GPS- ja sensoridatan hyödyntäminen mobiililaitteiden pelimoottoriarkkitehtuurissa

Seppo Tompuri

Pro gradu -tutkielma
HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

Helsinki, 30. huhtikuuta 2014

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta/Osasto – Fakultet/Sektion –Faculty/Section	Laitos – Institution – Department	
Matemaattis-luonnontieteellinen	Tietojenkäsittelytieteen laitos	
Tekijä – Författare – Author		
Seppo Tompuri		
Työn nimi – Arbetets titel – Title		
GPS- ja sensoridatan hyödyntäminen mobiililaitteiden pelimoottoriarkkitehtuurissa		
Oppiaine – Läroämne – Subject		
Tietojenkäsittelytiede		
Työn laji – Arbetets art – Level	Aika – Datum – Month and year	Sivumäärä – Sidoantal – Number of pages
Pro gradu -tutkielma	30.4.2014	75 sivua + 7 liitesivua

Tiivistelmä – Referat – Abstract

Tietokonepelit on kehitetty perinteisesti joko pöytätietokoneille tai pelikonsoleille. Räjähdysmäisesti kasvaneet mobiilipelimarkkinat ovat kuitenkin haastaneet nämä pelialustat. Mobiilipelien kehitys tapahtuu usein samalla tyylillä kuin pöytätietokoneille ja pelikonsoleille, vaikka niistä löytyy pöytätietokoneista ja pelikonsoleista poikkeavaa tekniikkaa, joka mahdollistaa perinteisestä poikkeavan käyttäjäsyötteen. Nykyaikaisista mobiililaitteista löytyy muun muassa erilaisia ympäristöä tarkkailevia sensoreita sekä usein myös GPS-vastaanotin.

GPS-vastaanotin tarjoaa pelin käyttöön pelaajan sijaintitiedon. Sensorit puolestaan tarkkailevat mobiililaitteen ympäristöä, kuten mobiililaitteen kiihtyvyyksiä kolmessa ulottuvuudessa. Sensoreilta saatua dataa voidaan käyttää käyttäjäsyötteessä joko suoraan tai muokattuna. Niiltä saadun datan avulla on myös mahdollista päätellä pelaajan tekemät eleet, jotka voidaan hahmontunnistuksen avulla sitoa osaksi pelin käyttäjäsyötettä. Tällainen mobiililaitteiden mahdollistama uudenlainen käyttäjäsyöte mahdollistaa uudenlaisia peligenrejä, jotka käyttävät pelaajan liikettä ja sijaintitietoa osana pelimekaniikkaa.

Tämä työ esittelee ne pelimoottorin osat jotka ovat mukana GPS- ja sensoridatan keräämisessä ja käsittelyssä sekä esittelee tässä työssä suunnitellun GPS- ja sensoridataa hyödyntävän arkkitehtuurimallin mobiililaitteille. Työssä perehdytään aluksi aiheeseen liittyviin käsitteisiin sekä pelin reaaliaikaisuudesta huolehtivaan pelisilmukkaan ja sen erilaisiin arkkitehtuurimalleihin. Tämän jälkeen käydään läpi pelimoottorien suoritusaikaisesta arkkitehtuurista ne puolet, jotka liittyvät GPS- ja sensoridatan keräämiseen. Lopuksi esitellään ja arvioidaan tässä työssä suunniteltu arkkitehtuurimalli GPS- ja sensoridatan hyödyntämiselle mobiililaitteiden pelimoottoriarkkitehtuurissa. Työn ohessa tämän ehdotetun arkkitehtuurimallin toimivuus todennetaan Windows Phone 8 laitealustalle tehdyllä toteutuksella, jonka lähdekoodia käytetään apuna ehdotetun mallin esittelyssä.

ACM Computing Classification System (CCS):
Software and its engineering → Interactive games
Computer systems organization → Multicore architectures
Software and its engineering → Software architectures

Avainsanat – Nyckelord – Keywords

Mobiilipeli, mobiilipelimoottori, pelimoottori, sensoridata, paikkatieto

Säilytyspaikka – Förvaringställe – Where deposited

Helsingin yliopiston kirjaston E-thesis arkisto

Muita tietoja – Övriga uppgifter – Additional information

Sisältö

1 Johdanto **1**

2 Keskeiset käsitteet **3**

2.1 Mobiililaite....3

2.2 Sensori....6

2.3 GPS-vastaanotin....7

2.4 Pelimoottori....8

3 Pelisilmukan arkkitehtuuri **15**

3.1 Pelisilmukan päivitystehtävien rinnakkaistaminen....15

3.2 Yleiset pelisilmukkamallit....19

3.3 Mobiililaitteiden vaatimukset pelisilmukalle....24

3.4 Pelisilmukka-arkkitehtuurin suunnittelu mobiililaitteelle....27

4 Pelimoottorin sisäinen kommunikointi **31**

4.1 Tarkkailijamalli....31

4.2 Tapahtumajärjestelmä....32

4.3 Liitutaulumalli....32

5 Käyttäjäsyöte pelimoottoriarkkitehtuurissa **33**

5.1 Käyttäjäsyötteen käsittelyn pääperiaatteet....33

5.2 GPS- ja sensoridata käyttäjäsyötteessä....35

5.3 Hahmontunnistus käyttäjäsyötteestä....41

6 Ehdotettu arkkitehtuurimalli GPS- ja sensoridatan käsittelyyn **48**

6.1 Ehdotetun mallin tausta....48

6.2 Pelisilmukan arkkitehtuuri ehdotetussa mallissa....50

6.3 Käyttäjäsyötteen käsittely ehdotetussa mallissa....54

6.4 Ehdotetun mallin kokonaisarkkitehtuuri....59

7 Ehdotetun mallin toimivuus mobiilipeleissä **62**

7.1 Pelien asettamat yleiset vaatimukset käyttäjäsyötteelle....62

7.2 Sensoridatan käyttö pelien tarvitsemilla tavoilla....66

7.3 GPS-datan käyttö pelien tarvitsemilla tavoilla....67

7.4 Ehdotetun mallin skaalautuvuus....68

8 Yhteenveto **68**

Lähteet **71**

Liite 1. Yleiset ominaisuudet tämän hetken mobiililaitteissa **76**

Liite 2. GPS ja sensorit tämän hetken mobiililaitteissa **77**

Liite 3: Mobiililaitekäyttöjärjestelmien yleisyys 2013 **78**

Liite 4. Yksisäikeinen vakiotaajuudellinen sitomaton pelisilmukka **79**

Liite 5. Ehdotetun mallin mukainen pelisilmukan toteutus kaksiytimisille mobiililaitteille **80**

Liite 6. Ehdotetun mallin mukainen pelisilmukan toteutus neliytimisille mobiililaitteille - Päivitystehtävä **81**

Liite 7. Liitutaulumallin toteutus **82**

1 Johdanto

Mobiililaite voi tarkoittaa mitä tahansa liikuteltavaa elektronista laitetta kuten matkapuhelinta, kannettavaa tietokonetta tai taulutietokonetta [Oxf14, Cam14]. Tämä työ kohdistuu mobiililaitteista älypuhelimiin ja taulutietokoneisiin, sillä niistä löytyy perinteisistä pöytätietokoneista poikkeavaa tekniikkaa, joka mahdollistaa uudenlaisen käyttäjäsyötteen hyödyntämisen sovelluksissa. Nykyaikaisista mobiililaitteista löytyy yleensä muun muassa GPS-vastaanotin sekä erilaisia ympäristöä tarkkailevia sensoreita, joita voidaan hyödyntää käyttäjäsyötteen keräämisessä. Kyseiset käyttäjäsyötekanavat ovat mielenkiintoisia etenkin pelisovellusten kannalta, koska ne mahdollistavat uudentyylisiä pelityyppejä ottamalla muun muassa pelaajan liike-eleet ja paikkatiedon osaksi pelimekaniikkaa. Liike-eleet saadaan osaksi käyttäjäsyötettä hahmontunnistuksen avulla. Sen käyttäminen osana pelin käyttäjäsyötettä eroaa kuitenkin käyttäjäsyötelaitteiden käyttämisestä ja vaatii syötteen keräämisen lisäksi normaalista poikkeavaa käyttäjäsyötteen käsittelyä. Tämän vuoksi hahmontunnistuksen toteuttamiseen pelimoottorissa tulee kiinnittää erityistä huomiota.

Nykyiset pelit luodaan yleensä jonkin valmiin pelimoottorin päälle. Pelimoottoriksi kutsutaan sitä yleistettävää osaa pelin toteutuksesta, jota ei ole sidottu pelikohtaiseen mekaniikkaan. Pelimoottori hoitaa monien muiden toimien lisäksi yleensä myös käyttäjäsyötteen keräämisen ja käsittelyn pelinkehittäjän puolesta, ja pelinkehittäjä hyödyntää tarjottua syötedataa tietämättä, kuinka data on syötelaitteelta saatu ja kuinka sitä on mahdollisesti käsitelty. Saatavilla olevat pelimoottorit ja niihin liittyvä materiaali keskittyvät yleensä pöytätietokoneissa ja pelikonsoleissa suoritettaviin peleihin. Vaikka tämän työn aihetta sivuavia tutkimuksia löytyykin [esimerkiksi APH09 ja JoC09], niin tutkimuksia GPS-vastaanottimen ja sensoreiden hyödyntämisestä mobiililaitteiden pelimoottorin arkkitehtuurissa ei löydy. Olemassaolevat GPS- ja sensoridataa hyödyntävät pelit eivät paljasta toteutukseensa liittyviä yksityiskohtia, joten jos mobiilipelin kehittäjä haluaa pelinsä käyttävän GPS- ja sensoridataa osana pelin käyttäjäsyötettä, eikä hän halua tai pysty, tekemään peliään jonkin tarjolla olevan pelimoottorin päälle, täytyy hänen tehdä omat yksilölliset ratkaisunsa näiden tekniikoiden hyödyntämiseen.

Tämän työn tavoitteena on ehdottaa arkkitehtuurimallia GPS- ja sensoridatan

hyödyntämiseen mobiililaitteiden pelimoottorissa. Ehdotetun arkkitehtuurimallin (jatkossa *ehdotettu malli*) suunnitteluun kuuluu pelimoottorista ne osat jotka ovat mukana GPS- ja sensoridatan lukemisessa ja käsittelyssä. Tavoitteena ehdotetun mallin suunnittelussa on luoda sellainen skaalautuva ja alustariippumaton arkkitehtuurimalli GPS- ja sensoridataa hyödyntämiselle peleissä, jonka pohjalta on mahdollista kehittää pelejä mobiililaitteille. Tämän vuoksi ehdotetun mallin toimivuutta myös arvioidaan tutkimalla mobiililaitteiden pelimoottorien vaatimuksia käyttäjäsyötteen keräämiselle ja käsittelylle.

Ehdotetun mallin toimivuus käytännössä todennetaan Windows Phone 8 -laitealustalle tehdyllä toteutuksella, ja toteutuksen lähdekoodia käytetään apuna esittelemään ehdotettua mallia ja sen mahdollista toteutustapaa. Ehdotettu malli on laitealustariippumaton, joten toimivuuden todentamisessa käytetyllä toteutusympäristöllä ei ole vaikutusta. Windows Phone 8 valittiin toteutusympäristöksi, koska tutkimuksen tekijän aikaisempi kokemus tuki kyseisen laitealustan käyttöä.

Mobiililaitteiden rajallinen virtalähde asettaa peleille omanlaisia rajoituksia verrattuna verkkovirtaan kytkettyihin pöytätietokoneisiin ja pelikonsoleihin. Mobiililaitteissa suoritettavien pelien arkkitehtuuri tuleekin suunnitella kuormittamaan mahdollisimman vähän prosessoriytimiä, jotta ne kuluttaisivat osaltaan mahdollisimman vähän virtaa. Tämä taas onnistuu vähentämällä suoritettavien tehtävien aiheuttamaa prosessoriytimien kuormitusta. Koska pelin eri tehtävät suoritetaan toistuvasti pelisilmukka-arkkitehtuurin määrittämällä tavalla, vaikuttaa tehtävien toistonopeus suoraan prosessoriytimien kuormitukseen. Yksi tavoite ehdotetulle arkkitehtuurimallille onkin optimoida pelisilmukan tehtävien toistonopeudet, jolloin mobiililaitteen virrankulutus saadaan minimoitua pelin suorituksen aikana.

Tämä työ on lähestymistavaltaan suunnittelutieteellinen, eli konstruktiivinen tutkimus. Aluksi kerätään tietoa työhön kuuluvista aihealuista, ja tämän tiedon pohjalta suunnitellaan arkkitehtuurimalli GPS- ja sensoridatan hyödyntämiseen mobiililaitteiden pelimoottoriarkkitehtuurissa. Vaikka ehdotettu malli on tämän työn pääasiallinen konstruktiivinen osuus, kuuluu työhön myös ehdotetun mallin pohjalta tehty toteutus Windows Phone 8 -laitealustalle. Tämän toteutuksen avulla todennetaan ehdotetun mallin toimivuus. Toteutuksen lähdekoodia käytetään esittelemään suunniteltua arkkitehtuurimallia ja mahdollista toteutustapaa sen osille.

Jotta GPS- ja sensoridataan liittyvät osuudet voidaan lisätä pelimoottoriarkkitehtuuriin, täytyy ensin ymmärtää pelimoottorin suoritusaikaisen osan yleinen arkkitehtuuri, sekä käyttäjäsyötteen lukeminen ja käsittely siinä. Luvussa 2 perehdytään keskeisiin käsitteisiin, kuten pelimoottoriin ja sen komponentteihin. Luvussa 3 tutkitaan pelimoottorien suoritusaikaiseen osaan liittyviä erilaisia pelisilmukkamalleja. Pelisilmukka määrää missä järjestyksessä ja millä tyylillä pelin eri perustoiminnot, kuten käyttäjäsyötteen kerääminen ja prosessointi, suoritetaan, joten se on keskeinen osa myös GPS- ja sensoridatan käyttämisessä.

Tämän työn kannalta merkittävää on myös se, kuinka pelin eri komponentit kommunikoivat keskenään, jotta GPS- ja sensoridatan vaikutus saadaan välitettyä tarvittaviin pelin komponentteihin. Luvussa 4 perehdytään miten pelit hoitavat tämän sisäisen kommunikoinnin eri komponenttien välillä. Luvussa 5 käsitellään käyttäjäsyötteen keräämistä ja käsittelyä yleisellä tasolla, sekä miten GPS- ja sensoridata sidotaan käyttäjäsyötteeseen ja kuinka jatkuvasta GPS- ja sensoridatavirrasta voidaan havaita hahmokuvioita eli eleitä. Luvussa 5 käydään lisäksi läpi kuinka nämä eleet voidaan sitoa osaksi käyttäjäsyötettä. Luvussa 6 esitellään tässä työssä ehdotettua mallia GPS- ja sensoridatan hyödyntämiseen mobiililaitteiden peleissä. Luvussa 7 tarkastellaan, kuinka tämä ehdotettu malli täyttää pelien yleiset vaatimukset käyttäjäsyötteen käsittelylle ja olemassaolevien pelien tavalle hyödyntää GPS- tai sensoridataa sekä tarkastellaan ehdotetun mallin skaalautuvuutta uusien sensoreiden ja kasvavien prosessoriytimien suhteen. Lopun yhteenvedossa koostetaan työssä käsitellyt asiat ja työn tulokset sekä esitellään mahdolliset tämän työn tuloksia hyödyntävät jatkotutkimusaiheet.

2 Keskeiset käsitteet

Tässä luvussa määritellään tälle työlle keskeiset käsitteet. Tämän lisäksi tässä luvussa tarkastellaan mobiililaitteiden ja pelisilmukoiden yleisiä ominaisuuksia.

2.1 Mobiililaite

2.1.1 Määritelmä

Mobiililaite määritellään liikuteltavaksi elektroniseksi laitteeksi, kuten matkapuhelin tai taulutietokone [Oxf14, Cam14]. Mobiililaitteista tässä luvussa keskitytään taulutietokoneisiin ja älypuhelimiin, koska niistä löytyy tämän työn kannalta mielenkiintoiset GPS-vastaanotin ja erilaisia sensoreita.

Älypuhelin määritellään ”matkapuhelimeksi, joka pystyy suorittamaan monia tietokoneelle tyypillisiä toimintoja, jolla on yleensä suhteellisen suuri näyttö sekä käyttöjärjestelmä joka pystyy suorittamaan yleisiä ohjelmia” [Oxf14]. Älypuhelimelle ei ole yhtä yleistä ja yksiselitteistä määritelmää, mutta Oxfordin sanakirjan määritelmä noudattaa muiden määritelmien [Cam14, Tec12, Wik14] konsensusta. Muiden lähteiden määritelmistä selviää, että älypuhelimen odotetaan tarjoavan myös paljon muita ominaisuuksia, kuten

- erilaisia yhteystekniikoita, kuten langaton verkko (WLAN), Bluetooth ja USB,
- kamera,
- GPS, tai muu GNNS (*Global Navigation Satelite System*) -vastaanotin ja
- sovelluksia, kuten valokuvien ja videoiden galleria, valokuvien muokkaus, musiikkisoitin, sähköpostien ja mediaa sisältävien viestin lähetys, kolmannen osapuolen sovellusten lataus ja asennus, navigaattori ja www-selain.

Taulutietokone taas määritellään pieneksi tietokoneeksi, jota käytetään hiiren ja näppäimistön sijaan näyttöä koskettamalla [Oxf14, Cam14]. Taulutietokoneilta odotetaan perusominaisuuksia, kuten

- erilaisia sensoreita, kuten kamera, mikrofoni, kiihtyvyysanturi ja kosketusnäyttö,
- näytön koko seitsemästä tuumasta ylöspäin,
- yhteystekniikoita, kuten langaton verkko (WLAN), Bluetooth ja USB,
- kamera,
- GPS,
- sovelluksia: www-selain, kolmannen osapuolen sovellusten lataus ja asennus, videoiden katselu, sähköpostien ja mediaa sisältävien viestin lähetys ja

- mahdollisesti matkapuhelinten toiminnallisuutta: viestit, kaiutinpuhelin, osoitekirja, videopuhelut [Wik14].

Vertaamalla taulutietokoneilta ja älypuhelimilta odotettuja yleisiä ominaisuuksia, huomataan niiden samankaltaisuus. Taulutietokoneet ja älypuhelimet ovat ominaisuksiltaan lähellä toisiaan, mutta siinä missä älypuhelimet hoitavat matkapuhelinten tehtäviä, voidaan taulutietokoneiden ajatella hoitavan tietokoneiden tehtäviä. Erona näiden laiteryhmien välillä on lähinnä matkapuhelinominaisuudet, joita taulutietokoneissa ei yleensä ole. Tässä työssä oletetaan mobiililaitteista löytyvän GPS-vastaanottimen ja käyttäjäsyötelaitteina toimivien sensoreiden sopivan yhteen yllä olevien määritelmien kanssa.

2.1.2 Yleiset ominaisuudet

Mobiililaitteiden myyntitilastoista saadaan selville eniten myyneet mobiililaitteet. Tutkimalla näiden nykyaikana eniten myytyjen mobiililaitteiden ominaisuuksia, voidaan taas arvioida tällä hetkellä yleisimpien käytössä olevien mobiililaitteiden ominaisuuksia. Nykyaikaisten mobiililaitteiden yleisiä ominaisuuksia tutkittiin keräämällä tietoja myydyimmistä älypuhelimista ja taulutietokoneista. Älypuhelimista tutkittiin tuoreinta löydettyä myyntitilastoa maailmanlaajuisesta älypuhelinten myynnistä. Se listasi vuoden 2014 helmikuussa eniten myyneet laitemallit. Yhteensä älypuhelimia tutkimukseen valittiin 11 kappaletta: kymmenen eniten maailmassa myynyttä ja yksi eniten myynyt Windows Phone -laitealustan älypuhelin koska yksikään kymmenestä eniten myyneestä älypuhelimesta ei olut sellainen. Taulutietokoneille ei ollut saatavilla mallikohtaisia myyntitilastoja, mutta niille löytyi valmistajakohtaiset myyntitilastot. Tähän työhön tutkittavaksi valitut taulutietokoneet saatiin valitsemalla vuonna 2013 eniten maailmanlaajuisesti myyneiltä valmistajilta 1-2 suosittua laitemallia. Yhteensä taulutietokoneita tutkimukseen valittiin 11 kappaletta.

Mobiililaitteista kerätyt tiedot näkyvät liitteessä 1, josta käy muun muassa ilmi, että älypuhelimien ja taulutietokoneiden suoritustehoon vaikuttavat tekniikat, kuten CPU, GPU ja muisti, ovat lähellä toisiaan. GPS-vastaanottimen ja sensoreiden löytymistä nykyisistä mobiililaitteista käsitellään seuraavassa luvussa.

Suurin ero mobiililaiteryhmien välillä on taulutietokoneiden suurempi näyttö, ja se, ettei kaikissa taulutietokoneissa ole operaattoripohjaista verkkoyhteyttä. Molempien

laiteryhmien laitteista löytyy pääsääntöisesti moniydinprosessori, grafiikkaprosessori, GPS, erilaisia sensoreita ja kosketusnäyttö.

Suurimmassa osassa tämän hetken älypuhelimia ja taulutietokoneita on sellainen moniydinprosessori, jossa on kaksi tai neljä ydintä, ja erillinen suoritin grafiikalle [Liite 1]. Lisäksi muistia laitteissa on yleensä vähintään 512 MB. Kaikista listatuista älypuhelimista ja taulutietokoneista löytyy kiihtyvyysanturi, GPS, kosketusnäyttö ja monipuoliset yhteystekniikat, kuten WLAN, USB ja Bluetooth. Kaikista valituista laitteista löytyi grafiikan prosessointiyksikkö (GPU), jollainen puuttuu nykyään vain joistakin halvimmista älypuhelimista.

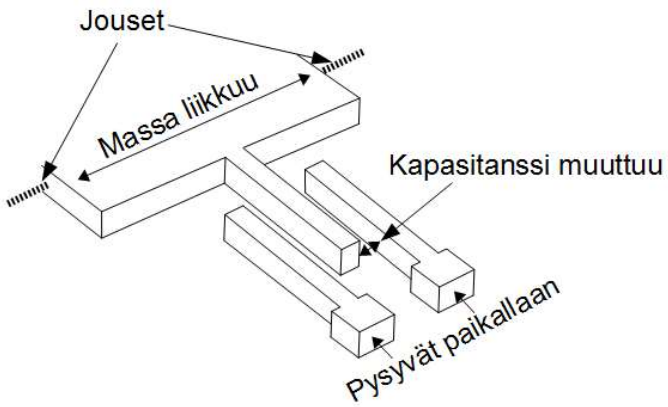
Itse tutkittu otos ei tuonut esille eroavaisuuksia edellisessä luvussa olevien älypuhelinten ja taulutietokoneiden yleisiin määritelmiin, vaan se vahvistaa niiden paikkansa pitävyyden. Kaikista valituista mobiililaitteista löytyvät ne ominaisuudet, joita niiltä näiden määritelmien mukaan myös odotetaan.

2.2 Sensori

Sensori on muuntaja, joka mittaa fyysistä ulottuvuutta ja muuntaa sen muotoon, jota ulkoinen tarkkailija, tai väline, voi lukea [Wik14c]. Esimerkki yksinkertaisesta sensorista on nestepohjainen lämpömittari, jossa lasiputki on täytetty niin sanottuun kalibrointipisteeseen asti esimerkiksi petrolilla. Tällainen lämpömittari muuntaa lämpötilasuureen nesteen laajenemisen ja supistumisen kautta havainnoitavaksi lämpötilaksi. Vastaava lämpötilanmittaus voidaan tehdä myös termoparilla, joka sopii paremmin elektronisiin laitteisiin. Termopari toimintaidea perustuu kahden eri metallin väliseen jännitteeseen, joka muuttuu lämpötilan mukana.

Mobiililaitteiden yleisin sensori on kiihtyvyysanturi, joka muuttaa fyysisesti havaittavan kiihtyvyyden jännitevaihtelun avulla mitattavaksi suureeksi. Mobiililaitteissa yleisesti käytetty kiihtyvyysanturi koostuu jousten varassa olevasta massasta, ja massaan kiinnitetystä ulokkeesta jonka etäisyys vaihtelee kiinteistä seinäkkeistä kiihtyvyyden mukana [Kuva 1]. Etäisyyden vaihtelu taas aiheuttaa mitattavassa kapasitanssissa muutoksia. Yksi tällainen anturi mittaa kiihtyvyyden yhdessä tasossa, joten niitä pitää olla kaksi tai useampi kiihtyvyyden kolmiulotteiseen mittaamiseen.

Jos mikrofoni ja kosketusnäyttö jätetään laskuista, kiihtyvyysanturi on ainoa asemansa vakiinnuttanut sensori nykyaikaisissa mobiililaitteissa [Liite 2]. Älypuhelimissa näytön sammuttava läheisyyssensori löytyy kaikista laitteista, mutta taulutietokoneista se yleensä puuttuu. Ympäristön valoisuuden tunnistin löytyy melkein kaikista mobiililaitteista.



Kuva 1: Mobiililaitteiden kiihtyvyysanturin toimintaperiaate, muokattu lähteestä [Mac14].

Magnetometri, jota kutsutaan myös digitaaliseksi kompassiksi, sekä gyroskooppi löytyvät lähes kaikista älypuhelimista ja taulutietokoneista, mutta aivan halvimmista älypuhelinmalleista ne voivat puuttua. Uusin mobiililaitteisiin tullut sensori on sormenjälkisensori, joka on käytössä Applen uusimmassa älypuhelinten lippulaivamallissa iPhone 5s. Toinen sensorikohtainen erikoisuus on ilmanpainesensori, barometri, jonka Samsung on lisännyt joihinkin laitteisiinsa.

Sensoreista kiihtyvyysanturi, gyroskooppi ja magnetometri ovat yleisimmät käyttäjäsyötelaitteina käytetyt sensorit. Ne tarjoavat pelin käyttöön muun muassa laitteen kallistuskulman kaikissa kolmessa ulottuvuudessa, kiihtyvyyden kaikissa kolmessa ulottuvuudessa ja laitteen suuntautumisen reaalimaailmassa. Magnetometri mittaa ympäröiviä magneettikenttiä, eli myös maan magneettikenttiä, ja sen avulla voidaan toteuttaa kompassi. Gyroskooppi mittaa laitteen kallistuskulmia kolmessa ulottuvuudessa. Kiihtyvyysanturi tarjoaa erilaisia mahdollisuuksia käyttäjäsyötteelle. Kiihtyvyyttä tarkkailemalla on mahdollista laskea mobiililaitteen tekemän liikkeen suunta kolmessa ulottuvuudessa, suunnan muutos ja voimakkuus. Jokainen mobiililaitteella tehty liike pitää sisällään erilaisen kiihtyvyysjäljen, tavallaan sormenjäljen, josta voidaan arvioida millainen liike on kyseessä [NKC12]. Kiihtyvyysanturilta saatavaa dataa voidaan käyttää siis suoran peliohjauksen lisäksi myös pelaajan tekemien liike-eleiden tunnistamiseen [KPT10].

2.3 GPS-vastaanotin

GPS-vastaanotin on yleistynyt mobiililaitteissa. Se löytyy miltei kaikista älypuhelimista ja taulutietokoneista [Liite 2]. GPS (*Global Positioning System*) on yleisin mobiililaitteiden käyttämä satelliittipaikannusjärjestelmä (*Global Navigation Satelite System*, GNNS), jonka on kehittänyt ja rahoittanut Yhdysvaltain puolustusministeriö [Wik14b]. Se perustuu 24:ään maata kiertävään satelliittiin, joiden lähettämien signaalien avulla voidaan laskea GPS-vastaanottimen sijainti. Satelliitit lähettävät GPS-vastaanottimelle atomikellon ajan ja navigaatiosignaalin. GPS-vastaanotin pystyy määrittämään sijaintinsa, kunhan se pystyy vastaanottamaan signaalin vähintään neljältä satelliitilta. GPS-paikannuksen tarkkuus siviilikäytössä parani huomattavasti vuoden 2000 alussa, kun Yhdysvaltojen puolustusministeriö lopetti signaalin tahallisen heikentämisen. GPS-järjestelmällä paikka voidaan määritellä muutaman metrin tarkkuudella vaakasuunnassa ja 2-3 kertaa heikommalla tarkkuudella korkeussuunnassa. Mobiililaitteiden laitealustat tarjoavat sovellukselle yleensä käyttöön seuraavia tietoja: leveysaste, pituusaste, korkeus, horisontaalinen tarkkuus, vertikaalinen tarkkuus ja nopeus [Loc14, CLL14, Geo14].

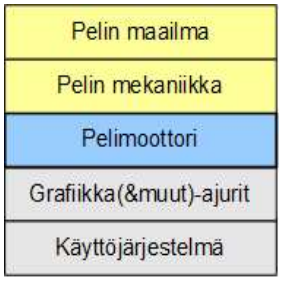
2.4 Pelimoottori

Tieteellisiä tutkimuksia pelimoottoreista on tehty vähän, joten lähdeaineistona pelimoottoreita tutkiessa voitaisiin käyttää nykyaikaisia kaupallisia pelejä, joiden pelimoottorien arkkitehtuurien rakentamiseen on panostettu paljon aikaa ja rahaa. Tämä ei kuitenkaan ole mahdollista, sillä niiden arkkitehtuuri on tarkoin varjeltu salaisuus, eikä niissä tehtyjä ratkaisuja voida näin ollen tutkia [JZS12]. Viitteitä kaupallisten pelien käyttämistä pelimoottoriarkkitehtuuriratkaisuista saadaan kuitenkin pelin ammattilaisten julkaisemien artikkeleiden kautta. Nämä pelimoottoriarkkitehtuuria käsittelevät artikkelit voivat käsitellä esimerkiksi toteutettujen pelien arkkitehtuuria tai pelikehittäjien kokemuksia pelimoottorien arkkitehtuurista ja sen kehittämisestä [Wit14].

2.4.1 Määritelmä

Termi ”pelimoottori” syntyi 90-luvun puolessavälissä *Doom*-pelin mukana [Gre09 s. 11]. Kyseisen pelin arkkitehtuuri oli suunniteltu siten, että se erotti yleiset toteutusosat,

kuten grafiikan piirtämisen ja törmäyksen tunnistamisen, pelin yksilöivästä osasta, kuten äänien ja grafiikan varsinaisesta pelikohtaisesta käsittelylogiikasta. Tällainen toteutusosien erottelu mahdollisti muun muassa *Doom*-pelin lisensoinnin ulkopuolisille tahoille ja toisenlaisten pelien kehittämisen samalle pohjalle. *Doom*-pelin näyttämän esimerkin vanassa syntyi muitakin uusia itsenäisiä pelejä, kuten *Unreal*, jotka suunniteltiin alusta asti ottamaan huomioon komponenttien uudelleenkäytettävyys. Näiden pelien pelimoottorit tehtiin helposti muokattavaksi käyttämällä esimerkiksi omaa skriptauskieltä. Pelimoottori sijoittuu pelin rakenteessa eräänlaiseksi rajapinnaksi pelikohtaisen toteutuksen ja suoritusjärjestelmän palveluiden väliin [Kuva 2]. Se toimii tavallaan rajapintana pelin yleisten komponenttien ja sen yksilöivien komponenttien välillä.



Kuva 2: Pelimoottorin sijoittuminen järjestelmässä, muokattu lähteestä, muokattu lähteestä [LeJ02].

Pelin ja sen moottorin raja on usein hämärä, sillä jotkut pelit eivät pyri erottamaan niiden moottoriosaa varsinaisesta pelikohtaisesta toteutuksesta [Gre09 s. 12]. Pelimoottorin komponenttien pelikohtaisuus vaihtelee paljon eri pelien välillä. Toisessa pelissä piirtämisestä vastaava komponentti voi tietää tarkasti kuinka pelin tietty olio piirretään, kun taas toisessa pelissä sillä on käytössä vain yleiset työkalut, ja piirrettävän peliolion yksilölliset piirteet voivat tulla datana ulkopuolelta. Tällöin kyseessä olisi datavetoinen (*data-driven*) arkkitehtuuri, jossa pelin olioiden ominaisuudet määritellään ulkoapäin tulevalla datalla. Jos peli on taas toteutettu tyylillä, jossa peliin kuuluvia sääntöjä tai logiikkaa on niin sanotusti kovakoodattu sisälle lähdekoodiin, on samalle pohjalle hankala, ellei mahdoton, tuottaa toisenlaista peliä. Yleistäen *pelimoottori*-termiä tulisi käyttää ohjelmistosta, jota voidaan laajentaa ja käyttää erilaisten pelien tekemiseen.

