Janne Kauppinen

Wgpu geneerisenä laskentarajapintana

Tietotekniikan pro gradu -tutkielma

4. maaliskuuta 2023

Jyväskylän yliopisto

Informaatioteknologian tiedekunta

Tekijä: Janne Kauppinen

Yhteystiedot: janne.a.kauppinen@student.jyu.fi

Ohjaajat: Sampsa Kiiskinen ja Tuomo Rossi

Työn nimi: Wgpu geneerisenä laskentarajapintana

Title in English: Wgpu as a generic compute API

Työ: Pro gradu -tutkielma

Opintosuunta: Ohjelmistotekniikka

Sivumäärä: 62+6

Tiivistelmä: Tässä tutkimuksessa tutkitaan wgpu-kirjaston käytettävyyttä GPU-laskentaan. Wgpu on Rust-ohjelmointikielellä toteutettu grafiikka- ja laskentarajapinta, joka on suunniteltu toimimaan useassa eri ajoympäristössä. Tutkimusta varten kehitetään wgpu-sovellus, joka luo pistepilvidatasta etäisyyskentän. Etäisyyskentän luonti toteutetaan eikonal-yhtälön ratkaisijalla. Kenttää visualisoidaan sphere tracing -tekniikalla. Ohjelman algoritmit toteutetaan WGSL-varjostinohjelmointikielellä, ja ohjelma testataan Windows-, Linux- ja macOS-käyttöjärjestelmissä. Ohjelma käännetään natiiveiksi ohjelmiksi ja WebAsseblyksi. WebAssembly-versiota testataan eri käyttöjärjestelmissä Firefox Nightly web-selaimessa, jolloin käytettävissä oleva grafiikka- ja laskentarajapinta on web-selaimen WebGPU-rajapinta. Wgpu on teknologiana vielä uusi ja keskeneräinen, mikä käy ilmi tutkimuksen testeissä. Tutkimus osoittaa kuitenkin sen, että teknologialla on potentiaalia.

Avainsanat: wgpu, WebGPU, WGSL, GPU-laskenta, WebAssembly, Rust, JavaScript, eikonal-yhtälö, fast iterative method, sphere tracing, pistepilvi

Abstract: This study investigates the usability of the wgpu library for GPU computing. Wgpu is a graphics and computing interface implemented in Rust, and it is designed to work in several different runtime environments. In this study, a wgpu application is being developed that generates a distance field from point cloud data. The distance field is generated using an eikonal equation solver, and it is visualized using a sphere tracing technique. The algorithms

are implemented in WGSL shading language, and the program is tested on Windows, Linux and macOS operating systems. The program is compiled into native programs and WebAssembly. The WebAssembly version is tested on different operating systems using the Firefox Nightly web browser. In the WebAssembly version, the program uses the WebGPU interface of the web browser. This study confirms that wgpu is still a work-in-progress technology. However, research shows that technology has potential.

Keywords: wgpu, WebGPU, WGSL, GPU computing, WebAssembly, Rust, JavaScript, eikonal equation, fast iterative method, sphere tracing, point cloud

Termiluettelo

GPU	Grafiikkaprosessori (engl. *Graphical procressin unit*).
GPGPU	Tekniikka, jossa grafiikkasuorittimella suoritetaan laskentaa (engl. *General-purpose computing on graphics processing units*).
WebGPU	Web-selainympäristöön kehitteillä oleva grafiikka- ja laskentarajapintaspesifikaatio.
wgpu	Rust-ohjelmointikielellä toteutettu, WebGPU-rajapintaan perustuva, grafiikka- ja laskentarajapinta.
Varjostinohjelma	Näytönohjaimella suoritettava tietokoneohjelma (engl. *shader*).
WGSL	WebGPU-rajapinnan tukema varjostinohjelmointikieli.
WebAssembly	Matalan tason binäärikoodiformaatti, jota voidaan suorittaa esimerkiksi web-selaimessa.
Web IDL	Rajapintamäärittely kieli, joka on kehitetty erityisesti web-ympäristön rajapintojen määrittelyä varten.
Eikonal-yhtälö	Epälineaarinen Hamilton-Jacobi-osittaisdifferentiaaliyhtälö, jolla ratkaistaan itsestään poispäin etenevän aallon saapumisaikoja.
Fast iterative method	Listan prosessointiin perustuva eikonal-yhtälön ratkaisualgoritmi.
Sphere tracing	Säteenheittotekniikka, jolla pyritään visualisoimaan epäsuoria pintoja hyödyntäen geometrisia etäisyyksiä.

Kuviot

Taulukot

Sisältö

1 Johdanto

Wgpu (Mages 2022a) on Rust-ohjelmointikielellä toteutettu grafiikka- ja laskentarajapinta, joka perustuu WebGPU-rajapintaspesifikaatioon (W3C 2022a). Wgpu on suunniteltu toimimaan eri käyttöjärjestelmissä siten, että se käyttää käyttöjärjestelmien tukemia natiiveja grafiikka- ja laskentarajapintoja, kuten esimerkiksi Vulkan-, D3D12- ja metal-rajapintoja. Vulkan, D3D12 ja metal mahdollistavat grafiikkasuorittimella (engl. *graphics processing unit*, lyhennettynä GPU) toteutettavan geneerisen laskennan (engl. *general-purpose computing on graphics processing units, lyhennettynä GPGPU*). Tästä syystä myös wgpu-rajapintaa voidaan käyttää GPU-laskennan toteuttamiseen sellaisissa käyttöjärjestelmissä, jotka tarjoavat Vulkan-, D3D12- tai metal-rajapintatuen.

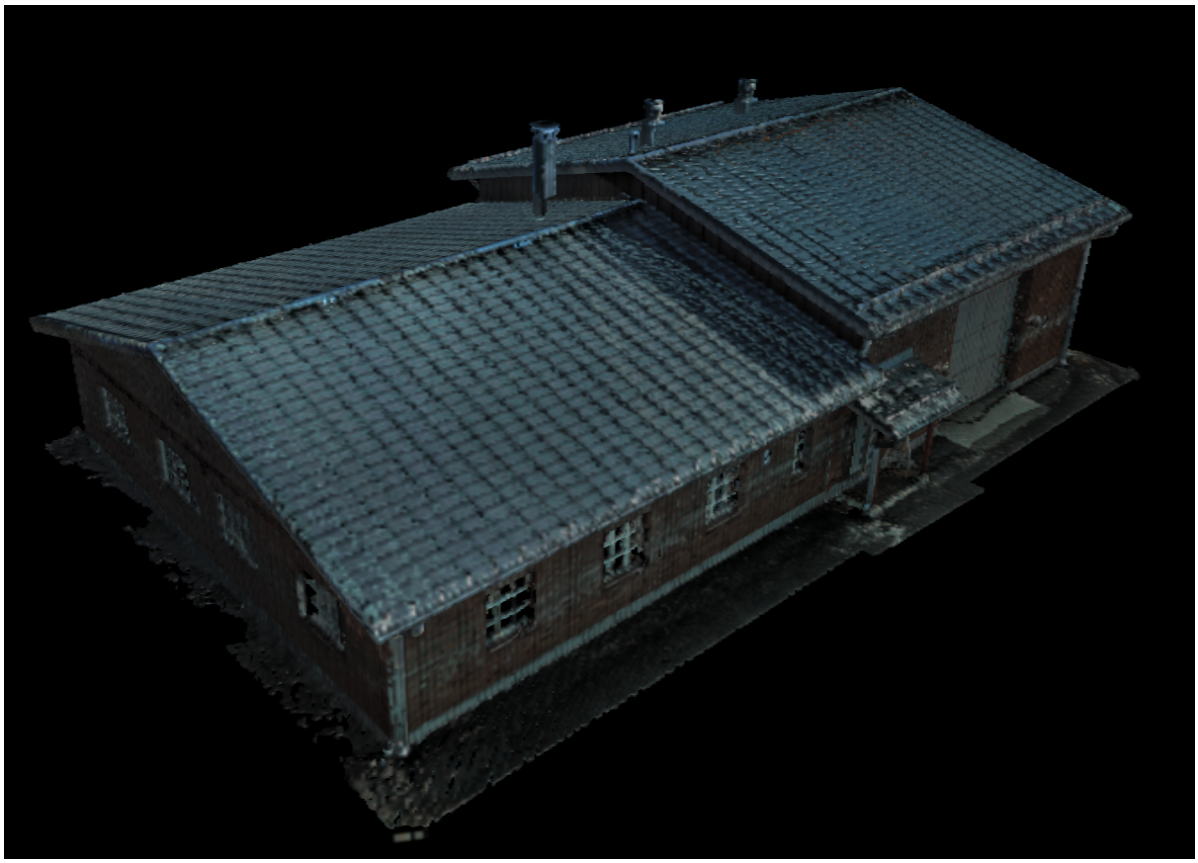
Wgpu-sovellukset on mahdollista kääntää WebAssemblyksi (lyhennettynä Wasm), mikä mahdollistaa sovellusten suorittamisen web-selaimissa. Web-selaimessa toimiva wgpu-sovellus käyttää selaimen omaa WebGPU-rajapintaa, joka on JavaScript-rajapinta WebGPU-rajapintaspesifikaatiosta. Selaimessa toimiva WebGPU-rajapinta käyttää wgpu:n tavoin käyttöjärjestelmän Vulkan-, D3D12- ja metal-rajapintoja, mikä mahdollistaa GPU-laskennan myös web-selaimissa.

Tässä tutkielmassa tutkitaan wgpu:n soveltuvuutta geneeriseen GPU-laskentaan käyttöjärjestelmäriippumattomuuden näkökulmasta. Tutkimusta varten kehitetään Rust-ohjelmointikielellä wgpu-sovellus, joka muodostaa syötteenä annetusta pistepilvidatasta etäisyyskentän (engl. *distance field*). Etäisyyskentän luonti toteutetaan eikonal-yhtälön avulla, ja yhtälön ratkaisualgoritmin pohjana käytetään Yuhao Huangin esittämää nopeaiterointimentelmää (Huang 2021) (engl. *fast iterative method*, lyhennettynä FIM). Etäisyyskenttää visualisoidaan sphere tracing -tekniikalla (Hart 1996).

Tutkimuksessa käytettävät algoritmit ohjelmoidaan wgpu:n tukemalla WebGPU Shading Language (lyhennettynä WGSL) (W3C 2022b) -varjostinohjelmointikielellä. Tutkimuksen varjostinohjelmat suoritetaan näytönohjaimella, ja ne muodostavat tutkimuksessa käytettävän geneerisen GPU-laskennan. Ohjelma testataan Linux-, Windows- ja macOS-käyttöjärjestelmissä natiiviksi käännettyinä ohjelmina ja web-selaimessa WebAssemblyksi kään-

nettynä ohjelmana. Tarkoituksena on tutkia, kuinka hyvin wgpu-rajapinta soveltuu GPU-laskentaan näissä ajoympäristöissä.

Tutkimusta tehtäessä wgpu-rajapinta on vielä kovan kehityksen alla. Sekä wgpu, WebGPU että WGSL ovat uusia teknologioita. Tästä syystä nämä teknologiat ovat kiinnostavia tutkimuskohteita. Web-selainten GPU-laskentatuki on ollut tähän asti rajoittunutta. Selainympäristön WebGPU-rajapinta mahdollistaa geneerisen GPU-laskennan web-selaimissa, mikä on yleisesti katsottuna hyödyllistä. Koska wgpu-sovelluksen WebAssembly-versio käyttää web-selaimen WebGPU-rajapintaa, tutkimus antaa tietoa myös selaimen WebGPU:n soveltuvuudesta GPU-laskentaan. Tutkimuksessa käytetään Gisgro Oy:lta (Gisgro Oy 2022) saatua laserskannattua pistedataa (Gisgro Oy, n.d.).



Kuvio 1: Kuvakaappaus ohjelman tuottamasta kuvasta.

2 Grafiikka- ja laskentarajapinnat

Tutkielmassa tutkitaan wgpu-rajapinnan soveltuvuutta geneeriseen GPU-laskentaan. Wgpu on grafiikka- ja laskentarajapinta, joka mahdollistaa GPU-laskennan. Tämän lisäksi wgpu on suunniteltu toimimaan useassa eri käyttöjärjestelmässä ja web-selaimessa. Wgpu poikkeaa monesta muusta rajapinnasta ajoympäristöriippumattomuuden suhteen. Monet grafiikka- ja laskentarajapinnat ovat ajoympäristöriippuvaisia. Ajoympäristöriippumattoman sovelluksen toteuttaminen tällaisilla rajapinnoilla saattaa vaatia useamman eri rajapinnan käytön, mikä vaikeuttaa ohjelmien kehittämistä ja ylläpitoa. Tässä luvussa käydään läpi grafiikka- ja laskentarajapintoihin liittyvää teoriaa, ja lopuksi selitetään miten ja millä rajoitteilla wgpu pystyy tarjoamaan ajoympäristöriippumatonta GPU-laskentaa.

2.1 Rajapinnat yleisesti

Grafiikka- ja laskentarajapinnat mahdollistavat GPU-laitteiden käytön ohjelmoinnissa. Rajapintoja on erilaisia, ja useimmiten ne painottuvat enemmän joko grafiikan renderöintiin tai laskentaan. Uusimmat grafiikkarajapinnat tarjoavat renderöinnin lisäksi myös rajapinnan GPU-laskentaan laskentavarjostinohjelmien (engl. *compute shader*) avulla. Tosin grafiikkarajapintojen tarjoamat laskentamahdollisuudet ovat usein rajoitetumpia verrattuna laskentaan erikoistuneisiin rajapintoihin. Grafiikka- ja laskentarajapinnan integrointi samaan rajapintaan on kuitenkin perusteltua, koska silloin sama rajapinta tarjoaa ohjelmointimahdollisuudet sekä grafiikkaan että laskentaan, jolloin erillistä laskentarajapintaa ei välttämättä tarvita.

Rajapintavalintaa ohjaavat useat eri asiat kuten esimerkiksi se, miten GPU:ta halutaan hyödyntää, ja mitä ominaisuuksia rajapinta ja sitä käyttävät laitteet tukevat. Rajapinnoissa on myös eroja sen suhteen, missä arkkitehtuurissa ohjelmaa halutaan kehittää tai suorittaa. Kaikki näytönohjaimet tai käyttöjärjestelmät eivät tue kaikkia grafiikka- ja laskentarajapintoja. Eroja on myös siinä, miten eri rajapintoja voidaan integroida samaan ohjelmaan. Esimerkiksi rinnakkaislaskentaan varta vasten kehitety OpenCL (Group 2022a) ja OpenGL-grafiikkarajapintaa (Group 2022b) on mahdollista sisällyttää samaan sovellukseen siten, et-

tä ne käyttävät yhteisiä GPU:n resursseja. Rajapintojen yhdistely ei ole kuitenkaan aina mahdollista. Grafiikka- ja laskentarajapinnoista on usein myös eri versioita. Esimerkiksi OpenGL-rajapinnasta on tätä tutkielmaa tehtäessä 20 eri versiota (Group 2022c). Näytönohjaimissa on eroja sen suhteen, mitä rajapintoja ja rajapintojen versioita ne tukevat. Tämä rajoittaa ohjelmassa käytettävän rajapinnan valintaa.

2.2 Natiivit rajapinnat

Grafiikka- ja laskentarajapinnat voivat olla suunniteltu siten, että ne ovat luonteeltaan natiiveja rajapintoja. Natiivit rajapinnat tarjoavat matalan tason ohjelmointirajapinnan, jonka avulla ohjelmoija pääsee käyttämään GPU-laitetta. Tämän lisäksi natiivit rajapinnat tarjoavat rajapinnan GPU-laitteille, jonka avulla laitteiden valmistajat voivat ohjelmoida laiteajureita. Ajureiden avulla GPU-laite voidaan yhdistää natiiviin rajapintaan. Tämä mahdollistaa GPU:n ja rajapinnan välisen kommunikoinnin. Natiivit rajapinnat ovat käyttöjärjestelmäriippuvaisia, joten niitä voidaan käyttää ainoastaan silloin, jos käyttöjärjestelmä tukee käytettävää rajapintaa. Tämän tutkielman kannalta erityisen tärkeitä natiiveja grafiikka- ja laskentarajapintoja ovat Vulkan (Group 2022d), Direct3D 12 (lyhennettynä D3D12) (Microsoft 2022a) ja Metal (Inc 2022).

Vulkan, D3D12 ja Metal ovat uuden sukupolven natiiveja grafiikka- ja laskentarajapintoja. Rajapinnat ovat erikoistuneet ensisijaisesti grafiikan renderöintiin, mutta niihin on rakennettu GPU-laskentaan tarvittavat ominaisuudet. Nämä rajapinnat eroavat toisistaan muun muassa toimintaperiaatteiden, ominaisuuksien ja käyttöjärjestelmätukien perusteella. Rajapinnat eroavat myös sen suhteen, mitä GPU:lla suoritettavia varjostinohjelmointikieliä (engl. *shading language*) ne tukevat.

Vulkan on Chronos Groupin (Group 2022e) kehittämä matalan tason grafiikka- ja laskentarajapinta-standardi. Vulkan käyttää varjostinohjelmointikielenään SPIR-V-välikieltä (Group 2022f). Natiivi Vulkan-tuki löytyy Windows- ja Linux-käyttöjärjestelmistä.

D3D12 on Microsoftin kehittämä ja ylläpitämä rajapinta, joka toimii Windows-käyttöjärjestelmissä. Se käyttää varjostinohjelmointikielenään High-Level Shader Language (lyhennettynä HLSL) -kieltä. Metal on vastaavasti Applen kehittämä macOS- ja iOS-laitteille kehi-

tetty rajapinta. Metal käyttää varjostinohjelmointikielenään Metal Shading Language -kieltä (lyhennettynä MSL) .

2.3 Implisiittiset rajapinnat

On olemassa myös rajapintoja, jotka ovat luonteeltaan abstraktioita jo olemassa olevista rajapinnoista. Tällaiset rajapinnat eivät toimi suoraan GPU-laitteiden kanssa, vaan ne toimivat välityskerroksena eri grafiikka- ja laskentarajapintojen välillä. ANGLE (lyhenne sanoista *Almost Native Graphics Layer Engine*) (Google 2022a), MoltenVK (Group 2022g) ja wgpu ovat tämänkaltaisia rajapintoja. Tällaiset rajapinnat mahdollistavat natiivien rajapintojen käytön sellaisissa ympäristöissä, missä niitä olisi vaikea tai jopa mahdotonta kutsua suoraan. Esimerkiksi ANGLE kykenee muuttamaan sulautettujen järjestelmiä varten kehitettyjen OpenGL ES (*OpenGL for Embedded Systems*) 2 ja 3 (Group 2022h) -rajapintojen kutsuja toisille rajapinnoille. Tämän ominaisuuden vuoksi ANGLE-rajapintaa käytetään esimerkiksi Windows-ympäristössä siten, että web-selaimen WebGL-rajapintakutsut (Group 2022i) muutetaan Windowsin natiivien rajapintojen, kuten esimerkiksi OpenGL (Group 2022b) tai Direct3D 11 (lyhennettynä D3D11) (Microsoft 2022b) mukaisiksi kutsuiksi. MoltenVK puolestaan mahdollistaa Vulkan-rajapinnan toiminnan Metal-rajapinnan päällä. Futhark (DIKU 2022) on puhdas funktio-ohjelmointikieli, joka generoi Furhark-kielellä määritetyt ohjelmat muun muassa OpenCL- tai Cuda-ohjelmiksi.

Implisiittiset grafiikka- ja laskentarajapinnat voivat tarjota oman ohjelmointirajapinnan, ja piilottaa ohjelmoijalta varsinaiset natiivit ohjelmointirajapinnat. Tämä helpottaa ohjelman toteuttamista tilanteissa, jossa ohjelman on tarkoitus toimia useissa eri ajoympäristöissä. Tällaisissa tilanteissa ohjelmoija voi toteuttaa ohjelman käyttäen yhtä rajapintaa, eikä ohjelmoijan tarvitse kehittää ja ylläpitää useammalla natiivilla rajapinnalla tehtyjä ohjelmia. Tällaisia grafiikka- ja laskentarajapintoja ovat esimerkiksi WebGPU ja wgpu.

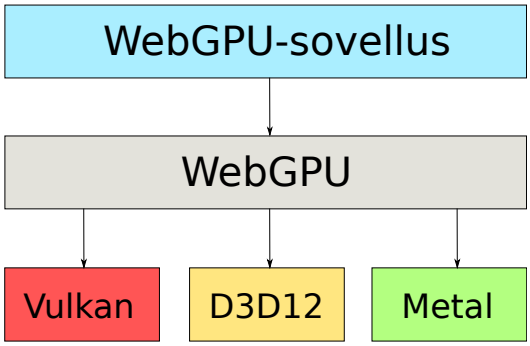
2.4 WebGPU

WebGPU (W3C 2022a) on GPU for the Web Communityn (W3C 2022c) kehittämä, World Wide Web Consortiumin (lyhyemmin W3C) (W3C 2022d) lisenssin alaisuudessa julkaistu,

grafiikka- ja laskentarajapintaspesifikaatio. Sen tarkoitus on tarjota web-selainympäristöön ohjelmointirajapinta, joka hyödyntää käyttöjärjestelmässä olevia natiiveja grafiikka- ja laskentarajapintoja, erityisesti Vulkan-, D3D12- ja Metal-rajapintoja. Tätä tutkielmaa tehdessä WebGPU:sta on kaksi toteutusta. Ensimmäinen on tämän tutkielman kannalta kiinnostava wgpu-rajapinta, joka toteuttaa WebGPU:n toiminnallisuuden Mozillan Gecko-selainohjelmistolle. Dawn (Google 2022b) on puolestaan C++-toteutus Googlen Chromium-verkkoselainprojektille. Varsinainen selainten tarjoama WebGPU-ohjelmointirajapintakieli on JavaScript.

