Valtteri Vallius

Läpinäkyvyyden reaaliaikaisen renderöimisen perusteet

Tietotekniikan kandidaatintutkielma

2. elokuuta 2020

Jyväskylän yliopisto

Informaatioteknologian tiedekunta

Tekijä: Valtteri Vallius

Yhteystiedot: `vaakakva@student.jyu.fi`

Ohjaaja: Timo Tiihonen

Työn nimi: Läpinäkyvyyden reaaliaikaisen renderöimisen perusteet

Title in English: Basics of Rendering Transparency in Real-Time

Työ: Kandidaatintutkielma

Opintosuunta: Tietotekniikka

Sivumäärä: 42+0

Tiivistelmä: Nykyaikaiset näytönohjaimet kykenevät renderöimään kolmiulotteisia näkymiä reaaliajassa tehokkaasti. Monimutkaisen, päällekkäisen ja monitasoisen läpinäkyvyyden tehokas primitiivipohjainen renderöiminen näytönohjaimilla on kuitenkin osoittautunut haasteelliseksi ongelmaksi suorituskyvyn näkökulmasta. Tässä kandidaatintutkielmassa tehdään katsaus läpinäkyvyyden reaaliaikaisen renderöimisen perusteisiin. Lisäksi tarkastellaan muutamia menetelmiä, joiden avulla voidaan renderöidä tehokkaasti myös vaativampia läpinäkyvyyttä sisältäviä skenaarioita videopeleissä tai vastaavissa reaaliaikaisissa grafiikkasovelluksissa.

Avainsanat: OpenGL, tietokonegrafiikka, läpinäkyvyys, reaaliaikainen renderöiminen, järjestysriippumaton läpinäkyvyys

Abstract: Modern graphics processing units are capable of rendering three-dimensional scenes in real-time efficiently. However, complex and overlapping primitive-based real-time transparency remains problematic when it comes to overall performance. This bachelor's thesis governs the basics of rendering primitive-based transparency and takes a peek at various transparency rendering techniques that are suited for real-time applications such as video games or similar computer graphics applications.

Keywords: OpenGL, computer graphics, transparency, real-time rendering, alpha blending, alpha compositing, order-independent transparency

Esipuhe

Alunperin idea kandidaatintutkielmalleni lähti siitä, kun pelasin talvisena iltana Crash Bandicoot 1 -peliä Playstation 1 -konsolilla. Pelissä on mahdollista kerätä isoja värikkäitä jalokiviä, joiden yksityiskohtien kehittämiseen on mielestäni käytetty ihailtava määrä aikaa. Jalokivet ovat pyöriviä, kolmiulotteisia ja niiden pinnat ovat liukuvasti läpinäkyviä. Lisäksi jalokiviin on lisätty kimaltelua jäljitteleviä heijastuksia, jotka tosin ovat toteutettu kaksiulotteisilla kuva-animaatioilla.

Jalokivistä hurmaantuneena päätin, että haluan ohjelmoida itse vastaavanlaisia jalokiviä OpenGL-rajapintaa hyödyntäen. Kun pääsin siihen vaiheeseen, että olin tehnyt teksturoidun jalokiven Blender-ohjelmassa ja ladannut sen onnistuneesti OpenGL-sovellukseeni, oli aika kokeilla läpinäkyvyysefektejä muuntamalla fragment shaderin avulla pikseleiden *alpha*-arvoa. Tämän tehdessäni tulos ei ollut lainkaan sitä mitä odotin, sillä läpinäkyvyys näytti kaikin puolin väärältä.

Lähdin selvittämään, mistä tämä johtuu, ja pian huomasin, että läpinäkyvyyden renderöiminen onkin itseasiassa paljon monimutkaisempi asia kuin mitä aluksi kuvittelin. Samoihin aikoihin minun piti valita kandidaatintutkielmalleni aihe, ja kuten arvata saattaa, aiheekseni valikoitui läpinäkyvyyden reaaliaikaisen renderöimisen perusteet.

Jyväskylässä 2. elokuuta 2020

Valtteri Vallius

Termiluettelo

Blending	*Blending*-termillä viitataan operaatioon, jonka avulla kaksi väriä yhdistetään yhdeksi uudeksi väriksi.
Emissio	Emissiivisillä väreillä viitataan väreihin, joiden kontribuutio kokonaiskirkkauteen yhdistettäessä muiden värien kanssa on aina positiivinen. Emissiivisiä värejä voidaan käyttää esimerkiksi tulen tai muiden kirkkaiden partikkeliefektien renderöimisessä.
Fragmentti (engl. fragment)	Primitiivin pinnalla oleva pieni alue, joka on muutamaa poikkeusta huomioimatta rinnastettavissa näytöllä esitettävään lopulliseen pikseliin.
Geometria	Geometrialla viitataan kokonaisuuteen, joka koostuu useasta primitiivistä. Geometrialla voidaan tarkoittaa esimerkiksi kaikkia niitä primitiivejä, joista yksi pelihahmo koostuu. Näytönohjaimelle voidaan määrittää piirrettävä geometria yhtenäisenä sarjana primitiivejä, jolloin kaikki geometrian primitiivit piirretään samanaikaisesti hyödyntäen näytönohjaimen rinnakkaislaskentaominaisuuksia.
Heijastus	Tarkoittaa kappaleiden pinnoista ulospäin heijastuvia valonsäteitä.
Kuvapuskuritekstuuri	Näytönohjaimen muistissa sijaitseva tekstuuri, johon voidaan renderöidä samalla tavalla kuin näyttölaitteen ruudulle. Kuvapuskuritekstuuriin voidaan tallentaa arvoja muistiin, ja niitä voidaan näytteistää myöhemmin muiden renderöintioperaatioiden yhteydessä.
Näkymä	Näyttölaitteen ruudulle piirtyvä näkymä, joka on renderöity näytönohjainta hyödyntäen. Näkymässä näkyy tyypillisesti näyttölaitteen kaksiulotteiseen koordinaattiavaruuteen projektoitu geometria. Näkymällä viitataan tyypillisesti renderöinnin lopulliseen tulokseen.

Objekti	Objekti-termiä käytetään ilmaisemaan renderöitävää kokonaisuutta. Kokonaisuus voi koostua useammasta erillisestä geometriasta tai vain yhdestä primitiivistä.
OIT	*Order-Independent Transparency* eli järjestysriippumaton läpinäkyvyys. *OIT*-menetelmille on tyypillistä, että niissä läpinäkyvien primitiivien renderöiminen ei vaadi järjestelyä syvyyden suhteen.
Piirtokutsu	Komento jolla näytönohjainta pyydetään renderöimään määritetty geometria. Geometrian lisäksi piirtokutsun lopputulokseen vaikuttavat hetkellisesti aktiivisena olevat shader-ohjelmat sekä muut konfiguroitavat näytönohjaimen tila-attribuutit.
Piirtosilmukka (engl. render loop)	Piirtosilmukaksi kutsutaan sitä kohtaa ohjelmasta, jossa tyypillisesti kaikki grafiikan piirtämiseen liittyvät funktiokutsut tapahtuvat. Piirtosilmukkan päätteeksi näyttölaitteelle päivitetään uusi päivitetty kuva ohjelman grafiikasta. Piirtosilmukkaa pyritään suorittamaan usein yhtä monta kertaa sekunnissa kuin mikä on näyttölaitteen päivitystaajuus.
Primitiivi (engl. primitive)	Tarkoittaa näytönohjaimella renderöitävää yksinkertaista monikulmiota, joka on määritetty kärkipisteiden avulla. Kärkipisteiden sisälle jäävä tila kattaa primitiivin pinnan, jonka sisälle jäävien fragmenttien väriarvot voidaan asettaa tai laskea halutunlaiseksi fragment shaderilla. Kolmio lienee relevantein esimerkki primitiviistä.
RGBA	RGBA-termillä viitataan värikomponentteihin, missä *R* kuvastaa punaisen väriosuuden painoarvoa lopullisessa värissä. Vastaavasti *G* ja *B* kuvastavat vihreän ja sinisen värien painoarvoja. *A*:n arvo viittaa värin läpinäkyvyyden asteeseen (*alpha*). Komponenttien mahdolliset arvot jakavat yleensä saman skaalan. Riippuen esitystavasta, jokainen arvo voi olla esimerkiksi liukulukuina esitettynä väliltä *0.0-1.0* tai kokonaislukuina esitettynä väliltä *0-255*. Liukulukuina esitetty arvoväli *0.0-1.0* lie-

	nee tyypillisin käytetty skaala, sillä se kertoo prosentuaalisen painoarvon selvemmin. (Porter ja Duff 1984)
Shader-ohjelmat	Shader-ohjelmat ovat näytönohjaimella suoritettavia pieniä ohjelmia, joiden avulla määritetään sitä miten näytönohjain piirtää sille lähetetyn geometrian. Shader-ohjelmia on eri tyyppisiä jokaiseen erilliseen näytönohjaimen piirtoprosessin vaiheeseen. OpenGL-standardissa tyypillisimmät ja pakolliset shader-ohjelman tyypit ovat *vertex shader* ja *fragment shader*.

Kuviot

Kuvio 1. Ongelmallisesti asettuneet läpinäkyvät primitiivit 11
Kuvio 2. Syvyysfunktion toiminta WBOIT-menetelmässä 17
Kuvio 3. Stochastic Transparency -implementaatio 20
Kuvio 4. Puutteellinen satunnaisfunktio Stochastic Transparency -menetelmässä 26

Sisällys

1 JOHDANTO 1

2 LÄPINÄKYVYYDEN MALLINTAMINEN 3
- 2.1 Näkymäpohjaiset mallit 3
- 2.2 Näkymien laskeminen 5

3 LÄPINÄKYVYYDEN REAALIAIKASEEN RENDERÖIMISEEN LIITTYVÄT HAASTEET JA ONGELMAT 9

4 KESKEISET LÄPINÄKYVYYDEN RENDERÖINTIMENETELMÄT 12
- 4.1 Sort-Independent Alpha Blending 12
- 4.2 Weighted Average 14
- 4.3 Blended Weighted Order-Independent Transparency 16
- 4.4 Screen-Door Transparency 19
- 4.5 Stochastic Transparency 20
- 4.6 Depth-peeling 27

5 YHTEENVETO JA POHDINTA 29

LÄHTEET 32

1 Johdanto

Tässä kandidaatintutkielmassa tarkastellaan ja tutkitaan läpinäkyvyyttä ilmiönä reaaliaikaisessa renderöimisessä. Tarkoituksena on tehdä katsaus siihen, mitä läpinäkyvyys itseasiassa tarkoittaa teknisestä näkökulmasta, kun läpinäkyvyyttä käsitellään reaaliaikaisen tietokonegrafiikan kontekstissa. Tutkielma pyrkii määrittelemään läpinäkyvyyden renderöimisen keskeiset perusteet selkeästi ja ymmärrettävästi. Perusteiden lisäksi tehdään katsaus monimutkaisempiin läpinäkyvyyden renderöimistä käsitteleviin menetelmiin, joiden valinnassa on painotettu suorituskykyä realismin ja visuaalisen oikeellisuuden sijasta. Tälläisien menetelmien implementoiminen ja hyödyntäminen soveltuu parhaiten reaaliaikaisiin tietokoneohjelmiin, kuten videopeleihin tai audiovisuaalisiin taideteoksiin, sillä näissä suorituskyky ja ohjelman responsiivisuus saattavat usein olla tärkeämpiä kuin visuaalinen oikeellisuus ja tarkkuus. Nämä asiat huomioiden, tästä kandidaatintutkielmasta saattaa olla hyötyä sellaiselle lukijalle, jolla on aiempaa kokemusta tietokonegrafiikasta ja joka etsii sopivia ratkaisuja ja vastauksia monimutkaisempaa läpinäkyvyyttä sisältävien ohjelmien toteutukseen.

