Point Animation

pluginin kehitys Messiah-ohjelmalle

Viestintä
3d-animaatio ja -visualisointi
Opinnäytetyö
6.6.2010

Pekka Kytölä

Metropolia

Metropolia

Kulttuurialat

TIIVISTELMÄSIVU

Koulutusohjelma	Suuntautumisvaihtoehto
Viestintä	3d-animointi ja -visualisointi

Tekijä
Pekka Kytölä

Työn nimi
Point Animation

Työn ohjaaja/ohjaajat
Jaro Lehtonen

Työn laji	Aika	Numeroidut sivut + liitteiden sivut
Opinnäytetyö	06.06.2010	48

TIIVISTELMÄ

Opinnäytetyönä tehtiin Point Animation -plugin Messiah-ohjelmaan. Point Animation mahdollistaa käyttäjäystävällisen verteksien manipuloinnin ja animoinnin. 3D-mallia voidaan muokata tarvittaessa joka kuvassa, jolloin väliin jääneet kuvat interpoloidaan animaation sulavuuden varmistamiseksi.

Point Animationin kehityksessä suurin ongelma oli verteksien liikkeiden relatiivisuus; erilaisten deformereiden vaikutusta mallin suuntaan ja sijaintiin on käytännössä mahdotonta ennakoida. Tähän ongelmaan kirjoittaja ehdottaa ratkaisua, jonka periaate pohjautuu teksturoinnissa käytettyyn tangetspace-tekniikkaan. Tangentspace-tekniikan hahmottamiseksi työssä käsitellään 3d-grafiikassa käytettäviä matriiseja.

Point Animation julkaistiin osana Messiah-ohjelmaa. Sitä on käytetty kiireisen aikataulun mainosproduktioissa tehokkaaseen virheiden korjaamiseen animaatiosta, jossa perinteisesti on käytetty morph-tekniikkaa. Point Animationin avulla voidaan luoda uusi kerros verteksianimaatiota entisen päälle ja se täydentää hyvin luilla tai morph-tekniikalla toteutettua animaatiota.

Teos/Esitys/Produktio

Säilytyspaikka
Metropolia Ammattikorkeakoulun kirjasto, Tikkurilan toimipiste

Avainsanat
Point Animation, Messiah, plugin, verteksi, animaatio, tangentspace, matriisi, morph

Metropolia

Culture

Degree Programme in	Specialisation
Media	3D Animation and Visualisation

Author
Pekka Kytölä

Title
Point Animation – Developing a Plugin for Messiah

Tutor(s)
Jaro Lehtonen

Type of Work	Date	Number of pages + appendices
Thesis	6.6.2010	48

The present thesis aimed at developing a plugin suitable for the intuitive vertex manipulation and animation. The essential objective was to enable the editing of 3d models in desired frames, while also performing spline interpolation for the remainder of the frames or in-betweens.

The end product of the thesis was programmed in C language as a plugin for Messiah animation software. Point Animation evolved to its present form within three cycles of research and development.

As different deformers modify the form, location and orientation of the 3d models in diverse fashion, the relativity of a vertex proved to be a concept of high complexity. The author of this thesis proposes building a tangent space coordinate system that has its roots in texturing and normal mapping. The nature of matrices is explored to gain full understanding of the tangent space.

Tangent space provides a logical and intuitive coordinate system wherein vertex animation can be constituted. Point Animation demonstrated itself to be a good supplementary tool that can produce an additional layer of vertex animation on top of the ordinary deformers like skeletal systems and morph targets.

Work / Performance / Project

Place of Storage
Metropolia Ammattikorkeakoulun kirjasto, Tikkurilan toimipiste

Keywords
Point Animation, Messiah, plugin, vertex, animation, tangent space, matrix, morph

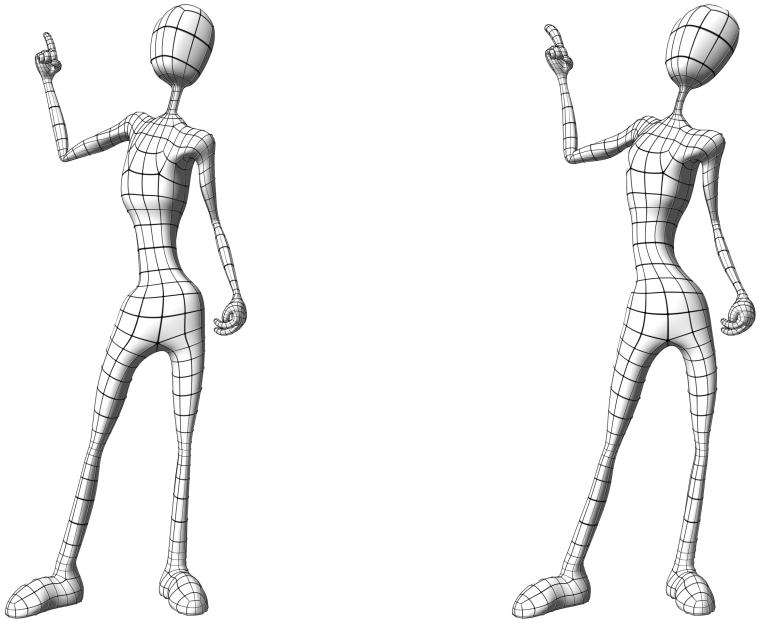
Sisällys

1 JOHDANTO....2
2 VERTEKSIANIMAATIO....3
2.1 Morphit....3
2.2 Klusterit....5
2.3 AniSculpt....6
3 MESSIAH....7
3.1 Filosofia....8
3.2 SDK....9
4 POINT ANIMATION....10
5 RELATIIVISUUS....13
5.1 Matriisit 3d-grafiikassa....15
5.2 Tangent space....27
6 YHTEENVETO....47
LÄHTEET....48

1 JOHDANTO

Opinnäytetyö keskittyy Point Animation -työkalun kehittämiseen, joka mahdollistaa mielekkään verteksien animoinnin. Kynän ja paperin sallima muotojen vapaus on työlästä ja teknisesti vaikeaa toteuttaa 3d-ohjelmistoilla. Yksittäisen kuvan muotojen muokkaus ei ole vielä ongelmallista, mutta useampien satojen, jopa tuhansien, peräkkäisten kuvien muokkaus paremman liikkeen illuusion luomiseksi on haastavaa. Perinteisen animaattorin on helppo piirtämällä määritellä hahmon muoto, venyttää ja likistää sitä tahtonsa mukaan, jopa räjäyttää. 3d-hahmon kohdalla se ei ole yhtä helppoa, sillä hahmossa on perinteisesti tietty määrä verteksejä, ja näiden verteksien täytyy seurata tiettyjä luita. Käytännössä 3d-animaattori ei voi hankkiutua eroon vertekseistä eikä taikoa niitä lisää.

Teoriassa hyvin suunniteltua hahmoa voidaan venyttää tarpeen mukaan, mutta käytännössä hahmon muoto ei kuitenkaan aina ole sitä, mitä halutaan. Usein tulee vastaan tilanne, jossa muutamaa verteksiä siirtämällä muodossa havaittu virhe saadaan korjattua. Jos hahmo on huonosti suunniteltu ja tällaisia ”muotovirheitä” esiintyy jatkuvasti, silloin tulee palata takaisin suunnitteluvaiheeseen. Usein toistuvien ongelmien korjailu jälkikäteen on hidasta ja turhauttavaa. Harvoin esiintyvien virheiden kohdalla asia on usein päinvastainen, muodon korjaaminen suunnittelupöydällä saattaa aiheuttaa paljon hankalampia ja useammin esiintyviä ongelmia.



Kuva 1. Malli ennen ja jälkeen Point Animationilla tehtyjä muutoksia

Harvoin toistuvien virheiden vaivaton korjaus muodossa, mallin verteksien muokkaus ja piirrosmaisemman muodon mahdollistaminen (kuva 1) olivat Point Animationin kehityksessä tärkeimmät vaikuttimet. Työkalu toteutettiin plugin-tyyppisenä Messiah-ohjelmalle.

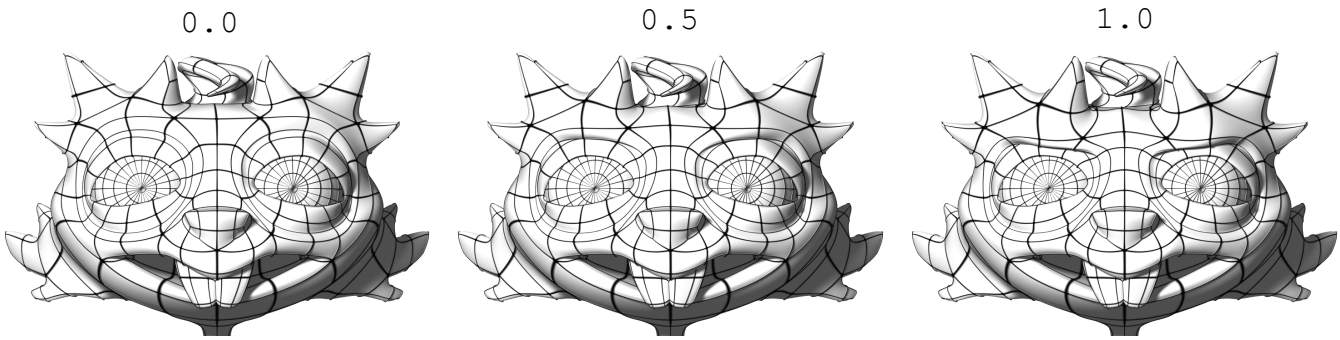
2 VERTEKSIANIMAATIO

Perinteisesti yksittäisten verteksien tarkka animointi on ollut mahdollista vain morph target -tekniikalla (morph) tai sitomalla yksi tai useampi verteksi toiseen objektiin, jota ohjelmasta riippuen kutsutaan eri nimellä, tässä työssä käytän nimitystä klusteri. Blenderin käyttäjille tuttu AniSculpt-tekniikka valjastaa Blenderin skulptaustyökalut animaatiotarkoituksiin. AniSculpt ei siis ole plugin, vaan työtapa, joka soveltuu uniikin verteksianimaation työstämiseen. Uniikilla tarkoitan tässä tapauksessa sitä, että jokaisessa animaation kuvassa muotoa voi olla korjattu tavalla, joka ei toistu minkään säännön mukaan.

2.1 Morphit

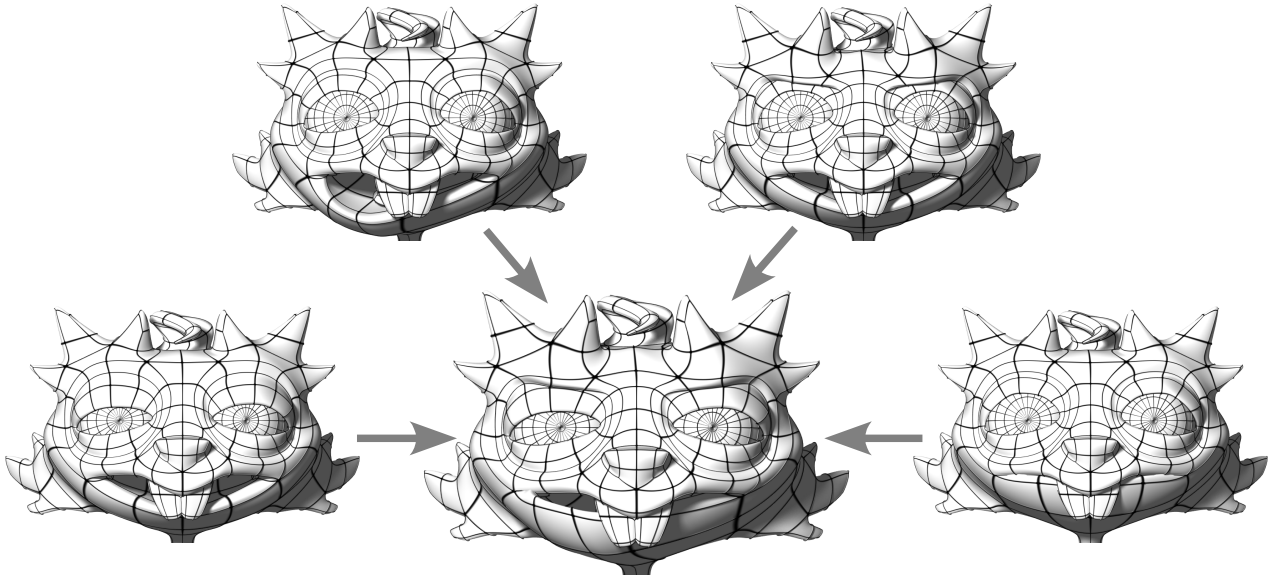
Morph (muuttua, muuttaa muotoa) on 3d-objektin muunnelma, jonka verteksien paikat voivat vaihdella. Sen muotoa on muokattu. Sulava muutos alkuperäisen objektin ja

morph-objektin välillä saadaan aikaan yksinkertaisella verteksien paikkojen interpoloinnilla (kuva 2). Tällaista interpolointia on laajalti käytetty kasvojen animointiin. (Pandzic & Forchheimer 2002, 67.)



Kuva 2. Kulmia muuttavan morphin vaikutus malliin

Morphit toimivat hyvin kasvojen animointiin ja muuhun verteksianimaatioon, joka toistuu usein. Morphit interpoloidaan, ennen kuin luut vaikuttavat vertekseihin, tästä syystä yksittäisen ongelman korjaaminen morphilla on usein kankeaa ja hankalaa. Morphin voi interpoloida absoluuttisesti, jolloin verteksit menevät määriteltyihin paikkoihin välittämättä siitä, missä ne sillä hetkellä ovat, mutta additiiviset tai relatiiviset morphit ovat paljon monikäyttöisempiä, sillä niitä voidaan pinota päällekkäin, jolloin niiden vaikutukset kumuloituvat (kuva 3). Hyvin suunnitelluilla additiivisilla morpheilla voidaan tehdä mm. kokonaisvaltaista kasvoanimaatiota.



Kuva 3. Erilaisten morphien yhteisvaikutus malliin

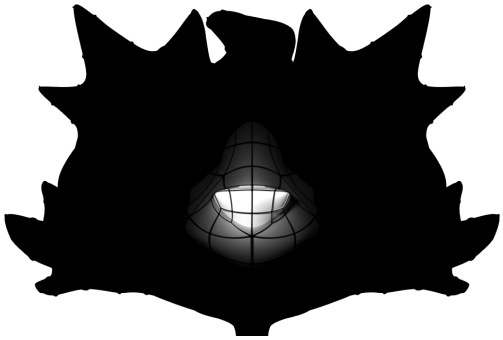
Yhdessä morphissa jokaisella verteksillä on 3d-vektori, joka määrittelee, mihin suuntaan verteksin täytyy liikkua alkuperäiseltä paikaltaan. Kertomalla vektorit skalaarilla voidaan morphia helposti interpoloida. Kun kyseessä on useampia morpheja, saadaan lopputulos laskemalla morphien skaalatut vektorit yhteen.

2.2 Klusterit

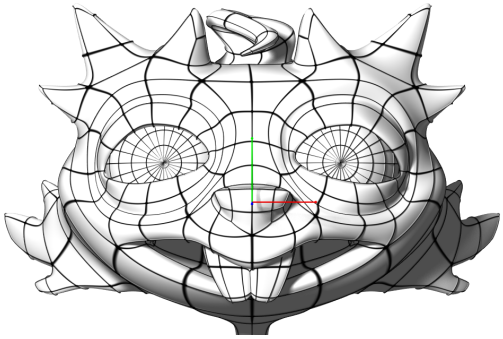
Klusterit mahdollistavat painotettujen verteksiryhmien määrittelyn. Kun klusteria liikutetaan tai pyöritetään, verteksit seuraavat klusteria määritellyn painoarvon verran. Klustereita voidaan käyttää sellaisenaan tai muiden muokkaustoimintojen rinnalla. (Meade & Arima 2004, 217.)

Käytännössä klusteri koostuu yhdestä animoitavasta elementistä ja listasta, jossa jokaista verteksiä kohden on jokin arvo, useimmiten väliltä 0–1. Elementillä tässä työssä tarkoitetaan 3d-ohjelmistoissa esiintyvää kaikista yksinkertaisinta animoitavaa ”kappaletta”, jolla on paikka, suunta ja skaala. Se, millä nimellä tällaista elementtiä kutsutaan, vaihtelee riippuen ohjelmistosta. Yleiseimmät nimet ovat Null (Softimage, LightWave 3D ja Messiah), Dummy (3D Max), Empty (Blender) ja Group (Maya).

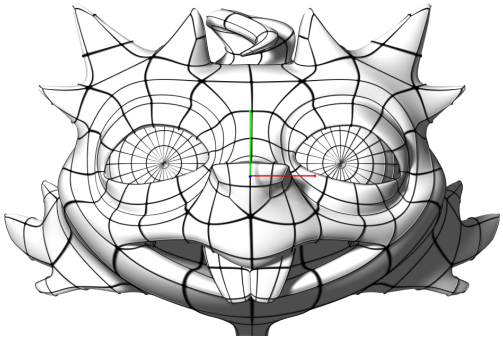
Klusterin toimintaperiaate on yksinkertainen: elementillä määritellään, mihin verteksit sijoitetaan ja skalaareilla määritellään, kuinka paljon verteksit liikkuvat kyseiseen paikkaan. Elementtiä voi siis siirtää, kääntää ja skaalata, näin sillä on helppo tehdä yksinkertaista sekundääristä animaatiota, kuten pientä eloa hahmon mahan rasvakerrokseen. Kuvassa 4 nähdään klusterin vaikutusalue ja miten klusterin liike vaikuttaa hahmon nenän vertekseihin.



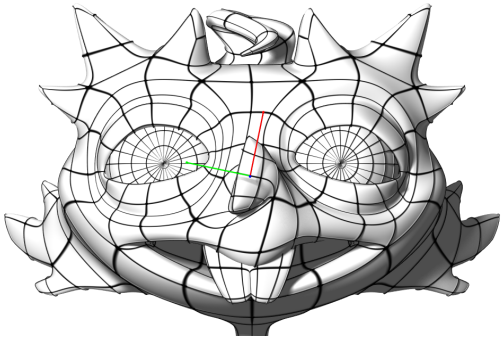
Klusterin vaikutusalue nenässä



Klusteri ilman muutosta



Klusteria siirretty ylös



Klusteria siirretty ylös ja kierretty

Kuva 4. Havainnekuva klusterin toiminnasta

2.3 AniSculpt

AniSculpt-tekniikka perustuu Blenderin skulptaustyökaluihin ja morpheihin. Morphien relatiivisuusongelma kierrettiin ensimmäisessä ”versiossa” tallentamalla verteksien liike tiedostoon. Animaatiosta tehtiin näin staattinen, jolloin morphit voitiin interpoloida animaation jälkeen. Jos animaatiota tahdottiin muuttaa, täytyi verteksien liike tallentaa uudestaan. Niiltä osin kun animaatiota oli muutettu, oli todennäköistä, että AniSculpt-tekniikalla toteutetut verteksianimaatiot piti tehdä uudestaan.