WebGPU toimii välityskerroksena Vulkan-, D3D12- ja Metal-rajapinnoille. Päätös siitä, mitä natiivia grafiikkarajapintaa käytetään, tapahtuu ohjelman suorituksen aikana. Yhteensopivuuden vuoksi WebGPU on suunniteltu siten, että se tarjoaa yhteisiä GPU-ohjelmointimahdollisuuksia, jotka ovat toteutettavissa Vulkan-, D3D12- ja Metal-rajapinnoilla. Tämä karsii pois tietyt rajapintakohtaiset piirteet, mutta samalla se laajentaa ohjelman ajoympäristömahdollisuuksia.

Koska WebGPU käyttää ajonaikana Vulkan-, D3D12- tai metal-rajapintaa, sen avulla voidaan suorittaa GPU-laskentaa web-selaimessa. Web-selaimissa tapahtuva GPU-laskenta on aikaisemmin ollut rajoitettua. Web-selaimia varten kehitetty WebGL-rajapinta on tuetuin web-ympäristössä oleva grafiikkarajapinta. Vaikka WebGL on laajasti tuettu web-selaimissa, se soveltuu melko heikosti esimerkiksi GPU-laskentaan, koska siitä puuttuu tuki laskentavarjostinohjelmille. WebGL:ään on kehitetty laajennus, WebGL 2.0 Compute (Group 2022j), joka mahdollistaa laskentavarjostinohjelmien käytön. WebCL (Group 2022k) on JavaScript-ohjelmointirajapinta, joka mahdollistaa OpenCL-laskentarajapinnan käytön web-selaimissa. Sekä WebGL:n laskentavarjostinohjelmalaajennos että WebCL ovat jääneet taka-alalle tällä hetkellä kehitteillä olevan WebGPU-rajapinnan vuoksi.



Kuvio 2: WebGPU-rajapintaa käyttävän web-sovelluksen lopullinen rajapinta määräytyy sovelluksen ajoympäristön mukaan. Kuvassa WebGPU-sovellus käyttää WebGPU-rajapintaa, joka on ohjelmoijalle näkyvä ohjelmointirajapinta. Vulkan, D3D12 ja Metal ovat käyttöjärjestelmien natiiveja rajapintoja. Nuolet ilmaisevat käskyjen kulkusuunnan.

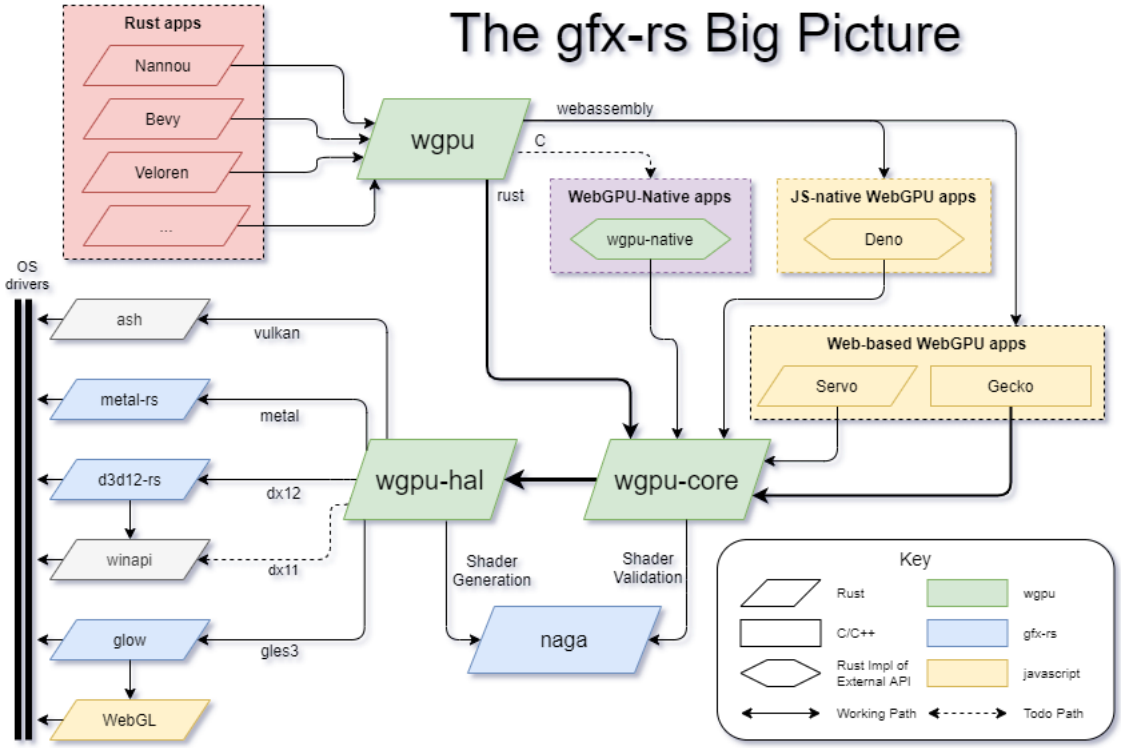
Tätä tutkielmaa tehtäessä WebGPU on vielä luonnosvaiheessa oleva rajapinta. WebGPU on jo osittain tuettu tunnetuimmissa web-selaimissa, mutta sen käyttö on kuitenkin tällä hetkellä rajoitettua (W3C 2022e). Esimerkiksi Firefox-selaimista ainoastaan Firefox Nightly:ssa on tuki WebGPU:lle. WebGPU on herättänyt kiinnostusta erityisesti sen tarjoaman laskentavarjostinohjelmatuen vuoksi. Tällä hetkellä esimerkiksi WebDNN (Hidaka ym. 2017) ja Tensorflow.js (Smilkov ym. 2019) (Tensorflow 2022) käyttävät toteutuksissaan WebGPU-rajapintaa.

2.5 Wgpu

Wgpu (Mages 2022a) on Rust-ohjelmointikielellä toteutettu, WebGPU-rajapintaan perustuva, grafiikka- ja laskentarajapinta. Wgpu toimii WebGPU:n tavoin välityskerroksena usealle eri grafiikka-rajapinnalle, erityisesti Vulkan-, Metal-, D3D12-, D3D11- ja OpenGL ES 3 -rajapinnoille 1. Wgpu:lla voidaan tehdä natiiveja ohjelmia, mutta se kääntyy myös WebAssemblyksi, jolloin ohjelmaa on mahdollista suorittaa myös web-selaimessa käyttäen selaimen WebGPU-rajapintaa. Wgpu toimii myös Firefox-selaimen WebGPU-rajapinnan sisäisenä toteutuksena.

Wgpu koostuu useasta eri moduulista 3. Wgpu-moduuli tarjoaa varsinaisen ohjelmointirajapinnan käyttäjille, wgpu-core on matalan tason sisäinen toteutus ja wgpu-hal on välitys-

kerros toisiin rajapintoihin. Wgpu-hal käyttää hyväkseen myös muita välityskerroksia, kuten esimerkiksi glow (Groves 2022) ja ANGLE-rajapintaa. Wgpu-rajapintaan on integroitu myös naga-kirjasto (Mages 2022b), jolla voidaan muuttaa eri varjostinohjelmakielillä kirjoitettuja ohjelmia toisille varjostinohjelmointikielille.



Kuvio 3: Wgpu:n arkkitehtuuri (Mages 2022c). Wgpu on varsinainen ohjelmointirajapinta, joka näkyy käyttäjälle. Wgpu-hal (hardware abrstraction layer) toimii abstraktiokerroksena eri rajapinnoille. Nagaa käytetään varjostinohjelmien muuntamisessa ja validoinnissa. Varjostinohjelmien lopullinen kääntäminen ja rajapintakutsut tapahtuvat käyttöjärjestelmän laiteajuritasolla (OS drivers).

3 Varjostinohjelmat

GPU:lla suoritettavia ohjelmia kutsutaan varjostinohjelmiksi (engl. *shader*). Varjostinohjelmat kirjoitetaan usein jollain täsmäkielellä kuten esimerkiksi OpenGL Shading Language-(lyhemmin GLSL), HLSL- tai MSL-kielellä. Eri grafiikka- ja laskentarajapinnat tukevat eri varjostinohjelmointikieliä. Varjostinohjelmien lähdekoodi muutetaan ajettavaksi GPU-ohjelmaksi grafiikka- ja laskentarajapinnan funktiolla, jolloin varjostinohjelmat käännetään suoritettavaan muotoon GPU-laitteen ajureilla. Tämän jälkeen GPU:lla sijaitsevaa varjostinohjelmaa voi suorittaa isäntäohjelmastasta käyttäen grafiikka- ja laskentarajapintaa. Joissain tapauksissa varjostinohjelmat käännetään ensin jollekin välikielelle, kuten esimerkiksi SPIR-V (Standard, Portable Intermediate Representation - V), josta näytönohjaimen ajuri kääntää koodin suoritettavaksi ohjelmiksi.

WebGPU käyttää varjostinohjelmointikielenään WGSL:ää (W3C 2022b), joka on myös tämän tutkielman sovelluksen käyttämä varjostinohjelmointikieli. WGSL on tätä kirjoitelmaa tehtäessä luonnosvaiheessa, mutta sillä voi jo nyt toteuttaa toimivia varjostinohjelmia. Natiivit rajapinnat eivät tue suoraan WGSL-kieltä. WGSL-ohjelmat muutetaan ajon aikana natiivien rajapintojen tukemiksi varjostinohjelmiksi, erityisesti SPIR-V-, HLSL- tai MSL-ohjelmiksi (Malyshau 2022a), jonka jälkeen natiivi rajapinta siirtää ohjelmat näytönohjaimen ajurille lopullista kääntämistä varten. Wgpu-rajapinta käyttää varjostinohjelmien välisiin muunnoksiin naga-kirjastoa. Tästä syystä wgpu tukee samoja varjostinohjelmointikieliä kuin naga. Nagan lähde- ja kohdekielituet on esitetty taulukoissa 2 ja 3. Googlen Dawn-rajapinnan käyttämä varjostinohjelmien muunnoskirjasto on nimeltään Tint (Google 2022c).

Käyttöjärjestelmätuet (wgpu)			
API	Windows	Linux & Android	macOS & iOS
Vulkan	✅	✅	🆗 (vulkan-portability)
Metal			✅
DX12	✅ (vain W10+)		
DX11	🛠️		
GLES3		🆗	
ANGLE	🆗	🆗	🆗 (vain macOS)

Taulukko 1: Wgpu-rajapintatuet eri käyttöjärjestelmille (Mages 2022a).

Nagan front-end tuet		
Front-end	Status	Huomiot
SPIR-V (binary)	✅	
WGSL	✅	Täysin validoitu
GLSL	🆗	Ainoastaan GLSL 440+ ja Vulkan-semantiikat

Taulukko 2: Naga-kirjaston lähdekielituet (Mages 2022b).

Nagan back-end tuet		
Back-end	Status	Huomiot
SPIR-V	✅	
WGSL	🆗	
Metal	✅	
HLSL	✅	Shader Model 5.0+ (DirectX 11+)
GLSL	🆗	GLSL 330+ ja GLSL ES 300+
DOT (GraphViz)	🆗	Ei ole varjostinohjelmointikieli

Taulukko 3: Naga-kirjaston kohdekielituet (Mages 2022b).

✅ = Ensisijainen tuki 🆗 = Best effort -tuki 🛠️ = Ei tuettu, mutta tuki on kehitteillä

4 Rust, JavaScript ja WebAssembly

Tutkimuksen sovellus kääntyy sekä natiiviksi ohjelmaksi että web-selaimessa toimivaksi web-sovellukseksi. Seuraavaksi käydään läpi tutkimuksen kannalta oleelliset ohjelmointikieliin liittyvät termit. Lisäksi käydään läpi se, miten Rust-ohjelmointikielellä toteutettu ohjelma käännetään web-selaimessa toimivaksi ohjelmaksi. Erityisen tärkeitä asioita ovat JavaScriptin ajoympräistö ja siihen liittyvä *Promise*-rakenne, koska ne aiheuttavat ongelmia tutkielman ohjelman web-käännöksessä.

Tutkielman sovellus on toteutettu Rust-ohjelmointikielellä. Rust (Team 2022) on Mozilla Foundationin kehittämä, staattisesti tyypitetty ohjelmointikieli, jossa ei ole roskienkeruuta. Rustista on pyritty kehittämään muistiturvallinen ja nopea ohjelmointikieli erityisesti matalantason ohjelmointiin, ja se on suunniteltu toimimaan myös web-selaimissa WebAssembly-muodossa.

JavaScript on dynaamisesti tyypitetty, ECMAScript-standardiin (International 2022a) perustuva, ohjelmointikieli. JavaScript on erityisen vahvasti tuettu web-selainympäristöissä, mikä mahdollistaa kielen käytön HTML-sivuissa. JavaScript-ajoympäristö on yksisäikeinen ja luonteeltaan asynkroninen. Tämä tarkoittaa sitä, että web-sivulla suoritettavaa JavaScript-koodia ei suoriteta välittömästi, vaan päätös operaatioiden suoritusjärjestyksestä jätetään ajoympäristön varaan. Ajoympäristö sisältää yhden pinokoneen, joka suorittaa yhtä JavaScript-koodia kerrallaan. Ajoympäristössä ylläpidetään useita eri jonoja, joihin tallennetaan asynkronisia operaatioita odottamaan suoritusvuoroaan. Tällä tavalla ajoympäristö pystyy aikatauluttamaan ja jakamaan suoritusvuoroja useamman eri operaation kesken.

JavaScriptin *Promise* (International 2022b) on rakenne, jonka avulla JavaScript-operaation suoritus voidaan jättää taka-alalle. *Promise* voi olla kolmessa eri tilassa: pending, fulfilled tai rejected. Jos *Promise*-objektiin kapseloitu operaatio ei ole saanut suoritustaan päätökseen, ajoympäristö asettaa sen pending tilaan. Tämän jälkeen *Promise* asetetaan ajoympäristön osoittamaan jonoon, ja suoritusvuoro annetaan jollekin toiselle JavaScript-operaatiolle. Tällä tavalla operaatio ei blokkaa koodia suorittavaa pinokonetta, ja sen myötä JavaScript-ajoympäristöä. Ajoympäristö tarkistaa aika ajoin *Promise*-objektin kapseloiman operaation

tilan. Jos operaatio ei ole saanut suoritustaan päätökseen, *Promise* jätetään pending tilaan ja se asetetaan takaisin jonoon. Jos suoritettava operaatio saadaan päätökseen, *Promise*-objekti siirtyy joko fulfilled tai rejected tilaan. Riippuen siitä kumpaan tilaan *Promise*-objekti jää, ohjelmoija voi määritellä jatko-operaatioita.

WebAssembly (Wasm) on W3C Community Groupin kehittämä, matalan tason binäärikoodiformaatti. WebAssembly tarjoaa eri alustoille helposti siirrettävän virtuaalikäskyjoukon, jota voidaan suorittaa eri ympäristöissä, kuten esimerkiksi web-selaimissa (Watt, Rossberg ja Pichon-Pharabod 2019). WebAssembly on matalantason kieli, jonka käskyt on suunniteltu siten, että ne kääntyvät suoraan koodia suorittavalle prosessorille. WebAssembly mahdollistaa myös sellaisten ohjelmien suorittamisen web-selaimissa, jotka eivät ole toteutettu JavaScriptilla. Toisin kuin JavaScript, WebAssemblya ei tulkita ja käännetä ajon aikana web-selaimessa toimiviksi ohjelmiksi, vaan ohjelma käännetään esimerkiksi Rust-kääntäjällä suoraan laitteistoläheiseen formaattiin. Tämä mahdollistaa esimerkiksi sen, että WebAssembly-koodi voidaan optimoida kokonaisuudessaan jo käännösvaiheessa. Käännöksen tuloksena syntyy Wasm-tiedosto, joka liitetään mukaan osaksi HTML-sivua. WebAssemblya ei ole suunniteltu ainoastaan web-selaimia varten, mutta toimiakseen web-selainympäristössä, se tarvitsee kuitenkin JavaScriptia. Käännöstyökalut generoivat tyypillisesti ohjelman suorittamista varten välttämättömät JavaScript-koodit.

Koska Rust-ohjelmointikielellä toteutettuja ohjelmia on mahdollista kääntää WebAssemblyksi, myös wgpu-kirjasto on suunniteltu siten, että se on käännettävissä WebAssembly-muotoon. Tätä tarkoitusta varten on kehitetty erilaisia työkaluja ja kirjastoja. WebAssemblysta voidaan kutsua Javascript-funktioita, ja JavaScriptista voidaan kutsua WebAssemblya. Tämän prosessin helpottamiseksi on kehitetty wasm-bindgen työkalu (rustwasm 2022). Wasm-bindgen auttaa määrittämään ne JavaScript-funktiot, joita halutaan kutsua Wasm-koodista tai Wasm-funktioita, joita halutaan kutsua JavaScriptistä. WebGPU-spesifien komponenttien määrittämiseen hyödynnetään wasm-bindgenin web-sys-kirjastoa. Wasm-bingen-futures-kirjasto toimii Rust-ohjelmointikielen *Future* (Rust-lang 2022) ja JavaScriptin *Promise*-rakenteen välisenä muunnoskirjastona. Rustin *Future* ja JavaScriptin *Promise* ovat kumpikin rakenteita asynkronisten operaatioiden käsittelyyn.

Rust-ohjelmointikielellä kirjoitettu koodi kääntyy suurimmalta osin WebAssemblyksi, mutta

esimerkiksi WebGPU-kutsujen kohdalla Wasm-koodi kutsuu JavaScript-rajapinnan mukaisia WebGPU-funktioita. JavaScript- ja Wasm-tietorakenteet eivät ole välttämättä suoraan yhteensopivia, vaan tätä tarkoitusta varten täytyy toteuttaa serialisointi- ja deserialisointifunktioita. Rust-kääntäjä ja apukirjastot generoivat automaattisesti tätä varten kaikki tarvittavat JavaScript- ja Wasm-koodit, joita tarvitaan, jotta ohjelma on suoritettavissa web-selaimessa. Prosessi on pitkälti automatisoitu, eikä käyttäjän tarvitse välttämättä tietää WebAssemblysta juuri mitään. Toisinaan automaattisesti generoitua JavaScript-koodia joudutaan muuttamaan käsin, kuten tämän tutkielman ohjelmassa on tehty. Tässä tapauksessa kyseessä on lähinnä funktioiden nimien muuttelua, mikä johtuu siitä, että wgpu ja Firefoxin WebGPU-rajapinta kehittyvät eri tahtiin. Nämä ongelmat korjaantuvat yleensä Firefoxin päivitysten myötä.

Tässä tutkielmassa wgpu-ohjelma käännetään WebAssemblyksi ja JavaScriptiksi. Käännös tehdään siihen tarkoitukseen toteutetuilla Rust-työkalulla, ja käännetty Wasm- ja JavaScript-tiedosto integroidaan osaksi HTML-sivua. Tällä tavalla Rust-ohjelmointikielellä kirjoitetut wgpu-ohjelmat on mahdollista suorittaa web-selainympäristössä. Tällä tavalla myös wgpu-kirjaston tukemat WGSL-laskentavarjostinohjelmat on mahdollista suorittaa web-selaimessa.

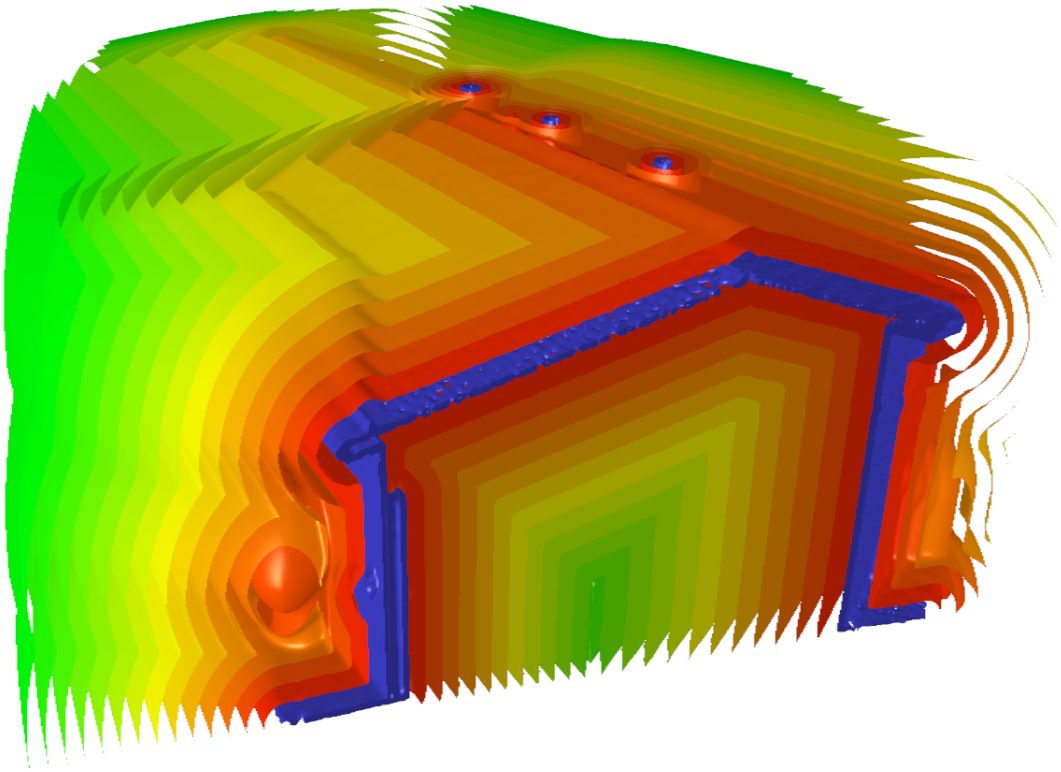
5 Eikonal-yhtälö

Tämän tutkielman sovelluksessa luodaan pistepilvidatasta etäisyyskenttä. Etäisyyskentän luonti toteutetaan käyttäen eikonal-yhtälön ratkaisualgoritmia. Seuraavaksi käydään läpi eikonal-yhtälö, ja esitellään joitakin yhtälön ratkaisualgoritmeja.

