Tampereen yliopisto

Otto-Ville Savolainen

SISÄTILAPAIKANNUKSELLA KERÄTYN DATAN VISUALISOINTI JAVASCRIPTIN 3D-KIRJASTOILLA

Informaatioteknologian ja viestinnän tiedekunta
Pro gradu -tutkielma
Toukokuu 2019

TIIVISTELMÄ

Otto-Ville Savolainen: Sisätilapaikannuksella kerätyn datan visualisointi JavaScriptin 3D-kirjastoilla
Ohjaaja: Timo Poranen
Pro gradu -tutkielma
Tampereen yliopisto
Tietojenkäsittelytieteiden tutkinto-ohjelma
Toukokuu 2019

Kiinnostus sisätilapaikannukseen on kasvanut viime aikoina eri toimialoilla, kuten teollisuudessa ja urheilussa. Kasvanut kiinnostus on johtanut sisätilapaikannusta koskeviin tutkimuksiin, joiden pohjalta Bluetooth Low Energy on todettu tällä hetkellä yhdeksi sisätilapaikannukseen parhaiten sopivimmista teknologioista. Sen riittävän pitkä kantama, kustannustehokkuus ja matala virrankulutus tekee teknologiasta hyvin soveltuvan vaihtoehdon esimerkiksi nopeatempoisen sisätiloissa tapahtuvan urheilun, kuten jääkiekon, seurantaan.

Sisätilapaikannuksen avulla kerätyn sijaintidatan avulla jääkiekko-ottelua voidaan seurata ja analysoida reaaliajassa, automaattisesti ja kokonaisvaltaisesti älykiekkojärjestelmä Wisehockeyn avulla. Wisehockey-järjestelmä tarjoaa myös Angularilla, TypeScriptillä ja Three.js:llä toteutetun 3D-mediatoistimen, Wiseplayerin, jonka avulla ottelun kohokohtia voidaan katsoa reaaliajassa tai ottelun jälkeen 3D-mallien avulla.

Koska JavaScriptille on olemassa kaksi hyvin suosittua ja laajalti käytössä olevaa 3D-kirjastoa Three.js ja Babylon.js, tutkielmassa toteutettiin Wiseplayerin toimintaa imitoivat sovellukset käyttäen kumpaakin kirjastoa. Käytettyjä kirjastoja vertailtiin tutkielmassa yleisellä tasolla, sekä erilaisten laatutekijöiden avulla. Näitä laatutekijöitä olivat ylläpidettävyys, käytettävyys ja tehokkuus. Vertailun tarkoituksena on perehtyä kahden aiemmin mainitun kirjaston heikkouksiin ja vahvuuksiin ja tutkia kuinka ne soveltuvat Wiseplayerin käyttöön.

Vertailun pohjalta tehtyjen havaintojen perusteella todettiin, että molemmat kirjastot ovat soveltuvia Wiseplayerin käytettäväksi. Vaikka kirjastot ovat hyvin samankaltaisia, Three.js todettiin soveltuvammaksi pieniin ja kevyihin projekteihin, kun taas Babylon.js soveltuu suurempiin ja raskaampaa 3D-grafiikkaa käsitteleviin projekteihin. Tästä syystä ei nähty syytä vaihtaa Three.js:ää Babylon.js:ään Wiseplayerin tämän hetkisen toteutuksen puitteissa. Wiseplayerin kehittyessä raskaammaksi ja monipuolisemmaksi sovellukseksi kirjaston vaihtoa olisi kuitenkin syytä harkita.

Avainsanat: älykiekko, wisehockey, javascript, 3D, kolmiulotteisuus, datan visualisointi

Tämän julkaisun alkuperäisyys on tarkastettu Turnitin OriginalityCheck –ohjelmalla.

Sisällys

1 Johdanto 1

2 Paikannusteknologiat 3

2.1 Satelliittipaikannus 3

2.2 Sisätilapaikannus 4

2.2.1 WiFi 5

2.2.2 Bluetooth 5

2.2.3 Ultra-wideband 6

2.2.4 Kameraseuranta 6

3 Ohjelmistojen laatustandardit 8

3.1 McCallin laatutekijät 10

3.2 ISO 9126 12

3.3 Laatutekijöiden valinta 14

4 JavaScript ja 3D 16

4.1 WebGL 17

4.2 TypeScript 18

4.3 Angular 20

4.4 3D-kirjastot 21

5 Wisehockey 23

5.1 Quuppa 23

5.2 Arkkitehtuuri 23

5.2.1 Webportal 25

5.2.2 Wiseplayer 27

6 3D-kirjastojen vertailu 29

6.1 Three.js toteutus 30

6.2 Babylon.js toteutus 36

6.3 Yleinen vertailu 40

6.4 Ylläpidettävyys 40

6.5 Käytettävyys 42

6.6 Tehokkuus 43

7 Yhteenveto 48

8 Viiteluettelo 50

1 Johdanto

Tässä tutkielmassa tarkastellaan sisätilapaikannuksen avulla kerätyn datan visualisointia JavaScriptin 3D-kirjastoilla ja vertaillaan kahden suosituimman kirjaston soveltuvuutta jääkiekko-ottelusta kerätyn sijaintidatan visualisointiin. Datan visualisoinnilla tarkoitetaan kerätyn tiedon esittämistä käyttäjäystävällisemmässä muodossa, joka tässä tutkielmassa tarkoittaa jääkiekko-otteluiden esittämistä 3D-grafiikan avulla.

Tutkielman tavoitteena on tutkia ja todeta, onko Wisehockey-järjestelmällä pelattujen otteluiden visualisointiin tarkoitetun Wiseplayerin käyttämä 3D-kirjasto Three.js paras vaihtoehto tähän tehtävään. Koska Wisehockey on kasvava ja kehittyvä järjestelmä, on tärkeää varmistua käytettävien teknologioiden sopivuudesta ja tutustua vaihtoehtoisiin teknologioihin välttääkseen mahdolliset ongelmat tulevaisuudessa. Three.js on toiminut Wiseplayerin käytössä hyvin, mutta vaihtoehtoisten toteutustapojen tutkiminen voi avata ideoita ja mahdollisuuksia uusille ominaisuuksille, joita nykyinen kirjasto ei välttämättä pysty tarjoamaan.

Tutkimus alkaa kertomalla ulko- ja sisätilapaikannukseen käytettävistä teknologioista luvussa kaksi. Tässä luvussa käydään läpi yleisimmät ja parhaiten soveltuvat teknologiat sekä perehdytään lyhyesti niiden toimintaperiaatteisiin. Ulkotilapaikannuksessa keskitytään vain satelliittipaikannukseen, kun taas sisätilapaikannuksen osalta paneudutaan WiFi-paikannukseen, Bluetooth Low Energy ja Ultra-wideband -teknologioihin ja niiden avulla paikantamiseen sekä kameraseurantaan.

Luvussa kolme tutustutaan ohjelmistojen laatustandardeihin kahden laatutekijöitä koostavan mallin avulla. McCallin laatutekijät koostuvat yhdestätoista ja ISO 9126 -standardi kuudesta laatutekijästä, joita voidaan hyödyntää ohjelmiston laadun mittaamisessa. Molemmat mallit sisältävät osittain samoja laatutekijöitä, joista tutkielmassa käytettäviksi valikoitui ylläpidettävyys, käytettävyys ja tehokkuus.

Neljäs luku paneutuu web- ja 3D-ohjelmointiin kertomalla JavaScriptistä, sen historiasta ja nykyajasta. JavaScriptin lisäksi paneudutaan Wiseplayerissa käytettäviin TypeScriptiin ja Angulariin, jotka ovat syntyneet JavaScriptin seurauksena. Näiden lisäksi 3D-ohjelmoinnin osalta keskitytään WebGL:ään, joka tarjoaa moderneille selaimille mahdollisuuden esittää 2D- ja 3D-grafiikkaa. Lisäksi luvussa neljä esitellään tutkimukseen valitut 3D-kirjastot Three.js ja Babylon.js.

Viides luku esittelee Wisehockey-järjestelmää ja sen arkkitehtuuria. Tähän liittyy oleellisesti suomalainen paikannusteknologiaan erikoistunut yritys Quuppa, joka tarjoaa paikannusteknologian Wisehockey-järjestelmän käytettäväksi. Arkkitehtuuri käydään läpi hyvin yleisellä tasolla, jonka jälkeen esitellään Wisehockey-järjestelmää käyttäville tahoille tarkoitetut Webportal ja Wiseplayer. Näiden palveluiden kautta kerättyä dataa on mahdollista tarkastella käyttäjäystävällisestä näkökulmasta.

Luvussa kuusi siirrytään varsinaiseen 3D-kirjastojen vertailuun. Tutkimus toteutettiin tekemällä kaksi identtisesti toimivaa Wiseplayerin toimintoja imitoivaa ohjelmaa. Ohjelmat toteutettiin käyttäen Three.js ja Babylon.js kirjastoja siten, että kumpikin ohjelma hyödyntää vain toista kirjastoista. Toteutettujen ohjelmien toiminnot valikoituivat Wiseplayerin pääominaisuuksien pohjalta, eli ohjelma renderoi selaimeen kaukalon, kiekon sekä 12 pelaajan 3D-mallit. Kaukalo pyörii oman x- ja y-akselinsa ympäri imitoiden Wiseplayerin ominaisuutta, jossa käyttäjän on mahdollista kallistaa kaukaloa ja tarkastella ottelua halutusta kuvakulmasta. Kiekon ja pelaajien 3D-mallit liikkuvat sattumanvaraisesti ennalta määrätyn alueen sisällä kuvastaen sitä, kuinka Wiseplayerissa pelaajat ja kiekko liikkuvat kerättyjen sijaintidatapisteiden avulla ympäri kaukaloa.

Tutkimuksessa vertailtiin valittujen 3D-kirjastojen yleistä tilannetta GitHubissa, sekä niiden ylläpidettävyyttä, käytettävyyttä ja tehokkuutta. Yleisellä tilanteella tarkoitetaan GitHubissa saatujen tähtien ja edistäjien määrää, sekä viimeisimmän stabiilin version julkaisuajankohtaa. Ylläpidettävyyttä tutkittiin määrittelemällä neljä uutta ominaisuutta, jotka lisättäisiin toteutettuihin ohjelmiin ja katsottiin, onko näiden uusien ominaisuuksien lisääminen mahdollista ja kuinka suuri vaiva niiden toteuttamisesta syntyisi. Käytettävyyttä tutkittiin tarkastelemalla valittujen kirjastojen dokumentaatiota ja käyttöohjeita, toteutettujen ohjelmakoodien luettavuutta sekä kirjastojen käyttöönoton helppoutta. Tehokkuuden osalta puolestaan tarkasteltiin asennettavien kirjastojen pakettien kokoa, toteutettujen ohjelmien FPS-lukemia selaimessa, niiden suorituskykyä Chromen tarjoaman kehittäjätyökalujen yhteenvetodiagrammin avulla sekä tuotantokelpoisen version koontiaikaa.

Lopuksi luvussa seitsemän kerrotaan tutkimuksen tulokset ja vedetään yhteen lopullinen päätös siitä, onko Wiseplayeria syytä siirtyä toteuttamaan toisella kirjastolla. Lisäksi viimeisessä luvussa tiivistetään molempien kirjastojen hyvät ja huonot puolet sekä niiden soveltuvuudet erilaisiin käyttötarkoituksiin.

2 Paikannusteknologiat

Paikantaminen on prosessi, jossa määritetään kohteen sijainti. Paikantaminen on viime aikoina ollut aktiivinen tutkimuskohde ja uusia paikannusjärjestelmiä kehitetään jatkuvasti. Paikantaminen voidaan jakaa kahteen osa-alueeseen riippuen siitä missä paikantaminen tapahtuu: ulkotilapaikannukseen ja sisätilapaikannukseen. Ulkotilapaikannuksessa tunnetuin järjestelmä on satelliittipaikannus, kun taas sisätilapaikannus riippuu niin monen tekijän summasta, ettei yhtä ja ainoaa järjestelmää voida nostaa ylitse muiden.

Vaikka monet ulkotila- ja sisätilapaikannusjärjestelmät käyttävät samoja periaatteita, poikkeavat nämä kaksi paikannusteknologiaa suuresti toisistaan. Sisätilapaikannus on usein vaativampaa, sillä sisätiloissa on monesti useita objekteja kuten seiniä ja huonekaluja, jotka heijastavat signaaleja ja aiheuttavat niille kulkuesteitä. Nämä puolestaan aiheuttavat viivettä paikantamisessa, vaimentavat signaalien vahvuuksia ja heikentävät sijainnin tarkkuutta. [Alarifi *et al.* 2016]

Noin joka kolmas älypuhelimen omistaja käyttää sijaintiin perustuvia palveluja kuten navigointia, sosiaalisia palveluja ja seurantaa aktiivisesti [Duggan ja Smith 2013]. Ulkotilanpaikannuksessa käytettävät satelliittijärjestelmät ovat laajalti käytössä ulkona liikkuessa, mutta sisätilanpaikannuksessa on vielä omat haasteensa [Basiri *et al. 2017].*

2.1 Satelliittipaikannus

Satelliittipaikannusjärjestelmät ovat ulkoilmapaikannukseen tarkoitettuja järjestelmiä ja ne hyödyntävät paikannuksessa satelliitteja. Paikannus perustuu ajanhetkiin: avaruudessa kulkeva satelliitti lähettää aikatiedon sisältävän signaalin maanpinnalla olevaan vastaanottimeen, joka laskee saapumisajan X kaavalla

$$X = V - L,$$

jossa V tarkoittaa signaalin vastaanottamisajanhetkeä, ja L puolestaan signaalin lähetysajanhetkeä, jonka satelliitti on sisällyttänyt signaaliin. Mikäli vastaanotin tietää signaalin nopeuden, sen on yksinkertaista laskea satelliitin etäisyys kertomalla X signaalin nopeudella. Kun vastaanotin on saanut laskettua tarpeeksi monen navigaatiosatelliitin etäisyyden, sen on mahdollista muodostaa saamiensa etäisyyksien perusteella tarkka sijainti. [Prasad ja Ruggieri 2005] Voimassaolevat satelliittipaikannusjärjestelmät voidaan luokitella karkeasti kolmeen luokkaan*: Global Navigation Satellite Systems* (GNSS), *Regional Navigation Satellite Systems* (RNSS) ja *Satellite-based Augmentation Systems* (SBAS). [Rao *et al.* 2012]

Tällä hetkellä toiminnassa on kaksi GNSS-järjestelmää, jotka ovat toimineet lähes kaksi vuosikymmentä: amerikkalaisten kehittämä *GPS* ja venäläisten kehittämä *GLONASS*. Lähitulevaisuudessa näiden kahden rinnalle on kuitenkin tulossa Euroopan

Unionin *Galileo*-sateliittipaikannusjärjestelmä ja Kiinan lanseeraama *COMPASS*. Nämä uudet järjestelmät on määrä saada täysin valmiiksi vuonna 2020. [Rao *et al.* 2012]

RNSS-järjestelmät poikkeavat GNSS-järjestelmistä siinä, että ne tarjoavat paikannusjärjestelmät kattavat vain tietyn alueen tai valtion, jonka seurauksena niiden satelliittien määrät ovat huomattavasti GNSS-järjestelmiä pienemmät: tämän hetkiset GNSS-järjestelmät käyttävät 24 satelliittia, kun taas RNSS-järjestelmät vain 5–7 satelliittia. Ne kykenevät toimimaan täysin itsenäisesti tai osana GNSS-järjestelmää, mutta jälkimmäisessä tapauksessa järjestelmien on oltava täysin yhteensopivia keskenään. Tällä hetkellä suurimmat RNSS-järjestelmät ovat Japanin *Quasi-Zenith Satellite System* (QZSS), Intian *Indian Regional Navigation Satellite System* (IRNSS) ja Kiinan *Beidou-1*, jonka pohjalta GNSS-järjestelmä *COMPASS* laajenee. [Rao *et al.* 2012]

Kriittisissä tehtävissä, kuten siviililennoissa ja meriliikenteessä pelkän GNSS varassa suunnistaminen ei ole välttämättä riittävän tarkkaa tai saatu sijaintidata ei ole sopivaa, jonka seurauksena luotiin SBAS-järjestelmät. SBAS-järjestelmät muodostavat maanpinnalla verkoston useista asemista, joiden avulla valvotaan GNSS- ja geostationaarisiasatelliitteja. Valvonnan avulla pyritään korjaamaan virheellisiä tietoja ja tuomaan korjattua ja ehjää tietoa julki, jonka avulla SBAS-järjestelmää käyttävä laite saa tuotettua tarkempaa sijaintia loppukäyttäjälle. Yhdysvaltain ilmailuhallinnon sponsoroima *Wide Area Augmentation System* (WAAS), Euroopan avaruusjärjestön sponsoroima *European Geostationary Navigation Overlay Service* (EGNOS), Japanin sponsoroima *Multifunctional Satellite-Based Augmentation System* (MSAS) ja Intian tukema *GPS and Geoaugmented Navigation* (GAGAN) muodostavat lähes koko maapallon kattavan SBAS-järjestelmän, joka tukee pääasiassa lentopaikannusta. [Rao *et al.* 2012]

2.2 Sisätilapaikannus

Sisätilapaikannusjärjestelmällä (indoor positioning system, IPS) tarkoitetaan paikannusjärjestelmää, joka paikantaa kohdetta suljetun sisäalueen, esimerkiksi rakennuksen sisällä jatkuvasti ja reaaliajassa. Sisätilapaikannusjärjestelmät skaalautuvat useaan eri käyttötarkoitukseen ja niitä käytetään esimerkiksi tavaroiden paikantamiseen yksityishenkilöiden kotona, näkövammaisten ohjeistamiseen julkisissa rakennuksissa ja lasten seurantaan tapahtumissa, joissa on suuria ihmismassoja. Etenkin sairaalat ovat omaksuneet sisätilapaikannusjärjestelmät itselleen, sillä niiden avulla on helppoa seurata potilaita ja ehkäistä kalliiden laitteiden varastamista seurannan avulla. [Alarifi *et al.* 2016]

Sisätilapaikannus perustuu pääsääntöisesti lähettimistä ja vastaanottimista koostuvaan asetelmaan, joka tarjoaa kaksi lähestymistapaa määritellä kohteen sijainti sisätiloissa: ohjelmakohtainen lähestymistapa ja palvelinkohtainen lähestymistapa. Ohjelmakohtaisessa lähestymistavassa ideana on se, että sijainti määritellään paikannettavassa laitteessa olevan erillisen ohjelman avulla. Tämä lähestymistapa mahdollistaa sen, että

esimerkiksi paikannettava kohde tietää ja näkee itse oman sijaintinsa kartalla. Palvelinkohtaisessa lähestymistavassa kyseessä on yksisuuntainen kommunikaatio lähettimen ja vastaanottimen välillä. Lähetin lähettää signaaleja, joita vastaanotin ottaa vastaan ja ohjaa palvelimelle käsiteltäväksi. Palvelin käsittelee vastaanotetun tiedon ja muodostaa sen perusteella sijainnin paikannettavalle kohteelle. [Insoft 2018]

Yangin ja Shaon [2015] mukaan kasvava kysyntä markkinoilla sisätilanpaikannukselle on herättänyt kiinnostusta niin tutkimuskeskuksissa kuin teollisuusyhtiöissäkin, jonka vuoksi sisätilapaikannuksen voidaan nähdä kehittyvän lähivuosina huomattavasti. Alarifi ja muut [2016] mainitsevat sisätilapaikannuksessa olevan viisi pääasiallista laatuvaatimusta järjestelmälle: sijainnin tarkkuus, alueen kattavuus, sijainnin päivitysnopeus, infrastruktuurin vaikutus ja sattumanvaraisten häiriöiden vaikutukset. Basiri ja muut [2017] kuitenkin huomauttavat sisätilapaikannuksen sisältävän vielä haasteita paikannusjärjestelmien laatuvaatimuksissa, yksityisyydessä ja sisällön, kuten karttojen saatavuudessa. Näiden haasteiden rinnalle Yang ja Shao [2015] nostavat tarkkojen paikannusjärjestelmien korkeat kustannukset ja sisätilojen vaativan ympäristön.

2.2.1 WiFi

Chandlerin ja Mundayn [2016] määritelmän mukaan WiFi on laajalti tiedossa oleva nimitys IEEE 802 -standardisoidulle tiedonsiirrolle. Se tarkoittaa paikallista, lyhyen kantaman verkkoyhteyttä, joka käyttää radiosignaaleja tiedon lähettämiseen ja vastaanottamiseen. Yang ja Shao [2015] epäilevät, että WiFistä tulisi merkittävä väline sisätilapaikannuksessa sen laajan levinneisyyden ja helpon käyttöönoton vuoksi, sillä jo olemassa olevia WiFi-tukiasemia ja laitteita on mahdollista käyttää sisätilapaikannuksessa. Tämän väitteen lisäksi Insoft [2018] näkee WiFi-paikannuksen etuina sen suuren kantaman (n. 150 metriä) ja paikannusmahdollisuuden myös rakennusten eri kerrosten välillä.