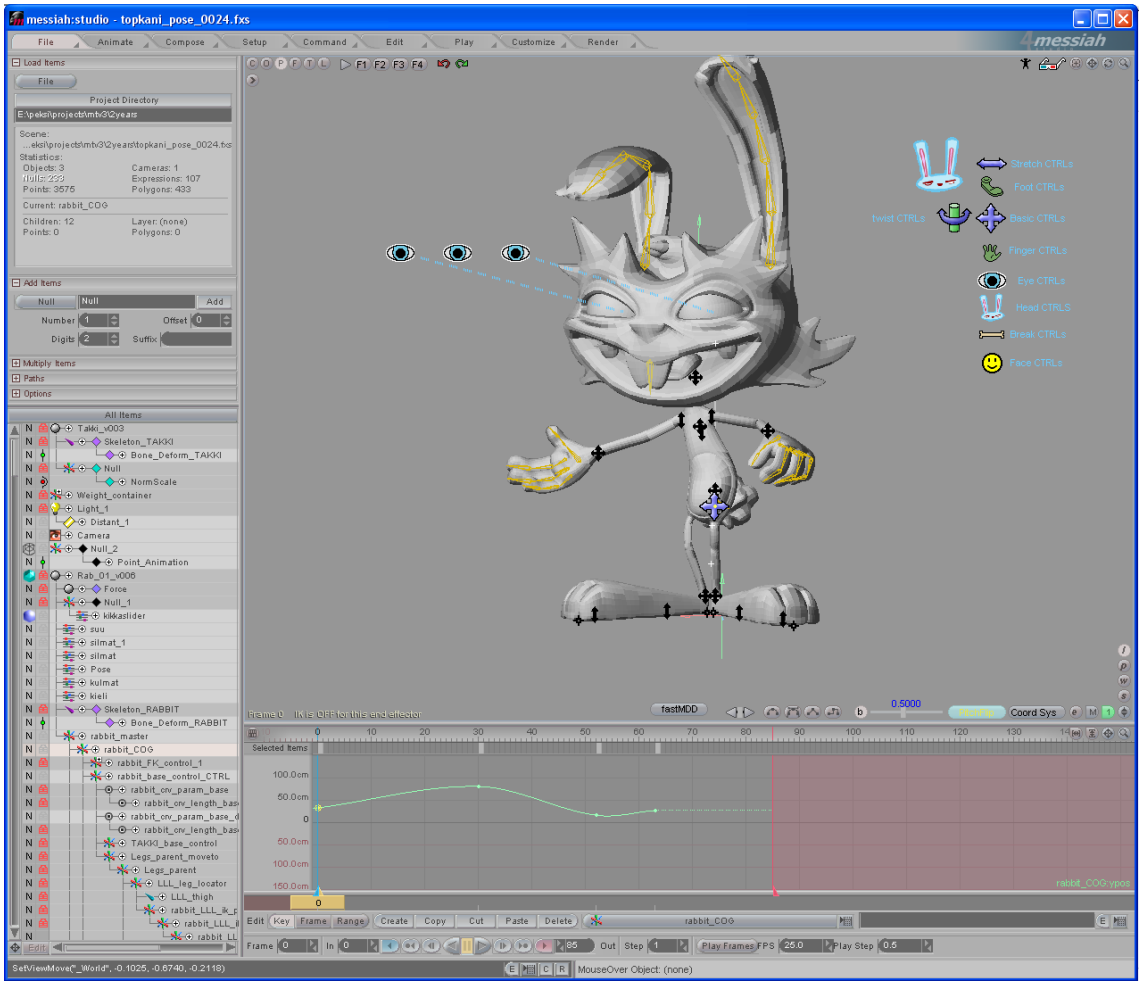
AniSculpt-tekniikan ottaessa tuulta siipien alle Blenderin kehittäjäyhteisö esitti ratkaisun, jolla päästiin eroon verteksien liikkeiden tallentamisesta. Näin alla olevaa animaatiota voitiin muokata ilman pelkoa siitä, että AniSculpt -tekniikalla toteutettu animaatio pitäisi tehdä uudestaan.

AniSculpt-tekniikka on kolmesta esitellystä tavasta tehdä verteksianimaatiota lähimpänä Point Animationia. Ajallisesti AniSculptin ensimmäisen ”version” aikaan Point

Animationissa oli jo ratkaistu relatiivisuusongelma, mutta verteksien manipulointi oli vielä kankeaa.

3 MESSIAH

Project: messiah kehitettiin alun perin Station X -studion sisällä sen omiin tarpeisiin. Station X käytti tuotannoissaan LightWave 3D -ohjelmistoa; Messiah olikin aluksi animaatio-plugini kyseiseen ohjelmistoon. 1999 Messiah julkaistiin ensi kertaa suurelle yleisölle pluginina, mutta vuonna 2001 Messiah oli kehittynyt jo itsenäiseksi ohjelmaksi. Samalla kun Messiah kehittyi pluginista itsenäiseksi ohjelmistoksi olivat Messiahin kehittäjät irtaantuneet Station X -studiosta ja perustaneet Project: messiah Groupin.



Kuva 5. Messiah 4.5

Messiahin kehitys toi mukanaan myös renderöinnin; aikoinaan paljon huomiota kerännyt Arnold renderin koodi lisensoitiin ja sisällytettiin Messiahiin. Arnold render oli siihen aikaan nopea Global Illumination -renderöijä. Renderöinnin ohella yhteentoimivuutta muiden ohjelmistojen kanssa kehitettiin plugineilla, jotka mahdollistivat animaation välittömän päivityksen Messiahin ja käytetyimpien 3d-ohjelmistojen välillä. Tänä päivänä käytetyin tapa siirtää animaatioita Messiahista on MDD-formaatti, joka sisältää verteksien liikkeen sellaisenaan. Messiahin version 4.5 (kuva 5) myötä tuli myös mahdollisuus tallentaa FBX- ja collada-tiedostoja.

3.1 Filosofia

Messiahin taustoista johtuen, sillä ei voi luoda uutta geometriaa tai muokata topologiaa. Mallit Messiahiin voi ladata tällä hetkellä OBJ-, DXF-, LWO- tai LXO-formaatissa, eli käytännössä 3d-mallit voidaan tehdä millä vain mallinusohjelmalla. Mallin deformaukseen on Messiahissa tarjolla paljon työkaluja, mutta aikaa ennen Point Animationia ei millään niistä voinut vaivattomasti muokata yksittäisiä verteksejä. Suurin syy tähän on Messiahin kehittäjien valitsema filosofia: työkalujen on toimittava silloinkin, kun mallin geometria vaihtuu. Kärjistettynä tämä filosofia rajaa pois kaiken toiminallisuuden, joka on riippuvainen verteksien järjestyksestä.

Messiah tukee morpheja, vaikka ne ovatkin riippuvaisia verteksien järjestyksestä. Käytännössä morpheja ei kuitenkaan määritellä Messiahin sisällä, vaan morphit luetaan 3d-malleista, joihin referoidaan. Näin ollen Messiahissa itsessään ei luoda mitään verteksien järjestyksestä riippuvaa tietoa. Mallia muokattaessa täytyy pitää huoli siitä, että morphit päivitetään. Helpointa se on LightWaven mallintajalla, sillä morphit tallentuvat suoraan LWO-tiedostoon.

Point Animation on yksinkertaistettuna vain morpheja toistensa jälkeen, mutta nämä morphit animaattori luo Messiahin sisällä sen hetkiselle 3d-mallille. Toisin sanoen Point Animation on Messiahin filosofian vastainen työkalu.

3.2 SDK

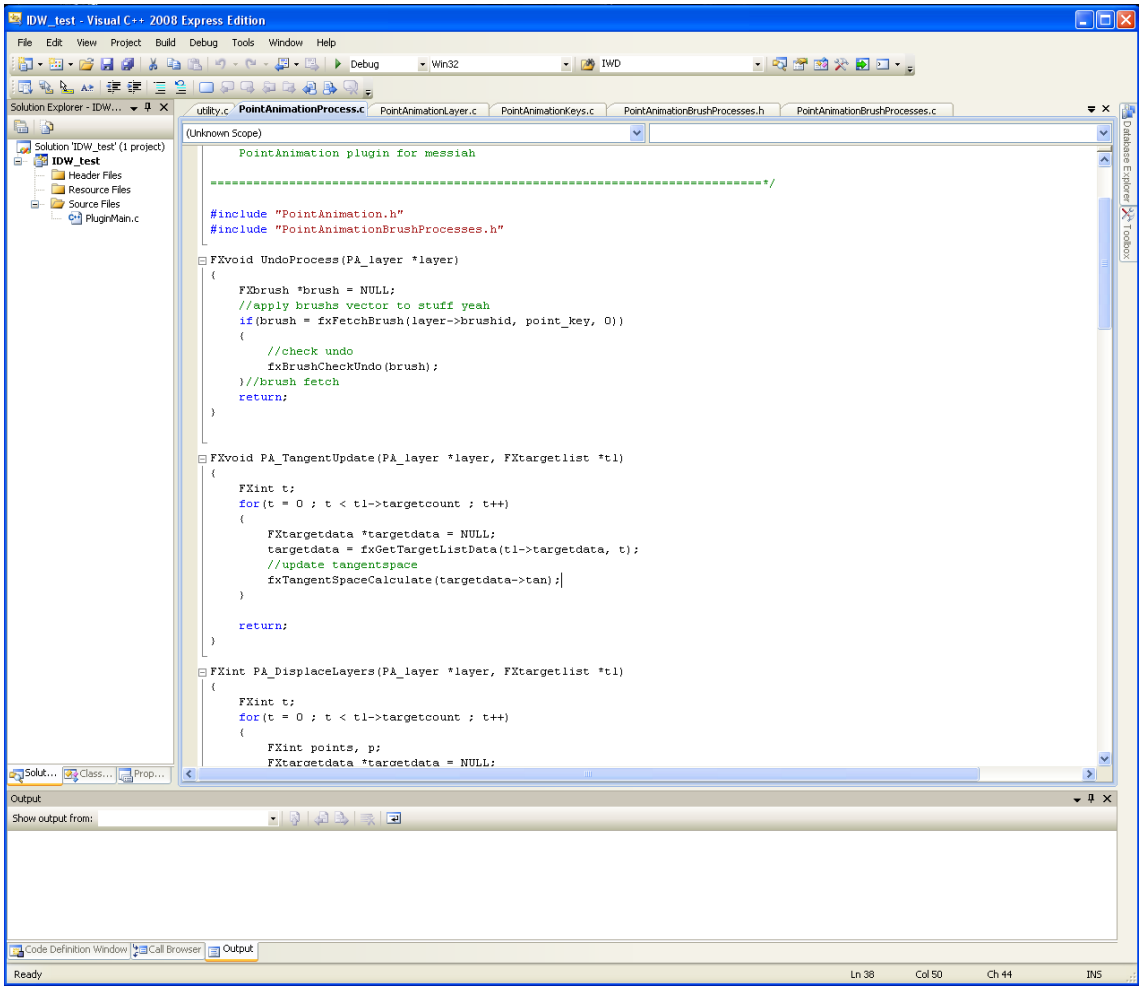
Software Development Kit, lyhyemmin SDK, on eräänlainen kirjastokokoelma, joka helpottaa ohjelmistojen kehittämistä. Tällaiset kirjastot sisältävät toimivia kokonaisuuksia, jotka ovat usein helppokäyttöisiä ja mahdollisimman optimoituja tarkoitukseensa. SDK sisältää vähintään yhden API:n, joitakin apuohelmia, kattavan ohjeistuksen, esimerkkiohjelmia ja mahdollisesti asennusohjelman. (Bolton.)

API lyhenne tulee sanoista Application Programming Interface, suomeksi rajapinta. Rajapinta on kokoelma käskyjä, joilla voidaan ohjata ohjelmistoa. SDK:n avulla voidaan siis ulkoistaa monimutkaisia palasia ohjelmistosta ja välttää näin ollen ”pyörän keksimistä uudelleen”.

Messiahin SDK koostuu Messiah API:sta, esimerkeistä ja ohjeistuksesta. Messiah API sisältää käskyjä, joilla ohjelmoija voi ohjata Messiahia. Täytyy kuitenkin huomioida, että Messiahin sisäinen kehitysryhmä ei itse käytä SDK:ta, vaan ohjelmoivat suoraan Messiahin ydintä. Tätä mahdollisuutta ei ole ulkopuolisilla, meidän on tyytyminen SDK:hon. Tästä johtuen Messiahin SDK on valittevavasti hyvin suppea ja asettaa rajoitteita sille, minkälaisia plugineja sillä voi saada aikaan.

Messiah SDK:n parissa työskennellessä törmää usein tilanteeseen, jossa huomaa ettei löydy käskyä, jolla saisi tarvittavan arvon joko haettua tai asetettua. Point Animationin kohdalla tätä tapahtui tämän tästä, mutta Messiahin kehittäjät olivat hyvin avuliaita ja lisäsivät käskyjä SDK:hon tarpeen mukaan.

Point Animationin ohjelmointi tapahtui Microsoft Visual C++ 2008 Express Edition ohjelmointiympäristössä (kuva 6), joka on kaikkien ladattavissa ilmaiseksi Microsoftin verkkosivustoilta.



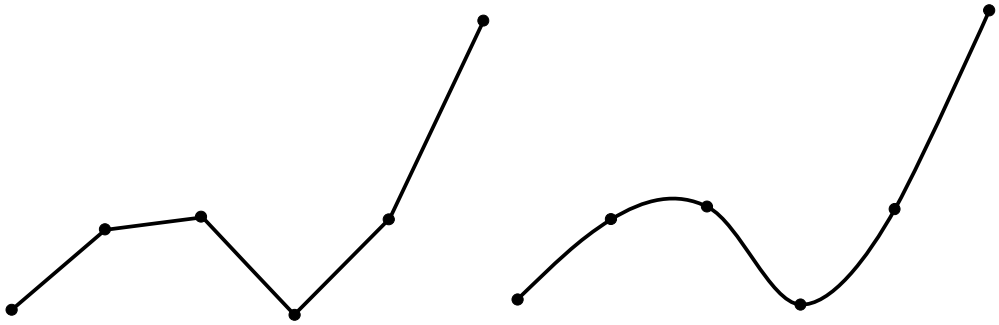
Kuva 6. Microsoft Visual C++ 2008 Express Edition ohjelmointiympäristö

4 POINT ANIMATION

Point Animationin ensimmäinen toimiva versio oli kankea käyttää ja laskennallisesti raskas. Jokaista verteksiä kohden luotiin kaksi Messiah-elementtiä (Null), jotka täyttivät listaa, josta animaattori valitsee, mitä elementtiä kontrolloi. Se kuitenkin osoitti tekijälleen, että idea oli toteutettavissa, oli vain suoritettava elementtien käyttämät laskutoimitukset omassa ohjelmassa sisäisesti. Tämä oli kuitenkin iso prosessi, sillä 3d-grafiikassa käytetty matematiikka ei ollut vielä tekijän hallussa.

Ensimmäisessä versiossa kokeiltiin myös Corrective Morph -tekniikan automatisointia, mutta se pudotettiin pois listalta liiallisen monimutkaisuuden välttämiseksi. Corrective Morph -tekniikka lyhyesti tarkoittaa morphien sitomista esimerkiksi luiden asentoon. Tämä päätettiin toteuttaa erillisessä pluginissa, joka tosin on vielä kehitysvaiheessa.

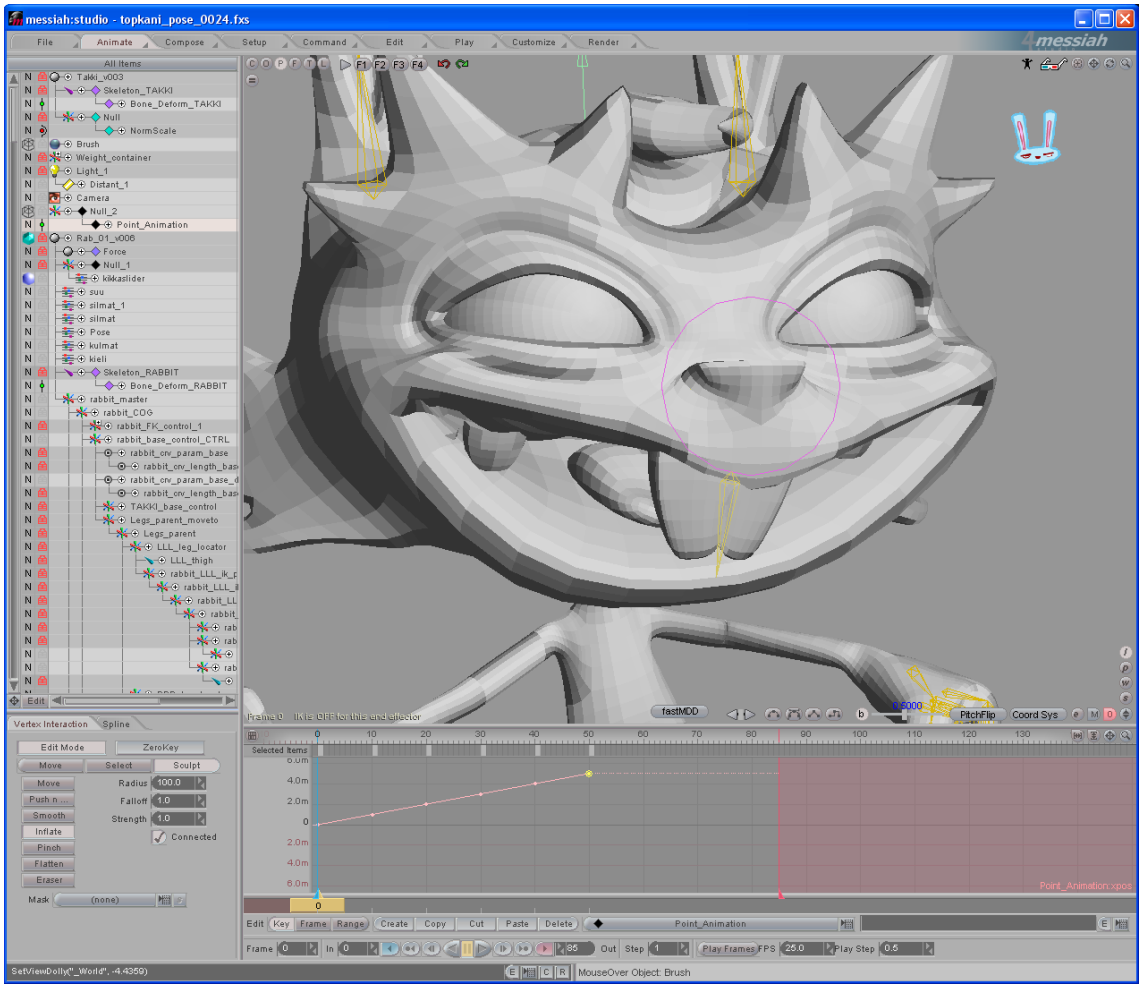
Isoimmaksi pähkinäksi matematiikan osalta muodostui kysymys: mihin yksittäisen verteksin paikka on relatiivinen? Tähän löytyi melkein suora vastaus teksturointi puolelta löytyvästä tekniikasta – tangent space, joka vaatii mm. 3d-grafiikassa käytettävien matriisien ymmärrystä. Myös verteksien liike oli hyvin tärkeä, sillä lineaarinen interpolaatio ei ollut visuaalisesti miellyttävän näköistä (kuva 7). Tarkoitukseen paremmin sopiva pehmeä spline-interpolaatio vaati jälleen uuden asian omaksumista.



