

# **Rust peliohjelmoinnissa**

Victor Bankowski, Antti Karjalainen ja Janne Pulkkinen

Seminaariraportti  
HELSINGIN YLIOPISTO  
Tietojenkäsittelytieteen laitos

Helsinki, 3. 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 3. joulukuuta 2017	Sivumäärä — Sidoantal — Number of pages 9
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 avainsana 1, avainsana 2, avainsana 3			
Säilytyspaikka — Förvaringsställe — Where deposited			
Muita tietoja — Övriga uppgifter — Additional information			

## Sisältö

<b>1</b>	<b>Johdanto</b>	<b>1</b>
<b>2</b>	<b>Historia</b>	<b>1</b>
<b>3</b>	<b>Perusteet</b>	<b>2</b>
3.1	Omistajuus . . . . .	3
3.2	Lainaaminen ja elinajat . . . . .	5
<b>4</b>	<b>Vertailu C-kieliin</b>	<b>5</b>
4.1	Kehitystyökalut . . . . .	5
4.2	Käyttöjärjestelmät . . . . .	6
<b>5</b>	<b>Peliohjelmointi Rustilla</b>	<b>7</b>
5.1	Pelimoottorit . . . . .	7
5.2	ECS . . . . .	7
5.3	Grafikka . . . . .	8
	<b>Lähteet</b>	<b>8</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 käytetään missä?]. Käännettävänä ohjelmointikielenä C ja C++ -kielten tavoin Rust mahdollistaa suorituskäytön ja hallittua muistin käyttöä vaativien sovellusten kehittämisen esimerkiksi sulautetuissa järjestelmissä. Edellä mainituista kielistä poiketen Rust kuitenkin estää yleisiä C-kielissä esiintyviä muistinhallintaa ja kilpatilanteita koskevia ongelmia, mahdollistaen kuitenkin vastaavan suorituskäytön ajettavassa ohjelmassa. Rust ratkaisee nämä ongelmat käyttämällä muistinhallinnassa omistajuuden (”ownership”) ja lainaamisen (”borrowing”) käsitteitä. Tämä estää mahdolliset virhetilanteet jo ohjelman käännösvaiheessa vaatimatta virtuaalikoneen, kääntäjän tai tulkin käyttöä ohjelman suorituksen aikana.

# 2 Historia

Rust-kielen kehitys alkoi vuonna 2006 Graydon Hoaren sivuprojektina, jollaisena se jatkui yli kolmen vuoden ajan<sup>1</sup>. 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<sup>2</sup>. Servo-projektiin kuuluva CSS-moottori Stylo on otettu käyttöön Mozilla Firefox -selaimen uusissa kehitysversioissa<sup>3</sup>.

Muihin Rust-kieltä hyödyntäviin organisaatioihin kuuluu muun muassa Dropbox ja Canonical<sup>4</sup>.

---

<sup>1</sup><https://www.rust-lang.org/en-US/faq.html>

<sup>2</sup><https://hacks.mozilla.org/2017/08/inside-a-super-fast-css-engine-quantum-css-aka-stylo/>

<sup>3</sup><https://blog.mozilla.org/blog/2017/09/26/firefox-quantum-beta-developer-edition/>

<sup>4</sup><https://www.rust-lang.org/en-US/friends.html>

### 3 Perusteet

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

```
$ cargo run  
Hei maailma
```

Koodi 1: Tulostaa "Hei maailma"

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 makro. Tulostuskomento on toteutettu makrolla tekstin muotoilun helpottamiseksi. Koodissa 2 on esimerkki tästä.

