Henrik Paananen

Skriptikielen toteutus pelimoottoriin

Tietotekniikan kandidaatintutkielma

20. joulukuuta 2012

Jyväskylän yliopisto

Tietotekniikan laitos

Tekijä: Henrik Paananen

Yhteystiedot: henrik.j.paanane@student.jyu.fi

Ohjaajat: Anneli Heimbürger ja Sanna Mönkölä

Työn nimi: Skriptikielen toteutus pelimoottoriin

Title in English: Scripting language implementation for game engines

Työ: Kandidaatintutkielma

Suuntautumisvaihtoehto: Ohjelmistotekniikka

Sivumäärä: 29+3

Tiivistelmä: Tämä tutkimus esittää yksinkertaisen skriptikielen SimpScript. Kielen tarkoituksena on toimia pelien skriptikielenä. Jotta SimpScriptin voitiin todeta soveltuvaksi pelinkehitykseen, suoritettiin sen ja muutaman yleisesti peleissä käytetyn skriptikielen ominaisuuksien vertailu. Lopputuloksena huomattiin, että SimpScriptistä puuttuu monia ominaisuuksia, joita muut ratkaisut sisältävät. Kuitenkin SimpScriptin jatkokehityksen myötä, nämä ominaisuudet olisi toteutettavissa.

Avainsanat: skriptikieli, toteutus, pelimoottori, skriptimoottori, vertailu

Abstract: In this research a simple scripting language SimpScript is introduced. The purpose of the language is to be used in game scripting. To prove that SimpScript is applicable to game development a comparison was conducted between it and some common scripting languages used in games. As a result it was discovered that many features that are present in the other scripting languages were still missing from SimpScript. These features could be implemented to the language with some further development.

Keywords: scripting language, implementation, game engine, scripting engine, comparison

Termiluettelo

Staattinen tyypitys	Ohjelmointikielen ominaisuus päättää muuttujien, funktioiden ja mahdollisten muiden kielen primitiivien tyypitys käännösaikana.
Dynaaminen tyypitys	Staattisen tyypityksen vastakohta. Kielen primitiiveillä ei ole tyyppiä. Ainoastaan ajonaikaisilla arvoilla on tyyppi ja tyypitys siis tapahtuu vasta suoritusvaiheessa.

Kuviot

Sisältö

1 Johdanto

Peliohjelmoinnissa on jo pitkään toteutettu ajatusta pelin uudelleenkäytettävien osien ja pelilogiikan toisistaan erottamista. Tällä ajatusmallilla päädytään laajennettavaan ja uudelleenkäytettävään pelimoottoriin. Jotta pelilogiikka voitaisiin erottaa pelimoottorista selkeästi, ja että sitä voitaisiin muokata mahdollisimman helposti, voi olla järkevintä toteuttaa pelimoottoriin skriptausmahdollisuus erillisellä skriptikielellä (Anderson 2011).

Pelilogiikan kirjoittaminen korkeamman tason ohjelmointikielellä on siis suotavaa, koska se helpottaa pelilogiikan ymmärtämistä ja nopeuttaa pelin kehittämistä – varsinkin kokeiluvaiheessa (engl. *prototyping phase*). Kehityssyklin nopeutuminen perustuu siihen, että pelin skriptejä voidaan muokata ja ladata uudelleen pelin ajoaikana ilman, että peliä tarvitsee kääntää tai käynnistää uudelleen (Anderson 2011). Toisaalta skriptikielen syntaksi ja muoto on myös yleensä pelimoottorin toteutuskieleen verrattuna selkeämpää ja nopeampaa kirjoittaa (Anderson 2011). Joskus skriptikielen muoto voi olla tarpeeksi yksinkertaista, että myös esimerkiksi pelin kenttäsuunnittelijat voivat sitä käyttää. Tällöin kenttäsuunnittelijat voivat itsenäsesti prototyypittää erilaisia skriptattuja tapahtumia pelikenttiin vaivaamatta kiireisiä pelisuunnittelijoita.

Mahdollisuudet erilaisiin skriptattavuuden toteutuksiin ovat lähes rajattomat. Monet upotettavat yleiskäyttöiset skriptikielet soveltuvat myös pelimoottorin skriptaamiseen. Joitain suosituimpia pelejen kanssa käytettyjä kieliä ovat esimerkiksi Lua, Python, JavaScript ja UrealScript. Näistä Lua, Python ja JavaScript ovat yleistarkoituksellisia skriptikieliä, kun taas UnrealScript on tarkoitettu nimenomaan Unreal Engine -pelimoottorin skriptaamiseen. Kaikki neljä joko ovat oliopohjaisia tai tukevat olio-ohjelmointia. Niille yhteistä on siis korkea abstraktiotaso.

Monesti valmiit ratkaisut ovat tarpeeksi hyviä, mutta välillä tehokkuus syistä (suorituskyky, muistin kulutus tai kehitysnopeus) voi olla järkevämpää toteuttaa omalle pelimoottorille dedikoitu skriptikieli ja skriptimoottori. Toisaalta omassa skriptausratkaisussaan voi myös tuoda pelille ominaisia ominaisuuksia helposti saataville esimerkiksi erikoistuneen syntaksin kautta (Anderson 2011).

Tässä tutkimuksessa esitetään yksinkertainen pelilogiikan ohjelmointiin soveltuva skriptikieli SimpScript. Kielen soveltuvuutta peliohjelmointiin arvioidaan vertaamalla sitä muihin yleisesti peliohjelmoinnissa käytettyihin kieliin. Luvussa kaksi käydään läpi yleistä skriptikielien ja skriptimoottorien taustaa sekä niiden toteuttamiseen vaadittavia osia. Luvussa neljä esitellään SimpScript. Siinä kerrotaan kielen syntaksista, kääntäjän ja virtuaalikoneen toteutuksesta sekä skriptimoottorin liittämistä isäntäjärjestelmään. Luku viisi vertailee SimpScriptiä muihin skriptausratkaisuihin. Lopuksi johtopäätöksissä arvioidaan SimpScript-kielen onnistumista ja erityisen pelejä varten suunnitellun skriptikielen toteuttamisen järkevyyttä. Lisäksi esitetään SimpScriptin jatkokehityksen vaatimuksia.

2 Tausta

Pelimoottorin skriptaamiseen tarvitaan niin kutsuttu skriptimoottori (engl. *scripting engine*). Se koostuu osista, jotka hoitavat skriptien kääntämisen ja suorittamisen sekä niiden liittämisen isäntäjärjestelmään. Tässä tapauksessa isäntäjärjestelmä on siis pelimoottori.

Riippuu pelimoottorista kuinka se käyttää skriptausmahdollisuuksia hyväkseen. Erilaiset pelien skriptausjärjestelmät voidaan luokitella niiden käyttötarkoitusken mukaan. Andersonin (2011) tutkimuksen luokittelussa yksinkertaisimmillaan skripti tiomii vain alustaakseen pelimaailman muuttujia. Jotkin skriptit suoritetaan vain tietyn tapahtuman sattuessa. Osa taas toimii jokaisella pelin suorituskierroksella päivittääkseen pelimaailmaa. (Anderson 2011).

2.1 Skriptikielet

Skriptikielen määritelmä vaihtelee kohdealueesta riippuen. Yleensä vastakkain asetetaan systeemiohjelmointikielet ja skriptikielet. Ousterhout (1998) esittää skriptikielen ja systeemiohjelmointikielen eroiksi kaksi asiaa: skriptikieli on yleensä tyypitön ja se on tulkattu. Tämä ei tietenkään anna juurikaan kuvaa siitä, mikä on skriptikieli ja mikä ei.

Bloom et al. (2009) karakterisoivat skriptikielien yleisiksi ominaisuuksiksi kevyen syntaksin, heikon tietokapseloinnin, dynaamisen tyypityksen, tehokkaat koostetietotyypit ja mahdollisuuden suorittaa valmiit osat ei-valmiista ohjelmasta.

Wikipedia (*Wikipedia: "Scripting Language"*) taas listaa skriptikielien eri muotoja: työn ohjaus- ja komentokielet (engl. *job control languages and shell*), GUI skriptaus, sovelluskohtaiset kielet, tekstinkäsittelykielet ja jatkekielet/upotetut kielet (engl. *extension/embeddable languages*). Tässä luokituksessa peliohjelmoinnissa käytettyt skriptikielet voivat kuulua joko sovelluskohtaisiin tai upotettuihin kieliin.

