Selainpohjaiset pelit

Panu Hokkanen

Tampereen yliopisto
Luonnontieteiden tiedekunta
Tietojenkäsittelytieteiden tutkinto-ohjelma
Pro gradu -tutkielma
Ohjaaja: Timo Poranen
Heinäkuu 2017

Tampereen yliopisto
Luonnontieteiden tiedekunta
Tietojenkäsittelytieteiden tutkinto-ohjelma
Panu Hokkanen: Selainpohjaiset pelit
Pro gradu -tutkielma, 51 sivua
Heinäkuu 2017

Ohjelmistokehitys on siirtymässä kohti web-painotteista kehitystä ja työpöytäsovelluksia tehdäänkin koko ajan vähemmän. Selainpohjaiset pelit taas ovat murrosvaiheessa, sillä vanhat teknologiat ovat poistumassa käytöstä. Erilaisia korvaavia teknologioita on kehitteillä, mutta näillä ei ole vielä toteutettu suurehkoja selainpohjaisia pelejä.

Tutkielmassa perehdytään peleihin ja pelinkehitykseen sekä pohditaan, miten ohjelmistokehitys ja pelinkehitys eroavat toisistaan. Selainpohjaisten peliteknologioiden menneisyys ja nykytila ovat myös tarkastelussa. Tällä hetkellä JavaScript-ohjelmointikieli on varteenotettava vaihtoehto tulevien selainpelien pääasialliseksi kehityskieleksi. WebGL-teknologia taas mahdollistaa 3d-pohjaisten pelien suorittamisen selaimessa.

Tutkielmassa toteutettiin selainpohjainen peli case-esimerkkinä. Selainpohjainen pelinkehitys havaittiin kuitenkin haastavaksi prosessiksi. Ehdotettu teknologia, JavaScript, sopii ohjelmointikielenä hyvin pelinkehitykseen. Ongelmia muodostuu kuitenkin muiden selainpohjaisten teknologioiden johdosta, sillä 3d-selainteknologiat kärsivät muun muassa suorituskykyongelmista ja dokumentaation puutteellisuudesta.

Avainsanat ja -sanonnat: selainpohjainen peli, JavaScript, pelimoottori, pelinkehitys, peli, WebGL

Sisällysluettelo

1 Johdanto **1**

2 Selainpohjainen peli **4**

2.1 Pelin määritelmä *4*

2.2 Pelinkehitysprosessi *4*

2.3 Esimerkkejä selainpohjaisista peleistä *8*

3 Selainpohjaisten peliteknologioiden nykytila **11**

3.1 Selaimet ja esityskerros *11*

3.2 Palvelinpuoli *12*

3.3 Java *13*

3.4 Flash *14*

3.5 Unity ja Unity Web Player *14*

4 JavaScript selainpohjaisissa peleissä **16**

4.1 JavaScript kielenä *16*

4.2 JavaScript pelinkehityksessä *20*

4.3 JavaScript 3d-pelimoottorit *21*

4.4 Hyödyt ja haitat *23*

5 Case-esimerkki selainpohjaisen pelin toteutuksesta **25**

5.1 Pelin esittely ja vaatimusmäärittely *25*

5.2 Tekninen toteutus *27*

5.3 Toteutuksen arviointi *37*

6 Yhteenveto **46**

Viiteluettelo **49**

1 Johdanto

Työpöytäsovelluksia tehdään yhä vähemmän ja useat ohjelmistoyritykset ovat siirtymässä selainpohjaisiin sovelluksiin. Selaimia varten tehdyillä sovelluksilla on monia hyviä puolia sekä käyttäjille että ohjelmistokehittäjille. Ohjelmistoja ei tarvitse kääntää erikseen eri laiteympäristöille, vaan kehittäjät voivat luottaa siihen, että selainpohjainen sovellus toimii esimerkiksi pöytätietokoneen, tabletin ja kännykän selaimissa. Käyttöliittymän luominen on myös helpompaa hyödyntämällä selaimen tarjoamia yksinkertaisia teknologioita, kuten HTML-kuvauskieltä ja CSS-tyylimäärittelyjä. Web-sovelluksien suorituskyky on myös kasvanut vuosien saatossa hyväksyttävälle tasolle, erityisesti JavaScript-kielen edistysaskelten johdosta.

Selainpohjaisia pelejä on ollut olemassa selaimien syntyajasta lähtien. Alun perin yksinkertaisista, pääasiassa tekstipohjaisista selainpeleistä on edistytty huimasti. Teknologiat, kuten Flash ja Javan sovelmat, mahdollistivat jo selaimien alkuaikoina korkeatasoisen grafiikan käytön selaimissa. Nykyisin kyseiset teknologiat on korvannut JavaScript-pohjainen teknologia, WebGL, joka mahdollistaa esimerkiksi 3d-grafiikan ajamisen selaimessa ilman mitään lisäosia. JavaScriptillä toteutettuja laajoja selainpohjaisia pelejä on kuitenkin olemassa erittäin niukasti. Selainpohjaisten pelien hyviä puolia ovat muun muassa laitteistoriippumattomuus ja helppo lähestyttävyys, sillä pelejä ei tarvitse asentaa isäntälaitteelle, vaan ne suoritetaan selainikkunassa. Silti peliyhtiöt luovat mieluummin pelejään suoraan työpöydälle, mobiiliin tai tableteille, hyödyntämättä selainpohjaisuutta. Tällöin pelit joudutaan kääntämään jokaiselle alustalle erikseen ja mahdollinen työn määrä kasvaa entisestään.

Tutkimuksen tavoitteena on tehdä katsaus selainpohjaisten pelien nykyisiin toteutustekniikoihin ja tutkia, mitä haasteita selainpohjaisen pelin toteuttamisessa on. Tavoitteena on myös tutkia selainpohjaisten pelien kehitysprosessia ja laatua.

Teoreettisena viitekehyksenä käytetään suunnittelutiedettä (design science) [Hevner et al., 2004]. Suunnittelutiede määrittelee useita suuntaviivoja, joita tutkimuksen tulisi noudattaa. Tutkimuksen tulisi tuottaa artefakti jonkin käsitteen, mallin, metodin tai toteutuksen osalta (design as an artefact). Tässä tutkimuksessa artefakti konkretisoituu selainpohjaisella teknologialla toteutettuna pelinä. Selainpohjaisuus itsessään on relevantti tutkimuksen kohde pelien osalta, sillä varteenotettavat pelit ovat häviämässä selaimista (problem relevance). Toteutettua artefaktia eli peliä arvioidaan erilaisten ohjelmisto- ja peliprojektien laadun kriteerien pohjalta (design evaluation) ja täten artefakti itsessään edistää selkeästi ja todennettavasti tutkimusta (research contributions). Tutkimuksessa toteutettavan artefaktin hyödyllisyyden, laadun ja tehokkuuden arvioinnissa voidaan käyttää viittä erityyppistä menetelmää. Tässä

tutkimuksessa hyödynnetään lähinnä tarkkailumenetelmää (observational), jossa pelin toimintaa tarkkaillaan aidossa ympäristössä.

Suunnittelutieteessä yritetään luoda asioita, jotka hyödyttävät ihmisiä jollain käytännöllisellä tasolla, kun taas tavanomaisissa luonnontieteissä tutkimuksen pääasiallinen tarkoitus on ymmärtää todellisuutta tarkemmin. March ja Smith [1995] ehdottavat, että tietojärjestelmiä koskevissa tutkimuksissa on hyödynnettävä sekä luonnontieteellisiä että suunnittelutieteellisiä menetelmiä. He myös esittelevät kaksiulotteisen kehyksen, jonka avulla on mahdollista arvioida ratkaisujen pätevyyttä. Taulukko 1 sisältää kyseisen tutkimuskehyksen, jonka pystyakselilla ovat suunnittelutieteellisen tutkimuksen artefaktit kuten käsitteet, mallit, metodit ja toteutukset. Vaaka-akselilla taas ovat tutkimusaktiviteetit. Näistä teoretisointi ja oikeutus ovat enemmän luonnontieteellisiä mittareita, kun taas artefaktin rakentamisella ja arvioinnilla on enemmän suunnittelutieteellinen tarkoitus.

	Rakentaminen	**Arviointi**	**Teoretisointi**	**Oikeutus**
Käsitteet				
Mallit				
Metodit				
Toteutukset				

Taulukko 1: Tutkimuskehys. [March and Smith, 1995]

Luvussa 2 käsitellään yleisesti pelin ja erityisesti selainpohjaisten pelien määritelmää. Myös ohjelmistokehityksen keskeisiä termejä käydään läpi ja näitä vertaillaan pelinkehitysprosessiin. Luvun 2 lopussa esitellään erilaisia merkityksellisiä selainpohjaisia pelejä. Luvun 3 aiheena on selainpohjaisten peliteknologioiden nykytila, jossa käydään läpi erilaisia ohjelmointikieliä ja teknologioita, kuten Java sovelmat, Flash, Unity ja JavaScript. Luvussa 4 JavaScript otetaan tarkemmin käsittelyyn selainpohjaisten pelien kehitystä ajatellen. 3d- ja 2d-ympäristöjä vertaillaan keskenään ja erilaisia 3d-pelimoottoreita käydään läpi. Pohdinnassa ovat myös JavaScript-kielellä toteutettujen pelien hyödyt ja haitat. Luvussa 5 annetaan case-esimerkki selainpohjaisen pelin toteuttamisesta hyödyntäen uusimpia teknologioita. Case-esimerkin kautta pohditaan selainpohjaisuuden tuomia etuja ja haittoja. Lopuksi luvussa 6 käsitellään lyhyesti selainpohjaisten pelien tulevaisuutta ja mahdollisuutta sulauttaa pelejä verkkosivujen yhteyteen. Luvussa annetaan myös vastaus sille, miksi

selainpohjaiset teknologiat eivät ole ainakaan vielä vakiinnuttaneet asemaansa peliteollisuudessa ja miten tätä tilannetta voisi muuttaa.

2 Selainpohjainen peli

2.1 Pelin määritelmä

Smed ja Hakonen [2003] ehdottavat, että peli koostuu kolmesta elementistä: pelaajista, jotka osallistuvat peliin, säännöistä, jotka määrittelevät pelin, sekä erilaisista pelitavoitteista. He avaavat vielä käsitteitä esittelemällä osa-alueita yhdistäviä tekijöitä. Pelin tulee ensinnäkin sisältää haastetta. Haaste taas syntyy pelin säännöistä ja pelaajien sopimuksesta noudattaa kyseisiä sääntöjä. Tämän lisäksi tarvitaan jokin konflikti, esimerkiksi vastapelaaja, hankaloittamaan tavoitteiden saavuttamista. Myös satunnaisuus on yksi tapa lisätä peliin haastetta, jolloin pelaaja ei voi ennustaa pelin kulkua. Kolmanneksi pelissä tulee olla myös jokin sääntöjä konkretisoiva osuus. Tämä käsittää käytännössä sen, miten peli esitetään visuaalisessa muodossa.

On olemassa myös muita ajanvietteitä, joita ei sellaisenaan kuitenkaan voi laskea peleiksi. Erilaiset pulmat ja arvoitukset (puzzle), kuten palapelit, haastavat pelaajan ongelmanratkaisukyvyn. Ratkaisun selvittyä ongelman viehätys kuitenkin hiipuu konfliktin ja interaktiivisten elementtien puuttumisen johdosta.

Myös tarina (story) on pulmien tavoin tärkeä osa-alue pelissä, mutta yksinään se ei täytä pelin määritelmää. Peleissä oleva tarina eroaa tavanomaisesta lineaarisesta tarinankerronnasta, sillä pelaaja voi usein vaikuttaa tämän kulkuun. Erilaiset leikkivälineet (toy) ovat olennainen osa pelin representaatiota, mutta eivät toimi yksinään.

Perinteisten tietokonepelien rinnalla on jo koko internet-selainten olemassaolon ajan ollut erilaisia selainpohjaisia pelejä. Mikäli käyttäjä on asentanut jonkin selainpohjaisen pelin vaatiman teknologian, pystyy hän pelaamaan kaikkia kyseisen teknologian pelejä. Tietokonepelit ja nykyaikaiset konsolipelit tulee periaatteessa aina asentaa laitteen muistiin, mikä on aikaa vievää ja saattaa kasvattaa käyttäjän kynnystä kokeilla uutta peliä. Selainpohjaiset pelit ovat usein pelattavissa jo muutamien sekuntien sisään tietylle sivulle saapumisen jälkeen. Ongelmaksi voi kuitenkin nousta se, jos käyttäjä joutuu lataamaan jonkin selainpohjaisen pelin tarvitseman teknologian, kuten Javan, laitteelleen. Tähän käytetty aika saattaa saada käyttäjän toisiin aatoksiin pelin pelaamisesta.

2.2 Pelinkehitysprosessi

Projektilla tarkoitetaan jotain suunniteltua aktiviteettia. Hughes ja Cotterell [2009] määrittelevät ohjelmistoprojektin koostuvan useista ominaisuuksista. Esimerkiksi projektin kesto, laajuus, saatavilla olevat resurssit, suunnittelun tärkeys, projektiryhmän koko ja tehtävien haasteellisuus ovat keskeisiä ohjelmistoprojektin osa-alueita. Mitä

enemmän näitä osa-alueita sisällytetään projektiin, sitä haasteellisemmaksi se muuttuu. Esimerkiksi suurta projektiryhmää on huomattavasti vaikeampaa hallita ajankäytön kannalta, koska jokaiselle ryhmäläiselle on keksittävä tekemistä, joka ei estä muita ihmisiä tekemästä omia työtehtäviään.

Ohjelmistoprojekti koostuu yleensä erilaisista työvaiheista, jotka noudattavat usein jotain hyväksi havaittua mallia. Näistä yksi tunnetuimmista on vesiputousmalli. Tämä sisältää seitsemän pääkohtaa: vaatimusmäärittelyn, suunnitteluvaiheen, toteutusvaiheen, integraatiovaiheen, testausvaiheen, asennusvaiheen ja ylläpitovaiheen. Mallia on kritisoitu erityisesti siitä, että aikaisempiin työvaiheisiin ei tulisi palata enää projektin loppuvaiheessa. Täten projektin alkuvaiheessa tehdyt virheet saattavat tulla kalliiksi. Yleisesti ottaen erilaisia virheitä ja väärinymmärryksiä on mahdoton välttää projektin määrittely- ja toteutusvaiheessa. Tämän vuoksi ohjelmistojen kehityksessä hyödynnetään nykyisin lähinnä ketterän ohjelmistokehityksen malleja. Nämä sisältävät samoja työvaiheita kuin aikaisemmin esitelty vesiputousmalli, mutta projekti jaetaan lyhyen määräajan kestäviin iteraatioihin, joita kutsutaan sprinteiksi (sprint). Täten on mahdollista reagoida erilaisiin ongelmiin ja muutoksiin paljon dynaamisemmin ja muun muassa muuttaa erilaisia projektin määrittelyjä selkeämmiksi.

Ohjelmistoprojektin vaatimusten määrittely on yksi tärkeimmistä ja vaikeimmista työvaiheista. Prosessin tarkoituksena on muodostaa vaatimusmäärittelydokumentti, joka kertoisi yleisesti, millaista järjestelmää projektiryhmä on rakentamassa ja millaisia tavoitteita ja vaatimuksia ohjelmistoprojektilla on. Yleisesti ottaen vaatimukset jaotellaan kahteen osa-alueeseen, toiminnallisiin (functional) ja ei-toiminnallisiin (non-functional) vaatimuksiin. Toiminnalliset vaatimukset kuvailevat, miten järjestelmä reagoi erilaisiin syötteisiin toimintansa ja sisältönsä puolesta. Tämän tyyppiset vaatimukset saattavat vaihdella yleisen järjestelmäkuvauksen ja tietyn osa-alueen yksityiskohtaisen kuvauksen välillä. Toiminnallisten vaatimusten tärkein funktio on kuitenkin luoda yleinen kuva järjestelmästä ja käyttäjien tarpeista. [Sommerville, 2009]

Ei-toiminnalliset vaatimukset käsittelevät pääsääntöisesti ohjelmistoprojektin laatu- ja resursointivaatimuksia. Aika ja raha ovat resursointikysymyksiä, jotka vaikuttavat epäsuorasti projektin toiminnallisiin vaatimuksiin ja laatuvaatimuksiin. ISO 9126 -standardi [ISO-9126, 1996] määrittelee kuusi pääkriteeriä ohjelmistojen laadun evaluointiin: toiminnallisuuden, luotettavuuden, käytettävyyden, tehokkuuden, ylläpidettävyyden ja siirrettävyyden. Vuonna 2011 julkaistiin ISO 9126 -standardin seuraaja, ISO 25010 [ISO-25010, 2011]. Kyseinen standardi määrittelee kahdeksan ohjelmistotuotteen laadun vaatimusta kuuden sijaan. Nämä ovat toiminallisuuksien soveltuvuus (functional suitability), tehokkuus (performance efficiency),

yhteensopivuus (compatibility), käytettävyys (usability), luotettavuus (reliability), turvallisuus (security), ylläpidettävyys (maintainability) ja siirrettävyys (portability).

Vaatimusten määrittelyssä on hyvä kiinnittää projektin tärkeimmät laatuvaatimukset, sillä näitä kaikkia on mahdotonta toteuttaa samanaikaisesti. Sommerville [2009] ehdottaa, että tärkeimmät laatuvaatimukset koskevat erityisesti luotettavuutta ja tehokkuutta. Luotettavuus kertoo ohjelmiston yleisestä suoriutumisesta, esimerkiksi kyvystä palautua erilaisista virhetilanteista ja tiloista. Järjestelmän tulisi olla tehokas, sillä käyttäjät hylkäävät hitaat ohjelmistot. Ohjelmiston laatuvaatimukset koskettavat sekä ohjelmistokehittäjiä että loppukäyttäjiä. Tällöin on mahdollista, että kehittäjien näkemys laatuvaatimuksista eroaa merkittävästi loppukäyttäjien odotuksista. Loppukäyttäjät saattavat olettaa esimerkiksi ohjelmiston turvallisuusratkaisujen olevan eri tasolla kuin mitä kehittäjät ovat ajatelleet. Ylläpidettävyys ja siirrettävyys vaikuttavat myös enemmän laatukriteereinä ohjelmiston kehittäjiin kuin itse loppukäyttäjiin. [Hughes and Cotterell, 2009]

Perinteinen ohjelmistoprojekti ja peliprojekti eroavat toisistaan useilla eri osa-alueilla. Murphy-Hill ja muut [2014] haastattelivat tutkimuksessaan useita pelialalla työskenteleviä henkilöitä, jotka toimivat muun muassa ohjelmoijina ja projektipäällikköinä. Vaatimuksien määrittelyssä peleillä on usein vain yksi tärkeä vaatimus; pelin tulee olla hauska. Tämäkin vaatimus on usein subjektiivinen, sillä jokin määritelty ominaisuus voi osoittautua myöhemmässä vaiheessa tylsäksi [Murphy-Hill et al., 2014]. Pelin hauskuus voidaan nähdä kokoelmaksi erilaisia perinteisen ohjelmistoprojektin laatuvaatimuksia. Pelin suorituskyky vaikuttaa olennaisesti käyttäjän eli pelaajan pelikokemukseen positiivisesti tai negatiivisesti ja onkin yksi tärkeimmistä pelien laatuvaatimuksista. Pelin käytettävyys on myös tärkeää, sillä esimerkiksi huonosti opittavissa oleva, virhealtis ja ruma käyttöliittymä saattaa karkottaa pelaajat. Pelien ehkä suurin ero tavallisiin ohjelmistoihin on käyttöaika. Peleissä käyttäjä saattaa viettää useita tunteja jonkun tehtävän tekemiseen, kun taas perinteisessä ohjelmistossa tehtävä yritetään saada suoritettua mahdollisimman nopeasti. Tällöin hyvän käytettävyyden ja käyttöliittymän merkitys korostuu entisestään.