WiFi-paikannus vaatii kuitenkin huomattavan määrän laskentatehoa, jonka seurauksena energiankulutus on suuri eikä etenkään mobiililaitteiden akut kestä pitkäkestoista sijainnin seurantaa WiFi:n avulla. Tähän ongelmaan on kuitenkin kehitelty erityisesti älypuhelimia varten suunniteltuja algoritmeja, jotka vaatisivat vähemmän laskentatehoa ja olisivat tätä myöten energiaystävällisempiä. [Bisio *et al.* 2013] Myös Insoft [2018] nostaa WiFi-paikannuksen yhdeksi heikkoudeksi virrankulutuksen. Tämän lisäksi WiFi-paikannuksen epätarkkuus (5-15 metriä) ja mahdolliset viiveet WiFissä nousevat haittapuoliksi tässä paikannusmenetelmässä.

2.2.2 Bluetooth

Bluetooth on maailmanlaajuinen standardi laitteiden väliselle langattomalle kommunikoinnille. Bluetoothille tyypillisiä ominaisuuksia ovat lyhyt kantama, pieni virrankulutus, matalat kustannukset, pieni koko ja turvallisuus. Nykyään lähes jokainen kannettava laite

tarjoaa mahdollisuuden Bluetooth-yhteydelle, ja Bluetoothia käytetään näiden lisäksi esimerkiksi myös langattomissa kuulokkeissa, kaiutinjärjestelmissä, kameroissa ja pelikonsoleissa. *Bluetooth Low Energy* (BLE) on Bluetooth-teknologian uusin parannus, joka lisättiin Bluetooth 4.0:n laatuvaatimuksiin. Vaikka tavallinen Bluetooth on matala virrankulutteinen, niin BLE:n avulla virrankulutusta on saatu pienennettyä entisestään, ja laite voi toimia sormipariston kokoisella akulla jopa vuosia ilman akun lataamista tai vaihtamista. [Gupta 2013]

Jotta virrankulutus saadaan mahdollisimman pieneksi, on viestien välityksen oltava nopeaa. Tämän vuoksi BLE käyttää viestien välityksessä neljääkymmentä 2MHz levyistä kanavaa, joista kolmea voidaan hyödyntää sijainnin määrittämiseksi [Zhuang *et al.* 2016]. Erittäin matalan virrankulutuksen ja pienten kustannusten lisäksi Insoft [2018] näkee BLE:n eduiksi sen tarkkuuden (jopa yksi metri) ja joustavan integroinnin nykyisiin järjestelmiin. Haittapuoliksi kuitenkin nousevat vaatimukset lisälaitteille, WiFi-paikannukseen verrattuna lyhyen kantaman (noin 70 metriä) ja häiriöherkkyyden radiohäirinnän ilmentyessä. Näistä huolimatta Basiri ja muut [2017] väittävät BLE:n olevan tällä hetkellä sisätilapaikannukseen ja seurantaan parhaiten sopiva teknologia.

2.2.3 Ultra-wideband

Alarifin ja muiden [2016] mukaan *ultra-wideband* (UWB) on tuorein, tarkin ja kaikkein lupaavin sisätilapaikannusteknologia nykypäivänä. UWB:llä tarkoitetaan radioteknologiaa, joka tarjoaa erittäin nopean tiedonvälityksen ja kommunikoinnin kahden kohteen välillä. UWB perustuu tiheästi toistuviin, lyhyihin pulsseihin ja laajaan, jopa yli 10GHz kaistanleveyteen, joiden avulla signaali läpäisee tehokkaasti sisätiloissa olevat esteet ja tarjoaa erittäin tarkkaa paikannustietoa.

Insoft [2018] kertoo UWB:n olevan tällä hetkellä käytössä järjestelmissä, joissa tarvitaan tarkkaa sijaintitietoa sekä kaksi- että kolmiulotteisesti. Sijainnin tarkkuus saadaan määriteltyä UWB:n avulla jopa alle 30 senttimetrin tarkkuudella ja sijainti päivittyy jopa 100 kertaa sekunnissa. Heikkoutena esimerkiksi WiFi- ja BLE-paikannusteknologioihin verrattuna UWB-järjestelmät kustantavat huomattavasti enemmän ja akunkesto on heikompi kuin BLE:ssä.

2.2.4 Kameraseuranta

Gasparin ja Oliveiran [2009] mukaan erilaisia ”lähetin-vastaanotin” -ratkaisuja haastamaan on noussut myös paikantaminen kameran avulla. Kameraseuranta mahdollistaa sisätilapaikannuksen pienillä investoinneilla, minkä vuoksi sen suosio ja soveltuvuus on kasvanut sisätilapaikannuksen markkinoilla.

Kameraseuranta perustuu kuvien prosessointiin. Koska videokamera kuvaa useita kuvia sekunnissa, täytyy kameraseurantajärjestelmän prosessoida kuvajonoja ja päätellä niiden perusteella yhden tai useamman kohteen liikehdintää. Jotta tätä liikehdintää voidaan päätellä, täytyy kuvasta ensiksi tunnistaa tarkkailtavat kohteet. Viimeisimmät teknologiat hyödyntävät ennustavia suotimia, jotka arvioivat tarkkailtavan kohteen tilan seuraavassa kuvajonon kuvassa. Ennustavista suotimista etenkin Kalman ja Particle ovat laajalti käytettyjä. Kalman-suotimen ehdottomana etuna on sen kyky ennustaa tarkkailtavan kohteen mennyttä, nykyistä ja tulevaa tilaa. Jos tarkkailtava kohde esimerkiksi peittyy jonkin toisen elementin taakse ja sen todellinen tila kadotetaan, Kalman-suodin pystyy päättelemään sen. Particle-suodin puolestaan kykenee ennustamaan vain tarkkailtavan kohteen tilaa seuraavassa kuvassa. [Morais *et al.* 2012]

Erityisesti futsal, eli sisäjalkapallo on kiinnostanut niin teknisesti kuin tieteellisestikin kameraseurantaa kehittäviä tahoja. Lajin dynaaminen luonne ja pienessä tilassa tapahtuvan pelin vuoksi joukkueet joutuvat miettimään taktiikoita ja hiomaan kuvioita huippuunsa, jonka vuoksi ulkopuolelta tuleva automaattinen taktinen, fyysinen ja fysiologinen analyysi on erittäin tervetullutta. Lajin seuranta useiden kameroiden avulla useista eri tarkkailupisteistä tallentaa tärkeitä redundansseja automaattisen prosessoinnin käytettäväksi. Tarkempien tulosten takaamiseksi voidaan käyttää useampia tarkkailupisteitä ja tiedon integrointiprosesseja. [Morais *et al.* 2012]

Morais ja muut [2012] toteavat, että kameraseurannan suurimmat ongelmat ilmentyvät, kun seurattavien kohteiden reitit risteävät tai kulkevat yhtenäisesti. Risteystilanteessa seuranta tosin harvoin häiriintyy, mutta tarkkailtavien kohteiden yhtenäinen reitti sulauttaa usein tarkkailtavat kohteet yhdeksi kohteeksi. Näistä ongelmista huolimatta vastaavanlaista järjestelmää on mahdollista käyttää myös muissa urheilulajeissa sekä ympäristöissä, kuten turvallisuuden tarkkailussa ja ihmismäärien laskennassa.

3 Ohjelmistojen laatustandardit

Naik ja Tripathy [2008] kertovat tietokoneiden alkuaikoina ohjelmistokehittäjien keskittyneen pääsääntöisesti kehittämään ohjelmistojen toiminnallisuuksia, sillä valtaosa ohjelmistojen loppukäyttäjistä olivat korkeasti koulutettuja ammattilaisia, kuten matemaatikkoja, tiedemiehiä ja insinöörejä. Henkilökohtaisten tietokoneiden ja internetin kehityksen myötä ohjelmistokehittäjien tuli kuitenkin kiinnittää huomiota myös ohjelmistojen käytettävyyteen ja käyttäjäkokemukseen. Tutkimuksen mukaan tuotteen laatu voidaan huomata viiden eri näkökulman kautta:

1. Transsendenttinen näkökulma
2. Käyttäjänäkökulma
3. Valmistusnäkökulma
4. Tuotenäkökulma
5. Arvoon perustuva näkökulma

Transsendenttisessa näkökulmassa laatu on jotain sellaista, joka huomataan kokemuksen kautta eikä sitä voida tuoda jäljitettävään muotoon tai mitata. Tällä tarkoitetaan sitä, että laatua on vaikea kuvailla, mutta laadukkaan tuotteen tullessa vastaan sen tunnistaa. [Naik ja Tripathy 2008]

Käyttäjänäkökulma puolestaan liittyy siihen, kuinka tuote vastaa käyttäjän tarpeisiin ja odotuksiin. Tämä ei tarkoita ainoastaan sitä mitä tuote tarjoaa, vaan myös mahdollisissa sopimuksissa luvattuja kohtia esimerkiksi takuuehdoista. Koska käyttäjät kokevat tuotteet eri tavoin, tuotteen voidaan nähdä olevan laadukas, jos se tyydyttää suuren osan käyttäjäkunnastaan. Tämän vuoksi onkin tärkeää tutkia mitä ominaisuuksia käyttäjät arvostavat ja mitkä he mieltävät tärkeiksi. [Naik ja Tripathy 2008]

Valmistusnäkökulmassa tuotteen laatu muodostuu aivan valmistusprosessin alusta asti. Jos tuotteen valmistuksessa noudatetaan tiettyjä standardeja, tuote voidaan luokitella laadukkaaksi. Tämä näkökulma on kuitenkin herättänyt kritiikkiä, sillä joidenkin mukaan tällä varmistetaan vain tuotteiden yhdenmukaisuus, ei laadukkuus. Toisaalta jos tuotteen valmistusprosessi on tarkkaan määritelty, sitä on myös helppo kehittää, mikä puolestaan johtaa entistä laadukkaampiin tuotteisiin. [Naik ja Tripathy 2008]

Tuotenäkökulman keskeinen teesi on se, että mikäli tuote on valmistettu laadukkaista sisäisistä ominaisuuksista, niin se on ulkoisesti laadukas. Tässä näkökulmassa siis luodaan suhde tuotteen sisäisten ominaisuuksien ja ulkoisen laadun välille. Esimerkiksi jos ohjelmisto on hyvin modulaarinen, niin se helpottaa tuotteen testausta, ylläpidettävyyttä ja näin ollen myös ulkoista laatua. Sisäiset ominaisuudet ovat tässä näkökulmassa mitattavissa olevia määreitä, jotta laatua voidaan tarkastella objektiivisesti. [Naik ja Tripathy 2008]

Arvoon perustuva näkökulma yhdistää tuotteen erinomaisuuden ja arvon, jossa syvin ajatus on se, kuinka paljon asiakas on valmis maksamaan tuotteen laadusta. Todellisuudessa laatu on arvotonta, jos tuote ei ole taloudellisesti järkevä. [Naik ja Tripathy 2008]

Jørgensenin [1999] mukaan suurin osa ohjelmistojen laatustandardeista ja viitekehyksistä vaativat tai ainakin suosittelevat ohjelmistojen laadun mittaamista. Tällä hetkellä esille tuotavien vaatimusten ja niiden mittaamisen välillä on kuitenkin selkeä näkemysero. Tästä hyvänä esimerkkinä toimii ohjelmistojen laatustandardin ISO 9000-3 kohta 6.4.1, jonka Kehoe ja Jarvis [2012] ilmaisevat seuraavasti:

> "There are currently no universally accepted measures of software quality … The supplier of software products should collect and act on quantitative measures of the quality of these software products."

Tällä tarkoitetaan siis sitä, ettei ohjelmistojen laadun mittaamiseksi ole olemassa universaalisti hyväksyttyä mittausmenetelmää, vaan kehittäjät ovat itse vastuussa kvantitatiivisten tilastojen keräämisestä ohjelmistojen laadun mittaamiseksi.

Yleisimmin käytetyt tavat ohjelmistojen laadun määrittelemiseksi ovat todennäköisesti ISO 8402-1986 ja IEEE 610.12-1990 kaltaisten standardien laatutekijöiden käyttäminen, käyttäjätyytyväisyyden hyödyntäminen tai ohjelmistossa ilmenneiden virheiden ja odottamattomien käyttäytymisten havainnointi. Nämä kaikki ohjelmistojen laadun määrittelyyn käytetyt tavat perustuvat yhteiseen näkemykseen siitä, mitä ohjelmiston laadulla tarkoitetaan. [Jørgensen 1999]

Laatutekijöillä tarkoitetaan tässä yhteydessä ISO 8402-1986 -standardin määritelmän mukaan tuotteen ominaisuuksien ja piirteiden kokonaisuutta, joka vastaa tuotteeseen kohdistuviin odotuksiin. Tällaisia tekijöitä ovat muun muassa tehokkuus, joustavuus ja ylläpidettävyys. Tässä on kuitenkin huomattava se, että osa tekijöistä, kuten ylläpidettävyys, ei ole suoraan mitattavissa oleva tekijä. Tämän vuoksi täysin empiiristä yhteyttä laatutekijöiden ja tuotteen laadun välillä on mahdotonta vahvistaa. [Jørgensen 1999]

Jørgensenin [1999] mielestä tuotteen laadun mittaaminen käyttäjätyytyväisyyden avulla edellyttää yleisesti hyväksytyn tavan tyytyväisyyden identifioimiseksi. Tämä tarkoittaa sitä, että tuotteen laadun ymmärtämiseksi täytyy olla yhtenäinen tapa määritellä se, milloin tuote on saman laatuinen, parempilaatuisempi tai huonompilaatuisempi kuin verrokkituote. Tässä voidaan käyttää esimerkiksi tyytyväisten käyttäjien osuutta kaikista vastaajista mitta-asteikkona, kuten myös Naik ja Tripathy [2008] totesivat. Käyttäjätyytyväisyyden käyttäminen tuotteen laadun kuvaamiseksi on kuitenkin herättänyt ristiriitaisia tunteita, sillä tyytyväisyys on hyvin subjektiivinen käsite: tarkoittaako käyttäjän A tyytyväisyys samaa kuin käyttäjän B tyytyväisyys? Jos tuotteen käyttäjä C on ollut kehittämässä tuotetta, hänen tyytyväisyytensä lopputulokseen poikkeaa todennäköisesti pelkän loppukäyttäjän tyytyväisyydestä. Lisäksi Jørgensen [1999] myös toteaa, ettei tuotteen laatua ja käyttäjätyytyväisyyttä saisi suoraan sekoittaa keskenään.

Ohjelmistoteollisuudessa yleisimmin käytetty tapa tuotteen laadun toteamiseksi on havaittujen virheiden määrä tuotteessa. IEEE 1044-1993 standardi määrittelee virheen oletetusta poikkeavaksi tilaksi. Jotta mittaus voidaan toteuttaa, täytyy ensiksi määritellä tilanteet, jotka voidaan luokitella normaalista poikkeaviksi ja sopia, voidaanko jokin normaalista poikkeava tila luokitella useaksi virheeksi. Koska esimerkiksi kymmenen virhettä pienessä ohjelmistossa tarkoittaa eri asiaa kuin kymmenen virhettä suuressa ohjelmistossa, virheiden lukumäärä täytyy suhteuttaa ohjelmiston kokoon, esimerkiksi koodirivien määrään. Tämän avulla mittaustavasta saadaan universaali ja mahdollistetaan erilaisten ohjelmistojen laatuvertailu. Tosin tämäkään tapa ei takaa täysin ongelmatonta vertailua, sillä virheiden vakavuudet poikkeavat paljon toisistaan. Esimerkiksi kaksi väärän väristä painiketta käyttöliittymässä ei vastaa vakavuudeltaan yhtä matemaattista laskuvirhettä pankin ohjelmistossa. [Jørgensen 1999]

3.1 McCallin laatutekijät

Ohjelmistojen laadun ymmärtäminen mitattavana suureena juontaa juurensa 1970-luvun puoliväliin, kun McCall kahden ystävänsä kanssa ryhtyivät opiskelemaan konseptia ohjelmistojen laatutekijöistä ja -kriteereistä. Naik ja Tripathy [2008] lisäävät termin ”laatu” ISO 8402-1986 määritelmään sen, että laatutekijät ovat ohjelmiston ulkoisia ominaisuuksia. Kukin sidosryhmä on kiinnostunut eri laatutekijöistä ja eri laajuisesti. Asiakkaat haluavat usein tehokkaan ja luotettavan sovelluksen, kun taas kehittäjät arvostavat tuotteen ylläpidettävyyttä ja siirrettävyyttä. Laadunvarmistajat puolestaan haluavat, että sovellus olisi mahdollisimman testattava.

Naik ja Tripathy [2008] korostavat McCallin laatutekijöiden koostuvan yhdestätoista eri tekijästä:

1. Oikeellisuus
2. Luotettavuus
3. Tehokkuus
4. Eheys
5. Käytettävyys
6. Ylläpidettävyys
7. Testattavuus
8. Joustavuus
9. Siirrettävyys
10. Uudelleenkäytettävyys
11. Yhteensopivuus

Oikeellisuudella laatutekijänä tarkoitetaan sitä, että ohjelmisto täyttää eksplisiittisesti sille asetetut toiminnalliset vaatimukset ja implisiittisesti sille asetetut ei-toiminnalliset vaatimukset.

Luotettavuudella puolestaan tarkoitetaan sitä, että ohjelmisto ei päädy virhetilanteisiin, virhetilanteista toivutaan nopeasti tai jos virhetilanteet eivät vaikuta suoritettavan tehtävän toteuttamiseen. Luotettavuus on käyttäjän käsitys ohjelmistosta, jonka vuoksi täysin oikeellinen ohjelma voi olla epäluotettava. Sama pätee myös toisinpäin: virheellinenkin sovellus voi puolestaan olla luotettava.

Tehokkuus liittyy siihen, kuinka paljon sovellus käyttää resursseja, kuten laskentatehoa, muistia, levytilaa ja virtaa. Mitä vähemmän sovellus vie resursseja, sen parempi. Erityisesti tehokkuus korostuu nykypäivänä, sillä monien mobiililaitteiden määrä on kasvanut ja resurssit ovat rajalliset.

Ohjelmiston eheydellä tarkoitetaan taas sitä, kuinka luvattomien henkilöiden pääsyä ohjelmistoon tai sen tarjoamaan dataan voidaan kontrolloida tai estää. Tämäkin laatutekijä on korostunut viime aikoina internetpohjaisten palveluiden ja useita käyttäjiä tukevien ohjelmistojen myötä.

Käytettävyys on sitä, kuinka helpoksi käyttäjät kokevat ohjelmiston käytön. Sen rooli ohjelmiston laadun määrittymisessä on suuri, sillä ohjelmiston tarjoamat toiminnallisuudet ovat turhia, mikäli niitä ei osata käyttää. Toisaalta hyvä ja helppo käyttöliittymäkään ei voi korvata esimerkiksi luotettavuutta.

Ylläpidettävyys kuvastaa ohjelmiston huoltotoimenpiteiden helppoutta ja edullisuutta. Huoltotoimenpiteitä on kolmea erilaista: *parantava huolto* (corrective maintenance), *mukautuva huolto* (adaptive maintenance) ja *kehittävä huolto* (perfective mainentance). Parantavalla huollolla pyritään poistamaan ohjelmistossa huomattuja virheitä, esimerkiksi tietyn toimenpiteen aiheuttamaa ohjelmiston kaatumista. Mukautuvalla huollolla muutetaan ohjelmisto vastaamaan muuttunutta toimintaympäristöä, kuten esimerkiksi käyttöjärjestelmän vaihdoksesta aiheutuvia muutoksia. Kehittävällä huollolla puolestaan kehitetään ohjelmiston ominaisuuksia ja laatua, jos esimerkiksi asiakkaalta tulee toiveita uusista ominaisuuksista.

Ohjelmiston vaatimusten varmistaminen on tärkeä osa kehitysprosessia. Mikäli ohjelmistoa ei voida testata, on vaarana päästää rikkinäisiä toimintoja tuotantoon. Testattavuudella tarkoitetaan ohjelmiston ominaisuuksien ja vaatimusten toiminnan varmistamista ja se tulisi pitää mielessä ohjelmistokehityksen jokaisessa vaiheessa.

Joustavuus heijastuu kustannuksiin, jotka syntyvät toimivan ohjelmiston muutoksista. Mitä enemmän muutoksia tehdään joustamattomaan, mutta toimivaan ohjelmistoon, sitä enemmän ne tulevat kustantamaan. Ohjelmiston joustavuutta on helppoa mitata kysymyksellä: kuinka helppoa ohjelmistoon on tuoda uusi ominaisuus?

Ohjelmiston siirrettävyys kuvastaa sitä, kuinka helposti ohjelmiston saa toimimaan toisessa ympäristössä. Ympäristö koostuu useista eri tekijöistä, kuten laitteistosta, käyttöjärjestelmästä ja hajautuneisuudesta. Kehittäjien kannalta siirrettävyys on merkittävä laatutekijä, sillä ohjelmiston kyky tukea mahdollisimman montaa erilaista ajoympäristöä lisää sen markkinapotentiaalia. Lisäksi se tarjoaa asiakkaille joustoa, sillä he voivat vaihtaa vapaasti omia laitteistojaan ilman pelkoa ohjelmiston toimimattomuudesta.

