Aalto-yliopisto
Perustieteiden korkeakoulu
Tietotekniikan koulutusohjelma

# WebGL-rajapinta pelinkehittäjän näkökulmasta

**Kandidaatintyö**

**13. huhtikuuta 2014**

**Atte Isopuro**

Aalto-yliopisto
Perustieteiden korkeakoulu
Tietotekniikan koulutusohjelma

KANDIDAATINTYÖN
TIIVISTELMÄ

| | |
|---|---|
| **Tekijä:** | Atte Isopuro |
| **Työn nimi:** | WebGL-rajapinta pelinkehittäjän näkökulmasta |
| **Päiväys:** | 13. huhtikuuta 2014 |
| **Sivumäärä:** | 27 |
| **Pääaine:** | Ohjelmistotekniikka |
| **Koodi:** | T3001 |
| **Vastuuopettaja:** | Ma professori Tomi Janhunen |
| **Työn ohjaaja(t):** | Professori Jukka Nurminen (Tietoliikenneohjelmistot) |

Tässä kandidaatintyössä tarkastellaan HTML5-specifikaation WebGL-rajapintaa pelinkehittäjän näkökulmasta. Työssä tutkitaan kirjallisuuskatsauksen avulla WebGL-rajapinnan erityispiirteitä ja soveltuvuuksia.

Aikaisemmista tutkimuksista havaitaan että pelien graafisen esityksen laadulla on vaikutus intensiivisten pelien pelattavuuteen. Varsinkin ruudun päivitystiheyden huomataan haittaavan pelattavuutta ollessaan matala.

WebGL on huomattavasti nopeampi kuin Canvas 2D Context-toteutukset ja selvästi hitaampi kuin natiivi OpenGL/C++-toteutus. Mittaustuloksia tutkimalla huomataan eron johtuvan pääosin JavaScript-koodin hitaudesta.

Asynkronista lataamista ja matriisilaskentaa ei ole toteutettu itse kirjastoon. Todetaan että puutteet on joko korjattava itse tai on käytettävä ulkoista kirjastoa. Itse koodi on monimutkaisempaa kuin Canvas 2D Context-koodi, mutta ei eroa suuresti natiivista OpenGL-koodista.

WebGL-sovellusten huomataan myös olevan riippuvaisempia alustoista kuin Canvas 2D Context-sovellukset. Eri selainten välillä on toiminnallisuuseroja. Erilaiset selain-näytönohjain yhdistelmät voivat myös olla kokonaan toimimattomia. Todetaan että WebGL on alustasta riippumattomampi kuin OpenGL-sovellus. Kuitenkin WebGL-kehittäjän kannattaa huomioida että sovellus ei yllä yksinkertaisemman Canvas 2D Context-sovelluksen alustariippumattomuuteen.

WebGL-sovellukset toimivat parhaiten korkeatasoisen natiivin ja yksinkertaisen verkkosovelluksen välimaastossa. Kehittäjän ei suositella pyrkiä natiivin sovelluksen suorituskykyyn eikä myöskään olettaa yksinkertaisemman pelin alustariippumattomuutta.

| | |
|---|---|
| **Avainsanat:** | webgl, html5, peli, pelinkehitys, suorituskyky, ongelmia, edut |
| **Kieli:** | Suomi |

# Sisältö

# 1 Johdanto

Tämä kandidaatintyö käsittelee HTML5:n WebGL-ohjelmointirajapintaa pelinkehittäjän näkökulmasta.

HTML5-spesifikaatio on mahdollistanut kaikenlaisen median esittämisen verkossa ilman, että käyttäjän tarvitsee asentaa erillisiä lisäosia. Yksi näistä uusista rajapinnoista on WebGL, jonka kautta verkkosivun on mahdollista käyttää tietokoneen näytönohjainta teoriassa alustasta riippumatta.

Erityisesti monimutkaista ja tarkkaa interaktiota vaativat pelit tarvitsevat nopeaa grafiikan piirtämistä ruudulle. Jokaiselle alustalle erikseen räätälöidyt ratkaisut ovat kuitenkin kalliita ja hankalia ylläpitää, joten alustasta riippumaton grafiikkaohjelmointi on erittäin kiinnostavaa tästä näkökulmasta.

Kirjallisuudessa on käsitelty WebGL:n käyttömahdollisuuksia erinäisillä aloilla. Useimmat tällaiset käsittelyt ovat kuitenkin yksittäisiä sovelluksia. Yleistä arviota WebGL:n eduista ja haitoista pelinkehityksessä ei ole tehty.

Tämän kandidaatintyön tavoitteena on kerätä pelinkehittäjän kannalta hyödyllisiä havaintoja WebGL:stä. Työssä pyritään tunnistamaan sellaiset WebGL:n ominaispiirteet jotka pelinkehittäjän tulisi ottaa huomioon. Tämä työ ei tule käsittelemään muita selaimen grafiikan esitykseen liittyviä teknologioita, kuten Canvas 2D context tai Flash. Teknologioita verrataan WebGL-rajapintaan, mutta niiden erityispiirteitä ei tutkita tässä työssä tarkemmin.

Työ toteutetaan kirjallisuuskatsauksena. Aineisto koostuu yliopistojen julkaisuista, tieteellisistä artikkeleista ja kirjoista jotka käsittelevät WebGL-rajapintaa. Erityisesti viitataan teksteihin joissa on konkreettisten toteutusten yhteydessä havaittu WebGL:n ominaispiirteitä. Aiheesta ei ole löytynyt suuria määriä kirjallisuutta, joten tieteelliset lähteet on tulkittu hyväksyttäviksi kunhan ne on julkaistu jonkin yliopiston toimesta tai ne löytyvät Scopus-tietokannasta.

Ensiksi alustetaan aiheen taustaa: esitellään HTML5, JavaScript, grafiikan piirtämisen rajapintoja. Seuraavaksi esitellään lyhyesti WebGL-rajapinnalla tehtyjä sovelluksia. Tämän jälkeen eritellään pelinkehittäjän tarpeita ja käydään läpi WebGL-rajapinnasta tehdyt havainnot hänen näkökulmasta. Lopuksi havainnoista tehdään lyhyt yhteenveto, jossa esitetään suositus minkälaisiin projekteihin WebGL-rajapinnan käyttö soveltuu.

# 2 Aiheeseen liittyvät teknologiat

## 2.1 Selainteknologiat

Seuraavaksi esitellään tämän työn kannalta keskeisimmät verkkoselaimissa käytetyt teknologiat.

### 2.1.1 HTML5

HTML5 on uusin versio **H**yper**T**ext **M**arkup **L**anguage-kielestä[1]. Yksi HTML5-kielen tuomia uudistuksia on erinäisten media-formaattien sisäistäminen itse spesifikaatioon erinäisten rajapintojen kautta[2]. Näin ollen sisällöntuottajien ja käyttäjien ei tarvitse huolehtia kolmansien osapuolten liitännäisistä kuten Flash. Tällöin sisällöntuottaja voi olla varma siitä että kaikki ne joilla on spesifikaatiota noudattava selain pääsevät käsiksi hänen tuottamaan sisältöön. Täten alustojen erilaisuuksien ei pitäisi ainakaan teoriassa vaikuttaa sovellukseen, jolloin sisällöntuottaja voi keskittyä luomaan sisältöä ainoastaan yhdelle alustalle.

### 2.1.2 JavaScript

JavaScript-kieli on laajasti käytössä oleva ohjelmointikieli. Kieli on toteutus ECMAScript-standardista[3] jota käytetään pääasiassa verkkosivujen ohjelmointiin[4]. Kieli mahdollistaa verkkosivujen reaaliaikaisen muuttamisen, animoinnit sekä käyttäjän interaktiot. Useimmat HTML5-kielen määrittämät rajapinnat ovat juuri JavaScript-ohjelmia varten tarkoitettuja.

