Samuli Helin

3D-mallien käsittely web-ympäristössä

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Mediatekniikan koulutusohjelma

Insinöörityö

10.5.2017

Metropolia

Tekijä Otsikko Sivumäärä Aika	Samuli Helin 3D-mallien käsittely web-ympäristössä 40 sivua + 1 liitettä 10.5.2017
Tutkinto	Insinööri (AMK)
Koulutusohjelma	Mediatekniikka
Suuntautumisvaihtoehto	Digitaalinen media
Ohjaajat	Toimitusjohtaja Pasi Porramo Yliopettaja Harri Airaksinen

Insinöörityön tarkoituksena oli kehittää 3D-mallien visualisointityökalu WebG-tekniikalla ja vertailla WebGL-sovellusten kehitysalustoja ja -ohjelmointikirjastoja. Työ tehtiin ohjelmisto-alan yritykselle.

Työssä perehdyttiin pintapuolisesti WebGL:n historiaan, ominaisuuksiin ja kehitystyökaluihin. WebGL on ohjelmointirajapinta, joka mahdollistaa laitekiihdytetyn grafiikan esittämisen verkkoselaimissa. Vertailun perusteella valittiin Unity sopivaksi kehitysalustaksi.

Visualisointityökalun alustavassa suunnittelussa vertailtiin eri 3D-mallien käyttämistä ja tuotantoprosessia Unity-kehitystyökalulla. Visualisointityökalun yksi keskeinen toiminto on saada 3D-mallit ladattua sovellukseen katseltavaksi verkkoyhteyden välityksellä sovelluksen ollessa jo käynnissä. Kehitystä varten otettiin käyttöön kaksi C#-kirjastoa, jotka mahdollistavat tämän FBX- ja OBJ-mallien kanssa. Lisäksi tutkittiin 3D-malliformaattien kompressointia ja sen hyödyllisyyttä työkalun näkökulmasta. Tuloksena selvisi, että ASCII-muotoinen OBJ-tiedosto on paras pakkautumaan verrattuna FBX-formaatin binääri- ja ASCII-muotoisiin tiedostoihin.

Visualisointityökalun tekeminen aloitettiin kehitysympäristön pystyttämisellä, 3D-mallien lataajien koodin tutkimisella ja niiden implementoinnilla työkaluun. Kehitysvaiheessa ilmenneitä ongelmia analysoitiin ja korjattiin. Suurin ongelma oli 3D-mallien parsimisen ohjelmakoodi, joka ei suoriutunut selainympäristössä ja tästä syystä itse visualisointityökalu ei valmistunut.

Tutkimuksista saatujen havaintojen ja kohdattujen ongelmien perusteella vaikuttaa siltä, että 3D-mallien lataaminen verkkoyhteyden välityksellä ja esittäminen selaimessa kannattaa tehdä jollakin JavaScript-ohjelmointikirjastolla eikä toisesta kielestä käännetyllä ohjelmointikoodilla. Insinöörityön tilaajayritys suunnittelee jatkokehitystä.

Avainsanat	WebGL, Unity, 3D-tiedostojen prosessointi, 3D-formaatit

Author	Samuli Helin
Title	Processing 3D-models in web environment
Number of Pages	40 pages + 1 appendices
Date	5 May 2017
Degree	Bachelor of Engineering
Degree Programme	Media Technology
Specialisation option	Digital Media
Instructors	Pasi Porramo, CEO Harri Airaksinen, Principal Lecturer

Goal of the thesis was to develop a visualization tool for 3D-models by using WebGL and to compare the WebGL- development platforms and programming libraries. This study was made for a company in software business.

WebGL's history, qualities and developing tools were studied in the thesis. WebGL is a programming interface which allows to present hardware accelerated graphics on web browsers. Based on comparison, Unity was chosen for suitable development platform.

The usage of different 3D-models and the actual production process with Unity development tool were compared during the visualization tool's preliminary planning process. One of the main goals was to load the 3D-models into the application to be viewed via internet while the application is running at the same time. Two C#-libraries capable of processing FBX- and OBJ-models were used in the development process. Furthermore 3D-model format's compression capability and compression's usefulness was also analyzed. As a result, it was found out that ASCII-formatted OBJ-file seemed to compress best compared to FBX-format's binary- ja ASCII-formatted files.

Building of the visualization tool started by setting up the development environment, investigating the codes of 3D-model-loaders and implementing them to the tool. Problems faced in the developmental phase were analyzed and repaired. The biggest problem seemed to be parsing the 3D-models in the code, which did not perform in web browser. As a result, the visualization tool was uncompleted.

The experience from the study alongside the faced problems led to the conclusion that loading 3D-models via internet access and viewing them in browser is to be done with JavaScript-library instead of code converted from another programming language. The software company is planning to conduct further studies on the matter.

Keywords	WebGL, Unity, 3D-file processing, 3D-formats

Sisällys

1 Johdanto

Insinöörityön tarkoituksena on tutkia WebGL-tekniikka ja kehittää WebGL:ää hyödyntävä 3D-visualisointityökalu 3D-mallien tarkastelemiseen selaimessa. Työ tehdään ohjelmistoalan yritykselle. Yrityksessä on tehty 3D-animaatioita ja ohjelmistoja muun muassa teollisuuden asiakkaille. Insinöörityössä tehtävä 3D-mallien visualisointityökalu on tarkoitus liittää yrityksen jo olemassa olevaan myyntikonfiguraattoriohjelmistoon.

Tavoitteena on saada tietoa WebGL-kehityksen mahdollisuuksista ja haasteista. WebGL-kehitystä koskevan tutkimuksen pohjalta valitaan asianmukaisin prosessi, jonka mukaan visualisointityökalua aletaan kehittää. Tavoitteena on myös päästä mahdollisimman lähelle valmista työkalua. Työkalun olisi tarkoitus olla loppukäyttäjälle suoraviivainen ja helppokäyttöinen.

Tärkeimmät asiat insinöörityössä ovat 3D-mallien käsittely ja käyttöönotto WebGL-kehitysalustoilla, kehitysalustojen vertailu, 3D-malliformaattien vertailu ja käytettävyyden tutkiminen web-sovelluksessa, 3D-mallien datan pakkaamisen hyödyt ja haasteet sekä WebGL-kehityksen erityispiirteiden ja rajoitteiden selvittäminen kehityksen aikana.

Insinöörityöraportti alkaa WebGL-tekniikan yleiskatsauksesta ja selaimessa suoritettavan grafiikan historiasta. Tämän jälkeen verrataan yleisimpiä WebGL-ohjelmointikirjastoja ja kehitysalustoja. Sovelluksen kehitys alkaa suunnittelulla ja valittujen työkalujen käyttöönotolla. Suunnitteluvaiheessa tutkitaan myös 3D-tiedostojen kompressointia ja sen hyötyjä sovelluksen käytettävyyden kannalta. Raportin lopussa esitellään visualisointisovelluksen kehityksen vaiheita ja kehitysvaiheessa ilmi tulleita haasteita. Viimeiseksi analysoidaan sovellusta ja kehitystyön aikana havaittujen asioiden pohjalta tehtyjä jatkokehityssuunnitelmia.

2 WebGL – laitekiihdytettyä grafiikkaa verkko-selaimessa

2.1 WebGL-ohjelmointi ja HTML5-kuvauskieli

WebGL (*Web Graphics Library*) on ohjelmarajapinta, joka tarjoaa mahdollisuuden suorittaa reaaliaikaista grafiikkaa verkkoselaimissa. WebGL (https://www.khronos.org/webgl/) toimii JavaScriptillä lähes kaikissa moderneissa verkkoselaimissa, mikä mahdollistaa kaksi- ja kolmiulotteisen grafiikan esittämistä hyödyntäen tietokoneessa olevaa näytönohjainta. (1.) Aikaisemmin grafiikan esittäminen verkkoselaimissa on tapahtunut jonkin kolmannen osapuolen tekemän selainlaajennoksen avulla, kuten Java, Flash tai Shockwave. WebGL:n tarkoitus on luoda standardi selaimille ja näin mahdollistaa monipuolista interaktiivista sisältöä selaimissa, mikä ei aiemmin ole ollut mahdollista. (1.)

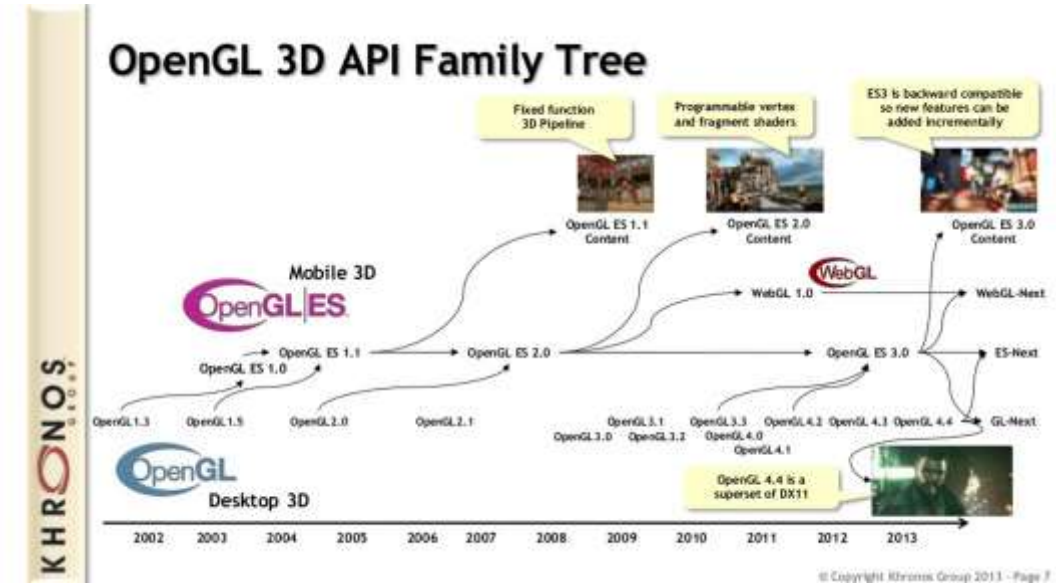
WebGL on Khronos Groupin kehittämä teknologia. Khronos on useiden yritysten muodostama ei-kaupallinen yhteisö, jonka päämääränä on kehittää ohjelmistorajapintoja ja työkaluja multimediasovellusten käyttöön. (2.) Yksi tällaisista rajapinnoista on myös tunnettu ja yleisesti käytössä oleva OpenGL-grafiikkakirjasto. OpenGL (Open Graphics Library) on laitteistoriippumaton (cross-platform) ohjelmointirajapinta, joka on suurin kilpailija Microsoftin Direct3D:lle. OpenGL:ää hyödyntävät niin 2D- kuin 3D-grafiikkaa käyttävät sovellukset monella eri alustalla. Esimerkiksi Linux-käyttöjärjestelmällä toimivat sovellukset käyttävät lähes poikkeuksetta OpenGL:ää hyödykseen. OpenGL:ää sanotaan graafisen teknologiateollisuuden perustaksi. (3.)

OpenGL:stä on myös kehitetty kevyempi versio OpenGL ES (embedded systems), joka on tarkoitettu grafiikkaohjelmoinnin rajapinnaksi laitteille, jotka tarvitsevat vain pelkistetysti graafisia prosessointiominaisuuksia. Näin ollen voidaan suunnitella yksinkertaisempia piirilevyjä, jotka täyttävät vain tiettyjä tarpeita, ja rajapinta saadaan pidettyä pienenä ja kevyenä, mikä tietenkin vähentää virrankulutusta ja laitteistovaatimuksia muiltakin osin. OpenGL ES on hyvin yleinen muun muassa matkapuhelimissa, tableteissa ja muissa pienissä laitteissa, jotka tarvitsevat rajatusti graafista prosessointia. (4.)

OpeGL:n luomaa pohjaa ja sen pelkistettyä OpenGL ES 2.0 -standardia hyödyntää myös WebGL, joka sitoo OpenGL:n rajapintaa käytettäväksi verkkoselaimissa JavaScript-kie-

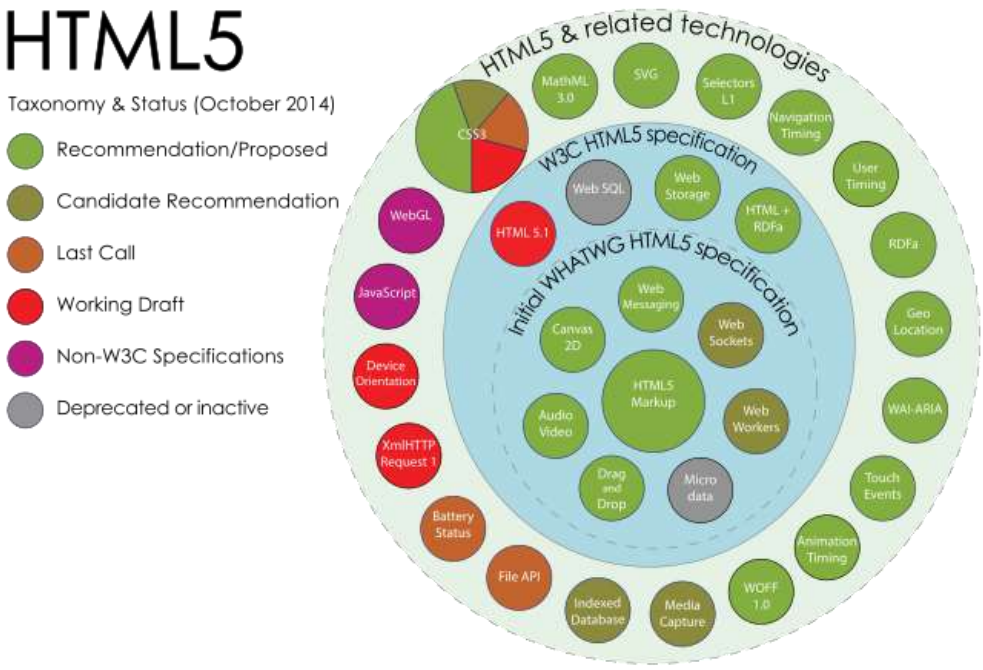
len avulla. (5.) Käytännössä WebGL on työkalupakki, joka käärii OpenGL:n toiminnallisuuden JavaScriptin sisälle, minkä avulla grafiikkaohjelmointia voidaan tehdä käyttämällä JavaScriptia ja sen syntaksia.

OpenGL on jakautunut eri osiin riippuen käyttöympäristöstä ja ominaisuuksien tarpeista. Kuvassa 1 nähdään aikajanalla OpenGL-rajapintaperheen versioita. Vuonna 2011 sulautettuja järjestelmiä varten kehitetystä OpenGL ES:stä on johdettu WebGL, joka tarjoaa laitteistokiihdytysominaisuuksia verkkoselaimella suoritettaviin sovelluksiin.



Kuva 1. OpenGL ja sen versioiden ja haarojen aikajanaa (6).

WebGL katsotaan osaksi moderneja web-teknologioita. Vaikka HTML5 tarkoittaa uusinta versiota HTML-kuvauskielestä (*markup language*), termiä HTML5 käytetään kuvan 2 mukaan myös kuvaamaan monia muitakin siihen liittyviä teknologioita, kuten CSS3-ulkoasumäärittelyjä, multimedian toisto-ominaisuuksia ja interaktiivisuutta muun käyttöjärjestelmän kanssa, kuten tiedostojen lähettäminen ja lataaminen *drag and drop -tyyppisesti,* niin kuin web-sovellus olisi osa muuta käyttöjärjestelmää ja sen ohjelmia, joita suoritetaan paikallisesti.



Kuva 2. Näkemyksiä siitä, mitä kaikkea HTML5 ja moderni web on (7).

HTML5:een liittyviä tekniikoita on tietenkin monta. Spesifikaation ulkopuolellakin on paljon toiminnallisuutta ja tekniikkaa, joiden päälle rakennetaan monipuolista liiketoimintaa Internetiä hyödyntävissä sovelluksissa.

2.2 Selaingrafiikan historiaa

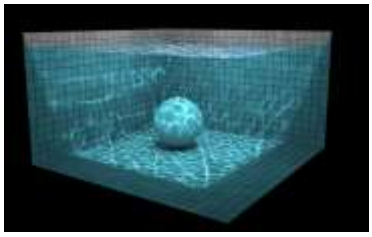
Aikaisemmin, vuosituhannen alussa reaaliaikaista grafiikkaa on saatu verkkoselaimiin erinäisten selainlaajennosten avulla. Eräitä laajasti käytössä olleita laajennoksia ovat muun muassa Java, Flash ja Shockwave. Näillä alustoilla tehtiin vuosituhannen alussa paljon pieniä pelejä ja animaatioita pelattavaksi ja nähtäväksi verkkosivuille. Tunnettuja verkkoselaimessa toimivia pelejä ovat esimerkiksi RuneScape (www.runescape.com) ja Habbo Hotel (www.habbo.fi), joita voisi sanoa pioneereiksi selainpelaamisen alalla. Molemmat toimivat verkkoselaimessa jo vuonna 2000 ja ovat vielä nykyäänkin toiminnassa.

Pisimmillään WebGL:n juuret ovat 1990-luvulla, jolloin grafiikkateknologian johtava yritys Silicon Graphics alkoi kehittää yhtenäistä grafiikkakirjastoa, jota hyödynnettäisiin muun muassa CAD-ohjelmistoissa. Tästä syntyi OpenGL 1.0, joka julkaistiin vuonna 1992. Siitä kasvoi useamman yrityksen yhteenliittymä, joka nykyisin tunnetaan nimellä Khronos Group, joka kehittää OpenGL:ää ja muita ohjelmointirajapintoja. (31.)

Ensimmäisiä yhtenäisten graafisten ominaisuuksien prototyyppejä selaimelle on kehittänyt Vladimir Vukićević työskennellessään Mozilla-säätiössä. Hän esitteli prototyyppiään 3D-canvas-nimisestä ominaisuudesta vuonna 2006, ja vuoden päästä sekä Mozilla- että Opera-selaimet olivat kehittäneet omat toteutuksensa tukeakseen tätä tekniikkaa. (32.)