Uudelleenkäytettävyys tarkoittaa puolestaan sitä, kuinka suurta osaa yhdestä tuotteesta voidaan hyödyntää suoraan tai pienillä muutoksilla toisessa tuotteessa. Uudelleenkäytettävyyden avulla on mahdollista säästää kustannuksissa, ajassa ja testauksessa. Uudelleenkäytettävyyden ei tarvitse välttämättä rajoittua itse tuotteeseen, vaan sitä voidaan hyödyntää myös kehitysprosessissa.

Viimeisenä laatutekijänä on yhteensopivuus, joka liittyy läheisesti uudelleenkäytettävyyteen. Yhteensopivuus on siis sitä, hyväksyykö yhden järjestelmän syöte toisen järjestelmän ulostulon. Esimerkkinä tässä tilanteessa toimii hyvin matkapuhelimet ja mobiiliverkko: saman puhelimen on kyettävä hyödyntämään useiden eri maiden mobiiliverkkoja.

Nämä yksitoista McCallin laatutekijää voidaan ryhmitellä Naikin ja Tripathyn [2008] mukaan kolmeen kategoriaan: *tuotteen toimintaan* (product operation), *tuotteen tarkastukseen* (product revision) ja *tuotteen muutokseen* (product transition). Laatutekijöistä viisi ensimmäistä (oikeellisuus, luotettavuus, tehokkuus, eheys ja käytettävyys) kuuluvat tuotteen toimintaan, kolme seuraavaa (ylläpidettävyys, testattavuus ja joustavuus) kuuluvat tuotteen tarkastukseen ja loput (siirrettävyys, uudelleenkäytettävyys ja yhteensopivuus) puolestaan tuotteen muutokseen.

3.2 ISO 9126

Asiantuntijat ympäri maailman ovat tehneet yhteistyötä määrittääkseen yleisen viitekehyksen ohjelmistojen laadulle. ISO:n (International Organisation for Standardization) alaisuudessa toimiva ryhmä standardisoi ohjelmistojen laadun ISO 9126 standardin alaisuuteen, ja se koostuu kuudesta itsenäisestä ohjelmistojen laatua kuvaavasta kategoriasta, jotka ovat:

1. Toiminnallisuus
2. Luotettavuus
3. Käytettävyys
4. Tehokkuus
5. Ylläpidettävyys
6. Siirrettävyys

Nämä kuusi kategoriaa pitävät sisällään pienempiä alapiirteitä. [Naik ja Tripathy 2008]

Toiminnallisuudella tarkoitetaan ISO 9126 standardissa ominaisuuksien kokoelmaa, joka tuottaa kokoelman toiminnallisuuksia. Nämä toiminnallisuudet puolestaan tuottavat käyttäjälle tyytyväisyyttä ja vastaavat käyttäjän tarpeisiin. Toiminnallisuuden alapiirteisiin lukeutuu soveltuvuus, tarkkuus, yhteentoimivuus ja turvallisuus. Nämä alapiirteet kuvastavat ohjelmiston kykyä toimittaa riittävän määrän toimintoja ennalta määrättyjen tehtävien hoitamiseksi, tuottaa oikea lopputulos tai seuraus, toimia yhden tai useamman ulkopuolisen järjestelmän kanssa sekä estää luvattomat yhteydet ja hyökkäykset. [Naik ja Tripathy 2008]

Luotettavuudella taas tarkoitetaan ominaisuuksien kokoelmaa, joka tuottaa ohjelmistolle kyvyn ylläpitää sen suorituskykyä tiettyjen olosuhteiden alaisuudessa ennalta määritetyn ajan. Luotettavuuden alapiirteisiin kuuluu ohjelmiston kypsyys, virheensieto ja palautuvuus. Kypsyys tarkoittaa ohjelmistoissa sitä, kuinka ohjelmisto kykenee välttämään häiriötilanteet tai kaatumisen, vaikka ohjelmistossa ilmenisi virhe. Tähän samaan aiheeseen viittaa myös virheensieto, sillä se kuvastaa ohjelmiston kykyä ylläpitää sen suorituskykyä virhetilanteissa. Palautuvuus taas kuvastaa näiden kahden alapiirteen jälkeistä tilaa kertomalla ohjelmiston kyvystä palautua toimintakykyiseksi ja palauttaa mahdollisesti hukkuneet tiedot kaatumisen jälkeen. [Naik ja Tripathy 2008]

Käytettävyys puolestaan kuvastaa ISO 9126 standardissa hyvin pitkälti samoja asioita kuin McCallin yhdessätoista laatukriteerissäkin. Se on ominaisuuksien kokoelma, joka kuvastaa ohjelmiston käyttämisestä syntyvää vaivaa. Käytettävyyteen sisältyy ohjelmiston ymmärrettävyys, opittavuus ja toimivuus. Nämä alapiirteet ovat hyvin pitkälti nimiensä mukaisia, sillä ne kuvastavat sitä, kuinka helppoa ohjelmistoa on ymmärtää, oppia ja kontrolloida sekä kuinka hyvin ohjelma toimii. [Naik ja Tripahty 2008]

Tehokkuus on myös melko yksiselitteinen tekijä, sillä yksinkertaisuudessaan se kertoo ohjelmiston suorituskyvystä ja sen vaatimien resurssien määrästä. Tehokkuutta voidaan mitata siihen sisältyvien alapiirteiden avulla. Ajankäytöllä voidaan kuvata sitä, kuinka kauan ohjelmistolla kestää vastata käyttäjälle tai kuinka kauan sillä kestää prosessoida sille annettuja tehtäviä. Resurssien käyttöasteella puolestaan mitataan sitä, kuinka paljon ohjelmisto käyttää ja varaa ulkopuolisia resursseja, kuten prosessorin suoritusaikaa ja keskusmuistia. [Naik ja Tripathy 2008]

Ylläpidettävyys taas kuvastaa ohjelmiston vaatimaa vaivaa, joka tarvitaan tiettyjen muutosten tekemiseen ohjelmiston ympäristössä, vaatimuksissa tai toiminnallisuuksissa. Nämä muutokset voivat olla esimerkiksi korjauksia, parannuksia ja mukauttamisia. Ylläpidettävyys pitää sisällään ohjelmiston analysoitavuuden, muutettavuuden, vakauden ja testattavuuden. Analysoitavuudella pyritään pitämään huolta siitä, että ohjelmisto on diagnosoitavissa mahdollisten virhetilanteiden sattuessa. Tämä tarkoittaa siis sitä, että virheen aiheuttanut tekijä on mahdollista löytää helposti. Muutettavuudella puolestaan

pyritään pitämään ohjelmisto mahdollisimman helposti muutettavana mahdollisten korjausten tai parannusten toteuttamiseksi. Vakaus kuvastaa ohjelmiston kykyä minimoida ohjelmiston yllättävät käytökset virhetilanteiden sattuessa ja testattavuus puolestaan ohjelmiston kykyä validoida siihen toteutettujen ominaisuuksien toiminta. Tässä on syytä huomata, että McCallin yhdessätoista laatutekijässä testattavuus oli korkean tason laatutekijä, kun taas ISO 9126 -standardissa se toimii alapiirteenä ylläpidettävyydessä. [Naik ja Tripathy 2008]

Viimeisenä pääkategoriana ISO 9126 standardissa on ohjelmiston siirrettävyys. Se on ominaisuuksien kokoelma, joka tarjoaa kyvyn siirtää ohjelmisto yhdestä ympäristöstä toiseen. Ympäristöllä voidaan tarkoittaa organisaatiota, laitteistoa tai ohjelmistoa. Näiden mukaan ohjelmiston ei tulisi välittää esimerkiksi siitä, missä yrityksessä sitä käytetään tai millä käyttöjärjestelmällä sitä ajetaan. Tähän pääkategoriaan kuuluvat mukautuvuus, asennettavuus, rinnakkaisuus ja korvattavuus. Näiden mukaan ohjelmiston tulisi olla siis mukautettavissa erilaisille alustoille ilman ylimääräisiä toimintoja, asennettavissa erilaisiin ympäristöihin, kyettävä elämään rinnakkain toisen itsenäisen ohjelmiston kanssa samassa järjestelmässä jakaen yhteisiä resursseja sekä kykyä käyttää samassa ympäristössä olevia muita ohjelmistoja. [Naik ja Tripathy 2008]

3.3 Laatutekijöiden valinta

Laadun mittaaminen ja sen määrittäminen on hyvin laaja konsepti, jonka vuoksi on tärkeää katsoa sitä mahdollisimman monesta erilaisesta näkökulmasta. Koska McCallin yksitoista laatutekijää ja ISO 9126 standardi keskittyvät molemmat todentamaan ohjelmistojen laatua, Naikin ja Tripathyn [2008] mielestä on luonnollista, että näiden kahden laatukokoelman välillä on selkeitä yhtäläisyyksiä. Luotettavuus, käytettävyys, tehokkuus, ylläpidettävyys ja siirrettävyys ovat molemmissa yhteisiä korkean tason piirteitä/tekijöitä, jonka vuoksi niitä voidaan pitää ohjelmiston laadun kannalta oleellisimpina tekijöinä. Naik ja Tripathy [2008] kuitenkin korostavat, että selkeää yksimielisyyttä merkittävimmistä laatutekijöistä ei ole olemassa, sillä ohjelmiston laatu on hyvin subjektiivinen käsite. ISO 9126 ja McCallin laatutekijöiden välillä on kuitenkin myös eroja: siinä missä ISO 9126 korostaa laatua tuottavien piirteiden olevan käyttäjille näkyviä, McCall mainitsee myös tuotteen sisäisen laadun. Tämän lisäksi McCallin laatutekijöiden mukaan yksi laatukriteeri voi vaikuttaa useaan laatutekijään, kun taas ISO 9126 standardissa jokainen alapiirre vaikuttaa vain yhteen laatupiirteeseen. Näiden havaintojen lisäksi on myös epäselvää mitkä laatutekijöistä ja -piirteistä ovat ylätason kriteereitä ja mitkä taas konkreettisempia kriteereitä. Esimerkiksi nykyään yhä useampi ohjelmisto on ajossa verkossa tai usean tietokoneen kommunikaatioverkostossa, jonka vuoksi ISO 9126 -standardissa määritelty alapiirre yhteentoimivuudesta voitaisiin hyvin nostaa ylätason laatupiirteeksi.

Naikin ja Tripathyn [2008] mukaan ISO 9126 ja McCall tarjoavat kattavan kokoelman laatutekijöitä, mutta niiden tarkoituksena on olla käytännössä vain suunnannäyttäjänä ohjelmiston laatua tarkkailtaessa. Kun yritys ymmärtää täysin omat tarpeensa, heidän on määriteltävä ja identifioitava itse laatutekijät, joita he tulevat tarvitsemaan ohjelmiston kehittämisessä ja jotka tyydyttävät heidän tarpeensa. Kokonaisvaltaisen ja ideaalin laadun saavuttaminen on asteittainen prosessi, jonka vuoksi on tärkeää ymmärtää edetä laadun toteuttamisessa.

4 JavaScript ja 3D

JavaScript on yksi maailman käytetyimmistä ohjelmointikielistä. Alun perin JavaScriptiä käytettiin vain lyhyiden ja yksinkertaisten skriptien kirjoittamiseen, mutta internetin räjähdysmäinen nousu ja yleistyminen 1990-luvun loppupuolelta alkaen sai JavaScriptin suosion nousemaan. Googlen sähköpostipalvelu Gmail ja karttapalvelu Google Maps 2000-luvun alkupuolella nostattivat JavaScriptin suosiota ennestään, kun selaimelle annettiin aiempaa suurempi rooli ohjelmalogiikan toteuttamisessa, eikä selaimen tehtävänä ollut enää näyttää vain palvelimelta saatua dataa. [Spencer ja Richards 2015]

JavaScript on tulkattava kieli, mikä tarkoittaa sitä, että JavaScript muunnetaan tietokoneen ymmärtämään muotoon vasta ajon aikana. JavaScript tuli ensimmäistä kertaa esille web-selain Netscape Navigator 2:n myötä, ja tuolloin sitä kutsuttiin LiveScriptiksi. Javan ollessa tuolloin pinnalla oleva kieli, Netscape päätti vaihtaa nimen houkuttelevammaksi JavaScriptiksi. Kun JavaScript lähti nousemaan, Microsoft toi omaan selaimeensa Internet Exploreriin uuden JavaScript brändin JScriptin. Siitä lähtien selainkehittäjät ovat julkaisseet JavaScriptistä kehittyneitä versioita ja sisällyttäneet ne heidän viimeisimpiin selaimiinsa. Vaikka näissä JavaScriptin versioissa on paljon yhteistä, yhteensopivuusongelmia eri selainten välillä voi silti ilmetä. Tämän vuoksi web-kehitys JavaScriptillä vaatii erityistä huolellisuutta, mikäli ohjelman halutaan toimivan jokaisella selaimella. [Wilton ja McPeak 2007]

Wilton ja McPeak [2007] toteavat jo vuonna 2007 JavaScriptin eduksi sen laajan levinneisyyden, kuten myös Spencer ja Richards [2015] lähes kymmenen vuotta myöhemminkin. Tämän lisäksi Wilton ja McPeak [2007] nostavat JavaScriptin hyödyiksi helpon saatavuuden, monipuolisen selaintuen ja mukautuvuuden myös muuhun kuin web-kehitykseen.

Mozillan [2019] mukaan JavaScript pohjautuu Ecma Internationalin standardisoimaan ECMAScriptiin (ES), jota ylläpidetään Ecma Internationalin kehittämissä ECMA-262 ja ECMA-402 spesifikaatioissa. Ensimmäinen ECMAScriptin versio (ES1) julkaistiin jo vuonna 1997, jolloin määriteltiin ECMAScriptin yleinen tarkoitus toimia JavaScriptin spesifikaationa ja järjestelmäriippumattomana ohjelmointikielenä. Toinen versio (ES2) julkaistiin vuotta myöhemmin, ja tässä versiossa tehtiin vain pieniä muutoksia spesifikaatioihin. Kolmas versio (ES3) julkaistiin vuonna 1999 ja se esitteli muutamia uusia ominaisuuksia. Tämä versio vastaa JavaScript 1.5:tä. Tämän version jälkeen meni kymmenen vuotta, kunnes vuonna 2009 julkaistiin viides versio (ES5) ECMA-262:sta. Tässä versiossa spesifioitiin JavaScriptiin *strict mode*, eli kehitystila, joka mahdollistaa siistimmän koodin kirjoittamisen esimerkiksi estämällä julistamattomien muuttujien käyttämisen. ES5 on tuettuna tällä hetkellä lähes jokaisessa nykyaikaisessa selaimessa. Vuonna 2015 julkaistiin ES6, joka toi mukanaan paljon uusia ominaisuuksia, kuten luokat, symbolit, uusia tietorakenteita ja nuolifunktiot. Tämän jälkeen ECMAScriptistä on

tullut vuosittain uusia versioita, joissa on tuotu mukana uusia spesifikaatioita JavaScriptille.

4.1 WebGL

Arora [2014] tiivistää *Web Graphics Libraryn* (WebGL) voittoa tavoittelemattoman konsortion, Khronos Groupin, tarjoamaksi JavaScript rajapinnaksi 2D ja 3D grafiikan renderointiin, joka pohjautuu ensisijaisesti mobiililaitteille suunniteltuun OpenGL ES 2.0:aan. Se on yksinkertainen, suoraviivainen ja nopea ohjelmointirajapinta, jonka avulla on mahdollista piirtää vaativampia 3D-malleja HTML:n canvas-elementille. WebGL:n reaaliaikainen renderointi avasi uusia mahdollisuuksia web-pohjaiseen videopelaamiseen ja tieteelliseen visualisointiin. Renderoinnilla tarkoitetaan prosessia, jossa mallista generoidaan kuva. Renderointi voidaan jaksaa kahteen prosessointitapaan: *ohjelmistopohjaiseen renderointiin* (software-based rendering) ja *laitteistopohjaiseen renderointiin* (hardware-based rendering). Mikäli renderoidessa käytetään tietokoneen pääprosessoria, puhutaan ohjelmistopohjaisesta renderoinnista, kun taas laitteistopohjaisessa renderoinnissa hyödynnetään tietokoneen näytönohjainta. Laitteistopohjaisen renderoinnin on todettu olevan huomattavasti tehokkaampi, sillä siinä renderointioperaatioita hoitaa erillinen, siihen tarkoitukseen soveltuva komponentti. [Cantor ja Jones 2012]

3D-mallin (meshin) jokainen taho on nimeltään *polygoni* (polygon). Polygoni koostuu kolmesta tai useammasta kulmasta, joita kutsutaan *vertekseiksi* (vertex). Muistin säästämiseksi jokainen verteksi indeksoidaan taulukkoon, polygonit pyrkivät käyttämään toistensa verteksejä ja renderoidessa pyritään käyttämään mahdollisimman vähäkulmaisia polygoneja. Mitä vähemmän polygonissa on kulmia, sitä sulavampi ja luontevampi pinta mallille on mahdollista toteuttaa. [Arora 2014]

Aroran [2014] mukaan mallit on ajettava renderointiputkiston läpi, ennen kuin ne ovat käyttäjän nähtävillä ruudulla. Ensiksi jokainen verteksi muunnetaan omaan sijaintiinsa, jossa otetaan huomioon myös kameran sijainti. Jokainen verteksi joka jää katselualueen ulkopuolelle leikataan. Tämän jälkeen määritellään mallin lopullinen koko ja sijainti ja se renderoidaan ruudulle. Nykyiset näytönohjaimet käyttävät ohjelmoitavia renderointiputkistoja, jotka mahdollistavat renderointiin liittyvän manipuloinnin ja omien funktioiden kirjoittamisen.

Mikäli WebGL:ää verrataan muihin grafiikkateknologioihin, kuten Java 3D:hen tai Flashiin, WebGL tarjoaa muutamia etuja. Ensiksikin WebGL:n muistinhallinta on automaattista, mikä tarkoittaa sitä, että se vapauttaa muistia sitä mukaa kun ei enää tarvitse sitä. Toiseksi, WebGL on hyvin laaja-alainen, sillä sitä tukee lähes jokainen JavaScriptiä tukeva selain. Tällaisia selaimia on nykyään lähes jokainen niin työpöytä- kuin mobiiliselaimistakin. Kolmanneksi, WebGL on tehokas sen laitteistopohjaisen renderoinnin hyödyntämisen myötä, eli se osaa käyttää näytönohjainta, mikäli sellainen on saatavilla.

Näiden lisäksi WebGL ei vaadi kääntämistä, koska se on toteutettu JavaScriptillä. Tämän vuoksi WebGL:n ohjelmointi on nopeaa, sillä muutoksia voidaan tehdä lennosta ja ne näkyvät heti selaimessa. [Cantor ja Jones 2012]

WebGL ohjelma vaatii neljä komponenttia luodakseen 3D skenen: HTML:n canvas-elementin, vähintään yhden 3D-mallin, valaistuksen ja kameran. Canvas-elementti toimii ikään kuin näyttämönä, johon 3D skene tullaan renderoimaan. 3D-mallit voidaan kuvitella näyttelijöinä, jotka lisätään näyttämölle, eli canvakselle kaikkien nähtäväksi. Jotta 3D-mallit saadaan näkyville, tulee näyttämölle lisätä valaistus. Valaistuksen avulla täysin mustalle canvakselle tuodaan valoa ja sen avulla 3D-malleille saadaan luotua heijastuksia ja fysiikan lakien mukaisia ominaisuuksia. Lopuksi tarvitaan kamera, jonka kautta kaikkea tätä voidaan katsoa erilaisista perspektiiveistä. [Cantor ja Jones 2012]

4.2 TypeScript

Vaikka JavaScriptin sanotaan olevan helposti lähestyttävä ja opittava ohjelmointikieli, kontrolli koodin ylläpidettävyydestä on myös helppo menettää ohjelman kasvaessa yhä suuremmaksi ja monimutkaisemmaksi. Osittain tästä syystä kehitettiin avoimen lähdekoodin TypeScript, jonka pääasiallinen tarkoitus on tarjota JavaScriptille tyypitys muiden olio-ohjelmointikielellisten piirteiden lisäksi. Tyypityksen avulla JavaScript kehityksestä pyritään tekemään helpompaa ja luotettavampaa, vaikka ohjelmakoodin määrä kasvaisikin. JavaScriptin ollessa tulkattava kieli, TypeScript on käännettävä ennen ohjelman suorittamista. [Nance 2014]

Nance [2014] vakuuttaa TypeScriptin käyttöönoton olevan yksinkertaista jo valmiissa JavaScript projekteissa, sillä se kääntyy puhtaaksi JavaScriptiksi eikä JavaScriptillä kirjoitettua koodia ole pakollista muuttaa TypeScriptiksi. TypeScriptin käyttöönotto ei siis pakota ohjelmaa käyttämään sen tarjoamia ominaisuuksia, vaan niitä voidaan tuoda mukaan projektiin omaan tahtiin ja osa-alueille, joihin sen itse haluaa.