Ilman syvällisempää tutustumista aiheeseen, läpinäkyvyys saattaa tuntua triviaalilta käsitteeltä. Tietokonegrafiikan kontekstissa läpinäkyvyys on kuitenkin yllättävän monimuotoinen aihe. Läpinäkyvyyden renderöiminen voidaan toteuttaa useaa erilaista lähestymistapaa mukaillen, joista jokaisella eri tavalla on omat heikkoudet ja vahvuudet, kun tapoja vertaillaan suorituskyvyn ja visuaalisen lopputuloksen näkökulmasta. Tässä tutkielmassa keskitytään kuitenkin pääsääntöisesti sellaisen läpinäkyvyyteen, mikä tuotetaan näytönohjaimen avulla renderöimällä osittain läpinäkyviä primitiivejä toistensa päälle.

Kun tutkimuksessa puhutaan läpinäkyvyydestä suhteessa objekteihin, primitiiveihin, tasoihin ja vastaaviin asioihin, niin lähtökohtaisesti oletetaan, että käsiteltävät asiat eivät ole täysin läpinäkyviä, vaan nimenomaan osittain läpinäkyviä. Jos tarkastelussa on täysin läpinäkyvä objekti, niin siitä on mainuttu erikseen, sillä riippuen käytettävän menetelmän toteutuksesta, tietyissä tapauksissa täysin läpinäkyvien objektien renderöiminen saattaa tuottaa sivuvaikutuksia lopputulokseen.

Säteenseurantaan liittyvä läpinäkyvyyttä ei juurikaan käsitellä, sillä siinä läpinäkyvien näky-

mien muodostaminen on triviaalimpaa toteuttaa kuin perinteisessä primitiiveihin pohjautuvassa renderöimisessä. Reaaliaikaiseen säteenseurantaan liittyy kuitenkin omat ongelmansa suorituskyvyn puolesta, joten ainakaan vielä sen avulla ei voida ratkaista kaikkia läpinäkyvyyden renderöimiseen liittyviä haasteita. Volumetriset renderöintimenetelmät, missä kolmiulotteisesta, korkean resoluution datasta näytteistämällä konstruoidaan näkymiä, ei myöskään itsessään sisälly tämän tutkielman aihepiiriin.

Tutkielman rakenne on jaoteltu seuraavanlaisiin aihealueisiin. Alussa käydään läpinäkyvyyden renderöimiseen liittyviä käsitteitä yleisellä tasolla, jotta myöhemmin tarkastelussa olevia algoritmeja, kaavoja sekä menetelmiä olisi helpompi selittää ja ymmärtää. Tämän jälkeen käsitellään blending-operaattoreita, joita hyödyntämällä voidaan yhdistellä läpinäkyväksi määritettyjä väriarvoja laskennallisesti. Kun blending-operaattoreista on selitetty perusteet, käydään seuraavaksi läpi läpinäkyvyyden reaaliaikaseen renderöimiseen liittyviä haasteita ja sitä miten ne ovat yhteydessä blending-operaattoreihin. Haasteiden jälkeen tehdään katsaus erilaisiin menetelmiin, joiden avulla pyritään renderöimään läpinäkyvyyttä tehokkaasti siten, että visuaalinen oikeellisuus pysyisi mahdollisimman hyvänä. Lopuksi pohditaan eri menetelmien eroavaisuuksia ja niiden soveltuvuutta erilaisiin tilanteisiin.

Kaikki tutkielmassa käsiteltävät kuvaukset näytönohjaimien ominaisuuksista ja toiminnoista pohjautuvat moderniin OpenGL-standardiin.

2 Läpinäkyvyyden mallintaminen

Akenine-Moller, Haines ja Hoffman (2008, s. 134) tekevät karkeat jaon läpinäkyvyyden suhteen tietokonegrafiikassa määrittämällä läpinäkyvyyden kahteen eri kategoriaan. Nämä kategoriat ovat näkymäpohjainen läpinäkyvyys (engl. *view-based*) ja valopohjainen läpinäkyvyys (engl. *light-based*). Näiden ero on se, että näkymäpohjaisessa tapauksessa itse kappaleen läpinäkyvyyttä on muutettu suoraan sekoittamalla keskenään kappaleen värejä sen takana tai edessä olevien pikseleiden väriarvojen kanssa. Valopohjaisessa tapauksessa viitataan siihen ilmiöön, missä valonsäteet kulkevat läpinäkyvien kappaleiden lävitse ja muokkaavaat mahdollisesti näkymän muitakin objekteja valon ja materian fysikaalisiin malleihin pohjautuen.

2.1 Näkymäpohjaiset mallit

Kun läpinäkyvyyttä tarkastellaan teknisestä näkökulmasta, niin on syytä puhua ensiksi värien rakenteesta lyhyesti. Tietokonegrafiikassa väri esitetään usein RGB-värimallia hyödyntäen, mikä tarkoittaa sitä että väri esitetään kolmen perusvärin yhdistelmänä. Perusväreinä, joiden yhdistelmästä värin lopullinen sävy määräytyy, toimivat punainen (engl. Red), vihreä (engl. Green) ja sininen (engl. Blue). Läpinäkyvyyden ilmaisemista varten on kuitenkin tarpeen, että RGB-yhdistelmään lisätään tietoa, jolla voidaan huomioida myös pikselin mahdollinen läpinäkyvyys. Porter ja Duff (1984) esittävät tavan ilmaista pikselin läpinäkyvyyttä lisäämällä pikselin värikomponentteihin yhden lisäkomponentin, *alpha*-komponentin, jonka tarkoituksena on määrittää pikselin läpinäkyvyys. Tähän esitystapaan viitataan vastaavasti lyhenteellä *RGBA*. Väriarvon kaikki komponentit saatetaan usein esittää sulkujen sisällä pilkuilla eroteltuna esimerkiksi seuraavasti (2.1),

$$(1.0,\ 0.0,\ 0.0,\ 1.0) \tag{2.1}$$

$$(Red, Green, Blue, Alpha) \tag{2.2}$$

missä siis kolme ensimmäistä liukulukua kuvastavat väriä ja neljäs liukuluku läpinäkyvyyttä.

Jos värikomponenttien yhteinen skaala on esitetty esimerkiksi käyttäen väliä *0.0-1.0*, niin tällöin *alpha*-komponentin ollessa *0.0* on pikseli täysin läpinäkyvä (engl. transparent), ja vastaavasti kun *alpha*-komponentin arvo on *1.0*, on pikseli läpinäkymätön (engl. *opaque*).

Tämä on tyypillinen tapa käsitellä ja tulkita läpinäkyviä värejä, mutta esitystapa ja merkitys saattaa vaihdella käytettävästä tekniikasta riippuen. Siksi on tärkeää aina varmistaa väriarvojen merkitys suhteessa käytettävään tekniikkaan. Tietyissä tilanteissa värien saatetaan haluta mallintavan esimerkiksi valonlähteitä, joita yhdistettäessä lopullisen väriarvon kirkkaus kasvaa inkrementaalisesti kirkkaammaksi. McGuire ja Bavoil (2013) kutsuvat tälläistä läpinäkyvyyttä emissiiviseksi (engl. emissive). Emissiivisen läpinäkyvyyden *alpha*-komponentin arvo ei välttämättä tarvitse rajoittua välille *0.0-1.0*, sillä *1.0* suurempia arvoja käyttämällä voidaan säädellä valon voimakkuutta laajemmin. Emissiivinen läpinäkyvyys voidaan toteuttaa muun muassa esikerrotun *alpha*-arvon avulla.

Toinen tapa esittää osittain läpinäkyviä värejä on kertoa jokainen värikomponentti *RGB alpha*-komponentilla *A*. Esimerkiksi väri (2.3) muuntuisi tällöin muotoon (2.4).

$$(1.0,\ 0.5,\ 0.0,\ 0.5) \tag{2.3}$$

$$(0.5,\ 0.25,\ 0.0,\ 0.5) \tag{2.4}$$

Tyypillisesti *alpha*-komponentilla esikerrottuja värejä käyttäessä värien merkityksen halutaan kuvastavan sellaista tilannetta, missä *alpha*-komponentin suuruus kuvastaa kyseisen värin kykyä päästää sen takana olevia värejä läpi ja *RGB*-komponenttien arvot kuvastavat kontribuutiota, joka takana oleviin väreihin lisätään. Jos *alpha*-komponentilla esikerrottua esitystapaa verrataan perinteiseen esitystapaan **täysin läpinäkyvän värin tapauksessa**, niin esikerrotun värin kaikkien *RGBA*-komponenttien arvojen on pakko olla 0.0, kun taas perinteisessä esitystavassa täysin läpinäkyvän värin RGB-arvot voivat olla mitä tahansa väliltä 0.0-1.0, mutta *alpha*-komponentin arvon tulee olla 0.0. (Microsoft 2020)

McGuire ja Bavoil (2013) suosittelevat esikerrottujen värien käyttämistä muutamasta syystä johtuen. Esikerrotuilla väreillä voidaan kontribuoida lopulliseen tulokseen emission avulla siinäkin tilanteessa, kun väri on täysin läpinäkyvä. Tämä saattaa olla hyödyllistä erilaisten visuaalisten tehosteiden toteuttamisessa. Lisäksi McGuire ja Bavoil (2013) kuvailevat esikerrottujen värien tuottavan vähemmän virheitä tekstuurifiltteröinnin sekä mipmap-tekstuurien generoimisen yhteydessä.

Emissiivisten esikerrottujen värien toimintaa voidaan havainnoilllistaa tilanteella, missä kaukaisuudesta kameraan päin lähestyvä säde kulkeutuu läpinäkyvien päällekkäisten fragmenttien lävitse muokaten jokaisen lävistyksen yhteydessä säteen väriarvoa. Säteen värin arvoksi alustetaan läpinäkyvien fragmenttien takana olevan ensimmäisen täysin läpinäkymättömän fragmentin väri, jota voidaan tämän esimerkin yhteydessä kutsua taustaväriksi. Tälläisessä tilanteessa jokaisella läpinäkyvällä fragmentilla on esikerrottu RGBA-arvo, jossa RGB-komponentit kuvastavat säteeseen summattavaa värikontribuutioita, ja A-komponentti kuvastaa peittoastetta. Peittoaste määrittää, kuinka paljon lävistettävä fragmentti suodattaa valonsäteen senhetkisestä kerrytetystä väristä pois. Kun kaikki läpinäkyvät fragmentit ovat käyty läpi, jäljelle jää lopullinen väri, johon on koostettu niin taustavärin kuin läpinäkyvien fragmenttien värikontribuutiot yhdeksi kokonaisuudeksi. (McGuire ja Bavoil 2013)

Edellä kuvattu tilanne ei anna kuitenkaan vastausta siihen, miten värien yhdistäminen käytännössä tapahtuu. Kahden läpinäkyvän värin yhdistämistä kutsutaan *blendaamiseksi*, ja siitä kerrotaan tarkemmin seuraavassa kappaleessa.

2.2 Näkymien laskeminen

Lopullinen kuva tai näkymä, joka piirtyy näyttölaitteelle, on lähes jokaisessa modernissa tietoteknisessä laitteessa kaksiulotteinen taulukko vierekkäisiä pikseleitä. Esimerkiksi tietokoneen näytössä ei ole syvyyssuunnassa päällekkäisiä pikseleitä, joita olisi mahdollista ohjailla erikseen ja täten saavuttaa päällekkäisiä, läpinäkyviä ilmiöitä fyysisellä tasolla näyttölaitteen toimesta.

