Ville Mönkkönen

# MONISÄIKEISTYS PELIMOOTTOREISSA

Tietojärjestelmätieteen
Pro gradu -tutkielma
10.06.2006

Jyväskylän yliopisto
Tietojenkäsittelytieteiden laitos
Jyväskylä

# TIIVISTELMÄ

Mönkkönen, Ville Samuli
Monisäikeistys pelimoottoreissa/Ville Mönkkönen
Jyväskylä: Jyväskylän yliopisto, 2006.
88 s.
Pro Gradu -tutkielma

Tässä tutkielmassa tarkastellaan korkean tason malleja pelimoottorien rinnakkaisohjelmointiin. Tavoitteena on löytää ja vertailla sellaisia malleja, jotka mahdollistaisivat moniytimisten prosessorien tehokkaan hyödyntämisen. Tutkielma pyrkii tarjoamaan riittävästi tietoa käytettävissä olevista vaihtoehdoista, jotta pelimoottoria kehitettäessä voidaan tehdä valinta eri ratkaisujen välillä.

Malleja etsitään kirjallisuuskatsauksen avulla. Yksi malli myös kehitetään tutkimusta varten. Esitettyjä malleja vertaillaan niiden laadullisten ominaisuuksien ja odotettavan suorituskyvyn suhteen. Vertailussa käytetään ohjelmistoarkkitehtuurien analysointiin tarkoitettua SAAM-menetelmää.

Tutkimuksen tuloksena on kolmen eri rinnakkaistusmallin esittely ja vertailu. Malleista kaksi perustuu pelimoottorin eri tehtävien jakamiseen eri säikeisiin. Yksi malli pohjautuu eri olioiden jakamiseen eri säikeisiin. Mallien välillä löydettiin eroja erityisesti niiden skaalautuvuudessa eri prosessoriydinmäärille sekä niiden vaatimista muutoksista nykyisiin pelimoottorin komponentteihin. Lisäksi mallien havaittiin soveltuvan huonosti Playstation 3:n Cell-prosessorille.

AVAINSANAT: rinnakkaisohjelmointi, pelimoottorit, moniydinprosessorit

# SISÄLLYSLUETTELO


1 Johdanto....5
1.1 Tutkimuksen tausta ja tutkimusongelma....5
1.2 Tutkimusmenetelmä....7
1.3 Tutkimusraportin rakenne....8

2 Rinnakkaiskäsittely....10
2.1 Johdanto....10
2.2 Rinnakkaisohjelmointi....11
2.2.1 Rinnakkaisohjelmiston suunnittelu....12
2.2.2 Tehtävien välinen viestintä....14
2.2.3 Käyttöjärjestelmän prosessit....15
2.2.4 Säikeet....16
2.2.5 Rinnakkaisuuden hyödyt....17
2.2.6 Rinnakkaisohjelmoinnin kuvaukset....19
2.2.7 Ongelmat rinnakkaisohjelmoinnissa....22
2.3 Moniydinprosessorit....25
2.3.1 PC- ja konsoliprosessorien moniydintekniikat....26
2.3.2 Hypersäikeistys....27
2.3.3 Moniydintekniikan käyttökohteet....28
2.3.4 Rinnakkaisohjelmointi moniydinprosessorilla....29
2.4 Yhteenveto....30

3 Pelimoottorit....32
3.1 Johdanto....32
3.1.1 Pelisilmukka....34
3.1.2 Oliomallin mukainen pelisilmukka....35
3.1.3 Pelimoottorin rakenne....35
3.2 Pelimoottorin komponentit....38
3.2.1 Grafiikka....39
3.2.2 Syöte....39
3.2.3 Äänet....40
3.2.4 Verkko....40
3.2.5 Skriptaus....41
3.2.6 Fysiikka....42
3.3 Yhteenveto....43

4 Monisäikeisen pelimoottorin mallit....44
4.1 Johdanto....44
4.2 Triviaalit ratkaisut....44
4.3 Komponenttien sisäinen rinnakkaistus ja tehtäväjonot....45
4.4 Tehtäväperustainen rinnakkaistus....47
4.5 Tietoperustainen rinnakkaistus....50
4.6 Yhteenveto....52

5 Monisäikeisen pelimoottorin mallien vertailu....53
5.1 Arvioinnin menetelmä....54
5.2 Suorituskyvyn arviointi....56
5.2.1 Skenaariot....56
5.2.2 Asynkroninen tehtäväpohjainen rinnakkaistus....57
5.2.3 Synkroninen tehtäväpohjainen rinnakkaistus....60
5.2.4 Olioperustainen rinnakkaistus....62
5.2.5 Suorituskyvyn arvioinnin yhteenveto....64
5.3 Laadun arviointi....67
5.3.1 Skenaariot....67
5.3.2 Asynkroninen tehtäväpohjainen rinnakkaistus....68
5.3.3 Synkroninen tehtäväpohjainen rinnakkaistus....72
5.3.4 Olioperustainen rinnakkaistus....74
5.3.5 Laadun arvioinnin yhteenveto....77

6 Yhteenveto....80

7 Lähdeluettelo....84

# 1 JOHDANTO

## 1.1 Tutkimuksen tausta ja tutkimusongelma

Prosessorien tehoa ei voida enää kasvattaa kellotaajuutta nostamalla siinä määrin kuin se tähän asti on ollut mahdollista. Tämä johtuu alati kasvavasta lämmöntuotosta, jonka perässä kuluttajatason jäähdytysjärjestelmät eivät pysy (Sutter, 2005). Niinpä prosessorivalmistajat Intel ja AMD ovat prosessorien tehoa kasvattaakseen siirtymässä moniytimisiin prosessoreihin. Moniytimellisessä prosessorissa useampi kuin yksi prosessoriydin sulautetaan samalle prosessorille niin, että ytimet jakavat osan prosessorin resursseista. Moniytimellisiä prosessoreja käytetään myös uusissa pelikonsoleissa: Microsoftin Xbox 360 konsolissa sekä Sonyn Playstation 3:ssa.

Useampi ydin ei automaattisesti tarkoita suoritustehon lisäystä ohjelmissa. Moniytimisyydestä hyötyvät ainoastaan ohjelmistot, joiden toiminta on mahdollista jakaa näille lisääntyneille ytimille. Tekniikoita, jotka mahdollistavat tämän kutsutaan *rinnakkaisohjelmoinniksi*. Rinnakkaisohjelmointi (concurrent programming, parallel programming tai multiprogramming) tarkoittaa Hansenin (1973) määritelmän mukaan niitä ohjelmointitekniikoita, joilla ohjataan rinnakkaisia prosesseja eli tehtäviä. Sutter (2005) ja Olukotun & Hammond (2005) maalaavat kauhukuvia siitä, että yksisäikeiset ohjelmistot eivät enää jatkossa juuri lainkaan nopeutuisi prosessorien päivittyessä, kuten tähän asti on aina tapahtunut. Edessä on suuri muutos kohti rinnakkaisohjelmoinnin aikakautta.

Varsinkin peleille on tyypillistä ottaa kaikki teho irti käytettävissä olevasta laitteistosta. Tästä johtuen peliteollisuus pyrkii kehittämään menetelmiä muuttaa pelit tukemaan moniytimisiä prosessoreita. Nykyään erotetaan toisistaan pelien sisältö ja tekninen toteutus. Pelin teknisestä toteutuksesta käytetään nimitystä *pelimoottori*. Jacobson & Lewis (2002) määrittelevät pelimoottorin simulaatiota ajavien moduulien kokoelmaksi, joka ei suoraan määrittele pelin toimintaa (game logic) tai pelin sisältöä. Blow (2004) vertailee vuoden 1994 ja 2004 pelimoottorin monimutkaisuutta arvioi-

malla pelimoottorin komponenttien määrää. Vuonna 1994 komponenttien määrä on ollut noin 5, mutta vuonna 2004 tyypillinen pelimoottori käyttää 15 erilaista komponenttia. Tällä perusteella pelit ohjelmistoina monimutkaistuvat jatkuvasti. Koska pelimoottorit ovat monimutkaisia sovelluksia, täytyy niiden rinnakkaisohjelmointiin kiinnittää erityistä huomiota. Monimutkaisuuden lisäksi ongelmana on, että pelisovelluksen prosessointi on luonteeltaan peräkkäistä – pelaajalle esitetään pelin tapahtumat peräkkäisten piirrettyjen ruutujen sarjana (Gabb & Lake, 2005).

Tässä kohtaa on syytä ottaa käyttöön pelimoottorien rakennetta ja toimintaa käsiteltäessä hyödyllinen käsite, nimittäin *ohjelmistoarkkitehtuuri.* Tässä tutkimuksessa käytettynä käsitettä parhaiten kuvaa IEEE standardi 1471, joka määrittelee ohjelmistoarkkitehtuurin *järjestelmän keskeiseksi rakenteeksi, jota ilmentävät sen komponentit, niiden väliset suhteet, komponenttien suhteet ympäristöön, ja periaatteet, jotka ohjaavat järjestelmän suunnittelua ja kehitystä*.

Tämän tutkimuksen tavoitteena on löytää ja vertailla pelimoottorin arkkitehtuurille malleja, jotka soveltuisivat kuluttajille suunnatuille moniytimisille prosessoreille. Pyrkimyksenä on tarjota riittävästi tietoa käytettävissä olevista vaihtoehdoista, jotta pelimoottoria kehitettäessä voidaan tehdä valinta eri ratkaisujen välillä.

Aikomuksena ei siis ole löytää valmiita arkkitehtuureja jotka suoraan soveltuisivat käytettäväksi jossakin pelimoottorissa. Tavoitteena on etsiä malleja, joiden perusteella pelimoottorin arkkitehtuuri voidaan suunnitella. Käsitteenä tämä tunnetaan nimellä *arkkitehtuurimalli* (architectural pattern). Buschmann et al. (1996) määrittelevät, että arkkitehtuurimalli ”*kuvaa pohjimmaista rakenteellista kaavaa ohjelmistolle. Se tarjoaa joukon ennalta määriteltyjä alijärjestelmiä, kuvaa niiden tehtävät, ja sisältää säännöt ja ohjeet niiden välisten suhteiden kehittämiseen.*”

## 1.2 Tutkimusmenetelmä

Erityisiä osaongelmia tutkimuksessa ovat:

- **Taustatietojen tarkastelu, johon kuuluu soveltuvien rinnakkaisohjelmoinnin alueiden, moniytimisten prosessorien ja pelimoottorien arkkitehtuurien ymmärtäminen.** Tämä toteutetaan pääosin kirjallisuuskatsauksena, mutta erityisesti pelimoottoreiden yhteydessä on kirjallisuuden vähyydestä johtuen tarpeen turvautua tarkastelemaan aiheeseen liittyvää teknistä dokumentaatiota.

- **Monisäikeisten pelimoottorin arkkitehtuurimallien löytäminen ja luominen.** Mallin luominen on tarpeellista, sillä johtuen aiheen tuoreudesta ei kirjallisuudesta ole löydettävissä riittävän paljon valmiita malleja.

- **Monisäikeisten pelimoottorin arkkitehtuurimallien vertailu.** Vertailu eri arkkitehtuurimallien välillä tehdään arvioimalla kutakin mallia Kazman et al. (1994) esittämällä skenaariopohjaisella SAAM-menetelmällä. Vertailun tavoitteena ei ole antaa selkeitä numeroarvoja, vaan tarjota tietoa eri mallien soveltuvuudesta eri tapauksissa.

Arkkitehtuurimallien vertailuun soveltuisi erinomaisesti niiden tutkiminen prototyyppien avulla. Prototyypit päätettiin kuitenkin jättää toteuttamatta, sillä niiden toteuttamatta jättäminen mahdollistaa kuitenkin laajemman yleistettävyyden – prototyypit johtaisivat väistämättä arvioihin jonkin yksittäisen pelin tai pelityypin näkökulmasta.

## 1.3 Tutkimusraportin rakenne

Luku 2 käsittelee tutkimuksen aiheen välitöntä teoriataustaa, eli rinnakkaisohjelmointia ja moniydinprosessoreita. Rinnakkaisohjelmoinnin tekniikoista perehdytään erityisesti säikeiden käyttöön. Luvussa käsitellään myös muualla tutkimuksessa käytettävää UML-notaatiota rinnakkaisten tehtävien kuvaamiseen.

Vaikka pelit ja pelimoottorit ovat viime vuosina nousseet lasten leikkikaluista hyväksytyksi akateemiseksi työkaluiksi, ei niiden tutkimus ole vielä edennyt kovin pitkälle. Esimerkiksi varsinaisten pelimoottorien arkkitehtuureja ei ole vieläkään tieteellisesti tutkittu, vaikka pelimoottoreja käytetäänkin jonkin verran tieteellisessä simuloinnissa (kts. Jacobson & Lewis, 2002). Pelimoottorit ja niiden toiminta ovat muutenkin vaikeasti lähestyttäviä aiheita, koska ne ovat liikesalaisuuksia. Tähän tutkimukseen sisältyykin lyhyt katsaus pelimoottoreiden rakenteeseen luvussa 3. Aihetta lähestytään esittelemällä aluksi pelimoottoreita yleisesti käyttäen lähteenä kirjallisuutta, ja tämän jälkeen tarkemmin käyttämällä lähteenä vapaasti saatavilla olevaa dokumentaatiota pelimoottoreiden komponenteista sekä Unreal 2 pelimoottorista.

Luvussa 4 käsitellään kirjallisuudesta löytyviä pelimoottorin rinnakkaistuksen malleja. Samassa luvussa esitellään myös tutkimuksen edistyessä kehitetty malli sekä varsinaisen vertailun ulkopuolelle jääviä rinnakkaistuksen ratkaisuja. Valitettavasti kirjallisuudesta löydettäviä malleja ei ole lähteissä esitetty erityisen yksityiskohtaisesti, joten analyysiin sisältyy myös mallien tarkentaminen tarpeellisilta osilta.

Luku 5 käsittää luvussa 4 esitettyjen pelimoottorin rinnakkaistuksen mallien vertailun suorituskyvyn ja laadullisten ominaisuuksien osalta sekä esittelee tarkemmin vertailuun käytettävää SAAM-menetelmää. Vertailun tulosten pohjalta pyritään muodostamaan johtopäätöksiä mallien soveltuvuudesta eri käyttötarkoituksiin. Luku 6 sisältää yhteenvedon tutkimuksen tavoitteista, menetelmistä ja tuloksista.

Tutkimuksen tuloksena on kolmen eri rinnakkaistusmallin esittely ja vertailu. Asynkronisen ja synkronisen tehtäväpohjaisen rinnakkaistuksen mallit perustuvat pelimoottorin eri tehtävien jakamiseen eri säikeisiin. Olioperustaisen rinnakkaistuksen malli pohjautuu eri olioiden jakamiseen eri säikeisiin. Mallien välillä löydettiin eroja erityisesti niiden skaalautuvuudessa eri prosessoriydinmäärille sekä niiden vaatimista muutoksista nykyisiin pelimoottorin komponentteihin. Kaikkien mallien havaittiin soveltuvan huonosti Playstation 3:n Cell-prosessorille, joskin tehtäväpohjaiset mallit saadaan prosessorilla toimimaan erikoisratkaisuilla. Suorituskyvyn puolesta kaikki mallit ovat yhdenvertaisia nykyisillä suoritinydinmäärillä, mutta tulevaisuudessa kymmenien ytimien suorittimilla olioperustainen malli on nopeampi paremman skaalautuvuutensa ansiosta.

# 2 RINNAKKAISKÄSITTELY

Tässä luvussa käsitellään rinnakkaisohjelmointia ja moniydinprosessoreita. Johdannossa esitellään lyhyesti käsite rinnakkaiskäsittely ja moniajo. Kappaleessa 2.2 perehdytään rinnakkaisohjelmointiin, huomio pidetään lähinnä rinnakkaisohjelmien suunnittelussa. Rinnakkaisohjelmoinnin tekniikoista perehdytään erityisesti säikeiden käyttöön. Kappaleessa käsitellään myös muualla tutkimuksessa käytettävää UML-notaatiota rinnakkaisten tehtävien kuvaamiseen. Tämän jälkeen kappaleessa 2.3 tutustutaan moniydinprosessoreihin, niiden mahdollisiin käyttökohteisiin sekä niille ohjelmistoja suunniteltaessa huomioon otettaviin seikkoihin.

## 2.1 Johdanto

Hazra (1995) määrittelee termin Parallel processing eli rinnakkaiskäsittely seuraavasti: ”Rinnakkaiskäsittely on tekniikka, joka korostaa laskennallisesta prosessista löytyvän samanaikaisuuden (rinnakkaisuuden) hyödyntämistä. Rinnakkaiskäsittely tarkoittaa useamman kuin yhden käskyn suorittamista yhtä aikaa, sen sijaan että käskyt suoritettaisiin yksi kerrallaan. Siksi rinnakkaiskäsittely edellyttää, että toiminnassa on useampi kuin yksi prosessi.”

Rinnakkaiskäsittelyssä on siis kyse useamman kuin yhden käskyn suorittamisesta yhtäaikaisesti. Kuitenkaan ei ole mahdollista suorittaa esimerkiksi saman ohjelman kahta peräkkäistä käskyä rinnakkain, koska edellinen käsky saattaa toimia seuraavan käskyn syötteenä. Rinnakkaiskäsittely onkin mahdollista vain, jos voidaan suorittaa rinnakkain useampaa *tehtävää* (task). Tehtävä tarkoittaa käsitteenä ajettavaa suorituksen polkua (Tanenbaum, 2001). Tehtävän käyttöjärjestelmätason alakäsite on *prosessi*, jota käsitellään kohdassa 2.2.3.

Jos käytössä ei ole kuin yksi suoritin, voidaan rinnakkaiskäsittelyä toteuttaa moniajon avulla. Moniajo (multitasking) tarkoittaa sitä, että rinnakkain suoritettavat tehtävät jakavat jonkin laitteistoresurssin, esimerkiksi prosessorin. Koska yksi prosessori

ei voi suorittaa useampaa kuin yhtä tehtävää yhtä aikaa, simuloidaan rinnakkaiskäsittelyä yksiprosessorisella järjestelmällä vaihtamalla suoritettavaa tehtävää hyvin nopeasti. Moniajo on mahdollista joko niin, että tehtävät luovuttavat itse kontrollin sopivan ajan välein seuraavalle tehtävälle, tai niin, että käyttöjärjestelmä jakaa suoritusaikaa tehtäville. Ensimmäisessä tapauksessa kyseessä on cooperative multitasking, joka on periaatteena ollut tunnettu ensimmäisten tietokoneiden ajoilta asti. Jälkimmäisessä tapauksessa kyseessä on preemptive multitasking. Se tarkoittaa moniajoa, jossa tehtävien ei itse tarvitse luovuttaa kontrollia toisille prosesseille, vaan käyttöjärjestelmän vuorotin (scheduler) huolehtii prosessoriajan jakamisesta. Jos käytössä on useampi suoritin, voidaan yhtäaikaisesti ajaa niin monta tehtävää kuin suorittimia on. Näin on menetelty jo ensimmäisten moniprosessoristen supertietokoneiden ajoista lähtien, mutta kuluttajille suunnitellut ratkaisut ovat vasta yleistymässä.

Joidenkin ohjelmien tarvitsee jakaa suorituksensa useampaan rinnakkain käsiteltävään osaan toimiakseen tyydyttävästi, esimerkkinä tästä palvelinohjelmistot. Osa ohjelmista puolestaan tarvitsee niin paljon suoritintehoa, että ne täytyy jakaa useammalle suorittimelle. Tästä esimerkkinä sääennustusten tekeminen. Ohjelmiston jakaminen useampaan rinnakkain suoritettavaan osaan on rinnakkaisohjelmointia, jota käsitellään seuraavassa kappaleessa. Ohjelma, jota ei ole rinnakkaistettu millään tavalla on *peräkkäisohjelma* (sequential program). Peräkkäisohjelma on sellainen ohjelma, joka suorittaa kaikki toimintonsa peräkkäisessä järjestyksessä, eli siinä ei ole siis rinnakkain suoritettavia osia.

## 2.2 Rinnakkaisohjelmointi

Rinnakkaisohjelmointi (concurrent programming tai multiprogramming) tarkoittaa Hansenin (1973) määritelmän mukaan niitä ohjelmointitekniikoita, joilla ohjataan rinnakkaisia prosesseja (tehtäviä). Hansenin näkemys on selvästi tekninen, ja jättää määritelmän ulkopuolelle tärkeän aiheen, rinnakkaisuuden suunnittelun. Akateeminen tutkimus on edelleen keskittynyt lähinnä uusien erityisesti rinnakkaisohjelmoin-

tiin soveltuvien ohjelmointikielten ja rinnakkaisuuden teorian ympärille. Tässä kohdassa pyritään tuomaan esille sellaisia asioita, mitä tarvitsee tietää rinnakkaisuudesta ohjelmistoa suunnitellessa.

### 2.2.1 Rinnakkaisohjelmiston suunnittelu

Rinnakkaissuoritettavaa ohjelmistoa voi suunnitella ja toteuttaa haluamallaan ohjelmistoprosessilla, mutta lisäksi täytyy huomioida tiettyjä rinnakkaisohjelmoinnin näkökulmia. Tässä kohdassa tarkastellaan sitä, mitä erityisiä tehtäviä sisältyy rinnakkaisohjelmointiin. Suuri osa asiaa koskevasta kirjallisuudesta käsittelee ongelmaa siitä näkökulmasta, että peräkkäisohjelma (sequential program) muutetaan rinnakkaisohjelmaksi (parallel program). Tämä on järkevä lähtökohta, koska harvoin ohjelmistoa suunnitellessa pystytään lähtemään puhtaalta pöydältä, vaan käytössä on yleensä ainakin aiempi versio, joka on usein toteutettu peräkkäisohjelmana.

Käsitellään asiaa yleisellä tasolla Fosterin (1995) mukaan. Foster esittää rinnakkaisohjelmoinnin tehtäviksi seuraavia: ongelman osittaminen rinnakkain ajettaviin osiin, rinnakkain ajettavien osien kommunikoinnin ja synkronoinnin suunnittelu sekä toisiinsa läheisesti liittyvien osien yhdistäminen. Fosterin listassa on myös ohjelmanosien asettaminen suoritettavaksi tietyillä suorittimilla, mutta se jätetään tästä esityksestä pois, koska se ei ole tarpeellista symmetrisillä moniprosessorijärjestelmillä. Se ei ole myöskään mahdollista kotitietokoneiden käyttöjärjestelmillä. Koska Foster ja monet muutkin aikalaisensa ovat keskittyneet lähinnä yksittäisten algoritmien rinnakkaisohjelmointiin, eivätkä kokonaisten ohjelmistojen rinnakkaisuuden suunnitteluun, on tähän listaan on syytä lisätä myös Barneyn (2005) esille tuoma ongelman ja ohjelman ymmärtäminen.

- Ongelman ja ohjelman ymmärtäminen. Barneyn mukaan ensin täytyy ymmärtää ositettavaa ongelmaa. Jos ollaan muuntamassa olemassa olevaa ohjelmistoa, täytyy pyrkiä ymmärtämään sen toimintaa. Ohjelmasta täytyy saada selville, mitä sen osia voitaisiin muuttaa rinnakkaissuoritettaviksi, ja mitä ei. Joitakin ongelmia ei yksinkertaisesti ole mahdollisia muuttaa rinnakkain ajettaviksi. Esimerkkinä Barney mainitsee algoritmin, joka laskee Fibonaccin lu-

vut. Koska seuraava luku riippuu aina edellisistä, ei ongelmaa voida rinnakkaistaa. Eräs tärkeä huomio rinnakkaissuoritettavia ohjelmanosia etsittäessä on, että koko rinnakkaisohjelmoinnin tarkoituksena on tehostaa ohjelman toimintaa. Ei siis ole järkeä muuttaa rinnakkaisiksi sellaisia osia, jotka eivät vaadi kovin paljoa suoritusaikaa.

- Ongelman osittaminen rinnakkain ajettaviin osiin. Foster olettaa, että käytössä on jokin selkeä ongelma tai algoritmi, joka ositetaan. Jos pyritään koko ohjelmiston kattavaan rinnakkaisohjelmointiin, täytyy tätä tehtävää ajatellen olla valmiina lista ositettavista ongelmista. Fosterin esittämä strategia on ensin jakaa ongelma hyvin pieniin osiin, ja myöhemmin yhdistää osia suorituskyvyn lisäämiseksi. Osittaminen voidaan Fosterin mukaan tehdä *tehtäväperustaisesti* (functional decomposition) tai *tietoperustaisesti* (domain decomposition), tai näiden yhdistelmänä. Tehtäväperustaisessa jakamisessa pyritään etsimään rinnakkaisia tehtäviä. Esimerkki tehtäväperustaisesta jaosta voi olla vaikkapa lentokentän hallintaohjelmisto, jossa yksi osa pitää huolen lentokoneiden reiteistä, ja toinen kentällä olevien matkustajien ohjaamisesta oikealle portille. Tietoperustainen jakaminen on ongelman jakamista rinnakkaisiin osiin, joille suoritetaan kaikille samat tehtävät. Lentokentän hallintaohjelmistoa esimerkkinä käytettäessä tällainen jako olisi toteutettavissa niin, että jokaista lentokonetta hoitaisi oma tehtävä.

