Reaaliaikaisen globaalin valaistuksen toteutus Godot-pelimoottorissa

Turun yliopisto
Tietotekniikan laitos
LuK-tutkielma
Tietojenkäsittelytiede
Joulukuu 2024
Mikko Salonen

TURUN YLIOPISTO
Tietotekniikan laitos

Mikko Salonen: Reaaliaikaisen globaalin valaistuksen toteutus Godot-pelimoottorissa

LuK-tutkielma, 38 s., 3 liites.
Tietojenkäsittelytiede
Joulukuu 2024

Tutkielma tarkastelee globaalin valaistuksen implementaatiota pelimoottoreissa keskittyen erityisesti Godot-pelimoottoriin. Tutkielmassa Godotin käytännön globaalin valaistuksen menetelmiä käydään läpi, ja niiden implementaatio sidotaan teoriaan. Työssä käsitellään erityisesti renderöintiyhtälöä ja Godotin tarjoamia erilaisia BRDF:iä ja niiden yksityiskohtia. Lisäksi esitellään uusia kehityksiä alalla Godotin tulevaisuuden kehityksen näkökulmasta, kuten polunseurantaa ja ReStir-algoritmia. Yksityiskohtiin liittyen tarkastellaan esilasketun valaistuksen menetelmiä kuten valokarttoja ja valokennoja. Valokennojen osalta tutustutaan kuinka harmonisia pallofunktioita voidaan soveltaa vähentämään tarvittavan muistin käyttöä. Renderöintiyhtälön osalta esitellään myös, mitä on ympäristön okkluusio ja kuinka se lasketaan SSAO-algoritmilla.
Godotin tällä hetkellä käytössä oleva globaalin valaistuksen menetelmien SDFGI:n ja vokselikartio jäljityksen toiminta on esitelty työssä. Tulevaisuudessa Godotin reaaliaikainen globaalin valaistuksen SDFGI-algoritmi on korvaantumassa HDDAGI-algoritmilla.

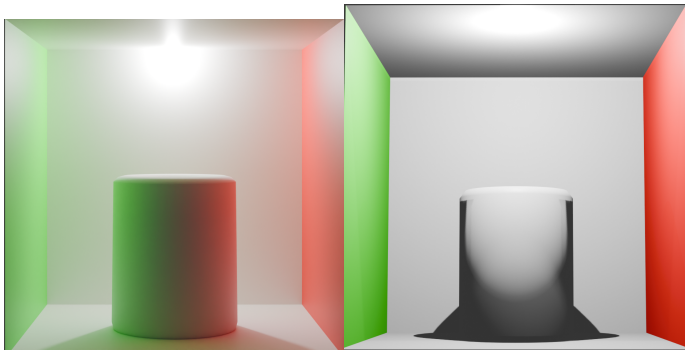
Asiasanat: Renderöinti, 3D-Grafiikka, Globaali valaistus, pelimoottorit, Godot

Sisällys

Kuvat

1 Johdanto

Globaalivalaistus (engl. *global illumination*) on ryhmä menetelmiä epäsuoran valon (engl. *indirect light*) laskemiseen kolmiulotteisissa virtuaaliympäristöissä. Ilman epäsuoraa valaistusta valaistus kokonaisuutena näyttää yleisesti ottaen selvästi keinotekoiselta. Fotorealistisen kuvan saamiseksi epäsuoran valaistuksen laskeminen on välttämätöntä koska epäsuoraa valaistusta tapahtuu aina todellisessa maailmassa. Kuvassa 1 nähdään epäsuoran valon puutteen vaikutus.



Kuva 1.1: Vasemmalla Blenderin Cycles Renderöijä (epäsuoralla valaistuksella polunjäljityksen kautta). Oikealla Eevee renderöijä ilman epäsuoraa valaistusta.

Sylinterin alaosa on aivan liian pimeä todellisuuteen nähden sillä valo ei heijastu siihen seinästä. Seinästä heijastuva valo ei myöskään värjää sylinteriä kuvassa jossa epäsuora valaistus puuttuu.

Epäsuoran valon laskeminen reaaliajassa tuo kuitenkin omat haasteensa, sillä videopelien yleensä halutaan renderöivän 60 kuvaa sekunnissa. Tämä jättää vain noin 0.0167 sekunttia valaistuslaskuille ottamatta huomioon kaikkia muita operaatioi-

ta kuten fysiikkalaskuja, pelilogiikkaa, jälkikäsittelyä ja transformaatio-operaatiota. Haastetta lisää se, että pelin aikana ympäristö usein muuttuu pelaajan ja ei-pelaajahahmojen(engl. *Non-player characters, NPC*) liikkumisen tai tuhoutuvien objektien seurauksena. Ideaalitapauksessa olisi haluttavaa, että valaistus reagoisi näihin ympäristön muutoksiin.

1.1 Tutkimuskysymys ja -menetelmä

Tässä työssä tulen käymään läpi menetelmiä joita pelimoottorit käyttävät reaaliaikaisen globaalin valaistuksen laskemiseen keskittyen Godot-pelimoottoriin.[1] Työ on ilmiötä selittävä, ja tietoa on haettu menetelmien lähdekoodeista sekä joko suoraan tai epäsuorasti menetelmien perustana olevista tutkimuspapereista. Tärkeinä lähteinä toimivat SIGGRAPH-konferenssissa julkaistut tutkimukset, ACM:n vuonna 1969 perustettu tietokonegrafiikkaan erikoistuva erikoisalayhteisö, jossa julkaistaan paljon alan kärkitutkimusta sekä akateemiselta, että yrityspuolelta.

1.2 Sanastoa

Työssä käytetään runsaasti alan erikoistermejä. Näistä keskeisimpiä avataan tässä.

Godot on Juan Linietskyn ja Ariel Manzur alunperin kehittämä pelimoottori se tehtiin avoimen koodin projektiksi vuonna 2014. Godot on suosittu erityisesti harrastelijoiden joukossa mutta myös jotkin studiot ovat ottaneet sitä käyttöön kuten esimerkiksi Blind squirrel games -pelissä *Sonic colors ultimate*, joka käytti muokattua versiota Godot-pelimoottorista.

Renderöintiyhtälö on yhtälö, joka ottaen parametriksi katseluvektorin ja valovektorin palauttaa pikselin värin huomioiden pisteen valaistuksen. Renderöintiyhtälöstä kerrotaan lisää luvussa 2.

[1]Godot-pelimoottori `https://godotengine.org/`

BRDF eli kaksisuuntainen heijastuksen jakaumafunktio on funktio joka ottaa parametriksi katseluvektorin ja valovektorin, ja palauttaa pinnasta heijastuvan valon määrän. BRDF:stä kerrotaan lisää luvussa 4.

Energian virtaus (engl. *flux*) on sekunnissa virtaavan energian määrä. esimerkiksi, jos valo säteilee 100 joulea energiaa 10 sekunnissa niin sen energian virtaus on 100J/10s=10W. [1]

Radianssi on tietystä suunnasta tulevan valon määrä. **Irradianssi** on puolestaan määritelty energiavirtauksen määränä rajatulla alueella. Eli toisin sanoen kaikista suunnista tulevan valaistuksen määrä yhdessä pisteessä (yks. W/m^2).

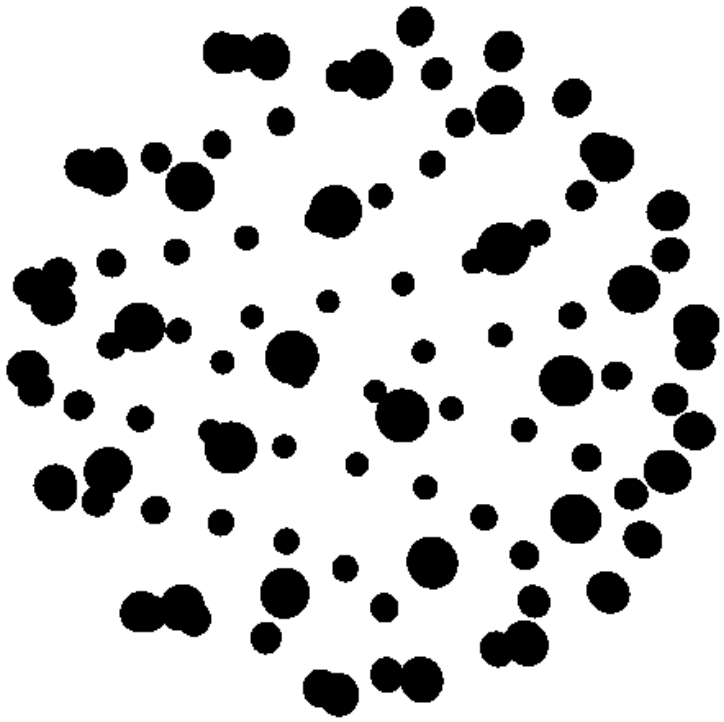
Z-puskuri (engl. *Z-Buffer*) on tekstuuri johon kirjoitetaan ruudun syvyysinformaatio erityisesti z-puskuri rasteroinnin yhteydessä. Syvyystietoa tarvitaan määrittämään, mikä geometria on näkyvissä.

G-puskuri (engl. *G-Buffer*) on tekstuuri johon kirjoitetaan muummuassa normaalit, albedo ja mahdollisesti muuta tietoa riippuen implementaatiosta. G-puskuria käytetään tyypillisesti viivytetyn varjostuksen (engl. *deferred shading*) yhteydessä, jotta valaistus saataisiin laskettua myöhemmässä vaiheessa. Viivytetyn varjostuksen yksityiskohdat jäävät tämän työn ulkopuolelle.

Vogelin spiraali on Helmut Vogelin vuonna 1978 kehittämä malli [2] auringonkukan mykerön spiraali-muodon generoimiseen, Godotissa tämä malli on implementoituna funktiossa Vogel_hemisphere, listaus 1, ja se on visualisoituna kuvassa 1.2 Shadertoy -työkalun[2] avulla. Godot käyttää sitä jakamaan pisteitä tasaisesti pallon pinnalle.

Digitaalinen differentiaalinen analysoija (engl. *Digital Differential Analyser*) on algoritmi viivan piirtämiseen ruudukossa ruutu puskurissa (engl. Frame buffer). Se voidaan myös laajentaan käsittelemään vokseli-ruudukoita nopeuttamaan säteenmarssitusta. [4]

[2]Shadertoy, työkalu osoitteessa https://www.shadertoy.com/



Kuva 1.2: Godotin *vogel_hemisphere* funktio [3] visualisoituna Shadertoyssa arvoilla (n, 100, 0) jossa n on numero 0 ja 100 välillä.

Lyhenne **MIP** tulee latinan fraasista "*Multim in parvo*", paljon pienessä. MIP-ketju, kartta tai pyramidi on ketju tekstuurista generoituja syvemmällä ketjussa pieneviä kuviä. Esimerkkinä 32x32 kuvan MIP ketju olisi seuraavanlainen: MIP0=32x32 → MIP1=16x16→MIP2=8x8→MIP3=4x4→MIP4=2x2→MIP5=1x1 MIP-ketjut saadaan laskettua yksinkertaisimmin tekstuureille, joiden koko on kahden potenssi. Aikaisemmat versiot joistain grafiikka ohjelmoinnin rajapinnoista tukivat vain tekstuureja joiden ulottuvuudet olivat kahden potensseja. Nykyiset rajapinnat tukevat myös tekstuureita, jotka eivät ole kahden potensseja kooltaan mutta algoritmi näiden käsittelyyn ei ole yhtä nopea. MIP-ketjut toimivat kuten yksityiskohtaisuustaso (engl. *level of detail*, LOD) mutta tekstuureille. Mikäli MIP-ketjua ei generoitaisi tekstuurille ja pikselin (engl. *pixel/picture element*) koko olisi huomattavasti suurempi, kuin tekselin(tekstuurin pikseli tai tekstuurin elementti) koko ruudulla, jouduttaisiin tekstuuri skaalaamaan reaaliajassa.

