Matriisilaskennan algoritmiikka 3D-renderöinnissä

LuK-tutkielma
TURUN YLIOPISTO
Tulevaisuuden teknologioiden laitos
Tietojenkäsittelytiede
2020
Taneli Tainio

TURUN YLIOPISTO
Tulevaisuuden teknologioiden laitos

Taneli Tainio: Matriisilaskennan algoritmiikka 3D-renderöinnissä

LuK-tutkielma, 22 s., 1 liite.
Tietojenkäsittelytiede
Joulukuu 2020

Kolmiulotteinen renderöinti on viihdeteollisuudessa hyvin tärkeää, erityisesti elokuvien ja televisiosarjojen valmistuksessa ja videopeliteollisuudessa. Kolmiulotteisessa renderöinnissä matematiikka ja algoritmiikka ovat tärkeitä aiheita, varsinkin vektori- ja matriisilaskenta.

Tässä tutkielmassa on tarkoitus selvittä mitkä ovat keskeisimmät kolmiulotteiseen renderöintiin liittyvistä vektori- ja matriisilaskuista, miten niiden laskeminen on mahdollista toteuttaa algoritmisesti ja minkälaisia ongelmia näissä algoritmeissa syntyy. Laskutoimitusten määritelmiin perustuen luodaan n ulotteisia vektoreita ja matriiseja käsittelevät algoritmit ja erityishuomiota kiinnitetään niiden aikakompleksisuuksiin sekä liukulukulaskuista muodostuviin virheisiin.

Koska kolmiulotteista renderöintiä on toteutettu jo 1900-luvulla, on kehitetty paljon erilaisia menetelmiä laskutoimitusten laskemisen nopeuttamiseksi, kuten rinnakkaislaskentaan erikoistettuja näytönohjaimia ja vektorilaskuja suoraan suorittavia prosessoreja. Tämänkaltaisien optimointimenetelmien toimivuus ja vaikutus määriteltyihin algoritmeihin on tutkielman loppuosan keskeisenä asiana.

Edellä mainittujen menetelmien lisäksi tutkitaan kvaternioiden mahdollisia hyötyjä kolmiulotteisessa renderöinnissä vertaamalla kolmiulotteisen mallin rotaatiotransformaation laskemiseen kuluvaa aikaa käyttäen kvaternioita ja matriiseja, sekä itse rotaatiotransformaatioiden muodostamiseen kuluvaa aikaa kvaternioita ja matriiseja käyttämällä.

Vektorilaskut ovat moderneilla laitteilla hyvin tuettuja operaatioita ja usein vievät vain yhden prosessorin käskyn verran. Matriisi-vektori-kertolaskut ilmenevät usein suurissa määrissä, esimerkiksi juuri kolmiulotteisen kappaleen transformaatioita laskiessa, jolloin näytönohjaimien mahdollistama satojen, jopa tuhansien, laskutoimitusten rinnakkaistaminen on tärkeämpää prosessorin tarjoamien laskutoimituksien suoraan tukeen verrattuna. Lisäksi kvaternioiden eduista huolimatta ne eivät täysin korvaa matriisien käyttöä kolmiulotteisessa renderöinnissä, mutta esimerkiksi rotaatiotransformaatioita laskiessa ovat matriisien käyttöä nopeampia.

Asiasanat: matriisilaskenta, 3D-renderöinti, kvaternio, algoritmiikka

UNIVERSITY OF TURKU
Department of Future Technologies

Taneli Tainio: The Algorithms of Matrix Calculations in 3D Rendering

Bachelor's Thesis, 22 p., 1 app. p.
Computer Science
December 2020

3D rendering is very common within the entertainment industry, especially in the creation of movies and TV-series and in the videogame industry. Mathematics and algorithms play an important role in 3D rendering, specifically vector and matrix calculations.

In this thesis the objective is to figure out the most important vector and matrix calculations in 3D rendering, how they can be computed and what problems do these algorithms have. Based on the definitions of the calculations, algorithms which operate on n dimensional vectors and matrices are created and special attention is placed on their growth rates and error from floating point calculations.

3D rendering has existed since the 20th century and because of this a lot of different methods for optimizations have been created, for example parallel computations of calculations via a graphics card and direct implementation of vector calculations within an instruction set. The effectiveness and effects of these kinds of methods are the focus of the latter part of this thesis.

In addition to the methods described previously, quaternions are studied as a means of optimization and they are compared against matrices with respect to speed in the calculations of a rotation transformation for a 3D model and the speed of constructing a 3D rotation matrix.

Vector calculations on modern devices are very well supported and often take only a single processor instruction to complete. Matrix-vector-multiplications are often encountered in large quantities, for example in calculating a transformation for a 3D model, where the massive potential for computing hundreds or thousands of calculations in parallel in a graphics card are more important than direct support for such calculations within an architecture. In addition, despite the benefits of quaternions they can not entirely replace the use of matrices within 3D rendering, but for example in calculating rotation matrices they are faster.

Keywords: matrix calculus, 3D rendering, quaternion, algorithms

Sisällys

1 Johdanto

Kolmiulotteinen renderöinti on nykymaailmassa hyvin yleistä. Sille on käyttöä erityisesti viihdeteollisuudessa elokuvien ja televisiosarjojen valmistuksessa, varsinkin animoitujen elokuvien ja sarjojen tuotannossa, sekä videopeliteollisuudessa mahdollistamassa todenmukaisien kuvien muodostamisen realiajassa kuluttajalaitteilla. Kolmiulotteisella renderöinnillä tarkoitetaan tässä tutkielmassa laskentaa, jota vaaditaan kuvien muodostamiseen tietokoneella kolmiulotteisista malleista. Tämän kuvauksen pohjalta on helppo päätellä, että kolmiulotteisessa renderöinnissä matematiikka on hyvinkin keskeinen konsepti, kuten myös algoritmiikka.

Tutkimuskysymykset

Kolmiulotteisiin avaruuksiin mahtuu paljon erilaisia matemaattisia konsepteja ja laskutoimituksia. Tässä tutkielmassa on tarkoitus selvittä mitkä ovat keskeisimmät näistä laskutoimituksista kolmiulotteisessa renderöinnissä. Mahdollisuuksia on vielä rajattu keskittymällä matriiseihin ja vektoreihin.

Toinen osa kolmiulotteisesta renderöinnistä on kuvan muodostaminen, ja se tapahtuu tietokoneella. Laskutoimituksien selvittämisen lisäksi on syytä tutkia miten ne on mahdollista toteuttaa tietokoneella laskettaviksi ja minkälaisia ongelmia näiden toteutuksien käytössä on.

Kolmiulotteista renderöintiä on toteutettu tietokoneilla jo 1900-luvulla, joten on helppo kuvitella laskutoimituksia laskevien menetelmien, ohjelmien ja algoritmien kehittyneen vuosien saatossa. Tässä tutkielmassa ei siis tyydytä vain toimivan algoritmin selvittämiseen, vaan tutkitaan mitä menetelmiä modernit tietokoneet hyödyntävät tutkielman keskeisten laskutoimitusten laskemiseen.

Tutkimusmenetelmät

Laskutoimitukset on mahdollista selvittää tutustumalla matriisilaskennan peruslaskutoimituksiin. Näille laskutoimituksille muodostetaan niiden määritelmiin perustuvat abstraktit toteutukset. Muodostettuihin toteutuksiin kohdennetaan tarkempaa tutkimusta niiden ominaisuuksista.

Optimointimenetelmiä tutkitaan muutamalla tavalla: selvittämällä mahdollisia prosessorien toteutuksissa olevia optimointeja ja tutustumalla vaihtoehtoisiin laskumenetelmiin. Prosessorien ja muiden erikoislaitteiden mahdollistamia optimointimenetelmiä on enemmän kuin tässä tutkielmassa on mahdollista käsitellä, joten niistä tutkitaan vain tärkeimpiä ja ilmeisimpiä. Vaihtoehtoisiin laskumenetelmiin lukeutuu kvaterniot, joiden tarjoamia vaihtoehtoja tutkitaan luvussa 6.

Huomautuksia

Tässä tutkielmassa on käytössä alfanumeerinen käytäntö lähdeviittauksille. Hakasulkeiden väliin jäävät kolme tai neljä kirjainta ja kaksi numeroa tarkoittavat viittausta, esimerkiksi [Knu98] on lähdeviittaus. Tämän lisäksi sivunumerot ja muut huomiot lähdeviittauksissa on merkitty hakasulkeiden sisälle: [Knu98, s.214]. Lähteet on listattu osiossa 7, jossa lähteet esiintyvät lähdeviittausmerkintöjen mukaan aakkosjärjestyksessä.

2 Liukuluvut

Tietokoneet ovat erinomaisia kokonaislukulaskuissa ja lähes poikkeuksetta myös virheettömiä kokonaislukuja käsitellessä. Kolmiulotteinen renderöinti kuitenkin toteutetaan käyttäen liukulukuja kokonaislukujen sijan. Jotta tämän tutkielman keskeisimpiä algoritmeja voi tutkia huolellisesti, on ensin käsiteltävä liukulukujen ominaisuuksia, etuja ja puutteita.

2.1 Esitys tietokoneissa

Jokainen realiluku on mahdollista esittää kaavan (1) esittämänä tulona. [Knu98, s.214]

$$a = f_a \cdot b^{e_a} \tag{1}$$

jossa a on mikä tahansa reaaliluku, b on kantaluku, f_a on b-kantainen murto-osa siten että $|f_a| < 1_b$ ja e_a on kantaluvun eksponentti. Esimerkiksi $43.65 = 0.4365 \cdot 10^2$. Tästä esityksestä on hyvä huomata, että kantaluku ei ole muuttuva, vaan on riippuvainen murto-osan esityksestä. Jos murto-osan esittää 2-kantaisena lukuna ja kiinnittää kantaluvuksi 2, saa kaavasta (1) muodostettua kaavan (2) esityksen.

$$a = f \cdot 2^e \tag{2}$$

Näin minkä tahansa reaaliluvun saa esitettyä kahdella toisistaan erillisellä luvulla: 2-kantaisella murto-osalla f jolla pätee $|f| < 1$ ja eksponentilla joka on kokonaisluku. Murto-osan voi vielä jakaa kahteen osaan irroittamalla merkin, jolloin tietokoneen liukulukuesityksen kolme kenttää ovat merkki, eksponentti ja murto-osa. [CHN99] Koska kentät ovat toisistaan erillisiä voi kenttien koot määrittää mielivaltaisesti tarpeen mukaan. Liukulukuja voi merkitä kaavan (3) esittämällä tavalla.

$$a = (e_a, f_a) \iff a = f_a \cdot 2^{e_a} \tag{3}$$

Merkillä ei ole kuin kaksi vaihtoehtoa, joten merkillä on aina yhden bitin mittainen kenttä joka kertoo onko luku negatiivinen (merkki = 1) vai positiivinen (merkki = 0). [iee19] Eksponentti on hyvä ilmaista kokonaislukuna esimerkiksi excess-q-esityksellä, jolloin tarvitaan myös jokin luku q. [Knu98] Murto-osa ilmaistaan usein kiinnitetyn pisteen lukuna siten, että desimaalipiste (*eng.* radix point) sijaitsee bittien vasemmalla puolella, jolloin $|f_a| < 1$. [CHN99] Usein murto-osaan sijoitetaan vasempaan reunaan piilotettu bitti joka on aina yksi, jolloin murto-osalle pätee epäyhtälö (4). [Knu98, CHN99]

$$\frac{1}{2} \leq |f_a| < 1 \tag{4}$$

Tässä esityksessä luvun tarkkuus riippuu murto-osan koosta, joten useimmiten liukulukuesityksissä sille onkin varattu eniten tilaa. Murto-osassa olevien bittien määrä p onkin hyvin tärkeä luku, sillä siitä saadaan seuraavassa luvussa käsiteltäville virheille yläraja. Eksponenttikentän koko puolestaan kertoo pienimmän ja suurimman esitettävän luvun kokoluokan. Esimerkiksi IEEE 754 -standardi 32-bittisille liukuluville määrittääkin seuravat koot eri kentille: s on 1 bitti, e on 8 bittiä ja f on 23 bittiä plus yksi piilotettu 1-bitti. [iee19]

2.2 Laskutoimitukset

Liukuluvuilla laskutoimitukset eivät ole niin yksinkertaisia kuin kokonaisluvuilla, sillä kuten luvussa 2.1 käsiteltiin, liukulukujen esitysmuoto on täysin mielivaltainen jolloin laskutoimituksien suorittamiseen tarvitaan omat algoritmit. Huomautettakoon, että modernit proessorit sulauttavat laskutoimitukset osaksi laitteistoa, muuntaen alla olevien algoritmien kaltaiset menetelmät kiinteiksi osiksi prosessoria. [CHN99] Ensimmäisenä käsittelyssä on Donald Knuthin esittämä yhteenlaskualgoritmi [Knu98, s. 216] erikoistettu 2-kantaisille liukuluvuille. Tämä algoritmi toimii suoraan myös vähennyslaskulle, kun syöte v korvataan syötteellä $-v$.

