Rust peliohjelmoinnissa

Victor Bankowski, Antti Karjalainen ja Janne Pulkkinen

Seminaariraportti
HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

Helsinki, 9. joulukuuta 2017

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Matemaattis-luonnontieteellinen	Tietojenkäsittelytieteen laitos

Tekijä — Författare — Author
Victor Bankowski, Antti Karjalainen ja Janne Pulkkinen

Työn nimi — Arbetets titel — Title
Rust peliohjelmoinnissa

Oppiaine — Läroämne — Subject
Tietojenkäsittelytiede

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Seminaariraportti	9. joulukuuta 2017	15

Tiivistelmä — Referat — Abstract

Rust on moderni Mozillan kehittämä järjestelmäohjelmointikieli. Sen tavoitteena on olla sekä turvallinen että nopea. Tämä on mahdollista käännösaikasten omistajuuden ja lainaamisen käsitteiden avulla.

Rustissa jokaisella on arvolla on yksikäsitteinen omistaja joka on vastuussa sen poistamisesta. Tämä omistajuuden käsite mahdollistaa automaattisen muistinhallinnan ilman roskienkeruuta.

Lainaaminen taas mahdollistaa arvon käytön ilman omistajuuden siirtämistä. Yhdestä arvosta voi olla joko yksi muokattava lainaus tai usea muokkaamaton lainaus. Tämä estää samanaikaisuus- ja muistiongelmia käännösaikaisesti.

Verrattuna perinteisiin järjestelmäohjelmointikieliin, kuten C ja C++:aan, Rustin kirjastovalikoima on vielä lapsenkengissä kielen suhteellisen nuoruuden takia. Rust kuitenki tukee C-kirjastojen käyttöä suoraan FFI-rajapinnan kautta.

Vaikka Rustille ei ole vielä Unityn tai Unreal Enginen kaltaisia kokonaisvaltaisia pelinkehitystyökaluja, sille on kuitenkin saatavilla useita eri pelinkehitykseen liittyviä kirjastoja. Pelimoottorikirjastoja ovat esimerkiksi Amethyst ja Piston. Grafiikkakirjastoista löytyy sidonnat tunnetuimmille grafiikkarajapinnoille. Lisäksi saatavilla on korkeamman tason grafiikkakirjastoja kuten Gfx-rs ja Glium.

Rustin ominaispiirteet vaikuttavat ohjelmistojen, esimerkiksi pelimoottorien, rakenteellisiin valintoihin. ECS (Entity Component System) on yksi Rustille sopiva pelimoottorirakenne ja siitä on useita toteutuksia on saatavilla. Näistä toteutuksista Specs on suosituin erityisesti sen samanaikaisuus ominaisuuksien takia.

Avainsanat — Nyckelord — Keywords
ohjelmointikieli, peliohjelmointi

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — Övriga uppgifter — Additional information

Sisältö

1 Johdanto

Rust on käännettävä ohjelmointikieli, jonka kehitystä tukee Mozilla-säätiö (Anderson et al., 2016). Mozilla käyttää kieltä uuden rinnakkaisuutta hyödyntävän Servo -selainmoottorin ohjelmointiin ja lisäksi sitä käyttävät useat organisaatiot kuten Dropbox, Canonical ja Npm (*Friends of Rust - The Rust Programming Language*, s.a.).

Järjestelmäohjelmointikielenä Rust mahdollistaa korkeaa suorituskykyä ja hallittua muistinkäyttöä vaativien sovellusten kehittämisen. Tästä on hyötyä esimerkiksi käyttöjärjestelmissä ja sulautetuissa järjestelmissä. C ja C++ -kielistä poiketen Rust kuitenkin estää yleisiä muistinhallintaa ja kilpatilanteita koskevia ongelmia. Rust ratkaisee nämä ongelmat käyttämällä muistinhallinnassa omistajuuden ("ownership") ja lainaamisen ("borrowing") käsitteitä, jotka estävät mahdolliset virhetilanteet jo ohjelman käännösvaiheessa. Näin Rust ei vaadi virtuaalikoneen tai tulkin käyttöä ohjelman suorituksen aikana.

2 Historia

Rust-kielen kehitys alkoi vuonna 2006 Graydon Hoaren sivuprojektina, jollaisena se jatkui yli kolmen vuoden ajan (*Frequently Asked Questions - The Rust Programming Language*, s.a.). Mozilla-säätiö osallistui kehitykseen ensimmäisen kerran vuonna 2009 ja on tukenut ohjelmointikielen kehitystä siitä lähtien. Nykyisin kieltä kehittävä ryhmä – *The Rust Team* – jakautuu osaryhmiin, jotka vastaavat kielen eri osa-alueista. Osa-alueisiin kuuluvat esimerkiksi kääntäjän kehittäminen, kielen ominaisuuksien suunnittelu ja dokumentaatio.