Peleissä on muuttuvammat vaatimukset kuin perinteisessä ohjelmistoprojektissa. Vaikka pelisuunnittelijalla olisikin tarkka näkemys pelin jostakin ominaisuudesta, voi tämä osoittautua pelattavuudeltaan huonoksi. Pelin virhetilanteet eivät ole niin pahoja kuin tavallisten ohjelmien virhetilanteet, koska peli on tarkoitettu viihdekäyttöön. Pelin vaatimuksiin vaikuttavat myös tätä edeltäneet versiot. Mikäli pelillä on aikaisempi versio, osa vaatimuksista saattaa tulla käyttäjiltä itseltään. Näin on mahdollista korjata aikaisempien versioiden puutteita tai miettiä, mitkä ominaisuudet aikaisemmissa

versioissa tukivat pelin hauskuusvaatimusta. Petrillo ja muut [2008] esittävät myös, että peliprojektien yksi suurimmista haasteista on pelin laajuus. Pelin kehitysprosessin aikana saatetaan lisätä uusia ominaisuuksia, joita ei alkuperäisessä suunnitteludokumentissa mainittu lainkaan (feature creep). [Murphy-Hill et al., 2014]

Peliprojektissa suunnitteluvaihe on lyhyempi kuin tavanomaisessa ohjelmistoprojektissa. Pelin arkkitehtuurin suunnitteluun ei usein kiinnitetä huomiota, koska taustalla on pelko siitä, että tehty työ menee hukkaan muutoksien seurauksena. Jos luotua peliarkkitehtuuria pitääkin jatkossa laajentaa, saattavat kehittäjät joutua ongelmiin suunnittelemattomuuden seurauksena. Koodia ei voi yleensä käyttää uudelleen eri peliprojektien välillä, sillä projektit vaativat usein hienosäätöä johonkin tiettyyn ongelmaan. Tavanomaisissa ohjelmistoprojekteissa muutoksiin varaudutaan nykyään paremmin, mutta peliprojekteissa kannustetaan erityisesti innovoimaan. [Murphy-Hill et al., 2014]

Vaikka tietyn pelin koodia on hankala hyödyntää uudestaan, erilaisia työkaluja voidaan hyödyntää useissa projekteissa. Pelimoottoreita, jotka ovat pelin toiminnallisuuden perusta, on mahdollista hyödyntää varsin erilaisissa peliprojekteissa. Erityisen tärkeä peleille on tuotantoketju (asset pipeline), joka tarkoittaa polkua konseptisuunnitteluvaiheesta siihen, että esimerkiksi 3d-malli on sisällytetty pelimaailmaan. Tätä ketjua varten pelinyhtiöt luovat usein omia ohjelmiaan, jotta kehitysprosessia saadaan nopeutettua ja standardoitua. [Murphy-Hill et al., 2014]

Pelejä testataan huomattavasti vähemmän hyödyntäen automaattitestejä kuin tavanomaisissa ohjelmistoprojekteissa. Yleensä testaus on manuaalista, eli käytännössä ihmiset pelaavat peliä ja havainnoivat sen toimivuutta ja käytettävyyttä. Automaattitestien luominen on haastavaa, sillä on hankala määrittää, mitä oikeanlainen toiminta tietyssä pelissä on. Testauksessa tulee myös ottaa huomioon kaikki pelin tilat, jotka pelaaja onnistuu luomaan, ottaen huomioon pelin yleisen satunnaisuuden. Pelin tulee yleensä myös toimia lukuisilla eri käyttöjärjestelmillä ja kokoonpanoilla, mikä lisää entisestään testaamisen monimutkaisuutta. [Murphy-Hill et al., 2014]

Peliprojektin johtohenkilöillä ei usein ole teknistä taustaa. Tämä hidastaa projektin edistymistä, sillä ohjelmoijat eivät usein voi puhua teknisistä ongelmistaan ja joidenkin asioiden priorisointi saattaa kärsiä huomattavasti [Murphy-Hill et al., 2014]. Kommunikaatio-ongelmia syntyy myös, koska peliprojekteissa on tavanomaista ohjelmistoprojektia enemmän taiteellisia ihmisiä, kuten muusikoita, käsikirjoittajia ja 3d-mallintajia [Petrillo et al., 2008].

2.3 Esimerkkejä selainpohjaisista peleistä

Varhaisimmat selainpohjaiset pelit ilmestyivät selainten yleistyessä 1990-luvun puolivälin jälkeen. Pelit olivat kuitenkin graafisesti hyvin yksinkertaisia, sillä selaimet eivät tuolloin vielä tukeneet nykyisten selainpohjaisten pelien hyödyntämiä teknologioita. Vanhatupa [2010] mainitsee, että yksi aikaisimmista selainpohjaisista peleistä on vuonna 1996 julkaistu Earth 2025. Kyseinen peli oli suunniteltu ensisijaisesti pelattavaksi selaimessa. Käytännössä peli on valtionhallintasimulaattori ja käyttöliittymässä käytetään HTML-koodia muun muassa nappuloissa ja lomakekentissä. Useat varhaiset selainpelit olivatkin pääasiassa tekstipohjaisia, esimerkiksi tekstiseikkailuja. Pelaajan tehtävänä oli lukea tapahtumat ruudulle näytetystä kuvasta tai tekstistä ja reagoida niihin kirjoittamalla tai painamalla ruudulla näkyvää painiketta.

RuneScape [2017] (kuva 1) on Jagex-yrityksen luoma massiivinen monen pelaajan roolipeli (Massive Multiplayer Online Role Playing Game). Peli noudattaa perinteisen tietokoneella pelattavan roolipelin kaavaa. Pelaaja ohjaa omaa hahmoaan, joka suorittaa erilaisia tehtäviä ja kerää itselleen kokemusta. Roolipelit kannustavat pelaajaa tutkimaan maailmaa ja taistelemaan erilaisia pelimaailman olentoja vastaan. RuneScapessa on mahdollista myös suorittaa tehtäviä muiden pelaajien kanssa. Muut pelaajat voivat myös toimia haastajina, sillä pelihahmot voivat taistella toisiaan vastaan. RuneScape ei sisällä massiivisena roolipelinä varsinaista pelin lopetusta, vaan peli jatkuu siihen asti, kun sen ylläpitäminen on kannattavaa. RuneScapen ensimmäinen versio julkaistiin vuonna 2001, mutta peli on saanut tämän jälkeen useita pelattavuutta ja ulkoasua parantavia päivityksiä. RuneScape oli yli kymmenen vuotta pelattavana selaimissa. Erityisesti pelin 3d-ulkoasu oli jotain, jota selainpeleissä ei aikaisemmin oltu totuttu näkemään. Nykyään peliä ei voi teknologiasyistä pelata kuin työpöytäsovelluksissa. Peli vaatii toimiakseen Java-ohjelmointikielen asentamisen tietokoneeseen.

Kuva 1: RuneScape.

Aapeli [2017] on vuonna 2002 avattu Playforia-yrityksen ylläpitämä peliyhteisösivusto, joka sisältää useita erilaisia pienempiä pelejä. Useimmat näistä ovat suoraan reaalimaailmasta digitoituja versioita, kuten shakki, biljardi ja minigolf. Sivusto sisältää myös muunlaisia pelejä ja näistä yksi tunnetuimmista on tykkipeli. Pelaaja ohjaa pelissä panssarivaunua tarkoituksenaan tuhota pelikentältä muut vastustajat. Pelaajan panssarivaunu ei voi liikkua, jolloin pelin haasteeksi muodostuu tykin osoittaminen oikeaan kohtaan, jotta ammus osuu vastustajaan. Peli sisältää erilaisia ammustyyppejä, joita voidaan hyödyntää vihollisen tuhoamiseen. Tykkipeli ja suurin osa Aapelin palvelun peleistä ovat pelattavissa vain oikeita ihmisiä vastaan. Pelit vaativat vaihdellen joko Java-ohjelmointikielen tai Adobe Flash Playerin asentamisen tietokoneelle. Täten esimerkiksi mobiili- ja tablettikäyttäjät eivät pääse pelaamaan sivuston pelejä, vaikka ne olisivat yksinkertaisuutensa vuoksi varsin sopivia näiden laitteiden käyttöön.

Habbo Hotelli [Habbo, 2017] (kuva 2) on suomalaisen Sulakkeen kehittämä ja julkaisema nuorisolle suunniteltu selainpohjainen peli. Pääominaisuuksiin kuuluu muun muassa muiden käyttäjien kanssa juttelu kirjoitetun kielen välityksellä ja omien hotellihuoneiden luominen ja sisustaminen. Peli julkaistiin alun perin jo vuonna 2000 ja sillä on edelleen varsin laaja käyttäjäkunta. Hotelleja löytyy nykyisellään useasta eri maasta ja käyttäjät voivat kirjautua hotelleihin yli sadasta eri maasta. Hotelliin on tullut vuosien varrella useita eri päivityksiä ja muutoksia, mutta pelin graafinen ulkoasu ja perusominaisuudet ovat pysyneet käytännössä muuttumattomina. Pelimoottorissa hyödynnetään edelleen Flash-pohjaista teknologiaa ja ActionScript-ohjelmointikieltä. Nykyisellään pelistä on olemassa myös mobiiliympäristöön suunniteltu versio, joka

käyttää hyväkseen alustojen natiiveja ohjelmointiteknologioita. Kehittäjät ovat maininneet, että Flash-version tuominen mobiililaitteille ei ole ollut helppo projekti, sillä Habbo sisältää paljon vuosien varrella toteutettua toiminallisuutta [Lappalainen, 2015]. Tämän vuoksi todennäköisesti myös uusimpia web-teknologioita hyödyntäviä versiota ei ole toteutettu, sillä tämä tarkoittaisi koko ohjelmiston rakentamista uudestaan. Uusimmat web-teknologiat voisivat kuitenkin mahdollistaa saman koodipohjan käytön sekä mobiili-, tablet- että työpöytäympäristöissä ja näin kasvattaa käyttäjäkuntaa entisestään.

Kuva 2: Habbo hotelli.

3 Selainpohjaisten peliteknologioiden nykytila

3.1 Selaimet ja esityskerros

Verkkoselain, eli lyhyesti vain selain, on sovellus, joka mahdollistaa internet-verkossa olevan hajautetun hypertekstijärjestelmän selaamisen. Selain hakee erilaisia dokumentteja (web-sivuja) palvelimilta ja palauttaa ne käyttäjän näytölle näkyvässä muodossa. Ensimmäisen selaimen kehitti Tim Berners-Lee vuonna 1990, jonka jälkeen useat yhtiöt ovat tuoneet omia selaintoteutuksiaan markkinoille. Nykyisin suosituimpia selaimia ovat Googlen Chrome, Mozillan Firefox, Microsoftin Edge, Applen Safari ja Opera Softwaren Opera.

Ohjelmistot rakentuvat erilaisista kerroksista, ja selainpohjaisissa teknologioissa pääosassa ovat kaksi kerrosta, esityskerros (front-end) ja palvelinkerros (back-end). Selaimet ja näihin liittyvät teknologiat, kuten HTML (Hypertext Markup Language), CSS (Cascading Style Sheets) ja JavaScript muodostavat pääasiallisen esityskerroksen. HTML-koodin tehtävä on kuvata web-sivun rakenne. Käytännössä sivustot muodostuvat HTML-elementeistä, jotka voivat olla sisäkkäin (block) tai perättäin (inline). Web-sivulle annetaan tyyliohjeita CSS-kielellä. Tyylittelyn avulla on mahdollista lisätä sivustolle erilaisia fontteja, värejä, animaatioita ja määrittää elementtien muotoa, asettelua ja kokoa. JavaScript-ohjelmointikieli mahdollistaa dynaamisen toiminnallisuuden. Esimerkiksi HTML-dokumentin rakenteesta muodostettu DOM-puuta (Document Object Model) on mahdollista muokata hyödyntäen JavaScript-kieltä. Esityskerrosta voidaan tietyissä tapauksissa kutsua myös asiakkaan puoleiseksi (client-side) kerrokseksi, koska tätä kerrosta ohjelman pääasiallinen käyttäjä operoi.

Selaimet rakentuvat tarkemmin seitsemästä eri osa-alueesta, joista näkyvin on käyttöliittymä. Tämä koostuu muun muassa osoiterivistä, navigaatiopainikkeista ja välilehdistä. Suurin osa selaimen käyttöliittymästä on kuitenkin varattu itse verkkosivujen esittämistä varten. Selainmoottorin tärkein tehtävä on vastata piirtomoottorin toiminnasta. Piirtomoottori vastaa pyydetyn sisällön näyttämisestä. Jos haluttu sisältö on esimerkiksi HTML-koodia, jäsentää piirtomoottori sen näkyväksi näytölle. Piirtomoottori myös hyödyntää erilaisia valmiiksi suunniteltuja komponentteja itse selainikkunan ja esimerkiksi pudotusvalikkojen piirtämiseen. Kaikki selaimet eivät hyödynnä samaa piirtomoottoria. Esimerkiksi Googlen Chrome ja Applen Safari käyttävät Webkit-moottoria, kun taas Mozillan Firefox hyödyntää Gecko-moottoria. [Garciel, 2009]

Selaimet sisältävät myös verkkoliikenteestä huolehtivat laitteistoriippumattoman toteutuksen ja yksinkertaisen tietovaraston esimerkiksi evästeiden (cookies)

säilömiseen. Selaimiin on rakennettu tuki JavaScriptille. Käytännössä selaimet sisältävät JavaScript-moottorin, joka vastaa koodin tulkkaamisesta ja ajamisesta. Kyseisiä moottoreita on lukuisia, joista suosituin on Googlen kehittämä V8-moottori. Muita mainittavia JavaScript-moottoreita ovat Mozillan kehittämä SpiderMonkey ja Applen JavaScriptCore. JavaScript-moottorit voidaan luokitella prosessikohtaisiksi virtuaalikoneiksi (virtual machine). Virtuaalikone emuloi oikean tietokoneen toimintaa ja erityisesti prosessikohtaisen virtuaalikoneen tarkoitus on suorittaa käytännössä laitteistosta riippumatonta ohjelmakoodia. [Garciel, 2009]

Useissa selaimissa on myös tuki liitännäisille. Liitännäinen on ohjelmistokomponentti, joka tarjoaa lisätoiminnallisuutta johonkin olemassa olevaan ohjelmaan. Selaimien liitännäiset, eli selainliitännäiset, voivat lisätä esimerkiksi virusturvan tai mahdollisuuden esittää jonkin tietyn tiedostotyypin tiedostoja, kuten videoita. Ehkä tunnetuin selainliitännäinen on Adobe Flash Player, joka mahdollistaa Flash-multimediaesitysten toiston. Myös Java-sovelmien ajon mahdollistava liitännäinen on ollut erittäin tunnettu ja käytetty. Pääasiallisesti selainliitännäisistä hyötyvätkin kolmannen osapuolen kehittäjät, jotka haluavat tuoda omia tuotteitaan selainkäyttöön. Näin vältetään myös lisensointiongelmat, kun liitännäisten ei tarvitse olla yhteydessä selaimen lähdekoodiin muuta kuin erillisen rajapinnan kautta. Selainliitännäiset mahdollistavatkin selaimen räätälöinnin käyttäjän tarpeita varten. Selainvalmistajat voivat kuitenkin estää liitännäisten käytön, mikäli niissä on esimerkiksi tietoturvariskejä. Applen Safarissa esimerkiksi Flash-liitännäisen käyttö ei ole enää sallittua.

3.2 Palvelinpuoli

Palvelinkerros on yleensä täysin eriytetty esityskerroksesta, sisältäen pääasiassa liiketoimintalogiikan ja jonkinlaisen tietosäiliön, esimerkiksi tietokannan. Palvelimen pääasiallinen tehtävä on vastata selaimen pyyntöihin palauttamalla staattisesti tai dynaamisesti generoituja web-sivuja.

Yksinkertaisissa selainpohjaisissa peleissä tarvitaan usein vain palvelin, joka lähettää pelin tarvitseman lähdekoodin ja graafiset tiedostot käyttäjän tietokoneelle. Tällöin toimintalogiikka on kokonaisuudessaan esityskerroksessa. Monimutkaisemmissa peleissä palvelin voi kuitenkin tehdä muutakin. Peleissä, joissa useampi pelaaja pelaa peliä samaan aikaan, tarvitaan usein palvelinta synkronoimaan tietoja kahden pelaajan välillä. Tällöin pelaajan selaimen on muodostettava jatkuva yhteys palvelimen kanssa. Useamman samanaikaisen pelaajan peleissä usein myös pelin varsinainen toimintalogiikka sijaitsee palvelinpuolella. Näin esityskerrosta käytetään vain esittämään pelaajalle peli ja vastaanottamaan syötteitä. Tällä tavoin on myös mahdollista estää pelaajien mahdolliset yritykset huijata pelissä.

3.3 Java

Java on alun perin Sun Microsystemsin (nykyisin omistajana Oracle) kehittämä oliopohjainen ohjelmointikieli. Javan lähdekoodia ei käännetä suoraan konekieleksi, vaan jokaista lähdekooditiedostoa varten luodaan aluksi tavukoodia sisältävä tiedosto. Tavukoodi on laitteiston ja käyttöjärjestelmän suhteen itsenäistä ohjelmakoodia, joka sisältää lähes kaiken alkuperäisen lähdekoodin informaatiosta. Käännösprosessin toisessa vaiheessa Javan virtuaalikone (Java Virtual Machine) suorittaa tavukooditiedostot ja muuntaa nämä konekoodillisiksi ohjeistuksiksi.

Java sisälsi jo julkaisuvaiheessa vuonna 1995 tuen myös selainpohjaisia sovelluksia varten. Nämä Java-sovelmat (Java applet) ovat Java-ohjelmia, jotka voidaan ajaa selaimesta käsin. Sovelmat voivat olla selaimessa HTML-koodin seassa tai kokonaan erillisessä ikkunassa. Sovelmat ajetaan täysin omassa erillisessä tilassaan, jolloin ne eivät teoriassa voi aiheuttaa tietoturvariskiä. Tällöin sovelma ei pääse käsiksi esimerkiksi käyttöjärjestelmän tiedostojärjestelmään. Java-sovelmat hyödyntävät laitteistokiihdytystä ja täten siis esimerkiksi näytönohjaimen tuomaa laskentaetua graafisissa sovelluksissa.

Java on ollut käytössä pelinkehityksessä vaihtelevalla menestyksellä. Varsinaisia suuria ja menestyneitä pelejä on ollut melko vähän. Työpöytäkäytössä esimerkiksi Minecraft [2017] ja Project Zomboid [2017] ovat erittäin menestyneitä. Minecraft on hiekkalaatikkopeli, jossa on mahdollisuus luoda rakennelmia ja seikkailla kolmiulotteisessa maailmassa. Project Zomboid taas on avoimen maailman selviytymiseen keskittyvä kauhupeli. Java-sovelmien ehdottomasti menestynein peli on jo esitelty RuneScape. Vaikka Java-koodin kirjoittaminen on teoriassa yksinkertaisempaa kuin esimerkiksi C- tai C++-koodin kirjoittaminen, on pelien kehittäminen hankalampaa. Java ei ole yhtä suorituskykyinen kieli kuin esimerkiksi C++ ja Java-koodin kääntäminen esimerkiksi konsolipeliksi on käytännössä mahdotonta.

