# Käänteinen debuggaus videopeleissä

Diplomityö
Turun yliopisto
Tulevaisuuden teknologioiden laitos
Ohjelmistotekniikka
2019
Olli Mannevaara

TURUN YLIOPISTO
Tulevaisuuden teknologioiden laitos

OLLI MANNEVAARA: Käänteinen debuggaus videopeleissä

Diplomityö, 62 s., 17 liites.
Ohjelmistotekniikka
Joulukuu 2019

---

Videopelien laadunvarmistuksella on merkittävä vaikutus pelien menestymiseen. Kun videopeli julkaistaan, sitä seuraa myyntijakso, jonka menestymiseen vaikuttaa pelissä ilmenevät bugit. Bugien ilmeneminen on todennäköisempään, mitä useampi pelaa peliä. Koska pelinkehittäjien resurssit laadunvarmistuksessa ovat rajalliset, kaikkia bugeja ei voida löytää ennen julkaisua.

Moderneissa pelialustoissa pelien suoritukseen vaikuttaa laitteiston resursseja jakava käyttöjärjestelmä. Resurssien jako on edellytys sille, että sovelluksia voidaan suorittaa rinnakkain. Epävarmuus suorituksessa vaikuttaa sovelluksen logiikan ajoituksiin. Jos sovellukseen vaikuttavat epädeterministiset syötteet voidaan tallentaa ja toistaa tarvittaessa, voidaan sovelluksen debuggaamiseen käyttää käänteisenä debuggaamisena tunnettua tekniikkaa. Käänteinen debuggaaminen mahdollistaa prosessin, jossa sovellusta voidaan eteenpäin suorituksen lisäksi suorittaa myös taaksepäin.

Koska pelit perustuvat sekä reaaliaikaiseen suoritukseen että pelin ja pelaajan väliseen vuorovaikutukseen, pitää peleille toteutetun käänteisen debuggaamisen vaatimukset määrittää erikseen. Käänteinen debuggaaminen on mahdollista toteuttaa hallitsemalla allaolevaa pelialustaa, mutta pelien näkökulmasta tämä vaikuttaa liian merkittävästi alustan suorituskykyyn. Käänteinen debuggaus voidaan tehdä pelin ominaisuudeksi, jolloin sen käyttö on suorituskykyisempää kuin koko alustaa nauhoittavat ratkaisut.

Tässä diplomityössä tutkitaan mahdollisuutta toteuttaa käänteinen debuggaaminen osaksi peliä. Käänteisen debuggauksen yhteensopivuutta peleihin selvitetään perehtymällä ohjelmistotuotannossa käytettyihin ratkaisuihin. Havaittujen vaatimusten pohjalta valitaan toteutusmalli videopelien käänteiselle debuggaamiselle ja sitä käyttäen toteutetaan yksinkertainen videopeli. Siitä ilmenee, että käänteinen debuggaaminen on mahdollista toteuttaa. Toteutuksesta käy myös ilmi interaktiivisen käänteisen debuggaamisen haasteet.

Asiasanat: videopelit, laadunvarmistus, käänteinen debuggaus

UNIVERSITY OF TURKU
Department of Future Technologies

OLLI MANNEVAARA: Reverse Debugging in Video Games

Master's Thesis, 62 p., 17 app. p.
Software Engineering
December 2019

---

Quality assurance has a significant effect on the success of a video game. When a game is released, it is followed by a sales period which is affected by the bugs present in the video game. The more players there are, the more likely it is that bugs are noticed. Because of the limited resources in video game development, all bugs cannot be detected and fixed before the release.

In modern platforms, resources are divided at the operating system level. The splitting of resources adds non-determinism to the execution of the software. If non-determinism can be recorded and replayed, the software can be re-executed deterministically which leads to more powerful debugging techniques. Debugging based on deterministic re-execution is known as reverse debugging. With reverse debugging it is possible to step forwards as well as backwards in the execution of a software.

Because video games depend on real-time execution and the interaction of the player, implementing reverse debugging for video games has different requirements from conventional software. Reverse debugging can be implemented on a platform level, but it will affect game performance negatively. If reverse debugging is implemented as a feature for the game, it will perform better than platform-wide solutions.

This thesis studies the possibility to implement reverse debugging as a feature in a video game. The compatibility of existing reverse debugging solutions for video games are evaluated. Appropriate model for reverse debugging in video games is proposed and the proposal is validated through the creation of a video game. Reverse debugging is successfully implemented in the game and additional requirements for interactive reverse debugging are shown.

Keywords: Video Games, Quality Assurance, Reverse Debugging

# Sisältö

# Listaukset

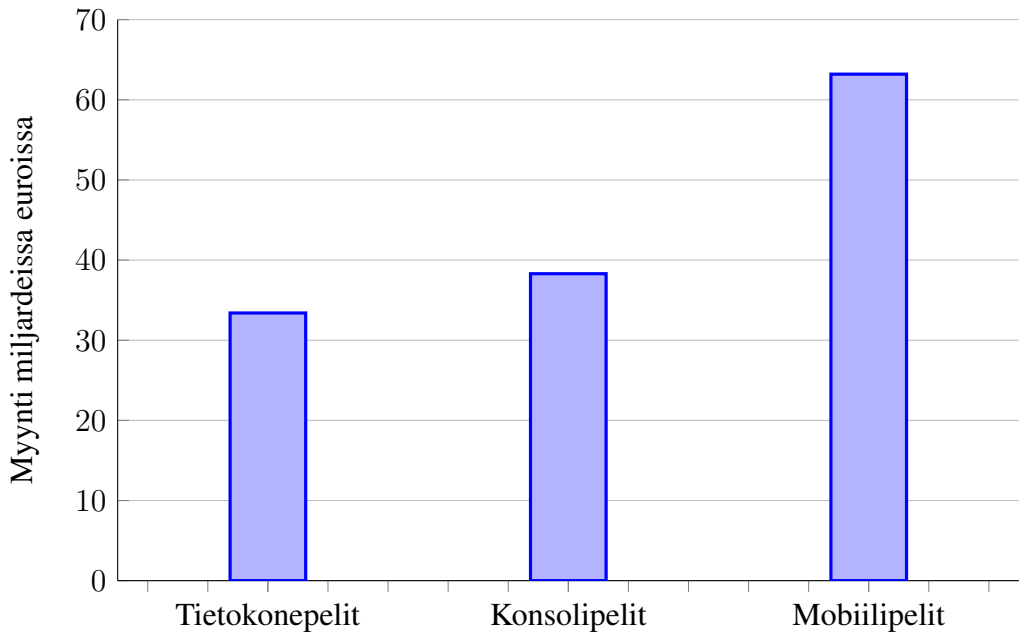
# Luku 1

# Johdanto

## 1.1 Tutkimuksen motivaatio

Videopeliteollisuus on yksi suurimmista viihdealan teollisuuksista. Sen suuruudeksi arvioitiin noin 135 miljardia dollaria vuonna 2018 [1]. Nykyisin videopeliteollisuuden merkittävimmät laitealustat ovat henkilökohtaiset tietokoneet (PC), pelikonsolit ja älypuhelimet (ks. kuva 1.1).
Laitealustat eroavat pääasiassa suorituskyvyyltään ja ulkomuodoltaan. Tietokoneiden etu on niiden päivitettävyys, taaksepäinyhteensopivuus ja kyky hyödyntää lähes kaikkia oheislaitteita. Pelikonsolien ja älypuhelimien etuna voidaan ajatella niiden suurta samankaltaisten laitteiden määrää sekä niiden edullisuutta tietokoneisiin nähden. Erityisesti pelikonsolien ja älypuhelimien mukana tulevat sensorit, peliohjaimet ja muut oheislaitteet antavat pelinkehittäjille mahdollisuuksia kehittää uusia pelaamisen muotoja.

Videopelien alkuhistorian yhteydessä usein puhutaan Atarin vuonna 1972 julkaisemasta Pong-videopelistä. Pong ei ole ensimmäinen videopeli teknisestä näkökulmasta, mutta sen asema ensimmäisenä merkittävänä kaupallisena menestyksenä antaa sille erityisaseman videopelien historiassa. Pongin vaikutus oli niin merkittävä, että ensimmäisen sukupolven videopelikonsolit tunnetaan myös Pong-klooneina. Atarin menestymisen myötä videopeli terminä vakiintui. [3]

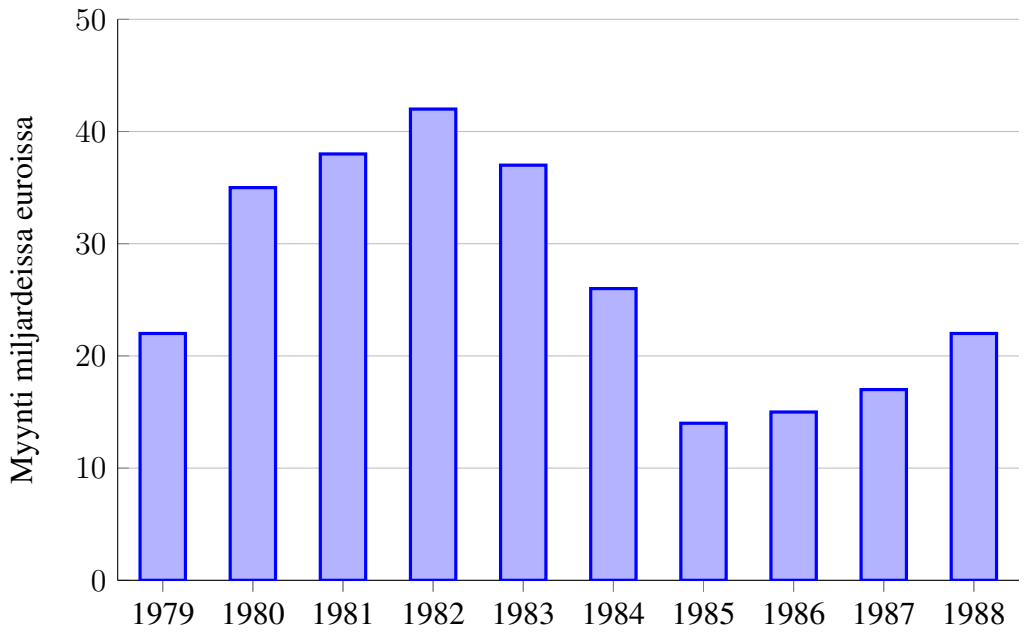



Kuva 1.1: Peliteollisuuden arvioitu myynti vuonna 2018 [2]

Videopeleistä tuli tunnettu viihteen muoto 1970-luvulla, jolloin ne yleistyivät pelihalleissa. Pelihalleissa olevat videopelit perustuivat maksullisten pelikertojen ansaintamalliin. Videopelien kehittäminen koostui pelaamiseen käytetyn laitteiston suunnittelemisesta ja sen ohjelmoinnista. Tietokonepelit olivat vielä harvinaisia, koska henkilökohtaisten tietokoneiden markkinat eivät olleet vielä kasvaneet. PC-markkinoiden syntymisen myötä syntyi myös markkinat tietokonepeleille. Pelihallipelien ja konsolipelien etu tietokonepeleihin nähden oli niiden erityistarkoitukseen rakennettu laitteisto.

Videopeliteollisuus kasvoi vuoteen 1982 asti (ks. kuva 1.2). Kasvun aikaa kutsutaan videopelien kultakaudeksi. Kultakausi alkoi vuonna 1978 japanilaisen yrityksen Taiton julkaisemasta Space Invader -videopelistä ja katsotaan loppuneen vuoden 1983 videopeliteollisuuden lamaan. Lamaa edeltävinä vuosina videopelaaminen oli jakautunut pelihallipeleihin ja pelikonsoleihin.

Laman taustalla oli asiakkaiden luottamuksen menettäminen. Atarin asema videopelikonsolien markkinajohtajana kostautui, kun kolmannen osapuolen pelivalmistajat tuot-




Kuva 1.2: Peliteollisuuden arvioitu myynti vuosina 1979-1988 [1]

tivat alustalle liikaa huonolaatuisia pelejä. Pelien laaduttomuus koski sekä pelejä sisällöllisesti että toiminnallisesti.

Videopelilamasta toivuttiin, kun Nintendo Entertainment System -pelikonsoli tuli Pohjois-Amerikan markkinoille. Nintendo varmisti oman hallinta-asemansa kolmannen osapuolen pelitaloihin määrittämällä ylärajan siihen, kuinka monta peliä saa julkaista vuodessa. Nintendon pelikonsolissa käytettiin pelikasetteja, joiden valmistaminen oli mahdollista ainoastaan Nintendon kautta. Nintendo tuotti pelikasetteja yksittäisinä tilauksina, joka johti siihen, että pelinkehittäjien piti ottaa rahallinen riski pelin markkinoille tuomisessa.

Web-sovelluksilla on myös tausta, joka sisältää laaduntakaamiseen liittyviä haasteita. JavaScript on web-sovelluksiin käytetty ohjelmointikieli, jonka syntymää seurasivat haasteet kolmannen osapuolen toteutuksista. Koska JavaScript oli lähtökohtaisesti vain tavallisiin verkkosivuihin logiikkaa lisäävä laajennus, verkkoselainten tuki sille muodostui pitkän ajan kuluessa. Se kehitettiin alun perin Netscape-verkkoselaimeen ja suunniteltiin lähestyttäväksi oman eheytensä kustannuksella.

Ahneus määritti laaduttomuuden videopelilamaan johtaneissa videopeleissä, mutta web-sovellusten kehityksessä haasteellisuutta tuli JavaScript-kielen alkeellisuudesta ja verkkoselainten välillä poikkeavista toteutuksista. Microsoftin Internet Explorer -verkkoselaimeen tuotettiin oma versio JavaScriptistä, joka tunnettiin JScriptinä.

Kun Microsoft saavutti merkittävän markkinajohtajan aseman verkkoselainten toimittajana, kehitettiin Internet Exploreriin uusia laajennuksia, jotka pirstaloivat web-sovellusten kehityksen alustakohtaisemmaksi. Verkkoselainten välistä yhteensopivuutta pystyttiin paikkaamaan alustakohtaisella logiikalla, joka vaikutti haitallisesti web-sovellusten kehityksessä, vaikka Internet Explorer menetti markkinajohtajan asemansa.

Konsolille tehtyjen videopelien pelaamiseen on tuotettu emulaattoreiksi kutsuttuja sovelluksia. Emulaattorit ovat videopelien pelaamiseen käytettyjä alustoja matkivia eli emuloivia sovelluksia. Emulaation haasteellisuus riippuu emuloidun alustan monimutkaisuudesta. Varhaisten konsolien tarkka emulointi on tullut mahdolliseksi nopeilla tietokoneilla. Emulaattoreilla pystytään pelaamaan videopelejä, mutta niillä pystytään myös tarkastelemaan videopelejä alkuperäistä pelialustaa paremmin.

Emulaattorit, jotka pystyvät sekä pelaamaan että palaamaan peluussa aikaisempaan tilaan, kutsutaan uudelleennauhoitusemulaattoreiksi (*rerecording emulator*). Uudelleennauhoituksella voidaan peleissä olevia virhetiloja tarkastella toistuvasti, palaamalla virhettä aikaisempaan tilaan. Tällaista virhetilojen korjaamista kutsutaan käänteiseksi debuggaamiseksi (*reverse debugging*).

Käänteinen debuggaus on usein ollut erityisten virtuaalisoitujen ympäristöjen kuten virtuaalikoneiden ja emulaattorien työkalu. Tekniikka on pystytty onnistuneesti tuottamaan web-sovelluksiin käyttämällä funktionaalista ohjelmointia. Funktionaalinen ohjelmointi on tehokas tapa varmistaa tarkka yhteys logiikan lähtö- ja loppuarvojen välille, mutta se ei ole edellytys käänteiselle debuggaamiselle.

Videopelien laadulla on merkitys niiden viihdyttävyydessä. Heikko laadunvarmistus voi johtaa bugeihin, jotka voivat olla haitallisia pelaamisessa tai ne voivat olla huomiota

herättäviä peliarvosteluissa. Bugien ilmeneminen on todennäköistä, kun peliä pelaa huomattavasti suurempi määrä pelaajia kuin pelinvalmistajan oma pelitestaus pystyi käsittelemään. Käänteinen debuggaaminen edustaa sekä tapaa antaa pelinkehittäjille työkalu poistaa bugien todentamisen epävarmuus että mahdollistaa varman tavan dokumentoida bugit laadunvarmistusjärjestelmään.

Videopeleillä on merkittävä arvo alustariippumattomuudesta. Mitä useammalla laitteelle peli voidaan tuottaa, sitä enemmän on mahdollisia asiakkaita. Käänteinen debuggaaminen voidaan tuottaa alustakohtaisesti pelin ulkopuolelle tai alustariippumattomasti peliin itseensä. Web-sovelluksissa käytetty käänteisen debuggaamisen toteutus on yksi mahdollinen tapa tuoda käänteinen debuggaaminen videopeleihin.

## 1.2 Tutkimuskysymykset

Tämä diplomityö tutkii mahdollisuutta toteuttaa videopeleissä käänteinen debuggaus. Työssä ei määritetä uutta käänteiseen debuggaukseen perustuvan debuggerin vaatimuksia, vaan pohditaan, miten käänteinen debuggaus voidaan mahdollistaa tekemällä muutoksia pelin logiikkaan. Muutokset logiikassa voivat olla itsessään pelin laatua parantavia ja hyödyllisiä tavallisten debuggausmenetelmien näkökulmasta.

Työn tutkimuskysymykset ovat:

1. Voidaanko videopeliä debugata käänteisesti?
2. Miten käänteistä debuggausta saadaan interaktiivisemmaksi?

Ensimmäinen kysymys etsii vastausta mahdollisuuteen toteuttaa käänteinen debuggaus. Käänteisen debuggauksen onnistumisen määrittäminen on tarpeellista työn näkökulmasta. Videopelien käänteisessä debuggauksessa bugi pitää pystyä toistamaan. Käänteisen debuggaamisen asiayhteydessä kyky toistaa bugi implikoi, että toistettu bugi tapahtuu niin kuin se tapahtui ensimmäistä kertaa ja että bugi voidaan toistaa mielivaltaisen monta kertaa. Tätä tarkastellaan aliluvussa 1.2.1.

