Kasimir Ilmonen

Erilaisten peleissä käytettävien pääsilmukoiden esittely

Tietotekniikan pro gradu -tutkielma

25. toukokuuta 2019

Jyväskylän yliopisto

Informaatioteknologian tiedekunta

Tekijä: Kasimir Ilmonen

Yhteystiedot: jyrysanteri@hotmail.com

Ohjaajat: Jonne Itkonen ja Paavo Nieminen

Työn nimi: Erilaisten peleissä käytettävien pääsilmukoiden esittely

Title in English: Presentation of different game loops

Työ: Pro gradu -tutkielma

Opintosuunta: Pelit ja pelillisyys

Sivumäärä: 137+0

Tiivistelmä: Tutkimuksessa tarkastellaan peleissä käytettäviä pääsilmukoita, jotka mahdollistavat reaaliaikaisuuden peleissä. Tarkoituksena on esitellä ja vertailla erilaisia pääsilmukkamalleja, jotta lukija osaisi valita omaan käyttöönsä sopivimman mallin. Esittely tehdään vahvasti lähdeaineiston pohjalta, mutta vertailussa myös tarkastellaan itse tekemääni peliä, jolle on toteutettu useita erilaisia pääsilmukoita. Tutkimuksessa tarkastellaan myös muiden tekemien pelien tai pelimoottorien käyttämiä pääsilmukoita kuten esimerkiksi Unityn ja Quaken.

Avainsanat: pelit, ohjelmointi, pääsilmukat, peliohjelmointi, rinnakkaisohjelmointi

Abstract: This research studies game loops which are structure that makes it possible to have real time video games. The purpose of the research is to present and compare different game loops so that reader would have knowledge to choose the best game loop for his or her game. The presentation heavily uses existing materials but it also studies the self created game, which uses more than one different game loops. Additionally the research studies game loops of other makers like game loops in Unity and Quake.

Keywords: games, programming, game loop, game programming, parallel programming

Kuviot

Kuvio 1. Pääsilmukan perusmallin rakenne ... 5
Kuvio 2. Eri päivitystapojen vertailu ... 7
Kuvio 3. Viivastymisestä toipuva pääsilmukkamalli ... 17
Kuvio 4. Interpoloivan pääsilmukan rakenne ... 19
Kuvio 5. Esimerkki interpolaatiosta ... 20
Kuvio 6. Säännöllinen ja epäsäännöllien päivitys erikseen -pääsilmukamalli ... 23
Kuvio 7. Esimerkki kuinka pankkisiirto voi mennä pieleen ... 29
Kuvio 8. Pääsilmukka, jossa tehtäviä irrotettu eri säikeisiin. ... 37
Kuvio 9. Haarautuva pääsilmukka ... 40
Kuvio 10. Tehtävän sisäinen rinnakkaistus ... 42
Kuvio 11. Unityn käyttämä pääsilmukka ... 58
Kuvio 12. Spacewarin käyttämä pääsilmukka ... 70
Kuvio 13. Kuva toteutetusta pelistä ... 75

Sisältö

1 JOHDANTO 1

2 PÄÄSILMUKAN PERUSMALLI 3
2.1 Osat ja rakenne 3
2.2 Päivitystavat 6
2.2.1 Vakiolla päivittyvä versio 7
2.2.2 Säännöllisesti päivittyvä versio 8
2.2.3 Ajan mukaan päivittyvä versio 10
2.3 Pelin deterministisyys 11

3 RINNAKKAISTAMATTOMAT PÄÄSILMUKKAMALLIT 16
3.1 Toistuvasti päivitys -malli 16
3.2 Interpoloiva malli 18
3.3 Säännöllinen ja epäsäännöllinen päivitys erikseen 22

4 RINNAKKAISTAMINEN YLEISESTI 25
4.1 Hienojakoisuus 26
4.2 Riippuvuudet rinnakkaistettaessa 27
4.3 Synkronointi 28
4.3.1 Lukot 29
4.3.2 Transaktiomuisti 30
4.3.3 Etenemisen estäminen 31
4.3.4 Viestien lähetys säikeiden välillä 31
4.4 Amdahlin laki 32
4.5 Rinnakkaistuksen tavat 33
4.5.1 Datan rinnakkaistus 33
4.5.2 Tehtävien rinnakkaistus 34

5 RINNAKKAISUUDESTA PÄÄSILMUKOISSA 36
5.1 Osien puhdas irrottaminen omaan säikeeseen 36
5.2 Suoritus haarautuu useisiin säikeisiin 39
5.3 Tehtävän sisäinen rinnakkaistus 41
5.4 Säieallas 44
5.5 Grafiikkasuorittimen käyttäminen rinnakkaistaessa 46

6 VALMIIDEN TOTEUTUKSIEN TARKASTELU 48
6.1 Phaser 49
6.1.1 Phaserin pääsilmukan rakenne 49
6.1.2 Phaserin pääsilmukan päivitystiheys 51
6.1.3 Phaserin yhteenveto 53
6.2 Unity 54
6.2.1 Unityn toiminnasta yleisesti 54
6.2.2 Unityn pääsilmukan rakenne ja päivitystiheys 56

6.2.3 Rinnakkaistus Unityssä ... 59
6.3 Quake ... 61
6.3.1 Quaken palvelimen rakenne ja päivitystiheys ... 62
6.3.2 Quaken asiakkaan rakenne ja päivitystiheys ... 64
6.3.3 Rinnakkaistus Quakessa ... 66
6.4 Spacewar ... 68
6.4.1 Spacewarin pääsilmukan rakenne ja toiminta ... 69
6.4.2 Spacewarin päivitystiheys ... 71
6.4.3 Spacewarin yhteenveto ... 73

7 OMAN TOTEUTUKSEN TARKASTELU ... 74
7.1 Pelin kuvaus ... 75
7.2 Pelin toteuttaminen ... 76
7.2.1 Työkalujen valinta ... 77
7.2.2 Syötteiden käsittelyn toteutus ... 78
7.2.3 Fysiikkamoottorin toteutus ... 79
7.2.4 Animaatioiden toteutus ... 81
7.2.5 Pelimaailman toteutus ... 82
7.2.6 Pääsilmukoiden yleinen toteutus ... 83
7.3 Tutkittavien pääsilmukoiden kuvaukset ... 84
7.3.1 Ajan mukaan päivittyvän pääsilmukan kuvaus ... 85
7.3.2 Säännöllisesti päivittyvän pääsilmukan kuvaus ... 86
7.3.3 Erilliset päivitykset -pääsilmukan kuvaus ... 87
7.3.4 Interpoloivan pääsilmukan kuvaus ... 88
7.4 Havaintoja silmukoiden toiminnasta erilaisilla päivitystiheyksillä ... 90
7.4.1 Ajan mukaan päivittyvän pääsilmukan havainnot ... 90
7.4.2 Säännöllisesti päivittyvän pääsilmukan havainnot ... 91
7.4.3 Erilliset päivitykset -pääsilmukan havainnot ... 92
7.4.4 Interpoloivan pääsilmukan havainnot ... 94
7.5 Tutkittavien pääsilmukoiden deterministisyys ... 96
7.5.1 Ajan mukaan päivittyvän pääsilmukan deterministisyys ... 97
7.5.2 Säännöllisesti päivittyvän pääsilmukan deterministisyys ... 97
7.5.3 Erilliset päivitykset -pääsilmukan deterministisyys ... 97
7.5.4 Interpoloivan pääsilmukan deterministisyys ... 99
7.6 Tutkittavien pääsilmukoiden toteutuksen vaativuus ... 100
7.6.1 Ajan mukaan päivittyvän pääsilmukan vaativuus ... 101
7.6.2 Säännöllisesti päivittyvän pääsilmukan vaativuus ... 101
7.6.3 Erilliset päivitykset -pääsilmukan vaativuus ... 102
7.6.4 Interpoloivan pääsilmukan vaativuus ... 103
7.7 Yhteenveto omasta pelistä ... 104

8 YHTEENVETO ... 106

9 LIITTEET ... 107

LÄHTEET 126

1 Johdanto

Tässä työssä jatketaan kandidaatin tutkimukseni aihetta eli pelien pääsilmukoita. Videopelit ovat toiminnaltaan muista ohjelmista poikkeavia, koska niissä ohjelman tilan tulee pystyä päivittymään käyttäjän syötteistä riippumattomasti. Käytännössä tämä tarkoittaa, että vaikka käyttäjä ei painaisikaan yhtään näppäintä, niin pelissä viholliset liikkuvat siitä huolimatta. Useimmissa ohjelmissa taas ohjelma pysähtyy odottamaan käyttäjän syötettä, joten pelit tarvitsevat omanlaisensa rakenteen käyttäjän syötteestä riippumatonta päivittämistä varten. Tätä varten on kehitetty omanlaisensa rakenne eli pääsilmukat.

Pääsilmukoista on kuitenkin olemassa useita perusmallia kehittyneempiä malleja, jotka tarjoavat erilaisia ominaisuuksia toteutettavalle pelille. Koska nämä ominaisuudet vaikuttavat merkittävästi pelin toimintaan, on pelin kehittäjän tärkeää osata valita omaan peliinsä sopivin pääsilmukkamalli. Tätä varten tässä työssä pyritään esittelemään pelien kehittäjille erilaisia pääsilmukkamalleja, ja samalla vertailemalla malleja keskenään, tarjotaan kehittäjille tietoa, jonka pohjalta valita käytettävä pääsilmukka. Esimerkiksi jotkin mallit tarjoavat pelille paremman suorituskyvyn, kun taas toiset mallit tarjoavat vakaammin toimivan fysiikkamoottorin ja pelin logiikan. Tässä työssä myös pyritään arvioimaan pääsilmukoiden työläyttä ja haasteellisuutta, mitkä ovat pääsilmukan valinnalle merkittäviä ominaisuuksia.

Vertailut ja pääsilmukoiden esittelyt tehdään vahvasti lähdekirjallisuutta hyödyntäen, mutta myös hyödynnetään minun itseni tekemiä havaintoja. Pääsilmukoiden vertailua varten myös tarkastellaan itse tekemääni peliä, joka käyttää useita erilaisia pääsilmukoita. Toteuttamalla samalle pelille useita pääsilmukoita pystytään parempiin arvioimaan, kuinka työläitä pääsilmukoiden toteutukset ovat.

Tutkimukseni tuokin pääsilmukoiden tutkimukseen lisäarvoa, koska harvoissa löytämissäni tutkimuksissa oli kokoavasti esitelty pääsilmukoita. Tutkimukset keskittyivät lähinnä vain esittelemään yksittäisiä pääsilmukkamalleja, ja kuinka esitelty malli on jotakin mallia parempi. Tutkimuksissa ei myöskään ollut toteutettu samalle pelille useampia pääsilmukoita, minkä takia pääsilmukoiden vertailut perustuivat lähinnä vain teoreettiseen tarkasteluun ja kirjoittajien omiin aiempiin kokemuksiin.

Tämä työ koostuu kolmesta isommasta kokonaisuudesta. Ensimmäisessä kokonaisuudessa eli luvuissa Pääsilmukan perusmalli, Rinnakkaistamattomat pääsilmukkamallit, Rinnakkaistaminen yleisesti ja Rinnakkaisuudesta pääsilmukoissa käsitellään erilaisia pääsilmukoita ja niihin liittyvää teoriaa. Toisessa osiossa eli luvussa Valmiiden toteutuksien tarkastelu taas tarkastellaan valmiiden pelien ja pelimoottorien käyttämiä pääsilmukoita ja verrataan niitä esiteltyyn teoriaan. Kolmannessa osiossa eli luvussa Oman toteutuksen tarkastelu taas tarkastellaan itse tekemääni peliä ja sen käyttämiä pääsilmukoita. Lisäksi tämä työ sisältää myös johdannon ja yhteenvedon.

2 Pääsilmukan perusmalli

Videopelit tuovat omanlaisiaan haasteita ohjelmistokehitykseen. Esimerkiksi niissä on vahva vaatimus reaaliaikaisuudelle. Shin ja Ramanathan (1994) mukaan reaaliaikaisuus tarkoittaa, että tehtävällä on määräaika, jota ennen tehtävän tulee olla suoritettu tai tehtävä epäonnistuu. Esimerkiksi peleissä tämä tarkoittaa, että näppäimistön painalluksen aiheuttaman tapahtuman, kuten pelihahmon liikkeen, tulee toteutua ja päivittyä ruudulle tarpeeksi lyhyessä ajassa, jotta pelattavuus ei kärsisi. Kuitenkin peleille määräajan asettaminen on joustavaa ja pikemminkin toivotaan mahdollisimman nopeaa suoritusta. Tällainen joustava määräaika voidaan tulkita Shin ja Ramanathan (1994) käyttämässä luokituksessa olevan pehmeä määräaika (eng. soft) eli tehtävän suorituksen tuloksen arvo heikkenee ajan myötä. Yleisenä ohjenuorana kuitenkin Valente, Conci ja Feijó (2005) antaa vähimmäisvaatimukseksi 16 päivitystä sekunnissa, jotta peli olisi vielä vuorovaikutteisesti pelattavissa. Ihanteelliseksi rajaksi taas annetaan 50-60 päivitystä sekunnissa.

Jotta reaaliaikaisuus toteutuisi peleissä, niin peleissä hyödynnetään yleisesti pääsilmukoita, koska niiden rakenne mahdollistaa jatkuvan ja nopean pelitilan päivittämisen käyttäjän syötteistä. Lisäksi pääsilmukat mahdollistavat pelitilan päivittämisen vaikkei käyttäjä antaisikaan syötettä. Esimerkiksi tekoälyvastustajat voivat toimia vapaasti pelissä, vaikka pelaaja ei tekisikään mitään. Pelimaailman päivityksen jatkumisen kannalta pelaajan syöte on siis tarpeeton ja syötteen puute voidaankin tulkita syötteeksi itsessään.

Tässä luvussa käsitellään yleisesti tunnettua pääsilmukan perusmallia. Perusmallin esittelyssä käydään läpi myös yleistä teoriaa pääsilmukoista kuten pääsilmukan eri osat ja käytettävään päivitysaikaan liittyvät seikat. Useimmat muut mallit pohjautuvat vahvasti perusmalliin, joten samat osat esiintyvät niissäkin, ja siksi pääsilmukan perusmallin tuntemus on hyvin tärkeää pääsilmukoihin perehdyttäessä.

2.1 Osat ja rakenne

Pääsilmukka rakenteena määrittelee, missä järjestyksessä pelin suoritukseen kuuluvat tehtävät suoritetaan (Tulip, Bekkema ja Nesbitt 2006). Pääsilmukan perusmallissa tehtävät on

jaettu kolmeen eri ryhmään, jotka ovat syöte, päivitys ja vaste (Valente, Conci ja Feijó 2005). Syötteellä tarkoitetaan yleensä pelaajalta tulevaa syötettä, joita ovat esimerkiksi näppäimen painallus tai peliohjaimen tatin liikuttaminen (Valente, Conci ja Feijó 2005). Lisäksi myös verkosta tulevat viestit voidaan nähdä syötteenä (Tulip, Bekkema ja Nesbitt 2006).

Saadun syötteen avulla päivitetään pelimaailman tilaa esimerkiksi liikuttamalla pelihahmoa. Lisäksi päivitykseen kuuluu myös käyttäjän syötteestä riippumattomia päivityksiä, kuten tekoälyä käyttävien vihollisten toimet, pelimaailman fysiikasimulaation päivitys ja muu pelin logiikka (Tulip, Bekkema ja Nesbitt 2006). Toisaalta pelitilan päivityksessä voi syntyä uusia tehtäviä, jotka tulee myös käsitellä (Tulip, Bekkema ja Nesbitt 2006). Esimerkiksi fysiikkasimulaatiossa kohteiden siirtäminen voi aiheuttaa niden välille törmäyksiä, jotka tulee myös ratkoa.

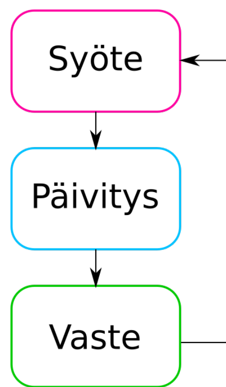
Syötteiden lisäksi päivityksellä voi olla päivitysaika, joka kuvaa kuinka paljon pelimaailmaa päivitetään. Yhtenä päivitysaikana voidaan käyttää viime päivityksestä kulunutta aikaa, aihetta käsitellään tarkemmin luvussa Päivitystavat.

Pelimaailman päivityksen jälkeen pelaajalle annetaan vaste, joka useimmiten tarkoittaa kuvan piirtämistä näytölle pelimaailmasta (Tulip, Bekkema ja Nesbitt 2006). Vasteeksi voidaan myös ajatella kuuluvan audio eli pelin äänet ja peliohjaimen tärinä. Kuitenkaan ne eivät ole välttämättä täysin riippuvaisia päivityksestä, toisin kuin kuvan piirtäminen näytölle, mitä varten koko päivityksen pitää olla valmis. Esimerkiksi, jos pelihahmo kuolee, niin voidaan vain laittaa kuolinääni soimaan ja tärisyttää ohjainta ilman, että pelimaailman muilla muutoksilla on väliä. Näin ollen audio ja ohjaimen tärisyttäminen ovat enemmänkin reaktioita pelimaailman yksittäisiin tapahtumiin, eivätkä anna laajempaa palautetta pelimaailman tilasta, toisin kuin näytölle piirtyvä kuva.

Poikkeuksena kuitenkin ovat kolmiulotteiset pelit, joissa pyritään mallintamaan akustiikkaa kolmiulotteisessa ympäristössä. Tällöin mallinnettaessa ääniaalto lähtee kohteesta ja heijastuu esteistä, kuten seinistä, ennen kuin kuulija vastaanottaa ääniaallon, mikä vaikuttaa siihen minkälaisena ääni kuullaan (Gardner 1999). Selkeästi myös kuulijan etäisyys äänilähteeseen vaikuttaa ääniaaltoon vaimentamalla kuultua ääntä (Gardner 1999). Tämän perusteella on selkeää, että pelimaailmassa ympäristö vaikuttaa kuultuun äänen, minkä takia joudutaan päi-

vittämään pelimaailman tila kokonaan, jotta saadaan kaikki ääneen vaikuttavat tekijät huomioitua oikein.

Vasteen antamisen jälkeen palataan takaisin syötevaiheeseen, jolloin muodostuu silmukkarakenne, jota havainnollistetaan kuvassa 1. Silmukka pyörähtää läpi peleissä tyypillisesti 30-60 kertaa sekunnissa, jolloin saadaan käyttäjältä syöte todella pienin väliajoin ja vastaavasti annetaan vaste todella usein, jolloin pelaajalle muodostuu vaikutelma jatkuvasta pelimaailman muutoksesta. Esimerkiksi pelihahmon raajat antavat sulavan liikkeen vaikutelman käveltäessä sen sijaan, että niiden asennot muuttuisivat näkyvin vaihein.



Kuvio 1. Pääsilmukan perusmallin rakenne

Vaikka pääsilmukan perusmallissa esitetään selkeä järjestys eri tehtäväryhmillä eli syötteellä, päivityksellä ja vasteella, niin todellisuudessa ei välttämättä ole tarvetta näin tiukalle järjestykselle. Monet asiat päivityksessä eivät välttämättä vaadi kaikkia käyttäjän syötteitä ja monet vasteet eivät vaadi koko pelimaailman päivitystä. Esimerkiksi tekoälyvastustajat eivät välttämättä ole kiinnostuneita pelaajan näppäinten painalluksista ja voivat toimia niistä riippumattomasti. Tämän takia tekoäly vastustajat voivat tehdä omat päivityksensä jo ennen käyttäjän syötettä.

Tällä samalla logiikalla myös muut syötteestä riippumattomat päivitykset voidaan tehdä joko ennen tai jälkeen syötteen. Itse asiassa käyttäjän syötteistä riippuvia tehtäviä voi olla todella vähän, ja useimmiten ne koskevat vain pelattavaa hahmoa. Tällöin toteutuksessa voi olla luontevampaa kysyä syötteitä hajallaan niitä vaativia päivityksiä tehtäessä. Esimerkiksi kun tullaan pelihahmon sijaintia päivittävään tehtävään, niin tällöin voidaan tarkistaa nuolinäppäinten tilat, joiden perusteella sijainti muuttuu. Kuitenkaan toteutuksen toimivuuden kan-

nalta ei ole juurikaan merkitystä pidetäänkö syötteiden keruu selkeästi omana ryhmänään vai sekoitetaanko ne osaksi päivitystä.

Tiukan tulkinnan mukaan syötteet tulisi kuitenkin kerätä pääsilmukan perusmallissa keskitetysti ennen päivitysvaiheen aloittamista. Käytännössä tämä tarkottaisi sitä, että esimerkiksi tarkkailtavien näppäinten tilat kerättäisiin talteen johonkin tietorakenteeseen, josta sitten päivitysvaiheessa tarkistetaan niiden tilanne. Tämä mahdollistaisi helpommin syötteiden esikäsittelyn ja esimerkiksi voidaan tallentaa näppäimen tilasta lisäksi tiedot, onko se juuri noussut pohjasta tai juuri painettu pohjaan sen sijaan, että tarkailtaisiin vain, onko näppäin pohjassa vai ei.

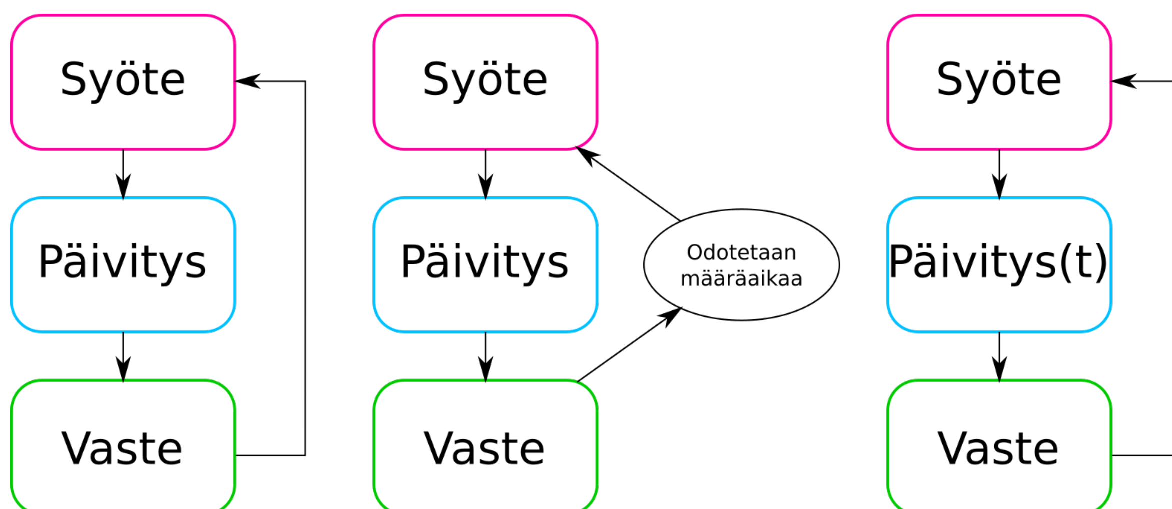
Keskitetty syötteiden keruu myös voisi helpottaa pelin toimintoihin sidottujen näppäinten vaihtamista. Esimerkiksi voitaisiin helposti yhdessä paikassa määrittää uudestaan pelissä käytettävät näppäimet sen sijaan, että tarvitsisi tehdä muutokset useassa paikassa pelin lähdekoodia. Toisaalta haittana pelin kehittäminen saattaisi käydä kankeammaksi, sillä pitäisi pelin toimintoa tehtäessä käydä syötevaiheessa määrittelemessä vaaditut näppäimet erikseen sen sijaan, että vain kysyisi yhdellä rivillä, onko tietty näppäin pohjassa.

2.2 Päivitystavat

Pääsilmukoiden päivitysvaiheessa on ongelmana, kuinka paljon pelimaailmaa päivitetään eli esimerkiksi kuinka paljon pelihahmoa liikutetaan kullakin silmukan kierroksella. Tähän ongelmaan on kehitetty useampi erilainen päivitystapa, joita voidaan käyttää yhdessä pääsilmukan perusmallin rakenteen kanssa. Myös muissa pääsilmukan malleissa voidaan käyttää tässä luvussa esiteltäviä päivitystapoja ja siksi tässä työssä luokitellaan eri päivitystapoja käyttävät pääsilmukan toteutukset saman mallin eri versioiksi. Vaikka käytettävä päivitystapa vaikuttaakin huomattavasti toteutukseen, niin tehtävien suoritusjärjestys pysyy kuitenkin samana. Tässä työssä käytetäänkin tehtävien suoritusjärjestystä pääsilmukan määrittelyssä ja siksi on luontevaa pitää samaa suoritusjärjestystä käyttäviä pääsilmukoita samaan malliin kuuluviksi.

Tässä luvussa esitellään kolme erilaista päivitystapaa, jotka kaikki ovat käytettävissä pääsilmukan perusmallissa. Eri versioiden väliset rakenteelliset erot näkyvät kuvassa 2, jossa pe-

rusmalliin on tehty kutakin versiota vastaavat muutokset. Eri päivitystavat tarjoavat erilaisia ominaisuuksia pääsilmukalle, joten suositeltava päivitystapa riippuu pelilaitteen ja toteutettavan pelin vaatimuksista.



Kuvio 2. Vakiolla päivittyvä versio, säännöllisesti päivittyvä versio ja ajan mukaan päivittyvä versio pääsilmukan perusmallista

2.2.1 Vakiolla päivittyvä versio

Yksinkertaisimmassa päivitystavassa asetetaan tietty vakiomäärä, joka päivityksessä tehdään (Valente, Conci ja Feijó 2005). Tässä työssä käytämme tästä ratkaisusta termiä vakioitu päivitystapa. Kyseisessä versiossa esimerkiksi pelihahmo voisi liikkua aina yhden pikselin verran kullakin pääsilmukkakierroksella. Tällöin kuitenkin pelin nopeus tulee suoraan siitä, että kuinka nopeasti kukin silmukkakierros saadaan tehtyä. Tämä taas riippuu suoraan laitteiston suorituskyvystä, jolloin peli toimii eri pelilaitteilla eri nopeuksilla (Valente, Conci ja Feijó 2005). Esimerkiksi kaksi kertaa tehokkaammalla tietokoneella pelihahmot liikkuisivat kaksi kertaa nopeammin, jolloin pelikokemus muuttuisi huomattavasti. Tämän takia peli joudut-taisiin suunnittelemaan toimimaan vain yhdenlaisilla pelilaitteilla, mikä ei ole todellakaan kannattavaa.

Toisaalta vaikka voitaisiinkin sopeutua tähän vaatimukseen, niin pelilaitteiden suorituskyky ei ole välttämättä tasaista, jolloin peli saattaisi välillä hidastua tai kiihtyä yllättäen. Myös peli saattaa muuttua pelin aikana raskaammaksi suorittaa, jos esimerkiksi pelissä luodaan paljon

lisää vihollisia, jolloin silmukan kierrosta ei voida toteuttaa jo totutussa ajassa. Ratkaisu ei siis ole todellakaan käyttökelpoinen, ja mielestäni sitä ei tulisi käyttää sellaisenaan paitsi poikkeustapauksissa. Vakioitu päivitystapa esitelläänkin lähinnä vain historiallisista syistä, ja koska muut versiot voidaan johtaa siitä.

Mielenkiintoisen havainto kuitenkin voidaan tehdä Spacewaria tarkastelemalla, jossa vakiolla päivittämisen ongelmaa on jossain määrin onnistuttu korjaamaan. Kyseisessä ratkaisussa kellon käyttämisen sijaan laskettiin suoritettavien komentojen määrää ja niin avulla varmistettiin, että jokainen silmukan kierros on suunnilleen yhtä pitkä. Tähän ratkaisuun luultavimmin päädyttiin, koska ei ollut kelloa käytössä. Ratkaisu korjaa kuitenkin vain samalla pelilaitteella pelattaessa, ja edelleen pelien nopeudet eroavat huomattavasti eri tehoisia pelilaitteita käytettäessä. Aihetta käsitellään tarkemmin luvussa Spacewarin päivitystiheys.

2.2.2 Säännöllisesti päivittyvä versio

Vakiolla päivittyvästä versiosta saadaan kuitenkin yleisesti käyttökelpoinen, jos silmukka asetetaan toistumaan tietyin väliajoin, kuten esimerkiksi 30 kertaa sekunnissa (Valente, Conci ja Feijó 2005). Tämä voidaan käytännössä toteuttaa katsomalla silmukan alussa, onko tarpeeksi aikaa kulunut viime päivityksestä. Jos ei ole, niin silmukan vaiheet (eli syöte, päivitys ja vaste) ohitetaan. Lisäksi peliä suorittavan säikeen suoritus voitaisiin keskeyttää hetkeksi, kun kuitenkin tiedetään, että seuraavaan silmukan suorituskertaan on aikaa. Tässä työssä tämän luvun ratkaisusta käytetään termiä säännöllinen päivitystapa.

Koska silmukka päivittyy tietyin säännöllisin väliajoin, niin pelin nopeus on kaikilla laitteilla pääsääntöisesti sama (Valente, Conci ja Feijó 2005). Ongelmia kuitenkin tulee, jos pelilaitteen suorituskyky ei pysty takaamaan silmukan suoritusta seuraavaan väliaikaan mennessä, mikä saattaisi näkyä pelin hidastumisena (Witters 2009). Tämän takia peli pitäisi suunnitella vähemmän suorituskykyiset laitteet mielessä tai asettaa pelille tarpeeksi korkeat laitteistovaatimukset. Toisaalta heikommille laitteille suunnittelu on myös ongelmallista. Tällöin tehokkaammilla laitteilla peli ei toimi paremmin kuin heikommilla laitteilla, jolloin parempi suorituskyky jää turhaksi ja peli ei siten skaalaudu suorituskyvyn suhteen (Valente, Conci ja Feijó 2005). Esimerkiksi pelaajalle ei pystytä tarjoamaan suurempaa ruudunpäivitysnopeut-

ta paremmilla laiteilla. Witters (2009) mainitsee kuitenkin, että tämä voi olla mobiilipeleissä etu, koska laiteen ei tarvitse silloin toimia täydellä teholla, mikä säästää mobiililaitteen akkua.

Toisaalta koska peli päivittyy säännöllisin väliajoin aina saman verran, niin pelikerran tapahtumat ovat helposti toisinnettavissa pelkästään tallentamalla kaikki syötteet (Valente, Conci ja Feijó 2005). Tämän ansiosta peli on deterministinen, mikä tarjoaa pelille helpomman toteutuksen joillekin ominaisuuksille, kuten esimerkiksi pelikerran uudelleen katsomisen ja verkkopelin toteutuksen. Luvussa Pelin deterministisyys käsitellään tarkemmin pelin deterministisyyttä ja sen hyötyjä.

Fiedler (2004) myös ohjaa käyttämään säännöllistä päivitystapaa fysiikkasimulaatioissa (eli fysiikkamoottoreissa), jotta jokainen pelin suorituskerta olisi samanlainen. Perusteluna tuodaan myös esille yleinen fysiikan simuloinnissa esiintyvä ongelma, jossa pelimaailman kappaleiden nopeuden kasvaessa, ne voivat mennä seinistä läpi. Tämä tyypillisesti johtuu fysiikkamoottorin toteutuksesta, jossa törmäyksiä etsitään vain uuden lasketun sijainnin alueelta, jolloin jää huomaamatta kaikki uuden ja vanhan sijainnin väliset esteet. Ongelmia ei tule niin kauan kuin kappaleen sijainnin muutos pysyy pienenä, koska silloin uuden ja vanhan sijainnin väliin ei edes mahtuisi esteitä.

Sijainnin laskemisen kaavasta eli

$$uusiSijainti = vanhaSijainti + nopeus * aika$$

voidaan päätellä, että sijainnin muutos tulee suoraan kappaleen nopeudesta ja päivityksessä käytettävästä ajasta. Jomman kumman kasvaessa käy todennäköisemmäksi, että kappaleet alkavat läpäistä seiniä, ja siksi tulisikin estää tai varautua niiden kasvamiseen. Nopeus voidaan pelissä useimmiten rajata tai ennakoida tarpeeksi hyvin, mutta päivityksessä käytettävä aika riippuu päivitystavasta. Fiedler (2004) ohjeistaakin käyttämään säännöllistä päivitystapaa, jotta fysiikkamoottorille pystyttäisiin takaamaan tarpeeksi pieni päivitysaika.

Vaihtoehtoisesti voidaan myös käyttää fysiikkamoottorissa törmäystentarkastelussa tarkempia algoritmeja, jotka huomioisivat paremmin nopeasti liikkuvat kohteet. Tällaiset algoritmit kuitenkin luultavasti olisivat haastavampia toteuttaa ja suorituskyvyllisesti raskaampia. Kuitenkin seinien läpäiseminen on vain yksi ongelma. Fiedler (2004) tuo esille ajatusta, että

päivitysajan kasvaessa simulaation tarkkuus alkaa kärsiä ja tilanne käy hallitsemattomaksi. Säännöllisesti päivittyvässä päivitystavassa voidaan kuitenkin itse määritellä, kuinka usein päivitetään ja siten valita tarpeeksi pieni päivitysaika.

2.2.3 Ajan mukaan päivittyvä versio

Kolmannessa vaihtoehtoisessa versiossa taas ei puututa pääsilmukan kierrosnopeuteen vaan, huomioidaan jokaisen silmukan kierrokseen kuluva aika päivityksessä (Valente, Conci ja Feijó 2005; Witters 2009). Päivityksille annetaan parametrinä viimepäivityksestä kulunut aika, jota hyödynnetään laskuissa (Valente, Conci ja Feijó 2005). Tämä on sikäli luonnollinen ratkaisu fysiikkamoottoreissa, koska fysiikassa kohteen sijainti voidaan laskea kaavalla:

$$uusiSijainti = vanhaSijainti + nopeus * aika$$

Sijainnin laskemiseen tarvitsee siis vain tietää kohteen senhetkinen nopeus ja vanha sijainti, kun taas aika saadaan annettuna parametrinä. Vastaavasti muutkin päivitykset voidaan sitoa aikaan. Tätä kolmatta vaihtoehtoista ratkaisua kutsutaan tässä työssä ajan mukaan päivittyväksi versioksi.

Ajan mukaan päivittyvässä versiossa silmukka käydään läpi, niin monta kertaa kuin vain suorituskyky mahdollistaa. Hitaammilla laitteistoilla päivitetään siis harvemmin, kun taas tehokkaammilla useammin, jolloin saadaan hyödynnettyä suorituskyky kokonaan toisin kuin säännöllisessä päivitystavassa (Valente, Conci ja Feijó 2005). Ajan avulla päivitettäessä saadaan lähes vastaava pelitila riippumatta, kuinka usein päivitetään, jolloin pelikokemus pysyy pitkälti samanlaisena. Kuitenkin pieniä eroja tulee muodostumaan pelin myötä, joten peli ei ole deterministinen, jolloin pelikerta on vaikeammin toistettavissa (Valente, Conci ja Feijó 2005). Pelikertaa toistettaessa pitäisi syötteiden lisäksi käyttää toistettavan pelikerran päivityksessä käytettäviä aikoja, mikä tekee toteutuksesta haastavampaa.

Kuitenkin ajan mukaan päivittyvässä versiossa ajan käyttämisen ansiosta tapahtumien hallitseminen on helpompaa. Esimerkiksi on huomattavasti mukavampaa määritellä, että kohteen nopeus on 5 yksikköä sekunnissa kuin, että kohde liikkuu 0,002323 yksikköä joka päivityksellä, joista jälkimmäistä vaihtoehtoa voitaisiin hyödyntää säännöllisessä päivitystavassa. Kuitenkin säännöllisessä päivitystavassa tulee heti haasteita, jos halutaan muuttaa kuinka

usein pääsilmukan tehtävät toteutetaan, jolloin jouduttaisiin muokkaamaan kaikki päivitykset uudestaan tukemaan uutta päivitysnopeutta. Tämän takia myös säännöllisessä päivitystavassa olisi mielestäni harkittavissa hyödyntää aikaa päivityksiä laskettaessa, vaikka käytettävä aika olisikin vakio.

Tässä päivitystavassa on ongelmia, jos päivitysten välinen aika kasvaa liian suureksi esimerkiksi hetkellisen suorituskyvyn rasituksen takia (Witters 2009). Tällöin kuten luvussa Säännöllisesti päivittyvä versio todettiin, alkaa pelin tarkkuus kärsimään, ja esimerkiksi fysiikkasimulaatiossa voi ilmetä vakavia ongelmia. Myös pelaaminen voi käydä vaikeaksi, koska käyttäjä ei pysty välttämättä reagoimaan ajoissa (Witters 2009). Esimerkiksi, jos pelaaja lähestyy pelihahmollaan rotkon reunaa, niin sekunnin mittainen aika, jolloin päivitystä ei tapahdu, tekee hypyn ajoittamisen todella vaikeaksi. Näin varsinkin, jos pitkä päivitysväli tulee yllättäen. Säännöllisesti päivittyvässä versiossa ongelmaa ei ole, koska peli viivästyy eli on käytännössä hidastunut, kun suorituskyky ei pysy päivitysnopeudessa. Esitelty ongelma ilmenee helpoiten, jos pelilaitteiston suorituskyky on matala (Witters 2009).

Kuitenkin Witters (2009) mukaan ongelmia voi myös ilmetä tehokkaammillakin laitteistoilla lukujen käsittelyissä. Koska tietokoneet eivät yleisesti ottaen pysty tarkasti käsittelemään kaikkia desimaalilukuja, niin laskutoimitukset niillä voivat aiheuttaa erikoisia ja haastavia ongelmia korjata (Witters 2009). Esimerkiksi 0,1 on luku jota, tietokone ei pysty kunnolla käsittelemään, jolloin sille tehtävät laskutoimitukset antavan aina hieman väärän tuloksen. Yksittäisessä laskussa tämä ei ole ongelma, mutta kun laskua toistetaan useasti, niin virhe alkaa kasaantua. Tämä korostuu, jos päivitystiheys on suuri, jolloin myös laskujakin tehdään enemmän (Witters 2009). Ennen pitkää virhe kasvaa tarpeeksi suureksi ja se alkaa näkyä pelin toiminnassa ja laskuissa voi ilmaantua suuriakin ongelmia. Erityisen pirullisen ongelmasta tekee se, että tämän korjaaminen on haastavaa, jos halutaan käyttää tätä päivitystapaa (Witters 2009).

2.3 Pelin deterministisyys

Tässä työssä pelin deterministisyydellä tarkoitetaan, että pelikerta toistuu aina täysin samanlaisena, jos pelikerta alkaa samasta lähtötilanteesta ja sille annetaan samat syötteet. Ei deter-

ministisessä pelissä taas samoilla syötteillä pelikerran tapahtumat eivät toistu samanlaisena, mikä aiheuttaa haasteita joitakin ominaisuuksia toteutettaessa pelille. Deterministisyyden sijaan voidaan myös puhua toistettavuudesta (eng. reproducibility), kuten Dickinson (2001) puhuu, koska deterministiset pääsilmukat mahdollistavat pelin helpon uudelleen toisintamisen. Toistettavuus on kuitenkin enemmän kohdealue (eli pelit) kohtainen termi, kun taas tietotekniikassa puhutaan oman kokemukseni mukaan samasta ilmiöstä yleisemmin deterministisyytenä. Lisäksi pelin helppo toistettavuus on vain yksi ominaisuus, joka seuraa deterministisyydestä ja siksi on mukavampi puhua suoraan deterministisyydestä muita ominaisuuksia listattaessa.

Pelin deterministisyys ei kuitenkaan yleisesti ottaen näy pelaajalle, koska pelaaja tuskin pystyy toistamaan oman pelikertansa sekunnin sadasosien tarkkuudella. Nämä pienet erot syötteiden ajoituksissa johtavat väistämättä erilaiseen pelikertaan. Tämän takia pelin deterministisyys on pikemminkin pelin kehittäjää hyödyttävä ominaisuus, mikä auttaa pelin kehittämisessä ja tiettyjen pelin ominaisuuksien toteutuksissa.

Pääsilmukoiden kannalta deterministisyys riippuu suoraan käytettävästä pääsilmukkamallista. Jos käytettävä pääsilmukka ei ole deterministinen, niin deterministisyyttä ei voida käytännöllisellä tavalla saavuttaa. Tämän takia käytettäväksi valitun pääsilmukkamallin merkitys kasvaa, jos pelille halutaan deterministisyydestä tulevia ominaisuuksia. Kuitenkin deterministinen toteutus saattaa tuoda joitain rajoitteita, jotka tukevat ei deterministisen mallin valintaa, jos deterministisyyttä ei tarvita.

Deterministisyyden hyötynä on, että tallentamalla pelikerran syötteet ja lähtötilan, voimme helposti toteuttaa pelille uusinnan katsomisominaisuuden (eli pelaaja voi katsoa pelikerran jälkeen omaa vanhaa pelikertaansa), koska deterministisissä peleissä pelikerta toistuu samoilla syötteillä samanlaisena. Tämä onnistuu syöttämällä pelikerralta kerätyt syötteet samassa järjestyksessä pelimoottorille, jolloin deterministisyyden mukaisesti peli toistuu täysin samanlaisena (Dickinson 2001). Jos peli ei olisi deterministinen, niin emme voisi olla varmoja, että peli toistuisi samanlaisena, ja joutuisimme esimerkiksi tallentamaan pelin syötteiden sijaan koko pelimaailman tilan, mikä veisi huomattavasti enemmän muistia eikä olisi muutenkaan käytännöllistä (Dickinson 2001). Sen sijaan syötteiden kerääminen pelin aikana on suhteellisen helppoa, ja siten uusinnan katsomisen toteuttaminen on huomattavasti

helpompaa toteuttaa.

Toisena hyötynä on lisäksi, että myös verkkopeli voidaan toteuttaa helpommin (Dickinson 2001). Koska pelikerta toistuu samanlaisena samoilla syötteillä, niin peli voidaan synkronoida verkon yli lähettämällä pelkästään kunkin käyttäjän syötteet (Dickinson 2001). Tällä tavalla verkkopeli on todella helppoa toteuttaa ja kehittäjän ei tarvitse juurikaan pelätä, että eri paikalliset pelit eroaisivat toisistansa. Myös verkkoyhteydelle on helpompaa, kun voidaan lähettää pienempiä viestejä lähettämällä ainoastaan syötteet sen sijaan, että lähettäisimme isompia pelimaailmaa koskevia tietoja (Dickinson 2001). Myös vältytään ylimääräiseltä suunnittelulta, kun helposti tiedetään, mitkä asiat (eli syötteet) pitää lähettää muille pelaajille ilman, että sitä tarvitsee sen enempää selvitellä (Dickinson 2001).