Javan tulevaisuus selainpohjaisten pelien alustana näyttää erityisen huonolta lukuisista syistä johtuen. Javassa itsessään on ollut useita tietoturvaongelmia, jotka ovat vähentäneet kielen ja sen ohjelmien käyttäjäkuntaa. Yleensä virallinen suositus on ollut näissä tilanteissa poistaa Java kokonaan käyttöjärjestelmästä, mikä taas estää sovelmien käytön selaimissa. Java-sovelmat alkavat olla myös yleisesti vanhentunutta teknologiaa. Erityisesti vanhemmilla Java-versioilla kirjoitetut sovelmat sisältävät turvallisuusongelmia, ja jotta näitä voi edes ajaa selaimessa, tulee Javan asetuksista säätää turvallisuusasetuksia heikommiksi. Turvallisuusasetuksien muuttaminen voi olla peruskäyttäjälle haasteellista ja arveluttavaa.

JavaScript on ohjelmointikielenä kasvattanut suosiotaan ja saanut suorituskykyparannuksia, erityisesti Googlen V8 JavaScript-moottorin takia. Viidennen HTML-standardin (HTML5) esittelemä canvas-elementti mahdollistaa 2d-piirtämisen ja WebGL (Web Graphics Library) [WebGL, 2017] 3d-piirtämisen hyödyntäen laitteistokiihdytystä ja vähentäen entisestää tarvetta Javan käytölle. Nämä esitellyt uudet ratkaisut toimivat myös asentamatta mitään kolmannen osapuolen sovelluksia, toisin kuin Java. Nykyiset mobiili- ja tablettiselaimet eivät myös yleensä tue Javaa. Myös työpöytäselaimet, kuten Chrome, ovat lopettaneet tukensa Java-sovelmille.

3.4 Flash

Adobe Flash on Adobe Systemsin kehittämä multimedia-alusta, jonka avulla on mahdollista luoda muun muassa animaatioita, mobiili- ja selainpelejä ja erilaisia työpöytä- ja mobiilisovelluksia. Luodut sovellukset toimivat selainpohjaisessa ympäristössä hyödyntäen Adobe Flash Playeriä, joka on asennettava erillisenä selainliitännäisenä työpöytäselaimissa. Erityisesti vähän ohjelmointikokemusta omaavat ihmiset ovat pitäneet Flash-teknologian elossa, sillä Adobe tarjoaa suhteellisen helpon ohjelmointiympäristön, jossa on mahdollista muun muassa luoda helposti käyttöliittymäelementtejä ja muuta sisältöä.

Adobe Flash sovellusten tai pelien ohjelmoinnissa käytettävä kieli on ActionScript, joka pohjautuu löyhästi JavaScript-ohjelmointikieleen. Flashiä on käytetty huomattavan paljon erilaisissa peleissä vaikkakin suurin osa näistä on melko pieniä. Esimerkiksi suurin osa Facebook-peleistä on toteutettu Flashiä hyödyntäen.

Koska Adobe Flash Player on tullut tiensä päähän erinäisten ongelmien vuoksi, ovat Flash-kehittäjät siirtyneet kohti WebGL-teknologiaa. Kyseinen teknologia mahdollistaa Flash-sovellusten ajamisen selaimessa ilman liitännäisen asennusta. Tämä kuitenkin tarkoittaa sitä, että Flash-pohjaiset selainpelit tulee kääntää WebGL-muotoon, eli käytännössä JavaScript-kielelle. Käännettyyn peliin täytyy myös sisällyttää kaikki pelin tarvitsemat kirjastot, jolloin paketti voi paisua melko suureksi.

3.5 Unity ja Unity Web Player

Unity [2017] on Unity Technologies -yrityksen kehittämä 3d-pelimoottori, jonka ensimmäinen versio valmistui vuonna 2005. Nykyisin pelimoottori on jo versiossa viisi. Vuonna 2010 Unity lanseerasi Asset Storen, johon käyttäjät voivat luoda maksullisia komponentteja muiden käytettäväksi pelinkehitysprosessia varten. Unity tarjoaa erittäin modernin ja helppokäyttöisen pelinkehitysympäristön. Unityn keskipisteessä on itse kehitysalusta (editor), jota käyttämällä kehittäjä voi lisätä peliin erilaisia

peliobjekteja kuten 3d-malleja, ääniä, kameroita ja valoja. Näiden peliobjektien käyttäytymistä voidaan hallinnoida kooditiedostojen avulla, joita Unity kutsuu skripteiksi. Unity sallii kolmen eri ohjelmointikielen käytön skripteissä. Microsoftin kehittämä C# on kielistä käytetyin UnityScriptin tullessa seuraavana. UnityScript muistuttaa JavaScriptiä, mutta käytännössä näillä ei ole kovinkaan paljon yhteistä. Kolmas mahdollinen skriptauskieli on Boo, jonka käyttö on hyvin vähäistä.

Unityn yksi suurimmista eduista on mahdollisuus kääntää peli samalla lähdekoodilla eri alustoille. Unity mahdollistaakin kääntämisen muun muassa tietokoneiden eri käyttöjärjestelmille (Windows, OsX, Linux), pelikonsoleille, mobiililaitteille ja selaimille. Unity Web Player on selainliitännäinen (browser plugin), joka suunniteltiin alun perin Unityn selainpohjaisten pelien suorittamista varten. Tätä liitännäistä tukevat vain Windows- ja Mac-pohjaiset alustat. Liitännäinen tarjoaa suuren osan Unityn ominaisuuksista, ja teoriassa selainpeli voisi näyttää täysin identtiseltä kuin mille tahansa muulle alustalle toteutettu peli. Web Playeristä kuitenkin puuttuu muutama olennainen osa turvallisuussyistä, kuten oikeudet käyttäjän hyödyntämän käyttöjärjestelmän tiedostojärjestelmän muokkaamiseen. Suuret selaimet kuten Google Chrome ja Microsoft Edge ovat kuitenkin jo lopettaneet Unity Web Playerin tukemisen. Liitännäistä hyväksikäyttäen ei ole myöskään julkaistu suuria selainpohjaisia pelejä, vaan Web Player on jäänyt lähinnä pelinkehittäjien leikkikaluksi ja työkaluksi tuoda omia tuotoksiaan nopeasti muiden nähtäväksi.

Koska selainliitännäinen on tullut tiensä päähän, on Unity siirtynyt käyttämään WebGL-teknologiaa, joka ei vaadi erillisen liitännäisen asentamista. Teoriassa myös laitteistoyhteensopivuuden pitäisi olla rajaton, sillä WebGL vaatii toimiakseen vain modernin selainympäristön. PlayCanvas [PlayCanvas, 2017] on erityisesti WebGL-teknologiaa varten kehitetty JavaScript-pelimoottori ja kehitysympäristö. Eastcott [2016], yksi tämän moottorin pääkehittäjistä, vertaili Unityä ja PlayCanvas-pelimoottoria. Unityn WebGL-ratkaisu on hänen mukaansa tiedostokooltaan moninkertaisesti suurempi. PlayCanvas käyttää hyväkseen puhdasta JavaScriptiä, kun taas Unityllä toteutetun pelin kääntäminen esimerkiksi C#-kielestä JavaScriptiksi on monimutkainen prosessi, joka synnyttää paljon ylimääräistä koodia. Täten Unityllä toteutetun pelin ensimmäinen käynnistyskerta on myös huomattavasti PlayCanvasta hitaampi. Unityn WebGL-pelien paisunut koodimäärä saattaa myös johtaa joidenkin mobiiliselainten kaatumiseen muistin loppumisen vuoksi. Unityn WebGL tuki on siis edelleen varsin keskeneräinen.

4 JavaScript selainpohjaisissa peleissä

4.1 JavaScript kielenä

JavaScript on oliopohjainen ja alustariippumaton ohjelmointikieli, joka julkaistiin alun perin vuonna 1995. Kielen pääasiallinen tarkoitus oli tuoda staattisille web-sivustoille dynaamista sisältöä suhteellisen vaivattomasti.

Kehittäjät ovat aikaisemmin suhtautuneet JavaScriptiin melko kielteisesti. Usein JavaScript nähdään enemmän työkaluna kuin varsinaisena ohjelmointikielenä. Web-ohjelmoinnissa ollaan aikaisemmin totuttu siihen, että JavaScriptiä hyödynnetään vain käyttäjälle näkyvän sisällön muokkaamiseen ja joidenkin pienten ulkoasua parantavien animaatioiden toteuttamiseen. Aikaisemmat web-teknologiat ovatkin nojanneet pääasiassa templaattipohjaisiin (template) ratkaisuihin, jossa koko web-sivun sisältö on luotu palvelinpuolella. Tällöin suurin osa sisällöstä on ollut täysin staattista ja laajemmat muutokset sivuille ovat vaatineet palvelimen luomaan uuden templaatin vanhan tilalle. Nykyisin on mahdollista kysyä palvelimelta vain pelkkää dataa ja reagoida JavaScriptin avulla sivuston rakenteen muutoksiin.

JavaScript jaetaan pääasiassa kahteen eri sovellusalueeseen. Selainpuolen JavaScript keskittyy pääasiassa käyttäjän selaimen ja DOM-mallin (Document Object Model) manipuloimiseen. JavaScriptin avulla on mahdollista lisätä helposti uusia elementtejä HTML-dokumenttiin tai esimerkiksi ottaa vastaan käyttäjän syötteitä näppäimistöltä. Tätä prosessia helpottamaan luotiin jQuery-kirjasto, jonka avulla on mahdollista esimerkiksi luoda animaatioita, käsitellä näppäinsyötteitä ja valita kätevästi eri HTML-elementtejä. Web-pohjaisten sovellusten monimutkaistumisen myötä markkinoille on ilmestynyt uusia kirjastoja, jotka entisestään parantavat JavaScriptiä. Nämä kirjastot, kuten AngularJS ja React, ovat vieneet JavaScriptin kehitystä modulaarisempaan suuntaan. Modulaarisuus on tuonut parempia käytäntöjä erityisesti JavaScript-koodin jakamiseen järkeviin kokonaisuuksiin.

JavaScriptiä käytetään yleisesti myös muuallakin kuin selaimessa. NodeJS on JavaScript-ajoympäristö (runtime), joka hyödyntää Googlen kehittämää V8 JavaScript-moottoria. Kyseinen ajoympäristö mahdollistaa JavaScriptin suorittamisen komentoriviltä, jolloin se sopii mainiosti työpöytäkäyttöön tai palvelinympäristöön. Palvelinpuolen JavaScript onkin toinen yleinen kielen käyttötarkoitus.

JavaScriptissä perustyyppejä ovat vain numerot, merkkijonot, totuusarvomuuttujat, sekä null ja undefined. Kaikki muut määritellään objekteiksi (object). Perustyypit ovat ensisijaisesti muuttumattomia (immutable), eli näitä muutettaessa luodaan aina uusi

instanssi. JavaScriptissä taulukot, funktiot, säännölliset lausekkeet (regular expressions) ja objektit itsessään ovat objekteja. Objektilla tarkoitetaan ominaisuuksien säiliötä, jossa jokaisella ominaisuudella on nimi ja arvo. Arvo voi käytännössä olla mikä tahansa, paitsi undefined-muuttujan arvoksi ei voi asettaa mitään. Objektit voivat sisältää käytännössä mitä tahansa, esimerkiksi toisia objekteja. [Crockford, 2008]

JavaScriptin yksi kiistanalaisista ominaisuuksista on dynaaminen tyypitys, sillä mikä tahansa muuttuja voi sisältää mitä tahansa dataa. Useimpien ohjelmointikielien staattinen tyypitys takaa, että muuttujien tyypit ovat oikeanlaiset ajon aikana. JavaScript ei tuo samanlaista turvaa. Tätä ongelmaa ratkomaan on kehitetty kirjastoja, kuten TypeScript, jotka lisäävät kieleen tyypityksen. Tämä ei kuitenkaan takaa, että ohjelmakoodi itsessään olisi täysin ongelmatonta. Lisäksi kehittäjät joutuvat kirjoittamaan koodia, joka ei ole enää täysin puhdasta JavaScriptiä. Näin kielen päälle rakennetaan vain ylimääräinen abstraktiokerros, jonka todellinen hyöty on kyseenalainen.

Objektit voivat myös periä toisten objektien ominaisuuksia. Tätä kutsutaan prototyyppiseksi periyttämiseksi (prototypical inheritance). Tämä periyttämistapa on niin sanottua klassista periyttämistä huomattavasti yksikertaisempi malli, sillä esimerkiksi Java-ohjelmointikielessä käytettyjen abstraktien luokkien käyttöä voidaan välttää. Useat JavaScript-ohjelmoijat pitävät klassista periytymistä liian rajoittavana ja vanhentuneena mallina. Elliot [2014] esittää, että periyttäminen luo useita ongelmia. Tiukat vanhempi-lapsi-suhteet eivät välttämättä riitä jokaiseen käyttötapauksen läpikäymiseen, esimerkiksi joitain vanhempien ominaisuuksia ei tarvittaisi ollenkaan lapsessa. Luokkien refaktorointi voi olla hankalaa, mikäli useat lapset käyttävät kyseistä luokkaa vanhempanaan. JavaScript-ohjelmoinnissa tulisikin siis ensisijaisesti noudattaa prototyypillisen periytymisen mallia, eikä yrittää muuttaa kieltä ja sen periaatteita. Uusin JavaScript-määrittely, ECMAScript 2015 (ES6 tai ES2015), lisää kieleen myös luokat, jotka voivat periä asioita käyttäen perinteistä extends-avainsanaa. Tämä on kuitenkin lisätty kieleen vain, jotta uudet ohjelmoijat tuntisivat olonsa tutuksi. Pinnan alla kaikki toimii kuitenkin prototyyppiperiytymisen avulla, mikä toisaalta lisää kielen sekavuutta entisestään.

Kuten jo aikaisemmin todettiin, myös funktiot ovat JavaScriptissä objekteja. Täten näitä voidaan hyödyntää kuin mitä tahansa muutakin arvoa. Tämä mahdollistaa funktioiden säilyttämisen muuttujissa, taulukoissa tai objekteissa. Koska funktiot itsessään ovat objekteja, voidaan näihin sisällyttää uusia funktioita. Funktioille annettavat parametrit voivat olla mitä tahansa ja puuttuvat parametrit korvataan vain undefined-muuttujalla. Funktioita on mahdollista antaa myös argumentteina toisille

funktioille ja funktiot voivat palauttaa toisia funktioita. Tämä mahdollistaa funktionaalisen ohjelmointiparadigman JavaScriptissä. [Crockford, 2008]

JavaScriptin ensimmäinen versio kehitettiin erittäin nopeasti. Täten kieli sisältää edelleen suorastaan haitallisia ominaisuuksia ja sudenkuoppia, joihin muihin ohjelmointikieliin tottuneiden ohjelmoijien on helppo pudota. Mikäli muuttujien alustamisessa ei käytetä var-, let- tai const-määrittelyjä, liitetään muuttuja globaaliin nimiavaruuteen. Tällöin muuttuja on kaikille näkyvissä esimerkiksi selaimen globaalissa window-objektissa yhtenä arvona. Mikäli muuttuja alustetaan kaikkien funktioiden ulkopuolella, siirtyy se myös globaaliin nimiavaruuteen. Tämä voi aiheuttaa erilaisia sivuvaikutuksia ja nimikonflikteja. [Crockford, 2008]

Useimmissa ohjelmointikielissä koodilohkossa määritellyt muuttujat eivät näy lohkon ulkopuolelle. JavaScript ei kuitenkaan noudata tätä periaatetta, mikäli muuttuja alustetaan var-sanalla. Funktion sisällä olevat muuttujat ovat käytettävissä missä osassa funktiota tahansa. Esimerkiksi if-lohkossa alustettu muuttuja näkyy myös lohkon ulkopuolelle [Crockford, 2008]. Modernissa JavaScriptissä tämä ongelma on kuitenkin kierrettävissä käyttämällä ECMAScript 2015 -määrittelyn mukaisia const- ja let-sanoja. Nämä noudattavat tavanomaista muuttujien lohkonäkyvyyden periaatetta.

Koodiesimerkeissä 1 ja 2 on havainnollistettu yksinkertaisen JavaScript-ohjelmakoodin toimintaa. Koodiesimerkki 1 hyödyntää ECMAScript 5 -standardia. Ohjelma sisältää todo-funktion, joka sisältää yksityisen items-taulukon. Tähän taulukkoon on mahdollista lisätä uusia tehtäviä, poistaa tehtäviä ja listata kaikki mahdolliset tehtävät. Nämä mahdolliset toiminnot palautetaan todo-funktion return-lauseessa. Yksi JavaScriptin voimakkaimmista ominaisuuksista on sulkeuma, jota molemmissa koodiesimerkeissä hyödynnetään estämään items-taulukon näkyminen julkisesti. Koska add-, remove- ja list-funktioilla on pääsy ulompaan näkyvyysalueeseen, eli todo-funktion alaisuuteen, on items-taulukko näiden funktioiden muokattavissa. JavaScriptissä lähes ainoa tapa määritellä näkyvyyttä pohjautuu funktionäkyvyyteen, sillä kaikki funktiot ovat oletuksena sulkeumia. Koodiesimerkki 2 sisältää muutamia yksinkertaistuksia ensimmäiseen esimerkkiin nähden. Muuttujat alustetaan hyödyntäen const-avainsanaa, jolloin näitä ei voi esitellä uudelleen. Todo-funktion palauttamaa rajapintaa on myös yksinkertaistettu. Uusi objektisyntaksi mahdollistaa rajapinnan funktioiden määrittelyn yksinkertaisemmalla tavalla. Poistofunktiossa tapahtuvan poistettavan indeksin selvitys on yksinkertaisempaa, koska uuden standardin mukana on tullut useita apufunktioita. ECMAScript 6 on täysin takaisinyhteensopiva ECMAScript 5 -standardin kanssa, jolloin olisikin mahdollista kirjoittaa ECMAScript 5 -lähdekoodia uudemman standardin sekaan.

```
var todo = function () {
  var items = [];

  return {
    add: function (id, task, description) {
      items.push({
        id: id,
        task: task,
        description: description
      });
    },
    remove: function (id) {
      var index = null;
      for (var i = 0; i < items.length; i++) {
        if (items[i].id === id ) index = i;
      }
      items.splice(index, 1);
    },
    list: function () {
      return items;
    }
  };
}();

todo.add(1, 'thesis work', 'programming thesis case example');
todo.add(2, 'thesis work', 'writing thesis');
todo.add(3, 'thesis work', 'writing thesis');
todo.remove(2);
```

Koodikatkelma 1: Tehtävälistaus käyttäen ECMAScript 5 -standardia.

```
const todo = function () {
  const items = []

  return {
    add (id, task, description) {
      items.push({id, task, description})
    },
    remove (id) {
      items.splice(items.findIndex(item => item.id === id), 1)
    },
    list () {
      return items
    }
  }
}()

todo.add(1, 'thesis work', 'programming thesis case example')
todo.add(2, 'thesis work', 'writing thesis')
todo.add(3, 'thesis work', 'writing thesis')
todo.remove(2)
```

Koodikatkelma 2: Tehtävälistaus käyttäen ECMAScript 6 -standardia.