Toinen kysymys etsii käytäntöä, millä käänteistä debuggaamista voidaan soveltaa videopelien laadunvarmistuksessa. Vaikka bugi voidaan toistaa, mitä suorituskykyvaatimuksia bugin toistaminen asettaa käänteiselle debuggaamiselle? Koska bugin korjaaminen voi edellyttää useita toistoja, että bugi ymmärretään, käänteisen debuggaamisen optimointi sellaiseksi, että sitä pystytään soveltamaan on tärkeää. Käänteisen debuggaamisen suorituskykyvaatimuksia tarkastellaan aliluvussa 1.2.2.

### 1.2.1 Bugien toistamisen määritys

Erityistä videopeleissä verrattuna muihin sovelluksiin on se, että pelien suoritus perustuu pelaajan tekemiin syötteisiin. Pelaajan syötteet ovat arvoja, jotka ovat pelin logiikan näkökulmasta ilmaistu ennalta-arvaamattomasti ajassa. Arvaamattomuus tekee peleistä erityisen epädeterministisiä. Käänteinen debuggaus siis edellyttää menetelmää, jolla epädeterministiset syötteet saadaan kaapattua niin, että ne voidaan toistaa sovelluksen suorituksessa, jonka seurauksena bugi toistuu pelissä tarkasti.

Käänteisessä debuggaamisessa haasteellista on myös vielä tuntemattomien bugien tallentaminen. Kun bugi tapahtuu, eikä siihen johtaneita syötteitä ole nauhoitettu, voi bugin toistaminen olla epävarmaa. Sama bugi voi toistua toisilla syötteillä, mutta riippuen syötteiden määrästä ja laadusta voi saman bugin toistaminen olla vaikeaa. Tuntemattomien bugien nauhoittamisen ongelma voidaan ratkaista nauhoittamalla kaikki pelin suorittamiseen liittyvä epädeterminismi, riippumatta siitä onko tarkoitus tallentaa bugeja.

Videopelien käänteisen debuggauksen hyvä toteutus on oikeellisuuden lisäksi myös suorituskykyinen. Suorituskykyinen epädeterminismin nauhoitus häiritsee pelin antamaa kokemusta minimaalisesti, jolloin se soveltuu ajettavaksi pelitestaamisessa ja ad hoc-testaamisessa. Minimivaatimukset käänteisessä debuggaamisessa pitäisi olla niin alhaiset, että sitä voidaan käyttää aina, varautuen myös harvinaisiin ja vaikeasti toistettaviin bugeihin.

### 1.2.2 Interaktiivisen debuggaamisen määritys

Kun bugien toistaminen on mahdollista, tulee käänteisen debuggauksen työskentelymallin käytettävyys tärkeäksi. Voidaan ajatella, että tavallinen debuggaus on myös käänteistä debuggaamista, kun bugit voidaan toistaa tarkasti ja ilman rajoituksia. Debuggaamisen näkökulmasta yksinkertaisin tapa edetä ajassa taaksepäin on aloittaa debuggaaminen uudelleen alusta ja pysäyttää sovelluksen suoritus aikaisempaan kohtaan.

Käänteisen debuggaamisen työskentelytapa ei ole kovinkaan interaktiivinen, jos se tarkoittaa sovelluksen käynnistämistä lähtötilasta, johon simuloidaan kaikki aikaisemmat epädeterministiset vaikuttajat nauhoitettujen arvojen perusteella. Jos sovelluksen aikaisempi suoritus rasitti käytettyä suoritinta maksimaalisesti, ei ohjelman suoritusta pystytä nopeuttamaan ilman nopeampaa suoritinta. Videopeleissä suoritusnopeutta mitataan usein ruudunpäivityksissä. Jos ruudunpäivitysnopeus rajoitetaan alle laitteiston maksiminopeuden, voidaan videopelin näkökulmasta aikaa suorittaa reaaliaikaa nopeammin debuggaamisessa. Debuggaus ei ole kuitenkaan tarpeeksi interaktiivista, jos debuggaamisen nopeus on suoraan verrannollinen pelin alkuperäiseen suoritukseen menneeseen aikaan. Hyvä interaktiivinen toteutus videopelin käänteiselle debuggaukselle mahdollistaa bugien tarkastelun katkeamattomana prosessina.

## 1.3 Tutkimusmenetelmä ja työn rajaus

Työssä tutkitaan web-sovelluksia ja niissä sovellettua Redux-kirjastoa, joka mahdollistaa käänteisen debuggauksen. Web-sovellusten käänteisen debuggaamisen haasteet ja ratkaisut pyritään tunnistamaan ja pohditaan niiden soveltuvuutta videopeleissä. Ratkaisuja todennetaan tekemällä niistä käänteisesti debuggatava peli LÖVE-pelimoottorissa.

LÖVE on avoimeen lähdekoodiin perustuva pelimoottori, jota ohjelmoidaan Lua-kielellä. LÖVE-pelimoottori tarjoaa ohjelmointirajapinnat syötelaitteiden lukemiseen, grafiikan piirtämiseen ja äänien soittamiseen. Lua-kielellä on paljon samankaltaisuuksia

web-sovellusten kehityksessä käytetyn JavaScript-kielen kanssa. Molemmat ovat dynaamisesti tyypitettyjä ja roskankeruuseen perustuvia ohjelmointikieliä.

Työssä keskitytään selvittämään käänteisen debuggauksen käyttömahdollisuus videopeleissä. Lähestyminen tapahtuu yksittäisen kehitykseen käytetyn tietokoneen sisällä, jossa pelin bugit on myös todettu. Työhön tuotettu pelilogiikka on yksisäikeistä. Työssä ei pyritä ratkaisemaan moniytimisen suorittimen tai oheislaitteiden, kuten näytönohjaimen tai äänikortin, käytöstä syntyviä haasteita.

## 1.4 Työn rakenne

Luvussa 1 esitellään työn motivaatio ja tutkimuskysymykset. Työn aihetta esitellään kertomalla videopelien ja laadunvarmistuksen suhteesta. Tutkimuskysymysten ongelmat esitellään ja niistä johdetaan laatukriteerit työhön toteutetulle pelille.

Luvussa 2 esitellään käänteisen debuggaamisen käsite. Kahta ohjelmistoteollisuudessa käytettyä käänteisen debuggaamisen työkalua tutkitaan ja käänteiselle debuggaamiselle esitellään kaksi vaihtoehtoista toteutustapaa.

Luvussa 3 ilmaistaan varhaisten videopelien toistettavuuden piirteitä. Pikapelaaminen ja emulaattorien rooli pikapelaamisessa esitellään. Käänteisen debuggaamisen tuottamien syötetiedostojen arvo kommunikoinnin ja laadunvarmistuksen välineenä tuodaan esiin.

Luvussa 4 esitellään web-sovellusten eri osat ja käsitellään web-ohjelmointiin syntyneitä mahdollisuuksia debugata käänteisesti. Käänteisen debuggaamisen mahdollistavaa Redux-kirjastoa tarkastellaan lähemmin.

Luvussa 5 esitellään videopeleissä käytetyt päivitysrutiini ja piirtorutiini. Niitä hallitsevasta pelisilmukasta esitellään kaksi vaihtoehtoa. Vaihtoehdoista tuodaan esiin käänteisessä debuggaamisessa tarvittavia piirteitä.

Luvussa 6 esitellään videopeleille soveltuva käänteinen debuggaus. Osaan debuggaamiseen liittyviin yksityiskohtiin esitellään toteutusmalleja. Lopuksi käydään läpi työhön tuotetun käänteisesti debuggattavan pelin haasteet ja miten ne ratkaistiin.

Luvussa 7 katsotaan ohjelmistotuotannossa käytetyn käänteisen debuggaamisen kehittymisen suuntaa. Eteenpäin kehittyneen debuggaamisen yhteensopivuutta videopeleihin pohditaan. Lopuksi puhutaan käänteisen debuggaamisen hyödyistä pelisuunnittelun työkaluna.

Luvussa 8 kerrataan tutkimuskysymykset ja kootaan yhteen havainnot työstä. Tuotetun pelin toteutusta verrataan luvussa 1 esitettyihin tavotteisiin.

# Luku 2

# Käänteinen debuggaus

## 2.1 Ohjelmistojen debuggaus

Kun ohjelmassa huomataan virhetila eli bugi, sitä voidaan tarkastella ajamalla ohjelma samoilla syötteillä uudelleen niin, että bugi toistuu. Paremman ymmärryksen avulla bugi voidaan korjata. Bugien korjaamista kutsutaan debuggaamiseksi. Lähestyttävä ja yksinkertainen debuggaustekniikka on muuttaa ohjelman logiikkaa niin, että sen suoritus tuottaa luettavia debuggausta auttavia arvoja, joissa on mahdollisesti bugin aiheuttamiseen viittaavaa tietoa.

Bugien tarkastelun tukena voidaan käyttää virheenjäljitysohjelmaa eli debuggeria. Debuggerilla pystytään lisäämään ohjelman lähdekoodiin pysäytyspisteitä (*breakpoints*). Pysäytyspisteiden kohdalla ohjelman suoritus pysähtyy ja ohjelman tilaa voidaan tarkastella debuggerissa. Debuggeri tarjoaa työkalut kontrolloida sovelluksen suoritusta lähdekoodin rivin tai lauseen tarkkuudella.

Ohjelmointiympäristöjen tarjoamissa debuggereissa muuttujien arvoja pystytään tarkastelemaan ja toteutuksesta riippuen, voidaan myös tehdä muutoksia ajettuun logiikkaan kesken ohjelman suorituksen. Pysäytyspisteet merkitään tietyille lähdekoodin riveille debuggerin ymmärtämällä muodolla. Debuggeri kiinnitetään ajettavaan sovellukseen ja se

pysäyttää logiikan suorituksen asetettuihin pysäytyspisteisiin ja luo esityksen sovelluksessa käytetyistä muuttujista ja muistin tilasta.

Jos ohjelmiston suoritus riippuu lähtöasetusten lisäksi muista ajonaikaisista tekijöistä, voi bugin toistaminen olla epävarmaa. Ajonaikaisia arvaamattomasti vaikuttavia tekijöitä voi olla käyttöjärjestelmä ja muut ohjelmat, tietokoneen komponentit ja muut tietokoneeseen yhteydessä olevat laitteet sekä sovelluksen käyttäjä.

## 2.2 Käänteisen debuggauksen määritelmä

Tavanomaisesti debuggauksessa ohjelmaa suoritetaan tarkasti asetettuihin pysätyspisteisiin asti, jonka jälkeen ohjelman tilaa tarkastellaan. Debuggausprosessi alkaa vasta kun sovellus on päätynyt bugiin ja debuggausta suorittava kehittäjä on suunnitellut, miten hän alkaa tutkimaan bugia. Debuggausta kutsutaan sykliseksi (*cyclical debugging*), kun ohjelmaa suoritetaan toistuvasti uudelleen, että bugi toistuu [4]. Suorituksen epävarmuutta lisäävät ulkopuoliset tekijät, jotka pitää myös ymmärtää, että bugi voidaan toistaa. Pahimmillaan nämä ovat epädeterministisiä syötteitä, joita ei pystytä selvittämään tai toistamaan luotettavasti. Epävarmoja syötteitä voivat olla esimerkiksi käyttäjän tekemät hiiren klikkaukset ja nappien painamiset.

Käänteinen debuggaus usein kuvataan tekniikkana, jossa ohjelmiston debuggaus aloitetaan sovelluksen kaatuneesta tilasta. Ihanteellisesti käänteisessä debuggauksessa bugin olemassaolosta ei tarvitse tietää etukäteen. Kun bugi tapahtuu, aloitetaan ohjelman tarkastelu sovelluksen virheellisest tai kaatuneesta tilasta. Debuggauksessa bugia voidaan tarkastella vaihe vaiheelta tavallisen debuggauksen tavoin, mutta ohjelmiston suoritusta voidaan myös peruuttaa aikaisempiin vaiheisiin. [4]

## 2.3 Käänteisen debuggauksen toteutustavat

Käänteisestä debuggamisesta on olemassaolevia toteutuksia. Ne tuovat parannuksia olemassaoleviin debuggereihin tai ne ovat kokonaan uusia, käänteiseen debuggaukseen suunniteltuja. Tässä diplomityössä tarkastellaan lyhyesti seuraavia *nauhoita ja toista* -debuggauksen toteutuksia:

- Mozillan *RR*
- Microsoftin Time Travel Debugging (*TTD*)

Lisäksi on olemassa muutamia työkaluja, joilla pystytään nauhoittamaan ja toistamaan pelin tai sovelluksen suoritus tarkasti. Seuraavia työkaluja tarkastellaan, koska ne sisältävät samanlaisia ominaisuuksia, jotka muistuttavat käänteistä debuggaamista:

- Pelikonsoleista tehdyt emulaattorit
- *Redux*-kirjasto web-sovelluksissa

Käänteistä debuggaamista voidaan kutsua monilla eri nimillä. *Nauhoita ja toista* -termiä ja *aikamatkustus debuggaus* -termiä käytetään keskenään vaihtokelpoisesti. Käänteinen debuggaaminen voidaan kategorisoida toteutuksen mukaan. Sovelluksen toistaminen perustuu alkuperäisen suorituksen tallentamiseen syöte- ja jäljitystiedostoiksi.

### 2.3.1 Syöte- ja jäljitystiedostot

Tiedoston muodostuksessa voidaan käyttää kahta vastakkaista tapaa. Käänteisen debuggaamisen tueksi muodostettu tiedosto voi sisältää puhtaasti vain ohjelman suorituksesta kaapattuja syötteitä tai se voi sisältää ohjelman tilasta tehtyjä tallennuksia. Pelkkiä syötteitä sisältävää tiedostoa kutsutaan syötetiedostoksi (*input file*). Syötetiedoston avulla voidaan täysin deterministisen sovelluksen suoritus toistaa tarkasti, niinkuin se tapahtui, kun syötetiedosto muodostettiin. Syötetiedoston muodostuksessa kerätään vain suorituksessa tapahtuvat epädeterministiset syötteet. Tiedostoa käytetään bugin toistamiseen simuloi-

malla epädeterministiset syötteet ohjelman suorituksessa. Syklisen debuggaamisen näkökulmasta bugin toistamisen epävarmuus on ratkaistu käyttämällä syötetiedostoa.

Käänteisessä debuggauksessa käytetty tiedosto, joka kuvaa sovelluksen suorituksen pohjalta tallennettuja tiloja, kutsutaan jäljitystiedostoksi (*tracefile*). Kun jäljitystiedosto kuvaa sovelluksen tilan muuttumista alusta loppuun, ei sovellusta tarvitse enää suorittaa uudelleen. Tarkan jäljitystiedoston avulla voidaan tehdä päätelmiä sovelluksen suorituksen kohdista, jotka johtivat bugiin. Jos jotain osaa sovelluksen suorituksesta ei ole tallennettu, tietoa ei pystytä luomaan jälkeenpäin.

Jäljitystiedoston vahvuuksiin kuuluu sen riippumattomuus alkuperäisestä suoritusympäristöstä. Käänteinen debuggaus jäljitystiedostolla on sovelluksen suorituksen esityksen pohjautuvaa debuggaamista. Koska sovellusta ei enää suoriteta ja vain tarkastellaan jäljitystiedostoon merkittyjä tallennuksia sovelluksen tilasta, oheislaitteiden toimintaan perustuvat bugit voivat olla vaikeita ymmärtää. Riippumattomuus suoritettavasta ohjelmasta edellyttää sitä, että jäljitystiedosto ei erottele sovelluksen epädeterministisiä osia sen deterministisistä osista, vaan siinä pyritään tallentamaan kaikki sovelluksen yksityiskohdat.

Syötetiedostoon pohjautuva käänteinen debuggaus on riippuvainen sovelluksen suorittamisesta. Koska syötetiedosto sisältää tiedot vain siitä, miten sovellusta suoritetaan, kaikki muu tieto pitää lukea sovelluksesta itsestään, kun bugia toistetaan. Jos kaikkea epädeterminismiä ei ole tallennettu syötetiedostoon, bugia ei voida toistaa varmasti. Jos syötetiedostoon pohjautuvassa käänteisessä debuggauksessa ei voida olettaa, että sovelluksen suorituksen deterministiset osat toimivat samalla tavalla muissa suoritusympäristöissä, niin käänteistä debuggausta ei voida suorittaa kuin sillä koneella, jossa bugiin johtavat syötteet tallennettiin.

Syötetiedostoa voidaan käyttää bugin toistamiseen, minkä pohjalta voidaan tuottaa jäljitystiedosto. Jäljitystiedosto pitää sisällään sovelluksen tilan muutokset sen suorituksen ajalta, mutta siirtyminen tilojen väliltä ei välttämättä ole yksiselitteistä. Jäljitystiedostosta

ei voida päätellä, miten tilojen välillä siirryttiin ja siksi siitä ei voida muodostaa syötetiedostoa.

Syötetiedoston näkökulmasta jäljitystiedostoa voidaan ajatella käänteisen debuggauksen tila-aika optimointina. Siirtyminen tilojen välillä nopeutuu, mutta tilavaatimukset debuggaamiselle kasvavat. Syötetiedoston tueksi muodostettu jäljitystiedostoa voidaan pienentää tallentamalla siihen vain tietyt ajanhetket. Ajanhetkien välissä olevat tilat voidaan saavuttaa suorittamalla sovellusta eteenpäin käyttäen syötetiedostoa.

### 2.3.2 Nauhoita ja toista -debuggaaminen

*RR* on Mozillan kehittämä työkalu käänteiseen debuggaamiseen. Mozilla kutsuu prosessia *nauhoita ja toista* -debuggaamiseksi. Mozillan kehittäjät tunnistivat, että ohjelmistotuotannossa käytetään paljon aikaa deterministisen bugien debuggaamiseen. Käänteinen debuggaaminen on tehokas työkalu epädeterministisen bugien ymmärtämiseen ja sitä haluttiin käyttää Mozillalla tai ymmärtää, miksi sellaista järjestelmää ei voida tuottaa. [5]

*RR* ei edellytä muutoksia ohjelmakoodiin ja toimii muuttamattomalla Linuxin ytimellä (*kernel*). Sen toiminta perustuu x86-64 -arkkitehtuurin suorittimiin, joissa on luotettavat laitteiston suoritusta seuraavat laskurit (*performance counter*). Luotettavat laskurit ovat osa Intelin *Sandy Bridge* ja sitä uudempia suoritinarkkitehtuureja. *RR* tarjoaa laajasti käytettyyn *GDB*-debuggeriin (*the GNU project debugger*) käänteiseen debuggaamiseen soveltuvan debuggausagentin. *RR* luotiin kaupallisten sovellusten käänteiseen debuggaamiseen, joista merkittävin on Mozillan tuottama verkkoselain, *Firefox*. Vaikka *Firefox* on *RR*:n ensisijainen käyttökohde, voidaan sitä käyttää muihinkin sovelluksiin, joita debugataan *GDB*:llä. [5]

Microsoftin *TTD* perustuu yksityiskohtaisen jäljitystiedoston (*tracefile*) nauhoittamiseen [6]. *TTD* on uusi ominaisuus Windowsille tuotetussa *WinDbg*-debuggerissa [7]. Vaikka *TTD*:llä on paljon samankaltaisia piirteitä *RR*:n kanssa, ne eroavat toteutukseltaan toistaa sovelluksen suoritus. *RR* seuraa tarkasti sovelluksen suoritusta hyödyntämällä Linuxin

ydintä ja sen tarjoamia palveluita. *RR*:n nauhoituksissa voidaan tunnistaa epädeterministiset syötteet ja erottaa ne deterministisestä ohjelmalogiikasta. Uudessa suorituksessa syötteet voidaan toistaa tarkasti, jolloin bugi saadaan toistettua.