Anderson (2011), tutkiessaan pelien skriptausjärjestelmien luokitusta, käyttää skriptikielen määritelmänä kieltä tai systeemiä, joka mahdollistaa ohjelmalogiikan muokkaamisen ilman uudelleen kääntämisen tarvetta. Tämä määritelmä sopii hyvin pelien skriptikieliin.

2.2 Kääntäjätekniikka

Koska lopullinen päämäärä on saada lähdekoodin esittämä ohjelma suoritetuksi, pitää se muuntaa ensin sopivaan muotoon. Tätä ohjelmakoodin saattamista tietokoneen ymmärtämään muotoon kutsutaan kääntämiseksi. Tästä vastaa kyseistä kieltä varten suunniteltu kääntäjä. Kuten Mogensen (2010) kertoo, yleensä kääntäjä toteutetaan useista peräkkäin suorittavista osista. Osat linkittyvät toisiinsa siten, että aina edellinen osa tuottaa seuraavalle syötteen. (Mogensen 2010). Jako osiin on tarpeen, sillä kääntäjän toteuttamien yhdestä osasta voi olla hyvin haastavaa.

Mukaillen Mogensenin esittämää mallia kääntäjän rakenne vastaa yleensä seuraavaa:

1. **Leksikaalinen analyysi (tokenisoija)**
   Tekstuaalisesta lähdekoodista erotellaan merkkijonoja, joka vastaavat kielen perusyksiköitä eli tokeneita.
2. **Syntaktinen analyysi (parseri)**
   Tokenijonosta parsitaan kielen syntaksia vastaavia lauseita, joista muodostetaan abstrakti syntaksipuu.
3. **Semanttinen analyysi ja tyyppitarkistus**
   Syntaksipuu käydään läpi ja varmistetaan, että jokainen lause on oikeaoppinen. Samalla suoritetaan syntaksipuun solmujen tyypitys tarvittavilta osin.
4. **Välimuotoisen koodin generointi**
   Syntaksipuusta muodostetaan yksinkertaisia peräkkäisiä käskyjä.
5. **Rekisterien varaus**
   Muuttujille varataan rekisterit.
6. **Konekielisen tavukoodin generointi**
   Muodostetaan lopullinen konekielinen tavukoodi, jossa käskyt on muutettu niitä vastaviksi luvuiksi.

Näiden vaiheiden lisäksi on yleistä suorittaa eri määrä optimointivaiheita. Tämä rakenne voi vaihdella siten, että siitä puuttuu osa vaiheista tai ne suoritetaan eri järjestyksessä. Toisaalta eri kääntäjät saattavat lisätä uusia vaiheita yllä esitettyjen väliin. (Mogensen 2010).

2.3 Virtuaalikoneet

Jotta isäntäjärjestelmä pystyy suorittamaan skriptikieltä on sen ensin muunnettava tekstuaalinen lähdekoodi sellaiseen muotoon, että sitä voidaan suorittaa. Ohjelman suorituksen hoi-

taa erityinen tulkki, joka suorittaa kääntäjän tuottamaa esitystä alkuperäisestä lähdekoodista. Diehl, Hartel ja Sestoft (2000) kutsuvat näitä tulkkeja yhteisellä nimellä abstrakti kone tai virtuaalikone.

Erilaiset tulkit tulkitsevat eri vaiheen käännöstä. Yleensä tulkilla käsitetään nimenomaan sellaista suorituskonetta, joka suorittaa lause lauseelta kääntämätöntä ohjelmakoodia. Tällä tavoin suoritettavaa ohjelmointikieltä kutsutaan tulkatuksi kieleksi (engl. *interpreted language*). Mogensenin (2010) mukaan tulkki toimii kuten kääntäjä luodessaan abstraktin syntaksipuun solmuja lause lauseelta lähdekoodista. Tulkki vaan ei tuota lopullista konekieltä käännöksestä, vaan suorittaa ohjelmaa evaluoimalla näitä yksittäisiä syntaksipuun solmuja. Tällöin siis kääntäjän tarvitsee suorittaa vain vaiheet leksikaalisesta analyysista syntaktiseen tai semanttiseen analyysiin asti. Tämän lisäksi tulkki kääntää koodin palan aina joka kerta alusta asti kun se siihen törmää.

Toinen tapa tulkkaamiseen on kääntää ensin lähdekoodi tavukoodiksi, jota tulkkaava virtuaalikone sitten suorittaa. Virtuaalikoneen tulkitsemaa ohjelmointikieltä ei yleensä sanota tulkatuksi kieleksi, vaan esimerkiksi virtuaalikoneella suoritettavaksi kieleksi (engl. *virtual machine executed language*). Tavukoodi on yksittäisistä peräkkäin suoritettavista käkyistä koostuva jono, joka kuvaa ohjelman suorituksen. Diehl, Hartel ja Sestoft (2000) kertovat, että tällaista käskyjonoa suorittava virtuaalikone sisältää yleensä ohjelmasäilön ja tilan. Koneen tila taas sisältää pinon ja eri määrän rekistereitä. Rekistereistä yksi, niin kutsuttu ohjelmalaskuri (engl. *program counter*), pitää yllä tietoa siitä mikä käsky suoritetaan seuraavaksi. Ohjelman suoritus tapahtuu suoritussilmukassa (engl. *execution loop*), jossa käsitellään yksi käsky kerrallaan edeten tavukoodia ohjelmalaskurin mukaan. (Diehl, Hartel ja Sestoft 2000).

Andersonin (2011) mukaan, vaikka monet skriptikielet ovat tulkattuja yleensä kuitenkin skriptit käännetään ensin virtuaalikoneen ymmärtämäksi tavukoodiksi. Tällöin saadaan skriptien suorituksesta nopeampaa verrattuna lause lauseelta tulkattuihin (Anderson 2011).

2.4 Skriptimoottori

Skriptimoottori voidaan käsittää skriptikielen toteutuksena. Se sisältää kaiken mitä tarvitaan skriptien suorittamiseen isäntäjärjestelmässä. Siihen kuuluu siis skriptikielen kääntäjä, käännöksen tulkkaaja (tulkki tai virtuaalikone) ja isäntäjärjestelmän ja skriptijärjestelmän yhteen liittämiseen vaadittavat osat.

3 Yleiset pelien skriptikielet

Vielä vähän aikaa sitten peleissä ei ollut mahdollisuutta käyttää järeitä skriptausratkaisuja johtuen laitteiston rajallisesta suorituskyvystä. Nykyään kuitenkin on mahdollista totetuttaa yhä suurempi osa pelilogiikasta jollain tuottavuutta nostavalla skriptikielellä.

3.1 Esimerkkejä

Seuraavaksi esitetään muutamia esimerkkejä eri pelimoottoreissa tai pelieissä käytetyistä skriptikielistä. Epic Gamesin Unreal Engine -pelimoottori käyttää omaa UnrealScript skriptikieltä (*UnrealScript*). Samoin GarageGames käyttää Torque 3D -pelimoottorissa omaa TorqueScript-kieltä (*GarageGames*). Unity-pelimoottori mahdollistaa muista poiketen kahden eri skriptikielen käytön: C# ja JavaScript (*Unity scripting*). Wikipedia(*Wikipedia: "Category:Lua-scripted video games"*) listaa Lua-skriptikieltä käyttäviä pelejä valtavat määrät, joista esimerkkejä ovat Blizzardin World of Warcraft, Crytekin Far Cry ja suomalaisen Bug Bearin Flat Out 2. Blender 3d-sisällön tuotanto-ohjelma sisältää pelimoottorin Blender Game Engine, joka käyttää skriptauksessa Pythonia (*Blender*). Myös indie pelinkehittäjät käyttävät nykyään skriptaamismahdollisuuksia. Esimerkiksi Frictional Gamesin HPL1-moottori (*Frictional Games Wiki*) ja Wolfire Gamesin Overgrowth-peli (*Wolfire Games*) käyttävät molemmat AngelScript-kieltä.

On huomattava, että Epic Gamesin tuleva Unreal Engine 4 -pelimoottori ei enää sisällä UnrealScript-kieltä, vaan skriptaaminen hoidetaan visuaalisella skriptityökalu *Kismetillä*[1] tai suoraan C++-kielellä (Schultz 2012). Samoin Crytekin CryEngine 3 mahdollistaa pelilogiikan ohjaamisen visuaalisen työkalun, *Flow Graph Editorin*, avulla (*Flow Graph Editor*). On siis huomattavissa, että tulevaisuudessa pelien skriptaaminen halutaan tehdä mahdollisimman helpoksi, jolloin luonnollisena vaihtoehtona on toteuttaa tämä visuaalisella ohjelmointityökalulla. Tässä tutkimuksessa keskitytään kuitenkin perinteisiin tekstuaalisiin skriptausvaihtoehtoihin.

