Tommi Teistelä

Skriptikielet verkkopeleissä

Tietotekniikan pro gradu -tutkielma

2. joulukuuta 2015

Jyväskylän yliopisto

Tietotekniikan laitos

Tekijä: Tommi Teistelä

Yhteystiedot: tommi.teistela@jyu.fi

Ohjaaja: Raino Mäkinen, Antti-Jussi Lakanen

Työn nimi: Skriptikielet verkkopeleissä

Title in English: Scripting languages in multiplayer games

Työ: Pro gradu -tutkielma

Suuntautumisvaihtoehto: Ohjelmistotekniikka

Sivumäärä: 76+8

Tiivistelmä: Tutkielmassa perehdytään verkkopelien toteutustekniikoihin ja usealla kielellä toteutettuihin ohjelmistoihin. Konstruktiivisessa osassa näiden avulla rakennetaan yksinkertainen ohjelmistokehys nopeaan verkkopelikehitykseen. Kehyksen toimivuutta arvioidaan eri menetelmillä.

Avainsanat: verkkopelit, skriptikielet, sisällönluonti

Abstract: This thesis studies the implementation of online games and multi-language software development. A simple rapid application development framework for online games is developed based on these studies. The framework's performance is analyzed using various methods.

Keywords: network games, scripting languages, content creation

Termiluettelo

Bullet	Avoimen lähdekoodin fysiikkamoottori, eli ohjelmakirjasto fysiikan (klassisen mekaniikan) simulointiin.
C++11	C++-kielen standardi vuodelta 2011.
Lua	Monissa peleissä käytetty yksinkertainen juontokieli.
Replikaatio	Tietokannan tai vastaavan hajautetun järjestelmän tilan synkronointi.
Serialisointi	Rakenteellisen tiedon muuntaminen muotoon, jossa se voidaan tallentaa tai lähettää.
Skripti- eli juontokieli	Ohjelmointikieli, jonka tarkoitus on mukauttaa, laajentaa ja automatisoida olemassaolevaa järjestelmää.
Verkkokoodi	Pelin tai vastaavan ohjelman osa, jonka tehtävänä on viestittää ohjelman tilaa verkon ylitse.

Kuviot

Taulukot

Sisältö

1 Johdanto

Tässä tutkielmassa perehdytään verkkopelien yleisiin toimintaperiaatteisiin, peleihin upotettuihin juonto- eli skriptikieliin ja tutkitaan kielten yhteistoimintaa pelin verkkototeutuksen, eli sen "verkkokoodin" kanssa. Ensimmäisissä luvuissa tarkastellaan pelien verkkototeutusten toimintaperiaatteita ja upotettuja juontokieliä ohjelmistotekniikan näkökulmasta. Työn konstruktiivisena osana taustatutkimuksessa löydettyjä ratkaisuja sovelletaan verkkototeutusta vailla olevaan pelikehykseen. Toteutuksen onnistumista mitataan kehittämällä alustalle pieniä verkkopelejä ja tutkimalla niiden toimivuutta käytännössä.

1.1 Työn taustaa

Internet-yhteyksien lähes täydellinen saatavuus ja pelialan nousu Hollywood-elokuvien tasoiseksi mediateollisuudeksi on nostanut myös verkkopelaamisen ennennäkemättömään suosioon. Tästä huolimatta pelien verkkototeutusten laatu on yleinen kritiikin kohde peliarvosteluissa ja peliyhteisöjen keskustelusivustoilla. Vaikka moninpeliin keskittyvän tuotteen kehitykseen ja markkinointiin olisi käytetty kymmeniä miljoonia dollareita, julkaisun jälkeen sen verkkototeutus saatetaan havaita puutteelliseksi (DICE 2015). Yhteys palvelimeen katkeilee hyvissäkin verkko-olosuhteissa, pelin kulku ei näytä samalta eri pelaajien näyttöruuduilla ja pelimekaniikka toimii (tahattomasti) eri tavalla yksin- ja moninpelitilanteissa. Kun julkaisuajan suorituskykyongelmat vähitellen häviävät ja näkyvimmät puutteet on saatu korjattua, huijarit ovat jo löytäneet pelin verkkototeutuksen heikkoudet ja pystyvät vapaasti rikkomaan pelin sääntöjä. Heikkoa verkkototeutusta on vaikea hyväksyä julkaisussa, jonka mahdollisuudet suosioon ovat lähes pelkästään sen moninpelissä.

Kirjoittajan henkilökohtaisena ohjelmointiharrastuksena alkanut grafiikka- ja pelirunkojen tutkiminen eteni vähitellen tilanteeseen, jossa suurin tuntematon tekijä oli verkkopeli. Selvästi kalliskin peliprojekti voi sortua loppukirissään kömpelöihin ratkaisuihin verkkopelinsä toteutuksessa. On kuitenkin olemassa myös hyvin suun-

niteltuja ja dokumentoituja ratkaisuja, jotka voivat toimia pienellä vaivalla monenlaisissa peleissä. Löytyisikö näistä riittävästi tietoa, että omaan pelikehykseen olisi mahdollista kehittää helposti laajennettava ja riittävän suorituskykyinen verkkototeutus? Tämä kysymys johti tutkimukseen pelien verkkototeutuksista, joka lopulta kehittyi tämän opinnäytetyön pohjaksi.

1.2 Tutkimusongelma

Työn keskipisteenä on suurimmaksi osaksi C++-kielellä toteutettu pelikehys, Kuu. Sen kehitys alkoi kirjoittajan harrastusprojektista, jonka tavoitteena oli kehittää nopea OpenGL-pohjainen grafiikkarunko, mutta lisäämällä valikoituja osia muista onnistuneista projekteista ja opinnäytteistä siitä oli mahdollista koota yksinkertainen, mutta toimiva ohjelmistokehys pieniin pelikokeiluihin. Kehityksessä pyrittiin alusta alkaen välttämään liiallista riippuvuutta tietystä alustasta ja suosimaan C++11:n entistä kattavampaa vakiokirjastoa sekä useille alustoille saatavilla olevia SDL2:n kaltaisia työkalukirjastoja. Tämän ansiosta projekti on käytettävissä ainakin Windows- ja Linux-pohjaisilla käyttöjärjestelmillä ja todennäköisesti monilla muillakin alustoilla.

Työn alussa runkoon sisältyi alkeellinen tuki tapahtumankäsittelyn mukauttamiselle Lua-skriptikielellä. Suunnitellessa verkkopelitoiminnallisuutta heräsi kysymys tällaisen upotetun kielen roolista verkkopelissä. Joitakin ohjelmointikieliä on suunniteltu jo lähtökohdistaan verkkopelien toteuttamiseen: *Unreal*-pelimoottorissa käytetty *UnrealScript*-kieli antaa kehittäjän kuvailla pelin skriptiosiossa, mitä olion tilan muutoksia tai metodikutsuja replikoidaan verkkopelissä ja miten. Tällaisella kielellä kehitetty pelimekaniikka voi toimia hyvin vaivattomasti verkkopelissäkin. Erikoistuneet kielet ovat kuitenkin työläitä toteuttaa ja vaativat alkuperäiseen suoritusympäristöönsä sidottuina ylimääräistä opettelua kehittäjiltä.

Lua-kielen luonteen ja aiemman web-kehityskokemuksen takia kehyksen skriptirajapinta päätyi muistuttamaan suureksi osaksi monia web-kehyksiä, erityisesti nk. *microframeworkeja*, jotka pystyvät esittämään helppoja rajapintoja monimutkaisille

toimenpiteille ohjelmointikieltensä korkean tason ominaisuuksien ansiosta. Voitaisiinko näitä soveltaa pelikehyksessä? Skriptikielen upotuksessa tehty työ vaikuttaa jo päällekkäiseltä etäkutsujen toteuttamisen kanssa: Metodien nimet tai muut tunnisteet, parametrit, kutsukäytännöt ja monet muut yksityiskohdat on huomioitava sekä kielen rajapinnassa että etäkutsujen toteutuksessa. Ottaen huomioon tällaisen kielen monipuolisuuden, olisiko mahdollista kehittää verkkototeutus, joka lainaa toimintaperiaatteita joissakin kaupallisissa peleissä toimiviksi havaituista ratkaisuista, mutta tarjoaa ne kehittäjälle erikoistuneen kielen tai matalan tason pelilogiikkaan tunkeutuvan verkkoviestityksen sijaan hyödyntäen skriptitulkin ja sen rajapinnan ominaisuuksia?

1.3 Tutkielman rakenne ja tavoitteet

Tutkielma on luonteeltaan pääosin konstruktiivinen, eli tutkimuksen kohteena on sen yhteydessä kehitetty ohjelmistoartifakti: verkkopelikokeiluihin soveltuva pelikehys upotetulla skriptikielellä. Tehtävää lähestytään suunnittelutieteen (March ja Smith 1995) tapaan: aluksi selvitetään taustatiedon ja teorian kautta, miten verkkopelit ja upotetut skriptikielet yleisesti toimivat. Tämän tiedon pohjalta pyritään kehittämään ratkaisu omaan ongelmaan. Toteutuksen onnistumista arvioidaan kehittämällä pelikehyksen päälle yksinkertaisia testipelejä.

Ensimmäisissä luvuissa kerätään taustatietoa verkkopelien ja upotettujen skriptikielten toiminnasta. Luvussa 2 tutkitaan, miten verkkopelit toimivat teoriassa ja tutustutaan muutamiin olemassaoleviin ratkaisuihin. Luku 3 käsittelee Lua-kielen ominaisuuksia ja sen upottamista C++-ohjelmaan. Näiden pohjalta kehitetyn pelikehyksen toimintaa kuvaillaan luvussa 4. Kehyksen toimivuutta tutkitaan luvussa 5 sille kehitettyjen testipelien avulla.

Tutkielman päätavoitteena on löytää tapoja toteuttaa upotetun skriptikielen avulla suorituskykyisiä ja helppokäyttöisiä työkaluja reaaliaikaisten verkkopelien kehittämiseen.

2 Verkkopeleistä

Tämä luku käsittelee verkkopelien taustaa ja yleisiä toimintaperiaatteita. Aluksi tutustutaan lyhyesti pelien historiaan ja määritellään niiden yleisiä käsitteitä. Pelien verkkoratkaisuja analysoidaan hajautettujen järjestelmien käsitteiden pohjalta. Näiden ratkaisujen tunnistetut ominaisuudet ohjaavat myöhemmissä luvuissa kehitettävän verkkopelialustan toteutusta.

Verkkopelit ovat eräänlaisia hajautettuja järjestelmiä, eli kokonaisuuksia, jotka muodostuvat useasta itsenäisestä, mutta keskenään ohjelmallisesti viestivästä yksiköstä (Mullender 1993). *Viestinvälityksellä* jaettu tieto sallii yksiköiden välisen yhteistyön kohti laajempaa tavoitetta. Verkkopelissä jaettu tieto on sen *pelimaailman* tila. Tämä maailma noudattaa tiettyjä sääntöjä, joiden rajoissa pelaajat voivat vaikuttaa pelin etenemiseen tuottamallaan *syötteellä*. Näitä sääntöjä kutsutaan myös *pelimekaniikaksi*. Verkkopelin tapaa ylläpitää jaettua tilaansa, viestittää pelaajien tuottamia syötteitä ja pelimaailmaan aiheutuneita muutoksia kutsutaan yleisesti pelin *verkkokoodiksi*.

Monet kuvaukset verkkopelien toiminnasta luokittelevat ne verkkoarkkitehtuuriensa perusteella kahteen perustyyppiin: asiakas ja palvelin -ja vertaisverkkosovelluksiksi. Koska myös näiden luokkien sisällä esiintyy paljon vaihtelua, tässä on pyritty löytämään luokitteluun toinen akseli replikaatiossa viestitettävän tiedon perusteella. Replikaatiotekniikoita ja verkkoarkkitehtuureja käsitellään osioissa 2.2 ja 2.3.

Mitä tahansa verkkopeliä rajoittaa lopulta käytössä olevan verkkoyhteyden laatu. Monissa hajautetuissa järjestelmissä verkkoviive ei ole toteutuksessa edes huomioitavan arvoinen tekijä: käyttäjä ei odota täysin välitöntä vastausta avatessaan webpohjaisen palvelun seuraavaa sivua, eikä videopalvelussa ole juurikaan väliä viiveellä, jos yhteyden siirtonopeus riittää videon toimittamiseen sen odotetulla toistonopeudella. Reaaliaikaista vastetta tavoittelevissa peleissä verkkoviiveestä voi kuitenkin kehittyä suuri ongelma. Verkkoyhteyksien yleisiin haasteisiin perehdytään tarkemmin osiossa 2.4.

Luvun lopussa osiossa 2.5 tutustutaan vielä tarkemmin muutamaan tunnettuun käytännön ratkaisuun ja pyritään luokittelemaan niitä aiempien havaintojen pohjalta. Näin voidaan arvioida verkkoratkaisujen toimivuutta peligenrekohtaisesti.

2.1 Verkkopelien taustaa

Joitakin nykyisiä verkkopelejä muistuttavia ratkaisuja käytettiin jo 1980-luvulla sotasimulaatioissa (Smed, Kaukoranta ja Hakonen 2002), mutta 1990-luvulle asti useimmat verkkopelit olivat yleensä epäinteraktiivisia "purkkipelejä", joiden tekstipohjainen käyttöliittymä ja vuoropohjainen pelimekaniikka eivät edellyttäneet paljoa pelin palvelimelta, asiakkaalta tai niitä yhdistävältä verkkoyhteydeltäkään. Tyypillisessä reaaliaikaisessa moninpelissä pelaajat kokoontuivat saman pelilaitteen ympärille pelaamaan peliä jaetulla näyttöruudulla. Oman näytön antaminen jokaiselle pelaajalle oli lähinnä pelihalleissa nähty kuriositeetti.

Yksi ensimmäisiä todellisia verkkopelihittejä oli vuonna 1993 ilmestynyt *Doom*, joka tuki useita moninpelimuotoja (nolla)modeemiyhteyksillä ja lähiverkoissa. Pelin suosiosta kertoo se, että vuorokauden sisällä sen ilmestymisestä pelin verkkoliikenne aiheutti jo käyttöhäiriöitä yliopistoverkoissa. Pelin kehittäjillä oli kokemusta vain pienistä, paikallisista lähiverkoista, eikä sen tapaa viestiä koko verkkoa kuormittavalla broadcast-liikenteellä nähty ongelmana (Armitage ym. 2006, s. 13-14). Pelin verkkoliikenne estettiin pian monien organisaatioiden verkoissa, mutta Doom oli jo PC-pelaamisen ja verkkopelien läpimurto.

Internet-yhteydet yleistyivät nopeasti 1990-luvun puolivälissä, mutta vastaavaa läpimurtoa Internet-pelaamisessa ei tapahtunut heti. Vuonna 1996 odotukset olivat korkealla, kun Doomin tekijät julkaisivat uuden *Quake*-räiskintäpelinsä. Sen verkkoviiveelle herkkä moninpeli kuitenkin osoittautui lähes pelikelvottomaksi silloin yleisillä puhelinmodeemia käyttävillä Internet-liittymillä. Ongelma oli havaittu vasta pelin kehityksen loppuvaiheessa, koska sen kehittäjillä oli Doom-pelin menestyksen jäljiltä käytössään nopeat laajakaistayhteydet (Armitage ym. 2006, s. 19-21). Internet-pelattavuuden parantamiseksi sen pelimoottorista kehitettiin uusi versio,

joka sai nimen *QuakeWorld*. Se osoittautui paljon toimivammaksi vaihtelevanlaatuisilla Internet-yhteyksillä ja popularisoi useita tapoja välttää verkkoviiveen ongelmia nopeassa toimintapelissä. Useat suositut räiskintäpelisarjat, kuten *Half-Life*, *Counter-Strike* ja *Call of Duty*, saivat alkunsa Quake-pelien moottoreita käyttäen. Osiossa 2.5 tutustutaan tarkemmin yhteen versioon näiden verkkototeutuksista.

Seuraavina vuosina Internet-pelaamisen suosio kasvoi nopeasti ja ensimmäiset nykyaikaiset massiivimoninpelit ilmestyivät markkinoille. Vuodesta 2005 alkaen verkkopelaaminen alkoi yleistymään myös konsolialustoilla nk. seitsemännen sukupolven pelikonsolien saapuessa. Luotettavat laajakaistayhteydet ovat nykyisin paljon yleisempiä kuin verkkopelaamisen alkuaikoina. Silloista tilannetta vastaavia viive- ja luotettavuusongelmia kohdataan yhä jatkuvasti yleistyvissä mobiiliverkoissa, ja näitä verkkoja käyttävien mobiilipelien suosio on myös kasvanut räjähdysmäisesti. Integraatio erilaisiin sosiaaliverkkoihin on myös kasvattanut pelien verkkotoimintojen monimutkaisuutta. Näyttää myös siltä, että jaettua ympäristöä käyttävät virtuaalitodellisuus- ja lisätyn todellisuuden sovellukset tulevat yleistymään rajusti lähivuosina. Näiden toteutus tulee todennäköisesti vaatimaan verkkopelien kaltaisia tekniikoita.

2.2 Pelimaailman tila ja replikaatio

Pelin jaetun tilan eli pelimaailman viestitystä vastaava tehtävä tietokannoissa ja yleisemmin hajatetuissa järjestelmissä on *replikaatio*, joka pyrkii pitämään useamman järjestelmän osan samassa tilassa. Vaikka tiedon hajauttamisen tavoite on hieman erilainen — usein toimintavarmuuden tai suorituskyvyn parantaminen — hajautettujen tietokantojen replikaatiossa tunnetaan kaksi käsitettä (Pedone ym. 2000), jotka voidaan rinnastaa verkkopeleissä käytettyihin menetelmiin.

Aktiivinen replikaatio suorittaa samat toimenpiteet (samassa järjestyksessä) kaikissa tietokannan kopioissa, ja olettaa näiden johtavan yhteisestä lähtötilasta yhteiseen lopputilaan.

Passiivinen replikaatio suorittaa toimenpiteet vain yhdessä tietokannassa, ja välit-

tää tietueihin tehdyt muutokset muihin tietokannan kopioihin.

Verkkopeleissä vahvinta *aktiivista* replikaatiota edustaa pelin synkronointi pelkästään jakamalla pelaajien syötteitä jokaiselle pelin osallistujalle. Jotta verkkopelin jaettu tila säilyisi yhdenmukaisena pelkästään tämän avulla, jokaisen osallistujan on alustettava pelimaailmansa samaan tilaan ja pystyttävä suorittamaan pelin kaikki toiminnot samalla tavalla. Vastaavasti *passiivisen* replikaation välittämä tieto koostuu muutoksista pelimaailman tilaan ilman erityistä tietoa niitä tuottaneista tapahtumista. Matalimmalla tasollaan passiivisesti replikoituun pelimaailmaan osallistuvat ohjelmat ovat eräänlaisia etäpäätteitä, jotka eivät tunne pelimekaniikkaa ollenkaan, mutta pystyvät esittämään pelin tilaa pelaajalle ja lähettämään syötettä palvelimelle.

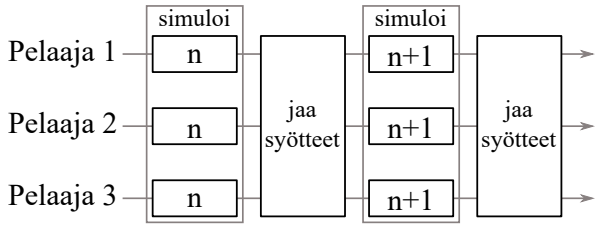
Kun peli ei luota pelkästään aktiiviseen syötetapahtumien välitykseen, viestitystapojen sekoittaminen on yleistä. Pelin osat voivat usein toimia itsenäisesti toisistaan ja hyötyä erilaisista viestitysmenetelmistä. Osallistujille annettavan vastuun määrä ja muutosten luonne vaikuttavat menetelmän sopivuuteen: Seurauksiltaan monimutkaiset tapahtumat, kuten pelihahmon kykyjen laukaiseminen tai uuden pelaajan liittyminen peliin, voivat olla tehokkaampia viestittää aktiivisen replikaation tapaan kuin tiedottamalla jokainen tapahtuman aiheuttama muutos erikseen.

2.2.1 Aktiivinen replikaatio ja determinismin ongelma

Puhdas aktiivinen tai *tapahtumavetoinen* replikaatio tekee vahvan oletuksen synkronoitavan järjestelmän determinismistä: samojen syötteiden ajaminen samassa lähtötilassa oleviin tietokannan kopioihin (samassa järjestyksessä) johtaa kaikissa niistä samaan lopputilaan (Pedone ym. 2000). Tietokannan tapauksessa syötteet voivat olla esimerkiksi muutoksia aiheuttavia SQL-lauseita tai korkeamman tason rajapinnan etäkutsuja, peleissä pelaajan napinpainalluksia tai muita vastaavia syötteitä. Verkkopelin toiminta voi perustua pelkästään tällaiseen syötteiden viestitykseen. Mallilla on vahvuuksia, jotka tekevät siitä käytännöllisen tietynlaisessa verkkopelissä. Pelimaailmalta se kuitenkin edellyttää *vahvaa toistettavuutta*, jonka saavutta-

minen ei ole aina helppoa.

Jotta järjestelmän jaettu tila pysyisi eheänä pelkästään tällä menetelmällä, syötteille on voitava asettaa selkeä järjestys. Monimutkaisissa järjestelmissä tämä voi edellyttää ylimääräisen järjestyksestä huolehtivan kerroksen lisäämistä järjestelmään (Défago, Schiper ja Urbán 2004). Verkkopelissä luonnollinen järjestys on yleensä pelin simulaatioaskel. Askelten pituus on käytännössä pidettävä vakiona jo pelin luotettavan toiminnan kannalta, joten ne voidaan numeroida yksiselitteisesti ja syötteet sitoa tiettyyn askelnumeroon. Jokaisen askeleen edellä pelimaailman kopiot odottavat syötettä osallistujilta askeltaen simulaatiota vasta, kun kaikkien pelaajien syöte on vastaanotettu. Kun kaikkien pelimaailman kopioiden lähtötila on sama, syötteiden ajaminen kopioihin samassa järjestyksessä pystyy (teoriassa) pitämään ne yhdenmukaisissa tiloissa. Tällaista ratkaisua voidaan kuvailla *sekventiaalisesti konsistentiksi* hajautetuksi järjestelmäksi. Peleissä tällaisesta synkronoinnista askeleittain käytetään usein termiä *lockstep*.



Kuvio 1. Lockstep-synkronoitu verkkopeli.