- Rinnakkain ajettavien osien kommunikoinnin ja synkronoinnin suunnittelu. Eräs suurimmista teknisistä ongelmista rinnakkaisohjelmoinnissa on tehtävien välinen yhteistoiminta. Esimerkiksi edellistä lentokentän hallintaohjelmistoa ajatellen, täytyy jokaisen lentokoneen reittiä ohjaavan tehtävän kommunikoida muiden tehtävien kanssa – muuten lentokoneet saattaisivat törmätä. Säikeiden välisen kommunikoinnin eräs suurin ongelma on tehtävien välisen vuorovaikutuksen epädeterministisessä luonteessa, ei voida koskaan tietää missä tilassa ohjelmisto on milläkin ajan hetkellä, ja mahdollisia huomioon otettavia tiloja on hyvin paljon. Perustilanne on senkaltainen, että jos

tehtävät A ja B voivat tapahtua missä tahansa järjestyksessä, mitä seuraa jos A suoritetaan ennen B:tä, mitä päinvastaisesta tapauksesta seuraa, ja mitä seuraa jos ne suoritetaan yhtä aikaa osittain tai kokonaan.

- Osien yhdistäminen. Fosterin ideana oli jakaa tehtävät ensin hyvin pieniin osiin, ja tarpeen mukaan yhdistää osia peräkkäissuoritettaviksi isommiksi osiksi. Edelleen lentokentänhallintaohjelmisto-esimerkkiä jatkaen voitaisiin esimerkiksi yhdistää useampi lentokone samaan ohjausyksikköön. Osien määrän pitäisi Fosterin mukaan vastata järjestelmän prosessorien määrää, ja se voi olla mahdollista määrittää ajonaikaisesti. Yksi näkökulma osia yhdistettäessä on pyrkiä minimoimaan osien välinen kommunikaatio yhdistämällä osia, jotka ovat kiinteästi toisistaan riippuvaisia. Muita tehtävien ryhmittämisstrategioita on esittänyt Douglass (1998).

### 2.2.2 Tehtävien välinen viestintä

Rinnakkaisohjelmointiin on olemassa useita erilaisia malleja. Kaikissa peruskäsitteenä on tehtävä, joka toimii rinnakkain muiden suoritettavien tehtävien kanssa. Seuraavassa esitellään muutamia rinnakkaisohjelmointimalleja. Lähteenä on käytetty Barneyn (2005) tutoriaalia.

- Jaettu muisti. Rinnakkaisilla tehtävillä on jaettu muistiavaruus, jonka välityksellä ne viestivät keskenään. Jaetut muistioperaatiot sarjallistetaan esimerkiksi lukkojen tai semaforien avulla. Ongelmallisena pidetään sitä, että kaikki muisti on jaettua, jolloin muistioperaatioiden hallinta tulee vaikeaksi ymmärtää ja suunnitella.
- Säikeet. Säikeiden voidaan ajatella perustuvan jaetun muistin malliin. Säiemallissa pääohjelman ajatellaan haarautuvan useaan rinnakkain suoritettavaan alirutiiniin, jotka kommunikoivat yhteisen muistiavaruuden välityksellä. Jaetun muistin mallista poiketen säikeillä on lisäksi oma yksityinen muistialueensa. Säikeiden kesken jaetaan myös varatut laiteresurssit. Ajatellen yhdellä tietokoneella suoritettavia ohjelmia, säikeistys on näistä malleista yleisimmin käytössä oleva, ja siihen perehdytään tarkemmin kohdassa 2.2.4.

- Viestinvälitys (Message Passing). Rinnakkaiset tehtävät kommunikoivat keskenään jonkinlaisten viestien välityksellä, muutoin ne ovat eristettyjä toisistaan. Näin toimivat esimerkiksi eri tietokoneille hajautetut sovellukset, jotka välittävät viestejä verkkoyhteyden yli.
- Rinnakkaisdata (data parallel). Rinnakkain asetetaan tehtävien sijaan data, jolle suoritetaan tiettyjä määriteltyjä tehtäviä. Tämä vastaa edellä kuvattua tietoperustaista osittamisstrategiaa.

### 2.2.3 Käyttöjärjestelmän prosessit

Siirryttäessä lähemmäs toteutuksen tasoa täytyy ottaa käyttöön prosessin käsite. Nykyisissä käyttöjärjestelmissä puhutaan rinnakkain tehtävistä ja prosesseista. Prosessi mielletään usein käyttöjärjestelmätason käsitteeksi, ja tehtävää prosessin yläkäsitteeksi.

Tanenbaumin (2001) määritelmän mukaan prosessi (process) on ohjelman suoritettava ilmentymä, johon kuuluu:

- Muisti, joka sisältää ajettavan ohjelmakoodin ja ajonaikaista dataa.
- Prosessille annetut käyttöjärjestelmän resurssit, kuten tiedostokahvat.
- Tieto prosessin omistajasta ja suoritusoikeuksista
- Prosessin tila ja suoritus, esimerkiksi prosessin rekisterien arvot ja muistin fyysiset osoitteet.

Prosessista erotetaan siis suoritus ja resurssit. Nykyisissä käyttöjärjestelmissä prosessit ovat suojattuja toisiltaan, eli ne eivät jaa muistiresursseja, ja niiden välinen kommunikointi on vaikeaa. Tästä syystä pelkästään prosessien avulla ei esimerkiksi yksittäinen ohjelma pysty osittamaan työtaakkaansa kovin tehokkaasti, vaan tarvitaan säikeen käsite.

### 2.2.4 Säikeet

Jokaisella prosessilla on vähintään yksi säie. Säie on se osa prosessista, joka voidaan asettaa suoritettavaksi. Käytännössä säikeet ovat saman ohjelman sisällä olevia rinnakkain suoritettavia ohjelmanosia, jotka pääsevät käsiksi yhteiseen muistiavaruuteen. Nykyaikaisissa käyttöjärjestelmissä prosessilla voi olla useita säikeitä. Useamman kuin yhden säikeen prosesseja kutsutaan *monisäikeisiksi* (multithreaded). (Tanenbaum, 2001)

Jokaisella säikeellä on oma laskuri, joka kertoo mikä on seuraava suoritettava käsky. Säie pitää myös yllä tietoa prosessorin rekistereistä, jotta se voi jatkaa toimintaansa suoritusvuoron saatuaan. Tanenbaum (2001, 81) sanoo, että säikeet ovat prosesseille sama kuin prosessit ovat yhdelle tietokoneelle. Säikeet jakavat keskenään muistiavaruuden ja tiedostokahvat siinä missä prosessit jakavat keskenään levyt, tulostimet ja muut korkean tason resurssit. Säikeitä kutsutaankin joskus kevyiksi prosesseiksi, ja moniajon ja rinnakkaissuorituksen käsitteet toimivat niille samoin kuin prosesseille.

Nykyisistä valtakielistä esimerkiksi Java tukee säikeiden käyttöä jo standardinsa puolesta. Valitettavasti näin ei ole asian laita kaikkien kielien suhteen. Säikeiden käytön nykytilannetta esimerkiksi C++-kielessä kuvaa se, että säikeet eivät kuulu ANSI C++ standardiin. Säikeiden käyttäminen C++:lla on niin monimutkaista, että siitä ei ilmeisesti ole päästy yksimielisyyteen edes standardisointikomiteassa. Lähimpänä jonkinlaista teollisuuden nykystandardia on OpenMP. OpenMP on monialustaiseen rinnakkaisohjelmointiin tarkoitettu rajapinta, joka ansioituu lähinnä jaetun muistialueen hallinnassa. OpenMP ei sisällä tehtävien rinnakkaistamiseen minkäänlaisia rakenteita. OpenMP Architecture Review Board (2005) avoimesti tunnustaa, että tehtävien rinnakkaistaminen tullaan sisällyttämään standardiin kunhan asiasta päästään yksimielisyyteen. Yhteisen standardin puutteesta johtuen säikeiden käyttö tekee C++-lähdekoodista vaikeaa siirtää laitealustojen ja kääntäjäympäristöjen välillä.

### 2.2.5 Rinnakkaisuuden hyödyt

Tanenbaum (2001, 85-90) luettelee syitä säikeiden käyttöön ohjelmoinnissa:

- Tärkein syy säikeiden käyttöön on mahdollistaa ohjelmanosien samanaikainen suoritus. Aina ei ole mahdollista tai tehokasta tehdä esimerkiksi jotain toimintoa aliohjelmakutsuna, vaan se voidaan suorittaa omana säikeenään. Tyypillinen tapaus on esimerkiksi ohjelma, joka suorittaa jotain hidasta laskentatoimintoa, ja haluaa näyttää käyttäjälle edistymistä kuvaavan kuvaajan. Ei ole järkevää välillä keskeyttää laskentaa kuvaajan esitystä varten, vaan kuvaajan päivitys voidaan tehdä omassa säikeessään.
- Säikeitä on nopeaa luoda verrattuna esimerkiksi kokonaisiin prosesseihin.
- Säikeiden käytöllä voidaan nopeuttaa suoritusta, jos ohjelma joutuu jatkuvasti odottamaan esimerkiksi jotain laiteresurssia. Ohjelma voi asettaa säikeen odottamaan laitetta, ja jatkaa muuten toimintaansa.
- Monisäikeinen ohjelma toimii tehokkaammin moniprosessorisella tietokoneella kuin yksisäikeinen.

Amdahl (1967) on esittänyt yksinkertaisen laskukaavan, jonka avulla voidaan laskea algoritmin tehonlisäys, jos osa siitä muutetaan rinnakkain suoritettavaksi. Alun perin Amdahl käytti kaavaansa argumenttina sen puolesta, että pelkästään prosessoreja lisäämällä ei voida algoritmin suoritusta nopeuttaa loputtomasti, vaan algoritmi asettaa suorituskyvyn lisäykselle nopeasti saavutettavan ylärajan. Sittemmin kaavan oikeellisuutta on kritisoinut voimakkaasti Gustafson (1988), mutta sen on todistanut toimivaksi Krishnaprasad (2001). Kaava 1 esittää Amdahlin lakia formuloituna moniprosessorijärjestelmille.

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

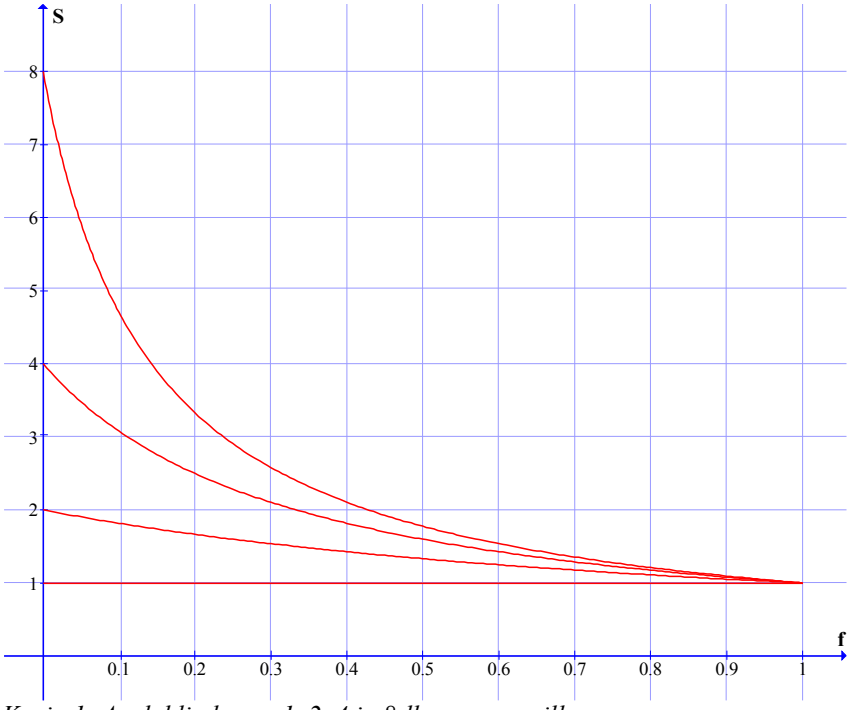
*Kaava 1: Amdahlin kaava rinnakkaisalgoritmin tehon laskentaan*

S = Rinnakkain suoritettavan algoritmin nopeuden suhde peräkkäin suoritettavaan algoritmiin

f = Kuinka suurta osaa algoritmista ei voida suorittaa rinnakkain

N = prosessorien määrä

Esimerkiksi jos kaavassa f, eli algoritmin peräkkäin suoritettava osa, lähestyy nollaa, saadaan S lähestyy N:ää. Tämä tarkoittaisi, että algoritmi nopeutuu suoraan siinä suhteessa miten monta prosessoria on käytössä. Kaava siis olettaa, että algoritmi on jaettu yhtä moneen rinnakkain suoritettavaan osaan kuin on prosessoreja. Kuvio 1 havainnollistaa kaavan antamia arvoja N:n arvoilla 1, 2, 4 ja 8.



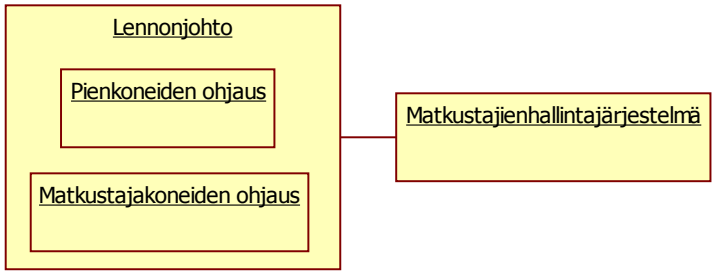

*Kuvio 1: Amdahlin kaava 1, 2, 4 ja 8:lla prosessorilla*

### 2.2.6 Rinnakkaisohjelmoinnin kuvaukset

Rinnakkaistettavien ohjelmistojen suunnittelussa tärkeä tekijä on suunnitelmien esitykseen käytettävä visuaalinen malli, josta voidaan hahmottaa rinnakkaisten ohjelmanosien konfiguraatio ja toiminta. Tässä kappaleessa esitetään yksi mahdollisuus kuvata rinnakkaisohjelmia. Koska UML on nykyään standardin asemassa oleva mallinnuskieli, on järkevää käyttää sitä myös rinnakkaisohjelmoinnin rakenteiden kuvaamiseen. Tässä kappaleessa tutustutaan siihen, mitä UML:n kaavioita voidaan esimerkiksi käyttää suunniteltaessa rinnakkaissuoritettavia ohjelmia. Tarkoituksena ei ole toimia UML-oppaana, vaan esitellä miten rinnakkaisohjelmoinnin erityistapauksia voidaan mallintaa.

Eräs saatavilla oleva UML-laajennos, joka on osittain suunniteltu rinnakkaisohjelmien kuvaamiseen, on Performance Prophet -projektin kehittelemä laajennos (Fahringer & Pllana, 2002). Laajennoksen kaikkia käsitteitä on kuitenkin vaikea ymmärtää, koska siinä on myös hyvin paljon erityisnotaatiota muihin tarkoituksiin. Tästä syystä sitä ei esitellä tarkemmin.

Yksi mahdollisuus mallintaa rinnakkaisuutta UML:n avulla on käyttää aktiivisia olioita (active object) oliokaaviossa. UML jakaa oliot aktiivisiin ja passiivisiin. Aktiivisien olioiden ajatellaan sisältävän jonkinlaisen säikeen, ja ne voivat toimia yhtäaikaisesti sekä kutsua passiivisten olioiden operaatioita. UML:n oliolähtöinen ajattelutapa sisältää rinnakkaiskäsittelyn implisiittisesti. Douglass (1998) kutsuu sellaista kaaviota, jossa on näkyvissä vain aktiivisia olioita, nimellä System task diagram. Aktiivisten olioiden sisään kuvataan niiden kontrolloimat passiiviset oliot. Tämä kaavio kuvaa suoraan järjestelmän säikeet ja niiden suhteet toisiinsa. Kuvio 2 esittää esimerkkitilannetta lentokentän hallintajärjestelmästä. Kuviossa aktiivisia olioita, jotka siis sisältävät säikeen, ovat lennonjohto ja matkustajien hallintajärjestelmä. Lennonjohto käyttää passiivisia olioita pienkoneiden ohjaus ja matkustajakoneiden ohjaus. Lennonjohdon ja matkustajien hallintajärjestelmän täytyy kommunikoida keskenään, jotta lentokoneet ja matkustajat osuvat samalle portille lentokentällä.

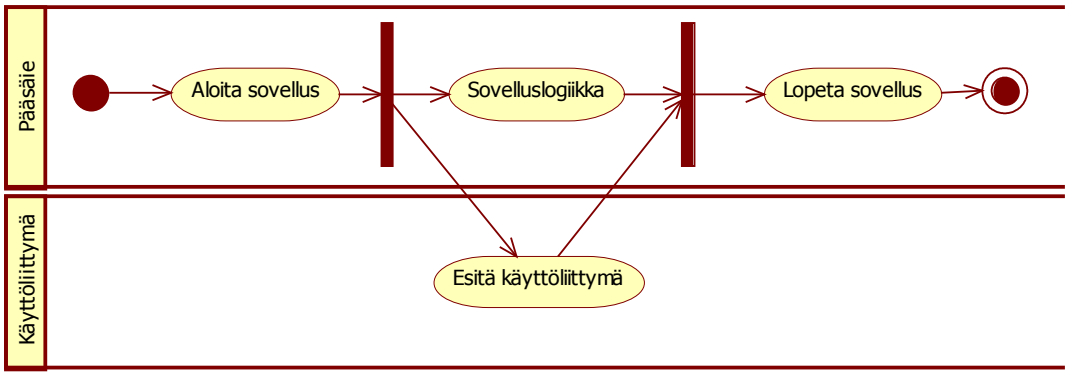



*Kuvio 2: Aktiiviset ja passiiviset oliot*

Luokka- ja oliokaavioiden avulla ei kuitenkaan voi esittää tarkemmin toimintojen suoritusta, vaan siihen täytyy käyttää muita kaavioita. Rinnakkaisten tehtävien mallintamiseen sopii hyvin aktiviteettikaavio. Aktiviteettikaavion käytöstä rinnakkaisohjelmointiin ei ole olemassa hyvää lähdettä, joten tässä kohdassa olevat ideat on tulkittu suoraan UML spesifikaation versiosta 2. Kuviossa 3 on kuvattu tilanne, jossa kuvitteellisen algoritmin suoritus haarautuu (fork) kahteen rinnakkaiseen tehtävään, ja lopussa molempien tehtävien tulokset yhdistetään (join) palautettavaksi. Haarautuminen tarkoittaa siis uuden säikeen luomista, mutta se ei ota kantaa, jatkuuko entisen säikeen toiminta. Säikeen päättymistä merkitään yhdisteellä, mutta sekään ei ota kantaa, päättyvätkö kaikki siihen tulevat säikeet ja uusi alkaa, vai jatkuuko yhdisteeseen tulevista säikeistä yksi muiden päättyessä. UML versiossa 2 on otettu huomioon myös se, että jokin tehtävä saattaa päättyä ilman, että sen tuloksia yhdistetään rinnakkaisten tehtävien kanssa. Aiemmin määrittelyn mukaan jokaista haarautumaa tuli myös vastata yksi yhdiste. On huomattava, että kuviossa ei oteta kantaa siihen, kuinka tieto välitetään kahdelta algoritmia suorittavalta säikeeltä tulosten yhdistämiseksi.

*Kuvio 3: Rinnakkaiset tehtävät UML:n aktiviteettikaaviona*

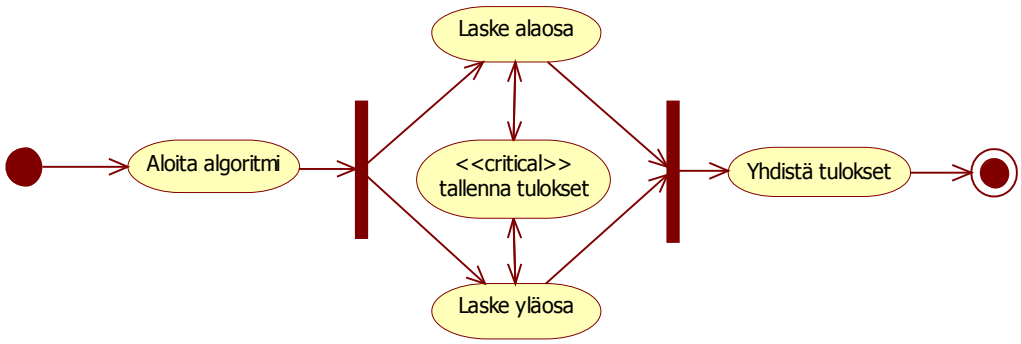
Jos halutaan erottaa ohjelman säikeet toisistaan paremmin, on aktiviteettikaaviossa mahdollista käyttää jakajia (partition). Kuviossa 4 esitetään sovelluksen elinkaari kahtena säikeenä. Kuviosta ei yksinkertaisuuden vuoksi käy ilmi esimerkiksi käyttöliittymän ja sovelluslogiikan vuorovaikutus, mutta myös tämä voitaisiin kuvata kaavion avulla.




*Kuvio 4: Säikeitä voidaan erotella UML:n jakajien avulla*

Tiedon välittämiseen säikeiden välillä tarvitaan jaettu muistialue. Jaetun muistialueen esittämistä varten Fahringer & Pllana (2002) ovat esittäneet mielenkiintoisen idean aktiviteettikaavion jaettujen aliaktiviteettien käyttämisestä. Seuraavassa esityksessä on hiukan yksinkertaistettu heidän malliaan. Jaettua muistialuetta voidaan ajatella esimerkiksi hallittavan käyttämällä kaikille rinnakkaisille tehtäville yhteistä sarjallistuvaa aktiviteettia. UML:n avulla tämän aktiviteetin sarjallistuvuutta ei voi suoraan kuvata, mutta sitä varten voidaan luoda uusi stereotyyppi, <<critical>>. Critical tarkoittaa kriittistä koodialuetta, ja tällä stereotyypillä merkittyyn aktiviteettiin pääsee

käsiksi vain yksi säie kerrallaan. Kuviossa 5 on sama algoritmi kuin kuviossa 3, mutta nyt on käytetty kriittistä aktiviteettia ”tallenna tulokset”, jonka avulla algoritmin osasuoritusten tulokset välitetään tulosten yhdistämiseksi.




*Kuvio 5: Kriittisen koodialueen esittäminen UML:n stereotyyppinä*

### 2.2.7 Ongelmat rinnakkaisohjelmoinnissa

Tutkimuksessaan aivojen moniajosta Jiang, Saxe & Kanwisher (2004) tulivat tulokseen, että ihmisaivot eivät pysty rinnakkaissuoritukseen kovin helposti. Aivot vaikuttavat asettavan ajatuksia jonoon, ja suorittavat niitä nopeasti vuorotellen vähän samaan tapaan kuin tietokoneet. Aivojen toiminta solutasolla perustuu nimenomaan hermosolujen rinnakkaiseen toimintaan, mutta rinnakkaisuus ei heijastu ajatusten tasolle - kahta asiaa on vaikeaa ajatella yhtä aikaa. Tästä ei tietenkään voida vetää yhteyksiä rinnakkaisohjelmoinnin haastavuuteen, mutta se antaa mielenkiintoisen näkökulman asiaan – eräs rinnakkaisohjelmointia rajoittava tekijä voi olla oma rajallinen kapasiteettimme. Asiaa käsittelevässä kirjallisuudessa pääasiallisiksi ongelmiksi tunnistetaan kuitenkin työkalujen puutteellisuus sekä rinnakkaisohjelmoinnin tekniikoiden ja kielien kehittymättömyys.

Sutter ja Larus (2005) kuvaavat tilannetta seuraavasti. Tähän mennessä säikeitä on käytetty menestyksekkäästi palvelinohjelmistoissa, sillä nämä ovat luonnostaan rinnakkaistuvia ongelmia. Palvelin voidaan yksinkertaisesti laittaa palvelemaan jokaista

asiakasta omassa säikeessään, eikä juuri muuta tarvitse tehdä. Palvelimen säikeet ovat yleensä melkein itsenäisiä, niiden ei tarvitse tietää toistensa toiminnasta, tai tehdä yhteistyötä muiden säikeiden kanssa. Asiakasohjelmiston (client program) säikeistäminen on huomattavasti ongelmallisempaa, sillä asiakasohjelmisto ei välttämättä suorita mitään suuria laskentatoimintoja, jotka olisivat helposti ositettavissa. Asiakasohjelmisto täytyykin useimmissa tapauksissa osittaa esimerkiksi jakamalla esimerkiksi käyttöliittymä ja ohjelman toiminta omiksi osikseen. Nämä osat toimivat yhdessä ja jakavat tietoa keskenään hyvin monilla ja monimutkaisilla tavoilla.

Teknisellä tasolla rinnakkaisohjelmoinnin ongelmat liittynevät lähinnä siihen, että käytössä olevat ohjelmointimenetelmät ovat vanhentuneita. Vertailtaessa edellä esitettyjä monisäikeisen ohjelmoinnin menetelmiä Hansenin (1973) näkemyksiin 1960-luvun rinnakkaisohjelmointimallien kehityksestä, voidaan löytää hyvin paljon yhtäläisyyksiä. Hansenin esityksestä löytyvät kriittiset alueet (critical section) ja semaforit on alun perin tarkoitettu yksittäisten algoritmien rinnakkaisohjelmointiin, ei kokonaisten monimutkaisten ohjelmistojen rinnakkaisohjelmointiin.