Seuraavaksi tarkastellaan joitain yleisten peleissä käytettyjen skriptikielien ominaisuuksia. Tarkasteluun valittuja kieliä ovat Pyhton, Lua, UnrealScript, TorqueScript sekä AngelScript.

1. Kismet on mukana jo Unreal Engine 3:ssa (*UnrealScript*)

3.2 Dynaamisuus

Vaikka yleensä skriptikielet ovat dynaamisia (Bloom et al. 2009), ei pelien skriptikielissä aina näin ole. Esimerkkinä juuri UnrealScript ei ole dynaaminen, vaan sen tyypitys on staattinen. Toisaalta TorqueScript, vaikkakin pelimoottorikohtainen, on dynaaminen (*GarageGames*). Samoin yleiset skriptikielet Python ja Lua ovat dynaamisia (*Python Programming Language* 2012; *The Programming Language Lua* 2012). Vaikka AngelScript ei ole varsinaisesti pelkästään pelien ohjelmointiin tarkoitettu, on se staattinen tyypitykseltään (*AngelScript* 2012). Tähän vaikuttanee sen vahva sidos C++-kieleen, joka on staattisesti tyypitetty.

On siis huomattu, että peleissä käytettyjä skriptikieliä on sekä staattisia että dynaamisia riippumatta siitä, onko kyseessä yleiskäyttöinen vai peleille suunnattu kieli. Tämä viittaa siihen, että skriptikielen dynaamisuudella ei olisi liiaksi vaikutusta kielen soveltuvuudelle peliohjelmointiin.

3.3 Muistinhallinta

Yleensä korkean abstraktiotason skriptikielissä ei erityisesti tarvitse huolehtia muistinkäsittelystä vaan se tapahtuu automaattisesti. Tämä tarkoittaa sitä, että skriptin varaama muisti vapautetaan automaattisesti, ilman ohjelmoijan erillistä käskyä, kun sitä ei enää käytetä. Tämä tietenkin helpottaa ohjelmoijan työtä, koska tällöin hän voi keskittyä paremmin ylemmän tason logiikan suunnitteluun. Ongelmiakin automaattisesta muistinhallinnasta saattaa syntyä. Esimerkiksi automaattisen roskienkeruun ajoitusta on vaikea arvioida. Jos roskienkeruu tapahtuu väärään aikaan, voi pelin suorituskyky pudota.

Kaikki tarkasteltavat kielet tukevat automaattista muistinhallintaa (*Python Programming Language* 2012; *The Programming Language Lua* 2012; *UnrealScript Language Reference; AngelScript* 2012). TorqueScriptin osalta asiaa ei voitu tarkistaa, sillä Garage Games ei asiasta mainitse sivuillaan. Kuitenkin tarkasteltaessa TorqueScript-esimerkkejä on havaittavissa, että eksplisiittistä muistinhallintaa ei tarvita.

3.4 Upotettavuus

Pelien skriptikielille on erittäin tärkeää upotettavuus (engl. *embeddability*). Upotettavuudella tarkoitetaan sitä, kuinka hyvin kielen pystyy liittämään toiseen järjestelmään – pelien tapauksessa siis pelimoottoriin. Ilman upotettavuutta ei skriptikieltä pysty siis käyttämään peliskriptaukseen.

UnrealScriptin ja TorqueScriptin osalta upotettavuus on varsin vähäistä, mutta riittävää. UnrealScript on osa Unreal Engine -pelimoottoria, eikä sitä siis ole mahdollista käyttää muissa isäntäjärjestelmissä. Samoin TorqueScript on osa Torque 3d -pelimoottoria, eikä toimi yleisenä skriptiratkaisuna. Molemmat ovat siis pelimoottorikohtaisia.

Sen sijaan Lua, Python ja AngelScript ovat yleisiä skriptikieliä ja niiden upottaminen eri isäntäjärjestelmiin on mahdollista. Lua on kirjoitettu C-kielellä ja sen väitetään kääntyvän kaikilla alustoilla, joille on C-kääntäjä (*The Programming Language Lua* 2012). Sen käyttäminen useista erikielisistä isäntäjärjestelmistä on mahdollista ja sen helppoutta on pidetty erityisesti silmällä kielenkehityksen aikana (Ierusalimschy, Figueiredo ja Celes 2007). Lisäksi Luan integroimista osaksi isäntäjärjestelmää helpottaa sille kehitetty Luabind-kirjasto, joka mahdollistaa Luan liittämisen C++-kieliseen ohjelmaan template-metaohjelmoinnilla (*Luabind*).

Pythonille on useita eri toteutuksia, joista ainakin perinteinen toteutus, CPython, on myös kirjoitettu C:llä. Pythonin liittäminen C- tai C++-kieliseen isäntäohjelmaan tapahtuu helpohkosti kattavan rajapinnan kautta. Kuten Lualle, myös Pythonille löytyy liittämistä helpottavia kirjastoja, joista esimerkkinä Boost.Python (*Boost.Python*). Näiden kielien lisäksi Pythonia voidaan käyttää useilla muilla kielillä ohjelmoitujen isäntäjärjestelmien jatkamiseen käyttämällä esimerkiksi Jython-toteutusta Java Virtual Machine -yhteensopivien kielien kanssa tai IronPython-toteutusta .NET-yhteensopivien kielien kanssa (*Python Programming Language* 2012).

AngelScript on erittäin alustariippumaton. Se toimii useilla eri suorittimilla ja käyttöjärjestelmillä. Lisäksi sille löytyy C-rajapinta, jonka avulla monet muutkin kuin C++-kielellä ohjelmoidut järjestelmät pääsevät AngelScript-kirjastoon käsiksi. (*AngelScript* 2012).

4 SimpScript

SimpScript on imperatiivinen C:n sukuinen skriptikieli staattisella tyypityksellä. Sen pääsääntöinen tarkoitus on toimia yleisenä pelien skriptikielenä, mutta sitä on toki mahdollista käyttää muuhunkin kuin pelimoottorin skriptaamiseen. SimpScript tukee jossain määrin modulaarista ohjelmointia, jossa ohjelma koostetaan useista lähdekooditiedostoista. Se on toteutettu C++-kielellä. Seuraavissa luvuissa käsitellään SimpScriptin syntaksia sekä hieman sen toteutusyksityiskohtia.

4.1 Syntaksi ja semantiikka

Kuten sanottu, SimpScriptin syntaksi pohjautuu C-kieleen. Se on kuitenkin joiltain osin yksinkertaisempi kuin C. SimpScript skriptiohjelman rakenne muisuttaa C:tä ja myös moduuliensa osalta hieman esimerkiksi Javaa. Ohjelman alussa esitellään kyseisen moduulin nimi. Tämä tapahtuu *module*-avainsanalla, kuten Javassa esitellään käytettävä pakettinimi *package*-avainsanalla. Moduulin nimen tulee vastata lähdekooditiedoston nimeä ilman tiedostopäätettä.

Seuraavaksi ohjelmaan tuodaan käytetyt ulkoiset moduulit *import*-lauseilla. Tämäkin vastaa enemmän Javan moduulisysteemiä [1] kuin C:n tekstuaalista sisällytystä. C:ssä siis ei varsinaisesti ole kielen sisäistä modulaarisuuden tukea, vaan esiprosessoinnin aikana eri tiedostoja yhdistellään include-makrojen mukaan. SimpScript taas kykenee tuomaan samaan käännösyksikköön toisten moduulien sisältämiä symboleja kääntämällä ensin tuodut moduulit.

Ulkoisten moduulien tuonnin jälkeen alkaa ohjelman varsinainen koodi, joka sisältää eri määrän funktioita, tyyppimäärittelyjä ja globaaleja muuttujia. Näiden osalta kieli vaikuttaa varsin tarkalta kopiolta C-kielestä.

Ohjelman suorituksen aloituskohtana voi toimia mikä tahansa skriptifunktio. Riippuu isäntäjärjestelmästä, kuinka se skriptejä käyttää ja mistä funktiosta se aloittaa suorituksen. Peleissä skripti voi esimerkiksi suorittaa peliobjektin päivitystä jokaisen logiikkapäivityksen yhteydessä kutsumalla peliobjektiin liitetyn skriptin Update-funktiota.