*TTD*:llä pystyy käänteisesti debuggaamaan niitä ajettavia tiedostoja, mitä *WinDbg*:llä pystytään debuggaamaan. Kun sovellusta ajetaan, lisätään sen alhaisen tason ohjelmalogiikkaan uusia käskyjä (*code instrumentation*), jotka tuottavat kiinnitetylle debuggerille informaatiota ohjelman suorituksesta. *TTD*:n etu *RR*:ään nähden on sen kyky seurata rinnakkain suoritettavia säikeitä niin, että niiden suoritusjärjestys pystytään myös esittämään käänteisessä debuggaamisessa. [6]

Koska *RR* ei muuta ajettavia sovelluksia, vaan turvautuu suoritinarkkitehtuuriin ja Linuxin ytimeen, kärsii se puuttuvista rinnakaisen suorituksen seurantaan tarvituista ominaisuuksista. Sen ratkaisu monisäikeisen sovelluksen suorittamiseen on ajaa ohjelmaa yksi säie kerrallaan. Nykyisissä käyttöjärjestelmissä tapa, millä laskenta-aikaa annetaan sovelluksille, johtaa mahdollisesti säikeiden väliseen kilpailutilanteeseen (*race condition*). Kilpailutilanteet ovat sovelluksen näkökulmasta epädeterministisiä ja ne pitää myös tallentaa. Mozillan *RR*:n toteutuksessa sovelluksen suorituksen kaappausta yksinkertaistetaan estämällä säikeiden rinnakkainen suoritus [5].

*TTD*:ssä jäljitystiedostolla ja ohjelman lähdekoodista muodostetuilla debuggaus merkeillä (*debugging symbol*) voidaan bugin toistumista havainnollistaa aivan kuin sovellusta ajettaisiin uudelleen. Todellisuudessa sovellusta ei debuggauksen yhteydessä enää suoriteta, vaan esitetyt tiedot haetaan jäljitystiedostosta. Koska sovellusta ei suoriteta oikeasti, sen suoritusaikaiset sivuvaikutukset jäävät myös ilmaisematta. Sivuvaikutuksiin lukeutuu tiedostojen kirjoittaminen ja tietyissä tilanteissa myös käyttöliittymän tai muun sovelluksen grafiikan esittäminen ja äänien soittaminen.

*TTD*-demossa muutaman virkkeen kirjoittaminen Muistio-sovellukseen tuottaa yli 200 megatavun jäljitystiedoston. Demossa käänteinen debuggaus *WinDbg*:llä ei pysty esittämään muistion käyttöliittymää, johon virkkeet alunperin kirjoitettiin. Tekstit saadaan nä-

kymään käyttäen debuggauksen tueksi luotua JavaScript-komentosarjaa, joka osaa käyttää jäljitystiedostoa niin, että tekstit saadaan esitettyä *WinDbg*:n käyttöliittymässä. Koska jokainen kaapattu ajanhetki on verrannollinen tavallisen debuggauksen pysäytyspisteeseen, suorittamisen toistaminen edellyttää, että dataa tallennetaan paljon. Jäljitystiedostoon tallennetun suorituksen ulkopuolelle jäänyttä logiikkaa ei voida tutkia. [7]

Työskentelytapa *nauhoita ja toista* -debuggaamisessa tapahtuu havaitsemalla bugin olemassaolo debuggerin ollessa poissa käytöstä, jonka jälkeen debuggerin päällä ollessa bugi toistetaan. Bugin toistossa debuggeri kerää tietoja kaikilta suorituksen ajanhetkiltä. *RR*:n ja *TTD*:n käyttö muistuttaa syklistä debuggamista, mutta ratkaisee bugin toistamisen epävarmuuden tallentamalla toistetun bugin, ettei sitä tarvitse tuottaa kolmatta kertaa.

Käänteisen debuggaamisen toteuttaminen *nauhoita ja toista* -lähestymisellä on myös hyvä alusta ohjelmalliseen bugien etsimiseen. Jäljitystiedostoa voidaan ajatella tietokantana, jolla voidaan esimerkiksi ilmaista, mitkä osat ohjelmasta lukevat mistäkin muistin kohdasta. Tekemällä hakuja, jotka ilmaisevat muistin lukemista näiltä paikoilta, voi olla tehokas tapa ymmärtää bugeja paremmin. *TTD*:ssä jäljitystiedostoon perustuvaa hakulogiikkaa voidaan jo hyödyntää [7]. Hyvän jäljitystiedoston muodostaminen aiheuttaa suorituksenaikaisia haasteita, erityisesti jäljitystiedoston suuresta kasvunopeudesta ja varsinaisen ohjelman hidastumisesta. Koska jäljitystiedosto kasvaa nopeasti, debuggerin kiinnittäminen sovellukseen on järkevää vasta, kun bugi osataan toistaa nopeasti.

# Luku 3

# Videopelien houkutustilat ja pikapelaaminen

Pelihallien kultainen aikakausi, joka alkoi vuonna 1978 ja jatkui vuoteen 1983, toi perinteisten kolikkopelien rinnalle videopelejä. Pelihallien videopelit olivat tarkoitukseensa rakennettuja tietokoneita. Ne muodostuivat piirilevystä, jossa oli pelin logiikka, kuvaputkimonitorista, josta peliä katsottiin ja kontrolleista, joilla videopeliä pelattiin. Kontrollien lisäksi pelihallin kabineteissa oli kolikon syöttöaukko, joita käyttämällä pystyi ostamaan pelikertoja.

Kultakauden parhaimpia videopelejä olivat [8]:

- Space Invaders (Taito, 1978)
- Asteroids (Atari, 1979)
- Pacman (Namco, 1980)
- Galaga (Namco, 1981)
- Donkey Kong Jr. (Nintendo, 1982)

Pelihallissa olevien videopelien tuottoa mitattiin niiden kerääminen kolikoiden määrässä. Videopelien vaikeusasteella pystyttiin vaikuttamaan pelikertojen pituuteen, mutta pelaajien ja pelikertojen määrä perustui pelin laatuun. Pelaajien huomiota herättävät seikat pelihallin videopelissä olivat kabinetin ulkoasu ja muoto, pelin kontrollit ja peli itse.

Peli itsessään pystyi esittämään itseään, jos se oli ohjelmoitu niin. Videopelin itseään pelaavaa tilaa kutsutaan videopelin houkutustilaksi.

Pelihalliessa syntyneet odotukset vaikuttivat myös kotikonsolien peleihin. Yksityiskäyttöön myydyt pelikonsolit pyrkivät tarjoamaan pelikokemuksen, joka vastasi pelihallien videopelejä. Konsoleille tehdyt videopelit antavat pelaajalle mahdollisuuden kehittyä pelin pelaamisessa ilman rahallista riskiä. Varhaisten Nintendo Entertainment System -pelikonsolin videopelien vaikeusasteesta syntyi termi *Nintendo vaikea* (*Nintendo hard*). Pelihalleissa syntyneiden odotusten lisäksi konsolipelien huomattava hinta pelihallissa pelattavan videopelin pelikertaan nähden, antoi hyvän perustelun ajatukselle, että peliä tulisi pelata kauan. Taustalla on mahdollisesti ajatus, että uudelleenpelattavuuden arvo heikentyisi, jos pelaaja pääsisi pelin läpi.

Houkutustilojen olemassaolo ilmeni myös kotikonsolien videopeleissä. Konsolipeleissä olevat houkutustilat ovat mielenkiintoisia, koska konsolien ansaintamalli poikkeaa pelihallin videopelien ansaintamallista. Houkutustilaa voidaan ajatella opetusvälineenä, jossa esitetään videopelin sisältöä ja pelaamista. Näin pelaajan, jolle peli on vielä vieras, on helpompi ymmärtää, onko peli sellainen, mitä hän haluaisi pelata. Pelihallipelejä merkittävämpi rahallinen riski on jo otettu, kun konsolipeli ostetaan ja houkutustila voidaan kokea vasta kun peliä pelataan. Ostopäätökseen on voinut vaikuttaa pelin pakkaus. Pakkaus ei välttämättä edusta peliä kovinkaan tarkasti, jos siihen ei ole sisällytetty kuvakaappauksia. Kuvakaappausten sisällyttäminen ei edellytä houkutustilan sisällyttämistä peliin.

Houkutustila on hyödyllinen konsolipeleille, jos pelit ovat esillä julkisella paikalla. Sekä peliä myyvä liike että peli itse keräävät itseensä huomiota, mikä voi johtaa myyntiin. Vaikeissa peleissä houkutustila voi tarjota pelaajalle vilauksia pelin myöhäisemmistä osista tai pelityyleistä, joita hän ei ole vielä kokeillut. Houkutustilaa voidaan myös käyttää esittelemään pelin pelisilmukkaan yhteensopimattomia asioita, kuten esimerkiksi pelin tarinaa.

Varhaiset videopelit ovat suosittuja pikapelaamisessa (ks. luku 3.3). Aikaisista pelikonsolisukupolvista on tuotettu sovelluksia, jotka pystyvät simuloimaan pelikonsolien laitteistoa. Sovelluksia kutsutaan emulaattoreiksi (*emulator*). Emulaatoreilla pystytään pelaamaan alkuperäisistä peleistä kopioituja ROM-tiedostoja (*Read only memory*). Sovelluksen muodossa olevat pelikonsolit ja tiedostojen muodossa olevat pelit ovat videopelin säilyttämisessä arvokas työkalu. Emulaattorien olemassaolo on edistänyt vanhojen videopelien saatavuutta ja niiden ympärille muodostuneita pikapelaamisen yhteisöjä.

Emulaattorit eivät ole välttämättä tarkkoja simulaatioita alkuperäisistä pelikonsoleista. Pelikonsolien emulaattoreissa priorisoidaan pelien pelattavuutta, joka tekee suorituskyvystä tärkeämpää kuin tarkkuudesta. Emulaattoreihin voidaan ohjelmoida peliohjaimen syötteitä, jolloin peli pystyy pelaamaan itseään. Puhtaasti peliohjaimen syötteillä luotu houkutustilaa muistuttava peluu kuvaa, kuinka determinististä emulaattoreissa ajetut videopelit ovat. Samat syötteet voidaan suorittaa ohjelmoitavalla peliohjaimella, oikeassa konsolissa. Tälläistä testaamista kutsutaan konsolivahvistukseksi ja se osoittaa, kuinka samankaltainen emulaattori on verrattuna todelliseen pelikonsoliin.

## 3.1 Houkutustilan toteutustavat

Houkutustila on videopeliin toteutettu tila, joka aktivoituu, kun peliä ei pelata. Houkutustilan tavoite on houkutella seuraava pelaaja pelaamaan. Ennen videopelien yleistymistä flippereissä hyödynnettiin niissä olevia elektronisia elementtejä mm. havainnollistamaan, mistä pistemäärästä saa uuden kuulan ja mitä pelikerta maksaa. Videopelin houkutustila voidaan toteuttaa monilla eri tavoilla. Tapoihin lukeutuu nauhoitettu pelaaminen, synteettinen pelaaminen, audiovisuaalinen esitys pelin tavoitteista tai muu laitteiston suorituskyvyn demonstrointi. Kiinnostava käynnistysvalikko tai taustamusiikki voidaan myös tulkita houkutustilaksi.

Nauhoitettu pelaaminen tarkoittaa aikaisemmin pelattua pelikertaa, joka nauhoitettiin syötteiksi, joita käyttäen peli saadaan toistamaan peluu. Nauhoitettu liitetään osaksi pelin ROM:mia, jossa pelin toimintalogiikka on määritetty. Nauhoitettu pelikerta on tehokas tapa opettaa ympärille kerääntyneille, miten peliä pelataan. Nauhoitettu pelaaminen edellyttää, että peli toimii täysin deterministisesti.

Synteettinen pelaaminen tarkoittaa tekoälypelaajan pelaamista. Peleissä, joissa tekoäly pystyy pelaamaan ihmisen tavoin, voidaan nauhoitetun pelaamisen sijaan käyttää synteettistä pelaamista. Synteettisellä peluulla peliin voidaan toteuttaa houkutustila niin, että se toistaa itseään vähemmän. Ihanteellisesti pienistä lähtötilan muutoksista syntyy täysin erilainen pelikerta. Koska deterministisesti toimiva peli tuottaa aina samalla tavalla käyttäytyvän tekoälyn, voidaan houkutustilaa parantaa sisällyttämällä epädeterminismiä tuottava osa, joka muuttaa houkutustilan toimintaa. Yksinkertainen epädeterminismiä tuottava osa on reaaliaikaa mittaava kello. Synteettinen peluu soveltuu erityisesti tappelupeleihin ja ajopeleihin, joissa tekoäly toimii puuttuvien pelaajien sijaisina.

Muu audiovisuaalinen esitys voi yksinkertaisimmillaan näyttää piste- tai aikahyökkäyslistaa. Aikahyökkäys on pelimuoto, jossa pelaaja pisteiden sijaan tavoittelee mahdollisimman nopeaa läpäisyaikaa. Peli voi kertoa itsestään tai tarinastaan kuvilla, videolla tai äänellä. Nauhoitettua tai synteettistä pelaamista voidaan täydentää audiovisuaalisin keinoin tuotetuilla ohjeilla, joista pelaaja saa esimerkiksi paremman havainnollistuksen, miten pelin kontrollit toimivat. Audiovisuaalinen esitys ja synteettinen pelaaminen eivät riipu videopelin deterministisyydestä.

## 3.2 Bugien dokumentointi syötetiedostolla

Ohjelmistokehityksessä hyödynnetään virheenseurantaohjelmia ja testejä. Testeissä ajetaan ohjelman logiikkaa ja verrataan sen tuottamaa ulostuloarvoa ja sivuvaikutuksia oletettuun lopputulokseen. Testejä, jotka testaavat ohjelmistomoduulin toiminnallisuutta mah-

dollisimman suppeasti kutsutaan yksikkötesteiksi. Testiautomaatiolla voidaan ajaa yksikkötestejä samalla kun ohjelmistoa tai peliä kehitetään. Testiautomaatio ilmoittaa, jos korjattu bugi ilmetä uudelleen. Korjattuja, uudelleen ilmentyviä bugeja kutsutaan regressioiksi.

Testien laatiminen voi olla haastavaa, jos testattavan logiikan vaikutukset ovat moninaiset. Mahdollisuus bugeihin kasvaa, kun logiikan määrä kasvaa. Useiden ohjelmistomoduulien yhteensopivuuden testaamista kutsutaan integraatiotestaamiseksi. Kun testejä tehdään regressioiden seuraamiseen, on ohjelmistokehitys reaktiivista. Testin muodostaminen on helpompaa, kun bugia pystyy tutkimaan käänteisellä debuggaamisella. Jäljitystiedosto (luku 2.3.1) auttaa bugin ymmärtämiseen, minkä perusteella testi voidaan muodostaa. Testiin sisällytetään sellaiset raja-arvot, mitkä aiheuttavat bugin, mutta ne ovat pohjimmiltaan tulkinta jäljitystiedostoon tallentuneista syötteistä.

Syötetiedostoon perustuvaa käänteistä debuggausta voidaan myös hyödyntää testien muodostamisessa. Erona jäljitystiedostoon perustuvassa käänteisessä debuggauksessa on se, että jäljitystiedostoa käytettäessä ohjelmistoa suoritetaan. Sovelluksen logiikkaan voidaan kiinnittää väittämiä tarkistavia lausekkeita (*assert statement*). Niitä käytetään testeissä ja ne sisältävät testatusta logiikasta toivottuja arvoja. Testiautomaatiossa yksikkötestit ovat tapa keskittää testaaminen pois tavallisesta ohjelmalogiikasta. Keskitettyinä testit ovat yksinkertaisempia ajaa kuin sovellus itse, eivätkä lisää lähdekoodin kompleksisuutta. Väittämiä tarkistavat lausekkeet ovat myös hyödyllisiä sovelluksen logiikan testaamisessa, kun testiautomaatio osaa suorittaa varsinaista ohjelmaa syötetiedoston avulla.

Syötetiedostolla voidaan myös käyttää samaa lähestymistä kuin jäljitystiedostolla, missä bugia vain tutkitaan ja opituista asioista johdetaan testi. Regression tunnistaminen useita eri raja-arvoja käyttäen, voi tarjota paremman suojan kuin vain pelkkä syötetiedostoon perustuva bugin toisto. Erityinen piirre syötetiedostoilla on se, että ne voidaan tallentaa levylle ja myös esittää toiselle kehittäjälle toisessa tietokoneessa, jos tietokoneen

arkkitehtuuri on sama [9]. Syötetiedostojen jakaminen on yleistä pikapelaamisessa, jossa pelien alustat ovat emuloituja.

## 3.3 Pikapelaaminen

Pikapelaaminen (*speedrunning*) on videopelaamisen muoto, jossa pyritään läpäisemään videopeli mahdollisimman nopeasti. Pelin läpäiseminen määritetään pelikohtaisesti ja pelaamiseen voi sisältyä pelin ulkopuolella määritettyjä sääntöjä. Pikapeluun sääntöjä voidaan verrata monien urheilulajien eri lajimuotoihin. Pikapelaamisen lajimuotoja kutsutaan kategorioiksi (*category*).

