Juha Räsänen

Säteenmarssinta musiikin visualisoinnissa

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikan tutkinto-ohjelma

Insinöörityö

28.11.2022

Tiivistelmä

Tekijä:	Juha Räsänen
Otsikko:	Säteenmarssinta musiikin visualisoinnissa
Sivumäärä:	37
Aika:	28.11.2022
Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Tieto- ja viestintätekniikka
Ammatillinen pääaine:	Pelisovellukset
Ohjaaja:	Lehtori Miikka Mäki-Uuro

Insinöörityön tarkoituksena oli tutkia grafiikan tuottamista musiikin visualisointiin hyödyntämällä säteenmarssintaa sekä etsiä säteenmarssinnan rajoitteita grafiikan tuottamisessa. Työssä perehdyttiin eri konseptien ja käytännöllisten menetelmien hyödyntämiseen säteenmarssintaa käytettäessä.

Insinöörityöprojektissa tehtiin erilaisia näkymiä OpenGL-projektiin, johon käyttäjä pystyi syöttämään musiikkiraidan, jota lukemalla sovellus muutti grafiikkaa musiikin eri taajuuksien tahtiin reaaliajassa. Toteutuksessa oli pohjalla aikaisempi OpenGL-projekti, johon halutut 3D-objektit pystyttiin lataamaan muistiin ja asettamaan näkymään. Näkymän pääkamera ja varjostimien asetus objekteihin olivat valmiina ominaisuuksina OpenGL-projektissa. Työssä pisin aika kului kirjoitettaessa fragment-varjostimia ja muokattaessa projektia, jotta se veisi musiikkiraidasta tietoa fragment-varjostimeen säteenmarssintaa varten.

Insinöörityöstä voidaan hahmottaa, minkälaisia ominaisuuksia säteenmarssinalla on musiikin visualisoinnissa, kuten etäisyyskenttien muotojen muokkaaminen musiikin tahtiin, myös, minkälaisia rajoituksia ja haastavia tekijöitä pitää ottaa huomioon kuten suorituskyvyn laskeminen monien etäisyyskenttien samanaikaisessa piirtämisessä.

Avainsanat: musiikki, visualisointi, säteenmarssinta, grafiikka

Abstract

Author:	Juha Räsänen
Title:	Raymarching in music visualization
Number of Pages:	37
Date:	28 November 2022
Degree:	Bachelor of Engineering
Degree Programme:	Information and Communications Technology
Professional Major:	Game Applications
Supervisor:	Miikka Mäki-Uuro, Senior Lecturer

The purpose of the thesis was to investigate the production of graphics for the visualization of music by utilizing raymarching and to study the limitations of raymarching in the production of graphics. The work introduced the utilization of different concepts and practical methods when using raymarching.

In the thesis, various visuals were made for the OpenGL project, into which the user could enter a music track. By reading that track the application changed the graphics based on the rhythm of the different frequencies of the music in real time. The implementation was based on a former OpenGL project in which the desired 3D objects could be loaded into memory and displayed. The main camera of the scene and the loading of shaders for the objects were ready-made features in the OpenGL project. Most of the time for the thesis project was spent writing fragment shader and editing the project so that it would import information from the music track to the fragment shader for ray marching.

Keywords: Music, Visualization, Raymarching, Graphics

Sisällys

1 Johdanto 1

2 Musiikin visualisointi 2

3 Säteenmarssinta 3

4 FFT-algoritmi 6

5 Fragment-varjostin 6

6 Uv-koordinaatit 8

7 Etäisyyskentät (SDF) 9

8 Säteenmarssinnan menetelmiä 12

8.1 Etäisyyskenttien muotojen muuttaminen 12
8.2 Värit ja materiaalit 16
8.3 Valolähteet ja varjot 19
8.4 Sumentaminen ja liike-epäterävyys 20

9 Säteenmarssinta musiikin visualisoinnissa toteutus 22

9.1 Säteenmarssinnan toteutus 24
9.2 Musiikin lukeminen ja tuominen varjostimeen 26
9.3 Etäisyyskentän luominen ja taajuusarvojen käyttäminen etäisyyskentän muodon muokkaamiseen 27

10 Projektin toteutuksessa esiintyneet ongelmat 28

10.1 Säteenmarssintaan tutustuminen 28
10.2 Kahdelle akselille muuttaminen 31

11 Yhteenveto 33

Lähteet 35

Lyhenteet

glsl: OpenGL Shading Language. C-kielen tapainen varjostinkieli, jota käytetään OpenGL:ssä.

FFT: Fast Fourier transform. FFT:llä muutetaan signaali muotoon, jolla taajuudet saadaan eriteltyä toisistaan.

hlsl: High-level shader language. C-kielen tapainen varjostinkieli, jota voidaan käyttää DirectX:ssä.

SDF: Signed distance field. Funktio, jolla esitetään etäisyyskenttä.

uv-kartta: Keino, jolla saadaan säteenmarssinnan säteille asetettua fragmentit oikeisiin ikkunan koordinaatteihin.

1 Johdanto

Insinöörityössä perehdytään siihen, mitä säteenmarssinta on ja miten sitä voidaan hyödyntää graafisessa visualisoinnissa. Lisäksi työssä käydään läpi toteutustapa ja menetelmiä, joiden avulla säteenmarssinnasta saadaan enemmän hyötyä.

Säteenmarssinta on tekniikka, jonka avulla voidaan renderöidä tietokonegrafiikkaa käyttäen etäisyyskenttiä. Tekniikkaa on harvoin käytetty peleissä, elokuvissa ja mediassa yleensä. Insinöörityössä luetaan musiikkiraitaa, jonka datalla tehdään muutoksia etäisyyskenttiin taajuuksien amplitudien mukaan eli muokataan objektin muotoa reaaliajassa. Tietoa tekniikasta on ollut ainakin vuodesta 1989 Sandinin, Hartin ja Kauffmanin kirjoittamasta tutkielmasta päätellen [1].

Itse musiikkia käytetään mm. ohjelmien ja elokuvien tunnelman luomiseen. Musiikilla voidaan vahvistaa kohtauksen katselemisesta aiheutuvia tunteita. Kuuroilla ja huonokuuloisilla henkilöillä musiikin kuuntelun esteet vähentävät kuitenkin mahdollisuuksia päästä kokemaan musiikkikokemuksia, koska musiikki ei ole tälle väestölle sopivassa muodossa, joten musiikkia visualisoidaan [2].

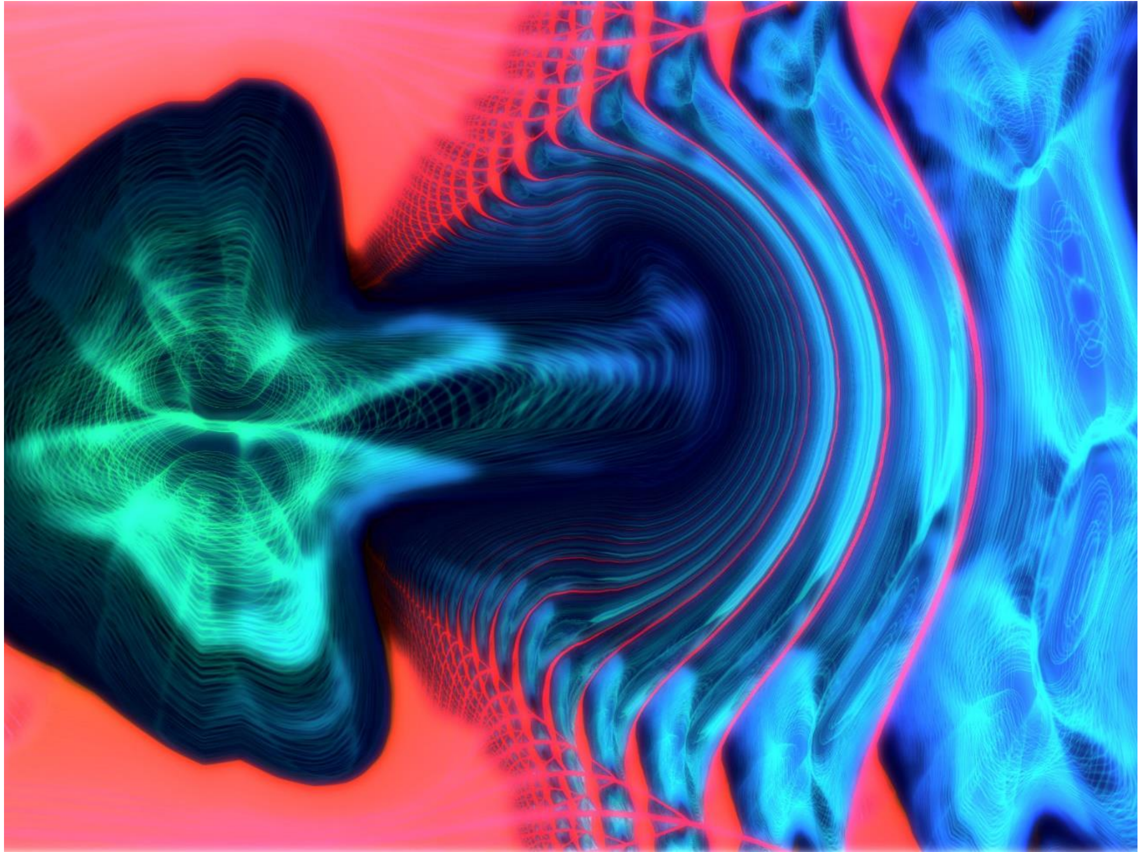
Luvussa 2 käsitellään, mitä musiikin visualisointi on, käymällä läpi variaatioita toimintaperiaatteesta ja ohjelmistojen historiasta. Sen jälkeen luvussa 3 käsitellään, mitä säteenmarssinta on. Luku 4 käsittelee FFT:n hyödyntämistä erityisesti musiikin visualisointiin käyttämällä fragment-varjostimia. Luvussa 5 käydään läpi, mikä fragment-varjostimen tehtävä on rasteroinissa ja kuinka fragment-varjostinta voidaan hyödyntää säteenmarssinnassa. Luku 6 käsittelee, kuinka uv-koordinaatit määritellään fragment-varjostimessa säteiden suuntausta varten. Luvussa 7 käydään läpi, mitä etäisyyskentät (SDF) ovat ja kuinka ne ovat keskeisiä säteenmarssinnassa. Luvussa 8 käydään läpi menetelmiä, joiden avulla voidaan tehostaa näkymää mm. värittämisessä ja etäisyyskenttien muotojen muokkauksissa. Luvussa 9 käydään läpi insinöörityöprojektin toteutusta. Luvussa 10 kerrotaan haasteista, joita insinöörityöprojektin toteutuksessa tuli vastaan ja sitä, millaisiin ratkaisuihin päädyttiin.

2 Musiikin visualisointi

Musiikin visualisointia nähdään mediasoitinohjelmistoissa. Visualisoinneissa generoidaan animoitua kuvaa musiikista riippuen. Kuva on usein generoitu reaaliajassa ja synkronoitu musiikin tahtiin. Vaikuttavissa musiikin visualisoinneissa generoidut kuvat seuraavat mahdollisimman tarkasti musiikkiraidan spektrikomponentteja, kuten taajuus ja amplitudi.

Musiikin visualisointi voidaan jakaa etukäteen ja reaaliajassa generoituun. Etukäteen generoitua musiikin visualisointia esiintyy mm. musiikkivideoissa, joissa on voitu käyttää tukena videoeditointiohjelmistoa. Reaaliajassa luotu visualisointi voi muuttua erilaiseksi jokaisen eri musiikkiraidan mukaan.

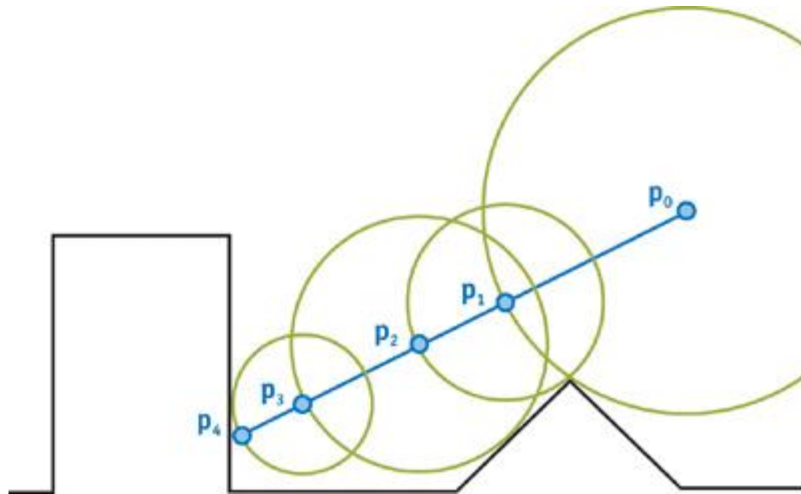
Ensimmäinen musiikin visualisointi oli Atari Video Music, joka tehtiin vuonna 1976. Yksi ensimmäisistä kotitietokoneille tehdyistä musiikin visualisointiohjelmista oli Cthugha vuonna 1993. 1990-luvulla esiintynyt demoskene kehitti reaaliaikatekniikoita musiikin visualisointiin tietokoneilla. Silloin ilmestyivät sellaiset ohjelmistot kuin Cubic player vuonna 1994 ja Inertia Player 1995. Musiikin visualisointi yleistyi 1990-luvun loppupuolella. Silloin puolestaan ilmestyivät mm. ohjelmistot kuten Winamp vuonna 1997 ja Audion vuonna 1999. [3.] Kuvassa 1 näkyy musiikin visualisointia Winamp-musiikinsoittimessa.



Kuva 1. Winamp-musiikkisoittiminen Milkdrop-laajennuksen tuottamaa musiikin visualisointia reaaliajassa [4].

