Teemu Pullinen

Musiikin visualisointi Unityn Shader Graph -työkalulla

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikan tutkinto-ohjelma

Insinöörityö

29.5.2024

Tiivistelmä

Tekijä:	Teemu Pullinen
Otsikko:	Musiikin visualisointi Unityn Shader Graph -työkalulla
Sivumäärä:	39 sivua
Aika:	29.5.2024

Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Tieto- ja viestintätekniikan tutkinto-ohjelma
Ammatillinen pääaine:	Pelisovellukset
Ohjaajat:	Lehtori Miikka Mäki-Uuro

Insinöörityön tarkoituksena oli tutkia varjostimien luomista Shader Graph -työkalulla, sekä niiden hyödyntämistä musiikin visualisoinnissa. Työ toteutettiin Unity-pelimoottorilla ja C#-ohjelmointikielellä. Varjostimet luotiin visuaalisesti Unityn Shader Graph -työkalulla.

Shader Graph on Unity-pelimoottorin työkalu, jolla pystytään luomaan monimutkaisia varjostimia täysin visuaalisesti kirjoittamatta minkäänlaista koodia. Toimintaperiaate on erilaisten solmujen (engl. node) yhdistely halutun efektin aikaansaamiseksi.

Insinöörityössä päädyttiin toteuttamaan kaksi erilaista varjostinta, jotka reagoivat eri tavoilla musiikkiin. Toteutuksessa keskityttiin myös iskuntunnistukseen musiikissa. Myös erilaisia varjostimien hyödyntämistapoja musiikin visualisoinnissa tutkittiin. Vaikka varjostimet luotiin visuaalisesti, käytettiin varjostimien muuttujien arvojen kontrolloinnissa ohjelmointia.

Lopputuloksena oli kaksi hyvin erilaista varjostinta, joissa lähestytään visualisointia eri tavoin. Kehitetyssä verteksivarjostimessa keskityttiin lähinnä iskuntunnistukseen, kun taas fragmenttivarjostimen kehityksessä tavoitteena oli näyttävä visualisointi, vaikkei se olisikaan reaaliaikaista. Työn tulos osoittaa työkalun soveltuvan musiikin visualisointiin erinomaisesti. Insinöörityöraportista saa hyvän yleiskäsityksen työkalun sekä siihen liittyvien teknologioiden toiminnasta.

Avainsanat: visualisointi, musiikki, varjostimet, Unity

Abstract

Author:	Teemu Pullinen
Title:	Music visualization with Unity Shader Graph
Number of Pages:	39 pages
Date:	29 May 2024
Degree:	Bachelor of Engineering
Degree Programme:	Information and Communications Technology
Professional Major:	Game Applications
Supervisors:	Miikka Mäki-Uuro, Senior Lecturer

The purpose of this final year project was to investigate the production of shaders with Shader Graph, and their utilization in music visualization. The project was developed in the Unity game engine, using the C# programming language. Shaders were created visually with Unity's Shader Graph tool.

Shader Graph is a tool for the Unity game engine, which allows the creation of complex shaders visually. Various nodes are combined to achieve the desired effect.

Two different shaders were developed, which respond to music in different ways. The implementation also focused on beat detection in music. Various ways of utilizing shaders in music visualization were also investigated. Although the shaders were created visually, programming was used to control the values of the shader variables.

The result of the project was two very different shaders, a vertex shader and a fragment shader, which approach visualization in different ways. The results of the project showed that the tool is excellently suited for music visualization.

Keywords: visualization, music, shaders, Unity

Sisällys

Lyhenteet

BPM: *Beats per minute*. Musiikkikappaleen iskujen määrä minuutissa.

HDRP: *High Definition Render Pipeline*. Unityn renderöintiputki, joka on tarkoitettu eritoten moderneille alustoille.

HLSL: *High-Level Shader Language.* Varjostimien ohjelmointiin käytetty ohjelmointikieli.

LTS: *Long Term Support.* Versio ohjelmistosta, jota tuetaan pitkäaikaisesti, usein jopa vuosien ajan.

URP: *Universal Render Pipeline*. Unityn renderöintiputki, joka on yhteensopiva todella monien eri alustojen kanssa.

VR: *Virtual Reality.* Virtuaalitodellisuus. Tietokonesimulaation tuottamien aistimusten avulla luotu keinotekoinen ympäristö.

1 Johdanto

Insinöörityön tavoitteena oli tutkia erilaisia tapoja hyödyntää Unityn Shader Graph -työkalua musiikin visualisointiin. Työssä tutkittiin myös erilaisia käytännön totetustapoja, joiden avulla visualisointi onnistuu Unity-pelimoottorissa.

Työn vaatima ohjelmointi toteutettiin C#-ohjelmointikielellä, jota Unity-pelimoottori käyttää. Vaikka Shader Graph onkin luonteeltaan visuaalinen, eli sen käyttämiseen ei välttämättä tarvitse ohjelmointikokemusta, on musiikin visualisoinnin kannalta kuitenkin useimmiten tarpeellista käsitellä varjostimien muuttujia koodin avulla.

Insinöörityön aiheen valinta pohjautui vahvasti insinöörityöntekijän omaan kokemukseen ja aiempiin aihepiiriin liittyviin projekteihin. Musiikin visualisoinnista Unity-pelimoottorilla on tehty muitakin insinööritöitä, jotka kuitenkin käsittelevät kaikki hieman erilaisia aihealueita ja toteutustapoja.

Luvussa 2 käsitellään musiikin visualisointia yleisesti tutustuen hieman käyttökohteisiin, historiaan ja tekniikoiden kehitykseen. Luvussa 3 tutustutaan tarkemmin Shader Graph -työkaluun sekä erityisesti sen musiikin visualisoinnin kannalta tärkeisiin osa-alueisiin. Luvussa 4 tutkitaan musiikin analysointia Unityn ja C#-ohjelmointikielen avulla. Luku 5 käsittelee insinöörityön toteutusta ja sen lopputulosta.

2 Musiikin visualisointi

Musiikin visualisoinnissa on kyse äänen esittämisestä visuaalisessa muodossa. Visualisoidessa yritetään tarttua eritoten ihmisille helposti kuultavissa oleviin piirteisiin, kuten tempoon, taajuuteen ja melodiaan. Visuaalinen esitys usein sykkii, virtaa tai liikkuu musiikin tahdissa tarjoten kuuntelijalle entistä mukaansatempaavamman kokemuksen.

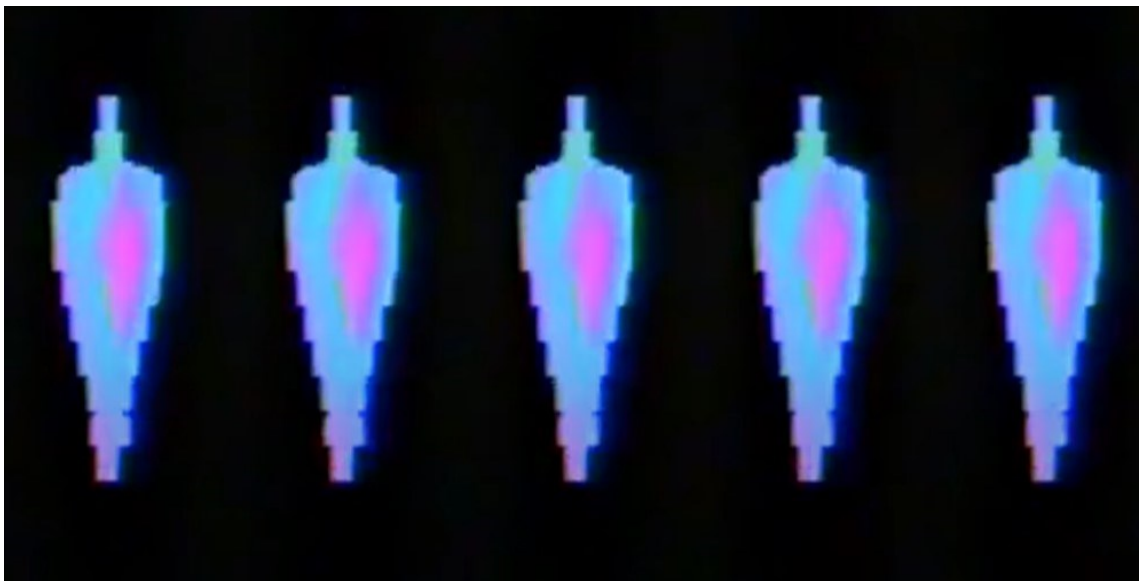
Käyttökohteita, joihin moni on saattanut törmätä, ovat esimerkiksi tietokoneen median soittamiseen tarkoitetut ohjelmistot sekä YouTuben kaltaisiin suoratoistovideopalveluihin ladattujen musiikkikappaleiden visuaaliset esitykset.

Nykyään saatavilla on useita palveluita, joissa käyttäjä voi ilmaiseksi tai pientä maksua vastaan ladata haluamalleen kappaleelle näyttävän visualisaation. Tunnettuja palveluita ovat esimerkiksi renderforest.com sekä musicvid.org.

2.1 Historia

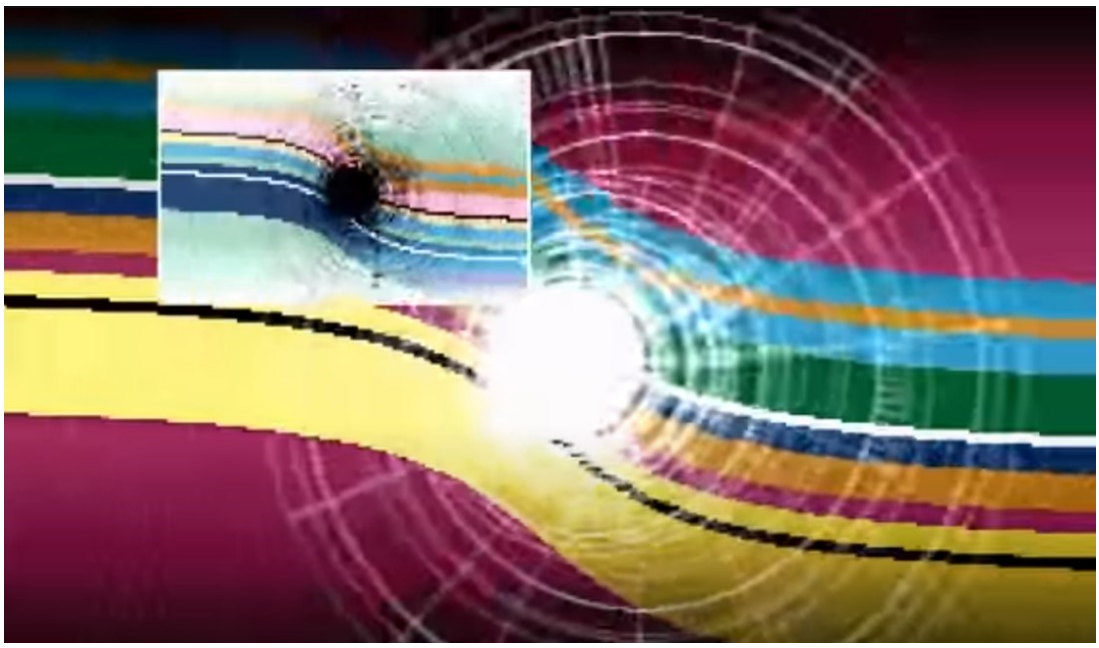
Musiikin eri osa-alueiden ilmaiseminen visuaalisin keinoin juontaa juurensa vähintäänkin 1500-luvulle saakka. Tuolloin taidemaalari Giuseppe Arcimboldo alkoi tutkimaan tapoja, joilla nuotit voisi esittää väreinä. Arcimboldon ratkaisu erosi useiden ajattelutavasta siten, että hän käytti tummia värejä kuvaamaan korkeita nuotteja ja vaaleita värejä kuvaamaan matalia nuotteja. [1; 2.]

Ensimmäinen elektroninen musiikin visualisointia varten kehitetty laite oli Atari Video Music, jonka kehitti amerikkalainen Atari. Laitteen pystyi kytkemään televisioon, jolloin näytöllä näkyi erilaisia liikkuvia kuvioita, jotka reagoivat musiikkiin. Kuvassa 1 näkyy laitteen luomaa visualisaatiota. Tavoitteena oli tehdä laite, joka täydentäisi hifikaiuttimien omistajien kuuntelukokemusta.



Kuva 1. Atari Video Music -laitteen luomaa yksinkertaista grafiikkaa [3].

1980- ja 1990-luvujen vaihteessa visualisointia vei eteenpäin demoskenen kehittyminen. Demoskene on alakulttuuri, jossa käyttäjät luovat niin kutsuttaja demoja. Nämä demot ovat graafisia efektejä ja musiikkia sisältäviä tietokoneohjelmia [4]. Kuvassa 2 näkyy esimerkki värikkäästä erilaisia muotoja sisältävästä demosta. Demoskenen vaikutus ulottui myös pelimaailmaan, sillä esimerkiksi suomalaisen Remedy-peliyrityksen perustajat kuuluivat Future Crew -demoryhmään, ja demoskene oli mukana myös Remedyn ensimmäisen pelin ideoimisessa. Suomalainen demoskene on myös osa kansallista Elävän perinnön kansallista luetteloa. [5.]



Kuva 2. Acme - 303 DOS -demon grafiikkaa vuodelta 1997 [6].

Tavallisten kuluttajien keskuuteen musiikin visualisointi rantautui vasta 1990-luvun lopulla ja vuosituhannen vaihteessa. Vuonna 1997 julkaistu Winamp-mediasoitinohjelmisto sisälsi tuen käyttäjien ohjelmoimille liitännäisille. Näistä liitännäisistä musiikin visualisointiin tarkoitettu MilkDrop nousi suureen suosioon.