Verkkopelissä tällaisen replikaation suurimmat edut ovat yksinkertaisuus ja vähäinen verkkoliikenne. Jos pelin tila voidaan synkronoida pelkästään pelaajien syötteitä viestittämällä, verkkokoodin ei tarvitse tunkeutua sen muihin osiin. Lisäksi lähetettävän tiedon määrää rajoittaa vahvasti ihmispelaajan kyky tuottaa syötettä. Tämä sopii hyvin vertaisverkkoarkkitehtuuria käyttäviin peleihin, joissa siirrettävän tiedon määrä kasvaa nopeasti pelaajien lukumäärän kasvaessa. Kyky toistaa pelimaailman simulaatio pelkästään kootun syötteen perusteella tekee myös pelin taltiointi- ja uudelleentoistotoiminnot (engl. *replay*) helposti toteutettaviksi ja vähän tilaa vaativiksi. Koska jokaisen pelaajan tieto pelimaailmaan annetusta syötteestä on täydellinen, kuka tahansa pelin osallistujista voi myöhemmin tutkia taltiointia ja ke-

hittää taitojaan. Tällaisia taltiointeja on myös käytetty tekoälytutkimuksessa (Hsieh ja Sun 2008).

Käytetty replikaatiomenetelmä vaikuttaa pelaajien kykyyn huijata pelissä. Koska tässä tapauksessa jokaisella pelin osallistujalla on hallussaan täydellinen kopio pelimaailmasta ja peli voi jatkua vain syötteiden samanlaisen käsittelyn ansiosta, virheellisillä syötteillä huijaaminen on vaikeaa tai mahdotonta. Toisaalta tietoa pelimaailman tilasta ei voida kätkeä täydellisesti tällaisen pelin osallistujalta: Yleinen huijaus näin toimivissa strategiapeleissä on nk. "karttahacki" (engl. *map hack*), joka paljastaa koko pelialueen pelaajalle. Monen kilpailuhenkisemmän pelin mukana asennetaan tietokoneelle erillisiä ohjelmistoja, jotka pyrkivät havaitsemaan esimerkiksi peliprosessin muistiin koskemista (Armitage ym. 2006, s. 117).

Tällaisen replikaation toimivuus verkkopelissä riippuu suorituskyvyltään sekä verkkoyhteydeltään heikoimman osallistujan tasosta. Jos yksikin pelaaja ei pysty suorittamaan simulaatiota tavoitenopeudella, pelin tahtia on hidastettava kaikilla tai hidas pelaaja on poistettava pelistä. Vertaisverkkoarkkitehtuuria käyttäessä viive on ainakin yhtä suuri kuin hitain tahojen välinen verkkoyhteys. Käytännössä syötettä on vielä puskuroitava hieman viiveen vaihtelun peittämiseksi, tai liike pelimaailmassa voi edetä selvästi havaittavina "pulsseina" syöteviestien saapuessa epätasaisesti. Suuren pelimaailman alkutilan välittäminen uudelle pelaajalle voi myös olla raskasta.

Pienikin puute tällaisen verkkopelin toistettavuudessa muuttuu helposti huomaamattomasta "ominaisuudesta" pelin pysäyttäväksi ongelmaksi. Toistettavuusongelmia voi seurata eroista tietokoneiden ja niiden ohjelmakirjastojen toiminnassa. Yleiset ohjelmointivirheet, kuten muistin alustamatta jättäminen, voivat myös aiheuttaa lähes huomaamatonta epädeterminististä käytöstä. Vaikka synkronointivirhe ei näkyisi välittömästi, se voi kasvaa nopeasti näkyviin mittoihin eräänlaisen "perhosvaikutuksen" kautta väärin synkronoituneen tiedon vaikuttaessa muun järjestelmän toimintaan. Tällaisten virheiden tutkiminen voi olla hyvin työlästä ja edellyttää erityisten heuristiikkojen rakentamista peliin ("Synchronous RTS engines and a tale of desyncs" 2015) (Terrano ja Bettner 2001).

Yksi mahdollinen toistettavuusongelmien lähde on liukulukuoperaatioiden toteutus. Vanhemmissa peleissä liukulukuja pyrittiin välttämään suorituskykyongelmien takia, mutta nykyään niiden poistaminen pelimekaniikasta voi olla hyvin epäkäytännöllistä. Liukulukujen toteutusta koskeva IEEE 754 -standardi kattaa hyvin liukulukujen muistiformaatit ja perusoperaatiot, mutta jättää mm. trigonometriset funktiot ja tietyt tyyppimuunnokset suosituksiksi toteutuksille (IEEE Task P754 2008). Eri kääntäjät voivat tuottaa liukulukuoperaatioihin eri tavoilla käyttäytyvää koodia. Optimointeja ja muita poikkeamia standardeista voidaan kieltää kääntäjäparametreilla, mutta etenkin raskaissa fysiikkasimulaatioissa käytetyt pitkälle optimoidut kirjastot voivat olla hankalia toistettavuuden kannalta. Prosessorin tyyppi, sen lisäkäskykannat, ydinten määrä (säikeistetyssä simulaatiossa) ja jopa käyttöjärjestelmä voivat vaikuttaa liukulukupohjaisten simulaatioiden tuloksiin (Fiedler 2010). Grafiikkasuorittimien kaltaisten erillisten laskentalaitteiden käyttö voi lisätä tähän omat ongelmansa: mm. Nvidian PhysX-fysiikkakirjasto tukee GPU-laskennan käyttöä simulaatiossa. Tämän kirjaston dokumentaatio toteaa yksinkertaisesti: *"The first problem here is that the PhysX SDK is not deterministic."* (NVIDIA Corporation 2015) [1]

Pieniä poikkeamia voidaan havaita ajoissa laskemalla tarkistussummia pelin tilasta. Jos saman simulaatioaskeleen jälkeen osallistujien tarkistussummat eroavat toisistaan, pelimaailman synkronointi on epäonnistunut ja sen tila on korjattava ennen pelin jatkamista. Tämä voi tapahtua esimerkiksi pyytämällä kaikkia pelaajia liittymään peliin uudestaan: Civilization IV -peli tarjoaa tässä tilanteessa kaikille pelaajille pelin tallentamista. Yksi pelaajista voi tämän jälkeen perustaa tallennetusta tilasta uuden pelin. Koska tilanne seurasi pelien tilojen erkanemisesta toisistaan, uuden pelin tila voi erota jollakin tapaa muiden pelaajien aiemmin näkemästä tilasta.

1. Joissakin fysiikkamoottoreissa lievä satunnaisuus voi olla myös tahallinen ominaisuus numeerisen vakauden parantamiseksi.

Haasteistaan huolimatta tällainen replikaatio voi olla tehokasta peleissä, joissa:

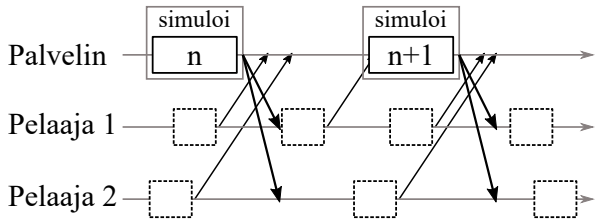
- Pelin toiminta voidaan toistaa täydellisesti.
- Pelimaailman alkutila voidaan viestittää kokonaan.
- Peliä voidaan suorittaa riittävän nopeasti kaikilla osallistujilla.

Nämä ehdot käytännössä sulkevat pois massiivimoninpelit, joissa pelimaailman tila on liian suuri, ja esimerkiksi raskasta fysiikkasimulaatiota käyttävät pelit, joissa jokaiselta pelin osallistujalta ei voida odottaa kykyä suorittaa simulaatiota tavoitenopeudella — jos simulaatio edes on toistettavissa. Tyypillinen käyttötapaus tällaiselle tapahtumavetoiselle replikaatiolle on strategia- tai roolipeli, jossa pelikentän koko on rajallinen ja pelaaja ei suoraan ohjaa mitään, vaan antaa pelimaailmaan vaikuttavia käskyjä. Toistettavuuden ja suorituskykyerojen ongelmat eivät ole yhtä vakavia konsolialustoilla, joilla pelaajien voidaan odottaa käyttävän samanlaista laitteistoa.

2.2.2 Passiivinen replikaatio ja suorituskyky

On selvää, ettei edellä esitelty replikaatiotekniikka sovellu kaikkiin peleihin. Hyvin suurissa tai muuten hankalasti täysin synkronoituvissa pelimaailmoissa voi olla käytännöllisempää jättää sen kokonaisen simulaation suorittaminen pois yksittäisten osallistujien tehtävistä ja jakaa pelimaailman tilaa jollakin muulla tavalla. Hajautetuissa tietokannoissa *passiivinen replikaatio* on toimintamalli, jossa tapahtumia käsitellään vain yhdessä tietokannan kopiossa ja muutokset viestitetään muille päivitettyjen tietueiden sisältönä (Pedone ym. 2000). Verkkopelissä tällainen malli tarkoittaa suorituksen siirtämistä *auktoriteetille*, joka vastaa tietyn pelimaailman osan tarkasta simulaatiosta ja viestittää siinä tapahtuvia muutoksia. Tyypillisesti auktoriteetti lähes kaikelle pelin toiminnalle on palvelin, joka voi olla yksi pelaajista tai erillinen prosessi.

Suurin etu tässä on se, ettei järjestelmän koko toimintaa tarvitse pystyä suorittamaan jokaisessa tietokoneessa. Verkkopelissä riittää, kun päivityksien vastaanottajat pystyvät esittämään ne riittävän uskottavasti. Vastaanottaja voi viiveen ja harvan päivitystahdin peittämiseksi suorittaa yksinkertaistettua versiota simulaatios-



Kuvio 2. Passiivisesti replikoitu asiakas-palvelin-verkkopeli.

ta. Ristiriitojen ratkaisu on yksinkertaista: uusi tilapäivitys ylikirjoittaa aiemman tai ennustetun tilan. Pienten muutosten luotettava viestitys ja järjestäminen ei myöskään ole täysin kriittistä.

Yleisiä haasteita tässä ovat viestityksen hallinta ja sen palvelimelle asettamat vaatimukset. Kun replikaatio ei tapahdu pelkästään peliin annettavan syötteen tasolla, verkkokoodi voi helposti levitä pelin muihin osiin ja hankaloittaa niiden kehittämistä ja ylläpitoa. Muutoksien viestittäminen voi myös vaatia paljon verkkoyhteydeltä. Viestityksen vaatimuksia voidaan yrittää hallita pelaajien havainnointikyvyn perusteella: Jos pelaajat pystyvät havainnoimaan vain rajallista aluetta, alueen ulkopuolella tapahtuvia muutoksia ei tarvitse viestittää. Täyttä numeerista tarkkuutta ei myöskään välttämättä tarvita vastaanottavassa päässä. Esimerkiksi 32-bittinen luku on selvästi tarpeettoman tarkka esitys pelihahmon katseen suunnalle, jos tiedon perusteella vain kierretään toisen pelaajan ruudulla näkyvää hahmografiikkaa.

Palvelimen ei tarvitse olla pelimaailman ainoa auktoriteetti. On mahdollista, että jokainen pelaaja vastaa esimerkiksi oman pelihahmonsa liikesimulaatiosta kokonaan. Näin pelaajan ohjaustuntuma ei kärsi helposti verkkoviiveestä. Viive voi kuitenkin paljastua, jos esimerkiksi pelaajan näkemästä tilanteesta jäljessä oleva palvelin toteaa pelaajan saaneen osuman, vaikka pelaaja näki tekevänsä väistöliikkeen. Tämä on yleistä massiivimoninpeleissä, joissa palvelimen toimintakyky voi vaihdella paljon pelin tilanteen mukaan. Auktoriteetin siirtäminen asiakkaille tekee myös mm. pelihahmon liikettä manipuloivat huijaukset helpoiksi toteuttaa. Peli voi näiden ehkäisemiseksi yrittää arvioida palvelinpäässään liikkeen “epäilyttävyyttä” — tai yksinkertaisesti tehdä järjestelyjä pelaajien valvontaan ja huijaamisen ilmoittamiseen.

Äärimmäisessä tapauksessa pelaaja ei ole pelkästään vastuussa hahmonsa liikkeistä, vaan myös muusta toiminnasta pelimaailmassa. Tällaisen auktoriteetin antaminen jokaiselle pelin osallistujalle on kuitenkin vaarallista monella tapaa. Tilannetta voidaan verrata leikkikenttään, jossa pelin osallistujat huutavat kilpaa "olen täällä" ja "osuin sinuun". Tämä voi olla vaikea saada toimimaan uskottavasti monimutkaisessa pelissä. Tällaiset ratkaisut voivat olla toimivampia konsolialustoilla, joilla huijaaminen peliohjelmaa muokkaamalla on harvinaista.

Passiivisesta replikaatiomallista hyötyvät eniten verkkopelit, joissa:

- Pelimaailman koko tai suorituskykyvaatimukset eivät suosi koko pelin suorittamista kaikilla osallistujilla.
- Muutoksia voidaan seurata ja viestittää tehokkaasti.

2.3 Verkkoarkkitehtuurit

Edellä kävimme läpi kaksi erilaista tapaa viestittää pelimaailman tilaa verkon yli. Näiden toimintaperiaatteiden lisäksi voimme luokitella verkkopelejä ja muita hajautettuja järjestelmiä niiden muodostamien verkkoyhteyksien perusteella:

Palvelinarkkitehtuurissa asiakkaat ottavat yhteyden palvelimeen, joka vastaa kaikesta tiedonvälityksestä ja mahdollisesti koko jaetusta tilasta.

Vertaisverkkoarkkitehtuuria käyttävässä järjestelmässä eri osat päivittävät jaettua tilaansa viestimällä keskenään.

Pelissä on hyvin yleistä sekoittaa eri replikaatiomenetelmiä, mutta verkkoarkkitehtuurit ovat lähes aina joko-tai-vaihtoehto; on hyvin harvinaista, että pelimaailman synkronointi käyttää samanaikaisesti useaa erilaista verkkoarkkitehtuuria. Tähän ei lasketa ulkoista palvelinta, jota käytetään esimerkiksi pelien muodostamiseen tai tulosten tilastointiin.

2.3.1 Vertaisverkkoarkkitehtuuri

Vertaisverkkoarkkitehtuuria (engl. *peer-to-peer, P2P*) käyttävissä peleissä pelimaailmaa ja siihen liittyvää viestitystä ei hallitse keskitetty palvelin, vaan pelin osallistujat muodostavat keskenään yhteyksiä ja viestittävät toisilleen päivityksiä pelin tilaan. Tämä ratkaisu voi toimia hyvin lähiverkoissa, joissa viiveet ovat matalia ja yhteyden ottaminen toisiin pelaajiin helppoa.

Nykyään hyvin yleiset osoitteenmuunnos (engl. *NAT, Network Address Translation*)- ja palomuuriratkaisut voivat johtaa ongelmiin Internet-pelissä: Palomuuri ei päästä ulkoa tulevia paketteja läpi tai NAT-reititin ei tiedä, mille lähiverkon tietokoneelle paketti pitäisi välittää (Armitage ym. 2006, s. 61–64). Yleinen ratkaisu tähän on se, että peliä aloittaessa erillinen palvelin tarjoaa nk. *matchmaking*-palvelua ja mahdollisesti auttaa yhteyden muodostamisessa hankalien Internet-yhteysratkaisujen läpi [2]. Hyvä esimerkki tästä on Steam, digitaalijakelu- ja peliyhteisöpalvelu, jonka Steamworks-ohjelmakirjaston toimintoihin kuuluu pelisessioiden muodostaminen sekä palvelinta käyttävissä että vertaisverkkoarkkitehtuuriin luottavissa peleissä (“Steamworks Documentation” 2015).

Tämän lisäksi vertaisverkkoarkkitehtuuria käyttävien pelien tyypillisiä haasteita ovat replikaatiomenetelmästä riippuen joko suuret viiveet, kun yksi osallistujista ei pysy pelin tahdissa tai kärsii yhteysongelmista, tai löyhemmässä synkronisaatiossa näkyvä epädeterministinen käytös, jossa muut pelaajat näyttävät toimivan pelisääntöjen vastaisesti. Tyypillinen esimerkki jälkimmäisen ongelmista on tilanne, jossa pelaaja näkee jo ehtineensä suojaan, kun toinen pelaaja ilmoittaa osuneensa tämän pelihahmoon. Harva peli edes yrittää ratkaista tämän tilanteen monimutkaisemmin kuin hyväksymällä, että osuman ilmoittaja on aina oikeassa.

Puhdasta vertaisverkkomallia käyttävät pelit näyttävät nykyään olevan melko harvinaisia. Monet konsolipelit, jotka yleisesti mielletään vertaisverkkotapaan toimiviksi, sopivat keskenään palvelimena toimivasta osallistujasta. Vertaisverkkoarkkitehtuurin soveltamisesta massiivisiin pelimaailmoihin on myös tehty paljon tut-

2. Yhteysreitin avaamisesta NAT-reitityksen läpi käytetään usein termiä *hole punching*.

kimusta, mutta käytännössä toimivat ratkaisut eivät ainakaan ole yleisiä. Halpa palvelinkapasiteetti, NAT-ratkaisujen yleisyys sekä verkkopelien kehittyminen yhä enemmän lisensoitaviksi tai vuokrattaviksi palveluiksi ovat voineet vaikuttaa tähän.

2.3.2 Palvelinarkkitehtuuri

Palvelinarkkitehtuurissa asiakkaat eivät kommunikoi keskenään, vaan muodostavat yhteyden tietoa välittävään palvelimeen. Se voi vastata koko pelimaailman simulaatiosta, tai toimia pelkästään syötteiden välittäjänä. Selvä etu tässä on se, että palvelimen käyttö välttää vertaisverkon ongelmat käyttökelpoisten yhteyksien muodostamisessa pelin osallistujien välille.

Palvelimena voi toimia yksi pelaajista (nk. *listen*-palvelin) tai pelkästään palvelimena toimiva erillinen prosessi (engl. *dedicated server*). Pelaajan perustamissa palvelimissa on kaksi pääongelmaa. Jos pelin perustaja haluaa lopettaa, verkkopelin on pysähdyttävä tai jotenkin neuvoteltava uudesta palvelimesta (jos palvelin edes jakoi tarpeeksi tietoa tämän mahdollistamiseksi). Muut pelaajat voivat myös kokea palvelimena toimivan pelaajan matalan viiveen epäreiluksi. Suurien tai jatkuvassa käytössä olevien virtuaalimaailmojen tapauksessa erillisen palvelimen käyttö on luonnollinen ratkaisu. Hyvin raskaissa massiivimoninpeleissä voidaan käyttää useita palvelimia, jolloin asiakasohjelman tai jonkinlaisen välityspalvelimen on huolehdittava viestien välittymisestä pelimaailman oikealle palvelimelle.

Palvelimen käyttäminen ei välttämättä tarkoita asiakkaiden jättämistä pelkiksi "päätteiksi". Vain viestinvälittäjänä toimivaa palvelinta voidaan kutsua viesti- tai pakettipalvelimeksi (engl. *packet server*). Tällainen palvelin voi auttaa aktiivireplikoidun järjestelmän toimintaa selkeyttämällä syötteiden järjestämistä: Jos viestien oikea järjestys on se, missä palvelin vastaanotti ne, pelin ei tarvitse erikseen odottaa syötettä jokaiselta asiakkaalta kuten puhtaan vertaisverkon tapauksessa (Waveren 2006). Hajautettujen järjestelmien yhteydessä tällaista prosessia kutsuttaisiin järjestelmän *sekvensseriksi* (Défago, Schiper ja Urbán 2004).

Raskaammassa (auktoritatiivisessä) palvelinarkkitehtuurissa pelin simulaatio on suurimmaksi osaksi palvelimen vastuulla, ja pelin asiakasohjelman rooli on lähinnä tuottaa tälle visualisaatio ja käyttöliittymä. Tällä on etunsa: Kun pelin todellinen tila on palvelimen hallinnassa, asiakkaiden resursseja voidaan säästää rajaamalla kerralla paljastettavaa osaa pelimaailmasta. Pelimekaniikkaan voidaan myös tehdä pieniä muutoksia ilman asiakasohjelman päivitystä (käyttöliittymän, verkkokoodin ym. yhteensopivuustekijöiden rajoissa). Koska asiakkaat eivät välttämättä tiedä pelimaailmasta paljoa, mahdolliset huijaukset tällaisissa peleissä rajoittuvat lähinnä asiakasohjelman muokkaamiseen esimerkiksi seinien läpi näkemiseksi (engl. *wallhack*) tai tähtäämisen kaltaisen toiminnan automatisoinniksi (engl. *aimbot*). Palvelimen käyttö voi myös helpottaa pelin integraatiota ulkoisiin palveluihin, kuten pistetilastoihin, sosiaalitoimintoihin ja pelin sisäisiin ostoksiin.

2.4 Verkkoyhteyksistä

Replikaatiotekniikasta riippumatta verkkopeli tarvitsee toimiakseen verkkoyhteyden, joka on käytännössä toteutettava TCP- tai UDP-tiedonsiirtoprotokollan päälle. Tämän toteutuksella on seurauksia verkkopelin toimivuudelle etenkin langattomissa verkoissa, joissa viiveet ja pakettien hävikki voivat vaihdella paljon.

2.4.1 Tiedonsiirto

Pelien käytännön verkkototeutus voidaan usein jakaa kahteen osaan: istuntoa ja tiedonvälitystä hallitsevaan matalan tason protokollaan ja verkkokoodiin, joka viestii tämän protokollan ylitse pitääkseen eri pelien tilan samana. Tyypillinen pelin verkkototeutus on rakennettu UDP-protokollan päälle, sillä TCP:n yritys toteuttaa luotettava kaksisuuntainen “sarjaväylä” vastaa huonosti pelien viestitystarpeita etenkin epäluotettavissa langattomissa verkoissa (Wang, Jarrett ja Sorteberg 2009):

- **Viestien häviäminen** voi olla pelin kannalta siedettävämpää kuin niiden uudelleenlähettämisestä aiheutuva viive. Kun vanhemman viestin saapumista odottaa jonossa uudempia, jotka puretaan saman pelimaailman osan tilan pääl-

le välittömästi puuttuvan paketin saapumisen jälkeen, vanhan viestin luotettavasta saapumisesta ei ole hyötyä.

- **Saapumisjärjestyksen** ei tarvitse olla täydellinen, kun viestit vaikuttavat pelin eri osiin: Esimerkiksi pelin sisäisellä chat-toiminnolla tuskin on merkittävää vuorovaikutusta pelimaailman tilan kanssa. Useamman TCP-yhteyden käyttäminen näille on mahdollista, mutta kuluttaa palomuurien ja NAT-laitteiden resursseja, voi vaikuttaa siirtonopeuksiin arvaamattomasti eikä peli voi näin hallita saatavilla olevan kaistan jakoa erillisten viestikanavien käyttöön.