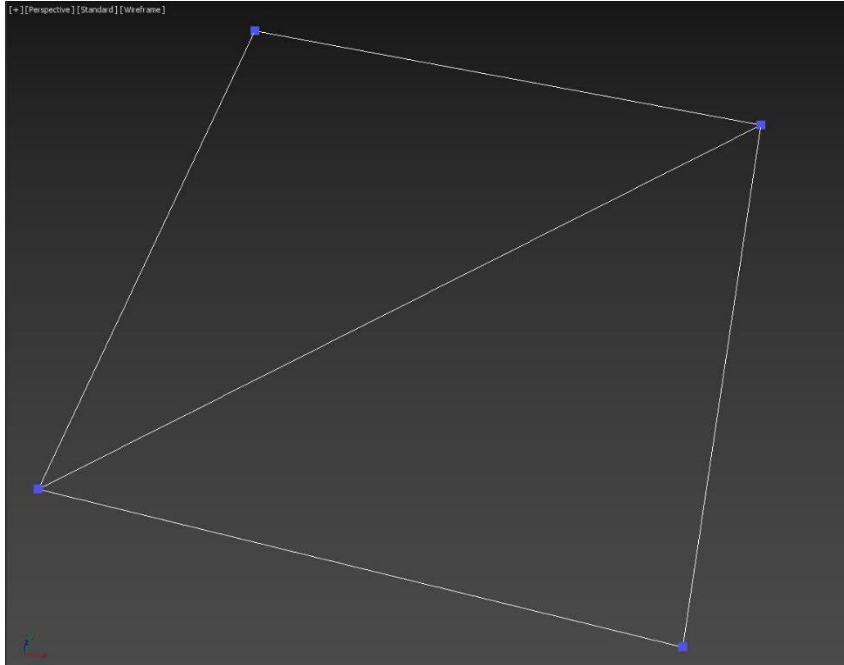
3 Säteenmarssinta

Tässä luvussa selitetään, mitä säteenmarssinta (Raymarching) on käyttäen 3D-grafiikkamoottoreita, jotka tukevat fragment-varjostinominaisuutta. Säteenmarssinnalla pystytään renderöimään monimutkaisia näkymiä reaaliajassa. Tekniikkaa kutsutaan myös pallomarssimiseksi, jonka tavoite on löytää objektit, jotka on esitetty etäisyyskentillä. Tässä pallomarssimisessa toistetaan askelia kysymällä, kuinka lähellä ollaan etäisyyskentän pinnasta. Tätä jatketaan niin kauan, kunnes pinta on löydetty hyväksyttävällä tarkkuudella. [5.] (Ks. kuva 2.)



Kuva 2. Säteenmarssinnan optimointikoodi nimeltä pallomarssinta. P esittää kyseisen askeleen sijaintia, josta haetaan etäisyys lähimpään etäisyyskentän pintaan. Tämän jälkeen marssitaan säteen suunnassa etäisyyden pituinen matka syvemmälle, jolloin kysytään, ollaanko hyväksyttävällä etäisyydellä pinnasta. Jos etäisyys on hyväksytyllä etäisyydellä osumaa varten, asetetaan väri kyseiselle pikselille, johon säde osui. [6.]

Jotta säteenmarssinta voidaan aloittaa, tarvitaan tapa, jolla voidaan suorittaa koodia, joka määrittää ikkunan jokaisen pikselin värin. Tähän varjostimet (Shader) ovat mainioita, koska ne ovat erikoistuneet kyseisen tapaiseen menetelmään. Tässä insinöörityössä hyödynnetään OpenGL-fragment-varjostinominaisuutta. Fragment-varjostin tarvitsee toimiakseen polygoniverkon ja kameran. Polygoniverkon muodolla ei ole merkitystä, mutta käytännöllisin on siisteyden vuoksi levymäinen taso, joka voidaan esittää vähintään kuudella verteksillä (ks. kuva 3). Kamera voidaan luoda käyttämällä OpenGL-rajapintaa tai Unity-pelimoottorissa olevaa oletuspääkameraa. Kamera asetetaan niin, että objekti peittää kokonaan kameran näkökentän. Tällöin jokainen kohta objektista, joka on kameran näkökentän peitossa, kutsuu fragment-varjostinta.



Kuva 3. Tason muotoinen polygoniverkko wireframe-moodissa. Wireframe visualisoi sinisillä neliöillä verteksejä ja valkoisilla viivoilla reunoja (edges). Kuvan polygoniverkko on kolmiopolygoniverkko (triangle mesh). [7.]

Joskus säteenmarssinta sekoitetaan säteenseurantaan. Lähtökohtaisesti molemmissa tapauksissa säteitä haluttaisiin lähettää ikkunan jokaista pikseliä kohti. Kuitenkin, jos ikkunan resoluutio on 320 x 200, pitää lähettää 64 000 sädettä jokaisella ikkunan päivityksellä. Prosessoreille tämä olisi lähes mahdotonta suorittaa reaaliajassa ja näytönohjaimillekin raskasta. Säteenseurannassa säteiden määrää voidaan vähentää, mutta samalla menetetään kuvan laatua. Nykynäytönohjaimet ovat kuitenkin niin nopeita, että usein korkeillakin resoluutioilla pystytään käyttämään tekniikoita reaaliajassa [8]. Säteenseurantaa on pitkään hyödynnetty realistisen valaisun toteuttamiseen renderöinnissä, ja sen käyttö on yleistynyt reaaliaikaisissa peleissä. Säteenmarssinta on puolestaan variaatio säteen suuntauksesta (ray casting), joka sallii käyttää kohteita, joille ei ole analyyttistä kaavaa. Tällöin leikkauspistettä säteen kanssa ei voida yksinkertaisesti laskea ratkaisemalla algebrallinen yhtälö [8]. Toisin sanoen säteenmarssintaa käytettäessä objektit piirretään käyttämällä etäisyyskenttiä.

4 FFT-algoritmi

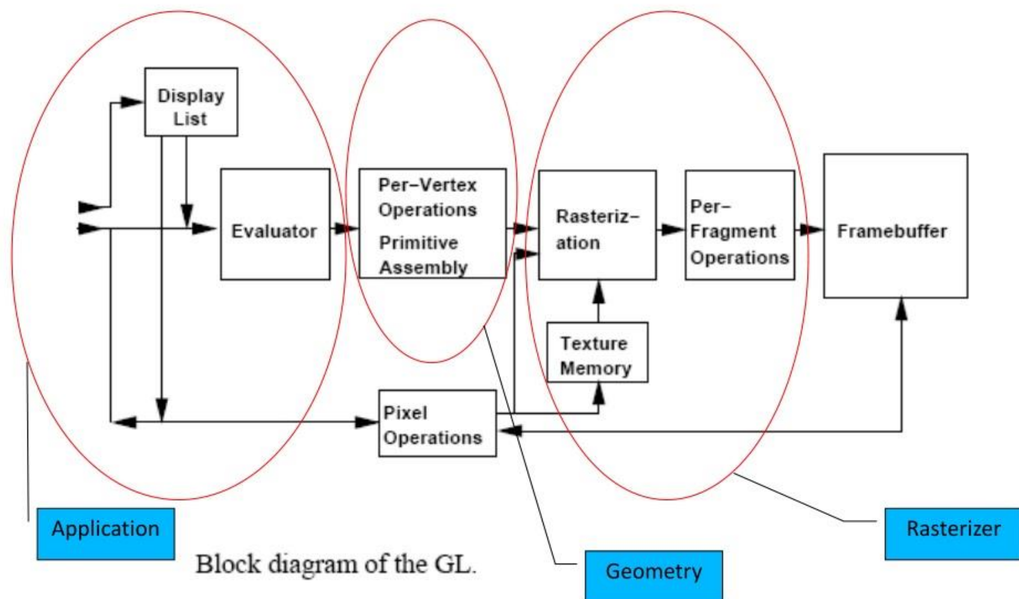
Fast fourier transform (FFT) eli nopea Fourier-muunnos on algoritmi, jolla voidaan muuttaa signaali yksittäisiksi spektrikomponenteiksi. Menetelmällä saadaan tietoa signaalin taajuuksista. Signaalista otetaan näytteitä tietyn ajanjakson aikana ja jaetaan signaali taajuuskomponentteihin [9].

Insinöörityöprojektissa käytettiin Eigen FFT-kirjastoa [9] muuttamaan ääniraidasta saadut ääninäytteet muotoon, jolla niitä voidaan käyttää fragment-varjostimessa musiikin tahtiin. Nämä ääninäytteet saatiin käyttämällä SFML-kirjaston [11] soundbufferin lukemaa ääniraitaa. On syytä huomioida, että ääniraitaa voidaan joko lukea etukäteen tallentamalla kaikki muutetut arvot tai lukea raitaa reaaliajassa syöttämällä tietoa suoraan fragment-varjostimeen. Lukemalla tiedot etukäteen voidaan tehdä musiikkivisualisointiin paremmin muutoksia vertailemalla taajuuksien amplitudeja eri kohdilta raitaa etukäteen. Jos musiikkiraidassa havaitaan huomattavia eroja, voidaan luoda varjostimessa vaihtoehtoisa visualisointeja, joiden välillä vaihdella. Riippuen musiikkiraidan pituudesta tämä prosessi voi kuitenkin viedä aikaa, mikä näkyy ohjelman pidemmässä käynnistysajassa, se vie myös paljon muistia. Vaikka raidan arvot lasketaan etukäteen, varjostimeen itsessään mahtuu vain 1024 ulkopuolista muuttujaa (uniformi) kerrallaan. Kun on saatu arvot FFT:stä, voidaan asettaa uniformit fragment-varjostimeen ja määritellä uniformi ohjelman pääkoodissa. Pääkoodista lähetetään halutut arvot varjostimelle. Tarvittaessa tilaa viediä arvoja voidaan karsia ikkunasta, koska suurin osa taajuuksista on hyvin samankokoista vierekkäisillä ääninäytteillä.

5 Fragment-varjostin

Fragment-varjostin on osa varjostinkokonaisuutta, joka käsittelee rasteroinnin tuottaman fragmentin joukoksi värejä ja yhdeksi syvyysarvoksi. Fragment-varjostinvaihe suoritetaan rasteroinnin jälkeen (ks. kuva 4). Jokainen polygoniverkon peittämä pikseli koostuu useasta fragmentista. [12.] Suurin osa fragment-

varjostimen ominaisuuksista määritellään varjostimen ulkopuolella, muun muassa mitkä värit prosessin lopuksi asetetaan pikselikohtaisesti ikkunan koordinaattien kohtaan. Tällöin käytetään tietoina esim. ikkunan sijaintia, oliko fragmentti luotu edestä vai takaa, ja stencil-arvoa. [13.]



Kuva 4. OpenGL:n toiminta. Rasterroinnin jälkeen fragment-varjostinkutsut tapahtuvat osassa nimeltään ”Per fragment operations”. Säteenmarssinta sijaitsee tässä vaiheessa. [14.]

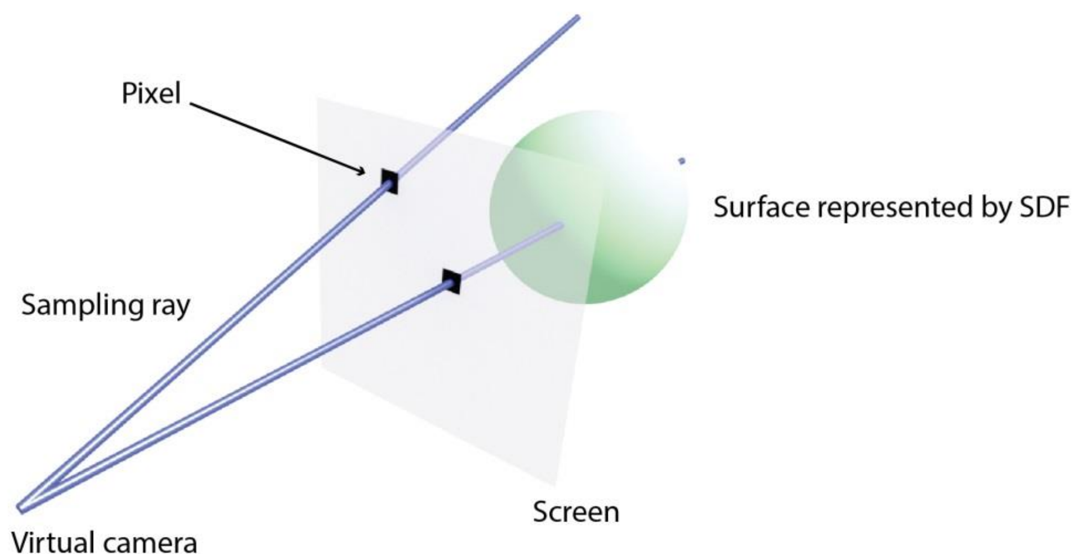
Musiikkivisualisoinnissa ollaan kiinnostuneita uniformista, jonka avulla voidaan antaa arvoja fragment-varjostimelle varjostimen ulkopuolelta. Tässä tapauksessa arvot, jotka saatiin FFT:n kautta, halutaan antaa varjostimelle käyttöön. Varjostimelle voi asettaa enintään 1024 uniformia, mikä tarkoittaa sitä, että on hyvin mahdollista, että joudutaan karsimaan arvoja, joita saatiin FFT:stä ulos. Fragment-varjostimessa on sisäänrakennettuja syötteitä. Näistä säteenmarssintaa varten hyödynnetään in vec4 gl_fragCoord -syötettä, joka sisältää fragmentin sijainnin ikkunassa. X-, y- ja z-komponentit ovat fragmentin ikkunasijainnit. Z-arvo kirjotetaan syvyyspuskuriin, jos gl_FragDepth ei kirjoiteta tässä varjostin

vaiheessa mitään arvoa [12]. Säteenmarssinnassa z-arvoa ei tarvitse käyttää, ellei näkymässä haluta käyttää samanaikaisesti rasteroituja objekteja.

OpenGL-Atomic counters -ominaisuutta käyttämällä on mahdollista selvittää, kuinka monta fragment-varjostinkutsua OpenGL-projektissa tehdään jokaisella ruudunpäivityksellä. Tämä voi olla hyödyllinen työkalu, jos haluaa arvioida ohjelman suorituskykyä [15], sillä fragment-varjostinkutsuja voi olla enemmän kuin ikkunassa on pikseleitä. Varjostinkutsujen määrän tietäminen on hyödyllistä, koska säteenmarssinnassa lähetetään täsmälleen sama määrä säteitä jokaisessa ruudun päivityksessä. Työkalu on kehitetty yleiseen varjostimien virheenkorjaamiseen, joten siinä on myös eri ominaisuuksia verteksi- ja fragment-varjostimien tietojen hakuun.