2.2 Käyttö kuulovammaisten apuna

Musiikin visualisointia voidaan hyödyntää myös kuulovammaisten musiikin ystävien tai muusikoiden apuna. Kuulovammaiset muusikot hyötyvät soitinten värinästä ja pienistä visuaalisista muutoksista. Näin ollen monet kokevat elektronisten soitinten soittamisen vaikeammaksi ja saattavat hyötyä suuresti visuaalisesta sekä haptisesta palautteesta [7].

Visualisoinnista voi olla apua myös kuulovammaisten kokemuksen parantamisessa esimerkiksi keikoilla ja musiikkia sisältävissä tapahtumissa. Musiikin muuntaminen valoksi ja haptisiksi tuntemuksiksi toisi täysin uuden ulottuvuuden kuulovammaisen tapahtumakokemukseen. Samaa teknologiaa voidaan kuitenkin hyödyntää myös kuulovammaisten jokapäiväisen elämän ja turvallisuuden

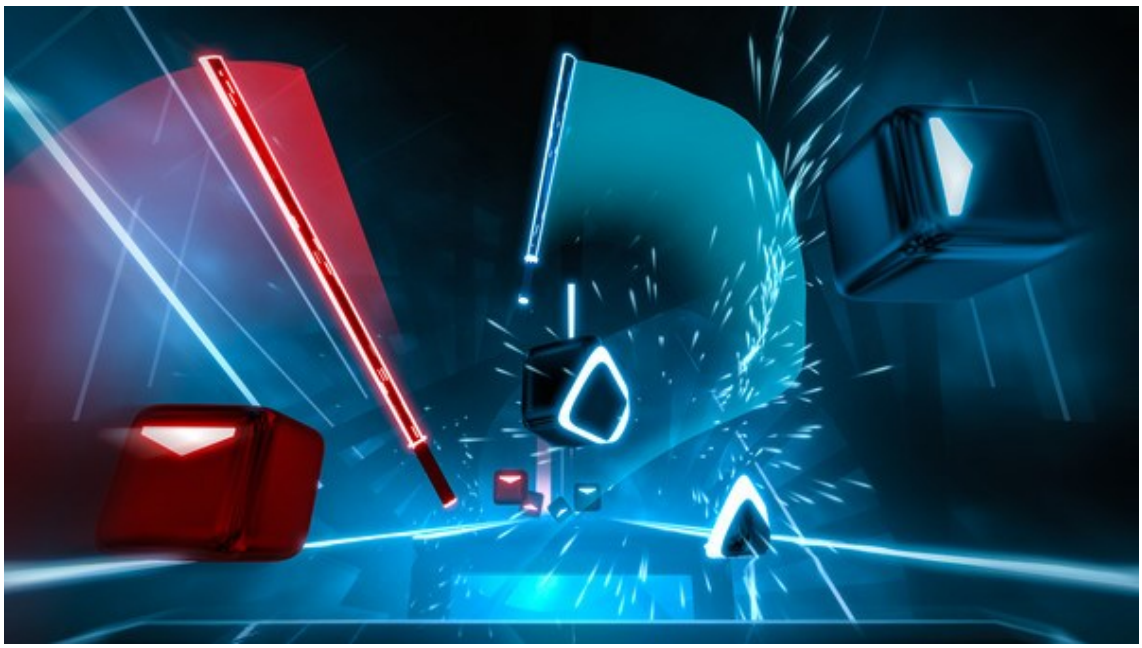
parantamiseksi. Visuaalisesti tai tuntoaistilla välitetty palaute voi korvata esimerkiksi ovikellon tai palohälyttimen äänen. [8.]

Musiikin visualisoinnilla voidaan myös opettaa kuulovammaisille lapsille ääneen liittyviä konsepteja, joita voisi muuten olla vaikea hahmottaa. Tästä hyvänä esimerkkinä toimii New Yorkissa sijaitseva Cooper Union -korkeakoulu, joka toteutti kuulovammaisille suunnatun installaation, jossa lapset pääsevät tutkailemaan erilaisia ääniä valon ja värähtelyjen avulla. [9.]

2.3 Musiikin visualisointi ja analysointi videopeleissä

Musiikin visualisointi tai sen sitominen erilaisiin elementteihin on melko yleistä videopeleissä. Yleisin esimerkki tästä on rytmipelit, jossa pelaajan pelikokemus paranee huomattavasti sillä, että pelin vaatimat toiminnot ja musiikki vastaavat toisiaan. Rytmipeleissä ei kuitenkaan useimmiten käytetä apuna musiikin automatisoitua analysointia, vaan pelimaailman tapahtumat sovitetaan manuaalisesti musiikkiin.

Esimerkiksi VR-pelissä Beat Saber musiikin iskut näkyvät pelaajalle lähestyvinä kuutioina. Kuitenkin myös Beat Saberissa toiminnot on sovitettu musiikkiin käsin sen sijaan, että ne olisivat automaattisesti analysoituja. Kuvassa 3 näkyy Beat Saber -pelin maailmaa pelaajan näkökulmasta.



Kuva 3. Musiikin tahdissa lähestyviä kuutioita pelissä Beat Saber [10].

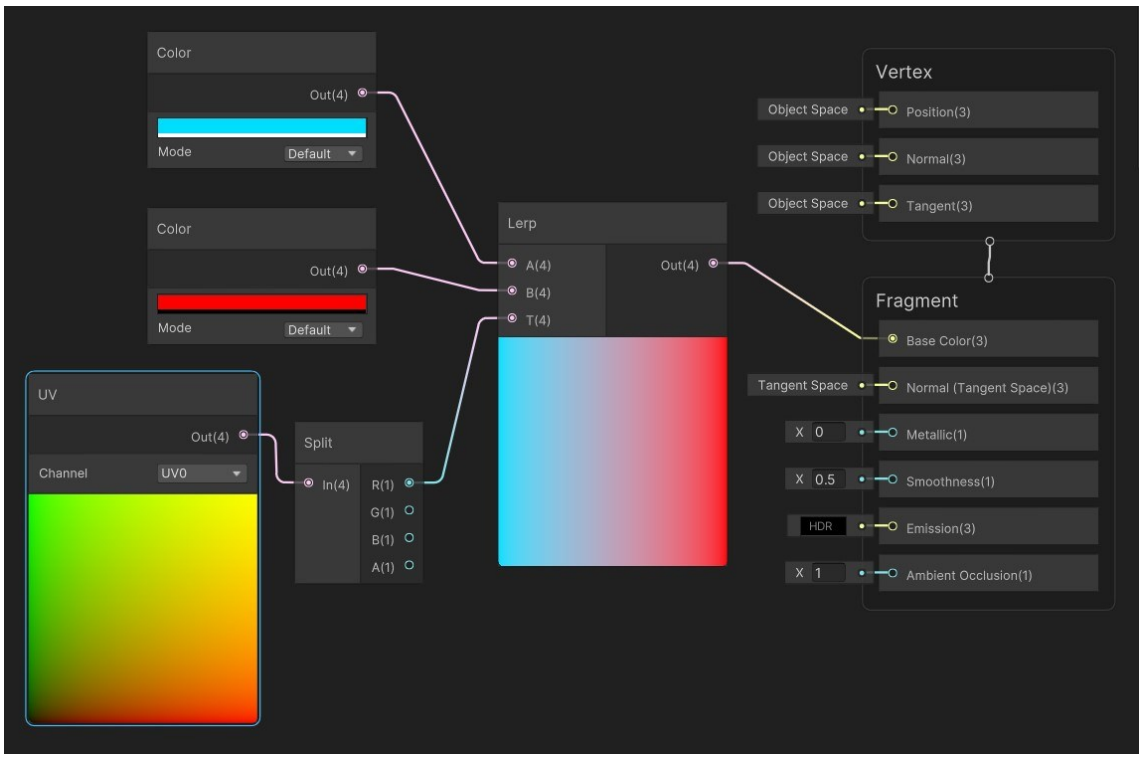
Giant Scam -kehittäjän pelissä Chop It yhdistyy hienosti musiikin analysointi ja sen visuaalinen esittäminen. Pelaaja voi halutessaan käyttää analysointityökalua, joka muuntaa kappaleen rytmipelin tasoksi. Chop it hyödyntää etukäteisanalysointia. [11.]

Hyvä esimerkki täysin erilaisesta lähestymistavasta musiikin visualisoinnin ja pelin yhdistämiseen on peli nimeltään Spectrum Valley. Pelissä pelaaja voi valita haluamansa kappaleen, joka muokkaa koko pelimaailman liikettä. Maa aaltoilee ja puut sykkivät musiikin tahtiin. Muutokset vaikuttavat suoraan pelimaailmaan, eli ärhäkkäästi aaltoilevan kadun päällä ajavat autotkin lentelevät sinne tänne. Kuvassa 4 näkyy esimerkki tilanteesta, jossa pelimaailma aaltoilee musiikin vaikutuksesta.

Kuva 4. Musiikin vaikutuksesta aaltoileva pelimaailma pelissä Spectrum Valley [12].

3 Shader Graph

Shader Graph on Unityn sisäinen työkalu, jolla voi luoda varjostimia visuaalisesti (ks. kuva 5). Tavallisista varjostimien tekotavoista poiketen, ohjelmointiosaaminen ei siis ole välttämätöntä [13]. Jos ohjelmointia ei halua hyödyntää ollenkaan, on kuitenkin valittava varjostin, jonka toimintaan ei tarvitse kajota ajonaikaisesti lainkaan. Hyviä esimerkkejä ovat esimerkiksi huojuvat oksat ja ruoho, tai rauhallisesti aaltoileva vesi. Tilanteissa, joissa varjostimen toimintaa on mukautettava ajonaikaisesti, tulevat ohjelmointitaidot usein tarpeeseen. Näissä tapauksissa Shader Graphin kautta luotuja muuttujia voidaan käyttää suoraan koodissa, kunhan ne asetetaan näkyviksi "exposed" -valinnalla.



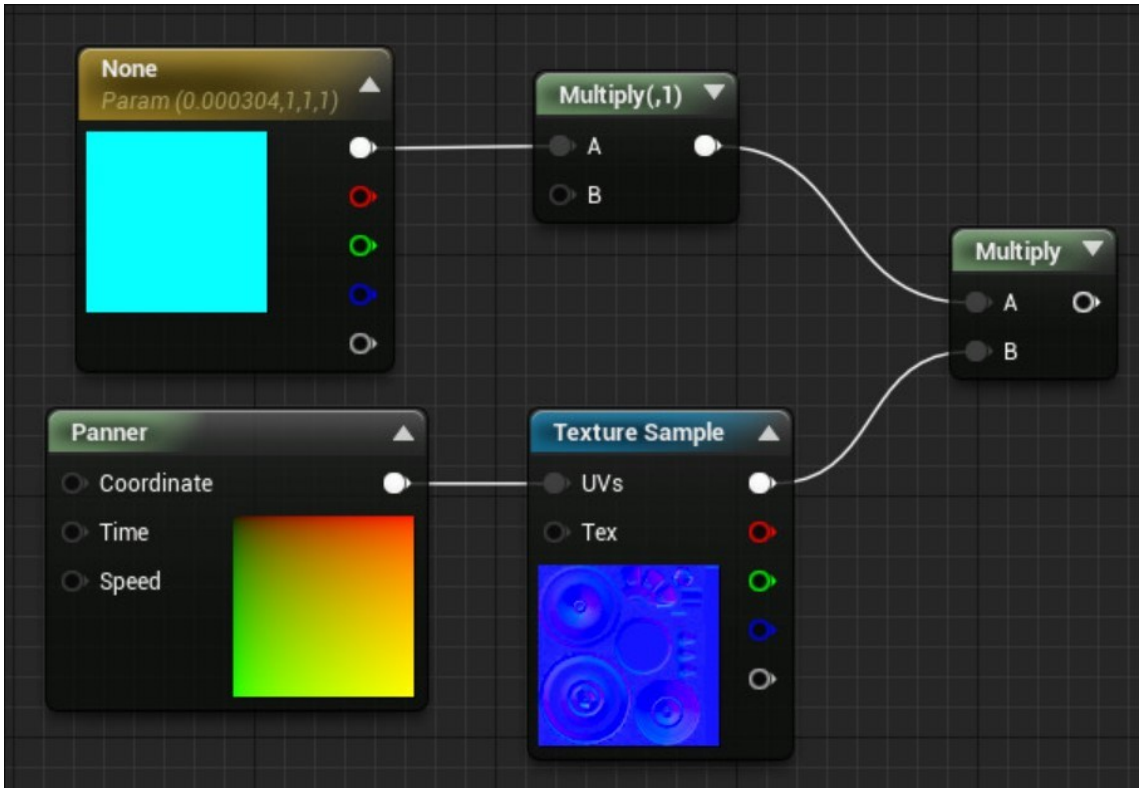
Kuva 5. Yksinkertainen Shader Graph -työkalulla luotu varjostin.

Ensimmäiset Shader Graphia tukevat Unityn versiot julkaistiin vuonna 2018. Aiemmissa versoissa varjostimien luominen vaati kolmannen osapuolen lisäosan, tai varjostin oli ohjelmoitava itse. Eräs suosituimmista visuaalisista varjostimien kehittämiseen tarkoitetuista lisäosista on Unityn Asset Storesta ostettavissa oleva Amplify Shader Editor. Yhtenä suurena Shader Graphin etuna on, että käyttäjä näkee varjostimen aiheuttamat muutokset materiaaliin reaaliaikaisesti varjostinta työstäessään [14]. Tämä voi nopeuttaa kehitysprosessia suuresti, sillä aikaa vievä ohjelman kääntäminen ja pelinäkymässä testaaminen voidaan useissa vaiheissa jättää kokonaan tekemättä.