TypeScript tarjoaa etenkin olio-ohjelmointiin tottuneille kehittäjille kattavan paketin tyyppejä ja käyttöä helpottavia tasoja, kuten luokat ja rajapinnat. Perintä on kuitenkin TypeScriptissä prototyyppipohjainen kuten JavaScriptissäkin, eikä luokkapohjainen kuten useimmissa olio-ohjelmointikielissä. Tyypityksen ja käännettävyyden ansiosta mahdolliset koodivirheet saadaan kiinni aikaisessa vaiheessa, luokat helpottavat suurten projektien kehittämistä ja rajapinnat puolestaan mahdollistavat laajalti käytettävien komponenttien ja kirjastojen luomisen. Näiden lisäksi TypeScript tarjoaa näkyvyydet rajoittamaan objektien välisiä interaktioita. [Nance 2014]

Koodikatkelma 1 sisältää vertailun JavaScriptin ja TypeScriptin välillä merkkijonon, numeron tai totuusarvon sisältävän muuttujan luomisessa. Siinä missä JavaScript ei vaadi käyttäjäänsä kertomaan muuttujatyyppiä, TypeScript mahdollistaa tämän tiedon antamisen ja näin ollen helpottaa muuttujan myöhempää käyttöä tarjoamalla enemmän tietoa muuttujasta ja sen tarjoamista funktioista. Katkelmasta on syytä huomata myös se, että

molemmat tavat määritellä muuttujia on sallittu samassa tiedostossa, sillä kuten Nance [2014] toteaa, TypeScript kääntyy JavaScriptille ja näin ollen hyväksyy myös sen kirjoittamisen samassa tiedostossa.

```
1  // JavaScript muuttujat
2  let jsString = 'string muuttuja';
3  let jsNumber = 123;
4  let jsBoolean = true;
5
6  // TypeScript muuttujat
7  let tsString: string = 'string muuttuja';
8  let tsNumber: number = 123;
9  let tsBoolean: boolean = true;
```

Koodikatkelma 1. Vertailu muuttujien määrityksestä.

Funktioiden tyyppimäärittely tapahtuu vastaavalla tavalla kuin muuttujienkin tyyppimäärittely. Mikäli funktio ei palauta mitään arvoa, sen tyypiksi määritetään *void*. Muussa tapauksessa tyypiksi määritetään funktion palauttaman arvon tyyppi. Koodikatkelmassa 2 näkyy, kuinka TypeScriptillä toteutetun funktion parametrille määritetään tyypiksi *string* ja kuinka funktio palauttaa arvon, jonka tyyppi on *number*.

```
1  // JavaScript funktio
2  function jsFoo(bar) {
3    return 123;
4  }
5
6  // TypeScript funktio
7  function tsFoo(bar: string): number {
8    return 123;
9  }
```

Koodikatkelma 2. Funktioiden vertailu.

Yksi TypeScriptin ydinpiirteistä on sen kyky tarjota *rajapintoja* (interface) omien tyyppien luomiseksi. Nämä rajapinnat toimivat eräänlaisina sopimuksina siitä, minkälaisia ominaisuuksia objektit tarjoavat ohjelman sisällä ja sen ulkopuolella. Jos ulkopuoliselta palvelimelta ladataan esimerkiksi henkilötietoja, ei puhtaalla JavaScriptillä toteutetulla ohjelmalla ole tietoa siitä, missä muodossa nämä tiedot tulevat, mitä ominaisuuksia ne tarjoavat ja kuinka niitä tulisi käsitellä. Koodikatkelmassa 3 toteutetaan henkilötietoja vastaava rajapinta, jonka avulla palvelimelta ladatut henkilötiedot voidaan käsitellä aina

ennalta määritetyssä muodossa. Tämän rajanpinnan mukaan palvelimelta ladatuista henkilötiedoista tulee löytyä aina henkilön nimi ja ikä. Kysymysmerkki *Person*-rajapinnan *age*-ominaisuuden perässä tarkoittaa sitä, että ominaisuus on vapaaehtoinen, eli sen arvo voi olla myös *null*. Toteutettua rajapintaa voidaan käyttää esimerkiksi muuttujien ja funktioiden tyyppimäärittelyissä nyt samalla tavalla kuin mitä tahansa muutakin tyyppiä. [TypeScript Documentation 2019]

```
1  // TypeScript Person rajapinta
2  interface Person {
3    name: string;
4    age?: number;
5  }
```

Koodikatkelma 3. Henkilöä kuvaava rajapinta.

4.3 Angular

Spencerin ja Richardsin [2015] mukaan JavaScriptin yleistyttyä ja sen roolin kasvaessa sen ympärille alkoi syntyä useita työkaluja ja viitekehyksiä ohjelmoinnin helpottamiseksi, joista yksi oli AngularJS. AngularJS on Googlen tarjoama avoimen lähdekoodin client-puolen viitekehys, joka on kirjoitettu JavaScriptillä. Sen ensimmäinen versio (1.0) julkaistiin kesäkuussa 2012, mutta sen kehittäminen aloitettiin jo vuonna 2009. AngularJS tarjoaa huomattavia etuja modernien *single-page application* (SPA) -arkkitehtuuria noudattavien websovelluksien kehittämiseen, sillä sen avulla *dokumenttioliomallit* (document object model, DOM) eivät vaadi eksplisiittistä päivittämistä, vaan se kykenee tarkkailemaan käyttäjän toimintoja, selaimen tapahtumia ja mallien muutoksia päivittääkseen vain tarvittavat sivupohjat oikeaan aikaan. AngularJS tarjoaa myös mielenkiintoisen ja laajennettavan komponenttialajärjestelmän sekä mahdollisuuden tulkata uusia HTML-tageja ja ominaisuuksia. Näiden lisäksi AngularJS sisältää sisäänrakennetun tuen *riippuvuusinjektiolle* (dependency injection, DI), jonka avulla websovellus on helppoa rakentaa pienistä ja helposti testattavista yksittäisistä palveluista. [Darwin ja Kozlowski 2013]

Darwin ja Kozlowski [2013] ylistävät yhtä AngularJS:n ehdottomista eduista: sen ympärillä olevaa yhteisöä ja yhteisön tarjoamaa tukea. AngularJS:llä on muun muassa oma sähköpostilista, IRC-kanava, oma tagi Stackoverflowssa, blogi sekä Twitter-tili, joiden kautta käyttäjät voivat kommunikoida, pyytää apua ja keskustella ongelmatilanteista.

Vaikka AngularJS on hyvin vakaalla pohjalla oleva viitekehys ja sitä käyttää jo miljoonat web-kehittäjät, sen kehitys päätettiin siirtää viimeisimmän version (1.7) jälkeen kolmen vuoden mittaiseen *pitkäaikaistukeen* (long term support, LTS). Tällä tarkoitetaan sitä, ettei AngularJS:ään tehdä enää pieniäkään rikkovia muutoksia sisältäviä päivityksiä,

vaan keskitytään tekemään korjauksia, mikäli huomataan vakavia yhteensopivuus- tai turvallisuusongelmia. AngularJS:n seuraaja Angular, jonka ensimmäinen versio (2.0) julkaistiin Kremerin [2016] mukaan syyskuussa 2016 jakaa kuitenkin saman perusfilosofian AngularJS:n kanssa. Se on kasvanut viisi kertaa nopeammin kuin AngularJS julkaisun jälkeen ja noin vuoden jälkeen julkaisusta sen käyttäjäkunnan koko ohitti AngularJS:n käyttäjäkunnan. Angularin kehitys on kuitenkin vielä hyvällä mallilla, ja sen kehittäjät ovatkin luvanneet jatkaa Angularin päivittämistä tulevaisuudessakin. [Darwin 2018]

Hartmann [2017] kertoo artikkelissaan Angularin olevan viisi kertaa nopeampi kuin AngularJS. Tämän voidaan nähdä johtuvat Angulariin tuodusta *ahead-of-time compilation* (AOT) ominaisuudesta, joka kääntää sovelluksen ennen kuin lähettää sen selaimelle. Tämän seurauksena selaimet voivat suorittaa sivuston nopeammin, koska Angularin kääntäjä paketoi sovelluksen pienempään tilaan kuin AngularJS:n kääntäjä. Jotta pakettikoko saatiin pienemmäksi, Angular jätti joitain paketteja, kuten animaatiopaketin, pois Angularin ydinpaketista. Pakettikoon pienentymisen ja siitä seuraavan nopeuden lisäksi tämä parantaa Angularin turvallisuutta verrattuna AngularJS:ään.

Hartmannin [2017] mukaan Angular on toteutettu mobiilikäyttöisellä lähestymistavalla ottamalla mallia muista mobiilikielistä, kuten React Nativesta. Esimerkiksi *laiska lataus* (lazy loading) mahdollistaa Angularin käyttämien moduulien kääntämisen vasta silloin kun niitä käytetään, joka puolestaan tekee ohjelmakoodista kevyemmän ja nopeammin latautuvan, kun kaikkea ohjelmakoodia ei tarvitse kääntää heti sovelluksen käynnistyessä. Näiden ominaisuuksien lisäksi migraatio AngularJS:stä Angulariin on vaiheittain, eli vaihtoa ei ole pakko tehdä yhdellä kertaa.

Vaikka varsinaista tutkimusta Angularin räjähdysmäiseen suosioon AngularJS:ään verrattuna ei ole saatavilla, edellä lueteltujen ominaisuuksien voidaan nähdä olevan syy sen suosioon. Etenkin suorituskykyyn liittyvät tekijät ovat kehittäjille hyvin läheisiä, sillä suuri osa web-kehityksestä on ohjelmakoodin kääntämistä. Mitä vähemmän tähän kuluu aikaa, sitä miellyttävämpää ja nopeampaa sovelluksen kehittäminen on. Lisäksi Angularin mobiilikäyttöinen lähestymistapa tarjoaa yhden mahdollisen syyn lisää Angularin suosioon, sillä erilaisten mobiilialustojen kasvu on ollut nopeaa viime vuosikymmenen aikana.

4.4 3D-kirjastot

Viimevuosien aikana selaimet ovat voimistuneet ja niiden kyky esittää monimutkaisiakin sovelluksia ja grafiikoita on parantunut huomattavasti. Suurin osa näistä grafiikoista on edelleen 2D-grafiikkaa, mutta kuten kohdassa 4.1 mainittiin, niin useimmat nykyaikaiset selaimet ovat ottaneet käyttöönsä WebGL:n, jonka avulla myös 3D-grafiikkaa on mahdollista näyttää selaimessa. Ohjelmointi suoraan WebGL:ään on kuitenkin erittäin monimutkaista, sillä se vaatii WebGL:n sisäisten yksityiskohtien tuntemisen ja monimutkaisen

varjostinkielen (shader language) ymmärtämistä, jotta WebGL:stä saisi kaiken sen tarjoaman hyödyn irti. Tästä syystä JavaScriptille on kehitetty erilaisia 3D-kirjastoja, joiden kautta WebGL ohjelmointia pyritään yksinkertaistamaan ja helpottamaan. [Dirksen 2013]

JavaScriptille on tarjolla lukuisia erilaisia 3D-kirjastoja, mutta vertailuun valikoitui avoimeen lähdekoodiin perustuvat Three.js ja Babylon.js. Nämä valikoituivat vertailtaviksi siitä syystä, että molemmat ovat aktiivisesti kehittyviä kirjastoja, joilla on suuri määrä edistäjiä GitHubissa.

Three.js on avoimen lähdekoodin 3D-kirjasto JavaScriptille, joka toimii lähes jokaisella nykyaikaisella selaimella. Three.js:n on alun perin luonut Ricardo Cabello ja sen tarkoituksena on tarjota käyttäjälle helppokäyttöinen ja kevyt 3D-kirjasto, joka tarjoaa Canvas 2D, SVG, CSS3D sekä WebGL renderoijat. Sen suosio on kehittynyt suuresti, ja nykyään sitä käytetään jopa suurten brändien ja artistien nettisivuilla luodakseen käyttäjille mukaansatempaavan kokemuksen. [Sukin 2013]

Three.js:llä on monipuolinen yhteisötuki erilaisten sosiaalisten kanavien kautta, joissa käyttäjät voi ilmoittaa ongelmista, kysyä kysymyksiä ja tarjota apua. Näitä sosiaalisia kanavia ovat muun muassa Slack, IRC, Three.js:n oma foorumi, GitHub ja oma tagi Stackoverflowssa. Näiden lisäksi Three.js tarjoaa myös esimerkkejä ja demoja erilaisten 3D-mallien toteuttamiseksi. [Three.js 2019]

Moreau-Mathis [2016] kertoo Babylon.js:n olevan 3D-kirjasto, joka mahdollistaa niin 3D-websovellusten kuin -pelienkin luomisen. Sen on alun perin luotu Microsoftilla työskentelevien ohjelmistoalan asiantuntijoiden toimesta heidän vapaa-ajallaan, ja se on suunniteltu helposti lähestyttäväksi ja kaikkien saatavilla olevaksi 3D-moottoriksi. Koska Babylon.js on Microsoftilla työskentelevien ohjelmistokehittäjien ja -suunnittelijoiden käsialaa, se on myös toteutettu Microsoftin kehittämällä TypeScriptillä. Kuten Three.js, myös Babylon.js tarjoaa käyttäjilleen monipuolisen yhteisötuen. Babylon.js:llä on kattavasti esimerkkejä ja ohjeita erilaisista 3D-sovelluksista, käyttäjille suunnattu foorumi kysymyksiä, bugiraportteja, esimerkkejä ja ilmoituksia varten, GitHub lähdekoodin tarkasteluun sekä myös oma tagi Stackoverflowssa. [Babylon.js dokumentaatio 2019a]

5 Wisehockey

Wisehockey on tamperelaisen Bitwise Oy:n toteuttama älykiekkojärjestelmä, joka kykenee analysoimaan jääkiekkopelin kulkua reaaliajassa ja automaattisesti. Jääkiekko-otteluiden tilastointi ja analysointi on manuaalisesti toteutettuna hyvin rajallista ja hidasta, minkä vuoksi yksinkertaisten pelitilastojen laskenta on kestänyt jopa tunteja. Ammattiurheilussa harjoittelun tueksi tarvitaan yhä tarkempaa ja yhä enemmän helposti saatavilla olevaa dataa, joiden avulla valmentajat ja pelaajat kykenevät entistä parempiin suorituksiin. Harjoittelun ja valmennuksen lisäksi tätä samaa dataa voidaan hyödyntää fanien kokemuksien rikastuttamisessa tarjoamalla esimerkiksi mielenkiintoisia ottelutilastoja erätauoilla [Bitwise 2019].

5.1 Quuppa

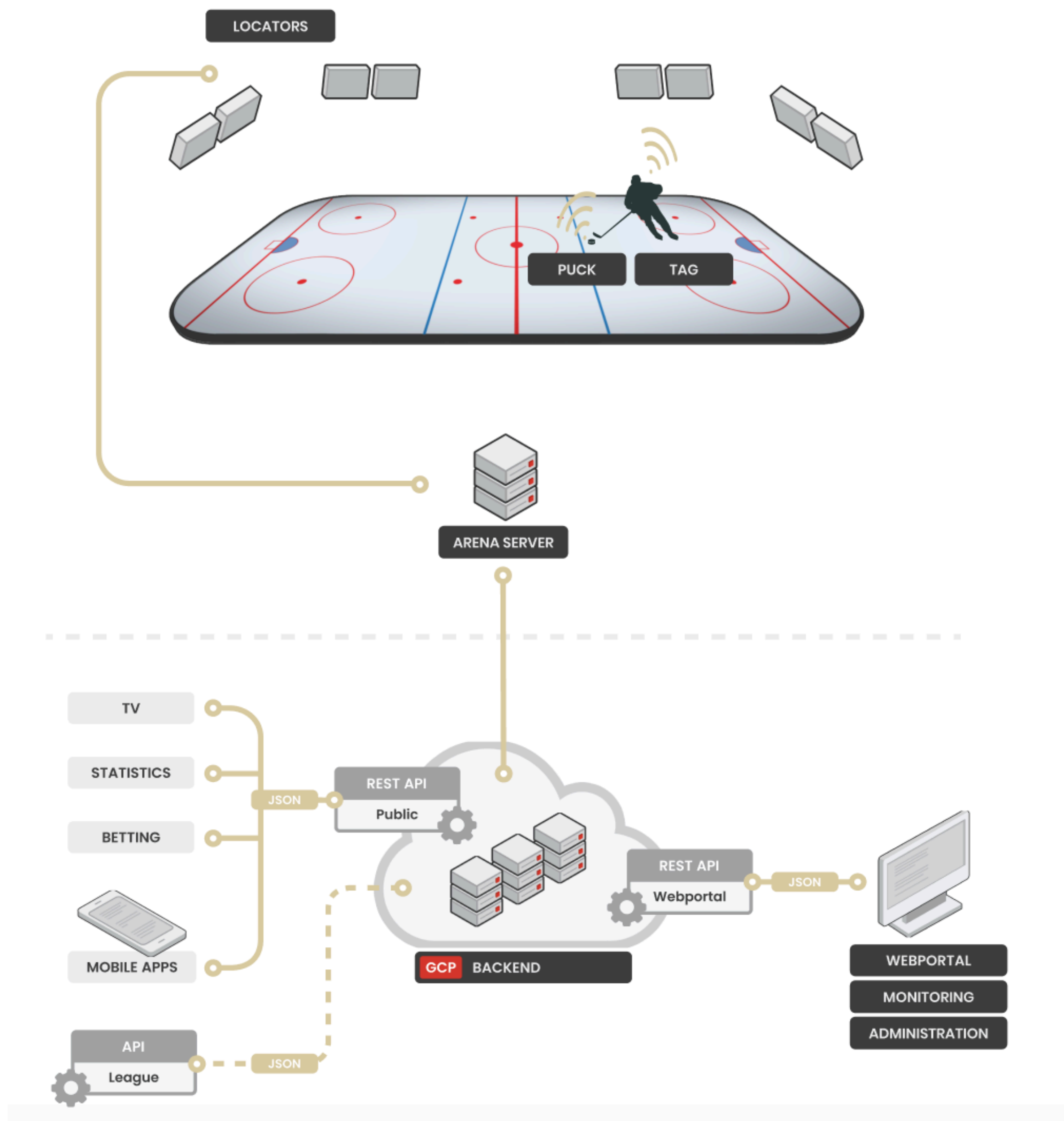
Quuppa Oy on suomalainen teknologiayritys, jonka tarjoama *Quuppa Intelligent Location System* mahdollistaa reaaliaikaisen seurannan jopa muutaman senttimetrin tarkkuudella. Quupan tarjoama järjestelmä perustuu *angle-of-arrival* (AOA) paikannusmenetelmään ja se käyttää paikannukseen BLE-teknologiaan pohjautuvaa laitteistoa. Järjestelmä on käytössä tällä hetkellä ympäri maailman erilaisissa ympäristöissä, kuten logistiikassa, terveydenhuollossa ja urheilussa. [Quuppa 2019]

Quuppa [2019] lupaa järjestelmänsä kykenevän alle kymmenen senttimetrin tarkkuuteen myös nopeasti liikkuvien kohteiden kohdalla ja haastavissa ympäristöissä. Järjestelmä kykenee seuraamaan niin tageja kuin älypuhelimiakin ja tarjoaa näiden avulla paikannusta, arviointia tai pelkästään olemassaolon havainnointia. Järjestelmän mainostetaan olevan edullinen, helppokäyttöinen, pitkäikäinen ja huoltovapaa.

5.2 Arkkitehtuuri

Wisehockey-järjestelmän arkkitehtuuri perustuu kahteen pääelementtiin: Quupan tarjoamaan paikannusjärjestelmään ja reaaliaikaiseen analytiikkaan. Wisehockey-järjestelmän peruslaitteisto koostuu lokaattoreista (locators), pelaajatageista (tag), älykiekoista (puck) ja palvelimesta (arena server), jotka ovat nähtävillä kuvassa 1. [Bitwise 2019]

Wisehockey-järjestelmä koostuu kolmesta tasosta, joista ylimmällä tasolla on hallin kaukalo ja ottelun seuranta. Ensiksi lokaattorit asennetaan hallin kattoon kaukalon yläpuolelle ja ne kalibroidaan kaukalon mittojen mukaisiksi. Kun lokaattorit ovat asennettu ja kalibroitu, ne ovat kykeneviä seuraamaan kaukalossa liikkuvia BLE-tageja, jotka ovat sijoitettuina pelaajiin ja kiekkoon. Lokaattorit keräävät sijaintidataa 50 millisekunnin välein tageista ja lähettävät sen arkkitehtuurissa seuraavalle tasolle paikalliselle palvelimelle.



Kuva 1. Wisehockey-järjestelmän arkkitehtuuri.

Hallilla sijaitseva palvelin käsittelee lokaattoreilta saatua dataa, jonka jälkeen se lähettää käsitellyn datan Googlen tarjoamaan *Google Cloud Platform* (GCP) pilvilaskentapalveluun, jossa on ajossa Wisehockey-järjestelmän backend, eli käyttäjältä piilossa oleva ohjelmisto, joka hoitaa käyttäjälle esitettävän datan käsittelyn käyttäjän ymmärtämään muotoon. Backendin tehtävänä on hoitaa tämä data sitä käyttävien tahojen saataville, kuten esimerkiksi TV-lähetyksiin, vedonlyöntiin, mobiilisovelluksiin ja jokaiselle järjestelmää hyödyntävän joukkueen Webportaaliin, jonka kautta joukkueet ja niiden parissa työskentelevät toimihenkilöt voivat tarkastella joukkueensa tilastoja. Tämä kerätty data toimitetaan *REST-arkkitehtuurimallia* (representational state transfer) hyödyntävien *ohjelmointirajapintojen* (application programming interface, API) avulla *JSON*-muotoisena (JavaScript Object Notation). Webportalista kerrotaan tarkemmin alakohdassa 5.2.1.

