Ohjelmoitavat sävyttimet

Janne Timonen

Seminaaritutkielma
HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

Helsinki, 17. joulukuuta 2015

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Matemaattis-luonnontieteellinen	Tietojenkäsittelytieteen laitos

Tekijä — Författare — Author
Janne Timonen

Työn nimi — Arbetets titel — Title
Ohjelmoitavat sävyttimet

Oppiaine — Läroämne — Subject
Tietojenkäsittelytiede

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Seminaaritutkielma	17. joulukuuta 2015	18

Tiivistelmä — Referat — Abstract

Seminaarityö käsittelee ohjelmoitavia sävyttimiä, jotka ovat erityisesti tietokonepelien 3D-grafiikan tuotossa tehokkaita välineitä määritellä grafiikan piirtoon liittyviä ominaisuuksia ja luoda erilaisia graafisia tehokeinoja. Nykyään tietokonepeleihin sävyttimiä ohjelmoidaan pääasiassa kahden suosituimman grafiikkaohjelmointirajapinnan (OpenGL ja Direct3D) sävytinkielillä, jotka ovat GLSL ja HLSL. Tekstissä tarkastellaan Shader Model 5.0 mukaisen grafiikkaliukuhihnan ohjelmoitavia sävyttimiä sekä liukuhihnan ulkopuolista laskentasävytintä. Sävyttimistä käydään läpi niiden perustoiminnallisuudet sekä toimintalogiikka. Erityisesti kiinnitetään huomiota erilaisiin graafisiin tekniikoihin, joita tietyillä sävyttimillä on mahdollista toteuttaa.

Avainsanat — Nyckelord — Keywords
3D-grafiikka, ohjelmoitavat sävyttimet, reaaliaikainen grafiikka

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — Övriga uppgifter — Additional information

Sisältö

1 Johdanto

Sävyttimet ovat ohjelmia, joiden tehtävänä *grafiikkaliukuhihnalla* (asteittainen kuvantuottoprosessi) on *sävyttää* eli tuottaa tietyillä tavoilla dataa kuvaksi. Tämä voi tarkoittaa esimerkiksi jonkin objektin piirtämistä sijainnin mukaan, pikselikohtaista värinmääritystä, pinnanmuotojen simulointia tai muita erikoistehostemaisiakin keinoja.

Sävytin ottaa syötteenään elementin, kuten esimerkiksi monikulmion kärkipisteen, kokonaisen monikulmion tai yksittäisen pikselin. Saadusta syötteestä sävytin tuottaa tuloksena muunnettuja elementtejä, joiden määrä voi vaihdella nollasta useaan riippuen sävyttimestä, ja sen suorittamasta tehtävästä [Gre14, s. 500]. Sävyttimien käyttö mahdollistaa myös hyvin rinnakkaisuuden käytön, kun muunnoksia tehdään suurille datamäärille kerrallaan, esimerkiksi kaikille ruudun pikseleille. Moderneilla grafiikkapiireillä onkin useita sävytinliukuhihnoja rinnakkaisuusmahdollisuuksien hyödyntämiseksi.

Tutkielmassa tarkastellaan aluksi hieman yleisesti 3D-grafiikkaa, jotta sävyttimien roolia grafiikan tuottamisessa voi ymmärtää paremmin ja hahmottaa niiden tehtävää, minkä jälkeen käydään läpi sävyttimien historian päävaiheet ja kuinka nykyisiin ohjelmoitaviin sävyttimiin on päädytty. Tämän jälkeen siirrytään asian varsinaiseen ytimeen eli ohjelmoitaviin sävyttimiin, ja käydään vaiheittain läpi tällä hetkellä käytettävien sävyttimien toimintaa, mitä niillä on mahdollista tehdä ja kuinka se tapahtuu. Lopuksi tarkastellaan hieman tarkemmin kuinka sävytin voidaan kääntää ja sitoa mukaan grafiikan tuottoon.

2 3D-grafiikka

3D-grafiikassa tuotetaan kolmiulotteisista malleista kaksiulotteinen esitys eli esimerkiksi katsojan näkemä kuva tietokoneen ruudulla. Erityisesti tämän tutkielman tapauksessa 3D-grafiikan reaaliaikaisen piirron menetelmänä käsitellään *rasterointia* eli kuvan muuttamista pikseleillä esitettävään muotoon niin kutsutuksi rasterikuvaksi eli kaksiulotteiseksi pikseliruudukoksi. Rasteroinnissa kolmiulotteiset mallit on projisoitu ja leikattu piirtoikkunaan halutulla tavalla, minkä jälkeen kuvaa piirretään kaksiulotteiseen niin kutsuttuun rasteriruudukkuun, joka käytännössä koostuu pikseleistä. Tarkkaa reaaliaikaisuutta vaativien grafiikkasovellusten, kuten pelien, tapauksessa nopea kuvan piirtäminen nousee tärkeäksi vaatimukseksi, jolloin polygoneista eli monikulmioista tuotetaan rasteroimalla kaksiulotteista kuvaa. Tavoitteena on saavuttaa ruudunpäivitysnopeus, joka vaikuttaa ihmisen silmään sulavalta, eli ainakin noin 30 ruutua sekunnissa [Gre14, s. 444-445].

Ei-reaaliaikaisiin 3D-grafiikan piirtotapoihin lukeutuu esimerkiksi *säteenjäljitys* (Ray tracing), jossa valaistus ja valon heijastukset lasketaan simuloiden tarkemmin todellista maailmaa. Esimerkiksi voidaan laskea pinnalta heijastuneen valon heijastuminen uudestaan joltain toiselta pinnalta [Puh08, s.405-406]. Tällainen piirto on hidasta, eikä vielä tällä hetkellä ole järkevästi hyödynnettävissä reaaliaikaisessa käytössä, kuten peleissä. Ennakkoon prosessoituna piirto silti tuottaa lähes fotorealistista kuvaa. Säteenjäljitykseen tai vastaaviin tekniikoihin ei tässä tutkielmassa enää palata.

Sävyttimien ymmärtämisen kannalta on hyödyllistä tuntea myös erilaiset 3D-grafiikkaan liittyvät *koordinaattijärjestelmät* (tai avaruudet), joissa 3D-malleja käsitellään. Näihin lukeutuvat muun muassa *malli-*, *maailma-*, *näkymä-* ja *projektiokoordinaatisto*. Mallikoordinaatisto pitää sisällään vain jonkin tietyn 3D-mallin, esimerkiksi yksittäisen puun tai hahmon. Maailmakoordinaatisto käsittää avaruuden, johon objektit siirretään mallikoordinaatistosta ja jossa ne sijaitsevat absoluuttisesti tietyllä paikalla. Näkymäkoordinaatistossa asioita tarkastellaan kameran näkökulmasta, ja projektiokoordinaatistossa näkymään on lisätty myös perspektiivi. Sävyttimet siirtyvät koordinaatistosta toiseen tekemällä *muunnoksia* eli tekemällä matriisilaskutoimituksia kulloiseenkin sijaintikoordinaatistoon [Puh08, s. 167-171]. Muunnoksista koordinatistojen välillä käsitellään tarkemmin kärkipistesävyttimestä kertovassa luvussa.