### 2.1.3 Canvas-elementti

`<canvas>`-elementti on HTML5-spesifikaation määrittelemä elementti jonka voi sisällyttää verkkosivulleen HTML5-koodissa. Elementin tarkoitus on toimia piirtoalustana: ohjelmoija voi rajapintojen kautta piirtää `<canvas>`-elementille grafiikkaa. `<canvas>`-elementin piirtämiseen tarkoitetut rajapinnat ovat tämän työn käsittelemä WebGL-rajapinta ja yksinkertaisempi Canvas 2D Context.[5]

## 2.2 Intensiivinen grafiikka ja näytönohjaimen rajapinnat

Intensiivisellä grafiikalla tarkoitetaan tässä yhteydessä grafiikkaa jonka piirtäminen vaatii huomattavan määrän laskentatehoa. Esimerkiksi yksittäisen digitaalisen valokuvan piirtäminen ruudulle ei ole intensiivistä grafiikkaa. Fysikaalisen ilmiön (kuten veden)

simuloiminen ja piirtäminen ruudulle taas on hyvin intensiivistä ja voi veden liikkeistä ja koostumuksesta riippuen vaatia suuria määriä laskentatehoa. 3D-grafiikka on melkein aina paljon raskaampaa kuin 2D-grafiikka. Kuitenkin piirron intensiivisyys riippuu ainoastaan laskennan määrästä. 2D-kuva jossa on paljon objekteja ja korkea resoluutio voi myös olla raskas piirtää. Intensiivinen grafiikka ei siis rajoitu ainoastaan 3D-grafiikkaan.

### 2.2.1 Peligrafiikka

Pelialalla intensiivinen grafiikka yhdistyy useimmiten fysiikan simulointiin, kolmiuloitteisiin ympäristöihin ja monimutkaisiin esineisiin pelimaailmassa. Vaikka tällaisia asioita voidaan periaatteessa simuloida tietokoneen pääsuorittimella, teho jää nopeasti riittämättömäksi. Liian intensiivinen grafiikka hidastaisi konetta sen verran että käyttökokemus kärsii. Graafisesti monimutkaisten pelien laskenta on näin ollen siirrettävä yleispätevästä suorittimesta näytönohjaimeen, joka on graafiseen laskentaan erikoistunut laitekomponentti.

### 2.2.2 Näytönohjain

Näytönohjain on tietokoneeseen liitettävä erillinen lisälaite, jolla on oma muisti ja oma graafiseen laskentaan erikoistunut suoritin. Näin ollen laskenta joka suoritetaan näytönohjaimessa ei käytä yhtä paljon tietokoneen resursseja. Näytönohjaimen käyttäminen koodista on kuitenkin monimutkaisempaa kuin pelkän suorittimen käskyttäminen. Ohjelmoijan on varmistettava, että käskyjen lisäksi myös tarvittava muisti siirretään näytönohjaimelle. Näytönohjain on useimmiten myös erillinen komponentti, joten käyttöjärjestelmä tarvitsee erillisen ajurin tai ohjaimen voidakseen käyttää sitä. Näitä ajureita voidaan käyttää erityisillä grafiikkaohjelmointiin tarkoitetuilla rajapinnoilla.

### 2.2.3 OpenGL-rajapinta

OpenGL (**Open G**raphics **L**ibrary) on alusta- ja kieliriippumaton ohjelmointirajapinta tietokoneiden näytönohjaimille[6]. Ohjelmoija voi käyttää OpenGL-kirjastoa tehdäkseen kutsuja näytönohjaimelle, ilman että hänen tarvitsee käyttää näytönohjaimen omaa, matalan tason käskykantaa. Vastaava teknologia on DirectX[7], Microsoftin ylläpitämä lisensioitava rajapinta-kirjasto, joka on tarkoitettu yksinomaan Windows-käyttöjärjestelmälle.

### 2.2.4 OpenGL ES-rajapinta

OpenGL ES (**E**mbedded **S**ystems) on OpenGL-kirjaston versio joka on tarkoitettu käytettäväksi sulautetuissa järjestelmissä, kuten pelikonsoleissa, puhelimissa ja tablettitietokoneissa. OpenGL ES on rajatumpi versio täydestä OpenGL-kirjastosta, jotta se olisi mahdollisimman laajasti käytettävä.[8]

### 2.2.5 WebGL-rajapinta

WebGL (**Web G**raphics **L**ibrary) on JavaScript-rajapinta `<canvas>`-elementtiä varten. WebGL perustuu pitkälti OpenGL ES:ään[9]. Näin ollen WebGL on samalla lailla rajattu: näitä rajallisuuksia tarkastellaan tarkemmin kappaleessa 5.2. Rajapinta sallii näytönohjaimen käskyttämisen JavaScript-koodista. JavaScript-ohjelmoija voi siis piirtää `<canvas>`-elementille kuvia WebGL-rajapinnan kautta samalla tavoin kuin natiivisovellus tekisi esimerkiksi C++-kieltä ja OpenGL-rajapintaa käyttäen. Canvas 2D Context-rajapinta voi ainoastaan käyttää suoritinta, eikä sitä ole tarkoitettu monimutkaisen 3D-grafiikan piirtoon.

# 3 WebGL-rajapinnan nykytilanne

WebGL-rajapinnalla on toteutettu monia erilaisia projekteja. Jimenez tutkimusryhmineen[10] sekä Mobeen ja Feng[11] ovat toteuttaneet kolmiuloitteisen tiedon visualisointimenetelmiä. Kyseiset toteutukset mahdollistavat kolmiuloitteisten mallien tutkimisen selaimessa, erityisesti lääketieteellisissä tutkimuksissa käytettäviä malleja. Näin lääkärit voivat tutkia esimerkiksi magneettikuvantamisessa tuotettuja malleja potilaan aivoista.

WebGL-rajapinnalla on myös tehty tuotteiden esikatseluun tarkoitettuja sovelluksia. Esimerkiksi [12] on autojen esikatselua varten tarkoitettu sovellus. Priyopradono ja Perdana[13] ovat tehneet asuntojen virtuaalista esikatselua varten tarkoitetun sovelluksen. Vaikka kummatkaan sovellukset eivät ole kaupallisessa käytössä ne esimerkillistävät WebGL-rajapinnan mahdollisuuksia.

Tämän työn kannalta kiinnostavimmat sovellukset ovat kuitenkin WebGL-rajapinnalla tehdyt pelit, joita on monia[14]. Tunnetuimpia lienee Google:n itse teettämä WebGL-versio Quake 2-pelistä[15]. Pelin iästä huolimatta kyseessä on pelikokemus joka vaatii nopeaa ja tasaista grafiikan piirtoa, joten pelin toteuttaminen ilman näytönohjainta ei luultavasti olisi mahdollista suurimmassa osassa nykyisiä tietokoneita.

Kuitenkaan WebGL-versioiden tekeminen peleistä ei vaikuta olevan suuressa suosiossa. Aiheesta ei löydy kirjallisuutta, mutta WebGL.com-sivuston katsotuimpien projektien listasta[14] voi vetää jonkinlaisia johtopäätöksiä: viiden katsotuimman projektin joukossa on yksi peli, joka on Unreal Engine 3-pelimoottorin esittelyyn tarkoitettu demo[16]: loput ovat jollain muulla tapaa WebGL-rajapintaa käyttäviä projekteja.

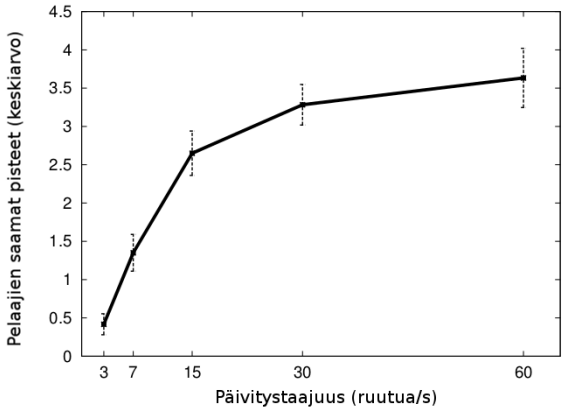
Syitä epäsuosiolle saattaa olla monia, mutta tärkein saattaa olla rajapinnan nuoruus. Ensinnäkin teknologiaa ei ole välttämättä saatu toimimaan kunnolla kaikissa selaimissa: eroavaisuuksista selainten välillä keskustellaan enemmän kappaleessa 5.4. Erot saattavat antaa vaikutelman, että teknologia ei vielä ole täysin valmis. Suurempi syy saattaa kuitenkin olla, että WebGL ei sovellu mihinkään olemassaolevaan toimintatapaan. WebGL-rajapinta on sinänsä kuin Canvas 2D Context: se mahdollistaa grafiikan piirtämisen selaimessa.