Yhtäaikainen TCP- ja UDP-yhteyksien käyttö pelkän TCP-yhteyden ongelmien kiertämiseen voi aiheuttaa epävakautta tiedonsiirtonopeuksissa, jos verkkoliikenne kulkee yhdenkin QoS (engl. *Quality of Service*) -järjestelmän läpi. Moni QoS-ratkaisu suosii UDP-liikennettä olettaen sen olevan aikakriittistä medialähetystä. Lisäksi TCP:n "ikkuna"-pohjainen lähetysnopeuden hallinta soveltuu paremmin jatkuvaan tiedonsiirtoon kuin epätasaiseen, aikakriittiseen viestitykseen ja voi johtaa huomattavaan vaihteluun yhteyden siirtonopeuksissa (Chen ym. 2006). Joitakin TCP-protokollan ongelmia voidaan välttää asettamalla TCP-soketille asetuksia, jotka ovat valitettavasti hyvin alustakohtaisia.

2.4.2 Viiveen kätkeminen

Pelatessa lähiverkoissa yhteyden laadusta ei juuri tarvitse välittää. Verkosta aiheutuva viive on yleensä pienempi kuin pelisilmukan suorittamiseen kuluva aika, pakettien hävikki käytännössä olematon ja siirtonopeus vähintäänkin riittävä. Internet-yhteyteen siirryttäessä nämä oletukset voidaan hylätä nopeasti. Useimmissa saatavilla olevissa Internet-yhteyksissä tiedonsiirtonopeus riittää hyvin pelien käyttöön, mutta etenkin langattomissa verkoissa esiintyy usein arvaamattomia vaihteluita viiveessä ja pakettien häviössä.

Verkkoyhteys ei ole ainoa asia, joka voi aiheuttaa viivettä pelin toimintaan. Monet niistä — kuten näyttö- ja syötelaitteiden sekä käyttöjärjestelmän toteutuksesta aiheutuvat viiveet — eivät kuitenkaan ole pelin hallittavissa. Yhteysongelmien

havaittavuutta ja peliin aiheutuvaa häiriötä voidaan vähentää verkkokoodin optimointien lisäksi pelin ja sen käyttöliittymän sopivalla suunnittelulla (Brun, Safaei ja Boustead 2006).

Jonkinlainen mittatikku hyväksyttävälle verkkoviiveelle saadaan pelaajista itsestään. Ihmisen reaktioaikaa erilaisiin ärsykkeisiin on tutkittu yli vuosisadan ajan. Yksinkertaisimmissa koetilanteissa kokonaisviive visuaalisen ärsykkeen näyttämisen ja napin painalluksen välillä on n. 150-300 millisekuntia. IEEE:n hajautettujen interaktiivisten simulaatioiden standardi [3] suosittelee pitämään järjestelmän viiveen alle 100 millisekunnissa ("IEEE Standard for Distributed Interactive Simulation - Communication Services and Profiles" 1996). Samanlaisia rajoja tukevat myös oikeilla peleillä tehdyt käytännön tutkimukset. Yhdessä tapauksessa (Chen, Huang ja Lei 2006) havaittiin pelaajien pelisessioiden lyhenevän neljäsosaan, kun hidastempoisen MMO-pelin viive kasvoi 250 millisekuntiin. Viiveen voimakas vaihtelu (engl. *jitter*) voi lisätä sen häiritsevyyttä huomattavasti, kun keskimääräinen viive on jo korkea (Beznosyk ym. 2011).

Jatkuvan liikkeen tapauksessa rajallista lähetysnopeutta ja (epätasaista) viivettä voidaan kätkeä olettamalla, että liike jatkuu ainakin lyhyen ajan ilman uutta tietoa palvelimelta tai muilta pelaajilta. Kappaleen nykyisistä liiketiedoista voidaan ennustaa seuraavia liikkeitä jatkamalla kappaleiden liikeratoja. Kun uusi tieto liikkeestä saadaan, ennuste voidaan korvata sillä. Verkkopelien asiayhteydessä tätä kutsutaan usein ennustamiseksi asiakaspäässä (engl. *client-side prediction*). Laajemmassa kontekstissa vastaavaa tekniikkaa, nk. laskelmasuunnistusta (engl. *dead reckoning*) käytetään mm. navigointilaitteissa.

Kun uusi liiketieto ja paikallinen ennuste eivät vastaa toisiaan hyvin, kappaleen näkyvä sijainti voi muuttua äkilisesti. Peliyhteisöissä ilmiötä kutsutaan *kuminauhaefektiksi* (engl. *rubberbanding*) [4] tai *warppaamiseksi* (engl. *warping*). Kun poikkeamat ovat riittävän pieniä, asiakaspäässä voidaan yrittää kätkeä niitä säätämällä kappaleiden

3. DIS; Distributed Interactive Simulation.

4. Kuminauha-termiä käytetään myös häiritsevästi pelaajan suoritukseen mukautuvasta pelin tekoälystä.

liikettä siten, että ne siirtyvät vähitellen kohti oikeaa liikerataansa (Smed, Kaukoranta ja Hakonen 2002). Ennusteita luova asiakasohjelma voidaan huomioida jo palvelimella esimerkiksi lähettämällä vähemmän päivityksiä, kun asiakkaan voidaan olettaa ennustavan liikettä oikein. Passiivisessa replikaatiossa ongelmaa voidaan ehkäistä yksinkertaisesti nostamalla lähetystahtia tai viestittämällä liikettä korkeammalla asteella: Pelkkien sijaintien lisäksi voidaan lähettää nopeuksia, kiihtyvyyksiä tai jopa hahmoille annettuja syötteitä, jos vastaanottaja pystyy käsittelemään niitä samalla tavalla.

Kappaleiden liikkeen lisäksi ennustamista voidaan soveltaa myös muuhun toimintaan. Verkkoyhteydestä johtuva lyhyt viive pelaajan hyökkäyksen laukaisussa on huomattavasti vaikeampi havaita, jos *kaikkiin* hyökkäyksiin lisätään pieni viive, kuten lyhyt animaatio ennen hyökkäyksen toteutumista. Jos hyökkäystä ei voida keskeyttää (tai sen keskeytyminen ei ole kovin yleistä), pelin asiakasohjelma voi alkaa näyttämään sen animaatiota paikallisesti jo pelaajan painaessa nappia, ja siten luoda lähes aina kestävän illuusion viiveettömyydestä.

Palvelin voi myös osallistua pelaajien viiveen kätkemiseen. Kun pelimaailma on riittävän yksinkertainen, palvelin voi helposti säilyttää muutamia aiempia versioita siitä. Kun palvelin saa viestin pelaajan aikakriittisestä syötteestä, se voi käsitellä tapahtuman sitä tilaa vasten, joka pelaajan viiveen perusteella oli ajankohtainen viestin lähtiessä (Bernier 2001). Näin pelaajan toiminta kärsii vähemmän viiveestä. Hajautettujen järjestelmien termeissä tätä voitaisiin kutsua karkeaksi *time warp*-synkronisaatioksi; kun tapahtumat on käsitelty väärässä järjestyksessä, järjestelmän tilaa "kelataan" aiempaan pisteeseen ristiriidan ratkaisemiseksi. Liian pitkälle vietynä pelaajat voivat kokea tämän epäreiluksi: nopeassa räiskintäpelissä suuresta viiveestä kärsivä pelaaja voi helposti osua toisiin pelaajiin, jotka näkivät jo ehtineensä suojaan. Eräässä tutkimuksessa havaittiin, että tätä menetelmää käyttäväs-sä *Counter-Strike*-pelissä viive vaikutti hyvin vähän useimpien pelaajien pisteisiin, mutta erittäin korkeasta viiveestä kärsivät pelaajat saattoivat jopa menestyä muita paremmin (Dick, Wellnitz ja Wolf 2005).

Viivettä voidaan kätkeä ennustamalla sekä passiivi -että aktiivireplikoiduissa peli-

maailmassa. Jälkimmäisessä tapauksessa pelin asiakkaan on säilytettävä ennustetun tilan lisäksi kopiota pelin viimeisimmästä oikeasta tilasta. Ohjelman on siten suoritettava pelin mekaniikkaa kokonaisudessaan ja samalla ajettava siitä ennustettua versiota ainakin osittain.

Viimeinen ja vahvin keino viiveen kätkemiseen on pelaajien auktoriteetin lisääminen. Jos pelihahmon liiketuntumaa ei saada ennustamallakaan riittävän hyväksi, palvelin voi antaa pelaajan määrätä oman liikkeensä suoraan. Tässä tapauksessa pelipalvelin on passiivisessa roolissa pelaajan liikkeen viestittämisessä ja joutuu asettamaan paljon luottamusta pelaajaan. Jos mikään tekninen keino ei auta, viiveen häiritsevyyteen voidaan vaikuttaa sopivalla pelisuunnittelulla. Laitteiston rajoitukset ovat ohjanneet tietokonepelien suunnittelua niiden alkuajoilta asti, eikä Internet-ympäristö ole tähän poikkeus.

2.5 Olemassaolevia ratkaisuja

Tässä osiossa tutustutaan vielä tarkemmin muutamaan verkkopeliin, joiden toteutuksesta löytyi teknistä tietoa dokumentaation, kehittäjäpäiväkirjojen tai julkaistun lähdekoodin muodossa. Tiedon pohjalta voidaan arvioida erilaisten skripti- ja verkkokoodiratkaisujen soveltuvuutta oman pelikehyksen tarpeisiin.

Quake-pelien moottorit [5] olivat ensimmäisiä 3D-pelimoottoreita, joissa nopea pelimekaniikka yhdistyi toimivaan Internet-pelaamiseen ja kattavaan *modattavuuteen*. Ensimmäinen peli käytti pelimekaniikkansa toteutuksessa omaa *QuakeC*-kieltään, jota käännettiin erillisellä työkalulla tavukoodiksi. Kieli oli ominaisuuksiltaan alkeellinen, mutta salli silti pelin muokkaamisen ennennäkemättömillä tavoilla. Yksinkertainen verkkototeutus jätti pelimekaniikan lähes pelkästään palvelimen tehtäväksi; pelin asiakkaan käyttöliittymää ei juuri ollut mahdollista muokata QuakeC:llä. Asiakkaan näppäimenpainalluksiin oli kuitenkin mahdollista sitoa palvelimelle lähetettäviä yhden tavun “impulse”-viestejä, joiden avulla modit lisäsivät

5. Pelien kehittäjä iD Software käyttää kahden ensimmäisen pelin moottoreista nykyään nimeä *iD Tech 2*.

omia syötetoimintojaan (kuten alkeellisia valikoita) peliin. Alkuperäisen verkkopelitoteutuksen ongelmia paikanneen QuakeWorld-julkaisun jälkeen näiden verkkopeli on perustunut pelimekaniikkaa ajavaan palvelimeen, joka viestittää pelimaailman tilaa asiakkaille UDP-paketteina.

Quake III Arena -pelin verkkokoodi ("Quake III Arena GPL Source Release" 2015) on näistä ehkäpä paras esimerkki passiivisen replikaation periaatteista: Palvelin ja asiakas toimivat hyvin eri tavoilla, ja asiakkaan tarvitsee tietää varsin vähän pelin tilasta. Pelin osallistujat lähettävät säännöllisesti syötetilojaan (tavoitellun liikkeen ja katseen suunnat, valittu ase, painikkeiden tila) palvelimelle. Palvelin lähettää pelaajille pelimaailman tilaa, joka on rajattu muutoksiin edellisestä asiakkaan kuittaamasta päivityksestä. Palvelin varastoi näitä tiloja (engl. *snapshot*) 32 kappaletta jokaista asiakasta kohti; jos asiakas ei onnistu kuittaamaan tilapäivityksiä riittävän tiheään, yhteys lopetetaan.

Tämä toimii hyvin pienissä pelimaailmoissa, joissa liikkuvia kappaleita on rajallinen määrä, mutta pelimaailman monimutkaisuuden kasvaessa aiempien tilojen varastoiminen ja pakkaus lähetettäväksi tekevät palvelimen hyvin raskaaksi. Kun päivitykset kasvavat liian suuriksi ja verkkoyhteyden on pilkottava ne osiin siirtoa varten, niiden häviäminen siirrossa muuttuu myös paljon todennäköisemmäksi. C-kielellä toteutettu peli on myös työläs ylläpitää: Korkeamman tason abstraktioiden puutteessa uudentyyppisen kappaleen toteuttaminen verkkopeliin vaatii muutoksia moneen osaan pelin lähdekoodia (Waveren 2006). Paloja Quake-moottoreista ja niiden kehitystyökaluista löytyy vieläkin Valve Softwaren *Source*-moottorista, jota on käytetty mm. *Half-Life 2* -ja *Team Fortress 2* -peleissä. Kehittäjädokumentaationsa perusteella *Source* käyttää hyvin samanlaista verkkokoodia, mutta lisää siihen *time warp*-toiminnon viiveen korjaamiseen (Bernier 2001; "Source Multiplayer Networking" 2015).

Quake III:sta eteenpäin kehitetty Doom 3 muutti verkkototeutusta siten, että pelin asiakkaat pystyivät ajamaan suurta osaa pelimekaniikasta. Tämä helpotti yksinpelin toteutusta poistamalla siitä suurimman osan palvelimen ja asiakkaan välisestä viestinnästä. Koska asiakkaat pystyivät tämän myötä suorittamaan paikallisesti lähes

kokonaista versiota pelimekaniikasta, verkkopelissä ennustetun liikkeen tarkkuutta pystyttiin parantamaan tiedolla muiden hahmojen suunnittelemista liikkeistä.

Unreal Engine ("Unreal Engine" 2015) on yksi suosituimmista kaupallisista pelimoottoreista, ja on toiminut satojen kaupallisten pelien alustana sitten vuonna 1997 julkaistun *Unreal*-pelin. Moottorin tuki verkkopeleille ja moottorin sisäiselle skriptaukselle on ollut huomattavan kehittynyt sen ensimmäisistä versioista asti. Verkkokoodi Unreal Enginessä toimii asiakas ja palvelin -mallin mukaisesti. Pelin korkeamman tason toiminnot on toteutettu moottorin versioissa 1, 2 ja 3 upotetulla *UnrealScript*-kielellä[6], jolla voidaan toteuttaa peliin uusia luokkia ja määritellä, miten niiden jäsenmuuttujat ja metodikutsut käyttäytyvät verkkopelissä. Pelimaailman `NetMode` -asetus ilmaisee, onko kyseessä yksinpeli, erillinen palvelin, asiakas tai nk. *listen*-palvelin, jolla on myös paikallinen pelaaja. Jäsenmuuttujien arvojen muuttaminen ja metodikutsut voidaan automaattisesti muuttaa etäviesteiksi verkkopelitilassa. Pelin kappaleilla on tieto niiden omistajasta ja auktoriteetistä, ja näiden perusteella voidaan rajoittaa sallittuja etäkutsuja. Palvelin viestittää passiivisesti asiakkailleen pelimaailman kappaleiden jatkuvasti muuttuvia ominaisuuksia, kuten sijaintia, nopeutta ja animaatiotilaa. Kaistavaatimuksia voidaan hallita säätelemällä tämän replikaation lähetystiheyttä; etäkutsuja ei voida karsia näin. Kehittäjän dokumentaatio varoittaakin kutsumasta replikoitavia metodeja liian usein, jotta yhteys palvelimeen ei tukkiutuisi. Yhteysviivettä pyritään kätkemään ennustamalla pelaajan ja ympäristön liikettä paikallisesti.

Strategiapelien verkkototeutus käyttää yleensä "lockstep"-mallia eli aktiivista replikaatiota. Tyypillinen strategiapelin kenttä on kaksiulotteinen kartta, jossa pelaajat komentavat omistamiaan yksiköitä. Pelin tahti on usein räiskintäpeliä hitaampi, ja sen simulaatio hyvin yksinkertaista. Koska kaikkien pelaajien voidaan odottaa pystyvän suorittamaan peliä täydellisesti, aiemmin esitelty aktiivisen replikaation malli sopii näihin peleihin hyvin. Koska pelaaja ei suoraan ohjaa minkäänlaista pelihahmoa, pieni viive ei haittaa tuntumaa peliin ja syötettä voidaan puskuroida yhtey-

6. Unreal-moottorin 4. versio on pyrkinyt suosimaan C++-koodin käyttöä nopeussyistä. Yksinkertaisissa toiminnoissa voidaan käyttää Blueprint-nimistä visuaalista ohjelmointikieltä.

songelmien peittämiseksi. Näin *Age of Empires* -strategiapeli saavutti tuhansien yksiköiden tilan synkronoinnin puhelinmodeemiyhteyksillä vuonna 1997 (Terrano ja Bettner 2001). Suosittu *Starcraft*-pelisarja näyttää käyttävän samanlaista tekniikkaa.

Massiivimoninpelit käyttävät lähes aina omia erikoistuneita kehyksiään, mutta niiden toimintaperiaatteet suosivat selvästi palvelinmallia. Pelien yleisestä suunnittelusta, käyttäytymisestä korkean viiveen aikana ja pelaajayhteisöjen keksimistä huijausmenetelmistä voidaan päätellä, että pelimaailman replikointi näissä peleissä on suureksi osaksi passiivista (Yan ja Randell 2005). Palvelin on vastuussa valtaosasta pelimaailman toiminnoista, mutta usein jättää pelihahmojen liikkeen asiakkaiden vastuulle. Tämä pitää pelien liiketuntuman hyvälaatuisena lähes joka tilanteessa, mutta palvelimen tieto pelaajan sijainnista voi poiketa huomattavasti siitä, mitä pelaaja ruudullaan näkee. Hahmon kykyjen aktivointi ja muu vuorovaikutus pelimaailman kanssa tapahtuu tyypillisesti vain palvelimen kautta. Tästä seuraa myös genren tyypillinen merkki yhteysongelmasta: vain oma hahmo näyttää liikkuvan ympäriinsä, eikä pysty tekemään mitään tämän lisäksi.

Alan kirjallisuus tuntee monia näitä kehittyneempiä ratkaisuja; erityisesti vuoden 2000 paikkeilla kehitettiin useita prototyyppejä vertaisverkkoarkkitehtuuria käyttävistä massiivimoninpeleistä, joiden luvattiin skaalautuvan paremmin valtaviin käyttäjämääriin. Näistä yksikään ei näytä edenneen kaupalliseksi tuotteeksi asti.

Taulukossa 1 on lyhyt yhteenveto havainnoista.

replikaatiomalli	**arkkitehtuuri**	**huomioita**
aktiivinen	palvelin	Yksinkertainen, palvelin viestipuskurina.
aktiivinen	vertaisverkko	Yksinkertainen. Viiveongelmia.
passiivinen	palvelin	Perinteinen “client-server”.
passiivinen	vertaisverkko	Ristiriidat hankalia selvittää. Huijaus helppoa.

Taulukko 1. Yhteenveto pelien verkkototeutuksista.

3 Lua-kieli pelikehyksessä

Pelikehyksen korkean tason pelimekaniikan ja verkkoviestinnän toteutukseen valittiin Lua-skriptikieli. Koska kielen ominaisuudet ovat tärkeitä pelikehyksen toiminnan ja sen verkkototeutuksen kannalta, tässä luvussa käydään läpi kieli yleisesti ja sen upottaminen moderniin C++-sovellukseen.

3.1 Miksi upotettu kieli?

Käytännön pakosta pelien kehittäminen usein aloitetaan C:n tai C++:n kaltaisella järjestelmäohjelmointikielellä. Tarve hallita suurta määrää muistia ilman roskienkeruun odottamattomia pysähdyksiä tai hukkatilaa sekä käyttää helposti järjestelmän ja erilaisten työkalukirjastojen toimintoja ovat vieläkin painavia tekijöitä peliohjelmoinnissa. Tällaiset kielet vaativat kuitenkin erityistä huomiota muistinhallintaan ja alustaläheisyyden tai suorituskyvyn takia määrittelemättä jätettyihin käytöksiin. Tyypillisiä vaaroja ovat alustamattomat muuttujat, taulukoiden ja muistialueiden rajojen ylittäminen, muistin vuotaminen ja arvaamaton käytös virhetilanteissa. Hyvin korkealla tasolla toimiva, moneen paikkaan ulottuva ja suhteellisen vähän pelin suoritusajasta vievä pelimekaniikka vastaa heikosti järjestelmäohjelmointikielen suunnittelun lähtökohtia.

Monet sovellukset, jotka edellyttävät matalan tason suorituskykyä ja korkean tason toiminnallisuutta, ovat lähestyneet tätä ongelmaa toteuttamalla osia ohjelmasta toisella ohjelmointikielellä, joka soveltuu paremmin korkean tason toiminnallisuuden rakentamiseen ja kokemattomien kehittäjien käytettäväksi. Skriptikieli määritellään yleisesti työkaluksi, jonka tarkoituksena on mukauttaa ja automatisoida olemassaolevan järjestelmän toimintaa. Esimerkiksi ECMAScript-standardin (ECMA International 2011) määrittelee termin näin:

> *A scripting language is a programming language that is used to manipulate, customise, and automate the facilities of an existing system. In such systems, useful functionality is already available through a user interface, and the scrip-*

> *ting language is a mechanism for exposing that functionality to program control. In this way, the existing system is said to provide a host environment of objects and facilities, which completes the capabilities of the scripting language. A scripting language is intended for use by both professional and non-professional programmers.*

Määritelmän kuvailema mahdollisuus sovelluksen tai järjestelmän laajentamiseen tai mukauttamiseen ilman erityistä ammattitaitoa tai syvällistä tietoa sen toiminnasta on juuri se, mitä pelin korkeammalla tasolla tarvitaan. Lyhyesti:

- Yksinkertaisempi kieli laskee kynnystä ohjelmoinnin aloittamiseen.
- Kääntämiseen ei tarvita hankalaa kehitysympäristön pystyttämistä.
- Käytös virhetilanteissa on vahvasti määritelty.
- Ohjelmakoodi voidaan eristää muusta ympäristöstä (engl. *sandboxing*).
- Muualta saadun koodin ajaminen on pienempi tietoturvariski.

Nämä eivät ole etuja pelkästään pelin kehityksessä: pelien suosio voi seurata juuri mukauttamisen helppoudesta. Mukauttaminen, eli *modaus* tai *modaaminen* (engl. *modding*) pitää useita vanhoja pelejä harrastajien jatkuvassa suosiossa. Moni uusi peli on syntynyt juuri harrastajien modausprojektina, ja usein hyötypelien kehittämisessä lähdetään liikkeelle olemassaolevan pelin mukauttamisesta. Pelin valmis sisältö voi olla suureksi avuksi, jos kehittäjillä ei ole riittävää kokemusta esimerkiksi hahmoanimaatioiden luonnissa. Tällaisesta aloittava ohjelmoija tai pelisuunnittelija voi keskittyä tiettyihin pelinkehityksen tai pelimekaniikan osiin (Scacchi 2011) ja siten osallistua mittakaavaltaan suuren projektin kehitykseen ilman organisoitunutta kehitystiimiä. Ohjelmistoa, joka on suunniteltu juuri tarjoamaan mukautettavuutta ja helppoja työnkulkuja pelisisällön luontiin, kutsutaan usein pelimoottoriksi (El-Nasr ja Smith 2006) (Phelps ja Parks 2004) tai pelikehykseksi.