Sisäänrakennettuja tietotyyppejä SimpScriptissä ovat

1. Tässä ei viitata Javan tulevaan Java Module System -spesifikaatioon vaan nykyiseen pakettipohjaiseen malliin

- etumerkilliset kokonaislukutyypit: char, short, int ja long (1, 2, 4 ja 8 tavua)
- etumerkittömät: uchar, ushort, uint ja ulong (1, 2, 4 ja 8 tavua)
- liukulukutyypit float (4 tavua) ja double (8 tavua).

Näiden tietotyyppien lisäksi on mahdollisuus luoda rakenteisia, koostettuja tietotyyppejä *struct*-avainsanalla. *Union*-avainsanalla voi luoda yhdistetyttyjä tietotyyppejä. Avainsanalla *typealias* on mahdollisuus nimetä uudelleen jokin tyyppi toiseksi. Muista tietotyypeistä on myös mahdollista luoda eriulotteisia staattisen kokoisia taulukoita.

Listauksessa 1 on esimerkki SimpScript ohjelmasta, joka laskee fibonaccin lukuja. Listauksesta käy ilmi SimpScriptin yksinkertainen C-kieltä muistuttava syntaksi.

Listaus 1. SimpScript ohjelma, joka laskee fibonaccin lukuja

```
1  module fibo;
2
3  import io;
4
5  uint fibo(uint x)
6  {
7      if (x == 0) return 0;
8      if (x == 1) return 1;
9      return fibo(x−1) + fibo(x−2);
10 }
11
12 int main()
13 {
14     uint n = 40;
15     uint fn = fibo(n);
16     io::print_fmt2("fibo(%u) == %u", &n, &fn);
17
18     return 0;
19 }
```

Listauksesta voi huomata myös, että SimpScript ei tällä hetkellä tue muuttuvaa määrää parametrejä tai funktioiden ylikuormitusta. Tästä syystä formatoitua tulostusta varten on olemassa eri versiot eri määrille parametrejä: io::print_fmt*N*, jossa *N* on parametrien lukumäärä. Kuten C:ssä, parametrin tyyppi päätellään tyyppiturvattomasti parametrina annetusta formaattimerkkijonosta.

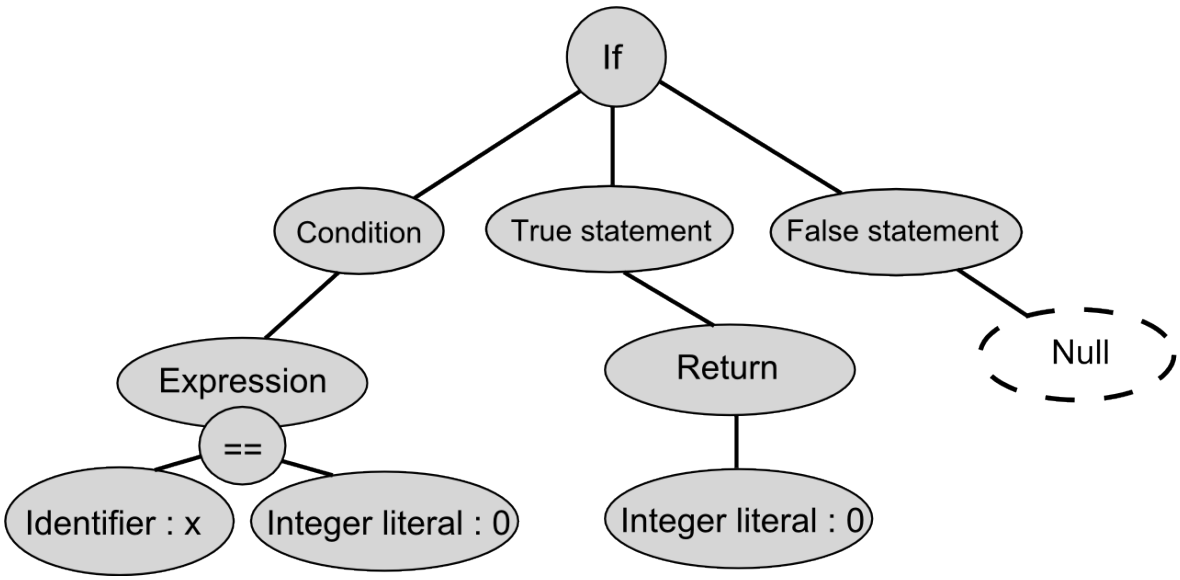
4.2 Kääntäjä

SimpScriptin kääntäjä koostuu viidestä eri moduulista: Tokenizer, Parser, SemanticCheck, ILGenerator ja ByteCodeGenerator. Tämä rakenne vastaa hyvin luvussa 2.2 esitettyä kääntäjän rakennetta.

Tokenizer toimii leksikaalisena analysoijana ja se irroittaa tekstuaalisesta lähdekoodista SimpScript-kielen tokeneita. Näitä ovat esimerkiksi avainsanat, kuten *module*, *if*, *uint* ja *typealias*, sekä erilaiset operaattorit ja välimerkit.

Parser parsii tokenijonosta abstraktin syntaksipuun – eli toimii siis syntaktisena analysoijana. Syntaksipuun solmut koostuvat yhdestä tai useammasta tokenista ja kuvaavat SimpScriptin yksittäisiä lauseita. Kuvio 1 esittää lauseen (*) muodon, kun se on parsittu ja lisätty abstraktiin syntaksipuuhun.

```
if (x == 0) return 0;                                                    (*)
```



Kuvio 1. Osa abstraktia syntaksipuuta.

Kuvion jokainen ellipsi kuvaa yhtä tokenia. Koska kyseisellä *if*-lauseella ei ole *else*-haaraa, on kuvion False statement -solmu tyhjä.

Ohjelman semantiikan tarkistaa SemanticCheck. Koska SimpScript on staattisesti tyypitetty, päättää se myös syntaksipuun eri solmujen tyypityksen.

ILGenerator luo syntaksipuusta välimuotokielisen (engl. *intermediate language*) esityksen. Välimuotokieli muistuttaa jo hyvin paljon lopullista tavukoodia.

Lopulta ByteCodeGenerator laskee lopullisen tavukoodin tulevan koon, asettaa nimettyjen kohteiden (funktiot, globaalit muuttujat) sijainnit ja muuntaa välimuotokieliset käskyt tavukoodiksi.

4.3 Virtuaalikone

SimpScriptin virtuaalikone perustuu rekistereihin. Se sisältää yleisrekisterit A, B, C ja D. Näitä käytetään aritmetiikan suoritukseen ja muistiosoitteiden esittämiseen. Lisäksi SimpScriptissä on erikoisrekisterit: ohjelmalaskuri PC, pino-osoitin SP, kantaosoitin BP ja lippurekisteri F. Ohjelmalaskuri sisältää ohjelmakoodin nykyisen suorituskohdan. Pino-osoitin osoittaa pinon huipun, joka kasvaa alaspäin. Kantaosoitinta käytetään tallentamaan pinokehyksen kanta. Lippurekisteriä käytetään tallentamaan edellisen käskyn aiheuttama tilamuutos, jotta ohjelmassa voidaan suorittaa ehdollista haarautumista esimerkiksi *if*-lauseen yhteydessä. Jokainen rekisteri on 64 bitin kokoinen paitsi lippurekisteri, joka on 8 bittiä.

Virtuaalikoneen sisäinen toimita pohjautuu tavukoodin suoritussilmukkaan. Silmukassa switch-case-rakenteessa käydään läpi kaikki mahdolliset virtuaalikoneen käskyt. Kun ohjelman suoritus aloitetaan, luetaan ohjelmalaskurin osoittamasta kohdasta koodia käsky. Tämä käsky annetaan switch-case -rakenteelle, jossa käskyn vaatimat toimenpiteet suoritetaan. Listaus 2 esittää pseudokoodina silmukan toiminnan.

Listaus 2. Virtuaalikoneen suoritussilmukka pseudokoodina

```
1  while (!exit)
2  {
3      opcode = ReadOpcode();
4      switch (opcode)
5      {
6      case OP_LOAD:
7          Load(); break;
8      case OP_STORE:
9          Store(); break;
10     case OP_ADD:
11         Add(); break;
12     /* ... */
13     default:
14         Error("Invalid opcode!");
15     }
16 }
```

4.4 Liittäminen isäntäjärjestelmään

Tällä hetkellä SimpScript -skriptimoottorin voi liittää vain C++-kieliseen ohjelmaan. SimpScripting liittäminen isäntäjärjestelmään tapahtuu rekisteröimällä moottorin tietoon tarvittavat globaalit funktiot tai luokkien metodit ennen skriptien kääntämistä. Rekisteröinti aiheuttaa sen, että skriptiä kääntäessään SimpScript-kääntäjä löytää reflektiotietokannastaan kyseiset funktiot ja metodit ja pystyy siten käyttämään niitä.