Eikonal-yhtälö on epälineaarinen Hamilton-Jacobi-osittaisdifferentiaaliyhtälö aallon reunan etenemisongelmien ratkaisemiseksi 5.1. Yhtälö tulee vastaan erilaisissa tilanteissa, jossa alkutilanteessa määritelty aallon reuna etenee monotonisesti itsestään pois päin reunan normaalin suuntaisesti. Yhtälöstä ratkaistaan tyypillisesti aallon reunan saapumisaikoja laskenta-alueella $\Omega \subset \mathbb{R}^n$. Yhtälöllä on erilaisia sovelluskohteita, kuten esimerkiksi konenäkö, kuvankäsittely, tietokonegrafiikka, geotieteet ja lääketieteellisten kuvien analysointi (Jeong ja Whitaker 2008). Yhtälö voidaan esittää seuraavassa muodossa.

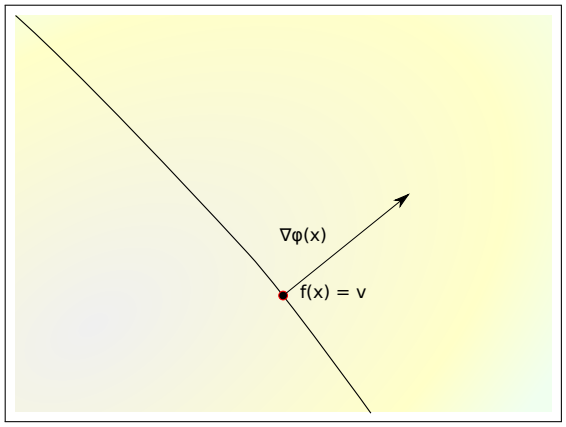
$$\begin{cases} \|\nabla \varphi(x)\| = 1/f(x), & x \in \Omega \\ \varphi(x) = g(x), & x \in \partial\Omega \end{cases} \tag{5.1}$$

Yhtälössä ratkaistavana oleva $\varphi(x)$ on aika, jolloin aallon reuna saavuttaa pisteen x. Yhtälössä $\nabla\varphi(x)$ on saapumisaikapinnan 4 gradientti pisteessä x. Funktio $f(x)$ määrittää reunan nopeuden pisteessä x. Jos $f(x) = 1$ koko laskenta-alueella, niin silloin $\varphi(x)$ on lyhin euklidinen etäisyys pisteestä x aallon alkureunaan $\partial\Omega$ (Jones, Baerentzen ja Sramek 2006). Nopeusfunktio f ei saa vaihtaa merkkiä, ja se on usein positiivinen. Jos $f(x) = 0$, niin tällöin aalto ei saavuta koskaan pistettä x (Tor Gillberg ym. 2014). Funktio g määrittää alkutilanteessa olevan aallon reunan saapumisajat. Useimmiten $g = 0$, jolloin aallon alkutila on objektin pinta, josta halutut saapumisajat tai euklidiset etäisyydet halutaan ratkaista.

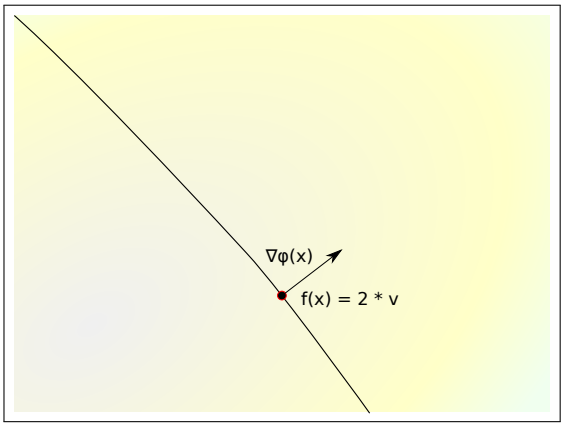


Kuvio 4: Tutkielman ohjelmalla luotu poikkileikkauskuva eikonal-yhtälön aallon eri saapumisajoista. Kuvassa aallon alkuperäinen reuna on esitetty sinisellä värillä. Etenevän aallon saapumisajat kasvavat punaisesta väristä kohti vihreää väriä. Tässä aallon etenemisnopeus on kaikkialla yksi. Tästä johtuen eikonal-yhtälön aallon arvot ovat tässä lyhyimpiä euklidisia etäisyyksiä aallon alkureunaan.

Nopeusinformaatio ja aallon alkureunan valinta määräävät aallon etenemisen ja sen myötä aallon saapumisajat. Yhtälöstä havaitaan, että aallon nopeus ja reunan gradientin normi ovat kääntäen verrannollisia. Tämä tarkoittaa sitä, että saapumisaikakentän kasvunopeus on kääntäen verrannollinen suhteessa aallon nopeuteen 5 6.

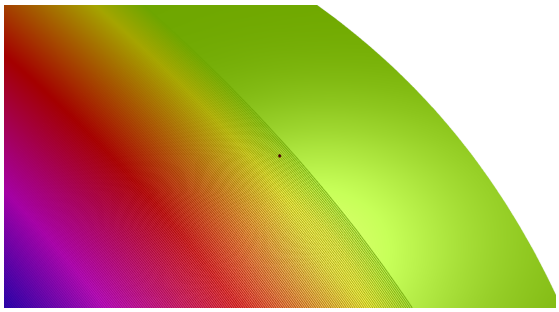


(a) Gradienttivektori ja nopeusarvo pisteessä x.

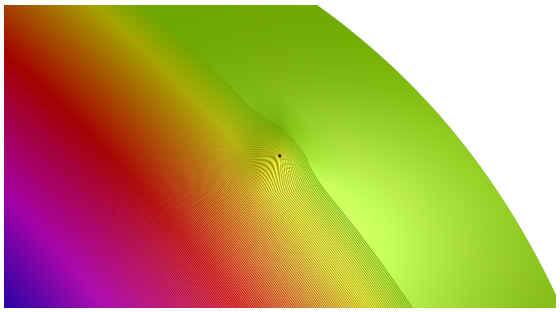


(b) Gradienttivektori pisteessä x, kun nopeusarvo kaksinkertaistetaan.

Kuvio 5: Kuvassa on esitetty eikonal-yhtälön gradienttivektori pisteessä x. Nopeusfunktio f antaa aallon nopeuden v pisteessä x. Gradienttivektori kuvaa saapumisaikakentän suurimman muutosnopeuden ja suunnan. Kuvissa havainnollistetaan nopeusarvon ja gradienttivektorin normin kääntäen verrannollisuutta. Oikeanpuoleisessa kuvassa nopeusarvo kaksinkertaistetaan, jolloin gradienttivektorin pituus pienenee samassa suhteessa. Tällöin kentän saapumisajat kasvavat hitaammin pisteen x kohdalla.



(a) Poikkileikkauskuva aallon etenemisestä. Musta piste on näytteistyspiste.



(b) Näytteistyspisteen nopeusarvo kaksinkertaistetaan.

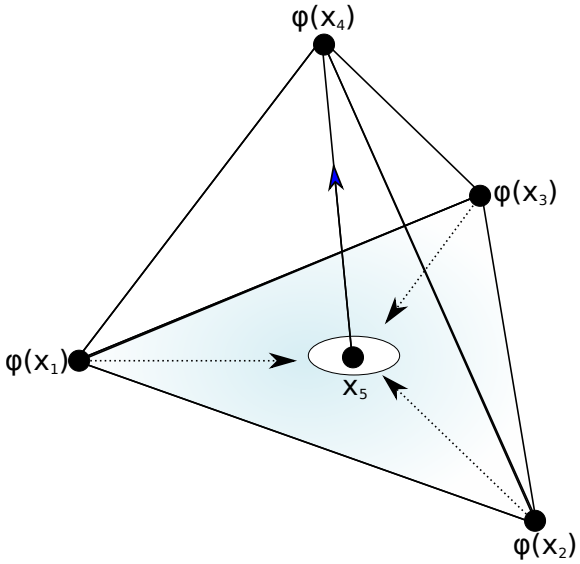
Kuvio 6: Ohjelmasta otettuja poikkileikkauskuvia 3D-laskenta-alueen aallon saapumisaikarintamasta. Saapumisaikakentän arvot kasvavat sinisestä väristä kohti vihreää väriä. Kuvassa havainnollistetaan saapumisaikakentän muutosta, kun yksittäisen eikonal-yhtälön laskenta-alueen pisteen nopeus kaksinkertaistetaan. Nopeuden kasvatus näkyy aallon saapumisaikojen muutoksena pisteen ympärillä. Nopeusarvoa voidaan tulkita aallon virtausnopeutena.

Nopeusinformaatio voi olla reaaliluku, jolloin arvo pitää sisällään ainoastaan aallon nopeuden. Tällaisessa tapauksessa suuretta, jossa laskenta-alueen nopeusinformaatiolla on vain määrä mutta ei suuntaa, kutsutaan isotrooppiseksi nopeusinformaatioksi. Jos nopeusinformaatiolla on määrän lisäksi myös suunta, nopeusinformaatiota sanotaan anisotrooppiseksi. Anisotrooppisen nopeusinformaation yhteydessä yhtälö 5.1 voidaan esittää seuraavassa muodossa, kun laskenta-alue on $\Omega \subset \mathbb{R}^3$.

$$\begin{cases} \sqrt{(\nabla\varphi(x))^T\, M(x)\, \nabla\varphi(x)} = 1, & x \in \Omega \\ \varphi(x) = g(x), & x \in \partial\Omega \end{cases} \tag{5.2}$$

Tässä $\varphi(x)$ on aallon saapumisaika ja $\nabla\varphi(x)$ on saapumisaikapinnan gradientti pisteessä x. Funktio g määrittää aallon reunan alkutilanteessa. Anisotrooppinen nopeusinformaatio on $M(x) \in \mathbb{R}^3 \to \mathbb{R}^3$ missä M on symmetrinen, positiividefiniitti $3x3$-matriisi (Fu, Kirby ja Whitaker 2013). Tässä yhteydessä matriisi M on metrinen tensori, joka määrittää laskenta-alueelle metriikan. Jos M on identiteettimatriisi koko laskenta-alueella Ω, yhtälöllä saadaan ratkaistuksi lyhin euklidinen etäisyys pisteestä x alkutilanteessa määritettyyn reunaan.

Anisotrooppisen nopeusinformaation kohdalla laskenta-alue on usein ositettu rajatilavuuksiin, kuten esimerkiksi nelitahokkaisiin 7. Tällöin alueen diskretisoidut aallon saapumisajat tallennetaan nelitahokkaiden kulmapisteisiin, ja metrinen tensori antaa metriikan yksittäisten nelitahokkaiden sisään. Tällä tavalla etenevän aallon laskentaan saadaan mukaan myös nopeuden lisäksi nopeuden suunta.



Kuvio 7: Esimerkki Ganellarin ja Haasen nelitahokas rakenteesta (Ganellari ja Haase 2016). Aallon sisään tulo x_5 määräytyy nelitahokkaan x_1, x_2 ja x_3 eikonal-arvoista. Aallon saapumisaikaa $\varphi(x_4)$ päivitetään pisteestä x_5 pisteeseen x_4 kulkevan aallon reunan gradienttivektorin mukaan. Nelitahokkaalle määritelty metrinen tensori antaa metriikan nelitahokkaan sisään.

5.1 Viskositeettiratkaisu

Staattisille Hamilton-Jacobi-yhtälöille on tyypillistä se, että ne tuottavat moniarvoisia ratkaisuja (Tor Gillberg ym. 2014). Eikonal-yhtälön kohdalla ollaan usein kiinnostuttu pienimmästä mahdollisesti ratkaisusta, jota kutsutaan viskositeettiratkaisuksi (Michael G. Crandall 1983). Tätä ratkaisua voidaan tulkita myös ensimmäiseksi aallon saapumisajaksi (Tor Gillberg ym. 2014). Euklidisen etäisyysfunktion tapauksessa viskositeettiratkaisu on lyhin etäisyys aallon alkuperäiseen reunaan.

(a) Moniarvoinen ratkaisu.

(b) Yksikäsitteinen ratkaisu.

Kuvio 8: Esimerkkejä eikonal-yhtälön erilaisista ratkaisuista.

5.2 Yhtälön ratkaisualgoritmeja

Eikonal-yhtälöä ei voida yleensä ratkaista analyyttisesti, vaan approksimaatio yhtälön ratkaisulle haetaan numeerisesti jollakin algoritmilla. Tyypillisesti tarkasteltavana oleva alue diskretisoidaan äärelliseen määrään näytteistyspisteitä. Algoritmille määritellään aallon reunan alkutila, mikä tarkoittaa sitä, että lasketaan reunan läheisyydessä oleviin näytteistyspisteisiin ensimmäiset arvot. Laskenta-alueelle määritellään myös nopeusinformaatio. Algoritmit laskevat kaikkiin laskenta-alueen näytteistyspisteisiin aallon saapumisajat. Tämän jälkeen laskenta-alueesta voidaan laskea aallon saapumisaika tietyssä koordinaatissa interpoloimalla arvo laskenta-alueen näytteistyspisteistä.

Yhtälön ratkaisuun kehitetyt algoritmit perustuvat laskenta-alueen näytteistyspisteiden naapuriarvojen vertailuun ja niihin määritettyihin nopeusarvoihin. Laskennassa pyritään etenemään pienemmistä arvoista kohti suurempia arvoja käyttämällä jotain tunnettua viskositeettiratkaisun approksimointiskeemaa, kuten esimerkiksi Godunovin-tyyppistä Rouy-Tourin skeemaa (Rouy ja Tourin 1992) (Yang 2019). Rouy-Tourin skeema voidaan kirjoittaa seuraavaan muotoon:

$$\begin{aligned}&\big(\max(D_{i,j,k}^{-x}\varphi, -D_{i,j,k}{}^{+x}\varphi, 0)^2 + \\ &\max(D_{i,j,k}{}^{-y}\varphi, -D_{i,j,k}{}^{+y}\varphi, 0)^2 + \\ &\max(D_{i,j,k}{}^{-z}\varphi, -D_{i,j,k}{}^{+z}\varphi, 0)^2\big)^{1/2} = \frac{1}{F_{i,j,k}}.\end{aligned} \tag{5.3}$$

Erityisesti karteesisen koordinaatiston mukaisesti diskretisoidussa laskenta-joukossa operaattorit

$$\begin{aligned}D_{i,j,k}{}^{+x} &= \frac{\varphi_{i+1,j,k} - \varphi_{i,j,k}}{\Delta x} \\ D_{i,j,k}{}^{-x} &= \frac{\varphi_{i,j,k} - \varphi_{i-1,j,k}}{\Delta x}\end{aligned} \tag{5.4}$$

määrittävät äärelliset eteen- ja taaksepäin erotusosamäärät derivaatalle $\partial\varphi/\partial x$. Vastaavat derivaatat saadaan y- ja z-suuntiin. Laskenta-alueen pisteiden indeksoinnissa käytetään koordinaatiston mukaisia indeksejä i, j ja k. Derivaatat lasketaan kunkin akselin mukaisista naapuriarvoista siten, että pienempi naapuriarvo valitaan mukaan laskentaan. Tällä tavalla saa-

daan määriteltyä etenevän aallon gradienttivektorin suunta. Rouy-Tourin skeemassa $F_{i,j,k}$ on aallon nopeus laskenta-alueen pisteessä (i, j, k). Rouy-Tourin skeema kirjoitetaan usein kvadraattiseen muotoon (Huang 2021) ennen kaavan toteuttamista ohjelmakoodiin.

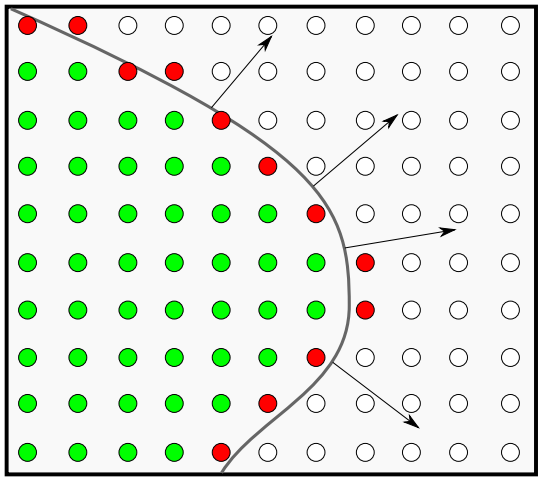
Eikonal yhtälöllä on erityyppisiä ratkaisualgoritmeja, kuten esimerkiksi fast marching method, fast sweeping method ja fast iterative method. Myös erilaisia hybridi-algoritmeja on kehitetty, kuten esimerkiksi the heap-cell method (Chacon ja Vladimirsky 2015).

5.2.1 Fast marching method

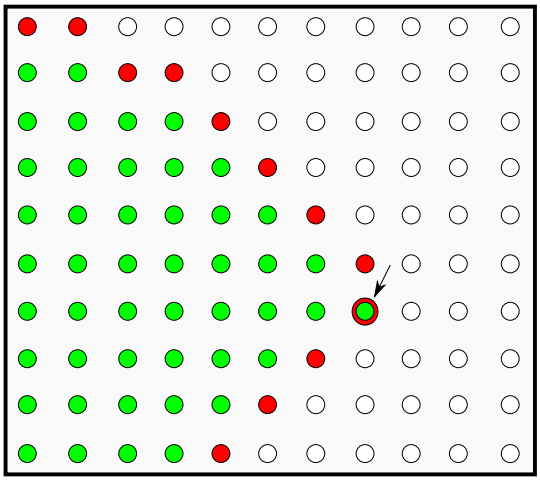
Fast marching method (lyhyemmin FMM) on numeerinen eikonal-yhtälön ratkaisualgoritmi, joka on variaatio Dijkstran algoritmista (Dijkstra 1959). FMM:n esittivät ensimmäisinä Tsitsiklis (Tsitsiklis 1994) ja Sethian (Sethian 1996) toisistaan riippumatta (Jones, Baerentzen ja Sramek 2006).

FMM:n laskenta-alueen näytteistyspisteet jaetaan tyypillisesti kolmeen eri kategoriaan: tunnettuihin (engl. *known*), kapean nauhan (engl. *narrow band*) ja kaukaisiin (engl. *far*) solmuihin. Tunnetut solmut ovat solmuja, joihin on jo laskettu lopullinen eikonal-yhtälön viskositeetti ratkaisu, eikä niiden arvot enää muutu. Kapean nauhan solmut ovat tunnettujen solmujen naapuriarvoja, joiden arvot voivat vielä muuttua. Kaukaiset solmut ovat solmuja, joiden arvoja ei ole vielä laskettu lainkaan. Algoritmissa eikonal-arvoja lasketaan ainoastaan tunnettujen arvojen perusteella.

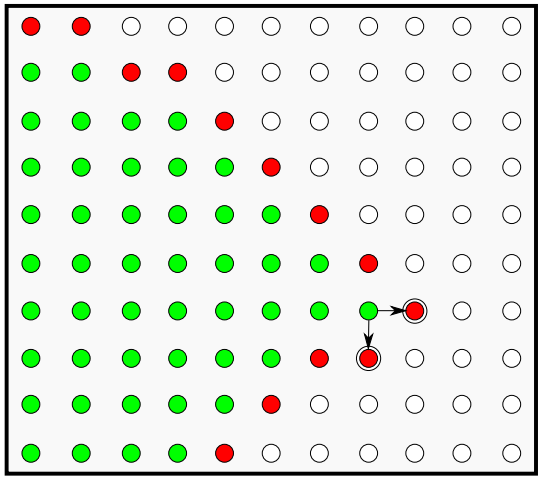
Kapean nauhan solmut kuuluvat kapean nauhan vyöhykkeeseen. Kapea nauha on algoritmissa sen hetkinen aallon etenemisrintama 9a. Kapean nauhan joukosta valitaan aika ajoin pienimmän arvon omaava solmu, joka sitten muutetaan tunnetuksi solmuksi 9b. Tämän jälkeen solmun kaukaiset naapurisolmut otetaan mukaan kapeaan nauhaan 9c. Uuden tunnetun solmun kaikki kapeannauhan pisteiden arvot lasketaan uudestaan 9d, sillä uusi tunnettu solmu saattaa vaikuttaa sen viereisten kapean nauhan solmujen arvoihin.