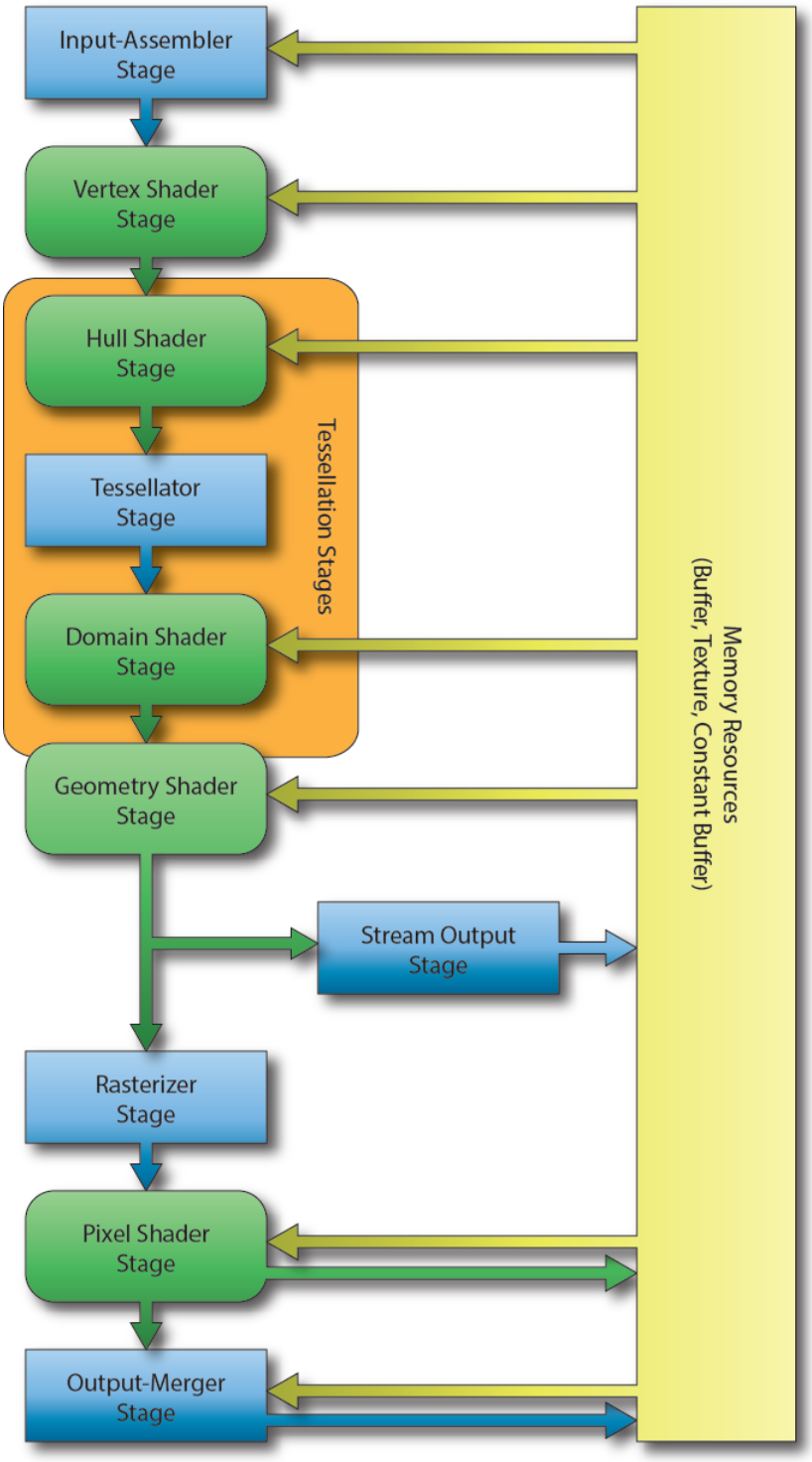
2.1 Grafiikkaliukuhihna

Rasterointiin tähtäävässä 3D-grafiikassa hyödynnetään niin kutsuttua liukuhihnaa (pipeline), joka on pääasiassa sarja tietyssä järjestyksessä tehtäviä askeleita tai työvaiheita, joilla 3D-malleista luodaan 2D-rasterikuva. Liukuhihna ja erityisesti *grafiikkaliukuhihna* on siis prosessi, jolla lopullinen kuva tuotetaan. Peligrafiikan tuottamiseen rasteroinnin keinoilla suosituimmat

grafiikkaohjelmointirajapinnat ovat OpenGL ja Direct3D, jotka kummatkin toimivat samanlaisen grafiikkaliukuhihna-ajattelun mukaisesti [Khr15] [Mic11].

Valmistajien grafiikkapiirien tukemia sävyttimiä eritellään *Shader Model* -versioinnin avulla, joka kertoo millaista grafiikkaliukuhihnaa ja mitä sävyttimiä jokin laite tukee. Tutkielman tarkastelema grafiikkaliukuhihna on Shader Model 5.0:n mukainen, jolloin voidaan esitellä kaikki nykyään peleissä mahdollisesti käytössä olevat sävyttimet. Grafiikkaliukuhihna koostuu kärkipiste-, tesselaatio-, geometria- ja pikselisävyttimestä [Mic11].

Grafiikkaliukuhihnaan kuuluu myös joitain kiinteitä vaiheita, jotka eivät varsinaisesti ole sävyttimiä, tai ainakaan ohjelmoitavia sellaisia. Esimerkiksi ennen varsinaista grafiikkapiirissä suoritettavia vaiheita on prosessorilla suoritettava sovellusvaihe, jossa muun muassa 3D-mallit valmistellaan sellaiseen muotoon, että grafiikkapiirtojärjestelmä voi niitä käyttää [Puh08, s. 164-165]. Geometriasävyttimen jälkeinen vaihtoehtoinen tulostevirta (Stream output stage) taas voi ohjata primitiividatan takaisin liukuhihnan aiempiin vaiheisiin. Rasterointivaiheessa puolestaan hoidetaan rasterointi. Näihin kiinteisiin vaiheisiin ei myöhemmin enää juurikaan paneuduta lukuun ottamatta tilanteita, joissa ne ovat sävyttimen käytön kannalta oleellisia. Liukuhihnan vaiheet ja järjestys ovat esitetty kuvassa 1 [Mic11].