Algoritmi A (*Yhteenlasku*) Olkoon $u = (e_u, f_u)$ ja $v = (e_v, f_v)$ normalisoituja liukulukusyötteitä. Tämä algoritmi laskee $w \approx u + v$. [Knu98, Algorithm A, s. 216]

A1. (*Pura*) Parametreina saadut bittijonot karsitaan eksponenteiksi e_u ja e_v ja murto-osiksi f_u ja f_v.

A2. (*Oleta* $e_u \geq e_v$) Jos $e_u < e_v$ vaihdetaan u ja v.

A3. (*Aseta* e_w) Asetetaan $e_w = e_u$.

A4. (*Tarkista* $e_u - e_v$) Jos $e_u - e_v \geq p + 2$ asetetaan $f_w = f_u$ ja siirrytään vaiheeseen **A7**.

A5. (*Sovita*) Bittisiirretään f_v oikealle $e_u - e_v$ bittiä.

A6. (*Summaa*) Asetetaan $f_w = f_u + f_v$.

A7. (*Normalisoi*) Suoritetaan normalisointialgoritmi w:lle. [Knu98, Algorithm N, s. 216]

Algoritmin vaiheessa A7 liukuluku $w = (e_w, f_w)$ saattaa olla epänormaalissa muodossa, jolloin epäyhtälön (4) ehdot eivät täyty. Tällöin liukuluku on normalisoitava, jotta sen käyttöä voi jaktaa muiden liukulukujen tavoin. [Knu98, s.216] Normalisointi ei sisälly tämän tutkielman aihepiiriin, joten normalisoinnin yksityiskohdat sivutetaan. Vaiheet A4 ja A5 ovat erityisen tärkeitä liukulukujen virheitä pohdittaessa. Knuthin algoritmi kerto- ja jakolaskuille on algoritmin A kanssa tärkeä.

Algoritmi B (*Kerto- ja jakolasku*) Olkoon $u = (e_u, f_u)$ ja $v = (e_v, f_v)$ normalisoituja liukulukuja. Tämä algoritmi laskee kertolaskun $w \approx u \cdot v$ tai jakolaskun $w \approx u/v$. [Knu98, s. 217]

B1. (*Pura*) Karsi parametreina saadut bittijonot eksponentteihin e_u, e_v ja murto-osiin f_u, f_v.

B2. (*Operoi*) Aseta

$$\begin{array}{lll} e_w = e_u + e_v - q, & f_w = f_u \cdot f_v, & \text{kertolaskussa;} \\ e_w = e_u - e_v + q + 1, & f_w = (2^{-1} f_u)/f_v, & \text{jakolaskussa.} \end{array}$$

B3. (*Normalisoi*) Suoritetaan w:lle normalisointialgoritmi N.

Algoritmista B on hyvä huomata, että vaiheessa B2 molempien parametrien murto-osat kerrotaan suoraan keskenään ja jakolaskussa f_u menettää yhden bitin oikealta.

Algoritmien A ja B suoritusaika on selvästi vastaavia kokonaislukulaskuja hitaampaa. Tämän näkee jo siitä, että kummassakin algoritmissa suoritetaan useampi kokonaislukulaskutoimitus ja algoritmissa B kertoessa kertolasku ja jakaessa jakolasku. Sen lisäksi kummankin algoritmin

ensimmäinen vaihe ei välttämättä ole triviaali ja normalisointi, joka päättää kummankin algoritmin, kestää oman aikansa. Vaikka erikoistettu laitteisto poistaakin tarpeen käyttää kokonaislukuja hyödyntäviä algoritmeja liukuluvuilla laskemiseen, on liukulukulaskut silti kokonaislukulaskuja merkittävästi hitaampia.

2.3 Ongelmat ja laskuvirheet

Liukuluvut ovat luonnostaan epätarkkoja lukuja, sillä niiden esityksessä käytetty murto-osa pystyy säilyttämään vain p merkitsevää numeroa. Algoritmissa A p esiintyy vaiheessa A4, missä verrataan parametrilukujen eksponenttien erotusta lukuun $p+2$ ja mikäli erotus on suurempi ei yhteenlaskua edes vaivauduta laskemaan ja tulos on $w = u$. Tämä johtuu siitä, että jos eksponenttien erotus on tarpeeksi suuri, vaiheessa A5 murto-osaa f_v siirrettäisiin niin paljon oikealle, että se katoaisi vallan ja vaihe A6 muuttuu laskuksi $f_w = f_u + 0$.

Algoritmissa B huomataankin, että murto-osia f_u ja f_v ei muuteta, vaan niillä operoidaan suoraan mikä ei aiheuta enempää virhettä kuin mitä luvuissa u ja v jo valmiina oli (sillä kummatkin hyvin todennäköisesti ovat likiarvoja). Kuitenkin Molempien algoritmien viimeisenä vaiheena oleva normalisointi voi aiheuttaa murto-osan jakamista tai kertomista ja pyöristämistä, jolloin siitä tulee varmasti likiarvo.

Jokaiselle liukulukuja laskevalle laitteelle on olemassa sellainen luku ϵ_m, joka kertoo kuinka tarkasti laite voi laskea liukulukuja. Jos suorittaa n laskutoimitusta liukuluvuilla, niin kumulatiivinen virhe laskuissa saattaa helposti olla luokkaa $n\epsilon_m$, ja vaikka ϵ_m onkin erittäin pieni, suurilla määrillä laskutoimituksia sekin saadaan suureksi, joten on usein kannattavaa tehdä laskutoimituksia suurempaa tarkkuutta olevilla liukuluvuilla kuin mitä lopullisessa vastauksessa tarvitsee, esimerkiksi jos vastausta tarvitaan 32-bittisen liukuluvun täydellä tarkkuudella, on kannattavaa suorittaa laskutoimitukset 64-bittisillä liukuluvuilla ja muuntaa vastaus lopuksi 32-bittiseen muotoon. Tämä eliminoi virheen kokoluokasta $n\epsilon_m$ luokkaan ϵ_m, sillä tarkkuutta pienentäessä jonkin verran joudutaan pyöristämään.[PTVF07]

Kuten jo yllä mainittiin, liukuluvuilla suoritettavat laskut kestävät merkittävästi kauemmin kuin kokonaisluvuilla suoritetut laskut. Tämä, kuten laskuvirheetkin, on hyvin kumuloituva, tosin vaikka laskuvirheeltä voi välttyä käyttämällä korkeamman tarkkuuden liukulukuja, ei sama toimi kumpaankaan suuntaan nopeudessa. Tämä johtuu siitä, että prosessorin liukulukuoperaatiot ovat nopeampia kuin mikään liukulukuja laskeva ohjelma ja eri tarkkuuksilla olevat liukuluvut vaativat saman ajan samoissa operaatioissa, eli esim. 64- ja 32-bittisillä liukuluvuilla tehtävät laskutoimitukset kestävät yhtä kauan suorittaa.

Liukulukuja käytettäessä voitetaankin jo siinä, että kaiken matematiikan ei tarvitse olla diskreettiä, vaan voimme hyödyntää lähes jatkuvaa esitystä realilukujen joukosta. Esimerkiksi kolmiulotteinen avaruus on liukulukujen avulla mahdollista mallintaa hyvinkin suurella tarkkuudella. Tulemmekin huomaamaan, että jopa 32-bittinen liukuluku tarjoaa suurista virheistään huolimatta[1] suuremman tarkkuuden kuin mitä kolmiulotteisessa renderöinnissä usein tarvitaan.

[1] ja matemaatikon suruksi

3 Matriisilaskenta

Kolmiulotteisen renderöinnin keskeisimmät laskutoimitukset ovat vektori- ja matriisilaskuja. Luvussa 4 käsitellään tässä luvussa esitettäviä laskutoimituksia algoritmisesti. Tämä luku pohjautuu Eric Lengyelin kirjaan Mathematics for 3D Game Programming and Computer Graphics [Len12], joka sisältää tämän tutkielman ulkopuoleisia lauseita ja todistuksia.

3.1 Vektorilaskut

Tämän luvun jokaisessa kaavassa pätee $\mathbf{v}, \mathbf{u} \in \mathbb{R}^n$, eli $\mathbf{v}$ ja $\mathbf{u}$ ovat n-ulotteisia vektoreita. Tämän lisäksi merkitään vektoreita kaavan (5) mukaisesti.

$$\mathbf{v} = (v_1, v_2, \ldots, v_n) \tag{5}$$

Erityistä huomiota saavat kolmi- ja neliulotteiset vektorit, joilla pätee seuraavat merkinnät: $v_x = v_1, v_y = v_2, v_z = v_3$ ja $v_w = v_4$. Kaavassa (6) on vektorien $\mathbf{v}$ ja $\mathbf{u}$ summan määritelmä. [Len12] Vektorin jokainen alkio on mahdollista kertoa yhdellä luvulla, skalaarilla, kaavan (7) mukaisesti. [Len12]

$$\mathbf{v} + \mathbf{u} = (v_1 + u_1, v_2 + u_2, \ldots, v_n + u_n) \tag{6}$$

$$a\mathbf{v} = (av_1, av_2, \ldots, av_n) \tag{7}$$

Lengyelin kirjan lause 2.1 antaa laskutoimituksille (6) ja (7) assosiatiivisuus-, kommutatiivisuus- ja distributiivisuuslait. [Len12, s. 12]

Vektorin voi määritellä siten, että sillä on suunta ja koko, tai magnitudi. Kaavassa (5) olevasta esitysmuodosta saamme vektorin magnitudin kaavan (8) avulla. [Len12, s.13]

$$\|\mathbf{v}\| = \sqrt{v_x^2 + v_y^2 + v_z^2 + v_w^2} \tag{8}$$

Kertomalla minkä tahansa vektorin $\mathbf{v}$ luvulla $\frac{1}{\|\mathbf{v}\|}$ saamme vektorin $\mathbf{v}$ kanssa samansuuntaisen ja yksikkömittaisen ($\|\mathbf{v}\| = 1$) vektorin, yksikkövektorin. [Len12, s. 13] Kaksi vektoria on mahdollista kertoa keskenään kahdella eri tavalla, joista ensimmäisenä on pistetulo. Pistetulo kertoo kaavan (9) (tai (10)) tavoin vektorit siten, että tulo on skalaari. [Len12, määritelmä 2.3, s.15]

$$\mathbf{v} \cdot \mathbf{u} = v_1u_1 + v_2u_2 + \cdots + v_nu_n \tag{9}$$

$$\mathbf{v} \cdot \mathbf{u} = \sum_{i=1}^{n} v_iu_i \tag{10}$$

Pistetulosta saatu skalaari kertoo laskettavien vektorien välisestä kulmasta seuraavaa: jos pistetulo on negatiivinen on vektorien välinen kulma yli 90°, jos pistetulo on positiivinen, on kulma alle 90° ja jos pistetulo on 0, on vektorien välinen kulma tasan 90°. [Len12, lause 2.4, s. 15] Kolmiulotteisia vektoreita on myös mahdollista kertoa keskenään kaavan (11) tavoin siten, että tuloksena on uusi vektori. [Len12, määritelmä 2.6, s.20]

$$\mathbf{v} \times \mathbf{u} = (v_yu_z - v_zu_y, v_zu_x - v_xu_z, v_xu_y - v_yu_x) \tag{11}$$

Tätä kertolaskua nimitetään ristituloksi ja sen seurauksena on sellainen vektori joka on kohtisuorassa kumpaankin alkuperäiseen vektoriin, eli $(\mathbf{v} \times \mathbf{u}) \cdot \mathbf{v} = 0$ ja $(\mathbf{v} \times \mathbf{u}) \cdot \mathbf{u} = 0$. [Len12, lause 2.8, s.21] Ristitulolle löytyy eräänlainen määritelmä n-ulotteisille vektoreille, [GZ14] mutta se ei ole tämän tutkielman aiheen kannalta olellinen.