Ohjelmalistaus 1 Tyyppiluokka 'Vogel hemisphere'. [3]

```
1  const float PI = 3.14159265f;
2  const float GOLDEN_ANGLE = PI * (3.0 - sqrt(5.0));
3
4  vec3 vogel_hemisphere(uint p_index, uint p_count, float p_offset) {
5          float r = sqrt(float(p_index) + 0.5f) / sqrt(float(p_count));
6          float theta = float(p_index) * GOLDEN_ANGLE + p_offset;
7          float y = cos(r * PI * 0.5);
8          float l = sin(r * PI * 0.5);
9          return vec3(l * cos(theta), l * sin(theta), y * (float(p_index
               & 1) * 2.0 - 1.0));
10 }
```

Vokseli eli volumetrinen pikseli. On pikselin vastine kolmessa ulottuvuudessa. Vokselit esitetään yleensä kolmiulotteisena ruudukkona joka voidaan pakata kolmiulotteiseen tekstuuriin. Monille varmasti tutuin esimerkki vokselien käytöstä on Minecraftin proseduraalinen maaston generointi (engl. *procedural terrain generation*), mutta vokseleja hyödynnetään myös esimerkiksi valaistusmenetelmissä ja simulaatioissa.

Harva kahdeksan oksainen vokselipuu Harva kahdeksan oksainen vokselipuu eli (engl. *Sparce Voxel Octree, SVO*) on hierarkinen rakenne, jolla monimutkaista geometriaa saadaan esitettyä tehokkaasti vokseleina. Rakenne rakennetaan jakamalla vokseli aina kahdeksaan ali-vokseliin kunnes ollaan määrätyllä syvyydellä puussa, tai kunnes vokseli pystyy täydellisesti esittämään sisältämänsä geometrian.

1.3 Rakenne

Luvussa 2.1 käsitellään kevyesti 3D-valaistuksen teoreettista perustaa. Jonka jälkeen luvussa 2 käydään läpi renderöintiyhtälöä ja miten se liittyy valaistus menetelmiin. Luvussa 4 käsitellään tarkemmin renderöintiyhtälön BRDF-termiä ja esitellään Godot-pelimoottorissa käytettävissä olevia BRDF-toteutuksia. Luvussa 5 esitellään Godotin nykyisiä globaalin valaistuksen menetelmiä. Seuraavaksi luvussa 6 käsitellään säteenjäljitystä ja sen toimintaa. Kappaleessa käsitellään myös, miten

sitä käytetään tällä hetkellä pelimoottoreissa. Lopuksi käsitellään globaalin valaistuksen tulevaisuutta pelimoottoreissa. Sen jälkeen keskitytään syvemmin tiettyihin nykyisesti käytössä oleviin menetelmiin. Lopuksi tuodaan esille uusimpia menetelmiä.

2 Renderöintiyhtälö

Tässä luvussa käsitellään renderöintiyhtälöä ja sen teoriaa. Ensin käydään läpi mikä renderöinti yhtälöä yleisesti. Sen jälkeen pohjustetaan sen teoriaa. Sen jälkeen esitellään lyhyesti PBR-renderöinninn käsite. Lopuksi käydään läpi ympäristön okkluusiota ja sitä, miten se sopii renderöinti yhtälöön.

Renderöintiyhtälö esiteltiin James Kajiyan [5] julkaisussa *The Rendering Equation* (1986). Se toimii perustana renderöintitekniikoille. Renderöintiyhtälölle löytyy useita eri muotoja riippuen lähteestä jota käytetään. On esimerkiksi muotoja jotka jättävät säteily termin (engl. *emission*) pois yhtälöstä. Kajiya esittää seuraavanlaisen muodon.

$$L_o(\omega_0) = L_e(\omega_i, \omega_o) + \int_\Omega f_i(\omega_i) \cos\theta_i f_r(\omega_i, \omega_0)\, d\omega_i$$

Renderöintiyhtälö koostuu kolmesta osasta BRDF (*engl. bidirectional reflectance distribution function*), valotermistä joka kertoo kuinka paljon valoa tulee tietysta suunnasta ω_i ja pinnan normaalin ja valonsäteen muodostaman kulman kosinista. Johan Heindrich Lambertin *Photometrie. Photometria, sive De mensura et gradibus luminis, colorum et umbrae* teoksessa esittämän Lambertin kosini laki sanoo, pintaan tuleva irradianssi vaihtelee valon saapumiskulman kosinin mukaan. Käytännössä se lasketaan valo-vektorin ja normaalin pistetulona. Saatu yhtälö integroidaan hemisfäärin yli, jotta saadaan laskettua kaikkien suuntien valojen kumulatiivinen vaikutus pisteen valaistukseen. Käytännössä renderöintiyhtälön tapauksessa inte-

graalin tulos on yhtäsuuri kaikkien suuntien valokontribuution keskiarvon kanssa. $\cos\theta_i$ on yhtälössä Lambertin diffuusi eli otetaan valonsäteen ja pinnan normaalin kulma. Se voidaan myös laskea valonsäteen ja pinnan normaalin normalisoitujen vektoreiden pistetulona.

Valaistuksen ongelma on mahdollista formuloida myös toisin. Eric Veach:kin polkuintegraali formulaatio(engl. *Path Integral Formulation*). Siinä missä Kajiyan muoto on integraali kaikista suunnista tulevasta valosta, Veachin yhtälö on integraali kaikkien valopolkujen yli.

2.1 Sädeoptiikka

Valo koostuu hiukkasista joita kutsutaan fotoneiksi. Sädeoptiikka kuvailee näiden hiukkasten kulkua. Sitä voidaan mallintaa säderakenteella. Sädeoptiikka tekee muutaman yksinkertaistavan oletuksen valosta. Valo kulkee suoraa polkua homogeenisessa väliaineessa. Valo taittuu ja saattaa jakautua kahtia kahden erillaisen aineen rajapinnassa, jos refraktinen indeksi muuttuu. Valo saattaa imeytyä tai heijastua. Tämä on yksinkertaistus todellisuudesta mutta antaa hyvän lähtökohdan valon mallintamiseen.

Käytännössä valoa voidaan kuvata säderakenteella. Voimme mallintaa tätä säderakennetta käyttäen kolmea kolmiulotteisesta vektoria jotka määräävät säteen lähtöpisteen, suunnan ja värin. Koska olemme kiinnostuneita katsojan silmään päätyvästä valosta merkitään sitä symbolilla $L_o(\omega_o)$ Yksinkertaisin tapaus on mikäli katsomme suoraan valon lähdettä. Silloin $L_o(\omega_o) = L_e(\omega_i, \omega_o)$ eli pinnan väri on pinnan säteilemän valon määrä. Tarkastaan seuraavaksi suoraa valaistusta, eli tapausta jossa valo heijastuu pinnasta suoraan silmään. Kuten voidaan huomata, kun katsomme mitä tahansa esinettä ympärillämme pinta on kirkkaimmillaan kun valo on suoraan sen yläpuolella, ja pimeimmillään, kun valo paistaa siitä pois päin. Tiedetään siis, että on olemassa suhde valon suuntavektorin ja pinnan normaalivektorin, eli vektorin,

joka on orthogonaali pinnan pisteessä muodostaman tason, välillä. Renderöinnissä operoidaan tyypillisesti kolmioilla. Määritellään verteksi pisteeksi kolmiulotteisessa avaruudessa, ja kolmio rakenteeksi joka koostuu kolmesta verteksistä *v1, v2, v3*. Kahden vektorin ristitulo antaa meille niiden määrittämän pinnan kanssa ortogonaalisen vektorin. On tärkeä huomata, että järjestyksellä on tässä väliä sillä

$$a \times b = -(b \times a).$$

Kolmion normaali saadaan siis kaavalla

$$normalisoi((v1 - v2) \times (v1 - v3)).$$

Kosini-funktio antaa ykkösen mikäli vektorien välinen kulma on nolla, ja nollan, jos vektorien välinen kulma on 90 astetta ja miinus ykkösen, jos kulma on 180 astetta. Koska säde törmää pintaan, osa sen energiasta imeytyy siihen muuttaen valon väriä. Tätä voidaan mallintaa ottamalla Hadamardin tulo (Hadamardin tulossa jokainen vektorin elementti kerrotaan toisen vektorin vastaavalla elementillä.) väri vektorin ja pinnan värivektorin kanssa. Nyt saadaan yhtälöksi

$$L_o(\omega_o) = L_e(\omega_i, \omega_o) + albedo \circ lightColor \cdot cos\theta$$

Tämä ei kuitenkaan mallinna kiiltäviä materiaaleja joten on selvää, että ei ole riittävä. Yllä oleva malli olettaa, että valo heijastuu kaikkiin suuntiin tasaisesti. Ei ole materiaaleja joissa näin käy täydellisesti mutta esimerkiksi karkea päällystämätön puupinta on melko lähellä. Yleensä materiaalit kiiltävät ainakin hieman. Suurempi osa valosta heijastuu heijastusvektorin suuntaan. Lisätään yhtälöön termi kuvaa-

maan tätä

$$L_o(\omega_o) = L_e(\omega_i, \omega_o) + albedo \circ lightColor \cdot cos\theta + (normalisoi(\omega_i + \omega_o) \cdot normal)^{kiiltavyys}$$

Tällä yhtälöllä saadaan mallinnettua suoraa valaistusta yhdestä valonlähteestä. Jotta saadaan mukaan suora valaistus kaikista valonlähteistä voidaan summata yhtälö kaikille valonlähteille.

$$L_o(\omega_o) = L_e(\omega_i, \omega_o) + \sum_{i=0}^{n} albedo \circ lightColor \cdot cos\theta + (normalisoi(\omega_i + \omega_o) \cdot normal)^{kiiltavyys}$$

On tärkeää huomata, että fyysisesti realistisissa malleissa vain emissiiviset pinnat lähettävät enemmän valoa kuin ottavat vastaan. Sillä energiaa ei ilmesty tyhjästä.

2.2 PBR-materiaalimallit

Lähes kaikki modernit renderöijät ovat PBR-renderöijiä (engl. *Physically Based Rendering*). PBR renderöijät noudattavat muutamaa perusperiaatetta jotta malli olisi PBR sen täytyy noudattaa energian konservaation periaatetta, sen pitää perustua mikrofasetti teoriaan. Energian konservaation periaate kertoo meille, että $\Phi_i - \Phi_o = \Phi_e - \Phi_a$. [1] Eli pinnan sisään ja ulos tulevan energian ero on yhtäsuuri, kuin ero pinnan säteilemän ja absorboiman energian välillä.