Suurimpiin Rustia käyttäviin projekteihin kuuluu Mozillan kehittämä Servo -selainmoottori. Sen tavoitteisiin kuuluu sivun piirtämisen, HTML-datan parsimisen ja muiden verkkoselaimen piiriin kuuluvien tehtävien rinnakkaistaminen (*Inside a super fast CSS engine: Quantum CSS (aka Stylo)*, s.a.). Servo-projektiin kuuluva CSS-moottori Stylo on otettu käyttöön Mozilla Firefox -verkkoselaimen uusissa versioissa (*Introducing the New Firefox: Firefox Quantum*, s.a.).

3 Perusteet

Ohjelmointikieleen tutustuessa on tapana kirjoittaa klassinen "Hei maailma" -ohjelma joka tulostaa kyseisen lauseen. Koodiliitteessä 1 on kyseisen ohjelman Rust-toteutus, joka ei poikkea juurikaan muista proseduraalisilla kielillä tehdyistä toteutuksista. Suurin ero on tulostuskomennossa, joka on toteutettu makrolla tekstin muotoilun helpottamiseksi. Koodiliitteessä 2 on esimerkki tekstin muotoilusta.

Rustissa muuttujat määritellään käyttäen **let** avainsanaa ja ne eivät ole oletusarvoisesti muokattavissa. Muokattavat muuttujat määritellään käyttäen avainsanayhdistelmää **let mut**. Muuttujan tyyppi määritellään sen nimen jälkeen kaksoispisteellä erotettuna. Esimerkiksi koodiliitteessä 2 muuttuja nimeltä *s_luku* on 32-bittinen etumerkillinen kokonaisluku.

Useimmissa tapauksissa muuttujan tyypin voi jättää merkitsemättä, koska Rust osaa päätellä sen käännösaikana. Kuitenkin funktioiden parametrien ja palautusarvon tyypit täytyy aina merkata. Funktion palautusarvon tyyppi määritellään `->` nuolen jälkeen. Koodiliitteessä 3 olevan funktion parametri ja palautusarvo ovat molemmat 64-bittisiä etumerkittömiä kokonaislukuja.

Rustin **for**-silmukat ovat *for–each*-tyyppisiä eli niissä käydään kaikki sille annetun iteraattorin alkiot läpi. Esimerkiksi koodiliitteessä 3 käydään kaikki välin $[1, n + 1)$ kokonaislukuarvot läpi.

Rustissa ei tarvitse käyttää **return** avainsanaa, jos arvo palautetaan funktion lopussa ja tällöin kyseisen rivin puolipiste jätetään pois.

3.1 Omistajuus

Muistinhallinta on tärkeä osa ohjelmien toimintaa. Tätä varten monissa kielissä käytetään automaattista roskienkeruuta, joka vähentää ohjelmoijan vastuuta, mutta samalla myös vähentää ohjelmoijan mahdollisuuksia vaikuttaa muistinhallintaan.

Pelimoottoreissa muistinhallinnan optimointi on tärkeää hyvän suorituskyvyn saavuttamiseksi. Tämän takia pelimoottorit toteutetaan usein kielillä, joissa muistinhallinta on manuaalista. Tämän kaltaisissa kielissä jää ohjelmoijan vastuuksi varata ja vapauttaa muisti oikeaoppisesti. Tästä johtuen kyseiset kielet ovat alttiita muistihallintavirheille. (*Common Vulnerabilities and Exposures*, s.a.)

Rust-ohjelmointikielessä ei käytetä automaattista roskienkeruuta, mutta siinä ei myöskään jätetä muistinhallintaa täysin ohjelmoijan vastuulle. Tämä on mahdollista, koska Rust takaa sen että jokaisella varatulla muistialueella on yksikäsitteinen omistaja, joka on vastuussa sen vapauttamisesta. Koska Rustissa omistajuus on käännösaikainen käsite, on se yhtä tehokasta kuin manuaalinen muistinhallinta. (Klabnik & Nichols, 2018)

Ohjelmointikielen tasolla muuttuja omistaa arvonsa. Kun muuttuja poistuu näkyvyysalueelta, niin sen omistama arvo vapautetaan. Muuttuja voi siirtää omistamansa arvon jollekin muuttujalle tai funktiolle parametriksi, jolloin myös sen omistajuus siirtyy. Tämän jälkeen alkuperäinen muuttuja ei voi käyttää arvoa sillä se ei enää omista sitä.

