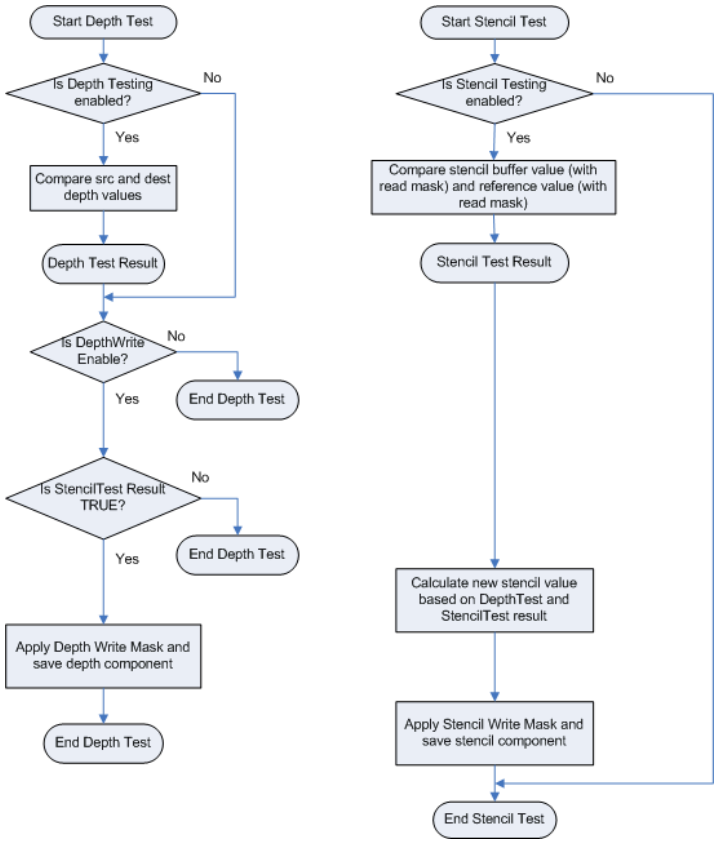


KUVA 10. Pikseli shaderin rakenne (Viljanen 2012)

3.5 Ruudun kokoaminen (Output Merger)

Prosessin viimeisenä vaiheena suoritetaan pikseli shaderissä käsiteltyjen rasterien kokoaminen, jossa ratkotaan pikseleiden päällekkäisyydet käyttämällä syvyyspuskuria (z-buffer) ja/tai stensiilipuskuria (stencil buffer). Näiden puskurien toiminnallisuutta on havainnollistettu kuvassa 11.

Kuvan kokoaminen toteutetaan taustapuskuriin (back buffer). Taustapuskuriin renderöity kuva voidaan lukea tekstuuriin (Render to Texture) tai siirtää ruutupuskurin kautta tietokoneen ruudulle. Seuraava kuva luodaan palaamalla pipelinen alkuun ja toistamalla vaiheet uudelleen. Kaikkiaan kuvia pitäisi kyetä renderöimään sekunnissa vähintään 60, että liike näyttäisi miellyttävältä. (Microsoft 2012e.)



KUVA 11. Direct3D 11 Output Merger vaiheen kaavio (Microsoft 2012b)

4 SHADER-OHJELMOINTI JA SYNTAKSIT

4.1 Ehtolauseet ja loopit

Kuten ohjelmointikielissä yleensäkin, myös shadereissä pätevät perinteiset prosessin kulkuun vaikuttavat loogiset operaatiot. Näillä operaatioilla voidaan asettaa ehtoja sekä suorittaa toistuvia ohjelman osia eli loopeja. (Engel 2004a, 42; Microsoft 2012b.)

Listassa käytössä olevat operaatiot:

- break
- continue
- discard
- do
- for
- if
- switch
- while.

4.2 Suorakulmainen koordinaatisto

Suorakulmainen koordinaatisto koostuu kolmesta ulottuvuudesta, joita merkitään kirjaimilla x, y ja z. Tässä koordinaatistossa ovat 3d-mallin verteksit, jossa jokaisen verteksin sijainti määrittyy kolmikomponenttisella vektorilla. Koordinaattien tarkkuus ja lukujen suuruus määräytyy muuttujan tyypistä. Toisenkaltainen tapa on olettaa datan olevan rajattu välille 0 – 1. Silloin vektoria kutsutaan yksikkövektoriksi. Esimerkiksi UV-kartan koordinaatit eroavat verteksien koordinaateista olemalla kaksikomponenttisina yksikkövektoreina. Kummassakin tapauksessa ohjelmakoodissa käytetään vektoreiden perässä vain x-, y- ja z-merkintöjä. (Microsoft 2012j.)

4.3 Pikselit ja tekselit shader-ohjelmassa

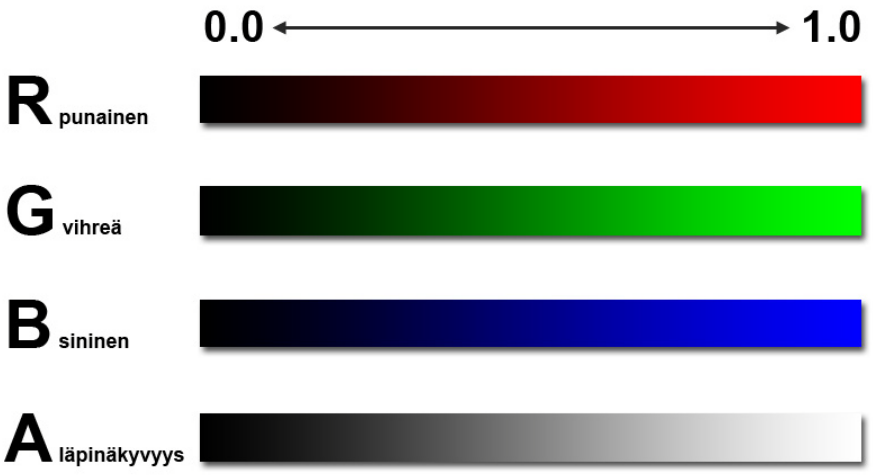
Pikseli shader on käsitteenä hieman harhaan johtava, koska sillä ei yleisesti käsitellä tiettyä pikseliä. Pikseleiden käsittely tapahtuu aina kokonaisvaltaisesti suorittamalla shaderohjelma jokaisessa näkyvässä kuvan kohdassa ts. fragmentissa.

Fragmenttien väriarvoihin voidaan vaikuttaa mm. tekstuureilla, jotka vaativat oman käsittelytekniikkansa. Tekstuureissa ohjelman on otettava huomioon näennäisesti vinot ja skaalautuneet pintojen pikselit. Nämä pikselit muuttuvat tekseleiksi eli tekstuurin pikseleiksi. Olennaisena osana tekseleitä ovat erilaiset suodatustekniikat (texture filtering), joilla vältetään laskostumien muodostumista ja rakeisuutta. Siksi tekstuureiden kanssa käytetään erityisiä samplereita (Texture Sampler). Tekselit ovat isossa osassa kuvan kokoa muuttaessa, jolloin suodatustekniikat tulevat tarpeeseen. Suodatustekniikoita ovat esimerkiksi lähimmän pisteen kanssa interpolointi, bilineaarinen, trilineaarinen ja anisotrooppinen suodatus. (Microsoft 2012i.)

4.4 Värikanavat

Yleisesti tietokoneiden värit tunnetaan 24-bittisinä tai 8 bittiä/kanava, sisältäen punaisen, vihreän ja sinisen värikanavan. 8 bittiä antaa kokonaislukuna mahdollisuuden 256 eri vaihtoehtoon välillä 0 - 255. Muita mahdollisia bittisyyksiä tekstuurille on 16 bittiä/kanava ja 32 bittiä/kanava, joissa tarkkuutta on huomattavasti enemmän. Ne olisi kokonaislukuina tulkiten huonoja, kun pikseleille tehdään laskutoimituksia, joiden lopputulos pitäisi jäädä näiden ääripäiden sisäpuolelle. Siksi värit tulkitaankin yksikkömuodossa 0.0 – 1.0 väliltä. Vaikka luvut ovat lähtökohtaisesti yksikössä, se ei estä ylittämästä tätä väliä, jos laskutoimitus on summaus, vähennys tai kertolasku luvulla yli 1.0:n tai jakolasku alle 1.0:n. Koska kanavan väri tulkitaan tavallisena liukulukumuuttujana, mahdollisuus negatiivisiin ja ylittäviin lukuihin on olemassa. Se ei ole haitta vaan mahdollisuus tuottaa dynaamisia, isoihin lukuihin perustuvia ja monimutkaisia laskukaavoja, joiden lopputulos tulisi olla esitettävällä 0.0 – 1.0 alueella. Koodissa voidaan suodattaa pois väriavaruuden väliin osumattomat luvut ts. rajata käyttämällä max() ja min() funktioita. Kuvassa 12 yksikkömuotoisuus on havainnollistettu eri kanavin. (Microsoft 2012j.)

Yhden värikanavan pitäminen näytönohjaimen muistissa rendattavan ruudun aikana onnistuu jotain muuttujaa tai vektoria käyttämällä. Eri bittisiä tekstuureita voidaan sekoittaa helposti keskenään, kunhan lopputulos on float-muotoinen liukuluku. (Microsoft 2012j.)



KUVA 12. Värit ja läpinäkyvyys yksikkömuodossa (Viljanen 2012)

4.5 Muuttujat

Muuttujatyyppejä on useampia, ja ne ovat tässä, kuten kaikessa muussakin ohjelmoinnissa, paikka, jossa säilytetään prosessoitavaa tietoa. Shadereissä se tarkoittaa yleensä jonkin erityisen lukuarvon ottamista keskusmuistista tai tiedon väliaikaista säilytystä. Shaderien muuttujissa ei säily tieto renderöityjen kuvien yli vaan jokainen muuttuja on aina alustettu ennen seuraavan kuvan renderöintiä. Usein sisältö on lukuarvo koordinaatistosta tai värikanavasta. DirectX 9:ssä muuttujia ei voida siirtää suoraan näytönohjaimelta takaisin prosessorille. Uudemmissa versioissa datavirran palauttamiseen on mahdollisuus. Yleisimpänä lukumuuttujista käytetään Float (32 bit, liukuluku) -muuttujaa. Selkeyden takia voi olla järkevää käyttää aina liukulukumuuttujaa. Muita Floatista merkittävästi eroavia muuttujatyyppejä ovat seuraavat: Bool (true/false), Int (32 bit, kokonaisluku), Uint (32 bit, etumerkitön/positiivinen kokonaisluku). (Engel 2004a, 9, 10; Microsoft 2012j.)

Lisäksi Float:n kaltaisia liukulukumuuttujia löytyy tarkkuudeltaan sen kummaltakin puolelta. Half on muuttuja, joka nimensä mukaisesti on puolet Floatista eli 16 bittiä ja kaksinkertainen Float eli Double on 64 bittiä.

Listassa ohjelmoitavat muuttujatyypit (huom. etumerkit pienellä):

- bool
- int
- uint
- half
- float
- double.

Esimerkiksi:

- bool kytkin = false;
- int laskuri = 0;
- float etaisyys = 0.0f;

Lisäksi muuttujiin pätee taulukkorakenne eli yhteen muuttujaan voi luoda useamman alkion useampaan ulottuvuuteen.

Esimerkiksi:

float kernel[4] = {2.0, 1.3, 5.2, 1.0};

4.6 Datatyypit

4.6.1 Vektorit

Jokaisesta muuttujatyypistä on olemassa vektorimuotoinen datatyyppi. Niiden vektorimuodot muistuttavat läheisesti rakenteita (struct). Jokaisen muuttujatyypin vektorin komponenttien lukumäärä voidaan valita seuraavasti:

- bool2, bool3, bool4
- int2, int3, int4
- uint2, uint3, uint4
- half2, half3, half4
- float2, float3, float4
- double2, double3, double4

Vektorien komponentteihin pääsee käsiksi kahdella eri tavalla. Vaikka tapoja on kaksi, se ei lisää monimutkaisuutta vaan usein erottaa vektorin sisällä olevan tiedon laadun ohjelmoitaessa. Ne syyt löytyvät koordinaatiston ja värikanavien tulkinnasta. Esimerkissä havainnollistettu kaksi eri tapaa:

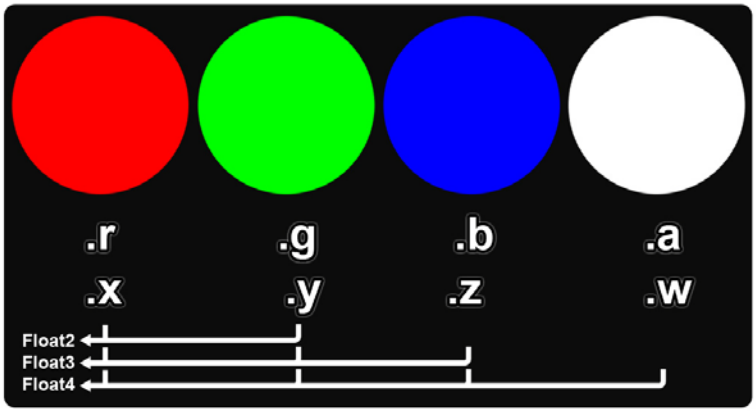
```
float4 varimuuttuja = float3(0.0, 0.0, 1.0, 0.5);
varimuuttuja.r; // 0.0
varimuuttuja.g; // 0.0
varimuuttuja.b; // 1.0
varimuuttuja.a; // 0.5
```

tai

```
float3 position : POSITION;
position.x;
position.y;
position.z;
float4 tapauksessa myös: position.w;
```

Kuten ensimmäisestä esimerkistä näkee, rgba tarkoittaa red, green, blue ja alpha eli punainen, vihreä, sininen ja läpinäkyvyysarvo.

Toisessa esimerkissä on havainnollistettu yleisesti koordinaatistossa käytetyt x-, y- ja z-suunnat sekä w. Varsinainen rgba- ja xyzw-tyyppisten merkintöjen hienous löytyy siitä, että niitä voidaan käyttää täysin ristiin tiedosta riippumatta. Kahdentyyppinen ratkaisu on grafiikkaohjelmoinnissa toimiva, koska sillä voidaan selkeyttää, minkä laatuista dataa missäkin vaiheessa vektori sisältää. Lisäksi geometrisena tulkittava informaatio muuttuu useassa tilanteessa väreiksi tavalla tai toisella. (Engel 2004a, 10, 11, 14-16; Microsoft 2012j.)



KUVA 13. Eri komponenttisten vektoreiden havainnollistus (Viljanen 2012)

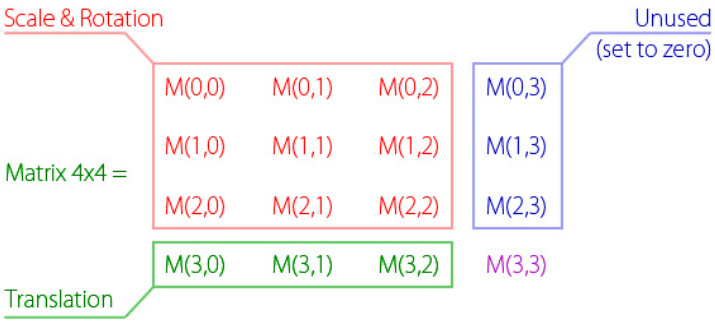
Vektoreille erityisiä laskutoimituksia ovat etenkin pistetulo dot() ja ristitulo cross(). Lisäksi vektorin pituus saadaan length()-funktiolla ja kahden vektorin välinen etäisyys dist()-funktiolla.

4.6.2 Matriisit

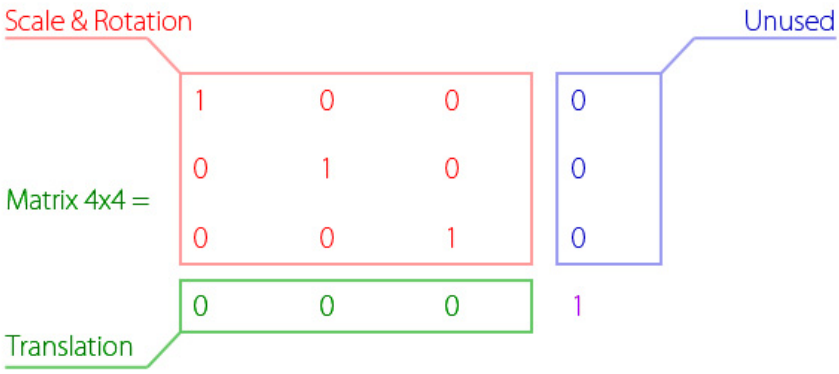
Kuten jo aiemmin mainitut muuttujat ja vektorit, niin hyvin samantyyppisiä ovat matriisit. Matriisit koostuvat vektorien komponenttien lukumäärästä potenssiin kaksi, mikä tavallaan lisää vektoreihin verrattuna toisen ulottuvuuden. Käytettävät matriisit ovat aina riveiltään ja sarakkeiltaan yhtä suuria: vähintään 2 riviä ja 2 saraketta ja enintään 4 riviä ja 4 saraketta. Samat muuttujatyypit toimivat tässäkin, mutta merkintätavat ovat seuraavat (Engel 2004a, 11, 12, 14-16; Microsoft 2012j.):

- bool2x2, bool3x3, bool4x4
- int2x2, int3x3, int4x4
- uint2x2, uint3x3, uint4x4
- half2x2, half3x3, half4x4
- float2x2, float3x3, float4x4
- double2x2, double3x3, double4x4.

Matriisien hyödyntäminen on hyvin olennaista kolmiulotteisen grafiikan piirtämisessä. Niiden avulla voidaan tehdä erilaisia verteksien sijaintien eli geometrisen muodon muunnoksia. Muunnos tarkoittaa yleensä usean matriisin ja vektorin keskinäistä kertolaskua, jolloin tuloksena on matriisiin nähden muuntunut vektori. Kuvassa 14 on havainnollistettu matriisin rakennetta ja kuvassa 15 matriisia, joka säilyttää sen kanssa kerrotun matriisin entisellään. Skaalaus on siis 1, 1, 1. (Koci 2010; Microsoft 2012j.)



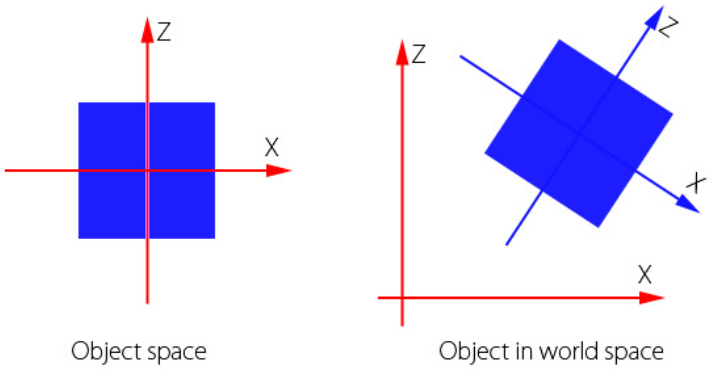
KUVA 14. 4x4 matriisi (Koci 2010)



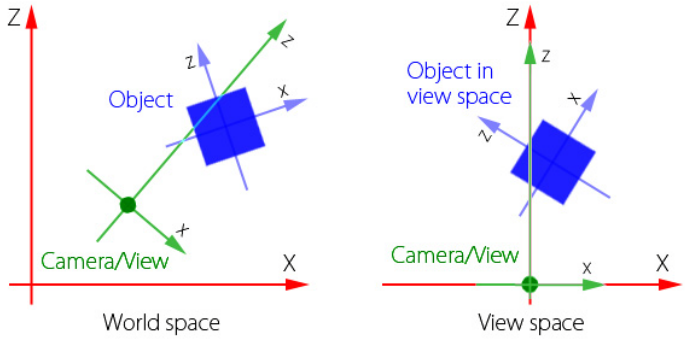
KUVA 15. Matriisi skaalassa 1:1 (Koci 2010)

Yleisin matriisi lienee kooltaan 4x4-kokoinen ja kyseisen matriisin kentät sisältävät sijainnin, rotaation ja skaalan, jotka on havainnollistettu kuvissa 14 ja 15. Matriisin tietoihin pääsee käsiksi kaksiulotteisen taulukon tapaan käyttämällä kahta hakasulun sisäistä indeksiä, esimerkiksi: muuttuja = matrix[0][2].

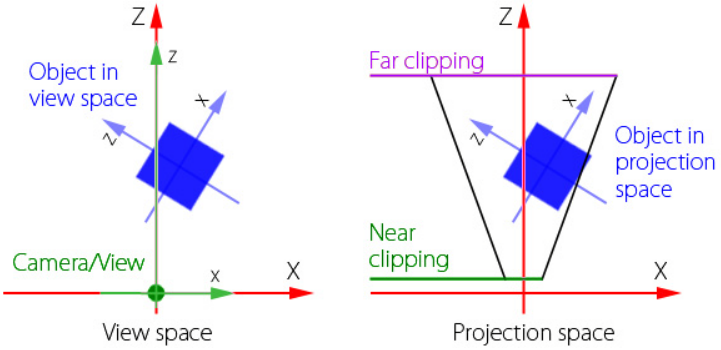
Kun 3d-objektin jokaisen verteksin sijaintivektori kerrotaan tämän kaltaisella matriisilla, lopputulos on matriisiin annettujen arvojen mukainen. Useampia matriiseja voi yhdistää keskenään, millä saadaan erilaisia muodon muutoksia aikaan. Sellainen voi olla mm. WorldViewProjection matriisi. WorldViewProjection matriisi projisoi verteksit kaksiulotteisena tietokoneen ruudulle. Kyseisen matriisin toiminnallisuus on havainnollistettu kuvissa 16 - 18. WorldViewProjection matriisin toteutus lasketaan nimensä mukaisessa järjestyksessä näin: World * View * Projection (Koci 2010; Microsoft 2012j.)