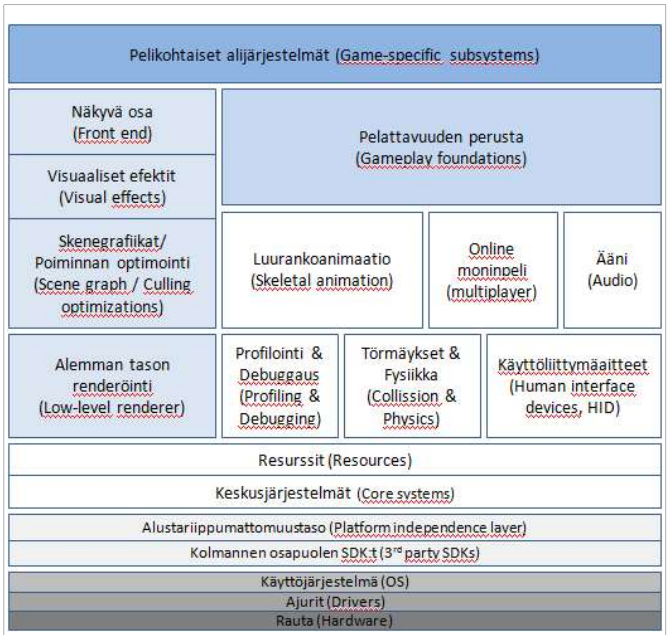
Tämä työ noudattaa pelimoottorin ideaa erottaa pelikohtainen toteutus suoritusjärjestelmän palveluista. Ehdotettu arkkitehtuurimalli mahdollistaa pelin mekaniikan ja pelin olioiden toimintalogiikan erottamisen GPS- ja sensoridatan keräämisestä ja käsittelystä.

Pelimoottori voidaan jakaa kahteen osaan, joista toinen sisältää pelinkehittämiseen liittyvät työkalut, ja toinen pelin suoritusaikaisen osan (*runtime engine*) [Gre09 s. 28].

Tässä työssä kiinnostuksen kohteena näistä on jälkimmäinen, joka sisältää pelin suorittamiseen kuuluvat osa-alueet, kuten pelisilmukan ja käyttäjäsyötteen lukemisen. Jatkossa tässä työssä termiä ”pelimoottori” käytettäessä viitataan tähän suoritusaikaiseen osaan.

2.4.2 Suoritusaikaisen osan arkkitehtuuri

Suoritusaikaisen pelimoottorin arkkitehtuuri (*runtime engine architecture*) jakautuu moneen ryhmään [Gre09 s. 28]. Jokainen näistä ryhmistä sisältää useamman komponentin, jotka hoitavat ryhmään kuuluvia tehtäviä [Kuva 3]. Näistä osa-alueista käyttöliittymälaitteet ja pelattavuuden perusta liittyvät käyttäjän syötteen keräämiseen ja käsittelyyn, joten ne ovat tämän työn kannalta kiinnostavimmat osa-alueet.



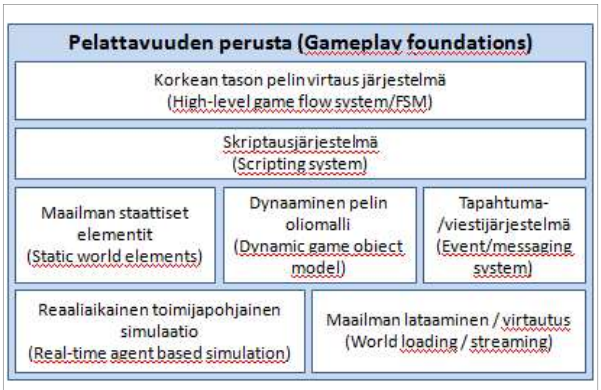
Kuva 3: Pelimoottorin komponenttiryhmät, muokattu lähteestä [Gre09 s. 28].

Pelit ovat interaktiivisia sovelluksia, joiden tulee prosessoida käyttöliittymälaitteilta (*interface devices*) tuleva käyttäjän syöte reaaliajassa [Gre09 s. 43]. Käyttöliittymälaitteita ovat kaikki sellaiset laitteet, jotka mahdollistavat pelaajan lähettää käskyjä suoritettavalle sovellukselle. Yleisiä käyttöliittymälaitteita ovat näppäimistö, hiiri, joystick ja pad-ohjain. Näitä laitteita kutsutaan usein myös pelaajan I/O-laitteiksi, koska tällaiset laitteet voivat tukea sisään tulevan syötteen lisäksi myös syötettä toiseen suuntaan, eli peliltä pelaajan suuntaan. Tällaisia käyttöliittymälaitteita ovat muun muassa tärinää tukevat (*force feedback*)-ohjaimet. Mobiililaite toimii sekä suoritusjärjestelmänä, että käyttäjäsyötelaitteena, eli niiden tapauksessa käyttöliittymälaite ei eroa peliä suorittavasta laitteesta. Näin ollen mobiililaitteita käsiteltäessä käyttöliittymälaitteen määrittelyä voidaan venyttää koskemaan kaikkia mobiililaitteiden tukemia käyttöliittymätekniikoita, kuten kosketusnäyttö, GPS ja eri sensorit.

Suoritusaikaisen osan arkkitehtuurissa käyttöliittymäkerros määrittelee miten mobiililaite käsittelee käyttöliittymälaitteita. Pelisovelluksien käyttöliittymäkerrokseen kuuluu yleensä HID-laitteista huolehtiva ohjelmistokomponentti, joka erottaa käyttäjäsyötelaitteesta lähtevän datan pelikohtaisesta ylemmän tason käyttäjäsyötteestä, ja tarjoaa tämän datan prosessoituna käyttäjäsyötteenä ylemmälle tasolle [Gre09 s. 43]. Tällainen HID-laitteista huolehtiva ohjelmistokomponentti käsittelee käyttöliittymälaitteelta tulevaa raakadataa eri tavoin, kuten poistaa nappien painamisen yhteydessä tapahtuvat tahattomat toistot tai tulkkaa ja pehmentää kiihtyvyysanturilta tulevan datan. Pelimoottorin käyttöliittymälaitteista vastaavan komponentin tehtävät voidaan mobiililaitteissa yleistää kattamaan kaikki eri mobiililaitteiden tukemat käyttöliittymätekniikat, kuten kosketusnäyttö, GPS ja eri sensorit. Tällöin se prosessoi kaikilta näiltä tekniikoilta tulevan käyttöliittymädatan ja tarjoaa sen prosessoituna ylemmän tason komponenttien käyttöön. Esimerkiksi kiihtyvyysanturin ja gyroskoopin datojen avulla voidaan pelille tarjota mobiililaitteen kallistuskulma yhtenä käyttäjäsyötteenä.

2.4.3 Pelattavuuden perusta (gameplay foundation)

Pelattavuuden perusta on yksi pelimoootoriin kuuluva osa-alue. Pelattavuuden perustaa käsiteltäessä termi ”pelattavuus” viittaa pelin virtuaalimaailmaa ohjaaviin sääntöihin, pelaajan ja muiden hahmojen tai olioiden ominaisuuksiin, pelaajan tavoitteisiin sekä pelissä tapahtuvaan toimintaan [Gre09 s. 45]. Useimpien pelimoottoreiden sisältämä pelattavuuden perustataso (*gameplay foundation layer*) tarjoaa apukeinot pelikohtaisen logiikan sisällyttämiseen pelimoottoriin. Näihin apukeinoihin lasketaan pelin toteutusarkkitehtuuriin liittyviä osia, kuten pelin olioiden tallennus ja käsittely, pelin reaaliaikaisuuden toteutus (eli pelisilmukka), tapahtumajärjestelmä ja pelimaailman hallinta [Kuva 4].



Kuva 4: Pelattavuuden perustatason arkkitehtuuri, muokattu lähteestä [Gre09 s. 45].

Pelattavuuden perustaso toimii rajapintana pelimoottorin alempien tasojen ja pelikohtaisen sisällön välillä, ja se voidaan toteuttaa joko pelin alemman tason

ohjelmointikielellä, skriptauskielellä, tai molemmilla. Tähän tasoon kuuluva tapahtumajärjestelmä on yksi mahdollisuus toteuttaa GPS- ja sensoridatan käsittely ja vaikutus pelin tilaan. Lisäksi tähän tasoon kuuluva pelisilmukan toteutus on tiukasti sidoksissa käyttäjäsyötteen käsittelyyn, ja pelisilmukoiden toimintalogiikan ja arkkitehtuurin ymmärtäminen on keskeistä tämän työn kannalta. Tässä työssä ehdotettu malli liittyy pelin interaktiivisuuteen, reaaliaikaisuuteen ja sisäiseen viestintään, joten sen suunnittelu vaatii tutustumista pelimoottorin pelattavuuden perusta -komponentista dynaamiseen pelin oliomalliin, tapahtuma/viestijärjestelmään ja reaaliaikaiseen toimijapohjaiseen simulaatioon.

2.4.4 Pelisilmukka

Pelisilmukka määrää suoritusaikaisen pelimoottoriarkkitehtuurin suuntaviivat kuten sen, tukeeko pelimoottori useaa prosessoria ja kuinka käyttäjäsyöte kerätään ja käsitellään. Tässä luvussa tarkastellaan aluksi sitä, mikä pelisilmukka on ja mitä perustehtäviä sille kuuluu. Tämän jälkeen tutustutaan deterministisyyteen pelisilmukoissa eli siihen, mitä pelisilmukalta vaaditaan, että käyttäjäsyötteeseen pohjautuvat tapahtumat olisivat toistettavissa. Näiden pohjatietojen läpikäynnin jälkeen tarkastellaan erilaisia pelisilmukkamalleja, tavoitteena löytää sopiva silmukkamalli GPS- ja sensoridatan keräämiseen ja käsittelyyn. Tässä työssä käytetään tapaa jakaa pelisilmukkamallit sidottuihin ja sitomattomiin sen mukaan, onko niiden suorittamat eri tehtävät sidoksissa toisiinsa [VCF05].

Pelisilmukan perustehtävät

Interaktiivisten reaaliaikaohjelmien sisällä pyörii eri tehtäviä toistava silmukka, jota pelisovelluksissa kutsutaan pelisilmukaksi [JZS12]. Pelisilmukan tehtävänä on lukea ja käsitellä pelaajan syöte, sekä esittää pelin tila pelaajalle reaaliaikaisesti kuvalla ja äänellä. Pelisilmukkamallit ovat lähestymistapoja organisoida pelin eri osien suoritusjärjestys. Se kertoo muun muassa sen, tukeeko pelimoottori useaa prosessoria, missä järjestyksessä ja kuinka usein pelin vaatimat toimet suoritetaan ja kuinka käyttäjäsyöte kerätään ja käsitellään. Pelisilmukoilla on yleisesti kolme perusosaa, jotka ovat syötteen lukeminen, pelin tilan päivitys ja pelin tilan esittäminen pelaajalle [VCF05, JZS12]. Syötteen lukemiseen kuuluu käyttäjäsyötteen kerääminen syötevälineiltä, kuten näppäimistöltä tai kosketusnäytöltä. Tämä voidaan ajatella pelin

syötteenä (*game input*), jota kutsutaan joskus syötelaitekyselyksi (*input device query*). Pelin tilan päivityksessä prosessoidaan keräty käyttäjäsyötteen vaikutus, eli lasketaan pelin sääntöjen ja käyttäjäsyötteen vaikutus pelin tilaan sekä mallinnetaan fysiikka ja suoritetaan tekoäly. Pelin päivitysvaiheesta voidaan puhua päivitystehtävinä (*update tasks*) tai päivitysvaiheena (*update stage*). Yleensä peli esittää tilansa pelaajalle kuvan ja äänen avulla, ja tätä tehtävää voidaan kutsua visualisoinniksi ja auralisaatioksi, eli kuva- ja äänitilan luonniksi (*game visualization and auralization*), tai esitysvaiheeksi (*presentation stage*).

Pelisilmukan arkkitehtuuri määrää muun muassa missä järjestyksessä nämä pelin eri tehtävät suoritetaan, kuten missä vaiheessa suoritusta käyttäjän syöte luetaan ja käsitellään. Alla on esimerkki pseudokoodilla yksinkertaisesta pelisilmukasta:

```
while(peli on käynnissä)
{
        lue_syöte();
        toteuta_pelin_logiikka();
        liikuta_pelin_hahmoja();
        piirrä_grafiikka_soita_äänet();
}
```

Reaaliaikaan pohjautuvissa järjestelmissä, eli peleissäkin, täytyy edellä kuvatut pelisilmukan kolme perusosaa pystyä suorittamaan reaaliajassa tai sovelluksen ja käyttäjän välinen interaktiivisuus kärsii. Yleinen tapa mitata pelin reaaliaikaisuutta on näytölle piirrettyjen ruutujen määrä sekunnissa (*frames per second*, FPS). Alarajana interaktiivisuuden tuntemukselle pidetään 16 ruudun piirtämistä näytölle sekunnissa ja optimaalisena 50-60 ruutua sekunnissa [JZC08]. Tämä tapa ei tosin kerro koko totuutta sellaisissa monisäikeisissä pelisilmukoissa, joissa piirtäminen tapahtuu erillisessä säikeessä. Niissä totuudenmukaisempaa on FPS-arvon lisäksi tarkkailla kuinka monta päivitystä sekunnissa tapahtuu (*updates per second*, UPS).

Edellä esiteltyjen pelisilmukan perusosien suoritusjärjestys kuvaa pelin kokonaisarkkitehtuuria ja kertoo, minkä tyyppisiin peleihin ja ympäristöihin kyseinen pelimoottori sopii [VCF05]. Jotkin pelisilmukkamallit tukevat esimerkiksi montaa suoritinta, kun taas toisenlaiset pelisilmukkamallit on suunniteltu ajettavaksi yksisäikeisenä, tai jotkin pelisilmukat voivat maksimoida grafiikan piirtämisen muiden tehtävien edelle.

Pelisilmukkamallit voidaan jakaa kahteen ryhmään sen mukaan, onko pelisilmukan

kolmas vaihe, eli piirtovaihe, sidoksissa kahden ensimmäisen pelisilmukan osan suorituksesta [VCF05]. Pelisilmukkamallia voidaan siis kutsua joko sidotuksi (*coupled*) tai sitomattomaksi (*uncoupled*) sen mukaan, ovatko nämä pelisilmukan suorittamat tehtävät sidoksissa keskenään.

Deterministisyys pelisilmukkamalleissa

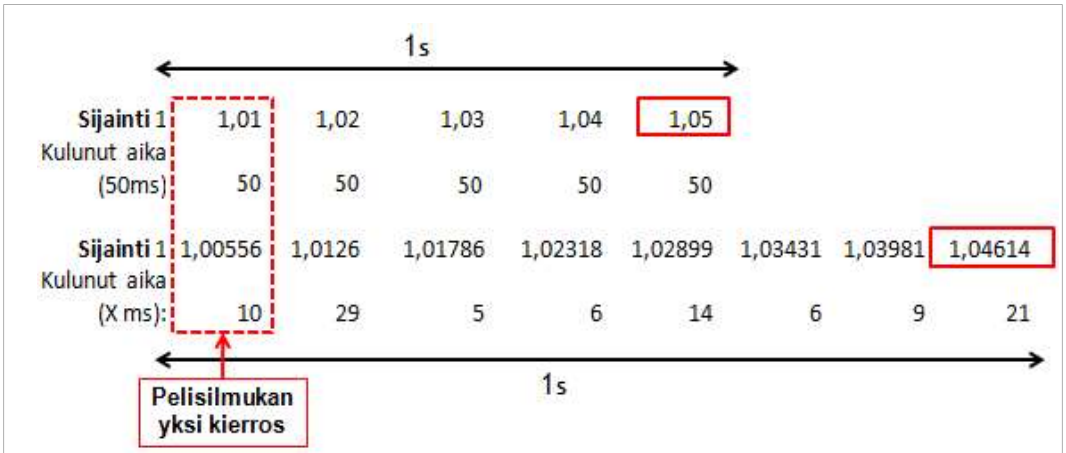
Deterministisessä pelisilmukassa peli saavuttaa aina saman lopputilan tietystä alkutilasta, kun käytetään samaa käyttäjäsyötettä. Jos pelin päivitystehtävien suoritustiheys vaihtelee tietyn aikajakson sisällä, vaikuttaa tämä vaihtelu pelin tilaan aikajakson lopussa [Gre09 s. 316]. Tällaisen käytöksen sallivaa pelisilmukkamallia kutsutaan ei-deterministiseksi. Ei-deterministisessä pelisilmukkamallissa lopputila eroaa eri suorituskertojen välillä.

Deterministisyyden noudattamisesta voi olla kahdenlaista hyötyä [Gre09 s. 316]. Ensinnäkin se helpottaa bugien toistamista, koska tietty syöte antaa aina saman lopputuloksen. Toiseksi se mahdollistaa niin sanotun toistotoiminnon (*replay*) käyttämisen peleissä, eli pelitapahtumat pystyy toistamaan ja katselemaan uudelleen itse tapahtuman jälkeen. Lisäksi verkkopelien toteutus voi olla helpompaa deterministisellä pelisilmukkamallilla, jossa toisen pelaajan tilan selvittämiseen riittää kyseisen pelaajan käyttäjäsyötteen lähettäminen.

Pelisilmukoiden toteutuksessa käytetään yleisesti muuttujaa kuluneelle ajalle (*elapsed time*). Tämä muuttuja viittaa aikaan, joka on kulunut edellisestä pelisilmukan suoritussyklistä, ja sen avulla päivitystoiminnot suhteutetaan kuluneeseen aikaan, ja pelitapahtumat tapahtuvat pelaajan kannalta suhteutetulla nopeudella. Päivitystoimintojen suhteuttaminen vaatii kuitenkin arvojen pyöristämistä [VCF05]. Näin ollen, kun pelisilmukka pyörii vaihtelevalla kierrosnopeudella, aiheuttaa saman käyttäjäsyötteen läpiajo eri kerroilla todennäköisesti erilaisen lopputilan.

Kun ajatellaan esimerkiksi tilannetta, jossa ei-deterministinen pelisilmukkamalli käsittelee täsmälleen saman syötteen täsmälleen samassa aikaikkunassa kaksi eri kertaa. Ei-deterministinen pelisilmukkamalli suorittaa pelin tilaan liittyviä päivitystoimia niin nopealla kierrosnopeudella, kuin järjestelmä suinkin mahdollistaa, eli pelisilmukka pyörähtää kummallakin kerralla luultavasti eri kertamäärän verran. Pelisilmukan toimet voisivat esimerkiksi tapahtua ensimmäisellä kerralla yhden sekunnin aikana viisi kertaa,

ja toisella kerralla kahdeksan kertaa [Kuva 5]. Tällöin data-arvojen pyöristämisestä tulevat erot voivat johtaa erilaiseen lopputulokseen eri suorituskertojen välillä. Jos pelitilan päivityksen suoritustiheyttä ei ole vakiinnutettu, ei välttämättä saavuteta täsmälleen samanlaista lopputilaa, koska pelisilmukan kierrosnopeus voi vaihdella esimerkiksi järjestelmän vaihtelevan kuormitustilanteen takia.



Kuva 5: Esimerkki deterministisyydestä; pelin olion sijainti yhden sekunnin kuluttua

3 Pelisilmukan arkkitehtuuri

Pelisilmukka muodostaa tärkeän osan pelin arkkitehtuurista, ja tässä luvussa perehdytään tarkemmin puoliin siihen, mitä pitää ottaa huomioon suunniteltaessa arkkitehtuuria mobiililaitteiden pelisilmukalle. Aluksi tässä luvussa tarkastellaan tapoja jakaa pelisilmukan tehtäviä rinnakkaisiin säikeisiin, jotta moniydinprosessorista voidaan kuormittaa useita ytimiä. Tämän jälkeen perehdytään erilaisiin yleisiin pelisilmukkamalleihin. Käytetty pelisilmukkamallien tarkastelu pohjautuu tapaan jakaa pelisilmukat sidottuihin ja sitomattomiin malleihin sen mukaan onko pelisilmukan piirtämisosa riippuvainen muista pelisilmukan osista [VCF05]. Yleiset pelisilmukkamallit eivät kuitenkaan sovellu sellaisenaan mobiililaitteille, sillä mobiililaitteiden ominaisuudet asettavat pelisilmukoille omat vaatimuksensa.

3.1 Pelisilmukan päivitystehtävien rinnakkaistaminen

Nykyiset mobiililaitteet sisältävät moniydinprosessorin. Tällaisten moniprosessoritietokoneiden tehokkuus saadaan hyödynnettyä peleissä paremmin, jos

monisäikeisyys huomioidaan jo pelisilmukassa eikä vain yksittäisten komponenttien tasolla [Mön06]. Monisäikeisyyden hyödyntäminen voidaan jakaa kahteen tyyliryhmään sen mukaan, rinnastetaanko tehtävät funktioiden vai datan samankaltaisuuden mukaan. Säikeisiin jako tehdään joko jakamalla itsenäiset tehtävät omiin säikeisiinsä tai samaa dataa käyttävät tehtävät omiin säikeisiinsä.

Pelin tehtävien jakaminen useaan säikeeseen tuo mukanaan tehokkuuden lisäksi myös ongelmia. Monet pelin tehtävistä ovat sidoksissa toisiinsa, ja tämä rajoittaa niiden irrottamista itsenäisiksi säikeiksi [LaG05]. Kuvaruutujen (*frame*) prosessoinnin sisäiset, sekä niiden väliset, sidonnaisuudet rajoittavat mahdollisuuksia rinnakkaiseen prosessointiin. Alla joitakin esimerkkejä peliosien sidonnaisuudesta:

- Jotkin pelihahmot eivät voi liikkua ennen, kuin AI-alijärjestelmä kertoo niille mihin liikkua.

- Ääniä ei voida valmistella, ennen kuin kuvaan liittyvät toiminnot ja tapahtumat on päätetty.

- Törmäyksentunnistus on jatkuva prosessi, joka voi keskeyttää muut laskennat.

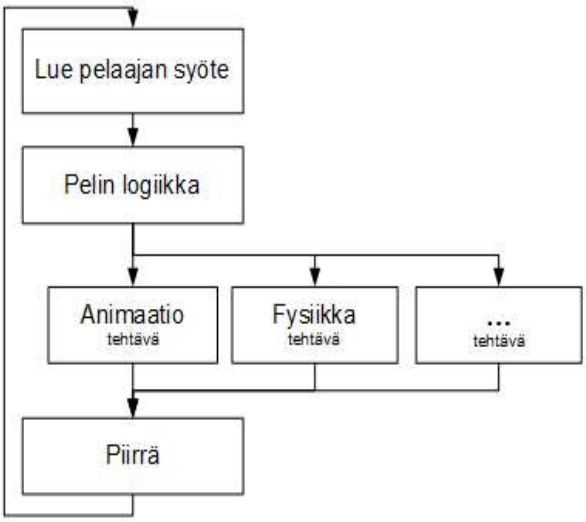
- Grafiikan piirtäminen ei voi tapahtua, ennen kuin kuvan lopullinen tila on päätetty.

3.1.1 Tehtäväpohjainen rinnakkaisuus

Synkroninen funktiorinnakkaisuus

Synkronisessa funktiorinnakkaisessa mallissa (*Synchronous function parallel model*) etsitään sellaiset tehtävät, jotka voidaan suorittaa rinnakkain [Mön06]. Malli mukailee haarauta/yhdistä (*fork/join*) -lähestymistapaa, mutta työtehtävät ovat itsenäisiä, eikä niiden tuloksia yhdistetä [Gre09 s. 328]. Haarautettavien tehtävien tulee siis olla itsenäisiä ja riippumattomia muiden rinnakkaisten tehtävien tuloksista. Jokainen itsenäinen tehtävä suoritetaan omassa säikeessään ensimmäisessä vapautuvassa prosessoriytimessä. Pelin fysiikkaan liittyvät tehtävät voidaan esimerkiksi suorittaa samanaikaisesti animaation laskemisen kanssa [Kuva 6].

Tämän mallin huonona puolena on skaalautuvuuden rajoittuneisuus, sillä itsenäisiä tehtäviä voidaan löytää vain rajoitetusti. Näin ollen törmätään ylärajaan sille, kuinka montaa suoritinta malli voi hyödyntää. Eli jos löydetään *n* itsenäiseksi erotettavaa tehtävää, voi malli hyödyntää *n* prosessoriydintä.

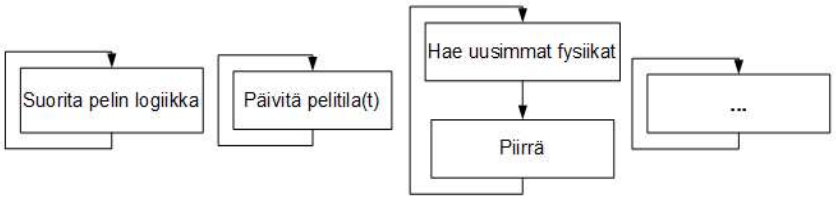


Kuva 6: Synkroninen funktiorinnakkainen malli, muokattu lähteestä [Mön06].

Asynkroninen funktiorinnakkaisuus

Toisessa tehtävät toisistaan erottavassa mallissa pelisilmukan vaatimat toimet suoritetaan ilman varsinaista pääpelisilmukkaa [Mön06]. Niin sanotussa asynkronisessa funktiorinnakkaisessa mallissa jokainen tehtävä erotetaan omaan säikeeseensä [Kuva 7]. Jokainen tehtävä päivittää itseään itsenäisesti omassa tahdissaan käyttäen tuoreinta saatavissa olevaa dataa. Datan jakaminen kaikille komponenteille voidaan toteuttaa eri tavoin, esimerkiksi käyttämällä luvussa 4.3 esiteltyä liitutaulumallin mukaista lähestymistapaa, jossa etukäteen määritelty datavarasto pitää sisällään uusimman kaikille yhteisen datan.

Asynkronisessa funktiorinnakkaisessa mallissa esimerkiksi piirtotehtävä ei odota meneillään olevan

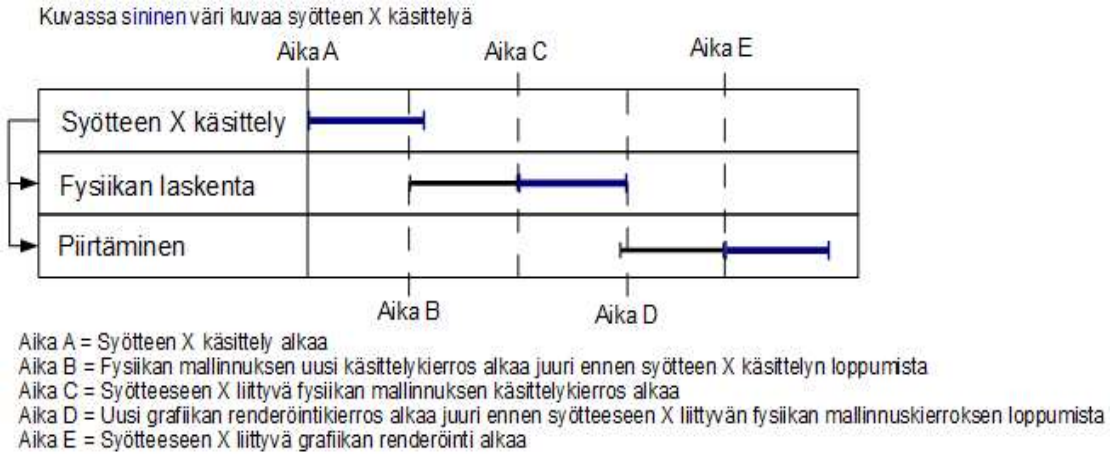


Kuva 7: Asynkroninen funktiorinnakkainen malli, muokattu lähteestä [Mön06].

fysiikanmallinnuskierroksen loppumista, vaan piirtää grafiikan edellisen mallinnuksen pohjalta, käyttäen tuoreinta saatavilla olevaa dataa. Tällä tyylillä voidaan erottaa toisistaan riippuvat tehtävät omiin säikeisiinsä. Kuten synkroninen funktiorinnakkainen malli, myös tämä asynkroninen funktiorinnakkainen malli rajoittuu skaalautuvuudessa sen mukaan, kuinka monta erillistä tehtävää pelimoottorista löytyy. Toisaalta käyttämällä uusinta saatavilla olevaa dataa voidaan erillisiin säikeisiin erottaa

sellaisiakin tehtäviä, jotka eivät ole täysin itsenäisiä, eli toisistaan riippuvat tehtävät voivat toimia omissa säikeissä.

Edellä kuvattu asynkroninen funktiorinnakkainen malli tuo kuitenkin ajoitukseen liittyvän ongelman, joka on hyvä ottaa huomioon suunnitteluvaiheessa [Mön06]. Ongelmaa voidaan käsitellä esimerkin kautta [Kuva 8]. Esimerkissä syötteen käsittely-, fysiikan laskenta- ja piirtämistehtävä käsittelevät kaikki samaa dataa. Kun käyttäjä välittää pelille komennon, se käsitellään ensimmäiseksi syötetehtävän säikeessä. Mutta jos fysiikan laskentatehtävä on juuri aloittanut uuden käsittelykierroksen juuri ennen käyttäjäsyötetehtävän valmistumista, pääsee tämä uusi käyttäjäsyöte vasta seuraavalle fysiikanmallinnuksen kierrokselle. Samoin, jos piirtämistehtävä on aloittanut uuden kierroksen juuri ennen fysiikanmallinnuksen valmistumista, pääsee käyttäjäsyötteestä seurannut fysiikanmallinnusdata vasta seuraavalle piirtämiskierrokselle. Tällaisessa niin sanotussa pahimman tapauksen skenaariossa käyttäjäsyötteen pääsy piirtämiseen kestää noin kaksi kertaa optimaalista skenaariota kauemmin, jossa jokaisen tehtävän käsittelykierros alkaa heti kun aiempi tehtävä on valmistunut. Käsittelyaika jokaiselle käyttäjäsyötteelle piirtämiseen kestää yleistäen siis jotakin optimaalisen ja pahimman skenaarion väliltä.



Kuva 8: Esimerkki asynkronisessa funktiorinnakkaisessa mallissa esiintyvästä ajoitusongelmasta, muokattu lähteestä [Mön06].

Ajoitusongelmaa voidaan pienentää asettamalla tehtäville vakioidut toistonopeudet ja synkronoimalla toistonopeudet sopiviksi toisiin tehtäviin nähden [Mön06]. Piirtämistehtävän kierrosnopeus voidaan esimerkiksi säätää kaksinkertaiseksi syötteen lukemistehtävään nähden, jolloin piirtämistehtävän aloitusta joudutaan odottamaan

vähemmän aikaa. Tämä vähentää kyseistä ongelmaa, muttei poista sitä kokonaan. Koska asynkronisessa funktiorinnakkaisessa mallissa tehtävät eivät kommunikoi suoraan keskenään, vaan jaetun datavaraston kautta, täytyy pelimoottorin komponenttien voida pyytää tarvitsemansa datan uusin versio käyttöönsä aina tarvittaessa. Eli tarvitun datan täytyy aina olla pelin komponenttien käytettävissä.