4.2 JavaScript pelinkehityksessä

Peliprojektissa joudutaan heti alussa miettimään pääasiallinen alusta, jolle peli toteutetaan. Yksi suurimmista kysymyksistä on, kannattaako selainpohjaiseen teknologiaan käyttää resursseja esimerkiksi tavanomaisten työpöytäteknologioiden sijaan. Teknologian valintaan liittyy usein erilaisia kriteerejä, jotka määrittelevät teknologian kannattavuutta kehittäjiä ajatellen. Kyseiset kriteerit liittyvät usein siihen, onko uusi teknologia jollain tavalla parempi kuin vanha. Mikäli selainpohjaisilla teknologioilla toteutetut pelit olisivat suorituskykyisempiä ja sisältäisivät hyödyllisiä ominaisuuksia ja työkaluja, olisi valinta helppo. Nykyisellään selainpohjaiset peliteknologiat eivät täysin pääse esimerkiksi natiiviin työpöytäympäristöön toteutetun pelien suorituskyvylliselle tasolle, sillä teknologia on verrattain uutta. Työkalut ja teknologian ominaisuudet ovat myös toistaiseksi sekavalla tasolla. Laajasta selainpohjaisesta JavaScript-pelistä ei ole myöskään olemassa esimerkkitoteutusta. Peliyhtiöt eivät uskalla panostaa rahaa projekteihin, jotka eivät mahdollisesti onnistu, jos kehittämiseen tarvittavat työkalut eivät ole tarpeeksi valmiita. Esimerkiksi reaaliaikaisen selainpohjaisen pelin toteuttaminen JavaScriptillä on edelleen suorituskyvyllisesti vaikeaa. JavaScript on kielenä itsessään nuori verrattuna C- tai C++-kieliin. Kieli on myös melko sekavasti toteutettu.

JavaScript ei aikaisemmin voinut hyödyntää laskennassa esimerkiksi laitteistokiihdytystä, mutta tämä on muuttunut selainten saaman WebGL-tuen myötä. WebGL on vapaasti käytettävä, useamman alustan (cross-platform) ohjelmointirajapinta, joka mahdollistaa 3d-piirtämisen selaimessa. Käytännössä WebGL on mukaelma OpenGL ES 2.0 -rajapinnasta, joka on ollut käytössä erityisesti mobiililaitteissa. WebGL hyödyntää HTML5-standardissa esiteltyä canvas-elementtiä pyytämällä tältä erityisen piirtokontekstin. Canvas-elementti voidaan sijoittaa mihin tahansa HTML-dokumentin näkyvissä olevan rakenteen sisään. WebGL avaa mahdollisuuden käyttää laitteistokiihdytystä, mikä hyödyttää erityisesti raskaiden 3d-ohjelmien ajamisessa selaimessa. Aikaisemmin kehittäjät joutuivat turvautumaan esimerkiksi selainliitännäisiin ja muihin hankaliin ratkaisuihin tuodakseen 3d-kiihdytettyä sisältöä käyttäjilleen. WebGL-rajapinnasta on tulossa uusi versio, WebGL 2, joka tuo OpenGL ES 3.0 -rajapinnan ominaisuuksia saataville. [Parisi, 2012]

Aikaisemmat esitellyt ohjelmointikielet, kuten Java, Flash, ja kehitysympäristöt kuten Unity, ovat yrittäneet jo vuosia parantaa selainpohjaisten pelien asemaa. Nykyisellään kyseiset teknologiat ovat kuitenkin sellaisenaan kuolemassa pois selainkäytöstä. Nämä ovat kuitenkin alkaneet hyödyntää JavaScriptia ja selaimen WebGL-tukea peliprojekteissaan. Tämä on toteutettu tarjoamalla kehittäjille mahdollisuus muuntaa näillä teknologioilla toteutetut pelit WebGL-muotoon. Koska WebGL on käytännössä

JavaScript ohjelmointirajapinta, käännetään pelit käytännössä JavaScriptiksi. Miksi pelejä ei siis kirjoiteta puhtaana JavaScriptinä alusta alkaen?

4.3 JavaScript 3d-pelimoottorit

Tässä tutkielmassa tarkastellaan vain 3d-pelimoottoreita, jotka tukevat WebGL-rajapintaa ja täten toteuttavat erilaisia graafisia ominaisuuksia. Näitä ovat muun muassa valot, varjostukset ja erilaiset materiaalit. Pelimoottorin tulee pystyä jonkinasteiseen fysiikka- ja törmäyksenlaskentaan. Pelimoottoria voidaan hyödyntää kirjoittamalla puhdasta JavaScriptiä. Mukaan ei siis lasketa kirjastoja, joiden avulla pelejä kirjoitettaisiin jollain toisella ohjelmointikielellä ja vain käännetään JavaScriptiksi, kun peliohjelmaa halutaan suorittaa selaimessa. TypeScriptin käytöllä tai muilla JavaScriptin lisäosilla ei ole väliä. 3d-ympäristön esityksessä pelimoottorin tulee hyödyntää näkymiä (scene). Näkymän objektit muodostavat keskenään graafin, jolloin on mahdollista tietää, mitä näkymä pitää sisällään. Pelimoottorin tulee tukea äänen ja musiikin toistoa ilman kolmannen osapuolen kirjastoja, jolloin pelkkään grafiikkaan keskittyvät teknologiat, kuten Three.js eivät pääse mukaan tarkasteluun. Pelimoottorin lisenssin tulee myös sallia käyttö ilman maksusuorituksia.

BabylonJS [2017] on Microsoftin työntekijöiden vapaa-ajallaan rakentama 3d-pelimoottori, joka hyödyntää WebGL-teknologiaa. Pelimoottorin pääominaisuuksiin kuuluvat monipuolinen näkymägraafi, joka voi sisältää valoja, kameroita, materiaaleja, animaatioita ja ääniä. Pelimoottorissa on myös sisäänrakennettu tuki yksinkertaisille törmäyksille ja integraatio monimutkaisemmalle fysiikkamallinnukselle kolmannen osapuolen fysiikkamoottorien avulla. Myös animaatioiden ja partikkeleiden esitys onnistuu. Pelimoottori sisältää äänien ja musiikkien esitykseen soveltuvan toteutuksen.

BabylonJS sisältää suhteellisen kattavan dokumentaation pelimoottorin ominaisuuksista. Pelimoottorin verkkosivuilta löytyy myös useita ohjeita (tutorial), jotka entisestään selventävät käyttäjille erilaisia toimintoja ja näiden käyttöä. Näissä ohjeistuksissa on usein viittauksia BabylonJS Playground -sivustolle. Tämä on selaimessa toimiva BabylonJS-instanssi, jonka avulla on mahdollista muodostaa 3d-näkymiä. Näitä näkymiä on mahdollista jakaa muille käyttäjille suhteellisen vaivattomasti ja näin esimerkiksi ongelmien ratkaiseminen helpottuu. BabylonJS-verkkosivuilla olevalla foorumilla käydään keskusteluja pelimoottoriin ja pelinkehitykseen liittyvistä ongelmista. Pääkehittäjät vastaavat foorumilla aktiivisesti viesteihin ja kehitysehdotuksiin. Keskusteluissa on usein mukana Playground-linkki, jonka avulla useat käyttäjät voivat tutkia ja muokata samaa koodia. BabylonJS on avoimen lähdekoodin ohjelmistoprojekti, joka täten mahdollistaa muiden kehittäjien osallistumisen pelimoottorin kehittämiseen. Myös erilaisten julkisten pelimoottoriliitännäisten luominen on mahdollista.

BabylonJS tarjoaa valmiin pelimoottorin, mutta kaikki muu on kehittäjästä kiinni. Toista ääripäätä edustaa PlayCanvas-pelimoottori, joka ollut vuodesta 2014 avoimen lähdekoodin projekti. PlayCanvas sisältää myös monipuolisen valikoiman eri ominaisuuksia, jotka ovat osittain samoja kuin mitä BabylonJS tarjoaa. PlayCanvas sisältää myös erittäin monipuolisen näkymägraafin, johon voi liittää erilaisia komponentteja kuten valoja, kameroita, materiaaleja ja 3d-malleja. Pelimoottoriin on integroitu suoraan ammo.js-fysiikkamoottori, joka mahdollistaa monipuolisen törmäys- ja fysiikkalaskennan. PlayCanvas hyödyntää BabylonJS:n tapaan äänien ja musiikin toistoon soveltuvaa pelimoottorin sisäistä toteutusta, joka pystyy toistamaan 3d-tilassa ääniä sijaintikohtaisesti. PlayCanvas sisältää myös vahvasti Unityn kehitysalustaa muistuttavan skriptausjärjestelmän, joka taas puuttuu kokonaan BabylonJS-pelimoottorista. Tämä järjestelmä sallii skriptien kiinnittämisen erilaisiin näkymässä oleviin objekteihin. Skripti voi muuttaa objektin tilaa tai lisätä siihen jonkin ominaisuuden. Esimerkiksi näppäinsyötteillä liikkuva pallo voi olla yhden skriptin toteutus.

PlayCanvas sisältää BabylonJS:n tapaan kattavan dokumentaation pelimoottorin verkkosivuilla. Kehittäjät voivat kirjautua sivustolle sisään omilla tunnuksillaan ja nähdä muiden käyttäjien peliprojekteja sekä tarvittaessa osallistua näiden kehittämiseen. PlayCanvaksella on myös aktiivinen keskustelufoorumi, jossa on mahdollista pyytää apua pelimoottoria koskevissa ongelmissa. Kaikkein mielenkiintoisin ominaisuus, jonka PlayCanvas-ympäristö tarjoaa, on PlayCanvas-kehitysalusta (editor). Pelimoottoria on toki mahdollisuus käyttää ilman kehitysalustaa lataamalla lähdekooditiedostot erikseen. Tällöin pelimoottoria voidaan käyttää samaan tapaan kuin esimerkiksi BabylonJS-pelimoottoria. PlayCanvas-kehitysalusta käyttäytyy täysin eri tavalla ja sitä voisi verrata jopa Unity-pelimoottorin tarjoamaan kehitysalustaan. Kehittäjä voi tarkastella näkymiä suoraan selaimen kehitysalustasta ja lisätä näihin sisältöä hyödyntäen graafista käyttöliittymää. Luotuihin objekteihin on mahdollista lisätä skriptejä, jotka muuttavat objektien käyttäytymistä. Skriptejä on mahdollista muokata suoraan selaimesta, sillä PlayCanvas tarjoaa myös selainpohjaisen tekstieditorin. Peliohjelman suorittaminen onnistuu myös vain nappia painamalla ja tällöin peli ilmestyy uuteen selainikkunaan. Mikäli jotain peliin liittyvää skriptiä muokataan pelin suorituksen aikana, luodaan peliin liittyvät tiedostot uudestaan ja näin muutokset ovat heti havaittavissa. PlayCanvas-editori tarjoaa myös mahdollisuuden useamman henkilön samanaikaiseen näkymän muokkaamiseen. Esimerkiksi Unityn käyttämässä editorissa tämä ei ole mahdollista.

4.4 Hyödyt ja haitat

Erityisesti web-sovellusten kehittäjille JavaScript on erittäin tuttu ja turvallinen työkalu. Nykyään suurin osa selainpuolen tai näkymäkerroksen kehityksestä tapahtuu hyödyntäen JavaScriptin erinäisiä ohjelmistokehyksiä. JavaScriptin saapuminen myös palvelinpuolen maailmaan on saanut ohjelmistokehittäjät ohjelmoimaan JavaScriptillä. Tästä syntyykin massiivinen hyöty, sillä koko ohjelmisto on mahdollista ohjelmoida täysin JavaScriptiä hyödyntäen. Sama hyöty on saavutettaessa myös peliohjelmoinnissa ja erityisesti selainpohjaisten pelien ohjelmoinnissa. Esiteltyjen pelimoottorien avulla on mahdollista ohjelmoida peli hyödyntäen pelkkää JavaScriptiä. Sama ohjelmointikieli edesauttaa ongelmien ratkomisessa, kun projektin kaikki työntekijät voivat hyödyntää yhteistä lähdekoodia.

JavaScript voi myös tuoda ongelmia, sillä kieli sisältää useita kyseenalaisia ominaisuuksia, joista osa esiteltiin jo aikaisemmin. Vaikka kieli itsessään muistuttaa esimerkiksi Javan tai C:n syntaksia, tulee kehittäjien ymmärtää kielen ominaispiirteet ja mahdolliset sudenkuopat tarkasti. Lisäksi ongelmia voi tulla helposti erityisesti suurempia ohjelmistokokonaisuuksia rakentaessa, sillä JavaScript-maailmassa ohjelmistokehyksiä ja kirjastoja on lukematon määrä. Kehittäjien täytyy olla hyvin perillä siitä, mikä on tällä hetkellä se parhain ja käytetyin ratkaisu tiettyyn sovellusalueeseen. Vaikka JavaScript on kielenä erittäin lähestyttävä, saattavat kolmannen osapuolen kehykset tai kirjastot olla aivan päinvastaista ja tuoda esille melko monimutkaisia tapoja tehdä asioita.

Hiemankin laajemman JavaScript-sovelluksen kehittäminen saattaa olla melko haastavaa, koska kehittäjät joutuvat kokoamaan projektin hyödyntämällä useita kolmannen osapuolen toteutuksia. Näiden laatu voi myös vaihdella erittäin radikaalisti, eikä ongelmattomuutta voi taata edes suurien yritysten tuki. Päivitykset kolmannen osapuolen toteutuksiin saattavat rikkoa koodia, joka hyödyntää näitä. Tämä voi aiheuttaa projektiin lisäkustannuksia, kun kehittäjät yrittävät selvittää, mistä ongelma johtuu.

Myös selainpohjaisuus tuo itsessään ongelmia pelinkehitysprosessiin. Selainpohjaisten pelien ajoympäristönä toimii aina selain. Selaimia on kuitenkin lukuisia eri versioita ja vieläpä eri valmistajilta. Myös JavaScriptin suorittaminen näissä selaimissa voi olla ongelmallista, sillä selaimilla on myös erilaisia JavaScript-moottorin toteutuksia. Tämä voi hankaloittaa kehitysprosessia, mikäli saman ohjelmakoodin suorituksessa on selainkohtaisia eroja. Toisaalta tämä on mahdollista nähdä myös vahvuutena. Millään muulla ohjelmointikielellä ei ole näin montaa eri ajoympäristön tai komentotulkin toteutusta kuin JavaScriptillä. Tämä luo eräänlaista kilpailua selainkehittäjien

keskuuteen ja näin jokainen pyrkii olemaan parempi kuin kilpailijansa, kehittäen ja parantaen JavaScript-ympäristöjään.

Eri selaimien lisäksi käyttäjillä, eli pelaajilla, on myös käytössään erilaisia käyttöjärjestelmiä. Perinteistä työpöytäkäyttöä edustavat Windows-, Linux- ja OS X-käyttöjärjestelmät. Näille selainpohjaisten pelien kehittäminen ei teoriassa eroa paljoakaan työpöytäsovelluksista, sillä peleissä toiminnot tehdään yleensä hiirtä tai näppäimistöä käyttäen, tai joissain tapauksissa konsolipeleistä tuttua ohjainta hyödyntäen. Myös kuvasuhteet pysyvät suhteellisen samoina työpöytäkäyttöisissä selainpohjaisissa peleissä. Vaikeampaa on kuitenkin saada selainpohjainen peli toimimaan mobiili- tai tabletympäristössä, sillä nämä laitteet sisältävät useita eri käyttöjärjestelmiä, kuten Android ja iOS. Lisäksi käyttäjien suosimia selaimia on huomattavasti enemmän mobiiliympäristössä kuin vaikkapa työpöytäkäytössä. Tämä lisää entisestään ajoympäristön monimuotoisuutta, mikä täytyy huomioida kehitysprosessissa. Myös kuvasuhteet vaihtelevat mobiililaitteissa erityisesti vaaka- ja pystytilojen takia. Lisäksi mobiililaitteet ottavat kosketusnäytön takia erityyppisiä syötteitä kuin työpöydällä ajettavat selainpohjaiset pelit. Mobiililaitteissa on myös tuki laitteen asennon havaitsemiseen. Kaikki tämä tulee ottaa huomioon selainpohjaisen pelin toteuttamisessa, sillä esimerkiksi pöytätietokoneelle tarkoitettu peli ei välttämättä kovinkaan helposti sovellu suoraan mobiililaitteille.

5 Case-esimerkki selainpohjaisen pelin toteutuksesta

5.1 Pelin esittely ja vaatimusmäärittely

Vaatimusmäärittelyä laadittaessa hyödynnettiin pelisuunnitelmaa (game design document) pelin kuvauksessa ja vaatimusten määrittelyssä. Pelin tarina, hahmot, peliympäristö, pelattavuus, grafiikka, äänet ja musiikki sekä käyttöliittymä muodostavat yhdessä pelisuunnitelman karkean rakenteen. Suunnitelmaan ei ole kuitenkaan pakollista sisältyä aivan jokaista edellä mainituista kohdista, sillä jokainen peli on yksilöllinen. Pelisuunnitelma vastaa ohjelmistosuunnittelijan näkökulmasta pääasiassa toiminnallisia vaatimuksia.

Pelaaja ohjaa tulevaisuudessa elävää tutkimusmatkaajaa. Maapallon pinta on saastunut vuosisatoja sitten käyttökelvottomaksi. Ihmiskunta rakensi tällöin itselleen lentäviä alustoja, joiden päälle on ollut mahdollista pystyttää rakennuksia ja viljellä maata. Pelaajan asuttamassa lentoalustan moottorissa on kuitenkin ongelmia ja varaosia ei ole saatavilla. Pelaajan täytyy matkata keräämään puuttuvia osia muilta lentoalustoilta hyödyntäen kauko-ohjattavaa robottia. Kun pelaaja on saanut kaikki osat kerättyä, hän palaa omalle kotialustalleen ja estää sen putoamisen. Mikäli pelaaja ei saa kerättyä tarvittavia varaosia tietyssä ajassa, putoaa kotilentoalusta ja pelin joutuu aloittamaan alusta. Pelaaja kohtaa matkallaan myös muita alustoja, jotka ovat putoamisvaarassa. Muilla alustoilla on myös vihamielisiä otuksia, jotka yrittävät estää pelaajaa suorittamasta tehtäväänsä. Otukset voivat vahingoittaa pelaajaa kosketuksellaan, mutta nämä on mahdollista tuhota hyödyntäen räjähteitä. Toinen vaihtoehto on koettaa välttää otuksia kokonaan. Pelin idea pohjautuu etäisesti Bomberman-peliin [1983].