Shader Graphillä tehdyt varjostimet koostuvat ”nodeista” eli solmuista, joista jokaisella on oma tehtävänsä. Solmut voivat esittää arvoja, matemaattisia funktioita tai esimerkiksi kuvioita. Solmu voi ottaa monta arvoa sisäänsä ja antaa monta arvoa ulos. Esimerkiksi ”Position” -solmu antaa verteksin koordinaatit, ja x-, y- ja z-koordinaatteja voidaan käyttää erikseen ”Split” -solmun avulla. Vannoutuneemmille varjostimien luojille löytyy myös mahdollisuus luoda omia

solmuja käyttäen ”Custom node” -solmuja. Itse luodut solmut käyttävät C#- ja HLSL -ohjelmointikieliä [15].

Myös monet muut ohjelmistot sisältävät Unityn Shader Graphin kaltaisia ratkaisuja varjostimien tekemiseen. Esimerkiksi Unityn suurimpana kilpailijana pidetty Unreal Engine -pelimoottori tarjoaa käyttäjilleen samankaltaisen visuaalisen käyttöliittymän varjostimien kehittämiseen (ks. kuva 6). Myös lähtökohtaisesti 3D-mallien suunnitteluun ja animoimiseen tarkoitettu Blender sisältää hyvin samankaltaisen toiminnallisuuden.



Kuva 6. Unreal Engine -pelimoottorin Material Editor -käyttöliittymällä luotu varjostin [16].

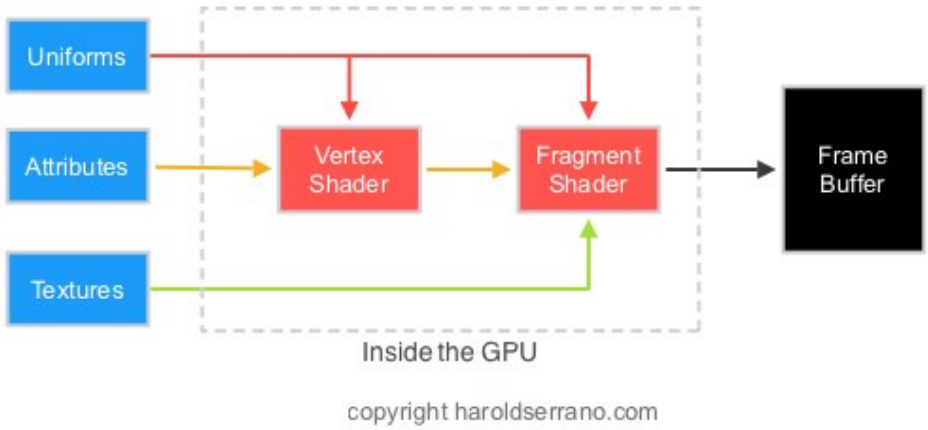
3.1 Varjostimet

Varjostin on ohjelma, joka suoritetaan yleensä näytönohjaimella. Varjostimet ovat usein lyhyitä ohjelmia, jotka ovat melko nopeita suorittaa. Varjostimet on kuitenkin usein suoritettava hyvin suurelle ryhmälle kohteita eli useimmiten vertekseille ja näkyviin renderöitäville pikseleille.

Varjostimia on monia eri tyyppejä. Näistä tärkeimpinä voi pitää verteksi- ja fragmenttivarjostimia. Verteksivarjostimilla voidaan vaikuttaa näkymän geometriaan, kameraan ja projektioon. Fragmenttivarjostimilla voidaan sen sijaan vaikuttaa geometrian ulkonäköön, eli väreihin, valaistukseen ja tekstuureihin. [17.]

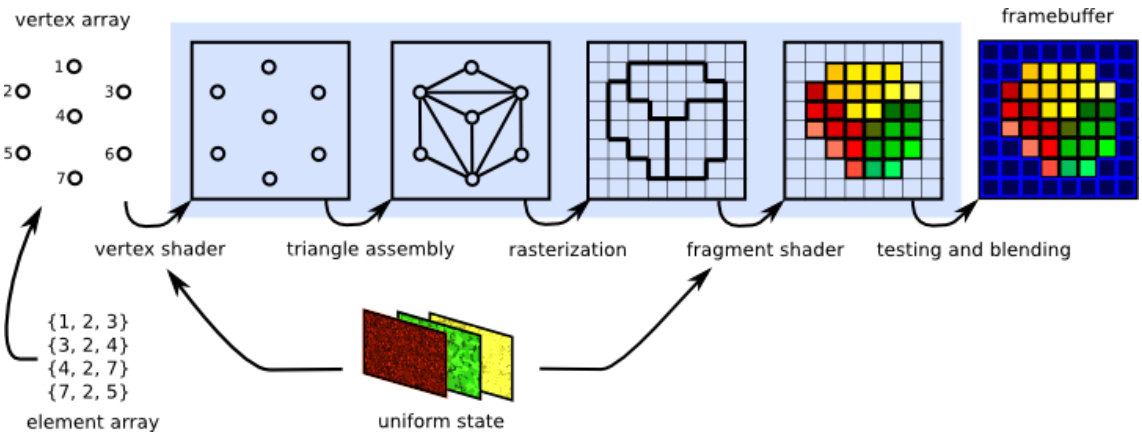
Verteksivarjostimien tapauksessa ohjelma, eli varjostin, suoritetaan jokaiselle verteksille. Verteksivarjostimet ovat näin ollen vastuussa objektin oikeasta paikasta, rotaatiosta ja koosta. Fragmenttivarjostimet suoritetaan jokaiselle fragmentille, jotka ovat näkyvissä ruudulla. Fragmenttivarjostimia kutsutaan myös pikselivarjostimiksi, ja yksinkertaistetusti voikin sanoa, että fragmenttivarjostimissa varjostin suoritetaan jokaiselle renderöitävälle pikselille, vaikka fragmentti ja pikseli eivät olekaan tarkalleen sama asia. [18.]

Uniformit ovat globaaleja muuttujia, joita varjostin ei voi muokata. Uniform-nimitys johtuu siitä, että muuttujan arvot eivät muutu yhden kutsun aikana, vaan ne ovat sen ajan yhdenmukaisia (engl. uniform). Koska uniformit ovat globaaleja muuttujia, pystyy mikä tahansa varjostin käyttämään uniform-muuttujan arvoa. Kuten kuvassa 7 näkyy, muuttumattomat uniform-muuttujat ovat kaikkien varjostimien käytettävissä, toisin kuin esimerkiksi uv-koordinaatit, jotka ovat luettavissa vain verteksivarjostimille. Uniform-muuttujat myös pitävät arvonsa, kunnes niitä muutetaan tai ne nollataan. [19.]



Kuva 7. Uniform-muuttujien toimintatapa varjostimissa [20].

Renderöintiputki on lista vaiheita, joiden avulla esimerkiksi pelin kolmiulotteinen maailma muutetaan näytölle renderöitävään muotoon. Renderöintiputkia on erilaisia, mutta ne muistuttavat suurilta osin toisiaan. Renderöintiputken toiminta voidaan jakaa karkeasti kolmeen vaiheeseen: sovellukseen, geometriaan ja rasterointiin. Kuvassa 8 näkyy tarkempi esitys renderöintiputkesta.

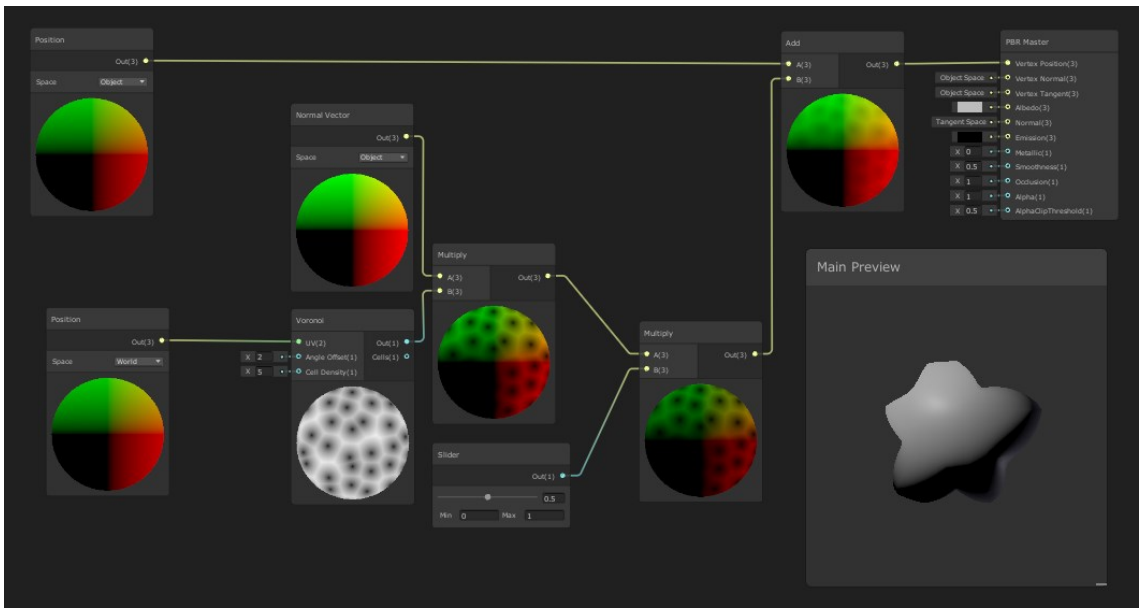


Kuva 8 OpenGL-rajapinnan käyttämä renderöintiputki [21].

3.2 Verteksivarjostimet Shader Graphissä

Verteksi (engl. vertex) on kärkipiste, jossa kolmion kaksi sivua kohtaavat. Näin ollen jokainen kolmio koostuu kolmesta verteksistä. Verteksejä käytetään geometristen muotojen määrittelyyn kolmiulotteisessa tilassa, jonka takia niillä on oltava x-, y- ja z-koordinaatit. Graafista piirtämistä varten vertekseille voidaan antaa myös muita ominaisuuksia, kuten värejä ja tekstuureja. Varjostimet muokkaavat vain näiden ominaisuuksien arvoja eivätkä ominaisuuksia itsessään [22].

Verteksien arvojen muokkaaminen varjostimilla avaa ovet todella näyttävien varjostimien tekemiseen, sillä kyse ei ole vain objektin tekstuurista, vaan myös sen muotoa ja liikettä voi muovata mielensä mukaan. Esimerkki erään verteksivarjostimen vaikutuksesta pyöreään muotoon näkyy kuvan 7 oikeassa alanurkassa.



Kuva 9. Esimerkki yksinkertaisesta Shader Graph -työkalulla toteutetusta verteksivarjostimesta [23].

Shader Graph tarjoaa monia näppäriä solmuja, joiden avulla muodon verteksien kontrollointi helpottuu. Verteksin paikkaa muuttaessa on käytettävä Shader Graphin ”Position” -solmua, joka sisältää käsiteltävän verteksin koordinaatit, eli paikan tilassa. Näitä koordinaatteja voidaan kontrolloida lisäämällä tai kertomalla, tai esimerkiksi kohinan, kuten Voronoi-kohinan avulla. Myös erilaisille kohinoille löytyvät omat solmunsa, joita käyttäjä voi hyödyntää suoraan. Viimeiseksi muokattu ”Position” -arvo yhdistetään varjostimen ”Position” -kohtaan, ja yksinkertainen verteksivarjostin on valmis. [23; 24.]

Edellä mainittuja keinoja käyttämällä voidaan saada aikaan toimivia varjostimia, tai ainakin niiden prototyyppejä todella lyhyessä ajassa, vain muutamaa solmua hyödyntämällä. Esimerkiksi yksinkertaisen aaltoilevan meren voi saada aikaan hetkessä, vaikkapa muuttamalla verteksin koordinaatteja siniaallon avulla, jolle löytyy myös oma solmunsa.

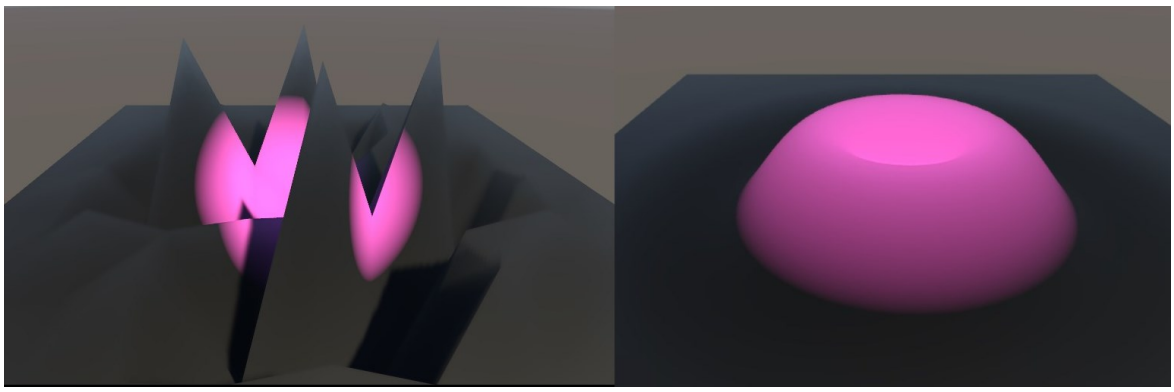
3.3 Unityn renderöintiputkien erot

Unity-pelimoottori sisältää monia erilaisia renderöintiputkia. Renderöintiputken voi valita suoraan tehdessään Unity-projektin, mutta sen voi vaihtaa myös jälkeenpäin. Käyttäjän on kuitenkin hyvä tietää, että näiden renderöintiputkien välillä on eroja, jotka vaikuttavat myös Shader Graphin kanssa työskentelyyn.

