KARELIA-AMMATTIKORKEAKOULU
Tietotekniikan koulutusohjelma

Juuso Kuokkanen

WEBGL-RAJAPINTA JA 3D-GRAFIIKKAKIRJASTON LUOMINEN

Opinnäytetyö
Toukokuu 2015

Karelia AMMATTIKORKEAKOULU

OPINNÄYTETYÖ
Toukokuu 2015
Tietotekniikan koulutusohjelma

Karjalankatu 3
80200 JOENSUU
(013) 260 600

Juuso Kuokkanen

Nimeke
WebGL-rajapinta ja 3D-grafiikkakirjaston luominen

Toimeksiantaja
Karelia-amk

Tiivistelmä

Opinnäytetyössä tutkittiin 3D-grafiikkaa ja sen ohjelmointia WebGL-ohjelmointirajapinnan avulla, ja opinnäytetyön tuotoksena tuotettiin WebGL-rajapintaa hyödyntävä 3D-grafiikkakirjaston prototyyppi, jolla voi luoda 3D-grafiikkaa hyödyntäviä sovelluksia, jotka toimivat selaimessa. Aihe valittiin opinnäytetyön tekijän henkilökohtaisesta kiinnostuksesta 3D-grafiikkaa kohtaan.

Työssä tutkitaan 3D-grafiikkaan liittyviä matemaattisia käsitteitä ja laskutoimituksia sekä selvitetään, kuinka niitä voidaan toteuttaa ohjelmoimalla luodun kirjaston käyttöön. Määritettyjä matemaattisia luokkia hyödynnetään 3D-avaruudessa suoritettavien transformaatioiden määrittelyssä. Työssä tutustutaan WebGL-ohjelmointirajapinnan keskeisiin käsitteisiin ja toimintoihin laitteistokiihdytettyä 3D-grafiikkaa luodessa, ja opinnäytetyössä kehitetään tapa hyödyntää rajapinnan toiminnallisuuksia kirjastossa määritettyjen luokkien ja funktioiden avulla. Työssä selvitetään, kuinka COLLADA-tiedostoja voidaan hyödyntää 3D-mallien siirtämiseen ohjelmistojen välillä, ja kirjasto määrittelee tavan mallien lataamiseen kirjaston käyttöön COLLADA-tiedostoja hyödyntäen. Työssä tutkitaan piirrettävien 3D-mallien teksturointia ja sekä 3D-avaruuteen määritettävien valojen määrittelyä.

Opinnäytetyön tuloksena kertyi mittavasti tietoa 3D-grafiikasta ja sen ohjelmoinnista. Opittuja asioita hyödynnettiin 3D-grafiikkakirjaston prototyypin luomisessa, millä tutkittuja asioita voidaan toteuttaa käytännössä ja jota voidaan hyödyntää 3D-grafiikaa käyttävissä selainsovelluksissa. Työ toteuttaa suurimman osan tavoitteista, joita työlle oli suunnitelmassa asetettu, ja se tarjoaa hyvän pohjan jatkokehitystä varten.

Kieli
suomi

Sivuja 129
Liitteet 3
Liitesivumäärä 33

Asiasanat
3D-grafiikka, WebGL-ohjelmointirajapinta, COLLADA, 3D-grafiikkakirjasto,

THESIS
May 2015
Degree Programme in Information Technology
Karjalankatu 3
FI 80200 JOENSUU
FINLAND
+358 13 260 600

Author(s)
Juuso Kuokkanen

Title
WebGL Interface and Development of 3D Graphics Library

Commissioned by
Karelia University of Applied Sciences

Abstract

In this thesis, 3D graphics and 3D graphics programming were studied by using WebGL application programming interface. As a product of the thesis, a prototype of a 3D graphics library was developed, which uses the functionalities of the WebGL API to create an easier way to develop web-applications that use 3D-graphics. The subject was picked due to the author's personal interests towards 3D-graphics.

Mathematic subjects and calculations related to 3D graphics programming were studied in this thesis, and a way was developed to perform them in programmed form for the use of the developed library. Defined mathematical classes are used for defining transformations that are performed in 3D space. The thesis studied the basic concepts and functionalities of the WebGL API for creating hardware accelerated 3D graphics, and a way to use these functionalities of the interface was developed with the classes and functions of the created library. The thesis explains how COLLADA file format can be used for transferring 3D models between programs. The library develops a way to load a model from COLLADA-files to be used with the developed library. How 3D models can be textured and how lights can be defined into 3D space were also studied in this thesis.

As a result of the thesis, a great deal of information was obtained about 3D graphics and programming of graphics. The things learned were used to create a prototype of a 3D graphics library, which can be used to test the learned theories in practice, and to create web-programs that use 3D graphics. The thesis managed to achieve most of the original goals, and it provides a good template for further development.

Language
Finnish

Pages 129
Appendices 3
Pages of Appendices 33

Keywords

3D graphics, WebGL application programming interface, COLLADA, 3D graphics library

Sisältö

Liitteet

Lyhenteet

API	Application programming interface, suomeksi ohjelmointirajapinta, on määritelmä, joka sisältää toiminnallisuuksia, protokollia ja työkaluja sovellusten valmistukseen.
COLLADA	COLLAborative Design Activity on Khronos Groupin hallitsema XML-skeema, jolla voidaan siirtää 3D-grafiikan eri resursseja eri sovellusten välillä.
GLSL	OpenGL shading language, OpenGL-rajapinnalle määritetty ohjelmointikieli rajapinnan käyttämien varjostinsovellusten luomiseen.
HTML5	HyperText Markup Language, HTML-merkintäkielen viides versio, jota käytetään verkkosivujen luomiseen.
IBO	Index Buffer Object, indeksitietoa sisältävä puskuri grafiikkaohjelmoinnissa.
OpenGL ES	Open Graphics Library for Embedded Systems, Khronos Groupin kehittämä ja hallitsema ohjelmointirajapinta 3D tietokonegrafiikan luomiseen sulatetuissa järjestelmissä, kuten puhelimissa.
VBO	Vertex Buffer Object, verteksidataa sisältävä puskuri grafiikkaohjelmoinnissa.
WebGL	Web Graphics Library, Khronos Groupin kehittämä ja hallitsema JavaScript-rajapinta reaaliaikaisen 3D tietokonegrafiikan luomiseen selaimessa.

1 Johdanto

Tämä dokumentti on Karelia-ammattikoulun tietotekniikan opintojen opinnäytetyön raportti, joka käsittelee opinnäytetyönä valmistetun JavaScript-kielisen WebGL-rajapintaa hyödyntävän 3D-grafiikkakirjaston valmistusta. Opinnäytetyön tuotoksena saadun prototyyppiasteen 3D-grafiikkakirjaston määrittelemien toiminnallisuuksien avulla on tarkoitus pystyä luomaan nopeammin selaimessa toimivia 3D-grafiikkaa hyödyntäviä sovelluksia ilman, että käyttäjän tarvitsee huolehtia WebGL-rajapinnan matalan tason toiminnoista. Aihevalinta perustuu henkilökohtaiseen kiinnostukseen 3D-grafiikkaohjelmointia kohtaan.

Työ pyrkii kattamaan yleisimpien 3D-grafiikkaan liittyvien toimintojen läpikäynnin. Näihin lukeutuvat 3D-grafiikkaan liittyvät yleiset matemaattiset käsitteet, ja niiden toteuttaminen ohjelmoimalla, kamerat 3D-grafiikassa, 3D-mallien lataaminen sovellukseen tiedostosta, 3D-mallien teksturointi sekä erilaisten valolähteiden luominen. Opinnäytetyössä pyritään luomaan kattava katsaus 3D-grafiikkaan liittyvistä käsitteistä, WebGL-rajapinnasta ja hyödyntämään opittuja tietoja modulaarisen 3D-grafiikkakirjaston luomisessa, millä tutkittuja asioita voidaan soveltaa 3D-grafiikkaa hyödyntävissä selainpohjaisissa sovelluksissa.

Opinnäytetyö ja sen raportti on tarkoitettu 3D-grafiikasta ja 3D grafiikanohjelmoinnista kiinnostuneille sovelluskehittäjille. Työ käsittelee laajasti 3D-grafiikkaan liittyvää matematiikkaa, WebGL-rajapinnan toimintaa ja paljon erilaisia 3D-grafiikkaan liittyviä käsitteitä.

2 3D-grafiikan yleiset matemaattiset käsitteet ja niiden toteutus

Kirjasto määrittelee joukon matemaattisia luokkia ja funktioita, joiden tarjoamia toiminnallisuuksia voidaan hyödyntää 3D-grafiikan muodostuksessa. Seuraa-

vissa luvuissa käydään läpi, millaisia erilaisia matemaattisia käsitteitä ja laskutoimituksia kirjastossa on määritelty ja miten ne on toteutettu osaksi kirjastoa.

2.1 Vektorit ja niiden operaatiot

Tietokonegrafiikassa käsitellään tietoa geometrisista kohteista, jotka sijaitsevat usein kaksi- tai kolmiulotteisessa avaruudessa. Pisteiden sijainteja avaruudessa esitetään koordinaattien avulla. 3D-avaruudessa nämä koordinaatit koostuvat kolmesta eri arvosta, jotka määrittelevät pisteen sijainnin x- y- ja z-akseleiden suuntaisesti. (Puhakka 2008, 29–31.)

Vektorit muodostuvat myös koordinaateista avaruudessa, mutta vektori merkitsee hieman eri asiaa kuin piste avaruudessa. Vektorit määrittelevät siirtymän paikasta a paikkaan b, eli vektorilla voitaisiin esittää kahden pisteen välistä siirtymää. Vektoreilla on kaksi suuretta, suuruus ja suunta. Suuruus määrittää matkan, joka siirrytään, ja suunta määrittää suunnan, johon kuljetaan. (Dunn & Parberry 2011, 35.)

Vektoria esittävä muuttuja kaavassa merkitään lihavoituna pienaakkosena, ja vektorin komponenttien arvot esitetään hakasulkujen sisässä olevalla yksiulotteisella taulukolla kaavan 1 mukaisesti. Vektorin muuttujan merkintätapaa käytetään raportissa esitetyissä kaavoissa, paitsi myöhemmän luvun 2.3 kvaternio esimerkeissä, joissa vektorin päälle lisätään nuolimerkintä, jotta vektori ja kvaternio voitaisiin helpommin erottaa toisistaan kaavoissa.

$$\boldsymbol{v} = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \tag{1}$$

Kirjastossa määriteltyä Vector3f-luokkaa käytetään kuitenkin sijaintien ja siirtymisen määrittämiseen kirjaston toiminnallisuuksien kanssa. Sillä voidaan määritellä eri kappaleiden sijainteja 3D-avaruudessa, se voi määrittää suunnan, johon jokin asia osoittaa avaruudessa ja sitä käytetään myös muihin tarkoituksiin. Esimerkiksi vektoreita käytetään valojen kanssa pinnalle valaistuksen seurauk-

sena muodostuvien värien määrittämiseen normaalivektoreiden ja valon kulkusuunnan määrittävän vektorin avulla.

Kirjastossa määritelty Vector3f-luokka (WEBGL_LIB.Math.Entities.Vector3f) määrittelee kolme jäsenmuuttujaa: x-, y- ja z-muuttujat. Näitä muuttujia käytetään esittämään vektorin määrittämä sijainti tai suunta 3D-avaruudessa avaruuden x-, y- ja z-akseleiden mukaisissa suunnissa. Kun Vector3f-luokan objekti luodaan, annetaan kullekin koordinaatille sijoitettava arvo parametrina.

```
WEBGL_LIB.Math.Entities.Vector3f = function(x, y, z){
    this.x = x;
    this.y = y;
    this.z = z;
};
```

2.1.1 Vektorin peruslaskutoimituksia

Vektorin negaatio tapahtuu muuttamalla vektorin jokainen komponentti käänteiseksi arvoksi (Dunn & Parberry 2011, 44). Kaava 2 esittää vektorin negaation suorittamisen.

$$-\begin{bmatrix} 4 \\ 8.5 \\ -3 \end{bmatrix} = \begin{bmatrix} -4 \\ -8.5 \\ 3 \end{bmatrix} \quad (2)$$

Tuloksena saatava vektori on vastakkaisen suuntainen verrattuna alkuperäiseen, mutta molempien vektoreiden siirtymien suuruus on sama (Dunn & Parberry 2011, 44–45). Kirjastossa vektorin negaatio on toteutettu Vector3f-luokalle määritetyllä negate-metodilla, joka asettaa luokan x-, y- ja z-muuttujat käänteisiksi arvoiksi, muuttaen näin muuttujien sisältämät arvot vastakkaismerkkisiksi.

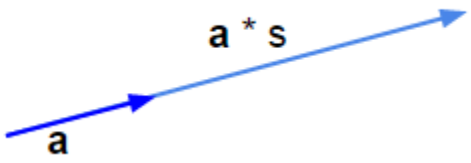
```
WEBGL_LIB.Math.Entities.Vector3f.prototype.negate = function(){
    this.x = -this.x;
    this.y = -this.y;
```

```
    this.z = -this.z;
}
```

Vektorin kertominen skalaarilla tapahtuu kertomalla jokainen vektorin komponentti erikseen skalaariarvolla (Dunn & Parberry 2011, 45). Kaavassa 3 esitetään vektorin kertominen skalaariarvolla.

$$4 * \begin{bmatrix} 2 \\ 1.5 \\ -3 \end{bmatrix} = \begin{bmatrix} 8 \\ 6 \\ -12 \end{bmatrix} \quad (3)$$

Positiivisella skalaariarvolla kertominen vaikuttaa vektorin suuruuteen muuntamalla vektorin suuruutta skalaarin määrittämän arvon mukaisesti, mutta se ei muuta vektorin suuntaa (Dunn & Parberry 2011, 45–46). Jos kertova skalaariarvo on 2, tuplaa se vektorin määrittämän pituuden. Kuvassa 1 havainnollistetaan, kuinka positiivisella skalaarilla kertominen kasvattaa vektorin pituutta samassa suunnassa, johon se osoittaa.



Kuva 1. Vektorin kertominen positiivisella skalaarilla kasvattaa vektorin pituutta, muttei muuta sen suuntaa.

Vector3f-luokassa vektorin kertominen skalaarilla on toteutettu luokalle määritetyllä mulScal-metodilla. Metodi palauttaa uuden Vector3f-olion, jonka arvot muodostetaan kertomalla vektorin jokainen komponentti parametrina annetulla skalaariarvolla, ja sijoittamalla lopputulos palautettavan vektorin vastaaviin komponentteihin.

```
WEBGL_LIB.Math.Entities.Vector3f.prototype.mulScal = function(scalar)
    var result = new WEBGL_LIB.Math.Entities.Vector3f(0,0,0);
    result.x = this.x*scalar;
```

```
    result.y = this.y*scalar;
    result.z = this.z*scalar;
    return result;
};
```

Vektorin jakaminen skalaarilla tuottaa vektorin, jonka pituus on muuttunut jakavan arvon vaikutuksesta. Jos vektoria jaettaisiin arvolla 2, puolittuisi vektorin pituus, mutta sen suunta säilyisi samana. Vektorin jakaminen skalaarilla on hyvin oleellista, kun vektori halutaan muuttaa yksikkövektoriksi. (Ks. luku 2.1.2.)

Kirjastossa vektorin jakaminen skalaariarvolla on toteutettu Vector3f-luokan divScal-metodilla, joka ottaa parametrina jakamiseen käytettävän skalaariarvon. Skalaariarvolla jaetaan jokainen metodia kutsuneen vektorin komponentti, ja tulokset sijoitetaan uuteen vektoriin, joka palautetaan paluuarvona. Alla esitetään vektorin jakolaskun toteuttava osuus metodin koodista.

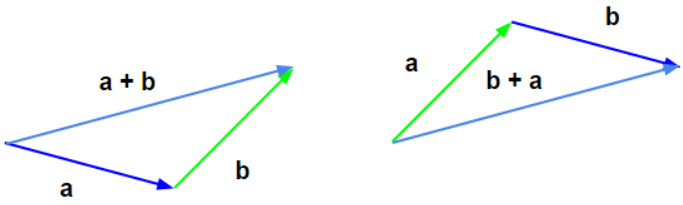
```
WEBGL_LIB.Math.Entities.Vector3f.prototype.divScal = function(scalar){
    var result = new WEBGL_LIB.Math.Entities.Vector3f(0,0,0);
    result.x = this.x/scalar;
    result.y = this.y/scalar;
    result.z = this.z/scalar;
    return result;
};
```

Vektoreita ei voi yhteenlaskea tai vähentää skalaarin tai toisen eri avaruudessa olevan vektorin kanssa. Kaksi samassa avaruudessa olevaa vektoria voidaan kuitenkin yhteenlaskea ja vähentää toisistaan. (Dunn & Parberry 2011, 47.) Kaavassa 4 esitetään kahden vektorin välinen yhteenlasku ja kaavassa 5 kahden vektorin välinen vähennyslasku.

$$\begin{bmatrix}1\\2\\3\end{bmatrix}+\begin{bmatrix}4.5\\-2\\3\end{bmatrix}=\begin{bmatrix}1+4.5\\2+(-2)\\3+3\end{bmatrix}=\begin{bmatrix}5.5\\0\\6\end{bmatrix} \quad (4)$$

$$\begin{bmatrix}1\\2\\3\end{bmatrix} - \begin{bmatrix}4.5\\-2\\3\end{bmatrix} = \begin{bmatrix}1-4.5\\2-(-2)\\3-3\end{bmatrix} = \begin{bmatrix}3.5\\4\\0\end{bmatrix} \quad (5)$$

Kuten skalaareilla suoritettavissa laskutoimituksissa, vektoreiden yhteenlaskussa laskujärjestyksellä ei ole väliä, mutta vähennyslaskussa järjestys vaikuttaa lopputulokseen. Geometrisesti kahden vektorin välisestä yhteenlaskusta saatava tulos tuottaisi uuden vektorin, joka vastaisi kuvassa 2 muodostettua vektoria. Kun molemmat vektorit asetellaan peräjälkeen toisistaan, osoittaa tuloksena saatava vektori samaan pisteeseen, kuin mihin peräjälkeiset vektorit päättyvät. Sama periaate pätee silloinkin, kun yhteenlaskettavia vektoreita olisi useampia peräkkäin. (Dunn & Parberry 2011, 48–49.)

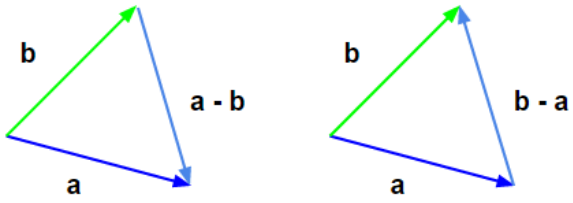


Kuva 2. Vektorien välinen yhteenlasku tuottaa uuden vektorin, ja tuloksena saatava vektori on sama laskujärjestyksestä huolimatta.

Vektoreiden välinen yhteenlasku toteutetaan kirjastossa Vector3f-luokan addVect-metodilla, joka vastaanottaa parametrina yhteenlaskettavan vektorin. Metodi palauttaa uuden vektorin, jonka arvo muodostuu metodia kutsuneen vektorin ja parametrina annetun vektorin välille suoritetusta yhteenlaskusta. Alla esitetään vektoreiden yhteenlaskun suorittavan metodin koodi.

```
WEBGL_LIB.Math.Entities.Vector3f.prototype.addVect = function(vector3f){
    var result = new WEBGL_LIB.Math.Entities.Vector3f(0,0,0);
    result.x = this.x+vector3f.x;
    result.y = this.y+vector3f.y;
    result.z = this.z+vector3f.z;
    return result;
};
```

Vektoreiden vähennyslasku tuottaa vektorin, joka on sama kuin siirtymä vähentävän vektorin päästä vähennettävän vektorin päähän. Tätä havainnollistetaan kuvassa (kuva 3).



Kuva 3. Vektorien välinen vähennyslasku tuottaa uuden vektorin, joka määrittää siirtymän vähennettävän vektorin määrittämään sijaintiin vähentävän vektorin sijainnista.

Kirjastossa vektoreiden välinen vähennyslasku toteutetaan Vector3f-luokan subVect-metodilla, joka ottaa parametrina vähentävän vektorin, jonka sisältämät arvot vähennetään sitä kutsuneen vektorin arvoista. Tuloksena palautetaan uusi vektori, joka vastaa vektoreiden välistä vähennyslaskua. Alla esitetään vektoreiden vähennyslaskun suorittavan metodin koodi.

```
WEBGL_LIB.Math.Entities.Vector3f.prototype.subVect = function(vector3f){
var result = new WEBGL_LIB.Math.Entities.Vector3f(0,0,0);
    result.x = this.x-vector3f.x;
    result.y = this.y-vector3f.y;
    result.z = this.z-vector3f.z;
    return result;
};
```

2.1.2 Vektorin suuruus, yksikkövektorit ja etäisyys

Vektorin suuruus tai pituus voidaan määritellä laskemalla neliöjuuri vektorin komponenttien neliöiden summasta. Kaavassa 6 nähdään vektorin pituuden laskeminen, jossa pituuden laskua merkitään kahdella viivalla vektoria esittävän muuttujan ympärillä.

$$\|\boldsymbol{v}\| = \sqrt{v_x^2 + v_y^2 + v_z^2}, \tag{6}$$

$$\|[5, -4,7]\| = \sqrt{5^2 + (-4)^2 + 7^2} = \sqrt{25 + 16 + 49} = \sqrt{90}$$

Huomattavaa pituuden laskemisessa on, että tulos on aina positiivinen. Vektorin pituus ei voi olla negatiivinen, olipa vektorin suunta mikä tahansa. (Dunn & Parberry 2011, 51–52.)

Vektorin pituus lasketaan kirjastossa Vector3f-luokan length-metodilla. Metodi laskee sitä kutsuneen kutsuneen vektorin komponenttien neliöt kertomalla ne itsellään ja laskemalla komponenttien neliöistä muodostuneet arvot yhteen, mistä saatavasta summasta lasketaan neliöjuuri. Metodi palauttaa paluuarvona liukuluvun, joka esittää vektorin pituuden.

```
WEBGL_LIB.Math.Entities.Vector3f.prototype.length = function(){
    var result = Math.sqrt(this.x * this.x + this.y * this.y +
                                             this.z * this.z);
    return result;
};
```

Yksikkövektorit, joita toisinaan kutsutaan myös normalisoiduiksi vektoreiksi tai normaaleiksi, ovat yhden yksikön pituisia vektoreita. Monissa tilanteissa on vain tärkeää tietää vektorin suunta, eikä vektorin pituudella ole väliä. Tällaisia tapauksia ovat esimerkiksi 3D-mallin pintojen normaalivektorit. Mikä tahansa vektori (nollavektoria lukuun ottamatta) voidaan muuttaa yksikkövektoriksi kaavan 7 määrittelemällä tavalla. Yksikkövektoria esitetään kaavoissa käyttämällä hattumerkintää vektorimuuttujaa esittävän kirjaimen päällä. (Dunn & Parberry 2011, 53–54.)

$$\hat{\boldsymbol{v}} = \frac{\boldsymbol{v}}{\|\boldsymbol{v}\|}, \tag{7}$$

$$\frac{[2,2,2]}{\sqrt{2^2 + 2^2 + 2^2}} = \frac{[2,2,2]}{\sqrt{4 + 4 + 4}} = \frac{[2,2,2]}{\sqrt{12}}$$

$$= \left[\frac{2}{\sqrt{12}}, \quad \frac{2}{\sqrt{12}}, \quad \frac{2}{\sqrt{12}}\right] \approx [0.57735 \quad 0.57735 \quad 0.57735]$$

Tätä prosessia kutsutaan usein vektorin normalisoimiseksi. Vektori jaetaan vektorin pituudella, jolloin tuloksena saadaan vektori, jonka pituus on 1. Kaavassa 8 lasketaan yllä esitetyn normalisoidun vektorin pituus, ja laskusta saatava tulos on 1.

$$\left\|\left[\frac{2}{\sqrt{12,}} \quad \frac{2}{\sqrt{12,}} \quad \frac{2}{\sqrt{12}}\right]\right\| = \sqrt{\frac{2}{\sqrt{12}}^{2} + \frac{2}{\sqrt{12}}^{2} + \frac{2}{\sqrt{12}}^{2}} = 1 \tag{8}$$

Huolimatta siitä, onko normalisoitavan vektorin alkuperäinen pituus enemmän kuin 1 tai vähemmän kuin 1, normalisoitava vektori tulee aina saamaan pituuden 1. Ainoastaan vektori, jonka kaikki komponentit ovat nolla-arvoisia, ei normalisoidu, koska se ei määrittele minkäänlaista siirtymää. (Dunn & Parberry 2011, 54–55.)

Kirjastossa vektorin normalisoiva toiminto on toteutettu Vector3f-luokan normalize-metodilla, joka hyödyntää vektorin pituuden palauttavaa length-metodia, ja vektorin skalaarilla jakavaa divScal-metodia normalisoinnissa. Metodia kutsunut vektori jaetaan omalla pituudellaan, ja tuloksena saatu vektori palautetaan funktion paluuarvona. Alla koodi, joka toteuttaa vektorin normalisoinnin metodissa.

```
WEBGL_LIB.Math.Entities.Vector3f.prototype.normalize = function(){
    return this.divScal(this.length());
};
```

Kahden vektorin välinen etäisyys saadaan helposti laskettua hyödyntämällä vektorin pituuden laskemista ja vektoreiden välistä erotusta. Kahden vektorin vähennyslaskulla saadaan kahden vektorin välinen vektori, ja laskemalla vektorin pituus voidaan selvittää kahden vektorin välinen etäisyys. Vektoreiden välisen etäisyyden laskeva distance-funktio esitellään kaavassa 9. (Dunn & Parberry 2011, 55–56.)

$$distance(\boldsymbol{a},\boldsymbol{b}) = \|\boldsymbol{b}-\boldsymbol{a}\| = \sqrt{(b_x-a_x)^2+\left(b_y-a_y\right)^2+(b_z-a_z)^2}, \quad (9)$$

$$distance([3,3,3],[6,6,6]) = \|[6,6,6]-[3,3,3]\|$$

$$= \sqrt{(6-3)^2+(6-3)^2+(6-3)^2} = \sqrt{9+9+9} = \sqrt{27} = 5.19615$$

Kirjastossa vektoreiden välinen etäisyys lasketaan Vector3f-luokan distance-metodilla, joka laskee vektoreiden välisen erotuksen subVect-metodilla. Tämän jälkeen erotuksesta saatavan vektorin pituus lasketaan, ja laskettu pituus palautetaan metodin paluuarvona. Alla olevassa koodissa esitetään vektoreiden välisen etäisyyden laskeva metodi.

```
WEBGL_LIB.Math.Entities.Vector3f.prototype.distance =
                                                function(vector3f){
      var new_vector = this.subVect(vector3f);
      return new_vector.length();
};
```

2.1.3 Vektorin piste- ja ristitulo

Kahden vektorin välinen tulo voidaan laskea kahdella eri tavalla: pistetulolla ja ristitulolla. Pistetulo on hyvin usein käytetty vektorioperaatio, ja se on siksi syytä osata. Nimi pistetulo tulee laskun matemaattisesta merkintätavasta, jossa vektorien välinen pistetulo merkitään pistekertomerkillä. (Dunn & Parberry 2011, 56.) Kaavassa 10 nähdään kahden vektorin välisen pistetulon merkintätapa.

$$\boldsymbol{a}\cdot\boldsymbol{b} \quad (10)$$

Vektoreiden pistetulon laskeminen on helppo laskutoimitus. Lasku suoritetaan kaavassa 11 esitetyllä tavalla, eli kertomalla molempien vektoreiden vastaavat komponentit keskenään, ja laskemalla komponenttien kertolaskujen tulokset yhteen. Tuloksena saatu summa on vektorin pistetulon arvo.

$$\boldsymbol{a} \cdot \boldsymbol{b} = \begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix} \cdot \begin{bmatrix} b_x \\ b_y \\ b_z \end{bmatrix} = a_x b_x + a_y b_y + a_z b_{z,} \tag{11}$$

$$\begin{bmatrix} 5 \\ 6 \\ -5 \end{bmatrix} \cdot \begin{bmatrix} 8 \\ -5.5 \\ -7 \end{bmatrix} = (5)(8) + (6)(-5.5) + (-5)(-7) = 40 + (-33) + 35 = 42$$

Pistetulon kanssa laskujärjestyksellä ei ole väliä, koska samat komponentit kerrotaan yhteen joka tapauksessa (Dunn & Parberry 2011, 57). Tämän vuoksi kaavassa 12 esitetty väite on tosi.

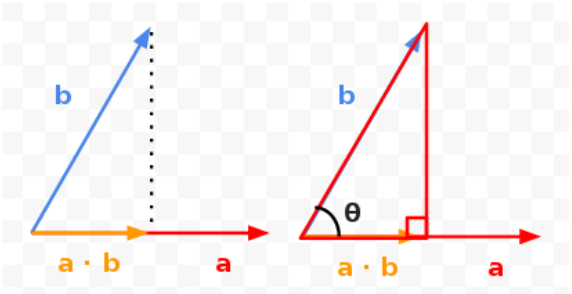
$$\boldsymbol{a} \cdot \boldsymbol{b} = \boldsymbol{b} \cdot \boldsymbol{a} \tag{12}$$

Jotta pistetulon merkityksen voisi ymmärtää paremmin, käsitellään nopeasti mitä pistetulo tekee geometrisesti. Pistetulon voidaan ajatella määrittelevän kahden vektorin välisen kulman. Kahden vektorin pistetulo on sama, kuin niiden välisen kulman kosini kerrottuna vektoreiden pituuksilla. Kaavan 13 mukaisesti, vektoreiden välinen kulma voidaan laskea kahden vektorin pistetulon tulokseen avulla. Jakamalla pistetulon tulos vektoreiden pituuksien tulolla ja ottamalla tuloksesta arkuskosini, saadaan vektoreiden välinen kulma selvitettyä. Kuvassa 4 esitetään kahden vektorin pistetulon muodostamaa suhdetta. (Dunn & Parberry 2011, 64–65.)

$$\boldsymbol{a} \cdot \boldsymbol{b} = \|\boldsymbol{a}\|\|\boldsymbol{b}\| cos\theta, \tag{13}$$

$$cos\theta = \frac{\boldsymbol{a} \cdot \boldsymbol{b}}{\|\boldsymbol{a}\|\|\boldsymbol{b}\|},$$

$$\theta = arccos\left(\frac{\boldsymbol{a} \cdot \boldsymbol{b}}{\|\boldsymbol{a}\|\|\boldsymbol{b}\|}\right),$$

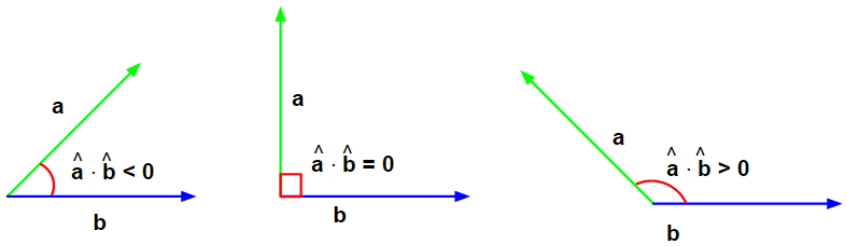


Kuva 4. Pistetulon tuloksella on vahva yhteys kahden vektorin väliseen kulmaan.

Jos molemmat pistetulon vektoreista ovat yksikköpituudessa, on pistetulon tulos silloin suoraan vektoreiden välisen kulman kosini. Käyttämällä pistetulossa yksikkövektoreita, ei kulman suuruuden saamiseksi tarvitse suorittaa pistetulon tuloksen jakamista vektoreiden pituudella, vaan kulma voidaan ottaa suoraan laskemalla arkuskosini pistetulon tuloksesta. Yksikkövektoreiden pistetulo on esitetty kaavassa 14. (Dunn & Parberry 2011, 65.)

$$\hat{\boldsymbol{a}} \cdot \hat{\boldsymbol{b}} = cos\theta, \tag{14}$$
$$\theta = arccos(\hat{\boldsymbol{a}} \cdot \hat{\boldsymbol{b}})$$

Vektorien välisen kulman suuruus kertoo paljon siitä, millainen suhde vektoreilla on toisiinsa nähden. Kulman ollessa 0 <= θ > 90, osoittavat vektorit enemmän samaan suuntaan, ja pistetulon arvo on suurempi kuin 0. Kulman ollessa 90 astetta, ovat vektorit kohtisuorassa keskenään, ja silloin pistetulon suuruus on 0. Kulman ollessa 90 < θ <= 180 astetta, osoittavat vektorit eniten toisistaan poispäin, joten pistetulon suuruus on silloin pienempi kuin 0. (Dunn & Parberry 2011, 66.) Kuvassa 5 esitetään yksikkövektoreiden välisten kulmien ja pistetulon tuloksen suhdetta. Yksikkövektorin käytöllä voimme päätellä helpommin vektoreiden välisen kulman, joten niitä pyritään käyttämään aina, kun vektoreiden välisen kulman selvittäminen on oleellista.



Kuva 5. Yksikkövektoreiden välisen pistetulon tuloksesta voidaan päätellä vektoreiden välinen kulma.

Vektoreiden pistetulolla on monia käyttötapoja, kuten pinnalle kohdistuvan valaistuksen laskeminen suorittamalla pistetulo pinnan normaalivektorin ja normalisoidun valon kulkusuuntaa esittävän vektorin kanssa. Valaistuksesta ja sen toteutuksesta keskustellaan lisää myöhemmissä kappaleissa.

Kirjastossa vektorin pistetulo suoritetaan Vector3f-luokan dotProduct-metodilla, joka ottaa parametrina vektorin, jonka kanssa pistetulo suoritetaan metodia kutsuneen vektorin kanssa. Vektoreiden komponenttien arvot kerrotaan keskenään, ja tulojen tulokset lasketaan yhteen. Tuloksena saatu skalaariarvo palautetaan metodin paluuarvona. Alla olevassa koodissa nähdään pistetulon suorittavan metodin suorittama toiminnallisuus.

```
WEBGL_LIB.Math.Entities.Vector3f.prototype.dotProduct =
                                                    function(vector3f){
    var result = this.x*vector3f.x + this.y*vector3f.y +
                                        this.z*vector3f.z;
    return result;
};
```

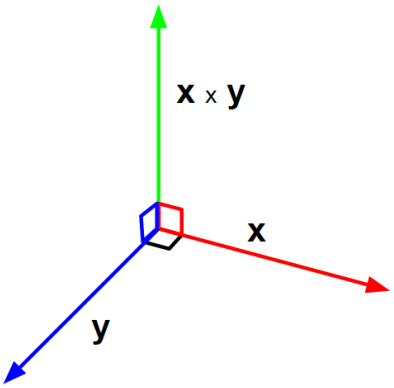
Vektorin ristitulo toimii vain kolmiulotteisten vektoreiden kanssa, vaikkakin kaksiulotteisilla vektoreilla on myös omanlaisensa kaksiulotteinen ristitulo, johon tässä opinnäytetyössä ei oteta kantaa. Kahden kolmiulotteisen vektorin ristitulo tuottaa uuden kolmiulotteisen vektorin, joka on kohtisuora molempiin ristitulossa käytettyihin vektoreihin nähden. (Puhakka 2008, 36–37.) Kaavassa 15 nähdään kahden vektorin ristitulon suoritus.

$$\boldsymbol{a} \times \boldsymbol{b} = \begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix} \times \begin{bmatrix} b_x \\ b_y \\ b_z \end{bmatrix} = \begin{bmatrix} a_y b_z - a_z b_y \\ a_z b_x - a_x b_z \\ a_x b_y - a_y b_x \end{bmatrix} \quad (15)$$

Kaavan 15 mukaisesti, ristitulossa tuloksena saatavassa vektorissa kerrotaan ristiin muut vektoreiden komponentit, paitsi se jota sillä hetkellä lasketaan. Esimerkiksi tuloksena saatavan vektorin x-komponenttia laskettaessa kerrotaan vain tulon suorittavien vektoreiden y- ja z-komponentteja.

Hyvä esimerkki ristitulon muodostamasta kohtisuorasta vektorista voisi olla ristitulon suorittaminen vektoreille x = [1, 0, 0] ja y = [0, 1, 0], jolloin tuloksena saatava vektori olisi [0, 0, 1], joka on kohtisuora molempiin vektoreihin nähden. Geometrisesti katsottuna, ristitulo tuottaa uuden vektorin, joka on kohtisuora

ristituloa varten käytettyihin vektoreihin nähden. Hahmotelma ristitulon muodostamasta uudesta vektorista esitetään kuvassa 6, jossa vektoreiden kohtiosuoraisuutta tuloksena saatavaan vektoriin nähden esitetään kulmamerkkien avulla.



Kuva 6. Ristitulo tuottaa tuloksena uuden vektorin, joka on kohtisuora ristituloon käytettyihin vektoreihin nähden.

Ristitulon suorittamisessa on huomioitava, että vektoreiden kertomisjärjestyksellä on väliä ristitulossa, koska ristitulon suorittaminen eri järjestyksessä tuottaa vektorin, joka osoittaa vastakkaiseen suuntaan (Dunn & Parberry 2011, 67).

Kirjaston Vector3f-luokalle määritelty ristitulon suorittava crossProduct-metodi on esitetty alla olevalla koodilla, joka toteuttaa sitä kutsuneen ja parametrina annetun Vector3f-muutujien välisen ristitulon kaavan 15 mukaisesti.

```
WEBGL_LIB.Math.Entities.Vector3f.prototype.crossProduct =
                                                    function(vector3f){
    var new_x = this.y * vector3f.z - this.z * vector3f.y;
    var new_y = this.z * vector3f.x - this.x * vector3f.z;
    var new_z = this.x * vector3f.y - this.y * vector3f.x;
    return new WEBGL_LIB.Math.Entities.Vector3f(new_x, new_y, new_z);

};
```

2.2 Matriisit ja niiden operaatiot

Matriisit ovat kaksiulotteinen taulukko, jossa on lukuja sijoitettuna riveille ja sarakkeille. Kuten vektori on yksiulotteinen taulukko jossa on lukuja, on matriisi kaksiulotteinen taulukko jossa on lukuja. (Dunn & Parberry 2011, 113.)

Matriisin koko määrittyy siitä, kuinka monta riviä ja saraketta matriisi sisältää. Esimerkiksi kaavassa 16 esitetty matriisi on 4 x 3 matriisi. Matriisin kokoa määritettäessä ensin lasketaan rivien määrä, minkä jälkeen tulee sarakkeiden määrä (Dunn & Parberry 2011, 115). Matriisia esittävä muuttuja merkitään kaavoissa lihavoidulla suuraakkosella kaikissa kaavoissa, joissa matriiseja esitetään. Matriisin sisältämiä arvoja esitetään hakasuluissa olevalla kaksiulotteisella taulukolla.

$$\boldsymbol{M} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{16}$$

Yleisimpiä käytössä olevia matriiseja 3D-grafiikassa ovat neliömatriisit, eli matriisit joissa on yhtä paljon rivejä ja sarakkeita, kuten 3 x 3 tai 4 x 4 matriisit. Yksi tapa merkitä matriisin elementtien sijainteja on merkitä rivin ja sarakkeen sijainti alaviitteenä. Tämä esitystapa nähdään kaavassa 17, jossa ensimmäinen luku kertoo elementin rivin matriisi taulukossa, ja toinen luku kertoo elementin sarakkeen taulukossa. (Dunn & Parberry 2011, 115.)

$$\boldsymbol{M} = \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix} \tag{17}$$

Kirjastossa on määritelty kaksi eri luokkaa matriiseille, jotka ovat Matrix3f ja Matrix4f. Matrix3f-luokka (WEBGL_LIB.Math.Entities.Matrix3f) määrittelee 3 x 3 matriisin, joka muodostetaan luokan Float32Array-jäsenmuuttujataulukkoon, joka on 9 alkiota sisältävä taulukko. Matrix3f-luokka voidaan alustaa antamalla

parametreina taulukko, joka sisältää luokan taulukkoon sijoitettavat arvot. Jos arvoja ei anneta erikseen, muodostetaan matriisi siten, että se on identiteettimatriisi (Ks. luku 2.2.1). Alla olevassa koodissa nähdään Matrix3f-luokalle määritelty rakentajafunktio.

```
WEBGL_LIB.Math.Entities.Matrix3f = function(matrixArray){
    if(!matrixArray || matrixArray.length !== 9){
      this.array = new Float32Array(9);
      this.array[0] = 1.0; //m11
      this.array[1] = 0.0; //m21
      this.array[2] = 0.0; //m31
      this.array[3] = 0.0; //m12
      this.array[4] = 1.0; //m22
      this.array[5] = 0.0; //m32
      this.array[6] = 0.0; //m13
      this.array[7] = 0.0; //m23
      this.array[8] = 1.0; //m33
    }else{
      this.array = new Float32Array(matrixArray);
    }
};
```

Matriisia ilmaiseva taulukko muodostetaan siten, että matriisi muodostetaan sarakkeiden suuntaisesti. Ensimmäiset kolme alkiota määrittelevät matriisin ensimmäisen sarakkeen elementit, seuraavat kolme toisen sarakkeen elementit ja niin edelleen. Samat toiminnallisuudet pätevät myös 4 x 4 matriisia esittävään Matrix4f-luokkaan (WEBGL_LIB.Math.Entities.Matrix4f), jonka sisältämä jäsenmuuttujataulukko määrittelee 16 alkiota. Alla olevassa koodissa nähdään Matrix4f-luokan rakentajafunktio.

```
WEBGL_LIB.Math.Entities.Matrix4f = function(matrixArray){
    if(!matrixArray || matrixArray.length !== 16){
      this.array = new Float32Array(16);
      this.array[0] = 1.0; //m11
```

```
        this.array[1] = 0.0; //m21
        this.array[2] = 0.0; //m31
        this.array[3] = 0.0; //m41
        this.array[4] = 0.0; //m12
        this.array[5] = 1.0; //m22
        this.array[6] = 0.0; //m32
        this.array[7] = 0.0; //m42
        this.array[8] = 0.0; //m13
        this.array[9] = 0.0; //m23
        this.array[10] = 1.0; //m33
        this.array[11] = 0.0; //m43
        this.array[12] = 0.0; //m14
        this.array[13] = 0.0; //m24
        this.array[14] = 0.0; //m34
        this.array[15] = 1.0; //m44
    }else{
        this.array = new Float32Array(matrixArray);
    }
};
```

2.2.1 Matriisin diagonaali, identiteettimatriisi ja traspoosi

Matriisin diagonaalilla tarkoitetaan neliömatriisin elementtejä, jotka kulkevat matriisin taulukon vasemmasta yläkulmasta oikeaan alareunaan. 3 x 3 matriisin muodostaman diagonaalin arvot on merkitty kaavassa 18 punaisella.

$$\boldsymbol{M} = \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix} \tag{18}$$

Diagonaali on tärkeä esimerkiksi yksikkö- tai identiteettimatriisi käsitteissä. Identiteettimatriisi on matriisi, jossa matriisin diagonaalilla sijaitsevien elementtien arvot ovat 1, ja kaikkien muiden matriisin elementtien arvot ovat 0. (Puhakka 2008, 84.) Esimerkki 3 x 3 identiteettimatriisista nähdään kaavassa 19.

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (19)$$

Identiteettimatriisin erikoisuus on siinä, että sillä kerrottaessa mitä tahansa muuta matriisia, ei kerrottavan matriisin sisältämät arvot muutu. Identiteettimatriisi on matriiseille sama asia, kuin mitä luku 1 on skalaariarvoille, koska luvulla 1 kertominen ei muuta kerrottavaa lukua. (Dunn & Parberry 2011, 116.)

Matriisin transpoosilla tarkoitetaan, että matriisin rivit muuttuvat sarakkeiksi, ja sarakkeet muuttuvat riveiksi. Matriisin transpoosi merkitään kaavoissa, kuten kaavassa 20, yläindeksissä olevalla T-kirjaimella matriisimuuttujaa esittävän lihavoidun suuraakkosen viereen, tai vaihtoehtoisesti matriisia esittävän taulukon viereen. (Puhakka 2008, 84.)

$$\boldsymbol{M} = \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix}, \boldsymbol{M}^T = \begin{bmatrix} m_{11} & m_{21} & m_{31} \\ m_{12} & m_{22} & m_{32} \\ m_{13} & m_{23} & m_{33} \end{bmatrix} \quad (20)$$

Kirjastossa transpoosi toteutetaan Matrix3f-luokan olioille transposeMatrix-metodin avulla. Metodi palauttaa uuden matriisin, joka alustetaan sitä kutsuneen matriisin taulukon arvoilla. Arvot on muodostettu uudessa matriisissa siten, että metodia kutsuneen matriisin rivien arvot sijoittuvat sarakkeiden suuntaisesti ja sarakkeiden suuntaiset arvot sijoittuvat rivien suuntaisesti. Alla olevassa koodissa nähdään Matrix3f-luokan määrittelemä transpose-metodi.

```
WEBGL_LIB.Math.Entities.Matrix3f.prototype.transposeMatrix = function(){
    var a = this.array;
    return new WEBGL_LIB.Math.Entities.Matrix3f(
                        [a[0],a[3],a[6],a[1],a[4],a[7],a[2],a[5],a[8]]);
};
```

2.2.2 Matriisin kertominen

Matriiseille suoritettavista tuloista yksinkertaisin on kertoa matriisi skalaariarvolla. Tämä tapahtuu kertomalla matriisin jokainen arvo skalaarilla kaavan 21 mukaisesti.