Peliympäristönä toimivat lentävät alustat, joiden koko ja muoto vaihtelevat. Alustat rakennetaan joko satunnaisesti hyödyntäen yksinkertaista algoritmia tai käsin jonkin mallin mukaisesti. Erityisesti satunnaisuus lisää pelin uudelleenpeluuarvoa, sillä jokainen pelikerta on erilainen. Pelikenttä muodostuu ruudukosta ja jokainen peliruutu voi sisältää esteen. Esteet ovat käytännössä laatikkoja, jotka sijaitsevat peliruutujen päällä. Nämä esteet ovat tuhottavissa erityisillä pommeilla, joita pelaaja voi asettaa lentotasolle. Kun pommi asetetaan lentotasolle, räjähtää se hetken päästä, tuhoten samalla osan lentotasosta. Kaikki esteet eivät kuitenkaan ole tuhottavissa. Jokaisella tasolla on myös rajattu määrä ruutuja, joissa pelaajan tarvitsemat varaosat saattavat sijaita. Varaosat on piilotettu esteiden sisään, joten pelaajan täytyy tuhota nämä edetäkseen pelissä. Kun pelaaja löytää vähintään yhden varaosan joltakin lentoalustalta, on hänen mahdollista siirtyä seuraavalle tasolle.

Pelin käynnistyessä pelaaja on päävalikossa, jossa hänelle esitellään tarvittavat pelikontrollit. Päävalikko sisältää vain yhden painikkeen, joka käynnistää itse pelin.

Pelaaja ohjaa pelihahmoaan näppäimistöä tai peliohjainta käyttäen. Peli on kuvattu kolmannesta persoonasta, eikä kameran kuvakulmaa ole mahdollista liikuttaa. Pelihahmo liikkuu kaksiulotteisesti joko vasemmalle tai oikealle, ylös tai alas. Pelihahmon muut toiminnot on rajattu pommien asettamiseen ympäri pelikenttää. Pelaajalla on myös mahdollisuus keskeyttää peli ja palata pelin päävalikkoon. Kun pelaaja on suorittanut kaikki pelin edellyttämät tehtävät, palaa hän päävalikkoon. Pelin pelaaminen on mahdollista lopettaa sulkemalla pelin välilehti selaimesta tai koko selainikkuna.

Peli sisältää yksinkertaisia 3d-malleja ja muuta lisenssimaksutonta grafiikkaa. Animaatiot toteutetaan pelin sisäisesti muuntamalla 3d-objektien kiertoa ja sijaintia. Peli sisältää myös muutamia partikkelitehosteita, esimerkiksi räjähdyksissä. Äänimaailma on melko yksinkertainen. Pelissä on toistuva musiikkiraita ja muutamia eri ääniefektejä muun muassa pelihahmon vahingoittumista sekä liikkumista ja räjähdyksiä varten. Käyttöliittymä on myös hyvin minimalistinen. Pelaaja näkee lentoalustan korjaamiseen tarvittavan varaosien kokonaismäärän ja tämänhetkisten varaosien määrän näytöllä. Ruudulla näkyvät myös pelaajan mahdolliset toiminnot räjähteiden jättämisestä pelin keskeyttämiseen.

Pelisuunnitelman lisäksi tulee huomioida erilaisia laadullisia vaatimuksia, jotka liittyvät pelin oikeellisuuteen, käytettävyyteen, suorituskykyyn, luotettavuuteen, siirrettävyyteen, uudelleenkäytettävyyteen ja ylläpidettävyyteen. Nämä vaatimukset määrittelevät erityisesti ohjelmiston laatua. Pelin tärkein laatuvaatimus on suorituskyky, sillä käyttäjät hylkäävät helposti huonosti optimoidun pelin. Toteutettavan pelin ruudunpäivitysnopeuden tulee pysyä tasaisena tapahtuipa pelissä mitä tahansa. Pelaajan tulee myös tietää, mitä toimintoja on mahdollista milloinkin hyödyntää. Täten käytettävyys on myös olennaisessa asemassa pelin toteutuksessa. Luotettavuus on myös tärkeä ominaisuus. Vaikka pelissä voi esimerkiksi satunnaisuuden takia tapahtua virhetilanteita, tulee pelin toipua näistä ilman, että pelin suoritus keskeytyy. Käytännössä siis jokin ohjelman osa-alue voi joutua poikkeustilaan, joka saattaa vaikuttaa pelaajan pelikokemukseen tai jäädä täysin huomaamattomaksi. On tärkeää kuitenkin huolehtia, että lentoalustan rakentamisen algoritmi toimii oikein, ettei pelihahmo esimerkiksi putoa lopulta tyhjyyteen.

Pelin pääasiallinen suoritusympäristö on web-selain. Koska selaimia ja mahdollisia käyttöjärjestelmiä on useita kymmeniä, rajoitetaan mahdollisten yhdistelmien lukumäärää. Pelin tulee olla pelattavissa alusta loppuun työpöytäselaimissa näppäimistöllisen tietokoneen avulla ainakin Windows- ja Mac-käyttöjärjestelmissä. Myös mobiiliympäristössä kuten tableteilla tai puhelimilla pelin tulee olla ainakin suoritettavissa. Vaikka näppäinsyötteitä ei luettaisikaan, tulee pelin 3d-ympäristön

toimia jollain siedettävällä tasolla myös mobiiliselaimilla. Mobiilikäyttöjärjestelminä toimivat Googlen Android ja Applen iOs. Pääasiallisena kehitysselaimena käytetään Google Chrome -selainta, sillä se on tällä hetkellä suosituin selain. Pelin tulee myös toimia Apple Safarilla, Microsoft Edgellä ja Mozilla Firefoxilla, jotka ovat myös hyvin suosittuja selaimia. Mobiiliympäristössä hyödynnämme samojen valmistajien selaimia. [StatCounter, 2016]

5.2 Tekninen toteutus

Peli toteutetaan täysin JavaScript-ohjelmointikielellä hyödyntäen ECMAScript 6 -standardia. Uusi standardi mahdollistaa muun muassa luokkasyntaksin käyttämisen pelin arkkitehtuurin suunnittelussa. Peli rakennetaan hyödyntäen moduuleja, jotka mahdollistavat ohjelmakoodin paloittelemisen järkeviin kokonaisuuksiin. Yksi moduuli käsittää käytännössä yhden JavaScript-luokan, joka taas voi hyödyntää muita moduuleja. Luokkamoduuli pitää sisällään pääasiassa attribuutteja, rakentajan sekä muita luokkakohtaisia funktioita.

Pelin toteuttamisessa hyödynnetään BabylonJS-pelimoottoria. Kyseinen pelimoottori valittiin case-esimerkin pääasialliseksi teknologiaksi, koska se sallii kehittäjälle paljon vapauksia tarjoten samalla jonkinlaisen ohjelmointikehyksen pelin ympärille. Toinen varteenotettava teknologia, PlayCanvas, vaatii ohjelmoijaa liiaksi toteuttamaan pelin tietyllä tavalla skriptausta hyödyntäen. Tällöin pelin rakenteen muodostaminen ei ole tarkasti ohjelmoijan käsissä. Täten hyödyntämällä BabylonJS-moottoria voidaan paremmin havainnoida, kuinka haastavaa pelin rakenteen muodostaminen on täysin tyhjästä.

BabylonJS-pelimoottori tarjoaa käytännössä piirtosilmukan, jonka avulla on mahdollista piirtää grafiikkaa HTML canvas -elementin sisällä olevalle alueelle. Kyseisen silmukan avulla on mahdollista myös siirtää ja kääntää näkymässä olevia 3d-objekteja, käytännössä siis animoida näiden liikkeitä. BabylonJS huolehtii myös itse näkymän objektien hallinnoimisesta. Äänimaailman luomisessa hyödynnetään BabylonJS-moottorin tarjoamaa äänimoottoria, joka tarjoaa työvälineitä musiikin ja äänien käsittelyyn.

Fysiikkamallinnusta ei pelissä hyödynnetä, sillä vaatimusmäärittelyssä ei tullut vastaan tilannetta, joka vaatisi tämän käyttöä. Lisäksi BabylonJS-pelimoottorin hyödyntämät fysiikkakirjastot hidastavat pelin ruudunpäivitysnopeutta huomattavasti. Pelihahmon liikkuminen pitäisi myös toteuttaa fysiikkamoottorilla sysäämällä hahmoa eri suuntiin. Myös pelihahmon rotaatiossa tulisi hyödyntää fysiikkamallinnusta, mikä monimutkaistaisi kehitysprosessia entisestään. Pelin liikkumisessa tullaankin

hyödyntämään BabylonJS-moottorin omaa törmäyksenlaskentaa ja yksinkertaista painovoimamallinnusta pelihahmon tippuessa lentoalustalta.

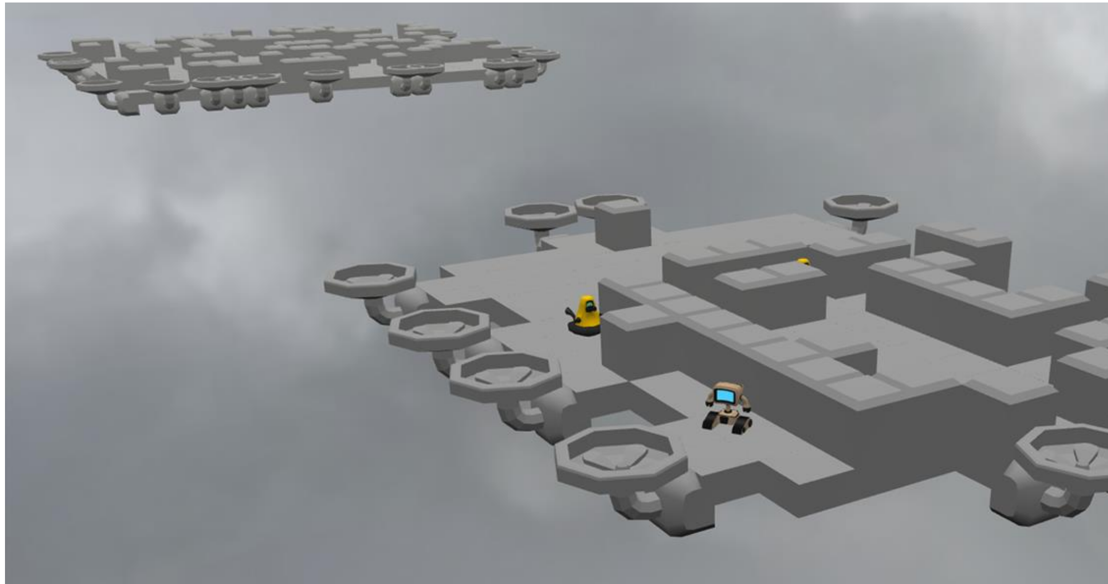
Yksi pelin avainominaisuus on satunnaisten pelitasojen, eli lentoalustojen, muodostaminen pelin suorituksen aikana. Pelissä tulee olla mahdollisuus luoda kyseisiä lentoalustoja käsin tai täysin satunnaisesti, hyödyntäen jonkinlaista mallia. Lentoalustoilla olevat pelaajia jahtaavat viholliset käyttävät kolmannen osapuolen toteuttamaa polunetsintää löytääkseen lyhyimmän polun pelaajan luokse. Tähän tarvitaan navigaatioverkko (navmesh), jonka muodostaminen tulee tapahtua dynaamisesti. Jos pelaaja tuhoaa pommeillaan lentoalustan lattiapalan, tulee vihollisten polku ja navigaatioverkko luoda uudestaan. Tämä voi kuitenkin olla varsin raskas operaatio, jota saattaa olla tarpeen optimoida.

Pelin toteuttaminen aloitettiin rakentamalla prototyyppejä BabylonJS-pelimoottorilla, jotta muodostuisi käsitys moottorin toiminnasta. Tämän jälkeen hahmoteltiin luokkia, jotka peli tulisi ainakin tarvitsemaan. Moduulipohjaisen JavaScript-sovelluksen rakentamiseen tarvitaan niputtaja (bundler), joka osaa muodostaa moduuleista ja näiden riippuvuussuhteista staattisen kokonaisuuden. Tämä kokonaisuus on käytännössä yksi JavaScript-tiedosto, joka sisältää kaiken pelin ajamiseen tarvittavan koodin. Peliprojektin niputtajaksi valittiin Webpack. Myös kolmannen osapuolen pakettien hallintaan tarvittiin työkalu. Tällä hetkellä näistä tunnetuin on npm (node packet manager), joka vaatii toimiakseen Node.js-ajoympäristön.

Projektipohjan ja pelin käynnistävän luokan luomisen jälkeen oli mahdollista siirtyä varsinaisen peliohjelmoinnin pariin. Alussa tutkittiin, mikä olisi parhain tapa tuoda peliin erilaista sisältöä, kuten 3d-malleja tai tekstuureja. BabylonJS hyödyntää omaa .babylon-tiedostoformaattiaan 3d-mallien määrityksessä. Käytännössä kaikki data kyseiseissä tiedostoissa on json-muodossa, eli JavaScript-objekteina. BabylonJS-laajennosten avulla on mahdollista muuntaa suoraan erilaisista tunnetuista 3d-tiedostoformaateista sopivia tiedostoja pelin ladatessa. Tämä voi kuitenkin aiheuttaa ongelmia esimerkiksi suorituskyvyn kanssa. Kaikki 3d-mallit päätettiin muuntaa suoraan 3d-mallinnusohjelmasta .babylon-formaattiin.

Kun tiedostojen tuonti peliin onnistui, alkoi itse pelilogiikan ohjelmointi. Aluksi rakennettiin yksinkertainen algoritmi, joka pystyi muodostamaan satunnaisesti lentotasoja kentälle. Lentotason palaset ovat tietyn kokoisia, joten nämä oli suhteellisen helppoa asetella ohjelmallisesti vierekkäin (kuva 3). Satunnaisten lentotasojen muodostamisen lisäksi peliin tehtiin mahdollisuus tuoda käsin tehtyjä lentotasoja peliin json-formaatissa, eli käytännössä JavaScript-objekteina. Näin oli helpompi testata pelin toiminnallisuutta ja luoda mielenkiintoisempia tasoja pelaajille. Lentotasoja

muodostaessa kävi ilmi, että satojen 3d-objektien luominen uudestaan on erittäin hidasta ja tämä myös hidastaa pelimoottoria huomattavasti. 3d-objekteista on mahdollista luoda myös instansseja, eli käytännössä täydellisiä klooneja alkuperäisestä 3d-objektista. Nämä kloonit jakavat alkuperäisen objektin geometrian, eikä näitä voi muuttaa. Tämä on kuitenkin ihanteellista esimerkiksi pelitason eli lentoalustan luomisessa, sillä käytämme uudestaan samoja rakennuspalikoita.



Kuva 3: Lentotasojen muodostaminen vierekkäin.

Pelin lentotasojen muodostamisen jälkeen toteutettiin itse pelihahmo ja tämän yksinkertaisten näppäinsyötteiden hallinta. Näppäinsyötteiden käsittelyn jälkeen pelihahmo laitettiin liikkumaan kentällä. Liikkumisessa hyödynnetään BabylonJS-moottorin törmäyksenlaskentaa, joka myös pitää pelihahmon lentotason pinnalla ja estää tämän uppoamisen seinien sisään. Pelihahmolle lisättiin myös mahdollisuus jättää pommeja ja kerätä varaosia, jotka syntyvät räjähdyksien yhteydessä.

Seuraavaksi alettiin toteuttaa pelaajaa seuraavia vihollisia. Tässä hyödynnettiin kolmannen osapuolen kirjastoa navigaatioverkon muodostamisessa. Kirjasto hoitaa reitin laskemisen, mikäli sille syötetään jonkin liikuttavan alueen 3d-malli. Käytännössä tämä malli on rakennettava dynaamisesti aina kentän muuttuessa, esimerkiksi pommin räjähdyksen seurauksena. Viholliset seuraavat pelaajaa, mikäli tämä on navigaatioverkon avulla saavutettavissa. Jo alussa havaittiin, että navigaatioverkon päivitys on suhteellisen raskas operaatio.

Vihollisten toteuttamisen jälkeen käytettiin hieman aikaa lentotasoja muodostavan algoritmin paranteluun. Useampien lentotasojen samanaikainen esittäminen on

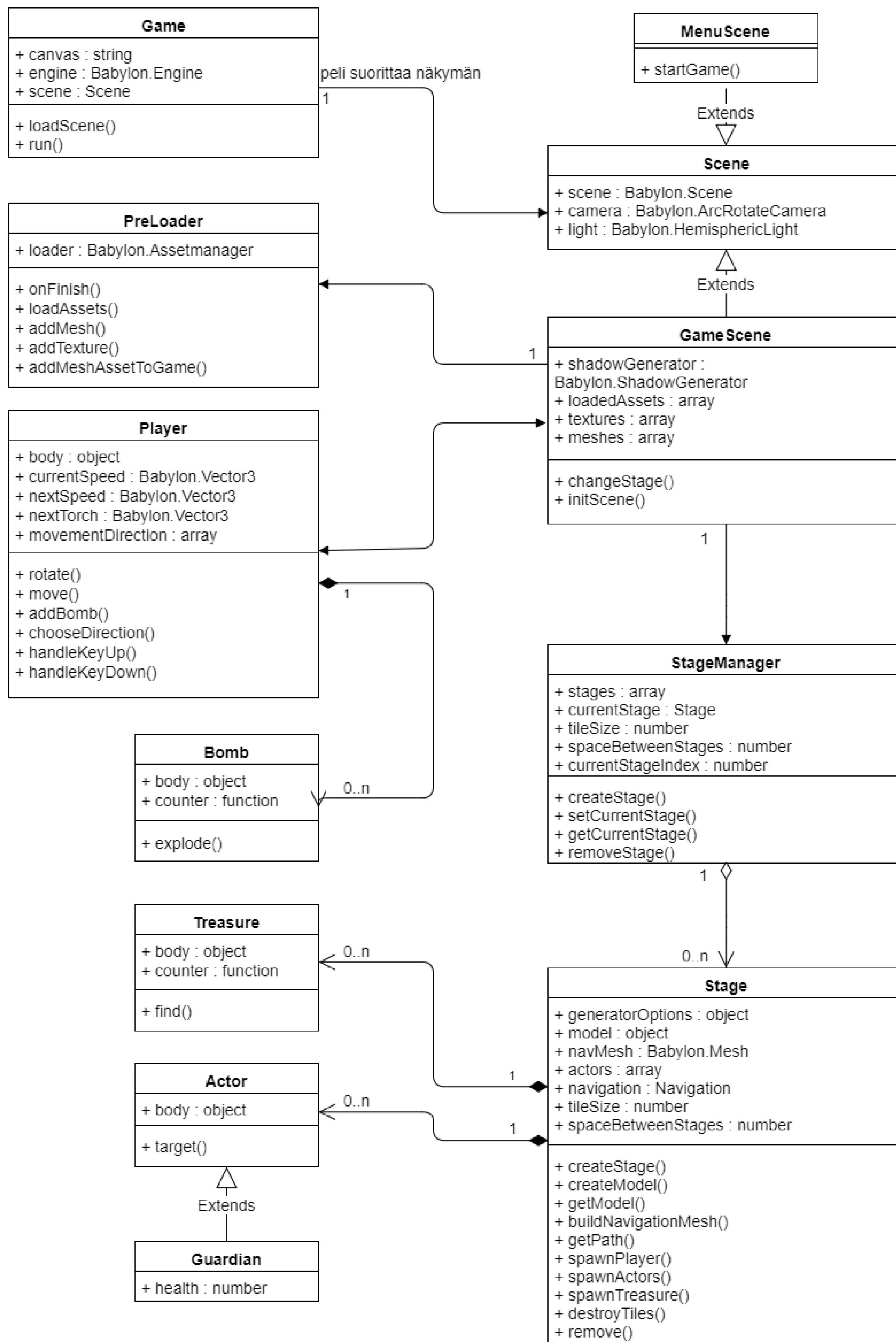
suhteellisen raskas prosessi, joten näitä on tarvittaessa ladattava dynaamisesti. Lisäksi tasoille, joilla ei ole vihollisia, ei tarvitse rakentaa navigaatioverkkoa ollenkaan.