Vaikkei peliin tehtäisikään uudelleen katsomisominaisuutta tai verkkopeliä, niin deterministisyydestä hyödytään myös kehitysvaiheessa. Koska peli voidaan toistaa samanlaisena, niin voimme myös helposti toisintaa kaikki ne pelikerrat, joissa pelissä ilmenee vikoja (Dickinson 2001). Tämän ansiosta voidaan helpommin selvittää, mistä vika johtuu ja siten ratkaista ongelma vähemmällä vaivalla. Myös automatisoitujen vikailmoitusten lähettäminen kehittäjälle on helpompaa, koska kehittäjä voi itse toisintaa vian omalla laitteellaan pelkkien syötteiden ja pelin alkutilanteen avulla (Dickinson 2001).

Kerättäviä syötteitä on käyttäjän syötteiden lisäksi mahdollisesti myös verkon välityksellä tulevat viestit, jotka voivat vaikuttaa pelin toimintaan. Lisäksi on myös olemassa ulkoisia lähteitä, jotka voivat vaikuttaa pelin kulkuun, jotka tulee käsitellä (Dickinson 2001). Erityisesti peleissä tyypilliset satunnaisluvut ovat sellaisia, jotka voivat vaihdella helposti pelikertojen välillä (Dickinson 2001).

Satunnaislukuja varten on kehitetty näennäissatunnaislukugeneraattoreita, jotka antavat jollakin matemaattisella kaavalla lukuja sarjana ulos (Press ym. 2007). Luvut eivät ole täysin satunnaisia, mutta ovat usein tarpeeksi hyviä käytettäväksi sovellusalueella (Press ym. 2007). Generaattoreille annetaan usein siemenluku, jolla generaattori alustetaan (Press ym. 2007). Siemenluku on sikäli oleellinen, että samalla siemenluvulla sama generaattori antaa aina saman lukusarjan ulos (Press ym. 2007). Tämän ominaisuuden takia, tallentamalla siemenluvun muistiin, voimme helposti toistaa samat satunnaisluvut pelikertojen välillä, jolloin de-

terministisyys toteutuu (Dickinson 2001). Vaihtoehtoisesti voidaan vain käsitellä satunnaislukuja syötteiden tavalla eli tallentamalla jokainen käytetty satunnaisluku pelikerran aikana.

Yksi merkittävimpiä esteitä deterministisyydelle on, että peleissä usein hyödynnetään kulunutta aikaa pelilogiikassa, kuten esimerkiksi luvussa Ajan mukaan päivittyvä versio tehdään. Tämä estää deterministisyyden, koska päivitysaika voi vaihdella jokaisella pääsilmukan kierroksella hyvin epäsäännöllisesti, mikä ei ole toistettavissa seuraavalla pelin suorituskerralla (Dickinson 2001). Koska käyttöjärjestelmä pyörittää muitakin ohjelmia taustalla, niin suorittimen kuorma voi olla hyvinkin vaihtelevaa ja siksi myös pääsilmukan kierrosten suoritusaika vaihtelee (Dickinson 2001). Pienikin poikkeama käytettävässä päivitysajassa aiheuttaa poikkeavuuksia pelimaailman päivityksessä, mikä rikkoo deterministisyyden. Yksinkertaisin tapa säilyttää pääsilmukan determininistisyys, onkin käyttää säännöllistä päivitystapaa, jossa käytettävä päivitysaika on ennalta määritelty (Dickinson 2001). Tällöin kaikkissa päivityksissä käytettävät päivitysajat ovat samoja suorituskerrasta riippumatta ja siten aika ei enää vaikuta pelin logiikkaan.

Vaihtoehtoisesti, jos halutaan vain toistaa peli samanlaisena, niin voidaan syötteiden lisäksi tallentaa myös kaikki käytetyt päivitysajat. Peliä toistettaessa vain huomioisimme tämän ja käyttäisimme tallentamaamme aikaa. Tämä kuitenkin toisi ylimääräistä vaivaa itse pelin toteutukseen ja hankaloittaisi pelin toistamisen toteutusta. Lisäksi käytettävät päivitysajat saattaisivat vaihdella huomattavasti ja pelin toistamisessa voisi näkyä päivityskatkoja, niissä kohdissa, joissa peli on hidastunut merkittävästi. Käytännössä voisi ilmetä pelikertaa katsottaessa jopa sekunnin pätkiä, jolloin peli ei päivity. Peli ei olisi tällöin myöskään enää oikeasti deterministinen, vaikka pelikerta voitaisiinkin toistaa täysin samanlaisena. Deteministisyyden puute saattaisi vaikeuttaa joidenkin ominaisuuksien toteutusta verrattuna oikeasti deterministiseen versioon.

Lisäksi deterministisyyden uhkana Dickinson (2001) mainitsee liukulukujen käsittelyn. Eri laitteet käyttävät eri suorittimia, joissa on erilaisia matematiikkasuorittimia (eng. Floating point unit eli FPU) (Dickinson 2001). Eri matematiikkasuorittimet saattavat antaa hieman erilaisia tuloksia tietyissä laskuissa (Dickinson 2001). Myös sovelluksen eri käännökset (eng. software builds) voivat aiheuttaa erilaista käytöstä samalla laitteistollakin (Dickinson 2001). Dickinson (2001) ehdottaa ratkaisuksi kokonaislukujen käyttämistä liukulukujen sijaan ti-

lanteissa, jotka vaikuttavat pelin logiikkaan. Kuitenkaan esimerkiksi kuvan piirtämiseen liittyvän datan kohdalla tämä ei ole useinkaan tarpeellista, koska sillä on vaikutusta lähinnä vain palautteeseen eikä pelin logiikkaan (Dickinson 2001).

3 Rinnakkaistamattomat pääsilmukkamallit

Pääsilmukan perusvaatimuksena on usein reaaliaikaisuus ja jatkuva pelitilan päivittäminen, mikä onnistuu useimmiten pääsilmukan perusmallilla. Kuitenkin reaaliaikaisuus saattaa kärsiä, jos pelimaailma on suuri, tai se on muuten suorituskyvylle raskasta päivittää. Tällöin pelilaite ei välttämättä pysty toteuttamaan pelitilan päivittämistä tarpeeksi nopeasti, jolloin päivitystiheys laskee ja pelattavuus kärsii. Tämän takia on kehitetty pääsilmukan perusmallista kehittyneempiä malleja, jotka tarjoavat paremman suorituskyvyn muun muassa säikeistämisen avulla. Toisaalta parempi suorituskyky voidaan myös ottaa pelin sisällön suunnittelussa huomioon ja esimerkiksi pystytään luomaan suurempi ja monimutkaisempi pelimaailma.

Toisaalta suorituskyvyn lisäämisen lisäksi erilaisilla pääsilmukan malleilla voidaan ratkaista muitakin perusmallin puutteita, joita havaittiin luvussa Päivitystavat, jossa esiteltiin erilaisia päivitystapoja. Päivitystavat ovat myös keskeisessä osassa eri pääsilmukan malleissa, koska eri päivitystapojen puutteita ollaan pyritty ratkaisemaan muokkaamalla perusmallia.

3.1 Toistuvasti päivitys -malli

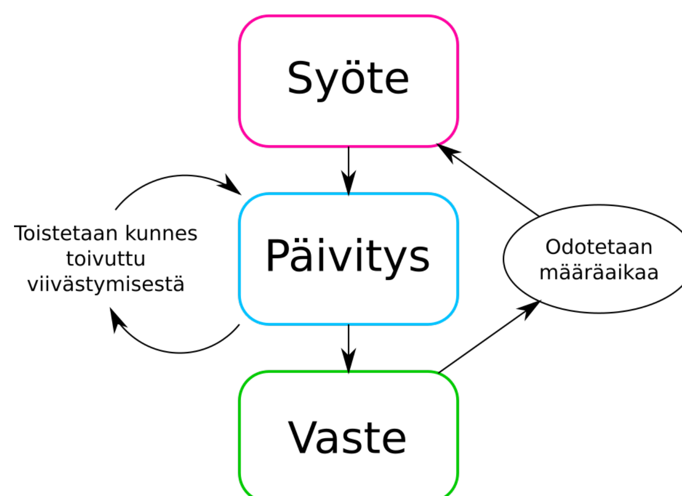
Pääsilmukan perusmalliin parannusta tarjoaa Fiedler (2004) esittelemällä uuden mallin. Samankaltaisia ideoita pääsilmukan mallissa on kuitenkin esitellyt myös Valente, Conci ja Feijó (2005) hieman erilaisella tarkoituksella ja eri lähtökohdista. Fiedler (2004) tuo vahvasti esille uudessa mallissa käytettävän säännöllisen päivitystavan etuja. Valente, Conci ja Feijó (2005) taas tuo enemmän esille, kuinka mallin tarjoama pääsilmukka pystyy toipumaan viivästyksistä ilman, että säännöllisesti päivittyvän mallin toisinnettavuus kärsii.

Esitellyssä pääsilmukan mallissa oleellista on, että se käyttää säännöllistä päivitystapaa (Fiedler 2004) (Valente, Conci ja Feijó 2005). Uutena rakenteellisena ominaisuutena on taas, että päivitysvaiheessa voidaan tehdä useampi päivitys kerralla ennen vastetta, kunnes ollaan toivuttu viivästyksestä (Fiedler 2004) (Valente, Conci ja Feijó 2005). Esimerkiksi, jos syystä tai toisesta pääsilmukka on jäänyt jälkeen 0,3 sekuntia, niin suoritetaan säännöllisen päivityksen mukaisia päivityksiä askeleittain kunnes ollaan saatu kurottua aika kiinni. Tässä tulee siis muistaa, että säännöllisessä päivitystavassa päivitetään aina samalla päivitysajalla, joten

päivitysajan ollessa alle puolet 0,3 sekunnista, joudutaan tekemään useampi päivitys kerralla. Tällöin siis jätetään vaste palauttamatta, kunnes ollaan saatu kaikki päivitykset tehtyä, mikä säästää hieman tehokkuutta, ja ei anneta käyttäjälle vanhentunutta vastetta.

Uudessa mallissa esiintyy kuitenkin puutteita, jos viivästys ei ole väliaikainen ja johtuu laitteiston suorituskyvystä (Fiedler 2004). Jos päivityksen laskemisessa menee kauemmin kuin mitä päivitysaika on, niin ei ole mahdollista saada viivästystä kurottua umpeen vaan sen sijaan viivästys vain kasvaa (Fiedler 2004). Käyttäjälle tämä mahdollisesti näkyisi pelin hyytymisenä, koska suoritus olisi vain päivitysvaiheessa eikä koskaan piirtäisi pelaajalle kuvaa tai palauttaisi muutakaan vastetta. Tämän takia käytettävä päivitysaika tulisi valita tarpeeksi suureksi, jotta päivitys ehditään tekemään (Fiedler 2004). Voidaan myös vaihtoehtoisesti asettaa maksimimäärä, kuinka monta päivitystä voidaan kerralla tehdä, jotta käyttäjä saa edes välillä vasteen. Tällöin kuitenkin peli päivittyy harvemmin ja vaihtelevin välein, mikä käyttäjälle näkyy pelin hidastumisena (Witters 2009). Tällöin kuitenkin säilytetään toistettavuus ja ratkaisu toimii hyvin väliaikaisten viivästymisten ratkaisussa.

Mallin esittelyissä ei tuoda vahvasti esille, toistetaanko myös syötteen kerääminen päivitysvaiheen toiston yhteydessä. Kuitenkin Fiedler (2004) esittelemässä lähdekoodissa ei ollenkaan mainita syötettä, mikä saattaa antaa ymmärtää, että syöteen keruu on osa käytettävää päivitysvaihetta. Toisaalta oman tulkintani mukaan syötteen keruuseen ei tehdä mallissa oleellisia muutoksia ja siten on pelin kehittäjän itsensä valittavissa.



Kuvio 3. Viivästymisestä toipuva pääsilmukkamalli.

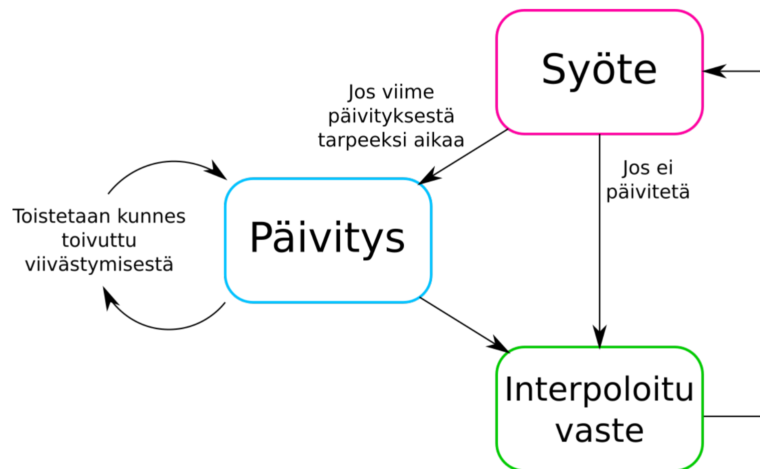
Huomioitavaa on, että alkuperäisessä lähdekoodiversiossa (Fiedler 2004) ei pysähdytty odottamaan määräaikaa silmukan kierrosten välillä, jossa päivitettävä määrä hoidettin ajan tarkkailun avulla päivitysvaiheessa. Kuitenkin lopputuloksena päivitys ohitettiin, jos edellisestä päivityksestä ei ollut kulunut tarpeeksi aikaa. Vaste taas annettiin joka silmukan kierroksella, vaikka se olikin oman tulkintani mukaan turhaa, koska esimerkiksi näytölle piirrettävään kuvaan ei tule muutoksia päivityksen puutteessa. Oletettavasti turha vasteen antaminen tosin ei ollut esittelyssä oleellista, koska esitetty lähdekoodi oli vain välivaihe ennen luvussa Interpoloiva malli esiteltävää mallia varten, jossa vasteen antaminen ilman päivitystä ei jää turhaksi.

3.2 Interpoloiva malli

Luvussa Toistuvasti päivitys -malli esitelty pääsilmukan malli on vain esiaste Fiedler (2004) esittämälle mallille. Lopullisessa versiossa pyritään poistamaan kokonaan riippuvuus päivitystiheyden ja vasteen antamistiheyden väliltä (Fiedler 2004). Käytännössä halutaan mahdollistaa, että pelin tilaa voidaan päivittää eri tahdissa kuin, mitä näytölle piirtäminen tapahtuu (Fiedler 2004). Esitetyssä uudessa mallissa on jopa mahdollistaa piirtää kuva useammin kuin pelitilaa edes päivitetään. Pääsilmukan perusmallissa tämä ei ole mitenkään mahdollista, koska jos pelin tilaa ei päivitetä, niin myöskään piirrettävä kuva ei muutu. Tämän takia perusmallissa ei voida päivittää kuvaa näytöllä useammin kuin, mitä pelin tilaa päivitetään. Esitelty uusi malli ei ole kuitenkaan samalla tavalla rajoittunut kuin pääsilmukan perusmalli.

Uusi malli (katso kuva 4) pohjautuu luvussa Toistuvasti päivitys -malli esiteltyyn malliin ja se lähtee liikkeelle ideasta, että näytön virkistystaajuus ei ole luonnostaankaan sama kuin, mitä peliä päivitetään (Fiedler 2004). Esimerkiksi näyttö saattaa päivittyä 60 kertaa sekunnissa, mutta peli päivittyy vain 40 kertaa sekunnissa. Tämän seurauksena on hyvin epätodennäköistä, että näytön päivittyminen osuisi samaan aikaan kuin, mitä peli päivittyy. Näytölle saattaisi siis piirtyä kuva pelin päivitysten välillä, jolloin kuva tulee hieman myöhässä (Fiedler 2004).

Tätä varten uudessa mallissa pyritään korjaamaan vanhaa pelitilaa huomioimalla päivityksen ja kuvan piirron välinen aika (Fiedler 2004). Koska kuvan piirto osuu päivityskertojen väliin,



Kuvio 4. Interpoloivan pääsilmukan rakenne. Huomioitavaa, että malli ei ota voimakkaasti kantaa syötteen sijaintiin, ja sen voi vaihtoehtoisesti laittaa osaksi päivitystä.

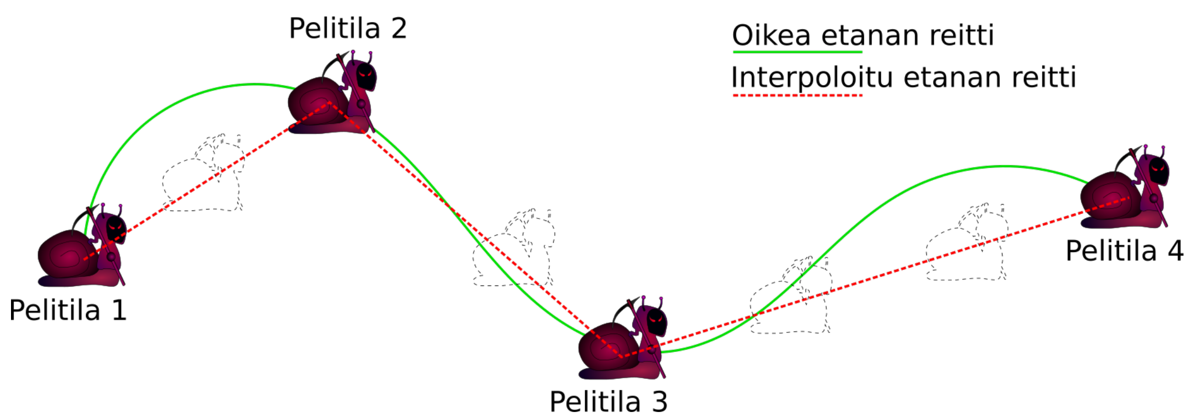
niin voidaan ajanhetkestä käyttää lukuarvoa nollan ja yhden väliltä (Fiedler 2004). Esimerkiksi, jos kuvan piirtäminen osuu puoleen väliin, niin lukuarvo on 0,5. Luvusta 0,5 voidaan ajatella, että viimeksi saatua pelitilaa tulee päivittää puolet seuraavan pelitilan muutoksista. Tätä varten olemme laskeneet ennakkoon seuraavan pelitilan, johon vertaamme viimeksi saatua pelitilaa (Fiedler 2004).

Interpoloimalla näiden viimeksi saadun ja tulevan pelitilan välillä pystymme laskemaan piirtämistä varten arvion, minkälainen pelitila kuuluisi olla piirrettävällä hetkellä (Fiedler 2004). Kuvassa 5 on esimerkki interpolaatiosta. Yksinkertaisella lineaarisella interpoloinnilla arvolla 0,5 saamme tarpeeksi hyviä tuloksia pelitilasta, mikä olisi laskettujen pelitilojen keskikohdassa. Esimerkiksi, jos tiedämme kappaleen olevan paikassa A, mutta seuraavassa pelitilassa se on paikassa B, niin ajanhetkeä 0,5 vastaavassa pelitilassa kappaleen sijainti saadaan laskemalla kaavalla:

$$valisijainti = sijaintiA + 0,5 * (sijaintiB - sijaintiA)$$

Edellä kuvattu lineaarinen interpolointi on kohtuullisen helppoa tehdä ja se on tarpeeksi hyvä kuvaa piirrettäessä (Fiedler 2004). Tarkkuudesta ei tarvitse juurikaan murehtia, koska pelitilan päivittäminen tapahtuu todella usein, jolloin interpoloitava väli on usein sekunnin sadasosien luokkaa. Interpolointi kahden pelitilan välillä on myös huomattavasti kevyempää suorituskyvylle, kuin pelitilan laskeminen kokonaan uusiksi. Tosin tulee huomioida, että

interpoloitu tulos on tarkoitettu vain piirtämistä varten, eikä sitä hyödynnetä pelilogiikassa. Tällöin pystymme säilyttämään säännöllisesti päivittyvän päivitystavan hyödyt, mutta onnistumme antamaan käyttäjälle vasteen niin usein kuin vain suorituskyky sallii (Fiedler 2004). Säännöllisesti päivittyvän päivitystavan ongelma olikin, että se skaalautui huonosti suorituskyvyn suhteen, mikä tässä mallissa on onnistuttu korjaamaan.



Kuvio 5. Esimerkki kuinka etanan interpoloitu sijainti vaihtelee pelitilojen välillä. Yhtenäinen viiva kuvaa etanan kulkemaa reittiä ja värilliset etanat päivityksessä laskettuja sijainteja. Katkoviiva viiva kuvaa interpoloitua reittiä ja katkoviivaiset etanat kuvaavat vasteessa käytettyä etanan arvioituja sijainteja. Käyttäjälle näkyy vain katkoviivaiset etanat, kun taas pelilogiikka hyödyntää värjättyjä etanoita. Tämä toimii kunhan muutokset ovat suhteellisen pieniä eli päivitysaika on tarpeeksi lyhyt.

Toisaalta mallissa on kuitenkin ongelmallista, että siinä hyödynnetään tulevaa pelitilaa. Tällöin ei voida luonnollisestikaan tietää, minkälaisia syötteitä käyttäjä antaa tulevaa päivitystä varten, jolloin joudutaan tyytymään käyttämään vanhoja syötteitä tulevaa pelitilaa laskettaessa. Tämän seurauksena saadut syötteet vaikuttavat vasta seuraavassa päivityksessä, jolloin muodostuu viivettä syötteen ja päivityksen välille, kun syötteet vaikuttavat aina yhden päivityksen verran myöhässä. Tämä ei kuitenkaan ole iso ongelma, koska peleissä on lyhyet päivitysvälit, jolloin viive on sadasosa sekuntien luokkaa, mikä ei ole useimmissa peleissä ongelma.

Toisaalta vaihtoehtoisesti voidaan myös saada piirrettävä kuva ekstrapoloimalla (Käyttäjätunnus Animmaniac 2016). Ekstrapoloidessa lasketaan tuleva tila vanhojen tilojen avulla, kun taas interpoloidessa lasketaan tiedettyjen tilojen välissä oleva tila ympärillä olevien tilo-

jen avulla. Ekstrapolointia käyttämällä vältetään käyttäjän syötteiden käsittelyn viive, koska ekstrapoloidessa ei tarvita syötteitä tulevan pelitilan laskemisessa.

Kuitenkin selkeänä haittapuolena on, että ekstrapoloinnissa oletetaan muutosten jatkuvan samanlaisina. Esimerkiksi jos auto liikkuu 100 kilmotriä tunnissa, niin osaamme laskea auton tulevan sijainnin, jos liike pysyy samana. Kuitenkin laskettu tulos on virheellinen, jos auton nopeus tai suunta muuttuu. Vastaavasti ekstrapoloimalla saatu piirrettävä kuva pelin tilasta voi olla virheellinen, jos nopeus tai suunta muuttuvat äkillisesti (Käyttäjätunnus Animmaniac 2016). Tällöin seuraavan pelitilan päivityttyä saattaa kuvassa olla huomattavia muutoksia, koska ekstrapoloitu kuva ei vastaakaan päivittynyttä pelitilaa.

Vastaavasti ekstrapolointi ei myöskään välttämättä havaitse törmäyksiä, jolloin saatetaan piirtää kappale jonkin törmättävän kappaleen päälle (Käyttäjätunnus Animmaniac 2016). Pelin logiikan kannalta tämä ei ole kuitenkaan ongelma, koska oikeaa pelitilaa laskettaessa ekstrapoloidulla kuvalla ei ole merkitystä, mutta se voi olla pelaajalle visuaallisesti hieman häiritsevää. Pienillä päivitysväleillä kuitenkin erot ovat suhteellisen pieniä, jolloin ekstrapoloinnista johtuva hetkellinen virheellinen kuva ei ole kovinkaan huomattava ongelma.

Kuitenkin sekä interpoloivan tai ekstrapoloivan pääsilmukan haasteena on toteutusten työläys, sillä tulee toteuttaa useille erityyppisille arvoille niiden interpolointi. Lukuarvoille ja vektoreille interpolointi on helppoa toteuttaa, mutta myös 3d mallien dynamiikkaa onnistuu vaikkakin hieman haastavasti. Tällöin tulee orientaatio tallentaa kvaternioina ja käyttää spherical linear interpolationia (eli Slerp). (Fiedler 2004)

Kaiken kaikkia esitetty malli onnistuu toistettavuudessa ja samaan aikaan tarjoaa vasteen annon niin usein, kuin vain suorituskyky mahdollistaa. Tämä ratkaisee yhden merkittävimmistä säännöllisen päivitystavan ongelmista. Malli vaatii kuitenkin hieman ylimääräistä työpanostusta interpoloinnin takia, mutta mielestäni se ei ole kuitenkaan liian suuri vaatimus. Todellisena haittana interpolaatiolle on kuitenkin pieni viive syötteen käsittelyssä, mikä saattaa olla joissakin peleissä selkeästi epätoivottavaa. Lisäksi peli voi viedä enemmän muistia, jos interpolointia varten säilytetään yhtä aikaa kahta erillistä pelitilaa, joiden välillä interpoloidaan.

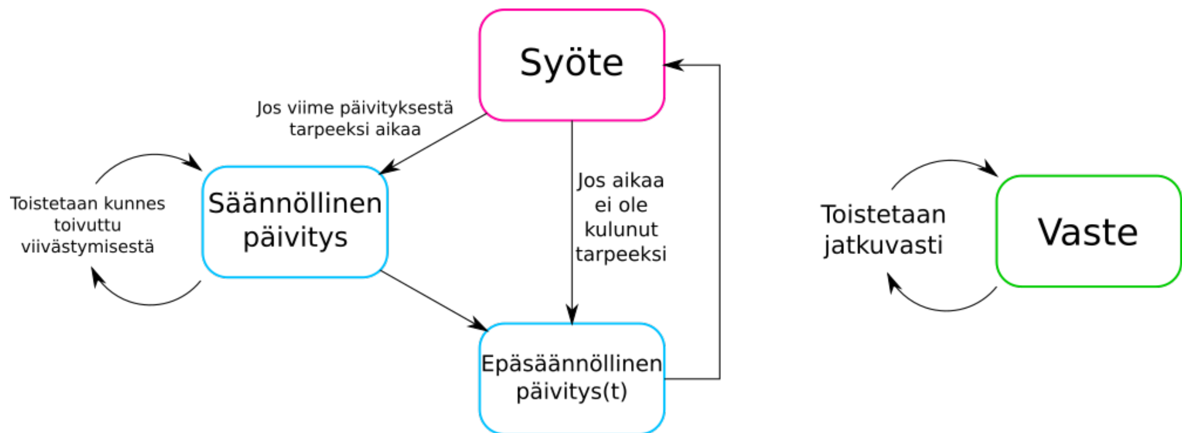
3.3 Säännöllinen ja epäsäännöllinen päivitys erikseen

Luvussa Interpoloiva malli esitellyn mallin tavoin myös Valente, Conci ja Feijó (2005) esittää mallin, jossa säilytetään deterministisyys ilman, että vasteen palauttamistiheys rajoitetaan samaksi kuin säännöllisesti tapahtuvalla päivityksellä. Tämä on toteutettu jakamalla päivitykset kahteen ryhmään eli säännöllisiin päivityksiin ja epäsäännöllisiin päivityksiin (Valente, Conci ja Feijó 2005). Säännöllisiin päivityksiin on tarkoitus laskea sellaiset tehtävät, jotka ovat pelin deterministisyyden kannalta välttämättömiä. Tällaisia ovat monet pelilogiikkaan liittyvät osat kuten fysiikkamoottori ja tekoälyä käyttävät vastustajat. Toisaalta pelikokemus ei välttämättä muutenkaan parane, vaikka nämä tehtävät suoritettaisiinkin mahdollisimman usein (Valente, Conci ja Feijó 2005). Tärkeintä on kuitenkin, että ne suoritetaan tarpeeksi usein, mikä säännöllisessä päivitystavassa voidaan taata.

Epäsäännöllisiin päivityksiin taas on tarkoituksena lukea sellaiset toiminnot, jotka eivät vaikuta pelin deterministisyyteen. Hyötyäksemme jaottelusta, tulisi epäsäännöllisesti päivitettävien tehtävien olla sellaisia, jotka hyötyvät mahdollisesti suuremmasta päivitystiheydestä (Valente, Conci ja Feijó 2005). Nämä ehdot täyttää monet pelilogiikasta irralliset graafiset tehtävät. Esimerkiksi autopelissä nopeusmittariin nopeuden laskeminen on tällainen, koska peli ei oletettavasti käytä itse samaa mittaria. Suorituskyvyllisesti raskaampana esimerkkinä taas voisi olla pilvien liikkeiden tai veden aaltoilun simulointi, jotka molemmat hyötyvät suuremmasta päivitystiheydestä tarjoamalla käyttäjälle sulavammin päivittyvää kuvaa.

Valente, Conci ja Feijó (2005) esittämästä pseudokoodista ja kaaviosta käy ilmi, että käytännössä esitetty malli toimii siten, että ensiksi kerätään syöte, jonka jälkeen tarkastetaan kumpi päivityksistä suoritetaan ensin. Jos viimeisestä säännöllisestä päivityksestä on kulunut tarpeeksi aikaa, niin sitten tehdään säännöllinen päivitys, jonka jälkeen tehdään epäsäännöllinen päivitys. Jos taas viime päivityksestä ei ole kulunut tarpeeksi aikaa, niin tehdään suoraan epäsäännöllinen päivitys. Paremmin suoritusjärjestys ilmenee kuvasta 6. Huomioitavaa on myös, että esitelty malli käyttää säännöllisessä päivityksessä ideaa, jossa säännöllinen päivitys toistetaan kunnes ollaan kurottu jälkeen jääty aika umpeen (Valente, Conci ja Feijó 2005). Tätä käsiteltiin tarkemmin luvussa Toistuvasti päivitys -malli. Lisäksi mallissa on erotettu vasteen palauttaminen omaan säikeeseensä (Valente, Conci ja Feijó 2005). Toisaalta vasteen palauttamisen erottaminen omaan säikeeseen ei ole mallissa sinänsä oleellista

ja vasteen palauttaminen voitaisiin sijoittaa myös heti epäsäännöllisen päivityksen jälkeen. Tämän takia aihetta ei käsitellä tässä luvussa vaan luvussa Osien puhdas irrottaminen omaan säikeeseen yhdessä muun monisäikeisyyden kanssa.



Kuvio 6. Mallissa on eroteltu epäsäännöllinen ja säännöllinen päivitys erilleen. Lisäksi vaste on omassa säikeessään, mutta sen voi sinänsä sijoittaa tapahtumaan myös epäsäännöllisen päivityksen jälkeen.

Luokittelu säännöllisiin ja epäsäännöllisiin tehtäviin onnistuu säilyttämään säännöllisestä päivitystavasta tutun deterministisyyden tarjoten samalla ainakin osittain paremman pelikokemuksen suuremman päivitystiheyden muodossa. Puutteena on kuitenkin se, että deterministisyyden säilyttämiseksi ei voida päivittää kaikkea mahdollisimman usein vaan joudutaan tyytymään vain osaan päivitettävistä asioista. Tämän suhteen luvussa Interpoloiva malli esitetty malli on parempi, koska siinä pystytään antamaan kaikista pelimaailman kohteista vaste kuinka usein tahansa. Toisaalta tämä vaste on interpoloinnin tulos, eikä se siten ole välttämättä tarkka, kun taas tässä luvussa esitelty malli saa vasteen suoraan päivityksen tuloksena. Tosin käyttäjä ei välttämättä edes huomaa toteutuksissa ilmeneviä pieniä eroavaisuuksia.

Interpoloinnin toteuttaminen tuo lisää työtä, kun taas tämän luvun mallissa lisää työtä tulee säännöllisten ja epäsäännöllisten tehtävien erottelusta ja toteutuksen suunnittelusta. Suorituskykyjä taas on haastavaa verrata mallien välillä, koska interpoloinnin työläys on tapauskohtaista kuten myös itse päivityksen työläys. Joissakin peleissä myös saattaa helpommin löytyä tehtäviä, jotka voidaan sijoittaa epäsäännölliseen päivitykseen ja siten tarjoaa parempia tuloksia. Joissakin peleissä taas tällaisten tehtävien tunnistaminen voi olla haastavaa tai

niitä ei välttämättä edes ole.

4 Rinnakkaistaminen yleisesti

Rinnakkaislaskennassa (eng. parallel computing) laskettava ongelma jaetaan pienempiin osiin, jotta osat voitaisiin laskea erikseen (Barney ym. 2010). Tällöin voidaan käyttää useita suorittimia tai suorittimen ytimiä samanaikaisesti, jolloin ongelma ratkeaa mahdollisesti nopeammin. Esimerkiksi kaksiytimisellä suorittimella voitaisiin teoriassa ratkaista sama ongelma kaksi kertaa nopeammin kuin ilman rinnakkaistusta. Rinnakkaislaskenta on sikäli oleellista, koska suorittimien nopeuksien kasvattaminen on käymässä valmistajille yhä haastavammaksi (Barney ym. 2010). Sen sijaan on helpompaa tehdä moniydinsuoritin, jolloin rinnakkaislaskennan avulla saadaan kuitenkin sama tai jopa suurempi nopeushyöty kuin mitä yksittäisen suorittimen nopeuden kasvattamisella voitaisiin edes saada (Barney ym. 2010). Lisäksi koska nykypäivänä monet uudet suorittimet ovat moniytimisiä, niin on vain järkevää hyödyntää rinnnakkaistusta sovellusta tehtäessä.

Monisäikeisyys taas on yksi tapa toteuttaa rinnakkaisuus ohjelmoinnissa. Säikeiden avulla ohjelma voi suorittaa useita eri tehtäviä samanaikaisesti (Barney ym. 2010). Tehtäviä varten säikeillä on oma muistinsa, mutta lisäksi pääsy ohjelman yhteiseen muistiin, mitä voidaan myös käyttää hyväksi säikeiden välisessä synkronisaatiossa (Barney ym. 2010). Yhteistä muistia käytettäessä tulee kuitenkin pitää huolta, etteivät eri säikeet käsittele samaa muistialuetta samanaikaisesti (Barney ym. 2010). Yhteisen muistin hallintaa ja säikeiden synkronointia lukuunottamatta säikeden ohjelmoiminen ei poikkea juurikaan tavallisesta ohjelmoinnista. Ohjelmoija voi käyttää säikeitä joustavasti ja täysin kontrolloida säikeen tekemisiä (Damon 2011). Laajan kontrollin ansiosta monisäikeisyys soveltuu hyvin peleihin, joissa voi olla hyvinkin erilaisia tehtäviä, joita halutaan suorittaa samanaikaisesti. Kuitenkin käyttöä rajoittaa säikeiden välisen synkronoinnin ja muistinkäsittelyn haasteet, jotka ovat rinnakkaistuksessa muutenkin yleinen ongelma.

Rinnakkaisuus rikkookin helposti ohjelman deterministisyyden (käsitelty luvussa Pelin deterministisyys), koska useita eri tehtäviä tehdään samanaikaisesti eikä niiden suoritusjärjestyksestä voida olla varmoja. Esimerkiksi, jos kaksi eri säiettä tallentavat johonkin muuttujaan eri arvon, niin vain jälkimmäinen niistä jää voimaan. Koska säikeet toimivat samanaikaisesti, niin kumpi tahansa säie voi ehtiä tallentamaan arvon ensin. Säikeiden nopeus voi myös vaih-

della suorituskertojen välillä ja siksi eri kerralla eri säie voi ehtiä ensin. Tämän seurauksena muuttujaan voi jäädä eri arvo voimaan eri suorituskerroilla, ja siten deterministisyys ei toteudu. Riippumattomuudella ja hyvällä säikeiden välisellä synkronoinnilla voidaan onnistua välttämään tällaiset ongelmat, mutta se vaatii ylimääräistä huolellisuutta ohjelmoijalta.

Rinnakkaistaessa täytyy myös muistaa, että säikeistys ei tarjoa suoraan rinnakkaisuuden nopeushyötyjä, jos käytettävä rauta ei tue rinnakkaistusta ylipäätään. Esimerkiksi, jos suoritin ei ole moniydinsuoritin, niin se ei pysty suorittamaan useita säikeitä oikeasti samanaikaisesti, jolloin suoritusaika ei parane. Itseasiassa suoritusaika voi säikeistyksen myötä jopa heikentyä tällaisessa tilanteessa. Säikeiden hallinta, kuten niiden luominen ja tuhoaminen, ovat kohtuullisen raskaita tehtäviä ja siten säikeistys voi heikentää pelin suorituskykyä (Tulip, Bekkema ja Nesbitt 2006). Tätä varten rinnakkaistaessa tulisi rinnakkaistettavan tehtävien olla sopivan hienojakoista.

4.1 Hienojakoisuus

Hienojakoisuudella (eng. granularity) tarkoitetaan, kuinka pieniin osiin rinnakkaistettava tehtävä on jaettu (Tulip, Bekkema ja Nesbitt 2006). Jos tehtävä jaetaan liian pieniin osiin, niin saadaan paljon pieniä tehtäviä. Tällöin kuitenkin joudutaan luomaan todella paljon säikeitä, jolloin säikeiden hallintaan voi kulua enemmän suorituskykyä, kuin mitä rinnakkaistuksesta edes saadaan (Tulip, Bekkema ja Nesbitt 2006). Lisäksi ei ole muutenkaan tarpeellista jakaa tehtävää tuhansiin osiin, jos suorittimella on huomattavasti vähemmän ytimiä käytössä. Käytännössä esimerkiksi tekoäly vastustajien päivittäminen voidaan suorittaa ryhmittäin eli jaetaan kaikki tekoäly vastustajat ryhmiin. Kutakin ryhmää varten luodaan sitten oma säie, joka huolehtii kaikkien ryhmän jäsenten tekoälyn päivittämisestä.

Toisaalta jos tehtävä jaetaan liian suuriin osiin, niin osien suoritus ei välttämättä jakaudu tasaisesti suorittimen ytimien välille (Tulip, Bekkema ja Nesbitt 2006). Tällöin saattaa tulla tilanteita, joissa yksi säie suorittaa omaa osaansa, kun muut säikeet ovat jo valmiita. Tällöin muut säikeet joutuvat odottamaan tätä yhtä säiettä. Tätä varten tulee jakaa suoritettava kuorma tasaisesti ytimien välillä, jotta kaikki tehtävän osat tulisivat suoritetuksi lähes samaanaikaan. Kuorman jakaminen on helpompaa, jos osat ovat pieniä, jolloin mahdollinen

odotusaika jää pieneksi (Tulip, Bekkema ja Nesbitt 2006). Liian pienet osat taas aiheuttavat jo todetusti turhaa säikeiden luontia, mitä myös yritetään välttää. Tämän takia on tärkeää löytää tehtävälle oikea jako sopivan kokoisten osien saamiseksi.

4.2 Riippuvuudet rinnakkaistettaessa

Rinnakkaistettaessa olisi toivottavaa, että rinnakkaistettavien osat olisivat toisistaan riippumattomia (El Rhalibi, Merabti ja Shen 2006). Jos rinnakkaistettavat osat ovat toisistaan riippuvia, niin tämän seurauksena voi tulla epätoivottua ja epädeterminististä käytöstä. Tämän takia rinnakkaistettaessa olisi hyvä olla tietoinen eri osien välisistä riippuvuuksista. Riippumattomuuden tarkasteluun El Rhalibi, Merabti ja Shen (2006) esitteleekin tarkastelutavan, jossa tehdään riippuvuusanalyysi (eng. dependency analysis), jolla tunnistetaan, mitkä osat ei voi suorittaa samanaikaisesti (El Rhalibi, Merabti ja Shen 2006).

Yhtenä merkkinä riippuvuudesta on, jos tehtävä tuottaa dataa toiselle tehtävälle (El Rhalibi, Merabti ja Shen 2006). Esimerkiksi jos tehtävä A tuottaa listan lukuja, ja tehtävä B laskee tehtävässä A tuotettujen lukujen keskiarvon, niin luonnollisestikaan keskiarvoa ei voida laskea ennen kuin tehtävä A on edes tuottanut tarvittavat luvut. Tästä El Rhalibi, Merabti ja Shen (2006) käyttää englannin kielistä termiä flow dependence.

Toisena merkkinä riippuvuudesta on, myös käänteinen tapaus, jossa jälkimmäisenä suoritettu tehtävä vaikuttaisi aiemmin tehdyn tehtävän syötteeseen (El Rhalibi, Merabti ja Shen 2006). Tällä tarkoitetaan, että jos esimerkiksi lasketaan listan lukujen keskiarvo, niin voimme vapaasti muokata listaa keskiarvon laskemisen jälkeen, mutta jos muokkaamme listaa ennen keskiarvon laskemista, niin laskettu keskiarvo muuttuu. Riippuvuus siis rajoittaa, että keskiarvon laskeminen tulee suorittaa ennen kuin listaa voidaan muokata, mikä estää rinnakkaisuuden toteuttamisen. Tästä El Rhalibi, Merabti ja Shen (2006) käyttää englannin kielistä termiä antidependence.

Kolmantena merkkinä riippuvuudesta on, jos kaksi eri tehtävää tallentaa eri arvon samaan muistialueeseen (El Rhalibi, Merabti ja Shen 2006). Tällöin rinnakkaistettaessa ei voida tietää kumpi arvo jää voimaan, mikä on epätoivottu ilmiö ja rikkoo muun muassa determinististisyyden. Tästä El Rhalibi, Merabti ja Shen (2006) käyttää englannin kielistä termiä output

dependence.

Neljäntenä merkkinä riippuvuudesta on, jos kaksi eri tehtävää käsittelevät samaa tiedostoa (El Rhalibi, Merabti ja Shen 2006). Tämä ei olisi ongelma, jos molemmat vain lukisivat tiedostoa. Kuitenkin jos toinen tehtävä muokkaa kyseistä tiedostoa, niin silloin voi ilmetä ongelmia. Tästä El Rhalibi, Merabti ja Shen (2006) käyttää englannin kielistä termiä I/O dependence.