Kuva 1: Grafiikkaliukuhihna

3 Sävyttimien historiaa

1990-luvulta alkaen grafiikanpiirto on käynyt läpi erilaisia vaiheita. Laitteistotasolla on edetty harppauksin, mutta myös keinot tuottaa kuvaa ovat muuttuneet. Prosessorilla tehtävästä laskentatyöstä on edetty erillisten 3D-grafiikkakiihdyttimien kautta nykyisenkaltaisiin grafiikkapiireihin, joiden laskentateho ja tuki erilaisille teknologioille on moninkertainen verrattuna varhaisemman sukupolven laitteisiin. Sävyttimien kohdalla ohjelmointitavat ovat kulkeneet muokkaamattoman kiinteän liukuhihnan ja vaivalloisen konekielityylisen ohjelmoinnin kautta nykyisiin korkean tason sävytinkielisiin ohjelmiin. Tässä luvussa tarkastellaan hieman sävyttimien historian pääkohtia.

3.1 Kiinteä liukuhihna

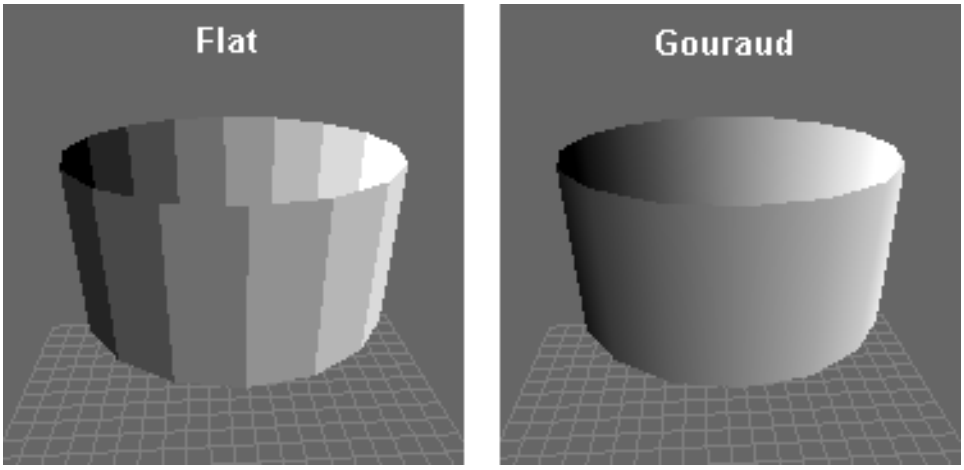
Ennen ensimmäisiä *grafiikkakiihdyttimiä*, joihin kuului muun muassa 3Dfx:n Voodoo, grafiikan piirto tapahtui prosessorilla, jonka työtaakkaa erilliset grafiikkakiihdyttimet kehitettiin vähentämään. Vapaasti ohjelmoitavia sävyttimiä ei vielä tuolloin kuitenkaan ollut, ja grafiikkaa tuotettiin käyttämällä hyväksi grafiikkapiirien *kiinteää liukuhihnaa* (Fixed-function pipeline) [She08, s. 11]. Kehittäjä saattoi siis antaa raskaat laskutyöt grafiikkakiihdyttimelle hoidettavaksi käyttäen kiinteää liukuhihnaa, mutta itse liukuhihnan suorittamiin funktioihin ei voinut puuttua muuten kuin parametrien avulla.

Kiinteä liukuhihna näytönohjaimessa nopeutti laskentaa ja toi mukanaan mahdollisuuksia luoda standardioperaatioiden rajoissa graafisia tehokeinoja ja efektejä, kuten esimerkiksi Goraud-sävytyksen, josta on esimerkki kuvassa 2 [Bur09]. Verrattuna monikulmion tasaiseen väritykseen Gouraud-sävytyksessä kärkipisteiden väriarvot interpoloidaan monikulmion sisällä, jolloin saadaan tasaisempi ja luonnollisemman näköinen heijastus kuin tasaisella sävytyksellä [Gou71].

Myöhemmin tulivat ensimmäiset *ohjelmoitavaa grafiikkaliukuhihnaa* tukevat näytönohjainpiirit, joissa kiinteän liukuhihnan vaiheita pystyi korvaamaan omilla vapaasti ohjelmoitavilla sävyttimillä. Korvaamalla kiinteän liukuhihnan laskenta nykyajan grafiikkapiirien tukemilla ohjelmoitavilla sävyttimillä voidaan vapaasti muokata sävyttimien käyttäytymistä ja saavutetaan mahdollisuus piirtää kuvaa enemminkin luovuuden rajoissa kuin ennaltamääriteltyjen ehtojen.

3.2 Varhaiset ohjelmoitavat sävyttimet

Ensimmäiset ohjelmoitavan liukuhihnan sävytinmallit tukivat ainoastaan alemman tason konekieliin pohjautuvilla kielillä ohjelmointia. Tällaiset kielet lainasivat periaatteensa RISC-prosessoriarkkitehtuurista, jossa konekielen käskykanta on yksinkertainen ja rajoitettu. Sen lisäksi, että kehittäjien



Kuva 2: Tasainen ja Gouraud-sävytys

täytyi luoda sävytin, täytyi se luoda lisäksi usein erikseen sekä OpenGL- että Direct3D-rajapinnoille johtuen näiden kahden suosituimman rajapinnan kielien poikkeavuuksista.

Aluksi laitteet tukivat ainoastaan kärkipiste- ja pikselisävytintä, laitekohtaisesti jopa vain ensimmäistä näistä. Silloiset rajoitukset käskyjen määrässä ja rekistereiden koossa tekivät silti vielä joistain asioista haasteellisia toteuttaa. Esimerkiksi pikselikohtainen valaistus, jonka nykyään voi ajatella olevan suoraviivaisesti toteutettavissa, oli tuolloin hankala toteuttaa laitteiston rajoituksista johtuen ja vaati kehittäjiltä erilaisia keinoja toimiakseen reaaliajassa [She08, s. 174-176].

Eräs edelläkävijöitä sävyttimien saralla oli tietokoneanimaatioelokuvistaan tunnettu Pixar-yhtiö jo 1980-luvun lopussa kehittämällään *RenderMan*-kielellä, jota on käytetty satojen elokuvien visuaalisiin efekteihin ja animaatioelokuvien piirtämiseen [Pix15].

3.3 Korkean tason sävytinkielet

Ohjelmoitavien sävyttimien alkuaikojen jälkeen alettiin sävyttimien luomisessa siirtyä alemman tason kielistä korkean tason sävytinkieliin. Alemman tason kieliin verrattuna korkean tason kielillä on useita hyötyjä, ja ne pätevät myös korkean tason sävytinohjelmoinnissa: helpompi luettavuus, kirjoitettavuus, muokattavuus, virheiden etsintä ja löytäminen sekä nopeampi kehitysvauhti [She08, s.183-185].