WebGL-pelit vaikuttavat kuitenkin olevan työläämpiä tuottaa kuin Canvas 2D Context-rajapinnalla. Näin ollen ilmaisten WebGL-pelien tekeminen ei välttämättä ole järkevää. Kuitenkin useimmat selainpelit ovat maksuttomia, eivätkä ihmiset välttämättä ole tähän mennessä halunneet maksaa. WebGL-koodin monimutkaisuus saattaa myös muodostaa kynnyksen jonka ylittäminen ei ole kokeellisten tai taiteellisten pelien kehittäjille mielekästä. Suuremmat firmat taas perustavat toimintamallinsa pitkälti tuotteistamiseen, jossa yksi peli myydään yhdelle ostajalle. WebGL-rajapinnan toiminnallisuus tekee

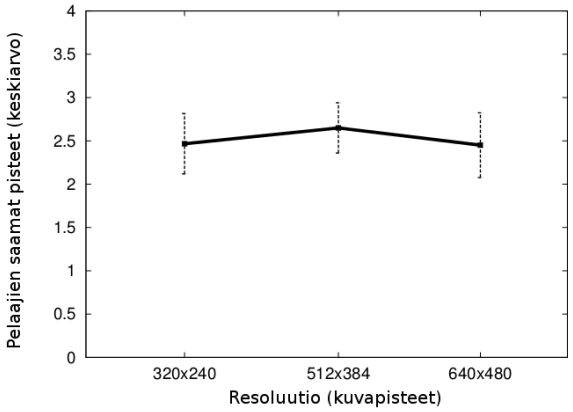
tällaisen mahdottomaksi, sillä pelin lähdekoodi on pakko jakaa avoimesti sillä verkkosivulla jolla peliä pelataan. Uudet toimintamallit saattavat toimia paremmin WebGL-peleille, mutta yllä mainitut syyt voivat selittää WebGL-rajapinnan nykyisen epäsuosion.

# 4 Pelinkehittäjän tarpeet

Erityisesti nopeita reaktioita vaativien pelien pelattavuus kärsii huomattavasti liian pienillä ruudun päivitysnopeuksilla. Ero pelaajan pelaamiskyvyssä korkeiden ja matalien piirtotaajuuksien välillä on huomattava: kuvat 1 ja 2 havainnollistavat Claypoolin tutkimusryhmän[17] saamia tuloksia tutkittaessa kuvanlaadun ja piirtotiheyden vaikutusta pelaajiin.

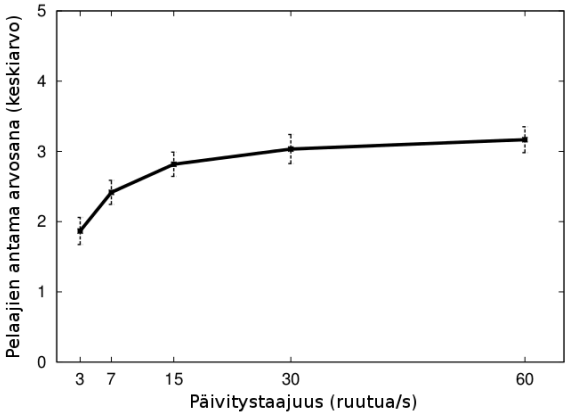



**(a)** Pelaajien saamat pisteet suhteutettuna päivitystiheyteen. Resoluutio 512x384 kuvapistettä.

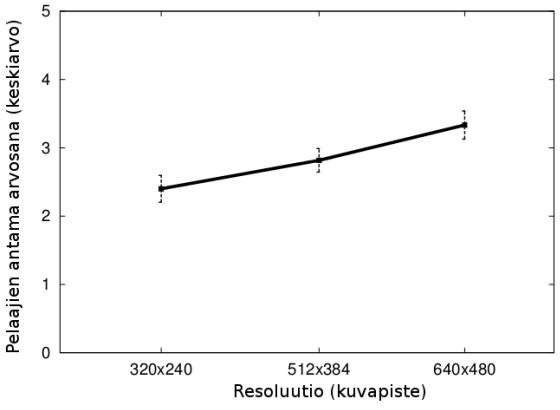



**(b)** Pelaajien saamat pisteet suhteutettuna resoluutioon. Päivitystiheys on 15 ruutua sekunnissa.

**Kuva 1:** Claypoolin tutkimusryhmän[17] havaintoja grafiikan laadun vaikutuksista pelaamiskykyyn. Suurempi pistemäärä kuvastaa parempaa pelimenestystä.




**(a)** Pelaajien antamat arvosanat suhteutettuna päivitystiheyteen. Resoluutio on 512x384 kuvapistettä.




**(b)** Pelaajien antamat arvosanat suhteutettuna resoluutioon. Päivitystiheys on 15 ruutua sekunnissa.

**Kuva 2:** Claypoolin tutkimusryhmän[17] saamia kyselytuloksia. Kyselyissä pyydettiin koehenkilöitä antamaan pelin visuaaliselle laadulle arvosana asteikolla 0-5.

Kuvan 1a mukaan piirtotaajudella on suuri vaikutus pelaajiin jotka pelaavat intensiivistä, nopeaa reagointikykyä vaativaa peliä. Claypoolin tutkimusryhmän löydökset[17]

osoittavat näin ollen yhden pelinkehittäjän konkreettisen tarpeen. Liian vähäinen piirtotaajuus voi pahimmassa tapauksessa tehdä pelistä pelaamattoman. 30 ruutua/s on kuvien perusteella se piste, jossa päivitystiheyden lisäämisen vaikutukset alkavat loiventua. Sitä alemmissa määrissä pelattavuus alkaa kuitenkin kärsiä nopeasti.

Kuvien 1a ja 2a perusteella voidaan lisäksi päätellä että päivitystiheydellä on suuri vaikutus pelaajan kokemuksiin pelistä erityisesti tiheyden ollessa matala. Toisaalla vertaamalla kuvaa 2b muihin voimme nähdä että päivitystiheyden noustessa yli 30 ruutuun sekunnissa resoluutiolla on suurempi vaikutus pelaajien mielipiteeseen pelin visuaalisesta laadusta[17].

Näin ollen pelin sunnittelijan kannattaa rajata pelinsä sisältö sellaiseksi että se pystytään esittämään tarvittavan sujuvassa muodossa. Tällöin joidenkin pelityyppien näyttävyys saattaa rajautua huomattavasti jos ne ovat ollenkaan toteutettavissa. Jos WebGL on toimiva työkalu, se lisäisi pelinkehittäjien ilmaisuvoimaa selaimessa pelattavilla peleillä. Paljon laskentatehoa vaativat pelityypit ovat tähän mennessä olleet pakostakin natiivisovelluksia: jos WebGL toimii hyvin, se mahdollistaisi helpomman ja halvemman kehitystyön tällaisille peleille.

# 5 WebGL-rajapinnan ominaisuuksia pelinkehittäjän näkökulmasta

## 5.1 Suorituskyky

Selkein WebGL-rajapinnan tarjoama etu on suorituskyky. WebGL-rajapinta sallii laskennan suorittamisen näytönohjaimessa suorittimen sijaan, mikä on huomattavasti nopeampaa. Kuvasta 3 nähdään että Canvas 2D Context ja Flash ovat vertailussa selvästi hitaampia. C++-pohjainen yksinkertainen OpenGL-sovellus on yllättäen WebGL-sovellusta hitaampi: natiivi toteutus täytyy optimoida ennen kuin se on tehokkaampi kuin WebGL-toteutus.[18] Ero ei suurimmaksi osaksi kuitenkaan johdu WebGL-koodin ja natiivin OpenGL-koodin välisistä eroista. Koska WebGL-rajapintaa kutsutaan verkkosivun koodista, selaimen on suoritettava JavaScript-koodia jokaista piirtoa varten. JavaScript-koodi on muutamia erikoistapauksia lukuunottamatta paljon hitaampaa kuin C++-koodi[19]. Täten suurin osa WebGL-koodin piirtämisajasta kuluu JavaScriptin kääntämiseen ja ajamiseen[18].