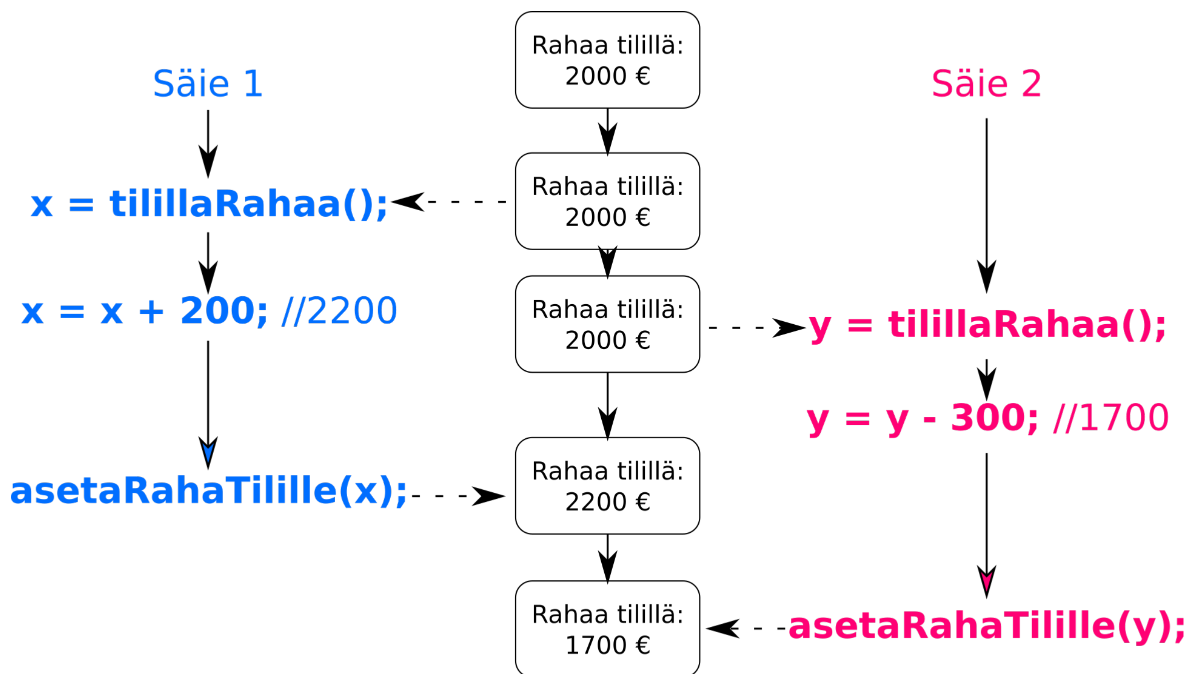
Viidentä merkkinä El Rhalibi, Merabti ja Shen (2006) mainitsee tuntemattoman riippuvuuden (eng. unknown dependence), joka saadaan, jos tehtävä käyttää epäsuorasti saatua dataa. Itse voisin kuvitella tämän tarkoittavan, että jos esimerkiksi jostain isosta tietorakenteesta pyydetään jotain dataa käyttöön, niin ei välttämättä tiedetä, mikä aiempi tehtävä on tuottanut kyseisen datan.

Näiden merkkien avulla voidaan löytää rinnakkaistusta estävät riippuvuudet ja siten myös havaita, mitkä tehtävät voidaan rinnakkaistaa. Löydettyjen riippuvuuksien pohjalta voidaan myös luoda riippuvuuskaavio, jolla nähdään muun muassa, missä järjestyksessä tehtävien on tapahduttava, ja mitkä tehtävät voidaan rinnakkaistaa (El Rhalibi, Merabti ja Shen 2006). Kuitenkin myös toisistaan riippuvia tehtäviä voidaan jonkin verran mahdollisesti rinnakkaistaa, jos synkronoinnista huolehditaan. Tällöin kuitenkin joudutaan ehkä luopumaan deterministisyydestä, mutta kuitenkin voidaan synkronoinnilla välttää ohjelman toiminnan rikkovat ongelmat.

4.3 Synkronointi

Eräs rinnakkaisuuden merkittävimpiä ongelmia on, jos eri säikeet käsittelevät samoja resursseja samaan aikaan muokaten niitä. Esimerkiksi kuvassa 7 näkyy, kuinka samanaikaisten tilisiirtojen seurauksena tilille jää virheellinen määrä rahaa, koska molemmat säikeet muokkasivat vapaasti samoja resursseja. Tämän perusteella on selvää, että useiden säikeiden ei voi antaa vapaasti muokata samoja resursseja, jotta vältyttäisiin virheelliseltä käytökseltä. Ongelmien välttämiseksi säikeiden välillä tulee käyttää synkronisaatiota esimerkiksi lukkoja käyttämällä (Barney ym. 2010). Synkronisaatiolla voidaan varmistaa, että eri säikeet suorittavat tapahtumia toimivassa järjestyksessä eivätkä muokkaa muiden säikeiden käyttämiä

resursseja kesken kaiken.



Kuvio 7. Kaaviossa näkyy kuinka pankkisiirto menee helposti pieleen, jos kaksi eri säiettä muokkaa tiliä yhtä aikaa ilman synkronointia säikeiden välillä. Tilille on tarkoituksena lisätä 200 euroa ja vähentää 300 euroa. Kuitenkin vain jälkimmäinen muutos jää voimaan, jolloin tilin omistaja kärsii 200 euron tappiot.

4.3.1 Lukot

Säikeistys toimii hyvin tilanteissa, joissa eri tehtävien välillä on vain vähän riippuvuuksia, koska tällöin tehtävistä muodostetut säikeet eivät käsittele samoja resursseja niin paljoa (El Rhalibi, Merabti ja Shen 2006). Harvat tapaukset taas voidaan käsitellä säikeiden välisellä synkronoinnilla esimerkiksi lukkoja (eng. lock) käyttämällä (Damon 2011). Lukoilla tarkoitetaan säikeistyksessä käytettävää rakennetta, jossa jokin resurssi (esim. osa muistia) merkitään lukituksi, kun säie käyttää sitä (Jones 2007). Tällöin muut säikeet eivät voi käyttää lukittua resurssia ennen kuin lukko avataan (Jones 2007). Lukkojen käyttö kuitenkin heikentää helposti suorituskykyä, jos niitä joudutaan käyttämään liian paljon, koska muut säikeet joutuvat odottamaan lukon avautumista (Damon 2011). Tämän takia pitäisi pyrkiä säikeistämään niin, että lukkoja tarvittaisiin mahdollisimman vähän.

Lukkojen käytössä on lisäksi ongelmana, että ohjelman suoritus voi jäädä pysyvästi jumiin, jos kaksi eri säiettä lukitsevat toistensa tarvitsemat resurssit eli tapahtuu lukkiutuma (eng. deadlock) (Silverman 2008). Esimerkiksi jos samaanaikaan säie 1 lukitsee resurssin A ja säie 2 lukitsee resurssin B, niin tulee lukkiutuma, jos molemmat haluavat lisäksi toistensa resurssit. Tällöin kumpikin säie odottaa toisen säikeen resurssia ennen kuin jatkaa ohjelman suoritusta, jolloin kumpikaan säie ei voi jatkaa ohjelman suoritusta. Eräs ratkaisu tähän ongelmaan on määrätä lukitseminen toimimaan niin, että resurssit lukitaan aina samassa järjestyksessä (Silverman 2008). Esimerkiksi ensin yritetään lukita resurssi A, jonka jälkeen resurssi B, jonka jälkeen resurssi C ja niin edelleen. Tällöin aiemman esimerkin tapauksessa molemmat säikeet yrittäisivät lukita ensin resurssin A ja vasta sen jälkeen resurssin B, jolloin lukkiutumista ei pääse tapahtumaan.

4.3.2 Transaktiomuisti

Transaktiomuisti (eng. transactional memory) on toinen ratkaisu, jota voidaan käyttää lukkojen sijaan rinnakkaisuuden ongelmien välttämiseksi (Silverman 2008). Käytännössä transaktiomuistilla säie laskee kaikki muutokset omaan muistiinsa eikä aseta muutoksia voimaan ennen kuin vasta lopuksi (Jones 2007). Ennen muutosten asettamista voimaan kuitenkin säie tarkistaa, onko mikään käytetty resurssi ehtinyt muuttua tehtävän suorituksen aikana (Jones 2007). Jos mikään resurssi ei ole muuttunut, niin muutokset voidaan asettaa voimaan kaikki kerralla (Jones 2007). Jos taas jokin resurssi on muuttunut, niin omat muutosehdotukset ovat vanhentuneita ja ne hylätään ja aloitetaan tehtävän suoritus alusta (Jones 2007).

Koska transaktiomuisti ei lukitse resurseja, niin lukkiutumisia ei voi tapahtua, ja siten vältytään tarkalta lukkojen suunnittelulta ja saadaan yksinkertaisempi toteutus (Silverman 2008). Transaktiomuistin yksinkertaisuus voi olla lukkoja parempi silloin, jos ennen tehtävän suoritusta ei voida tietää, mitkä kaikki resurssit tarvitsee lukita tehtävää varten (Silverman 2008). Tällöin säikeet eivät voi lukita resursseja samassa järjestyksessä, jolloin lukkiutumisen välttäminen käy haastavammaksi lukkoja käytettäessä (Silverman 2008).

Transaktiomuistin käyttö saattaa kuitenkin heikentää ohjelman suorituskykyä lukkojen käyttöä enemmän, koska tehtävät voidaan joutua suorittamaan uudestaan (Silverman 2008). Li-

säksi transaktiomuistin toteutus itsessään vie suorituskykyä ja siten heikentää rinnakkaistuksesta saatavaa hyötyä (Silverman 2008). Erityisesti jos transaktiomuisti on toteutettu ohjelmiston puolella laitteiston sijaan, niin suorituskyky saattaa kärsiä (Silverman 2008).

4.3.3 Etenemisen estäminen

Yksi ratkaisu synkronoinnille on käyttää estettä (eng. barrier), jonka kohdalla odotetaan kaikkien tehtävien valmistumista (Barney ym. 2010). Kun esteeseen liittyvät tehtävät ovat valmistuneet, niin toteutetaan synkronointi kaikkien tehtävien välillä ennen kuin jatketaan ohjelman suoritusta uusilla tehtävillä (Barney ym. 2010). Estämällä ohjelman eteneminen esteen ohi, voidaan varmistaa, että esteen jälkeisiä tehtäviä varten suoritettavat tehtävät ovat varmasti suoritettu.

Toisaalta esteen kohdalla voidaan myös yhdistää eri tehtävien tuloksia suuremmaksi kokonaisuudeksi. Esimerkiksi fysiikkamoottorissa voitaisiin päivittää kaikkien kappaleiden sijainti erillisesti, mutta esteen kohdalla yhdistettäisiin tulokset tarkastamalla tapahtuiko törmäyksiä kappaleiden välillä. Törmäysten perusteella taas voitaisiin laskea törmänneiden kappaleiden sijainnit uudelleen ja tarkemmin. Tällöin esteen tarkoituksena on suorittaa tehtäviä rinnakkain niin paljon kuin pystytään, ja sen jälkeen suoritetaan sellaiset tehtävät, joita ei voida rinnakkaistettuna suorittaa tehokkaasti.

4.3.4 Viestien lähetys säikeiden välillä

Säikeet voivat kommunikoida toistensa kanssa joko yhteisen muistin tai varsinaisten viestien avulla (Barney ym. 2010). Kommunikoimalla säikeet voivat muun muassa kertoa toisilleen tarvittavia tietoja, jolloin tietoa lähettävä säie takaa, että tieto on turvallista luovuttaa eikä aiheuta ongelmia ohjelman suoritukselle. Tällöin myös voidaan paremmin hallita, milloin säikeet hyödyntävät toiselle säikeelle kuuluvaa tietoa, esimerkiksi asettamalla säie odottamaan tarvittavan tiedon valmistumista toisessa säikeessä (Barney ym. 2010).

4.4 Amdahlin laki

Vaikka rinnakkaistamalla saadaan hyviä tuloksia aikaiseksi, niin aina ei ole mahdollista tai kannattavaa toteuttaa rinnakkaistusta koko ohjelmalle. Osittaisen rinnakkaisuuden nopeushyötyä kuvaa Amdahl (1967) esittämä kaava, joka yleisesti tunnetaan Amdahlin lakina (eng. Amdahl's law). Kaava voidaan esittää muodossa

$$S = \frac{1}{F + (1 - F)/N}$$

jossa S on rinnakkaistuksesta saatava nopeushyöty, N on suorittimien lukumäärä ja F on rinnakkaistamattoman koodin osuus ohjelmasta (Tulip, Bekkema ja Nesbitt 2006). Koodin osuus ohjelmasta voidaan saada mittaamalla, kuinka paljon suorittimen aikaa ohjelman osa käyttää suhteessa muihin osiin.

Amdahlin laista käy ilmi, että rinnakkaistamaton koodi rajoittaa suurinta mahdollista nopeushyötyä (Tulip, Bekkema ja Nesbitt 2006). Esimerkiksi, jos koodista on rinnakkaistettu vain 10 %, niin suurin mahdollinen nopeus on vain 1,11 kertainen. Jos taas rinnakkaistetun koodin osuus on 90 %, niin suurin mahdollinen nopeus on jopa kymmenkertainen. Tällöin jo pelkästään kahdella suorittimella saavutetaan nopeudeksi 1,81 kertaisuus. Tämä on huomattavasti enemmän, kuin mitä 10 % rinnakkaistetulla osuudella ja äärettömällä määrällä suorittimia voitaisiin nopeudeksi koskaan edes saada.

Tämä osoittaa, että rinnakkaisuuden määrällä on huomattava merkitys siihen, että kuinka paljon ohjelman suoritus nopeutuu ja jo pieni määrä suorittimia riittää huomattavaan nopeuden kasvuun, jos rinnakkaistettu osuus on suuri (Tulip, Bekkema ja Nesbitt 2006). Saatu tulos ohjaakin pyrkimään rinnakkaistaa mahdollisimman suuren osan ohjelmasta, sillä vähäinen rinnakkaisuus käy melko merkityksettömäksi ohjelman suoritusnopeuden kannalta. Toisaalta tämän voi nähdä myös niin, että rinnakkaisuus kannattaa toteuttaa ohjelman osalle vain, jos sen osuus ohjelman suorituksesta on huomattavan suuri. Amdahlin lakia käytettäessä tulee myös muistaa, että se antaa vain teoreettisesti suurimman nopeuden eikä se huomioi rinnakkaistuksesta tulevaa ylimääräistä kuormitusta.

4.5 Rinnakkaistuksen tavat

Rinnakkaisuudelle on kaksi erilaista toisistaan poikkeavaa tapaa, jotka ovat datan rinnakkaistus (eng. data parallelism) ja tehtävien rinnakkaisuus (eng. task parallelism) (Damon 2011). Nämä tavat ovat toisistaan hyvin poikkeavia ja kummallekin on omat tilanteet, joissa tulee käyttää. Samassa ohjelmassa onkin hyvä hyödyntää molempia tapoja rinnakkaistaa, jotta saataisiin rinnaikkaistettua mahdollisimman suuri osa ohjelmasta.

4.5.1 Datan rinnakkaistus

Datan rinnakkaistuksella tarkoitetaan, että sama tehtävä tehdään usealle eri datajoukolle rinnakkaisesti (Damon 2011). Esimerkiksi sama laskuoperaatio voidaan tehdä kaikille taulukon arvoille rinnakkaisesti. Datajoukoille suoritettava tehtävän täytyy kuitenkin olla muista datajoukoista riippumatonta, jotta tehtävä voitaisiin toteuttaa useille datajoukoille samanaikaisesti (Tulip, Bekkema ja Nesbitt 2006). Esimerkiksi taulukon alkioiden summan laskeminen ei ole riippumaton tehtävä, koska siinä tarvitaan muiden taulukon alkioiden arvoja. Sen sijaan, jos jokaista taulukon alkiota kasvatetaan yhdellä, niin silloin tehtävä voidaan toteuttaa riippumattomasti.

Suorituskyvyille raskaat silmukat ovat hyviä kohteita datan rinnakkaisuudelle (Damon 2011). Tämä johtuu siitä, että yleensä silmukat toistavat saman tehtävän useasti, mutta eri datajoukolle. Raskaat tehtävät taas ovat sikäli parempia kohteita rinnakkaistukselle, koska silloin Amdahlin lain mukaan saatava nopeushyöty on suurin. Raskaita tehtäviä rinnakkaistettaessa saadaan myös helpommin hyötyjä, kun otetaan huomioon rinnakkaistuksen toteutuksesta tuleva ylimääräinen kuorma suorituskyvylle.

Datan rinnakkaisuus on suhteellisen helppoa tehdä, koska data on lähtökohtaisesti riippumatonta toisistaan, jolloin ei tule synkronoinnnin kanssa haasteita. Tämän ansiosta rinnakkaistus voidaan toteuttaa myös työkalujen avulla, ja esimerkiksi Damon (2011) esittelee joitakin suhteellisen helppoja tapoja toteuttaa rinnakkaistus työkaluilla. Yhdessä esitellyssä tavassa riittää yksi pieni lisä komento kääntäjälle, jotta kääntäjä tekee itse rinnakkaistuksen sopivaksi näkemilleen silmukoille. Toisessa esitellyssä tavassa taas ohjelmoija merkitsee kääntäjälle, mitkä kohdat halutaan rinnakkaistaa. Kuitenkin tulee huomoida, että kyseinen julkaisu on

kohtuullisen vanha ja esitetty tapa toimii vain tietyssä ympäristössä. Kuitenkin se osoittaa, että sopivia työkaluja on olemassa datan rinnakkaistamiselle. Lisäksi, koska työkalut voivat suhteellisen helposti tehdä rinnakkaistuksen, niin luultavasti myös ohjelmoija itse voi toteuttaa datan rinnakkaistamisen melko helposti tarvittaessa.

4.5.2 Tehtävien rinnakkaistus

Tehtävien rinnakkaistuksella tarkoitetaan, että useita eri tehtäviä tehdään rinnakkain (Damon 2011). Tehtävät voivat käyttää samoja tai eri datajoukkoja toistensa kanssa. Kuitenkin tehtäviä rinnakkaistettaessa olisi hyvä, että tehtävät olisivat toisistaan riippumattomia, jotta rinnakkaisuuden toteuttaminen olisi helpompaa (Damon 2011). Jos eri tehtävät käsittelevät esimerkiksi samaa dataa, niin rinnakkaisuuden toimivuuden takaamiseksi joudutaan synkronoimaan eri tehtävien välillä. Tämä aiheuttaa ylimääräistä haastetta toteutukselle ja kuormaa suorituskyvylle, mikä ei ole toivottavaa.

Synkronoinnin tarve myös tekee automaattisen rinnakkaistamisen vaikeammaksi työkaluille, koska ne eivät pysty niin hyvin käsittelemään riippuvuuksia toisin kuin datan rinnakkaisuudessa, jossa riippuvuuksia ei ole. Joitakin toisistaan riippumattomia tehtäviä voidaan rinnakkaistaa työkalujen avulla kohtuu helposti kuten Damon (2011) esittelee. Kuitenkin säikeiden avulla voidaan itse käsin toteuttaa tehtävien rinnakkaisuus, sillä säikeitä voidaan käsitellä hyvin tarkasti (Damon 2011). Käytännössä jokaista rinnakkaistettavaa tehtävää varten luodaan oma säie, jossa tehtävä toteutetaan (Damon 2011). Säikeet voivat kommunikoida keskenään, mikä mahdollistaa tarkemman synkronoinnin niiden välillä, jolloin voidaan rinnakkaistaa myös toisistaan riippuvia tehtäviä. Kuitenkin säikeiden käyttäminen itse on aikaa vievämpää ja virheille alttiimpaa kuin valmiiden työkalujen käyttäminen (Damon 2011). Säikeitä käyttämällä voidaan kuitenkin hyödyntää rinnakkaisuutta laajemmin ja mahdollisesti tehokkaammin.

Tehtävien rinnakkaisuus on laajemmin käytettävissä kuin datan rinnakkaisuus, joka toimii hyvin lähinnä vain silmukoissa. Tehtäviä rinnakkaistettaessa taas voidaan rinnakkaistaa mitä tahansa toisistaan riippumattomia tehtäviä. Esimerkiksi keskustelusovelluksessa yksi tehtävä voisi olla käyttäjän syötteen lukeminen ja viestin lähettäminen muille keskysteluryhmän

jäsenille, samaan aikaan kun toinen tehtävä lukee muilta käyttäjiltä tulevia viestejä ja tulostaa ne näytölle. Näillä tehtävillä on hyvin vähän riippuvuuksia ja siksi ne ovat helposti rinnakkaistettavissa. Jonkinlaista synkronisaatiota taas voitaisiin tarvita, jos molemmat tehtävät kirjoittavat samaan tekstilaatikkoon, mutta tämä ei ole suurikaan haaste.

5 Rinnakkaisuudesta pääsilmukoissa

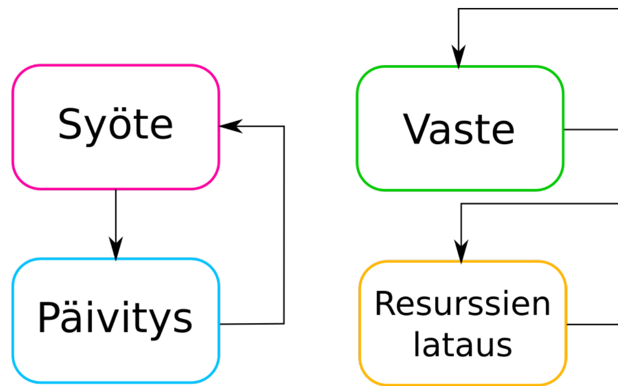
Koska rinnakkaisuudella voidaan saavuttaa huomattavia hyötyjä ohjelman suorituskyvyssä, niin rinnakkaisuuden käyttäminen on suositeltava ratkaisu peleissä, sillä peleillä on hyvin korkeat vaatimukset suorituskyvylle. Parempi suorituskyky voi tarjota paremman pelikokemuksen esimerkiksi suuremman ruudunpäivitysnopeuden muodossa. Kuitenkin pelien monimutkaisuuden takia rinnakkaisuuden toteuttaminen ei ole kovinkaan helppoa tai itsestään selvää. Tässä luvussa esitellään joitakin pääsilmukkamalleja, jotka käyttävät rinnakkaisuutta hyväkseen.

Tähän asti tässä työssä ollaan puhuttu pitkälti vain kolmesta pääsilmukan vaiheesta eli syötteen keruusta, päivityksestä ja vasteen palauttamisesta. Tämä karkea jako on varsin toimiva yksisäikeisissä pääsilmukoissa, koska niissä tehtävien suoritusjärjestyksellä ei ole juurikaan vaikutusta suorituskykyyn. Monisäikeisissä pääsilmukoissa taas halutaan löytää tehtäväkokonaisuuksia, jotka ovat helposti rinnakkaistettavissa, mikä ei karkealla vaiheisiin jaolla onnistu. Tämän takia monisäikeistä pääsilmukkaa suunnitellessa tulee tarkastella pääsilmukassa olevia pienempiä tehtäväkokonaisuuksia, joita ovat esimerkiksi päivityksestä löytyvät fysiikan mallinnus, partikkelisysteemi ja tekoälyn toiminta. Useimmat tehtävät ovat kuitenkin pelikohtaisia ja siksi tässä työssä käsitellään näitä osia vain yleisellä tasolla tai esimerkkeinä.

5.1 Osien puhdas irrottaminen omaan säikeeseen

Eräs tapa hyödyntää säikeistystä pääsilmukoissa on irrottaa jokin tai jotkin tehtäväkokonaisuudet omiin säikeisiinsä (katso kuva 8) (Lake ja Gabb 2005). Tällöin irrotettu tehtäväkokonaisuus voi toimia itsenäisesti pääsilmukasta irrallisena, minkä ansiosta tehtäväkokonaisuus voidaan suorittaa eri tahdissa kuin pääsilmukan tehtävät. Joidenkin tehtävien tiheämpi päivitys voi tarjota paremman pelikokemuksen, kuten esimerkiksi tiheämpi animointi tarjoaa sulavampaa pelikuvaa (Valente, Conci ja Feijó 2005). Tiheämpi päivitys saavutetaan, koska omaan säikeeseen irrotetun tehtävän ei tarvitse odottaa muiden tehtävien valmistumisia, vaan irrotettu tehtävä voidaan suorittaa niin usein kuin, mitä laitteisto mahdollistaa. Toisaalta kaikki tehtävät eivät hyödy suuremmasta päivitystiheydestä. Esimerkiksi syötteen keräämi-

sen päivitystiheyden kasvattaminen ei paranna pelikokemusta, koska syötteitä hyödynnetään vain päivityksessä. Jos syötteet kerätään päivitystä useammin, niin tiheämpi kerääminen jää turhaksi ja siksi syötteen keräämisen ja siihen liittyvien päivityksien tulisi tapahtua samassa tahdissa.



Kuvio 8. Pääsilmukka, jossa osa tehtävistä on eri säikeissä.

Eri tahtista tehtäväkokonaisuuksien suoritusta esiteltiin jo luvussa Interpoloiva malli. Kuitenkin omaan säikeeseen irrottaminen tarjoaa lisäksi mahdollisuuden tehtävien saman aikaiseen suorittamiseen, joten esimerkiksi voidaan piirtää kuvaa samaan aikaan kun lasketaan seuraavaa päivitystä. Rinnakkaistuksen ansiosta tämä lisää pelin tehokkuutta, jolloin jää enemmän resursseja muuhun (McShaffry 2014). Lisäksi saman aikaisuus voi tarjota myös muita käytännön hyötyjä.

Esimerkiksi resurssien lataaminen on oikein hyvä tehdä omassa säikeessään, koska se on yleensä hidasta (Tulip, Bekkema ja Nesbitt 2006). Säikeistyksen ansiosta peli pystyy toimimaan normaalisti samaan aikaan, kun resursseja ladataan toisessa säikeessä jopa useita sekunteja. Esimerkiksi kun lähestytään seuraavaa pelimaailman aluetta, niin alueen resursseja voidaan ladata taustalla ilman, että pelin tarvitsee pysähtyä latauksen ajaksi. Resurssien latauksen voisi toki toteuttaa pääsilmukassa niin, että joka silmukan kierroksella luettaisiin tiedostosta hieman tietoa, mutta se olisi käytännössä vaikeaa toteuttaa ja se mahdollisesti hidastaisi pelin muuta toimintaa. Sen sijaan erillistä säiettä käyttämällä resurssien lataus on huomattavasti helpompaa toteuttaa ja luultavasti suorituskyvyn kannalta tehokkaampaa rinnakkaistuksesta johtuen. Resurssien lataus voidaan myös irrottaa huoletta omaan säikeeseensä, koska sillä ei ole juurikaan riippuvuuksia muihin tehtäväkokonaisuuksiin.

Toinen hyvä vaihtoehto on irrottaa kuvan piirtäminen omaan säikeeseensä, koska suuri osa piirtämiseen kuluvasta ajasta kuluu näytönohjainta odotellessa (McShaffry 2014). Lisäksi riippuvuuksien suhteen kuvan piirtäminen on hyvin irrallinen muista pelin osista. Se tarvitsee vain kopion pelimaailman sen hetkisestä tilasta, kun taas muut pelin osat eivät yleensä hyödynnä ollenkaan piirtämisestä saatuja tuloksia (Tulip, Bekkema ja Nesbitt 2006). Tämän yksisuuntaisen riippuvuuden ansiosta pelin tilan päivittämisen jälkeen voidaan huoletta aloitta kuvan piirtäminen erillisessä säikeessä ja samaan aikaan jo laskea seuraavaa pelimaailman tilaa. Kun kuvan piirtäminen on saatu tehtyä, niin voidaan odottaa seuraavaa päivitettyä pelimaailman versiota tai vaihtoehtoisesti voidaan käyttää viimeisintä versiota pelimaailmasta. Myös luvussa Interpoloiva malli esitetty ratkaisu eli interpolointi eri versioiden väliltä voisi oikein hyvin toimia.

Rinnakkaistettaville tehtäville onkin yleisesti hyvin toivottavaa, että ne olisivat riippumattomia toisistaan (El Rhalibi, Merabti ja Shen 2006). Jos tehtävät ovat riippuvaisia toisistaan, niin tehtäviä ei välttämättä pystytä suorittamaan samanaikaisesti. Riippuvuudet tehtävien välillä näkyy usein siinä, että käytetään samoja resursseja, mikä voi aiheuttaa epädeterministisyyttä (katso luku Pelin deterministisyys) ja ongelmia pelin logiikassa. Tästä on esimerkkinä kuva 7, jossa näkyy kuinka pankkien tilisiirto voi mennä pieleen rinnakkaistettaessa.

Yksi ratkaisu on käyttää synkronointia säikeiden välillä, jolla voidaan jonkin verran korjata ja hallita tilannetta vaikka joitakin riippuvuuksia olisikin (McShaffry 2014). Kuitenkin saatava tehokkuus voi kärsiä runsaan synkronoinnin käytön takia, jolloin saatava suorituskyky saattaa jäädä vähäiseksi (Lake ja Gabb 2005). Esimerkiksi lukkoja käytettäessä rinnakkaistuksesta saatavat hyödyt heikkenevät hieman, koska tehtävät joutuvat odottamaan lukkojen avautumisia. Synkronoinnin tarpeen välttämiseksi voidaan myös irrottaa useampia tehtäväkokonaisuuksia kerralla omaan säikeeseen (Lake ja Gabb 2005). Tällöin toisistaan vahvasti riippuvaiset tehtäväkokonaisuudet voidaan laittaa samaan säikeeseen, jolloin synkronointia ei tarvitse tehdä niiden välillä (Lake ja Gabb 2005). Tehtävien välisiä riippuvuuuksia käsitellään tarkemmin luvussa Riippuvuudet rinnakkaistettaessa.

Toinen ratkaisu on asettaa rinnakkaistetut tehtävät käyttämään aina viimeistä valmista päivitettyä versiota datasta sen sijaan, että käytettäisiin keskeneräistä versiota (Lake ja Gabb 2005). Tällöin ei tule tilannetta, jossa käytettävä data muuttuisi kesken tehtävän suorituksen,

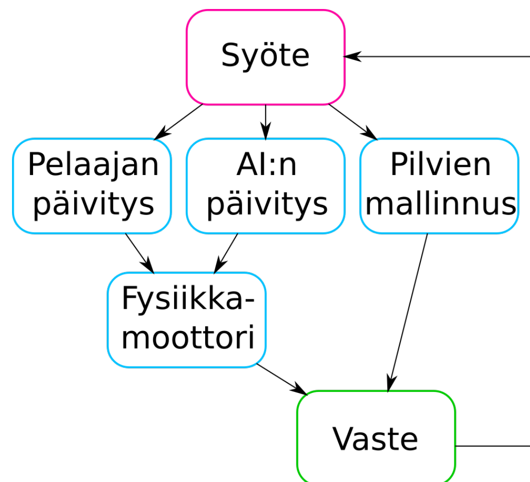
mikä takaa pelin oikean toiminnan. Kuitenkin tämä ratkaisu ei takaa deterministisyyttä, koska emme voi tietää, mitä versiota tehtävät käyttävät. Ratkaisu kuitenkin tarjoaa kohtuullisen helpon tavan rinnakkaistaa kaikki tehtäväkokonaisuudet omiin säikeisiin ilman, että pelin logiikka häiriintyy pahasti.

Resurssien latauksen ja kuvan piirtämisen lisäksi voidaan irrottaa myös muita tehtäväkokonaisuuksia omiin säikeisiin, mutta edellä kuvatut ovat yleisesti toimivimpia ja siksi tässä mainittu. Yleisesti ottaen omaan säikeeseen irrottaminen tarjoaa rinnakkaistuksen ansiosta paremman suorituskyvyn, mutta siitä hyötyvät myös tehtäväkokonaisuudet, jotka tarvitsevat mahdollisimman suuren päivitystiheyden. Ratkaisun etuna on myös se, että ratkaisussa ei alun jälkeen luoda tai tuhota säikeitä, jolloin vältytään säikeiden luomisen aiheuttamilta kustannuksilta suorituskykyyn. Kuitenkin ratkaisun heikkoutena on sen huono skaalautuvuus (Mönkkönen 2006). Ratkaisussa säikeitä voi olla korkeintaan käytössä saman verran kuin tehtäväkokonaisuuksia, mikä on puute, jos laitteisto tarjoaa tätä enemmän suorittimen ytimiä (Mönkkönen 2006; Best ym. 2009). Kuitenkin ratkaisu on hyvä lähtökohta ja sitä voidaan käyttää yhdessä muiden ratkaisujen kanssa.

5.2 Suoritus haarautuu useisiin säikeisiin

Toinen tapa rinnakkaistaa pääsilmukassa tehtäviä on haarauttaa tehtävien suoritus useisiin eri rinnakkaisesti suoritettaviin haaroihin (katso kuva 9). Kun kaikkien haarojen sisältämät tehtävät on suoritettu, niin haarat yhdistetään takaisin yhteen. Käytännössä tämä tarkoittaa, että haarautettaessa pääsäikeessä luodaan jokaista uutta haaraa kohti uusi säie, jossa tehtävät suoritetaan. Yhdistäessä taas ylimääräiset säikeet tuhotaan ja jatketaan suoritusta pääsäikeessä. Haarat voivat lisäksi haarautua haaran sisällä uudestaan.

Eri haarojen tehtävien tulee olla toisistaan riippumattomia, jotta eri haarojen välillä ei tarvitsi kommunikoida keskenään (Mönkkönen 2006). Tämä mahdollistaa helpomman rinnakkaisuuden toteutuksen, koska synkronisaatiota ei tarvitse toteuttaa, vaan voidaan käyttää tehtävien toteutuksia lähes sellaisinaan (Mönkkönen 2006). Tämä myös takaa paremmin pääsilmukalle deterministisyyden, koska rinnakkaistetaan vain toisistaan riippumattomia tehtäviä, joiden suoritusjärjestyksellä ei ole väliä.



Kuvio 9. Pääsilmukka, jossa pelattavan hahmon päivittäminen, tekoälyvihollisten päivittäminen ja pilvien mallinnus on haarautettu eri säikeisiin.

Kuitenkin jotta voisimme tunnistaa, mitkä tehtävät ovat toisistaan riippumattomia ja siten rinnakkaistettavissa, niin tulee ensin tunnistaa yksittäiset pääsilmukassa suoritettavat tehtävät (Andrews 2009b). Tämä on yleisesti ottaen helppoa, koska yleensä jo peliä suunniteltaessa, peli kootaan näistä tehtävistä (Andrews 2009b). Kuitenkin haasteena on, jos toteutuksessa on yhdistetty joitakin tehtäviä toisiinsa. Esimerkiksi jos tekoälyn päivitys huolehtii myös tekoälyä käyttävän vihollisen liikuttamisesta, on kyseisen tehtävän rinnakkaistaminen vaikeampaa, koska sillä on sekä tekoälyn, että fyysikan mallinnuksen riippuvuuksia. Rinnakkaistamisen kannalta olisikin parempi, jos toteutuksessa tehtävät olisivat eroteltu mahdollimman selkeästi toisistaan irrallisiksi.

Myös jotta tiedettäisiin, missä kohdissa tehtävien suoritus voidaan haarauttaa, tulee ensin selvittää tehtävien väliset riippuvuudet (Andrews 2009b). Tässä erittäin hyvänä työkaluna toimii riippuvuuskaavio (eng. dependency graph), josta näkee suoraan, missä järjestyksessä tehtävät tulee suorittaa, ja mitkä voidaan suorittaa rinnakkain. Aiheesta on artikkeli El Rhalibi, Merabti ja Shen (2006), jossa on tunnistettu kirjoittajien omasta pelistä tehtävien väliset riippuvuudet ja luotu sen pohjalta riippuvuuskaavio, jota käytettiin rinnakkaistettaessa. Kyseinen artikkeli kannattaa lukea, jos riippuvuuskaavion luomisesta haluaa oikean esimerkin. Itse käsittelen riippuvuuksia luvussa Riippuvuudet rinnakkaistettaessa, mutta en niin tarkasti enkä myöskään konkreettisella esimerkillä.

Tehtävien väliset riippuvuudet eivät kuitenkaan aina estä rinnakkaistusta kokonaan. Esimerkiksi jos tehtävä tuottaa listaan dataa, niin toinen tehtävä voi alkaa jo käsittelemään jo luotua dataa listasta, vaikka edellinen tehtävä ei ole kokonaan valmis (Best ym. 2009). Esimerkiksi törmäysten käsittely voi toimia kaksi vaiheisena niin, että ensimmäisessä vaiheessa etsitään, mitkä kohteet törmäävät ja toisessa vaiheessa käsitellään törmäys itsessään (Best ym. 2009). Tällöin nämä toimivat tuottaja/kuluttaja periaatteella (eng. producer/consumer), jossa ensimmäinen tuottaa dataa, jota jälkimmäinen käyttää (Best ym. 2009).

Toisena esimerkkinä kuinka rinnakkaistus voidaan toteuttaa riippuvuudesta huolimatta on, jos kaksi tehtävää käyttävät samaa dataa, joista vain toinen muokkaa sitä. Tällöin rinnakkaistus onnistuu, jos toiselle annetaan vain kopio kyseisestä datasta ennen kuin toinen ehtii muokkaamaan sitä. Vaihtoehtoisesti molemmat voivat lukea samaa dataa samaanaikaan, mutta dataan tehdyt muutokset asetetaan voimaan vasta, kun molemmat tehtävät on suoritettu.

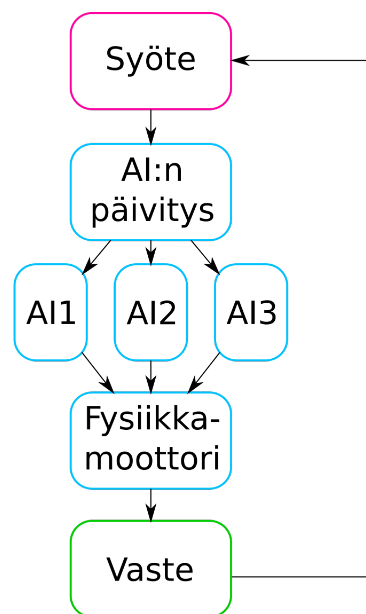
Pääsilmukan haarauttaminen mahdollistaa deterministisyyden säilyttävän rinnakkaisuuden, joka voidaan myös toteuttaa ilman suuria muutoksia peliin. Kuitenkin ratkaisu sisältää joitakin haasteita. Ensinnäkin riippuvuuksien määrittäminen pelistä on suhteellisen työlästä ja virhealtista tehdä jälkeenpäin. Lisäksi jatkuva säikeiden luonti ja tuhoaminen heikentää saatavaa hyötyä suorituskyvylle. Myös rinnakkaistettavien tehtävien määrä on hyvin rajallinen, jolloin tehtävien määrää useammat suorittimen ytimet jäävät turhiksi eli malli skaalautuu huonosti (Mönkkönen 2006). Nämä ongelmat ovat kuitenkin liittyvät lähinnä vain ratkaisun rajalliseen hyötyyn tai käyttöön ottamisen haasteisiin, mutta pääsilmukan toiminnalle itselleen se ei aseta lähes ollenkaan rajoitteita, joten ratkaisu toimii hyvin yhdessä muiden rinnakkaistuksen ratkaisujen kanssa.

5.3 Tehtävän sisäinen rinnakkaistus

Vaikka tehtävien mukaan rinnakkaistaminen on oikein toimiva ratkaisu, niin se ei ole kuitenkaan riittävä. Tehtävät voivat esimerkiksi olla liian isoja, jotta säikeistettävät osat olisivat tarpeeksi hienojakoisia. Esimerkiksi jos yksi tehtävä (kuten tekoäly) veisi 90 % pelin suoritusajasta, niin Amdahlin lain (katso luku Amdahlin laki) avulla voidaan päätellä, että jäljelle

jäävän 10 % säikeistäminen ei voi tuoda merkittäviä nopeushyötyjä. Tämän takia tulisi tehtävien rinnakkaistuksen sijaan pyrkiä rinnakkaistamaan isot aikaa vievät tehtävät niiden itsensä sisällä. Tässä voimme hyödyntää datan rinnakkaistusta (katso luku Datan rinnakkaistus).

Datan rinnakkaistusta voidaan käyttää tilanteissa, joissa sama tehtävä tehdään useille eri kohteille (Damon 2011). Esimerkiksi jos pelissä on useita tekoäly vihollisia, niin jokaiselle niistä suoritetaan omat tekoälyyn liittyvät toiminnot (katso kuva 10). Nämä toiminnot eivät ole toisistaan riippuvia ja siksi voidaan suorittaa jokaisen tekoälyvastustajan toiminnot eri säikeissä. Käytännössä kuitenkin kannattaa jakaa yksittäisten kohteiden sijaan ryhmittäin, koska muuten säikeitä voi olla paljon enemmän kuin, mitä ytimiä on käytössä (Best ym. 2009). Tällöin tulee turhaa säikeiden luontia ja tuhoamista, mikä heikentää saatavaa hyötyä suorituskyvylle. Esimerkiksi jos tekoälyvihollisia on 80 ja ytimiä on käytössä kahdeksan, niin eräs tasainen jako voisi olla niin, että jokainen kahdeksasta säikeestä vastaa kymmenestä tekoälyvihollisesta kerralla.



Kuvio 10. Pääsilmukka, jossa tekoälyn päivitys on rinnakkaistettu.

Datan rinnakkaistuksella voimme rinnakkaistaa yksittäisen tehtävän sisällä, mikä tarjoaa mahdollisuuden hienojakoisempaan rinnakkaistukseen ja siten tasaisempaan kuormitukseen suorittimien välillä. Tehtävien mukaisessa rinnakkaistuksessa myös erillisiä säikeitä voi olla korkeintaan vain saman verran kuin, mitä tehtäviä on (Mönkkönen 2006; Best ym. 2009).

Tämä nousee rajoitteeksi, jos on tätä enemmän suorittimen ytimiä käytössä, jolloin ylimääräiset ytimet jäävät käyttämättä (Mönkkönen 2006; Best ym. 2009). Datan rinnakkaistuksella taas voidaan hyödyntää huomattavasti useampaa suorittimen ydintä, ja lisäksi voidaan jakaa paljon tarkemmin, minkä kokoisia datajoukkoja säikeet suorittavat, ja siten suoraan hallita käytettävien säikeiden määrää (Mönkkönen 2006). Tämä tarjoaa paljon paremman skaalautuvuuden kuin, mitä voisimme saada pelkästään tehtävien rinnakkaistamisella (Mönkkönen 2006).

Lisäksi koska datan rinnakkaistaminen tapahtuu yleensä tehtävän sisällä, niin sillä ei ole juurikaan vaikutuksia itse pääsilmukan rakenteeseen, vaan voidaan kohdella sitä tehtävän sisäisenä toteutuksena. Tämän ansiosta dataa rinnakkaistaminen on suhteellisen helppoa ja mutkatonta toteuttaa. Lisäksi kuten luvussa Datan rinnakkaistus todettiin, niin datan rinnakkaistus voidaan mahdollisesti toteuttaa myös työkalujen avulla, mikä vähentää kehittäjän taakkaa.

Joissakin tilanteissa voi kuitenkin olla hyödyllistä toteuttaa datan rinnakkaistaminen itse manuaalisesti (Best ym. 2009). Esimerkiksi, jos haluamme rinnakkaistaa sellaisen tehtävän, jossa eri rinnakkaistettavat kohteet saattaisivat vaikuttaa toisiinsa, niin silloin oma manuaalinen toteutus käy tarpeelliseksi (Best ym. 2009). Esimerkiksi fysiikkaamoottorissa kappaleen liikuttaminen on toisista kappaleista riippuvaa, koska kappaleiden välillä voi tapahtua törmäyksiä. Tätä varten kappaleen liikettä päivitettäessä joudutaan myös katsomaan tietoa muista kappaleista. Tällöin kuitenkin rinnakkaistetussa toteutuksessa voi tulla ongelmia, jos toinen säie on muokkaamassa kyseistä kappaletta. Tästä voi seurata virheellistä datan käyttöä, jossa luettu data ehtii muuttua toisen säikeen toimesta ennen kuin dataa on käytetty. Tämä voidaan estää säikeiden välisellä synkronoinnilla, mikä kuitenkin aiheuttaa ylimääräistä rasitusta suorituskyvylle, kun esimerkiksi odotetaan lukkojen avautumisia (Silverman 2008).