Liitekoodissa 5 havainnollistetaan, mitä tapahtuu kun muuttujaa yritetään käyttää sen arvon siirron jälkeen. Virheviestistä nähdään selvästi missä arvon siirto ja missä virheellinen muuttujan uudelleenkäyttö tapahtuvat. Lisäksi viestissä huomautetaan *Copy*-trait toteutuksen puuttumisesta *String*-tyypille. Rustissa siirrot ovat aina muistisiirtoja pinossa. *Copy*-traitin

toteuttavat tyypit takaavat, että kaikki niihin liittyvä data sijaitsee siellä. Tämän takia niiden siirrossa tehdään ns. syväkopiointi. *Syväkopiointi*(Deep copy) tarkoittaa kaiken muuttujaan liittyvän datan kopioimista. Syväkopioinnin vastakohta on *pinnallinen kopiointi*(Shallow copy), jossa vain viittaukset kopioidaan. Koska siirto on syväkopio *Copy*-traitin toteuttaville tyypeille, ei tämän tyyppisillä muuttujilla ole siirtosemantiikkaa.

3.2 Lainaaminen ja elinajat

Jos Rustissa olisi pelkästään omistajuuden käsite, niin funktioiden pitäisi palauttaa parametrinsa, jotta niitä voitaisiin käyttää uudelleen funktiokutsun jälkeen. Parametrien uudelleenkäyttö on kuitenkin todella yleistä, jonka takia Rustissa on tätä varten käytössä lainaamisen käsite. Sen sijaan että omistajuus siirrettäisiin, arvoa voidaan lainata. Lainauksen aikana alkuperäinen omistaja ei voi käyttää arvoa, mutta se saa sen takaisin käyttöönsä heti, kun lainaus on loppunut. Rustissa on kaksi eri lainaustyyppiä: Muokkaamaton ja muokattava lainaus.

Muokkaamattomassa lainauksessa lainaaja ei pysty muokkaamaan lainattua arvoa, mutta lainauksia voi olla useita kappaleita samanaikaisesti (aliasiointi). *Muokattavia lainauksia* saa sen sijaan olla vain yksi arvoa kohden, mutta niitä on mahdollista muokata. Samasta arvosta ei voi myöskään olla muokattavaa ja muokkaamatonta lainausta samanaikaisesti. Nämä säännöt takaavat sen että aliasointia ja muokkaamista ei voi tapahtua samanaikaisesti.

Tällainen aliasoinnin ja muokattavuuden kahtiajako välttää muistiturvallisuusongelmia kuten iteraattorin mitätöimistä (iterator invalidation). Koodiliitteessä 6 on esimerkki iteraattorin mitätöitymistilanteesta. Virheviestistä näkee kuinka lainauksen säännöt estävät ongelman käännösaikana. Useimmissa kielissä, kuten Javassa ja C++:ssa, vastaavanlainen koodi kääntyisi ja siitä aiheutuvat ongelmat huomattaisiin vasta ajonaikana. Java heittäisi tässä tapauksessa poikkeuksen, kun taas C++:ssa siitä voisi pahimmassa tapauksessa seurata se että iteraattori osoittaisi vapautettuun muistiin.

Lainaaminen auttaa myös estämään kilpatilanteista johtuvia samanaikaisuusongelmia. Kilpatilanne voi tapahtua vain, jos useampi taho pääsee samanaikaisesti käsiksi arvoon ja ainakin yksi niistä muokkaa sitä. Rustin lainaamissäännöt nimenomaan eivät salli tätä.

Kuten jo aiemmin mainittiin, arvot vapautetaan aina silloin kun niiden omistajat poistuvat näkyvyysalueiltansa. Vapauttamisen jälkeen kaikki arvoon liittyvät lainaukset osoittaisivat vapautettuun muistiin eli ne olisivat niin sanottuja roikkuvia viittauksia (Dangling references). Koodiliitteessä 7 on esimerkki juuri tällaisesta virheellisestä lainauksesta. Jotta tätä ei tapahtuisi, lainauksien oikeellisuutta pitää seurata jollain tavalla.

Rustissa kaikilla lainauksilla on tätä varten *elinaika*, jonka täytyy aina olla lyhyempi kuin sen lainaaman arvon elinaika. Elinaika pystytään useimmissa tapauksissa päättelemään koodista automaattisesti. Joskus tämä ei

ole kuitenkaan mahdollista, jolloin ohjelmoijan täytyy merkata se koodissa erillisellä elinaikamerkillä. Koodiliite-esimerkissä 8 on kaksi funktiota. Ensimmäisessä ei ole elinaikamerkintää, jonka takia Rust ei tiedä kummasta lainauksesta palautusarvona oleva lainaus tulee. Elinaikamerkatussa versiossa taas elinaikamerkintä yhdistää parametrin b elinajan palautusarvon elinaikaan, jolloin Rust tietää kuinka pitkään palautusarvoa voi käyttää.