Tässä työssä käsiteltävän kehyksen skriptikieleksi valittiin Lua. Myös JavaScript- ja Python-kielten käyttöä harkittiin, mutta Lua osoittautui näitä yksinkertaisemmaksi ratkaisuksi.

JavaScript on selvästi hyvin tunnettu web-sovellusten ja palvelintenkin ohjelmoin-

tikielenä. Eri JavaScript-tulkkeja on kuitenkin hyvin monta ja ne vaihtelevat paljon C-rajapintojensa ja tuettujen kielen versioiden suhteen. Tehokkaat tulkit ovat myös hyvin monimutkaisia: Googlen JavaScript-kone *V8* sisältää noin 600 000 koodiriviä. Vertailun vuoksi tätä kirjoittaessa koko työn käsittelemä pelikehys koostui noin 20 000 rivistä C++-lähdekoodia, ja V8:aan verrattavissa oleva LuaJIT n. 70 000 rivistä. Useiden JavaScript-koneiden dokumentaatio oli myös puutteellista tai vanhentunutta, erityisesti web-ympäristön ulkopuolella käyttämisen suhteen.

Python-kieli on houkutteleva vaihtoehto siistin kielioppinsa ja yleisen käyttäjäystävällisyytensä takia. Vaikka sitä on käytetty muutamissa peleissä upotettuna skriptikielenä, näyttää kuitenkin siltä, että kieli soveltuu paremmin erillisiä ohjelmakirjastoja yhdistäväksi "liimaksi" kuin olemassaolevan sovelluksen automaatioon. Pythonin C-rajapinta on hyvin monimutkainen ja vahvasti sidoksissa tulkin sisäisiin toteutusyksityiskohtiin. Se sisältää yli 600 funktiota, kun Lua 5.1:n vastaava luku on 79. Kielen "virallinen" toteutus CPython ei myöskään ole erityisen suorituskykyinen edes tulkkaavien virtuaalikoneiden joukossa. Nopeampi ja suurimmaksi osaksi yhteensopiva *just-in-time*-kääntämistä käyttävä PyPy on hyvin suuri ohjelmisto, ja sen soveltuvuudesta pelikäyttöön ei ollut saatavilla tietoa työtä aloittaessa [1]. Pythonin suurin etu, eli sen valtava kirjastokokoelma, ei myöskään ole erityisen kiinnostava upottaessa kieltä suuremman ohjelman automaatiotyökaluksi.

3.2 Lua-kielestä yleisesti

Lua-ohjelmointikieltä on kehitetty vuodesta 1993 alkaen Rio de Janeiron katolisessa yliopistossa [2]. Sen alkuperäisenä tavoitteena oli automatisoida monimutkaisia tieteellisen laskennan eräajoja. 1990-luvun kuluessa kieli kehittyi pienen piirinsä automaatiotyökalusta monipuoliseksi ohjelmointikieleksi, joka on poiminut vaikutteita mm. Scheme-, Modula- ja AWK-kielistä. Kielen selkeys, yksinkertainen C-rajapinta ja virallisen toteutuksen yhteensopivuus useiden laitteistoalustojen kanssa ovat teh-

1. PyPyyn on lähiaikoina lisätty *generational garbage collection*-toiminto, jonka pitäisi nopeuttaa sen toimintaa peleissä.

2. PUC-Rio — Pontifícia Universidade Católica do Rio de Janeiro

neet siitä suositun upotettuna ohjelmointikielenä monissa peleissä ja hyötyohjelmissa (Ierusalimschy, Figueiredo ja Celes 2007).

Lua on dynaamisesti tyypitetty ja luonteeltaan pääosin proseduraalinen kieli. Sen kielioppi muistuttaa ulkoasultaan Modula-kielen vaikutteiden kautta hieman ALGOL- ja Pascal-kieliä. Tietorakenteidensa osalta Lua on kuitenkin lähempänä JavaScriptiä: sen rakenteinen tietotyyppi *table*, eli taulu, on nimensä mukaisesti hajautustaulu — järjestämätön joukko avain ja arvo -pareja [3] Sen yleinen nimiavaruus on myös taulu, joka voidaan tarvittaessa vaihtaa funktiokohtaisesti. Koska Luan "eval"-kutsun vastine `load` ei suorita lähdekoodia heti, vaan palauttaa sen käännettynä funktiona, koodia voidaan eristää muusta ohjelmasta vaihtamalla sen nimiavaruutta ennen funktion suorittamista.

Luan perustietotyypit taulujen lisäksi ovat *nil*, *boolean*, *number*, *string*, *function*, *userdata* ja *thread*. Kielen upotuksessa erityisen tärkeä on *userdata*, joka käärii muistiosoitteen joko Luan tai muun ohjelman varaamaan muistiin. Tämä mahdollistaa toisen kielen olioiden tuonnin Lua-virtuaalikoneen tilaan, ja koneen automaattisen roskienkeruun hallintaan.

Arvojen käyttäytymistä kielessä voidaan mukauttaa asettamalla niille *metatauluja* (engl. *metatable*), joiden jäsenet määrittävät toiminnan mm. jäsenarvon noutamisessa tai asettamisessa (`__index` , `__newindex`), aritmeettisissä operaatioissa (`__add` , ym.) ja roskienkeruussa (`__gc`). Metatauluihin asetettuja funktioita kutsutaan myös *metametodeiksi*. Vaikka Lua ei suoraan tarjoa kieliopissaan monia olio-ohjelmoinnin yleisiä käsitteitä, suuri osa niistä voidaan toteuttaa metatauluja käyttäen. Esimerkiksi luokka- tai prototyyppipohjainen (nk. delegaatioon perustuva) perintä voidaan toteuttaa viittaamalla "luokkaolioon" jäsenarvoihin liittyvissä metametodeissa. Tämä kuvaa hyvin kielen tavoitetta pitää sen ominaisuuksien määrä pienenä, mutta valikoida ominaisuudet siten, että ne ovat mahdollisimman monikäyttöisiä. Luan vakiokirjasto on melko pieni ja rajoittunut. Kieltä upottavan ohjelman odotetaan toteuttavan tarvittavat toiminnot itse.

3. Useimmat tulkit toteuttavat taulujen [1, 2, 3...]-avaimia käyttävän osan erikoistapauksena suorituskyvyn takia.

Pienenä myönnytyksenä olio-ohjelmoinnille kieliopissa on apuväline metodikutsuille: `olio:metodi(a)` on lyhyempi tapa ilmaista `olio.metodi(olio, a)`. Vastaavasti voidaan tehdä funktion määrittelyssä, jolloin sen parametrilistan alkuun lisätään "self"-muuttuja. Metodikutsun sekoittaminen itsenäisen funktion kutsuun on yleinen virhe. Toisaalta metodikäsitteen jättäminen kieliopin tasolle tähän tapaan on yksinkertaisempaa kuin esimerkiksi JavaScript-kielen sidotut funktiot ja *this*-sana.

Lua-kielestä on olemassa useita, suurimmaksi osaksi yhteensopivia toteutuksia. Tässä työssä käytettiin referenssitoteutuksen lisäksi *LuaJIT*-tulkkia. LuaJIT on itsenäinen Lua-toteutus, joka pyrkii parantamaan suorituskykyä "just-in-time"-kääntämisellä ja huolella optimoidulla, assembly-kielellä kirjoitetulla tavukooditulkilla. LuaJIT on käytännössä täysin yhteensopiva Lua 5.1:n kanssa ja usein paljon sitä nopeampi, saavuttaen monissa testiohjelmissa nopeimmat JavaScript-tulkit ja jopa Javan ja C#:n virtuaalikoneet. Tämän nopeuden ansiosta sen Lua-murretta on käytetty myös kääntäjien kohdekielenä. Tässä työssä sen suorituskykyetua rajoittaa kuitenkin kielen C-rajapinnan läpi tehtävien kutsujen määrä. LuaJIT-tulkki tarjoaa ulkoisen C-yhteensopivan ohjelmakirjaston kutsumiseen rajapinnan, FFI:n (Foreign Function Interface), joka ei kuitenkaan taivu helposti Luan upottavan C++-ohjelman kutsumiseen (Pall 2007). Muita toteutuksia ovat mm. pelikonsoleille tarkoitettu kaupallinen *Havok Script*, joka pyrkii rajoittamaan tulkin muistinkäyttöä ja *LuaJ*, joka pystyy kääntämään Lua-tavukoodia Java-virtuaalikoneen käyttämäksi tavukoodiksi.

3.3 Lua C++-ohjelmassa

Yksi Luan päätavoitteista on pitkään ollut helppo upotettavuus C:llä (tai yhteensopivalla kielellä) kirjoitettuun ohjelmaan. Kielen on tarkoitus pystyä sekä laajentamaan C:tä että tulemaan C:n laajennettavaksi: Lua-kielistä ohjelmaa voidaan kutsua C-ohjelmasta, ja C-ohjelmaa voidaan kutsua Lua-ohjelmasta. Tässä osiossa käsitellään lyhyesti pelikehyksessä käytetyn Lua- ja C++ -kielten sidonnan periaatteet.

Luan C-ohjelmointirajapinnan ydin on Lua-koneen pino, jolla kaikki tiedonvälitys

virtuaalikoneen tilaan ja siitä ulos toteutetaan. Rajapinnan kutsuilla voidaan luoda pinoon uusia tai hakea jo Luan nimiavaruudessa sijaitsevia arvoja sekä käyttää niitä kielen perusoperaatioihin. Funktioiden kutsuminen, taulujen indeksöinti ja muut operaatiot poistavat pinosta arvoja ja lisäävät paluuarvonsa pinoon. Kun Lua-ohjelma kutsuu C-kielistä funktiota, sille annetut parametrit nostetaan pinoon, josta funktio voi lukea ne. Arvoja palautetaan Lua-koneelle lisäämällä ne pinoon ja palauttamalla funktiosta paluuarvojen lukumäärä.

```
int lua_c_function(lua_State *L) {
    lua_pushnumber(L, 42); /* nosta pinoon luku 42  */
    return 1;              /* paluuarvojen määrä on 1 */
}

void registerAPI(lua_State *L) {
    /* nosta funktio pinoon */
    lua_pushcfunction(L, lua_c_function);
    /* ota pinon ylin arvo ja lisää se nimiavaruuteen */
    lua_setglobal(L, "get_answer");
}
```

Listaus 3.1. Esimerkki Luan C-rajapinnan käytöstä.

Pinon tilasta huolehtiminen on C-funktion vastuulla. Pinossa olevia arvoja voidaan lukea ulos, mutta toisin kuin monien muiden skriptikielten rajapinnoissa, Lua-tilan muistiin ei voida viitata suoraan: monimutkaisempaa tietoa voidaan käsitellä vain pinon läpi. Tämä toisaalta välttää ongelmia, joista mm. Python-kielen C-rajapinta kärsii: jotta Python-tulkki ei hävittäisi oliota, johon C-ohjelma viittaa vielä, ohjelman on päivitettävä käsin olion viitelaskurin arvoa. Viat viitelaskurien käsittelyssä ovat usein vaikeita havaita, koska niiden ilmeneminen riippuu kielen roskienkeruun käytöksestä (Li ja Tan 2014; Muhammad ja Ierusalimschy 2007). Viitelaskureita on myös yhtä monta kuin käsiteltäviä olioita, mutta Luan pinoja on vain yksi.

3.3.1 C++-tyyppi Lua-ohjelmassa

Jotta voisimme käsitellä C++-olioita Lua-ympäristössä turvallisesti, tarvitsemme menetelmän niiden elinkaaren hallintaan Lua-ympäristön sisältä. Tähän voidaan käyttää Luan *userdata*-tyyppiä. Userdata-olioita ei voi luoda kielessä suoraan, mutta niitä voidaan tuottaa virtuaalikoneen ohjelmointirajapinnan kutsuilla. Oliot voidaan jakaa kahteen tyyppiin: nk. kevyeen userdataan, joka käärii vain heikon viitteen, eli käytännössä raa'an muistiosoitteen, ja "täyteen" userdataan, joka viittaa Lua-koneen varaamaan ja hallitsemaan muistiin. Tähän muistiin voidaan luoda C++-olio varaamalla ensin sopivan kokoinen userdata ja konstruktoimalla olio sen sisälle C++:n "placement new"-toiminnon avulla:

```
T *obj = static_cast<T*>(lua_newuserdata(L, sizeof(T)));
new (obj) T;
```

Listaus 3.2. C++-olion luominen Luan muistissa.

"Täysi" userdata on Luan roskienkeruun hallinnassa, ja jos sillä on `__gc` -metametodi, metodia kutsutaan juuri ennen userdatan muistin vapauttamista. Jotta C++-olio purettaisiin oikein roskienkeruun kerätessä userdata-olion, roskienkeruun käsittelijäksi voidaan asettaa `lua_CFunction` -funktio [4], joka kutsuu eksplisiittisesti C++-olion destruktoria:

```
static int finalize(lua_State *L) {
    T *obj = static_cast<T*>(lua_touserdata(L, 1));
    obj->~T();
    return 0; // ei paluuarvoja
}
```

Listaus 3.3. C++-olion siivoaminen Luan roskienkeruussa.

Jotta metodia ei pystyisi poistamaan metataulusta, tällaisen olion koko metataulu voidaan kätkeä Lua-ohjelmalta. Asettamalla sen `__metatable` -kentäksi jokin muu arvo, esim. `false` , pääsy metatauluun Lua-kielestä estyy. Tässä on myös hyvä huomata, että ulkoisesti varatun muistin käsittely C- tai C++-ohjelmassa voi

4. typedef int (*lua_CFunction) (lua_State *L);

tuottaa ongelmia muistiosoitteiden kohdistuksen (engl. *alignment*) kanssa. Järjestelmäohjelmointikielen oma muistinvaraustoiminto tyypillisesti palauttaa muistiosoitteita, jotka täyttävät suorittimen korkeimman kohdistusvaatimuksen (esim. 16 tavua). Vaikka useimmat x86:n konekielen käskyt pystyvät toimimaan väärin kohdistettujen osoitteiden kanssa, mm. SSE-käskykantojen kanssa tästä voi seurata ongelmia. Tyypillinen oire on ohjelman epävakaus korkeammilla kääntäjäoptimoinnin tasoilla. Ongelman havaitsemiseksi tässä käytetty Lua-sidonta tarkistaa virtuaalikoneelta saatujen muistiosoitteiden kelpoisuuden vertaamalla niitä arvoon, jonka `std::alignment_of<T>` antaa tyypille.

3.3.2 Tyypityksen säilyttäminen

Kun Lua-ympäristöön tuodaan C++:n tyyppejä ja niitä käyttäviä funktioita, syntyy mahdollisuus virheeseen, jossa vääränlaista C++-oliota edustava userdata annetaan funktion parametriksi. C++:n ajonaikaisen tyyppitiedon [5] käyttöä on pitkään vältelty peleissä kyseisen ominaisuuden heikkojen toteutuksien ja niiden suorituskykyvaikutusten takia, mutta tässä se on hyvin käytännöllinen ratkaisu. Ajonaikaisen tiedon avulla voidaan estää tyyppivirhe esimerkiksi seuraavasti:

- Määritetään kaikille Lua-ympäristössä käytettäville C++-luokille yhteinen pohjaluokka (tässä *SOBase*).
- Tehdään pohjaluokasta *polymorfinen* esim. asettamalla sille virtuaalinen destruktori, jotta kaikki siitä johdetut luokat ovat myös polymorfisia. C++-kääntäjän on tuotettava tällaisen luokan olioihin tyyppitieto (esim. GCC:n *vtable*-viittaus).
- Userdata-olion tyypitöntä osoitinta käsitellessä suoritetaan ensin `static_cast` -muunnos osoitteeksi pohjaluokkaan. Tämä ei ole tyyppivirhe, jos kaikki Lua-ympäristöön luodut userdatat viittaavat pohjaluokan tai siihen perustuvan luokan olioon.
- Yritetään muuntaa pohjaluokan olion osoite `dynamic_cast` -muunnoksella odotetun tyyppiseksi osoitteeksi. Jos käsiteltävä olio ei ole tätä tyyppiä, muunnos epäonnistuu ja palauttaa `nullptr` :n.

5. RTTI, Run-Time Type Information

Jos Lua-ympäristössä käytettäviä luokkia koostetaan C++:n moniperinnällä, jokaisen niistä on ensimmäisenä perittävä pohjaluokka tai sen johdannainen; muuten `this` -osoitteet eivät välttämättä vastaa toisiaan pohjaluokan ja perityn luokan toteutuksissa. Tyyppivirheestä voidaan tiedottaa Lua-ohjelmaan kutsumalla `lua_error` -funktiota. Jos Lua-kirjasto on käännetty C++-kielisenä ja poikkeukset kytkettynä päälle, kutsu poistuu C++-funktiosta takaisin Lua-tulkkiin `throw` :lla, joka purkaa C++-oliot pinosta oikein. Voimme määritellä yleisen C++-mallin (engl. *template*) tälle.

```
template<class T>
static T* getInstanceOrNull(lua_State *L, int stackIndex = 1) {
    SOBase *ptr = static_cast<SOBase*>(lua_touserdata(L,
        stackIndex));
    return ptr ? dynamic_cast<T*>(ptr) : nullptr;
}

template<class T>
static T* getInstance(lua_State *L, int index = 1) {
    T* obj = getInstanceOrNull<T>(L, index);
    if(!obj) {
        luaL_error(L, "Value at %d not an instance of %s!",
                   index, T::getClassName());
    }
    return obj;
}
```

Listaus 3.4. Tyypityksen varmistaminen Lua-rajapinnassa.

3.3.3 Metodien ja jäsenmuuttujien sidonta

Jotta Lua-ympäristöön tuoduista C++-olioista olisi oikeasti hyötyä, niiden metodeja ja jäsenmuuttujia on pystyttävä tarjoamaan käytettäviksi Lua-kielessä. Luan C-rajapinnassa jokainen C-kielinen funktio ottaa parametrikseen käytetyn Lua-tilan (*lua_State**), jonka pinoon on nostettu funktiokutsun parametrit valmiiksi. Funktion odotetaan palauttavan kokonaisluvun, joka ilmoittaa montako arvoa pinosta poimitaan paluuarvoiksi. Omalle userdata-tyypille voidaan toteuttaa jäsenmuut-

tujia, metodeja ja tarvittaessa aritmeettiset perusoperaatiot sopivalla metataulul-la. Luokkaa rekisteröitäessä kerätään sen C-funktiona toteutetut toiminnot piilotet-tuun listaan, josta metataulun toteutus osaa löytää ne. Rajapinnan rakentaminen käsin on melko yksinkertaista mutta työlästä. Työn ja "boilerplate"-koodin mää-rää voidaan vähentää paljon metaohjelmoimalla esimerkiksi käyttäen C:n esikääntä-jää sekä C++11-standardin tarjoamia anonyymejä (lambda) funktioita. Käyttämällä edellisessä osiossa kuvailtua tyyppitarkastusta sekä pohjatyypeille toteutettuja apu-funktiomalleja `luaX_getvar<T>` ja `luaX_pushvar<T>` voimme kääriä C++-olion get/set-metodit Luan C-funktioon seuraavasti:

```
#define PROPERTY(T, GETTER, SETTER) (lua_CFunction)( \
[](lua_State *L_) -> int { \
    auto self = getInstance(L_); // määritelty luokassa \
    int top = lua_gettop(L_); \
    if(top > 1) { \
        self->SETTER(luaX_getvar<T>(L_, 2)); \
        lua_pop(L_, top - 1); \
    } else { \
        luaX_pushvar(L_, self->GETTER()); \
    } \
    return 1; // palauta tämä tai noudettu jäsenmuuttujan arvo \
})

// Staattinen funktio, jota kutsutaan rekisteröidessä luokkaa
   Lua-ympäristöön.
// Tuottaa taulun, josta haetaan jäsenmuuttujia kääriviä funktioita.
void Class::pushMemberTable(lua_State *L) {
    lua_newtable(L);
    luaX_pushvar(L, PROPERTY(float, getScale, setScale));
    lua_setfield(L, -2, "scale");
}
```

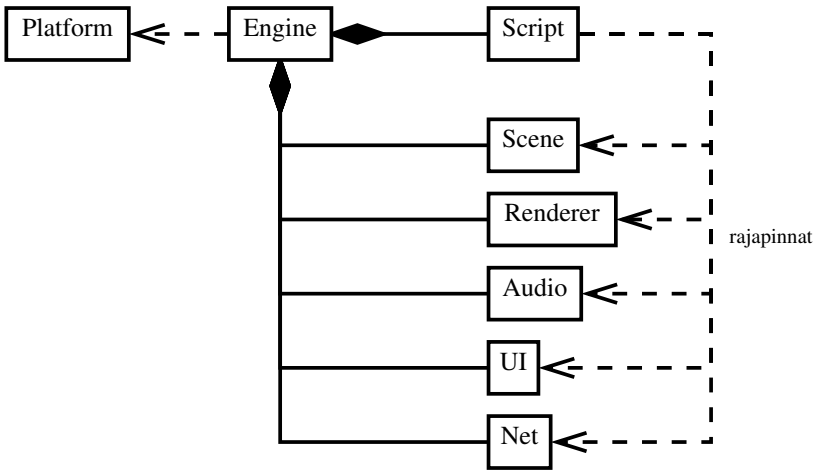
Listaus 3.5. Jäsenmuuttuja Lua-oliossa.

Oliota luodessa sille asetetaan metataulu, joka etsii `__index` ja `__newindex` -metodeissaan tästä "jäsentaulusta" sopivaa funktiota. Jos se löytyy taulusta, jäse-narvon noutaminen tai kirjoittaminen tuottaa kutsun taulun C-funktioon. Samaa

funktiota voidaan käyttää molempiin havaitsemalla kutsun tarkoitus pinoon nostettujen arvojen lukumäärästä.

4 Verkkopelikehyksen toteutus

Tässä luvussa kuvaillaan työssä kehitetty järjestelmä, joka on verkkopelien ketterään pelikehitykseen suunnattu *microframework*, eli kevyt ohjelmistoalusta tai kehys. Kehyksen perusta on toteutettu C++-kielellä, mutta suuri osa käyttöliittymästä, pelimekaniikasta ja sen verkkoviestityksestä on toteutettu upotetun Lua-koneen avulla. Pelimaailman matalan tason simulaatiota suoritetaan ohjelman C++-kielisessä osiossa ja Bullet-fysiikkakirjastossa (Coumans ym. 2013) sekä pelin palvelimella että asiakkailla. Tämän yläpuolella toimiva pelimekaniikka ja pelin käyttöliittymä on toteutettu ohjelman Lua-kielisessä osassa. Verkkopelissä suuri osa pelin verkkokoodista myös välittää viestejä serialisoitujen Lua-tietorakenteiden muodossa. Kehykselle luodaan pelejä lisäämällä simuloituun pelimaailmaan kappaleita, joihin rakennetaan pelimekaniikkaa asettamalla niille tapahtumankäsittelijöitä. Pelaajan syöte käsitellään myös käyttöliittymän skriptissä määritellyissä tapahtumankäsittelijöissä.