3.1.2 Datan käsittelyyn pohjautuva rinnakkaisuus

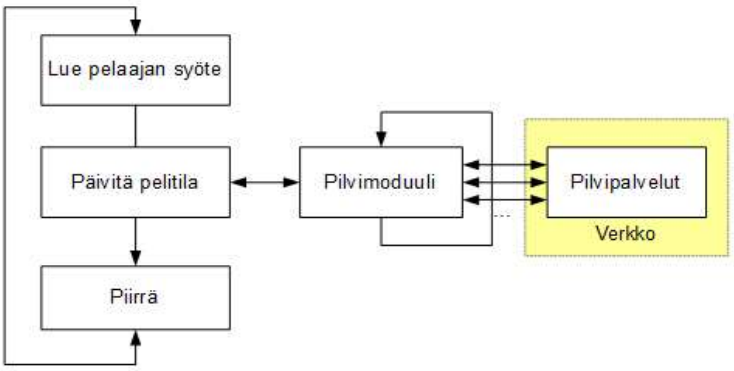
Pelisilmukan päivitystehtävien jakaminen useampaan säikeeseen voi pohjautua myös datan rinnakkaisuuteen [Mön06]. Tällainen pelisilmukkamalli pyrkii löytämään yhteenkuuluvia dataryhmiä, joiden tehtävien suoritus voitaisiin tehdä yhtä aikaa rinnakkain. Kun käsitellään pelejä, voi tällainen dataryhmä olla esimerkiksi jokin pelin luokka, ja siitä luodut oliot. Esimerkiksi lentosimulaattoripelissä lentokoneoliot voitaisiin jakaa kahteen osaan, jotka käsitellään omissa säikeissään rinnakkain.

Datan rinnakkaiseen käsittelyyn pohjautuvassa mallissa ratkaistavia asioita on säikeiden tasapainotus maksimaallisen suoritintehon saavuttamiseksi, ja se, kuinka olio toisesta säikeestä kommunikoi toisen säikeen olion kanssa. Tällainen pelisilmukkamalli ei välttämättä ole vielä nykypäivänä tarpeellinen, koska edellä esitellyt tehtäväpohjaiseen rinnakkaistamiseen liittyvät mallit riittävät erottamaan tarpeeksi erillisiä säikeitä tämän hetken moniydinprosessoreille. Tilanne muuttuu, kun tehtäväpohjaisten rinnakkaisuusmallien skaalautuvuuden yläraja saavutetaan (tietokoneiden suorittimien määrä nousee suuremmaksi kuin pelistä erotettavat itsenäisten tehtävien määrä) voi tämän mallin tarpeellisuus nousta.

Pilvipalveluita hyödyntävä malli

Esimerkki datan rinnakkaisen prosessoinnin hyödyntämisestä on mobiililaitteille kehitetty pilvipalveluita hyödyntävä pelisilmukkamalli [VCF05]. Tämä malli kasvattaa pelisilmukan perustehtäviä lisäämällä siihen omassa säikeessä suoritettavan komponentin, joka hoitaa yhteyden pilvipalveluun [Kuva 9]. Laajennusosa huolehtii pelin ja pilvipalveluiden välisestä kommunikoinnista. Pilvimoduuli taas luo jokaiselle pilvipalvelupyynnölle oman itsenäisen säikeensä, eli se hyödyntää datan rinnakkaista prosessointia. Tällaisen mallin avulla mobiilipeli pystyy käyttämään pilviverkon avulla sellaisia tehtäviä ja palveluita joihin mobiililaitteen muisti tai prosessointiteho ei riitä. Samaa periaatetta noudattaen myös niin sanottuja perinteisiä pelisilmukkamalleja

voidaan laajentaa haluttuun suuntaan lisäämällä haluttu omassa säikeessään pyörivä tehtävä muiden tehtävien rinnalle.

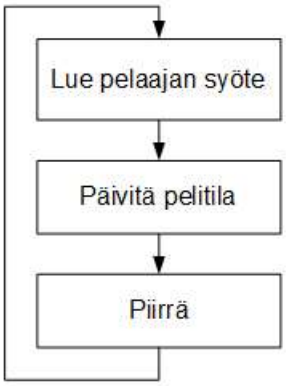


Kuva 9: Pilvipalveluita käyttävä pelisilmukkamalli, muokattu lähteestä [VCF05].

3.2 Yleiset pelisilmukkamallit

3.2.1 Sidottu pelisilmukkamalli

Yksinkertaisin pelisilmukkamalli on niin kutsuttu yksinkertainen sidottu malli (*Simple coupled model*) [VCF05]. Tässä mallissa pelisilmukkaan kuuluvat tehtävät suoritetaan perätysten yhdessä silmukassa [Kuva 10]. Tätä mallia on käytetty paljon sen yksinkertaisuuden vuoksi, ja se voikin toimia suljetuissa järjestelmissä, kuten pelikonsolit, joissa suoritusympäristöt ovat homogeenisiä. Tämä malli ei kuitenkaan ole hyvä valinta peleihin, joita pelataan laitteistoltaan heterogeenisissä järjestelmissä, koska mallin suoritusnopeus on suoraan sidoksissa suorittavan järjestelmän tehoihin. Tässä mallissa pelin osia ajetaan niin nopeasti kuin suoritinteho sen sallii, eli pelisilmukan suoritusnopeus vaihtelee järjestelmän tehojen mukaan. Pelisilmukan toteutusnopeuteen vaikuttavat tällöin muun muassa suorittavan tietokoneen tehokkuus ja kyseisen hetken kuormitustilanne. Alla olevassa pseudokoodissa pelisilmukka siirtää pelihahmoa 4 yksikköä jokaisella kierroksella:



Kuva 10: Yksinkertainen sidottu malli, muokattu lähteestä [VCF05].

```
while(true) {
        pelihahmo.paikkaX = pelihahmo.paikkaX + 4;
        pelihahmo.Piirrä();
}
```

Pelihahmo liikkumisnopeus on siis suoraan verrannollinen silmukan toistotiheyteen. Pelihahmo liikkuu siis eri nopeutta suoritettaessa pelisilmukkaa esimerkiksi 30 tai 60 kertaa sekunnissa. Useat 1980- ja 90-luvun tietokonepelit kärsivät tämänkaltaisen

pelisilmukkamallin ongelmista ja olivat mahdottomia pelata kehittyneemmillä ja nopeammilla prosessoreilla pelin pyöriessä kuin pikakelauksella. Vastakohtana heterogeenisille tietokoneympäristöille on homogeeniset järjestelmät, kuten pelikonsolit, joissa järjestelmien suoritusteho on samanlainen eri laitteistojen välillä, ja tämä pelisilmukkamalli voikin toimia hyvin sellaisissa laitteistoissa.

Yksinkertaisimmasta pelisilmukkamallista seuraava aste on pakottaa silmukalle jokin sopivan alhainen tietty kierrosnopeus ja suorittaa pelisilmukkaa tällä vakionopeudella [VCF05]. Tällöin puhutaan synkronoidusta sidotusta mallista (*syncronized coupled model*). Tällöin pelisilmukka pakotetaan suorittamaan itseään esimerkiksi 30 kertaa sekunnissa (30 Hz), jolloin pelisilmukkaa toistetaan tuolla nopeudella riippumatta suoritusjärjestelmän tehoista. Tämän mallin huonona puolena on sen turha rajoittuneisuus, sillä tehokkaampien koneiden parempaa suorituskykyä ei hyödynnetä lainkaan, kun pelin kaikki toimet pakotetaan tähän tiettyyn nopeuteen. Toinen huono puoli ilmenee järjestelmissä, jotka eivät pysty pyörittämään pelisilmukkaa tavoitenopeudessa, jolloin pelimaailman tapahtumat tapahtuvat kuin hidastetussa filmissä. Kuten edellinen yksinkertainen sidottu malli, myös tämä synkronoitu sidottu malli voi toimia hyvin homogeenisissä järjestelmissä, joissa vakioitu kierrosnopeus voidaan optimoida kyseiselle järjestelmälle sopivaksi.

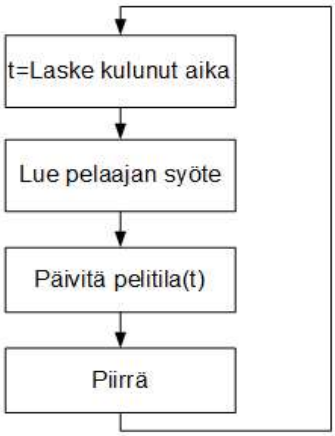
3.2.2 Sitomattomat pelisilmukkamallit

Kun edellisessä luvussa tarkasteltiin pelisilmukkamalleja, joissa pelisilmukan eri vaiheet olivat sidoksissa toisiinsa, tässä luvussa tarkastellaan pelisilmukkamalleja, joissa pelisilmukan piirto osan riippuvuus toisiin osiin on rikottu. Yleisin tapa rikkoa tämä riippuvuus on erottaa pelin piirtovaihe pelisilmukan muista vaiheista, jolloin piirtovaihe voi pyöriä niin nopeasti kuin pystyy, ja muut toimet tapahtuvat siitä riippumattomalla nopeudella [VCF05].

Yksisäikeinen sitomaton malli

Yksisäikeisestä pelisilmukasta saadaan teoriassa sitomaton, kun päivitystehtävä ja piirtämistehtävä käyttävät prosessorin suoritustehoja hyväkseen itsenäisesti, riippumatta toisistaan [JSZ12]. Yksisäikeisessä sitomattomassa pelisilmukkamallissa kaikki pelisilmukan tehtävät ovat samassa silmukkarakenteessa, mutta päivitystehtävät ottavat huomioon kuluneen reaaliajan [Kuva 11]. Aikamuuttuja kertoo

pelinpäivitysoperaatioille kuinka kauan edellisestä päivityssilmukan suorituksesta on kulunut. Päivitystehtävä voi sitten aikamuuttujan avulla suhteuttaa päivitysoperaatioiden suuruudet. Mitä enemmän aikaa on kulunut, sitä suurempi vaikutus päivityksellä tulisi olla pelin tilaan.

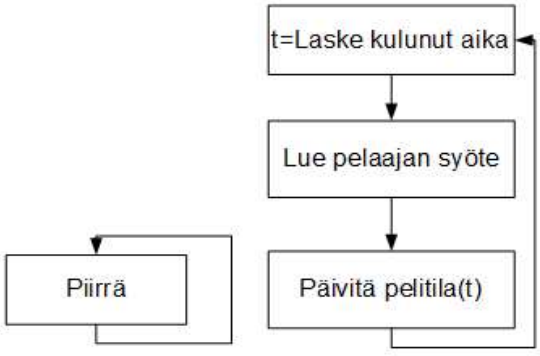


Kuva 11: Yksisäikeinen sitomaton malli, muokattu lähteestä [VCF05].

Pelisilmukan päivitysosa huolehtii useista tehtävistä, kuten fysiikan mallintamisesta sekä tekoälyn ja pelilogiikan suorittamisesta. Nämä kaikki osa-alueet eivät kuitenkaan välttämättä hyödy suorittamisesta jokaisella pelisilmukan suoritussyklillä, vaan niiden jatkuva suoritus syö turhaan suoritintehoa. Toisaalta jotkin pelin päivittämiseen liittyvät tehtävät, kuten tekoäly, saattavat hyötyä mahdollisimman tiheästä päivityksestä.

Monisäikeinen sitomaton malli

Mobiililaitteen moniydinprosessorin täysimääräinen hyödyntäminen vaatii vähintään hyödynnettävien prosessoriytimien määrän suoritussäikeitä. Yksinkertaisin tapa luoda monisäikeinen pelisilmukka on erottaa piirtämisestä vastaava osa omaan silmukkaansa [VCF05]. Tämä olisi kuitenkin naiivi ratkaisu, sillä tällaisessa mallissa törmättäisiin samoihin ongelmiin kuin edellä esitellyn yksinkertaisen sidotun mallin kanssa, jossa suoritusjärjestelmän tehokkuus määräisi pelitapahtumien etenemisnopeuden. Tämä ongelma voidaan kuitenkin korjata poistamalla linkitys pelin tilan laskemisen ja pelisilmukan suoritusnopeuden väliltä [Kuva 12]. Tämä onnistuu käyttämällä aikamuuttujaa, kuten yksisäikeisessä pelisilmukkamallissa. Monisäikeiseksi sitomattomaksi malliksi (*multi-thread uncoupled model*) kutsuttua pelisilmukkamallia käyttämällä tehtäväsilmukoita suoritetaan koneen tehojen mukaisesti. Tehokkaampi kone suorittaa tehtäväsilmukoita useammin, tuottaen sujuvamman ruudunpäivityksen ja antaen enemmän aikaa pelin logiikan

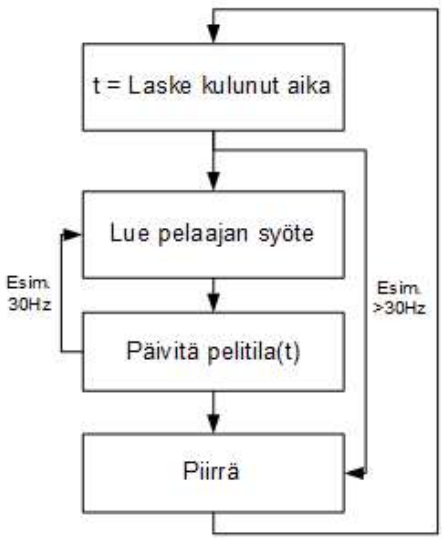


Kuva 12: Monisäikeinen sitomaton malli, muokattu lähteestä [VCF05].

laskemisille, kuin tehottomampi kone mutta silti suorittaen peliä reaaliaikaan suhteutettuna. Tällainen malli törmää kuitenkin tyypillisiin monisäikeisten ohjelmien ongelmiin, kuten datan pääsyoikeuksien hallintaan ja synkronointiin, jotka täytyy ottaa huomioon kehitysvaiheessa. Jos esimerkiksi vaihe piirtää kuvaa johon liittyvä tieto on vasta osittain päivitetty, ei tulokseksi tule konsistenssi kuva.

Yksisäikeinen vakiotaajuudellinen sitomaton malli

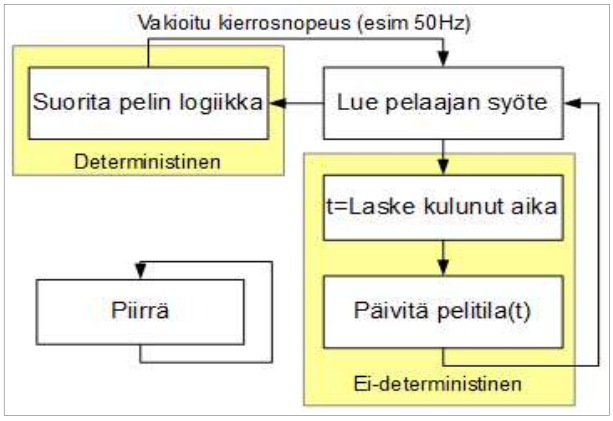
Yksisäikeisessä vakiotaajuudellisessa sitomattomassa mallissa [Dic01] yhdistetään synkronoitu sidottu malli ja yksisäikeinen sitomaton malli. Synkronoitua sidottua mallia parantamalla asetetaan vain pelin päivitystoimet toteutumaan vakioidulla taajuudella. Vakiotaajuudellinen yksisäikeinen sitomaton pelisilmukkamalli täyttää determinismin vaatimukset. Malli saavuttaa tämän suorittamalla syötteen luvun ja tilan päivitystoimet vakioidulla taajuudella, esimerkiksi 10 ms välein (100 Hz) [Kuva 13]. Jos edellisestä kerrasta ei ole kulunut tätä aikaa, hyppää silmukka syötteen lukemisen ja päivityksen yli suoraan piirtämiseen. Jos päivitystoimet ovat taas jääneet piirtämisestä jälkeen, toistetaan syötteen luku- ja päivitystoimet tarpeeksi monta kertaa, jotta ne saavuttavat piirtämisen vaiheen. Tehtävän vakiotaajuudellinen suoritus voidaan toteuttaa käyttämällä apumuuttujaa ajalle, jonka avulla pidetään kirjaa siitä, kauanko tehtävän edellisestä suorittamisesta on kulunut aikaa ja jos edellisestä suorituksesta ei ole kulunut riittävästi aikaa, hypätään kyseisen tehtävän yli suorittamaan seuraavaa tehtävää [Liite 4].



Kuva 13: Vakiotaajuudellinen yksisäikeinen sitomaton pelisilmukkamalli, muokattu lähteestä [Dic01].

Nopeissa järjestelmissä tämä malli toimii hyvin ja vakiinnuttaa syötteen luku- ja päivitystiheyden. Hitaissa järjestelmissä, joissa pelisilmukka ei pysy suunnitellussa miniminopeudessa, eli esimerkkitapauksessa jokainen pelisilmukka kestää yli 10ms, pyörii peli hidastettuna ja kärsii nykimisestä. Toisaalta minimikierrosnopeus voidaan suunnitella siten, että on epätodennäköistä, että peliä tullaan suorittamaan liian hitaassa järjestelmässä.

Edellä kuvattu malli toimii käytännössä useissa tilanteissa, ja sitä käytetään yleisesti,

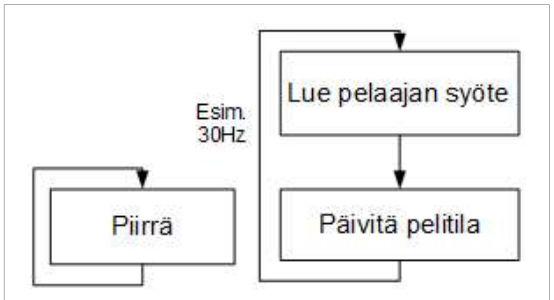


Kuva 14: Variaatio vakiotaajuudellisesta monisäikeisestä mallista, muokattu lähteestä [VCF05].

kuten avoimen lähdekoodin Cocos2D pelimoottorissa [JZS12]. Kyseessä on kuitenkin yksisäikeinen silmukkamalli, joka ei hyödynnä useampaa prosessoria. Lisäksi jotkin pelisilmukan vaiheet, kuten pelin tekoälyn ja logiikan käsittely, eivät välttämättä hyödy mitään siitä, että niitä ajetaan mahdollisimman usein.

Vakiotaajuudellinen monisäikeinen sitomaton malli

Monisäikeisestä sitomattomasta pelisilmukkamallista saadaan deterministisyyden täyttämä vakioimalla päivitystehtävien suoritusnopeus [Kuva 14]. Tämä pelisilmukkamalli on muuten sama, kuin edellä esitetty monisäikeinen sitomaton pelisilmukkamalli, mutta syötteen lukemistehtävän ja pelintilan päivitysvaiheen säikeeseen on lisätty aikamuuttuja, jonka avulla kyseistä säiettä suoritetaan halutulla vakioidulla nopeudella. Tällaista sitomattoman pelisilmukkamallin monisäikeistä versiota voidaan kutsua vakiotaajuudelliseksi sitomattomaksi malliksi. Tästä pelisilmukkamallista on johdettu kehittyneempi versio, jossa pelin päivitystehtävät erotetaan kahteen haaraan [VCF05, kuva 15]. Toista näistä päivitystehtävien haaroista toteutetaan niin nopeasti kuin mahdollista käyttäen aikamuuttujaa sitomaan tehtävät reaaliaikaan. Toiseen päivityshaaran taas sijoitetaan sellaiset toimet jotka tulisi suorittaa vakiotaajuudella determinismin saavuttamiseksi.



Kuva 15: Vakiotaajuudellinen monisäikeinen sitomaton pelisilmukkamalli.

3.3 Mobiililaitteiden vaatimukset pelisilmukalle

Tekninen kehitys tuo mobiililaitteisiin jatkuvasti muun muassa yhä nopeampia prosessoreita ja tarkempia näyttöjä. Kääntöpuolena tälle kehittyvälle tekniikalle on kuitenkin suurempi virrankulutus, johon mobiililaitteiden akut evät pysty vastaamaan. Tämän vuoksi mobiililaitteiden akut ovat suurin hidaste mobiililaitteiden teknisessä kehittymisessä [CBC10]. Toisin kuin verkkovirtaan kytketyissä tietokoneissa, kuten pöytätietokoneissa ja pelikonsoleissa, täytyy ohjelmistokehittäjän ottaa huomioon laitteen virrankulutus sekä ymmärtää suorituskyvyn ja akunkeston välinen yhteys. Verkkovirtaan kytketty reaaliaikainen järjestelmä pyrkii tyypillisesti maksimoimaan suoritusnopeuden käyttämällä prosessoreita, ja muita resursseja, mahdollisimman tehokkaasti. Sama tyyli ei kuitenkaan toimi mobiilisovelluksissa, joiden tulisi toimia energiatehokkaasti.

Mobiililaitteen kuluttaman virran määrä riippuu monista tekijöistä, kuten näytön koosta ja kirkkaustasosta, prosessoreiden kuormituksesta ja aktiivisista sensoreista. Radiotaajuuksia käyttävät tekniikat, kuten Bluetooth, GPS ja GPRS, aiheuttavat suurimman osan toteuttamansa sovelluksen virrankulutuksesta [MLF08, CBC10, APH09]. N95 älypuhelimella tehdyistä virrankulutusmittauksista selviää muun muassa se, että GPS ja GSM-radio ovat selvästi eniten virtaa kuluttavat osat [Kuva 16, APH09]. Ne kuluttavat esimerkiksi moninkertaisen määrän virtaa kiihtyvyysanturiin nähden.

Sensori	Akun kesto (noin tuntia)	Sähkönkulutus (mW)
Videokamera	3,5h	1258
IEEE 802.11 (WLAN)	6,7h	661
GPS (ulkona)	7,1h	623
GPS (sisällä)	11,6h	383
Mikrofoni	13,6h	329
Bluetooth	21,0h	211
Kiihtyvyysanturi	45,9h	96
Kaikki sensorit pois	170,6h	26

Kuva 16: Eri ominaisuuksien virrankulutus, muokattu lähteestä [APH09].

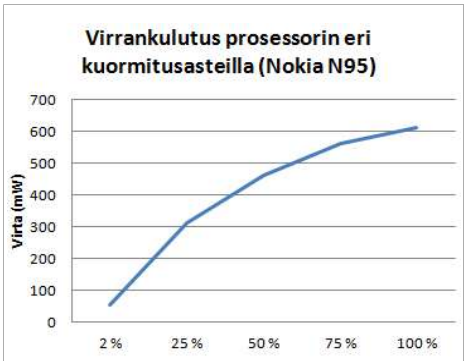
Sensoreiden lukeminen ja niiltä saatavan datan prosessointi voi olla merkittävä osa sovelluksen virrankulutusta [MLF08]. Ohjelmiston arkkitehtuuria mietittäessä tulisi muistaa, ettei sensoreita tulisi käyttää siten, että ne aiheuttavat hankaluuksia laitteen muuhun käyttöön, kuten akun yllättävän nopean loppumisen [PFW11]. Jos sensoreiden avulla haluttu toiminnallisuus ei välttämättä vaadi niiden jatkuvaa päällä oloa, voidaan virtaa säästää käyttämällä niitä jaksoissa (*duty cycles*), jolloin haluttu toiminto saavutetaan aktivoimalla sensori tarvittavaksi ajaksi, jonka jälkeen sensori laitetaan nukkumaan tietyksi ajaksi. Kuva 17 havainnollistaa jaksotusta eri sensoreille.

Jaksotusta käytettäessä on otettava huomioon kuitenkin se, että sensoreiden käytön minimointi voi heikentää muun muassa eleiden tunnistamisen varmuutta, joten sensoridatan lukemisväliä ei voi venyttää liian pitkäksi [MLF08]. Ohjelmistokehittäjän haasteena onkin löytää sopiva tasapaino näiden tekniikoiden mahdollisimman vähäisen käyttämisen ja mahdollisimman hyvän käyttökokemuksen välillä.

Sensori	Työjaksot	Laskenta-aika/otos	Virta(J)/otos
Kiihtyvyysanturi	6s tarkkailua + 10s tauko	< 0.1s	0.359
Mikrofoni	5s + 180s	0.5s - 10s	1.114
GPS	Kysely joka 20s, lopetus 5min	< 0.1s	6.616
WiFi-skannaus	Aktivointi tapahtumana	< 0.1s	2.852

Kuva 17: Esimerkki työn jaksotuksesta, muokattu lähteestä [WLA09].

Muiden laitteiden lisäksi mobiililaitteissa virtaa kuluttaa laitteen prosessoriytimet. Prosessoriytimien kuormitus kasvattaa virrankulutusta kuvan 18 havainnollistamalla tavalla [PFW11]. Vaikka kyseessä on vain yhden älypuhelimen virrankulutus, voidaan kuvasta näkyvä idea yleistää koskemaan kaikkia mobiililaitteita, koska kaikkien mobiililaitteiden toimintaperiaate on kuitenkin samankaltainen. Eli kun prosessorin kuormitus kasvaa, kasvaa myös virrankulutus.



Kuva 18: Prosessorin kuormituksen vaikutus virrankulutukseen, muokattu lähteestä [PFW11].

Kuten pelisilmukan perustehtäviä käsittelevässä luvussa 3 todettiin, määrää pelisilmukan arkkitehtuuri sen, missä järjestyksessä ja kuinka usein pelin eri toimet suoritetaan. Toisin sanoen, pelisilmukan arkkitehtuuri vaikuttaa suoraan siihen, kuinka laitteen prosessoreita kuormitetaan. Pelisilmukan arkkitehtuuri tuleekin suunnitella pyrkien minimoimaan mobiililaitteen prosessorien kuormitus. Toisin sanoen pelisilmukan arkkitehtuuri tulee suunnitella siten, ettei pelisilmukan tehtäviä suoriteta liian usein. Pelisilmukan turhan tiheä suoritus aiheuttaa prosessorien turhaa kuormitusta ja sitä kautta turhaa akun kulutusta. Toisin kuin pöytätietokoneissa tai pelikonsoleissa, mobiililaitteiden pelisilmukoiden tulisi suorittaa vaadittuja toimia vain niin usein, kun on tarpeen, ei niin nopeasti kuin mahdollista.

Yksi keino prosessorin kuormituksesta johtuvan virrankulutuksen vähentämiseksi on siirtää prosessoria paljon kuormittavat tehtävät palvelimella suoritettavaksi [CBC10]. Kyseinen toimintatapa vähentää virrankulutusta, jos tiedonsiirto tapahtuu langatonta lähiverkkoa käyttäen. Jos tiedonsiirto tapahtuu käyttäen 3G:tä, ei virtaa kuitenkaan

säästy, koska kyseinen tiedonsiirtotekniikka kuluttaa liikaa virtaa. 2G-verkkoa käytettäessä virtaa ei kulu yhtä paljoa, kuin 3G:tä käytettäessä, mutta verkkoyhteyden nopeus ei välttämättä ole riittävä reaaliaikaisille peleille. Mobiililaitteita käytettäessä ei voida myöskään luottaa siihen, että verkkoyhteys on aina saatavilla, joten verkkoyhteyttä käyttävien ominaisuuksien tulisi toimia pelissä myös ilman tätä yhteyttä.

Toinen keino virransäästöön on pelin tilan huomiointi. Pelisilmukka voi ottaa huomioon pelin tilan, ja dynaamisesti vähentää prosessorin kuormitusta sekä sensoreiden tarkkailua [DiC13]. Esimerkiksi valikon ollessa auki pelin ei tarvitse piirtää grafiikkaa yhtä intensiivisesti [Imp14]. Sama toimii myös pelitilan päivittämiseen, eli pelin tilaa ei tarvitse päivittää esimerkiksi valikon ollessa auki yhtä usein, kuin pelaamisen ollessa aktiivista.

Mobiililaitteet mahdollistavat myös erilaiset käyttäjän tekemät eleet osana käyttäjäsyötettä, joko kosketusnäytöltä tai kiihtyvyysanturilta. Näin ollen mobiililaitteelle suunnitellun pelisilmukka-arkkitehtuurin tulisi tukea myös hahmontunnistusta. Hahmontunnistuksessa sensoreilta tuleva data täytyy poimia vähintään 20 Hz nopeudella, jotta mahdolliset hahmot siitä voidaan tunnistaa, mutta varsinaiselle hahmontunnistukselle riittää hitaampikin nopeus [AAF09, BKV97]. Hahmontunnistustehtävä tarkkailee muistiin tallennettavasta datasta merkkejä eleen alkamisesta. Se tarkistaa kerääntyneen sensoridatan esimerkiksi kymmenen kertaa sekunnissa (10 Hz). Jos se havaitsee eleen alkamisen, voidaan hahmontunnistuksen tarkkailunopeus kiihdyttää syötteen lukunopeuden tasolle, jotta eleen lopetus huomataan mahdollisimman nopeasti. Hahmontunnistuksen 10 Hz nopeus riittää hyvin eleiden aloituksen tarkkailuun, koska eleet kestävät harvoin alle 0.1s. Tosin, vaikka näin lyhyt ele olisi käytössä, havaittaisiin sekin, mutta joitakin kymmenesosasekunteja myöhässä, jota käyttäjä ei käytännössä välttämättä huomaisi.

3.4 Pelisilmukka-arkkitehtuurin suunnittelu mobiililaitteelle

Suunniteltaessa pelisilmukkamallin arkkitehtuuria mobiililaitteille voidaan pohjaksi ottaa jokin luvussa 3.2 esitellyistä malleista. Koska tämän hetken mobiililaitteet sisältävät moniydinprosessorin, on luonnollista ottaa lähtökohdaksi usean prosessorin tukeminen suunniteltavassa pelisilmukka-arkkitehtuurissa. Pelisilmukka suorittaa sille asetettuja tehtäviä tehokkaasti, kun se hyödyntää useampaa suoritinyksikköä yhtä aikaa

mutta optimitilanteessa siinä ei kuitenkaan suoriteta useampaa säiettä kuin suoritusjärjestelmässä on prosessoriytimiä. Jos tehtäväsilmukoita on useampia kuin suorittavia ytimiä, voi jokin silmukka joutua odottamaan suoritusaikaa kriittisellä hetkellä, joka taas heijastuu käyttäjäkokemukseen.

Deterministisyys on toivottava yleinen ominaisuus pelisilmukalta, ja mobiililaitteiden pelisilmukan arkkitehtuurille voidaan tavoitteeksi ottaa useamman prosessorin hyödyntämisen lisäksi deterministisyys. Pelisilmukalle arkkitehtuurin suunnittelussa tulisi ottaa huomioon prosessorin kuormituksen minimointi, joten, toisin kuin esimerkiksi pöytätietokoneissa ja pelikonsoleissa, pelisilmukan piirtotehtävälle tulee asettaa maksimiraja silmukan toistonopeudelle.

Tavoitellut suoritusnopeudet pelisilmukan eri tehtäville vaihtelevat. Piirtämiselle pidetään optimaalisena 50-60 Hz nopeutta [JZS12], ja tavoitteena pelisilmukan arkkitehtuuria suunniteltaessa on synkronoida se alemman tason grafiikan käsittelijöiden nopeuteen, jotka pyörivät usein 30 Hz tai 60 Hz nopeudella [Gre09 s. 304]. Näin ollen piirtämisen tavoitteelliseksi nopeudeksi saadaan 60 Hz.

Pelin päivitystehtävät synkronoidaan usein pelin piirtämiseen, sillä täsmälleen saman tilanteen, eli kuvan, piirtäminen useaan kertaan on turhaa. Jos pelin päivitystehtävät tehdään harvemmin, kuin pelin tila piirretään, voi piirtotehtävä ehtiä piirtämään pelin tietyn tilan useampaan kertaan ennen seuraavaa päivitystä. Jos pelin tila päivitetään esimerkiksi 30 kertaa sekunnissa, ja piirtäminen tapahtuu 60 kertaa sekunnissa, piirretään jokainen pelin tila keskimäärin kaksi kertaa. Mielekkäämpää olisi piirtää vain pelin tilassa tapahtuvia muutoksia. Toisaalta jos pelin tila päivitetään 30 Hz nopeudella ja piirtäminen tehdään samalla nopeudella, ei 30 Hz piirtonopeus tarjoa parasta mahdollista sulavuuden tuntua. Toisaalta päivitystehtävät eivät usein hyödy suorittamisesta yhtä usein piirtämisen kanssa, ja aiheuttavat vain turhaa järjestelmän kuormittamista.

Pelin näytölle piirtäminen pysyy sulavana vaikka päivitystehtävä suoritetaan sitä harvemmin, kun apuna käytetään niin sanottua animaation interpolointia, eli ennustamista. Tällöin piirtotehtävä ei piirrä pelin oliota sinne, mihin se piirsi sen aiemmalla kerralla, vaan interpolointiarvon avulla se voi laskea kohdan mihin pelin olio olisi liikkunut edellisten tietojen perusteella, ja piirtää sen tähän arvioituun kohtaan [Wik14d]. Interpoloinnilla voidaan arvioida pelin olion tilassa tapahtuva muutos, vaikka

sen varsinainen tila ei olisikaan muuttunut.