4 Vertailu C-kieliin

4.1 Kehitystyökalut

C ja C++ -ohjelmointikielet ovat Rustin tavoin käännettäviä ohjelmointikieliä, jotka soveltuvat suorituskykyä vaativien sovelluksien kehittämiseen. Tällaisiin sovelluksiin kuuluvat muun muassa käyttöjärjestelmät, sulautetut järjestelmät ja pelimoottorit, joista esimerkiksi *Unity* ja *Unreal Engine* on kirjoitettu C++ -kieltä käyttäen. Rustille on saatavilla pelimoottorikirjastoja kuten Piston ja Amethyst, mutta Unityn tai Unreal Enginen kaltaisia kokonaisvaltaisia pelinkehityskokonaisuuksia ei toistaiseksi ole saatavilla Rust-kielelle.(*Are we game yet? - Rust*, s.a.-a)

Rust-lähdekoodin kääntämiseen käytetään rustc -työkalua, joka on kirjoitettu Rust-kielellä. Kääntäjä hyödyntää LLVM-kääntäjäinfrastruktuurin työkaluja, ja hyötyy siten kyseistä projektia koskevista suorituskykyparannuksista. (*Frequently Asked Questions - The Rust Programming Language*, s.a.) C++ -kielestä poiketen Rustille ei ole kuitenkaan saatavilla vaihtoehtoisia kääntäjiä. Laajassa käytössä oleviin C++ -kielen kääntäjiin kuuluu muun muassa LLVM-projektin *Clang*, *GNU GCC* ja *Intel C++ Compiler*.

Koska Rust-kieleen ei kuulu ajonaikaista virtuaalikonetta tai tulkkia, Rust-koodia on C-kielten tavoin mahdollista hyödyntää korkeamman tason kielissä kuten Pythonissa tai Rubyssa. Tässä tapauksessa sovelluksen tehokasta suorituskykyä ja muistinkäyttöä vaativat osat voidaan kirjoittaa Rustilla.(*Rust Inside Other Languages - Rust*, s.a.) C-kielellä kirjoitettuja kirjastoja on myös mahdollista käyttää Rustilla toteuttamalla kirjastolle sidonnat, jotka käyttävät kirjaston funktioita FFI (*Foreign Function Interface*) -rajapinnan kautta. C-kielellä kirjoitettuja kirjastoja ei siis tarvitse kirjoittaa uudelleen, jotta niitä voidaan hyödyntää Rust-kielessä. Kuitenkaan C:llä kirjoitetut rajapinnat eivät suoraan käytettyinä noudata rustin turvakäytänteitä. Tämän takia suorien sidontakirjastojen lisäksi C-kirjastoille tehdään usein Rustille idiomaattisempia rajapintakirjastoja, jotka pyrkivät estämään alkuperäisen rajapinnan väärinkäytön ja siitä johtuvat muistivirheet.

4.2 Käyttöjärjestelmät

C++ -kieli ja sitä edeltävä C-kieli ovat yleisiä käyttöjärjestelmien ohjelmointikielinä: *Microsoft Windows* on ohjelmoitu C++ -kielellä, *FreeBSD*

on ohjelmoitu käyttäen sekä C että C++-kieliä ja *Linux* käyttää C-kieltä. Rust olisi muistinhallintansa puolesta otollinen käyttöjärjestelmäytimien ohjelmointiin, jossa muistinhallintavirheillä voi olla vakavia seurauksia järjestelmän vakauden ja tietoturvallisuuden kannalta. *Redox* on Rust-kielellä kirjoitettu mikroydinrakennetta noudattava käyttöjärjestelmä.(*The Redox Operating-System*, s.a.) Redox on vielä aikaisessa kehitysvaiheessa eikä siten sovellu jokapäiväiseen käyttöön.

Tock on Rust-kielellä toteutettu sulautettu käyttöjärjestelmä.(Levy et al., 2015) Tavallisista käyttöjärjestelmistä poiketen sulautetuissa käyttöjärjestelmissä on tiukat resurssivaatimukset. Tock on suunniteltu muun muassa toimimaan ympäristössä, jossa käytössä on vain 64 kilotavua keskusmuistia. Rustin käyttö on vaikuttanut Tockin käyttöjärjestelmäytimessä tehtyihin valintoihin sen muistinhallinnan takia.(Levy et al., 2017)