Kuvio 3. Pelikehyksen yleinen rakenne.

Kuviossa 3 on kuvattu kehyksen rakenne yleisellä tasolla. Tutkielman kannalta epäoleelliset toteutusyksityiskohdat, kuten käytetty OpenGL-grafiikkarajapinta ja resurssitiedostojen hallinta, on tässä jätetty pois. Kehyksessä *Engine* alustaa yksi kerrallaan perustoiminnot, kuten pelikentän (*Scene*), piirtäjän (*Renderer*), äänitoiminnot

(*Audio*), käyttöliittymäkirjaston (*UI*) ja verkkotoiminnot (*Net*). *Script*-luokka hallitsee Lua-tulkkia ja toteuttaa apuvälineitä Lua-skriptien hallintaan ja virheenkorjaukseen. Useimmat moottorin osat tarjoavat aiemmassa luvussa kuvattuun tapaan rajapintoja Lua-ympäristöön. Pelikenttäluokka huolehtii pelimaailman tietorakenteista ja fysiikkasimulaatiosta sekä rekisteröi skriptiympäristöön itsensä lisäksi pelin olioiden pohjaluokat.

```
void Engine::run() {
    updateClock();
    mNet->beginFrame();
    platform::dispatchEvents();    // paikalliset syötteet
    mNet->dispatchEvents();        // verkkosyöte

    // kutsu kierrostapahtumankäsittelijää
    callScriptHandlers("tick", dt);

    // kerää piirtolista kentästä nykyisestä kuvakulmasta
    mRenderer->prepareDraw(mScene, viewParams);
    mAudio->update(mScene, viewParams);

    // ajetaan piirron aikana peliä taustasäikeessä
    auto simulate = [&]() {
        // uusia askelia lasketaan kunnes nykyhetki on saavutettu
        mScene->simulateWithCallback([&](float simDt) {
            // kutsutaan kerran joka askeleella
            callScriptHandlers("sim", simDt);
        });
        mScene->replicatePhysics(mNet);
    };
    mSimWorkers->post(simulate);  // aloita taustasimulaatio

    mRenderer->executeDraw();     // piirrä kenttä
    mSimWorkers->wait();          // odota simulaation valmistumista

    mNet->endFrame();             // lähetä puskuroidut viestit
    mUI->draw();                  // piirrä käyttöliittymä
```

```
        mWindow->present();              // esitä piirretty kuva

        loadPendingResources();
    }
```

Listaus 4.1. Yksinkertaistettu pääsilmukka.

Listaus 4.1 esittää tiivistetyn version kehyksen "pääsilmukkana" toimivasta metodista. Joka kierroksella suoritetaan ensin yleinen tapahtumankäsittely. Tämän jälkeen kootaan lista kentän piirrettävästä sisällöstä. Lista ei suoraan viittaa kentän sisältöön, mutta sisältää juuri riittävästi tietoa sen piirtämiseen. Pelimaailman simulaatiota voidaan näin alkaa suorittamaan taustasäikeessä ilman, että samanaikainen piirto häiriintyy. OpenGL-kontekstia ei voida käytännössä kutsua useammasta säikeestä kerrallaan, joten vain ohjelman pääsäikeen annetaan käyttää sitä. Pelikenttä suorittaa suurimman osan fysiikkasimulaatiostaan Bullet-kirjastossa, joka pyrkii edistämään tilaansa tasapituisin askelein ja siten voi kutsua annettua simulaatioaskeleen käsittelijää kerran, useita kertoja tai ei ollenkaan [1]. Tämän kautta jokaisen fysiikka-askeleen edellä kutsutaan skriptiympäristössä rekisteröityä tapahtumankäsittelijää. Näin Lua-kielellä kirjoitettu pelimekaniikka voi edetä tasaista tahtia ilman, että piirtonopeus vaikuttaa peliin. Simulaation ja pelikentän piirron jälkeen piirretään käyttöliittymä, esitetään kuva ja ladataan odottavia resursseja tarpeen mukaan.

4.1 Pelimaailma ja Lua-rajapinta

Yksi työn päätavoitteista oli luoda kehykselle ohjelmointirajapinta, joka on kokeneemmalle kehittäjälle mahdollisimman helposti omaksuttavissa, ja kokemattomalle ainakin ulkoisesti selkeä. Lua-kielen yksinkertainen kielioppi tekee jo lähdekoodista melko luettavaa, mutta kokeneemman kehittäjän tapauksessa voimme myös pyrkiä hyötymään aiemmasta kokemuksesta. Kuten jo aiemmin todettiin, Lua muistuttaa toimintatavoiltaan pitkälti yleisessä käytössä olevaa JavaScript-kieltä. Koska suurella osalla kehittäjistä lienee ainakin hieman kokemusta kehityksestä tällä kie-

1. Bullet-kirjasto osaa automaattisesti kätkeä piirtotahdista poikkeavaa simulaatiotahtia antamalla kappaleiden sijainniksi väliarvoja.

lellä, pelimaailman rajapinnan suunnittelussa otettiin mallia suosituista JavaScript-kirjastoista. Sisällönluonnin helpottamiseksi kolmiulotteisen maailman koordinaatisto vastaa oletustilassaan Blender-mallinnustyökalua: Z-akseli on "ylös"-suunta ja Y-akseli osoittaa eteenpäin.

Pelimaailman perusolioluokka on *SceneObject*. Tämä sisältää perusominaisuudet, joita tarvitaan matalan tason fysiikkasimulaatiossa: mm. sijainti ja kierto, nopeus ja kiertonopeus, massa [2] ja muoto. Yksinkertaisten geometristen kappaleiden lisäksi kappaleen muotona voidaan käyttää myös grafiikassa käytettäviä malleja. Olioita voidaan liittää toisiinsa kiinteästi tai löysästi. Kiinteässä liitoksessa oliot muodostavat puumaisen hierarkian, jonka jäsenet eivät pysty liikkumaan fysiikkasimulaatiossa toisiinsa nähden. Tällaisille yhdistelmäkappaleille muodostetaan automaattisesti Bullet-simulaation sisälle vastaava *compound shape*-muoto. Löysä liitos säilyttää kappaleet pelimaailmassa ja luo kappaleiden välille rajoitteen, kuten nivelen tai jousen, joka käsitellään fysiikkasimulaatiossa.

Näiden ominaisuuksien lisäksi pelin olioille voidaan asettaa lehtimäisiä aliobjekteja, jotka määrittelevät niiden ulkoasun:

- *SceneMesh* antaa kappaleelle näkyvän muodon, kolmiulotteisen mallin.
- *SceneLight* toimii valonlähteenä pelimaailmassa.
- *SceneSound* tuottaa ääntä.
- *SceneEffect* on animoitu hiukkasefekti, kuten räjähdys tai savuvana.

Koska aliobjektit eivät ole suoraan mukana pelimaailman simulaatiossa, pelkästään palvelimena toimiva prosessi voi suurimmaksi osaksi ohittaa niiden käsittelyn. Tässä kehitetyssä verkkopelissä aliobjekteja ei myöskään replikoida automaattisesti; kappaleet huolehtivat näistä esitysyksityiskohdistaan metodeissa, jotka suoritetaan myös pelin asiakkailla. Näiden käyttämiin resursseihin viitataan Lua-ympäristössä suoraan tiedostonimillä. Jos samaa resurssia käytetään useammassa paikassa, tiedosto ladataan vain kerran. Koska raskaiden grafiikkaresurssien lataaminen kes-

2. Bullet-fysiikkamoottorin numeeristen rajoituksien takia sen fysiikkasimulaatioon annettavien arvojen "mittakaavaa" voidaan säätää kertoimella pelin luonteen mukaan.

ken pelin simulaation voisi aiheuttaa häiritsemää nykimistä ja vaikeuttaa piirron säikeistystä, resursseja hallitseva välimuistiluokka ei lue niitä heti. Uutta resurssia pyytäessä luokka luo ja palauttaa tyhjän resurssin, jota ei voida käyttää piirtämisessä. Väliaikainen sisältö korvataan oikealla resurssilla, kun sen lataus on valmis.

Kappaleiden pohjaluokkaa voidaan laajentaa Lua-kielellä määritetyllä "luokalla". Näin tuotettuja luokkia voidaan käyttää alkuperäisen pohjaluokan tapaan pelimaailmassa. Kuten aiemmin havaittiin, Lua-kieli itse tarjoaa keinoja perinnän toteuttamiseen, mutta ei juurikaan siihen erikoistunutta kielioppia tai valmiita kirjastotyökaluja. Kieltä käyttävä ohjelma voi toteuttaa oman luokka- ja perintämekanisminsa. Tässä käytetty tapa omien luokkien ja perinnän toteuttamiseen perustuu monissa JavaScript-kirjastoissa nähtyyn käytäntöön, joka tunnetaan myös nimellä *Simple JavaScript Inheritance*. Luokkaa laajennetaan kutsumalla sen `extend`-metodia parametrinä taulu alaluokan uusia ominaisuuksia. Tämä luo uuden luokkaolion, jonka käyttämä metataulu viittaa aiempaan luokkaan. Luokasta voidaan luoda pelikenttään soveltuva olio kutsumalla `new`-metodia, vapaaehtoisena parametrinä taulu oliolle heti asetettavia ominaisuuksia. Luokkaolion kutsuminen suoraan tuottaa myös kutsun tähän metodiin. Pienenä erona JavaScript-kirjastoista tunnettuun malliin alaluokille on annettava määrittelyhetkellä nimi pelin verkkoviestityksen, tallennusten ja bugien jäljittämisen käyttöön. Listauksessa 4.2 on lyhyt esimerkki uuden luokan lisäämisestä peliin.

```
-- Luo "SceneObject.MyClass" -luokka.
MyClass = SceneObject:extend('MyClass', {
    -- Luokan yleiset ominaisuudet.

    -- Konstruktori
    init = function(self, args)
        SceneObject.init(self, args) -- "super"-kutsu
        self.scale = args.scale or 2.0
        self.lamp = SceneLight { color = vec3(1, 0, 0) }
        self:attach(self.lamp)
        -- ...
    end,
```

```
    -- Kutsutaan olion koskettaessa toista.
    -- Argumentit ovat törmäävät oliot, kosketuspiste maailman
    -- koordinaateissa ja normaalivektori toisen olion pinnassa.
    onHit = function(self, other, location, normal)
        -- ...
    end,
})

-- Luo 'MyClass'-olio.
local object = MyClass()


-- Luo isompi 'MyClass'-olio.
local biggerObject = MyClass { scale = 4.0 }
```

Listaus 4.2. C++-pohjaluokan laajentaminen Lua-koodissa.

Pelimaailma näkyy Lua-ympäristössä *scene*-oliona. Sen kautta voidaan lisätä pelikenttään kappaleita ja hallita pelimaailman perusominaisuuksia, kuten painovoimaa ja ympäristön valaistusta:

```
scene.gravity = vec3(0, 0, -9.8)

local bigObject = MyClass { scale = 4.0 }
scene:add(bigObject)
```

Listaus 4.3. Pelimaailman hallinta Lua-koodissa.

Monissa tapauksissa olion (kuten hiukkas- tai ääniefektin) käytös sen luonnin jälkeen ei juuri tarvitse huomiota Lua-koodissa. Näille voidaan asettaa `disposable` -lippu, joka ilmoittaa, että olio voidaan poistaa automaattisesti, kun se koostuu vain päättyneistä efekteistä. Vain palvelimena toimivassa peliprosessissa nämä voidaan jättää lisäämättä kenttään, mutta niiden luonnista tiedotetaan asiakkaille.

Jotta pelimaailma käyttäytyisi mahdollisimman tasaisesti riippumatta tietokoneen suorituskyvystä, rajapintaan toteutettiin kaksi erillistä tapahtumaa moottorin "kierrokselle". Ensimmäinen näistä laukaistaan aina, kun moottorin "pääsilmukka" ajetaan ja peli piirtää kuvan ruudulle, ja toinen aina, kun Bullet-fysiikkakirjasto las-

kee uuden simulaatioaskeleen. Pääsilmukan tapahtumankäsittelijässä peli voi tutkia käyttäjän syötettä, siirtää pelin "kameraa" ja tehdä muita käyttöliittymään liittyviä toimenpiteitä, jotka pelin kannattaa toteuttaa mahdollisimman sulavasti. Pelkkänä palvelimena toimiessa tätä ei koskaan kutsuta. Jälkimmäistä pyritään kutsumaan moottorin piirtonopeudesta riippumatta tasaista tahtia, joka voidaan määrittää kentän fysiikkasimulaation asetuksissa. Suorittamalla pelimekaniikka pääsääntöisesti tämän tapahtuman yhteydessä peli voi toimia ennakoitavalla tavalla laitteiston suorituskyvystä riippumatta. Pelin olioiden tapahtumankäsittelijöitä kutsutaan fysiikkasimulaation askelten tahdissa: `onSim` -metodeja ennen jokaista fysiikka-askelta ja `onHit` -metodeja askelten jälkeen niin monta kertaa kuin kappale törmäsi. Törmäyksiin liittyvän tapahtumankäsittelyn lykkääminen askeleen loppuun yksinkertaistaa fysiikkasimulaation toteutusta ja sallii simulaation suorittamisen useassa säikeessä.

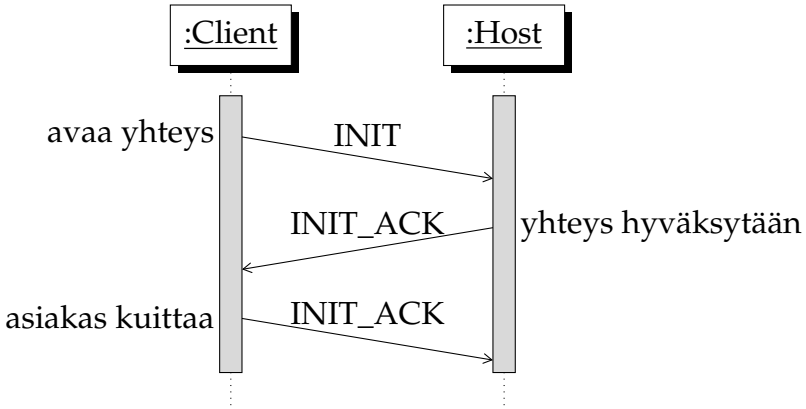
Pitäessä fysiikkasimulaation askelta vakiopituisena on sulavan liikkeen kannalta tärkeää, että pelin animaatio voi edetä siitä riippumattomalla nopeudella. Bullet-kirjasto pystyy tätä varten laskemaan väliarvoja peräkkäisten simulaatioaskeltensa tuloksille. Tästä seuraa se, että pelin kappaleilla on kaksi sijaintia: väliarvoista saatu sulavasti etenevä, mutta epätarkka tila, ja viimeisimmän lasketun fysiikka-askeleen tulos. Jotta tässäkään piirtonopeus ei vaikuttaisi pelimekaniikkaan, fysiikkasimulaatiota käyttävien olioiden sijainnin, nopeuden ja muiden ominaisuuksien lukeminen skriptissä palauttaa ne viimeisimmän fysiikka-askeleen tiedon mukaan ohittaen väliarvot. Kamera- ja käyttöliittymäkäyttöä varten luokkiin lisättiin erilliset vain luettavissa olevat jäsenmuuttujat, joista voidaan noutaa piirrossa käytetyt väliarvoversiot näistä.

4.2 Matalan tason verkkototeutus

Kehyksen verkkotoiminnallisuus toteutettiin luvun 2 havaintojen pohjalta. Syntynyt verkkototeutus noudattaa aiemmin käsiteltyä asiakas ja palvelin -mallia, jossa yksi pelaaja (tai itsenäinen palvelinprosessi) toimii palvelimena ja muut pelaajat liittyvät peliin ottamalla yhteyden palvelimeen.

4.2.1 Istunnonhallinta

Koska UDP-protokolla ei sisällä minkäänlaista tukea istunnonhallintaan, sovellusprotokollan tasolla on toteutettava jonkinlainen istuntojen seuranta. Tässä toteutetussa järjestelmässä istunto palvelimelle muodostetaan UDP-paketteina viestitettävällä kolmivaiheisella kättelyllä:



Kuvio 4. Yhteyden alustus.

- Asiakasohjelma lähettää palvelimelle INIT-paketin, joka sisältää peli/versiokohtaisen tunnisteen.
- Jos palvelimen pelitunniste täsmää ja se voi hyväksyä uuden yhteyden, se vastaa INIT_ACK-paketilla.
- Asiakasohjelma kuittaa vastauksen lähettämällä INIT_ACK-paketin takaisin.

Kättely muistuttaa TCP-protokollan menettelyä, jossa yhteyden avaavaan SYN-pakettiin vastataan SYN-ACK:lla ja vastaus kuitataan ACK-paketilla. Tähän perustuva palvelunestohyökkäys, jossa palvelimen resursseja kulutetaan tahallisesti aloittamalla SYN-kättelyjä on teoriassa mahdollinen myös pelikehyksen istuntoprotokollalla. Useamman yhteyden avaamisella samaan palvelimeen samasta lähteestä ei tässä kuitenkaan ole rehellistä käyttötarkoitusta, joten palvelin voi tarvittaessa rajoittaa keskeneräisten kättelyiden lukumäärää aggressiivisesti.

Kättelyn yhteydessä alustetaan arvio yhteyden viiveestä. Tätä päivitetään lähettämällä säännöllisin väliajoin PING-paketteja, johon sisältyy lähettäjän paikallinen ai-

kaleima. Tähän vastataan tähän PING_ACK-paketilla. Tieto viiveestä ohjaa kuittaamattomien pakettien uudelleenlähetysaikaa. Tämä toimii myös yhteyden valvontatoimintona, jos pelissä ei lyhyeen aikaan tapahdu mitään: Yhteyden oletetaan katkenneen, jos yhtään protokollan pakettia ei ole vastaanotettu 10 sekunnin aikana. Tämä katkaisee myös keskeneräiseksi jääneen kättely-yrityksen. Virhetilanteessa kumpi tahansa pää voi lähettää ERROR-paketin ja lopettaa yhteyden ylläpitämisen.

Muutokset yhteyden tilassa viestitetään Lua-ympäristöön Net-luokan *join-* ja *quit* -tapahtumankäsittelijöiden kautta. Näiden päälle voidaan rakentaa korkeamman tason toimintoja peliin liittymiseen ja siitä poistumiseen. Pelaajan liittyessä tapahtuva pelin tilan viestitys ja pelaajalistan muutoksen tiedottaminen toteutetaan näissä. Kun korkean tason yhteys on saatu, palvelin alustaa uudelle asiakkaalle tilan, johon täytetään pelaajan tiedot asiakkaan viestittämien tietojen perusteella. Vasta tämän jälkeen pelaajan hahmo voidaan lisätä pelimaailmaan. Peliin liittyminen voidaan vielä tässä vaiheessa torjua, jos palvelin katsoo sen tarpeelliseksi.

4.2.2 Tiedonvälitys ja luotettavuus

Matalalla tasolla protokolla toteuttaa istunnonhallinnan lisäksi kaksi viestien perustyyppiä: *luotettavat* ja *epäluotettavat* viestit. Luotettavien viestien toimitus pyritään takaamaan mahdollisimman hyvin protokollan tasolla siten, että viestin lähettäjä puskuroi ja uudelleenlähettää viestejä, kunnes vastaanottaja on kuitannut niiden saapumisen. Nämä luovutetaan protokollan käyttäjälle vain oikeassa järjestyksessä: Viestijono odottaa tarvittaessa välistä jääneen paketin uudelleenlähetystä ja saapumista. Näin taataan, että alustan päälle rakennettu verkkopeli toimii ainakin tärkeimmiltä osiltaan mahdollisimman järkevästi vaihtelevissa verkko-olosuhteissa. Epäluotettavien viestien tapauksessa järjestys- ja toimitustarkistukset sivuutetaan: paketti lähetetään mahdollisimman pian ja vain yhden kerran. Tämä on käytännöllistä pelin osissa, joissa tiettyä arvoa, kuten pistelaskuria, kappaleen sijaintia tai pelaajan syötteen tilaa päivitetään hyvin nopeasti eikä (todennäköisesti jo vanhentuneen tiedon) uudelleenlähettämisellä saavuteta käytännön hyötyä.

Viestityksen siirtonopeutta hallitaan yksinkertaisesti: sekä luotettavilla että epäluotettavilla pakettityypeillä on oma ylärajansa sekunnissa lähetettävälle tietomäärälle. Yhteyden laatua arvioidaan luotettavien pakettien uudelleenlähetystarpeen perusteella. Jotta huonosti toimiva verkkoyhteys ei tukkiutuisi täysin, lähetysnopeuden ylärajaa skaalataan pakettien toimituksen onnistumissuhteella (0..1). Koska kaikkien epäluotettavien viestien ei odoteta saapuvan perille, niiden lähetystä voidaan rajoittaa yksinkertaisesti jättämällä osa paketeista lähettämättä.

Nopeatempoinen peli pystyy helposti tuottamaan kymmeniä tai satoja yksittäisiä viestejä jokaisella simulaatioaskeleellaan. Jotta UDP-pakettien määrä ei aiheuttaisi ongelmia heikkotehoisten reitittimen ja helposti tukkiutuvien mobiiliyhteyksien kanssa, samaan protokollatason pakettiin pyritään yhdistämään useita viestejä. Molemmilla perusviestityypeillä on omat puskurinsa, joihin viestejä kerätään etukäteen asetettuun kokorajaan (tässä 1400 tavua) asti. Liiallisen viiveen välttämiseksi puskurit tyhjennetään viimeistään jokaisen kehyksen "kierroksen" lopussa. Tämä on käytännössä korkeamman tason toteutus joidenkin TCP/IP-pinojen tukemalle *TCP_CORK*-lipulle, jolla sovellus voi ohjata verkkopinon puskurointikäytöstä oman tilanteensa mukaan.

Tyypillisenä dynaamisena kielenä suuri osa Lua-kielen käsittelemästä tiedosta sisältää tekstiä merkkijonojen muodossa. Koska tekstisisältö on yleensä hyvin pakkautuvaa, pakettien puskurointiin lisättiin *zlib*-kirjastoa käyttävä pakkaustoiminto. Luotettavien viestien taatusta toimitusjärjestyksestä johtuen samaa zlib-tilaa voidaan käyttää niille koko istunnon ajan sekä lähettäjän että vastaanottajan päässä. Koska epäluotettavat viestit voivat jäädä saapumatta, niille omistettu pakkaustila on nollattava jokaisen viestin yhteydessä. Tämä ja epäluotettavien viestien sisältö (suureksi osaksi C++-koodista lähetettyä liukulukupainotteista fysiikkatilaa) jättivät pakkauksen hyödyn epäluotettavassa lähetyksessä vähäiseksi.