Kuva 7. Lineaarinen- ja spline-interpolaatio

Muutaman kuukauden päästä ensimmäisestä versiosta saatiin toinen kehitysversio valmiiksi. Tässä relatiivisuusongelma ja spline-interpolaatio oli ratkaistu, mutta Messiah SDK:n rajoittuneisuudesta johtuen oli käytettävyys vielä raskasta. Turhaan luotavista elementeistä oli päästy eroon, mutta tilalle oli tullut ikonit. Jokaista verteksiä kohden luotiin yksi ympyrän muotoinen pieni ikoni, joita kutsutaan Messiahissa armatureiksi. Käytännössä animaattorin piti raahata tällaista ikonia vaikuttaakseen verteksin paikkaan. Teoriassa se toimi, mutta käytännössä armatureja ei ollut suunniteltu käytettäväksi näin, monimutkaisuudessaan ja lukumäärässään ne haittasivat käytettävyyttä. Versiossa oli mukana alustavat kokeilut erilaisille manipulointityökaluille.

Kolmas kehitysversio sisällytettiin lukuisten parannusten jälkeen Messiahiin; Point Animation oli virallisesti julkaistu. Armaturet olivat historiaa ja verteksien liikuttelu oli jouhevaa. Myös manipulointityökalut olivat jalostuneet ja osa niistä toimii samalla periaattella kuin tämän päivän mallinnusohjelmista löytyvät sculpting-työkalut. Tällaiset työkalut mahdollistavat mallin muokkauksen virtuaalisen siveltimen vedoilla, ja työtapa muistuttaa hieman kuvanveistäjän työtapaa.



Kuva 8. Point Animationin käyttöliittymä vasemmassa alakulmassa

Käytännössä Messiahin SDK ei tarjoa kovinkaan kummoisia ratkaisuja luoda Point Animationin kaltaisen pluginin vaatimia käyttöliittymiä. Messiah tarjoaa jokaiselle pluginille oman tilansa, johon on helppo luoda erilaisia nappeja, valitsimia sekä teksti- ja numerokenttiä, mutta 3d-ikkunan tapahtumiin ei voi suoraan reagoida. Kuvan 8 vasemmassa alakulmassa on nähtävissä plugineille varattu tila käyttöliittymän rakentamiseen. 3D-ikkunaan reagoidakseen Point Animation onkin jaettu kahteen erilliseen pluginiin, jotka keskustelevat keskenään. Erillinen armature-plugin rekisteröi käyttäjän klikkailut 3d-ikkunassa ja välittää jalostetun tiedon itse deformer-pluginille.

Armature-pluginia luodessa tavoitteena oli saada siitä yleishyödyllinen komponentti, jonka avulla jokainen plugineja Messiahille kehittävä voisi helposti ottaa vastaan käyttäjän klikkailuja 3d-ikkunassa ja reagoida niihin. Periaatteessa armature-pluginin kanssa ”keskustelu” onnistuu, mutta helppous hautautui kokeiluiden, testien ja

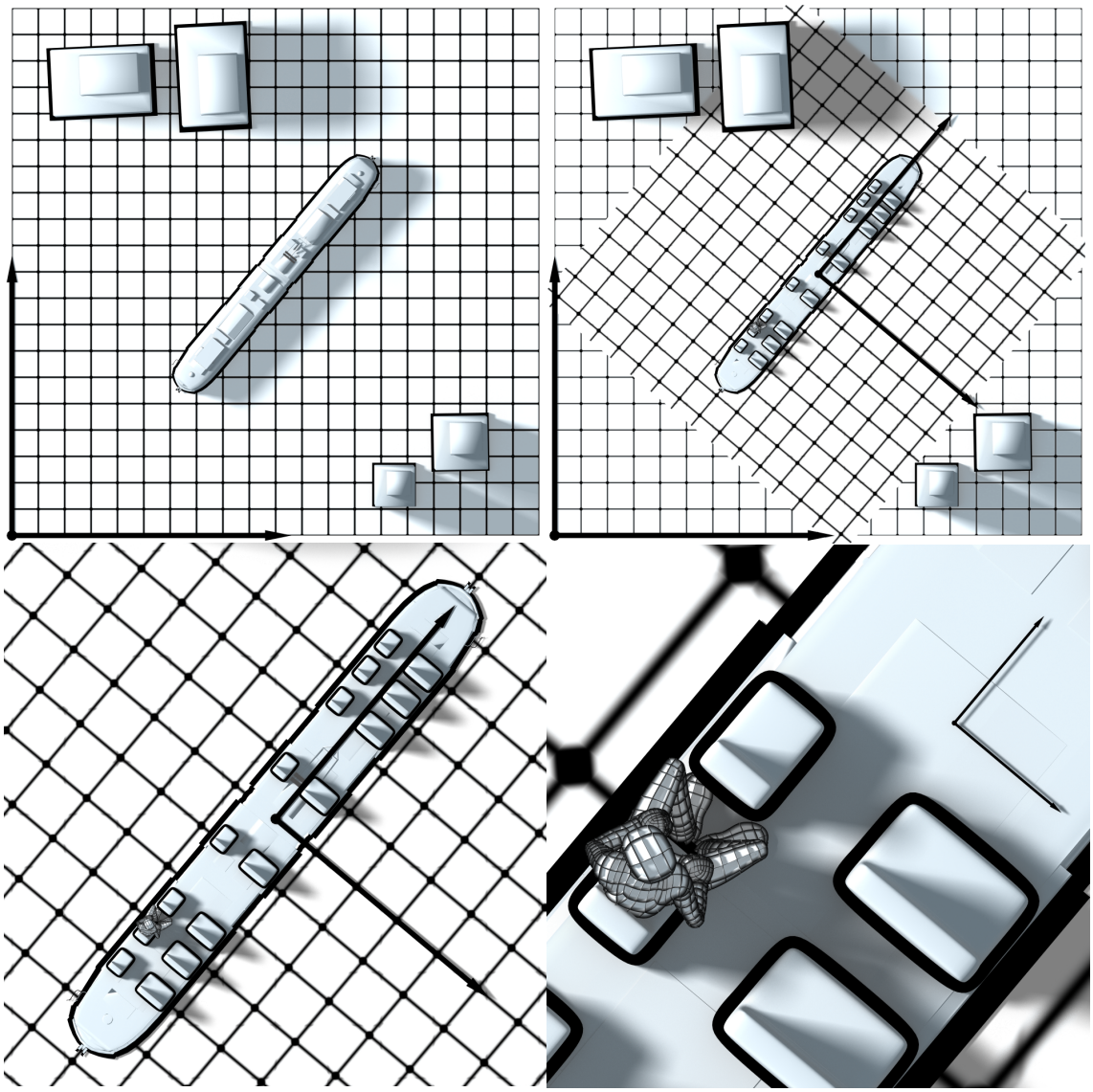
epäselvän päämäärän alle. Nyt kun päämäärä on selkeä, olisi tällaisen julkisen komponentin uudelleen luominen käyttäjäyställisempänä mahdollista.

Käytännössä armature-plugin luo yhden ison näkymättömän ikonin, jonka klikkailuja se nuuskii. Messiahin SDK:n ollessa suppea, täytyi matalan tason funktiot luoda lähes kaikki itse. Vaikka hyvin perustavaa laatua olevien funktioiden puuttuminen aiheuttaa yleensä hammasten kiristelyä, niin toisaalta oppimisprosessina se oli mitä mainioin, sillä tavoin 3d-grafiikassa käytettävästä matematiikasta on pakko ottaa perusteellisesti selvää. Syvempi ymmärrys 3d-grafiikassa käytetystä matematiikasta auttaa ymmärtämään 3d-ohjelmistojen toimintaa. Point Animationin kehityksessä tulivat tutuiksi mm. seuraavat käsitteet ja niihin liittyvä matematiikka: raytrace, triangulation, barycentric coordinates, vector, matrix, tangent space ja spline interpolation.

5 RELATIIVISUUS

Suurin ongelma, jonka ratkaisu tekee Point Animationista erityisen, on se mihin yksittäisen verteksin paikka on relatiivinen. Tämän relatiivisuuden selittämiseksi täytyy turvautua analogiaan, verrattavissa olevaan yksinkertaisempaan esimerkkiin:

Tarkastelkaamme kaupungin karttaa, joka on jaettu x- ja y-akseliin. Kaupungissa kulkee ratikka, joka ajaa kiskoja pitkin. Ratikassa istuu henkilö toiseksi viimeisessä penkkirivistössä yksittäisellä paikalla. Missä henkilö sijaitsee? Voisimme tarkkailla jatkuvasti henkilön sijaintia kaupungin koordinaatistossa, mutta toisaalta on ehkä helpompi ilmaista, että henkilö istuu ratikan toiseksi viimeisen penkkirivistön yksittäisellä paikalla, kuin tarkkoja x- ja y-arvoja. Kun tiedämme missä ratikka on, tiedämme tarkalleen missä kyseinen henkilö on sillä hetkellä. Voimme kuvitella ratikalle, joka sijaitsee kaupungin koordinaatistossa, oman koordinaatiston, jossa henkilö sijaitsee (kuva 9).



Kuva 9. Raitiovaunun koordinaatisto

Asian voi viedä vielä pidemmälle, voimme sanoa, että kaupunki sijaitsee maapallon koordinaatistossa, maapallo aurinkokuntamme koordinaatistossa, aurinkokunta galaksimme koordinaatistossa, galaksimme maailmankaikkeuden koordinaatistossa. On hankala kuvitella koordinaatistoa, jossa maailmankaikkeus sijaitsee, joten sopikaamme, että se on ylin hierarkiassa, nk. absoluuttinen kordinaatisto. Ensi käden tieto henkilön sijainnista kaupungin tai maapallon koordinaatisossa on vielä suhteellisen hyödyllistä informaatiota, mutta sijainti maailmankaikkeudessa tuskin on oleellista. Oleellisempaa henkilön kannalta on se, missä ratikan uloskäynti on, ja milloin on aika jäädä kyydistä.

3d-grafiikan kannalta on olennaista, että tiedämme sijainnin absoluuttisessa koordinaatistossa (world space). Tietokoneet toimivat kuitenkin matematiikan, eivät

aistihavaintojen, varassa ja onneksemme matematiikan avulla voimme esittää erilaisia koordinaatistoja sekä siirtyä koordinaatistosta toiseen vaivatta. Tällaiset sisäkkäiset koordinaatistot myös vähentävät tarvittavien laskutoimitusten määrää ja näin ollen konetehoa vapautuu muihin tehtäviin.

5.1 Matriisit 3d-grafiikassa

Matriiseilla on perustavanlaatuinen merkitys 3d-grafiikassa, jossa niitä pääasiassa käytetään kuvaamaan kahden eri koordinaatiston suhdetta. Matriisit määrittelevät miten vektori siirretään koordinaatistosta toiseen laskennallisesti. (Dunn & Parberry 2002, 83.)

Matriisi on kaksiulotteinen taulukko, joka koostuu riveistä ja sarakkeista. Matriisin alkiot sisältävät yleensä reaalilukuja. Taulukko voi olla minkä kokoinen tahansa eikä rivien ja sarakkeiden määrän tarvitse täsmätä. Matriisin voidaan sanoa olevan kokoa *m x n*, missä *m* kertoo rivien ja *n* sarakkeiden lukumäärän. Alla oleva esimerkki on 4x5 kokoinen suorakaiteen muotoinen matriisi. Matriisi, jossa on yhtä monta riviä ja saraketta kutsutaan neliömatriisiksi.

$$\begin{bmatrix} 3{,}2 & 0 & 9 & 5 & 2 \\ 0 & 1 & 14 & 1 & 2 \\ 100 & 0 & 1 & 2 & 3 \\ 0 & 1 & 5{,}5 & 2 & 0 \end{bmatrix}$$

Matriiseja voidaan käyttää mm. lineaarisen yhtälöryhmän ratkaisemiseen, mutta tässä työssä käsitellään vain rotaatiomatriiseja, jotka ovat 3d-grafiikassa kokoa 3x3 tai 4x4. Rotaatiomatriisilla voidaan kuvata missä asennossa kappale on. Rotaatiomatriisi siis kertoo enemmän kuin mihin suuntaan ratikan nokka osoittaa; se kertoo myös onko ratikka vaikkapa katollaan tai kyljellään.

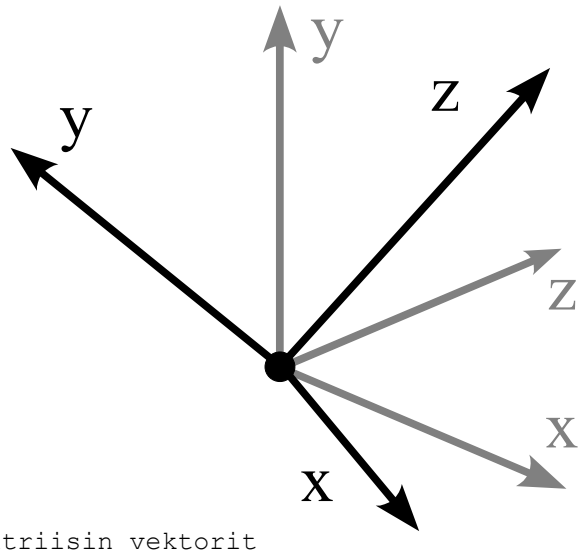
Ensikertalaiselle matriisin ja kappaleen asennon yhteys vaikuttaa varmasti hämärältä. Asiaa helpottaa, jos ajattelee jokaista matriisin saraketta vektorina, joista ensimmäinen on x, toinen y ja kolmas z. Toisin sanoen matriisin ensimmäinen rivi sisältää näiden

vektorien x-komponentin, toinen y-komponentin ja kolmas z-komponentin. Selvennyksen vuoksi alla olevaan matriisiin on lukujen sijaan merkitty vektorit ja niiden komponentit alaindeksissä.

$$\begin{bmatrix} x_x & y_x & z_x \\ x_y & y_y & z_y \\ x_z & y_z & z_z \end{bmatrix}$$

Piirtämällä nämä vektorit voimme tarkastella matriisia visuaalisesti. Kuvassa 10 on matriisi ja piirros sen vektoreista. Harmaat nuolet kuvastavat koordinaatiston x-, y ja z-akseleita.

$$\begin{bmatrix} 0{,}906 & -0{,}299 & 0{,}299 \\ 0{,}000 & 0{,}707 & 0{,}707 \\ -0{,}423 & -0{,}641 & 0{,}641 \end{bmatrix}$$

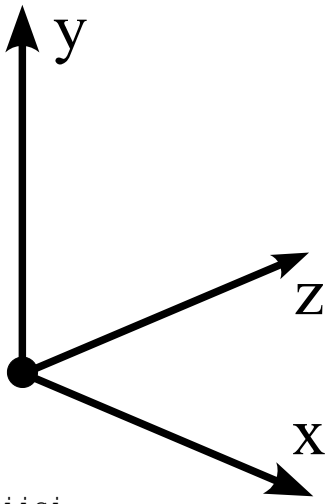


Kuva 10. Matriisin vektorit

Eli toisin sanoen kuvassa 10 esitetyn matriisin x-vektorin ”pää” sijaitsee koordinaatistossa pisteessä x 0,906, y 0 ja z -0,434. Se on luettavissa suoraan matriisin ensimmäisestä sarakkeesta.

Havainnollistetaan asiaa lisää. Yksi merkittävä matriisi on niin kutsuttu identiteettimatriisi, jonka kaikki arvot ovat diagonaalia lukuun ottamatta nollia. Identiteettimatriisin diagonaalille sijoittuvat luvut ovat ykkösiä; jos diagonaalin luvut olisivat jotakin muuta kuin pelkkiä ykkösiä tai nollia, olisi kyseessä diagonaalimatriisi. Jos matriisin kaikki alkiot olisivat nollia, se olisi nollamatriisi. 3x3 kokoisen identiteettimatriisin tapauksessa arvon yksi saavat alkiot x_x, y_y ja z_z. Identiteettimatriisi merkitään yleisesti isolla I:llä.

$$I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$



Kuva 11. Identiteettimatriisi

Kuten kuvasta 11 huomaamme sijoittuu identiteettimatriisi täydellisesti koordinaatiston akseleiden suuntaisesti. Tästä syystä emme edes näe koordinaatiston harmaita akseleita. Sama toistuu luvuissa, x-vektori sijaitsee koordinaateissa x 1, y 0 ja z 0. Identiteettimatriisi siis edustaa matriisia, jossa ei ole muutosta. Yksinkertaistettuna se on ikäänkuin matriisien luku yksi, kun sillä kertoo niin yhtälö tai luku ei muutu.

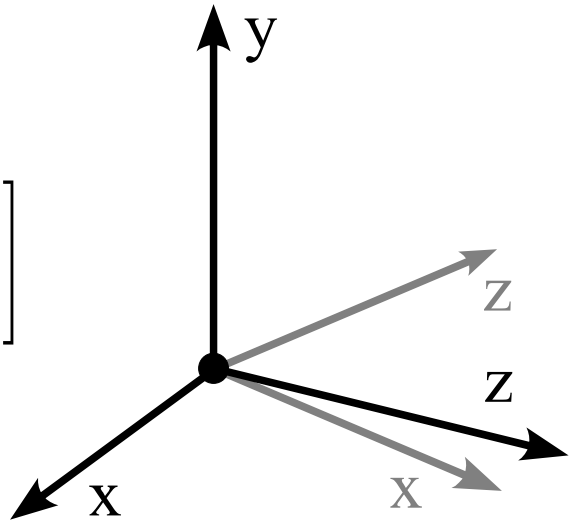
$$1 * (w+5+2*s) = w+5+2*s$$

Samalla tavalla kun jotain kerrotaan identiteettimatriisilla pysyy se muuttumattomana. Tässä esimerkkinä matriisien kertolasku.

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x_x & y_x & z_x \\ x_y & y_y & z_y \\ x_z & y_z & z_z \end{bmatrix} = \begin{bmatrix} x_x & y_x & z_x \\ x_y & y_y & z_y \\ x_z & y_z & z_z \end{bmatrix}$$

Identiteettimatriisilla kertomisen hyöty on siis kyseenalaista, mutta auttaa ymmärtämään mistä matriiseissa on kyse. Asiat menevät mielenkiintoisemmiksi kun kumpikaan kertolaskun matriiseista ei ole identiteettimatriisi. Kuvassa 12 on matriisi M_1. Se on käytännössä koordinaatisto, mitä on kierretty y-akselia, jota tarkastellaan origosta käsin, pitkin 75 astetta vastapäivään.