Käytäntö on osoittanut, että monisäikeisen ohjelman vianmääritys on hyvin vaikeaa nykyisillä työkaluilla, sillä säikeet eivät aina toimi samassa järjestyksessä. Esimerkiksi suorituksen pysäyttäminen mahdollisen virheen kohdalla ei säikeisessä ohjelmassa paljasta sitä, mikä johti virheelliseen arvoon, sillä arvoa on saattanut muuttaa jokin muu säie. Virheet ovat tyypillisesti myös satunnaisesti ilmeneviä.

Ratkaisuna rinnakkaisohjelmoinnin monimutkaisuuteen on vuosikymmenien ajan kehitetty erilaisia rinnakkaisohjelmointiin erityisesti tarkoitettuja ohjelmointikieliä ja tekniikoita. Yksi lupaavimmista tekniikoista on jo yli vuosikymmenen ajan suunniteltu transactional memory. Kyseessä on ratkaisu siihen ongelmaan, että jos kaksi säiettä käyttää yhteistä muistia, toinen voi lukea sitä yhtä aikaa kuin toinen kirjoittaa, jolloin lukija saa virheellistä tietoa, tai molemmat voivat yrittää kirjoittaa muistiin yhtä aikaa, jolloin muistiin tallettuu virheellistä tietoa. Herlihyn ja Mossin (1993) ajatuksena oli, että ohjelmoijat voivat määritellä sopivaksi katsomansa muistialueet atomisiksi, jolloin laitteisto pitää huolen siitä, että vain yksi prosessi pääsee käsiksi

tietoon kerrallaan. Tämä toteutetaan hiukan samaan tapaan kuin tietokannoissa. Prosessit saavat lukea ja kirjoittaa muistialueille vapaasti, ja jos ne sattuvat toimimaan samalla alueella, tapahtumalokin avulla tarkistetaan lopputuloksen oikeellisuus. Jos oikeellisuus ei täyty, laitteisto hoitaa automaattisesti muistitapahtumien uudelleenajon.

Eräs mielenkiintoinen kehitysaskel rinnakkaisohjelmoinnin helpottamiseksi on kääntäjän tekemä automaattinen rinnakkaistus (auto-parallelization), joka on mukana Intelin C++ -kääntäjän versiossa 9 erillisenä optiona. Intelin mukaan kääntäjä osaa muuttaa kiinteän kokoiset silmukat säikeiksi, esimerkiksi silmukka ”for(int i=0;i<100;i++)” muutettaisiin kahdeksi rinnakkaiseksi säikeeksi, joista toisessa laskettaisiin silmukan arvot 0-49, ja toisessa arvot 50-99. Automatiikka on kuitenkin rajoittunut vain sellaisiin silmukoihin, joiden koko on kääntäjän nähtävissä. Suorituskykymittauksia ei valitettavasti ollut saatavilla tätä kirjoitettaessa, mutta voidaan helposti kuvitella, miksi tämänkaltainen toiminta ei tuota tyydyttäviä tuloksia. Jos esimerkiksi silmukan seuraavan kierroksen toiminta riippuu edellisen kierroksen toiminnasta, tuottaa rinnakkaistus tässä tilanteessa heti virheellisiä arvoja.

Eräs mahdollinen syy siihen, miksi uudet ohjelmointikielet eivät ole vielä tuoneet merkittävää muutosta rinnakkaisohjelmointiin on siinä, että ohjelmoijat eivät mielellään halua opetella uusia kieliä jo osaamiensa lisäksi. Monissa rinnakkaisohjelmoinnin tekniikoita edistävässä projekteissa pyritäänkin nykyään kehittämään laajennoksia muun muassa C ja C++ -kieliin. Valtakielistä Java on ollut edelläkävijän roolissa monisäikeistyksen suhteen, sillä Javan standardiin monisäikeistys on kuulunut jo pitkään. Uusien tekniikoiden yleistyminen on kuitenkin ollut tähän mennessä hidasta, ehkä osaksi siitä johtuen, että rinnakkaisohjelmointia on joutunut harrastamaan vain pieni joukko ohjelmoijia. Todennäköisesti tilanne tulee kuitenkin muuttumaan, kun moniydinprosessorit yleistyvät. Moniydinprosessoreihin perehdytään seuraavassa kappaleessa.

## 2.3 Moniydinprosessorit

C++ -standardointikomitean johtaja Herb Sutter (2005) on kirjoittanut artikkelin siitä, miksi Intel ja AMD ovat siirtymässä valmistamaan moniydinprosessoreja. Seuraavassa on johdannoksi lyhyt tiivistelmä tuosta artikkelista. Samoja ajatuksia ovat esittäneet myös moniydintekniikan pioneerit Olukotun & Hammond (2005).

Tähän mennessä prosessorinvalmistajat ovat pystyneet nostamaan prosessorien suorituskykyä nostamalla kellotaajuutta, optimoimalla suoritusjärjestystä sekä kasvattamalla prosessorin välimuistia. Ohjelmien käyttö on nopeutunut suoraan prosessorin tehoa nostamalla, eikä ohjelmoijien ole tarvinnut tehdä asian hyväksi juuri mitään. Prosessorien tehoa kasvattaessa aletaan kuitenkin törmäämään fyysisiin esteisiin. Kellotaajuuden nostamista vaikeuttavat yhä kasvava lämmöntuotto, lisääntyvä virrankulutus ja prosessorin sisäiset jännitevuodot. Käytännössä kellotaajuuksia ei ole juuri voitu nostaa vuoden 2003 jälkeen. Välimuistin kasvatus on edelleen mahdollista, mutta lisääntyneestä transistorimääristä johtuen prosessorien viivanleveyttä joudutaan jatkuvasti pienentämään, jotta ydin ei kasvaisi liian suureksi. Viivanleveyden pienentämisellä on rajansa, sillä kovin pieniä johtimia pitkin sähkö ei enää pysty kulkemaan.

Olukotun & Hammond (2005) korostavat erityisesti lisääntynyttä sähkönkulutusta, joka on noussut yhdestä watista yli sataan wattiin pelkästään prosessorin osalta, samalla kun jäähdytysratkaisut eivät ole kehittyneet yhtä nopeasti. He myös huomauttavat, että tähän mennessä tehty käskyjen suoritusjärjestyksen optimointi on tosiasiassa tapahtunut kehittämällä menetelmiä, joilla käskyjä on voitu suorittaa rinnakkain saman ytimen sisällä entistä tehokkaammin. Tämän menetelmän rajat ovat tulleet vastaan, koska järjestyksen optimointiin käytettävä logiikka vie jatkuvasti yhä suuremman osan prosessorin transistoreista.

Koska vanhat menetelmät eivät enää sovellu prosessoritehon kasvattamiseen, Sutter (2005) ennustaa, että prosessorien tehonlisäys tulee lähivuosina tulemaan moniytimisyydestä, hypersäikeistyksestä (hyperthreading) ja välimuistin kasvattamisesta. Hy-

persäikeistys tarkoittaa, että yhdellä prosessoriytimellä voidaan suorittaa joitakin käskyjä rinnakkaisista säikeistä yhtäaikaisesti. Moniytimisyys tarkoittaa, että samassa prosessorissa on rinnakkaisia ytimiä, joilla voidaan jokaisella suorittaa omaa säiettään.

Sutterin mukaan jatkossa yksisäikeiset ohjelmistot eivät enää siis hyödy prosessorien nopeutumisesta ilman muutoksia, ja hän ennustaakin monisäikeisestä ohjelmoinnista yhtä suurta ajatusmaailman muutosta kuin aikanaan oliomallista. Onhan tähän mennessä suurin osa ohjelmoijista on nimittäin pystynyt välttämään täysin kaikenlaisen rinnakkaisohjelmoinnin.

### 2.3.1 PC- ja konsoliprosessorien moniydintekniikat

Tekninen termi moniytimisyydelle on CMP, Chip Multiprocessor. AMD:n ja Intelin vuonna 2004 esittelemien PC-prosessoreiden moniydintekniikka kulkee nimellä SMP, Symmetric Multiprocessing. SMP:ssä sirulle yhdistetyt prosessorit ovat kaikki samanlaisia, eikä niillä ole erikoistehtäviä (McDougall, 2005). Käytännössä SMP-prosessorit näkyvät käyttöjärjestelmälle useampana erillisenä prosessorina.

Uusien pelikonsoleiden prosessoreista tarkempaa tietoa on olemassa Microsoftin Xbox 360:sta ja Sonyn Playstation 3:sta. Tätä kirjoittaessa Nintendo ei ole vielä tiedottanut Revolution-konsolinsa teknisistä yksityiskohdista, mutta ilmeisesti se ei tule käyttämään moniydintekniikkaa. Sekä Xboxin että Playstationin prosessoreiden valmistamisesta vastaa IBM, ja ne perustuvat IBM:n Power-arkkitehtuuriin. Prosessorit ovat kuitenkin hyvin erilaisia. Artikkelissaan Stokes (2005) kertoo, että Xbox-konsolin Xenon-prosessori sisältää kolme samanlaista ydintä, joiden tekninen toteutus muistuttaa PC-prosessoreiden toteutusta. Seebachin (2005) mukaan Playstationin Cell-prosessori puolestaan sisältää yhden pääasiallisen ytimen, ja seitsemän pienempää suoritusyksikköä, joita kutsutaan nimellä SPE (Synergistic Processing Element). SPE-yksiköiden käyttö edellyttää erilaisia ratkaisuja kuin PC-prosessoreita tai Xenon-prosessoria hyödyntäessä. SPE-yksikkö ei pääse käsiksi ohjelman yhteiseen muistialueeseen, vaan sille täytyy erityisesti siirtää sen käsiteltäväksi tarkoitettu data ja ohjelmakäskyt. Lisäksi yksiköiden suorituskyky ja jopa liukulukulaskennan tark-

kuus on pienempi kuin itse pääytimen. Niinpä SPE-yksiköiden käyttäminen eroaa merkittävästi tavallisten säikeiden käytöstä, sillä ei voida enää olettaa, että kaikilla säikeillä on käytettävissä samat resurssit. Epäsymmetristä mallia voisi helposti kritisoida sen hyödyntämisen vaikeudesta. Epäsymmetrisyydellä on kuitenkin hyvätkin puolensa. Pienien prosessointiyksiköiden lisääminen on kustannustehokasta, ja niitä mahtuu samalle alalle enemmän kuin täysikokoisia symmetrisiä yksiköitä. Lisäksi epäsymmetriset yksiköt voidaan optimoida paremmin vastaamaan jotakin tiettyä laskennan tarvetta. Balakrishnan et al. (2005) myös huomauttavat, että ohjelmissa on väistämättä enemmän tai vähemmän pitkiä osia, jotka täytyy suorittaa peräkkäin. Tällöin on eduksi, että jokin prosessoriydin pystyy suoriutumaan näiden läpikäymisestä nopeammin kuin muut ytimet.

Stokesin mukaan kummassakaan konsoliprosessorissa ei ole PC-prosessoreiden tapaan käskytason rinnakkaistusta (Instruction Level Parallelism), joka mahdollistaa käskytasolla esimerkiksi ehtolauseiden nopeamman suorituksen. Tämän seurauksena molemmat konsoliprosessorit suoriutuvat merkittävästi heikommin ehtolauseita sisältävästä ohjelmakoodista, kuten esimerkiksi tekoälyn laskennasta. Tuloksena on, että sovellusten kehittäminen monialustaisesti PC- ja konsolilaitteille edellyttää eri rajoitusten huomioon ottamista.

### 2.3.2 Hypersäikeistys

Eräs ero Intelin ja AMD:n prosessoreiden välillä on Intelin hypersäikeistys, joka on periaatteena tunnettu 1950-luvulta saakka. Eggers et al. (1997) kutsuvat tätä nimellä SMT, Simultaneous Multithreading, ja mielenkiintoisesti he esittävät tekniikkaa tavallaan uutuutena. Seuraava kuvaus noudattaa heidän esitystään: SMT-tekniikan ajatuksena on vähentää prosessorin tyhjäkäyntiä suorittamalla vuorotellen useampaa säiettä. Kun prosessori suorittaa jonkin hitaan toiminnon, kuten muistihaun, se joutuu usein odottamaan jonkin aikaa haun tulosta ennen kuin voi jatkaa. SMT-prosessori voi haun valmistumista odottaessaan ottaa suoritukseen jonkin toisen säikeen. Prosessori voi myös ottaa suoritukseen samasta säikeestä jonkin sopivan käskyn. Hypersäikeistystä käytetään siis yhden ytimen suorituksen tehostamiseen. Hypersäikeistävä prosessoriydin näkyy käyttöjärjestelmälle kahtena ytimenä, jolloin esimerkiksi Inte-

lin D-mallin prosessorit näkyvät käyttöjärjestelmälle neljänä prosessorina. Ilmeisesti Intel on kuitenkin luopumassa hypersäikeistystä, koska sen hyödyt eivät ole verrattavissa todelliseen moniytimisyyteen. Uudet konsoliprosessorit Xenon ja Cell kuitenkin toimivat hypersäikeistettyinä.

### 2.3.3 Moniydintekniikan käyttökohteet

Luultavasti merkittävimmät markkinat moniytimisillä prosessoreilla ovat nyt palvelinratkaisuissa, joissa normaalisti käytetään moniprosessorijärjestelmiä. Moniytimisyys voi suoraan moninkertaistaa loogisten prosessorien määrän palvelimessa, mikä merkitsee säästöä laitepaikoissa ja sähkönkulutuksessa. Palvelinkäytössä yksittäisen prosessorin nopeus ei ole niin tärkeää kuin rinnakkaisten säikeiden määrä. Tätä ajatusta hyödyntää Sunin Niagara, joka perustuu hyvin monen heikkotehoisen ytimen yhdistämiseen. (Olukotun & Hammond, 2005).

Sutter (2005) mainitsee yhtenä nykyisenä sovellusalueena ohjelmointikielten kääntäjät, jotka ovat luonnostaan monisäikeistyviä. Ohjelmistoyrityksillä ei välttämättä ole mahdollisuutta hankkia ohjelmistokehittäjilleen erillisiä moniprosessorityöasemia ohjelmien kääntämiseen, mutta moniydinprosessorilla varustettu työasema saattaa hyvin riittää tähän tarkoitukseen.

AMD kertoo markkinointipapereissaan moniytimisyyden olevan keino parantaa vasteaikoja suoritettaessa useita yhtäaikaisia sovelluksia työpöytäkäytössä. Sovellusten nopeutuminen listataan vasta tulevaisuudessa tapahtuvaksi muutokseksi, ja todennäköisiä sovellusalueita ovat heidän mukaansa puheen tunnistaminen ja tekoäly sekä vuosikymmeniä odotettu digitaalinen koti. Vaikka moniytimisyyttä ei vielä aggressiivisesti markkinoida tavallisille kuluttajille, on kuitenkin luultavaa, että näin tullaan toimimaan tulevaisuudessa. Onkin sitten kokonaan eri asia, kuinka paljon suorituskykyä puheen tunnistaminen ja digitaalinen koti tarvitsevat, ja eikö nykyisten järjestelmien suorituskyky riitä niihin.

AMD:n mukaan myös nykyisten sovellusten käyttö saattaa nopeutua, vaikka niitä ei olisikaan suunniteltu monisäikeisiksi. Tämä on mahdollista, koska käyttöjärjestelmissä on usein toiminnassa useita rinnakkaisia prosesseja, ja niiden suorittaminen eri suoritinytimillä vapauttaa yksittäisen prosessin käyttöön enemmän suoritinkapasiteettia. On kuitenkin selvää, että moniytimisyydestä ei ole muutoin hyötyä yksittäisen sovelluksen nopeudelle, ellei sovellusta suunnitella tukemaan rinnakkaistusta alusta pitäen.

Kotikäyttäjille moniytimisyydestä ei ehkä ole vielä tarpeeksi suurta hyötyä kuluihin nähden. Monet käyttöjärjestelmät eivät vielä täysin sisäisesti ole optimoitu moniytimisyydelle, FreeBSD:llä työ on vielä kesken (McDougall, 2005), Windows XP ei suorituskykynsä suhteen hyödy moniytimisyydestä lainkaan. Microsoft ilmoittaa uuden Windows Vistan hyötyvän toiminnassaan moniytimisistä prosessoreista. Toisaalta Linux on tukenut moniytimisyyttä erinomaisesti jo ytimen versiosta 2.4 lähtien (McDougall, 2005). Sutter ja Larus (2005) jopa ennustavat, että vanhat yksisäikeiset ohjelmat tulevat tulevaisuudessa toimimaan hitaammin, kun prosessoriytimiä yksinkertaistetaan, jotta niitä mahtuisi enemmän samalle sirulle.

### 2.3.4 Rinnakkaisohjelmointi moniydinprosessorilla

Tässä kappaleessa tarkastellaan sitä, mitä erityisiä asioita tulisi ottaa huomioon ohjelmoitaessa monisäikeistä ohjelmaa moniydinprosessorille. Aiheen tuoreudesta johtuen siitä ei ole kirjoitettu vielä kovin paljon, ovathan ensimmäiset kuluttajille suunnatut PC-laitteiden moniydinprosessorit vasta tulossa markkinoille. Vanhemmissa moniydinprosessorijärjestelmissä puolestaan on keskitytty lähinnä erityisiin ongelma-alueisiin, kuten palvelinjärjestelmiin, ja niitä koskeva aineisto ei helposti sovi yleistettäväksi.

Erityisesti Intelin markkinointiaineistosta voi saada kuvan, että kaikissa moniydinprosessoreissa on ytimien kesken jaettu L2-tason välimuisti. Tavallisessa moniprosessorijärjestelmässä säikeiden välinen kommunikointi täytyy hoitaa hitaan systeemiväylän (system bus) kautta, jolloin ohjelmistot on suunniteltava niin, että niiden säikeet eivät ole joudu kommunikoimaan toistensa kanssa kovin paljoa. Jaetun väli-

muistin ansiosta tiedon välittäminen säikeiden välillä on huomattavasti nopeampaa, kuitenkin sillä kustannuksella, että prosessoriytimet joutuvat vuorottelemaan välimuistin käytössä. Olukotun & Hammond (2005) kertovat, että tämän takia ohjelmoijien ei tulevaisuudessa tarvitse välittää erityisen paljon säikeiden itsenäisyydestä, vaan lähinnä synkronoinnin toimivuudesta. Välimuistin arkkitehtuurissa on kuitenkin mallikohtaisia eroja sekä Intelin että AMD:n prosessoreissa; osassa L2-tason välimuisti on nimittäin jaettu ytimien kesken, osassa ei. Onkin siis varminta pyrkiä säikeistämään ohjelmat edelleen välttäen säikeiden välistä kommunikointia mahdollisimman paljon. Sutter ja Larus (2005) ounastelevat, että moniytimisillä prosessoreilla yksittäisten säikeiden luonti olisi vähemmän tehoa vaativa operaatio kuin muunlaisilla järjestelmillä, jolloin ohjelmistot voisivat käyttää säikeitä yhä pienempien ongelmien ratkaisuun menettämättä suoritusaikaa niitä luotaessa.

McDougall (2005) korostaa erityisesti skaalautuvuuden varmistamista. Hän tarkoittaa, että tulevaisuudessa voi olla suuriakin eroja sen mukaan, montako prosessoriydintä kussakin järjestelmässä on, joten ohjelmiston täytyy pystyä valitsemaan montako säiettä se käyttää. Ei ole lainkaan kaukaa haettua, että muutaman vuoden päästä tavallisessa prosessorissa on kymmeniä ytimiä. Toisaalta vanhemmat tietokoneet voivat olla edelleen yksiytimisiä. Jotta ohjelmisto toimisi mahdollisimman optimoidusti, sen kannattaa valita säikeidensä määrä suoritinydinten määrän mukaan. On mietittävä, voidaanko ohjelmisto suunnitella niin, että säikeiden määrä valitaan ohjelmiston käynnistyessä, vai täytyykö ohjelmistosta tehdä erilaisia versioita eri ydinmäärille. Tyystin eri asia on, voiko jonkinlaista ohjelmistoa edes säikeistää skaalautuvasti.

## 2.4 Yhteenveto

Edellä määriteltiin rinnakkaiskäsittely tarkoittamaan useamman kuin yhden käskyn suorittamisesta yhtäaikaisesti. Ohjelman jakaminen rinnakkain suoritettaviin osiin on rinnakkaisohjelmointia. Pääasialliset osittamisen perusteet ovat tehtäväperustainen ja tietoperustainen jakaminen. Tehtäväperustaisessa jakamisessa ohjelmasta etsitään

erillisiä tehtäviä, jotka voidaan suorittaa rinnakkain. Tietoperustaisessa jakamisessa suoritetaan usealle eri tiedolle rinnakkain samat tehtävät. Muutamia menetelmiä rinnakkain suoritettavien ohjelmanosien kommunikointiin ovat jaetun muistialueen käyttö, säikeiden käyttö ja kommunikointi viestien avulla.

Moniydinprosessorit ovat juuri tulleet markkinoille, ja niiden täysimittainen hyödyntäminen edellyttää rinnakkaisohjelmoinnin tekniikoiden käyttämistä ohjelmistonkehityksessä. Valitettavasti uusien konsolien prosessorit eroavat sekä toisistaan että PC-prosessoreista, joten sovellusten kehittäminen monialustaisesti PC- ja konsolilaitteille edellyttää laitealustakohtaisia ratkaisuja.

# 3 PELIMOOTTORIT

Jotta voitaisiin paremmin ymmärtää monisäikeisten pelimoottorin arkkitehtuurimalleja, tutustutaan tässä luvussa itse pelimoottoreihin ja niiden rakenteeseen. Aihetta lähestytään esittelemällä aluksi pelimoottoreita yleisesti käyttäen lähteenä kirjallisuutta, ja tämän jälkeen tarkemmin käyttämällä lähteenä vapaasti saatavilla olevaa dokumentaatiota pelimoottoreiden komponenteista sekä Unreal 2 pelimoottorista.

## 3.1 Johdanto

Ajatus pelimoottorien taustalla on puhtaasti kaupallinen. Koska koko peliteollisuuden olemassaolon ajan pelit ovat jatkuvasti muuttuneet monimutkaisemmiksi ja siten kalliimmiksi kehittää, on taloudellinen riski yhden pelin tekemisessä jatkuvasti kasvanut. Yksi ratkaisu ongelmaan on hajottaa riskiä useamman pelin kesken, tekemällä samalla pelin perusteknologialla monta pelejä. Tätä varten tekniset ratkaisut pelin taustalla täytyy erottaa pelin sisällöstä erilliseksi pelimoottoriksi, jota voidaan käyttää joko saman pelitalon muissa peleissä tai myydä eteenpäin. Jacobson & Lewis (2002) määrittelevät pelimoottorin simulaatiota ajavien moduulien kokoelmaksi, joka ei suoraan määrittele pelin toimintaa (game logic) tai pelin sisältöä. Pelimoottori voi hoitaa esimerkiksi syötteen vastaanottamisen, grafiikan piirron ja äänien toiston sekä pelin olioiden liikkumisen ja fysiikan mallinnuksen. Tässä luvussa luonnehditaan pelimoottoreiden rakennetta sekä perehdytään pelimoottorin muutamien keskeisimpien komponenttien tehtäviin ja toimintaan.

Vaikka pelit ja pelimoottorit ovat viime vuosina nousseet leikkikaluista hyväksytyiksi akateemisiksi työkaluiksi, ei niiden tutkimus ole vielä edennyt kovin pitkälle. Esimerkiksi pelimoottorien arkkitehtuureja ei ole vieläkään tieteellisesti tutkittu, vaikka pelimoottoreja käytetäänkin jonkin verran tieteellisessä simuloinnissa (kts. Jacobson & Lewis, 2002).

Mukaillen Jacobsonin ja Lewisin kuviota, kuvio 6 esittää pelimoottorin paikkaa järjestelmässä. Pelimoottori toimii käyttöjärjestelmän ajurien ja pelin toimintalogiikan välissä. Pelimoottorit ovat yhteydessä käyttöjärjestelmään jonkin sopivan rajapinnan, kuten DirectX:n kautta. Varsinaiselle pelin toimintalogiikalle pelimoottori voi olla musta laatikko, jota komennetaan rajapinnan kautta tekemään matalan tason tehtäviä. Pelin toimintalogiikka -kerrokseen kuuluvat esimerkiksi sellaiset tehtävät kuin olioiden liikkumaan komentaminen ja pelin voittajan päättäminen. Tehtävänjako ei tietenkään ole mikään standardi, vaan se saattaa olla erilainen eri peleissä. Itse pelimoottorissa ei yleensä ole mitään tiettyyn yksittäiseen peliin viittaavia toimintoja, joka saattaisi häiritä sen käyttämistä jossain toisessa pelissä.

*Kuvio 6: Pelimoottorin paikka järjestelmässä*

Tässä kohtaa on hyvä huomauttaa, että termin ”moottori” käyttö on kirjallisuudessa hyvin kirjavaa, se vaikuttaa jopa jonkinlaiselta muotisanalta. Pelimoottorin yksittäisistä komponenteista käytetään esimerkiksi nimityksiä äänimoottori, grafiikkamoottori tai fysiikkamoottori. Moottorin määritelmiä on myös yhtä monta kuin on kirjojakin. Tässä tutkimuksessa pyritään välttämään moottori-termin käyttöä muutoin kuin pelimoottorin yhteydessä. Tässäkin yhteydessä se esiintyy lähinnä nimityksenä, ei vaatimuksena tietystä toiminnallisuudesta. Toinen huomioitava seikka on, että ajoittain esiintyy tulkintoja, joissa pelimoottorin grafiikkakomponentista käytetään itse asiassa nimitystä pelimoottori. Esimerkkinä tästä Eberlyn (2004) kirja käsittelee otsikkonsa ja esipuheensa puolesta pelimoottoreita, mutta sisältönsä puolesta kolmiulotteisen grafiikan piirtämistä. Tulkinta luultavasti juontaa juurensa pelimoottoreiden alkuajoilta, jolloin pelimoottorit todella olivatkin lähinnä grafiikan piirtämiseen tarkoitettuja kirjastoja.