Seuraavaksi päätettiin tehdä itse peli pelattavaksi. Alussa luodaan muutama lentotaso valmiiksi. Pelaaja aloittaa näistä ensimmäiseltä ja siirtyy tasolta toiselle kerättyään tietyn määrän varaosia jokaiselta lentotasolta. Peli päättyy, kun varaosia on kerätty tarvittava määrä.

Kun pelistä oli muodostunut pelattava kokonaisuus, kiinnitettiin huomio käyttöliittymään ja äänimaailman muodostamiseen. Pelin käyttöliittymän rakentaminen aiheutti ongelmia huonojen vaihtoehtojen vuoksi. Aikaisemmin kolmannen osapuolen BabylonJS-käyttöliittymäkirjasto oli vanhentunut ja uusi kirjasto oli vasta tuloillaan. Koska pelissä on varsin yksinkertainen käyttöliittymä, päädyttiin se rakentamaan ulos BabylonJS-canvakselta. Käyttöliittymäkomponentit toteutettiin vue.js-kirjastolla, joka on tunnettu front-end-kirjasto. Käytännössä käyttöliittymä rakennettiin canvas-elementin päälle käyttäen tavanomaista HTML-syntaksia.

Lopuksi pelin ulkoasullisia ominaisuuksia paranneltiin ja pelialustojen osasille lisättiin muun muassa varjostuksia. Vaikka kyseiset toiminnallisuudet eivät sinänsä olleet pelin toiminnan kannalta pakollisia, oli tarpeellista kokeilla, kuinka helposti nämä olisi mahdollista liittää osaksi peliä. Myös mahdolliset vaikutukset pelin suorituskykyyn oli hyvä havainnoida, sillä tämä on yksi tärkeimmistä pelin laatuun vaikuttavista tekijöistä.

Pelin rakenne on esitetty UML-kaaviona kaaviossa 1. Kaaviosta voidaan havaita, että pelin rakenne on melko yksinkertainen. Kaikki attribuutit ovat pelissä julkisia, sillä JavaScriptin luokkamäärittelyssä on hankala määritellä yksityisiä muuttujia tai funktioita muuten kuin jättämällä ne pois itse luokkamäärittelystä. Yksinkertaistamisen vuoksi luokkakaaviosta puuttuu myös käyttöliittymän ja äänen hallintaan erikoistuneet luokat. Lisäksi melkein jokaisella jollain tasolla GameScene-luokkaan liittyvällä luokalla on viittaus GameScene-olioon, eikä tätä ole selkeyden vuoksi lisätty UML-kaavioon. Viittaus on tarpeellinen, sillä sen avulla on saatavilla viite itse BabylonJS-pelinäkymään, joka on taas välttämätön esimerkiksi lisättäessä tai poistettaessa peliobjekteja.



Kaavio 1: UML-kaavio pelin luokkarakenteesta.

Koodikatkelma 3 sisältää pelitoteutuksen Game-luokan. Game-luokka on pelin käynnistävä luokka, jonka päätehtävä on hallinnoida pelin eri näkymiä ja ylläpitää pääpiirtosilmukkaa. Luokka sisältää käytännössä rakentajan ja erilaisia funktioita. Rakentajassa HTML-dokumentista haetaan tarvittava canvas-elementti, sillä pelimoottorin instanssi tarvitsee tähän viitteen, ja näkymälle asetetaan toistaiseksi null-arvo. Nyt on mahdollista ladata näkymiä peliin loadScene-funktiolla ja käynnistää pelin piirtosilmukka run-funktiolla. Näkymien latausfunktiossa alustetaan GameScene- ja MenuScene-tyyppiset luokkaoliot, jotka periytyvät Scene-luokasta. MenuScene-luokka sisältää yksinkertaisen valikon, josta on mahdollista käynnistää peli. Käytännössä siis käynnistyspainike kutsuu Game-luokan loadScene-funktiota tietyllä määritellyllä parametrilla.

```
import Babylon from 'babylonjs'
import GameScene from './scenes/GameScene'
import MenuScene from './scenes/MenuScene'

export default class Game {
  constructor(canvasName) {
    this.canvas = document.getElementById(canvasName)
    this.engine = new Babylon.Engine(this.canvas, true)
    this.scene = null
  }

  static SCENES = {
    'GAME': 1,
    'MENU': 2
  };

  loadScene(sceneName) {
    if (this.scene) {
      this.scene.scene.dispose()
      this.scene = null
    }

    if (sceneName === Game.SCENES.MENU) {
      this.scene = new MenuScene(this.canvas, this.engine)
    } else if (sceneName === Game.SCENES.GAME) {
      this.scene = new GameScene(this.canvas, this.engine)
    }

    return this
  }

  run() {
    if (this.scene) {
      this.scene.scene.executeWhenReady(() => {
        this.engine.runRenderLoop(() => {
          this.scene.scene.render()
        })
      })
```

```
      window.addEventListener('resize', () => {
        this.engine.resize()
      })
    }

    return this
  }
}
```

Koodikatkelma 3: Pelin käynnistävä luokka.

Pelin logiikan pääasiallinen toteutus sijaitsee GameScene-luokassa (koodikatkelma 4), joka periytyy Scene-luokasta ja käytännössä alustaa pelin päänäkymän. Alustamisessa hyödynnetään useita apuluokkia, jotka tuodaan luokkaan import-lauseiden avulla. GameScene-luokan rakentajassa määritellään aluksi yleisiä näkymän attribuutteja, kuten painovoiman hallinta, törmäyksien käsittely, sumu ja varjot. Tämän jälkeen PreLoader-luokka lataa 3d-mallit, tekstuurit ja äänitiedostot GameScene-luokan käyttöön. PreLoader-luokka hyödyntää BabylonJS-pelimoottorin AssetsManager-luokkaa, joka huolehtii asynkronisesta tiedostojen lataamisesta. PreLoader-luokka ottaa vastaan myös callback-funktion ja pelinäkymän kontekstin onFinish-funktiossa. Callback-funktioksi asetetaan GameScene-luokan initScene-funktio, joka huolehtii varsinaisen pelilogiikan alustamisesta. Päänäkymän changeStage-funktiota käytetään alustamaan uusi lentotaso ja poistamaan vanha käytöstä, mikäli pelaaja haluaa siirtyä seuraavalle tasolle. Teoriassa olisi mahdollista säilyttää vanhat lentotasot muistissa mallin avulla, ilman fyysisiä 3d-malleja. Tällöin pelaaja voisi palata edeltäville tasoille.

```
import Babylon from 'babylonjs'
import _ from 'lodash'

import Scene from './Scene'
import StageManager from '../scripts/stage/StageManager'
import Player from '../scripts/player/Player'
import Guardian from '../scripts/actor/Guardian'
import PreLoader from '../helpers/Preloader'

export default class GameScene extends Scene {
  constructor(canvas, engine) {
    super(canvas, engine)

    /* Gravity and collisions */
    this.scene.gravity = new Babylon.Vector3(0, -1, 0)
    this.scene.collisionsEnabled = true

    /* Fog */
    this.scene.fogMode = Babylon.Scene.FOGMODE_EXP
    this.scene.fogColor = new Babylon.Color3(0.9, 0.9, 0.85)
    this.scene.fogDensity = 0.0001
```

```
    /* Shadows */
    this.shadowGenerator = new Babylon.ShadowGenerator(1024,
this.directionalLight)
    this.shadowGenerator.useBlurExponentialShadowMap = true

    this.scene.actionManager = new Babylon.ActionManager(this.scene)

    /* Init assets */
    this.loadedAssets = {}
    this.meshes = ['game']
    this.textures = []

    /* Static levels for demoing purposes */
    this.demo = ['basic-1', 'basic-2']

    new PreLoader(this)
      .loadAssets(this.meshes, this.textures)
      .onFinish(this.initScene, this)

    /* Game state */
    this.health = 100
    this.scrap = 0
    this.scrapGoal = _.sample([5, 10, 15])

    this.state = this.scene.onBeforeRenderObservable.add(() => {
      if (this.scrap >= this.scrapGoal) {
        console.log('YOU WIN')
      }
    })
  }

  changeStage() {
    const demostage = this.demo.length > 0 ? this.demo.shift() :
undefined

    this.stage = demostage
      ? this.stageManager.createStage(demostage)
      : this.stageManager.createStage()
    this.stage.spawnActors()
    this.stage.spawnPlayer()

    this.directionalLight.position = this.player.body.position

    this.stageManager.removeStage(0)
  }

  initScene() {
    try {
      this.stageManager = new StageManager(this)
      this.stage = this.stageManager.createStage('start')
      this.stage.spawnActors()

      /* Init player */
```

```
      this.player = new Player(this)
      this.stage.spawnPlayer()
    }
    catch(e) {
      console.log(e)
    }
  }
}
```

Koodikatkelma 4: Pelin päänäkymä.

Kun kaikki tarvittavat resurssit on ladattu, rakennetaan itse peli pelattavaksi kokonaisuudeksi. StageManager-luokka luo ja poistaa lentotasoja dynaamisesti. Aktiivisena olevista lentotasoista pidetään kirjaa stages-taulukossa. Lentotasot on varastoitu Stage-luokkaan, joka sisältää lentoalustan rakentamiseen tarvittavan mallin. Malli sisältää lentotason nimen, aloitus- ja lopetusruudun koordinaatit ja mahdolliset viholliset. Myös itse lentotason rakentamiseen tarvittavat lattia- ja seinäpalaset sekä lentotason ilmassa pitävät potkurit löytyvät mallista. Mallit rakennetaan joko satunnaisesti algoritmin avulla tai manuaalisesti. Manuaaliseen määrittelyyn käytetään hyväksi erillistä tiedostoa, johon kaikki mahdolliset käsin kirjoitetut tasot on koostettu. Kun lentotason rakenne on luotu, rakennetaan lentotasolle navigaatioverkko. Tämä on käytännössä 3d-malli, jonka päällä mahdolliset pelaajaa jahtaavat viholliset voivat liikkua. Mikäli lentotaso ei sisällä yhtäkään vihollista, navigaatioverkkoa ei tarvitse luoda. Navigaatioverkon luomisen jälkeen alustetaan tason mahdolliset viholliset ja asetetaan ne seuraamaan pelaajaa tietyn ajan kuluttua tämän saapumisesta tasolle. Viholliset periytyvät Actor-luokasta.

Vihollisten alustamisen jälkeen alustetaan itse pelaajan ohjaama pelihahmo, robotti, pelikentälle lentotason aloitusruutuun. Pelihahmo on koostettu Player-luokkaan, joka ylläpitää käytännössä tietoa pelaajan liikkumisesta nuolinäppäimien avulla. Pelin pääkamera on myös asetettu seuraamaan pelaajan liikkeen mukana.

Pelihahmon alustamisen jälkeen peli on valmis pelattavaksi. Pelaaja voi liikkua ympäri tasoa ja jättää pommeja. Pommit räjäyttävät lentotason lattia- ja seinäpaloja pois, jolloin navigaatioverkko lasketaan uudestaan. Seinän räjäyttäminen voi mahdollisesti synnyttää pelaajan kerättäviä varaosia. Kun pelaaja on löytänyt yhdenkin varaosan, avataan lentoalustan lopetusruutu. Kun pelaaja siirtyy lopetusruutuun, alustetaan seuraava lentotaso ja tämän mahdolliset viholliset. Pelaajan siirryttyä uudelle lentotasolle poistaa StageManager-luokka vanhan tason käytöstä. Mikäli pelaaja on kerännyt tarvittavan määrän varaosia, ei seuraavaa tasoa luoda, vaan pelaaja on voittanut pelin ja siirtyy takaisin päävalikkoon.

Pelaaja voi myös epäonnistua suorittamaan keräystehtävän joko putoamalla alustalta tai joutumalla vihollisen kiinniottamaksi. Tällöin peli keskeytyy ja palataan takaisin päävalikkoon.

Koodikatkelma 5 on yksinkertainen esimerkki modulaarisesta pommi-luokasta. Pommi on tässä tapauksessa olio, joka sisältää vain kolme attribuuttia. Body-attribuutti on pommin fyysinen esitys pelikentällä, eli käytännössä 3d-malli. Game-attribuutti taas on referenssi GameScene-luokkaolioon. Counter-attribuutti on taas BabylonJS-pelimoottorin Observable-mallin laskurifunktio. Käytännössä laskuri rekisteröidään suoritettavaksi ennen jokaista piirtosilmukkaa. Laskuri pitää kirjaa kuluneesta ajasta ja myös muuttaa pommin kokoa suuremmasta pienempään aaltomaisesti. Pommi räjähtää, kun sen luomisesta on kulunut kolme sekuntia. Tällöin piirretään säde pommin ja lentotason jonkin pohjapalasen väliin. Näin on mahdollista saada selville tietyn palasen sijainti ja muuttaa lentotason rakennetta, poistaen siitä osia. Lopuksi pommin kaikki referenssit tuhotaan, jotta ei synny muistivuotoa.

```
import Babylon from 'babylonjs'

export default class Bomb {
  constructor(game, mesh, position) {
    this.body = mesh.createInstance('Bomb');
    this.body.setEnabled(true);
    this.body.position = new Babylon.Vector3(position.x, position.y,
position.z);
    this.game = game;

    let timeInSeconds = 0;

    this.counter = this.game.scene.onBeforeRenderObservable.add(()
=> {
      timeInSeconds += 0.001 *
this.game.scene.getEngine().getDeltaTime();
      const amplitude = 0.1;
      const speed = 1;
      const scaling = amplitude * Math.sin(timeInSeconds * 2 *
Math.PI * speed - Math.PI/2) + 1;

      this.body.scaling.x = scaling;
      this.body.scaling.y = scaling;
      this.body.scaling.z = scaling;

      if (timeInSeconds >= 3) {
        this.explode();
      }
    });
  }
```

```
  explode() {
    const rayCurrentTile = new Babylon.Ray(
      new Babylon.Vector3(this.body.position.x,
this.body.position.y, this.body.position.z),
      new Babylon.Vector3(0, -1, 0));
    const result = this.game.scene.pickWithRay(rayCurrentTile);

    if (result.hit && result.pickedMesh.modelPosition) {
      this.game.stage.destroyTiles(result.pickedMesh.modelPosition);
    }

    this.body.dispose();
    this.game.scene.onBeforeRenderObservable.remove(this.counter);
    delete this;
 }
}
```

Koodikatkelma 5: Modulaarinen pommi-luokka.

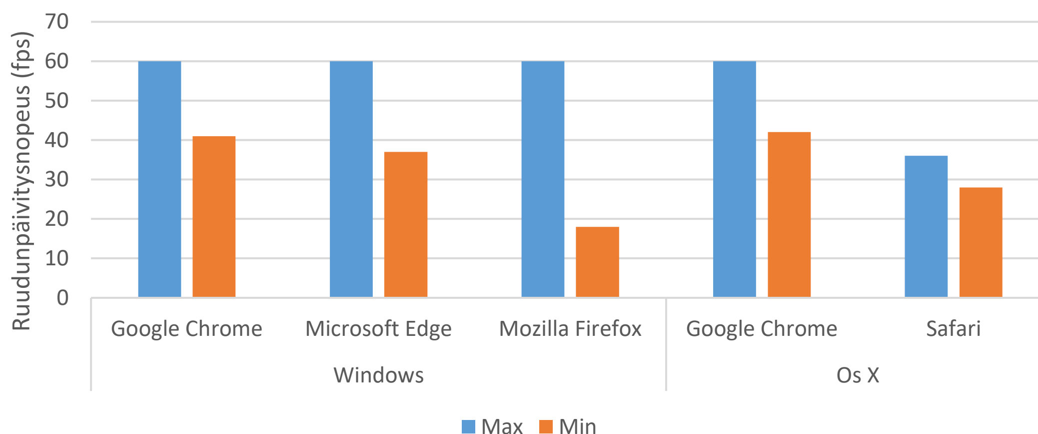
Pelin lähdekoodi on saatavilla Github-palvelussa [Hokkanen, 2017].

5.3 Toteutuksen arviointi

Case-esimerkin kautta toteutettu peli on aikaisemmin esitellyn tutkimuskehyksen mukaisen tutkimuksen toteutus. Tutkimuksen aikana on hyödynnetty useita tutkimusaktiviteetteja. Pelin rakentaminen kuvattiin tarkasti aikaisemmissa luvuissa. Tässä kohdassa keskitytään case-esimerkin toteutuksen arviointiin, teoretisointiin ja oikeuttamiseen. Pelin toteutuksen arvioinnissa hyödynnetään aikaisemmin esitellyn ISO 25010 -standardin mukaisia ohjelmistojen laadun kriteerejä ja vaatimuksia. Tärkeimmät kriteerit käsittelevät pelin suorituskykyä, käytettävyyttä ja luotettavuutta, sillä niillä on suora yhteys siihen, että pelin pelaaminen on hauskaa. Toisekseen pelin kehitystä arvioidaan myös ohjelmistokehittäjän näkökulmasta. Arvioinnissa käydään läpi valitun teknologian, kehyksien ja kirjastojen hyviä ja huonoja puolia.

Pelin hauskuuden kannalta ehkä tärkein ominaisuus on pelin suorituskyky. Pelin suorituskykyä mitattiin useilla tavoilla. Aluksi valittiin mitattavat käyttöjärjestelmät. Nämä olivat lopulta Windows, Os X, Android ja iOs. Laitteistoiksi valikoituivat pöytätietokone, kannettava tietokone, tablet-tietokone ja älypuhelin. Mittausta varten jokaiselle laitteisto- ja käyttöjärjestelmäyhdistelmälle valittiin useampi selain, jotka ovat yleisesti käytössä. Näitä ovat muun muassa Google Chrome, Internet Explorer, Mozilla Firefox ja Apple Safari. Pelin suorituskyvyn mittarina toimi pelimoottorin ruudunpäivitysnopeus (frames per second) pelimaailman luomisen jälkeen. Suorituskyvyn mittauksessa otettiin huomioon myös pelissä esiintyvät varjostukset ja tekstuurit, joilla on potentiaalia laskea kuvataajuutta.

Kuvassa 4 on esitetty mitatut ruudunpäivitysnopeudet korkeimmillaan ja alhaisimmillaan työpöytäselaimissa kahdella eri käyttöjärjestelmällä. Ruudunpäivitysnopeus on parhaimmillaan työpöytäselaimilla. Tämä johtuu muun muassa modernista näytönohjaimesta, joka pystyy ajamaan raskaitakin pelisovelluksia. MacBook-pro kannettavalla tietokoneella taas Google Chrome suorittaa peliä paremmin kuin Applen oma Safari-selain. Huomattavaa on, että MacBookissa on Retina-näyttö, jolloin pikselitiheys on kaksinkertainen normaaliin näyttöön verrattuna. Selainikkuna pienennettiin täten työpöytäselaimen kokoa vastaamaan, sillä muuten resoluutio olisi ollut melko valtaisa. BabylonJS ei osaa pelissä hyödynnetyssä moottorin versiossa 2.5 ottaa huomioon pikselitiheyttä. Työpöydällä pelattuna peli toimii kohtuullisen luotettavasti. Peli suoriutuu useimmissa työpöytäselaimissa pääosin päivittämällä ruutua 60 kertaa sekunnissa, mikä on varsin riittävä useimpiin peleihin. Yleisesti pelit tuntuvat sulavilta, kun ruudunpäivitysnopeus on ainakin 30. Case-esimerkin pelissä kuitenkin joillakin työpöytäselaimilla oli havaittavissa ajoittain erittäin alhaisia päivitysnopeuksia. Nämä ajoittuivat yleensä hetkeen, jolloin pommi räjähti kentällä tai jotain muuta raskaampaa laskentaa tapahtui. Tablet- ja puhelinkäytössä ruudunpäivitys on erittäin surkea. Koska case-esimerkkiä ei voi varsinaisesti pelata puhelimella tai tabletilla, arvioitiin suorituskykyä vain avaamalla peli selaimessa. Tällöin ruudunpäivitysnopeus vaihteli 10-20 välillä, mikä ei ole alkuunkaan hyväksyttävissä rajoissa. Todennäköisesti tämä laskisi entisestään, mikäli peliä voisi pelata ja esimerkiksi pudottaa pommin maahan.