Synkroinnin aiheuttamaa rasitusta kuitenkin voidaan vähentää joissakin tapauksissa heuristisilla menetelmillä. Esimerkiksi voidaan heuristisesti sanoa, että kilometrin päässä toisistaan olevat kappaleet tuskin törmäävät lähiaikoina. Samalla tavalla eri huoneissa olevien kappaleiden välinen törmäys on epätodennäköistä ja jos törmäys tapahtuu, niin se tapahtuu huoneiden reunoilla. Silverman (2008) esittääkin, että alueellista jakoa voidaan käyttää hyödyksi rinnakkaistettaessa niin, että yksi säie olisi vastuussa kullakin alueella olevien kohteiden

päivittämisestä. Samaa aluetta päivitettäessä tulee vähemmän tilanteita, joissa luettaisiin sellaista dataa, jota toinen säie käsittelisi, mikä vähentää synkroinnista aiheutuvaa kuormaa. Synkronoinnissa Silverman (2008) käyttää transaktiomuistia, sen yksinkertaisuuden ja toimintavarmuuden takia. Alueellista jakoa rinnakkaistuksessa hyödyntää myös (Abdelkhalek ja Bilas 2004).

Alueellista jakoa voidaan siis hyödyntää dataa rinnakkaistettaessa vaikka rinnakkaistettavien kohteiden välillä olisikin riippuvuuksia, mikä tarjoaa enemmän mahdollisuuksia rinnakkaistettavuudelle. Samalla tavalla myös muita menetelmiä voidaan mahdollisesti hyödyntää vastaavasti. Alueellinen jako on kuitenkin yleisesti toimiva ja käytettävissä oleva ratkaisu peleissä, koska monissa peleissä yleensä on vähintäänkin kaksiulotteinen maailma. Alueellista jakamista käytetään peleissä myös muuhunkin kuin pelkkään rinnakkaistamiseen, joten sen hyödyntäminen myös rinnakkaistamisessa on luontevaa. Joka tapauksessa on siis mahdollista rinnakkaistaa myös sellaisia datajoukkoja, jotka voivat olla toisistaan riippuvia. Tämä tosin vaatii ylimääräistä suunnittelua verrattuna toisistaan riippumattomien datajoukkojen rinnakkaistamiseen.

5.4 Säieallas

Luvuissa Tehtävän sisäinen rinnakkaistus ja Suoritus haarautuu useisiin säikeisiin esittellyissä rinnakkaistuksen tavoissa molemmissa on ongelmana, että niissä luodaan ja tuhotaan jatkuvasti säikeitä. Tähän ratkaisuna on käyttää säieallasta, jolla vältytään uusien säikeiden luomiselta (Tulip, Bekkema ja Nesbitt 2006). Uusien säikeiden luomisen sijaan, ratkaisussa käytetään uudelleen jo luotuja säikeitä, joille vain annetaan uusi tehtävä aina edellisen tehtävän suorittamisen jälkeen (Tulip, Bekkema ja Nesbitt 2006).

Käytännössä säieallas voidaan toteuttaa käyttämällä erillistä komponenttia, joka huolehtii tehtävien hallinnasta (eng. task manager) (Andrews 2009a). Tehtävien hallinnan vastuulla on jakaa säikeille aina seuraava suoritettava tehtävä tai kertoa, että seuraavaa tehtävää ei voida vielä suorittaa riippuvuuksista johtuen. Tehtävien hallinnan vastuulla on siis varmistaa, että tehtävät suoritetaan oikeassa järjestyksessä, jotta peli toimisi oikein (Tulip, Bekkema ja Nesbitt 2006). Tehtävien varsinainen suoritus taas on pelin alussa luotujen säikeiden vas-

tuulla, jotka aina edellisen tehtävän suoritettuaan ilmoittavat tehtävien hallinnalle tehtävän suorituksesta ja pyytävät uutta tehtävää. Tämän ansiosta säikeitä ei tarvitse luoda muuta kuin pelin alussa suorittimen ytimiä vastaava määrä (Andrews 2009a).

Tehtävien hallinnalle alussa annetaan suoritettavien tehtävien lisäksi tieto, missä järjestyksessä tehtävät voidaan suorittaa ja mitkä tehtävät voidaan suorittaa rinnakkaisesti. Tämä ei poikkea juurikaan luvun Suoritus haarautuu useisiin säikeisiin ratkaisusta, mutta säieallasta käytettäessä joudutaan keskitetysti määrittelemään tehtävien järjestys. Sinänsä on toki mahdollista, että tehtävä aina lopuksi lisäisi seuraavan tehtävän tehtävän hallinnalle, mutta haasteeksi tulee tällöin tilanteet, joissa eri haarat yhdistetään. Tällöin on hankala sanoa kumman haaran tulee lisätä seuraava tehtävä ja miten se tiedottaa, että toinenkin haara tulee olla suoritettu ennen kuin tehtävä aloitetaan. Myös keskitetty tehtävien listaus mahdollistaa helpomman suoritusjärjestyksen tarkastelun ja hallinnan.

Kuitenkin dataa rinnakkaistettaessa (käsitelty luvussa Tehtävän sisäinen rinnakkaistus) on oikein hyvä antaa tehtävälle oikeus lisätä itse luodut tehtävät tehtävien hallinnalle (Andrews 2009a). Käytännössä esimerkiksi tekoälyn päivittäminen olisi oma tehtävänsä, mutta se voisi luoda uusia alitehtäviä jokaista yksittäisen tekoälyn päivittämistä kohti. Tällöin tehtävät voivat hoitaa datan rinnakkaistuksen vapaasti ilman suuriakaan muutoksia muuhun toimintaan.

Säikeiden luomisen ja tuhoamisen vähentämisen lisäksi säieallas tarjoaa paremman mahdollisuuden tasata kuormaa eri suorittimen ytimien välillä, koska tehtävät jaetaan suorituksen aikana vapaille säikeille (Tulip, Bekkema ja Nesbitt 2006). Tällöin ei tule tilannetta, jossa yhdellä säikeellä olisi vielä useita tehtäviä suoritettavana, kun muut ovat jo valmiita. Lisäksi koska säikeitä ei tarvitse turhaan luoda, niin dataa rinnakkaistettaessa voidaan ryhmitellä tehtävät pienemmiksi osiksi. Tällöin tehtävät ovat hienojakoisempia (katso luku Hienojakoisuus) ja siten muiden säikeiden odotusaika vähenee.

Säieallas siis tarjoaa hyvän ratkaisun, jolla täydentää lukujen Suoritus haarautuu useisiin säikeisiin ja Tehtävän sisäinen rinnakkaistus rinnakkaistuksen toteutuksia. Säiealtaan toteuttaminen vaatii kuitenkin jonkin verran lisää työtä ja voi olla haastavaa. Kuitenkin säiealtaita on laajasti tutkittu ja niistä luulisi löytyvän jopa valmiita toteutuksia, joita voi käyttää apuna omassa pelissä.

5.5 Grafiikkasuorittimen käyttäminen rinnakkaistaessa

Oman mielenkiintoisen lisänsä pääsilmukan rinnakkaistamiseen tuo grafiikkasuorittimen (eng. GPU eli Graphical Processing Unit) käyttäminen. Tavallisesti grafiikkasuorittimia käytetään vain grafiikan laskemisessa, mutta niitä voidaan käyttää myös yleisissä tehtävissä (liittyy englannin kieliseen käsitteeseen GPGPU eli General-Purpose Computation on Graphical Processing Units). Koska grafiikkasuorittimet ovat tavallisia suorittimia tehokkaampia rinnakkaistetuissa tehtävissä, niin grafiikkasuorittimien käyttäminen on kannattavaa rinnakkaistettujen tehtävien ratkaisuissa (Joselli ym. 2008). Grafiikkasuorittimien käyttäminen ei ole kuitenkaan itselleni kovinkaan tuttua, joten aiheen hyödyllisyydestä huolimatta käsittelen aihetta vain pinnallisesti.

Grafiikkasuorittimia voidaan käyttää esimerkiksi fysiikan mallinnuksessa, koska useimmiten fysiikanmallinnus on hyvin rinnakkaistuva ongelma (Joselli ja Clua 2009). Myös erilaisten isojen joukkojen mallinnuksessa grafiikkasuoritin on toimiva ratkaisu, kuten esimerkiksi mallinnettaessa kadulla käveleviä ihmisiä tai eläinlaumaa (Joselli ja Clua 2009). Lisäksi grafiikkasuoritinta voidaan käytää tekoälyn toiminnassa laajemminkin ja jopa itse pelin logiikassa (Joselli ja Clua 2009).

Laajemman rinnakkaistamisen ansiosta grafiikkasuoritin vaikuttaisikin toimivan paremmin tehtävissä, jotka skaalautuvat paremmin useammaksi pieneksi tehtäväksi. Tämän ja esimerkkien perusteella luvun Tehtävän sisäinen rinnakkaistus ratkaisu saattaisi soveltua paremmin grafiikkasuorittimelle sen sijaan, että suoritettaisiin useita eri tehtäviä samanaikaisesti grafiikkasuorittimella, sillä ne rinnakkaistuvat vain hyvin rajallisesti. Sen sijaan esimerkiksi joukkoja mallinnettaessa, joukossa voi olla tuhansia samankaltaisia yksilöitä, jolloin tehtävä on hyvin rinnakkaistuva.

Grafiikkasuorittimen käyttöä voidaan optimoida käyttämällä automaattista tehtävien jakoa grafiikkasuorittimen ja tavallisen suorittimen välillä. Automaattisella tehtävien jaolla voidaan hyödyntää tehokkaasti molempia suorittimia antamalla tehtävä aina suoritettavaksi sille suorittimelle, joka sillä hetkellä pystyy tehtävän parhaiten suorittamaan (Joselli ym. 2010). Kehittäjä ei aina pysty tehokkaasti sanomaan, mitkä tehtävät kannattaa suorittaa milläkin suorittimella, koska tilanne voi muuttua, jolloin toisen suorittimen käyttäminen onkin parem-

pi ratkaisu. Esimerkiksi jos laitteisto muuttuu, niin silloin tehtävän tehokkuus eri suorittimella voi muuttua aiemmasta oletuksesta. Toisaalta suorittimet voivat suorittaa eri tehtäviä samanaikaisesti, jolloin voidaan myös jakaa tehtäviä vapaalle suorittimelle (Joselli ym. 2010).

Automaattinen tehtävienjako kuitenkin vaatii, että tehtävälle on toteutukset olemassa molemmille suorittimille, jolloin tarvittaessa tehtävä voidaan toteuttaa kummalla tahansa suorittimella (Joselli ym. 2010). Kaikkille tehtäville ei välttämättä tarvitse tai edes kannata tehdä molempia toteutuksia, koska jotkin tehtävät ovat huomattavasti tehokkaampia tai vain vaikeampia toteuttaa toisella suorittimella. Esimerkiksi grafiikan piirtäminen on järkevää jättää suosiolla grafiikkasuorittimen vastuulle.

Grafiikkasuorittimien käyttäminen muussakin kuin grafiikassa on hyvä tapa tehostaa pelin toimintaa. Kuitenkin oman käsityksen mukaan se voi olla kohtuullisen haastavaa ja vaatii aiheeseen perehtymistä, joskin työkalujen käyttäminen varmaan helpottaa käyttöä. Saatavat hyödyt eivät välttämättä ole kuitenkaan tarpeellisia pienemmissä peleissä. Lisäksi on myös mahdollista käyttää valmiita pelimoottoreita tai kirjastoja, jotka jo valmiiksi hyödyntävät grafiikkasuoritinta, jolloin itse ei tarvitse tarkemmin miettiä niiden toteutuksia.

6 Valmiiden toteutuksien tarkastelu

Tässä osiossa pyrin tarkastelemaan valmiita toteutuksia pääsilmukoista eli miten pääsilmukkoja on toteutettu valitsemmissani peleissä ja pelimoottoreissa. Tarkoituksena on tutkia, minkälaisia pääsilmukoita oikeasti käytetään, ja kuinka esittelemäni teoria näkyy niissä. Lisäksi haen samalla käytännön esimerkkiä omaa pääsilmukan toteutustani varten. Erityisesti tarkastelen minkälaista rakennetta tarkasteltava pääsilmukka käyttää, ja kuinka usein päivitykset tapahtuvat.

Tutkimuksen merkittävimpänä haasteena on kuitenkin tutkittavan tiedon vaikea saatavuus. Esimerkiksi kaupallisten pelien lähdekoodeja tai dokumentaatioita julkaistaan hyvin vähän, minkä takia pelin toimintaan on hankalaa perehtyä. Tämä rajoittaa huomattavasti, mitä pelejä voidaan valita tutkittaviksi. Tämä ei kuitenkaan estä täysin tutkimuksen tekoa, koska pelien sijaan tutkimuksessa voidaan käyttää myös pelkkää pelimoottoria. Esimerkiksi Unity ja Unreal Engine ovat kaupallisesti laajassa käytössä olevia pelimoottoreita, joiden lähdekoodia on mahdollista lukea ja niistä on dokumentaatiota saatavilla. Mainittuja moottoreita myydään juurikin pelimoottoreina pelien kehittäjille ja siksi niiden toimintatapaa halutaan esitellä avoimesti.

Valittavien pelien tai pelimoottorien tulee myös olla kohtuullisen merkittäviä ja laajalti tunnettuja, jotta saadaan käsitystä, minkälaiset pääsilmukat voivat oikeasti menestyä. Lisäksi on toivottavaa, että ainakin osa valittavista pelimoottoreista tai peleistä olisi uudehkoja, jotta saataisiin selville käytettävien pääsilmukoiden nykyinen tilanne. Lisäksi rinnakkaistus on ollut yleisesti pelilaitteiston tukemaa vain rajallisen aikaa, ja siksi vanhemmissa peleissä ei ole luultavasti ollut tarvetta käyttää rinnakkaistettuja pääsilmukoita. Tuoreemmissa peleissä on luultavammin suurempi mahdollisuus, että rinnakkaistus on otettu pääsilmukassa käyttettäväksi. Kuitenkin rinnakkaistetut pääsilmukat on yksi tämän työn merkittävimmistä aiheista ja siksi on toivottavaa, että ainakin jossakin valituista peleistä hyödynnettäisiin pääsilmukan rinnakkaistusta.

6.1 Phaser

Phaser on Photon Stormin kehittämä TypeScriptillä toteutettu pelimootori, jonka ensimmäinen versio julkaistiin vuonna 2013 (Davey 2013). Phaser on kirjoitushetkellä yhä aktiivisessa kehityksessä ja kirjoitushetken uusin versio 3.16.2 julkaistiin 11.2.2019 (Davey 2019). Phaserillä on mahdollista toteuttaa kaksiulotteisia pelejä JavaScriptillä, jotka toimivat selaimessa HTML5:n avulla (Davey 2013). Phaser käyttää MIT-lisenssiä ja sitä kehitetään avoimesti GitHubissa (Photon Storm 2019). Sen ansiosta Phaserin lähdekoodiin on mahdollista päästä käsiksi kuten myös moottorin dokumentaatioon, mikä mahdollistaa sen toiminnan tutkimisen. Phaser on myös ainakin oman käsitykseni mukaan kohtuullisen laajasti tunnettu ja käytössä oleva pelimoottori, joten sen tutkiminen on mielekästä, kun halutaan yleistä käsitystä käytettävien pelimoottorien tilanteesta. Lisäksi Phaserin valintaan tutkittavaksi vaikutti huomattavasti myös se, että minulla itselläni on henkilökohtaista kokemusta Phaserin käyttämisestä pelien tekemiseen.

Phaserin käyttämän pääsilmukan tutkimisessa käytän ensisijaisesti sen lähdekoodia ja siihen liittyviä dokumentteja. Keskityn myös tarkastelemaan vain Phaserin kirjoitushetken uusinta versiota eli versiota 3.16.1. Lähdekoodi on haettu osoitteesta `http://phaser.io/download/release/3.16.1` ja sieltä ladattu Phaser zip-pakettina. Tutkimisen tarkoitusena on selvittää, minkälaista pääsilmukkaa Phaser käyttää, ja kuinka yhtenevä se on esittelemäni teorian kanssa.

6.1.1 Phaserin pääsilmukan rakenne

Phaserin lähdekoodia tutkimalla voidaan havaita, että Phaser ei toimi perinteisen ohjelmoinnista tutun silmukan (kuten esim. while-silmukan) avulla. Tämä näkyy koodilistauksesta 9.2, jossa kommenteissa näkyy, että kyseistä pelin päivittämistä kutsuvaa funktiota kutsutaan aina, kun selain päivittää tilaansa, tapahtuu Request Animation Frame tai ajastimen kutsusta. Tämän myötä ei ole siis tarpeellista toteuttaa Phaserille omaa silmukkaa, koska päivitystä voidaan kutsua muun muassa selaimen avulla ajastinta käyttämällä. Siten varsinaisen silmukan toiminta on jätetty selaimen vastuulle, joka huolehtii päivityskutsuista.

Kun tämä TimeStep-luokan step-funktiota on kutsuttu, niin se suorittaa omia toimenpitei-

tään ja laskee muun muassa päivityksessä käytettävän ajan (tästä tarkemmin luvussa Phaserin pääsilmukan päivitystiheys). Tämän jälkeen se kutsuu lopussa tiedostosta Game.js löytyvää step-funktiota (löytyy listauksesta 9.3), jonka vastuulla on päivityksen suorittaminen tarkemmin, antaen sille parametreinä päivitysaikaan liittyviä tietoja.

Game.js tiedoston step-funktio ei itsessään vielä suorita päivityksiä, mutta se lähettää tapahtumailmoituksia (eng. event), jotka kertovat päivityksen edistymisestä. Nämä tapahtumailmoitukset voivat aktivoida muita järjestelmiä kuten lisäosien toimintaa, ja ilmoitusten lähettäminen on siten hyvin olennaista pelimoottorin toiminnalle. Ilmoitusten välissä Game.js kutsuu tarkemmin vielä SceneManager-luokan update-funktiota (listauksessa hieman harhauttavasti SceneManager luokan olio on nimellä "scene"), joka käytännössä kutsuu jokaisen skenen omaa päivitysfunktiota välivaiheiden kautta.

Skenen oma päivitysfunktio on käytännössä pelin tekijän itsensä luoma päivitysfunktio (tarkkaan ottaen moottorilla on oma tyhjä update-funktio skenelle, mutta pelin tekijän oma update-funktio korvaa sen). Esimerkkipohja Phaserillä tehtävän pelin koodista on tutkittavissa listauksessa 9.1. Tämän listauksen update-funktio siis suoritetaan, kun Phaser pelimoottori toteuttaa päivityksen, joten kehittäjän ei tarvitse tietää pelimoottorin toiminnasta paljoakaan vaan riittää pelkästään oman päivityksen kirjoittaminen.

Kun päivitykset on saatu tehtyä, niin lopulta palataan takaisin Game.js tiedoston step-funktion suoritukseen, jossa jatketaan toimintaa piirtämällä näytölle kuvaa pelimaailman tilasta. Käytännössä tämä tapahtuu vastaavasti kuin päivittäminen eli kutsutaan SceneManager-luokkaa, joka huolehtii tarkemmin jokaisen skenen piirtämisestä. Kuvan piirtämiseen en juurikaan perehtynyt, mutta en usko, että siinä tapahtuu mitään erityisen merkittävää.

Syötteitä kuten esimerkiksi näppäinten painalluksia voidaan hyödyntää Phaserissä ainakin kahdella eri tavalla. Ensimmäinen tapa on hyödyntää syötteitä omassa update-funktiossa, jossa voidaan vain kysyä näppäimen tilaa ja tehdä asioita sen perusteella, jolloin syötteiden käsittely ja päivitys limittyvät toisiinsa. Toinen vaihotehto on rekisteröityä kuuntelemaan näppäimen painallusta tapahtumana. Esimerkiksi omassa create-funktiossa voidaan asettaa, että jos jokin näppäin menee pohjaan, niin kutsutaan tiettyä funktiota.

Tällöin näppäimen tilan kuuntelemisen asettaminen on kertaluontoista eikä näppäimen ti-

laa tarvitse kysyä jokaisella päivityskerralla. Phaser Dokumentaatio (2019) mukaan kuunteleminen tapahtuu ilmeisesti DOM Keyboard Eventeille, mikä käsittääkseni tarkoittaa, että selaimelta tulee ilmoitus, että näppäin on painettu, ja moottori reagoi tähän ilmoitukseen kutsumalla pelin kehittäjän asettamaa funktiota. En ole kuitenkaan itse kovinkaan tietoinen selaimen toiminnasta, joten en ole täysin varma asiasta.

Edellä kuvattujen pelin perustoimintojen (syöte, päivitys ja vaste) lisäksi Phaser tarjoaa valmiin fysiikkamoottorin, jota voi käyttää. Moottoreita on tarjolla kolme erilaista versiota, jotka erikoistuvat erilaisiin fysiikan mallinuksiin, mutta tässä työssä keskitymme vain yhteen niistä eli Arcade-fysiikan toimintaan. Arcade-fysiikan päivittäminen tapahtuu reaktiona update-tapahtumailmoitukseen, joka lähetetään vastaavasti kuin Game.js tiedoston step-funktiossa (katso koodilistaus 9.3). Update-ilmoitus lähetetään juuri ennen kuin kutsutaan pelin kehittäjän itsensä toteuttamaa update-funktiota, jolloin siis pelin fysiikka ja mahdollisesti muut update-ilmoitusta kuuntelevat osat tapahtuvat ennen kuin pelin kehittäjän oma update-funktio suoritetaan. Käytännössä arcade-fysiikan päivittäminen tapahtuu World.js tiedoston update-funktiossa (katso listaus 9.4).

6.1.2 Phaserin pääsilmukan päivitystiheys

Arcade-fysiikan päivityksessä hyödynnetään saatua tietoa siitä, että kuinka paljon aikaa on kulunut viimeisestä päivityskerrasta. Lisäksi funktiolla on tieto käytettävästä päivitystiheydestä (esim. 60 kertaa sekunnissa on oletuksena). Jos aikaa on kulunut vähemmän kuin, mitä käytettävä päivitystiheys ohjaa, niin päivitystä ei suoriteta. Jos taas aikaa on kulunut enemmän, niin päivityksiä toteutetaan mahdollisesti useampia kerralla, kunnes ollaan saatu kurottua päivitystiheys kiinni. Esimerkiksi jos aikaa on kulunut viime päivityksestä kaksi ja puoli kertaa päivitysväli, niin päivityksiä tehdään kaksi. Huomioitavaa on myös, että varsinaista päivitysfunktiota kutsuttaessa (eli this.step(fixedDelta);) käytetään parametrinä aikaa, vaikka aika onkin vakio. Tämä mahdollistaa helpomman päivitystiheyden vaihtamisen, kun kaikki muutokset määritellään suhteessa aikaan, jolloin peli toimii lähes täysin samalla tavalla vaikka päivitystiheys muuttuisikin.

Ratkaisu muistuttaa hyvin paljon luvussa Toistuvasti päivitys -malli esitettyä ratkaisua, jos-

sa myös toistettiin päivitys mahdollisesti useaan kertaan. Tämä on sikäli luonnollista, koska myös kehittäjät ovat luultavasti hyödyntäneet kyseisessä luvussa käyttämääni lähdettä, sillä kehittäjät viittavat samaan lähteeseen uutiskirjeessään Davey (2017). Uutiskirjeessä myös mainitaan, kuinka fysiikkamoottori hyödyntää säännöllistä päivitystä (katso luku Säännöllisesti päivittyvä versio), minkä on tarkoitus taata fysiikkamoottorin deterministinen toiminta.

Kuitenkin Phaserissä käytettävä ratkaisu eroaa luvun Toistuvasti päivitys -malli ratkaisusta siinä, että säännöllistä päivitystä käytetään vain fysiikkamoottorissa. Pelin kehittäjän itsensä tekemä päivitys taas päivittyy suoraan edellisestä päivityksestä kuluneen ajan mukaan (katso luku Ajan mukaan päivittyvä versio). Ratkaisu muistuttaa luvun Säännöllinen ja epäsäännöllinen päivitys erikseen ratkaisua, jossa deterministisyyden kannalta oleelliset osiot suoritetaan säännöllisellä päivityksellä ja muut epäsäännöllisellä päivityksellä. Pelin kehittäjän oma päivitys ei tosin tällainen välttämättä ole, mutta esimerkiksi Davey (2017) mainitsee, että moottorin sisäisesti muun muassa animaatio järjestelmä käyttää epäsäännöllistä päivitystä.

Kehittäjä perusteli Davey (2017) uutiskirjeessä ajan käyttämistä päivityksessä sillä, että selaimet eivät toimi samalla tavalla kuin perinteiset konsolit ja laitteet. Samalla kehittäjä viittaasi foorumiviestiin Gullen (2016), jonka perusteella arvelisin kyseessä olleen, että selaimissa ei pystytä niin helposti tarkkailemaan näytön piirtotiheyttä (eng. display rate). Gullen (2016) antaa ymmärtää, että säännöllinen päivitys ja siten myös säännöllinen kuvan piirto aiheuttaa epätasaista käyttäjää häiritsevää liikettä, kun päivitys ja kuvan piirto näytölle eivät osu samaan aikaan. Mallin alkuperäisessä julkaisussa eli Fiedler (2004) tämä ei sinänsä ollut ongelma, koska käytettiin interpolaatiota (katso luku Interpoloiva malli), mutta Phaserin tapauksessa interpolaatiota ei käytetä.

Vaikka pelin kehittäjän omassa update funktiossa käytetäänkin viime päivityksestä kulunutta aikaa hyödyksi, niin moottorin lähdekoodista (katso listaus 9.2) ja Davey (2017) uutiskirjeestä käy ilmi, että asia ei ole niin yksinkertainen. Tarkkaan ottaen pelin kehittäjän omalle update funktiolle annettu aikaa vastaava parametri on käsitelty erilaisten selaimeen liittyvien ominaisuuksien takia. Lähdekoodin (katso listaus 9.2 kommenteista käy ilmi, että muun muassa välilehden vaihtamisessa ilmenevään päivitysajan vaihtelevuuteen on varauduttu pitämällä pieni tauko, jolloin ei luoteta saatuihin päivitysaikoihin sellaisenaan.

Uutiskirjeessä käsittelyä kuvataan niin, että viimeisimpien päivitysten väliset ajat tallennetaan taulukkoon sen jälkeen, kun ne on pakotettu järkevälle välille (Davey 2017). Tämän jälkeen otetaan taulukon ajoista keskiarvo, jota käytetään päivitysaikana päivityksissä (Davey 2017). Myös kirjoitushetkellä lähdekoodissa näin näyttäisi tapahtuvan joidenkin muiden käsittelyiden lisäksi. Pienenä lisähuomiona on, että päivitysajan lisäksi lasketaan lähdekoodissa interpolaatiossa käytettävä arvo eli missä kohtaa ollaan ihanteellisten päivityskertojen välillä (esim. fysiikkamoottorin 60 fps on ihanteellinen päivitystiheys). Tätä arvoa ei kuitenkaan käsittääkseni käytetä missään, mutta se voi vihjata siihen, että joskus tulevaisuudessa moottori saattaisi tukea interpoloimalla saatavaa pävitystä kuten luvun Interpoloiva malli ratkaisussa.

6.1.3 Phaserin yhteenveto

Phaserin arcade-fysiikkamoottorissa näkyy selkeää samankaltaisuutta luvussa Toistuvasti päivitys -malli, jolla on pyritty fysiikkamoottorin deterministiseen toimintaan käyttämällä säännöllistä päivitystapaa. Kuitenkaann kaikki moottorin päivitykset eivät käytä säännöllistä päivitystapaa, jolloin Phaserillä tehtävä peli ei ole välttämättä täysin deterministinen. Esimerkiksi pelin kehittäjän itse tekemä update-funktio ja animaatiot käyttävät säännöllisen päivityksen sijaan muuttuvaa päivitysaikaa. Animaatiot eivät lähtökohtaisesti vaikuta deterministisyyteen haittaavasti, mutta pelin kehittäjän oma update-funktio voi tehdä niin. Phaserin kehittäjät ovat kuitenkin tietoisia tästä päätöksestä ja perustelevat ratkaisua selaimen erityispiirteillä.

Kaiken kaikkiaan Phaserin pääsilmukan rakenne vaikuttaa tarkkaan harkitulta ja suunnitellulta, ja siinä on samoja ominaisuuksia kuin, mitä on havaittu tämän työn teoriaosuudessa. Eniten tämän työn pääsilmukkamalleista Phaserin toiminta muistuttaa luvussa Säännöllinen ja epäsäännöllinen päivitys erikseen esiteltyä mallia, jossa on myös eroteltu päivitys kahteen eri osaan eli säännölliseen ja epäsäännölliseen päivitykseen. Phaser omalla toiminnallaan osoittaakin, että luvuissa Toistuvasti päivitys -malli ja Säännöllinen ja epäsäännöllinen päivitys erikseen esitellyt mallit ovat käyttökelpoisia ja ihan oikeassakin käytössä, eivätkä ole vain pelkkää teoriaa.

6.2 Unity

Unity on kaupallinen pelimoottori, jolla voidaan toteuttaa 2D ja 3D pelejä, ja sen ensimmäinen versio julkaistiin 2005, ja se on yhä aktiivisessa kehityksessä (Haas 2014). Unity Technologiesin kehittämä Unity eroaa kuitenkin Phaserin kaltaisista pelimoottoreista siinä, että se sisältää myös editorin, jonka avulla Unityllä tehtävät pelit on tarkoitus toteuttaa. Karkeasti sanottuna Unityllä on jopa mahdollista toteuttaa pelejä vaikkei osaisikaan ohjelmoida, mikä esimerkiksi Phaserin tapauksessa ei ole mahdollista. Unity myös tukee pelien tekemistä useille eri alustoille kuten monille konsoleille, mikä mahdollistaa pelin helpon sovittamisen (eng. porting) pienellä vaivalla.

Vaikka Unity on kaupallinen tuote, niin siitä on saatavilla täysin toimiva ilmainen versio, jota saa lähes vapaasti käyttää. Myös kaupallinen käyttö ja pelien julkaisu on sallittua kunhan yrityksen vuosituotto on alle $100 000 (tarkemmat ehdot kannattaa vielä tarkistaa, mutta perusidea on tämä) (Unity Technologies 2019b). Unity onkin yksi suurimpia kaupallisia pelimoottoreita ja siksi sen käyttämän pääsilmukan tutkiminen on siten tutkimisen arvoista. Lisäksi Unityn toiminta on hyvin dokumentoitua, mikä mahdollistaa tutkimisen. Myös Unityn C# lähdekoodia on julkaistu, joten sen tarkastelu on mahdollista, vaikkei lähdekoodi olekaan täysin avoimella lisenssillä (a reference-only lisenssi) (Unity Technologies 2018d).

Vaihtoehtoisesti olisin voinut myös valita Unityn sijaan Unreal Engine -pelimoottorin, joka vastaa melko pitkälti Unityä, mutta itselläni ei ole juurikaan kokemusta sen käyttämisestä. Sen sijaan Unity on itselleni huomattavasti tutumpi ja siten koen sen helpommaksi tutkia. Toisaalta olisi sinänsä mielenkiintoista tutkia molempia ja verrata niitä keskenään, koska ne ovat toistensa kilpailijoita, mutta ikävästi tutkimukseen on rajallisesti aikaa käytettävissä, ja haluan mahdollisimman monipuolisen tutkimuksen eri peleistä ja pelimoottoreista.

6.2.1 Unityn toiminnasta yleisesti

Unityllä pelejä tehtäessä on käytettävä ajatusmaailma hyvin oliosuuntautunutta ja lähes kaikki pelissä käsitellään olioiden kautta. Esimerkiksi jopa kamera nähdään omana olionaan. Myös jos pelinkehittäjä haluaa käyttää omaa skriptiä, niin lähtökohtaisesti se pitää asettaa osaksi jotakin oliota vaikkei olio tekisikään mitään muuta. Skriptit toimivatkin oliokohtai-

sesti olioiden osina. Esimerkiksi auto-oliolla voi olla osina oma tekoäly, fysiikka ja muita vastaavia, jotka toimivat toisistaan riippumattomina, mikä mahdollistaa uusien ominaisuuksien helpon lisäämisen, kun tarvitsee vain lisätä uusi osa olioon. Esimerkiksi, jos halutaan, että auto muuttuu punaiseksi törmätessä, niin tarvitsee vain lisätä uusi skripti oliolle sen sijaan, että muokattaisiin vanhoja skriptejä. Luonnollisesti sama punaiseksi tekevä skripti voidaan antaa samanaikaisesti useille eri olioille käyttöön. Unity myös sisältää valmiita osia, joita voidaan käyttää helposti omissa olioissa. Esimerkiksi jos olion halutaan noudattavan 2D-fysiikkaa, niin tätä tarkoitusta varten on osia valmiina, joita voidaan käyttää hyödyksi.

Osien lisäksi olioilla voi olla muita olioita lapsina, mikä lisää hierarkista ajattelua. Esimerkiksi auto-oliolla voi olla lapsina renkaat, jotka toimivat sinänsä autosta irrallisina, mutta voivat olla yhteydessä. Esimerkiksi lapsien sijainti lasketaan suhteessa isäntäolioon, joten isännän sijainnin muuttuessa muuttuu myös lasten sijainti. Pääasiallisesti kuitenkin kaikki pelin oliot ovat yhdessä listassa.

Skriptit taas toimivat olioissa niin, että ne reagoivat erilaisiin tapahtumiin (esimerkki Unity-skriptistä koodilistauksessa 9.5) (Unity Technologies 2018b). Esimerkiksi monissa skripteissä on Start-funktio, jota kutsutaan ennen olion ensimmäistä päivityskertaa (Unity Technologies 2018b). Update-funktiota taas kutsutaan aina jokaisella pelin päivityskerralla. Nämä funktiot ovat valmiiksi nimettyjä ja ne tulevat MonoBehaviour-luokasta, jonka oman skriptin luokka perii. Muita MonoBehaviourin-luokan funktioita on esimerkiksi olion törmätessä toiseen olioon kutsuttava OnCollisionEnter-funktio. Näitä kaikkia funktioita kutsutaan varsinaisen pelimoottorin puolelta, jolloin suoritus siirtyy omaan skriptiin, jonka jälkeen suoritus palautetaan pelimoottorin puolelle (Unity Technologies 2018b). Tämän ratkaisun seurauksena ohjelmoijan tehtäväksi jää vain kertoa oliolle, kuinka se reagoi erilaisiin tapahtumiin ilman, että tarvitsee tuntea pelimoottorin toimintaa tarkemmin.

Unityn käyttämä ratkaisu on samankaltainen kuin, mitä Phaser käyttää. Molemmat toimivat ajatuksella, että pelimoottori huolehtii ohjemoijan luomien funktioiden kutsumisesta. Kuitenkin erona on se, että Unityssä on useita olioita ja siten myös eri skriptejä joita pelimoottori kutsuu, kun taas Phaserissä on esimerkiksi vain yksi update-funktio, jota pelimoottori kutsuu, jolloin jätetään pelin kehittäjän vastuulle huolehtia päivityksen toteuttaminen kaikille pelin kohteille. Ratkaisujen erot saattavat rohkaista erilaiseen tapaan ohjelmoida. Phaseris-

sä luultavammin tehdään päivitykset eri olioille keskitetysti yhdestä paikasta käsin, kun taas Unityssä päivitykset tehdään helpommin useissa eri olioissa erillisesti. Esimerkiksi Phaserissä ratkaisu voisi olla käydä kaikki autot läpi silmukassa punaiseksi muuttamista varten, kun taas Unityssä auto itse katsoisi muuttuako punaiseksi.

6.2.2 Unityn pääsilmukan rakenne ja päivitystiheys

Unityn oma dokumentaatio (Unity Technologies 2018c) kuvaa hyvin, missä järjestyksessä yksittäisen skriptin funktioita kutsutaan ja se esitetään selkeästi kaavion avulla. Tapahtumien järjestys ja kaavion silmukka rakenne viittavaat sinänsä pääsilmukoihin, mutta ei ole täysin varmaa, että yksittäisten skriptien suoritusjärjestys on sama kuin koko pelin pääsilmukan. Lähinnä erona voi olla, että teoriassa eri skriptien suoritus voisi olla eri vaiheissa samaan aikaan. Esimerkiksi toinen skripti voisi olla tekemässä animaation päivitystä siinä vaiheessa kun toinen skripti on vasta pelilogiikan päivityksessä. Tämän ominaisuuden haittana tosin olisi hankalasti ennakoitava käytös, kun skripti ei voisi tietää, missä vaiheessa muut skriptit ovat. Esimerkiksi, jos olion päivitys muuttaa olion väriä muiden olioiden sijainnin mukaan, niin olisi hyvä tietää, ovatko muut skriptit jo muuttaneet niiden sijainteja. Erityisesti deterministisyyden kannalta tämä on hyvin oleellinen tieto.

Edellä kuvatun hankaluuden takia arvelisin Unityn toimivan tavalla, jossa kaikkien skriptien samaa funktiota kutsutaan järjestyksessä ennen kuin siirrytään seuraavan funktion kutsumiseen. Käytännössä esimerkiksi Unity suorittaisi kaikkien olioiden päivityksen ennen kuin se siirtyisi suorittamaan animaation päivityksen kaikille olioille. Kuvaamani ratkaisu olisi mielestäni yksinkertaisin ratkaisu, joka mahdollistaisi deterministisyyden ja olisi todella helppoa toteuttaa ja ymmärtää. Lisäksi olettaen, että Unity ei hyödynnä rinnakkaisuutta, skriptien erilaisista suoritusnopeuksista ei olisi mitään hyötyä. Myöskään edes rinnakkaistettaessa se ei olisi välttämättä kannattavaa, koska deterministisyys kärsisi todella helposti.

Lisäksi jotkin tehtävät ovat luonteeltaan sellaisia, että ne ovat helpointa suorittaa kaikille olioille kerralla. Esimerkiksi fysiikan mallinnuksessa ja törmäyskäsittelyssä on oleellista, että voidaan päivittää kaikkien olioiden sijaintia samanaikaisesti. Tämän takia halutaan kaikkien skriptien olevan samassa vaiheessa eli esimerkin tapauksessa kaikki olisivat samaanai-

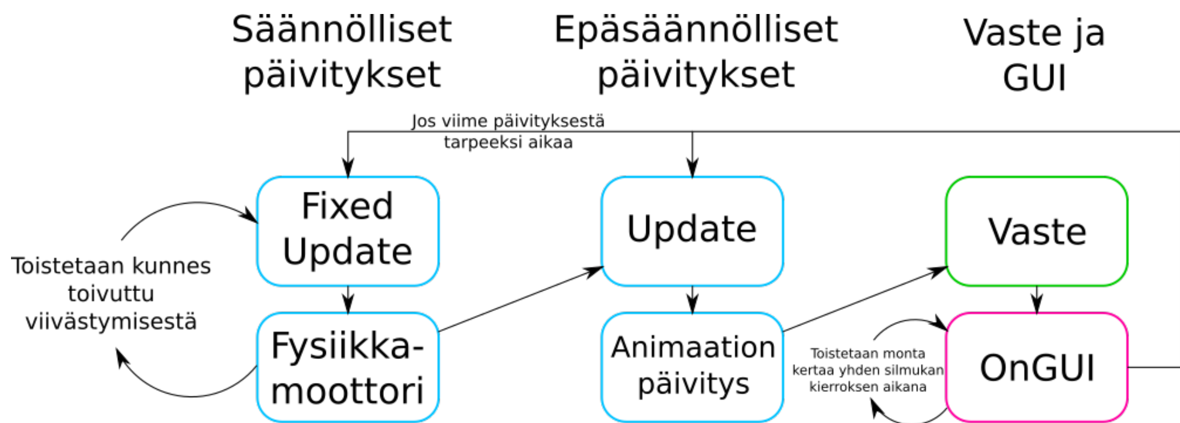
kaan fysiikan päivityksessä.

Edellä kuvattujen syiden, ja vastakkaisten todisteiden puuteen takia oletan, että Unityn skriptit suoritetaan samassa tahdissa, jolloin skriptien suorituksen silmukkarakenne voidaan yleistää koko pelin suoritusta kuvaavaksi pääsilmukaksi. Käytännössä tämä vain tarkoittaa, että pääsilmukan rakenne on pitkälti sama kuin skriptin käyttämän silmukan. Myös vaikka jos oletus on väärässä, niin skriptin suoritusjärjestyksen kuvaaminen jo itsessään on hyvin oleellinen osa Unityn toimintaa ja siitä voidaan saada tietoa, miten oman pääsilmukan voisi toteuttaa, mikä on yksi tämän työn tavoitteista.

Pääsilmukan tarkastelu skriptin käyttämän silmukan kautta voi tosin aiheuttaa, että jätetään huomioitta joitakin asioita, jotka tapahtuvat vain oikean pääsilmukan puolella. Esimerkiksi päivitysaikaan liittyvät muokkaukset, joita tehtiin myös Phaserissä (katso luku Phaserin pääsilmukan päivitystiheys), jäävät tarkastelematta. Toisaalta omien havaintojeni mukaan Unityn dokumentaatio kertoo pelimoottorin toiminnasta vain pelin kehittäjää koskevat tiedot. Tällaisia ovat esimerkiksi tieto, kuinka pelin kehittäjä voi suorittaa päivityksen Update-funktiossa, jossa voidaan käyttää päivitysaikaa. Tällöin ei kuitenkaan ole tarpeellista kertoa tarkkaan, kuinka päivitysaika saadaan laskettua pelimoottorin puolella.

Joka tapauksessa Unity Technologies (2018c) esittää kaavion, jossa kuvataan, missä järjestyksessä skriptien funktioita kutsutaan. Tästä kaaviosta on tehty tähän työhän yksinkertaistettu versio eli kuva 11, joka noudattaa muiden tämän työn kaavioiden tyyliä. Unity Technologies (2018c) esittämästä kaaviosta käy ilmi, että silmukan alussa suoritetaan säännölliset päivitykset kuten fysiikan päivitys ja siihen liittyvät tapahtumat. Myös FixedUpdate-funktiota kutsutaan, jossa pelin kehittäjä voi tehdä ne päivitykset, jotka halutaan suorittaa säännöllisesti (Unity Technologies 2018c). Tällaisia ovat esimerkiksi deterministisyyteen vaikuttavat päivitykset.

Säännöllinen päivitys noudattaa samaa ideaa kuin luvussa Toistuvasti päivitys -malli eli päivitykset voidaan suorittaa useasti peräkkäin, jos edellisestä päivityksestä on kulunut liian paljon aikaa. Vastaavasti päivitykset voidaan myös kokonaan ohittaa, jos aikaa on kulunut liian vähän (Unity Technologies 2018c). Tällä ratkaisulla halutaan luultavimmin taata fysiikkamoottorille varma ja deterministinen toiminta, ja samalla annetaan myös kehittäjälle



Kuvio 11. Yksinkertaistettu kaavio Unityn skriptien käyttämästä silmukkarakenteesta. Pohjautuu Unity Technologies (2018c) esittämään kaavioon, joka on tähän yksinkertaistettu ja muutettu muiden tämän työn kaavioiden tyyliseksi.

mahdollisuus tehdä osa omista päivityksistään säännöllisesti, jos kehittäjä kokee sen tarpeelliseksi.