Tästä johtuen monimutkaisien näkymien ja kuvien muodostaminen tietokoneella ja niiden esittäminen näyttölaitteella on prosessi, missä väriarvot koostetaan ohjelmallisesti tietoko-

neen muistissa yhdeksi kaksiulotteiseksi taulukoksi väriarvoja, jotka voidaan koostamisen jälkeen lähettää ja esittää näyttölaitteella. Koska jokainen näyttölaitteen pikseli voi esittää kerrallaan vain yhden värin, tarvitaan väriarvojen yhdistämiseen mekanismeja sellaisia tilanteita varten, missä useampi läpinäkyvä väriarvo olisi päällekkäin saman näyttölaitteella esitettävän pikselin kohdan päällä.

Päällekkäisten läpinäkyvien värien yhdistämiseen voidaan käyttää erilaisia blending-operaattoreita. Porter ja Duff (1984) esittävät 12 erilaista operaattoria, joiden avulla värien yhdistäminen voidaan toteuttaa. Näistä 12 operaattoreista he sanovat ainakin muutaman olevan hyödyllisiä, joista hyödyllisin lienee *OVER*-operaattori.

$$C_{dst} = C_{src} \cdot A_{src} + (1 - A_{src}) \cdot C_{dst} \tag{2.5}$$

OVER-operaattoria (2.5) voidaan käyttää sellaisessa tilanteessa, missä taustavärin päälle halutaan renderöidä jokin osittain läpinäkyvä väri. C_{dst} tarkoittaa taustalla olevaa väriä ja C_{src} taustan päälle renderöitävää väriä. Värien C_{src} ja C_{dst} oletetaan olevan RGB-muodossa (2). A_{src} tarkoittaa taustavärin päälle rendöitävän värin läpinäkyvyyttä eli sen *alpha*-komponenttia. Kaava on esittetty siten, että laskettu tulos asetetaan suoraan taustavärin uudeksi arvoksi, sillä tämä on tyypillinen toteutus näytönohjaimien ohjelmointirajapinnoissa (The Khronos Group 2014).

Kuten *OVER*-operaattorin kaavasta (2.5) voi päätellä, kun taustan päälle blendattavan väriarvon *alpha*-komponentti A_{src} on arvoltaan *0.0*, niin tällöin kaava supistuu muotoon $C_{dst} = C_{dst}$, eli pelkkä taustaväri jää jäljelle. Jos taas päälle blendattavan värin *alpha*-komponentin A_{src} arvo on *1.0*, niin tällöin kaava supistuu muotoon $C_{dst} = C_{src}$, eli taustavärin vaikutus häviää kokonaan ja vain päälle blendattava väri jää jäljelle. A_{src} arvon ollessa jotakin välillä $0.0 - 1.0$ lopputuloksena on väriyhdistelmä C_{dst}:n ja C_{src}:n väliltä. (Porter ja Duff 1984; NVIDIA GameWorks 2014)

Jos OVER-operaattoria haluaa käyttää *alpha*-komponentilla esikerrottujen värien kanssa (2.1), niin tällöin OVER-operaattori tulee muuttaa seuraavanlaiseen (2.6)

$$C_{dst} = C_{src} + (1 - A_{src}) \cdot C_{dst} \tag{2.6}$$

muotoon. Nyt kaavasta (2.6) on vain poistettu A_{src}, jolla C_{src} kerrottiin *OVER*-operaattorin alkuperäisessä muodossa (2.5). Tämä käy järkeen, sillä kun käytössä on *alpha*-arvolla esikerrotut värit, niin kaavan (2.6) C_{src} arvo kuvastaa jo valmiiksi samaa asiaa kuin $C_{src} \cdot A_{src}$. Lisäksi kuten aiemmin todettiin (2.1), niin *alpha*-arvolla esikerrotun *RGBA*-värin *alpha*-komponentti määrittää sitä miten paljon kyseisen värin takana olevasta väristä, eli taustaväristä, suodatetaan pois: $(1 - A_{src}) \cdot C_{dst}$.

OpenGL-rajapinnassa on sisäänrakennettuna tuki blending-operaattoreille. OpenGL-standardi tarjoaa keinon kustomoida blending-operaattorin toimintaa valmiiden vakioiden avulla. (Sellers, Wright ja Haemel 2014, s. 357)

Blendattavaksi tarkoitettua geometriaa tulee piirtää vain silloin kun blendaus on kytketty päälle, mikä tapahtuu käyttämällä komentoa *glEnable* listauksen (2.1) mukaisesti.

Listing 2.1. "Blendauksen aktivoiminen OpenGL-rajapinnan avulla."

```
1 glEnable(GL_BLEND);
2 // Blendattava geometria piirretään tässä välissä.
3 glDisable(GL_BLEND);
```

Kaava (2.7) kuvastaa miten OpenGL käsittelee sisäisesti blending-operaattoreita.

$$C_R = C_S \cdot F_S + C_D \cdot F_D \tag{2.7}$$

Kaavassa (2.7) C_R tarkoittaa blending-operaattorin tuottamaa väriä, C_S seuraavaksi piirrettävän fragmentin väriarvoa ja C_D kuvapuskurissa ennestään olevaa väriä. F_S ja F_D ovat määritettävissä olevia OpenGL-standardin tarjoamia vakioita, joiden avulla voidaan säätää miten blending-operaattori toimii. Esimerkiksi asettamalla F_S -parametrin arvoksi vakion *GL_ZERO* ja F_D-parametrin arvoksi vakion *GL_ONE*, muuntuisivat parametrit F_S nollaksi ja F_D ykköseksi, jolloin seuraavaksi piirrettävän fragmentin väri supistuisi kaavasta kokonaan

pois ja kuvapuskurissa ennaltaan oleva väri tulisi blendauksen tulokseksi C_R. Jos haluttaisiin käyttää tavanomaista *OVER*-operaattoria (2.5), niin tämä tapahtuisi asettamalla F_S arvoksi *GL_SRC_ALPHA*, ja F_S arvoksi *GL_ONE_MINUS_SRC_ALPHA* (Sellers, Wright ja Haemel 2014, s. 357)

Kaavan (2.7) F_S ja F_D -parametrien arvoja voidaan muuttaa funktionkutsun (2.2)

Listing 2.2. "*glBlendFunc*-funktion määritelmä."

```
1 glBlendFunc(GLenum sfactor, GLenum dfactor)
```

avulla, missä F_S vastaa *sfactor*-arvoa ja F_D *dfactor*-arvoa (The Khronos Group 2014).

3 Läpinäkyvyyden reaaliaikaseen renderöimiseen liittyvät haasteet ja ongelmat

Nykyiset näytönohjaimet pystyvät renderöimään **läpinäkymättömiä**, fragmenteista koostuvia primitiivejä tehokkaasti hyödyntäen rinnakkaislaskentaa ja syvyyspuskuria (engl. *depth buffer*). Tämä perustuu siihen, että läpinäkymättömien fragmenttien taakse ei voi nähdä, joten lopputuloksen kannalta ei ole väliä, mitä läpinäkymättömän fragmentin takana on. Tälläinen syvyyteen perustuva suodatus voidaan toteuttaa syvyyspuskuria hyödyntäen, mikä löytyy sisäänrakennettuna jokaisesta nykyaikaisesta näytönohjaimesta.

Syvyyspuskuri on kaksiulotteinen taulukko syvyysarvoja, mikä kattaa tyypillisesti kokonaan sen alueen missä renderöiminen tapahtuu. Jokaiseen fragmenttiin kiinnitetään syvyysarvo, joka kuvastaa sen etäisyyttä näkymäalueen pinnasta. Kun fragmentti halutaan renderöidä, voidaan syvyyspuskurin avulla tarkastaa, onko renderöitävän fragmentin syvyysarvo pienempi kuin mitä syvyyspuskurissa jo ennestään on. Jos fragmentin syvyysarvo on pienempi, fragmentti hyväksytään renderöitävien fragmenttien joukkoon, ja syvyyspuskuriin tallennetaan juuri renderöidyn fragmentin syvyysarvo seuraavaa syvyystarkastusta varten. Tämä on tyypillinen tapa hyödyntää syvyyspuskuria, mutta sen voi ohjelmoida tekemään myös erilaisia tarkastuksia ja suodatuksia. (Sellers, Wright ja Haemel 2014, s. 46)

Kuitenkin kun kyseessä on tilanne, missä halutaan renderöidä eri syvyyden omaavia päällekkäisiä läpinäkyviä fragmentteja, muodostuu ongelmaksi se, että syvyyspuskurin hyväksymä läpinäkymätön fragmentti tulee yhdistää läpinäkyvien fragmenttien kanssa yhdeksi fragmentiksi (Carpenter 1984).

Tästä johtuen läpinäkyvyyttä renderöiville menetelmille on tyypillistä, että kaikki näkymän läpinäkymättömät primitiivit renderöidään aina ensiksi yhtenäiseksi taustaksi hyödyntäen syvyyspuskuria, jonka jälkeen syvyyspuskuri kytketään pois käytöstä ja keskitytään renderöimään näkymän läpinäkyviä primitiivejä taustan päälle. (Sellers, Wright ja Haemel 2014, s. 558)

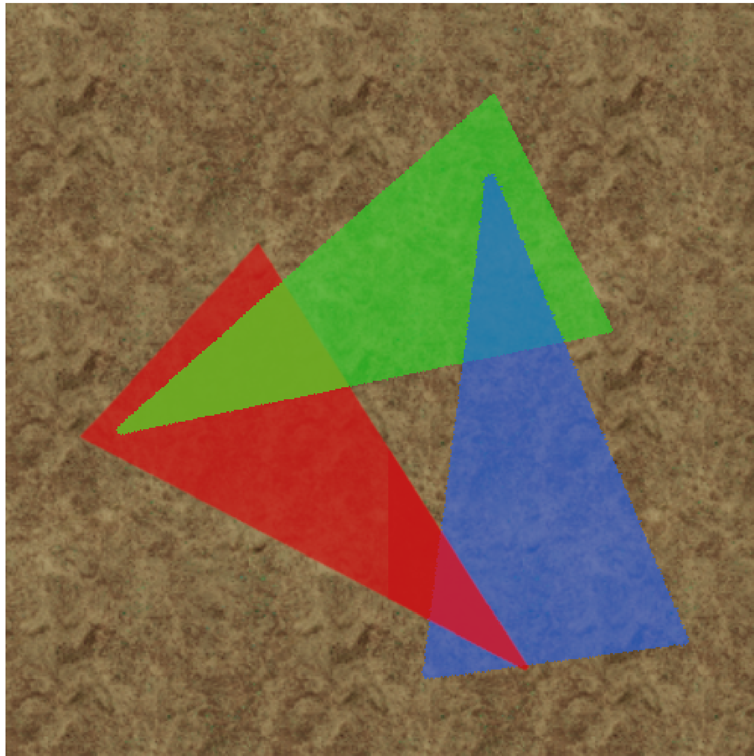
Blending-operaattori voi yhdistää kerrallaan vain kaksi väriä, joten tilanteessa jossa yhdis-

tettäviä päällekkäisiä värejä on enemmän kuin kaksi, on mahdollista yhdistää värit useassa eri järjestyksessä. Tämä on ongelmallista, sillä riippuen blending-operaattorin toteutuksesta, operaattori ei välttämättä ole kommutatiivinen. Toisin sanoen, värien yhdistämisen lopputulos saattaa vaihdella riippuen siitä missä järjestyksessä värit yhdistettiin. Tästä syystä jos käytettävä blending-operaattori ei ole kommutatiivinen, tulee yhdistettävät fragmentit järjestää aina oikeaan järjestykseen. (Sellers, Wright ja Haemel 2014, s. 558)

OVER-operaattoria käytettäessä primitiivit tulee lajitella syvyysjärjestyksessä kauimmaisista lähimpään, missä kameran sijainti kuvastaa lähintä sijaintia. Käytännössä tämä järjestely tulee tehdä aina uudestaan ennen jokaista piirtokutsua, sillä reaaliaikaisessa sovelluksessa kamera sekä piirrettävät objektit vaihtavat paikkaa jatkuvasti. Jos kuvitellaan tilanne, missä renderöitäviä läpinäkyviä pelihahmoja on yhtäaikaisesti useita näkymässä, niin tällöin jokaisen hahmon jokainen primitiivi tulisi lajitella erikseen. Lajiteltavien primitiivien määrä saattaa nopeasti kasvaa kymmeniin tai satoihin tuhansiin riippuen hahmojen geometrian kompleksisuudesta, joten tälläinen lajittelu ei ole suorituskyvyn näkökulmasta skaalautuva ratkaisu geometrian kasvaessa.