Tyypillisesti pikapeluussa tavoitellaan mahdollisimman nopeaa etenemistä pelin alusta pelin loppuun. Loppuun pääseminen voidaan suorittaa ohittamalla mahdollisimman paljon pelin sisällöstä tai käänteisesti suorittamalla kaikki pelin sisältö. Pelin sisällön tehokas ohittaminen yleensä perustuu pelin etenemisjärjestyksen rikkomiseen (*sequence break*). Etenemisjärjestyksen rikkominen voi tarkoittaa pelin myöhäisempiin vaiheisiin tarkoitettujen esteiden ohittamista pelissä olevien bugien avulla. Bugien salliminen on yksi säännöistä, mitkä määrittävät pikapelaamisen kategorian.

Apuohjelmia hyödyntävä tai kokonaan tietokoneohjelman suorittava pikapelaaminen luokitellaan työkaluavusteiseksi pikapelaamiseksi (*tool-assisted speedrun, TAS*). Työkaluavusteisen pikapelaamisen vastakohta on ihmisen pelaama reaaliaikahyökkäykseksi (*real-time attack, RTA*) kutsuttu kategoria.

Pikapelaamista voidaan verrata sävellyksiin ja muusikoihin. Pikapelaamisessa reaaliaikahyökkäyksiin suunnitellaan reitti (*route*), jossa pelaaja pyrkii pysymään parhaansa mukaan. Sävellysten esittämisessä myös tavoitellaan tietynlaista tulkintaa. Työkaluavusteista pikapelaamista voidaan verrata tietokoneen soittamaan musiikkiin, jossa varsinaisia instrumentteja ei välttämättä ole tai esitettyä sävellystä ei ihminen pystyisi soittamaan.

Vaikka reaaliaikahyökkäykset ja työkaluavusteiset pikapeluut eivät kilpaile keskenään, tapahtuu niiden välillä vuorovaikuttamista. Työkaluavusteisesti voidaan todentaa, miten soveltuvia eri reitit ovat reaaliaikahyökkäyksiin. Reaaliaikahyökkäyksten spontaanisuus voi törmätä täysin odottamattomiin bugeihin, jotka voivat antaa uusia lähestymissuuntia työkaluavusteiseen pikapelaamiseen.

Reaaliaikahyökkäyksten todentaminen tapahtuu pelistä saadulla videokaappauksella. Suorituksia arvioiva raati käyttää videoita nopeimpien pikapelaajien listan ylläpitämisessä. Videopelien laadunvarmistuksen näkökulmasta video voi olla todiste siitä, että bugi on olemassa ja antaa vihjeitä siitä, miten sen voi toistaa.

Pelaajan syötteet voidaan nauhoittaa syötetiedostoksi (*input file*). Jos pelin alustasta on tarkka emulaattori, voidaan syötteet synkronoida siihen ja toistaa peli tarkasti. Emulaattorissa syötetiedostolla voidaan bugiin perehtyä visuaalisen tarkastelun lisäksi myös katsomalla pelin tilaa emuloidun alustan muistissa. Syötetiedostoa manipuloimalla voidaan bugin toistamiseen tarvittuja raja-arvoa selvittää tarkemmin.

# Luku 4

# Arkkitehtuuri web-sovelluksissa

Web-sovellukset ovat verkkoselaimessa suoritettavia sovelluksia. Erona tavallisiin verkkosivuihin, web-sovellukset toimivat vuorovaikutuksessa käyttäjän kanssa. Web-sovellusten toiminnalisuuden toteuttamiseen käytetään JavaScript-kieltä, jolla voidaan manipuloida tavallisia verkkosivuja toimimaan älykkäämmin. Web-sovelluksesta riippuen toiminnalisuus voi olla riippuvainen palvelimesta. Web-sovelluksen mahdollinen logiikan jako on, että selaimessa suoritetaan käyttöliittymään liittyvää logiikkaa ja palvelimella suoritetaan sovellukseen tilan manipulointiin liittyvää logiikkaa.

Web-sovellusten etu on yhteensopivuus niillä alustoilla, joilla on nykyaikainen verkkoselain millä pystytään suorittamaan web-sovellusta. Koska verkkoselaimia on toteutettu hyvin laajalle laitekannalle, web-sovellukset ovat erityisen saavutettavia. Saavuttavuus kuitenkin vaatii web-sovellusten kehittäjiltä näkemystä, miten monilla eri laitteilla ja eri olosuhteissa sovellusta tullaan käyttämään. Työpöytäkoneen, kannettavan tietokoneen ja mobiililaitteen väliset erot näyttöjen koossa ja kannettavuudessa on merkittävä.

Jatkuvasti laajentuva laitekanta ja verkkoselainten viive uusimpien web-standardien omaksumisessa asettavat rajoituksia web-sovellusten suunnitteluun sekä toteuttamiseen. Ikivihreiksi luokitellut selaimet ovat sellaisia joilla automaattisiin päivityksiin on tehty merkittäviä panostuksia, jotta loppukäyttäjillä olisi pääsy uusiin ominaisuuksiin. Päivittäminen uusiin selainversioihin suurella mittakaavalla ei kuitenkaan tapahdu välittömästi.

## 4.1 Nykyaikainen JavaScript

Web-kehityksessä ei voida kuitenkaan olettaa, että uusimpia verkkoselainten ominaisuuksia voidaan ottaa käyttöön, heti kun ne julkaistaan. Merkittävä määrä käyttäjistä voi edelleen käyttää verkkoselaimen vanhaa versiota, missä ominaisuutta ei tueta. Web-sovellusten toteuttamisessa osalle uusista ominaisuuksista voidaan kumminkin toteuttaa riittävän tarkasti taaksepäin yhteensopiva versio. Paikattu ominaisuus voidaan ottaa käyttöön jos web-sovellus on ladattu verkkoselaimeen, jossa uutta ominaisuutta ei vielä tueta. Taaksepäinyhteensopivuuden lisääminen web-sovellukseen koodianalyysin ja automaation avulla synnytti uuden työskentelymallin, jossa hyödynnetään kääntäjää (*transpiler*).

Kääntämällä uusimpaa JavaScript version mukaista koodia vanhempaan, laajasti tuettuun JavaScriptin versioon, mahdollistuu web-sovellusten kehityksessä valinta käyttää uusimpaa JavaScriptia. Ikivihreiden verkkoselainten ja kääntäjien avulla uusimpien web-standardien käyttö web-sovellusten kehityksessä on helpompaa. Yksi käytetyimmistä kääntäjistä on Babel. Se luotiin alunperin muuttamaan ECMAScript 6 standardin mukaista JavaScriptiä ECMAScript 5 standardin mukaiseksi [10]. Babelin avulla voidaan myös kehittää omia ominaisuuksia JavaScript-kieleen. Virallisia, vielä julkaisemattomia ominaisuuksia on toteutettu kääntäjiin ennen niiden julkaisua. Tulevaisuuden JavaScript-kielen versiosta puhutaan termillä ECMAScript Next. Kääntäjien hyöty ECMAScript-standardin kehityksessä on ollut mahdollistaa niihin ehdotettujen ominaisuuksien testaamista oikeissa ohjelmissa, joka on nopeuttanut standardin kehittämisessä (ks. taulukko 4.1). [10]

## 4.2 Palvelinohjelma ja asiakasohjelma

Web-sovellukset muodostuvat palvelin- ja asiakasohjelmista. Palvelinohjelmaa suoritetaan palvelinkoneessa, joka palvelee useita siihen yhdistäviä asiakasohjelmia. Asiakasohjelma on verkkoselaimessa suoritettava sovellus, joka myös muodostaa käyttäjälle käyttö-

| Versio | Julkaisu | Nimi |
|---|---|---|
| 1 | Kesäkuu 1997 | |
| 2 | Kesäkuu 1998 | |
| 3 | Joulukuu 1999 | |
| 4 | Hylätty | |
| 5 | Joulukuu 2009 | |
| 5.1 | Kesäkuu 2011 | |
| 6 | Kesäkuu 2015 | ECMAScript 2015 (ES2015) |
| 7 | Kesäkuu 2016 | ECMAScript 2016 (ES2016) |
| 8 | Kesäkuu 2017 | ECMAScript 2017 (ES2017) |
| 9 | Kesäkuu 2018 | ECMAScript 2018 (ES2018) |
| 10 | Kesäkuu 2019 | ECMAScript 2019 (ES2019) |

Taulukko 4.1: ECMAScript standardien julkaisut [11]

liittymän. Palvelinohjelma voi yhdistää muihin palvelinkoneen sovelluksiin, esimerkiksi tietokantaan.

Web-sovellus käynnistyy, kun selaimella navigoidaan web-sovelluksen verkko-osoitteeseen. Palvelin lähettää web-sovelluksen vastaamalla sivunlatauspyyntöön hypertekstimerkkaus-kielellä (*hypertext markup language, HTML*). Vastauksessa on ilmaistu mistä muut sovelluksen tarvitsemat tiedostot ladataan. Web-sovellus muodostuu HTML:n lisäksi tyyliohjeista (*cascading style sheets, CSS*) ja JavaScriptilla kirjoitetusta logiikasta.

Palvelin lähettää web-sovelluksen tilan joko alustavassa sivulatauksessa tai asiakasohjelma itse lataa sen käyttämällä tilan synkronointiin tehtyä logiikkaa. Sivulatausten sijasta asiakasohjelma voi myös tehdä pyyntöjä palvelimen ohjelmointirajapintaan (*application programming interface, API*) ja vastausten perusteella manipuloida käyttöliittymän sisältöä.

## 4.3 Monen sivun sovellukset

Asiakasohjelman käyttöliittymä voidaan toteuttaa perustumaan HTML-muotoisiin vastauksiin. Sovellus jossa sivujen välillä navigoidaan lataamalla kokonaan uusi sivu, kutsutaan monen sivun sovellukseksi. Kun palvelin lähettää vastauksia asiakasohjelmalle ne ovat HTML-muodossa ja ilmaisevat asiakasohjelman käyttöliittymän nykyisen tilan. Jotta palvelinohjelma pystyy vastaamaan tarkasti, sen pitää kyetä erottamaan pyynnöt käyttäjäkohtaisesti ja laskemaan käyttöliittymän esitysmuoto ja sen tila vastaukseen.

Vaihtoehtoinen lähestyminen web-sovelluksen toteuttamiseen on siirtää sovelluksen tilan esittämisen vastuu palvelinohjelmasta asiakasohjelmaan. Käyttöliittymästä tulee älykkäämpi, kun logiikka huomioi sovelluksen tilan alueittain ja tietää, mitkä toiminnot muuttavat tilaa milläkin tavalla. Kun käyttöliittymä olettaa palvelimen vastauksen sisällön ja välittömästi esittää sen seuraukset, tuntuu sovellus nopeammalta. Tällaista käyttöliittymää kutsutaan optimistiseksi käyttöliittymäksi (*optimistic user interface*).

JavaScriptilla voidaan päivittää käyttöliittymää tekemättä sivulatauksia. Palvelin tarjoaa asiakasohjelmalle API:n, jonka avulla sovelluksen tilaa voidaan manipuloida ja tarkastaa. Rajapinnasta saadut vastaukset ovat HTML-muodon sijasta XML- tai JSON-muotoisia (*javascript object notation, JSON*). JavaScriptia, jolla tehdään palvelimeen rajapintaan perustuvia muutoksia sovelluksessa, kutsutaan asynkrooniseksi JavaScriptiksi (*asynchronous javascript + XML, AJAX*). Asynkrooninen JavaScript perustuu sovelluksen logiikassa tehtyihin hypertekstin siirtoprotokolan (*hypertext transfer protocol, HTTP*) mukaisiin pyyntöihin. Palvelimelle tehtyjä pyyntöjä pystytään seuraamaan JavaScriptin tapahtumasilmukan avulla. Kun palvelimen vastaukset saapuvat selaimeen, lisätään niiden vastaukset HTTP-pyyntöä mallintavaan olioon. Tapahtumasilmukassa olevaa tarkistuslogiikkaa ajetaan niin kauan, että vastaus löytyy oliosta, jonka jälkeen vastausta voidaan käyttää web-sovelluksessa.

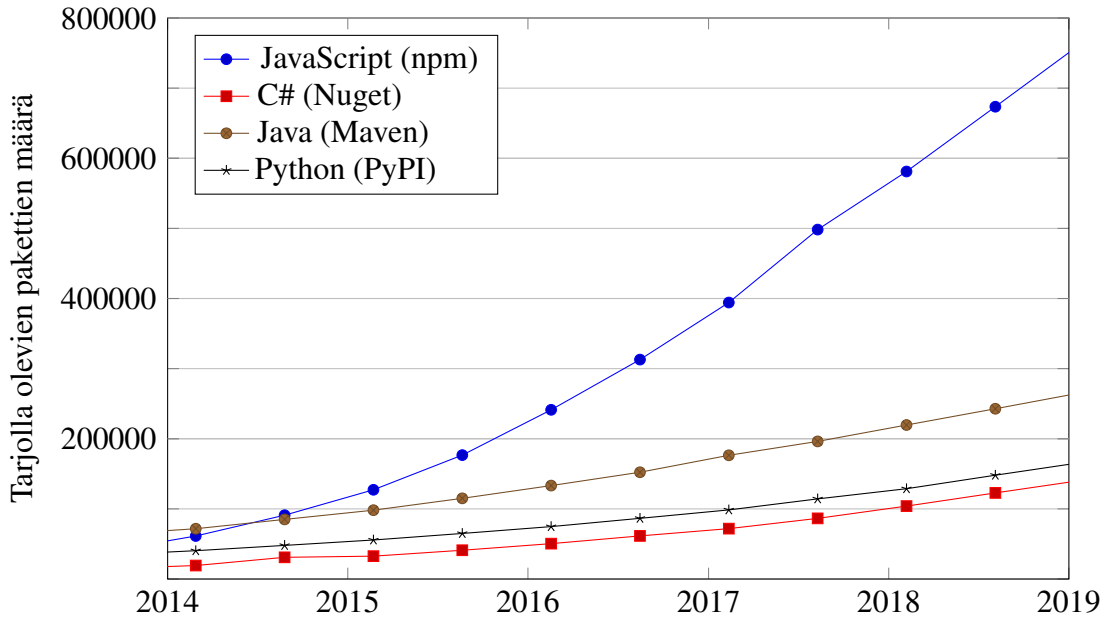
## 4.4 Yhden sivun sovellukset

Yhden sivun sovellus (*single page application, SPA*) on toteutustapa asiakasohjelmalle, jossa kaikki käyttöliittymän esitysmuodot ja muutokset toteutetaan alustavan sivulatauksesta saadun sivun sisälle. Asiakasohjelman ja palvelimen välinen kommunikointi tapahtuu asynkroonisella JavaScriptilla. Sovelluksen käyttäjäkokemus paranee, kun sivulatauksista johtuvia viiveitä ei ole. Palvelimen rajapinnoista saadut vastaukset sisältävät saman viiveen, minkä sivulataus sisältäisi, mutta asiakasohjelma pitää hallinnan sovellukseen ilman, että verkkoselain keskeyttää sitä. Sivulatausten sijasta asiakasohjelmassa voidaan esittää latausruutuja tai hyödyntää muita tekniikoita, joilla web-sovelluksen käyttö ei keskeydy.

Asiakasohjelman käyttöliittymä muodostuu alustavassa sivulatauksessa ladatuista HTML:stä, CSS:stä ja JavaScriptistä. Selain muodostaa näistä dokumenttioliomallin (*document object model, DOM*) ja tyylioliomallin (*cascading style sheet object model, CSSOM*). Yhdessä ne muodostavat piirtopuun (*render tree*) [12].

DOM:in ja CSSOM:in manipulointi voi pakottaa selaimen rakentamaan piirtopuun uudelleen, mikä voi ilmetä tökkivästi toimivana käyttöliittymänä. Käyttöliittymien toteuttamiseen on luotu useita JavaScript-kirjastoja, jotka optimoivat suorituskykyä tekemällä DOM-manipuloinnin älykkäästi.

Ilmaisukykyisempien ECMAScript standardien jälkeen JavaScriptin rooli kasvoi merkittävästi web-sovellusten toteuttamisessa (ks. kuva 4.1). Kääntäjiin perustuvan kehitystyön yleistyminen auttoi uusien täsmäkielten kehittämisessä (*domain-specific language*). Täsmäkielillä voidaan ilmaista web-sovelluksen käyttöliittymä HTML:n sijasta JavaScriptissa kirjoitetulla logiikalla.




Kuva 4.1: Javascriptin kasvu vuosina 2014-2019 [13]

## 4.5 Käyttöliittymän toteuttaminen React-kirjastolla

React on kirjasto, joka on suunniteltu web-sovelluksille käyttöliittymien toteuttamiseen [14]. Sen on kehittänyt Facebook. React esittelee web-kehitykseen uuden sanaston, joka kuvaa niitä abstraktioita, joita Reactissa hyödynnetään muodostamaan web-sovellus. Keskeisin käsite on komponentti (*component*), joka muodostaa pienimmän yksikön käyttöliittymän muodostuksessa. Komponenttien toteutuksessa ilmaistaan, mistä muista komponenteista tai HTML-elementeistä se koostuu, millainen sen tila on ja millaisia kuuntelijoita siihen on kiinnitetty. Komponentteja voidaan ajatella samalla tavalla kuin HTML:ssä käytettyjä elementtejä, mutta niitä ei voida käyttää ilman Reactia.

Reactissa web-sovellus toteutetaan JavaScript-tiedostoihin. HTML:llä tehtyjä tiedostoja ei käytetä ja tyyliohjeet voidaan myös määrittää JavaScript-tiedostoissa. Reactia varten on kehitetty JavaScriptista eteenpäin jalostettu versio, joka tunnetaan nimellä *JavaScript XML* (*JSX*). JSX toimii täsmäkielenä, mikä korvaa HTML:n Reactilla kehitetyissä web-sovelluksissa. JSX-kieli on Reactin suosiman Babel-kääntäjän tukema muoto, josta tuotetaan web-sovellus verkkoselaimen ymmärtämässä muodossa.

### 4.5.1 Virtuaalinen DOM

React edustaa lähestymistä DOM:in hallintaan, jossa käytettyjen abstraktioiden, kuten komponenttien kautta voidaan deklaratiivisesti ilmaista, millainen käyttöliittymä halutaan. DOM:in alustava muodostus ja ylläpito perustuu virtuaaliseen DOM:iin. Virtuaalinen DOM on DOM:ia kuvaava erillinen tietorakenne, johon kerätään DOM:iin halutut muutokset. Erillisen tietorakenteen avulla voidaan pienillä suorituskykyvaatimuksilla kerätä halutut muutokset. Virtuaalista DOM:ia ja oikeaa DOM:ia voidaan vertailla keskenään, tunnistaa niiden erot ja suorittaa minimoitu määrä muutoksia oikeaan DOM:iin, jotta siitä saadaan samanlainen kuin virtuaalisesta DOM:ista.