3.2 Matriisilaskut

Matriisi on taulukko numeroita, joka karakterisoidaan sen rivien ja sarakkeiden määrillä ja yksikäsiteisesti määritellään sen sisältävillä numeroilla, kuten esitetty kaavassa (12). [Len12, s.31] Matriisia jossa on n riviä ja m saraketta kutsutaan $n \times m$ matriisiksi, kuten esimerkiksi kaavan (12) matriisi $\mathbf{A}$ on $n \times m$ matriisi.. [Len12, s.31] Lisäksi jos sarakkeiden ja rivien määrä on sama, eli $n = m$ on kyseessä neliömatriisi. [Len12, s.31] Luvussa 3.1 käsitellyt vektorit ovat sellaisia matriiseja joilla on joko $n = 1$ tai $m = 1$. [Len12, s.12] Vektoria, jonka rivien määrä on yksi (eli $n = 1$) kutsutaan vaakavektoriksi, kun taas vektoria jonka sarakkeiden määrä on yksi ($m = 1$) kutsutaan pystyvektoriksi. [Len12, s.12]

$$\mathbf{A} = \begin{bmatrix} A_{11} & A_{12} & \dots & A_{1m} \\ A_{21} & A_{22} & \dots & A_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & \dots & A_{nm} \end{bmatrix} \tag{12}$$

$$\mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix} = \begin{bmatrix} v_1 & v_2 & \dots & v_n \end{bmatrix}^{\mathrm{T}} \tag{13}$$

Yläindeksillä T ($\mathbf{A}^{\mathrm{T}}$) tarkoitetaan matriisin transpoosia, jossa rivit muutetaan sarakkeiksi ja sarakkeet riveiksi, kuten yhtälössä (13) ja 2×2-neliömatriisille yhtälössä (14).

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}^{\mathrm{T}} = \begin{bmatrix} A_{11} & A_{21} \\ A_{12} & A_{22} \end{bmatrix} \tag{14}$$

Vektoreilla tiedetään olevan yhteenlasku määriteltynä ja koska vektorit ovat erityisiä matriiseja, on selvää, että matriiseillakin on määriteltynä yhteenlasku. Kahden matriisin $\mathbf{A}$ ja $\mathbf{B}$ yhteenlasku on määritelty kaavan (15) esittämästi alkioittain, kuten vektoreillakin. [Len12, s.32]

$$\mathbf{A} + \mathbf{B} = \begin{bmatrix} A_{11} + B_{11} & A_{12} + B_{12} & \dots & A_{1n} + B_{1n} \\ A_{21} + B_{21} & A_{22} + B_{22} & \dots & A_{2n} + B_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n1} + B_{n1} & A_{n2} + B_{n2} & \dots & A_{nn} + B_{nn} \end{bmatrix} \tag{15}$$

Matriisin kertominen skalaarilla on määritelty kaavan (16) mukaisesti, kuten vektoreillekin, ja

kertolasku on kommutatiivinen. [Len12, s.32]

$$a\mathbf{A} = \mathbf{A}a = \begin{bmatrix} aA_{11} & aA_{12} & \dots & aA_{1n} \\ aA_{21} & aA_{22} & \dots & aA_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ aA_{n1} & aA_{n2} & \dots & aA_{nn} \end{bmatrix} \tag{16}$$

Kahden matriisin välinen kertolasku ei ole yhtä suoraviivainen kuin yhteenlasku ja skalaarilla kertominen. Jotta kaksi matriisia $\mathbf{A}$ ja $\mathbf{B}$ on mahdollista kertoa yhteen, on $\mathbf{A}$:n sarakkeiden määrän oltava sama kuin $\mathbf{B}$:n rivien määrä, eli $n \times m$ matriisin $\mathbf{A}$ ja $m \times l$ matriisin $\mathbf{B}$ kertolasku on määritelty. [Len12, s.32] Edellisestä lauseesta näkee suoraan, että kahden ei-neliömatriisin kertolasku ei ole kommutatiivinen. Matriisien kertolasku lasketaan tulosmatriisiin alkioittain, ja jokainen alkio on sitä vastaavan $\mathbf{A}$:n rivin ja $\mathbf{B}$:n sarakkeen pistetulo. [Len12, s.32] Matriisikertolasku $\mathbf{C} = \mathbf{AB}$ on esitetty alkiokohtaisesti kaavassa (17) hyödyntäen pistetulon kaavaa (10).

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj} \tag{17}$$

Matriisien kertolaskusta on hyvä huomata, että vektorin $\mathbf{v}$ ja matriisin $\mathbf{A}$ kertolasku on määritelty vain järjestyksessä $\mathbf{Av}$. Kuitenkin kertolasku vektorin transpoosin $\mathbf{v}^{\mathrm{T}}$ ja matriisin välillä on määritelty järjestyksessä $\mathbf{v}^{\mathrm{T}}\mathbf{A}$. Nämä huomiot seuraavat suoraan matriisikertolaskun määritelmästä ja erityisesti vaatimuksesta, että vasemmanpuoleisen matriisin sarakkeiden ja oikeanpuoleisen rivien lukumäärän on täsmättävä. Matriisikertolasku ei ole kommutatiivinen, mutta täyttää assosiatiivi- ja distributiivilait. [Len12, lause 3.2, s.33]

4 Algoritmiikka

Luvussa 3 esitellyt laskutoimitukset ovat tärkeitä kolmiulotteisessa renderöinnissä, mutta jotta niitä on mahdollista käyttää, on ensin muodostettava laskutoimituksia suorittavat algoritmit. Kuten luvussa 2.3 mainittiin, tietokoneet eivät ole täsmällisiä realilukujen suhteen, joten kun realilukuinen vektoriavaruus kohtaa tietokoneiden rajoitukset on luvussa 2 käsiteltyjä asioita sovellettava algoritmien muodostamisen ja analysoinnin lisäksi.

4.1 Vektorilaskujen algoritmit

Kuten luvussa 3.2 ilmeni, vektorit ovat erikoisia matriiseja. Tämä yhteys kantautuu myös algoritmeihin, joten algoritmien tutkiminen on hyvä aloittaa vektoreista jolloin matriisien algoritmit ovat helpommin ymmärrettävissä.

4.1.1 Vektorien yhteenlasku ja vektori-skalari -kertolasku

Tutkitaan ensimmäisenä kahden vektorin välistä yhteenlaskua. Kahden vektorin yhteenlaskun määritelmä antaa suoraan toimivan algoritmin, joka on esitelty alla.

Algoritmi Y (*Vektorien yhteenlasku*) Olkoon $\mathbf{u}, \mathbf{v} \in \mathbb{R}^n$ annettuja vektoreita. Tämä algoritmi laskee $\mathbf{w} = \mathbf{u} + \mathbf{v}$.

Y1. (*Alusta*) Luo $\mathbf{w} = \mathbf{0}$ ja $i = 1$.

Y2. (*Laske*) Aseta $w_i = u_i + v_i$.

Y3. (*Askel*) Aseta $i = i + 1$.

Y4. (*Tarkista*) Jos $i \leq n$ siirry vaiheeseen **Y2**.

Y5. (*Palauta*) Palauta $\mathbf{w}$.

Algoritmi Y tiedetään oikeaksi, sillä vaihe Y2 on yleinen kaava kahden vektorin i:nnen elementin summalle. Nähdään myös, että i saa arvot yhdestä n:ään, joten jokainen vektorien elementti tulee käytyä läpi. Vaihe Y4 on ainoa vaihe jossa siirrytään taaksepäin. Huomataan että takaisin siirrytään tismalleen silloin kun $i \leq n$ ja jokaisella paluukerralla i:n arvo kasvaa yhdellä. Taaksepäin palataankin tämän perusteella n kertaa. Jokaisella kierroksella tapahtuu kaksi yksinkertaista laskutoimitusta: liukulukujen yhteenlasku ja kokonaisluvun korotus. Nämä ovat vakioaikaisia toimintoja, joten taulukon 1 mukaan saamme algoritmin Y aikakompleksisuuden. Laskemalla jokaisen vaiheen suoritusmäärän yhteen saamme aikakompleksisuudeksi $3n + 2 = O(n)$. Huomataan myös, että algoritmin Y suoritusaika on täysin sidoksissa n:ään, joten voimme tiivistää sen aikakompleksisuudeksi $\Theta(n)$.

Vaihe	**Määrä**
Y1	1
Y2	n
Y3	n
Y4	n
Y5	1

Taulukko 1: Algoritmin Y vaiheiden suoritusmäärä.

Kun n on tunnettu ja pieni, erityisesti $n = 3$ tai $n = 4$, algoritmin silmukan voi avata ja indeksimuuttujan i voi poistaa. Tämä jättää algoritmin Y aikakompleksisuudeksi $\Theta(1)$ ja sitä voidaan käsitellä alkeisoperaation tavoin. Koska jokainen laskutoimitus on toisistaan erillinen, eli yhden laskutoimituksen tulosta ei käytetä myöhemmin algoritmissa, on lopullinen virhe ϵ_m jokaista w_i kohti, jossa ϵ_m on luvussa 2.3 esitelty virheluku. Virhe on käytännössä niin pieni kuin mahdollista, joten esitelty algoritmi Y on kelvollinen toteutus vektorien summausalgoritmiksi.

Vektorin ja skalarin kertolaskualgoritmin saa algoritmista Y muuttamalla vaiheen Y2 laskutoimitus kertolaskuksi ja asettamalla $v_i = a$ jokaisella $i = 1, 2, \ldots, n$, missä a on kertova skalari. Helposti huomataan, että koko vaiheen Y2 voi muuttaa muotoon

Y'2. (*Laske*) $w_i = u_i \cdot a$.

jolloin säästyy $n - 1$ lukuarvon verran muistia ja yhden vektorin luominen.

4.1.2 Vektorien pistetulo

Vektorien pistetulo onkin hieman mielenkiintoisempi laskea kuin yhteenlasku ja seuraavaa algoritmia voimmekin hyödyntää myöhemmin osiossa 4.2.2.

Algoritmi P (*Pistetulo*) Tämä algoritmi laskee kahden syötevektorin $\mathbf{u}, \mathbf{v} \in \mathbb{R}^n$ pistetulon $p = \mathbf{u} \cdot \mathbf{v}$.

P1. (*Alusta*) Aseta $p = 0, i = 1$.

P2. (*Laske*) Laske $p = p + (u_i \cdot v_i)$.

P4. (*Askel*) Aseta $i = i + 1$.

P5. (*Tarkista*) Jos $i \leq n$ palaa vaiheeseen **P2**.

P6. (*Lopeta*) Palauta p.

Algoritmien P ja Y rakenteet ovat hyvinkin samanlaisia. Merkittävimpinä eroina on algoritmissa P oleva kertolasku yhteenlaskun lisäksi vaiheessa P2 ja tulos on vektorin sijasta skalaari. Yhtenevän rakenteensa vuoksi kummallakin algoritmilla on sama aikakompleksisuusluokka, nimellisesti $\Theta(n)$. On kuitenkin syytä huomata, että algoritmissa P suoritetaan yhden laskutoimituksen sijasta kaksi laskentavaiheessa. Tästä syystä algoritmilla P on hitaampi suoritusaika kuin algoritmilla Y, joskin vain vakiokertoimen verran.

Algoritmi P käyttää akkumulointimenetelmää tulosta laskiessa, joten algoritmi käyttää hyvin vähän muistia, vain kahden lukuarvon verran, joista indeksimuuttuja i on todennäköisesti kokonaislukumuuttuja ja jos $n \leq 255$ voi viedä vain yhden tavun verran. Tulosmuuttuja p voi olla samaa datatyyppiä kuin vektorien luvut, tai mikäli tarvetta säilyttää mahdollisimman korkea tarkkuus: korkeamman tarkkuuden liukuluku. Algoritmi P on hyvä esimerkki tilanteesta, jossa liukulukulaskennan virheet saattavat vaikuttaa.