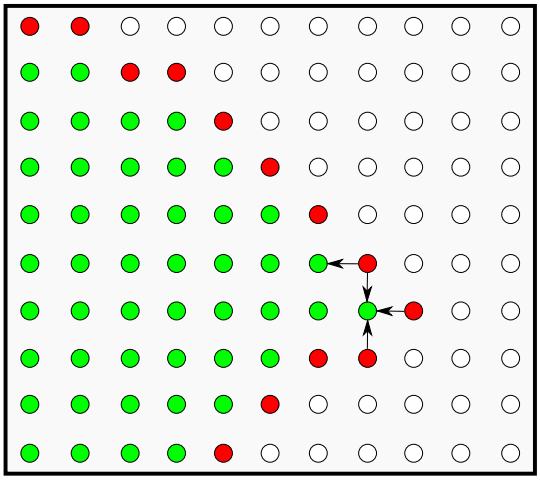
(a) Aallon etenemissuunta on aallon reunan normaalin suuntaan. Nuolet osoittavat etenemissuunnan.



(b) Nuolen osoittama solmu omaa kapean nauhan solmuista pienimmän arvon. Solmu poistetaan kapeasta nauhasta ja muutetaan tunnetuksi solmuksi.



(c) Kapeaan nauhaan lisätään uuden tunnetun solmun far-naapurisolmut. Nuolet osoittavat mukaan otettavat solmut.



(d) Uuden tunnetun solmun viereisten kapean nauhan solmujen arvot päivitetään nuolten osoittamista tunnetuista solmuista.

Kuvio 9: Kuvaus FMM:n etenemisestä. Vihreät solmut ovat tunnettuja solmuja, joiden arvot eivät enää muutu. Punaiset nauhasolmut muodostavat etenevän kapean nauhan vyöhykkeen. Valkoiset solmut ovat kaukaisia solmuja, joiden arvot ovat vielä määrittämättä. Kohdat b, c ja d muodostavat silmukan, jota toistetaan, kunnes kapea nauha on kulkenut laskenta-alueen läpi. Algoritmin suorituksen lopussa kaikki solmut ovat tunnettuja solmuja.

Algoritmi etenee tiukasti pienimmästä arvosta kohti suurempia arvoja. Tästä syystä kapeannauhan joukkoa ylläpidetään aputietorakenteessa, joka on usein minimi kekorakenne (engl. *min heap*). Tähän tietorakenteeseen tallennetaan, päivitetään ja poistetaan kapean nauhan solmuja käyttäen kekolajittelualgoritmia. Aputietorakenne toimii kiihdytysrakenteena kapean nauhan pisteiden käsittelyssä. Algoritmi on luonteeltaan toistorakenne, jota toistetaan, kunnes kaikki solmut ovat tunnettuja solmuja. Algoritmin toiminta on esitetty kuvassa 9.

Koska algoritmin tunnettuja arvoja ei voi enää muokata, algoritmin täytyy kontrolloida jollakin aputietorakenteella, että se etenee aina pienimmästä arvoista kohti suurempia arvoja. Tästä syystä FMM on vaikea rinnakkaistaa, koska laskennalla on tarkka suoritusjärjestys. FMM:sta on olemassa rinnakkaistettuja versioita, kuten esimerkiksi (Yang 2019). Tätä tutkielmaa tehdessä toteutettiin GPU:lla rinnakkaistettu FMM, mutta sitä ei otettu käyttöön itse sovellukseen.

5.2.2 Fast sweeping method

Fast sweeping method (lyhemmin FSM) (Tsai ym. 2003) on eikonal-yhtälön ratkaisualgoritmi, jossa laskenta-alueen arvoja lasketaan siten, että alue pyyhkäistään läpi eri suunnista. Päivitetyt arvot lasketaan entuudestaan tunnettujen pienempien arvojen perusteella. Esimerkiksi karteesisen ruudukon tapauksessa pyyhkäisysuuntia on 2^n missä n on dimensioiden lukumäärä. Pyyhkäisyjen myötä laskenta-alueiden pisteet konvergoituvat kohti viskositeettiratkaisua.

5.2.3 Fast iterative method

Fast iterative method (lyhemmin FIM) (Jeong ja Whitaker 2008) on eikonal-yhtälön ratkaisualgoritmi, joka on riippumaton tietystä laskentajärjestyksestä. Algoritmin ei tarvitse ylläpitää mitään erityistä järjestettyä tietorakennetta. Tämän lisäksi laskenta-alueen näytteistyspisteitä voidaan päivittää samanaikaisesti. Tästä syystä FIM on helposti rinnakkaistuva algoritmi, ja se sopii hyvin GPU:lla suoritettavaksi.

Algoritmin näytteistyspisteet jaotellaan lähde- ja aktiivipisteiksi. Lähdepisteet ovat pisteitä, joihin on laskettu arvo, ja joiden arvoja hyödynnetään, kun lasketaan toisia eikonal-arvoja.

FIM ylläpitää kapean nauhan listaa nimeltään aktiivilista (engl. *active list*) (Jeong ja Whitaker 2008). Listassa ylläpidetään laskenta-alueen pisteitä, joiden arvo voi muuttua.

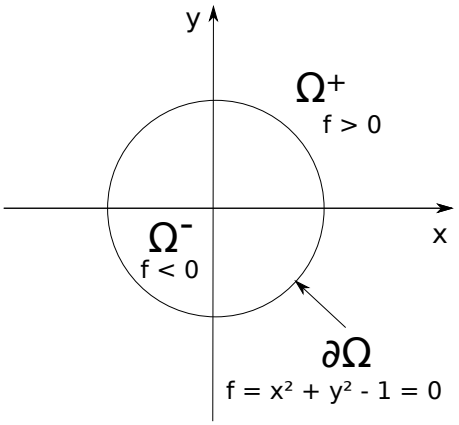
Algoritmin alussa määritellään ensimmäiset lähdepisteet. Näiden pisteiden naapuripisteet otetaan mukaan aktiivilistaan, jos ne eivät ole itse lähdepisteitä. Tämän jälkeen algoritmi etenee iteraatioissa. Iteraatiossa aktiivilistan arvot lasketaan rinnakkain, ja jos jokin arvo ei muutu, piste poistetaan aktiivilistasta. Poistetun pisteen naapuripisteiden arvot lasketaan uudestaan, ja jos jonkun naapuripisteen arvo muuttuu, piste otetaan mukaan aktiivilistaan. Iteraatioita toistetaan, kunnes yksikään arvo ei muutu, jolloin aktiivilista jää tyhjäksi.

Toisin kun FMM:n kohdalla, FIM:lle on tyypillistä se, että jo tunnetut arvot voivat muuttua. Eikonal-pisteiden arvot pysyvät joko muuttumattomina tai pienenevät. Arvot suppenevat iteraatioiden myötä kohti viskositeettiratkaisua. Algoritmi on saatu päätökseen, kun yksikään arvo ei enää muutu.

6 Epäsuorat pinnat ja niiden visualisointi

Tässä työssä luodaan ohjelma, joka visualisoi etäisyyskenttää sphere tracing -tekniikalla. Kyseessä on epäsuoran pinnan visualisointi, jossa hyödynnetään etäisyysfunktiota. Seuraavaksi käydään läpi epäsuorat pinnat, euklidinen etäisyysfunktio ja sphere tracing -tekniikka.

6.1 Epäsuora pinta



Kuvio 10: Epäsuora esitysmuoto yhtälöstä $x^2 + y^2 = 1$.

Olkoon $\Omega \subset \mathbb{R}^n$. Epäsuora funktio (engl. *implicit function*) määritellään funktion $f : \Omega \to \mathbb{R}$ tasa-arvojoukkona

$$\{p \in \Omega \mid f(p) = 0\} \tag{6.1}$$

Joidenkin yhtälöiden arvojoukot voidaan jakaa kolmeen eri osa-alueeseen: pinnan ulkopuoli Ω^+, sisäpuoli Ω^- ja reuna $\partial\Omega$, ja jossa reuna jakaa selvästi alueen sisä- ja ulkoalueeseen (Osher ja Fedkiw 2003). Vaikka esimerkin 10 kohdalla reunan määrittäminen on melko yksinkertaista, yleisesti käyrien kohdalla reunan määrittäminen ei välttämättä ole helppoa (Osher ja Fedkiw 2003). Jos $\Omega \subset \mathbb{R}^3$ niin tällöin tasa-arvojoukkoa

$$\{p \in \Omega \mid f(p) = a\} \tag{6.2}$$

kutsutaan termillä *isosurface* ja vakio a on *isovalue*. Tutkielman ohjelmalla voidaan luoda kuvia etenevän aallon saapumisaikapinnoista 4. Kuvassa samalla värillä esitetyt tasa-arvojoukot edustavat pistejoukkoa (*isosurface*), jossa aallon saapumisaika a on sama. Kuvassa on visualisoitu useampi pistejoukko, joissa vakio $a \in \{0,4,8,\ldots,60\}$ tarkoittaa lyhyimpiä euklidisia etäisyyksiä aallon lähtöreunaan.

6.2 Etäisyysfunktiot

Olkoon $\Omega \subset \mathbb{R}^n$. Etäisyysfunktio (engl. *distance function*) on jatkuva funktio, joka antaa pisteestä $p \in \Omega$ lyhyimmän etäisyyden reunaan $\partial\Omega$. Etäisyysfunktio $d : \Omega \to \mathbb{R}$ voidaan määritellä seuraavasti:

$$d(p) = \inf \|x - p\|, \qquad x \in \partial\Omega \tag{6.3}$$

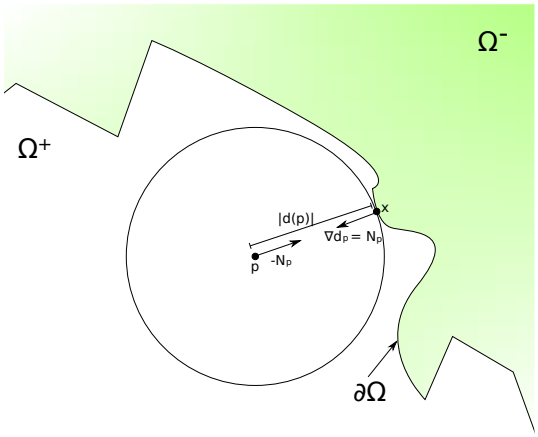
Etäisyysfunktion gradientin normille on voimassa ehto $\|\nabla d\| = 1$ lähes kaikkialla (Jones, Baerentzen ja Sramek 2006). Gradienttia ei ole kuitenkaan määritelty, jos pisteellä p ei ole yksikäsitteistä lyhyimmän etäisyyden omaavaa reunapistettä. Jos gradientti on määritelty pisteessä $p \in \Omega$, niin tällöin gradientti on kohtisuorassa lyhimmän etäisyyden reunapisteestä $x \in \partial\Omega$ pisteeseen p 11. Koska etäisyysfunktioiden kohdalla gradientti on kaikkialla yksi, gradientti voidaan tulkita myös normaalivektoriksi.

Etumerkillinen etäisyysfunktio (engl. *signed distance function*) on epäsuora funktio ϕ siten, että $|\phi(p)| = d(p)$ *kaikilla* $p \in \Omega$ ja

$$\phi(p) = \begin{cases} 0\,, & p \in \partial\Omega \\ -d(p)\,, & p \in \Omega^- \\ d(p)\,, & p \in \Omega^+ \end{cases} \tag{6.4}$$

Funktion etumerkki määräytyy sen mukaan, sijaitseeko piste p reunan sisä- tai ulkopuolella. Etäisyysfunktioille on yhteistä se, että ne ovat jatkuvia kaikkialla (Jones, Baerentzen ja Sramek 2006). Etumerkittömät etäisyysfunktiot ovat differentioituvia kaikkialla paitsi pis-

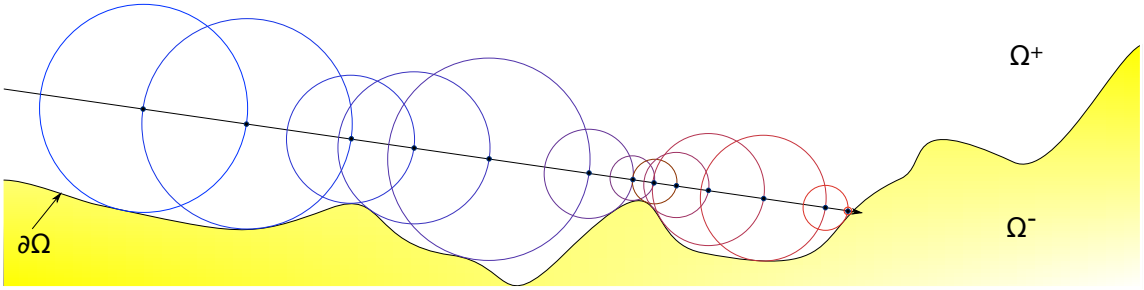
teissä, joissa on useampi kuin yksi lähimmän etäisyyden omaava reunapiste, tai jos pisteet sijaitsevat reunalla (Jones, Baerentzen ja Sramek 2006). Jos etumerkittömän etäisyysfunktion piste sijaistee reunalla, derivaattojen määrittäminen vaikeutuu (Osher ja Fedkiw 2003). Etumerkilliset etäisyysfunktiot ovat monotonisia reunalla. Tästä syystä ne ovat paremmin differentioituvia reunalla verrattuna etumerkittömiin etäisyysfunktioihin (Osher ja Fedkiw 2003). Reunan derivaattojen määrittäminen on tärkeää visualisointivaiheessa, koska niiden avulla voidaan määrittää pinnan normaalivektori. Pinnan normaalivektoria tarvitaan valaistuksen laskemisessa.



Kuvio 11: Jos etäisyysfunktion d gradientti on olemassa pisteessä p, gradienttia ∇d_p voidaan käyttää hyväksi lähimmän reunapisteen x määrittämisessä. Koska $\|\nabla d\| = 1$ niin etäisyysfunktion gradientti pisteessä p voidaan tulkita normaalivektoriksi. Merkitään $\nabla d_p = N_p$. Lähin reunapiste saadaan seuraavasti: $x = p + d(p) \cdot (-N_p)$.

Etäisyysfunktioita voidaan määritellä erilaisille geometrisille kappaleille, ja niitä voidaan yhdistellä eri operaatioilla kuten esimerkiksi yhdisteellä ja leikkauksella (Osher ja Fedkiw 2003). Inigo Quilez on koonnut erilaisia GLSL-varjostinkielen toteutuksia etäisyysfunktiolle (Quilez 2022). Kaikkia etäisyysfunktioita ei voida kuitenkaan määritellä valmiiden etäisyysfunktioiden avulla. Tällöin laskenta-alue Ω voidaan diskretisoida, ja alueen pisteille määritellään etäisyysarvot jollakin menetelmällä. Lopputuloksena saadaan diskreetti skalaarikenttä, jossa kentän arvot ovat lyhyimpiä etäisyyksiä laskenta-alueen geometriaan. Tällaista diskretisoitua skalaarikenttää kutsutaan etäisyyskentäksi (engl. *distance field*). Tällöin etäisyysfunktio voidaan määritellä etäisyyskentän arvojen avulla käyttäen interpolointia.

6.3 Sphere tracing



Kuvio 12: Kuva yksittäisen säteen etenemisestä sphere tracing -algoritmissa. Tarkoituksena on etsiä säteen ja epäsuoran pinnan törmäyskoordinaatti. Säde etenee säteen vektorin suuntaisesti etäisyysfunktion antaman etäisyyden verran eteenpäin. Tätä operaatiota toistetaan kunnes etäisyysfunktion antama arvo on riittävän pieni.

Sphere tracing on tekniikka epäsuorien pintojen visualisointiin. Algoritmi käyttää hyväkseen tietoa geometrisista etäisyyksistä (Hart 1996). Kyseessä on eräänlainen säteenheittomenetelmä (engl. *ray casting*), jossa sädettä pitkin edetään etäisyysfunktion antaman etäisyyden verran eteenpäin. Etäisyysfunktio $f \in \Omega \rightarrow \mathbb{R}$. antaa lyhyimmän etäisyyden visualisoitavan kappaleen reunaan. Saadun etäisyysarvon avulla voidaan määritellä pisteen kohdalle ympyrä, tai kolmiulotteisen avaruuden kohdalla pallo, jonka sisäpuolella ei voi olla reunapisteitä. Lähin reunapiste sijaitsee ympyrän tai pallon pinnalla. Sädettä voidaan siis edetä etäisyysfunktiosta saadun arvon verran eteenpäin ilman että säde ohittaa visualisoitavaa pintaa.

Visualisoitava geometria on usein epäsuora pinta, jolla ei ole välttämättä tarkasti määriteltyä reunaa. Tästä syystä algoritmille määritellään ε-arvo, joka määrittää etäisyyden, jolloin ollaan riittävän lähellä visualisoitavaa pintaa. Jos etäisyysfunktion antama arvo on pienempi kuin ε-arvo, säde törmää visualisoitavan objektin pintaan. Tällä tavalla pyritään etsimään riittävän hyvä approksimaatio säteen ja pinnan väliselle törmäyspisteelle. Säteen törmäyspisteen määrittämisestä voidaan luopua, jos sädettä on edetty liian pitkälle tai jos hyppyjä on tehty tietty määrä. Algoritmia voidaan myös optimoida monin eri tavoin (Keinert ym. 2014) (Bálint ja Valasek 2018).

7 Tutkimus

Tässä tutkimuksessa tutkitaan wgpu-rajapinnan käytettävyyttä ympäristöriippumattomaan GPU-laskentaan. Wgpu mahdollistaa WGSL-laskentavarjostinohjelmien käytön Windows-, Linux- ja macOS-käyttöjärjestelmissä. Natiivikäännösten lisäksi wgpu mahdollistaa WebAssembly-käännöksen, ja sen myötä WGSL-ohjelmien suorittamisen web-selaimessa. Tutkimuksen tarkoituksena on selvittää se, onko wgpu teknologiana jo niin kehittynyt, että sen avulla voidaan kehittää GPU-laskentaan perustuvia ohjelmia.

Tutkimusta varten toteutetaan wgpu-rajapintaa käyttämä tietokoneohjelma, joka rakentaa pistepilvidatasta euklidisen etäisyyskentän käyttäen eikonal-yhtälön ratkaisualgoritmia. Ohjelma visualisoi etäisyyskenttään tallennettua geometriaa sädetekniikalla. Ohjelman algoritmit on ohjelmoitu WGSL-varjostinohjelmointikielellä, mikä mahdollistaa GPU-laskennan testaamisen. Koska kyseessä on ohjelman kehityksen rinnalla tehtyä tutkimusta, tutkimusmenetelmäksi valitaan suunnittelutiede (Hevner ym. 2004).

Ohjelman natiivit versiot ja WebAssembly-käännös suoritetaan Windows-, Linux- ja macOS-käyttöjärjestelmissä. WebAssembly-versio ladataan samalta web-sivulta kaikkiin testauskoneisiin. Tällä tavalla varmistetaan se, että sama WebAssembly-ohjelma suoritetaan kaikissa testikoneissa. Tutkimuksen kannalta oleellisimmat natiivit rajapinnat ovat Vulkan, D3D12 ja Metal, koska ne ovat wgpu:n kehittäjien mukaan ensisijaisesti tuettuja rajapintoja 1. WebGPU-rajapintaa testataan käyttäen Firefox Nightly web-selainta.

Tärkein wgpu:n käytettävyyttä mittaava ominaisuus on ohjelman toimivuus. Tässä tutkimuksessa toimivuus tarkoittaa sitä, että ohjelma luo pistedatasta etäisyyskentän ja visualisoi etäisyyskenttään tallennettua geometriaa. Ohjelman toimiessa testaajan tulisi nähdä ohjelman piirtämä kuva 1. Jos testattava ohjelma ei toimi odotetulla tavalla, testaaja dokumentoi ohjelman tulostamat logitiedot ja mahdolliset virheilmoitukset. Käytettävyyttä pyritään arvioimaan myös ohjelman kehityksen aikana tulevista havainnoista. Ohjelman kehityksen yhteydessä esiintyvät wgpu:n toiminnallisuuden ongelmat dokumentoidaan, jos ne vaikuttavat merkittävästi ohjelman toimivuuteen tai algoritmien toteutukseen.

8 Ohjelman toteutus

Isäntäohjelma ohjelmoidaan Rust-ohjelmointikielellä, ja se perustuu wgpu-rajapintaan. Ohjelman käyttämät algoritmit toteutetaan WGSL-varjostinohjelmointikielellä. Naga-kirjastoa käytetään varjostinohjelmien välisiin muunnoksiin ja validointiin. Ikkunan hallinnointiin ja tapahtuman käsittelyyn käytetään winit-kirjastoa (rust-windowing 2022).

Vaikka pääohjelma toteutetaan Rustilla, varsinainen algoritmeihin liittyvä logiikka sijoitetaan pääosin WGSL-ohjelmiin. WGSL-ohjelmat suoritetaan näytönohjaimella, ja ne suunnitellaan Single Instruction Multiple Data (lyhyemmin SIMD) (Flynn 1972) arkkitehtuurin mukaisesti rinnakkaisuuden mahdollistamiseksi.