Jotta primitiivejä voitaisiin ensinnäkin vertailla syvyysjärjestyksessä, täytyy jokaiselle primitiiville laskea arvo, jonka mukaan syvyysjärjestely tehdään. Olettaen että jokainen primitiivin kärkipiste on kolmiulotteinen piste, joka muodostuu x, y ja z -komponeista, joista z-komponentti kuvastaa syvyyttä, niin yksinkertainen tapa saada vertailtava arvo syvyysjärjestelyä varten on laskea primitiivin kärkipisteiden z-koordinaattien keskiarvo ja käyttää sitä.

Kuitenkin vaikka jokainen läpinäkyvä primitiivi lajiteltaisiinkin oikein ja blendattaisiin järjestyksessä OVER-operaattoria käyttäen, niin se ei siltikään takaisi virheetöntä tulosta. On nimittäin mahdollista, että eri primitiivien pinnat leikkaavat toisensa tai kärkipisteet asettuvat muuten syvyyden suhteen ongelmallisesti (1) fragmenttitasolla, jolloin primitiivin kärkipisteiden keskiarvon avulla laskettu syvyys ei riitä enää tuottamaan oikeaa tulosta.



Kuvio 1. Primitiivikohtaisen syvyyden laskemisen kannalta ongelmallisesti asettuneet kolmion kärkipisteet. Kuvan mukainen tilanne on mahdotonta renderöidä, jos jokaiselle primitiivillä on yksi keskiarvollinen syvyys, johon nojaten päätetään missä järjestyksessä yhden primitiivin kaikki fragmentit piirretään.

4 Keskeiset läpinäkyvyyden renderöintimenetelmät

Tässä kappaleessa tehdään katsaus muutamaan läpinäkyvyyden renderöimiseen keskittyvään menetelmään.

4.1 Sort-Independent Alpha Blending

Läpinäkyvyyden renderöimisessä *Order-Independent Transparency (OIT)* -termillä viitataan menetelmiin, joilla pyritään ohittamaan vaatimus piirrettävien läpinäkyvien primitiivien järjestämisestä syvyyssuuntaisesti. Kuten aiemmin todettu, reaaliaikainen piirrettävien objektien järjestäminen syvyyssuunnissa on ongelmallista suorituskyvyn näkökulmasta, joten *OIT*-menetelmät tarjoavat tehokkaita ratkaisuja tarkkuuden kustannuksella.

McGuire ja Bavoil (2013) mukaan Meshkin (2007) esitteli Order-Independent Transparency -menetelmästä ensimmäisen version, jota kutsutaan nimellä *Sort-Independent Alpha Blending*. Meshkin (2007) tekee katsauksen *OVER*-operaattoriin laventamalla sen kaikki osat yksinkertaiseen muotoon plus-, miinus- ja kertolaskuja. Tästä tuloksesta Meshkin (2007) erottelee järjestysriippuvaiset sekä -riippumattomat osat. Poistamalla kaikki ei-kommutatiiviset osat kokonaan on mahdollista rakentaa uusi blending-operaattori, joka toimii mutta tuottaa virheellisiä tuloksia tietyissä tilanteissa.

Listing 4.1. "Sort Independent Alpha Blending -implementaation pseudokoodi."

```
1   // 1. vaihe:
2       // Aktivoi näytönohjaimen blendaus sekä
3       // lisäävä blendaus (additive blending).
4       glEnable(GL_BLEND)
5       glBlendFunc(GL_ONE, GL_ONE)
6       drawTransparentSurfaces()
7
8       // [fragment shader]:
9       // Renderöi kuvapuskuritekstuureihin t1 ja t2
10      // fragment shaderilla kaikki läpinäkyvät primitiivit.
11      t1.rgb = src.rgb * src.a
```

```
12          t1.a = src.rgb
13          t2.rgb = 1.0 / src.a
14          t2.a = src.a
15      // 2. vaihe:
16          // Aktivoi kertova blendaus (multiplicative blending).
17          glBlendFunc(GL_ZERO, GL_SRC_COLOR)
18          drawTransparentSurfaces()
19
20          // [fragment shader]:
21          // Renderöi kuvapuskuritekstuuriin t3
22          // fragment shaderilla kaikki läpinäkyvät tekstuurit
23          // uudestaan.
24          t3.rgb = src.rgb
25          t3.a = src.a
26      // 3. vaihe:
27          // Deaktivoi blendaus ja piirrä
28          // yhdistetty tulos ruudulle.
29          glDisable(GL_BLEND)
30          drawScreenQuad()
31
32          // [fragment shader]:
33          // Yhdistä kuvapuskuritekstuurit (t1, t2, t3)
34          // ja tausta (bg) fragment shaderillä lopulliseksi
35          // tulokseksi.
36          out = t1.rgb - bg.rgb * t1.a +
37                bg.rgb * t3.a +
38                bg.rgb * t2.rgb * t3.a
```

Meshkin (2007) esittää toteutuksen *Sort-Independent Alpha Blending* -menetelmälle, mikä koostuu kolmesta erillisestä vaiheesta. Renderöinti voidaan seuraavan pseudokoodin (4.1) mukaisesti, missä *src* on piirrettävä läpinäkyvä pikseli ja *t1*, *t2* ja *t3* ovat vastaavasti ensimmäinen, toinen ja kolmas kuvapuskuritekstuuri. Lisäksi menetelmä olettaa, että tausta (*bg*) on renderöity omaan kuvapuskuritekstuuriin ennen ensimmäistä vaihetta.

Ensimmäisessä vaiheessa kaikki läpinäkyvät primitiivit piirretään kahteen erilliseen kuvapuskuritekstuuriin samanaikaisesti. Lisäksi ennen renderöimistä näytönohjaimen blendaus on aktivoitava ja blending-operaattoriksi on määritettävä lisäävä blendaus (engl. *additive blending*). Lisäävä blendaus tarkoittaa sitä, että kun kuvapuskuritekstuuriin sijoitetaan arvo, niin sijoitettavaan arvoon lisätään kuvapuskuritekstuurissa ennestään oleva arvo.

Toisessa vaiheessa läpinäkyvät primitiivit piirretään uudestaan kuvapuskuritekstuuriin *t3*. Kuten ensimmäisessä vaiheessa, läpinäkyvien primitiivien fragmentit tallennetaan kuvapuskuritekstuuriin kerryttäen, mutta tällä kertaa sijoituksen yhteydessä käytetään kertolaskua, eli kertovaa blendausta (engl. multiplicative blending). Meshkin (2007) Mukaan toinen vaihe voidaan jättää kokonaan pois, mutta tällöin lopputuloksen tarkkuus kärsii hieman.

Kun ensimmäinen ja toinen vaihe ovat suoritettu, seuraa 3. vaihe, jossa tulokset yhdistetään yhdeksi kokonaisuudeksi. Tässä kohtaa oletetaan, että kaikki ei-läpinäkyvät primitiivit ovat renderöity yhteen kuvapuskuritekstuuriin, joka toimii läpinäkyvien primitiivien taustana (*bg*). Jokaisesta kuvapuskuritekstuurista voidaan noutaa tarvittavat värikomponentit, jotka tarvitaan lopullisen värin muodostamisessa. Tässä vaiheessa ei enää tarvitse käyttää näytönohjaimen blendausta, joten se asetetaan pois päältä. Meshkin (2007) suosittelee käytettävän kuvapuskuritekstuureissa 64-bittistä RGBA-formaattia, missä jokaisen yksittäisen värikomponentin tarkkuus on tällöin 16 bittiä ($16 \cdot 4 = 64$).

Sort-Independent Transparency sopii Meshkin (2007) mukaan parhaiten käytettäväksi useiden partikkeleiden kanssa, sillä partikkeleita on usein liian monta järjestettäväksi, ja partikkeleiden tarkkuus ei ole yleensä tärkeää niiden suhteellisen pienen kokonsa takia. Menetelmä on alttiimpi virheille, kun läpinäkyvien objektien keskenäinen *alpha*-arvo vaihtelee paljon, ja lisäksi pienen *alpha*-arvon omaavat objektit toimivat paremmin kuin vain hieman läpinäkyvät objektit. (Meshkin 2007)

4.2 Weighted Average

Bavoil ja Myers (2008) esittävät parannellun version *Sort-Independent Transparency* -menetelmästä, jota he kutsuvat nimellä *Weighted Average*. Bavoil ja Myers (2008) tekevät havainnon, että jos monta päällekkäistä läpinäkyvää väriä ovat täysin samat, niin tällöin värien järjestyksellä

ei ole lopputuloksen kanssa merkitystä, kun värit blendataan yhteen. Jos taas päällekkäiset värit eroavat toisistaan, niin lopullinen yhdistetty väri voidaan approksimoida laskemalla päällekkäisten värien keskiarvo.

Listing 4.2. "Weighted Average -implementaation pseudokoodi."