Pelinkehittäjän kannattaa siis ottaa huomioon, että WebGL on huomattavasti hitaampi kuin optimoitu natiivi sovellus, mikä oli odotettavissa. Huomionarvoisempaa on kuitenkin, että WebGL ei itse ole pääsyyllinen kyseiseen eroon: Taulukosta 1 ilmenee, että vaikka WebGL-koodin optimointi lyhensi piirtämisaikaa noin 80 %, kokonaisaika pieneni ainoastaan noin 23 %[1].

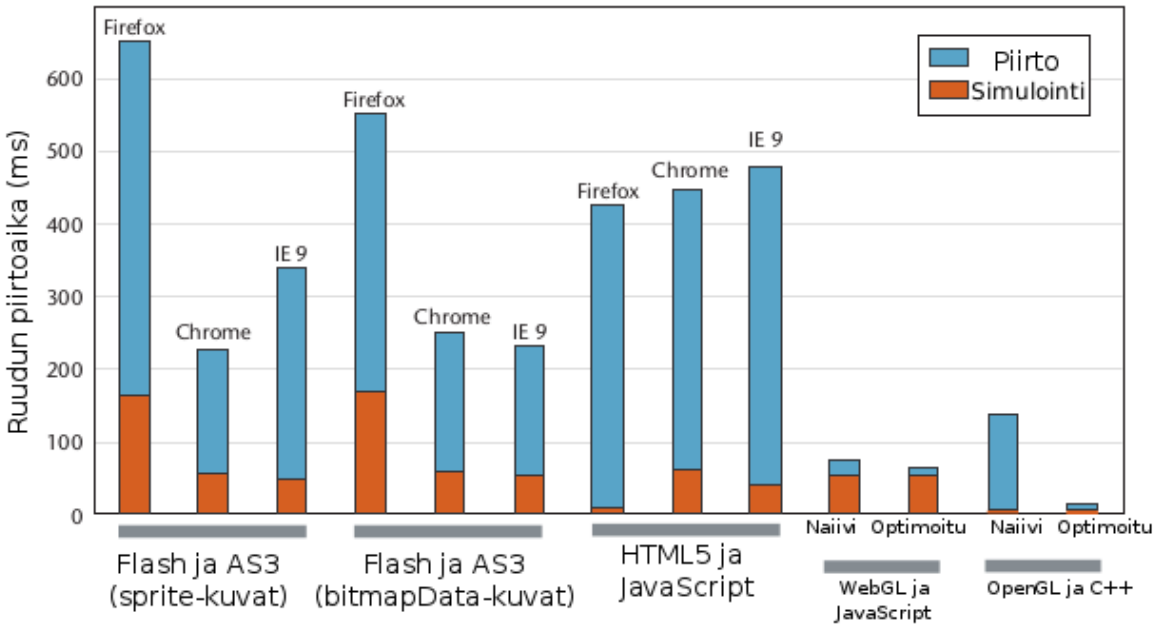
**Taulukko 1:** Hoetzleinin[18][20] mittaustulokset erilaisista toteutuksista piirtäessä 100000 objektia, paitsi viimeisessä sarakkeessa.

| **Implementaatio** | **Simulointi (ms)** | **Piirto (ms)** | **Kokonaisaika (ms)** | **30 Hz maksimi**[a] |
|---|---|---|---|---|
| Canvas 2D Context, Firefox | 9,7 | 417,0 | 426,7 | 7000 |
| Canvas 2D Context, Chrome | 59,3 | 391,0 | 450,3 | 4000 |
| Canvas 2D Context, IE 9 | 39,0 | 443,3 | 482,3 | 6000 |
| WebGL, naiivi, Chrome | 54,0 | 23,0 | 77,0 | 70000 |
| WebGL, optimoitu, Chrome | 54,7 | 4,3 | 59,0 | 80000 |
| OpenGL, naiivi | 3,3 | 137,9 | 141,2 | 20000 |
| OpenGL, optimoitu | 3,3 | 7,5 | 10,8 | 400000 |

[a]Suurin määrä objekteja joita voitiin piirtää ja myös yltää 30 ruudun/sekunti päivitystiheyteen.

Pelinkehittäjälle olennaista on huomata, että kuvan 3 ”WebGL ja JavaScript” -osion

[1]WebGL, sprites, Chrome: piirtämisaika 4,3 ms, kokonaisaika 59,0 ms. WebGL, VBOs, Chrome: piirtämisaika 23,0 ms, kokonaisaika 77,0 ms. 1 - (4,3 / 23,0) = **0,813**. 1 - (59 / 77) = **23,4**.




**Kuva 3:** Hoetzleinin[18] havainnot eri teknologioiden nopeuksista. Oranssit palkit kuvastavat simulointiaikaa, siis aikaa joka vaadittiin piirrettävien objektien uusien sijaintien laskennassa (Simulation). Siniset palkit kuvaavat itse piirtämiseen (Rendering) kulunutta aikaa. Matalammat palkit vastaavat nopeampaa laskentaa. Mittauksessa piirrettiin 100000 objektia.

kummassakin toteutuksessa WebGL-koodin osuus kuluneesta ajasta on paljon pienempi kuin JavaScript-koodin. Kehittäjä saattaa siis päästä paljon parempiin tuloksiin, jos hän keskittyy WebGL-koodin sijasta JavaScript-koodin optimointiin.

JavaScriptin toimintaa voi nopeuttaa niin sanotulla esikääntämisellä. Esikääntämisessä JavaScript-koodi muunnetaan niin että toiminnallisuus ei muutu, mutta koodin struktuuri on tehokkaampaa. Tunnetuin esikääntäjä lienee Google:n Closure-kääntäjä[21]. Myös muut parannukset ovat mahdollisia. Hoetzleinin[18] toteutuksessa esimerkiksi fysiikan laskenta suoritetaan JavaScript-koodissa. Tällaisen laskennan sirtäminen näytönohjaimeen saattaisi nopeuttaa piirtämistä huomattavasti.

Kuten kappaleessa 4 ja erityisesti kuvassa 1a esitettiin, on 30 ruudun/s päivitystiheys jonkinlainen minimi peleissä. Taulukon 1 ”30 Hz maksimi-sarakkeesta voidaan nähdä että WebGL-koodilla toteutettu sovellus pystyy piirtämään jopa yli kymmenen kertaa niin paljon esineitä kuin Canvas 2D Context-toteutukset jos pyritään 30 ruudun/s päivitystiheyteen.

## 5.2 WebGL-kirjaston piirteitä

Pelinkehittäjän on aina hyvä tietää, mihin valittu teknologia pystyy ja mitä puutteita sillä on. Di Benetto[22] on tunnistanut puutteita WebGL-kirjastosta.

### 5.2.1 Lataaminen

WebGL on matalan tason kirjasto joka oletusarvoisesti toimii sekä tietokoneilla että mobiililaitteilla. Tästä syystä siitä saattaa puuttua toiminallisuutta joka löytyy oletusarvona suuremmista kirjastoista. Yksi tärkeä ominaisuus nykyaikaisissa grafiikkakirjastoissa on asynkroninen lataaminen. Kuvaa piirrettäessä tietokoneen on haettava muistista erinäistä tietoa: tekstuureita, malleja ynnä muuta. Jos tällaiset tiedostot pitää ladata yksi kerrallaan ohjelma saattaa lakata reagoimasta käyttäjään kunnes kaikki tarvittava tieto on ladattu. Asynkroninen lataaminen sallii eri osien lataamisen samanaikaisesti, jolloin suoritin voi samalla reagoida käyttäjään. JavaScript salli tällaisen asynkronisuuden ainoastaan Web Workers rajapinnan kautta[23], joten ominaisuutta ei löydy oletuksena WebGL-kirjastoa käytettäessä.

### 5.2.2 HTML-elementtien rajapinnat

Yksi HTML5-kielen hyödyllisistä ominaisuuksista ovat elementti-kohtaiset rajapinnat. Esimerkiksi `<img>`-elemetin JavaScript-rajapinnasta löytyy onload-funktio, joka sallii kehittäjän määritellä mitä tehdään kun kyseisen elementin määrittelemä kuva on ladattu valmiiksi. JavaScriptissä tai WebGL:ssä ei ole tällaista toiminnallisuutta 3D-objekteille: HTML5 ei määrittele 3D-objekteille erityisiä elementtejä tai rajapintoja.