Shader Graph -työkalun kanssa työskennellessä valinta on tehtävä HDRP- ja URP-renderöintiputkien välillä. URP:n suurin vahvuus on sen tuettujen alustojen määrä. Sen sijaan HDRP tarjoaa enemmän, mitä tulee valaistukseen ja varjoihin. [25.]

Paras vaihtoehto puhtaasti Shader Graphin kannalta on HDRP-renderöintiputki, joka mahdollistaa kaikkien käytettävissä olevien toimintojen käytön. URP-renderöintiputkesta puuttuu monia solmuja (engl. node), jotka ovat käytettävissä HDRP:ssä. Esimerkiksi Diffusion profile- ja Emission exposure -solmut puuttuvat URP:stä täysin. Näin ollen ennen renderöintiputken valintaa olisi hyvä kartoittaa toiminnot, joita tulee varmasti tarvitsemaan projektissaan.

Lisäksi URP:stä puuttuu esimerkiksi tesselaatio, joka vaikeuttaa Shader Graphin kanssa työskentelyä. Tesselaation avulla on mahdollista käyttää malleja, joissa on verrattain vähän verteksejä, sillä tesselaation avulla niitä voidaan lisätä tarvittaessa. Tämä tulee tarpeeseen Unityn omien 3D-mallien, kuten pallojen ja tasojen kanssa työskennellessä, ja ilman tesselaatiota onkin lähes välttämätöntä tehdä erilliset suuremman tarkkuuden 3D-mallit, jotta varjostin pääsee oikeuksiinsa.



Kuva 10. Sama aaltokuvioita synnyttävä varjostin matalan (vas.) ja suuren tarkkuuden (oik.) 3D-mallilla.

Kuva 10 havainnollistaa hyvin, kuinka matalan tarkkuuden malli ei pysty synnyttämään haluttua efektiä käytettävissä olevien verteksien vähäisen lukumäärän takia.

3.4 Varjostimien parametrien hallitseminen

Vaikka on totta, ettei Shader Graphin käyttämiseen välttämättä tarvitse ohjelmointitaitoja sen visuaalisen luonteen takia, laajentaa ohjelmoinnin käyttäminen kuitenkin mahdollisuuksia suuresti. Tämän lisäksi arvojen muuttaminen ajonaikaisesti on hyvin yksinkertaista, joten kokematonkin ohjelmoija saa muutettua varjostimensa arvoja vain yhdellä koodirivillä.

Arvot eivät muutu varjostinkohtaisesti vaan materiaalikohtaisesti. Eli jos halutaan, että eri objekteilla on sama varjostin, joiden sisäiset arvot ovat kuitenkin eriävät, on jokaiselle objektille luotava oma materiaalinsa, jonka varjostimeksi on valittu haluttu Shader Graphilla luotu varjostin.

Nimet, joilla muuttujiin viitataan koodissa, voidaan tarkastaa suoraan varjostimesta. Lähtökohtaisesti Unity vain lisää muuttujan nimen eteen alaviivan, eli esimerkiksi muuttujaa ”speed” käsitellään koodissa viittaamalla siihen ”_speed” -nimellä.

On hyvä pitää mielessä, että arvot, joita saadaan esimerkiksi ääntä analysoimalla, joudutaan usein vielä käsittelemään ennen niiden syöttämistä varjostimeen mielekkään efektin aikaansaamiseksi. Esimerkkikoodissa 1 ”newValue” kuvaa uutta arvoa, joka muuttujalle asetetaan. Muut muuttujatyypit toimivat intuitiivisesti saman kaavan mukaan. Jos oletetaan, että kyseessä on float -tyyppinen muuttuja, voidaan sen arvo muuttaa seuraavan koodiesimerkin mukaisesti.

```
SetFloat (_speed, newValue);
```

Esimerkkikoodi 1. Esimerkki varjostimen liukulukutyyppisen muuttujan arvon muuttamisesta

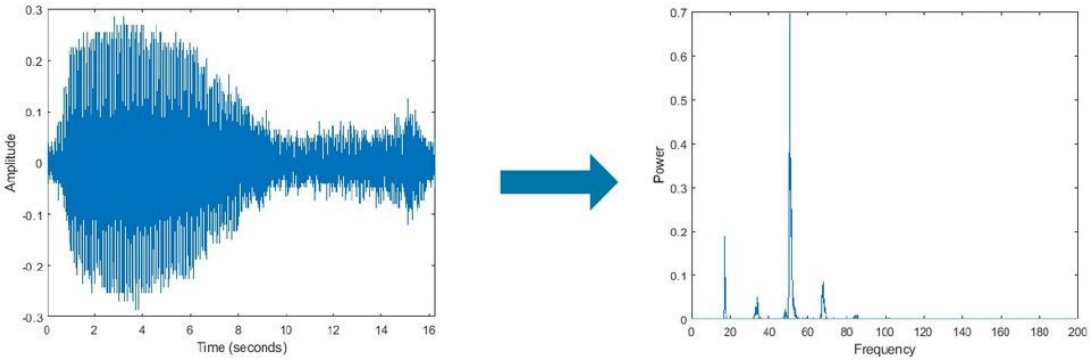
Parametreja voidaan käsitellä näppärästi myös ilman ohjelmointia. Esimerkiksi Unityn Timeline-työkalun avulla arvoja voidaan muuttaa haluttuina ajanhetkinä. Prosessi on melko työläs eritoten pitkien musiikkileikkeiden visualisointiin, mutta täysin toimiva vaihtoehto yksittäisiin kappaleisiin. Arvojen muuttaminen käsin antaa myös suuremman vapauden räätälöidä varjostimen liikkeitä musiikin mukaan, ja ihmisen on helppo poimia musiikista yksityiskohtia, joita koneellisella analysoinnilla olisi vaikeata löytää.

4 Musiikin analysointi Unityssä

Shader Graphin visuaalisesta luonteesta poiketen musiikin analysointi Unity-pelimoottorilla vaatii jo hieman ohjelmointikokemusta. Käyttäjää on kuitenkin helpottamassa Unityn sisäänrakennettuja metodeja, joilla äänestä saadaan irti hyvin paljon tietoa.

Tärkein näistä metodeista on ”GetSpectrumData()”, joka muuntaa äänen pieniksi näytteiksi (engl. sample), jotka sisältävät taajuuksia ja amplitudeja. Metodi toimii käyttämällä Fast fourier transform (FFT) -algoritmia, jonka avulla voimme tarkastella tietyllä aikavälillä tallennettua dataa eri taajuuksilta. Algoritmi muuntaa siis datan, jossa on eri amplitudeja eri ajankohdissa, dataksi, jossa on eri amplitudeja eri taajuuksilla (ks. kuva 10). Ilman kykyä tarkastella eri taajuuksien tai taajuusryhmien amplitudeja, olisivat myös mahdollisuudet visualisointiin

melko heikot. Näin ollen FFT-algoritmia voidaan pitää yhtenä visualisoinnin kulmakivenä. [26; 27.]



Kuva 11. Ääniraita sinivalaan ääntelystä muunnettuna FFT-algoritmilla [28].

GetSpectrumData on Unityn ”AudioSource” -nimisen komponentin metodi, eli ensin täytyy luoda AudioSource, joka soittaa äänen. Tämän jälkeen tarvitaan myös AudioSourcelle annettava ”AudioClip”, eli ääniraita, joka soitetaan.

GetSpectrumData-metodin toimintaan voidaan vaikuttaa argumenteilla. Koodiesimerkissä 2 sulkeiden sisältä löytyvät argumentit, jotka metodi tarvitsee. Argumenteista ensimmäinen, ”samples” on liukuluvuista (engl. float) koostuva lista, johon taajuusalueilta otetut näytteet tallennetaan. Järkevintä on käyttää etukäteen luotua listaa, jonka metodi täyttää automaattisesti täyteen näytteitä. Näin ollen listan alkioiden määrällä määritetään myös se, kuinka pieniin osiin spektri jaetaan. ”Channel” on kanava, jolta näyte otetaan, ja ikkuna (engl. window) on käyttäjän valitsema FFTWindow, joita on monia erilaisia [29]. FFTWindowin valinta vaikuttaa siihen, kuinka spektridata lasketaan. Lisäksi ikkunan valinnalla voidaan vähentää arvojen vuotoa taajuusalueesta toiseen.

```
public void GetSpectrumData(float[] samples, int channel, FFTWindow
window);
```

Esimerkkikoodi 2. GetSpectrumData-metodin esittely

Näytteiden oton jälkeen käytettävissä on lista eri taajuusalueiden amplitudeista, ja dataa voidaan käyttää jo tässä vaiheessa apuna visualisointiin. Usein

näytteet kuitenkin jaotellaan uudelleen ihmisille intuitiivisiin ryhmiin. Yhden näytteen kattama taajuusalue saadaan jakamalla näytenopeus (engl. sampling rate) kahdella ja sitten näytteiden määrällä. Esimerkiksi jos 44100 Hz:n signaali on jaettu 512 näytteen listaan, käsittää listan jokainen näyte noin 43 Hz:n alueen koko spektristä. Esimerkkikoodissa 3 tallennetaan taajuusalueen amplitudit juuri kuvatulla tavalla, mikä antaa metodille 512 alkion listan.

```
float[] spectrum = new float[512];
public void GetSpectrumData(spectrum, 0, FFTWindow.BlackmanHarris);
```

Esimerkkikoodi 3. Koodipätkä luo liukulukulistan ja täyttää sen näytteillä. FFTWindowiksi on valittu BlackmanHarris-algoritmi.

Ennen projektin aloittamista ajatusvirheiden välttämikseksi on järkevää tarkastaa, että oikeat taajuudet aktivoivat oikeat taajuusalueet. Tämä onnistuu esimerkiksi lataamalla kuulon, tai kuulokkeiden testaamista varten tehdyn äänileikkeen, jossa kunkin ajanhetken tarkka taajuus on helppo tarkistaa.

16	17	18	19	20
0.0287	0.1313	0.1797	0.0867	0.0097

Kuva 12. 800 Hz:n ääninäytteestä otettu GetSpectrumData.

Kuvasta 11 nähdään, että otetuista 512 näytteestä toimintaa on odotetusti eniten alueilla 17 ja 18.

4.1 Ajonaikainen analysointi

Suoraviivaisin tapa analysoida ääntä on tehdä se ajonaikaisesti. Tähän liittyy kuitenkin joitakin ongelmia, eritoten käytössä olevan informaation suhteen. Pitempiaikaista analysointia voi vaatia esimerkiksi kappaleen BPM:n, eli iskujen minuutissa määrittäminen, tai eri taajuusalueiden keskimääräisten amplitudien laskeminen.

Ajonaikaisella analysoinnilla voidaan tarkoittaa mikrofonista saadun tai esimerkiksi internetselaimessa suoratoistettavan musiikin analysointia. Määritelmän täyttää myös Unityyn tuodun äänitiedoston analysointi niin, ettei sitä ole käyty läpi etukäteen. Tilanteet ovat kuitenkin käytännössä hieman erilaisia.

Mikrofonin kautta tulevan äänen käyttäminen Unityssä on melko vaivatonta, kuten nähdään koodiesimerkissä 4. Mikrofonin voi asettaa äänilähteeksi suoraan, eli koodi ei oikeastaan vaadi muutoksia. Tämän jälkeen mikrofonista saatua äänidataa voidaan analysoida samalla tavalla kuin mitä tahansa Unityyn tuotua ääniraitaa. [30.]

```
_myMic = Microphone.devices[0].ToString();
_source.clip = Microphone.Start(_myMic, true, 10,
AudioSettings.outputSampleRate);
```

Esimerkkikoodi 4. Ensin haetaan aktiivinen mikrofoni, jonka jälkeen se asetetaan äänilähteeksi.

Joitain ajonaikaisen analysoinnin heikkouksia voidaan yrittää kiertää käyttämällä apuna vanhaa dataa. Esimerkiksi niin kutsutulla Spectral flux -algoritmilla voidaan etsiä signaalista huippukohtia jo muutaman kymmenen ruudunpäivityksen aikana, joka saattaa tarkoittaa ajassa vain sekunnin murto-osaa [31]. Joitain vertailuja pystytään tekemään jopa tämänhetkisen ja aiemman ruudun arvojen välillä, aikana, joka ei ole loppukäyttäjän havaittavissa.

Lisäksi on mahdollista käyttää erilaisia kikkoja, jotka piilottavat analysointiin kuluneen aikaviiveen loppukäyttäjältä. Eräs yksinkertainen ratkaisu on käyttää Unityssä kahta äänilähdettä (engl. AudioSource), jotka soittavat ääniraitaa eri kohdissa. Edellä olevaa äänilähdettä käytetään analysointiin, ja jäljessä olevaa musiikin varsinaista soittamista varten.

4.2 Etukäteisanalysointi

Etukäteisanalysoinnilla tarkoitetaan analysointitapaa, jossa ääniraita käydään joko kokonaan etukäteen läpi, tai sitä puskuroidaan niin paljon, että tarvittava data saadaan hankittua. Tämän metodin suurimpana vahvuutena on, että

käytössä on kaikki informaatio, joka koko kappaleesta saadaan irti. Näin ollen esimerkiksi pitkien aikojen keskiarvoja ja huippukohtia voidaan käyttää apuna visualisoinnissa.

Etukäteisanalysoinnissa aikarajoitukset katoavat lähes kokonaan, jolloin prosessointitehosta tai aikavaatimuksista ei tarvitse juurikaan välittää. Unityssä etukäteisanalysointi vaatii kuitenkin hieman enemmän vaivaa, sillä ajonaikaisille metodeille ei löydy suoria tai ainakaan yhtä näppäriä vastineita. Samalla käyttökohteista jotkin supistuvat pois, sillä livemusiikin analysointi ja visualisointi on mahdotonta. Etukäteisanalysoinnissa on kuitenkin myös mahdollista toteuttaa laajempaa, suurempaa prosessointitehoa vaativaa musiikin analysointia, sillä tuloksen ei täydy olla valmiina aina seuraavaa renderöitävää ruutua varten.