Ohjelmoitavien sävyttimien tultua kasvoi myös tarve korkean tason sävytinkielille, joista mainittavimmiksi kieliksi muodostuivat C-pohjaiset Nvidian Cg (C for Graphics), Microsoftin HLSL (High Level Shading Language) ja OpenGL:n GLSL [Nvi03] [Mic11] [Khr15]. Cg ja HLSL syntyivät Microsoftin ja Nvidian yhteistyöprojektin tuloksena, ja ne ovatkin kielinä lähes identtiset

[She08, s. 198]. Näistä kielistä Cg on jo vanhentunut, eikä sitä enää päivitetä [Nvi12]. HLSL ja GLSL ovat monin osin samankaltaisia, ja ne tukevat samanlaisia sisäänrakennettuja funktioita. Eroina on muun muassa joidenkin samojen datatyyppien poikkeavat nimeämiskäytännöt. Esimerkiksi neljän komponentin vektori on GLSL-kielellä ilmaistuna `vec4`, kun taas HLSL-kielellä se on `float4` [She08, s. 198]. GLSL on lisäksi alustariippumaton, missä taas HLSL tähtää pääasiallisesti Direct3D:n kautta Windows-ympäristön käyttöön [Khr15] [Mic11]. Myöhemmät koodiesimerkit ja käytetty sävytinterminologia ovat pääasiassa Direct3D:n ja HLSL:n käytäntöjen mukaista.

4 Ohjelmoitavat sävyttimet

Ohjelmoitavien sävyttimien osalta grafiikkaliukuhihna rakentuu pääpiirteissään kärkipiste-, tesselaatio, geometria- ja pikselisävyttimestä. Yksi sävytinvaihe ottaa syötteenään vastaan edellisen tulosteen, joten jokainen vaihe voi jatkaa seuraavan datan työstämistä, kun on saanut edellisen työn valmiiksi. Tämä grafiikkaliukuhihnan malli mahdollistaa sävytinprosessoinnin rinnakkaistamisen erittäin hyvin [Ake02, s. 10-12].

Tesselaatiosävytin ja geometriasävytin ovat uudempia tulokkaita, ja niiden käyttö ei ole pakollista, jolloin niiden toiminnallisuus voidaan jättää pois tai korvata muilla tavoin. Esimerkiksi ennen tesselaatiolle omistettua omaa sävytintä tesselaatiosävytys toteutettiin geometriasävyttimen avulla [Sch14]. Tässä luvussa tarkastellaan erilaisia ohjelmoitavia sävyttimiä siinä järjestyksessä, jossa ne toimivat grafiikkaliukuhihnalla.

4.1 Kärkipistesävytin

Kärkipistesävytin ajetaan kerran jokaiselle monikulmion kärkipisteelle. Kärkipistesävytin ottaa syötteenään kärkipisteen *attribuuttitiedon*, joka sisältää muun muassa kyseisen kärkipisteen sijainnin malli- tai maailmakoordinaatistossa sekä pinnan normaalivektorin. Tulosteena kärkipistesävytin antaa yhden kärkipisteen, joka on käynyt läpi valaistuksen ja koordinaatiston muunnosvaiheet ja joka ilmaistaan *normalisoidussa kuvausavaruudessa* [Gre14, s. 500]. Yleisesti kärkipistesävytin voi muokata monikulmion kärkipisteen, normaalin, tekstuurikoordinaattien ja paikan arvoja. Sävyttimellä voi luoda esimerkiksi tuulessa heiluvat puiden oksat tai vedenpinnan väreilyn.

Kärkipistesävyttimen tärkeimpiin tehtäviin kuuluu tehdä tarvittavat koordinaatistomuunnokset syötteinä saaduille kärkipisteille. Ensimmäisenä tehdään *mallimuunnos*, eli muunnos mallikoordinaatistosta maailmakoordinaatistoon, jolloin kärkipiste sidotaan omasta paikallisesta koordinaatistostaan maailmanäkymään. Tämän jälkeen tehdään *katsojamuunnos* maailmakoordinaatistosta katsojan koordinaatistoon, jolloin origo on "kameran" näkökulma. Viimeiseksi suoritetaan *projektio-* tai *perspektiivimuunnos*, jolloin tuotetaan renessanssiajan kuvataiteesta tuttu luonnollisen elämän persepektiiviä jäljittelevä kuva, jossa kauempana olevat kohteet näkyvät pienempinä *näköfrustumissa* [Puh08, s. 386-389]. Näköfrustumi on suorakulmainen kuvausikkuna, joka projektion keskipisteestä (katsojan silmästä) lähtien muodostaa ääretömyyteen jatkuvan pyramidin, jonka sisällä katsojalle näkyvät asiat ovat [Puh08, s. 187-188]. Vähintään kärkipistesävyttimen tulee siis antaa tuloksena kärkipiste muunnettuna normalisoituun kuva-avaruuteen *uniformina* eli vakiomuotoisena ja kaikille sävyttimille yhteisenä tietona [Puh08, s. 388].

Kärkipistesävytin on pakollinen vaihe grafiikkaliukuhihnalla, ja sen täytyy vähintään kuljettaa lävitseen syötteenä saatu kärkipiste, vaikkei laskusuorituksia tehtäisikään. Alla HLSL-kielellä kirjoitettu yksinkertainen

kärkipistesävytin syötteineen ja tuloksineen [Mic11]:

```
// Input / Output structures

struct VS_INPUT
{
    float4 vPosition    : POSITION;
    float3 vNormal      : NORMAL;
    float2 vTexcoord    : TEXCOORD0;
};

struct VS_OUTPUT
{
    float3 vNormal      : NORMAL;
    float2 vTexcoord    : TEXCOORD0;
    float4 vPosition    : SV_POSITION;
};

// Vertex Shader

VS_OUTPUT VSMain( VS_INPUT Input )
{
    VS_OUTPUT Output;

    Output.vPosition = mul( Input.vPosition,
        g_mWorldViewProjection );
    Output.vNormal = mul( Input.vNormal, (float3x3)g_mWorld );
    Output.vTexcoord = Input.vTexcoord;

    return Output;
}
```

Syöte- ja tulos-structeissa on määritelty vektoreina (`float`n) kärkipisteen sijainti, normaali ja tekstuurikoordinaatti. `VSMain`-metodi ottaa syötteenään kyseiset arvot ja tekee sijainnille sekä normaalille matriisimuunnokset. Sijainnille tehdään maailma-katsojamuunnos katsojan koordinaatistoon ja normaalille maailmamuunnos. Tekstuurikoordinaatti kulkee esimerkissä sävyttimen läpi muuttumatta. Vaikka esimerkki on yksinkertainen, noudattelee sävyttimien ohjelmointi samaa logiikkaa grafiikkaliukuhihnan kaikissa vaiheissa.