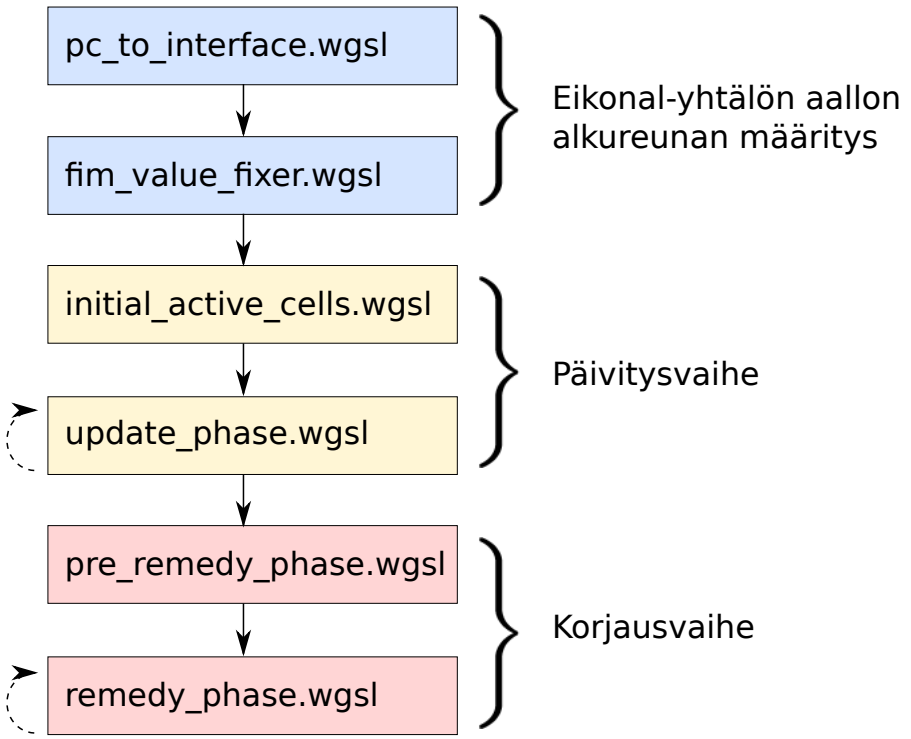
GPU:lla suoritettavat käskyt suoritetaan rinnakkaisesti siten, että useampi säie suorittaa samanaikaisesti samaa komentoa. Tämä johtaa usein tilanteisiin, jossa säikeet tekevät yhtäaikaisesti luku- tai kirjoitusoperaatioita samoihin muistipaikkoihin. Samanaikaiset luku- ja kirjoitusoperaatiot halutaan toisinaan määritellä atomisiksi siten, että ainoastaan yksi säie voi tehdä luku- tai kirjoitusoperaation kerrallaan. Tätä tarkoitusta varten WGSL-kieleen on määritelty useita erilaisia atomisia funktioita (W3C 2022f). Atomiset funktiot ovat merkittävässä roolissa tutkimuksen WGSL-ohjelmissa. Tutkimuksen sovellus voidaan jakaa loogisesti kolmeen eri osaan: ohjelman alustukseen, etäisyyskentän generointiin ja datan visualisointiin.

8.1 Ohjelman alustus

Ohjelman alustuksessa luodaan kaikki ohjelman suorituksen kannalta tarvittavat resurssit. Tässä vaiheessa ohjelma voidaan pakottaa käyttämään jotain tiettyä natiivia grafiikka- ja laskentarajapintaa, kuten esimerkiksi Vulkania. Ohjelman alustuksen yhteydessä validoidaan ja käännetään kaikki ohjelman tarvitsemat WGSL-ohjelmat. Alustusvaiheessa määritellään myös laskenta-alueen dimensiot. Laskenta-alue on tässä ohjelmassa kolmiulotteinen karteesinen ruudukko. Ohjelma tarkastaa myös sen, että käytettävissä oleva GPU-laite täyttää ohjelman vaatimukset. Tämä vaihe on tutkielman kannalta erityisen kriittinen vaihe, sillä osa testiajojen ongelmista ilmenevät tässä vaiheessa.

8.2 Etäisyyskentän luonti

Etäisyyskentän luonti toteutetaan eikonal-yhtälön ratkaisualgoritmilla, joka pohjautuu Yuhao Huangin esittämään FIM-algoritmiin (Huang 2021). Ohjelman algoritmista on erotettavissa selvästi seuraavat vaiheet: alkureunan rakentaminen, päivitys- ja korjausvaihe. FIM-algoritmi on jaettu kuuteen peräkkäin suoritettavaan WGSL-ohjelmaan 13.



Kuvio 13: Kuvassa on esitetty FIM-algoritmin kaikki WGSL-ohjelmat. Nuolet kertovat ohjelmien suoritusjärjestyksen. Sinisellä värillä on merkitty eikonal-yhtälön alkureunan rakennusvaiheen ohjelmat. Keltaisella värillä esitetään päivitysvaiheen ohjelmat ja punaisella värillä esitetään korjausvaiheen ohjelmat. Katkoviivanuolilla esitetyt ohjelmat perustuvat iteraatioiden toistamiseen. Muut ohjelmat ovat luonteeltaan läpipyyhkäisyalgoritmeja.

8.2.1 Aallon alkureunan rakentaminen

Algoritmissa määritellään aluksi eikonal-yhtälön aallon alkureuna. Alkureuna luodaan käyttäen pistepilvidataa. Yksittäinen pistepilvidatan piste pitää sisällään tiedon pisteen koordinaatista ja väriarvosta. Isäntä-ohjelma laskee tarvittavat skaalauskertoimet, jotta data saadaan sovitettua eikonal-yhtälön laskenta-alueen sisälle. Varsinainen aallon alkureunan mää-

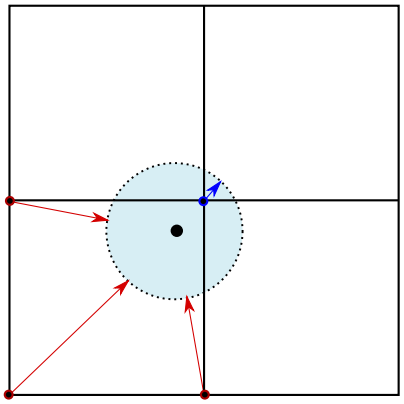
rittäminen toteutetaan GPU:lla käyttäen *pc_to_interface.wgsl*-ohjelmaa.

Pistedatan pisteillä ei ole tilavuutta. Tästä johtuen sphere tracer -visualisointi ei tuota järkevää kuvaa, koska säteet eivät juurikaan koskaan osu suoraan pisteisiin. Tästä syystä pistedatan pisteet tulkitaan palloiksi. Pallojen säteiden määrittäminen vaikuttaa merkittävästi etäisyyskentän arvoihin, mikä näkyy myös selvästi visualisoinnissa 23a 23b 23c 24a 24b 24c.

Pallojen etäisyydet lasketaan niitä ympäröiviin eikonal-pisteisiin 14, mutta ainoastaan lyhyimmät etäisyydet tallennetaan. Uuden pienemmän etäisyyden löytyessä tallennetaan myös pisteen väri. Tällä tavalla saadaan määritettyä aallon reunalle lyhyimmät etäisyydet pistedatasta muodostettuihin palloihin ja väriarvot. Eikonal-pisteet, joihin tallentuu etäisyysarvo luokitellaan lähdepisteiksi 15a 21a.

Aallon alkureunan rakennusvaihe on rinnakkaistettu 20. Tästä syystä vain yksi säie saa suorittaa kerrallaan etäisyysarvon vertailun ja päivityksen samaan muistipaikkaan. Tämä toiminnallisuus toteutetaan WGSL:n *atomicMin*-funktion avulla. Funktio ottaa argumenttina viiteen muistipaikkaan ja kokonaislukuarvon. Jos argumenttina annettu arvo on pienempi kuin muistipaikassa sijaitseva arvo, pienempi arvo korvaa muistipaikan arvon. Muussa tapauksessa funktio ei muuta muistipaikan arvoa. Funktio päivittää säieturvallisesti muistipaikkaan pienimmän arvon.

Koska WGSL:n atomiset funktiot voivat käsitellä ainoastaan kokonaislukuja, etäisyydet muutetaan kokonaislukumuotoon ennen arvojen vertailua ja päivittämistä. Alkureunan määrityksen jälkeen suoritetaan *fim_value_fixer.wgsl*-ohjelma, joka muuttaa kaikki laskenta-alueen eikonal-arvot takaisin liukuluvuiksi.



Kuvio 14: Yksittäisen datapisteen päivitys eikonal-näytteistyspisteisiin. Asiaa on havainnollistettu kaksi ulotteisena. Ohjelmassa suoritetaan sama operaatio, mutta kolmiulotteisena. Punaisella värillä esitetään positiivista euklidista etäisyyttä näytteistyspisteestä pallon pintaan. Sininen väri ilmaisee negatiivista etäisyyttä. Etäisyysarvot päivitetään eikonal-pisteisiin vain, jos löytyy uusi pienempi arvo. Samalla tallennetaan myös pisteen väri arvo.

8.2.2 Päivitysvaihe

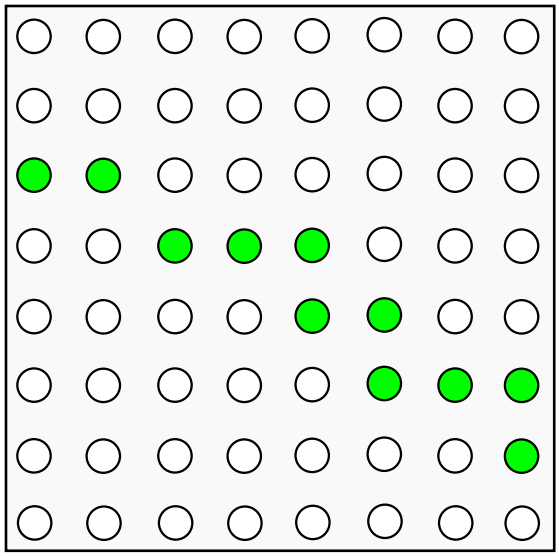
Päivitysvaihe alkaa *initial_active_cells.wgsl*-ohjelman suorituksella 13. Ohjelma käy laskenta-alueen läpi siten, että lähdepisteiden naapuripisteet luokitellaan aktiivipisteiksi. Nämä aktiivipisteet tallennetaan aktiivilistaan arvojen päivittämistä varten 15b 21b. Varsinainen päivitysvaihe suoritetaan *update_phase.wgsl*-ohjelmalla 13, ja se etenee iteraatioissa. Iteraatiossa aktiivilistan pisteiden arvot lasketaan uudestaan naapuripisteiden arvojen perusteella 15c. Jos uusi arvo on pienempi kuin pisteen edellinen arvo, piste pysyy aktiivilistassa ja sen arvo päivitetään. Muussa tapauksessa piste luokitellaan lähdepisteeksi ja se poistetaan aktiivilistasta. Poistetun pisteen ei-tunnetut naapuripisteet lisätään aktiivilistaan 15d. Iteraatiota toistetaan, kunnes mikään arvo ei muutu ja aktiivilista on tyhjä. Päivitysvaiheen tuloksena syntyy approksimaatio eikonal-yhtälön ratkaisulle. Kyseessä ei ole kuitenkaan lopullinen tulos, vaan lopulliset arvot lasketaan korjausvaiheessa.

8.2.3 Korjausvaihe

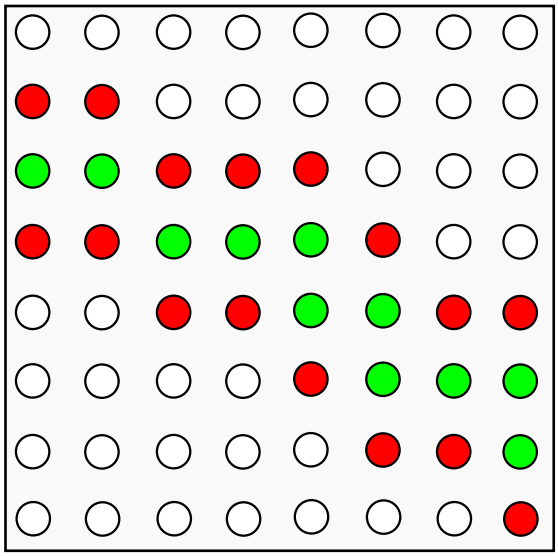
Korjausvaiheessa kaikki laskenta-alueen eikonal-pisteiden arvot lasketaan uudestaan, ja ne pisteet, joiden arvot eivät muutu asetetaan lähdepisteiksi 22a. Muut pisteet lisätään aktiivi-

listaan 22b. Tämä operaatio suoritetaan *pre_remedy_phase.wgsl*-ohjelmalla 13. Tällä tavalla saadaan määriteltyä korjausvaiheen lähtötila.

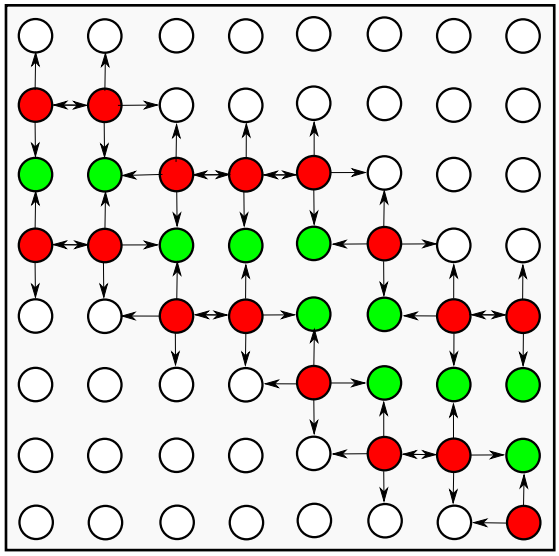
FIM-algoritmin *remedy_phase.wgsl*-ohjelma 13 suorittaa varsinaisen korjausvaiheen. Korjausvaihe etenee päivitysvaiheen tapaan iteraatioina. Päivitys- ja korjausvaiheet eroavat kuitenkin sen suhteen, miten aktiivilistaa rakennetaan. Jos korjausvaiheessa aktiivilistan pisteen arvo ei muutu, piste poistetaan aktiivilistasta eikä sen naapuripisteitä oteta mukaan listaan. Jos arvo pienenee, kaikki sitä ympäröivät lähdepisteet otetaan mukaan aktiivilistaan. Korjausvaihe päättyy, kun aktiivilistassa olevien pisteiden arvot eivät enää muutu. Tällöin viimeiset aktiivilistan pisteet poistetaan, ja uusia pisteitä ei enää tule listaan. Aktiivilista jää tyhjäksi, ja tämän seurauksena iteraatiosilmukan suoritus keskeytetään. Lopputulokseksi saadaan eikonal-yhtälön viskositeettiratkaisu.



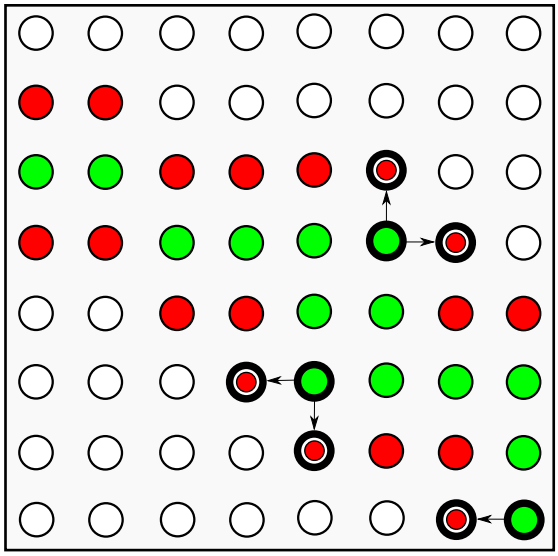
(a) Pistedatasta luodut ensimmäiset lähdepisteet (alkureunan määritys).



(b) Ensimmäisten aktiivipisteiden määrittäminen.



(c) Aktiivilistan pisteiden eikonal-arvojen päivitys. Kaikki aktiivilistan pisteiden arvot lasketaan uudestaan. Nuolet ilmaisevat sen mistä pisteistä arvoja haetaan, kun lasketaan uusi arvo.



(d) Aktiivilistaan jää ne pisteet, joiden arvot pienenivät. Tummennetut vihreät pisteet ovat pisteitä, joiden arvot eivät muuttuneet. Nämä pisteet poistetaan listasta. Nuolien osoittamat ei-lähdepisteet lisätään aktiivilistaan.

Kuvio 15: FIM-algoritmin vaiheita. Kuvassa (a) esitetään aallon alkureunan määrittämisen jälkeistä tilaa. Kuvassa (b) esitetään *initial_active_cells.wgsl*-ohjelman päätöstilaa. Kuvat (c) ja (d) kuvaavat päivitysvaiheen iteraatiota (*update_phase.wgsl*).

8.3 WGSL-tietorakenteet ja FIM-iteraatiot

FIM-algoritmin kannalta tärkeimmät tietorakenteet on kuvattu kuvassa 16. Seuraavaksi käsitellään lyhyesti laskenta-alueen pisteet, aktiivilista ja iteraatio.

8.3.1 Laskenta-alueen pisteet

Eikonal-yhtälön laskenta-alueen pisteet ovat määritelty *FimCell*-tietueina 16. Laskenta-alueen pisteeseen tallennetaan pisteen luokittelutagi, eikonal-arvo ja väri. Pisteen luokitus kertoo sen, onko kyseessä esimerkiksi lähde- tai aktiivipiste. Väri kenttään määritellään väriarvo aallon reunanmuodostuksessa 8.2.1. Väriarvoa ei kuitenkaan käytetä etäisyyskentän rakentamisessa, vaan sitä käytetään ainoastaan ohjelman visualisointivaiheessa. *FimCell*-tietueet muodostavat eikonal-yhtälön laskenta-alueen pisteet, ja ne tallennetaan fim_data-taulukkoon.

8.3.2 Aktiivilista

Ohjelman aktiivilista on *active_list*-niminen taulukko. Taulukkoon ei tallenneta varsinaisia laskenta-alueen pisteitä, vaan sinne tallennetaan $FimCellRef$-tietueita. $FimCellRef$-tietue sisältää tiedon prosessoitavana olevan eikonal-pisteen sijainnista fim_data-taulukossa. Indeksin avulla päästään viittaamaan varsinaiseen eikonal-pisteen dataan. Tietueessa on myös kenttä väliaikaiselle eikonal-arvolle.

Aktiivilistan prosessointi tehdään rinnakkain. Tästä syystä laskentajärjestystä ei voida tietää etukäteen. GPU-laite tekee päätöksen siitä, missä järjestyksessä eri säikeet tekevät laskentaa. Laskentajärjestys voi vaihdella eri ajokerroilla. Tästä syystä varsinaisia fim_data-taulukon eikonal-arvoja ei saa päivittää kesken aktiivilistan arvojen uudelleen laskemista, vaan arvojen täytyy pysyä muuttumattomina laskuoperaatioiden ajan. Uudelleen lasketut eikonal-arvot tallennetaan ensin $FimCellRef$-tietueen väliaikaiseen arvo muuttujaan, ja arvot päivitetään varsinaiseen tietorakenteeseen vasta silloin, kun kaikki aktiivilistan arvot ovat laskettu. Tällä tavalla laskentajärjestys ei vaikuta laskennan lopputulokseen.

Vaikka loogisesti katsottuna FIM-algoritmissa on ainoastaan yksi aktiivilista, käytännön toteutuksessa *active_list*-taulukkoon on varattu tilaa kahdelle aktiivilistalle. Aktiivilista raken-

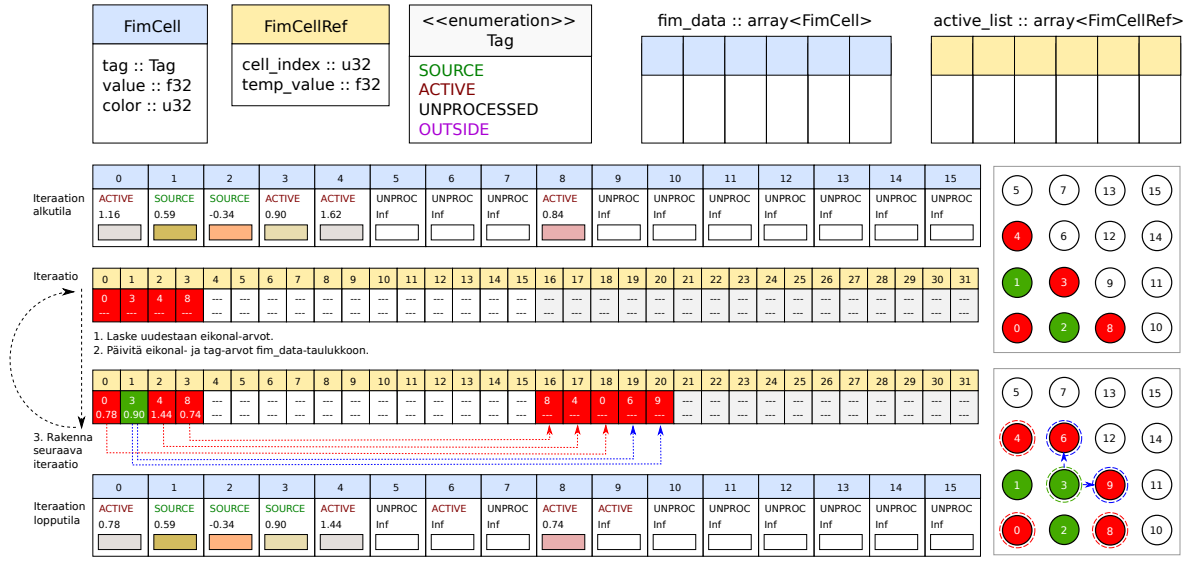
netaan joka iteraatio uudestaan, ja edellisen aktiivilistan tilaa tarvitaan uuden listan rakentamista varten. Tästä syystä aktiivilistaa rakennetaan eri puolelle *active_list*-taulukkoa.