Kuva 4: Toteutetun pelin ruudunpäivitysnopeudet työpöytäselaimilla.

Vaikka toteutettu peli toimii monessa eri laitteessa, on erittäin hankala taata samanlainen pelikokemus tai yleensäkin se, että peli näyttää ja toimii samalla tavalla. Hidastempoisissa peleissä, kuten strategiapeleissä tai vuoropohjaisissa peleissä, suorituskyvyn ja ruudunpäivityksen heitteleminen ei ole kovin kriittistä. Nopeutta ja tarkkuutta vaativissa peleissä taas ruudunpäivityksen on oltava tarkka, tai käyttäjä

turhautuu helposti. Case-esimerkissä toteutetussa pelissä käyttäjältä odotetaan suhteellisen nopeita päätöksiä pelihahmon ohjaamisessa ja ruudunpäivitysnopeuden äkilliset muutokset voivat aiheuttaa erinäisiä ongelmia, esimerkiksi pelihahmon putoamisen tasolta.

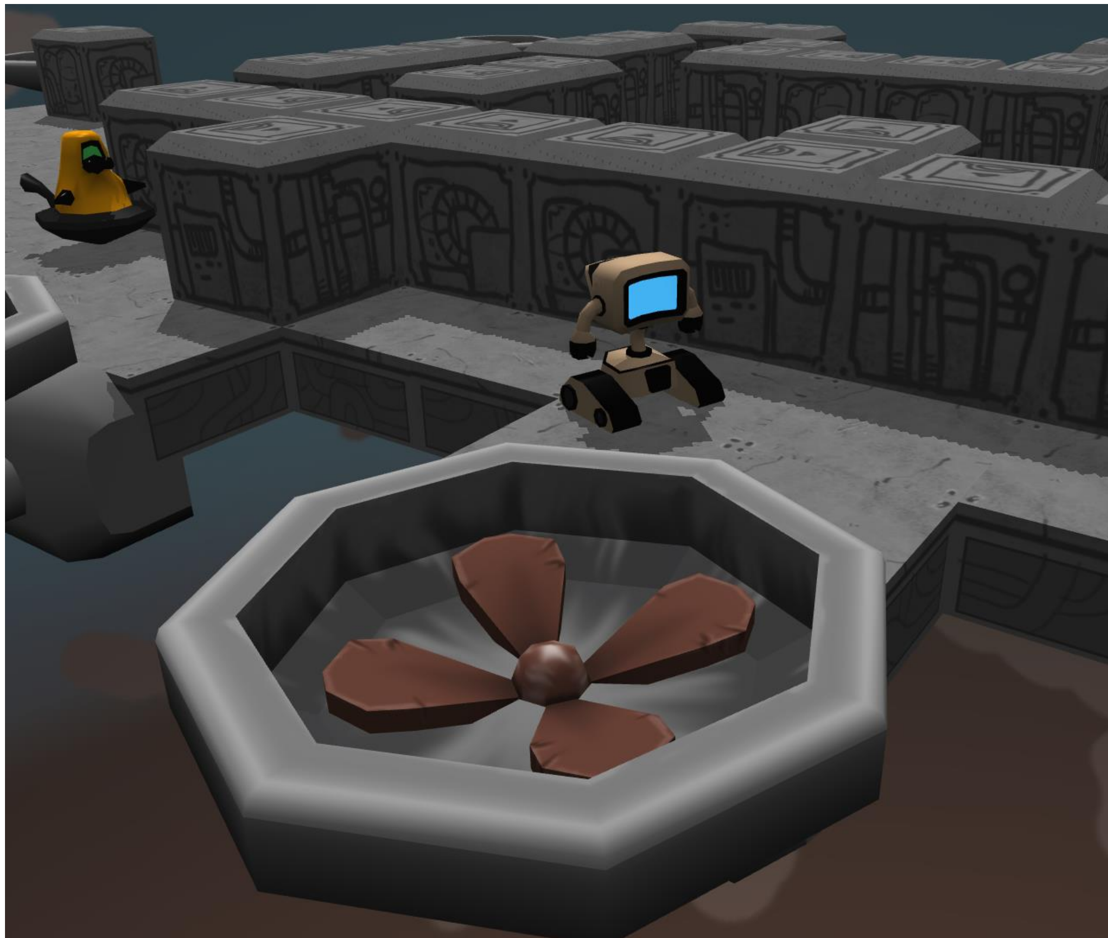
Ruudunpäivitysarvojen minimi- ja maksimiarvot vaihtelevat kohtuullisen paljon erityisesti työpöytätietokoneiden ja kannettavien tietokoneiden selaimissa. WebGL on suhteellisen uusi teknologia ja se näkyy erityisesti pelimoottorin suorituskyvyssä. BabylonJS-pelimoottorin käytön yhteydessä täytyy huomioida, että jo pienikin määrä yksinkertaisia 3d-objekteja, vaikkapa tasoja, saattaa aiheuttaa suorituskyvyllisiä ongelmia. Tässä tapauksessa identtiset 3d-objektit tulee yhdistää BabylonJS-moottorissa hyödyntäen createInstance-funktiota, joka luo objektista kopion. Näitä instanssikopioita ei voi käytännössä enää muokata muuten kuin niiden sijainnin osalta. Jatkuva tarve optimoida peliä on kuitenkin yleinen ongelma erityisesti selainpohjaisten pelien kehitysprosessissa. Yleisesti ottaen ennenaikaista optimointia pidetään turhana, mutta WebGL-peleissä sille voi olla tarvetta. Pelit saattavat muuten suorittua erittäin alhaisella ruudunpäivitysnopeudella. Yhtenäinen ruudunpäivitysnopeus on hankala toteuttaa selainpohjaisissa peleissä. Ongelma korostuu erityisesti johtuen erilaisista laitekokoonpanoista, joita on useita tietokonemaailmassa. Esimerkiksi MacBook-tietokoneiden Retina-näyttö esittää kuvan käytännössä kaksinkertaisena tavallisiin näyttöihin verrattuna. Tähän ongelmaan tulee reagoida pienentämällä esitettävää pelialuetta puolella ja skaalaamalla se kuitenkin koko näytön kokoiseksi.

BabylonJS ei sisällä fysiikkamoottoria, vaan mahdollisuuden käyttää erillistä kolmannen osapuolen ratkaisua. Käytännössä siis fysiikkamoottori ladataan globaaliin nimiavaruuteen, jolloin sitä voidaan hyödyntää näkymissä. Fysiikkamoottori täytyy erikseen aktivoida käyttöön. Tällä hetkellä fysiikkamoottoreiksi voi valita kaksi mahdollista ratkaisua. Ensimmäinen on Cannon.js, joka on kirjoitettu täysin JavaScriptiä hyödyntäen. Toinen ratkaisu on Oimo.js, ActionScript3-pohjalta käännetty JavaScript-fysiikkamoottori. Näistä Cannon.js tukee useampia ominaisuuksia, mutta on jonkin verran Oimo.js-moottoria hitaampi. Kaiken kaikkiaan fysiikkalaskennan tuominen pelinäkymään hidastaa suorituskykyä huomattavasti. Jo muutaman hieman monimutkaisemman 3d-mallin sisällyttäminen fysiikkoineen saa aikaan pelin huomattavan hidastumisen. Fysiikkamoottorit tuovat myös muita haasteita itse pelinkehitysprosessiin. Esimerkiksi pelihahmon liikuttaminen on toteutettava käyttäen fysiikkamallinnusta antamalla tälle työntövoima tiettyyn suuntaan, mikä on huomattavasti haastavampaa kuin vain muuttaa pelihahmon sijaintia. Samalla täytyy myös muistaa kääntää hahmoa hyödyntäen fysiikkamoottoria. Näistä syistä johtuen case-esimerkissä ei ole käytössä fysiikkamallinnusta. Ilman fysiikoita pelimoottori laskee vain yksinkertaisia törmäyksiä ja hahmoa voidaan kääntää ja liikuttaa helposti.

Yksinkertaista painovoimaa on myös mahdollista simuloida ilman fysiikkamoottoria ja täten esimerkiksi pelihahmon putoamista pelialueelta. Yleisesti ottaen vaikuttaa siltä, että BabylonJS-pelimoottoria hyödyntävien fysiikkapohjaisten pelien täytyy olla erittäin yksinkertaisia toimiakseen luotettavasti selaimessa. Vähänkin suuremmat pelit tulevat todennäköisesti kaatumaan suorituskykyongelmiin.

Reitinlaskennasta tuli myös suorituskyvyllinen ongelma, joka on edelleen ratkaisematta case-esimerkissä. Lentotasolle määritetään sen luomisen yhteydessä erillinen navigaatioverkko, jota pitkin viholliset osaavat liikkua. Mikäli pelaaja räjäyttää seinämää tai lattiaa, lasketaan koko navigaatioverkko uudelleen. Uuden navigaatioverkon muodostaminen on kuitenkin erittäin raskas prosessi, joka aiheuttaa hetkittäisiä piikkejä pelin ruudunpäivityksessä jopa työpöytätietokoneilla, joiden pitäisi sallia vaativampienkin laskutoimitusten suorittaminen. JavaScript on kuitenkin kielenä yksisäikeinen (single-threaded), mikä tekee monimutkaisten operaatioiden suorittamisesta ongelmallista. Yksi mahdollinen ratkaisu on hyödyntää web-työläisiä (web-worker), joiden avulla on mahdollista suorittaa koodia eri säikeissä. Tämä teknologia on kuitenkin verrattain uutta, eikä tukea ole saatavilla moniin selaimiin, ei esimerkiksi Microsoft Edgeen. Lisäksi web-työläisten käyttö on melko monimutkaista, sillä suoritettava koodi on eristettävä täysin muusta koodipohjasta. Toinen tapa on laskea vain osa navigaatioverkosta uudestaan. Tämä kuitenkin vaatii erityistä tietämystä reitinlaskennasta ja 3d-mallien rakentamisesta.

Toteutettu peli ladataan pelaajan käyttöön kuin mikä tahansa muukin verkkosivusto. Taustalla on oltava jokin web-palvelin, joka osaa tarjoilla staattista sisältöä. Peli latautuu erittäin nopeasti pelattavaan tilaan. Jokaisella tutkittavalla laitteella peli oli käyttövalmis alle viidessä sekunnissa. Pelin sisältävällä web-sivustolla ladataan CSS-tyylit, minifioitu JavaScript-lähdekoodi, 3d-mallit pakattuna babylon-tiedostoformaattiin sekä kuvatiedostot, jotka toimivat lähinnä pelin 3d-mallien tekstuureina. Kaikkinensa pelin näyttämiseksi ladataan hieman yli 6 megatavua tietoa, mikä on melko vähän nykystandardeihin nähden. Laajempaa peliä toteuttaessa 3d-malleja voi kuitenkin olla huomattavasti enemmän kuin case-esimerkin pelissä ja täten esimerkiksi tekstuurien lataus voi tulla ongelmalliseksi. Kuvatiedostot vievät tällöin suurimman osan latausajasta, jolloin ne täytyy jollain tavalla yhdistää tai pakata pienemmiksi. Kuvassa 5 on esimerkki pelin viimeistellymmästä ulkoasusta.



Kuva 5: Ruudunkaappaus case-esimerkin viimeistellymmästä ulkoasusta.

Käytettävyys on toinen tärkeä kriteeri pelin laadun arvioinnissa. Käyttöliittymän luominen BabylonJS-pelimoottorilla on haasteellista. Moottori ja dokumentaatio eivät itsessään kannusta kehittäjää mihinkään suuntaan ja käyttöliittymien luomisessa voidaankin hyödyntää kahta eri tyyliä. Käyttöliittymä on mahdollista toteuttaa hyödyntäen HTML- ja CSS-tekniikoita. Tämä tarkoittaa käytännössä sitä, että käyttöliittymäelementit luodaan muun muassa HTML-lomakkeiden nappuloilla ja tekstikentillä. Nämä elementit asetetaan suoraan pelin rajaaman alueen päälle. Ongelmaksi tässä muodostuu se, että elementit eivät suoraan skaalaudu esimerkiksi ikkunan kokoa muuttamalla. Tämä on ratkaistavissa hyödyntäen ehdollisia CSS-tyylejä, jolloin kehittäjä voi itse määritellä, milloin käyttöliittymän elementit esimerkiksi skaalautuvat pienemmiksi. Lisäksi on mahdollista määrittää HTML-käyttöliittymäelementeille prosenttikoot, jolloin skaalautuminen olisi välitöntä. Skaalautuvuuden lisäksi ongelmana on, että käyttöliittymä on irrallaan BabylonJS-pelimoottorin muusta toiminnasta. Elementit eivät näy näkymägraafissa ja esimerkiksi vaihdettaessa pelin tilaa tai näkymää näiden piilotus tulee tehdä manuaalisesti. Myös syötteiden, kuten painikkeiden painallus, tulee käsitellä manuaalisesti hyödyntäen JavaScript-tapahtumia (events). Tästä voi tulla huomattavan paljon lisätyötä ja

koodipohja saattaa erkaantua hieman varsinaisen pelin koodista, koska itse käyttöliittymän tyylittelyt tehdään CSS-määrittelyjen avulla.

Toinen ratkaisu on tehdä käyttöliittymäelementit hyödyntäen BabylonJS-pelimoottoria. Tällöin elementit rakennetaan suoraan canvas-elementtiin erillisellä laajennoksella. Laajennoksia ei kuitenkaan ole saatavilla muuta kuin bGui-käyttöliittymäkirjasto, joka on vanhentunut ja poistunut käytöstä. BabylonJS-kehittäjät ovat rakentamassa 2d-grafiikan piirtämiseen tarkoitettua laajennosta, Canvas2D:tä. Laajennoksen avulla olisi tulevaisuudessa helppo manipuloida 2d-grafiikkaa ja erityisesti luoda käyttöliittymiä tai vaikkapa 2d-pelejä. Nykyisellään Canvas2D on kuitenkin varsin alkuvaiheessa oleva projekti, jonka dokumentaatio on täysin keskeneräinen. Tämän vuoksi on parempi tehdä käyttöliittymä hyödyntäen HTML- ja CSS-tekniikoita, kunnes Canvas2D-laajennus on aikuistunut. Toteutetun pelin käytettävyys on kuitenkin riittävä, sillä peliin oli mahdollista toteuttaa yksinkertainen käyttöliittymä ja valikkorakenne hyödyntäen HTML- ja CSS-tekniikoita. Monimutkaisemmissa peleissä olisi ehkä parempi hyödyntää Canvas2D-pohjaista ratkaisua, jotta ohjelmakoodi ja käyttöliittymäkoodi olisivat paremmin yhteydessä toistensa kanssa.

Case-esimerkissä luotettavuus on ollut tärkeä osa-alue koko kehitysprosessin aikana. Lopullinen pelin suoritus ei saa missään vaiheessa loppua ennenaikaisesti esimerkiksi virheen osalta ja mikäli virheitä tulee, pitäisi pelin selviytyä niistä ilman, että pelaaja joutuu keskeyttämään pelin pelaamisen. Kehitysprosessin aikana korjattiin muun muassa ongelmia lentotasojen muodostamisessa. Mikäli nämä ongelmat olisivat jääneet pelin lähdekoodiin, saattaisi pelaaja pudota loputtomaan tyhjyyteen tai pelin suoritus keskeytyisi. Tällaiset näkyvät ongelmat on helppo paikantaa ja korjata. Toteutetun pelin luotettavuutta horjuttavat kuitenkin tietyt selainteknologiaperäiset asiat. JavaScript-sovellus voi käyttää niin paljon muistia kuin selain vaan antaa myöten. Ongelmaksi muodostuu se, että selain ei saa käyttöönsä koko käyttöjärjestelmän käyttömuistia, vaan vain osan tästä. Tämä jakaantuu lisäksi käyttöön eri välilehtien välille, jolloin on hankala sanoa, paljonko muistia esimerkiksi case-esimerkin peli voi hyödyntää. Tämä saattaa johtaa mahdolliseen pelin suorituksen keskeytymiseen, mikäli muistin käyttö nousee radikaalisti, esimerkiksi muistivuodon (memory leak) seurauksena. Toteutetussa pelissä ei kiinnitetty huomiota muistinhallintaan. Toteutuksessa onkin havaittavissa piikkejä suorituskyvyn osalta, kun JavaScriptin oma roskankeruu (garbage collection) käynnistyy. McAnlis [2013] esittää, että erityisesti JavaScript-pohjaisten pelien tulisi hyödyntää objektiallasta (object-pool), jonka avulla on mahdollista kierrättää objekteja. Kaikki tarvittavat objektit luotaisiin heti pelin käynnistyessä, jolloin niitä voisi hakea ja käytön jälkeen palauttaa takaisin altaaseen. Tällä tavalla olisi mahdollista hallita muistin käyttöä paljon paremmin. Lisäksi muistivuotojen paikantaminen ja korjaaminen helpottuisi, mikä taas lisäisi pelin

luotettavuutta. Esimerkiksi koodikatkelman 5 kaltaisessa tilanteessa olisi mahdollista pommin räjähtämisen jälkeen palauttaa referenssi pommista takaisin objektialtaaseen. Nykyisellään jokainen pommin räjähdys aloittaa roskankeruutapahtuman, joka on osittain vastuussa esimerkiksi kuvassa 4 esitetyistä alhaisista ruudunpäivitysarvoista.

JavaScript on kielenä siirtymässä modulaarisempaan suuntaan. Tämä tarkoittaa käytännössä sitä, että ohjelma koostuu erillisistä osista, jotka pystyvät toimimaan itsenäisesti. Nämä osat voivat kuitenkin tarvita toimiakseen muita moduuleita. JavaScriptin vanhemmalla syntaksilla tämä toteutettiin käytännössä lataamalla JavaScript-tiedostot peräkkäin globaaliin nimiavaruuteen window-muuttujan alaisuuteen. Mikäli jokin moduuli, kirjasto tai ohjelmistokehys tarvitsi toista moduulia toimiakseen, tuli se esitellä ennen kyseistä moduulia lataamalla se globaaliin nimiavaruuteen. BabylonJS on rakennettu osittain hyödyntäen uusinta JavaScript-standardia. Uudemman standardin myötä moduulien lataamisessa ei tarvitsisi enää asettaa kaikkia moduuleja globaaliin nimiavaruuteen, vaan olisi mahdollista ladata tarvittavat moduulit yksitellen. BabylonJS hakee kuitenkin esimerkiksi tarvitsemaansa fysiikkakirjastoa edelleen globaalista nimiavaruudesta, jolloin uusi moduulien latausperiaate ei toimi ollenkaan. Koska kyseisen fysiikkakirjaston oletetaan löytyvän globaalista window-muuttujasta, ainoa ratkaisu on ladata fysiikkamoduuli esimerkiksi toteutettavan pelin päämoduulissa ja asettaa se eksplisiittisesti window-muuttujaan. Tämä on kuitenkin vain väliaikainen korjaus, jolle tulisi löytää jonkin ratkaisu tulevaisuudessa.