Backend käyttää työskentelyssään myös Wisehockey-järjestelmää hyödyntäviä jääkiekkoliigoja, ja hakee esimerkiksi joukkueiden ottelukokoonpanot liigojen tarjoamien ohjelmointirajapintojen kautta.

5.2.1 Webportal

Webportal on Wisehockey-järjestelmään kuuluva web-pohjainen portaali, jonka kautta Wisehockey-järjestelmää käyttävät tahot voivat tarkastella kerättyjä tilastoja käyttäjäystävällisesti graafisen käyttöliittymän kautta (kuva 2). Webportaaliin kootaan jokaisen älykiekolla pelatun ottelun tilastot, joita voidaan tarkastella koko kauden mittakaavassa tai pelkästään yhden valitun ottelun osalta. Koko kauden osalta Webportaalista nähdään joukkueen perustilastot, jotka ovat:

- pelatut ottelut,
- voitot, häviöt, ja tasapelit,
- pisteet ja pisteprosentti,
- jatkoaikavoitot ja -häviöt,
- tehdyt ja päästetyt maalit yhteensä, sekä per ottelu,
- yli- ja alivoimaprosentti,
- laukaukset per ottelu ja
- aloitusvoittoprosentti.

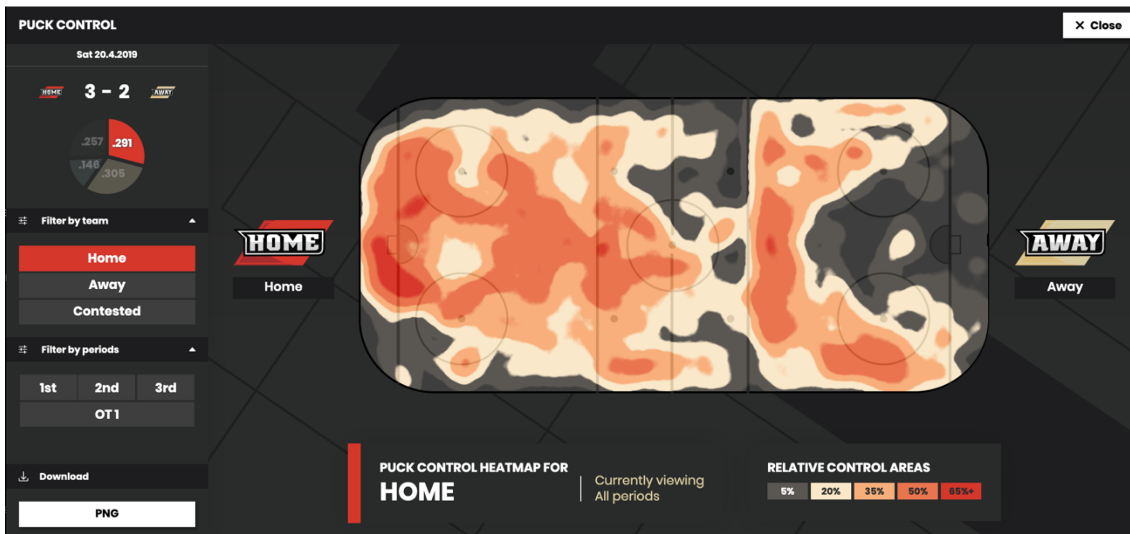
Kuva 2. Webportaalin etusivunäkymä.

Lisäksi koko kauden osalta on mahdollista tarkastella myös pelaajien tilastoja, ja jokaisesta pelaajasta on nähtävissä seuraavat tilastot:

- maalit, syötöt ja pisteet,
- +/- -tilastot,

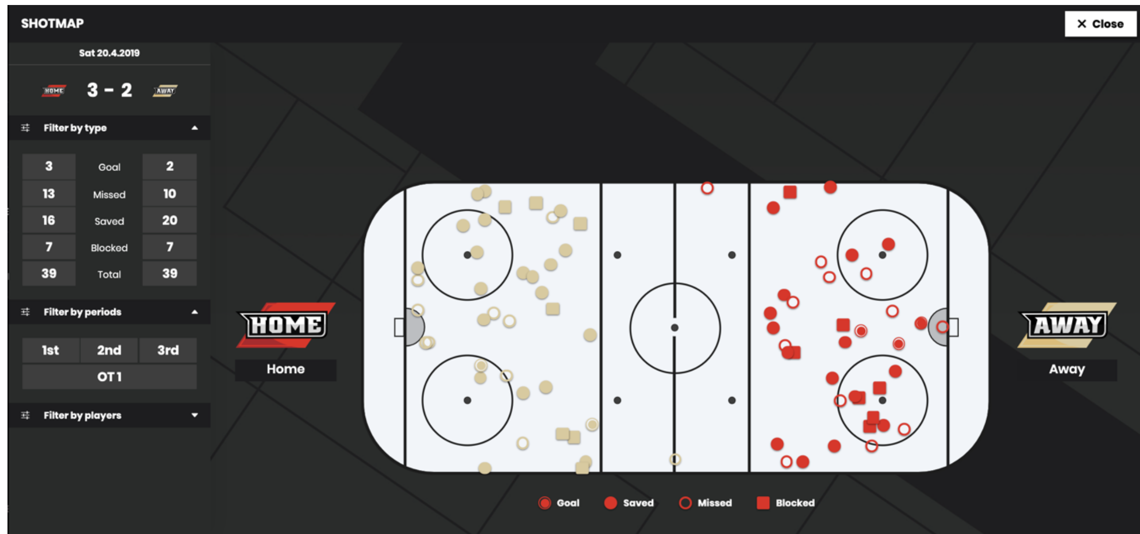
- jäähyminuutit,
- laukaukset ja laukaisuprosentti,
- aloitusvoitot ja voittoprosentti,
- jääaika ja kuljettu matka ja
- huippu- ja keskinopeus.

Nämä samat pelaajista kerätyt tilastot ovat tarkasteltavissa myös ottelukohtaisesti ja niistä on mahdollista ladata PDF- tai CSV-tiedosto muunlaista tarkastelua varten. Jokaisesta ottelusta on mahdollista tarkastella myös ottelukohtaisia tilastoja, ja ne poikkeavat hieman koko kaudelta kerätyistä tilastoista. Ottelukohtaisia ottelutilastoja ovat muun muassa kiekkokontrolli ja laukaisukartta. Kiekkokontrolli kertoo sen, kuinka kiekkoa on hallittu ottelun aikana. Kiekkokontrolli kuvataan Webportaalissa prosentuaalisesti, ajallisesti ja lämpökarttana, joista lämpökartta on mahdollista ladata myös omalle koneelle muuta käyttöä varten (kuva 3).



Kuva 3. Lämpökartta esimerkkiottelun kiekkokontrollista.

Laukaisukartta on karttakuva siitä, kuinka ottelussa tapahtuneet laukaukset jakautuvat kentällä. Laukaisukartasta on mahdollista *filtteröidä*, eli suodattaa kummankin joukkueen osalta laukaukset, jotka ovat menneet maaliin, menneet ohi maalin, tulleet torjutuksi tai tulleet blokatuksi. Lisäksi laukaisukartasta on mahdollista suodattaa ottelussa pelannei-den pelaajien laukauksia.



Kuva 4. Esimerkkiottelun laukaisukartta.

5.2.2 Wiseplayer

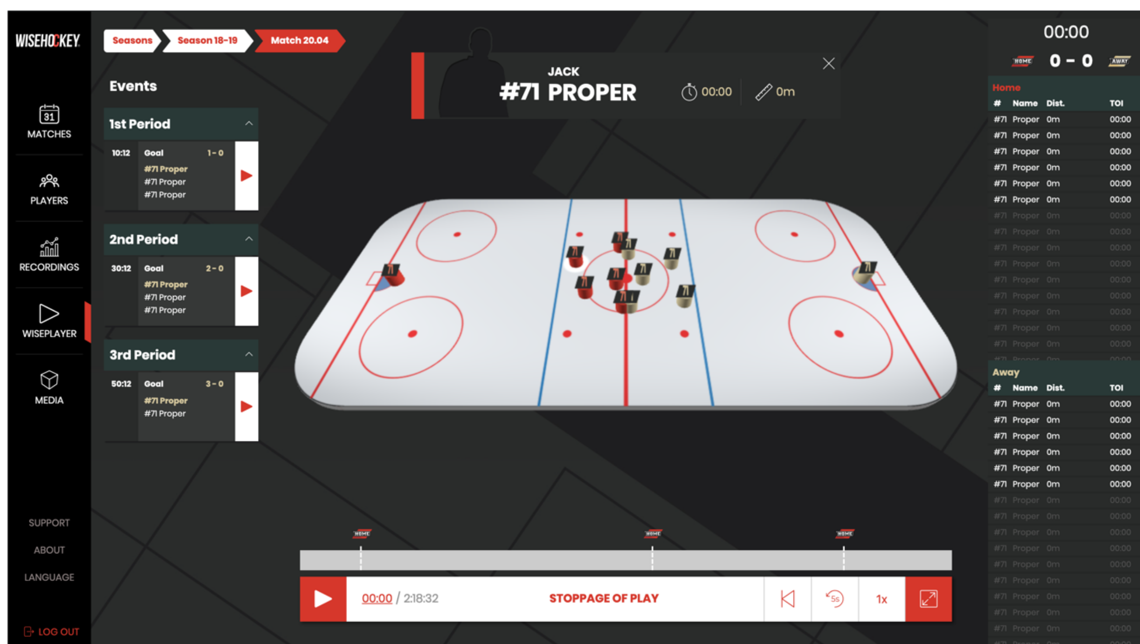
Wiseplayer on Webportalista löytyvä 3D-mediatoistin, jonka avulla voidaan visualisoida Wisehockey-järjestelmää hyödyntäviä jääkiekko-otteluita kolmiulotteisten mallien avulla (kuva 5). Kun Wisehockey-ottelu aloitetaan, siitä on mahdollista luoda tallenne. Tallenteeseen tallennetaan kiekon ja jokaisen pelaajan sijaintitiedot koko ottelun ajalta, jolloin ottelu on mahdollista toistaa myös jälkikäteen. Wiseplayer hyödyntää näitä tallenteita käymällä läpi tallenteen jokaisen sijaintia kuvastavan datapisteen ja liikuttamalla HTML-canvaksella piirrettyjä 3D-malleja näiden datapisteiden perusteella.

Wiseplayer on toteutettu TypeScriptillä ja Angular 6:lla, ja se hyödyntää Three.js 3D-kirjastoa datan visualisoinnissa. Wiseplayerin käyttämät tämän hetkiset mallit ovat toteutettu avoimen lähdekoodin 3D-mallinnusohjelma Blenderillä ja ne ovat hyvin yksinkertaisia: kotijoukkueen pelaajat ovat punaisia, vierasjoukkueen pelaajat kultaisia ja tuomarit mustavalko-raidallisia lieriöitä. Kiekko on matala musta lieriö kuten reaalimaailmassakin, ja kaukalo on kaukalon muotoinen malli, jonka pintaan on liitetty tekstuuri erottamaan eri kaukalon osat ja elementit, kuten puna- ja siniviivat sekä B-pisteet. Nämä mallit ovat nähtävillä kuvasta 5.

Wiseplayerin käyttäminen aloitetaan valitsemalla ensiksi tallenne, jota halutaan katsoa. Wiseplayerin alareunassa on ottelun aikajana, josta näkee ottelussa syntyneet maalit ja kuinka ne asettuvat aikajanalle. Maalimerkintää kuvastaa maalin tehneen joukkueen logo, jota painamalla Wiseplayer-tallenne siirtyy maalin tapahtuma-ajanhetkeen. Aikajanan alapuolelta löytyy toimintapalkki, jossa on Wiseplayerin varsinaiset toiminnallisuudet. Toimintapalkin vasemmanreunimmainen painike käynnistää ja pysäyttää tallenteen toistamisen. Tämän painikkeen vierestä löytyy tallenteen nykyinen ajanhetki, sekä tallenteen kokonaispituus. Toimintapalkin keskellä oleva ”Stoppage of play” -teksti ilmoittaa,

onko ottelussa pelikatko vai ei. Sen oikealta puolelta löytyvä painike kelaa tallenteen alkuun, kun taas sen viereinen painike kelaa tallennetta viisi sekuntia taaksepäin. Toiseksi reunimmainen painike oikealta katsottuna avaa valikon, josta voidaan valita Wiseplayerin nopeus: 0,5-kertainen-, normaali- tai tuplanopeus. Aivan oikeanreunimmainen painike puolestaan avaa Wiseplayerin koko näytölle, niin sanottuun *fullscreen-tilaan*. Edellä mainitut toiminnot ovat esillä kuvassa 5.

Wiseplayerin oikeasta reunasta löytyy pelikellonaika sekä käynnissä oleva erä ja maalitilanne. Näiden alla on koti- ja vierasjoukkueen pelaajalistaukset, joissa sillä hetkellä kentällä olevat pelaajat ovat korostettuina. Kunkin pelaajan kohdalla näkyy listassa myös pelaajan kerryttämä jääaika ja luisteltu matka. Jos pelaajalistauksesta valitsee pelaajan, kyseisen pelaajan 3D-mallin ympärille ilmestyy valokeila (keskiympyrän vasemmalla puolella kuvassa 5), joka seuraa pelaajaa kentällä. Tämän avulla tietyn pelaajan seuraaminen on helpompaa, sillä se erottuu muiden 3D-mallien seasta selkeämmin. Valitun pelaajan kuva, tiedot ja tilastot tulevat myös nähtäville kaukalon yläpuolelle.



Kuva 5. Esimerkkiottelun Wiseplayer-tallenne.

6 3D-kirjastojen vertailu

Kohdassa 4.4 esitettyjen 3D-kirjastojen vertailun pohjaksi valitsin luvussa 3 esitettyjen laatustandardien määrityksistä löytyviä ylätason laatutekijöitä. Nämä valitut laatutekijät ovat käytettävyys, tehokkuus, sekä ylläpidettävyys. Koska vertailuun valikoituneet 3D-kirjastot ovat avoimen lähdekoodin kirjastoja, tarkastelen valittujen laatutekijöiden lisäksi tarkastella tutkimuksessani myös projektien yleistä tilannetta GitHubissa. Tähän yleiseen tilanteeseen lukeutuu projektin viimeisin *stabiili* (stable) versio ja sen julkaisuajankohta, GitHub *edistäjien* (contributor) määrä ja projektin saamat *tähdet* (stars). Nämä kyseiset tiedot ovat julkisia ja ne saadaan selville kunkin GitHub projektin *säiliöstä* (repository).

GitHub Helpin [2019] mukaan tähdet kertovat sen, kuinka moni käyttäjä on antanut projektille tai aiheelle tähden. Sen avulla käyttäjät pystyvät seuraamaan projektin kulkua ja siihen liittyviä aiheita helposti ja nopeasti uutissyötteestään. Tämän lisäksi tähdillä voidaan osoittaa kunnioitusta projektia ja sen tekijöitä kohtaan ja GitHubin tilastoissa projektien sijoitus on usein yhteydessä sen saamien tähtien määrään. Tällä tarkoitetaan sitä, että projekteja etsiessä enemmän tähtiä saaneet projektit sijoittuvat korkeammalle hakutuloksissa.

GitHubin edistäjällä puolestaan tarkoitetaan henkilöä, joka ei suoranaisesti ole töissä projektissa tai sitä työstävässä ryhmässä, mutta haluaa tukea projektin kehitystä raportoimalla löydetyistä ongelmista ja korjaamalla niitä omissa *haaroissaan* (branch). Edistäjät eivät voi siis tuoda suoraan omaa koodiaan projektiin, vaan muutokset täytyy ensin hyväksyttää projektin varsinaisilla kehittäjillä tai työntekijöillä. On kuitenkin syytä huomata, että vertailua varten kerätyt tilastot ovat tarkistettu 13.2.2019 ja tilanne elää vertailtavien kirjastojen kohdalla jatkuvasti.

Aloitin vertailun asentamalla ensiksi globaalin *komentoliittymän* (command-line interface, CLI) Angularille komennolla *npm install -g @angular/cli*. Tämän jälkeen loin Angular CLI:n avulla kaksi identtistä, mutta täysin erillistä Angular-projektia komennoilla *ng new three-js* ja *ng new babylon-js.* Angular CLI:n tarjoama *ng new* komento luo heti ajovalmiin Angular-projektin, josta löytyy juurikansio, projektin alustava arkkitehtuuri, testiprojekti ja konfiguraatiotiedostot. Kumpaankin projektiin asensin projektin nimen mukaisen 3D-kirjastopaketin siten, että three-js -projektin juuressa ajoin komennon *npm install three* ja babylon-js -projektin juuressa ajoin komennon *npm install babylonjs.*

Tämän jälkeen loin suunnitelman, kuinka Wiseplayeria olisi viisainta imitoida siten, että 3D-kirjastojen vertailu olisi mahdollisimman monipuolista luomatta kuitenkaan täysin identtistä järjestelmää Babylon.js:lle. Täysin identtisen järjestelmän luominen olisi tässä vaiheessa turhaa, sillä vertailun painopiste on 3D-kirjastoissa ja niiden ominaisuuksissa, eikä esimerkiksi Wiseplayerin tarjoamia ottelu- ja tilastotietoja tarvitse ottaa mukaan vertailuun. Wiseplayerin imitoimisessa päädyin ratkaisuun, jossa Angular-ohjelma

renderoi valittuja 3D-kirjastoja käyttäen selaimeen jääkiekkokaukalon, pelaajien ja kiekon 3D-mallit. Ohjelma pyörittää 3D-mallinnettua kaukaloa oman x- ja y-akselinsa ympäri, sekä liikuttaa pelaajia ja kiekkoa xy-koordinaatistossa 0.1 koordinaatiopisteen verran sallitun alueen sisällä. Tämä kuvastaa hyvin pitkälti Wiseplayerin pääperiaatetta, jossa pelaajat ja kiekko liikkuvat 3D-maailmassa jääkiekko-ottelusta tallennetun sijaintidatan perusteella ja käyttäjä voi katsoa Wiseplayeria haluamastaan kuvakulmasta annettujen rajojen sisällä. Vaikka Wiseplayer on toteutettu Angular 6:lla, nämä tutkimusta varten toteutetut Wiseplayer-imitaatiot tehtiin käyttäen Angularin versiota 7, sillä Angular 6:n on määrä siirtyä pitkäaikaistukeen 3.11.2019 [Angular 2019b], mikä tarkoittaa sitä, että Wiseplayerin päivittäminen uudempaan Angulariin tulee ajankohtaiseksi ennen edellä mainittua ajankohtaa.

6.1 Three.js toteutus

Ensimmäiseksi luodaan vertailussa käytettävä Wiseplayer-imitaatio Three.js:llä. Angularin luomaan app-komponentin templateen (HTML-tiedosto) luodaan aluksi div-elementti, johon renderoijan luoma canvas-elementti sisällytetään (koodikatkelma 4).

```
1    <div #canvasContainer class="canvasContainer"></div>
```

Koodikatkelma 4. Templaten div-elementti canvas-elementtiä varten.

Tämän jälkeen tuodaan app-komponentin tarvitsemat moduulit sen käyttöön lisäämällä tiedoston alkuun *importit* (koodikatkelma 5). Tämän Wiseplayer-imitaation tarvitsemien moduulien määrä on melko vähäinen, sillä se käyttää vain Angularin itsensä tarjoamia moduuleja, sekä Three.js 3D-kirjaston tarjoamia moduuleja. Näistä huomionarvoisin on Three.js:n GLTFLoader, jonka avulla glb-formaatissa olevat 3D-mallit voidaan ladata Three.js:n käytettäväksi.

```
1    import { Component, ViewChild, ElementRef } from '@angular/core';
2    import * as THREE from 'three';
3    import { GLTFLoader } from 'three/examples/jsm/loaders/GLTFLoader';
```

Koodikatkelma 5. Three.js projektin moduuli importit.