PBR-materiaali mallissa on vähintään kolme termiä: Mallin pohjaväri(albedo), metallisuus ja karkeus. Dielectric tai metallisuus joka liittyy materiaalin sähkönjohtamiseen. Vaikka metallisuus yleensä määritetään arvona ykkösen ja nollan välillä jossa nolla on dielectrinen ja yksi on metallinen käytännössä tämä arvo muutamia poikkeuksia lukuunottamatta joko yksi tai nolla. Viimeisenä välttämättömänä terminä on karkeus(engl. *roughness*) eli kuinka tasainen mallin pinta on mikroskooppisella tasolla. Esimerkiksi unity-pelimoottorin standardivarjostimessa tämä määritellään

termillä tasaisuus (engl. *smoothness*), joka tarkoittaa samaa asiaa mutta toisin päin eli tasaisuus = 1 - karkeus. Godot-pelimoottorissa tämä on määritetty karkeutena. Suuri karkeusarvo vaikuttaa valaistukseen muuttamalla materiaalista heijastuvan valon säteen kulmaa satunnaisesti. Tällöin materiaalin heijastus on tarkka. Karkeus on sumuinen kun karkeus on hieman suurempi kuin 0 ja heijastus ei ole näkyvä kun karkeus on 1.

2.3 Ympäristön okkluusio

Aiemmin ja taiteellisista valinnoista riippuen vielä nykyäänkin epäsuoraa valaistusta ollaan jäljitelty ympäristön okkluusio (engl. *ambient occlusion*, AO). AO ei ole fotorealistinen efekti vaan ennemminkin karkea approksimaatio globaalista valaistuksesta. AO perustuu oletukseen siitä, että jos pinnan pisteen lähellä on geometriaa vähemmän valoa pääsee siihen pisteeseen. Eli mitä enemmän pisteen lähiympäristö on asioita sitä vähemmän valoa pisteeseen pääsee. AO saadaan yhdistettyä renderöintiyhtälöön seuraavasti. Renderöintiyhtälön

$$L_o(\omega_0) = \int_\Omega f_i(\omega_i) \cos\theta_i f_r(\omega_i, \omega_0)\, d\omega_i$$

valotermif_i jaetaan termiin

$$f_{suora}$$

ja

$$f_{epsuora}$$

jossa epäsuora valaistus lasketaan

$$Ambient * K_a \frac{1}{\pi} \int_\Omega V(\omega_i) cos(\theta) d\Omega$$

jossa K_a on materiaalitermistä esilaskettuvakio. *Ambient* on vakio, joka määrittää valon määrän. $(\int_\Omega V(\omega_i)cos(\theta))$ on näkyvyystermi joka kertoo kuinka suuri osa pisteen ympärillä olevasta hemisfääristä on geometrian sisällä. Tämän arviointiin on monia eri tapoja. Yksi yleisimmistä on Crytek:in Crysis peliä varten vuonna 2007 kehittämä SSAO (engl. *Screen Space Ambient Occlusion*) jossa otetaan otantoja hemisfäärin sisältä ja käytetään G-puskurin (tai z-puskurin riippuen käytetäänkö forward vai deferred shading:iä) syvyys- ja normaali arvoja arvioimaan onko otanta geometrian sisällä. Jos se on, $V(\omega_i)$ on 0. Muuten se on 1. SSAO on nykyään yleinen AO tekniikka pelimoottoreissa ja myös Godot-pelimoottorissa on sen implementaatio.

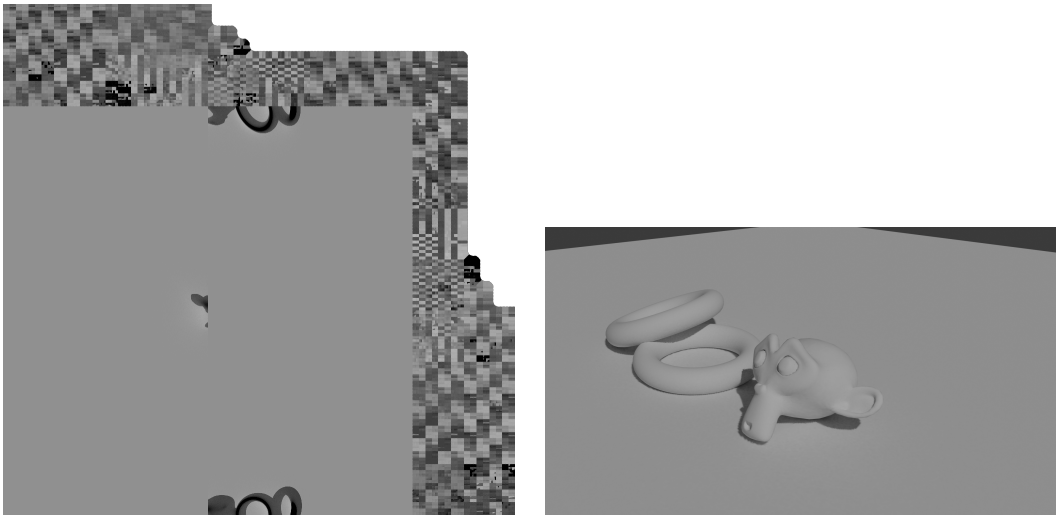
3 Esilasketut menetelmät

Tässä luvussa käsitellään esilaskettuja menetelmiä globaaliin valaistukseen. Luku aloitetaan käsittelemällä valokarttoja. Sen jälkeen esitellään kvuaperustainen valaistus (engl. *Image-based Lighting, IBL*). Lopuksi käsitellään harmonisia pallofunktioiden matematiikkaa.

3.1 Valokartat

Toinen menetelmä on laskea globaalivalaistus valmiiksi valokarttoihin (engl. *lightmaps*) tai valokennoihin (engl. *light probes*). Valokartta on tekstuuri, johon on leivottu (engl. *baked*) objektin valaistusarvot yleensä pelimoottorissa mutta myös 3D-mallinnusohjelmat kuten Blender kykenevät myös leipomaan valaistuksen tekstuuriin.

Kuvassa a on kuvan b leivottu valokartta. Valokartta kerrotaan renderöidessä objektin albedolla, jotta saadaan esitettävä väri (Jälkikäsittelyä huomioon ottamatta). IBL on tekniikka jossa pisteen irradianssi tallennetaan valokennoon. Valokenno kuvaa pisteeseen kaikista suunnista tulevan valon määrää eli irradianssia. Tämä määrä voidaan tallentaa joko kuutiokarttana (engl. *cube map*) tai harmonisen pallofunktion kertoimina. Kummassakin tapauksessa valo saadaan ottamalla otanta kuutiokartasta tai funktiosta heijastusvekorin suunnassa. Tämä antaa realistisen tuloksen pisteen lähellä tai, jos ympäristön esineet ovat kaukana pisteestä. Jotta valaistus saadaan toimimaan kaikissa tilanteissa tarvitaan lisää valokennoja joita interpoloi-



(a) Leivottu valokartta. (b) Lähdeskene.

Kuva 3.1: Valokartta

daan kameran sijainnin mukaan. Koska valokennoja tarvitaan monia ja GPU:n (engl. *engl. Graphics Processing Unit*) muisti on usein prosessointitehoa rajoittavampi tekijä, on järkevää tallentaa valokennot käyttäen harmonisten pallofunktioiden kertoimia. Kuutiokartan tapauksessa karkeutta saadaan simuloitua ottamalla otannat eri MIP-tason kuvista. Kun taas harmonisilla pallofunktioilla Otannan otto tapahtuu projektoimalla valaistusfunktion ja siirtymäfunktion harmoniseen pallofunktioon ja ottamalla niiden pistetulon. [6]

3.2 Harmoniset pallofunktiot

Harmoniset pallofunktiot ovat Laplacen yhtälön ratkaisut, jotka ovat rajattuja palloon. Ne ovat pallolle vastaavat mitä Fourierin kannat ovat yksikköympyrälle. Harmonisten pallofunktioiden kannat ovat muotoa

$$Y_l^m(\theta,\phi) = K_l^m e^{im\phi} P_l^{|m|}(cos(\theta)), l \in N, -l <= m <= l$$

Jossa P_l^m ovat vastaavat Legendren polynomi ja K_l^m ovat normalisaatio vakiot. Annetulle järjestysluvulle l on 2l+1 kertoimia eli jos l=N niin kertoimien määrä järjes-

tysluvulle on (2*N+1) ja tarvittavien kertoimien määrä on yhteensä $(N+1)^2$. Mitä suurempi järjestysluku on sitä tarkempaa tietoa pystytään tallentamaan. Funktio f(s) saadaan projektoitua integroimalla f(s) kerrottuna kantafunktioilla. Eli kerroin c_l^m saadaan kaavalla $c_l^m = \int_s f(s) y_l^m(s) ds$ Alkuperäisen funktion approksimaatio saadaan uudelleen rakennettua kertoimista kaavalla $f(s) \approx \sum_{l=0}^{n} \sum_{m=-l}^{l} c_l^m y_l^m(s)$

Pelimoottoreissa käytetään tyypillisesti maksimissaan toisen tai kolmannen asteen pallofunktioita. [7] Godot valitsee muuttujan SH_SIZE mukaan joko toisen tai kolmannen asteen pallofunktion ja allokoi 3*16 tai 3*9+1 pituisen taulukon kertoimille. SDFGI:tä laskiessa Godot ampuu valokennosta Vogelin spiraalilla tasaisesti joka suuntaan säteitä, ja tallentaa valoantureihin radianssin ja irradianssin. SDFGI:tä käydään tarkemmin läpi luvussa 5.1 [8] [9]

4 Kaksisuuntaiset heijastuksen jakaumafunktiot

Luvussa käsitellään Kaksisuuntaisia heijastus jakaumafunktiota (engl. *Bidirectional Reflectance Distribution Function*, BRDF) Ensin yleisesti ja sen jälkeen esitellen muutamaa Godotin käyttämää BRDF-toteutusta.

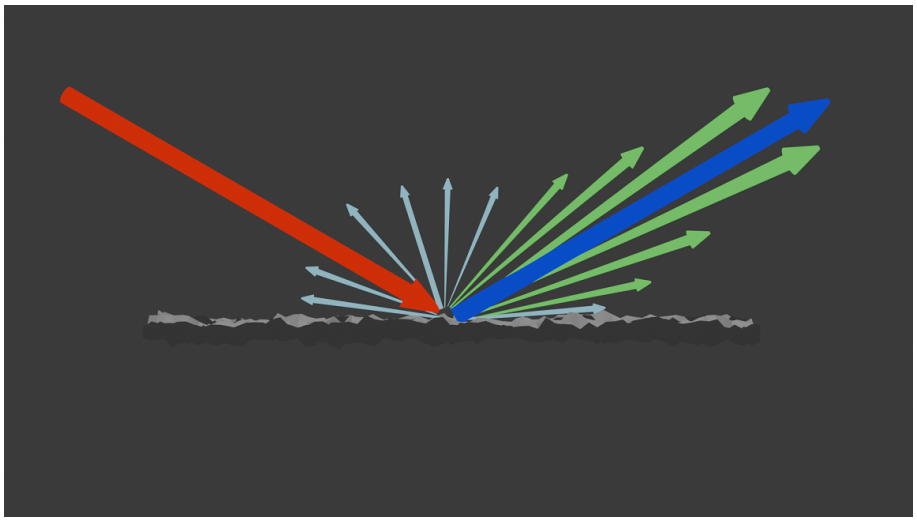
4.1 Yleisesti

BRDF on funktio joka määrää miten valo heijastuu. BRDF koostuu kahdesta osasta, diffuusista ja spekulaarisesta termistä. Diffuusi termi kuvaa valoa joka heijastuu pinnasta tasaisesti joka suuntaan, kun taas spekulaari termi kuvaa pinnan normaalin suhteen heijastuvaa valoa. Diffuusit heijastukset näyttävät samalta riippumatta siitä mistä suunnasta katsoja katsoo pistettä, kun taas spekulaarit heijastukset muuttuvat riippuen katselu kulmasta. Seuraavan sivun kuva 4.1 visualisoi BRDF:n käyttäytymistä.