OpenGL:llä ei kuitenkaan suoraan ole mitään tekemistä verkkoselainten kanssa. OpeGL:stä on jatkettu erillinen ominaisuuksiltaan riisutumpi rajapinta OpenGL ES, josta taas WebGL on johdettu. Tämän historian tunteminen on hyödyllistä, sillä WebGL-osaaja voi hyödyntää taitojaan muuallakin graafisten sovellusten kehityksessä. Vuonna 2009 Khronos aloitti WebGL-standardin kehityksen yhdessä monen Internet-alan yhtiön kanssa, muun muassa Google, Mozilla ja Opera. Ensimmäinen versio WebGL-standardista julkaistiin vuonna 2011. (33.)

Ensi kosketukseni minulle WebGL:ään oli vuonna 2011, kun löysin Evan Wallacen tekemän WebGL-demon, jossa esiteltiin valon taittumista simuloivaa tekniikka. Demossa on allas täynnä vettä ja veden pintaan voi luoda aaltoja. Veden pinnan epätasaisuus taittaa valoa, ja taittumisen intensiteettikohdat näkyvät kirkkaina kuvioina altaan pohjalla, kuten kuvassa 3 näkyy. Tämä demo oli siinä mielessä hämmästyttävä, että vielä nykyäänkin monet 3D-mallinnukseen ja renderöintiin tarkoitetut sovellukset välttävät tämänkaltaista valon taittumisen simulointia, sillä se on hyvin raskasta.



Kuva 3. Ewan Wallacen WebGL-Water-demo (9).

Wallacen demo kuitenkin pyöri reaaliajassa piirtäen monta kymmentä kuvaa sekunnissa. Demo simuloi valon taittumista nokkelilla tekniikoilla hyödyntäen näytönohjainta efektin luomiseen. (8.) Simulaation näyttävyydestä huolimatta se toimi tyydyttävästi viitisen vuotta vanhassa työkoneessani ilman erillistä näytönohjaintakaan.

2.3 WebGL-ohjelmointi ja ylätason kirjastot

Ohjelmointikielet ja -tekniikat koostuvat monesta kerroksesta. Kun puhutaan alemman kerroksen ohjelmarajapinnoista (low-level API), tarkoitetaan sellaisia ohjelmointikieliä tai tekniikoita, jotka ovat lähellä laitteistoa. Nämä ohjelmakoodit todennäköisesti ovat osa tai heti seuraavana laiteajurien jälkeen. Laiteajurit ovat tietenkin käyttöjärjestelmän jatkeena syöttämässä ja lukemassa dataa suoraan esimerkiksi näytönohjaimelta. (10.)

Korkeamman tason (high-level) ohjelmistot, ohjelmakoodit ja tekniikat ovat yleensä paljon selkeämpiä ymmärtää, ja ne on kehitetty alempien kerrosten päälle helpottamaan ja nopeuttamaan sovelluskehitystä abstrahoimalla monimutkaisia toimintoja. Yksinkertainen esimerkki on esimerkiksi useasta ohjelmasta löytyvä tallennustoiminto. Jotain tietoa halutaan tallentaa johonkin formaattiin tietokoneen kiintolevylle. Hyvin epätodennäköisesti jokainen sovellus sisältää oman ohjelmoidun toteutuksensa tiedostojärjestelmän ja kiintolevyn kanssa keskusteluun. Hyvin todennäköisesti nämä ohjelmat käyttävät jonkin ohjelmointikielen olemassa olevaa toimintoa, joka taas on rakennettu käyttöjärjestelmän tarjoamien kirjastojen päälle. (10.)

WebGL on edellä mainittu alatason ohjelmistorajapinta. Tämä tarkoittaa, että se on hierarkiassa lähellä fyysistä laitetta, eli tässä tapauksessa näytönohjainta. WebGL sisältää toimintoja näytönohjaimen laskentaominaisuuksien käyttöön, ja näin ollen sen opettelu ja käyttäminen korkeamman tason asioihin, kuten kolmen kuution piirtämiseen, on hyvin hankalaa. Jopa Khronoksen verkkosivuilla sanotaan, että opettelu ei ole heikkohermoiselle. (11.) Ohjelmoijan on tunnettava paljon monimutkaisia asioita, kuten varjostusohjelmien käyttö (*shaders*) ja matriisimatematiikka ja ymmärtää, miten näytönohjain vastaanottaa, varastoi ja prosessoi kolmiulotteista dataa. (Liite 1.) 3D-mallit koostuvat useista pisteistä, ja kolme pistettä muodostaa kolmion. Useampi kolmio muodostaa pinnan. Täytyy ymmärtää, miten tämän pinnan päälle voidaan piirtää ja kartoittaa tekstuuri, esimerkiksi kuvatiedosto, sekä osata ohjelmoida pinnan ominaisuuksia esimerkiksi valon heijastamisen suhteen. Tämänkaltaiset toiminnot vaativat huomattavan määrän matematiikkaa ja tehokkaita tapoja hyödyntää monimutkaisia konsepteja, kuten puskureita (*buffers*), koodin yhtäaikaista suoritusta (*parallel processing*) ja manuaalista muistinhallintaa näytönohjaimen omassa käyttömuistissa.

Otetaan käytännön esimerkki: kolmion piirtäminen. GitHubissa (github.com) on OpenGL ES 3 -rajapinnan esimerkki kolmion piirtämisestä. (12.) Koodi on myös tämän raportin

liitteenä nimellä *Koodiesimerkki 1.* Koodi on monimutkaista, mikä johtuu tietenkin siitä, että koodin on ohjeistettava monimutkaista virtapiiriä toimimaan niin, että data, jonka se käsittelee näyttää näytöllä kolmioilta. Koodinpätkässä luodaan varjostinohjelma ja liitetään se kolmion verteksijoukkoon. Hyvin monta monimutkaista asiaa pitää tapahtua, jotta saadaan yksinkertainen kolmio piirrettyä.

Kuvitellaan, että halutaan tehdä monipuolinen ja graafisesti näyttävä peli. Olisi typerää lähteä tekemään korkean tason toimintoja käyttäen vastaavanlaista koodia. Sen sijaan nokkela ohjelmoija rakentaa itselleen aputoimintoja ja työkaluja, esimerkiksi funktion, jolle voidaan syöttää yleisesti ymmärrettäviä arvoja. Liitteen 1 koodiesimerkki voidaan hyödyntää ja niin sanotusti *kääriä* korkeamman tason funktioksi *GLdrawTriangle(float x1, float y1, float x2, float y2, float x3, float y3).* Tämä keksitty funktio ottaa vastaan kuusi desimaalilukua, jotka määrittäisivät kolmion kolme pistettä kaksiulotteisessa tasossa. Funktio on kuvaava ja yksiselitteinen ja kätkee sisäänsä paljon monimutkaisempia toimintoja itse näytönohjaimen käskemiseen. Tässä piilee korkeamman tason ohjelmointikielien ja -kirjastojen ydinajatus.

WebGL:n päälle on kehitetty useita kirjastoja ja ohjelmistointegraatioita helpottamaan sisällön luontia ja pienentämään ohjelmoijien taakkaa kätkemällä monimutkaisen virtapiirin ohjelmointia korkeamman tason työkalujen alle (13). Tällaiset korkeamman tason kirjastot ovat hyvin tärkeitä sovelluskehityksen kannalta, sillä ne mahdollistavat ohjelmoijien keskittymisen suurempiin kokonaisuuksiin ja toimintoihin sen sijaan, että he käyttäisivät aikansa miettiäkseen, miten ladataan jonkin kappaleen väri näytönohjaimen muistiin tai miten mallin pyörittämisen matematiikka koodataan. Sen sijaan korkean tason kirjastot tarjoavat tällaisia toimintoja helppoina funktioina vaikkapa kappaleen pyörittämiseen *rotateObject(int objectID, float, x, float y, float z)* tai vaikkapa olio-ohjelmointityylisesti *object.setColor(Color.hex("fab6fa"));*

WebGL-kirjastot ovat lähes poikkeuksetta JavaScript-kirjastoja, jotka sisältävät ison joukon toimintoja ja funktioita nopeuttamaan sovelluksen ohjelmointia. Kirjastoissa on myös paljon eroja siinä, mitä kaikkia toimintoja ne sisältävät. Osa kirjastoista saattaa tarjota toimintoja pelkästään grafiikan piirtämiseen, kun taas jokin toinen kirjasto saattaa tarjota toimintoja myös ääniefektien, verkkoyhteyden käytön ja käyttäjäsyötön käsittelyyn. Jokin kirjasto saattaa olla toisen kirjaston päälle rakennettu ja laajentaa toiminnallisuutta tai yhdistää muitakin toimintoja. Esimerkki tällaisesta on *WhitestormJS (www.whsjs.io)*, joka perustuu *three.js*-kirjastoon (www.threejs.org) ja yhdistää *bullet*-fysiikkakirjaston. (14.)

Nämä kirjastot eivät tietenkään ole mitään sovelluksia eivätkä tarjoa graafisia käyttöliittymiä sisällön, kuten 3D-mallien, valojen tai materiaalien tuottamiseen. Saatavilla on myös laajempia kehitysympäristöjä, jotka tarjoavat laajemmin ominaisuuksia, kuin pelkästään ohjelmointirajapinnan WebGL:ää varten. Niitä käsitellään luvuissa 2.4 ja 2.5.

ThreeJS-kirjasto on tunnettu, mrdoob-nimisen ohjelmoijan aloittama avoimen lähdekoodin projekti, joka on edelleen aktiivisessa kehityksessä. Projekti on hyvin suosittu. Siihen on tehty yli 18 000 lisäystä tai muutosta, sitä on julkaistu 76 versiota ja sen kehittämiseen on osallistunut lähes 800 henkilöä. (16.) Kirjaston tasosta kertoo sen käyttö suurienkin yhtiöiden toimesta. Esimerkiksi Ford, Renault ja Porsche ovat kehittäneet kirjaston avulla visualisointisovelluksen autojen ja varustelun esittelemiseksi interaktiivisella tavalla. Muutkin suuret yhtiöt ovat hyödyntäneet kirjastoa markkinointiin, kuten Apple, Panasonic ja Autodesk. Autodesk hyödyntää WebGL:ää myös omassa A360-ohjelmassaan. (15.)

ThreeJS:n etusivulla on esillä kattava lista tuotantoja, jotka hyödyntävät sitä, mikä varmasti auttaa kirjaston näkyvyyteen. ThreeJS tarjoaa myös erikseen sivustot kirjaston dokumentaatioon ja pitkän listan yksityiskohtaisia koodiesimerkkejä. Koodiesimerkistä 2 voidaan katsoa, kuinka kirjasto otetaan käyttöön lisäämällä HTML-dokumentin *head*-osioon linkki ThreeJS-kirjaston JavaScript-tiedostoon. Tämän jälkeen voidaan kirjoittaa JavaScript-koodia ja hyödyntää threeJS-kirjaston toiminta 3D-sisällön luomiseksi verkkosivulle. Koodissa alustetaan *scene* ja perspektiivikamera sekä WebGL-renderöijä. Geometriaa voidaan luoda selkeillä komennoilla, kuten esimerkkikoodissa näkyy. Geometria ja materiaali luodaan erikseen, ja ne liitetään ja muodostetaan *pinnaksi* (mesh).

```
<html>
<head>
   <title>My first three.js app</title>
   <style> body { margin: 0; } canvas { width: 100%; height: 100% } </style>
</head>
<body>
  <script src="js/three.js"></script>
  <script>
     var scene = new THREE.Scene();
     var camera = new THREE.PerspectiveCamera( 75, window.innerWidth/window.inner-
Height, 0.1, 1000 );
     var renderer = new THREE.WebGLRenderer();
     renderer.setSize( window.innerWidth, window.innerHeight );
     document.body.appendChild( renderer.domElement );
     var geometry = new THREE.BoxGeometry( 1, 1, 1 );
     var material = new THREE.MeshBasicMaterial( { color: 0x00ff00 } );
     var cube = new THREE.Mesh( geometry, material ); scene.add( cube );
     camera.position.z = 5;
     var render = function () { requestAnimationFrame( render ); }
     cube.rotation.x += 0.1; cube.rotation.y += 0.1;
```

```
    renderer.render(scene, camera); };
    render();
</script>
</body>
</html>
```

Koodiesimerkki 2. ThreeJS-esimerkkikoodia HTML-tiedostossa. Koodi piirtää vihreän pyörivän kuution (17).

BabylonJS (www.babylonjs.com/) on ThreeJS:n ohella hyvin kattava ja monipuolinen JavaScript-kirjasto WebGL-tuottamiseen. Kuten ThreeJS, BabylonJS on vain ohjelmointirajapinta WebGL:n grafiikkaominaisuuksiin eikä itsessään mikään sovellus sisällön tuottamiseen. Kaikki 3D-mallit ja materiaalit on tehtävä kolmannen osapuolen ohjelmistoilla, kuten 3DS Max ja Photoshop. BabylonJS-kirjasto on saatavilla Githubista, missä siihen on tehty lähes 6 000 muutosta ja lisäystä, sillä on 45 julkaistua versiota ja reilu 100 osallistunutta kehittäjää. (19.) Kirjasto on aktiivisessa kehityksessä, ja viimeisimmät muutokset on tehty muutaman päivän sisällä. Eräitä merkittäviä tahoja, jotka ovat käyttäneet hyödyksi BabylonJS:ää, ovat muun muassa Dolby Digital, Xbox.com ja National Geographics. (18.)

BabylonJS tukee monia kehittyneitä graafisia ominaisuuksia, kuten fyysistä renderöintimallia (PBS), dynaamisia varjoja, pehmeitä varjoja, instansointia, volumetrista valaistusta ja renderöinnin jälkeisiä efektejä (post-processing). BabylonJS tukee myös fysiikkalaskentaa erillisen cannon.js-fysiikkakirjaston avulla. Kokonainen lista ominaisuuksista interaktiivisine esimerkkeineen löytyy kirjaston sivuilta. (18.)

Koodiesimerkissä 3 on mallikoodi BabylonJS:n käytöstä. HTML-dokumenttiin liitetään JavaScript-kirjastot, ja sen jälkeen voidaan käyttää niitä 3D-toimintojen tekemiseen kuten ThreeJS:kin. Huomattava seikka on, miten samankaltainen syntaksi BabylonJS:ssa on ThreeJS:ään verrattuna. Käyttöönoton prosessikin on lähes täysin samanlainen.

```
<!DOCTYPE html>
<html>
<head>
   <meta http-equiv="Content-Type" content="text/html" charset="utf-8"/>
   <title>Babylon - Getting Started</title>
   <script src="babylon.2.3.debug.js"></script>
   <style>
      html, body {
         overflow: hidden;
         width   : 100%;
         height  : 100%;
         margin  : 0;
         padding : 0;
```

```
      }
      #renderCanvas {
         width   : 100%;
         height  : 100%;
         touch-action: none;
      }
   </style>
</head>
<body>
   <canvas id="renderCanvas"></canvas>
   <script>
      window.addEventListener('DOMContentLoaded', function(){
         // get the canvas DOM element
         var canvas = document.getElementById('renderCanvas');
         // load the 3D engine
         var engine = new BABYLON.Engine(canvas, true);
         // createScene function that creates and return the scene
         var createScene = function(){
            // create a basic BJS Scene object
            var scene = new BABYLON.Scene(engine);
            // create a FreeCamera, and set its position to (x:0, y:5, z:-10)
            var camera = new BABYLON.FreeCamera('camera1', new BABYLON.Vector3(0, 5,-
10), scene);
            // target the camera to scene origin
            camera.setTarget(BABYLON.Vector3.Zero());
            // attach the camera to the canvas
            camera.attachControl(canvas, false);
           // create a basic light, aiming 0,1,0 - meaning, to the sky
            var light = new BABYLON.HemisphericLight('light1', new BABYLON.Vec-
tor3(0,1,0), scene);
            // create a built-in "sphere" shape; its constructor takes 5 params: name, width,
depth, subdivisions, scene
            var sphere = BABYLON.Mesh.CreateSphere('sphere1', 16, 2, scene);
            // move the sphere upward 1/2 of its height
            sphere.position.y = 1;
            // create a built-in "ground" shape; its constructor takes the same 5 params as
the sphere's one
            var ground = BABYLON.Mesh.CreateGround('ground1', 6, 6, 2, scene);
            // return the created scene
            return scene;
         }
         // call the createScene function
         var scene = createScene();
         // run the render loop
         engine.runRenderLoop(function(){
            scene.render();
         });
         // the canvas/window resize event handler
         window.addEventListener('resize', function(){
            engine.resize();
         });
      });
   </script>
</body>
</html>
```

Koodiesimerkki 3. Dokumentaatiosta BabylonJS-sivulta (20).

Yksi huomioitava kehitysalusta pelimoottorien rinnalle on *Blend4Web (*www.blend4web.com/en/*),* jota esimerkiksi NASA käytti Curiosity-Mars-mönkijän esittelyyn (21). Blend4Web on erikoinen lajissaan, sillä se laajentaa Blender 3D -mallinnusohjelman (*www.blender.org/*) toiminnallisuutta siten, että kaikki Blenderissä tehty sisältö voidaan puskea suoraan siitä ulos suoritettavaksi selaimeen. Blender on pääasiassa täysiverinen työkalu 3D-mallinnukseen, -efektisimulointiin ja perinteiseen kuva- tai animaatiorenderöintiin. Graafisen verkkosisällön toimintoja ja sovelluksen logiikkaa kehitetään Blenderissä node-järjestelmää käyttäen (27) ja kirjoittamalla koodia JavaScriptillä. Etuna on Blenderin tarjoama graafinen käyttöliittymä ja iso määrä toimintoja 3D-ympäristöjen, animaatioiden ja materiaalien tekemiseen.

Tämän opinnäytetyön tarkoituksen puolesta Blend4Web on mielenkiintoinen, sillä se tukee Blenderin avulla useaa eri 3D-malliformaattia. Myös liitännäisenä 3D-mallinnusohjelmaan se tarjoaa kattavammin työkaluja ja käyttäjäystävällisemmän lähestymistavan WebGL-maailmaan kuin ThreeJS ja BabylonJS. Toisaalta on myös täysin mahdollista vaikkapa luoda 3D-malleja Blenderissä ja viedä ne jälkeenpäin ThreeJS-sovellukseen.