Seuraavaksi määritetään projektin käyttämät luokkamuuttujat (koodikatkelma 6). App-komponentin ensimmäinen luokkamuuttuja on canvasContainer, joka viittaa koodikatkelmassa 4 esitettyyn div-elementtiin. Renderer, scene ja camera ovat WebGL:n vaatimia peruskomponentteja, jotka esiteltiin kohdassa 4.1. Loader on jo aiemmin mainitsemamme GLTFLoader, jonka avulla tietyssä 3D-formaatissa olevat mallit voidaan ladata Three.js:n käytettäväksi. Rink, puck, awayPlayer ja homePlayer ovat muuttujia, joihin

asetetaan Three.js:n käyttämät 3D-mallit. AwayPlayers ja homePlayers ovat puolestaan listoja, jotka pitävät sisällään awayPlayerista ja homePlayerista luodut koti- ja vierasjoukkueen pelaajat. Viimeiset luokkamuuttujat ovat xLimit ja yLimit, jotka kuvastavat koordinaatiston raja-arvoja x- ja y-koordinaateille, eli liikuteltavat kohteet eivät voi mennä näiden koordinaattipisteiden yli.

```
11  export class AppComponent {
12    @ViewChild('canvasContainer') canvasContainer: ElementRef;
13
14    private renderer: THREE.WebGLRenderer = new THREE.WebGLRenderer({antialias: true});
15    private scene: THREE.Scene = new THREE.Scene();
16    private camera: THREE.Camera = new THREE.PerspectiveCamera( 75, window.innerWidth / window.innerHeight, 0.1, 1000);
17    private light = new THREE.HemisphereLight(0xffffbb, 0x080820, 5);
18    private loader: GLTFLoader = new GLTFLoader();
19    private rink: THREE.Mesh;
20    private awayPlayer: THREE.Mesh;
21    private homePlayer: THREE.Mesh;
22    private puck: THREE.Mesh;
23
24    private awayPlayers: THREE.Mesh[] = [];
25    private homePlayers: THREE.Mesh[] = [];
26
27    private yLimit: number = 50;
28    private xLimit: number = 50;
```

Koodikatkelma 6. Luokkamuuttujien määrittäminen.

Luokkamuuttujien alustamisen jälkeen määritetään niiden ominaisuuksille tarvittavat arvot, ladataan käytettävät 3D-mallit, lisätään valaistus skenelle, ja liitetään canvas koodikatkelmassa 4 esitettyyn div-elementtiin ja aloitetaan animaatio ngOnInit-funktiossa (koodikatkelma 7). Tämä funktio on Angular-komponentin *elinkaarifunktio* (lifecycle hook), joka ajetaan komponentin elinaikana vain kerran, aina silloin kuin komponentti luodaan [Angular Guide 2019a]. Tästä johtuen se on luonnollinen paikka luoda ja määrittää asioita, jotka tarvitsevat tehdä vain kerran.

```
30  ngOnInit() {
31    this.renderer.physicallyCorrectLights = true;
32    this.renderer.gammaOutput = true;
33    this.renderer.gammaFactor = 2.2;
34    this.renderer.setClearColor(0xcccccc);
35    this.renderer.setPixelRatio(window.devicePixelRatio);
36    this.renderer.setSize(window.innerWidth, window.innerWidth);
37    this.camera.position.z = 80;
38
39    this.scene.add(this.light);
40    this.loadRink();
41    this.loadPuck();
42    this.loadPlayer(true, this.homePlayer, this.homePlayers);
43    this.loadPlayer(false, this.awayPlayer, this.awayPlayers);
44
45    this.renderer.setSize(window.innerWidth, window.innerHeight);
46    this.canvasContainer.nativeElement.appendChild(this.renderer.domElement);
47
48    this.animate();
49  }
```

Koodikatkelma 7. NgOnInit-funktio ja sen sisältämät määritykset.

Käytettävien 3D-mallien lataaminen tapahtuu koodikatkelmassa 8 näkyvien loadRink-, loadPuck- ja loadPlayer-funktioiden avulla. Funktioissa käytetään Three.js:n tarjoamaa GLTFLoaderia jolle syötetään polku, josta haluttu 3D-malli ladataan. Tässä tapauksessa käytettävät 3D-mallit ovat tallennettuina projektin *assetteihin*, jonne voi säilöä esimerkiksi projektin tarvitsemia kuvia ja käännöstiedostoja, jotta niitä ei tarvitse hakea verkon yli ulkoisesta palvelusta. LoadPuck-funktiossa ladattu kiekon 3D-malli tallennetaan sille varattuun THREE.Mesh-tyyppiseen luokkamuuttujaan ja sille arvotaan satunnainen sijainti sallittujen rajojen sisältä JavaScriptin tarjoaman Math.random-funtkion avulla. Tämän jälkeen kiekon mallin userData-attribuuttiin arvotaan samaa Math.random-funtkiota käyttäen suunta, johon malli lähtee liikkumaan. Lopuksi malli lisätään skeneen, jolloin se tulee käyttäjän nähtäville selaimessa.

```
51    loadPuck(): void {
52      this.loader.load('/assets/puck.glb',
53      function (gltf: any) {
54          this.puck = gltf.scene.children[0];
55          this.puck.position.x = Math.random() * this.xLimit * 2 - this.xLimit;
56          this.puck.position.y = Math.random() * this.yLimit * 2 - this.yLimit;
57          this.puck.userData = {x: Math.random() >= 0.5 ? 0.1 : -0.1, y: Math.random() >= 0.5 ? 0.1 : -0.1};
58          this.scene.add(this.puck);
59        }.bind(this),
60        function (xhr) {
61        },
62        function (error) {
63        });
64    }
65
66    loadRink(): void {
67      this.loader.load('/assets/rink.glb',
68      function (gltf: any) {
69          this.rink = gltf.scene.children[0];
70          this.scene.add(this.rink);
71        }.bind(this),
72        function (xhr) {
73        },
74        function (error) {
75        });
76    }
77
78    loadPlayer(home: boolean, mesh: THREE.Mesh, array: THREE.Mesh[]): void {
79      this.loader.load('/assets/' + (home ? 'home.glb' : 'away.glb'),
80      function (gltf: any) {
81          mesh = gltf.scene.children[0];
82          this.populatePlayers(mesh, array);
83        }.bind(this),
84        function (xhr) {
85        },
86        function (error) {
87        });
88    }
```

Koodikatkelma 8. 3D-mallien latausfunktiot.

Kaukalon 3D-malli ladataan loadRink-funktiossa, jossa kaukalon malli tallennetaan kiekon tavoin sille varattuun THREE.Mesh-tyyppiseen luokkamuuttujaan. Koska kaukalon ei tarvitse liikkua skenessä, sille ei tarvitse määrittää sijaintia eikä suuntaa. Jos näitä arvoja ei määritetä, ne asetetaan oletusarvoisesti keskelle skeneä x-, y-, ja z-koordinaatiopisteeseen (0, 0, 0). Lopuksi kaukalon malli lisätään myös käyttäjän nähtäville skeneen.

Pelaajien lataaminen loadPlayer-funktiossa poikkeaa hieman aiemmista latausfunktioista, sillä funktiolle on määritetty parametrit kertomaan sen, ladataanko vieras- vai kotijoukkueen pelaajien mallit. Ensimmäinen parametri *home* on boolean-tyyppinen, joka kertoo sen, ladataanko koti- vai vierasjoukkueen malli. Toisena parametrina *mesh* puolestaan kertoo sen THREE.Mesh-tyyppisen luokkamuuttujan, johon ladattu malli halutaan tallentaa. Viimeisen parametrin *array* avulla taas määritetään se, mihin THREE.Mesh-tyyppejä sisältävään luokkamuuttujalistaan ladatuista malleista luodaan kopioita. Funktion sisällä ladattu malli tallennetaan jälleen kerran parametrina annettuun THREE.Mesh-tyyppiseen luokkamuuttujaan, jonka jälkeen siirrytään populatePlayers-funktioon (koodikatkelma 9), joka ottaa parametrinaan loadPlayers-funktiolle annetut *mesh* ja *array* parametrit.

```
90  populatePlayers(playerMesh: THREE.Mesh, playerArray: THREE.Mesh[]): void {
91    for (let i = 0; i < 6; i++) {
92      const meshCopy = playerMesh.clone();
93      meshCopy.position.x = Math.random() * this.xLimit * 2 - this.xLimit;
94      meshCopy.position.y = Math.random() * this.yLimit * 2 - this.yLimit;
95      meshCopy.userData = {x: Math.random() >= 0.5 ? 0.1 : -0.1, y: Math.random() >= 0.5 ? 0.1 : -0.1};
96      this.scene.add(meshCopy);
97      playerArray.push(meshCopy);
98    }
99  }
```

Koodikatkelma 9. PopulatePlayers-funktio.

PopulatePlayers-funktio menee for-looppiin, jota ajetaan kuuden iteraation verran. Jokaisessa iteraatiossa se luo parametrina annetusta THREE.Mesh-tyyppisestä *playerMesh* oliosta kopion, arpoo sille sijainnin ja suunnan samalla tavalla kuin loadPuck-funtkiossa kiekon 3D-mallille, lisää sen skeneen ja puskee parametrina annettuun *playerArray* THREE.Mesh-tyyppejä sisältävään listaan.

3D-mallien lataamisten jälkeen ngOnInit-funktiossa määritetään renderoijan koko ja liitetään se canvas-elementtiin. Tämän jälkeen kutsutaan enää 3D-mallien liikuttelusta vastaavaa animate-funktiota, joka on esitetty koodikatkelmassa 10. Animate-funktio liittää selainikkunaan *tapahtumakuuntelijan* (event listener), joka kuuntelee HTML-dokumenttia. Kun koko dokumentti on ladattu ja jäsennelty, eikä yhtäkään tyyliä, kuvaa tai muuta dokumentin osaa enää odoteta, laukaistaan koodikatkelmassa 10 näkyvä render-funktio [MDN web docs 2019a].

```
101    animate(): void {
102      window.addEventListener('DOMContentLoaded', () => {
103        this.render();
104      });
105    }
106
107    moveMesh(mesh: THREE.Mesh): void {
108      mesh.position.x += mesh.userData.x;
109      mesh.position.y += mesh.userData.y;
110
111      if (mesh.position.x < -this.xLimit || mesh.position.x > this.xLimit) {
112        mesh.userData.x = -mesh.userData.x;
113      }
114      if (mesh.position.y < -this.yLimit || mesh.position.y > this.yLimit) {
115        mesh.userData.y = -mesh.userData.y;
116      }
117    }
118
119    render(): void {
120      requestAnimationFrame(() => {
121        this.render();
122      });
123
124      this.moveMesh(this.puck);
125
126      for (const homePlayer of this.homePlayers) {
127        this.moveMesh(homePlayer);
128      }
129
130      for (const awayPlayer of this.awayPlayers) {
131        this.moveMesh(awayPlayer);
132      }
133
134      if (this.rink) {
135        this.rink.rotation.x += 0.005;
136        this.rink.rotation.y += 0.005;
137      }
138
139      this.renderer.render(this.scene, this.camera);
140    }
```

Koodikatkelma 10. Animoinnista ja renderoinnista vastaavat funktiot.

Render-funktio alkaa kutsumalla requestAnimationFrame-funktiota, joka kertoo selaimelle, että haluamme suorittaa animaation ja vaadimme, että selain kutsuu requestAnimationFrame-funktion sisällä olevaa funktiota ennen seuraavaa selainikkunan ”uudelleenpiirtoa”. Uudelleenpiirtämisellä tarkoitetaan tässä yhteydessä sitä, kun animaation aikana 3D-mallit liikkuvat ja mallien uudet sijainnit tulee piirtää selainikkunaan. [MDN web docs 2019b]

Tämän jälkeen render-funktio kutsuu moveMesh-funktiota, jonka vastuulla on sille parametrina annetun THREE.Mesh-tyyppisen olion, eli tässä tapauksessa skenessä näkyvän 3D-mallin, liikuttaminen (koodikatkelma 10). Ensiksi parametrina annettua mallia liikutetaan mallin userData-attribuutista löytyvän arvon verran x- ja y-koordinaatistoissa. Tämän jälkeen katsotaan, onko malli vielä sallittujen rajojen sisäpuolella. Mikäli näin ei

ole, mallin userData-attribuuteista löytyvät suunnat muutetaan päinvastaisiksi, jolloin mallin seuraavat liikkeet kohdistuvat päinvastaiseen suuntaan.

Render-funktio liikuttaa ensimmäiseksi kiekon 3D-mallia, jonka jälkeen se käy läpi koti- ja vieraspelaajien 3D-malleista koostuvat pelaajalistat ja liikuttaa jokaista näistä listoista löytyvää pelaajaa moveMesh-funktion avulla. Pelaajien 3D-mallien liikuttamisen jälkeen kiekkoa pyöritetään 0.05 koordinaatiopistettä eteenpäin sekä x- ja y-koordinaatistossa. Lopuksi render-funktio käskyttää Three.js:n tarjoamaa WebGL renderoijaa suorittamaan renderoinnin.

Näiden edellä mainittujen ja esitettyjen funktioiden avulla saadaan luotua Three.js toteutus, jossa selainikkunassa näkyy ja liikkuu kaukalon, kiekon ja kahdentoista kenttäpelaajan 3D-mallit. Tätä toteutusta käytetään tutkielman vertailussa seuraavaksi toteutettavan Babylon.js toteutuksen rinnalla.

6.2 Babylon.js toteutus

Babylon.js toteutuksen luominen poikkeaa hieman Three.js toteutuksesta, sillä siinä missä projektin app-komponentin templateen Three.js:llä luodaan div-elementti (koodikatkelma 4), niin Babylon.js:llä templateen luodaan suoraan canvas-elementti (koodikatkelma 11).

```
1    <canvas id="canvas" #canvas></canvas>
```

Koodikatkelma 11. Babylon.js toteutuksen canvas-elementti.

App-komponentin alustus on hyvin samankaltainen kuin Three.js toteutuksessakin, sillä sen alussa määritetään komponentin tarvitsemat moduulit (koodikatkelma 12). Molempien toteutuksien käyttämät moduulit ovat hyvin samankaltaiset, sillä ne käyttävät samoja Angularin tarjoamia moduuleja. Ainoat poikkeavuudet tulevat 3D-kirjastoissa.

```
1    import { Component, ViewChild, ElementRef, OnInit } from '@angular/core';
2    import * as BABYLON from 'babylonjs';
3    import 'babylonjs-loaders';
```

Koodikatkelma 12. Babylon.js toteutuksen tarvitsemat moduulit.

Seuraavaksi Babylon.js toteutukseen määritetään luokkamuuttujat, joita on selvästi vähemmän kuin Three.js toteutuksessa (koodikatkelma 13). Huomionarvoista luokkamuuttujissa on se, canvasElement-muuttuja viittaa app-komponentin templatessa sijaitsevaan canvas-elementtiin, kun taas canvas-muuttuja on muuttuja, joka annetaan myöhemmin (koodikatkelmassa 14) BABYLON.Engine tyyppiä olevalle engine-muuttujalle. Näiden

lisäksi määritetään luokkamuuttujissa myös Three.js toteutuksessa käytetyt skene muuttujassa scene, sekä raja-arvot x- ja y-koordinaateille muuttujissa yLimit ja xLimit.

```
10  export class AppComponent implements OnInit {
11    @ViewChild('canvas') canvasElement: ElementRef;
12    private canvas: HTMLCanvasElement;
13    private engine: BABYLON.Engine;
14    private scene: BABYLON.Scene;
15
16    private yLimit: number = 50;
17    private xLimit: number = 50;
```

Koodikatkelma 13. Babylon.js toteutuksen luokkamuuttujat.

Babylon.js toteutuksen ngOnInit-funktiossa (koodikatkelma 14) canvas-muuttujaksi asetetaan app-komponentin templatesta löytyvä ja koodikatkelmassa 11 esitelty canvas-elementti. Tässä funktiossa luodaan myös WebGL:n tarvitsemat renderoija, skene, valot ja kamera. Näiden jälkeen ladataan kiekon ja kaukalon 3D-mallit, sekä kuusi kappaletta koti- ja vierasjoukkueen pelaajien 3D-malleja kutsumalle niiden lataamiseen tarkoitettuja funktioita. Lopuksi käskytetään render-funktiota, joka aloittaa renderoinnin.

```
19  ngOnInit(): void {
20    this.canvas = this.canvasElement.nativeElement;
21    this.engine = new BABYLON.Engine(this.canvas, true);
22    this.scene = new BABYLON.Scene(this.engine);
23    const light = new BABYLON.HemisphericLight('light1', new BABYLON.Vector3(0, 1, 0), this.scene);
24    const camera = new BABYLON.ArcRotateCamera("camera1", Math.PI / 2, Math.PI / 2, 150, new BABYLON.Vector3(0, 1, 0), this.scene);
25
26    this.loadPlayerOrPuck('puck', this.scene);
27    this.loadRink(this.scene);
28
29    for (let i = 0; i < 6; i++) {
30      this.loadPlayerOrPuck('home', this.scene);
31      this.loadPlayerOrPuck('away', this.scene);
32    }
33
34    this.render();
35  }
```

Koodikatkelma 14. Babylon.js toteutuksen ngOnInit-funktio.

Kiekon ja pelaajien 3D-mallit ladataan samaa funktiota hyödyntäen, sillä niiden toimintaperiaatteet ovat täysin identtiset: ladataan malli, arvotaan mallille sijainti ja suunta ja liikutetaan mallia ennen jokaista renderointia (koodikatkelma 15).

Mallien lataaminen tapahtuu BABYLON.SceneLoaderin avulla kutsumalla Import-Mesh-funktiota. Import-Mesh-funktio ottaa useita parametreja, joista ensimmäisessä voidaan syöttää ladattavien mallien nimet. Jos parametri jätetään tyhjäksi, funktio lataa kaikki mallit tiedostosta. Toinen ja kolmas parametri määrittää kohteen, josta ladattavan mallin tiedosto löytyy siten, että toiseen parametriin syötetään polku ja kolmanteen parametriin tiedoston nimi. Neljäs parametri on skene, johon ladattu malli halutaan tuoda ja viidentenä puolestaan niin kutsuttu *callback*-funktio, jota kutsutaan 3D-mallin latauksen onnistuttua.

```
37    loadPlayerOrPuck(teamOrPuck: string, scene: BABYLON.Scene): void {
38      const xLimit = this.xLimit;
39      const yLimit = this.yLimit;
40
41      BABYLON.SceneLoader.ImportMesh("", "assets/", teamOrPuck + ".glb", scene, function (newMeshes) {
42        const mesh = newMeshes[1];
43        mesh.position.x = Math.random() * xLimit * 2 - xLimit;
44        mesh.position.y = Math.random() * yLimit * 2 - yLimit;
45        mesh.metadata = {x: Math.random() >= 0.5 ? 0.1 : -0.1, y: Math.random() >= 0.5 ? 0.1 : -0.1};
46        mesh.rotationQuaternion = null;
47        mesh.doNotSyncBoundingInfo = true;
48
49        scene.registerBeforeRender(function () {
50          mesh.position.x += mesh.metadata.x;
51          mesh.position.y += mesh.metadata.y;
52
53          if (mesh.position.x < -xLimit || mesh.position.x > xLimit) {
54            mesh.metadata.x = -mesh.metadata.x;
55          }
56
57          if (mesh.position.y < -yLimit || mesh.position.y > yLimit) {
58            mesh.metadata.y = -mesh.metadata.y;
59          }
60        });
61      });
62    }
```

Koodikatkelma 15. Pelaajien ja kiekon 3D-mallin lataus Babylon.js toteutuksessa.

Mallin sijainti ja suunta arvotaan täysin samalla tapaa kuin Three.js toteutuksessakin, eli JavaScriptin Math.random-funktion avulla. Three.js toteutuksesta poiketen mallille arvottu suunta annetaan ladatun mallin metadata-attribuutille, sillä BABYLON.Mesh-tyyppi ei sisällä userData-attribuuttia.

Babylon.js dokumentaation [2019b] mukaan rotation-attribuutin käyttäminen mallin kiertämisessä ulkopuolelta ladatuissa malleissa vaatii rotationQuaternion-attribuutin asettamisen tyhjäksi, eli *nulliksi*. Lisäksi skenen optimoimiseksi Babylon.js dokumentaatio [2019c] suosittelee 3D-mallien doNotSyncBoundingInfo-attribuutin asettamista trueksi, sillä se estää tietojen päivittämisen ja näin ollen nopeuttaa laskentaa. Tämä oli välttämätöntä myös siitä syystä, että mallit kävivät selaimessa näkyvillä satunnaisesti vilahtaen ilman kyseisen attribuutin manipulointia.

3D-mallien liikuttaminen tapahtuu kutsumalla scene-muuttujan registerBeforeRender-funktio, joka ottaa parametrinaan funktion. Tämän parametrina annettavan funktion voi määrittää tekemään erilaisia toimenpiteitä, jotka suoritetaan aina ennen renderointia, eli ennen jokaista selaimen näyttämää uutta kuvaruutua. Tässä tapauksessa käytämme samanlaista logiikkaa 3D-mallien liikuttamiseksi kuin Three.js toteutuksessakin, eli ensiksi siirrämme mallia x- ja y-koordinaatistossa mallin metadatasta löytyvään suuntaan. Tämän jälkeen katsomme, onko mallin sijainti sallittujen rajojen sisäpuolella. Mikäli näin ei ole, niin muutetaan mallin suuntaa päinvastaiseksi, jolloin seuraavalla kerralla malli liikkuu poispäin rajasta.