6 Uv-koordinaatit

Vaikka fragment-varjostin suoritetaan jokaiselle fragmentille polygoniverkon pinnassa, näkymän kameralle pitää kuitenkin erikseen esittää koordinaatit, joiden läpi säteet risteytyvät (ks. kuva 5). Tällä saadaan jokaiselle säteelle määrättyä oikea fragmentti, joista pikseli muodostuu. Tällöin myös oletetaan, että säteen etäisyys objektiin on tarpeeksi pieni, jotta se voidaan hyväksyä osumaksi.



Kuva 5. Säteenmarssinnan oleelliset komponentit. SDF (Signed Distance Field) esittää etäisyyskenttää, Screen esittää uv-koordinaatteja ja sampling ray säteenmarssijan sädettä. Säde on yksinkertaistettu jättämällä pois pallomarssinnan askeleet. [16.]

Termiä uv-koordinaatti ei saa kuitenkaan sekoittaa rasteroinnissa käytettyyn uv-karttaan. Rasteroinnissa uv-karttoja käytetään asettamaan tekstuurit kolmiulotteisen polygoniverkon pintaan, kun taas säteenmarssinnassa käytetyt uv-koordinaatit asettavat säteet kulkemaan kohdista, jotka vastaavat ikkunan oikeita pikseleitä. Jos uv-koordinaattiarvot eroavat ikkunan resoluutiosta voi syntyä tilanne, jossa kuva on joko zoomattu ulos tai sisään. Kuva voi myös näyttää venytetyltä tai litistetyltä. Tapauksissa, joissa havaitaan jokin em. ilmiöistä, voidaan olettaa, että vika löytyy uv-koordinaateista. [17.]

7 Etäisyyskentät (SDF)

SDF(Signed distance field) eli etäisyyskenttä on funktio, jolla voidaan esittää alue tai muoto (ks. esimerkkikoodi 1).

```
(esimerkkikoodi 1). float sdSphere( vec3 p, float s )
{
  return length(p)-s;
}
```

Esimerkkikoodi 1. Esimerkki siitä, miltä etäisyyskenttä pallolle näyttäisi funktiona kirjoitettuna. Koodissa vec3 p vastaa säteen sijaintia ja arvo s pallon sädettä. Muuttamalla s-arvoa pallon koko muuttuu. [5.]

Käyttämällä säteenmarssintaa pystytään lähettämään niin kutsuttuja säteitä fragment-varjostimen sisällä määritellystä virtuaalisesta kamerasta, jolla voidaan etsiä näkymästä, missä lähin etäisyyskentän pinta on kyseisestä säteestä. Fragment-varjostimen avulla värin asettaminen oikeisiin pikseleihin tapahtuu automaattisesti, kun uv-koordinaatit on asetettu. Eräs etäisyyskenttien hyödyllisimmistä ominaisuuksista on kyky muuttaa etäisyyskenttien muotoa käyttämällä muuntajaa. Muuntajien avulla jokaisessa tilanteessa ei tarvitse luoda uutta funk-

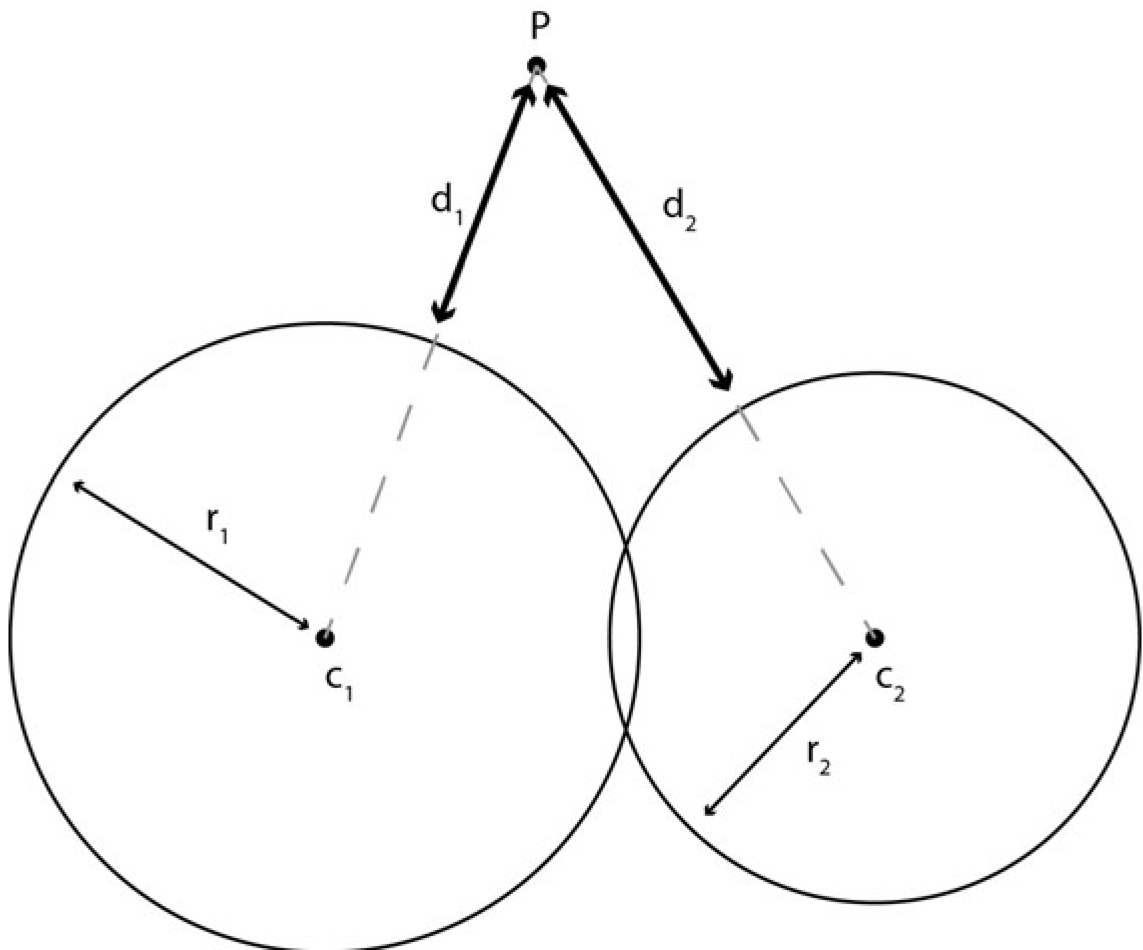
tiota, jolla olisi tietyn näköinen muoto. Muuntajalla voitaisiin esimerkiksi venyttää, pyöristää ja pyörittää. Lisäksi, jos muuntajan arvoa muutetaan ajan myötä, voidaan tehdä etäisyyskenttään reaaliajassa sulavia muodonmuutoksia.

Lisäksi, kun etäisyyskentät palauttavat osuman kaikista kohdista etäisyyskentän pinnasta keskipisteeseen, on tässä valmiina tilavuusominaisuuksia, joiden avulla voidaan tehdä monia graafisia visualisointeja. Joskus haluttua muotoa ei saada aikaan muuttamalla etäisyyskentän muotoa käyttämällä muuntajia. Sen sijaan voidaan liittää muita etäisyyskenttiä yhteen. Näissä tapauksissa voidaan hyödyntää funktioita, joilla saadaan objektit liitettyä yhteen sileästi. Vaihtoehtoisesti muita etäisyyskenttiä voidaan käyttää poistamaan toisen etäisyyskentän muodosta osia pois. Tällaisessa tapauksessa tulevat etäisyyskentän tilavuusominaisuudet hyvin esille. Jos etäisyyskenttien mittakaavaa muutetaan, voi olla hyvä huomioida, että samalla tila joko tiivistyy tai laajenee. Tämä voisi olla ongelma tapauksissa, joissa säde ei kulkisi haluttua matkaa, koska säteen kulku etäisyyskentän sisällä voi joko nopeutua tai hidastua riippuen siitä, mitä muutoksia etäisyyskentälle tehtiin. [5.]

Etäisyyskentät määrittävät, miten objekti renderöidään säteenmarssinnassa, kun taas rasteroinnissa 3D-mallit renderöidään polygoniverkolla. Etäisyyskenttä-funktio ottaa sisään pisteen ja palauttaa etäisyyden kyseisestä pisteestä pintaan, joka on lähimpänä objektia. Jos piste on objektin sisällä, etäisyyskenttä palauttaa negatiivisen arvon [5]. Etäisyyskentillä luodut objektit eivät pysty suoraan vuorovaikuttamaan polygoniverkkopohjaisten objektien kanssa, minkä seurauksena säteenmarssitut objektit piirtyvät aina muiden objektien päälle, vaikka näkymän muiden objektien pitäisi olla lähempänä näkymän katsojaa. Ongelmana on, että säteenmarssitut objektit eivät ota syvyyttä huomioon rasteroinnissa. Jotta ongelma saadaan ratkaistua, pitää saada selville etäisyys jokaisesta säteestä lähimpään polygoniverkkopohjaiseen objektiin. Jos säteenmarssija ylittää tämän pisteen, säde keskeytetään ja renderöidään polygoniverkko-objekti etäisyyskenttä-objektin sijaan. [5.]

Etäisyyskenttiä voidaan piirtää myös kaksiulotteisesti, ne kuormittavat näytönohjainta vähemmän. Kaksiulotteisia etäisyyskenttiä löytyy netistä valmiina funktioina. Haluttaessa kaksiulotteisia etäisyyskenttiä voidaan hyödyntää samanaikaisesti kolmiulotteisissakin näkymissä. Samoin myös muodon muokkaamisoperaatiot, tilan toisto ja etäisyyskenttien yhdistäminen toimivat samalla tavalla kuin kolmiulotteisien etäisyyskenttien kanssa. [18.]

Lisättäessä enemmän etäisyyskenttiä näkymään säteenmarssijan pitää vertailla jokaisella askeleella, mikä on lyhyin etäisyys lähimmän etäisyyskentän pintaan. (Ks. kuva 6.)



Kuva 6. Etäisyyskentät palauttavat arvot, joita vertaillaan toisiinsa. Lyhyimmän etäisyyden pituus otetaan talteen ja marssija liikkuu eteenpäin tämän matkan. Pidemmän matkan harppaaminen voisi johtaa ongelmiin. [16.]

Jos marssija ei tutki jokaisella askeleella, säde voi mennä ohi etäisyyskentästä, johon säteen olisi kuulunut osua. Myös, jos säteen askelväli on liian pitkä, säde voi mennä läpi etäisyyskentästä ilman osumaa. Kun näkymässä on useita etäisyyskenttiä, voi säde saavuttaa maksimiaskelmääränsä ennen osumista näkymän taaimmaiseen etäisyyskenttään. (Ks. kuva 10 sivulta 19.) Tällöin takana olevaa etäisyyskenttää ei saada piirrettyä kokonaan ja edessä olevan etäisyyskentän ympärille muodostuu musta ääriviiva. Jotta ohjelman suorituskyky voidaan pitää korkealla, tavoitteena olisi aina saada haluttu näkymä aikaan käyttämällä mahdollisimman vähän etäisyyskenttiä. Mitä enemmän etäisyyskenttiä on, sitä enemmän joudutaan tekemään etäisyyksien vertailuja marssijan silmukassa. (Ks. kuva 6.)

Tämän vuoksi on suotavaa käyttää sellaisia keinoja kuin etäisyyskentän peilaaminen, mitä voidaan tehdä halutulla akselilla joko näkymämaailman koordinaateissa tai etäisyyskentän paikallisissa koordinaateissa. Lisäksi etäisyyskenttää voidaan toistaa loputtomiin jokaiseen ilmansuuntaan tai toistoa voidaan myös rajoittaa tarpeen mukaan. Huomattavaa on, että toistoa käytettäessä kenttien muotojen vääristyminen johtaa muidenkin toistoa käyttävien etäisyyskenttien vääristymiseen. Jos kenttiä halutaan tarkoituksella vääristää toisistaan eroaviksi, näkymämaailman sijainnilla voidaan saada haluttu vaikutus.

8 Säteenmarssinnan menetelmiä

Tässä luvussa käydään läpi useita menetelmiä, joiden avulla voidaan saada enemmän hyötyä pallomarssinnasta. Etäisyyskenttien muotoja voidaan muokata, asettaa etäisyyskenttään erilaisia värejä ja valolähteitä, lisätä hehkumisefekti ja sumentaa näkymää.

8.1 Etäisyyskenttien muotojen muuttaminen

SDF (Signed distance field) -etäisyyskenttä on funktio, jolla voidaan esittää alue tai muoto. Nämä etäisyyskentän arvot vastaavat sijaintiarvoja. Hakemalla pa-

luuarvoja etäisyyskentältä saadaan matka tarkkailupisteestä etäisyyskentän pintaan. Hyödyntämällä vääristymiä pystytään tehostamaan etäisyyskenttien muotoja tai löytää tapa, jolla yhdistää etäisyyskenttiä toisiinsa. Huomattavaa on, että luultavasti vääristettäessä etäisyyskenttiä vaadittavat askelmäärät pallomarssijalle kasvavat tilan tiivistymisen mukaan. Lisäksi etäisyyskentän muotoa voidaan muokata käyttämällä hyväksi kohinan sekoittavaa vaikutusta. Kohinalla voidaan myös värittää etäisyyskenttää.