WGSL-ohjelmassa ylläpidetään iteraatiolukumäärää. Päätös siitä, mistä indeksistä aktiivilistan alkioita luetaan ja mistä seuraavan aktiivilistan rakennus aloitetaan, riippuu iteraatioarvon jaollisuudesta. Jos iteraation numero on parillinen, aktiivilistan alkioita luetaan *active_list*-taulukon alusta, ja seuraavan iteraation aktiivilistaa aletaan rakentaa taulukon puolivälistä. Jos iteraation numero on pariton, aktiivilistan lukeminen aloitetaan taulukon puolivälistä ja seuraavan iteraation aktiivilistan rakennus alkaa taulukon alusta. Esimerkki iteraation 16 numero on parillinen.

8.3.3 FIM-iteraatio

FIM-algoritmin päivitys- ja korjausvaihe ovat toistorakenteita, joissa yhtä toistoa kutsutaan iteraatioksi. Yksi iteraatio koostuu kolmesta erillisestä ja rinnakkaisesti suoritettavasta vaiheesta. Ensimmäinen vaihe on aktiivilistan pisteiden eikonal-arvojen uudelleen laskeminen. Seuraava vaihe on eikonal- ja tag-arvojen päivittäminen laskenta-alueeseen. Kolmas vaihe on seuraavan iteraation aktiivilistan rakentaminen. Iteraatioita suoritetaan ikuisessa silmukassa, ja silmukka katkeaa, kun aktiivilistan alkioiden lukumäärä on nolla.

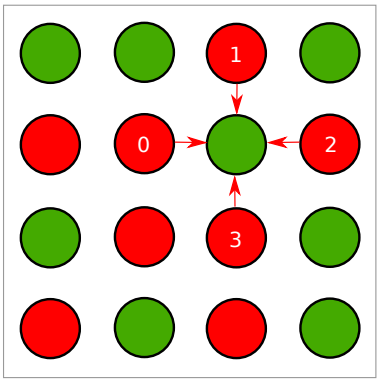
Kuvassa 16 esitetään kaavio yhdestä FIM-algoritmin päivitysvaiheen iteraatiosta. Kuvassa esitetään sekä *fim_data*- että *active_list*-taulukoiden lähtö- ja lopputila. Varsinaiset iteraation operaatiot kuvataan aktiivilistan välisillä muutoksilla ja nuolilla.



Kuvio 16: Esimerkki FIM-algoritmin päivitysvaiheiteraatiosta. Vasemmalla on WGSL-taulukoiden tila. Oikealla on laskenta-alueen visualisointi, jossa numerolla esitetään pisteen muistipaikkaa ja värillä esitetään pisteen tag-arvoa. Vihreä väri ilmaisee lähdepistettä, ja punainen väri aktiivipistettä. Sinisellä katkoviivalla esitetään aktiivilistasta poistetun eikonal-pisteen naapuripistetietojen tallennusta. Punaisella katkoviivalla esitetään aktiivilistassa pysyvien eikonal-pistetietojen kopiointia. Iteraatiot muodostavat toistorakenteen, jossa kuvan alempi aktiivilistan tila muuttuu seuraavan iteraation lähtötilaksi.

8.3.4 Aktiivilistan päivitys atomicExchange-funktiolla

Aktiivilista on tässä sovelluksessa joukko, jossa jokainen *FimCellRef*-tietue on uniikki. Tämä tarkoittaa sitä, että *FimRefCell*-tietue voi viitata täsmälleen yhteen *fim_data*-taulukon alkioon. Tämän toiminnallisuuden saavuttamiseksi WGSL-ohjelmissa käytetään WGSL:n *atomicExchange*-funktiota (W3C 2022f). Funktio toimii siten, että kokonaislukumuistipaikkaan tallennetaan säieturvallisesti uusi arvo, ja funktio antaa paluuarvona muistipaikan edellisen arvon. Funktio vaihtaa nimensä mukaisesti muistipaikan sisällön.



Kuvio 17: Esimerkki FIM-algoritmin korjausvaiheesta, jossa neljä eri aktiivilistan pistettä yrittää muuttaa saman eikonal-pisteen aktiiviseksi. Tilanne on rinnakkaistettu, joten eri aktiivilistan pistettä edustaa eri säie. Operaatiota suorittavat säikeet on esitetty numeroilla, ja nuolilla esitetään minkä eikonal-pisteen tag-arvoa säie pyrkii muuttamaan. Pisteen tiedot voidaan tallentaa aktiivilistaan ainoastaan kerran iteraatiossa, joten vain yksi säie saa suorittaa tämän operaation.

Aktiivilistan päivitysprosessissa kopioidaan ja luodaan uusia uniikkeja $FimCellRef$-tietuita. Jos korjausvaiheessa aktiivilistan pisteen arvo pienenee, ympärillä olevat lähdepisteet lisätään aktiivilistaan. Prosessi on rinnakkaistettu, joten useampi säie voi yrittää muuttaa saman naapuripisteen tilaa aktiiviseksi, ja tallentaa pisteen tiedot seuraavaan iteraation aktiivilistaan 17. Saman tag-arvon tallentaminen useaan kertaan ei ole ohjelman logiikan näkökulmasta vahingollista, mutta eikonal-pisteen tiedot voidaan tallentaa aktiivilistaan vain kerran iteraatiossa. Tästä syystä vain yksi säie saa tallentaa pisteen tiedot aktiivilistaan.

Ongelma voidaan ratkaista *atomicExchange*-funktiolla. Useat säikeet saavat muuttaa pisteen tilaa *atomicExchange*-funktiolla, mutta ainoastaan se säie, joka muuttaa ensimmäisenä muistipaikan $ACTIVE$-arvoksi saa päivittää pisteen aktiivilistaan. Koska *atomicExchange*-funktio palauttaa edellisen tag-arvon, säie pystyy tarkistamaan mikä korvattu arvo on. Jos palautusarvo on $SOURCE$-arvo, säie saa tallentaa pisteen tiedot aktiivilistaan. Seuraavat säikeet saavat *atomicExchange*-funktion paluuarvona sinne jo tallennetun ACTIVE-arvon, jolloin ne eivät saa tallentaa pistettä aktiivilistaan.

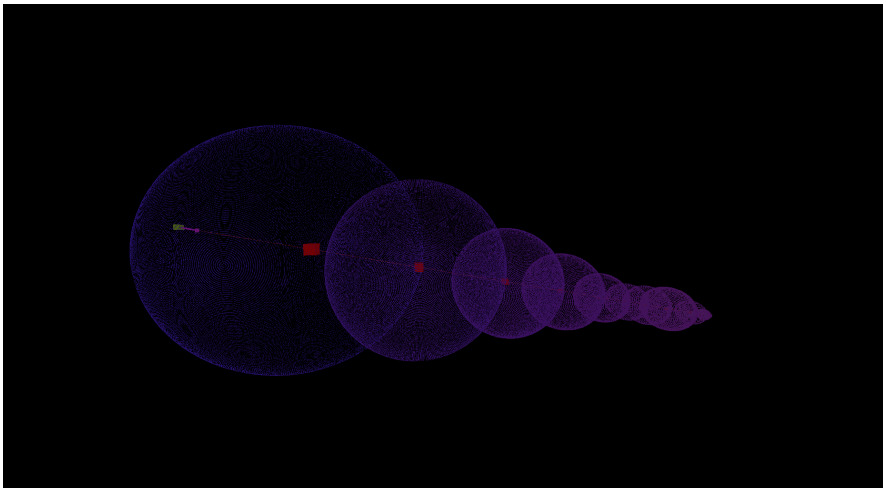
Atomisten funktioiden käyttö on FIM-algoritmin toteutuksen kannalta erityisen tärkeää. Ato-

miset funktiot toimivat hyvin monissa testiajossa, mutta erityisesti *atomicExchange*-funktio osoittautuu ongelmaksi tietyissä ajoympäristöissä.

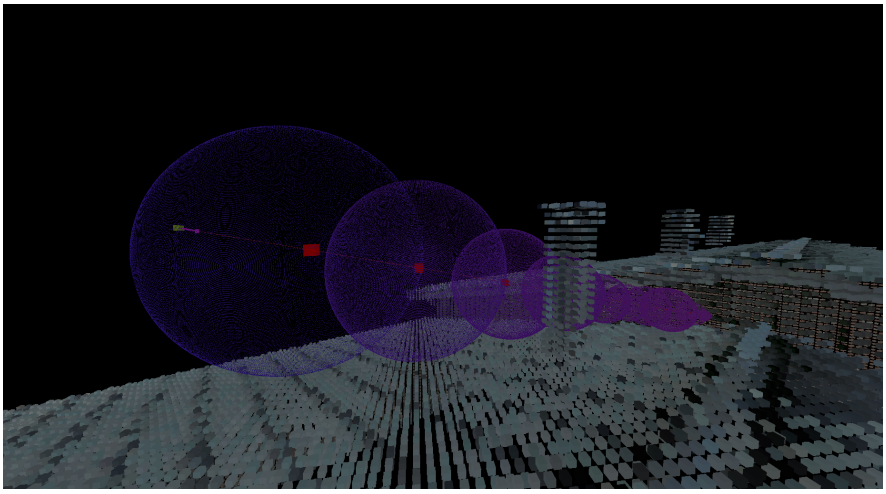
8.4 Etäisyyskentän visualisointi

Ohjelman suorituksessa viimeinen vaihe on datan visualisointi. Kyseessä on yksi WGSL-ohjelma, joka muodostaa kuvan aallon saapumisaikarintamasta. Tarkoituksena on etsiä säteiden törmäyskoordinaatit, joissa etäisyyskentän arvot ovat lähellä nollaa. Kyseessä on pistepilvidatasta muodostettujen pallojen visualisointi. Visualisoitavan pinnan ja säteiden välisiä törmäyskoordinaatteja ei ratkaista analyyttisesti, vaan törmäyskoordinaatit haetaan sphere tracing -tekniikalla 6.3. Eikonal-yhtälöllä luotu etäisyyskenttä toimii algoritmin kiihdytysrakenteena.

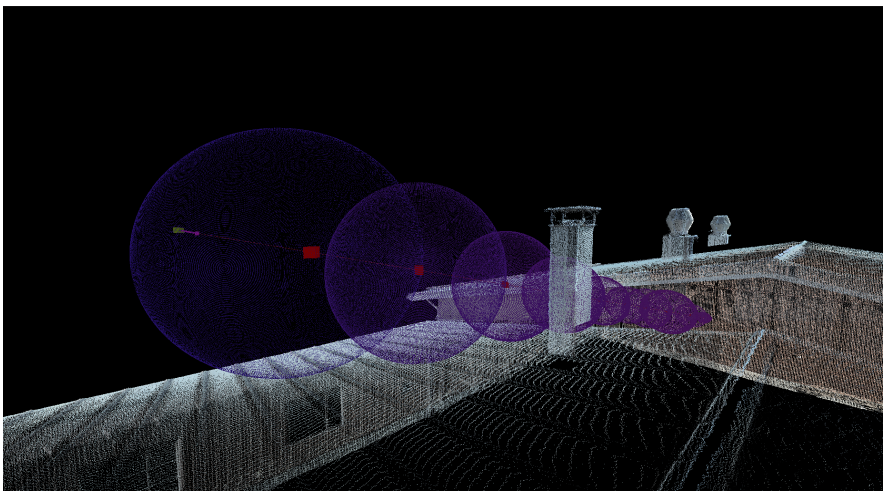
Sphere tracer-algoritmi suoritetaan 64 säikeen ryhmissä. Jokainen säie laskee ruudulle piirrettävälle tekstuurin tekselille värin säteen törmäyskoordinaatin perusteella, tai jättää värin taustaväriksi, jos törmäystä ei tapahdu. Säteiden törmäyskoordinaatit haetaan sphere tracing -algoritmilla etäisyyskentän arvojen perusteella fim_data-taulukosta 16. Sphere tracerin käyttämä etäisyysfunktio toteutetaan etäisyyskentän arvojen interpoloinnin avulla. Pikselin väri lasketaan törmäyskoordinaatin ympäröivien laskenta-alueen pisteiden väriarvojen perusteella. Säteen valaistus lasketaan Phong-varjostuksen avulla. Sekä etäisyyskentän että värien interpoloinnissa käytetään algoritmia, jossa hyödynnetään kentän gradienttia (Csébfalvi 2019).



(a) Yhden säteen sphere tracer näkymä.



(b) Näkymään lisätty eikonal-yhtälön aallon alkureunan näytteistyspisteet.



(c) Näkymään lisätty pistepilviaineisto (Gisgro Oy, n.d.).

Kuvio 18: Ohjelmasta otettuja kuvia säteen etenemisestä.

9 Tulokset ja niiden analyysi

Ohjelmaa testattiin yhteensä kahdeksalla tietokoneella, ja testaustulokset koottiin taulukkoon 4. Tämän tutkimuksen perusteella Vulkan-rajapinta suoriutui testeistä parhaiten. Varsinaista syytä Vulkan-rajapinnan toiminnalliseen menestykseen ei löytynyt. Yksi mahdollinen syy Vulkanin menestykseen on se, että wgpu-kirjaston kehittäjät tuntuvat suosivan Linux-kehitysympäristöä. Tästä syystä Linux ja sen tukema Vulkan-rajapinta ovat mahdollisesti olleet ensisijaisia testauskohteita.

Toiminnallisuuden ongelmat painottuivat erityisesti macOS-koneisiin ja Intelin näytönohjaimiin. Ongelmia esiintyi myös Windows-ajoympäristössä, kun natiivina rajapintana käytettiin D3D12-rajapintaa. Web-selaimessa testattu WebGPU-versio toimi ainoastaan Linux- ja Windows-ympäristössä.

Testidata					
OS	Näytönohjain	Vulkan	D3D12	Metal	Firefox
Win10	NVIDIA GeForce RTX 3060	Pass	Fail		Pass
Win10	NVIDIA GeForce GTX 960	Pass	Fail		Pass
macOS	NVidia GeForce GT 755M			Fail	Fail
macOS	AMD Radeon Pro 575			Fail	Fail
Linux	AMD Radeon 7 Graphics (Renoir)	Pass			Pass
Linux	NVIDIA RTX A2000 Mobile	Pass			Pass
Linux	Intel HD Graphics 620	Fail			Fail
Linux	Intel HD Graphics 4000	Fail			Fail

Taulukko 4: Tutkimuksessa suoritettujen testiajojen tulokset.

9.1 WebAssembly-versio

Firefoxin WebGPU-toteutus etenee hitaammin kuin wgpu-rajapinnan toteutus. Tästä syystä wgpu:n uusimpia versioita ole mahdollistaa kääntää Firefoxissa toimiviksi WebAssembly-ohjelmiksi. WebGPU:n spesifikaatiomääritykset tehdään käyttäen Web IDL-notaatiota (we-

bidl 2022). Web IDL -kielellä määritettyjä rakenteita käytetään pohjana, kun määritellään konkreettisia WebGPU-toteutuksia (Malyshau 2022b). Firefoxin WebGPU-rajapinnan päivittäminen on monimutkainen prosessi, ja se on todennäköisesti yksi syy siihen miksi wgpu ja WebGPU kehittyvät eri tahtiin. Koska wgpu ja Firefoxin WebGPU-rajapinta kehittyivät eri tahtiin, WebAssembly-käännös jätettiin testausvaiheessa viimeisimpään Linuxilla toimivaan versioon.

Firefox-versio toimi niissä Linux-koneissa, joissa oli joko NVidia- tai Radeon-näytönohjain. WebAssembly-käännös toimi moitteettomasti myös Windows-koneilla. Intelin näytönohjainten ja macOS-käyttöjärjestelmien kohdalla web-sivua ei saatu toimimaan. Ohjelman toteutuksen aikana havaittiin myös ongelmia GPU:n puskurien lukemisessa. Tätä tarkoitusta varten kehitetty *MapAsync*-funktio ei toiminut kunnolla WebAssembly-versiossa.

9.2 MapAsync-funktio

MapAsync on WebGPU:n funktio, jolla isäntäohjelma voi varata itselleen luku- tai kirjoitusoikeuden GPU:n hallitsemaan puskuriin. Kun GPU:lla toimivat varjostinohjelmat eivät enää käytä puskuria, puskuri annetaan sitä pyytäneen isäntäohjelman haltuun. Tämän jälkeen isäntäohjelmasta voidaan suorittaa muistioperaatioita isäntäohjelman ja GPU-puskurin välillä. Kun isäntäohjelma ei enää tarvitse puskuria, puskuri vapautetaan GPU:n käyttöön kutsumalla *unMap*-funktiota.

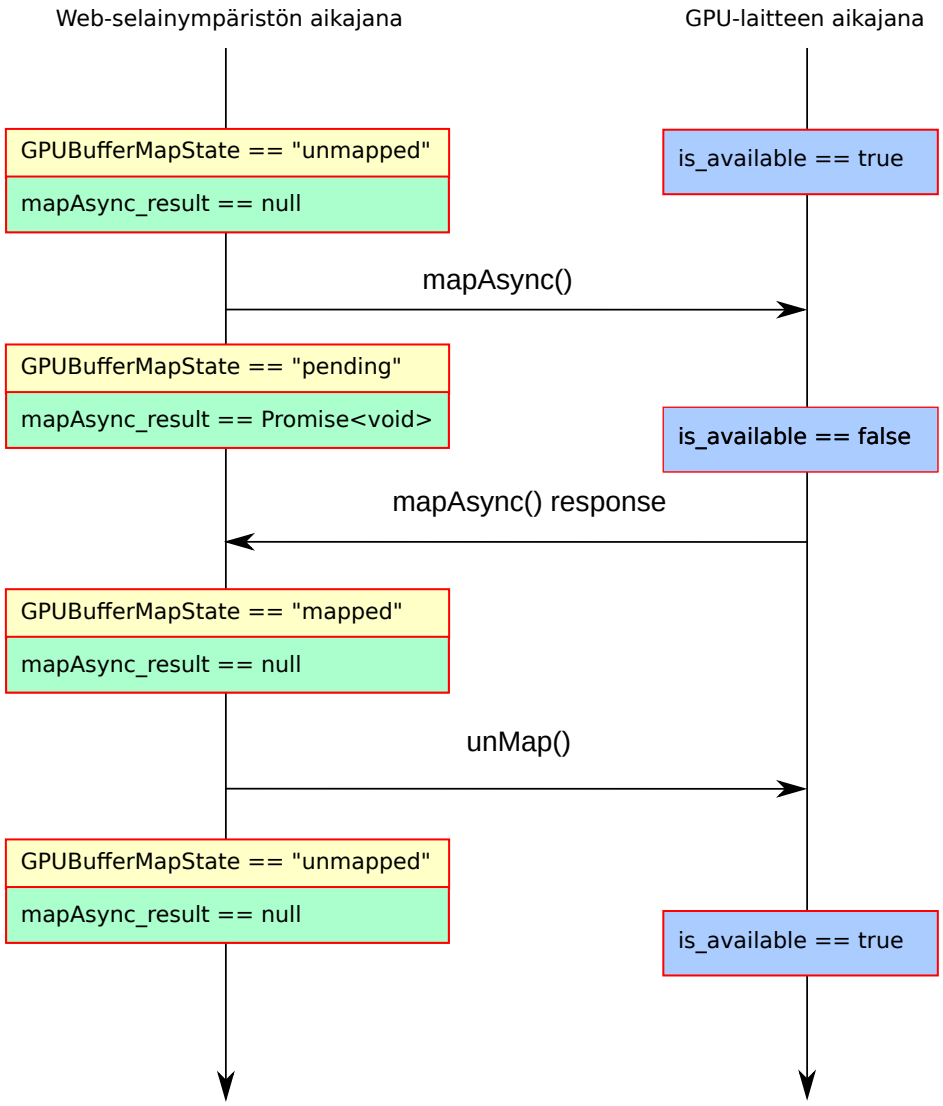
9.2.1 WebAssembly-versio

WebAssembly-käännöksessä Rust koodiin kirjoitettu *MapAsync*-funktio muuttuu WebGPU-funktioksi. Kuvassa 19 on esitetty esimerkki funktion onnistuneesta kutsusta web-ympäristössä. Kuvassa isäntäojelman ja GPU-laitteen tilamuutoksia havainnollistetaan aikajanoilla. Kuvasta puuttuu *Queue*-aikajana. WebGPU:ssa *Queue* toimii käskyjonona, johon lisätään isäntäohjelmasta määritettyjä GPU-operaatioita. GPU suorittaa jonossa olevia käskyjä FIFO (*first in first out*) periaatteella.

WebGPU:n *MapAsync*-kutsu jättää GPU-laitteelle puskurin käyttöpyynnön, ja funktio palauttaa JavaScriptin *Promise*-olion. Jos puskuri on GPU-laitteella vapaana, *Promise*-olio rat-

kaistaan heti, ja GPU-puskuri varataan isäntäohjelmalle luku- tai kirjoitusoperaatioita varten. Muussa tapauksessa *Promise*-olio jää pending-tilaan kunnes WebGPU:n *Queue*-jonoon asetetut GPU-käskyt on suoritettu loppuun. Tällä varmistetaan se, että kaikki jonoon tallennetut käskyt on suoritettu loppuun, ennen kuin puskuri luovutetaan isäntäohjelman käyttöön. JavaScript-ajoympäristö tarkistaa *Promise*-objektin tilan aika ajoin. Jos *Promise*-objekti on pending-tilassa ja puskuri on vapaana, *Promise*-olio ratkaistaan ja isäntäohjelma voi tehdä puskuriin muistioperaatiota. Muussa tapauksessa *Promise*-objekti jätetään selaimen ajoympäristöön seuraavaa tarkistusta varten.