Kaukalon 3D-mallin lataaminen on hyvin samankaltainen toimenpide kuin pelaajien ja kiekon 3D-mallienkin, ja se on esitelty koodikatkelmassa 16. Mallin lataamiseksi hyödynnetään samaa BABYLON.SceneLoader-luokkaa ja sen tarjoamaa ImportMesh-funktiota muuttamalla ainoastaan tiedostonimi vastaamaan kaukalon 3D-mallin glb-tiedoston nimeä. Kaukalon 3D-mallin onnistuneen latauksen jälkeen suoritettava callback-funktio poikkeaa pelaajien ja kiekon mallien latauksen callback-funktiosta siten, ettei kaukaloa haluta liikuttaa, vaan ainoastaan kiertää. Tämä tapahtuu vastaavalla tavalla kuin aikaisempienkin mallien kanssa kutsumalla scene-muuttujan registerBeforeRender-funktiota ja antamalla tälle parametriksi funktion, joka kiertää kaukaloa 0.01 pistettä x- ja y-koordinaatistossa.

```
64    loadRink(scene: BABYLON.Scene): void {
65      BABYLON.SceneLoader.ImportMesh("", "assets/", "rink.glb", scene, function (newMeshes) {
66        const mesh = newMeshes[1];
67        mesh.rotationQuaternion = null;
68        mesh.doNotSyncBoundingInfo = true;
69
70        scene.registerBeforeRender(function () {
71          mesh.rotation.x += 0.01;
72          mesh.rotation.y += 0.01;
73        });
74      });
75    }
```

Koodikatkelma 16. Babylon.js toteutuksen kaukalon 3D-mallin latausfunktio.

Lopuksi ngOnInit-funktion kutsuman render-funktion sisällä kutsutaan engine-muuttujan runRenderLoop-funktiota, joka laukaisee renderoinnin. RunRenderLoop-funktiolle annetaan parametriksi toinen funktio, joka halutaan suoritettavan jatkuvasti renderoinnin yhteydessä. Tässä tapauksessa funktio tarkistaa onko scene-muuttuja valmis renderoitavaksi, eli onko se ladannut esimerkiksi kaikki tekstuurit ja muut tarvittavat elementit. Mikäli skene on valmis renderoitavaksi, niin kutsutaan scene-muuttujan render-funktiota, joka renderoi skenen (koodikatkelma 17).

```
77    render(): void {
78      this.engine.runRenderLoop(() => {
79        if (this.scene.isReady()) {
80          this.scene.render();
81        }
82      });
83    }
```

Koodikatkelma 17. Babylon.js toteutuksen renderoinnista vastaava funktio.

6.3 Yleinen vertailu

Taulukossa 1 on vertailtu Three.js:n ja Babylon.js:n viimeisimpien stabiilien versioiden yleisiä ominaisuuksia. Three.js:n viimeisin stabiili versio 0.103.3 on julkaistu 27.3.2019, sillä on edistäjiä 1067 ja tähtiä se on saavuttanut 50559 [GitHub Three.js 2019].

Babylon.js:n viimeisin stabiili versio 3.3.0 on puolestaan hieman vanhempi, sillä se on julkaistu 1.10.2018. Edistäjiä kirjastolla on 220 ja tähtiä se on saanut kerättyä 8683. [GitHub Babylon.js 2019]

Näiden yleisten ominaisuuksien perusteella voidaan siis melko suoraviivaisesti todeta, että Three.js on suositumpi ja sillä on suurempi edistäjäkunta raportoimassa havaituista virheistä.

	Three.js	**Babylon.js**
Versio	0.103.0 Julkaistu 27.3.2019	3.3.0 Julkaistu 1.10.2018
Edistäjät	1067	220
Tähdet	50559	8683

Taulukko 1. Three.js ja Babylon.js yleisten ominaisuuksien vertailua.

6.4 Ylläpidettävyys

Luvussa 3 läpikäydyt McCallin laatutekijät ja ISO 9126 -standardi pitivät sisällään laatutekijän, jota kutsutaan ylläpidettävyydeksi. Tällä tarkoitettiin toimenpiteitä ja niistä syntyvää vaivaa, joka tarvitaan esimerkiksi ohjelmiston muuttamiseksi, parantamiseksi tai korjaamiseksi. Vertailua varten valittiin neljä yksinkertaista muutosta liittyen malleihin ja niiden ominaisuuksiin, jotta vertailtavilla kirjastoilla tehtyjen toteutusten ylläpidettävyyttä voitaisiin tutkia. Nämä tutkittavat muutokset ovat mallin nimi, koko, näkyvyys ja väri.

Kummassakin toteutuksessa mallit ovat selkeästi haettavissa skenestä ja jokainen malli on käsiteltävissä erikseen, mikä on hyvä ja selkeä merkki ylläpidettävyydestä. Three.js tarjoaa mallin nimen muuttamiseksi Object3D-luokalle name-attribuutin, jota muuttamalla mallin nimeä voidaan vaihtaa. Tämä Object3D-luokka on yläluokka, josta periytyy muun muassa Three.js toteutuksen yhteydessä esitetyissä koodikatkelmissakin näkyvä THREE.Mesh-luokka, joka ilmentää skenessä näkyvät 3D-mallit. Vastaavasti Babylon.js tarjoaa omassa BABYLON.Mesh-luokassaan name-attribuutin, jonka avulla mallin nimeä voidaan muuttaa. 3D-mallin nimen muuttaminen on oleellinen osa ohjelmiston toimintaa, sillä sen ja mallin id:n avulla skenestä on mahdollista etsiä juuri tietty malli. Erona nimessä ja id:ssä on kuitenkin se, että id on hyvin usein uniikki ja yksilöivä

tunniste, jota ei voi muuttaa. Tämä aiheuttaa ongelmaksi sen, että tietyn mallin löytäminen skenestä id:n avulla vaatii id:n tietämistä ennestään, kun taas nimeksi voidaan antaa käyttäjän itsensä haluama merkkijono.

Mallin koon muuttaminen on Three.js kirjastolla vastaavanlainen toimenpide kuin mallin nimenkin muuttaminen. Object3D-luokka tarjoaa scale-attribuutin, joka on tyyppiä Vector3. Vector3-luokka kuvastaa 3D-vektoria, joka on x-, y- ja z-koordinaatteja esittävä numerokolmikko. Vector3-luokka tarjoaa set-funktion, jolle voidaan antaa parametrina x-, y-, ja z-koordinaatit, jotka asetetaan kyseiseen vektoriin. Mallin koon muuttaminen siis Three.js toteutuksessa tapahtuu esimerkiksi *malli.scale.set(2, 2, 2)* kutsulla, jolloin mallin kokoa skaalataan kaksinkertaiseksi jokaiseen suuntaan.

Babylon.js tarjoaa lähes identtisen lähestymistavan mallin koon muuttamiseksi, sillä BABYLON.Mesh-luokka omistaa scaling-attribuutin, joka on vastaavalla tavalla saman toimintaperiaatteen omaava Vector3-luokka. Babylon.js:n Vector3-luokka tarjoaa myös set-funktion, joka ottaa parametrinaan kolme arvoa, jotka kuvastavat x-, y-, ja z-koordinaatteja. Tämän seurauksena Babylon.js kirjastossa mallin kokoa voidaan muuttaa kutsumalla *malli.scaling.set(2, 2, 2)*, joka tekee saman operaation kuin Three.js toteutuskin.

3D-mallien näkyvyyden ylläpito on aiempien muutosten kanssa hyvin samankaltainen. Babylon.js tarjoaa näkyvyyden muuttamiseksi BABYLON.Mesh-luokalle kaksi mahdollista lähestymistapaa: isVisible- ja visibility-attribuutin. Näiden kahden välinen eroavaisuus on niiden tyypit, sillä isVisible-attribuutti on boolean-arvo, kun taas visibility-attribuutti on tyypiltään number. IsVisible-attribuutilla voidaan määrittää, onko malli näkyvissä vai ei yksinkertaisesti antamalla attribuutille arvoksi true tai false. Visibility-attribuutti puolestaan ottaa arvoja aina numeroiden 0 ja 1 välillä, jolloin kohteen näkyvyyttä voidaan säätää hieman monipuolisemmin, esimerkiksi arvolla 0.5 kohteen läpinäkyvyys on puolittunut. Three.js ei tarjoa aivan näin monipuolista näkyvyyden säätöä, sillä sen Object3D-luokka tarjoaa vain visible-attribuutin, joka on toiminnaltaan täysin BABYLON.Mesh-luokan isVisible-attribuutin kaltainen.

Viimeinen vertailuun valittu muutos on 3D-mallin värin muuttaminen. Three.js tarjoaa tähänkin toimenpiteeseen melko suoraviivaisen ratkaisun, sillä THREE.Mesh-luokka tarjoaa material-attribuutin, joka on tyypiltään THREE.Material. Tämä luokka toimii yläluokkana useammalle alaluokalle, joista yksi on myös THREE.Mesh-luokan materiaalinmäärityksessä oletusarvoisena toimiva THREE.MeshBasicMaterial. Kyseinen luokka ottaa *rakentajassaan* (constructor) parametriksi objektin, jolle voidaan määritellä yksi tai useampi ominaisuus. Yksi näistä ominaisuuksista on color, joka määritetään heksadesimaalisena merkkijonona, kuten esimerkiksi valkoista väriä kuvastava 0xFFFFFF. Kun uusi THREE.MeshBasicMaterial on luotu, se voidaan antaa THREE.Mesh-tyyppiselle muuttujalle material-attribuutiksi, jolloin mallin ulkoasu muuttuu määritetyn mukaiseksi.

Mallin värin muutos Babylon.js:sä on jälleen kerran hyvin vastaavanlainen kuin Three.js:säkin. BABYLON.Mesh-luokka pitää material-attribuutissa tiedot mallin materiaalista, joka on BABYLON.Material-tyyppinen. BABYLON.Material on yläluokka Babylon.js:n erilaisille materiaalityypeille, kuten esimerkiksi oletusarvoiselle BABYLON.StandardMaterial-luokalle. Kun uusi BABYLON.StandardMaterial-muuttuja luodaan, sille voidaan määritellä materiaalin perusväritys muuttamalla BABYLON.Color3-tyyppistä diffuseColor-attribuuttia. BABYLON.Color3 on luokka, jonka avulla voidaan määritellä väri antamalla rakentajan parametriksi väriarvot number-tyyppeinä arvojen ollessa väliltä 0-1. Ensimmäinen parametri määrittää punaisen värin, toinen parametri vihreän ja kolmas parametri sinisen. Kun diffuseColor-attribuutti on määritetty ja BABYLON.Mesh-muuttujan material-attribuutti on asetettu BABYLON.StandardMaterialiksi, niin 3D-mallin ulkoasu muuttuu määritetyn materiaalin mukaiseksi.

Edellä mainittujen muutosten osalta Babylon.js ja Three.js ovat ylläpidettävyydeltään hyvin samankaltaisia. Molemmat pitävät sisällään hyvin samankaltaisia luokkia ja attribuutteja, joita manipuloimalla ohjelman muuttaminen, parantaminen ja korjaaminen onnistuu kohtalaisen vaivattomasti. On kuitenkin syytä muistaa, että tehdyt toimenpiteet ylläpidettävyyden testaamiseksi ovat hyvin yleisiä 3D-maailmaan liittyviä ominaisuuksia. Kokonaisvaltaisempi testaus ylläpidettävyyden toteamiseksi olisi paikallaan, mikäli eroavaisuuksia kirjastojen välillä haluttaisiin tutkia. Näiden toimenpiteiden pohjalta voidaan kuitenkin todeta, että yleisellä tasolla molemmat kirjastot ovat hyvin ylläpidettäviä, eikä niiden väliltä löydy suuria eroja vaivannäössä tiettyjen koodimuutosten aikaansaamiseksi.

6.5 Käytettävyys

Kuten luvussa 3 mainittiin, niin käytettävyydellä tarkoitetaan ohjelmiston käyttämisestä syntyvää vaivaa ja siihen sisältyy ymmärrettävyys, opittavuus ja toimivuus. Babylon.js toteutuksen toimeenpanemisesta syntyi pieniä haasteita, sillä Babylon.js:n viimeisin vakaa versio 3.3.0 ei tue TypeScriptin versiota 3.2.4, joka oli käytössä Three.js toteutusta luodessa. Tämän seurauksena TypeScriptin versiota kokeiltiin pudottaa viimeisimmäksi Babylon.js version 3.3.0 kanssa yhteensopivaksi, mutta tämän seurauksena uusimmat Angularit eivät enää tukeneet niin alhaista TypeScriptin versiota. Jotta toteutuksesta saataisiin mahdollisimman yhteneväinen Three.js toteutuksen kanssa, ainoaksi vaihtoehdoksi jäi ottaa käyttöön uusin Babylon.js:n versio, joka on vielä betavaiheessa oleva 4.0.0. Tämän avulla kuitenkin uusimmat Angularit ja TypeScriptin versiot toimivat keskenään.

Tämän riippuvuusongelman lisäksi Babylon.js vaatii erillisen paketin asentamista (babylonjs-loaders), jotta esimerkiksi Wiseplayerin käyttämiä glb-tiedstoja voidaan ladata ohjelman käytettäväksi. Nämä edellä mainitut tekijät heikentävät kyseisen kirjaston

käytettävyyttä merkittävästi, sillä sen käyttämisestä syntynyt vaiva ja ohjelmiston toimivuus olivat huomattavasti heikommat kuin Three.js 3D-kirjaston, jonka toteutuksen alulle saattaminen sujui täysin ongelmitta eikä ulkopuolisia paketteja tarvittu ominaisuuksien käyttämiseksi.

Ohjelmakoodia tarkastellessa puolestaan voidaan huomata Babylon.js:n käytettävyyttä puoltavia tekijöitä. Se tarvitsee muun muassa huomattavasti vähemmän luokkamuuttujia ja ohjelmakoodin rivimäärä on selkeästi pienempi kuin Three.js toteutuksessa. Uskon näiden asioiden lisäävän ohjelmiston ymmärrettävyyttä ja opittavuutta, sillä lyhyt ja yksinkertainen koodi on usein helpommin ymmärrettävää ja opittavaa kuin pitkä ja monimutkainen. Toisaalta on syytä kuitenkin muistaa, että nämä toteutukset ovat vain pieni osa varsinaista Wiseplayeria, eivätkä ne toteutukseltaankaan kuvasta Wiseplayerin toimintaa täydellisesti. Tästä syystä esimerkiksi noin 50 koodirivin lyhemmyys ei ole kovin merkittävä tekijä todellisessa Wiseplayerissa, jossa koodirivejä on huomattavasti enemmän.

Käytettävyyteen liittyviin tekijöihin, kuten opittavuuteen ja ymmärrettävyyteen vaikuttaa oleellisesti myös käyttöohjeet ja muut dokumentaatiot. Dokumentaatioiden osalta molemmat kirjastot suoriutuvat hyvin, sillä ne sisältävät konkreettisia esimerkkejä erilaisten asioiden toteuttamiseksi sekä yksityiskohtaisempia selostuksia esimerkiksi eri funktioista ja niiden käyttötavoista ja -tarkoituksista. Three.js toteutuksen aikaansaamisessa käyttöohjeille ei ollut käyttöä niin paljoa kuin Babylon.js toteutuksen implementoimisessa, mutta tämän uskon johtuvan hyvin pitkälti siitä syystä, että Three.js oli 3D-kirjastona töiden kautta jo ennestään huomattavasti tutumpi kuin Babylon.js. Tästä huolimatta Babylon.js toteutuksen aikaansaanti oli alun yhteensopivuusongelmien jälkeen mutkatonta.

6.6 Tehokkuus

Luvussa 3 esitettyjen laatutekijöiden mukaan tehokkuudella tarkoitetaan yksinkertaisuudessaan sitä, kuinka paljon sovellus käyttää resursseja. Ensimmäisenä tehokkuuteen liittyvänä piirteenä tarkastellaan paketin kokoa, jolla tarkoitetaan ohjelman lähdekoodin viemää tilaa pienennettynä ja pakattuna, eli kuinka pieneen tilaan ohjelma saadaan pakattua ja kuinka paljon tämä paketti vie levytilaa. Levytilan viemisen lisäksi paketin koko kertoo usein myös sen, kuinka nopeasti paketti esimerkiksi asentuu projektiin ja kuinka paljon aikaa sen kääntämiseen kuluu. Mitä pienempi paketti on, sen nopeammin siihen kohdistetut operaatiot usein suoriutuvat.

Pienennys tapahtuu poistamalla lähdekoodista kaikki kommentit, välilyönnit ja ylimääräiset rivinvaihdot. Tällä saadaan siis karsittua kaikki tilaa vievät ja ohjelmakoodin kääntymisen kannalta turhat tekijät pois. Koodikatkelmassa 17 näkyy yksinkertainen for-

silmukka, joka tulostaa selaimen konsoliin kymmenen kertaa ”Hello world!”. Silmukan yläpuolella on kommentti, joka kertoo mitä kyseinen koodilohko tekee.

```
1  // This is a for-loop that logs "Hello world!" ten times
2  for (var i = 0; i < 10; i++) {
3    console.log("Hello world!");
4  }
```

Koodikatkelma 17. Yksinkertainen for-silmukka kommentteineen.

Koodikatkelmassa 18 puolestaan näkyy tismalleen sama koodi kuin koodikatkelmassa 17, mutta pienennettynä. Tässä huomataan, kuinka koodista on poistettu kommentit, välilyönnit ja rivinvaihdot. Alkuperäisen, koodikatkelmassa 17 näkyvän koodin koko on 123b, kun taas pienennetyn, koodikatkelmassa 18 näkyvän koodin koko on 50b. Tilaa säästyy siis 73b pienennyksen ansiosta. Tämän kokoisessa esimerkissä tilan säästö ei ole huomattava, mutta mitä suuremmaksi projekti kasvaa, sen suuremmat hyödyt koodin pienennyksestä saadaan.

```
1  for(var i=0;i<10;i++){console.log("Hello world!")}
```

Koodikatkelma 18. Pienennetty for-silmukka koodikatkelmasta 17.

Three.js 3D-kirjaston paketin pienennetty koko on 565kB, ja pakattuna se pienenee entisestään kokoon 136.5kB [GitHub Three.js 2019]. Babylon.js on puolestaan pienennettynä kooltaan noin 2mB (eli noin 2000kB) ja pakattuna se pienenee kokoon 455.6kB [GitHub Babylon.js 2019]. Näiden tietojen perusteella voidaan siis todeta, että Three.js on kooltaan yli kolme kertaa pienempi kuin Babylon.js.

Seuraavaksi tarkastellaan toteutettujen Wiseplayer-imitaatioiden tehokkuutta Chrome-selaimen *kehittäjätyökalujen* (developer tools) avulla. Kehittäjätyökalut tarjoavat web-kehittäjille erilaisia työkaluja, joiden avulla sivustoa voidaan seurata kehittäjän kannalta oleellisista näkökulmista. Niiden avulla esimerkiksi sivustojen muokkaaminen ja erilaisten tyylien kokeilu onnistuu ilman ohjelmakoodiin kajoamista ja sivuston tehokkuutta ja erilaisiin toimenpiteisiin kuluvaa aikaa voidaan diagnosoida. Näiden työkalujen avulla sivustojen kehitys on nopeampaa ja helpompaa, sekä saadaan konkreettista hyötyä käyttäjille paneutumalla oikeisiin ongelmakohtiin [Google Developers 2019a].

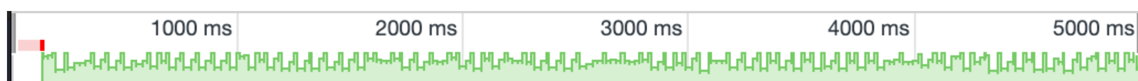
Molempien toteutusten tehokkuustestit ajetaan samalla laitteella identtisissä ympäristöissä, jotta testin tulokset olisivat vertailtavissa keskenään. Laitteena toimii 12-tuumaisella Retina-näytöllä varustettu vuoden 2017 MacBook, jossa on 1,3GHz Intel i5 proses-

sori, 8 Gt 1867 MHz LPDDR3 keskusmuistia, Intel HD Graphics 615 1536 Mt näytönohjain ja 512Gt SSD. Laitteen käyttöjärjestelmän versio on macOS Mojave 10.14.3 (18D109). Testit ajetaan Chrome-selaimella sen kattavien kehittäjätyökalujen vuoksi. Käytettävän Chrome-selaimen versio on testausajanhetkellä tuorein 64-bittinen versio 73.0.3683.103.

Tehokkuustestaus tapahtui mittaamalla Chrome-selaimen kehittäjätyökalun suorituskyky diagnoosityökalulla noin viiden sekunnin mittainen diagnosointijakso, jonka aikana selain mittasi sivustolla tapahtuvia aktiviteetteja ja niiden viemää aikaa. Ensimmäisenä tarkastellaan sitä, kuinka monta kuvaruutua per sekunti (frames per second, FPS) selain kykenee näyttämään kussakin toteutuksessa. Kuvassa 6 näkyy Three.js toteutuksen FPS-graafi. Mikäli graafissa näkyy punaista, se tarkoittaa usein sitä, että FPS on sillä hetkellä ollut niin matala, että sillä on ollut vaikutusta käyttäjäkokemukseen. Vihreä palkki puolestaan kuvastaa sen hetkistä FPS-lukemaa, ja mitä korkeampi FPS on, sitä sulavammalta animaatio näyttää selaimessa [Google Developers 2019b]. Three.js toteutuksessa punaista ajanjaksoa ei ole kuin heti mittauksen aloituksessa, joka voi selittyä esimerkiksi mittauksen aloittamisesta johtuvalla hetkellisellä kuormituksella ja siitä seuraavalla suorituskyvyn laskulla. Tämän jälkeen FPS on ollut vihreää, eli käyttäjäkokemuksen kannalta riittävän hyvää.