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

Koodi 2: Tulostaa satunnaisluvun

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

```
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.

Useimmissa tapauksissa muuttujan tyyppin voi jättää merkkeamatta, koska Rust osaa päätellä sen käännösaikana. Kuitenkin funktioiden parametrien

ja palautusarvon tyypit täytyy merkata, koska Rustissa ei ole ohjelmanlaajuista tyyppipäätelyä. Funktion palautusarvon tyyppi määritellään nuolen  $\rightarrow$  jälkeen. Koodissa 3 funktion parametri  $n$  ja palautusarvo ovat 64-bittisiä etumerkittömiä kokonaislukuja.

Rustin **for**-silmukat ovat *for-each*-tyyppisiä, jossa 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 merkata puolipistettä.

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

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

### 3.1 Omistajuus

Muistinhallinta on tärkeä osa ohjelmien toimintaa. Tätä varten monissa kielissä käytetään automaattista roskienkeruuta. Tämä vähentää ohjelmoijan vastuuta, mutta samalla myös vähentää mahdollisuuksia vaikuttaa muistinhallintaan. Tietyissä suorituskyykriittisissä ongelmissa automaattinen roskienkeruu saattaa tehdä epäoptimaalisia ratkaisuja. Tällaisia ongelmia esiintyy pelimoottoreissa esimerkiksi fysiikanmallinmuksessa. Tämän takia pelimoottorit toteutetaan usein kielillä joissa ei muistinhallinta on manuaalista. Manuaalisesti muistinhallitsevissa kielissä jää ohjelmoijan vastuuksi varata ja vapauttaa muisti oikeaoppisesti, josta johtuen ne ovat alttiita muistihallintavirheille.

Rust-ohjelmointikielessä ei käytetä automaattista roskienkeruuta, mutta siinä ei myöskään jätetä muistinhallintaa täysin ohjelmoijan vastuulle. Tämä on mahdollista, koska kieli takaa sen että jokaisella varatulla muistialueella on yksikäsittäinen omistaja, joka on vastuussa sen vapauttamisesta. Koska omistajuus on käännösaikainen käsite Rustissa, 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 omistajuus siirtyy. Tämän jälkeen alkuperäinen muuttuja ei voi enää käyttää arvoa sillä se ei 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. Syväkopiointin vastakohta on *pinnallinen kopiointi*. (Shallow copy) Koska siirto on syväkopio *Copy*-traitin toteuttaville tyypeille, ei tämän tyyppisillä muuttujilla ole siirtos.

```
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.

Ohjelmointikielet vaativat muuttujien luomisen lisäksi niiden poistamisen. Käännetyissä kielissä tämä voi vaatia ohjelmoijalta muuttujien manuaalisen poistamista, kun niitä ei enää käytetä. Esimerkiksi C käyttää manuaalista muistinhallintaa. Useissa tulkkia käyttävissä kielissä muuttujaan kohdistuvia viittauksia lasketaan suorituksen aikana; kun muuttujaan ei enää viitata, automaattinen roskienkeräys ennen pitkää löytää käyttämättömän muuttujan ja poistaa sen. Tämä ei vaadi ohjelmoijalta yleensä lisätöitä, mutta

roskankerääjä lisää ohjelman resurssivaatimuksia. Roskienkeräystä käyttäviin kieliin kuuluvat muun muassa Python ja Java.

Rust varmistaa muuttujien muistinhallinnan käyttämällä omistamisen käsitettä, joka ei vaadi suorituksen aikaisen roskankeruun käyttöä, eikä myöskään muuttujien manuaalista poistamista. Kun muuttuja luodaan missä tahansa osassa koodia, sille määritellään omistaja muuttujan vaikutusalueen perusteella. Esimerkiksi metodissa main luodut muuttujat saavat omistajakseen kyseisen metodin. Kun kyseisen main-metodin suoritus on päättynyt, kyseisen metodin omistamat muuttujat poistetaan välittömästi ja niiden varaama muistialue palautetaan järjestelmän käyttöön. Tätä periaatetta on havainnollistettu esimerkissä 2.

### 3.2 Lainaaminen ja elinajat

Rust-kielen muistinhallintaan kuuluu omistajuuden lisäksi kaksi merkittävää alakäsitettä: lainaaminen ja elin aika. Lainaaminen tarkoittaa muuttujan viittauksen antamista väliaikaisesti johonkin toiseen skoppiin; esimerkiksi toiseen metodiin. Useimmista kielistä poiketen Rust kuitenkin määrittää invariantin, joka tarkastetaan käännösvaiheessa: muuttujalla voi olla joko useita muuttumattomia (vain-luku) viittauksia, yksi muuttava (luku ja kirjoitus) viittaus, mutta ei kummankin tyyppistä viittausta samanaikaisesti. Tällä estetään virhetilanteet, jossa yksi tai useampi taho lukee muuttujaa samaan aikaan kun sitä muutetaan. Kielen standardikirjastossa on saatavilla erilaisia synkronointialkioita, jotka sallivat esimerkiksi ”yksi kirjoittaja, usea lukija” -malliset tilanteet.