Säännöllisten päivitysten jälkeen tulee epäsäännölliset päivitykset, kuten kehittäjän oman Update-funktion kutsu ja animaatioiden päivitys (Unity Technologies 2018c). Animaatiot eivät luultavimmin vaikuta pelin toimintaan, joten deterministisyyden kannalta niitä ei tarvitse suorittaa säännöllisesti ja lisäksi animaatiot hyötyvät mahdollisesti epäsäännöllistä päivitystä tiheämmästä päivitysnopeudesta (Valente, Conci ja Feijó 2005). Update-funktio taas sisältää ne toiminnot, jotka kehittäjä on funktioon asettanut, mutta hyvä käytäntö varmaan olisi, että siellä toteutettaisiin sellaiset päivitykset, jotka eivät vaikuta deterministisyyteen. Tosin omien kokemukseni mukaan Update-funktioon laitetaan aivan kaikki päivitykset sen sijaan, että käytettäisiin FixedUpdate-funktiota. Tämä ei sinänsä ole haitallista, jos ei aioita hyödyntää deterministisyyttä, mutta joka tapauksessa Unity ainakin tarjoaa mahdollisuuden helposti erotella säännölliset ja epäsäännölliset päivitykset toisistaan.

Säännöllisten päivitysten jälkeen piirretään pelimaailmasta kuva näytölle ilman sen kummempia (Unity Technologies 2018c). Tämä vaihe on tosin jaettu useisiin osiin, jotta pelin kehittäjä voi itse vaikuttaa piirtämisen välivaiheisiin. Tämän jälkeen kutsutaan OnGUI-funktiota (Unity Technologies 2018c). Tämän tekee mielenkiintoiseksi se, että sitä voidaan kutsua useita kertoja peräkkäin. Unity Technologies (2018c) dokumentaation mukaan OnGUI-

funktiota kutsutaan seurauksena graafisen käyttöliittymän tapahtumista. Uskoakseni tämä vain tarkoittaa, että jokaista tapahtumaa kohti kutsutaan OnGUI-funktio kerran. Yksityiskohdat eivät ole kuitenkaan oleellisia pääsilmukan kannalta vaan liittyy enemmän graafisen käyttöliittymän toteutuksen suunnitteluun kuin pääsilmukoihin. Kun kaikki OnGUI-kutsut on suoritettu, niin palataan takaisin silmukan alkuun lopun viimeistelyiden jälkeen.

Edellä kuvattu silmukan rakenne on hieman yksinkertaistettu, jotta tulisi perusidea selville, ja sen takia osa funktiokutsuista on jätetty mainitsematta tai ne on kuvattu kokonaisuutena. Esimerkiki Unity tukee LateUpdate-funktiota, joka voidaan suorittaa epäsäännöllisten päivitysten lopuksi (Unity Technologies 2018c). Jos yksittäisten funktiokutsujen järjestyksen haluaa tietää, niin kannattaa katsoa Unity Technologies (2018c) esittämä kaavio.

Kaiken kaikkiaan Unityn skriptien silmukan rakenne muistuttaa luvussa Säännöllinen ja epäsäännöllinen päivitys erikseen esiteltyä mallia, jossa on myös eroteltu epäsäännölliset ja säännölliset päivitykset erilleen. Myös Phaser käyttää samaa ajatusta fysiikkamoottoreissa (katso luku Phaserin pääsilmukan rakenne), vaikkei se ole aivan yhtä pitkälle vienytkään erottelua. Esimerkiksi Phaserillä on vain yksi update-funktio, joka toimii epäsäännöllisellä päivityksellä, kun taas Unity tukee sekä epäsäännöllistä (Update-funktio) että säännöllistä päivitystä (FixedUpdate-funktio) samanaikaisesti. Myös huomattavaa on, että molemmat pelimoottorit (Unity ja Phaser) ovat päätyneet deterministiseen eli säännölliseen päivitysratkaisuun fysiikkamoottorien suhteen. Tämä vaikuttaa viittaavan siihen, että yleisesti fysiikkamoottoreissa koetaan säännöllinen päivitys välttämättömäksi.

6.2.3 Rinnakkaistus Unityssä

Koska Unityn dokumentaatio on tehty pelien kehittäjille, niin Unityn omaa toimintaa selitetään melko vähän, minkä takia on hankalaa arvioida kuinka rinnakkaistettu Unity on. Toisaalta dokumentaatiossa tuskin erityisesti korostettaisiin edes, jos Unity ei olisi rinnakkaistettu, joten rinnakkaistettamattomuudesta on hankalaa saada varmoja todisteita. Kuitenkin löytyy ainakin joitakin lähteitä, joista käy selväksi, että osa Unitystä toimii rinnakkaistetusti.

Unity Technologies, Anthony (2014) blogin julkaisun mukaan Unity siiryi käyttämään monisäikeistä vaatteiden simulaatiota. Samassa julkaisussa myös kerrottiin, kuinka uuden PhysX3

kehityspakin (eng. System Development Kit eli SDK) avulla voidaan jatkossa hyödyntää monisäikeisyyttä fysiikan mallinnuksessa. Julkaisu on tosin useamman vuoden vanha, joten tilanne on voinut muuttua. Uudemmassa Yakovlev (2018) Unityn blogikirjoituksessa selkeästi sanotaan Unityn fysiikkamoottorin käyttävän säikeistystä, joten siirtyminen rinnakkaistettuun fysiikkamoottoriin ilmeisesti tapahtui onnistuneesti. Mielenkiintoisesti artikkelissa sanotaan myös, että Unityn käyttämä PhysX varmistaa fysiikkasimulaatiossa saman lopputuloksen samoilla syötteillä eli takaa simulaation deterministisyyden, mikä viittaa siihen, että ainakin Unityn fysiikkamoottori on pyritty tekemään deterministiseksi. Tämä on sinänsä hieno saavutus, koska se on samanaikaisesti rinnakkaistettu fysiikkamoottori, joten kehityksessä on varmasti ollut omia haasteitansa. Yleensä kuitenkin rinnakkaistus helposti estää deterministisyyden.

Fysiikkamoottorin lisäksi ilmeisesti myös kuvan piirtäminen näytölle on mahdollista tehdä monisäikeisesti. Tämä käy ilmi Unityn tutoriaalista (Unity Technologies 2019a), jossa kerrotaan Unityn tarjoamista eri vaihtoehdoista kuvan piirrolle. Monisäikeisyys tosin vaatii käytettävän laitteiston ja grafiikka sovellusliittymän (eng. Application Programming Interface eli API) tuen, mikä voi estää monisäikeisyyden joillakin laitteilla.

Edellä kuvattujen pelimoottorin sisäisten ominaisuuksien lisäksi Unity tarjoaa työkaluja, joita kehittäjä voi käyttää rinnakkaistuksessa apuna. Esimerkiksi resurssien lataus on mahdollista tehdä asynkronisesti (Unity Technologies 2018e). Käytännössä tämä tarkoittaa, että resursseja voidaan ladata taustalla muun pelin toimiessa normaalisti, mikä on erityisen hyödyllistä, jos latauksessa kestää useampia sekunteja. Tämä ratkaisu muistuttaa luvussa Osien puhdas irrottaminen omaan säikeeseen esiteltyjä ratkaisuja, joissa myös mainittiin, että resurssien lataus on usein helposti rinnakkaistettavissa. Toisaalta dokumentaatio ei suoraan sano, että resurssien latauksen toteutus hyödyntäisi monisäikeistystä, mutta asynkronisyys jo itsessään oli tärkein syy, miksi resurssien lataus kannattaisi rinnakkaistaa.

Tarkempaa rinnakkaistusta varten Unity tarjoaa oman järjestelmän C# Job System, jolla on tarkoitus voida toteuttaa rinnakkaistus turvallisesti ja yksinkertaisesti (Unity Technologies 2018a). Järjestelmän avulla kehittäjä voi luodan varsinaisten säikeiden sijaan tehtäviä, jotka pelimoottorin itsensä luomat säikeet suorittavat (Unity Technologies 2018a). Järjestelmä sisältää erilaisia rinnakkaistusta helpottavia työkaluja kuten erillisen tavan rinnakkaistaa sil-

mukoita, mikä laskee kehittäjän kynnystä rinnakkaistaa (Unity Technologies 2018a).

Lisäksi koska Unityllä tehtävät pelit sisältävät aivan tavallisia C# ohjelmakoodia, niin rinnakkaistus on mahdollista suorittaa myös täysin itse manuaalisesti. Tässä tapauksessa kuitenkin rinnakkaistusta rajoittaa se, että Unityn sovellusliittymä (API) ei ole säieturvallinen (Bonastre 2016). Tämän takia Unity sovellusliittymää tulee kutsua vain pääsäikeestä (eng. main thread), mikä saattaa hankaloittaa joidenkin toteutusten tekemistä (Bonastre 2016). Lähde on tosin muutaman vuoden vanha, joten tilanteeseen on voinut tulla muutoksia, mutta pikaisella tarkastuksella ainakin foorumien vastauksissa asian sanotaan yhä olevan näin eikä löytynyt väitettä kumoavaa tietoa asiasta.

Löydettyjen lähteiden nojalla Unity vaikuttaisi käyttävän rinnakkaistusta lähinnä yksittäisten tehtävien sisällä, mutta tämä voi johtua löytämieni lähteiden rajallisuudesta. Joka tapauksessa Unity käyttää onnistuneesti rinnakkaistusta yksittäisissä tehtäväkokonaisuuksissa kuten fysiikkamoottorissa. Yksittäisten tehtäväkokonaisuuksien rinnakkaistaminen on kuitenkin helpompaa toteuttaa ja pienemmällä todennäköisyydellä aiheuttaa ongelmia kuten epädeterministisyyttä. Lisäksi se, että Unity tarjoaa työkaluja kehittäjälle rinnakkaistuksen toteuttamiseksi itse, laskee kehittäjän kynnystä hyödyntää rinnakkaisuutta suorituskyvylle raskaissa tehtävissä, mikä mahdollistaa pelien tehokkaamman toiminnan.

6.3 Quake

Quake on ensimmäisen persoonan ammuntapeli, jonka id Software julkaisi vuonna 1996 (id Software 2019a). Kolmiulotteisen OpenGL:ää käyttävän grafiikan lisäksi Quake sisälsi myös mahdollisuuden pelata verkon yli muiden pelaajien kanssa moninpelinä (id Software 2019a). Nämä ominaisuudet olivat aikaansa nähden kohtuullisen uusia ja kehittyneitä, mikä luultavimmin vaikutti pelin suosioon ja merkittävyyteen.

Koska Quake on kohtuullisen laajasti tunnettu peli, niin se on sikäli otollinen kohde tutkittavaksi. Pelin tutkiminen on myös mahdollista, koska sen lähdekoodi on julkaistu, mikä on kaupallisissa peleissä melko harvinaista. Lähdekoodin julkaisemisen jälkeen, koodia on jo valmiiksi analysoitu, mikä myös helpottaa tutkimuksen tekemistä, koska voin hyödyntää aiempia analyysejä omassa tarkastelussani. Lisäksi onnistuin myös löytämään valmiin tutki-

muksen, jossa oli toteutettu rinnakkaistus Quaken serverille, jolloin kyseiseen tutkimukseen vertaamineen tuo mielenkiintoista lisäarvoa omalle tutkimukselleni.

Vaikka Quake on kohtuullisen vanha peli, eikä se siten tuo tuoreinta näkökulmaa peleissä käytettävistä pääsilmukoista, se on kuitenkin kaupallisessa tarkoituksessa kehitetty varsinainen peli eikä pelkästään pelimoottori. Se siten tuo mahdollisesti erilaista näkökulmaa pääsilmukoiden tarkasteluun kuin tässä työssä käytetyt Unity ja Phaser, jotka ovat puhtaita pelimoottoreita. Uutena näkökulmana voidaan mahdollisesti havaita käyttötarkoitukseensa erikoistuneempi ratkaisu kuin, mitä yleiskäyttöiset pelimoottorit tarjoavat.

Quaken lähdekoodin laajuuden takia päädyin käyttämään oman lähdekoodin tarkastelun apuna valmista analyysiä Quaken toiminnasta. Erään analyysin eli Sanglard (2009) avulla havaitsin, että Quaken käyttämä pääsilmukka löytyy sys_win.c tiedostosta. Tällaisia tiedostoja löytyy kuitenkin Quake projektista ainakin kolme kappaletta, joissa kaikissa oli oma pääsilmukan toteutus. Eroavaa ratkaisuissa on, että (oletettavasti) yksi niistä on perusversio, johon sisältyy sekä asiakkaan, että palvelimen toiminta. Muissa tiedostoissa on taas Quake World -päivityksen mukana tulleet asiakas- ja palvelinversiot pääsilmukasta erotettuina toisistaan omiin tiedostoihinsa. Tässä työssä keskitytäänkin Quake World -päivityksen versioihin pääsilmukasta. Perusteena ratkaisulle on, että päivityksessä asiakas ja palvelin on erotettu selkeämmin toisistaan erilleen, mikä helpottaa pääsilmukoiden tarkastelua.

6.3.1 Quaken palvelimen rakenne ja päivitystiheys

Quake World -päivityksen palvelimen pääsilmukkasta (katso koodilistaus 9.7) nähdään, että varsinainen pääsilmukan toiminta tapahtuu SV_Frame-funktiossa, jolle annetaan parametrinä viime päivityksestä kulunut aika. SV_Frame-funktiosta (katso listaus 9.8) havaitaan, että alkuvalmisteluiden jälkeen ensimmäisenä toteutetaan pelimaailman fysiikan päivittäminen, niiltä osin kuin, mitä ilman käyttäjien syötteitä pystytään. Itsenäisten päivitysten jälkeen vasta luetaan asiakkailta tulleet viestit ja toteutetaan niihin liittyvät komennot. Lopuksi lähetetään asiakkaille viestit pelimaailman tilasta.

Edellä kuvattu toiminta voidaan yleistää tässä työssä käsiteltyyn teoriaan ajattelemalla asiakkailta tulleita viestejä syötteenä ja asiakkaille meneviä viestejä vasteena. Päivitykset taas

suoritetaan kahdessa vaiheessa eli päivitykset, jotka voidaan suorittaa ilman syötteitä ja päivitykset, joihin tarvitaan syötteitä. Nämä päivitykset mielenkiintoisesti suoritetaan eri vaiheissa niin, että syötteistä riippumattomat päivitykset suoritetaan ensin.

Tällä on mahdollisesti pyritty antamaan riippumattomille syötteille niin sanotusti etuoikeus, jotta ammukset ja vastaavat varmasti liikkuvat ensin. Toisaalta syötteet eivät välttämättä muuta pelimaailman tilaa samalla tavalla kuin syötteistä riippumattomat päivitykset. Esimerkiksi aiemmassa päivityksessä saatetaan liikuttaa kohteita nopeuden mukaan, kun taas syötteistä riippuvassa päivityksessä mahdollisesti muutetaan kohteen nopeutta itsessään ilman, että sijaintia muutetaan. Tällaisessa ratkaisussa on luontevaa erottaa päivitykset erilleen toisistaan.

Tämä on kuitenkin vain valistunut arvaus, kuinka Quaken serveri saattaisi toimia. SV_Physics-funktiota ja sen kutsumia funktioita tutkimalla vaikuttaisi kuitenkin, että teoria ei pitäisi paikkansa, sillä SV_Physics-funktio vaikuttaisi tarkoituksella ohittavan pelihahmojen fysiikan päivittämisen (lukee suoraan ”clients are run directly from packets”). Tällä saatetaan toki viitata johonkin muuhun myös, koska herää kysymys, että päivitetäänkö hahmon liikettä tilanteissa, joissa viestiä ei tullut asiakkaalta. Varman vastauksen saaminen toimintaan on kuitenkin turhaan haastavaa, koska pitäisi perehtyä Quaken oman komentojärjestelmän toimintaan. Tarkka toiminta ei myöskään ole pääsilmukan ja siten aiheen kannalta oleellista, joten jätän komentojen tutkimisen väliin. Mahdollisesti erillisillä päivityksillä on myös vain haluttu selkeästi erottaa pelaajien toiminnot erilleen pelimaailman yleisistä päivityksistä, jotta koodia olisi helpompi ymmärtää ja suunnitella.

Koodia tarkastelemalla syötteistä riippumaton päivitys vaikuttaisi käyttävän viime päivityksestä kulunutta aikaa suoraan (katso luku Ajan mukaan päivittyvä versio). Funktio kutsuja seuraamalla löytyy lopulta SV_Physics_Toss-funktiosta koodirivi ”VectorScale (ent->v.velocity, host_frametime, move);”, joka viittaa vahvasti tähän. Tämä on sinänsä mielenkiintoinen havainto, koska uudemmissa pelimoottoreissa kuten Unityssä ja Phaserissä käytettiin luvussa Toistuvasti päivitys -malli esiteltyä ratkaisua, jossa päivitys tapahtuu säännöllisellä päivitystavalla. Muuttuvan päivitysajan käyttämisessä ongelmana on lähinnä se, että peli ei ole silloin deterministinen (katso luku Pelin deterministisyys), mutta tämä ei ole iso puute, jos pelissä ei aiota hyödyntää deterministisyyttä tarvitsevia ominaisuuksia.

Kaiken kaikkiaan Quaken palvelimen pääsilmukka toimii melko yksinkertaisesti. Käytännössä ensimmäisenä vain suoritetaan syötteestä riippumattomat päivitykset, jonka jälkeen luetaan asiakkaiden viestit (syöteet) ja toteutetaan niihin liittyvät toiminnot (syötteestä riippuva päivitys). Lopuksi lähetetään asiakkaille tieto pelimaailman tilasta (vaste).

6.3.2 Quaken asiakkaan rakenne ja päivitystiheys

Quake World -päivityksen asiakkaan pääsilmukasta (katso koodilistaus 9.6) nähdään, että varsinainen pääsilmukan toiminta tapahtuu Host_Frame-funktiossa (katso listaus 9.9), jolle annetaan parametrinä viime päivityksestä kulunut aika. Host_Frame-funktion alussa tehdään päivitysajalle tarkistuksia, kuten varmistetaan, että edellisestä päivityksestä on kulunut tarpeeksi aikaa. Perusversion _Host_Frame-funktion kommenteissa päivitysnopeuden rajoittamista oltiin perusteltu sillä, että liian suuri päivitysnopeus voi johtaa palvelimen tukkeutumiseen viestien määrän takia. Toisaalta liian korkea päivitysnopeus ei enää muutenkaan hyödytä, koska esimerkiksi näytön kuvan piirtämistiheys ei pysy välttämättä mukana.

Toisena rajoituksena päivitysajalle Host_Frame-funktiossa vaikuttaisi olevan se, että päivitysaika voi olla korkeintaan tietyn suuruinen. Jos päivitysaika on rajaa korkeampi, niin päivitysajaksi asetetaan yläraja. Tätä rajoitusta ei lähdekoodissa kommentoida, mutta arvelisin sen liittyvän siihen, että halutaan varmistaa päivitysten olevan tarpeeksi pieniä. Kuten luvussa Säännöllisesti päivittyvä versio todettiin, niin liian suurilla päivitysväleillä voi ilmetä epätoivottua käytöstä, kuten kohteiden menemistä seinien läpi. Vaikkei Quake käytäkään säännöllistä päivitystapaa, niin päivitysajalle on ilmeisesti asetettu rajat, joiden puitteissa pystytään takaamaan pelin oikea toiminta tarpeeksi hyvin.

Vaikka päivitysaikaa rajoitetaankin, niin se ei kuitenkaan hidasta pelin toimintaa, mikä on ehdottoman tärkeää palvelimilla toimivissa peleissä. Hidastamisen sijaan pikemminkin kulutetaan jälkeen jäädystä ajasta hieman joka silmukan kierroksella kunnes ollaan saatu kurottua jälkeen jääty aika kiinni. Tämä ratkaisu muistuttaakin hieman luvussa Toistuvasti päivitys -malli esiteltyä ratkaisua, jossa myös vastaavasti huomioidaan tilanne, jossa on jääty päivityksissä jälkeen. Kuitenkin selkeänä erona on, että luvun Toistuvasti päivitys -malli ratkaisussa toistetaan vain päivitysvaihetta, kun taas Quaken asiakkaan käyttämässä ratkaisussa

toistetaan kaikki silmukan vaiheet.

Päivitysajan käsittelyn jälkeen Quaken asiakkaassa kerätään syötteet syötelaitteilta kuten hiireltä (kommentoitu *"allow mice or other external controllers to add commands"*), jonka jälkeen komennot suoritetaan. Tämän jälkeen luetaan palvelimelta tulleet viestit, jonka jälkeen lähetetään omaa tilaa kuvaavat viestit palvelimelle. Lopuksi toteutetaan muut vasteet kuten näytön kuvan piirtäminen ja audion soittaminen. Mielenkiintoisesti kuitenkin piirtämistä ennen ennakoidaan pelimaailman tilaa.

Ennakoimisella on pyritty ratkaisemaan verkkoviestinnästä johtuva viive, jonka seurauksena asiakkaan pelimaailman tilanne on hieman myöhässä verrattuna palvelimen pelimaailman tilanteeseen (Sanglard 2009). Käytännössä tämä näkyisi siinä, että jopa pelaajan omat komennot tapahtuisivat pienellä viiveellä, joka haittaa pelikokemusta. Ennakoimalla pystytään arvioimaan, minkälainen pelimaailman tilanne on palvelimen puolella, jossa varsinainen pelin päivitykset oikeasti tapahtuvat.

Sanglard (2009) mukaan Quakessa ennakointi tapahtuu kokonaan asiakkaan puolella ja se on jaettu kahteen loogisesti erilaiseen kokonaisuuteen eli oman toiminnan ennakointiin ja muiden pelaajien ennakointiin, mikä ilmenee myös melko selkeästi lähdekoodista (katso koodilistaus 9.9). Oman toiminnan ennakointi tapahtuu toteuttamalla omat paikallisesti luodut komennot viimeisimpään palvelimelta saatuun tietoon pelimaailmasta (Sanglard 2009). Käytännössä vain paikallista tietoa pelimaailman tilasta päivitetään omilla komennoilla. Tämä on sikäli tarkka ennuste, koska se sisältää myös törmäystarkastelun (Sanglard 2009).

Muille pelaajille luonnollisestikaan ei voida toteuttaa ennakointia samalla tavalla, koska ei voida tietää muiden pelaajien tekemiä komentoja (Sanglard 2009). Tämän takia ennakointi muille pelaajille tehdään ekstrapolaation (eng. extrapolation) avulla (Sanglard 2009). Ekstrapolaatio on sama kuin interpolointi, mutta sen sijaan, että arvioitaisiin uusi arvo kahden jo tunnetun arvon välille, niin arvioidaankin uusi arvo välin ulkopuolelta. Esimerkiksi arvioidaan, missä pelihahmo on hetken kuluttua, jos pelihahmo liikkuu samalla nopeudella samaan suuntaan. Tämä ratkaisu muistuttaa hieman luvussa Interpoloiva malli esiteltyä ratkaisua, mutta interpoloinnin sijaan vain ekstrapoloidaan, mikä on toteutuksen kannalta pieni muutos.

Quaken Quake World -päivityksen asiakas on kaiken kaikkiaan melko yksinkertainen eikä tapahtumien järjestyksessä ole mitään erityistä. Kuitenkin päivitysajan rajoittaminen on mielenkiintoinen ominaisuus, mikä luultavimmin huolehtii siitä, ettei simulaatiossa tehdä liian suuria päivityksiä kerralla. Ennakoinnin käyttö taas tuo selvästi esille, että palvelinta käyttävät moninpelit kohtaavat omanlaisiaan haasteita, jotka voivat vaikuttaa pääsilmukkaan. Toisaalta palvelimen ja asiakkaan yhteistoiminta ylipäätään on omanlaisensa haaste, joka vaatii suunnittelua.

6.3.3 Rinnakkaistus Quakessa

Quake ei varsinaisesti hyödynnä rinnakkaistusta toteutuksessaan johtuen siitä, että kun peli julkaistiin, niin silloin ei ollut juurikaan moniytimisiä suorittimia kuluttajien saatavilla. Tosin moninpelin toteutus voidaan nähdä rinnakkaistettuna sovelluksena, jonka suoritus on jaettu useille eri laitteille. Moninpelin toteutuksessa voidaankin havaita samoja haasteita kuten rinnakkaistettaessa yleensäkin, kuten esimerkiksi asiakkaiden ja palvelimen välinen synkronointi.

Kuitenkin vaikkei alkuperäisessä Quakessa olekaan varsinaista rinnakkaistusta, niin onnistuin löytämään Abdelkhalek ja Bilas (2004) tutkimuksen, jossa Quaken palvelin muokattiin hyödyntämään rinnakkaistusta. Abdelkhalek ja Bilas (2004) tutkimuksessa onnistuttiin parantamaan palvelimen suorituskykyä niin, että palvelin pystyi käsittelemään 25 % enemmän pelaajia, mikä osoittaa pelien käyttämien palvelimien voivan hyötyä rinnakkaistuksesta huomattavasti. Tässä luvussa tutustutaan kyseisen tutkimuksen toteutustapaan sillä se tarjoaa toimivan ratkaisun rinnakkaistaa Quake ja siten voidaan katsoa, kuinka se yleistyy tässä työssä esitettyyn teoriaan.

Abdelkhalek ja Bilas (2004) tutkimuksessa eroteltiin palvelimen tehtävät kolmeen vaiheeseen: pelimaailman päivitys, asiakkailta tulleiden viestien käsittely ja viestien lähettäminen asiakkaille. Tutkimalla vaiheiden suoritusaikoja Abdelkhalek ja Bilas (2004) havaittiin, että maailman päivitykseen kului vain 5 % suoritusajasta, joten sen rinnakkaistaminen ei koettu otolliseksi. Tämä ratkaisu on Amdahlin lain (katso luku Amdahlin laki) kannalta järkevää, koska 5 % osuutta ei voida juurikaan tehostaa verrattuna muihin vaihtoehtoihin. Tämän ta-

kia tutkimuksessa päädyttiin rinnakkaistamaan vain asiakkailta tulleiden viestien käsittely ja viestin lähettäminen asiakkaille.

Näiden vaiheiden rinnakkaistuksessa päädyttiin ratkaisuun, jossa yksi säie huolehtii ryhmästä asiakkaita niin viestien käsittelyssä kuin viestien lähettämisessä (Abdelkhalek ja Bilas 2004). Käytetty ratkaisu on siis datan rinnakkaistamista (katso luku Datan rinnakkaistus), mikä vaikuttaa hyvältä ratkaisulta, koska datan rinnakkaistaminen tarjoaa lähtäkohtaisesti paremman skaalautuvuuden kuin tehtävien rinnakkaistaminen. Palvelimissa muutenkin skaalautuvuus on oleellista, koska asiakkaiden määrä voidaan teoriassa kasvattaa huomattavasti.

Asiakkaiden rinnakkaistus on myös viestien vastaanottamisen ja lähettämisen kannalta helppoa, koska niissä ei ole juurikaan tarvetta synkronoinnille (Abdelkhalek ja Bilas 2004). Kuitenkin viestien käsittelyssä, jossa voidaan joutua liikuttamaan pelihahmoja, joudutaan käyttämään synkronointia apuna (Abdelkhalek ja Bilas 2004). Synkronointia varten tutkimuksen ratkaisussa päädyttiin käyttämään alueellisia lukkoja apuna (Abdelkhalek ja Bilas 2004). Käytännössä ennen kohteen liikuttamista lukittiin kaikki kohteen ympärillä olevat alueet, joihin liikuttaminen saattaisi vaikuttaa esimerkiksi törmäyksen kautta. Tämä oli sikäli luonteva ratkaisu, koska Abdelkhalek ja Bilas (2004) mukaan Quaken palvelin jo valmiiksi käytti puurakennetta (käytetty ratkaisu oli tarkkaan ottaen englanniksi *areanode tree*) hahmottamaan, mihin kohteisiin kappaleet voivat vaikuttaa toiminnallaan. Alueellisia lukkoja pohdin myös itse luvussa Tehtävän sisäinen rinnakkaistus.

Alueiden lukitsemisen ansiosta synkronointi onnistuu hyvin ilman, että tulee rinnakkaistuksen ongelmia. Kuitenkin rinnakkaistettu versio saattaa suorittaa komentoja kohteille vaihtelevassa järjestyksessä suorituskertojen välillä, kun taas alkuperäisessä versiossa ne tapahtuivat samassa järjestyksessä (Abdelkhalek ja Bilas 2004). Tämän takia ratkaisu ei ole deterministinen, eikä peliä siten voida helposti toisintaa. Toisaalta oman analyysini pohjalta alkuperäinen versio ei vaikuttanut muutenkaan deterministiseltä, joten rinnakkaistaminen ei sinänsä aiheuta uusia ongelmia.

Kaiken kaikkiaan Abdelkhalek ja Bilas (2004) tutkimuksessa toteutettu rinnakkaistus on varsin onnistunut, koska saavutettiin 25 % suurempi pelaajamäärä. Suoritusajasta kuitenkin 35

% kului lukkojen avautumisia odotellessa, mitä voidaan tarkemmalla lukkojen käytöllä vähentää (Abdelkhalek ja Bilas 2004). Myös muiden säikeiden suoritusten odottamiseen kului huomattavasti aikaa eli Abdelkhalek ja Bilas (2004) mukaan jopa 40 % suoritusajasta. Tämä viittaa oman näkemykseni mukaan siihen, että tehtävien jakamista eri säikeiden välillä tulisi vielä hioa ja mahdollisesti siirtyä käyttämään säieallasta. Tutkimuksessa taas käytettiin ratkaisua, jossa säikeille ennalta määrättiin asiakkaat, joiden viestejä käsitelevät. Rinnakkaistuksen onnistumisen lisäksi on rohkaisevaa, että Abdelkhalek ja Bilas (2004) tutkimuksessa käytettiin onnistuneesti alueellisia lukkoja hyödyksi, sillä se osoittaa, että esittelemäni teoria asiasta on ihan käytännössä toimiva ratkaisu.

6.4 Spacewar

Spacewar! (tässä työssä käytetään nimitystä Spacewar) on PDP-1:lle kehitetty tietokonepeli, joka kehitettiin vuosina 1961-1962 (Computer History Museum 2019). Pelissä kaksi pelaajaa ohjaavat avaruusaluksia tarkoituksena tuhota toisen pelaajan alus (Computer History Museum 2019). Pelimaailman keskellä on myös tähti, joka imee pelaajien aluksia itseänsä kohti. Lisäksi peli sisältää taustalla oikeaa maailmaa vastaavan tähtitaivaan.

Spacewar on sikäli merkittävä videopeli, koska se on yksi ensimmäisimpiä kehitettyjä videopelejä, ja siten sen käyttämän pääsilmukan tutkiminen tuo mielenkiintoista näkökulmaa tähän työhön. Spacewariaa kehittäessä ei välttämättä ollut tarjolla paljoakaan tietoa pääsilmukoista, tai miten ylipäätään videopelejä voitaisiin edes tehdä. Tämän takia on kiinnostavaa tutkia, minkälaiseen ratkaisuun Spacewarissa on päädytty käyttämään. Lisäksi Spacewar on toiminut esikuvana monille myöhemmistä videopeleistä, minkä takia Spacewarin käyttämät ratkaisut ovat voineet periytyä myöhemmillekin peleille.

Vaikka Spacewar on vanha videopeli, eikä se siten tarjoa kuvaa käytettävien pääsilmukoiden nykytilanteesta, niin sen vertaaminen uudempiin peleihin tuo tutkimukselle erilaista näkökulmaa. Esimerkiksi voidaan verrata Spacewarin käyttämää toteutusta uudempien pelien toteutuksiin ja tarkastella, minkälaisia muutoksia on tapahtunut käytetyissä pääsilmukoissa vuosikymmenten kuluessa. Toisaalta on myös mielenkiintoista tarkastella, onko silloinen laitteisto tuonut pääsilmukalle omia rajoitteitaan. Esimerkiksi jos laitteistolla ei ole sisäis-

tä kelloa, niin useimmat päivitystavat eivät ole mahdollisia toteuttaa vaan voidaan joutua käyttämään luvussa Vakiolla päivittyvä versio esiteltyä ratkaisua.

Spacewarin tutkiminen on mahdollista, koska verkosta onnistuin löytämään verkosta pelin tulkittavissa olevan lähdekoodin, jota on kommentoitu valmiiksi. Lisäksi Spacewarin toiminnasta on olemassa jo valmis analyysi, mikä helpottaa tutkimuksen tekoa huomattavasti.

6.4.1 Spacewarin pääsilmukan rakenne ja toiminta

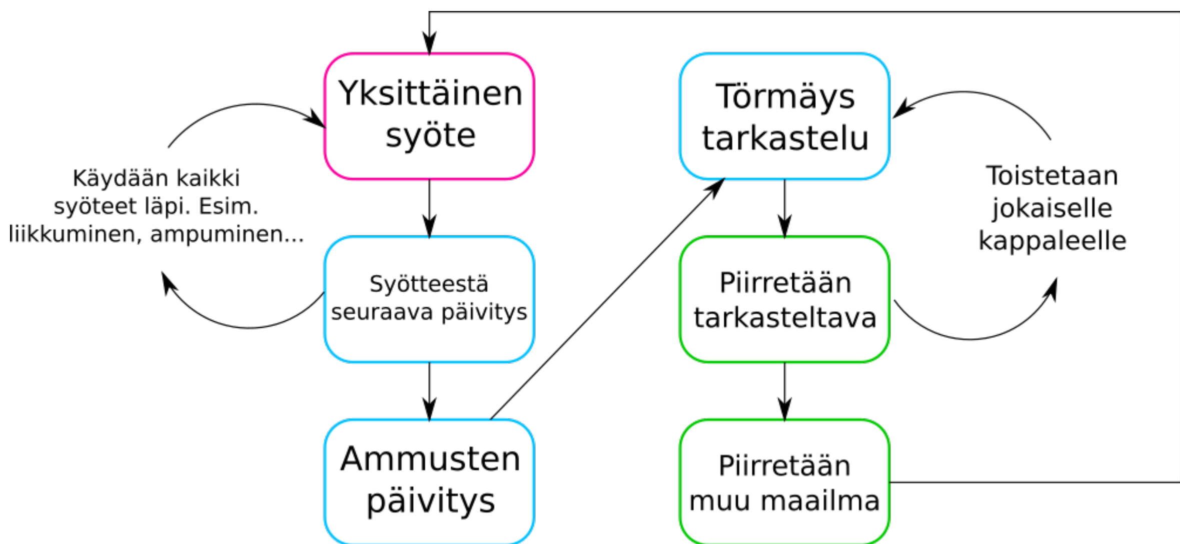
Spacewarin lähdekoodi on kirjoitettu PDP-1:n käyttämällä symbolisella konekielellä kuitenkin niin, että makrot ovat käytössä (Landsteiner 2015). Koska symbolinen konekieli ei ole kovinkaan helposti ymmärrettävää verrattuna moderneihin ohjelmointikieliin, niin sen tulkitseminen on haastavaa, vaikka lähdekoodi sisältääkin kommentteja. Vaikeus ilmenee esimerkiksi silmukoiden hankalana ymmärtämisenä, koska käytännössä silmukkarakenne on toteutettu käyttämällä komentoa, jolla hypätään suoraan haluttuun kohtaan koodia. Tämän takia silmukkarakenteiden hahmottaminen on vaikeaa, koska pitää tulkita isoja osia lähdekoodista sitä varten. Onneksi kuitenkin Spacewarin lähdekoodista on olemassa valmiiksi hyvä analyysi eli Landsteiner (2015), johon oma pääsilmukan tarkasteluni perustuu.

Kuitenkaan valmiista analyysistä huolimatta en pystynyt täysin varmistamaan, missä järjestyksessä tehtävät suoritettiin pääsilmukassa. Onnistuin kuitenkin havaitsemaan, mitä tehtäviä tehdään ja kuinka ne ovat eroteltu toisistaan. Tässä luvussa tuleekin huomioida, että se on vain oma lähdekoodin ja Landsteiner (2015) -analyysin pohjalta luomani käsitys pelin toiminnasta. On hyvinkin mahdollista, että olen ymmärtänyt joidenkin tapahtumien järjestyksen väärin, mutta tarkka järjestys ei ole kuitenkaan yleensä oleellista paitsi rinnakkaistetuissa ohjelmissa.

Silmukassa (katso kuva 12) uskoakseni kerätään melko alussa käyttäjältä kääntymiseen vaikuttavia syötteitä, joiden perusteella alusta käännetään. Tämän jälkeen kerätään eteenpäin liikkumiseen vaikuttavat syötteet, jonka jälkeen päivitetään alusten sijainteja pelimaailmassa huomioiden alusten nopeudet ja keskellä olevan tähden vetovoima (Landsteiner 2015). Liikkeen päivittämisen jälkeen siirrytään käsittelemään muita syötteitä ja niiden seurauksia kuten ampuminen ja ”hyperspace” liike (Landsteiner 2015). Myös ammusten tilannetta päivitetään

kuten niiden liikkuminen ja häviäminen, kun ne ovat kulkeneet tarpeeksi kauan.

Kun kaikkia kohteita on liikutettu, niin koodissa siirrytään syötteistä riippumattomiin päivityksiin, kuten törmäystarkasteluihin ja törmäyksistä seuraaviin räjähtelyihin. Törmäysten käsittely tehdään silmukassa, ja samassa silmukassa (en ole aivan varma, mutta luulisin) piirretään tarkasteltava kohde näyttölaitteelle. Saman silmukan käyttäminen on sikäli perusteltua, koska Landsteiner (2015) mukaan yhdessä piirtokomennossa menee hetki aikaa, minkä takia useita piirtokomentoja ei ole hyvä suorittaa peräkkäin.



Kuvio 12. Oma käsitykseni Spacewarin käyttämästä pääsilmukasta. Huomioitavaa on, että piirtämistä tehdään päivitysten välillä, kun törmäystarkastusten yhteydessä piirretään tarkasteltava kohde.

Nykyisissä laitteissa tämä ei ole ongelma, koska piirtokomennot voidaan aina laittaa puskuriin, jolloin komentoja voi olla useita odottamassa. Vaihtoehtoisesti Spacewarin toteutuksessa olisi voitu käyttää piirtämiskomentoa niin, että odotetaan piirtämisen valmistumista, mutta tällöin tulisi turhaa odottelua verrattuna siihen, että levitetään piirtämiskomennot laajemmalti (Landsteiner 2015). Törmäysten käsittelyn ja kohteiden piirtämisen jälkeen lopuksi piirretään pelissä taustalla oleva tähtitaivas ja keskellä pelimaailmaa alati muuttuva tähti, minkä jälkeen palataan silmukan alkuun (Landsteiner 2015).

Keskellä olevan tähden piirtämisessä on sikäli mielenkiintoista se, että tähteä piirrettäessä hyödynnetään satunnaislukua. Kyseessä ei ole kuitenkaan oikea satunnaisuus vaan pi-

kemminkin luku, jota muutetaan kaavalla joka kerta, kun kyseistä satunnaislukua kysytään (Landsteiner 2015). Tämän seurauksena satunnaisluku toistuu samana pitkänä lukusarjana, minkä seurauksena samaa siemenlukua käytettäessä saadaan aina samat luvut samassa järjestyksessä pelikerrasta riippumatta. Tämän ansiosta käytetty satunnaisuus ei haittaa pelin deterministisyyttä vaikkei deterministisyyttä hyödynnetäkään pelin toiminnassa.

Alkuperäisessä Spacewarissa on myös sikäli mielenkiintoinen ominaisuus, että alukset ja ammukset jättävät jälkeensä jälkikuvia, jotka katoavat hitaasti. Tämä ei ilmeisesti ole tarkoituksella tehty ominaisuus, koska Landsteiner (2015) mukaan tämä johtuu näyttölaitteesta. Kyseisen näyttölaiteen muodostamat piirrokset vain luonnostaan tummuvat hitaasti, minkä seurauksena pelissä näkyy jälkikuvia (Landsteiner 2015). Tämän havaitsin itse emulaattorissa, jossa ominaisuus on myös mallinnettu, jolloin ammuksista jää jälkeensä pieni tummuva vana, mikä näyttää oikein kivalta.

6.4.2 Spacewarin päivitystiheys

Spacewarin käyttämä pääsilmukka on sikäli mielenkiintoinen ja poikkeava nykyisistä pääsilmukoista, koska se ei käytä ollenkaan kellosta saatavaa aikaa apuna päivityksissä. Tämä johtuu luonnollisesti siitä, että PDP-1 ei tarjoa sisäistä kelloa, minkä takia päivitykset tulee tehdä vakioarvoilla (Landsteiner 2015). Käytetty ratkaisu muistuttaa siten luvussa Vakiolla päivittyvä versio ratkaisua ja Spacewarin toiminta on determinististä ajan käyttämisen puuttumisen ansiosta. PDP-1:n kellon puutteesta huolimatta, on Spacewarissa pyritty tasaiseen päivitysnopeuteen käyttämällä hyödyksi suoritettujen komentojen laskemista (Landsteiner 2015).

Käytännössä pelin alussa on arvioitu, kuinka monta komentoa silmukan suorituksessa menee ja silmukan aikana lisätään muuttuvissa kohdissa käytettyjen komentojen määrä talteen (Landsteiner 2015). Muuttuvia osia tulee esimerkiksi olemassa olevien ammusten määrästä, joka muuttuu pelin aikana. Pääsilmukan lopussa verrataan käytettyjen komentojen määrää arvioon, ja lasketaan, kuinka monta komentoa suoritus jäi vajaaksi arviosta (Landsteiner 2015). Vajaiden komentojen määrä korvataan tekemällä saman verran tyhjiä komentoja, ennen kuin siirrytään pääsilmukan seuraavalle kierokselle (Landsteiner 2015). Tämä ratkaisu

takaa sen, että jokaisella pääsilmukan kierroksella suoritetaan saman verran komentoja, ja siten aikaa kuluu lähes saman verran jokaisella kieroksella.

Arvion käyttäminen takaa kuitenkin vain yksittäisellä pelikerralla tasaisen päivitysnopeuden, mutta jos laitteistoa vaihdetaan tai muutetaan, niin pelin nopeus muuttuu myös. Tämä ei ole sinänsä suuri ongelma, koska Spacewar tehtiin alkujaan toimimaan vain PDP-1:llä, jolloin laitteisto muuttuminen radikaalisti ei ole juurikaan ongelma. Toisekseen PDP-1 koneita oli maailmassa ylipäätään melko vähän, jolloin peli tuskin tehtiin kovinkaan laaja käyttäjäkunta mielessä.