4.2 Tesselaatiosävytin

Tutkielman sävyttimistä uusin on Shader Model 5.0:n (rajapinnoissa DirectX 11 ja OpenGL 4.0) myötä tullut tesselaatiosävytin. *Tesselaatiossa monikulmioverkko* (polygon mesh) eli kärkipisteistä ja reunaviivoista kohtaava kolmioiden joukko [Puh08, s. 49-50], jaetaan pienempiin osasiin, kuten vaikkapa kolmio kahteen pienempään kolmioon [Nvi10]. Lyhyesti sanottuna tesselaatio on monikulmioiden rikkomista ja jakamista pienempiin ja hienompiin osasiin.

Tesselaatiosävytin on jaettu kolmeen vaiheeseen, joista ensimmäinen ja kolmas ovat ohjelmoitavia sävytinvaiheita. Vaiheet alusta alkaen ovat *Hull Shader*, kiinteä tesselaatiovaihe *Tessellation Stage* ja *Domain Shader* [Mic11]. OpenGL-terminologiassa vastaavat vaiheiden nimet ovat Control, Primitive Generator ja Evaluation. Tesselaatiossa ensimmäisen sävytinvaiheen tehtävä on määrittää paljonko syötettä tesseloidaan. Kiinteä tesselaatiovaihe hoitaa tämän jälkeen varsinaisen laskutyön saamansa syötteen perusteella, minkä jälkeen kolmannen vaiheen sävytin työstää tesseloidun datan ja määrittää kärkipisteiden sijainnit. Käytännössä tesselaation kolmas vaihe toimii kuten tavallinen kärkipistesävytin [Mic11].

Menetelmänä tessellointi ei välttämättä tuo suoraan ulkonäöllisiä mullistuksia esimerkiksi pelien ulkonäköön, sillä ulkoasun kannalta ei ole eroa piirretäänkö vaikkapa neliö kahden vai kymmenien kolmioiden avulla. Sen sijaan yhdistämällä tesselaatioon muita tekniikoita ja laittamalla monikulmioista pilkotut palaset esittämään uutta informaatiota saadaan piirtoa monimutkaisemmaksi ja todellisemman näköiseksi [Nvi10]. Esimerkiksi monimutkaisen geometrian luominen lennosta on ennen tesselaatiosävytintä ollut vaikeata, kun työ on tehty ensin prosessorilla ja ladattu sen jälkeen grafiikkapiirille. Shader Model 5 -version myötä grafiikkapiirien tukiessa tesselaatiota laitetasolla voidaan tesselointi tehdä tehokkaasti [Sch14].

Eräs tesseloinnin tekniikka on *nyrjäyttäminen* tai *nyrjäytyskartta* (Displacement mapping), jossa tesseloidun pinnan kärkipisteitä nostetaan tai lasketaan korkeusattribuutin perusteella, jolloin saadaan luotua epätasaisia pintoja [Nvi10]. Epätasainen geometria luodaan siis aidosti verrattuna vaikkapa tavalliseen pinnan kuhmutukseen, joka on pikseleiden väriarvoilla luotu illuusio epätasaisuudesta [Gre14]. Nyrjäytystekniikkaa havainnollistaa kuva 3. Tessellointi säästää myös muistia ja kaistanleveyttä mahdollistamalla yksityiskohtaiset pinnat pieniresoluutioisilla tekstuureilla [Mic11] [Nvi10].

Tesselaation avulla saavutettava keino on myös 3D-mallien silottaminen normaalipiste-kolmioiden (PN-triangle) avulla [Vla01]. Tekniikassa karkean mallin litteät kolmiot korvataan kaarevilla PN-kolmioilla, jotka tesselaation avulla jaetaan useammiksi pieniksi kolmioiksi. Kaarevuus on toteutettu niin kutsutun Bézier-pinnan ja siihen liittyvien kontrollipisteiden avulla. Kaareutuva pinta muodostuu, kun tesseloituja uusia monikulmioita venytetään kohti määriteltyjä kontrollipisteitä. Esimerkiksi kuvassa 4 näkyvän Stalker: Call of Pripyat -pelin hahmon kaasunaamarin suodattimen reunoja on saatu tesselaatiolla luonnollisemmalla tavalla kaartuviksi (DX11) verrattuna yksinkertaisempaan ja vähäisemmillä monikulmioilla piirrettyyn malliin (DX10). Tarkempi matemaattinen selitys PN-kolmioille löytyy lähdeartikkelista [Vla01].

Dynaamisella tesselaatiolla voidaan skaalata mallien piirron tarkkuutta näkyvyyden suhteen muuttamalla yksityiskohtien määrää (LOD - Level of detail) lennosta [Gre14, s. 448]. Tällöin esimerkiksi avarassa ulkoilmapelinäkymässä kaukaa katsottuna jostain mallista piirretään malli muutamalla

Kuva 3: Nyrjäytyskartan realistisemmat pintaerot

Kuva 4: Monikulmiomallin silottaminen tesselaation avulla pelissä Stalker: Call of Pripyat

monikulmiolla. Lähestyttäessä monikulmioiden määrää lisätään dynaamisesti, kunnes läheltä katsottuna malli voidaan piirtää tarkasti tuhansilla monikulmioilla. Näin voidaan myös estää objektien yhtäkkinen tyhjästä ilmestyminen, jossa yksityiskohtainen malli täytyy piirtää kerralla kokonaisuudessaan [Nvi10].

4.3 Geometriasävytin

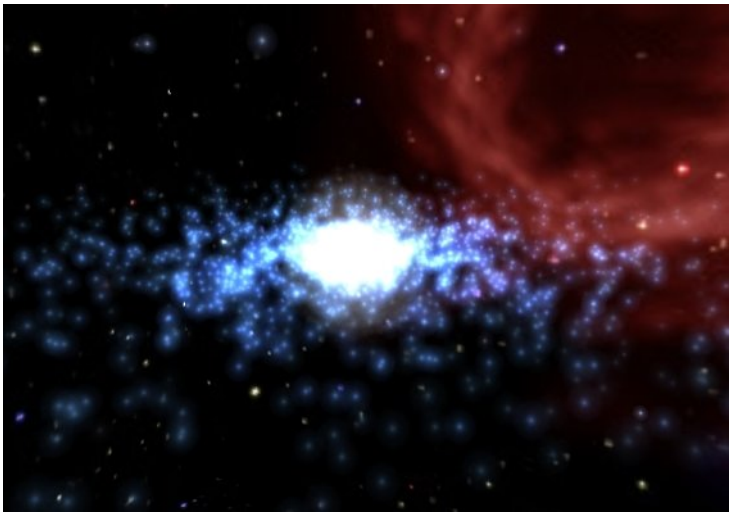
Geometriasävytin on kärkipiste- ja pikselisävyttimiin verrattuna uudempi tekniikka sen tultua esitellyksi Shader Model 4.0:n ja DirectX 10:n myötä. Geometriasävytin sijaitsee grafiikkaliukuhihnalla tesselaatiosävyttimen jälkeen ja ennen pikselisävytintä. Geometriasävytin on vaihtoehtoinen osa liukuhihnaa [Mic11].