Ohjelman kehityksen alkuvaiheessa huomattiin, että *mapAsync*-funktio ei toiminut odotetusti web-ympäristöissä. WebAssembly-versio kaatui tietyissä tilanteissa virheilmoitukseen, mikä liittyi siihen, että isäntäohjelmasta yritettiin lukea GPU-puskurin dataa ilman puskurin lukulupaa. Ongelmaksi muodostui se, että ohjelman suoritus eteni puskurinluku-funktioon ennen kuin *mapAsync*-funktion palauttama *Promise*-objekti oli ratkaistu onnistuneesti. Tämä ongelma ilmeni erityisesti silloin, kun *mapAsync*-funktiota yritettiin kutsua winit-tapahtumasilmukan sisältä. *MapAsync*-funktio loi *Promise*-olion, mutta *Promise* jäi välittömästi pending tilaan, ja sen suoritus jätettiin ajoympäristön varaan. Ohjelman suoritus eteneni suoraan puskurinluku funktioon, josta aiheutui virheilmoitus. *MapAsync*-funktiota pystyi kuitenkin kutsumaan ennen winit-silmukkaan siirtymistä, mutta ei enää sen jälkeen. Tämä tarkoitti sitä, että GPU-puskureita pystyi lukemaan ainoastaan ohjelman alustusvaiheessa, mutta ei enää sen jälkeen.



Kuvio 19: Onnistuneen *MapAsync*-funktiokutsun kuvaus web-ympäristössä. Vasemmalla esitetään isäntäohjelmassa olevan puskurin tilaa. Oikealla esitetään GPU-laitteen puskurin tilaa. *GPUBufferMapState*-muuttuja kuvaa puskurin varaustilaa isäntäohjelmassa. Kuvassa esitetty *mapAsync_result* kuvaa *mapAsync*-funktiokutsun palautusarvon tilaa. Kuvassa *is_available* kuvaa puskurin varausmahdollisuutta GPU-laitteella. *MapAsync*-funktiota voidaan kutsua ainoastaan silloin kun *mapAsync_result* arvo on null. Muussa tapauksessa funktiota on jo kutsuttu ja funktiokutsusta aiheutuu virhe. Isäntäohjelma voi tehdä muistioperaatiota ainoastaan silloin kun *GPUBufferMapState* on mapped-tilassa. Kun isäntäohjelman muistioperaatiot on suoritettu, isäntäohjelma kutsuu *unMap*-funktiota, joka vapauttaa GPU-puskurin muita operaatioita varten.

9.2.2 Natiivit versiot

MapAsync-funktio toimi natiiviksi käännettyjen ohjelmien kohdalla hyvin, koska natiivit ohjelmat mahdollistivat ohjelman pysäytyksen ennen puskurin lukuoperaatiota. *MapAsync*-funktion jälkeen kutsuttiin wgpu-rajapinnan *Device*-objektin *poll*-funktiota, joka pysäytti ohjelman suorituksen kunnes *Queue* oli tyhjentynyt suoritettavista käskyistä. Tämän jälkeen puskurin lukeminen onnistui ilman ongelmia. Poikkeuksen tähän teki Intelin näytönojaimet. Ohjelma kaatui *mapAsync*-funktion kutsuun, kun ohjelmaa ajettiin Intelin näytönojaimilla. Tähän ei löytynyt selittävää syytä.

9.2.3 Vaikutus algoritmien toteutukseen

Ohjelman WGLS-ohjelmat toteutettiin siten, että niiden toteutukset eivät riippuneet *mapAsync*-funktion käytöstä. Tämä tarkoitti sitä, että ohjelmalogiikkaa siirrettiin pois isäntäohjelmasta WGSL-ohjelmiin. Ohjelman kehitystä varten kehitetyt debuggaus-toiminnot piti kuitenkin jättää pois lopullisesta WebAssembly-versiosta, koska niiden toiminta perustui *mapAsync*-funktioon.

WebGPU on tätä tutkimusta tehdessä edelleenkin kehitystyön alla, ja sen *mapAsync*-funktion toiminnallisuudesta käydään keskustelua. Osa ihmisistä haluaisivat tehdä funktiosta synkronisen, jolloin se pysäyttäisi web-ympäristön suorituksen funktiokutsun ajaksi. Perusteena tähän on se, että pysäytys on niin lyhyt, että se ei haittaa selainympäristön toimintaa. Toiset puolestaan haluavat pitää funktion asynkronisena, koska Javascript-ympäristö on luonteeltaan asynkroninen. Tällä hetkellä WebGPU:n *mapAsync*-toteutus on asynkroninen.

Wgpu:n uusimmissa versioissa *mapAsync*-funktio on muutettu asynkronisesta muodosta takaisinkutsuperusteiseksi funktioksi. Tämä ei kuitenkaan toiminut testatuilla Intelin näytönohjaimilla, eikä sitä päässyt testaamaan WebAssembly-käännöksenä. Syynä tähän oli se, että Firefoxin WebGPU-rajapinta kehittyi hitaammin kuin Rust-kirjaston wgpu-toteutus.

9.3 Metal-rajapinnan ongelmat

Tutkimuksessa oli käytössä kaksi macOS-tietokonetta. Ohjelma ei toiminut kummassakaan tietokoneessa. Natiivikäännöksessä ongelmaksi muodostui naga-kirjasto. Naga-kirjaston tehtävä on muuttaa ajon aikana WGSL-lähdekoodi natiivin rajapinnan tukemaan varjostinohjelma muotoon. MacOS-laitteiden kohdalla natiivi grafiikka- ja laskentarajapinta on Metal ja sen tukema varjostinohjelmointikieli on MSL.

Ongelmaksi muodostui FIM-algoritmin käyttämä *atomicExchange*-funktio 8.3.4. Ohjelma kaatui kummankin macOS-laitteen kohdalla MSL-varjostusohjelman validointi virheilmoitukseen. Koodi, josta tulee virhe-ilmoitus on naga-kirjaston generoimaa ohjelmakoodia. Naga generoi WGSL:n *atomicExchange*- funktion parametrit väärin MSL:n vastaavaan *atomic_store_explicit*-funktioon. Tämä on dokumentoitu virhe naga-kirjastossa, jota ei ole vielä tätä tutkielmaa tehtäesssä korjattu. Ainakin tämä ongelma esti ohjelman toiminnan kummallakin macOS-koneella.

Myös Webassembly-versio kaatui kummallakin macOS-koneella. Syynä tähän ei ollut edellä mainittu nagan muunnosongelma, vaan ohjelma kaatui jo ennen varjostinohjelmien kääntämisvaihetta. Virhe-ilmoitus liittyi siihen, että HTML-canvas objektilta ei saatu piirtämistä varten tarvittavaa piirtokontekstia. Tarkkaa syytä virheilmoitukseen ei saatu selvitettyä.

9.4 D3D12-rajapinnan ongelmat

Windows-koneet suoriutuivat Vulkan- ja WebAssembly-testeistä hyvin, mutta D3D12-rajapinnan testit epäonnistuivat. D3D12-rajapinnan varjostinohjelmointikieli on HLSL. Virheilmoitukset liittyivät nagan generoimaan HLSL-koodiin. Virheilmoitukset liittyvät erityisesti varjostinohjelmien huonosti määritettyihin synkronointikohtiin. Tutkimuksessa jäi epäselväksi se, johtuiko virhe naga-kirjastosta vai WGSL-spesifikaation keskeneräisyydestä. Joka tapauksessa D3D12-testit kaatuvat kummassakin Windows koneessa samaan virheilmoitukseen.

10 Johtopäätökset

Wgpu ja WebGPU olivat tätä tutkimusta tehdessä uusia ja keskeneräisiä teknologioita. Spesifikaatiot eivät olleet vielä ottaneet lopullista muotoaan, ja kirjastoja päivittiin ripeään tahtiin. Kirjastojen virheiden raportointiosuuksissa oli tätä tutkimusta tehdessä vielä paljon korjaamattomia virheitä. Nämä seikat olivat tiedossa tutkimusta tehtäessä, ja ne vaikeuttivat omalta osaltaan ohjelman toteuttamista.

Tutkimuksen testitulokset eivät olleet kovinkaan yllättäviä. Tutkimus osoitti sen, että teknologia oli edelleenkin keskeneräistä. Testit osoittivat toisaalta sen, että Firefox-selaimessa voitiin suorittaa rajoitetusti GPU-laskentaa. Tämä oli jo itsessään iso edistys verrattuna WebGL:n tarjoamiin laskentamahdollisuuksiin. Tutkimuksessa kävi myös ilmi se, että WebAssembly oli varteenotettava vaihtoehto puhtaasti JavaScriptilla tehtyihin ohjelmiin. Tutkielman ohjelmaan ei jouduttu kirjoittamaan riviäkään JavaScriptia. Kaikki WebAssembly-versioon vaadittava JavaScript-koodi generoitiin työkaluilla ja kirjastoilla.

Tutkimuksen aikana kehitettiin wgpu-sovelluksia, jotka jätettiin pois varsinaisesta tutkimuksesta. Tutkielmasta jäi pois muun muassa GPU:lla toimiva marching cubes ohjelma 25e, tilantäyttävien käyrien visualisointi ohjelmat 25a 25b 25c 25d ja rinnakkaistettu FMM 25f. Vaikka juuri tämän tutkimuksen ohjelma ei kääntynytkään kaikilla koneilla ja rajapinnoilla natiiveiksi ohjelmiksi, tutkimuksen ulkopuolelle jääneet wgpu-sovellukset toimivat kuitenkin Vulkan-, D3D12- että Metal-rajapinnoilla.

Tutkielman teknologiat eivät tämän tutkimuksen mukaan sovellu tällaisenaan laajamittaiseen geneeriseen GPU-ohjelmointiin, mutta perusteet sille näyttäisi olevan olemassa. Vaikka tutkimuksesta saatiin lisää informaatiota valituista teknologioista, jäljelle jäi paljon kysymyksiä vaille vastausta. Erityisesti GPU:n taulukoiden lukeminen isäntäohjelmaan on erittäin kriittinen ja tärkeä ominaisuus, jotta GPU-laskentaa voisi hyödyntää paremmin. Vaikka ongelma todettiin, varsinaisia ratkaisuehdotuksia ei tässä tutkielmassa saatu selvitettyä. Tästä aiheesta olisi hyvä olla lisää tutkimusta, jotta tämä rajoite saataisiin mahdollisesti korjattua. Tässä tutkimuksessa ei myöskään tutkittu lainkaan Googlen dawn-kirjaston toiminnallisuutta ja Chrome-selaimen WebGPU-kehitysastetta.

Lähteet

Bálint, Csaba, ja Gábor Valasek. 2018. “Accelerating Sphere Tracing”. Teoksessa *Eurographics 2018 – Short Papers,* toimittanut Olga Diamanti ja Amir Vaxman. doi:10.2312/egs.20181037.

Chacon, Adam, ja Alexander Vladimirsky. 2015. “A Parallel Two-Scale Method for Eikonal Equations”. *SIAM Journal on Scientific Computing* 37:A156–A180. doi:10.1137/120881956.

Csébfalvi, Balázs. 2019. “Beyond Trilinear Interpolation: Higher Quality for Free”. *ACM Transitions on Graphics* 38 (4): 1–8. doi:10.1145/3306346.3323032.

Dijkstra, Edsger Wybe. 1959. “A note on two problems in connexion with graphs”. *Numerische Mathematik* 1 (1): 269–271. doi:10.1007/BF01386390.

DIKU. 2022. Viitattu 13. elokuuta. https://futhark-lang.org/.

Flynn, Michael J. 1972. “Some Computer Organizations and Their Effectiveness”. *IEEE Transactions on Computers* C–21 (9): 948–960. doi:10.1109/TC.1972.5009071.

Fu, Zhisong, Robert M. Kirby ja Ross T. Whitaker. 2013. “A Fast Iterative Method for Solving the Eikonal Equation on Tetrahedral Domains”. *SIAM Journal on Scientific Computing* 35 (5): C473–C494. doi:10.1137/120881956.

Ganellari, Daniel, ja Gundolf Haase. 2016. “Fast many-core solvers for the Eikonal equations in cardiovascular simulations”. Teoksessa *2016 International Conference on High Performance Computing & Simulation (HPCS),* 278–285. doi:10.1109/HPCSim.2016.7568347.

Gisgro Oy. 2022. Viitattu 13. elokuuta. https://gisgro.com/.

———. n.d. “Laserskannattu pistepilvidata”. Lisensoitu aineisto.

Google. 2022a. Viitattu 13. elokuuta. https://chromium.googlesource.com/angle/angle/.

———. 2022b. Viitattu 13. elokuuta. https://dawn.googlesource.com/dawn.

Google. 2022c. Viitattu 13. elokuuta. https://dawn.googlesource.com/tint.

Group, The Khronos. 2022a. Viitattu 13. elokuuta. https://www.khronos.org/opencl/.

———. 2022b. Viitattu 13. elokuuta. https://www.opengl.org/.

———. 2022c. Viitattu 13. elokuuta. https://www.khronos.org/opengl/wiki/History_of_OpenGL.

———. 2022d. Viitattu 13. elokuuta. https://www.vulkan.org/.

———. 2022e. Viitattu 13. elokuuta. https://www.khronos.org.

———. 2022f. Viitattu 13. elokuuta. https://www.khronos.org/spir/.

———. 2022g. Viitattu 13. elokuuta. https://github.com/KhronosGroup/MoltenVK.

———. 2022h. Viitattu 13. elokuuta. https://registry.khronos.org/OpenGL-Refpages/.

———. 2022i. Viitattu 13. elokuuta. https://www.khronos.org/webgl/.

———. 2022j. Viitattu 13. elokuuta. https://registry.khronos.org/webgl/specs/latest/2.0-compute/.

———. 2022k. Viitattu 13. elokuuta. https://registry.khronos.org/webcl/specs/latest/1.0/.

Groves, Josh. 2022. Viitattu 13. elokuuta. https://github.com/grovesNL/glow.

Hart, John C. 1996. “Sphere tracing: a geometric method for the antialiased ray tracing of implicit surfaces”. *The Visual Computer* 12 (10): 527–545. doi:10.1007/s003710050084.

Hevner, Alan R., Salvatore T. March, Jinsoo Park ja Sudha Ram. 2004. “Design Science in Information Systems Research”. *MIS Quarterly* 28 (1): 75–105.

Hidaka, Masatoshi, Yuichiro Kikura, Yoshitaka Ushiku ja Tatsuya Harada. 2017. “WebDNN: Fastest DNN Execution Framework on Web Browser”. Teoksessa *Proceedings of the 25th ACM International Conference on Multimedia,* 1213–1216. MM ’17. doi:10.1145/3123266.3129394.

Huang, Yuhao. 2021. “Improved Fast Iterative Algorithm for Eikonal Equation for GPU Computing”. *arXiv preprint arXiv:2106.15869.* doi:10.48550/ARXIV.2106.15869.

Inc, Apple. 2022. Viitattu 13. elokuuta. https://developer.apple.com/metal/.

International, Ecma. 2022a. Viitattu 13. elokuuta. https://262.ecma-international.org/.

———. 2022b. Viitattu 13. elokuuta. https://tc39.es/ecma262/#sec-promise-objects.

Jeong, Won-Ki, ja Ross T. Whitaker. 2008. “A Fast Iterative Method for Eikonal Equations”. *SIAM Journal on Scientific Computing* 30 (5): 2512–2534. doi:10.1137/060670298.

Jones, Mark W, J Andreas Baerentzen ja Milos Sramek. 2006. “3D distance fields: a survey of techniques and applications”. *IEEE Transactions on Visualization and Computer Graphics* 12 (4): 581–599. doi:10.1109/TVCG.2006.56.

Keinert, Benjamin, Henry Schäfer, Johann Korndörfer, Urs Ganse ja Marc Stamminger. 2014. “Enhanced Sphere Tracing”. Teoksessa *Smart Tools and Apps for Graphics – Eurographics Italian Chapter Conference,* toimittanut Andrea Giachetti. The Eurographics Association. doi:10.2312/stag.20141233.

Mages, Rust Graphics. 2022a. Viitattu 13. elokuuta. https://github.com/gfx-rs/wgpu.

———. 2022b. Viitattu 13. elokuuta. https://github.com/gfx-rs/naga.

———. 2022c. Viitattu 13. elokuuta. https://github.com/gfx-rs/wgpu/blob/master/etc/big-picture.png.

Malyshau, Dzmitry. 2022a. Viitattu 13. elokuuta. https://mozillagfx.wordpress.com/2021/03/10/webgpu-progress/.

Malyshau, Dzmitry. 2022b. Viitattu 13. elokuuta. http://kvark.github.io/web/gpu/gecko/2019/12/10/gecko-webgpu.html.

Michael G. Crandall, Pierre-Louis Lions. 1983. "Viscosity solutions of Hamilton–Jacobi equations". *Transactions of the American mathematical society* 277 (1): 1–42.

Microsoft. 2022a. Viitattu 13. elokuuta. https://microsoft.github.io/DirectX-Specs/.

———. 2022b. Viitattu 13. elokuuta. https://microsoft.github.io/DirectX-Specs/d3d/archive/D3D11_3_FunctionalSpec.htm.

Osher, Stanley, ja Ronald Fedkiw. 2003. *The Level Set Methods and Dynamic Implicit Surfaces*. Nide 153. Applied Mathematical Sciences. Springer New York. ISBN: 0-387-95482-1.

Quilez, Inigo. 2022. Viitattu 13. elokuuta. https://iquilezles.org/articles/distfunctions/.

Rouy, Elisabeth, ja Agnes Tourin. 1992. "A viscosity solutions approach to shape-from-shading". *SIAM Journal on Numerical Analysis* 29 (3): 867–884.

Rust-lang. 2022. Viitattu 13. elokuuta. https://doc.rust-lang.org/std/future/trait.Future.html.

rust-windowing. 2022. Viitattu 13. elokuuta. https://github.com/rust-windowing/winit.

rustwasm. 2022. Viitattu 13. elokuuta. https://github.com/rustwasm/wasm-bindgen.

Sethian, James A. 1996. "A fast marching level set method for monotonically advancing fronts". *Proceedings of the National Academy of Sciences* 93 (4): 1591–1595. doi:10.1073/pnas.93.4.1591.

Smilkov, Daniel, Nikhil Thorat, Yannick Assogba, Charles Nicholson, Nick Kreeger, Ping Yu, Shanqing Cai, Eric Nielsen, David Soegel, Stan Bileschi ym. 2019. “TensorFlow.js: Machine Learning For The Web and Beyond”. Teoksessa *Proceedings of Machine Learning and Systems,* toimittanut A. Talwalkar, V. Smith ja M. Zaharia, 1:309–321. doi:10.48550/arXiv.1901.05350.

Team, Rust. 2022. Viitattu 13. elokuuta. https://www.rust-lang.org/.

Tensorflow. 2022. Viitattu 13. elokuuta. https://github.com/tensorflow/tfjs/.

Tor Gillberg, Are Magnus Bruaset, Øyvind Hjelle ja Mohammed Sourouri. 2014. “Parallel solutions of static Hamilton–Jacobi equations for simulations of geological folds”. *Journal of Mathematics in Industry* 4:1–31. doi:10.1186/2190-5983-4-10.

Tsai, Yen-Hsi Richard, Li-Tien Cheng, Stanley Osher ja Hong-Kai Zhao. 2003. “Fast Sweeping Algorithms for a Class of Hamilton–Jacobi Equations”. *SIAM Journal on Numerical Analysis* 41 (2): 673–694. doi:10.1137/S0036142901396533.

Tsitsiklis, John N. 1994. “Efficient algorithms for globally optimal trajectories”. Teoksessa *Proceedings of 1994 33rd IEEE Conference on Decision and Control,* 2:1368–1373. doi:10.1109/CDC.1994.411258.

W3C. 2022a. Viitattu 13. elokuuta. https://www.w3.org/TR/webgpu/.

———. 2022b. Viitattu 13. elokuuta. https://www.w3.org/TR/WGSL/.

———. 2022c. Viitattu 13. elokuuta. https://www.w3.org/community/gpu/.

———. 2022d. Viitattu 13. elokuuta. https://www.w3.org/.

———. 2022e. Viitattu 13. elokuuta. https://github.com/gpuweb/gpuweb/wiki/Implementation-Status.

———. 2022f. Viitattu 13. elokuuta. https://www.w3.org/TR/WGSL/#atomic-rmw.