Kuten luvussa 2.3 mainittiin liukulukulaskuissa tapahtuva virhe on kumulatiivista, ja yllä todettiin algoritmin P käyttävän kumuloivaa laskentamenetelmää, joten on syytä tutkia mahdollista virhettä. Muuttujaan p tehdään muutoksia vain vaiheessa P2. Luomisvaiheessa virhettä ei ole, sillä p:n arvoksi asetetaan nolla ja nolla on mahdollista esittää täsmällisesti. Vaiheessa P2 tapahtuu kaksi laskutoimitusta: vektorien komponenttien välinen kertolasku ja tulon ja p:n yhteenlasku. Koska tämä vaihe suoritetaan täsmälleen kerran jokaista i:n arvoa kohti, on virhe yhdellä silmukan kierroksella $2\epsilon_m$.

Vaihe P2 suoritetaan n kertaa, jolloin lopullinen virhe on $2n\epsilon_m$. Virhe on siis riippuvainen n:stä, joten sen lisäksi, että suurella n:llä algoritmin suoritus kestää kauemmin, on myös tuloksessa enemmän virhettä. Kolmiulotteisessa renderöinnissä usein $n = 3$ tai $n = 4$, jolloin virhe on väliltä $[6\epsilon_m, 8\epsilon_m]$ ja erityisesti kolmiulotteisessa renderöinnissä on hyväksyttävien rajojen sisällä. On syytä huomata, että tämä johtaa suurempiin virheisiin tulosta käyttävissä laskuissa.

4.1.3 Vektorien ristitulo

Kahden vektorin välinen ristitulo on, kuten luvussa 3.1 todettiin, määritelty vain kolmelle ulottuvuudelle.[2] Ristituloa laskeva algoritmi on siitä erikoinen, että syötteen koko on kiinnitetty, joten se ei ole kaikkien määritelmien mukaan algoritmi. Ristitulosta kuitenkin ilmenee ominaisuuksia joita ei aiemmissa algoritmeissa ole, joten ristituloa laskevaa algoritmia on silti kannattava tutkia.

[2]Tämän tutkielman puitteissa.

Algoritmi R (*Ristitulo*) Tämä algoritmi laskee kahden kolmiulotteisen syötevektorin $\mathbf{u}, \mathbf{v} \in \mathbb{R}^3$ ristitulon $\mathbf{w} = \mathbf{u} \times \mathbf{v}$.

R1. (*Alusta*) Luo $\mathbf{w} = \mathbf{0}$.

R2. (*Laske x*) Aseta $w_x = u_z v_y - u_y v_z$.

R3. (*Laske y*) Aseta $w_y = u_x v_z - u_z v_x$.

R4. (*Laske z*) Aseta $w_z = u_x v_y - u_y v_x$.

R5. (*Lopeta*) Palauta $\mathbf{w}$.

Monimutkaisin asia algoritmissa R onkin laskutoimitukset jotka saadaan suoraan ristitulon matemaattisesta määritelmästä. Koska algoritmilla R on kiinteä määrä syötettä (kaksi kolmiulotteista vektoria) ja yhdessäkään vaiheessa ei synny haaraa on algoritmin R aikakompleksisuus $\Theta(1)$ ja se myös tuottaa kiinteän määrän virhettä. Jokainen tulosvektorin $\mathbf{w}$ komponentti lasketaan erillisillä laskuilla on kasaantuva virhe komponenttikohtainen. Kaksi kertolaskua ja vähennyslasku tuottavat kumulatiivisen virheen $3\epsilon_m$ jokaista tulosvektorin komponenttia kohti. Tässä on syytä huomata, että syötevektorien jokaista komponenttia käytetään kahden tulosvektorin komponentin laskemisessa, joten suuri virhe yhdessä syötevektorin komponentissa haarautuu suuremmaksi virheeksi kahdessa tulosvektorin komponentissa.

4.2 Matriisilaskujen algoritmit

Hyödyntämällä luvun 4.1 algoritmeja ja tietoa matriisien ja vektorien yhteydestä, on tämän luvun algoritmit helposti muodostettavissa. Lukuun 4.2.2 on syytä kiinnittää erityistä huomiota, sillä siinä käsitellyt asiat ovat erityisen tärkeitä myöhemmin luvuissa 5 ja 6.

4.2.1 Matriisien yhteenlasku

Kahden vektorin yhteenlaskua laskevaa algoritmia Y on helppo jatkaa ympäröimällä vaiheet Y2-Y4 toiseen silmukkaan jossa indeksimuuttujana $j = 1, 2, \ldots, m$. Muuttamalla esiintyvät vektorit matriiseiksi saammekin toteutettua kahden $n \times m$ matriisin yhteenlaskun laskevan algoritmin.

Algoritmi M (*Matriisien yhteenlasku*) Tämä algoritmi laskee kahden matriisin $\mathbf{A}, \mathbf{B} \in \mathbb{R}^n \times \mathbb{R}^m$ summan $\mathbf{C} = \mathbf{A} + \mathbf{B}$.

M1. (*Alusta*) Aseta $i = j = 1$ ja luo tyhjä $n \times m$ matriisi $\mathbf{C}$.

M2. (*Laske*) Aseta $c_{ij} = a_{ij} + b_{ij}$.

M3. (*Askel rivi*) Aseta $i = i + 1$.

M4. (*Tarkista rivi*) Jos $i \leq n$ palaa vaiheeseen **M2**.

M5. (*Askel sarake*) Aseta $j = j + 1$ ja $i = 1$.

M6. (*Tarkista sarake*) Jos $j \leq m$ palaa vaiheeseen **M2**.

M7. (*Lopeta*) Palauta $\mathbf{C}$.

Koska algoritmi M on jatke algoritmille Y, pätee monet luvussa 4.1.1 tehdyt väittet myös algoritmiin M. Virhe on komponenttikohtainen ja vain ϵ_m kokoinen yhden laskutoimituksen johdosta. Suurin muutos näiden algoritmien välillä onkin ympäröivän silmukan lisääminen, joka

vaikuttaa merkittävästi aikakompleksisuuteen. Sisempi silmukka, eli vaiheet M2-M4 suoritetaan n kertaa. Kun $i = n + 1$ algoritmi siirtyy korottamaan j:tä ja tarkistamaan onko $j \leq m$ ja mikäli on jatkuu algoritmin suoritus vaiheesta M2 jolloin koska i:n arvo asetettiin samaan mikä algoritmin suoritusta aloittaessa, suoritetaan sisempi silmukka jälleen n kertaa. Tämä toistetaan jokaiselle j:n arvolle joita on m kappaletta. Algoritmin M aikakompleksisuus on siis $\underbrace{n + n + \cdots + n}_{m \text{ kertaa}} = nm = O(nm)$.

$O(nm)$ vaikuttaa kovin erikoiselta aikakompleksisuudeksi, ja johtuu täysin siitä, että se kertoo enemmän kuin O-notaatiolla yleensä kerrotaan. Vaikka algoritmin M suoritus riippuu n ja m arvoista, voimme yksinkertaistaa aikakompleksisuuden ylärajaa kun huomataan, että algoritmin M silmukoiden järjestystä voi vaihtaa vaikuttamatta algoritmin oikeellisuuteen. Vaihtamalla vaiheissa M3-M6 i:n ja j:n paikkaa saamme tismalleen saman algoritmin, joka tosin laskee rivi kerrallaan yhteenlaskua eikä sarake kerrallaan. Voimmekin siis merkitä seuraavassa laskussa $n = \max\{n, m\}$ jolloin saamme aikakompleksisuudeksi tutumman $O(n^2)$.

Algoritmi M on yleisempi muoto algoritmille Y, jonka aikakompleksisuus on $O(n)$, ja kun $m = 1$ ovat algoritmit M ja Y suoritukseltaan lähes identtiset. Algoritmissa M on vain yhden muuttujan alustus, yksi korotus ja yksi tarkistus enemmän kuin algoritmissa Y. Tästä johtuen algoritmille M pätee alaraja $\Omega(n)$.

4.2.2 Matriisien kertolasku

Kertolaskun laskevan algoritmin saa muodostettua suoraan kaavasta (17) hyödyntämällä algoritmeista M ja P opittua tietoa.

Algoritmi K (*Matriisien kertolasku*) Tämä algoritmi laskee matriisien $\mathbf{A} \in \mathbb{R}^{m \times n}$ ja $\mathbf{B} \in \mathbb{R}^{n \times l}$ matriisitulon $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$.

K1. (*Alusta*) Aseta $i = j = k = 0$ ja luo $m \times l$ nollamatriisi $\mathbf{C}$.

K2. (*Laske*) Aseta $c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$.

K3. (*Askel k*) Aseta $k = k + 1$.

K4. (*Tarkista k*) Jos $k \leq n$ palaa vaiheeseen **K2**.

K5. (*Askel rivi*) Aseta $i = i + 1$ ja $k = 1$.

K6. (*Tarkista rivi*) Jos $i \leq m$ palaa vaiheeseen **K2**.

K7. (*Askel sarake*) Aseta $j = j + 1$ ja $i = 1$.

K8. (*Tarkista sarake*) Jos $j \leq l$ palaa vaiheeseen **K2**.

K9. (*Lopeta*) Palauta $\mathbf{C}$.

Algoritmin K paikkaansapitävyydestä voi varmistua vertaamalla yllä olevaa esitystä esimerkiksi Skienan antamaan esitykseen [Ski12, s.401] tai toteutukseen [Ski12, s.45]. Käyttämällä samanlaista päättelyä kuin aiempien algoritmien kohdalla, näemme helposti, että algoritmin K aikakompleksisuus on luokkaa $\Theta(nml)$ ja lyhyemmin $\Theta(n^3)$. On olemassa myös toisenlaisia algoritmeja jotka laskevat kahden matriisin tulon. Näiden algoritmien aikakompleksisuus on usein alle $O(n^3)$, mutta käytännössä ovat nopeampia vain, kun n on suuri. Esimerkiksi Strassen algoritmilla on aikakompleksisuus luokkaa $O(n^{2.81})$, mutta toimii algoritmin K toteutusta nopeammin

vasta kun $n \approx 100$ tai suurempi. [Ski12, s.402] Kolmiulotteisessa renderöinnissä kuitenkin useimmiten $n \leq 4$, joten algoritmi K on käytännössä kannattavin, sillä se on myös hyvin yksinkertainen toteuttaa. Luvussa 6.2 on esitetty hieman algoritmin K toteutuksen suorituskyvystä.

Varovainen analyysi algoritmista K paljastaa jokaisen tulosmatriisin alkion c_{ij} olevan pistetulo lähtömatriisin $\mathbf{A}$ rivivektorin i ja $\mathbf{B}$ pystyvektorin j välillä. Pistetuloa laskevan algoritmin P todettiin tuottavan $2n\epsilon_m$ suuruisen virheen, josta seuraa, että algoritmi K tuottaa saman $2n\epsilon_m$ suuruisen virheen jokaiselle tulosmatriisin alkiolle. Kuten pistetulonkin kohdalla, tämä on hyväksyttävän pieni virhe kolmiulotteisessa renderöinnissä.