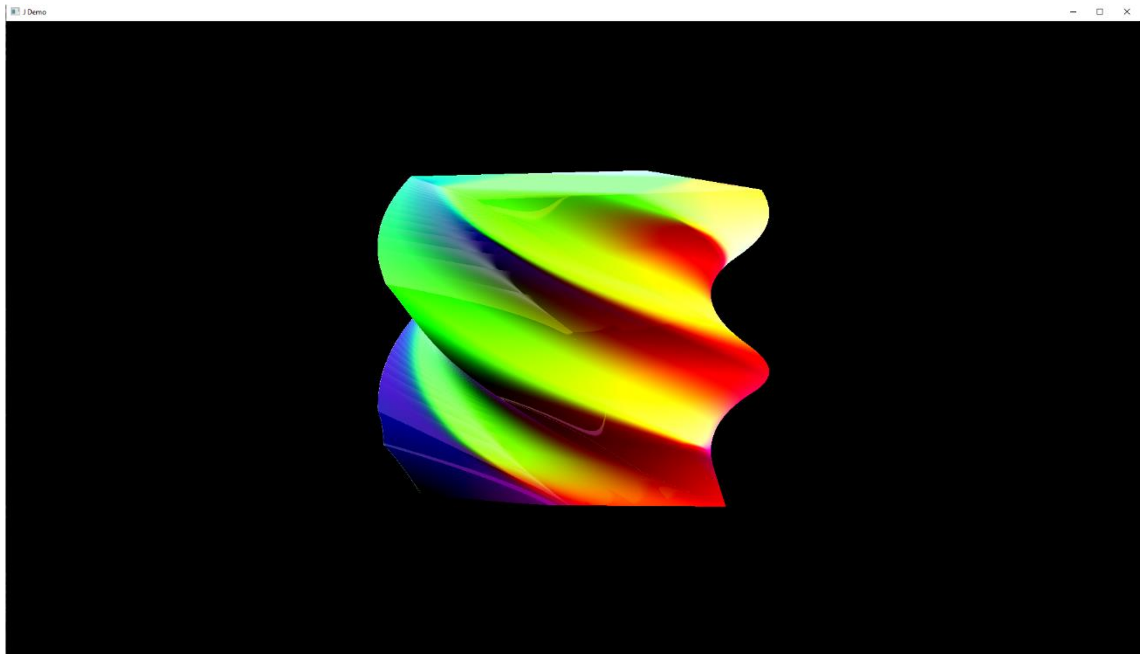
Kohinan käyttö on hyvin yleistä tapauksissa, joissa generoidaan proseduraalisesti tekstuureja ja geometriaa. Yksi käytetyimmistä kohinakuvioista on Voronoin kuvio, koska sen toteuttaminen on suhteellisen yksinkertaista ja sen käyttäminen on kevyttä suorituskykyyn nähden [19]. Kohinaa voidaan käyttää sekoittamaan etäisyyskentän muotoja monilla tavoilla erityisesti etäisyyskenttien volumetristen ominaisuuksien avulla. Kohinaa voidaan myös käyttää värittämiseen, mutta arvoja pitää säätää tarkasti. Helposti muuten tulee aika sekalaista värisotkua, josta on vaikeaa ottaa selvää. Käytettäessä värittämiseen samaa kohinakuviota kuin etäisyyskentän muodon muuttamiseen saadaan sovitettua efektit yhteen. [19; 20.]

Kun käytetään säteenmarssintaa musiikin visualisointiin, on etäisyyskenttien muokkaus mahdollisesti tärkein menetelmä, koska sen avulla voidaan muokata objektin muotoa musiikkiraidasta saatujen näytteiden avulla huomioiden, että nämä on saatu käyttämällä FFT-algoritmia. Suorat näytteet musiikkiraidasta (ilman FFT:tä) voivat parhaimmillaan sisältää vain signaalin taajuuksien kokonaisen amplitudin. Olettaen ääninäytteiden olevan kelpoisessa muodossa etäisyyskentän muotoa voidaan muokata lähes rajattomasti käyttäen erilaisia operaatioita kuten etäisyyskentän siirtämistä, vääntämistä ja taivuttamista. Esimerkkikoodissa 2 on tapa, jolla vääntämistä voidaan toteuttaa.

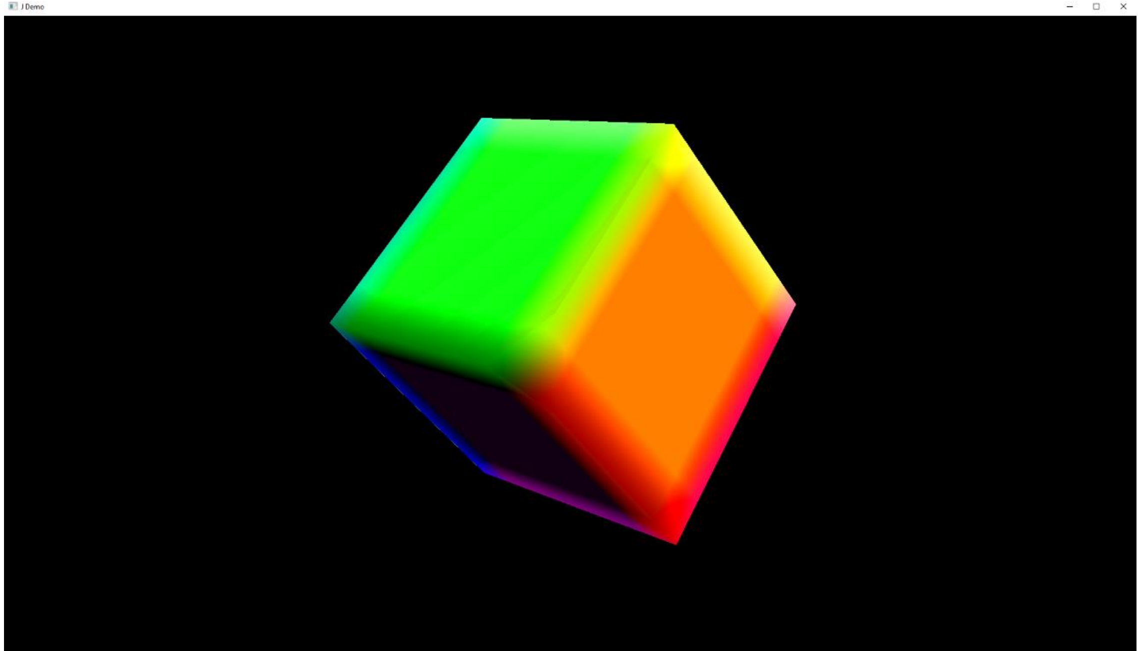
```
float opTwist( in sdf3d primitive, in vec3 p )
{
    const float k = .5; // or some other amount
    float c = cos(k*p.y);
    float s = sin(k*p.y);
    mat2  m = mat2(c,-s,s,c);
    vec3  q = vec3(m*p.xz,p.y);
    return primitive(q);
}
```

Esimerkkikoodi 2. Funktio, jolla voidaan vääntää etäisyyskenttää [5].

Myös kuvat 7 ja 8 visualisoivat, miltä etäisyyskentän muodon muuttaminen voi näyttää ennen muutosta ja sen jälkeen. Lisäksi voidaan hyödyntää säteenmarssijan säteiden x-, y-, ja-z-sijainti komponentteja muokkaamaan etäisyyskentän muotoa eri kohdissa erilaisilla voimakkuuksilla.



Kuva 7. Etäisyyskentän vääntäminen hyödyntämällä funktiota, joka näkyy esimerkkikoodissa 2.



Kuva 8. Objekti ennen etäisyyskentän vääntämistä.

Jotkut muokkausoperaatiot tekevät etäisyyskentistä epäeuklidisia, joten niitä käytettäessä pitää olla varovainen, koska ne vaikuttavat säteenmarssintaan. Tällöin mahdollisesti säteenmarssijan askelkokoa pitää pienentää [5]. Tilan tiivistyessä säteenmarssijan säteen pitää liikkua lyhyempiä matkoja, sillä marssija voi joutua tekemään enemmän askelia päästäkseen kohteeseen. Haluttaessa voidaan nostaa säteenmarssija silmukan hyväksyttävää maksimiaskelmäärää. Tämä aiheuttaa kyllä enemmän rasitusta näytönohjaimelle. Muokkausoperaatioita voidaan tehdä monia samanaikaisesti, mutta joissakin tapauksissa etäisyyskentät voivat muokkautua ennalta-arvaamattomilla tavoilla, vaikka etäisyyskentän vääntäminen ja siirtäminen toimisivat erillisissä tapauksissa tarpeen mukaan. Yhtäaikainen käyttö voi johtaa muotoihin, joita ei haluta. Sen sijaan musiikin visualisointiin epämääräiset muodonmuutokset voivat olla jopa haluttuja. Tällä tavalla saadut muodonmuutokset voivat olla hyvin epävakaita, joten niiden hienosäätäminen täytyy tehdä yksinkertaisesti kokeilemalla.

Fragment-varjostimelle pystytään tuomaan uniformien avulla tekstuureja, joiden avulla voidaan muokata etäisyyskentän muotoa, jos musiikin ääniraidasta saa-

duilla näytteillä saadaan generoitua tekstuuri. Tekstuuria voidaan käyttää samalla tavalla kuin korkeusmappeja (heightmap) käytetään muokkaamaan objektien pintoja, olettaen, että tekstuuria päivitetään myös musiikin tahdissa jokaisessa ruudun päivityksessä. Tällaista toteutusta hyödynnetään Shadertoy-sivulla [21] vaihtoehtoisilla kanavilla.

8.2 Värit ja materiaalit

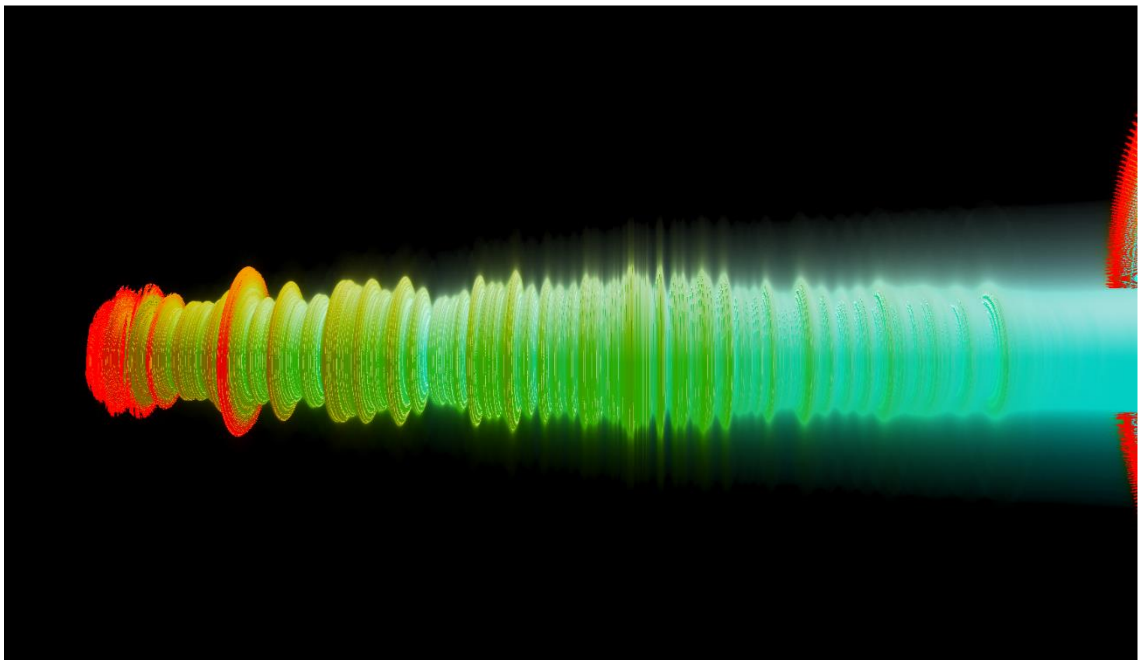
Etäisyyskentillä luotujen objektien värittäminen onnistuu helpoiten hyödyntämällä kenttien sijainteja. Sekä fragment-varjostimen palauttama väri että säteenmarssinnassa käytetty avaruuskenttä on esitetty käyttäen vector3:sta, jossa sijantikomponentit ovat x, y, z ja varjostimen värivektorin komponentit ovat R, G, B. Sijaintivektori voidaan asettaa varjostimen palauttamaan väriin. Tämän avulla saadaan piirretylle objektille täsmälleen samanlainen väritys, kuin miltä objekti näyttäisi rasterointia käyttämällä, kun sen väri määräytyisi uv-kanavan mukaan. Jos objektien väri asetetaan käyttämällä näkymämaailman koordinaatteja, menetetään kuitenkin paljon värien hallinnassa, koska objektien väri voimistuu nopeasti mitä kauemmaksi objekteja liikutetaan poispäin näkymämaailman lähtöpaikasta. Lisäksi myös tietyn värin asettaminen haluttuun kohtaan on hankalaa.

Jos varjostimen palauttamat RGB-väriarvot ovat kaikki kolme nolla [vec3(0,0,0)], on tuloksena musta väri, ja jos kaikki väriarvot ovat ykkösiä [vec3(1,1,1)], on tuloksena valkoinen väri. Punainen väri tulee arvoilla vec3(1,0,0), vihreä arvoilla vec3(0,1,0) ja sininen arvoilla vec3(0,0,1). Vec3(1,1,0) antaa väriksi keltaisen.

Esimerkiksi musiikin visualisointia tehdessä ääniraidan taajuuksien amplitudin pituus voidaan sitoa maailmannäkymän y-akseliin värin muokkaamiseksi. Voidaan siis asettaa esimerkiksi niin, että alhaisella amplitudilla väri on sininen, keskipitkillä vihreä ja korkeilla punainen. Väri muuttuu tällöin asteittain. Samalla tavoin myös objektin muodon muutos voidaan sitoa sijantiin y-akselilla. Vaihtoehtoisesti väriä voidaan lisätä tilanteissa, joissa taajuuden amplitudi on hyvin korkea verrattuna musiikkiraidan keskiarvoiseen amplitudiin. Yleisesti amplitudit

matalammilla taajuuksilla ovat huomattavasti pidempiä verrattuna amplitudeihin korkeammilla taajuuksilla.

Projektissa käytettiin samaa funktiota värin muuttamiseen, kuin mitä käytettiin etäisyyskentän muodon muuttamiseen. Sen avulla etäisyyskentän väri muuttui samoilla kohdilla, kuin missä etäisyyskentän muoto muuttui. Funktio perustui siis fragment-varjostimen näkymämaailman koordinaattiin. Kuvassa 9 näkyy musiikin visualisoija toiminnassa ja se, kuinka väri muuttuu eri taajuuksien amplitudien mukaan. Objektille oli asetettu pohjaväri saman funktion avulla. Lisäksi oli kaksi apuarvoa, joilla pohja-arvoja muutettiin. Apuarvoista toinen vastasi korkeita amplitudiarvoja ja toinen matalia amplitudiarvoja. Niillä painotettiin, mihin suuntaan värin kuuluisi vaihtua. Lopputuloksessa värit vaihtuvat pehmeästi väristä toiseen.