Diffuusi on kuvassa merkittu vaaleansinisillä nuolilla jotka osoittavat jokaiseen suuntaan, spekulaari on merkitty vaaleanvihreällä. Kuten kuvasta näkee spekulaariheijastus on vahvimmillaan, kun heijastusvektori osoittaa kameraan. Jos merkitään katseluvektoria V ja valovektoria L. Koska BRDF on kaksisuuntainen (engl. *bidirectional*) ei ole väliä lasketaanko $BRDF(V, L)$ vai toisin päin eli $BRDF(L, V)$.

Diffuuseja BRDF:iä on monia ja ne voidaan vaihtaa toiseen muuttamatta muuta

osaa koodista. Godotissa on valmiiksi ohjelmoituna diffuuseista *Burley*, *Lambert*, *Lambert Wrap*, ja *toon* ja spekulaarisia *schlickGGX* sekä *toon*. On myös mahdollista kirjoittaa omia BRDF:iä (Käyttämällä godotin omaa varjostin kieltä, joka perustuu GLSL:n). Alla avaan näistä muutamaa tarkemmin



Kuva 4.1: BRDF:n kaksiulotteinen visualisaatio

4.2 Lambert

Lambert on yksinkertaisin kaava diffuusille valaistukselle $N \cdot L$ jossa N on normalisoitu normaalivektori ja L on Normalisoitu valovektori. Teknisesti ottaen yhtälö pitäisi kertoa normalisaatiovakio $\frac{1}{\pi}$ jotta yhtälö seuraisi energian säilyvyyden lakia, mutta erityisesti pelimoottorien yhteydessä tämä usein jätetään pois. Tämä johtuu siitä että on taiteilijoille helpompaa työskennellä arvoilla joissa puhdas valkoinen on ykkönen π:n sijaan

4.3 Lambert wrapped

Normaalissa lambertin valaistuksessa mikäli normaalin ja valovektorin kulma on suurempi kuin 90 astetta pintaan ei pääse yhtään valoa. Tämä ei kuitenkaan välttä-

mättä ole haluttavaa sillä se piilottaa pimeän puolen yksityiskohtia ja saattaa näyttää rumalta. Lambert wrap toimii siirtämällä Lambert funktion välille [0,1] kaavalla $\frac{(N \cdot L)+1}{2}$ $\frac{(N \cdot L)+1}{(1+roughness)*(1+roughness)*\pi}$

4.4 Toon-valaistus

Toon-valaistuksessa Normaalin BRDF:n tulos posteroidaan eli värien määrää vähennetään tarkoituksella tyylitellyn tuloksen saamiseksi. Yleensä tämä tehdään valitsemalla manuaalisesti jokin väri paletti ja kartoittamalla arvot siihen BRDF:n arvot. Godotin implementaatio poikkeaa tästä hieman. $diffuse_toon = smoothstep(-roughness, max(roughness, 0.01), NdotL) * (1.0/M_PI)$ $mid = (1.0 - roughness)^2$ $intensity = smoothstep(mid - roughness * 0.5, mid + roughness * 0.5, RdotV) * mid$ Godot käyttää paletin sijaan askel funktiota eri valaistus tasojen määrittämiseen. karkeus määrittää kuinka jyrkkä siirtymä eri valaistus tasojen välillä on.

4.5 Burley

Burley on Disneyn Brent Burleyn 2012 julkaistussa paperissa [10] esitetty mikrofasettimalliin pohjautuva BRDF. Mikrofasettimallin ideana on laskea pikseliin vaikuttavien mikrofacettien keskiarvo yksinkertaisella BRDF:llä. Mikrofacettimalliin pohjautuvan BRDF:n yleinen muoto on

$$\int_\Omega p_m(L,V,m)D(m)G_2(L,V,m)\frac{\langle m \cdot L\rangle}{|N \cdot L|}\frac{\langle m \cdot V\rangle}{|N \cdot V|}dm$$

Jossa integroidaan mikrofasetin normaalien yli. Termi

$$p_m(L,V,m)$$

funktiossa on mikrofasetin BRDF, termi

$$D(m)$$

on normaali m:n tiheysfunktio.

$$G_2(L, V, m)$$

on geometrinen okkluusio termi. Se on todennäköisyys, että mikrofasetti m:stä on suora linja sekä valoon L, että katselijaan V.

Diffuusivalaistus saadaan Burleyn mukaan kaavalla

$$f_d = \frac{baseColor}{\pi}(1 + (F_{D90} - 1)(1 - cos\theta_l)^5)(1 + (F_{D90} - 1)(1 - cos\theta_v)^5)$$

Godotin toteutus siitä on seuraavanlainen.

```
1 float FD90_minus_1 = 2.0 * cLdotH * cLdotH * roughness -
      0.5;
2 float FdV = 1.0 + FD90_minus_1 * pow(cNdotV,5);
3 float FdL = 1.0 + FD90_minus_1 * pow(cNdotL,5);
4 diffuse_brdf_NL = (1.0 / M_PI) * FdV * FdL * cNdotL;
```

Koodissa kosini on korvattu muuttujilla cNdotV ja cNdotL jotka ovat normaalivektorin nollan ja ykkösen välille rajatut pistetulot valovektorin (cNdotL) ja katsevektorin(cNdotV) välillä.

Spekulaariselle valaistukselle tiheysfunktio D(m) on GGX:ssä

$$D_{GGX}(h) = \frac{\alpha^2}{\pi(\langle N \cdot h \rangle^2(\alpha^2 - 1) + 1)^2}$$

$$\alpha = r_p^2$$

jossa

$$r_p$$

on materiaalin karkeus. Godot käyttää Earl Hammon, Jr. Game Developers Conference (GDC) puheessa [11] esittelemää approksimaatiota termille G2. G2 saadaan laskettua käyttämällä raytrace derivaatiota. Approksimaation tarkat matemaattiset perustelut jäävät työn ulkopuolelle, mutta se saadaan sievennettyä muotoon

$$G_2 = \frac{2|N \cdot L||N \cdot V|}{lerp(2|N \cdot L||N \cdot V|, |N \cdot L| + |N \cdot V|, \alpha)}$$

josta osoittaja peruuntuu täydessä BRDF:ssä. On olemassa myös muita tiheysfunktioita joita voidaan käyttää GGX:n sijaan kuten *Beckmann* mutta Godotissa on implementoituna vain GGX joten ne jäävät tämän työn ulkopuolelle.

4.6 Schlick GGX

Schlick GGX oli osana Burleyn BRDF:ää mutta koska Godot sisältää vain sen ja *toon* spekulaarit heijastukset, tulen käsittelemään sitä omassa kappaleessaan. Fresnel-yhtälöt määrittävät mikä osa valosta heijastuu mikrofaceteista joiden puolivektori(engl. *half vector*) on sama kuin pinnan normaalivektori. Yleisesti Fresnel-yhtälöiden mukaan mitä syvemmässä kulmassa valo osuu pintaan. sitä peilimäisemmin valo heijastuu. Fresnel-yhtälöiden laskeminen on kuitenkin erittäin haastavaa laskennallisesti, joten tyypillisesti pelimoottorit käyttävät approksimaatiota. Paperissa *An inexpensive BRDF model for physically based rendering* [12] Cristoph Schlick esitti seuraavan Cookin ja Torrancen työhön perustuvan approksimaation, joka tunnetaan Schlickin approksimaationa.

$$F_\lambda(u) = f_\lambda + (1 - f_\lambda)(1 - u)^5$$

Godot käyttää juuri tätä approksimaatiota yhtälöissään, eli Godotin lähdekoodissa.

```
1  vec3 F = f0 + (f90 - f0) * cLdotH5;
```

4.7 Blinn-Phong heijastusmalli

Phong valaistus oli yksi aikaisimmista valaistus malleista. Se koostuu Lambertin diffuusista ja Phong spekulaari -osasta. Eli sen yhtälö on

$$objektinvri * (ambient + vari_i * (N \cdot L_i) + vari_i * (R \cdot V)^{smoothness}),$$

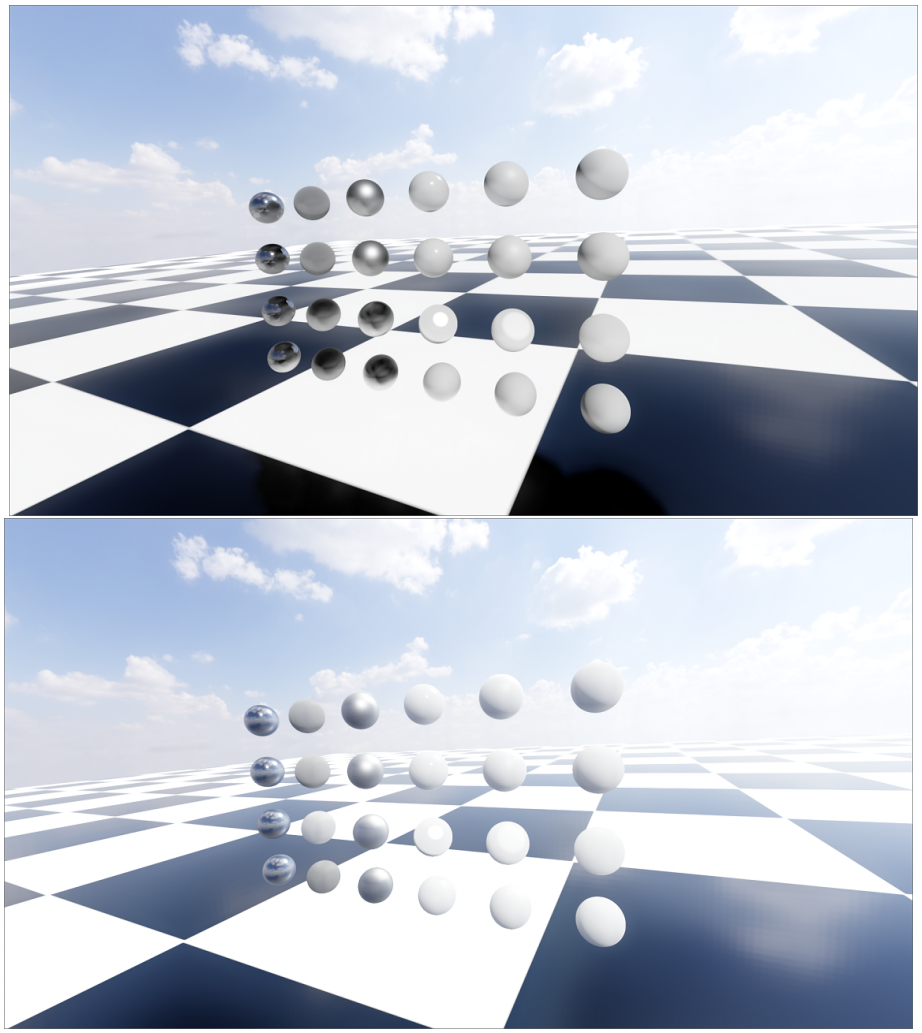
jossa R on heijastusvektori, L on valovektori ja V on katseluvektori. Spekulaarin kokoa voidaan säätää lisäämällä termiin smoothness eksponentti. Summaamalla yhteen jokaisen valon vaikutus saadaan kokonaisvalaistus laskettavassa fragmentissa. James F. Blinn kehitti menetelmää käyttämään heijastus vektorin sijaan valovektorin ja katseluvektorin puolikasvektoria $norm(V + L)$. Blinnin kaavassa katseluvektorin sijaan pistetulo otetaan puolikasvektorin ja normaalivektorin välillä. Eli sen kaava on

$$H = \frac{\langle V + L \rangle}{|V + L|}$$

$$albedo * (valonVari * max(\frac{\langle N \cdot L \rangle}{|N \cdot L|}, 0) + valonVari * spekulaarinVari * max(\frac{\langle N \cdot H \rangle}{|N \cdot H|}, 0)^{1-karkeus})$$

Jos normaali N ja puolikasvektori H ovat yhdensuuntaiset eli H:n ja N normalisoitu pistetulo on 1, se tarkoittaa, että valo heijastuu suoraan pinnan kautta katsojaan. yhtälön ambient termi on vakio, joka säätää pinnan kirkkautta. Blinn-Phong on yksi yksinkertaisimmista varjostus malleista ja oli käytössä openGL:n fixed function pipelinessa.