Pelin olion liikkuminen voi perustua esimerkiksi liikevektoreihin, joiden perusteella piirtotehtävä voi arvioida sen sijainnissa tapahtuneen muutoksen edellisen sijaintitiedon perusteella. Interpoloidessa piirtotehtävällä välitetään tieto siitä, kuinka lähellä/kaukana seuraava päivitystehtävä on edellisestä (0-100%), ja tätä lukua käytetään apuna sijainnissa tapahtuneen muutoksen arvioinnissa. Käytännössä pelin päivitystehtäviä ei siis tarvitse suorittaa yhtä tiheällä nopeudella, kuin piirtämistä. Alla oleva koodipätkä esittää kuinka interpolointi voidaan toteuttaa kooditasolla:

```
// Laske interpolointiarvo, jotta piirtäminen voi ennustaa piirrettävien olioiden tulevan sijainnin
float interpolation = execution_time / update_time;
// Piirrä & näytä
m_renderer->Render(interpolation, render_time);
m_renderer->Present();
```

Pelin päivitystehtävien haluttu suoritusnopeus riippuu paljon päivitystehtävien luonteesta, sillä jotkin tehtävät voidaan haluta tehdä mahdollisimman nopeasti (yli 100 Hz), kun taas esimerkiksi tekoälyn suorittamiseen voi riittää kerta tai kaksi sekunnissa (1 Hz tai 2 Hz) [Gre09 s. 305]. Yleensä toimiva päivitystiheys on jossain näiden välissä. Yleensä päivitystehtävien suoritustiheys kannattaa synkronoida piirtämisen kanssa niin, että ne tapahtuvat loogisessa järjestyksessä. Esimerkiksi jos piirtämisen suoritusnopeus on 60 Hz, on loogista laittaa päivitystehtävät tapahtumaan 30 Hz nopeudella, jolloin päivityksen tapahtuvat keskimäärin kerran kahta piirtosykliä kohden [Wit14]. Käyttäjäsyötteen lukemiselle riittää piirtämistä alempi suoritustiheys. Eleiden tunnistamiseen pohjautuvalle sovellukselle riittää käyttäjäsyötteen lukemisnopeudeksi 20 Hz [AAF09, BKV97], joten on perusteltua olettaa, että myös muulle syötteelle riittää sama kierrosnopeus.

Pelisilmukan tarkempaa arkkitehtuuria ja toteutustapaa suunniteltaessa tulisi ottaa huomioon mobiililaitekohtaiset tavoitteet [CVC10], eli

1. prosessorin kuormittamisen, eli pelisilmukan tehtävien suorittamisen minimointi
    - huomioimalla pelin tila pelisilmukan vaiheiden suorittamisessa,
    - pelisilmukan eri tehtävien maksimisuoritustiheyden rajoittaminen,
    - pelisilmukan piirtotehtävän suorittaminen vain tarvittaessa,
2. GPS:n ja sensoreiden aktiivisena pitämisen minimointi,

3. WLAN- ja GSM –verkkojen yli lähetettävän datamäärän minimointi ja
4. raskaiden laskemistehtävien ulkoistaminen palvelimelle, jos WLAN-verkko käytettävissä.

Ensimmäinen kohta pitää sisällään kaikki toimet, jotka voidaan tehdä mobiililaitteen prosessorin työtehtävien keventämiseksi pelisilmukan osalta. Nämä kolme kohtaa olisi hyvä ottaa huomioon jo pelisilmukan arkkitehtuuria suunniteltaessa.

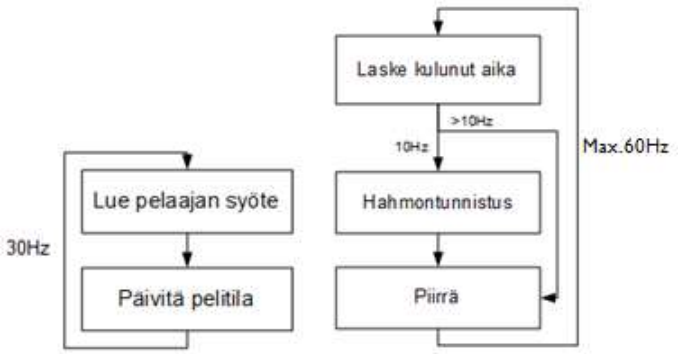
Toinen kohta viittaa neljännessä luvussa käsiteltyyn työn vaiheistamiseen lepotaukoineen GPS:n ja sensoreiden käsittelyssä. Eli jos vain mahdollista pidetään näitä laitteita aktiivisena vain vaaditun ajan verran, eikä lueta niiltä tulevaa dataa jokaisella pelisilmukan kierroksella.

Kolmas kohta viittaa siihen, ettei dataa lähetetä jatkuvasti jokaisella pelisilmukan kierroksella, vaan pyritään minimoimaan lähetetyn ja vastaanotetun datan määrä. Tähän voidaan pyrkiä esimerkiksi keräämällä data paketteihin ja mahdollisesti pakkaamalla se pienemmäksi tai lähettämällä vain välttämätön data.

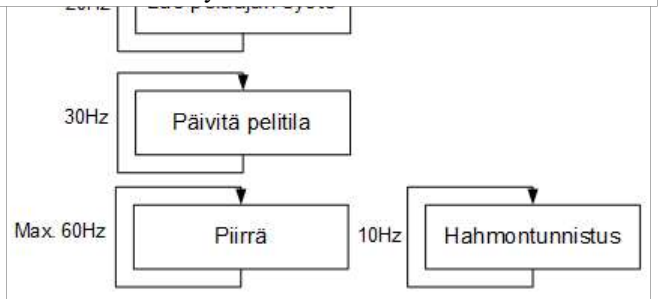
Neljännessä kohdassa viitataan ulkoisen palvelimen käyttöön laskentatehoa vaativien tehtävien suorittamisessa. Tämä keino on todettu toimivaksi, jos käytössä on langaton lähiverkko. Eli tämä keino ei ole yleispätevä ja aina käytettävissä oleva, mutta joissakin tilanteissa ja peleissä mahdollisesti toimiva optio.

Kun suorituslaitteessa on kaksi prosessoria, tai prosessoriydintä, on kannattavaa jakaa pelisilmukka kahteen säikeeseen, jotta se hyödyntää niitä molempia. Kaksiytimisille mobiililaitteille soveltuva determinismin huomioon ottavan pelisilmukkamalli ottaa huomioon tässä työssä käsitellyt tavoitekierrosnopeudet pelisilmukan eri tehtäville [Kuva 19]. Tällöin piirtotehtävälle on asetettu maksimirajaksi 60 Hz ja pelaajan syötteen lukemiselle ja pelin tilan päivitykselle on asetettu vakioiduksi nopeudeksi 30 Hz, joka on puolet piirtotiheydestä. 30 Hz nopeudella tapahtuva syötteen lukeminen on riittävä myös hahmontunnistusta varten. Varsinainen hahmontunnistus voidaan liittää piirtämisestä huolehtivaan säikeeseen luvussa 3 esitellyn yksisäikeisen vakiotaajuudellisen pelisilmukkamallin mukaisesti. Hahmontunnistus toimii luvussa 4.6 esitetyllä tavalla. Hahmontunnistus tarkistaa kerätyn syötteen, ja tutkii havaitaanko siitä eleen aloitus tai, jos meneillään on ollut eleen tarkkailu, loppuvan eleen tunnistus. Jos hahmontunnistus havaitsee eleen aloituksen, sen toistonopeus voidaan kiihdyttää

syötteen lukemisen tasolle, jotta eleen lopetus havaitaan mahdollisimman nopeasti. Jos hahmontunnistus taas havaitsee lopetuksen eleelle, se tarkistaa vastaako eleestä kerätty data mitään tunnettua eleen mallia, ja vastaavuuden löytyessä, se aktivoi tarvittavan tapahtuman, sekä palautuu hitaammalle 10 Hz toistonopeudelle.



Kuva 19: Determinismin huomioonottava pelisilmukkamalli kaksiytimisille mobiililaitteille.



Kuva 20: Determinismin huomioonottava pelisilmukamalli neliytimisille mobiililaitteille.

Neljä suoritinyksikköä sisältävät laitteet pystyvät suorittamaan neljää säiettä samanaikaisesti, eikä yksikään säie joudu odottamaan suoritinaikaa prosessoriytimeltä. Eli pelisilmukan arkkitehtuuri voidaan suunnitella käyttämään useampaa säiettä kuin kaksiytimisten arkkitehtuurien kanssa. Tällöin syötteen lukeminen, pelitilan päivitys, piirtäminen ja hahmontunnistus voivat kaikki tapahtua omassa säikeessään [Kuva 20].

Neliytimisen mobiililaitteen pelisilmukka-arkkitehtuurissa syötteen luku tapahtuu 20 Hz, pelitilan päivitys 30 Hz, hahmontunnistus 10Hz ja piirtäminen maksimissaan 60 Hz nopeudella. Jos peli ei tarvitse hahmontunnistustehtävää, jää yksi prosessoriydin vapaaksi. Sitä voidaan tällöin mahdollisesti käyttää jonkin muun tehtävän suorittamiseen.

4 Pelimoottorin sisäinen kommunikointi

Alkeellisin tapa toteuttaa pelimoottorin komponenttien kommunikointi on niin sanotusti kovakoodata komponenttien yhteydet toisiinsa. Pelin olio voi esimerkiksi tarkistaa tietyn toisen olion tilan joka pelisilmukan kierros. Tällainen tapa on kuitenkin skaalautumaton, herkkä virheille ja oliomallin vastainen. Pelit tarvitsevat yleisesti jonkinlaisen abstraktin viestijärjestelmän, jotta pelin komponentit tai oliot voivat

kommunikoida keskenään [Gre09 s. 712]. Tässä luvussa perehdytään aluksi tapahtumatietoja välittävän viestijärjestelmän perusteisiin, jonka jälkeen tutustutaan liitutaulumallin mukaisesti toteutettuun datan jakamiseen, jossa tieto välitetään komponenttien välillä julkisella ”ilmoitustaululla”. Tässä työssä ehdotettu malli sisältää sekä tapahtumajärjestelmän että liitutaulumallin mukaisen julkisen datavaraston.

4.1 Tarkkailijamalli

Peleissä on usein sellaisia olioita, jotka ovat kiinnostuneita muiden olioiden tiloista sillä ne voivat vaikuttaa sen omaan tilaan. Toisen olion tilasta kiinnostunutta oliota kutsutaan tarkkailijaksi, ja sen tulisi tällaisessa tilanteessa saada tietää toisessa oliossa tapahtuvista tilamuutoksista. Tämä voidaan toteuttaa niin sanotun julkaise-kirjaudu (*publish-subscibe*) -järjestelmän avulla. Tällaisen järjestelmän mukaan suunnitellussa pelin arkkitehtuurissa oliot tarkkailevan toisten niitä kiinnostavien olioiden tilaa.

Julkaise-kirjaudu-järjestelmän mukainen malli on yleinen tapa toteuttaa toisten olioiden tilamuutoksien vaikutus muihin olioihin. Tämän mallin mukaan toteutettuna pelin oliot pitävät listaa niistä toisista olioista (tarkkailijoista), joita kiinnostaa siinä tapahtuvat muutokset. Kun tällaisen olion tilassa tapahtuu muutos, ilmoittaa se siitä jokaiselle listalle olevalle, yleensä aktivoimalla kiinnostuneen olion niin sanotun takaisinkutsufunktion (*callback function*). Tällaista takaisinkutsutyyliä noudatettaessa tarkkaileva olio ilmoittaa tarkkailemalleen oliolle sen funktion, jonka se haluaa aktivoitavaksi tilanmuutoksen yhteydessä.

4.2 Tapahtumajärjestelmä

Tarkkailijamalliin perustuvaa julkaise-kirjaudu-järjestelmää voidaan hyödyntää myös yleisemmällä tasolla käyttösyötteen välittämisessä pelin olioille. Pelien olioita ei yleensä kiinnosta syötelaitteen tilan tarkastaminen jokaisella pelisilmukan kierroksella, vaan niitä kiinnostaa enemmän syötelaitteen tilassa tapahtuvat muutokset eli pelit ovat luontaisesti tapahtumaorientoituneita [Gre09 sivut 342 ja 773]. Pelin osien välinen kommunikaatio sisältää usein tietoa pelin tilaan vaikuttavasta tapahtumista, ja tästä niiden välisestä kommunikoinnista huolehtivaa viestijärjestelmää kutsutaan usein tapahtumajärjestelmäksi (*event system*). Peleissä tapahtuma viittaa sellaiseen tapahtumaan joka vaikuttaa pelin tilaan jollakin tavalla. Yleisimpiä pelin olioita

kiinnostavia tapahtumia ovat syötelaitteiden nappien painallusten aloitus- tai lopetushetket. Tällaisia tapahtumia voi syntyä esimerkiksi, kun pelaaja painaa peliohjaimen nappia tai pelin vihollishahmo havaitsee pelaajan.

Tapahtumajärjestelmä mahdollistaa pelimoottorin olioiden rekisteröidä kiinnostuksensa tietyntyyppisiä tapahtumia kohtaan, jolloin se tietää lähettää niille tiedon jos niitä kiinnostava tapahtuma aktivoituu [Gre09 s. 773]. Tapahtumajärjestelmä pohjautuu luvun alussa esiteltyyn tarkkailumallin ideaan. Tapahtumajärjestelmä rakentaa kokonaisen viestijärjestelmän tarkkailumallin idean pohjalle. Se mahdollistaa olioiden rekisteröidä kiinnostuksensa tiettyjä tapahtumia kohtaan samalla poimien tiedon niiden takaisinkutsufunktioista, joita se kutsuu tapahtumien aktivoituessa.

Tapahtumajärjestelmä voidaan toteuttaa joko etukäteen määritellyin staattisin linkityksin tapahtumien ja takaisinkutsufunktioiden välillä, joustavammin mahdollistaen dynaaminen linkitys niiden välille [Gre09 s. 775]. Tapahtumajärjestelmä voi tukea myös ajoitettuja tapahtumia, jolloin tapahtuma ei aiheuta välitöntä takaisinkutsufunktion kutsua, vaan kutsu tehdään tietyn ajan kuluttua. Ajoitetut tapahtumat mahdollistavat jaksolliset tapahtumat, eli jokin tapahtuma voi luoda uuden samanlaisen tapahtuman tietyn ajan kuluttua, koska tapahtuma voi rekisteröidä itsensä toistumaan tulevaisuudessa.

4.3 Liitutaulumalli

Datan jakaminen pelin sisäisten komponenttien välillä voidaan tehdä tapahtumajärjestelmän lisäksi käyttämällä liitutaulumallin (*blackboard-pattern*) mukaista datavarastoa. Liitutaulumallissa määritellään etukäteen julkisen datavaraston rakenne, jota käytetään ilmoitustaulun tavoin datan julkiseen jakamiseen [Wik14d]. Pelin eri komponentit tallentavat ja lukevat sieltä dataa suoritusaikaisesti, eikä sinne tallennettava data ole sidoksissa mihinkään. Komponentit tallentavat datavarastoon dataa. Datan jo ollessa siellä, kirjoitetaan tallennettava data sen päälle, jolloin datasta on tallessa aina uusin versio. Datan varastosta lukevan komponentin pitää tietää haluamansa datan tunnistin (*id*), mutta sen ei tarvitse tietää luettavan datan alkuperää.

5 Käyttäjäsyöte pelimoottoriarkkitehtuurissa

Ennen GPS- ja sensoridatan lukemiseen ja käsittelyyn liittyvien spesifisten puolien tarkastelua meidän tulisi ymmärtää kuinka käyttäjäsyötteen kerääminen ja käsittely tapahtuu pelimoottoriarkkitehtuurissa yleisellä tasolla. Tässä luvussa käydään läpi pelien käyttäjäsyötteen keräämisen ja käsittelyn pääperiaatteet sekä kuinka pelien käyttäjäsyötteessä usein hyödynnetty valvoja-malli (*observer-pattern*) ja tapahtumapohjainen viestijärjestelmä toimivat pelien käyttäjäsyötteessä.

5.1 Käyttäjäsyötteen käsittelyn pääperiaatteet

Käyttäjäsyötelaitteet voidaan jakaa kahteen osaan sen mukaan kuinka ne tarjoavat tilatietonsa pelin käyttöön [Gre09 s. 341]. Jotkut syötelaitteet tuottavat dataa tilastaan jatkuvana virtana, ja peli kysyy niiden sen hetkistä tilaa yleensä jokaisella pelisilmukan kierroksella (*polling*). Toisenlaiset syötelaitteet taas lähettävät syötedataa tietokoneelle vain silloin, kun niiden tilassa tapahtuu muutos. Muutoksen tapahtuessa ne lähettävät tietokoneelle rautatason keskeytyksen, jolloin tietokone lukee kyseisen laitteen tilan muistiinsa. Pelin voi tämän jälkeen käyttää muistiin tallennettua syötelaitteen tilatietoa haluamallaan tavalla, esimerkiksi sen päivitystehtävä voi lukea sen muistista, ja päivittää pelin tilaa tarvittavalla tavalla.

Suoraviivaisin tapa käsitellä käyttäjältä tuleva syöte on tarkastaa esimerkiksi hiiren tai näppäimen painallustieto peliolion tilapäivityksen yhteydessä ja reagoida tarvittavalla tavalla. Tällaiseen niin sanottuun kovakoodattuun tapaan sisältyy kuitenkin haittapuolia. Koska pelioliot on sidottu kyseisessä mallissa syötelaitteisiin kooditasolla, skaalautuu tällainen malli huonosti eri laitejärjestelmien tai syötelaitteiden suhteen. Lisäksi tästä tiukasta sitomisesta seuraa se, ettei pelitapahtumaan vaikuttavaa syötettä voida muuttaa suoritusaikaisesti, kuten vaihtaa toiminnon aktivoivaa näppäintä. Tällainen tyyli luo myös paljon toistuvaa koodia, mikä hankaloittaa koodin ylläpidettävyyttä ja muuteltavuutta.

Peliolioiden linkitys syötelaitteisiin voidaan rikkoa abstraktoimalla käyttäjäsyöte [Gre09 s. 350]. Peliolioita ei yleensä kiinnosta suoranaisesti syötelaitteelta tuleva data, vaan se, tuleeko niiden reagoida käyttäjäsyötteeseen. Pelihahmoa vastaavan olion tulisi esimerkiksi tietää tuleeko sen liikkua eteenpäin, eikä sitä, onko tietty näppäin

painettuna. Tämän tiedon se voisi esimerkiksi saada lukemalla muuttujan arvon toiselta luokalta, joka hallinnoi abstrakteja muuttujia, joihin käyttäjäsyöte vaikuttaa. Tällainen niin sanottu kontrolliluokka voisi esimerkiksi päivittää muuttujan *m_liiku* arvoksi *true*, kun tietty näppäimistön näppäin on painettuna. Muuttuja *m_liiku* olisi tässä esimerkissä jonkin käyttäjäsyötelaitteen tietyn näppäimen abstraktio, jolloin pelihahmoa vastaava olio tarkkailee tämän *m_liiku*-muuttujan tilaa, eikä suoraan syötelaitteen näppäimen tilaa. Tällainen syötteen abstraktointi mahdollistaa esimerkiksi syötteeseen käytettyjen näppäinten suoritusaikaisen muuttamisen.

Pelin oliolla voi olla käyttäjäsyötteeseen kahdenlainen suhde. Pelin olio voi olla sidottu niin tiukasti käyttäjäsyötteeseen, että sen pitää reagoida jokaiseen muutokseen kyseisessä käyttäjäsyötteessä. Toisaalta sille voi riittää se, että se reagoi tuoreimpaan käyttäjäsyötelaitteelta saatuun dataan kun sitä päivitetään. Pelin olion ollessa tiukasti sidottu käyttäjäsyötteeseen on luvussa 4.2 käsitelty tapahtumajärjestelmä yksi mahdollinen tapa varmistaa pelin olion reagointi jokaiseen syötteessä tapahtuvaan muutokseen. Tällöin jokainen käyttäjäsyötteessä tapahtuva muutos synnyttää tapahtuman, joka aiheuttaa pelin olion takaisinkutsufunktion kutsumisen ja pelin olioon tehdään sen haluama päivitys.

Jos pelin oliolle riittää sen tilan päivittäminen pelisilmukan päivitystehtävän suorituksen yhteydessä, eikä sen tarvitse välttämättä reagoida jokaiseen muutokseen käyttäjäsyötteessä, voidaan käyttäjäsyötteen välitys pelin oliolle toteuttaa ilman tapahtumajärjestelmää. Luvussa 4.2 esitelty liitutaulumalli on yksi yleinen malli pelin olioiden väliseen datan jakamiseen. Sen avulla käyttäjäsyöte voidaan tallentaa julkiseen varastoon, josta pelin oliot voivat lukea uusimman saatavilla olevan tiedon niitä kiinnostavasta käyttäjäsyötteestä.

5.2 GPS- ja sensoridata käyttäjäsyötteessä

GPS- ja sensoridatan käsittely eroaa perinteisten syötevälineiden, kuten näppäimistön ja hiiren, syötteen käsittelystä. Tässä luvussa perehdytään GPS- ja sensoridatan luonteeseen yleisesti sekä mobiililaitteen kannalta. Tämän jälkeen tarkastellaan millaisia käsittelymalleja tällaisen datan käsittelyyn on esitelty aiemmissa tutkimuksissa.

5.2.1 Yleistä GPS- ja sensoridatasta

Sensorit tuottavat jatkuvaa reaaliaikaista dataa havainnoistaan, ja ratkaistava kysymys on se, kuinka sensorin havainnoinnista syntymä datavirta saatetaan mobiililaitteessa suoritettavan sovelluksen käyttöön. Jotta GPS- ja sensoridatalle soveltuva käyttäjäsyötteen käsittely voidaan suunnitella, tulee ensin ymmärtää niiden luonne ja asema käyttäjäsyötteessä.

GPS-dataa käytetään joko kertaluonteisina sijaintikyselyinä, tai jatkuvana sijainnin päivityksenä. GPS-vastaanotin tarjoaa sovellukselle sijaintitiedon pituus- ja leveysasteena, tarkemmin GPS:n toiminta esiteltiin luvussa 2.4. Kolme suosituinta mobiililaitekäyttöjärjestelmää [Liite 3], eli Android, iOS ja Windows Phone antavat sovelluskehittäjälle mahdollisuuden pyytää laitealustalta kertaluontoista sijaintitietoa, tai jatkuvaa sijainninpäivitystä [Loc14, CLL14, Geo14]. Jatkuva sijainnin päivitys on jokaisessa näistä laitealustoista toteutettu takaisinkutsufunktion avulla.

Kertaluontoisesti pyydetyn sijaintitiedon avulla sovellus voi esimerkiksi merkitä sijainnin karttapohjalle. Jatkuvan GPS-datan avulla sovellus voisi taas laskea esimerkiksi käyttäjän liikkumisnopeuden. Saatu sijaintitieto voidaan tallentaa joko suoraan sovelluksen olioiden käyttöön, tai sitä voidaan käyttää tuottamaan abstraktioita käyttäjäsyötteestä. Tällaisena käyttäjäsyöteabstraktiona voi toimia sijaintitietojen, ja niiden aikaleimojen, avulla laskettu horisontaalinen ja/tai vertikaalinen nopeus.

Kuten GPS-data, myös sensoreilta luettu data voi vaatia käsittelyä. Sovelluksen olio ei välttämättä pysty käyttämään sensoreilta tulevaa dataa sellaisenaan, vaan se voi tarvita sensoreilta saadun datan avulla muodostetun käyttäjäsyöteabstraktion. Esimerkiksi pelin olio voi vaatia mobiililaitteen kallistuskulman, eikä kiihtyvyysanturin tuottamia kiihtyvyysarvoja kolmen ulottuvuuden suhteen. Tällöin kiihtyvyyksien avulla voidaan muodostaa käyttäjäsyöteabstraktio laitteen kallistuskulmalle. Sensoreiden tuottaman datan reaaliaikaisen käsittelyn lisäksi se voidaan myös tallentaa muistiin, ja muistiin kerääntyvästä datasta voidaan tämän jälkeen tunnistaa tuttuja malleja. Hahmontunnistusta ja sen käyttöä mobiililaitteiden käyttäjäsyötteessä käsitellään tarkemmin luvussa 5.3.

Sensoreilta saatu käyttäjäsyöte voi olla luonteeltaan joko välitöntä tai välillistä. Esimerkiksi laitteen kallistamisen tai sijainnin muutos voi vaikuttaa välittömästi

pelihahmoon, kun taas hahmontunnistukseen pohjautuva käyttäjäsyöte aktivoi tapahtuman vasta, kun pelaaja saa tehtyä hahmokuvion, eli eleen, loppuun. Järjestelmä ei voi tietää eleen alussa minkälaisen hahmokuvion pelaaja aikoo tehdä, vaan vaikutus peliin tapahtuu vasta hahmokuvion valmistuttua.

5.2.2 GPS- ja sensoridatan käsittely mobiililaitteessa

GPS-vastaanottimen ja sensoreiden käyttö mobiililaitteen sovelluksessa poikkeaa perinteisten syötelaitteiden, kuten hiiren tai näppäimistön, käyttämisestä pöytätietokoneessa. Kehitettäessä sensoreita hyödyntävää mobiilisovellusta, tulisi ottaa huomioon

1. resurssien käyttö,
2. skaalautuvuus,
3. datan saatavuus,
4. datan yksityisyys,
5. käyttöliittymä dataan ja
6. ohjelmalliset rajoitteet [WeL12].

Nämä huomioitavat kohdat toimivat myös pelisovelluksissa. Ensimmäinen ja toinen kohta ovat itsestään selviä ja toteutuvat myös peleissä. Myös listan kolmas kohta toteutuu pelien, ja tämän työn ehdottamassa arkkitehtuurimallissa. Pelin, ja pelimoottoriarkkitehtuurin, täytyy mahdollistaa sensoreiden tuottaman datan suora käyttö, eikä vain käyttäjäsyöteabstraktioiden käyttö. Syy on sama kuin muissakin sovelluksissa: muokattu data menettää tarkkuutta.

Listan neljäs kohta toteutuu myös peleissä. Mobiililaitteet ovat käyttökontekstiltaan henkilökohtaisia laitteitta, ja GPS-vastaanottimen, sekä sensoreiden keräämä data voidaan luokitella yksityiseksi dataksi. Tämän datan käyttämisessä, ja etenkin palvelimelle lataamisessa, täytyy ottaa huomioon yksityisyyssuoja ja turvallisuus. Käytännössä tällainen data pitää pyrkiä pitämään ensisijaisesti laitteen muistissa ja välttää datan lähettämistä palvelimelle. Jos data täytyy välittää palvelimelle, täytyy sovelluksen käyttäjältä pyytää lupa siihen, ja data täytyy salata.

Listan viides kohta ei sovellu täysin pelien kontekstiin. Kohta viittaa siihen, että

sensoreiden keräämän datan tulisi olla käytettävissä eri laitealustoilla. Tämän kohdan lähtöajatuksena on sellainen sovellus, jota käytetään vain sensoridatan keräämiseen, jolloin käyttäjää kiinnostaa sensoreiden keräämä data, ja hän haluaa lukea tätä dataa eri laitealustoilla. Peleissä tämä kohta tulee huomioida lähinnä siinä tapauksessa, kun kyseessä on monialustapeli jota pelataan mobiililaitteen lisäksi pöytätietokoneella. Tällöin mobiililaitteen keräämän GPS- ja sensoridatan tulisi olla luettavissa sekä mobiililaitteella, että pöytätietokoneella.

Myös listan kuudes kohta toteutuu peleissä, sillä jotkut pelit voivat käyttää keskuspalvelinta datan jakamiseen tai prosessointiin. Keskuspalvelinta käyttävä peli voi esimerkiksi kerätä pelaajien sensoritietoja tälle palvelimelle, jolloin pelaajien sensoridataa voitaisiin hyödyntää eri pelaajien mobiililaitteissa. Toinen mahdollinen palvelimen käyttökohde on sensoridatan prosessointi palvelimella, jolloin mobiililaitteen ei kuormitu.

Edellä olleen listan ensimmäinen kohta viittaa siihen, että ympäristöä jatkuvasti tarkkailevan mobiiliohjelman suunnitteluun liittyy monia haasteita [MLF08]. Yksi tällainen haaste on mobiililaitteen rajalliset resurssit, joihin järjestelmän sisäisillä prosesseilla on etuoikeus. Näin ollen mobiililaitteissa ajettavat kolmannen osapuolen sovellukset eivät välttämättä saa haluamiaan resursseja käyttöönsä juuri silloin kun ne niitä pyytävät, ja ne täytyy suunnitella ottamaan huomioon tämä resurssien rajallisuus. Rajallisiin resursseihin liittyy myös se, että mobiilisovellusten suunnittelussa täytyy ottaa huomioon myös suorituksen yllättävät keskeytyksen, joko resurssien katoamisen tai muun syyn takia. Toinen huomioon otettava haaste on tietoturvallisuus. Mobiililaitteen monet resurssit ovat tiukasti kontrolloituja, eikä niihin ole suoraa pääsyä. Tällaisia resursseja ovat muun muassa Bluetooth ja GPS-vastaanotin, joiden käyttö edellyttää käyttäjän hyväksyntää. Kolmantena haasteena on mobiililaitteiden virrankulutus, joka täytyy huomioida etenkin virtaa paljon kuluttavien ominaisuuksien, kuten radiotaajuuksia käyttävien tekniikoiden ja GPS:n käytössä. Ympäristön havainnointiin liittyvät prosessoria kuormittavat algoritmit voivat myös kuluttaa paljon akun virtatasoa jos niiden annetaan toimia kontrolloimattomasti.

Kuten luvussa 3.3 kirjoitettiin, joissakin tilanteissa voi olla mahdollista käyttää jaksottamista (*duty-cycles*) rajoittamaan GPS-vastaanottimen ja sensoreiden aiheuttamaa virrankulutusta [WLA09]. GPS-vastaanottimen ja sensoreiden aktiivisuuden

jaksottaminen mukailee luvussa 3.2.2 esitellyn yksisäikeisen vakiotaajuudellisten pelisilmukkamallin ideaa, jossa säie laitetaan yhden suorituskerran jälkeen aina nukkumaan hetkeksi. Jaksotettaessa GPS-vastaanottimen, tai jonkin sensorin, toiminta, voidaan kyseinen laite aktivoida esimerkiksi kerran minuutissa havainnoimaan kymmenen sekunnin ajaksi. Jos GPS-vastaanotin, tai sensori, havaitsee aktiivisena ollessaan tarpeen aktiivisemmalle toiminnalle, voidaan sen aktiivisuustasoa nostaa.

Aktiivisuuden jaksottamista voidaan soveltaa esimerkiksi GPS-vastaanottimen, ja sensoreiden lukemisessa sekä hahmontunnistuksessa. Pelin käyttäessä GPS-vastaanotinta, voidaan se aktivoida esimerkiksi kahden minuutin välein tarkistamaan onko pelaaja siirtynyt ulos. Jos pelaaja on ulkotilassa voidaan sen aktiivisuustasoa nostaa. Sensorit taas voivat oletuksena tarjota dataa esimerkiksi 60 Hz nopeudella, mutta käyttäjäsyötteeksi riittää, kuten luvussa 3.4 kirjoitettiin, 20 Hz nopeudella luettu data, jolloin datan lukeminen sensoreilta voidaan aktivoida 50 ms välein. Hahmontunnistustehtävä taas voi aktivoitua esimerkiksi kerran sekunnissa tarkistamaan kerääntyneen syötteen mahdollisten tunnistettavien hahmojen varalta.

Jos pelin käyttämän käyttäjäsyötteen vaikutus ei ole reaaliaikaista, vaan pelin mekaniikka hyödyntää käsiteltyä dataa jälkeenpäin, voidaan käyttäjäsyötteen käsittely mahdollisesti siirtää palvelimelle jolloin ei kuormiteta mobiililaitteen prosessoria [WeL12]. Tällöin mobiililaitteen keräämä syötedata lähetetään esimerkiksi joka kymmenes sekunti palvelimelle prosessoitavaksi, josta mobiililaite voi myöhemmin lukea prosessoidut käyttäjäsyöteabstraktiot.