### 5.2.3 Eroavaisuudet täysimittaiseen kirjastoon

Tärkeä asia pitää mielessä on, että WebGL perustuu OpenGL ES-rajapintaan, joka vuorostaan sisältää vain osan täysimittaisen OpenGL-kirjaston toiminnallisuudesta (kuten OpenGL 4.0)[24]. Näin ollen kehittäjä joka on kokenut OpenGL-koodaaja ei välttämättä ole niin tehokas kuin voisi toivoa: WebGL-kirjaston rajallisuus OpenGL-kirjastoon verrattuna saattaa vaatia sopeutumista.

Kuitenkin perustyökalut muotojen piirtämiseen ovat rajapinnoissa samat. Taulukosta 2 nähdään, että melkein kaikki OpenGL-rajapinnan primitiivit löytyvät myös WebGL-rajapinnasta. Ainoa poikkeus on muoto (Patch) joka on tarkoitettu mosaiikkien luomiseen erilaisista primitiiveistä[25]. Näin ollen pelinkehittäjällä on käytössään samat perusprimittivit kummassakin rajapinnassa.

### 5.2.4 Laskentakirjastot

Korkeammalla tasolla WebGL-rajapinnassa on kuitenkin puutteita verratessa OpenGL-rajapintaan. 3D-grafiikan piirtämisessä käytetään huomattavia määriä erilaista matriisilaskentaa. Näitä funktioita ei löydy WebGL-rajapinnan kirjastosta, joten

**Taulukko 2:** OpenGL-[25] ja WebGL-rajapinnan[9, 5.14] primitiivit.

| **Muoto** | **OpenGL 4.0** | **WebGL 1.0** |
|---|---|---|
| Pisteitä | `GL_POINT` | `POINTS` |
| Viivoja | `GL_LINES` | `LINES` |
| Jatkuva viiva | `GL_LINE_STRIP` | `LINE_STRIP` |
| Jatkuva, sulkeutuva viiva | `GL_LINE_LOOP` | `LINE_LOOP` |
| Kolmioita | `GL_TRIANGLES` | `TRIANGLES` |
| Sarja kolmioita | `GL_TRIANGLE_STRIP` | `TRIANGLE_STRIP` |
| Kolmioita joilla yhteinen kulma | `GL_TRIANGLE_FAN` | `TRIANGLE_FAN` |
| Muoto (Patch) | `GL_PATCH` | Ei |

kehittäjän on joko toteutettava ne itse tai käytettävä kolmannen osapuolen kehittämää kirjastoa. Jos kehittäjän tarkoituksiin sopivaa, laadukasta kirjastoa ei ole tiedossa se merkitsee mahdollisesti huomattavaa riskiä kehityksessä.

## 5.3 WebGL-koodin Helppokäyttöisyys

Tässä osiossa helppokäyttöisyyttä katsastetaan vertailemalla WebGL-koodin monimutkaisuutta muihin toteutuksiin. Kirjaston helppokäyttöisyyttä voi myös katsastella esimerkiksi dokumentaation laadun ja koodin struktuurin näkökulmista. Tällainen tarkka katsastus ei kuitenkaan kuulu tämän työn puitteisiin, joten tässä keskitytään ainoastaan lähdekoodin monimutkaisuuden vertailuun.

Verrattavat toteutukset ovat erittäin tiiviisti kirjoitettu, joten niiden toiminnallisuus selitetään alustavasti koodin yhteydessä. Lähdekoodit ovat erinäisten toteutusten piirtosilmukoita, kaikki Hoetzleinin[18] tutkimuksesta. Piirtosilmukka yksinkertaisesti tarkoittaa sitä osaa koodista joka piirtää kuvan ruudulle. Ennen piirtosilmukoiden ajoa on jokaisessa toteutuksessa laskettu piirrettävien esineiden uudet sijainnit jollakin metodilla joka on kaikissa toteutuksissa hyvin samanlainen[18].

Koodi 1 kuvastaa Canvas 2D Context -rajapintaa hyödyntävää JavaScript-kielen toteutusta. Tämä on hyvin yksinkertainen piirtosilmukka joka ajetaan kokonaan suorittimessa. Vertaamalla tätä koodeihin 2 ja 3 nähdään että koodin määrä moninkertaistuu, jota voi pitää selvänä merkkinä monimutkaistumisesta.

**Koodi 1:** 2D Context-rajapintaa käyttävä, JavaScript-kielellä toteutettu piirtosilmukka[18]

```
1 for( var i = 0, j = particles.length; i < j; i++ )
2     context.drawImage ( ball_img, particles[i].posX, particles[i].posY );
```

**Koodi 2:** OpenGL-rajapintaa käyttävä, C++-kielellä toteutettu optimoitu[2]piirtosilmukka[18]

```
float* dat = bufdat;
for (int n=0; n < num_p; n++ ) {
   *dat++ = pos[n].x; *dat++ = pos[n].y;
   *dat++ = pos[n].x+32; *dat++ = pos[n].y;
   *dat++ = pos[n].x+32; *dat++ = pos[n].y+32;
   *dat++ = pos[n].x; *dat++ = pos[n].y+32;
}
glEnable ( GL_TEXTURE_2D );
glBindTexture ( GL_TEXTURE_2D, img.getGLID() );
glBindBufferARB ( GL_ARRAY_BUFFER_ARB, vbo );
glBufferDataARB ( GL_ARRAY_BUFFER_ARB, sizeof(float)*2*4*num_p, bufdat,
    GL_DYNAMIC_DRAW_ARB );
glEnableClientState ( GL_VERTEX_ARRAY );
glVertexPointer ( 2, GL_FLOAT, 0, 0 );
glBindBufferARB ( GL_ARRAY_BUFFER_ARB, vbotex );
glEnableClientState ( GL_TEXTURE_COORD_ARRAY );
glTexCoordPointer(2, GL_FLOAT, 0, 0 );
glDrawArrays ( GL_QUADS, 0, num_p );
```

**Koodi 3:** WebGL-rajapintaa käyttävä, JavaScript-kielellä toteutettu optimoitu[3]piirtosilmukka[26]

```
for( var i = 0, j=0, k=0; i < num_particles; i++ ) {
    vertices[j++] = particles[i].posX;      vertices[j++] = particles[i].posY;
}
gl.bindBuffer(gl.ARRAY_BUFFER, geomVB);
gl.bufferData(gl.ARRAY_BUFFER, vertices, gl.DYNAMIC_DRAW);
gl.viewport(0, 0, gl.viewportWidth, gl.viewportHeight);
gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
mat4.identity(pMatrix);
mat4.scale ( pMatrix, [2.0/SCREEN_WIDTH, -2.0/SCREEN_HEIGHT, 1] );
mat4.translate ( pMatrix, [-(SCREEN_WIDTH)/2.0, -(SCREEN_HEIGHT)/2.0, 0] );
mat4.identity(mvMatrix);
gl.bindBuffer(gl.ARRAY_BUFFER, geomVB);
gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute, geomVB.itemSize,
    gl.FLOAT, false, 0, 0);
gl.bindBuffer(gl.ARRAY_BUFFER, geomTB);
gl.vertexAttribPointer(shaderProgram.textureCoordAttribute, geomTB.itemSize, gl.FLOAT,
    false, 0, 0);
gl.activeTexture(gl.TEXTURE0);
gl.bindTexture(gl.TEXTURE_2D, neheTexture);
gl.uniform1i(shaderProgram.samplerUniform, 0);
setMatrixUniforms();
gl.drawArrays(gl.POINTS, 0, num_particles );
```

Toisin kuin Canvas 2D Context-rajapinnan tapauksessa laskentaa ei suoriteta samassa

[2]Sen sijaan että piirtokutsu lähetettäisiin jokaiselle objektille erikseen, Hoetzlein siirtää kaikki uudet sijainnit yhdellä kertaa näytönohjaimen muistiin. Näin muistiväylän käyttö on paljon tehokkaampaa ja nopeampaa.[18]

[3]Ei-optimoidussa WebGL-koodissa Hoetzlein joutuu eksplisiittisesti kertomaan jokaisen objektin kulmat. Tässä koodissa Hoetzlein käyttää WebGL point sprite-objekteja.[18] Tällöin näytönohjaimelle ei tarvitse siirtää yhtä monta päivitettyä pistetietoa.

muistissa. Kaikkien piirrettävien objektien uudet sijainnit on siirrettävä näytönohjaimen muistiin. OpenGL- ja WebGL-rajapintojen monimutkaisemmassa koodissa täytyy ensin sitoa objektien kuvat laskettuihin uusiin koordinaatteihin, tämä tieto on siirrettävä näytönohjaimelle ja lopulta näytönohjainta on pyydettävä piirtämään kuva.