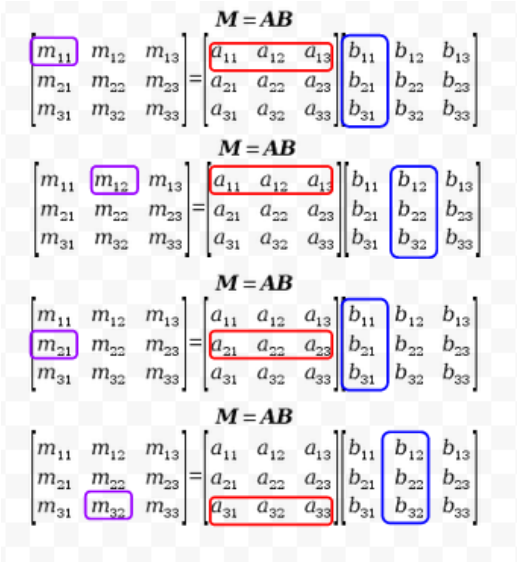
$$k\boldsymbol{M} = \begin{bmatrix} m_{11} & m_{21} & m_{31} \\ m_{12} & m_{22} & m_{32} \\ m_{13} & m_{23} & m_{33} \end{bmatrix} \tag{21}$$
$$= k \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix} = \begin{bmatrix} km_{11} & km_{12} & km_{13} \\ km_{21} & km_{22} & km_{23} \\ km_{31} & km_{32} & km_{33} \end{bmatrix}$$

Kahden matriisin kertomisesta keskenään saadaan tuloksena uusi matriisi. Kahta matriisia kerrottaessa kertolaskun ensimmäisen matriisin sarakkeiden määrän täytyy olla sama, kuin toisen matriisin rivien määrä. Huomattavaa on myös se, että tuloksena saatavalla matriisilla on yhtä paljon rivejä, mitä tulon ensimmäisellä matriisilla, sekä yhtä paljon sarakkeita, kuin tulon toisella matriisilla. (Dunn & Parberry 2011, 118.) Matriisien kertomisessa tapahtuvaa matriisin koon muuttumista havainnollistetaan kaavassa 22.

$$[4 \times 2] \cdot [2 \times 5] \rightarrow [4 \times 5], \tag{22}$$
$$\begin{bmatrix} ? & ? \\ ? & ? \\ ? & ? \\ ? & ? \end{bmatrix} \cdot \begin{bmatrix} ? & ? & ? & ? & ? \\ ? & ? & ? & ? & ? \end{bmatrix} \rightarrow \begin{bmatrix} ? & ? & ? & ? & ? \\ ? & ? & ? & ? & ? \\ ? & ? & ? & ? & ? \\ ? & ? & ? & ? & ? \end{bmatrix},$$

Neliömatriisien kohdalla, koon muutosta ei tapahdu, sillä matriisien on oltava samaa kokoa, jotta tulo onnistuisi. Tulon suorittaminen on melko työläs, mutta suoraviivainen prosessi. Yksi tapa ajatella tuloa on miettiä tuloksena saatavan matriisin elementtien alaindeksien lukuja. Kaavassa 23 esitetyssä tilanteessa vasemmalla on tuloksena saatava 3 x 3 matriisi **M**, ja oikealla on kaksi tulossa käytettyä 3 x 3 matriisia **A** ja **B**. Kuvassa 7 hahmotetaan, kuinka matriisin **M** jokaisen elementin alaindeksi kertoo suoraan, mikä rivi matriisista **A** pitää kertoa matriisin **B** sarakkeen kanssa. Näitä rivejä ja sarakkeita voi pitää vektoreina,

jotka esimerkiksi laskettaessa arvoa matriisin **M** elementille m_{11} olisivat [a_{11}, a_{12}, a_{13}] sekä [b_{11}, b_{21}, b_{31}]. Tämän jälkeen suoritetaan vain pistetulo näille kahdelle poimitulle osiolle samaan tapaan, kuin kahden vektorin pistetulon kanssa. (Dunn & Parberry 2011, 118.)



Kuva 7. Matriisin yksittäisten alkioiden arvojen laskeminen tapahtuu ristitulon suorittamisella kerrottavan matriisin rivien ja kertovan matriisin sarakkeiden välillä.

$$\boldsymbol{M = AB} \qquad \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} \tag{23}$$

Kaavassa 24 esitetään, kuinka matriisi taulukon ensimmäisen alkion m_{11} arvo lasketaan suorittamalla pistetulo kerrottavan matriisin **A** ensimmäisen rivin suuntaisten alkioiden ja kertovan matriisin **B** ensimmäisen sarakkeen suuntaisten alkioiden arvojen välillä. Jos laskettava alkio olisi m_{21}, suoritettaisiin tulo matriisin **A** toisen rivin suuntaisten alkioiden ja kertovan matriisin **B** ensimmäisen sarakkeen suuntaisten alkioiden arvojen välillä, kuten kaavassa 24 esitetty.

$$\begin{aligned} m_{11} &= a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31}, \\ m_{21} &= a_{21}b_{11} + a_{22}b_{21} + a_{23}b_{31} \end{aligned} \tag{24}$$

Sama laskutoimitus täytyy toteuttaa tuloksena saatavan matriisin jokaiselle alkiolle, minkä vuoksi laskua voi pitää melko työläänä. Kaavassa 25 esitettään, kuinka kahden 3 x 3 matriisin suorittaman matriisitulon laskeminen lasketaan kokonaisuudessaan. 4 x 4 matriisilla tämä kaava on vielä suurempi.

$$\boldsymbol{AB} = \boldsymbol{M} \tag{25}$$

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} =$$

$$\begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} & a_{11}b_{12} + a_{12}b_{22} + a_{13}b_{32} & a_{11}b_{13} + a_{12}b_{23} + a_{13}b_{33} \\ a_{21}b_{11} + a_{22}b_{21} + a_{23}b_{31} & a_{21}b_{12} + a_{22}b_{22} + a_{23}b_{32} & a_{21}b_{13} + a_{22}b_{23} + a_{23}b_{33} \\ a_{31}b_{11} + a_{32}b_{21} + a_{33}b_{31} & a_{31}b_{12} + a_{32}b_{22} + a_{33}b_{32} & a_{31}b_{13} + a_{32}b_{23} + a_{33}b_{33} \end{bmatrix}$$

Kirjastossa kahden samaa kokoa olevan matriisin välinen tulo voidaan suorittaa käyttämällä molemmille matriisiluokille määritettyä matrixMult-metodia. Metodi ottaa vastaan parametrina samaa tyyppiä olevan matriisin, joka toimii kertovana matriisina metodia kutsuneelle matriisille. Tuloksena palautettavan matriisin jokaiselle alkiolle määritetään arvo suorittamalla matriisitulo metodia kutsuneen matriisin taulukon rivien suuntaisten arvojen ja parametrina saadun matriisi sarakkeiden suuntaisten arvojen välillä. Alla olevassa koodissa esitetään Matrix3f-luokan multMatrix-metodin suorittama tulo matriisien välillä.

```
WEBGL_LIB.Math.Entities.Matrix3f.prototype.matrixMult = function(mat3){
    var result = new WEBGL_LIB.Math.Entities.Matrix3f();
    result.array[0] = this.array[0] * mat3.array[0] +
                      this.array[3] * mat3.array[1] +
                      this.array[6] * mat3.array[2]; //m11
    result.array[1] = this.array[1] * mat3.array[0] +
                      this.array[4] * mat3.array[1] +
                      this.array[7] * mat3.array[2]; //m21
    result.array[2] = this.array[2] * mat3.array[0] +
                      this.array[5] * mat3.array[1] +
                      this.array[8] * mat3.array[2]; //m31
    result.array[3] = this.array[0] * mat3.array[3] +
                      this.array[3] * mat3.array[4] +
                      this.array[6] * mat3.array[5]; //m12
```

```
    result.array[4] = this.array[1] * mat3.array[3] +
                      this.array[4] * mat3.array[4] +
                      this.array[7] * mat3.array[5]; //m22
    result.array[5] = this.array[2] * mat3.array[3] +
                      this.array[5] * mat3.array[4] +
                      this.array[8] * mat3.array[5]; //m32
    result.array[6] = this.array[0] * mat3.array[6] +
                      this.array[3] * mat3.array[7] +
                      this.array[6] * mat3.array[8]; //m13
    result.array[7] = this.array[1] * mat3.array[6] +
                      this.array[4] * mat3.array[7] +
                      this.array[7] * mat3.array[8]; //m23
    result.array[8] = this.array[2] * mat3.array[6] +
                      this.array[5] * mat3.array[7] +
                      this.array[8] * mat3.array[8]; //m33
    return result;
}
```

Vektoreita voi pitää yksiulotteisina matriiseina, joilla on yksi rivi tai yksi sarake. Tämän vuoksi vektori voidaan kertoa samaan tapaan matriisin kanssa, kuin kaksi matriisia keskenään. Huomioon on kuitenkin otettava se, että vektori voidaan esittää sekä rivi että sarake muodossa. (Dunn & Parberry 2011, 121–122.) Kaavan 26 mukaisessa esityksessä olevaa vasenta rivivektoria voidaan pitää 1 X 3 matriisina, ja oikeaa sarakevektoria 3 X 1 matriisina.

$$[x \quad y \quad z] = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \tag{26}$$

Jos vektori on rivimuodossa, on vektorin oltava kertolaskussa ennen matriisia, ja jos se on sarakemuodossa, on sen oltava matriisin jälkeen. Matriisin ja vektorin kertolasku tuottaa tuloksena uuden rivi- tai sarakevektorin. (Dunn & Parberry 2011, 122.) Huomattavaa on että vektorin ulottuvuus määrää myös minkä suuruinen kertomisessa käytettävän matriisin on oltava. Kaavassa 27 esitetään kolmiulotteisen rivivektorin ja sarakevektorin kertominen 3 x 3 matriisin avulla.

Kaavasta huomataan myös, että tuloksena saatava vektori on erilainen riippuen siitä, suoritettaanko tulo rivi- vai sarakevektorilla.

$$\begin{bmatrix} x & y & z \end{bmatrix} \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix} \tag{27}$$

$$= \begin{bmatrix} xm_{11} + ym_{21} + zm_{31} & xm_{12} + ym_{22} + zm_{32} & xm_{13} + ym_{23} + zm_{33} \end{bmatrix},$$

$$\begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} xm_{11} + ym_{12} + zm_{13} \\ xm_{21} + ym_{22} + zm_{23} \\ xm_{31} + ym_{32} + zm_{33} \end{bmatrix}$$

Kirjastossa matriisin ja vektorin väliselle tulolle on määritelty Matrix3f-luokan multVec3-metodi, joka suorittaa tulon sitä kutsuvan matriisin ja parametrina annetun Vector3f-objektin välillä. Tulo suoritetaan sarakevektorin mallisesti, eli matriisin rivien suuntaiset arvot kerrotaan vektorin arvojen kanssa. Tulos muodostetaan uutteen vektoriin, joka palautetaan paluuarvona. Alla oleva koodi esittelee multVec3-metodin suorittamaa tuloa matriisin ja vektorin välillä.

```
WEBGL_LIB.Math.Entities.Matrix3f.prototype.multVec3 =
                                                    function(vector3f){
    var result = new WEBGL_LIB.Math.Entities.Vector3f(0, 0, 0);
    result.x = vector3f.x * this.array[0] + vector3f.y * this.array[3] +
                                            vector3f.z * this.array[6];
    result.y = vector3f.x * this.array[1] + vector3f.y * this.array[4] +
                                            vector3f.z * this.array[7];
    result.z = vector3f.x * this.array[2] + vector3f.y * this.array[5] +
                                            vector3f.z * this.array[8];
    return result;
};
```

2.2.3 Matriisin determinantti ja käänteismatriisi

Neliömatriiseille on olemassa erityinen skalaarilukuarvo nimeltään matriisin determinantti. Determinantti 2x2 matriisille lasketaan kaavan 28 mukaisella lasku-

toimituksella ja 3 x 3 matriisin determinantti lasketaan kaavan 29 mukaisella laskutoimituksella. (Dunn & Parberry 2011, 162–163.)

$$det\boldsymbol{M} = det\begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{bmatrix} \\ = m_{11}m_{22} - m_{12}m_{21} \tag{28}$$

$$det\boldsymbol{M} = det\begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix} \\ = m_{11}m_{22}m_{33} + m_{12}m_{23}m_{31} + m_{13}m_{21}m_{32} \\ -m_{11}m_{23}m_{32} - m_{12}m_{21}m_{33} - m_{13}m_{22}m_{31} \tag{29}$$

3 x 3 matriisin determinantti on mahdollista laskea myös vektoreiden avulla kaavassa 30 esitetyllä tavalla. Kaavassa 30 3 x 3 matriisin jokainen rivi muunnetaan 3D vektoreiksi esim. $\mathbf{v_1}$, $\mathbf{v_2}$ ja $\mathbf{v_3}$, minkä jälkeen $\mathbf{v_1}$ ja $\mathbf{v_2}$ suoritetaan ristitulo, ja tästä saatavalle vektorille suoritetaan pistetulo vektorin $\mathbf{v_3}$ kanssa. (Dunn & Parberry 2011, 163.)

$$det\boldsymbol{M} = det\begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix} \\ \boldsymbol{v_1} = [m_{11} \quad m_{12} \quad m_{13}] \\ \rightarrow \boldsymbol{v_2} = [m_{21} \quad m_{22} \quad m_{23}] \rightarrow (\boldsymbol{v_1} \times \boldsymbol{v_2}) \cdot \boldsymbol{v_3} \\ \boldsymbol{v_3} = [m_{31} \quad m_{32} \quad m_{33}] \tag{30}$$

Kirjastossa Matrix3f-luokalle on määritelty determinant-metodi, joka palauttaa paluuarvona sitä kutsuneen matriisin determinantin määrittävän liukulukuarvon, joka on muodostettu kaavan 29 mukaisella tavalla. Alla olevassa koodissa esitellään metodin toiminta.

```
WEBGL_LIB.Math.Entities.Matrix3f.prototype.determinant = function(){
    var a = this.array;
    return a[0]*a[4]*a[8] + a[3]*a[7]*a[2] + a[6]*a[1]*a[5] –
           a[0]*a[7]*a[5] - a[3]*a[1]*a[8] - a[6]*a[4]*a[2];
};
```

Matriisin determinantin ja myöhemmin liittomatriisin laskemiseen liittyy matriisin alkioiden komplementit eli kofaktorit ja alideterminantit. Pienemmillä matriiseilla determinantin laskemiseen kofaktoreita ja alideterminantteja ei välttämättä tarvita, mutta aina 4 x 4 matriisista suurempiin matriiseihin niiden käyttäminen on helpompi laskutapa, ja ne ovat hyödyllisiä myös liittomatriisin laskemisessa. (Dunn & Parberry 2011, 164.)

Alideterminantit ovat determinantteja, joissa determinantin laskemiseen käytetään vain osaa matriisista. Alideterminantti merkittään usein yläindeksissä olevilla luvuilla kaavan 31 mukaisella tavalla, jotka kertovat, mikä rivi ja sarake matriisista tulee poistaa, minkä jälkeen jäljelle jääneestä matriisin osasta lasketaan determinantti:

$$\boldsymbol{M} = \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix}, \tag{31}$$

$$\boldsymbol{M}^{\{11\}} = \begin{bmatrix} \cancel{m_{11}} & \cancel{m_{12}} & \cancel{m_{13}} \\ \cancel{m_{21}} & m_{22} & m_{23} \\ \cancel{m_{31}} & m_{32} & m_{33} \end{bmatrix} = det \begin{bmatrix} m_{22} & m_{23} \\ m_{32} & m_{33} \end{bmatrix} = m_{22}m_{33} - m_{23}m_{32,}$$

$$\boldsymbol{M}^{\{12\}} = \begin{bmatrix} \cancel{m_{11}} & \cancel{m_{12}} & \cancel{m_{13}} \\ m_{21} & \cancel{m_{22}} & m_{23} \\ m_{31} & \cancel{m_{32}} & m_{33} \end{bmatrix} = det \begin{bmatrix} m_{21} & m_{23} \\ m_{31} & m_{33} \end{bmatrix} = m_{21}m_{33} - m_{23}m_{32,},$$

$$\boldsymbol{M}^{\{21\}} = \begin{bmatrix} \cancel{m_{11}} & m_{12} & m_{13} \\ \cancel{m_{21}} & \cancel{m_{22}} & \cancel{m_{23}} \\ \cancel{m_{31}} & m_{32} & m_{33} \end{bmatrix} = det \begin{bmatrix} m_{12} & m_{13} \\ m_{32} & m_{33} \end{bmatrix} = m_{12}m_{33} - m_{13}m_{32,},$$

Neliömatriisin kofaktorit tietylle riville ja sarakkeelle on sama alideterminantti samoilla rivi- ja sarakearvoilla, mutta kofaktorin sijainnista riippuen, saatu alideterminantti täytyy kertoa luvulla 1 tai -1. Tämä nähdään kaavassa 32. Jos täysin kofaktoroitu tulosmatriisi transponoidaan, tuloksena saatu matriisi on samalla myös adjungoitu 4 x 4 matriisi. Adjungoitua matriisia tarvitaan matriisin käänteismatriisin laskemisessa. (Dunn & Parberry 2011, 164, 169.)

4 x 4 matriisin tapauksessa matriisin determinantti voidaan laskea laskemalla kofaktorit matriisin ensimmäiseltä riviltä, ja kertomalla ne ensimmäisen rivin vastaavan elementin arvolla kaavan 33 mukaisesti, jossa esitettynä 4 x 4 matriisin determinantin laskeminen kofaktoreiden ja alideterminanttien avulla. Alidetermi-

nanttien avulla voidaan laskea sekä isompien ja pienempienkin matriisien determinantteja samalla periaatteella. (Dunn & Parberry 2011, 165.)

$$\begin{bmatrix} + & - & + & - \\ - & + & - & + \\ + & - & + & - \\ - & + & - & + \end{bmatrix}, \boldsymbol{M} = \begin{bmatrix} C^{\{11\}} & C^{\{12\}} & C^{\{13\}} & C^{\{14\}} \\ C^{\{21\}} & C^{\{22\}} & C^{\{23\}} & C^{\{24\}} \\ C^{\{31\}} & C^{\{32\}} & C^{\{33\}} & C^{\{34\}} \\ C^{\{41\}} & C^{\{42\}} & C^{\{43\}} & C^{\{44\}} \end{bmatrix} = \begin{bmatrix} +M^{\{11\}} & -M^{\{12\}} & +M^{\{13\}} & -M^{\{14\}} \\ -M^{\{21\}} & +M^{\{22\}} & -M^{\{23\}} & +M^{\{24\}} \\ +M^{\{31\}} & -M^{\{32\}} & +M^{\{33\}} & -M^{\{34\}} \\ -M^{\{41\}} & +M^{\{42\}} & -M^{\{43\}} & +M^{\{44\}} \end{bmatrix}, \tag{32}$$

$$= \begin{bmatrix} +det\begin{bmatrix} m_{22} & m_{23} & m_{24} \\ m_{32} & m_{33} & m_{34} \\ m_{42} & m_{43} & m_{44} \end{bmatrix} & -det\begin{bmatrix} m_{21} & m_{23} & m_{24} \\ m_{31} & m_{33} & m_{34} \\ m_{41} & m_{43} & m_{44} \end{bmatrix} & +det\begin{bmatrix} m_{21} & m_{22} & m_{24} \\ m_{31} & m_{32} & m_{34} \\ m_{41} & m_{42} & m_{44} \end{bmatrix} & -det\begin{bmatrix} m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \\ m_{41} & m_{42} & m_{43} \end{bmatrix} \\ -det\begin{bmatrix} m_{12} & m_{13} & m_{14} \\ m_{32} & m_{33} & m_{34} \\ m_{42} & m_{43} & m_{44} \end{bmatrix} & +det\begin{bmatrix} m_{11} & m_{13} & m_{14} \\ m_{31} & m_{33} & m_{34} \\ m_{41} & m_{43} & m_{44} \end{bmatrix} & -det\begin{bmatrix} m_{11} & m_{12} & m_{14} \\ m_{31} & m_{32} & m_{34} \\ m_{41} & m_{42} & m_{44} \end{bmatrix} & +det\begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{31} & m_{32} & m_{33} \\ m_{41} & m_{42} & m_{43} \end{bmatrix} \\ +det\begin{bmatrix} m_{12} & m_{13} & m_{14} \\ m_{22} & m_{23} & m_{24} \\ m_{42} & m_{43} & m_{44} \end{bmatrix} & -det\begin{bmatrix} m_{11} & m_{13} & m_{14} \\ m_{21} & m_{23} & m_{24} \\ m_{41} & m_{43} & m_{44} \end{bmatrix} & +det\begin{bmatrix} m_{11} & m_{12} & m_{14} \\ m_{21} & m_{22} & m_{24} \\ m_{41} & m_{42} & m_{44} \end{bmatrix} & -det\begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{41} & m_{42} & m_{43} \end{bmatrix} \\ -det\begin{bmatrix} m_{12} & m_{13} & m_{14} \\ m_{22} & m_{23} & m_{24} \\ m_{32} & m_{33} & m_{34} \end{bmatrix} & +det\begin{bmatrix} m_{11} & m_{13} & m_{14} \\ m_{21} & m_{23} & m_{24} \\ m_{31} & m_{33} & m_{34} \end{bmatrix} & -det\begin{bmatrix} m_{11} & m_{12} & m_{14} \\ m_{21} & m_{22} & m_{24} \\ m_{31} & m_{32} & m_{34} \end{bmatrix} & +det\begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix} \end{bmatrix},$$

$$= det\begin{bmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ m_{41} & m_{42} & m_{43} & m_{44} \end{bmatrix} \tag{33}$$

$$= m_{11}M^{\{11\}} - m_{12}M^{\{12\}} + m_{13}M^{\{13\}} - m_{14}M^{\{14\}}$$

$$= m_{11}det\begin{bmatrix} m_{22} & m_{23} & m_{24} \\ m_{32} & m_{33} & m_{34} \\ m_{42} & m_{43} & m_{44} \end{bmatrix} - m_{12}det\begin{bmatrix} m_{21} & m_{23} & m_{24} \\ m_{31} & m_{33} & m_{34} \\ m_{41} & m_{43} & m_{44} \end{bmatrix}$$

$$+m_{13}det\begin{bmatrix} m_{21} & m_{22} & m_{24} \\ m_{31} & m_{32} & m_{34} \\ m_{41} & m_{42} & m_{44} \end{bmatrix} - m_{14}det\begin{bmatrix} m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \\ m_{41} & m_{42} & m_{43} \end{bmatrix},$$

Kofaktoreiden laskemista hyödynnetään matriisin liittomatriisin, eli adjungoidun matriisin laskemisessa. Liittomatriisi lasketaan muodostamalla uusi matriisi matriisin kofaktoreista, jolle suoritetaan tämän jälkeen transpoosi. Kaavassa 34 esitetään 3 x 3 matriisin liittomatriisin laskeminen muodostamalla ensin matriisin kofaktoreista uusi matriisi, jolle suoritetaan lopuksi transpoosi. (Dunn & Parberry 2011, 169.)

$$adj\boldsymbol{M} = \begin{bmatrix} C^{\{11\}} & C^{\{12\}} & C^{\{13\}} \\ C^{\{21\}} & C^{\{22\}} & C^{\{23\}} \\ C^{\{31\}} & C^{\{32\}} & C^{\{33\}} \end{bmatrix}^T, \tag{34}$$

$$adj\boldsymbol{M} = \begin{bmatrix} +C^{\{11\}} & -C^{\{12\}} & +C^{\{13\}} \\ -C^{\{21\}} & +C^{\{22\}} & -C^{\{23\}} \\ +C^{\{31\}} & -C^{\{32\}} & +C^{\{33\}} \end{bmatrix}^T$$

$$= \begin{bmatrix} +det\begin{bmatrix} m_{22} & m_{23} \\ m_{32} & m_{33} \end{bmatrix} & -det\begin{bmatrix} m_{21} & m_{23} \\ m_{31} & m_{33} \end{bmatrix} & +det\begin{bmatrix} m_{21} & m_{22} \\ m_{31} & m_{32} \end{bmatrix} \\ -det\begin{bmatrix} m_{12} & m_{13} \\ m_{32} & m_{33} \end{bmatrix} & +det\begin{bmatrix} m_{11} & m_{13} \\ m_{31} & m_{33} \end{bmatrix} & -det\begin{bmatrix} m_{11} & m_{12} \\ m_{31} & m_{32} \end{bmatrix} \\ +det\begin{bmatrix} m_{12} & m_{13} \\ m_{22} & m_{23} \end{bmatrix} & -det\begin{bmatrix} m_{11} & m_{13} \\ m_{21} & m_{23} \end{bmatrix} & +det\begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{bmatrix} \end{bmatrix}^T$$

$$= \begin{bmatrix} +det\begin{bmatrix} m_{22} & m_{23} \\ m_{32} & m_{33} \end{bmatrix} & -det\begin{bmatrix} m_{12} & m_{13} \\ m_{32} & m_{33} \end{bmatrix} & +det\begin{bmatrix} m_{12} & m_{13} \\ m_{22} & m_{23} \end{bmatrix} \\ -det\begin{bmatrix} m_{21} & m_{23} \\ m_{31} & m_{33} \end{bmatrix} & +det\begin{bmatrix} m_{11} & m_{13} \\ m_{31} & m_{33} \end{bmatrix} & -det\begin{bmatrix} m_{11} & m_{13} \\ m_{21} & m_{23} \end{bmatrix} \\ +det\begin{bmatrix} m_{21} & m_{22} \\ m_{31} & m_{32} \end{bmatrix} & -det\begin{bmatrix} m_{11} & m_{12} \\ m_{31} & m_{32} \end{bmatrix} & +det\begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{bmatrix} \end{bmatrix}$$

Kirjastossa Matrix3f-luokalle on määritetty adjointMatrix-metodi, joka palauttaa paluuarvona uuden matriisin, joka on metodia kutsuneen matriisin adjungtio. Palautettavan matriisin jokaiseen alkioon lasketaan metodia kutsuneen matriisin kofaktorit, ja palautettavalle matriisille suoritettaan lopuksi transpoosi Matrix3f-luokan transpose-metodin avulla.

```
WEBGL_LIB.Math.Entities.Matrix3f.prototype.adjointMatrix = function(){
    var a = this.array;
    var resultArray = [];
    resultArray.push( a[4]*a[8]-a[7]*a[5]);   //+Cof 11
    resultArray.push(-(a[3]*a[8]-a[6]*a[5])); //-Cof 21
    resultArray.push( a[3]*a[7]-a[6]*a[4]);   //+Cof 31
    resultArray.push(-(a[1]*a[8]-a[7]*a[2])); //-Cof 12
    resultArray.push( a[0]*a[8]-a[6]*a[2]);   //+Cof 22
    resultArray.push(-(a[0]*a[7]-a[6]*a[1])); //-Cof 32
    resultArray.push( a[1]*a[5]-a[4]*a[2]);   //+Cof 13
    resultArray.push(-(a[0]*a[5]-a[3]*a[2])); //-Cof 23
    resultArray.push( a[0]*a[4]-a[3]*a[1]);   //+Cof 33

    return new WEBGL_LIB.Math.Entities.Matrix3f(resultArray).
```

```
                                                    transposeMatrix();
};
```

Matriisin käänteismatriisi on matriisi, joka määrittää käänteisen transformaatioin alkuperäiseen matriisiin määrittämään transformaatioon verrattuna. Jos matriisi suorittaisi vektorille jonkin transformaation kuten translaation johonkin suuntaan, suorittaisi matriisin käänteismatriisi vektorille translaation vastakkaiseen suuntaan. Jos transformaatiomatriisi ja sen käänteismatriisi suoritettaisiin peräjälkeen, ei vektorille tapahtuisi mitään, sillä matriisit kumoavat toistensa muutokset. Jos matriisi kerrotaan omalla käänteismatriisillaan, saadaan tulokseksi identiteettimatriisi. (Dunn & Parberry 2011, 171.)

Matriisin käänteismatriisi voidaan laskea jakamalla adjungoitu matriisi matriisin determinantilla. Jakamisen voi suorittaa helpommin jakamalla arvo 1 determinantilla, ja kertomalla adjungoidun matriisin siitä saatavalla skalaari arvolla. Kaavassa 35 esitetään matriisin käänteismatriisin laskemiseen käytettävä kaava.

$$\boldsymbol{M}^{-1} = \frac{adj\boldsymbol{M}}{det\boldsymbol{M}} = \frac{1}{det\boldsymbol{M}} adj\boldsymbol{M} \tag{35}$$

Kirjastossa Matrix3f-luokalle on määritelty inverseMatrix-metodi, jolla voidaan luoda käänteismatriisi metodia kutsuneen matriisin sisältämistä arvoista. Metodi käyttää hyödykseen luokalle määriteltyjä adjointMatrix- ja determinant-metodeita. Käänteismatriisi muodostetaan kaavan 35 mukaisella tavalla, ja metodia kutsuneen matriisin arvoista muodostettu käänteismatriisi palautetaan paluuarvona. Alla oleva koodi esittää metodin suorittaman toiminnallisuuden.

```
WEBGL_LIB.Math.Entities.Matrix3f.prototype.inverseMatrix = function(){
      var adjMat = new WEBGL_LIB.Math.Entities.Matrix3f(this.array).
                                                    adjointMatrix();
      var det = this.determinant();
      adjMat.mulScal(1/det);
      return adjMat;
};
```

2.2.4 Transformaatioiden määrittäminen matriiseilla

Homogeenisillä koordinaateilla tarkoitetaan sitä, kun 2D tai 3D avaruuden vektoreille ja pisteille annetaan ylimääräinen komponentti *w*, joka laajentaa avaruutta 2D avaruudesta homogeeniseen 3D avaruuteen, ja 3D avaruutta homogeeniseen 4D avaruuteen (kaava 36). Homogeeniset koordinaatit voidaan muuntaa 2D tai 3D sijainneiksi jakamalla muut komponentit w:n arvolla, jos w != 0 (kaava 37). (Dunn & Parberry 2011, 177.)

$$[x \quad y] \rightarrow [x \quad y \quad w] \qquad (36)$$
$$[x \quad y \quad z] \rightarrow [x \quad y \quad z \quad w]$$

$$[x \quad y \quad w] \rightarrow [x/w \quad y/w] \rightarrow [x \quad y] \qquad (37)$$
$$[x \quad y \quad z \quad w] \rightarrow [x/w \quad y/w \quad z/w] \rightarrow [x \quad y \quad z]$$

Tärkeimpiä syitä käyttää homogeenisiä vektoreita, on sen luoma notaatiollinen yksinkertaisuus translaation toteuttamisessa, sekä perspektiivillisen projektion toteuttamisessa (Dunn & Parberry 2011, 178). Näistä syistä johtuen, 3D-grafiikassa käytetään 4D vektoreita ja 4 x 4 matriiseja. Kirjasto ei erikseen määrittele luokkaa neliulotteisille vektoreille, mutta niiden kanssa joudutaan toimimaan WebGL-rajapinnan käyttöön määritettyjen varjostimien (engl. shaders) kanssa, jossa translaatioita ja projektioita määritteleviä matriiseja sijoitetaan piirrettävien 3D-mallien muodostamiin vektoriarvoihin.

Matriiseilla voidaan kätevästi esittää monia transformaatioita, joita vektoreille halutaan suorittaa, ja nämä transformaatiot sijoitetaan vektoriin, suorittamalla vektorin ja matriisin välinen tulo. Eri transformaatiomatriisit voidaan yhdistää yhdeksi kertomalla matriisit keskenään, minkä jälkeen tuloksena saatavalla matriisilla voidaan suorittaa kaikki transformaatiot kertomalla kaikki halutut vektorit tuloksena saadun matriisin kanssa. (Puhakka 2008, 96.)

Geometrisesti katsottuna, matriisien voidaan olettaa toteuttavan muutoksia kaavan 38 mukaisesti. Kaavassa määritetään perusvektorit, jotka osoittavat x-. y- ja

z-akselin suuntaisesti. Kun nuo vektorit kerrotaan matriisin kanssa, huomataan mitkä matriisin alkioista vaikuttavat kunkin akselin suuntaisesti vektoriin. (Dunn & Parberry 2011, 125.)

$$\boldsymbol{Mx} = \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} m_{11} \\ m_{12} \\ m_{13} \end{bmatrix}, \tag{38}$$

$$\boldsymbol{My} = \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} m_{21} \\ m_{22} \\ m_{23} \end{bmatrix},$$

$$\boldsymbol{Mz} = \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} m_{31} \\ m_{32} \\ m_{33} \end{bmatrix},$$

Lineaariset transformaatiot ovat toteutettavissa 3 x 3 matriiseilla, mutta jotta niihin voidaan helposti yhdistää kappaleen translaatio, täytyy ne siirtää 4 x 4 matriisiin, jolla transformaatiot voidaan helposti yhdistää homogeenisiin 4D vektoreihin. Tämä ei ole kuitenkaan iso muunnos, joten lineaariset transformaatiot esitellään kaavoissa 3 x 3 matriiseina.

Skaalaus on transformaatioista yksinkertaisin, ja siinä kappaletta skaalataan avaruuden jokaisen akselin suunnassa, pitäen kuitenkin kappaleen keskipisteen paikallaan (Puhakka 2008, 90). Skaalauksen muodostava matriisi on esitetty kaavassa 39. Jokaiselle akselille asetetaan oma skaalauskerroin, jonka verran kappaletta skaalataan. Jos skaalauksen määrä on 1, ei kappale skaalaudu akselin suunnassa. Kaavassa on 3 x 3 skaalausmatriisi **S**, jossa arvo s_x skaalaa kappaletta x-akselin suuntaisesti, arvo s_y y-akselin suuntaisesti ja arvo s_x z-akselin suuntaisesti.

$$\mathbf{S} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{bmatrix} \tag{39}$$

Kirjastossa on määritelty funktio WEBGL_LIB.Math-nimiavaruuteen nimeltä getScaleMat4f, joka ottaa parametreina x, y ja z akseleiden suunnassa toteuttavan skaalauksen, jonka palautettavan matriisin tulee määrittää. Arvoja käytetään uuden Matrix4f-objektin alustuksessa, ja tuloksena saatava matriisi määrit-

telee skaalauksen. Alla olevassa koodissa nähdään skaalausmatriisin muodostavan getScaleMat4f-funktion toiminta.

```
WEBGL_LIB.Math.getScaleMat4f = function(sx, sy, sz){
      var scaleMat4 = new WEBGL_LIB.Math.Entities.Matrix4f([
                                                sx, 0, 0, 0,
                                                0, sy, 0, 0,
                                                0, 0, sz, 0,
                                                0, 0, 0, 1 ]);
      return scaleMat4;
};
```

Kappaleen kiertäminen avaruuden perusakseleiden ympäri voidaan toteuttaa melko yksinkertaisilla matriiseilla. Kaavassa 40 esitetään, kuinka kunkin akselin ympäri radiaaneina ilmoitetun kulman θ suuruisen kierron toteuttavat matriisit voidaan muodostaa. Matriisi $\mathbf{R_x}$ kiertää kappaletta x-akselin ympäri, matriisi $\mathbf{R_y}$ kiertää kappaletta y-akselin ympäri ja matriisi $\mathbf{R_z}$ kiertää kappaletta z-akselin ympäri. (Dunn & Parberry 2011, 140–141.) Tämän jälkeen matriisit voidaan kertoa keskenään, jolloin saadaan yksi matriisi, joka kiertää kappaletta jokaisen kolmen akselin ympäri halutun määrän.

$$\mathbf{R_x} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & \sin\theta \\ 0 & -\sin\theta & \cos\theta \end{bmatrix}$$
$$\mathbf{R_y} = \begin{bmatrix} \cos\theta & 0 & -\sin\theta \\ 0 & 1 & 0 \\ \sin\theta & 0 & \cos\theta \end{bmatrix} \quad (40)$$
$$\mathbf{R_z} = \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Kirjastossa on määritelty WEBGL_LIB.Math-nimiavaruuteen funktiot, jotka määrittelevät kierron x-, y- ja z-akseleiden ympäri. Näiden funktioiden nimet ovat getXRotationMat4f, getYRotationMat4f ja getZRotationMat4f, ja ne kaikki ottavat parametreina asteina annetun kulman, joka on akselin ympäri toteutettavan kierron määrä. Asteet muutetaan radiaaneiksi ennen matriisin muodostamista, ja matriisi muodostettaan kaavan 40 mukaisella tavalla jokaisessa funktiossa.

Alla olevassa koodissa esitetään getXRotationMat4f-funktio, joka palauttaa paluuarvona matriisin, joka määrittelee parametrina annetun kulman suuruisen kierron x-akselin ympäri.

```
WEBGL_LIB.Math.getXRotationMat4f = function(rotationAngle){
    var rotationRad = rotationAngle * (Math.PI/180.0);
    var rotMat4 = new WEBGL_LIB.Math.Entities.Matrix4f([
                  1, 0, 0, 0,
                  0, Math.cos(rotationRad), -Math.sin(rotationRad), 0,
                  0, Math.sin(rotationRad), Math.cos(rotationRad), 0,
                  0, 0, 0, 1 ]);
      return rotMat4;
};
```

Kierto mihin tahansa suuntaan kulkevan akselin ympäri sen sijaan on monimutkaisempi matriisi laskea. Kierto satunnaiseen suuntaan osoittavan akselin ympäri on myös viisaampaa toteuttaa kvaternioiden avulla siksi, että se vaatii vähemmän komponentteja (4 lukua 16 luvun sijaan 4 x 4 matriiseilla), sekä kvaternioiden hyöty kierron interpoloinnissa kahden orientaation välillä, mikä on hyödyllistä esim. animaatioissa. (Puhakka 2008, 112.)

Translaation esittämistä varten käytetään 4 x 4 matriiseja, jolloin niiden esittämisestä tulee yksinkertaista, ja niiden liittäminen muihin transformaatiomatriiseihin on helppoa.

Translaatio tapahtuu kaavan 41 mukaisesti sijoittamalla kunkin akselin suunnassa haluttu translaation määrä kaavassa esitetyn matriisin mukaisesti. Komponentti t_x määrittää siirtymän x-akselin suunnassa, t_y y-akselin suunnassa ja t_z z-akselin suunnassa.

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{41}$$

.

Kirjastossa on määritelty muutamia erilaisia funktioita translaatioin totuttavan matriisin muodostamiseen. Yksi näistä on WEBGL_LIB.Math-nimiavaruudessa sijaitseva getTranslationMat4f-funktio, joka ottaa parametrina kolme arvoa, jotka määrittävät toteutettavan siirtymän x-, y- ja z-akseleiden suuntaisesti. Paluuarvona palautettaan uusi Matrix4f-objekti, joka määrittelee parametreina annettujen arvojen mukaisen siirtymän. Alla olevassa koodissa nähdään getTranslationMat4f-funktion toiminta.

```
WEBGL_LIB.Math.getTranslationMat4f = function(x, y, z){
      var transMat4 = new WEBGL_LIB.Math.Entities.Matrix4f([
                                                1, 0, 0, 0,
                                                0, 1, 0, 0,
                                                0, 0, 1, 0,
                                                x, y, z, 1 ]);
      return transMat4;
};
```

2.3 Kvaterniot ja niiden oparaatiot

Kvaternioita alettiin hyödyntää kiertojen ja orientaatioiden esittämiseen 3D-grafiikassa 1980-luvulla. Kvaterniot ovat kompleksilukuja. Kompleksiluvulla on sekä reaaliosa, että imaginaariosa *i*. Imaginaariosa noudattaa kaavan 42 mukaista periaatetta, jossa imaginaariosan kertominen itsellään tuottaa tulokseksi arvon -1. (Puhakka 2008, 107.)

$$i^2 = -1 \tag{42}$$

Kompleksiluku a + bi voidaan ymmärtää myös lukupariksi *(a, b)*, jolloin se voidaan suoraan muuttaa pisteeksi 2D-tasolla eli kompleksitasolla. (Puhakka 2008, 108.)

Kompleksiluvun laajentaminen kvaternioksi tapahtuu lisäämällä käyttöön kolme imaginaariyksikköä, ja kaavan 43 mukaisesti, kvaternion **q** esittäminen tapahtuu

yhdistelmänä, jossa on yksi reaaliluku ja kolme imaginaarilukua. Imaginaariyksiköt käyttäytyvät myös samalla tavalla kerrottaessa itsellään, ja jos kaikki imaginaariyksiköt kerrotaan keskenään. (Puhakka 2008, 109.)

$$\boldsymbol{q} = a + bi + cj + dk \tag{43}$$
$$i^2 = j^2 = k^2 = -1 = ijk$$

3D grafiikassa kvaternio esitetään usein vektorin tapaisena objektina, jossa imaginaariosana toimii 3D vektori, ja reaaliosana skalaari luku. Kaavassa 44 nähdään kuinka kvaternio on määritelty skalaariarvosta, ja vektorista. (Puhakka 2008, 110.) Kaavoissa joissa käsitellään kvaternioita, sekä kvaterniomuuttujat että vektorimuuttujat esitetään lihavoituina pienaakkosina, mutta vektori muuttujia korostetaan muuttujan päälle asetetulla nuolimerkinnällä, kuten kaavassa 44. Tällä selkeytetään milloin kyseessä on vektori ja milloin kvaternio.

$$\boldsymbol{q} = (s, \vec{\boldsymbol{v}}) = [s \quad v_x \quad v_y \quad v_z] = s + v_x i + v_y j + v_z k, \tag{44}$$

Skalaarikomponenttiin viitataan usein kirjaimella *w*. Vektorin komponentteihin viitataan joko yksittäisellä kirjaimella, tai vektorin yksittäisinä komponentteina. (Dunn & Parberry 2011, 247–248.) Tämä esitystapa nähdään kaavassa 45.

$$[s \quad \vec{\boldsymbol{v}}] \rightarrow [w \quad \vec{\boldsymbol{v}}], \tag{45}$$
$$[s \quad v_x \quad v_y \quad v_z] \rightarrow [w \quad x \quad y \quad z]$$

Kirjastossa kvaternioiden esittämiseen on luotu oma luokka nimeltä Quaternion (WEBGL_LIB.Quaternion), joka muodostaa kvaternion kahdella jäsenmuuttujalla, joista toinen on skalaariarvoinen w arvo, ja toinen vektoria esittävä Vector3f-objekti. Alla olevassa koodissa nähdään Quaternion-luokan rakentajafunktio, joka ottaa vastaan kaksi parametria, reaaliosan määrittävän skalaariarvon ja imaginaariosana toimivan Vector3f-olion.

```
WEBGL_LIB.Math.Entities.Quaternion = function(w, vector3f){
      this.w = 1.0;
      this.vector = new WEBGL_LIB.Math.Entities.Vector3f(0.0, 0.0, 0.0);
```

```
        if(w){
              this.w = w;
        }
        if(vector3f){
               this.vector.x = vector3f.x;
               this.vector.y = vector3f.y;
               this.vector.z = vector3f.z;
        }
};
```

Imaginaariluvulla voidaan kaksiulotteisessa kompleksitasossa suorittaa pisteelle kierto kulman θ verran origon ympäri kertomalla se kaavassa 46 esitetyllä kompleksiluvulla (Puhakka 2008, 111).

$$e^{i\theta} = cos\theta + i\ sin\theta, \tag{46}$$

Kvaterniot helpottavat rotaation esittämistä 3D:ssä. Yleisesti voidaan ajatella, että w on sama kuin suorittavan kierron suuruus, joka tapahtuu, ja **v** edustaa akselin suuntaa, jonka ympäri kierto tapahtuu. Kaavassa 47 nähdään kierron θ suorittavan kvaternion muodostaminen. Kulmana θ esitetyn kierron määrä puolitetaan jakamalla se kahdella, minkä jälkeen kvaternion w-komponentin arvo asetetaan puolitetun kierron kosinilla, ja kvaternion vektorin arvot asetetaan kertomalla vektorin arvot puolitetun kierron sinillä. Vektorin määrittelemä suunta ennen puolikkaan kulman sinillä kertomista määrittää suunnan, jonka ympäri kierto tehdään. (Puhakka 2008, 111–112.)

$$[w \quad \vec{\boldsymbol{v}}] = [cos(\theta/2) \quad sin(\theta/2)\vec{\boldsymbol{v}}], \tag{47}$$
$$[w \quad x \quad y \quad z] = [cos(\theta/2) \quad sin(\theta/2)n_x \quad sin(\theta/2)n_y \quad sin(\theta/2)n_z]$$

Kaava Q. 3D avaruudessa kierron suorittavan kvaternion määrittäminen.