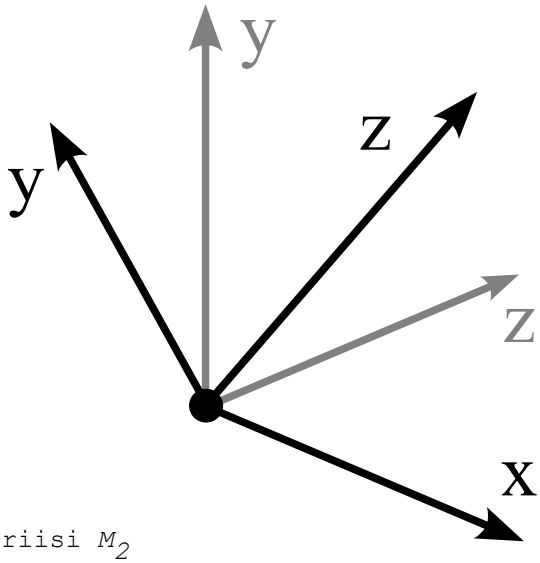
$$M_1 = \begin{bmatrix} 0{,}259 & 0{,}000 & 0{,}966 \\ 0{,}000 & 1{,}000 & 0{,}000 \\ -0{,}966 & 0{,}000 & 0{,}259 \end{bmatrix}$$



Kuva 12. Matriisi M_1

Kuvan 13 matriisia, M_2:sta, on kierretty koordinaatiston x-akselin ympäri 30 astetta ylöspäin.

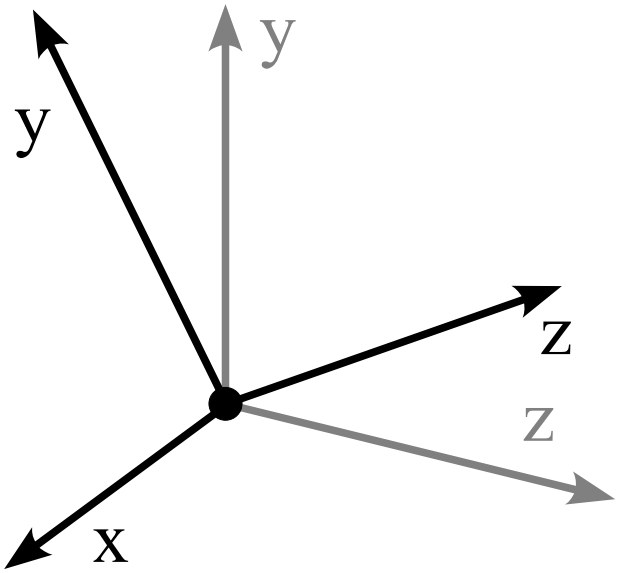
$$M_2 = \begin{bmatrix} 1{,}000 & 0{,}000 & 0{,}000 \\ 0{,}000 & 0{,}866 & 0{,}500 \\ 0{,}000 & -0{,}500 & 0{,}866 \end{bmatrix}$$



Kuva 13. Matriisi M_2

Seuraavaksi, sen kummemmin paneutumatta matriisien laskusääntöihin, kerromme matriisit keskenään.

$$M_1 * M_2 = \begin{bmatrix} 0{,}259 & -0{,}483 & 0{,}837 \\ 0.000 & 0{,}866 & 0{,}500 \\ -0{,}966 & -0{,}129 & 0{,}224 \end{bmatrix}$$



Kuva 14. Matriisien M_1 ja M_2 tulo

Kun visualisoimme laskutoimituksen tulokseksi putkahtaneen matriisin, voimme huomata mielenkiintoisen ilmiön; matriisia M_1 on kierretty 30 astetta ylöspäin sen oman x-vektorin ympäri. Kuvasta 14 on jätetty pois koordinaatiston akselit selkeyden vuoksi. Kuvassa 14 matriisi M_1 on harmaa ja $M_1 * M_2$ kertortolaskun tulo on musta.

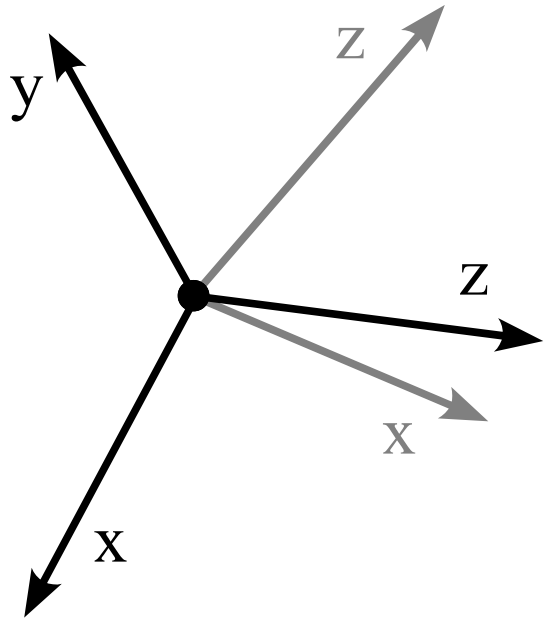
Reaaliluvuilla suoritettavien kertolaskujen suoritusjärjestyksellä ei ole väliä. Kertolasku on kommutatiivinen eli vaihdannainen. Matriisien välinen kertolasku ei ole kommutatiivinen eli jos järjestystä vaihdetaan niin tulos muuttuu.

$$2{,}1 * 327 = 327 * 2{,}1$$

$$M_1 * M_2 \neq M_2 * M_1$$

Kun matriisien laskujärjestys vaihdetaan muotoon $M_2 * M_1$ on lopputulos sama kuin matriisia M_2 kierrettäisiin sen oman y-vektorin ympäri 75 astetta.

$$M_2 * M_1 = \begin{bmatrix} 0{,}259 & 0{,}000 & 0{,}966 \\ -0{,}483 & 0{,}866 & 0{,}129 \\ -0{,}837 & -0{,}500 & 0{,}224 \end{bmatrix}$$



Kuva 15. Matriisien tulo järjestyksessä $M_2 * M_1$

Kuvassa 15 on harmaalla M_2 matriisin vektorit ja mustalla $M_2 * M_1$ tulon matriisin vektorit. Y-vektorit ovat molemmilla matriiseilla samat. Matriiseja voidaan ketjuttaa kertolaskulla niin monta kuin on tarve ja jokainen matriisi edustaa omanlaistaan koordinaatistoa. Matriisit mahdollistavat sisäkkäisten koordinaatistojen määrittelemisen ja niiden välillä ”liikkumisen” vaivatta. 3d-graafikon luodessa ohjelmistossa hierarkioita hän käytännössä määrittelee koordinaatistoja suhteessa toisiin koordinaatistoihin. Lähes poikkeuksetta jokaisessa 3d-ohjelmistossa koordinaatistot ovat toteutettu matriiseilla. Nämä matriisit eivät välttämättä näy ohjelmiston käyttäjälle, mutta 3d-ohjelmistoa, sen pluginia tai skriptiä tehdessä ne ovat yksi tärkeimmistä työkaluista.

Identiteettimatriisi on oiva työkalu havainnollistamaan miten voidaan kääntää matriisi takaisin niin ettei siinä ole muutosta. Tarkastellaan ensin hieman helpompaa tapausta, käänteislukua. Yksinkertainen määritelmä käänteisluvulle on se, että luvun ja sen käänteisluvun kertolaskun tulo on yksi. Hieman mutkikkaammin määriteltynä luvun käänteisluvun käänteisluku on sama luku. Tästä esimerkkinä luku viisi ja sen käänteisluku.

$$5 * x = 1 \qquad \frac{1}{5} * x = 1$$

$$x = \frac{1}{5} \qquad x = 5$$

Vastaavasti on olemassa käänteismatriisi, jonka kertolaskun tulo alkuperäisen matriisin kanssa on identiteettimatriisi.

$$M_1 * M_2 = I$$

Mielivaltaisen matriisin käänteismatriisin selvittäminen ei ole mutkatonta ja joka kerta sellaista ei edes välttämättä löydy, mutta onneksemme rotaatiomatriisin ominaisuuksiin kuuluu ortogonaalisuus. Ortogonaalisuus tarkoittaa suorakulmaisuutta ja käytännössä sen näkee siinä, että ortogonaalisen matriisin vektorit ovat 90 asteen kulmassa toisiinsa nähden. Toinen ortogonaalisuuden vaatimus koskee matriisin vektorien pituutta, sillä niiden pitää olla yksi eli yksikkövektoreita. Tällaisen ortogonaalisen matriisin käänteismatriisi on suoraan sen transpoosi. Matriisin transpoosi on matriisi, jonka rivit ja sarakkeet ovat vaihtaneet paikkaa, jolloin ensimmäisestä rivistä tulee ensimmäinen sarake ja päinvastoin. Transpoosi merkitään matriisin yläindeksiin isolla *T*:llä.

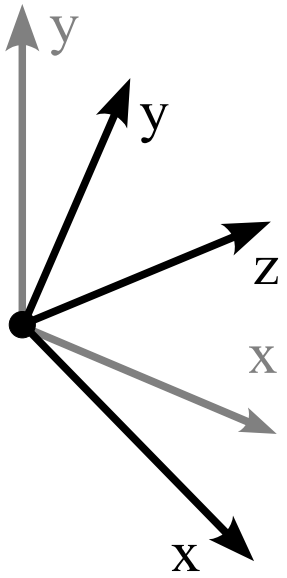
$$M = \begin{bmatrix} x_x & y_x & z_x \\ x_y & y_y & z_y \\ x_z & y_z & z_z \end{bmatrix} \qquad M^T = \begin{bmatrix} x_x & x_y & x_z \\ y_x & y_y & y_z \\ z_x & z_y & z_z \end{bmatrix}$$

Selvennyksen vuoksi: ortogonaalisen matriisin käänteismatriisi on transpoosi. Matriisin ja käänteismatriisin tulo on identiteettimatriisi. Toisin sanoen ortogonaalisen matriisin ja sen transpoosin tulo on identiteettimatriisi.

$$M * M^T = I$$

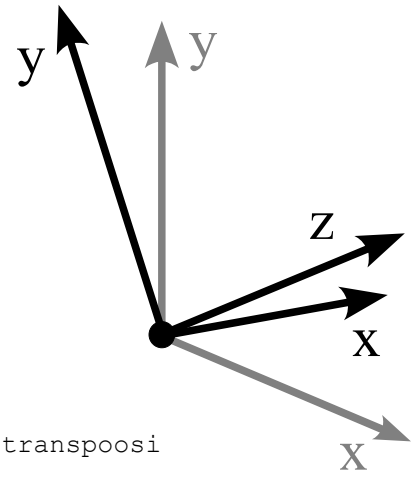
Eli vaihtamalla matriisin rivien ja sarakkeiden paikkaa saadaan aikaan toinen matriisi, jonka avulla voidaan liikkua ensimmäisen matriisin määrittelemästä koordinaatistosta takaisin koordinaatistoon, joka ei sisällä muutosta. Havainnollistetaan tämä matriisin avulla, joka on kääntynyt koordinaatiston z-akselin ympäri 25 astetta myötäpäivään ja sen käänteismatriisilla (kuvat 16, 17 ja 18).

$$M = \begin{bmatrix} 0{,}906 & 0{,}423 & 0{,}000 \\ -0{,}423 & 0{,}906 & 0{,}000 \\ 0{,}000 & 0{,}000 & 1{,}000 \end{bmatrix}$$



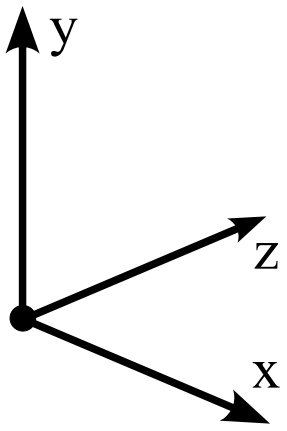
Kuva 16. Matriisi *M*

$$M * M^T = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$



Kuva 17. Matriisin *M* transpoosi

$$M^T = \begin{bmatrix} 0{,}906 & 0{,}423 & 0{,}000 \\ -0{,}423 & 0{,}906 & 0{,}000 \\ 0{,}000 & 0{,}000 & 1{,}000 \end{bmatrix}$$



Kuva 18. Matriisin *M* ja sen transpoosin tulo

Myös vektori voidaan mieltää matriisiksi, jossa joko sarakkeita tai rivejä on vain yksi. Se onko vektori rivi- vai sarakemuodossa on merkityksellistä matemaattisesti, sillä se määrää, miten vektorin ja matriisin voi kertoa keskenään. Alla esimerkkinä miltä sama 3d-vektori rivi- ja sarakemuodossa (oikean puoleinen) näyttävät.

$$\begin{bmatrix} 1 & 0{,}5 & 3 \end{bmatrix} \qquad \begin{bmatrix} 1 \\ 0{,}5 \\ 3 \end{bmatrix}$$

Vektorin voi kertoa matriisin kanssa, mutta laskujärjestys riippuu vektorin muodosta. Rivimuodossa järjestys on *v*M* ja sarakemuodossa *M*v*. Laskutoimitusten järjestystä ei voi kääntää, sillä ne ovat määrittelemättömiä. Vastaus on aina vektori, joka on samaa muotoa kuin alkuperäinen vektori eli rivimuodossa olevan vektorin ja matriisin tulo ei voi olla muuta kuin rivimuotoinen vektori.

$$\begin{bmatrix} 1 & 0{,}5 & 3 \end{bmatrix} * \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0{,}5 & -1 & 3 \end{bmatrix}$$

$$\begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 \\ 0{,}5 \\ 3 \end{bmatrix} = \begin{bmatrix} -0{,}5 \\ 1 \\ 3 \end{bmatrix}$$

Huomaa, että laskutoimitusten vastaus on eri, vaikka vektorit ovat samat! Tämän takia on merkityksellistä kumpaa muotoa käytetään. Tässä työssä käytämme sarakemuotoa siitä syystä, että Messiahin SDK käyttää niitä ja eri muotojen sotkeminen aiheuttaisi ylimääräistä pään vaivaa. Jos käyttäisimme rivimuotoa joutuisimme transponoimaan matriisit, joita olemme käyttäneet tähän mennessä sekä kääntämään niiden laskujärjestyksen. Toinen vaihtoehto olisi ollut päättää alunperin, että matriisin vektorit ovat riveillä sarakkeiden sijaan. Alla olevaa kannattaa tarkastella, kun rivit ja sarakkeet menevät sekaisin.

$$\begin{bmatrix} x_x & y_x & z_x \\ x_y & y_y & z_y \\ x_z & y_z & z_z \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \end{bmatrix} = OK!$$

$$\begin{bmatrix} x & y & z \end{bmatrix} * \begin{bmatrix} x_x & x_y & x_z \\ y_x & y_y & y_z \\ z_x & z_y & z_z \end{bmatrix} = OK!$$

Matriisin ja vektorin kertolaskun tulo on vektori, jota on muutettu matriisin avulla. Jos otetaan vektori, joka sijaitsee tietyssä paikassa absoluuttisessa kordinaatistossa, ja kerrotaan se matriisilla on lopputulos vektori, jota on rotaatiomatriisin tapauksessa käännetty.

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 0{,}5 \\ 2{,}5 \\ 1{,}12 \end{bmatrix} = \begin{bmatrix} 0{,}5 \\ 2{,}5 \\ 1{,}12 \end{bmatrix}$$

Identiteettimatriisilla kertominen ei muuta vektoria lainkaan.

$$\begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ -1 \\ 0 \end{bmatrix}$$

Yllä olevassa esimerkissä vektori on samansuuntainen absoluuttisen koordinaatiston x-akselin kanssa ja matriisin x-vektori osoittaa suoraan alaspäin. Matriisi on siis kiertynyt 90 astetta z-akselin ympäri. Tulo on sen mukainen eli vektori on kääntynyt 90 astetta z-akselin ympäri ja osoittaa nyt samaan suuntaan kuin matriisin x-vektori. Kokeillaan samaa matriisia vektoriin, joka on z-akselin suuntainen.

$$\begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

Nyt tulo ei muuttunut ja tämä johtuu siitä, että matriisi on kiertynyt z-akselia pitkin ja sen z-vektori on edelleen samansuuntainan kuin koordinaatiston z-akseli.

Sovitaan, että kaupunki sijaitsee absoluuttisessa koordinaatistossa vektorissa *k*. Kaupungissa kulkee raitivaunu, jonka tämän hetkistä suuntaa ilmaistaan matriisilla *R*. Raitiovaunun sijainti kaupungin koordinaatistossa on tiedossa ja se on merkitty vektorilla *r*. Merkitään vielä raitiovaunussa olevan henkilön sijainti raitiovaunun koordinaatistossa vektorilla *h*. Missä vektorissa henkilö sijaitsee absoluuttisessa koordinaatistossa? Alla annettu arvot näille laskutoimituksen palasille. Vektoreiden yläpuolella on nuoli tai viiva, joka kuvastaa sitä, että kyseinen symboli on vektori.

$$R = \begin{bmatrix} -0{,}446 & -0{,}742 & 0{,}500 \\ -0{,}857 & 0{,}515 & 0{,}000 \\ -0{,}258 & -0{,}429 & -0{,}866 \end{bmatrix}$$

$$\vec{k} = \begin{bmatrix} 2{,}5 \\ 0 \\ 10 \end{bmatrix} \quad \vec{r} = \begin{bmatrix} 20 \\ 1{,}45 \\ 25 \end{bmatrix} \quad \vec{h} = \begin{bmatrix} -0{,}91 \\ 0{,}8 \\ -5{,}86 \end{bmatrix}$$

Periaatteessa voisimme myös määritellä kaupungille matriisin, mutta oletetaan, että kaupunki ei tässä tapauksessa ole kääntynyt sijoillaan, vaikka 3d-ohjelmistossa se olisi mahdollista.