Kuten 4.2 kuvista huomaa, kuvassa jossa SDFGI ei ole käytössä ympäristön värit eivät heijastu materiaalin pinnasta vain skyboxin (laatikkomainen taivas tekstuuri.) kun taas SDFGI esimerkissä (ylempi) myös ympäristön objektit vaikuttavat esineen valaistukseen. Parhaiten tämä näkyy metallisissa objekteissa joissa SDFGI versiossa alla oleva shakkilautakuvio selvästi heijastuu palloista.



Kuva 4.2: Kuvassa eri BRDF:t eri karkeus arvoilla metalleille ja dielektrisille materiaaleille. karkeus 0,0.5,1 burley SlickGGX lambert SlickGGX toon toon ja burley ilman spekulaarisia heijastuksia.

5 Etumerkkinen etäisyyskenttä globaali valaistus -algoritmi

Luvussa käydään läpi Godot-pelimoottorin SDFGI-algoritmi. Ensin algoritmin kulkua käydään läpi yleisesti sen jälkeen esitellään SDF:t joita algoritmi käyttää säteenmarssituksessa askelkoon määrittämisessä. Sen jälkeen käsitellään Jump flood -algoritmia jolla Godot generoi SDFGI:n käyttämän etäisyyskentän jonka jälkeen käsitellään valokennoja. lopuksi käsitellään vokselikartiojäljitystä.

5.1 SDFGI-algoritmi

SDFGI on DDGI:hin(engl. *Dynamic Diffuse Global Illumination*)[13] perustuva menetelmä joka käyttää säteenjäljityksen sijaan säteenmarssitusta. SDFGI käyttää etumerkkistä etäisyysfunktiota. Se generoi 128^3 3D etäisyyskenttä tekstuurin, jonka se generoi käyttämällä jump flood algoritmia. Aluksi se alustaa $8 * 8 * 8$ koon ryhmän GPU-ytimiä joiden globaalia invokaatio ID:tä se käyttää käsiteltävän sijainnin globaalina sijaintina. [14][3]

SDFGI käyttää säteenjäljityksen sijaan käyttää sphere racingiä, joka on huomattavasti nopeampi, valaistuksen laskemiseen. Myös SDF ray-marching nimellä tunnettu pallonjäljitys on variaatio säteenmarssitusalgoritmista, jossa säteen askeleen koko määräytyy lähimmän geometrisen pinnan etäisyydestä säteen ampumapistees-

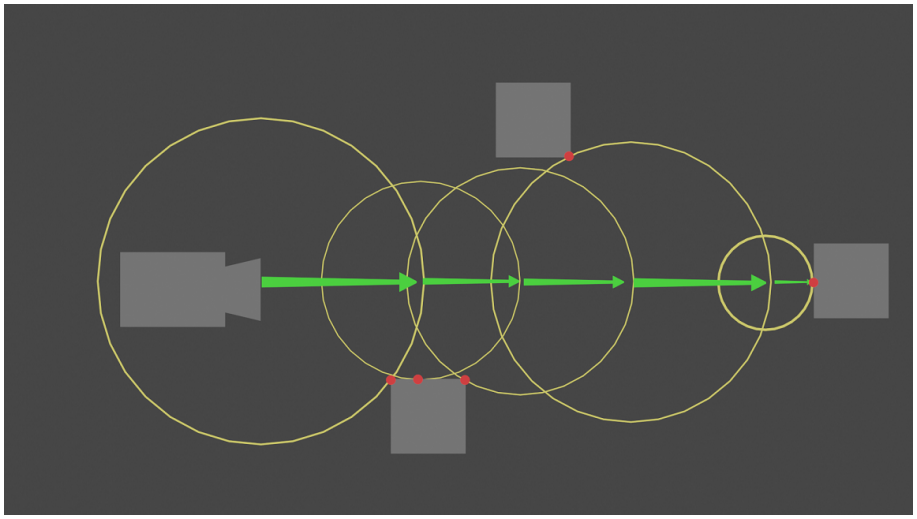
tä. Tätä toistetaan kunnes säde joko törmää pintaan tai menee käsiteltävän alueen ulkopuolelle.

Ohjelmalistaus 2 Tyyppiluokka 'palloseuranta'.

```
1  while (advance < max_advance) {
2          //read how much to advance from SDF
3          uvw = (pos + ray_dir * advance) * pos_to_uvw;
4
5          float distance = texture(sampler3D(sdf_cascades[j],
               linear_sampler), uvw).r * 255.0 - 1.0;
6          if (distance < 0.05) {
7                  //consider hit
8                  hit = true;
9                  break;
10         }
11
12         advance += distance;
13 }
```

Yllä olevasta koodi katkelmasta on jätetty pois valokennoihin ja cascaadeiden hallintaan liittyvä liittyvä koodi luettavuuden säilyttämiseksi. Näitä algoritmin osia tullaan käymään läpi omissa kappaleissa. Rivillä 3 algoritmi laskee säteen nykyisen sijainnin. Se käyttää sitä rivillä 5 3D tekstuurikoordinaattina ottaessaan näytteen 3D-tekstuurista johon on kirjoitettu aiemmassa kappaleessa kuvatulla tavalla lasketu kohtauksen etäisyyskenttä. Jos kentän etäisyys nykyisessä pisteessä on pienempi kuin 0.05 kynnysarvo, olemme törmänneet pintaan. Muussa tapauksessa jatkamme säteen marssimista kunnes säteen pituus ylittää uloimman kaskadin rajan.

Pallonjäljitys on huomattavasti nopeampi kuin säteen jäljitys, jos käytössä ei ole säteenjäljitykseen erikoistuneita ytimiä. Tämä johtuu siitä, että pallonjäljitys suorittaa säteen leikkaustestin käyttämällä kohtauksen SDF:ää. Tämän ansiosta se toimii myös vanhemmalla laitteistolla. Seuraavassa kuvassa demonstroidaan sphere tracing algoritmin etenemistä.



Kuva 5.1: kuvassa nähdään Sphere tracingin eteneminen. Säteen askeleet on merkitty vihreällä nuolella punaiset pisteet ovat säteen jokaisella askeleella lähimmät geometria pisteet. keltaiset ympyrät näyttävät puolestaan lähtöpisteen etäisyyden lähimmästä geometriasta. kuten kuvasta näkyy askeleen pituus on ympyrän säde.

5.2 Etumerkkiset etäisyysfunktiot

Tässä luvussa tarkastellaan etumerkkisiä etäisyysfunktioita (engl. *signed distance functions, SDF*). SDF:t ovat funktioita, jotka laskevat annetun pisteen etäisyyden lähimmästä pinnasta. Ne antavat positiivisen arvon, kun piste on pinnan ulkopuolella ja negatiivisen arvon kun piste on pinnan sisäpuolella. Alla olevassa kuvassa 5.2 on ympyrän SDF (Yhtälö alapuolella) pakattuna kuvan punaiseen kanavaan. Musta alue on ympyrän sisällä. Punainen ympyrän ulkopuolella

$$\sqrt{(x0 - x1)^2 + (y0 - y1)^2} - radius$$

SDF:ää käyttävät menetelmät ovat yksi keino saavuttaa globaali valaistus myös heikommilla GPU:illa jotka eivät tue laitteistolla kiihdytettyä säteenjäljitystä. Se on olennainen osa monia nykyisiä globaalin valaistuksen tekniikoita muunmuassa Godotin SDFGI:tä joka on yhtenä globaalin valaistuksen vaihtoehtona ja Unreal-pelimoottorin Lumenia, jossa se toimii varavaihtoehtona säteenjäljitykselle.

Kuva 5.2: Pallon SDF.

5.3 Jump flood -algoritmi

Jump flood algoritmi on menetelmä jolla pystymme generoimaan etäisyys kentän sille annetuista pisteistä. Siinä satunnaisesti valittujen seed pisteiden UV-koordinaatit levitetään kuvan jokaiseen pisteeseen niin, että jokainen tyhjä piste saa lähimmän siemen pisteen UV-koordinaatin arvokseen. Kaksi ulotteisella etäisyyskentällä algoritmi etenee seuraavasti. Tallennetaan lähtö pisteiden textuuri koordinaatit kuvan punainen ja vihreä kanaviin. lasketaan askelkoko joka saadaan kaavasta $AskelKoko = 2^{(maxIteraatio - nykyinenIteraatio)}$ Jokaista kohde textuurin pikseliä kohden, jos yhdenkään Askelkoon päässä olevan pikselin viittaaman pisteen etäisyys nykyisestä koordinaatista on pienempi kuin nykyisen pikselin etäisyys viittaamansa pisteen koordinaatista, aseta pikseli viittaamaan lähemmän pisteen koordinaattiin. tätä jatketaan puolittamalla askelkoko joka iteraation välillä kunnes askelkoko on yksi.

Jotta saamme generoitua kolme ulotteisen etäisyyskentän on käytettävä tavallisen textuurin sijasta 3D-tekstuuria ja kahdeksan viereisen pikselin sijaan tarkastelemme 26 viereistä pikseliä. [15]

Liitteessä A esitellään Godotin implementaatio jump flood -algoritmista. Lähdekoodi sisältää vielä optimisaatioita jumpfloodille mutta tulen käsittelemään vain yllä olevaa koodia sillä se on helppolukuisempi ja riittää etäisyyskentän laskemiseen. Ensimmäisellä 3-4 riveillä haetaan globaali invokaatio id:n eli $gl_WorkGroupID *$

$gl_WorkGroupSize + gl_LocalInvocationID$ joka antaa nykyisen säikeen globaalin sijainnin. Tätä tullaan käyttämään koordinaattina 3D-tekstuurille. Sen jälkeen se ladataan tekstuurista kyseistä koordinaattia vastaava RGBA-arvo. On tärkeää huomata, että tässä RGB arvot ovat xyz -koordinaatteja. Rivillä 9 algoritmi tarkistaa onko nykyinen koordinaatti jonkin lähtöpisteen päällä Jos näin on, voidaan lopettaa kyseisen koordinaatin tarkastelu sillä tiedetään että pisteen etäisyys itsestään on 0. Näin saadaan vältettyä turhaa laskentaa. Sen jälkeen riveillä 30-44 käydään for-silmukalla läpi jokainen offsetti ja verrataan käsiteltävän vokselin etäisyyttä rgb-arvoissa viitatun pisteeseen etäisyyteen jokaista offsetin päässä olevan vokselin rgb-arvoissa viitatun pisteen etäisyyteen nykyisen vokselin koordinaattiin, jos jokin etäisyys on lyhyempi vaihdetaan viittaus rgb-arvoissa lähimmän pisteen koordinaatteihin, ja jos ei säilytetään arvo ennallaan. Lopuksi rivillä 46 tallennetaan pikselin uusi arvo kuvaan.