2.4 Unity- ja Unreal Engine -pelimoottorit

Lukuisien ohjelmointikirjastojen lisäksi myös jotkut pelimoottorit tarjoavat WebGL-tuen. Pelimoottorien etu 3D-sisällön luomiseksi on se, että sisältävät vielä enemmän ominaisuuksia ja työkaluja kuin pelkkä grafiikkakirjasto. Monet pelimoottorit tarjoavat myös graafisen käyttöliittymän, kuten kuvassa 4 näkyy, ja paljon toimintoja, mikä auttaa kehittäjiä keskittymään omaan tuotokseensa ja antaa pelimoottorin sekä sen työkalujen hoitaa loput. Pelimoottori yleensä kattaa valtavan määrän tekniikoita, kuten grafiikka, fysiikka, verkkoyhteys, ääni ja efektit, animaatio, käyttöliittymä, tekoäly ja navigaatio, virtuaalitodellisuus ja kääntäminen suoritettavaksi sovellukseksi.

Kuva 4. Unity-pelimoottorin editori, jossa sisältö ja peliominaisuudet luodaan (26).

Unity (https://unity3d.com/) on pelimoottori, jonka ensimmäinen versio julkaistiin kesäkuussa 2005. Siitä lähtien sitä on kehitetty huomattavasti, ja tällä hetkellä Unityllä tehtyjä sovelluksia on yli 2,5 miljardissa laitteessa. (22.) Unity on siitä mielenkiintoinen pelinkehitysympäristö, että sen avulla voi luoda pelejä todella monelle eri alustalle, kuten PC:lle, Linuxille tai Playstationille (24). Ottaen huomioon, että lähes kaikki nykyaikaiset selaimet jopa mobiililaitteilla tukevat WebGL:ää, Unity on todella kilpailukykyinen alusta minkä tahansa kokoluokan projektin rakentamiseen, ja se tukee myös useita VR-laitteita. Referenssit ja dokumentaatio ovat todella kattavia ja hyvin tehtyjä. Unityn ekosysteemiä laajentaa huomattavasti mahdollisuus kenen tahansa julkaista sisältöä tai toiminnallisuuta Unityn kauppaan, jopa ilmaiseksi. (23.)

Unity sopii niin yksittäisille pelinkehittäjille kuin isoille studioillekin. Sen oma editori on ohjelmoitavissa ja laajennettavissa (25) ja näin kehittäjät voivat kehittää lisää täysin integroituja, yrityksen sisäisiä työkaluja. Vielä kun melko uusi VR-laitteiden rajapinta WebVR (https://webvr.info/) kypsyy hieman, on hyvin helppo nähdä, miten tulevaisuuden VR-sovellukset ja pelit voivat toimia vaikkapa SaaS-periaattella suoraan selaimessa.

Unreal Engine (www.unrealengine.com/) on Epic Gamesin kehittämä ja ylläpitämä pelimoottori ja tuotantoalusta. Tällä hetkellä pelimoottorista on saatavilla versio neljä. Unreal on tunnettu sen graafisesta näyttävyydestä ja hyvästä suorituskyvystä. Unreal ja Unity ovat hyvät kilpailijat keskenään, ja pintapuolisesti ne muistuttavakin toisiaan. Niillä molemmilla on suuri kehittäjäpohja, molemmilla alustoilla on tehty kaupallisesi menestyneitä pelejä, niiden ekosysteemi on hyvin samanlainen ja molemmat tarjoavat mahdollisuuden kehittää pelejä ja sovelluksia monelle eri alustalle.

WebGL-kehityksestä on dokumentaatiota Unreal Enginellä. Tällä hetkellä se vaikuttaa olevan hieman jäljessä Unityä ja Unityn tarjolla olevaa dokumentaatiota. WebGL-osio Unrealin dokumentaatiossa on vielä hyvin tynkä, ja kehitysympäristön ilmoitetaan olevan vielä kesken. Tästä syystä en valitsisi Unrealia tuotantoasteen WebGL-sovelluskehitykseen. (28.)

2.5 WebGL-kehityksen haasteet ja kehitysalustojen ominaisuudet

WebGL-kehitys on ottanut muutaman vuoden aikana aimo harppauksia, ja yksi kehittäjää askarruttavista asioista onkin, minkä kirjaston tai kehitysympäristön avulla lähtisi tekemään tuotostaan. Valinnanvaraa on paljon, ja huomioon pitää ottaa projektin koko, saatavilla olevat ja tarvittavat työkalut ja se, kuinka kriittistä on saada apua työkalujen kehittäjiltä ongelmatilanteissa. ThreeJS vaikuttaa sopivalta pienien demojen tekoon, missä ei tarvitse käsitellä kompleksia geometriaa. Avuksi voi toki ottaa jonkin 3D-mallinnusohjelman. Suurempia projekteja varten valitsisin jonkin kokonaisvaltaisemman ja laajennettavissa olevan alustan, kuten Unityn, jonka tuki ja kaupallinen palvelu ovat kohdallaan. Isompi projekti tarvitsee myös enemmän työtä sisällön- ja projektinhallintaan, missä Unity tai Unreal erottuvat joukosta edukseen. Kumpikaan ei tue vielä VR-laitteistoa WebGL-sovelluksissa, sillä WebVR-rajapinta on vielä lapsenkengissään: sitä tuetaan ainoastaan muutaman selaimen alpha-versioissa. (29; 30.)

Kehitysalustan valinta on tärkeä asia, sillä se määrittää myös projektin tulevaisuutta. Pitää ottaa huomioon, onko tulevia ominaisuuksia mahdollista kehittää nykyisellä versiolla ja onko alustaan odotettavissa päivityksiä. Vertailussa katson käyttöönottoa, ominaisuustarjontaa ja työkalujen saatavuutta. Lyhytkatseisesti enemmän ominaisuuksia sisältävä vaihtoehto ei tietenkään ole paras, vaan pitää myös miettiä alustan ja työkalujen sopivuutta tilaajan näkökulmasta. On tärkeää ymmärtää, että kehitettävä visualisointityökalu ei tule tilaajayrityksen sisäiseen käyttöön vaan osana tuotetta, jota käyttää iso kirjo loppukäyttäjiä, joiden tekninen tietotaitotaso vaihtelee paljon. Lopputuotteen pitää olla helppokäyttöinen. Jätän vertailussa Unreal Enginen pois, koska sen tuki WebGL-kehitykselle on liian puutteellinen.

ThreeJS on tämän projektin kannalta houkutteleva, sillä se tukee nykyisellään FBX (FilmBox) -mallien tuontia suorituksen aikana. Muitakin formaatteja on käytettävissä. (34.) Tämä on tärkeä ominaisuus, mikä puuttuu Unitystä. Unityssä prosessi menee niin,

että ensin kaikki mallit ja muut tuodaan projektiin, ja sen jälkeen projekti paketoidaan ulos WebGL-muotoon. Tämä on melko kätevää, mutta projektin kasvaessa yli 10 megatavuun alkaa latausaika kasvaa. On myös hyvin epäkäytännöllistä ladata koko projektia, jos siitä käytetään vain yhtä mallia ja materiaalia. Unityssä tämä ongelma ratkaistaan AssetBundle-nimisellä tekniikalla. AssetBundleja käsitellään laajemmin luvussa 3.2.

ThreeJS-kirjaston käyttöönotto on nopeampaa kuin Unityn. Se on minimoituna noin puoli megatavua, ja sen käyttöönotto kestää alle minuutin. Kirjasto linkitetään HTML-sivulle, minkä jälkeen voit alkaa ohjelmoida sisältöä. Tässä on myös ThreeJS:n suurin puute. Jos haluaa kameran tiettyyn pisteeseen, se on koodattava kyseiseen pisteeseen. Samoin kaikki muukin toiminnallisuus pitää ohjelmoida itse, ja äkkiä tämä käy vaivalloiseksi. Jälkeenpäin jos haluaa muuttaa materiaalia, pitää muutokset koodata tai muokata tekstuurikuvia kuvankäsittelyohjelmalla. ThreeJS:lle on kyllä olemassa kevyt editori selaimessa (35), mutta se on ominaisuuksiltaan hyvin alkeellinen, enkä saanut malliani näkyviin siinä.

BabylonJS on hyvin ThreeJS:n kaltainen. Se on kirjasto, eikä työkaluja ole saatavilla, paitsi *playground*iksi nimetty kevyt editori testaamiseen. Pidän enemmän BabylonJS:n dokumentaatiosta, ja ilmeisesti se on jäsennelty siten, että esimerkkejä voi tehdä samalla playgroundin avulla, mikä tehostaa oppimista. Käyttökokemuksen osalta valitsisin ThreeJS- tai BabylonJS-kirjaston yksinkertaisten graafisten sovellusten tekemiseen selaimelle. Unityä käytetään sen editoriohjelmalla, josta lopputulos pakataan ulos esimerkiksi WebGL-muotoon. Unityn kanssa aloittaminen on hieman raskaampaa kuin aiempana käsiteltyjen kirjastojen. Unity pitää ladata ja asentaa, ja koodausta varten pitää asentaa erikseen jokin koodieditori, mielellään kattava IDE, kuten Visual Studio (www.visualstudio.com/) tai Monodevelop (www.monodevelop.com/). Näiden ohjelmien yhteenlaskettu koko on monta gigatavua, minkä takia asentaminen kestää hitaalla Internet-yhteydellä toista tuntia. Myös projektin vienti WebGL-muotoon selaimella testattavaksi kestää pahimmillaan useita minuutteja. (36; 37.)

Unityn parhaita puolia ovat sen kattava dokumentaatio ja loistavat selostetut esimerkkivideot. Vaikkakin kehittäjällä on mahdollisuus käyttää JavaScript-tyylistä UnityScriptiä koodikielenä, suosittelen vahvasti kirjoittamaan koodia C#:llä. C# on kielenä hyvin Javakielen kaltainen ja kehitystyökalut sille ovat erinomaiset. Täysi oliopohjainen kehitystuki, tarvittavien nimiavaruuksien automaattinen lisäys, koodin automaattitäydennys ja metodien ylikirjoittaminen ovat hyviä esimerkkejä kehittämistä avustavista toiminnoista. En

ole ainakaan tietoinen yhdestäkään IDE:stä, joka tarjoaisi yhtä laadukkaat kehitystyökalut JavaScriptille. JetBrainsin (www.jetbrains.com/) kaupalliset kehittimet ovat saaneet kehuja, mutta en ole käyttänyt niitä. Unityn kanssa ohjelmoin C#:lla, sillä Visual Studio on monipuolinen työkalu. Visual Studiota tarvitaan, jos kehittää sisältöä Microsoftin Hololenselle (37). Vaihtoehtoinen C#-kehitysympäristö Monodevelop on saatavilla MacOS X:lle ja Linuxille.

Unityssä on myös paljon sovelluslogiikan kehittämistä helpottavia toimintoja. Kappaleet, joilla pitää olla fysiikkaa, ovat helposti luotavissa liittämällä niille fysiikkakomponentti. Törmäystunnistus on myös tehtävissä komponenteilla, pelimoottori hoitaa loput. Törmäystapahtumiin pääsee koodissa helposti käsiksi. Lisäksi Unityssä on sisäänrakennettua verkkopelitoiminnallisuutta tukevia komponentteja, jos sellaisiin on tarvetta. Unityssä on myös helppo testata kehitysvaiheen eri iteraatioita, sillä editorissa voi siirtyä pelitilaan suoraan play-napista. (38.)

Selaimessa testaamiseksi Unity-projekti pitää editorista puskea ulos WebGL-muotoon. Asetuksista riippuen WebGL-muotoisen projektin ”*buildaaminen”* eli koodin kääntäminen, kestää noin minuutista useampaan minuuttiin. Tähän vaikuttaa projektin koko, 3D-mallien määrä ja yksityiskohtaisuus, tekstuurien koko ja määrä. Tuotantoversio on tietenkin syytä pitää mahdollisimman pienenä, jotta latausajat pysyvät myös lyhyinä. Pakkaaminen hidastaa buildaamista, joten sitä ei kannata käyttää kehityksen aikana. Projekti latautuu tietenkin paikallisesti, eikä Internetin välityksellä. Tiedot voidaan ensimmäisen latauskerran jälkeen asettaa välimuistiin. Koko sovellusta ei näin tarvitse ladata kokonaan uudestaan käyttökertojen välillä.

3 Visualisointityökalun suunnittelu

3.1 Unityn valinta WebGL-kehitysalustaksi

Suurimmat syyt Unityn valitsemiseen kehitysalustaksi ovat sen monipuoliset työkalut ja hyvä monen alustan tuki. Insinöörityön tilaajayrityksessä on kehitetty ohjelmia Unityllä ennenkin, joten valinta on lähes itsestään selvää. Kaikista vaihtoehdoista Unity tukee parhaiten yrityksen nykyistä osaamista. Yrityksen 3D-mallintajat osaavat myöskin käyttää Unityn perusominaisuuksia.

Työssä tehtävä työkalu ei ole monimutkainen, joten ohjelmointikirjastoa hyödyntämällä voisi työkalun myös kehittää. Tarkoitus on tehdä sen käyttämisestä loppuasiakkaalle mahdollisimman helppoa ja jatkokehityksestä ja uusien toimintojen tekemisestä sulavaa. Tietenkin yrityksessä halutaan, että tarpeen tullen toimintoja voidaan laajentaa. Unityn tarjoamien ominaisuuksien määrä on sellainen, että kaikki kuviteltavissa olevat tarpeet voidaan täyttää. Unityn haaste on, että se ei kaikista ominaisuuksistaan huolimatta tue mallien tuontia suorituksen aikana. Se on monimutkainen toiminto, jonka kehittäminen itse tyhjästä olisi erittäin haasteellista. Katsoin ThreeJS:n FBX-lataajaa, ja se oli 3 241 riviä pitkä koodi. (39.)

Yritykseen ostettiin UniFBX-lataaja (42), jota on tarkoitus käyttää FBX-mallien lataamiseen verkon välityksellä. Toinen käytettävä formaatti on OBJ, jolle on saatavilla maksullinen ja ilmainen lataaja (40; 41). Vielä tässä kohtaa en tiedä, kehitetäänkö tuki yhdelle formaatille vai molemmille. Tämän insinöörityön puitteissa testaan ja vertaan kyllä molempia ja vielä Unityn AssetBundle-tekniikkaa. Ensisijaisesti valittiin FBX-lataaja, sillä FBX on yleinen formaatti esimerkiksi Autodeskin ohjelmissa ja käytettävissä monissa CAD-ohjelmistoissa (Computer Aided Design). Näin sovellusta käyttävän tahon on helppo vain parilla klikkauksella tallentaa oma mallinsa FBX-muotoon ja käyttää sitä visualisointityökalun kanssa.

3.2 3D-mallien käyttäminen Unityssä

Unityn prosessi projektien kokoamiseen tapahtuu editorissa. Projektin kansiorakenteeseen tuodaan kaikki tekstuurit, äänitiedostot ja 3D-mallit, jotka kaikki käännetään toimimaan Unity-pelimoottorin tarvitsemalla tavalla. Unity voi ottaa 3D-malleja vastaan neljällä eri formaatilla, jotka ovat FBX-, MAX-, OBJ- ja blend-tiedostoformaatit (43). 3D-malleista voi valita tuontiasetusten perusteella niiden skaalan ja muita asetuksia.

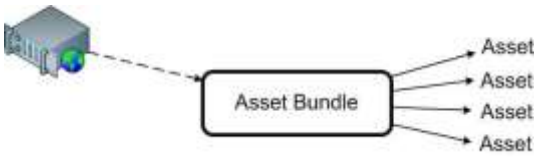
FBX-formaatti on Autodeskin omistama (44) hyvin yleinen tiedostoformaatti, joka varastoi useita 3D-ympäristön ominaisuuksia, kuten geometriaa, valoja, kameroita ja animaatioita. Autodesk julkaisee FBX-formaatin kehitysympäristön eli SDK:n (Software Development Kit), joka tarjoaa työkalut FBX-luku- ja kirjoitustoimintojen tekemiseen muihinkin 3D-sisällönluontiohjelmistoihin (45). FBX-formaatti onkin tuettuna useimmissa 3D-mallinnustyökaluissa, kuten Autodeskin 3Ds Maxissa, Mayassa, Maxonin Cinema 4D:ssä ja Blender-säätiön Blenderissä.

OBJ-tiedostoformaatti on avoin formaatti ja yleisesti tuettuna 3D-mallinnusohjelmissa, kuten Blenderissä ja 3D Studio Maxissa. Sitä myös tukevat monet WebGL-kirjastot, kuten molemmat aikaisemmin esitellyt ThreeJS ja BabylonJS. OBJ-formaatti varastoi ainoastaan 3D-geometriaan liittyvät asiat, kuten verteksit eli pisteet 3D-avaruudessa, pisteistä muodostuvat pinnat (*faces*) ja normaalit sekä UV-koordinaatit tekstuureita varten (46). OBJ-formaatti sopii silti hyvin tässä insinöörityössä tehtävään työkaluun. Unityyn on saatavilla ilmainen työkalu OBJ-formaattisten tiedostojen lataamiseen suorituksen aikana, joten pidän formaattia mielenkiintoisena vaihtoehtona. Työkalun käytettävyyden kannalta ei olisi ollenkaan huono asia, jos se tukisi toistakin yleisesti käytössä olevaa formaattia.

Max-tiedostoformaatti on Autodeskin *3D Studio Maxin* (www.autodesk.fi/products/3ds-max/overview) tiedostoformaatti. Max-tiedosto kattaa koko projektin 3D-ympäristön: kaikki objektit, materiaalit, renderöintiasetukset ja niin edelleen. Max-tiedostoja voi myös tuoda Unityyn ja saadaan eriteltyä objektit, niiden animaatiot ja ympäristön valoineen (48). Hyvä puoli tässä formaatissa on, että sen voi tallentaa suoraan Unityn projektikansioon, ja muutosten tekeminen 3DS Maxissa päivittyy myös Unityn projektiin nopeasti.

Myös Blender 3D-mallinusohjelman tiedostoja voi tuoda suoraan Unity-projektiin, ja jos muuttaa jotain 3D-mallissa, se päivittyy myös Unityyn. Blender-tiedostoformaatin tuki Unityssä ei ole oma erillinen tuki, vaan pinnan alla se toimii käyttäen Blender FBX exporteria. (47.)