Sensoridataan pohjautuvan käyttäjäsyötteen kerääminen ja käsittely voidaan jakaa viiteen vaiheeseen, joista kaksi ensimmäistä ovat yleisiä aina tapahtuvia vaiheita. Vaiheet 3,4 ja 5 liittyvät hahmontunnistukseen, jota käsitellään tarkemmin luvussa 5.3 [WeL12]. Nämä viisi vaihetta ovat

1. datan kerääminen sensoreilta,
2. sensoreilta kerätyn datan prosessointi,
3. valinnan analysointi ja/tai mallin soveltaminen,
4. datan ja tiedon raportointi ja
5. oppiminen tai mallin luominen.

Yllä kuvatut vaiheet mahdollistavat sensoreiden tuottaman datan käyttämisen monipuolisesti erityyppisissä sensoridataa käyttävissä ohjelmissa. Vaiheet toteutetaan järjestyksessä, poikkeuksena viides vaihe, joka on ulkopuolinen prosessi, joka voi tapahtua missä tahansa välissä.

Ensimmäisessä vaiheessa tapahtuva datankeräys tapahtuu joitakin kertoja sekunnissa. Jos sensoridataa käytetään hahmontunnistukseen kätketyn Markovin mallin avulla, on ehdoton alaraja sensoridatan poimimistiheydelle 8 Hz [Pyl05]. Käytännössä poimintatiheyden tulisi olla 19 Hz, jotta saavutetaan lähes 100% hahmontunnistusvarmuus. Yli 35 Hz nopeudella tapahtuva datanpoiminta ei tarjoa lisävarmuutta hahmontunnistukseen. Yleinen syötteenkeräysnopeus hahmontunnistukselle on 20 Hz [WeL12, AAF09].

Vaiheessa kaksi kerätty raakadata yhdistetään yhdeksi datamalliksi, joka esittää dataa useiden ominaisuuksien avulla, kuten keskimääräinen kiihtyvyys [WeL12]. Datan yhdistäminen voidaan tehdä joka kerta, tai esimerkiksi kerran sekunnissa.

Vaiheessa kolme verrataan edellisessä vaiheessa luotua datamallia malleihin tietokannassa. Tietokantaan on kerätty etukäteen erilaisia käyttäjäsyötteenä toimivia malleja. Vertailun tulos tallennetaan datana.

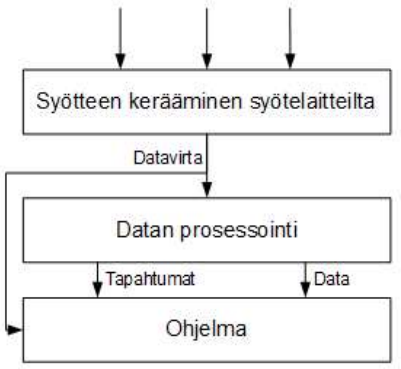
Vaiheessa neljä tarkastetaan vaiheen kolme tulos, ja jos vastaavuus johonkin tietokannassa olevaan malliin oli löytynyt, eli havaittiin esimerkiksi tietty ele, aktivoidaan vastaava pelin tilaan vaikuttava tapahtuma.

Viides vaihe ei linkity suoranaisesti toisiin vaiheisiin, vaan siinä luodaan vertailumalli tietokantaan, jota vaihe kolme käyttää vertailupohjana. Vertailumalli voidaan luoda joko etukäteen kerätyn datan avulla tai käytön aikana käyttäjän syöttämän datan perusteella.

5.2.3 Olemassa olevia malleja GPS- ja sensoridatan käsittelyyn

Syötteen käsittely voidaan yleistää tapahtuvaksi kahdessa tasossa ennen syötteen tarjoamista ohjelmalle [IlK02]. Kahdessa tasossa tapahtuvassa syötteen käsittelyssä ensimmäinen taso kerää syötteet eri laitteilta, jonka jälkeen raakadataa käsitellään toisessa tasossa [Kuva 21]. Raakadatan käsittely voi muodostaa datasta abstrakteja syötteitä, havaita hahmokuvioita ja luoda tapahtumia. Prosessointikerroksen tarkoitus on yksinkertaistaa ja yleistää ohjelmien rajapinta eri syötelaitteiden dataan. Jotkin ohjelmat voivat tosin haluta käyttää suoraan saatua raakadataa ilman prosessointitason

vaikutusta. Ideaalitapauksessa sovellus rekisteröi ne syötelaitteet, jotka sitä kiinnostavat, ja sitten syöte välitetään sovellukselle takaisinkutsufunktioiden avulla [IlK02].



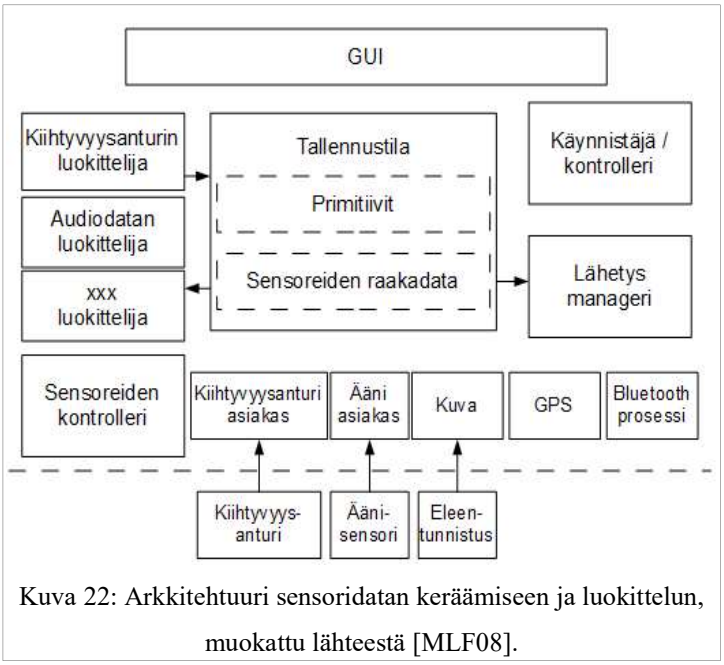
Kuva 21: Syötteen kerääminen sensoreilta, muokattu lähteestä [IlK02].

Yksi mahdollinen ratkaisu GPS- ja sensoridatan käsittelyyn on Raennon et al esittelemä mobiilisovellusten arkkitehtuurimalli joka voisi toimia myös peleissä [ROP05]. Kyseinen malli keskittyy tallentamaan ja tarjoamaan eteenpäin sensoreiden tiloissa tapahtuvat muutokset. Malli käyttää niin sanottua *Sensors*-komponenttia sensoritiedon lukemiseen, prosessointiin ja välittämiseen sovelluksille. Kyseinen komponentti toimii siis välikerroksena sensoreiden ja varsinaisten sovellusten välillä, jolloin sovelluksen ei siis tarvitse huolehtia mistä sen käyttämä data on lähtöisin. Kyseisessä arkkitehtuurimallissa käytetään luvussa 4.1 esiteltyä julkaise-kirjaudu (*publish-subscribe*) -mallia komponenttien välisen kommunikoinnin toteutukseen.

Raento et al toteavat työnsä lopussa heidän mallinsa toimivan kuitenkin paremmin luvussa 4.3 esitellyn liitutaulumallin mukaisesti toteutettuna [ROP05]. Liitutaulumallin mukaisesti toteutetulla datan jakamisella *Sensors*-moduulin ei tarvitsisi pitää kirjaa kenelle data pitää toimittaa, ja sovelluksen olioiden ei tarvitsisi tietää mitä sensoreita niiden tulisi kuunnella. Tällöin sensoridatasta huolehtiva ohjelmaosa ei toimittaisi dataa suoraan tietylle sovellukselle, vaan liitutaulumallin mukaiseen datavarastoon, eli niin sanotulle ilmoitustaululle, josta ohjelma lukee haluamansa datan ilman tietoa siitä, mistä kyseinen data on tullut.

Miluzzon et al esittelemässä arkkitehtuurimallissa GPS- ja sensoridatan kerääminen tehdään tehtäväpohjaisesti [MLF08]. GPS-vastaanottimelle ja jokaiselle sensorille luodaan oma tehtäväsäie, joka kerää laitteelta tulevan syötteen. Mallissa GPS-vastaanottimen ja sensoreiden tuottama data luokitellaan niin kutsutuiksi primitiiveiksi, eli abstraktioiksi, ennen datan varsinaista käyttöä. Abstraktioiksi voidaan kutsua muun muassa kiihtyvyysanturin datavirrasta luokiteltuja aktiviteetteja, kuten kävely, istuminen ja seisominen. Miluzzon et al esittelemässä mallissa luokittelijatehtävät prosessoivat niin sanotun raakadata, ja tuottavat sen pohjalta käyttäjäsyö-teabstraktioita.

Komponenttien välinen kommunikointi hoidetaan mallissa luvussa 4.3 esitellyn liitutaulumallin mukaisella jaetulla datavarastolla [Kuva 22]. Datavarastoon tallennetaan sekä GPS-vastaanottimelta ja sensoreilta saatu data, että tästä datasta prosessoidut abstraktiot.



Kuva 22: Arkkitehtuuri sensoridatan keräämiseen ja luokittelun, muokattu lähteestä [MLF08].

Edellä esitellyn mallin tapaisissa tehtäväpohjaisissa arkkitehtuurimalleissa käyttöliittymälaitteilta tuleva data käsitellään itsenäisten tehtävien avulla [JZS12]. Tällaisessa mallissa jokaiselle käyttöliittymälaitteelle, siis myös sensoreille, luodaan oma tehtävä, joka pyörii itsenäisessä säikeessä ja kerää kyseisen syötelaitteen syötteen. Syötelaitekohtaiset tehtävät tarkkailevat sisääntulevaa syötettä ja käsittelevät sen tarvittavalla tavalla. Tehtäväpohjaisessa arkkitehtuurissa syötetehtävät voivat joko aktivoida tarvittavat tapahtumat, tai ne voivat välittää syötedatan eteenpäin. Tehtäväpohjainen arkkitehtuuri ei välttämättä toimi peleissä hyvin, sillä suoritusnopeutta vaativissa peleissä prosessoriytimien määrän ylittämä tehtäväsäiemäärä voi aiheuttaa nykimistä pelikokemuksessa, kun kriittiset säikeet voivat joutua odottamaan vapautuvaa prosessoriydintä.

5.3 Hahmontunnistus käyttäjäsyötteestä

Sensoridatan tuominen sellaisenaan pelin käyttöön ei ole riittävää, jos pelissä halutaan hyödyntää mobiililaitteiden sensoreiden mahdollistamien eleiden käyttö käyttäjäsyötteessä. Sensoreiden tarjoamasta tiedosta on mahdollista hahmottaa erilaisia kuviomalleja, eli eleitä, joiden mukaan peli pystyy päättelemään käyttäjän välittämän syötteen. Tässä työssä perehdytään hahmontunnistukseen, jotta ymmärretään sen asettamat vaatimukset sensoridatan keräämiselle ja käsittelylle.

Ele voidaan määritellä HID-laitteelta tulevana sarjana käyttäjäsyötteitä tietyn aikajakson

sisällä [Gre09 s. 356]. Yksinkertaisin ele on kenties kahden syötteen sarja, joista ensimmäinen osa on napin painallus alas, ja jälkimmäinen napin painalluksen lopetus.

Toisin kuin sensorilta saatu reaaliaikainen syötedata, eivät hahmontunnistukseen pohjautuvat eleet toimi reaaliaikaisena käyttäjäsyötteenä. Peli ei reagoi välittömästi eleen alkamiseen, vaan tehty ele aktivoi tapahtuman vasta, kun sensorilta on saatu kaikki eleeseen kuuluva data. Jos mobiililaitteen peli käyttää käyttäjäsyötteenä esimerkiksi ilmaan piirrettyjä hahmokuvioita, ja pelaaja aloittaa ilmaan piirtämisen, ei piirtämisen aloitus aktivoi pelistä vielä mitään. Kun pelaaja lopettaa ilmaan piirtämisen, pelin hahmontunnistus huomaa muodostuneen hahmokuvion, ja aktivoi hahmokuvioon sidotun tapahtuman.

5.3.1 Hahmontunnistuksen pääperiaatteet

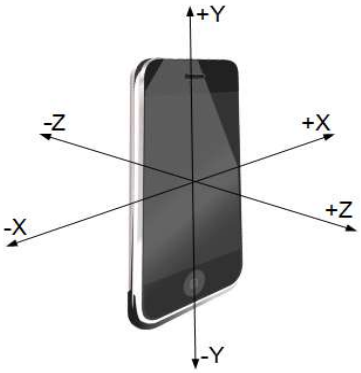
Yleisesti eleen tunnistus toteutetaan pitämällä muistissa lyhyt historia käyttäjän välittämistä syötteistä, ja tarkkailemalla toteuttaako muistiin kasaantuva syötesarja jonkin tiedossa olevan syötemallin [Gre09 s. 356]. Jos uutta syötettä ei tule tietyn ajan sisällä, tyhjennetään muistiin tallennettu syötehistoria. Hahmontunnistus voidaan jakaa kahteen osaan, joista ensimmäisessä koostetaan tietokanta tunnistettaville eleille, ja toisessa tunnistetaan eleitä vertaamalla syötesarjasta rakennettuja malleja mallitietokantaan [JoC09].

Hahmontunnistus pohjautuu mitattavien hahmojen erotteluun sisään tulevasta datasta ja niiden luokitteluun määriteltyihin kategorioihin [Män01, SPH08]. Sama periaate toimii kaikilla tunnistettavilla hahmoilla, oli kyseessä sitten liikkeeseen, kosketukseen, tai mihin tahansa muuhun syötekanavaan pohjautuva kuvio. Sisääntulevasta signaalista mitataan hahmo, joka luokitellaan vertaamalla sitä muistiin tallennettuihin malleihin. Jos mitatulle hahmolle löytyy luokka, ollaan tunnistettu malli.

Tunnistettavan hahmon määrittelee siihen liittyvät ominaisuudet, kuten sen lähettämän signaalin, esimerkiksi liikevektorin, vaihtelut [Män01]. Tunnistettavan hahmon määrittelevä ominaisuus voi olla joko yksittäinen mittaus, tai joukko mittauksia, ja niiden käsittelyn tuottama tulos. Prosessointia vaativat mittaustulokset käsitellään usein joukkona, ja tuloksena on usein vähemmän tilaa vievä hahmon määrittävä datajoukko.

Liikkeeseen pohjautuva hahmontunnistus perustuu kiihtyvyysanturilta mitattuun dataan. Kiihtyvyysanturilta tulevassa datassa on mukana kolme arvoa, jotka kuvaavat sen

hetkistä kiihtyvyyttä kaikissa kolmessa ulottuvuudessa [Kuva 23]. Jokainen sisääntulevasta kolmesta signaalista kuvaa kiihtyvyyttä yhdessä suunnassa 3D-avaruudessa kuvan 20 mukaisesti. Kiihtyvyysanturin välittämät arvot vaihtelevat tyypillisesti [-3,...,3]g välillä, jossa g tarkoittaa maan vetovoimaa (~9.81m/s^2) [JoC09].



Kuva 23: X-, Y- ja Z-akselit joiden suhteen kiihtyvyyttä mitataan.

Hahmontunnistuksen perusta on tunnistettavan hahmon mallintaminen käsiteltävään muotoon. Tunnistettavaa hahmoa kuvaava data täytyy käsitellä jonkin mallin mukaisesti, jotta hahmon tunnistus voidaan tehdä. Hahmontunnistamiseen on monenlaisia lähestymistapoja, kuten: raaka voima (b*rute force*), sumea logiikka (*fuzzy logic*), Gaborin aallokemuutos (g*abor wavelet transform*), kätketty Markovin malli (*hidden Markov model*), tukivektorikone (s*upport vector machine*) ja neuroverkot (*neural networks*) [JoC09]. Kätketty Markovin malli (*hidden Markov model*, HMM) on yleinen hahmontunnistustapa, ja sitä käytetään monissa hahmontunnistukseen liittyvissä tutkimuksissa [muun muassa JoC09, Män01, SPH08]. Se valittiin myös tähän työhön käytettäväksi hahmontunnistuksessa, koska sitä käytetään yleisesti kiihtyvyysanturiin pohjautuvien eleiden tunnistamisessa.

5.3.2 Hahmontunnistuksen vaiheet

Tyypillinen vaihejako hahmontunnistamiselle pitää sisällään poiminnan, mallin käyttämisen ja luokittelun [SPH08]. Aluksi sensoreilta tuleva data, kerätään datajoukoksi. Tämän jälkeen tähän kerättyyn datajoukkoon sovelletaan jotain käsittelymallia, joka tässä työssä on HMM, jota käsitellään tarkemmin luvussa 5.3.4. Lopuksi saatu malli pyritään luokittelemaan vertaamalla sitä mallitietokantaan. Jos löydetään vastaavuus tehdyn mallin ja kannassa olevan mallin välillä, on hahmo tunnistettu.

Hahmontunnistaminen voidaan jakaa neljään tai viiteen yleiseen osa-alueeseen, riippuen käytettävästä syöttötekniikasta [JoC09]. Kiihtyvyysanturia käyttävien eleiden täytyy tehdä yksi vaihe enemmän, kuin esimerkiksi kosketusnäyttöön pohjautuvien eleissä. Kiihtyvyysantureilta saatu data käsitellään niin sanotussa poimintavaiheessa, jolloin

käsiteltävän datan määrä vähenee. Hahmontunnistamiseen kuuluvat työvaiheet ovat

1. segmentointi, jolla päätellään milloin ele alkaa ja loppuu,
2. suodatus, jolla poistetaan jatkoon pääsevästä syötteestä turhat eleeseen kuulumattomat osat,
3. poiminta (*quantitizer*), jolla erotetaan syötevirrasta pienempiä arvojoukkoja esimerkiksi *k-means*-algoritmin avulla,
4. mallin luominen, jolla eleestä tehdään käsiteltävä malli ja
5. luokittelija, jolla verrataan havaitusta eleestä tehtyä mallia tietokannassa oleviin eleisiin ja havaitaan mahdolliset vastaavuudet.

Kolmas näistä työvaiheista, poiminta, tehdään siis vain liike-eleisiin pohjautuvien eleiden tunnistamisessa. Siinä kiihtyvyysanturilta tuleva data jaotellaan valittuun määrään arvojoukkoja, jolloin datan määrä vähenee ja koko datajoukon käsittely HMM:n avulla on mahdollista [JoC09].

Jotta jokin hahmo, eli ele, voidaan tunnistaa, täytyy pelin ensin havaita elettä kuvaavan syötesarjan alkaminen. Tämän jälkeen pelin täytyy tietää milloin elettä vastaavan syötesarjan lukeminen lopetetaan. Kosketukseen perustuvassa eleessä aloitus- ja lopetushetki tulevat suoraan näytön kosketuksen alkamis- ja loppumishetkistä, mutta kiihtyvyysanturiin pohjautuvat liike-eleet ovat hankalampia aloituksen ja lopetuksen suhteen. Kiihtyvyysanturi lähettää dataa jatkuvana virtana yleensä 20-80 Hz taajuudella, ja ongelmana on tietää, milloin eleen tunnistamisen pitäisi käynnistyä.

Käytännössä liikkeeseen perustuvan eleen aloitushetken päättelemiseksi täytyy tarkastella kiihtyvyyteen perustuvien eleiden yleistä luonnetta ja löytää niille sopivia ominaisominaisuuksia [JoC09]. Tyypillinen liike-ele alkaa nopealla kiihdytyksellä, pitää sisällään jatkuvaa suunnan muutosta ja kestää kauemmin kuin 0.3 sekuntia mutta enintään kaksi sekuntia. Liike-eleen alku ja loppukohta voidaan löytää Euclidean etäisyyden (*Euclidean distance*) laskemiskaavalla [JoC09, kuva 24]. Kaavan avulla voidaan laskea sen hetkisen kiihtyvyyden muutosmäärä laskemalla sen hetkinen kiihtyvyysvektori. Jos kaavasta saatava arvo on suurempi kuin 0.3, niin eleen segmentointi voidaan aloittaa, ja jos eleen mittaaminen on jo käynnissä, ja arvo tippuu alle 0.1, niin ele voidaan katsoa

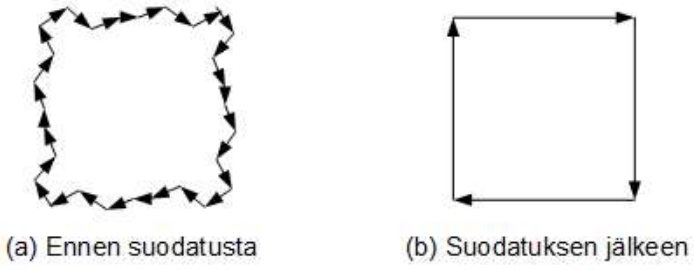
$$D = \sqrt{(x_k - x_{k-1})^2 + (y_k - y_{k-1})^2 + (z_k - z_{k-1})^2}$$

Kuva 24: Vektorin muutoksen suuruus [JoC09].

loppuneeksi.

5.3.3 Datan suodatus

Sensoreilta tulevassa datassa on yleensä mukana paljon eleeseen kuulumatonta häiriödataa, ja etenkin kiihtyvyysantureiden kanssa käytetään niin sanottua suodatusvaihetta vähentämään tätä häiriötä. Suodatusmenetelminä on käytetty muun muassa kiihtyvyysanturilta tulevaan dataan käytetty alipäästösuodin (*low pass filter*), jolla voidaan poistaa kiihtyvyystiedosta painovoiman vaikutus, ja raja-arvo -suodinta (*idle threshold filter*), jolla voidaan suodattaa datasta pois turhat arvot [JoC09, SPH08]. Raja-arvo -suodinta voidaan käyttää joko kiihtyvyysdataan, josta on poistettu alipäästösuotimella maan vetovoiman osuus, tai dataan, jossa maan vetovoima on mukana. Käytännössä raja-arvo -suotimena voidaan käyttää esimerkiksi mallia, jossa datasta poistetaan arvot, jotka ovat kuvassa 21 näkyvän kaavan mukaisesti laskettuna alle 0.2 tai 1.2g arvoisia, riippuen onko maan vetovoima, *g*, mukana käsiteltävässä datassa. Jos vektorin koko on raja-arvoa pienempi, sitä ei poimita jatkoon. Suotimena voidaan käyttää myös vektorien samankaltaisuutta tutkivaa suodinta (*directorial equivalence filter*). Se tutkii ja suodattaa pois kiihtyvyysvektorit, jotka ovat liian samanlaisia edellisten vektorien kanssa. Vektori jätetään valitsematta, jos yksikään sen x, y, tai z komponenteista ei eroa selvästi edellisistä jo valituista vektoreista. Toisin sanoen $|a_c^{(n)} - a_c^{(n-1)}| \leq e$ kaikille c:n arvoille, jossa $c \in \{x, y, z\}$ ja e:n arvoksi voidaan valita 0.2 [SPH08]. Vektoridatasta karsiutuu tällaisen suodatuksen avulla pois häiriö, ja optimaalisessa tilanteessa neliön muotoisen eleen muodostamat suuntavektorit yhdistyvät, ja lopullinen ele pitää vain neljä suuntavektoria suodatuksen jälkeen [Kuva 25].



Kuva 25: Esimerkki suotimen soveltamisesta vektoridataan, muokattu lähteestä [JoC09].

5.3.4 Kätketty Markovin malli

Hahmontunnistamisen neljännessä vaiheessa ele mallinnetaan malliksi, jota voidaan

käyttää datan käsittelyssä. Tässä työssä ehdotettun mallin vaatimuksena on hahmontunnistuksen tukeminen, ja jotta hahmontunnistuksen vaatimukset käyttäjäsyötteen käsittelylle voidaan ymmärtää, tutustutaan tässä luvussa yhteen eleen mallinnustapaan.

Yksi yleinen tapa mallintaa eleet on käyttää kätkettyä Markovin mallia (*hidden Markov model*, HMM). Sitä on käytetty useissa hahmontunnistukseen liittyvissä tutkimuksissa meleiden mallinnustapana [esimerkiksi JoC09, SPH08]. Avainajatus HMM-pohjaisessa hahmontunnistuksessa on luoda etukäteen HMM:n mukainen malli halutuista eleistä, jonka jälkeen suoritusaikaisesti HMM:llä mallinnettuja eleitä voidaan verrata näihin etukäteen tallennettuihin malleihin. Jos vastaavuus löytyy, on suoritusaikainen ele tunnistettu.

HMM on kaksinkertainen stokastinen prosessi, jossa toinen stokastinen prosessi on piilossa ja toinen esillä ja tarkkailtavana [RJ86]. Piilossa olevaa stokastista prosessia voidaan tarkkailla toisen esillä olevan stokastisen prosessin avulla, joka tuottaa sarjan havaittavia symboleita kuvaamaan kätkettyä stokastista prosessia. HMM:n mallintama ele rakentuu siis datasarjasta, joista jokainen data on stokastisen prosessin tuottama havaintosymboli.

HMM:n avulla käsitellään syötteenä tuleva datasarja, ja tästä sarjasta luodaan malli joka kuvaa elettä. Datasarjan muuntaminen malliksi tapahtuu tilojen ja niiden välisten tilasiirtymien avulla. HMM:n pääidea voidaan ajatella kolmena piirteenä, jotka ovat

1. äärellinen määrä (*N*) tiloja,

2. siirtyminen jokaisella ajanhetkellä *t* uuteen tilaan käyttäen mallin siirtymätodennäköisyyksiä (*transition probabilities*) ja

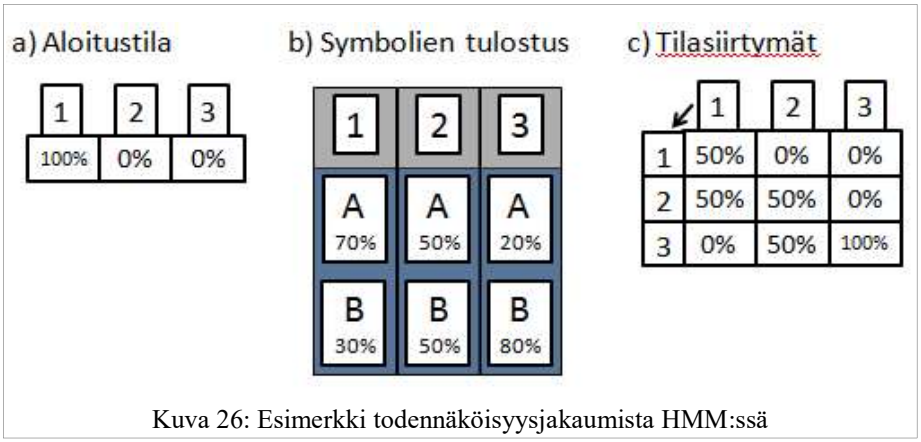
3. tilasta kertovan havainnoitavan symbolin, joita on M kappaletta, tulostaminen jokaisen tilasiirtymän jälkeen [RJ86].

Jokaisella tilalla on yksilölliset todennäköisyydet jokaisen havaintosymbolin tulostukseen. HMM:ssä käytetään matriisia esittämään symbolien tulostustodennäköisyydet (*emission probabilities*) eri tiloissa. HMM:llä voi olla esimerkiksi kolme tilaa ja kaksi tulostussymbolia, jolloin jokaiselle kolmelle tilalle määritellään

- todennäköisyys olla alkutila,

- todennäköisyydet kummankin havaintosymbolin tulostukselle ja
- todennäköisyydet siirtymiselle mihin tahansa kolmesta tilasta [Kuva 26].

Tilasiirtymä tilasta toiseen sisältää mahdolisuuden siirtyä takaisin samaan tilaan, eli mallia esittävä datasarja Tällaisen esimerkin tapauksessa Kyseisessä esimerkissä yhdestä tilasta siirrytään aina joko samaan tilaan, tai seuraavaan tilaan. Kolmannessa, eli viimeisessä tilassa, siirrytään aina takaisin samaan tilaan.

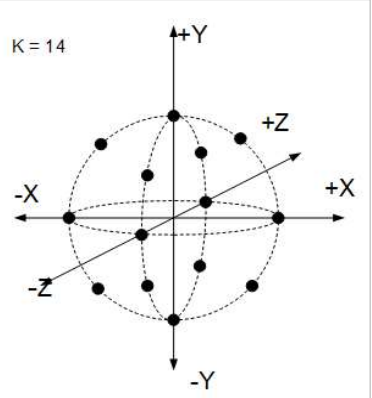


Kuva 26: Esimerkki todennäköisyysjakaumista HMM:ssä

Jos HMM-pohjaisessa hahmontunnistuksessa käytetään kiihtyvyysanturia, täytyy huomioon ottaa se, että kiihtyvyysanturin tarjoaman datan määrä on niin runsasta, ettei HMM pysty käsittelemään sitä yhdellä kerralla [SPH08]. Tähän tarjoaa ratkaisun luvussa 7.1.1 mainittu hahmontunnistuksen vaiheisiin sisältyvä poimintavaihe (*quantitizer*), joka tehdään vain liike-eleiden kiihtyvyysanturidatalle. Siinä kiihtyvyysanturilta tullut data jaetaan osajoukkoihin, jolloin datan kokonaismäärää kuvaa saatu jakautuma.

Jokaisen kiihtyvyysvektorin käsittely aiheuttaisi runsaasti datan prosessointia, ja yleisesti käytetäänkin mallia, jossa kiihtyvyysanturilta tulevat kiihtyvyysvektorit jaetaan eri akseleiden kiihtyvyysarvon pohjalta lähimpään pisteeseen k-keskiarvo algoritmin avulla (*k-means algorithm*) [JoC09]. Tällöin jokainen kiihtyvyysvektori kuvataan osana jotakin osajoukkoa, eli algoritmin avulla koko datajoukko esitetään joukkona keskiarvoja. Kolmiulotteisissa eleissä toimivaksi on todettu osajoukkojen jakauma, jossa jokainen kiihtyvyysvektori sijoitetaan yhteen 14:sta osajoukosta [SPH08, kuva 27]. Datan määrän vähentyminen poimintavaiheessa säästää myös mobiililaitteen muistia, sillä käsitelty data sisältää raakadataa vähemmän tietoa.

5.3.5 Tunnistettavan hahmon luokittelu



Kuva 27: Ryhmäpisteiden sijoittuminen 3D-avaruuteen, muokattu lähteestä [SPH08].

Hahmontunnistuksessa luokittelijaa käytetään löytämään mitatulle mallille paras vastaavuus mallitietokannasta [JoC09]. Sen tehtävänä on verrata luotua mallia tietokannassa oleviin vertailumalleihin, ja löytää todennäköisin vastaavuus. Jotkut hahmontunnistusjärjestelmät käyttävät omia luokittelija-algoritmeja, mutta yleisemmin käytössä on jokin yleisessä tiedossa oleva luokittelija. Joitakin yleisiä tapoja toteuttaa hahmontunnistuksessa käyte luokittelija ovat naiivi Bayes -algoritmi (*Naive Bayes algorithm*), jota kutsutaan myös yksinkertaiseksi todennäköisyysluokittelijaksi (*simple probabilistic classifier*) [JoC09, SPH08], valintapuu (*decission tree*) [MLF08], K-lähintä naapuria (*K-nearest neighbor*), tukivektorikone (*support vector machine*) [AlI13] ja eteneenpäinsyöttöalgoritmia (*forward algorithm*) [EAA08].

Tässä työssä luokittelijana käytetään eteneenpäinsyöttöalgoritmia, joka laskee todennäköisyyden HMM-mallin läpi etenemiselle käytettäessä annettua sarjaa. Toisin sanoen algoritmi antaa todennäköisyyden sille, vastaako annettu sarja tietokannassa olevaa mallia [EAA08]. Eri luokittelijoiden toimivuudesta hahmontunnistuksessa on tehty joitakin tutkimuksia, ja luokittelijatyypeistä valintapuuhun pohjautuvat ratkaisut vaikuttavat olevan kaikkein luotettavimpia tunnistamisen suhteen [AlI13, BaI04].