Jotta algoritmi ei ylikirjoittaisi pikseliä ennen kuin toinen säe ehtii lukemaan sen arvon. Julkaisussa algoritmiin käytetään ping pong menetelmää eli on kaksi kuva puskuria (engl. *Image buffer*) joita vaihdetaan keskenään jokaisen iteraation jälkeen. Yllä olevassa koodissa nämä puskurit ovat $src_positions$, $dst_positions$.

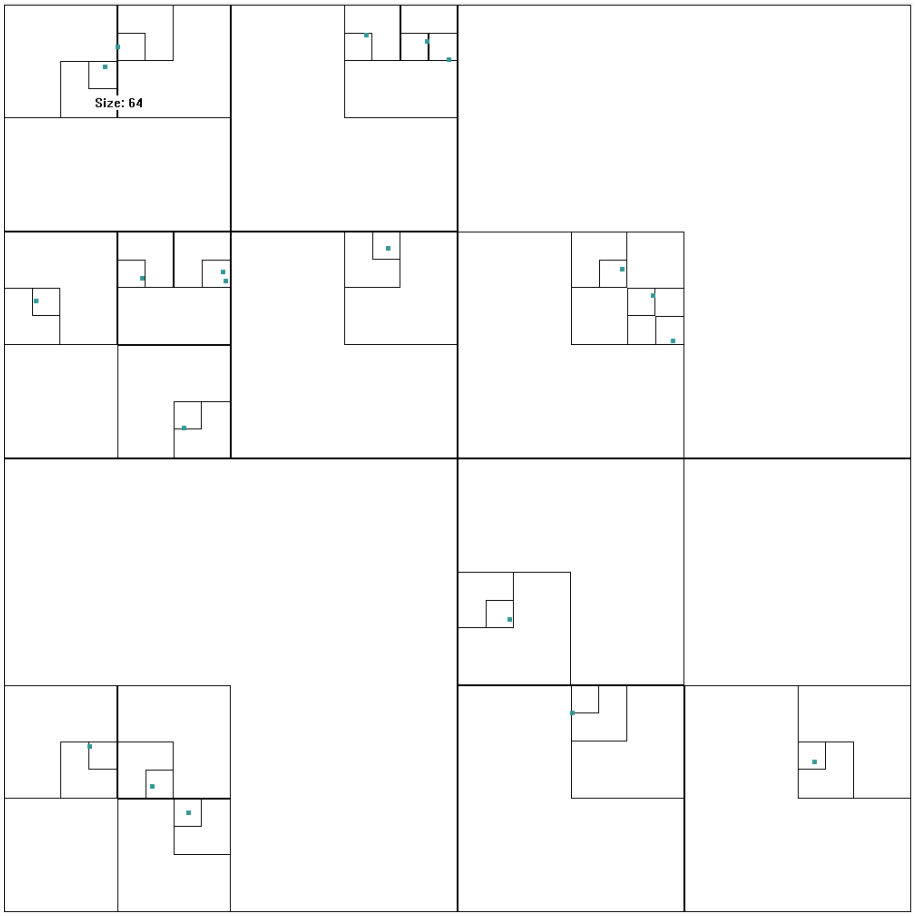
5.4 Valokennot

Valokennot (engl. *light probe*) [16] ovat valaistus menetelmä joka on nykyään käytössä lähes jokaisessa pelimoottorissa Godotin valaistus pohjautuu parannettuun versioon menetelmästä (Dynamic Diffuse Global Illumination with Ray-Traced Irradiance Fields 2019)[13]. Ongelmana valokennoissa on, että mikäli kenno on asetettu huonosti syntyy niin sanottua valovuotoa. esimerkiksi, jos anturi jää avautuvan oven taakse saattaa oveen ilmestyä musta varjo, sillä kun valaistusta laskiessa otetaan otantoja ympäröivistä valokennoista, jos joku kennoista on seinän toisella puolella valo vuotaa seinän läpi interpolaation vuoksi. Menetelmässä valokennojen

painoarvoon vaikuttavat kolmi eri tekijää. Varjostettavan pisteen etäisyys valokennosta, normalisoitu Lambert termi eli osoittaako pinnan normaali kohti valokennoa, ja kuinka suuri osa kohtauksen geometriasta varjostaa valokennoa. menetelmä laskee sen ampumalla säteitä valokennosta ympäriinsä ja rakentamalla tilatollisen mallin radiaalisesta etäisyydestä geometriaan. Varjostuksen aikana se käyttää Tšebyšovin epäyhtälöä okkluusion laskemiseen. Näiden painojen avulla metodissa interpoloidaan kahdeksan ympäröivän valokennon välillä. Menetelmä laskee lähellä olevan valaistuksen tarkemmin kuin kaukana olevan valaistuksen jakamalla alueen kaskadeihin ja päivittämällä kaukana olevan valaistuksen harvemmin kuin lähellä olevan valaistuksen.

5.5 Vokselikartiojäljitys

Godotin vokselikartiojäljitys (engl. *Voxel cone tracing*) perustuu dokumentaation mukaan Nvidean vuonna 2011 julkaisemaan paperiin, jossa Valaistava alue vokseloidaan 3D-tekstuuriin ja sen jälkeen jäljitetään ottamalla otantoja tekstuurin MIP-kartoista. Godotissa vokselikartiojäljitystä varten määrätään ensin kuutio voluumi jonka sisälle jaetaan valokennoja joiden valaistus lasketaan vokselikartiojäljityksellä. Nvidean julkaisun mukaan kohtauksesta rakennetaan SVO eli (Sparce voxel octree), SVO on rakenne jossa vokseleita jaetaan kahdeksaan alivokseliin kunnes saavutetaan maksimisyvyys tai kunnes vokselin sisältämän geometrian määrä alittaa jonkin mielivaltaisen alarajan. SVO:n rakennus tapahtuu paperin mukaan rasteroimalla mesh kaikissa kolmessa scenen akselissa niin että Viewportin resoluutio vastaa SVO:n maksimi syvyyttä. Vaikka dynaamiset ja staattiset objektit ovat samassa SVO:ssa, joka vain dynaamisten objektion arvot päivitetään jokaisen framin aikana sen sijaan, että SVO rakennettaisiin aina uudestaan. Jokainen vokseli esittää alempien tasojen valo käyttäytymistä tilastollisesti ja sisältää myös okluusio arvon joka esittää prosenttia säteistä jotka vokseli peittää. Epäsuoran valaistuksen arvioimiseksi SVO:n



Kuva 5.3: Kuvassa satunnaisista pisteistä generoitu quad tree.

lehtiin tallennetaan siihen saapuva radianssi. Radianssi myös propagoidaan puun ylempiin kerroksiin otantojen ottamistta varten. Vokselin esityksen matemaattiset yksityiskohdat jäävät tämän tutkielman ulkopuolelle. Kuvassa 5.3 nähdään miten quad tree rakentuu pisteiden ympärille. Quadtree on 2D-versio octreestä. Tilan jakaminen tapahtuu täysin samalla tavalla kolmiuloitteisessa tapauksessa jako tapahtuu kahdeksaan kuutioon neljän neliön sijaan. [17][18]

Kuten liitteen B koodiriviltä 19 nähdään, säteenmarssitusta jatketaan kunnes joko etäisyys ylittää säteen maksimipituuden, tai kun kertynyt okkluusio on 1. Rivillä 30 käytetään textureLOD-funktiota ottamaan otanta MIP:istä. [17] [3]

6 Säteenjäljitys

Yleisesti moderneissa pelimoottoreissa lasketaan ensin valaistus käyttämällä perinteisiä menetelmiä halpana pohjana säteenjäljitykselle. Säteenjäljitys (engl. *ray-racing*) on menetelmä jossa ammutaan säteitä avaruuteen, ja testataan törmäävätkö ne kohtauksen objekteihin. Suoran valaistuksen tapauksessa halutaan tietää heijastuuko kamerasta ammuttu säde suoraan valonlähdettä kohti. Tämä tehdään jokaiselle kameran pikselille erikseen. Säteenjäljitys ratkaisee myös monimutkaisempia valaistusongelmia, kuten heijastukset, varjot, diffuusivalaistus ja muita ominaisuuksia. Mutta nämä vaativat huomattavasti enemmän säteitä, ja näitä tullaan käsittelemään tarkemmin myöhemmin tässä luvussa.

Menetelmän ongelmana on sen laskennallinen vaativuus. Tämä johtuu siitä, että ammuttavien säteiden määrä on valtava. Lisäksi tilannetta mutkistaa se, että tarvittavien heijastusten määrä ei ole sama kaikille ammutuille säteille. Jotkin säteet vaativat enemmän ammuttuja säteitä, jotta pikselin realistinen valaistus pystytään laskemaan. Tämä vaikeuttaa taakan jakamista GPU:n säikeille sillä heijastunut säde on riippuvainen sitä edeltävästä säteestä.

Kun kirjoitetaan säteenjäljitysalgoritmia GPU:lle, ensimmäisenä saattaa tulla mieleen ajaa algoritmi *fragmenttivarjostin*. Tämä ei kuitenkaan ole paras vaihtoehto, ongelmana säteenjäljityksen ajamisessa *fragmenttivarjostimessa* on, että jotkin GPU:t kaatavat ohjelman mikäli GPU:n säikeen suorittaminen kestää liian pitkään. Niinpä on hyödyllistä ensin renderöidä säteenjäljitys erilliseen tekstuuriin GPU:n

muistissa käyttämällä compute varjostinta ja sen jälkeen renderöimällä lasketun tekstuurin fragmenttivarjostimessa. Näin computevarjostimessa työtaakka pystytään jakamaan pienempiin GPU:n työryhmiin. Säde voidaan kuvata kahden vektorin avulla, toinen osoittaa säteen lähtöpisteen avaruudessa ja toinen on yksikkövektori joka osoittaa säteen suunnan, jotta pystytään testaamaan törmääkö säde polygonimallin kanssa, on testattava törmääkö säde yhteenkään polygonimallin kolmioon. Kuten voi kuvitella tämä on äärimmäisen raskasta kokonaiselle kohtaukselle. Niinpä jotta säteenjäljitys saadaan toimimaan reaaliajassa, tarvitaan omistetun laitteiston lisäksi kiihdytysrakenne kuten BVH-puu (engl. *Bounding Volume Hierarchy*). Joka jakaa kaikki objektit hierarkisiin alueisiin. Säteen jäljetyksessä tätä voidaan hyödyntää vähentämällä leikkaustestien määrää sillä mikäli säde ei törmää voluumiin voidaan ohittaa leikkaustestit kaikkiin sen lapsiin hierarkiassa.