## 4 Vertailu C-kieliin

### 4.1 Kehitystyökalut

C ja C++ -ohjelmointikielien ovat Rustin tavoin käännettäviä ohjelmointikieliä, jotka soveltuvat suorituskäyttöä 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 suorituskäyttöparametreista. (*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 muuta tulkia, Rust-koodia on C-kielten tavoin mahdollista hyödyntää korkeamman tason kielissä kuten Pythonissa tai Rubyssa. Tässä tapauksessa sovelluksen tehokasta suorituskkyä 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 uudelleenkirjoittaa, jotta niitä voidaan hyödyntää Rust-kielissä. Tässä tapauksessa ohjelmoinjan 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 muistinhallinta-virheillä voi olla vakavia seurauksia järjestelmän vakauden ja tietoturvallisuuden kannalta. *Redox* on Rust-kielillä kirjoitettu mikroydin-rakennetta hyödyntävä käyttöjärjestelmä. (*The Redox Operating-System*, s.a.) Redox on vielä aikaisessa kehitysvaiheessa eikä siten sovellu jokapäiväiseen käyttöön.

*Tock* on Rust-kielillä 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 käyttöjärjestelmäyttimeen, mikä ei muuten tukisi Rustin käyttämää muistinhallintaa.

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 käännetty 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 logiikkaa ohjautuu mahdollisimman paljon ulkoisen datan perusteella. Tällöin on mahdollista muokata ohjelman toimintaa ilman sen uudelleenkäyntämistä. (*Glossary*, s.a.)

Amethystin ominaisuuksiin kuuluvat yksinkertainen pelitilan hallinta pinoautomaatilla, skriptaus API, 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 aliasoinnin ja muokattavuuden samanaikaisuuden. Tämä vaikuttaa pelimoottorin suunnittelussa tehtäviin rakententeellisiin valintoihin. Esimerkiksi syklisten viittausten käyttö vaikeutuu huomattavasti. Tällaisia viittauksia voi esiintyä esimerkiksi maailman ja sen sisältävien olioiden välillä. Rustissa sykliset viittaukset voidaan toteuttaa älysoitin yhdistelmillä, mutta niiden käyttö vähentää suoritusnopeutta lisäämällä ajonaikaisia tarkistuksia. Lisäksi niiden käyttö siirtää osan vastuusta ylläpitää lainaamisen sääntöjä ohjelmoijalle.

Vaihtoehtoinen tapa strukturoida peli on käyttää ECS (Entity Component System) -mallia, jossa peli koostuu entiteetteistä, komponenteista ja

systeemeistä.

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

Entiteetit ovat vain tunnuksia, joihin voi liittää eri komponentteja. Peli-hahmot, ammukset 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.

Systeemit prosessoivat entiteettejä lukemalla tai muokkaamalla niihin liittyviä komponentteja. Ne prosessoivat vain entiteettejä, joilta löytyy kaikki kyseisen systeemin tarvitsemat komponentit. Esimerkiksi yksinkertainen fysiikkasysteemi voisi vaatia entiteeteiltä sijainti-, nopeus- ja kiihtyvyysskomponentit toimintaansa varten. Renderöintisysteemi 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 systeemit prosessoivat 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 systeemejä samanaikaisesti, jos niiden lukemista ja muokausvaatimukset komponenttisäiliöille eivät ole päällekkäisiä. Siinä systeemeille voidaan asettaa riippuvuussuhteita, jotka vaikuttavat niiden ajamisjärjestykseen. Lisäksi Specsissä pystyy samanaikaistamaan komponenttiyhdistelmien läpikäyntiä Systeemeiden sisällä.

Specissä eri komponenteille voidaan valita erilaisia säiliöitä niiden käyttötarkoituksen mukaan.

### 5.3 Grafiikka

Rustissa on sitomiskirjastoja

## 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#how-fast-is-rust>

*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>

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>