Lähdetään liikkelle raitiovaunun sijainnista absoluuttisessa koordinaatistossa. Raitiovaunu sijaitsee vektorissa *r* suhteessa kaupunkiin ja kaupunki vektorissa *k* suhteessa absoluuttiseen koordinaatistoon. Lisäämällä vektori *r* vektori *k*:hon saadaan raitiovaunun sijainti selville.

$$\vec{k} + \vec{r} = \begin{bmatrix} 22{,}5 \\ 1{,}45 \\ 35 \end{bmatrix}$$

Jos raitiovaunun suunta olisi sellainen, että sen matriisi olisi identiteettimatriisi, riittäisi että lisäämme vektoreihin *k* ja *r* vielä vektorin *h*. Tässä esimerkissä raitiovaunu on kuitenkin kääntynyt ja näin ollen vektori *h* täytyy ensin muuntaa absoluuttiseen

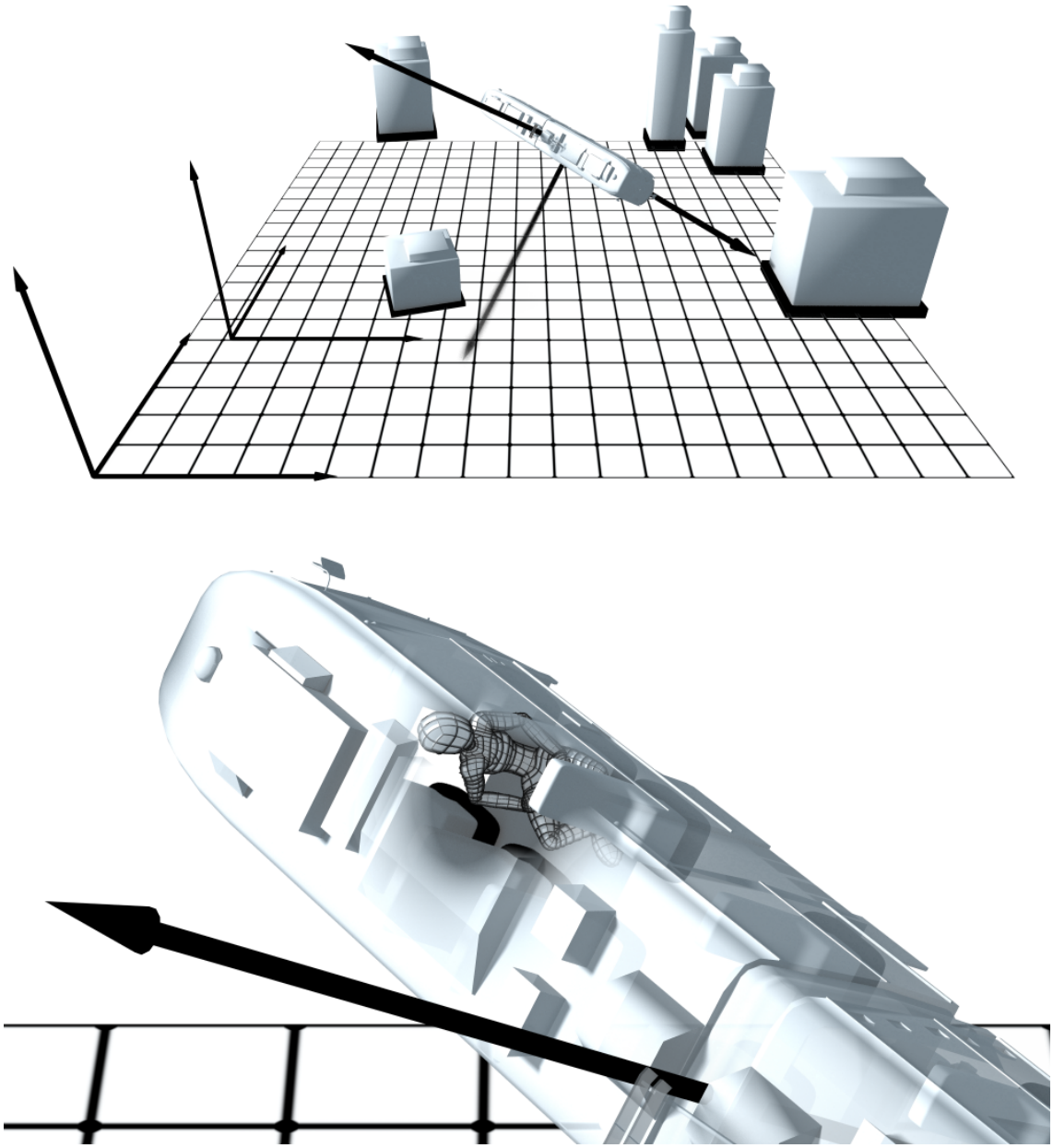
koordinaatistoon. Onneksemme kaupunki ei ole pyörinyt, joten selviämme tehtävästä käyttämällä raitiovaunun matriisia *R*. Jos kaupunki olisi eri asennossa, joutuisime kiertämään raitiovaunun matriisia ensin kaupungin matriisilla. Tällä tavoin saadaan matriisi, jonka avulla voidaan liikkua absoluuttisesta koordinaatistosta suoraan raitiovaunun koordinaatistoon ja sen käänteismatriisilla toisinpäin. Alla esimerkkinä jos kaupungilla olisikin matriisi *K*.

$$\vec{k} + \vec{K}r + RK\vec{h}$$

Raitiovaunun matriisilla *R* voidaan vektori *h* muuntaa raitiovaunun koordinaatistosta absoluuttiseen koordinaatistoon. Pyörittämättömän kaupungin matriisi *K* voidaan mieltää identiteettimatriisiksi, jolloin laskutoimituksen kannalta sillä ei ole merkitystä ja voidaan jättää pois yhtälöstä.

$$\vec{k} + \vec{r} + R^T\vec{h} = \begin{bmatrix} 19{,}382031 \\ 2{,}642053 \\ 39{,}966385 \end{bmatrix}$$

Kuten nokkelimmat jo matriisin ja vektoreiden sisällöstä pystyivät päättelemään, on raitiovaunu juuri kaatumassa kaupungin keskustan tuntumassa (kuva 19).



Kuva 19. Raitiovaunun kyydissä on mies

5.2 Tangent space

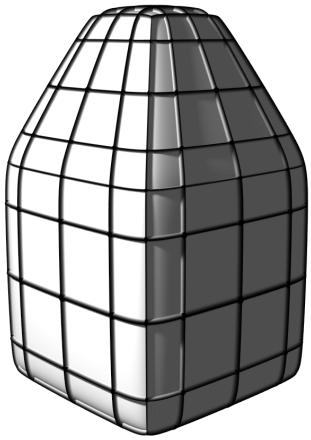
Mihin verteksi on relatiivinen? Sinänsä Point Animation ei eroa hirveästi tutusta morph-tekniikasta. Morphi itsessään on vain lista vektoreita mihin suuntaan verteksin tulisi liikkua. Point Animation on käytännössä lista morpheja, joiden interpolointi on riippuvainen siitä, missä järjestyksessä nämä morphit ovat. Point Animationin tarjoamat työkalut näiden morphien muokkaamiseen on vain käyttöliittymä kysymys, sillä sisällä oleva data on edelleen vain vektoreita. Oleellisempaa on kysyä mihin nämä vektorit

ovat relatiivisia. Ennen kuin paneudutaan Point Animationin relatiivisuuteen, niin tarkastellaan ensin tavallisen morphin käyttäytymistä.

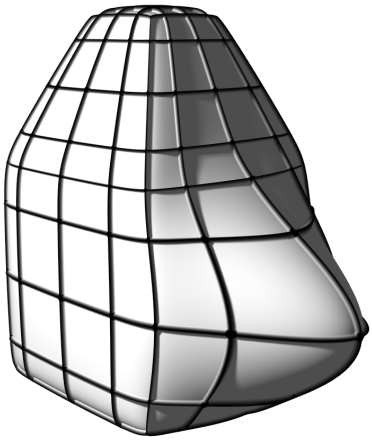
Koska morphin vektorit ovat kaikki samalla lailla relatiivisia, riittää että keskitymme yhden verteksin liikkeisiin ja muutokseen. LightWavessa on olemassa absoluuttinen morphi, joka on kaikista yksinkertaisin morph-tyyppi. Käytännössä absoluuttisen morphin vektorit ovat relatiivisia absoluuttiseen koordinaatistoon. Absoluuttisella morphilla mallin verteksit pakotetaan juuri niille sijoilleen, miten ne ovat morphissa määritelty, riippumatta siitä, missä mallin verteksit sijaitsevat. Absoluuttisia morpheja käytetään hyvin harvoin.

Yleisempi morphi, ja miten se useimmissa ohjelmissa on toteutettu, kulkee nimellä relatiivinen morphi. Relatiivisen morphin vektorien lähtöpaikka on relatiivinen vertekseihin, mutta vektorin suunnan relatiivisuus riippuu toteutustavasta. Normaalisti on tarjolla kaksi vaihtoehtoa; joko vektorin suunta ei ole riippuvainen mistään tai sitten se kääntyy 3d-mallin matriisin mukana. Morphit, jonka vektorit käännetään 3d-mallin mukana, ovat käytännöllisiä kappaleiden kanssa, joita animoidaan suoraan ilman deformaatiota. Tällä tavoin morphilla voidaan tuoda orgaanisuutta muuten jäykkään kappaleeseen.

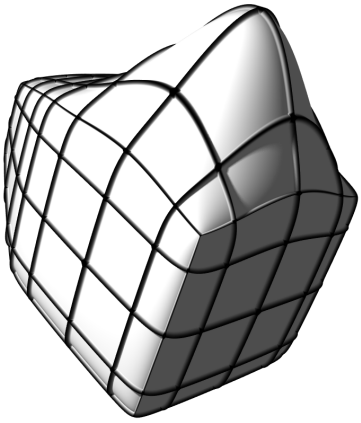
Morphia, jonka vektorit eivät käänny 3d-mallin mukana, voidaan käyttää kun mallia liikutetaan lähinnä deformaamalla, jolloin voidaan välttää joitakin turhia laskutoimituksia ja vapauttaa koneen resursseja muihin tehtäviin. Valitettavasti morphit täytyy interpoloida ennen deformausta, jolloin niiden määritteleminen ei aina ole helppoa. Tutuin esimerkki tällaisesta deformauksesta on luurankoon sidottu malli. Kuvassa 20 esimerkki mallin mukana kääntyvästä morphista ja morphista, joka ei käänny mukana.



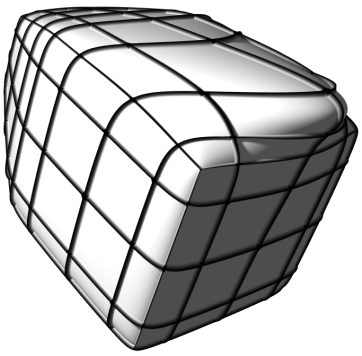
Lähtötilanne



Morphi



Vektorit kääntyvät objektin mukana



Vektorit eivät käänny

Kuva 20. Erilaisten morphien vaikutukset

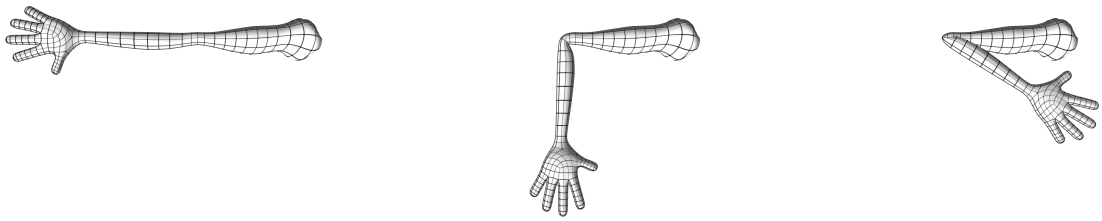
Käytännössä corrective-morphi tehdään korjaamaan jossain tietyssä luun asennossa esiintyvää ongelmaa mallin muodossa. Esimerkiksi olkapää on yleensä hyvin hankala rigata sen laajan liikeradan vuoksi. Jos muotoa korjataan kun hahmon luu on asennossa, jossa ongelma esiintyy, voitaisiin se esittää näin:

$$\vec{v}\,' = SSD(\vec{v}) + \vec{d}$$

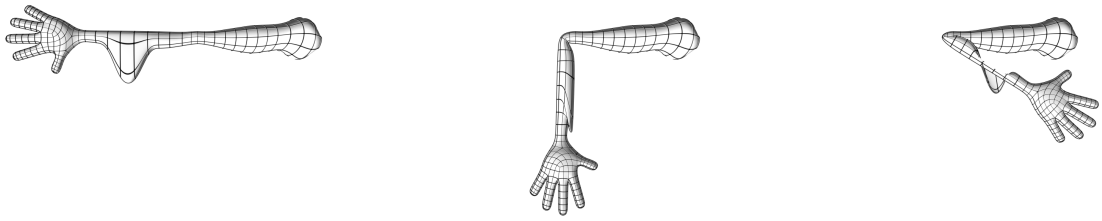
Tässä siis *v* on verteksin alkuperäinen paikka, *SSD()* (funktio) on yleisin animaatio-ohjelmissa käytetty tapa, millä hahmoa liikutetaan luiden avulla (skeletal subspace deformation), *d* (delta) on vektori, jolla morphi korjaa kyseistä verteksiä ja *v'* on lopullinen paikka, missä verteksi sijaitsee absoluuttisessa koordinaatistossa. Yhtälön ainoa tuntematon on *d*, mutta sen ratkaisu on helppoa.

$$\vec{d} = \vec{v}' - SSD(\vec{v})$$

Ongelmaksi muodostuu se, että *d* on tässä tapauksessa relatiivinen absoluuttiseen koordinaatistoon. Jos SSD-funktio kääntää hahmoa 180 astetta niin vektori *d* ei käänny mukana. Kuvassa 21 nähdään SSD-funktio n vaikutus malliin ja kuvassa 22 morphin vaikutus SSD-funktion jälkeen.



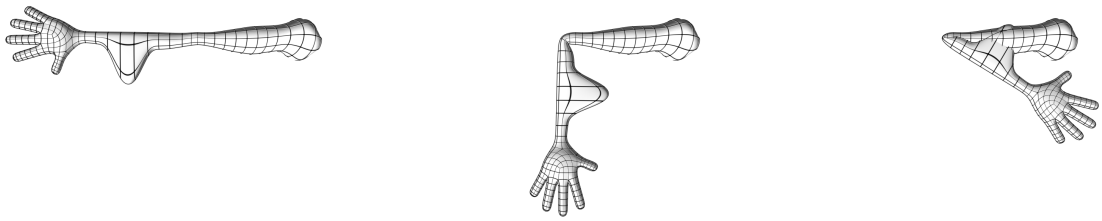
Kuva 21. SSD-funktion vaikutus malliin



Kuva 22. Morphin vaikutus SSD-funktion jälkeen

Jotta morphi kääntyisi mukana täytyy myös se ajaa SSD-funktion läpi, tällöin yhtälö muuttaa muotoaan. Kuvassa 23 on nähtävissä, miten ennen SSD-funktiota interpoloitu morphi käyttäytyy.

$$\vec{v}' = SSD(\vec{v} + \vec{d})$$



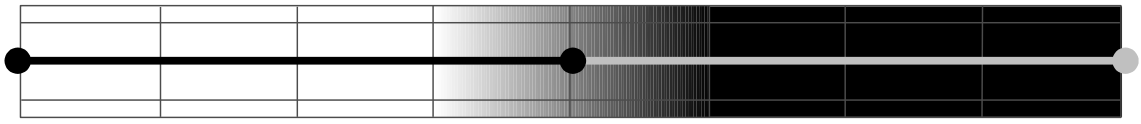
Kuva 23. Morphin vaikutus enen SSD() funktiota

Tästä *d*:n ratkaiseminen onkin jo vaikeampaa, sillä pitäisi tietää mikä on SSD-funktion käänteisfunktio.

$$\vec{v} + \vec{d} = SSDinverse(\vec{v}')$$

$$\vec{d} = SSDinverse(\vec{v}') - \vec{v}$$

Tutkitaan hieman SSD-funktion luonnetta yksinkertaisella sylinterillä ja kahdella luulla. Aloitetaan määrittelemällä luiden painoarvot eli kuinka eri luut vaikuttavat vertekseihin. Kuvassa 24 näkyy, että mustalla merkityn luun vaikutusalue on suurin valkoisella alueella ja vaaleanharmaan luun mustalla alueella. Harmaan alueen vaikutus jakaantuu molemmille luille. Käytännössä vaakasuunnassa keskimmäisiin vertekseihin molemmat luut vaikuttavat yhtä paljon ja kaikkiin muihin vertekseihin vaikuttaa ainoastaan jompi kumpi luista. Kunkin luun vaikutus verteksiin on arvo nollan ja yhden väliltä ja jokaisen luun yhteenlaskettu vaikutus verteksiin on normaalisti yksi.



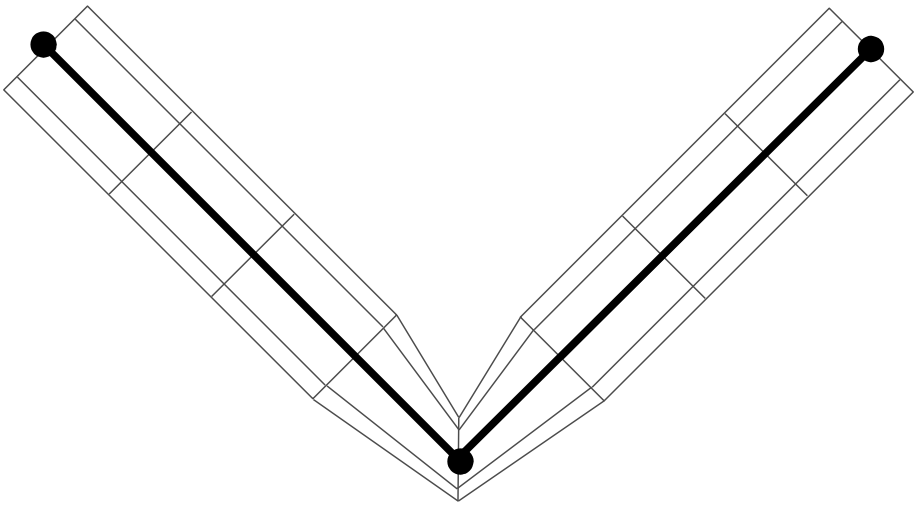
Kuva 24. Kahden luun vaikutuksen jakaantuminen malliin

SSD-funktio kuuluu seuraavasti:

$$\vec{v}' = SSD(\vec{v}) = \sum_{i}^{n} (w_i * T_i * (\vec{v} - \vec{t}_i)) + \vec{t}_i$$

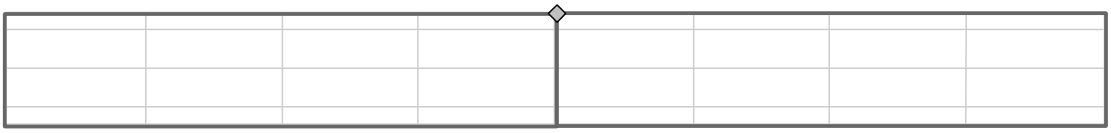
$$T_i = A_i * S_i$$

Verteksin alkuperäisestä paikkavektorista *v* vähennetään ensin luun sijainti nk. sitomisasennossa (bind pose, rest pose) *t*, jotta saadaan vektori, joka kuvastaa verteksin paikkaa suhteessa luuhun. Tämä vektori kerrotaan matriisilla *T* ja luun painoarvolla *w*. Matriisi *T* on luun nykyisen matriisin ja sitomisasennon käänteismatriisin tulo eli käytännössä matriisi, jonka avulla voidaan muuttaa vektoreita sitomisasennosta nykyiseen asentoon. Lopuksi lisätään vielä vektori *t,* joka vähennettiin vektorista *v* ennen matriisilla kääntöä. Nämä vaiheet toistetaan, kunnes jokainen luu on käyty läpi ja summattu lopputulokseen. Symboli *n* merkitsee luiden lukumäärää ja *i* luun indeksiä.