Tässä tutkimuksessa on pyritty välttämään käsitteen *peli* määrittelemistä. Peli ei nimittäin ole vain yleistetty käsite pelimoottorista, vaan peli on jotain, mihin pelimoottoria voi käyttää. Pelimoottorit eivät siis suoranaisesti liity peleihin, vaan pelimoottoria voi käyttää esimerkiksi tieteelliseen simulointiin (Jacobson & Lewis, 2002).

### 3.1.1 Pelisilmukka

Grafiikan näyttäminen peleissä perustuu ruutujen (frame) näyttämiseen. Kaikki ruudulla näkyvä piirretään sopivia grafiikkarutiineja käyttäen ensin johonkin osaan näytönohjaimen muistia, ja kun kaikki tarpeellinen on piirretty, piirros siirretään näytöllä näkyvään osaan muistia. Tämä välivaihe on tarpeellinen, jotta pelaaja ei näkisi kuinka grafiikka piirtyy näytölle. Koska jokaista ruudun piirtoa varten täytyy tietysti pelin edetä jonkin verran, on pelit yleensä suunniteltu yhden silmukan, pelisilmukan (game loop), ympärille. Seuraava pseudokoodi valottaa hyvin yksinkertaisen esimerkkipelisilmukan toimintaa:

```
while(pelaaja ei ole lopettanut){
      kerää syöte();
      pelin logiikka();
      liikuta hahmoja();
      piirrä grafiikka();
}
```

Silmukan sisältö riippuu tietysti pelimoottorin ominaisuuksista, mutta arkkitehtuuriltaan tällaiset pelimoottorit muistuttavat tavallaan eräänlaista tietovuomallia, jossa pelin datalle suoritetaan erilaisia tehtäviä sen edetessä silmukan kierroksessa. Mitä nopeammin silmukka kulkee läpi, sitä suurempi on pelin ruudunpäivitysnopeus, ja sitä sujuvammalta peli tuntuu. Jotta jokaisen ruudunpäivityksen aika olisi kutakuinkin sama kuin muidenkin, ei pelisilmukan toimintoihin saa sijoittaa mitään yksittäisiä aikaa vieviä toimintoja.

### 3.1.2 Oliomallin mukainen pelisilmukka

Edellisessä esimerkissä ollut pelisilmukka oli rakenteisen ohjelmointimallin mukainen. Nykyaikaiset pelimoottorit, kuten Unreal Engine 3 ja Doom 3, on tehty oliomallin mukaan. Niiden arkkitehtuuri on valitettavasti salainen, mutta todennäköisesti niiden pelisilmukka muistuttaa seuraavaa:

```
while(pelaaja ei ole lopettanut){
      kerää syöte();
      for(jokaiselle pelin oliolle){
            olio.suoritaTehtävät();
            olio.piirrä();
      }
}
```

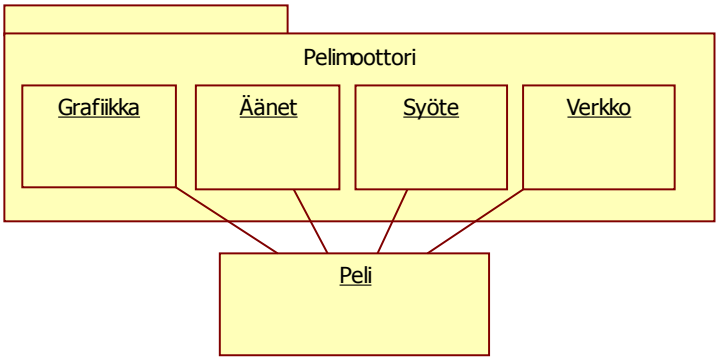
Oliomallissa pelikohtaiset erot tulevat suoritaTehtävät-rutiinista. Jos olion esimerkiksi täytyy sanoa jotain, suoritaTehtävät-rutiini kutsuu pelimoottorin äänentoistotoimintoa. Ajatuksena on perinnän avulla tehdä pelimoottorin perusluokasta esimerkiksi autoja ja ihmisiä, ja toteuttaa näille jokaiselle omantyyppisensä toiminta. Esimerkiksi Unreal Engine 3:ssa on valmiina tuki pelaajahahmoille, ei-pelaajahahmoille, varastoille (inventory), aseille, ja herätteille (trigger).

Koska oliomallin mukaisessa pelisilmukassakin täytyy esimerkiksi olioiden sijaintia ja animaatiota päivittää joka kierroksella, sisältää edellisen esimerkin suoritaTehtävät-metodi yleensä enemmän tai vähemmän samat tehtävät kuin rakenteisen mallin koko silmukka. Niinpä oliomallissakaan ei voida päästä puhtaaseen tapahtumapohjaiseen käsittelyyn.

### 3.1.3 Pelimoottorin rakenne

Zerbst (2004, 36) esittelee yhden toteutuksen yksinkertaisesta pelimoottorista, joka on toteutettu kuvion 7 mukaisella arkkitehtuurilla. Pelimoottori paljastaa pelille erilliset rajapinnat grafiikan piirtämiseksi, äänien soittamiseksi, syötteen vastaanottamiseksi ja verkon käyttämiseksi. Zerbstin pelimoottori on siis vain kokoelma erilaisia

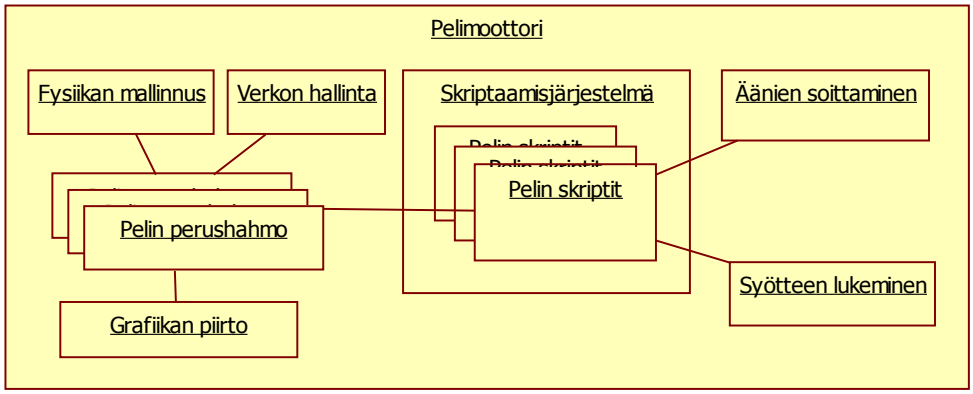
toimintoja, joiden välillä ei välttämättä ole mitään yhteistoimintaa. Andreoli et al. (2005) esittävät myös samantyyppisen näkemyksen pelimoottorin ja pelin suhteesta, jossa pelimoottoriin kuuluisivat vain matalan tason komponentit grafiikan, äänien ja verkon hallintaan. Kuitenkin kaupallisiin pelimoottoreihin kuuluu hyvin paljon korkeankin tason ominaisuuksia. Näkemyseron takana lienee se ongelma, että pelimoottorin käsite ei ole vakioitunut.




*Kuvio 7: Zerbstin (2004) pelimoottorin arkkitehtuuri*

Nykyisten kaupallisten pelimoottoreiden todellisiin arkkitehtuureihin ei ole mahdollista päästä tutustumaan, sillä niitä pidetään liikesalaisuuksina. Toista sukupolvea kaupallisissa pelimoottoreissa edustaa vuonna 2003 julkaistu Unreal Engine 2, jonka dokumentaatio on osittain julkista. Dokumentaation mukaan Unreal Engineä käytetään pääasiassa erillisen moottoria varten kehitetyn niin sanotun skriptauskielen, UnrealScriptin, kautta. Pelimoottori hoitaa sisäisesti kaiken olioiden päivityksestä ja verkon hallinnasta grafiikan piirtämiseen. Pelin kehittäjän vastuulle jää antaa olioille komentoja kielen avulla. Esimerkiksi luodin osuessa pelaajaan pelimoottori käynnistää pelinkehittäjän luoman funktion, joka pyytää pelimoottoria soittamaan äänen, vähentää pelaajan terveyspisteitä ja pyytää pelimoottoria poistamaan luodin maailmasta. Pelimoottori paljastaa skriptausjärjestelmälle tietyn rajapinnan sekä antaa sen käyttöön joitain keskeisiä olioita. Skriptauskielellä tehdyistä funktioista käytetään paremman suomennoksen puuttuessa nimitystä skripti.

Oliomallinen pelimoottori, jossa on erillinen skriptauskieli, voisi näyttää esimerkiksi kuvion 8 mukaiselta. Skriptausjärjestelmä sisältää nyt pelin toimintalogiikan, ja se on kosketuksissa pelimoottoriin sekä pelimoottorin tarjoamien rajapintojen että tietyntyyppisten olioiden kautta. Kuviossa pelin perusolioluokka on nimeltään Pelin perushahmo. Voidaan ajatella, että tästä luokasta peritään kaikki pelissä näkyvät oliot, ja skriptin kautta voidaan komentaa esimerkiksi tällaista oliota liikkumaan. Tällöin pelimoottori hoitaa automaattisesti olion liikuttamisen, sen mahdollisen välityksen verkon yli muille pelaajille sekä piirtämisen.

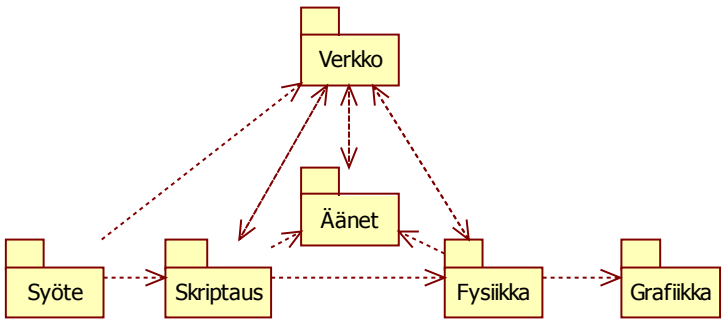



*Kuvio 8: Pelimoottori, jossa on skriptausjärjestelmä*

## 3.2 Pelimoottorin komponentit

Unreal Engine, niin kuin suurin osa muistakin pelimoottoreista, perustuu komponenttien käyttöön. Kaupallisista pelimoottoreista löytyy hyvin usein joko Ageia PhysX tai Havok -fysiikkakomponentti. Komponenttien käyttäminen on pelimoottoreissa varsin luonnollinen ratkaisu, sillä pelimoottorin eri tehtävät ovat usein erillisiä – syötteen lukeminen ei liity suoraan mitenkään grafiikan piirtämiseen, eikä äänien toistaminen verkon hallintaan. Useisiin tehtäviin on saatavilla valmiita komponentteja, vaikkakin kaupallisissa pelimoottoreissa komponentit valmistetaan usein itse.

Seuraavassa esitellään muutamia pelimoottoreiden tärkeimpiä komponentteja sekä niiden kytkentöjä toisiinsa. Kuvio 9 esittää yhteenvetoa näistä kytkennöistä. Esitettävien komponenttien lista ei ole kattava, vaan listasta on jätetty tarkoituksella pois muun muassa tekoäly, koska tekoäly on olennaisilta osiltaan pelikohtaista. Monissa pelimoottoreissa on kuitenkin tekoälyn tukitoimintoja, kuten reitinetsintä, mutta niiden merkitys koko pelimoottorin toiminnan kannalta on pieni. Tässä esityksessä tekoälyn voidaan ajatella liittyvät skriptauskomponenttiin.




*Kuvio 9: Komponenttien välisen kytkennät*

### 3.2.1 Grafiikka

Grafiikkakomponentin tarkoitus on komentaa grafiikkarajapintaa piirtämään pelin 3d-mallit. Yleisimmät rajapinnat 3d-grafiikan piirtoon ovat DirectX ja OpenGL. Ensin komponentin täytyy pyytää rajapintaa tyhjentämään näyttö, sitten rajapinnalle annetaan yksi kerrallaan piirrettäväksi näytölle tulevat mallit, ja lopuksi rajapintaa pyydetään näyttämään kuva.

Cheah & Ng (2005) kertovat, että grafiikkakomponentissa käytetään tietovarastona näkymägraafia (scene graph), joka kontrolloi mallien piirtojärjestystä sekä lisää suorituskykyä huolehtimalla, että vain näkyviin tulevat mallit piirretään. Näkymägraafi toimii myös grafiikkakomponentin rajapintana sovellukselle, joka voi esimerkiksi ladata malleja näkymägraafiin sekä siirtää mallien paikkaa ja asettaa niihin animaation.

Grafiikkakomponentti on kytköksissä pelimoottorin olioiden sijainnista vastaavaan komponenttiin, esimerkiksi fysiikkakomponenttiin, niin, että jokaisella ruudun päivityksellä olioiden päivitetyt sijainnit ja animaatioiden tilat täytyy päivittää näkymägraafiin.

### 3.2.2 Syöte

Windows tarjoaa tapahtumapohjaisen syötteen hallinnan, eli järjestelmä luo uuden tapahtuman jokaiselle näppäimen painallukselle. Astley & Hawkins (2002, 478) kertovat, että Windowsin normaali järjestelmä ei tarjoa peleille riittävän reaaliaikaista tietoa syöttölaitteiden tilasta. He suosittelevatkin DirectX:n käyttämistä. DirectX:n syötön hallinta toimii niin, että update-käskyllä otetaan syöttölaitteiden senhetkinen tila suoraan ajureilta.

Astley ja Hawkins esittävät yksinkertaisen esimerkkitoteutuksen syötettä hallitsevasta komponentista. Se toimii niin, että komponenttia täytyy pyytää päivittämään tilansa jokaisella pelisilmukan kierroksella. Tämän jälkeen komponentilta voidaan kysyä, missä tilassa mikäkin näppäin oli, ja missä paikassa hiiri sijaitsi. Tällainen kompo-

nentti ei siis suoraan riipu mistään muusta komponentista, mutta muut komponentit voivat käyttää sitä selvittääkseen syöttölaitteiden tilan. Tavallisesti syötteeseen reagointi on pelin skriptien tehtävä.

### 3.2.3 Äänet

Nykyaikaisissa peleissä äänien paikka määritetään kolmiulotteisesti. Tunnetuin äänien toistamiseen tarkoitettu komponentti on OpenAL. OpenAL:n dokumentaation mukaan komponenttia käytetään sijoittamalla kolmiulotteiseen maailmaan äänilähteitä sekä kuuntelija. Äänilähteet soittavat niille määrättyä ääntä annetuilla parametreilla, joita voivat olla esimerkiksi kaiun määrä ja kuvitteellisen tilan koko. Kuuntelijan sijainnin perusteella määritetään äänien sijainti pelaajan äänimaailmassa.

Samoin kuin grafiikkakomponentti, äänikomponentti on myös sidoksissa fysiikkakomponenttiin pelin olioiden sijainnin puolesta. Ainakin kuuntelijan sijainti täytyy päivittää fysiikkakomponentin mukaisesti. Toisaalta kaikki äänet eivät välttämättä ole kytköksissä pelissä näkyviin olioihin, vaan äänien toistamista voidaan pyytää myös sopivissa tilanteissa skriptauskomponentista.

### 3.2.4 Verkko

Monissa pelimoottoreissa on erityisesti niitä varten suunniteltu verkkokomponentti, joten niiden toteutuksesta on vaikea sanoa mitään varmaan. Yleisiä käytössä olevia valmiita ratkaisuja ovat Microsoftin DirectPlay ja Rakkarsoftin RakNet. RakNet lienee näistä korkeamman tason ratkaisu. Syy siihen, miksi pelimoottoreissa käytetään erityisesti sitä varten suunniteltuja ratkaisuja on, että yleiskäyttöiset ratkaisut voivat ovat hitaita ja saattavat välittää verkon yli paljon turhaakin tietoa. Esimerkkinä yleiskäyttöisyydestä RakNet tarjoaa palvelua, joka välittää tietyn olion täydellisen tilan ja kaikki tilan muutokset muille yhteydessä oleville tietokoneille. Tällöin välitetään mahdollisesti turhiakin muutoksia, sekä muutoksia tarpeettoman suurella tarkkuudella. Esimerkiksi pelaajan sijaintijollakin akselilla saatetaan välittää 32 bittinä, kun 16 riittäisi.

Pelimoottoreissa käytetäänkin verkkokomponenttia usein matalammalla tasolla, välittämään optimoituja pelipalvelimen viestejä asiakkaille ja toisinpäin. RakNet toimii tässä suhteessa niin, että se kerää kaikki tiettynä aikana tulleet viestit ja lähettää ne tiivistetyssä muodossa optimoiden pakettien kokoa. Viestien vastaanottamista varten jokaisella pelisilmukan kierroksella kysytään verkkokomponentilta, onko viestejä tullut.

Verkkokomponentin kytkökset muihin komponentteihin ovat hyvin monimutkaisia ja riippuvat paljolti pelimoottorista. Mahdollisia kytköksiä ovat kytkös fysiikkakomponenttiin pelin olioiden paikkojen määrittämisessä, kytkös syötettä hallitsevaan komponenttiin pelaajien siirtojen välittämiseksi palvelimelle, kytkös ääniä toistavaan komponenttiin äänien toistamiseksi asiakastietokoneilla sekä paljon muita kytköksiä riippuen pelimoottorin toiminnasta.

### 3.2.5 Skriptaus

Skriptauskomponentin tarkoitus on erottaa pelin sisältö ja toimintalogiikka pelimoottorista. Pelin sisältö määritellään näinollen yleensä skriptien avulla. Varanesen (2002) mukaan skriptejä käytetään, jotta pelin sisällöllisien muutoksien tekemiseen, kuten jonkin olion ominaisuuden muuttamiseen, ei tarvitsisi vaivata ohjelmoijia. Yksi etu skriptien käytöstä on myös, että muutokset eivät vaadi koko sovelluksen uudelleen kääntämistä, vaan skripteihin tehdyt muutokset luetaan esimerkiksi pelin käynnistyessä. Skriptauskomponenttia voidaan käyttää myös tekoälyn rakentamiseen. Skriptinä voi olla esimerkiksi päätöspuu, joka tietyn tapahtuman yhteydessä päättää, miten oliot siihen reagoivat.

Varanese (2002) jakaa skriptauskielet kahteen kategoriaan, komentopohjaisiin kieliin ja proseduraalisiin kieliin. Komentopohjaisissa kielissä kirjoitetaan kaikki tietyn skriptin pelimoottorille antamat käskyt peräkkäin, eikä niissä ole korkean tason rakenteita kuten silmukoita. Proseduraaliset skriptauskielet, kuten Lua ja Python puolestaan vastaavat C:tä tai Pascalia. Viime aikoina skriptauskieliin on myös tullut olio-

maisia piirteitä, esimerkkinä tästä C++ -syntaksia muistuttava Angelscript. Skriptauskieliä yhdistävä tekijä on kielien kääntäminen jonkinlaiseksi tavukoodiksi, jota ajetaan skriptauskomponentin avulla.

Varanesen mukaan pelimoottori voi kutsua tiettyä ladattua skriptiä ja välittää sille parametreja. Skripti voi puolestaan kutsua pelimoottorin funktioita sille avatun rajapinnan kautta. Milloin skriptejä kutsutaan ja mitä rajapintoja niille avataan, on tietysti täysin pelimoottorikohtainen asia. Esimerkiksi Unreal Engine painottaa skriptien toimintaa erityisesti pelin olioiden kautta. Kun oliolle tapahtuu jotain, kutsutaan tapahtumalle kirjoitettua skriptiä, joka puolestaan voi vaikuttaa pelimoottoriin lähinnä itse olioon sisällytettyjen toimintojen kautta.

### 3.2.6 Fysiikka

Fysiikkakomponentti liikuttaa pelissä olevia esineitä tiettyjen fysikaalisten mallien mukaisesti. Fysiikkakomponentti voi esimerkiksi mallintaa pelissä olevaa autoa ottaen huomioon renkaiden kitkan, iskunvaimentimet ja moottorin tehon. Conger (2004, 4) luettelee fysiikan käyttökohteita peleissä: liikkuminen, pyöriminen, kitka, ilmanvastus, painovoima, törmäykset, räjähdykset, joustavat ja kiinteät kappaleet sekä aallot.

Eräs suosittu ilmainen fysiikkakomponentti on Newton Game Dynamics, joka toimii sen dokumentaation mukaan seuraavasti. Uusia kappaleita luotaessa niille annetaan jokin fysikaalinen malli, esimerkiksi kuutio, pallo, tai kolmioista koostuva 3d-malli. Jokaisella pelisilmukan päivityksellä kappaleille annetaan kullakin hetkellä vaikuttavat voimat sekä liikkeen että pyörimisen suhteen. Nämä tiedot tulevat tavallisesti skripteiltä, esimerkiksi pelaajan liikkuessa skripti antaa fysiikkakomponentille käskyn työntää pelaajahahmoa tietyllä voimalla. Tämän jälkeen kutsutaan komponentin päivitä-toimintoa, jolle annetaan parametrina, montako sekuntia simulaatiota lasketaan eteenpäin. Päivityksen jälkeen kappaleiden uudet sijainnit ja kulmat voidaan kysyä fysiikkakomponentilta ja siirtää grafiikkakomponentin käyttöön.

## 3.3 Yhteenveto

Tämän luvun tavoitteena oli perehdyttää lukija pelimoottoreiden toimintaan ja rakenteeseen. Pelistä voidaan erottaa sen sisältö ja sen tekninen toteutus. Pelimoottori on pelin tekninen osa, jonka tehtäviin usein kuuluvat esimerkiksi grafiikan piirtäminen, äänien toistaminen, fysiikan mallinnus sekä verkon ja syötteen hallinta. Näiden toimintojen väliset kytkennät ovat heikkoja, joten ne on mahdollista toteuttaa erillisten komponenttien avulla.

Pelimoottorin simulaatiota päivittävää ohjelmanosaa kutsutaan pelisilmukaksi. Se on loputon silmukka, joka jokaisella kierroksella päivittää pelin maailmaa vähän eteenpäin sekä piirtää kuvan ruudulle. Nykyaikaiset pelimoottorit on toteutettu oliomallin mukaisesti, jolloin pelisilmukan tehtävänä onkin pyytää jokaista oliota päivittämään tilaansa.

Tässä luvussa esitetty on väistämättä yksinkertaistettu näkemys pelimoottoreista. Todellisissa pelimoottoreissa on luonnollisesti myös sellaisia toimintoja, joita tässä ei ole tuotu esille. Myöskään nykyaikaisten kaupallisten pelimoottoreiden rakenteesta on vaikea sanoa mitään varmaa, sillä niiden tiedot ovat liikesalaisuuksia.

# 4 MONISÄIKEISEN PELIMOOTTORIN MALLIT

## 4.1 Johdanto

Tässä luvussa perehdytään pelimoottoreiden rinnakkaisohjelmoinnin eri keinoihin. Aluksi esitellään ratkaisuja, jotka saattaisivat tulla mieleen pelimoottoreita heikosti tunteville ja jotka ovat ratkaisuina toimimattomia. Tämän jälkeen esitellään pelimoottorin komponenttien sisäistä rinnakkaisohjelmointia. Komponenttien sisäistä monisäikeistystä ei oteta mukaan varsinaiseen vertailuun, sillä tavoitteena on tutkia ratkaisuja, jotka käsittävät koko pelimoottorin toiminnan. Sellaisia ratkaisuja esitelläänkin lopuksi kohdissa 4.4 ja 4.5. Kappale 4.4 esittelee kahta kirjallisuudesta löydettyä mallia, jotka perustuvat pelimoottorin eri tehtävien sijoittamiseen eri säikeisiin. Kappale 4.5 esittelee tämän tutkimuksen yhteydessä ideoitua mallia, joka perustuu pelin eri olioiden toimintojen sijoittamiseen eri säikeisiin. Mallien välillä voidaan nähdä selvä jako ongelman osittamisen strategioissa tehtäväperustaisiin ja tietoperustaisiin. Näitä strategioita on käsitellyt muun muassa Foster (1995).

## 4.2 Triviaalit ratkaisut

Koska pelisilmukka jakautuu selkeästi peräkkäisiin piirrettäviin ruutuihin, olisi yksinkertaisin ratkaisu jakaa yksittäisten pelisilmukan kierrosten käsittely ja piirtäminen eri säikeisiin (Gabb & Lake, 2005). Tämä ei ole mahdollista, koska ruudut ovat luonteeltaan peräkkäisiä. Nimittäin ainakin kappaleiden paikka yhdellä ruudulla riippuu sen paikasta edellisellä ruudulla. Eräänä ongelmana olisi myös, mihin ruutuun milläkin ajan hetkellä kaapattu syöte kuuluisi. Lisäksi piirtämällä useita ruutuja yhtäaikaisesti ne pitäisi kuitenkin esittää pelaajalle oikeassa järjestyksessä ja oikeaan aikaan, joka kumoaisi hyödyn niiden piirtämisestä etukäteen.