Luotu kirjasto määrittelee WEBGL_LIB.Math-nimiavaruudessa sijaitsevan getRotationQuat-funktion, joka ottaa parametreina kierron suuruuden määrittävän radiaani kulman, sekä Vector3f-objektin, joka määrittää akselin jonka ympäri

kierto toteutetaan. Annettu kulma puolitetaan, ja akselin määrittelevä vektori normalisoidaan, minkä jälkeen arvoista muodostetaan uusi Quaternion-objekti, jonka arvot alustetaan kaavan 47 mukaisesti kosini ja sini funktioiden avulla. Kierron määrittelevän kvaternioinin muodostavan getRotationQuat-funktion toiminta nähdään alla olevassa koodissa.

```
WEBGL_LIB.Math.getRotationQuat = function(angle, axisVector){
    axisVector.normalize();
    var rot = angle ? angle / 2 : 0;
    var qX = axisVector.x * (Math.sin(rot));
    var qY = axisVector.y * (Math.sin(rot));
    var qZ = axisVector.z * (Math.sin(rot));
    var qW = Math.cos(rot);
    var rotQuat = new WEBGL_LIB.Math.Entities.Quaternion(qW,
            new WEBGL_LIB.Math.Entities.Vector3f(qX, qY, qZ));
    rotQuat.normalize();
    return rotQuat;
};
```

Kahta kompleksilukua laskettaessassa yhteen, kompleksi- ja imaginaariosat lasketaan erikseen yhteen kaavan 48 mukaisesti. Kompleksilukujen reaaliosat ja toisiaan vastaavat imaginaariosat lasketaan erikseen yhteen.
(Puhakka 2008, 107–108.)

$$a + bi, \quad c + di, \quad (a + bi) + (c + di) = a + c + (b + d)i \tag{48}$$

Kun yhteenlasku suoritetaan kvaternioille, toimii yhteenlasku täysin samalla periaatteella. Kaavassa 49 nähdään, kuinka skalaariarvolla ja vektoriarvolla esitetyt kvaterniot lasketaan yhteen. Kvaternioiden kierron suuruuden määrittelevät skalaariarvot lasketaan yhteen, ja imaginaari osana toimivat vektorit lasketaan yhteen vektoreiden yhteenlaskulla. (Puhakka 2008, 109-110.)

$$\boldsymbol{q_1} + \boldsymbol{q_2} = \left(s_1 + s_{2,}\overrightarrow{\boldsymbol{v_1}} + \overrightarrow{\boldsymbol{v_2}}\right), \tag{49}$$

Kirjastossa määritetylle Quaternion-luokalle on yhteenlaskua varten määritelty addQuaternion-metodi, joka suorittaa yhteenlaskun sitä kutsuneen ja parametrina annetun Quaternion-objektin välillä, ja palauttaa tuloksen uutena Quaternion-objektina. Alla olevassa koodissa nähdään addQuaternion-metodin toiminta.

```
WEBGL_LIB.Math.Entities.Quaternion.prototype.addQuaternion =
                                                    function(quaternion){
      var result = new WEBGL_LIB.Math.Entities.Quaternion();
      result.w = this.w + quaternion.w;
      result.vector = this.vector.addVect(quaternion.vector);
      return result;
};
```

Kaavassa 50 esitetyn kahden imaginaariluvun kertolaskussa huomataan, kuinka imaginaari luku *i* muuttuu luvuksi -1, kun se kerrotaan itsensä kanssa potenssilla (Puhakka 2008, 108).

$$\begin{aligned} &(a + bi)(c + di) \\ &= ac + adi + bci + bdi^2 \\ &= ac + (ad + bc)i + bd(-1) \\ &= ac - bd + (ad + bc)i \end{aligned} \tag{50}$$

Kvaternion yhteen ja vähennyslasku toimivat samalla periaatteella, kuin normaalien kompleksilukujen, mutta niiden kertominen keskenään on edellyttää, että tietää mitä kunkin imaginaari yksikön kertominen keskenään tuottaa tulokseksi. Johtamalla kaavasta 50 saadaan kaavan 51 esittämät tulokset. Imaginaariyksiköt tuottavat toisensa syklisesti ja antikommutoivat, eli kerrottavien järjestystä vaihtamalla tuloksen etumerkki vaihtuu päinvastaiseksi. (Puhakka 2008, 109.)

$$ij = -ji = k,$$
$$jk = -kj = i,$$
$$ki = -ik = j \tag{51}$$

Kvaternion kertominen onnistuu hyötämällä kaavan 51 määrittelemän tiedon avulla kaavan 52 mukaisesti (Puhakka 2008, 109). Kyseessä on melko suurikokoinen laskutoimitus, joten välivaiheita ei yritetä edes esittää kaavassa, vaan siinä on näkyvillä lopullinen supistettu tulos kaava, jota voidaan hyödyntää kahden kvaternion kertomisessa.

$$\boldsymbol{q_1} = a_1 + b_1 i + c_1 j + d_1 k, \tag{52}$$
$$\boldsymbol{q_2} = a_2 + b_2 i + c_2 j + d_2 k,$$
$$\boldsymbol{q_1 q_2} = a_1 a_2 - b_1 b_2 - c_1 c_2 - d_1 d_2$$
$$+(a_1 b_2 + b_1 a_2 + c_1 d_2 - d_1 c_2) i$$
$$+(a_1 c_2 - b_1 d_2 + c_1 a_2 + d_1 b_2) j$$
$$+(a_1 d_2 + b_1 c_2 - c_1 b_2 + d_1 a_2) k$$

Kun kvaterniot esitetään skalaariarvon vektorin avulla, muuttuu kertolasku seuraavaan kaavassa 53 esitettyyn muotoon, joka on kätevämpi laskutoimitus kirjaston Quaternion-luokan kanssa (Puhakka 2008, 110).

$$\boldsymbol{q_1 q_2} = \left(s_1 s_2 - \overrightarrow{\boldsymbol{v_1}} \cdot \overrightarrow{\boldsymbol{v_2}}, s_1 \overrightarrow{\boldsymbol{v_2}} + s_2 \overrightarrow{\boldsymbol{v_1}} + \overrightarrow{\boldsymbol{v_1}} \times \overrightarrow{\boldsymbol{v_2}}\right) \tag{53}$$

Quaternion-luokalle on määritetty multQuaternion-metodi, joka kertoo metodia kutsuneen Quaternion-olion parametrina annetun Quaternion-olion kanssa kaavan 53 esittämän laskutavan mukaisesti, ja palauttaa tuloksen uutena Quaternion-oliona. Kvaternioiden kertolaskun suorittavan metodin toiminta on esitettynä alla olevassa koodissa .

```
WEBGL_LIB.Math.Entities.Quaternion.prototype.multQuaternion =
                                                function(quaternion){
    var newW = this.w * quaternion.w –
               this.vector.dotProduct(quaternion.vector);
    var v1 = quaternion.vector.mulScal(this.w);
    var v2 = this.vector.mulScal(quaternion.w);
```

```
        var v3 = this.vector.crossProduct(quaternion.vector);
        var newVec = v1.addVect(v2.addVect(v3));

        return new WEBGL_LIB.Math.Entities.Quaternion(newW, newVec);
};
```

Kvaternion konjugaatti tai liittoluku, voidaan määritellä asettamalla kvaternion imaginaariosat negatiivisiksi. Konjugaatti merkitään kaavoissa yläindeksissä olevalla tähtimerkillä, kuten kaavassa 54. Kun imaginaari muodostetaan vektorista, suoritetaan vektorin arvoille negaatio. (Puhakka 2008, 109.)

$$\boldsymbol{q}^* = a - bi - cj - dk, \qquad (54)$$
$$\boldsymbol{q}^* = (s, -\vec{\boldsymbol{v}})$$

Kirjasto määrittelee Quaternion-luokalle getConjugateQuaternion-metodin, joka palauttaa paluuarvona uuden Quaternion-olion, joka on metodia kutsuneen Quaternion-olion konjugaatti. Palautettavan kvaternion arvot alustetaan metodia kutsuneen kvaternion arvoilla, mutta asetetun vektorin arvoille suoritetaan negaatio. Alla olevaa koodi esittää metodin toimintaa.

```
WEBGL_LIB.Math.Entities.Quaternion.prototype.getConjugateQuaternion =
                                                function(){
    return new WEBGL_LIB.Math.Entities.Quaternion(
                this.w,
                new WEBGL_LIB.Math.Entities.Vector3f(
                                -this.vector.x,
                                -this.vector.y,
                                -this.vector.z));
};
```

Vektoria voidaan kiertää kvaternion avulla, kun vektori muunnetaan puhtaaksi kvaternioksi. Puhdas kvaternio on kvaternio, jonka reaaliosa on 0. Puhdas kvaternio muodostetaan vektorin avulla asettamalla reaaliosa nollaksi, ja kvaternion

imaginaari osaksi asetetaan vektori kaavan 55 mukaisesti. (Puhakka 2008, 110.)

$$\boldsymbol{v} = (0, \vec{\boldsymbol{v}}) \tag{55}$$

Kun vektori on muutettu puhtaaksi kvaternioksi, voidaan sen kierto suorittaa kertomalla kierron määrittelevä kvaternio vektorista muodostetulla puhtaalla kvaterniolla ja kertomalla niiden tulos sen jälkeen kiertokvaternion konjugaatilla kaavan 56 mukaisesti. (Puhakka 2008, 111.)

$$\boldsymbol{q}\boldsymbol{v}\boldsymbol{q}^{*} \tag{56}$$

Kirjastossa Vector3f-luokalle on määritelty rotateWithQuaternion-metodi, jolle voidaan antaa parametrina kierron totuttava kvaternio. Metodia kutsuneen vektorin arvoista muodostetaan uusi Quaternion-objekti, joka on puhdas kvaternio. Parametrina annetulle kierron toteuttavalle kvaterniolle noudetaan myös konjugaatti, jota käytetään vektorin sisältävän puhtaan kvaternion kertomiseen, kun puhdas kvaternio on ensin kerrottu kierron määrittelevän kvaternion kanssa. Lopuksi palautetaan uusi vektori, jonka arvot muodostetaan kierretyn vektorin sisältävän puhtaan kvaternion x-, y- ja z-arvoista. Alla olevassa koodissa nähdään metodin toiminta vaihe vaiheelta.

```
WEBGL_LIB.Math.Entities.Vector3f.prototype.rotateWithQuaternion= func-
tion(rotQuat){
    var rotConj = rotQuat.getConjugateQuaternion();
    var vQuat = new WEBGL_LIB.Math.Entities.Quaternion(0, this.clone());
    var q1 = rotQuat.multQuaternion(vQuat);
    var resultQuat = q1.multQuaternion(rotConj);
    return new WEBGL_LIB.Math.Entities.Vector3f(
          resultQuat.vector.x, resultQuat.vector.y,resultQuat.vector.z);
};
```

Jotta kvaternion määrittelemiä kiertoja voisi siirtää WebGL-rajapinnan varjostinsovellusten käyttöön, täytyy niistä muodostaa matriiseja, jotka määrittelevät

saman kierron, minkä kvaternio määrittää. Kvaternion määrittämä kierto voidaan muodostaa 3 x 3 matriisille kaavassa 57 olevan mallin mukaisesti, mistä nähdään miten kukin komponentti sijoittuu matriisin alkioille. (Dunn & Parberry 2011, 269.)

$$\mathrm{w} + \mathrm{xi} + \mathrm{yj} + \mathrm{zk} = \begin{bmatrix} 1-2y^2-2z^2 & 2yx+2wz & 2xz-2wy \\ 2xy-2wz & 1-2x^2-2z^2 & 2yz+2wx \\ 2xz+2wy & 2yz-2wx & 1-2x^2-2y^2 \end{bmatrix} \quad (57)$$

Kirjastossa Quaternion-luokalle on määritelty getRotationMatrix-metodi, joka palauttaa 4 x 4 matriisin, joka sisältää metodia kutsuneen kvaternion määrittelemän kierron. Kierron toteuttava arvot sijoitetaan matriisin 3 x 3 osuudelle kaavan 57 mukaisesti, eikä matriisi toteuta kierron lisäksi muita transformaatioita. Alla nähdään metodin toteuttava koodi.

```
WEBGL_LIB.Math.Entities.Quaternion.prototype.getRotationMatrix =
                                                    function(){
    var x = this.vector.x;
    var y = this.vector.y;
    var z = this.vector.z;
    var w = this.w;
    var pX = (x * x);
    var pY = (y * y);
    var pZ = (z * z);

    var mat4Array = [
        1 - 2 * pY - 2 * pZ,     //m11
        2 * x * y - 2 * w * z,   //m21
        2 * x * z + 2 * w * y,   //m31
        0,                       //m41
        2 * x * y  + 2 * w * z,  //m12
        1 - 2 * pX - 2 * pZ,     //m22
        2 * y * z - 2 * w * x,   //m32
        0,                       //m42
        2 * x * z - 2 * w * y,   //m13
```

```
        2 * y * z + 2 * w * x,   //m23
        1 - 2 * pX - 2 * pY,     //m33
        0,                       //m43
        0,                       //m14
        0,                       //m24
        0,                       //m34
        1                        //m44
    ];

    return new WEBGL_LIB.Math.Entities.Matrix4f(mat4Array);
};
```

Kompleksiluvun itseisarvo eli suuruus voidaan laskea kertomalla kompleksiluku omalla konjugaatillaan ja laskemalla tuloksen neliöjuuri. Suuruuden laskeminen esitetty kaavassa 58. (Puhakka 2008, 108.)

$$
\begin{aligned}
|k| &= \sqrt{kk^*} \\
&= \sqrt{(a+bi)(a-bi)} \\
&= \sqrt{aa - abi + abi - b(-b)i^2} \\
&= \sqrt{aa - bb(-1)} \\
&= \sqrt{a^2 + b^2}
\end{aligned}
\tag{58}
$$

Kvaternion konjugaatille pätee sama periaate kuin kompleksiluvun konjugaatille, eli imaginaari osien etumerkin muuttaminen. Kvaternion ja sen konjugaatin kertominen yhteen on sama kuin neliulotteisen vektorin pistetulo itsensä kanssa, ja suuruuden laskeminen tapahtuu ottamalla neliöjuuri kvaternion ja sen konjugaatin tulosta kaavan 59 mukaisesti. (Puhakka 2008, 109–110.)

$$
\begin{aligned}
|\boldsymbol{q}| &= \sqrt{\boldsymbol{q}\boldsymbol{q}^*} \\
&= \sqrt{(a+bi+ci+di)(a-bi-ci-di)} \\
&= \sqrt{a^2 + b^2 + c^2 + d^2}
\end{aligned}
\tag{59}
$$

Kirjastossa Quaternion-luokalle on määritelty quaternionLength-metodi, jolla voidaan hakea sitä kutsuneen kvaternion suuruus. Quaternion-objektin kierron määrittelevä skalaarikomponentin ja suunnan määrittelevän vektorin komponenttien arvot lasketaan yhteen, ja yhteenlaskun tuloksesta otetaan neliöjuuri, jonka tulos palautetaan paluuarvona. Alla olevassa koodissa nähdään metodin toiminta.

```
WEBGL_LIB.Math.Entities.Quaternion.prototype.quaternionLenght = func-
tion(){
    var vLenght = this.vector.length();
    var v = this.vector;
    return Math.sqrt(
                this.w * this.w + v.x * v.x + v.y * v.y + v.z * v.z);
};
```

Kvaternion pituutta hyödynnetään kvaternion normalisoinnissa, jossa kvaternion suuruus asetetaan vastaamaan arvoa 1. Tämä tapahtuu samaan tapaan kuin vektorin normalisointi, eli kvaternion kaikki komponentit jaetaan kvaternion suuruudella, minkä jälkeen kvaternion suuruus on 1. (confuted 2015) Kirjasto määrittelee Quaternion-luokalle normalize-metodin, joka normalisoi sitä kutsuneen kvaternion, noutamalla kvaternion suuruuden quaternionLength-metodin avulla, ja jakamalla kvaternion komponentit metodista saadulla pituudella. Metodin toiminnan toteuttava koodi on esitettynä alla.

```
WEBGL_LIB.Math.Entities.Quaternion.prototype.normalize = function(){
    var length = this.quaternionLenght();
    this.vector.x /= length;
    this.vector.y /= length;
    this.vector.z /= length;
    this.w /= length;
};
```

3 WebGL-rajapinta ja sen historia

WebGL on Khronos Groupin julkaisema ja ylläpitämä matalan tason 3D-grafiikka API (Application programming interface). WebGL-rajapinnalla voidaan luoda 3D grafiikkaa verkkoselaimessa ilman erillisiä liitännäissovelluksia, hyödyntäen HTML5-kielen canvas-elementtiä. WebGL API perustuu toiseen grafiikkarajapintaan, OpenGL ES 2.0, jolla voidaan luoda 3D grafiikkaa hyödyntäviä sovelluksia sulautettuihin järjestelmiin, kuten älypuhelimiin. WebGL API:n käyttö muistuttaa hyvin paljon OpenGL ES 2.0 API:n käyttötapaa, joten siirtymä näiden rajapintojen välillä on suhteellisen helppoa. (Khronos Group 2015a.)

WebGL API tarjoaa monenlaisia hyötyjä ohjelmoijalle. Se tuo laitteistokiihdytetyn 3D-grafiikan selaimiin, jolloin grafiikan piirtämiseen käytettävissä oleva suoritusteho kasvaa huomattavasti. Se toimii useimmilla uusilla selaimilla ilman erillisiä liitännäisiä, sekä useammalla eri käyttöalustalla, kuten tietokoneella ja mobiililaitteilla, joiden grafiikkasuorittimet tukevat WebGL rajapintaa. WebGL ohjelmistojen kehitys on myös ketterää, koska koodia ei tarvitse linkittää tai kääntää ennen sen ajamista. (Khronos Group 2011.)

WebGL rajapinta perustuu OpenGL ES 2.0 rajapintaan, joka perustuu OpenGL 2.0 rajapintaan. OpenGL 2.0 julkaistiin vuonna 2004, ja se toi ohjelmoijille mukanaan mahdollisuuden kirjoittaa verteksi- ja fragmenttivarjostimet, jota piirtolinjasto (engl. rendering pipeline) käyttää grafiikkaa piirrettäessä. OpenGL ES 2.0 rajapinta julkaistiin vuonna 2007, ja sen tarkoitus oli tuoda laitteistokiihdytettyä 3D-grafiikkaa tietokoneita vähätehoisemmille sulatetuille järjestelmille, kuten älypuhelimet. (Danchilla 2012, xxi.)

Kuitenkin, koska OpenGL ES 2.0 rajapinta on suunnattu toimimaan sulautetuissa järjestelmissä, on se rajoitetumpi toiminnallisuuksiltaan kuin OpenGL 2.0 rajapinta. Useimmat OpenGL 2.0 rajapinnan toiminnallisuudet kuitenkin löytyvät silti OpenGL ES 2.0 rajapinnasta. (Danchilla 2012, xxi.)

WebGL otti kehityksessään ensiaskeleet vuonna 2006, kun Vladimar Vukićević työskenteli Canvas 3D -prototyypin kanssa. Prototyyppi käytti OpenGL 2.0 rajapintaa verkkosovelluksissa. Vuonna 2006, Khronos Group muodosti työryhmän luomaan WebGL rajapintaa, ja vuonna 2011, WebGL rajapinnan version 1.0 määritelmä oli valmis. (Danchilla 2012, xxi.)

3.2 WebGL-rajapinnalla piirtäminen

WebGL-rajapinnalla piirrettäessä, tulee ennen piirtoa suorittaa muutamia toimenpiteitä, ennen kuin rajapinnalla voidaan piirtää kuvapuskuriin. Ensin tarvitaan tapa kommunikoida WebGL-rajapinnan kanssa, mikä onnistuu canvas-elementin ja siitä saatavan WebGLRenderingContext-olion avulla. (Khronos Group 2015b.)

Tämän jälkeen voidaan muokata piirtoon vaikuttavia asetuksia, kuten millä värillä piirtopuskurin pikselit alustetaan uutta kuvaa piirrettäessä, tai mitä testejä fragmenteille halutaan suorittaa ennen kuin ne piirretään kuvapuskuriin.

WebGL-rajapinnan piirtolinjaston käyttöön tulee määritellä käytettävät verteksi- ja fragmenttivarjostimet. Varjostimet luodaan ohjelmoimalla niille GLSL-kielellä kirjoitetut lähdekoodit, jotka määrittelevät varjostimien toiminnan. Luotuihin varjostimiin sijoitetut lähdekoodit käännetään näytönohjaimen käsittelemään muotoon, minkä jälkeen ne voidaan sijoittaa piirron yhteydessä käytössä oleviksi varjostimiksi.

Jotta malli voidaan piirtää WebGL-rajapinnalla, tulee luoda yksi tai useampi puskuri, jotka sisältävät verteksidatan. Verteksidataa käytetään kuvien piirtämisessä. 3D-mallin verteksit, sekä verteksien mahdolliset muut ominaisuudet, kuten tekstuurikoordinaatit ja normaalivektorit sijoitetaan verteksipuskureihin. Verteksipuskureita kutsutaan myös usein lyhenteellä VBO, joka tulee englanninkielisestä termistä “Vertex Buffer Object”. Toinen määriteltävä puskurityyppi on indeksipuskuri, joka määrittelee järjestyksen luotujen verteksipuskureiden sisältämän verteksidatan käsittelylle piirrettäviä geometrisia primitiivejä muodosta-

essa. Verteksien käyttöjärjestyksen määrittteleviä puskureita kutsutaan indeksipuskuriksi, tai lyhennetysti IBO, joka tulee englannin kielen termistä “Index Buffer Object”. (Cantor & Jones 2012, 23-24.)

Jotta puskureihin sijoitetut verteksitiedot voitaisiin siirtää piirtolinjastoon verteksivarjostimen käyttöön, tulee verteksivarjostimeen määritellä attribute-muuttujia. Luodut attribute-muuttujat osoitetaan käyttämään jonkin verteksipuskurin sisältämää tietoa. Verteksivarjostin käsittelee attribute-muuttujiin liitettyjä puskureita verteksi kerrallaan. (Cantor & Jones 2012, 25-26.)

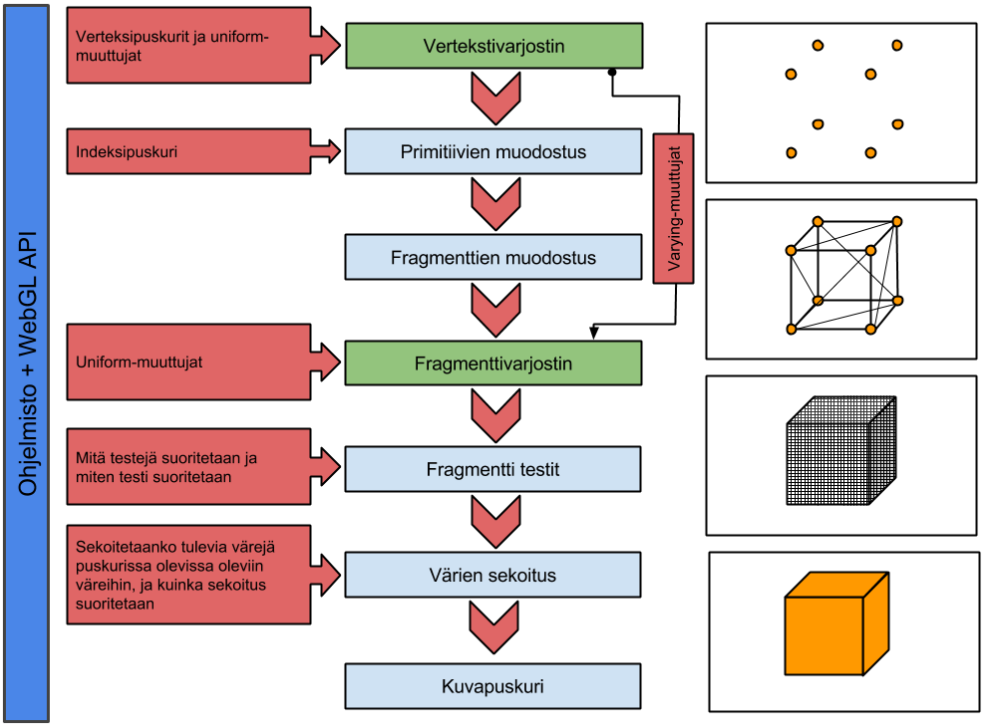
Ennen piirtoa voidaan myös määritellä molemmille varjostimille käytössä olevia uniform-muuttujia. Uniform-muuttujiin voidaan tallentaa mm. verteksidatalle suoritettavia transformaatioita matriiseina, valojen sijainteja vektoreina tai muuta tietoa, joilla vaikutetaan piirrettävään lopputulokseen muokkaamalla verteksejä ja fragmentteja varjostimissa. (Cantor & Jones 2012, 25-26.)

Kun verteksipuskurit ja mahdolliset muut tiedot on liitetty varjostimien attribute- ja uniform-muuttujiin, voidaan piirtofunktioita kutsua. Jos indeksipuskuri on määritelty, otetaan se käyttöön piirron yhteydessä, jolloin puskureiden verteksidataa käytetään piirrettävien primitiivien muodostuksessa indeksipuskurin määrittelemässä järjestyksessä. (Cantor & Jones 2012, 24-25.)

Kun piirtofunktiota on kutsuttu, käynnistetään prosessi, joka siirtää annetut verteksipuskurit kuvapuskuriin (engl. framebuffer). Kuvapuskuri on kaksiulotteinen taulukko joka sisältää fragmentit, jotka muodostuvat prosessin käsittelemästä verteksidatasta. (Cantor & Jones 2012, 25.) Tätä prosessia voidaan kutsua grafiikkalinjastoksi (engl. graphics pipeline) tai piirtolinjastoksi (Danchilla 2012, 33).

Kuvassa 8 nähdään yksinkertaistettu kuvaus piirtolinjastosta ja sen eri vaiheista. Oikealta nähdään kuinka piirtolinjaston eri vaiheisiin voidaan vaikuttaa. Oikealta nähdään kuinka piirtolinjaston eri vaiheisiin voidaan vaikuttaa WebGL-rajapinnan toiminnallisuuksien avulla ennen fragmenttien sijoittamista kuvapuskuriin. Varjostimiin voidaan vaikuttaa liittämällä verteksidataa sisältäviä verteksipuskureita verteksivarjostimeen, sekä molempiin varjostimiin voidaan liittää

uniform-muuttujia. Verteksivarjostin voi myös siirtää muuttujia fragmenttivarjostimelle varying-muuttujien avulla, jotka on määritelty verteksivarjostimen lähdekoodissa. Fragmenttivarjostimen kuvapuskurille lähettämiin fragmentteihin voidaan vaikuttaa ottamalla käyttöön eri testauksia, sekä määrittelemällä millaisia funktioita testaus käyttää. Testaus vertaa kuvapuskurissa olevien fragmenttien ja kuvapuskuriin tulevien fragmenttien arvoja keskenään, ja päättää, kirjoitetaanko tulevaa fragmenttia olemassa olevan fragmentin tilalle. Värien sekoituksella tarkoitetaan vaihetta, jossa kuvapuskurissa olevien fragmenttien värejä sekoitetaan tulevien fragmenttien värien kanssa. Tähän vaiheeseen voidaan vaikuttaa tähän ottamalla värien sekoituksen käyttöön, ja määrittelemällä, kuinka kuvapuskurissa olevan fragmentin väri ja sen tilalle tulevan fragmentin väri sekoittuvat keskenään.



Kuva 8. Vasemmalla kuvataan kuinka ohjelmoija voi vaikuttaa piirtolinjaston eri vaiheisiin ohjelmistosta ja oikealla on karkea kuvaus eri vaiheiden tuloksista muodostamista tuloksista.

3.2.1 Piirtokontekstin luominen ja piirtoasetukset

Jotta WebGL-rajapinnalle voitaisiin antaa käskyjä, täytyy käyttöön saada WebGLRenderingContext-olio, jonka avulla voidaan kutsua WebGL rajapinnan toiminnallisuuksia (Khronos Group 2015b).

Tämän konteksti-objekti saadaan luomalla ensin HTML5-kielen canvas-elementti, josta voidaan hakea piirtokonteksti canvas-elementin getContext-funktion avulla. Kun saatua kontekstia käytetään piirtämiseen, piirretään kuva siihen canvas-elementtiin, jolla konteksti luotiin.

```
var canvas = document.getElementById(“canvas-id”);
var gl = canvas.getContext(“webgl”);
```

Jos selain ja alusta tukevat WebGL-rajapintaa, kontekstin luominen onnistuu ja getContext-funktion paluuarvona saadaan WebGLRenderingContext-olio, jolla päästään käsiksi WebGL-rajapinnan funktioihin ja muuttujiin. Alla oleva esimerkkikoodi asettaa RGB-värin, jolla kuvapuskurin väripuskurin (engl. color buffer) arvot alustetaan rajapinnan clear-funktion kutsun yhteydessä.

```
// Params (Red, Green, Blue)
gl.clearColor(1.0, 0.0, 0.0);
gl.clear(gl.COLOR_BUFFER_BIT);
```

Usein WebGL-rajapinnalla piirrettäessä otetaan käyttöön myös fragmenttitestausvaiheen syvyystestaus (engl. depth testing). Syvyystestauksella voidaan varmistaa, ettei sellaisia primitiivejä tai kappaleita näy lopullisessa kuvassa, joiden tulisi jäädä toisten kappaleiden taakse. Syvyystestaus otetaan käyttöön rajapinnan enable-funktion avulla, jolle annetaan parametrina syvyystestaukseen viittaava rajapinnan muuttuja. Kun syvyystestaus on käytössä, puhdistetaan myös kuvapuskurin syvyyspuskurin (engl. depth buffer) sisältämät syvyysarvot kuvapuskurin väriarvojen lisäksi käytettäessä rajapinnan clear-funktiota. (Danchilla 2012, 30–31.) Alla olevassa esimerkissä syvyystestaus

otetaan käyttöön enable-funktiolla, ja kuvapuskurin väri- ja syvyysarvot puhdistetaan clear-funktion avulla.

```
gl.enable(gl.DEPTH_TEST);
// clear both color buffer and depth buffer of the framebuffer
gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
```

3.2.2 Varjostimet

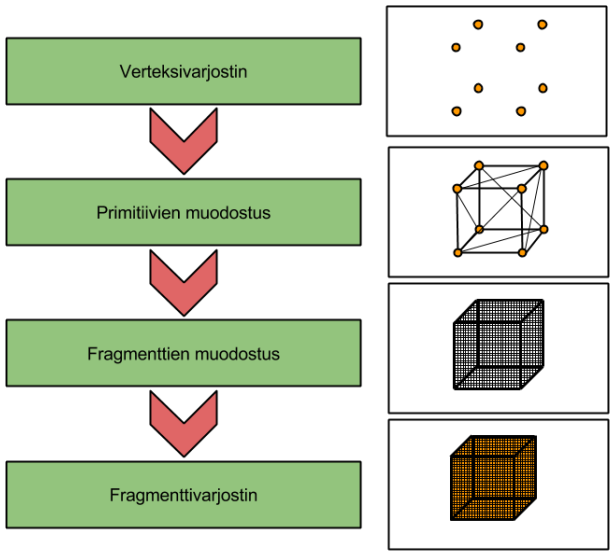
WebGL-rajapinta käyttää kahta erityyppistä, täysin ohjelmoitavaa varjostinta, joilla voidaan vaikuttaa merkittävästi piirtolinjaston tuottamaan kuvaan. Nämä varjostimet ovat verteksivarjostin ja fragmenttivarjostin.

Verteksivarjostin on vastuussa yksittäisten verteksien ja niihin liitettyjen muiden ominaisuuksien käsittelystä. Se suorittaa vertekseille sijaintikoordinaattien muutoksia, ja asettaa vähintään jokaisen verteksipuskurilla sille käsiteltäväksi annetun verteksin lopullisen sijainnin. Vaihtoehtoisesti, verteksivarjostin voi muokata myös muita verteksien ominaisuuksia, kuten verteksien värejä, normaalivektoreita tai tekstuuri koordinaatteja, jos ohjelmoija on sellaisia määritellyt. Verteksivarjostin voi myös suorittaa laskutoimituksia ja lähettää arvoja fragmenttivarjostimelle. (Danchilla 2012, 36.)

Fragmenttivarjostin käsittelee yksittäisiä fragmentteja, jotka muodostetaan verteksivarjostimen käsittelemien verteksien avulla muodostetuista primitiiveistä. Se määrittelee vähintään kuvapuskuriin piirrettävien fragmenttien värin, mutta se voi myös suorittaa monia eri toiminnallisuuksia tämän värin selvittämiseen, kuten värien poimimisen tekstuureista tekstuurikoordinaattien avulla, tai laskemalla valaistuksen vaikutuksen lopulliseen värityksen. Vaihtoehtoisesti fragmenttivarjostin kykenee myös hävittämään fragmentteja. (Danchilla 2012, 37.)

Kuvassa 9 on karkea kuvaus siitä, miten piirtolinjastossa verteksidata muodostuu fragmenteiksi. Verteksivarjostin asettaa verteksien sijainnin ja voi siirtää verteksien muita ominaisuuksia käytettäväksi fragmenttivarjostimelle, kuten verteksin värin, joka kuvan esimerkissä on oranssi. Kun verteksivarjostin on sijoittanut

kaikki verteksit lopullisiin sijainteihinsa muodostettavaan kuvaan, muodostetaan verteksien väleille kolmioprimitiivejä, jotka muodostavat laatikkogeometrian. Näiden muodostettujen primitiivien välille muodostetaan fragmentteja, jotka ovat pikseleiden kaltaisia pieniä palasia, jotka voidaan kirjoittaa lopulliseen kuvapuskuriin. Fragmenttivarjostin asettaa fragmenteille värityksen, minkä jälkeen fragmentit siirtyvät mahdollisiin käytössä oleviin fragmenttitestauksiin, kuten syvyystestaukseen.



Kuva 9. Fragmenttien muodostaminen verteksivarjostimesta fragmenttivarjostimelle.

WebGL-rajapinnan varjostimet ovat ohjelmoitavissa GLSL-kielellä, joka on lyhenne englanninkielisestä nimestä “OpenGL Shading Language”. WebGL-rajapinnan käyttämä versio GLSL-kielestä on GLSL-kielen versioon 1.20 perustuva GLSL ES-kieli, joka on OpenGL ES 2.0 rajapinnalle valmistettu varjostinkieli, joka sen kautta periytyi WebGL-rajapinnan käyttöön. (Danchilla 2012, 35–36.)

GLSL-kieli pohjautuu C++-kieleen. Vaikka verteksivarjostin ja fragmenttivarjostin muodostetaan kahdesta eri lähdekoodista, linkittyvät ne toisiinsa muodostaen yhden ohjelman, jonka näytönohjain ajaa. Verteksivarjostin käsittelee verteksin kerrallaan, ja yhdellä verteksillä voi olla monia eri ominaisuuksia sijainnin lisäksi. Fragmenttivarjostin käsittelee primitiiveistä muodostettuja fragmentteja,

ja se interpoloi verteksivarjostimelta lähetettyä verteksidataa. (Danchilla 2012, 36.)

Varjostimien luominen ja käyttöönotto suoritetaan muutamassa eri vaiheessa. Ensin luodaan tyhjä varjostin, joka voidaan asettaa olemaan tyypiltään verteksi- tai fragmenttivarjostimeksi. Luotuun varjostimeen sijoitetaan tekstimuotoinen lähdekoodi, joka määrittää kuinka varjostin käsittelee verteksejä tai fragmentteja piirrettäessä. Tämän jälkeen varjostimeen sijoitettu lähdekoodi käännetään näytönohjaimen käyttämään muotoon. Kun varjostin on luotu ja sillä on lähdekoodi, luodaan varjostinohjelma, johon luotu varjostin tullaan kiinnittämään ja linkittämään. Varjostinohjelma asettaa siihen asetut varjostimet piirtolinjastossa käytettäviksi verteksi- ja fragmenttivarjostimiksi. (Danchilla 2012, 38.) Luotua varjostinohjelmaa käytetään monesti parametrina funktioissa silloin, kun siihen kiinnitettyjen ja linkitettyjen varjostimien attribute- ja uniform-muuttujien sijainteja haetaan, jotta niihin voitaisiin sijoittaa piirtämisessä käytettävää tietoa, kuten verteksejä ja transformaatiomatriiseja.

Alla olevassa esimerkkikoodissa luodaan verteksi- ja fragmenttivarjostimet, joihin lisätään käytettävät lähdekoodit. Tämän jälkeen luodut varjostimet lisätään varjostinohjelmaan, joka sijoittaa ne piirron yhteydessä käytettäviksi varjostimiksi piirtolinjastossa.

```
// Create both vertex and fragment shaders
var vertexShader = gl.createProgram(gl.VERTEX_SHADER);
var fragmentShader = gl.createProgram(gl.FRAGMENT_SHADER);

// Attach shader source codes to shaders
gl.shaderSource(vertexShader, vertexSourceCode);
gl.shaderSource(vertexShader, fragmentSourceCode);

// Create program to which the shaders
// will be attached and linked to
var shaderProgram = gl.createProgram();
```

```
// Attach shaders to created program
gl.attachShader(shaderProgram, vertexShader);
gl.attachShader(shaderProgram, fragmentShader);

// Link the program
gl.linkProgram(shaderProgram);

// Set the program with the shaders attached
// to it to be used with drawing
gl.useProgram(shaderProgram);
```

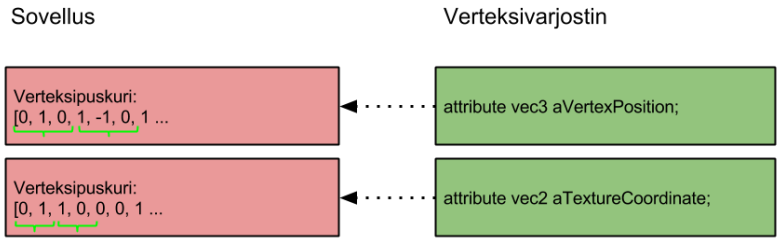
Verteksivarjostimen lähdekoodin tulee vähintään asettaa verteksin lopullinen sijainti piirrettävälle ruudulle. Verteksin sijaintiin voidaan vaikuttaa transformaatiomatriiseilla, jotka muokkaavat verteksien sijainnin mallin paikallisesta avaruudesta kamera-avaruuteen ja lisäksi verteksien muuttaminen kamera-avaruudesta projektiomatriisilla lopulliseen näyttöavaruuteen. Alla oleva koodi on esimerkki verteksivarjostimesta, joka ottaa vastaan verteksit attribute-muuttujaan verteksipuskurilta, ja sijoittaa verteksit yksi kerrallaan lopulliseksi verteksin sijainniksi ruudulla.

```
attribute vec3 aVertexPosition;

void(main) void{
      gl_Position = vec4(aVertexPosition, 1.0);
}
```

Attribute-muuttuja aVertexPosition osoittaa verteksipuskuriin, joka sisältää ohjelmoijan määrittelemät verteksit, jotka halutaan piirtää. Verteksivarjostin käy jokaisen puskurissa olevan verteksin läpi, ja sijoittaa sen gl_Position-muuttujaan. Muuttuja gl_Position on yksi verteksivarjostimen ulos lähtevistä arvoista, ja se määrittää verteksin lopullisen sijainnin piirrettävässä kuvassa (OpenGL.org 2015).

Verteksivarjostimen käyttämä attribute-muuttuja on varjostimelle sisään tuleva muuttuja, joka saadaan WebGL-rajapintaa hyödyntävältä sovellukselta. Attribute-muuttuja osoittaa verteksipuskuriin, joka on määritelty sovelluksessa. Koska verteksivarjostin ajetaan jokaiselle verteksille erikseen, muuttuu attribute-muuttujan arvo joka kerta käsittelemään seuraavan verteksin puskurista. (Cantor & Jones 2012, 26.)

Alla olevan kuvan (kuva 10) mukaisessa tapauksessa, verteksivarjostimessa on kaksi attribute-muuttujaa, jotka osoittavat molemmat eri verteksipuskureihin. Toinen puskureista sisältää verteksin sijaintia esittävää tietoa, ja toinen verteksin tekstuurikoordinaatteja. Koska sijainti tarvitsee kolme komponenttia ilmaisemaan sijainnin x, y ja z koordinaateilla, muodostuu yksi käsiteltävä verteksi kolmesta puskurin peräkkäisestä alkiosta. Tekstuurikoordinaatit muodostuvat kahdesta eri arvosta, jotka kertovat mihin pisteeseen verteksi osoittaa tekstuurina käytettävässä kuvassa, jolloin tekstuurikoordinaatteja varten luotu attribute-muuttuja osoittaa kahteen puskurin alkioon kerrallaan.



Kuva 10. Attribute-muuttujat osoittavat niille tarkoitettuihin verteksipuskureihin, josta ne poimivat verteksidatan verteksi kerrallaan.

Attribute-muuttujien lisäksi sovellus voi lähettää molemmille varjostimille käytettäväksi uniform-muuttujia. Nämä muuttujat ovat vakioita koko piirron suorituksen ajan. (Cantor & Jones 2012, 24) Uniform-muuttujiin voidaan lähettää monentyyppistä dataa käytettäväksi piirtoa varten, kuten valojen tietoja tai transformaatio matriiseja. Alla olevassa esimerkissä verteksivarjostimen verteksien lopullista sijaintia ruudulla muokataan transformaatiomatriisissa määritellyillä transformaatioilla. GLSL-kielessä samassa avaruudessa olevilla matriisimuuttujilla voidaan kertoa samassa avaruudessa olevia vektorimuuttujia, ja tulos on samassa avaruudessa oleva vektori, jota matriisi on muuttanut. Tämän vuoksi,

koska puskuriin verteksit on määritelty 3D-vektoreina, muutetaan vektori 4D-vektoriksi, minkä jälkeen se voidaan kertoa 4 x 4 matriisimuuttujan kanssa. Muuntaminen vec3-tyypistä vec4-tyyppiin on välttämätöntä myös siksi, että gl_Position-muuttuja ottaa vastaan vain vec4-tyyppisiä muuttujia.

```
attribute vec3 aVertexPosition;
uniform mat4 uTransformationMatrix;

void(main) void{
  gl_Position = uTransformationMatrix * vec4(aVertexPosition, 1.0);
}
```

Uniform-muuttujien käyttämän tiedon asettaminen varjostimiin tapahtuu kaksi-osaisessa vaiheessa. Ensin uniform-muuttujan sijainti täytyy noutaa rajapinnan getUniformLocation-funktiolla, joka ottaa parametreina käytössä olevan var-josinohjelman, johon varjostin on kiinnitetty, sekä uniform-muuttujan nimen sa-massa muodossa kuin se on varjostimessa määritelty. (Danchilla 2012, 7.) Alla olevassa koodissa haetaan yllä esitetyn verteksivarjostimen määrittelemä mat4-tyyppisen uniform-muuttujan sijainti.

```
var matrixUniform = gl.getUnifromLocation(shaderProgram,
                                          “uTransformationMatrix”);
```

Kun sijainti on noudettu, voidaan arvo sijoittaa uniform-muuttujalle käytettäväksi piirron ajaksi. Koska uniform-muuttujat voivat olla tyypiltään muun muassa erisuuruisia matriiseja, vektoreita tai tavallisia liukulukuja tai kokonaislukuja, on WebGL-rajapinnassa määritelty monta eri funktiota, jotka voivat sijoittaa arvon erityyppisiin uniform-muuttujiin. Esimerkiksi transformaatiomatriisia esittävän mat4-tyyppisen uniform-muutujan arvon asettaminen tapahtuisi alla olevan esi-merkin mukaisesti taulukkomuuttujalla ja rajapinnan uniformMatrix4fv-funktiolla. Koodissa uniformMatrix4fv-funktio ottaa parametreina noudetun uniform-muuttujan sijainnin, tiedon siitä käännetäänkö matriisi siirron yhteydessä, sekä lopuksi matriisin dataa esittävän Float32Array-tyyppisen taulukon, jossa 4 x 4 matriisin tapauksessa on 16 alkiota (Microsoft 2015a).

```
var matrixData = new Float32Array([
      1.0, 0.0, 0.0, 0.0,
      0.0, 1.0, 0.0, 0.0,
      0.0, 0.0, 1.0, 0.0,
      0.0, 0.0, 0.0, 1.0,
]);
gl.uniformMatrix4fv(matrixUniform, false, matrixData);
```

Taulukossa 1 esitetään, kuinka funktion nimi muuttuu eri matriisityypeillä. Funktiot toimivat muuten samoin, mutta annettavan matriisia esittävän taulukon kokoa on muutettava tyypin mukaisesti.

Taulukko 1. Eri matriisityyppisten uniform-muuttujien alustusfunktiot matriisi tyypeittäin.

Uniform-muuttujan tyyppi ja matriisin koko	Funktion nimi
uniform mat4 / 4 x 4 matriisi	gl.uniformMatrix4fv()
uniform mat3 / 3 x 3 matriisi	gl.uniformMatrix3fv()
uniform mat2 / 2 x 2 matriisi	gl.uniformMatrix2fv()

Fragmenttivarjostin käsittelee yksitellen kaikki fragmentit, jotka on muodostettu verteksivarjostimen ulosantamista vertekseistä. Se on vastuussa muodostetun kuvan fragmenttien, eli pikseleiden värien asettamisesta. (Danchilla 2012, 35–36.)

Fragmenttien värit määritellään sijoittamalla fragmenttivarjostimen sisäänrakennetun ulos lähtevään muuttujaan gl_FragColor nelikomponenttisen vec4 tyyppisen arvon. Arvo ilmaisee fragmentin lopullisen värin RGBA-muodossa (Red, Green, Blue, Alpha). (Wikibooks 2015.) Värikanaville annetut liukulukuarvot voivat vaihdella jokaiselle värikanavalle lukujen 0-1 välillä.

Alla olevassa esimerkissä on luotu fragmenttivarjostin, joka asettaa kaikki fragmentit punaisiksi. Alpha kanavan arvo on yksi, mikä merkitsee, että värissä ei ole läpinäkyvyyttä.

```
precision mediump float;
void main(void){
      gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
}
```

Usein kuitenkin värin muodostamiseen käytettävä tieto tulee sovelluksesta. Yksi tapa määritellä piirrettävien fragmenttien väri olisi antaa väriarvo uniform-muuttujana sovelluksesta, jolloin samaa uniform-arvoa käytettäisiin kaikille fragmenteille.

```
precision mediump float;
uniform vec3 uColor;
void main(void){
      gl_FragColor = vec4(uColor, 1.0);
}
```

Vec3-tyyppisen uniform-muuttujan arvo voidaan asettaa WebGL-rajapinnan uniform3fv- funktiolla. Funktio ottaa parametreina unifrom-muuttujan sijainnin, ja vektorin dataa esittävän Float32Array-tyyppisen taulukon, joka sisältää kolme alkiota, eli yhden arvon vektorin jokaiselle komponentille. (Microsoft 2015b.) Alla esimerkki funktion käytöstä vec3-tyyppisen uniform-muuttujan arvon alustamiseen.

```
var colorUniform = gl.getUnifromLocation(shaderProgram, “uColor”);
var vectorArray = new Float32Array([0.0, 1.0, 0.0]);
gl.uniform3fv(colorUniform, vectorArray);
```

Taulukossa 2 esitetään, kuinka funktion nimi muuttuu eri vektorityypeillä. Funktiot toimivat muuten samoin, mutta annettavan vektoria esittävän taulukon kokoa on muutettava tyypin mukaisesti.