AssetBundlet ovat Unityn sisäänrakennettu ominaisuus varastoida dataa ja ladata sitä suorituksen aikana, kun niitä tarvitaan. AssetBundlet luodaan Unityn editorissa valitsemalla halutut *assetit* johonkin nimettyyn AsseBundleen, joka on käytännössä paketti erilaisia sisällön osia, kuten äänitiedostoja tai 3D-malleja. Valmis AssetBundle viedään johonkin ulkoiseen varastoon, esimerkiksi FTP-palvelimelle, mistä se on ladattavissa. Ohjelman suorituksen aikana voidaan kuvan 5 esittämällä tavalla ladata AssetBundleja verkkoyhteyden avulla ja purkaa ne käyttökelpoisiin osiin, kuten ääniefekteihin tai 3D-malleiksi ja niiden materiaaleiksi. AssetBundlejen käyttöä varten on ohjelmoitava itse toimintoja, mutta Unityn dokumentaatiossa on hyviä esimerkkejä.



Kuva 5. AssetBundlejen latausprosessi palvelimelta käyttökelpoisiksi osiksi. Luotaessa AssetBundleja prosessi on käänteinen (49).

Latausaikojen lyhentämiseksi AssetBundleja voi pakata LZMA- tai LZ4-pakkausalgoritmilla (50). Pakkaaminen pienentää tiedoston kokoa huomattavasti ja näin lyhentää latausaikaa ja verkkoyhteyden kaistankäyttöä. LZMA-algoritmi on näistä kahdesta parempi pakkaamaan dataa, ja se on oletus AssetBundlejen pakkamiselle. LZMA-algoritmin huono puoli on sillä pakatun tiedon suhteellisen pitkä purkuprosessin aika (50). Verkon välityksellä ladatun tiedon latausajan lisäksi purku ottaa oman aikansa ja näin vaikuttaa kokonaisaikaan, kunnes tarvittava tieto on saatu purettua käyttökelpoiseen muotoon. LZMA-algoritmia käytettäessä pitää huomioida, että koko tietopaketti pitää purkaa, ennen kuin siitä voidaan lukea osia (50). Tämä saattaa hidastaa toimintaa merkittävästi esimerkiksi tilanteessa, jossa AssetBundleen on sisällytetty paljon eri 3D-malleja ja purettaessa sieltä halutaan ottaa 3D-ympäristöön vain yksi tai muutama malli, mutta koko paketti on purettava ensin.

Vaihtoehto AssetBundlejen pakkaamiselle on LZ4-algoritmi. Verrattuna LZMA-algoritmiin LZ4-algoritmilla pakattu AssetBundle on suuremman kokoinen kuin LZMA-algoritmilla kompressoitu. LZ4:n etuna algoritmi pystyy käsittelemään pakattua dataa *paloina* (chunks) ja tämä auttaa siihen, että pakatusta AssetBundlesta voidaan ottaa tarvittava malli tai muu tiedosto nopeammin kuin LZMA-algoritmilla pakatusta tiedostosta (50). Pakattua dataa ei siis tarvitse purkaa kokonaan, mikä on suuri etu ladattaessa tiettyä mallia AssetBundlesta, johon on paketoitu useampi malli. Assetbundleja voi käyttää myös pakkaamattomana. Tämä johtaa suuriin tiedostokokoihin, mutta tiedostojen käsittelyaika pienenee, kunhan tiedosto on ensin ladattu. (50.)

Tiedon nopeaan saatavuuteen voidaan vaikuttaa siirtämällä ladatut tiedot välimustiin. Tämä nopeuttaa ohjelman suoritusta ja latausaikaa seuraavalla kerralla, kun samoja AssetBundleja ei tarvitse ladata uudestaan. Välimuistiin siirrettyä dataa voi myös pakata ja näin pienentää säilytyksen kokoa. Kun latauksessa otetaan vastaan dataa, Unity purkaa sen ja pakkaa uudestaan LZ4-algoritmilla, kunnes lataus on valmis. Pakattu tieto säästää noin 40–60 % tilaa verrattuna pakkaamattomaan tietoon. Vielä tässä vaiheessa en tiedä, kuinka suuria malleja kehitettävässä työkalussa olisi tarkoitus käyttää, mutta saattaa olla tarpeellista jossakin kohtaa miettiä pakkauksen käyttöä mallien varastointiin ja verkon välityksellä siirtämiseen. (50.)

AssetBundlet ovat erittäin käyttökelpoinen ominaisuus datan varastoinnille palvelimelle. AssetBundle-tekniikka auttaa pienentämään julkaistavan ohjelman kokoa, kun kaikkea sisältöä ei tarvitse sisällyttää julkaisuun. Otetaan vaikka esimerkkinä suuri selaimessa toimiva peli. On epäkäytännöllistä ladata kymmeniä gigatavuja dataa kerralla muistiin. Tavallisesti ohjelmat varastoivat dataa kiintolevylle, AssetBundlet verkkoon, joista ladataan suorituksen aikana vain tarpeelliset asiat muistiin.

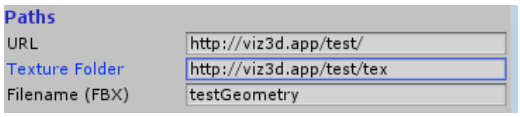
3.3 3D-mallien tuontikoodien tutkiminen

Keskustelin työpaikalla eri vaihtoehdoista ja toimintatavoista työkalun toteutuksessa. AssetBundlet olisi saatavilla oleva tekniikka, mutta se ei valitettavasti ole sopiva kehitettävän työkalun tarpeisiin. AssetBundleja käyttämällä, yrityksen pitäisi saada asiakkaan mallit ensin ja prosessoida ne Unityn läpi AssetBundleiksi ja sen jälkeen laittaa palvelimelle asiakkaan konfiguraattorisovellukseen. Asiakkaan pitää itsenäisesti saada liitettyä 3D-malli esimerkiksi johonkin tuotteeseen suoraan konfiguraattorisovelluksessa ilman,

että yrityksen 3D-mallintajien tarvitsee käyttää aikaa mallin muokkaamiseen. Sovelluksen ja visualisointityökalun on oltava yksinkertainen. Käytännön tasolla olisi parasta, jos loppukäyttäjä voisi vain osoittaa 3D-mallin tiedoston. Tämä tiedosto tallennettaisiin palvelimelle, ja viittaus olemassa olevasta tiedostosta kirjattaisiin tietokantaan.

Testasin FBX-lataajaa ensin Unityn editorissa, missä lataus tuntui toimivan hyvin ja FBX-lataaja pystyy lataamaan myös tiedostopolusta kiintolevyltä. Kiintolevyltä lataaminen ei tietenkään ole mahdollista selaimessa, ja muutenkin tarkoitus on ladata malleja verkon välityksellä palvelimelta. Pystytin virtuaalipalvelimen omalle koneelleni ja asensin yrityksen sovellusprojektin, jolle visualisointityökalua on tarkoitus kehittää.

FBX-lataaja käyttää asetuksissaan kolmea tekstikenttää tiedoston määrittelemiseen. Polku, jossa tiedosto on, polku, jossa on materiaalit ja mikä on tiedoston nimi. Polkuun piti jättää tyhjä tila tiedostonimelle, ja tiedoston nimi pitää kirjoittaa ilman päätettä kuten kuvasta 6 näkee. Sama pätee web-osoitteisiin. URL:iin pitää jäättä tila erikseen tiedostonimelle. FBX-lataaja kyllä täyttää sen automaattisesti.



Kuva 6. FBX-lataajan tiedostopolkuasetukset.

Testauksessa OBJ-lataaja osoittautui tukevan vain tiedostojen lukemista kiintolevyltä. WebGL-ympäristössä tämä ei siis toimi. Tutkin Unityn dokumentaatiota, jossa on osio verkkoyhteystoimintojen käyttämiselle ja sain hyvän kuvan siitä, miten saisin lataajan toimimaan verkon välityksellä. Nykyisellään koodi lukee tekstiä tiedostosta, mutta voin luultavasti käyttää Unityn WWW-luokkaa lataamaan 3D-mallin sellaisenaan ja syöttää saadun vastauksen kokonaisuudessaan OBJ-lataajaan. Lataaja lukisi tiedon samalla tavalla rivi riviltä. Tästä lisää luvussa 4.2

3.4 Yhdistäminen muun web-sovelluksen ja palvelimen kanssa

Unityn WebGL-dokumentaatiossa esitetään muutama tapa, jolla WebGL-sovelluksen saa keskustelemaan muun sivun ja web-skriptien kanssa (51). *Application.ExternalCall*- ja *Application.ExternalEval*-funktiot ovat hyödyksi, kun C#-koodista halutaan kutsua

muuta verkkosivun JavaScript-koodia. Funktioita havainnollistetaan koodiesimerkissä 4. *ExternallCall*-funktio vaikuttaa erityisen sopivalta, koska sillä voi kutsua JavaScript-funktiota nimellä ja syöttää tarvittava määrä argumentteja tekstinä tai lukuarvoina.

```
Application.ExternalCall("TestiFunktio", "teksiä parametrinä", 2, 4.2F);
// tai
TestiFunktio(”tekstiä”, 2, 4.2);
```

Koodiesimerkki 4. JavaScript-funktioiden kutsu C#-koodista.

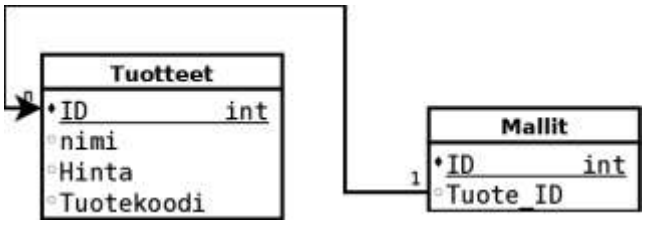
Verkkosivulla olevasta JavaScript-koodista voi tehdä kutsuja WebGL-sovelluksesta *sendMessage*-komennolla. JavaScript-puolella tarvitsee tietää vain objektin nimi 3D-kohtauksessa ja kutsua sillä olevaa komponenttia, jolla on julkinen metodi. Tästä on malli koodiesimerkissä 5. Komponentit ovat C#-luokkia, joilla määritetään objektien toiminnallisuus. *SendMessage*-funktio on WebGL-sovellusinstanssin metodi, eikä sitä kutsuta sellaisenaan. Lisätietoa on luvussa 4.1.

```
sendMessage(”FBX-loader”,”setURL”, ”https://testurl.net/product/model/144”);
```

Koodiesimerkki 5. WebGL-sovelluksen koodin kutsu JavaScriptistä.

Esimerkiksi voitaisiin JavaScript-koodilla antaa 3D-mallin verkko-osoite WebGL-sovellukseen ladattavaksi. Tämä taas asettaisi WebGL-puolella FBX-lataajan tai OBJ-lataajan URL:in. Näin olisi siis näkymätön nimetty objekti, kuin *FBX-loader* ja sillä olisi koodikomponentti, jossa olisi funktio *setURL*, joka ottaa vastaan tekstinä osoitteen, josta ladataan 3D-malli visualisointityökaluun.

Sovelluksen verkkopalvelinympäristö toteutettiin LAMP-*stackilla*. Se koostuu Linux-käyttöjärjestelmästä, Apache2-palvelinsovelluksesta, MySQL-tietokantapalvelimesta ja PHP-ohjelmointiympäristöstä. Tällä hetkellä tietokantaan on määritelty jo tuotemallien rakenne, ja ajattelin, että 3D-malleja varten voisi olla oma taulu tietokannassa. Tiedostot tallennettaisiin sovelluksen hakemistoon, josta ne olisivat luettavissa jollain reititetyllä URL:lla. Osoitereititys on tapa määrittää, mitä verkko-osoitteita sivustolla on käytössä. Kuvassa 7 havainnollistetaan, kuinka 3D-malli voisi olla kirjattuna uniikilla tietokanta-ID:llä. Tällä ID:llä malli voitaisiin tallentaa levylle esimerkiksi tiedostoksi 244.fbx. Saman tietorivin kanssa olisi viittaus tuotetaulun ID:een.

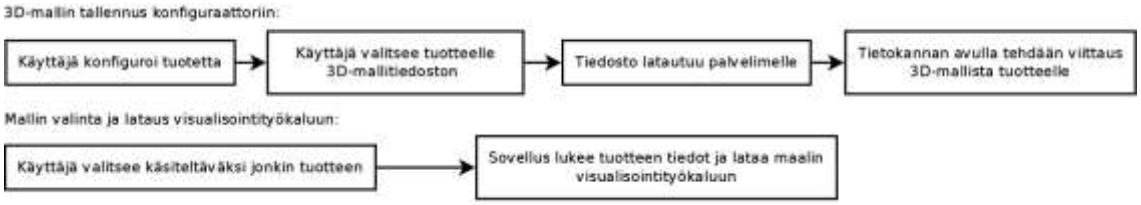


Kuva 7. Yksinkertainen (pseudo)relaatiomalli tietokannasta.

Ohjelmoinnin kannalta tuotteilla on koodattu ORM-malli toimintoineen. ORM-malli on koodiluokka, joka määrittää tietokannan tietueen toiminnallisuutta koodin puolella. ORM-malleihin voi koodata relaation, ja ORM-järjestelmä tekee tarvittavat tietokantakyselyt. Tätä relaatiota tai sen olemassaoloa voi sitten hyödyntää työkaluissa. Jos relaatiota ei ole, ei tarvitse sivulle ladata visualisointityökaluakaan. Jos taas relaatio on olemassa, voidaan 3D-mallin mallista lukea ID ja käyttää sitä mallinhakuosoitteessa.

3.5 Ohjelman ja visualisointityökalun suunniteltu toiminta

Kuvassa 8 nähdään yksinkertainen kaavio siitä, minkälainen prosessi 3D-mallin tallentaminen tuotekonfiguroinnissa voisi olla ja mitä loppukäyttäjän näkökulmasta tapahtuu, kun hän ottaa tuotteen käyttöön ja malli avautuu visualisointityökaluun. Loppukäyttäjä käyttää web-pohjaista konfigurointisovellusta, jossa hän voi tallentaa tuotteen konfiguroinnin yhteydessä 3D-mallin tuotteelle.



Kuva 8. 3D-mallin lataus- ja käyttöprosessi loppukäyttäjälle.

Kun loppukäyttäjä tallentaa tuotteen tiedot, ne tallennetaan palvelimelle tietokantaan ja 3D-malli ladataan myös palvelimen levylle. Tietokantaan kirjataan tiedot kyseisestä 3D-mallista, joka viittaa tietokannassa olevaan tuoteriviin. Näin 3D-mallit voidaan tietokantarelaation avulla kirjata omaan tauluunsa. Loppukäyttäjän ei tarvitse tehdä mitään avatakseen 3D-mallia visualisointityökaluun.

3.6 3D-mallien pakkauksesta

Olin hyvin vakuuttunut Unityn AssetBundleista ja erityisesti niiden pakkaamista koskevasta dokumentaatiosta. AssetBundlejen dokumentaatiossa kerrotaan, kuinka paljon tiedoston kompressointi säästää levytilaa ja verkkokaistaa. Kompressointi säästää tapauksesta riippuen tilaa. Riippuu tosin täysin datasta, kuinka paljon tilaa voidaan säästää. Toisenlainen data pakkautuu paremmin kuin toinen. Esimerkiksi kuva- tai videotiedostoja ei saa paljoa enempää pakattua, koska ne yleensä ovat jo sellaisessa formaatissa, missä niiden tiedostokoko on hyvin optimoitu. Tekstitiedostot, kuten lähdekoodi, ovat yleensä hyvin pakkautuvaa dataa.

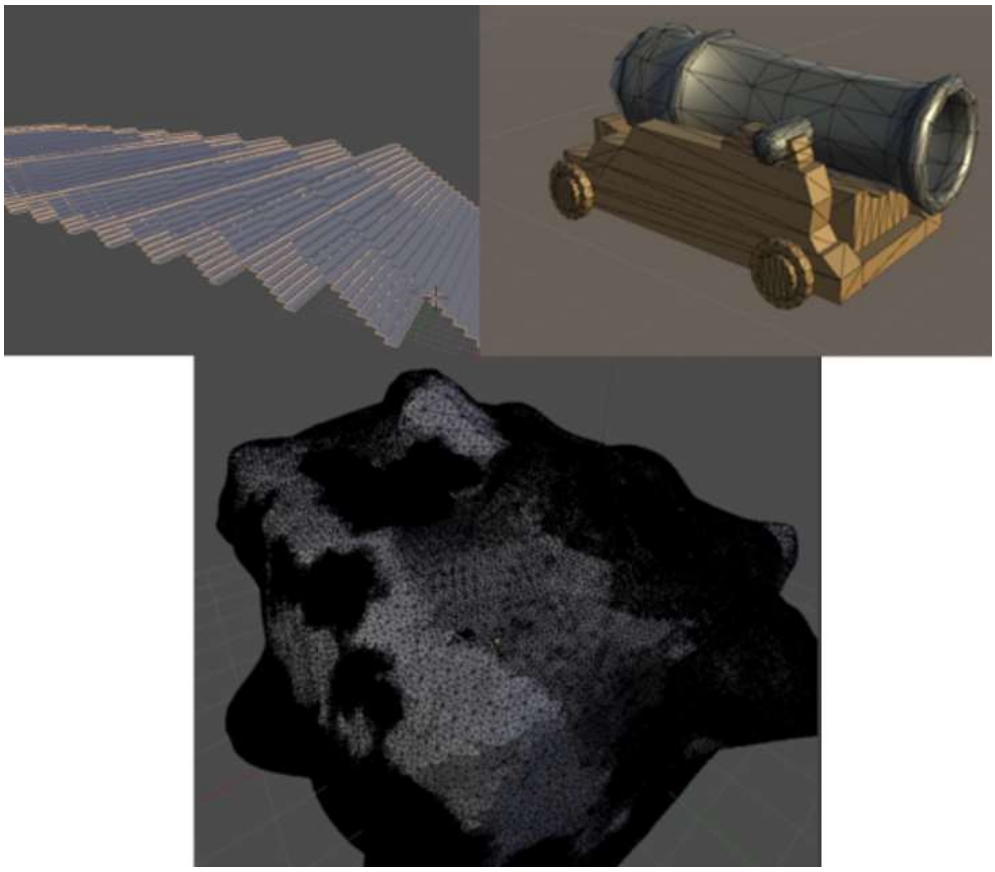
En voi tässä vaiheessa tietää, miten suuria 3D-mallitiedostoja loppukäyttäjät tulevat käyttämään, mutta optimoidut latausajat ja optimoitu palvelimen levytilan käyttö ovat mielestäni tärkeitä asioita ja syytä ottaa huomioon. Koska sovelluksen luonteen ja prosessin takia AssetBundleja ei voi käyttää, on tutkittava vaihtoehtoja datan pakkaukselle. Täytyy tutkia, onko tiedostoa mahdollista pakata asiakkaan selaimella esimerkiksi jollain JavaScript-koodilla vai voidaanko pakkaaminen suorittaa vasta, kun tiedosto on saapunut palvelimelle. Purkamisessa pitää selvittää, puretaanko pakattu tiedosto selaimessa JavaScript-koodilla vai onko mahdollista kirjoittaa purkukoodi Unityssä C#:lla. Hyvin todennäköisesti tämä on mahdollista, mutta pitää selvittää, toimiiko purkukoodi käännettynä WebGL-ympäristössä selaimella ja miten paljon purkaminen vaikuttaa suorituskykyyn.

