

# **Rust peliohjelmoinnissa**

Victor Bankowski, Antti Karjalainen ja Janne Pulkkinen

Seminaariraportti  
HELSINGIN YLIOPISTO  
Tietojenkäsittelytieteen laitos

Helsinki, 7. joulukuuta 2017

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 Seminaariraportti		Aika — Datum — Month and year 7. joulukuuta 2017	Sivumäärä — Sidoantal — Number of pages 14
Tiivistelmä — Referat — Abstract  <p>Seminaariraportti tarkastelee Rust-ohjelmointikieltä, sen ominaisuuksia ja eroavaisuuksia muihin laajassa käytössä oleviin ohjelmointikieliin, ja sen soveltuvuutta peliohjelmointiin.</p> <p>Raportin toinen ja kolmas kappale käsittelevät Rust-ohjelmoinnin historiaa, perusteita ja ohjelmoinnissa tärkeitä piirteitä, kuten omistajuutta, lainaamista ja muuttujien elinikää. Ohjelmointikielen ominaisuuksia esitellään koodiesimerkeillä.</p> <p>Neljäs kappale keskittyy Rustin vertailuun C ja C++ -ohjelmointikielien kanssa käyttäen vertailukohteina käyttöjärjestelmiä ja kehitystyökaluja.</p> <p>Viides kappale esittelee Rustille saatavilla olevia peliohjelmointiin soveltuvia kirjastoja ja työkaluja.</p>			
Avainsanat — Nyckelord — Keywords ohjelmointikieli, peliohjelmointi			
Säilytyspaikka — Förvaringsställe — Where deposited			
Muita tietoja — Övriga uppgifter — Additional information			

# Sisältö

<b>1 Johdanto</b>	<b>1</b>
<b>2 Historia</b>	<b>1</b>
<b>3 Perusteet</b>	<b>1</b>
3.1 Omistajuus . . . . .	2
3.2 Lainaaminen ja elinajat . . . . .	3
<b>4 Vertailu C-kieliin</b>	<b>4</b>
4.1 Kehitystyökalut . . . . .	4
4.2 Käyttöjärjestelmät . . . . .	4
<b>5 Peliohjelmointi Rustilla</b>	<b>5</b>
5.1 Pelimoottorit . . . . .	5
5.2 ECS . . . . .	6
5.3 Grafiikka . . . . .	7
<b>6 Yhteenveto</b>	<b>8</b>
<b>Lähteet</b>	<b>8</b>
<b>A Koodiesimerkit</b>	<b>10</b>

# 1 Johdanto

Rust on käännettävä ohjelmointikieli, jonka kehitystä tukee Mozilla-säätiö (Anderson et al., 2016). Mozilla käyttää kieltä uuden rinnakkaisuutta hyödyntävän Servo -internet-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. Tämä on hyödyllistä 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 käännös-vaiheessa. 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 -web-selainmoottori. Sen tavoitteisiin kuuluu sivun piirtämisen, HTML-datan parsimisen ja muiden web-selaimen 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 -selaimen 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. Koodi 1 on kyseisen ohjelman Rust-toteutus. Kyseinen toteutus ei poikkea juurikaan muiden proseduraalisten kielten toteutuksista. Suurin ero muiden kielten toteutuksiin on se että tulostuskomento on toteutettu makrolla tekstin muotoilun helpottamiseksi. Koodissa 2 on esimerkki tästä.

Rustissa muuttujat määritellään käyttäen **let** avainsanaa ja muuttujan tyyppi erotellaan kaksoispisteellä. Koodissa 2 muuttuja nimeltä *s\_luku* on 32-bittinen etumerkillinen kokonaisluku. Muuttujat oletusarvoisesti eivät ole muokattavissa. Muokattavat muuttujat määritellään käyttäen avainsanayhdistelmää **let mut**.

Useimmissa tapauksissa muuttujan tyyppin voi jättää merkitsemättä, koska Rust osaa päätellä sen käännösaikana. Kuitenkin funktioiden parametrien ja palautusarvon tyytit täytyy merkata, koska Rustissa ei ole ohjelmanlaajuisia tyyppipäätelyä. Funktion palautusarvon tyyppi määritellään  $\rightarrow$  nuolen jälkeen. Koodissa 3 funktion parametri *n* ja palautusarvo ovat 64-bittisiä etumerkittömiä kokonaislukuja.

Rustin **for**-silmukat ovat *for-each*-tyyppisiä, joissa käydään iteraattorin kaikki alkiot läpi. Esimerkiksi koodissa 3 käydään kaikki välin  $[1, n + 1)$  kokonaislukuarvot läpi.

Rustissa ei tarvitse käyttää **return** avainsanaa, jos arvo palautetaan funktion lopussa. Tällöin ei myöskään merkitä puolipistettä.

### 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. Tietyissä suorituskykykriittisissä ongelmissa automaattinen roskienkeruu saattaa myös tehdä epäoptimaalisia ratkaisuja.