Nykyisissä tietokoneissa kuitenkaan samanlainen ratkaisu ei toimisi ollenkaan koneiden laajan vaihtelevuuden takia ja koska nykyiset ohjelmistot toimivat useimmiten käyttöjärjestelmän päällä. Tällöin pelin taustalla voi toimia useita eri ohjelmia, jotka voivat vaikuttaa pelattavan pelin suoritusnopeuteen. Lisäksi nykyisissä tietokoneissa on lähes poikkeuksetta sisäinen kello, jonka ansiosta voidaan aikaa käyttää hyödyksi päivityksiä tehdessä.

Toisaalta ei edes PDP-1 kanssa ollut täysin ongelmatonta tehdä päivitys ilman kellosta saatavaa aikaa (kelloa ei ollut, joten ei ollut vaihtoehtoja). PDP-1:een jossa spacewar toimi, tehtiin eräässä vaiheessa päivitys, joka mahdollisti suunnilleen neljänneksen nopeammat laskutoimitukset (Landsteiner 2015). Tämän seurauksena pelistä tuli nopeampi kuin aikaisemmin, mikä nosti pelin vaikeustasoa (Landsteiner 2015). Tämän takia kehittäjät päätyivät uudessa versiossa kasvattamaan arvioitua komentojen määrää, jolloin pääsilmukan lopussa käytettiin enemmän aikaa tyhjiin komentoihin siten hidastaen pelin vanhan nopeuden tasolle (Landsteiner 2015).

Vakioitu eli ajasta riippumaton päivitys näkyy pelin toiminnassa muun muassa siinä, että Landsteiner (2015) mukaan taustalla olevat tähdet oli asetettu siirtymään joka kuudestoista pääsilmukan kierros oikealle päin hieman. Lisäksi myös ilmeisesti ammusten elinikä on riippuvaista pääsilmukan kierrosten määrästä. Sen sijaan esimerkiksi alusten ja ammusten liikkuttaminen on huomattavasti yksinkertaisempaa, koska ne voidaan vapaasti toteuttaa jokaisella silmukan kierroksella. Ainoastaan nopeuksissa on haasteena se, että niitä on hyvin hankala saada kokeilematta oikein. Myös se, että jos peliä myöhemmin muokattaisiin raskaammaksi eli komentojen määrä kasvaisi, niin tällöin pelin kohteet vaikuttaisivat hidastu-

van ja sen korjaamiseksi pitäisi muuttaa nopeuksia uudestaan kokeilemalla.

6.4.3 Spacewarin yhteenveto

Spacewarin pääsilmukassa ja toiminnassa näkyy selkeästi sen aikakaudesta johtuvia rajoitteita. Erityisesti kuvan piirtämisessä on päädytty nykyään epätyypilliseen ratkaisuun eli annetaan piirtolaitteelle päivitysten välissä piirtämiskomentoja, jotta ei jouduttaisi odottamaan niiden valmistumisia ennen seuraavaa piirtämiskomentoa.

Hieman vastaavasti on myös toimittu syötteiden kanssa, kun on päädytty toteuttamaan syötteiden aiheuttamat päivitykset heti yksittäisten syötteiden keräämisten jälkeen. Esimerkiksi ensin kerätään kääntymiseen vaikuttavat syötteet, jonka pohjalta käännetään alusta, jonka jälkeen tehdään sama liikkumiselle, ampumiselle ja muille syötteistä riippuville toiminnoille. Vastaavasti varmaan toimitaan nykyäänkin, mutta arvelisin, että Spacewaria tehdessä oli ohjelmoinnin kannalta huomattavasti yksinkertaisempaa heti hyödyntää syöte kuin tallentaa se jonnekin laitteen muistiin.

Laitteiston muodostama rajoiteet näkyvät myös selkeästi laitteen sisäisen kellon puutteena, kun on jouduttu hyödyntämään suoritettujen komentojen laskemista, jotta saatiin tasainen päivitystiheys. Periaatteessa kyseisellä ratkaisulla on saavutettu lukujen Vakiolla päivittyvä versio ja Säännöllisesti päivittyvä versio ratkaisujen välimuoto, joka pitkälti korjaa monia vakiolla päivittämisestä seuraavia haasteita.

Kaiken kaikkiaan Spacewarin pääsilmukka on melko hajanainen eikä siitä voida kunnolla erotella teoriasta tuttuja päävaiheita (eli syöte, päivitys ja vaste) toisistaan, vaan ne limittyvät toistensa päälle. Toisaalta tehtävien suoritusjärjestyksillä ei ole juurikaan väliä muuta kuin rinnakkaistettaessa, joten tätä ei voida nähdä varsinaisena puutteena. Lisäksi eri vaiheiden limittyminen voidaan perustella, ja ylipäätään peli on onnistuneesti ja näppärästi toteutettu silloisista laitteiston rajoitteista huolimatta.

7 Oman toteutuksen tarkastelu

Tässä osiossa tarkastellaan itse tekemääni peliä, jonka toteutuksessa on pyritty hyödyntämään jo aiemmissa luvuissa käsiteltyjä asioita pääsilmukoihin liittyen. Käytännössä tämä tapahtui toteuttamalla tehdystä pelistä useita eri versioita, jotka käyttävät eri pääsilmukkamalleja. Eri versioita vertaamalla pystytään selkeämmin havaitsemaan kullekin mallille tyypillisiä ominaisuuksia varsinkin, kun varsinainen peli pysyy kaikissa eri versioissa samana.

Pelin eri versioiden toteutusten vertailun lisäksi pystyn höydyntämään tarkastelussa omia kokemuksiani eri versioiden tekemisestä. Esimerkiksi pystyn arvioimaan omaa näkökulmaani, kuinka vaikeita ja työläitä erilaisten pääsilmukoiden toteutukset ovat tehdä. Omat vaikutelmat ovat kuitenkin vain suuntaa antavia, koska joku toinen henkilö tai toista peliä toteutettaessa kokemukset voivat olla hyvinkin erilaisia.

Toisaalta jotkin pääsilmukkamallit voivat olla paremmin soveltuvia juuri tässä työssä toteutetulle pelille, joten ylipäätään eri mallien vertailu on osittain puutteellista. Esimerkiksi koska toteutettu peli on kohtuullisen pieni ja käyttää vain kaksiulotteista grafiikkaa, niin pelillä ei siten ole suuria vaatimuksia suorituskyvylle, minkä takia suorituskyvyn tarkastelu voi olla puutteellista.

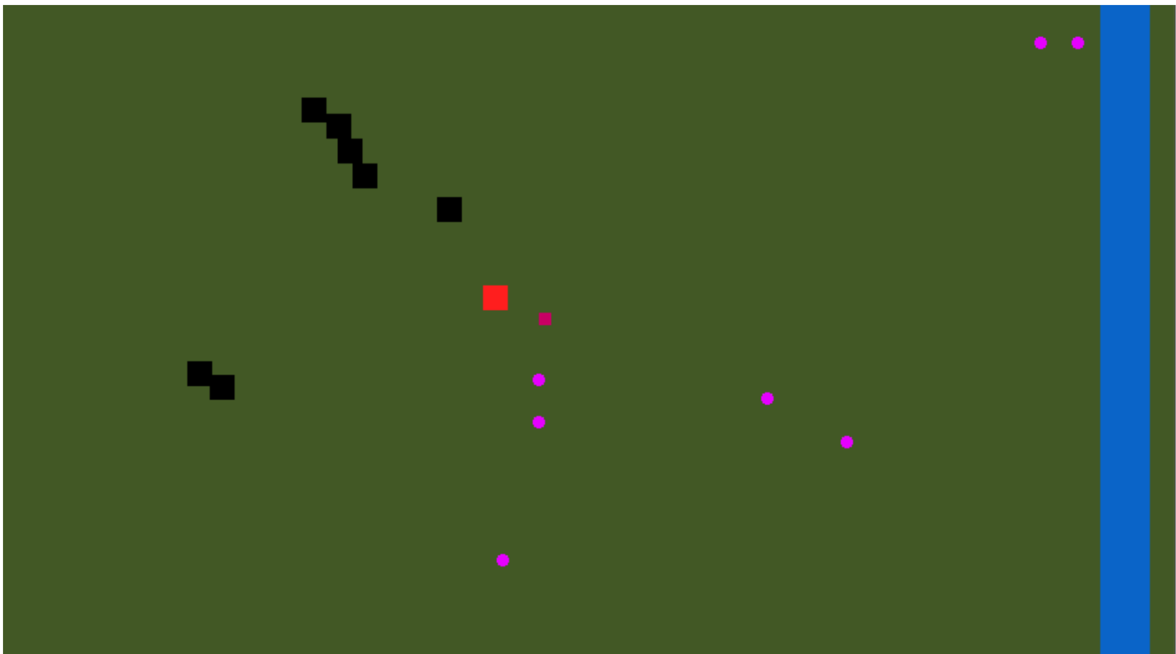
Tutkimusmenetelmän puutteista huolimatta saadaan ainakin jonkinlaista kuvaa eri pääsilmukkamallien toteutusten eroista yksittäisen pelin tapauksessa. Tutkimusmenetelmä on myös sikäli mielenkiintoinen, koska en ainakaan itse löytynyt juurikaan tutkimuksia, jossa olisi samalle pelille kokeiltu erilaisia pääsilmukoita. Löytämissäni tutkimuksissa on lisäksi myös melko vähän ylipäätään verrattu erilaisia pääsilmukoita. Pienen poikkeuksen muodostavat tosin tapaukset, joissa on rinnakkaistamaton pääsilmukka muokattu hyödyntämään rinnakkaisuutta, kuten esimerkiksi tutkimuksessa Abdelkhalek ja Bilas (2004).

Koska eri pääsilmukan toteutuksia ei siis ole verrattu juurikaan aiemmin, niin oma tutkimukseni tuo arvokasta lisää pääsilmukoiden tutkimukseen. Lisäksi olisi mielestäni myös hieman kiusallista kirjoittaa asiantuntevasti erilaisista pääsilmukkamalleista, jos en olisi itse koskaan kokeillut toteuttaa niitä. Kokeilemalla toteuttaa eri pääsilmukkamalleja saan itse käytännön kokemusta pääsilmukoista ja pystyn paljon vakuuttavammin ja luotettavammin kertomaan

niistä tämän tutkielman kautta. Tämän takia pidän omien pääsilmukoiden toteutusten tekemistä hyvin oleellisena osana tätä työtä.

7.1 Pelin kuvaus

Toteutettu peli (katso kuva 13) on kohtuullisen yksinkertainen kaksiulotteista grafiikkaa käyttävä peli, jossa pelaaja ohjaa pelihahmoa ja ampuu jahtaavia vihollisia. Pelialueena toimii suorakaiteen muotoinen tyhjä alue, jota rajaa seinät, joista ei voi (tai ei ainakaan pitäisi voida) mennä lävitse. Fysiikkan mallinnusta käytetään myös eri kappaleiden välillä eli esimerkiksi pelaajan ohjaama hahmo törmää vihollisiin, kun taas ammukset tuhoutuvat aina törmätessään.



Kuvio 13. Kuva toteutetusta pelistä.

Pelaaja voi ohjata pelihahmon liikettä pysty ja vaakasuunnassa ja lisäksi myös viistoon. Ammuksia taas voi ampua vain pelihahmon katsomaan suuntaan eli suuntaan, johon pelihahmo on viimeksi liikkunut. Peli ei sisällä kirjoitushetkellä muita kontrolleja, ja peli ei myöskään tue mitään valikko-ominaisuuksia (esim. pelin asettaminen tauolle, eng. pause). Yksinkertaisesti peli myös alkaa suoraan ohjelman käynnistyttyä.

Vihollisia luodaan peliin säännöllisin väliajoin ja on mahdollista, että useampi vihollinen on samaan aikaan olemassa. Luodut viholliset käyttävät alkeellista tekoälyä, joka ohjaa ne liikkumaan suoraan kohti pelihahmoa. Vihollisten törmäyksellä pelihahmoon ei tosin ole mitään erikoisvaikutusta (esim. pelihahmo ei kuole, menetä elämää tai muuta vastaavaa), mutta ne voivat estää pelaajaa liikkumasta, jos useampi vihollinen ympäröi pelihahmon johtuen fysiikan mallinnuksesta.

Viholliset tuhoutuvat ammusten osuessa niihin, jolloin vihollinen poistetaan pelistä tuohoutumisanimaation käynnistyessä. Kyseinen animaatiossa vihollisen kokoinen punainen neliö kutistuu olemattomaksi. Animoidulla neliöllä ei ole kuitenkaan fysiikkaa ja on pelkästään kosmeettinen ominaisuus. Vastaavasti myös ammusten tuhoutumiselle on olemassa oma animaatio, joka näyttää siltä, että ammus vaihtaa väriä ja kutistuu olemattomaksi jatkaessaan liikettään törmätyn kohteen läpi.

Esitelty peli on melko yksinkertainen toteuttaa, mutta se tarjoaa samalla useita erilaisia päivitysvaiheita, kuten tekoälyn päivitys, fysiikkamoottorin päivitys ja animaatioiden päivitys, jotka ovat peleissä hyvin tyypillisiä ominaisuuksia. Useiden erilaisten päivitysvaiheiden avulla voidaan laajemmin kokeilla erilaisten pääsilmukoiden toteutusten vaikutuksia pelin toimintaan. Erityisesti fysiikkamoottorin sisältyminen tutkimuksessa käytettävään peliin on tärkää, koska esimerkiksi Phaserin ja Unityn analyysissä käy selkeästi ilmi, että se vaatii erikoiskäsittelyä. Monipuolisten ominaisuuksien ansiosta esitelty peli soveltuukin hyvin käytettäväksi tässä tutkimuksessa ja samalla se mielestäni edustaa onnistuneesti tyypillistä kaksiulotteista peliä, jolloin tulokset voidaan helpommin yleistää koskemaan muitakin samantyyppisiä pelejä.

7.2 Pelin toteuttaminen

Pelin lähdekoodi löytyy osoitteesta `https://github.com/1kasu/gpeli`. Koska lähdekoodi voi muuttua tämän työn julkaisemisen jälkeen, niin tarkastelemani lähdekoodiversio löytyy ”graduversio” haarasta (suora osoite `https://github.com/1kasu/gpeli/tree/graduversio`). Aikomuksena ei ole muokata tätä haaraa kuin korkeintaan poistamalla tyhjiä rivejä, lisäämällä kommentteja tai parantelemalla dokumentaatiota, joten tässä

työssä esiteltyjä tuloksia voi verrata välittämättä uusista muutoksista.

Tässä luvussa käsittelen pelin tekemisessä käyttämiäni työkaluja ja niiden valintaa. Lisäksi käsittelen lyhyesti joitakin kokonaisuuksia pelin toiminnasta, jotka ovat oleellisia pääsilmukan kannalta. Toteutusten kuvaus sisältää myös pääsilmukan toimintaa liittymätöntä tietoa, joka kuitenkin tarjoaa tietoa niille, jotka ovat toteutetun pelin tarkemmasta toiminnasta kiinnostuneita.

7.2.1 Työkalujen valinta

Pelin ohjelmointia varten minun piti valita tarvittavat työkalut ja ohjelmointiympäristö. Esimerkiksi kaksiulotteisen grafiikan piirtämiseen tarvitsin oman ohjelmointikirjaston, jolla voidaan luoda ikkuna, johon voi piirtää yksinkertaisia muotoja. Vastaavasti tarvitsin myös kirjaston, jolla lukea käyttäjän syötteitä eli käytännössä näppäimistön painalluksia kätevästi.

Peligrafiikan piirtämiseen löytyi työkaluja, mutta useimmat niistä olivat osana valmista pelimoottoria, joissa erilaisten itse tehtyjen pääsilmukoiden toteuttaminen ei onnistu kovinkaan luontevasti. Yhtenä vaihtoehtona olisi ollut käyttää jotain matalan tason grafiikkakirjastoa suoraan kuten esimerkiksi OpenGL:ää (Open Graphics Library), mutta silloin olisi pitänyt opetella huomattavasti asioita jo pelkän neliön piirtämistä varten.

OpenGL:n sijaan päädyinkin käyttämään SDL:ää (Simple DirectMedia Layer), joka on erityisesti pelien tekemiseen suunniteltu ohjelmakirjasto. Sen avulla voidaan muun muassa piirtää yksinkertaista grafiikkaa ja hallita käyttäjältä tulevia syötteitä. SDL:llä ei myöskään tarjoa (tai ei ainakaan pakota käyttämään) valmista pääsilmukkaa, jolloin pystyy tarkasti hallitsemaan itse toteutetun pääsilmukan rakennetta. SDL:ää pystyi lisäksi käyttämään Rust ja C# ohjelmointikielillä olemassa olevien kolmannen osapuolien kirjastojen avulla.

Mainittuja kieliä harkitsin käytettäväksi, koska molemmat olivat itselleni tuttuja ja arvelin niiden soveltuvan suunnitellun pelin tekemiseen. Erityisesti C#:lla arvelin pelin toteuttamisen olevan kohtuu helppoa kunhan vain saan projektin alkuun. Rustilla taas arvelin työskentelyn olevan hieman haastavampaa, koska se sisältää useita muista ohjelmointikielestä poikkeavia ominaisuuksia. Toisaalta arvelin, että teoriassa Rustilla voisi olla rinnakkaistus vähemmän virhealtista kielen tarjoamien tarkistusten ansiosta.

Lopulta päädyinkin valitsemaan Rustin, koska sillä sovellusprojektin pystyttäminen on oman kokemukseni mukaan yksinkertaisempaa paketinhallintatyökalu (eng. package manager) Cargon ansiosta. Lisäksi olin jo aiemmin toteuttanut pienen projektin Rustilla ja SDL:llä onnistuneesti, mikä lisäsi luottamustani valintaan. Toisaalta halusin myös haastaa itseäni käyttämällä itselleni hieman vähemmän tuttua ohjelmointikieltä (eli Rustia).

Ohjelmointiympäristönä taas päädyin käyttämään Eclipselle tehtyä Corrosion-lisäosaa, joka on suunniteltu erityisesti Rust-projekteja varten. Corrosion tarjosi useita Rust-projektissa hyödyllisiä ominaisuuksia yhdessä kokonaisuudessa, mitä pidin yksinkertaisempana ratkaisuna, kuin ruveta lisämään johonkin editoriin käsin useita erillisiä lisäosia. Versionhallintaa varten taas valitsin GitHubin, koska halusin toteutettavan pelin olevan helposti näytettävissä työnhaussa.

7.2.2 Syötteiden käsittelyn toteutus

Syötteiden kerääminen ja päivittäminen tapahtuu toteutetussa pelissä keskitetysti luokan *Syotteet* kautta. Tarkkaan ottaen kyseessä on *struct*, joka vastaa Rustissa luokkaa. Luokan edustajalle pitää antaa näppäimet, joiden tilaa se kuuntelee. Yksittäisen näppäimen tilaa voidaan kysyä luokan edustajalta, ja mahdollisia tiloja on näppäimen pohjassa olon lisäksi tieto, onko näppäin juuri vapautettu pohjasta tai juuri painettu pohjaan.

Tarkemmalla erittelyllä on haluttu mahdollistaa kehittäjälle helppo tapa luoda tapahtumia, jotka liittyvät vain näppäimen painamiseen pelkän pohjassa olon sijaan. Esimerkiksi ampuminen tehdään vain kerran, kun näppäin painetaan pohjaan. Jos näppäintä pidetään pohjassa, mitään ei tapahdu. Tämä on myös syynä siihen, että näppäimet pitää ensin rekisteröidä kuunneltaviksi, koska jos näppäintä ei olla aiemmin kuunneltu, niin ei tiedetä, kuinka pitkään se on ollut pohjassa.

Näppäinten tiloja ei lueta kysyttäessä uudestaan vaan näppäinten tilat pitää kerralla päivittää kutsumalla paivita_nappainten_tilat-funktiota. Tällä on pyritty käytökseen, jossa näppäimen tila ei voi muuttua kesken päivityksen. Lisäksi ratkaisu on välttämätön, koska muuten ei voitaisi kovinkaan helposti tarkkailla, onko näppäin painettu pohjaan edellisen päivityksen jälkeen.

7.2.3 Fysiikkamoottorin toteutus

Peliä varten tein sille oman yksinkertaisen fysiikkamoottorin, joka osaa laskea kappaleille uudet sijainnit niiden nopeuksien avulla. Käytännössä kappaleen uusi sijainti tulee suoraan kaavalla:

$$uusiSijainti = vanhaSijainti + kappaleenNopeus * paivitysaika$$

Kuitenkin jos uuden sijainnin kohdalla on toinen kappale eli tapahtuu törmäys, niin kappaleen siirtyminen perutaan ja törmäystapahtuma merkitään talteen, jotta se voidaan käsitellä myöhemmin.

Tapahtumajärjestys on tarkkaan ottaen niin, että ensin lasketaan kaikille kappaleille uusi sijainti, mutta vanha sijainti säilytetään muistissa. Tämän jälkeen lasketaan kaikilla uusilla sijainneilla törmäykset ja törmäystapahtumat asetetaan muistiin. Törmäysten tarkastelun jälkeen perutaan kaikkien törmänneiden kappaleiden liike eli asetaan sijainniksi vanha sijainti.

Käytetyllä ratkaisulla on pyritty siihen, että kappaleiden järjestyksellä listassa ei olisi ollenkaan väliä. Toiseksi, jos siirtyminen peruttaisiin kesken törmäystarkastelun, niin tulisi tilanteita, joissa kappale A törmäisi kappaleeseen B, mutta kappale B ei törmäisi kappaleeseen A, mikä olisi ristiriitaista. Toki olisi mahdollista käsitellä törmäyksiä niin, että törmäys tapahtuma tallentuisi muodossa A ja B törmäsivät sen sijaan, että on vain muodossa A törmäsi B:hen.

Lisäksi jos liike peruttaisiin kesken törmäystarkastelun, niin olisi mahdollista, että jotkin kappaleet törmäisivät jo törmänneeseen kappaleeseen, mutta alkuperäisessä sijainnissa. Tällainen käytös olisi kuitenkin hyvin vaihtelevaa, ja vain osa kappaleista tarkistaisi törmäyksen vanhassa sijainnissa, kun taas aiemmat kappaleet olisivat tarkastaneet törmäyksen vain uudessa sijainnissa. Esimerkiksi kappale A liikkuu kappaleen B alkuperäiselle sijainnille. Kappale B törmää johonkin toiseen kappaleeseen ja joutuu palaamaan alkuperäiselle sijainnilleen. Tämän jälkeen kappale C vertaa itseänsä kappaleeseen B ja havaitsee törmäyksen kappaleen B alkuperäisessä paikassa. Tällöin kappale A onnistui siirtymään kappaleen B alkuperäiselle paikalle onnistuneesti, mutta kappale C ei.

Lisäksi myöhemmin tulevat kappaleet voisivat siirtyä kappaleen B uudelle sijainnille, kos-

ka kappaleen B liikkuminen peruutettiin. Tämän perusteella käy ilmi, että kappaleiden lopulliset sijainnit ovat riippuvaisia, missä järjestyksessä ne ovat listassa, mitä en itse pitänyt toivottavana käytöksenä.

Kuitenkin oma toteutukseni ei ole täydellinen ja kappaleet voivat yhä joutua toistensa sisälle. Näin käy esimerkiksi jos kappaleen B liike perutaan, mutta kappale A on siirtynyt kappaleen B alkuperäiselle paikalle. Tämän korjaamiseksi törmäystarkastelu pitäisi tehdä uudestaan kaikille kappaleille, jotta havaittaisiin törmäykset niissä tilanteissa, joissa törmätään jonkin kappaleen alkuperäisessä paikassa. Tällöin kuitenkin sama tilanne yhä toistuisi ja käytännössä jouduttaisiin toteuttamaan törmäystarkastelu kaikille kappaleille niin kauan kuin havaitaan yksikin törmäys, mikä voisi olla laskennallisesti raskasta ilman optimointia.

Itse päätinkin tyytyä käymään törmäystarkastelun läpi vain kerran. Päätöstä perustelin sillä, että pelissäni on hyvin vähän liikkuvia kohteita ja esimerkiksi ammukset ja viholliset tuhoutuvat törmätessään toisiinsa, jolloin päällekkäisyys ei ole ongelma. Myös vaikka kappaleet menisivät toistensa päälle, niin se olisi melko pientä eikä edes välttämättä havaittavissa.

Törmäystarkastelu taas toimii vain vaakatasossa olevien suorakulmioiden ja ympyräiden välillä. Pidin tätä omaa peliäni varten riittävänä ratkaisuna. Ympyröiden välillä törmäys tarkistetaan helposti vertaamalla säteiden summaa keskipisteiden etäisyyteen. Suorakulmioiden välillä taas lähdin ajatuksesta, että törmäystä ei tapahdu, jos kappaleen A alareuna on korkeammalla kuin kappaleen B yläreuna ja niin edelleen. Lähteen Mozilla (2019) perusteella ratkaisuni vaikuttaisi olevan yleisemmin tunnettu ratkaisu nimeltään Axis-Aligned Bounding Box (lyhyesti AABB). Ratkaisun rajoite tosin on, että molempien kappaleiden pitää olla vaakatasossa olevia suorakulmoita (Mozilla 2019).

Suorakulmioiden ja ympyröiden välille taas en käyttänyt mitään yleisesti tunnettua ratkaisua vaan heuristisesti loin oman, joka ainakin vaikuttaisi toimivan, mutta ei ole välttämättä tehokas. Vaihtoehtoisesti olisin voinut käyttää valmiiksi tunnettua ratkaisua. Esimerkiksi Separating Axis Theorem (lyhyesti SAT) voi havaita törmäyksen monikulmioiden välillä, mutta myös monikulmion ja ympyrän välillä (Sevenson 2009). SAT:n avulla olisin voinut myös paremmin tarkkailla omien suorakulmioideni törmäyksiä, koska SAT toimii paljon yleisemmin. Esimerkiksi suorakulmioideni ei olisi tarvinnut olla vaakatasossa tai suorakulmoiden

sijaan ne olisivat voineet olla vaikka kolmioita.

7.2.4 Animaatioiden toteutus

Animaatiot on pyritty toteuttamaan ajatuksella, että animaatiosta voidaan pyytää kuvaa miltä tahansa ajanhetkeltä. Esimerkiksi voitaisiin pyytää animaatiota muodostamaan kuva 2,3 sekuntia animaation alkamisesta. Tätä varten tehdyt animaatiot hyödyntävät lineaarista interpolaatiota, jolloin kehittäjän tarvitsee vain antaa animaation alku- ja lopputila, joiden väliä interpoloidaan. Esimerkiksi alkutilaksi voidaan asettaa neliön koko alussa ja lopuksi koko lopussa. Tällöin interpolaation avulla neliö näyttää muuttuvan animaation edetessä ja esimerkiksi vihollisten tuhoutumisanimaatiossa neliö kutistuu olemattomaksi. Lisäksi käytetty lineaarinen interpolaatio on toteutettu yleiskäyttöiseksi, ja myös luvussa Interpoloivan pääsilmukan kuvaus käytetty ratkaisu hyödyntää samaa funktiota.

Animaatioiden päivitys toimii muista päivityksistä hieman poikkeavasti, koska animaatiot tarvitsevat tiedon, kuinka paljon animaation alusta on kulunut aikaa. Tätä varten toteutuksessa animaatioilla on alkamisaika, joka on suhteessa aikaan jolloin peli käynnistyi. Animaatioiden päivitykselle (ja myös muille päivityksille) annetaan päivitysajan lisäksi tieto, kuinka paljon aikaa on kulunut pelin käynnistymisestä. Vertaamalla tätä aikaa animaation alkamisaikaan, animaatio pystyy laskemaan animaation sen hetkisen tilan. Vaihtoehtoisesti animaatio voisi muistaa, aiempien päivitysten päivitysaikojen summan ja käyttää sitä apuna, mutta tällöin voisi tulla ongelmia, jos animaatiota ei jostain syystä haluttaisikaan päivittää jokaisella silmukan kierroksella.

Lisäksi animaatiojärjestelmään kuuluu myös animaatioiden elinikä, koska käytetyt animaatiot ovat vain muutaman sekunnin mittaisia korkeintaan. Elinikä toimii käytännössä vain niin, että aina animaatioita kutsuttaessa ensin verrataan pelin alusta kulunutta aikaa animaation kuolinaikaan. Tarvittaessa animaatio poistetaan, jos se todetaan vanhentuneeksi. Vaihtoehtoinen ratkaisu olisi ollut toteuttaa animaatioiden vanheneminen niin, että animaatio sanoisi itse vanhentuneensa saavutettuaan lopputilansa.

Animaatiojärjestelmän sisällyttäminen osaksi peliä on tutkimuksen kannalta tärkeää, koska sen toiminnalla ei ole juurikaan vaikutusta pelin loogiseen toimintaan. Tämän ansiosta

teorian mukaan erityisesti luvussa Säännöllinen ja epäsäännöllinen päivitys erikseen esitelty pääsilmukka hyötyy tästä ominaisuudesta. Toisaalta animaatiot ovat muutenkin yleisiä peleissä ja niiden käyttäminen tutkimuksessa on siten toivottavaa.

7.2.5 Pelimaailman toteutus

Pelimaailmaa kuvastavan luokan on tarkoitus sisältää kaikki pelimaailmassa olevat kappaleet. Lisäksi kappaleilla voi olla erilaisia ominaisuuksia kuten tekoäly, fysiikka, piirrettävyys tai muuta vastaavaa. Tätä varten kappaleet sisältävän listan lisäksi maailma pitää yllä listaa kappaleilla olevista ominaisuuksista. Esimerkiksi on olemassa lista kaikista tekoälyistä, joilla kullakin tekoälyllä on viite kappaleeseen, jota tekoäly koskee. Tällä ratkaisulla maailmalta voidaan helposti pyytää kaikki tekoälyt yhdessä listassa, kun esimerkiksi halutaan päivittää tekoälyjen tilaa.

Kun kullekin kappaleen mahdolliselle ominaisuudelle on oma listansa, niin ei tarvitse kysyä kaikilta kappaleilta, onko niillä jokin tietty ominaisuus. Esimerkiksi tekoäly vihollisia on vain muutama eikä olisi tehokasta käydä kaikkia kappaleita läpi niiden keräämiseksi. Menetetty tehokkuus olisi sinänsä merkittävä haitta, koska kappaleiden määrä voi kasvaa huomattavasti, ja etsiminen tulisi tehdä jokaisella silmukan kierroksella.

Tästä huolimatta omassa toteutuksessani on puutteena, että jos esimerkiksi tekoäly haluaa muuttaa kappaleen fysiikassa nopeutta, niin tällöin joudutaan käymään fysiiikat sisältävä lista läpi, jotta löydetään samaan kappaleeseen liitetty fysiikka. Tämä johtuu siitä, että vaikka tekoälyt ja muut ominaisuudet sisältävät viitteen kappaleeseen, niin kappale ei sisällä viitettä näihin ominaisuuksiin, jolloin kappale ei voi tehokkaasti antaa sille liitettyjä ominaisuuksia. Ominaisuuksien antaminen onnistuu tarvittaessa, mutta haluttu ominaisuus jouduttaan etsimään listasta, mikä ei ole tehokasta.

Yhtenä hieman erityisenä erikoisominaisuutena kappaleilla on, että kaikilla kappaleilla on muisti vanhasta sijainnista. Tätä ominaisuutta tarvitaan interpoloivan pääsilmukan toteutuksessa (katso luku Interpoloivan pääsilmukan kuvaus). Ominaisuutta ei kuitenkaan hyödynnetä muissa silmukoissa, ja siten se on ainoita pääsilmukkakohtaisia ominaisuuksia pelimaailmalla. Muut pelimaailman ominaisuudet toimivat samalla tavalla käytettävästä pääsilmu-

kasta riippumatta.

Ehkä merkittävimpänä ongelmana toteutetulle pelimaailmalle on, että se ei ole helposti rinnakkaistettavissa. Käytetystä toteutustavasta johtuen, jos yksi päivitys muokkaa pelimaailmaa, niin muut päivitykset eivät voi muokata pelimaailmaa samanaikaisesti. Tämä ei onnistu vaikka päivitykset olisivat todellisuudessa toisistaan riippumattomia. Tämä johtuu Rustin omasta ominaisuudesta, koska Rust on hyvin tarkka sen suhteen etteivät säikeet voi muokata samaa resurssia samanaikaisesti. Koska piilotin pelimaailman toteutusta pelimaailman omiin toteutuksiin, niin Rustin näkökulmasta pelimaailman muokkaaminen sen omilla funktiolla lukitsee koko pelimaailman pois muilta säikeiltä.

Ajatuksena tämä on oikein järkevää, koska esimerkiksi periaatteessa tekoälyn ja pelihahmon päivitys voivat koskea samaa kappaletta, jolloin eri säikeet pyrkisivät muokkaamaan samaa kappaletta. Kuitenkin kehittäjänä itse tiedän, että pelihahmolla ei ole tekoälyä, niin voisin toteuttaa rinnakkaistuksen itse näille. Tämä ei ole kuitenkaan nykyisellä toteutuksella käytännöllisesti toteutettavissa.

Ongelman korjaamiseksi tulisi muokata pelimaailmasta avoimempi niin, että kehittäjä voi paremmin itse määrätä, mitä kaikkea resursseja eri funktiot käyttävät. Esimerkiksi animaatioita varten ei tarvitse lukita fysiikkaa, jolloin fysiikka ja animaatiot voivat toimia toisistaan riippumatta eri säikeissä. Ikävä kyllä rinnakkaistuksen mahdollistaminen vaatisi huomattavia muutoksia pelimaailman toimintaan ja siihen ei tutkimuksen puitteissa ole aikaa.

7.2.6 Pääsilmukoiden yleinen toteutus

Pelin käyttämät pääsilmukat on toteutettu niin, että silmukan voi vaihtaa helposti ilman, että pelin logiikkaan tarvitsee tehdä muutoksia. Koska kaikki silmukat toteuttavat saman rajapinnan (Rustissa tarkkaan ottaen *trait* vastaa rajapintaa), on silmukoiden käyttämisessä vain erona, kuinka kukin silmukka alustetaan. Alustuksessa käytettäviä parametrejä on muun muassa silmukan käyttämä kuvan piirtäjä, päivitystiheys ja käytettävät päivitykset.

Päivitysten asettaminen on oleellista, koska ne sisältävät pelin logiikan. Päivitykset on myös eriytetty pieniksi kokonaisuuksiksi, jotta luvussa Erilliset päivitykset -pääsilmukan kuvaus esitellyssä ratkaisussa voisi jakaa yksittäiset päivitykset säännöllisiin ja epäsäännöllisiin päi-

vityksiin. Lisäksi erottelu pieniin ryhmiin mahdollistaa helpomman rinnakkaistuksen.

Luonnollisesti myös kaikki päivitykset toteuttavat saman päivityksille yhteisen rajapinnan, minkä ansiosta uusien päivitysten tekeminen ja käyttöön ottaminen on suhteellisen helppoa. Käytännössä päivityksen rajapinta vain vaatii, että päivitys alustaa pelimaailmaa tarvitsevansa verran ja sillä on päivitysfunktio, joka huolehtii varsinaisesta päivityksestä. Käytetty ratkaisu muistuttaa hyvin paljon Unityn käyttämää, jossa on myös useita päivityksiä, joiden yhteydessä on alustus (katso luku Unityn toiminnasta yleisesti). Toisaalta myös Phaserissä on päivitys ja alustus yhteydessä toisiinsa, mikä voidaan havaita listauksesta 9.1.

7.3 Tutkittavien pääsilmukoiden kuvaukset

Tehdylle pelille onnistuin toteuttamaan neljä erilaista pääsilmukkaa, jotka hyödyntävät tässä työssä esiteltyjä malleja. Toteutetut pääsilmukat ovat kaikki yksisäikeisiä johtuen siitä, että aika ei riittänyt monisäikeisten pääsilmukoiden toteuttamiseen. Lisäksi monisäikeisen pääsilmukan toteuttaminen olisi vaatinut huomattavia muutoksia pelin muuhun toimintaan ja siksi ne jätettiin tästä työstä pois. Tässä luvussa pyritään kuvaamaan toteutettujen pääsilmukoiden toimintaa ja perustellaan niiden toteutuksissa tehtyjä valintoja.

Ensimmäiseksi toteuttavaksi pääsilmukaksi valittiin ajan mukaan päivittyvä pääsilmukka (käsitelty luvussa Ajan mukaan päivittyvä versio). Kyseinen pääsilmukka on hyvin yleinen ja sen toteuttaminen toimii hyvänä pohjana muille pääsilmukoille. Esimerkiksi Quake vaikuttaisi käyttävän kyseiseen pääsilmukkaan pohjautuvaa silmukkaa omassa toiminnassaan. Tosin Quake vaikuttaisi hyödyntävän lisäksi myös luvun Toistuvasti päivitys -malli ratkaisua.

Vastaavasti toteutettavaksi pääsilmukaksi valittiin myös säännöllisesti päivittyvä silmukka (käsitelty luvussa Säännöllisesti päivittyvä versio), joka on toinen hyvin yleinen pääsilmukkamalli. Monet teoriassa käsitellyistä silmukoista pohjautuvat näihin kahteen pääsilmukkaan, ja siten molemmat silmukat ovat tutkimuksen kannalta tärkeitä. Lisäksi mainitut silmukat ovat hyvin samanlaisia ja eroa on vain päivitystavassa, minkä ansiosta molempien toteuttaminen ei juurikaan lisää työmäärää.

Monimutkaisemmaksi toteutettavaksi pääsilmukaksi valittiin luvussa Säännöllinen ja epäsäännöllinen päivitys erikseen käsitelty pääsilmukka, joka yhdistää säännöllisesti päivittyvän ja ajan mukaan päivittyvän pääsilmukan ominaisuuksia. Päätöksen taustalla on se, että silmukka vaikuttaa teorian pohjalta mielestäni hyvin lupaavalta, mutta se ei uskoakseni ole edes kovinkaan työläs toteuttaa. Lisäksi Phaser ja Unity molemmat vaikuttivat käyttävän samankaltaista pääsilmukkaa, mikä osoittaa silmukan olevan laajasti käyttökelpoinen. Erityisesti voidaan todeta Phaseriä ja Unityä tarkasteltua, että fysiikkamoottorille koetaan tärkeäksi, että sen päivittäminen on säännöllistä.

Neljänneksi toteuttavaksi silmukaksi valittiin luvussa Interpoloiva malli esitelty pääsilmukka. Kyseinen silmukka vaikutti todella kehittyneeltä ja interpolaation käyttäminen tuo mielestäni mielenkiintoisen lisän silmukan toimintaan. Erityisesti itseäni kiinnosti ajatus siitä, että kuvan päivitys tapahtuu useammin, kuin mitä varsinainen pelin tilan päivittäminen, mikä muissa pääsilmukoissa ei useinkaan ole mahdollista.

7.3.1 Ajan mukaan päivittyvän pääsilmukan kuvaus

Toteutettu ajan mukaan päivittyvä silmukka (katso lähdekoodi koodilistauksesta 9.10) pohjautuu vahvasti luvussa Ajan mukaan päivittyvä versio esiteltyyn versioon pääsilmukan perusmallista. Pienenä erona tosin on, että on erikseen eroteltu pelin sulkeminen ikkunan kulmassa olevaa rastia hiirellä klikkaamalla tai Esc-näppäintä painamalla. Tämä ratkaisu on ainakin kehitysvaiheessa hyödyllinen, koska pelin saa suljettua vaikkei varsinaista syötteiden keräämistä tai päivitystä ollut vielä toteutettu. Lisäksi peliä suljettaessa ei muutenkaan haluta päivittää pelin tilaa, joten ratkaisulla ei ole vaikutusta pääsilmukan toimintaan.

Sulkemisen tarkastamisen jälkeen päivitetään tarkkailtavien syötteiden tilaa eli käytännössä, mitkä näppäimet ovat pohjassa ja mitkä eivät. Tämän jälkeen suoritetaan päivitykset pelimaailmalle käyttäen syötteitä ja päivitysaikaa. Päivitysaikana toimii tieto edellisestä päivityksestä kulunut aika. Lisäksi päivitykselle annetaan myös pelin käynnistymisestä kulunut aika, jota hyödynnetään muun muassa animaatioissa. Lopuksi piirretään pelimaailman päivittynyt tila näytölle.

Huomioitavaa on myös, että peli mahdollisesti päivittyy huomattavasti useammin kuin mitä

näyttö pystyy piirtämään kuvaa. Tämän takia harkittavissa olisi, että jokaisen silmukan kierroksen jälkeen keskeytettäisiin säikeen toiminta pieneksi hetkeksi, jolloin peli ei olisi koko aikaa aktiivisena. Tämä saattaisi vähentää pelin aiheuttamaa kuormitusta suorittimelle ja mobiililaitteissa saattaisi siten vähentää myös akun kulutusta. Tätä ratkaisua ei ole kuitenkaan toteutettu omassa toteutuksessani, koska en pitänyt sitä tarpeellisena tutkimuksen kannalta.

Lisäksi omassa toteutuksessani esiintyy muutenkin mystinen ilmiö, jossa muutaman sekunnin välein päivitysväli kasvaa moninkertaiseksi, enkä siksi halunnut ylimääräisiä häiriötekijöitä. Esiintyvä häiriö on huomattava, koska se aiheuttaa näkyvän kuvan jähmettymisen, johtuen pitkästä päivitysvälistä. Se ei kuitenkaan haittaa tutkimuksen tekemistä, mutta julkaistavassa pelissä se olisi huomattava ongelma. Itselläni ei ole tarkkaa tietoa eikä aikaa selvittää, mistä ilmiö johtuu. Kuitenkin arvelisin, että sillä saattaisi olla jotakin tekemistä SDL:n ja erityisesti kuvan piirtämisen kanssa.

7.3.2 Säännöllisesti päivittyvän pääsilmukan kuvaus

Säännöllisesti päivittyvä silmukka (katso lähdekoodi listauksesta 9.11) pohjautuu vastaavasti lukuun Säännöllisesti päivittyvä versio. Toteutukseltaan silmukka on hyvin samanlainen kuin ajan mukaan päivittyvä silmukka, ja erona on lähinnä vain kuinka usein päivitys tehdään. Päivitystiheys on mahdollista antaa silmukkaa alustaessa, jolloin se annetaan muodossa päivitysten määrä sekunnissa (esim. 60 päivitystä sekunnissa).

Päivitys on toteutettu niin, että ennen päivitystä tarkistetaan, onko edellisestä päivityksestä kulunut tarpeeksi aikaa. Jos aikaa on kulunut tarpeeksi, niin siirrytään toteuttamaan päivitys silmukalle määrätyllä päivitysajalla. Esimerkiksi jos silmukan päivitystiheys on 60 kertaa sekunnissa, niin käytettävä päivitysaika on aina noin 0,016666 (= 1/60) sekuntia.