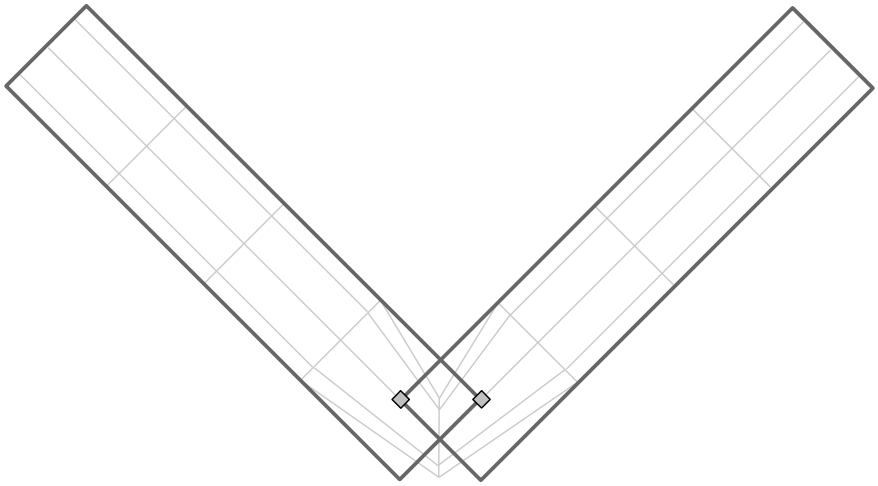
Kuva 25. SSD() funktion vaikutus kun luiden asento muuttuu

Esimerkin kiinnostavimmat verteksit sijaitsevat kuvan 24 harmaalla alueella, jossa molemmat luut vaikuttavat näihin vertekseihin yhtä paljon eli arvolla 0,5. Kuvassa 25 on nähtävillä mallin muuttuminen luiden mukana. Tarkastellaan tarkemmin luiden muutoksen vaikutusta harmaan alueen vertekseistä ylimpänä sijaitsevaan. Selkeyden vuoksi kuvassa 26 on verteksin paikka sitomisasennossa ja luut tumman harmaina nelikulmioina. Huomaa kuinka molemmista suorakulmioista yksi kulma sijoittuu tarkasteltavan verteksin kanssa samaan paikkaan.



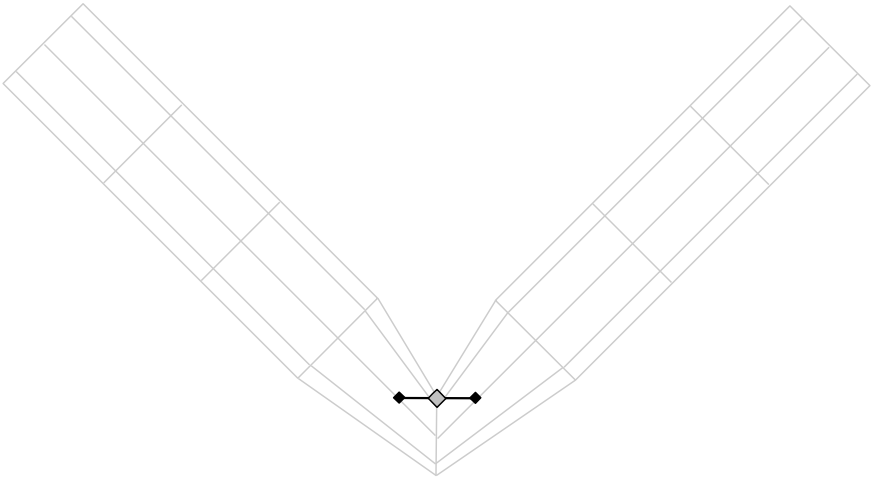
Kuva 26. Luut ja tarkasteltava verteksi

Käännetään luita siten, että niiden välille muodostuu 90 asteen kulma. Kunkin luun aiheuttama muutos verteksiin on helppo määritellä, kun käännämme luuta kuvastavaa suorakulmiota ja tarkastelemme, mihin sen kulma sijoittuu. Koska luita on kaksi, on verteksikin piirretty kahteen kertaan (kuva 27). Matemaattisesti nämä paikat selvitetään matriisien avulla.



Kuva 27. Luut ja tarkasteltava verteksi muuttuneessa asennossa

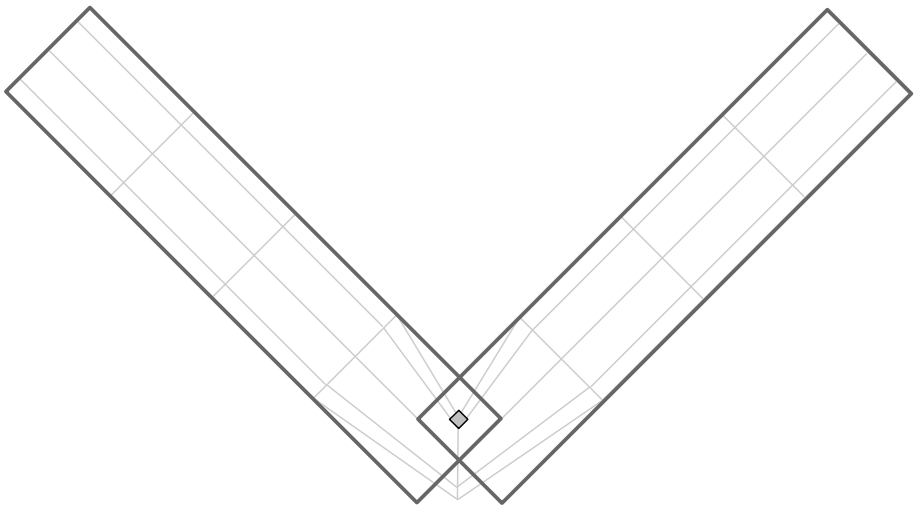
Koska luita on vain kaksi ja molempien painoarvo tarkasteltavan verteksin osalta on 0,5, on lopullinen verteksin paikka tasan kuvan 27 verteksien välissä. Kun piirretään jana näiden verteksien väliin ja katsotaan, missä sen keskikohta on, huomataan, että se on samassa paikassa kuin vaalean harmaalla piirretyn rautalankamallin verteksi (kuva 28).



Kuva 28. Verteksin lopullinen interpoloitu paikka

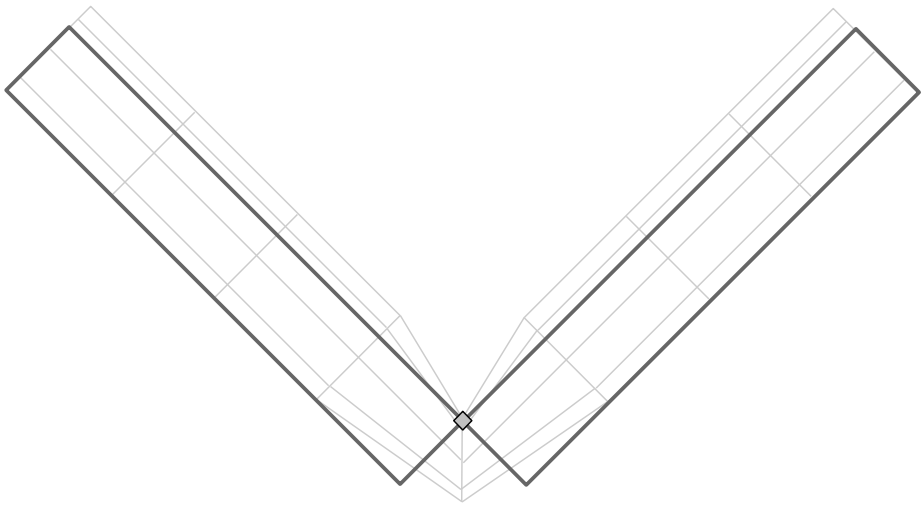
Tämä oli hieman yksinkertaistettu esimerkki SSD-funktion toiminnasta, sillä normaalisti 3d-ohjelmissa käytetään 4x4 matriiseja, jolla voidaan rotaation lisäksi määritellä tieto paikasta. Myöskään skaalausta ei otettu huomioon millään tavalla. Tämä esimerkki riittää kuitenkin kuvastamaan miten luiden asento vaikuttaa malliin, joten koetetaan kääntää malli tästä takaisin siihen, mistä lähdettiin. Ensimmäinen ero huomataan kun

piirretään samaiset suorakulmiot luiden tilalle ja verteksin nykyinen paikka, sillä suorakulmioiden kulmat eivät osu samaan kohtaan verteksin kanssa (kuva 29).



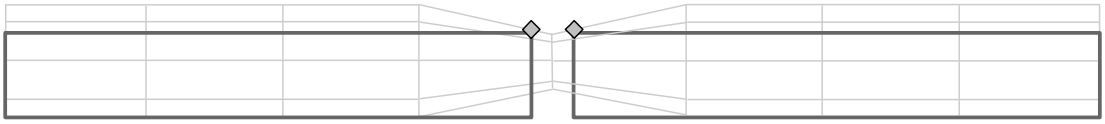
Kuva 29. Verteksin suhde luihin muuttunut SSD() funktion jälkeen

Visualisoidaan luut uusiksi niin, että kulmat osuvat verteksin päälle, jolloin saamme selkeämmän käsityksen, miten luiden kääntäminen takaisin vaikuttaa malliin (kuva 30). Luiden ja verteksien väliset painoarvot eivät ole muuttuneet.



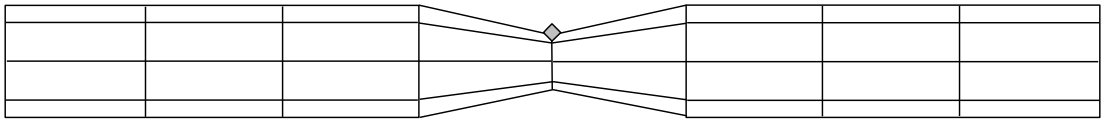
Kuva 30. Luiden uudelleen sitominen deformattuun malliin

Kun luut käännetään takaisin suoraan käyttäen yllä olevaa asentoa sitomisasentona, niin huomataan, että verteksi ei ole samassa paikassa kuin alkuperäisen sitomisasennon mallissa (kuva 31).



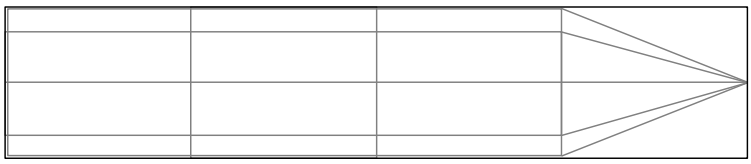
Kuva 31. SSD() funktion vaikutus jo aiemmin deformattuun malliin

Koko malli on hieman luhistunut niiltä osin, missä vertekseihin vaikuttavat molemmat luut (kuva 32).



Kuva 32. Mallin luhistuminen

Tästä voimme päätellä, ettei vektorin kääntäminen toimivaksi morphiksi ole aivan triviaalia ja vaikka tutkiskelemalla tätä kyseistä ongelmaa lisää pääsisimme jonkinlaiseen ratkaisuun, on mallin luhistuminen silti ongelma. Jos luut käännetään 90 asteen sijasta 180 asteen kulmaan, luhistuvat nämä keskimmäiset verteksit yhteen pisteeseen eli singulariteettiin (kuva 33).



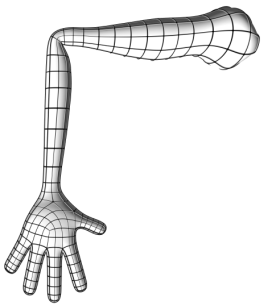
Kuva 33. Mallin luhistuminen luun 180 asteen käännöksen jälkeen

Point Animationissa käytetty tekniikka ei ratkaise singulariteetin tuomia ongelmia, toisin sanoen, jos Point Animation interpoloidaan ennen luita, on lopputulos sama kuin morphilla, joka vaikuttaa ennen luita. Eli periaatteessa SSD-funktion käänteisfunktion ratkaiseminen voisi mahdollistaa interpoloinnin ennen kuin luut vaikuttavat, mutta tämä ratkaisu toimisi vain SSD-funktion tapauksessa. Deformereita on monenlaisia ja niiden kaikkien käänteisfunktioiden selvittäminen on käytännössä mahdotonta.

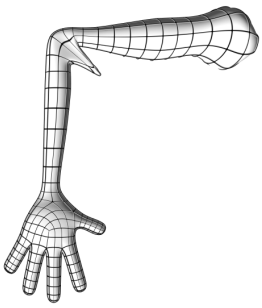
Yksi lähestymistapa, jota kaavailtiin käytettäväksi Point Animationissa, perustui siihen, että käyttäjä tai algoritmi valitsisi mihin kunkin verteksin siirtymävektori olisi relatiivinen. Tämä tarkoittaisi sitä, että vektorit käännettäisiin yhdellä matriisilla ja lisättäisiin malliin vasta SSD-funktion jälkeen.

$$\vec{v}\,' = SSD(\vec{v}) + M * \vec{d}$$

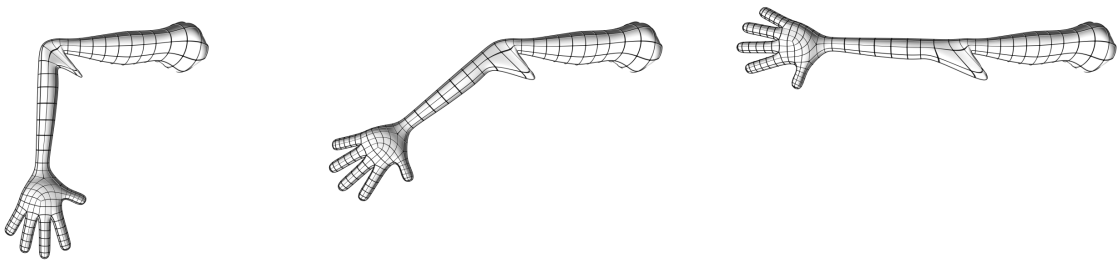
Ajatus siitä että käyttäjä itse joutuisi valitsemaan objektin joka tarjoaa sopivan koordinaatiston hylättiin epäkäytännöllisenä. Suunniteltu algoritmilla tapahtuva valinta taas olisi toiminut vain luu-deformaation kanssa, eli Point Animation olisi ollut hyödytön työkalu muiden deformereiden yhteydessä käytettynä. Algoritmi valitsee isoimman painoarvon omaavan luun matriisin koordinaatistoksi. Tällaisesta yhteen luuhun relatiivisuudesta aiheutuu epätoivottuja ominaisuuksia (kuva 34).



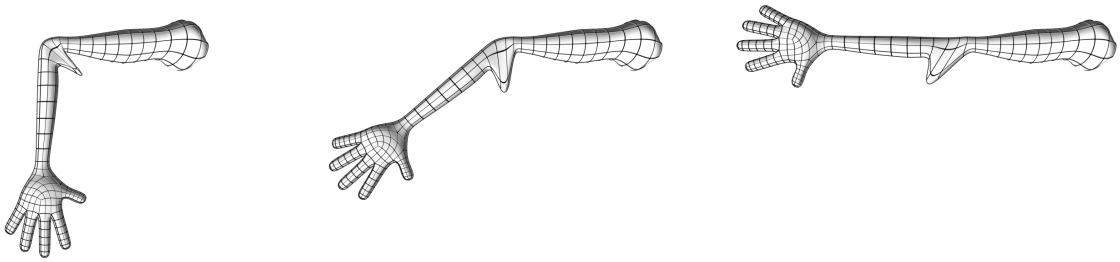
Deformattu malli



Deformatussa asennossa määritelty morphi



Olkavarteen relatiivisen morphin kääntö takaisin sitomistilanteeseen

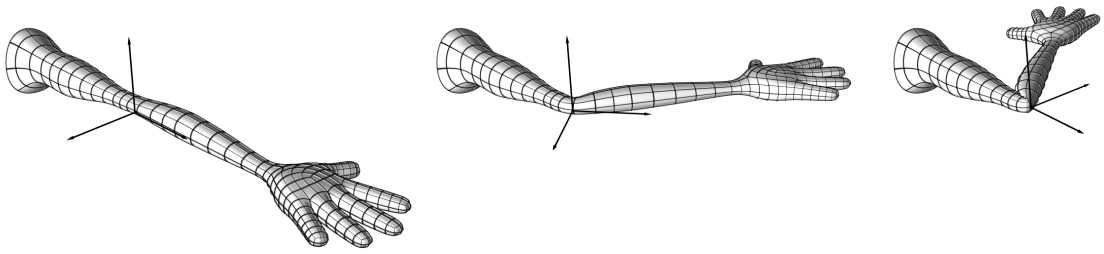


Kyynärvarteen relatiivisen morphin kääntö takaisin sitomistilanteeseen

Kuva 34. Yhteen luuhun relatiivinen morphi

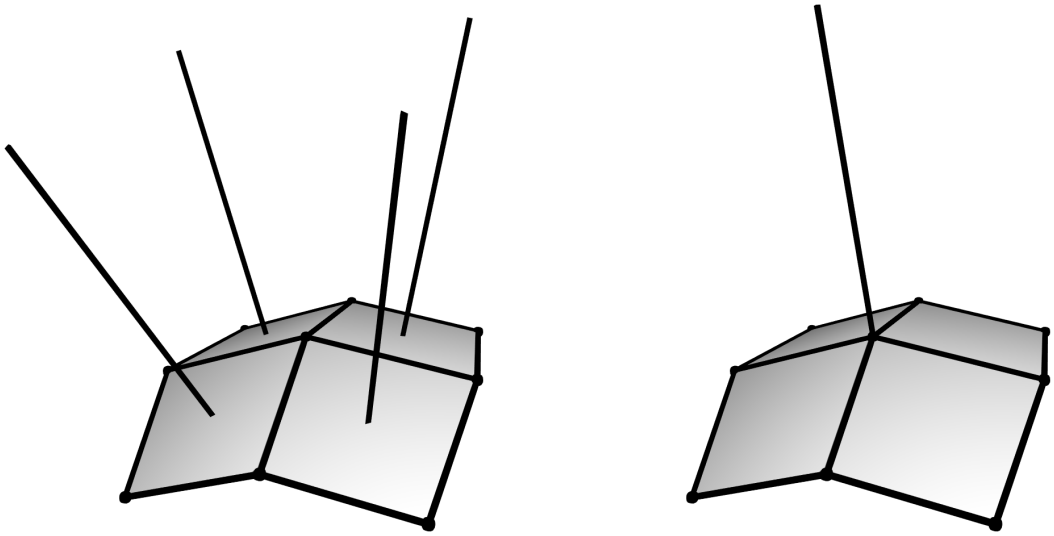
Point Animationin tapauksessa päädyttiin lopulta etsimään verteksille sopiva koordinaatisto deformatusta 3d-mallista itsestään. Tällaisen toimivan koordinaatiston löytäminen oli haastavin osuus Point Animationin kehityksessä. Useiden testien jälkeen päädyttiin kuitenkin tekniikkaan, joka muistuttaa hyvin paljon teksturoinnissa käytettävää tangent space normal mappingia. Käytännössä se mahdollistaa Point Animationin siirtymävektorien interpoloinnin jo deformatulla mallilla. Tekniikassa on joitakin rajoittavia tekijöitä, mutta jos välttää rikkomasta tiettyjä sääntöjä se mahdollistaa toimivan ja loogisen koordinaatiston, jossa vektorit sijaitsevat.

Tavoitteena oli löytää kolme toisiinsa nähden kohtisuorassa olevaa yksikkövektoria, jotka määrittelevät koordinaatiston. Luonnollinen valinta kahdeksi ensimmäiseksi vektoriksi olisivat pinnan suuntaisella tangenttitasolla sijaitsevat vektorit, jotka olisivat toisiinsa nähden 90 asteen kulmassa. Kolmas vektori saadaan näiden kahden vektorin ristitulosta. Oikeastaan tämä kolmas vektori on pinnan tangenttitason normaali eli kohtisuorassa pintaa nähden. Tangenttitason suunta ja paikka riippuu tarkasteltavasta pinnan pisteestä. Kuvassa 35 on koordinaatisto, joka elää kyynärpään mukana.