Rust tukee virallisesti 32- ja 64-bittisiä Microsoft Windows, *Linux* ja *OS X* -käyttöjärjestelmiä.(*Rust Platform Support - The Rust Programming Language*, s.a.) Muille alustoille, kuten ARM-arkkitehtuurille ja *iOS* ja *Android* -mobiilikäyttöjärjestelmille on rajatumpi tuki: alustoille on saatavilla valmiiksi käännetyt stadardikirjastot ja kehitystyökalut, mutta automatisoituja testejä ei ajeta niille julkaisun yhteydessä eikä niitä voi siten pitää yhtä toimintavarmoina. iOS ja Android -käyttöjärjestelmille on olemassa virallisesti tuetut C ja C++ -kielille tarkoitetut kehitystyökalut. Esimerkiksi *Android NDK* sallii sovellusten kehittämisen C ja C++ -kieliä käyttäen, tarjoten myös rajapinnat esimerkiksi 3D-piirtämistä, äänentoistoa ja säikeiden hallintaa varten.(*Getting Started with the NDK | Android Developers*, s.a.)

5 Peliohjelmointi Rustilla

5.1 Pelimoottorit

Kuten aiemmin jo mainittiin Rustissa ei ole vielä Unityn tai Unrealin kaltaisia pelimoottoreita. Kuitenkin pelinkehitystä varten Rustille on kehitteillä pelimoottorikirjastoja. Tälläisiä ovat esimerkiksi Piston ja Amethyst.

Näistä Piston on täysin modulaarinen pelimoottori eli se on käytännössä kokoelma erilaisia pelinkehityskirjastoja. Se koostuu muutamasta ydinkirjastosta ja isosta määrästä apukirjastoja. Ydinkirjastoihin kuuluvat *pistoncore-input* syötteenhallintakirjasto, *pistoncore-window* ikkunointikirjasto ja *pistoncore-event_loop* pelisilmukanhallintakirjasto. Apukirjastojen joukosta löytyy esimerkiksi *vecmath*, joka on yksinkertainen vektorimatematiikkakirjasto, *piston3d-cam* 3D-kameran hallintakirjasto ja *texture_packer* tekstuurinpakkauskirjasto. Pistonin modulaarisuudesta kertoo se että sen ikkunointi- sekä 2d-grafiikkakirjastoilla on useampi backend.

Amethyst(*Amethyst | Data-oriented and data-driven game engine*, s.a.) taas on dataorientoitunut ja dataohjattu pelimoottori, joka on saanut inspiraationsa Bitsquid Enginestä (nykyisin Autodesk Stingray)(*Amethyst*, s.a.).

Dataorientoitunut ohjelmointi on ohjelmointiparadigma, joka keskittyy dataan. Siinä kiinnitetään erityisesti huomiota datan tyyppiin, sen esitysmuotoon muistissa ja kuinka se prosessoidaan. *Dataohjattu suunnittelumalli* tarkoittaa taas sitä että ohjelman logiikkaa ohjautuu mahdollisimman paljon ulkoisen datan perusteella. Tällöin on mahdollista muokata ohjelman toimintaa ilman sen uudelleenkääntämistä. (*Glossary*, s.a.)

Amethystin ominaisuuksiin kuuluvat yksinkertainen pelitilan hallinta pinoautomaatilla, skriptausrajapinta, gfx-rs -kirjastoon pohjautuva renderöinti ja specs-kirjastoon pohjautuva ECS-malli.

Piston ja Amethyst noudattavat kummatkin hyvin Unixmaista lähestymistapaa: molemmat koostuvat pienistä integroiduista palasista. Amethyst muodostaa näistä kahdesta selkeämmän kokonaisuuden kun taas Piston on modulaarisempi kokonaisuus erilaisia kirjastoja.

5.2 ECS

Rustin omistajuuden ja lainaamisen mekanismit estävät aliasioinnin ja muokattavuuden samanaikaisuuden. Tämä vaikuttaa pelimoottorin suunnittelussa tehtäviin rakenteellisiin valintoihin. Esimerkiksi syklisten viittausten käyttö vaikeutuu huomattavasti. Tällaisia viittauksia voi esiintyä esimerkiksi maailman ja sen sisältävien olioiden välillä. Rustissa sykliset viittaukset voidaan toteuttaa älyosoitin-yhdistelmillä, mutta niiden käyttö vähentää suoritustehoa lisäämällä ajonaikaisia tarkistuksia ja lisäksi siirtää osan vastuusta ylläpitää lainaamisen sääntöjä ohjelmoijalle.

Vaihtoehtoinen tapa jäsennellä peli on käyttää ECS (Entity Component System) -mallia, jossa peli koostuu entiteeteistä, komponenteista ja järjestelmistä.