Erityisen kiinnostava tapaus saadaan algoritmista K kun lähtömatriisi $\mathbf{B}$ on $n \times 1$ matriisi, eli vektori. Yksi kolmiulotteisen renderöinnin keskeisimpiä laskutoimituksia onkin matriisin ja vektorin välinen kertolasku, sillä pisteet esitetään vektoreina ja jotta ne saataisiin oikeille paikoilleen ruudulla tarvitsee niitä muuntaa. Tähän muuntamiseen käytetään matriiseja jotka toteuttavat yhtälön (18).

$$\mathbf{v}' = \mathbf{M} \cdot \mathbf{v} \tag{18}$$

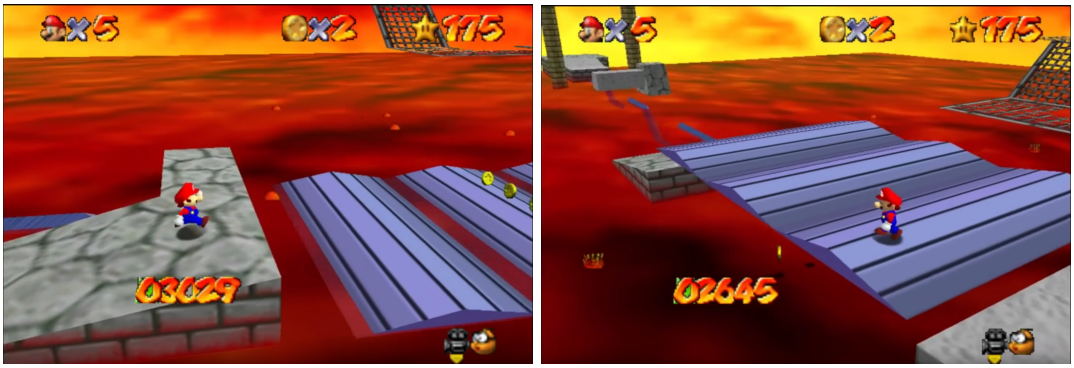
Tässä $\mathbf{v}$ on jokin lähtöpiste, $\mathbf{v}'$ on maalipiste ja $\mathbf{M}$ on transformaatiomatriisi. [Len12, s.67] Algoritmilla K saa laskettua matriisin ja vektorin väliset kertolaskut, mutta koska tämä laskutoimitus on niin tärkeä, on kannattavaa toteuttaa erikoistettu versio algoritmista K joka jättää vaiheet K7 ja K8 pois. Kun $l = 1$ on algoritmin K aikakompleksisuus $\Theta(nm) = \Theta(n^2)$.

4.3 Algoritmien analyysiä

Jokainen luvuissa 4.1 ja 4.2 esiintynyt algoritmi esitettiin geneerisesti (*eng.* generic), eli niiden esityksissä ei otettu kantaa laskutoimituksien kestoihin, datatyyppeihin tai syötteen kokoon. Analysoitaessa oletuksena oli, että datatyyppinä on liukuluvut mutta tämä ei ole osana algoritmien määritelmää. Jotkin ohjelmointikielet mahdollistavat funktioiden määrittämisen vastaavalla tavalla geneerisesti. Esimerkiksi C++:ssa on mahdollista määrittää matriisien kertolaskufunktio riippumaan datatyypistä ja matriisien koista. Tämä menettely on kuitenkin vain C++ standardin määräämä kääntäjän tarjoama työkalu. Lopullinen tuotettu ohjelma voi kuitenkin sisältää jokaiselle n:lle ja datatyypille oman, erillisen, funktionsa.[3] Tästä syystä on hyvä tutkia hieman tarkemmin kolmiulotteisessa renderöinnissä yleisimpiä tapauksia, eli $n = 3$ ja $n = 4$.

Kun n on kiinnitetty, oli se sitten 3 tai 4, on mahdollista keskittyä aikakompleksisuuden sijan algoritmin suoritusaikaan. Tätä harkitessa laskutoimitusten kestot muuttuvat merkityksellisiksi ja jokaisen haaran todennäköisyydet olellisiksi. Modernit prosessorit kykenevät ennakoimaan ohjelman suoritusta haarautumistilanteissa, jolloin ohjelman suoritus voi välttyä välimuistihudeilta (*eng.* cache miss) ja prosessorien käskyjen suorituksen putkitus ei keskeydy. [Knu05] Algoritmeissa esiintyvät laskutoimitukset ovat kaikki alkeellisia joten niiden suoritusnopeus ei ole riippuvainen toteutuksesta vaan laitteistosta. Toisaalta kun $n = 4$ jokaisen indeksimuuttujan arvojoukko on 1–4 jolloin on todennäköistä, että haarautumiset ja indeksimuuttujien käsittely vie lähes yhtä monta käskyä kuin silmukoiden avaaminen. Tämä ei välttämättä päde matriisien kertolaskuille, jossa laskutoimitus tehdään $4 \cdot 1 \cdot 4 = 16$ kertaa ja toistetuille indekseille. Toisaalta esimerkiksi vektorien yhteenlaskussa vaihe Y2 suoritetaan vain neljä kertaa erillisille indekseille, jolloin sil-

[3] Tämän voi tarkistaa toteuttamalla yksinkertaisen geneerisen funktion ja kääntämällä sen ilman optimointia. K.s. Liite A.



Kuva 1: Simuloitu esitys Super Mario 64-pelissä ilmenevästä liukulukujen pyöristysvirheestä. Peli [Nin96] kuvat [bad18]

mukan avaaminen saattaa olla kannattavaa. Tämän tason optimointi on kuitenkin usein hyvä jättää kääntäjän vastuulle, sillä modernit kääntäjät osaavat muodostaa saman tuloksen tuottavia, mutta merkittävästi nopeampia ja tilatehokkaampia ohjelmia kuin ohjelmoijat.[God19] Lisäksi luvussa 5 ilmenee modernien prosessorien todellinen kyky optimoida käsiteltyjä algoritmeja.

Nopeuden rinnalla yksi merkittävä tekijä algoritmien toiminnassa on niiden tuottama virhe. Jos tässä tutkielmassa esitetyt vektori- ja matriisilaskenta-algoritmit toteutetaan kokonaislukutyyppiä käyttäville vektoreille ja matriiseille on niiden tuottama virhe nolla, sillä kokonaislukulaskut ovat täsmällisiä. Kuitenkin kuten luvussa 2.3 kerrottiin: liukulukulaskut ovat luonnostaan virheellisiä. Jokaisen algoritmin kohdalla onkin esitetty niiden tuottaman virheen yläraja-arvio. Koska virhe on myös kumuloituvaa, on tärkeää osata toimia tämän puitteissa. Sekä luonnolliset laskuvirheet, että pyöristyksestä aiheutuvat virheet kumuloituvat ajan myötä ja voivat aiheuttaa ongelmia sekä ohjelman rendröinnissä että mahdollisessa fysiikkamoottorissa. Yksi esimerkki kumuloituvasta liukulukuvirheestä on Nintendon Super Mario 64-pelissä (1996), jossa pitkän odotuksen jälkeen joidenkin edestakaista liikerataa pitkin kulkevien esteiden liikerata itsessään liikkuu. [bad18] Tämä on havainnollistettu kuvassa 1.

Kuten jo mainittu, kolmiulotteisessa renderöinnissä usein käytetään neliulotteisia vektoreita ja 4×4 matriiseja kuvaamaan kolmiulotteista avaruutta. Tätä menettelyä kutsutaan homogeenisiksi koordinaateiksi (*eng.* homogeneous coordinates).[Len12, luku 4.4] Homogeeniset koordinaatit mahdollistavat kaikki samat transformaatiot kuin kolmiulotteiset koordinaatit sekä translaation (*eng.* translation) sulauttamisen transformaatiomatriisiin (*eng.* transformation matrix). [Len12] Transformaatiomatriiseista tarkemmin luvussa 5.

Kiinnitetään $n = 4$ ja tutkitaan käsiteltyjen algoritmien termikohtaista virhettä ja laskutoimitusten kokonaismäärää. Arvot esitetty taulukossa 2 seuraavasti: virhe on kumuloituvan virheen yläraja tuloksen yhtä termiä kohden ja laskutoimitukset on yhteenlaskujen ja kertolaskujen mukaan jaoteltu. Algoritmissa R on oletettu $u_4 = v_4 = w_4 = 1$ ja algoritmi K' on matriisien kertolaskun erikoistettu versio matriisi-vektori -kertolaskulle.

Jokaisen algoritmin virhetermi vaikuttaa hyvin pieneltä. Tämä pitäisi paikkansa mikäli algoritmit suoritettaisiin aina täsmällisillä lähtöarvoilla ja niiden tuloksia ei käytettäisi muissa laskuissa. Kuitenkaan tämä ei toteudu tosimaalimassa, vaan esimerkiksi transformaatiomatriisit joita

Algoritmi	Virhe	Laskutoimitukset
Y	ϵ_m	$4y$
P	$8\epsilon_m$	$4y$ $4k$
R	$3\epsilon_m$	$3y$ $6k$
M	ϵ_m	$16y$
K	$8\epsilon_m$	$64y$ $64k$
K'	$8\epsilon_m$	$16y$ $16k$

Taulukko 2: Algoritmien termikohtaiset virheet ja laskutoimitusten kokonaismäärät kun $n = 4$.

käytetään kappaleen pisteitä esittävien vektorien kertomiseen ovat usean alkeellisemman transformaatiomatriisin tulo. Laskutoimituksen $\mathbf{D} = \mathbf{C} \cdot \mathbf{A} \cdot \mathbf{B}$ laskemisessa käyttäen algoritmia K matriisien $\mathbf{A}$ ja $\mathbf{B}$ kertolaskun tulo kertoo vielä matriisin $\mathbf{C}$, jolloin yhden $\mathbf{D}$:n alkion lopullinen virheen yläraja on $8\epsilon_m \cdot 4 = 64\epsilon_m$, olettaen matriisien $\mathbf{A}, \mathbf{B}$ ja $\mathbf{C}$ täsmällisyyden. Lopullisen pisteen, joka kerrotaan matriisilla $\mathbf{D}$, virhe saattaa olla jopa $64\epsilon_m \cdot 4 = 256\epsilon_m$.

Taulukon 2 listaamat laskutoimitusten määrät jokaiselle algoritmille on vakio ja algoritmin suoritusnopeudet ovatkin hyvin pitkälti näistä alkeellisista laskutoimituksista kiinni. Mikäli yhteen- ja kertolaskut liukuluvilla vievät yhtä paljon aikaa, kuten esimerkiksi MMIX-tietokoneessa on mallinnettu [Knu05], voitaisiin laskea suoraan algoritmien suoritusajat. Kuitenkin monet prosessorit tukevat arkkitehtuurissaan käskyjä, jotka tekevät useamman laskutoimituksen kerralla. Tähän asti algoritmien suoritusaikaa on harkittu vain yksittäiselle algoritmille, mutta käytännössä joitain esitettyjä algoritmeja on suoritettava satoja tuhansia kertoja hyvin pienessä ajassa (esimerkiksi matriisi-vektori -kertolaskua).

5 Erikoistettujen laitteiden vaikutus

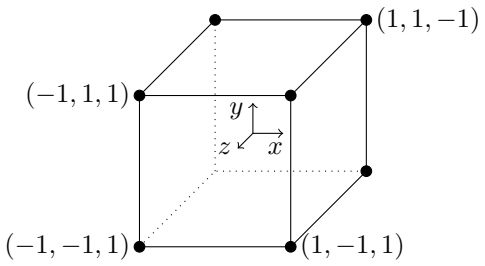
Modernit tietokoneet sisältävät sekä hyvin tehokkaan yleiseen laskentaan ja datankäsittelyyn tarkoitetun keskusprosessorin ja pienempiin, mutta määrällisesti moninkertaisesti vaativampiin laskutoimituksiin tarkoitetun grafiikkaprosessorin sisältävän näytönohjaimen muiden osien lisäksi. Eräs tärkeä modernien näytönohjaimien kehitystä ajava teollisuus on ollut videopeliteollisuus, joka tarjoaa aiemmin tässä tutkielmassa käsitellyille asioille konkreettisen toteutuksen nykymaailmassa.

5.1 Näytönohjaimet ja videopelit

Näytönohjain on tietokoneessa oleva komponentti joka sisältää laskentaan ja rinnakkaistamiseen erikoistetun prosessorin, grafiikkaprosessorin, ja sille tarpeelliset muut osat, kuten oman käyttömuistin ja liittimet sekä tietokoneen ulkopuolelle näyttöihin, että keskusprosessoriin. Modernissa tietokoneessa näytönohjaimen tehtävä on toteuttaa miljoonia laskutoimituksia sekunnissa ja värittää näytölle ilmestyvä jokainen pikseli tietyn väriseksi. Konkreettisin esimerkki näytönohjaimen toiminnasta on videopeleissä ja tämän tutkielman kannalta erityisesti kolmiulotteisissa videopeleissä.