KUVA 16. Geometrinen objekti World matriisimuunnoksen jälkeen (Koci 2010)



KUVA 17. WorldView matriisi (Koci 2010)



KUVA 18. WorldViewProjection matriisi (Koci 2010)

4.6.3 Tekstuurit

Tekstuurit ovat eräs yleisimmistä datatyypeistä, jotka esitellään koodin alussa. Tekstuurin käsittely on kuitenkin vähän monimutkaisempaa kuin esimerkiksi tavallisen muuttujan. Se johtuu pääasiassa siitä, että näytönohjaimet on rakennettu laskemaan erilaisia kuvan suodatuksia ja interpolointeja, kuten mipmappausta ja laskostumien korjausta (anti-aliasing). Vaihetta kutsutaan ns. tekstuuri samplerin esittelyksi. (Engel 2004a, 17-19; Microsoft 2012j.)

Samplereita on useampaa vaihtoehtoa ja ne riippuvat osittain tekstuurin ulottuvuuksista. Esittelytyyppejä ovat mm. seuraavat:

- Texture1D (yksiulotteinen pikseliraita)
  - sampler1D
  - tex1D( );
- Texture2D (kaksiulotteinen pikselikartta)
  - sampler2D
  - tex2D( );

- TextureCube (kuutta pikselikarttaa käyttävää kuutioprojektiokartta)
  - samplerCube
  - texCube( );
- Texture3D (kolmiulotteinen vokselikartta)
  - sampler3D
  - tex3D( );

Samplerille voi määritellä lisäksi suodatukseen ja jatkuvuuteen liittyviä parametrejä. Seuraavassa esimerkissä tekstuurin esittely kokonaisuudessaan:

```
texture Color : TEXTURE0;
sampler2D ColorSampler = sampler_state
{
        Texture = <Color>;
        MinFilter = LINEAR;
        MagFilter = LINEAR;
        MipFilter = LINEAR;
        AddressU = Wrap;      // Tekstuuri jatkuu UV-kartan yli
        AddressV = Clamp;     // Tekstuuri ei jatku UV-kartan yli
};
```

Tekstuurin pikseleihin osoitetaan kaksikomponenttisen vektorin avulla eli yleensä UV-kartalla. Tekstuurin lukemisessa käytetään funktioita tex1D, tex2D, texCube ja tex3D joihin annetaan sampleri ja koordinaatti. Kaksiulotteisen tekstuurin lukeminen pikseliohjelmassa toteutetaan näin:

```
float4 Color = tex2D(ColorSampler, IN.UV.xy);
```

5 SHADER-OHJELMAN RAKENNE

5.1 Verteksiohjelma

Verteksiohjelma on funktio, joka shaderissä suoritetaan ensimmäisenä. Se on siis verteksi shaderin ohjelmoitava osuus. Kuten jo aiemmin on kerrottu, verteksiohjelma käsittelee 3d-mallin geometriaa verteksien tasolla, sisältäen verteksien sijainnin, normaalit ja UV-koordinaatit. (Engel 2004a, 17.)

Verteksiohjelmalle pitää ensimmäisenä määrittää rakenne (struct), johon määritetään muuttujat ja se mistä kyseiset muuttujat saavat tiedon. Huomion arvoista ovat etenkin verteksi shaderin semantiikat muuttujien perässä (esim. POSITION, TEXCOORD0). (Engel 2004a, 27-30; Microsoft 2012h; Microsoft 2012j; Act-3D 2012b.)

Seuraavassa esimerkissä on esiteltynä verteksin rakenne ja yleisimmät sen sisään tulevat tiedot:

```
struct v_data {
        float4 position             : POSITION;
        float3 tangent              : TANGENT;
        float3 normal               : NORMAL;
        float2 texCoord             : TEXCOORD0;
};
```

Samalla pitää määritellä myöhemmin tarvittava pikseliohjelman tietorakenne. Siitä esimerkki seuraavana:

```
struct p_data {
        float4 position          : POSITION;
        float2 texCoord          : TEXCOORD1;
        float3 worldTangent      : TEXCOORD2;
        float3 worldBinormal     : TEXCOORD3;
        float3 worldNormal       : TEXCOORD4;
};
```

Nyt voidaan luoda varsinainen ohjelma, joka on muotoa funktio. Funktion tyyppinä pitää olla pikseliohjelman rakenne, koska verteksiohjelmasta palautettava tieto on sen muotoista. Funktion ensimmäiseksi parametriksi määritellään juuri esitelty verteksi rakenne. Verteksiohjelman sisältö käy ilmi seuraavassa esimerkissä:

```
p_data VOhjelma(v_data In)
{
    p_data Out = (p_data)0;          // Ulos menevän tiedon alustus
    Out.position = mul(In.position, wvp);  // Kamera ja projisointi
    Out.worldNormal = In.normal;                    // Normaalit
    Out.worldTangent = In.tangent;                  // Tangentit
    Out.worldBinormal = normalize(cross(Out.worldNormal,
    Out.worldTangent)).xyz;                         // Binormaalit
    Out.texCoord = In.texCoord;                     // UV-koordinaatit

    return Out;
}
```

5.2 Pikseliohjelma

Rasteroidut pinnat eli verteksiohjelman tuottamaa dataa käytetään pikseliohjelmassa. Pikseliohjelma on geometrian pinnan olemukseen vaikuttavan kuvankäsittelyllisen osuuden pääfunktio. Sen sisällä voidaan toteuttaa useita erilaisia algoritmeja ja kehittää uusia. Vaikutelmat erilaisten materiaalien ilmiöistä ja jälkikäsittelyefekteistä luodaan pikseliohjelmaan. Pikseliohjelman sisällöllä on siis merkittävä vaikutus ruudulle piirtyvään kuvaan. Alla on esimerkki, jossa 3d-malli valaistaan ja siihen yhdistetään diffuse-tekstuuri. Pikseliohjelma palauttaa COLOR-semantiikkaan perustuvan float4-vektorin. (Microsoft 2012h; Microsoft 2012j.)

```
float4 POhjelma(p_data In) : COLOR
{
    // Väridata diffuse tekstuurista
    float4 DiffuseMap = tex2D(DiffuseMapSampler, In.texCoord.xy);
    // Normaalien valmistelua
```

```
        float3 Nn = normalize(In.worldNormal);
        float3 Tn = normalize(In.worldTangent);
        float3 Bn = normalize(In.worldBinormal);
        float3 normals = ( Nn * 1.0) + (0.5 * Tn ) + (0.5 * -Bn);
        // Varjostus
        float4 Shading = max(0.5, dot(normalize(light_dir[0]), -
        normalize(mul(normals, objectOrient))));
        // Varjostuksen ja diffuse tekstuurin sekoitus
        return Shading*DiffuseMap;
    }
```

5.3 Funktioiden luonti

Omien funktioiden luominen on tilanteesta riippuen kannattavaa. Yleisiä ja hyviä ohjelmointitapoja noudattaen, toistuvat ja erityiset algoritmit on hyvä jakaa omiin funktioihin. Esimerkkinä useamman tekselin väliset laskutoimitukset, esimerkiksi sumennukset. Omat funktiot toimivat C- ja C++-funktioiden tavoin. Funktion eteen määritellään minkä tyyppisen muuttujan se palauttaa ja parametreiksi voi laittaa tarvittavia muuttujia. (Microsoft 2012j.)

Esimerkiksi funktio, joka sumentaa kuvaa voidaan esittää näin:

```
    float4 BlurPixels(float2 texCoord, float blurSize)
    {
        float4 blur = 1.0;
        for (int i = 0; i < 16; i++) {
                blur += tex2D(preRenderSampler,
                max(min(texCoord+blurSize*filterTable[i].xy, 1), 0));
        }
        return blur*0.0625;
    }
```

5.4 Valmiit funktiot

HLSL sisältää paljon valmiita matemaattisia funktioita, joista on paljon hyötyä. Esimerkiksi monikäyttöinen lineaarisen interpoloinnin funktio lerp(), etäisyys distance(), matriisien ja vektoreiden kertolasku mul(), ristitulo cross() ja pistetulo dot() ovat paljon käytettyjä funktioita. Numeeristen arvojen rajaaminen min() ja max() funktioilla on toisinaan hyödyllistä. Vektorin pituuden normalisointi onnistuu normalize()-funktiolla. Monikäyttöiset trigonometriset funktiot kuten sin(), cos(), tan() ja niiden käänteisfunktiot löytyvät myös. Kahdelle sivulle ylettyvässä kuvassa 19 on lueteltuna käytettävissä olevia funktioita, joiden perässä selostus englanniksi ja vähimmäisvaatimus ShaderModel versiosta. (Engel 2004a, 19-25.)

abort	Terminates the current draw or dispatch call being executed. 4
abs	Absolute value (per component). 11
acos	Returns the arccosine of each component of x. 11
all	Test if all components of x are nonzero. 11
AllMemoryBarrier	Blocks execution of all threads in a group until all memory accesses have been completed. 5
AllMemoryBarrierWithGroupSync	Blocks execution of all threads in a group until all memory accesses have been completed and all threads in the group have reached this call. 5
any	Test if any component of x is nonzero. 11
asdouble	Reinterprets a cast value into a double. 5
asfloat	Convert the input type to a float. 4
asin	Returns the arcsine of each component of x. 11
asint	Convert the input type to an integer. 4
asuint	Reinterprets the bit pattern of a 64-bit type to a uint. 5
asuint	Convert the input type to an unsigned integer. 4
atan	Returns the arctangent of x. 11
atan2	Returns the arctangent of of two values (x,y). 11
ceil	Returns the smallest integer which is greater than or equal to x. 11
clamp	Clamps x to the range [min, max]. 11
clip	Discards the current pixel, if any component of x is less than zero. 11
cos	Returns the cosine of x. 11
cosh	Returns the hyperbolic cosine of x. 11
countbits	Counts the number of bits (per component) in the input integer. 5
cross	Returns the cross product of two 3D vectors. 11
D3DCOLORtoUBYTE4	Swizzles and scales components of the 4D vector xto compensate for the lack of UBYTE4 support in some hardware. 11
ddx	Returns the partial derivative of x with respect to the screen-space x-coordinate. 21
ddx_coarse	Computes a low precision partial derivative with respect to the screen-space x-coordinate. 5
ddx_fine	Computes a high precision partial derivative with respect to the screen-space x-coordinate. 5
ddy	Returns the partial derivative of x with respect to the screen-space y-coordinate. 21
ddy_coarse	Computes a low precision partial derivative with respect to the screen-space y-coordinate. 5
ddy_fine	Computes a high precision partial derivative with respect to the screen-space y-coordinate. 5
degrees	Converts x from radians to degrees. 11
determinant	Returns the determinant of the square matrix m. 11
DeviceMemoryBarrier	Blocks execution of all threads in a group until all device memory accesses have been completed. 5
DeviceMemoryBarrierWithGroupSync	Blocks execution of all threads in a group until all device memory accesses have been completed and all threads in the group have reached this call. 5
distance	Returns the distance between two points. 11
dot	Returns the dot product of two vectors. 1
dst	Calculates a distance vector. 5
errorf	Submits an error message to the information queue. 4
EvaluateAttributeAtCentroid	Evaluates at the pixel centroid. 5
EvaluateAttributeAtSample	Evaluates at the indexed sample location. 5
EvaluateAttributeSnapped	Evaluates at the pixel centroid with an offset. 5
exp	Returns the base-e exponent. 11
exp2	Base 2 exponent (per component). 11
f16tof32	Converts the float16 stored in the low-half of the uint to a float. 5
f32tof16	Converts an input into a float16 type. 5
faceforward	Returns -n * sign(dot(i, ng)). 11
firstbithigh	Gets the location of the first set bit starting from the highest order bit and working downward, per component. 5
firstbitlow	Returns the location of the first set bit starting from the lowest order bit and working upward, per component. 5
floor	Returns the greatest integer which is less than or equal to x. 11
fmod	Returns the floating point remainder of x/y. 11
frac	Returns the fractional part of x. 11
frexp	Returns the mantissa and exponent of x.21
fwidth	Returns abs(ddx(x)) + abs(ddy(x)) 21
GetRenderTargetSampleCount	Returns the number of render-target samples. 4
GetRenderTargetSamplePosition	Returns a sample position (x,y) for a given sample index.4
GroupMemoryBarrier	Blocks execution of all threads in a group until all group shared accesses have been completed. 5
GroupMemoryBarrierWithGroupSync	Blocks execution of all threads in a group until all group shared accesses have been completed and all threads in the group have reached this call.5
InterlockedAdd	Performs a guaranteed atomic add of value to the dest resource variable.5
InterlockedAnd	Performs a guaranteed atomic and. 5
InterlockedCompareExchange	Atomically compares the input to the comparison value and exchanges the result.5
InterlockedCompareStore	Atomically compares the input to the comparison value. 5
InterlockedExchange	Assigns value to dest and returns the original value. 5
InterlockedMax	Performs a guaranteed atomic max. 5
InterlockedMin	Performs a guaranteed atomic min. 5
InterlockedOr	Performs a guaranteed atomic or. 5
InterlockedXor	Performs a guaranteed atomic xor. 5
isfinite	Returns true if x is finite, false otherwise.11
isinf	Returns true if x is +INF or -INF, false otherwise. 11
isnan	Returns true if x is NAN or QNAN, false otherwise. 11
ldexp	Returns x * 2exp 11
length	Returns the length of the vector v. 11
lerp	Returns x + s(y - x). 11
lit	Returns a lighting vector (ambient, diffuse, specular, 1) 11

KUVA 19. Luettelo käytettävistä funktioista (Microsoft 2012c; Viljanen 2012)

Luettelo eri HLSL ShaderModel versioissa käytettävistä funktioista jatkuu tässä.

log	Returns the base-e logarithm of x.	11
log10	Returns the base-10 logarithm of x.	11
log2	Returns the base-2 logarithm of x.	11
mad	Performs an arithmetic multiply/add operation on three values.	5
max	Selects the greater of x and y.	11
min	Selects the lesser of x and y.	11
modf	Splits the value x into fractional and integer parts.	11
msad4	Compares a 4-byte reference value and an 8-byte source value and accumulates a vector of 4 sums.	5
mul	Performs matrix multiplication using x and y.	1
noise	Generates a random value using the Perlin-noise algorithm.	11
normalize	Returns a normalized vector.	11
pow	Returns xy.	11
printf	Submits a custom shader message to the information queue.	4
Process2DQuadTessFactorsAvg	Generates the corrected tessellation factors for a quad patch.	5
Process2DQuadTessFactorsMax	Generates the corrected tessellation factors for a quad patch.	5
Process2DQuadTessFactorsMin	Generates the corrected tessellation factors for a quad patch.	5
ProcessIsolineTessFactors	Generates the rounded tessellation factors for an isoline.	5
ProcessQuadTessFactorsAvg	Generates the corrected tessellation factors for a quad patch.	5
ProcessQuadTessFactorsMax	Generates the corrected tessellation factors for a quad patch.	5
ProcessQuadTessFactorsMin	Generates the corrected tessellation factors for a quad patch.	5
ProcessTriTessFactorsAvg	Generates the corrected tessellation factors for a tri patch.	5
ProcessTriTessFactorsMax	Generates the corrected tessellation factors for a tri patch.	5
ProcessTriTessFactorsMin	Generates the corrected tessellation factors for a tri patch.	5
radians	Converts x from degrees to radians.	1
rcp	Calculates a fast, approximate, per-component reciprocal.	5
reflect	Returns a reflection vector.	1
refract	Returns the refraction vector.	11
reversebits	Reverses the order of the bits, per component.	5
round	Rounds x to the nearest integer	11
rsqrt	Returns 1 / sqrt(x)	11
saturate	Clamps x to the range [0, 1]	1
sign	Computes the sign of x.	11
sin	Returns the sine of x	11
sincos	Returns the sine and cosine of x.	11
sinh	Returns the hyperbolic sine of x	11
smoothstep	Returns a smooth Hermite interpolation between 0 and 1.	11
sqrt	Square root (per component)	11
step	Returns (x >= a) ? 1 : 0	11
tan	Returns the tangent of x	11
tanh	Returns the hyperbolic tangent of x	11
tex1D(s, t)	1D texture lookup.	1
tex1D(s, t, ddx, ddy)	1D texture lookup.	21
tex1Dbias	1D texture lookup with bias.	21
tex1Dgrad	1D texture lookup with a gradient.	21
tex1Dlod	1D texture lookup with LOD.	31
tex1Dproj	1D texture lookup with projective divide.	21
tex2D(s, t)	2D texture lookup.	11
tex2D(s, t, ddx, ddy)	2D texture lookup.	21
tex2Dbias	2D texture lookup with bias.	21
tex2Dgrad	2D texture lookup with a gradient.	21
tex2Dlod	2D texture lookup with LOD.	3
tex2Dproj	2D texture lookup with projective divide.	21
tex3D(s, t)	3D texture lookup.	11
tex3D(s, t, ddx, ddy)	3D texture lookup.	21
tex3Dbias	3D texture lookup with bias.	21
tex3Dgrad	3D texture lookup with a gradient.	21
tex3Dlod	3D texture lookup with LOD.	31
tex3Dproj	3D texture lookup with projective divide.	21
texCUBE(s, t)	Cube texture lookup.	11
texCUBE(s, t, ddx, ddy)	Cube texture lookup.	21
texCUBEbias	Cube texture lookup with bias.	21
texCUBEgrad	Cube texture lookup with a gradient.	21
texCUBElod	Cube texture lookup with LOD.	31
texCUBEproj	Cube texture lookup with projective divide.	21
transpose	Returns the transpose of the matrix m.	1
trunc	Truncates floating-point value(s) to integer value(s)	1

KUVA 19. Luettelo käytettävistä funktioista (Microsoft 2012c; Viljanen 2012)

5.5 Technique- ja pass-määritelmä

Technique-määritelmä on shaderin loppuun kirjoitettu osuus. Siinä määritellään mm. rasterointiin liittyviä parametrejä, kuten näkymättömien pintojen leikkaus eli ns. CullMode ja käytettävät verteksi- ja pikseliohjelmat. Lisäksi on mahdollista määritellä muita syvyyspuskurointiin ja läpinäkyvyyksiin liittyviä parametrejä.

Pass-määritelmällä voi vaikuttaa renderöityihin kertoihin eri tavoilla saman 3d-objektin kohdalla. Se liittyy osaksi MRT (Multiple Render Targets) -tekniikalla renderöintiä, josta lisää luvussa 6.2. Pass-määritelmää voi käyttää esimerkiksi saman objektin erilaisten tilojen esittämiseen, kuten siirtämällä hiiren kursorin objektin päälle aiheuttaisi jonkin muutoksen sen ulkoasussa.