Toinen yksinkertainen ratkaisu on käyttää automaattisia työkaluja, esimerkiksi kappaleessa 2.2.7 esitetyn Intelin C++-kääntäjän automatiikkaa, säikeistyksen toteuttamiseen. Automaattiset työkalut pystyvät kuitenkin vain hyvin pienimuotoiseen säikeistykseen, vaikka niiden markkinointipuheissa toisin mainitaankin. Kääntäjät eivät edelleenkään voi optimoida kuin yksinkertaisia silmukoita, joissa on kiinteä kierrosmäärä, ei aliohjelmakutsuja eikä keskeytyksiä. Näin yksinkertaisia silmukoita ei etenkään peleissä ole merkittävissä määrin, eivätkä ne useimmiten muodosta merkittävää suorituskykypullonkaulaa. Ylimääräisten säikeiden luonti voi jopa haitata suorituskykyä johtuen säikeiden luonnin hitaudesta verrattuna silmukan suorittamisen viemään aikaan.

## 4.3 Komponenttien sisäinen rinnakkaistus ja tehtäväjonot

Yksi mahdollisuus lisätä rinnakkaisuutta pelimoottorin toimintaan on muokata komponenttien sisäinen toiminta käyttämään useita säikeitä. Monia peleissä käytettyjä algoritmeja, esimerkiksi reitinhaku- ja järjestämisalgoritmeja, on mahdollista toteuttaa rinnakkaisohjelmoinnin avulla. Joidenkin pelimoottorin komponenttien toimintaa on myös mahdollista jakaa alitehtäviin, jotka suoritetaan erillisissä säikeissä. Joidenkin komponenttien toimintaa puolestaan ei ole mahdollista rinnakkaistaa. Esimerkiksi grafiikkakomponentissa rinnakkaistus ei onnistuisi kovin helposti, sillä näytönohjaimissa ei ole rinnakkaistuksen mahdollistavia toimintoja. Igehy et al. (1998) tosin ovat esittäneet OpenGL-rajapinnan laajentamista mahdollistamaan useiden säikeiden yhtäaikaisen piirtämisen, mutta muutoksia ei vielä ole tehty. Näytönohjaimelle kuvat onkin piirrettävä peräkkäin, ainakin toistaiseksi. Esimerkkinä komponentin sisäisestä rinnakkaistuksesta on fysiikkakomponentti Havok, joka pystyy siirtämään osan fysiikan laskennasta pieniin alitehtäviin, jotka suoritetaan erityisen tehtäväjonon avulla.

Tehtäväjonon käyttämistä on lyhyesti luonnehtinut Wilson (2006). Ajatuksena on löytää isommasta kokonaisuudesta pieniä tehtäviä, jotka voidaan laittaa jonoon odottamaan suoritusta. Jonoa ylläpitävä järjestelmä asettaa tehtäviä suoritettavaksi vapaille prosessoriytimille. Ajatuksen taustalla on Playstation 3:n Cell-prosessori, jonka

rinnakkaisprosessointiyksiköiden toiminta on rajattu vain 256 kilotavuun paikallista muistia. Tämäkin määrä täytyy vielä jakaa suoritettavan koodin ja käsiteltävän datan kesken. Rinnakkain ajettavat tehtävät eivät voi kommunikoida keskenään, ja pääsäikeenkin kanssa vain erityisten muistisiirtojen avulla. Tämän rajoituksen vuoksi Cell-prosessorille täytyy antaa rinnakkain suoritettavaksi varsin pieniä tehtäviä, ja tehtäväpaketin mukaan täytyy laittaa kaikki tarpeellinen tieto tehtävän suoritusta varten. Vaikka tehtäväjonomalli soveltuukin erityisesti Playstationille, voidaan sitä Wilsonin mukaan emuloida myös PC- ja Xbox suorittimissa – joskin hitaasti.

Esimerkkinä tehtäväjonon käyttämisestä voidaan ottaa fysiikkakomponentin törmäystarkistuksien tekeminen. Kun komponentti on löytänyt joukon mahdollisia törmäyksiä, se luo tehtäviä, joissa tarkistetaan yksi tai useampi törmäys ja reagoidaan törmäyksiin. Sillä aikaa kun törmäystarkistuksia suoritetaan tehtäväjonossa, voi fysiikkakomponentti jo etsiä seuraavia mahdollisia törmäyksiä.

Komponenttien sisäistä rinnakkaistusta käytetään ainakin Unreal 3 -pelimoottorissa, joka oli markkinoiden ensimmäisiä moniytimisyyttä tukevia pelimoottoreita. Unreal-pelimoottorin suunnittelijan Tim Sweeneyn haastattelu Anandtech-sivustolla (De Gelas, 2005) paljastaa, että pelimoottorin monisäikeistyksessä he ovat keskittyneet yksittäisten suljettujen järjestelmien toiminnan tehostamiseen. Sweeney suosittelee, että laajempaan säikeistykseen ei kannata ryhtyä, koska tämä moninkertaistaa kehityskustannukset.

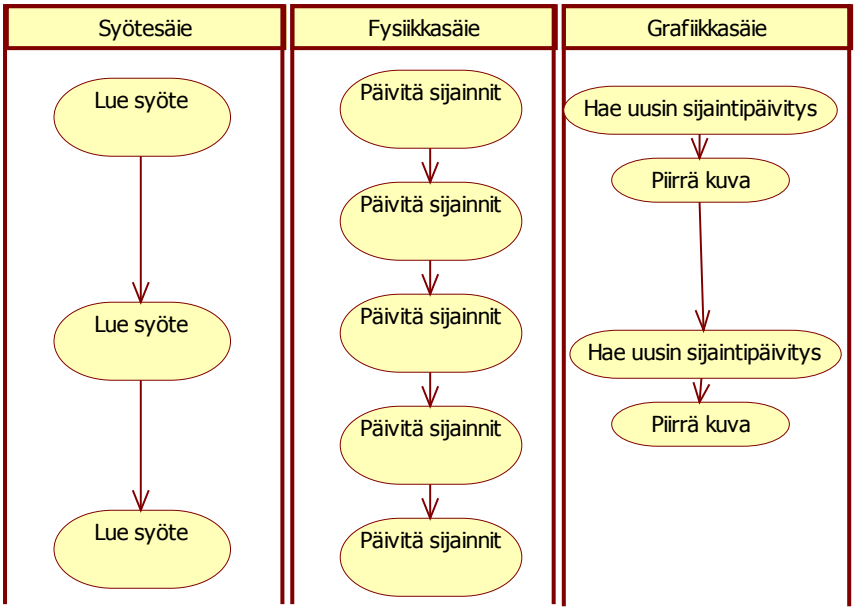
Amdahlin lain mukaan algoritmien optimoinneista saatava nopeushyöty on riippuvainen siitä, kuinka suuri osa algoritmista voidaan rinnakkaistaa. Tämän lisäksi se riippuu siitä, kuinka suuri tehtävä on kyseessä. Pienellä tehtävällä saattaa monisäikeistys jopa hidastaa toimintaa, johtuen säikeiden luomisesta johtuvasta kuormituksesta. Voidaan olettaa, että pelkästään komponenttien sisäisellä optimoinneilla ei voida päästä täyteen käyttöasteeseen moniydinprosessoreille, sillä aina jotakin osaa algoritmista ei voida rinnakkaistaa. Komponenttien sisäisen rinnakkaistuksen etuihin voidaan lukea yksinkertaisuuden – sitä voidaan helposti soveltaa olemassa oleviin pelimoottoreihin.

Komponenttien sisäistä säikeistystä ei käsitellä tätä pidemmälle, sillä sen soveltamiseksi ei tarvita erityistä korkean tason suunnittelua, ja sitä voidaan käyttää täydentämään muita esitettyjä säikeistyksen malleja.

## 4.4 Tehtäväperustainen rinnakkaistus

Gabb ja Lake (2005) esittävät pelimoottorin eri komponenttien jakamista erillisiin säikeisiin, esimerkiksi niin, että fysiikkakomponentti päivittäisi kappaleiden sijaintia omaan tahtiinsa ja grafiikkakomponentti kävisi sopivan ajan välein katsomassa, mikä on fysiikkakomponentin uusin päivitys, ja piirtäisi kappaleet sen mukaan. Tällä periaatteella varsinaisesta pelisilmukasta voitaisiin päästä kokonaan eroon. Kuvio 10, joka mukailee Gabbin ja Laken kuviota, hahmottaa tätä mallia. Gabbin ja Laken mukaan komponenttien päivitysnopeutta täytyy säätää komponenttikohtaisesti, eli esimerkiksi syötekomponentti voisi päivittyä useammin kuin fysiikkakomponentti. Jatkossa tästä mallista tullaan käyttämään nimeä *asynkroninen tehtäväpohjainen rinnakkaistus*.

Gabb ja Lake (2005) eivät suoraan kerro, kuinka säikeiden välinen viestintä olisi tarkoitus hoitaa, mutta heidän kuviostaan voisi ymmärtää, että tarkoitus on käyttää eräänlaisia inkrementtejä. Esimerkiksi fysiikkasäie pitää sisällään tietoa maailman senhetkisestä tilasta, ja kun säie päivittää tilaa jonkin aikayksikön verran eteenpäin, se kopioi tilasta eräänlaisen muille säikeille luettavaksi tarjottavan kopion. Näin fysiikan laskemista ei tarvitse keskeyttää, kun esimerkiksi grafiikkasäie tarvitsee tietää olioiden sijainnin. On harmillista, että Gabb ja Lake eivät ole tarkemmin esitelleet malliaan tai analysoineet sen sopivuutta tai sopimattomuutta johonkin käyttötarkoitukseen.

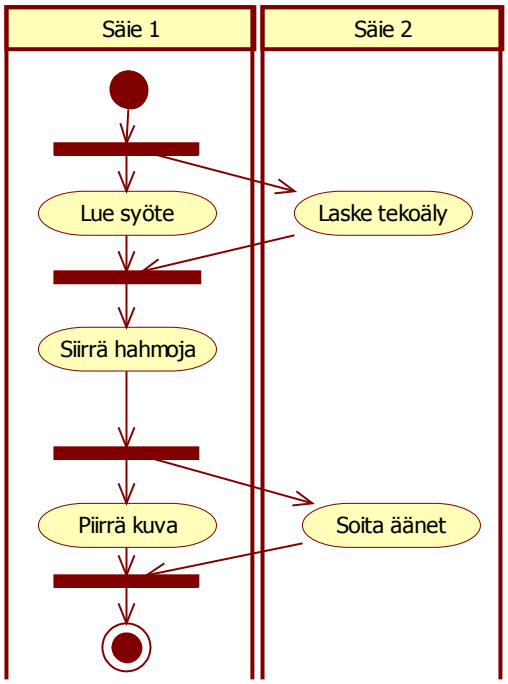



*Kuvio 10: Gabbin ja Laken (2005) asynkroninen tehtäväpohjainen rinnakkaistus. Tässä esimerkissä syötteen lukeminen, sijaintien päivittäminen ja kuvan piirto on sijoitettu eri säikeisiin.*

El Rhalibi et al. (2005) esittävät myös samantyyppisen idean. Heidän mallinsa perustuu pelisovelluksen toiminnallisuuden osioimiseen pieniin osiin, joista kootaan eräänlainen *edeltäjyysgraafi* (dependency graph, kts. esim. Pingali et al., 1991). Tähän graafiin kuvataan, minkä osan toiminta on riippuvainen minkäkin osan toiminnasta. Esimerkkeinä rinnakkaisista osista El Rhalibi et al. antavat näppäimistösyötteen lukemisen ja välittämisen syötettä tarvitseville osille, näyttömuistin tyhjentämisen ennen ruudun piirtämistä, maailman kolmiulotteiseen näyttämiseen tarvittavan kameramatriisin laskemisen, kuvassa näkymättömien osien etsimisen sekä taivaan piirtämisen.

Löydetyt osat ja edeltäjyysgraafi syötetään ohjelmakehikkoon, jota he kutsuvat nimellä *Concurrent Game Programming Framework.* Kehikko hoitaa automaattisesti tehtävien rinnakkaistuksen graafin avulla. El Rhalibi et al. eivät kuitenkaan kerro tarkemmin miten järjestelmä toimii. Päättelemällä voidaan kuitenkin arvioida, että se pyrkii etsimään edeltäjyysgraafista rinnakkaisia riippumattomia polkuja ja suorittaa näitä omissa säikeissään. Järjestelmä toimii pelisilmukan pohjalta, eli rinnakkaistus

tapahtuu vain yhden silmukan kierroksen sisällä. Kuvio 11 esittää esimerkkiä tästä mallista. Kuviossa pelimoottorin yksi kierros on jaettu kahteen säikeeseen, joissa pyritään suorittamaan rinnakkain toisistaan riippumattomia tehtäviä. Tästä mallista tullaan jatkossa käyttämään nimitystä *synkroninen tehtäväpohjainen rinnakkaistus*.



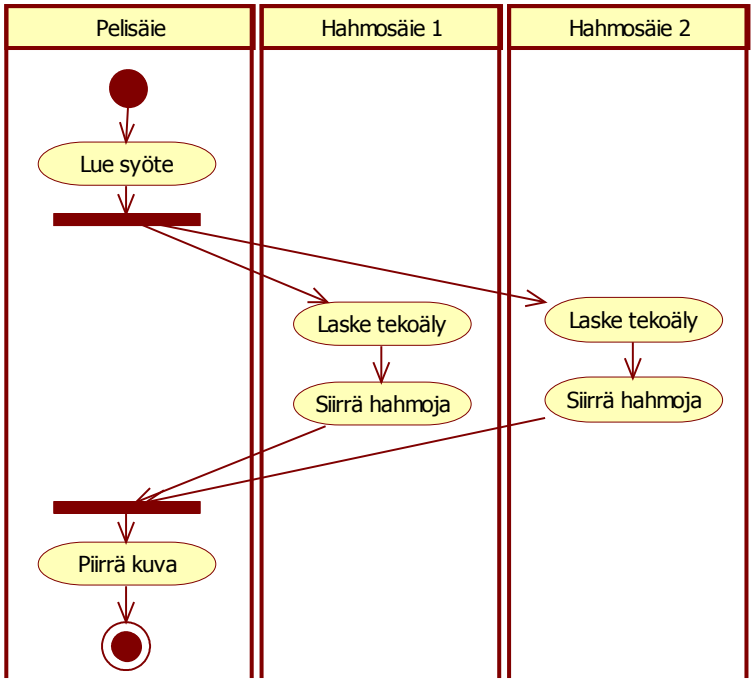

*Kuvio 11: El Rhalibi et al. (2005) esittämän synkronisen tehtäväpohjaisen rinnakkaistuksen mukainen esimerkki.*

Edellä kuvatut mallit perustuivat siis molemmat pelimoottorin eri tehtävien rinnakkaistamiseen. Asynkroninen malli pohjautuu siihen, että tehtävät toimivat toisistaan suoraan riippumattomissa säikeissä. Jokainen säie myöskin jatkaa tehtävän suorittamista jatkuvasti. Synkroninen malli pohjautuu puolestaan siihen, että pelisilmukan sisältä pyritään löytämään toisistaan riippumattomia suorituksen polkuja, ja luodaan näitä varten oma lyhytaikainen säikeensä. Kun polun sisältämät tehtävät on yhdellä silmukan kierroksella suoritettu, säie poistetaan, ja käynnistetään uudelleen seuraavalla kierroksella.

## 4.5 Tietoperustainen rinnakkaistus

Kuten kohdassa 2.2.1 mainittiin, voidaan ohjelma rinnakkaistaa joko tehtäväperustaisesti tai tietoperustaisesti. Koska kirjallisuudessa esitetyt menetelmät eivät kata tietoperustaista rinnakkaistusta, ideoitiin tätä tutkimusta varten erityinen tietoperustaisen rinnakkaistuksen malli käyttäen hyväksi kohdassa 2.2.1 esitettyä Fosterin (1995) menetelmää. Ensimmäisenä täytyy valita, minkä tiedon mukaan jaottelu tehdään. Vaihtoehtoja ovat ainakin pelin oliot ja jopa pelaajan antamat yksittäiset komennot. Jälkimmäisen voi suoraan sivuuttaa, koska pelaaja yleensä odottaa komentojensa toteutuvan antamassaan järjestyksessä. Suora oliopohjainen jako puolestaan tarkoittaisi eri olioiden toiminnan jakamista eri säikeisiin. Jaottelun perustaksi otetaan pelin sisältämät oliot, ja jatkossa tämän jaottelun mukaisesta mallista käytetään nimitystä *olioperustainen rinnakkaistus*.

Olioperustaisen rinnakkaistuksen ajatuksena on siirtää eri säikeisiin eri olioille suoritettavat toiminnot. Näin esimerkiksi jos pelissä on kymmenen oliota, voitaisiin luoda kaksi säiettä jotka kukin toteuttavat viiteen olioon liittyvät toiminnot. Näitä toimintoja voivat olla ovat esimerkiksi tekoälyn ja fysiikan laskeminen. Grafiikkakomponentin täytyy piirtämistä varten tietää kaikkien olioiden tila jollakin tietyllä ajan hetkellä, jolloin tämänkin mallin mukainen jaottelu täytyy synkronisoida pelisilmukan kierroksen mukaan. Kuvio 12 hahmottaa olioperustaisen rinnakkaistuksen mukaista pelisilmukkaa. Siinä oliot on jaettu kahteen eri säikeeseen, ja niille suoritetaan rinnakkain muutamia toimintoja.




*Kuvio 12: Olioperustainen rinnakkaistus.*

Yhtenä pohtimisen aiheena on, miten minimoidaan säikeiden välistä viestintää ja siten synkronoinnin tarvetta. Pelin eri olioiden väliseen kommunikointiin on tarvetta monissa tilanteissa. Esimerkiksi fysiikkakomponentin täytyy pystyä laskemaan olion törmäyksiä sen lähellä oleviin olioihin. Myös skripteillä on tarve tietää tietyn olion lähellä sijaitsevia olioita, esimerkiksi jotta pelaaja voisi käyttää pelissä olevaa valokatkaisijaa. Koska pelit simuloivat jossain määrin todellista maailmaa, on niin, että oliot vaikuttavat lähinnä vieressään oleviin olioihin. Tällä perusteella on järkevää valita samaan säikeeseen toisiensa lähellä olevia olioita. Näin yhteen säikeeseen voitaisiin valita vaikkapa jollain tietyllä alueella olevat oliot. Viestinnän tarve tässä mallissa tulee raja-alueilla olevista olioista sekä olioiden liikkumisesta alueiden välillä. Pelin oliot täytyy pystyä jollain keinolla määräämään tietylle oliosäikeelle. Tätä varten kannattanee luoda erityinen järjestelmä, joka osaa määrittää sopivan säiemäärän ja aluekoon sekä tarkkailee olioiden liikkumista alueiden välillä, ja siirtää tarvittaessa olion säikeestä toiseen. Lisäksi järjestelmän täytyy osata asettaa uudet luodut oliot sopivalle säikeelle.

## 4.6 Yhteenveto

Tässä luvussa esiteltiin kolme pääasiallista ratkaisua pelimoottorin rinnakkaisohjelmointiin. Ensimmäinen on pelimoottorin komponenttien sisäinen säikeistys, joka on mahdollista vain tietyille komponenteille, ja silloinkin vain rajoitetusti. Toinen on tehtäväperustainen rinnakkaistus, josta käsiteltiin kahta eri mallia. Tehtäväperustaisen rinnakkaistuksen mallit pohjautuvat rinnakkaisten tehtävien löytämiseen pelimoottorista, ja näiden tehtävien sijoittamiseen rinnakkain suoritettaviin säikeisiin. Asynkroninen malli eroaa synkronisesta mallista siinä, että asynkroninen ei ole sidottu pelisilmukkaan, jolloin sen rinnakkaiset toiminnot voivat päivittyä eri nopeuksilla. Kolmas ratkaisu on olioperustainen rinnakkaistus, jossa ajatuksena on siirtää eri säikeisiin eri olioille suoritettavat toiminnot.

# 5 MONISÄIKEISEN PELIMOOTTORIN MALLIEN VERTAILU

Tässä luvussa analysoidaan ja vertaillaan edellisessä luvussa esitetyistä pelimoottorin säikeistyksen ratkaisuista kolmea. Nämä ovat asynkronisen ja synkronisen tehtäväpohjaisen rinnakkaistuksen sekä oliopohjaisen rinnakkaistuksen mallit. Komponenttien sisäinen monisäikeistys jätetään vertailusta pois, sillä tavoitteena on tutkia ratkaisuja, jotka käsittävät koko pelimoottorin toiminnan. Vertailtavat tekijät ovat mallin odotettu nopeus ja mallin laadulliset ominaisuudet. Pelimoottorin toiminnan nopeus on suoraan näkyvissä pelaajille, ja se on hyvin tärkeä kriteeri valittaessa pelimoottorin arkkitehtuuria. Laadullisiin ominaisuuksiin kuuluvat sellaiset kuten arkkitehtuurin päivitettävyys, muunneltavuus, skaalautuvuus ja käyttöönoton monimutkaisuus. Vertailussa on tärkeää, että se on yleistettävissä eri tyyppisiin pelimoottoreihin. Koska eri pelimoottoreissa voi olla hyvinkin erilaisia komponentteja, ei vertailua varten kiinnitetä mitään erityistä komponenttikonfiguraatiota, vaan joitakin tyypillisimpiä komponentteja käytetään vain esimerkinomaisesti.

Rinnakkaissovelluksen suorituskyvyn vertailuun on olemassa useita menetelmiä. Esimerkkeinä voidaan mainita Amdahlin (1967) laskukaava, jonka syötteenä toimii ainoastaan sovelluksen rinnakkaisuuden aste sekä prosessorien määrä. Toista ääripäätä edustaa Adven ja Vernonin (2004) esittämä monimutkainen task graph analysis -menetelmä, joka erityistä edeltäjyysgraafia käyttäen mallintaa sovelluksen suorituskykyä algoritmisesti ottaen huomioon monenlaisia tekijöitä, kuten esimerkiksi viestintään käytettävän menetelmän hitauden. Kummankaan käyttäminen ei ole tässä tapauksessa mahdollista, sillä jälkimmäinen edellyttää käytettävän metriikoita rinnakkaistettavien osien suorituskyvyn selvittämiseen, ja ensimmäinen edellyttää tarkkoja matemaattisia arvioita siitä, kuinka suuri osa ohjelmistosta on rinnakkaistettavissa. Periaatteessa olisi mahdollista antaa arvauksiin pohjautuvia arvioita yksittäisten komponenttien suoritusajoista, mutta todellisuudessa toimintojen suhteelliset kestoajat riippuvat hyvin paljon pelimoottorin toteutuksesta ja jopa käytössä olevasta lait-

teistosta. Tästä näkökulmasta eri pelimoottorin rinnakkaistuksen mallien suorituskykyä ei voida arvioida formaalein matemaattisin keinoin, vaan täytyy turvautua analyysin keinoihin.

## 5.1 Arvioinnin menetelmä

Vertailun tarkoituksena ei ole löytää yhtä parasta mallia tiettyyn käyttötarkoitukseen, ovathan jokaisen pelimoottorin tarpeet erilaiset. Vertailua ei näin ollen kannata toteuttaa tapaustutkimuksena, sillä se rajoittaisi väistämättä tulosten yleistettävyyttä. Mallien ominaisuuksien vertailuun ei ole myöskään järkevää käyttää arviointikehikkoa, koska mallien toiminnasta ei ole kokemuksia, eikä malleista ole suoraan nähtävissä riittävästi laadullisia ominaisuuksia. Sopiva lähtökohta sekä mallien laadullisten ominaisuuksien että suorituskyvyn analysointiin löytyy ohjelmistoarkkitehtuureista.

Kuten johdannossa todettiin, on malleja mahdollista ajatella arkkitehtuurimalleina. Valitettavasti arkkitehtuurimallien keskinäiseen vertailuun ja analysointiin ei ole olemassa suoria menetelmiä. Koska arkkitehtuurimallit ovat kuitenkin vain yhden abstraktioasteen korkeammalla kuin varsinaiset arkkitehtuurit, voidaan mallien analysointiin käyttää tietyin varauksin arkkitehtuurien analysointiin käytettäviä menetelmiä. Erona on, että mallien ominaisuuksia analysoitaessa joudutaan ensin päättelemään, millaisiin arkkitehtuuritason ratkaisuihin mallien soveltaminen johtaa.

Dobrica ja Niemelä (2002) ovat tehneet katsauksen arkkitehtuurien analyysin menetelmistä. Heidän suosituksiensa pohjalta vertailuun parhaiten soveltuvat menetelmät olisivat Kazman et al. (1998) esittämä Architecture Tradeoff Analysis Method, eli ATAM sekä Kazman et al. (1994) esittämä Software Architecture Analysis Method, eli SAAM. ATAM on menetelmänä jonkin verran hienostuneempi ja monimutkaisempi kuin SAAM, ja ATAM soveltuu paremmin erillisten laatuattribuuttien arvioimiseen. Sovellettavaksi menetelmäksi valitaan SAAM, jonka valintaa puoltaa sen yksinkertaisuus ja ymmärrettävyys.

SAAM sisältää seuraavat viisi askelta. Lähteenä Kazman et al. (1994).