Geometriasävytin ottaa syötteenään n-kärkipisteestä muodostuvia primitiivejä, kuten pisteitä (n=1), suoria (n=2) tai kolmioita (n=3). Syötteistä geometriasävytin muokkaa, valikoi ja jopa luo uusia primitiivejä [Gre14, s. 500]. Sävytin voi siis antaa tuloksena syötteestä riippumattoman määrän geometrisiä primitiivejä, jotka eivät välttämättä ole samaa tyyppiä kuin syötteenä saadut. Geometriasävytin voi esimerkiksi yhdistellä yksittäisiä kärkipisteitä, koota kolmioita uudeksi monikulmioksi tai hylätä syötteenä saadun primitiivin kokonaan. Toisin kuin kärkipistesävytin, joka kykenee käsittelemään vain yhden monikulmion kärkipisteen kerrallaan, on geometriasävyttimellä täydelliset tiedot käsittelemästään primitiivistä [Khr15]. Täten se pystyy käsittelemään useista kärkipisteistä koostuvaa primitiiviä [Mic11].

Geometriasävyttimellä tyypillisesti luotuihin asioihin kuuluvat esimerkiksi *partikkelijärjestelmät* ja *mainostaulut*. Partikkelijärjestelmillä voidaan luoda partikkeliefektejä kuten tuli, savu ja kipinät. Efektien luomisessa käytetään pieniä geometrisiä objekteja, kuten kahdesta kolmiosta koostuvia nelikulmioita. Partikkelijärjestelmä voi esimerkiksi määritellä yksittäisen partikkelin eliniän tietyn aikaikkunan sisällä esimerkiksi kipinäryöpyn tapauksessa [Gre14, s. 533-534]. Tällainen partikkelien syntyminen ja kuoleminen käyttää hyväkseen grafiikkaliukuhihnan kiinteää tulostevirtaa ohjaamaan partikkeliprimitiivit takaisin sävyttimelle, joka sen jälkeen voi hylätä syötteenä saadun primitiivin [Mic11].

Koska partikkeliobjektit ovat litteitä, ovat ne usein mainostaulutettu eli partikkelipinnan normaali osoittaa aina koordinaatiston kameran suuntaan. Esimerkiksi katsojalle näkyvä linssiheijastus katsottaessa valonlähdettä ensimmäisen persoonan peleissä on tästä hyvä esimerkki. Mainostaulutuksesta toinen hyvä esimerkki ovat kaksiulotteiset *spritet* eli 3D-maailmaan sijoitettavat bittikartat [Puh08, s. 216-217]. Partikkelijärjestelmistä ja mainostauluttaminen näkyy kuvaesimerkissä 5 [Tsi06].

Ennen tesselaatiosävyttimen tuloa geometriaävyttimellä hoidettiin myös esimerkiksi aiemmin esitelty dynaaminen tesselaatio [Mic11]. Vaikka geometriasävyttimellä onkin mahdollista tehdä yksinkertaista tesselaatiota, muodostuu se grafiikkaliukuhihnalle helposti pullonkaulaksi. Geometriasävyttimen parhaat käyttötapaukset juuri sellaisia, joissa sävytin voi luoda yhdestä syöteprimitiivistä useita uusia primitiivejä [Sch14].



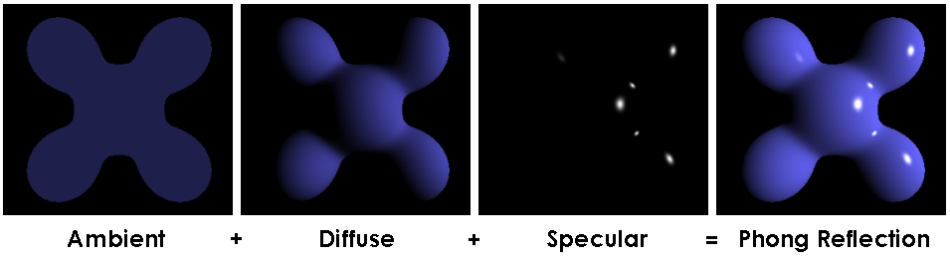
Kuva 5: Partikkelijärjestelmä ja mainostauluttaminen

4.4 Pikselisävytin

Pikselisävytin (OpenGL-terminologiassa fragmenttisävytin) on grafiikkaliukuhihnalla kiinteän rasterointivaiheen jälkeen, ja se laskee muunnoksia pikselikohtaisesti. Rasteroitujen primitiivien jokaiselle pikselille kutsutaan erikseen pikselisävytintä, joka laskee pikselille väriarvon ja *Z-syvyyden* eli pikselin etäisyyden koordinaatiston origosta. Pikselin väriarvo sekä Z-syvyys lasketaan syötteenä saatun vektorimuotoisen datan, kuten normaalivektorin, värin, tekstuurikoordinaattien, interpoloitujen valonlähteiden suuntien ja katsojan suunnan, perusteella. Erityisesti pikseliin kohdistuva valaistuksen laskenta voidaan laskea näiden tietojen perusteella [Puh08, s. 389-390].

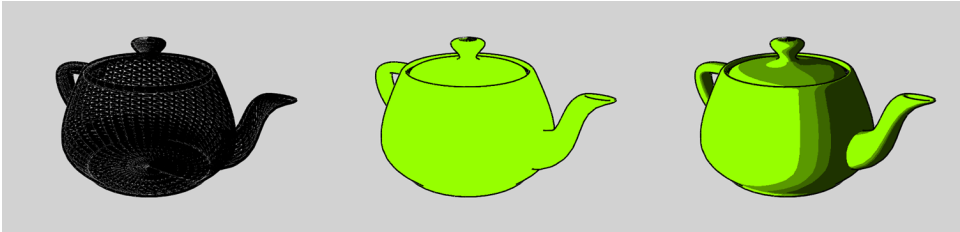
Monet näyttävät tietokonepeleissä käytettävät tehostekeinot, kuten pinnan kuhmutus tai metallipinnoista heijastuvan valon käyttäytymistä mallintava *Fresnel-heijastus* [Laz05], luodaan juuri pikselisävyttimien tasolla. Myös Gouraud-sävytystä realistisempi *Phong*-sävytys luodaan pikselisävyttimellä. Phong-sävytyksessä yhdistyy taustavalo (ambient) sekä *diffuusi* ja *spekulaari* heijastuminen. Taustavalossa kaikkiin pinnan pisteisiin kohdistuu yhtä voimakas valo. Diffuusissa heijastumisessa valo heijastuu pinnasta useaan eri suuntaan, kun taas spekulaarissa vain yhteen. Diffuusilla heijastumisella mallinnetaan siis valon heijastumista mattamaisilta pinnoilta ja spekulaarilla heijastumista kiiltäviltä pinnoilta [Puh08, s. 231-239]. Yhdistämällä nämä valaisumallit on lopputulos todellisemmalta näyttävä kuin Goraud-sävytyksessä. Phong-sävytyksen eri komponentit ovat esitelty kuvassa 6 [Smi06]. Pikselisävyttimen tärkeimpiin tehtäviin lukeutuvatkin teksturointi ja valaistuksen laskenta.