Kuva 6. Three.js toteutuksen FPS-graafi.

Kuvassa 7 näkyy puolestaan Babylon.js toteutuksen vastaava FPS-graafi. Tästä graafista voidaan huomata samanlainen punainen alue heti mittauksen alkaessa, joka viittaa todennäköisesti samanlaiseen kuormituksen alkamiseen ja siitä johtuvaan suorituskyvyn laskuun kuin Three.js toteutuksessakin. Babylon.js toteutuksen FPS-graafin vihreää osuutta verrattaessa Three.js toteutuksen FPS-graafin vihreään osuuteen voidaan huomata, että Babylon.js toteutus suoriutuu hieman korkeammilla ja säännöllisemmillä FPS lukemilla.

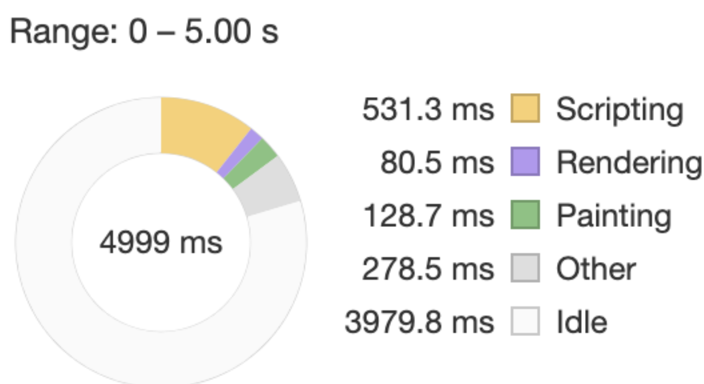


Kuva 7. Babylon.js toteutuksen FPS-graafi.

Seuraavaksi siirrytään tarkastelemaan Chromen tarjoaman diagnoosityökalun yhteenvetodiagrammia. Tämä diagrammi kertoo sen, kuinka paljon prosessori on käyttänyt aikaa erilaisiin toimenpiteisiin, kuten komentosarjojen suorittamiseen (scripting), renderointiin (rendering), piirtämiseen (painting), muihin toimenpiteisiin (other) ja kuinka paljon sille on jäänyt aikaa odotella seuraavia käskyjä (idle). Mitä vähemmän diagrammissa näkyy

odotusaikaa, sitä suuremmalla kuormalla prosessori on ollut sovellusta suorittaessa. Koska renderointi on tärkeä osa 3D-grafiikoiden parissa työskentelyä, sen osuus on tässä testissä merkittävässä roolissa. Mitä vähemmän prosessorin voidaan nähdä käyttäneen aikaa mallien renderoimiseen, sitä kevyempänä 3D-kirjasto voidaan nähdä.

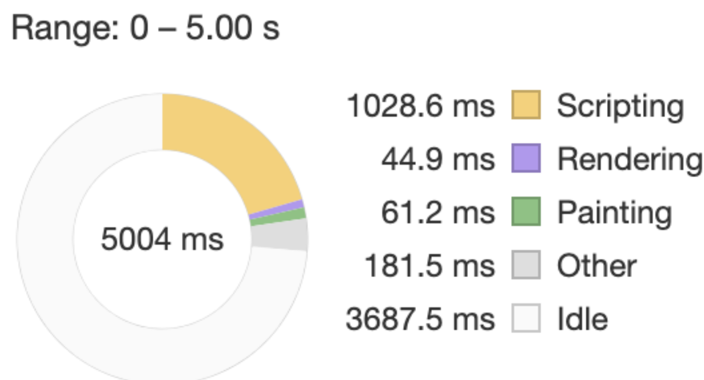
Kuvassa 8 on esitetty Three.js toteutuksen yhteenvetodiagrammi. Tästä nähdään, että viiden sekunnin mittauksen aikana komentosarjojen suoritukseen prosessorilla on kulunut aikaa noin puolisekuntia, renderointiin 80ms, piirtämiseen 128ms ja muihin toimenpiteisiin 278ms. Prosessorille on jäänyt aikaa uusien käskyjen odottamiseen lähes neljä sekuntia. Yhteenvetodiagrammin perusteella voidaan siis todeta, ettei Wiseplayer-imitaatio Three.js-kirjastolla toteuttaessa kuormita prosessoria juurikaan.



Kuva 8. Three.js toteutuksen yhteenvetodiagrammi.

Vastaavasti Babylon.js toteutuksen yhteenvetodiagrammi on esitelty kuvassa 9. Tästä voidaan tehdä havaintona se, että komentosarjojen suoritukseen prosessori on käyttänyt noin sekunnin, renderointiin noin 45ms, piirtämiseen 61ms ja muihin toimenpiteisiin 181ms. Kun näihin toimenpiteisiin kulunut aika vähennetään viidestä sekunnista, niin saadaan aika, jonka prosessori on käyttänyt uusien komentojen odottamiseen. Babylon.js toteutuksessa tämä odotusaika on n. 3,6 sekuntia.

Näiden kahden yhteenvetodiagrammin perusteella voidaan todeta, että Three.js on kokonaisuutena hieman Babylon.js:ää tehokkaampi Wiseplayer-imitaatiossa. Vaikka Babylon.js kuluttaa lähes puolet vähemmän prosessoriaikaa renderoinnissa ja piirtämisessä, sen komentosarjojen suoritukseen kuluu lähes kaksinkertaisesti Three.js komentosarjojen suoritusaika.



Kuva 9. Babylon.js toteutuksen yhteenvetodiagrammi.

Viimeinen tehokkuuteen liittyvä toimenpide on toteutettujen ohjelmien *koontiaika* (build time). Tämä testi toteutetaan ajamalla toteutettujen ohjelmahakemistojen juuressa Angularin tarjoama *ng build --prod* komento, joka kääntää ja kokoaa ohjelmiston. Tämä komento koostaa ohjelmasta ”tuotantoversion”, jota voidaan käyttää siinä vaiheessa, kun ohjelma on valmis tuotantoon.

Three.js toteutuksessa aikaa komennon suorittamiseen kului 37240ms, kun taas Babylon.js:llä vastaavan komennon suorittamiseen kului 81622ms, eli yli kaksinkertaisesti. Vaikka tällä testillä ei ole varsinaista merkitystä ohjelmiston tehokkuuden kannalta, sen avulla voidaan saada osviittaa kirjastojen koosta ja sen vaikutuksesta silloin, kun kirjasto on asennettuna projektiin. Kuten aiemmin jo mainittiin, niin Babylon.js on kirjastona yli kolme kertaa suurempi kuin Three.js, joka näyttäisi myös konkretisoituvan koontiajoissa. Lisäksi kuten aiemmin todettu, web-ohjelmointi on suurilta osin ohjelmakoodien kääntämistä ja lyhyemmät käännösajat parantavat kirjaston käytöstä syntyvää kokemusta. Eli kirjastojen koolla ja niiden käännösajoilla on vaikutusta myös käytettävyyteen.

Toteutettujen tehokkuustestien perusteella voidaan todeta, että kokonaisuutena Three.js on tehokkaampi kirjasto. Tämä käy ilmi kirjaston koosta, Chromen kehittäjätyökalun avulla koostetusta yhteenvetodiagrammista ja ohjelman koontiajasta. Babylon.js puolestaan jää kokonaisuutena hieman Three.js:n taakse, mutta sen kyky toimia animoinnin ja 3D-grafiikan kanssa ja niiden osalta oleellisempien tehtävien parissa on Three.js:ää edellä. Tämä tulee ilmi sen selkeästi paremmista FPS luvuista sekä yhteenvetodiagrammin renderointiin ja piirtämiseen kulutetuista ajoista. Vaikka muut osa-alueet jäävät Three.js:n varjoon, nämä kolme ovat toteutuksien kannalta kaikkein keskeisimmät osa-alueet. Tästä syystä voidaankin todeta, että pieniin ja melko yksinkertaisiin projekteihin Three.js on erittäin soveltuva kirjasto sen tehokkuuden kannalta, mutta suurempiin ja vaativampiin projekteihin, kuten esimerkiksi 3D-selainpeleihin olisi syytä valita Babylon.js.

7 Yhteenveto

Three.js ja Babylon.js ovat molemmat suuren suosion saaneita 3D-kirjastoja JavaScriptille, jotka tarjoavat erittäin monipuoliset käyttötarkoitukset ja niiden ollessa avoimen lähdekoodin kirjastoja puuttuvien ominaisuuksien toteuttaminen olisi mahdollista myös oma-aloitteisesti. Vaikka vertailuun valittiin McCallin laatutekijöistä ja ISO 9126 -standardista löytyviä yhteisiä tekijöitä, on syytä kuitenkin muistaa vertailun pienimuotoisuus ja spesifisyys, jonka vuoksi tuloksia ei voida yleistää.

Tutkimuksessa tehdyn yleisen vertailun perusteella Three.js on suositumpi ja suuremman edistäjäkunnan omaava kirjasto. Se tuottaa versiopäivityksiä tiheään tahtiin, minkä voidaan todeta olevan kehittyvän ja ylläpidettävän ohjelmiston positiivinen ominaisuus. Toisaalta tämä ominaisuus saattaa johtaa myös siihen, että kirjastoa käyttävien kehittäjien täytyy opetella kirjaston päivitysten yhteydessä uusia asioita, mikäli vanhat käytännöt ovat muuttuneet päivitysten myötä. Kirjastojen saamien tähtien ja edistäjien lukumäärien mukaan tämä ei kuitenkaan näyttäisi olevan ongelma Three.js:lle, sillä se on onnistunut saamaan selkeästi enemmän suosiota GitHubissa.

Ylläpidettävyyden näkökulmasta kirjastojen välillä ei ole suurta eroavaisuutta. Molemmat kirjastot kykenivät tarjoamaan ennalta määritetyille ominaisuuksille selkeät toteutustavat. Pieniä eroavaisuuksia kirjastojen välillä oli esimerkiksi niiden tarjoamien funktioiden ja attribuuttien osalta, mutta molemmissa kirjastoissa nämä olivat kuitenkin selkeästi tunnistettavissa eikä epäjohdonmukaisuuksia nimeämisten osalta ollut havaittavissa. Tavat, joilla ennalta määritetyt ominaisuudet toteutettaisiin, olivat molemmissa kirjastoissa lähes identtiset, jonka vuoksi kirjastoja ei voida asettaa paremmuusjärjestykseen ylläpidettävyyden osalta tämän tutkimuksen perusteella. On kuitenkin syytä ottaa huomioon, ettei tutkimuksessa toteutettu testi anna kovin kattavaa kuvaa kirjastojen ylläpidettävyydestä. Ylläpidettävyys on moniulotteinen käsite, jonka kokonaisvaltainen toteaminen ei muutaman ominaisuuden toteuttamistapojen perusteella ole mahdollista.

Käytettävyyden osalta tulokset olivat puolestaan hieman yksiselitteisempiä. Babylon.js kirjaston käyttöönotto oli selkeästi Three.js:ää heikompi, sillä sen yhteensopivuus esimerkiksi TypeScriptin ja Angularin kanssa oli huonompi kuin Three.js:llä. Lisäksi Babylon.js vaatii erillisen riippuvuuden asentamisen Loadereiden käyttämiseksi, mikä heikentää sen käytettävyyttä. Ideaalitilanne olisi, että yksi paketti tarjoaisi kaikki tarvittavat moduulit. Ohjelmakoodia tarkastellessa käytettävyys kallistuu kuitenkin Babylon.js:n puolelle, sillä samanlaisen ohjelman toteuttamiseksi se ei vaadi niin paljoa koodirivejä kuin Three.js, eikä se käytä niin paljoa luokkamuuttujia. Nämä puoltavat Babylon.js:n käytettävyyttä ohjelmakoodia katsoessa. Tästä huolimatta Three.js:n kyky tarjota kaikki tarvittavat moduulit yhden paketin sisältä ja yhteensopivuus TypeScriptin ja Angularin kanssa ovat käytettävyyden kannalta merkittävämmät tekijät, jonka vuoksi Three.js voidaan katsoa käytettävyydeltään onnistuneemmaksi kirjastoksi.

Viimeinen vertailtava laatutekijä oli tehokkuus, joka muodosti mielenkiintoisen asetelman näiden kahden kirjaston välillä. Three.js antoi tehokkaamman kokonaiskuvan olemalla muun muassa selkeästi pienempikokoisempi kirjasto, käyttämällä vähemmän prosessoriaikaa tehtävien suorittamiseksi sekä tarjoamalla huomattavasti lyhyemmän koontiajan. Näiden perusteella Three.js:n voitaisiin kuvitella olevan parempi kirjasto tehokkuuden kannalta. Babylon.js kuitenkin haastoi Three.js:n ehkä oleellisimpien osa-alueiden kohdalla, kun puhutaan 3D-grafiikasta ja sen visualisoimisesta, sillä Chromen kehittäjätyökaluilla mitattuna Babylon.js:n FPS lukemat olivat korkeammat ja säännöllisemmät sekä sen käyttämä prosessoriaika renderointiin ja piirtämiseen oli noin puolet pienempi kuin Three.js:llä. Näissä osa-alueissa pärjääminen takaa suurimmaksi osaksi sen, että 3D-grafiikka ja sen animointi näyttää sulavalta käyttäjälle. Tästä syystä tehokkuuden kannalta yksiselitteistä voittajaa ei ole, vaan voidaan todeta kummankin kirjaston soveltuvan erilaisiin tarpeisiin. Pienissä projekteissa Three.js vakuuttaa sen keveydellään, kun taas merkittävämmissä projekteissa Babylon.js voittaa suorituskyvyssä tarjoamalla paremman tehokkuuden tärkeimmillä osa-alueilla.

Kaiken kaikkiaan molemmat kirjastot ovat erittäin hyvin soveltuvia Wiseplayerin käyttöön. Wiseplayerin käyttötarkoitus on kuitenkin melko spesifi eikä se 3D-graafisesta näkökulmasta ole kovin raskas sovellus, jonka vuoksi siirtyminen Three.js:stä Babylon.js:ään ei ole välttämätöntä. On kuitenkin hyvä pitää mielessä, että mikäli Wiseplayer kasvaa vaativammaksi ja monipuolisemmaksi 3D-toistimeksi, niin siirtymistä Babylon.js:ään olisi syytä harkita. Kirjastot ovat kuitenkin arkkitehtuuriltaan hyvin samankaltaisia, jonka vuoksi siirtyminen kirjastosta toiseen voisi olla kohtuullisen vaivatonta.

8 Viiteluettelo

Angular Guide. 2019a. *Lifecycle Hooks*. <https://angular.io/guide/lifecycle-hooks> (Käytetty 3.4.2019).

Angular Guide. 2019b. *Angular versioning and releases.* <https://angular.io/guide/releases> (Käytetty 10.4.2019).

Sumeet Arora. 2014. *WebGL Game Development*. Packt Publishing Ltd.

Babylon.js dokumentaatio. 2019a. <https://doc.babylonjs.com/> (Käytetty 13.2.2019).

Babylon.js dokumentaatio. 2019b. *Euler Angles and Quaternions.* <https://doc.babylonjs.com/resources/rotation_conventions> (Käytetty 10.4.2019).

Babylon.js dokumentaatio. 2019c. *Optimize your scene.* <https://doc.babylonjs.com/how_to/optimizing_your_scene#not-updating-the-bounding-info> (Käytetty 10.4.2019).

Anahid Basiri, Elena Simona Lohan, Terry Moore, Adam Winstanley, Pekka Peltola, Chris Hill, Pouria Amirian and Pedro Figueiredo e Silva. 2017. Indoor location based services challenges, requirements and usability of current solutions. *Computer Science Review,* 24, 1–12.

Igor Bisio, Fabio Lavagetto, Mario Marchese and Andrea Sciarrone. 2013. Energy efficient WiFi-based fingerprinting foor indoor positioning with smartphones. *IEEE Global Communications Conference (GLOBECOM),* 4639–4643.

Bitwise. 2019. *Wisehockey Solution Brief.* <https://wisehockey.com/wp-content/uploads/2018/05/wh_solution_brief.pdf> (Käytetty 20.2.2019).

Diego Cantor and Brandon Jones. 2012. *WebGL Beginner's Guide*. Packt Publishing Ltd.

Daniel Chandler and Rod Munday. 2016. *A Dictionary of Media and Communication (2 ed.).* Oxford University Press.

Pete Bacon Darwin. 2018. *Stable AngularJS and Long Term Support.* <https://blog.angular.io/stable-angularjs-and-long-term-support-7e077635ee9c> (Käytetty 1.1.2019).

Peter Bacon Darwin and Pawel Kozlowski. 2013. *AngularJS Web Application Development.* Packt Publishing Ltd.

Jos Dirksen. 2013. *Learning Three.js: The JavaScript 3D Library for WebGL*. Packt Publishing Ltd.

Maeve Duggan and Aaron Smith. 2013. *42% of online adults use multiple social networking sites, but Facebook remains the platform of choice*. Social Media Update 2013. Pew Research Center.

Tiago Gaspar and Paulo Oliveira. 2009. Single Pan and Tilt Camera Indoor Positioning and Tracking System. In: *Proceedings of the European Control Conference*, 2792–2797.

GitHub Help. 2019. *About stars.* <https://help.github.com/articles/about-stars/> (Käytetty 13.2.2019).

GitHub Threejs. 2019. <https://github.com/mrdoob/three.js> (Käytetty 15.4.2019).

GitHub Babylon.js. 2019. <https://github.com/BabylonJS/Babylon.js> (Käytetty 15.4.2019).

Google Developers. 2019a. *Chrome DevTools*. <https://developers.google.com/web/tools/chrome-devtools/> (Käytetty 19.4.2019).

Google Developers. 2019b. *Get Started with Analyzing Runtime Performance*. <https://developers.google.com/web/tools/chrome-devtools/evaluate-performance/> (Käytetty 19.4.2019).

Naresh Gupta. 2013. *Inside Bluetooth Low Energy*. Artech House.

Bruno Hartmann. 2017. Migrate to Angular: why and how you should do it. UruIT Blog. <*https://www.uruit.com/blog/angular-1-vs-2-migrate/*> (Käytetty 6.2.2019).

Insoft. 2018. Indoor Positioning Services. <https://cdn.infsoft.com/www/images/solutions/basics/whitepaper/infsoft-Whitepaper-EN-Indoor-Positioning_download.pdf> (Käytetty 29.11.2018).

Magne Jørgensen. 1999. Software quality measurements. *Advances in Engineering Software 53*, 12, 907–912.

Raymond Kehoe and Alka Jarvis. 2012. *ISO 9000-3: A Tool for Software Product and Process Improvement.* Springer Science & Business Media.

Jules Kremer. 2016. *AngularJS*. <https://blog.angularjs.org/2016/09/angular2-final.html> (Käytetty 1.1.2019).

MDN web docs. 2019a. *Window: DOMContentLoaded event*. <https://developer.mozilla.org/en-US/docs/Web/API/Window/DOMContentLoaded_event> (Käytetty 9.4.2019).

MDN web docs. 2019b. *window.requestAnimationFrame().* <https://developer.mozilla.org/en-US/docs/Web/API/window/requestAnimationFrame> (Käytetty 9.4.2019).

Erikson Morais, Siome Goldenstein, Anselmo Ferreira and Anderson Rocha. 2012. Automatic Tracking of Indoor Soccer Players Using Videos from Multiple Cameras. In: *Conference on Graphics, Patterns and Images*, 174–181.

Julien Moreau-Mathis. 2016. *Babylon.js Essentials*. Packt Publishing Ltd.

Mozilla web docs. *JavaScript language resources. <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Language_Resources*> (Käytetty 6.2.2019).

Kshirasagar Naik and Priyadarshi Tripathy. 2008. *Software Testing and Quality Assurance: Theroy and Practice.* John Wiley & Sons, Inc.

Christopher Nance. 2014. *TypeScript Essentials*. Packt Publishing Ltd.

Ramjee Prasad and Marina Ruggieri. 2005. *Applied Satellite Navigation Using GPS, GALILEO and Augmentation Systems*. Artech House.

Quuppa. 2019. <https://quuppa.com/>. (Käytetty 20.2.2019).

B. Rama Rao, W. Kunysz, R. Fante and K. McDonald. 2012. *GPS/GNSS Antennas.* Artech House.

Lawrence D. Spencer and Seth Richards. 2015. *Reliable JavaScript: How to Code Safely in the World's Most Dangerous Language*.

Isaac Sukin. 2013. *Game Development with Three.js*. Packt Publishing Ltd.

Three.js. <https://threejs.org/> (Käytetty 1.1.2019).

TypeScript Documentation. *Interfaces.* <https://www.typescriptlang.org/docs/handbook/interfaces.html> (Käytetty 10.2.2019).

Paul Wilton and Jeremy McPeak. 2007. *Beginning JavaScript.* Wiley Publishing, Inc.

Chouchang Yang and Huai-rong Shao. 2015. WiFi-based indoor positioning. *IEEE Communications Magazine 53,* 3, 150–157.

Yuan Zhuang, Jun Yang, You Li, Longning Qi and Naser El-Sheimy. 2016. Smartphone-based Indoor Localization with Bluetooth Low Energy Beacons. *Sensors*, 16, 596.