Yksi tärkeä seikka on selvittää, minkälainen pakkausprosessi olisi sopiva työkalun kannalta. Jos pakkaaminen tapahtuu palvelimella, se voidaan eriyttää omaksi prosessikseen ja näin se ei estä loppukäyttäjää jatkamasta muiden toimien parissa sillä aikaa, kun palvelimelle ladattu tiedosto pakataan. Toisessa tapauksessa pakkaaminen tapahtuisi loppukäyttäjän selaimella, mutta olen epäileväinen sen suhteen pakkaamisen muistivaatimusten vuoksi. Ainakin jos halutaan hyvin korkea kompressiosuhde, muistia yleensä tarvitaan noin 1–4 gigatavua, mikä on liikaa selaimella suoritettavaan koodiin. Muistivaatimuksesta voidaan esimerkkinä esitellä avoimen lähdekoodin 7Zip-pakkausohjelma. 7Zipin (www.7-zip.org/) *normal*-pakkaustaso vaatii arvioilta noin 1,2 gigatavua, mutta siitä seuraava kevyempi *fast*-pakkaustaso noin 128 megatavua eli pyöreästi 0,13 gigatavua. 7Zip-ohjelma on saatavilla Linuxille komentorivityökaluna, minkä vuoksi se sopisi palvelimelle 3D-mallien pakkaajaksi.

7Zip käyttää omassa .7z-formaatissaan LZMA-pakkausalgoritmia, ja tämä algoritmi on myös tuettu xz-formaatin kanssa, joka on yleinen Linuxilla käytettävä pakkausformaatti, joka myös käyttää LZMA-pakkausalgoritmia. Xz- ja 7z-tiedostoformaatit eivät ole yhteensopivia keskenään, vaikka ne käyttävät ja tukevat samaa LZMA-pakkausalgoritmia. Xz-formaattia kirjoittava ohjelma on myös saatavilla Linux-käyttöjärjestelmälle, ja se on usein jopa esiasennettua. Tämä tekee siitä myös houkuttelevan vaihtoehdon palvelinpuolen pakkaukselle tiedostojen säilytystä varten. (52; 53; 54.)

Purkamiseen näen kaksi mahdollisuutta: joko selaimessa muun sivun kanssa toimiva JavaScript-koodi, joka tukee aikaisemmin esiteltyjä tiedostoformaatteja ja syöttää puretun datan tai Unityssä C#:lla totetutettu purkukoodi, joka tietenkin kääntyy JavaScriptiksi WebGL-muotoon julkaistaessa. Ongelma on, että en vielä tiedä, minkälaisia JavaScript-pohjaisia pakkausalgoritmitoteutuksia on saatavilla, joita saisi nopeasti käyttöön web-sovelluksessa. En myöskään tiedä, miten C#-toteutukset toimivat WebGL-ympäristössä. 7Zipin SDK on saatavilla niin kutsutusti ”*public domain*”, eli sitä voitaisiin käyttää kehitettävässä työkalussa. LZMA SKD tarjoaa C#-lähdekoodin LZMA-kompressoinnille ja dekompressoinnille. (52; 53; 54.) Vaikka tätä voisikin käyttää Unityn päässä, pitäisi palvelinpäässä pystyä pakkaamaan 3D-mallitiedostot suoraan LZMA-formaattiin, eikä 7zip- tai xz-formaattiin. Vielä en tiedä, miten tämä onnistuu. Löysin Unityn foorumilta Kaarlo Räihän kirjoittaman koodin LZMA-pakkaukseen ja purkuun, joka on jo muokattu Unitylle sopivaksi (70). Tämä voisi olla yksi mahdollinen vaihtoehto Unityssä tapahtuvan tiedostopurkamisen toteuttamiselle. Osa koodeista ja kirjastoista näyttää toimivan siten, että niille pitää antaa zip-tiedoston polku ja purkupolku, mikä ei tietenkään ole mahdollista selaimessa. Pakattu data pitäisi saada auki muistissa eikä purettua johonkin hakemistoon.

Tein tutkimuksia FBX-tiedostojen pakkaamisesta, koska sen on suunniteltu olevan ensisijainen tiedostomuoto visualisointityökalussa. Testasin kolmea eri mallia, joista kaksi tein nopeasti itse Blenderissä. Mallit ovat esillä kuvassa 9. Ensimmäinen 3D-malli on yksi kuutio, joka on muotoiltu niin pieniin osiin, että siinä on noin miljoona verteksiä eli pistettä avaruudessa. Toinen malli on joukko kuutioita, joita on kopioitu siten, että niistä muodostuu portaat, ja vielä nämä portaat on kopioitu, niin että niitä on monta peräkkäin. Kolmas malli, johon testasin pakkausta, on pieni laivan tykkimalli, jota käytettiin eräässä peliohjelmointikurssissa.



Kuva 9. Pakkaustestissä käytetyt mallit.

Tykki oli ensimmäinen malli, jota yritin pakata. Taulukosta 1 voidaan nähdä, että tykkimalli oli 261 kilotavua suuri ja pakkasin sen 7Zipin pikatoiminnolla. Kokeilin myös manuaalisesti *normal*- ja *maximum*-pakkaustasoja, ja maximum antoi saman 21 kilotavun tiedoston kuin pikapakkaustoiminto, jossa ei ole asetuksia. Taulukossa 1 *pakkaus%* tarkoittaa sitä, kuinka paljon alkuperäisestä tiedostokoosta saatiin vähennettyä pois. Tykkimallista sain vähennettyä kilotavujen tarkkuudella 92 % alkuperäisestä koosta. Tämä oli mielestäni niin suuri määrä, että päätin tehdä ison 3D-mallin Blenderissä, jossa olisi paljon dataa. Tällainen tiedosto vastaa arvioltani paremmin sellaisia malleja, joita työkalulla todennäköisesti tullaan käsittelemään.

Tässä vaiheessa en vielä ollut tietoinen FBX-tiedostoformaatin binääri- ja ASCII-muotoisista tallennusformaateista. Tein kuutiomallin, jossa oli noin miljoona verteksiä ja paljon pieniä muotoja. Tallensin mallin Blenderistä FBX-muotoon, ja tiedostokooksi muodostui 59 megatavua. Pakkasin sen samalla tavalla kuin tykkimallin, mutta kokoa lähti vain 6,78 %. Tämä aiheutti paljon hämmennystä. Edellinen malli pakkautui huomattavasti

enemmän kuin tämä miljoonan verteksin kuutio. Ensin ajattelin sen olevan jotenkin mallista johtuva ilmiö. Arvelin, että yksinkertaisempi malli on paremmin pakkautuvaa dataa.

Taulukko 1. FBX-tiedostojen binääri- ja ASCII-formaatin kompressointi.

3D-malli	FBX BINARY	FBX ASCII	FBX BINARY (kompressoitu)	FBX ASCII (kompressoitu)	Pakkaus% BIN	Pakkaus% ASCII
Miljoona verteksiä	59 Mt	266 Mt	55 Mt	45 Mt	6,78 %	83,08 %
Kopioituja kuutioita	256 kt	2691 kt	112 kt	61 kt	56 %	98 %
Tykki	-	261 kt	-	21 kt	-	92 %

UniFBX-lataajan dokumentaatiossa mainitaan, että lataaja tukee ASCII-muotoista FBX-tiedostoa. Avasin miljoonan verteksin mallini uudestaan ja tarkistin Blenderin FBX-exporterin asetukset. Oletuksena oli valittuna *FBX 7.4 binary* -formaatti. Vaihtoehtoisesti pystyy valitsemaan *FBX 6.1 ASCII* -formaatin. Tallensin saman mallin FBX-tiedoston ASCII-formaattiin ja sain mallin tiedostokooksi 266 megatavua, joka on noin neljä ja puoli kertainen binääriformaattin verrattuna. Tämä on huono asia latausaikojen ja kaistankäytön puolesta, jos tuetut ASCII-formaattiset FBX-tiedostot ovat huomattavasti suurempia kuin binäärimuotoiset. En ollut kovin toiveikas tämän tiedoston pakkauksen suhteen, koska ajattelin, että edellisen pakkaustestin tulos tulisi toistumaan, ja ajattelin huonon pakkaussuhteen johtuvan siitä, että Blender jo jollain tavalla osaa tallentaa FBX-tiedoston optimaalisesti. Pakatun ASCII-muotoisen FBX-tiedoston koko kuitenkin yllätti minut täysin. Kuten taulukosta 1 voidaan nähdä, pakattu ASCII-muotoinen FBX-tiedosto on pienempi kuin pakattu binäärimuotoinen FBX-tiedosto ja pakkauksessa saatiin tiedostokokoa pienennettyä 83 %. Ero on merkittävä, koska aluksi ASCII-FBX oli paljon suurempi kuin binääri-FBX. Tällä löydöllä voi olla suuri merkitys työkalun kehitykselle, jos päätetään kehittää kompressoiduille tiedostoille tuki palvelinpäähän ja purkutoiminnallisuudelle selaimen päähän.

Kuitenkin minulle jäi vielä halu selvittää, voisiko mallin geometrialla olla tekemistä kompressointikyvyn kanssa. Tein mallin, jossa samaa kuutiota kopioitiin moneen suuntaan, ja tuloksena syntyi portaikko, jossa kuusitahkoista kappaletta oli monta vierekkäin muodostaen rivin, ja tätä riviä toistetaan viistosti ylöspäin muodostaen porrasaskelmia. Koko askelmia kopioin vielä huvin vuoksi perätysten, jotta mallitiedostosta tulisi hieman suurempi. Tallensin mallin sekä binäärimuotoiseksi että ASCII-muotoiseksi FBX-tiedostoksi. Binääritiedoston koko oli 256 kilotavua ja ASCII-tiedoston 2 691 kilotavua eli noin kymmenen kertaa suurempi, vaikka edellinen erittäin kompleksi malli oli 4,5 kertaa suurempi. Jälleen pakkaustulokset yllättivät. Pakkauksen jälkeen binäärimuotoinen tiedosto oli 61

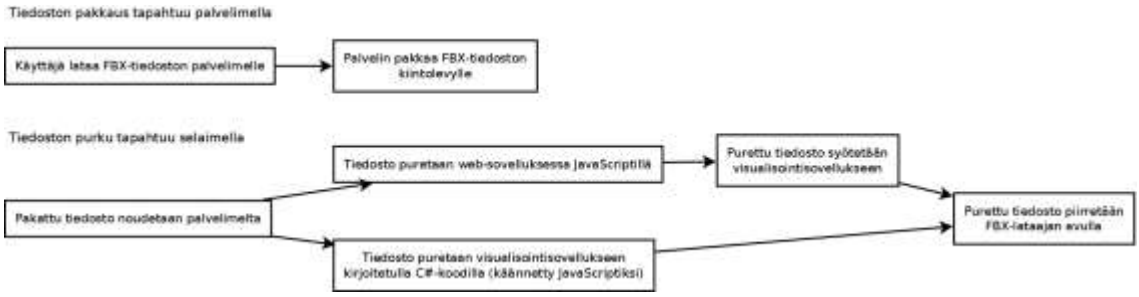
kilotavua eli tiedostokoosta hävisi 56 % vaikka edellisellä binääritiedostolla tiedostokokoa lähti vain 6,78 %. ASCII-tiedosto pakkautui taas todella hyvin. Tiedostokoosta lähti noin 98 %, mikä on todella suuri määrä ja parempi kuin edellisen mallin ASCII-tiedoston pakkauksessa. Näyttäisi siltä, että mallilla on merkitystä siihen, kuinka hyvin se pakkautuu.

Portaikkomallin hyvään pakkautuvuuteen vaikuttaa luultavasti se, että siinä toistuu paljon samoja pinnan muotoja ja vain verteksien paikka avaruudessa muuttuu. Jokainen porras koostuu monesta samanlaisesta kuutiosta, joiden tahkot osoittava samoihin suuntiin. Pintojen normaalit ovat siis samat. Tiedä datan pakkaamisesta sen, että toistuvaa dataa on helppo pakata. Sen sijaan, että kirjataan viisisataa kertaa ”*pinta 1 osoittaa ylös, pinta2 osoittaa ylös, pinta3 osoittaa ylös… pinta500 osoittaa ylös.”*, voidaan tieto pakata esimerkiksi muotoon ”*pinnat1-500: ylös*”. Tämänkaltainen toistuvuus näyttäisi vaikuttavan merkittävästi myös binäärimuotoisen FBX-tiedoston pakkautuvuuteen.

Testasin vielä OBJ-formaatin tiedostokoot ja pakkautuvuuden miljoonan verteksin mallilla ja portaikkomallilla. Pakkamattomana tiedostokoot olivat binääri-FBX:n ja ASCII-FBX:n väliltä. Miljoonan verteksin kuutio oli 164 megatavua ja portaikkomalli 1 035 kilotavua. Pakattuina OBJ-tiedostot olivat 33 megatavua (pakattu ASCII-FBX 45 Mt) ja portaikkomalli 51 kilotavua (pakattu ASCII-FBX 61 kt). Molemmat voittavat pakatut ASCII-FBX-tiedostot tiedostokoossa. OBJ-formaatti voisi olla paras vaihtoehto pakattuna, jos pakattujen tiedostojen käsittelyyn kehitetään tuki. Myös pakkaamattomana OBJ-formaatti voittaa ASCII-FBX -formaatin tiedostokoossa. Isompi miljoonan verteksin kuutio OBJ-formaatissa on noin 100 megatavua pienempi kuin ASCII-formaatin FBX-tiedosto. OBJ-lataajan implementointi visualisointityökaluun vaikuttaa näillä tiedoilla melko järjelliseltä. Tekovaiheessa tulee ilmi, kumpaa formaattia ja lataajaa on parempi käyttää.

Tiedostojen pakkaus on hyvin mielenkiintoinen aihe, ja nämä testit osoittavat pakkaamisen varastoinnin ja verkon välityksellä siirtämisen kannalta erittäin hyödylliseksi. Täytyy kuitenkin todeta, että tässä tapauksessa varsinkin purkamisen toteuttaminen vaikuttaa melko hankalalta, koska purkamisen täytyisi tapahtua loppukäyttäjän koneella (client-side). Ei ole järkevää purkaa dataa palvelimella ja sen jälkeen siirtää sitä verkon kautta loppukäyttäjälle visualisointityökaluun avattavaksi, koska kaistaa tarvitaan silloin todella paljon enemmän. Pakkaaminen palvelimella on yksinkertainen operaatio, kuten kuvasta 10 voidaan nähdä. Pakkausta palvelimen päässä ei tule olemaan vaikea toteuttaa, koska pakkausohjelmistoja ja kirjastoja on tarjolla useita ja tiedostosta pakkaaminen pakattuun

tiedostoon on mahdollista tehdä. Joskus pakkaustyökaluja on jo valmiiksi asennettu käyttöjärjestelmän mukana. Pakkaamisprosessi voidaan käynnistää esimerkiksi PHP-ohjelmointikielellä kutsumalla käyttöjärjestelmän komentoja.



Kuva 10. Pakkaus- ja purkuprosessien jäsentely.

Loppukäyttäjän päässä tapahtuva purkaminen sen sijaan on vielä kehityksen kannalta epäselvä asia. Pitäisi kehittää ja testata, miten esimerkiksi 7ZIP- tai XZ-formaattisia tiedostopaketteja voisi avata selaimen päässä. Purettua dataa ei voi tietenkään syöttää tiedostoon, vaan se pitäisi purkaa suoraan laitteen käyttömuistiin sovelluksessa. Joitain kirjastoja on saatavilla JavaScript- ja C#-kielille, mutta tässä vaiheessa on vaikea sanoa, mikä olisi paras tähän tarpeeseen. Saatavilla olevan muistin määrä on myös rajoittava tekijä, joten gigatavun 3D-mallia on luultavasti turha edes yrittää avata visualisointityökalussa.

Pakkaustoiminnallisuus on olosuhteiden myötä sen verran monimutkainen, että se jää luultavasti viimeiseksi kehitysjärjestyksessä, jos sille edes jää aikaa tässä insinöörityössä. Tärkein ominaisuus on kuitenkin saada malli yksinkertaisesti näkyviin visualisoitavaksi. Voi olla, että pakkaaminen jää kokonaan tämän raportin ulkopuolelle, mutta ainakin käsitys ja tieto sen hyödyistä on olemassa.

4 Visualisointityökalun tekeminen

4.1 Yhdistäminen web-sovellukseen

Aloitin visualisointityökalun yhdistämisen tekemällä WebGL-sovellukseeni tekstialueen, johon voisin yrittää lähettää tietoa muun verkkosivun puolelta. Ajatuksena oli, että teen tekstikentän ja napin HTML:llä verkkosivulle, josta lähetetään tekstikentän teksti WebGL-sovelluksen sisälle. Kirjoitin napille klikkauksen kohdalla suoritettavan funktion ”*testUnityAlert*”. Tämä funktio kutsuu *sendMessage*-funktiota. Tein Unityssä tekstiobjektin, joka on sovelluksen yläreunassa. Nimesin objektin *UnityAlert*:iksi ja ohjelmoin sille funktion, joka ottaa vastaan tekstimuuttujan ja asettaa tekstin objektin tekstikomponentin tekstiksi, mikä tulee ruudulle näkyviin.