Videopeli on tietokoneohjelma, jonka ensisijainen tarkoitus on viihdyttää käyttäjää. Videopelien rakenteeseen sisältyy hyvin paljon eri osia, mutta tässä tutkielmassa ainoa merkittävä osa on renderöinti. Videopeleissä renderöitävät kappaleet esitetään niiden ääripisteiden avulla: esi-

merkiksi kuutiossa on kahdeksan ääripistettä ja ne riittävät kuution renderöimiseksi ruudulle. Jokainen piste esitetään käyttäen kolmea koordinaattia: x, y ja z koordinaattia siten, että origo on yleensä kappaleen massakeskipisteessä tai muussa käytännöllisessä paikassa. Esimerkki tälläisestä esityksestä on kuvassa 2, jossa origo on kuution massakeskipisteessä.



Kuva 2: Kuution esitys pisteiden avulla kolmiulotteisessa avaruudessa.

Tämä kuution esitys yhdistetään johonkin objektiin pelimaailmassa, jolla on omat sijaintikoordinaatit ilmaisemaan objektin sijaintia pelimaailmassa. Pelimaailman origosta ei ole mitään takeita, se saattaa olla jossain pelimaailman keskellä, nurkassa tai ei missään erityisessä paikassa. Jos edellä mainitun objektin yrittäisi renderöidä käyttäen suoraan kuution koordinaatteja se renderöitäisiin täysin väärään paikkaan. Tämän vuoksi kuution esityksen pisteitä on siirrettävä oikeaan paikkaan. Tätä kutsutaan translaatioksi (*eng.* translation) ja se on eräänlainen transformaatio.

Vaikka kuutio saataisiin renderöityä oikeaan paikkaan, se ei välttämättä ole silti täysin oikein. Objekti todennäköisesti ei ole täysin suorassa minkään koordinaattiakselin kanssa, eli sitä on käännetty. Tämän käännön on myös heijastuttava kuution pisteisiin. Tämän lisäksi objekti ei välttämättä ole yksikkökokoinen, eli kuution pisteiden välejä on muutettava, skaalattava. Kääntö ja skaalaus ovat molemmat transformaatioita.

Jotta tämän konseptuaalisen kuution saisi renderöityä näytölle on edellä käsiteltyä konkretisoitava. Kuution pisteet esitetään vektorimuodossa käyttäen kolmiulotteisia vektoreita. Transformaatiot voi esittää joko vektorien ja vektoriyhteenlaskun muodossa, kuten translaation, tai matriisien ja matriisi-vektori-kertolaskun muodossa, kuten kääntö ja skaalaus. [SRH16] Luvussa 4 esitetyt algoritmit tulevatkin nyt hyvään käyttöön. Lisäksi translaation, käännön ja skaalauksen voi yhdistää yhteen 4×4 matriisiin käyttäen sivulla 13 mainittua menetelmää: homogeenista koordinaattijärjestelmää. [Len12] Transformaatioita esittävää matriisia kutsutaan transformaatiomatriisiksi.

Tässä kohdassa on huomattava, että matriisi-vektori-kertolasku operoi vain yhdellä vektorilla kerralla ja kuution eri pisteitä esittää usea vektori. Kahdeksan pisteen kuution renderöiminen vaatisi kahdeksan matriisi-vektori-kertolaskua transformaatiomatriisin muodostamiseen vaadittavien laskujen lisäksi. Onneksi jokainen piste voidaan muuntaa saman transformaatiomatriisin avulla, joten transformaatiomatriisi tarvitsee muodostaa vain kerran yhtä objektia kohden. Tämä kuitenkin jättää kahdeksan matriisi-vektori-kertolaskua, joiden määrää ei voi vähentää, sillä jokainen piste on erillinen. Kahdeksan kertolaskua ei vaikuta kovin suurelta määrältä, eikä se olekaan. Ongelmaksi koituu kun objektia esittävä 3D-malli (*eng.* 3D model) sisältää kymmeniä tai satoja tuhansia pisteitä, kuten modernien pelien 3D-malleilla on tapana.[SRH16]

Prosessorilla kestää jo kauan laskea jokainen kertolasku yhden objektin 3D-mallin pisteille ja lähes poikkeuksetta jokainen peli sisältää enemmän kuin yhden renderöitävän objektin, olivat ne sitten samaa tai eri 3D-mallia käyttävät objektit. Tämä moninkertaistaa kertolaskujen määrän ja lisäksi jokaiselle objektille on luotava oma transformaatiomatriisi. Tähän tehtävään luotiin näytönohjaimet, joiden prosessorit ovat erikoistettu satojen tuhansien, jopa miljoonien, laskutoimitusten laskemiseen sekunnissa.

5.2 Rinnakkaistaminen

Objektien renderöintiin vaadittavat sadat tuhannet kertolaskut ovat jokainen toisistaan erillisiä, tarkoittaen, että yhtä pistettä koskeva kertolasku ei vaikuta muiden pisteiden kertolaskuun, sillä jokainen kerrottava piste tuottaa yhden uuden pisteen joka on transformoitu piste. Kuvitellen, että transformaatio on funktio joka ottaa syötteeksi satoja tuhansia pisteitä, se antaa myös tulokseksi tismalleen saman määrän pisteitä ja samassa järjestyksessä.

Peräkkäisten laskutoimitusten sijan nämä sadat tuhannet laskutoimitukset voi toteuttaa rinnakkain, juurikin siitä syystä, että yhden tulos ei vaikuta muiden laskujen tuloksiin. Keskusprosessorilla rinnakkaistaminen usein tarkoittaa muutaman, kahdesta kymmeneen, säikeen samanaikaista suoritusta. Nämä säikeet kuitenkin voivat toteuttaa mitä vaan laskuja ja muistin manipulointeja. Näytönohjaimen grafiikkaprosessori puolestaan on kehitetty rinnakkaistamaan mahdollisimman paljon. Siinä missä keskusprosessorilla on usein yhdestä kahdeksaan ydintä, on grafiikkaprosessorilla satoja, jopa tuhansia ytimiä. [SRH16]

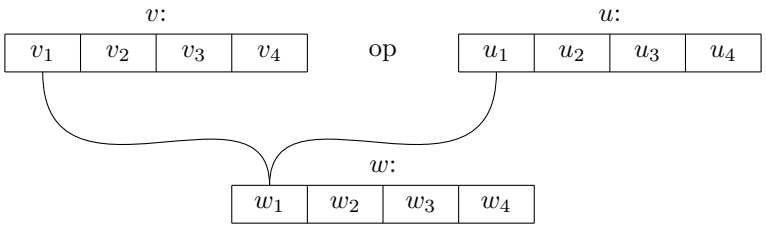
3D-mallin sisältämät pisteet vektorimuodossa syötetään näytönohjaimelle transformaatiomatriisin kanssa, jossa ne talletetaan näytönohjaimen omaan muistiin, videomuistiin. [SRH16] Tämän jälkeen grafiikkaprosessorissa suoritetaan pieni ohjelma, joka luo jokaiselle ytimelle säikeen ja antaa näille myös tiedon mikä pätkä 3D-mallin pisteistä kuuluu millekin säikeelle. [SRH16] Säikeissä sitten suoritetaan pieni ohjelma joka laskee sille kuuluvan pistejonon jokaiselle pisteelle transformaation ja tallettaa tulospisteet esimerkiksi takaisin videomuistiin seuraavaa käsittelyä varten. [SRH16] Näitä näytönohjaimessa suoritettavia pieniä ohjelmia kutsutaan varjostimiksi (*eng.* shader). [SRH16]

Grafiikkaprosessorilla rinnakkaistaminen, kuten keskusprosessorillakin, ei ole maksuton operaatio, vaan vaatii jonkin verran aikaa. Edellä tätä kuvattiin pienenä ohjelmana joka valmistelee varjostimien suoritusympäristöä. Todellisuudessa tämä ohjelma saatetaan suorittaa grafiikkaprosessorissa tai se voidaan suorittaa keskusprosessorissa. [SRH16] Kummmassakin tapauksessa tämä pieni ohjelma on suoritettava, muuten varjostimet eivät voisi tehdä mitään, ja suoritukseen kuluu jonkin verran aikaa.

Toisin kuin keskusprosessorilla, grafiikkaprosessorilla on harvemmin tarvetta vaihtaa suoritettavaa ohjelmaa kesken suorituksen, joten varjostimien suoritus ei keskeydy normaalitapauksessa ennen kuin jokainen tulosarvo on saatu laskettua. Tämä tarjoaa erityisen hyvän ja tehokkaan ympäristön monimutkaisten tai usean 3D-mallin tai objektin renderöimiseksi. Kuten luvussa 5.1 mainittiin, usealla eri objektilla voi olla sama 3D-malli käytössä. Tämä sopii hyvin näytönohjaimelle, sillä yhdestä 3D-mallista saa usean eri objektin renderöityä eri transformaatiomatriiseja käyttämällä.

5.3 Prosessorin erikoistaminen

Yksi tekijä modernien prosessorien valtavassa suoritustehossa on prosessoreissa olevat erikoistetut käskyt samanaikaiselle laskennalle. AMD64-arkkitehtuuri sisältää niin kutsuttuja vektorilaskuja, joissa vektori tarkoittaa listaa numeroista. Nämä käskyt toteuttavat jonkin laskutoimituksen jokaiselle kahden vektorin luvulle samaan tyyliin kuin esimerkiksi algoritmi Y laskee kahden vektorin luvut yhteen: jokainen tulosvektorin alkio on lähtövektorien vastaavan alkion tulos.



Kuva 3: Vektorilasku $w = v$ op u, jossa w, v ja u vektoreita ja op jokin skalaarilaskutoimitus.

Kuva 3 havainnollistaa joitakin prosessorin vektorilaskutoimitusta. Näissä on siis huomattava, että osallistuvat vektorit ovat vain listoja luvuista, eivätkä välttämättä matemaattisia vektoreita, joita luvussa 3.1 käsiteltiin. Modernien prosessorien putkituksen ansiosta tälläiset käskyt mahdollistavat hyvin matalan tason rinnakkaistamista. Jokaista alkiota vastaavat laskutoimitukset eivät välttämättä tapahdu samanaikaisesti, mutta jokainen on valmis käskyn suorituksen loputtua, joten on mahdollista antaa prosessorille käsky laskea kahden nelialkioisen vektorin alkiokohtainen tulo ja toteuttaa muita käskyjä sillävälin kun prosessorin liukulukupiirit toteuttavat laskutoimituskäskyä.

Tässä tutkielmassa käsiteltiin paljon sellaisia algoritmeja, joissa laskutoimitukset koostuivat kertolaskusta ja yhteenlaskusta. Erityisesti algoritmi K, jossa jokaisen tulosmatriisin alkiota laskettaessa tarvitsee laskea 4 kertolaskua ja yhteenlaskua, olettaen 4×4 matriisit. Kertolasku-yhteenlasku onkin erittäin yleinen yhdistelmä laskutoimituksia ja se tunnetaankin myös nimellä SAXPY (Scalar A X Plus Y). AMD64-arkkitehtuuriakin suunnitellessa tämän operaation yleinen käyttö tunnettiin ja sitä varten kehitettiin prosessorin käsky nimeltä `VFMADDPS` (Multiply and Add Packed Single-Precision Floating Point). [Adv20, s.617] Tämä on vain yksi useasta, samankaltaisesta, käskystä jotka operoivat vektorimuodossa olevilla liuku-luvuilla.

Näiden käskyjen avulla on mahdollista tiivistää tässä tutkielmassa käsitellyt algoritmit vain muutamien prosessorin käskyjen mittaisiksi. Tämä on yksi syy miksi näihin algoritmeihin kuluvaa aikaa on hankala mitata täsmällisesti, sillä jokainen CISC-prosessori toteuttaa nämä käskyt eri tavalla. Osa laskee samanaikaisesti jokaisen alkion laskutoimituksen ja tulos on valmis yhteen laskutoimitukseen kuluvassa ajassa, toiset puolestaan laskevat jokaisen erillisen laskutoimituksen erikseen. Ainoa yhteinen tekijä on, että laskeminen suoritettaan samalla käskyllä ja lopputulos on sama.