Komponentit (Component) ovat datasäiliöitä, joiden avulla ylläpidetään pelitilaan liittyviä tietoja. Esimerkiksi pelihahmon sijainti, tekstuuri ja tiimi voidaan tallentaa komponentteina.

Entiteetit (Entity) ovat tunnuksia, joihin voidaan liittää vaihteleva määrä eri komponentteja. Pelihahmot, ammukset ja partikkeligeneraattorit ovat esimerkkejä mahdollisista entiteeteistä. Partikkeligeneraattoriin ja pelihahmoihin liittyy todennäköisesti hyvin erilaiset komponentit. Esimerkiksi partikkeligeneraattori ei tarvitse välttämättä tiimikomponenttia, mutta se voi tarvita komponentteja partikkelien käyttäytymiseen liittyen, joita taas pelihahmot eivät tarvitse.

Järjestelmät (System) prosessoivat entiteettejä lukemalla tai muokkaamalla niihin liittyviä komponentteja. Niiden käsittelemiltä entiteeteiltä täytyy löytyä systeemin vaatimat komponentit. Esimerkiksi yksinkertainen fysiikkajärjestelmä voisi vaatia entiteeteiltä sijainti-, nopeus- ja kiihtyvyyskomponentit toimintaansa varten. Toisaalta renderöintijärjestelmä voisi sen sijaan vaatia sijainti-, tekstuuri-, malli- ja sävytinkomponentit.

ECS-lähestymistapa on tehokas, koska eri komponentit voidaan tallentaa omiin yhtenäisiin tietorakenteisiinsa. Tämä on järkevää välimuistitehokkuuden kannalta, sillä järjestelmät käsittelevät peräkkäin suuren määrän samantyyppisiä komponentteja. ECS tekee myös selvän jaon datan ja toiminnallisuuden välillä, joka lisää pelimoottorin modulaarisuutta.

Rustissa on useita ECS-kirjastoja, joista yksi suosituimpia on *Specs* (Specs Parallel ECS)(*Are we game yet? - Rust*, s.a.-b). Specsiä käytetään esimerkiksi aiemmin mainitussa Amethyst-pelimoottorissa.

Specs pystyy ajamaan järjestelmiä samanaikaisesti, jos niiden lukemis- ja muokkausvaatimukset eivät ole päällekkäisiä komponenttisäiliöille. Siinä järjestelmille voidaan myös asettaa riippuvuussuhteita, jotka vaikuttavat järjestelmien ajamisjärjestykseen. Lisäksi Specsissä pystyy samanaikaistamaan komponenttiyhdistelmien läpikäyntiä järjestelmien sisällä.

Specissä eri komponenteille voidaan valita erilaisia säiliöitä niiden käyttötarkoituksen mukaan. $VecStorage$ käyttää sisäisesti yksinkertaista dynaamisesti kasvavaa taulukkoa, joka on muistitehokas, kun komponentti löytyy lähes kaikilta entiteeteiltä.

Jos komponentti ei löydy tarpeeksi monelta entiteetiltä, $VecStorage$:n sisäiseen taulukkoon jää suuria aukkoja, koska siinä entiteettien tunnukset ovat suoraan taulukon indeksejä. $DenseVecStorage$ korjaa tämän ongelman käyttämällä uudelleenohjaustaulukkomenetelmää. Siinä on datataulukon lisäksi kaksi aputaulukkoa, joiden avulla varsinainen data pystytään tallentamaan tiiviisti.

$HashMapStorage$:ssa komponentit tallennetaan hajautustauluun niin että entiteetti toimii avaimena ja komponentti arvona. Sen iteroiminen on hitaampaa, mutta se on muistitehokkaampi, jos komponentti liittyy vain pieneen määrään entiteettejä.

Komponentti, joka ei sisällä varsinaista dataa vaan toimii vain tunnistimena, kannattaa se säilöä $NullStorage$:ssa. Esimerkiksi jonkun tietyn vihollistyypin tekoälyjärjestelmä voi tunnistaa kaikki oikeantyyppiset entiteetit tämänlaisen komponentin avulla.

5.3 Grafiikka

Rustille on tehty suoria sidoksia yleisimmille grafiikkarajapinnoille, kuten OpenGL:lle ja Vulkanille. Rustille on myös olemassa grafiikkakirjastoja, jotka käyttävät näitä suoria sidoksia ytimessään, mutta tarjoavat Rust-ohjelmoijalle idiomaattisemman rajapinnan. Tällaisia kirjastoja ovat esimerkiksi gfx-rs ja glium.