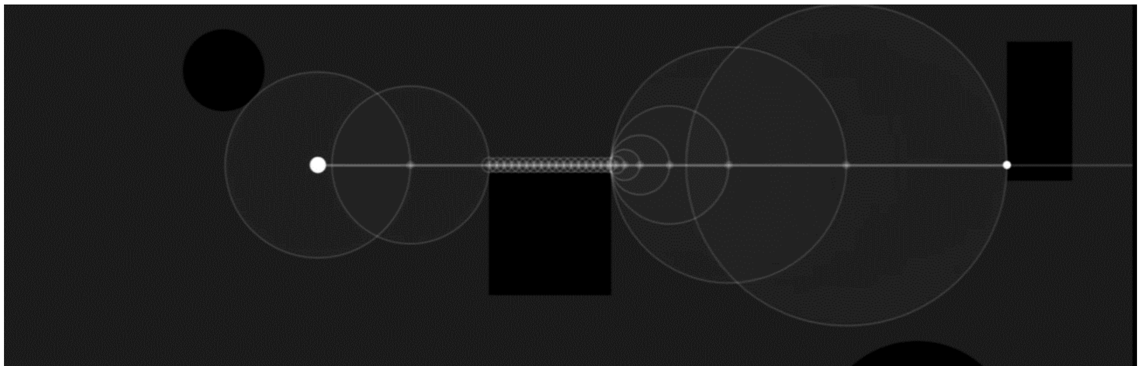
Kuva 9. Musiikkivisualisoinnista, jossa väriä lisätään etäisyyskenttään riippuen kyseisen taajuuden amplitudista.

Värittämistä ei ole kuitenkaan rajoitettu suoraan RGB-arvojen muuttamiseen. Fragment-varjostimeen voidaan haluttaessa tuoda tekstuureja, joilla voidaan saada tietynnäköisiä pintoja objekteille. Tyypillistä on, että halutaan kuitenkin

rikkoa tekstuurin toistumista lisäämällä sen kierteeseen pseudosatunnaisia arvoja. Yksinkertaisesti voidaan vaikka peilikuvata tekstuuria x- ja y-akselilla.

Hehkuva efekti

Käytettäessä pallomarssimista säteen askelmäärä kasvaa huomattavasti mitä lähempänä säde liikkuu etäisyyskentän pintaa (ks. kuva 10, jossa visualisoidaan säteenmarssijan kulku etäisyyskentän pinnan läheltä.)



Kuva 10. Säteen kulkiessa lähellä etäisyyskenttää huomataan askelmäärien kasvavan riippuen tarkkailtavan askeleen etäisyydestä etäisyyskentän pintaan. Lisäämällä väriä, kun etäisyys on tarpeeksi pieni, saadaan aikaan hehkuva efekti. [22.]

Samasta asiasta on myös koodiesimerkki 3, jossa näkyy, kuinka lisäämällä väriä etäisyyskentän pinnan lähellä saadaan aikaan hehkumista.

```
for(int i = 0; i < 100; i++) {
 d = dist(p);
   p += d * rayDir;
   if (d<0.5)
   {
         col +=  mix(objcol, vec3(.005,.005,.005), 1);
   }
     if (d<0.01)
   {
         col +=  mix(objcol, vec3(.01,.01,.01), 1);
           break;
   }
}    return col;
```

Esimerkkikoodi 3. Koodinpätkä säteenmarssijasta, jossa väriä lisätään pienissä määrissä, vaikka säde ei ole suoraan osunut etäisyyskenttään, mutta on kuitenkin hyvin lähellä etäisyyskentän pintaa. Tämän seurauksena objekti saa hehkuvan efektin. Tässä tapauksessa säteenmarssijan silmukkaa ei katkaista kuten säteen osuessa etäisyyskenttään.

Asettamalla säteenmarssija silmukkaan tarkistusetäisyydelle (joka on isompi kuin törmäykseksi hyväksyttävä etäisyys) voidaan lisätä valittua väriä pienissä määrissä jokaisella säteenmarssijan askeleella. Muuttamalla hyväksyttävää etäisyyttä ja sitä, kuinka paljon väriä lisätään jokaisella askeleella, voidaan säätää objektin reunoille muodostuvaa värin hehkumista. Tarpeen mukaan värin lisäämisen määrää voidaan kasvattaa jokaisella askeleella. Jos taas hehkumista halutaan leveämmälle alueelle, voidaan kasvattaa hyväksyttävää etäisyyttä askeleiden välillä. Samalla lisättävää värimäärää voi olla hyvä laskea, ettei hehkumisefekti tule liian voimakkaaksi.

Efektin lisääminen aiheuttaa hyvin mitättömän lisäkuormituksen suorituskyvylle, koska säteenmarssinnalle vaadittavat etäisyydet askeleilla joudutaan joka tapauksessa laskemaan.

8.3 Valolähteet ja varjot

Säteenmarssintaa käytettäessä varjojen toteuttaminen on yksi tekniikan vahvuuksista, koska tekniikalla saadaan näkymästä kokonaista tietoa. Pisteiden varjostamiseksi säteenmarssinnalla voidaan helposti tutkia ympärillä olevaa geometriaa käyttämällä etäisyysfunktiota. Rasterointitekniikassa puolestaan joudutaan tekemään varjokarttoja myöhempään käyttöön tai lähettää säteitä, joilla

selvittää, mitkä osat geometriasta ovat varjon peitossa. Koska säteenmarssinnassa tietoa geometriasta saadaan selville huomattavasti yksinkertaisemmin käyttämällä etäisyysfunktiota, monet realistiset varjostus- ja valaistustekniikat ovat helppoja toteuttaa. [23; 24.]

Objektien varjostamisessa voidaan luoda valolähde tekemällä vektori, jolla on suunta. Säteen osuessa etäisyyskenttään voidaan ottaa pinnasta normaali ja tehdä ristitulo vektorin ja normaalin välillä. Asettamalla saatu valoarvo väriin käyttämällä mix- tai lerp-funktioita sekoittamaan objektien värit valon tuottamien varjostuksen kanssa, voidaan saada aikaan yksinkertainen varjostustoteutus (ks. esimerkkikoodi 4). Toteutuksessa objektit eivät kuitenkaan luo varjoja muihin objekteihin, joten kappaleet vaikuttavat irrallisilta.

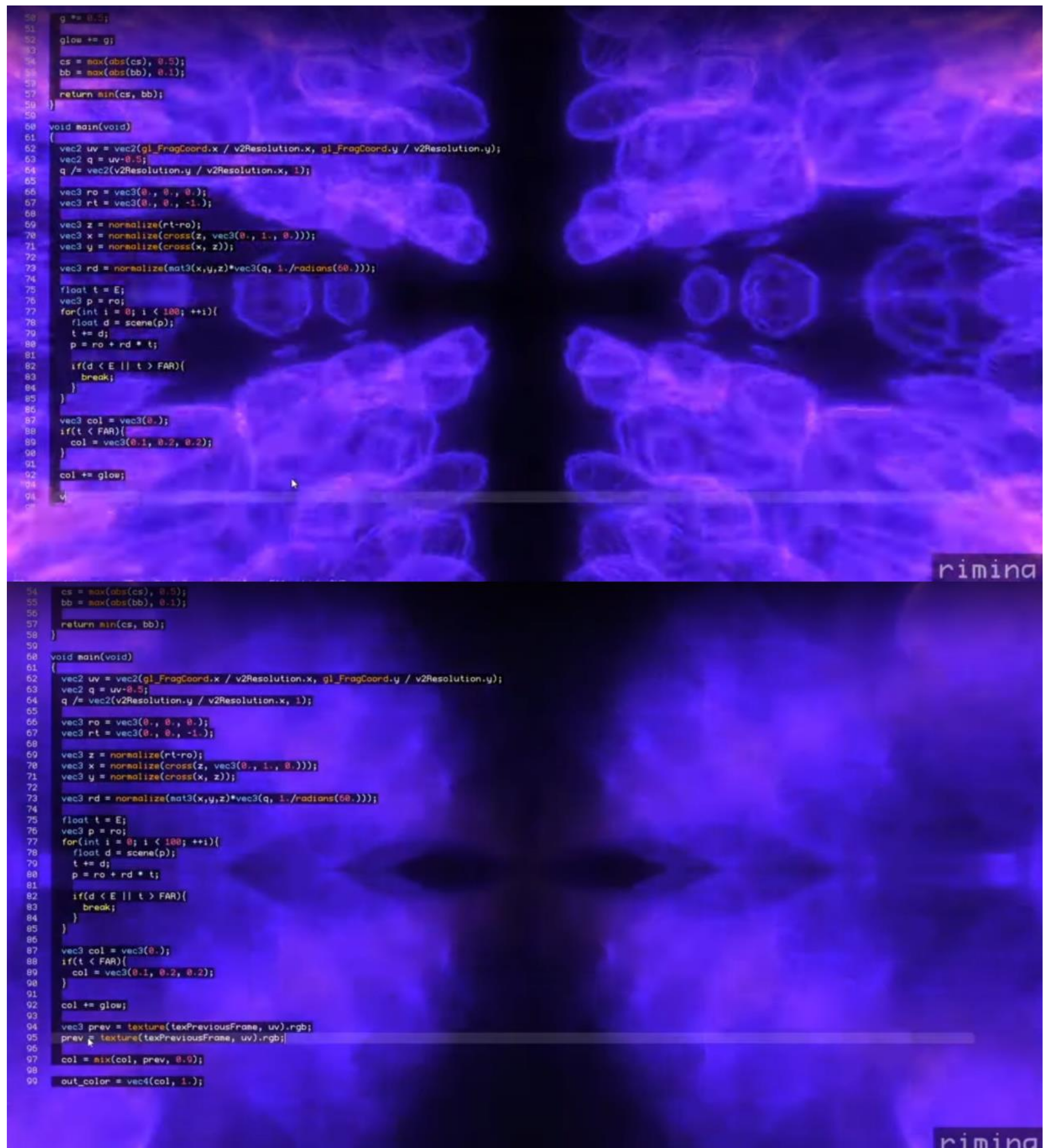
```
if (d<0.01)
{
     vec3 n = normal(p);
     vec3 l = normalize(vec3(1,1.,1.));
     float light = max(dot(n,l),0.);
     col = mix(objcol, vec3(.1,.1,.1), light);
     break;
}
```

Esimerkkikoodi 4. Toteutus objektien varjostamisesta. Varjostus tehdään säteen osuessa etäisyyskenttään

Selvärajaisten varjojen tekemisessä lähetetään etäisyyskentän pintaan osunut säde kohti valolähdettä. Jos säteen välissä on jotain, voidaan olettaa, että kohta on varjon peitossa. Jos halutaan pehmeärajaiset varjot, voidaan ottaa huomioon, kuinka läheltä etäisyyskenttää säde kulki ja kuinka kaukana säteen osumapiste on valolähteestä. Mitä lähempänä säde kulki etäisyyskenttää ja mitä kauempana osumapiste oli valolähteestä, sitä tummempi varjon kuuluisi olla.

8.4 Sumentaminen ja liike-epäterävyys

Joissakin tapauksissa voidaan haluta kuvaan sumentavaa vääristystä tai liikkeen vaikutusta vastaavaa epäterävöittämistä (motion blur) korostamaan vauhdin tuntua tai muuta epäselvyyttä. Kuvassa 11 näkyy ero ilman sumentamista ja sumentamisen kanssa.



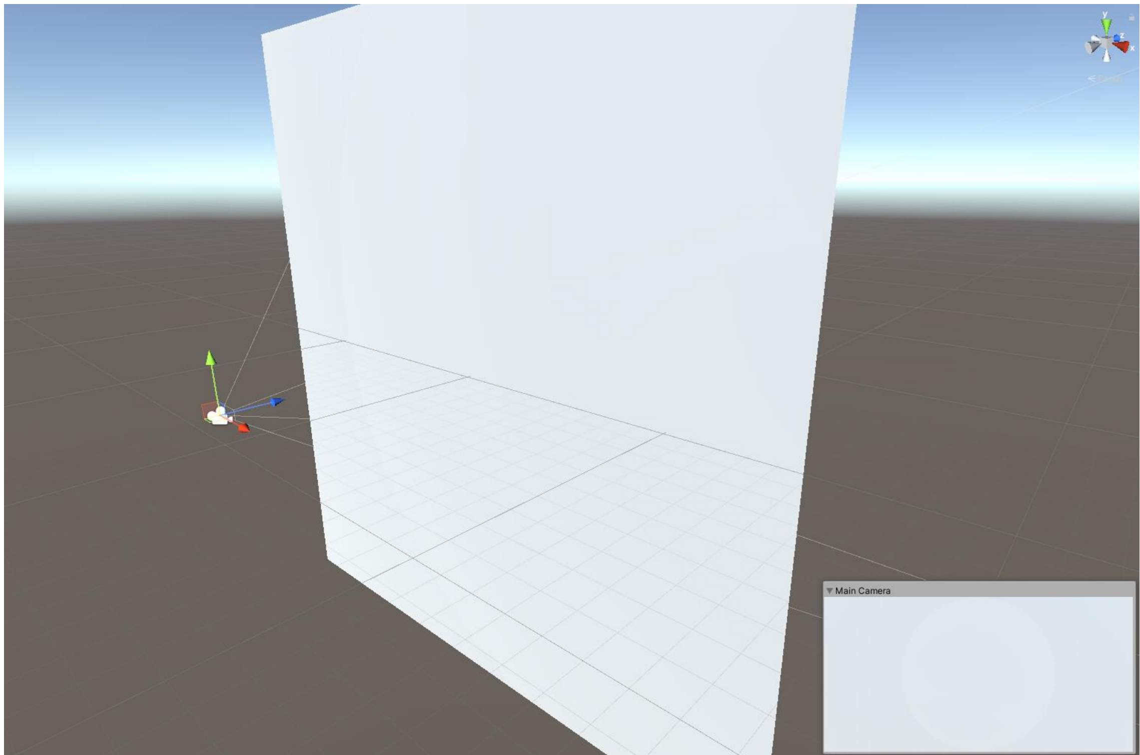
Kuva 11. Assembly 2022:n shader showdown -kilpailussa käytetty liikkeen epäterävöittämisen efekti, jossa ylempi kuva on näkymä ennen sumentamista [25]. Kilpailussa oli framebuffer-ominaisuus käytössä, joten ei ole nähtävissä, kuinka tieto edellisistä kuvanpäivityksistä oli saatu. Mutta tekstuurin värien sekoittaminen reaaliajassa laskettujen värien kanssa on näkyvissä.