Verteksi- ja pikseliohjelmien esittely technique-määritelmässä mahdollistaa eri pääfunktioiden versioiden suorittamisen samasta ohjelmasta vaihtamalla yhtä parametria. Siinä voi lisäksi määritellä saman ohjelman toimimaan eri ShaderModel-versioilla ja ratkaista näin kattavampi toimivuus eri-ikäisissä järjestelmissä. Seuraavana esimerkki, millaiselta SM 3.0:n technique-määritelmä voi näyttää:

```
technique ShaderEsimerkki_SM3
{
    pass P0
    {
            CullMode = None;
            VertexShader = compile vs_3_0 VOhjelma();
            PixelShader = compile ps_3_0 POhjelma();
    }
}
```

5.6 Havainnollistavia esimerkkejä

5.6.1 Dynaaminen pintamateriaali

Kuvassa 20 näkyvä koodi toimii esimerkiksi maalatun metallin pintamateriaalina. Koodi yhdistää sille annetun geometrian ja neljä tekstuuria keskenään. Tekstuureina se käyttää diffuse-, normal-, environment- ja reflection-karttoja. Verteksi- ja pikseli-shaderit ovat kumpikin esiteltyinä, ja ne tulkitaan ShaderModel 3.0 formaatin mukaan. SM 3.0 varaa näytönohjaimelta muistia sille kuuluvien rajojen sekä esimerkin ohjelman asettaman tarpeen puitteissa.

```
// ExcavatorMaterial Shader
// Vesa Viljanen 24.2.2012

int UseTangent  : UseTangent;

float4x4 objectOrient : CHANNELMATRIX0;

texture DiffuseMap : TEXTURE0;
sampler2D DiffuseMapSampler = sampler_state
{
        Texture = <DiffuseMap>;
        MinFilter = LINEAR;
        MagFilter = LINEAR;
        MipFilter = LINEAR;
};

texture NormalMap : TEXTURE1;
sampler2D NormalMapSampler = sampler_state
{
        Texture = <NormalMap>;
        MinFilter = LINEAR;
        MagFilter = LINEAR;
        MipFilter = LINEAR;
};

texture EnvMap : TEXTURE2;
sampler2D EnvMapSampler = sampler_state
{
        Texture = <EnvMap>;
        MinFilter = LINEAR;
        MagFilter = LINEAR;
        MipFilter = LINEAR;
};

texture RefMap : TEXTURE3;
sampler2D RefMapSampler = sampler_state
{
        Texture = <RefMap>;
        MinFilter = point;
        MagFilter = point;
        MipFilter = point;
};

float3 light_dir[8]         : LIGHTDIR;

float4x4 wvp : WorldViewProjection;
float4x4 vp : ViewProjection;

struct a2v {
   float4 position              : POSITION;
   float3 tangent               : TANGENT;
   float3 normal                : NORMAL;
   float2 texCoord              : TEXCOORD0;
};

struct v2f {
   float4 position              : POSITION;
   float2 texCoord              : TEXCOORD1;
   float3 worldTangent          : TEXCOORD2;
   float3 worldBinormal         : TEXCOORD3;
   float3 worldNormal           : TEXCOORD4;
};

// Verteksi varjostin //
v2f av(a2v In)
{
   v2f Out = (v2f)0;
   Out.position = mul(In.position, wvp);
   Out.worldNormal = In.normal;
   Out.worldTangent = In.tangent;
   Out.worldBinormal = normalize(cross(Out.worldNormal, Out.worldTangent)).xyz;
   Out.texCoord = In.texCoord;

   return Out;
}

// Pikseli varjostin //
float4 af(v2f In) : COLOR
{
float4 DiffuseMap = tex2D(DiffuseMapSampler, In.texCoord.xy);
float4 RefMap = tex2D(RefMapSampler, In.texCoord.xy);
float3 NormalMap = tex2D(NormalMapSampler, In.texCoord.xy);
NormalMap = NormalMap.xyz * 2 - 1;

float3 Nn = normalize(In.worldNormal);
float3 Tn = normalize(In.worldTangent);
float3 Bn = normalize(In.worldBinormal);
float3 WorldXform = ( Nn * NormalMap.z ) + (NormalMap.x * Tn ) + ( NormalMap.y * -Bn);

float4 Shading = max(0.5, dot(normalize(light_dir[0]), -normalize(mul(WorldXform, objectOrient))));
float4 EnvMap = tex2D(EnvMapSampler, normalize(mul(WorldXform, wvp))*0.5+0.5);
RefMap = EnvMap*EnvMap*RefMap*0.2;

return Shading*DiffuseMap+RefMap;
}

technique Complete
{
   pass main
   {
      VertexShader = compile vs_3_0 av();
      PixelShader = compile ps_3_0 af();
   }
}
```

KUVA 20. Pintamateriaali -ohjelma, HLSL (Viljanen 2012)

5.6.2 Syvyyskartta

Syvyyskartan voi laskea shader-ohjelman sisällä. Sen laskeminen on suhteellisen yksinkertaista ja auttaa hahmottamaan verteksi shaderin ja pikseli shaderin yhteyttä. Kuvassa 21 esitetty koodi suorittaa operaation, jossa verteksien sijainnit projisoidaan perspektiivisenä kaksiulotteiselle pinnalle. Muunnettujen verteksien sijainnin z-komponentti asettuu ruutuun katsottuna syvyyssuuntaisesti. Tästä saadaan syvyysinformaatio, joka rasteroidaan ja säädetään pikseliohjelmassa näkymään sopivalta alueelta.

```
// DepthShader
// Vesa Viljanen 2012

float4x4 wvp : WorldViewProjection;

struct a2v {
	float4 position		: POSITION;
};

struct v2f {
	float4 position		: POSITION;
	float4 depthMap		: TEXCOORD0;
};

v2f av(a2v In)
{
	v2f Out = (v2f)0;
	Out.position = mul(In.position, wvp);
	Out.depthMap = Out.position.xyzw;

	return Out;
}

float4 af(v2f In) : COLOR
{
	return In.depthMap * 0.02;
}

technique Complete
{
   pass main
   {
	VertexShader = compile vs_2_0 av();
	PixelShader = compile ps_2_0 af();
   }
}
```

KUVA 21. Syvyyskartan laskenta -ohjelma, HLSL (Viljanen 2012)

6 MUITA SHADEREIHIN LIITTYVIÄ TEKNIIKOITA

6.1 Tekstuuriin renderöinti (Render to texture)

Back buffer eli taustapuskuri on dynaaminen muistialue, johon renderöitävät kuvat koostetaan. Taustapuskuri pitää tallessa viimeisimmän renderöidyn kuvan, kunnes sen päälle kirjoitetaan uusi. Joka kierroksen lopussa taustapuskuriin koostettu kuva siirretään ruutupuskuriin (frame buffer) eli tulostetaan tietokoneen näytölle. Taustapuskuri voidaan kuitenkin lukea tekstuuriin useasti jo ennen lopullisen kuvan muodostamista. Silloin se antaa väylän hyödyntää useita väliaikaisia renderöintejä ns. passeja (pass). Passien koostaminen tapahtuu myöhemmin osana lopullisen kuvan renderöintiä. Passien avulla koostettua kuvaa voidaan kutsua myös termillä Deffered Shading. (Engel 2009, 9-10.)

Lisäksi tekstuuri antaa väylän palauttaa dataa näytönohjaimelta takaisin keskusmuistiin. Tekniikalla voi luoda esimerkiksi omia fysiikkasimulaatioita. Tekstuurin datan lukeminen keskusmuistiin ei onnistu shaderohjelmassa, vaan se pitää ohjelmoida osana Direct3D-ohjelmaa. Tekstuuriin renderöinti on tärkeässä osassa mm. seuraavien efektien luonnissa:

- värimäärittely ja kompositointiefektit (DoF, sumu)
- Screen-space Ambient Occlusion (SSAO)
- varjot (Shadow Mapping)
- GPGPU-laskenta käyttäen tekstuureita virtoina keskusmuistin suuntaan.

6.2 Usean kerroksen renderöinti (Multiple Render Targets, MRT)

MRT liittyy olennaisena osana useamman vaiheen eli passin renderöintiin samaa shader-ohjelmaa käyttäen. MRT helpottaa erilaisten ns. Deffered Shading tekniikoiden käyttöä. Deffered Shading -tekniikassa renderöidään erilaisia kerroksia tekstuureina. Kerroksista koostetaan lopullinen kuva yhdistämällä ne keskenään. Luvun 5.5 technique- ja pass-määritelmässä vaikutetaan tähän osuuteen. (Act-3D 2012a.)

6.3 Verteksitekstuurit (Vertex Texture Fetch, VTF)

VTF on lyhenne sanoista Vertex Texture Fetch, ja se tarkoittaa ns. tekstuurin käyttämistä verteksi shaderissä eli ts. tekstuuri voidaan tulkita esimerkiksi korkeuskarttana. Tämä on mahdollista Direct3D 9 ja sen jälkeen tulleissa näytönohjaimissa, mutta pienin poikkeuksin. Esimerkiksi Nvidian 6- ja 7-sarjan näytönohjainten verteksitekstuurit ei tue muita kuin 32-bittisiä bittikarttoja, mutta siitä uudemmat tukevat yleisempiä 8-bittisiäkin, joiden tarkkuus on toki heikompi.

Verteksitekstuuri esitellään kuin muutkin tekstuurit, mutta tekstuurin samplerissa suodatukset jätetään pistemäiseksi (point). Verteksiohjelman koodissa näytteenottajan lukeminen tapahtuu tex2Dlod()-funktiolla. (NVIDIA 2004.)

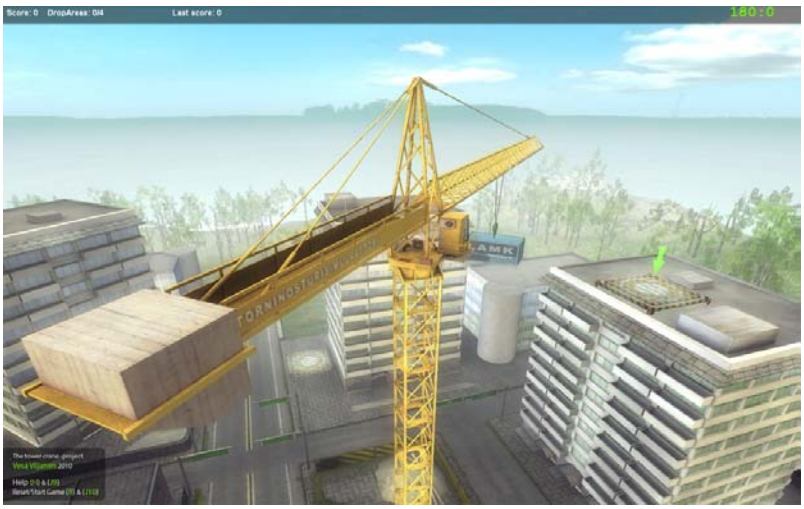
6.4 GPGPU (General Purposes on GPU)

Nykypäivänä GPU:lla suoritetaan paljon muutakin laskentaa kuin vain reaaliaikaisen grafiikan tuottamista. Sen on tehnyt mahdolliseksi uudet tekniikat, kuten CUDA ja OpenCL. GPGPU suorittamista voi tehdä DirectX 11 ja sitä tukevat näytönohjaimet, koska ne mahdollistavat datavirtojen palauttamisen keskusmuistiin. DirectX 11:een on luotu erillinen Compute Shader -vaihe, jolla helpotetaan keskusmuistin ja näytönohjaimen välistä toimintaa. Aiemmissa versioissa datan lukemista keskusmuistiin on ollut mahdollista toteuttaa vain tekstuuriin renderöinnin kautta tai ei ollenkaan.

7 CASE, KAIVINKONESIMULAATTORI

7.1 Kaivinkonesimulaattorista yleisesti

Ajatus kaivinkonesimulaattorin kehittämiseen lähti ideasta viedä aiemmin kehitettyä kuvassa 22 näkyvää torninosturisimulaattoria eteenpäin, mutta uusin haastein. Vaihtoehtona ei ollut torninosturin kelpuuttaminen jatkokehitettäväksi, vaan halu luoda uusi kokonaisuus puhtaalta pöydältä ja jotain toiminnallisuudeltaan dynaamisempaa. Kaivinkoneessa toteutuu torninosturin kaltainen kiertyvä liike, johon on yhdistetty tela-alustainen liikkuminen. Operointi kaivinkoneen kauhalla muodostuu kolmen akselin, kahden puomin ja kauhan samanaikaisesta käytöstä.



KUVA 22. Torninosturisimulaattori (Viljanen 2012)

Alun perin kaivinkonesimulaattorissa itse simulaation kehittäminen piti olla tärkeimmässä osassa ja perehtyä enemmän kaivinkoneen fyysiseen olemukseen. Lähdemateriaalia kaivinkonesimulaattoreista on saatavilla kuitenkin suhteellisen vähän. Lähteiden hankintaan auttoi osaltaan Sipoon ja Porvoon rajalla sijaitsevalla Anttilan sähköasemalla käynti ja kuvaus. Kuvatut videot ovat pääasiassa kaivinkoneen sivulta kuvattuja, joissa erottuvat selvästi puomien ja kauhan liikeradat, sekä heilahdukset. Kuva 23 on kaapattu kuvatusta videosta.

KUVA 23. Konevuori Oy:n kaivinkone Anttilan sähköaseman laajennustöissä tammikussa 2012 (Viljanen 2012)

Lisäksi simulaattoria varten täytyi rakentaa ohjaimet, joilla oikeankaltainen ohjaus onnistuisi. Loppujen lopuksi kokonaisuuden isoimmaksi osaksi muodostui shadereiden kehittäminen ja hyödyntäminen. Kuvassa 24 on nähtävissä kaivinkonesimulaattorin kokonaisuus käyttövalmiina. Case-osuus on jaettu kolmeen osioon, joissa käsitellään ohjainten, kaivinkoneen mallinnuksen ja shadereiden toteutusta. Kokonaisuus pohjautuu Quest3d-moottoriin, Lua-skriptikielen avulla luotuihin loogisiin rakenteisiin ja HLSL shader-ohjelmointiin. Lisäksi mukaan on lisätty muutamia ääniä tehostamaan vaikutelmaa ja antamaan tunnelmaa.

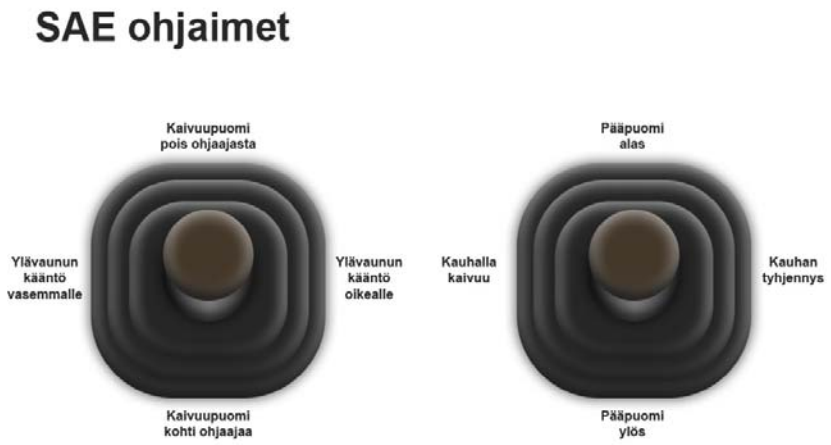
KUVA 24. Kaivinkonesimulaattori kokonaisuudessaan (Viljanen 2012)

7.2 Ohjaimet

7.2.1 Ajo- ja kaivuuohjainten teoriaa

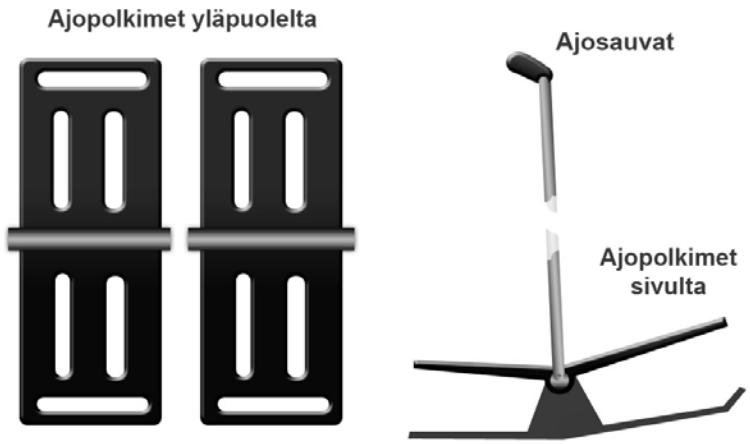
Kaivinkoneen ohjaukseen on mahdollista käyttää valmiita peliohjaimia, kuten ns. gamepadia tai joystickeja. Kaivinkoneen ohjaamoon verraten kohdistetumpi ratkaisu on rakentaa ohjaimet itse tai hankkia valmis ratkaisu joltain simulaattoriohjainten rakentajalta. Budjetista riippuen ohjaimiin tai ohjaamoon on mahdollista käyttää rahaa suuriakin summia. Erilaisilla virtuaalitodellisuutta tehostavilla laitteilla voidaan budjetista riippuen saada vaikuttaviakin kokemuksia.

Tärkeimmät ohjaimet kaivinkoneessa ovat yleensä SAE-standardien mukaiset joystick-malliset kaivuuohjaimet. SAE-ohjaimet toimivat kuvan 25 kaltaisiin suuntiin.



KUVA 25. SAE-standardin mukaiset kaivinkoneen ohjaimet (Viljanen 2012)

Seuraavaksi tärkeimmät ovat kaivinkoneen ajopolkimet ja ajosauvat. Ajopolkimet ja ajosauvat ovat yleensä mekaanisesti yhdistetyt toisiinsa oikeissa kaivinkoneissa. Kaivinkoneen ohjaus tapahtuu kummankin telaketjun nopeutta säätelemällä erikseen. Ajo-ohjaimia pitää olla siis kaksi eteen ja taaksepäin liikkuvaa laitetta, jotta kumpaakin telaketjua voidaan ohjata eteen- ja taaksepäin erikseen.



KUVA 26. Kaivinkoneen ajopolkimet ja -sauvat (Viljanen 2012)

7.2.2 Ajo-ohjainten toteutus

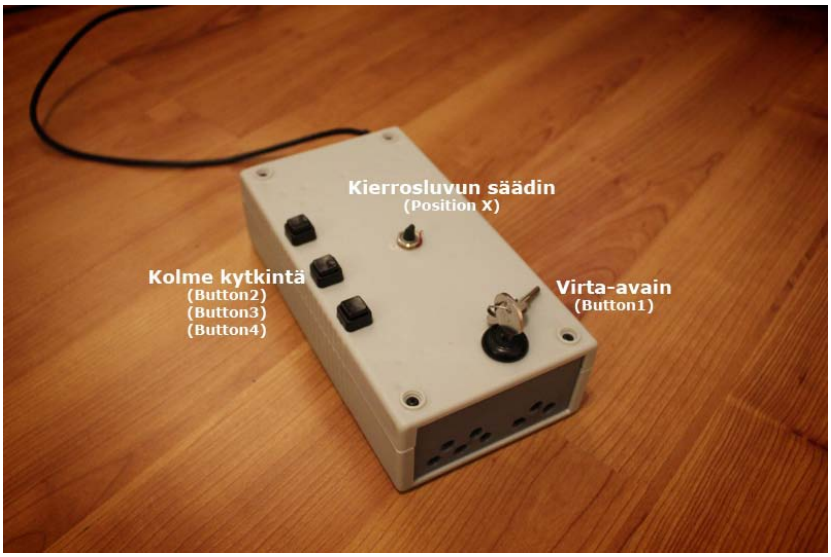
Peliohjaimet on helpoin toteuttaa jonkin valmiin usb-peliohjaimen pohjalta. Esimerkiksi joystickeissa on jo kaksi mahdollista potentiometriä, joilla mekaaninen pyörimisliike saadaan muunnettua jännitteen muutokseksi. Potentiometrin vaikutus jännitteeseen voidaan taas tulkita Analog-to-Digital- eli A/D-piirillä, jonka tietokoneen pitää seuraavaksi tulkita peliohjaimena. Tärkeämpää on oikeastaan löytää näitä valmiita piirilevyjä, joissa on useampi A/D-muuntimen liitäntä ja joitain kytkinliitäntöjä. Potentiometrejä ja sähkökytkimiä eli mm. nappeja saa sähkötarvikekaupasta kohtalaisen edulliseen hintaan. Valmiita peliohjainten piirilevyjäkin saa tilattua Internetistä.

Sopivien ohjainten puuttuessa voi toteuttaa esimerkiksi kuvan 27 kaltaiset ohjaimet. Hyvä lähtökohta kaivinkoneen ohjainten kehittämiseen on esimerkiksi hankkia ratti ja polkimet -yhdistelmä. Polkimien kohdalla ongelmaksi muodostuu liikeratojen yksisuuntaisuus. Sen voi korjata poistamalla olemassa olevan jousen ja kiristämällä polkimen kahden eri suuntaan kiskovan jousen väliin. Tasapaino ja kireys on olennaista löytää mahdollisimman keskelle, jotta digitaalisen liikeradan keskipiste voidaan siirtää mahdollisimman lähelle polkimen nykyistä keskiasentoa. Lisäksi polkimen pinta pitää rakentaa uudelleen, jotta sitä voidaan painaa jalalla kumpaankin suuntaan. Uudistetussa rakenteessa on käytetty kulmarautoja ja jousia. Polkimen pinnat ovat peltiä ja kumimattoa.

KUVA 27. Ajopolkimet ratti ja poljin yhdistelmästä (Viljanen 2012)

Lisäksi ratti- ja poljinyhdistelmästä voi ratin potentiometriä ja ratissa olevia nappeja hyödyntämällä luoda ohjaamon fyysistä käyttöliittymää lisää. Ratin tilalle voi rakentaa mm. pyöritettävän kierrosluvun säätimen. Kuvassa 28 oleva pieni nuppi on potentiometrin akseli, joka pitäisi korvata isommalla säätimellä.

Koteloidussa ohjainlaatikossa eli ns. kojelaudassa on kolme painokytkintä ja käynnistysavain. Teknisesti kotelon sisällä on Thrustmaster Force Feedback Wheel -ratista sen koko piirilevy, jonka painokytkimet on johdotettu ja kytketty uudelleen em. kytkentöihin. Käynnistysavaimen ollessa käyntiasennossa Button1:n arvo on 1 ja muulloin 0. Käynnissä virta kulkee jatkuvasti kytkimen läpi, tietokoneella tieto ei muutu ja kaivinkone pysyy käynnissä. Käynnistysavain ei vaikuta koko ohjainpiirin virran kulkuun, joten se ei ole ohjainlaatikon virta-avain. Silloin kaikki muutkin ohjaimet toimivat jatkuvasti, mutta täydellisemmän virta-avain merkityksen voisi simuloida ohjelmassa. Kolme muuta kytkintä on käyttövapaana, ja niihin voi lisätä esimerkiksi resetoinnin tai apuikkunan näkymisen. Yhdessä näistä on käytössä kuvakulman vaihto.



KUVA 28. Ohjainlaatikon (ns. kojelauta) käyttöliittymä (Viljanen 2012)

Kuvassa 28 näkyy, mitkä napit ja akselit tietokone tulkitsee peliohjaimelta. Kuva 29 kertoo liitäntöjen ja virtakytkimen sijainnit.

KUVA 29. Ohjainlaatikon kytkennät (Viljanen 2012)

7.2.3 Kaivuuohjainten toteutus

Kaivuuohjaimet voi rakentaa kahden samanlaisen joystickin pohjalta, jolloin kummassakin on samanlainen käyttötuntuma. Tässä tapauksessa se ei ollut kuitenkaan mahdollista Quest3d-ohjelman asettaman enintään kahden peliohjaimen rajoitteen takia. Joystickit ts. kaivuuohjaimet voi rakentaa nekin itse, vaikka käyttäen jotain valmista ratkaisua pohjalla. Kuvassa näkyvät ohjaimet on rakennettu yhden lennokkisimulaattorin ohjaimen pohjalta. Ohjain sisälsi valmiiksi mekaanisen rakenteen, joissa ohjainsauvat ovat kiinni. Ohjainsauvojen akseleiden kautta kääntyvä liike saadaan siirtymään yhteensä neljään potentiometriin, joiden jännite tulkitaan A/D- tai muun peliohjainpiirin avulla peliohjaimeksi. Aiemmin lähekkäin kytketyt ohjainsauvat piti siirtää kauemmas toisistaan jatkamalla johtoja pidemmiksi. Lisäksi ohjaimen sauvat voi joutua kiristämään ulospäin kulmia kohden, jotta niihin saadaan riittävä jäykkyys ja palautus keskiasentoon. Pienet heilahdukset ja vinot keskiasennot näkyvät helposti tietokoneen tulkitsemana. Näistä aiheutuvat häiriöt voidaan poistaa jälkikäteen, mutta huonoimmassa tapauksessa se tuhoaa ohjaimen tarkkuusaluetta liikaa. Kuvassa 30 näkyy, että ohjaimet ovat koteloitu ja sauvojen tartuntapintana on vanhat maastopyörän gripit.