Kuitenkin on mahdollista, että päivityksestä voidaan niin sanotusti myöhästyä, jos edellisestä päivityksestä onkin kulunut enemmän aikaa kuin, mitä asetettu päivitysaika on. Myöhästymisestä huolimatta seuraava päivitys asetetaan tapahtumaan vasta annetun päivitysajan kuluttua, minkä seurauksena pelin sisäinen aika voi jäädä todellista aikaa jälkeen. Esimerkiksi vaikka annettu päivitystiheys on 60 kertaa sekunnissa, niin se ei tarkoita, että 60 päivityksen jälkeen on kulunut tasan yksi sekunti. Koska jokin päivitys on voinut tapahtua myöhässä,

niin sen seurauksena kaikki muutkin päivitykset myöhästyvät oletetusta ajasta, jolloin viivettä syntyy ennen pitkää.

Käytännössä päivityksestä myöhästymisen pitäisi näkyä pelaajalle pelin hetkellisenä pysähtymisenä, mikä saattaa haitata pelikokemusta. Ajan mukaisessa päivityksessä taas pystytään paremmin sopeutumaan eri kokoisiin päivitysaikoihin, jolloin vältetään myöhästymisen aiheuttamat ongelmat. Tarkemmin aihetta käsitellään luvuissa Säännöllisesti päivittyvä versio ja Ajan mukaan päivittyvä versio. Huomioitavaa on myös, että pelin alussa kulunut kokonaisaikaan lasketaan vain päivitetty aika, jolloin myöhästyminen ei näy kokonaisajassa. Tämän seurauksena kokonaisaika jää myös jälkeen todellisesta ajasta.

Kun tarkistetaan, onko edellisestä päivityksestä kulunut tarpeeksi aikaa, on huomioitava, että toteutuksessa lisäksi mahdollisesti keskeytetään säikeen toiminta. Jos seuraavaan päivitykseen on huomattavan paljon aikaa, niin tällöin säikeen toiminta keskeytetään muutamaksi millisekunniksi. Tällä on pyritty vähentämään suorittimen rasitusta, koska tiedämme, kuinka paljon aikaa on seuraavaan päivitykseen eikä siten sen jatkuvalle tarkistamiselle ole tarvetta.

7.3.3 Erilliset päivitykset -pääsilmukan kuvaus

Erilliset päivitykset omaava pääsilmukka (katso lähdekoodi listauksesta 9.12) pohjautuu luvussa Säännöllinen ja epäsäännöllinen päivitys erikseen esitettyyn pääsilmukkaan. Erityistä kyseisessä pääsilmukassa on, että siinä on eroteltu säännölliset ja epäsäännölliset päivitykset erilleen toisistansa. Toteutetun pelin tapauksessa säännöllisiä päivityksiä ovat muun muassa fysiikan päivittäminen ja tekoälyä käyttävien vihollisten päivittäminen. Epäsäännöllisiä päivityksiä taas ovat pelattavan hahmon päivittäminen syötteiden pohjalta ja animaatioiden päivitys.

Tällä jaolla on pyritty siihen, että pelin loogisen toiminnan kannalta oleelliset ominaisuudet päivittyvät deterministisesti (esim. fysiikkamoottori) käyttäen säännöllistä päivitystä. Samaan aikaan halutaan myös päivittää pelin logiikan kannalta epäoleellisia ominaisuuksia niin usein kuin vain laitteiston tehot mahdollistavat. Animaatiot hyötyvätkin mahdollisimman suuresta päivitystiheydestä ja niiden päivitys ei vaikuta pelin loogiseen toimintaan ja siksi sopivat erityisen hyvin epäsäännöllisesti päivitettäväksi. Aiheen teoreettista pohjaa on

käsitelty tarkemmin Säännöllinen ja epäsäännöllinen päivitys erikseen.

Erillisten päivitysten lisäksi huomioitavaa on, että säännöllinen päivitys on toteutettu silmukassa luvun Toistuvasti päivitys -malli ratkaisun avulla eli säännöllisiä päivityksiä voidaan tehdä useampi kerralla, kunnes on kurottu jälkeen jääty aika kiinni. Toisin kuin luvun Säännöllisesti päivittyvän pääsilmukan kuvaus toteutuksessa, tämän ratkaisun ansiosta tämä pääsilmukka ei jää jälkeen todellisesta ajasta vaikka päivitys myöhästyisikin huomattavasti.

Aikaan liittyen on myös oleellista, että eri päivitykset (eli säännöllinen ja epäsäännöllinen) käyttävät pelin käynnistymisestä kulunutta kokonaisaikaa hieman eri tavalla. Säännöllisen päivityksen tapauksessa kokonaisaika on aina päivitysvälin moninkerta. Esimerkiksi jos säännöllinen päivitys on asetettu tapahtumaan 0,016666 sekunnin välein ja säännöllisiä päivityksiä on tapahtunut sata, niin kokonaisajaksi saadaan 0,016666 kerrottuna sadalla.

Epäsäännöllien päivitys taas käyttää kokonaisaikana pelin käynnistymisestä kulunutta aikaa sellaisenaan. Eri päivitykset eivät voi käyttää samaa kokonaisaikaa, koska kokonaisaika kuvaa, kuinka paljon pelimaailmaa on yhteensä päivitetty. Luonnollisesti koska molemmat päivitykset ovat päivitettävän ajan suhteen erikokoisia, niin päivityksissä käytettyjen päivitysaikojen summa ei voi olla yhtä suuri. Esimerkiksi säännöllisiä päivityksiä voi olla tapahtunut yhteensä 6,12 sekunnin edestä, kun taas epäsäännöllisiä päivityksiä 6,1354 sekunnin edestä.

7.3.4 Interpoloivan pääsilmukan kuvaus

Interpoloiva pääsilmukka (katso lähdekoodi listauksesta 9.13) pohjautuu luvussa Interpoloiva malli esiteltyyn ratkaisuun. Kuitenkin ratkaisu on hyödyntää myös lukujen Toistuvasti päivitys -malli ja Säännöllinen ja epäsäännöllinen päivitys erikseen ratkaisuja kuten säännöllisen päivityksen toistaminen, kunnes on kurottu jälkeen jääty aika kiinni. Myös säännöllinen ja epäsäännöllinen päivitys on erotettu toisistansa, mitä ei alkuperäisessä lähteessä ollut käytetty. Päivitysten erottelun taustalla on se, lähdin kehittämään interpoloivaa silmukkaa luvun Erilliset päivitykset -pääsilmukan kuvaus toteutusta täydentämällä.

Uutena asiana interpoloiva silmukka tuo pelitilan interpoloinnin, jota käytetään kuvan piirtämisessä. Kuten luvun interpoloivan silmukan teoriassa kerrotaan, niin kuvan piirtämistä

varten voidaan laskea jokaisen kappaleen interpoloidut sijainnit nykyisen ja tulevan pelitilan väliltä. Varsinainen interpolointi on kuitenkin piilotettu pääsilmukalta, ja pääsilmukka tyytyy vain laskemaan interpolointiarvon, jolla kysytään piirettävät kappaleet pelimaailmalta. Pelimaailman puolella toteutuksessa käydään vain yksinkertaisesti kaikki kappaleet ja niiden muistissa olevat vanhat sijainnit läpi, ja niiden avulla lasketaan interpoloitu kappale.

Piirtäminen myös toimii interpoloinnin johdosta hieman muista silmukoista poikkeavasti. Muissa silmukoissa piirrettävät kappaleet ovat valmiina olemassa pelimaailman muistissa, mutta interpoloidessa taas joudutaan luomaan uusi lista interpoloiduille kappaleille. Tämän johdosta piirtämisen toteutus on hieman erilainen, koska interpoloidut kappaleet ovat vain väliaikaisia ja eivätkä siten ole osa pelimaailman tilaa. Lähinnä kuitenkin erona on vain, että piirtäminen on toteutettu eri funktiokutsulla parametrien tyyppien eroavaisuuksien takia.

Pienenä huomiona on, että myös kameran sijainti interpoloidaan kahden pelitilan välillä, koska muuten kameran liike ei olisi yhtä sulavaa kuin muu pelin toiminta. Kameran sijainnin interpolointi on toteutuksessa melko yksinkertaista, koska kameran sijainti on suoraan riippuvaista pelihahmon sijainnista. Tämän ansiosta kameran sijainti saadaan käytännössä interpoloimalla pelihahmon sijainti, mikä tehdään jo ennestään, kun interpoloidaan kaikki pelimaailman kappaleet.

Interpoloinnilla on pyritty mahdollistamaan se, että pystytään muodostamaan sulavasti päivittyvä kuva vaikka päivitys tehtäisiinkin harvoin. Tämä onnistuu, koska interpoloinnilla saadaan karkea arvio pelimaailmasta päivitysten väliltä eikä siten päivitystä tarvitse tehdä niin usein. Toisaalta ratkaisulla on myös pyritty takaamaan pelin deterministisyys, koska käytetään säännöllistä päivitystä.

Interpoloinnin lisäksi silmukan on mahdollista toimia myös ekstrapoloimalla eri ennustamalla kahden edellisen sijainnin perusteella kappaleen tuleva sijainti. Interpolointiin verrattuna ratkaisu tarjoaa nopeamman reaktion käyttäjän syötteisiin. Kuitenkin haittana on, että jos kappaleen nopeus tai suunta muuttuvat, niin tällöin käyttäjälle muutokset voivat näkyä hyvin äkillisinä. Muutostilanteissa näytetään käyttäjälle kuvaa sellaisesta pelitilasta, joka ei koskaan oikeasti toteudu edes.

Puutteista huolimatta toteutin ekstrapoloinnin kokeilemista varten, koska lineaarisen interpo-

laation muuttaminen lineaariseksi ekstrapolaatioksi käytännössä vaatii vain, että lisää arvon yksi osaksi interpolaatioarvoa. Tämä näkyy lähdekoodissa muuttujana self.ekstrapolointi_lisa, joka silmukkaa alustaessa asetetaan joko arvoksi yksi tai nolla.

Ekstrapoloinnin kokeilemiseen rohkaisi se, että ilmeisesti myös Quaksessa asiakkaan puolella ennakoidaan kappaleiden liikkeitä (katso luku Quaken asiakkaan rakenne ja päivitystiheys). Quakessa tosin taustalla on se, että kyseessä on verkkopeli, minkä seurauksena pelimaailman päivtykselle kertyy viivettä. Samaa ajatusta voidaan kuitenkin mielestäni myös testata pääsilmukkassakin laajemmin käytettynä.

7.4 Havaintoja silmukoiden toiminnasta erilaisilla päivitystiheyksillä

Silmukoita tarkastellessa on tärkeää kiinnittää huomiota, kuinka silmukka käyttäytyy erilaisilla päivitystiheyksillä. Erityisesti äärimmäisen suuri tai matala tiheys voi aiheuttaa ongelmia ja jopa virheellistä käytöstä pelin loogisen toiminnan kannalta. Tässä luvussa tarkastellaankin, kuinka kukin silmukka käyttäytyy eri päivitystiheyksillä. Varsinkin koska useimmat tutkittavista silmukoista sisältävät säännöllisen päivityksen, jonka voi vapaasti määrätä, on erilaisten päivitystiheyksien kokeileminen helposti toteutettavissa.

7.4.1 Ajan mukaan päivittyvän pääsilmukan havainnot

Toisin kuin muut tarkasteltavat silmukat, niin ajan mukaan päivittyvän silmukan päivitystiheyttä ei voida hallita ollenkaan. Päivittäminen tapahtuu niin nopeaa kuin vain laitteisto mahdollistaa, ja päivitystiheys voi vaihdellakin huomattavasti pelin suoritusaikana. Teorian pohjalta pystytään sanomaan, että ajan mukaan päivittyvä silmukka kohtaa haasteita, jos päivitystiheys on liian alhainen.

Peliä toteutettaessa eräässä vaiheessa ammukset eivät vielä tuhoutuneet seiniin osuessaan, mutta niillä oli yhä nopeus vaikka ne olivat pysähtyneet seinään kiinni. Tämän seurauksena ammukset saattoivat aika ajoin yhtä-äkkiä mennä seinistä lävitse. Tämän ilmiö korostui, koska pelissä on pienenä ongelmana, että muutaman sekunnin välein päivitysväli kasvaa huomattavasti. Tämän seurauksena tarpeeksi nopeat kappaleet menivät ohuiden seinien lävitse.

Ongelma on teorian (katso luku Ajan mukaan päivittyvä versio) puolelta tuttua ja kuten teoriassa todettiin, niin ajan mukaan päivittyvä silmukka vaatii tarpeeksi suuren päivitystiheyden tai ongelma pitää huomoida muualla toteutuksessa. Esimerkiksi fysiikkamoottorin toteutuksen tulisi itse tarkemmin havaita kappaleiden väliset törmäykset päivitysajasta riippumatta.

Quakessa ongelma oli ilmeisesti ratkaistu asettamalla päivitysajalle maksimi arvo, jolla vielä taataan pelin oikea toiminta (katso luku Quaken asiakkaan rakenne ja päivitystiheys). Tämä olisi oikein varteen otettava ratkaisu, jos haluttaisiin pelin varmasti toimivan oikein. Tällöin kuitenkin pitäisi vielä pelkän rajan lisäksi mahdollisesti toistaa päivityksiä useita peräkkäin, mikä muistuttaa luvun Toistuvasti päivitys -malli ratkaisua. Näin ilmeisesti tehtiin Quakessakin, mikä osoittaa, että ratkaisun olisi ainakin jossain määrin toimiva.

7.4.2 Säännöllisesti päivittyvän pääsilmukan havainnot

Säännöllisesti päivittyvälle silmukalle voidaan vapaasti asettaa käytettävä päivitystiheys. Tämä ei kuitenkaan välttämättä tarkoita, että päivitystiheys on oikeasti annettu, vaan se kertoo mihin pyritään. Luonnollisestikaan jos päivitys asetetaan tapahtumaan miljoona kertaa sekunnissa, niin laitteisto ei pysty sitä saavuttamaan suorituskyvyn puutteen takia.

Tämän takia onkin mielenkiintoista tarkastella, mitä tapahtuu jos asetettu päivitystiheys on suurempi kuin, mitä laitteisto pystyy saavuttamaan. Asettamalla päivitystiheydeksi 600 päivitystä sekunnissa sain tulokseksi, että pelin toiminta hidastuu. Käytännössä tämä näkyi siinä, että kaikkien kappaleiden (esim. ammusten) liike hidastui huomattavasti. Vielä suuremmilla päivitystiheyksillä peli hidastui entisestään, minkä perusteella voimme tehdä päätelmän, että suurempi päivitystiheys hidastaa enemmän pelin toimintaa.

Ilmiö johtuu luultavimmin siitä, että päivityksissä menee enemmän aikaa kuin mitä päivitettävä aika on. Esimerkiksi jos päivityksen suoritus vie aikaa 20 millisekuntia ja päivitysaika on 10 millisekuntia, niin tällöin 20 millisekunnin välein päivitetään 10 millisekunnin edestä pelimaailmaa. Tämä merkitsee, että pelin nopeus puolittuu, kun pelin päivittäminen tapahtuu hitaammin kuin, mitä aikaa oikeasti kuluu. Tämän perusteella päivitystiheyden tulee olla tarpeeksi pieni, jotta laitteisto pystyy suorittamaan sen tarpeeksi nopeasti.

Toinen ääripää päivitystiheydelle on tilanne, jossa päivitystiheys on pieni. Asettamalla pelin päivittymään vain kerran sekunnissa, havaitsin näkyvän eron pelin sulavuudessa. Esimerkiksi jos kuva päivittyy vain sekunnin välein, pelihahmot ja ammukset näyttävät siirtyvän äkillisesti pitkiä matkoja kerralla eikä siten tule vaikutelmaa kappaleiden tasaisesta ja sulavasta liikkeestä.

Suuren päivitysajan myötä myös törmäystarkastelun tarkkuus kärsi, jolloin ammukset ja pelihahmo itse pystyvät menemään seinien lävitse. Ilmiö on sama kuin luvussa Ajan mukaan päivittyvän pääsilmukan havainnot, jossa myös havaittiin pitkän päivitysvälin johtavan ongelmiin. Erona tosin on, että säännöllisesti päivitettäessä päivitysväli on asetettu valmiiksi ja siten voidaan itse valita tarpeeksi pieni päivitysväli ongelmien välttämiseksi. Tämä on teorian mukaan myös etu verrattuna ajan mukaan päivittämiseen, koska kehittäjä voi valita päivitystiheyden, joka varmasti toimii tarpeeksi hyvin.

Pienenä huomiona on myös, että melko tavallisessa päivitystiheydessä (eli 60 kertaa sekunnissa) esiintyi pieniä ongelmia. Muutaman sekunnin välein tapahtuva päivitysvälin äkillinen nousu näkyi säännöllisessä päivityksessä äkillisenä nytkähdyksenä. Päivitysvälin äkillinen nousu aiheuttaa siis pelille käytännössä hetkellisen pysähdyksen, jonka jälkeen peli jatkuu normaalisti.

Silmukka ei kuitenkaan huomioi pysähdystä mitenkään, minkä takia käyttäjä (oma havaintoni) kokee pientä epäsulavuutta kappaleiden liikkeessä. Tämä ei olisi ongelma, jos hetkellinen pysähtyminen tapahtuisi harvoin, mutta omassa toteutuksessani se tapahtuu aina muutaman sekunnin välein, mikä ei ole todellakaan suotavaa. Oikeasti julkaistavassa pelissä edellytettäisiinkin, että toistuva pelin pysähtyminen korjattaisiin.

7.4.3 Erilliset päivitykset -pääsilmukan havainnot

Erillisiä päivityksiä käytettäessä voidaan asettaa päivitystiheys säännöllisille päivityksille, kun taas epäsäännölliset tehdään niin usein kuin laitteisto mahdollistaa. Ero päivitysten välillä näkyykin selkeinten, kun asetetaan päivitystiheys pieneksi. Pienellä päivitystiheydellä pelin muu toiminta käytäytyy samalla tavalla kuin luvussa Säännöllisesti päivittyvän pääsilmukan havainnot, mutta animaatiot toimivat yhä sulavasti. Tämä on sikäli johdonmukaista,

koska teoriassa (katso luku Säännöllinen ja epäsäännöllinen päivitys erikseen) erityisesti korostettiin animaatioiden toimivan hyvin käytetyssä pääsilmukkamallissa.

Animaatiot ovat sulavia, koska niiden päivittäminen tehdään useammin kuin säännölliset päivitykset. Erillisten päivitysten käyttäminen vaikuttaakin olevan kehittyneempi versio säännöllisestä päivitystavasta, joka mahdollistaa useammin tehtävän päivityksen osalle toiminnoista.

Suurta päivitystiheyttä käytettäessä taas on mielenkiintoinen havainto, että peli toimii hyvin jopa todella suurilla päivitystiheyksillä. Esimerkiksi omalla laitteistollani 10 000 päivitystä sekunnissa onnistui vielä oikein sujuvasti, mikä luultavasti johtuu luvun Toistuvasti päivitys -malli ratkaisusta. Koska säännölliset päivitykset tehdään useita kerralla, niin ei ole tarvetta kuvan piirtämiselle. Luultavasti kuvan piirtäminen on se joka vie huomattavasti suoritusaikaa, ja sen takia päivitys voidaan tehdä useammin, kun jätetään kuvan piirtäminen pois.

Toisaalta tulee kuitenkin muistaa, että toteutettu peli on todella pieni, minkä takia päivitysten tekeminen ei ole suorituskyvylle niin raskasta. Suuremmissa peleissä säännöllisten päivitysten päivitystiheys ei voisi olla todellakaan yhtä suuri, koska tehtävät päivitykset olisivat luultavimmin raskaampia suorituskyvylle.

Jos taas päivitystiheyttä kasvatetaan vielä lisää, niin 40 000 kohdalla pelin toiminta alkaa selkeästi kärsiä. Peli toimii käynnistyttyään hetken normaalisti, mutta kuvan päivitys alkaa harventua ajan kuluessa kiihtyvällä tahdilla. Muutamien sekuntien kuluttua pelin käynistymisestä pelikuva päivittyy useamman sekunnin välein ja peli alkaa siten olla täysin pelikelvotonta.

Luultavammin ongelma johtuu päivitysten kasaantumisesta. Koska säännöllisten päivitysten suorituksessa menee enemmän aikaa kuin mitä uusia päivityksiä tulee tehtäväksi, niin seuraavalla silmukan kierroksella vääjäämättä on aina vain enemmän säännöllisiä päivityksiä, jotka tulisi tehdä. Tämän seurauksena jokainen silmukan kierros vie enemmän aikaa, mikä aiheuttaa harvemmin päivittyvän kuvan näytöllä.

Ongelma on vakava, koska liian suuri päivitystiheys tekee pelistä täysin pelikelvottoman. Toisaalta toteutetun pelin tapauksessa ongelma ei juurikaan esiinny, koska sitä varten tarvi-

taan tarpeettoman suuri päivitystiheys (ei siis ole mitään syytä, miksi päivitys suoritettaisiin tuhansia kertoja sekunnissa). Kuitenkin isommissa ja siten raskaammissa peleissä ongelma saattaa esiintyä varsinkin jos käytettävä laitteisto ei ole tarpeeksi tehokas.

Teorian puolella ongelmaan esitettään muutamia ratkaisuja, jotka kuitenkin toimivat vain väliaikaisiin viivästyksiin. Tämän takia on ehdottoman tärkeää, että pelille asetetaan tarpeeksi pieni päivitystiheys, jotta peli toimisi sujuvasti. Tämä ei ole kuitenkaan sinänsä uutta, koska muutkin säännölliset silmukat kärsivät samankaltaisista ongelmista. Erillisiä päivityksiä käytettäessä ongelma lähinnä vain korostuu, koska pelistä tosiaan tulee pelikelvoton sen sijaan, että peli vain hidastuisi.

7.4.4 Interpoloivan pääsilmukan havainnot

Interpoloiva pääsilmukan loogisesta toiminnassa voidaan havaita täysin samoja asioita kuin, mitä luvussa Erilliset päivitykset -pääsilmukan havainnot havaitaan. Tämä johtuu siitä, että toteutettu interpoloiva silmukka toimii kuvan piirtämistä lukuun ottamatta täysin samalla tavalla kuin erilliset päivitykset -silmukka. Tästä seuraa samanlainen käytös eri päivitystiheyksillä. Esimerkiksi pieni päivitystiheys aiheuttaa epätarkan fysiikan mallinnuksen (eli ammukset menevät seinistä herkästi läpi). Samoin taas liian suureksi asetetulla päivitystiheydellä pelistä tulee pelikelvoton jatkuvasti kasvavasta päivitysvälistä johtuen.

Kuitenkin uutena havaintona interpoloiva pääsilmukka tuo sen, että kuvan piirto on paljon kehittyneempi. Kappaleiden liikkeet ovat huomattavasti sulavampia, koska animaatioiden lisäksi myös kappaleiden piirrettävä sijainti päivitetään huomattavasti useammin kuin, mitä säännöllisiä päivityksiä tehdään. Erityisesti pienillä päivitystiheyksillä parannus korostuu, koska kappaleet liikkuvat täysin sulavasti vaikka säännöllinen päivitys asetettaisiin tapahtumaan vain kerran sekunnissa. Teorian (katso luku Interpoloiva malli) mukaan interpoloinnin tarkoitus onkin saavuttaa sulava kuvan päivitys, joten sen havaitseminen oli odotettavissa.

Pienenä huomiona tosin on, että sulava kappaleiden liikkuminen johtaa harhaan, että pelin päivitys tapahtuu tiheästi. Tämä ilmenee siitä, että kappaleet näyttävät liikkuvan sulavasti kohteiden lävitse, vaikka niiden kuuluisi törmätä. Luvussa Erilliset päivitykset -pääsilmukan havainnot samassa tilanteessa kappale liikkui aina ilmestymällä lyhyen matkan päähän, jol-

loin on selvää, että siirtymän väliin jääviä kohteita ei huomioida törmäystarkastelussa. Interpoloivassa silmukassa taas kappaleen kuva liikkuu toisen kappaleen ylitse, jolloin käyttäjän on vaikeampi havaita, mistä ongelma johtuu.

Toisena havaintona on myös, että koska ammukset luodaan epäsäännöllisissä päivityksissä, niin ammukset ovat hetken aikaa paikallaan ennen kuin ne lähtevät liikkeelle. Koska interpolointi tehdään kappaleen vanhan ja uuden sijainnin välillä, ei interpolointia voida tehdä juuri luoduille kappaleille, joilla ei vielä ole vanhaa sijaintia olemassa. Ongelman välttämiseksi olisi luultavasti parempi siirtää ammusten luominen (,joka on osa pelihahmon päivitystä) osaksi säännöllisiä päivityksiä. Tällöin ammukselle heti luomisen jälkeen laskettaisiin uusi sijainti, minkä seurauksena interpolointi olisi mahdollista vanhan ja uuden lasketun sijainnin välillä.

Pienenä ongelmana interpoloivassa silmukassa voidaan myös havaita viive käyttäjän syötteiden ja niistä seuraavien päivitysten välillä. Tämä ilmiö on käsitellään teorian puolella (katso luku Interpoloiva malli), mutta ilmiö on pelissä kohtuullisen pieni enkä olisi osannut kiinnittää siihen välttämättä huomiota ellei teoria olisi maininnut sitä. Esimerkiksi jos päivitystiheys on viisi säännöllistä päivitystä sekunnissa, niin pystyn itse hädin tuskin huomaamaan viiveen, mutta siitä huolimatta en koe sitä peliä häiritseväksi.

Paremmin ongelman pystyin havaitsemaan vielä pienemmillä päivitystiheyksillä, mutta peli alkaa tällöin jo muutenkin kärsiä fysiikkamoottorin tarkkuudesta liikaa. Lisäksi toteutettu peli ei vaikuta olevan kovinkaan vaativa viiveen suhteen, mutta jokin muu pelityyppi (esim. ensimmäisen persoonan ammuntapelit) saattaisi vaatia pienempää viivettä käyttäjän syötteiden ja toimintojen välille. Huomioitavaa on tosin, että vaikka viive pienenee suuremmilla päivitystiheyksillä, niin tällöin interpoloinnin tuomat hyödyt vähenevät. Interpoloinnilla pyritään kuvan päivityksen sulavuuteen, mutta suuri päivitystiheys takaa jo ennestään sen. Interpolointi tosin saattaa silti parantaa hieman sulavuuttaa, vaikka parannus olisikin suhteellisen pieni verrattuna pienempää päivitystiheyttä käytettäessä.

Interpoloinnin sijaan interpoloiva silmukka voidaan asettaa myös ekstrapoloimaan, jolloin tulee joitakin muutoksia. Yhtenä ideana ekstrapoloinnille olisi se, että ei tule interpoloinnin tavoin viivettä. Kuitenkin ongelma on, että pyritään ennustamaan kappaleen sijainti, mikä ei

onnistu, jos kappale muuttaa yllättäen suuntaa. Toteutuksessa voidaan havaita silloin äkillinen muutos kuvassa, jossa kappale vaihtaa yllättäen sijaintia mennäkseen toiseen suuntaan. Suunnan muutos ei siis tapahdu sulavasti vaan tapahtuu äkillisenä siirtymänä, joka pienillä päivitystiheyksillä voi olla todella suuri.

Ilmiö uskoakseni korostuu pelissäni, koska ohjattava pelihahmo muuttaa nopeutta äkillisesti eli esimerkiksi jos nopeus on aluksi 100, niin seuraavan päivityksen kohdalla se voi olla -100, jolloin mitään välivaiheita ei esiinny. Jos taas nopeuden muutos tapahtuisi asteittain ja esimerkiksi muutos sadasta miinus sataan kestäisi sekunnin ajan, niin uskoakseni ekstrapoloinnin aiheuttama muutos kuvassa tapahtuisi huomattavasti sulavammin. Toisaalta äkilliset muutokset muuttuisivat pienemmiksi, jos käytettäisiin suurempaa päivitystiheyttä säännöllisille päivityksille, mutta tällöin ekstrapoloinnin hyödyt (kuten myös interpoloinnin) vähenevät.

7.5 Tutkittavien pääsilmukoiden deterministisyys

Kuten luvussa Pelin deterministisyys havaitaan, niin pääsilmukan deterministisyys on hyvin oleellinen ominaisuus pääsilmukoita tarkastellessa. Deterministisyys helpottaa monien ominaisuuksien toteuttamista ja lisäksi tarjoaa helpon tavan toistaa mahdolliset virheelliset pelikerrat (esim. pelin kaatuminen). Koska erilaisten pääsilmukoiden deterministisyydellä on huomattavia eroja, on deterministisyys valittu yhdeksi vertailtavaksi ominaisuudeksi.

Deterministisyyden kannalta vertailussa tarvitsee kiinnittää huomiota lähinnä vain päivitysaikaan, koska pelissä ei ole juurikaan muita deterministisyyteen vaikuttavia ominaisuuksia. Esimerkiksi pelissä ei käytetä satunnaislukuja tai verkkoviestintää, jotka ovat tyypillisiä ominaisuuksia, joihin tulisi kiinnittää erityistä huomioita.

Kuitenkin vaikka tässä luvussa tarkastellaan deterministisyyttä, niin sitä ei kuitenkaan hyödynnetä pelin toteutuksessa. Esimerkiksi olisi mielenkiintoista kokeilla toteuttaa pelille mahdollisuus katsoa pelattu peli uudelleen, mutta ikävä kyllä ajan puutteen ja rajallisten resurssien takia deterministisyyden hyödyntämistä ei voida toteuttaa. Tässä luvussa tyydytäänkin vain tarkastelemaan ja arvioimaan, kuinka deterministisiä käytetyt pääsilmukat ovat.

7.5.1 Ajan mukaan päivittyvän pääsilmukan deterministisyys

Ajan mukaan päivittyvä pääsilmukka on teorian mukaan epädeterministinen johtuen juurikin viime päivityksestä kuluneen ajan käyttämisestä päivityksessä. Jokaisella pelikerralla saadaan erilaisia ja täysin satunnaisia päivitysaikoja, minkä seurauksena pelikerrat poikkeavat toisistaan vaikka yritettäisiin syöttää käyttäjän syötteet samanaikaisesti. Tämän takia esimerkiksi pelikerran uudelleen katsomisominaisuuden toteuttaminen olisi suhteellisen vaikeaa ja vaatisi, että käytetyt päivitysajat tallennetaan. Tämä ei olisi kuitenkaan ihanteellinen ratkaisu.

7.5.2 Säännöllisesti päivittyvän pääsilmukan deterministisyys

Säännöllisesti päivittyvä silmukka on teorian mukaan deterministinen. Myöskään toteutuksessa ei ole havaittavissa mitään sellaista, mikä saattaisi estää silmukan deterministisyyden, kuten satunnaislukujen käyttöä tai muuta vastaavaa. Säännöllinen päivitystavan etu onkin ajan mukaan päivittämiseen nähden se, että deterministisyys tulee säännöllisellä päivitystavalla luonnostaan. Säännöllisellä päivitystavalla pelin toistaminen onnistuu tallentamalla vain pelin alkutila ja kaikkien päivitysten käyttämät syötteet.

Toteutetussa pelissä käytännössä tarvitsisi tallentaa syötteiden tila aina kun kuunneltavien näppäinten tila päivitetään, mikä toteutuksessa on erityisen helppoa, koska syötteiden käsittely ja päivittäminen on tehty keskitetysti. Pelikerran toistamistakin varten tarvitsisi vain tehdä tavallista syötteiden käsittelyä muistuttava luokka, joka syötteiden lukemisen sijaan lukee näppäinten tilat tiedostosta. Tämän toteuttamiseen luultavasti kannattaisi tehdä syotteiden pyytämiselle oma rajapinta, jotta voisi helposti vaihtaa eri pelin ja toistamisen välillä.

7.5.3 Erilliset päivitykset -pääsilmukan deterministisyys

Erilliset päivitykset -pääsilmukka ei ole teorian mukaan välttämättä deterministinen, mutta sen on mahdollista olla. Deterministisyys riippu siitä, että mitkä kaikki päivitykset tehdään epäsäännöllisesti. Omassa toteutuksessani vain animaatioiden ja pelihahmon päivitys tehdään epäsäännöllisesti. Animaatioilla ei ole vaikutusta pelin loogiseen toimintaan eikä se siten vaikuta pelin deterministisyyteen. Sen sijaan pelattavan hahmon päivitys on ongelmal-

lisempi.

Pelihahmon päivityksessä voidaan muokata pelihahmon nopeutta, mutta pelihahmon sijaintia päivitetään vasta fysiikan päivityksessä. Tämän takia liikkuminen itsessään ei haittaa deterministisyyttä vaikkakin tekee pelin toisintamisesta hieman työläämpää. Pelihahmon nopeus on kuitenkin yksinkertaista käsitellä, koska pelihahmolle jää vain viimeisin nopeus voimaan vaikka nopeus vaihtuisikin monta kertaa ennen seuraavaa fysiikan päivitystä. Sen sijaan pelihahmon toinen ominaisuus eli ampuminen on paljon ongelmallisempi.

Ammuttaessa lisätään ammus heti pelimaailmaan, minkä seurauksena pelaaja voi ampua useita ammuksia ennen kuin seuraava fysiikan päivitys tehdään. Tämän takia voidaan luoda useita ammuksia riippuen siitä, montako epäsäännöllistä päivitystä pääsilmukka kerkeää tekemään ennen seuraavaa säännöllistä päivitystä. Tämän ongelmana on se, että toisella pelikerralla voidaan keretä tekemään epäsäännöllisiä päivityksiä vähemmän kuin, mitä luotavia ammuksia on tehty. Tämän takia peli ei ole sellaisenaan toistettavissa pelkkien syötteiden avulla.

Jotta peli voitaisiin toistaa, niin pitäisi erikseen tallentaa muistiin montako ammusta pelaaja on ampunut säännöllisten päivitysten välissä. Lisäksi peliä toistettaessa täytyisi tehdä ammusten lisäämiselle erikoiskäsittelyä, jotta kaikki ammukset lisättäisiin varmasti pelimaailmaan. Tarkemmin tarkasteltuna edes tämä ei riittäisi suoraan sellaisenaan, koska ammukset ammutaan siihen suuntaan, johon pelaajalla on sen hetkinen nopeus. Koska pelaaja voi vaihtaa suuntaa ampumisten välissä, niin eri ammukset voisivat mennä kaikki eri suuntiin. Tämän perusteella ammusten määrän lisäksi pitäisi myös ammusten suunnat tallentaa ja erikoiskäsitellä, mikä tekisi pelikerran toistamisen toteuttamisesta hyvin haastavaa ja työlästä.

Pelin deterministisyyden kannalta olisikin parempi siirtää pelihahmon päivitys osaksi säännöllistä päivitystä. Tällöin epäsäännölliseksi päivitykseksi jäisi vain animaatioiden päivitys, mikä ei vaikuta pelin deterministisyyteen. Kuitenkin toisaalta tulee huomoida se, että syötteiden päivitys tehdään epäsäännöllisesti, mikä aiheuttaa merkittäviä ongelmia. Koska ampuminen tehdään vain näppäintä painettaessa (eli jos näppäin pysyy pohjassa, niin ei ammuta), niin tällöin säännöllisen päivityksen tulisi tapahtua heti sen jälkeen, kun näppäin on painettu pohjaan.

Ikävä kyllä, koska syötteiden päivitys tehdään epäsäännöllisesti, niin on hyvinkin mahdollista, että syötteet päivitetään monta kertaa peräkkäin. Tämän seurauksena vaikka näppäin olisikin painettu pohjaan, niin se tulkitaan säännöllisen päivityksen kohdalla samalla tavalla kuin näppäin olisi ollut pitkään pohjassa. Ampumista ei siten tehtäisi vaikka käyttäjä olisikin painanut ampumisnäppäintä, mikä rikkoo hyvin vahvasti pelin toiminnan.

Ongelman korjaamiseksi tulisi myös syötteiden päivittäminen tehdä säännöllisesten päivitysten joukossa, mikä myös helpottaisi pelin toistettavuuden toteutusta. Korjausten jälkeen pelille toistettavuus voitaisiin tehdä täysin samalla tavalla kuin luvussa Säännöllisesti päivittyvän pääsilmukan deterministisyys. Muutoksista huolimatta silmukan toiminta pysyisi lähes täysin samanlaisena kuin ennen muutoksia. Verrattuna säännölliseen silmukkaan, parannuksena kuitenkin olisi animaatioiden sulava toimiminen käytettävästä päivitystiheydestä huolimatta.

7.5.4 Interpoloivan pääsilmukan deterministisyys

Toteutettu interpoloiva pääsilmukka toimii deterministisyyden suhteen täysin samalla tavalla kuin luvussa Erilliset päivitykset -pääsilmukan deterministisyys. Interpolointi vaikuttaa lähinnä vain kuvan piirtämiseen eikä pelin logiikkaan, ja siksi sillä ei ole vaikutusta silmukan deterministisyyteen. Näin ollen toteutettu interpoloiva pääsilmukka toimii deterministisesti pienillä aiemmassa luvussa mainituilla korjauksilla. Kuitenkin pienenä huomiona on, että korjaukset joilla pelihahmon päivitys siiirrettäisiin säännöllisesti päivitettäväksi, ratkaisisi myös erään visuaalisesti häiritsevän ongelman.

Kuten luvussa Interpoloivan pääsilmukan havainnot huomattiin, niin pienellä päivitystiheydellä näyttää siltä, että ammus on hetken paikallaan ennen kuin se lähtee liikkeelle. Tämä johtuu pitkälti siitä, että ammus luodaan epäsäännöllisessä päivityksessä, mutta sitä liikutetaan vasta säännöllisessä päivityksessä.

Ilmiö tapahtuu myös erilliset päivitykset -pääsilmukassa, mutta siinä se ei ole niin häiritsevää, koska pienellä päivitystiheydellä kaikki muutkin kappaleet pysyvät paikallaan. Interpoloivassa pääsilmukassa taas kaikki muut kappaleet liikkuvat interpolaation ansiosta sujuvasti, jolloin paikallaan oleva ammus korostuu erityisesti. Tämän takia käsitellyt korjaukset

olisivat erityisen suositeltavia interpoloivalle pääsilmukalle.

Korjausten jälkeen pelin toistaminen on helppoa toteuttaa interpoloiva pääsilmukalle, ja lisäksi koska pelikerta vain toistetaan, niin käyttäjältä ei tarvitse kysyä syötteitä. Tämän seurauksena interpolaatiossa ilmenevä syötteen ja sen käsittelyn viive muuttuu täysin merkityksettömäksi. Itse asiassa peliä toistettaessa voisi olla järkevää käyttää interpolaatiota vaikkei pääsilmukka oikean pelikerran aikana käyttäisikään. Interpoloimalla voidaan mahdollistaa pelikerran toistolle suurempi kuvan päivitystiheys kuin, mitä pelatessa on ollut käytössä.

7.6 Tutkittavien pääsilmukoiden toteutuksen vaativuus

Eräs tärkeä ominaisuus pääsilmukoita verrattaessa on, että kuinka työläitä ja vaikeita ne ovat toteuttaa. Lukijalle tämä tieto on oleellinen, jotta lukija osaisi paremmin arvioida, mitä pääsilmukkaa kannattaisi itse käyttää. Esimerkiksi jos paremmin soveltuvan pääsilmukan saa toteutettua pienellä lisävaivalla, niin luonnollisesti kannattaa valita paremmin soveltuva. Vastaavasti jos pääsilmukka on todella työlästä toteuttaa, niin kannattaa miettiä, onko kyseinen silmukka tarpeellinen.

Löytämissäni lähteissä ei juurikaan pohdittu pääsilmukoiden toteutusten työläyttä eikä varsinkaan verrattuna muihin silmukoihin, minkä takia jäi lukijan omalle vastuulle arvioida toteutuksen työläys. Tässä luvussa pyritäänkin luomaan karkeaa käsitystä, miten esitellyt silmukat eroavat toisistaan työläyden ja haastavuuden suhteen.

Vertailussa hyödynnetään omia kokemuksiani niiden toteuttamisesta, sillä toteutuksia tehdessä luonnollisesti sain jonkinlaista kuvaa työläydestä ja haastavuudesta. Lisäksi koska toteutettu peli tehtiin versionhallintaa käyttäen, niin pääsen käsiksi pelin eri versioihin, ja voin siten verrata muutosten määrää eri versioiden välillä. Kuitenkin esitellyt arviot ovat vain suuntaa antavia, ja esimerkiksi tietyn pääsilmukan työläys saattaa vähentyä merkittävästi, jos jo kehityksen alusta asti suunnitellaan toteutus käyttämään tiettyä pääsilmukkamallia. Omassa toteutuksessani itse vain muokkasin vanhaa pääsilmukkaa käyttävää toteutusta aina käyttämään myös uutta pääsilmukkaa, mikä lisäsi työläyttä.

7.6.1 Ajan mukaan päivittyvän pääsilmukan vaativuus

Ajan mukaan päivittyvän pääsilmukan tein ensimmäisenä, koska pidin sitä kaikista yksinkertaisimpana pääsilmukkana toteuttaa. Myös arvelin, että se on ominaisuuksiltaan hyvin yleisluontoinen ja muiden silmukoiden rakentaminen sen päälle olisi helppoa.

Silmukan helppous tulee siitä, että siinä hyvin suoraviivaisesti käytetään päivitysaikaa hyödyksi. Käytännössä vain lasketaan, kuinka paljon aikaa on kulunut edellisestä päivityksestä, ja annetaan se päivitysfunktiolle parametrinä. Koska muutkin toteutetut silmukat antavat jonkinlaisen päivitysajan päivitysfunktiolle, niin silmukka toimi hyvänä pohjana muille silmukoille. Silmukka myös tarvitsee vain yhdenkaltaisia päivityksiä, minkä ansiosta tarvitsin vain yhden päivitysfunktion toteutuksen, joka sitten huolehtii päivityksistä tarkemmin.

7.6.2 Säännöllisesti päivittyvän pääsilmukan vaativuus

Vertaamalla versionhallinnassa säännöllisesti päivittyvän pääsilmukan tuomia muutoksia lähdekoodissa, huomataan merkittäviä muutoksia vain lisätyn pääsilmukan sisältävässä tiedostossa. Tämä viittaa siihen, että silmukan lisääminen ei vaatinut lähdekoodin muokkaamista, joten voidaan keskittyä pelkästään säännöllisesti päivittyvän silmukan omaan työläyteen.

Vertaamalla ajan mukaan päivittyvää pääsilmukkaa ja säännöllisesti päivittyvää pääsilmukkaa havaitaan, että ne ovat suurelta osin samanlaisia. Käytännössä erona toteutuksissa on vain päivitysajan käsittelyssä. Ajan mukaan päivittyvässä pääsilmukassa edellisestä päivityksestä kulunut aika voidaan antaa sellaisenaan päivitysfunktiolle. Säännöllisessä päivityksessä joudutaan vertaamaan päivitysaikaa asetettuun päivitysväliin ja toimimaan sen mukaan. Käytännön tasolla tämä tarkoittaa paria ehtolausetta enemmän kuin ajan mukaan päivittyvässä silmukassa.