Listaus 3 näyttää esimerkin isäntäjärjestelmän rajapinnan rekisteröinnistä SimpScript-skrip-

timoottorille. Ensimmäinen rivi luo uuden skriptimoottorin *engine*. Metodi *SetMessageCallback* asettaa SimpScriptin virheviestien käsittelykutsun. Tämän avulla SimpScript pystyy kertomaan esimerkiksi käännöksessä tapahtuneista virheistä. Skriptimoottorin *RegisterGlobalFunction*-metodi rekisteröi isäntäjärjestelmän globaalin funktion; ensimmäinen parametri on funktion nimi skriptijärjestelmässä ja toinen on rekisteröitävä funktio. *RegisterType*-template-metodi rekisteröi skriptijärjestelmän tietoon uuden tyypin. Tämä on tarpeen ennen *MyClass::Method*-metodin rekisteröimistä, sillä skriptissä kyseinen metodi ottaa ensimmäiseksi parametrikseen *MyClass*-tyyppisen osoittimen. Tämä toimii siis eksplisiittisenä *this*-osoittimena metodille.

Listaus 3. Esimerkki rajapinnan rekisteröimisestä skriptimoottorille

```
1 simpScript::Engine *engine = new simpScript::Engine();
2 engine->SetMessageCallback(messageCallback);
3
4 if (!engine->RegisterGlobalFunction("print", print))
5         std::cout << "Unable to register print!" << std::endl;
6 if (!engine->RegisterType<MyClass>("MyClass"))
7         std::cout << "Unable to register MyClass!" << std::endl;
8 if (!engine->RegisterGlobalFunction("Method", &MyClass::Method))
9         std::cout << "Unable to register MyClass::Mehod!" << std::endl;
```

Jos funktio tai metodi on ulkoinen, eli isäntäjärjestelmästä rekisteröity, tuottaa kääntäjä sitä kutsuessaan eri käskyn kuin kutsuessaan SimpScriptille natiivia funktiota. Natiivin kutsun käskyn nimi on OP_CALL kun taas ulkoisen on OP_CALL_EXTERN. Näiden toiminta eroaa siinä, että OP_CALL käskyn parametrina on osoite, josta kutsuttava funktio alkaa, kun taas OP_CALL_EXTERN saa parametrikseen kyseisen ulkoisen funktion tunnisteluvun. Tunnisteluvun avulla virtuaalikone pystyy kutsumaan sille rekisteröityä funktiota tai metodia.

5 Vertailu

Eri kielten soveltuvuus peliohjelmointiin riippuu monista asioista. Tässä luvussa verrataan eri skriptausratkaisujen soveltuvuutta pelikäyttöön. Verratailtavia kieliä ovat Lua, Python, UnrealScript, TorqueScript, AngelScript ja SimpScript.

5.1 Arvioitu tuottavuus

Seuraavaksi arvioidaan kuinka hyvin eri skriptikielet parantavat tuottavuutta peliohjelmoinnissa. Tuottavuuden arviointi on vaikeaa, sillä on selvää, että eri henkilöt ovat tuottavampia käyttäessään eri työkaluja kuin toiset. Tuottavuuteen vaikuttavat kielen syntaktisen helppouden lisäksi kielelle saatavissa olevien työkalujen, kuten profilointi- ja debuggaustyökalujen, laatu ja soveltuvuus.

Kuten aiemmin jo luvussa 3.2 on todettu, vertailuun osallistuvista kielistä dynaamisia ovat Lua, Python ja TorqueScript. Nämä kielet mahdollistavat nopean kehityssyklin antamalla kehittäjille vapauden kirjoittaa skriptejä ilman, että heidän täytyy keskittyä tyypitystä. Tämän lisäksi ainakin Lua ja Python mahdollistavat myös keskeneräisen ohjelman ajamisen niiltä osin kuin se jo toimii.

Haittana näissä kielissä voi olla, että kehittäjät ovat tottuneet vahvaan tyypitykseen, eivätkä omaa aiempaa kokemusta dynaamisista kielistä. Totuttautuminen Luan ja Pythonin ohjelmointikäytänteisiin voi viedä aikaa ja vähentää tuottavuutta. Tämä on kuitenkin todennäköisesti vain väliaikainen pudotus tuottavuudessa, ja kun kehittäjät ovat tottuneet kyseessä olevaan kieleen, voi tuottavuus parantua huomattavasti. Toinen ongelma dynaamisissa kielissä voi olla merkilliset ohjelmointivirheet, jotka johtuvat esimerkiksi muuttujanimien kirjoitusvirheistä. Suorituskykyyn dynaaminen tyyppijärjestelmä aiheuttaa yleensä huomattavan kuorman (Ortin et al. 2010), joka voisi aiheuttaa ohjelmoijalle lisätyötä logiikan optimoinnissa. Mutta, koska dynaamisia kieliä kuitenkin käytetään pelien skriptaamiseen, ei tämä liene liian suuri haitta.

Loput kielistä – UnrealScript, AngelScript ja SimpScript – ovat staattisia. Staattisuus saattaa vähentää kehittäjien tuottavuutta, mutta vähentää samalla tyypitykseen liittyviä ohjelmointivirheitä (Ortin et al. 2010). Jos tyyppivirheet vältetään jo käännösaikana, säästyy ohjelmoijan aikaa, kun hänen ei tarvitse erikseen kitkeä tyyppivirheitä koodista. Staattisuudesta on myös etua optimointivaiheessa, sillä staattinen tyyppitieto auttaa kääntäjää generoimaan nopeam-

paa tavukoodia (Ortin ja Garcia 2010). Toisaalta myös koodin selkeyteen saattaa vaikuttaa staattisten kielien sisältämä tieto eri muuttujien ja funktioiden tai metodien tietotyypeistä.

Syntaksin osalta vertailu vaikeutuu, sillä on hyvin subjektiivista, kuinka helpoksi eri henkilöt syntaksin kokevat. Pythonin syntaksi sisältää hyvin vähän välimerkkejä ja on siis nopeaa kirjoittaa. Toisaalta joillekin Pythonin sisennysriippuvainen koodialueiden rajaus ei tunnu luontevalta. Luan syntaksi on tyhjämerkki-riippumaton ja siinä koodialueen alku ja loppu merkitään avainsanoin (esimerkiksi *function ... end*). Tämä lisää kirjoittamista, mutta saattaa helpottaa koodin lukemista. Samoin C- tai C++-tyyppisen syntaksin omaavat kielet, UnrealScript, TorqueScript, AngelScript ja SimpScript, ovat tyhjämerkki-riippumattomia. Näissä koodialueet rajataan aaltosuluilla avainsanojen sijaan.

Kaikki muut kielistä, paitsi SimpScript, helpottavat ohjelmoijan työtä automaattisella muistinhallinnalla. Vaikka automaattinen muistinhallinta ei suoraan poista muistiin liittyviä ongelmia, nopeuttaa se pelin tai muun ohjelman kehitystä huomattavasti. Toisaalta automaattinen muistinhallinta voi myös aiheuttaa ongelmia esimerkiksi muistirajoitteisessa ympäristössä. Tällöin, jos muistinhallintaa ei pystytä kontrolloimaan tai ohittamaan, saattaavat muistiongelmat vaikuttaa negatiivisesti tuottavuuteen.

Lua, UnrealScript ja AngelScript tarjoavat debuggerirajapinnan, jonka avulla kehittäjä voi itse toteuttaa debuggerin kielelle. (*Lua 5.2 Reference Manual; UnrealScript Language Reference; AngelScript* 2012). Python taas sisältää oman debuggerimoduulin (*The Python Debugger*) ja jotkin TorqueScriptiä tukevat kehitysympäristöt mahdollistavat kielen debuggauksen (*GarageGames*). SimpScript ei vielä sisällä debuggausmahdollisuutta ja on siis tältä osalta muita kieliä heikompi.

Erityisesti peliohjelmointiin suunnitellut kielet, UnrealScript ja TorqueScript, sisältävät valmiiksi peliohjelmointia helpottavia ominaisuuksia. UnrealScript tarjoaa kieleen valmiiksi tarjoamalla ajastin- ja tilapohjaisen ohjelmointimallin, latentit funktiot sekä helpottaa verkkoohjelmointia valmiilla rajapinnalla (*UnrealScript Language Reference*). TorqueScript tuo kieleen suoraan paljon ominaisuuksia pelimoottorin puolelta. Näistä esimerkkinä tapahtumalaukaisimet (engl. *trigger*), jotka mahdollistavat tietyn pelimaailman alueen tuottavan tapahtumia, kun peliobjekti tulee sen sisään tai lähtee siitä. Näihin tapahtumiin on mahdollista vastata skriptijärjestelmän puolelta. (*GarageGames*). SimpScript ei sisällä tällä hetkellä mitään peliohjelmointikeskeisiä ominaisuuksia, vaikka onkin erityisesti pelien skriptaamiseen suunnattu.