Koko ääniraidan datan saa haltuunsa käyttämällä Unityn metodia ” AudioClip.GetData”, jonka vastine ajonaikaisessa analysoinnissa on ” AudioSource.GetOutputData” [32]. Nämä eroavat toisistaan siten, että GetData palauttaa kaikki näytteet koko leikkeen pituudelta. Sen sijaan GetOutPutData palauttaa vain tämän ajanhetken datan. Lisäksi GetOutputData palauttaa stereoäänen muodossa, jossa vasemman (L) ja oikean kanavan (R) data on listattu vuorotellen, R-L-R-L. Jos datasta haluaa yhdenmuotoisen GetData-metodin kanssa, on se muunnettava yksikanavaiseksi laskemalla jokaisen vasen-oikea-parin keskiarvo, joka vaatii paljon prosessointitehoa.

Jo ääniraidan amplitudien aikaleimoilla voidaan saada aikaan jonkinlaista visualisointia. Aiempien esimerkkien kaltaista taajuuksien välistä amplitudien vertailua varten saatu data pitäisi vielä muuttaa muotoon, jossa taajuudet on erotettu toisistaan niin, että niiden amplitudeja voi vertailla. Ajonaikaisesti temppu onnistuu helposti GetSpectrumData-metodin FFT:n avulla, jota ei valitettavasti voi käyttää saatuun dataan. Datan käsittely on kuitenkin mahdollista ulkoisten kirjastojen avulla, jotka sisältävät Fourier Transform -implementaation.

5 Visualisointiohjelman toteutus

Projektissa toteutettiin kaksi hyvin erilaista varjostinta, joita myös hyödynnettiin eri tavoin. Visualisointiin kohinaa hyödyntävä varjostin on fragmenttivarjostin, kun taas 3D-mallien pinnanmuotoihin vaikuttava varjostin on verteksivarjostin. Verteksivarjostimessa muokataan verteksien koordinaatteja, kun taas fragmenttivarjostimella vaikutetaan verteksien välisten pikseleiden tekstuuriin ja väritykseen. Projektissa päädyttiin toteuttamaan molemmanlaiset varjostimet juuri niiden erilaisuuden, ja sen myötä kiinnostavuuden takia.

Työn toteutuksessa käytettiin Unity-pelimoottorin versiota 2021.3.21f. 3.21 on LTS, eli Long Term Support -versio, jota Unity on sitoutunut tukemaan pitkällä aikavälillä. Ohjelmointiympäristönä käytettiin Microsoft Visual Studiota, ja ohjelmointi tapahtui Unityn käyttämällä C#-ohjelmointikielellä. Alun perin projektissa oli tarkoitus käyttää Unityn sisäisiä 3D-malleja, mutta jälkeenpäin ulkoisen mallintamisohjelmiston käyttö osoittautui välttämättömäksi. Tarvittavat 3D-mallit tehtiin ilmaisella Blender-nimisellä kolmiulotteiseen mallinnukseen tarkoitetulla ohjelmistolla.

Samankaltaisen projektin olisi voinut toteuttaa myös muilla pelimoottoreilla, kuten Unreal Enginellä. Unityn valitseminen perustui lähinnä omaan kokemukseen ohjelmiston kanssa. Projektissa käytettiin Unityn omaa versionhallintaa.

Unity-projektia aloittaessa voi valita, haluaako projektissaan käyttää URP- vai HDRP-renderöintiputkea. Projekti aloitettiin käyttäen URP:tä, mutta puuttuvien toimintojen takia renderöintiputki oli vaihdettava myöhemmin HDRP:hen.

Projektia varten luotiin muutamia yksinkertaisia 3D-malleja, lähinnä tasoja (engl. plane) ja palloja. Unity sisältää vastaavat mallit jo itsessään, mutta niiden tarkkuus oli liian alhainen projektin toteutusta varten, josta löytyy esimerkki myös

luvusta 3.3. Tarvittavat korkeamman tarkkuuden 3D-mallit toteutettiin Blender-mallinnusohjelmistolla käyttäen sen Subdivide-työkalua (kuva 13), joka lisää malliin halutun määrän uusia verteksejä.

Kuva 13. Esimerkki Subdivide-työkalun toiminnasta [32].

5.1 Äänen analysointimenetelmät

Äänen analysoinnin perustana oli Unityn GetSpectrumData-metodi, joka antaa tietoa parhaillaan soivan ääniraidan spektristä. GetSpectrumData on myös helpoin tapa toteuttaa ajonaikainen äänen analysointi Unityllä. Koodiesimerkki 5 on esimerkki metodin käytöstä.

```
source.GetSpectrumData(samples, 0, FFTWindow.Blackman);
```

Esimerkkikoodi 5. Työssä käytetty GetSpectrumData-kutsu, jossa ”samples” on 512 liukuluvun lista.

Analysointia varten luotiin etukäteen liukulukulistoja. Suurin 512 alkion lista sisältää kaikki otetut näytteet. Tämä lista jaettiin eteenpäin pienempiin taajuusalueisiin, joita kertyi viisi kappaletta. Taajuusalueisiin jakamisen voisi toteuttaa monella eri tavalla, eikä mikään tapa ole välttämättä toista parempi. Huomioon oli otettava lähinnä oman projektin tarpeet. Tässä työssä muuttuvia arvoja ei tarvittu suurta määrää, joten alueiden määräksi jäi verrattain pienehkö viisi. Alueet yritettiin jakaa ihmiskorvalle intuitiivisella tavalla, tutkimalla monia eri lähteitä.

Projektissa käytetty tapa jakaa taajuudet muistutti kuvaesimerkkiä (kuva 14), joskin projektia varten jakoa pelkistettiin hieman. Työssä taajuudet jaettiin alabassoon, bassoon, alempaan keskialueeseen, korkeampaan keskialueeseen ja diskanttiin. Taajuusalueista jokaiselle laskettiin myös jokaisella ruudunpäivityksellä keskiarvo, jota käytettiin arvioimaan, onko taajuusalueella visualisoitavaa toimintaa. Keskiarvoille luotiin myös metodi, joka tarkistaa, ovatko arvot muuttuneet yli määritetyn raja-arvon verran sitten viime ruudunpäivityksen. Jos muutoksen suuruus ylitti raja-arvon, korotettiin keskiarvoa vain ennalta määritetyn arvon verran. Tällä yritettiin estää liian radikaalien muutosten päätyminen varjostimen muuttujiin, joka olisi voinut aiheuttaa negatiivisesti visuaaliseen kokemukseen.

Kuva 14. Esimerkki eräästä tavasta jakaa taajuudet alueisiin [33].

5.2 Iskuntunnistuksen toteutus

Projektin alkuvaiheessa iskuntunnistus toteutettiin hyvin yksinkertaisesti. Ajatuksena oli rakentaa toimiva pohja, jota voi lähteä kehittämään eteenpäin tarvittaessa. Taajuusalueiden jakaminen on myös iskuntunnistuksessa suuri apu. On huomattavasti helpompaa tunnistaa isku rajatulta alueelta koko spektrin sijasta. Tämä tosin tarkoittaa, että alueista jokainen on silti käytävä läpi ellei tiedä, mistä etsiä.

Ensimmäisessä toteutuksessa iskut tunnistettiin vain muutamalla if-lauseella, joskin hyvin onnistuneesti. Tästä esimerkkinä on esimerkkikoodi 6. Jokaiselle taajuusalueelle asetettiin rajat, joiden ylityttyä ohjelma tulkitsi iskun tapahtuneeksi. Rajojen hienosäätö vaati aikaa, sillä liian alhaiset rajat löytävät musiikista liikaa iskuja, ja liian korkeat jättävät päinvastoin iskuja huomaamatta. Paras suorituskyky saavutettiin lisäämällä iskuntunnistuksen perään lyhyt odotusaika, jottei samaa iskua tunnistettu kahdesti.

```
if (midRangeLoAvg > midRangeLoLimit && !wavecd ||  bassAvg > bassLimit
&& !wavecd)
        {
            wavecd = true;
            LaunchWaveRandomTile();
            StartCoroutine(WaitWaveCD());
        }
```

Esimerkkikoodi 6. Koodilla tarkastetaan, ylittääkö alempi keskialue tai basso ennalta saavutettua raja-arvoa. Ylitys vaikuttaa varjostimeen.

Tietyn musiikkileikkeen kanssa työskennellessä on helpompaa vetää selviä rajoja sille, kuinka kova piikki lasketaan iskuksi. Jos kappaleessa on kuitenkin monia eri osia, joiden välillä keskimääräinen amplitudi vaihtelee suuresti, tulee myös toteutuksesta monimutkaisempaa. Ratkaisuna tähän kokeiltiin aiemmin staattisten raja-arvojen dynaamista muuttamista ajonaikaisesti. Kun musiikin keskimääräinen amplitudi alenee, niin tekee myös raja-arvo ja päinvastoin. Esimerkkikoodi 7 sisältää esimerkin raja-arvon dynaamisesta päivityksestä.

```
void Update
{
     bassLimit = initialBassLimit + (GetAverageValue(bass) *
     bassThresholdMultiplier);
}

float GetAverageValue(float[] spectrum)
    {
        float sum = 0f;

        for (int i = 0; i < spectrum.Length; i++)
        {
            sum += spectrum[i];
        }
        return sum / spectrum.Length;
    }
```

Esimerkkikoodi 7. Update-funktion sisällä kutsutaan metodia, joka muokkaa basson taajuusalueen raja-arvoa nykyisen amplitudin mukaan

Yrityksenä tuoda työhön aiempaa elegantimpi ja helpommin erilaisiin ääniraitoihin mukautuva iskuntunnistus, lisättiin työhön myös Spectral flux-implementaatio. Spectral flux on tapa mitata spektrin muuttumista otantojen välillä, mikä paljastaa esimerkiksi intensiteettipiikkejä. Implementaatio yritettiin pitää melko yksinkertaisena, joten toteutuksessa pitäydyttiin ajonaikaisessa analysoinnissa. Tämä vähensi iskuntunnistuksen tarkkuutta jonkin verran, mutta tulokset olivat silti käyttökelpoisia.

Myös spectral fluxin kanssa käytettiin adaptiivista raja-arvoa, joka muuttui tarkasteltavan taajuusalueen keskimääräisen amplitudin mukaan. Lisäksi toimintavaltaan samankaltaisen, mutta epätarkan viimeisimmän kahden arvon vertailun sijaan toteutukseen lisättiin puskuri, jolla pystyi vertailemaan haluamaansa määrää ruutuja ajassa taaksepäin. Koodiesimerkki 8 sisältää osan projektissa käytetystä Spectral Flux -toteutuksesta ja sen puskurista.

```
void SpectralFlux()
    {
        for (int i = 0; i < 8; i++)
        {
            bassSpectrum[i] = samples[i];
        }
        float flux = CalculateSpectralFlux(bassSpectrum);

        if (flux > threshold)
        {
            Debug.Log("Beat!");
        }
       threshold = initialThreshold + (GetAverageValue(bassSpectrum) *
     thresholdMultiplier);

        prevbassSpectrum = bassSpectrum;
        spectrumBuffer[bufferIndex] = bassSpectrum;
        bufferIndex = (bufferIndex + 1) % bufferLength;
    }

float CalculateSpectralFlux(float[] currentSpectrum)
    {
        float flux = 0f;
        for (int i = 0; i < sampleAmount; i++)
        {
            float diff = Mathf.Max(0f, currentSpectrum[i] - GetSpect-
rumAvg(i));
            flux += diff;
        }
        return flux;
    }
float GetSpectrumAvg(int index)
    {
        float sum = 0f;
        for (int i = 0; i < bufferLength; i++)
        {
            sum += spectrumBuffer[i][index];
        }
        return sum / bufferLength;
    }
```

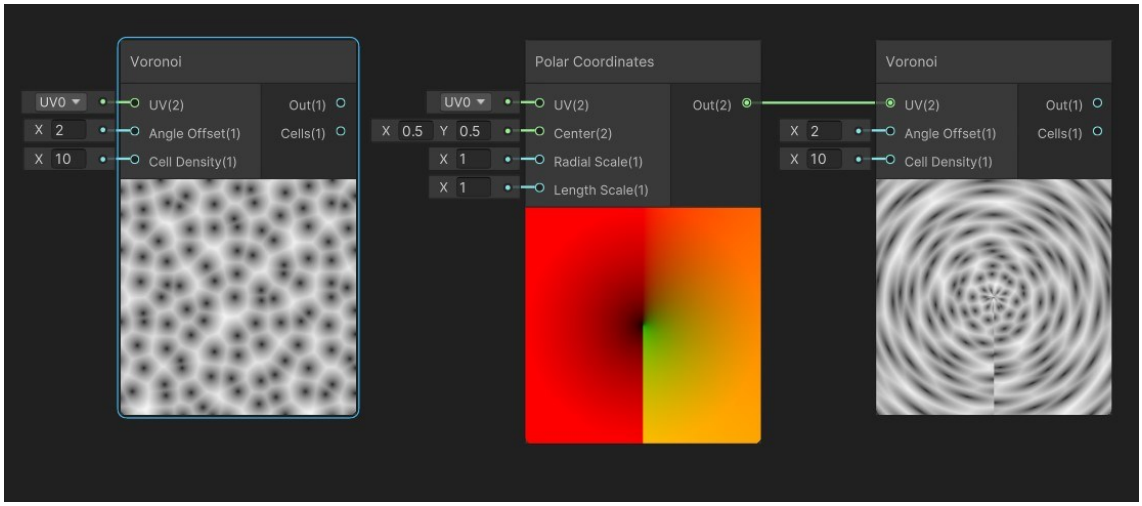
Esimerkkikoodi 8. Toteutuksen spectral flux -esimerkki, jossa käytetään vain spektrin bassotaajuuksia iskuntunnistukseen.