6 Ehdotettu arkkitehtuurimalli GPS- ja sensoridatan käsittelyyn

Tässä luvussa esitellään tämän työn ehdottama arkkitehtuurimalli GPS- ja sensoridatan sitomiseksi osaksi mobiililaitteen pelimoottoriarkkitehtuuria. Ehdotetusta arkkitehtuurimallista esitellään versio kaksi ja neliytimisille mobiililaitteille.

6.1 Ehdotetun mallin tausta

Tämän työn lopullinen tavoite ja kontribuutio en ehdottaa arkkitehtuurimallia joka

mahdollistaa GPS- ja sensoridatan hyödyntämisen mobiililaitteiden pelimoottoriarkkitehtuurissa. Tavoitteena suunnitellulle arkkitehtuurimallille on laitealustariippumattomuus, skaalautuvuus ja GPS- ja sensoridatan monipuolinen käyttäminen osana käyttäjäsyötettä.

Laitealustariippumattomuus viittaa siihen, että ehdotetun mallin tulee olla sellaisella yleisellä tasolla, ettei se sitoudu mihinkään tiettyyn laitealustaan, esimerkiksi sensoreiden hyödyntämisen suhteen. Skaalautuvus viittaa mallin skaalautumiseen mobiililaitteiden kehityksen myötä, esimerkiksi kasvavien prosessoriytimien ja sensoreiden määrän suhteen. GPS- ja sensoridatan monipuolinen käyttäminen viittaa siihen, että ehdotetun mallin ei tule rajoittaa mahdollisuuksia GPS- ja sensoridatan käyttämiseen. Riippuen pelistä, peli voi tarvita niiltä tulevan datan esimerkiksi sellaisenaan, käsiteltynä tai siirrettynä ulkoiselle palvelimelle. Ehdotetun mallin tulee myös mahdollistaa GPS-vastaanottimelta ja sensoreilta tulleen syötehistorian pitäminen tallessa, sillä hahmontunnistuksen toteutus vaatii yleensä sen [Gre09 s. 356].

Ehdotetun mallin piiriin kuuluu pelimoottorista dynaaminen pelin oliomalli (*Dynamic game object model*), tapahtuma/viestijärjestelmä (*Event/messaging system*) ja reaaliaikainen toimijapohjainen simulaatio (*Real-time agent based simulation*) [Gre09 s. 45]. Dynaaminen pelin oliomalli vastaa pelin interaktiivisuudesta, eli pelin käyttäjäsyötteen käsittelystä ja pelin olioiden reagoinnista syötteeseen. Tapahtuma/viestijärjestelmä vastaa pelin komponenttien välisestä kommunikoinnista ja pelin olioiden reagoinnista havaittuihin tapahtumiin. Reaaliaikainen toimijapohjainen simulaatio tarkoittaa pelin pelisilmukkaa, joka vastaa pelin tehtävien toistamisesta ja reaaliaikaisuudesta.

Dynaamisen pelin oliomallin lisäksi tarvitaan myös tapahtuma/viestijärjestelmä, koska pelin olioiden tilan päivitys ei aina liity käyttäjäsyötteeseen [Gre09 s. 342]. Pelin tilassa voi tapahtua käyttäjäsyötteestä riippumattomia muutoksia, jotka vaativat pelin olioiden reagointia. Tällainen muutos voi esimerkiksi olla pelin olion liikkuminen niin, että se törmää toiseen pelin olioon, jolloin tästä muutoksesta viestitetään pelin muille olioille.

Ehdotetun mallin täytyy olla yllä esiteltyjen pelimoottorin osien mukainen, jotta sitä voidaan hyödyntää mobiililaitteiden pelimoottorien toteutuksessa. Sen suunnittelu perustuu näiden kolmen pelimoottorin osan tutkimiseen GPS- ja sensoridatan näkökannalta.

Käyttäjäsyötteen osalta ehdotetun mallin tulee ottaa huomioon myös GPS-vastaanottimen ja sensorien asettamat vaatimukset sille. Käyttäjäsyötteen lukeminen ja käsittely tapahtuu omassa tehtävässään osana pelisilmukkaa. Suunniteltaessa ehdotettua mallia, tulee syötteen lukemisessa GPS-vastaanottimelta ja sensoreilta ottaa huomioon etenkin mobiililaitteen resurssien käyttö ja skaalatuvuus [WeL12]. Ehdotetun mallin tulee kytkeä GPS- ja sensoridatan lukeminen ja käsittely peliin mobiililaitetta mahdollisimman vähän kuormittavalla tavalla. Mobiililaitteen resurssit, eli akun kesto, prosessointiteho, muisti ja verkkoyhteydet ovat rajalliset, ja ehdotetun mallin tulee käyttää niitä maltillisesti. Ehdotetun malli tulee skaalautua mobiililaitteiden kehityksen mukana, eli sen tulee huomioida prosessoriytimien ja sensoreiden määrän kasvu.

Pelin yleinen toteutustapa ja pelin kehittäjän halu ohjaavat usein sitä, kuinka paljon pelissä käytetään tapahtumajärjestelmää pelin olioiden päivittämiseen. Äärimmilleen vietynä kaikki GPS-vastaanottimelta ja sensoreilta tuleva data voidaan käsitellä tapahtmapohjaisesti, mutta voi olla myös tilanteita, joissa niiltä tulleeseen dataan pohjautuva käyttäjäsyöte on hyvä käsitellä tapahtumapohjaisesti. Esimerkiksi hahmontunnistusta käytettäessä on luontevaa käyttää tapahtumajärjestelmää viestimään pelin olioille tapahtuneesta eleestä.

Pelin reaaliaikaisuudesta huolehtii pelisilmukka, joka toistaa pelin tarvitsemia tehtäviä. Yleensä pelisilmukat suunnitellaan maksimoimaan suoritusteho, mutta mobiililaitteilla optimointi tulee tehdä virrankulutuksen suhteen [CBC10]. Virrankulutus on kytköksissä prosessoriytimien kuormitukseen, joten tavoitteena ehdotetussa mallissa on suorittaa pelisilmukan tehtäviä vain tarvittavalla tiheydellä. Pelin tehtävien reaaliaikaiseen suorittamiseen liittyy myös mobiililaitteista löytyvien prosessoriytimien määrä. Tavoitteena ehdotussa mallissa on hyödyntää mobiililaitteista löytyvät prosessoriytimet, mutta välttää prosessoriytimien määrää suurempi tehtäväsäikeiden määrä.

6.2 Pelisilmukan arkkitehtuuri ehdotetussa mallissa

Ehdotetun mallin reaaliaikaisuudesta huolehtivan pelisilmukan arkkitehtuuri on suunniteltu luvussa 2.3 esiteltyjen tietojen pohjalta. Mobiililaitteesta saadaan ulos optimaalinen suoritusteho, jos pelin toiminta jaetaan yhtä moneen säikeeseen kuin suoritusjärjestelmässä on prosessoriytimiä. Joissakin tilanteissa prosessoriytimien määrää suurempi säiemäärä voi toimia tehokkuuden suhteen, mutta peleissä siihen

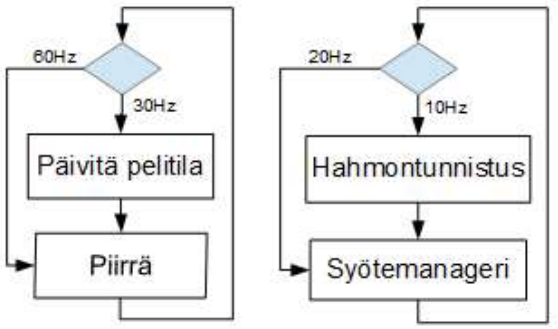
sisältyy riskejä. Jos pelissä suoritussäikeitä on enemmän kuin suorittavia prosessoriytimiä, voivat jotkin kriittiset säikeet joutua odottamaan vapaata suoritusaikaa kriittisellä hetkellä. Esimerkiksi jos piirtämisestä huolehtiva säie joutuu odottamaan suoritusaikaa prosessorilta, on tuloksena nytkähdys kuvassa, tai jos syötteen lukemisesta tai päivityksestä huolehtiva säie joutuu odottamaan, on tuloksena viive syötteen vaikutuksen välittymisessä pelin tilaan, eli pelin interaktiivisuus kärsii. Suositeltavaa onkin, ettei suoritettavassa ohjelmassa olisi useampia säikeitä, kuin suoritusjärjestelmässä on prosessoriyksikköjä. Eli pelien tapauksessa on parempi, jos pelisovellus käyttää maksimissaan prosessoriytimien määrän verran säikeitä. Esimerkiksi yksisäikeinen pelisilmukkamalli toimii käytännössä paremmin neliytimisen prosessorin kanssa, kuin neljäsäikeinen malli yksiytimisen prosessorin kanssa.

Tämä työ esittelee pelisilmukka-arkkitehtuurista version sekä kaksi, että neljäytimiselle mobiililaitteelle. Neljä prosessoriydintä sisältäviä mobiililaitteita on tällä hetkellä vain hintatason yläpäässä, joten neljäytimisen arkkitehtuurin käyttöön on syytä suhtautua varauksin. Neljän tehtäväsäikeen toimivuus kaksiytimisellä laitteella ei ole optimaalisinta, johtuen edellä esittelystä ongelmasta säikeitten ja prosessoriytimien määrän suhteen. Kahta säiettä käyttävä malli sen sijaan toimii optimaalisesti myös neljä prosessoriydintä sisältävässä mobiililaitteessa, vaikka se jättää hyödyntämättä neliytimisessä mobiililaitteessa kaksi prosessoriydintä.

Koska mobiililaitesovellusten tulee pyrkiä säästämään akun virtaa, tulee niiden myös pyrkiä mahdollisimman vähäiseen mobiililaitteen prosessoriytimien kuormittamiseen [PFW11]. Ehdotettu malli noudattaa tätä sääntöä ja se pyrkii suorittamaan pelisilmukan tehtävät ilman ylimääräistä prosessorin kuormittamista. Peleissä on usein tapana synkronoida pelin päivitystehtävät piirtämisen kanssa, jolloin pelin olioiden liikkuminen on näytöllä sulavaa. Jos pelin päivitystehtävät tehdään harvemmin, kuin pelin tila piirretään, voi piirtotehtävä ehtiä piirtämään pelin tietyn tilan useampaan kertaan ennen seuraavaa päivitystä. Jos pelin tila päivitetään esimerkiksi 30 kertaa sekunnissa (Hz), ja piirtäminen tapahtuu 60 kertaa sekunnissa, piirretään jokainen pelin tila keskimäärin kaksi kertaa. Mielekkäämpää olisi piirtää vain pelin tilassa tapahtuvia muutoksia. Toisaalta, jos pelin tila päivitetään 30 Hz nopeudella, ei 30 Hz piirtonopeus tarjoa sulavinta mahdollista ulosantia. Toisaalta päivitystehtävien suorittaminen yhtä usein piirtämisen kanssa on usein yliampuvaa. Ratkaisu pelin päivitystehtävien suorittamiseen

piirtonopeutta harvemmin, ja silti sulavan piirtämisen säilyttäminen, on interpolointi, jolla voidaan ennustaa pelin tilassa tapahtuvia muutoksia. Sen avulla piirtämistehtävä voi arvioida piirrettävien olioiden sen hetkisen sijainnin, ja piirtää ne sinne. Käytännössä pelin päivitystehtäviä ei siis tarvitse suorittaa yhtä tiheällä nopeudella, kuin piirtämistä.

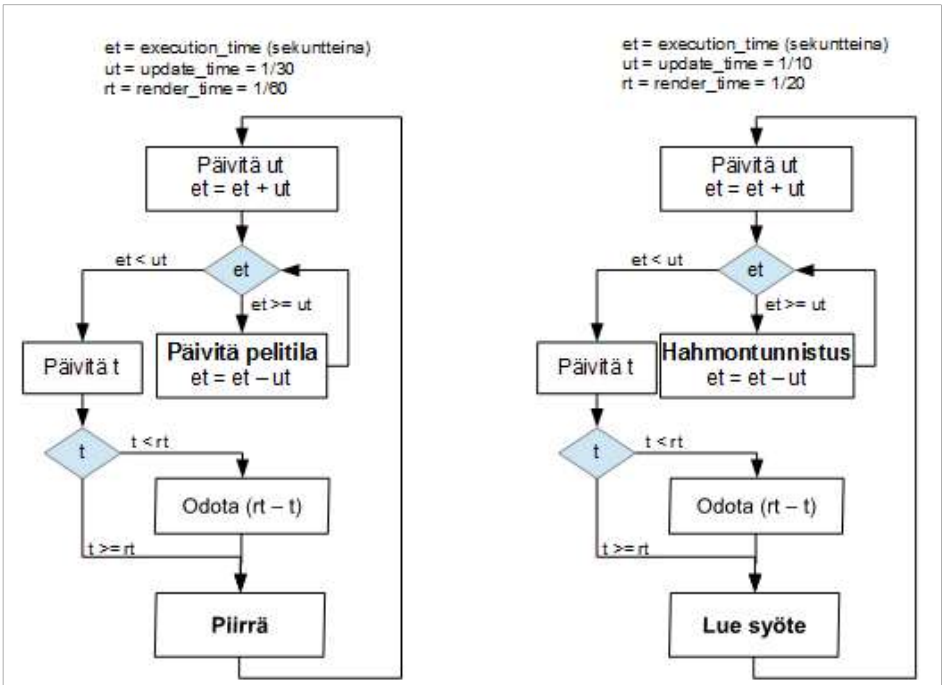
Ehdotetun mallin kaksiytimisille mobiililaitteille tarkoitetun version pelisilmukka-arkkitehtuuri on ottanut vaikutteita sekä Dickinsonin [Dic01] että Laken ja Gabbin tutkimuksista [LaG05]. Dickinson esitteli artikkelissaan yksisäikeisen vakiotaajuudellisen pelisilmukkamallin arkkitehtuurin, jonka erikoisuus on eri tehtävien suoritus yksilöllisillä vakionopeuksilla yhden säikeen sisällä. Tästä mallista poimittiin idea eri tehtävien suorittamiseen samassa säikeessä eri nopeuksilla, ja ehdotetussa mallissa sijoitetaan samaan säikeeseen joko päivitys- ja piirtotehtävät, tai hahmontunnistus- ja syötteenkäsittelytehtävät. Lake ja Gabb taas esittelivät artikkelissaan kuinka pelisilmukan eri tehtävät voidaan suorittaa asynkronisesti rinnakkain, ja tätä samaa mallia noudatetaan ehdotetun pelisilmukkamallin arkkitehtuurissa, kun nämä kaksi tehtäväsäiettä pyörivät toisistaan riippumatta [Kuva 28].



Kuva 28: Kaksiytimisille mobiililaitteille ehdotetun mallin pelisilmukka-arkkitehtuuri.

Kaksiytimisen version pelisilmukka-arkkitehtuurin molempia säikeitä toistetaan yksilöllisellä vakionopeudella, jonka lisäksi molempien säikeiden sisällä on toinen tehtävä, jolla on myös yksilöllinen vakioitu toistonopeus. Molempien säikeiden kaksi tehtävää ovat kytköksissä toisiinsa, ja niiden yksilölliset toistonopeudet tulee synkronoida yhteen, jotta molemmat tehtävät toistuisivat mahdollisimman tarkasti halutulla suoritustiheydellä.

Ehdotetussa mallissa piirtämistehtävän suoritustiheys on vakioitu 60 Hz:iin, ja pelin päivitystehtävän toistonopeus on synkronoitu piirtämistehtävän toistonopeuden kanssa ja vakioitu puoleen sen nopeudesta, eli 30 Hz:iin. Tällöin päivitystehtävä suoritetaan optimitilanteessa joka toisella kierroksella, kun piirtäminen tapahtuu 1/60 sekunnin



Kuva 29: Ehdotetun mallin kaksiytimisen version pelisilmukka-arkkitehtuuri avattuna.

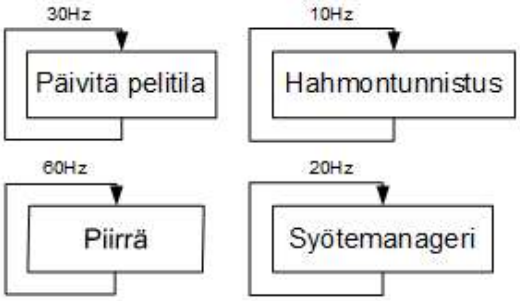
välein ja päivitys tapahtuu 1/30 sekunnin välein. Toisen säikeen syötteenkäsittelytehtävän nopeus on asetettu 20 Hz:iin, ja hahmontunnistustehtävän toistonopeus on asetettu puoleen siitä, 10 Hz:iin.

Ehdotetussa mallissa kaikkien tehtävien toistonopeutta tarkkaillaan aikamuuttujan avulla [Kuva 29]. Jos tehtävän säie suoriutuu niin nopeasti tehtävistään, että se toistuisi säädettyä vakionopeutta nopeammin, ei kyseistä tehtävää suoriteta, vaan siirrytään suorittamaan seuraavaa tehtävää tai pistetään säie odotustilaan liitteen 5 esittämällä tavalla. Kun säie menee odotustilaan, se ei kuormita prosessoria, jolloin mahdolliset muut odottavat tehtävät saavat suoritusaikaa, tai prosessori pääse lepotilaan, jolloin virtaa kuluu vähemmän.

Piirretyn animaation sulavuuden parantamisessa käytetty interpolointiarvo lasketaan ehdotetun mallin pelisilmukassa päivitystehtävän *execution_time*- ja *update_time*-muuttujien avulla [Liite 5]. Näiden muuttujien suhde kertoo käytännössä kuinka kaukana seuraavan päivitystehtävän suoritus on suhteessa edelliseen päivitystehtävän suoritukseen. *execution_time*-muuttujan arvo vähennetään *update_time*-muuttujaa pienemmäksi aina ennen interpolointiarvon laskemista, joten interpolointiarvo ei voi kasvaa suuremmaksi kuin 1.

Ehdotetun mallin versio neliytimisille mobiililaitteille pitää sisällään pelisilmukka-

arkkitehtuurin, joka on suunniteltu noudattaen Mönkkösen esittelemää asynkroniseen funktiorinnakkaisuuteen pohjautuvaa tehtävien rinnakkaistamista [Mön06]. Tässä pelisilmukkamallissa jokainen tehtävä suoritetaan omassa säikeessä omalla vakionopeudella [Kuva 30]. Säikeiden vakioidut toistonopeudet ovat samat, kuin kaksiytimisen version pelisilmukka-arkkitehtuurin tehtävien toistonopeudet. Koska jokainen säie pyörii itsenäisesti huolehtien vain yhden tehtävän suorittamisesta halutulla vakionopeudella, on toteutus yksinkertaisempi [Liite 6]. Kuten kaksiytimisten mobiililaitteiden versiossa, myös tässä versiossa käytetään apuna toistonopeuden vakioinnissa aikamuuttujaa, jonka avulla säie laitetaan tarvittaessa odottamaan ettei sitä toisteta liian nopeasti. Teoriassa säiettä toistetaan tietyllä vakionopeudella, mutta käytännössä toistonopeus voi heilahdella vähän, joten päivitystehtävässä voidaan käyttää apuna toistoajan aikamuuttujaa. Tämän aikamuuttujan avulla päivitystoimet voidaan suhteuttaa yksisäikeisen sitomattoman mallin mukaisesti [Luku 3.2].



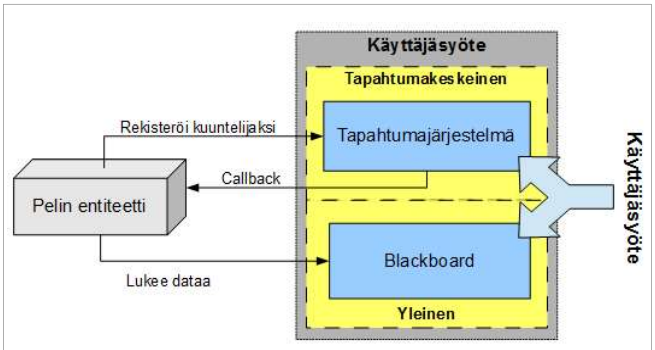
Kuva 30: Ehdotetun mallin mukainen pelisilmukkamalli neliytimisille mobiililaitteille.

6.3 Käyttäjäsyötteen käsittely ehdotetussa mallissa

Suunnittelu ehdotetulle mallille otti lähtökohdaksi Miluzzon ja kumppaneiden työssään [MLF08] esittelemän arkkitehtuurin, jossa jokaista sensoria vastaa oma asiakassäie joka tallentaa kyseisen sensorin puskeman datan ohjelman komponenttien jakamaan muistitilaan. Vaikka Miluzzo ja kumppanit eivät mainitse sitä, niin heidän kuvaama datan tallennus jaettuun muistiin tapahtuu luvussa 4.3 esitellyn liitutaulumallin mukaisesti. Liitutaulumallissa käytetään datan jakamiseen komponenttien välillä kaikkien saatavilla olevaa datavarastoa. Tämä tallennustila pitää sisällään uusimman version saatavilla olevasta datasta, joita eri komponentit tarvittaessa lukevat tai päivittävät. Liitutaulumallin mukaisen datavaraston sopivuudesta sensoridatan tallentamiseen antaa tukea Raennon ja kumppaneiden [ROP05] työ, jossa he päätyivät toteuttamaan sensoridatan jakamisen uudelleen käyttämällä liitutaulumallia. He toteavat kyseisen mallin soveltuvan julkaise-kirjaudu -mallia paremmin sensoridatan jakamiseen jolla he ensin toteuttivat sensoridatan jakamisen.

Liitutaulumallin mukaisen datavaraston toteutus täytyy tehdä siten että sinne voi tallentaa kaiken tyyppisiä muuttujia, koska etukäteen on mahdoton tietää minkä tyyppisiä muuttujia sinne tullaan tallentamaan. C++ -kielellä liitutaulumallin mukaisen datavaraston toteutuksessa voidaan käyttää niin sanottua *template*-luokkaa, joka mahdollistaa yleisten muuttujatyyppien käsittelyn [Liite 7].

Liitutaulumalli ei kuitenkaan poista tarvetta tapahtumakäsittelylle. Käyttäjäsyöte, tai siitä tulkattu abstraktio, voi laukaista sellaisen tarpeen pelin olion päivitykselle joka voisi jäädä pelioliolta huomaamatta. Tällöin liitutaulumallin sijaan vaikutusyhteys kannattaa luoda tapahtuman avulla. Ehdotettu malli käyttää käyttäjäsyötteestä havaittujen eleiden vaikutuksen välittämisessä tapahtumajärjestelmää ja sensoreilta tulevan datan tallennukseen liitutaulumallin mukaista julkista datavarastoa. Käyttäjäsyötteen välitys pelin olioille tapahtuu ehdotetussa mallissa siis joko tapahtumajärjestelmän kautta tai liitutaulumallisen julkisen datavaraston kautta [Kuva 31].



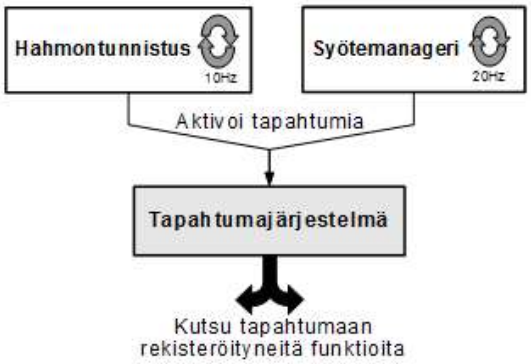
Kuva 31: Käyttäjäsyötteen käsittelyn kaksijakoisuus ehdotetussa mallissa.

Ehdotettu malli ottaa huomioon pelimoottoreiden yleisesti käyttämän tavan erottaa pelin olioiden lukema käyttäjäsyöte käyttäjäsyötelaitteesta abstraktoimalla syöte [Gre09 s. 350]. Pelimoottoreiden yleisesti tarjoama tapahtumajärjestelmä on yleinen tapa pelin varmistaa pelin olion reagointi käyttäjäsyötteessä tapahtuvan muutoksen [Gre09 s. 773], ja samanlaista järjestelmää käytetään myös ehdotetussa arkkitehtuurimallissa GPS-vastaanottimelta ja sensoreilta tulevalle syötedatalle.

6.3.1 Tapahtumajärjestelmän käyttö ehdotetussa mallissa

Ehdotettu arkkitehtuurimalli pitää sisällään myös tapahtumajärjestelmän ja tuen aktivoitaville tapahtumille. Ehdotetussa mallissa on kaksi komponenttia, jotka voivat aktivoida tapahtumia tapahtumajärjestelmän kautta. Tapahtumajärjestelmää käyttävät komponentit ovat hahmontunnistus ja syötemanageri [Kuva 32]. Nämä tehtävät suoritetaan vakioidulla toistonopeudella joko samassa säikeessä tai erillisissä säikeissä

riippuen suoritusjärjestelmän prosessoriytimien määrästä. Ne voivat tarpeen mukaan kutsua tapahtumajärjestelmäluokassa määriteltyä tapahtumaa, joka puolestaan aktivoi siihen rekisteröityneet takaisinkutsufunkti-ot.



Kuva 32: Tapahtumajärjestelmän käyttö ehdotetussa mallissa.

Syötemanageri voi aktivoida tapahtumia luetun käyttäjäsyötteen perusteella suoraan, tai hahmontunnistus voi tunnistaa eleen ja aktivoida tarvittavan tapahtuman. Tapahtuman tarkka aktivointityyli riippuu tapahtumajärjestelmän toteutuksesta. Tämän työn yhteydessä tehdyssä ehdotetun mallin toteutuksessa eleet aktivoidaan joko kutsumalla tapahtumajärjestelmän *ExecuteSignal*-funktiota halutuilla signaalien *id*-arvoilla

```
GameManager::GetEventSystem()->ExecuteSignal(1,2,3);
```

tai kutsumalla tapahtumajärjestelmän tapahtumafunktiota suoraan.

```
GameManager::GetEventSystem()->SigA();
```

Tapahtumajärjestelmän määrittelemiin tapahtumiin, joita Boost-kirjaston yhteydessä nimitetään myös signaaleiksi [Boo14], täytyy linkittää ne takaisinkutsufunktiot joita tapahtuman aktivoinnin yhteydessä kutsutaan. Ehdotetun mallin tapauksessa takaisinkutsufunktioiden linkitys tapahtumaan voidaan toteuttaa Boost-kirjaston avulla yhdellä komentorivillä.

```
SigA.connect(boost::bind(&Player::Jump, ptr_player));
```

Yllä oleva komento linkittää *Player*-luokkaan kuuluvan *ptr_player*-osoittimen osoittaman olion *Jump*-funktion *SigA*-tapahtumaan/signaaliin. Linkityksen jälkeen *SigA*-tapahtuman aktivointi, eli *SigA*-funktion kutsuminen, aiheuttaa kaikkien siihen linkitettyjen funktioiden kutsumisen.

6.3.2 GPS- ja sensoridatan käsittely mallissa

Ehdotetussa mallissa käytetään yhtä, syötemanageriksi kutsuttua, tehtävää keräämään data GPS-vastaanottimelta ja sensoreilta. Syötemanageritehtävä on asetettu toistumaan luvussa 3.3 käsiteltyjen tietojen perusteella 20 kertaa sekunnissa (20 Hz).

Syötemanageri tallentaa kerätyn datan edellä esiteltyyn julkiseen datavarastoon. Tämän lisäksi se voi myös käsitellä kerättyä dataa ja tallentaa käsittelyn tuloksena tulleet käyttäjäsyöteabstraktiot samaan julkiseen datavarastoon.

Ehdotetussa mallissa käytetään liitutaulumallin mukaista julkista datavarastoa komponenttien väliseen datan jakamiseen. Ehdotetussa mallissa tätä datavarastoa käytetään tallennuspaikkana tuoreimmalle GPS- ja sensoridatalle, abstraktoiduille käyttäjäsyötemuuttujille sekä hahmontunnistuksen tarvitsemien sensoreilla datajonojen tallentamiseen. Hahmontunnistuksen tehtäväsäiettä suoritetaan virransäästösyystä hitaammalla toistonopeudella kuin syötteen keräävä syötemanageritehtävää, tästä syystä hahmontunnistuksen käyttämiltä sensoreilta luettava data täytyy tallentaa jonoon, jotta hahmontunnistustehtävältä ei jää huomioimatta yksikään luettu syöte. Hahmontunnistustehtävän suorituksen yhteydessä se lukee ja tyhjentää datavarastossa olevan syötejonon.

Julkisen varaston käyttö vaatii tiedon halutun datan tunnisteesta. Jos pelin komponentti haluaa lukea esimerkiksi tuoreimman tiedon kiihtyvyysanturin ilmoittamalle kiihtyvyydelle x-akselin suhteen, täytyy sen tietää kyseisen datan tunniste. Kutsumalla julkisen varaston get-funktiota parametrina tunniste, saadaan paluuarvona haluttu data. Käytettäessä ehdotettua mallia, täytyy datavarastossa käytetyt tunnisteet sopia ja tiedottaa etukäteen.

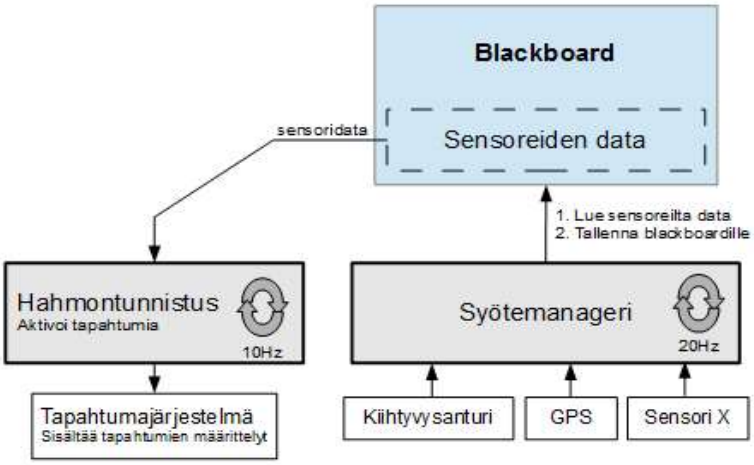
6.3.3 Hahmontunnistus mallissa

GPS- ja sensoridataan pohjautuvaan käyttäjäsyötteeseen kuuluu myös hahmontunnistus, jolla kerätyn datan avulla voidaan tunnistaa pelaajan tekemiä eleitä. Hahmontunnistustehtävä ja syötteen keräämistehtävä linkittyvät tiukasti yhteen. Ehdotetussa mallissa nämä tehtävät tapahtuvat joko yhdessä, tai kahdessa säikeessä, riippuen mobiililaitteen prosessoriytimien määrästä. Ehdotetun mallin versio kaksiytimisille mobiililaitteille suorittaa hahmontunnistuksen ja syötteen lukemisen saman säikeen sisällä, kun neljäytimisille mobiililaitteille ehdotettu malli suorittaa ne omissa säikeissään [Kuva 33, kuva 34].