Kuva 35. Mallin pintaan relatiivinen koordinaatisto

Point Animationin tapauksessa verteksi on tällainen pinnan piste, jota tarkastellaan, mutta koska Messiahiin tuotavat 3d-mallit koostuvat polygoneista, on verteksin kohdalla pinta geometrisesti epäjatkuva. Verteksi voidaan mieltää osaksi useampaa lähes tasomaista pintaa, jotka kohtaavat tässä pisteessä. Verteksille voidaan kuitenkin laskea normaali, joka on yhdistelmä tai keskiarvo polygonien normaaleista (kuva 36).



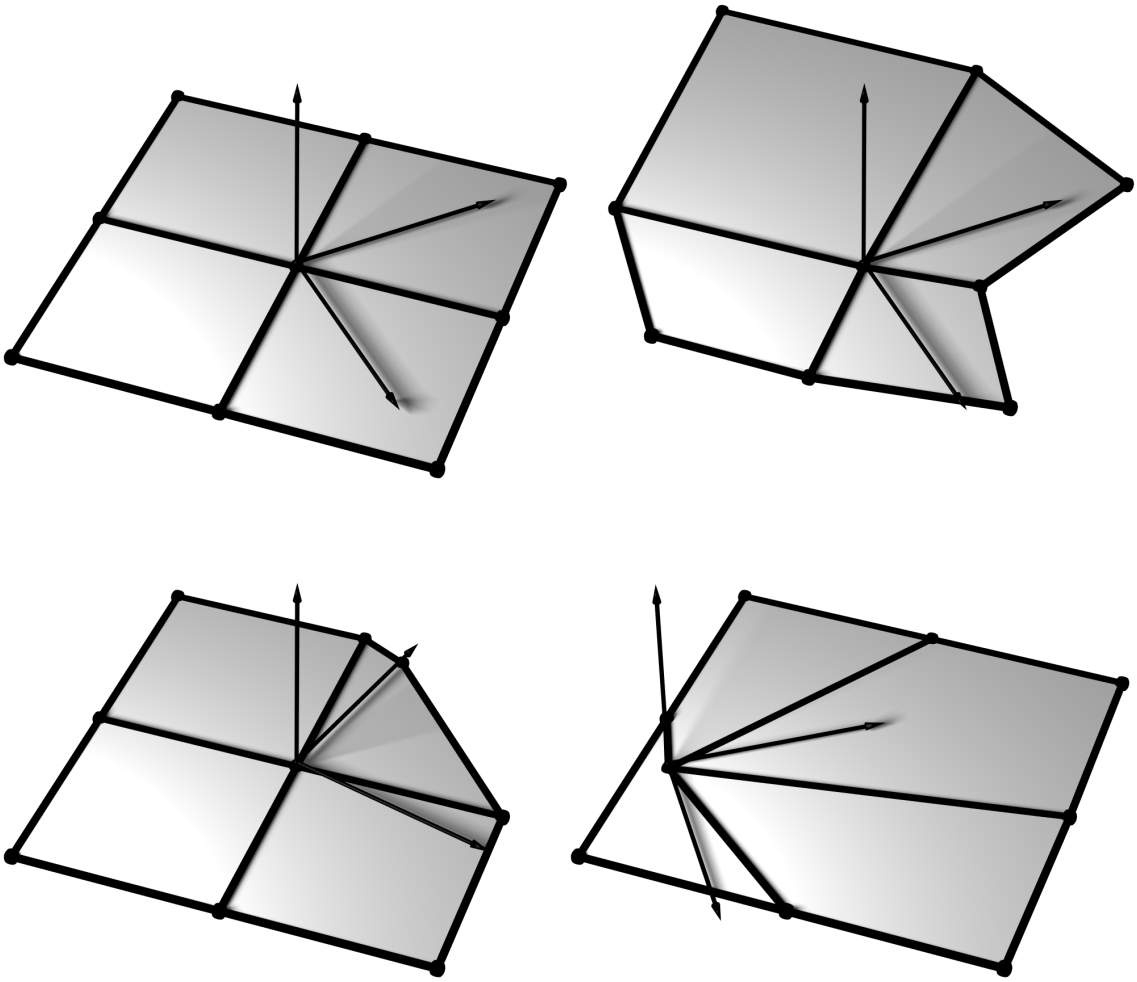
Kuva 36. Polygonien normaalit ja niistä laskettu verteksinormaali

Käytetyimmät polygonit 3d-grafiikassa ovat neli- tai kolmikulmaisia, mutta periaatteessa polygonisssa voi olla kuinka monta kulmaa tahansa. Oikeastaan polygon tarkoittaa kaksiulotteisia- eli tasokuviota, mutta 3d-grafiikassa mallit koostuvat polygoneista, joiden määritelmä on hieman löyhempi, eivätkä verteksit välttämättä sijaitse yhdellä tasolla. Polygonin normaali on kolmion tapauksessa helppo määritellä, sillä kolmio koostuu kolmesta verteksistä, jotka on yhdistetty toisiinsa. Melkein aina nämä kolme pistettä määrittelevät tason avaruudessa. Ainoastaan samalla viivalla sijaitsevat verteksit eivät muodosta tasoa ja tällöin ”kolmion” pinta-alakin on nolla. Jos yhdistävät viivat ajattelee vektoreina, ottaa niistä kahden ensimmäisen ristitulon ja normalisoi sen, on lopputuloksena normaalivektori, joka on kolmion määrittelemään pintaan nähden kohtisuorassa.

Kolmio (tri, trigon, triangle) on aina tasainen, mutta quadin (neljä kulmaa sisältävä polygoni, quadgon) kaikki verteksit eivät välttämättä sijaitse samalla tasolla. Normaalin määrittelemiseksi tällaisessa tapauksessa käydään polygonin kaikki verteksit läpi ja lasketaan ristitulo niistä lähteville reunavektoreille (edge). Haetaan eräänlainen keskiarvo, joka kuvastaa parhaiten polygonin normaalin yleistä suuntaa. Point Animationin tapauksessa verteksinormaaleja ei tarvinnut laskea itse, sillä ne laskettiin joka tapauksessa Messiahin toimesta ja olivat helposti saatavilla SDK:n kautta.

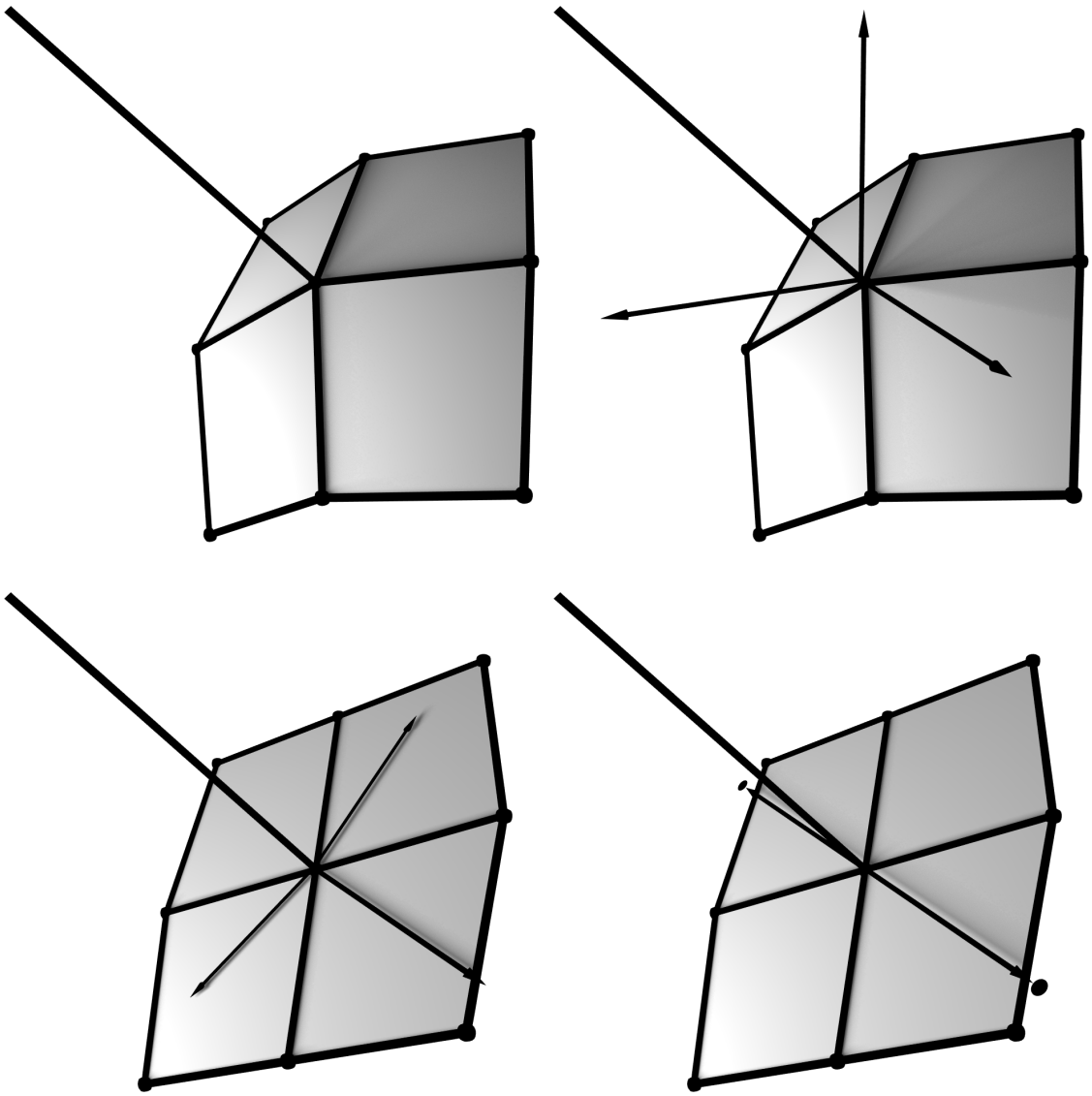
Verteksinormaali muodostaa koordinaatiston z-vektorin, jolloin x- ja y-vektoreiden täytyy sijaita tasolla, joka on kohtisuorassa verteksinormaaliin nähden. Teksturoinnissa, tai oikeastaan normal mappingissä, tangent spacen x- tai y-vektori haetaan UV-kartan avulla ja z-vektori on interpoloitu normaali kyseisessä pisteessä. Point Animationin kohdalla voitu luottaa siihen, että mallilla olisi UV-kartta, tai että kartassa ei olisi päällekkäisyyksiä. Koordinaatiston määrittelemiseksi tarvitaan z-vektorin ohella vain yksi vektori, jonka määrittelee verteksiä ympäröivä pinta. Ensimmäinen kokeilu oli vektori, joka kulki tarkasteltavasta vektorista johonkin sen naapuriverteksistä.

Tämä teki verteksin riippuvaiseksi kyseisestä naapuriverteksistä, eikä muiden naapuriverteksien liikkeet muuttaneet koordinaatistoa (kuva 37). Myöskään verteksistä kaikkiin naapurivertekseihin vedettyjen vektoreiden yhteenlasku ja normalisointi ei toiminut, sillä lopputulos saattoi olla nollavektori, jolla ei ole pituutta eikä suuntaa.



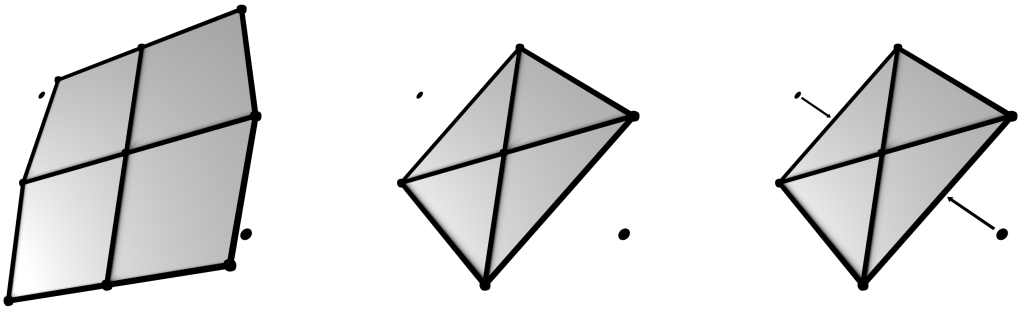
Kuva 37. Yhteen naapuriverteksiin relatiivinen koordinaatisto

Lopulta päädyttiin ratkaisuun, joka oli kaksivaiheinen; kerran suoritettava esivalmistelu ja joka ruudunpäivityksen yhteydessä toteutettava koordinaatiston laskeminen. Suurin osa työstä suoritetaan valmisteluvaiheessa ja ruudunpäivityksen yhteydessä tehtävien laskutoimitusten määrä on karsittu. Valmisteluvaiheessa absoluuttinen koordinaatisto ja verteksiä ympäröivä geometria litistetään verteksinormaalia pitkin tasoon. Litistetyistä absoluuttisen koordinaatiston vektoreista valitaan pisin ja merkitään muistiin mihin pisteeseen se päättyy. Tämän jälkeen vektori muutetaan negatiiviseksi, jolloin se menee vastakkaiseen suuntaan ja merkitään muistiin sen päätepiste (kuva 38).



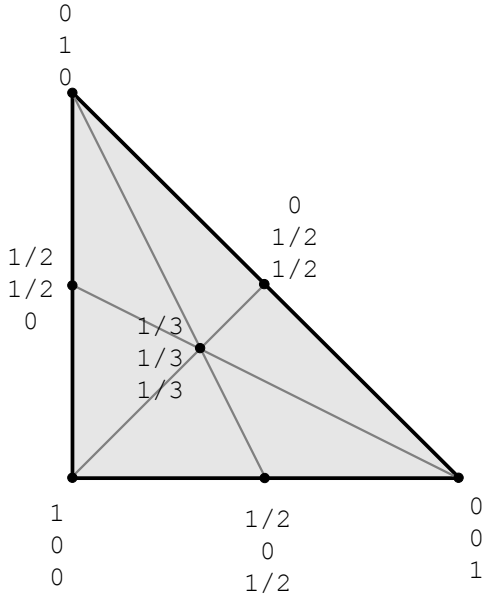
Kuva 38. Pinnan ja absoluuttisen koordinaatiston litistäminen verteksinormaalia pitkin

Tämän jälkeen haetaan mallista kaikki tarkasteltavaan verteksiin suoraan yhteydessä olevat naapuriverteksit. Samaan polygoniin kuuluvista vertekseistä muodostetaan kolmioita, jotka muodostavat yksinkertaisen yhdessä tasossa sijaitsevan pinnan. Tältä pinnalta etsitään pisteet, jotka ovat lähinnä aiemmin etsittyjen vektorin ja sen negaation päätepisteitä. Merkitään muistiin mitkä verteksit muodostavat kolmiot, joilla pinnalta löydetyt pisteet sijaitsevat (kuva 39).



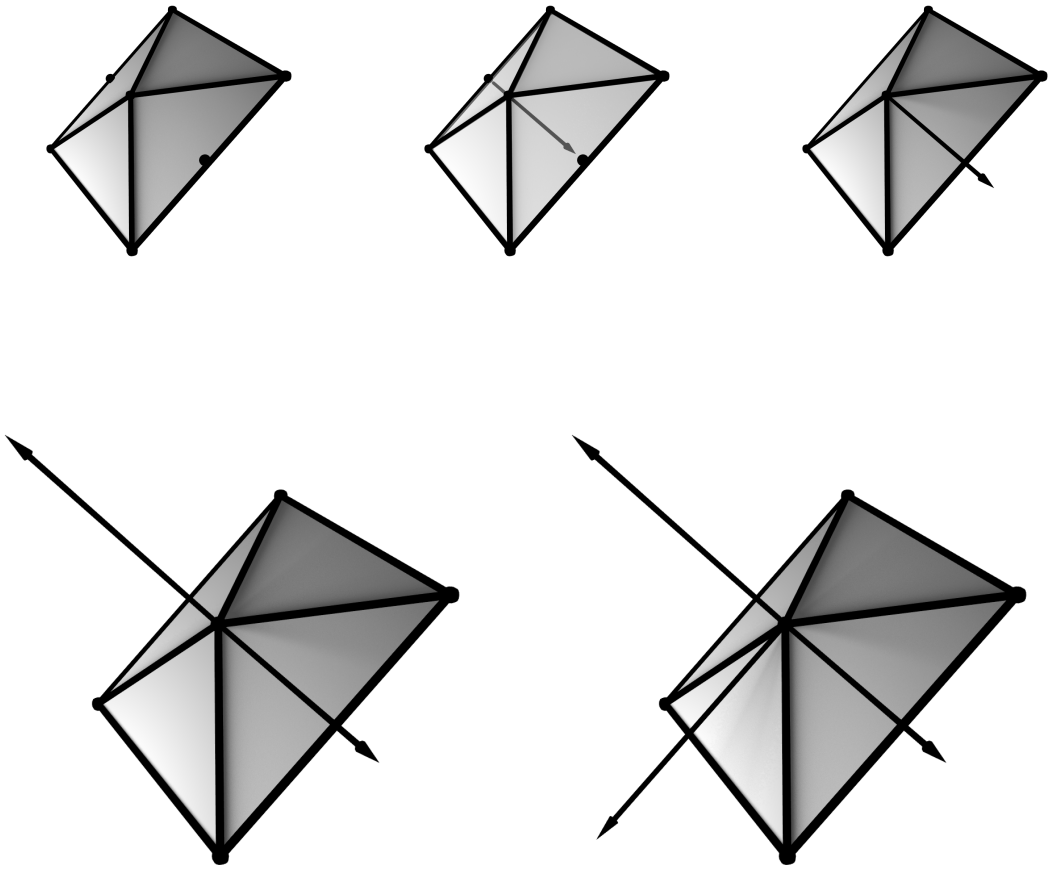
Kuva 39. Verteksiä ympäröivän pinnan yksinkertaistaminen kolmioihin ja niistä lähimpien pisteiden löytäminen

Tämän jälkeen määritellään barysentriset koordinaatit kolmioissa. Barysentriset koordinaatit ovat laaja aihe, mutta tässä tapauksessa riittää tieto: kun verteksien paikkavektorit kerrotaan niitä vastaavilla koordinaateilla, saadaan koordinaatteja vastaava paikka kolmiossa laskettua vaivatta. Eli barysentrisillä koordinaateilla voidaan määritellä pisteen paikka kolmion kärkipisteiden suhteen. Kuvassa 40 on esimerkkejä erilaisista barysentrisistä koordinaateista.