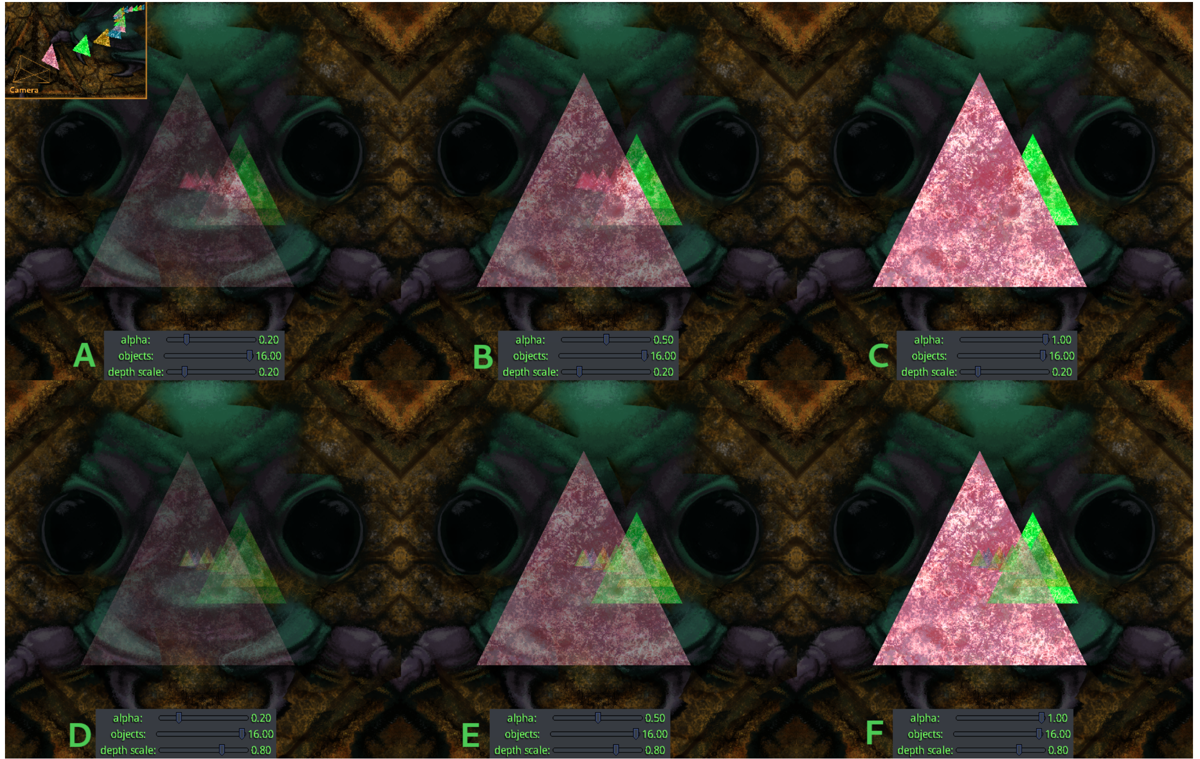
```
1
2    // 1. vaihe
3        // Piirretään läpinäkymättömät primitiivit näyttölaitteen
4        // kuvapuskuriin.
5        drawBackground()
6
7    // 2. vaihe:
8        // Piirretään läpinäkyvät primitiivit
9        // samanaikaisesti kahteen kuvapuskuritekstuuriin.
10       // Ensimmäiseen (t0) piirtyvät värit ja
11       // toiseen (t1) päällekkäisten läpinäkyvien
12       // primitiivien lukumäärä.
13
14       glDisable(GL_DEPTH_TEST)
15       glEnable(GL_BLEND)
16       glBlendFunc(GL_ONE, GL_ONE)
17       drawTransparentSurfaces()
18
19       // [fragment shader]:
20       t0.rgba = src.rgba;
21       t1.rgba = vec4(1);
22
23   // 3. vaihe:
24       // Piirretään koko näkymä ruudulle.
25       glEnable(GL_DEPTH_TEST)
26       glBlendFunc(GL_ONE_MINUS_SRC_ALPHA, GL_SRC_ALPHA)
27       drawScreenQuad()
28
29       // [fragment shader]:
```

```
30        // accum = kerrytetty kaikkien läpinäkyvien
31        // fragmenttien väri
32        vec4 accum = texelFetch(t0, ivec2(gl_FragCoord.xy), 0)
33        // n = päällekkäisten läpinäkyvien fragmenttien lukumäärä
34        float n = max(1.0, texelFetch(t1,
35                 ivec2(gl_FragCoord.xy), 0).r
36        float ao = pow(max(0.0, 1.0 - (accum.a / n)), n))
37        gl_FragColor = vec4(accum.rgb / max(accum.a, 0.0001)), ao)
```

McGuire ja Bavoil (2013) esittävät implementaation *Weighted Average* -metodille. Listaus (4.2) havainnollistaa menetelmän perus ideaa. Implementaatio koostuu kolmesta vaiheessa. Ensimmäisessä vaiheessa piirretään läpinäkymättömät primitiivit oletuskuvapuskuriin. Toisessa vaiheessa läpinäkyvät päällekkäiset primitiivit piirretään omaan kuvapuskuritekstuuriin. Lisäksi samalla kertaa toiseen kuvapuskuritekstuuriin lasketaan läpinäkyvien päällekkäisten primitiivien lukumäärä (n) jokaista kuvapuskurin fragmenttia kohden. Lopuksi kolmannessa vaiheessa läpinäkyvät värit kerrotaan painotetulla keskiarvolla ja yhdistetään oletuskuvapuskurin kanssa, mistä saadaan tulokseksi lopullinen näkymä.

4.3 Blended Weighted Order-Independent Transparency

McGuire ja Bavoil (2013) kehittämä *Blended Weighted Order-Independent Transparency (BWOIT)* on jatkumoa *Weighted Average* -menetelmälle. *BWOIT*-menetelmä parantelee *Weighted Average* -menetelmää huomioimalla myös tapaukset, joissa fragmentin värit tai *alpha*-arvo ovat arvoltaan *0.0*. Esimerkiksi *Weighted Average* -menetelmässä tälläisien täysin näkymättömien primitiivien lisääminen muiden primitiivien sekaan aiheuttaa vääristymää lopullisen väriarvon muodostamisessa, koska väriarvon laskeminen pohjautuu tavanomaiseen keskiarvoon. *BWOIT*-menetelmässä nämä täysin näkymättömät primitiivit eivät vaikuta lopulliseen väriarvoon. McGuire ja Bavoil (2013) esittämä kaava (4.1) havainnollistaa, miten *BWOIT*-menetelmä laskee näkymän taustan ja läpinäkyvien fragmenttien yhdistetyn lopullisen väriarvon C_f.



Kuvio 2. *Blended Weighted Order-Independent Tranparency* -implementaatio. 16 syvyyssuunnassa päällekkäistä läpinäkyvää kolmiota. Eri vaiheet kuvastavat miten menetelmä on riippuvainen oikeanlaisen syvyysfunktion (*depth scale*) säätämisestä eri tilanteita varten. Kuvissa **C** ja **F** kaikkien primitiivien läpinäkyvyys (*alpha*) on *1.0*, mutta tästä huolimatta kuvan **F** peitetyt kolmiot näkyvät päällimmäisen kolmion läpi, sillä syvyysfunktion parametrit ovat säädetty "väärin".

$$C_f = \frac{\sum_{i=1}^{n} C_i}{\sum_{i=1}^{n} \alpha_i}\left(1 - \prod_{i=1}^{n}(1-\alpha_i)\right) + C_0 \prod_{i=1}^{n}(1-\alpha_i) \tag{4.1}$$

Kun kaava (4.1) jaetaan osiin, niin $\frac{\sum_{i=1}^{n} C_i}{\sum_{i=1}^{n} \alpha_i}$ tarkoittaa painotettua keskiarvoa, joka saadaan laskemalla kaikkien läpinäkyvien päällekkäisten fragmenttien värit yhteen ja jakamalla värit värien *alpha*-arvojen summalla.

$\left(1 - \prod_{i=1}^{n}(1-\alpha_i)\right)$ laskee arvon, joka simuloi syvyyssuunnassa päällekkäisten läpinäkyvien primitiivien kokonaisabsorbiota. Asian voi ajatella myös niin, että kun valonsäde läpäisee usean osittain läpinäkyvän primitiivin, niin valonsäteen intensiteetti pienenee jokaisen läpäi-

syn yhteydessä. Tämä jäljelle jäänyt intensiteetti määrää sitten suhteen, jolla läpinäkyvien fragmenttien keskiarvollista väriä ja taustan väriä sekoitetaan lopullisen väriarvon C_f muodostamiseksi. Samaa ideaa käytetään myös *Stochastic Transparency* -menetelmässä, ja sen toteutuksen yksityiskohdista keskustellaan enemmän myöhemmin (4.6). Kaavan viimeinen osa $C_0 \prod_{i=1}^{n}(1-\alpha_i)$ kuvastaa taustavärin kontribuutiota.

Jos asetamme jokaiselle kaavan (4.1) osalle uuden tunnisteen (4.2) mukaisesti

$$\begin{aligned} C_{src} &= \frac{\sum_{i=1}^{n} C_i}{\sum_{i=1}^{n} \alpha_i} \\ A_{src} &= \prod_{i=1}^{n}(1-\alpha_i) \\ C_{dst} &= C_0 \end{aligned} \tag{4.2}$$

ja kokoamme sen uudestaan (4.3) tunnisteita hyödyntäen,

$$C_f = C_{src} \cdot \Big(1 - A_{src}\Big) + C_{dst} \cdot A_{src} \tag{4.3}$$

niin huomaamme, että nyt kaava (4.3) muistuttaa huomattavan paljon *OVER*-operaattoria (2.5), sillä vain A_{src} ja $(1-A_{src})$ ovat vaihtaneet paikkoja verrattuna *OVER*-operaattoriin. Kaava (4.1) toimii siis pohjimmiltaan täysin samalla periaatteella kuin *OVER*-operaattori.

Kaava (4.1) tuo parannusta *Weighted Average* -menetelmälle, mutta McGuire ja Bavoil (2013) lisäävät kaavaan vielä yhden osan, $w(z_i, \alpha_i)$, jonka myötä myös fragmenttien z-suuntaisella syvyydellä saadaan vaikutus siihen, kuinka paljon läpinäkyvien fragmenttien väriarvot kontribuoivat lopulliseen väriarvoon C_f.

$$C_f = \frac{\sum_{i=1}^{n} C_i \cdot w(z_i, \alpha_i)}{\sum_{i=1}^{n} \alpha_i \cdot w(z_i, \alpha_i)} \Big(1 - \prod_{i=1}^{n}(1-\alpha_i)\Big) + C_0 \prod_{i=1}^{n}(1-\alpha_i) \tag{4.4}$$

Syvyysfunktion $w(z_i, \alpha_i)$ voi määritellä itse haluamallaan tavalla, kunhan se palauttaa arvoja väliltä $0.01 < x < \infty$ ja on monotonisesti laskeva. Funktiolle annetaan parametrina käsiteltävän läpinäkyvän fragmentin syvyys z, jolloin se palauttaa syvyyttä vastaavan painoarvon.

Kun lähempi syvyys palauttaa suuremman arvon kuin kauempi syvyys, kontribuoivat kameraa lähempänä olevat fragmentit enemmän lopulliseen väriarvoon C_f.

NVIDIA GameWorks (2020) julkaisema esimerkki-implementaatio käyttää (4.5)

$$w(z) = \frac{0.03}{z^{4.0}} \tag{4.5}$$

mukaista syvyysfunktiota, jonka paluuarvo pakotetaan välille $10^{-2} < w(z) < 3 \cdot 10^{3}$ fragment shaderin *clamp*-funktiota hyödyntäen. McGuire ja Bavoil (2013) listaavat erilaisia variaatioita syvyysfunktiolle, joista jokainen variaatio soveltuu parhaiten eri tilantesiin missä kaikki läpinäkyvät objektit ovat eri etäisyyksillä suhteessa kameraan. Kuvio (2) havainnollistaa millainen vaikutus syvyysfunktiolla on lopullisen blendatun väriarvon muodostamisessa.

NVIDIA GameWorks (2020) on julkaissut vapaasti ladattavan *WBOIT*-implementaation lähdekoodin, ja siksi tässä kappaleessa ei käydä menetelmän kooditoteutusta lävitse sen enempää.