Toteutuksen saa hyvin aikaan hyödyntämällä, vaikka ottamalla edellisestä ruudunpäivityksestä muistiin kuva- tai muuta dataa, jota voidaan sekoittaa yhteen reaaliajassa olevaan ruudunpäivitykseen. Lisäämällä pientä vaihtelua joko uv-koordinaatteihin tai itse etäisyyskenttien sijainteihin voidaan säätää, kuinka voimakas sumennus on. Toteutuksessa on haastavaa, kuinka itse datan edellisistä

kuvanpäivityksistä saadaan minkäänlaista dataa ulos, koska kaikki näytönohjaimen puolella olevat laskutiedot menetetään jokaisen ruudunpäivityksen jälkeen. Tästä syystä data edellisistä ruudunpäivityksistä pitäisi olla jo käytettävässä muodossa, ennen kuin se on viety fragment-varjostimelle uniformien kautta. Kuitenkin shadertoyn [21] kaltainen ympäristö tukee frameBuffer-ominaisuutta, jolla voidaan hyödyntää edellisiä ruudunpäivityksiä. Puskurissa voidaan ladata vanhempi ruudunpäivitys tekstuuriin ja tehdä haluttuja muutoksia. Tekstuuri itse muodostuu värikoordinaateista, jotka voidaan summata säteenmarssijan reaaliajassa palauttamien väriarvojen kanssa.

9 Säteenmarssinta musiikin visualisoinnissa toteutus

Musiikin visualisoiminen säteenmarssinnalla on hyödyllistä tehdä 3D-ohjelmointirajapinnalla, jolla voidaan piirtää 3D-muotoja ja käyttää fragment-varjostimia. Tässä projektissa käytettiin OpenGL-rajapintaa, mutta mikä tahansa pelimoottori, kuten esimerkiksi Unity tai Unreal Engine, kelpaa myös, kunhan se vain tukee varjostimia. Sen jälkeen, kun uusi projekti on luotu, tarvitaan pääkamera ja polygoniverkko. Käytettäessä pelimoottoria pääkamera luodaan automaattisesti (ks. kuva 12).



Kuva 12. Unityssä luotu esimerkkikuva, jossa pääkamera on suunnattu päin tason muotoista polygoniverkkoa. Oleellista on, että kameran näkymä on kokonaan peitettynä. Taso saa kuitenkin ulottua yli näkymän ilman ongelmia. Pääkamera ei ole sama kuin fragment-varjostimessa oleva virtuaalinen kamera (eli säteiden lähtöpiste).

OpenGL:a käytettäessä pitää ensin luoda view-matriisi ja sille projektiomatriisi. Tämän jälkeen näkymä voidaan suunnata ja asettaa sille perspektiivi, josta saadaan aikaan pääkamera. Kun kamera toimii, tarvitaan polygoniverkko, jolla kutsutaan fragment-varjostinta. Tämän polygoniverkon muoto vaikuttaa vain suoritustehoon. Yksinkertaisimmillaan tason muotoinen polygoniverkko voi koostua vähintään kuudesta verteksistä. Mutta vaikka polygoniverkolla olisi enemmänkin verteksejä, vaikutuksen suorituskykyyn pitäisi olla mitättömän pieni. Polygoniverkko voidaan luoda halutussa 3D-ohjelmassa kuten 3ds Max tai Blender. Kun polygoniverkko on tuotu ohjelman näkymään, suunnataan kamera päin polygoniverkkoa niin, että polygoniverkko peittää kokonaan kameran näkymän. Tämä saa polygoniverkko-objektin kutsumaan siihen liitettyä fragment-varjostinta, jossa itse säteenmarssinta sijaitsee.

9.1 Säteenmarssinnan toteutus

Säteenmarssinnan kirjottamisen voi aloittaa luomalla uv-kartan, jossa tehdään vector2, joka ohjaa jokaisen säteen oikeaan suuntaan. Hyödyntämällä GLSL:n gl_FragCoord.xy-muuttujia voidaan erotella säteille omat kohdat uv-kartassa. Toinen tavoite on saada uv-koordinaateille oikea kuvasuhde, joka voi riippua ohjelmaan asetetun ikkunan resoluutiosta. Koordinaatit on hyvä vielä keskittää, koska gl_Fragcoord.xy-muuttujat alkavat iteroimaan kuvaa ruudun vasemmasta yläkulmasta ja säteiden suuntaus ei ole oikea. Tämä saadaan korjattua vähentämällä gl_FragCoord_xy-muuttujista 0,5. Kun uv-koordinaatit on asetettu, voidaan luoda virtuaalinen kamera luomalla vector3, joka on säteiden lähtökohta. Säteet suunnataan luomalla toinen vector3(uv.x, uv.y, 1). Seuraavaksi voidaan luoda fragment-varjostimen väriulostulo-vector4 fragColor, joka kutsuu säteenmarssinnan pääfunktiosilmukkaa.

Toteutuksessa halutaan hyödyntää tekniikkaa nimeltään pallomarssiminen. Tässä tavassa säteenmarssijan säde etenee jokaisella silmukan askeleella saman pituisen matkan, kuin missä lähin etäisyyskentän pinta havaitaan. Tämän avulla säteet pääsevät etenemään mahdollisimman vähillä askeleilla riippumatta siitä, kuinka kauas etäisyyskentät on asetettu ja myös pinnan sijainti voidaan havaita hyvällä tarkkuudella. Säteenmarssijafunktiolle annetaan aikaisemmin asetetut säteen aloituspaikka ja säteen suunta. Säteenmarssijan toimintaperiaate on ottaa askel eteenpäin kutsumalla etäisyysfunktiota, joka palauttaa marssijan silmukkaan tiedon etäisyydestä lähimmän havaitun etäisyyskentän pintaan. Jos tämä etäisyys on pienempi kuin osumaksi hyväksytty etäisyys, voidaan laittaa silmukka poikki ja asettaa fragmenttiin törmätyn etäisyyskentän väri. Tapauksessa, jossa säde ei törmännyt etäisyyskenttään, otetaan lähimmän etäisyyskentän matka talteen ja liikutaan säteellä samanpituinen matka eteenpäin. Säteelle on hyvä asettaa sallittu maksimiaskelmäärä, jotta tapauksissa, joissa säde ei tule törmäämään mihinkään, ei kulu liikaa suorituskykyä tai pahimmassa tapauksessa sovellus jumitu.

Perussäteenmarssijan viimeinen osa on etäisyysfunktio, jolla luodaan itse etäisyyskentät ja haluttaessa asetetaan niille värit. Tämä funktio ottaa sisään säteenmarssijan tutkittavan askeleen ja vertailee sitä muiden etäisyyskenttien etäisyyksien kanssa (ks. esimerkkikoodi 5).

```
#version 330 core
 out vec4 fragColor;
 in vec2 fragCoord;
uniform float time;
uniform float frequencyToSend;
uniform float[1022] magVecToSend;
vec3 objcol;

float sdSphere( vec3 p, float s )
{
     return length(p)-s;
}

float dist(vec3 p)
{
    float d = sdSphere(p , 1);
    objcol = p;
    return d;
}

vec3 March(vec3 rayOri, vec3 rayDir){
    vec3 p = rayOri, col = vec3(0.);
     float d = 0.;
     for(int i = 0; i < 100; i++) {
        d = dist(p);
        p += d * rayDir;
         if (d<0.01)
        {
            col +=  mix(objcol, vec3(.01,.01,.01), 1);
             break;
        }
     }
    return col;
}

void main()
{
    vec2 uv = (vec2(gl_FragCoord.x, gl_FragCoord.y) / vec2(1920, 1920)
-.5);
    vec3 rayOri = vec3(25,13,-60);
    vec3 rayDir = vec3(uv.x, uv.y, 1);
    vec3 col = March(rayOri, rayDir);
    fragColor = vec4(col, 1.);
}
```

Esimerkkikoodi 5. Esimerkki, jossa on piirretty pallo käyttämällä säteenmarssintaa.

GLSL-varjostinkielen min-funktio on hyödyllinen tässä tapauksessa, koska sillä voidaan helposti vertailla monimutkaisienkin kolmiulotteisten etäisyyskenttien pintojen etäisyyksiä toisiinsa. Musiikkivisualisoijalle hyödyllisimmät etäisyyskentät ovat muodot, joissa on erikseen esitetty x-,y- ja z-arvot, koska luultavasti valitaan joko yksi tai kaksi akselia, joissa etäisyyskentän muotoa muutetaan musiikista saatujen FFT-arvojen mukaan. Huomattavaa on, että vaikka etäisyyskenttiä voidaan luoda monia samanaikaiseen näkymään, ohjelman suorituskyky laskee nopeasti mitä enemmän etäisyyskenttiä luodaan näkymään. Tämän vuoksi olisi suositeltavaa luoda mahdollisimman vähän etäisyyskenttiä ja muunnella niitä, jotta saataisiin haluttuja näkymiä.

9.2 Musiikin lukeminen ja tuominen varjostimeen

Jotta saadaan tietoa säteenmarssijan etäisyyskenttien muodon muuttamisen, tarvitaan tapa, jolla saada luettua musiikkiraidasta näytteitä. Insinöörityöprojektissa käytettiin SFML-kirjaston äänimoduulia [11]. Lisäksi tarvitaan tapa, jolla muuttaa näytteet musiikkivisualisoijalle kelpoiseen muotoon. On suositeltavaa käyttää FFT-algoritmia. Projektissa käytettiin Eigen [10] ohjelmointikirjaston FFT-moduulia, joka puolestaan perustuu kissfft-moduuliin [26]. FFT:tä käytettäessä valitaan, kuinka monta näytettä otetaan ikkunaan. Tämän ikkunan suositeltu koko muuttuu riippuen luetun äänitiedoston näytteenottotaajuudesta [27].

Huomattavaa on, että fragment-varjostin tukee enintään 1024:ää uniformia. Sen vuoksi tämä arvo voi olla helppo valinta ikkunan koolle. Tapauksissa, joissa äänitiedostojen näytteenottotaajuus on hyvin korkea, tieto, joka saadaan mahtumaan 1024 näytteen ikkunaan, voi olla puutteellista. Onneksi kuitenkin suurin osa musiikkitiedostojen näytteenottotaajuuksista on joko 44,1 kHz tai 48 kHz. Molemmissa tapauksissa näytteiden määrän pitäisi olla kelvollinen. FFT-algoritmin jälkeen on hyödyllistä käyttää hanning-suodatinfunktiota taajuuksien pyöristämiseen. Jotta näytteet saadaan vietyä fragment-varjostimeen, FFT-arvot on hyvä laittaa lukujonoon. Tämä lukujono linkitetään varjostimen kanssa asettamalla sille ID ja uniformi. Lisäksi fragment-varjostimeen pitää luoda uniformi, jolla tieto otetaan vastaan.

9.3 Etäisyyskentän luominen ja taajuusarvojen käyttäminen etäisyyskentän muodon muokkaamiseen

Koodiriveille, joissa on silmukka, joka käsittelee säteenmarssijan askelia, voidaan laittaa funktio, jonka tehtävä on palauttaa liukuarvo, joka kertoo, kuinka pitkä etäisyys säteenmarssijan askeleen sijainnilta on lähimmän etäisyyskentän pintaan. Funktio tarvitsee säteen tämänhetkisen sijainnin silmukassa. Funktiota usein nimitetään etäisyysfunktioksi tai näkymäksi, koska siellä sijaitsevat etäisyyskentät, joilla objektit esitetään. Alkujaan funktioon halutaan luoda liukuarvo, jota voidaan kutsua etäisyydeksi. Tähän arvoon asetetaan halutun primitiivimuodon funktio. Tämä funktio tarvitsee etäisyysfunktiolle annetun tarkkailtavan pisteen sijainnin.

Lisäksi tarvittavia parametrejä ovat objektin sijainti ja sen koko. Se, missä muodossa parametrit täytyy antaa, vaihtelee riippuen siitä, minkä muotoinen primitiivi on kyseessä. Asettamalla funktio palauttamaan etäisyysarvo voidaan olettaa, että ohjelma nyt pystyy piirtämään objektin. Huomattavaa on, että useamman etäisyyskentän piirtämiseen samanaikaisesti kannattaa käyttää min-funktiota, jonka avulla voidaan vertailla useamman objektin etäisyyksiä toisistaan palauttamalla lähimmän etäisyys. Jotta voidaan käyttää äänidataa, tarvitaan tapa, jolla muunnetaan tieto lukujonosta. Lukujonossa täytyy tällöin taajuuksien amplitudien olla skaalattuja tarpeeksi pieniksi, jotta etäisyyskentän muotoa pystytään muuntamaan järkevästi. Tätä varten luodaan funktio, jonka tehtävä on ottaa sisään tarkkailtava sijainti, jota vertaillaan lukujonon arvoihin, jotka puolestaan vastaavat eri taajuuksia. Hyödyllistä olisi käyttää glsl-varjostinkielen mod-funktiota, jonka avulla pystytään kiertämään ääninäytelukujonon indeksejä asettamalla sijainti x-akselilla verrattavaksi ääninäytelukujonon kokoon. (Ks. esimerkkikoodi 6.)

```
float posToMag(float x){
float  section = fftMag[int(abs(mod((x), 1024)))];
return section;
}

float posToMagXZ(float x, float z){

float  section = fftMag[int(abs(mod(((length(x * x + z * z))),
1024)))];
return section;
```

Esimerkkikoodi 6. Kaksi versiota esimerkkikoodista, jonka avulla saadaan äänidatan lukujonoa muokkaamaan etäisyyskentän muotoa. Ylempi koodi toimii tapauksissa, joissa etäisyyskentän muotoa halutaan muuttaa käyttäen yhtä akselia. Alemmassa koodissa muokkaaminen saadaan vaikuttamaan kahdelle halutulle akselille.