Gfx-rs on abstrahoitu grafiikkakirjasto, joka noudattaa frontend-backend kahtiajakoa. Gfx-kirjaston frontend-osio on Vulkanin kaltainen, mutta sitä abstraktimpi rajapinta. Sille on toteutettu backendit Vulkanille, Direct3D 12:lle, Metalille ja OpenGL 2.1+/ES2:lle. Gfx on yksi suosituimpia grafiikkakirjastoja ja sitä tukevat esimerkiksi aiemmin mainitut Piston ja Amethyst

-pelimoottorit.

Glium on turvallinen rajapintakirjasto OpenGL:lle. Se pyrkii piilottamaan OpenGL:n tilakonemaisen toiminnan ja tarjoamaan Rust-kielelle soveltuvan tilattoman rajapinnan sen sijaan. Esimerkiksi OpenGL:n virhetilanteita pyritään välttämään täysin käännösaikaisesti. Vaikka Gliumin alkuperäinen kehittäjä on siirtynyt kehittämään Vulkano-kirjastoa, sen yhteisö ylläpitää sitä edelleen.

6 Yhteenveto

Seminaaritutkielmassa selostettiin Rustin historiaa ja sen tärkeitä periaatteita ohjelmoijan kannalta. Lisäksi tutkielmassa analysoitiin Rustille saatavia kehitystyökaluja sekä ohjelmistokehityksessä että peliohjelmoinnissa.

Rustille on nuoren ikänsä takia olemassa vähemmän kehitystyökaluja ja kirjastoja kuin muilla vakiintuneilla ohjelmointikielillä. Tästä huolimatta Rustille on saatavilla useisiin eri käyttötarkoituksiin, kuten peliohjelmointiin, soveltuvia kirjastoja. Ongelmaksi saattaa tosin muodostua kirjastojen ja työkalujen jatkuva muutos verrattuna alalla vakiintuneisiin kirjastoihin ja pelimoottoreihin.

Rust vaikuttaa ominaisuuksiensa puolesta sopivalta suorituskykyä vaativien ohjelmien, kuten pelien ja pelimoottorien kirjoittamiseen. Toisaalta, Rustin säännöt voivat aiheuttaa hankaluuksia tietyissä tilanteissa esimerkiksi, jos olioiden välillä on monimutkaisia viittaussuhteita. Tämä edellyttää usein muista kielistä poikkeavan lähestymistavan käyttöä.

Lähteet

Amethyst. (s.a.). Lainattu 2017-12-02, saatavilla `https://github.com/amethyst/amethyst/blob/develop/README.md#vision`

Amethyst | data-oriented and data-driven game engine. (s.a.). Lainattu 2017-12-03, saatavilla `https://www.amethyst.rs`

Anderson, B., Bergstrom, L., Goregaokar, M., Matthews, J., McAllister, K., Moffitt, J., & Sapin, S. (2016). Engineering the servo web browser engine using rust. Teoksessa *Proceedings of the 38th international conference on software engineering companion* (s. 81–89). New York, NY, USA: ACM. doi: 10.1145/2889160.2889229

Are we game yet? - rust. (s.a.-a). Lainattu 2017-10-26, saatavilla `http://arewegameyet.com/categories/engines.html`

Are we game yet? - rust. (s.a.-b). Lainattu 2017-11-30, saatavilla `https://arewegameyet.com/categories/ecs.html`

Common vulnerabilities and exposures. (s.a.). Lainattu 2017-12-07, saatavilla https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=buffer+overflow

Frequently asked questions - the rust programming language. (s.a.). Lainattu 2017-10-26, saatavilla https://www.rust-lang.org/en-US/faq.html

Friends of rust - the rust programming language. (s.a.). Lainattu 2017-12-04, saatavilla https://www.rust-lang.org/en-US/friends.html

Getting started with the ndk | android developers. (s.a.). Lainattu 2017-11-20, saatavilla https://developer.android.com/ndk/guides/index.html

Glossary. (s.a.). Lainattu 2017-12-02, saatavilla https://www.amethyst.rs/book/master/html/glossary.html

Inside a super fast css engine: Quantum css (aka stylo). (s.a.). Lainattu 2017-10-26, saatavilla https://hacks.mozilla.org/2017/08/inside-a-super-fast-css-engine-quantum-css-aka-stylo/

Introducing the new firefox: Firefox quantum. (s.a.). Lainattu 2017-12-04, saatavilla https://blog.mozilla.org/2017/11/14/introducing-firefox-quantum/

Klabnik, S., & Nichols, C. (2018). *The rust programming language.* No Starch Press. Lainattu saatavilla https://doc.rust-lang.org/book/second-edition

Levy, A., Andersen, M. P., Campbell, B., Culler, D., Dutta, P., Ghena, B., ... Pannuto, P. (2015). Ownership is theft: Experiences building an embedded os in rust. Teoksessa *Proceedings of the 8th workshop on programming languages and operating systems* (s. 21–26). New York, NY, USA: ACM. Lainattu saatavilla http://doi.acm.org.libproxy.helsinki.fi/10.1145/2818302.2818306 doi: 10.1145/2818302.2818306