Canvas 2D Context-rajapinta on helpompi käyttää. Jokaista objektia kohden kerrotaan objektin uusi sijainti sekä kuva joka kyseiseen sijaintiin tulisi piirtää. Tietoa ei tarvitse siirtää muistien välillä, sillä se löytyy valmiiksi samasta välimuistista mistä koodi ajetaan. Näin Canvas 2D Context-rajapinnan koodi pysyy paljon yksinkertaisempana.

Koodi 2 ja Koodi 3 eivät kuitenkaan eroa toisistaan yhtä paljon kuin koodista 1: C++-koodin rivimäärä ei eroa WebGL-koodin rivimäärästä yhtä dramaattisesti kuin JavaScript-koodin rivimäärästä. OpenGL- ja WebGL-rajapinnat tosin eroavat toisistaan kattavuuden suhteen, kuten kappaleessa 5.2 eriteltiin. Kyseiset eroavaisuudet kuitenkin koskivat pääasiassa sellaisia henkilöitä joille jompi kumpi rajapinta oli ennestään tuttu. Lähdekoodin tasolla voidaan olettaa eroja olevan vähemmän, joten kummatkin rajapinnat ovat oletettavasti yhtä helppoja käyttää. Näin ollen pelinkehittäjän ei luultavasti tarvitse huolehtia itse koodin monimutkaistumisesta OpenGL-rajapintaan verrattuna.

## 5.4 Alustariippumattomuus

Web-sovellusten suurin etu on alustariippumattomuus: saman sovelluksen pitäisi toimia kaikilla alustoilla jotka tukevat selainta, jolla sovellus toimii. Teoriassa kehittäjän tarvitsee ainoastaan varmistaa pelin toimivan niillä selaimilla joilla on surimmat käyttäjämäärät saadakseen pelilleen mahdollisimman suuren potentiaalisen yleisön. Parhaassa tapauksessa ainoastaan pieniä osia koodista tarvitsee muuttaa, jotta koodi saadaan toimimaan eri selaimissa. Natiivit sovellukset taas saattavat pahimmillaan vaatia täysin eri kielet eri alustoja varten, jolloin samasta toteutuksesta voi olla useita ylläpidettäviä kopioita. Tällöin selaimessa toimiva sovellus olisi huomattavasti helpompi ylläpitää.

Käytännössä selainten välillä saattaa kuitenkin olla suuriakin eroja. Varsinkin JavaScriptin kääntönopeudet vaihtelevat huomattavasti käytetyimpien selainten välillä[18]. Suurempi ongelma on kuitenkin WebGL-rajapinnan toteutusten erot: jotkin piirto-ominaisuudet saattavat toimia aivan eri lailla riippuen selaimesta[27]. Näin ollen kehittäjän ei välttämättä ole helppoa saada peliään toimimaan kaikilla selaimilla: jotkin piirtoratkaisut saattaa olla pakollista muuttaa perustavanlaatuisesti jotta tuote saadaan toimimaan toisella selaimella.

Suurin ongelma WebGL-sovellusten alustariippumattomuudelle on kuitenkin laitteistotuki. Koska WebGL-rajapinta toimii laitteen näytönohjaimen kanssa, ovat mahdolliset selain-laite yhdistelmät hyvin moninaiset. Näin ollen selainten ja

laitteiston yhteensopivuudet eivät ole taattuja, vaan laitteen toimivuus riippuu käytetystä selaimesta[28][29]. Kehittäjä ei näin ollen voi olla varma, että kaikki selaimen käyttäjät voivat pelata WebGL-pohjaisia pelejä.

Toisaalta on pidettävä mielessä, että selainvalmistajat pyrkivät tukemaan suurinta määrää käyttäjiä. Varsinkin jos WebGL:stä tulee suosittu teknologia voidaan olettaa, että selainvalmistajat varmistavat tuen ainakin kaikkein suosituimmille näytönohjainmalleille. Kehittäjä ei siis saa peliään toimimaan kaikkien potentiaalisten käyttäjien koneilla, mutta ero ei luultavasti ole dramaattinen.

# 6 Yhteenveto

Olemassaoleva kirjallisuus ei ole erityisen kattavaa, mutta materiaalista on mahdollista johtaa pelinkehittäjälle suuntaa-antavia johtopäätöksiä.

Ensinnäkin pelinkehittäjän on huomioitava WebGL-rajapinnan suorituskykyyn liittyvät ominaispiirteet. Huomionarvoista on että WebGL-koodin suorituskyky ei vaikuta lopputulokseen yhtä vahvasti kuin se JavaScript-koodi josta WebGL-koodia ajetaan, joten suorituskykyä parantaessa kehittäjän kannattaa ehdottomasti ensin keskittyä JavaScript-koodin nopeuttamiseen.

Toiseksi WebGL-kirjastossa saattaa olla puutteita: rajapinta on tarkoitettu toimimaan niin päätelaitteilla kuin mobiililaitteilla, joten sen on tarkoitus olla kevyt ja monikäyttöinen. Kehittäjän täytyy varmistaa että osaaminen riittää tai käyttää jotakin kehitystä tukevaa, WebGL-kirjaston päälle kehitettyä kirjastoa.

Kolmanneksi WebGL ei ole puhtaasti alusta-riippumaton: erot suorituskyvyssä ja lopputuloksissa sekä rajoitteet tuetuissa näytönohjaimissa tarkoittavat, että kehittäjä ei voi olettaa pelin toimivan yhtä laajalti kuin esimerkiksi Canvas 2D-Context-rajapintaan pohjautuva peli. Kuitenkin koodin ylläpidettävyydessä saavutetut hyödyt voivat olla mittavia.

Yleisvaikutelma WebGL-rajapinnasta kielii keskeneräisyyttä. Rajapinta on verrattain nuori, eivätkä selainvalmistajat ole kaikesta päätellen saaneet toteutuksiaan toimimaan täysin yhteneväisesti. Tulevaisuudessa nämä eroavaisuudet saattavat kadota selainvalmistajien parantaessa toteutuksiaan. Pelinkehittäjän kannalta on oleellisinta on että WebGL on muiden toteutuksien välimaastossa. WebGL-koodi on huomattavasti monimutkaisempaa kuin Canvas 2D-Context-rajapintaa käyttävä koodi, mutta pystyy paljon tehokkaampaan laskentaan. Kuitenkaan se ei yllä optimoidun, natiivin sovelluksen tasolle. WebGL-rajapinta on huomattavasti alustariippumattomampi kuin natiivi sovellus, mutta ei kuitenkaan ole yhtä universaali kuin yksinkertaisemmat sovellukset.

WebGL-rajapinta soveltunee parhaiten juuri peleihin jotka jäävät näiden kahden ääripään väliin. Pelit jotka vaativat sujuvaa piirtoa johon Canvas 2D Context-rajapinta ei yksinkertaisesti riitä, mutta jotka eivät kuitenkaan vaadi keskivertoa tehokkaampia tietokoneita. WebGL tarjoaa parempaa alustariippumattomuutta kuin natiivi OpenGL-koodi, joten WebGL-rajapinta soveltuu pienemmille kehittäjäryhmille joilla ei ole resursseja työstää montaa eri versiota samasta pelistä.

WebGL-rajapinnan ominaisuuksista ei tätä työtä varten löytynyt erityisen paljon aiheen kannalta relevanttia kirjallisuutta. Tulevaisuudessa WebGL-rajapinnan tutkimukset voisivat keskittyä esimerkiksi muistin käyttäytymiseen ja laskennan jakoon

näytönohjaimen ja JavaScript-koodin välillä. Esimerkiksi voitaisiin tutkia kuinka paljon JavaScript-koodin esikääntäminen tai monien erilaisten kuvien piirtäminen vaikuttaa surituskykyyn.