Virtuaalisen DOM:in avulla web-sovellusten kehittäminen Reactilla on deklaratiivista, jossa käyttöliittymää voidaan esittää datana ja varsinaisten muutosten takana on virtuaalisen ja aidon DOM:in sovittamiseen (*reconciliation*) suunniteltu algoritmi.

## 4.6 Tilanhallinta Redux-kirjastolla

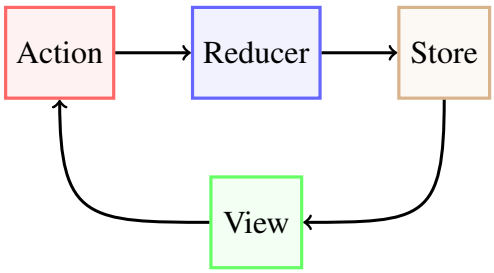
Redux on web-sovelluksille luotu kirjasto tilanhallintaan. Reduxista on virallinen kirjasto Reactille. Reduxilla keskitetään sovelluksen tila, joka mahdollistaa tehokkaita toimintoja kuten aikaisempaan tilaan peruuttamisen ja peruutuksen jälkeen tulevien tilojen toistamisen. Redux antaa työkalut sovelluksen tilan muutosten ymmärtämiseen. Reduxiin perustuva arkkitehtuuri web-sovelluksessa mahdollistaa käänteisen debuggaamisen [15].

Redux ja React toimivat hyvin yhdessä. React ilmaisee käyttöliittymien esitysmuodon deklaratiivisesti käyttäen tietorakenteita, jotka voidaan myös keskittää yhdeksi tietorakenteeksi. Keskitetyllä tietorakenteella vältetään datan tarpeeton duplikointi.

Reduxin kolme pääperiaatetta ovat:

- Koko sovelluksen tila on puurakenteessa yhden *varaston* sisällä.
- Ainut tapa muuttaa tilaa on lähettää *toiminto*, jossa kuvataan, mitä on tapahtunut.
- Tilan muuttamiseen käytetään sivuvaikuttuksetomia *vähentäjä*-funktioita.

Reduxissa eri osista puhutaan niiden omilla nimillä (ks. kuva 4.2). Varasto (*store*) on tietorakenne, jolla mallinnetaan sovelluksen tilaa. Pääperiatteen mukaisesti niitä on vain yksi, josta puhutaan myös yksittäisenä totuuden lähteenä. Kun sovelluksen tila määritetään pelkästään Reduxin avulla, kaikki sovelluksen osat tilaavat tarvitsemansa datan sieltä. Datan tilaaminen tapahtuu käyttäen seuraaja-mallia (*Observer pattern*). Varasto pitää kirjaa siitä, mikä osa sen datasta on tilattuna millekkin sovelluksen osalle. Kun data muuttuu, varasto lähettää muutoksen sen tilanneille.




Kuva 4.2: Reduxin eri osat

Toiminnot (*action*) ovat yksinkertaisia tietorakenteita, joita vähentäjä-funktiot (*reducer*) voivat käyttää (ks. listaus 4.1). Toimintoja tuotetaan näkymässä (*view*) esimerkiksi käyttäjän syötteistä. Toiminnot sisältävät vähintään tiedon niiden tyypistä, mutta usein niissä on myös varsinaista, tulevassa tilassa käytettävää dataa. Varaston alustuksessa vähentäjille määritetään vastuualueet samalla kun varasto alustetaan lähtötilaan. Kun toiminto lähetetään, kaikki vähentäjät vertaavat toiminnon tyyppiä nähdäkseen, onko se vähentäjän vastuualueeseen kuuluva.

```
1  // initial state is: { counter: 0 }
2  // example action: { type: "ADD", payload: 1 }
3
4  const reducer = (state, action) => {
5      switch(action.type) {
6          case "ADD": {
7              return state.counter + action.payload;
8          }
9
10         default: {
11             return state;
12         }
13     }
14 }
```

Listaus 4.1: Yksinkertainen vähentäjä (JavaScript)

### 4.6.1 Käänteinen debuggaus Reduxilla

Reduxin yhteydessä puhuttu aikamatkustus-debuggaaminen voidaan käsittää käänteisenä debuggaamisena, kun koko sovelluksen tila mallinnetaan sillä. Käytännössä tämä tarkoittaa sitä, että sovelluksen kaikki osat riippuvat yksittäisestä tietorakenteesta. Vähentäjä-funktioilla kuvataan tilasiirtymät. Käänteisen debuggaamisen toteuttaminen edellyttää, että kaikki tilamuutokset on toteutettu Reduxissa.

### 4.6.2 Jalostettuun syötteeseen perustuva käänteinen debuggaus

Kaikkia verkkoselaimessa tapahtuvia ilmiöitä ei ole mielekästä ilmaista Reduxin toimintoina. Ilmiön seuraaminen tarkoittaa sen mallintamista sovelluksen tilan tietorakenteessa, uuden vähentäjä-funktion tuottamista ja ilmiötä mallintavien toimintojen luomista. Jos ilmiötä ei tarvitse mallintaa Reduxissa, voidaan sen ohittamista perustella säästyvässä työmäärässä. Reduxiin pohjautuvien web-sovellusten kehityksessä tehdään oletuksia siitä, että verkkoselaimen yleisimmät toiminnot, kuten esimerkiksi kirjoituskenttään kirjoitta-

minen ja nappien eri tilat, toimivat luotettavasti ja niitä ei erikseen kannata tuoda varastoon. Redux ratkaisee ensisijaisesti tilanhallinnan haasteita, käänteisen debuggaamisen toteutuminen on toissijaista. Lomakeen täyttäminen on näkymätöntä Reduxille, jos niistä ei tuoteta toimintoja. Lomakkeen tallennus voidaan yhdistää varastoon tuottamalla siitä oma toiminto.

Lähetetty lomake ei välttämättä ilmene Reduxissa mitenkään. Vaikka lomakkeen vastaanottanut palvelin vastauksellaan muuttaisi sovelluksen tilaa, sen yhteys lähetettyyn lomakkeeseen ei ole välttämättä suoraviivaista.

Käänteisen debuggauksen toteuttaminen Reduxilla perustuu toimintojen ja niistä seuraavien tilojen tallentamiseen. Toimintoja voidaan ajatella jalostettuna syötteenä, joka pohjautuu selaimessa tehtyihin syötteisiin. Raakasyötteeksi lasketaan kirjoittaminen ja nappien painaminen. Tarkkaa aikakäsitettä ei erikseen muodostu, koska toimintojen välissä voi kulua vaihtelevasti aikaa. Toiminnot ja niistä seuraavat tilat kuitenkin tallennetaan järjestyksessä ja se mahdollistaa käänteisen debuggauksen. Reduxilla toteutettu käänteinen debuggaus on jäljitystiedostoon perustuva. Sovelluksen tilat tallentuvat tarkasti, mutta tilasiirtymien yksityiskohdat voivat olla liian kustannustehottomia mallintaa pelkästään käänteistä debuggausta varten.

# Luku 5

# Arkkitehtuuri videopeleissä

Videopelit ovat sovelluksia, joissa pelaaja aktiivisesti vaikuttaa pelin etenemiseen ohjaamalla sitä syötelaitteella. Pelaajat kokevat tekemiensä syötteiden seuraukset välittömänä palautteena, joka voi olla mm. kuvaa ja ääntä. Pelin suorituksen kesto voi olla kerrallaan useita tunteja ja suorituksen tallentaminen ja lataaminen on yleistä. Peräkkäisten pelikertojen yhteenlaskettu pituus voi olla jopa satoja tunteja.

Pelin suorituksella ei ole varsinaista loppua, mutta pelin asiayhteydessä loppu voidaan tunnistaa. Usein pelit suunnitellaan ydinsilmukka (*core loop*) -käsitteen ympärille. Siinä pelaajalle esitetään tavoitteita ja haasteita, jotka motivoivat häntä etenemään. Ydinsilmukka käsittelee peliä pelimekaniikkojen asiayhteydessä. Pelin tarinan tai selkeästi suunnitellun pelikerran loppu ei välttämättä tarkoita, että pelaaminen on ohi. Monissa peleissä on paljon uudelleenpeluuarvoa.

Videopeleissä ohjelmalogiikka on usein toteutettu silmukkaan, jota kutsutaan pelisilmukaksi (*game loop*) ja siinä kerätään pelaajan syötteet (*input*), päivitetään pelin tila (*update*) ja piirretään pelin tila ruudulle (*draw*). Jos pelisilmukka on toteutettu huonosti, se vaikuttaa pelin suoritettavuuteen ja pahimmillaan häiritsee pelimekaniikkoja ja ydinsilmukkaa.

## 5.1 Päivitys- ja piirtorutiinit

Ohjelmistoarkkitehtuurin näkökulmasta videopelit ovat sovelluksia, joissa logiikka hyödyntää oheislaitteita, jotka tuottavat kuvaa ja ääntä esittääkseen videopelin sisäisen tilan. Videopelien logiikka voidaan jakaa tilaa manipuloivaan ja tilaa esittäviin osuuksiin.

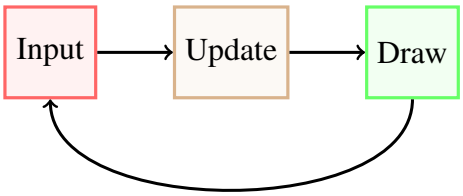
Pelin tilaa manipuloivaa osuutta voidaan kutsua päivitysrutiiniksi ja tilaa esittävää osuutta voidaan kutsua piirtorutiiniksi. Päivitysrutiini muuttaa pelin tilaa kuluneen ajan ja saatujen syötteiden pohjalta.

Usein peleissä mallinnetaan tilaa, joka muuttuu reaaliajassa. Päivitysrutiinin suoritus riippuu ajan kulumisesta ja se voidaan suorittaa jopa satoja kertoja sekunnissa. Vaikka videopelin tila on diskreetti, voidaan se päivittää niin usein, että pelin ruudunpäivitykset antavat uskottavan esityksen liikkuvista pelikappaleista. Videopelin ei tarvitse olla aikaan sidottu simulaatio pelimaailmasta, vaan sen suoritus voi olla myös ei-reaaliaikaista ja perustua pelkästään pelaajan tekemiin syötteisiin.

## 5.2 Pelisilmukka

Ennen päivitysrutiinin suorittamista luetaan pelaajan tekemät syötteet (ks. kuva 5.1). Syötteiden lukemisen jälkeen pelin tila päivitetään niiden mukaan ja muuttunut pelin tila esitetään pelaajalle piirtämällä se ruudulle. Logiikkaa suoritetaan silmukassa, jota kutsutaan pelisilmukaksi. Ajan kulkuun perustuvissa peleissä pelin päivitysrutiiniin syötetään viime päivityksestä mennyt aika parametrina päivitysrutiinille. Aika ja pelaajan tekemät syötteet voidaan ilmaista sivuvaikutuksina sen sijaan, että ne annettaisiin funktion argumentteina.

Ihanteellisessa päivitysrutiinissa menee aina vähemmän aikaa kuin ruudunpäivitykseen on varattu aikaa. Suorituksen säännöllisyyttä ei voida taata, koska nykyajan laitteistot suorittavat useita sovelluksia rinnakkain käyttäen käyttöjärjestelmän moniajoa (*multitasking*). Koska videopelin suoritusnopeudelle ei ole varmuutta, pitää päivitysrutiinia ajaa vaihtelevalla aikaparametrilla. Aikaparametria kutsutaan delta-ajaksi (*delta time*). Jos pe-




Kuva 5.1: Yksinkertaisen pelisilmukan eri osat

lin suoritusnopeutta ei ole rajoitettu, niin pelin sisäistä tilaa voidaan päivittää useammin kuin ruudunpäivityksiä tehdään.

Reaaliaikaisissa peleissä pelimaailmaa päivitetään riittävän usein, jotta sen simulointi olisi tarpeeksi tarkkaa ja jotta pelikokemus olisi sulava. Usein yhtä pelisilmukan iteraatiota seuraa ruudunpäivitys. Kannettavilla laitteilla pelaamista voidaan optimoida sähkönkulutuksen näkökulmasta, jolloin ruudunpäivitysten määrälle asetetaan yläraja, vaikka niitä pystyttäisiin suorittamaan huomattavasti enemmän.

Jos peli ei perustu reaaliaikaisuuteen, kannattaa ruudunpäivityksiä suorittaa ainoastaan silloin, kun pelin esitys muuttuu. Pelaajan syötteistä riippuvainen pelilogiikka on helpompi toteuttaa delta-aikaan perustuvan pelisilmukan sijasta tapahtumasilmukalla (*event loop*). Tapahtumasilmukka on rinnakkain suoritettujen ja asynkroonisten tapahtumien seuraamiseen soveltuva toteutusmalli. Sitä käytetään JavaScriptissa mahdollistamaan rinnakkaisesti ajettu logiikka.

### 5.2.1 Iteraatiopohjainen pelisilmukka

Päivitysrutiini, joka on herkkä muuttuvalle delta-ajalle, voidaan suorittaa myös vakioksi määritetyllä delta-ajalla. Herkkiin järjestelmiin lukeutuu fysiikkamoottorit, joita käytetään esimerkiksi kiinteiden kappaleiden törmäysten simuloimiseen (*rigid physics simulation*).

Vakioksi määritetyn delta-ajan käyttäminen edellyttää, että päivitysrutiinin suoritustiheyttä hallitaan erillisellä aikaa kerryttävällä muuttujalla (*time accumulator*). Tällaista

```
1  local accumulator = 0
2  local draw_frame = true
3  local frametime = 1/60
4
5  while true do
6    accumulator = accumulator + elapsed_time()
7
8    while accumulator >= frametime do
9      update( frametime )
10
11     accumulator = accumulator - frametime
12     draw_frame = true
13
14     accumulator = accumulator + elapsed_time()
15   end
16
17   if draw_frame == true then
18     draw()
19     draw_frame = false
20   end
21 end
```

Listaus 5.1: Iteraatiopohjainen pelisilmukka (Lua)

pelisilmukkaa kutsutaan iteraatiopohjaiseksi, koska se perustuu toistettujen iteraatioiden määrään, vaikka sen logiikassa käytettäisiin delta-aikaa ajan ilmaisemiseen. Aikaa kerätään aktiivisella odottamisella (*busy waiting*) (ks. listaus 5.1).

Aktiivisella odottamisella tarkoitetaan silmukkaa, jossa varsinainen ohjelmalogiikka on määritetty suoritettavaksi jonkun sivuvaikutuksiin perustuvan ehdon toteutuessa, jota silmukassa tarkastellaan. Aktiivisen odottamisen laatu perustuu silmukan tiheään toistotaajuuteen. Koska aktiivinen odottaminen ei tee mitään ennen kuin ehto on tosi, se tuhlaa suoritinaikaa. Käyttöjärjestelmä voi sammuttaa sovelluksen, jos näyttää siltä, että se on jumittunut ikuiseen silmukkaan. Aktiivisessa odottamisessa sovelluksen suorittaminen voidaan väliaikaisesti keskeyttää käyttämällä alustakohtaista *sleep*-funktiota. Jos *sleep*-

funktion käyttö tuottaa liian pitkän viiveen, voidaan aktiivisella odottamisella myös pitää käyttöjärjestelmän antama kontrolli.

### 5.2.2 Kuoleman spiraali iteraatiopohjaisessa pelisilmukassa

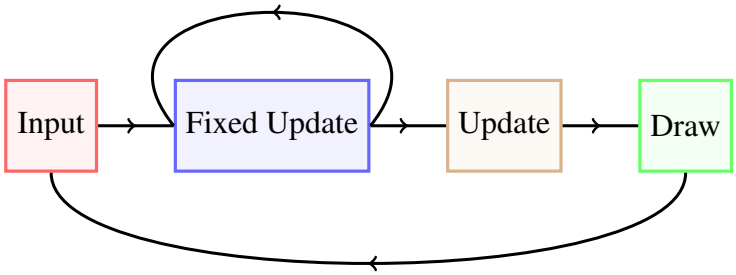
Kun aikaa on kerrytetty enemmän kuin ruudunpäivitykseen tarvitaan, kulutetaan sitä ruudunpäivitykseen tarvitulla määrällä. Jos simulaatio jää jälkeen useita ruudunpäivityksiä, voidaan kerrytettyä aikaa kuluttaa useiden ruudunpäivitysten edestä kerrallaan. Jos pelin tilan päivittämiseen menee kauemmin kuin mitä siinä kulutetaan aikaa, ajautuu peli kuoleman spiraaliin (*spiral of death*). Kuoleman spiraalissa peliä pyritään päivittämään mahdollisimman lähelle nykyhetkeä, mutta kulutettavan ajan määrä kasvaa jokaisessa iteraatiossa. [16]

Kuoleman spiraalin voi välttää pitämällä kirjaa peräkkäin suoritetuista päivitysrutiinin suorituksista ja pakottaa päivittämisen keskeytys. Kun aikaa ei pystytä reaaliajassa kuluttamaan eikä peli myöskään jää ikuiseen silmukkaan, peli toimii tavallista hitaammin. Kun peli on laitteistolle liian vaativa — riippumatta siitä käytetäänkö iteraatiopohjaista tai delta-aikaan pohjautuvaa silmukkaa — pelin ruudunpäivitysnopeus on huono. Sekä muuttuvaan delta-aikaan että iteraatioihin perustuvia silmukoita voidaan käyttää pelisilmukan sisällä samaan aikaan, jolloin pelin päivittämistä voidaan hallita paremmin.

### 5.2.3 Iteraatiohin ja delta-aikaan perustuva päivittäminen

Vaikka iteraatiopohjaista silmukkaa käytetään suorittamaan logiikkaa, joka vaatii vakioksi määritetyn delta-ajan, voidaan kaikki päivitysrutiinin logiikka kirjoittaa sen sisälle. Vakioksi määritetyn delta-ajan käyttäminen aikaan perustuvan logiikan suorittamisessa antaa pelille vakautta. Jos iteraatiopohjaisen silmukan sisällä suoritettava logiikka on liian hidasta, voidaan pelin päivitykseen tarvittava logiikka jakaa sen ja muuttuvaan delta-aikaan perustuvan päivitysrutiinin välille (ks. kuva 5.2). Yksi mahdollisuus on jättää vakioksi asetetusta delta-ajasta riippuva logiikka iteraatiopohjaiseen päivitysrutiiniin (*fixed*

*update*) ja kaikki muu logiikka muuttuvaa delta-aikaa käyttävään päivitysrutiiniin (*update*).