Tässä tutkielmassa on myös jokaisen algoritmin kohdalla käsitelty ilmenevää laskuvirhettä. Koska laskuvirhe on luontainen liukulukulaskuille, vaikuttaa se erikoistuneisimpiinkin prosessoreihin. On kuitenkin mahdollista vaikuttaa virheen kumulaatioon erilaisilla pyöristysmenetelmillä. Satunnaistamalla liukulukulaskujen pyöristämistä on mahdollista saavuttaa paljon pienempi virheiden kumulointi kuin esimerkiksi pyöristämällä lähimpään lukuun. [CG20] Tämän lisäksi sa-

tunnaistamisella on mahdollista saavuttaa kelvollinen tarkkuus jopa 16-bittisillä liukuluvuilla. [CG20]

6 Kvaterniot

Matriisit tarjoavat paljon mahdollisuuksia, mutta vastapainoksi niillä on myös rajoituksia. Esimerkiksi kahden neliömatriisin kertolasku on tunnetusti vaativa onglema. Tämän lisäksi esimerkiksi rotaatiotransformaatioiden käsittely matriiseilla saattaa koitua hankalaksi. Tässä luvussa tutustutaan kvaternioihin, jotka tarjoavat rotaatioille vaihtoehtoisen laskentatavan ja helpottavat niiden käsittelyä.

6.1 Kvaternioiden matematiikkaa

Kvaterniot ovat sellaisia lukuja, jotka kompleksilukujen tavoin sisältävät reaaliosan ja imaginääriosan, mutta kompleksiluvuista poiketen kvaternioilla on kolme imaginääriosaa. Kvaternioita usein esitetään muodossa

$$q = s + xi + yj + zk \tag{19}$$

jossa i, j ja k ovat sellaiset yksiköt jotka täyttävät kaavassa 20 olevat ehdot. [Vin11]

$$\begin{aligned} i^2 = j^2 = k^2 = ijk = -1 \\ ij = -ji = k \\ jk = -kj = i \\ ki = -ik = j \end{aligned} \tag{20}$$

Kvaternioita voi ajatella skalaari-vektori -parina siten, että kvaternion reaaliosoa on skalaari ja imaginääriosat muodostavat kolmiulotteisen vektorin. [Vin11]

$$q = [s, \mathbf{v}] \tag{21}$$

Kahdelle kvaterniolle $q_a, q_b \in \mathbb{H}$ on määritelty yhteen- ja vähennyslaskut (kaava 22) sekä kertolaskut (kaava 23). [Vin11]

$$q_a + q_b = s_a + s_b + (x_a + x_b)i + (y_a + y_b)j + (z_a + z_b)k \tag{22}$$

$$\begin{aligned} q_a q_b = \quad & (s_a s_b - x_a x_b - y_a y_b - z_a z_b) \\ & + (s_a x_b + x_a s_b + y_a z_b - z_a y_b)i \\ & + (s_a y_b + x_a z_b + y_a s_b - z_a x_b)j \\ & + (s_a z_b + x_a y_b + y_a x_b - z_a s_b)k \end{aligned} \tag{23}$$

Näiden lisäksi kvaternioille on myös määritelty konjugaatti: kvaternion $q = [s, \mathbf{v}]$ konjugaatti on $\bar{q} = [s, -\mathbf{v}]$. Kvaternion ja sen konjugaatin välinen tulo on sen normin (kaava 24) neliö, kuvattu kaavassa 25. [Vin11]

$$\|q\| = \sqrt{s^2 + x^2 + y^2 + z^2} \tag{24}$$

$$q\bar{q} = \bar{q}q = \|q\|^2 \tag{25}$$

Kvaternioille on myös määritelty kaavan 26 mukaisesti käänteiskvaternio käyttäen konjugaattia ja normia. [Vin11, s.65] Käänteiskvaternioille pätee kaavassa 27 esitetty ominaisuus. [Vin11, s.65]

$$q^{-1} = \frac{\bar{q}}{\|q\|^2} \tag{26}$$

$$qq^{-1} = [1, \mathbf{0}] = 1 + 0i + 0j + 0k = 1 \tag{27}$$

Kvaternio, jonka normi on 1 on yksikkökvaternio ja kvaternio jolla on $s = 0$ on puhdas kvaternio. Kvaternioiden matematiikkaan syventyminen on tämän tutkielman aiheen ulkopuolella. Kuitenkin on hyvä huomata, että kvaternioita laskiessa ne eivät käyttäydy tavallisten vektoreiden kanssa yhtenevästi. Kahden kvaternion tulo on uusi kvaternio ja kahden puhtaan kvaternion tulo ei ole puhdas kvaternio.

Luvussa 6.2 käsiteltävä kvaternioiden rotaatio on mahdollista esittää matriisimuodossa. Jonkin rotaatiota esittävän kvaternion q matriisimuodon $\mathbf{R}_q$ saa kaavasta 28. [Len12, s.86]

$$\mathbf{R}_q = \begin{bmatrix} 1 - 2y^2 - 2z^2 & 2xy - 2sz & 2xz + 2sy \\ 2xy + 2sz & 1 - 2x^2 - 2z^2 & 2yz - 2sx \\ 2xz - 2sy & 2yz + 2sx & 1 - 2x^2 - 2y^2 \end{bmatrix} \tag{28}$$

6.2 Kvaternioiden käyttö

Kvaternioita voi esittää tietokoneessa neliulotteisen vektorin tavoin. Kvaternion i, j ja k yksiköitä ei tarvitse esittää, sillä kaikki laskutoimitukset on toteutettavissa implisiittisillä imaginääriyksiköillä. Esimerkiksi kertolaskussa jokaisen komponentin voi laskea suoraan tukeutumalla kaavaan (23). Luvussa 4 esiintyvien algoritmien tavoin, kvaterniolaskuissakin esiintyy paljon kerto- ja yhteenlaskuja; saxpy-operaatioita.

Kahden kvaternion välinen kertolasku vaatii 16 saxpy-operaatiota ja kahden 3×3 matriisin kertolasku vaatii 27 saxpy-operaatiota. [Len12, s.84] Sekä kvaternioita, että matriiseja on mahdollista käyttää rotaatioiden esittämiseen ja koska usein yhtä renderöitävää objektia kohden tarvitaan usean rotaation yhdiste, on kertolaskuihin kuluva operaatioiden määrä varsin merkittävä. Esimerkiksi Euler-kulmien käyttö vaatii kolme rotaatiota, jolloin tarvitaan kolme matriisia tai kvaterniota esittämään jokaista rotaatiota ja lopullisen rotaation muodostamiseksi tarvitaan kolme kertolaskua. Jo Euler-kulmien käytössä huomataan, että $16 \cdot 3 = 48 < 81 = 27 \cdot 3$, eli kvaternioiden kertolaskuun tarvitaan noin 60 % matriisien kertolaskuihin kuluvista saxpy-operaatioista.

6.2.1 Rotaatiot

Pisteen rotaatio on mahdollista laskea käyttämällä kvaterniota tai matriisia. Kuitenkin matriisien kanssa tarvitsee laskea vain yksi matriisi-vektori-kertolasku kaavan (29) esittämällä tavalla, kun puolestaan kvaterniot vaativat kaksi kertolaskua ja piste on esitettävä kvaterniona, kaavan (30)

esittämällä tavalla.

$$\mathbf{p}' = \mathbf{R}\mathbf{p} \tag{29}$$

$$p' = \hat{q}p\hat{q}^{-1} \tag{30}$$

jossa $\mathbf{R}$ on rotaatiomatriisi, $\hat{q}$ vastaavan rotaatiokvaternion yksikkökvaternio, $\hat{q}^{-1}$ sen käänteiskvaternio, p pisteen $\mathbf{p}$ kvaternioesitys ja p' tavoitepisteen $\mathbf{p}'$ kvaternioesitys. [Len12, Vin11]

Pisteen voi esittää kvaterniona asettamalla kvaternion vektoriosaksi pistettä esittävän vektorin ja skalaariosaksi nollan: $p = [0, \mathbf{p}]$. [Vin11] Tietokoneella kaavan (30) toteuttamiseen voi käyttää pieniä oikoreittejä jättämällä pois pisteen $\mathbf{p}$ muuton kvaternioksi, jolloin tulokseksi saadaan suoraan vektori ja säästytään muutamalta $a \cdot 0$ kertolaskulta. Kuitenkin näitä oikoreittejä hyödyntämällä kvaternioiden käyttö pisteiden rotaatiotransformaatiossa on matriiseja hitaampaa, johtuen kahdesta tarvittavasta kertolaskusta.

Kirjoittajan suorittama 10 000 000 pisteen transformaation vertailu tuotti taulukossa 4 näkyvät tulokset. Tulokset saatiin kirjoittamalla C++-ohjelma, joka loi 10 000 000 satunaista pistettä kuution $[0, 1] \times [0, 1] \times [0, 1]$ sisältä ja mittaamalla kulunut aika jokaisen pisteen rotaatiotransformaation toteuttamisessa. Tulokset mitattiin käyttäen kolmea eri rotaatiota, joita vastaavat matriisit ja kvaterniot luotiin valmiiksi ennen ajanmittauksen aloittamista. Näiden kolmen eri rotaatiota käsittelevät mittaukset toistettiin kolme kertaa, eli jokaisella menetelmällä mitattiin yhteensä yhdeksän kertaa 10 000 000 pisteen transformointiin kuluva aika. Taulukossa on esitetty näiden yhdeksän mittauksen keskiarvo.

Taulukossa 5 on esitetty eri transformaatioita kuvaavien kvaternioiden ja matriisien muodostamista. Tämä vertailu toteutettiin muodostamalla kaksi, kolme tai kymmenen vastaavaa rotaatiota kuvaavaa kvaterniota ja matriisia ja niiden yhdistettä kuvaavan matriisin muodostamiseen kuluvaa aikaa mitattiin. Matriisien kohdalla kyse oli matriisien kumuloiva yhteen kertominen, kun taas kvaternioilla yhteen kertomisen lisäksi piti muodostaa lopullista transformaatiota kuvaava matriisi. Kumpikin vertailu toteutettiin käyttämällä samoja laskufunktioita ja vertaileva C++-ohjelma käännettiin GNU C++ -kääntäjällä käyttäen tason kaksi optimointia.

Laskutapa	**Kesto**
Kvaterniot ilman optimointeja	1.53072 s
Optimoimalla käänteiskvaternio	0.54890 s
Erikoistamalla kertolasku	0.52310 s
Matriisilaskuilla	0.12941 s

Kuva 4: Kvaterniokertolaskujen ja matriisi-vektori-kertolaskun ero laskenta-ajassa.

		Rotaatioita		
		2	3	10
Laskutapa	Kvaterniot	394 ns	526 ns	1099 ns
	Matriisit	423 ns	766 ns	1647 ns

Kuva 5: Samoja rotaatioita kuvaavien kvaternioiden ja matriisien yhteen kertomisen ero laskenta-ajassa.

Koska kvaternioilla laskettava pisteen transformaatio vaatii kaksi kvaterniokertolaskua ja koska kvaterniokertolasku on noin yhtä vaativa kuin matriisi-vektori-kertolasku, vaatien 16 kertolas-

kua ja 16 yhteenlaskua, eli 16 saxpy-operaatiota, on kvaternioiden käyttö pisteen transformaation laskemisessa merkittävästi hitaampaa kuin matriisien. Taulukon 4 lukemat antavat kvaterniomenetelmälle yli nelinkertaisen suoritusajan matriisimenetelmään verrattuna. Ensimmäisessä laskutavassa käänteiskvaternio laskettiin jokaisen pisteen mukana täydellisyyden vuoksi, ja on siten moninkertaisesti hitaampi kuin laskemalla käänteiskvaternio vain kerran. Tämä laskutapa on toteutettu siitä syystä, että kvaternioita käyttäessä pisteen transformointiin tarvitaan sekä rotaatiota kuvaava kvaternio, että sen käänteiskvaternio. Käänteiskvaternion laskemiseksi tarvitsee laskea kvaternion normi ja jakaa jokainen konjugaatin alkio normilla, kuten kaavassa (26) on esitetty.