# Lähteet


[1] W3C and WHATWG. HTML5 - A vocabulary and associated APIs for HTML and XHTML. Spesifikaatio, 17.12.2012 2012. Saatavissa http://www.w3.org/TR/2012/CR-html5-20121217/. Viitattu 21.02.2014.

[2] W3C ja WHATWG. HTML5 differences from HTML4: Introduction. Verkkosivu. Saatavissa http://www.w3.org/TR/2011/WD-html5-diff-20110405/#introduction. Viitattu 15.02.2014.

[3] ECMA International. ECMA-262 ECMAScript Language Definition. Spesifikaatio. Saatavissa http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-262.pdf. Viitattu 11.03.2014.

[4] Mozilla Developer Network. MDN - JavaScript. Verkkosivu. Saatavissa https://developer.mozilla.org/en-US/docs/Web/JavaScript. Viitattu 11.03.2014.

[5] WHATWG. HTML - 4.12.4 - The canvas element. Spesifikaatio. Saatavissa http://www.whatwg.org/specs/web-apps/current-work/multipage/the-canvas-element.html. Viitattu 21.03.2014.

[6] Khronos Group. The OpenGL Graphics System: A Specification, 2011. Saatavissa http://www.opengl.org/registry/doc/glspec40.core.20100311.pdf. Viitattu 28.02.2014.

[7] Microsoft. DirectX Graphics and Gaming. Verkkosivu. Saatavissa http://msdn.microsoft.com/en-us/library/windows/desktop/ee663274. Viitattu 28.02.2014.

[8] Khronos Group. OpenGL ES. Saatavissa http://www.khronos.org/opengles/. Viitattu 28.02.2014.

[9] Khronos Group. WebGL Specification. Verkkosivu, 2013. Saatavissa http://www.khronos.org/registry/webgl/specs/latest/1.0/. Viitattu 15.02.2014.

[10] Jesus Jimenez, Jaime Cruz ja Juan Ruiz de Miras. High performance 3D visualization on the Web: a biomedical case study. *International Work-Conference on Bioinformatics and Biomedical Engineering (IWBBIO)*, sivut 465–471, 2013. Saatavissa http://iwbbio.ugr.es/papers/iwbbio_078.pdf.

[11] Mobeen Movania Mobeen ja Lin Feng. High-Performance Volume Rendering on the Ubiquitous WebGL Platform. *2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS)*, sivut 381–388, 2012. Saatavissa `http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6332197`. DOI: 10.1109/HPCC.2012.58.

[12] +360°. Car Visualizer. Tuotedemo. Saatavissa `http://carvisualizer.plus360degrees.com/threejs/`. Viitattu 10.03.2014.

[13] Bentar Priyopradono ja Fajar A. Perdana. Web3D Publishing Interior Design and Residential Collection based on WebGL Technology. *International Journal of Computer Applications*, 64(1), 2013. Saatavissa `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.278.5819&rep=rep1&type=pdf`.

[14] WebGL.com. WebGL - Games. Saatavissa `http://www.webgl.com/category/webgl-games/`. Viitattu 07.03.2014.

[15] Stefan Haustein, Joel Webber, Matthias Büchner, Ray Cromwell ja moogle (käyttäjänimi). quake2-gwt-port. Verkkopeli. Saatavissa `http://quake2playn.appspot.com/`. Viitattu 10.03.2014.

[16] WebGL.com. WebGL Game Demo - Unreal Engine 3 (Epic Citadel). Verkkosivu. Saatavissa `http://www.webgl.com/2013/05/webgl-game-demo-unreal-engine-3-epic-citadel/`. Viitattu 10.03.2014.

[17] Mark Claypool, Kajal Claypool ja Feissal Damaa. The effects of frame rate and resolution on users playing First Person Shooter games. *Electronic Imaging 2006*. International Society for Optics and Photonics, 2006. Saatavissa `http://web.cs.wpi.edu/~claypool/papers/fr-rez/paper.pdf`. DOI: 10.1117/12.648609.

[18] Rama C. Hoetzlein. Graphics Performance in Rich Internet Applications. *Computer Graphics and Applications*, IEEE, 32(5):98–104, 2012. Saatavissa IEEE Explore, DOI: 10.1109/MCG.2012.102.

[19] Master's thesis. Fredrik Smedberg. Performance analysis of JavaScript. Pro gradu, Linköpings Universitet, 2010. Saatavissa `http://www.diva-portal.org/smash/get/diva2:321661/FULLTEXT01.pdf`. Viitattu 21.02.2014.

[20] Rama C. Hoetzlein. Sprite Testing for Rich Internet Applications - Raw Data. Mittaustuloksia Excel-taulukossa. Saatavilla http://www.rchoetzlein.com/theory/wp-content/uploads/2012/05/sprite_performance.xls. Viitattu 04.04.2014.

[21] Google Developers. Closure Tools - Compiler. Kehitystyökalu. Saatavissa https://developers.google.com/closure/compiler/. Viitattu 11.03.2014.

[22] Marco Di Benedetto, Federico Ponchio, Fabio Ganovelli ja Roberto Scopigno. SpiderGL: a JavaScript 3D graphics library for next-generation WWW. *Proceedings of the 15th International Conference on Web 3D Technology*, sivut 165–174. ACM, 2010. Saatavissa ACM Digital Library, DOI: 10.1145/1836049.1836075.

[23] WHATWG. HTML: 10 - Web workers, 21.02.2014. Saatavissa http://www.whatwg.org/specs/web-apps/current-work/multipage/workers.html. Viitattu 21.02.2014.

[24] Khronos Group. WebGL and OpenGL Differences. Verkkosivu. Saatavissa http://www.khronos.org/webgl/wiki_1_15/index.php/WebGL_and_OpenGL_Differences. Viitattu 21.02.2014.

[25] Khronos Group. OPenGL.org - Primitive. Wiki. Saatavissa https://www.opengl.org/wiki/Primitive. Viitattu 21.03.2014.

[26] Rama C. Hoetzlein. Sprite Testing for Rich Internet Applications - particles_sprite.html. HTML/JavaScript-koodi. ZIP-arkisto saatavissa http://www.rchoetzlein.com/theory/wp-content/uploads/2012/05/sprites_webgl.zip. Viitattu 09.04.2014.

[27] Master's thesis. Jari-Pekka Voutilainen. Vuorovaikutteisten web-sovellusten kehittäminen. Pro gradu, Tampereen Teknillinen Yliopisto, 2011. Saatavissa http://dspace.cc.tut.fi/dpub/bitstream/handle/123456789/20804/voutilainen.pdf?sequence=3.

[28] Mozilla.org. Blocklisting/Blocked Graphics Drivers. Yhteensopivuuslistaus. Saatavissa https://wiki.mozilla.org/Blocklisting/Blocked_Graphics_Drivers. Viitattu 18.02.2014.

[29] Khronos Group. BlacklistsAndWhitelists. Yhteensopivuuslistaus, 2014. Saatavissa http://www.khronos.org/webgl/wiki/BlacklistsAndWhitelists. Viitattu 15.02.2014.