4.4 Screen-Door Transparency

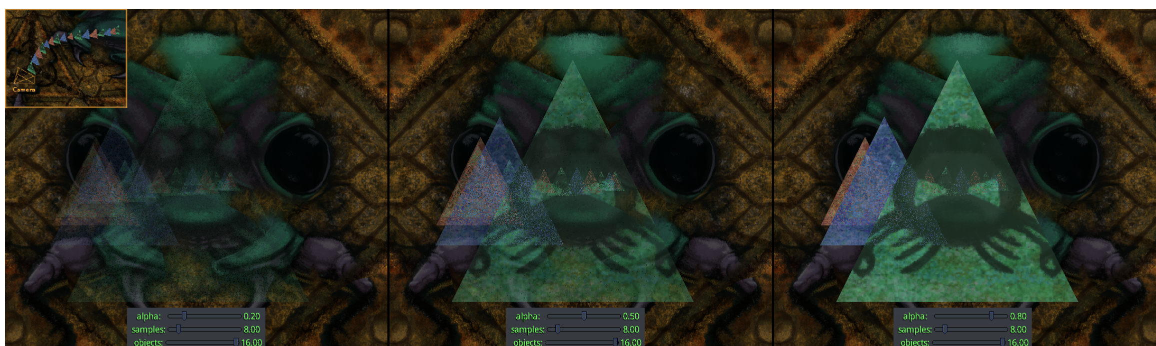
Screen-Door Transparency on yksinkertainen menetelmä, missä läpinäkyvyys toteutetaan ruutukuviota hyödyntämällä. Läpinäkyvän primitiivin joka toinen fragmentti hylätään täysin, jolloin primitiiville muodostuu shakkilautaa muistuttava ruutukuvio läpinäkyviä reikiä. Kun tälläinen reikäinen primitiivi renderöidään jonkin taustan päälle, saadaan primitiivi näyttämään osittain läpinäkyvältä.

Akenine-Moller, Haines ja Hoffman (2008, s. 135) kuvaavat menetelmän heikkoudeksi sen, että sillä voidaan mallintaa uskottavasti vain primitiivejä, joiden *alpha*-arvon oletetaan olevan tasan puolet. Jos halutaan eriasteista läpinäkyvyyttää, tulee läpinäkyvien reikien määrää suurentaa tai pienentää. Tämä tuottaa nopeasti epäuskottavia tuloksia. Lisäksi useamman kuin yhden läpinäkyvän primitiivin renderöiminen päällekkäin *Screen-Door Transparency* -menetelmällä tuottaa virheellisiä tuloksia tai ei ole lainkaan mahdollista. Esimerkiksi jos kaksi täsmälleen samankokoista punaista ja sinistä primitiiviä renderöidään päällekkäin

Screen-Door Transparency -menetelmällä käyttäen samaa läpinäkyvyysastetta, niin vain toinen toinen primitiiveistä jää näkyviin, sillä jompikumpi primitiiveistä ylikirjoittaa toisen fragmentit täysin, jos primitiivien läpinäkymättömät fragmentit osuvat täsmälleen päällekkäin.

Menetelmän vahvuuksina Akenine-Moller, Haines ja Hoffman (2008, s. 135) pitävät sen yksinkertaisuutta. *Screen-Door Transparency* ei vaadi blending-operaattoreiden käyttämistä tai laitteistoa, joka tukee niiden nopeaa laskentaa. Lisäksi menetelmän tuottamat tulokset eivät ole riippuvaisia osittain läpinäkyvien primitiivien syvyysjärjestyksestä, joten ne voidaan piirtää missä järjestyksessä tahansa yhtäaikaa tavallisten täysin läpinäkymättömien primitiivien kanssa.

4.5 Stochastic Transparency



Kuvio 3. Stochastic Transparency -implementaatio ja sen eri asteisen läpinäkyvyyden vertailu. Jokaisessa kuvassa on 16 syvyyssuunnassa päällekkäin olevaa teksturoitua kolmiota, joilla kaikkilla on kussakin tilanteessa sama läpinäkyvyys (*alpha*). Läpinäkyvyys prosentuaalisesti eri kuvissa vasemmalta oikealle: 20%, 50%, 80%.

Toinen tapa hyödyntää *Screen-Door Transparency* -menetelmän ideaa on todennäköisyyksien avulla. Voidaan päättää, että jokaisen fragmentin *alpha*-arvo kuvastaa todennäköisyysprosenttia fragmentin hylkäämiselle. Esimerkiksi jos fragmentin *alpha*-arvo on *0.5*, niin tällöin fragmentti piirretään *50* prosentin todennäköisyydellä, kun taas jos *alpha*-arvo on *1.0* niin fragmentti piirretään joka kerta. Reaaliaikaisissa sovelluksissa tälläinen renderöiminen tuottaa kuitenkin huomattavaa kohinaa fragmenttitasolla, koska läpinäkyvyyttä sisältäville

fragmenteille lasketaan uusi ja potentiaalisesti eri arvo jokaisella ruudunpäivityksellä.

Listing 4.3. "Stochastic Transparency -implementaatio: Alustus ja piirtosilmukan vaiheet."

```
1  // Alustus:
2  background = framebuffer(depth = true,
3                                   internalFormat = GL_RGB,
4                                   format = GL_RGB)
5  totalAlpha = framebuffer(depth = false,
6                                   internalFormat = GL_R16F,
7                                   format = GL_RED)
8  transparentObjects = framebuffer(depth = true,
9                               internalFormat = GL_RGBA16F,
10                              format = GL_RGBA)
11
12 // 1. vaihe:
13     // Piirrä läpinäkymätön geometria omaan
14     // kuvapuskuritekstuuriin.
15     background.renderOnThis {
16         renderOpaque()
17     }
18 // 2. vaihe:
19     // Laske läpinäkyvien primitiivien "alpha correction" -arvo
20     // erilliseen kuvapuskuritekstuuriin.
21     totalAlpha.renderOnThis {
22         glDisable(GL_CULL_FACE)
23         glEnable(GL_DEPTH_TEST)
24         glEnable(GL_BLEND)
25
26         glClearColor(1.0, 0.0, 0.0, 0.0)
27         glClear(GL_COLOR_BUFFER_BIT)
28         glBlendFunc(GL_ZERO, GL_ONE_MINUS_SRC_COLOR)
29
30         shader("totalAlpha") {
31             renderTransparentObjects()
32         }
33     }
34 // 3. vaihe:
35     // Piirrä läpinäkyvät primitiivit omaan kuvapuskuritekstuuriinsa
```

```
36      // niin monta kertaa kuin näytteitä (totalSamples) halutaan ottaa.
37      transparentObjects.renderOnThis {
38          glClearColor(0f, 0f, 0f, 0f)
39          glClear(GL_COLOR_BUFFER_BIT)
40          glDisable(GL_CULL_FACE)
41          glEnable(GL_DEPTH_TEST)
42          glDisable(GL_BLEND)
43
44          shader("stochasticTransparency") {
45              uniform("totalSamples", totalSamples)
46              for (i in 0 until totalSamples) {
47                  glClear(GL11.GL_DEPTH_BUFFER_BIT)
48                  uniform("currentSampleNumber", i)
49                  renderObjects()
50              }
51          }
52      }
53  // 4. vaihe: Yhdistetään lopullinen näkymä.
54      glDisable(GL_BLEND)
55      glEnable(GL_CULL_FACE)
56      glEnable(GL_DEPTH_TEST)
57      glCullFace(GL_BACK)
58      shader("combine") {
59          bindTexture("texture0", 0, background)
60          bindTexture("texture1", 1, totalAlpha)
61          bindTexture("texture2", 2, transparentObjects)
62          engine.screenQuad.render()
63      }
```

Enderton ym. (2010) kutsuvat tälläistä edellä kuvattua menetelmää nimellä *Stochastic Transparency*, josta he esittävät useita erilaisia implementaatiovariaatioita. Enderton ym. (2010) ehdottamassa yksinkertaisimmassa implementaatiossa piirtosilmukka on jaettu neljään erilliseen vaiheeseen, jotka näkyvät listauksessa (4.3).

Toteutuksen ensimmäisessä vaiheessa kaikki läpinäkymättömät primitiivit piirretään erilliseen kuvapuskuritekstuuriin.

Toisessa vaihessa kaikista läpinäkyvistä primitiiveistä lasketaan yhteinen *alpha*-arvo (α_{total}) omaan kuvapuskuritekstuuriin. Tämä arvo lasketaan vähentämällä jokaisen syvyyssuunnassa päällekkäisen läpinäkyvän fragmentin *alpha*-arvo arvosta 1.0, jonka jälkeen jokainen näistä erotuksista kerrotaan yhteen yhdeksi arvoksi. Enderton ym. (2010) esittävät vastaavan asian matemaattisen kaavan (4.6)

$$\alpha_{total} = \prod_{i=1}^{n} (1.0 - \alpha_i) \tag{4.6}$$

mukaisesti, missä α on fragmentin *alpha*-arvo ja n on päällekkäisten läpinäkyvien fragmenttien määrä. α_{total} voidaan laskea erilliseen kuvapuskuritekstuuriin hyödyntämällä näytönohjaimen blending-operaattoreita. Koska α_{total} tallentamiseen vaaditaan vain yksi liukuluku, kannattaa kuvapuskuritekstuuri alustaa sopivaa formaattia käyttäen. OpenGL-rajapintaa käyttäessä yhden liukuluvun tallennusta varten listauksen (4.4)

Listing 4.4. "Stochastic Transparency: α_{total}-arvoa varten käytettävä kuvapuskuriformaatti."

```
1 glTexImage2D(GL_TEXTURE_2D, 0, GL_R16F, width, height, 0, GL_RED,
      GL_FLOAT, 0);
```

mukainen formaatti lienee optimaalisin vaihtoehto α_{total}-arvon tallentamista varten. Kuten listauksen (4.3) toinen vaihe näyttää, kuvapuskuritekstuurin tulee ensiksi alustaa arvolla *1.0*, ja blending-operaattori tulee asettaa (4.5) mukaisesti.

Listing 4.5. "Stochastic Transparency: α_{total}-arvon laskemisessa käytettävä blending-operaattori."

```
1 glBlendFunc(GL_ZERO, GL_ONE_MINUS_SRC_COLOR)
```

Listing 4.6. "Stochastic Transparency -implementaatio: "Alpha-arvojen yhteenlaskeminen fragment shaderilla."

```
1 // "totalAlpha" fragment shader
2 #version 330
3 in vec2 vTexture;
4 out vec4 fragColor;
5 // uniform sampler2D texture0;
```

```
6   uniform float objectTransparency;
7   void main() {
8       // fragColor.r = texture(texture0, vTexture).w;
9       fragColor = vec4(objectTransparency, 0.0, 0.0, 0.0);
10  }
```

Tästä seuraa se, että α_{total}-arvot tallentuvat kuvapuskuritekstuuriin kaavan (4.6) mukaisesti, kun fragment shaderiltä ulos tulevan *vec4*-muuttujan *r*- tai *x*-komponenttiin sijoitetaan vain fragmentin läpinäkyvyys listauksen (4.6) mukaisesti.

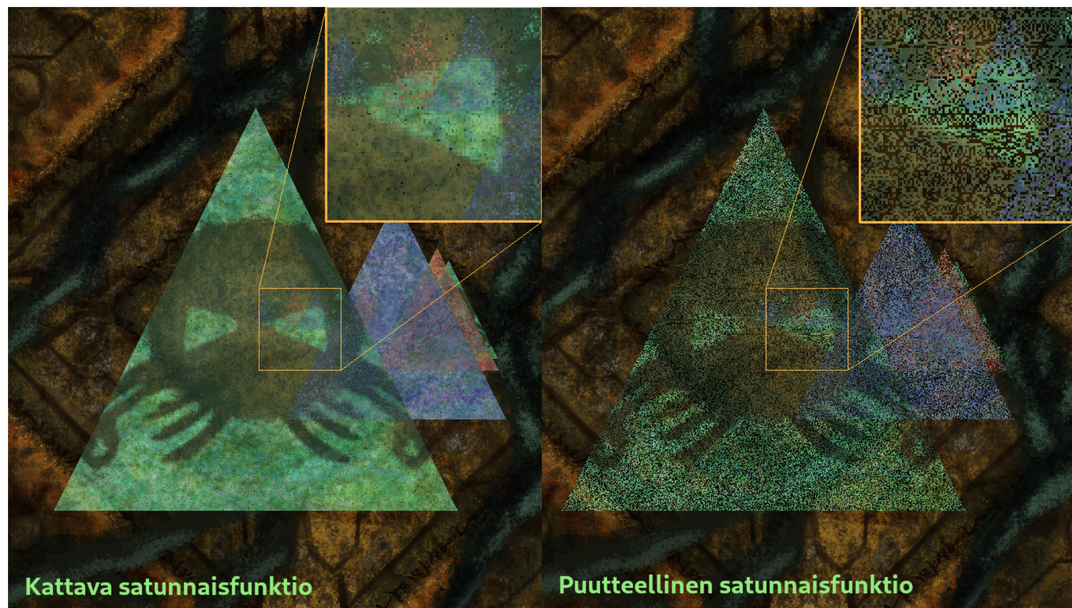
Kolmannessa vaiheessa renderöidään kaikki näkymän läpinäkyvät primitiivit omaan kuvapuskuritekstuuriin, jossa on myös syvyyspuskuri käytössä. Listauksen (4.3) kolmannen vaiheen mukaisesti, kuvapuskuritekstuurin kaikki pikselit alustetaan arvolla (0.0, 0.0, 0.0, 0.0), jonka jälkeen kaikki läpinäkyvät primitiivit renderöidään kuvapuskuriin niin monta kertaa kuin läpinäkyvyyttä halutaan näytteistää (*totalSamples*). Jokaisen piirtokerran välissä syvyyspuskurin arvot tyhjennetään.