5.2 Suoritusympäristö

Koska pelejä tehdään useille eri alustoille, kuten tietokoneet, pelikonsolit ja mobiililaitteet, on erittäin tärkeää, että skripatusratkaisu ei rajoita pelimoottorin suoritusalustavalintoja. Yleisesti käyetyt skriptiratkaisut ovatkin suurelta osin alustariippumattomia.

Python, Lua ja AngelScript ovat kaikki portattavia yleisäteviä skriptausratkaisuja. Niitä on siis mahdollisuus käyttää useammalla kuin yhdellä suoritusalustalla ja liittää eri isäntäjärjestelmiin. Tämä linee yksi niistä syistä, miksi näitä kieliä käytetään paljon peliohjelmoinnissa.

Python toimii ainakin yleisimmillä käyttöjärjestelmillä niin PC:llä kuin Mac:llä. Lisäksi siitä löytyy käännöksiä muutamille muille alustoille, joista ehkä mielenkiintoisimpana esimerkkinä pelejä ajatellen Sony PSP -käsikonsoli. (*Python Programming Language* 2012).

Kuten aiemmin luvussa 3.4 mainittiin, Lua on ohjelmoitu C:llä ja sen väitetään kääntyvän kaikilla alustoilla joille C-kääntäjä löytyy (*The Programming Language Lua* 2012). Crytenkin CryEngine 3 käyttää skriptaukseen Luaa (*The CryENGINE Scripting Manual*) ja pelimoottori toimii ainakin Windows PC:llä sekä Playstation 3 ja Xbox 360 -pelikonsoleilla (*CryEngine 3*). Tämä toimii todisteena siitä, että Lua todella toimii useammassa suoritusympäristössä.

AngelScript on erittäin alustariippumaton ja kirjoitettu C++-kielellä. Se toimii useilla eri suorittimilla ja käyttöjärjestelmillä. AngelScript-toteutus tarjoaa myös natiivit kutsukonventiot monelle yleisesti käyetylle prosessoriarkkitehtuurille – Playstation 3 ja Xbox 360 mukaan lukien – saavuttakseen nopean suorituskyvyn. (*AngelScript* 2012)

Unreal Script on osa Unreal Engineä, jolla pystyy kehittämään pelejä monelle eri alustalle (*Unreal Engine Game Platforms*). Ainut rajoitus on nimenomaa Unreal Engine. Siinä missä yleiset skriptikielet voi liittää lähes mihin tahansa pelimoottoriin, UnrealScript on osa Unreal Engine -pelimoottoria, ja siis rajoittunut sen tukemiin alustoihin. Samoin TorqueScript on rajoittunut vain Torque Enginen skriptaamiseen ja sen tukemiin alustoihin. Näitä alustoja ovat vain Windows sekä web-selaimet (*GarageGames*).

SimpScript ei tällä hetkellä tue muita kuin Windows-ympäristöä ja C++-kielistä isäntäjärjestelmää. Se on siis tältä osin selkeästi paljon rajoittuneempi kuin muut vertailun skriptausratkaisut.

5.3 Suorituskyky

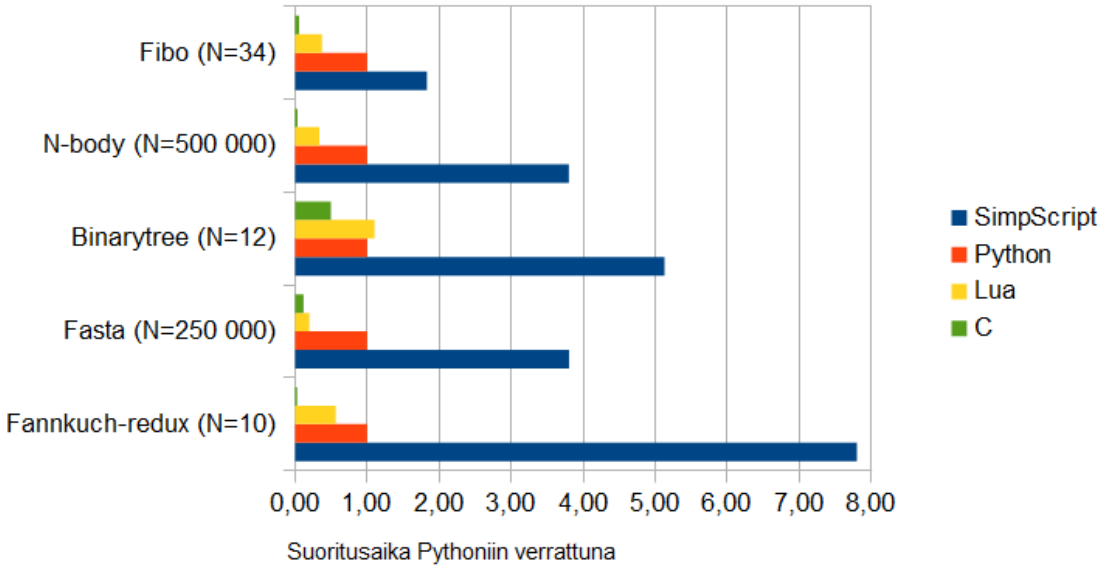
Suorituskykyä vertailtiin skriptikielien Lua[1], Python[2] ja SimpScript osalta. Lisäksi käytettiin C-kielistä toteutusta[3], jotta voitaisiin yleisesti verrata skriptikielien ja koneelle natiivisti käännetyn kielen tehokkuutta. Vertailu suoritettiin ajamalla eri testiohjelmia testattaville kielille käännettynä. Testiohjelmiksi valikoituivat *The Computer Language Benchmarks Game* 2012 -sivuston *binarytree, fannkuch-redux, fasta* sekä *n-body* -ohjelmat. Näiden testiohjelmien toteutukset kielille Lua, Python ja C on ladattu 11.–12.11.2012 välisenä aikana kyseiseltä sivustolta. SimpScript-versiot testiohjelmista on käännetty C-kielisistä toteutuksista. Lisäksi yhtenä testiohjelmana käytettiin fibonaccin lukujen laskemisen rekursiivista toteutusta. Testilaitteistona toimi Intel Core i7-3610QM prosessorilla ja 8 GB muistilla varustettu tietokone.

Jokainen testiohjelma ottaa syötteekseen kokonaisluvun N, joka esittää ongelman vaativuutta. Testit suoritettiin jokaisen ohjelman kohdalla kolmella eri N:n arvolla. Tällöin saatiin mitattua jokaisen testin aikavaativuuden kasvua N:n suhteen. Jotta satunnainen virhe suoritusajassa saatiin minimoitua, suoritettiin sama testi viisikertaa ja lopullisena tuloksena käytettiin näiden keskiarvoa. Kuviossa 2 on esitetty testiohjelmien suoritusaika eri skriptikielillä ja C:llä suhteutettuna Pythonin suoritusaikaan.

Vaikka SimpScriptin toteutuksessa ei vielä tässä vaiheessa ollut pääsääntöisenä tarkoituksena optimointi, suoritettiin virtuaalikoneelle kuitenkin muutamia yksinkertaisia optimointeja. Esimerkkeinä suoritetuista optimoinneista ovat paljon kutsuttujen funktioiden muuttaminen inline-funktioiksi ja erilaiset virtuaalikoneen rekisterien käytön optimoinnit. Näillä muutoksilla saatiin SimpScriptin suoritustehokkuutta yli kaksinkertaistettua. Kuviossa 3 on esitetty testiohjelmakohtaisesti optimoidun version suoritusaika alkuperäiseen verrattuna. Kuviossa 2 esitetyt SimpScriptin tulokset on mitattu optimoidulla versiolla.