Taulukko 2. Eri vektorityyppisten uniform-muuttujien alustusfunktiot vektori tyypeittäin.

Uniform-muuttujan tyyppi ja vektorin koko	Funktion nimi
uniform vec4 / 4D-vektori	gl.uniform4fv()
uniform vec3 / 3D-vektori	gl.uniform3fv()
uniform vec2 / 2D-vektori	gl.uniform2fv()

Jos väriarvojen tulee olla erilaisia, riippuen fragmenttien sijainnista primitiivien vertekseihin nähden, täytyy apuna käyttää vertekseille määriteltyjä ominaisuuksia, kuten värejä, normaalivektoreita tai tekstuurikoordinaatteja. Tämä tapahtuu käyttämällä varying-muuttujaa, jolla voidaan siirtää verteksivarjostimen käsittelemää tietoa suoraan verteksivarjostimelta fragmenttivarjostimelle. (Danchilla 2012, 21.)

Varying-muuttujaa käytetään määrittelemällä samanniminen ja tyyppinen varying-muuttuja sekä verteksivarjostimelle että fragmenttivarjostimelle. Varying-muuttuja on verteksivarjostimelta ulos lähtevä muuttuja, johon sijoitettu arvo siirtyy fragmenttivarjostimelle määriteltyyn varying-muuttujaan, joka toimii lähetetyn arvon vastaanottajana. (Danchilla 2012, 21.)

Alla olevassa esimerkkikoodissa on määritelty verteksivarjostin, joka verteksin lisäksi vastaanottaa verteksille määritetyn väriominaisuuden sille määritettyyn attribute-muuttujaan. Verteksivarjostin määrittelee varying-muuttujan, johon vastaanotettu arvo sijoitetaan main-funktiossa.

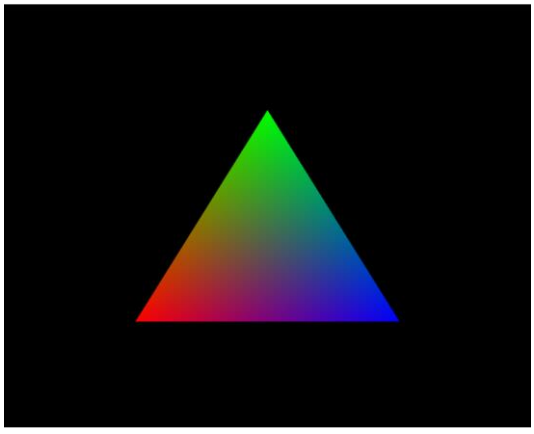
```
attribute vec3 aVertexPosition;
attribute vec3 aVertexColor;

varying vec3 vVertexColor;
void main(void){
      gl_Position = vec4(aVertexPosition, 1.0);
```

```
        vVertexColor = aVertexColor;
}
```

Seuraavassa koodissa määritellään fragmenttivarjostin, joka määrittelee verteksivarjostimen varying-muuttujaa vastaavan muuttujan, joka vastaanottaa verteksivarjostimelta tulevan verteksin väritiedon. Tämä arvo sijoitetaan fragmentin lopulliseksi väriksi.

```
precision mediump float;

varying vec3 vVertexColor;
void main(void){
        gl_FragColor = vec4(vVertexColor, 1.0);
}
```

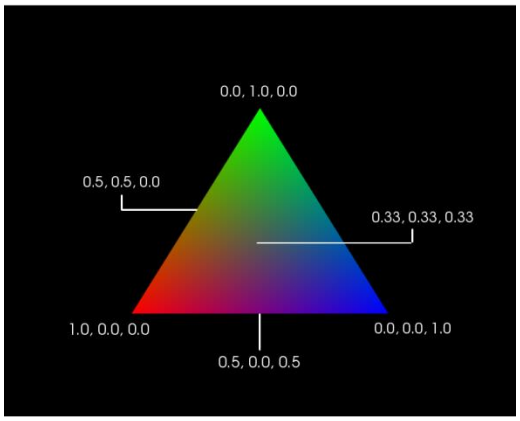
Kuvassa 11 on kuvakaappaus siitä, miten määritellyn kolmioprimitiivin värit muodostuvat, kun jokaisen kolmion verteksin väriominaisuus on asetettu eriväriseksi (punainen, vihreä ja sininen).



Kuva 11. Kuvakaappaus piirretystä kolmioprimitiivistä, jonka verteksien väriominaisuudet on siirretty varying-muuttujan avulla.

Kun tietoa lähetetään verteksivarjostimelta fragmenttivarjostimelle, jokaiselle fragmentille sijoitettu arvo on interpoloitu niiden verteksien arvoista, jotka muodostavat primitiivin. Yllä olevan kuvan (kuva 11) mukaisessa tilanteessa, lähete-

tyt väriarvot muuttuvat jokaiselle fragmentille sen perusteella, mikä on niiden sijainti kuhunkin primitiivin muodostamiseen käytettyyn verteksiin nähden. Mitä lähempänä tiettyä verteksiä fragmentti on, sitä lähempänä sen verteksin arvoja fragmentin arvot ovat. Kuvassa 12 kolmion kulmiin on merkitty kunkin verteksin väriarvo. Tämän lisäksi kuvaan on osoitettu muutamia väriarvoja eri sijainneista muodostetun kolmioprimitiivin pinnalta, jotka havainnollistavat fragmenttien saamien arvojen muutoksia interpoloinnin seurauksena.



Kuva 12. Interpoloinnin vaikutukset fragmenttien vastaanottamiin väriarvoihin eri sijainneissa.

Verteksien arvojen interpoloinnista fragmenteille on paljon hyötyä, kun fragmenttien värien määrittelemiseen käytetään tekstuureita, eli malli halutaan piirtää teksturoituna. Verteksit lähettäisivät fragmenttivarjostimelle tekstuurikoordinaatit, jotka interpoloituisivat jokaiselle fragmentille siten, että ne osoittavat juuri oikeaan pisteeseen tekstuurina käytettävässä kuvassa. Tekstuurikoordinaattien avulla koordinaattien osoittaman kuvan pikselin väriarvo voidaan poimia fragmentille asetettavaksi väriarvoksi.

3.2.3 Verteksipuskuri ja indeksipuskuri

Verteksipuskuri sisältää verteksidatan, josta piirrettävä geometria muodostetaan. Verteksit määrittävät 3D mallien sijainnit, eli mallin pintojen kulmat. Jokainen verteksi koostuu kolmesta liukuluvusta, jotka vastaavat verteksin x-, y- ja z-koordinaatteja avaruudessa. WebGL-rajapinnassa kaikki piirrettävän kappaleen

verteksit tulee määritellä JavaScript-kielen taulukko muuttujaan, josta verteksidata siirretään WebGLBuffer-olioon. (Cantor & Jones 2012, 24.)

Alla olevassa esimerkissä määritellään taulukko kolmio geometrian vertekseille, jotka sijoitetaan verteksipuskuriin.

```
var vertices = [
       -0.5, -0.5, 0.0,
       0.0, 0.5, 0.0,
       0.5, -0.5, 0.0
];
var vertexBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices),
                               gl.STATIC_DRAW);
```

Esimerkissä vertices-taulukko sisältää verteksit, jotka esittävät kolmion kulmien sijaintia 3D-avaruudessa. Verteksidataa varten luodaan uusi puskuri, joka luomisen jälkeen asetetaan aktiiviseksi verteksipuskuriksi. Kun tämä puskuri on aktivoitu, voidaan sille lähettää taulukossa oleva data.

Puskureita voidaan luoda rajapinnan createBuffer-funktiolla, joka palauttaa tyhjän WebGLBuffer-olion. Luotu puskuri voidaan aktivoida käyttämällä rajapinnan bindBuffer-funkiota. Kun puskureita aktivoidaan ja käsitellään, on tärkeää, että tyyppi, johon aktivoinnissa ja käsittelyssä viitataan, on sama. Yllä olevassa esimerkissä puskuri aktivoidaan ARRAY_BUFFER-tyyppisenä. ARRAY_BUFFER-tyyppisiä puskureita käytetään silloin kun puskureihin halutaan sijoittaa verteksidataa, kuten verteksejä, tekstuurikoordinaatteja ja normaalivektoreita. (Cantor & Jones 2012, 28.) Kun luotu puskuri on asetettu aktiiviseksi verteksipuskuriksi, vodaan puskuria käsitellä. Yllä olevassa esimerkkikoodissa, kun data sijoitetaan puskuriin, puskurin tyyppinä viitataan ARRAY_BUFFER-tyyppiin, jolloin annetun taulukon data siirtyy sillä hetkellä aktiivisena olevaan verteksipuskuriin.

Indeksipuskuri sisältää indeksidatan, jolla ilmaistaan, missä järjestyksessä verteksejä käsitellään. Indeksit ovat kokonaislukuarvoja, jotka kertovat WebGL-rajapinnalle, kuinka piirrettäviä verteksejä tulee yhdistää keskenään, jotta niistä muodostuisi pinnan muodostavia geometrisia primitiivejä. (Cantor & Jones 2012, 24.)

Indeksidata sijoitetaan samaan tapaan puskuriin, kuin verteksidata, mutta indeksipuskuria käsitellään erityyppisenä puskurina kuin verteksipuskuria. Indeksipuskurin aktivointiin ja käsittelyyn käytettävä tyyppi on ELEMENT_ARRAY_BUFFER. (Cantor & Jones 2012, 28.)

Alla olevassa esimerkkikoodissa luodaan taulukko, joka sisältää indeksidatan, ja joka sijoitetaan luotuun puskuriin, joka on aktivoitu indeksipuskuriksi. Yhdistettynä aikaisemman esimerkkikoodin verteksipuskurin luomisen ja alustamisen kanssa, indeksipuskuri kertoisi piirron yhteydessä WebGL-rajapintaa muodostamaan kolmion verteksipuskurin datasta samassa järjestyksessä, kuin verteksit on puskuriin määritelty.

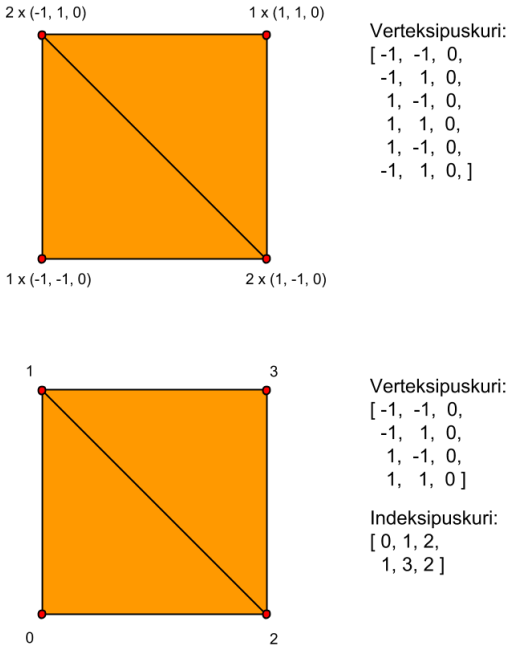
```
var indices = [ 0, 1, 2];
var indexBuffer = gl.createBuffer();
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, indexBuffer);
gl.bindData(gl.ELEMEMT_ARRAY_BUFFER, new Float32Array(indices),
                                        gl.STATIC_DRAW);
```

Kuten aikaisemmin mainittu, indeksipuskuri käyttää ELEMENT_ARRAY_BUFFER-tyyppiä. Tyyppiä lukuun ottamatta, indeksipuskurin aktivoiminen ja datan sijoittaminen toimii kuitenkin samalla tavalla kuin verteksipuskurilla.

Indeksien käyttäminen piirtämisessä tekee verteksien käsittelemisestä nopeampaa. Indeksien avulla tarvittavien verteksien määrää voidaan vähentää, mikä nopeuttaa verteksivarjostimen ajoa. Indeksejä käytetään primitiivien muodostuksessa yhdistämään samoja verteksejä useampaan eri primitiiviin, eli samaa

verteksiä voidaan käyttää useamman eri primitiivin muodostukseen. (Cantor & Jones 2012, 36.)

Kuvassa 13 on esimerkkinä neliön piirtäminen ilman indeksejä ja indeksien kanssa. Neliö muodostetaan kahdesta kolmioprimitiivistä. Ilman erikseen määriteltyjä indeksejä, neliön muodostamiseen tulisi määritellä kuusi verteksiä. Näistä kuudesta verteksistä kaksi ylänurkan verteksiä määrittelevät saman sijainnin, sekä kaksi alanurkan verteksiä määrittelevät saman sijainnin. Kun indeksipuskuri on käytössä, tarvitsee neliön muodostukseen määrittää vain neljä verteksiä, yhden jokaiselle nurkalle. Näihin vertekseihin viitataan indeksipuskurissa. Kuvan esimerkissä indeksipuskurissa viitataan vertekseihin 1 ja 2 kahdesti, koska ne ovat osa molempien kolmioiden muodostamista.



Kuva 13. Ylemmän neliön muodostaminen kahdesta neliöstä ilman erillistä indeksipuskuria vaatisi kuusi verteksiä, kun taas alemman neliön muodostaminen vaatii neljä, koska käytössä on indeksipuskuri.

Kun piirrettävien primitiivien määrä kasvaa, syntyisi samassa sijainnissa olevia verteksejä paljon enemmän, jolloin indeksipuskurin käyttö tuo selviä eroja suoritusnopeudessa.

Kun puskurit on muodostettu, ja niihin on sijoitettu haluttu data, voidaan puskureita käyttää piirtämisessä. Tämä tapahtuu kertomalla käytössä olevan verteksivarjostimen attribute-muuttujille, mihin puskuriin ne osoittavat. Jokainen attribute-muuttuja voi osoittaa vain yhteen verteksipuskuriin kerrallaan. (Cantor & Jones 2012, 31.)

Jotta attribute-muuttuja voidaan asettaa osoittamaan verteksipuskuriin, täytyy ensin hakea attribute-muuttujan sijainti käyttöön otetusta verteksivarjostimesta. WebGL-rajapinnan getAttribLocation-funktiota voidaan käyttää attribute-muuttujien sijainnin hakemiseen ohjelmasta, johon verteksivarjostin on liitetty. (Danchilla 2012, 7.) Alla olevassa koodiesimerkissä esitetään kuinka attribute-muuttujan sijainti haetaan. WebGL-rajapinnan getAttribLocation-funktiolle annetaan parametreina varjostinsovellus, johon verteksivarjostin on liitetty, ja attribute-muuttujan nimi siinä muodossa, kuin se on verteksivarjostimen lähdekoodissa kirjoitettu. Toiminnan periaate on siis sama kuin uniform-muuttujien sijainteja hakiessa getUniformLocation-funktion avulla.

```
var vertexAttribute = gl.getAttribLocation(shaderProgram,
                                           “aVertexPosition”);
```

Attribute-muuttujan osoittaminen tiettyyn verteksipuskuriin on kolmevaiheinen prosessi. Ensin, verteksipuskuri, johon verteksidataa on sijoitettu, aktivoidaan käytössä olevaksi ARRAY_BUFFER-tyyppiseksi puskuriksi. Tämän jälkeen attribute-muuttuja asetetaan ottamaan vastaan puskuriin muodostettua taulukko muotoista tietoa enableVertexAttribArray-funktion avulla. Kun haluttu verteksipuskuri on aktiivinen, ja attribuutti on valmisteltu vastaanottamaan puskuri, voidaan attribuutti osoittaa käyttämään aktiivisena olevaa puskuria WebGL-rajapinnan vertexAttribPointer-funktion avulla. (Danchilla 2012, 7.) Alla esitetään koodiesimerkki attribute-muuttujan osoittamisesta verteksipuskuriin.

```
gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
gl.enableVertexAttribArray(vertexAttribute);
gl.vertexAttribPointer(vertexAttribute, 3, gl.FLOAT, false, 0, 0);
```

Kun attribuutti osoitetaan haluttuun verteksipuskuriin vertexAttribPointer-funktion avulla, tulee parametreissa mainita, mihin attribuuttiin puskuri liitetään, kuinka monesta puskurin alkiosta yksi verteksi muodostuu kerrallaan (esim. sijaintia varten käytetään usein kolmea alkiota per verteksi, eli x, y ja z koordinaatit), ja mikä on puskurin arvojen tyyppi (float-liukuluku).

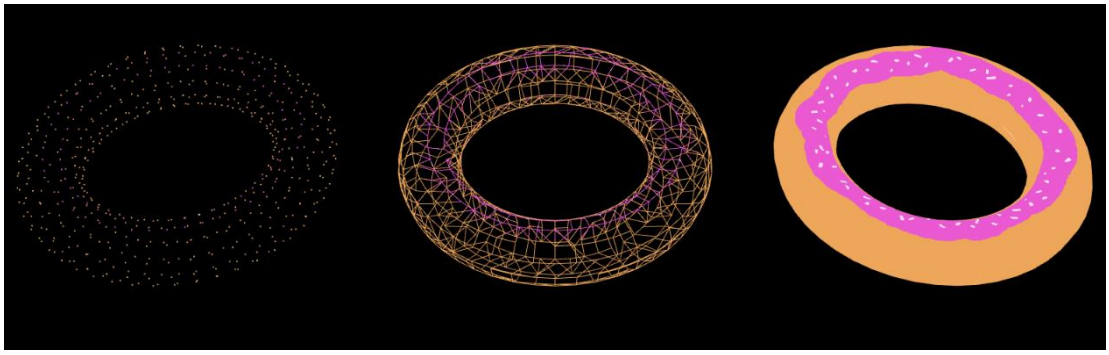
Kun kaikki verteksivarjostimeen määritellyt attribute-muuttujat on asetettu osoittamaan haluttuihin verteksipuskureihin, voidaan suorittaa puskureiden verteksidatan muuntaminen lopulliseksi kuvaksi piirtämällä puskurit. Jos käyttöön on määritelty myös indeksipuskuri, asetetaan se aktiiviseksi indeksipuskuriksi, ja kutsutaan sen jälkeen drawElements-piirtofunktiota. (Cantor & Jones 2012, 36–37.)

Alla olevassa koodissa esitetään, kuinka indeksipuskuria käytetään piirrossa piirrettävän kappaleen primitiivien muodostuksessa. Indeksipuskuri aktivoidaan käyttöön ELEMENT_ARRAY_BUFFER-tyyppisenä WebGL-rajapinnan bindBuffer-funktiolla. Kun indeksipuskuri on käytössä, kutsutaan WebGL-rajapinnan drawElements-funktiota, joka käyttää aktiivista indeksipuskuria piirrettävien primitiivien määrittämiseen. Rajapinnan drawElements-funktio ottaa parametreina primitiivien muodostustavan ilmaisevan arvon, piirrossa käytettävien indeksien määrän, ja indeksipuskurin läpikäyntiin määritetyn poikkeaman. Indeksipuskurin sisältämien indeksien määrä voidaan kertoa antamalla indeksipuskurin muodostamiseen käytetyn taulukon pituus. Poikkeama voidaan useimmissa tapauksissa asettaa arvolla 0, jolloin jokaista indeksipuskurin arvoa hyödynnetään primitiivien muodostamiseen.

```
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, indexBuffer);
gl.drawElements(gl.TRIANGLES, indices.length, 0);
```

Esimerkissä primitiivien muodostustavaksi on asetettu TRIANGLES, mikä muodostaa kolmioprimitiivejä jokaisesta kolmesta peräkkäisestä verteksistä, joihin indeksit viittaavat. Muodostustavaksi voidaan asettaa myös muita arvoja, kuten viivoja LINES-arvolla, tai pisteitä POINTS-arvolla. LINES-arvo muodostaa viivan jokaisen peräkkäin olevan verteksin väliin, ja POINTS-arvo piirtää jokaisen ver-

teksin yksittäisenä pisteenä lopulliseen kuvaan. (Cantor & Jones 2012, 43.) Alla kuvakaappaus (kuva 14) eri muodostustapojen käytöstä.



Kuva 14. Kuvakaappaus piirretystä donitsin näköisestä 3D-mallista, joka on piirretty käyttäen eri primitiivien muodostustapaa. Muodostustavat vasemmalta oikealle ovat POINTS, LINES ja TRIANGLES.

4 WEBGL_LIB-kirjaston luokat ja toiminnot

Opinnäytetyössä tuotettu WebGL-rajapintaa hyödyntävä JavaScript-kirjasto sisältää useita eri luokkia, joilla mahdollistetaan grafiikan piirtäminen kirjaston avulla. Renderer-luokka on vastuussa kommunikoinnista WebGL-rajapinnan kanssa, 3D-mallien lataamisesta COLLADA-tiedostoista, ladattujen mallien hallinnoinnista, tekstuurien lataamisesta, niiden luomisesta ja hallinnoimisesta, sekä kuvan piirtämisestä muita luokkia hyödyntäen.

BaseObject-luokka on vastuussa muiden luokkien käyttämien perustransformaatioiden toiminnallisuuksien määrittelystä. Luokka periyttää muille piirrossa käytettäville luokille perustransformaatiot, kuten rotaatiot, translaation ja skaalauksen.

MeshObejct-luokka on vastuussa piirrettävien kappaleiden määrittelystä käyttämällä Renderer-luokan lataamia malleja ja tekstuureita, sekä määrittää piirrettäville malleille suoritettavat transformaatiot hyödyntäen BaseObject-luokalta perittyjä toimintoja.

CameraObject-luokka on vastuussa maailmassa liikutettavan ja kierrettävän kameran määrittelystä, sekä projektion määrittelystä, millä piirrettävien kappaleiden verteksit muunnetaan lopulliseen sijaintiinsa piirrettävällä ruudulla. Se hyödyntää osaa MeshObject-luokan määrittelemistä transformaatioista, mutta vaatii joukon omia toimintoja kameran määrittämien transformaatioiden toteutukseen.

PointLightObject- ja SpotlightObject-luokkien avulla piirrettävään kuvaan voidaan määritellä piste- ja spottivaloja. PointLightObject- ja SpotlightObject-luokkien avulla voidaan vaikuttaa valojen vaikutusalueella olevien kappaleiden pinnan väritykseen luokkien määrittelemien valaistuksien avulla.

Seuraavissa kappaleissa käydään läpi yllä mainittujen luokkien määrittelemiä keskeisiä käsitteitä ja toimintoja läpi. Tarkempaa tietoa kirjaston jokaisen luokan sisältämistä jäsenmuuttujista ja funktioista löytyy liitteessä 1 esitetyissä taulukoista.

4.1 Renderer-luokka ja sen alustus

Renderer-luokka suorittaa alustuksensa aikana monia eri toiminnallisuuksia, jotka huolehtivat siitä, että luotu Renderer-olio on käyttökelpoinen piirtämistä varten alusta alkaen.

Renderer-luokan alustusfunktio luo canvas-elementin, josta se noutaa WebGLRenderingContext-olion, jota käytetään kaikissa WebGL-rajapintaa hyödyntävissä toiminnallisuuksissa. Kaikki Renderer-luokan maailmaan lisätyt MeshObject-, CameraObject- PointLightObject- ja SpotlightObject-luokan oliot tulevat käyttämään Renderer-luokan luomaa rajapintakontekstia silloin, kun niiden täytyy suorittaa toiminnallisuuksia WebGL-rajapinnan avulla. Konteksti luodaan rajapinnan createGLContext-metodilla, joka yrittää luoda rajapintakontekstin. Jos kontekstin luominen ei onnistu virheen tai yhteensopimattoman alustan vuoksi, ilmoitetaan siitä selaimen käyttäjälle.

Kun konteksti on käytössä, voidaan sitä alkaa hyödyntää WebGL-rajapinnan käyttämien piirtoasetuksien ja varjostimien alustamisessa. Oletuksellisesti käytettävät piirtoasetukset alustetaan kutsumalla setupRenderer-funktiota, joka asettaa kuvapuskureiden puhdistuksessa käytettävän taustavärin, ja ottaa fragmenttien syvyystestauksen käyttöön.

Piirrossa käytettävät varjostimet ja varjostinohjelma alustetaan Renderer-luokan luonnin yhteydessä kutsumalla luokan createShaders-metodia. Jos varjostimia ei ole asetettu Renderer-luokan shaders-objektiin ennen funktion ajamista, asettaa funktio oletusarvoisesti käytössä olevat varjostimet, jotka tukevat kaikkia kirjastolla käytettäviä toimintoja (transformaatiot, tekstuurit, valot). Renderer-luokan shaders-objekti sisältää tiedot varjostimen tyypille ja lähdekoodin, joita käytetään varjostimen luomisessa. Kun haluttu varjostin on luotu tyypin avulla ja varjostimen käyttämä lähdekoodi on asetettu, käännetään varjostimen lähdekoodi näytönohjaimen ymmärtämään muotoon, ja käännetty varjostin tallennetaan shaders-objektiin. Kun molemmat varjostimet on käännetty onnistuneesti, sijoitetaan varjostimet käytettäväksi varjostinohjelmaan, joka luodaan luokan program-jäsenmuuttujaan. Luokan program-jäsenmuuttujaan luotu varjostinohjelma linkitetään ja asetetaan käytössä olevaksi varjostinohjelmaksi.

Jos varjostimia haluaa muokata Renderer-olion luomisen jälkeen, tulee luokan shaders-objektin sisältämä lähdekoodi korvata halutusta varjostimesta, minkä jälkeen voidaan kutsua createShaders-metodia uudestaan, mikä asettaa uuden lähdekoodin varjostimen käyttöön. Tällöin on kuitenkin huomioitava, että annettu lähdekoodi sisältää vähintään samat attribute-, uniform- ja varying-muuttujat, jotka alustuksessa käytettävä lähdekoodi sisältä. Tämän jälkeen käyttäjä voi määritellä itse, kuinka näitä muuttujia käytetään sijoittamaan lopulliset verteksien sijainnit, ja fragmenttien värit.

Viimeinen tärkeä vaihe alustuksessa on luokan oletustekstuurin eli defaultTexture-jäsenmuuttujan käyttämän kuvan lataaminen muistiin. Tämä tapahtuu loadDefaultImage-metodilla. Metodi luo kutsuttaessaan uuden kuvan käyttäen kuvan lähteeksi asetettua Base64-muotoon kryptattua merkkijonoa, joka purettaessa muodostaa PNG-kuvan. Kun kuva on ladattu muistiin, tallennetaan se

Renderer-olion loadedImages-objektiin, josta sitä käytetään kun defaultTexture-jäsenmuuttuja alustetaan ensimmäisen piirron yhteydessä.

4.1.1 Maailma-objekti ja sen käyttö

Renderer-luokka määrittelee createWorld-metodin, jolla voidaan luoda uusia maailma-objekteja. Metodille annetaan parametrina luotavan maailma-objektin nimi, jolla se löytyy Renderer-olion worlds-objektista. Luokan worlds-objekti sisältää kaikki metodilla luodut maailma-objektit. Ensimmäinen metodilla luotu maailma-objekti asetetaan Renderer-luokan activeWorld-jäsenmuuttujan arvoksi, joka määrittää mistä maailma-objektista piirtoon käytettävät resurssit poimitaan render-metodin kutsun yhteydessä.

Maailma-objekti toimii piirtoon käytettävien resurssien ja asetusten kokoavana varastona, johon voidaan lisätä kaikki piirrettävät MeshObject-oliot. Piirrettävien olioiden valaistukseen voidaan vaikuttaa maailma-objektille asetetuilla globaalin valaistuksen asetuksilla, jotka määrittelevät maailman kaikille objekteille saman ambienttivalaistuksen ja suunnatun valaistuksen, jotka muodostuvat piirrettävien kappaleiden pinnoille. Maailmaan voidaan lisätä myös erillisiä PointLightObejct- ja SpotlightObject-olioita, jotka määrittelevät vain paikallisia valaistuksia. Paikalliset valaistukset vaikuttavat vain kappaleisiin niiden määrittelemillä vaikutusalueilla. Maailmalle määritetään myös vähintään yksi kamera CameraObject-olion avulla, mikä määrittää mistä maailman näkökulmasta piirrettävä kuva muodostetaan.

Maailma-objekti määrittelee useita metodeita, joilla siihen voidaan lisätä eri resursseja. MeshObject-olioita voidaan lisätä addMesh-metodilla, joka ottaa parametrina lisättävän MeshObject-olion. Metodilla lisätty MeshObject-olio on tämän jälkeen käytettävissä piirrossa.

CameraObject-luokan olioita voidaan lisätä maailma-objektiin addCamera-metodilla, joka ottaa parametrina lisättävän CameraObject-olion. Jos maailma-objekti ei sisällä aikaisemmin lisättyjä muita CameraObeject-olioita, asettaa metodi parametrina annetun CameraObject-olion maailma-olion käyttämäksi aktii-

viseksi kameraksi, jonka määrittelemiä transformaatioita ja projektiomatriisia käytetään piirrossa. Aktiivista CameraObject-oliota voidaan muuttaa setActive-Camera-metodilla, joka ottaa parametrina aktiiviseksi asetettavan CameraObject-olion.

Maailma-olioon voidaan lisätä PointLightObject- ja SpotlightObject-luokkien olioita addPointLight- ja addSpotlight-metodeilla, jotka ottavat parametrina lisättävän PointLightObject- tai SpotlightObject-olion. Tämän jälkeen olioiden määrittelemä paikallinen valaistus on käytettävissä piirrossa käytettävien MeshObeject-olioiden pinnan valaistuksen määrittelyssä, jos MeshObject-olio on valaistuksen määrittelemällä vaikutusalueella. Maailma-objekti mahdollistaa myös piirrossa käytettävän globaalin valaistuksen muuttamisen muokkaamalla objektin sisältämiä ambientLight-, directionalLightColor- ja directionalLightDir-muuttujien arvoja.

4.1.2 Tekstuurien muodostus

Renderer-luokka hallitsee kaikkien tekstuurien lataamista ja muodostamista. Kaikki käytettävät tekstuurit tallennetaan luokan loadedTextures-objektiin, josta niitä voidaan käyttää MeshObject-olioiden määrittelemien kappaleiden teksturoinnin luomiseen.

Jokaiselle loadedTextures-objektiin muodostetulle tekstuuri-objektille on määritetty tekstuurina käytettävä kuva, kuvasta muodostetun WebGLTexture-olio, sekä tekstuuri-objektin nimi, joka on sama kuin teksturi-objektin avain Renderer-olion loadedTextures-objektissa. Renderer-luokan loadImage-metodia käytetään muodostamaan tekstuuri-objekteja antamalla paramtereina kuvatiedostoon viittaava polku, ja tekstuuri-objektille asetettava nimi, jonka avulla tekstuuriobjektiin voidaan viitata kun sitä halutaan käsitellä. Metodi luo uuden Image-olion, johon parametrina annetun polun osoittama kuvatiedosto tallennetaan. Metodi määrittää Image-oliolle onload-funktion, jota kutsutaan kun kuvatiedosto on latautunut muistiin. Funktio luo metodia kutsuneen Renderer-olion loadedImages-objektiin uuden tekstuuri-objektin, johon ladattu kuva ja tekstuuri-objektille ase-

tettu nimi tallennetaan. Alla olevassa koodissa nähdään loadImage-metodin toiminta.

```
WEBGL_LIB.Renderer.prototype.loadImage = function(path, name){
    if(name && typeof(name) === "string"){
        var image = new Image();
        image.onload = (function(renderer, name){

            renderer.loadedTextures[name] = {
                image : image,
                texture : null,
                name : name
            };
        }(this, name));
        image.src = path;
    }else{
        console.error("renderer.loadImage: name string not provided");
    }
};
```

Kun tekstuurin käyttämä kuva on ladattu loadImage-metodilla, voidaan sitä käyttää luoman WebGLTexture-olio, jonka avulla ladattua kuvaa voidaan käyttää piirrettävän kappaleen tekstuurina WebGL-rajapinnalla piirrettäessä. WebGLTexture-olion luominen tapahtuu Renderer-luokan määrittelemän setupTexture-metodin avulla. Metodi ottaa parametrina loadedTextures-objektiin lisätyn tekstuuri-objektin, joka sisältää ladatun kuvan. Metodi tarkistaa, että kyseinen kuva on olemassa ja ladattu muistiin Image-olion complete-ominaisuuden avulla. Kun kuva on todettu käyttökelpoiseksi, luodaan parametrina annettuun tekstuuri-objektin texture-muuttujaan uusi WebGLTexture-olio WebGL-rajapinnan createTexture-funktion avulla, mikä palauttaa tyhjän WebGLTexture-olion. Palautettu olio asetetaan aktiiviseksi rajapinnan bindTexture-funktion avulla, minkä jälkeen siihen voidaan vaikuttaa rajapinnan eri funktioilla. Kun WebGLTexture-olio aktivoidaan, siirtyy se käsiteltäväksi sillä hetkellä aktiiviseen WebGL-rajapinnan tekstuuriyksikköön näytönohjaimelle. (Silicon Graphics 2006a.) Aktiivista teks-

tuuriyksikköä voidaan muuttaa rajapinnan activeTexture-funktion avulla, ja eri yksiköihin voidaan aktivoida eri tekstuureita. Tämä mahdollistaa usean eri tekstuurin käytön samanaikaisesti varjostimilla. (Silicon Graphics 2006b.)

WebGLTexture-olioille voidaan asettaa parametreja, jotka vaikuttavat olion käyttäytymiseen eri tilanteissa. Parametreja voidaan asettaa aktivoituun tekstuuriin käyttämällä WebGL-rajapinnan texParameteri- ja pixelStorei-funktioilla. Funktioilla voidaan määritellä useita eri asetuksia riippuen käytettävistä parametreista. Metodissa tekstuurille asetetaan vain muutamia perusasetuksia, kuten TEXTURE_WRAP_S- ja TEXTURE_WRAP_T-parametrit, jotka määrittävät kuinka tekstuuri käyttäytyy kun käytettävät tekstuurikoordinaatit ovat arvoalueen [0-1] ulkopuolella. CLAMP_TO_EDGE-asetus määrittää rajapintaa käyttämään asetetun tekstuurin reunimmaisten pikseleiden väriarvoja, jos tekstuurikoordinaatit menevät arvoalueen ulkopuolelle. Toiset metodissa määritetyt parametrit ovat TEXTURE_MIN_FILTER ja TEXTURE_MAG_FILTER. Parametrit vaikuttavat tekstuureista poimittujen arvojen suodattamiseen silloin kun pinnalle sijoitettu tekstuurin koko muuttuu pinnan ollessa kaukana tai pinnan koon muutoksen seurauksena. Metodi käyttää NEAREST-arvoa, joka poimii tekstuurista aina lähimmän arvon pikselin väriarvoksi. Tekstuurien suodattamiseen voidaan määritellä monia asetuksia, jotka parantavat paljon piirrettyjen kuvien laatua, mutta ne vaativat myös enemmän prosessointia. (Silicon Graphics 2006c.) Tulevaisuudessa kirjasto voisi tukea erilaisia teksturoinnin asetuksia, mutta tässä vaiheessa, asetukset ovat samat, eikä useamman tekstuurin samanaikaista käyttöä tueta.

Kun parametrit on asetettu, kutsutaan WebGL-rajapinnan texImage-funktiota, jolla aktiiviseen tekstuuriin voidaan lähettää tekstuurina käytettävä kuva. Kuvan lisäksi parametreissa määritetään mitä värikanavia kuvan arvoista käytetään RGBA-arvoilla, ja kuinka monta arvoa yhtä värikanavaa kohden on käytettävissä UNSIGNED_BYTE arvolla. Parametrina annettu nolla-arvo vaikuttaa kuvan yksityiskohtien tasoon, mikä voisi olla eriarvoinen, jos tekstuurin suodatuksen parametreja muutettaisiin tiettyihin asetuksiin. (Silicon Graphics 2006d.)

Kun aktiiviseen WebGLTexture-olioon on asetettu haluttu tekstuuri, voidaan tekstuuri aktivoida pois käytöstä kutsumalla rajapinnan bindTexture-funktiota null-arvolla. Tekstuuriksi asetettu kuva säilyy luodussa WebGLTexture-oliossa, ja se voidaan ottaa uudestaan käyttöön esimerkiksi piirron yhteydessä aktivoimalla se uudestaan bindTexture-funktiolla. Tekstuurin käyttöönottava setupTexture-metodi on esitetty kokonaisuudessaan seuraavassa koodissa.

```
WEBGL_LIB.Renderer.prototype.setupTexture = function(textureObj){
  if(textureObj){
    if(textureObj.image){
      if(textureObj.image.complete){
        var gl = this.gl;
        textureObj.texture = gl.createTexture();
        gl.activeTexture(gl.TEXTURE0);
        gl.bindTexture(gl.TEXTURE_2D, textureObj.texture);
        gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, true);
        gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S,
                                        gl.CLAMP_TO_EDGE);
        gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T,
                                        gl.CLAMP_TO_EDGE);
        gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER,
                                        gl.NEAREST);
        gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER,
                                        gl.NEAREST);
        gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA,
                                    gl.UNSIGNED_BYTE, textureObj.image);
        gl.bindTexture(gl.TEXTURE_2D, null);
      }
    }
  }
};
```

Kun tekstuureja käytetään piirrossa Renderer-luokan render-metodin kutsun yhteydessä, aktivoidaan MeshObject-olioon kiinnitytetyn tekstuuri-objektin sisäl-

tämä WebGLTexture-olio uudestaan, jolloin tekstuuri siirtyy aktiiviseen tekstuuriyksikköön näytönohjaimelle. Tähän tekstuuriyksikköön viitataan numerolla, joka lähetetään varjostimessa määritetylle sampler2D-tyyppiselle uniform-muuttujalle. Muuttuja kykenee viittaamaan WebGL-rajapinnan tekstuuriyksiköihin, joihin WebGLTexture-olioita on aktivoitu. Renderer-luokan render-metodi kutsuu seuraavaa koodia jokaiselle MeshObject-oliolle, jotka piirrossa käytetään, ja joille tekstuuri on asetettu käytettäväksi.

```
var texture = meshObject.textureObject.texture;
gl.activeTexture(gl.TEXTURE0);
gl.bindTexture(gl.TEXTURE_2D, texture);
var textureSamplerUniform = gl.getUniformLocation(this.program,
                                                  "uTextureSampler0");
gl.uniform1i(textureSamplerUniform, 0);
```

Kutsu asettaisi alla olevan varjostinkoodin mukaisen sampler2D-tyyppisen uniform-muuttujan osoittamaan TEXTURE0-tekstuuriyksikköön, joka sisältäisi aktivoidun WebGLTexture-olion määrittelemän tekstuurin. Tekstuurista voidaan tämän jälkeen poimia väriarvoja GLSL-kielen texture2D-funktion avulla. Funktio ottaa parametrina tekstuuriyksikköön osoittavan uniform-muuttujan, sekä kappaleelle määritetyt tekstuurikoordinaatit, joilla väri poimitaan tekstuurikuvasta koordinaattien määrittämästä sijainnista. Kutsuttu texture2D-funktio palauttaa vec4-tyyppisen muuttujan, joka sisältää tekstuurista poimitun RGBA-muotoisen väriarvon. Saatua arvoa voidaan hyödyntää monella tavalla, kuten esimerkiksi sijoittamalla se fragmenttivarjostimessa fragmentin lopulliseksi väriksi alla olevan koodin mukaisesti. Tekstuureja voidaan käyttää myös esimerkiksi korkeuskarttojen, varjokarttojen tai normaalikarttojen määrittelyyn, mutta luodussa kirjastossa tuetaan vain yhden tekstuurin lähettämistä kappaleen pintojen värien määrittämiseen.

```
uniform sampler2D uTextureSampler0;
...
vec4 surfaceColor = texture2D(uTextureSampler0, vec2(vTextureCoord.s,
                                                     vTextureCoord.t));
```

```
...
gl_FragColor = surfaceColor;
```

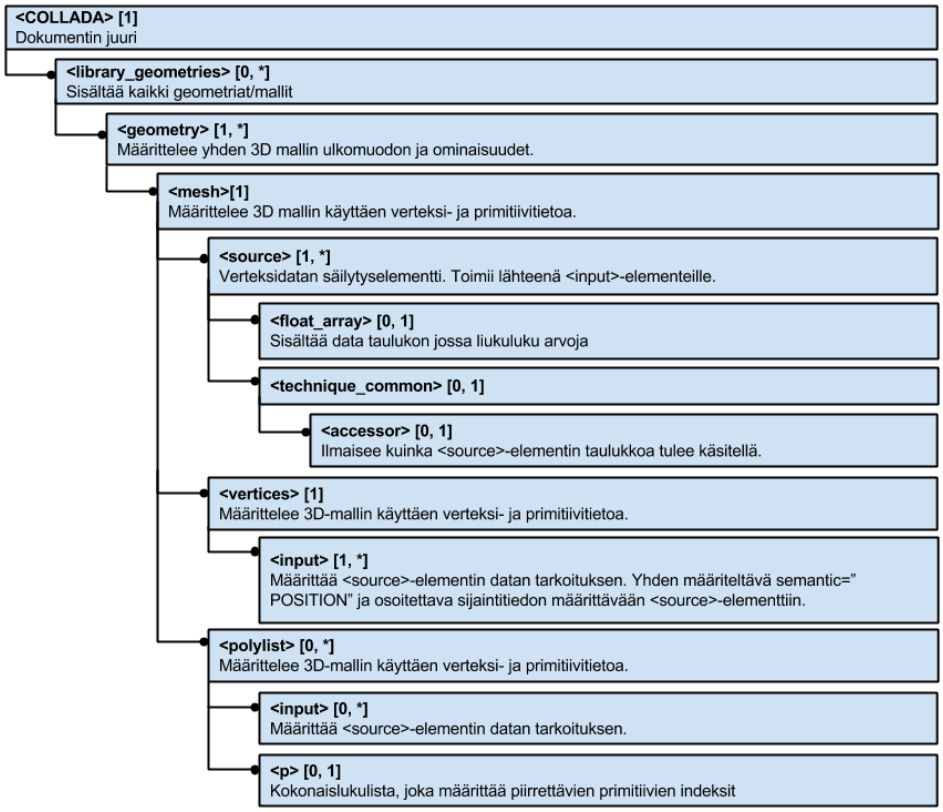
4.1.3 COLLADA ja COLLADA-dokumentin rakenne

COLLADA (COLLAborative Design Activity) on Khronos Groupin hallitsema XML-skeema. COLLADA määrittelee XML-pohjaisen skeeman, jolla pyritään tekemään 3D-datan siirtämisestä helpompaa eri ohjelmien välillä ilman että tietoa häviää. COLLADA 1.5 on nykyinen uusin määrittely tiedostomuodosta. (Khronos Group 2015c.)

COLLADA skeema määrittelee XML-elementit ja attribuutit jotka mahdollistavat monien eri ominaisuuksien esittämisen samassa dokumentissa. COLLADA-dokumentti voi määritellä mallin geometrioita, transformaatiohierarkioita, efektejä, materiaaleja, varjostimia, tekstuureita, valoja, kameroita, animaatioita, fyysikoita, käyttäjä dataa ja paljon muita ominaisuuksia. (Collada.org 2008.)

Yhdistämällä monien eri ominaisuuksien esittämisen yhteen tiedostomuotoon vältytään vaivalta luoda ja ylläpitää useita eri tiedoston käsittelijöitä. Kun käytettävät työkalut voivat tallentaa tietoa COLLADA-tiedostomuodossa, on kehittäjien helpompi työskennellä yhden tiedostomuodon parissa usean eri tiedostomuodon sijaan. (Collada.org 2008.)

Opinnäytetyössä kaikki mallidata, jota COLLADA-dokumentteihin muodostetaan, on luotu Blender -sovelluksessa. Kuvassa 15 kuvataan oleellisimpia elementtejä, joita kirjasto käsittelee ladattaessa 3D malleja Blender -sovelluksen luomasta COLLADA-dokumentista. Elementtejä, niiden sisäisiä rakenteita ja välisiä suhteita kuvataan tarkemmin myöhemmin, mutta kuva 15 antaa kokonaiskuvan mallidatan muodostavan elementin rakenteesta. Riippuen sovelluksesta, jossa mallit luodaan ja jolla COLLADA-dokumentti on generoitu, muodostetun dokumentin rakenne 3D-mallien esittämiseksi voi muodostua eri tavalla, mitä Blender -sovelluksella tehtäessä. Opinnäytetyössä käytetään Blender -sovelluksen versiota 2.74.



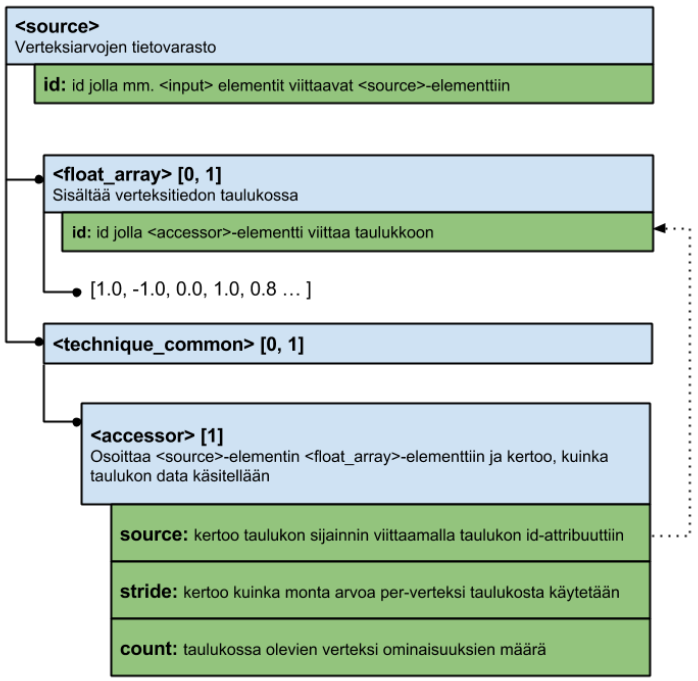
Kuva 15. Blender -sovelluksesta luodun COLLADA-dokumentin oleellisimmat elementit kirjaston purkaessa mallidataa.