Eräs tietokonepeleissäkin hyödynnetty graafinen tehokeino on pikselisävyttimellä tehtävä *Cel-sävytys*. Cel-sävytyksessä on mahdollista luoda



Kuva 6: Phong-sävytys

perinteisen kaksiulotteisen käsinpiirretyn piirretyn näköistä kuvaa. Sävytyksessä käytetään tasaisia värejä tarkoituksellisen epäaidon kuvan luomiseksi [Luq12]. Pikseleiden väriarvot lasketaan katsojan suunnan ja pinnan normaalivektorin avulla siten, että väriarvo muuttuu portaittain sitä tummemmaksi mitä enemmän pinta osoittaa pois katsojasta tai valonlähteestä [Puh08, s. 402-404]. Cel-sävytyksestä on esimerkki kuvassa 7 [Sou07].



Kuva 7: Cel-sävytys

4.5 Laskentasävytin

Shader Model 5.0 toi myös tuen *laskentasävyttimelle* (englanniksi Compute Shader). Laskentasävyttimen on enemmän yleiskäyttöinen ja löyhästi määritelty sävytin kuin muut tutkielmassa esitellyt sävyttimet. Laskentasävytin ei varsinaisesti kuulu grafiikkaliukuhihnalle, eikä sen syötteitä ja tuloksia ole samalla tavoin tarkasti määritelty kuin muilla sävyttimillä [Kes14]. Laskentasävytin voi siis ottaa syötteenään muutakin dataa kuin grafiikkalaskennassa käytettyjä objekteja kuten kolmioita. Eräs käyttötarkoitus laskentasävyttimelle onkin *GPGPU* eli yleiskäyttöinen laskenta grafiikkapiirillä, jolloin näytönohjainten rinnakkaisuutta hyödynnetään erilaisten raskaiden laskutöiden suorituksessa [Uni15]. Tietokonepelien kohdalla laskentasävytintä voidaan hyödyntää esimerkiksi fysiikan mallinnuksen tai tekoälyn toiminnan laskemisessa [Mic11, Luku: Compute Shader Overview]. Laskentasävyttimille on olemassa myös omia ohjelmointirajapintoja, joihin kuuluvat muun muassa Nvidian CUDA, Khronos Groupin OpenCL sekä Microsoftin DirectCompute.

5 Sävytinohjelman luominen ja käyttäminen

Grafiikkapiirit tukevat laitteistotasolla eri sävytinvaiheita, joten sävytin täytyy ensin ohjelmoida ja sen jälkeen ladata grafiikkapiirille suoritettavaksi grafiikkaliukuhihnalla. Korkean tason kielellä kirjoitettu sävytin käännetään ja tulos ladataan grafiikkapiirille oikean sävyttimen ohjelmakoodiksi. Direct3D:n ollessa kyseessä sävyttimen voi kääntää ennakkoon tai lennosta ajonaikana. Tässä luvussa käydään läpi esikäännetyn kärkipistesävyttimen käyttöönoton päävaiheet [Mic11].

Kirjoitettu .hlsl -sävytintiedosto tulee ensin kääntää sävytinobjektiksi, eli HLSL:n tapauksessa .cso-tiedostoksi (complied shader object). Kääntämiskutsun voi HLSL:n tapauksessa tehdä suoraan Microsoftin Visual Studio -kehitysympäristössä käyttäen fxc.exe HLSL-kääntäjää. Itse kääntökutsu on pääpiirteissään seuraavanlainen:

```
D3DCompileFromFile( srcFile, defines,
    D3D_COMPILE_STANDARD_FILE_INCLUDE, entryPoint, profile,
    flags, 0, &shaderBlob, &errorBlob );
```

Oleellista kutsussa ovat parametrit *srcFile* eli lähdetiedosto, sekä *defines*, jonka tulee viitata sävytinfunktion nimeen. Lähdetiedostona voisi olla esimerkiksi kärkipistesävytin VSexample.hlsl ja sävytinfunktiona tiedoston koodista löytyvä VSmain-funktio. Muista kääntöfunktion parametreista ei tarvitse tässä esimerkissä välittää. Kun sävytin on ladattu ja käännetty, kapsuloidaan se sävytinobjektiksi.

```
ID3D11VertexShader  *pVertexShader
ID3DBlob pBlob;

pd3dDevice->CreateVertexShader( pBlob->GetBufferPointer(), pBlob
    ->GetBufferSize(), &pVertexShader );
```

Funktiokutsun ensimmäinen parametri on blob-muotoinen muuttuja, jossa käännetty data sijaitsee. Toinen parametri on datan koko ja kolmas on sävytinobjektin sijainti.

Lopuksi käytetään setShader-funktiota sitomaan sävytinohjelma grafiikkaliukuhihnalle.

```
ID3D11VertexShader *pVertexShader
g_d3dDeviceContext->VSSetShader( pVertexShader, nullptr, 0 );
```

Esimerkki toi esille päävaiheet sävytinohjelman kääntämisestä ja sitomisesta grafiikkaliukuhihnaan. Tarkempi tarkastelu vaatisi oman tutkielmansa, jotta kaikki asiaan liittyvät yksityiskohdat olisi mahdollista käydä läpi.

6 Yhteenveto

Tutkielmassa esiteltiin 3D-grafiikassa ja erityisesti tietokonepelien grafiikan piirtämisessä käytettäviä ohjelmoitavia sävyttimiä, niiden historiaa ja mihin niitä voi käyttää. Sävyttimet muodostavat omat moduulinsa grafiikkaliukuhihnalla, joka on vaiheittainen prosessi ruudulla näkyvän grafiikan tuottamiseksi. Ohjelmoitavat sävyttimet ovat ohjelmia, jotka suoritetaan grafiikkapiirin tukeman grafiikkaliukuhihnan tietyissä vaiheissa.

Kärkipiste- ja pikselisävytin ovat siinä mielessä perinteisiä ohjelmoitavia sävyttimiä, että niitä on käytetty 3D-grafiikan tehokeinojen perustana pisimpään. Uudempia tulokkaita ovat geometriasävytin ja viimeisinpänä tesselaatiosävytin, joista etenkin viimeinen vaikuttaisi tarjoavan monia uudenlaisia keinoja luoda entistä realistisemman näköistä grafiikka reaaliajassa.