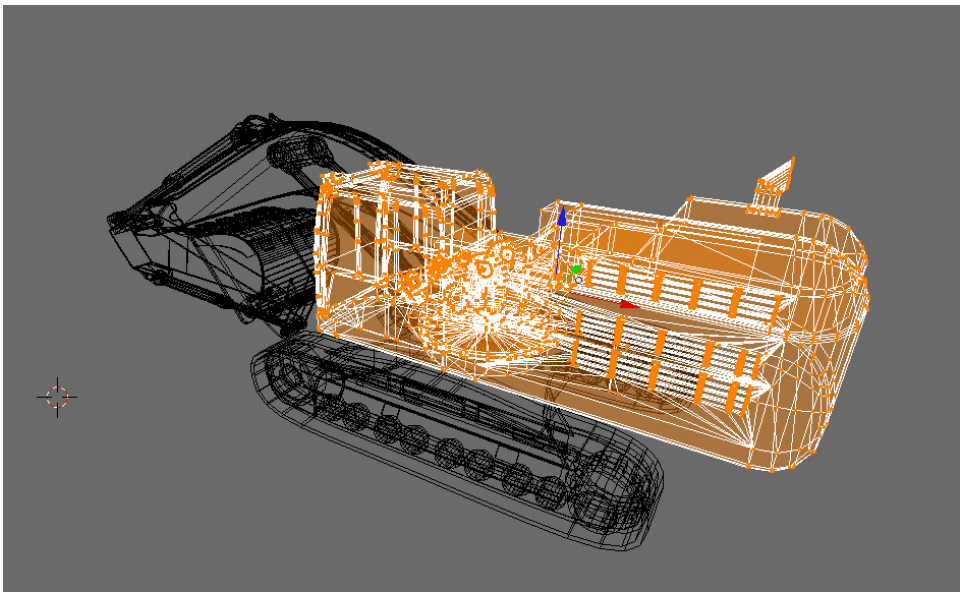
KUVA 30. Kaivinkonesimulaattorin kaivuuohjaimet, ”joystickit” (Viljanen 2012)

7.3 Mallinnus

7.3.1 Kaivinkoneen mallinnus

Mallintaessa objekteja peliin tai simulaattoriin pitää muistaa näytönohjaimen asettamat rajoitteet. Liian tarkka malli voi prosessorista ja näytönohjaimesta riippuen hidastaa koko ohjelman toiminnan. Epätarkasta mallista ei ole sikäli haittaa suorituskyvyn kannalta, mutta heikko graafinen ulkoasu voi pilata autenttisen tunnelman. Mallinnus on aloitettu etsimällä Internetistä ns. blueprinttejä eli pohjapiirustuksia kaivinkoneista. Yleensä kuvat ovat peräisin jostain jaossa olevasta ohjeesta liittyen oikean kokoluokan kaivinkoneeseen tai siitä tehtyyn pienoismalliin. Autenttisen kuvan käyttö työskentelyn ohessa auttaa hahmottamaan kolmiulotteisia muotoja. Esimerkissä mallinnusvaihe on toteutettu kokonaisuudessaan Blender-ohjelmalla.

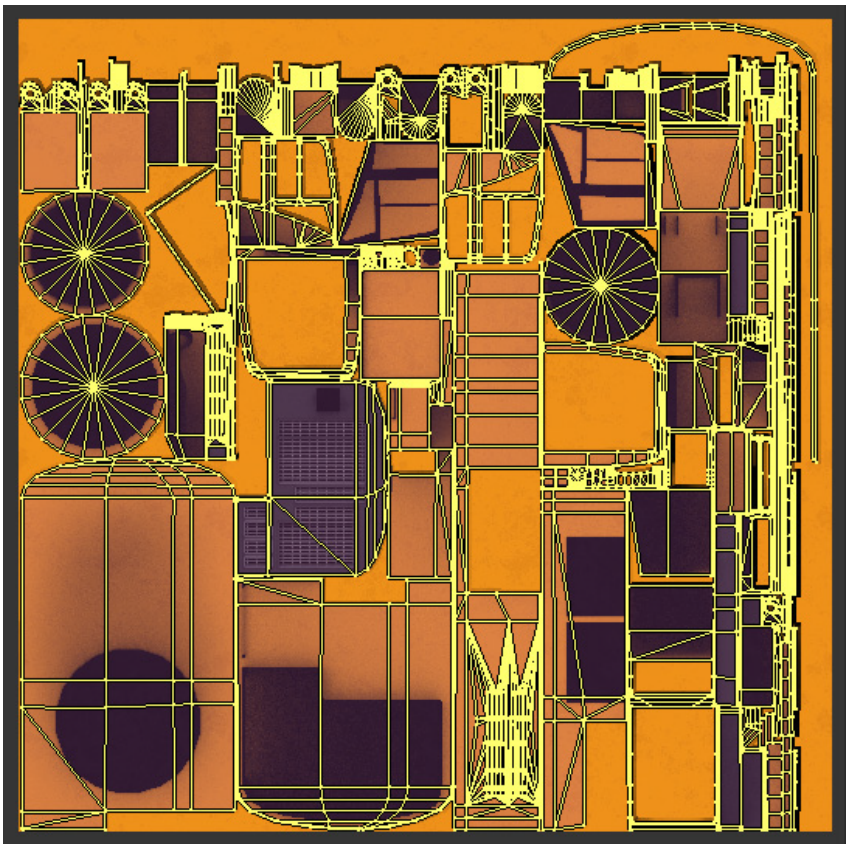
Kaivinkoneen kaltaisessa objektissa on paljon pursotettuja kaksiulotteisia tasopintoja, kuten puomissa. Ne on mahdollista mallintaa käyttämällä ns. splinetekniikkaa ja pursottamalla (extrude) haluttuun paksuuteen. Hyvä puoli spline tekniikalla mallintaessa on sen geometrisen tarkkuuden säätäminen jälkikäteen. Pyöreät ja monisuuntaiset muodot on mallinnettu ns. mesh-tekniikalla, jossa työskennellään suoraan pinnat muodostavien neliöiden ja kolmioiden avulla. Kuvassa 31 näkyy kaivinkoneen ylävaunun pintarakenne, joka on tyypiltään kolmioista koostuva mesh. (Wikipedia 2012d)



KUVA 31. Kaivinkoneen low-poly malli (Viljanen 2012)

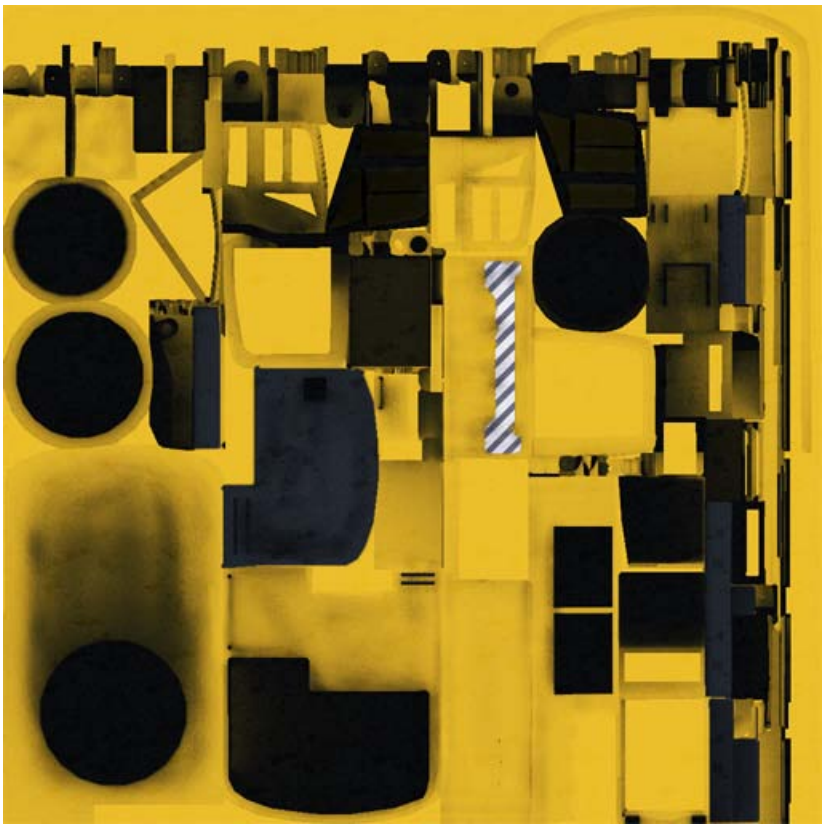
7.3.2 Kaivinkoneen teksturointi

Valmiin mallin pinnat kartoitetaan teksturointia varten. Kartoittamista kutsutaan UV-mappaukseksi. UV-mappaukseen on olemassa erilaisia algoritmeja. Niillä pintoja saadaan automaattisesti levitettyä tasopinnalle, kuten kuvassa 32 on esitetty. U- ja V-kirjaimet tulevat kaksiulotteisesta tasopinnasta, johon pinnat levitetään. Koordinaatisto koostuu kaksikomponenttisesta yksikkövektorista eli leveydestä ja korkeudesta, joiden kummankin arvot rajautuvat 0.0 - 1.0 välille. Yksikkövektorin käytön takana on se, että tekstuurin koko voidaan vaihtaa ilman tarvittavia operaatioita UV-mappaukseen. Esimerkin kuvassa 32, UV-mappaus on toteutettu Blenderillä.



KUVA 32. Kaivinkoneen UV-mappaus ja pohjalla tekstuuri (Viljanen 2012)

Kaivinkoneen taipuvat pinnat ovat haasteellisia, mutta puomin tasopinnat levittyvät helposti. Pinnat pitää asetella UV-karttaan mahdollisimman tarkasti ja oikeissa mittakaavoissa, jotta saadaan mahdollisimman korkea tarkkuus jokaiselle pinnalle. Kun kartoitustyö on valmis, aloitetaan kuvamuotoisen tekstuurin tuottaminen. Usein teksturoinnin apuna käytetään ns. highpoly-mallia, joka on lopullista low-poly-mallia tarkempi. Usein highpoly-mallissa on teräviä kulmia hiukan pyöristetty ja lisätty muita pieniä kuperuuksia tai koverruksia. Highpoly-mallista pystyy renderöimään yksityiskohtia vähemmän tarkemman mallin tekstuureiksi.



KUVA 33. Lopullinen diffuse tekstuuri (Viljanen 2012)

Useimmiten renderöidään ns. pintojen suuntia jäljittelevä ”normalmap” ja hajavaloa jäljittelevä ”ambient occlusion”. Ambient occlusion eli AO on hyvä pohja väritystä määrittelevälle diffuse-tekstuurille. AO ei aina vaadi highpoly-mallia, joten se voidaan renderöidä lowpoly-mallinkin pohjalta. Tekstuuria voi nyt maalata ja käsitellä lisää kuvankäsittelyohjelmassa. Mitä enemmän yksityiskohtia tekstuuriin tekee, sen paremmalta se yleensä näyttää. Esimerkiksi kuvan 33 tekstuurissa on käytetty Blenderillä renderöityä AO:ta ja Photoshop-ohjelmaa.

Halutessa tekstuureita pitää tuottaa useampia jo edellä mainittujen diffuse- ja normalmappien lisäksi. Muita tekstuureita ovat esimerkiksi kiillot määrittelevä specularmap ja heijastukset määrittelevä reflectionmap. Tekstuurien määrän määrittelee shader-ohjelman sisältö ja näytönohjaimen rajat.

7.4 Shaderit (Direct3D 9)

7.4.1 Kaivettava maasto

7.4.1.1 Alustus dynaamiseen maastoon

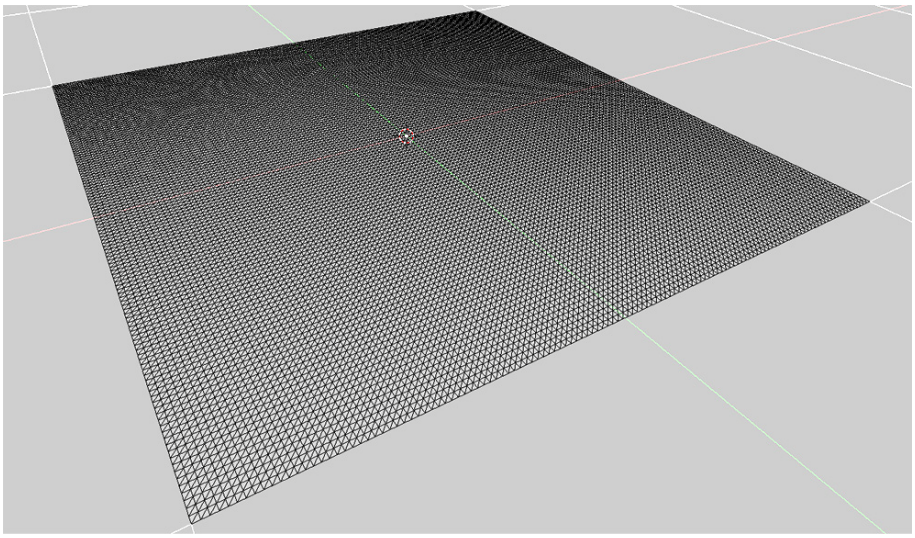
Dynaamisuus maastossa on mahdollista toteuttaa kahdella toisistaan merkittävästi eroavalla tavalla. Tavat ovat joko prosessorilla tai näytönohjaimella. Kehittäjän taidoista ja teknisistä vaatimuksista riippuen voidaan soveltaa kumpaakin tapaa eri tarkoituksiin. Lisäksi kummassakin tavassa on omat hyvät ja huonot puolensa.

Prosessorin hyvät puolet on esimerkiksi se, että maaston geometria on aina valmiina ennen kuin se piirretään. Suoritusta voidaan optimoida siten, että geometriset muutokset tehdään vain silloin kun maastoa muokataan. Muun ajan geometrinen data pysyy vakiona ja näytönohjaimelle siirrettäessä se piirretään kuin muutkin 3d-objektit. Lisäksi prosessorilaskenta mahdollistaa fysiikkamoottorin käytön, koska geometrian dataan on aina pääsy tietokoneen keskusmuistissa. Dynaaminen maasto voi kuitenkin tuoda ongelmia mm. fysiikkamoottorin kanssa, jos maaston päivittyminen törmäystarkastuksiin on hidasta. Muitakin nopeusongelmia voi esiintyä prosessorin laskentanopeuden ollessa suhteellisen hidas verrattuna näytönohjaimeen.

Näytönohjaimella toteutettu maasto ja sen muokkautuvuus on taas kohtalaisen nopeaa ja suoraviivaista, kun korkeuskartta tai kartat saadaan mukaan. Ongelmana tosin voi olla vanhemmat näytönohjaimet, jotka eivät tue tekstuurin suorittamista verteksi shaderissä eli VTF:ää (Vertex Texture Fetch). Muitakin heikkoja puolia näytönohjaimella toteutettuun maastoon löytyy, kuten verteksien normaalien puuttuminen ja laskettua geometriaa ei voida hyödyntää olemassa olevana tietona esimerkiksi fysiikkamoottorissa. Normaalit voidaan kuitenkin laskea korkeuskartan perusteella. Näytönohjain antaa varteenotettavan vaihtoehdon dynaamiselle maastolle, jos muita ominaisuuksia ei välttämättä tarvita kuin visuaalinen vaikutelma maastosta. Shaderien avulla voi toteuttaa yksinkertaisia törmäystarkastuksia helposti.

7.4.1.2 Valmistelu 3d-mallinnusohjelmassa

Casessa dynaaminen maasto on toteutettu täysin näytönohjainshadereillä. Prosessi lähtee liikkeelle siitä, että pohjaksi pitää mallintaa tasainen pinta (plane). Pinta pitää olla jaettu tasaisin välein niin moneen osaan kuin korkeuskartan tarkkuus tulee olemaan. Toisin sanoen verteksien pitää osua tekstuurissa jokaisen pikselin keskelle. Hyvä tarkkuus on esimerkiksi 128 x 128 verteksiä, joka on jo yhteensä 16384 verteksiä eli 32258 pintakolmiota. Seuraavaksi 3d-mallinnusohjelmassa on hyvä siirtää luodun planen keskipiste origoon ja samoin pivot-piste. Lisäksi pinnan mitat olisi hyvä säätää 1 x 1 kokoiseksi, koska sen voi skaalata lopullisessa ohjelmassa oikeaan kokoonsa. Skaalauksen voi tehdä verteksi shaderissä tai antamalla translaatiomatriisiin halutun skaalan. Kun pinnan UV-kartoituksen tekee 3d-mallinnusohjelmassa valmiiksi, se auttaa korkeuskartan kohdistamisessa. UV-kartta pitää siis olla ainoastaan projektio sen yläpuolelta, jossa uloimmat kulmat osuvat 0 ja 1 sijainteihin UV-kartassa.

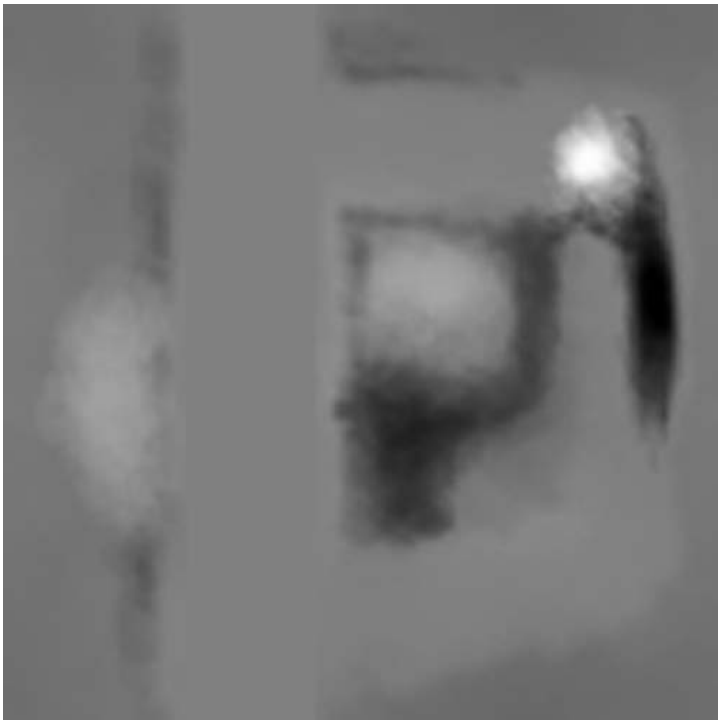


KUVA 34. Ruudukko -objekti (Viljanen 2012)

7.4.1.3 Korkeuskartan käyttöönotto ilman muokattavuutta

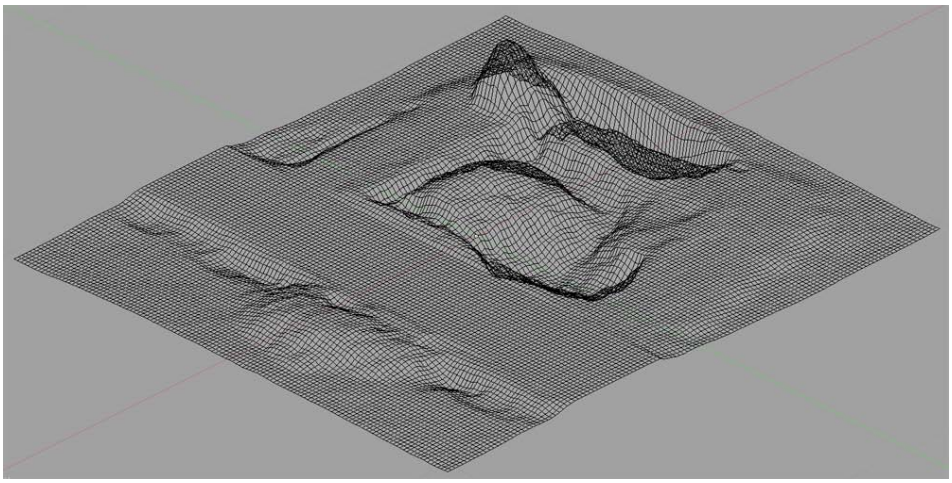
Objektin voi nyt siirtää haluamaan ohjelmaan, jossa varjostimien käyttö on mahdollista. Casessa ohjelmana on Quest3d ja käytössä ShaderModel 3.0 shaderit. 3d-objektin data pitää olla linkitettynä HLSL-objektiin, joka taas linkitetään Surface objektiin. HLSL-objektin sisään kirjoitetaan shader-koodi. Koodissa olennaisin

funktio on tex2Dlod(mapsampler, UV), joka mahdollistaa pääsyn tekstuuriin verteksi shaderissä. Edellämainittu funktio palauttaa nelikomponenttisen vektorin, jotka viittaavat tekstuurin kanaviin. Yleensä käytössä on vain harmaasävyinen korkeuskartta, kuten kuvassa 35 on esitetty. Sellaisen korkeuskartan datasta eli värikanavista voidaan hyödyntää esimerkiksi ainoastaan ensimmäistä kanavaa (r tai x). Kanava sijoitetaan seuraavaksi sijainnin toiseen komponenttiin eli y:n paikalle, joka on verteksin korkeus. Näin maasto saa muodot korkeuskartalta. Jatkokehitettävänä ratkaisuna korkeuskartan formaatti tulisi huomioida tarkempana 32-bittisenä harmaasävykarttana tai hyödyntää 8 bit/kanava RGB-kuvan jokaista kanavaa, jolloin tarkkuus olisi 24 bittiä. Näistä kohdistetuin ratkaisu olisi varmasti ensimmäinen.