Levy, A., Campbell, B., Ghena, B., Pannuto, P., Dutta, P., & Levis, P. (2017). The case for writing a kernel in rust. Teoksessa *Proceedings of the 8th asia-pacific workshop on systems* (s. 1:1–1:7). New York, NY, USA: ACM. Lainattu saatavilla http://doi.acm.org.libproxy.helsinki.fi/10.1145/3124680.3124717 doi: 10.1145/3124680.3124717

The redox operating-system. (s.a.). Lainattu 2017-10-26, saatavilla https://doc.redox-os.org/book/overview/what_redox_is.html

Rust inside other languages - rust. (s.a.). Lainattu 2017-11-01, saatavilla https://doc.rust-lang.org/1.2.0/book/rust-inside-other-languages.html

Rust platform support - the rust programming language. (s.a.). Lainattu 2017-11-20, saatavilla `https://forge.rust-lang.org/platform-support.html`

A Koodiesimerkit

```
fn main() {
    println!("Hei maailma");
}
```

```
$ cargo run
Hei maailma
```

Koodiliite 1: Tulostaa "Hei maailma"

```
fn main() {
    let s_luku: i32 = 4; // Valittu reilulla nopanheitolla.
    println!("Satunnaislukusi on {}", s_luku);
}
```

Koodiliite 2: Tulostaa satunnaisluvun

```
fn factorial(n: u64) -> u64 {
    let mut result = 1;
    for i in 1..(n + 1) {
        result *= i;
    }
    return result;
}
```

Koodiliite 3: Funktio joka laskee $n!$ iteratiivisesti.

```
fn factorial(n: u64) -> u64 {
    if n == 0 {
        1
    } else {
        n * factorial(n-1)
    }
}
```

Koodiliite 4: Funktio joka laskee $n!$ rekursiivisesti.

```rust
fn main() {
    let mut string = String::from("Hello, ");
    add_world(string);
    add_world(string);
}

fn add_world(mut string: String) {
    string.push_str("World!");
}
```

```
$ cargo run
error[E0382]: use of moved value: 'string'
 --> src/main.rs:4:15
  |
3 |     add_world(string);
  |               ------ value moved here
4 |     add_world(string);
  |               ^^^^^^ value used here after move
  |
  = note: move occurs because 'string' has type
  'std::string::String', which does not implement
  the 'Copy' trait
```

Koodiliite 5: Muuttujan käyttö omistajuuden siirron jälkeen.

```rust
fn main() {
    let mut list = vec![1,2,4,5,6,4,6,7];

    for n in &list {
        list.push(*n);
    }
}
```

```
$ cargo run
error[E0502]: cannot borrow ‘list‘ as mutable because it
is also borrowed as immutable
 --> src/main.rs:6:9
  |
5 |     for n in &list {
  |              ---- immutable borrow occurs here
6 |         list.push(*n);
  |         ^^^^ mutable borrow occurs here
7 |     }
  |     - immutable borrow ends here
```

Koodiliite 6: Iteraatio invalidaatio

```rust
fn ei_toimi<'a>() -> &'a String {
    let merkkijono = String::from("ei toimi");
    &merkkijono
}
```

```
$ cargo run
error[E0597]: `merkkijono` does not live long enough
 --> src/main.rs:3:6
  |
3 |     &merkkijono
  |      ^^^^^^^^^^ does not live long enough
4 | }
  | - borrowed value only lives until here
  |
note: borrowed value must be valid for the lifetime 'a as
defined on the function body at 1:1...
 --> src/main.rs:1:1
  |
1 | / fn ei_toimi<'a>() -> &'a String {
2 | |     let merkkijono = String::from("ei toimi");
3 | |     &merkkijono
4 | | }
  | |_^
```

Koodiliite 7: Roikkuva viittaus

```
fn ei_toimi(a: &String, b: &String) -> &String {
    println!("{}", a);
    b
}

fn toimii<'b>(a: &String, b: &'b String) -> &'b String {
    println!("{}", a);
    b
}
```

```
$ cargo run
error[E0106]: missing lifetime specifier
 --> src/main.rs:1:40
  |
1 | fn ei_toimi(a: &String, b: &String) -> &String {
  |                                        ^ expected
  |                                          lifetime
  |                                          parameter
  |
  = help: this function's return type contains a borrowed
  value, but the signature does not say whether it is
  borrowed from `a` or `b`
```

Koodiliite 8: Elinaikamerkinnän tarpeellisuus