Funktion palauttama arvo lisätään etäisyyskentän muodon y-akseliparametriin. Huomattavia asioita ovat, että kyseessä on yksittäinen etäisyyskenttä ja että käytössä on lukujono, jossa on 1024 arvoa, joiden halutaan muokkaavan samanaikaisesti etäisyyskentän muotoa. Lisäksi jokaisen uuden etäisyyskentän lisääminen näkymään tuo huomattavasti rasitusta ohjelman suorituskyvylle. Jos jokaiselle äänidatan arvolle tehtäisiin oma etäisyyskenttä, tämä vaatisi jokaisen etäisyyskentän vertailevan etäisyyksiään toisiinsa. Tässä toteutuksessa rasituksen oletetaan kasvavan lineaarista nopeammin.

10 Projektin toteutuksessa esiintyneet ongelmat

Tässä luvussa kerrotaan projektin toteutuksessa esiintyneistä haasteista.

10.1 Säteenmarssintaan tutustuminen

Ihan aluksi tietoa löytyi videolta, jossa tehtiin säteenmarssintaa sivulla nimeltään shadertoy [21]. Videolla esitetty koodi kotikoneella toistettuna toimi juuri niin kuin pitääkin. Kokeiltaessa, saisiko varjostinkoodin siirrettyä OpenGL-ympäristöön aiemmilla opiskelukursseilla tehdyn harjoitteluprojektin päälle, ilmeni, että shadertoy-sivulle oli kehitetty oma tapa, jolla varjostinkoodissa päästiin käsiksi ikkunan resoluutioon sekä kuinka gl_fragCoord toimi. Niiden vuoksi uv-koordinaatit olivat pahasti virheellisiä. Ohjelman toimiessa ikkuna oli joko täysin

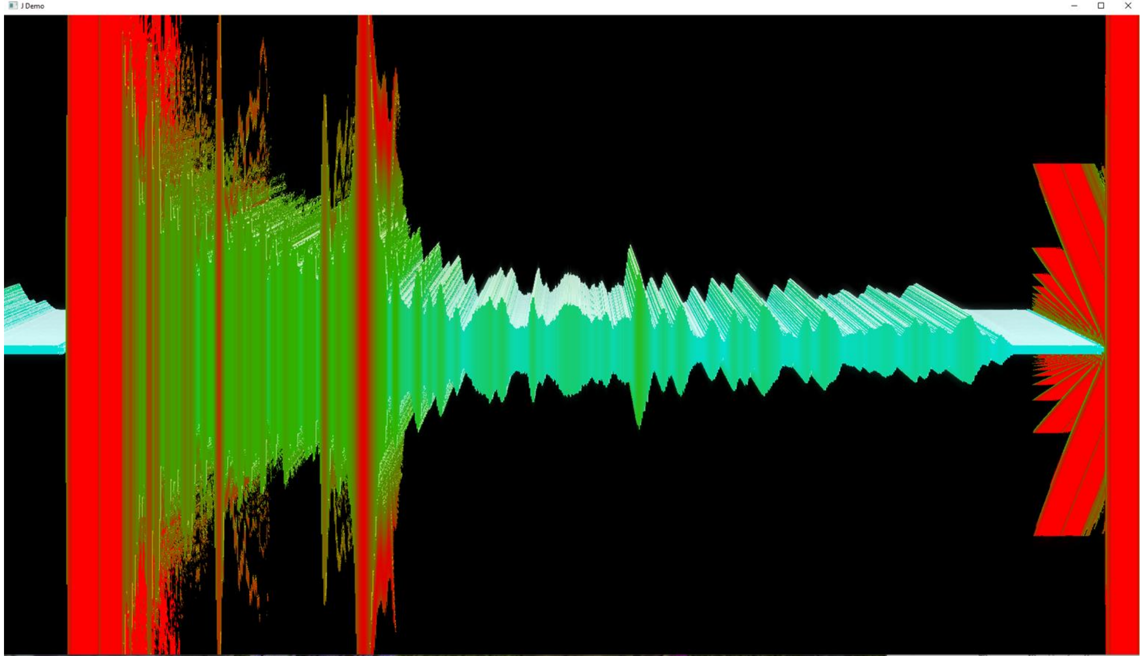
musta tai se välkkyi eri väreillä, joita oli alkuperäisessä varjostimessa. Ensimmäiseksi hämmennyksissäni yritin liikutella virtuaalisen kameran sijaintia, jos se olisi jotenkin kiinni etäisyyskentän pinnassa. Myöhemmin kokeilemalla uv-koordinaattien arvoihin sattumanvaraisia lukuja ilmeni, että etäisyyskenttä oli jonkin verran näkyvissä. Tästä varmistui, että ongelma oli uv-koordinaateissa, verkosta etsien löytyi tieto, mikä on gl_fragcoordin ja ikkunan resoluution yhteys. Tämän jälkeen kokeilemalla eri arvoja uv-koordinaatteihin etäisyyskenttä tuli näkyviin, mutta kaikki oli jostain syystä näytön vasemmassa yläkulmassa. Kuitenkin vähentämällä gl_fragcoordinaateja arvolla 0,5 pystyttiin helposti keskittämään kuva.

Seuraavat suuremmat ongelmat tulivat yritettäessä muokata etäisyyskentän muotoa musiikin tahdissa. Ennen kuin olin päättänyt tehdä työn aiheesta, olin ottanut musiikkitiedostosta muistiin, kuinka paljon ääniraidassa oli iskuja minuutissa. Tällä arvolla sai tehtyä yksinkertaisia etäisyyskenttien muutoksia musiikin tahtiin, mutta ilman muita ääniraidan tietoja, ei kovin kummoisia visualisointeja saa aikaiseksi. Täten tavoitteeksi tuli kerätä kyseisiä lisätietoja. Oli oletus, että käyttämällä jonkinlaista äänikirjastoa lukemaan ääniraitaa ja siirtämällä tiedot varjostimelle käyttöön, tämä toimisi suoraan ilman muuta säätöä. SFML-kirjastolla [10] saatiin musiikkiraidasta erilaisia arvoja tutkittavaksi Visual studion virheenkorjausominaisuuksilla (debuggeri). Arvot olivat kuitenkin hyvin sekavia, ja tämän jälkeen verkosta löytyikin tietoa, kuinka muut musiikin visualisointiohjelmat käyttävät FFT:tä musiikin näytteiden säätämiseen. Kuitenkin FFT:n käyttö oli isompi kynnys, kun asiasta ei ollut mitään aikaisempaa tietoa. Onneksi verkosta löytyi kuitenkin tietoa, kuinka käyttää EigenFFT-moduulia samantapaisessa tavoitteessa kuin omassa projektissa oli.

Lisäksi testailtaessa arvoja ilmeni tapaus, jossa ääniraita meni yhä enemmän pois tahdista musiikin kulkiessa eteenpäin. Ongelma johtui siitä, että vaikka SFML-äänikirjastolla [11] voidaan pyörittää ja lukea ääniraitoja käyttämällä siihen tehtyjä funktioita, oli huomioitava, oliko kyseessä stereo- vai monoraita, ja haettava oikea kohta ääniraidasta tällä tiedolla.

Ensimmäinen ongelma vietäessä FFT:stä saatuja arvoja varjostimelle oli saada ikkunan lukujoukko vastaamaan käytettyjä uniformeja. Alkuperäisessä harjoituksessa oli lähetetty yksittäisiä muuttujia käyttämällä glUniform1f:ää. Käydessä OpenGL-kirjastoa läpi oli alkuoletus, että glUniform1fv olisi jokin vektorimuuttuja, jota käytettäisiin arvojen viemiseen varjostimeen. Asiaa ei paljoa helpottanut se, miten OpenGL:n virallisella khronos.org-sivulla [13] näiden uniformien dokumentaatio oli esitetty. Sen sijaan dokumentaatiossa tuntui olevan järkeä vasta, kun arvojen lähettäminen fragment-varjostimeen oli saatu toimimaan kantapään kautta.

Kun musiikin ääniraidan tiedot ja säteenmarssinta oli saatu toimimaan, täytyi enää selvittää, miten lukujonon ääninäytteet saadaan piirtämään muunnoksia etäisyyskenttien muotoihin. Tiedossa oli jo etukäteen, että rasteroinnissa 1024 objektin pyörittäminen reaaliajassa ei ole luultavasti ongelma. Kuitenkin säteenmarssinnassa jokaisen yksittäisen etäisyyskentän lisääminen näkymään nostaa marssijan rasitusta. Täten asiaa pitää lähestyä eri näkökulmasta. Ratkaisussa piirrettiin yksi etäisyyskenttä, jonka muotoa muutettiin hyödyntämällä näkymän x-akselia ja mod-funktiota. Kaikille primitiivietäisyyskenttämuodoille tämä toteutus ei kuitenkaan ollut paras vaihtoehto, kun etäisyyskenttää piirrettiin eri kiertosunnista. Tapauksissa muodostuu digitaalisia artefakteja, jotka sotkevat tavoiteltua visualisointia (ks. kuva 13). Seuraavaan ratkaisuun valittiin ellipsin muotoinen etäisyyskenttä, koska sillä ei muodostunut digitaalisia artefakteja ja lopputulos oli miellyttävän näköinen. Luultavasti löytyy lisää tapoja, joilla muitakin primitiivietäisyyskenttämuotoja voitaisiin käyttää, jos ongelman lähestymistapa olisi erilainen.

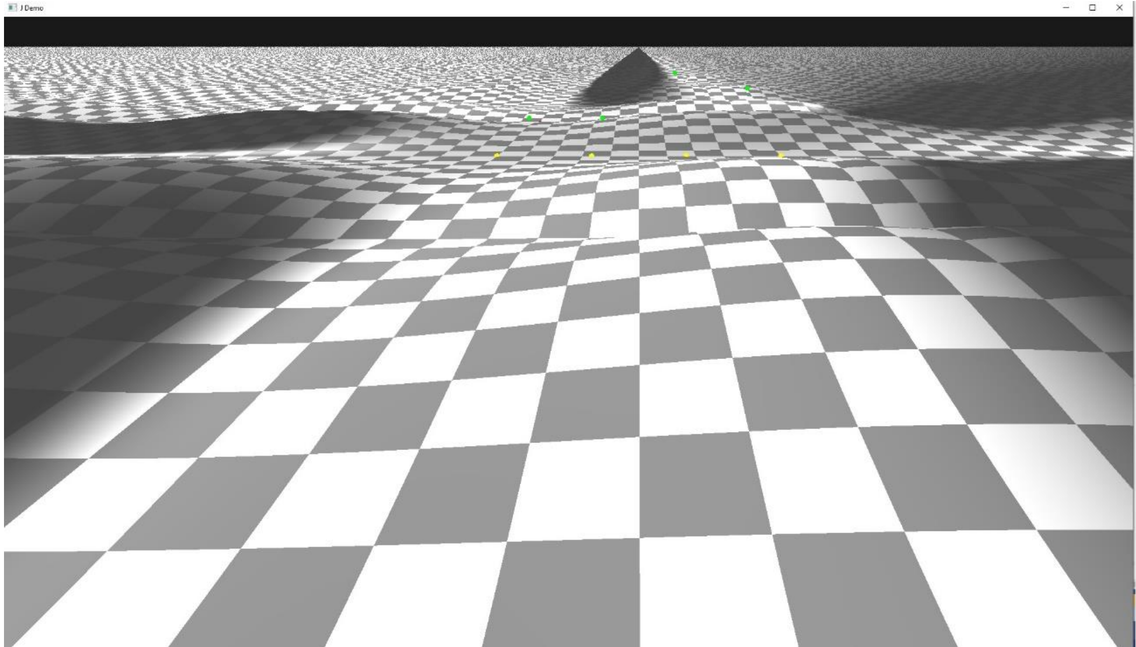


Kuva 13. Aikaisempaan versioon palautettu projekti, jossa digitaaliset artefaktit ovat näkyvissä. Tavoitteena oli saada umpinaiset palkit, mutta tapauksissa, joissa taajuuksien amplitudiarvo nousi näennäisesti sattumanvaraisen kynnyksen yli, etäisyyskenttä ei enää näyttänyt hyvältä ja piti löytää vaihtoehtoinen ratkaisu.

10.2 Kahdelle akselille muuttaminen

Ensimmäisessä prototyyppiprojektissa musiikin visualisointi näytti lopullisessa versiossaan koristeelliselta taajuusspektriltä. Tämän vuoksi seuraavan version tavoitteena oli saada hyödynnettyä enemmän näkymän tilaa käyttämällä myös z-akselia visualisoinnissa. Samaa funktiota, jota käytettiin muuntamaan etäisyyskenttää x-akselin suunnassa, ei voitu käyttää. Oletettavasti ongelma johtui samoista syistä kuin prototyyppiprojektissa esiintyneet graafiset artefaktit, mutta tällä kertaa ongelma esiintyi vielä voimakkaammin.