6.1 Polunseuranta

Polunseuranta (engl. *Pathtracing*) on menetelmä jossa kohtaus renderöidään kokonaan käyttämällä säteenjäljitystä renderöintiyhtälön numeeriseen ratkaisuun. Menetelmää kutsutaan myös Monte Carlo metodiksi. Käytännössä siinä ammutaan paljon säteitä säteen törmäyspisteestä satunnaisiin suuntiin, tai toisin sanoen otetaan otoksia funktiosta(engl. *sampling the function*), ja lasketaan niiden keskiarvo. Kun ollaan otettu tarpeeksi monen otannon keskiarvo approksimaatio lähestyy integraalin todellista arvoa. Eli

$$\int_A p(x)G(x)V(x)dA_x \approx \frac{1}{N}\sum V(x_i)G(x_i)L(x_i)f_r(x_i)$$

Tätä voidaan vielä nopeuttaa otannan harkinnanvaraisella kohdentamisella (engl. *importance sampling*). Eli ottamalla enemmän otantoja funktiosta arvoilla joilla funktion arvo on suuri. Jonka jälkeen otetaan painotettu keskiarvon funktion ar-

voista niin, että kohdilla joissa funktion arvo on matala on korkeampi paino kuin kohdilla joissa funktion arvo on suuri. Tämä muuttaa yhtälöä seuraavalla tavalla.

$$\int_A p(x)G(x)V(x)dA_x \approx \frac{1}{N}\sum \frac{V(x_i)G(x_i)L(x_i)f_r(x_i)}{p(x_i)}$$

jossa $p(x_i)$ on todennäköisyys, että näyte x_i valitaan. Tällä kompensoidaan sitä, että otantoja on enemmän kohdista joissa funktion arvo on suuri.

6.2 ReSTIR

Kuten edellisessä luvussa mainittiin näytteiden määrää voidaan vähentää ottamalla enemmän otantoja arvoilla joilla funktion arvo on suuri. Käytännössä funktion korkeiden ja matalien kohtien painotettu otantojen otto on kuitenkin haastavaa renderöintiyhtälön monimutkaisuuden vuoksi. ReSTIR on menetelmä suoran valaistuksen laskemiseksi, mutta se saadaan yleistettyä myös epäsuoraan valaistukseen.

ReSTIR lähestyy renderöintiyhtälöä eri tavalla aiemmassa kappaleessa esitellystä mallista. Sen sijaan, että se tutkisi jokaisesta suunnasta tulevaa valoa, se laskee valaistuksen pisteessä laskemalla renderöintiyhtälön integraalin kaikkien valonlähteiden pintojen A yli. Tämän laskemiseen renderöintiyhtälö muuttuu muotoon $\int_A L(x)p(x)G(x)V(x)dA_x$, jossa p(x) on BSDF. G on geometria-termi jaettuna valonlähteen etäisyyden neliöllä. Kun säteitä ammutaan hemisphäärissä käänteisen neliön laki sisältyy funktioon mutta kun integraali lasketaan valojen pinnan yli se täytyy ottaa huomioon eksplisiittisesti. Viimeinen termi on näkyvyys. Joka saa arvon 0 mikäli ei ole suoraa linjaa laskettavasta pisteestä valonlähteeseen. ReSTIR generoi ensin huonon jakauman otantoja esimerkiksi tasaisesti jokaisesta kohdasta. Sen jälkeen lasketaan alkuperäisen tiheysfunktion arvo valitussa pisteessä. Seuraavaksi valitaan painojen suhteen mukaan arvo y, joka sijoitetaan arvioitavaan funktioon ja annetaan sille painoksi huonojen näytteiden keskiarvo jaettuna y:n arvolla. Alku-

peräisessä julkaisussa käytettiin RIS (engl. *resampled Importance Sampling*) menetelmää. RIS mahdollistaa sen, että numeerinen integraatio voidaan jakaa niin, että jokainen termi voidaan arvioida eri näytteenottotaajuudella (engl. *engl. Sampling rate*). Tämä on hyödyllistä sillä voidaan käyttää halvemmin laskettavia termejä parantamaan kalliimpien näytteiden laatua. Harkinnanvaraisen kohdentamisesta saadaan renderöinti yhtälölle kaava

$$\int f_r(x)L(x)G(x)V(x)dx \approx \frac{1}{N}\sum[\frac{V(x_i)G(x_i)L(x_i)f_r(x_i)}{\int p(x_i)}]$$

Alla renderöinti yhtälö tullaan lyhentämään muotoon

$$\int f_r(x)L(x)G(x)V(x)dx = \int f(x)$$

selvyyden vuoksi. Eli saadaan $\int f(x) \approx \frac{1}{N}\sum\frac{f(x)}{p(x)}$. Jossa $p(x) = \frac{f(x)}{\int f(x)dx}$ Se voidaan jakaa edelleen muotoon.

$$\int f(x) \approx \frac{1}{N}\sum[\frac{f(x_i)}{p_0(x_i)}\frac{1}{M}\sum\frac{p_0(x_j)}{p(x_j)}]$$

approksimoimalla p(x) ja siitä yhä edelleen samalla tavalla. Kuten yhtälöstä näkee jokaisessa jaossa esitellään uusi termi, joilla voidaan lisätä tietoa yhtälöön. [19]

Esimerkiksi julkaisussa Spatiotemporal reservoir resampling for real-time ray tracing with dynamic direct lighting 2020 todetaan *Weighted resevoir sampling* avulla saadaan aiempien painojen summa ja uuden näytteen paino, joilla saadaan laskettua RIS. Koska *weighted resevoir samplingin* näytteitä voidaan yhdistää, voidaan aggregoida läheisten pikselien arvoja ja ajallisesti läheisten pikselien arvoja(edellisiä framejä) (engl. *temporal and spacial reuse*[20]). Näin saadaan lisättyä tosiasiallisten näytteiden määrää huomattavasti. [21]

Alkuperäinen ReSTIR laskee vain suoraa valaistusta mutta julkaisussa ReSTIR

GI: Path Resampling for Real-Time Path Tracing demonstroitiin miten menetelmä saadaan yleistettyä epäsuoraan valaistukseen. Menetelmän ongelmana on, että Talbotin alkuperäinen RIS olettaa, että otannat ovat toisistaan itsenäiset. Koska ReSTIR uudelleenkäyttää otantoja näytteet eivät ole itsenäisiä joten ei ole takuuta, että ReSTIR konvergoisi oikeaan tulokseen. [20] Julkaisussa [22] kirjoittajat esittelevät GRIS:n(*Generalized Resampled Importance Sampling*), joka parantaa RIS sallimalla korreloitujen näytteiden käytön.

Julkaisussa *Area ReSTIR:Resampling for Real-Time Defocus and Antialiasing 2024* [23], ReSTIR-menetelmää parannettiin laajentamalla integraation laskenta-aluetta 4D-säteavaruuteen jossa on mukana filmin ja linssin 2D-alueet. Tekemällä näin, he onnistuivat vähentämään kohinan määrää kuvassa erityisesti korkeataajuuksisissa kuvan osissa kuten hiuksissa.

6.3 BSDF

Uusimmat Unreal-pelimoottorin versiot mahdollistavat BSDF:n (engl. *Bidirectional Scattering Distribution Function.*) Tämän työn kirjoitushetkellä se on vielä kokeellinen ominaisuus, mutta tulevaisuudessa se saattaa korvata vanhaa BRDF käyttävän materiaalisysteemin. BSDF on BRDF:n laajennus, joka koostuu kahdesta termistä: materiaalin BRDF ja BTDF (engl. *Bidirectional Transmission Distribution Function*). BTDF-termi kuvaa sitä kuinka suuri osa valosta läpäisee materiaalin. Brent Burley [24] laajensi Disneyn BRDF:n BSDF:ksi. BSDF on jo käytössä offline renderöijissä kuten Blenderin Cyclesissä. Blenderin implementaatio perustuu OpenPBR-standardiin. [25] Kehityksestä päätellen BSDF tulee varmaankin tulevaisuudessa myös yleistymään reaaliaikaisissa renderöijissä.

6.4 HDDAGI

HDDAGI (*engl.* Hierarchical digital differential analyser) on Godotissa SDFGI:in tulevaisuudessa korvaava valaistus menetelmä. Se perustuu paperissa [26] esitettyyn tekniikkaan, jota on sovellettuna DDGI:hin. SDFGI suoritus kyky kärsii kun kamera liikkuu paljon, sillä SDF joudutaan generoimaan uudestaan. Sitä vastoin kameran liikkuminen ei vaikuta HDDGI:hin.

7 Yhteenveto

Tutkielmassa käytiin läpi eri globaalin valaistuksen menetelmiä erityisesti Godot-pelimoottorin näkökulmasta. Globaali valaistus on aktiivinen tutkimusala ja Godot käyttää vain pientä osaa olemassa olevista menetelmistä. Globaalin valaistuksen ja tietokonegrafiikan valaistuksen taustalla yleisestikin on renderöintiyhtälö josta kerrottiin tarkemmin luvussa 2. Se koostuu BRDF:stä, valo-termistä ja Lambertin termistä. BRDF jaotellaan usein diffuusiin ja spekulaariin osaan ja sitä kuvataan tarkemmin tutkielman luvussa 4 Yksinkertaisin BRDF on Lambertin diffuusi, jossa otetaan pistetulo valon suunnan ja pinnan normaalin välillä. Godotissa käytännössä valitaan valikosta käytettävä spekulaari- ja diffuusi- termi ja pelimoottori luo halutut varjostimet esiprosessoreilla (engl. *preprocessors*). Godotissa on käytettävissä diffuuseista Lambert, Burley, Lambert Wrap, ja toon ja spekulaarisia schlickGGX, toon, ja näiden lisäksi on myös mahdollista kirjoittaa omia varjostimia.

Godotin tällä hetkellä käyttämät globaalin valaistuksen tekniikat ovat SDFGI ja Voxel GI. Näistä SDFGI pohjautuu DDGI:hin ja jota kuvaillaan tarkemmin luvussa 5.1, yksinkertaistettuna se generoi SDF-esityksen geometriasta ja käyttää pallojäljitystä törmäyspisteiden löytämiseen. Hierarkinen digitaalinen diffrentiaali analysoija globaali valaistus (engl. *hierarchical digital diffrential analyser Global Illumination*, HDDGI) tulee korvaamaan SDFGI:n tulevaisuudessa. Voxel GI on toinen Godotin valaistusmenetelmä

Uutena kehityksenä valaistuksen alalla ovat radianssi kaskaadit (engl. *radience*

cascades), jotka jäävät tämän tutkielman ulkopuolelle, ja area ReStir, jota kuvataan tarkemmin luvussa 6. Koska halutaan esittää enemmän ja enemmän erillaisia materiaaleja käyttämällä samaa materiaalimallia ja koska GPU:den laskutehon lisääntyy koko ajan. Tulevaisuudessa todennäköisesti myös muutkin pelimoottorit siirtyvät käyttämään BSDF:iä. HDDAGI tulee korvaamaan SDFGI:n tulevaisuudessa mutta sen tarkempi kuvaus jää työn ulkopuolelle.

Koska säteenjäljitys pystyy mallintamaan valon ilmiöitä parhaiten nykyisistä tekniikoista ja GPU:den tehot alkavat hitaasti lähestymään tasoa jolla reaaliaikainen säteenjäljitys on on mahdollista. Säteenjäljityksen osa pelimoottoreiden valaistus laskuissa tulee todennäköisesti kasvamaan entisestään.

Koska AO:n laskeminen vaatii otantoja hemisfäärissä, samalla tavalla kuin säteenjäljityksessä. SSAO vaatii yhä liikaa otantoja siihen että kuvassa ei olisi lainkaan kohinaa vaatien pelin kehittäjiltä luovia ratkaisuja, kuten Persona 5 värin koostamiskuvion (engl. *Dither pattern*) AO:n kohinan piilottamiseen. Se voisi hyötya samanlaisesta temporaalisesta ja avaruudellisesta aggregoinnista kuin ReStir. Aemmassa tutkimuksessa ollaan jo kehitetty TSSAO -menetelmä (engl. *(temporal screen space ambient occlusion*) joka aggregoi temporaalista dataa. Mutta mahdollisesti lisäksi voitaisiin aggregoida avaruudellista dataa. Sillä lähellä olevat pikselit ovat todennäköisesti valaistu samalla tavalla. Tämä työ käsitteli Vain rajattua joukkoa globaalin valaistuksen menetelmiä. Muummuassa GI 1.0, virtuaaliset valot (engl. *Virtual lights*), *Metropolis* ja monet muut rajattiin työn ulkopuolelle.