Päänvaivaa aiheutti verkkosivun ja WebGL-sovelluksen välinen kutsu, joka ei toiminut. Dokumentaatiossa sanotaan, että *sendMessage*-funktio toimii sellaisenaan. Tämä on juuri se mitä tein, mutta selaimen konsoli antoi virheviestin ”*sendMessage is not defined*”.

Tutkin foorumeita, josko joku olisi kohdannut saman ongelman. Eräässä Unityn foorumin keskustelussa (55) on ilmeisesti esityksen tasolla olevia muutoksia WebGL-yhdistämisestä, sillä keskustelu oli otsikolla *white paper*. Kyseessä ei siis ollut kenenkään ongelma vaan tietopaketti tulevista muutoksista. Koodiesimerkissä 6 on foorumin esityksen esimerkki, jossa *sendMessage*-funktiota kutsutaan muuttujan läpi. Esimerkistä selvisi, miten tämä myGame-muuttuja on saatu. Unityn WebGL-sovellus liitetään HTML-elementtiin elementin ID-nimen perusteella.

```
<script> var myGame = UnityLoader.instantiate("gameContainerId", "http://mydo-
main/myfolder/Build/mygame.json");</script>

<script>myGame.SendMessage("myObject", "myFunction", "foobar");</script>
```

Koodiesimerkki 6. WebGL-sovelluksen alustus ja kutsu HTML-sivulla (55).

MyGame-muuttujaan alustetaan instanssi WebGL-sovelluksesta. Ensimmäinen parametri on HTML-elementin ID, johon sovellus halutaan ja toinen parametri on osoite, josta haetaan JSON-tiedosto (JavaScript Object Notation), joka sisältää polut sovelluksen dataan. Tarkastin saman tien oman HTML-pohjani, ja siellä oli samanlainen rivi. MyGame-muuttuja oli vain gameInstance. Ilmeisesti tässä foorumissa esitetyt asiat ovat jo käytössä, eikä niitä ole vielä päivitetty dokumentaatioon. Oma versioni Unitystä on 5.6 ja

tarkistin lukevani 5.6-dokumentaatiota. Muutin nappini koodin kutsumaan sendMessagea gameInstance-muuttujan läpi. Komento onnistui, mutta jostain syystä en pystynyt kirjoittamaan tekstiruutuun. Nappi kyllä toimi, ja tyhjä teksti kirjoitettiin WebGL-ruutuun alustavan ”change text from browser” -tekstin tilalle.

Vaikuttaa siltä, että WebGL-sovellus syö kaiken näppäimistösyötteen muulta selaimelta. Dokumentaatiossa on kohta, jossa puhutaan syötöstä WebGL-ympäristössä (56). WebGL-prosessoi kaiken näppäimistösyötteen, vaikka fokus ei ole sovelluksessa. Muutaakseen tätä kehittäjän on asetettava *WebGLInput.captureAllKeyboardInput:*n arvoksi *false*. Tein muutoksen koodiin ja puskin taas uuden version WebGL-muotona ulos Unitystä. Nyt pystyin kirjoittamaan verkkosivun tekstikenttiin. Nyt taas visualisointisovelluksen kameran liikuttaminen ei enää toiminut. Kameran liikuttaminen tapahtui oletuksena näppäimistöllä, ja nyt näppäinpainalluksia ei tunnistettu ollenkaan, vaikka klikkasin WebGL-sovelluselementtiä, joka siirtää fokuksen siihen.

Tein HTML-sivulle napin, jota painamalla lähetetään kutsu koodiin, joka vaihtaa *WebGLInput.captureAllKeyboardInput:*n arvon. Nappi toimii, ja näppäinkomentojen kaappausta koskevaa arvoa voi muuttaa. Tämä ei kuitenkaan ole kovin käytännöllistä loppukäyttäjän näkökulmasta. Mietin, olisiko mahdollista muuttaa Unityn puolen koodia siten, että se ottaa *boolean*-arvon (tosi/epätosi) vastaan ja asettaa kaappausmäärityksen sen mukaan. Verkkosivun puolella taas kuuntelisin input-elementtien fokus-tapahtumia ja fokuksen poistumistapahtumia. Verkkosivulla on käytössä *jQuery (*jquery.com/). Voin asettaa kaikille input-elementeille fokus- ja blur-eventit**,** kuten koodiesimerkissä 7 näkyy. Koodissa kutsutaan WebGL:ssä olevaa *AppScripts*-objektia, jolla on koodikomponentti, joss määritetään *SetKeyboardCapture*-funktio. Funktio asettaa näppäimistökaappauksen päälle tai pois.

```
$('input').focus(function(){
		toolInstance.SendMessage('AppScripts', 'SetKeyboardCapture', 0);
})
$('input').blur(function(){
		toolInstance.SendMessage('AppScripts', 'SetKeyboardCapture', 1);
});
```

Koodiesimerkki 7. Fokus- ja Blur-tapahtumin reagoivat koodit.

Tämä idea vaikutti toimivan oikein hyvin. Pystyin kirjoittamaan tekstikenttiin normaalisti, ja kun käyttäjä klikkaa takaisin visualisointityökaluun, sekin toimii niin kuin pitää. Tässä vaiheessa interaktio toimii, joten voin keskittyä 3D-mallin latauksen tekemiseen.

4.2 3D-mallin tuontijärjestelmä

UniFBX tukee sekä levyltä että verkon kautta lataamista, minkä vuoksi se sopii insinöörityön työkalun tarpeisiin oikein hyvin. FBX-lataajaan tarvitsee syöttää URL tai tiedostopolku, mistä se lataa mallin, minkä jälkeen se parsii luetusta 3D-mallin tiedostosta objektit ja niiden muodot ja muodostaa piirtokelpoisen mallin. Jotta tämä toimii suunnitellulla tavalla, aion tehdä toiminnon, joka ottaa vastaan URL-osoitteen tekstinä ja asettaa sen FBX-lataajan asetukseksi, ennen kuin lataus aloitetaan. Tämä on siis toiminto, joka saa tiedon muualta verkkosivun koodista. Testaan latausta ensin suoraan Unityn editorissa, koska WebGL-muotoon kääntäminen kestää joka kerta muutaman minuutin. Tämä on erityisen työlästä, jos pitää testata pieniä yksityiskohtia tai korjata kirjoitusvirheitä. Minulla on virtuaalikoneella pystytetty paikallinen palvelin, josta voin simuloida verkon välityksellä lataamista. Virtuaalikonetta ajetaan Oraclen Virtualbox (www.virtualbox.org/) -ohjelmassa ja on tehty valmiista Laravel Homestead (laravel.com/docs/5.3/homestead) -virtuaalikonemallista (box). Olemme yrityksessä ottaneet virtuaalikoneeseen Apache2-palvelinohjelmiston (httpd.apache.org/) käyttöön oletuksena olevan Nginxin tilalle.

Otin samalla ilmaiseksi saatavilla olevan *Runtime OBJ Importer* -koodipaketin (40) käyttöön visualisointityökaluprojektissa. Nopean katsauksen jälkeen huomasin, että OBJ-lataaja lukee vain kiintolevyltä tiedostosta. Myöhemmin tässä luvussa on esimerkki muutetusta koodista, joka lukee suuren tekstimuuttujan rivi riviltä. Katsoin FBX-lataajan koodista, miten se lukee verkkoyhteyden välityksellä tietoa, ja huomasin, että se on melko helppo prosessi. Pitäisi käyttää Unityn WWW-luokkaa ja lukea WWW-kutsun vastaus. Tästä on malli koodiesimerkissä 8.

FBX-lataajan ja OBJ-lataajan välillä on vain ratkaiseva ero. FBX-lataajan luokka laajentaa Unityn *Monobehaviour*-luokasta ja käyttää sen *Coroutine*-toiminnallisuutta. *Coroutine* on osa koodia, joka voi palauttaa suorituksen hetkellisesti muulle koodille esimerkiksi pitkään kestävässä *loopissa*. OBJ-lataaja ei peri *monobehaviouria,* ja sen metodit ovat staattisia eivätkä käytä *coroutinea*. En voi siis suoraan muuttaa OBJ-lataajaa toimimaan samoin. WWW-luokkaa on tarkoitus käyttää *Coroutinessa,* koska sen latausta on odotettava loppuun. Ei kuitenkaan haluta, että lataus keskeyttää kaiken muun toiminnon. Odotuksen voi hoitaa eri tavoilla, ja kaikki eivät toimi selainympäristössä. Lisätietoa ongelmasta myöhemmin tässä luvussa.

```
public void StartWWWRequester() {
   StartCoroutine("Requester", url);
}


private IEnumerator Requester(string netpath) {
   WWW www = new WWW(netpath);
   yield return www;

   if (www.error == null){
      OBJLoaderWWW.LoadOBJFile(www.text);
   }
}
```

Koodiesimerkki 8. WWW-kutsu *Coroutinella*. Tämä on sopiva selainympäristössä.

Muutin OBJ-luokan koodia siten, että se ei lue tiedostoa vaan tekstimuuttujan rivi riviltä, kuten koodiesimerkissä 9 näkyy. WWW-kutsun vastauksen teksti pitää jakaa osiin rivinvaihdon mukaan. Rivinvaihdon merkki on tekstieditorissa näkymätön ”/n”. Koodissa muuttuja-*s* on WWW-lataajasta saatu *www.text*-muuttuja eli WWW-kutsun vastauksen sisältö kokonaisuudessaan.

```
foreach (string ln in File.ReadAllLines(filename))
                // muutetaan alla olevaan muotoon
foreach (string ln in s.Split(new string[] { "\n" }, StringSplitOptions.RemoveEmptyEntries))
```

Koodiesimerkki 9. Muutettu koodi siten, että se lukee WWW-kutsun vastauksen tekstin rivi riviltä.

Nyt pystyin testaamaan latausta WebGL-puolella. Puskin WebGL-version ulos ja avasin sen selaimessa. Kokeilin molempia lataajia, eikä lataus toiminut. Selaimen konsoli osasi kertoa, että WWW-kutsua ei voi tehdä toiseen verkkodomainiin CORS (Cross-Origin Resource Sharing) periaatteiden (57) ja turvallisuuden takia. Minun oli siirrettävä WebGL-muotoon käännetty projekti konfiguraattorisovellukseen ja tehtävä visualisointityökalulle siihen oma sivu. Tämän jälkeen testasin latausta uudestaan, eikä CORS-varoitusta enää tullut. Malleja ei kuitenkaan tullut näkyviin ollenkaan, ja varattu muisti loppui kesken. Epäilin, että syy on edelleen WWW-kutsussa tai siinä, miten tiedostoja käsitellään. Molemmissa asioissa kyllä on syytä jumiutumiseen, mutta käyn läpi ensin WWW-kutsun. Tiedoston käsittelyn ongelmasta lisää luvussa 4.3.

Unityn forumilla oli esitetty ongelma, jossa muisti loppui kesken, kun ladattiin tietoa WWW-kutsun kanssa (58). Ongelman syynä on tapa, jolla kutsun valmistumista odotetaan. Se on mitään tekemätön *while loop,* joka lukee WWW-luokan *isDone*-muuttujaa. Unityn editorissa ja itsenäisessä sovelluksessa tämä toimii oikein hyvin, mutta WebGL-

ympäristössä se aiheuttaa ongelman, koska *while loop* ei palauta suoritusta takaisin selaimelle. (59). Forumilla kerrotaan, että *while loop*, joka odotti niin kauan, kuin WWW-luokan *isDone*-muuttuja ei ole tosi, pitää muuttaa *yield return*iksi. *Yield returneja* käytetään *coroutineissa*, mistä halutaan palauttaa komento takaisin muualle koodiin, kunnes on tarpeen palata taas *coroutineen*.

Testasin taas latausta sekä OBJ- että FBX-lataajalla, kun olin muuttanut WWW-kutsun odottamisen sopivaksi. Taas mallit latautuivat Unityn editorissa hyvin. Testasin kahta mallia. Noin 15 megatavun kokoinen FBX-malli ja yhden megatavun kokoinen OBJ-malli latautuivat 5–10 sekunnissa, mikä on siedättävä aika.

Uudelleentestaamisen jälkeen sama ongelma toistui. Halusin varmistua, että WWW-kutsu varmasti saa vastauksen, joten tein WebGL-sovellukseeni tekstikentän, johon syötin WWW-kutsun vastauksen OBJ-lataajan sijaan. Editorissa OBJ-mallin tiedoston ensimmäisiä rivejä tuli tekstikenttään näkyviin, lataus siis toimii. Vielä pitää selvittää, mikä jumittaa 3D-tiedoston parsimisen. Lataajien koodeissa on ehkä vieläkin jotakin, mikä ei toimi WebGL-ympäristössä. Virheviestejä ei ole, ja jos jokin ei pistä koodista silmään pitää laittaa Unityn asetuksista suurempi tuki WebGL-virheviesteille ja tehdä niin sanottu *developement build*, jossa on enemmän tukea virhetilanteiden analysoinnille (60). Näitä toimintoja käyttäessä WebGL-buildaamiseen kuluva aika vain lisääntyy merkittävästi, mikä hidastaa todella pahasti tämänkaltaista koodiongelmien korjaamista.

4.3 Ongelmia 3D-tiedoston parsimisen kanssa.

3D-mallien lataaminen ei vieläkään toiminut WebGL-puolella ja WWW-luokan ongelmien jälkeen ajattelin, että 3D-mallin latauksessa on vielä jotakin koodia, joka ei toimi WebGL-ympäristössä. Tutkiessani koodeja huomasin, että FBX-lataaja toteuttaa *IDisposable interfacen*. Selvitin C#-dokumentaatiosta, mitä kyseisen *interfacen* toteuttaminen tarkoittaa. *IDisposable-interfacella* toteutetaan toimintoja, joilla vapautetaan luokan käyttämiä manageroimattomia resursseja (61). Tarkoitus on ilmeisesti kutsua *Dispose*-metodia, kun muistissa olevan datan käyttö tiedetään päättyväksi. Tällä vähennetään syntyvän roskadatan määrää. Se havaitaan C#:n *garbage collectorilla*, joka aika ajoin siivoaa ja järjestää sovelluksen tarvitsevan muistin. Dokumentaatiosta selviää, että *IDisposablea* hyödyntäviä luokkia voidaan käyttää *using*-rakenteella esimerkiksi kuten koodiesimerkissä 10 näkyy. *Using-rakenne* on oikeastaan kauniimpi tapa implementoida ja käyttää

IDisposable-interfacen toimintoja. Käytännössä C#-koodintulkitsija kääntää tämän rakenteen *try-catch*-blokiksi ja lopuksi WWW-luokan käyttämät resurssit vapautetaan, niin kuin vapauttaminen luokkaan on toteutettu. Koodiesimerkissä on myös selainympäristössä toimimaton *while loop.* Toimiva koodiesimerkki on luvussa 4.2.

```
private IEnumerator Requester(string netpath) {
   using(WWW www = new WWW(netpath)){
      while(! www.isDone){}

      if (www.error == null){
         OBJLoaderWWW.LoadOBJFile(www.text);
      }
   }
}
```

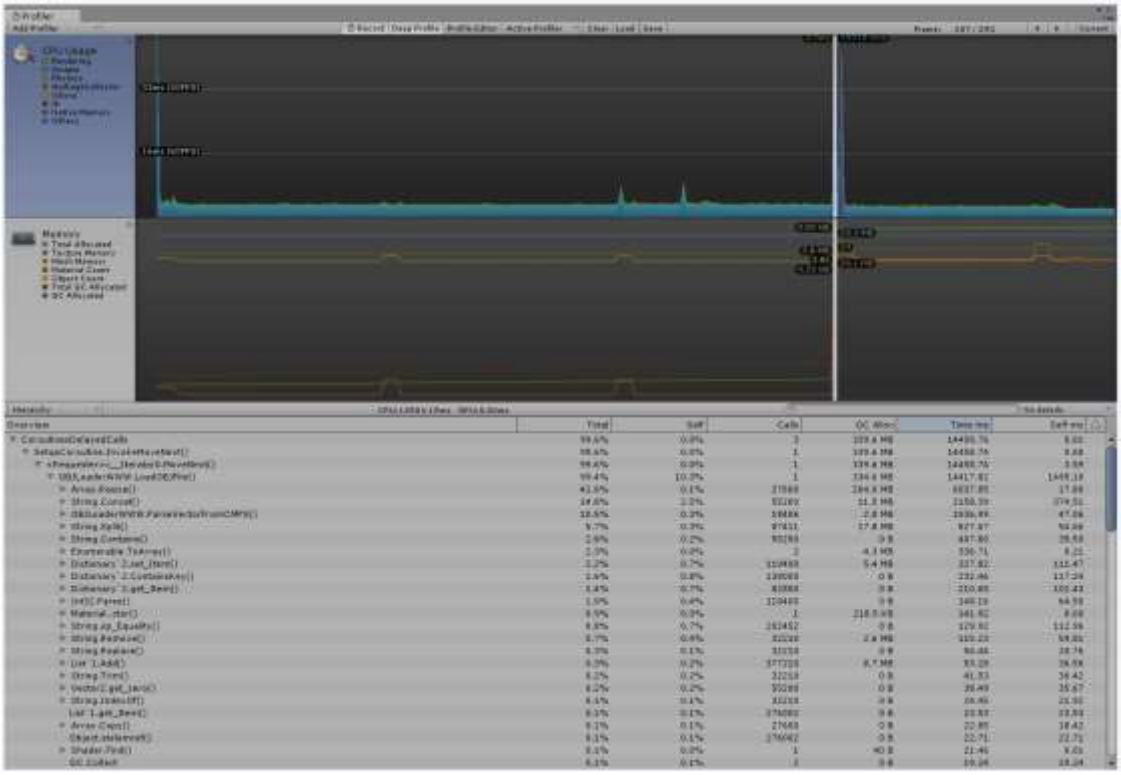
Koodiesimerkki 10. Using-rakenne C#-koodissa. Tämä ei toimi selainympäristössä.