5.3 Kohinavarjostimen toteutus Shader Graphissä

Kohinan avulla saavutettu efekti toteutettiin kokonaisuudessaan Unityn Shader Graph -työkalulla. Varjostin ei tarvitse valaistusta, joten varjostinta luodessa Shader Graphin asetukseksi valittiin ”unlit”. Varjostimen tärkeimpinä kulmakivinä toimivat kaksi Shader Graphin solmua, Polar Coordinates (suom. Napakoordinaatit) ja Voronoi Noise. Voronoi-kohinan sijasta olisi ollut mahdollista

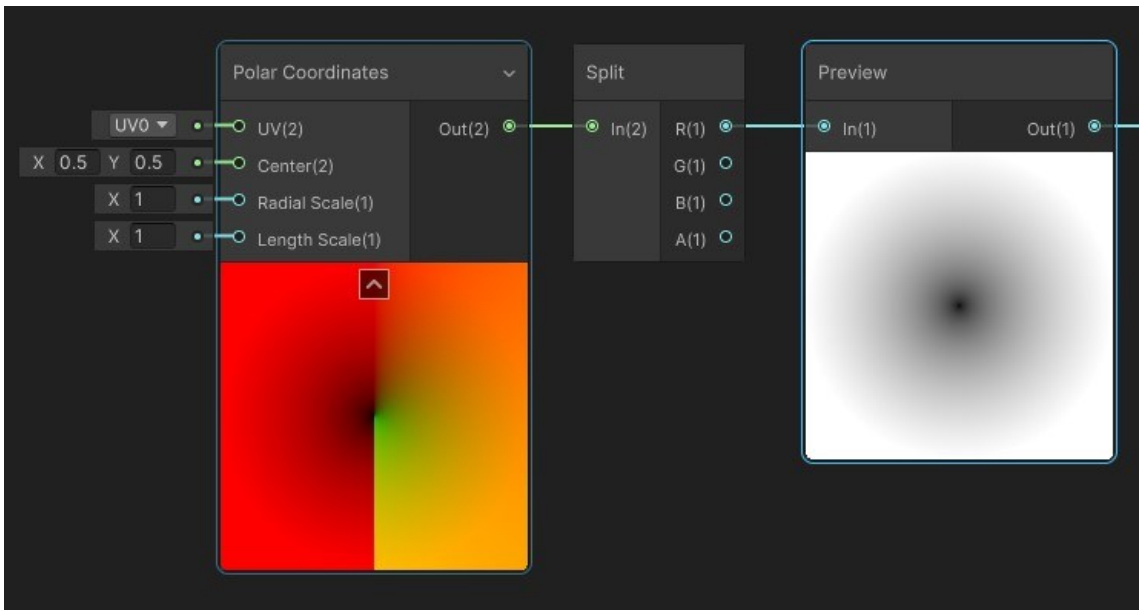
käyttää myös muuta kohinaa saavuttaakseen samankaltaisen efektin. Toinen täysin toimiva vaihtoehto olisi ollut esimerkiksi Simple Noise -solmu, ja päätös käyttää Voronoita johtui lähinnä sen helpommasta muokattavuudesta.

Polar Coordinates -solmu kietoo koordinaatit keskikohdan ympärille siten, että x- ja y-koordinaatit muunnetaan muotoon säde ja kiertokulma (ks. kuva 15). Säde on etäisyys keskipisteestä, kun taas kulma on kiertokulma vastapäivään positiivisesta x-akselista lähtien. Napakoordinaattien avulla saadaan aikaan helposti monia erilaisia pyöreitä kuvioita ja efektejä.



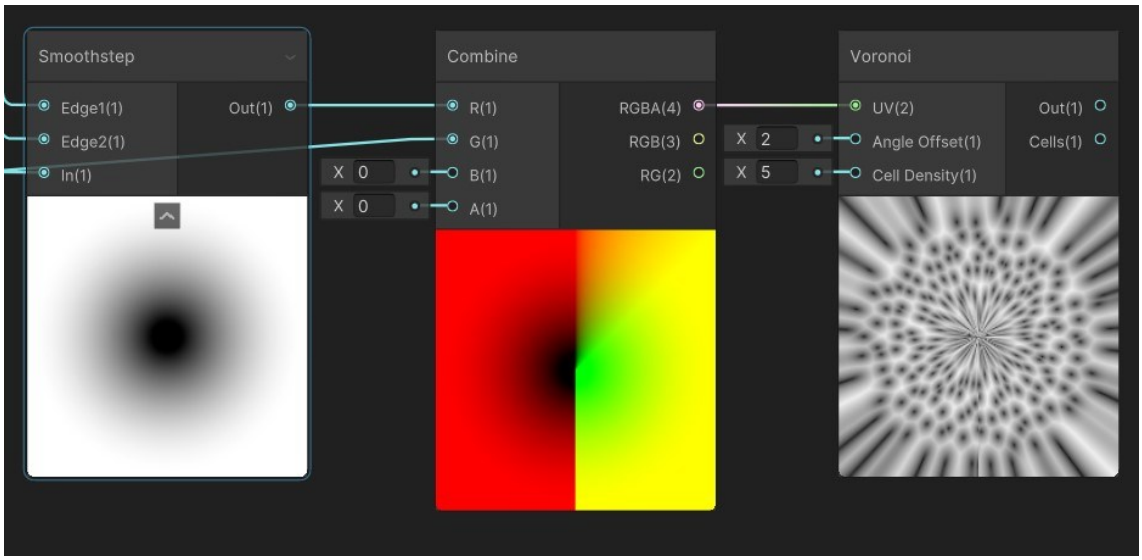
Kuva 15. Voronoi-kohinaa (vas.) ja napakoordinaattien avulla kiedottua Voronoi-kohinaa samoilla asetuksilla (oik.)

Tämän jälkeen liikettä saa aikaan vain lisäämällä Time-solmun, joka antaa pääsyn moniin erilaisiin ajan kulua kuvaaviin arvoihin. Projektia varten myös varjostimen keskustaa haluttiin kontrolloida erikseen. Tämän saavuttaminen onnistuu helposti ottamalla napakoordinaateista X-kanava (Split-solmussa R). Saatu tulos on gradientti etäisyyden mukaan navasta (ks. kuva 16).



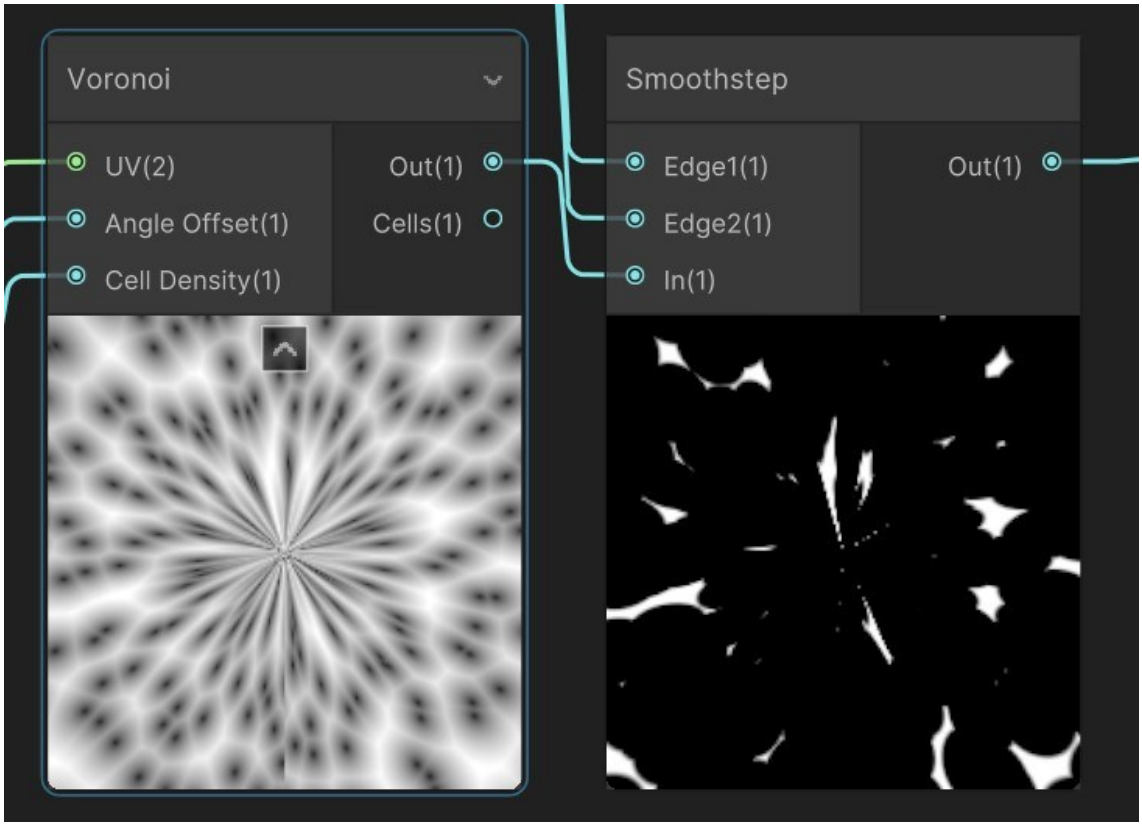
Kuva 16. Napakoordinaattien x-kanava antaa pääsyn gradienttiin navasta ulospäin.

Kuvassa 16 näkyvän mustan ja valkoisen alueen koon kontrolloimiseksi sekä gradientin jyrkkyyden kontrolloimiseksi lisättiin muutamia solmuja ja muuttujia. Tähän erittäin käteväksi osoittautui Smoothstep-solmu, jolla pystyy kontrolloimaan gradientin jyrkkyyttä sekä keskustan kokoa (ks. kuva 17).



Kuva 17. Gradientin kontrollointia Smoothstep-solmun avulla ja sen vaikutus Voronoi-kohinaan.

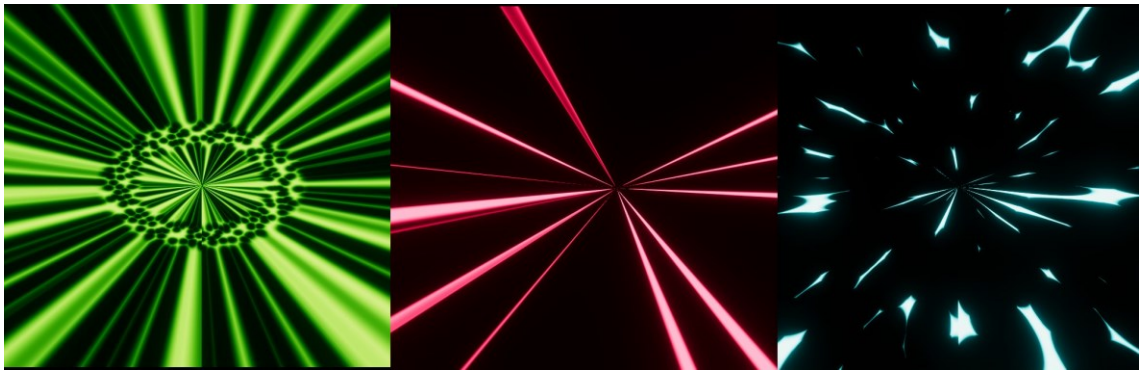
Saatuun tulokseen lisättiin vielä yksi Smoothstep-solmu, jonka avulla mustien ja valkoisten alueiden väliset gradientit pystyttiin joko pitämään asteittaisina, tai sulauttamaan alueet joko mustiksi tai valkoisiksi. Kuvassa 18 näkyy esimerkki, jossa Voronoi-kohina on muutettu tarkkareunaisiksi viiruiksi Smoothstep-solmun muuttujia käyttäen.



Kuva 18. Voronoi-kohinan alueiden sulauttamista yhteen viiruiksi Smoothstep-solmun avulla.

Itse varjostimen käyttö pelitilassa oli melko suoraviivaista. Ensin Unityssä luodaan niin sanottu scene, jonka sisälle peliobjektit luodaan. Unity luo automaattisesti myös kameran, jonka näkökulmasta pelimaailmaa katsotaan. Varjostimelle on myös luotava materiaali, jolle varjostin asetetaan. Tämän jälkeen materiaali voidaan asettaa halutulle peliobjektille. Työtä varten kätevin tapa oli luoda sceneen taso (engl. plane), joka asetettiin kameraan nähden niin, että se peitti koko näkymän. Näin luotu varjostin saatiin peittämään koko ruutu. Varjostimen värittömät osat olivat läpikuultavia, joten paremman näkyvyyden varmistamiseksi tausta muutettiin yksiväriseksi Skybox-asetuksista.

Lopputuloksena oli erittäin moniin erilaisiin efekteihin kykenevä varjostin, jolla pystyi helposti luomaan näyttäviä visualisaatioita (ks. kuva 19).



Kuva 19. Lopullisella varjostimella aikaansaatuja efektejä.