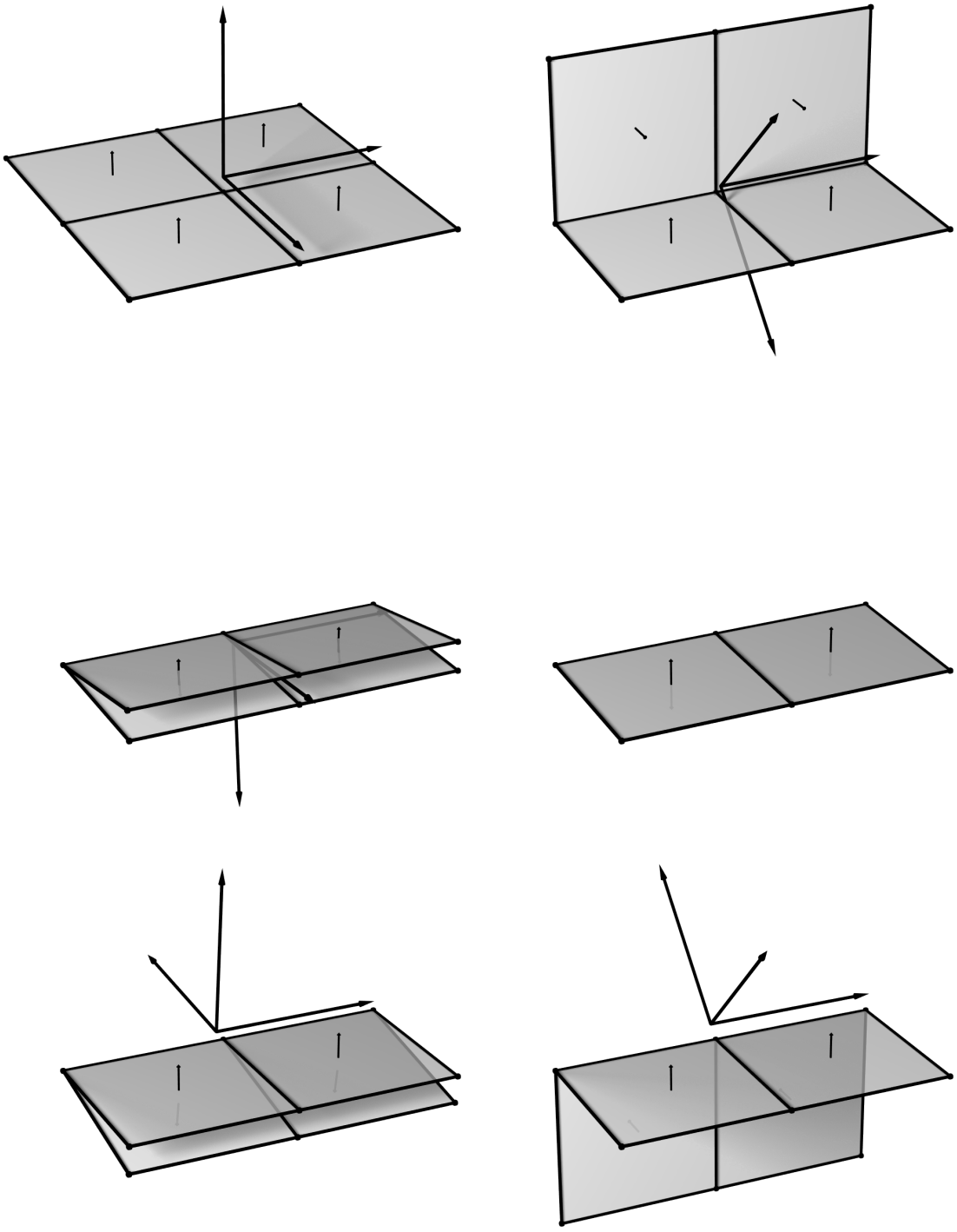
Kuva 40. Esimerkkejä barysentrisistä koordinaateista

Valmisteluvaiheesta saatujen kolmioden ja barysentristen koordinaattien avulla lasketaan joka päivityksen yhteydessä aiemmin haettujen pisteiden paikat. Muodostetaan vektori, joka alkaa ensimmäisestä pisteestä ja päättyy toiseen. Vektori normalisoidaan ja tehdään siitä koordinaatiston x-vektori. Z-vektoriksi valitaan sen hetkinen verteksinormaali ja lasketaan y-vektori x- ja z-vektorien ristitulolla. Tangent space -koordinaatisto verteksille on valmis (kuva 41). Kaikki tämä tietenkin toistetaan mallin jokaiselle verteksille, jolloin jokaisella on oma uniikki koordinaatistonsa, joka on suhteessa ympäröivän pinnan deformaatioon.



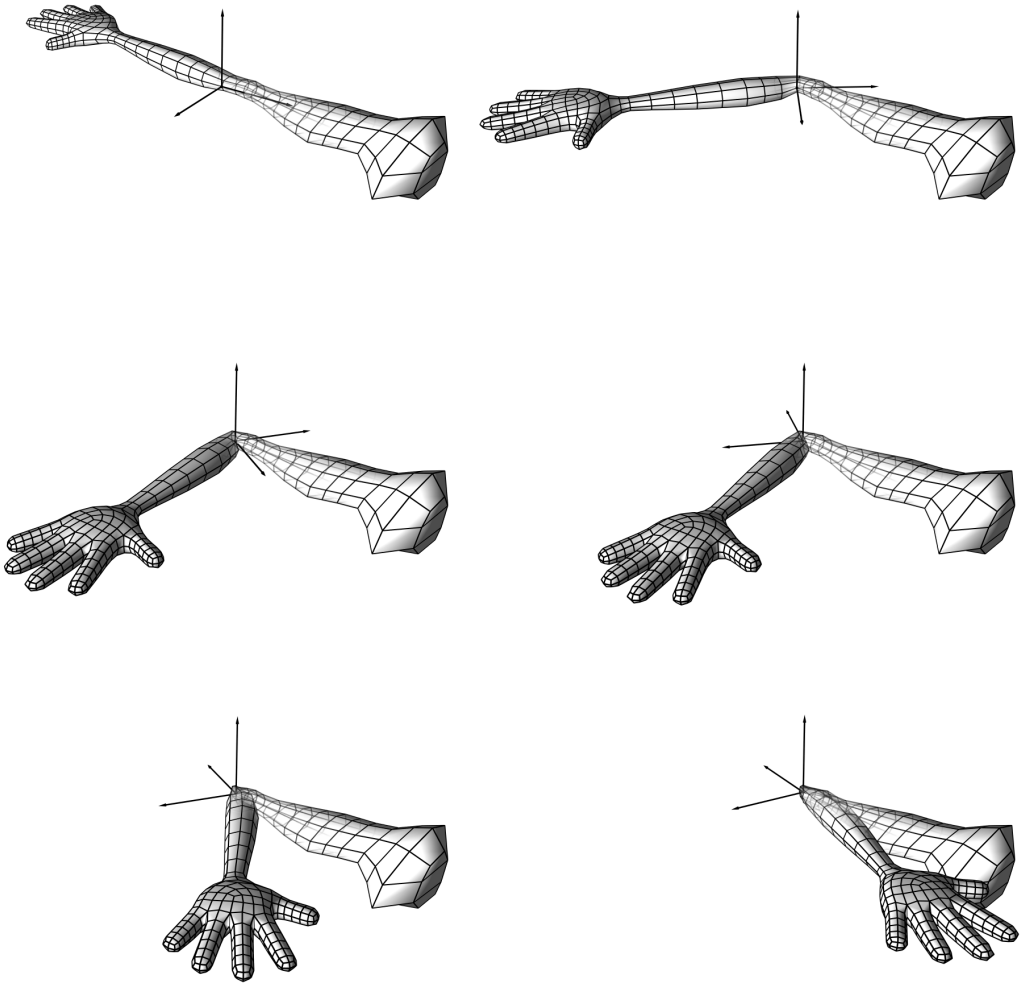
Kuva 41. Tangentspacen muodostaminen

Tangent spacen luonteeseen kuuluu koordinaatiston äkillinen ympäri pyörähtäminen. Tämä johtuu siitä, että verteksinormaali kiepsahtaa äkisti 180 astetta. Verteksinormaalin sekoaa, kun polygonit liikkuvat vastakkain ja yli toisistaan, jolloin polygonien normaalien summa lähestyy nollavektoria ja sen ylittäessään vaihtaa äkisti suuntaa.



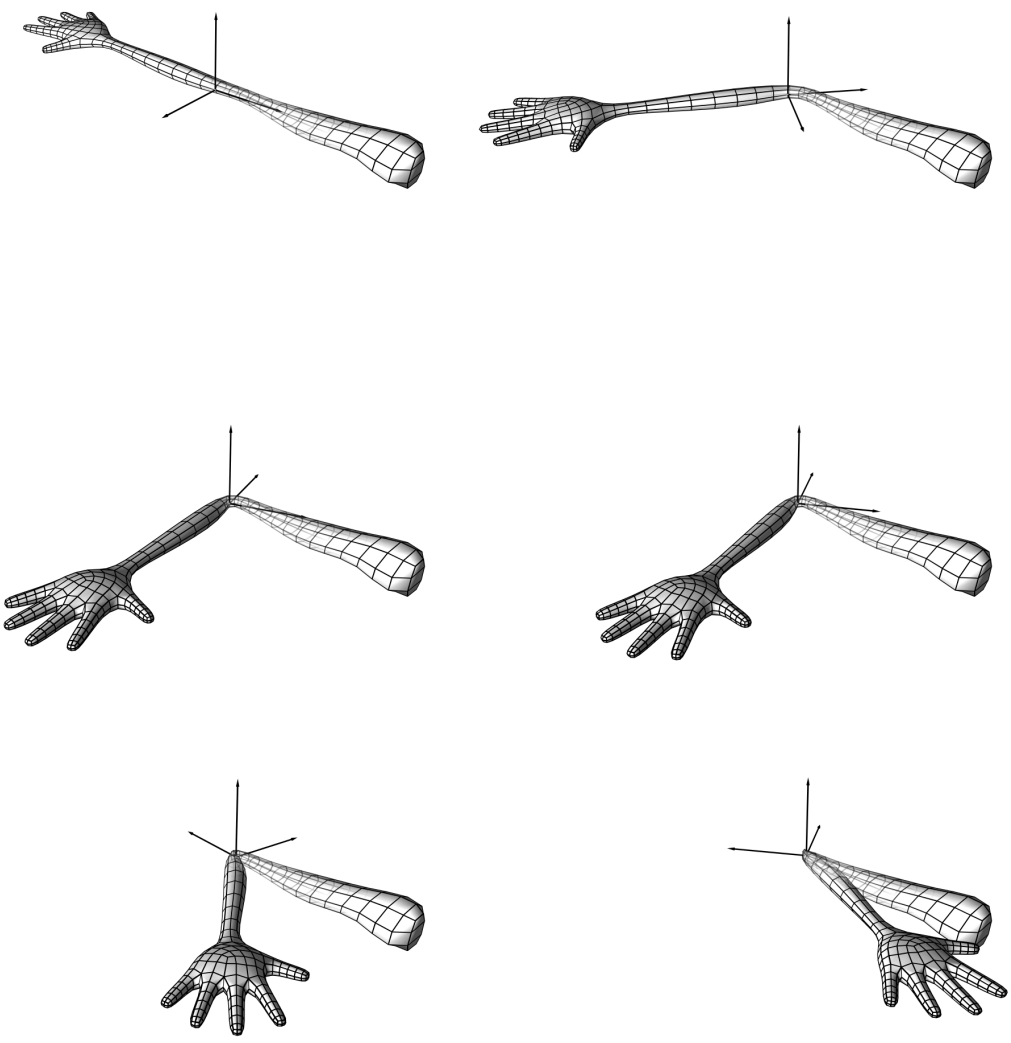
Kuva 42. Verteksinormaalin ja koordinaatiston äkillinen suunnan muuttuminen

Kuvan 42 tilanne on pahin mahdollinen: verteksinormaali on hetkellisesti nollavektori. Nollavektorin kaikki komponentit ovat nollia, joten sillä ei ole suuntaa eikä pituutta. Käytännössä verteksinormaali on harvoin nollavektori, mutta äkilliset suunnan vaihdokset esimerkiksi kyynärtaipeessa ovat ongelma (kuva 43).



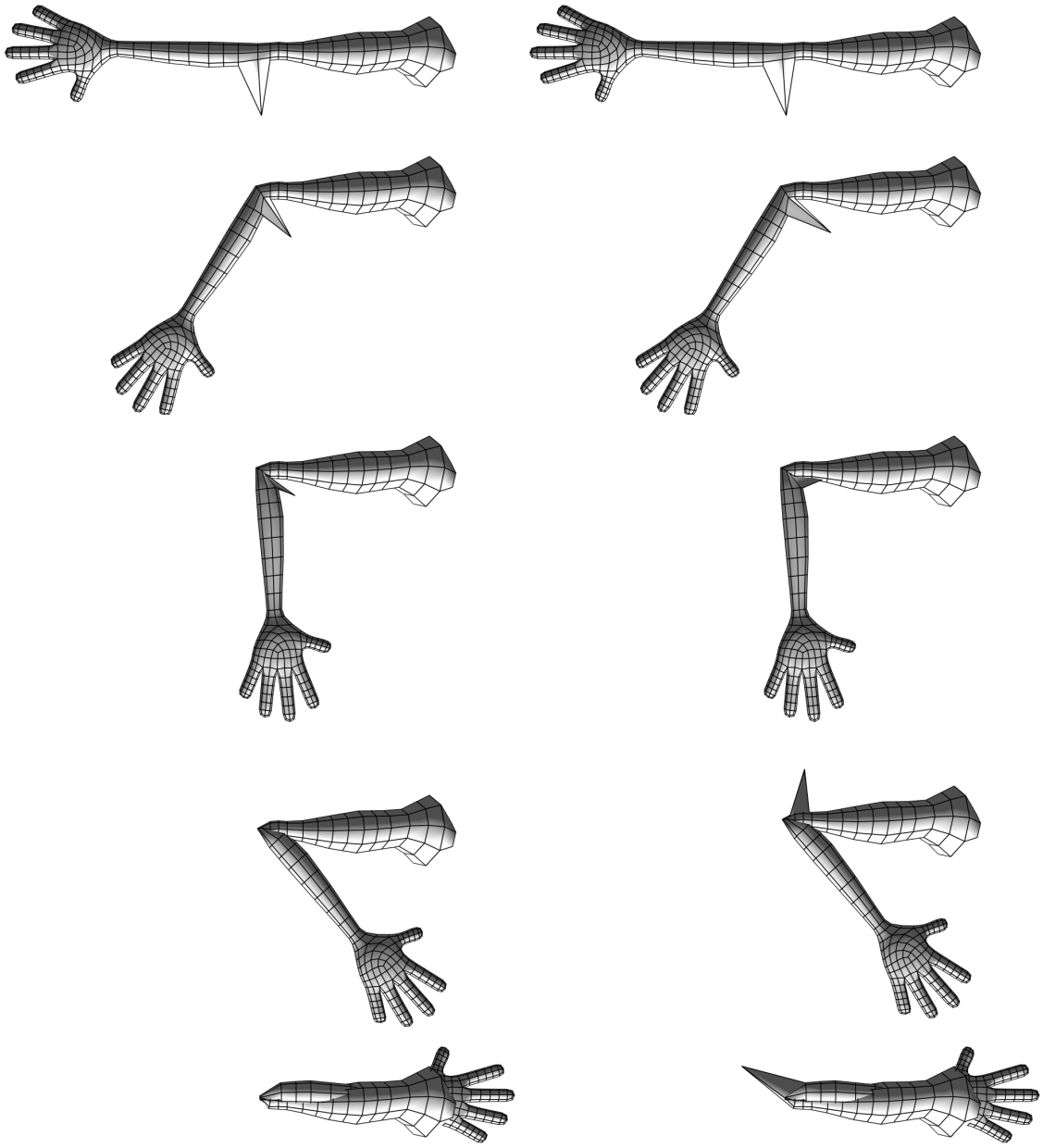
Kuva 43. Koordinaatiston äkillinen suunnan muuttuminen kyynärtaipeessa

Useiden paperille piirreltyjen käppyröiden jälkeen ratkaisu tähän ongelmaan oli yksinkertainen – relax. Relax-algoritmi kirjaimellisesti rentouttaa geometrian. Rentouttamalla geometriaa päästään ongelmallisista itsensä sisään kääntyvistä osista eroon. Relax-algoritmi ei toimi äärimmäisissä tapauksissa, mutta mahdollistaa kuitenkin laajemman liikeradan niveliin ja pehmentää koordinaatiston pyörähtämistä. Kuvassa 44 malli on ajettu relax-algoritmin läpi ennen koordinaatiston muodostusta.



Kuva 44. Koordinaatiston käyttäytyminen relax-algoritmin läpi ajetulla mallilla

Täytyy huomata, että tällainen relax-toimenpide suoritetaan Point Animationin sisällä sen omia laskutoimituksia varten, eikä käyttäjä näe sitä. Käytännössä verteksinormaalin äkilliseen suunnan vaihtumiseen näillä ratkaisuilla törmää hyvin harvoin, joten tämän takia tähän päädyttiin Point Animationin tapauksessa. Suunnan vaihtelua voi välttää ottamalla sen huomioon jo mallinnus- ja riggaus-vaiheissa. Äärimmäisessä tapauksessa normaali morphi luhistuu, joten molemmissa tekniikoissa törmätään ongelmiin yleensä samanlaisessa tilanteessa. Kuvassa 45 morphin ja Point Animationin ero lähtötilanteen ollessa samanlainen ja kyynärvarren kääntyessä 180 astetta.



Kuva 45. Morphin (vasemmalla) ja Point Animationin ero

6 YHTEENVETO

Point Animation julkaistiin osana Messiahia ja se vastaanotettiin suurella mielenkiinnolla. Vaikkakin ohjelma kaipaa lukusia parannuksia ja pientä viilausta, on sen potentiaali kuitenkin saanut Messiah-käyttäjän innostumaan siitä. Palaute on ollut lähinnä positiivista ja rakentavaa sekä erityistä kiitosta on tullut intuitiivisuudesta. Point Animationin kehitys ei ole enää yhtä aktiivista, mutta pientä edistystä tapahtuu aika ajoin.

Produktioissa Point Animationia on onnistuneesti käytetty studion sisällä jo jonkin aikaa, mutta jopa muutama käyttäjän tuotos on ollut näkyvillä, missä Point Animationia on hyödynnetty. Nopeissa mainosproduktioissa aikaa on usein vähän ja yleensä rigit jäävät toivottua kankeammiksi. Tällaisissa tapauksissa Point Animation on korvannut perinteisen tavan korjata virheet morpheilla.

Point Animation on kehityksen myötä paisunut vaikeasti hallittavaksi kasaksi koodia, joten jatkon kannalta koodin eri palasten pilkkominen helppokäyttöisiksi komponenteiksi on edessä jossain vaiheessa. Järkevän ohjelmistosuunnittelun saralla on vielä parannettavaa, mutta vähintään ohjelmoinnin perusteet tulivat selväksi projektin aikana.

Myös projektissa tarvitun matematiikan omaksuminen on avartanut ymmärrystä 3d-ohjelmien käyttäytymisestä ja sitä miten niissä esiintyviä ongelmia lähtee ratkaisemaan. Tarvittavan työkalun uupuessa sen kehittäminen itse ei ole enää vain kaunis ajatus.

LÄHTEET

Bolton, David. Definition of SDK. [verkkodokumentti]. Saatavuus <http://cplus.about.com/od/glossar1/g/sdkdefinition.htm> (luettu 6.6.2010).

Dunn, Fletcher & Parberry, Ian 20020. 3D Math Primer for Graphichs and Game Development. Texas: Wordware Publishing.

Meade, Tom & Arima, Shinsaku 2004. Maya 6: The Complete Reference. California: McGraw-Hill Professional.

Pandzic, Igor & Forchheimer, Robert 2002. MPEG-4 facial animation: the standard, implementation and applications. West Sussex, Englanti: John Wiley and Sons.