On mainittava, että SimpScriptin kääntäjä ei suorita minkäänlaisia optimointeja generoimalleen tavukoodille. On todennäköistä, että kielen nopeus kasvaisi huomattavasti, jos sen tavukoodi olisi optimoitu. Generoidun symbolisen tavukoodin tarkastelusta havaittiin, että se sisältää paljon toistoa, joka voitaisiin poistaa sopivilla opimoinneilla. Tavukoodi saattaisi kutistua arviolta noin puoleen nykyisestä koostaan. Jos oletetaan, että suoritusaika olisi suoraan verrannollinen tavukoodin määrään nähden, olisi tällöin saatu suoritusnopeus noin kaksinkertainen alkuperäiseen nähden. Kielen tehokkuutta olisi siis mahdollista saada vielä paran-

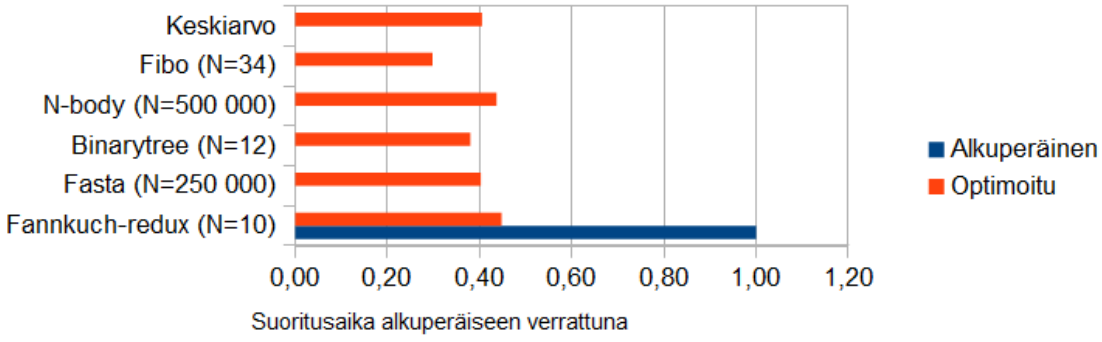
1. Lua for Windows, versio 5.1.4
2. CPython toteutus, versio 3.3.0
3. MinGW GCC -kääntäjä, versio 4.7.1



Kuvio 2. Suoritusaikavertailu, jossa ajat on normitettu Pythonin tuloksen mukaan.

nettua, mutta tarkka nopeusero nykyiseen verrattuna voidaan saada selville vasta optimoitua versiota testaamalla.

Vaikka SimpScriptin suorituskyky on testeissä keskimäärin noin kuusi kertaa hitaampi kuin Pyhtonin, ei tämä kuitenkaan tarkoita, etteikö se sopisi peliohjelmointiin. Kyseessä olevat testit ovat nimenomaa rasitustestejä, jotka mittaavat kielen tehokkuutta äärimmäistä suorituskykyä vaativissa tilanteissa. Ne eivät siis välttämättä ole sopivin mittari kielen pelien skriptaamiseen soveltuvaksi tai soveltumattomaski luokitteluun. Koska peliskriptit nimeno-



Kuvio 3. Suoritusaikavertailu alkuperäisen ja optimoidun SimpScriptin välillä.

maan suorittavat vain korkean tason pelilogiikkaa, eivätkä raskaita algoritmeja.

Testit siis antavat enemmän suuntaa sille, ettei SimpScriptin suoritusteho ole välttämättä ongelma sen pelikäyttöön. Todennäköisesti SimpScriptin suorituskyky on tarpeeksi hyvä reaaliaikaisen pelin korkean tason logiikan ohjaamiseen. Toisaalta, jos esimerkiksi pelien vaatimat useat ulkoisten funktioiden kutsut olisivat erityisen hitaita SimpScriptissä, voisi sen soveltuvuus reaaliaikaiseen suoritukseen heiketä huomattavasti.

6 Johtopäätökset

Tässä tutkimuksessa esitettiin pelien skriptaamiseen soveltuva ohjelmointikieli SimpScript. Vaikka SimpScript on vielä varhaisessa vaiheessa, on se kuitenkin toimiva skriptikielenä – se totetuttaa ainakin vähimmäisvaatimukset pelien skriptamiseen: skriptien liittäminen pelimoottoriin ja suorittaminen ilman pelin uudelleen kääntämistä. Vertailussa kuitenkin huomattiin, että monet ominaisuudet, jotka muista peleissä käytetyistä skriptausratkaisuista löytyvät, puuttuvat vielä SimpScriptistä.

SimpScript on staattisesti tyypitetty. Skriptikielet mielletään usein dynaamisiksi, koska dynaamisuus yleensä nostaa ohjelmoijan tuottavuutta. Huomattiin kuitenkin, että peleissä käytettyjen skriptikielien tyypityksellä ei näyttäisi olevan väliä, sillä näistä yleisimpien joukosta löytyy molempia, niin staattisia kuin dynaamisia kieliä. Kielen staattisuus ei siis ole ongelma, ja SimpScript on tältä osin pätevä pelikäyttöön.

Jos kieltä käytetään prototyyppien kehittämiseen, tällöin dynaamisesti tyypitetty kieli voi olla järkevä vaihtoehto. Staattisella tyypityksellä voidaan välttää ajonaikaisia tyyppivirheitä ja sopii siis paremmin jo kiinnitettyyn koodiin. Optimaalista voisi siis olla, että skriptikieli tukisi sekä dynaamista että staattista tyypitystä. Tällöin peliprojektin alun testausvaiheessa dynaamiset ominaisuudet lisäisivät tuottavuutta, ja kun koodi on jo valmis tuotantoon, voidaan se muuttaa käyttämään staattista tyypitystä tyyppivirheiden poistamiseksi ja optimaalisemman suorituskyvyn saavuttamiseksi. Tälläista tukee esimerkiksi skriptikieli Konoha (Kuramitsu 2010). Lisäksi dynaamisen ja staattisen tyypityksen yhdistämistä on jo tutkittu jonkinverran (Ortin et al. 2010; Ortin ja Garcia 2010; Wrigstad et al. 2010). On siis harkinnan arvoista pyrkiä tuomaan dynaamisen tyypityksen etuja SimpScript-kieleen.

Kaikki muut vertailtavista kielistä sisältivät automaattisen muistinhallinnan paitsi SimpScript. On selkeää, että korkean tason logiikkaa ohjelmoitaessa tuottavuutta nostaa, jos ohjelmoijan ei tarvitse keskittyä muistinhallintaan, vaan se tapahtuu automaattisesti. SimpScriptin jakokehityksessä tämä on siis tarpeen toteuttaa.

Suoritusympäristöjen ja -alustojen osalta eri ratkaisut olivat yleisesti hyvinkin riippumattomia. Yleiset skriptikielet toimivat useimmilla pelialustoilla, toiset paremmin toiset hieman huonommin. Samoin pelimoottorikohtaiset kielet olivat portattavia johtuen niiden pelimoottorin toimivuudesta useilla alustoilla. TorqueScriptin alustavaihtoehdot tosin olivat hieman rajoittuneemmat. Ainoana ei-siirrettävänä kielenä on SimpScript. Tältä osalta on SimpScriptin siis jatkossa parannettava, jos se pyrkii toimimaan yleisenä pelien skriptikielenä.

Yleisesti voidaan todeta, että SimpScript ei nykyisellään ole kilpailukykyinen muihin skriptiratkaisuihin verrattuna. Jos kuitenkin halutaan helpottaa pelimoottorin käyttöä esimerkiksi tuomalla pelimoottorin keskeisiä ominaisuuksia esille kielen syntaksiin, on pelimoottorikohtaisen kielen kehittäminen lähes ainut vaihtoehto. Tällöin SimpScript toimii esimerkkinä siitä, että yksinkertaisen kielen toteuttaminen onnistuu pienellä vaivalla, eikä vaadi järjettömiä resursseja. Kuitenkaan tätä vaihtoehtoa ei voi suositella pienemmille pelistudioille tai indiekehittäjille. Heidän kannattaa allokoida aikansa varsinaiseen peliin ja käyttää jotain valmista skriptiratkaisua tai pelimoottoria, joka sellaisen tarjoaa.

Lähteet

Anderson, E. F. 2011. “A Classification of Scripting Systems for Entertainment and Serious Computer Games”. Teoksessa *Games and Virtual Worlds for Serious Applications (VS-GAMES), 2011 Third International Conference on,* 47–54. doi:10.1109/VS-GAMES.2011.13.

AngelScript. 2012. Haettu 25. lokakuuta 2012. http://www.angelcode.com/angelscript/.

Blender. Haettu 20. marraskuuta 2012. http://www.blender.org/development/.

Bloom, Bard, John Field, Nathaniel Nystrom, Johan Östlund, Gregor Richards, Rok Strniša, Jan Vitek ja Tobias Wrigstad. 2009. “Thorn: robust, concurrent, extensible scripting on the JVM”. Teoksessa *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications,* 117–136. OOPSLA ’09. Orlando, Florida, USA: ACM. ISBN: 978-1-60558-766-0. doi:10.1145/1640089.1640098. http://doi.acm.org/10.1145/1640089.1640098.