KUVA 35. Korkeuskartta eli heightmap (Viljanen 2012)

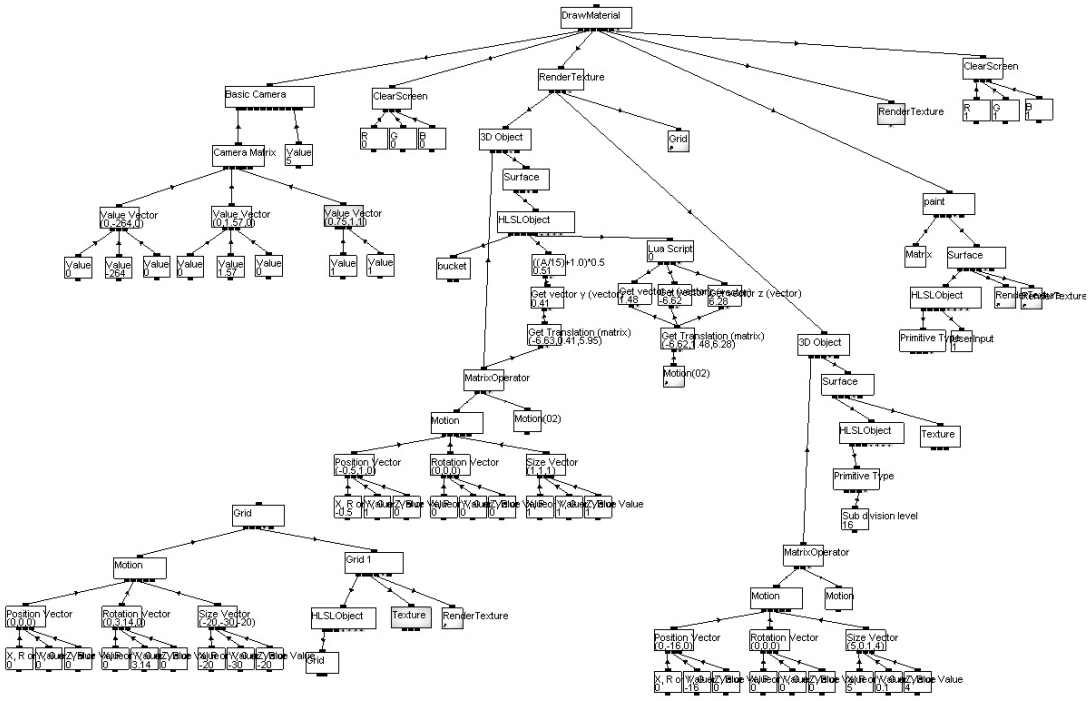
Jos korkeuskartan keskiharmaa halutaan saada pysymään nollatasolla, aiempi operaatio ei vielä riitä. Se voidaan ratkaista siten, että korkeuskartasta saadut korkeudet pitää vähentää 0.5:llä ja kertoa kahdella. Prosessi siirtää pinnan keskiharmaan nollatasolle, jolloin alimmillaan käydään -1:ssä ja korkeimmillaan 1:ssä. Jättämällä kahdella kertominen pois säilyttää korkeuseron edelleen samana, mutta keskiharmaa osuu silti nollatasolle. Käyttöönotettu korkeuskartta maaston pinnassa näyttäisi esimerkiksi kuvan 36 kaltaiselta.



KUVA 36. Korkeuskartta käytössä (Viljanen 2012)

7.4.1.4 Toteutus maaston muokkaukseen

Maaston muokkauksen toteutukseen on olemassa useita erilaisia lähestymistapoja ja ratkaisuja. Valmiita ratkaisuja saattaa löytyä Internetistä, mutta tässä tapauksessa osittain toimivan ratkaisun kehitti pienellä pohdinnallakin. Olennaisinta kaivuuoperaatiossa on ajatella prosessia kuin piirto-ohjelmana. Lyhyesti piirto-ohjelman logiikka on olla tyhjentämättä kuvaa jokaisen näytönohjaimen piirtämän ruudun jälkeen. Kun ruutua ei tyhjennetä, jokainen kuvan eteen tuotu näkyvä objekti jättää itsestään jäljen. Perinteisessä piirto-ohjelmassa siveltimen näkyvyys on yleensä toteutettu hiiren klikkauksella tai piirtopöydän kynän kärjen painalluksella. Jotta reaaliaikainen renderöinti ja shaderit saadaan toimimaan kuvaa tyhjentämättä, aihetta pitää lähestyä tekstuuriin renderöinnin kautta. Quest3d-ohjelmassa toteutetun rakenteen voi hahmottaa kuvasta 37 ja sen alla selostetuista vaiheista.



KUVA 37. Dynaamisen maaston kaivuu Quest3d:ssä (Viljanen 2012)

Ennen lopullisen maaston esittämistä pitää toteuttaa eräänlainen piirto-operaatio, jonka vaiheet casessa menevät seuraavanlaisesti:

- Tyhjennetään ruutu.
- Luodaan korkeuskarttaa hyödyntävästä maastosta kopio.
- Maaston culling käänteiseksi eli käänteiset normaalit.
- Luodaan kamera maaston pinnan alapuolelle kuvaamaan ylöspäin.
- Luodaan kaivinkoneen kauhasta kopio.
- Kaivinkoneen kauha toimii eräänlaisena siveltimenä, joka näytetään kauttaaltaan tasaisen vihreänä.
- Maaston pinnan läpäisevät kauhan vihreät osat renderöidään tekstuuriin.
- Renderöidyn tekstuurin vihreän kanavan ”sivellin” (kauha) piirtää tekstuuri T:n punaiseen kanavaan voimakkuudella, joka vastaa kauhan korkeutta.
- Tekstuuri T pidetään tallessa tekstuuriin renderöinnin avulla.
- Tyhjennetään ruutu ja jatketaan muihin renderöintioperaatioihin.

Kaivuutekstuuri piirtyy kuvan 38 kaltaisena. Punaiset jäljet merkitsevät kohtia, joita on kaivettu. Siniset jäljet merkitsevät telaketjujen jälkiä.

KUVA 38. Kaivuujäljet punaisella ja telaketjut sinisellä (Viljanen 2012)

7.4.1.5 Korkeus- ja kaivuukartan yhdistäminen

Karttojen kohdistaminen tulee olla otettu huomioon jo kaivuukartan renderöinnissä. Kohdistamista voi hienosäätää edelleen shaderohjelmassa. Luvussa 7.4.1.3 mainittua shadertoteutusta jatketaan edelleen yhdistämällä kaivuukartta korkeuskarttaan. Yhdistämisen voi toteuttaa esimerkiksi kuvassa 39 näkyvällä koodilla. Yhdistäminen on toteutettu erilliseen funktioon, joka suoritetaan verteksi shaderin puolella. Lopputuloksena kaivaminen ja kaivuukartan osio voi näyttää siltä kuin kuvasta 40 näkyy.

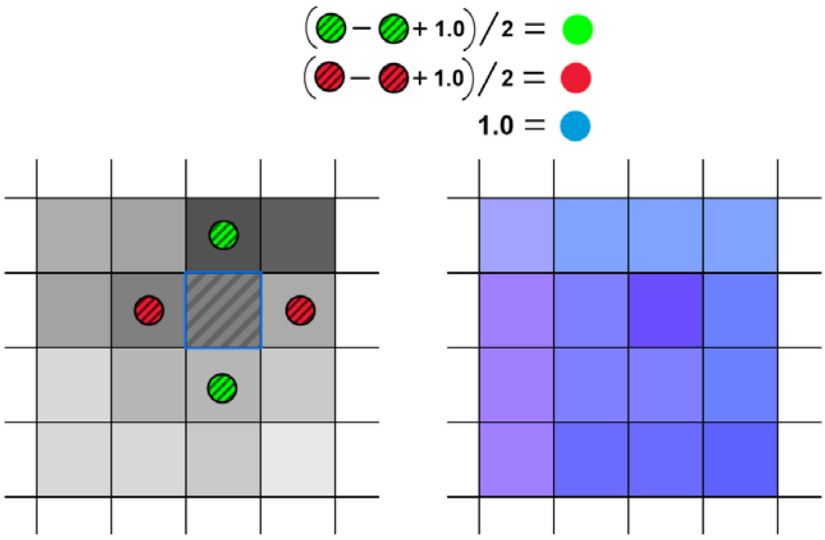
```
float4 heightCombine(float4 position, float2 texCoord)
{
	float4 HeightMap = tex2Dlod(HeightSampler, float4(texCoord.xy,0,0));
	float4 DiggedMap = tex2Dlod(DiggedSampler, float4(texCoord.x,-texCoord.y,0,0));

	if(DiggedMap.r > 0.005)
	{
		position.y = DiggedMap.r-0.5;
	}
	else
	{
		position.y = HeightMap.y-0.5;
	}

	return position;
}
```

KUVA 39. Korkeus- ja kaivuukarttojen yhdistäminen (Viljanen 2012)

KUVA 40. Kaivuri kaivamassa maastoa (Viljanen 2012)

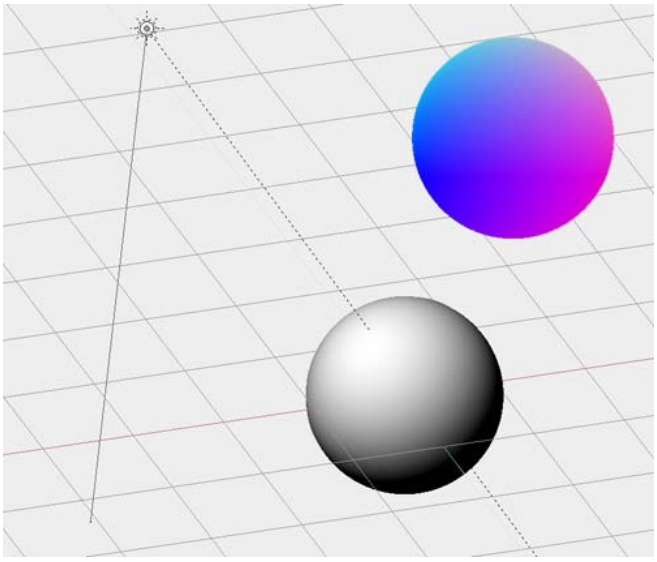
7.4.1.6 Maaston valaistuksen valmistelu

Aiemmin luotu maaston pintageometria ei sisällä valmiina verteksien normaaleja. Normaalit ovat olennaisimmassa osassa valaistuksen laskentaa, koska ne kertovat kyseisen pinnan alueen suunnan. Pinnan suuntaa ei voida pelkkien kolmioiden perusteella tietää, sillä olemassa on aina kaksi vaihtoehtoa. Tasopintaisen 3d-objektin normaalit voidaan laskea korkeuskartan perusteella kohtalaisen helposti. Laskenta toteutetaan silloin pikseli shaderissä tulkitsemalla korkeuskartan pikseleitä. Tässä tapauksessa voidaan käyttää ns. ristisuodatus-tekniikkaa, jossa kyseisen pikselin ympäriltä otetaan pikseleitä ja lasketaan keskelle jäävälle pikselille sen tangentiaalinen suunta. Kuva 41 havainnollistaa tämän tyyppisen suodatuksen toimintaperiaatetta. Lisäksi kuvassa 41 on havainnollistettu kanaviin siirrettyä informaatiota ja niiden laskentakaavoja. Jos normaalikartta visualisoidaan, sen pitäisi näyttää väritykseltään saman kuvan oikealle puolelle jäävältä ruudukolta.



KUVA 41. Pikselikohtaisten normaaleiden laskenta korkeuskartan pohjalta (Viljanen 2012)

Normaalien laskennan jälkeen pikseli shaderin ohjelmassa otetaan huomioon valonlähde tai valonlähteet. Yksinkertaisesti valonlähdettä vastakkain asettuvat normaalit tulkitaan valkoisena eli valaistaan ja käänteiset varjon puolelle jäävät ovat mustia. Kuva 42 on esimerkkinä valaistuksesta pallon pinnalla, mutta ajatus on sama geometriasta riippumatta.

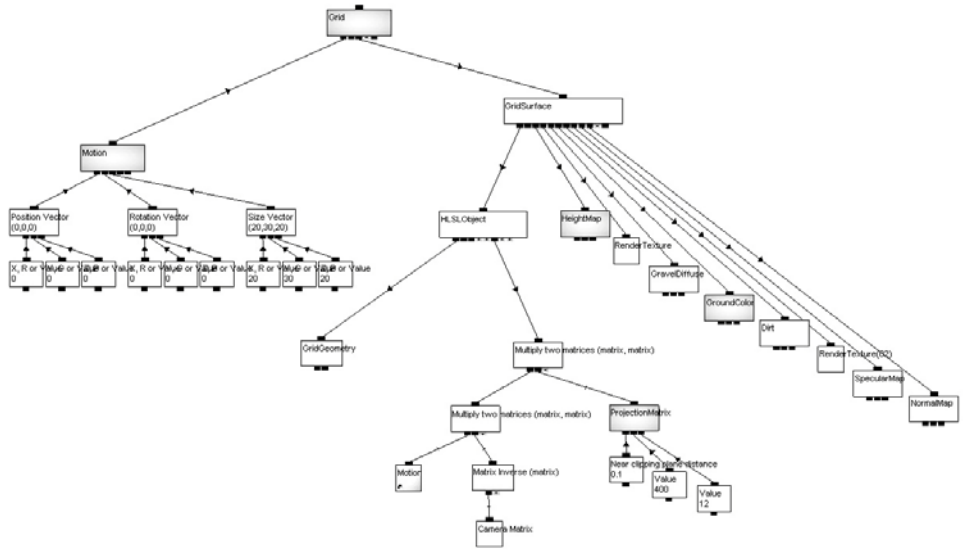


KUVA 42. Valaistu pallo ja normaaleiden tasolla (Viljanen 2012)

7.4.1.7 Maaston pintamateriaalit

Materiaalien kehittäminen on usein pääasiassa pikseli shaderin ohjelmointia ja tekstuureiden valmistelua. Pohjaksi lasketun valaistuksen kanssa voidaan sekoittaa useita erilaisia yksityiskohtia tuovia tekstuureita ns. detail-karttoja. Detail-kartaksi kelpaavat tekstuurit ovat jatkuvia ja saumattoman vaikutelman antavia (tile-kartta). Sellaisia karttoja voidaan luoda maastoksi mm. kohtisuoraan kuvattujen pintojen, kuten hiekan, soran, mullan tai nurmikon pohjalta tai generoimalla. Usein maaston väritystä tehostetaan vielä erillisellä värikartalla, jolla saadaan enemmän variaatiota. Kuvassa 43 nurmikon ja mullan erottaa toisistaan värikartta. Quest3D-ohjelmassa maasto-objekti voidaan esittää kuten kuvassa 44.

KUVA 43. Maaston materiaali (Viljanen 2012)



KUVA 44. Maaston Quest3d rakenne (Viljanen 2012)

7.4.2 Kaivinkoneen pintamateriaali

7.4.2.1 Materiaalin ominaisuuksien kartoitus ja valmistelu

Yleensä pintamateriaalin luonti lähtee liikkeelle materiaalin todellisten ominaisuuksien kartoittamisesta. Ominaisuuksien kartoittamisella voidaan vaikuttaa useimmiten shaderin varaamaan muistin määrään ja suoritusnopeuteen. Se voi tarkoittaa esimerkiksi sitä, että useampien tekstuurien määrä kannattaa rajoittaa shaderissä vain tarpeellisiin tekstuurikerroksiin. Muita ratkaisevia tekijöitä on ottaa huomioon esimerkiksi pinnan heijastavuus ja kiilto-ominaisuudet. Jos näitä ominaisuuksia ei tarvita, niitä ei myöskään kannata shaderiin ohjelmoida.

Pelkkä materiaalin tulkitseminen ei kuitenkaan hyödytä ilman ymmärtämistä teorian ja ohjelmoinnin tasolla. Useimmat tekniikat vaativat lähestymistä vektori- ja matriisi-laskennan oppien kautta. Olennaisinta on kuitenkin antaa vaikutelma, kuin pyrkiä realismiin. Internet ja alan kirjallisuus ovat hyviä lähteitä materiaalien ominaisuuksien ja erilaisten efektien ratkontaan. Omien kokeilujen kautta voi myös onnistua.

7.4.2.2 Valaistus ja pinnan epätasaisuudet

Verteksien normaalit ovat aina shaderien tärkeimmässä osassa, kun toteutetaan mm. pinnan valaistus. Verteksien normaalit kertovat kyseisen kulman suunnan ja sitä kautta pintojen suunnat. Kun 3d-objektin pinnat sidotaan kohti valonlähteen suuntavektoria, voidaaan sen pohjalta tulkita, mitä pintoja kohti valo osoittaa. Tässä pitää huomioida toistaiseksi heittovarjojen puuttuminen, joten valosta katsoen objektin etupintojen takana olevat pinnat käyttäytyvät kuten etualallakin.

Valaistuksen voi toteutaa joko verteksi shaderillä tai pikseli shaderillä. Verteksivalaistuksen (Vertex lighting) etu on nopeus, mutta se tuo helposti lowpoly-mallien kulmat esille, jolloin laatu jää huonoksi. Verteksivalaistuksessa valonlähde otetaan huomioon jo verteksiohjelman koodissa, joka muunnetaan varjostuksen harmaan sävyihin. Rasteroidut valaistuksen vektorit vastaavat lopullista valaistusta ja jälkikäsittelymahdollisuudet jäävät pieniksi.

Pikselivalaistuksessa (Per-pixel lighting) laatu saadaan paremmaksi, mutta suoritettavuudeltaan se on hieman raskaampi. Pikselivalaistuksessa käytetään rasteroituja normaaleita ja tehdään valonlähteeseen liittyvät operaatiot vasta pikseliohjelmassa. Lisäksi pikselivalaistukseen voidaan lisätä pinnan epätasaisuuksia käyttäen normaalikarttaa (normalmap). Normaalikarttoja on erilaisia generointitavasta riippuen, mutta näistä yleisin on ns. tangent normalmap. Tangentti-muotoinen kartta toimii vaikuttamalla jo rasteroituihin pikselivalaistuksen normaaleihin hienoin muutoksin. Valonlähde otetaan huomioon ja sen pohjalta luodaan harmaasävyinen varjostus. (Wikipedia 2012c)

7.4.2.3 Tekstuurikerrosten huomiointi ja lopputulos

- **Värityksen kartoitus (Diffuse)**

Kun pintaan halutaan luoda vaikutelma sen värityksestä, liasta, naarmuista ja muista yksityiskohdista, diffuse-tekstuuri luo tämän vaikutelman. Toisaalta lika ja naarmut voitaisiin toteuttaa dynaamisemmin luomalla niille oma tekstuuri.

Usein diffuse on toteutettu hajavaloa simuloivan Ambient Occlusion renderöinnin pohjalle. Ohjelmoinnin ja matematiikan tasolla diffuse-tekstuuri yhdistetään kertomalla (Multiply-sekoitustila) harmaasävyisen valaistuksen kanssa.

- **Vaikutelma ympäristöstä (Environment Mapping)**

Todellisuudessa kiiltäväksi maalattu pinta on heijastava, joten ympäristön heijastavia elementtejä pitää saada mukaan. Realistinen heijastus monen muotoisille pinnoille on kuitenkin liian raskasta toteuttaa reaaliaikaisena. Heijastukseen voi käyttää apuna kyseistä ympäristöä matkivaa ympäristön kartoittamista eli ns. environment mapping -tekstuuria. Kartta voi olla tallennettu esimerkiksi pallomaiseen (spherical) muotoon, josta haetaan kameran ja pintojen normaaleiden kautta näkyvät alueet. Tekniikka ei ole täysin realistinen, mutta sillä saa vaikutelman esimerkiksi ympäröivästä tilasta.

- **Heijastukset (Reflection Mapping)**

Useimmiten heijastus ei ole tasaista, vaan siihen vaikuttavat monet pienet tekijät, kuten kuluneet kohdat, liat ja erilaiset materiaalit. Peilin tai kromin tapauksessa pinta voisi olla täysin tasaisen heijastava, mutta pienin yksityiskohdin näistäkin saadaan näyttävämpiä. Heijastuksen määrään voidaan vaikuttaa heijastuskartan avulla pikselikohtaisesti.

- **Erillinen maski läpinäkyviin kohtiin**

Casessa ikkunat on yhdistetty samaan objektiin, mikä ei ole suorituskyvyn kannalta paras ratkaisu. Optimoidussa ratkaisussa ikkunat olisivat erillään muusta kaivinkoneesta ja käyttäisivät omaa shaderiään. Siitä riippumatta läpinäkyviin alueisiin voi toteuttaa oman maskin tai yhdistää se jonkin muun kartan (yleensä diffuse) alpha-kanavaksi. Maskilla vaikutetaan pikseleiden läpinäkyvyyteen pikselikohtaisin prosentein ts. yksikkömuodossa.

Valmis kaivinkoneen pintamateriaali näyttää casessa kuvan 45 mukaiselta. Pinnan kevyet epätasaisuudet luovat pintaan rouheamman vaikutelman. Huijatuja heijastuksia voi tuskin havaita still kuvasta, muuten kuin hydraulisylintereissä. Shaderohjelmana lopputulos näyttää kuvassa 46 näkyvältä koodilta.