Muistinhallinnan optimointiongelmia esiintyy pelimoottoreissa esimerkiksi fysiikanmallinnuksessa. 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.

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, vastaa se manuaalista muistinhallintaa.

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 enää käyttää arvoa sillä se ei enää omista sitä.

Koodissa 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 pinossa. 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 vain omistajuus, niin funktioiden pitäisi palauttaa parametrisia, jotta niitä voitaisiin käyttää uudelleen. 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 koska muokkaaminen ei ole mahdollista lainauksia voi olla useita kappaleita samanaikaisesti (aliasointi). *Muokattavia lainauksia* voi sen sijaan olla vain yksi, mutta koska aliasointia ei ole niitä on mahdollista muokata. Samasta arvosta ei voi myöskään olla muokattavaa ja muokkaamatonta lainausta samanaikaisesti. Tämänlainen kahtiajako välttää muistiturvallisuusongelmia kuten iteraattorin mitätöityminen. Koodissa 6 on esimerkki iteraattori invalidaatio tilanteesta. 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 se olisi muistiturvallisuusriski. 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). Jotta tätä ei tapahtuisi, lainauksien oikeellisuutta pitää seurata jollain tavalla. Koodissa 7 on esimerkki juuri tällaisesta virheellisestä lainauksesta.

Rustissa kaikilla lainauksilla on 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. 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 pitkää palautusarvoa voi käyttää.

## 4 Vertailu C-kieliin

### 4.1 Kehitystyökalut

C ja C++ -ohjelmointikielet ovat Rustin tavoin käännettäviä ohjelmointikieliä, jotka soveltuvat suorituskykyä vaativien sovellusten kehittämiseen. Näihin 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 tai Amethyst, mutta Unityn tai Unreal Enginen kaltaisia kokonaisvaltaisia pelinkehityskokonaisuuksia ei toistaiseksi ole saatavilla Rust-kielille. (*Are we game yet?* - Rust, s.a.-a)

Rust-lähdekoodin kääntämiseen käytetään rustc -työkalua, joka on kirjoitettu Rust-kielillä. Kääntäjä hyödyntää LLVM-kääntäjäinfrastruktuurin työkaluja, ja hyötyy siten kyseistä projektia koskevista suorituskykyparametreista. (*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-kielen 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-kielillä kirjoitettuja kirjastoja on myös mahdollista käyttää Rustilla toteuttamalla kirjastolle sidonnat, jotka käyttävät kirjaston funktioita FFI (*Foreign Function Interface*) -rajapinnan kautta. C-kielillä kirjastoja ei siis tarvitse kirjoittaa uudelleen, jotta niitä voidaan hyödyntää Rust-kielissä. Tässä tapauksessa ohjelmointijärjestelmän tulee luoda kirjastolle turvalliset sidonnat, jotka estävät mahdolliset virhetilanteet esimerkiksi muistinhallinnan osalta.

### 4.2 Käyttöjärjestelmät

C++ -kieli ja sitä edeltävä C-kieli ovat yleisiä käyttöjärjestelmien ohjelmointikielinä: *Microsoft Windows* on ohjelmoitu C++ -kielillä, *FreeBSD* on ohjelmoitu käyttäen sekä C++ ja C-kieliä ja *Linux* käyttää C-kieltä. Rust olisi muistinhallinnan puolesta otollinen käyttöjärjestelmän ytimen ohjelmointiin, jossa muistinhallintavirheillä voi olla vakavia seurauksia järjestelmän vakauden ja tietoturvallisuuden kannalta. *Redox* on Rust-kielillä

kirjoitettu mikroydinrakennetta hyödyntävä 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. Tämä on vaatinut lisäyksiä Tockin käyttöjärjestelmäyttimeen, mikä ei muuten tukisi Rustin käyttämää muistinhallintaa ja siten kyseistä ohjelmointikieltä.

Rust tukee virallisesti 32-bittisiä 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 kirjastot ja sovellukset, mutta automatisoituja testejä ei ajeta julkaisun yhteydessä eikä niitä voi pitää siten yhtä toimintavarmoina. *iOS* ja *Android* -käyttöjärjestelmille on olemassa virallisesti tuetut C ja C++ -kieliä käyttävät kehitystyökalut. Esimerkiksi *Android NDK* sallii sovellusten kehittämisen C ja C++ -kieliä käyttäen, tarjoten myös kirjastoja 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 tasoisia pelimoottoreita. Kuitenkin on kehitteillä kirjastoja, kuten Piston ja Amethyst.

Näistä Piston 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 sekä sen ikkunointi- että sen 2d-grafiikkakirjastolla 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 logiikka ohjautuu mahdollisimman pal-



jon ulkoisen datan perusteella. Tällöin on mahdollista muokata ohjelman toimintaa ilman sen uudelleenkäntämistä. (*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 aliasioidin 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 suoritusnopeutta lisäämällä ajonaikaisia tarkistuksia. Lisäksi niiden käyttö siirtää osan vastuusta ylläpitää lainaamisen sääntöjä ohjelmoijalle.