Kuva 5.2: Pelisilmukan eri osat

Iteraatioiden toistotaajuutta voidaan säätää niin, että vakioksi asetettua delta-aikaa käyttävää päivitysrutiinia suoritetaan tavoiteltua kuvanpäivitysnopeutta harvemmin ja muuttuvaan delta-aikaan perustuvaa päivitysrutiinia suoritetaan yhtä usein kuin ruudunpäivityksiä tehdään. Jos pelissä olevien kappaleiden sijainnit muuttuvat vain vakioksi asetetussa delta-aikaan perustuvassa päivitysrutiinissa, voidaan delta-aikaan pohjautuvaa päivitysrutiinia käyttää pelin edeltävän ja nykyisen tilan välin interpoloinnissa. Interpolointikohdan määrittämisessä käytetään kerryttäjään jäännyttä ajan määrää ja ruudunpäivitykseen tarvitun ajan suhdetta. Interpoloimalla pelin kahden tilan väliä voidaan pelin muutoksia esittää useammilla ruudunpäivityksillä kuin pelin tilan oikeasti pystyisi esittämään. Koska pelin esitys perustuu aikaisemman ja nykyisen tilan väliltä interpoloituun esitykseen, syötteiden viiveet koetaan pidempinä. [16]

# Luku 6

# Videopelien käänteinen debuggaus

## 6.1 Syötetiedoston ja jäljitystiedoston hyödyt ja haasteet

Käänteisen debuggaamisen toteuttaminen videopeliin voidaan perustaa syötetiedostoon tai jäljitystiedostoon. Syötetiedostossa pitää ratkaista lähtötilan ja pelaajan tekemien syötteiden tarkka tallennus sekä niiden ajoitus pelissä. Syötetiedosto voidaan valita, jos peli toimii deterministisesti. Epädeterministiset arvot, jotka vaikuttavat pelin suoritukseen, pitää tallentaa syötetiedostoon.

Jäljitystiedosto edustaa lähestymistä, jossa tiedot pelin tilasta tallennetaan ja todettuja bugeja tarkastellaan pelistä erillään. Bugi voi ilmetä monilla eri tavoilla, joista sovelluksen kaatuminen on ilmeisin. Bugit, jotka ovat ainoastaan pelaajan havaittavissa, hyötyvät pelin osallisuudesta käänteisen debuggauksen prosessissa. Pelaajalle selkeitä bugeja ovat kuvaan ja ääneen liittyvät bugit ja esimerkiksi pelikappaleiden väärä käyttäytyminen. Koska audiovisuaaliset bugit voivat olla hetkillisiä eli temporaalisia, niiden huomaaminen on helpompaa nauhoitetusta videosta. Jos jäljitystiedostoon sisällytetään pelin kuvan ja äänen tallennus, niin jäljitystiedoston koko kasvaa merkittävästi.

Tallennustilavaatimuksiltaan syötetiedosto ja jäljitystiedosto edustavat vastakkaisia ääripäitä. Syötetiedostolla mahdollisimman paljon peliin liittyvästä datasta voidaan joh-

taa suorittamalla peli käyttäen nauhoitettuja syötteitä. Jäljitystiedosto pyrkii tallentamaan kaiken niin että sovellusta ei tarvitsisi suorittaa uudelleen.

## 6.2 Desynkronisaatio debuggaamisessa

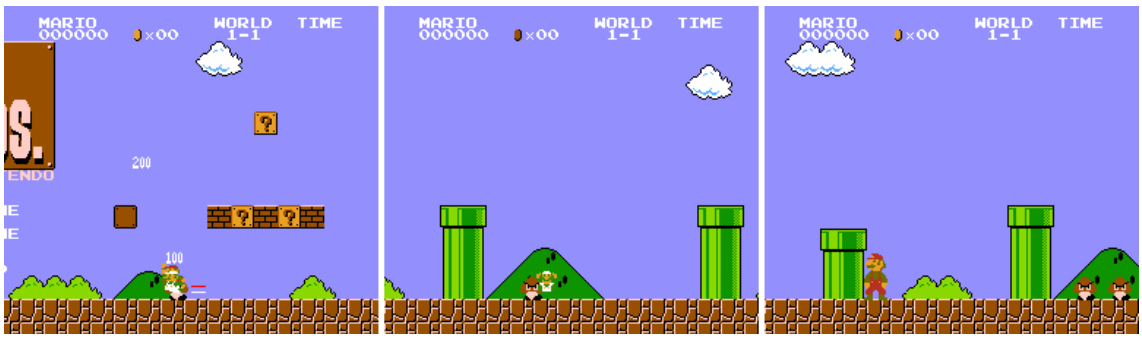
Desynkronisaatio tarkoittaa tilannetta, jossa käänteinen debuggaus tuottaa eri lopputuloksen kuin tallennettu peluu todellisuudessa oli. Desynkronisaatio on olemassaoleva termi, jota käytetään kuvailemaan synkronoituun simulaatioon perustuvassa moninpelissä tapahtuvaa virhetilaa. Synkronoitu simulaatioon on vaatimuksiltaan sama kuin käänteinen debuggaus. Osallistujien pitää kommunikoida toisilleen peliin vaikuttava epädeterminismi, jotta kaikki osallistujat voivat suorittaa peliä eteenpäin samalla tavalla [17]. Epädeterminismin vähentämiseen käytettyjä tekniikoita voidaan hyödyntää sekä moninpelin että käänteisen debuggaamisen toteuttamisessa.

Desynkronisaation havainnollistamiseen voidaan käyttää Super Mario Bros -videopelin houkutustilassa olevaa bugia. Super Mario Bros (Nintendo, 1985) on yksi suosituimmista peleistä pikapelaamisessa. Huomattava määrä pelin logiikasta on dokumentoitu sekä tekstinä että emulaattorilla ajettavien syötetiedostojen muodossa [18][19].

Houkutustila käynnistyy pelin aloitusvalikosta lyhyen odottamisen jälkeen. Houkutustilassa pelin päähenkilö Mario etenee kenttää eteenpäin, hyppää vihollisen päälle ja kerää sienen. Sen jälkeen Mario etenee kentässä, kohtaa uusia vihollisia ja kuolee. Nintendo Entertainment System -pelikonsolilla houkutustilan peluusta on kaksi versiota. Molemmissa versioissa näppäinsyötteet vaikuttavat olevan samat, joka viittaa desynkronisaatioon.

Kuin kaksoisheilurissa, pieni muutos peluussa voi aiheuttaa merkittävästi poikkeavan lopputuloksen. Houkutustilan kaksi versiota ovat identtiset ensimmäisen vihollisen tallomiseen asti. Vihollisen jälkeen peluut poikkeavat toisistaan. Poikkeavassa peluussa Mario onnistuu sienen keräämisessä, mutta on hieman eri kohdassa, josta johtuen seuraavat hy-

pyt epäonnistuvat ja Mario ei kohtaakaan vihollisia. Mario pitää henkensä, houkutustila pysähtyy ja palataan valikkoon (ks. kuva 6.1).




Kuva 6.1: Marion desynkronisaation lopputilat

Tarkemman tarkastelun tuloksena huomataan, että Mario talloo vihollisen eri korkeuksilla. Korkeusero aiheuttaa sen, että laskeutumisaika on versioiden välillä eri. Muuttunut laskeutumisaika vaikuttaa tulevaan juoksuun. Juoksusta seuraava hyppy koskettaa esteitä eri tavalla, minkä jälkeen desynkronisaatio on ilmeisen selvä.

## 6.3 Tilan sarjallistaminen

Pelit, jotka on suunniteltu pelattavaksi osissa tarvitsevat tallennustoiminnon (*save*), jotta peliä voidaan jatkaa pelin uudelleenkäynnistyksen jälkeen. Tilan sarjallistaminen (*serialization*) on tarpeellista, jos pelissä halutaan toteuttaa tallennustoiminto. Kun tila halutaan palauttaa lataustoiminnolla (*load*), puretaan sarjallistettu (*deserialization*) data ja alustetaan pelimaailma uudelleen. Sarjallistamisen rajoittaminen vain osaan pelimaailmaa voi olla riittävä, kun peliin tehdään tavallinen tallennustoiminto. Tallennetun tilan lataamisen jälkeen peli voi generoida pelikappaleita, jotka tallennuksessa sivuutettiin.

Tallennustoiminto voi olla myös pelimekaniikka, joka huomioi todellisessa maailmassa kuluneen ajan. Pelimekaniikka voi olla pelkästään visuaalinen, esimerkiksi pelin vuorokauden asettaminen todellisen kellonajan perusteella tai pelimaailmaan vaikuttava, esimerkiksi jäähtymismekaniikkojen (*cooldown*) muodossa.

Sarjallistamista tarvitaan käänteisessä debuggaamisessa mahdollistamaan tehokas palaaminen pelin aikaisempaan tilaan. Käänteinen debuggaus perustuu siihen, että menneisyyttä pystytään tarkastelemaan. Videopelien suorituksesta syntyvät tilat eivät ole muodostuneet yksiselitteisesti, joten aikaisemman tilan päätteleminen nykyisestä tilasta ei ole mahdollista. Aikaisempaan tilaan päästään, kun peliä suoritetaan alusta eteenpäin kunnes päästään haluttuun ajanhetkeen.

Sarjallistamisella voidaan tuottaa uusia lähtöjä käänteiselle debuggaamiselle. Lataamalla tarkasteluhetkeä lähimpänä oleva tila voidaan käänteistä debuggaamista tehdä interaktiivisemmin. Tilan palautuksen jälkeen pelin tilan pitää olla täysin sama kuin se oli sarjallistamisen aikaan. Jos käänteinen debuggaaminen perustuu syötetiedostoon, pelin suorittaminen lähtötilasta tallennettuun tilaan ja tilan lataamisen välillä ei pitäisi olla eroja. Sarjallistettuja tiloja voidaan tuottaa lisää, jos käänteinen debuggaaminen perustuu syötetiedostoon. Hyvä strategia sarjallistamisessa säästää levytilaa.

## 6.4 Epädeterminismin hallinta

### 6.4.1 Pelaajan syötteet

Ilmeisin pelinkehittäjän kontrollin ulkopuolella oleva asia on pelaaja. Videopelit ovat määritelmänsä mukaan interaktiivisia sovelluksia ja niitä pelaava päättää itse pelitapansa. Peliohjaimesta riippuen tallennettavan datan määrä voi olla suuri.

| Kontrolli | Bittimäärä | Tyypillinen lukumäärä |
|---|---|---|
| Nappi | 1 | 10 |
| Suuntanapit | 4 | 1 |
| Analoginen tikku | 20 | 2 |
| Liipasin | 10 | 2 |

Taulukko 6.1: Yleisimpien kontrollien tuottamat bittimäärät

Kontrolleista syntyvän bittivirtauksen nopeus perustuu videopelikohtaisesti ohjelmoituun kontrollien lukutaajuuteen ja niiden määrään (ks. taulukko 6.1). Useimmat televisiot ja tietokonemonitorit pystyvät päivittämään kuvansa 60 kertaa sekunnissa ja siksi videopeleiltäkin odotetaan vastaavaa nopeutta.

$$\frac{72\ bittiä}{kuvapäivitys} * \frac{60\ kuvapäivitystä}{sekunnissa} = 4,44kb/s$$

Tyypillisten kontrollien tallentamiseen menee vain noin neljä kilotavua sekunnissa (ks. yhtälö refequation:kontrollit). Harvinaisempia kontrolleja voivat olla kiihtyvyysanturit, gyroskooppi ja kompassi, mutta niiden lisäämät bittimäärät eivät ole suuruusluokaltaan isompia kuin tyypilliset kontrollit. Nykyisillä tallennnusmedioilla kontrollien tallentaminen pitäisi olla helppoa. Syötedatan tallentaminen on suurudeltaan paljon pienempää kuin videoiden tallentaminen. YouTuben suositusten mukaan lähetetyn monoäänen pitäisi olla *128 kb/s* ja 720p videon pitäisi olla *6,5 Mb/s* bittivirtauksiltaan [20]. YouTube:ssa olevien videoiden tapaan syötedataa voidaan myös pakata pienemmäksi.

Jos pelaaminen perustuu kameran kuvaan, voi syötteiden tallentaminen asettaa liian suuria vaatimuksia pelien nauhoittamiselle. Suurta tallennuskapasiteettia ja pitkiä pelisessioita vaativissa peleissä syötteiden tallentamisen sijaan voidaan pyrkiä tallentamaan syötteet niiden jalostetussa muodossa (luku 4.6.2). Syötteiden jalostettu muoto riippuu videopelin tyypistä ja siinä olevasta logiikasta. Jos peli tuottaa suuresta syötedatan määrästä pienen määrän pelissä käytettävää ohjausdataa, voidaan syötteiden tallennus toteuttaa syötteitä käsittelevän logiikan perään. Kun pelin toistossa käytetään suoraan jalostettua syötedataa, tarkoittaa se sitä, että syötedataa jalostavaa logiikkaa ei voida debugata käyttäen käänteistä debuggausta.

### 6.4.2 Ajan ilmaiseminen

Videopeleissä aika ilmaistaan päivitysrutiinille antamalla aikaisemmasta suorituksesta kulunut aika parametrinä. Aika-parametri eli delta-aika kertoo pelin logiikalle kuinka pitkälle pelin sisäistä tilaa pitää päivittää. Koska käyttöjärjestelmä määrittää milloin ja miten paljon peli saa suoritinaikaa, delta-ajan muutoksia ei voida ennustaa. Delta-aika on osa epädeterminismiä, joka pitää tallentaa, jotta käänteinen debuggaus voidaan toteuttaa. Usein peleissä aikaparametri annetaan tuplatarkkuuden liukulukuna, (*double*) joka tarkoittaa sitä, että delta-ajan koko on *64 bittiä*. Sen koko on samaa suuruusluokkaa kuin tyypillisistä kontrolleista syntyvä *74 bittiä* ruudunpäivitystä kohti.

Tavallisen pelisilmukan delta-ajan tallentamisesta seuraa se, että peluun toistossa on samat suoritinajan puutteesta seuranneet tauot, joita alkuperäisessä peluussa oli. Jos alkuperäisen peluun aikana saatuun suoritinaikaan tuli ruudunpäivitystä pidempi tauko, koetaan se myös käänteisessä debuggauksessa toistetussa peluussa. Pelissä käytetyn vaihtelevan delta-ajan pystyy muuttamaan vakioksi (ks. luku 5.2.1). Jos delta-aika on asetettu vakioksi, peluun aikana tapahtuneet tauot eivät näy käänteisessä debuggaamisessa.

### 6.4.3 Pelien suoritusjärjestys ja liukuluvut

Teknologian kehityksen myötä laitteiden kyky suorittaa videopelejä on tullut paremmaksi. Koska videopelit on kaupallinen viihteen ala, pelinkehittäjät kilpailevat keskenään asiakkaista ja siksi pelit ovat kehittyneet pelimaailmojen laajuudessa ja grafiikan laadussa. Parempien pelien taustalla on se, että peliteollisuus on tullut kypsemmäksi, mutta myös se, että pelien suorituskykyyn liittyvät haasteet ovat helpottuneet ajan kuluessa.

Suorittimien suoritusteho perustuu yhä voimakkaammin monen ytimen samanaikaiseen käyttämiseen. Usean ytimen hyödyntäminen samaan aikaan edellyttää, että peli toteutetaan monisäikeiseksi. Pelin eri osien riippuvuuksien ymmärtäminen helpottaa pelin logiikan jakamisessa rinnakkain ajettaviksi kokonaisuuksiksi.

Vaikka pelaamisen näkökulmasta pelin suoritetuksessa sallittu säikeiden eriaikaisuus ei vaikuttaisi pelikokemukseen, voi se haitata pelin toistamista käänteisessä debuggaamisessa. Logiikan suoritusjärjestys voi vaikuttaa pelin lopputulokseen, esimerkiksi muuttaa pelien logiikassa tapahtuvia ajoituksia. Yksisäikeisesti toimiva peli tuottaa odotetuimman käyttäytymisen, mutta kun peliä ajetaan moniajon mahdollistavassa ympäristössä, käyttöjärjestelmä voi vaikuttaa siihen odottamattomasti.

Liukuluku on tietotyyppi, jolla pystytään esittämään reaalilukuja. Ne on määritetty vakiopituisiksi ja ovat pituudeltaan usein *32* tai *64 bittiä*. Niillä voidaan esittää kokonaislukujen tietotyyppiin verrattuna huomattavasti pienempi minimiarvo ja suurempi maksimiarvo. Esitystarkkuudessa on pakko kompromisoida, koska reaalilukuja mahtuu äärettömästi kahden luvun välille. Tästä johtuen liukuvut eivät voi esittää kaikkia lukuja minimi- ja maksimiarvojensa väliltä.

Liukuluvut rakentuvat kolmesta osasta: etumerkistä, eksponentista ja mantissasta (ks. taulukko 6.2). Etumerkkiä ilmaistaan yhdellä bitillä ja loput biteistä jaetaan eksponentille ja mantissalle. Mantissa ilmaisee liukuvun merkitseviä lukuja ja ne on esitetty kantaluvun potensseina. Merkitseviä lukuja edeltää aina näkymätön ykkönen. Eksponentti ilmaisee liukuvun ensimmäisen merkitsevän luvun suuruuden. Esitys luetaan kuin se olisi etumerkitön kokonaisluku ja siitä vähennetään puolet kokonaislukuesityksen maksimiarvosta, pyöristettynä alaspäin).