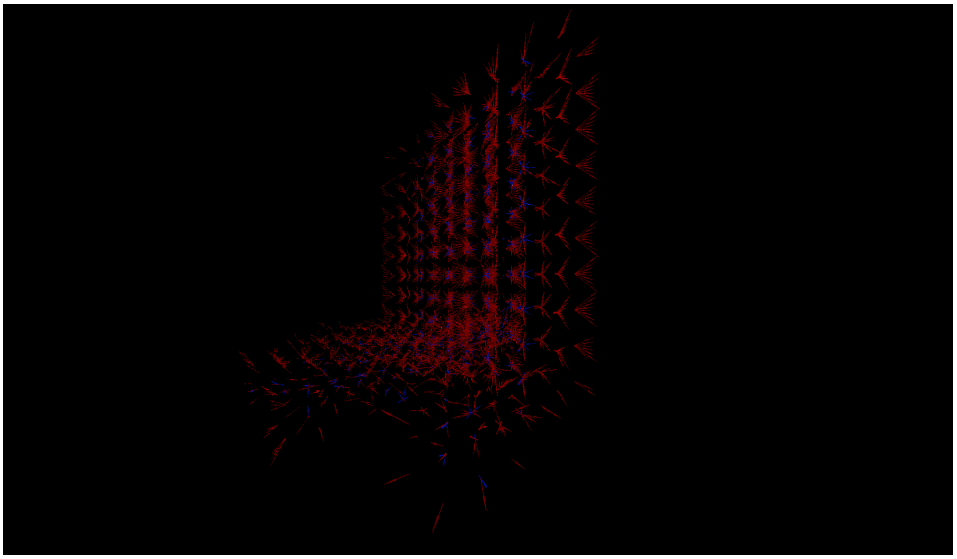
Watt, Conrad, Andreas Rossberg ja Jean Pichon-Pharabod. 2019. “Weakening WebAssembly”. *Proc. ACM Program. Lang.* 3 (OOPSLA). doi:10.1145/3360559.

webidl. 2022. Viitattu 13. elokuuta. https://webidl.spec.whatwg.org/.

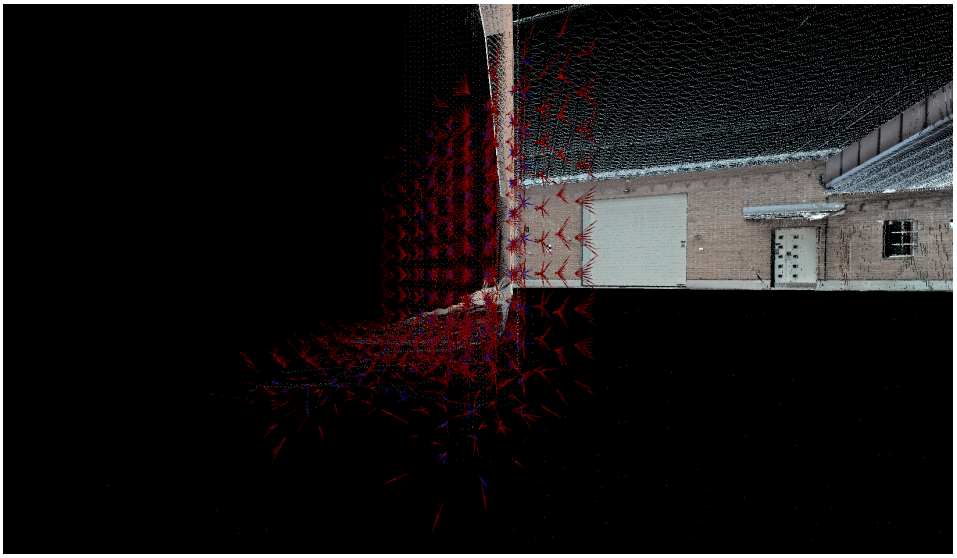
Yang, Jianming. 2019. “An Easily Implemented, Block-Based Fast Marching Method with Superior Sequential and Parallel Performance”. *SIAM Journal on Scientific Computing* 41 (5): C446–C478. doi:10.1137/18M1213464.

Liitteet

A Kuvia eikonal-yhtälön aallon reunan rakennusvaiheesta.



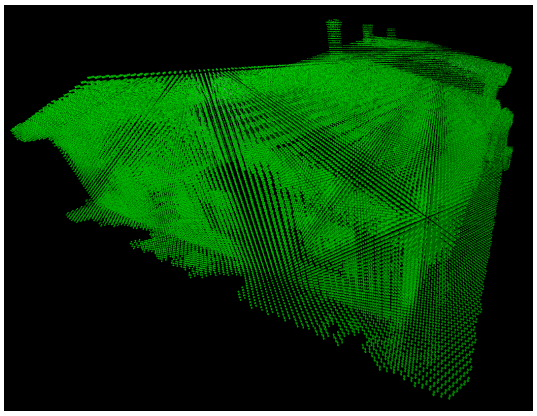
(a) Laskenta-alueen pisteiden etäisyys- ja väriarvoja päivitetään rinnakkain 1024 säikeen ryhmissä. Kuvassa visualisoitu yhden säieryhmän säikeet.



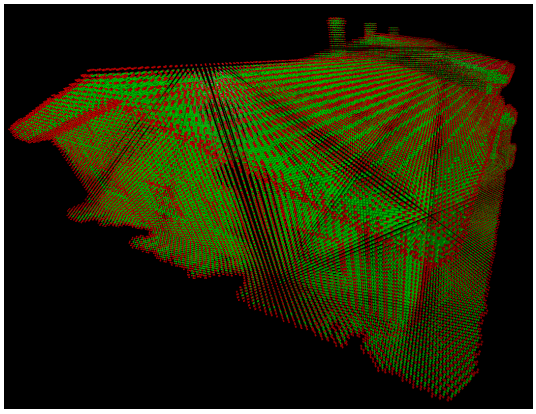
(b) Kuvaan lisätty laskennassa käytetty pistepilviaineisto (Gisgro Oy, n.d.).

Kuvio 20: Ohjelmasta otettuja kuvia alkureunan määrittämisvaiheesta (*pc_to_interface.wgsl*). Punainen nuoli kuvaa positiivista etäisyyttä, ja sininen nuoli kuvaa negatiivista etäisyyttä.

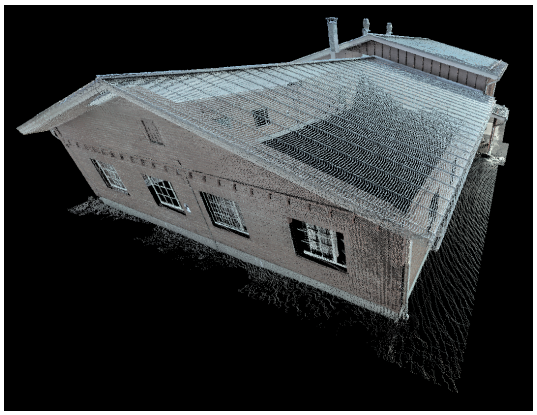
B Kuvia eikonal-yhtälön aallon alkureunasta.



(a) Pistedatasta luodut ensimmäiset lähdepisteet.



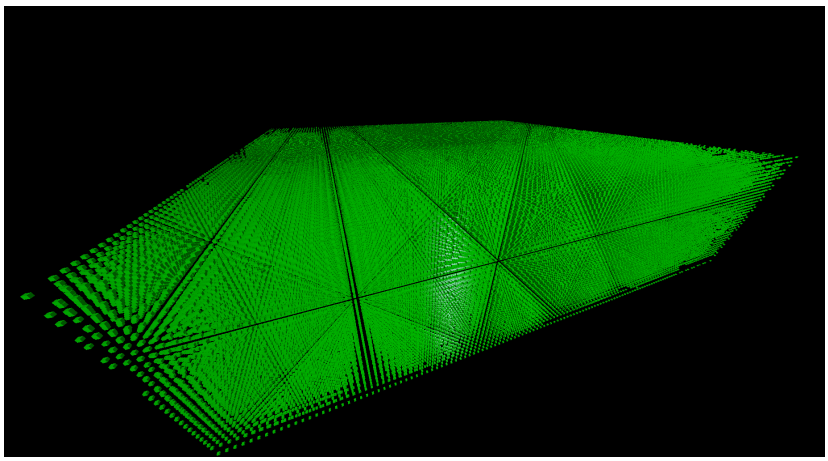
(b) Ensimmäisten aktiivipisteiden määrittäminen.



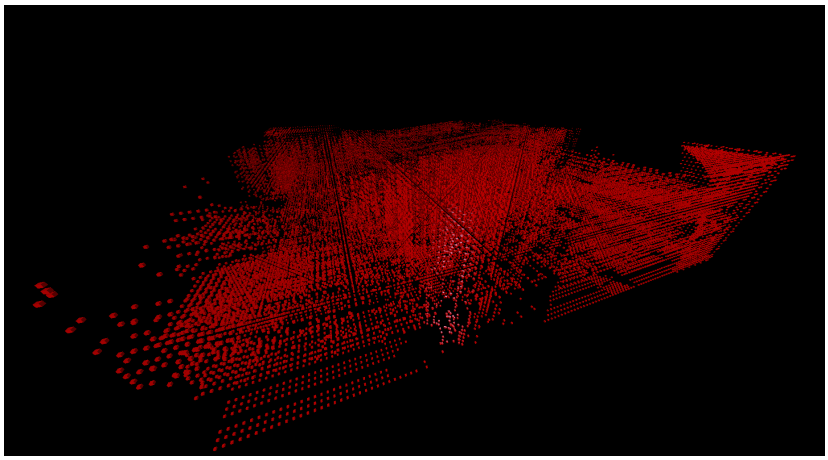
(c) Laskennassa käytetty pistepilviaineisto (Gisgro Oy, n.d.).

Kuvio 21: Ohjelmasta otettuja kuvia FIM-algoritmin alkureunan määrittämisvaiheen jälkeisestä tilasta.

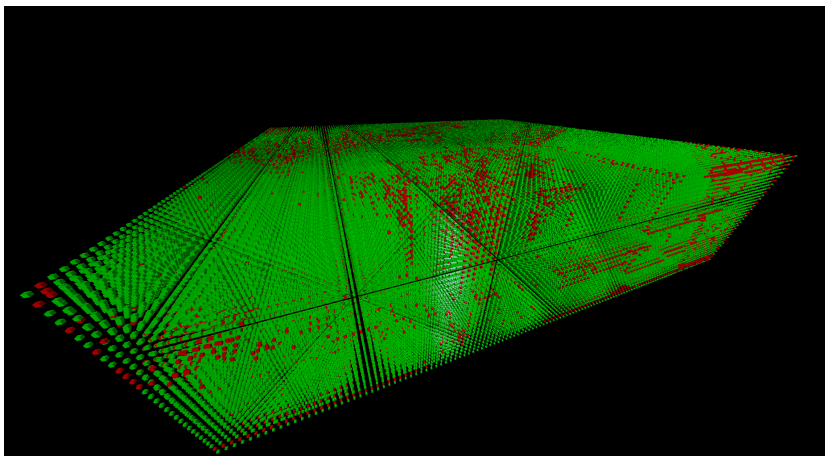
C Kuvia FIM-algoritmin korjausvaiheen alkutilasta.



(a) Lähdepisteet.



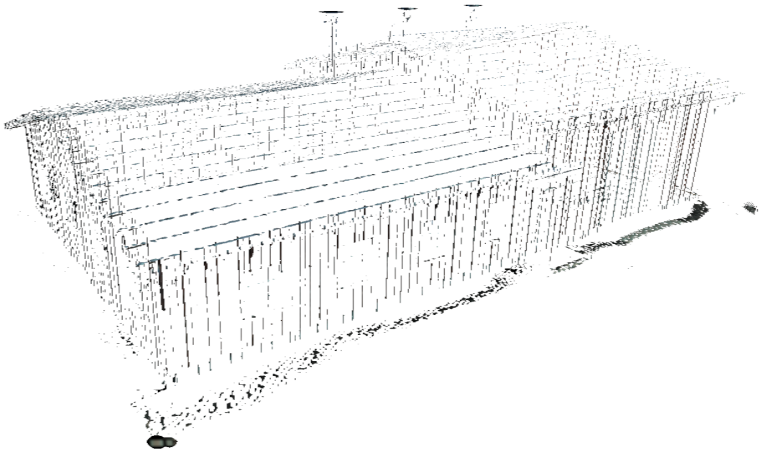
(b) Aktiivipisteet.



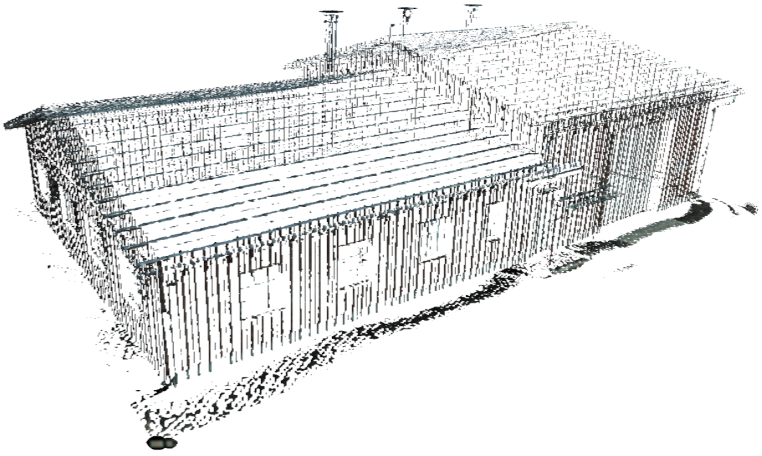
(c) Lähde- ja aktiivipisteet.

Kuvio 22: Lähde- ja aktiivipisteet *pre_remedy_phase.wgsl*-ohjelman suorituksen jälkeen.

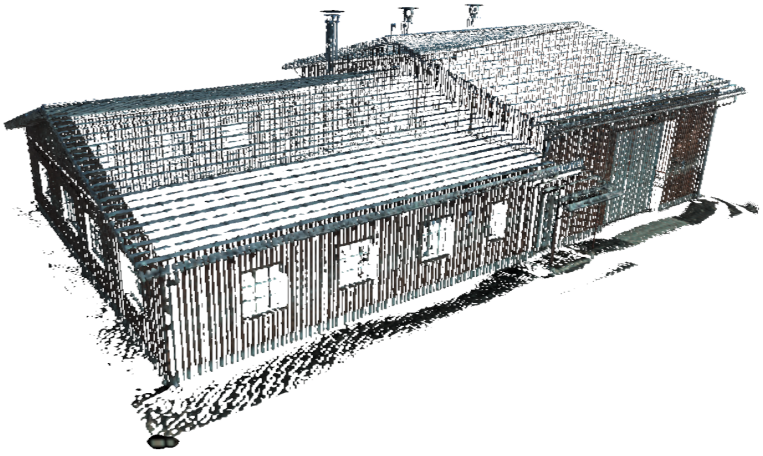
D Visualisointi kun pistedata tulkitaan erikokoisina palloina.



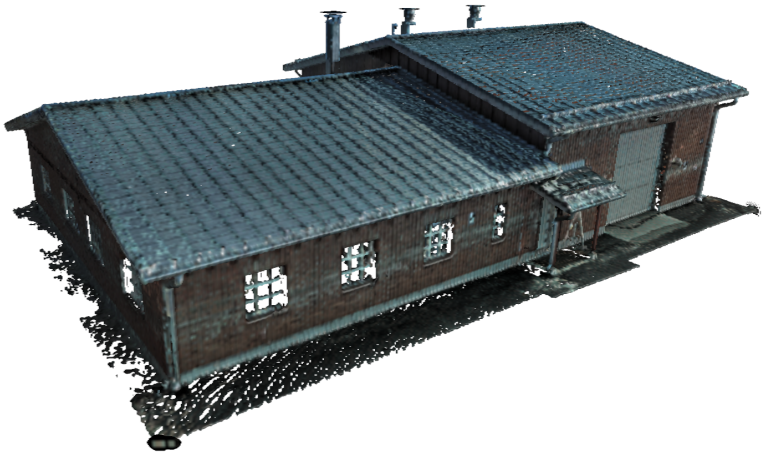
(a) Pistedata tulkitaan 0.01-säteisinä palloina.



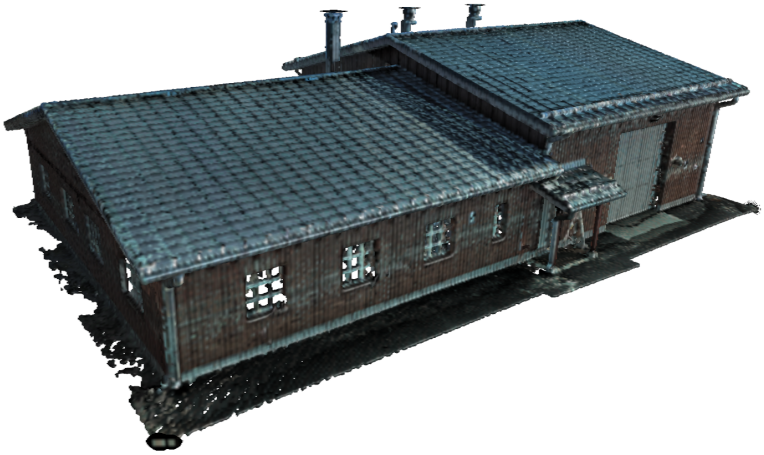
(b) Pistedata tulkitaan 0.1-säteisinä palloina.



(c) Pistedata tulkitaan 0.2-säteisinä palloina.



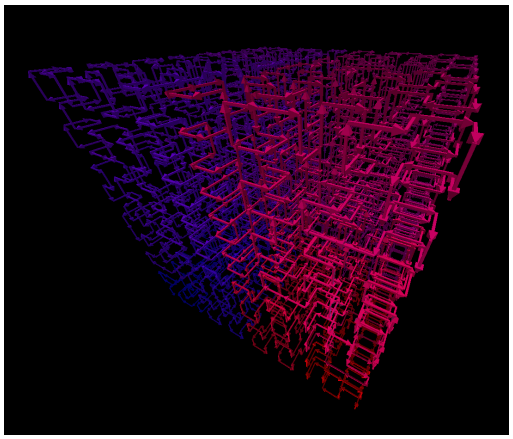
(a) Pistedata tulkitaan 0.5-säteisinä palloina.



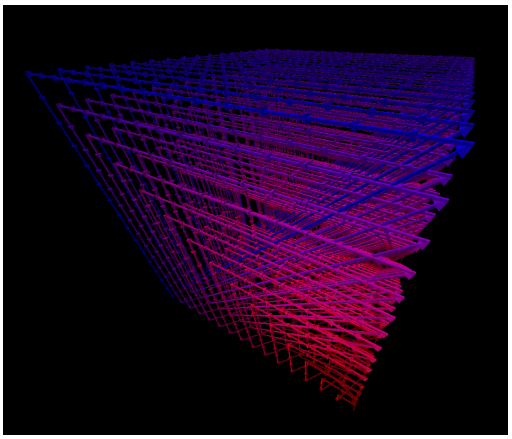
(b) Pistedata tulkitaan 0.75-säteisinä palloina.

(c) Pistedata tulkitaan 1.0-säteisinä palloina.

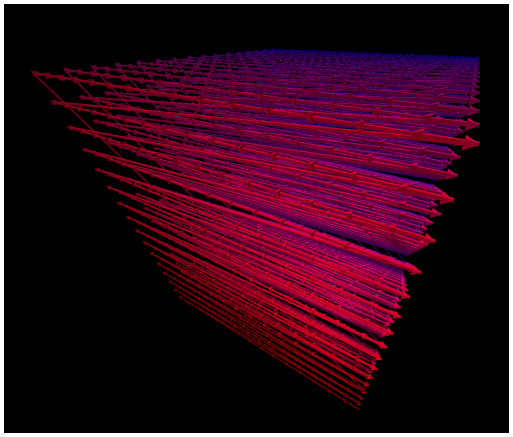
E Kuvia tutkielman ulkopuolelle jääneistä wgpu-ohjelmista.



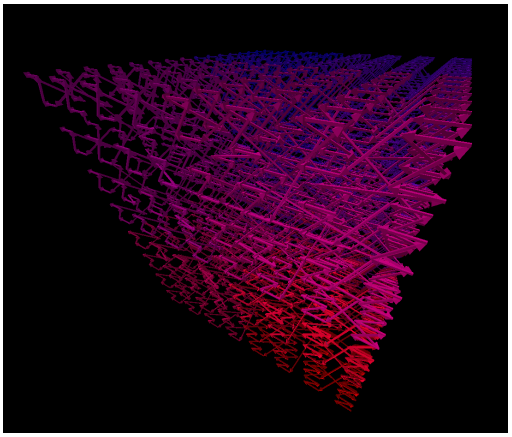
(a) Hilbert.



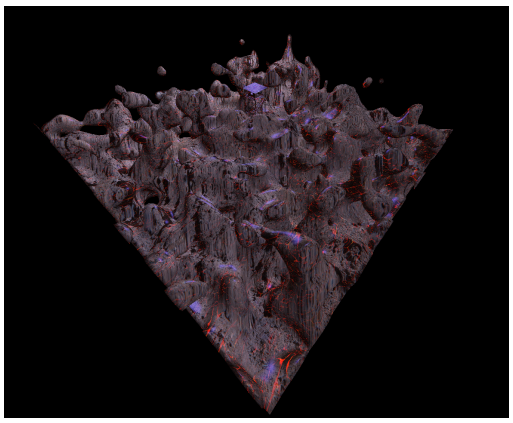
(b) Rosenberg-Strong.



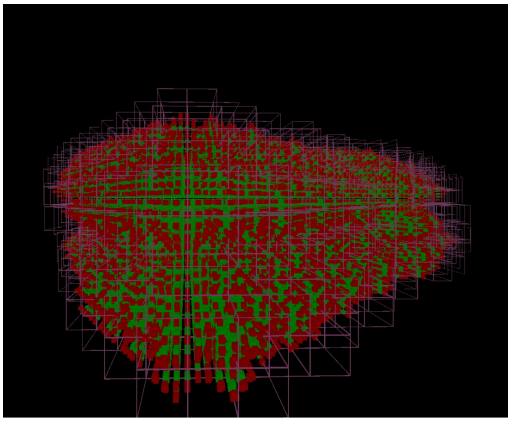
(c) Scan line.



(d) Morton code



(e) Marching cubes.



(f) Fast marching method debug näkymä.