Tämän perusteella totetettu säännöllisesti päivittyvä pääsilmukka on hieman työläämpi toteuttaa, mutta ei kovinkaan paljoa. Myös oman kokemukseni mukaan säännöllisen päivityksen toteuttaminen oli helppoa, ja teoriaan tutustumisen jälkeen ei ollenkaan työlästä. Käytännössä samalla vaivalla toteuttaa säännöllisesti päivittyvän pääsilmukan kuin ajan mukaan päivittyvän pääsilmukan.

7.6.3 Erilliset päivitykset -pääsilmukan vaativuus

Erilliset päivitykset -pääsilmukan lisäävässä versionhallinnan versiossa näkyy huomattavia muutoksia lähdekoodissa. Suurimmat muutokset tulevat siitä, että päivitykset on jouduttu pilkkomaan pienemmiksi. Sekä säännöllisesti päivittyvässä pääsilmukassa ja ajan mukaan päivittyvässä pääsilmukassa riittää vain yksi päivitysfunktio, jota kutsutaan. Sen sijaan erilliset päivitykset -pääsilmukassa tarvitaan ainakin kaksi eri päivitysfunktiota eli funktiot säännöllisiä ja epäsäännöllisiä päivityksiä varten.

Jotta päivitysten muokkaaminen olisi helpompaa, niin päädyin silmukkaa toteuttaessa ratkaisuun, jossa pilkoin jokaisen päivityksen omaksi pieneksi päivitykseksi, jotta voisin helposti vaihtaa yksittäisiä pieniä päivityksiä osaksi säännöllistä tai epäsäännöllistä päivitystä. Esimerkiksi yksittäisiä päivityksiä ovat pelihahmon päivitys, fysiikan päivitys, spawnerien päivitys ja muuta vastaavaa. Käyttämäni ratkaisu kuitenkin näkyi työläytenä, koska aiemmissa päivityksissä kaikki päivitykset olivat yhdessä ja samassa päivitysfunktiossa. Muutoksen seurauksena kaikki päivitykset piti erotella toisistansa, mihin meni huomattavasti aikaa.

Päivitysten erottamisen lisäksi työläyttä tuli ajan käsittelystä. Ennen muokkausta pelin alusta kulunut aika oli pelimaailman ominaisuus, mutta muokkauksen jälkeen siitä tuli päivityksen ominaisuus. Muutoksen taustalla oli se, että erilliset päivitykset -pääsilmukassa, on kaksi eri päivitystä, jotka käyttävät eri päivitysaikoja. Eri päivitysaikojen myötä, myös pelin alusta kulunut kokonaisaikakin on hieman erilainen. Ajan muokkaaminen ei sinänsä ollut iso muutos, mutta muutos piti tehdä useisiin eri tiedostoihin, joissa pelin alusta kulunutta aikaa käytetään.

Edellä kuvatut muokkaukset eivät sinänsä lisänneet peliin mitään, mutta niitä vaadittiin, jotta erilliset päivitykset -pääsilmukka voisi toimia. Työläys ei siten tullut koodin lisäämisestä vaan johtui puutteellisesta suunnitelusta. Jos alusta asti olisi ollut tarkoitus toteuttaa erilliset päivitykset -pääsilmukka, niin olisin ohjelmoinut pelin toimimaan alusta asti oikealla tavalla, eikä muutoksille olisi ollut tarvetta. Tämän takia työläys johtui siitä, että jälkeen päin muokattiin valmiiksi toimivaa ratkaisua eikä silmukan varsinaisista ominaisuuksista. Toisaalta kuitenkin voidaan huomioida, että erilliset päivitykset -pääsilmukka ei ole välttämättä pienellä vaivalla muokattavissa säänöllisesti tai ajan mukaan päivittyvästä pääsilmukasta.

Silmukan itsensä tuomaa työläyttä tuli taas silmukan omasta toteutuksesta. Vertaamalla erilliset päivitykset -pääsilmukkaa jo käsitellyihin silmukoihin, huomataan isoja eroja silmukoiden välillä. Koska erilliset päivitykset -pääsilmukka käyttää luvun Toistuvasti päivitys -malli ratkaisua, niin päivitysajan käsittely vaatii erikoiskäsittelyä, jotta säännöllinen päivitys toistetaan useita kertoja tarvittaessa. Lisäksi tehdään myös epäsäännöllinen päivitys, mutta epäsäännöllinen päivitys on käytännössä sama kuin ajan mukaan päivittyvässä silmukassa.

Oman kokemukseni mukaan säännöllisen päivityksen toistaminen monta kertaa vaatii tarkkaa suunnittelua ja teoriaan perehtymistä, mutta sen toteuttaminen ei ole varsinaisesti kovinkaan työlästä. Erilliset päivitykset -pääsilmukan työläys tuleekin lähinnä vain pelin muista muokkauksista, mutta nekin suurelta osin voidaan jättää vähälle huomiolle, jos jo peliä suunnitellessa otetaan silmukan vaatimukset huomioon. Jälkeen päin muokkaaminen on kuitenkin huomattavan työlästä, mutta onnistuu tarvittaessa.

7.6.4 Interpoloivan pääsilmukan vaativuus

Interpoloiva pääsilmukka pohjautuu vahvasti erilliset päivitykset -pääsilmukkaan, joten samat asiat esiintyvät kuin luvussa Erilliset päivitykset -pääsilmukan vaativuus. Kuitenkin interpolointi tuo omat lisänsä silmukan ja koko pelin toimintaan. Interpoloivaa pääsilmukkaa varten piti tehdä pieniä muokkauksia animaatioiden käsittelyyn ja kuvan piirtämiseen, mutta suunnitelulla nekin olisivat varmaan suoraan menneet oikein ilman, että olisi tarvinnut muokata jälkeenpäin.

Ihan uutena asiana tuli uudessa silmukassa se, että interpolointia varten tarvitsee muistaa kappaleiden vanhat sijainnit, mikä vaatii huomattavia lisäyksiä tietorakenteisiin. Itse onnistuin löytämään mielestäni ihan hyvän ratkaisun, koska ajattelin vanhojen tilojen säilytystä kappaleen lisäosana, ja lisäosia olin jo ennestään toteuttanut peliin. Kuitenkin tämäkin vaati tarkkaa suunnittelua enkä pitäisi itse sitä kovinkaan helppona toteuttaa tehokkaasti.

Varsinainen interpolointifunktion toteuttaminen taas ei tuonut itselleni lisää työtaakkaa, koska se oli toteutettu jo ennestään animaatioita varten. Interpolaatiofunktion tekeminen itsessään ei kuitenkaan ole kovinkaan työlästä, mutta itselläni meni huomattavasti aikaa siihen, että sain sen käyttämään rajapintoja tarkkojen tyyppien sijaan. Esimerkiksi rajapinnoilla to-

teutuksessa oli mahdollista interpoloida vektoreita tai liukulukuja samalla funktiolla. Tässä kuitenkin haasteena oli oma rajoittunut osaamiseni Rust ohjelmointikielestä, ja joku toinen olisi mahdollisesti toteuttanut interpoloinnin huomattavasti nopeammin.

Interpoloivan silmukan tekeminen taas ei ollut kovinkaan vaikeaa tai työlästä, ja käytännössä interpoloivalla silmukalla ja erilliset päivitykset -silmukalla on hyvin vähän eroja rakenteessa. Käytännössä vain kuvan piirtäminen käsitellään hieaman eri tavalla, mikä ei ole kovinkaan iso muutos. Ekstrapoloinnin toteuttaminen taas interpoloivalle silmukalle käytännössä vain vaati arvon yksi lisäämistä interpolaatioarvoon eli todella helposti ja pienellä vaivalla interpoloivasta silmukasta sai ekstrapoloivan.

Yleisesti ottaen kuitenkin pitäisin interpoloivaa silmukkaa melko haastavana suunnitella verrattuna jo esiteltyihin silmukoihin. Suurimpana haasteena on suunnitella, kuinka vanhat kappaleiden sijainnit säilytetään ja käsitellään. Itse interpolointifunktion toteuttaminen taas ei ollut kovinkaan vaikeaa itselleni, mutta se vaatii hieman matemaattista osaamista. Toisaalta minun tarvitsi vain interpoloida sijainteja, mutta jos olisi tarvinnut interpoloida kolmiulotteisen kappaleen kulmia, niin olisin ollut aivan hukassa. Teorian mukaan sen pitäisi olla mahdollista (katso sivu 21), mutta se vaatisi paljon enemmän perehtymistä. Toisaalta voidaan pohtia, olisiko kulman interpolointi edes välttämätöntä ja riittäisikö pelkän sijainnin interpoloiminen. Esimerkiksi Quake ennakoi pelkästään kappaleen sijainnin muutoksen eikä kulman muutosta (Sanglard 2009).

7.7 Yhteenveto omasta pelistä

Edellä esitelty peli on mielestäni toteutettu varsin onnistuneesti. Pelissä käytettiin useita erilaisia pääsilmukoita, ja pelin käyttämä pääsilmukka voidaan vaihtaa todella helposti. Näiden pääsilmukoiden toimintaa tarkkailemalla onnistuttiin vahvistamaan monia teoriassa esiteltyjä ominaisuuksia, ja jopa havaittiin joitakin teoriassa mainitsemattomia käyttäytymistapoja. Esimerkiksi oli mielenkiintoista havaita, että erilliset päivitykset -pääsilmukassa säännöllinen päivitys on mahdollista tehdä todella usein, ja omalla laitteellani jopa yli 10 000 kertaa ilman mitään ongelmia.

Myös tärkeä tulos on se, että valitut silmukat ylipäätään onnistuttiin toteuttamaan. Tämän

perusteella voidaan sanoa, että kaikki käsitellyt pääsilmukat ovat aivan hyvin toteutettavissa. Varsinkin arvokasta lisää tuo se, että käsiteltiin jokaisen pääsilmukan työläyttä. Tulosten perusteella voidaankin sanoa, että hyvällä suunnittelulla ajan mukaan päivittyvän, säännöllisesti päivittyvän ja erilliset päivitykset -pääsilmukan välillä ei ole juurikaan eroa työläyden suhteen. Erityisesti erilliset päivitykset -pääsilmukan kohdalla huolellisen suunnittelun vaatimus korostuu, mutta havaintojen perusteella toisaalta se tarjoaa paremmat ominaisuudet kuin toiset kaksi pääsilmukkaa. Paremmuus näkyy siinä, että erilliset päivitykset -pääsilmukan on mahdollista toimia deterministisesti, ja silti se tarjoaa samaan aikaan laadukkaat animaatiot.

Kuitenkin havaintojen perusteella laadun suhteen interpoloiva pääsilmukka on erilliset päivitykset -pääsilmukkaa parempi, koska se tarjoaa animaatioiden lisäksi muillekin kappaleille sulavat liikkeet (eli hyvän kuvan päivityksen). Pienenä haittana tosin on pieni viive syötteiden ja syötteistä seuraavien toimintojen välillä, mutta itse en kokenut sitä merkittäväksi ongelmaksi.

Merkittävä ongelma, joka voisi estää interpoloivan pääsilmukan käyttämisen, on luultavammin silmukan tuoma ylimääräinen työmäärä. Lisäksi interpoloiva pääsilmukka on oman kokemukseni mukaan huomattavasti haastavampi toteuttaa tehokkaasti, mutta se ei ole kuitenkaan ylitse pääsemätön este.

Tutkittaviksi valituissa pääsilmukoissa on tosin pienenä puutteena se, että yksikään niistä ei hyödynnä rinnakkaisuutta, vaikka tässä työssä aihetta käsiteltiin runsaasti. Syynä tähän oli yksinkertaisesti ajan puute ja rinnakkaisuuden toteuttaminen olisi vaatinut huomattavia muutoksia pelin toimintaan. Puute on merkittävä, koska rinnakkaistetut pääsilmukat pyrkivät parantamaan pelin suorituskykyä, mikä on peleille tärkeä ominaisuus. Rinnnakkaistettujen silmukoiden puutteen takia myös jätettiin eri silmukoiden suorituskyvyn vertailu tekemättä, koska teorian mukaan mikään käsitellyistä silmukoista ei pyrkinyt parantamaan tehokkuutta, mikä vähensi suorituskyvyn vertailun mielekkyyttä.

Toisaalta myös ajan puute vaikutti vertailtavien ominaisuuksien rajaamiseen. Suorituskyvyn lisäksi alkuperäisenä tarkoituksena olisi ollut myös verrata silmukoiden muistin käyttöä, mikä olisi luultavasti interpoloivan silmukan tapauksessa ollut kiintoisaa, mutta ajan puutteen takia se jouduttiin jättämään pois tutkimuksesta.

8 Yhteenveto

Tutkimuksessa esiteltiin pääsilmukkoihin liittyvää teoriaa ja erilaisia pääsilmukkamalleja. Rinnakkaistamattomissa pääsilmukoissa suurimpia eroja oli lähinnä vain päivitystavassa, mikä on sinänsä järkevää, koska tehtävien järjestyksellä ei ole niin paljon merkitystä, jos ei tehdä rinnakkaistusta. Rinnakkaistamattomissa pääsilmukoissa parannukset yleensä näkyivät kuvan päivityksen parempana sulavuutena tai sitten deterministisyydessä.

Rinnakkaistetut pääsilmukat taas keskittyivät hyvin vahvasti pelin suorituskyvyn parantamiseen, mihin rinnakkaistuksella yleensäkin pyritään. Kuitenkin havaintona voidaan tehdä, että rinnakkaistetut pääsilmukat ovat vaikeampia ja työläämpiä toteuttaa, mikä ilmeni myös siinä, että ne jäivät omassa pelissäni toteuttamatta. Työläyden suhteen taas rinnakkaistamattomat pääsilmukat olivat melko lailla saman tasoisia ja ainoastaan interpoloinnin lisääminen toi merkittävästi lisää työtaakkaa.

Tutkimuksessa mielestäni onnistuin esittelemään erilaisia pääsilmukoita hyvin ja samalla tarjosin tietoa niiden eroista. Näiden tietojen pohjalta lukijalla tulisi olla hyvät mahdollisuudet valita ja toteuttaa omaan tilanteeseensa sopivin pääsilmukka. Lukija voi myös hyödyntää tässä työssä esiteltyjä valmiiden toteutuksien lähdekoodia, mikä helpottaa oman pääsilmukan toteutusta.

Parannettavaa tutkimuksessa olisi voinut olla, jos olisi pelin toteutuksessa myös tehty rinnakkaistettu pääsilmukka, jolloin olisi saatu parempaa käsitystä rinnakkaistamisen haasteista. Myös mielenkiintoista olisi ollut verrata pelin suorituskykyä rinnakkaistamattoman ja rinnakkaistetun pääsilmukan välillä. Kuitenkaan rinnakkaisuutta ei voitu tehdä ajan puutteen vuoksi.

9 Liitteet

```
var config = {
    type: Phaser.CANVAS,
    width: 800,
    height: 600,
    backgroundColor: '#9adaea',
    useTicker: true,
    scene: {
        preload: preload,
        create: create,
        update: update
    }
};

var game = new Phaser.Game(config);

function preload ()
{
    //Ladataan esim. kuvia
}

function create ()
{
    // Luodaan pelimaailma, asetetaan nappaimet jne.
}

function update (time, delta)
{
    // Paivitetaan pelimaailman tilaa
}
```

Listing 9.1. Tämä on karkea pohja minkälainen Phaserillä tehty peli voisi tyypillisesti olla. Käytännössä alustuksen lisäksi pelinkehittäjä vain kertoo millä tavalla peliä päivitetään.

```
/**
 * The main step method. This is called each time the browser updates,
    either by Request Animation Frame,
 * or by Set Timeout. It is responsible for calculating the delta values,
     frame totals, cool down history and more.
 * ...
 */
step: function (time)
{
    // ...
    //  When a browser switches tab, then comes back again, it takes
       around 10 frames before
    //  the delta time settles down so we employ a 'cooling down' period
       before we start
    //  trusting the delta values again, to avoid spikes flooding through
        our delta average
    if (this._coolDown > 0 || !this.inFocus)
    {
        this._coolDown--;
        dt = Math.min(dt, this._target);
    }
    if (dt > this._min)
    {
        //  Probably super bad start time or browser tab context loss,
        //  so use the last 'sane' dt value
        dt = history[idx];

        //  Clamp delta to min (in case history has become corrupted
            somehow)
        dt = Math.min(dt, this._min);
    }

    //  Smooth out the delta over the previous X frames

    //  add the delta to the smoothing array
    history[idx] = dt;

```

```
32      //  adjusts the delta history array index based on the smoothing
            count
33      //  this stops the array growing beyond the size of deltaSmoothingMax
34      this.deltaIndex++;
35
36      if (this.deltaIndex > max)
37      {
38          this.deltaIndex = 0;
39      }
40
41      //  Delta Average
42      var avg = 0;
43      //  Loop the history array, adding the delta values together
44      for (var i = 0; i < max; i++)
45      {
46          avg += history[i];
47      }
48
49      //  Then divide by the array length to get the average delta
50      avg /= max;
51
52      //  Set as the world delta value
53      this.delta = avg;
54
55      //  Real-world timer advance
56      this.time += this.rawDelta;
57      // ...
58      this.callback(time, avg, interpolation);
59      // ...
60  },
```

Listing 9.2. Tämä listaus sisältää step funktion tiedostosta \phaser-3.16.1\src\core\TimeStep.js versio 3.16.1. Haettu osoitteesta `http://phaser.io/download/release/3.16.1`

```
/**
 * The main Game Step. Called automatically by the Time Step, once per
     browser frame (typically as a result of
 * Request Animation Frame, or Set Timeout on very old browsers.)
 * ...
 */
step: function (time, delta)
{
    // ...
    var eventEmitter = this.events;

    //  Global Managers like Input and Sound update in the prestep
    eventEmitter.emit(Events.PRE_STEP, time, delta);

    //  This is mostly meant for user-land code and plugins
    eventEmitter.emit(Events.STEP, time, delta);

    //  Update the Scene Manager and all active Scenes
    this.scene.update(time, delta);

    //  Our final event before rendering starts
    eventEmitter.emit(Events.POST_STEP, time, delta);

    var renderer = this.renderer;

    // ...
    //  The main render loop. Iterates all Scenes and all Cameras in
        those scenes, rendering to the renderer instance.
    this.scene.render(renderer);
    // ...
},
```

Listing 9.3. Tämä listaus sisältää step funktion tiedostosta \phaser-3.16.1\src\core\Game.js versio 3.16.1. Haettu osoitteesta `http://phaser.io/download/release/3.16.1`

```
/**
 * Advances the simulation based on the elapsed time and fps rate.
 *
 * This is called automatically by your Scene and does not need to be
   invoked directly.
 *
 * ...
 */
update: function (time, delta)
{
    // ...

    var stepsThisFrame = 0;
    var fixedDelta = this._frameTime;
    var msPerFrame = this._frameTimeMS * this.timeScale;

    this._elapsed += delta;
    this._late = false;

    while (this._elapsed >= msPerFrame)
    {
        this._elapsed -= msPerFrame;
        stepsThisFrame++;
        this.step(fixedDelta);
    }

    this.stepsLastFrame = stepsThisFrame;
    this._late = true;
},
```

Listing 9.4. Tämä listaus sisältää update funktion tiedostosta \phaser-3.16.1\src\physics\arcade\World.js versio 3.16.1. Haettu osoitteesta `http://phaser.io/download/release/3.16.1`

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class EsimerkkiSkripti : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {

    }
}
```

Listing 9.5. Esimerkki, miltä Unityn käyttämä skripti näyttää.

```
1  /* main window message loop */
2  while (1)
3  {
4      // yield the CPU for a little while when paused, minimized, or not
          the focus
5      if ((cl.paused && (!ActiveApp && !DDActive)) || Minimized ||
          block_drawing)
6      {
7          SleepUntilInput (PAUSE_SLEEP);
8          scr_skipupdate = 1;     // no point in bothering to draw
9      }
10     else if (!ActiveApp && !DDActive)
11     {
12         SleepUntilInput (NOT_FOCUS_SLEEP);
13     }
14
15     newtime = Sys_DoubleTime ();
16     time = newtime - oldtime;
17     Host_Frame (time);
18     oldtime = newtime;
19 }
```

Listing 9.6. Quaken asiakkaan pääsilmukka Quake World -päivityksestä. Koodi on WinMain-funktiosta QW\client\sys_win.c tiedostosta. Lähdekoodi ladattu lähteestä id Software (2019b).

```
//
// main loop
//
oldtime = Sys_DoubleTime () - 0.1;
while (1)
{
    // select on the net socket and stdin
    // the only reason we have a timeout at all is so that if the last
    // connected client times out, the message would not otherwise
    // be printed until the next event.
    FD_ZERO(&fdset);
    FD_SET(net_socket, &fdset);
    timeout.tv_sec = 0;
    timeout.tv_usec = 100;
    if (select (net_socket+1, &fdset, NULL, NULL, &timeout) == -1)
        continue;

    // find time passed since last cycle
    newtime = Sys_DoubleTime ();
    time = newtime - oldtime;
    oldtime = newtime;

    SV_Frame (time);
}
```

Listing 9.7. Quaken palvelimen pääsilmukka Quake World -päivityksestä. Koodi on WinMain-funktiosta QW\server\sys_win.c tiedostosta. Lähdekoodi ladattu lähteestä id Software (2019b).

```c
void SV_Frame (float time)
{
    // ...
    // decide the simulation time
    if (!sv.paused) {
        realtime += time;
        sv.time += time;
    }

    // check timeouts
    SV_CheckTimeouts ();

    // toggle the log buffer if full
    SV_CheckLog ();

    // move autonomous things around if enough time has passed
    if (!sv.paused)
        SV_Physics ();

    // get packets
    SV_ReadPackets ();

    // check for commands typed to the host
    SV_GetConsoleCommands ();

    // process console commands
    Cbuf_Execute ();
    SV_CheckVars ();

    // send messages back to the clients that had packets read this frame
    SV_SendClientMessages ();
    // ... Statistiikkaa yms.
}
```

Listing 9.8. Osa Quaken palvelimen SV_Frame-funktion Quake World -päivityksestä. Koodi on QW\server\sv_main.c tiedostosta. Lähdekoodi ladattu lähteestä id Software (2019b).

```
void Host_Frame (float time)
{
    // ...

    if (!cls.timedemo && realtime - oldrealtime < 1.0/fps)
        return;          // framerate is too high

    host_frametime = realtime - oldrealtime;
    oldrealtime = realtime;
    if (host_frametime > 0.2)
        host_frametime = 0.2;

    // get new key events
    Sys_SendKeyEvents ();

    // allow mice or other external controllers to add commands
    IN_Commands ();

    // process console commands
    Cbuf_Execute ();

    // fetch results from server
    CL_ReadPackets ();

    // send intentions now
    // resend a connection request if necessary
    if (cls.state == ca_disconnected) {
        CL_CheckForResend ();
    } else
        CL_SendCmd ();

    // Set up prediction for other players
    CL_SetUpPlayerPrediction(false);

    // do client side motion prediction
    CL_PredictMove ();

```

```
38      // Set up prediction for other players
39      CL_SetUpPlayerPrediction(true);
40
41      // build a refresh entity list
42      CL_EmitEntities ();
43
44      // ...
45
46      SCR_UpdateScreen ();
47
48      // ...
49
50      CDAudio_Update();
51
52      // ...
53  }
```

Listing 9.9. Osa Quaken asiakkaan Host_Frame-funktion Quake World -päivityksestä. Koodi on QW\client\cl_main.c tiedostosta. Lähdekoodi ladattu lähteestä id Software (2019b).

```
1  fn kaynnista_silmukka(&mut self) -> Result<(), String> {
2      // Alustetaan aikaan liittyvat muuttujat
3      let mut _timer = self.context.timer()?;
4      let mut peliaika = Instant::now();
5      let mut kokonaisaika_pelin_alusta = Duration::new(0, 0);
6      let mut vanha_peliaika = peliaika;
7      let mut paivitysaika;
8
9      // Alustetaan maailma
10     let mut maailma = Perusmaailma::new();
11     self.paivitys
12         .alusta(&mut maailma, &mut self.syotteet, &self.events);
13
14     // Varsinainen paasilmukka
15     'paasilmukka: loop {
16         // Kerataan tapahtumat
17         for event in self.events.poll_iter() {
18             match event {
19                 // Tarkistetaan suljetaanko peli (esim. ikkunan X
                       klikkaamalla tai Esc nappainta painamalla)
20                 Event::Quit { .. }
21                 | Event::KeyDown {
22                     keycode: Some(Keycode::Escape),
23                     ..
24                 } => {
25                     // Poistutaan paasilmukasta eli kaytannossa
                           lopetetaan peli
26                     break 'paasilmukka;
27                 }
28                 _ => {}
29             }
30         }
31         // Lasketaan paivitysaika ja paivitetaan kokonaisaikaa pelin
               alusta
32         peliaika = Instant::now();
33         paivitysaika = peliaika.duration_since(vanha_peliaika);
34         kokonaisaika_pelin_alusta += paivitysaika;
```

```
35          vanha_peliaika = peliaika;
36
37          // Paivitetaan syotteiden tilaa
38          self.syotteet.paivita_nappainten_tilat(&self.events);
39
40          // Paivitetaan maailman tilaa
41          self.paivitys.paivita(
42              &mut maailma,
43              &mut self.syotteet,
44              &Paivitysaika::new(&paivitysaika, &kokonaisaika_pelin_alusta)
45          );
46
47          // Poistetaan maailmasta poistettaviksi merkityt kappaleet
48          maailma.poista_poistettavat();
49
50          // Piirretaan maailma ja animaatiot
51          self.piirtaja.puhdista_kuva();
52          self.piirtaja.piirra_maailma(&maailma)?;
53          self.piirtaja.piirra_kappaleista(&maailma.animaatio_kuva)?;
54          self.piirtaja.esita_kuva();
55      }
56      Ok(())
57  }
```

Listing 9.10. Ajan mukaan päivittyvä pääsilmukka, tiedostosta perussilmukka.rs. Tiedosto löytyy osoiteesta https://github.com/1kasu/gpeli/blob/graduversio/src/silmukka/perussilmukka.rs.

```
fn kaynnista_silmukka(&mut self) -> Result<(), String> {
    // ...Alustuksia

    let lepoaika = 5;

    // ...Alustuksia

    'paasilmukka: loop {
        for event in self.events.poll_iter() {
           // ...Suljetaanko peli...
        }
        // Lasketaan paivitysaika
        peliaika = Instant::now();
        paivitysaika = peliaika.duration_since(vanha_peliaika);

        // Tarkistetaan onko viime paivityksesta kulunut tarpeeksi aikaa
        if paivitysaika < self.paivitysvali {
            // Jos seuraavaan paivitykseen on liian paljon aikaa, niin
            // keskeytetaan saikeen toiminta hetkeksi
            if (self.paivitysvali - paivitysaika).as_micros() > lepoaika
                {
                _timer.delay(lepoaika as u32);
            }
            // Palataan silmukan alkuun
            continue;
        } else {
            // Paivitetaan paivitysajat
            paivitysaika = self.paivitysvali
        }

        vanha_peliaika = peliaika;
        kokonaisaika_pelin_alusta += paivitysaika;

        // Paivitetaan syotteiden tila
        self.syotteet.paivita_nappainten_tilat(&self.events);

        // Paivitetaan pelimaailma
```

```
37          self.paivitys.paivita(
38              &mut maailma,
39              &mut self.syotteet,
40              &Paivitysaika::new(&paivitysaika, &kokonaisaika_pelin_alusta)
41          );
42  
43          // ...Piirretaan kuva
44      }
45      Ok(())
46  }
```

Listing 9.11. Säännöllisesti päivittyvä pääsilmukka, tiedostosta saannollinensilmukka.rs. Tiedosto löytyy osoitteesta `https://github.com/1kasu/gpeli/blob/graduversio/src/silmukka/saannollinensilmukka.rs`.

```
fn kaynnista_silmukka(&mut self) -> Result<(), String> {
    // ... Alustetaan

    'paasilmukka: loop {
        for event in self.events.poll_iter() {
            // ... Suljetaanko peli
        }

        // Lasketaan paivitysaika
        peliaika = Instant::now();
        paivitysaika = peliaika.duration_since(vanha_peliaika);
        kokonaisaika_pelin_alusta += paivitysaika;
        aikaa_seuraavaan_saannolliseen_paivitykseen += paivitysaika;
        vanha_peliaika = peliaika;

        // Paivitetaan syotteet
        self.syotteet.paivita_nappainten_tilat(&self.events);

        // Toteutetaan niin, monta saannollista paivitysta, kuin mita ollaan jaaty jalkeen
        while aikaa_seuraavaan_saannolliseen_paivitykseen >= self.paivitysvali {
            kokonaisaika_pelin_alusta_saannollinen += self.paivitysvali;
            // Suoritetaan saannollinen paivitys
            self.saannollinen_paivitys.paivita(
                &mut maailma,
                &mut self.syotteet,
                &Paivitysaika::new(&self.paivitysvali, &kokonaisaika_pelin_alusta_saannollinen)
            );
            aikaa_seuraavaan_saannolliseen_paivitykseen -= self.paivitysvali;
        }

        // Tehdaan epasaannollinen paivitys
        self.epasaannollinen_paivitys.paivita(
            &mut maailma,
```

```
34              &mut self.syotteet,
35              &Paivitysaika::new(&paivitysaika, &kokonaisaika_pelin_alusta)
36          );
37
38          // ... Piirretaan kuva yms.
39      }
40      Ok(())
41  }
```

Listing 9.12. Pääsilmukka joka sisältää erikseen säännöllisen ja epäsäännöllisen päivityksen. Pääsilmukka on tiedostosta erillisetpaivitykset.rs. Tiedosto löytyy osoiteesta `https://github.com/1kasu/gpeli/blob/graduversio/src/silmukka/erillisetpaivityksetsilmukka.rs`.

```
1  fn kaynnista_silmukka(&mut self) -> Result<(), String> {
2      // ... Alustetaan
3
4      'paasilmukka: loop {
5          for event in self.events.poll_iter() {
6              // ... Suljetaanko silmukka
7          }
8
9          // Lasketaan paivitysaika
10         peliaika = Instant::now();
11         paivitysaika = peliaika.duration_since(vanha_peliaika);
12         kokonaisaika_pelin_alusta += paivitysaika;
13         aikaa_seuraavaan_saannolliseen_paivitykseen += paivitysaika;
14         vanha_peliaika = peliaika;
15
16         // Paivitetaan syotteet
17         self.syotteet.paivita_nappainten_tilat(&self.events);
18
19         // Toteutetaan niin, monta saannollista paivitysta, kuin mita
               ollaan jaaty jalkeen
20         while aikaa_seuraavaan_saannolliseen_paivitykseen >= self.
               paivitysvali {
21             kokonaisaika_pelin_alusta_saannollinen += self.paivitysvali;
22             // Suoritetaan saannollinen paivitys
23             self.saannollinen_paivitys.paivita(
24                 &mut maailma,
25                 &mut self.syotteet,
26                 &Paivitysaika::new(&self.paivitysvali, &
                       kokonaisaika_pelin_alusta_saannollinen)
27             );
28             maailma.paivita_kappalemuistia();
29             aikaa_seuraavaan_saannolliseen_paivitykseen -= self.
                   paivitysvali;
30         }
31
32         // Tehdaan epasaannollinen paivitys
33         self.epasaannollinen_paivitys.paivita(
```

```
34                &mut maailma,
35                &mut self.syotteet,
36                &Paivitysaika::new(&paivitysaika, &kokonaisaika_pelin_alusta)
                      ,
37            );
38
39            maailma.poista_poistettavat();
40
41            let mut piirrettavat_kappaleet = Vec::new();
42
43            maailma.aseta_interpolaatio_arvo(
44                self.ekstrapolointi_lisa
45                    + aikaa_seuraavaan_saannolliseen_paivitykseen.as_micros()
                          as f32
46                        / self.paivitysvali.as_micros() as f32,
47            );
48
49            if let Some(kamera) = maailma.anna_kameran_sijainti() {
50                self.piirtaja.aseta_kameran_sijainti(kamera)?;
51            }
52
53            maailma.anna_piirrettavat(&mut piirrettavat_kappaleet);
54
55            // Piirretaan maailma ja animaatiot
56            self.piirtaja.puhdista_kuva();
57            self.piirtaja.piirra_kappaleista(&piirrettavat_kappaleet)?;
58            self.piirtaja.piirra_kappaleista(&maailma.animaatio_kuva)?;
59            self.piirtaja.esita_kuva();
60        }
61        Ok(())
62    }
```

Listing 9.13. Pääsilmukka, joka interpoloi piirrettävän kuvan kahden pelitilan väliltä. Löytyy tiedostosta
interpoloivasilmukka.rs. Tiedosto löytyy osoitteesta `https://github.com/1kasu/gpeli/blob/graduversio/src/silmukka/interpoloivasilmukka.rs`.

Lähteet

Abdelkhalek, Ahmed, ja Angelos Bilas. 2004. "Parallelization and performance of interactive multiplayer game servers". Teoksessa *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International,* 72. IEEE.

Amdahl, Gene M. 1967. "Validity of the single processor approach to achieving large scale computing capabilities". Teoksessa *Proceedings of the April 18-20, 1967, spring joint computer conference,* 483–485. ACM.

Andrews, Jeff. 2009a. "Designing the framework of a parallel game engine". *Intel Corporation, Santa Clara, CA, USA, June.*

———. 2009b. "Threading Basics for Games". Viitattu 25. toukokuuta 2019. `https://web.archive.org/web/20110312070719/http://software.intel.com:80/en-us/articles/threading-basics-for-games/`.

Barney, Blaise, ym. 2010. "Introduction to parallel computing". *Lawrence Livermore National Laboratory* 6 (13): 10.

Best, Micah J, Alexandra Fedorova, Ryan Dickie, Andrea Tagliasacchi, Alex Couture-Beil, Craig Mustard, Shane Mottishaw, Aron Brown, Zhi Feng Huang, Xiaoyuan Xu ym. 2009. "Searching for concurrent design patterns in video games". Teoksessa *European Conference on Parallel Processing,* 912–923. Springer.

Bonastre, Jordi. 2016. "Why should I use Threads instead of Coroutines?" Viitattu 25. toukokuuta 2019. `https://support.unity3d.com/hc/en-us/articles/208707516-Why-should-I-use-Threads-instead-of-Coroutines-`.

Computer History Museum. 2019. "Spacewar!" Viitattu 25. toukokuuta 2019. `https://web.archive.org/web/20181005130115/https://www.computerhistory.org/pdp-1/spacewar/`.

Damon, William. 2011. “Multithreaded Game Programming and Hyper-Threading Technology”. Viitattu 25. toukokuuta 2019. https://web.archive.org/web/20170527215625/https://software.intel.com/en-us/articles/multithreaded-game-programming-and-hyper-threading-technology.

Davey, Richard. 2013. “Announcing Phaser (Flixel HTML5) and our Adobe Max session”. Viitattu 25. toukokuuta 2019. https://web.archive.org/web/20180121071938/http://www.photonstorm.com/phaser/announcing-phaser-flixel-html5-and-our-adobe-max-session.

———. 2017. “Phaser 3 Dev Log # 78”. Viitattu 25. toukokuuta 2019. https://web.archive.org/web/20180406182750/https://phaser.io/phaser3/devlog/78.

———. 2019. “Phaser 3.16.2 Released”. Viitattu 25. toukokuuta 2019. https://phaser.io/news/2019/02/phaser-3162-released.

Dickinson, Patrick. 2001. “Instant Replay: Building a Game Engine with Reproducible Behavior”. Viitattu 25. toukokuuta 2019. https://web.archive.org/web/20160409150436/http://www.gamasutra.com/view/feature/131466/instant_replay_building_a_game_.php.

El Rhalibi, Abdennour, Madjid Merabti ja Yuanyuan Shen. 2006. “Improving game processing in multithreading and multiprocessor architecture”. Teoksessa *International Conference on Technologies for E-Learning and Digital Entertainment,* 669–679. Springer.

Fiedler, Glenn. 2004. “Fix Your Timestep”. Viitattu 25. toukokuuta 2019. https://web.archive.org/web/20181107181440/https://gafferongames.com/post/fix_your_timestep/.

Gardner, William G. 1999. “3D audio and acoustic environment modeling”. *Wave Arts, Inc.–1999.–109 p.*

Gullen, Ashley. 2016. “Progress in Framerate-Independent Platforming”. Viitattu 25. toukokuuta 2019. https://www.construct.net/en/forum/construct-2/general-discussion-17/progress-in-framerate-independ-116068/page-2#forumPost848224.

Haas, John K. 2014. “A history of the Unity game engine”.

id Software. 2019a. “Quake”. Viitattu 25. toukokuuta 2019. https://www.idsoftware.com/en-gb.

———. 2019b. “Quake”. Viitattu 25. toukokuuta 2019. https://github.com/id-Software/Quake.

Jones, Simon Peyton. 2007. “Beautiful concurrency”. *Beautiful Code: Leading Programmers Explain How They Think:* 385–406.

Joselli, Mark, ja Esteban Clua. 2009. “Gpuwars: Design and implementation of a gpgpu game”. Teoksessa *2009 VIII Brazilian Symposium on Games and Digital Entertainment,* 132–140. IEEE.

Joselli, Mark, Esteban Clua, Anselmo Montenegro, Aura Conci ja Paulo Pagliosa. 2008. “A new physics engine with automatic process distribution between CPU-GPU”. Teoksessa *Proceedings of the 2008 ACM SIGGRAPH symposium on Video games,* 149–156. ACM.

Joselli, Mark, Marcelo Zamith, Esteban Clua, Regina Leal-Toledo, Anselmo Montenegro, Luis Valente, Bruno Feijó ja Paulo Pagliosa. 2010. “An architeture with automatic load balancing for real-time simulation and visualization systems”. *JCIS-Journal of Computational Interdisciplinary Sciences* 1 (3): 207–224.

Käyttäjätunnus Animmaniac. 2016. “Progress in Framerate-Independent Platforming”. Viitattu 25. toukokuuta 2019. https://www.construct.net/en/forum/construct-2/general-discussion-17/progress-in-framerate-independ-116068/page-2#forumPost848227.

Lake, Adam, ja Henry Gabb. 2005. “Threading 3D Game Engine Basics”. Viitattu 25. toukokuuta 2019. https://web.archive.org/web/20170811061106/http://www.gamasutra.com:80/view/feature/130873/threading_3d_game_engine_basics.php.

Landsteiner, Norbert. 2015. “Inside Spacewar! A Software Archeological Approach to the First Video Game”. Viitattu 25. toukokuuta 2019. https://web.archive.org/web/20181004223858/https://www.masswerk.at/spacewar/inside/.

McShaffry, Mike. 2014. *Game coding complete.* Nelson Education.

Mozilla. 2019. “2D collision detection”. Viitattu 25. toukokuuta 2019. https://web.archive.org/web/20190406003701/https://developer.mozilla.org/en-US/docs/Games/Techniques/2D_collision_detection.

Mönkkönen, Ville. 2006. “Multithreaded Game Engine Architectures”. Viitattu 25. toukokuuta 2019. https://web.archive.org/web/20131114044711/http://www.gamasutra.com/view/feature/130247/multithreaded_game_engine_.php?page=1.

Phaser Dokumentaatio. 2019. “Phaser.Input.Keyboard.KeyboardPlugin”. Viitattu 25. toukokuuta 2019. https://photonstorm.github.io/phaser3-docs/Phaser.Input.Keyboard.KeyboardPlugin.html.

Photon Storm. 2019. “Phaser - HTML5 Game Framework”. Viitattu 25. toukokuuta 2019. https://github.com/photonstorm/phaser.

Press, William H, Saul A Teukolsky, William T Vetterling ja Brian P Flannery. 2007. *Numerical recipes 3rd edition: The art of scientific computing.* Cambridge university press.

Sanglard, Fabien. 2009. “Quake engine code review”. Viitattu 25. toukokuuta 2019. https://web.archive.org/web/20190204081458/http://fabiensanglard.net/quakeSource/index.php.

Sevenson, Andrew. 2009. “Separating Axis Theorem (SAT) Explanation”. Viitattu 25. toukokuuta 2019. https://web.archive.org/web/20190309060810/https://www.sevenson.com.au/actionscript/sat/.

Shin, Kang G, ja Parameswaran Ramanathan. 1994. “Real-time computing: A new discipline of computer science and engineering”. *Proceedings of the IEEE* 82 (1): 6–24.

Silverman, Michael. 2008. “Scalable Multithreaded Game Engines Using Transactional Memory”. *Undergraduate thesis, University of Rochester.*

Tulip, James, James Bekkema ja Keith Nesbitt. 2006. “Multi-threaded Game Engine Design”. Teoksessa *Proceedings of the 3rd Australasian Conference on Interactive Entertainment,* 9–14. IE ’06. Perth, Australia: Murdoch University. ISBN: 86905-902-5. http://dl.acm.org/citation.cfm?id=1231894.1231896.

Unity Technologies. 2018a. “C# Job System”. Viitattu 25. toukokuuta 2019. https://docs.unity3d.com/2018.3/Documentation/Manual/JobSystem.html.

———. 2018b. “Event Functions”. Viitattu 25. toukokuuta 2019. https://docs.unity3d.com/2018.3/Documentation/Manual/EventFunctions.html.

———. 2018c. “Execution Order of Event Functions”. Viitattu 25. toukokuuta 2019. https://docs.unity3d.com/2018.3/Documentation/Manual/ExecutionOrder.html.

———. 2018d. “Releasing the Unity C# source code”. Viitattu 25. toukokuuta 2019. https://blogs.unity3d.com/2018/03/26/releasing-the-unity-c-source-code/.

———. 2018e. “Resources.LoadAsync”. Viitattu 25. toukokuuta 2019. https://docs.unity3d.com/2018.3/Documentation/ScriptReference/Resources.LoadAsync.html.

———. 2019a. “Multithreaded Rendering & Graphics Jobs”. Viitattu 25. toukokuuta 2019. https://web.archive.org/web/20190208225727/https://unity3d.com/learn/tutorials/topics/best-practices/multithreaded-rendering-graphics-jobs.

———. 2019b. “Unity Personal”. Viitattu 25. toukokuuta 2019. https://store.unity.com/products/unity-personal.

Unity Technologies, Anthony. 2014. “High-performance physics in Unity 5”. Viitattu 25. toukokuuta 2019. https://web.archive.org/web/20170925193853/https://blogs.unity3d.com/2014/07/08/high-performance-physics-in-unity-5/.

Valente, Luis, Aura Conci ja Bruno Feijó. 2005. “Real time game loop models for single-player computer games”. Teoksessa *Proceedings of the IV Brazilian Symposium on Computer Games and Digital Entertainment,* 89:99.

Witters, Koen. 2009. “deWiTTERS Game Loop”. Viitattu 25. toukokuuta 2019. https://web.archive.org/web/20180911164428/http://www.koonsolo.com/news/dewitters-gameloop/.

Yakovlev, Anthony. 2018. “Physics Changes in Unity 2018.3 Beta”. Viitattu 25. toukokuuta 2019. https://web.archive.org/web/20190114194418/https://blogs.unity3d.com/2018/11/12/physics-changes-in-unity-2018-3-beta/.