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

Komponentit ovat datasäiliöitä, joita käytetään moniin eri peliin liittyviin toiminnallisiin. Esimerkiksi pelihahmon sijainti, nopeus, tekstuuri ja tiimi voidaan tallentaa komponentteina.

Entiteetit ovat vain tunnuksia, joihin voi liittää eri komponentteja. Peli-hahmot, ammuksien ja partikkeligeneraattorit ovat esimerkkejä entiteeteistä. Partikkeligeneraattoriin ja pelihahmoihin liittyy todennäköisesti hyvin erilaiset komponentit. Esimerkiksi partikkeligeneraattori ei tarvitse välttämättä tiimikomponenttia mutta se voi tarvita komponentteja partikkelien käyttäytymiseen liittyen, joita pelihahmot eivät tarvitse.

Järjestelmät prosessoivat entiteettejä lukemalla tai muokkaamalla niihin liittyviä komponentteja. Ne prosessoivat vain entiteettejä, joilta löytyy kaikki kyseisen järjestelmän tarvitsemat komponentit. Esimerkiksi yksinkertainen fysiikkajärjestelmä voisi vaatia entiteeteiltä sijainti-, nopeus- ja kiihtyvyysskomponentit toimintaansa varten. Renderöintijärjestelmä voisi taas 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). Specsia käytetään esimerkiksi aiemmin mainitussa Amethyst-pelimoottorissa.

Specs pystyy ajamaan järjestelmiä samanaikaisesti, jos niiden lukemis- ja muokausvaatimukset komponenttisäiliöille eivät ole päällekkäisiä. Siinä järjestelmille voidaan asettaa riippuvuussuhteita, jotka vaikuttavat niiden 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 entiteeteistä.

Komponentti, joka ei sisällä varsinaista dataa vaan toimii vain tunnistimenä, kannattaa säilöä *NullStorage*:ssa. Esimerkiksi jonkun tietyn vihollistyyppin 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-käyttäjälle idiomaattisen 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 grafiikkarajapinta. 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 sidoskirjasto OpenGL:lle. Se pyrkii piilottamaan OpenGL:n tilakonemaisen rajapinnan ja tarjoamaan Rust-kielille soveltuvan tilattoman rajapinnan sen tilalle. Se pyrkii esimerkiksi välttämään OpenGL:n virhetilanteita käännösaikaisesti. Vaikka Gliumin alkuperäinen kehittäjä on siirtynyt kehittämään Vulkano-kirjastoa, sen yhteisö ylläpitää sitä edelleen.

## 6 Yhteenveto

Seminaaritutkielmassa on selostettu Rustin historiaa ja tärkeitä periaatteita ohjelmoijan kannalta. Lisäksi tutkielmassa analysoitiin Rustille saatavia kehitystyökaluja sekä yleisessä ohjelmistokehityksessä ja peliohjelmoinnissa.

Rustille on olemassa vähäisen käyttöosuutensa takia vähemmän kehitystyökaluja ja kirjastoja kuin muille ohjelmointikielille. Tästä huolimatta Rustille on saatavilla useita eri käyttötapauksiin soveltuvia kirjastoja myös peliohjelmoinnin osalta. Ongelmaksi saattaa tällöin muodostua kielen ja sille saatavien työkalujen jatkuva muutos verrattuna alalla vakiintuneisiin kirjastoihin ja pelimoottoreihin kuten *Unity*.

Rust vaikuttaa ominaisuuksiensa puolesta sopivalta suorituskykyä vaativien ohjelmien kirjoittamiseen, johon kuuluvat pelit ja niiden alla toimivat pelimoottorit. Toisaalta, Rustin tarkistukset voivat tuottaa hankaluuksia tilanteissa, joissa olioiden välillä on monimutkaisia viittaussuhteita: nämä ovat erityisen mahdollisia peleissä, joissa tapahtuu hyvin paljon pelimaailman olioiden välistä kanssakäymistä. Tämä edellyttää muista kielistä poiketen erilaisen ohjelmointiparadigman käyttöä peliohjelmoinnissa.

## 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>

*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/>

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

*The redox operating-system.* (s.a.). Lainattu 2017-10-26, saatavilla [https://doc.redox-os.org/book/overview/what\\_redox\\_is.html](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
```

Koodi 1: Tulostaa "Hei maailma"

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

Koodi 2: Tulostaa satunnaisluvun

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

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

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

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

```

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

```

Koodi 5: Muuttujan käyttö omistajuuden siirron jälkeen.

```
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
```

Koodi 6: Iteraatio invalidaatio

```
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 | | }
  | |_^
```

Koodi 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`

```

Koodi 8: Elinaikamerinnän tarpeellisuus