Hulusi Onder on kirjoittanut artikkelin WebGL-kehityksestä, ja hän mainitsee, että *try-catch-blokit* eivät toimi (62). Jos *catch-bloki* ajetaan, sovellus kaatuu. Juuri tätä *IDisposable interface-* ja *using*-rakenne tekevät. Muokkasin koodia taas ja poistin molemmista WWW-lataajista *using*-rakenteen. Vasta tässä kohtaa sain ensimmäisen WWW-kutsun vastauksen esitettyä tekstikenttään OBJ-lataajalla. Sovellus ei nyt jumiudu antamatta mitään virheilmoitusta, kun WWW-kutsun vastauksen syöttää näkyviin tekstiksi.

WWW-kutsun koodin korjaamisen jälkeen ilmeni seuraava ongelma: OBJ-tiedostoa ladattaessa siis luetaan tieto rivi riviltä ja parsitaan siitä verteksit, pinnat, pintojen normaalit ja tekstuurien uv-koordinaatisto. Tämä on melko raskas prosessi, ja yhden megatavun OBJ-mallissa onkin noin 50 000 riviä tekstiä. Tässä kohtaa visualisointityökalun kehitys keskeytyi pahasti, sillä 3D-mallia luettaessa selaimella loppuu muisti. Yritin jopa määrittää sovelluksen tarvitseman muistin 1,5 gigatavuun, mutta silti selaimessa suoritettavalta sovellukselta loppuu muisti.

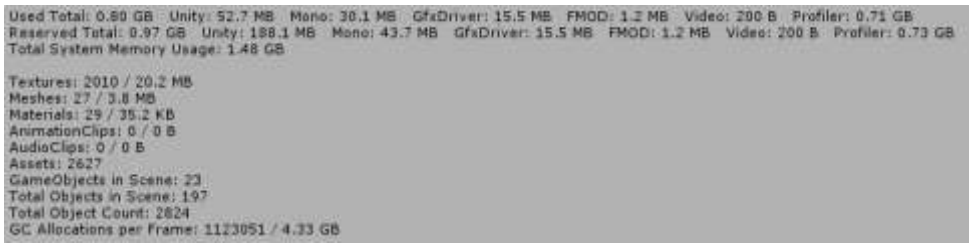
Tutkin Unityn profiloijalla, mitä oikein tapahtuu koodin suorituksen aikana. Kuvasta 11 näkyy, kuinka latauksen hetkellä ilmenee hyvin suuri piikki prosessointiajassa. Alapuolella listassa on paljon toimintoja, joita kutsutaan parsimisen aikana jopa satoja tuhansia kertoja. Tämä kaikki siis vain yhden megatavun tiedoston kanssa. Monimutkainen 3D-malli saattaa olla tekstimuotoisena tiedostona satoja megatavuja, ellei jopa enemmän. Näin suurissa tiedostoissa prosessointi veisi todella paljon aikaa. Eniten aikaa vie *Array.Resize*-toiminto. Luultavasti monet dynaamiset listat ja taulukot aiheuttavat tämän,

sillä parsimisessa on monta sisäkkäistä *looppia*, joissa lisätään dynaamisiin listoihin muun muassa verteksi-indeksejä.



Kuva 11. Profilointi 3D-tiedoston parsimisen hetkellä.

Array.Resize-funktiolla on vielä toinenkin mielenkiintoinen ilmiö. Toiminto näyttää aiheuttavan paljon roskamuistia. C#-dokumentaatiossa kerrotaan, että *Array.Resize*-funktio asettaa uuden taulukon ja kopioi tiedot vanhasta taulusta uuteen (63). Sitten se korjaa vanhan taulun uudella. Tämä toimii vain yksiulotteisilla taulukoilla. Profiloijan prosessilistassa se näyttää allokoivan noin 250 megatavua muistia, jonka *garbage collectorin* täytyy vapauttaa. Vielä suuremman numeron antaa muistiprofiloija, jonka tietoja esitetään kuvassa 12. Samalla hetkellä, kun 3D-mallia parsitaan, syntyy 4,33 gigatavua roskamuistia. Ei ole mikään ihme, että tätä toimintoa ei pysty suorittamaan selainympäristössä.



Kuva 12. Muistiprofilointi 3D-tiedoston parsimisen hetkellä.

Tekemieni testien pohjalta vaikuttaa siltä, että OBJ-lataajan koodissa on oikaistu jossakin kohtaa. Tällä hetkellä projektin tässä vaiheessa on aivan liian työlästä edes harkita koodin uudelleenkirjoittamista, vaikka lisenssi sen sallisikin. Lisenssissä mainitaan, että koodin osaa tai kokonaisuutta ei saa käyttää ja esittää omanaan. Tässä vaiheessa on vielä teidä, saisiko koodin muokattua niin, että se toimii myös WebGL-ympäristössä. Ainakin koodi on vain yhdessä tiedostossa, ja siinä on alle 500 riviä koodia.

Unityn WebGL-dokumentaatiossakin sanotaan *garbage collectorista*, että se suoritetaan vain, kun prosessin *pino (first-in-first-out datarakenne)* on tyhjä, eli aina muutaman *framen* välein. Normaaliolosuhteissa tällä ei ole merkitystä, mutta tämänkaltaisessa prosessissa tulos on toimimaton sovellus.

FBX-lataaja ei myöskään toimi WebGL-ympäristössä. Se ei ainakaan tässä kohtaa täytä sille varattua muistia, mutta jumittaa selaimen täysin. Voi hyvin olla, että koodissa on vielä osia, jotka eivät ole toimivia WebGL-ympäristössä. Niiden metsästäminen ja testaamien taas on erittäin työlästä, koska muutoksia pitää aina testata selaimessa ja projektin vienti WebGL-muotoon kestää aina kahdesta kolmeen minuuttia. FBX-lataajan koodin määrä on moninkertainen OBJ-lataajan nähden, ja tekijänoikeusasiatkin askarruttavat.

4.4 Vaihtoehtoisen järjestelmän testaaminen

Koska projektin aikataulu alkoi lähestyä loppua ja molemmat 3D-mallien lataajat osoittautuivat käyttökelvottomiksi WebGL-ympäristössä, päätin vielä palata ThreeJS-kirjastoon ja testata, toimiiko siinä 3D-mallin lataus. ThreeJS:ssä oleva OBJ-lataaja ainakin oli toteutettu melko samalla tavalla. Tein ThreeJS-esimerkin (65) pohjalta testisovelluksen,

joka lataa OBJ-tiedoston verkko-osoitteesta. Esimerkkikoodia piti muuttaa siten, että se hakee OBJ-mallin aikaisemmin tekemästäni osoitteesta verkkosovelluksesta. Kaikki koodi on puhdasta JavaScriptiä eikä sitä ole käännetty toisesta kielestä, niin kuin Unityn WebGL:n kohdalla tapahtuu. ThreeJS ja sen OBJ-lataaja on saatavilla avoimena lähdekoodina Githubista. (64.)

Ensin osoitin ThreeJS-lataajalle saman yhden megatavun tiedoston, jossa on kuutioita kopioituna. Malli latautui hyvin nopeasti näkyviin 3D-ympäristöön. Ainakin tämä koodi toimii niin kuin pitää. Syynä tähän on varmasti se, että se on koodattu suoraan selaimella suoritettavaksi. Ajattelin vielä kokeilla hieman raskaampaa mallia ja katsoa, tuleeko ThreeJS:llä raja vastaan. Käytin luvussa 3.6 tekemääni miljoonan verteksin mallia. Odotin tämänkokoisen tiedoston jumittavan myös ThreeJS:n tai käsittelyn kestävän todella kauan.

Positiivisena yllätyksenä raskas 3D-mallini tuli ruutuun näkyviin noin kahdenkymmen sekunnin kuluttua. Tämä on selkeä merkki, että tämänkaltaista toimintoa ei kannata tehdä Unityn kanssa. Ainakaan tällä hetkellä saatavilla olevat 3D-mallien lataajat eivät toimi WebGL-ympäristössä, ja syitä siihen voi olla monia. Ehkä koodi on kirjoitettu siten, että se ei käänny hyvin JavaScriptiksi, itse kääntäjä voi olla tehty niin, että se kääntää huonosti tietynlaista C#-koodia JavaScriptiksi tai koodi on optimoimatonta, mistä seuraa suorituskykyongelmia.

5 Tutkimustulosten analysointi ja projektin tulevaisuus

Unity on todella hyvä ja kattava työkalu, joka tarjoaa mahdollisuuden tehdä interaktiivista 3D-sisältöä monelle eri alustalle. Olemme yrityksessä käyttäneet sitä aikaisemminkin visualisointiin. Insinöörityön visualisointityökaluun suunniteltu toiminto, joka nojaa monimutkaiseen 3D-mallien parsimiseen suorituksen aikana, osoittautui sellaisenaan käyttökelvottomaksi WebGL-ympäristössä. Itsenäisenä sovelluksena tuotetussa ohjelmassa lataajia voisi kylläkin käyttää.

Aikaisemmasta kokemuksesta WebGL:n parissa osaan sanoa, että näyttäväkin 3D-sisältö suoriutuu hyvin selaimessa, ja on harmi, että tätä työkalua ei tällä tavalla saatu toimimaan. Työkalun konseptissa on monta mielenkiintoista seikkaa, joihin olisi ollut jän-

nittävää tutustua, jos 3D-mallien parsiminen toimisi selaimessa. Näistä toiminnoista itselleni mielenkiintoisin on ASCII- eli tekstimuotoisten 3D-mallitiedostojen pakkaus ja purkaminen. Tämänkaltaisella toiminnolla voisi vähentää verkon välityksellä kuljetettavan datan määrää helposti yli 50 %.

Käyn Unityn puolesta läpi 3D-mallien latauksen ja parsimisen, vaikka se ei WebGL-ympäristössä toimikaan. Yrityksessä tehdään muutakin 3D-sisältöä, ja suorituksenaikainen lataaminen voi olla hyödyllinen ratkaisu sellaisissa sovelluksissa.

Noin kuuden megatavun FBX-malli latautuu ja prosessoidaan noin viidessä sekunnissa, mikä on oikein hyvä aika. Isompi 15 megatavun FBX-tiedosto latautuu ja prosessoidaan noin neljässä sekunnissa, vaikka se on kolme kertaa suurempi. Tiedostokoolla ei näytä olevan vaikutusta lokaalissa ympäristössä. Latausaikaan voi aina arvioida noin 0,5–2 sekuntia megatavua kohti, joten isommalle 15 megatavun tiedostolle latausaika olisi näillä oletusarvoilla noin 30 sekuntia pahimmillaan.

3D-mallien lataajat eivät muokkausyrityksestä huolimatta toimi selaimessa, joten en pysty testaamaan, miten selainympäristö olisi vaikuttanut prosessointiin, jos ollenkaan. Se ainakin on selvää, että tallainen koodi on raskasta ja erityistilanteessa sen optimointi on ensisijaisen tärkeää. Selaimessa ainakin muistin saatavuus on rajoittava tekijä 3D-sisältöä tehdessä. Oletuksena varattava muisti Unityn WebGL-projektissa on noin puoli gigatavua. Rajaa voi toki nostaa, mutta on täysin selaimesta kiinni voiko se varata WebGL-sovellukselle niin paljoa muistia. (66.)

ThreeJS sen sijaan hämmästytti suorituskyvyllään. Yksinkertainen yhden megatavun kokoinen 3D-malli tuli ladatuksi ja näkyviin ruudulle alle kahdessa sekunnissa. Tämä on oikein hyvä vasteaika. Laitoin ThreeJS:n vielä rajumpaan testiin 150 megatavun mallilla, eikä selain jumiutunut. Testasin latausta perustoimistokoneella, jossa on kaksiydinsuoritin, eikä erillistä näytönohjainta ollenkaan. Lataukseen ja parsimiseen meni yhteensä noin 20 sekuntia, mutta tähän kannattaa lisätä todellisen verkkoyhteyden välityksellä tehtävä latausaika, joka voi olla jopa kymmeniä minuutteja 150 megatavun kokoiselle tiedostolle. Käytettävyys laskee huomattavasti 3D-mallien koon mukaan, koska latausaika kasvaa.

6 Yhteenveto

Insinöörityön tavoitteena ollut visualisointityökalu ei valmistunut, mutta WebGL-kehityksen haasteista saatiin arvokasta kokemusta. Itseäni jäi vielä askarruttamaan 3D-mallien parsimisen ohjelmointitapa, mutta toisaalta näen toimivaksi todetun ThreeJS:n käyttöönoton myös mahdollisena. Opin myös paljon datan pakkaamisesta ja siitä, kuinka paljon tallennustilaa sopivan tekniikan valinnalla voidaan säästää

3D-mallien suorituksenaikaisen lataamisen jatkokehitykselle keksin muutaman mahdollisuuden riippuen siitä, käytetäänkö lataajia verkkoympäristössä vai itsenäisenä sovelluksena. Nämä kehitysideat ovat yrityksessä tietenkin vielä harkinnan varassa. Alustavasti tämän opinnäytetyön tavoitteita ei tällä aikataululla saavutettu. Saatiin kuitenkin paljon tietoa ja kokemusta siitä, miten työkalua kannattaisi lähteä tekemään. Vielä tässä kohtaa ei ole selvillä, lähdetäänkö yrittämään C#-koodien muokkausta ja WebGL-ympäristöön sopimattoman koodin etsimistä ja korjaamista. Toimenpide on työläs, eikä etukäteen ole edes varmuutta, onko se mahdollista. Toinen vaihtoehto on tietenkin ThreeJS:n käyttö, koska se suoriutui testauksesta.

Itsenäisissä sovelluksissa 3D-mallien lataajat kyllä toimivat, mutta materiaalien saaminen objektiin mukaan ei tuntunut toimivan kovin hyvin. FBX-formaattiin pystyy sisällyttämään tekstuureja, mutta niiden muuttaminen materiaaliksi Unityssä riippuu siitä, onko 3D-mallintaja valmistellut mallin asianmukaisesti. Tämä ei sinänsä vaikuta 3D-mallien lataajiin, mutta vaikuttaa työkalun käytettävyyteen ja luonteeseen. Pitää siis vielä miettiä, mikä on työkalun käyttötarve ja miten sen pitää toimia, jos loppukäyttäjä on henkilö, joka ei ymmärrä tai ei voi vaikuttaa käyttämänsä 3D-mallin asianmukaisuuteen.

Verkkoympäristössä ThreeJS-kirjasto osoittautui toimivaksi vaihtoehdoksi 3D-mallien latauksen näkökulmasta. Valitettavasti menetetään kaikki Unityn tuomat hyödyt graafisesta editorista alkaen. Yksinkertaisen sovelluksen tekeminen ThreeJS:llä ei ole liian työläs tehtävä, mutta ominaisuuksien lisääntyessä kehittäminen käy hankalammaksi. Lisäksi yrityksen 3D-animaattorit osaavat käyttää hieman Unityä, ja he voivat dokumentaatiosta katsoa, miten jotakin ominaisuutta, esimerkiksi törmäystunnistusta, käytetään. ThreeJS:llä kaikki muutokset on tehtävä koodissa yksityiskohtaa myöden. Jopa yksittäisen valonlähteen siirto tarkoittaa koordinaatin muuttamista koodissa.

Tähän ongelmaan saattaisi auttaa Unityyn saatavilla oleva lisäosa *Three.js JSON Exporter* , joka mahdollistaa kohtausten viennin JSON-muodossa. Nämä tiedot ovat ladattavissa ThreeJS:ään JSON-lataajalla. Tämä mahdollistaisi sen, että 3D-animaattorit voisivat tehdä ympäristöjä Unityllä. Ohjelmoijat voisivat tuoda JSON-muotoisen ympäristön ThreeJS-sovellukseen. Saatavilla oleva *ThreeJS exporter* on vain tällä hetkellä tuettu vanhemmassa Unityn versiossa, mitä yrityksessä käytetään, joten tällä hetkellä ratkaisuksi siitä ei ole. Vaikka tässä vaiheessa aikataulu loppui kesken, 3D-mallien latauksen tutkimista jatketaan tässä työssä saatujen kokemusten mukaisesti.

Lähteet

1 WebGL Getting Started. 2011. Verkkodokumentti. Khronos Group. <https://www.khronos.org/webgl/wiki/Getting_Started>. 10.4.2011. Luettu 16.3.2017.

2 WebGL - OpenGL ES for the Web. 2017. Verkkodokumentti. Khronos Group. <https://www.khronos.org/webgl/>. Luettu 16.3.2017.

3 OpenGL Overview. 2016. Verkkodokumentti. Khronos Group. <https://www.opengl.org/about/#1>. Luettu 16.3.2017.

4 The Standard for Embedded Accelerated 3D Graphics. 2017. Verkkodokumentti. Khronos Group. < https://www.khronos.org/opengles/>. Luettu 16.3.2017.

5 Getting started with WebGL. 2017. Verkkodokumentti. Mozilla säätiö. <https://developer.mozilla.org/fi/docs/Web/API/WebGL_API/Tutorial/Getting_started_with_WebGL>. 23.2.2017. Luettu 17.3.2017.

6 Khronos Overview. 2013. Verkkodokumentti. Khronos Group. <https://www.slideshare.net/Khronos_Group/khronos-overviewnov13>. 21.11.2013. Luettu 17.3.2017.

7 Signer, Beat. 2012. HTML5 and the Open Web Platform. Verkkodokumentti. <https://www.slideshare.net/signer/html5-and-the-open-web-platform>. Luettu 18.3.2017.

8 Wallace, Evan. 2016. Rendering Realtime Caustics in WebGL. Verkkodokumentti. <https://medium.com/@evanwallace/rendering-realtime-caustics-in-webgl-2a99a29a0b2c#.9isvmxpwa>. 7.1.2016. Luettu 18.3.2017.

9 Wallace, Evan. 2011. WebGL Water. Verkkodokumentti <http://madebyevan.com/webgl-water/>. Luettu 17.3.2017.

10 Sawicki, Adam. 2015. Lower-Level Graphics API - What Does It Mean? Verkkodokumentti. <http://asawicki.info/news_1601_lower-level_graphics_api_-_what_does_it_mean.html>. 6.7.2015. Luettu 5.4.2017.