KUVA 45. Ruutukaappaus kaivinkoneen pintamateriaalista (Viljanen 2012)

```
// ExcavatorMaterial Shader
// Vesa Viljanen 24.2.2012

int UseTangent  : UseTangent;

float4x4 objectOrient : CHANNELMATRIX0;

texture DiffuseMap : TEXTURE0;
sampler2D DiffuseMapSampler = sampler_state
{
        Texture = <DiffuseMap>;
        MinFilter = LINEAR;
        MagFilter = LINEAR;
        MipFilter = LINEAR;
};

texture NormalMap : TEXTURE1;
sampler2D NormalMapSampler = sampler_state
{
        Texture = <NormalMap>;
        MinFilter = LINEAR;
        MagFilter = LINEAR;
        MipFilter = LINEAR;
};

texture EnvMap : TEXTURE2;
sampler2D EnvMapSampler = sampler_state
{
        Texture = <EnvMap>;
        MinFilter = LINEAR;
        MagFilter = LINEAR;
        MipFilter = LINEAR;
};

texture RefMap : TEXTURE3;
sampler2D RefMapSampler = sampler_state
{
        Texture = <RefMap>;
        MinFilter = point;
        MagFilter = point;
        MipFilter = point;
};

float3 light_dir[8]         : LIGHTDIR;

float4x4 wvp : WorldViewProjection;
float4x4 vp : ViewProjection;

struct a2v {
   float4 position              : POSITION;
   float3 tangent               : TANGENT;
   float3 normal                : NORMAL;
   float2 texCoord              : TEXCOORD0;
};

struct v2f {
   float4 position              : POSITION;
   float2 texCoord              : TEXCOORD1;
   float3 worldTangent          : TEXCOORD2;
   float3 worldBinormal         : TEXCOORD3;
   float3 worldNormal           : TEXCOORD4;
};

// Verteksi varjostin //
v2f av(a2v In)
{
   v2f Out = (v2f)0;
   Out.position = mul(In.position, wvp);
   Out.worldNormal = In.normal;
   Out.worldTangent = In.tangent;
   Out.worldBinormal = normalize(cross(Out.worldNormal, Out.worldTangent)).xyz;
   Out.texCoord = In.texCoord;

   return Out;
}

// Pikseli varjostin //
float4 af(v2f In) : COLOR
{
float4 DiffuseMap = tex2D(DiffuseMapSampler, In.texCoord.xy);
float4 RefMap = tex2D(RefMapSampler, In.texCoord.xy);
float3 NormalMap = tex2D(NormalMapSampler, In.texCoord.xy);
NormalMap = NormalMap.xyz * 2 - 1;

float3 Nn = normalize(In.worldNormal);
float3 Tn = normalize(In.worldTangent);
float3 Bn = normalize(In.worldBinormal);
float3 WorldXform = ( Nn * NormalMap.z ) + (NormalMap.x * Tn ) + ( NormalMap.y * -Bn);

float4 Shading = max(0.5, dot(normalize(light_dir[0]), -normalize(mul(WorldXform, objectOrient))));
float4 EnvMap = tex2D(EnvMapSampler, normalize(mul(WorldXform, wvp))*0.5+0.5);
RefMap = EnvMap*EnvMap*RefMap*0.2;

return Shading*DiffuseMap+RefMap;
}

technique Complete
{
   pass main
   {
      VertexShader = compile vs_3_0 av();
      PixelShader = compile ps_3_0 af();
   }
}
```

KUVA 46. Shaderohjelma kaivinkoneen pintamateriaaliin (Viljanen 2012)

7.4.3 Telaketjut

Telaketjujen toteutukseen on kaksi toisistaan poikkeavaa ratkaisua. Enemmän näytönohjaimen resursseja syövä vaihtoehto olisi toteuttaa telaketju geometrisesti eli telaketju telakenkineen. Kevyempi ja vähän vähemmän näyttävä ratkaisu on toteuttaa telaketjun liike liikuttamalla tekstuuria verteksi shaderin avulla. Jälkimmäinen usein riittää tarkoitukseen ja antaa kelvollisen lopputuloksen. Vastaan tulevia ongelmia voi olla mm. tekstuurin liikkeen synkronointi todellisen liikkeen suhteen. Tekstuurin sopivan liikkeen voi löytää liikuttelemalla kaivinkonetta edes takaisin ja hakemalla mahdollisimman lähelle osuva nopeuskerroin. Testailussa kannattaa ottaa huomioon se seikka, että sopiva suhdeluku on silloin, kun maaston kuvio verraten telaketjun kuvioon pysyy paikoillaan eikä liu. Kuvassa 47 on esitetty shader -ohjelman toteutus.

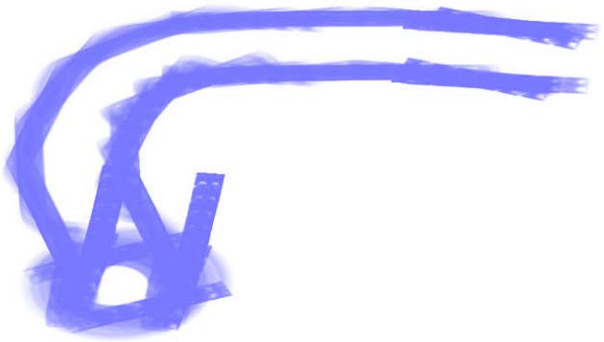
```
// ExcavatorTracks Shader
// Vesa Viljanen 13.3.2012

int UseTangent  : UseTangent;

texture DiffuseMap : TEXTURE0;
sampler2D DiffuseMapSampler = sampler_state
{
        Texture = <DiffuseMap>;
        MinFilter = LINEAR;
        MagFilter = LINEAR;
        MipFilter = LINEAR;
};

texture NormalMap : TEXTURE1;
sampler2D NormalMapSampler = sampler_state
{
        Texture = <NormalMap>;
        MinFilter = LINEAR;
        MagFilter = LINEAR;
        MipFilter = LINEAR;
};

texture GradeMap : TEXTURE2;
sampler2D GradeMapSampler = sampler_state
{
        Texture = <GradeMap>;
        MinFilter = LINEAR;
        MagFilter = LINEAR;
        MipFilter = LINEAR;
};

texture Diffuse2Map : TEXTURE3;
sampler2D Diffuse2MapSampler = sampler_state
{
        Texture = <Diffuse2Map>;
        MinFilter = LINEAR;
        MagFilter = LINEAR;
        MipFilter = LINEAR;
};

texture EnvMap : TEXTURE4;
sampler2D EnvMapSampler = sampler_state
{
        Texture = <EnvMap>;
        MinFilter = LINEAR;
        MagFilter = LINEAR;
        MipFilter = LINEAR;
};

float3 light_dir[8]         : LIGHTDIR;

float4x4 wvp : WorldViewProjection;
float4x4 vp : ViewProjection;

float trackleftPos : CHANNELVALUE0;

struct a2v {
        float4 position         : POSITION;
        float3 tangent          : TANGENT;
        float3 normal           : NORMAL;
        float4 blendWeights     : BLENDWEIGHT;
        float4 blendIndices     : BLENDINDICES;
        float2 texCoord         : TEXCOORD0;
};

struct v2f {
    float4 position             : POSITION;
    float2 texCoord             : TEXCOORD1;
    float3 worldTangent         : TEXCOORD2;
    float3 worldBinormal        : TEXCOORD3;
    float3 worldNormal          : TEXCOORD4;
    float3 depthMap             : TEXCOORD5;
};

v2f av(a2v In)
{
        v2f Out = (v2f)0;
        Out.position = mul(In.position, wvp);
        Out.worldNormal = In.normal;
        Out.worldTangent = In.tangent;
        Out.depthMap = Out.position.xyz;
        Out.worldBinormal = normalize(cross(Out.worldNormal, Out.worldTangent)).xyz;
        Out.texCoord = In.texCoord;
        return Out;
}

float4 af(v2f In) : COLOR
{
        float4 DiffuseMap = tex2D(DiffuseMapSampler, In.texCoord.xy);
        float4 Diffuse2Map = tex2D(Diffuse2MapSampler, float2(In.texCoord.x+trackleftPos, In.texCoord.y));
        float3 NormalMap = tex2D(NormalMapSampler, float2(In.texCoord.x+trackleftPos, In.texCoord.y));
        NormalMap = NormalMap.xyz * 2 - 1;
        float3 Nn = normalize(In.worldNormal);
        float3 Tn = normalize(In.worldTangent);
        float3 Bn = normalize(In.worldBinormal);
        float3 WorldXform = ( Nn * NormalMap.z ) + (NormalMap.x * Tn ) + ( NormalMap.y * -Bn);
        float4 GradeMap = tex2D(GradeMapSampler, float2(-0.5+1.497*max(0.5, dot(normalize(light_dir[0]), -normalize(WorldXform))), 0.0));
        float4 EnvMap = tex2D(EnvMapSampler, normalize(mul(WorldXform, wvp))*0.5+0.5);
        return GradeMap*DiffuseMap*Diffuse2Map+(EnvMap*EnvMap*Diffuse2Map.r);
}

technique excavatorTracks
{
    pass main
    {
        VertexShader = compile vs_3_0 av();
        PixelShader = compile ps_3_0 af();
    }
}
```

KUVA 47. Telaketjun shaderohjelma (Viljanen 2012)

Tekstuurilla toteutetussa versiossa pitää ensin luoda hihnamainen lenkin tekevä objekti 3d-mallinnus ohjelmassa. Mallinnuksessa tärkeimpänä vaiheena on objektin UV-kartan pintojen muokkaus täysin avatuiksi ja mahdollisimman vähillä vääristymillä. UV-kartan ns. suikaleet asetellaan päällekkäin tai vierekkäin, siten että pitkät sivut ovat vierekkäin. UV-kartan tulee olla sen kaltainen, että koko karttaa liikuttamalla yhden akselin suuntaan telaketjun tekstuuri liikkuu telaketjun pinnassa. Telaketjun tekstuuri pitää tehdä jatkuvaksi laidasta laitaan. Kuvassa 48 näkyy kaivinkoneen telaketjut.

KUVA 48. Telaketjut kaivinkoneessa (Viljanen 2012)

Vastaavalla tekniikalla kuin dynaamisen maaston muokkauksessa, saadaan toteutettua telaketjujen jäljet maaston pintaan. Case tapauksessa se on toteutettu maaston muokkauksen yhteydessä käyttämällä tekstuuriin tyhjäksi jäävää sinistä kanavaa. Kuvassa 49 telaketjujen jäljet on esitetty renderöidyn tekstuurin tasolta.



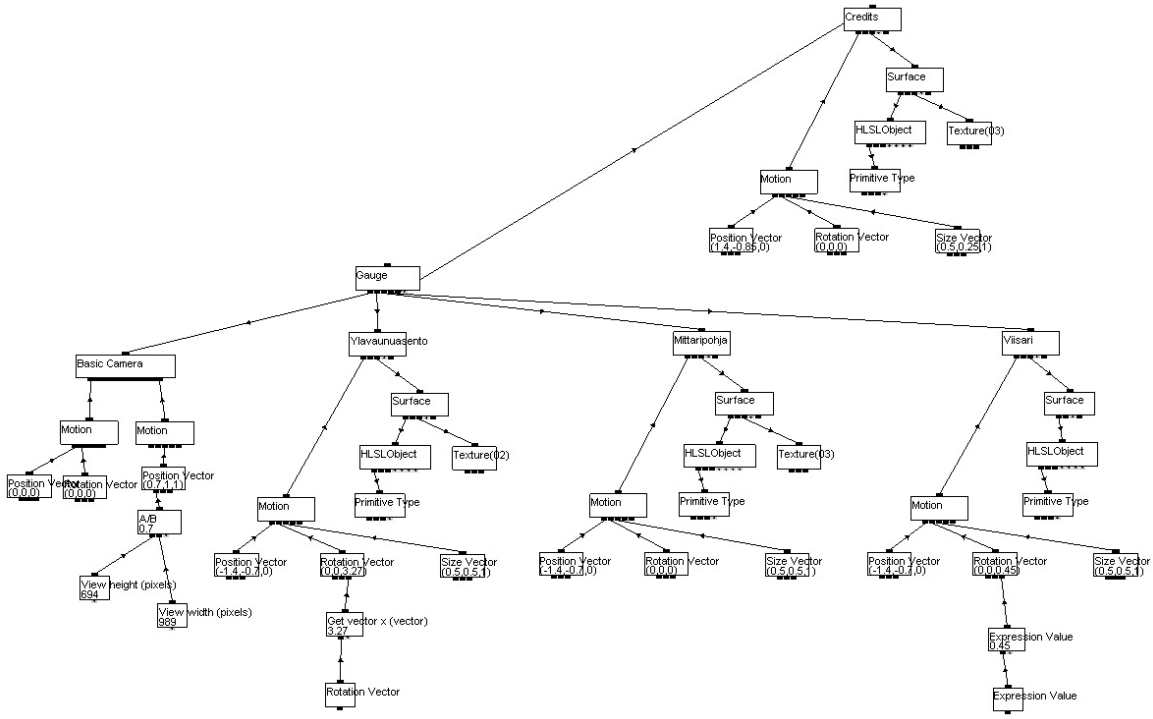
KUVA 49. Maastoon piirtyvät telaketjun jäljet (Viljanen 2012)

7.4.4 Mittaristo (HUD)

Virtuaaliympäristön lisäksi monesti tarvitaan erilaisia tulosteita, graafeja tai mittareita. Ne ovat tärkeitä, kun halutaan tietää jokin tieto tarkempana tai tulkita muuten kuin se olisi mahdollista virtuaaliympäristöstä itsestään.

Mittaristoa on hyvä lähteä kehittämään kartoittamalla esitettävät tiedot, mistä ohjelman kohdasta tieto löytyy ja millaista tieto on. Usein tieto joka liikkuu ohjelman takana, ei ole suoraan kelvollista omaan virtuaalimittaristoon. Silloin ratkai-

suna on luoda tarvittavia matemaattisia ja/tai loogisia operaatioita. Quest3d:ssä mahdollisuudet näihin operaatioihin eli lukujen muutoksiin ovat Expression Value tai Lua-script. Hierarkkinen rakenne Quest3d-ohjelmassa on kuvan 50 näköinen.



KUVA 50. Mittariston rakenne Quest3d:ssä (Viljanen 2012)

Case tapauksessa tarvittava tieto on moottorin kierrosluku ja kaivinkoneen ylävaunun rotaatio suhteessa sen alavaunuun. Moottorin kierroslukua ei tässä tapauksessa esitetä numeerisesti, joten tärkeämpää on saada kuvassa 51 näkyvä viisari ja pohja toimimaan näennäisesti keskenään. Ylävaunun suunnasta kertova lukuarvo kuitenkin kelpaa suoraan mittariston taustalla olevaan indikaattoriin, koska kummassakin objektissa on hyödynnetty Quest3d:n valmista translaatiomatriisin rotaatiovektoria. Ylävaunun suunnasta kertova indikaattori näkyy kuvassa 51 koko mittariston pohjalla. Kun vihreän alueen keskikohta on samassa linjassa mittariston yläosassa olevan pystyviivan kanssa, kaivinkoneen ajosuunta on normaali. Jos ylävaunu on käännetty tästä 180 astetta, ajosuunta on käänteinen.

KUVA 51. Mittaristo (Viljanen 2012)

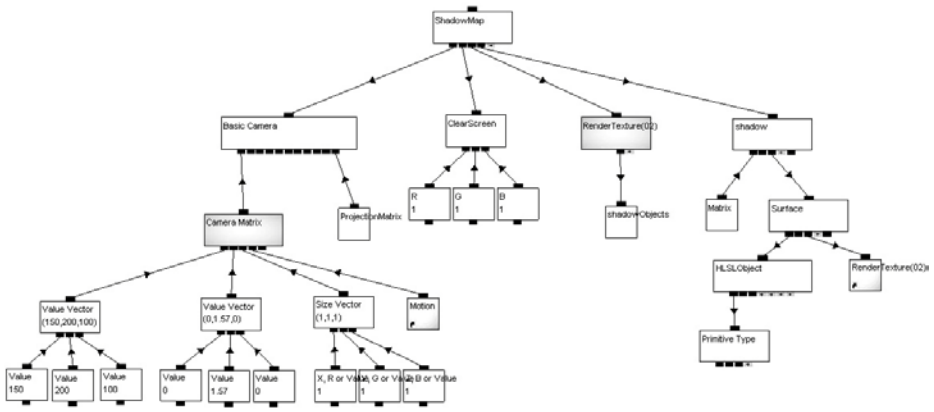
Pohjimmiltaan mittaristossa on päällekkäin kaksi neliöpintaa, joissa toisessa suuntaindikaattorin tekstuuri ja sen päällä olevassa pinnassa mittariston pohjan tekstuuri. Lisäksi viisari on toteutettu neliöpintaa hyödyntäen, mutta verteksi shaderillä muotoilemalla. Viisarin muodon kaventaminen kohti sen kärkeä on toteutettu kuvassa 52 näkyvällä koodilla. Koodissa x-akselin arvo vaikuttaa vähentävästi y-akseliin. Lopuksi verteksit siirretään niin, että haluttu viisarin keskipiste sijoittuu viisarin keskilinjaan ja lähemmäs paksua päätyä.

```
vertexToPixel vOhjelma(assemblyToVertex In)
{
	vertexToPixel Out = (vertexToPixel)0;
	In.position.x *= 0.03*(In.position.y*0.5+0.5);
	In.position.y *= 0.35;
	In.position.y -= 0.08;
	Out.position = mul(In.position, wvp);

	return Out;
}
```

KUVA 52. Viisarin verteksiohjelma toteutus. (Viljanen 2012)

7.4.5 Varjot (Shadow Mapping)

Kolmiulotteisuuden tehostamiseen, valonlähteiden olemassa oloon ja realistisemman lopputuloksen saavuttamiseen auttaa etenkin varjot. Varjojen luonti vaatii aiemmista shadereistä poikkeavaa lähestymistapaa ja muutenkin hieman enemmän perehtymistä shadereiden toteutustapoihin. Quest3d-ohjelmassa varjojen renderöinnin voi toteuttaa kuvan 53 kaltaisella rakenteella.

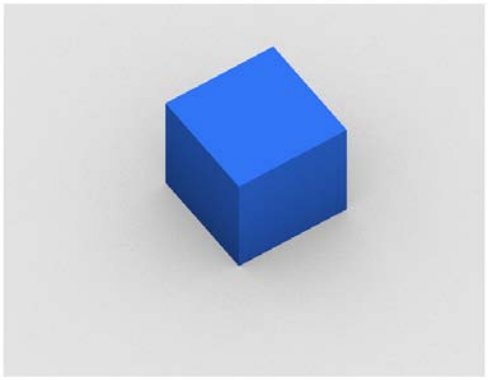


KUVA 53. Shadow Mapping -rakenne Quest3d:ssä (Viljanen 2012)

Olennaisena osana prosessia ovat tekstuuriin renderöinti ja projisointi. Projisointiin vaikuttaa valonlähteen tyyppi, joita yleensä ovat piste, spotti tai suorasuuntainen valonlähde ts. aurinko. Tässä tapauksessa valonlähteeksi on valittu auringon valo. Auringon voisi ajatella helposti todella isona pistemäisenä valon lähteenä. Sitähän se oikeastaan onkin, mutta ihmisen mittakaavassa maapallon pinnalla pistemäisyyttä ei varjoista pysty havaitsemaan. Erona näiden kahden valonlähteen välillä on varjojen keilamaisuus tai suoruus. Keilamaisuutta voi verrata suoraan kameralla läheltä otettuun kuvaan, jossa perspektiivi erottuu selvästi. Suorasuuntaisessa valossa kameralla on kuvattu kaukaista kohdetta, jolloin perspektiiviä ei tule.

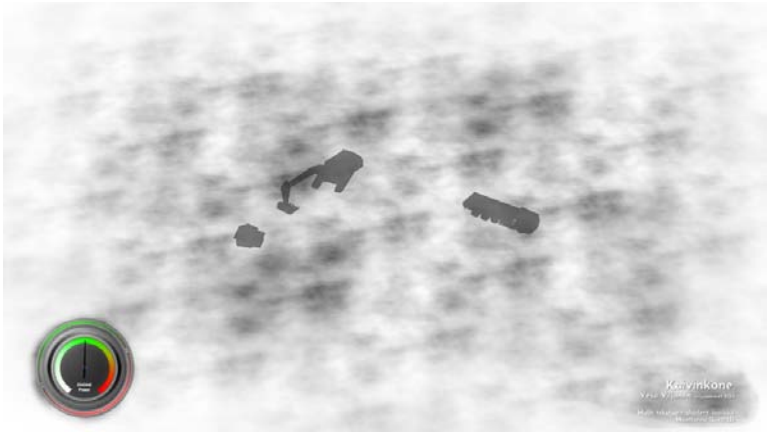
Shadow mapping tekniikassa valonlähde ajatellaankin eräänlaisena kamerana ja se on syy miksi valonlähteen tyyppi pitää tulkita oikein. Kuvassa 54 sininen kuutio havainnollistaa ajatusta, miten kuutio luo varjon vaalealle pinnalle omalla siluetillaan. Tätä renderöityä kuvaa kutsutaan varjokartaksi eli shadowmapiksi. Taustapuskuriin renderöity varjokartta tallennetaan tekstuuriin ja puskuri

tyhjennetään. Seuraavaksi puskuriin koostetaan esimerkiksi varsinaista virtuaaliympäristöä.



KUVA 54. Varjokartan projektio (Viljanen 2012)

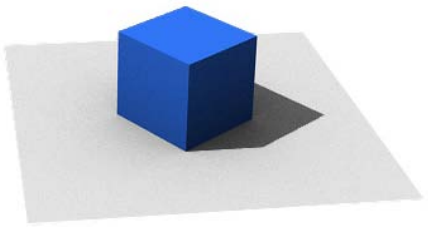
Renderöity varjokartta projisoidaan seuraavaksi varjon vastaanottavaan tai vastaanottaviin objekteihin. Tässä esimerkissä vastaanottava objekti on vaalea taso. Varjokartta pitää saada piirtymään tasoon, joten varjokarttaa käytetään sen yhtenä tekstuurina. Todellisuudessa varjokartta voisi näyttää kuvassa 55 näkyvältä harmaasävyisiltä silueteilta. Kuva on ruutukaappaus casesta, jossa erottuvat kaivinkone, kuorma-auto, varasto ja pilvet.