5.4 Verteksivarjostimen toteutus Shader Graphissä

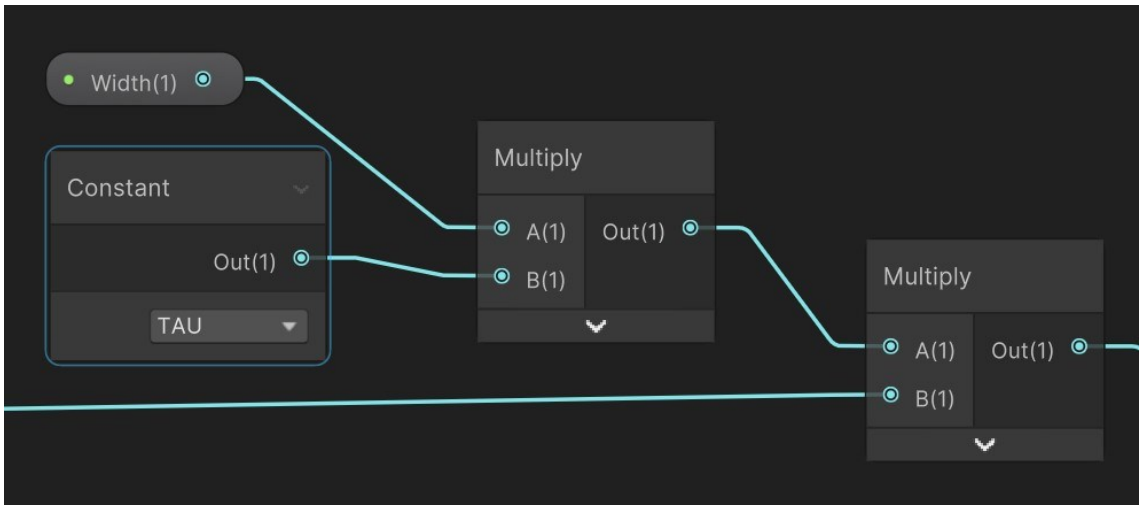
Verteksivarjostinta lähdettiin toteuttamaan erityisesti iskuntunnistusta silmällä pitäen. Näin ollen tarkoituksena oli luoda varjostin, jossa pystyy laukaisemaan yhden selkeän efektin koodin havaitessa iskun musiikista. Alkupisteenä käytettiin erilaisten varjostimien yhdistelmää, johon lähdettiin lisäämään tarvittavia toimintoja.

Verteksivarjostimen vahvuutena on mahdollisuus vaikuttaa verteksien koordinaatteihin, jonka avulla myös kappaleen muotoa voidaan muuttaa. Työssä tätä hyödynnettiin luomaan aaltoliike pinnalle. Aaltoliike eteni lähtöpisteestä rengasmaisesti ulospäin. Verteksin paikkaan tilassa päästään käsiksi Shader Graphin Position-solmulla. Samaa solmua käytettiin myös aaltoliikkeen alkupisteen määrittämiseen. Halutun keskipisteen ja verteksin välistä välimatkaa tarvittiin aaltoliikkeen luomiseksi, joten se otettiin vähentämällä Position-solmu halutusta keskipisteestä ja syöttämällä tulos Length-solmuun.

Varjostin oli luonteeltaan huomattavasti Voronoi-kohinan avulla toteutettua fragmenttivarjostinta monimutkaisempi. Varjostimen osiot voidaan jakaa karkeasti aaltojen paksuuteen (ks. kuva 20), aaltojen syntymistiheyteen, aaltojen

amplitudiin, aaltojen hiipumiseen ulospäin mentäessä, aaltojen hiipumiseen ajan kuluessa ja värin kontrollointiin.

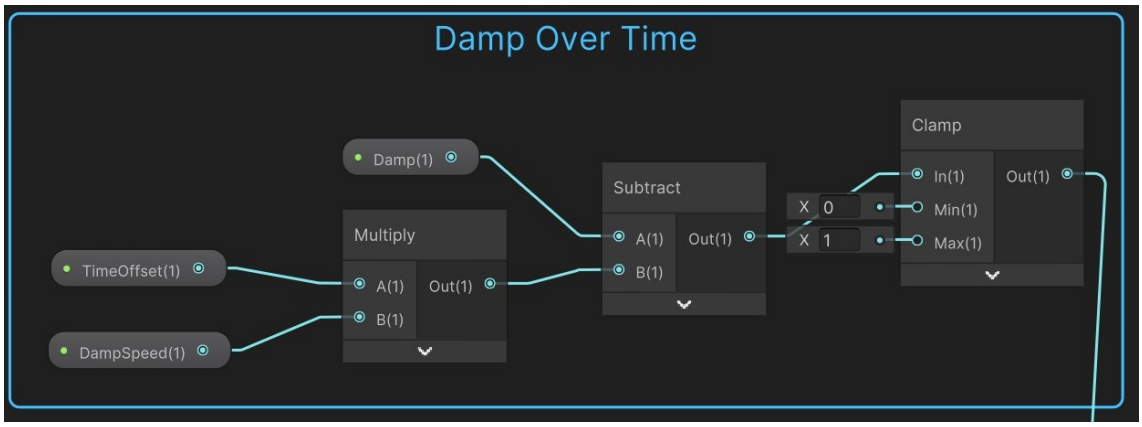
Olennainen osa varjostinta oli Normal Vector-solmu, jonka avulla saatiin tason normaalivektori. Tähän arvoon vaikuttamalla verteksejä päästiin liikuttelemaan ylös ja alas, joka sai aikaan halutun aaltomaisen efektin. Se, kuinka paljon kutakin verteksiä piti liikuttaa, riippui juuri aiemmin mainitusta etäisyydestä aallon keskustasta, joten lähes kaikissa osioissa tarvittiin aluksi otettua etäisyyttä.



Kuva 20. Aallon paksuuden määrittävä osio. Viimeiseen Multiply-solmuun liittyy etäisyys aallon keskipisteestä, eli Length-solmu.

Liikkeen aaltomaisuus saatiin aikaan Sine-solmulla, joka palauttaa annetun arvon sinin. Kun mukaan lisättiin aika, saatiin jo melko hyviä aaltoja aikaan. Tarkoituksena oli kuitenkin myös kontrolloida sitä, kuinka nopeasti koko aalto hiipuu tason pintaan, joten amplitudin kontrolloimiseksi ajan kuluessa luotiin oma osionsa.

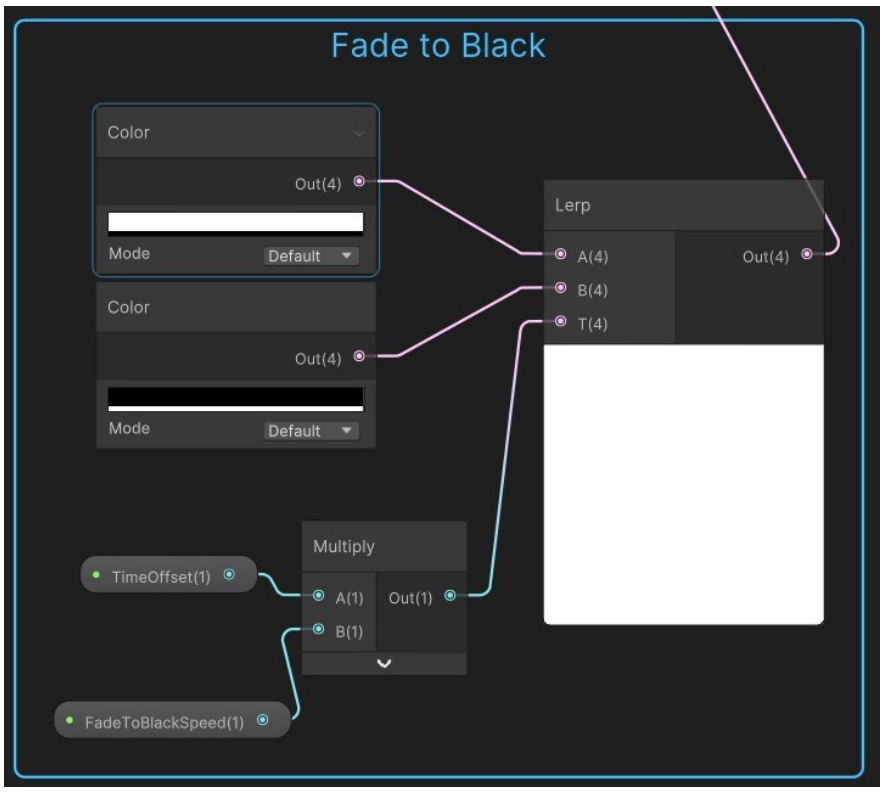
Kuten kuvassa 21 näkyy, Time-solmun käyttämisen sijasta aikaa kontrolloidaan omalla muuttujalla, joka on nimetty TimeOffset. Oman muuttujan luominen antaa mahdollisuuden ohjata ajan etenemisnopeutta koodilla, kuten myös tarvittaessa nollata varjostin alkuperäiseen tilaansa ennen efektin laukaisua. Tämä oli työn kannalta välttämätöntä, sillä sama efekti laukaistiin monia kertoja.



Kuva 21. Aallon hiipumisen nopeuden määrittävä osio.

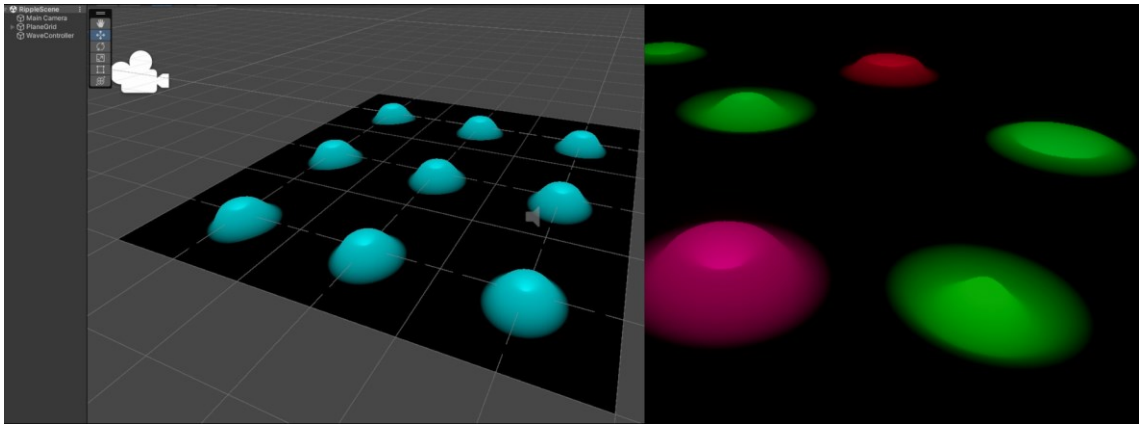
Koko aallon hiipumisen lisäksi luotiin myös mekanismi sille, kuinka aalto hiipuu edetessään keskustasta poispäin. Tähän sopi jälleen loistavasti Smoothstep-solmu, jolle syötettiin aluksi otettu etäisyys keskustasta sekä itse luodut muuttujat, joilla ohjattiin aaltoliikkeen laajuutta tasolla sekä sen hiipumista aaltoliikkeen reunoilla.

Aallon keskusta värjättiin sen erottamiseksi ja visuaalisen ilmeen vuoksi. Värjäämiseen käytettiin taas etäisyyttä keskustasta. Kehitysvaiheessa kokeiltiin myös versiota, jossa väri lisättiin sen perusteella, kuinka korkealla verteksi oli tasosta. Monimutkaisempi tapa ei kuitenkaan tuonut parannusta yleisilmeeseen. Työn kannalta myös värin piti hälvetä aallon hiipuessa, joten varjostimeen lisättiin Lerp-solmu, jonka avulla väriä pystyttiin muuttamaan hiljalleen kohti mustaa (ks. kuva 22).



Kuva 22. Osio varjostimesta, jossa Lerp-solmun avulla muutetaan aallon väri hiljalleen lähemmäksi mustaa.

Unityn Scene-näkymässä luotiin ruudukko tasoista, joista jokaiseen oli asetettu materiaali, jossa oli Shader Graphilla luotu varjostin. Kun koodi havaitsi iskun, aktivoitiin joku ruudukon tasoista (ks. kuva 23).



Kuva 23. Scene-näkymän tasoruudukko (vas.) ja musiikin iskuista aktivoituvia aaltoja (oik.)

Varjostin sopi parhaiten musiikkikappaleisiin, joiden iskut ovat hyvin selkeitä, kuten esimerkiksi pianomusiikkiin.

5.5 Varjostimien parametrien kontrollointiin käytetyt menetelmät

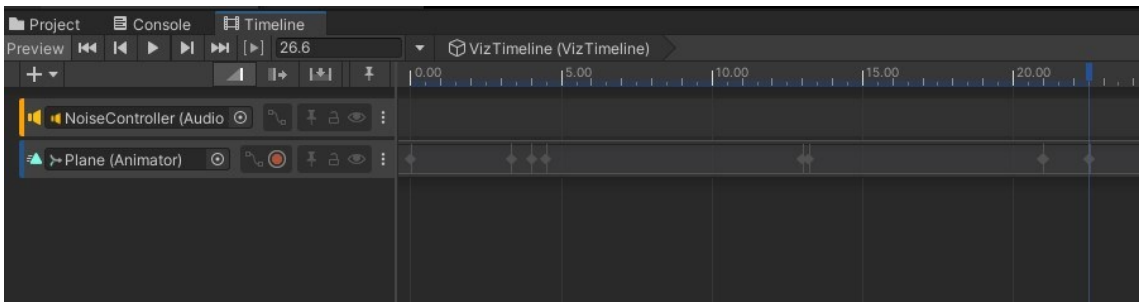
Iskuntunnistukseen perustuvaa varjostinta ohjattiin reaaliaikaisesti musiikin iskut tunnistavalla koodilla. Koodilla pystyttiin myös kontrolloimaan synnytettyjen aaltojen väriä, korkeutta ja niiden laantumisnopeutta. Nämä muuttujat pystyttiin sitomaan musiikista saatuun dataan, kuten eri taajuusalueiden amplitudiin. Myös varjostimen ajankulua ohjaavaa muuttujaa kontrolloitiin suoraan koodista, jolloin efekti saatiin helposti nollattua.

```
void LaunchWave()
    {
        //Random loc
        float x = Random.Range(-randomRadArea, randomRadArea);
        float y = Random.Range(-randomRadArea, randomRadArea);

        _mat.SetFloat("_EffectRadius", bassAvg);
        _mat.SetFloat("_GradientEdge", midRangeLoAvg);
        _mat.SetColor("_Color", Random.ColorHSV(0f, 1f, 1f, 1f, 1f,
1f));
        _mat.SetVector("_WaveOrigin", new Vector3(x, y, 0));
        deltaTime = 0;
    }
```

Esimerkkikoodi 9. Metodi, joka laukaisee aallon, kun isku tunnistetaan.