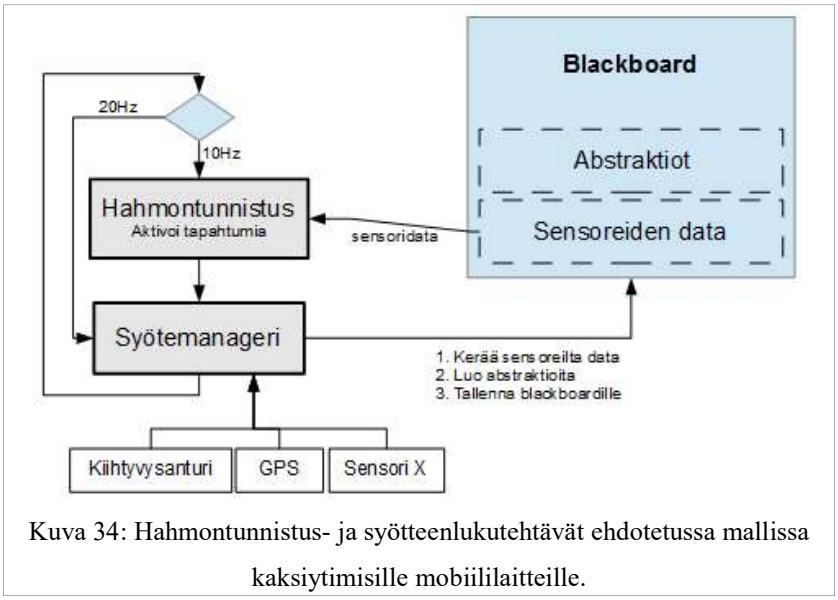
Kaksiytimisten mobiililaitteiden mallissa syötelaitteiden lukemis- ja hahmontunnistustehtävät on sijoitettu samaan säikeeseen luvussa 3.2.2 esitellyn yksisäikeisen vakiotaajuudellisen pelisilmukkamallin mukaisesti. Ehdotetussa mallissa

hahmontunnistus pyörii 10 Hz nopeudella ja syötteen lukeminen 20 Hz nopeudella. Toisin sanoen syötemanageri lukee syötelaitteilta arvon 20 kertaa sekunnissa, ja hahmontunnistus tarkistaa 10 kertaa sekunnissa onko syötejonoon ilmestynyt tunnistettava hahmo.

Hahmontunnistus- ja syötteen lukemisesta huolehtiva komponentti eivät ole tietoisia toisistaan, eivätkä ne kommunikoi suoraan keskenään, vaan tieto niiden välillä liikkuu edellä esitetyn liitutaulumallin mukaisen datavaraston kautta. Hahmontunnistus ei tosin syötä tietoja tunnistettavista hahmoista sinne, vaan eleen tunnistuksen yhteydessä se aktivoi eleeseen liitetyn tapahtuman tapahtumajärjestelmästä. Tapahtumajärjestelmä kutsuu kyseiseen tapahtumaan rekisteröityneiden olioiden takaisinkutsufunktioita luvussa 4.1 esitellyllä tavalla.



Kuva 33: Hahmontunnistus- ja syötteenlukutehtävät ehdotetussa mallissa neliytimisille mobiililaitteille.



Kuva 34: Hahmontunnistus- ja syötteenlukutehtävät ehdotetussa mallissa kaksiytimisille mobiililaitteille.

Hahmontunnistuksen toteutus ehdotetussa mallissa

Ehdotettu arkkitehtuurimalli ei ota kantaa hahmontunnistuksessa käytettävään toteutukseen ja se mahdollistaa erilaiset hahmontunnistuskomponentin toteutukset. Ehdotetun mallin toimivuutta hahmontunnistuksessa testattiin toteuttamalla kätkettyyn Markovin malliin (HMM) pohjautuva hahmontunnistuskomponentti.

HMM tarjoaa keinon eleiden tallentamiseen liikemalleina, mutta sen lisäksi hahmontunnistusta varten tarvitaan myös niin sanottu luokittelija, jonka avulla voidaan selvittää mikä tallennetuista malleista on lähimpänä annettua toista mallia.

Ehdotetun malli on riippumaton hahmontunnistuksen toteutuksesta ja sen testaamiseen riittää yksinkertaisesti toteutettu hahmontunnistuskomponentti. Yksinkertaisimman keinon hahmontunnistuksessa käytettävän luokittelijan toteutukseen tarjoaa eteenpäinsyöttöalgoritmi sillä sitä käytetään myös HMM:n toteutuksessa eikä se näin tuota lisätyötä. Luokittelija vertaa syötteenä annettua sarjaa jokaiseen tietokantaan tallennettuun HMM-malliin eteenpäinsyöttöalgoritmin avulla, ja valitsee niistä todennäköisimmän vastaavuuden

6.4 Ehdotetun mallin kokonaisarkkitehtuuri

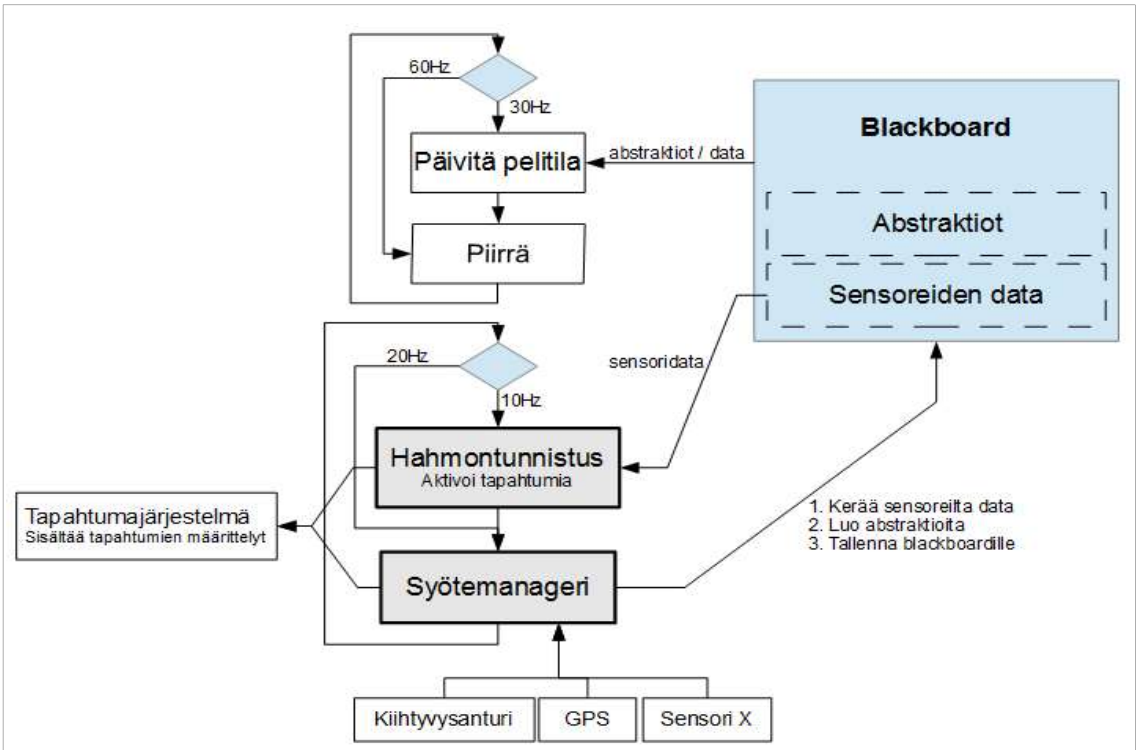
Ehdotettu malli sitoo yhteen edellä kuvatut ratkaisut pääpelisilmukalle, datan jakamiselle, syötteen lukemiselle ja hahmontunnistamiselle. Eri komponentit eivät ole

tietoisia toisistaan, vaan luvussa 4.3 esitellyn liitutaulumallin mukainen julkinen datavarasto sitoo ne epäsuorasti yhteen, ja ne kommunikoivat sen kautta toisilleen. Koska nykyaikaiset mobiililaitteet pitävät sisällään kaksi tai neljä prosessoriydintä, esitellään ehdotetusta arkkitehtuurimallista versio niille molemmille.

6.4.1 Ehdotetun mallin arkkitehtuuri kaksiytimisille mobiililaitteille

Ehdotetun mallin versio kaksiytimisille mobiililaitteille sisältää prosessoriytimien mukaisesti kaksi tehtäväsäiettä. Toiseen säikeeseen yhdistetään päivitys- ja piirtämistehtävät ja toiseen säikeeseen hahmontunnistus- ja syötemanageritehtävät [Kuva 35]. Ehdotetun mallin tehtävien toistonopeus on vakioitu luvussa 3.3 käsiteltyjen perusteiden pohjalta. Päivitystehtävä on synkronoitu piirtämistehtävään nähden siten, että optimitilanteessa pelitila päivitetään joka toista piirtokertaa ennen. Hahmontunnistuksen ja syötteen lukemisen suoritussuhde on samanlainen, eli optimitilanteessa hahmontunnistus suoritetaan joka toista syötemanageritehtävän suoritusta kohti.

Syötemanageri kerää GPS-vastaanottimelta ja sensoreilta datan, mahdollisesti käsittelee sitä, ja tallentaa niiltä saadun datan, sekä mahdollisesti siitä prosessoidut käyttäjäsyöteabstraktiot, datavarastoon. Sensoridatan tallennus datavarastoon tapahtuu jonomuodossa, eli syötemanageri työntää lukemansa sensoridatan sensoria vastaavaan datajonoon. Hahmontunnistustehtävä poimii suorituksensa yhteydessä datavarastosta tunnistukseen käytetyn sensorin syötejonon, ja tallentaa sen omaan muistiavaruuteensa jonka jälkeen se tyhjentää kyseisen jonon datavarastosta. Hahmontunnistus pitää tallessa omassa muistiavaruudessa kolmen sekunnin verran sensoridataa, ja se tarkistaa suorituksensa yhteydessä muodostaako sen muistiavaruuteen kerääntynyt syöte tunnistettavaa hahmomallia. Jos se havaitsee tunnistettavan hahmon, aktivoi se hahmoa vastaavan tapahtuman tapahtumajärjestelmän kautta ja tyhjentää muistiavaruutensa.

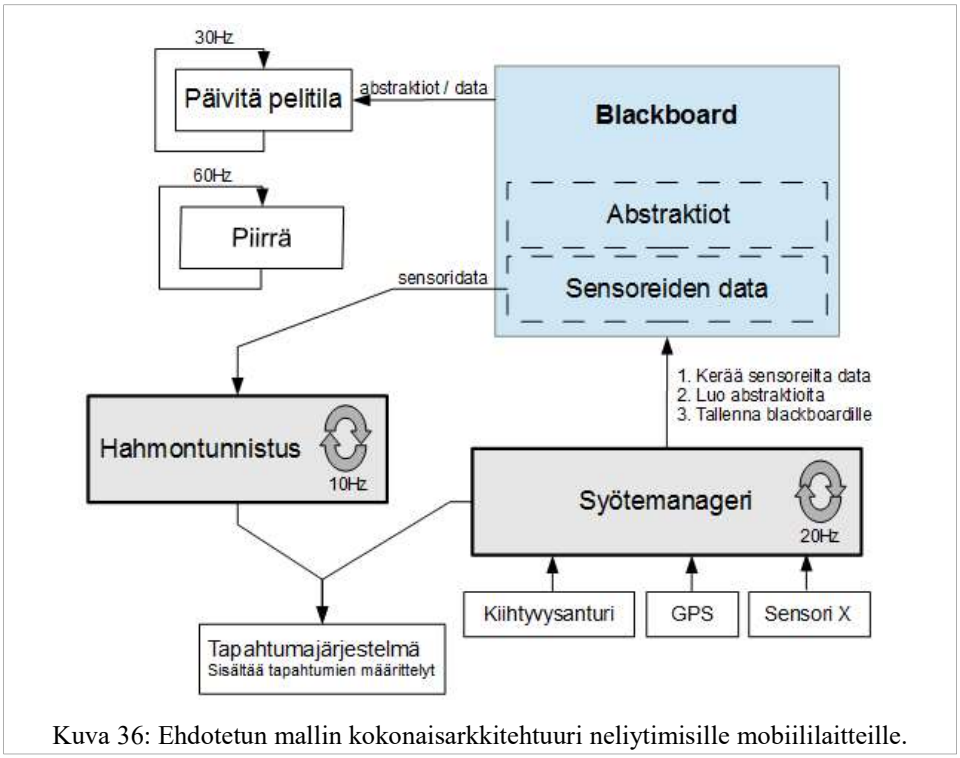


Kuva 35: Ehdotetun mallin kokonaisarkkitehtuuri kaksiytimisille mobiililaitteille.

Päivitystehtävä lukee jokaisen suorituksensa yhteydessä julkisesta datavarastosta tarvitsemansa syötedatan, ja päivittää pelin olioiden tilan. Piirtämistehtävä piirtää pelin oliot näytölle niiden tilatiedon perusteella.

6.4.2 Ehdotetun mallin arkkitehtuuri neliytimiselle mobiililaitteelle

Neliytimisille mobiililaitteille ehdotettu malli on muuten samankaltainen kaksiytimisten mobiililaitteiden mallin kanssa, mutta tässä mallissa jokainen mallin tehtävä erotetaan omaan itsenäiseen säikeeseen [Kuva 36]. Jokaiselle mallin säikeelle asetetaan luvun 3.3 tietojen pohjalta optimaalinen toistonopeus minimoimaan mobiililaitteen virrankulutus. Mallin tehtävät suoritetaan siis omissa säikeissään, ja niiden suoritusnopeudet voidaan säätää yksilöllisesti. Kuten kaksiytimisten mobiililaitteiden mallissa, myös tässä mallissa hahmontunnistus- ja syötemanageritehtävät ovat ainoat komponentit, jotka aktivoivat tapahtumajärjestelmäkomponentin kautta tehtäviä, ja data jaetaan komponenttien välillä liitutaulumallin mukaisesti toteutetun julkisen datavaraston kautta.



Kuva 36: Ehdotetun mallin kokonaisarkkitehtuuri neliytimisille mobiililaitteille.

7 Ehdotetun mallin toimivuus mobiilipeleissä

Tässä luvussa tarkastellaan kuinka toimiva ehdotettu malli on käytännössä mobiilipelien kontekstissa. Ensin tutkitaan kuinka pelit yleensä käsittelevät käyttäjäsyötettä, ja millaisia vaatimuksia ne asettavat käyttäjäsyötteelle, ja sitten tarkastellaan kuinka ehdotettu malli toimii niissä. Tämän jälkeen tutkitaan millaisin eri tavoin mobiilipelit käyttävät GPS- ja sensoridataa tällä hetkellä, sekä millaisia mahdollisia käyttökohteita ne tarjoavat peleille, ja täyttääkö ehdotettu malli niiden mahdolliset tarpeet. Lopuksi tarkastellaan ehdotetun mallin skaalautuvuutta mahdollisten uusien sensoreiden ja prosessoriytimien määrän kasvun kannalta.

7.1 Pelien asettamat yleiset vaatimukset käyttäjäsyötteelle

Yleinen vaadittu ominaisuus käyttäjäsyötteeltä on sen reaaliaikaisuus, eli vaatimus sen vaikutuksen välittymiseksi pelin tilaan reaaliajassa [Gre09 s.17]. Ehdotettu malli mahdollistaa tämän sekä liitutaulumallin mukaisen datavaraston kautta, jolloin pelin olio lukee syötetyypin arvon sieltä päivityksen yhteydessä, että tapahtumakäsittelyn kautta, jolloin syötemanageri aktivoi tapahtuman joka kutsuu siihen rekisteröityneiden olioiden takaisinkutsufunktioita.

Pelkkä syötteen reaaliaikaisuuden tukeminen ei kuitenkaan riitä, sillä erityyppiset pelit

voivat vaati käyttäjäsyötteen käsittelyltä erilaisia ominaisuuksia. Esimerkiksi tappelupelit vaativat käyttäjäsyötejärjestelmän, joka pystyy havaitsemaan käyttäjäsyötteestä malleja, kuten näppäinpainallusten yhdistelmiä (*chords*) [Gre09 s.17]. Ehdotettu malli ei estä tällaista käyttäjäsyötteen käsittelyä, sillä se ei ota kantaa söytemanagerin lopulliseen toteutukseen, ja oikeanlaisella toteutuksella se pystyy tekemään myös syöteyhdistelmien havaitsemisen. Tämän lisäksi käyttäjäsyötteestä huolehtiva rajapinta tulisi suunnitella siten, että se tukee erilaisia sekä olemassaolevia, että tulevaisuudessa mahdollisesti tulevia, käyttäjäsyötelaitteita (*Human Interface Devices*, HID) [Gre09 s.350], eli sen tulee skaalautua HID-laitteiden osalta. Mobiililaitteissa HID-laitteina toimivat erilaiset sensorit ja mobiililaitteesta mahdollisesti löytyvä GPS. Ehdotettu malli on suunniteltu komponenttipohjaiseksi, ja siihen pystytään lisäämään mukaan tulevaisuudessa mobiililaitteisiin tulevat sensorit.

Erilaiset mobiililaiteet voivat tarjota erilaisia HID-laitteita pelin käyttöön. Käyttäjäsyötteestä huolehtivan komponentin olisi hyvä pystyä linkittämään pelin ohjaamiseen tarvittavat syötekanavat tarjolla oleviin HID-laitteisiin suoritusaikaisesti. HID-laitteiden ja pelitapahtumien muokattavuus voidaan saavuttaa erottamalla HID-laitteet käyttäjäsyötteen varsinaisesta käsittelystä erillisellä määrittelytasolla. Tällaisen määrittelytason tulisi tarjota joitakin, ellei kaikkia seuraavista syötteenkäsittelytavoista

1. analogisen syötelaitteen kuolleen alueen (*dead zone*) määrittely,
2. analogisen signaalin suodatus,
3. tapahtumien havaitseminen, kuten painalluksen aloitus ja lopetus,
4. syöteyhdistelmien (*chords*) ja -sarjojen havaitseminen,
5. hahmontunnistus,
6. useiden HID-laitteiden tuki usealle pelaajalle,
7. eri laitejärjestelmien HID-laitteiden tukeminen,
8. HID-laitteelta tulevan syötteen suoritusaikainen linkkaus pelin tapahtumiin,
9. kontekstivaikutteinen syöte ja
10. Mahdollisuus väliaikaisesti poistaa käytöstä jotain syötekanavia [Gre09 s. 350].

Ensimmäinen kohta viittaa analogisten syötelaitteiden ongelmaan. HID-laitteet toimivat joko digitaalisesti tai analogisesti. Kun digitaaliset syötelaitteet tarjoavat vain tiedon siitä, onko kyseinen syöte kytkettynä päälle vai ei, niin analogiset syötelaitteet tarjoavat datan tietyltä alueelta, esimerkiksi arvon arvojen –32,768 ja 32,767 väliltä. Pelaaja voi haluta tällaisen syötelaitteen tuottavan myös nolladataa, mutta käytännössä tällainen syötelaite tuottaa helposti pieniarvoista syötettä vaikka pelaaja ei sitä haluaisi. Ratkaisu tähän on määritellä kuollut alue, jolloin saadaan analoginen syötelaite tuottamaan nolladataa, jos sen tuottama syöte on tarpeeksi lähellä nollatilaa [Gre09 s. 372]. Ehdotetussa mallissa kuollutta aluetta voidaan käyttää syötemanagerin yhteydessä, sen tehdessä GPS- tai sensoridatasta sellaisia abstraktoituja muuttujia, joiden arvon määrittelyssä sitä halutaan käyttää. Tällaisia abstrakteja muuttujia on esimerkiksi analogisesta kiihtyvyysanturidatasta määritettävät muuttujat, joiden perusteella voidaan liikutella jotain pelin oliota. Näin laitteen kallistelulla liikutettu pelin olio pysyy paikallaan, jos laite on melkein vaakatasossa sekä x- että y-akselien suhteen. Ilman kuollutta aluetta, pelin olio liikkuisi luultavasti hiukan johonkin suuntaan jatkuvasti, sillä käsin laitetta on hankala saada täyin vaakatasoon ja kiihtyvyysanturidatassa on mukana pientä häiriötä.

Listan toinen kohta liittyy analogisessa signaalissa yleisesti esiintyvään häiriöön (*noise*), joka voi aiheuttaa ei-halutun vaikutuksen pelissä [Gre09 s.373]. Tämän vuoksi monet pelit suodattavat syötelaitteelta tulevan analogisen datan. Yksi mahdollinen suodatuskeino on käyttää alipäästösuodinta (*low-pass filter*), jota käytetään useissa peleissä tasaamaan sisääntulevasta datasta suuret signaaliarvot pois. Ehdotettu malli ei ota kantaa käytettyihin suotimiin, mutta työhön liittyneessä ehdotetun mallin mukaisessa pelimoottorin toteutuksessa käytettiin alipäästösuodinta kiihtyvyysanturilta tulevaan dataan.

```
double InputManager::LowPassFilter(double from, double to, float weight)
{
	double filtered = from * (1.0f - weight) + to * weight;
	return filtered;
}
```

Ehdotetun mallin toteutuksessa alipäästösuotimelle annetaan kolme parametria; nykyinen arvo, sensorilta saatu arvo ja painoarvo. Painoarvo määrittää sen, mikä on nykyisen ja sensorilta saadun arvon suhde. Ehdotettu malli käyttää oletuksena painoarvoa 0,4, joka on todettu kokeilemalla toimivaksi kiihtyvyysanturille.

Kolmas kohta liittyy pelien tapahtumakeskeisyyteen. Alemman tason HID-rajapinta tarjoaa pelille yleensä HID-laitteiden sen hetkisen tilatiedon [Gre09 s.375]. Ehdotettu malli tekee tämän käyttämällä liitutaulumallin mukaista datavarastoa datan jakamiseen. Pelit ovat kuitenkin tapahtumakeskeisiä, ja niiden oliot ovat ensisijaisesti kiinnostuneita HID-laitteiden tarjoamassa syötteessä tapahtuvista muutoksista, eivätkä niiden tilan tarkastamisesta. Käyttäjäsyötteen käsittelyn tulisi siis tukea tapahtumakäsittelyä, jottei jokaisen olion tarvitse kysellä HID-laitteiden tilaa, vaan niille ilmoitetaan niitä kiinnostavista tapahtumista. Pelimoottorit pitävät yleensä sisällään tapahtumajärjestelmän, joka hoitaa muutoksista ilmoittamisen. Ehdotettu malli tukee tapahtumajärjestelmää, ja siinä syötemanageri aktivoi tapahtumajärjestelmän kautta syötettä vastaavat tapahtumat.

Listan viides kohta, hahmontunnistus, viittaa käyttäjäsyötteestä havaittuihin eleisiin. Ehdotettu malli tukee hahmontunnistusta, jota käsitellään yleisesti luvussa 5.3 ja sen mahdollista toteutusta ehdotetussa mallin mukaisessa pelimoottorissa luvussa 6.4.

Listan kuudes kohta ei sovellu hyvin mobiililaitteisiin. Se käsittelee useiden pelaajien HID-laitteiden käyttöä samassa suoritusjärjestelmässä. Mobiililaitteet ovat kuitenkin henkilökohtaisia tietokoneita ja suhteellisen pienikokoisia, eikä niiden äärellä pelaa yleensä kuin yksi pelaaja, joten mobiililaitteiden pelimoottorin ei tarvitse välttämättä tukea usean samanaikaisen pelaajan käyttäjäsyötelaitteita.

Listan seitsemäs, kahdeksas, yhdeksäs ja kymmenes kohta liittyvät HID-laitetason abstraktointiin, joka on aiheellinen myös mobiililaitteille. Seitsemännen kohdan suosituksen mukaisesti ehdotettu malli ei estä eri mobiililaitteiden ja eri mobiililaitekäyttöjärjestelmien käyttämistä, sillä siinä toteutetaan HID-laitteiden abstraktointi pelitapahtumista kahden määrittelytason avulla [Gre09 s.383]. HID-laitteiden käsittely tapahtuu alemmalla tasolla, ja pelikohtainen toteutus määritellään omalla tasolla, jolloin eri laitejärjestelmien erilaiset HID-laitteet voidaan linkata suoritusaikaisesti haluttuihin pelitapahtumiin [Kuva 37].

Kuva 37: Käyttäjäsyötteen abstraktointi HID-laitteesta, muokattu lähteestä [Gre09 s. 385].

Kuten monet pelit [Gre09 s.385], myös ehdotettu malli tarjoaa pelaajalle kahdeksannen kohdan mukaisesti mahdollisuuden valita kontrollit

joilla peliä ohjataan. Yksi yleinen tapaus, jossa pelaaja voi vaikuttaa pelikontrolleihin, on vertikaalisen liikkeen säätäminen, sillä toiset voivat haluta alaspäin suuntautuvan liikkeen kääntävän katselukulmaa ylöspäin ja toiset alaspäin. Kun HID-laitteen vaikutus peliin on abstraktoitu, voidaan ylös- ja alaskontrollit vaihtaa päikseen HID-laitteen ja pelivaikutuksen välillä.

Listan yhdeksännen ja kymmenennen kohdan toteutus on käytännössä mahdollista jos pelimoottorin syötteenkäsittely mahdollistaa syötteen abstraktoinnin. Yhdeksännen kohdan kontekstisidonnaisuus viittaa siihen että syötteen vaikutus peliin voidaan säätää pelin tilan mukaisesti. Esimerkiksi tietty syöte voi pelihahmon katsoessa ovea aukaista oven, kun se toisessa tilanteessa poimii katsotun esineen. Kohdan 10 mukaisesti taas joskus voi tulla tilanne, jossa jokin käyttäjäsyötekanava halutaan poistaa käytöstä, esimerkiksi pelin sisäisen videon pyöriessä.

Ehdotettu malli mahdollistaa syötteen abstraktoinnin ja siten myös kohtien 9 ja 10 toteutuksen. Ehdotettu malli tarjoaa mahdollisuuden syötteeseen mahdollisesti liittettyjen muuttujien suoritusaikaiseen lukemiseen, jolloin syötteen abstraktio voi syötteen lisäksi ottaa huomioon halutut pelin tilasta kertovat muuttujat. Samalla tyylillä voidaan toteuttaa syötteen poistaminen käytöstä, asettamalle syötteen abstraktiolle tila, jolloin se ei vaikuta pelin tilaan.

7.2 Sensoridatan käyttö pelien tarvitsemilla tavoilla

Mobiililaitteet luokitellaan henkilökohtaisiksi laitteiksi, ja niiden sensoreiden keräämä data on myös henkilökohtaisesti. Datan henkilökohtaisuuden vuoksi on yleisesti käyty keskustelua siitä, mitä sensoreiden keräämää dataa voidaan prosessoida pilvipalvelimilla, koska pilvipalveluiden käyttö voi rikkoa datan luottamuksellisuutta [LML10]. Ehdotettu malli ei ota kantaa pilvipalveluiden käyttöön sensoridatan käsittelyssä, eikä sisällä, muttei myöskään estä, niiden käyttöä.

Sensoreiden jatkuva käyttö kuluttaa mobiililaitteen akkua, ja voi siten haitata mobiililaitteen yleistä käyttökokemusta [LML10]. Ehdotetussa mallissa sensoreita luetaan nopeudella, joka on minimivaatimus hahmontunnistukselle. Jos käytetty laiteympäristö mahdollistaa, voi ohjelmistokehittäjä säätää sensoreiden näytteenottotaajuuden syötemanagerin 20 Hz nopeutta vastaavaksi. Tätä työtä varten Windows Phone 8 -laitealustalle toteutetussa ehdotetun mallin mukaisessa soveluksessa

sensoreiden näytteenantotiheys voitiin säätää halutuksi, joten kiihtyvyysanturin ja muiden sensoreiden näyttöönottotiheys voitiin säätää samaksi syötemanageritehtävän toistonopeuden kanssa, jolloin ne päivittivät antamansa datan 50 ms välein (20 Hz).

Ehdotettu malli vakioi kaikkien sensoreiden lukemisen syötemanageritehtävän suoritustiheyden mukaan. Jatkuvaan ympäristön tarkkailuun suunnitelluissa sovelluksissa jotkin sensorit voivat kuitenkin hyötyä tiheästä lukutiheydestä, kun toisille riittää harvempi lukeminen [LML10]. Toinen jatkuvaan ympäristön tarkkailuun perustuva puoli on se, että joidenkin sensoreiden datankäsittely on enemmän prosessoria kuormittavaa, kuin toisten. Nämä edelliset seikat huomioonottaen ehdotettua mallia voitaisiin parantaa sellaisia pelejä varten, jotka käyttävät useita sensoreita yhtäaikaisesti. Ehdotetun mallin syötemanageri voitaisiin hajottaa osiin niin, että jokainen sensori saisi oman säikeensä, joka lukee sensoria tarvittavalla tiheydellä ja antaa sensorilta saatavalle datalle tarvittavan verran prosessointiaikaa. Tällöin jokainen sensori luetaan optimitiheydellä ja jokaisen sensorin datan käsittely voidaan käsitellä yksilöllisesti, jolloin raskaudellaan vaihtelevat tehtävät pystyttäisiin suorittamaan optimaalisesti, kenties jopa pilvipalveluissa. Tällainen tehtävien hajauttaminen kuitenkin mutkistaisi ehdotettua mallia, eikä ole selvää saataisiinko siitä selvää hyötyä. Tämän työn pohjalta voitaisiin tehdä jatkotutkimus, jossa vertailtaisiin kyseisenlaisia malleja toisiinsa.

Sensoreilta saatavasta datasta pitää pystyä tulkkaamaan mitä pelaaja haluaa viestittää pelille [LML10]. Ehdotettu malli sisältää tällaisen sensoridatan tulkkauksen. Malli, jolla sensoridata siinä käsitellään, voidaan jakaa kahteen tyyliluokkaan käsittelytavan, eli tulkkaustavan, perusteella. Joko sensoridataa käytetään sellaisenaan (korkeintaan suodatettuna), tai siitä tunnistetaan eleitä hahmontunnistuksen avulla. Sensoridatan tulkkaukseen liittyvä hahmontunnistus vaatii tiedon keräämisen sensorilta vakioidulla nopeudella [LML10]. Ehdotettu malli tekee sen vakioidulla 20 Hz nopeudella.

7.3 GPS-datan käyttö pelien tarvitsemilla tavoilla

GPS-dataa luetaan kahdella eri tavalla, joko jatkuvasti tai kertaluontoisesti. Pidettäessä GPS:ää jatkuvasti aktiivisena kulutetaan samalla runsaasti virtaa akusta, sillä GPS on mobiililaitteissa yksi eniten virtaa syövistä tekniikoista. Virrankulutuksen kannalta parempi tapa käyttää GPS:ää on kertaluontoinen sijainnin kysyminen, jolloin GPS-rajapinta palauttaa yleensä sijainnin pituus- ja leveysasteena (*longitude* & *lattitude*).

Ehdotetussa mallissa GPS-datan kerääminen ja käsittely hoidetaan syötemanageritehtävässä. Jos pelille riittää kertaluontoiset sijaintitiedon kyselyt, esimerkiksi kerran sekunnissa, niin tällöin ehdotetussa mallissa oleva syötemanageri luo asynkronisen sijaintitiedon kyselyn noin joka kahdennellakymmenennellä suorituskerrallaan (20 Hz → 1 Hz) sekä käsittelee ja tallentaa sen ehdotetun mallin julkiseen datavarastoon. Ehdotetussa mallissa syötemanageri suoritetaan 20 Hz nopeudella, mutta sijaintitiedon harvempi päivitystiheys onnistuu luvussa 3.2.2 esitellyn vakiotaajuudellisen yksisäikeisen pelisilmukkamallin mukaisesti. Tällöin sijaintitiedon päivityskutsu tehdään GPS-rajapinnalle vain joka n:nellä syötemanagerin toteutuskierroksella.

Jos peli tarvitsee jatkuvan GPS-datan keräämisen, hoidetaan sijaintitiedon välitys pelin olioille tapahtumapohjaisesti, eikä ehdotetun mallin syötemanageri-komponentin välityksellä. Tällöin ne pelin oliot, jotka käyttävät jatkuvasti päivittyvää sijaintitietoa, rekisteröivät kiinnostuksensa GPS-datasta järjestelmän GPS-rajapinnalle luvussa 4 esitellyn valvojamallin mukaisesti. Tämän jälkeen GPS-rajapinta ilmoittaa jokaisesta sijaintipäivityksestä suoraan rekisteröityneille olioille.

7.4 Ehdotetun mallin skaalautuvuus

Ehdotettu malli on suunniteltu modulaariseksi, jolloin siihen on helppo lisätä eri tehtäviä hoitavia komponentteja. Lisäksi siinä data jaetaan liitutaulumallia noudattaen, jolloin mahdollisesti lisättävät komponentit saavat tarvitsevansa datan yhteisestä datavarastosta. Kohta joka ehdotetusta mallista tulee muuttaa, jos käytössä on useita sensoreita ja muita syötelaitteita samanaikaisesti, on syötemanagerin jakaminen useampaan osaan. Yhden syötemanagerisäikeen käyttämiseen monen syötelaitteen kanssa sisältyy riski, etenkin, kun ei ole tiedossa millaisia syötelaitteita tulevaisuuden mobiililaitteet tulevat sisältämään. Riskinä on se, että syötemanageritehtävän yhden kierroksen suorittamiseen kuluu yhteensä yli 50ms, jolloin syötemanageri ei ehdi pyöriä tavoitenopeudella 20 Hz. Mahdollinen ratkaisu tähän on luoda syötemanagerista oma säie jokaista tarkkailtavaa syötelaitetta kohden, tai toinen tapa olisi luoda kaksi tai useampi säie lukemaan ja käsittelemään ryhmää syötelaitteita. Tällä tyylillä kaikki syötelaitteet eivät olisi yhden säikeen suoritusnopeuden varassa. Toisaalta säikeiden kokonaismäärä kasvaisi, jolloin riski siihen, että jokin säie joutuu odottamaan

suoritusaikaa kriittisellä hetkellä.