Kuitenkin rotaatioiden yhdistäminen on kvaternioilla merkittävästi nopeampaa kuin matriiseilla. Jo kahden kvaternion yhteen kertominen ja matriisiksi muuttaminen suoraan kaavan (28) avulla on nopeampaa kuin kahden 3×3 matriisin kertominen yhteen. Rotaatiokvaternion ja -matriisin muodostaminen on merkittävä osa transformaatioiden käyttöä. Kaavassa (31) on mielivaltaisen akselin $\mathbf{v}$ ympäri tehdyn kulman θ suuruisen rotaation matriisi. [Len12, s.75] Kaavassa (32) on vastaavan rotaatiokvaternion muodostus. [Vin11, s.89]

$$\mathbf{R}_{\mathbf{v}}(\theta) = \begin{bmatrix} \cos\theta + (1-\cos\theta)v_x^2 & (1-\cos\theta)v_x v_y - \sin\theta v_z & (1-\cos\theta)v_x v_z + \sin\theta v_y \\ (1-\cos\theta)v_x v_y + \sin\theta v_z & \cos\theta + (1-\cos\theta)v_y^2 & (1-\cos\theta)v_y v_z - \sin\theta v_x \\ (1-\cos\theta)v_x v_z - \sin\theta v_y & (1-\cos\theta)v_y v_z + \sin\theta v_x & \cos\theta + (1-\cos\theta)v_z^2 \end{bmatrix} \tag{31}$$

$$q_{\mathbf{v}}(\theta) = [\cos\frac{\theta}{2}, \sin\frac{\theta}{2}\mathbf{v}] = \cos\frac{\theta}{2} + \sin\frac{\theta}{2}v_x i + \sin\frac{\theta}{2}v_y j + \sin\frac{\theta}{2}v_z k \tag{32}$$

Selvästi nähdään, että (32) sisältää vähemmän laskutoimituksia kuin (31) ja kummassakin tapauksessa saadaan vastaavaa rotaatiota kuvaava kvaternio tai matriisi. Näiden kahden yhtenevyyden näkee helposti soveltamalla kaavaa (28) rotaatiokvaternioon $q_{\mathbf{v}}(\theta)$. Kvaternioilla on paljon käyttöä erityisesti sellaisissa renderöintiympäristöissä, joissa jokaiselle objektille on laskettava paljon rotaatioita, kuten esimerkiksi lentosimulaatioissa. [Vin11]

6.2.2 Virheet

Luvussa 4.3 esiintyi kahden matriisin kertolaskun tuottama liukulukuvirhe, joka on sekä kumulatiivinen, että pysyvä. Taulukko 2 antoi 4×4 matriisien kertolaskulle $8\epsilon_m$ suuruisen termikohtaisen virheen, jonka vaikutus moninkertaistui myöhemmissä kertolaskuissa. Koska kvaternioita esitetään myös liukuluvuilla, on laskuista ilmenevää virhettä myös harkittava. Kahden kvaternion kertolasku tuottaa saman $8\epsilon_m$ suuruisen virheen, mikä on enemmän kuin 3×3 matriisin kertolaskusta ilmenevä $6\epsilon_m$. Kvaternion kuvaamiseen tarvitaan vain neljä lukua, mutta jokainen näistä luvuista esiintyy jokaisessa kertolaskun termissä. Tästä syystä kvaterniot ovat yhtä alttiita virheille kuin 4×4 matriisit.

6.2.3 Interpolointi

Kvaternioiden rotatiivisten ominaisuuksien lisäksi niillä on eräs toinenkin ominaisuus ja etu matriiseihin verrattuna. Animoinnissa on olemassa käsite avainkehyksistä (*eng.* keyframe) ja erityisemmin avainkohdista. Määrittelemällä tietyt pisteet joiden välillä pisteiden sijainti interpoloidaan on paljon muistiystävällisempi ja laajennettavampi menetelmä kuin jokaisen mahdollisen

animaatiossa olevan asennon pisteen tallentaminen. [Bob98] Esimerkiksi monissa peleissä kameran sijainti ja suunta interpoloidaan kahden pisteen välillä. [Bob98] Kvaternioilla on erityistä käyttöä kun halutaan interpoloida kaarevasti kahden pisteen välillä. Pallomainen lineaarinen interpolointi (*eng.* spherical linear interpolation) on yksi loistava käyttökohde kvaternioille. [Len12, s.86]

Kahden kvaternion q_1, q_2 välille saatava pallomainen lineaarinen interpolointi saadaan käyttämällä kaavaa (33), jossa kaavojen (34) ja (35) sijoitukset ovat yhtäpitävät. [Len12, s.89] Tätä menetelmää käytettiin esimerkiksi pelissä Tomb Raider (1997). [Bob98]

$$q(t) = \frac{\sin\theta(1-t)}{\sin\theta} q_1 + \frac{\sin\theta t}{\sin\theta} q_2 \tag{33}$$

$$\theta = \cos^{-1}(q_1 \cdot q_2) \tag{34}$$

$$\sin\theta = \sqrt{1 - (q_1 \cdot q_2)^2} \tag{35}$$

7 Yhteenveto

Kolmiulotteiseen renderöintiin sisältyy paljon algoritmeja, joista vain muutamat ja ilmeisimmät ovat tämän tutkielman kohteena. Vektori- ja matriisilaskut ovat merkittävä osa kolmiulotteista renderöintiä ja siksi niiden ominaisuuksiin paneuduttiinkin syvästi. Kuitenkin luvussa 5 huomattiin, että käytännössä monet operaatiot ovatkin jo laitteiston tukemia ja täten hyvin optimoituja. Tämän lisäksi luvussa 6 käsiteltiin konsepteja, jotka tehostivat esimerkiksi rotaatiotransformaatioiden laskemista. Jokaisen algoritmin kohdalla käsiteltiin myös liukulukulaskuissa esiintyvää virhetermiä ja miten suuresti se vaikuttaa.

Luvussa 5 huomattiin, että monet luvussa 4 tehdyt oletukset ovat liian yleisiä käytännössä. Esimerkiksi aikakompleksisuuksilla ei ole merkitystä ulottuvuuksien määrän ollessa kiinnitetty. Modernit prosessorit jopa tarjoavat suoraan käskyjä nopeuttamaan ja tiivistämään käsiteltyjen algoritmien suoritusta ja näytönohjaimet on suunniteltu suorittamaan tuhansia laskuja samanaikaisesti. Kuitenkin laitteistotuesta huolimatta kvaterniot osoittautuivat matriiseja tehokkaammiksi rotaatioiden käsittelyssä, kun taas tunnetut ja tehokkaat matriisien kertolaskualgoritmit, kuten Strassen algoritmi, osoittautuivat suoraa algoritmia hitaammiksi renderöinnissä käytettävillä matriiseilla.

Kuten ohjelmoinnissa yleisestikin, yhteen ongelmaan löytyy monia eri ratkaisuja jotka kaikki käyttäytyvät eri tavalla. Tämän vuoksi onkin tärkeää tuntea useita eri ratkaisuja ja niiden hyviä ja huonoja puolia, sillä harvemmin löytyy yhtä menetelmää joka on paras kaikissa tapauksissa. Tässä tutkielmassa juurikin käytiin läpi kolmiulotteisen renderöinnin tukipilareita, matriisilaskuja, ja tutkittiin mitä ominaisuuksia milläkin laskulla on. Tämän lisäksi modernien laitteiden erikoistamisen vaikutuksia käsiteltiin ja lopussa oli tutustuminen yhteen, mahdollisesti yllättävään, vaihtoehtoiseen menetelmään.

Viitteet

[Adv20] Advanced Micro Devices. *AMD64 Architecture Programmer's Manual Volume 4: 128-Bit and 256-Bit Media Instructions*, 2020.

[bad18] bad_boot. Bowser in the fire sea in 0x a presses? (new glitch explanation), 2018. `https://www.youtube.com/watch?v=MFxJuq3FRgI` (Viitattu 14.11.2020).

[Bob98] Nick Bobic. Rotating objects using quaternions. *Gamasutra*, 1998. `https://www.gamasutra.com/view/feature/131686/rotating_objects_using_quaternions.php`.

[CG20] Matteo Croci and Michael Bryce Giles. Effects of round-to-nearest and stochastic rounding in the numerical solution of the heat equation in low precision, 2020.

[CHN99] Marius Cornea-Hasegan and Bob Norin. Ia-64 floating point operations and the ieee standard for binary floating-point arithmetic. *Intel Technology Journal*, Q4, 1999.

[God19] Matt Godbolt. Keynote: What everyone should know about how amazing compilers are - matt godbolt [c++ on sea 2019], 2019. `https://www.youtube.com/watch?v=w0sz5WbS5AM` (Viitattu 14.11.2020).

[GZ14] Carlo Andrea Gonano and Riccardo Enrico Zich. Cross product in n dimensions - the doublewedge product. 2014.

[iee19] Ieee standard for floating-point arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, 2019.

[Knu98] Donald Knuth. *The Art of Computer Programming*, volume 2: Seminumerical Algorithms. Addison Wesley, 3rd edition, 1998.

[Knu05] Donald Knuth. *The Art of Computer Programming*, volume 1 Fascicle 1: MMIX A RISC Computer for the New Millenium. Addison Wesley, 2005.

[Len12] Eric Lengyel. *Mathematics for 3D Game Programming and Computer Graphics*. Course Technology, a part of Cengage Learning, 2012.

[Nin96] Nintendo. Super mario 64, 1996. Videopeli.

[PTVF07] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes*. Cambridge University Press, 3rd edition, 2007.

[Ski12] Steven Skiena. *The Algoritm Design Manual*. Springer, 2012.

[SRH16] Graham Sellers, Jr. Richard S. Wright, and Nicholas Haemel. *OpenGL Superbible*. Addison Wesley, 7th edition, 2016.

[Vin11] John Vince. *Quaternions for Computer Graphics*. 2011.

A Geneerisen funktion erillinen toteutus

Käsitellään seuraavaa lyhyttä C++ ohjelmaa, jossa on D-ulotteinen vektoriluokka ja näillä operoiva geneerinen funktio:

```
template <int D>
struct vector {
    float data[D];
};

template <int N>
float function(const vector<N>& v) {
    float a;
    for (int i = 0; i < N; ++i)
        a += v.data[i];
    return a;
}

int main() {
    vector<3> vec1{1.0f, 2.34f, 5.254f};
    vector<4> vec2{1.3f, 1.35f, 3.234f, 3.14f};
    float b = function<3>(vec1);
    float x = function<4>(vec2);
    return b+x;
}
```

GCC 10.2 kääntää tässä koodissa olevan funktion `function()` seuraavanlaisiksi kahdeksi alirutiiniksi, kun optimointitaso on asetettu nollaksi.

```
float function<3>(vector<3> const&):
                pushq   %rbp
                movq    %rsp, %rbp
                movq    %rdi, -24(%rbp)
                movl    $0, -8(%rbp)
                jmp     .L4
.L5:
                movq    -24(%rbp), %rax
                movl    -8(%rbp), %edx
                movslq  %edx, %rdx
                movss   (%rax,%rdx,4), %xmm0
                movss   -4(%rbp), %xmm1
                addss   %xmm1, %xmm0
                movss   %xmm0, -4(%rbp)
                addl    $1, -8(%rbp)
.L4:
                cmpl    $2, -8(%rbp)
                jle     .L5
                movss   -4(%rbp), %xmm0
                popq    %rbp
                ret
```

```
float function<4>(vector<4> const&):
                pushq   %rbp
                movq    %rsp, %rbp
                movq    %rdi, -24(%rbp)
                movl    $0, -8(%rbp)
                jmp     .L8
.L9:
                movq    -24(%rbp), %rax
                movl    -8(%rbp), %edx
                movslq  %edx, %rdx
                movss   (%rax,%rdx,4), %xmm0
                movss   -4(%rbp), %xmm1
                addss   %xmm1, %xmm0
                movss   %xmm0, -4(%rbp)
                addl    $1, -8(%rbp)
.L8:
                cmpl    $3, -8(%rbp)
                jle     .L9
                movss   -4(%rbp), %xmm0
                popq    %rbp
                ret
```

Kääntäjä on erikoistanut geneerisen D-ulotteisen syötevektorin ottavan funktion kahdeksi erikoistetuksi alirutiiniksi, joista toinen ottaa 3-ulotteisen vektorin ja toinen 4-ulotteisen vektorin.