1. Kuvaa kohdealueen toiminnallinen jako. Tarkoitus on löytää yhteinen käsitys siitä, mitä järjestelmä tekee.

2. Kartoita toiminnallisuus järjestelmän rakenteeseen. Järjestelmän toiminnallisuus, rakenne, ja toiminnallisuuden kartoitus rakenteeseen ovat kolme erillistä näkökulmaa tietojärjestelmän arkkitehtuuriin.

3. Valitse laatuattribuutit joiden mukaan järjestelmää analysoidaan.

4. Valitse skenaariot joiden avulla testataan valittuja laatuattribuutteja.

5. Arvioi, kuinka hyvin kukin arkkitehtuuri tukee valittuja skenaarioita.

Järjestelmän toiminnallisuus on kuvattu luvussa 3, missä on tarkemmin kuvattu pelisilmukan toimintaa ja pelimoottorin komponenttien tehtäviä. Askeleen 2 mukaisesti mallien rakennetta on tarkennettu luvussa 4, missä määrämuotoisen mallintamisen sijaan on painotettu sanallista kuvausta.

Skenaarioiden käsittely on jaettu kahteen osaan. Kohdassa 5.2 käsitellään suorituskykyyn liittyviä skenaarioita, ja kohdassa 5.3 laadullisiin tekijöihin liittyviä skenaarioita. Skenaarioiden valintaa on pyritty perustelemaan niitä esitellessä. SAAM-menetelmän eräs heikkous on siinä, että ei ole mahdollista tietää, kattavatko kehitetyt skenaariot kaikki mahdolliset kiinnostavat tapaukset. Niinpä on vaikea sanoa mitään varmaa arkkitehtuurimallien käyttäytymisestä käsiteltyjen skenaarioiden ulkopuolelta.

## 5.2 Suorituskyvyn arviointi

### 5.2.1 Skenaariot

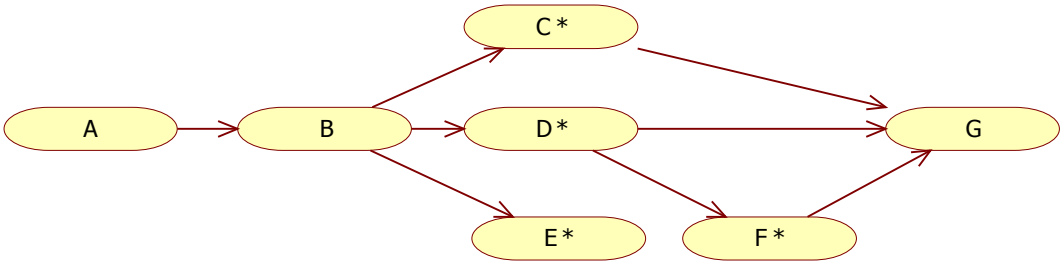
Normaalisti pelimoottorien suorituskykyä voidaan arvioida ruudunpäivityksen nopeudesta (frames per second), eli siitä, kuinka nopeasti yksi pelisilmukan kierros käydään läpi. Jos vertailtaisiin vain pelisilmukkaan pohjautuvia malleja, riittäisi vertailukohdaksi pelisilmukan kierros. Koska asynkronisen tehtäväpohjaisen rinnakkaistuksen malli ei ole sidottu pelisilmukkaan, täytyy löytää yleisemmällä tasolla oleva käsite, joka mittaa pelin suorituskykyä. Pelin suorituskyky näkyy pelaajalle suoraan animaation sujuvuutena, eli kuinka usein animaatio päivittyy jossain ajassa. Toinen usein unohdettava tekijä on pelin vasteaika pelaajan komennoille, eli kuinka kauan kestää esimerkiksi kääntymistä ohjaavan nappulan painamisesta hahmon kääntymiseen. Näistä tekijöistä voidaan johtaa pelin suorituskykyä mittaava suure, ruudunpäivityskierros. Määritellään, että *ruudunpäivityskierros on se aika, joka pelimoottorissa kuluu pelaajan suorittamasta toiminnosta siihen, että toiminnon seuraus voi näkyä ruudulla*. Ruudunpäivityskierros mittaa siis sekä pelin animaation sujuvuutta pelaajan näkökulmasta että pelin vasteaikaa pelaajan toimille.

Merkittävä ruudunpäivityksen nopeuteen vaikuttava seikka on se, kuinka monta oliota pelissä on. Jos esimerkiksi fysiikkakomponentin toiminnassa jokainen olio voi törmätä jokaisen muun olion kanssa minä hetkenä hyvänsä, on fysiikkakomponentin kuluttama aika sidoksissa olioiden määrän neliöön. Tietenkin, koska oliot todennäköisemmin voivat törmätä vain lähellä oleviin olioihin, voidaan toimintaa optimoida jonkin verran. Tulevaisuudessa olioiden määrä tulee arvatenkin kasvamaan, joten arkkitehtuurin täytyy toimia sujuvasti myös suurilla olioiden määrillä.

Nopeuteen vaikuttaa myös käytössä olevien prosessoriytimien määrä, sillä arkkitehtuurit skaalautuvat eri lailla eri säiemäärille. Yksinkertaisuuden vuoksi tarkastellaan vain kahta tapausta, nykyisin käytössä olevaa tuplaydinprosessoria sekä joskus tulevaisuudessa mahdollisesti käytettävissä olevaa hypoteettista 100-ytimistä suoritinta.

Edellä esitetystä johdetaan seuraavat skenaariot:

1. Ruudunpäivityskierros kun pelissä on 100 oliota ja käytössä on 2 ydintä.
2. Ruudunpäivityskierros kun pelissä on 10 000 oliota ja käytössä on 2 ydintä.
3. Ruudunpäivityskierros kun pelissä on 100 oliota ja käytössä on 100 ydintä.
4. Ruudunpäivityskierros kun pelissä on 10 000 oliota ja käytössä on 100 ydintä.
5. Nopeus Amdahlin lain (kaava 1) mukaan laskettuna, kun ydinmääräksi asetetaan 2 sekä 100, ja pelimoottorin suorittamat toiminnot ja toimintojen väliset suhteet on annettu kuvion 13 mukaisesti. Kaikkien toimintojen oletetaan vievän saman ajan.

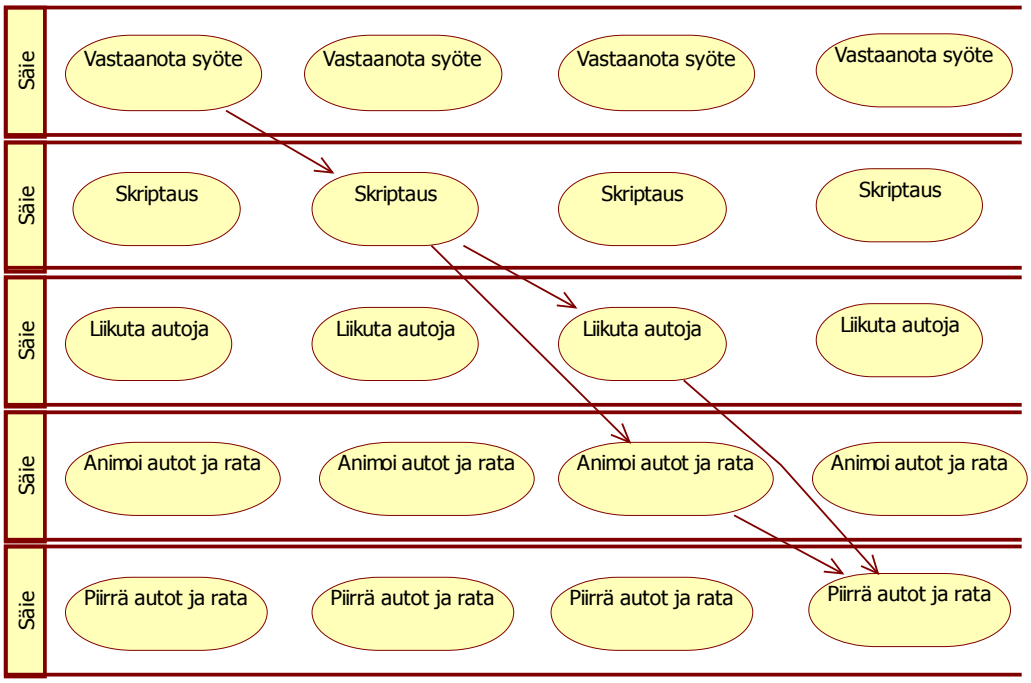



*Kuvio 13: Skenaarion 5 toiminnot. Tähdellä on merkitty toiminnot, jotka voidaan rinnakkaistaa tietoperustaisesti.*

### 5.2.2 Asynkroninen tehtäväpohjainen rinnakkaistus

Asynkroninen malli on siitä mielenkiintoinen, että periaatteessa sen kaikki tehtävät ovat rinnakkaistettuja. Jos prosessorissa vain on riittävästi tehoa, piirtyy näytölle yksi ruutu niin nopeasti tai hitaasti, kuin ruudulle piirtävä säie on määritelty päivittymään. Optimaalinen tapaus olisi sellainen, että syöte vastaanotetaan ennen fysiikan laskemista, joka valmistuu ennen näytölle piirtämistä. Kuvio 14 havainnollistaa suorituksen kulkua optimaalisessa tapauksessa, käytettäessä esimerkkinä yksinkertaista autopeliä. Optimitapauksessa eri ruudunpäivityskierroksen toiminnot valmistuvat peräk-

käin. Kuviosta huomataan myös, että fysiikan laskenta, eli autojen liikuttaminen, ja animointi, ovat loogisesti rinnakkaisia tehtäviä. Siksi ruudunpäivityskierroksen pituudeksi tulee optimitapauksessa neljä toimintoa.




*Kuvio 14: Asynkronisen tehtäväpohjaisen rinnakkaistuksen optimaalinen ruudunpäivityskierros*

Jos säikeiden välillä ei ole minkäänlaista synkronisaatiota, esimerkiksi niin, että ruudulle piirtäminen odottaisi fysiikan laskennan valmistumista, tai että fysiikan laskeminen odottaisi syötettä, voi pelin toiminnossa esiintyä merkittäviä viiveitä. Pahimmassa tapauksessa näytölle piirto valmistuu juuri ennen fysiikan laskennan valmistumista, ja fysiikan laskenta valmistuu vaikkapa juuri ennen syötteen vastaanottamista. Tällöin viive siitä, että pelaaja painaa nappulaa, siihen että pelaaja näkee nappulan painalluksen vaikutuksen ruudulla, pitenee liki kaksinkertaiseksi optimitapaukseen verrattuna. Jos ajatellaan kaikkien kuviossa 14 olevien toimintojen vievän yhden aikayksikön, pitenisi ruudunpäivityskierron tällöin neljästä yksiköstä kahdeksaan yk-

sikköön. Gabb ja Lake (2005) pyrkivät minimoimaan pahimman tapauksen viivettä ehdottamalla, että jotkin säikeet päivittyisivät useammin kuin muut säikeet, jolloin esimerkiksi olisi todennäköisempää, että syöte päivittyisi ennen fysiikan laskemista.

Mallin vahvuus on siinä, että koska rinnakkaiset tehtävät eivät ole keskenään synkronisoituja, voidaan prosessoriaikaa käyttää tehokkaasti hyväksi ajamalla kaikkia toimintoja yhtäaikaisesti. Sovelluksen rinnakkaistusasteeksi muodostuu tällöin hyvin korkea, jopa lähelle sataa prosenttia. Periaatteessa jos vain käytössä on riittävästi suoritinytimiä, voidaan mallin mukaisella säikeistyksellä saavuttaa liki lineaarinen nopeutus säikeiden määrään nähden. Tämä tietysti edellyttää, että sovelluksesta erotettavat säikeet olisivat likimain yhtä vaativia suoritusajan suhteen. Valitettavasti näin ei yleensä ole, vaan tekoälyn ja fysiikan laskeminen sekä grafiikan piirtäminen saattavat viedä leijonanosan suoritinajasta. Niinpä on syytä pohtia, onko esimerkiksi syötteen vastaanottamisen laittamisesta omaan säikeeseensä mitään hyötyä.

Mallin heikkous suorituskyvyn suhteen piilee edellä mainitussa suuressa ruudunpäivityskierroksen pituuden vaihtelussa, ja pahimman tapauksen pituudessa. Viiveen ja sen vaihtelun vaikutusta verkkopeleissä ovat tutkineet muun muassa Claypool ja Nichols (2004). Heidän tutkimuksensa mukaan viive käyttäjän suorittaman toiminnon ja sen ruudulla näkymisen välillä voi olla jopa reilusti yli sata millisekuntia ennen kuin se alkaa häiritä pelaajaa. Käytännössä viiveestä tuskin muodostuu noin suurta pahimmassakaan tapauksessa. Jos ruudunpäivitysnopeus muodostuu riittävän suureksi, ei pelaaja voi havaita kierroksen pituuden vaihtelua. Mallin käyttäytymistä todellisessa toteutuksessa olisi kuitenkin syytä tutkia tarkemmin.

**Skenaariot**

1 ja 2 Koska tehtävät voi jakaa kahdelle eri säikeelle, voidaan periaatteessa saavuttaa kaksinkertainen nopeus yksisäikeiseen verrattuna. Olioiden määrä vaikuttaa suorituskykyyn suoraan yksittäisten komponenttien suorituskyvyn mukaisesti.

**Skenaariot**

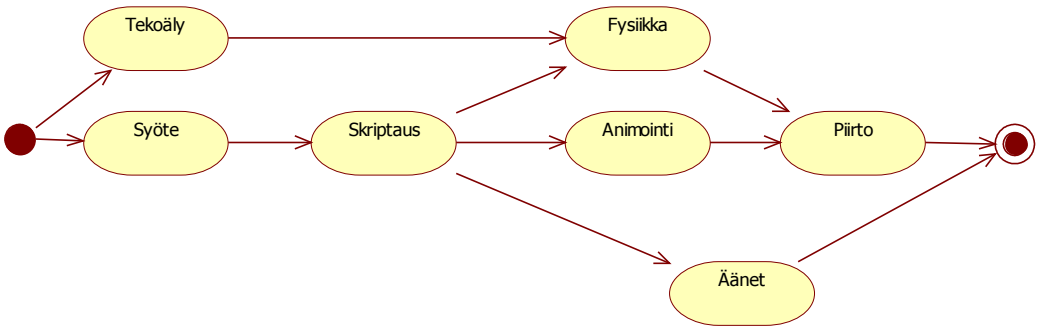
3 ja 4 Koska pelimoottorin tehtäviin jakamisessa on rajansa, ei voida saada käyttöön kaikkia prosessoriytimiä. Mahdollisten rinnakkaisten tehtävien määrä riippuu tietysti pelimoottorista, ja sitä voidaan kasvattaa jakamalla tehtävät rinnakkain suoritettaviin alitehtäviin. Tämä kuitenkin kasvattaa merkittävästi järjestelmän monimutkaisuutta, joten arkkitehtuurin rajat tulevat todellisuudessa vastaan jossain vaiheessa. Suorituskyky ei tämän rajan saavuttamisen jälkeen enää nouse prosessoriytimiä lisäämällä. Olioiden määrä vaikuttaa tässäkin tapauksessa suorituskykyyn suoraan yksittäisten komponenttien suorituskyvyn mukaisesti.

5 Koska asynkronisen mallin mukaisesti tehtävät eivät ole toisistaan riippuvaisia ajan suhteen, voidaan jokainen asetettu tehtävä suorittaa omassa säikeessään. Tällöin kerroin f on 0. Kun ytimiä on kaksi, antaa Amdahlin laki tulokseksi suoraan kaksinkertaisen nopeutuksen. Kun ytimiä on 100, voidaan rinnakkain suorittaa vain seitsemän tehtävää, koska tehtäviä ei ole enempää. Tällöin Amdahlin lain mukaan voitaisiin odottaa seitsenkertaista nopeutusta.

Kuten edellä on mainittu, voi pahimmassa tapauksessa asynkroninen malli tuottaa liki kaksi kertaa optimaalista tapausta hitaamman tuloksen, eli huonoimmassa tapauksessa tulokset ovat 1 ja 3,5. Keskimääräinen tulos on jossain parhaan ja huonoimman tuloksen välissä.

### 5.2.3 Synkroninen tehtäväpohjainen rinnakkaistus

Synkronoidun mallin mukaisesti pelimoottorin eri tehtävien valmistuminen on sidottu yhteen pelisilmukan kierrokseen. Koko kierroksen pituus voidaan laskea rinnakkaisten tehtävien kriittisen polun, eli pisimmän polun, pituudesta. Kuvion 15 esimerkissä rinnakkaisia polkuja ovat tekoäly->fysiikka->piirto, syöte->skriptaus->animointi->piirto, ja syöte->skriptaus->äänet. Koko kierroksen pituus riippuu siis siitä, mikä näistä poluista kestää pisimpään.




*Kuvio 15: Rinnakkaisia polkuja synkronisessa tehtäväpohjaisessa mallissa.*

Kun kriittinen polku tiedetään, sitä voidaan yrittää lyhentää siirtämällä sen tehtäviä muihin rinnakkaisiin polkuihin. Mallin rajoittava piirre on se, että rinnakkain voidaan asettaa vain todella toisistaan riippumattomia tehtäviä. Tehtävien järjestykselläkin on merkitystä kun jokin tehtävä odottaa syötteenään tietyn tehtävän tuloksia. Tämä rajoittaa tehokkaasti yhtäaikaisten rinnakkaisten tehtävien määrää. Edellisen esimerkin mukaisessa pelimoottorissa on helposti löydettävissä vain parhaimmillaan kolme rinnakkaista tehtävää: fysiikka, animointi ja äänet. Lisäksi täytyy ottaa huomioon, että jos kaikkein hitaimmat toiminnot täytyy edelleen suorittaa peräkkäin, mutta nopeasti toteutettavat toiminnot siirretään rinnakkain ajettavaksi, ei malli eroa yksisäikeisestä kuin rinnalle siirrettyjen tehtävien nopeuden osalta. Niinpä malli on rinnakkaistusasteeltaan pienempi kuin asynkroninen malli. Se, kuinka paljon pienempi se on, riippuu täysin pelimoottorista.

Hyvänä puolena suorituskyvyn suhteen voidaan pitää tasaista ruudunpäivitysnopeutta. Mallissa ei ole asynkronisen mallin tapaan sisäsyntyisiä vaihteluita kierroksen pituudessa.

**Skenaariot**

1 ja 2 Malli toimii kahdella prosessoriytimellä pienemmällä rinnakkaistuksen asteella kuin asynkroninen malli. Merkittäväksi ongelmaksi jää kriittisen suorituspolun pituus, joka käytännössä riippuu suoraan olioiden määrästä.

3 ja 4 Synkroninen tehtäväpohjainen jako skaalautuu huonosti suurille säiemäärille, sillä johtuen synkronisoinnin tarpeesta on rinnakkaisten tehtävien määrä rajallisempi kuin asynkronisessa mallissa.
Suorituskyky ei tämän rajan saavuttamisen jälkeen enää nouse prosessoriytimiä lisäämällä.
Olioiden määrä vaikuttaa suorituskykyyn suoraan yksittäisten komponenttien suorituskyvyn mukaisesti.

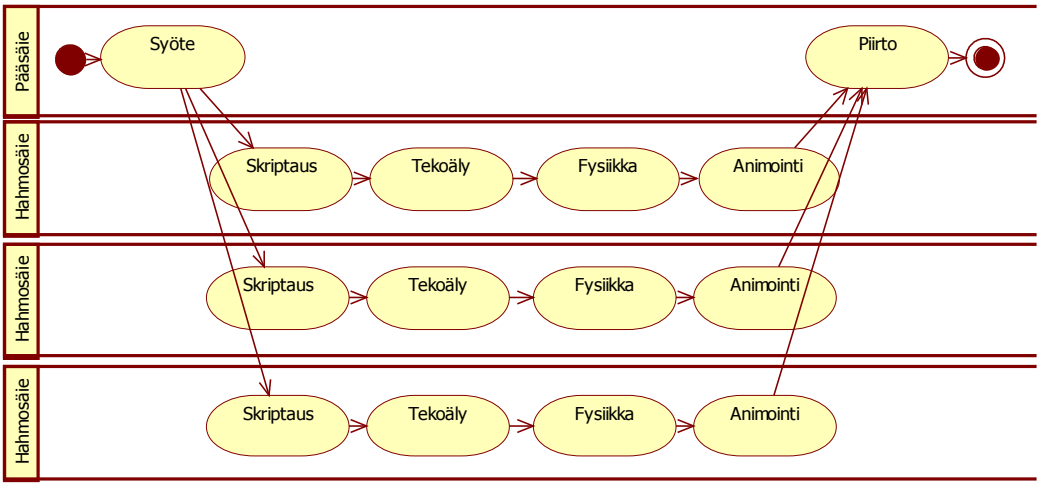
5 Kuvion 13 tehtävistä ainoastaan C, D, E ja F voidaan suorittaa rinnakkain. Kahdella ytimellä siis esimerkiksi ensin C ja D, ja sitten E ja F. Tässä tapauksessa f on 3/7, jolloin odotettavissa olevan nopeus olisi 1.4 kertainen. Tulos ei ole huono verrattuna asynkroniseen malliin, jonka nopeutus on jossain 1 ja 2 välissä.

Sadalla ytimellä mallin heikkous tulee selkeästi esiin. Kuvion 13 tehtävistä on mahdollista suorittaa tehtävät C, D ja E yhtä aikaa, mutta tällöin tehtävä F jouduttaisiin suorittamaan yksin. Tällöin f on 4/7, ja koska rinnakkain suoritettava osa on mahdollista jakaa vain kolmeen osaan, tulee tulokseksi myös 1.4.

### 5.2.4 Olioperustainen rinnakkaistus

Oliopohjaisen rinnakkaistuksen malli on myös sidottu pelisilmukan kierrokseen. Erona tehtäväpohjaiseen jakoon on se, että jako tapahtuu pelin olioiden suhteen. Malli on sidottu pelisilmukkaan, joka toisaalta tarkoittaa että mallissa ei ole asynkronisen mallin tapaan sisäsyntyisiä vaihteluita kierroksen pituudessa, mutta toisaalta jotkin pelisilmukan tehtävät täytyy suorittaa peräkkäin. Kuviossa 16 on esimerkki, jossa kaikki tehtävät paitsi syötteen hakeminen ja piirtäminen on sijoitettu rinnakkaisiin säikeisiin. Pelisilmukan kierroksen pituudeksi tulee tässäkin mallissa pisimmän po-

lun pituus. Optimaalisessa tilanteessa kaikki olioita käsittelevät säikeet vievät saman ajan, joka olisi siis kääntäen verrannollinen rinnakkaisten säikeiden määrään. Kuvion esimerkissä piirtämistä ei ole sijoitettu oliosäikeiden tehtäväksi, koska näytönohjaimelle piirtäessä täytyy joka tapauksessa tietää kaikkien olioiden tila.




*Kuvio 16: Olioperustainen rinnakkaistus kolmella oliosäikeellä.*

On myös mahdollista, että oliosäikeet eivät ole tasapainotettuja, jolloin jonkin säikeen suorittaminen vie suhteellisesti suuremman ajan kuin muiden. Tällöin pahimmassa tapauksessa suorituskyky olisi sama kuin yksisäikeisellä pelimoottorilla. Tasapainotukseen onkin mallissa kiinnitettävä erityistä huomiota. Toisena hitautta mahdollisesti lisäävänä tekijänä on tarve synkronisointiin säikeiden välillä esimerkiksi fysiikan tai tekoälyn laskennassa. Monissa tapauksissa tarvitaan kuitenkin tietää ainoastaan olioiden paikat edellisellä ruudunpäivityskierroksella, jolloin synkronisoinnille ei ole tarvetta. Kolmas suorituskykyyn vaikuttava asia on se osa pelisilmukasta, jota ei ole rinnakkaistettu. Tähän täytyy kuulua ainakin syötteen hallinta ja piirtäminen. Tätä voidaan pyrkiä minimoimaan esimerkiksi aloittamalla jonkin muun tehtävän suorittaminen, kun yksi säie piirtää kuvaa.

**Skenaariot**

1 Periaatteessa kahdella prosessoriytimellä voidaan jakaa sadan olion kuorma kahteen säikeeseen tasan, jolloin saavutetaan melkein kaksinkertainen nopeus yksisäikeiseen malliin verrattuna.

2 Jos olioita on 10 000, tulisi tällöin kumpaankin oliosäikeeseen 5000 oliota, jolloin nopeus olisi edelleen liki kaksinkertainen yksisäikeiseen malliin verrattuna.

3 Jos käytössä on sata prosessoriydintä ja sata oliota, voitaisiin tietysti jokaiselle oliolle osoittaa oma säikeensä. Tällöin voisi kuitenkin pelkästä säikeiden käyttämisen aiheuttamasta kuormasta muodostua suhteettoman suuri verrattuna yksittäisen säikeen tehtävän kokoon. Jos olioita on kovin vähän, ei toisaalta rinnakkaistuksestakaan ole merkittävää hyötyä.

4 10 000 oliolla ja sadalla ytimellä voitaisiin asettaa sata oliota jokaiselle säikeelle, jolloin nopeus olisi sama kuin yhdellä ytimellä sataa oliota käytettäessä. Suorituskyvyn puolesta olioperustainen rinnakkaistus soveltuukin erinomaisesti suurille oliomäärille, ja suurille prosessoriydinmäärille.