Kaikki geometrisiin malleihin liittyvä data löytyy <library_geometries>-elementistä, joka sisältää yhden tai useamman geometrian määriteltynä <geometry>-elementtinä (Barens & Finch 2008). <geometry>-elementti kategorisoi geometrisen elementin määrittelyn, joka tässä tapauksessa on <mesh>-elementti (Barens & Finch 2008). <mesh>-elementti määrittelee geometrisen mallin käyttäen verteksi- ja indeksitietoa. Tiedoston <mesh>-elementti määrittelee mallin verteksitiedot, jotka kirjasto tulee purkamaan. Nämä tiedot koostuvat verteksien sijainneista, verteksien normaalivektoreista ja verteksien tekstuurikoordinaateista. <mesh>-elementti määrittää myös primitiivien muodostamisjärjestyksen indekseillä. (Barens & Finch 2008.)

<mesh>-elementin saavuttamisen jälkeen alkaa verteksi- ja indeksitiedon kerääminen <mesh>-elementin sisältämistä tiedoista. Kaikki verteksidata (sijainnit, normaalit, tekstuurikoordinaatit) on sijoitettu erillisiin <source>-elementteihin, joita on oltava vähintään yksi mallin verteksejä varten. <source>-elementti on

tiedon säilytyspaikka, johon <input>-elementit viittaavat osoittaakseen, mihin tarkoitukseen olevaa tietoa <source>-elementti sisältää. 3D-malleja purettaessa tiedon tarkoitus on jokin verteksien ominaisuuksiin liittyvä tarkoitus. <source>-elementti tallentaa liukulukutyyppisen verteksitiedon <float_array>-elementtiin, josta kirjasto kerää verteksidatan myöhemmässä vaiheessa, kun tiedon tarkoitus selviää <input>-elementin avulla. <source>-elementti määrittää myös <common_techniques>-elementin alla <accessor>-elementin, joka kertoo siitä, miten <source>-elementin määrittelemän datataulukon, eli <float_array>-elementin sisältämää tietoa tulee käsitellä. <accessor>-elemetti kertoo kuinka monesta taulukon alkiosta yksi verteksi muodostuu stride-attribuutin avulla, sekä verteksien kokonaismäärän count-attribuutilla. (Barens & Finch 2008.) Esimerkkinä, jos kyseessä olisi mallin verteksien sijainnit, jotka muodostuvat x-, z- ja y-koordinaateista, olisi <accessor>-elementin stride-attribuutti silloin 3. Jos <source>-elementti sisältäisi mallin verteksien tekstuurikoordinaatit, jotka muodostuvat arvoista s ja t, olisi <accessor>-elementin stride-attribuutin arvo silloin 2.

Kuvassa 16 esitetään <source>-elementin sisäinen rakenne, millaisia attribuutteja eri elementeillä on, esittämään elementtien välisiä viittauksia toisiinsa.

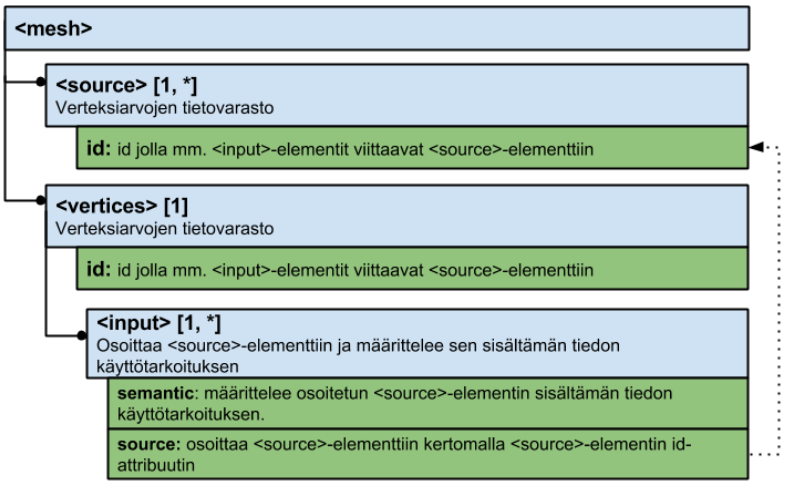


Kuva 16. Kuvaus<source>-elementin sisäisestä rakenteesta.

Alla esimerkki COLLADA-dokumenttiin muodostetusta <source>-elementistä. Kyseinen elementti määrittelee sijaintitiedot, joista 3D-mallin verteksit muodostetaan.

```
<source id="Boksi-mesh-positions">
<float_array id="Boksi-mesh-positions-array" count="24">
   -1 -1 -1 -1 1 -1 1 1 …
</float_array>
   <technique_common>
       <accessor source="#Boksi-mesh-positions-array" count="8"
                                                      stride="3">
           <param name="X" type="float"/>
           <param name="Y" type="float"/>
           <param name="Z" type="float"/>
       </accessor>
   </technique_common>
</source>
```

Jokaisen geometrian <mesh>-elementtiin on määritelty aina yksi <vertices>-elementti. <vertices>-elementti esittää vähintään sen, mistä mallin verteksit muodostavat sijaintitiedot löytyvät. Tämä tapahtuu <vertices>-elementin sisältämällä <input>-elementillä, joka osoittaa sijaintitiedot sisältävään <source>-elementtiin ja kertoo, että lähteen sisältämät verteksitiedot tulee tulkita mallien verteksien sijainteina. (Barens & Finch 2008.) Kuvassa 17 pyritään esittämään <vertices>-elementin rakenne, ja kuinka se osoittaa vähintään yhteen <source>-elementtiin, jossa verteksien sijaintitiedot määritellään, käyttämällä <input>-elementtiä. <vertices>-elementti voi määrittää muitakin <input>-elementtejä, mutta sijaintitiedot osoittava <input>-elementti on tällä hetkellä ainoa tärkeä kirjaston toiminnan kannalta.



Kuva 17. Kuvaus siitä, kuinka <vertices>-elementti rakentuu, ja kuinka se osoittaa sijainti tiedot sisältämään <source>-elementtiin <input>-elementin avulla.

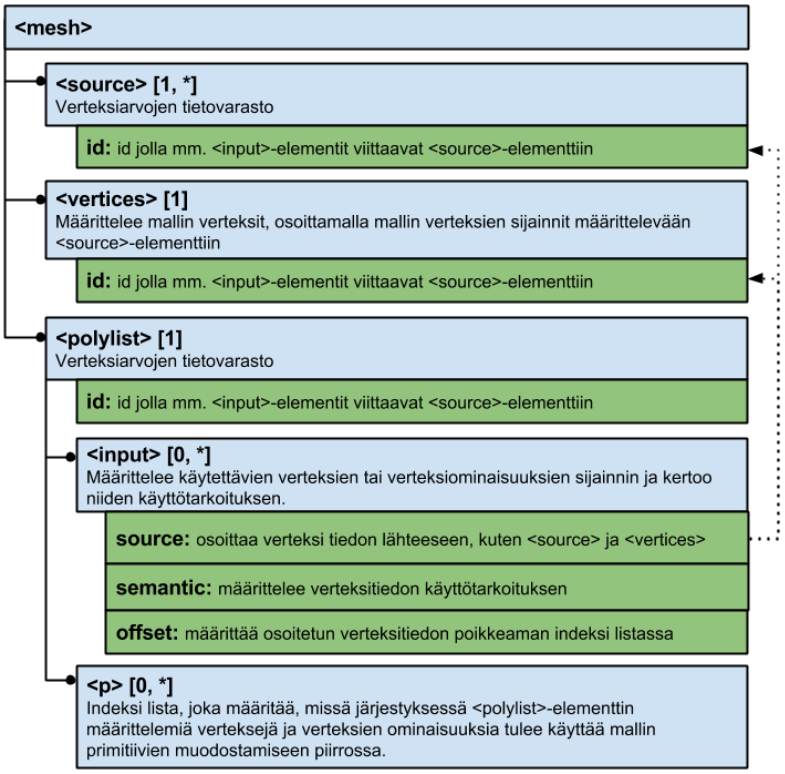
Alla on esimerkki COLLADA-dokumenttiin muodostetusta <vertices>-elementistä, joka viittaa <input>-elementillään aikaisemmin esitettyyn <source>-elementtiin, joka määritteli verteksien sijaintitiedot.

```
<vertices id="Boksi-mesh-vertices">
   <input semantic="POSITION" source="#Boksi-mesh-positions"/>
</vertices>
```

<input>-elementtejä käytetään osoittamaan ja määrittelemään <source>-elementtien sisältämän tiedon käyttötarkoitus. Käyttötarkoitus määritellään <input>-elementin semantic-attribuutin avulla, jolle voidaan määritellä arvo, joka kuvaa osoitettavan tiedon käyttötarkoitusta merkkijonolla. Yleisiä käytettäviä arvoja eri verteksitietojen käyttötarkoituksien ilmaisemiseen ovat POSITION, TEXCOORD ja NORMAL, mutta attribuutilla on myös monia muita mahdollisia arvoja. (Barens & Finch 2008.) Edellä mainitut arvot ovat kuitenkin ne arvot, joita kirjasto pystyy tässä vaiheessa käsittelemään mallien verteksitietoja purettaessa Blender -sovelluksesta tuodusta COLLADA-dokumentista.

Blender -soveluksesta tuodussa COLLADA-dokuementissa käytetään <polylist>-elementtiä määrittelemään mallin mahdolliset tekstuurikoordinaatit ja normaalivektorit. <polylist>-elementtiä käytetään sitomaan verteksit ja verteksien

ominaisuudet, kuten tekstuurikoordinaatit ja normaalivektorit, yhteen, ja organisoimaan nuo tiedot muodostettaviksi yksittäisiksi primitiiveiksi, jotka muodostavat yhdessä kokonaisen 3D-mallin. <polylist>-elementti kokoaa 3D-mallin määrittelemät verteksit ja verteksien ominaisuudet viittaamalla niiden sijainteihin dokumentissa, ja määrittelemällä niille indeksit, joilla määritellään verteksien ja niiden ominaisuuksien käyttöjärjestys primitiivien muodostamiseen. <polylist>-elementti määrittelee kaikki käytettävissä olevat verteksit ja verteksiominaisuudet osoittamalla <input>-elementeillä verteksit määrittelevään <vertices>-elementtiin, ja ominaisuudet määritteleviin <source>-elementteihin. Tämän lisäksi jokaiselle <input>-elementille on määritelty offset-attribuutti, joka kertoo verteksien ja eri verteksiominaisuuksien poikkeaman <polylist>-elementtiin muodostetussa indeksilistassa, joka on tallennettu <polylist>-elementin sisältämään <p>-elementtiin. (Barens & Finch 2008.) Kuvassa 18 hahmotetaan <polylist>-elementin <input>-elementtien ja verteksidatan määrittelevien <vertices>- ja <source>-elementtien suhdetta.

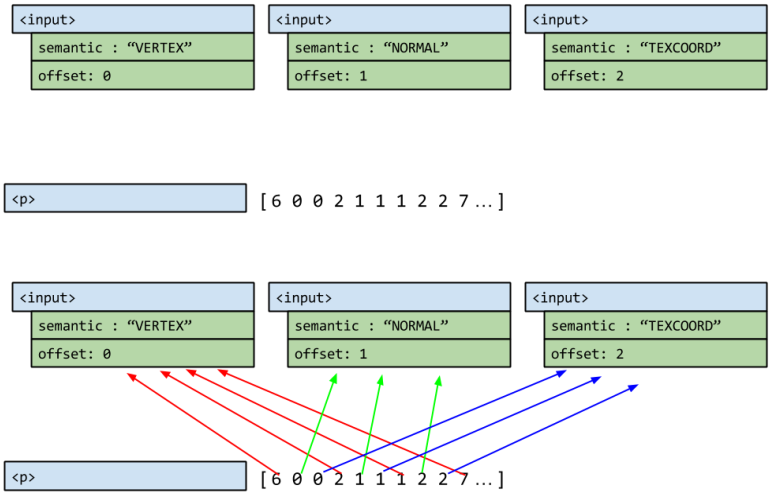


Kuva 18. <polylist>-elementti määrittelee, mistä mallin muodostavat verteksit ja verteksien muut ominaisuudet on määritelty.

Alla esimerkki COLLADA-dokumentin muodostamasta <polylist>-elementistä. <polylist>-elementti määrittää 3D-mallin muodostukseen käytettävät verteksit viittaamalla aikaisemmin esiteltyyn <vertices>-elementtiin <input>-elementillä, jonka semantic-attribuuttiin on sijoitettu arvo “VERTEX”. Vertekseihin viittaava <input>-elementti määrittelee poikkeaman, jota tullaan hyödyntämään luettaessa verteksien piirtojärjestyksen määrittelevän <p>-elementin käsittelyssä. <polylist>-elementti määrittelee myös 3D-mallin normaali- ja tekstuuriominaisuuksien sijainnit dokumentissa osoittamalla <source>-elementteihin, jotka sisältävät nuo tiedot. <polylist>-elementti määrittelee molemmille ominaisuuksille oman poikkeaman offset-attribuutissa.

```
<polylist count="12">
   <input semantic="VERTEX" source="#Boksi-mesh-vertices" offset="0"/>
   <input semantic="NORMAL" source="#Boksi-mesh-normals" offset="1"/>
   <input semantic="TEXCOORD" source="#Boksi-mesh-map" offset="2" />
   ...
   <p>6 0 0 2 1 1 1 2 2 7 ... </p>
</polylist>
```

<polylist>-elementin jokaiselle verteksiin ja verteksien ominaisuuksiin viittaavalle <input>-elementille on määritelty poikkeamaa kuvaava offset-attribuutti siksi, koska verteksien ja verteksien ominaisuuksien indeksit määrittelevä <p>-elementti määrittelee vuoron perään järjestyksen, jossa <input>-elementeillä määriteltyjä arvoja tulee käyttää. Poikkeaman avulla tiedetään, montako arvoa listassa tulee liikkua, jotta listassa oleva indeksi viittaa <input>-elementin määrittelemän tiedon käyttöjärjestykseen. Pienimmän poikkeaman omaava <input>-elementti sijoittuu aina ensimmäiseksi lukujärjestyksessä, ja suurimman omaava viimeiseksi lukujärjestyksessä. (Barens & Finch 2008.)



Kuva 19. Indeksien järjestys <p>-elementissä <input>-elementtien poikkeaman mukaan.

Yllä olevassa kuvassa (kuva 19) yritetään hahmottaa mihin <input>-elementtiin <p>-elementin määrittelemät indeksit viittaavat missäkin vaiheessa vuoronperään. Verteksien indeksit tulevat aina ensin kolmesta peräkkäisestä, toisena tulee normaalivektoreiden indeksi, ja kolmantena tekstuurikoordinaattien indeksi.

Opinnäytetyössä Blender -sovelluksesta tuodun COLLADA-dokumentin kanssa muodostui muutamia ongelmia muun muassa dokumenttiin muodostettujen indeksien erottelussa, sekä mallille muodostettujen tekstuurikoordinaattien kanssa. Näitä ongelmia käsitellään seuraavassa luvussa tarkemmin.

4.1.4 Mallidatan lataaminen COLLADA-dokumentista

Renderer-luokka määrittelee loadModelFromCollada-metodin, joka pystyy lataamaan COLLADA-tiedostoon tallennetut 3D-mallit käytettäväksi kirjaston avulla. 3D-mallit ladataan Renderer-luokan loadedModels-objektiin, johon jokaisesta ladatusta mallista voidaan tallentaa kaikki verteksidata kuten sijainnit, tekstuurikoordinaatit ja normaalit, sekä mallin indeksit, jotka kertovat verteksien käyttöjärjestyksen primitiivien muodostamiseen.

Renderer-luokan loadModelFromCollada-metodi ottaa parametreina vastaan COLLADA-dokumentin, joka on tuotu Blender -sovelluksesta, sekä nimen, jolla dokumentin sisältämä 3D-malli tallennetaan Renderer-luokan loadedModels-objektiin. Metodi aloittaa purkamisprosessin hakemalla kaikki <geometries>-elementit, jotka on määritelty <library_geometries>-elementin alle. Tämän jälkeen jokainen löydetty geometria käsitellään vuorotellen, purkamalla kaikki verteksit, verteksien ominaisuudet ja indeksit, jotka kullekin geometrialle on määritelty.

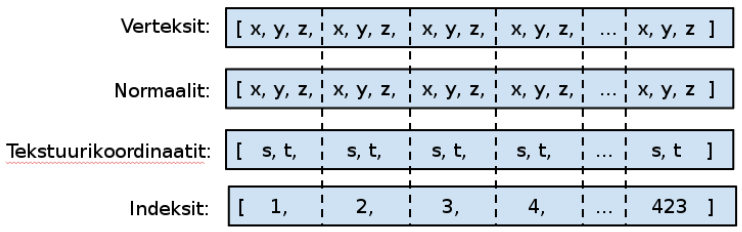
Metodi purkaa kaikkien COLLADA-dokumentissa määriteltyjen 3D-mallien tiedot Renderer-luokan loadedModels-objektiin, käymällä läpi jokaisen <library_geometries>-elementtiin sijoitetun <geometry>-elementin, jotka määrittelevät kukin yhden mallin. Jokaisen COLLADA-dokumentista löydetyn mallin purkaminen aloitetaan luomalla uusi tyhjä malli-objekti Renderer-luokan loadedModels-objektiin. Tyhjässä mallissa määritellään rakenne, jota käytetään purettavan mallin tietojen tallentamiseen purkamisvaiheessa, ja myöhemmin sen käyttämiseen piirrettävien MeshObject-olioiden luomisessa.

Kun tyhjä malli-objekti on luotu, alkaa metodi purkaa tietoa mallin määrittelevästä <mesh>-elementistä. Ensin dokumentista kerätään kaikki <source>-elementit, jotka sisältävät muun muassa mallin verteksit, normaalit ja tekstuurikoordinaatit, joita kirjastolla voidaan piirtämisessä käyttää. <source>-elementtien sisältämät tiedot tallennetaan myöhempää käyttöä varten objektiin, josta niitä on helpompi käsitellä.

Kun metodi on kerännyt kaiken verteksidatan <source>-elementeistä, selvittää metodi ensin, mikä <source>-elementti sisältää verteksien sijainnit määrittelevät tiedot. Tämä tapahtuu hakemalla mallin <vertices>-elementti, joka viittaa <input>-elementillään kyseiseen <source>-elementtiin. Kun sijainnit määrittelevä <source>-elementti löytyy, tallennetaan sen sisältämät tiedot aikaisemmin luotuun malli-objektiin. Tässä vaiheessa mallille on määritelty verteksit, joiden avulla se voidaan piirtää. Malliin saatetaan lisätä lisää verteksejä myöhemmin, kun tekstuuri koordinaatteja tarkistetaan metodin loppupuolella.

Kun metodi on käsitellyt <vertices>-elementin, siirtyy se käsittelemään <polylist>-elementissä määriteltyjä <input>- ja <p>-elementtejä. <polylist> kokoaa kaikki mallin verteksiominaisuudet viittaamalla niihin <input>-elementeillään. Jokainen <input>-elementti osoittaa joko mallin verteksit määrittävään <vertices>-elementtiin, tai <source>-elementteihin, jotka määrittelevät mallin käyttämät normaalivektorit ja tekstuurikoordinaatit. Jokaisen <input>-elementin semantic-attribuutin arvo tarkistetaan, minkä perusteella metodi tallentaa normaalivektorit tai tekstuurikoordinaatit sekä niiden indeksit malli-objektiin, tai parsii verteksien piirrossa käyttämät indeksit malli-objektiin. Indeksien parsimisessa otetaan huomioon kunkin <input>-elementin offset-attribuutin poikkeama arvo, sekä suurin offset-attribuutin arvo kaikkien input-elementtien kesken, jotta oikeat indeksiarvot saadaan eroteltua indeksit listaavasta <p>-elementistä.

Kuitenkin, tässä vaiheessa ongelmaksi muodostuu se, että Blender -sovelluksen tuomassa COLLADA-dokumentissa saatetaan määritellä useampia eri tekstuurikoordinaatteja, jotka osoittavat samaan verteksiin. Kirjastolla piirrettäessä, verteksipuskurit tulee muodostaa siten, että jokaiselle verteksille on omat tekstuurikoordinaatit ja normaalivektorit samoissa sijainneissa omissa puskureissaan verrattuna verteksin sijaintiin omassa puskurissaan. Tällöin kun indeksejä käytetään primitiivien muodostuksessa, sama indeksi osoittaa samaan sijaintiin kaikissa puskureissa, joita piirtämisessä käytetään. Kuvassa 20 pyritään hahmottamaan eri puskureiden arvojen sijoittumista indekseihin nähden asettamalla poikkiviivoja merkitsemään, kuinka eri puskureiden arvojen tulisi asettua indekseihin nähden.



Kuva 20. Hahmotelma siitä, kuinka eri puskureiden arvojen tulisi sijoittua puskuriin indeksi dataan nähden.

Käsiteltävä COLLADA-tiedosto saattaa kuitenkin määritellä mallille useampia tekstuurikoordinaatteja, kuin mitä verteksejä ja normaalivektoreita on määritelty. Tämän vuoksi metodin tulee selvittää, mihin verteksiin on liitetty mitäkin tekstuurikoordinaatteja käyttämällä hyödyksi indeksejä, jotka on määritelty dokumentissa vertekseille ja tekstuurikoordinaateille.

Metodi käy läpi kaikki vertekseille asetetut indeksit, ja tallentaa jokaisen kohdalla tiedon siitä, mitä tekstuurikoordinaatteja kunkin indeksin kohdalla on määritelty. Tekstuurikoordinaatin lisäksi talteen otetaan indeksi, jossa tekstuurikoordinaatin tulisi esiintyä lopullisessa taulukossa.

Jos samalla indeksillä tulee vastaan tekstuurikoordinaatti, joka on eriarvoinen kuin aikaisemmin tavattu koordinaatti, luodaan mallin verteksejä esittävän taulukon loppuun uusi verteksi, joka on kopio indeksin osoittamassa sijainnissa olevasta verteksistä. 3D-mallin normaalivektoreita esittävään taulukkoon luodaan myös kopio verteksin normaalivektorista. Kun kopiot on luotu, käsiteltävän indeksin tilalle sijoitetaan uusi arvo, joka osoittaa mallin verteksejä esittävän taulukon lopussa olevaan uuteen kopioituun verteksiin.

Jos käsiteltävä indeksi on käyty aikaisemmin läpi, mutta tekstuurikoordinaatti on jokin aikaisemmin vastaan tulleista koordinaateista, muutetaan käsiteltävä indeksi vastaamaan tekstuurikoordinaatille asetettua indeksiä, joka voi olla alkuperäinen indeksi, tai uusi indeksi, joka osoittaa uutteen verteksiin mallin verteksitaulukon lopussa.

Kun kaikki indeksit on käsitelty, tulisi verteksien ja normaalivektoreiden määrä vastata tekstuurikoordinaattien määrää. Kopioiden luominen on välttämätöntä, jos haluamme sijoittaa samassa sijainnissa olevaan verteksiin useamman eri tekstuurikoordinaatin. Kuution muotoisen mallin teksturointi on yksi hyvä esimerkki tästä tilanteesta, sillä jokainen verteksi on osa kolmea eri kuution sivua.

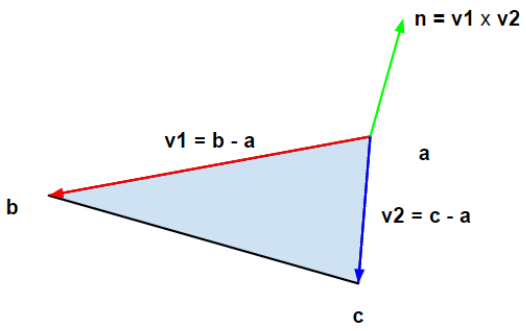
Kun metodi on luonut tarvittavat kopiot vertekseistä ja normaalivektoreista, järjestellään tekstuurikoordinaatit siten, että niistä muodostettu uusi taulukko on oikeassa järjestyksessä vertekseihin nähden. Tämän jälkeen mallin sisältämät

arvot vertekseille, normaalivektoreille ja tekstuurikoordinaateille sekä verteksien indekseille korvataan uusilla arvoilla.

Tämän jälkeen, jos COLLADA-dokumentti määrittelee useampia eri malleja, aloitetaan uuden mallin muodostaminen luokan loadedModels-objektiin. Malli on käytettävissä viittaamalla siihen loadedModels-objektissa mallille määritetyn nimen avulla, mikä annettiin parametrina loadModelsFromCollada-metodille.

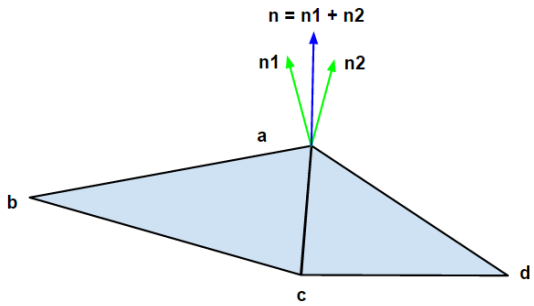
4.1.5 Normaalivektoreiden generointi

Renderer-luokka määrittää calculateNormals-metodin, joka laskee sille parametrina annetun loadedeModels-objektiin ladatun mallin normaalivektorit uudestaan, ja korvaa mallin normaalivektorit uusilla vektoreilla. Kolmioista muodostuvan tason normaalivektori voidaan määritellä tason pinnalla olevan pisteen ja kahden keskenään erisuuntaisella vektorilla, jotka määrittelevät pisteen kautta kulkevan tason suunnan. Vektoreille voidaan suorittaa ristitulo, josta tuloksena saatava vektoreihin nähden kohtisuora vektori määrittää pinnan normaalivektorin. (Puhakka 2008, 41–42.) Kuva 21 esittää tilanteen, jossa kolmiopinnan normaalivektori määritetään käyttämällä kolmion verteksien välille muodostuvia vektoreita. Kolmion verteksi **a** toimii pisteenä, josta määritetään kaksi vektoria (**v1** ja **v2**) verteksien **b** ja **c** välille. Näiden vektoreiden välille suoritetaan ristitulo, josta saadaan tuloksena vektori **n**, joka on pinnan normaalivektori, eli vektori, joka on pinnan kanssa kohtisuorassa.



Kuva 21. Pinnan normaalivektorin laskeminen pinnan verteksien välisten vektoreiden avulla

Pinnalle muodostettu verteksi voidaan sijoittaa kaikkien pinnalle kuuluvien verteksien normaalivektoreiksi. Jos yksittäinen verteksi on osana useampaa eri pintaa, verteksin uusi normaalivektori muodostaa laskemalla verteksin kaikilta pinnoilta saamat normaalivektorit keskenään yhteen. Näin kaikkien pintojen normaalivektorit vaikuttavat verteksien lopulliseen normaalivektoriin. Kuvassa 22 hahmotetaan verteksille **a** muodostettava normaalivektori kahdesta eri pinnasta, joiden muodostamista sitä käytetään.



Kuva 22. Verteksin lopullinen normaalivektori muodostetaan eri pinnoista saatavien normaalivektoreiden yhteenlaskun avulla.

Renderer-luokan calculateNormals-metodi ottaa parametreina ladatun mallin, ja tiedon, tuleeko lasketut vektorit kääntää osoittamaan vastakkaiseen suuntaan. Vastakkaiseen suuntaan osoittaminen voi olla siksi tärkeää, koska ristitulon kertomisjärjestys vaikuttaa lopputuloksena saatavan vektorin suuntaan. Jotkut mallit voivat olla määritellä piirrettävän mallin verteksit siten, että ristitulon laskeminen tuottaa vektorin, joka osoittaa mallin sisään mallista poispäin osoittavien normaalivektoreiden sijaan

Normaalien laskeminen suoritettaan muuttamalla mallin määrittelemät verteksit Vector3f-olioiksi, joilla vertekseille voidaan soveltaa Vector3f-luokan määrittelemiä toiminnallisuuksia. Metodi hyödyntää mallille määritettyjä indeksejä selvittämään sen, missä kaikissa pinnoissa samaa vektoria käytetään. Metodi käy mallin läpi pinta kerrallaan, ja laskee pinnan muodostaman normaalivektorin kuvan 21 mukaisella periaatteella. Kun normaalivektori on laskettu, tallennetaan se jokaiselle pinnan muodostavalle verteksille siten, että se yhdistetään samalle verteksille aikaisemmin määritettyjen normaalivektoreiden kanssa kuvan 22 mukaisella periaatteella, eli laskemalla vektorit yhteen. Kun kaikki normaalivek-

torit on laskettu, tallennetaan ne malliin uusina normaalivektoreina, jotka korvaavat vanhat.

4.1.6 Piirtäminen Renderer-luokalla

Renderer-luokan render-metodia kutsutaan silloin, kun kirjastolla halutaan piirtää kuva käyttäen aktiiviseksi asetettuun maailmaan lisättyjä MeshObject-, CameraObject-, PointLightObject- ja SpotlightObject-olioita.

Kun renderer-metodi kutsutaan, otetaan metodin alussa käyttöön Renderer-olion activeWorld-muuttujaan sijoitettu maailma-objekti, jonka sisältämiä tietoja halutaan käyttää kuvan muodostamiseen piirrossa. Maailma-objektille on annettu vähintään yksi CameraObject-olio, joka on asetettu maailma-objektin aktiiviseksi kameraksi. Aktiivisen CameraObject-olion määrittelemiä transformaatioita ja projektiomatriisia käytetään verteksien sijaintien muuntamiseen maailma-avaruudesta kamera-avaruuteen ja lopuksi kamera-avaruudesta ruutuavaruuteen, joka on määrittää verteksien lopullisen sijainnin piirrettävässä kuvassa. CameraObject-olion määrittelemät uniform-muuttujat säilyvät samoina jokaiselle piirrettävälle MeshObject-oliolle render-metodin käsittelyn edetessä.

CameraObject-olion määrittelemien arvojen lisäksi, kaikki valaistusta varten määritetyt arvot ja oliot ovat samat jokaiselle piirrossa käytätettävälle MeshObject-oliolle. Piirtoon käytettävästä maailma-objektista otetaan globaalin valaisutuksen määrittelevät arvot suunnatulle valolle ja ambienttivalolle, jotka lähetetään uniform-muuttujina varjostimille. Maailmaan voidaan lisätä myös paikallista valaistusta tuottavia PointLightObject- ja SpotlightObject-olioita. Paikallista valaistusta varten voidaan määritellä useita eri PointLightObject- ja SpotlightObject-oloita. Koska olioita voi olla useita, niitä varten on luotu uniform-muuttujat, jotka määrittävät useilta eri PointLightObject- ja SpotlightObject-olioilta saatuja arvoja taulukossa. Myös maailma-objektiin lisättyjen PointLightObject- ja SpotlightObject-oloiden määrä kerrotaan fragmenttivarjostimelle. Varjostin kykenee käyttämään piirrossa vain Renderer-luokan määrittämän maksimimäärän mukaisen määrän PointLightObject- ja SpotlightObject-olioita. Jos maksimimäärää

muutetaan, täytyy Renderer-luokan määrittelemät varjostinsovellukset kääntää uudestaan createShaders-metodilla.

Kun renderer-metodin piirtokutsun aikana vakioina säilyvät kamera- ja valaistusarvot on asetettu, voidaan maailmaan lisättyjä MeshObject-olioita alkaa käydä läpi. Jokainen MeshObject-olio piirretään yksi kerrallaan, jolloin niiden määrittämät attribute- ja uniform-muuttujien arvot muuttuvat joka WebGL-rajapinnan drawElements-funktion kutsun yhteydessä. Funktio käyttää asetettuja attribute- ja uniform-arvoja kuvan muodostamiseen kuvapuskuriin.

MeshObject-oliot käyttävät niiden osoittaman malli-objektin sisältämiä verteksi- ja indeksipuskureita attribute-muuttujien muodostamiseen, sekä drawElements-funktion kutsuun. Jokaisen MeshObject-olion määrittelemät transformaatiot muutetaan matriisiksi, joka voidaan lähettää varjostimille suorittamaan 3D-mallin verteksien muuntamisen maailma-avaruuden koordinaatteihin. MeshObject-olioiden määrittelemät pinnan ominaisuudet, kuten pinnan spekulaariheijastuksen ja pinnan värin arvot, siirtyvät uniform-muuttujina fragmenttivarjostimelle, jossa niitä käytetään pinnan värityksen määrittelyyn. Jos MeshObject-oliolle on asetettu käytettäväksi tekstuuri-objekti, käytetään sitä piirrettävän kappaleen pinnan värien määrittelyyn. MeshObject-olio käyttää Renderer-luokan määrittelemää oletustekstuuria niin kauan, kunnes mallille asetettu tekstuuri on valmis käytettäväksi, eli kun tekstuurin käyttämä kuva on ladattu muistiin. Lopuksi MeshObject-olion määrittelemillä arvoilla piirretään kuvapuskuriin kuva drawElements-kutsulla, joka käyttää olion viittaaman malli-objektin määrittelemiä indeksejä piirrettävän kappaleen primitiivien muodostamiseen.

Kuvapuskuria päivitetään silmukassa esiintyvillä MeshObject-olioiden määrittelemillä arvoilla, ja lopullinen kuva muodostuu kaikkien MeshObject-olioiden määrittelemistä kappaleista, ja niiden pinnoille kohdistuneista valaistuksista. Liitteessä 2 esitetyssä taulukossa on määritetty jokaisien varjostimissa määritetyn uniform- ja attribute-muuttujan tarkoitus piirrossa tarkemmin.

4.2 BaseObject-luokka ja sen alustus

BaseObject-luokka (WEBGL_LIB.BaseObject) toimii perustana jokaiselle muulle kirjastossa määritetylle luokalle, jonka tulee pystyä suorittamaan erilaisia transformaatioita 3D-avaruudessa. Tällaisia luokkia ovat MeshObject-, CameraObject-, PointLightObject- ja SpotlightObject-luokat.

BaseObject-luokan keskeisimpiin toimintoihin kuuluu tarjota toiminnot 3D-objektien translaatio-, rotaatio- ja skaalaustransformaatioiden suorittamiseen. Transformaatiot muokkaavat BaseObject-luokan transformations-objektiin tallennettujen vektori- ja kvaternio-objektien arvoja. Näiden objektien sisältämistä arvoista muodostetaan matriisi, joka toteuttaa kaikki objektille määritetyt transformaatiot niille vertekseille, jotka kerrotaan sen avulla.

Esimerkkinä luokan perimisestä toimii MeshObject-luokka. Kun Renderer-luokan oliolla suoritetaan piirtokomento, haetaan kaikista MeshObject-luokan oloista transformaatiot määrittävä matriisi BaseObject-luokasta perityn getTransformationMatrix-metodin avulla. Metodilla saatu matriisi lähetetään verteksivarjostimelle uniform-muuttujana, jolla MeshObject-oliolle määritetyt transformaatiot suoritetaan verteksivarjostimen käsittelemille vertekseille.

Kun BaseObject-luokkaa ja siitä periytettyjä luokkia alustetaan, asetetaan luokan transformations-objektin sisällä olevien kvaternioiden ja vektoreiden arvot siten, ettei objekti suorita minkäänlaisia transformaatioita siitä muodostetun transformaatiomatriisin avulla. Toisin sanoen, jos objektin avulla muodostetaan transformaatiomatriisi, tulisi tuon matriisin olla identiteettimatriisi, eli matriisi, joka ei muuta sillä kerrottuja muiden matriisien tai vektoreiden arvoja millään lailla.

BaseObject-luokan translations-objekti sisältää Quaternion-luokan oliot worldOrientation ja localOritentation, jotka määrittävät kappaleen kierron määrän. Jotta kvaterniot eivät määrittele minkäänlaista rotaatiota objektille, alustetaan niiden w-komponentit arvolla 1.0 ja vektori-komponenttien kaikki arvot arvolla 0.0, jotka

muodostavat identiteettikvaternion, eli kvaternion, joka ei määrittele minkäänlaista rotaatiota.

Rotaatiot määrittelevien kvaternioiden lisäksi alustetaan transformations-objektin sisältämät scale- ja translation-objektit, jotka ovat Vector3f-luokan olioita. Location-vektori määrittelee objektin suorittaman translaation suuruuden x-, y- ja z-akseleiden suunnissa. Näin ollen location-vektorin kaikki komponentit alustetaan arvolla 0, jolloin alustettu BaseObject-olio sijoittuu aina luodessa maailma-avaruuden origoon, eli pisteeseen (0, 0, 0).

Luokan transformations-objektin scale-vektori määrittelee luokan toteuttaman skaalauksen määrän x-, y- ja z-akseleiden suunnassa. Skaalauksen alustaminen eroaa translaation määrittävän vektorin alustamisesta siten, että scale-vektorin x-, y- ja z-komponenttien arvot tulee asettaa arvoksi 1.0. Jos skaalauksen suuruus on pienempi kuin 1.0, kutistuu kappale, ja jos se on suurempi, kappale kasvaa. Jos skaalaukseen suuruus on negatiivinen, kappale peilautuu. Alla olevassa koodissa nähdään BaseObject-luokan alustuksessa muodostetut arvot luokan transformations-objektin sisältämille transformaatiot määritteleville objekteille.

```
this.transformations = {
  worldOrientation : new WEBGL_LIB.Math.Entities.Quaternion(1.0,
                        new WEBGL_LIB.Math.Entities.Vector3f(0, 0, 0)),
  localOrientation : new WEBGL_LIB.Math.Entities.Quaternion(1.0,
                        new WEBGL_LIB.Math.Entities.Vector3f(0, 0, 0)),
  location : new WEBGL_LIB.Math.Entities.Vector3f(0, 0, 0),
  scale : new WEBGL_LIB.Math.Entities.Vector3f(1, 1, 1),
};
```

BaseObject-luokan metodit mahdollistavat kappaleen kiertämisen objekti- ja maailma-avaruuden akseleiden ympäri, sekä kappaleen translaation ja skaalauksen x-, y- ja z-akseleiden mukaisessa suunnassa.

Jokainen funktio muokkaa luokan translations-objektin sisältämien vektoreiden tai kvaternioiden arvoja. Kun BaseObject-luokan oliolle, tai siitä perityn luokan oliolle suoritetaan transformaatio, muokataan transformations-objektin sisältämien objektien arvoja silloin pysyvästi. Rotaatioita suorittavat metodit luovat parametreina annetuista arvoista uuden kvaternion, joka kerrotaan metodin suorittamaa rotaatiota vastaavan kvaternion kanssa. Jos metodi kiertää maailmaavaruuden akseleiden ympäri, muokataan translations-objektin worldOrientation-kvaternion arvoa. Jos metodi suorittaa kierron paikallisen objektiavaruuden akseleiden ympäri, käsitellään translations-objektin localOrientation-kvaternioita. Kvaternioiden tulosta saatu uusi kvaternio sijoitetaan translations-objektin uudeksi worldOrientation tai localOrientation arvoksi.

Translaation ja skaalauksen suoritettavien metodien tapauksessa muokataan transformations-objektin sisältämien translation- ja scale-vektorien arvoja lisäämällä parametreina annetut arvot vektoreiden komponenttien olemassa oleviin arvoihin. Alla olevassa koodissa on BaseObject-luokan määrittelemä translateWorldXYZ-metodi, jolla siirtää BaseObject-oliota x-, y- ja z-akseleiden mukaisissa suunnissa metodin parametrien avulla. Parametreina annetut arvot lisätään yksinkertaisesti kappaleen sijainnin määrittelevän vektorin komponenttien arvoihin.

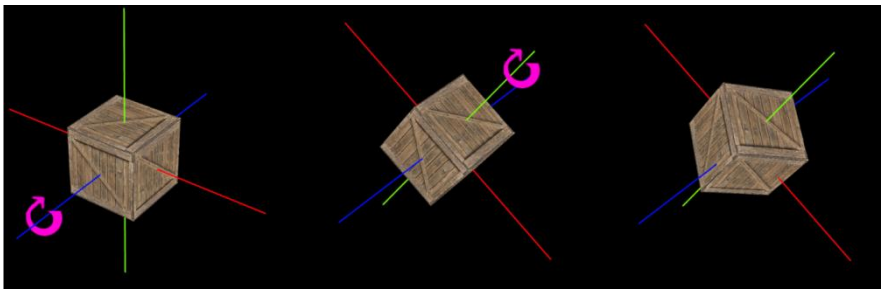
```
WEBGL_LIB.BaseObject.prototype.translateWorldXYZ =
                                                function(trX, trY,  trZ){
   this.transformations.location.x += trX;
   this.transformations.location.y += trY;
   this.transformations.location.z += trZ;
};
```

Alla olevassa koodissa on BaseObject-luokan määrittelemä rotateLocalXYZ-metodi rotaation suorittamiseen paikallisen objektiavaruuden x-, y- ja z-akseleiden ympäri. Rotaatio toteutetaan luomalla ensin parametrien avulla kvaterniot, jotka toteuttavat parametrina annetun kulman suuruisen kieroon x-, y- tai z-akselin ympäri. Kun kaikki kolme kvaterniota on luotu, yhdistetään niiden suorittamat kierrot kertomalla ne yhteen. Tämän jälkeen yhdistetty kvaternio kerro-

taan BaseObject-luokan paikallisen kierron ilmoittavan localOrientation-kvaternion kanssa, mikä muuttaa kappaleen sen hetkistä orientaatiota yhdistetyn kvaternion määrittelemän kierron mukaisesti.

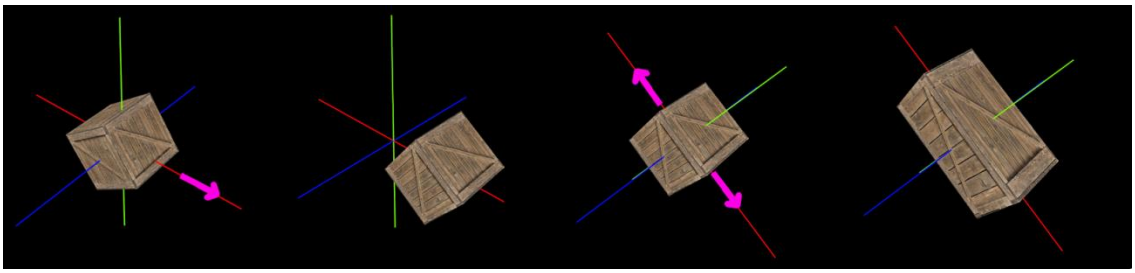
```
WEBGL_LIB.BaseObject.prototype.rotateLocalXYZ =
                                              function(angX, angY, angZ){
   var rotQuatX = WEBGL_LIB.Math.getRotationQuat(angX,
                                                    this.baseVectors.x);
   var rotQuatY = WEBGL_LIB.Math.getRotationQuat(angY,
                                                    this.baseVectors.y);
   var rotQuatZ = WEBGL_LIB.Math.getRotationQuat(angZ,
                                                    this.baseVectors.z);
   var rotQuat = rotQuatX.multQuaternion(
                               rotQuatY.multQuaternion(rotQuatZ));
   this.transformations.localOrientation = rotQuat.
          multQuaternion(this.transformations.localOrientation);
};
```

Ero objekti- ja maailma-avaruuden akseleiden ympäri suoritettavissa metodeissa on se, että objektiavaruudessa suoritettavat kierrot muuttuvat, jos kappaletta kierretään maailma-avaruuden akseleiden ympäri. Tätä tilannetta pyritään havainnollistamaan kuvassa 23, jossa kuutiokappaletta kierretään ensin maailma-avaruuden z-akselin ympäri. Kuvasta nähdään kuinka kappaleen paikalliset x-, y- ja z-akselit kiertyvät kappaleen mukana. Tämän jälkeen kappaletta kierretään paikallisen y-akselin ympäri, mikä muuttaa kappaleen orientaatiota, mutta ei vaikuta kappaleen kierrossa käytettävien paikallisten akseleiden orientaatioon. Tällä tavalla kirjaston avulla suoritettaville kappaleiden kierroille pyritään antamaan hieman lisää muokkausmahdollisuuksia.



Kuva 23. Kappaleen kiertäminen maailma-akseleiden ympäri vaikuttaa kappaleen paikallisten akseleiden orientaatioon.

Kappaleen translaatio tapahtuu aina maailman x-, y- ja z-akseleiden suuntaisesti, vaikka kappaletta kierretään maailma-avaruudessa, mutta kappaleen skaalaus tapahtuu kappaleen paikallisten akseleiden suuntaisesti. Tätä tilannetta havainnollistetaan kuvassa 24 suorittamalla ensin kierretylle kappaleelle translaation x-akselin suuntaisesti, minkä jälkeen kappaletta skaalataan x-akselin suuntaisesti. Kuvan tilanteesta voidaan havainnoida, kuinka translaatio tapahtuu maailma akseleiden suuntaisesti, ja skaalaus kierron seurauksena muuttuneiden paikallisten akseleiden suuntaisesti.



Kuva 24. Rotaatio tapahtuu aina maailman akseleiden suuntaisesti, ja skaalaus tapahtuu objektin paikallisten akseleiden suuntaisesti.