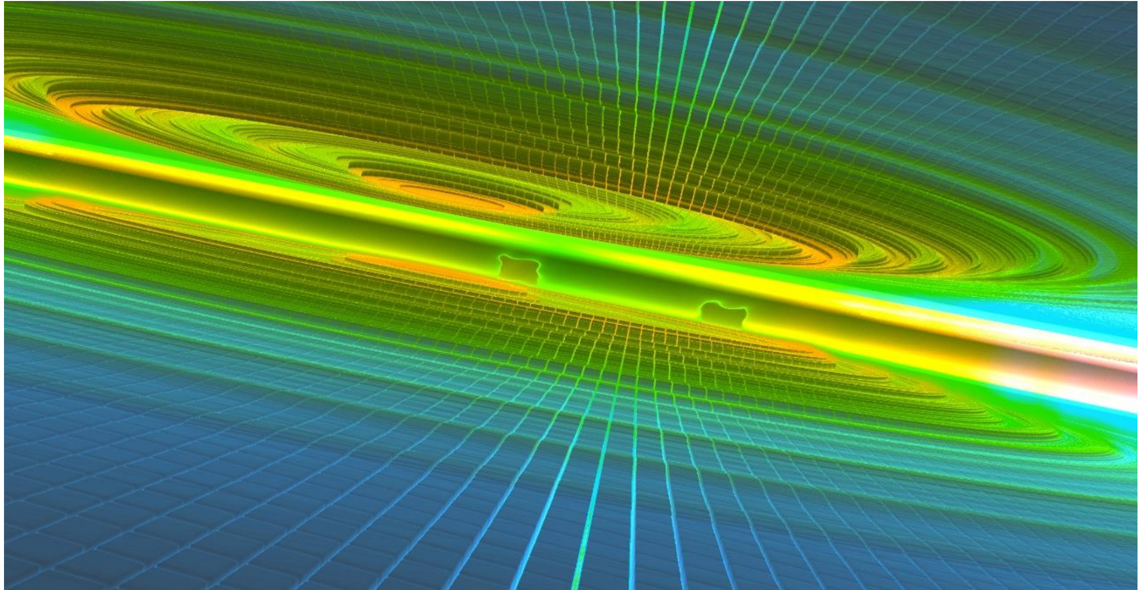
Sen sijaan, että muunnettaisiin etäisyyskenttää sijainnin mukaan, käytettäisiin kenttää, jossa olisi pisteitä, joita liikuttaa y-akselissa musiikin tahtiin. Verkkoaineistosta löytyi toteutus, jossa luotiin niin kutsuttuja kontrollipisteitä, jotka olivat toisissaan kiinni käyttämällä Bézier-käyrää [28]. (Ks. kuva 14.)



Kuva 14. Toteutus, jossa kontrollipisteitä liikutetaan käyttämällä musiikista saatua dataa. Kontrollipisteitä toteutuksessa oli myös hyvin rajoitettu määrä, mikä teki äänidatan hyödyntämisestä hankalaa. Ongelmana olivat erikoiset digitaaliset artefaktit näkymän varjoissa.

Vaikka nyt oli hyvin yksinkertaista laittaa musiikista saatua dataa kontrollipisteiden liikuttamiseen, menetettiin etäisyyskenttien vahvuuksia. Samankaltaisen visualisoinnin voisi luoda ilman säteenmarssintaa käyttämällä rasteroinnissa verteksi-varjostinta samalla, kun suorituskyky olisi huomattavasti parempi.

Tavoitteena oli yhä saada visualisointi, jossa etäisyyskenttää voitaisiin muovata käyttämällä useampaa akselia. Ratkaisun tulisi olla riippumaton kontrollipisteistä, joita käytettiin edellisessä versiossa, sillä ne rajoittivat liikaa sitä, kuinka useaa taajuutta pystyttiin tarkkailemaan samanaikaisesti. Etsittiin, millaisia toteutuksia musiikkivisualisointiin liittyvillä säteenmarssintaa hyödyntävillä varjostimilla oli tehty. Niiden avulla löytyi keino muokata etäisyyskenttää käyttämällä tekstuuria. Tekstuuri puolestaan oli proseduraalisesti generoitu äänestä saadusta datasta. Harmillisesti projekti oli jo venähtänyt niin pitkälle, että tämän toteutuksen opetteleminen veisi vielä enemmän aikaa. Onneksi toteutuksesta sai kuitenkin vinkkiä, miten muokata olemassa olevaa koodia (ks. kuva 15).



Kuva 15. Musiikin visualisoijan versio, jossa etäisyyskentän muotoa muokataan kahdella akselilla. Näkymässä olevat objektit on toteutettu käyttämällä GLSL-varjostinkielen mod-funktiota toistamaan kuutio-primitiivimuodon etäisyyskenttää loputtomasti jokaiseen suuntaan. Tumma osuus näkymän perällä ei johdu valonlähteestä. Se muodostuu säteiden saavuttaessa asetetut maksimiaskelmäärät säteenmarssijan silmukassa, jolloin marssinta katkaistaan.

Lisätavoitteena oli saada lisätä kaksi eriväristä valolähdettä näkymään, jotta visualisointi olisi kiinnostavamman näköinen. Kuitenkin lisättäessä näkymään uusi valolähde selvisi, että tapa, jolla valon lähteet oli toteutettu, teki niistä niin sanotut näkymämaailmanlaajuiset valon lähteet, jolloin valojen etäisyys toisistaan ei vaikuttanut valaistujen objektien kirkkauteen millään tavalla. Lopputuloksena objektien värit sekoittuivat kahden valolähteen väreistä niin, että katsoja ei huomannut näkymässä olevan enempää kuin yksi valon lähde. Lisäksi valon lähteille ei ollut kirjoitettu mitään tapaa heittää varjoja. Valon lähteet vain varjostivat objektien pintoja.

11 Yhteenveto

Vaikka säteenmarssintaa käytetään vielä vähän, on mahdollista, että tekniikan käyttö jatkaa yleistymistä sitä mukaa, kuin uudempien näytönohjaimien laskentateho kasvaa. Myös ohjelmointirajapinnat pelimoottoreissa voivat laskea kynnystä peleihin käyttökelpoisen säteenmarssinnan käyttöä. Säteenmarssinnan

käyttö voi myös hidastua riippuen siitä, kuinka nopeasti uudemmat näytöt kehittyvät ja niiden resoluutiot kasvavat, koska säteenmarssinnassa tarvittavien säteiden määrä kasvaa suhteessa ikkunoiden resoluution mukaan. Tämä johtaa korkeimpiin varjostinkutsuihin ja suorituskyvyn laskemiseen. Nykyisillä 4k-näytöillä koko kuvaruudun kattava säteenmarssinta saattaa olla näytönohjaimelle hyvin raskasta ja visualisoinnin saaminen toimimaan reaaliajassa ilman etukäteisnauhoitusta voi olla vaikeaa. Lisäksi, jos pelissä haluttaisiin käyttää säteenmarssintaa, voi olla vaikeaa arvioida, kuinka raskas kyseinen varjostin on, jos se on vuorovaikutteinen pelaajan liikkeiden kanssa. On oletettavaa, että pelit, jotka hyödyntäisivät säteenmarssintaa, käyttäisivät kuitenkin pääosin rasterointia pelin graafiseen osuuteen ja siihen pitäisi varata suurin osa näytönohjaimen tehosta.

Musiikin visualisointiin säteenmarssinta antaa mahdollisuuksia luoda erilaista graafista jälkeä verrattuna rasterointiin. Lopputulos kuitenkin on enemmän varjostinta koodaavan varassa eikä säteenmarssinnan rajoituksista riippuva. Itse musiikin visualisointiin ei ole väärää eikä oikeaa tapaa kirjoittaa varjostinkoodia. Kun käytetään säteenmarssintaa musiikin visualisointiin, päästään eroon rasteroinnissa käytettyjen verteksien rajoituksista. Muokkaamalla etäisyyskenttiä pystytään tekemään sulavia muutoksia musiikin tahtiin. Lisäksi volumetriset visualisoinnit voivat olla haluttuja missä tahansa peli- tai musiikkisovelluksessa.

Lähteet

1 Hart, John C.; Sandin, Daniel J. & Kauffman, Louis H. 1989. Ray Tracing Deterministic 3-D Fractals. Verkkoaineisto. <https://www.evl.uic.edu/hypercomplex/html/book/rtqjs.pdf>. Luettu 8.11.2022.

2 Fourney, David & Fels, Deborah. 2009. Creating access to music through visualization. Verkkoaineisto. <https://www.researchgate.net/publication/236660379_Creating_access_to_music_through_visualization> Luettu 16.9.2022.

3 Hartmann, Doreen. Computer Demos and the Demoscene. Verkkoaineisto. <https://web.archive.org/web/20140912161731/http://www.isea2010ruhr.org/files/redaktion/pdf/isea2010_proceedings_p11_hartmann.pdf>. Luettu 16.9.2022.

4 Phat Eo.S. rainbow bubble mid3 fuck me dood. Milkdrop. Verkkoaineisto. <https://archive.org/details/md_Phat_Eo.S._rainbow_bubble_mid3-fuck_me_dood>. Haettu 16.9.2022.

5 Quilez, Iniqo. 2021. 3D SDFs. Verkkoaineisto. <https://iquilezles.org/articles/distfunctions/>. Luettu 20.5.2022.

6 Donnelly, William. Per-Pixel Displacement Mapping with Distance Functions. Verkkoaineisto. <https://developer.nvidia.com/gpugems/gpugems2/part-i-geometric-complexity/chapter-8-pixel-displacement-mapping-distance-functions> Haettu 20.5.2022.

7 Toble, Robert F.& Maierhofer, Stefan. 2006. A Mesh Data Structure for Rendering and Subdivision. Verkkoaineisto. <http://wscg.zcu.cz/wscg2006/Papers_2006/Short/E17-full.pdf>. Luettu 17.10.2022.

8 Tutorial of Ray Casting, Ray Tracing, Path Tracing and Ray Marching. 2015. Verkkoaineisto. Starea. <https://blog.ruofeidu.com/tutorial-of-ray-casting-ray-tracing-and-ray-marching/>. Luettu 15.9.2022.

9 Fast Fourier Transformation FFT – Basics. Verkkoaineisto. NTiAudio. <https://www.nti-audio.com/en/support/know-how/fast-fourier-transform-fft>. Luettu 16.9.2022.

10 Jacob, Benoît.; Guennebaud, Gaël. EigenFFT. Verkkoaineisto. <https://eigen.tuxfamily.org/index.php?title=EigenFFT>. Tarkistettu 5.10.2022.

11 Gomila, Laurent. SFML. Verkkoaineisto. <https://www.sfml-dev.org/documentation/2.5.1/group__audio.php>. Tarkistettu 17.10.2022.

12 Fragment Shader. Verkkoaineisto. Khronos Group. Verkkoaineisto. Alfonse. <https://www.khronos.org/opengl/wiki/Fragment_Shader>. Luettu 15.9.2022.

13 OpenGL Overview. Verkkoaineisto. Khronos Group. Alfonse. <https://www.khronos.org/opengl/> Luettu 17.10.2022.

14 SlideServe OpenGL pipeline. Verkkoaineisto. <https://image4.slideserve.com/355438/opengl-pipeline-overview-l.jpg>. Haettu 5.10.2022.

15 OpenGL Atomic Counters. Verkkoaineisto. ARF. <http://www.lighthouse3d.com/tutorials/opengl-atomic-counters/> Verkkoaineisto Luettu 16.9.2022

16 Signed distance fields. Verkkoaineisto. Jasmcole. <https://jasmcole.com/2019/10/03/signed-distance-fields/> Haettu 20.5.2022

17 Steinrucken, Martijn. 2018. Ray Marching for Dummies! Verkkoaineisto. Youtube. <https://youtu.be/PGtv-dBi2wE >. Katsottu 8.6.2022.

18 Quilez, Iniqo. 2021. 2D distance functions. Verkkoaineisto. <https://iquilezles.org/articles/distfunctions2d/>. Luettu 17.9.2022.

19 Quilez, Iniqo. 2014. Voronoise. Verkkoaineisto <https://iquilezles.org/articles/voronoise/>. Luettu 6.6.2022.

20 Quilez, Iniqo. 2002. Domain warping. Verkkoaineisto. <https://iquilezles.org/articles/warp/>. Luettu 12.6.2022.

21 Featured shaders. Verkkoaineisto. Shadertoy. Quilez, Iniqo. <https://www.shadertoy.com/> Haettu 5.10.2022.

22 Lague, Sebastian. 2019. Coding Adventure: Ray Marching. Verkkoaineisto. Youtube. <https://youtu.be/Cp5WWtMoeKg>. Katsottu 8.6.2022.

23 Quilez, Iniqo. 2010. Soft shadows in raymarched SDFs. Verkkoaineisto. <https://iquilezles.org/articles/rmshadows/>. Luettu 6.6.2022.

24 Fenwick, Andy. 2013. Ray-marching soft shadows. Verkkoaineisto. <http://www.polygonpi.com/?p=318>. Luettu 20.8.2022.

25 Assembly Summer 2022 - Shader Showdown finals. 2022. Verkkoaineisto. Youtube. <https://youtu.be/6NQze2qPy4U?t=929>. Haettu 17.8.2022.

26 KissFFT. Verkkoaineisto. Conan. <https://conan.io/center/kissfft>. Haettu 5.10.2022.

27 Sample rate: Which to use? What is the best? 2019. Verkkoaineisto. Magroove. <https://magroove.com/blog/en-us/sample-rate/>. Luettu 8.6.2022.

28 Cubic Bezier Rectangle. 2015. Verkkoaineisto. Demofox. Shadertoy. <https://www.shadertoy.com/view/4tfXz2>. Luettu 31.6.2022.

29 Quilez, Iniqo. 2021. Raymarching SDFs. Verkkoaineisto. <https://iquilezles.org/articles/raymarchingdf/>. Luettu 20.5.2022.

30 Steinrucken, Martijn. 2021. Newton's Cradle: Setting up Materials. Verkkoaineisto. Youtube. <https://youtu.be/Agf188Q8EAc>. Katsottu 8.6.2022.

31 Quilez, Iniqo. 2015. Texturerepetition. Verkkoaineisto. <https://iquilezles.org/articles/texturerepetition/>. Luettu 6.6.2022.

32 Wong, Jamie. 2016. Ray Marching and Signed Distance Functions. Verkkoaineisto. <http://jamie-wong.com/2016/07/15/ray-marching-signed-distance-functions/> Luettu 24.11.2022.

33 Walczyk, Michael. 2021. Ray Marching and Signed Distance Functions. Verkkoaineisto. <https://michaelwalczyk.com/blog-ray-marching.html>. Luettu 24.11.2022.

34 GLSL in Unity. 2021. Verkkoaineisto. Unity Documentation. <https://docs.unity3d.com/Manual/SL-GLSLShaderPrograms.html>. Luettu 8.8.2022.

35 Raymarching Distance Fields: Concepts and Implementation in Unity. 2016. Verkkoaineisto. Adrian Biagioli. <https://adrianb.io/2016/10/01/raymarching.html> 24.11.2022.