# A Sammandrag av kandidatarbetet på svenska

Inom spelindustrin har spelgrafik spelat en stor roll: kraftigare datorer har möjliggjort användningen av mera detalj och komplexitet inom spelvärldar. Under de senaste åren har nya web-teknologier dessutom gjort det lätt att producera spel för ett stort antal konsumenter: i princip vem som helst som har en modern webbläsare. Dessa två spår av teknologisk framgång har dock svårt att mötas. Komplex grafik måste ritas med kraftiga datorkomponenter som webbläsare inte förr har kunnat utnyttja utan utomstående program. Den nya standarden WebGL bör möjliggjöra ritandet av komplicerad grafik utan att användaren behöver installera nya program. Speldesigners kunde i bästa fall göra grafiskt avancerade spel som är tillgängliga åt en stor publik utan att behöva göra enskilda versioner åt olika platformer och operativsystem. Detta arbete undersöker med hjälp av en litteraturöversikt WebGL från speldesigners perspektiv: huruvida teknologin är tillräckligt kraftig och hurudana för- och nackdelar dess användning innebär. Dessutom dras det slutsatser om hurudana projekt WebGL kunde vara passligt för, samt insikter om vilka saker det lönar sig att beakta då man använder programbibliotket. Teknologin som oftast används för att rita spelgrafik på internet är HTML5 canvas-elementets Canvas 2D Context API. Detta API låter en programmerare rita bilder på en canvas-ruta som i sin tur placeras på en webbsida. Allt räknande som krävs för att producera bilder med Canvas 2D Context görs i datorns centralprocessor. Detta fungerar bra så länge bilderna som ska ritas inte är för detaljerade. I så fall räcker inte centralprocessorns räknekapacitet, varmed datorn kan bli långsam och oresponsiv. En grafikprocessor är en datorkomponent som är speciellt gjord för grafisk räkning. Svårigheten med att använda grafiska processorer är att de ofta kräver så kallad intern kod. Intern kod är kod som är skriven speciellt för den maskinuppsättningen koden utförs på. Därmed krävs det olika kod för olika datorer och operativsystem. WebGL är liksom Canvas 2D Context ett API för canvas-elementet. Det tillåter en programmerare att rita bilder på en webbsida. WebGL skiljer sig dock från 2D Context i och med att man med hjälp av WebGL-kod kan använda datorns grafikprocessor för att utföra räkningarna som krävs för ritandet. Både WebGL och Canvas 2D Context används med hjälp av programmeringsspråket JavaScript. JavaScript används i webbläsare för att programmera webbsidor. Språket är mycket populärt och program som skrivs med det fungerar i princip i alla moderna webbläsare. För att kunna åstadkomma en sådan flexibilitet måste det dock göras kompromisser. JavaScript kod är i nästan alla fall mycket långsammare än intern kod som C++, vilket är det programmeringsspråket med vilket i princip alla grafiskt krävande spel är skrivna. Kraven spel ställer på mjukvara är något unika. I undersökningar har det påvisats att bildens resolution och i mindre grad uppdateringstakten starkt påverkar människors åsikt angående kvaliteten av video, där man med uppdateringstakten menar hastigheten med vilken en ny bild ritas på rutan. I spel kan man utöver spelarens

åsikt också mäta hur framgångsrik spelaren är. I undersökningar har man funnit att speciellt uppdateringstakten kan ha en dramatisk inverkan på spelförmåga då spelet kräver snabba reaktioner. Spelarnas åsikt om spelets kvalitet stiger dessutom i takt med rutans resolution. Spel som ska se bra ut och kräver snabba reaktioner kräver därmed mycket processorkraft för att fungera. En jämförelse av olika implementationer visar att WebGL är snabb, dock inte snabbast. Jämfört med Canvas 2D Context är WebGL mångfaldigt snabbare: ritande med Canvas 2D Context tar väl över 5 gånger den tid som med WebGL i samma webbläsare. Ändå tar ritande med WebGL i sin tur över 5 gånger den tid som intern C++-kod kräver. Ett intressant fynd är dock att det inte nödvändigtvis är WebGL API:et som står för största delen av skillnaden. Mätningar visar att framställningen av bilden (som utförs i grafiksprocessorn) i själva verket kan vara snabbare i WebGL än i intern kod. Den stora skillnaden inträffar i simulationen, det vill säga då datorn räknar ut var alla objekt som ska ritas befinner sig. I intern kod görs beräkningen i C++-kod, medan den i WebGL görs i JavaScript-kod. Skillnaden i hastighet mellan dessa språk leder därmed till den stora skillnaden i total ritningstid. Det är dock viktigt att inse att kodens snabbhet inte är den ända faktorn i hur bra ett grafiskt program fungerar. En viktig aspekt är bildkomponenternas laddande från minne. Då grafikprocessorn ritar en bild på rutan måste den veta vilka färger som ska ritas i vilka bildpunkter. Detta görs med hjälp av texturer och 3D-modeller som måste laddas från minne. WebGL kan inte ladda många olika saker på samma gång så som många andra programbibliotek. Ifall en scen som ska ritas är invecklad kan det behövas många olika saker från minne, vilka i WebGL måste laddas enskilt. Detta gör bildens uppdatering långsamt. Designern måste i sådana fall använda sig av något utomstående programbibliotek. När 3D-objekt väl är laddade kan dock designern inte direkt manipulera dem så som bild- och tet-objekt. HTML5 har API:n för de flesta media-element, så som bilder och ljud, som kan manipuleras direkt från JavaScript-kod. 3D-objekt har dock inget sådant API, och designern måste därmed manipulera dem via andra metoder. Manipulation av 3D-objekt görs via matematiska transformationer, oftast med hjälp av linjär algebra. Mera etablerade 3D-programbibliotek har vanligen inbyggda bibliotek av sådana matematiska funktioner. WebGL har dock inte sådana inbyggda funktioner, utan de måste antingen hittas i något annat bibliotek eller skrivas av designern. Olikheterna mellan WebGL och mera etablerade programbibliotek beror på att WebGL baserar sig på OpenGL ES (Open Graphics Library Embedded Systems). OpenGL ES är en förminskad version av ett etablerat programbibliotek, OpenGL, där asynkronisk laddning och linjär algebra är inbyggda. OpenGL ES är menat för att kunna användas i vad som hälst för datorsystem, även pekplattor och mobiler. Därmed är den simplifierad jämfört med det fullt utrustade OpenGL-programbiblioteket. Meningen är att WebGL, liksom OpenGL ES, ska kunna användas i vilken slags dator som helst. På kodnivå är WebGL dock inte värst mycket mera komplicerat än intern kod: en optimerad ritningsloop i WebGL är

23 rader medan samma i intern kod är 17. Den klara skillnaden ligger mellan WebGL och Canvas 2D Context. Att göra en ritningsloop för Canvas 2D Context API:et kräver 2 rader kod. Man kan därmed definitivt säga att koden blir betydligt mera invecklad då man byter från Canvas 2D Context till WebGL. Den stora teoretiska fördelen med WebGL är dess universalitet. Om designers behöver endast en kopia av spelkoden, istället för en för varje platform, blir spelproduktion mycket lättare och billigare. Dessvärre är WebGL i praktik inte lika universell som Canvas 2D Context. 2D Context fungerar med säkerhet där webbläsaren fungerar: om inte den ena fungerar så fungerar inte den andra. Men WebGL behöver grafikprocessorn för att fungera, medan webbläsaren behöver endast centralprocessorn. Därmed finns det många grafikprocessorer som inte fungerar tillsammans med vissa webbläsare. Designern kan alltså inte förvänta sig att ett WebGL-program skulle fungera lika vida som Canvas 2D Context-program. Det viktigaste sakerna speldesigners borde beakta gällande WebGL är valet av programbibliotek och hur koden skall optimeras. Valet av programbibliotek ger designers möjligheten att använda sådana funktionligheter som inte är inbyggda i WebGL, såsom asynkroniskt laddande. Därmed kan WebGL användas som ett mera komplett grafikprogrambibliotek. Det är också viktigt att koncentrera sig på JavaScript-koden då man optimerar spelprogrammet. På grund av JavaScript-kodens långsamhet kan det till exempel löna sig att förflytta så många räkningar som möjligt till grafikprocessorn, eftersom till exempel fysikaliska räkningar kan vara mycket snabbare att lösa där. WebGL ligger emellan Canvas 2D Context och intern kod. I fråga om snabbhet är WebGL klart bättre än 2D Context, men når inte nivån av intern kod. Å andra sidan är WebGL-kod mycket mera universellt funktionlig än intern kod, men är ändå inte på 2D Contexts nivå. Projekt där det lönar sig att använda WebGL bör därmed vara någonslags hybrider av dessa två ändor av spektret: inte så pass grafiskt krävande att det behövs kraftiga datorer för att rita spelvärlden, men ändå för invecklade för Canvas 2D Context att klara av. Man ska heller inte förvänta sig att spelet ska fungera på precis alla webbläsare och platformer, men man kan lita på att spelet kommer att fungera på flera platformer än om spelet gjordes med en kopia av intern kod. Det är viktigt att inte tro att WebGL är som internt kodade grafikprogrambibliotek som råkar fungera var som helst. WebGL är passlig för en ny nisch emellan små, simpla webbspel och stora, invecklade internt kodade spel. Speldesigners bör tänka på om deras projekt passar in i en sådan nisch.