BaseObject-luokka määrittää getTransformationMatrix-metodin, jonka avulla kappaleen määrittelemät rotaatio-, translaatio- ja skaalaustransformaatiot kootaan yhteen matriisiin. Matriisin avulla voidaan suorittaa transformaatioita vertekseille esimerkiksi piirron yhteydessä lähettämällä metodista saatu matriisi verteksivarjostimelle uniform-muuttujana.

Metodi luo skaalauksen ja translaation suorittavat matriisit kirjaston Math-nimiavaruuteen määritettyjen getScaleMat4f- ja getTranslationMat4f-funktioiden avulla. Funktiot ottavat vastaan parametreina kunkin akselin suunnassa suori-

tettavien skaalauksen ja translaatioiden määrän. Funktiot palauttavat tuloksena matriisin, joka suorittaa kyseisen funktion määrittelemän transformaation parametreina annettujen arvojen mukaisesti.

Rotaation suorittava matriisi muodostetaan kertomalla paikallisen rotaation muodostava localOrientation-kvaternio maailmassa tapahtuvan rotaation muodostavan worldOrientation-kvaternion kanssa. Kun rotaatiot on yhdistetty yhdeksi kvaternioksi, muodostetaan siitä matriisi Quaternion-luokalle määritetyllä getRotationMatrix-metodilla, joka muodostaa sitä kutsuneen Quaternion-olion määrittämän rotaation suorittavan matriisin, ja palauttaa sen metodin paluuarvona.

Lopuksi translaation, rotaation ja skaalauksen suorittavat matriisit yhdistetään kertomalla ne keskenään yhdeksi matriisiksi. Matriisien kertomisjärjestys on asetettu siten, että ensin skaalausmatriisi kerrotaan rotaatiomatriisilla, ja näistä saatu matriisi kerrotaan viimeisenä translaation suorittavalla matriisilla. Näin kappaleen rotaatio ja skaalaus tapahtuvat ennen kappaleen siirtoa, jotta kappaleen kierto suoritetaan oikein. Alla olevassa koodissa esitellään, kuinka getTransformationMatrix-metodi suorittaa ohjelmistossa BaseObject-luokalle ja siitä periytetyille luokille määritettyjen transformaatioiden yhdistämisen yhteen palautettavaan matriisiin.

```
WEBGL_LIB.BaseObject.prototype.getTransformationMatrix = function(){
    var transMat = WEBGL_LIB.Math.getTranslationMat4f(
                                this.transformations.location.x,
                                this.transformations.location.y,
                                this.transformations.location.z);
    var rotQuat = this.transformations.
                  localOrientation.multQuaternion(
                        this.transformations.worldOrientation);
    var rotMat = rotQuat.getRotationMatrix();
    var sclMat = WEBGL_LIB.Math.getScaleMat4f(
                                this.transformations.scale.x,
                                this.transformations.scale.y,
```

```
                                this.transformations.scale.z);
    return transMat.matrixMult(rotMat.matrixMult(sclMat));
}
```

4.3 MeshObject-luokka ja sen käyttö

MeshObject-luokka (WEBGL_LIB.MeshObject) perii BaseObject-luokan toiminnot, ja sitä käytetään piirrettävien kappaleen määrittämiseen, kappaleen transformaatioiden määrittelyyn, sekä kappaleen pinnan värityksen määritellyn. MeshObject-luokka käyttää Renderer-luokan lataamia 3D-malleja piirrettävän kappaleen määrittelyyn, ja määrittää piirrettävälle kappaleelle suoritettavat transformaatiot BaseObject-luokalta perittyjen toiminnallisuuksien avulla. Luokka voi hyödyntää Renderer-luokan lataamia tekstuureita pinnan värityksen määrittelyyn, tai vaihtoehtoisesti käyttää MeshObject-oliolle määritettyä yksittäistä väriä piirrettävän kappaleen värinä. MeshObject-luokka määrittele myös millaisen spekulaariheijastuksen kappaleen pinnoille osuvat valot muodostavat piirrossa.

MeshObject-olion alustuksen yhteydessä määritetään Renderer-luokan muodostama malli, jonka sisältämiä tietoja käytetään oliota piirrettäessä. MeshObject-luokka voi määritellä mallille suoritettavia transformaatioita käyttämällä BaseObject-luokalta perittyjä toiminnallisuuksia. Alla olevassa koodiesimerkissä luodaan uusi BaseObject-olio, ja asetetaan kappaleelle translaatio ja kierto.

```
var mesh = new WEBGL_LIB.MeshObject(
                            renderer.loadedModels["modelName"]);
mesh.translateWorldXYZ(5, 0, 4);
mesh.rotateLocalXYZ(Math.PI/4, Math.PI/6, 0);
```

MeshObject-olion avulla voidaan vaikuttaa piirrettävän kappaleen pinnan väritykseen määrittelemällä pinnan käyttämä väri sekä kappaleen muodostama spekulaariheijastus valon osuessa kappaleen pintaan. Luokka määrittelee kaksi metodia näiden ominaisuuksien muokkaamisen, setSurfaceColor- ja setSpecu-

larReflection-metodit. Pinnan väri määritetään setSurfaceColor-metodin parametrina ottamalla Vector3f-oliolla, joka määrittää pinnalle asetettavan RGB-värin. Pinnan muodostama spekulaariheijastus muodostetaan setSpecularReflection-metodin vastaanottamien parametreina ottamien arvojen avulla. Parametrit määrittävät spekulaariheijastuksen terävyyden liukulukuna ja heijastuksen värin vektorin avulla. Alla olevassa koodissa nähdään metodeiden käyttö.

```
mesh.setSpecularReflection(20,
                new WEBGL_LIB.Math.Entities.Vector3f(0.0, 0.0, 0.0));
mesh.setSurfaceColor(
                new WEBGL_LIB.Math.Entities.Vector3f(1.0, 0.0, 0.0));
```

Jos MeshObject-olion halutaan käyttävän tekstuuria yksittäisen värin sijaan, käytetään luokan määrittelemää setTexture-metodia. Metodi asettaa MeshObject-olion käyttämään parametrina annettua Renderer-luokan lataamaa ja muodostamaa tekstuuria piirrettävän kappaleen pinnan värin määrittelyyn. Tekstuuri korvaa oliolle määritetyn pinnan värin, eikä tekstuurin ja pinnan väriä voida sekoittaa keskenään. Alla olevassa koodissa nähdään esimerkki setTexture-metodin käytöstä.

```
mesh.setTexture(renderer.loadedTextures[“textureName”]);
```

4.4 CameraObject-luokka ja luokan käyttö

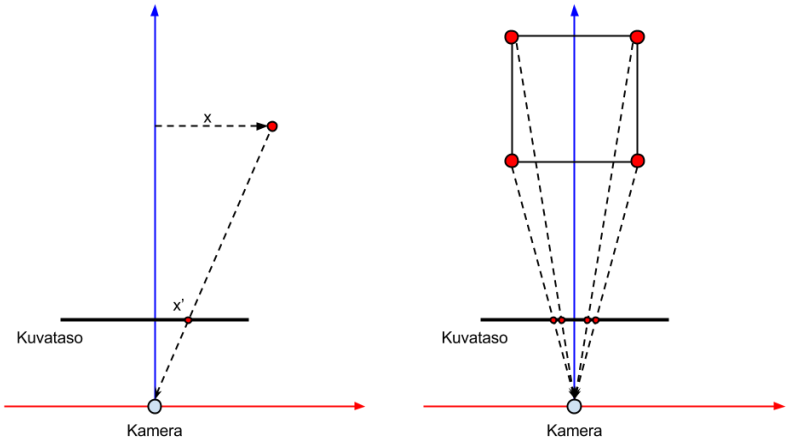
CameraObject-luokka (WEBGL_LIB.CameraObject) määrittelee kameran, jota voidaan liikuttaa ja kiertää maailmassa. Kamera määrittää myös perspektiivisen projektion toteuttavan matriisin, jonka avulla kameran näkemät 3D-mallien verteksit voidaan projektoida piirtoon käytettävän canvas-elementin muodostamaan 2D-avaruuteen, tai toisin sanoen ruudulle.

CameraObject-luokan oliota luodessa tulee parametreina antaa kameran määrittelemän projektiomatriisin muodostamiseen tarvittavat arvot. Projektiomatriisin muodostamiseen käytetään Math-nimiavaruudessa määritettyä getPerspective-

ProjMat4f-funktiota, joka ottaa parametreikseen näkökentän leveyden (engl. Field Of View) asteina, piirtopinnan leveyden ja korkeuden, sekä pienimmän ja suurimman etäisyyden, josta kappaleita projektoidaan matriisin avulla.

Perspektiiviprojektio on eniten käytetty tapa kaksiulotteisten kuvien muodostamiseen kolmiulotteisesta mallista. Perspektiiviprojektion tuottaa kuvan, jossa kauempana olevat kohteet näyttävät pienemmiltä kuin lähelle sijoitetut kohteet. (Puhakka 2008, 183–184.)

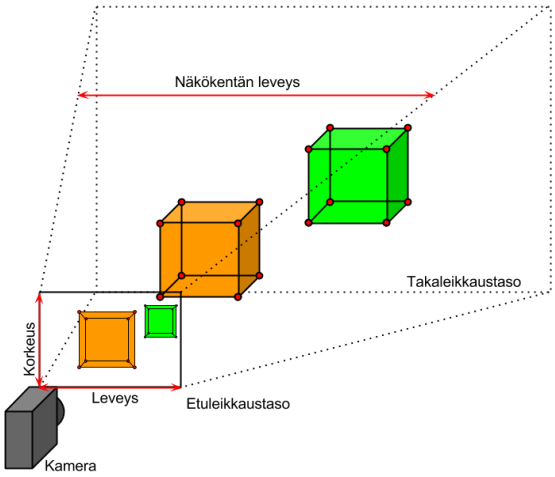
Projektion voidaan ajatella tapahtuvan siten, että avaruudessa olevan kappaleen vertekseistä kulkee suoria, jotka kaikki yhdistyvät lopulta kameraan, joka on projektion keskipisteessä. Nämä viivat kulkevat kuvatason läpi, ja tämä läpikulkukohta on verteksin lopullinen sijainti muodostetussa 2D-kuvassa. (Puhakka 2008, 183–184.) Kuvassa 25 pyritään hahmottamaan kuinka avaruudessa olevat verteksit sijoittuvat x-akselin suuntaisesti muodostettavaan kuvaan. Oikeanpuoleisessa esimerkissä huomataan, kuinka kamerasta kauempana olevat verteksit näkyvät pienempinä kuin lähempänä olevat verteksit.



Kuva 25: Projektiolla avaruudessa olevat verteksit sijoitetaan kaksiulotteiselle kuvatasolle, joka vastaa piirtämiseen käytettävää näyttöaluetta.

Perspektiiviselle projektiolle määritellyt arvot määrittelevät kartiomaisen alueen, jota kutsutaan näkökartioksi (engl. view frustum). Näkökartion sisällä olevat kappaleet heijastetaan piirtoalueelle, jonka sisällä olevat verteksit heijastettaan piirrettävän kuvan pinnalle. Kuvassa 26 havainnollistetaan muodostettua näkö-

kartiota, jonka sisälle sijoittuu kaksi kappaletta. Kaukaisemman kappaleen verteksit heijastuvat pienempinä piirtopinnalle kuin lähemmän kappaleen verteksit. Jos osa kappaleesta jää piirtopinnan ulkopuolelle, poistettaan se osa kappaleesta pois.

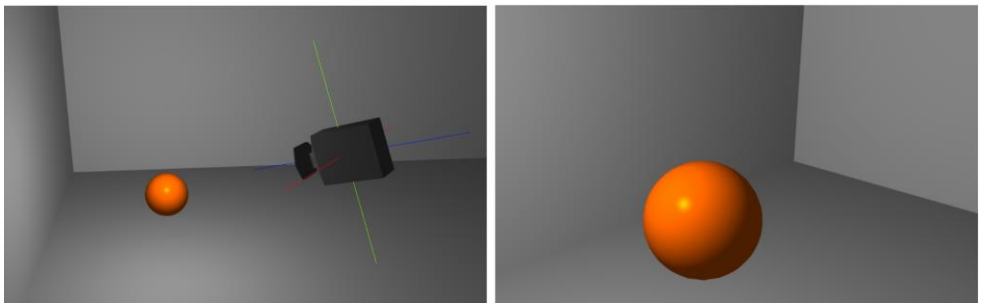


Kuva 26: Perspektiiviselle projektiolle määritetyt arvot muodostavat näkökartion.

Kirjaston CameraObject-luokalle määritetyt transformaatiot toimivat hieman erilailla verrattuna BaseObject-luokan määrittelemiin transformaatioihin. Piirrettäville 3D-malleille määritetyt transformaatiot esittävät kappaleen sijainnin maailma-avaruudessa. Kameran transformaatioiden avulla maailma-avaruuteen sijoitetut kappaleet siirretään kamera-avaruuteen, jossa kamera on avaruuden keskipisteessä, eli origossa. (Puhakka 2008, 173.) CameraObject-luokan määrittelemät transformaatiot tulee suorittaa kaikille maailmassa oleville kappaleille, jolloin ne siirtyvät kameran määrittelemään avaruuteen.

Kameran transformaatioiden perusta on määritellä kameralle vektorit, jotka määrittelevät kameran orientaation maailma-avaruudessa. Nämä vektorit määrittävät siis, mikä on kameran oikea, mikä on kameran ylös ja mikä on kameran eteen. Käytännössä vektorit kertovat siis kamera-avaruuden x-, y- ja z-akseleiden orientaation maailma-avaruudessa. (Puhakka 2008, 174–175.) Kuva 27 havainnollistaa kameran akseleiden orientaatiota maailma-avaruudessa, ja miltä maailmassa olevat mallit näyttävät kameran näkökulmasta. Vasemmassa

kuvassa kamerassa olevat viivat esittävät kameran määrittelemän kamera-avaruuden akseleita maailma-avaruudessa.



Kuva 27. Kameran orientaatio maailmassa ja maailma piirrettynä kameran näkökulmasta.

CameraObject-luokalle määritettyjen kiertojen esittämiseen käytetään luokalle määriteltyjä forward- ja up-jäsenmuuttujia, jotka esittävät kameran ”ylös”, ja kameran ”eteenpäin” suuntia maailma-avaruudessa. CameraObject-luokalle on määritetty rotateCamera-metodi määrittää näiden vektoreiden osoittamat suunnat, käyttämällä BaseObject-luokalta perittyjä worldOrientation- ja localOritentation-olioita paikallisten, ja maailma-avaruudessa suoritutettavien kiertojen esittämiseen. Metodi suorittaa kierrot käyttämällä BaseObject-luokalta perittyjä toimintoja kiertojen suorittamiseen paikallisten akseleiden ja maailman akseleiden ympäri, mutta kamera suorittaa nämä kierrot kamera-avaruudessa. Kun kierrot on määritetty, muodostaa metodi uuden kvaternion, joka sisältää worldOrientation- ja localOrientation-kvaternioihin määritetyt kameran kierrot yhdistettynä kvaternioiden kertolaskun avulla. Tämän jälkeen kameralle määritetään uudet arvot forward- ja up-vektoreihin kiertämällä vektoreita kameran kierrot määrittelevän kvaternion avulla. CameraObject-luokan rotateCamera-metodin toiminta nähdään alla olevassa koodissa.

```
WEBGL_LIB.CameraObject.prototype.rotateCamera = function(rotX, rotY){
    var baseZ = this.baseVectors.z.clone();
    baseZ.negate();
    var baseY = this.baseVectors.y.clone();
    this.rotateLocalXYZ(rotX, 0, 0);
    this.rotateWorldXYZ(0, rotY, 0);
```

```
    var rotQuat = this.transformations.worldOrientation.
                multQuaternion(this.transformations.localOrientation);
    this.forward = baseZ.rotateWithQuaternion(rotQuat);
    this.up = baseY.rotateWithQuaternion(rotQuat);
};
```

Kun kameralle on määritetty orientaatio up- ja forward-vektoreiden avulla, voidaan niiden arvoja käyttää rotaatiomatriisin muodostamiseen. CameraObject-luokka määrittä getCameraRotationMatrix-metodin. Metodi muodostaa kameran forward- ja up-vektoreiden avulla matriisin, jolla maailmassa olevat muut objektit voidaan kiertää CameraObject-olion määrittelemään kamera-avaruuteen. Matriisi muodostetaan kameran ”ylös”, ”oikealle” ja ”eteenpäin” osoittavien vektoreiden avulla sijoittamalla vektoreiden komponenttien arvot matriisille, joka palautetaan paluuarvona. Matriisiin muodostus perustuu kaavassa 38 (Ks. luku 2.2.4) tehtyyn huomioon matriisin eri arvojen vaikutuksesta tiettyihin akseleihin.

```
WEBGL_LIB.CameraObject.prototype.getCameraRotationMatrix = function(){
    var u = this.up.clone();
    var f = this.forward.clone();
    u.normalize();
    f.normalize();
    var r = this.getRight();

    var matArray = [
       r.x, u.x, f.x, 0.0,  // m11 m21 m31 m41
       r.y, u.y, f.y, 0.0,  // m12 m22 m32 m42
       r.z, u.z, f.z, 0.0,  // m13 m23 m33 m43
       0.0, 0.0, 0.0, 1.0   // m14 m24 m34 m44
    ];
    return new WEBGL_LIB.Math.Entities.Matrix4f(matArray);
};
```

CameraObject-oliolle voidaan määritellä translaatioita BaseObject-luokalta perittyjen toiminnallisuuksien avulla, tai käyttämällä kameraluokalle määritettyä

translateCamera-metodia, joka ottaa parametreina translaation suuntaa kuvaavan vektorin ja vektorin suunnassa kuljettavan matkan. Metodin avulla kameraa on kätevämpi siirtää esimerkiksi CameraObject-olion forward- ja up-vektoreiden määrittelemissä suunnissa, jolloin kameraa voidaan helposti liikuttaa eteenpäin.

CameraObject-luokan translaatiot voidaan muodostaa matriisille getCameraTranslationMatrix-metodilla. Metodi palauttaa matriisin, joka määrittää kameralle määritetyt translaatiot, mutta käyttää matriisin muodostamiseen käänteisiä arvoja. Tämä johtuu siitä, että kameran määrittelemät transformaatiot toteutetaan maailmassa oleville objekteille. Kaikki transformaatiot toteutetaan vastakkaisien suuntaan, jotta kamera vaikuttaisi liikkuvan kameran määrittelemään suuntaan. Esimerkiksi, jos kamera liikkuu z-akselin suunnassa positiiviseen suuntaan, siirtyvät maailman muut objektit, kuten mallit ja valot, negatiiviseen suuntaan. Alla olevassa koodissa nähdään getCameraTranslationMatrix-metodin toiminta, jossa metodi palauttaa kameran määrittelemän translaation matriisisin, jolla maailmassa olevia malleja tulee siirtää kameran liikkeen muodostamiseksi. Piirrossa, aktiivisen CameraObject-olion määrittelemät transformaatiot yhdistetään yhdeksi matriisiksi ja siirretään verteksivarjostimelle uniform-muuttujana.

```
WEBGL_LIB.CameraObject.prototype.getCameraTranslationMatrix =
                                                    function(){
return WEBGL_LIB.Math.getTranslationMat4f(
                            -this.transformations.location.x,
                            -this.transformations.location.y,
                            -this.transformations.location.z);
};
```

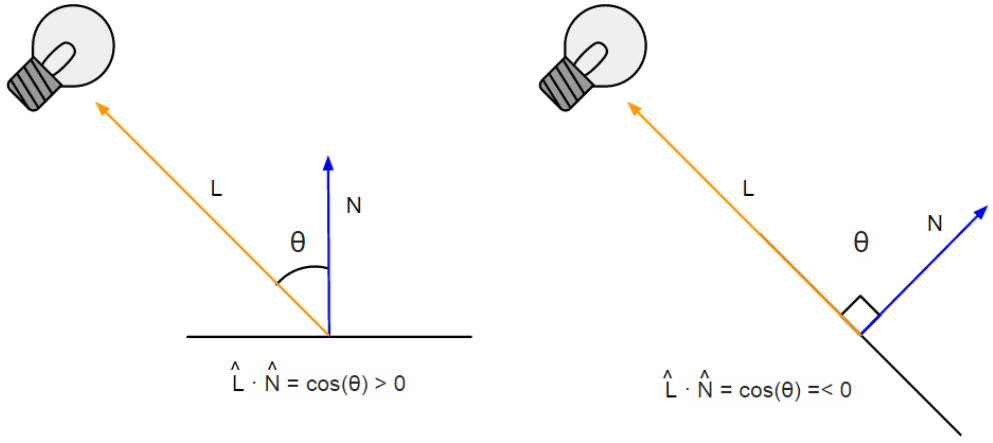
4.5 Valot kirjastossa

Kirjaston avulla voidaan toteuttaa piirrettävien objektien valaistusta neljällä eri tavalla. Kaksi tapaa liittyy piirtoon käytettävän maailman globaaliin valaistuksen määrittelyyn, ja kaksi liittyy maailmaan sijoitettavien piste- ja spottivalojen määrittelyyn. Piirrettäville MeshObject-luokan objekteille voidaan myös määritellä

pinnan kiiltoa simuloivia spekulaariheijastuksien asetuksia, jotka otetaan huomioon pinnan väritykseen vaikuttavia valaistuksia laskiessa. Kaikki kirjaston määrittelevät valaistukseen liittyvät arvot pinnan normaalivektoreita lukuun ottamatta siirretään laskettaviksi fragmenttivarjostimelle uniform-muuttujina. Varjostimessa piirrettävän kappaleen jokaiseen fragmenttiin vaikuttava valaistus lasketaan, ja se muokkaa fragmentin lopullista väritystä.

4.5.1 Pinnan heijastukset ja globaalit valot

Diffuusiheijastumisella tarkoitetaan valaistusta pinnasta heijastuvaa valoa, jonka arvo on sama riippumatta siitä, mistä suunnasta pintaa katsotaan (Puhakka 2008, 233). Kun kyseessä on valonlähde, joka määrittelee valolle kulkusuunnan, voidaan pinnalle kohdistuvan valaistuksen määrä laskea käyttämällä apuna pinnan normaalivektoreita. Kuvan 28 mukaisesti, pinnan diffuusiheijastus voidaan määritellä pinnan normaalivektorin ja valon kulkusuuntaa kuvaavan vektorin välisen kulman avulla. Kulma voidaan laskea kätevästi laskemalla pistetulo vektoreiden välillä. Pistetulon suorituksessa, valon kulkusuuntaa pinnasta valolle esittävän vektorin ja pinnan normaalivektorin tulee olla normalisoituja, jolloin niiden pituus on 1. Tuloksena saadaan luku, joka vastaa vektoreiden välisen kulman kosinia, eli arvo vaihtelee välillä [-1, 1]. Tämä arvo määrittää suoraan, kuinka paljon valoa pinta heijastaa. Jos valoa heijastetaan ja tuloksena saatavan arvon suuruus suurempi kuin 0, on vektoreiden välinen kulma pienempi kuin 90 astetta. Kun pistetulon suuruus yhtä suuri, tai pienempi kuin nolla, pinta ei heijasta valoa, eli vektoreiden välinen kulma on vähintään 90 astetta.



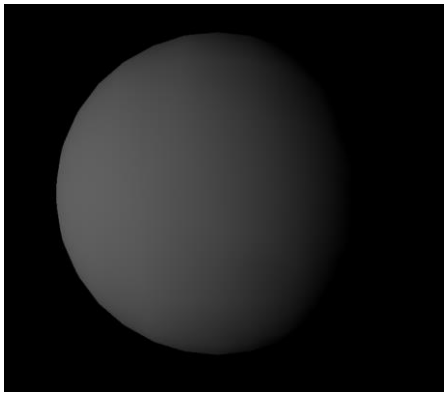
Kuva 28. Pinnasta heijastuvan diffuusiheijastuksen määrää määritetään pinnan normaalivektorin ja valonsäteen kulkusuuntaa esittävän vektorin välisen kulman avulla.

Kirjastoon määritetyllä fragmenttivarjostimella valojen tuottaman diffuusivalaistuksen arvo pinnalle voidaan laskea varjostimen lähdekoodiin määritetyllä calculateDiffuse-funktiolla. Funktio ottaa parametreina valon värin määrittelevän vektorin, valon kulkusuunnan määrittelevän vektorin ja valolle määritetyn heikentymisen liukulukuarvona. Funktio palauttaa valaistuksen määrää kuvaavan vektorin paluuarvona. Funktio hakee fragmentille verteksivarjostimelta lähetetyn normaalivektorin arvon vNormalVector-varying muuttujalta. Funktio laskee yksikköpituuteen muutettujen fragmentin normaalivektorin ja valon kulkusuunnan välisen kulman pistetulon avulla. Pistetulon tulos määrittää valon luoman diffuusi heijastuksen määrän fragmentille. Kulman määrittelyyn käytetään apuna max-funktiota, jolla varmistetaan, ettei arvo jää alle nollan, joka voisi vähentää jo olemassa olevan valon määrää. Lopuksi valon väriä esittävä vektori kerrotaan lasketun kulman ja valon hälventymistä määrittävän arvon kanssa. Valolle tapahtuva hälventyminen lasketaan erikseen ennen funktion kutsua, joten arvo muuttuu valon tyypistä ja valon ominaisuuksista riippuen. Alla olevassa koodissa nähdään fragmentille kohdistuvan diffuusiheijastuksen määrän laskeva funktio.

```
vec3 calculateDiffuse(vec3 lightColor, vec3 lightDirection,
                                        float attenuation){
```

```
    vec3 normal = normalize(vNormalVector);
    vec3 normLDir = normalize(lightDirection);
    float angle = max(dot(normLDir, normal), 0.0);
    return lightColor * angle * attenuation;
}
```

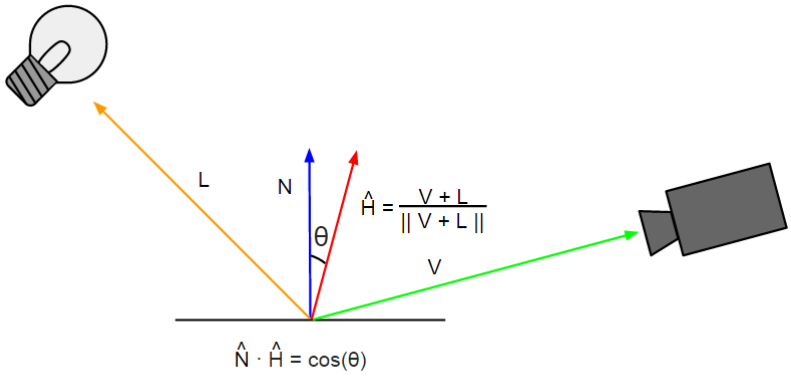
Funktion muodostama diffuusiheijastus muodostaa kuvan 29 kaltaisen heijastuksen pinnalle. Pinta vaikuttaa mattamaiselta, ja diffuusiheijastuksen määrä heikkenee, ja katoaa lopulta kokonaan, kun kappaleen pinnan normaalivektorit eivät osoita valonlähteen suuntaan.



Kuva 29. Esimerkki calculateDiffuse-funktion muodostamasta diffuusiheijastuksesta suunnatulle valolle.

Spekulaariheijastuksilla voidaan luoda piirrettävien kappaleiden pintoihin kiiltoa, joka on valaisevaa valonlähdettä heijastavia vääristyneitä ja sumentuneita muotoja piirrettävässä pinnassa. Kiillolla voidaan luoda vaikutelma pinnan materiaalista, kuten muovisesta, metallisesta tai lasisesta pinnasta. (Puhakka 2008, 237.)

Kirjasto käyttää kappaleen spekulaarien heijastuksien määrittelyyn Blinn-Phong heijastusmallia. Blinn-Phong heijastusmallissa heijastus määritellään laskemalla pistetulo normaalivektorin ja puolivektorin (engl. half vector) välillä, jotka ovat molemmat normalisoituja. Kuten kuvassa 30 esitetään, puolivektori H määritetään laskemalla yhteen valon kulkusuuntaa pinnalta valonlähteelle kuvaava vektori L ja katselusuuntaa pinnalta kameralle ilmaiseva vektori V, ja normalisoimalla yhteenlaskun tulosvektori. (Dykhta 2011.)



Kuva 30. Spekulaariheijastuksen määrittely Blinn-Phong heijastusmallilla.

Kun puolivektorin ja pinnan normaalin välinen kulma on selvitetty, muodostetaan spekulaariheijastuksen heijastusefektin suuruus korottamalla pistetulon tulos potenssiin pinnan kiiltävyyttä kuvaavalla arvolla. Mitä suurempi arvo potenssina on, sitä pienempiä ja terävämpiä kiiltoheijastukset ovat. (Dykhta 2011.) Tämä on esitettynä kaavassa 60.

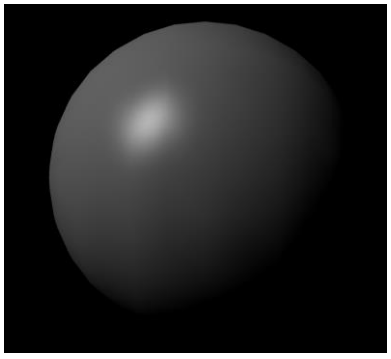
$$(\hat{\boldsymbol{n}} \cdot \hat{\mathbf{h}})^{s} \tag{60}$$

Kirjaston käyttämälle fragmenttivarjostimelle on varjostimen lähdekoodissa määritelty calculateSpecular-funktio, jolla voidaan laskea valosta syntyvän pinnalle muodostuvan spekulaariheijastuksen määrä sillä hetkellä käsiteltävälle fragmentille. Funktio ottaa parametreina valon värin, valon kulkusuunnan fragmentilta valonlähteelle ja valolle määritetyn hälventymisen. Ensin normaalivektorin ja valon suunnan pistetulon avulla tarkistetaan, että pinnalle kohdistuu valaistusta. Jos pinnalle heijastuu valoa, muodostetaan fragmentin ja kameran välinen vektori, jolla tiedetään mistä suunnasta fragmenttia katsotaan. Kun katselusuunta tiedetään, muodostetaan puolivektori, jota käytetään katselukulmasta muodostuvan spekulaariheijastuksen määrittämiseen suorittamalla pistetulo fragmentin normaalivektorin kanssa. Pistetulon tulos nostetaan potenssiin pinnalle asetetun spekulaarieksponentin avulla kaavan 60 mukaisesti. Lopuksi pinnalle asetetun spekulaariheijastukselle asetettu väri kerrotaan valon värin, lasketun spekulaariheijastusefektin ja valon hälventymistä ilmaisevan arvon kanssa. Näistä arvoista tuloksena saatu spekulaariaheijastusta esittävä vektori pa-

lautettaan paluuarvona. Fragmentin spekulaariheijastuksen määrittelevä GLSL-kielinen funktio nähdään alla olevassa koodissa.

```
vec3 calculateSpecular(vec3 lightColor, vec3 lightDirection,
float attenuation){
    float specularEffect = 0.0;
    vec3 normLDir = normalize(lightDirection);
    vec3 normal = normalize(vNormalVector);
    float diffuse = max(dot(normLDir, normal), 0.0);
    if(diffuse > 0.0){
        vec3 viewVector = normalize(uCameraPosition - vFragPosition);
        vec3 halfAngle = normalize(normLDir + viewVector);
        float specAngle = max(dot(normal, halfAngle), 0.0);
        specularEffect = pow(specAngle, uSpecular);
    }
    return uSpecularColor * lightColor * specularEffect *
                                          diffuse * attenuation;
}
```

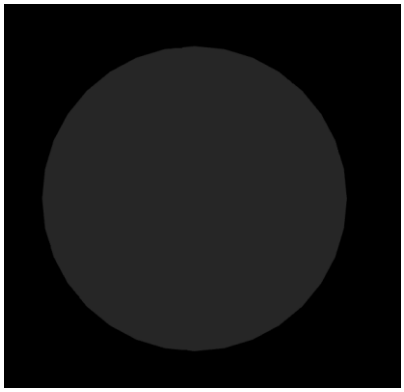
Kuvassa 31 nähdään calculateSpecular-funktion muodostama spekulaariheijastus suunnatulle valolle. Spekulaariheijastus ilmenee vaaleana alueena pinnassa, joka pyrkii osoittamaan valon tulosuuntaan.



Kuva 31. Spekulaariheijastus ilmenee vaaleampana alueena, joka osoittaa valon suuntaa.

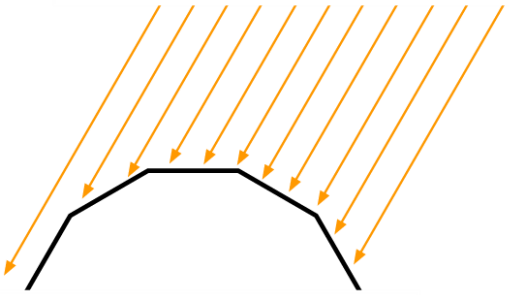
Renderer-luokan luomille maailmoille voidaan kirjastossa määritellä kaksi eri globaalia valaistusta: ambienttivalaistus ja suunnattu valaistus. Näitä kutsutaan globaaleiksi valaistuksiksi, koska ne ovat läsnä jokaiselle maailmaan määritetylle MeshObject-oliolle, olipa niiden sijainti maailmassa mikä tahansa.

Ambienttivalaistus on hyvin yksinkertainen valaistus, millä määritetään pinnoille muodostuvan diffuusiheijastuksen vähimmäismäärä, riippumatta pinnan orientaatioista ja normaalivektoreista. Ambienttivalo tuottaa tasaisesti valaistuja, varjottomia pintoja. Ambienttivalaistuksen avulla suunnatuilta valoilta varjoon jääneet osuudet pinnoista eivät jää täysin mustiksi pinnoiksi. (Puhakka 2008, 233.) Kuvassa 32 nähdään ambienttivalaistuksen tuottama tulos, kun kappaleen pinnalle ei muodostu muuta valaistusta.



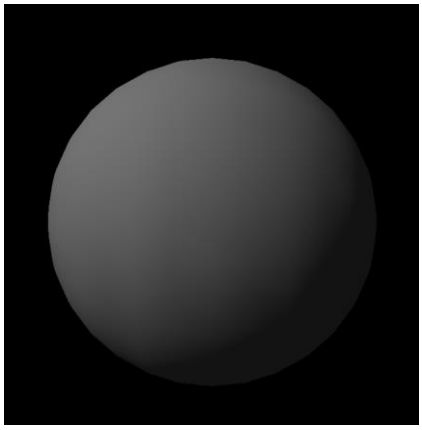
Kuva 32. Mallin valaistus kirjastolla, kun käytössä on vain ambienttivalaistus

Maailmaan voidaan myös määritellä suunnattu valo, joka on läsnä kaikille pinnoille maailmassa, mutta valolle on määriteltävä suunta, josta valo tulee. Tällaisella globaalilla suunnatulla valolla voidaan kätevästi mallintaa suuren valontähteen tuottamaa valaistusta maailmaan, kuten auringon tai kuun valoa. Suunnattu valo kulkee pinnalle aina samansuuntaisesti kuvan 33 mukaisella tavalla.



Kuva 33. Suunnattu valo tulee pinnalle aina samasta suunnasta

Suunnattu valo määritetään käytettäväksi piirrettävään maailmaan määrittelemällä valolle väri, ja suunta, jonka suuntaisesti suunnatun valon säteet kulkevat maailmassa. Kuvassa 34 nähdään mallille muodostettu valaistus, kun maailmaan on määritelty ambientti ja suunnattu valaistus.



Kuva 34. Maailman määritetyn suunnatun valon tuottama diffuusiheijastus yhdistettynä ambienttivalaistuksen tuottamaan vähimmäisdiffuusiheijastuksen kanssa.

Kuvan 34 mukaiset ambienttivalaistus ja suunnattu valaistus piirtämiseen käytettävällä maailmalle voidaan määritellä sijoittamalla alla olevan koodin mukaiset arvot Renderer-luokan aktiiviseen maailma-objektiin. Ambienttivalon ja suunnatun valon värit määritellään sijoittamalla RGB-värejä esittävät Vector3f-oliot maailman ambientLight- ja directionalLightColor-muuttujiin. Suunnatun valon kulkusuunta maailmassa määritetään directionalLightDir-muuttujaan asetetulla Vector3f-oliolla, joka on valon kulkusuuntaa kuvaava vektori.

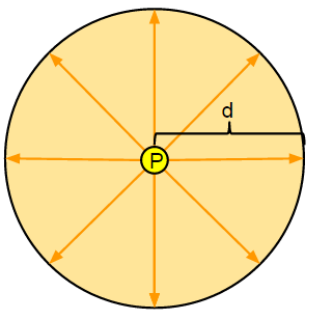
```
var ambientColor = new WEBGL_LIB.Math.Entities.Vector3f(
```

```
                                                    0.1, 0.1, 0.1);
var dirColor = new WEBGL_LIB.Math.Entities.Vector3f(
                                                    0.5, 0.5, 0.5);
var dirDirection = new WEBGL_LIB.Math.Entities.Vector3f(
                                                   -1.0, 1.0, 1.0);
renderer.activeWorld.ambientLight = ambientColor;
renderer.activeWorld.directionalLightColor = dirColor;
renderer.activeWorld.directionalLightDir = dirDirection;
```

Nämä arvot siirtyvät uniform-muuttujina fragmenttivarjostimelle, jossa niiden muodostama valaistus yhdistetään fragmentille muodostuneiden paikallisten valojen tuottaman valaistuksen kanssa. Paikallisia valoja kirjastossa ovat pistevalot ja spottivalot.

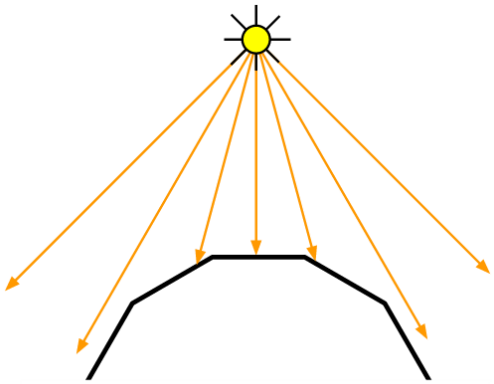
4.5.2 Pistevalot ja spottivalot

Pistevalo on paikallinen valo, koska sille määritellään sijainti maailmassa, josta se heijastaa valoa ympärillensä maailmassa. Pistevalon heijastaa valoa pallon muotoisena alueena ympärilleen kuvan 35 mukaisesti. Valon sijainti, joka kuvassa esitetään pisteenä P, määrittää alueen keskipisteen, ja valon maksimikantama määrittää alueen säteen.



Kuva 35. Pistevalon arvot muodostavat pallomaisen alueen, jonka sisällä valoa esiintyy.

Pinnalle heijastuvan valon määrä riippuu pistevalon sijainnista maailmassa pintaan nähden. Kuva 36 kuvaa pistevalon heijastamaan valon kulkua pinnalle.

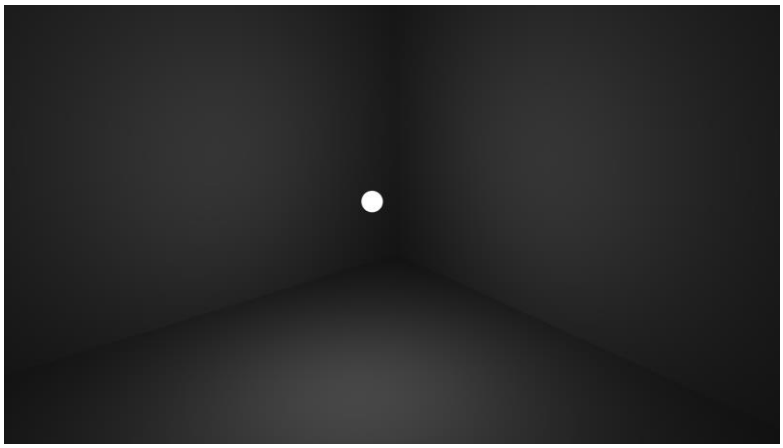


Kuva 36. Pistevalo heijastaa valoa sijainnistaan pinnoille.

Kirjastossa pistevalo määritellään PointLightObject-luokan avulla. Luokan alustuksen yhteydessä pistevalolle määritetään pistevalon sijainti maailmassa, pistevalon heijastaman valon väri ja pistevalon heijastaman valon maksimikantama avaruudessa. Kun PointLightObject-olio on luotu, voidaan se lisätä piirrettävään maailmaan maailma-objektin määrittelemällä addPointLight-funktiolla, jonka jälkeen valoa käytetään piirrossa pintojen valaisuun.

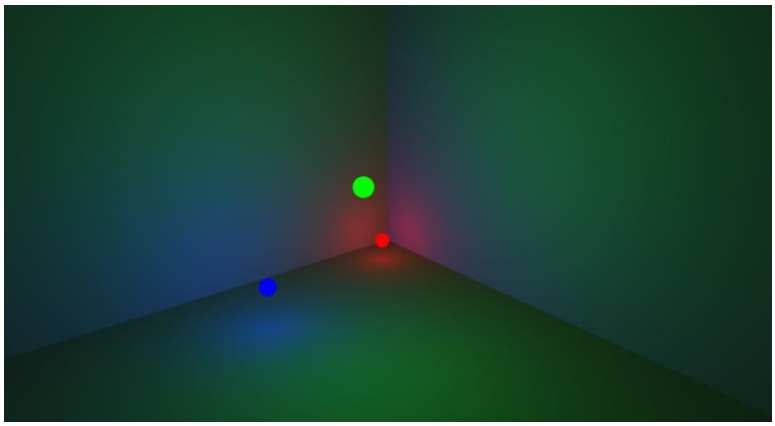
```
var pointLight = new WEBGL_LIB.PointLightObject(
            new WEBGL_LIB.Math.Entities.Vector3f(-2, 2, 0),
            new WEBGL_LIB.Math.Entities.Vector3f(0.5, 0.5, 0.5),
            20
);
renderer.activeWorld.addPointLight(pointLight);
```

Kuvassa 37 nähdään yksittäisen pistevalon muodostamat heijastukset ympärillä oleville pinnoille. Pistevalon sijainti nähdään valkoisena pisteenä kuvassa.



Kuva 37. Pistevalon muodostama valaistus ympärillä oleville pinnoille.

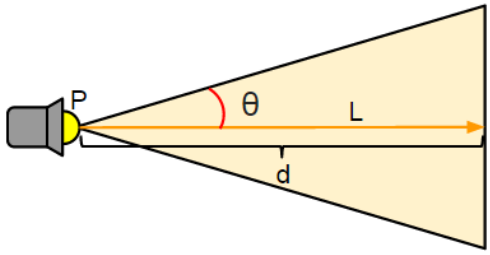
Kirjasto mahdollistaa myös useamman pistevalon määrittelyn kerrallaan kuvan 38 mukaisesti, jossa pistevaloille on asetettu eri värit. Kuvasta nähdään kuinka valojen tuottamat värit sekoittuvat toisiinsa.



Kuva 38. Kirjasto mahdollistaa useamman pistevalon määrittelyn, ja ne kaikki vaikuttavat yhdessä piirrettävän pinnan lopulliseen väritykseen.

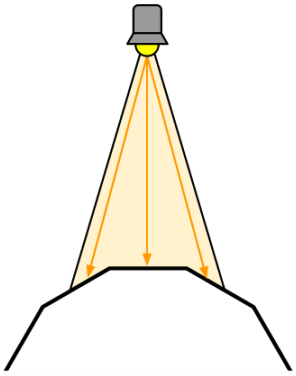
Spottivalot eroavat pistevaloista siten, että spottivalo heijastaa valonsa kartiomaisena alueena tiettyyn suuntaan, eikä kaikkialle ympärillensä, niin kuin pistevalo. Samoin kuin pistevalolle, määritetään spottivalolle sijainti, valonväri ja maksimimatka, jonka valo voi kulkea valonlähteestä avaruudessa. Näiden lisäksi, spottivalolle määritetään suunta, johon valokeilana heijastettava valo kohdistetaan, kulma, joka määrittää minkä kokoisena valokeila heijastetaan, ja eksponentti, joka määrittää miten paljon heijastetun valon voimakkuus heikkenee valokeilan reunoilla. Kuvassa 39 nähdään spottivalon arvojen muodostama valokeila. Piste P esittää spottivalon sijaintia maailmassa, vektori L valokeilan osoit-

tamaa suuntaa, kulma θ määrittää spottivalon valokeilan suuruuden, ja arvo d määrittää valokeilan kulkeman matkan.



Kuva 39. Spottivalolle määritetyt arvot määrittelevät kartion muotoisen valokeilan.

Kuvassa 40 nähdään hahmotelma spottivalon muodostamasta valokeilasta, ja valon kulkusuunnasta kappaleen pinnalle. Kuten pistevalon kanssa, valon kulkusuunta pinnalle määrittyy pistevalon sijainnin ja valaistavan pisteen sijainnin välisestä vektorista, mutta valoa voi kulkea vain spottivalon määrittelemän kulman suuruisella alueella.



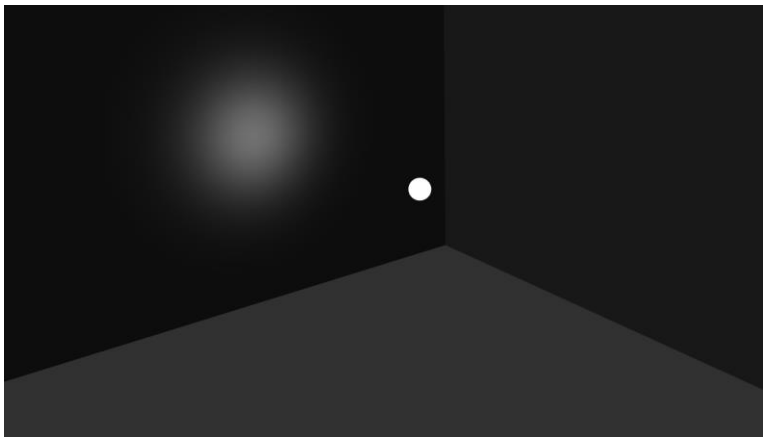
Kuva 40. Spottivalon muodostaman valon kulku valaistavalle pinnalle.