5 Kuvion 13 tehtävistä C, D, E ja F voidaan rinnakkaistaa tietoperustaisesti. Jos oletetaan, että pelissä on olioita vähintään ydinten määrä, saadaan f arvoksi 3/7. Tämä tuottaa ydinten määrällä 2 tuloksen 1,4, ja ydinten määrällä 100 tuloksen 2,3.

Tulosta tulkitessa on otettava huomioon, että normaalisti pelimoottorissa iso osa tehtävistä on jollain tapaa mahdollista rinnakkaistaa tietoperustaisesti. Tällöin odotettavissa oleva nopeutus on suurempi.

### 5.2.5 Suorituskyvyn arvioinnin yhteenveto

Kaikki analysoidut mallit soveltuvat suorituskykynsä puolesta rinnakkaistukseen kun kohdejärjestelmän prosessorissa on vain muutamia ytimiä. Luultavasti tällöin parhaimpaan suorituskykyyn pystyy asynkronisen rinnakkaistuksen malli, johtuen siitä,

että se ei juurikaan tarvitse synkronisaatiota säikeidensä välillä. Mallin haittapuolena on mahdollinen ruudunpäivityskierroksen pituuden vaihtelu, jota ei muilla pelisilmukkaan sidotuilla malleilla voi esiintyä.

Molemmat tehtäväpohjaisen rinnakkaistuksen mallit sisältävät suorituskyvyn nousulle ylärajan, joka tulee vastaan, kun rinnakkaisia tehtäviä ei ole enää mahdollista löytää enempää. Milloin raja tulee vastaan riippuu pelimoottorista, mutta todennäköisesti on vaikeaa löytää muutamaa rinnakkaista toimintoa enempää. El Rhalibi et al. (2005) ovat testanneet esittämäänsä mallia (tässä nimellä synkronoitu tehtäväpohjainen rinnakkaistus) prototyypin ja simuloidun moniydinjärjestelmän avulla. Verrattuna yksisäikeiseen sovellukseen he ovat kahdella säikeellä saaneet yli 80 prosentin nopeusedun. Neljällä säikeellä etu on ollut enää 130 prosenttia. Tämä tukee näkemystä tehtäväpohjaisten mallien nopeasti saavutettavasta suorituskyvyn ylärajasta. Suorituskyvyn ylärajan vastaan tulemista nopeuttaa myös se, että tehovaatimuksiltaan pieniä tehtäviä ei edes kannata sijoittaa rinnan suoritettavaksi. Niinpä esimerkiksi näppäimistösyötteen lukemista ei välttämättä kannata laskea rinnakkaiseksi tehtäväksi.

Olioperustaisen rinnakkaistuksen mallin rinnakkaistuksen yläraja tulee vastaan, kun rinnakkaisia olioita käsitteleviä säikeitä ei ole enää järkevää asettaa käsittelemään pienempää määrää olioita. Koska pelien oliomäärä kuitenkin kasvaa kaiken aikaa, voidaan sanoa, että yläraja nousee jatkuvasti.

Kumpikaan tehtäväpohjaisen rinnakkaistuksen malleista ei toimi hyvin suurilla olioiden määrillä, sillä määrän kasvaminen kasvattaa jatkuvasti yksittäisten säikeiden kuormaa. Olioperustaisen rinnakkaistuksen malli puolestaan voi jakaa suurten oliomäärien kuorman useille eri säikeille. Jos käytössä on vain muutama prosessoriydin, mallien välille ei tule todennäköisesti eroa suorituskyvyssä isoillakaan oliomäärillä. Jos käytössä on puolestaan kymmeniä tai satoja prosessoriytimiä, on olioperustainen rinnakkaistus tehokkain malli.

Vaikka olioperustainen rinnakkaistus näyttää ylivoimaisesti suorituskykyisemmältä mallilta suurilla oliomäärillä, ei sovi unohtaa, että tehtäväpohjaisten mallien suorituskykyä on mahdollista lisätä jakamalla yksittäiset tehtävät olioperustaisesti. Tällainen hybridimalli on täysin mahdollinen, vaikkakin luultavasti monimutkaisempi toteuttaa.

## 5.3 Laadun arviointi

### 5.3.1 Skenaariot

Pelimoottorin kehitystä voidaan nopeuttaa hankkimalla osa pelimoottorin komponenteista valmiina. Mahdollisuus käyttää tällaista komponenttia saattaa olla ratkaiseva tekijä pelimoottorin arkkitehtuurimallia valittaessa, etenkin jos komponentin toteuttaminen itse ei ole mahdollista esimerkiksi resurssien puutteen vuoksi. Kaikki valmiit komponentit eivät toimi monisäikeistettynä, joten on syytä tutkia, miten arkkitehtuuri tukee tällaisia komponentteja. Tarkoituksena oli alun perin ottaa tutkittavaksi skenaarioksi jonkin kaupallisen fysiikkakomponentin käyttö, mutta viime aikoina kaupalliset fysiikkakomponenttien valmistajat ovat lähteneet aktiivisesti tukemaan moniytimisyyttä. Niinpä tutkittavaksi skenaarioksi otetaan epäkaupallisen yksisäikeiseen fysiikan mallinnukseen suunnitellun Newton Game Dynamicsin käyttö pelimoottorissa.

Vaikka nykyään moniydinprosessoreissa on vain muutamia ytimiä, kasvaa ytimien määrä tulevaisuudessa luultavasti jopa satoihin. Toisaalta nykyaikaisissa tietokoneissa on yhä edelleen vain yksiytimisiä prosessoreja. Siksi olisi eduksi, jos arkkitehtuuri skaalautuisi hyvin eri ydinmäärille yhdestä eteenpäin. Tästä johdetaan skenaario 2: Pelimoottorin pitää skaalautua koneen prosessoriydinmäärälle käynnistyessään.

Monisäikeistys voi asettaa rajoituksia viestintään pelimoottorin eri komponenttien välillä. Säikeistyksen tehokkuuden näkökulmasta viestintää pitäisi olla mahdollisimman vähän, koska tämä edellyttää hidasta synkronointia yhteisen muistialueen käytössä (Ji, Felten & Li, 1998). Säikeiden välisen viestinnän määrä ja laatu korreloi myös kustannusten ja mahdollisten ohjelmointivirheiden kanssa. Tästä johdetaan skenaariot 3 – pelaaja painaa nappulaa ohjatakseen hahmoaan, ja 4 – skripti luo kaksi oliota fysiikkakomponentille tyhjään kohtaan pelimaailmaa.

Usein pelit toteutetaan yhtäaikaisesti sekä konsolialustoille että PC:lle. Monien komponenttien sisäistä toteutusta joudutaan väistämättä vaihtamaan eri alustoille. Toivottavaa olisi kuitenkin, että pelimoottorin perusarkkitehtuuriin ei tarvitsisi tehdä muutoksia. Tästä johdetaan skenaario 5: Peli toteutetaan monialustaisesti sekä PC, Xbox

360, että Playstation 3 alustoille. Oletusarvoisesti kaikki mallit soveltuvat toteutettaviksi PC-laitteilla, sillä rajoituksella, että PC-prosessorissa voi olla vaihtelevia määriä prosessoriytimiä. Huomautuksena on mainittava, että Xbox 360:n prosessorista käytetään tässä yhteydessä nimitystä Xenon, Playstation 3:n prosessorista nimitystä Cell, ja PC-prosessoreista ei käytetä erityistä nimitystä.

Skenaariot:

1. Pelimoottoriin asetetaan Newton Game Dynamics -fysiikkakomponentti.
2. Pelimoottorin pitää skaalautua koneen prosessoriydinmäärälle käynnistyessään.
3. Pelaaja painaa nappulaa ohjatakseen hahmoaan.
4. Skripti luo kaksi oliota fysiikkakomponentille tyhjään kohtaan pelimaailmaa.
5. Peli toteutetaan monialustaisesti sekä PC, Xbox 360, että Playstation 3 alustoille.

### 5.3.2 Asynkroninen tehtäväpohjainen rinnakkaistus

**Skenaariot**

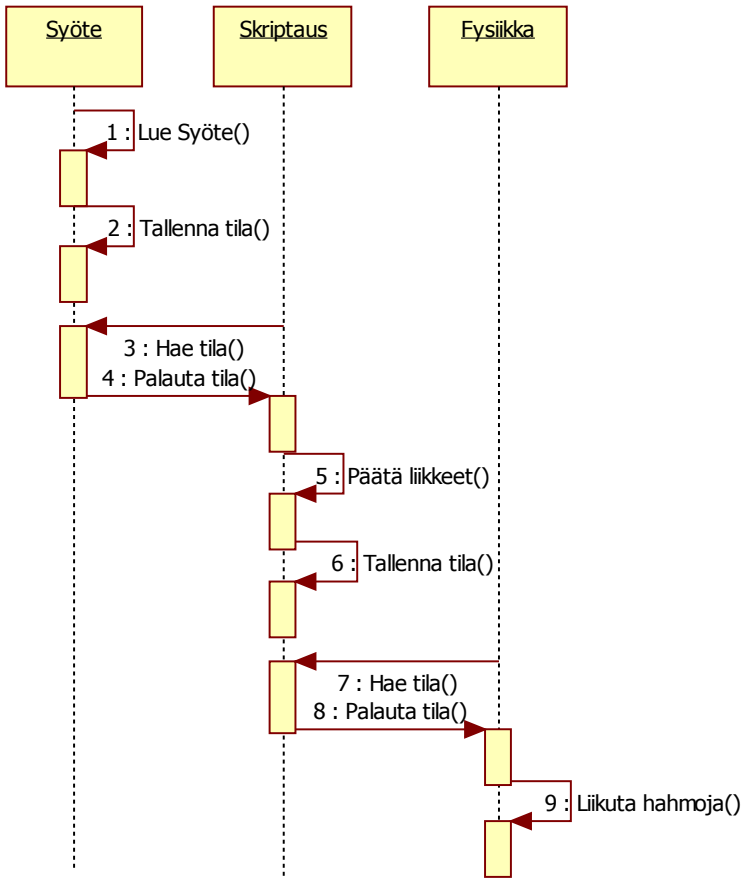
1 Newton Game Dynamics toimii mallin mukaisessa pelimoottorissa ilman erityisiä muutoksia. Ainoa huomioitava seikka on, että komponentilta tulevat tulokset, eli olioiden sijainnit ja tiedot olioiden törmäyksistä, täytyy koota omaan rakenteeseensa muille komponenteille syöttämistä varten. Muut komponentit eivät siis voi suoraan käydä kysymässä fysiikkakomponentin tietoja, koska komponenttia ei ole välttämättä suunniteltu säieturvalliseksi.

**Skenaariot**

2 Skaalautuminen eri määrille säikeitä on mahdollista muuttamalla joitakin tehtäviä suoritettavaksi peräkkäin. Tämä olisi mahdollista esimerkiksi valitsemalla säikeiden luomisen yhteydessä sopiva etukäteen suunniteltu säieasetelma. Johtuen komponenttien tiloihin perustuvaan viestinvälityksen tehtävien välillä, tehtävien siirtäminen peräkkäin suoritettaviksi ei luultavasti vaatisi muutoksia itse komponenttien toimintaan. Sopivat säiekonfiguraatiot ja niitä hallitsevat ohjelmanosat täytyy suunnitella eri säiemäärille erikseen. Säiemäärällä on myös yläraja, joka tulee vastaan, kun sopivia rinnakkaisia tehtäviä ei enää löydy enempää.

**Skenaariot**

3 Kuvio 17 esittää skenaarion toimintaa UML:n sekvenssikaaviona. Koska mallin mukaisesti komponenttien välinen viestintä täytyy hoitaa pitämällä kirjaa komponentin uusimman päivityksen aikaisesta tilasta, muistuttaa rakenne eräänlaista viestinvälitystä, jossa kaikki viestit lähetetään ja vastaanotetaan kerralla. Viestinnän kannalta tämä on varsin jäykkä rakenne, eikä mahdollista esimerkiksi tilannetta, jossa komponentti haluaa päivittää toisen komponentin tilaa, ja saada vastauksen heti.




*Kuvio 17: Asynkronisen tehtäväpohjaisen rinnakkaistuksen malli - Pelaaja painaa nappulaa ohjatakseen hahmoaan.*

**Skenaariot**

4 Skenaario 4 osoittaa merkittävän ongelman mallin viestinnässä. Koska skriptauskomponentti ei voi olla suorassa yhteydessä fysiikkakomponenttiin, sen täytyy olion luodakseen lähettää sille viesti, johon fysiikkakomponentti reagoi jollain ajan hetkellä. Skriptauskomponentti voi kuitenkin etsiä pelimaailmasta tyhjän paikan olion luomiselle, käyttäen hyväksi fysiikkakomponentin edellistä päivitettyä tilaa. Voisi luulla, että skriptauskomponentin olisi siis mahdollista etsiä tyhjä tila fysiikkamaailmasta, ja lähettää fysiikkakomponentille käskyn luoda siihen paikkaan uusi olio. Tässä piilee kuitenkin vaara, koska tyhjä tila on etsitty fysiikkakomponentin edellisen päivityksen tilanteesta, ja kun viesti olion luomisesta saapuu komponentille, voi paikalle olla siirtynyt joku muu olio. Erityisesti tämä tarkoittaa, että kahden olion yhtäaikainen luominen lähekkäin voi aiheuttaa ongelmia.

Ongelma täytyisi ratkaista ylikuormittamalla fysiikkakomponentin olionluomistoimintoa mahdollistamaan tyhjän tilan etsiminen luotavalle oliolle. Samanlaista ylikuormittamista tulisi käyttää muissa samankaltaisissa tilanteissa, joissa jokin komponentti haluaa päivittää toisen komponentin tilaa ja tarvitsee päivityksen tulokset heti käyttöön. Tässä nimenomaisessa tapauksessa voidaan toki argumentoida, että tällainen ylikuormittaminen on suositeltavampi keino hoitaa olion luominen tyhjään paikkaan maailmassa.

**Skenaariot**

5 Mallin hyödyntäminen Xenon-prosessorilla edellyttää vähintään kolmen rinnakkaisen tehtäväsäikeen löytämistä. Mallista on helppo löytää ainakin kolme tehtävää, joten malli soveltuu käytettäväksi Xenon-prosessorilla.

Asynkroninen tehtäväpohjainen malli soveltuu muita malleja paremmin Cell-prosessorin asymmetriselle tekniikalle. Mallista voi nimittäin olla mahdollista löytää pelimoottorista erityisiä SPE-yksikölle sopivia tehtäviä. Esimerkkinä tästä on äänien prosessointi, joka voidaan eriyttää omaksi rinnakkaiseksi prosessikseen. Koska Cell-prosessorissa on seitsemän SPE-yksikköä, voi prosessorin täysimittaiseen hyödyntämiseen riittää seitsemän tällaisen yksinkertaisen prosessin löytäminen. Pelimoottorista riippuen voi seitsemän SPE:lle sopivan tehtävän löytäminen vaatia kuitenkin jonkin verran mielikuvitusta. Lisäksi jos pyritään käyttämään samoja yksinkertaisia prosesseja PC- ja Xenon-prosessoreille, menee niiden monikäyttöisemmistä ytimistä löytyvä kapasiteetti hukkaan. Ottaen huomioon mallin huonon skaalautuvuuden eri prosessoriydinmäärille, lienee mallin mukaisesta pelimoottorista tehtävä eri versiot eri alustoille.

### 5.3.3 Synkroninen tehtäväpohjainen rinnakkaistus

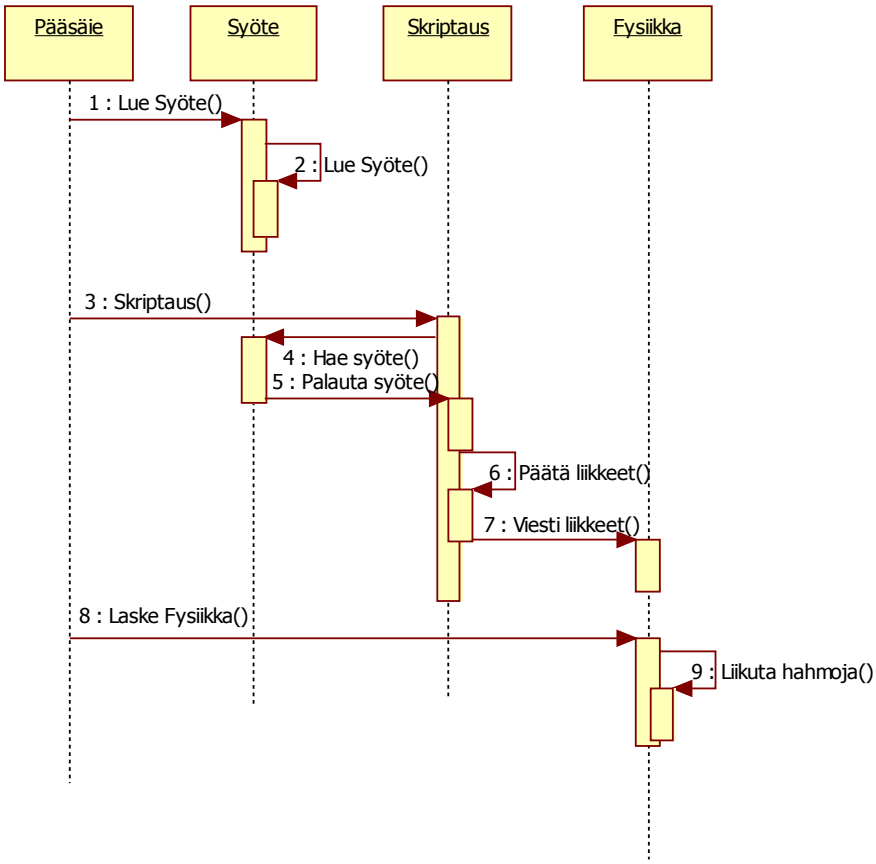
**Skenaariot**

1 Newton Game Dynamics toimii mallin mukaisessa pelimoottorissa ilman erityisiä muutoksia. Koska komponentti ei ole rakennettu säikeistystä silmällä pitäen, täytyy huomioida, että rinnalla toimivista säikeistä ei ole turvallista koskea komponentin tietoihin.

2 Skaalautuminen eri säiemäärille tarkoittaisi koko pelisilmukan kirjoittamista uusiksi jokaiselle eri määrälle. Säiemäärällä on yläraja, joka tulee vastaan, kun sopivia rinnakkaisia tehtäviä ei enää löydy enempää.

**Skenaariot**

3 Komponenttien välinen viestintä synkronisen tehtäväpohjaisen rinnakkaistuksen mallissa ei suuresti eroa yksisäikeisen pelimoottorin viestinnästä. Jos rinnakkaisten säikeiden välillä ei ole tarvetta viestintään, ei eroa ole lainkaan. Kuvio 18 esittää komponenttien välistä viestintää skenaarion 3 tapauksessa. Kuvion mukaisesti esimerkiksi mikään ei estä skriptauskomponenttia kommunikoimasta tietojaan fysiikkakomponentille suoraan. Erona asynkroniseen malliin on siis se, että tässä tapauksessa fysiikkakomponentin ei tarvitse erikseen käydä hakemassa tietoja skriptauskomponentilta. On makuasia kumpaa viestinnän mallia pitää soveltuvampana, mutta synkroninen malli mahdollistaa kiistatta läheisemmät kytkökset komponenttien välillä.




*Kuvio 18: Synkronisen tehtäväpohjaisen rinnakkaistuksen malli - Pelaaja painaa nappulaa ohjatakseen hahmoaan.*

**Skenaariot**

4 Synkronisen tehtäväpohjaisen rinnakkaistuksen mallissa skenaario 4 ei tuota ongelmia, mutta vain jos skriptaus ja fysiikan laskeminen eivät ole rinnakkaisia toimintoja. Tällöin olioiden luonti ja tyhjien paikkojen etsintä voitaisiin tehdä kuten missä tahansa yksisäikeisessä pelimoottorissa. Jos ne puolestaan olisivat rinnakkaisia toimintoja, muodostuisi säikeiden synkronoinnista merkittävä ongelma. Esimerkiksi jos skriptauskomponentti haluaisi lisätä olion fysiikkakomponentille juuri kun fysiikkakomponentti on laskemassa törmäyksiä, aiheutuisi tästä mahdollisesti virheitä komponentin toimintaan. Koska olion lisäys pitäisi tällöin estää, estäisi synkronointi kokonaan skriptauskomponentin suorituksen kunnes fysiikkakomponentti on laskunsa suorittanut.

5 Mallin hyödyntäminen Xenon-prosessorilla ei vaatine muita erikoisratkaisuja kuin säikeiden vähimmäismäärän huomioon ottamisen. Mallin hyödyntäminen Cell-prosessorilla ei puolestaan onnistu kunnolla koko pelimoottorin tasolla. Seitsemän rinnakkaisen tehtävän löytyminen on vielä vaikeampaa kuin asynkronisessa mallissa, sillä rinnakkaisten tehtävien täytyy tässä mallissa sijaita samalla tasolla edeltäjyysgraafissa. Cell-prosessorin hyödyntämiseksi olisikin tehtävä yksittäisiin komponentteihin sisäistä rinnakkaistusta. Mallissa on myös sama ongelma kuin asynkronisessa mallissa - jos pyritään käyttämään samoja Cell-prosessorille tarkoitettuja yksinkertaisia prosesseja PC- ja Xenon-prosessoreille, menee niiden monikäyttöisemmistä ja tehokkaammista ytimistä löytyvä kapasiteetti hukkaan.

### 5.3.4 Olioperustainen rinnakkaistus

Eräs mahdollinen ongelmia tuottava asia oliopohjaisen rinnakkaistuksen mallissa on, että pelin olioihin liittyvät toiminnot täytyy suunnitella erityisesti säikeistystä ajatellen. Esimerkkinä on jo annettu fysiikan mallinnus, joka täytyy pystyä jakamaan sopivan kokoisiin toisistaan mahdollisimman erillisiin osiin. Koska monet kaupalliset

komponentit on suunniteltu toimimaan yksisäikeisesti, saattaa olioperustaisen rinnakkaistuksen käyttö vaatia erityisten monisäikeistykseen sopivien komponenttien tekemistä.

**Skenaariot**

1 Newton Game Dynamicsin asettaminen olioperustaisen rinnakkaistuksen mallin mukaiseen pelimoottoriin on erittäin vaikeaa. Suurin este on, että komponentissa on update-metodi, joka päivittää koko mallinnetun maailman tilaa. Jotta jokaisesta oliosäikeestä voitaisiin päivittää vain kyseisen säikeen olioiden tilaa, täytyisi jokaisella säikeellä olla oma erillinen maailmansa. Tällöin eri säikeiden oliot eivät olisi missään tekemisissä toistensa kanssa. Tätä voitaisiin kiertää toisintamalla alueiden rajoilla olevat oliot vierekkäisten säikeiden fysiikkamaailmaan sekä päivittämällä toisinnettujen olioiden tilaa jatkuvasti säikeiden välillä. Valitettavasti tällaiset viritykset heikentävät suorituskykyä, ja ovat vaikeita toteuttaa. Todennäköisesti samanlaisia ongelmia on odotettavissa myös muissa yksisäikeisyyteen suunnitelluissa komponenteissa.

2 Skaalautuminen eri säiemäärille vaatii, että olioita eri säikeille määräävälle järjestelmälle annetaan tiedoksi haluttu säikeiden määrä. Koska arkkitehtuuri on suunniteltu niin, että säikeissä voi olla vaihteleva määrä olioita, ei mikään estä esimerkiksi muuttamasta järjestelmästä yksisäikeistä laittamalla kaikki pelin oliot samaan säikeeseen. Säiemäärällä ei käytännössä ole ylärajaa, joten malli soveltuu hyvin myös tulevaisuuden mahdollisesti kasvaville prosessoriydinmäärille.

3 Olioperustaisen rinnakkaistuksen mallin mukaisessa pelimoottorissa komponenttien välinen viestintä ei myöskään eroa yksisäikeisen pelimoottorin viestinnästä. Siinä missä synkronisessa tehtäväpohjaisessa mallissa viestintä rinnakkaisten tehtävien välillä on rajoittunutta, on olioperustaisessa rinnakkaistuksessa viestintä eri säikeissä olevien olioiden välillä on rajoittunutta. Olioperustainen rinnakkaistus ei siis aseta rajoituksia komponenttien väliseen viestintään oliosäikeen sisällä.

**Skenaariot**

4 Pelkästään olioiden luominen fysiikkakomponentille vaatii erityishuomiota. Jos olioita yritetään luoda kesken oliosäikeiden suorituksen, on kaksi vaihtoehtoa. Jos olio luodaan samalle alueelle, kuin säikeen käsittelemät oliot, toimii lisäys normaalisti. Jos taas halutaan lisätä olio eri alueelle, voi kyseisen alueen fysiikan mallinnus olla juuri kesken. Tällöin täytyisi odottaa mallinnuksen valmistumista, joka merkittävästi hidastaisi oliota luovan säikeen toimintaa. Sama ongelma tulee vastaan jos etsitään oliolle vapaata paikkaa.

Olioiden lisäämiseksi täytyykin joko käyttää varsin hidasta synkronointia säikeiden välillä, tai estää olioiden luonti oliosäikeiden sisältä. Tällöin olioiden luonti sallittaisiin ainoastaan joltain erilliseltä oliosäikeiden käynnistymistä edeltävältä koodialueelta.