Grafiikkaliukuhihnan järjestyksessä kärkipistesävyttimen tärkein tehtävä on laskea monikulmioiden kärkipisteille koordinaatistomuunnokset. Shader Model 5.0:n myötä laitteistotasolla tuettu tesselaatiosävytin on tehokas työkalu lisäyksityiskohtien luonnissa. Tesseloinnissa monikulmiot voidaan jakaa useisiin pienempiin osasiin, joita voidaan käyttää hyväksi uudenlaisten visuaalisten efektien, kuten nyrjäytyskartan, tekemisessä. Geometriasävytin puolestaan voi antaa tuloksenaan syötteestään riippumattoman määrän primitiivejä. Saamastaan syötteestä se voi luoda, valikoida, muokata tai hylätä tarpeelliseksi katsotun määrän tulosprimitiivejä. Tesselaatio- ja geometriasävytin ovat vaihtoehtoisia vaiheita grafiikkaliukuhihnalla, ja ollessaan käytössä ne tekevät osin samoja tehtäviä kuin kärkipistesävytin, esimerkiksi kärkipisteiden sijaintiin liittyvissä laskutoimituksissa.

Näiden vaiheiden jälkeen data siirtyy rasterointivaiheeseen, jossa rajatun kokoiseen kaksiulotteiseen piirtoikkunaan muodostetaan kuva pikseleistä. Rasteroinnin jälkeen työvuorossa on pikselisävytin, joka määrittää kunkin pikselin värin pikselikohtaisesti. Esimerkiksi valaisu ja erilaiset pintaefektit, kuten kuhmutus, tehdään pikselisävyttimellä.

Laskentasävytin ei ole varsinaisesti osa grafiikkaliukuhihnaa, mutta sen käsitteleminen samassa yhteydessä on silti luonnollista sen ollessa yleishyödyllinen sävytin. Vaikka laskentasävytin voikin suorittaa graafista prosessointia, ei sen syötteitä ja tuloksia ole rajattu pelkästään graafisiin primitiiveihin. Laskentasävyttimellä voidaan laskea grafiikkapiirissä muun muassa tekoälyn toimintaa ja fysiikanmallinnusta. Lisäksi sitä hyödynnetään erityisesti GPGPU:ssa, eli grafiikkapiirillä suoritettavassa yleishyödyllisessä laskennassa.

Itse sävyttimen kirjoittamisen yksityiskohdat riippuvat siitä, mitä sävyttimen halutaan tekevän. Päävaiheet luomisessa ja grafiikkapiirillä käyttöönotossa ovat yksinkertaistettavissa. Sävytinohjelma kirjoitetaan korkean tason kielellä, käännetään grafiikkapiirille ohjelmakoodiksi ja sidotaan grafiikkaliukuhihnaan suoritettavaksi sävytinohjelmaa vastaavalla sävyttimellä.

Lähteet

[Ake02] Akenine-Möller, Tomas ja Haines, Eric: *Real-Time Rendering*. A K Peters/CRC Press, 2. painos, 2002.

[Bur09] Burikin, Lucas: *D3D Shading Modes*, 2009. `https://commons.wikimedia.org/wiki/File:D3D_Shading_Modes.png#/media/File:D3D_Shading_Modes.png`, Public domain.

[Gou71] Gouraud, Henri: *Continuous shading of curved surfaces*. IEEE Transactions on Computers C–20 June: s. 623-629, 1971.

[Gre14] Gregory, Jason: *Game Engine Architecture*. A K Peters/CRC Press, 2. painos, 2014.

[Kes14] Kessenich, John, Baldwin, Dave ja Randi Rost: *The OpenGL® Shading Language*. `https://www.opengl.org/registry/doc/GLSLangSpec.4.40.pdf`, 2014.

[Khr15] Khronos Group ja SGI: *OpenGL*. `https://www.opengl.org/wiki/`, 2015.

[Laz05] Lazániy, István ja Szirmay-Kalos, László: *Fresnel term approximations for metals*. WSCG '2005: Short Papers: The 13th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision 2005 in co-operation with EUROGRAPHICS, s. 77-80., 2005.

[Luq12] Luque, Raul Reyes: *The Cel Shading Technique*. `https://raulreyesfinalproject.files.wordpress.com/2012/12/dissertation_cell-shading-raul_reyes_luque.pdf`, 2012.

[Mic11] Microsoft: *Direct3D 11 Graphics*. `https://msdn.microsoft.com/en-us/library/windows/desktop/ff476080(v=vs.85).aspx`, 2011.

[Nvi03] Nvidia Corporation: *The Cg Tutorial*. `http.developer.nvidia.com/CgTutorial/cg_tutorial_chapter01.html`, 2003.

[Nvi10] Nvidia Corporation: *DirectX 11 Tessellation*. `http://www.nvidia.com/object/tessellation.html`, 2010.

[Nvi12] Nvidia Corporation: *Cg Documentation*. `http://http.developer.nvidia.com/Cg/`, 2012.

[Pix15] Pixar: *A Brief Introduction To RenderMan*. `http://renderman.pixar.com/view/brief-introduction-to-renderman`, 2015.

[Puh08] Puhakka, Antti: *3D-grafiikka*. Talentum, 1. painos, 2008.

[Sch14] Schäfer, H. ja Niessner M. ja Keinert, B. ja Stamminger, M. ja C. Loop: *State of the Art Report on Real-time Rendering with Hardware Tessellation*. EUROGRAPHICS 2014/ S, 2014.

[She08] Sherrod, Allen: *Game Graphics Programming*. Charles River Media, 1. painos, 2008.

[Smi06] Smith, Brad: *Phong components version 4*, 2006. `https://en.wikipedia.org/wiki/Phong_reflection_model#/media/File:Phong_components_version_4.png`, CC BY-SA 3.0.

[Sou07] Sourd, Nicolas: *Utah Teapot Cel-shading*, 2007. `https://en.wikipedia.org/wiki/Cel_shading#/media/File:Celshading_teapot_large.png`, CC BY 2.5.

[Tsi06] Tsiombikas, John: *Ad hoc particle system used to simulate a galaxy*, 2006. `https://commons.wikimedia.org/wiki/File:Particle_sys_galaxy.jpg#/media/File:Particle_sys_galaxy.jpg`, Public domain.

[Uni15] Unity Technologies: *DirectX 11 and OpenGL Core*. `http://docs.unity3d.com/Manual/UsingDX11GL3Features.html`, 2015.

[Vla01] Vlachos, Alex ja Peters, Jörg ja Boyd, Chas ja Mitchell, Jason L.: *Curved PN triangles*. I3D '01 Proceedings of the 2001 symposium on Interactive 3D graphics, 2001.