8 Yhteenveto

Mobiililaitteiden yleisyys on kasvanut räjähdysmäisesti ja samalla niistä on muodostunut merkittävä pelialusta. Tämä työ yhdistää ne mobiililaitteiden pelimoottorin suunnitteluun kuuluvat osatekijät, jotka tarvitaan mobiililaitteille ominaisen tekniikan hyödyntämiseen pelin käyttäjäsyötteenä. Nykyiset mobiililaitteet sisältävät yleensä GPS-vastaanottimen ja kiihtyvyysanturin sekä mahdollisesti muita käyttäjäsyötteenä käytettäviä sensoreita, kuten gyroskoopin ja magnetometrin. Aikaisempia tutkimuksia näiden käyttäjäsyötelaitteiden yhdistämisestä mobiilipelimoottoreihin ei löytynyt, joten siinä mielessä tätä työtä voi pitää uraa uurtavana.

Aluksi tässä työssä tutustuttiin aiheeseen liittyviin käsitteisiin, pelimoottorin osiin ja käyttäjäsyötteen käsittelyyn. Tämän jälkeen esiteltiin tähän tietoon pohjautuvan ja tässä työssä suunniteltu arkkitehtuurimallin GPS- ja sensoridatan käyttämiselle mobiililaitteiden peleissä. Ehdotetun mallin avulla mobiilipelit pystyvät sitomaan GPS- ja sensoridatan tehokkaasti ja skaalautuvasti osaksi pelimoottoria. Koska nykyiset mobiililaitteet käyttävät pääsääntöisesti kaksi- tai neljäytimisiä prosessoreita, työssä suunniteltiin arkkitehtuuri molemman tyylisille laitteille.

GPS- ja sensoridatan tuominen raudalta pelisovelluksen käyttöön sisältää syötteen keräämisen, syötteen käsittelyn, mahdollisen hahmontunnistuksen ja datan välittämisen pelin olioille. Ehdotetussa mallissa syötemanageri kerää aluksi data GPS-vastaanottimelta ja eri sensoreilta. Tämän jälkeen se mahdollisesti käsittelee syötedataa ja tallentaa sen ja siitä luodut abstraktiot liitutaulumallin mukaiseen jaettuun datavarastoon. Rinnalla pyörivä hahmontunnistussäie lukee datavarastosta tarvittavat syötedatat ja havaitsee siitä mahdolliset hahmomallit. Havaitessaan tutun hahmokuvion, eleen, se kutsuu tapahtumajärjestelmästä eleeseen kuuluvaa tapahtumaa, joka aktivoi siihen rekisteröityneet funktiot. Pelin päivitystehtävä lukee jaettua datavarastoa, ja päivittää pelin olioiden tilan luetun datan mukaisesti. Pelin piirtämistehtävä piirtää pelin oliot niiden tilan mukaisesti.

Ehdotetun mallin toimivuus todennettiin toteuttamalla sen mukainen pelimoottori Windows Phone 8 -laitealustalle. Lisäksi, koska tavoitteena oli luoda arkkitehtuurimalli, jota voidaan käyttää mobiililaitteiden pelien kehityksessä, arvioitiin ehdotetun mallin toimivuutta mobiilipelimoottorien toteutuksessa tutkimalla pelimoottorien käyttäjäsyötteen käsittelyltä odotettuja yleisiä ominaisuuksia. Arvioinnin yhteydessä ehdotetun mallin todettiin noudattavan näitä vaatimuksia ja mahdollistavan niiden mukainen käyttäjäsyötteen käsittelyn. Toinen arviointikohde GPS- ja sensoridatan käyttö peleissä, ja ehdotetun mallin todettiin mahdollistavan GPS- ja sensoridatan keräämisen ja käsittelyn pelien mahdollisesti tarvitsemilla tavoilla, kuten eri nopeudella päivittyvien syötedatojen ja hahmontunnistuksen osalta. Lisäksi todettiin, ettei ehdotettu malli ota kantaa pilvipalveluiden käyttöön GPS- ja sensoridatan käsittelyssä, mutta ei myöskään estä niiden käyttöä. Mobiililaitteet asettavat sovelluksille vaatimuksen virrankulutuksen minimoinnista, ja ehdotetun mallin todettiin mahdollistavan virrankulutuksen minimoinnin mahdollistamalla syötteen keräämisen sensoreilta halutulla tiheydellä. Ehdotetun mallin todettiin myös skaalautuvan mobiililaitteiden prosessoriytimien ja sensorien määrän kasvaessa.

Tämä työ luo yleisen arkkitehtuuripohjan oikeaoppiseen GPS- ja sensoridatan käyttöön mobiilipelien pelimoottoriarkkitehtuurissa. Aihealuetta voitaisiin tutkia lisää ainakin kolmella tavalla. Ensinnäkään työssä ei oteta kantaa kaikista nykyaikaisista mobiililaitteista löytyvän grafiikankäsittelyprosessorin (GPU:n) hyödyntämiseen. Kyseistä prosessoria voitaisiin hyödyntää piirtämisen lisäksi muihin tehtäviin, kuten Joselli et al työssään toteaa [JZC08], ja kenties myös sensoreiden datan käsittelyyn. Toiseksi tämän työn aihealueeseen liittyy myös laitevalmistaja Applen uusimman älypuhelimen iPhone 5S:n esittelemä sensoreille erikoistettu erillinen prosessori, joten kyseisen laitearkkitehtuurin hyödyntämisen tutkiminen osuisi tarkasti tämän työn aihealueeseen. Kolmanneksi lisätutkimusta voitaisiin tehdä mobiilipelimoottorien skaalautuvuudesta prosessoriytimien, ja etenkin sensoreiden mukana, sillä tässä työssä ehdotetun mallin mukainen yksi syötteenkäsittelysäie ei välttämättä enää riitä, kun yhtäaikaisesti käytettävien sensoreiden määrä kasvaa suureksi.

Lähteet

AAF09 Amstutz, R., Amft, O., French, B., Smailagic, A., Siewiorek, D., Troster, G., Performance analysis of an HMM-based gesture recognition using a wristwatch device. *Proceedings of the International Conference on Computational Science and* Engineering, Vancouver, British Columbia, Canada, elokuu 2009, sivut 303-309.

AlI13 Anjum, A, Ilyas, M. U., Activity recognition using smartphone sensors. *Proceedings of the International Conference on Consumer Communications and Networking Conference*, Las Vegas, Nevada, USA, tammikuu 2013, sivut 914-919.

APH09 Abdesslem, F. B., Phillips, A., Henderson, T., Less is more: Energy-Efficient Mobile Sensing with SenseLess. *Proceedings on the 1st ACM workshop on networking, systems, and applications for mobile handhelds*, Barcelona, Espanja, elokuu 2009, sivut 61-62.

BaI04 Bao, L., Intille, S. S., Activity recognition from user-annotated acceleration data. *Proceedings of the 2nd International conference on pervasive computing*, Wien, Itävalta, huhtikuu 2004, sivut 1-17.

BKV97 Bouten, C.V., Koekkoek, K.T., Verduin, M., Kodde, R., Janssen, J.D., A triaxial accelerometer and portable data processing unit for the assessment of daily physical activity. *IEEE Transactions on Biomedical Engineering*, 44,3(1997), sivut 136–47.

Boo14 Boost C++ Libraries, 2014. http://www.boost.org/. [28.4.2014]

Cam14 Cambridge Dictionaries Online, 2014. http://dictionary.cambridge.org/dictionary/british, [18.2.2014].

CBC10 Cuervoy, E., Balasubramanianz, A., Cho, D., Wolmanx, A., Saroiux, S., Chandrax, R., Bahlx, P., MAUI: Making Smartphones Last Longer with Code Offload. *Proceedings of 8th international conference on Mobile systems, applications, and services*, San Francisco, California, USA, kesäkuu 2010, sivut 49-62.

CLL14 CLLocation Class Reference, iOS Developer Library, 2014. https://developer.apple.com/library/ios/documentation/CoreLocation/Reference/CLLocation_Class/CLLocation/CLLocation.html. [28.4.2014]

DiC13 Dietrich, B., Chakraborty, S., DEMO: Power Management using Game State Detection on Android Smartphones. *Proceedings of the 11th international conference on Mobile systems, applications, and services*, Taipei, Taiwan, kesäkuu 2013, sivut 493-494.

Dic01 Dickinson, P., Instant Replay: Building a Game Engine with Reproducible Behavior, 2001. http://www.gamasutra.com/view/feature/131466/instant_replay_building_a_game_.php, [21.1.2014].

Fli09 Flintham, M., *Supporting mobile mixed-reality experiences*. Tohtorin väitökirja, University of Nottingham. 2009.

LaG05 Lake, A., Gabb, H., Threading 3D Game Engine Basics. 2005. http://www.gamasutra.com/view/feature/2463/threading_3d_game_engine_basics.php, [28.1.2014]

Geo14 Geocoordinate class, Windows Dev Center. 2014. http://msdn.microsoft.com/en-us/library/windows/apps/windows.devices.geolocation. geocoordinate, [28.4.2014]

Gre09 Gregory, J., *Game Engine Architecture*. A. K. Peters, Ltd, 2009.

EAA08 Elmezain, M., Al-Hamadi, A., Appenrodt, J., Michaelis, B., A Hidden Markov Model-Based Continuous Gesture Recognition System for Hand Motion Trajectory. *Proceedings of the 19th International conference on pattern recognition*, Tampa, Florida, USA, joulukuu 2008, sivut 1-4.

IlK02 Ilmonen, T., Kontkanen, J., Software Architecture for Multimodal User Input – FLUID. *Proceedings of 7th International Workshop on User Interfaces for All*, Pariisi, Ranska, lokakuu 2002, sivut 319-338.

Imp14 Improving memory and power use for XNA games for Windows Phone. http://msdn.microsoft.com/en-us/library/windowsphone/develop/hh855082(v=vs.105).aspx [6.4.2014]

JoC09 Joselli, M., Clua, E., gRmobile: A Framework for Touch and Accelerometer Gesture Recognition for Mobile Games. *Proceedings of 8th Brazilian Symposium on Games and Digital Entertainment*, Rio de Janeiro, lokakuu 2009, sivut 141-150.

JSZ12 Joselli, M., Silva, J., R., Zamith, M., Clua, E., Pelegrino, M., Mendonça, E., Soluri, E., An Architecture for Game Interaction using Mobile. *Proceedings of the IEEE International Games Innovation Conference,* Rochester, New York, syyskuu 2012, Sivut 1-5.

JZC08 Joselli, M., Zamith, M., Clua, E., Montenegro, A., Leal-Toledo, R., Conci, A., Pagliosa, P., Valente, L., Feijo, B., An adaptative game loop architecture with automatic distribution of tasks between cpu and gpu. *Computers in Entertainment*, 7,4(2009), artikkelin numero 50.

JZS12 Joselli, M., Zamith, M., Silva, J., R., Valente, L., Clua, E., An Architecture for Mobile Games with Cloud Computing Module. *Proceedings of 9th Brazilian Symposium on Games and Digital Entertainment*, Salvador, Brasilia, marraskuu 2012, artikkeli 2.

KPT10 Kiili, K., Perttula, A., Tuomi, P., Suominen, M., Lindstedt, A.. Designing mobile multiplayer exergames for physical education. *Proceedings on International Conference on Mobile learning*, Porto, Portugali, helmikuu 2010, sivut 141-148.

LeJ02 Lewis, M., Jacobson, J., Game engines in scientific research. *Communications of the ACM*, 45,1 (2002), sivut 27-31.

LML10 Lane, N. D., Miluzzo, E., Lu, H., Peebles, D., Choudhury, T., Campbell, A. T.. A Survey of mobile phone sensing. *Communications Magazine, IEEE*, 48, 9 (2010), sivut 140-150.

Loc14 Location. Android Developers. 2014. http://developer.android.com/reference/android/location/Location.html, [28.4.2014]

Mac14 Accelerometer for your robot. Machinegrid. http://www.machinegrid. com/2008/12/accelerometers-for-your-robot-adxl202-and-memsic-2125/, [28.4.2014]

MKK04 Mäntyjärvi, J., Kela, J., Korpipää, P. and Kallio S. Enabling fast and effortless customisation in accelerometer based gesture interaction. *Proceedings of the 3rd International Conference on Mobile and Ubiquitous Multimedia*, College Park, Maryland, syyskuu 2004, sivut 25–31.

MLF08 Miluzzo, E., Lane, N. D., Fodor, K., Peterson, R., Lu, H., Musolesi, M., Eisenman, S. B., Zheng, X., Campbell, A. T., Sensing meets mobile social networks: the design, implementation and evaluation of the CenceMe application. *Proceedings of 6th ACM conference on Embedded Network Sensor Systems*, Raleigh, Pohjois Carolina, marraskuu 2008, sivut 337-350.

Män01 Mäntylä, VM., Discrete hidden Markov models with application to isolated user-dependent hand gesture recognition. *Valtion teknillinen tutkimuskeskus (VTT),* lisensiaatintyö, 2001.

Mön06 Mönkkönen, Multithreaded game engine architectures. 2006. http:// www.gamasutra.com/view/feature/130247/multithreaded_game_engine_ .php, [21.1.2010]

NKC12 Naqvi, N. Z., Kumar, A., Chauhan, A., Sahni, K., Step Counting Using Smartphone-Based Accelerometer. *International Journal on Computer Science and Engineering*. 4,5(2012), sivut 675-681.

Oxf14 Oxford dictionary. 2014. http://www.oxforddictionaries.com/, [18.2.2014].

PCM14 PC Magazine. Definition of smartphone. 2014. http://www.pcmag. com/encyclopedia/term/51537/smartphone, [18.2.2014]

PFW11 Perrucci, G.P., Fitzek, F.H.P, Widmer, J., Survey on Energy Consumption Entities on the Smartphone Platform. *Proceedings on the 73rd Vehicular Technology Conference*, Budabest, Unkari, 2011, sivut 1-6.

Pyl05 Pylvänäinen, T., Accelerometer Based Gesture Recognition Using Continuous HMMs. *Proceedings of 2nd Iberian conference on Pattern Recognition and Image Analysis*, Estoril, Portugali, *2005*, sivut 639-646.

ROP05 Raento, M., Oulasvirta, A., Petit, R., Toivonen, H.. ContextPhone: A Prototyping Platform for Context-Aware Mobile Applications. *IEEE, Pervasive Computing,* 4,2 (2005), sivut 51-59.

SPH08 Schlömer, T., Poppinga, B., Henze, N., Boll, S., Gesture recognition with a Wii controller. *Proceedings of 2nd International Conference of Tangible and embedded interaction*, Bonn, Saksa, 2008, sivut 11-14.

Tec12 TechTerms.com, Smartphone, 2014. http://www.techterms.com/definition/smartphone, [18.2.2014]

VCF05 Valente, L., Conci, A., Feijó, B., Real time game loop models for single-player computer games. *Proceedings of 2nd Brazilian Symposium on Games and Digital Entertainment*, 2005, sivut 89–99.

Wik14a Mobile operating system. Wikipedia. 2014. http://en.wikipedia.org/wiki/Mobile_operating_system, [19.2.2014]

Wik14b GPS. Wikipedia. 2014. http://fi.wikipedia.org/wiki/GPS, [24.4.2014]

Wik14c Sensor. Wikipedia. 2014. http://en.wikipedia.org/wiki/Sensor, [24.4.2014]

Wik14c Blackboard system. Wikipedia. 2014. http://en.wikipedia.org/wiki/Blackboard_system, [14.2.2014]

Wik14d Wikipedia, interpolation. 2014. http://en.wikipedia.org/wiki/Interpolation, 3.3.2014

Wit14 Witters, K., deWiTTERS Game Loop, 2014. http://www.koonsolo.com/news/dewitters-gameloop/, 3.3.2014

WeL12 Weiss, G. M., Lockhart, J. W., A comparison of alternative client/server architectures for ubiquitous mobile sensor-based applications.*Proceedings of the 14th ACM International Conference on Ubiquitous Computing*, Pittsburgh, Pennsylvania, syyskuu 2012, sivut 721-724.

WLA09 Wang, Y., Lin, J., Annavaram, M., Jacobson, Q. A., Hong, J., Krishnamachari, B., A Framework of Energy Efficient Mobile Sensing for Automatic User State Recognition. *Proceedings of the 7th*

international conference on Mobile systems, applications, and services, Krakova, Puola, kesäkuu 2009, sivut 179-192.

Liite 1. Yleiset ominaisuudet tämän hetken mobiililaitteissa

Älypuhelimet (Lähde: Counterpoint, helmikuu 2014. Hinta; hintaseuranta.com & aliexpress.com, 26.4.2014)

Maailmanlaajuinen myynti helmikuussa 2014 + vertailulaite Windows Phone 8 OS:llä

VALMISTAJA	MALLI	HINTA (€)	CPU	YTIMIÄ	GPU	MUISTI	OS
Apple	iPhone 5S	650	1.3 GHz	2	PowerVR G6430	1 GB	iOS
Apple	iPhone 5C	570	1.3 GHz Swift	2	PowerVR SGX 543MP3	1 GB	iOS
Samsung	Galaxy S4	490	1.9 GHz Krait 300	4	Adreno 320	2 GB	Android
Samsung	Note 3	550	2.3 GHz Krait 400	4	Adreno 330	3 GB	Android
Apple	iPhone 4S	330	1 GHz Cortex-A9	2	PowerVR SGX543MP2	512MB	iOS
Samsung	Galaxy S4 Mini	270	1.7 GHz Krait	2	Adreno 305	1,5GB	Android
Xiaomi	Hongmi Red Rice	130	1.5 GHz Cortex-A7	4	PowerVR SGX544MP2	1 GB	Android
Samsung	Galaxy S3	300	1 GHz Cortex-A9	2	Mali-400	1 GB	Android
Samsung	Galaxy S3 Mini	170	1 GHz Cortex-A9	2	Mali-400	1 GB	Android
Xiaomi	MI 3	315	2.3 GHz Krait 400	4	Adreno 330	2 GB	Android
+							
Nokia	Lumia 520	100	1.0 GHz Krait	2	Qualcomm Adreno 305	512MB	WP 8

Taulutietokoneet (Lähde: Gartner, helmikuu 2014. Hinta; hintaseuranta.com & aliexpress.com, 26.4.2014)

Suositut taulutietokoneet maailmanlaajuisesti eniten myyneiltä valmistajilta 2013 + vertailulaitteet Windows RT/8 OS:llä

VALMISTAJA	MALLI		CPU	YTIMIÄ	GPU	MUISTI	OS
Apple	iPad 2	340	1 GHz Cortex-A9	2	PowerVR SGX543MP2	512 MB	iOS
Apple	iPad Mini 2	300	1.3 GHz Cyclone	2	PowerVR G6430	1GB	iOS
Samsung	Galaxy Tab 3	250	1.6GHz Intel Z2560	2	Mali-400MP	1,5 GB	Android
Samsung	Galaxy Note 10.1	600	1.4 GHz Cortex-A9	4	Mali-400MP4	2 GB	Android
Asus	Memo Tab HD 7"	150	1.2 GHz Cortex-A7	4	PowerVR SGX544	1 GB	Android
Asus	VivoTab RT TF600T	300	1.3 GHz Cortex-A9	4	ULP GeForce	2 GB	Android
Amazon	Kindle Fire HDX	200	2.2 GHz Krait 400	4	Adreno 330	2 GB	Android
Lenovo	Yoga Tablet 10	250	1.2 GHz Cortex-A7	4	PowerVR SGX544	1 GB	Android
+							
Microsoft	Surface 2	400	1.7GHz NVIDIA Tegra 4	4	ULP GeForce	2 GB	Windows RT
Microsoft	Surface Pro 2	900	1.6-2.6GHz Intel i5 4200U	2	Intel HD Graphics 4400	4 GB	Windows 8
Nokia	Lumia 2520	600	2.2GHz Snapdragon 800	4	Adreno 330	2 GB	Windows RT

Liite 2. GPS ja sensorit tämän hetken mobiililaitteissa

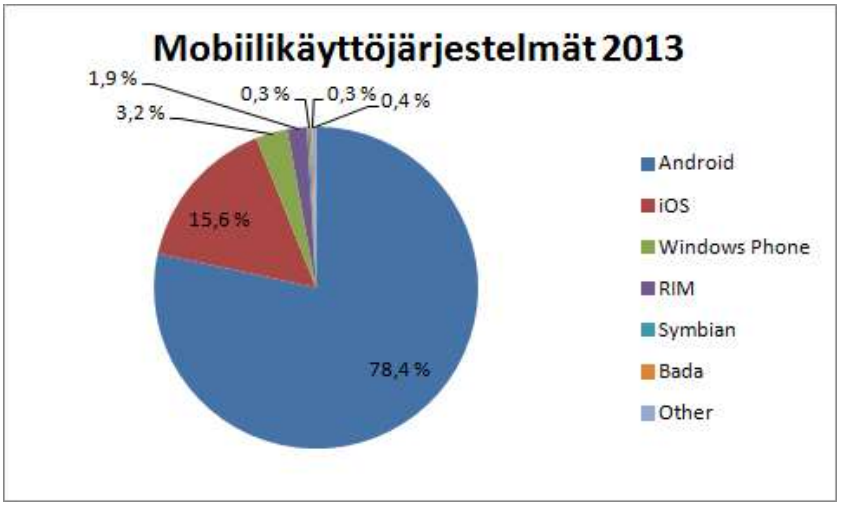
Älypuhelimet (Lähde: Counterpoint, helmikuu 2014. Hinta; hintaseuranta.com & aliexpress.com, 26.4.2014)
Maailmanlaajuinen myynti helmikuussa 2014 + vertailulaite Windows Phone 8 OS:llä

				SENSORIT						
VALMISTAJA	**MALLI**	**HINTA (€)**	**GPS**	**kiihtyvyys**	**magnetometri**	**gyro**	**läheisyys**	**valo**	**sormenjälki**	**barometri**
Apple	iPhone 5S	650	x	x	x	x	x	x	x	
Apple	iPhone 5C	570	x	x	x	x	x	x		
Samsung	Galaxy S4	490	x	x	x	x	x	x		x
Samsung	Note 3	550	x	x	x	x	x	x		x
Apple	iPhone 4S	330	x	x	x	x	x	x		
Samsung	Galaxy S4 Mini	270	x	x	x	x	x	x		
Xiaomi	Hongmi Red Rice	130	x	x	x	x	x			
Samsung	Galaxy S3	300	x	x	x	x	x	x		x
Samsung	Galaxy S3 Mini	170	x	x	x	x	x	x		
Xiaomi	MI 3	315	x	x	x	x	x			x
+										
Nokia	Lumia 520	100	x	x			x	x		

Taulutietokoneet (Lähde: Gartner, helmikuu 2014. Hinta; hintaseuranta.com & aliexpress.com, 26.4.2014))
Suositut taulutietokoneet maailmanlaajuisesti eniten myyneiltä valmistajilta 2013 + vertailulaitteet Windows RT/8 OS:llä

				SENSORIT						
VALMISTAJA	**MALLI**	**HINTA (€)**	**GPS**	**kiihtyvyys**	**magnetometri**	**gyro**	**läheisyys**	**valo**	**sormenjälki**	**barometri**
Apple	iPad 2	340	x	x		x		x		
Apple	iPad Mini 2	300	x	x		x		x		
Samsung	Galaxy Tab 3	250	x	x	x					
Samsung	Galaxy Note 10.1	600	x	x	x	x		x		
Asus	Memo Tab HD 7"	150	x	x	x					
Asus	VivoTab RT TF600T	300	x	x	x	x		x		
Amazon	Kindle Fire HDX	200	x	x		x		x		
Lenovo	Yoga Tablet 10	250	x	x	x			x		
+										
Microsoft	Surface 2	400	x	x	x	x	x	x		
Microsoft	Surface Pro 2	900	x	x	x	x	x	x		
Nokia	Lumia 2520	600	x	x	x	x	x	x		

Liite 3: Mobiililaitekäyttöjärjestelmien yleisyys 2013



Muokattu lähteestä [Wik14]

Liite 4. Yksisäikeinen vakiotaajuudellinen sitomaton pelisilmukka

```
// Tämä esimerkki näyttää, kuinka
// ns. yksisäikeinen sitomaton pelisilmukkamalli voidaan toteuttaa C++ -kielellä.
// Piirtämistehtävä toteutetaan pelisilmukan jokaisella kierroksella,
// MUTTA syötteen käsittely ja pelin päivitystoimet tehdään vain ~10 ms välein.

int execution_time; // apumuuttuja, jonka avulla päivitystehtävät tehdään 10ms välein
const int update_time = 10; //haluttu toistoväli päivitystehtäville (millisekuntteina) (10ms välein = 100 Hz)
bool playing = true;

while(playing) // jokainen silmukan toisto päivittää pelitilannetta yhden ruudun (frame) eteenpäin
{
	// Laske kulunut aika edellisestä ajan päivityksestä
	timer->Update();

	// Lisätään execution_time-muuttujaan kulunut aika
	execution_time += timer->Delta;

	// Jos execution_time-muuttujan arvo kasvoi > 10ms suorita päivitystehtävät
	// muuten hyppää päivitysten ohi ja tee pelkästään piirtämistehtävä
	while(execution_time > update_time) // suoritetaan jos nykyinen suoritusaika >10(ms)
	{
		// lue & tallenna pelaajan syöte
		HandlePlayerInputs();

		// päivitä pelin tila
		UpdatePlayer(update_time);
		UpdatePhysics(update_time);
		UpdateAI(update_time);
		// jne…

		// vähennä nykyisestä suoritusajasta 10(ms)
		execution_time -= update_time;
		// päivitykset while-loopissa
		// → jos execution_time on yhä > 10 (eli päivitykset jäljessä haluttua 100Hz nopeutta)
		//  toistetaan päivitysten tekeminen uudelleen.
	}

	// Piirrä
	RenderFrame();
}
```

Liite 5. Ehdotetun mallin mukainen pelisilmukan toteutus kaksiytimisille mobiililaitteille

```
// Ehdotetun mallin mukainen toteutus neliytimisille mobiililaitteille
// Lähdekoodi ehdotetun mallin mukaisesta toteutuksesta.
// Kehitysympäristönä Windows Phone 8
//
// Silmukassa suoritetaan pelin tilan päivitys 30 Hz toistonopeudella
// ja pelin tilan piirtäminen 60 Hz toistonopeudella

float render_time = 1/60;       // Haluttu suoritustiheys pelin piirtämiselle = 60 kertaa sekunnissa
float update_time = 1/30;       // Haluttu suoritustiheys pelin päivittämiselle = 30 kertaa sekunnissa
float execution_time = 0.0f;   // Apumuuttuja päivitystehtävän suoritustiheyden pitämiseen 30 Hz

while (running)
{
        // Prosessoi järjestelmän tapahtumat
        CoreWindow::GetForCurrentThread()->Dispatcher->
                ProcessEvents(CoreProcessEventsOption::ProcessAllIfPresent);

        // Laske kulunut aika edellisestä timer_update -päivityksestä
        timer_update->Update();
        execution_time += timer_update->Delta;

        // Suoritetaan vain, jos edellisestä suorituksesta kulunut yli 1/30s
        // Jos suorituksia jäänyt väliin (esim. kauan kestäneen piirtämisen vuoksi)
        // suoritetaan useampi kerta, jotta päästään suoritustiheyteen 30 Hz
        while(execution_time >= update_time)
        {
                m_game->Update(update_time);
                execution_time -= update_time;
        }

        // Pysähdytään odottamaan tarvittaessa, jotta piirtäminen tapahtuu
        // nopeudella 60 Hz
        timer_render->Update();
        if(timer_render->Delta < render_time)
        {
                wait(int(std::floor((render_time - timer->Delta)*1000 + 0.5f)));
                timer->Update();
        }

        // Laske interpolointiarvo, jotta piirtäminen voi ennustaa piirrettävien olioiden tulevan sijainnin
        float interpolation = execution_time / update_time;

        // Piirrä & näytä
        m_renderer->Render(interpolation, render_time);
        m_renderer->Present();
}
```

Liite 6. Ehdotetun mallin mukainen pelisilmukan toteutus neliytimisille mobiililaitteille - Päivitystehtävä

```
// Ehdotetun mallin mukainen toteutus neliytimisille mobiililaitteille
// → päivitystehtävän säie
// Lähdekoodi ehdotetun mallin mukaisesta toteutuksesta.
// Kehitysympäristönä Windows Phone 8

// Määrittele säie ja tallenna se muuttujaan thread_update
auto thread_update = ref new WorkItemHandler(
	[this](IAsyncAction^ thread_update)
{
	BasicTimer^ timer = ref new BasicTimer(); // Aikalaskuri säikeen toistoajalle

	// Päivitykset tapahtuvat teoriassa 1/30 sekunnin välein, mutta
	// käytännössä pieni heittely mahdollista → varmistetaan aikalaskurilla:
	BasicTimer^ timer_update = ref new BasicTimer();

	// Pakotetaan silmukan toistonopeus tähän (1/30 sekuntia, eli 30 Hz):
	float update_time = float(1/30);

	while (!m_windowClosed)
	{
		// Tarkista kauan edellisen silmukan suorittamiseen kului aikaa
		// ja laita säie odottamaan tarvittava aika
		timer->Update();
		if(timer->Delta < update_time)
		{
			wait(int(1000*(update_time - timer->Delta)));
			timer->Update(); // Poista odotettu aika aikalaskurista
		}

		// Lue ja prosessoi syöte
		timer_update->Update();
		m_game->Update(timer_update->Delta);
	}
});

// Päivityssäikeen suorittaminen pääohjelmasta käsin asynkronisesti
// → tallenna viittaus säikeeseen. Viittasta käyttämällä säie voidaan mm. lopettaa
m_thread_update = ThreadPool::RunAsync(thread_update, WorkItemPriority::Normal);
```

Liite 7. Liitutaulumallin toteutus

```
// Liitutaulumallin mukainen luokka julkisen datavaraston toteuttamiseen.
// Lähdekoodi ehdotetun mallin mukaisesta toteutuksesta.
#pragma once
#include "ParameterBase.h"
#include <string>
#include <map>

namespace SST
{
	class Blackboard
	{
	public:
		Blackboard() { };

		template <typename T>
		void set(std::string id, T value); // Aseta data liitutaululle
		template <typename T>
		T& get(std::string searched); // Lue data liitutaululta

	private:
		std::map<std::string, ParameterBase*> blackboard_map;
	};

	// Poimi data liitutaululta
	template <typename T>
	T& Blackboard::get(std::string searched) {
		if(blackboard_map.count(searched) == 0)
		{
			// Dataa ei löytynyt; lisää id:tä vastaava elementti liitutaululle arvolla nolla
			set<T>(searched, T(0));
		}

		// Hae pointteri etsittyyn elementtiin
		std::map<std::string, ParameterBase*>::iterator it = blackboard_map.find(searched);

		// Palauta liitutauluelementti
		return it->second->get<T>();
	}

	// Aseta data liitutaululle
	template <typename T>
	void Blackboard::set(std::string id, T value)
	{
		// Jos etsityn id:n mukainen olio löytyy,
		// päivitä arvo, muuten aseta uusi id/arvo -pari liitutaululle
		if(blackboard_map.count(id) > 0)
		{
			std::map<std::string, ParameterBase*>::iterator it = blackboard_map.find(id);
			it->second->setValue<T>(value);
		} else {
			ParameterBase *object = new Parameter<T>(value);
			blackboard_map.insert(std::pair<std::string, ParameterBase*>(id, object));
		}
	}
}
```