Spottivalojen määrittelyä varten kirjastoon on luotu SpotlightObject-luokka. Luokan olion alustuksen yhteydessä oliolle määritetään valon sijainti avaruudessa, heijastetun valon väri, heijastetun valon suurin kulkumatka avaruudessa, pistevalon osoittama sijainti avaruudessa, pistevalon valokeilan kulma ja pistevalon valokeilan reunojen valomäärän hälventyminen, alla olevan esimerkkikoodin mukaisesti. Spottivalot voidaan lisätä käytettäväksi maailmaan käyttämällä

addSpotlight-funktiota, joka lisää spottivalon käytettäväksi funktiota kutsuvaan maailma-objektiin. Maailmaan lisättyjen SpotlightObject-olioiden arvot siirtyvät uniform-muuttujina fragmenttivarjostimelle, jossa valojen määrittelemiä arvoja käytetään pinnalle syntyvän valaistuksen selvittämiseen.

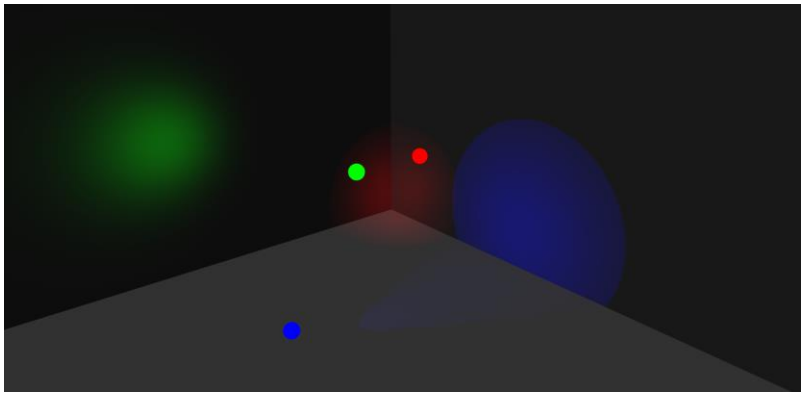
```
var spotlightPos = new WEBGL_LIB.Math.Entities.Vector3f(1, 1, 0);
var spotlightColor = new WEBGL_LIB.Math.Entities.Vector3f(0.5, 0.5,
                                                                0.5);
var spotlightDistance = 50;
var spotlightTarget = new WEBGL_LIB.Math.Entities.Vector3f(0, 0, 0);
var spotlightAngle = Math.PI/32;
var spotlightHardness = 1000;

var spotlight = new WEBGL_LIB.SpotlightObject(
                    spotlightPos,
                    spotlightColor,
                    spotlightDistance,
                    spotlightTarget,
                    spotlightAngle,
                    spotlightHardness,
);
renderer.addSpotlight(spotlight);
```

Kuvassa 41 nähdään yksittäisellä spottivalolla muodostettu valaistus. Valkoinen pallo kuvassa esittää valolähteen sijaintia avaruudessa, josta valokeila kohdistetaan vieressä olevaa seinää vasten.



Kuva 41. Yksittäisen spottivalo joka muodostaa valkoisen valaistuksen seinälle.

Kirjasto mahdollistaa myös useamman spottivalon määrittelemisen kerralla. Kuvassa 42 nähdään kolmen eri spottivalon muodostamat valaistukset. Kuva havainnollistaa myös spottivalolle asetetun valokeilan reunojen hälventymisen vaikutusta spottivalon reunojen terävyyteen.



Kuva 42. Kirjasto mahdollistaa useiden erilaisten spottivalojen määrittelyn.

4.5.3 Valaistuksen määrittely fragmenttivarjostimessa

Kaikkien valaistukseen käytettävät arvot verteksien normaalivektoreita lukuun ottamatta siirtyvät uniform-muuttujina fragmenttivarjostimelle. Kirjaston käyttämälle fragmenttivarjostimelle on määritelty uniform-muuttujat globaalien valaistuksien, paikallisten valaistuksen ja pinnan spekulaariheijastusten arvojen määrittelyyn.

Globaalin valaistuksen asetukset määritellään kolmella uniform-muuttujalla, joille lähetetään piirtoon käytettävän maailman ambienttivalon väri, ja suunnatun valon väri, sekä valon kulkusuunta. Alla nähdään näiden uniform-muuttujien määrittelyt varjostimen lähdekoodissa.

```
// Global lighting uniforms
uniform vec3 uDirLightColor;
uniform vec3 uDirLightDirection;
uniform vec3 uAmbientLight;
```

Pistevaloja varten on määrittely rakenne, jota käytetään sisällyttämään yhden maailmaan lisätyn PointLightObject-olion arvot. Jotta kirjastolla voidaan määritellä useampi pistevalo, on uniform-muuttuja määritelty taulukoksi, joka voi sisältää MAX_POINT_LIGHTS vakioarvon verran alkioita. Jokainen alkio on PointLight-rakenteen mukainen. MAX_POINT_LIGHTS määrittää maksimivalojen määrään piirrossa, ja sen arvoon voidaan vaikuttaa Renderer-luokan avulla. Fragmenttivarjostimelle annetaan myös uniform-muuttuja, joka määrittää käytössä olevien valojen määrän. Muuttujan arvo muuttuu sen perusteella, kuinka monta PointLightObject-oliota maailmaan on lisätty.

```
// Point lights struct/uniforms
uniform int uPointLightCount;
uniform struct PointLight {
      vec3 position;
      vec3 color;
      float length;
 } uPointLights[MAX_POINT_LIGHTS];
```

Spottivaloille määritetään samantapainen rakenteesta muodostettu uniform-muuttuja, joka voi ottaa useamman SpotlightObject-olion arvot vastaan määritettyyn taulukkoon. Erona pistevalon rakenteeseen nähden on spottivalolle määritettyjen muiden arvojen mukana oleminen. Maailmaan lisättyjen SpotlightObject-olioiden määrä annetaan myös uniform-muuttujana.

```
// Spotlight struct/uniforms
uniform int uSpotlightCount;
uniform struct Spotlight {
     vec3 position;
     vec3 target;
     vec3 color;
     float angle;
     float length;
     float hardness;
} uSpotlights[MAX_SPOTLIGHTS];
```

Valojen luomat heijastukset fragmenteille määritetään fragmenttivarjostimen pääfunktiossa, jossa valojen määrittely aloitetaan määrittämällä globaalien valaistusten vaikutus. Globaalin suunnatun valaistuksen muodostamat diffuusi- ja spekulaariheijastukset määritetään käyttämällä calculateDiffuse- ja calculateSpecular-funktioita. Suunnatun valaistuksen uniform-muuttujien arvot syötetään suoraan funktioille parametreina. Suunnatun valon suunta asetetaan käänteiseksi, koska funktiot olettavat valon kulkusuunnan olevan fragmentista valon lähteeseen päin. Valon hälventymiseksi määritetään arvo 1.0, koska globaalilla suunnatulla valolla ei tule olla hälventymistä. Kun suunnatun valon fragmentille luomat diffuusi- ja spekulaariheijastukset on muodostettu vektoreihin, yhdistetään niiden vaikutukset laskemalla vektorit yhteen. Lopullinen globaali valaistus muodostuu ambienttivalaistuksen määrittelevän vektorin ja suunnatun valaistuksen määrittelevien arvojen yhteenlaskulla. Alla katkelma varjostinkoodista, joka muodostaa fragmentin globaalin valaistuksen.

```
vec3 dirDiffuse = calculateDiffuse(uDirLightColor, -uDirLightDirection,
                                                          1.0);
vec3 dirSpecular = calculateSpecular(uDirLightColor,-uDirLightDirection,
                                                          1.0);
vec3 dirLighting = dirDiffuse + dirSpecular;
vec3 globalLighting = uAmbientLight + dirLighting;
```

Koska maailmassa voi olla useita piste- ja spottivaloja, lasketaan kaikkien niiden vaikutus yksitellen yhteen vektoriin, joka määrittää kaikkien fragmentille kohdistuneiden paikallisien valaistuksien yhteisvaikutuksen. Paikalliset valaistukset käydään yksitellen läpi silmukoissa, jotka kutsuvat applyPointLight- ja applySpotlight-funktioita.

Alla olevassa koodissa muodostetaan yksittäisen PointLightObject-olion fragmentille tuottama valaistus. Pistevalojen tuottaman valaistuksen määrittelyyn käytetään varjostimen applyPointLight –funktiota, joka ottaa parametrina pistevaloa esittävän PointLight-rakenteen. Oletusarvoisesti mitään valaistusta ei muodostu, joten palautusarvona käytettävä vektorimuuttuja alustetaan nolla-arvoilla. Tämän jälkeen lasketaan fragmentilta pistevalon lähteeseen kulkeva vektori lightRay-muuttujaan, josta otetaan ensin pinnan ja valonlähteen välinen etäisyys talteen distance-muuttujaan. Tämän jälkeen valoa esittävä vektori voidaan normalisoida. Jos valon etäisyys valaistavasta fragmentista on suurempi, kuin valolle määritetty maksimivalaistusetäisyys, ei valaistusta voi tapahtua, jolloin palautettavan lightingValue-muuttujan arvoa ei muuteta. Jos valo on kuitenkin valonkantaman sisällä, määritetään valon hälventyminen valon etäisyydestä riippuen valon maksimikantaman avulla. Mitä pienempi arvo hälventymiselle määräytyy, sitä vähemmän valoa valaistavalle fragmentille tuotetaan. Tämän jälkeen pistevalon tuottama diffuusi- ja sekulaariheijastukset määritetään calculateDiffuse- ja calculateSpecular-funktioiden avulla, ja niiden tulokset yhdistetään palautettavaan muuttujaan. Palautettu arvo yhdistetään muiden pistevalojen tuottamaan valaistukseen.

```
vec3 applyPointLight(PointLight pLight){
    vec3 lightingValue = vec3(0.0, 0.0, 0.0);
    vec3 lightRay = pLight.position - vFragPosition;
    float distance = length(lightRay);
    lightRay = normalize(lightRay);
    if(distance < pLight.length){
        float attenuation = clamp(1.0 - distance / pLight.length, 0.0,
                                                                1.0);
        attenuation *= attenuation;
```

```
        vec3 diffuse = calculateDiffuse(pLight.color, lightRay,
                                                    attenuation);
        vec3 specular = calculateSpecular(pLight.color, lightRay,
                                                      attenuation);
        lightingValue = diffuse + specular;
    }
    return lightingValue;
}
```

SpotlightObject-olioiden tuottama valaistus lasketaan alla olevassa koodissa esitetyllä applySpotlight-funktiolla. Funktio ottaa vastaan yhden spottivaloa esittävän Spotlight-rakentteen, jonka arvoja käytetään valaistuksen määrittelyyn. Spottivalon valaistuksen määrittely alkaa samalla tavalla kuin pistevalon määrittely, mutta funktio määrittelee myös spottivalon valokeilan osoittaman suunnan lDirection-muuttujaan. Muuttujan arvo lasketaan spottivalon valokeilan kohdistaman kohdesijainnin ja spottivalon oman sijainnin avulla. Jos fragmenttiin osuva valo spottivalosta muodostaa valoa, lasketaan sijoittuuko fragmentilta valoon kulkeva vektori spottivalon määrittelemän valokeilan sisälle. Tämä tapahtuu vertaamalla valokeilan keskellä kulkevaa suuntavektorin ja valonsäteen välistä kulmaa spottivalon määrittelemään kulmaan, joka on suurin kulma, jossa valo voi kulkea ollakseen valokeilan sisällä. Jos pinnalle kulkeva valonsäde on valokeilan sisällä, ja se on valokeilalle määritetyn etäisyyden sisällä, voidaan laskea valon tuottamat heijastukset fragmentille. Tässä vaiheessa määritetään valon hälventyminen valokeilan reunoilla nostamalla valokeilan suunnan ja valonsäteen välinen pistetulo potenssiin hälventymisen määrittelevällä valokeilan hardness-muuttujalla. Mitä suurempi arvo hardness-muuttujassa on, sitä enemmän valo hälventyy valokeilan reunoilla. Tämä arvo yhdistetään valon kulkumatkalle määritettyyn hälventymiseen, ja spottivalon muodostamat diffusi- ja spekulaariheijastukset lasketaan määritetyillä arvoilla. Lopuksi spottivalon muodostamat diffuusi- ja spekulaariheijastukset yhdistetään ja palautetaan paluuarvona yhdistettäväksi muiden piste- ja spottivalojen määrittelemien valaistusarvojen kanssa.

```
vec3 applySpotlight(Spotlight sLight){
    vec3 lightingValue = vec3(0.0, 0.0, 0.0);
```

```
    vec3 lDirection = normalize(sLight.target - sLight.position);
    vec3 lightRay = sLight.position - vFragPosition;
    float distance = length(lightRay);
    float lightValue = max(dot(normalize(vNormalVector),
                               normalize(lightRay)), 0.0);
    if(lightValue > 0.0){
        float spotEffect = dot(lDirection,
                               normalize(-lightRay));
        if(spotEffect > cos(sLight.angle)){
            if(distance < sLight.length){
                spotEffect = max(pow(spotEffect, sLight.hardness), 0.0);
                float attenuation = clamp(1.0 - distance/sLight.length,
                                          0.0, 1.0);
                vec3 diffuseLight = (sLight.color * lightValue *
                                     spotEffect * attenuation);
                vec3 specularLight = calculateSpecular(sLight.color,
                                                       lightRay,
                                                       attenuation);
                lightingValue = diffuseLight + specularLight;
            }
        }
    }
    return lightingValue;
}
```

Kun maailman globaalien ja paikallisten valojen määrittelemät heijastukset fragmentille on määritetty, yhdistetään niiden tuottamat valaistukset fragmentille määritettyyn väritykseen. Yhdistettyjen arvojen muodostama väri asetetaan fragmentin lopulliseksi väriksi. Valot vaikuttavat vain tekstuurin määrittämiin RGB-värien arvoihin, ja alpha-kanavan arvo säilytetään sellaisenaan seuraavan koodin mukaisesti.

```
gl_FragColor = vec4((surfaceColor.rgb * (globalLighting+localLighting)),
                                                       surfaceColor.a);
```

5 Kirjastolla luotu demosovellus

Liitteeseen 3 on lisätty esimerkki HTML-tiedostossa luodusta selainsovelluksesta, joka hyödyntää opinnäytetyössä luotua kirjastoa. Kirjasto on esimerkissä tallennettu tiedostonimellä webgl_lib.js, ja se liitetään käytettäväksi html-tiedostoon esimerkkisovelluksen alussa. Esimerkkisovelluksessa luodaan kuvan 43 mukainen canvas-elementti, jota päivitetään joka kerta, kun aikaisempi piirretty kuva on valmis canvas-elementin määrittelemän requestAnimationFrame-toiminnallisuuden avulla.

Kuva 43. Liitteessä 3 esitetyn esimerkkikoodin muodostama sovellus.

Luodussa sovelluksessa ladataan kaksi 3D-mallia kahdesta eri COLLADA -dokumentista. Ladattujen mallien normaalivektorit lasketaan uudestaan, minkä jälkeen niiden avulla muodostetaan kaksi MeshObject-oliota. Nämä oliot muodostavat kuvassa 43 näkyvät seinä- ja donitsigeometriat. Donitsigeometrian muodostavalle MeshObject-oliolle asetetaan Renderer-luokan lataama tekstuuri, ja sen pinnalle asetetaan spekulaarinen heijastus. Seinägeometrian muodostava MeshObject-olion pinnalle asetetaan yksittäinen väri, eikä se muodosta spekulaariheijastusta. Molemmille kappaleille määritetään erilaisia transformaatioita, ja donitsin muodostavaa MeshObject-oliota kierretään aina update-funktiossa, jolloin kappaletta kierretään aina kun uusi kuva piirretään. Sovelluk-

seen on määritelty myös PointLightObject- ja SpotLightObject-oliot, joita liikutetaan aina update-funktion kutsussa muuttamalla olioiden määrittelemiä arvoja pistevalon sijainnille ja spottivalon osoittamalle kohteelle. Kuvassa 43 liikettä ei voi tosin huomata.

6 Pohdinta

Opinnäytetyön tuloksena valmistettiin prototyyppi WebGL-rajapintaa hyödyntävästä grafiikka-kirjastosta, joka yksinkertaistaa 3D-grafiikkaa hyödyntävien sovelluksia luomista. Kirjaston avulla voidaan ladata Blender – sovelluksessa mallinnettuja malleja, jotka on tallennettu COLLADA- dokumenttiin. Kirjaston avulla voidaan myös ladata piirrettävien kappaleiden teksturointiin käytettäviä kuvia antamalla kuvaan johtava tiedostopolku. Ladatut mallit voivat määritellä mallin muodostavat verteksit, mallille määritetyt normaalivektorit ja tekstuurikoordinaatit. Ladattujen mallien avulla voidaan muodostaa piirrettäviä kappaleita, jotka määrittävät mallin käyttämän teksturiin, ja kappaleille voidaan asettaa transformaatiot, joilla mallein määrittelemiä tietoja voidaan muokata piirrettävässä kuvassa. Piirtoa varten voidaan määritellä useita erityyppisiä valaistusasetuksia, määrittelemällä globaaleja ambienttivaloja ja suunnattuja valoja, sekä määrittelemällä paikallisia valaistuksia muodostavia piste- ja spottivalo-objekteja. Piirrettäville kappaleille voidaan määritellä myös valaistuksen muodostamia heijastusefektejä kappaleen pinnalle.

Kirjaston toiminnallisuuksien toteuttamista varten tuli opinnäytetyötä tehdessä tutustua 3D-grafiikaan liittyvään matematiikkaan ja selvittää, millaisia merkityksiä esim. vektoreilla, matriiseilla ja kvaternioilla on 3D-grafiikassa, ja tämän tiedon avulla tuli kirjastoon kehittää joukko matemaattisia luokkia ja funktioita, jotka toimivat opitun teorian pohjalta.

Opinnäytetyö tutustui kattavasti eri WebGL-rajapinnan toiminnallisuuksiin, ja se selvitti, kuinka WebGL-rajapinta käsittelee piirrettävää tietoa, ja kuinka WebGL-

rajapinnan muodostamaan grafiikkaan voidaan vaikuttaa funktioilla, varjostinsovelluksilla ja niille lähetettävillä muuttujatiedoilla.

Tiedostojen purkamista varten opinnäytetyössä tutustuttiin COLLADA – dokumenttiin, ja siihen, kuinka COLLADA – dokumentti määrittelee siihen tallennettuja 3D-malleja. COLLADA – dokumentti ja sen rakenne ovat jo itsessään suuri aihepiiri, koska COLLADA -dokumentti voi määritellä hyvin laajasti erilaisia 3D-grafiikkaan liittyviä tietoja, malleista kokonaisiin valmiiksi määriteltyihin 3D-ympäristöihin. Myös COLLADA -dokumentista tietoa purkavan toiminnallisuuden määrittely vei suuren osan koodista, eikä sekään toimi vielä täysin ongelmitta.

Valmistettu kirjasto käsittelee siis paljon aiheita, joita alun perin oli tarkoitus opinnäytetyössä tutkia ja toteuttaa. Aiheet toimivat suurimmilta osin siten, miten niiden pitääkin, vaikka niiden toimintaa voisi muuttaa monipuolisemmaksi ja joidenkin kirjaston määrittelemien työkalujen toimintaa voisi uudelleen harkita, kuten Renderer-luokan toimimista resursseja kokoavana osana ja COLLADA – tiedostosta malleja purkavaa metodia, sekä matemaattisten luokkien sijaintia kirjastossa niiden hankalan nimiavaruusjoukon takia. Myös Varjot jäivät käsittelemättä työssä, koska niiden tekemiseen ei jäänyt aikaa enää opinnäytetyön valmistumiselle asetetulla aikataululla, joka oli asetettu keväälle 2015. Silti, kirjasto toteuttaa suurimman osan toiminnallisuuksista, joita alun perin oli tarkoitus toteuttaa.

Opinnäytetyötä varten koostettu teoria on hyvin suuri osa valmistettua työtä. Opinnäytetyötä varten kirjoitettua teoriaa on paljon, vaikkakin käsitellyt teoriat ovat kuitenkin aivan perustason asioita 3D-grafiikassa. Tästä voidaan päätellä jo kuinka laaja ja monimutkainen aihe 3D-grafiikka ja sen ohjelmointi voivat olla, muttei se silti ole mitään ylitsepääsemättömän vaikeaa, kunhan aiheelle antaa aikaa. Opinnäytetyön toivotaan antavan kattava katsaus työn aiheesta kiinnostuneelle 3D-grafiikan ohjelmointiin liittyvästä teoriasta ja käytännön toteutuksesta WebGL-rajapinnan avulla. Työssä pyrittiin olemaan mahdollisimman kuvaavia eri aiheiden teorioista, ja toteutetun kirjaston toiminnasta, joskin teorioita ja

toimintaa kuvaavat luvut voivat olla melko vaikeaselkoisia termistöiltään ja aiheen monimuotoisuuden vuoksi.

Opinnäytetyö on toiminut itse kehittäjälle hyvänä tapana oppia 3D-grafiikaan liittyvää teoriaa ja sen käytännön toteutusta, sekä opettanut yleisiä periaatteita grafiikka-rajapintojen toiminnasta. Vaikka toteutetulla kirjastolla ei ole suurta merkitystä työkaluna muille kehittäjille, voi sitä silti pitää hyvänä pohjana jatkokehitystä ajatellen. Prototyyppiasetteella toteutetun kirjaston käyttö ei ole välttämättä kovin itsestään selvää muille kuin itse kehittäjälle, mutta kehittämällä kirjaston toimintoja helppokäyttöisemmiksi, voitaisiin siitä saada aikaan jopa yleisempäänkin käyttöön tarkoitettu kirjasto. Kirjastolle voitaisiin kehittää lisää toiminnallisuuksia esim. varjojen, transformaatiohierarkioiden, animaatioiden, korkeuskarttojen ja normaalikarttojen käyttöön, jotka ovat monessa 3D-grafiikkaa käyttävässä sovelluksessa hyvin yleisiä toiminnallisuuksia. Tässä vaiheessa tuo vaihe ei kuitenkaan ole vielä lähellä. Kirjasto kehitettiin WebGL-rajapinnan 1.0 version avulla, mutta lähitulevaisuudessa Khronos Group julkaisee rajapinnasta uuden 2.0 version, joka tuo mukanaan joukon uusia toiminnallisuuksia.

Lähteet

Barens, M. & Finch, E. 2008. COLLADA – Digital Asset Schema Release 1.4.1. Khronos Group. https://www.khronos.org/files/collada_spec_1_4.pdf. 22.04.2015.

Cantor, D. & Jones, B. 2012. WebGL Beginners Guide. Birmingham: Packt Publishing Ltd.

Collada.org. 2008. COLLADA FAQ. Khronos Group. https://collada.org/mediawiki/index.php/COLLADA_FAQ. 21.04.2015.

confuted. 2015. Using Quaternion to Perform 3D rotations. Cprogramming.com. http://www.cprogramming.com/tutorial/3d/quaternions.html. 4.5.2015.

Danchilla, B. 2012. Beginning WebGL for HTML5. New York: Apress Media LLC.

Dunn, F. & Parberry, I. 2011. 3D Math Primer for Graphics and Game Development. Boca Raton: CRC Press Taylor and Francis Group.

Dykhta, I. 2011. Blinn-Phong reflection model in GLSL. Dykhta, I. http://www.sunandblackcat.com/tipFullView.php?l=eng&topicid=30. 4.5.2015.

Khronos Group. 2011. Getting Started. Khronos Group. https://www.khronos.org/webgl/wiki/Getting_Started. 17.4.2015.

Khronos Group. 2015a. OpenGL ES 2.0 for the Web. Khronos Group. https://www.khronos.org/webgl/. 17.4.2015.

Khronos Group. 2015b. WebGL Specification. Khronos Group. https://www.khronos.org/registry/webgl/specs/latest/1.0/. 17.04.2015.

Khronos Group. 2015c. 3D Asset Exchange Schema. Khronos Group. https://www.khronos.org/collada/. 21.04.2015.

Microsoft. 2015a. uniformMatrix4fv method. Microsoft. https://msdn.microsoft.com/en-us/library/ie/dn302458%28v=vs.85%29.aspx. 20.04.2015.

Microsoft. 2015b. uniform3fv method. Microsoft. https://msdn.microsoft.com/en-us/library/ie/dn302449%28v=vs.85%29.aspx. 20.04.2015.

OpenGL.org. 2015. Built-in Variable (GLSL). OpenGL.org. https://www.opengl.org/wiki/Built-in_Variable_%28GLSL%29. 19.04.2015.

Puhakka, A. 2008. 3D-grafiikka. Helsinki: Talentum Media.

Silicon Graphics. 2006a. glBindTexture. Khronos Group. https://www.khronos.org/opengles/sdk/docs/man/xhtml/glBindTexture.xml. 8.5.2015.

Silicon Graphics. 2006b. glActiveTexture. Khronos Group. https://www.khronos.org/opengles/sdk/docs/man/xhtml/glActiveTexture. 8.5.2015.

Silicon Graphics. 2006c. glTexParameter. Khronos Group. https://www.khronos.org/opengles/sdk/docs/man/xhtml/glTexParameter.xml. 8.5.2015.

Silicon Graphics. 2006d. glTexImage2D. Khronos Group. https://www.khronos.org/opengles/sdk/docs/man/xhtml/glTexImage2D.xml. 8.5.2015.

Wikibooks. 2012. OpenGL Programming/Shaders reference. Wikibooks. http://en.wikibooks.org/wiki/OpenGL_Programming/Shaders_reference. 19.04.2015.

Kirjaston luokkien kuvaukset

WEBGL_LIB.Renderer
Renderer-luokka on vastuussa kommunikoinnista WebGL-rajapinnan kanssa, ja se luo HTML5-elementin, josta kommunikaatioon tarvittava WebGLRendering-Context-olio saadaan käyttöön. Renderer-luokkaa käytetään 3D-mallien lataamiseen COLLADA -tiedostosta, sekä ladattujen mallien datan hallinnoinnista ja WebGL-rajapinnan käyttämien WebGLBuffer-olioiden luomisesta mallien sisältämistä tiedoista. Renderer-luokka voi ladata kuvia annetusta tiedostopolusta, ja se voi muodostaa piirrossa käytettäviä WebGLTexture-olioita ladatuista kuvista. Renderer-luokka määrittää maailma-objekteja, joihin voidaan asettaa piirrossa käytettäviä MeshObject-, CameraObject-, PointLightObject- ja SpotlightObject-oloita, sekä muokata maailmassa olevaa globaalia valaistusta. Renderer-luokka voi piirtää luotuun canvas-elementtiin kuvia käyttämällä render-metodia, joka käyttää aktiivista maailma-objektia ja sen sisältämiä asetuksia ja resursseja kuvan muodostamiseen.
Alustus: `WEBGL_LIB.Renderer(int width, int height)` width - määrittää luotavan canvas-elementin leveyden height – määrittää luotavan canvas-elementin korkeuden
Jäsenmuuttujat
`object canvas`
HTML5 canvas-elementti. joka toimii piirtoalustana, ja josta WebGL-rajapinnan konteksti noudetaan. Luokan luomisen jälkeen, canvas tulee lisätä HTML-dokumenttiin erikseen, jotta se olisi näkyvä elementti.
`object shaderConfs`
Objekti sisältää muuttujia, joilla voidaan vaikuttaa muodostettavien varjostimien vakioiden arvoihin.
`int shaderConfs.maxPointLights`
Määrittää fragmenttivarjostimen vastaanottamien PointLightObject-olioiden maksimäärän. Oletusarvoisesti 10.
`int shaderConfs.maxPointLights`
Määrittää fragmenttivarjostimen vastaanottamien SpotlightObject-olioiden maksimäärän. Oletusarvoisesti 10.

`WebGLRenderingContext gl`
WebGLRenderingContext-objekti, joka saadaan canvas-muuttujalta. Tätä käytetään kaikissa kutsuissa WebGL-rajapinnan kanssa.
`object shaders`
Objekti, joka sisältää piirrossa käytettävien verteksi- ja fragmentti varjostimien lähdekoodit, varjostimien tyypit ja käännetyt WebGLShader-objektit. jotka ovat käytössä olevat varjostimet.
`WebGLProgram program`
WebGLProgram-objekti, joka on käytössä oleva varjostinohjelma. Ohjelmaan on liitetty shaders-objektin sisältämät, käännetyt WebGLShader-objektit.
`object worlds`
Objekti, joka sisältää kaikki luodut maailmat. Maailmat sisältävät piirrossa käytettävät MeshObject-, PointLightObject-, SpotlightObject- ja CameraObject-luokkien oliot, sekä muita piirrossa käytettäviä asetuksia. Vain maailma, joka on asetettu activeWorld-objektiin, piirretään.
`object activeWorld`
Objekti, joka kertoo aktiivisen maailman, josta piirrossa käytettävät MeshObject-, PointLightObject-, SpotlightObject- ja CameraObject-luokkien oliot, sekä muut piirtoasetukset noudetaan. Osoittaa johonkin worlds-objektissa olevaan maailma-objektiin. Kun ensimmäinen maailma luodaan, asetetaan se oletuksella aktiiviseksi maailmaksi.
`object loadedModels`
Sisältää kaikkien COLLDADA -tiedostoista ladattujen mallien verteksi- ja indeksitiedot, sekä mallein sisältämistä arvoista muodostetut WebGLBuffer-puskurit, joita voidaan käyttää mallitietojen siirtämiseen verteksivarjostimelle piirrossa. Ladatut verteksitiedot koostuvat vertekseistä, tekstuurikoordinaateista sekä normaalivektoreista. Malleja muodostetaan loadModelFromCollada-metodin avulla, jossa objektiin tallennettujen mallien objektirakenne määritetään.
`boolean defaultImageLoaded`
Boolean arvo, joka ilmaisee sen, onko oletustekstuurin käyttämä kuva ladattu. käytetään render-metodin kanssa varmistamaan, ettei mitään piirretä ennen kuin oletustekstuurin kuva on ladattu muistiin. Kun kuva on ladattu, voidaan render-metodi ajaa kokonaan.

`object loadedTextures`
Sisältää kaikki loadImage-metodilla ladatut kuvat ja kuvista muodostetut WebGLTexture-oliot.
`object primitiveModes`
Objekti, joka sisältää piirtoon käytettävien primitiivin muodostus tapojen arvot, joita WebGL-rajapinta käyttää. Käytetään WebGL-rajapinnan drawElements-funktion kutsun yhteydessä. Piirrettävän primitiivin tyyppiä voidaan muuttaa MeshObject-olion primitiveMode-arvolla. Oletuksena MeshObject-luokan oliot käyttävät TRIANGLES-arvoa, joka piirtää kolmioprimitiivejä.
Metodit
`void createGLContext(void)`
Noutaa WebGLRenderingContext-olion luokan canvas-elementistä, ja sijoittaa noudetun kontekstin gl-jäsenmuuttujaan. Jos kontekstin luominen ei onnistu, käyttäjä saa alert-ilmoituksen tapahtuneesta virheestä.
`void setupRenderer(void)`
Asettaa muutamia WebGL-rajapinnan käyttämiä asetuksia piirron yhteydessä. Asettaa kuvapuskurin puhdistusvärin mustaksi, ottaa fragmenttien syvyystestauksen käyttöön ja määrittää fragmenttien syvyystestauksessa käytettävän oletusfunktion. Lopuksi funktio alustaa kuvapuskurin pikseleiden väriarvot ja syvyysarvot.
`void createShaders(void)`
Luo verteksi- ja fragmentti varjostimet, joita käytetään ruudun piirtämiseen. Jos varjostimia ei ole määritelty luokan shaders-objektiin, objektiin luodaan oletus verteksivarjostin ja fragmenttivarjostin. Shaders-objektiin asetettuja lähdekoodeja ja tyyppejä käytetään luomaan uudet varjostimet, jotka liitetään luokan program-varjostinohjelmaan. Lopuksi varjostinohjelma liitetään käytettäväksi piirtolinjastossa, jolloin luodut varjostimet asettuvat käyttöön. Mahdollisissa varjostimen kääntämisessä ilmaantuneissa varjostimien lähdekoodin syntaksivirheissä tai muissa virhetilanteissa funktio antaa virheilmoituksen, joka pyrkii ilmaisemaan virheen lähteen.
`void setupDefaultTexture(void)`
Luo WebGL-rajapinnan WebGLTexture-olion käyttäen oletuskuvaksi ladattua tekstuurikuvaa oletuskuvana, ja asettaa luodun tekstuuri objektin luokan de-

faultTexture-jäsenmuuttujaan. Kutsutaan ensimmäisen render-metodin kutsun yhteydessä, kun oletuskuva on ladattu muistiin.
`void loadDefualtImage(void)`
Metodi luo kuvan, jota käytetään oletustekstuurin luomiseen ensimmäisen piirron yhteydessä. Renderer-luokka ei voi suorittaa render-metodia, ennen kuin funktion määrittelemä oletuskuva on ladattu muistiin. Kuva luodaan merkkijonosta, joka sisältää Base64 muotoon kryptatun harmaan PNG-kuvatiedoston, jonka koko on 1 x 1 pikseliä.
`void createWorld(string name)`
Luo uuden maailmaa esittävän object-muuttujan, johon kaikki piirtämiseen käytettävät muut luokat voidaan sijoittaa, ja jolle voidaan antaa piirtämiseen vaikuttavia asetuksia. Luotu maailma-objekti sijoitetaan luokan worlds-objektiin käyttäen avaimena parametrina annettua nimeä. Jos nimiparametri jätetään antamatta, generoidaan avain automaattisesti. Jos aikaisempia maailoja ei ole luotu luokan worlds-jäsenmuuttujaan, asetettaan luotu maailma aktiiviseksi sijoittamalla se luokan activeWorld-jäsenmuuttujaan.
`void setActiveWolrd(string worldName)`
Asettaa luokan activeWorld-jäsenmuuttujan osoittamaan luokan worlds-objektiin luotuun maailma-objektiin. Parametrina annettu nimi tulee vastata luodun maailman avainta luokan worlds-objektissa, joka on luokan metodin createWorld-metodin kutsun yhteydessä annettu merkkijono, tai automaattisesti generoitu merkkijono, jos nimeä ei anneta erikseen parametrina.
`void loadImage(string path, string name)`
Metodilla voidaan ladata kuva parametrina annetusta merkkijonosta. Ladattu kuva tallennetaan luokan loadedTextures-objektiin, jonne kuvan avaimeksi asetetaan parametrina annettu nimi, tai automaattisesti generoitu merkkijono. Ladatusta kuvasta muodostetaan tekstuuri setupTexture-metodin avulla.
`void render(void)`
Metodi piirtää kuvan käyttäen kaikkia activeWorld-objektiin lisättyjä MeshObject-, PointLightObject-, SpotlightObject- ja CameraObject-luokkien olioita, sekä maailman muita piirtoasetuksia. Render-metodia ei voi suorittaa, ennen kuin oletustekstuurin käyttämä kuva on ladattu. Tämän jälkeen kaikki aktiivisen maailman MeshObject-luokan oliot piirretään yksi kerrallaan käyttäen kaikille samoja valistus ja kamera asetuksia. Jos MeshObject-oliolle on asetettu oma tekstuurina käytettävä kuva MeshObject-luokan textureImage-jäsenmuuttujaan, MeshObject-olion käyttämä tekstuuri luodaan siinä vaiheessa kun käytettävä

kuva on latautunut muistiin.
`void loadModelFromCollada(Document colladaDoc, string name)`
Metodilla voidaan ladata mallidataa luokan LoadedModels-objektiin parametrina annetusta COLLADA-dokumentista, joka on XML-pohjainen Document-objekti. Metodi lataa kaikkien COLLADA-dokumentissa määriteltyjen 3D-mallien verteksi- ja indeksitiedot. Ladattavat verteksitiedot ovat mallin verteksien sijainti, verteksien tekstuurikoordinaatit ja verteksien normaalit. Ladatut indeksitiedot sisältävät vain sijaintiverteksien piirtämisjärjestyksen. Jos ladattu malli ei sisällä tekstuurikoordinaatteja, muodostetaan mallille tekstuurikoordinaatit nolla arvoista, jotta malli toimisi piirrossa, vaikkei tekstuureita käytetäkään.
`void setupTexture(object texuteObject)`
Ottaa parametrina Renderer-luokan loadedTextures-objektiin loadImage-metodilla ladatun tekstuuri-objektin, joka sisältää kuvan, jonka avulla annettuun objektiin lisätään WebGLTexture-olion, jota voidaan käyttää piirrettävän MeshObject-olion teksturointiin.
`void calculateNormals (object modelObject, boolean inverse)`
Ottaa parametrina Renderer-luokan loadedModels-objektiin loadModelFromCollada-metodilla ladatun malli-objektin, joka sisältää mallin, jonka verteksi ja indeksi arvojen avulla mallille muodostetaan uudet normaalivektorit. Jos parametri inverse asetettu true-arvoksi, käännetään lasketut normaalivektorit osoittaman vastakkaiseen suuntaan.
`void setupBuffers(object modelObject)`
Ottaa parametrina Renderer-luokan loadedModels-objektiin loadModelFromCollada-metodilla ladatun malli-objektin, jonka sisältämistä arvoista muodostetaan objektiin WebGLBuffer-puskurit, joilla mallin arvot voidaan siirtää verteksivarjostimen attribute-muuttujille.
`void resizeRenderer(int width, int height)`
Muuttaa Renderer-luokan määrittelemän canvas-elementin kokoa parametreina annetun leveyden ja korkeuden mukaiseksi. Kertoo WebGL-rajapinnalle viewport-funkton avulla uudet dimensiot, jotta kuva piirretään samansuuruisena kuin canvas-elementti on määritelty. Jos Renderer-luokan activeWorld-objektille on määrittely piirrossa käytettävä CameraObject-olio, CameraObject-olion sisältämät projektiomatriisin asetukset päivitetään sen properties-jäsenmuutujaan, ja projektiomatriisi muodostetaan uudestaan resetProjection-metodin avulla.

Renderer.worlds maailma-objektit
`Renderer renderer`
Renderer-olio, joka loi maailma-objektin createWorld-metodilla.
`MeshObject meshes[]`
Taulukko, joka sisältää kaikki maailma-objektiin lisätyt MeshObject-oliot, joita käytetään piirrossa, jos maailmaobjekti on render-metodia kutsuneen Rendere-olion activeWorld-muuttujassa.
`PointLightObject pointLights[]`
Taulukko, joka sisältää kaikki maailma-objektiin lisätyt PointLightObject-oliot, joita käytetään piirrossa, jos maailmaobjekti on render-metodia kutsuneen Rendere-olion activeWorld-muuttujassa.
`SpotLightObejct spotLights[]`
Taulukko, joka sisältää kaikki maailma-objektiin lisätyt SpotlightObject-oliot, joita käytetään piirrossa, jos maailmaobjekti on render-metodia kutsuneen Rendere-olion activeWorld-muuttujassa.
`CameraObject cameras[]`
Taulukko, joka sisältää kaikki maailma-objektiin lisätyt CameraObject-oliot. Vain maailma-objektin acticeCamera-muuttujaan asetettua CameraObject-oliota käytetään piirrossa.
`Vector3f ambientLight`
Määrittää globaalin ambienttivalaistuksen värin, jolla maailmaan lisättyjä MeshObject-olioiden määrittelemiä kappaleiden pintoja valaistaan piirrossa.
`Vector3f directionalLightColor`
Määrittää globaalin suunnatun valaistuksen muodostaman valon värin, jolla maailmaan lisättyjä MeshObject-olioiden määrittelemiä kappaleiden pintoja valaistaan piirrossa.
`Vector3f directionalLightDir`
Määrittää globaalin suunnatun valaistuksen muodostaman valon kulkusuunnan maailma-avaruudessa, jolla maailmaan lisättyjä MeshObject-olioiden määrittelemiä kappaleiden pintoja valaistaan piirrossa.

`CameraObject activeCamera`
Määrittää CameraObject-olion, jonka määrittelemiä transformaatioita ja projektiomatriisia käytetään piirrossa silloin, kun maailma-objekti on asetettu render-metodia kutsuneen Renderer-olion activeWorld-muuttujaan.
`void setActiveCamera(CameraObject camera)`
Asettaa parametrina annetun CameraObject-olion maailman activeCamera-arvoksi, jota käytetään piirrossa käytettävänä kamerana.
`void addCamera(CameraObject camera)`
Lisää parametrina annetun CameraObject-olion maailma-objektin cameras-taulukkoon, ja jos ensimmäinen lisätty kamera, asettaa parametrina annetun CameraObject-olion maailman activeCamera-arvoksi, jota käytetään piirrossa käytettävänä kamerana.
`void addMesh(MeshObject mesh)`
Lisää parametrina annetun MeshObject -olion maailma-objektin meshes-taulukkoon
`void addPointLight(PointLightObject light)`
Lisää parametrina annetun PointLightObject -olion maailma-objektin point-Lights-taulukkoon
`void addSpotLight(SpotlightObject light)`
Lisää parametrina annetun SpotlightObject -olion maailma-objektin spotlights-taulukkoon

Renderer.loadedModels malli-objektit
`array vertices.data`
loadModelFromCollda-metodin lataaman mallin verteksit sisältävä taulukko.
`array normals.data`
loadModelFromCollda-metodin lataaman mallin normaalivektorit sisältävä taulukko.
`array textureCoords.data`
loadModelFromCollda-metodin lataaman mallin tekstuurikoordinaatit sisältävä

taulukko.
`array indices.vertexIndices`
loadModelFromCollda-metodin lataaman mallin verteksien indeksit sisältävä taulukko. Indeksit määrittävät verteksien, normaalivektoreiden ja tekstuurikoordinaattien käyttöjärjestyksen primitiivien muodostuksessa.
`WebGLBuffer buffers.vertexBufferObject`
Renderer-luokan lodadedModels-objektin sisältämän mallin verteksi datasta muodostettu gl.ARRAY_BUFFER tyyppinen WebGLBuffer-objekti, jota käytetään piirrossa siirtämään mallin verteksit verteksivarjostimelle attribute-muuttujana.
`WebGLBuffer buffers.normalBufferObject`
Renderer-luokan lodadedModels-objektin sisältämän mallin normaali datasta muodostettu gl.ARRAY_BUFFER tyyppinen WebGLBuffer-objekti, jota käytetään piirrossa siirtämään mallin normaalit verteksivarjostimelle attribute-muuttujana.
`WebGLBuffer buffers.textureCoordBufferObject`
Renderer-luokan lodadedModels-objektin sisältämän mallin tekstuurikoordinaatti datasta muodostettu gl.ARRAY_BUFFER tyyppinen WebGLBuffer-objekti, jota käytetään piirrossa siirtämään mallin tekstuuri koordinaatit verteksivarjostimelle attribute-muuttujana.
`WebGLBuffer indexBufferObject`
Renderer-luokan lodadedModels-objektin sisältämän mallin indeksi datasta muodostettu gl.ELEMENT_ARRAY_BUFFER WebGLBuffer-objekti, jota käytetään piirrossa määrittämään verteksien käyttöjärjestys primitiivien muodostamisessa.

Renderer.loadedTextures tekstuuri-objektit
`Image image`
Renderer-luokan loadImage-metodin lataama kuva. Käytetään WebGLTexture-olion muodostamiseen.
`WebGLTexture texture`
Tekstuuri-objektin kuvasta muodostettu WebGL-rajapinnan WebGLTexture-olio,

joka voidaan ottaa piirrossa käytettäväksi tekstuuriksi piirrettävän MeshObject-olion määrittämän mallin teksturointiin.
`string name`
Renderer-luokan loadImage-metodin lataamalle kuvalle annettu nimi, joka on sama kuin objektin avain Renderer.loadedTextures-objektissa.

WEBGL_LIB.BaseObject
BaseObject-luokka määrittää toiminnallisuudet perustransformaatioiden määrittelyyn, jotka voidaan periyttää muille luokille, jotka hyödyntävät transformaatioita.
Alustus: `WEBGL_LIB.BaseObject(void)`
Jäsenmuuttujat
`object transformations`
Objekti, joka kokoaa kaikki BaseObject-luokan oliolle suoritettavat transformaatiot sisälleen. Objektin sisälle määritellyistä transformaatioista muodostetaan kaikki olioille määritellyt transformaatiot suorittava 4x4 matriisi luokan getTransforamtionMatrix()-metodin kutsun yhteydessä.
`Quaternion transformations.worldOrienatation`
Kvaternio, joka määrittää kappaleelle suoritettavan kierron maailma-avaruudessa olevan akselin ympäri. Tätä kvaterniota muutetaan aina, kun BaseObject-luokan olioille suoritettaan kiertoja rotateWorld()- tai rotateWorldXYZ()-metodien avulla. Kvaterniosta muodostetaan matriisi, joka yhdistetään muihin MeshObject-luokan transformaatioihin getTransformationMatrix()-metodin kutsun yhteydessä.
`Quaternion transformations.localOrienatation`
Kvaternio, joka määrittää kappaleelle suoritettavan kierron paikallisessa objektiavaruudessa olevan akselin ympäri. Tätä kvaterniota muutetaan aina, kun BaseObject-luokan olioille suoritettaan kiertoja rotateLocal- tai rotateLocalXYZ-metodien avulla. Kvaterniosta muodostetaan matriisi, joka yhdistetään muihin MeshObject-luokan transformaatioihin getTransformationMatrix-metodin kutsun yhteydessä.