BabylonJS-pelimoottorin dokumentaatio on varsin puutteellinen, vaikka jokainen moottorin luokka onkin dokumentoitu automaattisen generaattorin avulla. Käytännön esimerkit ja ohjeet eivät ole kovinkaan kattavia, vaan keskittyvät lähinnä tiettyjen pelimoottorin ominaisuuksien esittelyyn. Pelin logiikan toteuttaminen ja muut työtehtävät jäävät täysin kehittäjän vastuulle. Muun muassa 3d-mallien animointi ja tuominen näkymiin on käsitelty hyvin pintapuolisesti esimerkeissä. Pelimoottorin dokumentaatio ei ohjeista kehittäjää mihinkään suuntaan. Esimerkiksi käyttöliittymistä saatavien syötteiden hallinnalle ei ole yhteistä ohjesäännöstöä. Tämän luulisi kuitenkin olevan avainasia, mikäli BabylonJS haluaa luokitella itsensä jatkossakin pelimoottoriksi. Dokumentaatio on jaoteltu erittäin sekavasti esimerkkien ja ohjeiden osalta, vaikka ne ovat olennainen osa oppimisprosessia. Case-esimerkin pelin luomisprosessin aikana hyödynnettiin runsaasti moottorin Playground-ominaisuutta, jonka avulla oli mahdollista rakentaa prototyyppejä erilaisista ominaisuuksista. BabylonJS-Playground sisältää myös itsessään useita koodiesimerkkejä moottorin ominaisuuksien käytöstä, mikä itsessään auttoi kehitystyössä melkeinpä enemmän kuin varsinainen dokumentaatio. Playground-skenaarioiden jakaminen osoittautui myös erittäin hyödylliseksi ominaisuudeksi. Kehittäjän on mahdollista jakaa tekemänsä

palanen ohjelmakoodia, ja joku toinen voi muokata sitä ja jakaa muokatun ohjelmakoodin takaisin. Esimerkiksi lentoalustojen navigaatioverkon hyödyntämän 3d-mallin ohjelmallisen muodostamisen ongelmatilanteessa jako-ominaisuus oli äärimmäisen hyödyllistä. Myös BabylonJS-foorumilla moottorin pääkehittäjät vastailivat suhteellisen ahkerasti pelin kehittämisessä vastaan tulleisiin ongelmatilanteisiin. Tämä kuitenkin toimii myös hidastavana tekijänä, sillä kehittäjä joutuu aluksi tutkimaan onko kyseiseen ongelmaan vastausta ja vasta tämän jälkeen mahdollisesti itse kirjoittamaan kysymyksen pitkine selostuksineen. Esimerkiksi fysiikkamoottorin käytöstä kehittäjät eivät osanneet antaa muuta apua kuin että sitä ei mahdollisesti kannata käyttää, sillä se monimutkaistaisi peliprojektia huomattavasti.

BabylonJS-pelimoottorin ehkä suurin ongelma kehittäjän näkökulmasta on editoriympäristön puuttuminen. Jokainen pelimoottori tarvitsee jonkinlaisen kehyksen ympärilleen, jotta kehittäjillä on jonkinasteiset suuntaviivat ja standardit sille, miten jotkin asiat tulisi hoitaa. Ilman näitä joudutaan jokaiseen projektiin niin sanotusti kehittämään pyörä uudestaan, jotta asioita saadaan tehtyä nopeasti ja järjestelmällisesti. Tämän ongelman ratkaisu olisi Unityn tai PlayCanvaksen kaltainen editoriympäristö, joka tarjoaisi kehittäjille tietyt raamit toteuttaa ominaisuuksia peliin. Editoriympäristö toisi mukanaan useita etuja, näistä ehkä tärkeimpänä pelin ohjelmakoodin rakenteen parantaminen. Nykyisellään BabylonJS mahdollistaa useita eri ohjelmointityylejä ja tapoja jäsennellä ohjelmakoodia, mutta näistä on hankala valita paras mahdollinen. Tämä aiheuttaa sen, että jokainen projekti voi olla rakenteeltaan hyvinkin erilainen ja täten vaikeuttaa entisestäänkin haastavaa pelinkehitysprosessia. Unityn ja PlayCanvaksen kaltainen skriptausjärjestelmä voisikin olla hyödyllinen myös BabylonJS-pelimoottorissa. Tällöin ohjelmakoodin rakenne pysyisi paremmin kasassa. Case-esimerkin pelissäkin tästä olisi ollut hyötyä, sillä pelin ohjelmakoodin jäsentämiseen ja luokkien suunnitteluun kului huomattavan paljon aikaa.

Pelin kehittäjien kannalta yksi hyödyllisimmistä asioita on varmasti JavaScript kielenä sen tunnettavuuden vuoksi. Nykyisin suurissa ohjelmistoprojekteissa hyödynnetään erityisen paljon JavaScriptiä. Mikäli case-esimerkin pelissä olisi tarvittu tietokantajärjestelmää ja palvelinpuolen toteutusta, olisi nämä ollut hyvä toteuttaa myös JavaScriptillä. Näin olisi ollut mahdollista kirjoittaa lähdekoodia vain yhdellä kielellä. Tämä on kehittäjien kannalta huomattavasti parempi ratkaisu, kuin useamman ohjelmointikielen sisällyttäminen projektiin. Case-esimerkissä hyödynnetään myös minifiointia, joka mahdollistaa JavaScriptin pakkaamisen pieneen tilaan yhdeksi nipuksi, joka sisältää kaikki tarvittavat kirjastot ja lähdekoodin. Tämä on selvä etu verrattuna Unityn WebGL-käännökseen. Tällöin mukaan täytyy ottaa useita C#-kirjastoja, jotka käännetään useiden prosessien kautta JavaScriptiksi. Tällöin pelin

käynnistysnopeus kärsii suuren tiedostokoon vuoksi ja koska Unityn tarvitsemat kirjastot ajetaan myös käynnistyksen yhteydessä.

JavaScript-lähdekoodin kääntäminen ja suorittaminen tapahtuvat selaimessa heti sivun latauksen jälkeen. Näin selainpohjaisten JavaScript-pelien kehittäjien ei tarvitse kääntää ohjelmakoodia millään tavalla, vaan uuden version suorittamiseen riittää web-sivun päivitys. Tämä on erittäin hyödyllistä kehittämisen kannalta, sillä aikaa ei mene hukkaan pitkiin käännösprosesseihin. Myös esimerkiksi Webpack-moduulipaketoija osaa paketoida moduulikohtaisesti lähdekooditiedostoja. Tämä tarkoittaa sitä, että kaikkia lähdekooditiedostoja ei paketoida uudelleen, mikäli jossain tiedostossa lähdekoodi muuttuu.

Selainpohjaisten pelien päivittäminen on huomattavasti helpompi prosessi kuin perinteisten työpöytä- ja mobiilipelien. Jos työpöytäsovellus on saanut päivityksen, täytyy useissa tapauksissa suuri osa sovelluksesta ladata uudelleen. Selainpohjaisissa peleissä koko peli pitää ladata uudelleen joka käynnistyskerralla, joten nämä on jo valmiiksi optimoitu päivityksiä varten.

6 Yhteenveto

JavaScriptin tulevaisuus vaikuttaa valoisalta. Vaikka ohjelmointikieli itsessään oli nopean kehitystyön tulos, on JavaScript kehittymässä entistä suorituskykyisemmäksi ja laadukkaammaksi uusien standardien myötä. Ohjelmistokehittäjät ottavat kielen nykyään paremmin vastaan kuin muutamia vuosia sitten. Tämä on seurausta muun muassa uusista ohjelmistokehyksistä ja kirjastoista, jotka ovat mahdollistaneet lähdekoodin ja yleisesti koko ohjelmiston rakentamisen järkevällä tavalla.

JavaScript onkin vakiinnuttanut asemansa myös selaimen dynaamisen toiminnallisuuden kielenä. Selaintoimittajat, kuten Google ja Microsoft, kilpailevat keskenään JavaScript-moottoreiden toteuttamisessa. Näin uudet suunnitellut muutokset kieleen toteutetaan mahdollisimman nopeasti jokaiseen suureen selaimeen. JavaScript on myös ainoa varteenotettava vaihtoehto toteuttaa dynaamista sisältöä selaimeen, koska muut teknologiat, kuten Java-sovelmat ja Flash-multimediasovellukset, ovat poistuneet käytöstä.

Laajoja JavaScript-pohjaisia pelejä on kuitenkin olemassa erittäin vähän. Näiden puuttumiselle on olemassa monia syitä. Yksi suurimmista on niin sanotun suunnannäyttäjän puuttuminen. Viimeisimmät suuret selainpohjaiset pelit toteutettiin 2000-luvun alussa, jonka jälkeen kehittäjät ovat olleet osittain teknologialimbossa. Vanhoilla teknologioilla olisi ollut turhaa aloittaa tekemään mitään, koska nämä tekniikat jouduttaisiin korvaamaan tulevaisuudessa. JavaScript-pohjaiset ratkaisut WebGL-rajapinnan avulla ovat taas olleet liian riskialtista toteuttaa, sillä kukaan ei ole näyttänyt esimerkillään sen olevan mahdollista. Selainpohjaisten pelien rahoitukseen ja ansaintamalliin ei ole myöskään työpöytäsovellusten tai mobiililaitteiden tapaista valmista kaavaa. Suunnannäyttäjän puuttuminen voi kuitenkin olla vain oire jostain muusta ongelmasta, joka mahdollisesti koskee vain JavaScriptillä toteutettuja pelejä.

Case-esimerkissä toteutettu peli ei ole kovinkaan laaja verrattuna muihin suurehkoihin selainpohjaisiin peleihin, mutta siitä on havaittavissa ongelmia, jotka selittävän suurten JavaScript-pohjaisten pelien puuttumisen markkinoilta. Yksi suurimmista on suorituskyvyllinen epävakaus. Erityisesti BabylonJS:llä toteutetun selainpohjaisen WebGL-pelin ruudunpäivitysnopeus voi vaihdella hyvinkin paljon eri laitteiden välillä, vaikka tämä on yksi teknologian tärkeimmistä myyntivalteista. Lisäksi suorituskyky tulee ottaa huomioon pelinkehityksessä heti alusta lähtien, mikä taas vie energiaa muulta kehitystyöltä

Yksi puute pelinkehitysprosessissa on myös työkalujen puutteellisuus. Case-esimerkissä hyödynnetyn BabylonJS-pelimoottorin dokumentaatio havaittiin

puutteelliseksi. Lisäksi kehittäjiä ei johdateta mihinkään suuntaan tai neuvota miten joitain ominaisuuksia, joita tarvitaan lähes joka pelissä, tulisi hyödyntää. Tilanne tulee tuskin parantumaan ennen kuin pelimoottoria käyttäviä kehittäjiä tulee enemmän ja pelejä vain yksinkertaisesti syntyy enemmän. Pelinkehitysprosessia voisi helpottaa myös editoriympäristö, joita esiintyy ainakin pelinkehitysympäristöissä kuten Unity ja PlayCanvas. Nämä itsessään sisältävät tarkat tavat jäsentää lähdekoodia skripteiksi. Skriptit taas itsessään ovat myös tietyllä tavalla muodostettu ja näillä on oma tapansa kommunikoida keskenään. BabylonJS:llä tehdyssä pelissä kehittäjä joutuu itse keksimään pyörän uudestaan. Sama koskee myös muun muassa käyttöliittymän luomista ja 3d-mallien tuontia, joihin on jo editoriympäristöissä valmiit kaavat rakennettuina.

Puhtaalla JavaScriptillä kirjoitettuja selainpohjaisia pelejä uhkaavat myös kehitysympäristöt, joissa itse selainpohjainen peli kehitetään jollain aivan muulla ohjelmointikielellä kuin JavaScriptillä. Esimerkiksi Unity sisältää mahdollisuuden kääntää peli JavaScriptiksi ja kirjoittaa itse peli toisella kielellä kuten C#. Käännetyt pelit sisältävät kuitenkin suorituskyvyllisiä ongelmia ja vievät huomattavasti enemmän tilaa kuin optimoitu puhtaalla JavaScriptillä toteutettu selainpohjainen peli. Kääntöprosessi voi myös mahdollisesti tuoda odottamattomia ongelmia ja näitä ongelmia on todennäköisesti hankala paikallistaa. Mikäli Unity onnistuu toteuttamaan käännösprosessin sujuvasti, haastaa se editorillaan puhtaat JavaScript-pohjaiset ratkaisut.

Muista ohjelmointikielistä JavaScriptiksi käännetyissä peleissä on kuitenkin myös se ongelma, että nämä pelit eivät kovin helposti voi kommunikoida muun web-sivuston kanssa. Puhtaat JavaScript-ratkaisut taas voivat luontaisesti manipuloida esimerkiksi web-sivuston rakennetta ja tämä on ehdottomasti yksi näiden suurimmista vahvuuksista. Pelin sisällyttäminen web-sivuston ympärille tuo monia etuja. Näin on mahdollista helposti sisällyttää esimerkiksi kirjautuminen osaksi peliä ja samalla kirjautua itse nettisivulle. HTML-pohjaisten käyttöliittymäelementtien hyödyntäminen on tällöin myös helpompaa, koska peli voi suoraan kommunikoida näiden kanssa.

Tulevaisuutta mietittäessä myös WebAssembly voi tuoda jonkinlaista muutosta JavaScript-pohjaiseen ohjelmistokehitykseen. WebAssembly mahdollistaa muiden kielien, kuten C ja C++, ajamisen selainikkunassa lähes natiivien työpöytäsovellusten nopeudella. WebAssembly on suunniteltu toimimaan JavaScriptin kanssa ja WebAssemblyn sisältämän JavaScript-rajapinnan kautta onkin tulevaisuudessa mahdollista ladata WebAssemblyllä toteutettuja moduuleita JavaScript-sovellukseen. Pelisovelluksissa WebAssemblystä saatava hyöty on helppo nähdä, sillä pelit

hyödyntävät usein raskaita operaatioita. Tällöin myös JavaScript-pohjaisten pelimoottorien on mahdollisesti siirryttävä käyttämään WebAssemblyä.

Viiteluettelo

[Aapeli, 2017] Aapeli, 2017, http://www.aapeli.com, Playforia.

[Agile Manifesto, 2001] Agile Manifesto, Manifesto for Agile Software Development. Available as http://agilemanifesto.org. Checked 7.2.2017.

[BabylonJS, 2017] BabylonJS, 2017, https://www.babylonjs.com, BabylonJS.

[Boehm et al., 1978] Barry Boehm, John Brown, Hans Kaspar, Myron Lipow, Gordon MacLeod, and Michael Merritt, *Characteristics of Software Quality*. North-Holland, 1978.

[Bomberman, 1983] Bomberman, 1983, Hudson Soft.

[Crockford, 2008] Douglas Crockford, *JavaScript: The Good Parts*. O'Reilly Media, 2008.

[Eastcott, 2016] Will Eastcott, *PlayCanvas versus Unity WebGL*. Available as https://blog.playcanvas.com/playcanvas-versus-unity-webgl. Checked 7.2.2017.

[Elliot, 2014] Eric Elliot, *Programming JavaScript Applications*. O'Reilly Media, 2014.

[Garciel, 2009] Tali Garciel, *Behind the scenes of modern web browsers*. Available as http://taligarsiel.com/Projects/howbrowserswork1.htm. Checked 9.3.2017.

[Habbo, 2017] Habbo, 2017, http://www.habbo.fi, Sulake Corporation Oy.

[Hevner et al., 2004] Alan Hevner, Salvatore March, Jinsoo Park, Sudha Ram, Design Science in Information Systems Research. *MIS Quarterly* 28 1, (2004) 75-105.

[Hokkanen, 2017] Panu Hokkanen, Bomb Quest Game 2017, https://github.com/panuchka/bomb-quest-game.

[Hughes & Cotterell, 2009] Bob Hughes and Mike Cotterell, *Software Project Management* Fifth Edition. McGraw-Hill Education, 2009.

[ISO 25010] ISO/IEC 25010:2011 Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models, https://www.iso.org/standard/35733.html.

[Lappalainen, 2015] Elina Lappalainen, *Sulakkeen Habbo Hotel yrittää uutta nousua mobiilissa - Pääseekö Suomi-pelien muinainen tähti jaloilleen?* Available as http://www.talouselama.fi/uutiset/sulakkeen-habbo-hotel-yrittaa-uutta-nousua-mobiilissa-paaseeko-suomi-pelien-muinainen-tahti-jaloilleen-3365291. Checked 20.7.2017

[March & Smith, 1995] Salvatore March and Gerald Smith, Design and natural science research on information technology, 1995. *Decision Support Systems*, 15, 251–266.

[McAnlis, 2013] Colt McAnlis, *Static Memory Javascript with Object Pools*. Available as https://www.html5rocks.com/en/tutorials/speed/static-mem-pools. Checked 28.6.2017.

[Minecraft, 2017] Minecraft, 2017, https://minecraft.net, Mojang.

[Murphy-Hill et al., 2014] Emerson Murphy-Hill, Thomas Zimmermann and Nachiappan Nagappan, Cowboys, Ankle Sprains, and Keepers of Quality: How Is Video Game Development Different from Software Development?. In: Proceedings of the *36th International Conference on Software Engineering*. ACM, 2014. p. 1-11.

[Parisi, 2012] Tony Parisi, *WebGL: Up and Running*. O'Reilly Media, 2012.

[Petrillo et al., 2008] Fábio Petrillo, Marcelo Pimenta, Francisco Trindade and Carlos Dietrich, Houston, we have a problem...: A Survey of Actual Problems in Computer Games Development. In: Proceedings of the *2008 ACM symposium on Applied computing*. ACM, 2008. p. 707-711.

[PlayCanvas, 2017] PlayCanvas, 2017, https://playcanvas.com, PlayCanvas.

[Project Zomboid, 2017] Project Zomboid, 2017, https://www.projectzomboid.com, The Indie Stone.

[RuneScape, 2017] RuneScape - Fantasy massively multiplayer online role-playing game, 2017, http://www.runescape.com, Jagex Ltd.

[Smed & Hakonen, 2003] Jouni Smed ja Harri Hakonen, *Towards a definition of a computer game* (pp. 1-3). Turku, Finland: Turku Centre for Computer Science, 2003.

[Sommerville, 2009] Ian Sommerville, *Software Engineering* Ninth Edition. Addison-Wesley, 2009.

[StatCounter, 2017] StatCounter GlobalStats, 2016, http://gs.statcounter.com, StatCounter.

[Unity, 2017] Unity, 2017, https://unity3d.com, Unity Technologies.

[Vanhatupa, 2010] Juha-Matti Vanhatupa, *Browser Games: The New Frontier of Social Gaming*. Springer, 2010.

[WebAssembly, 2017] WebAssembly, 2017, http://webassembly.org, W3C WebAssembly Community Group.

[WebGL, 2017] WebGL, 2017, https://www.khronos.org/webgl, Khronos Group.