Pakkaus ja puskurointi yhdessä laskivat istunnossa lähetettävien pakettien määrää huomattavasti: Pelaajan liittyessä testipeliin ja vastaanottaessa tietoa pelikentän tilasta lähetettävien pakettien määrä ja tietoliikenne yhteensä laskivat alle kymmenesosaan alkuperäisestä. Tiedon välittäminen zlib-kirjaston läpi myös lisäsi käytän-

nöllisesti tarkistussummien laskennan pakettien sisällölle.

4.3 Replikaation toteutus

Kehyksen fysiikkasimulaation luonteen takia pelikentän replikaatiossa päädyttiin passiiviseen malliin, jossa pelimaailman auktoriteettina toimiva palvelin suorittaa täydellistä versiota pelin simulaatiosta ja viestittää sen tilaa pelin asiakkaille. Istunto- ja tiedonvälityskerroksen päälle rakennettua viestinvälitystä käytetään tässä kahdella tavalla. Koska suuri osa pelimekaniikasta toteutetaan Lua-koodissa, siitä tehtävät etäkutsut myös muodostavat valtaosan pelin verkkokoodista. Suorituskykysyistä joitakin pelimaailman matalan tason yksityiskohtia (kappaleiden fysiikkasimulaation tiloja) viestitetään kuitenkin ohjelman C++-kielisestä osiosta. Näiden lähetystahtia voidaan hallita kappalekohtaisesti Lua-ympäristöstä pelin tarpeiden mukaan.

4.3.1 Viestinvälitys Lua-kielessä

Jotta asiakkaan ja palvelimen Lua-ympäristöjen välillä voitaisiin viestiä tehokkaasti, verkkoistunnon kautta on pystyttävä välittämään monimuotoista tietoa. Viestien lähettäminen ja vastaanottaminen toteutettiin kehyksen `Net` -komponentissa. Kun sen lähetysmetodia kutsutaan skriptiympäristöstä, Lua-koneen pinoon nostetut arvot käydään läpi sen ulkopuolella C++-koodissa. Pinon sisältö serialisoidaan muotoon, joka voidaan lähettää matalan tason protokollan viestipaketissa. Vastaanottavassa päässä viestin sisältö puretaan paikallisen Lua-koneen pinoon ja pelin viestinkäsittelijöitä kutsutaan parametrinä tämä viesti. Koska Luan tieto on rakenteeltaan hyvin samanlaista kuin esimerkiksi JavaScript-kielen, valmiin viestikirjaston käyttö serialisoinnissa olisi varmasti ollut mahdollista. Luan tietotyypit eivät kuitenkaan sopineet täydellisesti moneen näistä ja ulkoinen kirjasto olisi lisännyt oman hankaluutensa projektin riippuvuuksien hallintaan, joten kehykselle kehitettiin oma viestiformaatti.

Tämän suunnittelussa painotettiin yksinkertaisuutta ja nopeaa käsittelyä. Viestien

lähetysmuodossa jokaisen Lua-arvon aluksi lähetetään yhden tavun pituinen tyyppitunniste. Arvot *nil*, *true* ja *false* koostuvat pelkästään tunnisteestaan. Numero lähetetään tunnisteena ja sen raakana liukulukuesityksenä. Merkkijono koodataan lähettämällä tunnisteen jälkeen sen pituus (tai riittävän lyhyen merkkijonon tapauksessa pelkkä pituus, jolloin se tulkitaan samalla tunnisteeksi) ja raaka merkkidata. Taulu koodataan lähettämällä ensin taulun tyyppitunniste ja sen jälkeen peräkkäisiä avain-arvo-pareja. Taulun lähetys päättyy *nil*-avaimeen, joka on Luan kannalta yhtä kuin puuttuva pari.

Tämä matalan tason viestiformaatti toimii yksinkertaisen Lua-tiedon lähetyksessä hyvin. Viestitettävään tietoon kuuluu kuitenkin yksinkertaisten tyyppien lisäksi myös Lua-tilassa määriteltyjä luokkia, kuten kolmiulotteisia vektoreita. Näiden lähettäminen pelkästään serialisoimalla oliotauluja hävittää tiedon olioiden luokista, eli viitteet niiden metatauluihin. Näiden säilyttämiseksi luokille voidaan määrittää `__serialize` -metodi, joka voi kaapata tiedon olion tyypistä lähetettävään viestiin. Vastaanottavassa päässä näin viestitetyt oliot voidaan tunnistaa ja rakentaa uudestaan oikean luokan olioiksi.

4.3.2 Etäkutsujen toteutus

Viestitoiminnon päälle rakennetut etäkutsut toimivat kokonaan Lua-ympäristössä. Peliin liittyminen tuottaa istunnon ensimmäiset Lua-tason etäkutsut, joissa palvelin ja asiakasohjelma jakavat keskenään pelaajan profiilitiedot ja pelimaailman tilan. Kutsuissa ensimmäinen lähetettävä parametri on tunniste, joka ilmoittaa kutsun tyypin; pelin koodissa voidaan määritellä omia kutsutyyppejä mm. pelimaailmalle ja sen jäsenille, pelaajien syötteille ja muille moninpelin perustoiminnoille, kuten pistetilastojen päivittämiselle.

Suurin osa pelin viestityksestä liittyy sen pelimaailman sisältöön. Kun olio lisätään palvelimen maailmaan, sille annetaan uniikki tunniste, `uid` , jolla palvelin voi nimetä sen verkkoviesteissä. Olion lisääminen tai poistaminen tuottaa myös palvelimella automaattisesti viestin siitä kaikille pelin osallistujille, jos viestitystä ei ole

erikseen kielletty kappaleen `notifyMode` -asetuksella [3] Jotta palvelimen ja asiakasohjelman listat pelimaailman nimetyistä kappaleista eivät pääsisi erkanemaan toisistaan, lisäys ja poisto tiedotetaan aina luotettavina viesteinä. Uuden kappaleen lisäämisen yhteydessä lähetettävä tieto on Lua-taulu, joka sisältää olion luokan tunnisteen ja kaikki sen serialisoitavat jäsenarvot lukuunottamatta kappaleisiin liitettyjä esitysominaisuuksia, jotka käsitellään vain asiakkailla. Asiakkaan päässä taulusta "kopiokonstruktoidaan" olio, joka voidaan lisätä paikalliseen kopioon pelimaailmasta. Koska taulun jäseniin kuuluu kappaleen tunniste, palvelin voi myöhemmin viitata tässä luotuun kopioon muissa etäkutsuissa.

Kappaleiden lisäämisen ja poistamisen lisäksi olisi hyödyllistä, jos pystyisimme automaattisesti tuottamaan etäkutsuja niiden metodikutsuista *Unreal*-pelimoottorin kielen tapaan. Tämä on selvästi mahdollista toteuttaa yllä kuvatuilla viestitoiminnoilla, mutta kehittäjän kannalta olisi toivottavaa löytää etäkutsujen määrittämiseen selkeä rajapinta. Tähän löytyi käyttökelpoinen malli web-kehyksistä: Monet web-kehykset, kuten *Django* ja *Plone*, toteuttavat mm. automaattisia tietoturvatarkistuksia ja tietokantansa päivitystoimenpiteitä nk. *wrapper*-funktioissa tai *dekoraattoriluokissa*. Nämä voivat dynaamisen kielen tarjoamien ensimmäisen luokan funktioiden avulla kääriä helposti funktion tai metodin toiseen funktioon, joka lisää sen ympärille ylimääräistä toiminnallisuutta. Näistä saatiin seuraavanlainen malli pelin luokkien etäkutsujen määrittelylle:

```
MyClass.bounce = function(self, direction)
    self:playSound('bounce.ogg')
    self:applyImpulse(direction)
end
-- aseta MyClass['bounce'] kutsuttavaksi asiakkaillakin
net.replicateObjectMethod(MyClass, 'bounce')
```

Listaus 4.4. Metodin replikointimenetelmän asettaminen.

Kutsu korvaa käärittävän metodin siten, että kun sitä kutsutaan palvelimella, siitä tuotetaan automaattisesti etäkutsu, joka saa tunnisteekseen replikointimenetelmän

3. Tämä asetus on hyödyllinen, jos palvelimen on tarve lisätä pelimaailmaan lyhytikäinen efekti, joka ei tarvitse lisää viestitystä.

ja metodin nimen, ja sisällökseen kappaleen `uid`-tunnisteen ja metodille annetut argumentit. Kun verkkopeli ei ole käytössä, käärityt metodit ohittavat verkkorajapinnan kutsumisen ja kutsuvat vain alkuperäistä metodia annetuilla parametreillään.

Vastaava automaattinen viestitys voidaan toteuttaa jäsenmuuttujille metatauluja käyttäen. Antamalla Lua-ympäristölle pääsyn tauluun, joka kuvaa jäsenmuuttujien nimiä C++:n luokan setteri- ja getterifunktioihin, voimme ylikirjoittaa käytöksen asettaessa muuttujien arvoja. Kun tämä tapahtuu kappaleen omistajalla, muutoksesta voidaan tuottaa automaattisesti viesti metodin etäkutsun tapaan. Näin esimerkiksi palvelin voi tiedottaa pelihahmon osumapisteiden muutoksista.

```
MyClass.init = function(self, args)
    SceneObject.init(self, args)
    self.health = 100
end
net.replicateObjectProperty(MyClass, 'health')
```

Listaus 4.5. Jäsenmuuttujan automaattinen replikointi.

Etäkutsujen vastaanotto päästää verkosta tulevaa monimuotoista tietoa peliprosessin muistiin, joten se on heti mahdollinen tietoturva-aukko peliä ajavassa järjestelmässä. Koska tällaisten turvallisuusongelmien hyödyntäminen ei ole kovin harvinaista verkkopeleissä (Yan ja Randell 2005), viestityksen tietoturvan testaamiseen rakennettiin erillinen testiympäristö. Tässä ympäristössä sitä pystyttiin *fuzz*-testaamaan *afl-fuzz*-työkalulla, joka jalostaa geneettisellä algoritmillä virheellisiä syötteitä. Tämän avulla toteutuksesta löytyi nopeasti yksi mahdollinen palvelunestohaavoittuvuus, jossa Lua-virtuaalikoneen pino saavutti maksimikokonsa ja kone joutui pysäyttämään vastaanottavan ohjelman suorituksen. Koska tämän lisäksi viestinvälityksestä ei löydetty muita matalan tason ongelmia, viestimuoto lienee ainakin saavuttanut yksinkertaisuustavoitteensa.

Luan numerot ovat aina liukulukuja, jotka voivat reaalilukujen approksimoinnin lisäksi esittää virhetiloja *epäluvuilla* (engl. *NaN, Not a Number*). Näillä erikoisarvoilla on hyvin vähän käyttötarkoituksia oikein toimivassa pelimekaniikassa. Jos viestin kautta pelin tilaan pääsisi yksikin NaN-arvo, se voisi levitä pelissä tapahtuvan arit-

metiikan kautta ja lopulta pysäyttää pelin. Jotta palvelimen pelimaailmaa ei voisi sotkea lähettämällä vahingossa (tai tahallisesti) "haitallisia" liukulukuja, viestinvälityksen toteutus yksinkertaisesti hylkää näitä sisältävät viestit.

4.3.3 Fysiikkasimulaation synkronointi

Pelikentän fysiikkasimulaatio tapahtuu suurimmaksi osaksi C++-kielellä toteutetussa Bullet-kirjastossa. Jokainen kirjaston laskema simulaatioaskel voi päivittää liikkuvien kappaleiden "fysiikkatilaa" eli niiden sijaintia, nopeutta, kiertoasentoa ja kulmanopeutta. Vaikka palvelimella suoritetun simulaation tuloksia voitaisiin myös välittää Lua-etäkutsuissa [4], tehokkuussyistä on käytännöllisempää välittää tämä tieto suoraan matalammalla tasolla.

Kehyksen `Scene`-luokassa seurataan palvelimen jokaisella simulaatioaskeleella *aktiivisia* eli fysiikkasimulaatiossa liikkuvia kappaleita. Askeleen jälkeen palvelin käy läpi listan näistä ja tutkii, mistä kappaleista on tarvetta lähettää tilapäivitysviestejä asiakkaille. Monimutkainen heuristiikka kappaleen tilapäivityksen tarpeen arviointiin osoittautui vähemmän tarpeelliseksi kuin odotettiin: Oikein säädettynä fysiikkamoottori osaa huolehtia pysähtyneiden kappaleiden *deaktivoinnista*. Tämä päättää myös niihin liittyvän viestityksen.

Koska pelin asiakkaat suorittavat karkeasti palvelinta vastaavaa simulaatiota, sulava liike voidaan saavuttaa lähettämättä päivityksiä jokaisen simulaatioaskeleen yhteydessä. Riittääkin, kun päivitystahti estää näkyvät poikkeamat palvelimen ja asiakkaan fysiikkasimulaatioiden välillä. Palvelimen ja asiakkaan tilojen erojen kätkemiseksi asiakas ei muuta suoraan kappaleen sijaintia vastaanottaessaan hieman poikkeavan päivityksen, vaan säätää kappaleen nopeutta siten, että se pyrkii lähestymään palvelimen tiedottamaa sijaintia. Tämän "pehmeän korjauksen" vahvuus on suhteellinen poikkeaman pituuteen. Näin pelaajan ohjaama kappale voi pysyä pelaajan näkökulmasta hieman edellä palvelimen viestittämästä liikkeestä, mutta silti mukailla tilaansa palvelimella. Riittävän suuren poikkeaman tapauksessa kap-

4. Fysiikkareplikaation ensimmäinen toteutus jopa toimi näin.

pale siirretään palvelimen ilmoittamaan sijaintiin ilman yritystä kätkeä toimintaa.

Koska asiakkaiden fysiikkasimulaatioilla ei ole minkäänlaista "määräysvaltaa" pelin etenemisen kannalta, niiden paikallisia simulaatioita ei tarvitse suorittaa palvelimen tarkkuudella. Jos asiakkaan suorituskyky ei riitä, fysiikkasimulaation askelpituutta voidaan muuttaa. Koska asiakkaiden fysiikkasimulaation tilan ei odotetakaan olevan täysin sama kuin palvelimen, päivitykset voidaan toimittaa epäluotettavalla viestinvälityksellä. Näin pelin verkkoliikenne myös pystyy sopeutumaan hieman verkkoyhteyden laadun muutoksiin.

Kun asiakkaan tilan ei muutenkaan odoteta vastaavan täydellisesti palvelinta, verkkoyhteyttä voidaan säästää laskemalla tilapäivitysten tarkkuutta. Fysiikkasimulaation sisällä on usein hyödyllistä säilyttää lukuja korkealla tarkkuudella epätarkkuuksien kasautumisen välttämiseksi. Kun verkon ylitse lähetettävien päivitysten tarkoitus on vain sallia palvelimen käytöksen suurpiirteinen jäljittely asiakkaalla, emme tarvitse tällaista tarkkuutta. Kappaleiden sijaintien lähettäminen korkealla tarkkuudella on hyödyllistä suurten pelikenttien varalta, mutta selvästi niiden nopeudet eivät tule tarvitsemaan edes 32-bittisten liukulukujen täyttä tarkkuutta. Myös kappaleen kiertoasentoa kuvaava kvaternio lienee turha viestittää neljänä liukulukuna.

Tilan säästämiseksi kappaleen kiertoasentoa kuvaava kvaternio Q koodataan neljän 32-bittisen liukuluvun sijaan kolmena 16-bittisenä kokonaislukuna olettaen, että $|Q| = 1$ ja sen kaikki komponentit ovat välillä $[-1...1]$. Yhden kokonaisluvun alin bitti käytetään puuttuvan komponentin etumerkin välittämiseen. Kun nopeuksia käytetään vain paikalliseen ennustamiseen, niiden lähettäminen täydellä tarkkuudella on myös tarpeetonta. Käytetty fysiikkamoottori tuskin pystyy ratkaisemaan liikettä kovin tarkasti edes palvelimella, jos kappale pyörii satoja tai tuhansia kierroksia sekunnissa. Nopeuksien kasvaessa myös asiakkaan ennusteiden epätarkkuus kasvaa. Voimme säilyttää sopivasti tarkkuutta nopeuksissa esimerkiksi koodaamalla vektorien komponentit logaritmisesti:

```
int16_t logEncodeFloat(float v, float scale) {
    v = max(v, -scale);
```

```
    return std::copysign(logx(min(scale, std::abs(v) + 1), scale),
        v) * INT16_MAX;
}

float logDecodeFloat(int16_t intval, float scale) {
    float fval = ((float)abs(intval)) / INT16_MAX;
    return (pow(scale, fval) - 1) * (intval < 0 ? -1.0f : 1.0f);
}
```

Listaus 4.6. Nopeusvektoreiden koodaus replikaatiossa.

Tämä varaa suurimman osan kokonaisluvun arvojoukosta pienempien arvojen esittämiseen, mutta sallii silti suurien nopeuksien esittämisen valittuun skaalausarvoon asti [5]. Skaalausarvot löydettiin mittaamalla kehyksen fysiikan käyttäytymistä lyhyen aikaa ottaen huomioon pelien mahdolliset tarpeet sekä fysiikkamoottorin tarkkuuden, vaimennuskertoimet ym. tekijät. Lineaarisen nopeuden tapauksessa päädyttiin arvoon 10000 ja kulmanopeuden tapauksessa 200.

nimi	**tyyppi**	**sisältö**
type	uint8	viestin tyyppitunniste
uid	uint32	kappaleen tunniste
time	uint32	tilan aikaleima
flags	uint8	fysiikkatilan liput
pos	3x float	sijainti pelikentässä
vel	3x int16	pakattu absoluuttinen nopeus
angle	3x int16	pakattu kvaternio
avel	3x int16	pakattu Euler-kulmanopeus

Taulukko 2. Fysiikkaviestien muoto.

Taulukossa 2 on kuvattu muoto, jossa kappaleiden fysiikkatila lähetetään. Nämä muutokset pienensivät yksittäisen fysiikkaviestin kokoa 74 tavusta 40 tavuun. Pienemmät viestit sallivat nopeamman lähetystahdin kappaleille, jotka sitä tarvitsevat.

5. Vaihtoehto tälle voisi olla raaka 16-bittinen liukuluku, mutta monet suoritinarkkitehtuurit eivät tue sitä suoraan. Tässä ei myöskään ole tarvetta ilmaista hyvin pieniä lukuja tarkasti.

Tässä kuvattu paikallinen, mutta karkeasti palvelimen kuuluttamaa totuutta mukaileva fysiikkasimulaatio on eräänlainen aiemmassa luvussa kuvatun *client-side prediction*-menetelmän toteutus.

4.4 Käyttöliittymä ja pelaajan syöte

Peliin voidaan rakentaa yksinkertaisia käyttöliittymiä C++-pohjaisella kirjastolla, jonka luokat tarjotaan Lua-ympäristössä samalla tavalla kuin pelimaailmankin. Näillä voidaan asetella ruudulle kuvia ja tekstiä sekä käsitellä hiiren ja näppäimistön syötteitä. Pelaajien syöte pelissä toteutetaan myös tämän avulla.

Kun käyttöliittymä vastaanottaa pelaajan ohjaukseen sidottuja syötteitä, ne välitetään `Player`-luokan olioon, joka toimii rajapintana pelihahmon ohjaukselle. Yksinpelissä tästä voidaan hallita suoraan pelihahmoa. Verkkopeliin liittyessä yhteyden molemmissa päissä luodaan pelaajalle oma `Player`-olio. Pelaajan syöte viestitetään asiakkaalta palvelimelle käärimällä tämän luokan metodit siten, että kun niitä kutsutaan asiakkaalla, palvelimelle lähetetään automaattisesti etäkutsuja. Vastaavasti palvelin voi kutsua asiakkaan `Player`-oliota esimerkiksi ohjattavan hahmon valitsemiseksi. Näin verkkopelin syötteet voidaan toteuttaa ilman muita muutoksia pelaajaluokkaan. Listauksessa 4.7 on lyhyt esimerkki syötteen toteuttamisesta pelaajalle.

```
function Player:chat(message)
    -- Tuota chat-viesti kaikille pelaajille.
    -- 'chat'-viestit käsitellään käyttöliittymässä.
    net:call('chat', message, self:getName())
end
-- Käärі Player-luokan 'chat'-metodi siten, että
-- sen kutsu käsitellään palvelimen 'player'-oliossa.
-- Palvelimen viestinkäsittelyssä 'player' on aina
-- viestin lähettänyt pelaaja.
net.wrapClientServer(Player, 'player', 'chat')
```

Listaus 4.7. Uuden syötteen lisääminen pelaajalle.

Tietyt viiveelle herkät syötteet (esim. pelihahmon liikuttaminen) kääritään omalla funktiollaan, joka antaa niiden suorittua välittömästi myös asiakaspäässä odottamatta palvelimen vastausta. Ero pelaajan asiakaspäässä ennakoidun ja palvelimen viestittämän todellisen liikkeen välillä on harvoin suuri. Tämä on hieman korkeamman tason vastine aiemmin esitellylle fysiikkasimulaation *client-side prediction* -toteutukselle. Liitteessä A on kattavampi esimerkki pelaajaluokan toteutuksesta.

5 Kehyksen arviointi

Pelikehyksen toimivuuden tutkimiseksi sen päälle kehitettiin pieniä verkkopelikokeiluja. Tässä luvussa tarkastellaan niiden toimivuutta ohjelmistoarkkitehtuurin kannalta sekä testaamalla niitä synteettisesti ja käytännön verkkotilanteissa.

5.1 Testipelit

Kehitysprosessin yksinkertaisimpien kokeilujen lisäksi kehyksen päälle kehitettiin kaksi mainittavaa testipeliä. Nämä olivat "räiskintäpeli", joka pyrki toteuttamaan yksinkertaistetun mallin perinteisestä 3D-räiskinnästä ja "fysiikkapeli", jossa pelaajat rakentavat irtonaisista komponenteista taisteluajoneuvoja ja ohjaavat niitä pelimaailmassa. Näistä ensimmäinen vaikuttaa yksinkertaiselta toteuttaa, mutta pelin nopeus ja tarve tarkalle ohjaustuntumalle asettavat omat haasteensa kehyksen toimivuudelle. Jälkimmäinen testipeli on hitaampi, mutta huomattavasti monimutkaisempi ja vaatii verkkototeutukselta kykyä viestittää sekä monimuotoisia syötteitä että hankalia fysiikkasimulaation tilanteita tehokkaasti.

5.1.1 Räiskintäpeli

Ensimmäinen pelikokeilu kehyksellä oli alkeellinen toimintapeli, joka uudelleenkäytti luovasti testigrafiikkaa ym. resursseja nopeassa räiskinnässä. Mallia otettiin klassisista ensimmäisen persoonan räiskintäpeleistä: Pelaajat liikkuvat sokkeloisessa kolmiulotteisessa kentässä ja metsästävät toisiaan sci-fi-henkisillä aseilla.