Koska esitystarkkuus on rajallinen, kahden yhteenlasketun liukuluvun summa ei välttämättä ole osa liukuluvun mahdollisia esitysmuotoja. Jos summaa ei voida esittää sellaisenaan, valitaan summan ympäristöstä arvo, joka voidaan esittää liukulukuna. Liukulukujen esityskyky on parhaimmillaan lähellä nollaa ja se heikkenee mitä kauemmas edetään nollasta. Koska liukulukujen lukujana ei ole jatkuva, niille ei päde liitännäisyyslaki. Jos liitännäisyyslakia yritetään soveltaa liukuluvuille, voi välisummien virhe olla erisuuruinen eri laskujärjestysten välillä.

| Osa | Selitys |
|---|---|
| 0 01111111 10000000000000000 | Liukuluvun bittiesitys |
| 0 (Positiivinen) | Etumerkki |
| $01111111_2 = 127$ | Eksponentti |
| $\lfloor(2^8 - 1)/2\rfloor$ | -127 -koodaus |
| (1)10000000000000000 | Mantissa |
| $2^{127-127} + 2^{127-128} = 1,5$ | Kymmenjärjestelmäesitys |

Taulukko 6.2: Liukulukuesitys luvulle 1,5

Yhteistä muistia käyttävä monisäikeinen logiikka voi suoritusjärjestykseltään aiheuttaa laskujärjestyksen muuttumisen ja se aiheuttaa desynkronisaation. Jos peli toteutetaan kääntäjään perustuvalla ohjelmointikielellä, voi debug- ja julkaisuversion syötetiedostot aiheuttaa toisissaan desynkronisaatioita. Kääntäjässä käytetyt optimointiliput voivat järjestää laskutoimenpiteet suorituskykyisempään järjestykseen, jolloin liukulukulaskujen lopputulokset voivat muuttua. Suorittimissa voidaan hyödyntää laajennettua liukulukutarkkuutta (*extended precision*), joka aiheuttaa epäyhteensopivuutta suorittimissa, joissa liukuluja lasketaan eri tarkkuudella.

### 6.4.4 Satunnaisluvut ja kello

Satunnaisluvut videopeleissä on tapa toteuttaa logiikkaa, jonka toiminta on arvaamatonta. Aitoja satunnaislukuja voidaan tuottaa, jos pelialustassa on järjestelmän tai luonnonilmiöden epäjärjestystä (*entropy*) mittaava oheislaite. Aidot satunnaisluvut voivat antaa täyden varmuuden pelimekaniikoille, että niitä ei voida ennustaa. Jos satunnaislukuja ei voida johtaa kaavasta, ovat ne epädeterminismiä, joka pitää tallentaa syötetiedostoon.

Jos aitoja satunnaislukuja ei voida tuottaa, voidaan pelissä hyödyntää näennäissatunnaislukuja (*pseudorandom numbers*). Aidot satunnaisluvut voidaan korvata numeroita generoivalla algoritmilla, joka tuottaa hyvin kaaottisen numerosarjan. Koska numerot eivät

oikeasti ole satunnaislukuja, kutsutaan niitä pseudosatunnaisiksi. Vaikka satunnaisluvut olisivat pseudosatunnaisia, ne voivat olla pelaajan näkökulmasta täysin arvaamattomia. Koettua satunnaisuutta voidaan käyttää niin kuin se olisi aitoa satunnaisuutta.

Pseudosatunnaislukugeneraattori (*pseudorandom number generator*) on algoritmi, joka tuottaa pseudosatunnaislukuja. Generaattoria voidaan ajatella oliona, joka muuttaa sisäistä tilaansa jokaisen tuotetun satunnaisluvun jälkeen. Pseudosatunnaislukujen tuottamiseen käytetty algoritmi voi olla yleistä tietoa, mutta jos sen lähtötilaa ei pystytä päättelemään, tuotettuja lukuja on erityisen vaikea ennustaa. Alustukseen käytettyä arvoa kutsutaan siemenluvuksi (*seed number*)

Siemenluku voidaan valita vapaasti, mutta vakioksi asetettu lähtöarvo voi johtaa pelityyleihin, joita pelinkehittäjä ei pelilleen tarkoittanut. Epädeterministisen arvon käyttäminen siemenlukuna parantaa pelin satunnaislukujen arvaamattomuutta. Vaikka pelialustassa harvoin on aitoa satunnaisuutta tuottava oheislaite, niissä on usein kello, jonka antamat arvot ovat pelin logiikan näkökulmasta arvaamattomia. Hyvä pseudosatunnaisgeneraattori tuottaa pienimmästäkin muutoksesta siemenlukuun täysin erilaisen numerosarjan.

Kun käytetään pseudosatunnaislukugeneraattoria ainoastaan lähtöarvo pitää tallentaa syötetiedostoon. Pelaajan kontrollien tuottamat syötteet ovat epädeterministisiä ja niitä voidaan käyttää tarpeettomien satunnaislukujen tuottamiseen. Tarpeettomat satunnaisluvut muuttavat generaattorin sisäistä tilaa, joka vaikuttaa tuotettuihin satunnaislukuihin. Jos satunnaislukujen kulutus on arvaamaton, satunnaisuuteen pohjautuvia pelimekaniikkoja voidaan toteuttaa, vaikka satunnaislukugeneraattori alustettaisiin aina samalla tavalla. Koska satunnaisluvut perustuvat pelaajan syötteisiin, satunnaislukuja ei tarvitse tarkistaa erikseen, jotta ne voitaisiin toistaa käänteisessä debuggaamisessa.

Kelloa voidaan myös käyttää tavallisten pelimekaniikkojen toteuttamiseen. Päivämäärä-olion arvo on tarkkuudesta riippuen joko 32 bittiä tai 64 bittiä. Jos pelissä käytetty pelisilmukka perustuu muuttuvaan delta-aikaan (ks. luku 5), niin päivitysrutiinin suorituksen ajanhetki on epädeterministinen ja se pitää tallentaa syötetiedostoon. Vakioksi asetetus-

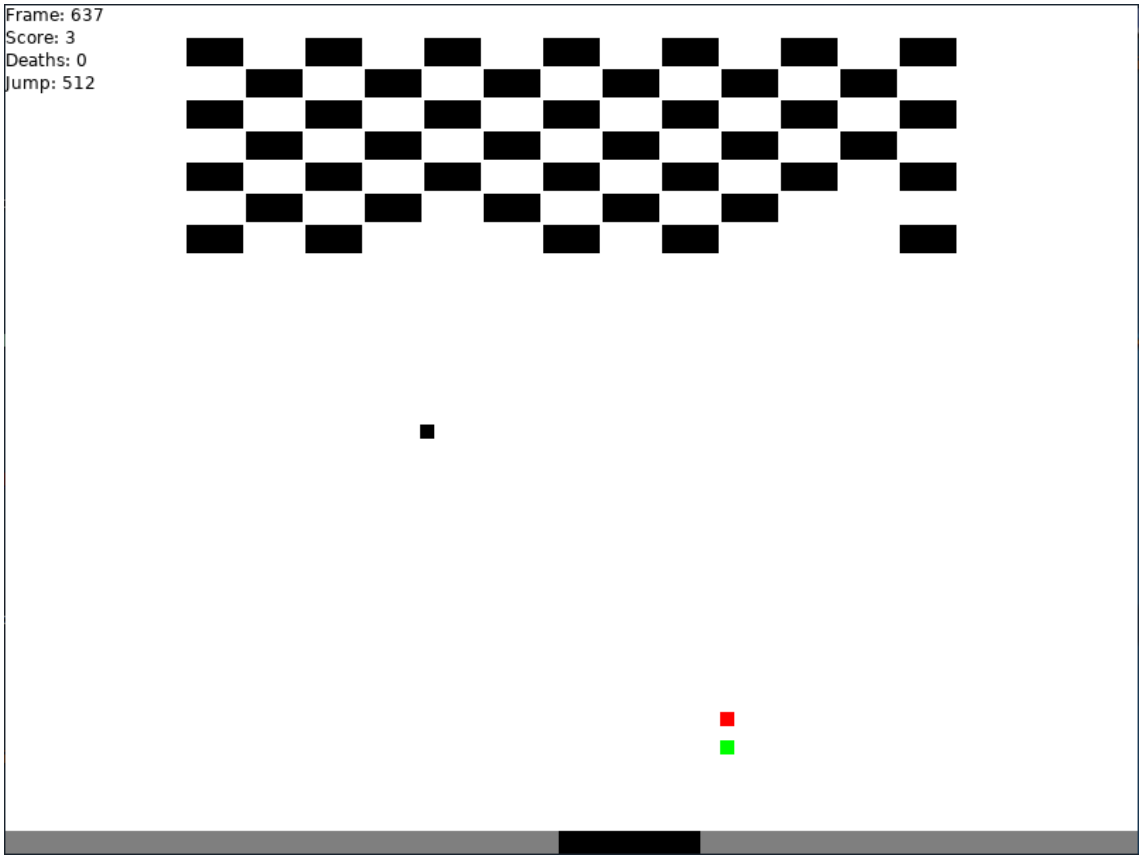
sa delta-aikaan perustuvassa pelisilmukassa ajan kulku voidaan esittää iteraatioina, jotka mallintavat sen kulkua tasaisesti. Riippuen pelisilmukan toteutuksesta ja pelialustassa olevien laskureiden laadusta, iteraatioita kontrolloiva, aikaa kerryttävä muuttuja voi sisältää virhettä. Virhe kerryttäjässä tarkoittaa, että pelin aika ei etene tarkasti reaaliajan mukaan. Jos pelisilmukan toteutus perustuu vakioksi asetettuun delta-aikaan, riittää että pelin käynnistyshetki tallennetaan syötetiedostoon ja suorituksessa nykyhetki voidaan johtaa laskemalla se pelin iteraatioiden määrästä.

## 6.5 Käänteisesti debuggatava Breakout-klooni

Tässä työssä käänteisen debuggaamisen toteutettavuutta ja käytettävyyttä tarkasteltiin toteuttamalla Breakout-pelistä klooni (ks. kuva 6.2). Peli toteutettiin käyttäen pelienkehitykseen tarkoitettua LÖVE-ohjelmistokehystä. Lisäksi pelin toteuttamista helpottamaan valittiin valmiit toteutukset vektori-luokalle ja törmäystarkastelulogiikalle. Pelikappaleiden paikat ja nopeuksien suunnat toteutettiin käyttäen vektoreita. Pelikappaleet ovat kaikki suorakulmaisia ja niiden törmäystarkastelu toteutettiin käyttäen akselinsuuntaisia rajauslaatikoita (*axis-aligned bounding boxes, AABB*).

### 6.5.1 Jäljitystiedoston muodostaminen syötetiedostosta

Käänteisen debuggauksen toteutustavaksi valittiin syötetiedoston muodostaminen. Syötetiedoston etu on se, että siitä voidaan myöhemmin muodostaa jäljitystiedosto. Syötetiedostoa käyttämällä bugit voidaan kokea samalla tavalla kuin ne koettiin, kun bugi havaittiin. Debuggaamisessa voidaan myös tehdä muutoksia pelin koodiin ja muodostaa ymmärrys bugista kokeiden kautta. Videopeleille suunnatun käänteisen debuggauksen vaatimusten (ks. luku 1.2.1) mukaan käänteinen debuggaus toteutettiin niin, että syötetiedosto muodostetaan tavallisen pelaamisen aikana. Koska bugi voi kaataa pelin milloin tahansa, pelaajan syötteet tallennettaan syötetiedostoon ennen päivitysrutiinin ajamista.




Kuva 6.2: Kuvakaappaus Breakout-kloonista

| Tiedostotyyppi | Koko | Pakattu koko | Suhde |
|---|---|---|---|
| Syötetiedosto | 720 018 tavua | 117 213 tavuaa | 16,3% |
| Jäljitystiedosto | 36 651 503 tavua | 1 085 530 tavua | 3,0% |

Taulukko 6.3: Syöte- ja jäljitystiedostojen koot

Käänteisen debuggaamisen käytettävyyttä parannettiin muodostamalla syötetiedostosta jäljitystiedosto. Jäljitystiedostoon tallennettiin pelimaailma sarjallistetussa muodossa, joka mahdollistaa sen, että nauhoitetussa peluussa voidaan ohittaa osa suorituksesta käyttämällä jäljitystiedostoa pelimaailman alustuksessa.

Jäljitystiedoston muodostaminen ei täysin toteuttanut työssä asetettua interaktiivisen debuggaamisen määritelmää (ks. luku 1.2.2). Toteutus pohjautuu siihen, että nauhoitetun peluun toistossa muodostetaan jäljitystiedosto. Tämä johtaa siihen, että peluun ensimmäisessä toistossa ei voida hypätä eteenpäin. Interaktiivisen käänteisen debuggaamisen määritelmä edellyttää, että syötetiedoston muodostuksen yhteydessä myös pelimaailmaa tallentava jäljitystiedosto muodostetaan.

Toteutukseen päädyttiin, koska pelin logiikassa haluttiin hyödyntää sarjallistamiseen luotua *bitser*-kirjastoa. Kaikki jäljitystiedostoon tallennettavat tilat kerättiin taulukkoon, joka tallennettiin tiedostoksi yhdellä kirjoituksella. *Bitser*-kirjasto ymmärtää Luan tietotyypit ja purkaa jäljitystiedostoon sarjallistetun datan sen alkuperäiseen muotoon. Oikeassa muodossa olevalla datalla on helppo alustaa pelimaailma.

Tavallisen peluun aikana tapahtuva sarjallistaminen edellyttää, että pelin tila tallennetaan ennen pelin päivittämistä seuraavaan tilaan. Koska pelin päivittäminen voi kaataa pelin, ei voida olettaa, että sarjallistaminen on mahdollista päivityksen jälkeen. Sarjallistaminen kannattaa toteuttaa peliin käytetyn ohjelmointikielen tai pelimoottorin tarjoamien työkalujen avulla. Jos pelimaailman esitykseen käytetään yksittäistä tietorakennetta, voidaan sarjallistamisessa käyttää rekursiota, joka vähentää sarjallistamisen kompleksisuutta.

Syötetiedoston etu puhtaasti jäljitystiedostoon perustuvan käänteiseen debuggaamiseen verrattuna on sen pienemmät tallennusvaatimukset. Jäljitystiedosto, joka täydentää syötetiedostoa voidaan toteuttaa harvemmaksi. Kaikkia tiloja ei tarvitse tallentaa, koska ne voidaan johtaa suorittamalla peliä eteenpäin käyttäen syötetiedostoa.

Breakout-kloonia nauhoitettiin pelaamalla 30 000 kuvanpäivityksen pituinen peli (ks. taulukko 6.3). Peli on suunniteltu pelattavaksi laitteella, joka pystyy esittämään 60 ruudunpäivitystä sekunnissa. Nauhoitettua syötetiedostoa käytettiin muodostamaan jäljitystiedosto, jossa kaikki pelin tilat tallennettiin syötetiedostoon.

Jäljitystiedostoon voidaan tallentaa vain ne pelin tilat, jotka mahdollistavat interaktiivisen käänteisen debuggaamisen. Pelistä riippuen tallennusten tiheys voi esimerkiksi olla kerran minuutissa tai kerran tunnissa. Jäljitystiedoston kokoa voidaan myös optimoida tekemällä peliin logiikkaa, joka valitsee siihen tallennettavat kohdat.

### 6.5.2 Ajan epälineaarisuus päivitysrutiinissa

Peli toteutettiin käyttäen LÖVE:n sisäänrakennettua delta-aikaan perustuvaa pelisilmukkaa. Pelisilmukassa päivitetään pelin tila suorittamalla päivitysrutiini muuttuvalla delta-ajalla. Breakout-kloonin suorituksessa delta-aika muuttui käyttöjärjestelmän antaman suoritusajan mukaan, joka näkyi käänteisessä debuggaamisessa epäjatkuvuutena. Epäjatkuvuuskohdissa peli hyppää yhden tai useamman ruudunpäivityksen yli ja sen huomaa visuaalisesti. Ajan epälineaarisuus ilmenee, kun syötetiedostosta luettu delta-aika poikkeaa odotettusta arvosta. Päivitysrutiinissa suuri delta-aika ilmenee liikkuvien pelikappaleiden odotettua suurempana paikanmuutoksena.

Peliin saadaan lineaarisesti etenevä aika, kun sen pelisilmukka toteutetaan iteraatiopohjaiseksi. Koska iteraatiopohjainen pelisilmukka myös päivittää pelin tilaa reaaliajan mukaan, katkokset laskenta-ajassa ilmenevät vastaavanlaisina hyppyinä. Vakioksi asetetussa delta-aikaan perustuvassa pelisilmukassa kerrytettyä aikaa kulutetaan ennaltamäärätyissä määrissä (ks. luku 5.2.1). Suoritusajan puutteesta toivutaan päivittämällä pelin tila useita kertoja peräkkäin. Peräkkäisissä päivityksissä käytetyt syötteet tallennetaan normaalisti syötetiedostoon, josta johtuu se, että pelin toistossa ei ole suoritusajan puutteesta ilmeneviä hyppyjä.

### 6.5.3 Iterointijärjestys ja väliohjelmistot

Pelin toteutuksessa huomattiin, että Luan standardikirjastossa kaikki iteraattorit eivät tarjoa deterministisesti määrittyvää iterointijärjestystä. Iteraattoreita käytetään *for*-silmukoissa. Silmukan iteroinnissa turvallisin lähestymistapa oli käyttää numeerista (*numeric*) *for*-silmukkaa tai semantiikaltaan parempaan *ipairs*-iteraattoria. Luan keskeisin tietotyyppi on assosatiivinen taulukko (*associative arrays*), jolla voidaan tallentaa avain-arvo pareja. Niitä käytetään muodostamaan tavallisten taulukoiden lisäksi monimutkaisia tietorakenteita. Avain-arvo -parien iterointiin käytetään *pairs*-iteraattoria, jonka iterointijärjestys on määrittelemätön. Lua on roskienkeruuseen perustuva ohjelmointikieli ja se myös hoitaa muistinhallinnan kehittäjän puolesta. Muistin alustuksen järjestykseen perustuva logiikka on epädeterminististä ja sitä ei voida nauhoittaa. Pelin toteutuksessa *pairs*-iteraattoria tarvittaan vain silloin kun iteraatiojärjestyksellä ei ole merkitystä.

Törmäystarkastelun toteutuksessa hyödynnettiin *bump*-kirjastoa. *Bump*-kirjastoa voidaan tulkita väliohjelmistona (*middleware*), joka tarjoaa oman rajapinnan akselinsuuntaisten suorakaiteiden siirtämiseen maailmaa mallintavien olioiden sisällä (*world*). Maailmaoliot mallintavat ainoastaan *bump*-kirjaston käsitystä pelimaailmasta. Koska pelimaailma ja törmäystarkastelussa käytetty maailma ovat kaksi eri käsitettä, pitää ne synkronoida pelin logiikassa erikseen vastaamaan toisiaan.