KUVA 55. Projisoitava varjokartta eli Shadow Map. (Viljanen 2012)

Sinisen kuution esimerkissä projisointi vastaanottavalle tasolle tapahtuu luomalla UV-kartta dynaamisesti käyttäen kolmiulotteisia verteksien sijainteja. Tason verteksi shaderissä verteksit projisoidaan UV-kartaksi käyttäen valonlähteen kameramatriisia. Taso itsessään voi sisältää jo oman UV-kartan, mutta se ei ole kelvollinen tässä tapauksessa. Pikseli shaderissä varjokartan UV:na käytetään dynaami-

sesti luotua projisoitua UV-karttaa. Lopputuloksen pitäisi näyttää vastaavalta kuin kuvassa 56.



KUVA 56. Varjo-objektin langettama varjo (Viljanen 2012)

Varjokartoitukseen liittyy kuitenkin useita ongelmia, joihin on olemassa erilaisia ratkaisuja. Yleisimmät ongelmat liittyvät varjokartan tekstuurin koon olessa rajoitetun kokoinen. Rajoituksen aiheuttama epätarkkuus erottuu, kun 3d-ympäristössä liikutaan. Tarkkuusongelmaan on olemassa ratkaisu, jossa renderöidään useampia varjokarttoja kohdistamalla varjokamera kohti virtuaalista liikkujaa zoomauksella. Renderöidyt eri tasoille asettuvat kartat yhdistetään erilaisin algoritmein, jotta karttojen väliin ei jäisi reunoja eli ns. artefakteja joita ei haluta näkyvän.

Toinen ongelma tai haluttu efekti on varjokarttaan luotujen siluettien teräväreunaisuuden pehmennys (Soft Shadows). Varjojen pehmennyksellä voi vaikuttaa varjojen epätarkkuuksiin tai luoda realistisempaa vaikutelmaa. Todellisuudessa varjo muuttuu etäisyyden kasvaessa sumeammaksi ja häilyvämmäksi. Se vaatisi lähestymistavaksi syvyyskartan käytön ja muita kehittyneempiä tekniikoita. Casessa varjojen pehmennys on toteutettu yksinkertaisella ratkaisulla sumentamalla varjoa kokonaisvaltaisesti. Lopputulos varjojen osalta erottuu kuvassa 57.

KUVA 57. Varjo projisoituna maastoon (Viljanen 2012)

7.4.6 Kuvashaderit (Image shaders)

7.4.6.1 Kuvashaderit yleisesti

Kuvashaderit (image shaders) ts. ruutuavaruuden shaderit (screen space shaders) ovat oikeastaan oma aihealueensa. Keskeisenä ajatuksena on toteuttaa efektejä, jotka perustuvat kuvankäsittelyohjelmista tuttuun jälkikäsittelyyn kuten värimäärittelyyn. Toteutukseltaan ne eivät eroa juurikaan muista shadereistä, mutta keskeisimmät käsitteet ovat tekstuuriin renderöinti ja jälkikäsittely pikseli shaderissä.

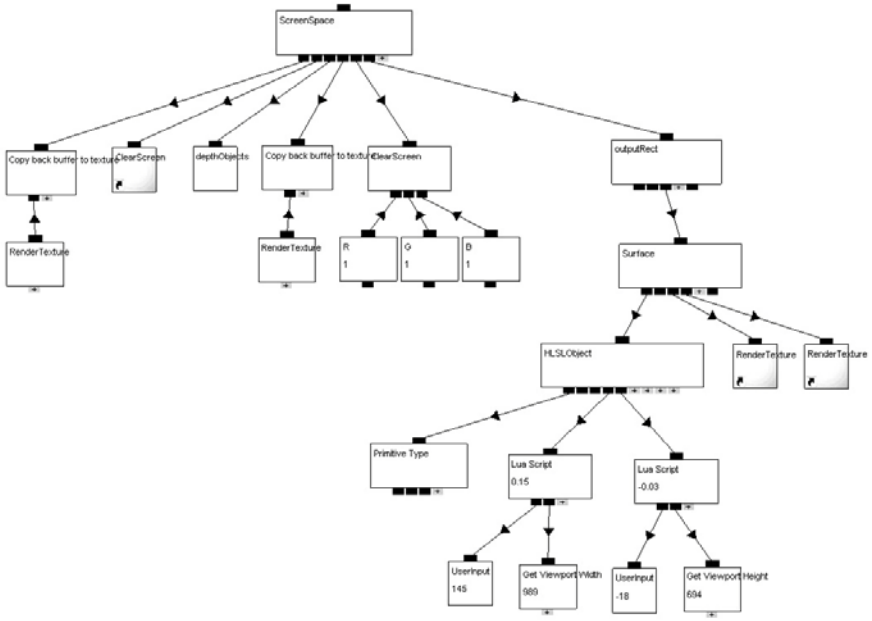
Kuvashadereitä varten käytetään hyväksi taustapuskuriin (back buffer) renderöityä kuvaa, joka normaalisti esitettäisiin jo ruudulla. Juuri ennen tätä vaihetta renderöity kuva luetaan tekstuuriksi ja käsitellään ruutuavaruudessa pikseli shaderillä. Vasta sen jälkeen taustapuskuriin renderöity kuva on valmis ruudulle.

Lisäksi kuvashadereissä käytetty tekniikka mahdollistaa reaaliaikaisen pass kompositoinnin ja muiden monimutkaisempien efektien laskennan ruutuavaruudessa. Esimerkkinä nykyään yleisesti käytössä olevasta efektistä on ns. Screen Space Ambient Occlusion eli SSAO, joka antaa vaikutelmaa hajavalosta ja varjostumista. Sen laskennassa hyödynnetään syvyyskarttaa ja kameraan suhteutettua normaalirenderöintiä.

7.4.6.2 Värimäärittely

Kuvankäsittelyohjelmista, kuten Photoshopista tutut sekoitustilat ja värisäätimet ja efektit voidaan toteuttaa ne pikseli shadereilläkin. Sekoitustilojen teoria pitää vain oivaltaa matemaattisesti. Pikseli shaderit ovat itse asiassa helposti paljon monipuolisempia sekoitustilojen käytössä kuin kuvankäsittelyohjelmat yleensä, koska ohjelmoimalla saa lähes loputtoman tavan käsitellä kuvaa.

Värimäärittelyä tehdessä Photoshop on erinomainen väline prototyyppien tekoon. Ruutukaapattuun kuvaan ilman värimäärittelyjä, voidaan soveltaa erilaisia värien säätötekniikoita. Kun lopputulos on miellyttävä, samaa voi jäljittää ohjelmoimalla shaderohjelmaan. Quest3d-ohjelmassa rakennetta voi toteuttaa, kuten kuvassa 58.



KUVA 58. Värimäärittelyyn luotu Quest3d-rakenne (Viljanen 2012)

Toteutuksen tasolla värimäärittelyssä luodaan suorakaide, joka piirretään koko ruudun kokoisena (ns. pixel perfect square). Verteksi shaderiin ohjelmoidaan tarvittavat operaatiot, jotta suorakaide pysyy staattisena suhteessa kameraan. Idea on vastaava kuin mittariston tai muiden ruutuun piirrettävien 2d-grafiikoiden kanssa. Värimäärittelyssä ja muissa koko ruudun kattavissa efekteissä olennaisinta on hyödyntää tekstuuriin renderöintiä. Alkuperäinen ja käsittelemätön kuva renderöidään tekstuuriin, jota käytetään kameran eteen luodun suorakaiteen tekstuurina. (Engel 2004b, 439-464.)

KUVA 59. Ilman värimäärittelyä, sumua ja syväterävyysefektiä (Viljanen 2012)

KUVA 60. Kuvashaderi käytössä (Viljanen 2012)

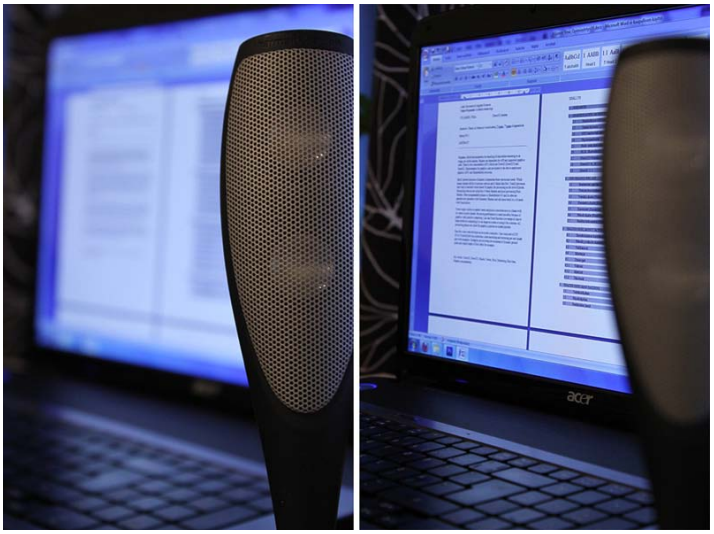
Nyt pikseli shaderin sisällä voidaan ottaa käsittelemätön renderöinti käyttöön, suorittaa värimäärittelyjä värikanavakohtaisesti ja renderöidä muutos lopullisena kuvana. Kuvassa 60 voi huomata eron värimääritellyn ja kuvassa 59 näkyvän määrittelemättömän ruutukaappauksen välillä.

Shader -ohjelmointiin ja jälkikäsittelyyn liittyvää kokeilun tuotosta:

- Suorittamalla kertolasku 1.0:n ylittämällä kertoimella koko vektorille tuo kuvaan lisää kirkkautta. Kertolasku alle 1.0:n vähentää kirkkautta. 1.0:lla kertominen ei muuta kuvaa millään tavoin.
- Summaamalla positiivinen luku vaalentaa kuvaa ja negatiivinen luku (vähennyslasku) tummentaa.
- Potenssilasku muuttaa lukuarvot kasvamaan eksponentiaalisesti mustasta valkoiseen. Näitä voi suoraan verrata ja toteuttaa esimerkiksi differentiaaliyhtälöillä.
- Jokaiselle kanavalle voi tehdä laskutoimituksia erikseen. Sijoittamalla nollan kaikkiin muihin kanaviin paitsi esimerkiksi punaiseen, nähdään pelkästään punaisen kanavan sisältö.
- Korvaamalla vihreä ja sininen punaisen kanavan informaatiolla, nähdään punaisen kanavan sisältö harmaasävyisenä.
- Kanavia voi yhdistää, kuten harmaasävykuvaksi muuttaessa värikanavien kesken lasketaan niiden keskiarvo ja jokainen kanava korvataan saadulla keskiarvolla. **RGB = (R+G+B)/3**

7.4.6.3 Syväterävyys (Depth of Field)

Syväterävyys tai syväepäterävyys on yksi luonnollisimmista ilmiöistä, joita esimerkiksi ihmisen silmä ja kameran linssi muodostavat kuvaan. Ilmiö perustuu siihen, että eri etäisyyksillä olevat kohteet tarkentuvat kuvaan kun linssin polttoväliä muutetaan. Kamerassa syväterävyyteen vaikuttavat lisäksi aukon arvo, jo mainittu polttoväli ja etäisyys kohteesta. Kuvioilmiötä joka epäterävyydessä erottuu, kutsutaan Bokeh efektiksi. Valokuvauksessa Bokeh efektin määrää himmentimen rakenne, jossa himmentimen muodon voi erottaa usein kirkkaiden kohteiden kohdalla selkeänä muotona. Välttämättä tätä efektiä ei ole tarve jäljitellä, vaan laskentatehoa säästävämpänä ja yksinkertaisempana tekniikkana voi käyttää esimerkiksi box blur tai gaussian blur efektejä. Syvyys suunnassa liukuva epäterävyys on efektinä tärkein.



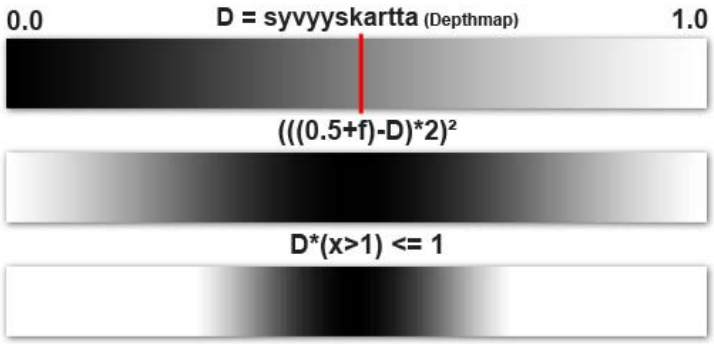
KUVA 61. Kameralla kuvattu esimerkki syväterävyydestä (Viljanen 2012)

Kuvassa 61 on verrattuna kaksi kuvaa, joissa toisessa tarkennus on kauempana ja toisessa lähempänä. Samankaltaisen ilmiön simulointiin on olemassa useita erilaisia ratkaisuja. Shadereillä se on kuitenkin hyvä pitää mahdollisimman suorituskykyisenä. Yleinen englanninkielinen termi on Depth of Field eli DOF.

Ensimmäisenä renderöintivaiheena näkyvistä objekteista luodaan syvyyskartta siihen luodun shaderin avulla tai mahdollisesti lukemalla tieto syvyyspuskurista. Syvyyskartta saadaan väliaikaisesti talteen tekstuuriin renderöinnin avulla. Tässä

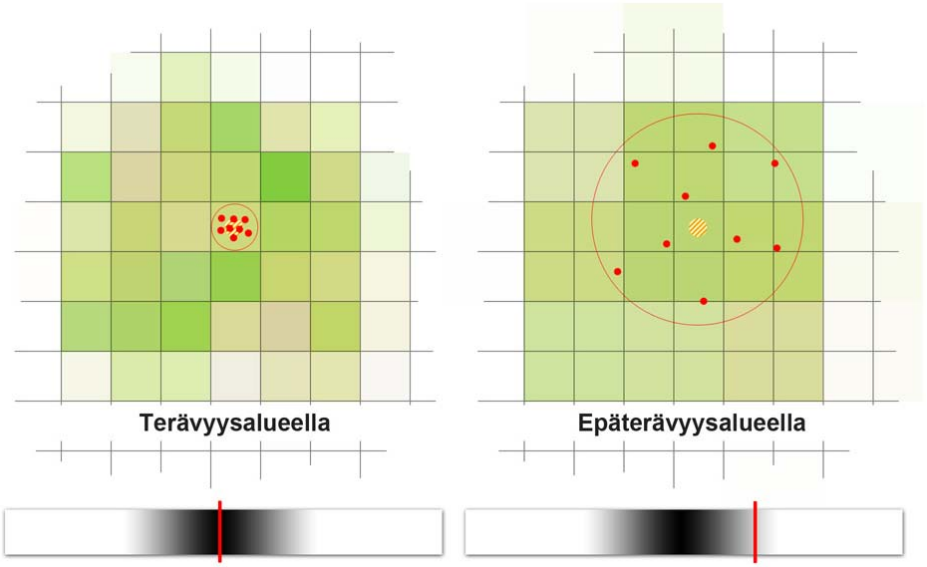
vaiheessa kannattaa huomioida jo luotu värimäärittely, johon syväterävyys on luontevaa yhdistää.

Laskennallisesti syvyyskarttaa pitää säätää pikseli shaderissä niin, että kohdistetun alueen etäisyys muuttuu. Syvyyskartan muutosta syväterävyyskartaksi on havainnollistettu kuvassa 62.



KUVA 62. Syvyyskartasta syväterävyyskartaksi (Viljanen 2012)

Varsinaisen sumennuksen periaate on se, että pohjaksi otetun kuvan pikselit korvataan kyseisten pikseleiden ympäriltä laskettujen pikseleiden tai tekseleiden keskiarvolla. Sumennuksen määrään vaikuttaa ympäriltä laskettujen pikseleiden hajautumisen suuruus. Hajautumista on havainnollistettu kuvassa 64 näkyvillä ruudukoilla, joissa yksi ruutu on yksi pikseli.



KUVA 63. Tekselistä poispäin hajautuvien alitekselien etäisyys määräytyy syväterävyysaluekartan avulla (Viljanen 2012)

Varsinaiseen syväterävyyteen voidaankin vaikuttaa eräänlaisella syväterävyyskartan ja sumennuksen yhdistelmällä. Kuvan 63 alareunan liukuväreissä musta alue tarkoittaa syvyysaluetta, jossa terävöityminen tapahtuu. Punainen viiva kertoo millä alueella kyseisessä pikselissä ollaan. Käyttämällä sumennuksen koon yhteydessä kuvan kaltaista lähestymistapaa, saadaan terävyys rajattua tietylle alueelle. Miten sitten haluaa esimerkiksi virtuaalitodellisuudessa polttoväliin vaikuttaa, jää kehittäjän omaksi ideoinniksi ja toteutukseksi. Casessa polttoväli muuttuu hiiren kursoria siirtämällä kohtaan, johon kuvan haluaa tarkentaa. Kuvan 64 ruutukaappaus kertoo miltä syväterävyyden muutos näyttää kaivinkonesimulaattorissa.

KUVA 64. Syväterävyys kaivinkonesimulaattorissa (Viljanen 2012)

8 YHTEENVETO

Syvällisempi perehtyminen shadereiden teoriaan ja sovelluskohteisiin auttoi ymmärtämään paljon uusia asioita tietokonegrafiikan luonnista ja mahdollisuuksista. Teoreettisten lähteiden selvittelystä ja case-osuuden teosta syntyi useita uusia ajatuksia, miten shadereitä voisi hyödyntää vieläkin tehokkaammin. Shadereiden ohjelmointi auttoi hahmottamaan näytönohjainten todellista laskentakykyä ja rinnakkaislaskennallista hyötyä. Pohjatietämyksestä oli apua aiheeseen syventymiseen.

Shader-käsitteiden monimutkaisuus selveni opinnäytetyötä tehdessä, mutta edelleen tietyt käsitteet vaikuttavat hankalasti selvennetyiltä osassa lähteitä. Pieniä ongelmia aiheutti mm. se, että useammat termit tarkoittavat lähes samoja asioita. Esimerkiksi ShaderModel ja HLSL tarkoittavat jossain mielessä samaa asiaa. ShaderModel-käsite on todennäköisesti haluttu suunnata enemmän ulospäin uusista näytönohjaimista kiinnostuneille ja niitä tarvitseville käyttäjille, kun taas shader-ohjelmoinnista kiinnostuvat voivat hakeutua HLSL-hakusanan alle. Direct3D-shaderien lisäksi puhutaan usein DirectX-shadereistä, jotka tarkoittavat samaa asiaa. Tavallaan voitaisiin puhua jopa Windows Vista ja Windows 7 shadereistäkin, koska huomasin ShaderModel-versioiden osuvan Windows-julkaisujen yhteyteen.

Mielestäni viimeisen 10 vuoden aikana tapahtunutta kehitystä oli vähän hankala hahmottaa pelkistä pipeline-kaavioista. Päädyin siihen, että pelien ulkoasullinen kehitys on johtunut enemmänkin ”näkymättömistä” parannuksista, kuten suorituskyvyn kasvamisesta ja sen hyödyntämisestä. Silti Direct3D 9:n pystyi toteamaan varteenotettavaksi vaihtoehdoksi vähintään perinteisemmän 3d-grafiikan reaaliaikaisessa renderöinnissä. Jatkossa nähdään varmasti entistäkin suorituskykyisempiä ja näyttävämpiä shader-tekniikoita. GPGPU-tekniikkaan panostaminen vaikuttaisi olevan tämän hetken suosituimpia alueita, joita kehitetään ja joista haetaan vauhtia laaja-alaisempaankin laskentaan. Todennäköisesti reaaliaikainen säteenjäljitys ja muut raskaat fysikaaliset ilmiöt tulevat kehittymään suorituskyvyn myötä. Näitä tekniikoita kuitenkin kehitetään koko ajan still-kuvien ja animaatioiden renderöintiin.

Quest3d-ohjelman ja sen tarjoaman kehitysympäristön puitteissa vaihtoehdoksi jäi toteuttaa ainoastaan Direct3D 9:ää tukevia shader-ohjelmia. Uudemman Direct3D-tekniikan käyttäminen olisi varmasti ollut parempi lähtökohta opinnäytetyön tekemiseen. Toisaalta käytössä olevalla Direct3D 9 -tekniikallakin pystyi shadereiden toimintaperiaatteen havainnollistamaan ja luomaan esimerkiksi toimivan prototyypin kaivettavasta maastosta. Toisessa kehitysympäristössä shaderkokeilujen laaja-alaisuus olisi jäänyt opinnäytetyössä huomattavasti suppeammaksi. Ohjainten rakentaminen oli kokonaan uusi asia ja työn sivussa syntynyt lisäprojekti. Rakentaminen toimi vastapainona shader-ohjelmoinnille ja vei hetkeksi ajatukset pois teoriasta ja ohjelmoinnista. Ohjainten osuus osana kokonaisuutta antoi uusia näkökulmia, täydensi opinnäytetyön sisältöä ja toimi ratkaisuna kaivinkonesimulaattorin ohjauksessa.

LÄHTEET

Painetut lähteet

Engel, W.F. 2004a. ShaderX 2: Introduction & Tutorials with DirectX 9 [tallennettu 12.2.2012]. Wordware Publishing, Inc. Saatavissa: http://tog.acm.org/resources/shaderx/Introductions_and_Tutorials_with_DirectX_9.pdf

Engel, W.F. 2004b. ShaderX 2: Shader programming Tips & Tricks with DirectX 9 [tallennettu 12.2.2012]. Wordware Publishing, Inc. Saatavissa: http://tog.acm.org/resources/shaderx/Tips_and_Tricks_with_DirectX_9.pdf

Engel, W.F. 2009. ShaderX 7: Advanced Rendering Techniques. Course Technology.