"Räiskintäpeliä" ohjataan hiirellä ja näppäimistöllä. Peliin tuotettavat syötteet ovat yksinkertaisia: pelaaja voi katsella ympäriinsä, pyrkiä liikkumaan vaakatasossa, hypätä, ampua ja vaihtaa asetta. Esineiden poimiminen tapahtuu automaattisesti koskettamalla niitä. Aiemmin todettiin hahmon tarkan ohjaustuntuman olevan hyvin tärkeä ensimmäisen persoonan näkökulmasta pelatessa. Tämän takia hahmon liikkeeseen ja katselusuuntaan liittyvät syötteet käärittiin verkkoviestitykseen siten,

että ne suoritetaan asiakkaalla välittömästi. Tämä kätkee suurimman osan näihin liittyvästä viiveestä. Ampuminen ja aseen vaihtaminen tuottavat myös etäkutsun, mutta näitä ei suoriteta välittömästi asiakkaan päässä; vain palvelin saa lisätä ammuksia pelikenttään tai hyväksyä aseen vaihtamisen.

Pelihahmon liikkeen toteuttaminen olemassaolevan fysiikkamoottorin ehdoilla lisäsi työhön oman haasteensa. Monet fysiikkapainotteisetkin pelit näyttävät toteuttavan jalan liikkuvien hahmojen liikefysiikan erikoistapauksina, jotka eivät täysin noudata fysiikan lakeja [1]. Jonkinlainen kompromissi tähän löydettiin toteuttamalla liike siten, että pelihahmon päivitysmetodi tuottaa voiman, joka pyrkii pitämään ohjattavan hahmon nopeuden suhteessa allaolevaan kappaleeseen uusimman liikesyötteen mukaisena. Hahmon liikkuessa tähän kappaleeseen kohdistetaan käänteinen voima. Pelihahmon toteutus esitetään tarkemmin liitteessä B.

Verkkopelin testausta helpottamaan pelille kehitettiin "botti" eli alkeellinen tekoäly, joka osaa tuottaa syötettä peliin. Botteja ajetaan pelin palvelimella ja ne lisäävät peliin omat pelaajaobjektinsa, joiden kautta ne voivat hallita pelihahmojaan. Yksinkertaiset botit eivät osaa juuri navigoida kentässä, mutta pystyvät vaeltamaan ja ampumaan satunnaisesti. Seuraamalla bottien liikettä oli helppo havaita puutteita asiakaspään liikkeen ennustamisessa.

5.1.2 Fysiikkapeli

Fysiikkapeli oli pelikokeiluista jälkimmäinen ja huomattavasti edellistä monimutkaisempi. Peliin liittyessään pelaajaa pyydetään valitsemaan ajoneuvoalusta, jolla pelaaja aloittaa pelissä. Pohjaksi voi valita yhden useasta saatavilla olevasta ajoneuvon rungosta (lentokone, ilmatyynyalus, helikopteri). Runkoon voidaan tämän jälkeen kiinnittää lisäosia, kuten siipiä, moottoreita ja erilaisia aseita. Osat maksavat pisteitä, joita voi ansaita etenemällä pelissä. Rungot ja niihin liitettävät osat toteutettiin kokonaan Lua-koodissa, mukaanlukien siipien lentodynamiikka. Vielä kehittyneempää rakentelutoimintoa suunniteltiin, mutta se jätettiin toteuttamatta, koska

1. Tyypillinen esimerkki tästä on hahmo, joka kykenee työntämään jalansijaansa.

hyvän käyttöliittymän kehittäminen toiminnoille olisi vaatinut paljon suunnittelutyötä.

Kuvio 5. Yhdestätoista osasta koottu lentokone fysiikkapelissä.

Pelaajat voivat ansaita pisteitä tuhoamalla merkittyjä kohteita ja muiden pelaajien rakentamia ajoneuvoja. Ajoneuvojen vahingonmallinnus perustuu kehyksen fysiikkasimulaatioon. Osien välisiä liitoksia seurataan niiden tuottamien voimien kautta; liikaa rasittuva liitos voi katketa. Osat voivat myös vaurioitua törmäyksissä ja saatuaan osumia ammuksista. Jos oma ajoneuvo jää jumiin pelikenttään, pelaaja voi itse tuhota sen. Aiemmin valittua laitetta voidaan vaihtaa ennen uuden lähettämistä peliin.

Ajoneuvot toteutettiin kehykseen laajentamalla ensin sen `SceneObject` -pohjaluokka `VehiclePart` -luokaksi. Tässä luokassa toteutettiin ajoneuvojen osien perusominaisuudet, kuten kestävyys-, hinta- ja pistetiedot, sekä perusmetodit kuten osien liittäminen toisiinsa, niiden ohjaus ja tuhoutuminen liiallisen vahingoittumisen seurauksena. Yksittäiset osat toteutettiin laajentamalla tätä luokkaa ja rekisteröimällä ne yleistä pelimekaniikkaa hallitsevaan `Game` -olioon, jotta käyttöliittymä pystyisi luettelemaan ne. Osat liitetään toisiinsa `StiffConstraint` -liitoksilla, jotka pyrkivät pitämään kappaleet samassa asennossa toisiinsa nähden.

Jotta pelin tahti ja taistelujen etäisyydet pysyisivät kohtuullisina pelattavuuden kannalta, todellisen maailman fysikaalisista ominaisuuksista joustettiin usealla taval-

la. Yhtenä esimerkkinä pelimaailman ilmakehän tiheys asetettiin hyvin korkeaksi [2]. Tämä rajoitti lentävien ajoneuvojen ja ammuksien nopeutta luonnollisella tavalla, ja teki erilaisten lentokoneiden käsittelystä huomattavasti helpompaa. Koska Bullet-kirjaston fysiikkasimulaatio toimii huomattavasti vakaammin, kun kappaleiden massat ovat pieniä, simulaation skaalauskertoimeksi asetettiin 1/100, eli 100 kg kappaleen massaksi annetaan Bullet-simulaatiossa 1. Vastaava säätö tehdään kertoimen mukaan kappaleisiin aiheutetuille voimille ja impulsseille.

Fysiikkapelin verkkototeutus rakennettiin räiskintäpelissä käyttökelpoisiksi havaittujen ratkaisujen päälle. Luonteeltaan jatkuvat tapahtumat, kuten pelaajien ohjainsyöte ja pelimaailman kappaleiden liike viestitetään käyttäen protokollan häviöllistä viestikanavaa. Ajoneuvojen valinta ja muokkaus moninpelissä toteutettiin käärimällä verkkopeliin vaikuttavat pelaaja- ja ajoneuvoluokan metodit siten, että niiden kutsuminen verkkopelissä viestittyy etäkutsuna palvelimelle.

Moniosaisen ajoneuvon rakentaminen pienemmistä osista paljasti nopeasti erikoisen ongelman fysiikkatilan replikoinnissa. Sen alkuperäinen toteutus lähetti kappaleiden tilapäivityksiä huomioimatta niiden riippuvuuksia toistensa tilasta. Kun asiakas vastaanotti uuden tilan vain osalle samaan ajoneuvoon kytketyistä kappaleista, sen paikallinen fysiikkasimulaatio vääristyi näkyvästi fysiikkamoottorin pyrkiessä korjaamaan kappaleiden erkanemista toisistaan. Fysiikkasimulaation replikaatiota muutettiin tämän takia siten, että toisiinsa kytketyt kappaleet pyritään päivittämään yhtäaikaisesti.

5.2 Verkkopelin toimivuus

Kehyksen empiiristä arviointia varten verkkototeutuksen eri tasoille toteutettiin mittareita tarkkailemaan lähetettyä ja vastaanotettua tietomäärää, viestien lukumäärää ja tiedonvälityksen luotettavuutta. Jotta pelin toimivuutta hankalissa verkko-olosuhteissa voitiin testata automaattisesti, verkkoprotokollan matalalle tasolle toteutettiin keinotekoinen viive ja pakettihäviö. Pelikehykseen lisättiin myös “päätön

2. Noin $12kg/m^3$

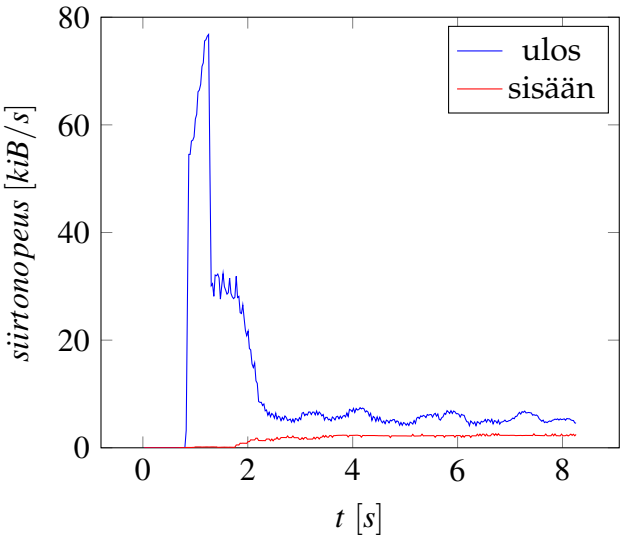
asiakas” -toiminto, joka salli asiakkaan ajamisen palvelimen tapaan komentoriviltä ilman grafiikkaa, ääntä tai käyttöliittymää. Näin oli mahdollista tuottaa useita kymmeniä “pelaajia” kuormittamaan palvelinta ja verkkoyhteyttä. Palvelimen kuorman tutkiminen näillä keinoilla viittasi siihen, että pelin fysiikkasimulaatio vaati paljon enemmän prosessoriaikaa kuin verkkototeutus.

Aiemmassa luvussa mainittu protokollatason toiminto pakettien yhdistämiseen ja pakkaamiseen lisättiin pian ensimmäisten testien jälkeen. Vaikka pelin aikana lähetettävien pakettien (ja tiedon) määrä ei ollut erityisen korkea, peliin liittyessä pelimaailman tilanteen viestittäminen tuotti useita tuhansia yksittäisiä viestejä, joiden lähettäminen yksittäisinä paketteina kuormitti etenkin mobiiliverkkoyhteyttä liikaa. Koska viestit lähetettiin heti yhteyden avaamisen jälkeen, protokollatasolla ei ollut vielä tässä vaiheessa tilaisuutta arvioida yhteyden todellista nopeutta ja säätää lähetystahtia sopivaksi.

Testit ajettiin 100 millisekunnin simuloidulla viiveellä Linux-alustalla. Käytännössä viive vaihteli välillä 100 – 120 ms laitteiston suorituskyvystä ja käyttöjärjestelmän aikataulutuksen tarkkuudesta riippuen. Kuvaajat tuotettiin keräämällä pelin tulostamaa mittaustietoa bash-skriptillä, joka samalla muutti tulosteen käytetyn kaaviotyökalun kanssa yhteensopivaan muotoon.

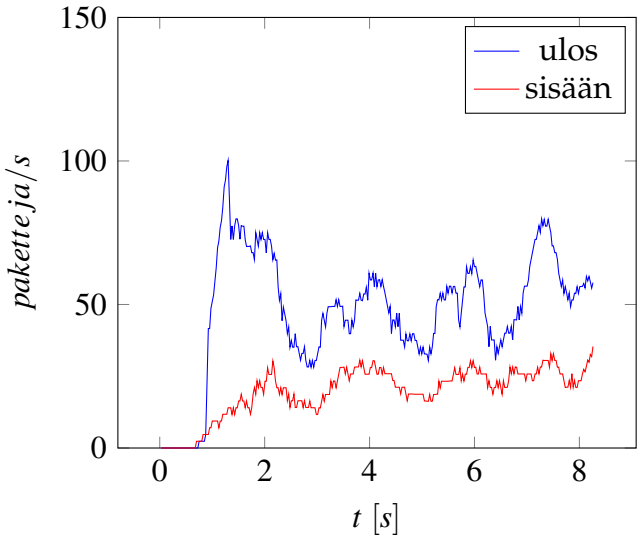
5.2.1 Räiskintäpelin testaus

Kuviossa 6 esitetään räiskintäpelin palvelimen mitattua verkkoliikennettä, kun yksi pelaaja liittyy palvelimelle, jossa liikkuu neljä palvelimen ohjaamaa “bottia”. Koska pelikentän rakenne lähetetään kokonaan palvelimelta, pelaajan liittymisestä seuraa hetkellinen piikki tiedonsiirrossa. Tämän jälkeen “räiskintäpeli” tuottaa hyvin vähän verkkoliikennettä.



Kuvio 6. Palvelimen tiedonsiirto räiskintäpelissä.

Lähetettyjen ja vastaanotettujen pakettien määrä vaihteli siirretyn tiedon määrää vähemmän. Tämä seuraa verkkototeutuksen tavasta yhdistää mahdollisuuksien mukaan samalla moottorin “kierroksella” lähetettyjä paketteja. Kun pelikenttää lähettäessä lähetysjonoon asetetaan paljon viestejä kerralla, samaan pakettiin voidaan helposti yhdistää useita viestejä. Tämän jälkeen viestejä lähetetään yhä lähes joka kierroksella, mutta lähetettävän tiedon määrä on pienempi; viestejä ei voida yhdistää aiheuttamatta ylimääräistä viivettä. Pakettien lähetystahti kuitenkin vaikuttaa riittävän matalalta jopa mobiilikäyttöön.



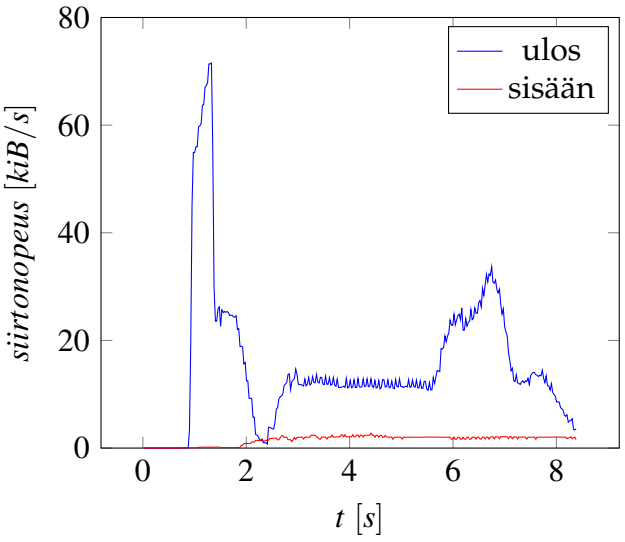
Kuvio 7. Pakettien lähetystiheys räiskintäpelissä.

Yritys pitää pelihahmojen liike mahdollisimman fysikaalisena ei ollut täysin ongelmaton. Pelaajien herkkyys viiveelle ja muille verkkopelien syöteongelmille riippuu vahvasti siitä, miten "henkilökohtaista" pelaajan ohjaus on. Ensimmäisen persoonan kuvakulmaa käyttävässä nopeassa pelissä ohjauksen on toimittava hyvin tarkasti. Kun näin simuloitu hahmo on jatkuvassa kosketuksessa ympäristöönsä, pelkkä tieto kappaleen viimeisimmästä sijainnista ja liikevektorista ei riittänyt uskottavaan liikkeen ennustamiseen muiden pelaajien tapauksessa. Tämä ratkaistiin ottamalla mallia *Doom 3*:n verkkopelin toteutuksesta ja tiedottamalla pelihahmoille annetusta liikesyötteestä pelin kaikille osallistujille. Tämän tiedon avulla asiakkaat pystyivät ennustamaan useimmissa tilanteissa liikettä riittävän tasaisesti. Koska lisätietoa syötteestä ei voitu helposti koodata yhtä tehokkaasti kuin kappaleiden matalan tason liiketilaa, sen lähettäminen nosti pelin tiedonsiirtovaatimuksia hieman. Havaittu parannus muiden pelaajien liikkeen tarkkuudessa näytti olevan tämän arvoista.

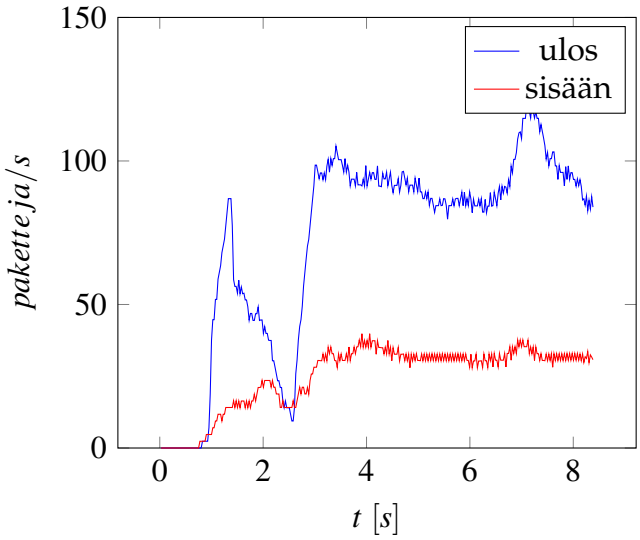
5.2.2 Fysiikkapelin testaus

Fysiikkapeli tuotti monimutkaisuutensa takia huomattavasti enemmän verkkoliikennettä. Tästä suurin osa koostui fysiikkasimulaation replikaatiosta, mutta ajoneuvojen luonti ja toiminta pelimaailmassa kasvattivat myös etäkutsujen määrää yksinkertaisempaan peliin verrattuna. Pelin kaistavaatimukset pysyivät silti jopa mobiiliverkkojen rajoissa. Verkkoviiveen vaikutus pelattavuuteen riippui vahvasti käsiteltävän ajoneuvon suorituskyvystä: hyvin nopea ajoneuvo oli vaikea hallita syötteen kärsiessä viiveestä. Kuvioissa 8 ja 9 on kuvattu palvelimelta mitattu tiedonsiirto, kun yksi pelaaja liittyy peliin, odottaa kentän latautumista, luo ajoneuvon ja antaa sen ajautua päin estettä. Kentän latautumisen jälkeinen piikki tiedonsiirrossa johtuu törmäyksestä, jossa ajoneuvon osat irtoavat toisistaan tai tuhoutuvat, käynnistäen samalla fysiikkasimulaation osalle kentän muista kappaleista.

Koska asiakas saa käyttöönsä ajoneuvon vain, kun pelikenttä on vastaanotettu, tätä ennen palvelin saa asiakkaalta vain kuittauksia pelikentän lähetyksestä. Yhteyden epäluotettavalla kanavalla lähetetyt pelaajan syötetilat käyttävät tämän jälkeen noin 2 kilotavua sekunnissa.



Kuvio 8. Palvelimen tiedonsiirto fysiikkapelissä.



Kuvio 9. Pakettien lähetystiheys fysiikkapelissä.

Pelejä testattiin näiden mittausten lisäksi lyhyesti käytännössä "3.5G"-mobiiliverkon ylitse. Keinotekoinen yhteyden laadun simulointi otettiin tässä pois käytöstä. Molemmat pelit osoittautuivat pelattaviksi, mutta korkeampi (ja rajusti vaihteleva) viive aiheutti havaittavaa häiriötä molemmissa. Parhaimmillaan yhteys oli pelin mittaamana viiveeltään noin 50 millisekuntia ilman mainittavaa pakettien häviötä; pahin mitattu viive oli yli 800 millisekuntia. Verkkoprotokollan luotettava viestinvälitys kesti pakettihävikin odotetulla tavalla. Pelissä hävinneet paketit olivat tunnistettavissa hitaana reaktiona luotettavia viestejä käyttäviin syötteisiin ja lievänä nykimisenä pelihahmojen ja ajoneuvojen liikkeissä. Viestien tehokas yhdistäminen paketteihin ja verkkototeutukselle asetetut rajat lähetettävien pakettien ja tavujen määrälle lyhyellä aikavälillä näyttävät ehkäisevän mobiiliyhteydelle tyypillistä tukkiutumista riittävästi.

Hieman odottamattomasti yhteyden laatuun vaikutti huomattavasti tapa liittää testitietokone mobiiliverkkoyhteyttä jakavaan Android-laitteeseen. USB-kaapelilla kytkettynä pakettien häviö oli aivan liian suuri eikä kumpaakaan peliä ollut mahdollista pelata. Edellä esitelty parempi tulos saavutettiin laitteen Bluetooth- ja Wi-Fi-yhteyksillä.

5.3 Arviointia ja jatkotutkimusaiheita

Pelien kehitystä monimutkaisti hieman tarkemman suunnittelun puute, mutta kehys soveltui niiden teknisiin tarpeisiin hyvin. Molemmat pelit pystyttiin kehittämään sen tarjoamalle alustalle ja muuttamaan verkkopeleiksi luvussa 4 esitellyillä keinoilla. C++-luokkien laajentaminen Lua-kielellä oli toimiva tapa rakentaa pelimekaniikkaa, ja jälkimmäisen kielen edut jäykkään järjestelmäohjelmointikieleen verrattuna näkyivät erityisesti fysiikkapelin käyttöliittymän ja etäkutsujen toteutuksessa.

Pelejä voitiin ajaa sekä Windows- että Linux-käyttöjärjestelmillä, ja verkkopelissä 32-bittisen Windows-ohjelman ja 64-bittisen Linux-ohjelman välillä ei havaittu yhteensopivuusongelmia. Heikkotehoisella Linux-tietokoneella pelien ruudunpäivitysnopeus jäi matalaksi ja vaihteli voimakkaasti, mutta profilointityökaluilla selvisi, että ongelma oli alustan OpenGL-tuessa. Kun grafiikka tehtiin mahdollisimman yksinkertaiseksi, fysiikkamoottori oli suurin tekijä kehyksen toimintanopeudessa. Verkkototeutus ei juuri erottunut pelin suorituskykyprofiloinnissa.

Tässä luvussa suoritetut mittaukset viittaavat siihen, että pelit ovat verkkoyhteysvaatimuksiltaan kohtuullisia ja toimivat myös käytännössä. Pelaajan ohjatessa monimutkaista ajoneuvoa fysiikkapelin pakettien lähetystahti oli melko korkea, mutta fysiikkasimulaatio ja sen synkronointi toimivat vakaasti. Yhteysongelmat heikensivät molemmissa peleissä ohjaustuntumaa havaittavasti, mutta näiden ulkopuolella pelit olivat pelattavia jopa hitaalla mobiiliyhteydellä.

OpenGL-pohjaisen grafiikkamoottorin, fysiikkamoottorin ja skriptitulkin sitominen toisiinsa oli varsin opettavainen kokemus olioiden elinkaarten ja riippuvuuksien hallinnasta. Bullet-fysiikkamoottori osoittautui käytössä yllättävän monipuoliseksi ja tarkaksi, mutta sen dokumentaatiossa on selvästi puutteita. Useamman oudon käytöksen selvittämisessä oli tarvetta tutkia kirjaston lähdekoodia dokumentoimattomien metodien ja jäsenmuuttujien tarkoituksen ymmärtämiseksi.