`Vector3f transformations.scale`
Vektori, joka määrittää BaseObject-luokan oliolle suoritettavan skaalauksen määrän x-, y- ja z-akseleiden suunnassa. Vektorin x, y ja z komponenttien arvoja voidaan päivittää luokan scaleXYZ()-metodin avulla, joka lisää parametreina annetut arvot vektorin komponenttien arvoihin. Vektorin arvoista muodostetaan skaalauksen suorittava matriisi, joka yhdistetään BaseObject-luokan olion muihin transformaatioihin getTransformationMatrix-metodin kutsun yhteydessä.
`Vector3f transformations.translation`
Vektori, joka määrittää BaseObject-luokan oliolle suoritettavan translaation määrän x-, y- ja z-akseleiden suunnassa. Vektorin x, y ja z komponenttien arvoja voidaan päivittää luokan translateWorldXYZ-metodin avulla, joka lisää parametreina annetut arvot vektorin komponenttien arvoihin. Vektorin arvoista muodostetaan translaation suorittava matriisi, joka yhdistettään BaseObject-luokan olion muihin transformaatioihin getTransformationMatrix-metodin kutsun yhteydessä.
`object baseVectors`
Määrittää kolme Vector3f-olioita, jotka osoittavat x-, y- ja z-akseleiden suuntaisesti positiiviseen suunnan.
Metodit
`void rotateLocal(float angle, WEBGL_LIB.Vector3f direction)`
Kiertää objektia sen paikallisessa objektiavaruudessa parametrina annetun kulman angle verran suunnaksi annetun vektorin direction ympäri. Parametreina annetuista arvoista muodostetaan kvaternio, joka kerrotaan objektin paikallista orientaatiota esittävän transformations.localOrientation-objektin kanssa, jolloin kappaleen orientaatio muuttuu paikallisessa objektiavaruudessa olevan akselin ympäri suoritettavan kierron mukaisesti.
`void rotateLocalXYZ(float angX, float angY, float angZ)`
Kiertää BaseObject-luokan oliota sen paikallisen objektiavaruuden perusakseleiden x, y ja z ympäri parametreina annettujen kulmien verran. Parametrina annettu angX määrittää kierron määrän paikallisen x-akselin ympäri, angY mää-

rittää kierron määrän paikallisen y-akselin ympäri, ja angZ määrittää kierron paikallisen z-akselin ympäri. Parametreina annetuista arvoista muodostetaan kvaternio, joka kerrotaan objektin paikallista orientaatiota esittävän transformations.localOrientation-objektin kanssa, jolloin kappaleen orientaatio muuttuu objektin paikallisen objektiavaruuden akseleiden ympäri suoritettavan kierron mukaisesti.
`void rotateWorld(float angle, WEBGL_LIB.Vector3f direction)`
Kiertää objektia maailma-avaruudessa parametrina annetun kulman angle verran suunnaksi annetun vektorin direction ympäri. Parametreina annetuista arvoista muodostetaan kvaternio, joka kerrotaan objektin maailma-avaruuden orientaatiota esittävän transformations.worldOrientation-objektin kanssa, jolloin kappaleen orientaatio muuttuu maailma-avaruudessa olevan akselin ympäri suoritettavan kierron mukaisesti.
`void rotateWorldXYZ(float angX, float angY, float angZ)`
Kiertää BaseObject-luokan oliota maailma-avaruuden perusakseleiden x, y ja z ympäri parametreina annettujen kulmien verran. Parametrina annettu angX määrittää kierron määrän maailman x-akselin ympäri, angY määrittää kierron määrän maailman y-akselin ympäri, ja angZ määrittää kierron maailman z-akselin ympäri. Parametreina annetuista arvoista muodostetaan kvaternio, joka kerrotaan objektin maailma-avaruuden orientaatiota esittävän transformations.worldOrientation-objektin kanssa, jolloin kappaleen orientaatio muuttuu maailma-avaruuden akseleiden ympäri suoritettavan kierron mukaisesti.
`void translateWorldXYZ(float trX, float trY, float trZ)`
Siirtää BaseObject-luokan oliota maailma-avaruuden x, y ja z suunnassa parametreina annettujen siirtymäarvojen verran. Parametreina annettu trX määrittää kappaleen siirtymän maailman x-akselin suunnassa, trY määrittää kappaleen siirtymän maailman y-akselin suunnassa ja trZ määrittää kappaleen siirtymän maailman z-akselin suunnassa. Siirtymät päivitetään luokan transformations.locations-objektiin, lisäämällä annetut arvot objektin x, y ja z arvoihin.
`void scaleXYZ(float sclX, float sclY, float sclZ)`
Skaalaa BaseObject-luokan oliota x-, y- ja z-akseleiden suunnassa paramet-

reina annettujen skaalausarvojen verran. Parametreina annettu sclX määrittää kappaleen skaalauksen x-akselin suunnassa, sclY määrittää kappaleen skaalauksen y-akselin suunnassa ja trZ määrittää kappaleen skaalauksen z-akselin suunnassa.

Skaalaus arvot päivitetään luokan transformations.scale-objektiin, lisäämällä annetut arvot objektin x, y ja z arvoihin.

`WEBGL_LIB.Matrix4f getTransformationMatrix(void)`

Luo objektille määritellyistä transformaatioista matriisit, jotka yhdistetään yhdeksi 4 x 4 matriisiksi, joka suorittaa kaikki mallille asetetut rotaatiot, translaatiot ja skaalaukset. Palauttaa WEBGL_LIB.Matrix4f tyyppisen matriisin.

WEBGL_LIB.MeshObject
Peritään: WEBGL_LIB.BaseObject

MeshObject-luokkaa käytetään piirrettävien kappaleiden määrittämiseen. MeshObject-olio käyttää Renderer-luokan loadedModels-objektiin ladattuja malli-objekteja ja loadedTextures-objektiin ladattuja tekstuuri-objekteja piirrettävän kappaleen määrittämiseen, ja hyödyntää BaseObject-luokalta perittyjä transformaatioita piirrettävien kappaleiden transformointiin.

Alustus: `WEBGL_LIB.MeshObject(object modelObject)`
modelObject - Parametrina annettu Renderer-luokan loadedModels-objektin sisältämä malli-objekti asetetaan MeshObject-luokan model-jäsenmuuttujan arvoksi, joka määrittää MeshObject-luokan käyttämän mallin piirrossa.

Jäsenmuuttujat

`object model`

Osoittaa Renderer-luokan loadedModels-objektin sisältämään malli-objektiin. Mallia käytetään määrittämään MeshObject-luokan käyttämät verteksit, tekstuurikoordinaatit ja normaalivektorit, sekä indeksit, joita mallin piirtämiseen käytetään.

`object textureObject`

Osoittaa Renderer-luokan loadedTextures-objektin sisältämään tekstuuri-objektiin, jonka sisältämää WebGLTexture-oliota käytetään MeshObject-olion määrittelemän kappaleen tekstuurina piirrossa.

`boolean useTexture`

Määrittää, käyttääkö MeshObject-olio tekstuuria, vai surfaceProperties.surfaceColor väriarvoa olion määrittämän kappaleen pinnan värien määrittämiseen piirrettäessä.
`object surfaceProperties`
Objekti, joka sisältää useita piirrettävän MeshObject-olion pinnan väritykseen vaikuttavia asetuksia.
`Vector3f surfaceProperties.surfaceColor`
Vektorilla muodostettu RGB-väriarvo, jota käytetään piirrettävän MeshObject-olion määrittelemän kappaleen pinnan värinä, jos tekstuuria ei käytetä.
`float surfaceProperties.specular`
Määrittää piirrettävän MeshObject-olion määrittämän kappaleen pinnalle muodostetun spekulaari heijastuksen silloin, kun valoa osuu kappaleen pinnalle.
`Vector3f surfaceProperties.specularColor`
Määrittää MeshObject-olion määrittelemän kappaleen pinnalle muodostetun spekulaariheijastuksen värin RGB-arvona vektorin avulla.
`boolean surfaceProperties.useLigthing`
Määrittää, käyttääkö MeshObject-olio piirrettäessä valaistusta, vai piirretäänkö kappale siten, ettei valaistus vaikuta kappaleen väritykseen.
`object world`
Osoittaa Renderer-olioon luotuun maailma-objektiin, jonka avulla piirtoon käytettäviä MeshObject-luokan olioita hallitaan.
`string primitiveMode`
Merkkijono, jonka arvolla voidaan vaikuttaa piirron muodostamiin primitiiveihin. Mahdollisia arvoja ovat “TRIANGLES”, “LINES”, “POINTS”, ja niillä viitataan piirrossa Renderer-luokan primitiveModes-objektin sisältämiin arvoihin valitsemaan haluttu primitiivien muodostustapa.
Metodit
`void setTexture(object textureObject)`

Metodilla MeshObject-olio asetetaan käyttämään parametrina annettua Renderer-luokan loadedTextures-objektin sisällä olevaa tekstuuri-objektia. Parametri sijoitetaan luokan textureObject-jäsenmuuttujaan.
`void setSurfaceColor(Vector3f surfaceColor)`
Asettaa MeshObject-olion surfaceProperties.surfaceColor-jäsenmuuttujan arvon.
`void setSpecularReflection(float specular, Vector3f specularColor)`
Asettaa MeshObject-olion surfaceProperties.specular- ja surfaceProperties.specularColor-jäsenmuuttujien arvot.

WEBGL_LIB.CameraObject **Peritään: WEBGL_LIB.BaseObject**
CameraObject-luokalla voidaan määrittää maailmassa liikutettava ja kierrettävä kamera, joka määrittää myös projektiomatriisin, jonka avulla piirrettävien MeshObject-olioiden määrittelemät kappaleet sijoitettaan ruudulle lopullisiin sijainteihinsa verteksivarjostimessa, ennen primitiivien muodostamista. CameraObject-olio voidaan asettaa käytettäväksi kameraksi piirrossa antamalla se Renderer-luokan määrittelemälle maailma-oliolle.
Alustus: `WEBGL_LIB.CameraObject(object properties)` properties – Määrittää CameraObject-olion määrittelemälle projektiomatriisille asetettavat arvot. Parametrin properties tulee sisältää alla olevat arvot: • fov – näkökentän leveys asteina. 140-150 asettaa näkökentän normaaliksi • width – piirrettävän kuvan leveys, esim. Canvas-elementin leveys • height - piirrettävän kuvan korkeus, esim. Canvas-elementin korkeus • nearClip – lyhin matka, jolla piirrettävä kappale näkyvä, esim. 0.1 • farClip - pisin matka, jolla piirrettävä kappale näkyvä, esim. 1000.0 Parametrina annettu arvo sijoitetaan suoraan CameraObject-olion

properties-jäsenmuutujaan, jonka sisältämiä arvoja voidaan käyttää projektiomatriisin muodostamiseen.
Jäsenmuuttujat
`object properties`
Properties-objektiin tallennetaan kameralle parametreina annetut kameran projektion asetukset.
`Matrix4f projectionMatrix`
Perspektiivisen projektion määrittelevä matriisi, jonka avulla piirrossa avaruudessa olevat kappaleet projektoidaan ruudulle. Projektiomatriisi muodostetaan CameraObject-luokan alustuksessa annetuista arvoista hyödyntäen Math-nimiavaruuden getPerspectiveProjMat4f-funktiota, joka luo perspektiivisen projektion toteuttavan matriisin. Matriisin määrittelyä varten kameralle on annettava alustuksen yhteydessä tietoina näkökentän laajuus asteina, piirtoalueen leveys ja korkeus, sekä tiedot lyhimmästä ja pisimmästä etäisyydestä, jotka projektio voi nähdä. Nämä arvot muodostavat pyramidin muotoisen alueen, jonka sisälle sijoittuvat kappaleet projektoidaan näytölle.
`Vector3f forward`
Vektori, joka määrittää suunnan joka on kameran näkökulmasta eteenpäin, eli suunta, johon kamera katsoo. Tämän suunnan tietäminen on tärkeää kameralle suoritettavien transformaatioiden vuoksi, joiden suoritustapa eroaa normaalista BaseObject-luokan määrittelemistä transformaatioista.
`Vector3f up`
Vektori, joka määrittää suunnan joka on kameran näkökulmasta ylös. Tämän suunnan tietäminen on tärkeää kameralle suoritettavien transformaatioiden vuoksi, joiden suoritustapa eroaa normaalista BaseObject-luokan määrittelemistä transformaatioista.
Metodit
`Vector3f getRight(void)`
Metodi, jolla voidaan hakea kameran näkökulmasta oikealle osoittava vektori. Kameran oikealle osoittava vektori saadaan kertomalla kameran näkökulmasta ylös osoittava up-vektori kameran näkökulmasta suoraan osoittavan forward-vektorin kanssa. Palauttaa paluuarvona vektorin, joka on suunta, joka on kame-

ran näkökulmasta oikea.
`void rotateCamera(float rotX, float rotY)`
Metodi, joka kiertää CameraObject-luokan forward- ja position-vektoreita x- ja y-akseleiden ympäri parametreina annettujen kulmien verran. Kiertoa varten luodaan kvaternio parametreina annettujen arvojen avulla, jolla forward- ja up-vektorit kierretään.
`void translateCamera(Vector3f dirVector, float amount)`
Metodi, jolla kameraa liikutetaan parametrina annetun vektorin suunnassa parametrina annetun määrän verran. Parametrien määrittämä translaatio lisätään BaseObject-luokalta perittyyn transformations-objektin locations-vektoriin.
`Matrix4f getCameraRotationMatrix(void)`
Metodi, jolla rakennetaan kameran rotaation suorittava matriisi. Matriisi muodostetaan ottamalla CameraObject-luokan forward- ja up-vektorit, sekä hakemalla kameran oikealle osoittava vektori luokan getRight-metodin avulla. Jokaisen kolmen vektorin x-, y- ja z-komponenttien arvot sijoitetaan matriisin muodostavaan taulukkoon siten, että ne määrittävät maailman x-, y- ja z-akseleiden orientaation, eli matriisi kiertää maailma-avaruutta (samalla kaikkia maailmassa olevia objekteja), joka saa näkymän vaikuttamaan siltä, että kameraa kierretään. Palauttaa 4x4 matriisin, joka suorittaa kameran määrittämän rotaation.
`Matrix4f getCameraTranslationMatrix(void)`
Metodi, joka käyttää translations-objektin locations-vektoriin sijoitettuja arvoja muodostamaan matriisin, jolla maailmaa siirretään vastakkaiseen suuntaan, johon kamera liikkuu kullakin akselilla. Palauttaa matriisin, joka siirtää maailmaa kameran liikkeiden mukaan.
`void resetProjection(void)`
Metodilla CameraObject-olion määrittelemä projectionMatrix-jäsenmuuttuja alustetaan uudestaan olion properties-jäsenmuuttujan sisältämien arvojen avulla. Arvot samat kuin alustuksessa jos properties-jäsenmuutujaan ei ole koskettu.

WEBGL_LIB. PointLightObject **Peritään: WEBGL_LIB.BaseObject**
PointLightObject-luokka määrittää paikallisen valaistuksen muodostavan piste-

valon, joka valaisee ympärillään olevia MeshObject-olioiden määrittelemiä kappaleita piirrossa

Alustus: `WEBGL_LIB.PointLightObject(Vector3f lightPosition, Vector3f lightColor, float lightLength)`

lightPosition – Pistevalon sijainnin maailma-avaruudessa määrittelevä vektori
lightColor – Pistevalon tuottaman valon RGB-värin määrittelevä vektori
lightLength – Pistevalon muodostaman valon maksimi kulkumatka avaruudessa

Jäsenmuuttujat

`Vector3f position`

position-vektori määrittää pistevalon sijainnin maailmassa x-, y- ja z-koordinaattien mukaisesti.

`Vector3f lightColor`

lightColor-vektori määrittää pistevalo lähettämän valon RGB-värin. Vektorin x-arvo vastaa valon punaista väriarvoa, y-arvo vastaa valon vihreää väriarvoa ja z-arvo sinistä valon väriarvoa.

`float lightLength`

lightLength on liukuluku arvo määrittää, kuinka kauas valo kulkee pistevalon lähteen sijainnista.

WEBGL_LIB. SpotlightObject
Peritään: WEBGL_LIB.BaseObject

SpotlightObject-luokka määrittää paikallisen valaistuksen muodostavan spottivalon, joka valaisee luokan arvoista määritetyn valokeilan muotoisella alueella olevia MeshObject-olioiden määrittelemiä kappaleita piirrossa

Alustus: `WEBGL_LIB.PointLightObject(Vector3f lightPosition, Vector3f lightColor, float lightLength, Vector3f target, float angle, float hardness)`

lightPosition – Spottivalon sijainnin maailma-avaruudessa määrittelevä vektori

lightColor – Spottivalon tuottaman valon RGB-värin määrittelevä vektori lightLength – Spottivalon muodostaman valon suurin kulkumatka avaruudessa target – Spottivalon valokeilan osoittama kohde avaruudessa. angle – Spottivalon muodostaman valokeilan koon määrittelevä kulma. hardness - Spottivalon valokeilan reunojen valon hälventyminen.
Jäsenmuuttujat
`Vector3f position`
position-vektori määrittää spottivalon sijainnin maailmassa x-, y- ja z-koordinaattien mukaisesti.
`Vector3f lightColor`
lightColor-vektori määrittää spottivalon lähettämän valon RGB-värin. Vektorin x-arvo vastaa valon punaista väriarvoa, y-arvo vastaa valon vihreää väriarvoa ja z-arvo sinistä valon väriarvoa.
`float lightLength`
lightLength on liukuluku arvo määrittää, kuinka kauas valo kulkee spottivalon sijainnista.
`Vector3f target`
target-vektori määrittää sijainnin, johon spottivalo kohdistaa heijastamansa valokeilan.
`float angle`
Määrittää spottivalon muodostaman valokeilan kulman, joka määrittää, missä kulmassa valokeila lähtee laajenemaan valolähteestä.
`float hardness`
Määrittää spottivalon muodostaman valokeilan reunojen hälventymisen. Mitä suurempi arvo hardness-jäsenmuuttuujaan määritetään, sitä enemmän se hälventää valokeilan reunojen valon määrää.

WEBGL_LIB.Math.Entities.Vector3f
Vector3f-luokka määrittää 3D-vektoria esittävän olion, jota voidaan hyödyntää

3D-grafiikan muodostamiseen.
Alustus: `WEBGL_LIB.Math.Entities.Vector3f(float x, float y, float z)` x – vektorin x-koordinaatti y – vektorin y-koordinaatti z – vektorin z-koordinaatti
Jäsenmuuttujat
`float x`
Vector3f-olion esittämän vektorin x-koordinaatti.
`float y`
Vector3f-olion esittämän vektorin y-koordinaatti.
`float z`
Vector3f-olion esittämän vektorin z-koordinaatti.
Metodit
`Vector3f clone(void)`
Metodi palauttaa uuden Vector3f-olion, jonka arvot muodostettu metodia kutsuneen Vector3f-olion arvoista. Palautettu vektori on kopio metodia kutsuneesta vektorista.
`array getArray(void)`
Metodi muodostaa Vector3f-olion määrittämistä arvoista taulukon, joka palautetaan paluuarvona.
`Vector3f mulScal(float scalar)`
Metodi palauttaa uuden Vector3f-olion, jonka arvot muodostettu metodia kutsuneen Vector3f-olion arvojen ja parametrina annetun skalaariarvon välisestä tulosta.
`Vector3f divScal(float scalar)`
Metodi palauttaa uuden Vector3f-olion, jonka arvot muodostettu metodia kutsu-

neen Vector3f-olion arvojen ja parametrina annetun skalaariarvon välisestä jakolaskusta.
`Vector3f addVect(Vector3f vector3f)`
Metodi palauttaa uuden Vector3f-olion, jonka arvot muodostettu metodia kutsuneen Vector3f-olion arvojen ja parametrina annetun Vector3f-olion arvojen välisestä yhteenlaskusta.
`Vector3f subVect(Vector3f vector3f)`
Metodi palauttaa uuden Vector3f-olion, jonka arvot muodostettu metodia kutsuneen Vector3f-olion arvojen ja parametrina annetun Vector3f-olion arvojen välisestä erotuksesta.
`float dotProduct(Vector3f vector3f)`
Metodi palauttaa liukulukuarvon, jonka arvo muodostettu metodia kutsuneen Vector3f-olion arvojen ja parametrina annetun Vector3f-olion arvojen välisestä pistetulosta
`Vector3f crossProduct(Vector3f vector3f)`
Metodi palauttaa uuden Vector3f-olion, jonka arvot muodostettu metodia kutsuneen Vector3f-olion arvojen ja parametrina annetun Vector3f-olion arvojen välisestä ristitulosta.
`Vector3f length(void)`
Metodi palauttaa vektorin pituuden määrittelevän liukulukuarvon, jonka arvo muodostettu metodia kutsuneen Vector3f-olion arvoista.
`void normalize(void)`
Metodi muuttaa sitä kutsuneen Vector3f-olion yksikkövektoriksi, eli vektorin pituus on metodin kutsun jälkeen 1.
`float distance(Vector3f vector3f)`
Metodi palauttaa metodia kutsuneen Vector3f-olion määrittelemän vektorin ja parametrina annetun Vector3f-olion määrittelemän vektorin välisen etäisyyden liukulukuarvona.
`void negate(void)`
Metodi muuttaa sitä kutsuneen Vector3f-olion sisältämät arvot negatiivisiksi.

`Vector3f rotateVectorAroundAxis(float angle, Vector3f axisVector)`
Metodi palauttaa uuden Vector3f-olion, jonka arvot muodostettu metodia kutsuneen Vector3f-olion arvojen ja parametreina annettujen kulman ja Vector3f-olion avulla. Parametrina annetut kulma ja vektori määrittävät kierron akselin ympäri muodostamalla annetulla arvoilla kvaternion, jolla metodia kutsuneen vektorin arvot muunnetaan, ja palautettu vektori on metodia kutsuneen vektorin kierretty versio.
`Vector3f rotateWithQuaternion(Quaternion rotQuat)`
Metodi palauttaa uuden Vector3f-olion, jonka arvot muodostettu metodia kutsuneen Vector3f-olion arvojen ja parametrina annetun Quaternion-olion arvojen avulla. Parametrina annetun kvaternion määrittelemä kierto muuttaa metodia kutsuneen vektorin arvot, ja palautettu vektori on metodia kutsuneen vektorin kierretty versio.

WEBGL_LIB.Math.Entities.Matrix3f
Matrix3f-luokka määrittää 3 x 3 neliömatriisia esittävän olion, jota voidaan hyödyntää 3D-grafiikan transformaatioiden muodostamiseen.
Alustus: `WEBGL_LIB.Math.Entities.Matrix3f(array matrixArray)` matrixArray – 3 x 3 matriisin määrittelevä yksiulotteinen taulukko. Sisältää yhdeksän liukuluku alkiota.
Jäsenmuuttujat
`Float32Array[9] array`
3 x 3 matriisin arvoja esittävä liukuluku-taulukko, jossa 9 arvoa. Arvot esitetty matriisin sarakkeiden suuntaisesti. alla olevan esimerkin tavoin. [0, 1, 2, 3, 4, 5, 6, 7, 8] 0 3 6 1 4 7 2 5 8
Metodit
`Matrix3f mulScal(float scalar)`

Metodi palauttaa uuden Matrix3f-olion, jonka arvot ovat muodostettu parametrina annetun skalaariarvon ja metodia kutsuneen Matrix3f-olion arvojen välisestä kertolaskusta.
`Matrix3f matrixMult(Matrix3f mat3)`
Metodi palauttaa uuden Matrix3f-olion, jonka arvot ovat muodostettu parametrina annetun Matrix3f-olion arvojen ja metodia kutsuneen Matrix3f-olion arvojen välisestä matriisitulosta.
`Matrix3f transposeMatrix(void)`
Metodi palauttaa uuden Matrix3f-olion, jonka arvot ovat muodostettu metodia kutsuneen Matrix3f-olion arvojen avulla. Palautettu Matrix3f-olio on metodia kutsuneen matriisin transpoosi, eli matriisin rivit ovat siirretty sarakkeiksi.
`Matrix3f adjointMatrix(void)`
Metodi palauttaa uuden Matrix3f-olion, jonka arvot ovat metodia kutsuneen Matrix3f-olion arvojen avulla. Palautettu Matrix3f-olio on metodia kutsuneen matriisin liittomatriisi.
`float determinant(void)`
Metodi palauttaa uuden lukulukuarvon, jonka arvo on muodostettu metodia kutsuneen Matrix3f-olion arvojen avulla. Palautettu liukuluku on metodia kutsuneen Matrix3f-olion determinantti.
`Matrix3f inverseMatrix(void)`
Metodi palauttaa uuden Matrix3f-olion, jonka arvot ovat muodostettu metodia kutsuneen Matrix3f-olion arvojen avulla. Palautettu Matrix3f-olio on metodia kutsuneen matriisin käänteismatriisi, eli se toteuttaa metodia kutsuneen matriisiin määrittelemät transformaatiot käänteisesti.
`Vector3f multVec3(Vector3f vector3f)`
Metodi palauttaa uuden Vector3f-olion, jonka arvot ovat muodostettu parametrina annetun Vector3f-olion ja metodia kutsuneen Matrix3f-olion arvojen avulla. Palautettu Vector3f-olio on parametrina annetun vektorin matriisilla transformoitu versio. 3 x 3 matriisi voi esimerkiksi kiertää ja skaalata vektoria.

WEBGL_LIB.Math.Entities.Matrix4f
Matrix4f-luokka määrittää 4 x 4 neliömatriisia esittävän olion, jota voidaan hyödyntää 3D-grafiikan transformaatioiden muodostamiseen.

Alustus: `WEBGL_LIB.Math.Entities.Matrix4f(array matrixArray)` matrixArray – 4 x 4 matriisin määrittelevä yksiulotteinen taulukko. Sisältää 16 liukuluku alkiota.
Jäsenmuuttujat
`Float32Array[16] array`
4 x 4 matriisin arvoja esittävä liukuluku-taulukko, jossa 16 arvoa. Arvot esitetään matriisin sarakkeiden suuntaisesti. alla olevan esimerkin tavoin. [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15] 0 4 8 12 1 5 9 13 2 6 10 14 3 7 11 15
Metodit
`Matrix3f get3x3Matrix(void)`
Palauttaa Matrix3f-olion, jonka arvot muodostettu metodia kutsuneen Matrix4f-olion arvoista, jotka ovat 3 x 3 matriisin ”alueella”.
`Matrix4f mulScal(float scalar)`
Metodi palauttaa uuden Matrix4f-olion, jonka arvot ovat muodostettu parametrina annetun skalaariarvon ja metodia kutsuneen Matrix4f-olion arvojen välisestä kertolaskusta.
`Matrix4f matrixMult(Matrix4f mat4)`
Metodi palauttaa uuden Matrix4f-olion, jonka arvot ovat muodostettu parametrina annetun Matrix4f-olion arvojen ja metodia kutsuneen Matrix4f-olion arvojen välisestä matriisitulosta.

WEBGL_LIB.Math.Entities.Quaternion
Quaternion-luokka muodostaa kvaterniota esittävän olion, jota voidaan hyödyntää 3D-grafiikan transformaatioiden muodostamiseen.

Alustus: `WEBGL_LIB.Math.Entities.Quaternion(float w, vector3f)` w – määrittää kvaternion reaaliosan, tai kvaternion määrittämän kierron määrän vector3f – määrittää kvaternion imaginaariosan, tai akselin, jonka ympäri kvaternion määrittämä kierto toteutetaan
Jäsenmuuttujat
`float w`
Quaternion-olion w-arvo määrittää kierron määrän akselin ympäri, kun se alustetaan kierron määrän määrittävän radiaanikulman kosiinilla (cos(θ)). Muutoin w-arvo esittää kvaternion reaaliosaa.
`Vector3f vector`
Quaternion-olion vector-jäsenmuuttuja määrittää akselin, jonka ympäri kierto voidaan toteuttaa, kun sen komponenttien arvot kerrotaan puolitetun kierron määrän määrittävän radiaanikulman sinillä ([x * sin(θ/2), y * sin(θ/2), z * sin(θ/2)]). Muutoin vektori esittää kvaternion imaginaari osia.
Metodit
`float quaternionLength(void)`
Palauttaa liukulukuarvon, joka on metodia kutsuneen Quaternion-olion suuruus/pituus.
`void normalize(void)`
Metodi muuttaa sitä kutsuneen Quaternion-olion arvoja jakamalla ne olion pituudella (quaternionLength), minkä jälkeen olio on yksikkökvaternio, eli olion pituus on 1.
`Quaternion addQuaternion(Quaternion quaternion)`
Metodi palauttaa uuden Quaternion-olion, jonka arvot ovat muodostettu parametrina annetun Quaternion-olion ja metodia kutsuneen Quaternion-olion yhteenlaskusta.
`Quaternion multQuaternion(Quaternion quaternion)`
Metodi palauttaa uuden Quaternion-olion, jonka arvot ovat muodostettu parametrina annetun Quaternion-olion ja metodia kutsuneen Quaternion-olion välisestä kvaterniotulosta.

`Quaternion multVector(Vector3f vector3f)`
Metodi palauttaa uuden Quaternion-olion, jonka arvot ovat muodostettu parametrina annetun Vector3f-olion ja metodia kutsuneen Quaternion-olion välisestä tulosta.
`Matrix4f getRotationMatrix(void)`
Metodi palauttaa uuden Matrix4f-olion, jonka arvot ovat muodostettu metodia kutsuneen Quaternion-olion määrittämistä arvoista. Metodia kutsuneen kvaternion määrittämä kierto sijoitetaan palautettavaan matriisiin.

Renderer.Math-nimiavaruuden funktiot
`float angleToRadians(float angle)`
Muuntaa parametrina annetut asteet radiaaneiksi.
`float radiansToAngle(float rad)`
Muuntaa parametrina annetut radiaanit asteiksi.
`Matrix4f getXRotationMat4f(float rotationAngle)`
Palauttaa paluuarvona Matrix4f-olion, joka määrittää x-akselin ympäri suoritettavan parametrina annettujen asteiden suuruisen kierron.
`Matrix4f getYRotationMat4f(float rotationAngle)`
Palauttaa paluuarvona Matrix4f-olion, joka määrittää y-akselin ympäri suoritettavan parametrina annettujen asteiden suuruisen kierron.
`Matrix4f getZRotationMat4f(float rotationAngle)`
Palauttaa paluuarvona Matrix4f-olion, joka määrittää z-akselin ympäri suoritettavan parametrina annettujen asteiden suuruisen kierron.
`Matrix4f getScaleMat4f(float x, float y, float z)`
Palauttaa paluuarvona Matrix4f-olion, joka määrittää skaalauksen suorittavan matriisin. Skaalauksen määrä x-, y- ja z-akseleiden suuntaisesti määritetään parametrina annetuilla liukulukuarvoilla.
`Matrix4f getTranslationMat4f(float x, float y, float z)`
Palauttaa paluuarvona Matrix4f-olion, joka määrittää translaation suorittavan matriisin. Translaation määrä x-, y- ja z-akseleiden suuntaisesti määritetään

parametrina annetuilla liukulukuarvoilla.
`Matrix4f getTranslationMat4fAlongDirection (Vector3f direction, float amount)`
Palauttaa paluuarvona Matrix4f-olion, joka määrittää translaation suorittavan matriisin. Translaation määrä määrittyy parametrina annetun Vector3f-oliolla esitetyn suunnan, ja liukulukuna annetun translaation määrän avulla. Matriisi määrittää siirron vektorin suunnassa, ja siirtoa tapahtuu parametrina annetun liukuluvun verran.
`getPerspectiveProjMat4f(float fov, float width, float height, float nearClip, float farClip)`
Palauttaa paluuarvona Matrix4f-olion, joka määrittää perspektiiviprojektion suorittavan matriisin. Matriisi muuttaa vektoreiden sijainnit ruutuavaruuteen, jossa vektoreiden sijainnit vastaavat niiden lopullista sijaintia projektioidulla tasolla tai kuvassa. fov määrittää projektion näkökentän leveyden, width ja height projektoitavan pinnan leveyden ja korkeuden, nearClip ja farClip lähimmän ja kaukaisimman matkan, josta vektoreita projektoidaan ruutuavaruuteen.
`Quaternion getRotationQuat(float angle, Vector3f axisVector)`
Palauttaa paluuarvona Quaternion-olion, joka määrittää parametreina annettujen arvojen mukaisen kierron. Angle parametri määrittää kierron määrän, ja axisVector vektori määrittää suunnan, jonka ympäri kierto tapahtuu.

VARJOSTIMIEN OMINAISUUDET

Varjostimien attribute-, uniform- ja varying-muuttujien merkitykset:

Attribute/Uniform	Tarkoitus
`attribute vec3 aPosition`	Osoittaa piirtoon käytettävän MeshObject-olion osoittaman Renderer.loadedModels-objektissa olevan mallin verteksit sisältämään puskuriin.
`attribute vec2 aTextureCoord`	Osoittaa piirtoon käytettävän MeshObject-olion osoittaman Renderer.loadedModels-objektissa olevan mallin tekstuurikoordinaatit sisältämään puskuriin.
`attribute vec3 aNormalVector`	Osoittaa piirtoon käytettävän MeshObject-olion osoittaman Renderer.loadedModels-objektissa olevan mallin normaalivektorit sisältämään puskuriin.
`uniform mat4 uModelMatrix`	Sisältää piirtoon käytettävän MeshObject-olion määrittelemät transformaatiot, jotka muodostettu 4x4-matriisiin. Matriisi muuntaa verteksien sijainnit piirrettävän mallin paikallisesta avaruudesta maailma-avaruuteen
`uniform mat4 uCameraMatrix`	Sisältää piirtoon käytettävän CameraObject-olion määrittelemät transformaatiot, jotka muodostettu 4x4-matriisiin. Matriisi muuntaa verteksien sijainnit maailma-avaruudesta kamera-avaruuteen.
`uniform mat4 uProjectionMatrix`	Sisältää piirtoon käytettävän CameraObject-olion määrittelemän 4x4-projektiomatriisin. Matriisi muuntaa verteksien kamera-avaruudesta ruutuavaruuteen, joka on verteksien lopullinen sijainti piirrettävässä kuvassa.
`uniform mat3 uNormalTransMatrix`	Normaalivektoreiden muuntamiseen käytettävä matriisi, joka on piirrettävän MeshObject-olion transformaatio matriisista muodostettu invertoitu ja transponoitu 3x3-matriisi, joka määrittää MeshObject-oliolle määritetyt rotaatiot ja skaalaukset mallin normaalivektoreille.
`uniform vec3 uCameraPosition`	Määrittää piirtoon käytettävän CameraObject-oliolle asetetun sijainnin maailma-avaruudessa.
`uniform sampler2D uTextureSampler0`	Määrittää tekstuuriyksikön, johon Renderer.loadedTextures-objektin sisältämä WebGLTexture-olion määrittelemä tekstuuri on aktivoitu bindTexture-funktiolla. Piirrettävä MeshObject-olio osoittaa Renderer.loadedTextures sisältämään objektiin, josta tekstuuri otetaan käyttöön.
`uniform float uSpecular`	Sisältää piirrettävän MeshObject-olion määrittelemän spekulaari heijastuksen arvon, jota käytetään spekulaariheijastuksen terävyyden ja koon

	määrittelyyn. Suurempi arvo luo terävämmän ja pienemmän vaalean heijastuksen piirrettävän kappaleen pintaan, kun siihen osuu valoa. (MeshObject.SurfaceProperties.specular)
`uniform vec3 uSpecularColor`	Sisältää piirrettävän MeshObject-olion määrittelemän spekulaari heijastuksen värin, jota käytetään spekulaariheijastuksen värinä. Usein spekulaari heijastuksen väri kannattaa jättää vaaleaksi arvoksi, mutta muilla väreillä voidaan saada aikaan mielenkiintoisia efektejä. (MeshObject.SurfaceProperties.specularColor)
`uniform bool uUseLigthing`	Sisältää piirrettävän MeshObject-olion määrittelemän boolean arvon jota käytetään määrittelemään, tulisiko piirrossa käytettäviä valoja soveltaa piirrettävään kappaleeseen. Oletusarvoisesti valoja käytetään, mutta asettamalla false-arvon, voidaan valojen vaikutus kappaleen väritykseen kumota, ja vain tekstuuria tai pinnan väriä käytetään värityksen määrittelyyn. (MeshObject.SurfaceProperties.useLigthing),
`uniform vec3 uSurfaceColor`	Sisältää piirrettävän MeshObject-olion määrittelemän väriarvon jota käytetään määrittelemään piirrettävän kappaleen pinnan väriksi yksittäinen väriarvo. Määriteltyä väriä käytetään silloin, kun teksturointia ei ole otettu kappaleelle käyttöön. (MeshObject.SurfaceProperties.surfaceColor),
`uniform bool uUseTexturing`	Sisältää piirrettävän MeshObject-olion määrittelemän boolean arvon (MeshObject.useTexture), jota käytetään määrittelemään, käyttääkö piirrettävä malli uTextureSampler0-uniform muuttujan määrittelemää tekstuuria pinnan värin määrittämiseen, vai uSurfaceColor-uniform muuttujan väri arvoa. Arvo on oletusarvoisesti false, mutta kun MeshObject-oliolle asetetaan tekstuuri setTexture-metodilla, muutetaan muuttujalle true-arvo.
`uniform vec3 uDirLightColor`	Sisältää piirtoon käytettävän maailman määrittelemän globaalin suunnatun valon värin (Renderer.activeWorld.directionalLightColor).
`uniform vec3 uDirLightDirection`	Sisältää piirtoon käytettävän maailman määrittelemän globaalin suunnatun valon kulkusuunnan avaruudessa (Renderer.activeWorld.directionalLightColor).
`uniform vec3 uAmbientLight`	Sisältää piirtoon käytettävän maailman määrittelemän globaalin ambienttivalon värin (Renderer.activeWorld.ambientLight).
`uniform int uPointLightCount`	Määrittää piirtoon käytettävän aktiivisen maailmaan lisättyjen PointLightObject-olioiden mää-

	rän.

`uniform struct PointLight {`
`  vec3 position;`
`  vec3 color;`
`  float length;`
`} uPointLights[MAX_POINT_LIGHTS]`
	Määrittelee kaikki piirrossa käytettävään maailmaan lisätyt PointLightObject-oliot taulukossa, joka sisältää PointLight-rakenteen mukaisia arvoja. Rakenne sisältää PointLightObject-olion määrittelemät pistevalon sijainnin maailma-avaruudessa, pistevalon tuottaman värin ja pistevalon muodostaman valon kulkeman maksimimatkan valonlähteen sijainnista.

`uniform int uSpotlightCount`	Määrittää piirtoon käytettävän aktiivisen maailmaan lisättyjen SpotlightObject-olioiden määrän.

`uniform struct Spotlight {`
`  vec3 position;`
`  vec3 target;`
`  vec3 color;`
`  float angle;`
`  float length;`
`  float hardness;`
`} uSpotlights[MAX_SPOTLIGHTS]`
	Määrittelee kaikki piirrossa käytettävään maailmaan lisätyt SpotlightObject-oliot taulukossa, joka sisältää Spotlight-rakenteen mukaisia arvoja. Rakenne sisältää SpotlightObject-olion määrittelemät spottivalon sijainnin maailma-avaruudessa, spottivalon valokeilan osoittaman kohteen maailma-avaruudessa, spottivalon tuottaman värin, spottivalon valokeilan koon määrittävän kulman, spottivalon muodostaman valon kulkeman maksimimatkan valonlähteen sijainnista ja spottivalon valokeilan valon hälventymisen valokeilan reunoilla.

`varying vec2 vTextureCoord`	Verteksivarjostimen lähettämät MeshObject-olion käyttämän mallin tekstuurikoordinaatit fragmenttivarjostimelle. Muodostettu suoraan attribute-muuttujan aTextureCoord arvoista
`varying vec3 vNormalVector`	Verteksivarjostimen lähettämät MeshObject-olion käyttämän mallin normaalivektorit fragmenttivarjostimelle. Muodostetaan kertomalla aNormalVector attribute-muuttujan arvo uniform-muuttujan uNormalTransMatrix määrittämällä transformaatiolla, ja normalisoimalla tuloksena saatu vektori.
`varying vec3 vFragPosition`	Verteksivarjostimen fragmenttivarjostimelle lähettämät verteksien sijaintikoordinaatit, jotka interpoloituvat fragmenttien sijainneiksi primitiivien muodostumisen kanssa.

Esimerkkisovellus opinnäytetyössä luodun kirjaston käytöstä

```
<html>
<head>
	<script src="js/webgl_library/webgl_lib.js"></script>
</head>
<body>
<script>
    // create renderer and add canvas element to document root, so it will be shown
    var renderer = new WEBGL_LIB.Renderer(window.innerWidth, window.innerHeight);
    document.body.appendChild(renderer.canvas);

    //Resize renderer-canvas when window resized
    window.onresize = function(){
        renderer.resizeRenderer(window.innerWidth, window.innerHeight);
    };

    // Define world
    renderer.createWorld("maailma");

    // Load COLLADA documents using XML-request
    var domReq = new XMLHttpRequest();
    domReq.open("GET","collada/walls.dae",false);
    domReq.send();
    colladaDoc1=domReq.responseXML;

    domReq.open("GET","collada/dounut.dae",false);
    domReq.send();
    colladaDoc2=domReq.responseXML;

    // Load 3D models from XML-documents that were requested
    renderer.loadModelFromCollada(colladaDoc1, "walls");
    renderer.loadModelFromCollada(colladaDoc2, "dounut");
    // Reset the normals of the loaded models
    renderer.calculateNormals(renderer.loadedModels["walls"]);
    renderer.calculateNormals(renderer.loadedModels["dounut"]);
    // Load texture Image-element from given path
    renderer.loadImage("images/dounut.png", "dounutTexture");
```

```
// Create MeshObjects from loaded models, and set set surface color and texture
// Wall mesh creation and settings
var wallsMesh = new WEBGL_LIB.MeshObject(renderer.loadedModels["walls"]);
wallsMesh.setSurfaceColor(new WEBGL_LIB.Math.Entities.Vector3f(0.8, 0.2, 0.5))

// Dounut mesh creation and settings
var dounutMesh = new WEBGL_LIB.MeshObject(renderer.loadedModels["dounut"]);
dounutMesh.setTexture(renderer.loadedTextures["dounutTexture"]);
dounutMesh.setSpecularReflection(36,
                          new WEBGL_LIB.Math.Entities.Vector3f(1, 1, 1));

// add meshes to world to be rendered
renderer.activeWorld.addMesh(wallsMesh);
renderer.activeWorld.addMesh(dounutMesh);

// scale, translate and rotate the world to its location in the background
wallsMesh.rotateWorldXYZ(Math.PI/2, Math.PI, 0);
wallsMesh.translateWorldXYZ(5, 0, -5);
wallsMesh.scaleXYZ(20, 20, 20);

// scale the counut by one unit (doubles the size of default scale mesh)
dounutMesh.scaleXYZ(1, 1, 1);

// examples of use of global lighting. Ambient is set, but directional light color
// is black so it won't affect lighting for now.
renderer.activeWorld.ambientLight = new WEBGL_LIB.Math.Entities.Vector3f(
                                                                  0.1, 0.1, 0.1);
renderer.activeWorld.directionalLightColor = new WEBGL_LIB.Math.Entities.Vector3f(
                                                                   0.0, 0.0, 0.0);
renderer.activeWorld.directionalLightDir = new WEBGL_LIB.Math.Entities.Vector3f(
                                                                0.3, -1.0, 0.0);
// create camera with perspective projection settings, and translate and rotate
// camera to new position from origin of the world
var camera = new WEBGL_LIB.CameraObject({
                                fov:140.0,
                                width:renderer.canvas.width,
                                height:renderer.canvas.height,
                                nearClip:0.1,
                                farClip:1000.0
});
renderer.activeWorld.addCamera(camera);
```

```
    camera.translateWorldXYZ(-10, 4, 16);
    camera.rotateCamera(-Math.PI/9, -Math.PI/4.5);

    //render first scene to canvas
    renderer.render();

    // create pointlght and spotlight and add them to world, so they will be used on
    // next render call
    var pointlight = new WEBGL_LIB.PointLightObject(
                new WEBGL_LIB.Math.Entities.Vector3f(0, 0, 0),//position
                new WEBGL_LIB.Math.Entities.Vector3f(1, 1, 1),// color
                100                                          //light distance
    );

    var spotlight = new WEBGL_LIB.SpotlightObject(
                new WEBGL_LIB.Math.Entities.Vector3f(-3, 3, 3), //position
                new WEBGL_LIB.Math.Entities.Vector3f(0, 0, 1), // color
                50,                                           //light distance
                new WEBGL_LIB.Math.Entities.Vector3f(0, 0, 0), //spotlight target
                Math.PI/10,                                   //spotlight cone angle
                30                                           //spot hardness
            );

    renderer.activeWorld.addPointLight(pointlight);
    renderer.activeWorld.addSpotlight(spotlight);

    // create steps to rotate the dounutMesh, move the
    // pointLight and alter the of spot light on update  function
    var stepX = 0.01;
    var stepY = 0.02;
    var pointMove = 0.0;
    var targetMove = 0.0;

    function update(){
        //rotate dounut around worlds x- and y-axis
        dounutMesh.rotateWorldXYZ(stepX, stepY, 0);

        //move pointlight back and forth along worlds x-axis
        var vect = new WEBGL_LIB.Math.Entities.Vector3f(
0.5 * Math.cos(pointMove), 0, 0);
        pointMove += 0.01;
```

```
        pointlight.position = pointlight.position.addVect(vect);

        // move the spotlight cone around by altering its target vector
        spotlight.target.z += 0.05 * Math.cos(targetMove);
        targetMove += 0.01;
        requestAnimationFrame(update, renderer.canvas);
        renderer.render();
    }
    update();

</script>
</body>
</html>
```