Sähköiset lähteet

Act-3D. 2012a. Quest3D HLSL Object [viitattu 17.3.2012]. Saatavissa: http://support.quest3d.com/index.php?title=HLSLObject

Act-3D. 2012b. Quest3D HLSL Semantics [viitattu 17.3.2012]. Saatavissa: http://support.quest3d.com/index.php?title=Quest3D_Semantics

Conitec Datasystems. 2011. Shaders [viitattu 6.3.2012]. Saatavissa: http://www.coniserver.net/wiki/index.php/Shader

Computer Hope. 2012. Microsoft DirectX [viitattu 14.3.2012]. Saatavissa: http://www.computerhope.com/directx.htm

Greenheck, D. 2011. Shaders Introduction [viitattu 18.3.2012]. Digitseven. Saatavissa: http://digitseven.com/shadersintro.aspx

Koci, R. 2010. World, View and Projection Matrix Unveiled [viitattu 6.3.2012]. Saatavissa: http://robertokoci.com/world-view-projection-matrix-unveiled/

Lumonix. 2012. ShaderFX [viitattu 13.3.2012]. Saatavissa: http://www.lumonix.net/shaderfx.html

Microsoft. 2012a. MSDN Device Types (Direct3D 9) [viitattu 27.3.2012]. Saatavissa: http://msdn.microsoft.com/en-us/library/windows/desktop/bb219625%28v=vs.85%29.aspx

Microsoft. 2012b. MSDN Flow control [viitattu 26.3.2012]. Saatavissa: http://msdn.microsoft.com/en-us/library/windows/desktop/bb509600%28v=vs.85%29.aspx

Microsoft. 2012c. MSDN Graphics Pipeline [viitattu 27.3.2012]. Saatavissa: http://msdn.microsoft.com/en-us/library/windows/desktop/ff476882%28v=vs.85%29.aspx

Microsoft. 2012d. MSDN Input-Assembler Stage [viitattu 10.3.2012]. Saatavissa: http://msdn.microsoft.com/en-us/library/windows/desktop/bb205116%28v=vs.85%29.aspx

Microsoft. 2012e. MSDN Output-Merger Stage [viitattu 10.3.2012]. Saatavissa: http://msdn.microsoft.com/en-us/library/windows/desktop/bb205120%28v=vs.85%29.aspx

Microsoft. 2012f. MSDN Pipeline Stages Direct3D 10 [viitattu 18.3.2012]. Saatavissa: http://msdn.microsoft.com/en-us/library/bb205123%28VS.85%29.aspx

Microsoft. 2012g. MSDN Shader Models vs Shader Profiles [viitattu 13.3.2012]. Saatavissa: http://msdn.microsoft.com/en-us/library/windows/desktop/bb509626%28v=vs.85%29.aspx

Microsoft. 2012h. MSDN Shader Stages [viitattu 14.3.2012]. Saatavissa: http://msdn.microsoft.com/en-us/library/windows/desktop/bb205146%28v=vs.85%29.aspx

Microsoft. 2012i. MSDN Texture Filtering (Direct3D 9) [viitattu 12.3.2012]. Saatavissa: http://msdn.microsoft.com/en-us/library/windows/desktop/bb206250%28v=vs.85%29.aspx

Microsoft. 2012j. MSDN Writing HLSL Shaders in Direct3D 9 [viitattu 6.3.2012]. Saatavissa: http://msdn.microsoft.com/en-us/library/windows/desktop/bb944006%28v=vs.85%29.aspx

Myllymaa, M. 2003. Tietokonegrafiikka / perusteet [viitattu 15.3.2012]. TKK. Saatavissa: http://www.tml.tkk.fi/Opinnot/T-111.300/2003kesa/kalvot/tg9.piilopinnat/sld001.htm

NVIDIA. 2004. Shader Model 3.0: Using Vertex Textures [tallennettu 12.2.2012]. NVIDIA Corporation. Saatavissa: http://developer.nvidia.com/system/files/akamai/gamedev/docs/Vertex_Textures.pdf?download=1

Samyn, K. 2011. HLSL Shaders [viitattu 25.3.2012]. Saatavissa: http://knol.google.com/k/hlsl-shaders

Wikipedia. 2012a. Graphics pipeline [viitattu 12.2.2012]. Saatavissa: http://en.wikipedia.org/wiki/Graphics_pipeline

Wikipedia. 2012b. Microsoft Direct3D [viitattu 20.3.2012]. Saatavissa: http://en.wikipedia.org/wiki/Microsoft_Direct3D

Wikipedia. 2012c. Normal mapping [viitattu 17.3.2012]. Saatavissa: http://en.wikipedia.org/wiki/Normal_mapping

Wikipedia. 2012d. Polygon modeling [viitattu 10.3.2012]. Saatavissa: http://en.wikipedia.org/wiki/Polygonal_modeling

Wikipedia. 2012e. Shader [viitattu 25.3.2012]. Saatavissa: http://en.wikipedia.org/wiki/Shader

Wikipedia. 2012f. Transform, clipping, and lighting [viitattu 25.3.2012]. Saatavissa: http://en.wikipedia.org/wiki/Transform,_clipping,_and_lighting

Kuvalähteet

KUVA 1. Viljanen, V. 2012. Lahden ammattikorkeakoulu.

KUVA 2. ExtremeTech. 2011. DirectX 11 hasn't yet caught fire – so what? [viitattu 1.4.2012]. Saatavissa http://www.extremetech.com/wp-content/uploads/2011/09/directx_logo-300x209.png

KUVA 3. Viljanen, V. 2012. Lahden ammattikorkeakoulu.

KUVA 4. Lumonix. 2012. ShaderFX [viitattu 15.3.2012]. Saatavissa: http://www.lumonix.net/Media/ShaderFX_30screen01.jpg

KUVA 5. Microsoft. 2012a. MSDN ShaderModels vs ShaderProfiles [viitattu 20.3.2012]. Saatavissa: http://msdn.microsoft.com/en-us/library/windows/desktop/bb509626%28v=vs.85%29.aspx

KUVA 5. Viljanen, V. 2012. Lahden ammattikorkeakoulu.

KUVAT 6, 7. Wikipedia. 2012. Graphics pipeline [viitattu 5.3.2012]. Saatavissa: http://upload.wikimedia.org/wikipedia/commons/b/bb/The_OpenGL_-_DirectX_graphics_pipeline.png

KUVA 8. Penfold, D. 2002. Tom's Hardware [viitattu 2.3.2012]. Saatavissa: http://www.tomshardware.com/reviews/vertex-shaders-pixel-shaders,411-8.html

KUVAT 6 - 10. Viljanen Vesa. 2012. Lahden ammattikorkeakoulu.

KUVA 11. Microsoft. 2012b. MSDN Output-Merger Stage [viitattu 20.3.2012]. Saatavissa: http://i.msdn.microsoft.com/dynimg/IC412554.png

KUVAT 12, 13. Viljanen, V. 2012. Lahden ammattikorkeakoulu.

KUVAT 14 - 18. Koci, R. 2010. World, View and Projection Matrix Unveiled [viitattu 15.3.2012]. Saatavissa: http://robertokoci.com/world-view-projection-matrix-unveiled/

KUVA 19. Microsoft. 2012c. MSDN Intrinsic Functions [viitattu 20.3.2012]. Saatavissa: http://msdn.microsoft.com/en-us/library/windows/desktop/ff471376%28v=vs.85%29.aspx

KUVAT 19 - 64. Viljanen, V. 2012. Lahden ammattikorkeakoulu.

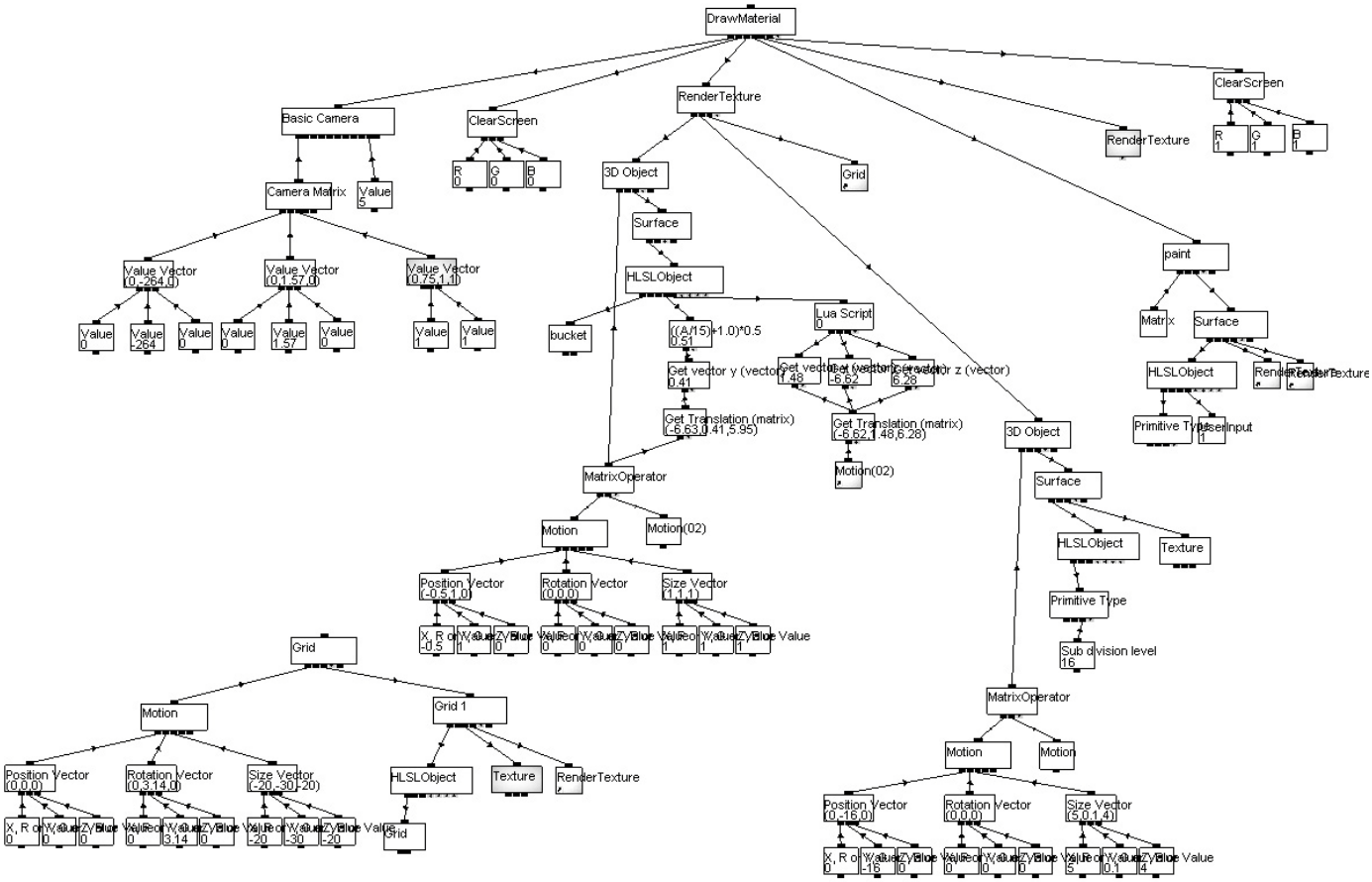
LIITTEET

Kaivinkonesimulaattori (exe-tiedosto)

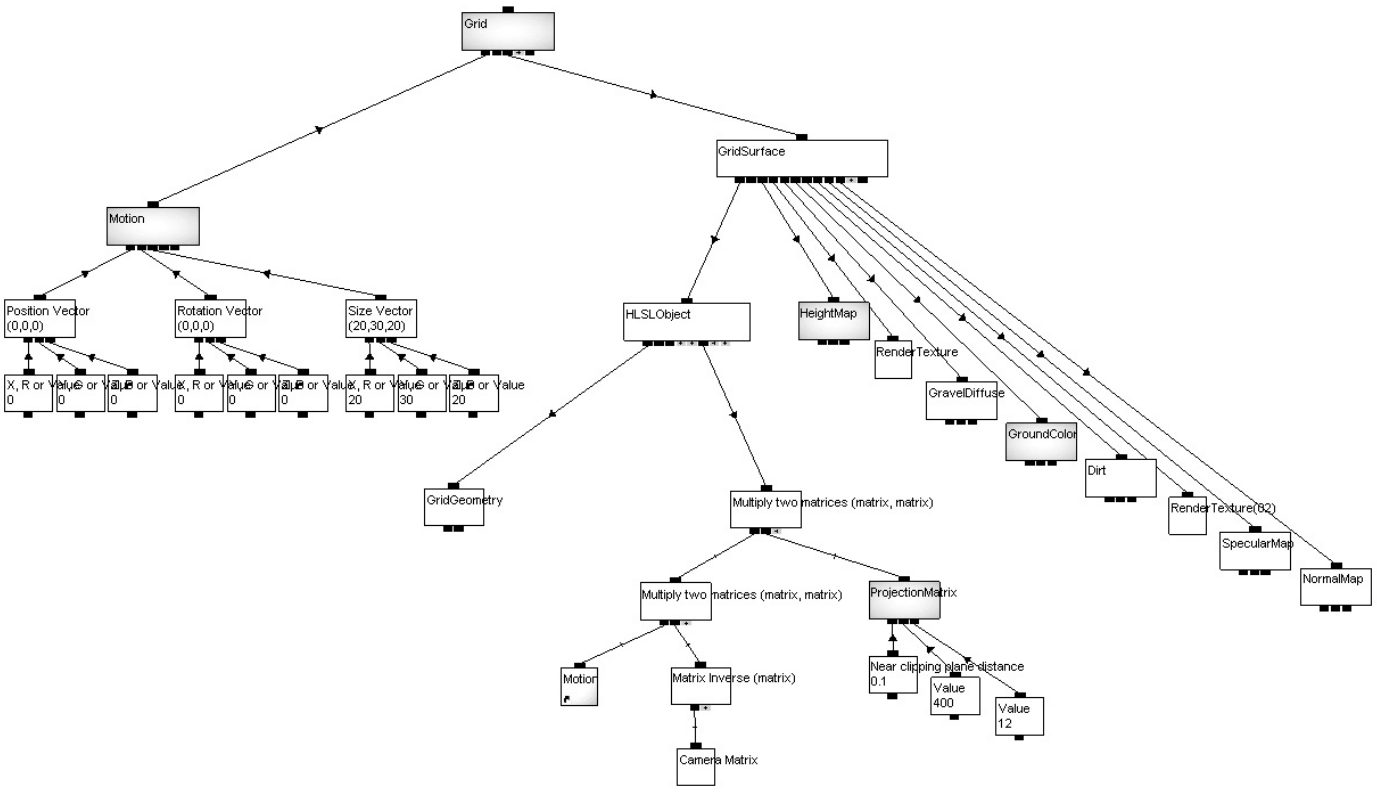
Quest3d-hiearkiarakenteiden kuvakaappauksia 2 sivua

Simulaattorin kuvakaappauksia 1 sivu

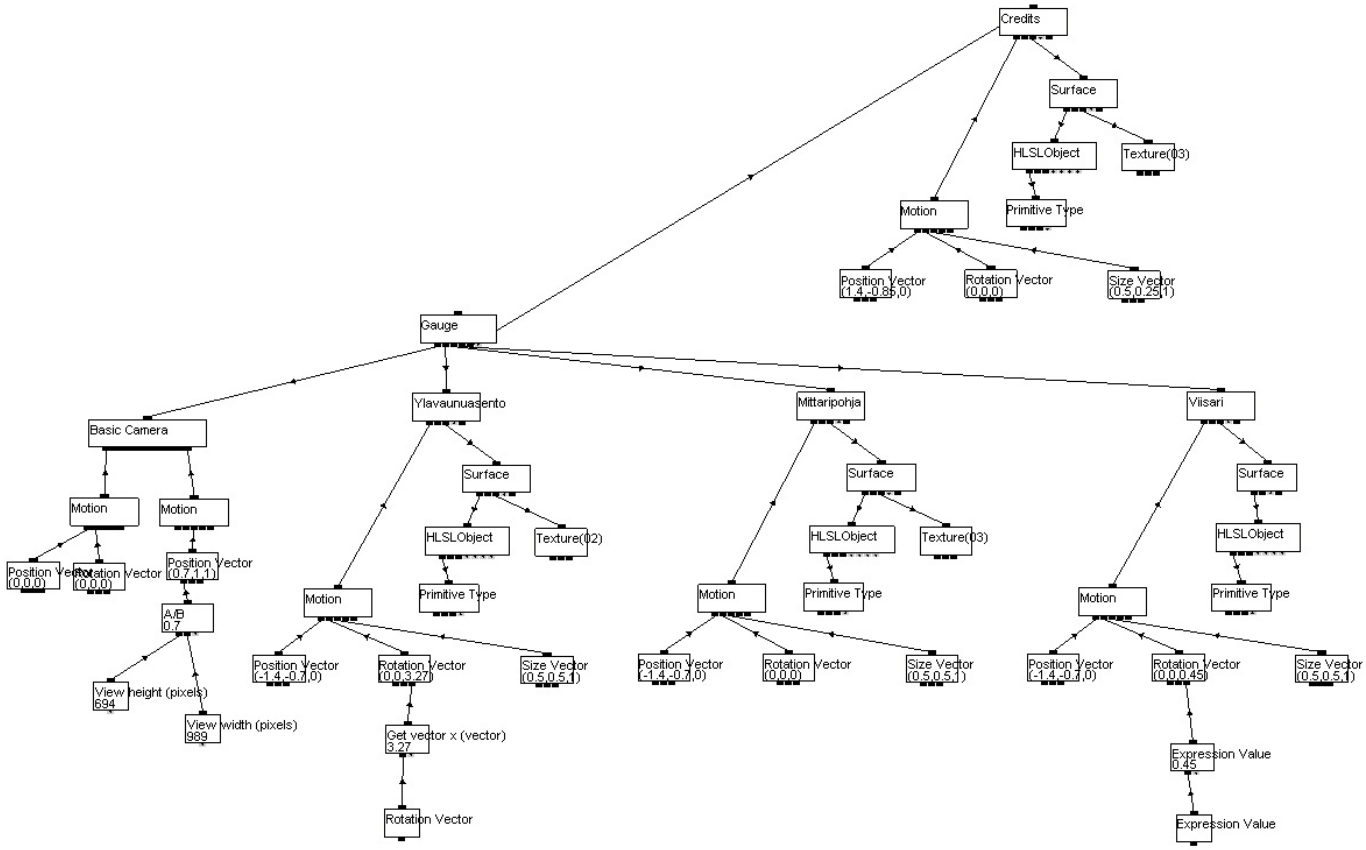
Maaston kaivaminen



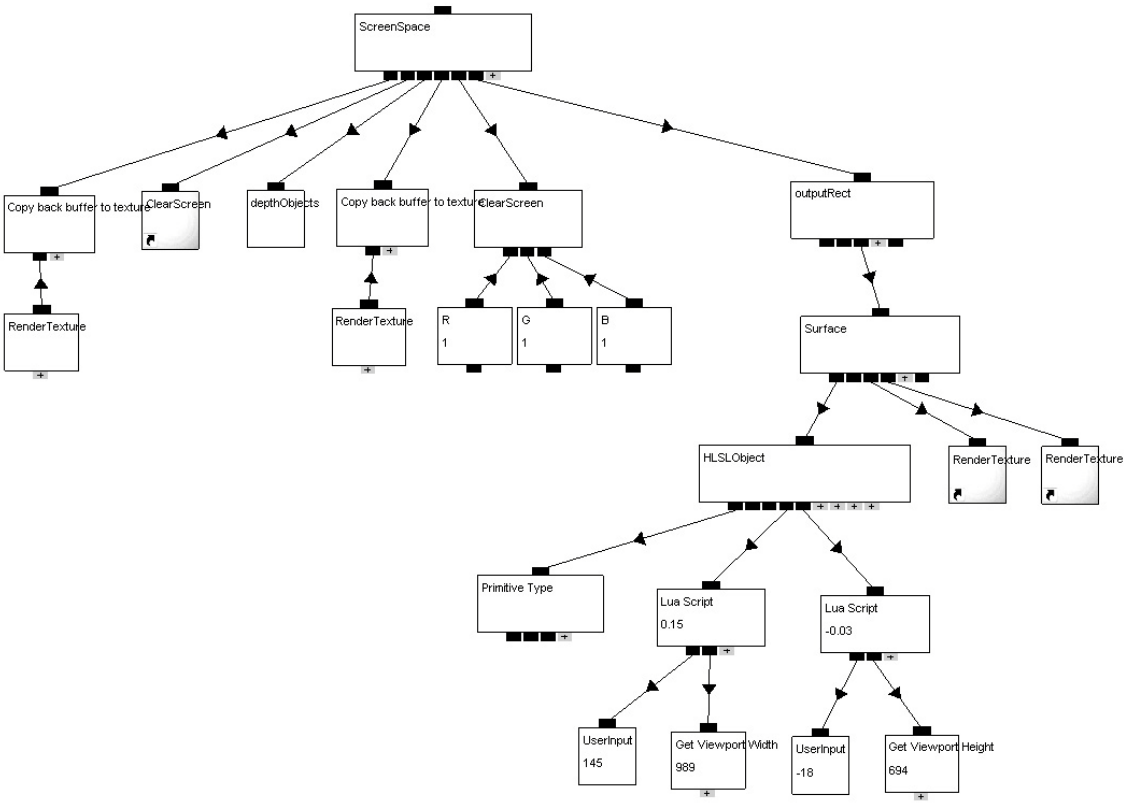
Maasto



Mittaristo



Kuva-shaderi




Täällä kaivan minä.
Kaivinkone.
Vesa Viljanen, viljanen.net 2012
Mallit, tekstuurit, shaderit, logiikka.
Moottorina Quest3D.


Täällä kaivan minä.
Kaivinkone.
Vesa Viljanen, viljanen.net 2012
Mallit, tekstuurit, shaderit, logiikka.
Moottorina Quest3D.