**Skenaariot**

5 Malli soveltuu erinomaisesti käytettäväksi PC-prosessoreissa ja Xenon-prosessorissa, johtuen mallin hyvästä skaalautuvuudesta eri ydinmäärille. Esimerkiksi Xenon-prosessorille malli voidaan asettaa jakamaan kaikki pelin oliot kolmeen eri säikeeseen, joista jokaiseen tulee liki yhtä suuri määrä olioita.

Mallin soveltuminen Cell-prosessorille on kyseenalaista. Ensimmäinen huomioon otettava seikka on, että koska SPE-yksiköissä on jopa eri tarkkuus liukulukuoperaatioille kuin pääytimessä, ei olioiden jakamista voi harkita SPE-yksiköiden ja pääytimen kesken. Siksi kaikki olioihin liittyvät tehtävät olisi suoritettava seitsemällä SPE-yksiköllä. Tämä puolestaan lienee mahdotonta, johtuen SPE-yksiköiden pienestä muisti- ja suorituskapasiteetista. Jos jotkin olioihin liittyvät tehtävät olisikin mahdollista suorittaa SPE-yksiköllä, olisi olioiden määrällä hyvin selkeä yläraja yksiköiden muistin loppuessa. Siksi malli ei kokonaisuudessaan ole toimiva ratkaisu Cell-prosessorilla käytettäväksi. Jos pelimoottori toteutetaan kuitenkin tämän mallin mukaisesti, voidaan se Cell-prosessoria varten muuttaa toimimaan yhden ytimen tilassa, jolloin kaikki olioiden toiminnot ajetaan pääytimessä. Tällöin voitaisiin ottaa käyttöön komponenttien sisäinen säikeistys, joka ajaisi sopivia tehtäviä SPE-yksiköillä.

### 5.3.5 Laadun arvioinnin yhteenveto

Monisäikeistystä tukemattomien komponenttien asennus on mahdollista molemmissa tehtäväpohjaisissa malleissa. Asynkronisessa mallissa vaaditaan jokaiselle komponentille erityisen viestintärakenteen toteuttamista. Synkronisessa mallissa ei viestinnän suhteen ole erityisvaatimuksia. Olioperustaiseen malliin monisäikeistystä tukemattoman komponentin asennus on erittäin vaikeaa, jos se on tarkoitus asettaa mallin mukaiseen oliosäikeeseen. Näillä perusteilla voidaan sanoa, että tehtäväpohjaiset mallit soveltuvat parhaiten olemassa olevien komponenttien käyttöön. Synkroninen

malli taipuu helpoiten olemassa olevan pelimoottorin monisäikeistykseen, eihän tällöin tarvitsisi tehdä juuri muuta kuin etsiä toisistaan mahdollisimman riippumattomat toiminnot ja asettaa ne eri säikeisiin.

Olioperustaisen rinnakkaistuksen malli edellyttää suuria muutoksia olioita käsittelevien komponenttien toimintaan, ja soveltunee sellaisiin tapauksiin, joissa pelimoottori on mahdollista suunnitella puhtaalta pöydältä. Osa pelimoottorin komponenteista täytyy ainakin toistaiseksi toteuttaa itse, sillä tällä hetkellä valmiina saatavilla olevat komponentit eivät välttämättä sovellu mallin mukaiseen pelimoottoriin. Myöskään ei ole varmuutta siitä, minkälaista rinnakkaistusta valmiit komponentit tulevat tulevaisuudessa tukemaan.

Skaalautuminen käynnistyksen yhteydessä eri prosessoriydinmäärille onnistuu helposti ainoastaan olioperustaisen rinnakkaistuksen mallin mukaiselta pelimoottorilta. Tehtäväpohjaisen rinnakkaistuksen mallit edellyttävät, että tietyt pelimoottorin osat kirjoitetaan eri lailla eri säiemäärille. Prosessoriytimien määrä on nykyään kuitenkin vielä varsin pieni, joten ylitsepääsemätön ongelma tämä ei ole. Tehtäväpohjaisen rinnakkaistuksen mallit eivät myöskään suoraan skaalaudu suurille ydinmäärille, koska rinnakkaisten tehtävien määrä on rajallinen. Olioperustainen rinnakkaistus puolestaan ei ole rajattu tässä suhteessa, joten se soveltunee hyvin käytettäväksi myös tulevaisuudessa. On huomattava, että alaspäin skaalautuminen aina yhteen säikeeseen asti ei ole suorituskyvyn kannalta olennaista, sillä yksikin prosessoriydin pystyy hyvin ajamaan useita säikeitä ilman merkittäviä suorituskykyongelmia.

Pelimoottorin komponenttien välinen viestintä eroaa yksisäikeisten pelimoottoreiden viestinnästä eniten asynkronisen rinnakkaistuksen mallissa, jossa komponenttien välinen viestintä täytyy hoitaa käyttämällä jonkinlaista viestinvälitystä. Koska komponentit eivät voi olla suoraan yhteydessä toisiinsa, aiheuttaa tämä myös ongelmia, jos jokin komponentti tarvitsee suoraan tietoja toiselta komponentilta. Nämä ongelmat ovat kuitenkin ratkaistavissa, eivätkä ne muodosta varsinaista estettä mallin käytölle. Synkronisen tehtäväpohjaisen rinnakkaistuksen malli vastaa lähimmin yksisäikeisen sovelluksen toimintaa, ja edellyttää erikoisratkaisuja ainoastaan, jos rinnakkaisten

toimintojen halutaan viestivän keskenään. Tällöinkin erikoisratkaisut rajoittuvat tavallisten rinnakkaisohjelmoinnin synkronisointimenetelmien käyttöön. Olioperustaisen rinnakkaistuksen malli toimii komponenttien välisen viestinnän suhteen myös samoin kuin normaali yksisäikeinenkin sovellus. Synkronointia tarvitaan mallissa ainoastaan rinnakkaisten oliosäikeiden viestinnässä.

Mallien soveltumisesta Xbox 360:n Xenon-prosessorille ei pystytty löytämään merkittäviä huomioita. Tämä johtunee siitä, että Xenon-prosessori yhdessä PC-prosessorien kanssa on suunniteltu varsin yleiskäyttöisiksi. Sen sijaan mallien soveltuminen Playstation 3:n Cell-prosessorille on varsin erilainen. Tehtäväpohjaiset mallit soveltuvat käytettäväksi Cell-prosessorilla, jos niistä on mahdollista löytää prosessorin SPE-yksiköille sopivia tehtäviä. Asynkronisessa mallissa tällaisten tehtävien löytäminen lienee helpompaa kuin synkronisessa mallissa, koska synkroninen malli vaatii rinnakkaisten tehtävien olevan samalla tasolla edeltäjyysgraafissa. Olioperustaisen rinnakkaistuksen malli puolestaan ei sovellu lainkaan Cell-prosessorin rinnakkaistukseen. Olioihin liittyvät tehtävät ovat liian monimuotoisia ja muistia vieviä, jotta niitä voitaisiin kokonaisuudessaan suorittaa SPE-yksiköllä.

Skenaarion 5 osalta ei käsitelty sitä, että molemmat konsoliprosessorit eroavat PC-prosessoreista myös käskytason rinnakkaistuksen osalta. Ero näkyy siten, että konsolien prosessorit suorittavat hitaammin koodia, jossa on ehtolauseita. Tällaista koodia pelimoottoreissa sijaitsee erityisesti tekoälyyn ja skriptaamiseen liittyvissä komponenteissa. Käytännössä ongelmaan täytyy varautua käyttämällä komponenttien toteutuksessa sellaisia algoritmeja ja menetelmiä, jotka sisältävät mahdollisimman vähän ehdollisuutta. Se ei siis ole arkkitehtuuritason ongelma.

# 6 YHTEENVETO

Tämän tutkimuksen tavoitteena oli löytää ja vertailla pelimoottorin arkkitehtuurin malleja, jotka soveltuisivat moniytimisille prosessoreille. Tavoitteena oli tarjota riittävästi tietoa käytettävissä olevista vaihtoehdoista, jotta pelimoottoria kehitettäessä voidaan tehdä valinta eri ratkaisujen välillä. Tutkimus jakaantui kolmeen osaan. Ensimmäisessä tarkasteltiin kirjallisuudesta löytyvää tietoa rinnakkaisohjelmoinnista, moniytimisistä prosessoreista sekä pelimoottorien arkkitehtuureista. Toisessa esiteltiin moniytimisille prosessoreille soveltuvia pelimoottorin arkkitehtuurin malleja. Kolmannessa osassa vertailtiin näitä malleja arvioimalla kutakin arkkitehtuuria Kazman et al. (1994) esittämällä skenaariopohjaisella SAAM-menetelmällä.

Luvussa 4 moniytimisille prosessoreille soveltuvia pelimoottorin arkkitehtuurin malleja esiteltiin kolme, joista kaksi perustui pelimoottorin eri tehtävien rinnakkaistamiseen, ja yksi rinnakkaistamiseen pelin olioiden avulla. Viimeksi mainittu malli ideoitiin tätä tutkimusta varten. Näiden mallien lisäksi käsiteltiin myös pelimoottorin komponenttien sisäistä rinnakkaistusta, jonka todettiin soveltuvan erityisesti Playstation 3:n käyttöön.

Rinnakkaistuksen mallit esiteltiin vain pintapuolisesti, vertailun vaatimalla tarkkuudella. Koska mallien käsittelyssä ei edetty lähemmäs toteutuksen tasoa, on mahdollista, että huomaamatta jäi joitakin ongelmia tai jollain tavoin merkityksellisiä seikkoja. Tällaisia voisivat olla esimerkiksi ongelmat liittyen asynkronisen tehtäväpohjaisen mallin säikeiden väliseen viestintään, tai ongelmat olioperustaisen rinnakkaistuksen mallin fysiikkakomponentin toteutuksessa. Pelimoottorin rinnakkaistuksesta kiinnostuneiden onkin syytä etsiä käytännön kokemuksia mallien käytöstä.

Luvussa 5 vertailtiin kolmea rinnakkaistuksen mallia Kazman et al. (1994) esittämällä Software Architecture Analysis Methodilla, eli SAAMilla. Helpoimmin lähestyttävä rinnakkaistuksen malli on synkroninen tehtäväpohjainen rinnakkaistus, sillä sen käyttäminen ei juuri eroa tavallisten yksisäikeisten pelimoottoreiden toiminnasta, ja se voi suoraan käyttää nykyisten pelimoottoreiden komponentteja. Molempien tehtä-

väpohjaisten mallien suorituskyky riippuu siitä, kuinka paljon rinnakkaisia tehtäviä on mahdollista löytää. Tehtävien tehovaatimus määrää myös rinnakkaisuudesta saatavan edun – hyvin pienien tehtävien rinnakkaistamisesta ei voida saada suurta hyötyä. Koska malli skaalautuu hyvin nykyisille prosessoriydinmäärille, voidaan mallin käyttöä suositella erityisesti lähitulevaisuuden pelimoottoreihin, ja olemassa oleviin pelimoottoreihin, joihin halutaan lisätä tuki moniytimisyydelle. Skaalaus tosin edellyttää erillisen pelisilmukan kirjoittamista eri säiemäärille.

Asynkronisen tehtäväpohjaisen rinnakkaistuksen malli tarjoaa hieman parempaa suorituskykyä ja skaalautuvuutta kuin synkroninen malli, mutta vastapuolena on perinteisistä pelimoottoreista poikkeava viestintä pelimoottorin komponenttien välillä. Ongelma ei ole kuitenkaan ylitsepääsemätön, joten myös tämän mallin käyttöä voidaan suositella lähitulevaisuuden pelimoottoreihin – ei kuitenkaan vanhojen pelimoottoreiden modernisointiin. Asynkronisen mallin ongelmana on se, että se ei ole sidottu pelisilmukkaan. Tästä johtuen mallin mukaisen pelimoottorin ruudunpäivityskierroksen pituudessa voi esiintyä vaihtelua. Vaihtelun merkitys kuitenkin pienenee ruudunpäivitysnopeuden kasvaessa.

Olioperustainen rinnakkaistus skaalautuu hyvin nykyisille ja tulevaisuuden prosessoriydinmäärille. Siksi sen käyttöä voitaisiin ajatella erityisesti tulevaisuuden pelimoottoreissa. Suorituskykynsä puolesta malli on erinomainen, sillä se pystyy hyödyntämään kaikkia käytössä olevia suoritinytimiä. Tosin nykyisillä suoritinydinmäärillä eroa tehtäväperustaisiin malleihin ei luultavasti synny, vaan ero näkyy vasta tulevaisuuden kymmenien ydinten suorittimilla. Nykyisissä pelimoottoreissa sen käyttöä rajoittaa erityisesti se, että valmiina saatavat komponentit eivät tue tätä rinnakkaistuksen mallia.

Mallien soveltuvuudesta PC- tai Xbox 360 laitteille ei löydetty eroja. Molemmat tehtäväpohjaiset mallit soveltuvat käyttöön Playstation 3:n Cell-prosessorilla, mutta edellyttävät erityisratkaisuja. Olioperustaisen rinnakkaistuksen malli puolestaan ei lainkaan käy Cell-prosessorille. Playstationille pelimoottoria kehitettäessä olisikin viisainta turvautua komponenttien sisäiseen säikeistykseen. Jos pelimoottorin on tar-

koitus toimia sekä Playstation 3:lla että PC:llä ja Xbox 360:lla, on mietittävä, kannattaako koko pelimoottorin laajuisten rinnakkaistusratkaisujen toteuttaminen lainkaan. Tässä tutkimuksessa käsitellyt mallit soveltuvat huonosti monialustakehitykseen. Yhteisen rinnakkaistuksen mallin löytäminen asymmetrisen Cell-prosessorin ja symmetristen PC ja Xenon -prosessoreiden välille olisikin merkittävä edistysaskel pelimoottoreiden kehityksessä.

Tutkitut mallit ja niiden vertailu soveltuu yleistettäviksi kaikkiin pelimoottoreihin, eikä rajoituksia erityisen pelityypin tai komponenttikonfiguraatioiden suhteen ole. Sen lisäksi, että tutkimus liittyy varsinaisten pelimoottoreiden kehityksen jatkumoon, liittyy tutkimus myös yleisempään suuntaukseen muuntaa yksisäikeisiä ohjelmia tukemaan rinnakkaistusta. Tutkimuksessa saatiin hyviä kokemuksia ohjelmistoarkkitehtuurien käytöstä tällaisen muunnoksen suunnittelussa.

Vertailuun sopivan menetelmän valintaan käytettiin hyvin paljon aikaa, ja sitä vaihdettiinkin kesken tutkimuksen. Ensimmäiseksi käytetty menetelmä oli Adven ja Vernonin (2004) esittämä task graph analysis täydennettynä Amdahlin (1967) laskukaavalla, mutta niiden havaittiin paremmin sopivan varsin matalalla tasolla kuvattujen rinnakkaisohjelmien formaaliin analysointiin. Ohjelmistoarkkitehtuurin menetelmien todettiin sopivan paremmin korkeammalla tasolla kuvattujen mallien analysointiin. SAAMin käytön eräänä ongelma oli, että vertailtavat mallit eivät suoraan vertaudu ohjelmistoarkkitehtuureihin, vaan ohjelmistoarkkitehtuuri-malleihin. Ero abstraktiotasossa ei kuitenkaan merkittävästi vaikeuttanut menetelmän käyttöä.

Tämä tutkimus on ensimmäinen yritys koota yhteen pelimoottoreiden rinnakkaistukseen liittyviä näkemyksiä. Rinnakkaistuksen malleja käsitellään tarkemmin tämän tutkimuksen pohjalta CMP Median Gamasutraan[1] kirjoitettavassa artikkelissa. Aiheesta riittää vielä runsaasti selvitettävää. Jatkotutkimukseen soveltuvia aiheita voisivat olla esimerkiksi seuraavat:

1 http://www.gamasutra.com/

- Mallien suorituskyvyn vertailu prototyypin avulla. Prototyypin avulla voidaan erityisesti selvittää, rajoittaako oliosäikeiden lisääntyessä lisääntyvä säikeiden välinen viestintä olioperustaisen rinnakkaistuksen mallin suorituskykyä.
- Olioperustaisen mallin vaatiman fysiikkakomponentin rakentaminen. Fysiikkakomponentin toiminnan jakaminen säikeisiin olioperustaisesti vaatinee muutoksia nykyisin käytössä oleviin algoritmeihin. Toimivan komponentin rakentaminen on edellytys todellisen pelimoottorin tekemiselle.
- Monialustakehitykseen soveltuvat rinnakkaistusmallin löytäminen. Tässä tutkimuksessa esitellyt mallit eivät sovellu hyvin edes Playstation 3 -konsolille pelimoottoria kehitettäessä. Konsoli- ja työpöytäjärjestelmien lisäksi täytyy huomioida yhä kasvavat mobiililaitemarkkinat, jonne moniytimisyys tullee ennen pitkää.

Toistaiseksi pelimoottoreiden ja peleissä käytettävien tekniikoiden tutkimus on rajoittunut lähinnä pelejä tuottavien yritysten sisäiseksi. Vaikka Suomessakin on muutaman viime vuoden aikana perustettu pelien tutkimuksen yksiköitä moniin yliopistoihin, ovat niiden tutkimuskohteet rajoittuneet lähinnä pelien humanistisiin ilmiöihin, kuten oppimiseen peleissä ja pelisuunnitteluun. Todellinen tieteellinen vuoropuhelu pelien tekniikasta ei ole vielä käynnistynyt, ei Suomessa eikä ulkomailla.

# 7 LÄHDELUETTELO

Adve V., Vernon M. 2004. Parallel Program Performance Prediction Using Deterministic Task Graph Analysis. ACM Transactions on Computer Systems 22(1), 94–136.

AMD. 2005. AMD Multi-Core Technology White Paper. Saatavilla www-osoitteessa <http://multicore.amd.com/>

Amdahl G. 1967. Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities. AFIPS Conference Proceedings, 483-485.

Andreoli R., De Chiara R., Erra U., Scarano V. 2005. Interactive 3D environments by using videogame engines. Proceedings of the Ninth International Conference on Information Visualisation, 515 – 520.

Astley D., Hawkins K. 2002. OpenGL Game Programming. Course Technology.

Balakrishnan S., Rajwar R., Upton M., Lai K. 2005. The Impact of Performance Asymmetry in Emerging Multicore Architectures. Proceedings of the 32nd Annual International Symposium on Computer Architecture, 506-517.

Barney B. 2005. Introduction to Parallel Computing [online]. Livermore Computing [viitattu 13.1.2006]. Saatavilla www-osoitteessa <http://www.llnl.gov/computing/tutorials/parallel_comp/>

Blow J. 2004. Game Development: Harder Than You Think. Queue Focus 1(10), 28 – 37.

Buschmann F., Meunier R., Rohnert H., Sommerlad P., Stal M., Sommerlad P., Stal M. 1996. Pattern-Oriented Software Architecture, Volume 1: A System of Patterns. John Wiley & Sons.

Cheah T., Ng K.-W. 2005. 2005. A practical implementation of a 3D game engine. Proceedings of the Computer Graphics, Imaging and Vision: New Trends, 351 – 358.

Claypool M., Nichols J. 2004. Games: The effects of latency on online madden NFL football. Proceedings of the 14th international workshop on Network and operating systems support for digital audio and video, 146 – 151.

Conger D. 2004. Physics Modeling for Game Programmers. Thomson Course Technology.

De Gelas J. 2005. The Quest for More Processing Power, Part Two: "Multi-core and multi-threaded gaming". [online] [viitattu 8.3.2006]. <http://www.anandtech.com/cpuchipsets/showdoc.aspx?i=2377>

Dobrica L., Niemelä E. 2002. A Survey on Software Architecture Analysis Methods. IEEE Computer Society, 638 – 653.

Douglass B. 1998. Real-Time UML. Addison Wesley.

Eberly D. 2004. 3D Game Engine Architecture. Morgan Kaufmann.

Eggers S.J., Emer J.S., Leby H.M., Lo J.L., Stamm R.L., Tullsen D.M. 1997. Simultaneous multithreading: a platform for next-generation processors. IEEE Micro 17(5), 12-19.

El Rhalibi A., England D. and Costa S. 2005. Game Engineering for a Multiprocessor Architecture. International DiGRA Conference 2005.

Epic Games. Unreal Engine 2 documentation. [online] [viitattu 31.1.2006]. <http://udn.epicgames.com/Two/WebHome>

Epic Games. Unreal Engine 3 Overview. [online] [viitattu 30.1.2006]. Saatavilla www-osoitteessa<http://www.unrealtechnology.com/>

Fahringer T., Pllana S. 2002. UML Based Modeling of Performance Oriented Parallel and Distributed Applications. Proceedings of the 2002 Winter Simulation Conference.

Foster I. 1995. Designing and Building Parallel Programs : Concepts and Tools for Parallel Software Engineering. Addison Wesley.

Gabb H., Lake A. 2005. Threading 3D Game Engine Basics. Gamasutra. [online] [viitattu 7.2.2006]. <http://www.gamasutra.com>

Gustafson J.L. 1988. Reevaluating Amdahl's Law. Communications of the ACM 31(5), 532-533.

Hansen P. 1973. Concurrent Programming Concepts. ACM Computing Surveys 5(4), 223-245.

Hazra T. K. 1995. Parallel Computing. Defining what it is and the doors it opens via transputers. IEEE Potentials 14(3), 17 – 20.

Herlihy M., Moss E. 1993. Transactional memory: architectural support for lock-free data structures. Proceedings of the 20th annual international symposium on Computer architecture, 289-300.

IEEE Standardi 1471-2000. IEEE Recommended practice for architectural description of software-intensive systems.

Igehy H., Stoll G., Hanrahan P. 1998. The Design of a Parallel Graphics Interface. Proceedings of the 25th annual conference on Computer graphics and interactive techniques, 141 – 150.

Intel. 2005. Overview: Auto-parallelization [online] [viitattu 25.1.2006]. Saatavilla www-osoitteessa http://www.intel.com/software/products/compilers/clin/docs/ug_cpp/lin1108.htm

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

Jerez J. 2005. Newton Game Dynamics 1.5 SDK documentation.

Ji M., Felten E., Li K. 1998. Performance measurements for multithreaded programs. Proceedings of the 1998 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems, 161 – 170.

Jiang Y., Saxe R., Kanwisher N. 2004. Department of Brain and Cognitive Sciences, Massachusetts Institute of Technology. Saatavilla www-osoitteessa <http://www.psychologicalscience.org/pdf/ps/jiang.pdf>

Kazman R., Bass L., Webb M., Abowd G. 1994. SAAM: a method for analyzing the properties of software architectures. 16th International Conference on Software Engineering, 81 – 90.

Kazman R., Bass L., Clements P., Abowd G. 1996. Scenario-Based Analysis of Software Architecture. IEEE Software 13(6), 47 – 55.

Kazman R., Klein M., Barbacci M., Longstaff T., Lipson H., Carriere J. 1998. The architecture tradeoff analysis method. Fourth IEEE International Conference on Engineering of Complex Computer Systems, 68 – 78.

Krishnaprasad S. 2001. Uses and abuses of Amdahl's law. Journal of Computing Sciences in Colleges 17(2), 288-293.

Larus J., Sutter H. 2005. Software and the concurrency revolution. ACM Queue 3(7), 54-62.

McDougall R. 2005. Extreme software scaling. ACM Queue 3(7), 36-46.

Microsoft. 2005. DirectX 9.0c SDK dokumentaatio.

Olukotun K., Hammond L. 2005. The future of microprocessors. ACM Queue 3(7), 26-34.

OMG. 2005. Unified Modeling Language Specification [online] [viitattu 18.1.2006]. Saatavilla www-osoitteessa <http://www.uml.org/>

OpenAL Programmers Guide. [online] [viitattu 1.2.2006]. <http://www.openal.org/>

OpenMP Architecture Review Board. 2005. Official OpenMP Specifications version 2.5. Saatavilla www-osoitteessa <http://www.openmp.org/>

Pingali K., Beck M., Johnson R., Moudgill M., Stodghill P. 1991. Dependence flow graphs: an algebraic approach to program dependencies. Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, 67 – 78.

Rakkarsoft. 2006. Raknet Manual. [online] [viitattu 1.2.2006]. <http://www.rakkarsoft.com/raknet/manual/>

Seebach P. 2005. Unleashing the power of the Cell Broadband Engine. IBM developerWorks. [online] [viitattu 20.3.2006]. <http://www-128.ibm.com/developerworks/power/library/pa-fpfunleashing/>

Stokes J. 2005. Inside the Xbox 360, Part II: the Xenon CPU. [online] [viitattu 20.3.2006]. <http://arstechnica.com/articles/paedia/cpu/xbox360-2.ars/1>

Sutter H. 2005. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. Dr. Dobb's Journal, 30(3).

Tanenbaum A. 2001. Modern Operating Systems. Prentice Hall.

The C++ Standards Committee. C++ standard [online] [viitattu 13.1.2006]. Saatavilla www-osoitteessa <http://www.open-std.org/jtc1/sc22/wg21/>

Tietotekniikan liitto. ATK-sanakirja [online]. [viitattu 9.1.2006]. Saatavilla www-osoitteessa <http://www.tt-tori.fi/atk-sanakirja/>

Varanese A. 2002. Game Scripting Mastery. Thomson Course Technology.

Wilson K. 2006. Managing Concurrency: Latent Futures, Parallel Lives. [online] [viitattu 22.3.2006]. <http://www.gamearchitect.net/Articles/ManagingConcurrency1.html>

Zerbst S. 2004. 3D Game Engine Programming. Course Technology Crisp.