Lähdeluettelo

[1] M. Pharr, G. Humphreys ja W. Jakob, ”Physically Based Rendering: From Theory to Implementation”, teoksessa Morgan Kaufmann, 2022, luku 14.4 The Light Transport Equation. url: https://pbr-book.org/3ed-2018/Light_Transport_I_Surface_Reflection/The_Light_Transport_Equation (viitattu 11. 12. 2024).

[2] H. Vogel, ”A better way to construct the sunflower head”, *Mathematical Biosciences*, vol. 44, nro 3, s. 179–189, 1979, ISSN: 0025-5564. DOI: 10.1016/0025-5564(79)90080-4.

[3] Godot Community, *Godot Engine – Multi-platform 2D and 3D game engine*, Version 4.2.2-stable, 2024. url: https://github.com/godotengine/godot/releases/tag/4.2.2-stable (viitattu 11. 12. 2024).

[4] J. Amanatides ja A. Woo, ”A Fast Voxel Traversal Algorithm for Ray Tracing”, *Dept. of Computer Science, University of Toronto*, 1987.

[5] J. T. Kajiya, ”The rendering equation”, teoksessa *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, sarja SIGGRAPH ’86, New York, NY, USA: Association for Computing Machinery, 1986, s. 143–150, ISBN: 0897911962. DOI: 10.1145/15922.15902.

[6] Y. O’Donnell. ”Precomputed Global Illumination in Frostbite”. (2018), url: https://www.ea.com/frostbite/news/precomputed-global-illumination-in-frostbite (viitattu 31. 05. 2024).

[7] P.-P. Sloan, ”Stupid Spherical Harmonics (SH) Tricks”, *Game Developers Conference (GDC) Companion*, 2008. url: http://www.ppsloan.org/publications (viitattu 11. 12. 2024).

[8] P.-P. Sloan, J. Kautz ja J. Snyder, ”Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments”, *ACM Trans. Graph.*, vol. 21, nro 3, s. 527–536, heinäkuu 2002, ISSN: 0730-0301. DOI: 10.1145/566654.566612.

[9] R. Green, ”Spherical Harmonic Lighting: The Gritty Details”, teoksessa *Game Developers Conference (GDC)*, 2003. url: https://gdcvault.com/play/1022720/Spherical-Harmonic-Lighting-The-Gritty (viitattu 11. 12. 2024).

[10] B. Burley ja Walt Disney Animation Studios, ”Physically-based shading at Disney”, teoksessa *ACM SIGGRAPH*, vol. 2012, vol. 2012, 2012, s. 1–7.

[11] J. Earl Hammon, ”PBR Diffuse Lighting for GGX+Smith Microsurfaces”, teoksessa *Game Developers Conference (GDC)*, 2017. url: https://www.gdcvault.com/play/1024478/PBR-Diffuse-Lighting-for-GGX (viitattu 11. 12. 2024).

[12] C. Schlick, ”An Inexpensive BRDF Model for Physically-based Rendering”, *Computer Graphics Forum*, vol. 13, nro 3, s. 233–246, elokuu 1994, ISSN: 1467-8659. DOI: 10.1111/1467-8659.1330233.

[13] Z. Majercik, J.-P. Guertin, D. Nowrouzezahrai ja M. McGuire, ”Dynamic Diffuse Global Illumination with Ray-Traced Irradiance Fields”, *Journal of Computer Graphics Techniques (JCGT)*, vol. 8, nro 2, s. 1–30, kesäkuu 2019, ISSN: 2331-7418. url: http://jcgt.org/published/0008/02/01/.

[14] J. Linietsky, *SDFGI - Solving the accessible Global Illumination problem in Godot*, Presented during a discussion with Godot maintainers, 2022. url: https://www.docdroid.net/YNntL0e/godot-sdfgi-pdf (viitattu 11. 12. 2024).

[15] G. Rong ja T.-S. Tan, "Jump Flooding in GPU with Applications to Voronoi Diagram and Distance Transform", *School of Computing, National University of Singapore*, 2005. url: http://www.comp.nus.edu.sg/~tants/jfa.html (viitattu 11. 12. 2024).

[16] G. Greger, P. Shirley, P. Hubbard ja D. Greenberg, "The Irradiance Volume", *Computer Graphics and Applications, IEEE*, vol. 18, s. 32–43, huhtikuu 1998. DOI: 10.1109/38.656788.

[17] C. Crassin, F. Neyret, M. Sainz, S. Green ja E. Eisemann, "Interactive indirect illumination using voxel cone tracing: a preview", teoksessa *Symposium on Interactive 3D Graphics and Games*, sarja I3D '11, San Francisco, California: Association for Computing Machinery, 2011, s. 207, ISBN: 9781450305655. DOI: 10.1145/1944745.1944787.

[18] Godot Engine, *Godot 3 Renderer Design Explained*, 2024. url: https://github.com/godotengine/godot-website/blob/76f350b62cdc21cc1ea3876574d12946b2bb42b7/collections/_article/godot-3-renderer-design-explained.md?plain=1#L216 (viitattu 11. 12. 2024).

[19] J. F. Talbot, *Importance resampling for global illumination*. Brigham Young University, 2005.

[20] Y. Ouyang, S. Liu, M. Kettunen, M. Pharr ja J. Pantaleoni, "ReSTIR GI: Path Resampling for Real-Time Path Tracing", *Computer Graphics Forum*, vol. 40, nro 8, s. 17–29, 2021. DOI: 10.1111/cgf.14378.

[21] B. Bitterli, C. Wyman, M. Pharr, P. Shirley, A. Lefohn ja W. Jarosz, "Spatiotemporal reservoir resampling for real-time ray tracing with dynamic direct lighting", *ACM Trans. Graph.*, vol. 39, nro 4, elokuu 2020, ISSN: 0730-0301. DOI: 10.1145/3386569.3392481.

[22] D. Lin, M. Kettunen, B. Bitterli, J. Pantaleoni, C. Yuksel ja C. Wyman, "Generalized Resampled Importance Sampling: Foundations of ReSTIR", vol. 41, nro 2, 75:1–75:23, elokuu 2022. DOI: 10.1145/3528223.3530158.

[23] S. Zhang, D. Lin, M. Kettunen, C. Yuksel ja C. Wyman, "Area ReSTIR: Resampling for Real-Time Defocus and Antialiasing", vol. 43, nro 4, heinäkuu 2024. DOI: 10.1145/3658210.

[24] B. Burley, "Extending the Disney BRDF to a BSDF with Integrated Subsurface Scattering", sarja SIGGRAPH '15: ACM SIGGRAPH 2015 Courses, Saatavilla https://blog.selfshadow.com/publications/s2015-shading-course/#course_content, ACM, heinäkuu 2015, ISBN: 9781450336345. DOI: 10.1145/2776880. (viitattu 11. 12. 2024).

[25] Z. Andersson, P. Edmondson, J. Guertault et al., "OpenPBR Surface Specification", Academy Software Foundation (ASWF), tekninen raportti, 2024. url: https://academysoftwarefoundation.github.io/OpenPBR/ (viitattu 11. 12. 2024).

[26] K. Museth, "Hierarchical digital differential analyzer for efficient ray-marching in OpenVDB", teoksessa *ACM SIGGRAPH 2014 Talks*, sarja SIGGRAPH '14, Vancouver, Canada: Association for Computing Machinery, 2014, ISBN: 9781450329606. DOI: 10.1145/2614106.2614136.

Liite A Jump flood -iteraatio

Ohjelmalistaus 3 Tyyppiluokka 'Jump flood -iteraatio'. [3]

```
#ifdef MODE_JUMPFLOOD

        //regular jumpflood, efficient for large steps, inefficient for small steps
        ivec3 pos = ivec3(gl_GlobalInvocationID.xyz);

        vec3 posf = vec3(pos);

        if (params.half_size) {
                posf = posf * 2.0 + 0.5;
        }

        uvec4 p = imageLoad(src_positions, pos);

        if (!params.half_size && p == uvec4(uvec3(pos), 255)) {
                imageStore(dst_positions, pos, p);
                return; //points to itself and valid, nothing better can be done, just pass
        }

        float p_dist;

        if (p.w != 0) {
                p_dist = distance(posf, vec3(p.xyz));
        } else {
                p_dist = 0.0; //should not matter
        }

        const uint offset_count = 26;
        const ivec3 offsets[offset_count] = ivec3[](offsetit vec 3 muodossa esim. vec3(1,1,1));

        for (uint i = 0; i < offset_count; i++) {
                ivec3 ofs = pos + offsets[i] * params.step_size;
                if (any(lessThan(ofs, ivec3(0))) || any(greaterThanEqual(ofs, ivec3(params.grid_size)))) {
                        continue;
                }
                uvec4 q = imageLoad(src_positions, ofs);

                if (q.w == 0) {
                        continue; //was not initialized yet, ignore
                }
                float q_dist = distance(posf, vec3(q.xyz));
                if (p.w == 0 || q_dist < p_dist) {
                        p = q; //just replace because current is unused
                        p_dist = q_dist;
                }
        }
        imageStore(dst_positions, pos, p);
#endif
```

Liite B Voxel cone tracing

Ohjelmalistaus 4 Tyyppiluokka 'Voxel Cone tracing'. [3]

```
1       vec3 cone_dirs[MAX_CONE_DIRS] = vec3[](
2                               vec3(0.0, 0.0, 1.0),
3                               vec3(0.866025, 0.0, 0.5),
4                               vec3(0.267617, 0.823639, 0.5),
5                               vec3(-0.700629, 0.509037, 0.5),
6                               vec3(-0.700629, -0.509037, 0.5),
7                               vec3(0.267617, -0.823639, 0.5));
8
9               float cone_weights[MAX_CONE_DIRS] = float[](0.25, 0.15, 0.15, 0.15, 0.15, 0.15);
10              float tan_half_angle = 0.577;
11
12              for (int i = 0; i < MAX_CONE_DIRS; i++) {
13                      vec3 direction = normal_mat * cone_dirs[i];
14                      vec4 color = vec4(0.0);
15                      {
16                              float dist = 1.5;
17                              float max_distance = length(vec3(params.limits));
18                              vec3 cell_size = 1.0 / vec3(params.limits);
19                              while (dist < max_distance && color.a < 0.95) {
20                                      float diameter = max(1.0, 2.0 * tan_half_angle * dist);
21                                      vec3 uvw_pos = (pos + dist * direction) * cell_size;
22                                      float half_diameter = diameter * 0.5;
23                                      //check if outside, then break
24                                      //if ( any(greaterThan(abs(uvw_pos - 0.5),vec3(0.5f + half_diameter * cell_size))
                                                ) ) {
25                                      // break;
26                                      //}
27
28                                      float log2_diameter = log2(diameter);
29                                      vec4 scolor = textureLod(sampler3D(color_texture, texture_sampler), uvw_pos,
                                                log2_diameter);
30                                      float a = (1.0 - color.a);
31                                      color += a * scolor;
32                                      dist += half_diameter;
33                              }
34                      }
35                      color *= cone_weights[i] * vec4(albedo.rgb, 1.0) * params.dynamic_range; //restore range
36                      accum += color.rgb;
37              }
38      }
39
40      outputs.data[cell_index] = vec4(accum, 0.0);
```