Boost.Python. Haettu 19. joulukuuta 2012. http://www.boost.org/doc/libs/1_46_1/libs/python/doc/.

CryEngine 3. Haettu 15. joulukuuta 2012. http://www.crytek.com/cryengine/cryengine3/overview.

Diehl, Stephan, Pieter Hartel ja Peter Sestoft. 2000. “Abstract machines for programming language implementation”. *Future Generation Computer Systems* 16 (7): 739–751. ISSN: 0167-739X. doi:10.1016/S0167-739X(99)00088-6. http://www.sciencedirect.com/science/article/pii/S0167739X99000886.

Flow Graph Editor. Haettu 19. joulukuuta 2012. http://freesdk.crydev.net/display/SDKDOC2/Flow+Graph+Editor.

Frictional Games Wiki. Haettu 20. marraskuuta 2012. http://wiki.frictionalgames.com/hpl1/start.

GarageGames. Haettu 20. marraskuuta 2012. http://www.garagegames.com/products/torque-3d/documentation.

Ierusalimschy, Roberto, Luiz Henrique de Figueiredo ja Waldemar Celes. 2007. “The evolution of Lua”. Teoksessa *Proceedings of the third ACM SIGPLAN conference on History of programming languages,* HOPL III. San Diego, California: ACM. ISBN: 978-1-59593-766-7. doi:10.1145/1238844.1238846. http://doi.acm.org/10.1145/1238844.1238846.

Kuramitsu, Kimio. 2010. “Konoha: implementing a static scripting language with dynamic behaviors”. Teoksessa *Workshop on Self-sustaining Systems (S3) 2010,* 21–29. doi:10.1145/1942793.1942797.

Lua 5.2 Reference Manual. Haettu 20. joulukuuta 2012. http://www.lua.org/manual/5.2/manual.html.

Luabind. Haettu 16. joulukuuta 2012. http://www.rasterbar.com/products/luabind.html.

Mogensen, Torben Ægidius. 2010. *Basics of Compiler Design.* http://www.diku.dk/~torbenm/Basics/.

Ortin, F., ja M. Garcia. 2010. “Supporting dynamic and static typing by means of union and intersection types”. Teoksessa *Progress in Informatics and Computing (PIC), 2010 IEEE International Conference on,* 993–999. Volyymi 2. doi:10.1109/PIC.2010.5687860.

Ortin, F., D. Zapico, J.B.G. Perez-Schofield ja M. Garcia. 2010. “Including both static and dynamic typing in the same programming language”. *Software, IET* 4, numero 4 (): 268–282. ISSN: 1751-8806. doi:10.1049/iet-sen.2009.0070.

Ousterhout, J. K. 1998. “Scripting: higher level programming for the 21st Century”. *Computer* 31, numero 3 (): 23–30. ISSN: 0018-9162. doi:10.1109/2.660187.

Python Programming Language. 2012. Haettu 25. lokakuuta 2012. http://www.python.org/.

Schultz, Warren. 2012. *Unreal Engine 4 - First Look.* Haettu 11. joulukuuta 2012. http://gameindustry.about.com/od/trends/a/Unreal-Engine-4-First-Look.htm.

The Computer Language Benchmarks Game. 2012. Haettu 25. lokakuuta 2012. http://shootout.alioth.debian.org/.

The CryENGINE Scripting Manual. Haettu 15. joulukuuta 2012. http://freesdk.crydev.net/display/SDKDOC5/Home.

The Programming Language Lua. 2012. Haettu 25. lokakuuta 2012. http://www.lua.org/home.html.

The Python Debugger. Haettu 20. joulukuuta 2012. http://docs.python.org/3.3/library/pdb.html.

Unity scripting. Haettu 20. marraskuuta 2012. http://unity3d.com/unity/workflow/scripting.

Unreal Engine Game Platforms. Haettu 15. joulukuuta 2012. http://www.unrealengine.com/en/platforms/.

UnrealScript. Haettu 20. marraskuuta 2012. http://udn.epicgames.com/Three/UnrealScriptHome.html.

UnrealScript Language Reference. Haettu 11. joulukuuta 2012. http://udn.epicgames.com/Three/UnrealScriptReference.html.

Wikipedia: "Scripting Language". Haettu 13. marraskuuta 2012. http://en.wikipedia.org/wiki/Scripting_language.

Wikipedia: "Category:Lua-scripted video games". Haettu 20. marraskuuta 2012. http://en.wikipedia.org/wiki/Category:Lua-scripted_video_games.

Wolfire Games. Haettu 20. marraskuuta 2012. http://blog.wolfire.com/2010/01/Choosing-a-scripting-language.

Wrigstad, Tobias, Francesco Zappa Nardelli, Sylvain Lebresne, Johan Östlund ja Jan Vitek. 2010. “Integrating typed and untyped code in a scripting language”. *SIGPLAN Not.* (New York, NY, USA) 45, numero 1 (): 377–388. ISSN: 0362-1340. doi:10.1145/1707801.1706343. http://doi.acm.org/10.1145/1707801.1706343.

Liitteet

A SimpScript bintree toteutus

Listauksessa 4 on esitetty *The Computer Language Benchmarks Game* sivuston bintree testiohjelman toteutus SimpScript kielellä. Ohjelmakoodi on käännetty Kevin Carsonin C-kielisestä versiosta.

Listaus 4. SimpScript versio bintree-testiohjelmasta

```
/*
    The Computer Language Shootout Benchmarks
    http://shootout.alioth.debian.org/

    bintree benchmark:
    Allocate and deallocate many many binary trees

    transtlated from Kevin Carson's C version
    by Henrik Paananen
*/

module bintree;

import io;

extern int pow(int b, int p);

extern void* malloc(ulong size);
extern void free(void *ptr);

struct node
{
    int item;
    node *left;
    node *right;
};

node* new_node(node *l, node *r, int i)
{
    node *ret;

    ret = malloc(sizeof(node));
    ret->left = l;
    ret->right = r;
    ret->item = i;

    return ret;
}

int node_check(node *n)
{
    if (n->left)
    {
        int lc = node_check(n->left);
        int rc = node_check(n->right);
```

```
46              return lc + n->item - rc;
47          }
48          return n->item;
49      }
50
51      node* bottom_up_tree(int item, int depth)
52      {
53          if (depth > 0)
54              return new_node(bottom_up_tree(2 * item - 1, depth - 1),
55                  bottom_up_tree(2 * item, depth - 1), item);
56          else
57              return new_node(0, 0, item);
58      }
59
60      void delete_tree(node *tree)
61      {
62          if (tree->left != 0) {
63              delete_tree(tree->left);
64              delete_tree(tree->right);
65          }
66          free(tree);
67      }
68
69      int main(int N)
70      {
71          uint depth;
72          uint minDepth;
73          uint maxDepth;
74          uint stretchDepth;
75
76          node *stretchTree;
77          node *longLivedTree;
78          node *tempTree;
79
80          minDepth = 4;
81
82          if ((minDepth + 2) > (uint)N)
83              maxDepth = minDepth + 2;
84          else
85              maxDepth = N;
86
87          stretchDepth = maxDepth + 1;
88
89          stretchTree = bottom_up_tree(0, stretchDepth);
90
91          int check = node_check(stretchTree);
92          io::print_fmt2("stretch tree of depth %u\t check: %i\n", &stretchDepth, &check);
93
94          delete_tree(stretchTree);
95
96          longLivedTree = bottom_up_tree(0, maxDepth);
97
98          for (depth = minDepth; depth <= maxDepth; depth += 2)
99          {
100             int i;
101             int iterations;
102             int check;
103
104             iterations = pow(2, maxDepth - depth + minDepth);
```

```
105
106         check = 0;
107
108         for (i = 1; i <= iterations; i++)
109         {
110             tempTree = bottom_up_tree(i, depth);
111             check += node_check(tempTree);
112             delete_tree(tempTree);
113
114             tempTree = bottom_up_tree(−i, depth);
115             check += node_check(tempTree);
116             delete_tree(tempTree);
117         }
118
119         int it2 = iterations ∗ 2;
120         io::print_fmt3("%i\t trees of depth %u\t check: %i\n", &it2, &depth, &check);
121     }
122
123     check = node_check(longLivedTree);
124     io::print_fmt2("long lived tree of depth %u\t check: %i\n", &maxDepth, &check);
125
126     return 0;
127 }
```