Listing 4.7. "Stochastic Transparency -implementaatio: "Läpinäkyvien primitiivien renderöiminen fragment shaderilla."

```
1   // "stochasticTransparency" fragment shader
2   #version 330
3   in vec2 vTexture;
4   // Satunnaisluvun generoimiseen käytetään
5   // koordinaatteja, joille ei ole tehty
6   // perspektiivikorjausta.
7   noperspective in vec4 noPersCoord;
8   out vec4 fragColor;
9
10  uniform sampler2D texture0;
11  uniform float objectTransparency;
12  uniform float totalSamples;
13  uniform float currentSampleNumber;
14
15  // näennäislukugeneraattori
16  float rand(vec2 co){
17      return fract(cos(dot(co.xy,
```

```
18                           vec2(63.8502,24.843))) * 85398.3795);
19  }
20
21  void main() {
22      float sampleI = currentSampleNumber;
23      vec3 randomCoords = vec3(noPersCoord.x + sampleI * 0.01,
24                               noPersCoord.y + sampleI * 0.4,
25                               noPersCoord.z / noPersCoord.w);
26      float randomValue = abs(rand(
27                              vec2(randomCoords.x * randomCoords.y,
28                              randomCoords.z * randomCoords.y)
29                              ));
30
31      if (randomValue > objectTransparency) {
32          discard;
33      }
34
35      vec4 textureColor = texture(texture0,
36                                  vTexture * textureScale);
37      fragColor = vec4(textureColor.x,
38                       textureColor.y,
39                       textureColor.z,
40                       1.0);
41  }
```

Listaus (4.7) näyttää piirrettävien läpinäkyvien primitiivien fragment shaderilla suoritettavan koodin. Koska *Stochastic Transparency* -menetelmän idea perustuu siihen, että renderöitävän läpinäkyvän fragmentin *alpha*-arvo määrittää todennäköisyyden sille, jätetäänkö fragmentti piirtämättä vai ei, niin jokaista läpinäkyvää fragmenttia varten täytyy generoida satunnaisluku. Bonzai Software (2013) ehdottaa satunnaislukujen generoimiseen esilasketun 3D-kohinatekstuurin luomista, josta voidaan noutaa satunnainen arvo väliltä *0.0-1.0* fragmentin x, y ja z -koordinaattien perusteella. 3D-kohinatekstuuri on todennäköisesti suorituskyvyn näkökulmasta yksi tehokkaimmista ratkaisuista, mutta kelvollisen satunnaisluvun voi myös generoida suoraan fragment shaderillä näennäislukugeneraattorifunktion avulla, jolle syötetään sekoitus fragmentin x, y ja z -koordinaateista sekä numero, joka kuvastaa kuin-



Kuvio 4. Puuttellinen ja huonosti toteutettu Stochastic Transparency -menetelmän satunnaisfunktio saattaa aiheuttaa huomattavaa kohinaa ja epätasaisuutta.

ka monetta näytteenottoiteraatiota ollaan piirtämässä (*currentSampleNumber*). Puuttellisesti toimiva satunnaislukugeneraattori (4) vaikuttaa menetelmän lopputulokseen negatiivisesti, joten sen implementoimiseen kannattaa kiinnittää huomiota.

Listing 4.8. "Stochastic Transparency: todennäköisyyden testaaminen fragmentin *alpha*-arvon perusteella."

```
1  if (randomValue > objectTransparency) {
2      discard;
3  }
```

Kun satunnaisluku on generoitu, voidaan tehdä yksinkertainen testi (4.8), joka päättää hyväksytäänkö näytteistettävä läpinäkyvä fragmentti vai ei.

Täytyy kuitenkin muistaa, että vaikka fragmentti hyväksyttäisiinkin, niin se saatetaan hylätä, jos joku muu syvyyssuunnassa lähempi läpinäkyvä fragmentti hyväksytään sen jälkeen. Syvyyspuskuri pitää huolen siitä, että näytteistetyistä fragmenteista kaikista lähin fragmentti hyväksytään kuvapuskuritekstuuriin.

Listing 4.9. "Stochastic Transparency: Lopullisen tuloksen yhdistäminen fragment shaderil-

la."

```
#version 330
in vec2 vTexture;

uniform sampler2D texture0;
uniform sampler2D texture1;
uniform sampler2D texture2;
out vec4 color;

void main() {
    vec4 t0 = texture(texture0, vTexture); // background
    vec4 t1 = texture(texture1, vTexture); // transparent objects
    vec4 t2 = texture(texture2, vTexture); // total alpha
    float alphaCorrection = t2.x;
    vec3 tmix = t0.rgb * alphaCorrection + t1.rgb * (1.0 -
        alphaCorrection);
    color = vec4(tmix, 1.0);
}
```

Neljännessä vaiheessa kaikki tarvittava tieto näkymän muodostamiseen on tallennettu kuvapuskuritekstuureihin, joten lopullinen näkymä voidaan nyt renderöidä yhdelle koko näyttölaitteen ruudun täyttävälle nelikulmiolle lukemalla kuvapuskuritekstuureista tarvittavat tiedot fragment shaderilla. Listaus (4.9) näyttää miten lopullinen ruudulle piirrettävän fragmentin väriarvot tulee yhdistää. Taustan värit (*t0.rgb*) kerrotaan lasketulla kokonaisalpha-arvolla (*alphaCorrection*) ja läpinäkyvien primitiivien värit (*t1.rgb*) kerrotaan arvolla $(1.0 - alphaCorrection)$.

4.6 Depth-peeling

Depth-peeling on läpinäkyvyyden renderöimisessä käytettävä menetelmä, missä läpinäkyvän objektin geometrian pinnat piirretään järjestyksessä syvyyssuunnassa lähimmästä kauimmaisimpaan. Menetelmän nimi tulee siitä, että jokaisen piirretyn pinnan jälkeen piirrettyn

pinnan fragmentit “kuoritaan pois“, jolloin objektin alla oleva seuraava pinta paljastuu seuraavaa piirtoiteraatiota varten. Fragmenttien kuoriminen tarkoittaa algoritmin toiminnan näkökulmasta sitä, että edellisten piirrettyjen pintojen fragmenttien syvyysarvoa käytetään suodattimena, jonka avulla voidaan suodattaa jo piirretyt, kameraa lähempänä olevat pinnat pois.

Jokaisen uuden syvemmän tason fragmentit voidaan blendata suoraan aikaisemman pinnan fragmenttien kanssa, koska blendattavat fragmentit ovat valmiiksi syvyysjärjestyksessä. Objektin syvyyssuunnassa olevien päällekkäisten tasojen määrä määrää, kuinka monta kertaa objekti täytyy piirtää uudestaan. Tämä on suorituskyvyn näkökulmasta raskasta, mutta siihen voidaan vaikuttaa lopettamalla syvyystasojen läpikäynti tietyn lukumäärän jälkeen. Tällöin objektin kaikki läpinäkyvät pinnat eivät tietenkään tule sisällytetyksi lopulliseen tulokseen mukaan, mutta jo muutaman pintakerroksen läpikäynti saattaa tietyissä tapauksissa tuottaa visuaalisesti riittävän lopputuloksen.

Bavoil ja Myers (2008) esittävät parannuksen *Depth-peeling*-menetelmään nimeltään *Dual Depth-peeling*, missä objektin fragmenttien kuoriminen suoritetaan yhtä aikaa kahdesta suunnasta, lähimmästä pinnasta syvimpään ja kauimmaisimmasta pinnasta lähimpään. Algoritmi pysähtyy niiden fragmenttien kohdalla, joissa havaitaan että molemmista suunnista lähestyvät kuorinnat saavuttavat saman fragmentin tai menevät toisistaan yli. Molempien suuntien kerryttämät kaksi eri blendattua lopputulosta voidaan myös blendata lopuksi yhteen, koska molemmat lopputulokset ovat sisäisesti blendattu yhteen noudattaen oikeaa syvyysjärjestystä.

Depth-peeling, missä objektin jokainen läpinäkyvä taso käydää läpi, tuottaa aina virheettömän oikean lopputuloksen (engl. *ground truth*). Tästä syystä *Depth-peeling*-menetelmää käytetään havainnollistamaan eroavaisuuksia muiden läpinäkyvyysmenetelmien välillä. NVIDIA GameWorks (2020) on julkaissut *Depth-peeling*-menetelmästä implementaation, jonka lähdekoodi on vapaasti ladattavissa.

5 Yhteenveto ja pohdinta

Primitiiveihin pohjautuvassa tietokonegrafiikassa yksinkertaista läpinäkyvyyttä on mahdollista renderöidä ilman syvällisempää perehtymistä aiheeseen, jos käytössä on helppokäyttöinen kirjasto tai ohjelmointirajapinta. OpenGL-rajapinnassa *OVER*-operaattorin käyttäminen ja *GL_BLEND*-tilan vaihteleminen päälle ja pois saattaa usein olla täysin riittävä keino tuottamaan vaatimukset täyttävää läpinäkyvää grafiikkaa. Vaikka OpenGL hoitaa läpinäkyvien värien sekoittamisen blending-operaattoreiden avulla lähes automaattisesti, on blending-operaattoreiden syvällisemmästä ymmärtämisestä usein apua myös muissakin ympäristöissä.

Jos kuitenkin halutaan tuottaa monimutkaista, visuaalisesti realistisen näköistä läpinäkyvyyttä, on syytä opetella läpinäkyvyyden renderöimisen perusteet huolellisesti. Perusteiden parissa saattaa kulua yllättäviäkin määriä aikaa, mutta niiden hallitseminen on lähes poikkeuksetta edellytettyä, jos pyrkii ymmärtämään ja implementoimaan itse edistyneempiä aiheeseen liittyviä menetelmiä.

Tämä kandidaatintutkielma rakentui täysin sitä mukaan, kun kirjoittaja oppi aiheesta itse lisää ilman mitään taustatietoja läpinäkyvyyden renderöimisestä. Tästä syystä tutkielman alkupuoli pyrki käymään läpinäkyvyyden renderöimisen perusteet kattavasti, jonka jälkeen vasta keskityttiin edistyneempiin menetelmiin ja niiden yksityiskohtiin. Lisäksi tutkielmassa pyrittiin painottamaan niiden kohtien selkeyttämistä ja avaamista, jotka tuntuivat kirjoittajasta haastavilta niitä opeteltaessa.