Testipelien rajallisen mittakaavan takia mallin skaalautuvuus hyvin suuriin pelimaailmoihin jäi tässä avoimeksi. Verkkopelin toimivuus tällaisessa ympäristössä

edellyttäisi, että palvelin voisi jollakin tapaa arvioida pelaajan havainnoimaa aluetta ja pitää kirjaa siitä, mistä kappaleista pelaajan tarvitsee tietää. Palvelin voisi toteuttaa tämän lähettämällä kappaleista lisäys- tai poistoviestejä, kun kappaleet tulevat havaintoalueelle tai poistuvat siltä. Erityistä huomiota vaatisivat tilanteet, joissa useat kappaleet ovat riippuvaisia toisistaan (kuten fysiikkapelin irtonaisista osista kootuissa ajoneuvoissa).

Viestityksen varomattomalla käytöllä pystyy helposti tuottamaan hyvin paljon verkkoliikennettä. Jonkinlainen toiminto lähetettävien viestien alkuperän jäljittämiseen olisi tämän selvittämisessä erittäin hyödyllinen. Automaattiset mittaukset olivat hyvä askel tähän suuntaan, mutta monimutkaisempi projekti voisi hyötyä optimointivaiheessaan kehittyneemmistä diagnostiikkatyökaluista. Viestinvälityksen rajoittamiseksi voisi olla hyödyllistä pystyä antamaan replikaatiojulistuksissa ehto sille, milloin jäsenarvon muutos tai metodikutsu tarvitsee viestittää.

Verkkoviestitystä olisi myös mahdollista optimoida olettamalla, että palvelimella ja asiakkaalla kääritään aina kaikki metodit verkkokutsuiksi samassa järjestyksessä. Näiden tunnisteina toimivat merkkijonot pakkautuvat hyvin, mutta vakaan järjestyksen takia tunnisteina voitaisiin myös käyttää pieniä kokonaislukuja. Koska tämä tarkoittaisi, että pienikin refaktorointi rikkoisi helposti verkkokoodin yhteensopivuuden, verkkopelin alustuksessa kannattaisi todennäköisesti lähettää jonkinlainen tarkistussumma viestitykseen käärityistä metodeista.

Koska kehyksen kentänmuokkaustyökalu kehitettiin pelien tapaan Lua-skriptinä, pelikenttien kollaboratiivinen muokkaus on periaattessa mahdollista. Tämän lisäksi upotettu Lua-kone sallisi myös pelin logiikan muokkaamisen ajon aikana. Tällainen toiminto voisi olla erittäin mielenkiintoinen esimerkiksi ohjelmoinnin opettamisen kannalta. Suurin riski tässä olisi se, että huolimaton käyttäjä voisi syöttää skriptin, joka saa palvelimen jumittumaan. On olemassa joitakin ratkaisuja Lua-tulkin suorituksen keskeyttämiseen sen ulkopuolelta esimerkiksi jonkinlaisesta “valvontasäikeestä”. Tällaisella olisi mahdollista pysäyttää skripti ja nostaa Lua-koneeseen virhe, joka lopettaa sen suorituksen.

6 Yhteenveto

Tutkielmassa lähdettiin tutkimaan verkkopelien toimintaperiaatteita ja toteutusta upotetun skriptikielen tarjoamaa abstraktiota hyödyntäen. Aluksi luotiin katsaus verkkopelien toteutustekniikoihin ja selvitettiin niiden vahvuuksia ja heikkouksia erilaisissa peleissä. Tämän jälkeen käytiin läpi valitun skriptikielen ominaisuuksia ja upottamista C++-ohjelmaan. Näiden pohjalta kehitettiin pelikehys, jonka verkkototeutus perustuu suureksi osaksi upotetussa skriptikielessä suoritettavaan viestintään. Kehyksen päälle rakennettiin kaksi testipeliä, joiden toimivuutta tutkittiin sekä keinotekoisesti että oikeissa verkkotilanteissa.

Työn tavoitteena oli löytää helppokäyttöisiä ja riittävän suorituskykyisiä menetelmiä toteuttaa verkkopelejä upotetun virtuaalikoneen avulla. Verkkopelikehys näyttää testipelien perusteella onnistuneen tässä. Kehitettyjä testipelejä ei ollut tarvetta kirjoittaa verkkokoodin ehdoilla, mutta ne pystyttiin luvussa 4 esitellyillä replikaatiomekanismeilla muuttamaan verkkopeleiksi hyvin pienellä vaivalla. Monimuotoisten viestien toteuttaminen järjestelmäohjelmointikielellä, joka ei tue riittävästi ajonaikaista tai edes käännöksenaikaista reflektiota, olisi vaatinut viestien määrittelemistä käsin yksi kerrallaan. Pelkästään C++-koodin kääntämiseen olisi tuhlautunut paljon lisää aikaa. Jos pelejä olisi tarvetta laajentaa, tämä ei myöskään vaatisi laajoja muutoksia lähdekoodiin verkkopelin takia.

Upotettu skriptikieli näytti olevan hyödyllinen abstraktiotyökalu sekä pelin korkean tason logiikan että sen verkkoviestityksen rakentamisessa. Serialisoituun skriptikielen tietoon perustuva viestitys pelissä näyttää olevan PC-laitteistolle tarpeeksi kevyttä, ettei sen vaikutus pelin suorituskykyyn näy pelimoottorin muiden toimintojen joukosta. Käytetty viestiformaatti vaatii Lua-kielen luonteen takia sellaisenaan paljon tilaa, mutta pakkautuu hyvin. Tutut käsitteet web-kehyksien kaltaisista muista verkkojärjestelmistä osoittautuivat tässä hyödylliseksi myös pelien verkkoviestityksen toteutuksessa.

Lähteet

Armitage, Grenville, Mark Claypool, Philip Branch, John Wiley, Grenville Armitage ja Mark Claypool. 2006. *Networking and Online Games - Understanding and Engineering Multiplayer Internet Games. United Kingdom.*

Bernier, Yahn W. 2001. "Latency compensating methods in client/server in-game protocol design and optimization". https://developer.valvesoftware.com/wiki/Latency_Compensating_Methods_in_Client/Server_In-game_Protocol_Design_and_Optimization.

Beznosyk, Anastasiia, Peter Quax, Karin Coninx ja Wim Lamotte. 2011. "Influence of Network Delay and Jitter on Cooperation in Multiplayer Games". Teoksessa *Proceedings of the 10th International Conference on Virtual Reality Continuum and Its Applications in Industry*, 351–354. VRCAI '11. Hong Kong, China: ACM. ISBN: 978-1-4503-1060-4. doi:10.1145/2087756.2087812. http://doi.acm.org/10.1145/2087756.2087812.

Brun, Jeremy, Farzad Safaei ja Paul Boustead. 2006. "Managing Latency and Fairness in Networked Games". *Commun. ACM* (New York, NY, USA) 49, numero 11 (marraskuu): 46–51. ISSN: 0001-0782. doi:10.1145/1167838.1167861. http://doi.acm.org/10.1145/1167838.1167861.

Chen, Kuan-Ta, Chun-Ying Huang, Polly Huang ja Chin-Laung Lei. 2006. "An Empirical Evaluation of TCP Performance in Online Games". Teoksessa *Proceedings of the 2006 ACM SIGCHI International Conference on Advances in Computer Entertainment Technology*. ACE '06. Hollywood, California: ACM. ISBN: 1-59593-380-8. doi:10.1145/1178823.1178830. http://doi.acm.org/10.1145/1178823.1178830.

Chen, Kuan-Ta, Polly Huang ja Chin-Laung Lei. 2006. "How Sensitive Are Online Gamers to Network Quality?" *Commun. ACM* (New York, NY, USA) 49, numero 11 (marraskuu): 34–38. ISSN: 0001-0782. doi:10.1145/1167838.1167859. http://doi.acm.org/10.1145/1167838.1167859.

Coumans, Erwin, ym. 2013. “Bullet physics library”. Viitattu 20. marraskuuta 2015. http://bulletphysics.org.

Défago, Xavier, André Schiper ja Péter Urbán. 2004. “Total Order Broadcast and Multicast Algorithms: Taxonomy and Survey”. *ACM Comput. Surv.* (New York, NY, USA) 36, numero 4 (joulukuu): 372–421. ISSN: 0360-0300. doi:10.1145/1041680.1041682. http://doi.acm.org/10.1145/1041680.1041682.

DICE. 2015. “Battlefield 4 release issues”. Viitattu 11. marraskuuta. http://battlelog.battlefield.com/bf4/news/view/addressing-netcode-in-bf4/.

Dick, Matthias, Oliver Wellnitz ja Lars Wolf. 2005. “Analysis of Factors Affecting Players’ Performance and Perception in Multiplayer Games”. Teoksessa *Proceedings of 4th ACM SIGCOMM Workshop on Network and System Support for Games*, 1–7. NetGames ’05. Hawthorne, NY: ACM. ISBN: 1-59593-156-2. doi:10.1145/1103599.1103624. http://doi.acm.org/10.1145/1103599.1103624.

ECMA International. 2011. *Standard ECMA-262 - ECMAScript Language Specification.* 5.1. Kesäkuu. http://www.ecma-international.org/publications/standards/Ecma-262.htm.

Fiedler, Glenn. 2010. “Floating point determinism”. http://gafferongames.com/networking-for-game-programmers/floating-point-determinism/.

Hsieh, Ji-Lung, ja C-T Sun. 2008. “Building a player strategy model by analyzing replays of real-time strategy games”. Teoksessa *Neural Networks, 2008. IJCNN 2008.(IEEE World Congress on Computational Intelligence). IEEE International Joint Conference on*, 3106–3111. IEEE.

“IEEE Standard for Distributed Interactive Simulation - Communication Services and Profiles”. 1996. *IEEE Std 1278.2-1995* (marraskuu): 1–20. doi:10.1109/IEEESTD.1996.80824.

IEEE Task P754. 2008. *IEEE 754-2008, Standard for Floating-Point Arithmetic.* 58. New York, NY, USA: IEEE, 29. elokuuta. ISBN: 0-7381-5753-8 (paper), 0-7381-5752-X (electronic). doi:http://dx.doi.org/10.1109/IEEESTD.2008.4610935. http://ieeexplore.ieee.org/servlet/opac?punumber=4610933.

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*, 1–26. HOPL III. San Diego, California: ACM. ISBN: 978-1-59593-766-7. doi:http://doi.acm.org/10.1145/1238844.1238846. http://doi.acm.org/10.1145/1238844.1238846.

Li, Siliang, ja Gang Tan. 2014. "Finding Reference-Counting Errors in Python/C Programs with Affine Analysis". Teoksessa *ECOOP 2014–Object-Oriented Programming*, 80–104. Springer.

March, Salvatore T., ja Gerald F. Smith. 1995. "Design and Natural Science Research on Information Technology". *Decis. Support Syst.* (Amsterdam, The Netherlands, The Netherlands) 15, numero 4 (joulukuu): 251–266. ISSN: 0167-9236. doi:10.1016/0167-9236(94)00041-2. http://dx.doi.org/10.1016/0167-9236(94)00041-2.

Muhammad, Hisham, ja Roberto Ierusalimschy. 2007. "C APIs in Extension and Extensible Languages."

Mullender, Sape J. 1993. "Introduction to Distributed Systems". *CERN EUROPEAN ORGANIZATION FOR NUCLEAR RESEARCH-REPORTS-CERN:* 29–29.

El-Nasr, Magy Seif, ja Brian K. Smith. 2006. "Learning Through Game Modding". *Comput. Entertain.* (New York, NY, USA) 4, numero 1 (tammikuu). ISSN: 1544-3574. doi:10.1145/1111293.1111301.http://doi.acm.org/10.1145/1111293.1111301.

NVIDIA Corporation. 2015. "PhysX Knowledge Base". Viitattu 7. marraskuuta 2015. http://www.nvidia.com/object/physx_knowledge_base.html.

Pall, Mike. 2007. "LuaJIT". *http://luajit.org.* Viitattu 20. helmikuuta 2015.

Pedone, Fernando, Matthias Wiesmann, André Schiper, Bettina Kemme ja Gustavo Alonso. 2000. "Understanding Replication in Databases and Distributed Systems." Teoksessa *ICDCS*, 464–474. IEEE Computer Society. ISBN: 0-7695-0601-1. http://dblp.uni-trier.de/db/conf/icdcs/icdcs2000.html#PedoneWSKA00.

Phelps, Andrew M., ja David M. Parks. 2004. "Fun and Games: Multi-Language Development". *Queue* (New York, NY, USA) 1, numero 10 (helmikuu): 46–56. ISSN: 1542-7730. doi:10.1145/971564.971592. http://doi.acm.org/10.1145/971564.971592.

"Quake III Arena GPL Source Release". 2015. Viitattu 20. helmikuuta. https://github.com/id-Software/Quake-III-Arena.

Scacchi, Walt. 2011. "Modding As a Basis for Developing Game Systems". Teoksessa *Proceedings of the 1st International Workshop on Games and Software Engineering*, 5–8. GAS '11. Waikiki, Honolulu, HI, USA: ACM. ISBN: 978-1-4503-0578-5. doi:10.1145/1984674.1984677. http://doi.acm.org/10.1145/1984674.1984677.

Smed, Jouni, Timo Kaukoranta ja Harri Hakonen. 2002. "A Review on Networking and Multiplayer Computer Games". Teoksessa *IN MULTIPLAYER COMPUTER GAMES, PROC. INT. CONF. ON APPLICATION AND DEVELOPMENT OF COMPUTER GAMES IN THE 21ST CENTURY*, 1–5.

"Source Multiplayer Networking". 2015. Viitattu 20. helmikuuta. https://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking.

"Steamworks Documentation". 2015. Viitattu 28. syyskuuta. https://partner.steamgames.com/documentation/api.

"Synchronous RTS engines and a tale of desyncs". 2015. Viitattu 20. helmikuuta. http://forrestthewoods.com/synchronous-rts-engines-and-a-tale-of-desyncs/.

Terrano, Mark, ja Paul Bettner. 2001. “1500 Archers on a 28.8: Network Programming in Age of Empires and Beyond”. Viitattu 20. helmikuuta 2015. http://www.gamasutra.com/view/feature/131503/1500_archers_on_a_288_network_.php?page=1.

“Unreal Engine”. 2015. Viitattu 20. helmikuuta. http://www.unrealengine.com.

Wang, Alf Inge, Martin Jarrett ja Eivind Sorteberg. 2009. “Experiences from Implementing a Mobile Multiplayer Real-time Game for Wireless Networks with High Latency”. *Int. J. Comput. Games Technol.* (New York, NY, United States) 2009 (tammikuu): 6:1–6:14. ISSN: 1687-7047. doi:10.1155/2009/530367. http://dx.doi.org/10.1155/2009/530367.

Waveren, J.M.P. van. 2006. “The Doom III Network Architecture”. http://mrelusive.com/publications/papers/The-DOOM-III-Network-Architecture.pdf.

Yan, Jeff, ja Brian Randell. 2005. “A Systematic Classification of Cheating in Online Games”. Teoksessa *Proceedings of 4th ACM SIGCOMM Workshop on Network and System Support for Games*, 1–9. NetGames ’05. Hawthorne, NY: ACM. ISBN: 1-59593-156-2. doi:10.1145/1103599.1103606. http://doi.acm.org/10.1145/1103599.1103606.

Liitteet

A Esimerkki pelaajaluokasta

Pelaajaluokka toimii kehyksessä rajapintana pelaajan käyttöliittymälle antaman syötteen ja pelimekaniikan välillä. Kun pelaaja liittyy palvelimelle asiakkaaksi, molempiin päihin luodaan pelaajaolio, jonka avulla voidaan viestiä pelaajan syötteitä ja yleistä tilaa (nimi, pisteet, pelissä vietetty aika) molempiin suuntiin.

```
require 'multiplayer'
require 'i18n'

Player = Class:extend('Player')

function Player:init(args)
    self.profile = { name = translate('Unnamed player') }
    self.weapon = 1
end

-- Attempts to select a weapon.
function Player:selectWeapon(num)
    self.weapon = num
    self:setActiveWeapon(num)
end
net.wrapClientServer(Player, 'player', 'selectWeapon')

-- This input produces a message for all users.
-- The 'chat' message is a custom type handled in the UI.
function Player:chat(message)
    net:call('chat', message, self:getName())
end
net.wrapClientServer(Player, 'player', 'chat')

-- Called by game to assign object being controlled.
function Player:assign(objectId)
    local object = scene:getObjectByUID(objectId)
```

```
    self.assigned = object
    if not object then
        return
    else
        object.netPhysicsRate = 4
        object.owner = self
    end
end
-- This wraps the command so that when called on a server, it is
    remote-called
-- on the relevant client.
-- Server-only remote calls are sent using reliable messaging by
    default.
net.wrapRemotePlayer(Player, 'player', 'assign')

-- Sets desired motion vector for whatever the player is controlling.
function Player:move(motion)
    if not self.assigned then return end

    local length = motion:length()
    if length > 1.0 then
        motion = motion / length
    end
    self.assigned.motion = motion
end
-- This wraps the method so it is replicated on the server, but also
    simulated
-- on the client right away. This is a form of client-side
    prediction.
-- Client-predicted calls are sent using lossy messaging by default.
net.wrapClientServer(Player, 'player', 'move')

-- Called by client/local player to indicate direction being looked
    at.
-- This may cause a vehicle to try to turn to that direction, etc.
function Player:turn(angle)
    if not self.assigned then return end
```

```
    self.assigned.dir = angle
end
net.wrapClientServer(Player, 'player', 'turn')

function Player:respawn()
    game:spawnPlayer(self)
end
net.wrapServerOnly(Player, 'player', 'respawn')

-- Shoots in a given direction.
function Player:shoot(dir)
    local character = self.assigned
    if character then
        character:shoot()
    end
end
-- This wraps the method so that when called on a client, it
   generates a remote
-- call for the server to execute.
-- Server-only remote calls are sent using reliable messaging by
   default.
net.wrapServerOnly(Player, 'player', 'shoot')

-- In the physics game, this disassembles the player's vehicle.
function Player:disassemble()
    local character = self.assigned
    if character.disassemble then
        character:disassemble()
    end
end
net.wrapServerOnly(Player, 'player', 'disassemble')

function Player:updateProfile(profile)
    for k, v in pairs(profile) do
        self.profile[k] = v
    end
end
```

```
function Player:getProfile()
    return self.profile
end

function Player:getName()
    return self:getProfile().name
end
```

Listaus 6.1. Pelaajaluokan toteutus.

B Räiskintäpelin hahmon toteutus

Listauksessa 6.2 on yksinkertaistettu esimerkki C++-pohjaluokan laajentamisesta omaksi luokaksi Lua-ympäristössä. Tämän luokan olio voi kävellä ja hyppiä pelimaailmassa sekä ampua aseellaan pelaajan syötteen mukaan.

```
FPSCharacter = SceneObject:extend('FPSCharacter', {
    init = function(self, args)
        SceneObject.init(self, args)
        self.physicsEnabled = true

        -- Players are axis-aligned capsules.
        -- Capsule shapes are two spheres with cylinder sides between
        -- x = radius, y = distance between sphere centers
        -- (actual height is thus y + 2x).
        self.physicsShape = 'capsule_z'
        self.physicsScale = vec3(0.4, 0.6, 0.0)

        self.restitution = 0.0
        self.friction = 0.5
        self.physicsTensor = vec3(0)
        self.mass = 70.0

        self.linearDamping = 0.5

        self.isJumping = false
```

```
        self.standingOn = nil
        self.standingPos = nil
        self.health = 100
        self.motion = vec3(0)

        self.cameraOffset = vec3(0, 0, 0.5)

        self:attach(SceneMesh {
            src = 'character-placeholder.obj',
            angle = quatFromAA(math.pi/2, vec3(1, 0, 0)),
            scale = 0.58,
            diffuse = 'test-diffuse-a.png',
            normal = 'test-normal.png',
            normalDetail = 'normal-detail.png',
            smoothness = 'test-smoothness.png',
        })
        -- attach weapon model separately so we can "animate" aiming
        self:attach(SceneMesh {
            pos = vec3(0.5, 0.2, 0.0),
            scale = 0.5,
            angle = quatFromAA(math.pi/2, vec3(1, 0, 0)),
            src = 'railgun.obj',
            diffuse = 'railgun-diffuse.png',
            normal = 'null-normal.png',
            normalDetail = 'brushedmetal-normal.png',
            smoothness = 'railgun-params.png'
        })
    end,
    onSpawn = function(self)
        -- sparkly respawn effects, etc. here
    end,
    onSim = function(self, dt)
        local motion = self.motion
        -- swizzle motion input into FPS controls
        motion = vec3(motion.x, motion.z, -motion.y)

        local maxVel = 12
```

```
local refVel
local controlStrength

if self.standingOn then
    controlStrength = 750
    refVel = self.standingOn.vel
else
    -- FPS characters can always do this somehow.
    controlStrength = 50
    refVel = vec3(0.0)
end

if self.dir then
    local intent = refVel + self.dir:rotate(motion) * maxVel
    local diff = intent - self.vel
    diff.z = 0
    if diff:lengthSquared() > 0 then
        local force = diff * controlStrength
        self:applyForce(force)
        if self.standingOn then
            self.standingOn:applyForce(-force,
                self.standingPos)
        end
    end

    -- Turn to face intended direction.
    self.angle = self.dir
end

if self.isJumping then
    if self.standingOn then
        local jump = vec3(0, 0, 700)
        self:applyImpulse(jump)
        self.standingOn:applyImpulse(-jump, self.standingPos)
    end
    self.isJumping = false
end
```

```
    self.standingOn = nil
end,
takeDamage = function(self, amount)
    self.health = self.health - amount
    if self.health <= 0 then
        scene:add(SmallExplosion {
            pos = self.pos,
            velofs = self.vel,
            maxlife = 1.0,
            color = vec4(0.2, 0.6, 2.0, 1.0),
        })
        scene:remove(self)
    end
end,
onHit = function(self, other, worldPos, normal, impulse)
    -- If the contact point's below us, we're standing on it.
    if (worldPos - self.pos):dot(scene.gravity) > 0 then
        self.standingOn = other
        if other then
            self.standingPos = other:toLocalPos(worldPos)
        end
    end
end,
shoot = function(self)
    if not self.dir then
        return
    end
    local direction = self.dir:rotate(vec3(0, 0, -1))
    local projectile = RailRound {
        pos = self.pos + direction * (1.0 + self.vel:length() *
            0.1),
        vel = self.vel + direction * 400,
        angle = self.dir,
        source = self,
        weapontype = 4,
        -- Prevent collision with projectiles and their owners.
```

```
            collisionGroup = self.collisionGroup
        }
        scene:add(projectile)
    end,
    jump = function(self)
        self.isJumping = true
    end,
    onDespawn = function(self)
        if self.owner then
            net:call('chat',
                ('%s died!'):tlformat(self.owner:getName()))
            self.owner:assign(nil)
        end
    end,
})
net.serverOnly(FPSCharacter, "takeDamage")
net.serverOnly(FPSCharacter, "onDespawn")
net.replicateObjectProperty(FPSCharacter, "health")
```

Listaus 6.2. Pelihahmon toteutus.