Kohinaa hyödyntävä verteksivarjostin ei sopinut aivan yhtä hyvin musiikista saatujen arvojen käyttämiseen. Käsittelemättömien arvojen syöttäminen suoraan varjostimeen tuotti helposti visuaalisen sekamelskan, josta oli vaikea löytää intuitiivista yhteyttä taustalla soivaan musiikkiin. Näin ollen kohinavarjostimella toteutettiin lyhyt demo, jossa arvoja ohjattiin käyttäen Unityn Timeline-työkalua (ks. kuva 24). Näin visualisaatiota pystyttiin kontrolloimaan tarkasti, pienimpiä yksityiskohtia myöten.



Kuva 24. Timeline-työkalu, jonka avulla varjostimen arvoja muutetaan haluttuina ajanhetkinä.

Kokeiluksi varjostimesta etsittiin kuitenkin myös muuttujia, jotka sopivat hyvin ajonaikaista visualisointia varten, ja visualisointi toimi myös käyttäen mikrofonista kaapattua ääntä.

5.6 Loppuanalyysi

Varjostimien toteuttaminen onnistui päämäärin hyvin. Varjostimista kumpikin on pätevä ratkaisu juuri niihin käyttötarkoituksiin, joihin ne luotiin. Samalla on kuitenkin todettava, että varjostimien käytettävyys monissa erilaisissa musiikkigenreissä ja kappaleissa jäi osittain vajavaiseksi. Monia erilaisia musiikkikappaleita saumattomasti visualisoivan varjostimen tekemiseksi olisi pitänyt tehdä muutoksia projektin rakenteeseen. Tärkeää olisi ollut mennä enemmän dynaamisuus edellä, jolloin eri nopeuksiset ja spektriltään täysin erilaiset kappaleet eivät olisi sekoittaneet varjostimen toimintaa. Tämän lisäksi iskuntunnistuksen implementoiminen vaati ennakoitua enemmän aikaa, joten sen kehitys jäi hieman puutteelliseksi, joskin lopputulos oli silti toimiva.

Ohjelmoinnin perustoteutus, eli datan saaminen, taajuuksien erottelu ja sen pohjalta varjostimeen vaikuttaminen sujui ongelmitta. Unityn tarjoamien metodien avulla työskentely oli erittäin kätevää. Etukäteisanalysoinnin toteuttaminen Unity-pelimoottorilla osoittautui kuitenkin hieman luultua vaikeammaksi, jonka takia se päätettiin jättää projektin tavoitteiden ulkopuolelle.

Erityisesti fragmenttivarjostimen toteutus sujui todella näppärästi. Hyvin yksinkertaisella idealla saatiin aikaan erittäin toimiva kokonaisuus, joka on

visuaalisesti näyttävä. Kohinaa hyödyntävä varjostin on myös hyvä esimerkki Shader Graph -työkalun vahvuuksista, sillä vain muutamalla solmulla pystyttiin luomaan melko monimutkainen efekti.

6 Yhteenveto

Musiikin visualisointi on tärkeä työkalu kuuntelukokemuksen parantamisessa. Visualisoinnilla on myös muita vähemmän tiedostettuja puolia, kuten käyttö kuulovammaisten apuna. Myös videopeleissä musiikkia voidaan visualisoida monin eri tavoin mukaansatempaavamman pelikokemuksen aikaansaamikseksi.

Musiikin visualisointiin on tarjolla monia kaupallisia, ja jopa ilmaisia ratkaisuja, joiden kanssa kilpaileminen olisi erittäin vaikeaa. Kuitenkin esimerkiksi omiin peliprojekteihin jonkunlaisen visualisoinnin sisällyttäminen voisi vaatia tämän insinöörityön kaltaisia itse kehitettyjä ratkaisuja.

Insinöörityön pohjalta voidaan todeta, että Unityn Shader Graph tarjoaa loistavat mahdollisuudet musiikin visualisointiin ja vaivattomaan varjostimien luomiseen. Sen vahvuutena on eritoten visuaalisuus, jonka ansiosta myös ne, joilla ei ole kokemusta varjostimien ohjelmoinnista voivat työskennellä niiden kanssa matalalla kynnyksellä.

Insinöörityössä luotiin kaksi toimivaa varjostinta, joiden avulla pystyttiin visualisoimaan musiikkia. Työn tarkoituksena oli tutkia erilaisia tapoja analysoida ja visualisoida musiikkia. Huomattiin, että Shader Graph antaa valtavasti vaihtoehtoja varjostimien visuaalisiin toteutustapoihin. Voitiin myös todeta, että Unityyn sisältyy monia musiikin analysoimista edesauttavia metodeja. Pelimoottori osoittautui oivaksi alustaksi musiikin visualisoinnille, sillä varjostimiin oli todella helppoa vaikuttaa koodin, tai muiden Unityn tarjoamien työkalujen avulla.

Lähteet

1 Knecht, Stacey. 2013. Arcimboldo and his Many Shades of Gray. Verkkoaineisto. Slavische Studies. <https://slavischestudies.wordpress.com/2013/09/22/arcimboldo-and-his-many-shades-of-gray/>. 22.9.2013. Luettu 6.4.2024.

2 Harmon, Paul. Brief History of Visual Music. Verkkoaineisto. Over Processed Thinking. <https://overprocessedthinking.com/brief-history-of-visual-music/>. Luettu 6.4.2024.

3 Taylor, Matthew. RetroTech: Atari Video Music - The Migraine Machine. Verkkoaineisto. Youtube. <https://www.youtube.com/watch?v=wle0eqBwtL8>. Haettu 6.4.2024.

4 Art-of-Coding, an initiative to enlist the demoscene as first digital culture on the list of UNESCO intangible world cultural heritage. Verkkoaineisto. Demoscene – The Art of Coding. <https://demoscene-the-art-of-coding.net/the-demoscene/>. Luettu 7.4.2024.

5 Saarenoja, Panu. 2020. Suomalaisenkin pelialan nousuun johtanut demoskene kapusi kansalliselle kulttuuriperintöjen listalle. Verkkoaineisto. Pelaaja. <https://www.pelaaja.fi/uutiset/suomalaisenkin-pelialan-nousuun-johtanut-demoskene-kapusi-kansalliselle-kulttuuriperintojen/>. 15.4.2020. Luettu 7.4.2024.

6 Acme: 303 | 1996 DOS DEMO | 4K | 60FPS. Verkkoaineisto. Youtube. <https://www.youtube.com/watch?v=9Wd3r-spp0Y>. Haettu 8.4.2024.

7 Music-making for the deaf. Verkkoaineisto. Science Daily. <https://www.sciencedaily.com/releases/2015/11/151118101815.htm>. 18.11.2015. Luettu 7.4.2024.

8 de Bastion, Myles. 2015. Cymatic Lighting: A Modern "Visual Sound" System for Deaf & Hard-of-Hearing. Verkkoaineisto. Devpost. < https://connectability.devpost.com/submissions/38464-cymatic-lighting-a-modern-visual-sound-system-for-deaf-hard-of-hearing>. Luettu 7.4.2024.

9 Varrasi, John. 2014. How Visuals Can Help Deaf Children 'Hear'. Verkkoaineisto. Live Science. <https://www.livescience.com/47004-visuals-help-deaf-children-experience-sound.html>. 25.7.2014. Luettu 16.4.2024.

10 Beat Saber. Verkkoaineisto. Steam. <https://store.steampowered.com/app/620980/Beat_Saber/>. Haettu 17.4.2024.

11 Keogh, Jesse. 2018. Adding Songs and Beatmaps to Chop It. Verkkoaineisto. Medium. <https://www.livescience.com/47004-visuals-help-deaf-children-experience-sound.html>. 26.8.2018. Luettu 13.4.2024.

12 Spectrum Valley. Verkkoaineisto. Itch.io. < https://beed28.itch.io/spectrum-valley>. Haettu 17.4.2024.

13 About Shader Graph. Verkkoaineisto. Unity. https://docs.unity3d.com/Packages/com.unity.shadergraph@17.0/manual/index.html>. Luettu 12.4.2024.

14 Introduction to ShaderGraph. Verkkoaineisto. Unity Learn. <https://learn.unity.com/tutorial/introduction-to-shader-graph#>. Päivitetty 9.6.2022 Luettu 7.4.2024.

15 Cooper, Tim. 2018. Introduction to Shader Graph: Build your shaders with a visual editor. Verkkoaineisto. Unity. <https://blog.unity.com/engine-platform/introduction-to-shader-graph-build-shaders-in-visual-editor>. 27.2.2018. Luettu 13.4.2024.

16 Material Editor Reference. Verkkoaineisto. Unreal Engine. <https://docs.unrealengine.com/4.27/en-US/RenderingAndGraphics/Materials/Editor/>. Haettu 15.4.2024.

17 Rodriguez, Jacobo. 2018. GLSL Essentials. Verkkoaineisto. Packt. <https://subscription.packtpub.com/book/programming/9781849698009/1/ch01lvl1sec10/types-of-shaders>. 31.12.2012. Luettu 1.5.2024.

18 Shader: Technical Overview. Verkkoaineisto. VFXDoc. <https://vfxdoc.readthedocs.io/en/latest/shaders/overview/>. Luettu 2.5.2024.

19 Uniform (GLSL). Verkkoaineisto. OpenGL Wiki. <https://www.khronos.org/opengl/wiki/Uniform_(GLSL)>. Päivitetty 18.5.2020. Luettu 2.5.2024.

20 Serrano, Harold. 2015. What is a Vertex Shader in OpenGL?. Verkkoaineisto. Harold Serrano. <https://www.haroldserrano.com/blog/what-is-a-vertex-shader-in-opengl>. 10.7.2015. Luettu 2.5.2024.

21 Chapter 2: The Graphics Pipeline. Verkkoaineisto. <https://graphicscompendium.com/intro/01-graphics-pipeline>. 10.7.2015. Haettu 2.5.2024.

22 Vertex Shaders. Verkkoaineisto. Nvidia. <https://www.nvidia.com/en-us/drivers/feature-vertexshader/>. Luettu 15.4.2024.

23 Shader Graph: Vertex Displacement. Verkkoaineisto. Unity Learn. <https://learn.unity.com/tutorial/shader-graph-vertex-displacement#5f500eb8edbc2a002074063d>. Päivitetty 12.10.2020. Haettu 15.4.2024.

24 Schell, Matt; Eng, Sam. 2019. Creating an Interactive Vertex Effect using Shader Graph. Verkkoaineisto. Unity. <https://blog.unity.com/engine-platform/creating-an-interactive-vertex-effect-using-shader-graph>. 12.2.2019. Luettu 12.4.2024.

25 Sacco, Michael. 2023. Unity: Understanding URP, HDRP, and Built-In Render Pipeline. Verkkoaineisto. Unity. <https://www.occasoftware.com/blog/unity-understanding-urp-hdrp-built-in>. 3.5.2023. Luettu 17.4.2024.

26 Olthof, Peter. 2016. Audio Visualization - Unity/C# Tutorial [Part 1 - FFT/Spectrum Theory]. Verkkoaineisto. Youtube. <https://www.youtube.com/watch?v=4Av788P9stk>. 17.9.2016. Katsottu 18.4.2024.

27 Lovell, Kathryn. 2017. The Fast Fourier Transform: Composite Audio Visualizer with R and Web Audio. Verkkoaineisto. Kathryn Lovell. <https://www.youtube.com/watch?v=4Av788P9stk>. 13.10.2017. Luettu 15.4.2024.

28 What Is a Fast Fourier Transform (FFT)? Verkkoaineisto. MathWorks. < https://www.mathworks.com/discovery/fft.html>. Luettu 12.4.2024

29 AudioSource.GetSpectrumData. Verkkoaineisto. Unity Documentation. <https://docs.unity3d.com/ScriptReference/AudioSource.GetSpectrumData.html>. Luettu 8.4.2024.

30 Olthof, Peter. 2017. Microphone Input Visuals - Unity/C# Tutorial [Part 1 - Get Microphone Devices]. Verkkoaineisto. Youtube. <https://www.youtube.com/watch?v=GHc9RF258VA>. 3.10.2017. Katsottu 12.4.2024.

31 Keogh, Jesse. 2018 Algorithmic Beat Mapping in Unity: Real-time Audio Analysis Using the Unity API. Verkkoaineisto. Medium. <https://medium.com/giant-scam/algorithmic-beat-mapping-in-unity-real-time-audio-analysis-using-the-unity-api-6e9595823ce4>. 27.2.2018. Luettu 13.4.2024.

32 Keogh, Jesse. 2018. Algorithmic Beat Mapping in Unity: Preprocessed Audio Analysis. Verkkoaineisto. Medium. <https://medium.com/giant-scam/algorithmic-beat-mapping-in-unity-preprocessed-audio-analysis-d41c339c135a>. 27.2.2018. Luettu 13.4.2024.

33 Subdivide. Verkkoaineisto. Blender 2.80 Manual. <https://docs.blender.org/manual/en/2.80/modeling/meshes/editing/subdividing/subdivide.html>. Haettu 13.4.2024.

34 The Audio Frequency Spectrum Explained. Verkkoaineisto. Headphonesty. <https://www.headphonesty.com/2020/02/audio-frequency-spectrum-explained/>. Haettu 18.4.2024.