Tutkielmassa esitetyt menetelmät läpinäkyvyyden renderöimiseksi eivät edusta läpinäkyvyyden renderöimiseen liittyvän aihepiirin kokonaiskuvaa, sillä läpinäkyvyyden renderöimistä varten on kehitelty paljon muitakin menetelmiä. Tässä tutkielmassa esitetyt menetelmät antavat hyvän yleiskuvan niin kutsutusta *Order-Independent Transparency (OIT)* -menetelmistä, joissa menetelmän pääidea pyrkii ratkaisemaan blending-operaattoreiden syvyysjärjestyksen liittyvän rajoitteen. *OIT*-menetelmien hyödyntäminen vaikuttaa olevan suosittu tapa renderöidä läpinäkyvyyttä, kun kyseessä on reaaliaikaiset sovellukset, joissa tehokkuus on tärkeämpää kuin tarkkuus ja oikeellisuus.

Esitetyistä menetelmistä *Depth-peeling*, *Blended Weighted Order-Independent Transparency*

ja *Stochastic Transparency* soveltuvat yleiskäyttöisen reaaliaikaisen läpinäkyvyyden renderöimiseen. On tosin syytä pohtia, mikä näistä kannattaa valita mihinkin tilanteeseen.

Kuten aiemmin todettu, *Depth-peeling*-menetelmät sopivat parhaiten tilanteisiin, missä oikeellisuus ja häiriötön lopputulos ovat tärkeysjärjestyksessä korkealla. *Depth-peeling* pystyy joustamaan tarvittaessa myös suorituskyvyn puolesta, jos kaikki läpinäkyviä tasoja ei käydä lävitse.

Blended Weighted Order-Independent Transparency lienee tehokkain menetelmä suorituskyvyn näkökulmasta, ja se pääsee tuloksissaan McGuire ja Bavoil (2013) mukaan myös todella lähelle syvyysjärjestetyn ja tavanomaisella *OVER*-operaattorilla blendatun läpinäkyvyyden tuloksia, kunhan *BWOIT*:n syvyysfunktion säätää oikein. Lisäksi partikkeli- ja savuefektien renderöiminen on luontevaa *BWOIT*:n avulla. McGuire ja Bavoil (2013) kuvailevat yhtä menetelmän vahvuutta sellaisen tilanteen avulla, missä kameran katselusuunnan kanssa kohtisuorassa olevien tekstuureista muodostettujen savupartikkeleiden syvyyssuuntainen liikehdintä ei aiheuta minkäänlaista äkkinäistä fragmenttien ponnahdusefektiä (engl. *popping*), mikä on ongelma esimerkiksi puhtaasti pelkkää *OVER*-operaattoria käyttäessä. Kun partikkeliefektien tekstuurit ovat kohtisuorassa kameran katselusuunnan kanssa, niin McGuire ja Bavoil (2013) mukaan laskentaa on mahdollista myös optimoida siirtämällä syvyysfunktion laskut vertex shaderille. Menetelmän heikkous on sen approksimointiin perustuva luonne, jonka käyttäytymistä saattaa olla vaikea ennustaa kaikissa erilaisissa tilanteissa. Lisäksi vaadittu syvyysfunktion manuaalinen säätäminen parhaiden tuloksien saavuttamiseksi tuo omat haasteensa menetelmän juostavuuden suhteen.

Stochastic Transparency-menetelmä on mielenkiintoinen vaihtoehto *Depth-peeling* ja *WBOIT* -menetelmille. Tasokkaan ja kohinasta minimoidun tarkkuuden saavuttaminen *Stochastic Transparency* -menetelmän avulla tuskin pärjää *WBOIT*:lle, kun vertaillaan puhdasta suorituskykyä. Toisaalta *Stochastic Transparency* ei ole riippuvainen manuaalisesta syvyysfunktion määrittelystä, ja soveltuu täten joustavammin erilaisiin tilanteisiin. Tässä tutkielmassa esitetty implementaatio *Stochastic Transparency* -menetelmästä on yksinkertainen, mutta Enderton ym. (2010) esittelevät myös useita muita variaatioita implementaatioista, jotka vähentävät kohinaa ja tehostavat suorituskykyä hyödyntäen muun muassa näytönohjaimen moninäytteistämistä (engl. *multi-sampling*) ja hienostuneempia todennäköisyyslaskuja. Nä-

mä ehdotetut implementaatiovariaatiot vaikuttivat melko monimutkaisilta, mutta syvällinen perehtyminen niihin saattaa olla vaivan arvoista, sillä menetelmissä käytettäviä ideoita voidaan hyödyntää esimerkiksi myös läpinäkyvien objektien varjokarttojen (engl. *shadow map*) muodostamisessa.

Edistyneempiä läpinäkyvyyteen liittyviä menetelmiä, joita tässä tutkimuksessa ei käsitelty, ovat muun muassa McGuire ja Mara (2016) *A Phenomenological Scattering Model for Order-Independent Transparency*. Kyseisessä tutkimuksessa esitellään reaaliaikaiseen renderöimiseen tarkoitettu *OIT*-menetelmä, joka pyrkii simuloimaan läpinäkyvyyteen liittyviä fysikaalisia ilmiöitä paremmin, kuten miten valonsäteet käyttäytyvät kulkiessaan jään tai värillisen lasin lävitse.

McGuire ja Enderton (2011) tutkimus *Colored Stochastic Shadow Maps* esittelee menetelmän läpinäkyvien objektien värillisten varjojen renderöimiseen, missä hyödynnetään nimensä mukaisesti samoja ideoita kuin *Stochastic Transparency* -menetelmässä.

Münstermann ym. (2018) ja Sharpe (2018) kutsuma *Moment Transparency* jatkaa *BWOIT*-menetelmän pohjan päälle parantamalla läpinäkyvien fragmenttien painoarvojen tarkempaa laskemista hyödyntämällä ideoita, jotka ovat alunperin esitetty varjokarttojen renderöimiseen tarkoitetussa *Moment Shadow Mapping* -tutkimuksesta, jonka ovat julkaisseet Peters ja Klein (2015).

Lähteet

Akenine-Moller, Tomas, Eric Haines ja Naty Hoffman. 2008. *Real-Time Rendering 3th Edition.* 3. painos. A K Peters/CRC Press, 25. heinäkuuta 2008.

Bavoil, Louis, ja Kevin P. Myers. 2008. "Order Independent Transparency with Dual Depth Peeling".

Bonzai Software. 2013. "Stochastic Order Independent Transparency". https://blog.bonzaisoftware.com/stochastic-order-independent-transparency/.

Carpenter, Loren. 1984. "The A -buffer, an antialiased hidden surface method". Teoksessa *Proceedings of the 11th annual conference on Computer graphics and interactive techniques,* 103–108. SIGGRAPH '84. New York, NY, USA: Association for Computing Machinery, 1. tammikuuta 1984. ISBN: 978-0-89791-138-2, viitattu 4. helmikuuta 2020. https://doi.org/10.1145/800031.808585. https://doi.org/10.1145/800031.808585.

Enderton, Eric, Erik Sintorn, Peter Shirley ja David Luebke. 2010. "Stochastic transparency". Teoksessa *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games,* 157–164. I3D '10. Washington, D.C.: Association for Computing Machinery, 19. helmikuuta 2010. ISBN: 978-1-60558-939-8, viitattu 28. huhtikuuta 2020. https://doi.org/10.1145/1730804.1730830. https://doi.org/10.1145/1730804.1730830.

McGuire, Morgan, ja Louis Bavoil. 2013. "Weighted Blended Order-Independent Transparency". *Journal of Computer Graphics Techniques Vol. 2, No. 2, 2013* 2 (2).

McGuire, Morgan, ja Eric Enderton. 2011. "Colored stochastic shadow maps". Teoksessa *Symposium on Interactive 3D Graphics and Games,* 89–96. I3D '11. San Francisco, California: Association for Computing Machinery, 18. helmikuuta 2011. ISBN: 978-1-4503-0565-5, viitattu 19. tammikuuta 2020. https://doi.org/10.1145/1944745.1944760. https://doi.org/10.1145/1944745.1944760.

McGuire, Morgan, ja Michael Mara. 2016. "A phenomenological scattering model for order-independent transparency". Teoksessa *Proceedings of the 20th ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games,* 149–158. I3D '16. Redmond, Washington: Association for Computing Machinery, 27. helmikuuta 2016. ISBN: 978-1-4503-4043-4, viitattu 27. tammikuuta 2020. https://doi.org/10.1145/2856400.2856418. https://doi.org/10.1145/2856400.2856418.

Meshkin, Houman. 2007. "Sort-Independent Alpha Blending". Game Developers Conference.

Microsoft. 2020. "Premultiplied alpha". Win2D Documentation, 4. heinäkuuta 2020. Documentation. Viitattu 4. heinäkuuta 2020. https://microsoft.github.io/Win2D/html/PremultipliedAlpha.htm.

Münstermann, Cedrick, Stefan Krumpen, Reinhard Klein ja Christoph Peters. 2018. "Moment-Based Order-Independent Transparency". *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 1, numero 1 (25. heinäkuuta 2018): 7:1–7:20. Viitattu 16. helmikuuta 2020. https://doi.org/10.1145/3203206. https://doi.org/10.1145/3203206.

NVIDIA GameWorks. 2014. "Transparency (or Translucency) Rendering". NVIDIA Developer, 20. lokakuuta 2014. Viitattu 9. maaliskuuta 2020. https://developer.nvidia.com/content/transparency-or-translucency-rendering.

———. 2020. "Weighted Blended OIT Sample". GAMEWORKS Library, 7. heinäkuuta 2020. Documentation. Viitattu 7. heinäkuuta 2020. https://docs.nvidia.com/gameworks/content/gameworkslibrary/graphicssamples/opengl_samples/weightedblendedoitsample.htm.

Peters, Christoph, ja Reinhard Klein. 2015. "Moment shadow mapping". Teoksessa *Proceedings of the 19th Symposium on Interactive 3D Graphics and Games,* 7–14. i3D '15. San Francisco, California: Association for Computing Machinery, 27. helmikuuta 2015. ISBN: 978-1-4503-3392-4, viitattu 3. maaliskuuta 2020. https://doi.org/10.1145/2699276.2699277. https://doi.org/10.1145/2699276.2699277.

Porter, Thomas, ja Tom Duff. 1984. “Compositing digital images”. Teoksessa *Proceedings of the 11th annual conference on Computer graphics and interactive techniques,* 253–259. SIGGRAPH ’84. New York, NY, USA: Association for Computing Machinery, 1. tammikuuta 1984. ISBN: 978-0-89791-138-2, viitattu 15. helmikuuta 2020. https://doi.org/10.1145/800031.808606. https://doi.org/10.1145/800031.808606.

Sellers, Graham, Richard S. Jr. Wright ja Nicholas Haemel. 2014. *OpenGL SuperBible: Comprehensive Tutorial and Reference.* 6. painos. Addison-Wesley.

Sharpe, Brian. 2018. “Moment transparency”. Teoksessa *Proceedings of the Conference on High-Performance Graphics,* 1–4. HPG ’18. Vancouver, British Columbia, Canada: Association for Computing Machinery, 10. elokuuta 2018. ISBN: 978-1-4503-5896-5, viitattu 18. helmikuuta 2020. https://doi.org/10.1145/3231578.3231585. https://doi.org/10.1145/3231578.3231585.

The Khronos Group. 2014. “glBlendFunc - OpenGL 4 Reference Pages”. Viitattu 10. maaliskuuta 2020. https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glBlendFunc.xhtml.