11 Getting Started. 2011. Verkkodokumentti. Khronos Group. <https://www.khronos.org/webgl/wiki/Getting_Started>. 10.4.2011. Luettu 17.3.2017.

12 Hello_Triangle.c. 2013. Verkkodokumentti. Github. <https://raw.githubusercontent.com/danginsburg/opengles3-book/master/Chapter_2/Hello_Triangle/Hello_Triangle.c>. 25.10.2013. Luettu 18.3.2017.

13 User Contributions. 2016. Verkkodokumentti. Khronos Group. <https://www.khronos.org/webgl/wiki/User_Contributions#Frameworks>. 19.6.2016. Luettu 16.3.2017.

14 Whitestrom JS. Verkkodokumentti. <https://whsjs.io/#/>. Luettu 19.3.2017.

15 Featured projects. 2017. Verkkodokumentti. Threejs.org. <https://threejs.org/>. Luettu 18.3.2017.

16 Three.js. 2017. Verkkodokumentti. Github. <https://github.com/mrdoob/three.js/>. 4.5.2017. Luettu 19.3.2017.

17 Creating a Scene. 2017. Verkkodokumentti. Three.js <https://threejs.org/docs/index.html#Manual/Getting_Started/Creating_a_scene>. Luettu 19.3.2017.

18 Babylon JS. 2017. Verkkodokumentti. <http://www.babylonjs.com>. Luettu 19.3.2017.

19 Babylon.js. 2017. Verkkodokumentti. Github. <https://github.com/BabylonJS/Babylon.js>. 3.5.2017. Luettu 19.3.2017.

20 Getting Started. 2017. Verkkodokumentti. BabylonJS.com <http://doc.babylonjs.com/>. Luettu 19.3.2017.

21 Unleashing the Power of 3D Internet. 2017. Verkkodokumentti. Blend4Web. <https://www.blend4web.com/en/>. Luettu 19.3.2017.

22 Unity GDC 2017 Keynote. 2017. Verkkodokumentti. Unity Technologies. <https://youtu.be/YHweZ8dhOJA?t=402>. Katsottu 19.3.2017.

23 Unity Asset Store. 2017. Verkkodokumentti. Unity Technologies. <https://www.assetstore.unity3d.com/en/>. Luettu 19.3.2017.

24 Unity. 2017. Verkkodokumentti. Unity Technologies. <https://unity3d.com/unity>. Luettu 19.3.2017.

25 Extending the Editor. 2017. Verkkodokumentti. Unity Technologies. <https://docs.unity3d.com/Manual/ExtendingTheEditor.html>. 21.4.2017. Luettu 20.3.2017.

26 The State of Unity on Linux. 2015. Verkkodokumentti. GamingOnLinux. <https://www.gamingonlinux.com/uploads/articles/article_images/1436702259blacksmith-2.png>. Luettu 20.3.2017.

27 Prakhov, Andrey. 2016. Blend4Web: Beginner's Guide. Chapter 7: Labyrinth Of Logic. Verkkodokumentti. <https://www.blend4web.com/en/community/article/307/>. 23.12.2016. Luettu 19.3.2017.

28 Getting Started: Developing HTML5 Projects. 2017. Verkkodokumentti. Epic Games. <https://docs.unrealengine.com/latest/INT/Platforms/HTML5/GettingStarted/index.html>. Luettu 20.3.2017.

29 WebGL and WebVR. 2016. Verkkodokumentti. Unity Technologies. <https://forum.unity3d.com/threads/webgl-and-webvr.390445/>. Luettu 18.3.2017.

30 WebVR - Bringing Virtual Reality to the Web. 2017. Verkkodokumentti. WebVR.info. <https://webvr.info/>. Luettu 20.3.2017.

31 History of OpenGL. 2017. Verkkodokumentti. Khronos Group. <https://www.khronos.org/opengl/wiki/History_of_OpenGL>. 13.1.2017. Luettu 17.3.2017.

32 Vukićević, Vladimir. 2007. Canvas 3D: GL poer, web-style. Verkkodokumentti. <https://web.archive.org/web/20110717224855/http://blog.vlad1.com/2007/11/26/canvas-3d-gl-power-web-style/>. 26.11.2007. Luettu 17.3.2017.

33 Brown, Wayne. 2015. Computer Graphics – A Breaf History. Verkkodokumentti. <http://learnwebgl.brown37.net/the_big_picture/webgl_history.html>. 1.2.2016. Luettu 17.3.2017.

34 Loader / FBX. 2017. Verkkodokumentti. Threejs.org. <https://threejs.org/examples/#webgl_loader_fbx>. Luettu 17.3.2017.

35 ThreeJS Editor. 2017. Verkkodokumentti. Threejs.org. <https://threejs.org/editor/>. Luettu 18.3.2017.

36 Unity Manual. 2017. Verkkodokumentti. Unity Technologies. <https://docs.unity3d.com/Manual/UnityManual.html>. 21.4.2017. Luettu 18.3.2017.

37 Holographic Academy. 2017. Verkkodokumentti. Microsoft. <https://developer.microsoft.com/en-us/windows/mixed-reality/academy>. Luettu 22.3.2017.

38 Manual Sections. 2017. Verkkodokumentti. Unity Technologies. <https://docs.unity3d.com/Manual/index.html>. 21.4.2017. Luettu 18.3.2017.

39 FBXLoader.js. 2017. Verkkodokumentti. Github. <https://github.com/mrdoob/three.js/blob/master/examples/js/loaders/FBXLoader.js>. 7.3.2017. Luettu 20.3.2017.

40 Runtime OBJ Importer. 2016. Verkkodokumentti. Unity Technologies. <https://www.assetstore.unity3d.com/en/#!/content/49547>. Luettu 20.3.2017.

41 ObjReader. 2015. Starscene Software. Verkkodokumentti. Unity Technologies. <https://www.assetstore.unity3d.com/en/#!/content/152>. Luettu 20.3.2017.

42 Sarria, Carlos. 2017. UniFBX (2). Verkkodokumentti. <https://www.assetstore.unity3d.com/en/#!/content/61108>. Luettu 20.3.2017.

43 How do I import models from my 3D app? 2017. Verkkodokumentti. Unity Technologies. <https://docs.unity3d.com/Manual/HOWTO-importObject.html>. Luettu 26.3.2017.

44 FBX binary file format specification. 2010. Verkkodokumentti. Blender Dev Blog. <https://code.blender.org/2013/08/fbx-binary-file-format-specification/>. 10.8.2013. Luettu 26.3.2017.

45 Autodesk FBX. 2017. Verkkodokumentti. Autodesk. <http://usa.autodesk.com/adsk/servlet/pc/item?siteID=123112&id=10775847>. Luettu 26.3.2017.

46 Bourke, Paul. 2017. Object Files. Verkkodokumentti. <http://paulbourke.net/dataformats/obj/>. Luettu 26.3.2017.

47 Importing Objects from Blender. 2017. Verkkodokumentti. Unity Technologies. <https://docs.unity3d.com/Manual/HOWTO-ImportObjectBlender.html>. 21.4.2017. Luettu 26.3.2017.

48 Importing Objects From 3D Studio Max. 2017. Verkkodokumentti. Unity Technologies. <https://docs.unity3d.com/Manual/HOWTO-ImportObjectMax.html> 21.4.2017. Luettu. 26.3.2017.

49 Asset Bundles. 2017. Verkkodokumentti. Unity Technologies. <https://docs.unity3d.com/Manual/AssetBundlesIntro.html>. 21.4.2017. Luettu 26.3.2017.

50 Asset Bundle Compression. 2017. Verkkodokumentti. Unity Technologies. <https://docs.unity3d.com/Manual/AssetBundleCompression.html>. Luettu 26.3.2017.

51 WebGL: Interacting with browser scripting. 2017. Verkkodokumentti. Unity Technologies. <https://docs.unity3d.com/Manual/webgl-interactingwithbrowserscripting.html> 21.4.2017. Luettu 26.3.2017.

52 LZMA SDK. 2016. Verkkodokumentti. 7-zip.org. <http://www.7-zip.org/sdk.html>. Luettu 29.3.2017.

53 7-Zip. 2016. Verkkodokumentti. 7-zip.org. <http://www.7-zip.org/>. Luettu 29.3.2017.

54 XZ Utils. 2016. Verkkodokumentti. tukaani.org. <http://tukaani.org/xz/> Luettu 29.3.2017.

55 New Unity WebGL Embedding API White Paper. 2016. Verkkodokumentti. Unity Technologies. <https://forum.unity3d.com/threads/new-unity-webgl-embedding-api-white-paper.430909/>. 13.9.2016. Luettu 3.4.2017.

56 Input in WebGL. 2017. Verkkodokumentti. Unity Technologies. <https://docs.unity3d.com/Manual/webgl-input.html>. 21.4.2017. Luettu 3.4.2017.

57 HTTP access control (CORS). 2017. Verkkodokumentti. Mozilla Developer Network. <https://developer.mozilla.org/en-US/docs/Web/HTTP/Access_control_CORS>. 18.4.2017. Luettu 3.4.2017.

58 WebGL "out of memory" + Unity 5.3.4f1. 2016. Verkkodokumentti. Unity Technologies. <https://forum.unity3d.com/threads/webgl-out-of-memory-unity-5-3-4f1.396568/>. 8.4.2016. Luettu 5.4.2017.

59 WebGL Networking. 2017. Verkkodokumentti. Unity Technologies. <https://docs.unity3d.com/Manual/webgl-networking.html>. 21.4.2017. Luettu 5.4.2017.

60 Debugging and troubleshooting WebGL builds. 2017. Verkkodokumentti. Unity Technologies. <https://docs.unity3d.com/Manual/webgl-debugging.html>. 21.4.2017. Luettu 3.4.2017.

61 IDisposable Interface. 2017. Verkkodokumentti. Microsoft Developer Network. <https://msdn.microsoft.com/en-us/library/system.idisposable(v=vs.110).aspx>. Luettu 7.4.2017.

62 Onder, Hulusi. 2016. Everything you need to know about Unity WebGL Building. Verkkodokumentti. <https://www.linkedin.com/pulse/everything-you-need-know-unity-webgl-building-hulusi-onder>. 20.2.2016. Luettu 17.3.2017.

63 Array.Resize<T>. 2017. Verkkodokumentti. Microsoft Developer Network. <https://msdn.microsoft.com/en-us/library/bb348051(v=vs.110).aspx>. Luettu 7.4.2017.

64 OBJLoader.js. 2017. Verkkodokumentti. Github. <https://github.com/mrdoob/three.js/blob/dev/examples/js/loaders/OBJLoader.js> Luettu 7.4.2017.

65 Loader / Obj. 2017. Verkkodokumentti. Threejs.org. <https://threejs.org/examples/#webgl_loader_obj>. Luettu 7.4.2017.

66 Memory Considerations when targeting WebGL. 2017. Verkkodokumentti. Unity Technologies. <https://docs.unity3d.com/Manual/webgl-memory.html>. 21.4.2017. Luettu 3.4.2017.

67 Embed Media. 2017. Verkkodokumentti. Autodesk. <https://knowledge.autodesk.com/support/3ds-max/learn-explore/caas/CloudHelp/cloudhelp/2017/ENU/3DSMax/files/GUID-FB993022-FD65-4ABD-8AF3-7F2CF613D71D-htm.html>. 7.4.2017. Luettu 7.4.2017.

68 How to import models with textures applied. 2015. Verkkodokumentti. Unity Technologies. <http://answers.unity3d.com/questions/886683/how-to-import-models-with-textures-applied-fbx-emb.html>. 2.2.2015. Luettu 7.4.2015.

69 Janssen, Nick. 2017. Three.js JSON Exporter. Verkkodokumentti. <https://www.assetstore.unity3d.com/en/#!/content/40550>. 20.1.2017. Luettu 9.4.2017.

70 Räihä, Kaarlo. 2015. Lzma-unity. Verkkodokumentti. <https://bitbucket.org/Agent_007/lzma-unity/src/4bb69b5b7353?at=master>. Luettu 29.3.2017.

Koodiesimerkki 1 - Hello_Triangle.c

Koodiesimerkki kolmion piirtämisestä OpenGL ES ohjelmoinnilla

```
// The MIT License (MIT)
//
// Copyright (c) 2013 Dan Ginsburg, Budirijanto Purnomo
//
// Permission is hereby granted, free of charge, to any person obtaining a copy
// of this software and associated documentation files (the "Software"), to deal
// in the Software without restriction, including without limitation the rights
// to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
// copies of the Software, and to permit persons to whom the Software is
// furnished to do so, subject to the following conditions:
//
// The above copyright notice and this permission notice shall be included in
// all copies or substantial portions of the Software.
//
// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
// IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
// FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
THE
// AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
// LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
FROM,
// OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
// THE SOFTWARE.

//
// Book:      OpenGL(R) ES 3.0 Programming Guide, 2nd Edition
// Authors:   Dan Ginsburg, Budirijanto Purnomo, Dave Shreiner, Aaftab Munshi
// ISBN-10:   0-321-93388-5
// ISBN-13:   978-0-321-93388-1
// Publisher: Addison-Wesley Professional
// URLs:      http://www.opengles-book.com
//            http://my.safaribooksonline.com/book/animation-and-3d/9780133440133
//
// Hello_Triangle.c
//
//    This is a simple example that draws a single triangle with
//    a minimal vertex/fragment shader.  The purpose of this
//    example is to demonstrate the basic concepts of
//    OpenGL ES 3.0 rendering.
#include "esUtil.h"

typedef struct{
   // Handle to a program object
   GLuint programObject;
} UserData;
///
// Create a shader object, load the shader source, and
// compile the shader.
//
GLuint LoadShader ( GLenum type, const char *shaderSrc ){
   GLuint shader;
   GLint compiled;
   // Create the shader object
```

```
   shader = glCreateShader ( type );
   if ( shader == 0 ){
      return 0;
   }
   // Load the shader source
   glShaderSource ( shader, 1, &shaderSrc, NULL );
   // Compile the shader
   glCompileShader ( shader );
   // Check the compile status
   glGetShaderiv ( shader, GL_COMPILE_STATUS, &compiled );
   if ( !compiled ){
      GLint infoLen = 0;
      glGetShaderiv ( shader, GL_INFO_LOG_LENGTH, &infoLen );
      if ( infoLen > 1 ){
         char *infoLog = malloc ( sizeof ( char ) * infoLen );
         glGetShaderInfoLog ( shader, infoLen, NULL, infoLog );
         esLogMessage ( "Error compiling shader:\n%s\n", infoLog );
         free ( infoLog );
      }
      glDeleteShader ( shader );
      return 0;
   }
   return shader;
}
///
// Initialize the shader and program object
//
int Init ( ESContext *esContext )
{
   UserData *userData = esContext->userData;
   char vShaderStr[] =
      "#version 300 es                          \n"
      "layout(location = 0) in vec4 vPosition;  \n"
      "void main()                              \n"
      "{                                        \n"
      "   gl_Position = vPosition;              \n"
      "}                                        \n";
   char fShaderStr[] =
      "#version 300 es                          \n"
      "precision mediump float;                 \n"
      "out vec4 fragColor;                      \n"
      "void main()                              \n"
      "{                                        \n"
      "   fragColor = vec4 ( 1.0, 0.0, 0.0, 1.0 );  \n"
      "}                                        \n";
   GLuint vertexShader;
   GLuint fragmentShader;
   GLuint programObject;
   GLint linked;
   // Load the vertex/fragment shaders
   vertexShader = LoadShader ( GL_VERTEX_SHADER, vShaderStr );
   fragmentShader = LoadShader ( GL_FRAGMENT_SHADER, fShaderStr );
   // Create the program object
   programObject = glCreateProgram ( );
   if ( programObject == 0 )
   {
      return 0;
   }
   glAttachShader ( programObject, vertexShader );
   glAttachShader ( programObject, fragmentShader );
```

```
   // Link the program
   glLinkProgram ( programObject );
   // Check the link status
   glGetProgramiv ( programObject, GL_LINK_STATUS, &linked );
   if ( !linked ){
      GLint infoLen = 0;
      glGetProgramiv ( programObject, GL_INFO_LOG_LENGTH, &infoLen );
      if ( infoLen > 1 )
      {
         char *infoLog = malloc ( sizeof ( char ) * infoLen );
         glGetProgramInfoLog ( programObject, infoLen, NULL, infoLog );
         esLogMessage ( "Error linking program:\n%s\n", infoLog );
         free ( infoLog );
      }
      glDeleteProgram ( programObject );
      return FALSE;
   }
   // Store the program object
   userData->programObject = programObject;
   glClearColor ( 1.0f, 1.0f, 1.0f, 0.0f );
   return TRUE;
}
///
// Draw a triangle using the shader pair created in Init()
//
void Draw ( ESContext *esContext ){
   UserData *userData = esContext->userData;
   GLfloat vVertices[] = {  0.0f,  0.5f, 0.0f,
                           -0.5f, -0.5f, 0.0f,
                            0.5f, -0.5f, 0.0f
                         };
   // Set the viewport
   glViewport ( 0, 0, esContext->width, esContext->height );
   // Clear the color buffer
   glClear ( GL_COLOR_BUFFER_BIT );
   // Use the program object
   glUseProgram ( userData->programObject );
   // Load the vertex data
   glVertexAttribPointer ( 0, 3, GL_FLOAT, GL_FALSE, 0, vVertices );
   glEnableVertexAttribArray ( 0 );
   glDrawArrays ( GL_TRIANGLES, 0, 3 );
}
void Shutdown ( ESContext *esContext ){
   UserData *userData = esContext->userData;
   glDeleteProgram ( userData->programObject );
}
int esMain ( ESContext *esContext ){
   esContext->userData = malloc ( sizeof ( UserData ) );
   esCreateWindow ( esContext, "Hello Triangle", 320, 240, ES_WINDOW_RGB );
   if ( !Init ( esContext ) )
   {
      return GL_FALSE;
   }
   esRegisterShutdownFunc ( esContext, Shutdown );
   esRegisterDrawFunc ( esContext, Draw );
   return GL_TRUE;
}
```