*Bump*-kirjasto valittiin, koska sillä voidaan välttää tunneloinniksi kutsutut bugit. Tunnelointi viittaa nopeiden pelikappaleiden kykyyn läpäistä muita pelikappaleita ja seiniä. Väliohjelmistot voivat osoittautua kustannustehokkaiksi ratkaisuiksi pelinkehityksessä. Pelimaailman fysiikanmallinnukseen on erityisesti useita eri vaihtoehtoja. Käänteisessä debuggaamisessa suljetun lähdekoodin väliohjelmistojen tilaa ei välttämättä pysty seuraamaan ja sarjallistamaan, mikä voi estää käänteisen debuggauksen toteuttamisen.

Väliohjelmistot ovat valmiita ohjelmistomoduuleja, joiden toteutukset antavat ratkaisuja ongelmiin. Käänteisessä debuggauksessa asetetut vaatimukset voivat olla tiukempia, mitä väliohjelmistoissa on noudatettu. Fysiikanmallinnuksessa voidaan hyödyntää satun-

naislukuja määrittämään samanaikaisten törmäysten iterointijärjestys tai parantamaan simulaation laatua [21]. Jos käytetyn satunnaislukugeneraattorin siemenlukua ei voi lukea ja asettaa syötetiedoston pohjalta, peli desynkronoituu käänteisessä debuggauksessa.

Väliohjelmistot tuovat lähdekoodiin omat käsitteensä. *Bump*-kirjasto tarjoaa maailmaolioiden lisäksi tavara-oliot (*item*), joita sen sisäisessä maailmassa liikutetaan. Törmäystarkastelu määrittää kappaleiden todellisen sijainnin, kun kappaleet liikkuvat. Todellinen sijainti synkronoidaan pelikappaleisiin, jotta ne voidaan piirtää oikeaan kohtaa ruutua. Kun pallo osuu tiileen, se kimpoaa siitä ja tiili tuhoutuu.

### 6.5.4 Pelikappaleiden temporaalisuus

Tiilien luominen ja tuhoaminen muodostuu ongelmalliseksi, kun käänteisessä debuggaamisessa palataan aikaisempiin tiloihin. Aikaisemmassa suorituksen ajanhetkessä pelikappaleiden lukumäärä ei ole välttämättä sama. Vaikka pelikappaleiden lukumäärä olisi sama, ei voida olettaa, että nykyhetken pelikappaleet ovat olemassa myös menneisyydessä.

Breakout-kloonin osalta tiilien temporaalisuus ei tuottanut ongelmia, koska käytetty törmäystarkastelu ei käytä suoria viittauksia pelikappaleisiin. Kaikki olemassaolevat tiilet voidaan tuhota ja tarvittava määrä tiiliä voidaan alustaa tuhottujen tiilien tilalle. Suorien viittausten sijaan törmäystarkastelu rakentaa pelikappaleista tietorakenteen, jota selaamalla löydetään keskenään törmäävät kappaleet.

Olioiden viittaaminen toisiin oliohin on tavallista korkean tason ohjelmointikielissä. Pelikappaleet jotka viittaavat toisiin pelikappaleisiin pitää sarjallistaa niin, että myös viitattu pelikappale sarjallistetaan. Koko pelimaailman esittäminen yksittäisellä tietorakenteella, jonka voi sarjallistaa rekursiivisesti, antaa joustavuutta aikamatkustamisessa. Kun ajassa matkustetaan, voidaan pelimaailman sarjallistettu tila purkaa tarkalleen sellaiseksi kuin se oli sarjallistamisen yhteydessä.

Breakout-kloonissa pelikappaleet esitettiin ylimmän näkyvyysalueen paikallisina muuttujina. Tiilet tallennettiin taulukkoon. Sarjallistamisen purkulogiikassa pelikappaleisiin

voitiin viitata niiden alustetuista koordinaatteista johdettujen tunnisteiden avulla. Koska suoria viittauksia ei sarjallistettu, piti tiilet alustaa käyttäen silmukkaa ja yhtäsuuruusvertailua. Kun sarjallistetun tiilen ja pelissä olevan tiilen tunnisteet olivat samat, kopioitiin sarjallistetusta tiilestä tiedot pelissä olevaan tiileen.

Osa sarjallistamisen haasteellisuudesta ratkaistiin käyttämällä olioiden kierrättämisen mallia (*object pool*). Olioiden kierrättämisellä ehkäistään muistin pirstaloitumista uudelleenkäyttämällä ei-aktiivisia olioita uusien aktiivisten olioiden luomisessa. [22]. Kun pelikappale tuhoutuu, se silti säilytetään pelin muistissa. Kun vastaavanlainen pelikappale halutaan luoda, katsotaan ensiksi, onko ei-aktiivisia pelikappaleita, jotka voidaan ottaa uudelleen käyttöön. Tehokkaamman muistinkäytön lisäksi tämä vähentää tarvetta ajaa roskienkeruualgoritmia.

Breakout-kloonissa kaikki tarvitut pelikappaleet luodaan pelin alustuksessa ja niitä ei tuhota missään vaiheessa. Kun pallo osuu tiileen, se siirretään pois pelialueelta. Kun kenttä läpäistään ja tarvitaan uusia tiiliä, palautetaan kentältä poistettuja tiiliä takaisin peliin.

# Luku 7

# Pohdintaa

## 7.1 Kaikkitietävä debuggaaminen

Tulevaisuuden käänteisen debuggaamisen prosessi tulee sisällyttämään yhä parempia debuggaamistekniikoita. Käänteistä debugaamista voidaan ajatella eräänlaisena uusia debuggaustekniikoita mahdollistavana alustana. Tyypillisesti debuggaamisessa keskitytään suorituksen yksittäisiin hetkiin, mutta tilan muuttumisen tarkastelunäkökulma voisi olla parempi kun tarkastellaan kokonaisia ajanjaksoja.

Microsoftin *TTD*:ssa ohjelman toistettua suoritusta voidaan tarkastella ohjelmallisesti. Ohjelman suorituksesta muodostetaan tietokanta, johon voidaan suorittaa hakuja käyttämällä Microsoftin kehittämää *LINQ*-syntaksilla. Ohjelmallisesti ajanjaksoja tarkastelevaa debuggaamista voidaan kutsua kaikkitietäväksi debuggaamiseksi (*omniscient debugging*).

Toinen kaikkitietävää debuggaamista hyödyntävä tuote on *Pernosco*. *Pernoscon* taustalla on Mozillan *RR*:n ylläpitäjät. Debuggaustyökalut tarjotaan palveluna (software as a service) ja se on yhteensopiva jatkuvaan integratioon perustuvaan tuotantoon. Pernosco ymmärtää *RR*:n tuottamia jäljitystiedostoja ja muodostaa niistä tietokantoja. [23]

Tietokantoja voidaan käyttää rajaamaan debuggaamisen näkökulmasta kiinnostavia kohtia. Ne voidaan ilmaista yksityiskohtaisella hakutermillä. Jos tietokanta tallennetaan aikasarjatietokannaksi (*time series database*), voidaan sillä suorittaa tilan muutokseen liit-

tyviä analyysejä. Tietokannasta saatu vastaus voi kuvailla esimerkiksi sitä, milloin tietty arvo sijoitettiin muuttujaan tai milloin funktiota ajettiin tietyillä argumenteillä.

## 7.2 Käänteinen debuggaaminen pelisuunnittelussa

Käänteinen debuggaus voi olla hyödyllinen pelien laadunvarmistuksessa, mutta myös pelien suunnittelemisessa. Kaikkitietävää debuggaamista voidaan käyttää tarkastamaan pelisuunnittelussa asetettuja ehtoja. Ehto voi esimerkiksi olla, että tasohyppelypelissä ei kerätä ainuttakaan kolikkoa. Automaattisesti testejä ajava järjestelmä voi etsiä ehdon täyttävän peluun suorittamalla peliin muodostettuja syötetiedostoja.

Käänteisen debuggaamisen toteuttanut peli voi käyttää toistettavuuttaan pelitestaamisessa. Toistettavuutta voidaan hyödyntää niin, että pelitestaaminen voidaan lopettaa mihin tahansa kohtaan peliä ja tulevilla pelitestauskerroilla peluu voidaan toistaa syötetiedostosta ja jatkaa siitä, mihin jäätiin. Pelitestaaminen voidaan keskittää myös niin, että eri pelin osat saavat enemmän testaamista kuin toiset. Syötetiedostoja voidaan käyttää myös tunnistamaan pelimekaniikkoihin tehdyt muutokset. Kun toistetut peluut desynkronisoituvat, voi pelinkehittäjä todeta desynkronisaation tarkoituksenmukaisuuden.

# Luku 8

# Yhteenveto

Tässä työssä johdettiin videopeleille soveltuva käänteisen debuggaamisen toteutusmalli. Työ rakentuu tutkimusosuudesta ja kokeellisesta osuudesta. Tutkimusosuudessa tutustuttiin Mozillan ja Microsoftin tekemiin käänteistä debuggaamista koskeviin tutkimuksiin. Tutkimuksista johdetut havainnot yhdistettiin videopelikonsolien emulaattoreissa käytettyihin laadunvarmistustekniikoihin ja eroteltiin kaksi toisiaan täydentävää käänteisen debuggaamisen toteutustapaa.

Lähestymistapojen soveltuvuutta pohditiin videopelien asettamien suoritusvaatimusten mukaan ja niistä valittiin toinen. Valitun toteutusmallin tueksi tutkittiin olemassaolevaa käänteisen debuggaamisen mahdollistavaa tilanhallintaa web-sovelluksista. Web-sovelluksista havaittiin, että debuggattavat osat voivat olla abstrakteja ja abstraktoitu logiikka ei välttämättä ole käänteisesti debugattavissa.

Työssä esitettiin kaksi tutkimuskysymystä:

1. Voidaanko videopeliä debugata käänteisesti?
2. Miten käänteistä debuggausta saadaan interaktiivisemmaksi?

Esitetyille kysymyksille tehtiin tarkat määritykset, joista voitiin myös johtaa toimintavaatimukset konkreettiselle toteutukselle.

Ensimmäiseen kysymykseen vastattiin, että videopelien suorittaminen ei ole peruutettavissa. Pelin aikaisemmat tilat voidaan peruuttamisen sijaan esittää sovelluksen tarkan

toistamisen avulla. Pelien toistamiseen esitettiin pessimistisen oletukset, jotka johtivat siihen johtopäätökseen, että videopelien käänteinen debuggaamisen mahdollistava nauhoitus pitää olla aina käytössä.

Toiseen kysymykseen vastattiin, että käänteinen debuggaaminen on liian hidasta jos se vie suoraan verrannollisesti aikaa alkuperäiseen nauhoitukseen nähden. Käänteisen debuggaamisen nopeuttamiseksi esitettiin ratkaisuna koko pelin tilan ajoittaista tallentamista, joka mahdollistaa käänteisen debuggaamisen aloittamisen keskeltä nauhoitettua peluuta.

Työn kokeellinen osuus perustui Breakout-pelistä tuotettuun klooniin, johon oli toteutettu käänteinen debuggaus. Pelissä pystyttiin onnistuneesti nauhoittamaan peliin vaikuttava epädeterminismi syötetiedostoksi, josta pelin suoritus pystytään toistamaan. Interaktiivinen käänteinen debuggaaminen pystyttiin tuottamaan, mutta peliin toteutettu tilan sarjallistaminen aiheutti odottamattoman suuren lähtökustannuksen. Käänteinen debuggaaminen muuttui interaktiiviseksi vasta, kun peluu toistettiin kokonaisuudessaan ja siitä muodostettiin jäljitystiedosto.

Nauhoitettujen syötteiden toistamiseen kuluu aikaa suoraan verrannollisesti nauhoitettuun peluuseen nähden, mikä on hitaampaa kuin mitä työssä vaaditiin. Jäljitystiedoston muodostamisen jälkeen käänteinen debuggaaminen on interaktiivista. Interaktiivisen debuggaamisen onnistuminen riippuu siitä, kuinka monta kertaa nauhoitettuja peluita pitää toistaa, jotta niiden esittämät bugit voidaan korjata. Paremmalla sarjallistamisen strategialla interaktiivisuuden toteuttamisen haasteita pystytään lievittämään tai kokonaan välttämään.

# Lähdeluettelo


[1] GamingScan 2020 Gaming Industry Statistics, Trends & Data (Large-Scale Study). URL: https://www.gamingscan.com/gaming-statistics/ [Online; Viitattu 2019-10-24].

[2] Newzoo 2018 Global Games Market per Device Segment (1867×1050). URL: https://newzoo.com/wp-content/uploads/2016/03/Newzoo_2018_Global_Games_Market_per_Device_Segment.png [Online; Viitattu 2019-03-11].

[3] Stuart Brown. The First Video Game. URL: https://www.youtube.com/watch?v=uHQ4WCU1WQc [Online; Viitattu 2019-10-24].

[4] Jakob Engblom. A review of reverse debugging. Teoksessa *Proceedings of the 2012 System, Software, SoC and Silicon Debug Conference*, ss. 1–6. ISSN: 2114-3684, 2114-3684.

[5] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll ja Nimrod Partush. Engineering Record And Replay For Deployability: Extended Technical Report. URL: http://arxiv.org/abs/1705.05937 [Online; Viitattu 2019-03-20]. 1705.05937.

[6] Sanjay Bhansali, Wen-Ke Chen, Stuart de Jong, Andrew Edwards, Ron Murray, Milenko Drinic, Darek Mihočka ja Joe Chau. Framework for Instruction-level Tracing and Analysis of Program Executions. s. 10.

[7] James McNellis, Jordi Mola ja Ken Sykes. “Time Travel Debugging: Root Causing Bugs in Commercial Scale Software” - YouTube. URL: https://www.youtube.com/watch?v=l1YJTg_A914 [Online; Viitattu 2019-11-17].

[8] The Top Coin-Operated Videogames of All Time - The International Arcade Museum. URL: https://www.arcade-museum.com/TOP100.php [Online; Viitattu 2019-10-24].

[9] Floating Point Determinism | Gaffer On Games. URL: https://gafferongames.com/post/floating_point_determinism/ [Online; Viitattu 2019-11-15].

[10] The State of Babel · Babel. URL: https://babeljs.io/blog/2016/12/07/the-state-of-babel [Online; Viitattu 2019-10-31].

[11] ECMAScript - Wikipedia. URL: https://en.wikipedia.org/wiki/ECMAScript [Online; Viitattu 2019-11-03]. Page Version ID: 922765351.

[12] Render-tree Construction, Layout, and Paint | Web Fundamentals. URL: https://developers.google.com/web/fundamentals/performance/critical-rendering-path/render-tree-construction [Online; Viitattu 2019-08-20].

[13] Modulecounts. URL: http://www.modulecounts.com/ [Online; Viitattu 2019-10-31].

[14] React – A JavaScript library for building user interfaces. URL: https://reactjs.org/index.html [Online; Viitattu 2019-03-19].

[15] Redux · A Predictable State Container for JS Apps. URL: https://redux.js.org/ [Online; Viitattu 2019-11-02].

[16] Glenn Fiedler. Fix Your Timestep! | Gaffer On Games. URL: https://gafferongames.com/post/fix_your_timestep/ [Online; Viitattu 2019-03-11].

[17] Jouni Smed ja Harri Hakonen. *Algorithms and Networking for Computer Games*. John Wiley & Sons, Ltd. ISBN 978-0-470-02975-6 978-0-470-01812-5.

[18] Super Mario Bros. - SDA Knowledge Base. URL: https://kb.speeddemosarchive.com/Super_Mario_Bros. [Online; Viitattu 2019-10-28].

[19] Guides - Super Mario Bros. - speedrun.com. URL: https://www.speedrun.com/smb1/guides [Online; Viitattu 2019-10-28].

[20] Latauksen suositellut koodausasetukset - YouTube Ohjeet. URL: https://support.google.com/youtube/answer/1722171?hl=fi [Online; Viitattu 2019-10-29].

[21] Deterministic Lockstep | Gaffer On Games. URL: https://gafferongames.com/post/deterministic_lockstep/ [Online; Viitattu 2019-11-12].

[22] Robert Nystrom. *Game programming patterns*. Genever Benning. ISBN 978-0-9905829-0-8. OCLC: 897177209.

[23] Pernosco. URL: https://pernos.co/ [Online; Viitattu 2019-11-29].

# Liite A

# Breakout-peli käänteisellä debuggauksella

```
local mode = "load"
local filename = "breakout_input_file.json"
local keyframe_interval = 10

-- Dependencies
-- https://github.com/vrld/hump
-- https://github.com/kikito/bump.lua

local vector = require "vector"
local bump   = require "bump"
local bitser = require "bitser"
local levels = require "levels"

local rng = love.math.newRandomGenerator(os.time())
local WIDTH, HEIGHT = love.graphics.getDimensions()

local red, green, blue = {1, 0, 0}, {0, 1, 0}, {0, 0, 1}
local white, black = {1, 1, 1}, {0, 0, 0}
local paddle_track = {0.5, 0.5, 0.5}

local foreground = black
local background = white
love.graphics.setColor(foreground)
```

```
24 love.graphics.setBackgroundColor(background)
25
26 local world = bump.newWorld()
27 local ordered_frames = {}
28 local keyframes = {}
29
30 function initialize()
31   local header_format = "c18"
32   local frame_format = "d d d"
33
34   if mode == "save" then
35     love.filesystem.remove(filename)
36
37     love.filesystem.append(filename, love.data.pack("string",
38             header_format, rng:getState()))
39   end
40
41   if mode == "load" then
42     local total_size = love.filesystem.getInfo(filename).size
43     local data = love.filesystem.read(filename)
44     local position = 1
45
46     local rng_state
47     rng_state, position = love.data.unpack(header_format, data,
48             position)
49
50     rng:setState(rng_state)
51
52     while (position < total_size) do
53       local framecount, x, dt
54       framecount, x, dt, position = love.data.unpack(frame_format,
55               data,position)
56
57       ordered_frames[framecount] = {}
58
59       ordered_frames[framecount].input = {}
60       ordered_frames[framecount].input.direction = {}
61       ordered_frames[framecount].input.direction.x = x
62
63       ordered_frames[framecount].dt = dt
64     end
65
66     keyframes = bitser.loads(love.filesystem.read(
```

```
            "breakout_trace_file.txt"))
  end
end

initialize()

local dt = 0
local input = {}

local level = 0
local score = 0
local deaths = 0
local framecount = 0
local pause_frame = 10000
local stop_to_pause_frame = false

local tunneler = {
  position = vector(0, 500),
  width = 10, height = 10,
  direction = vector(1, 0),
  speed = 100,
  color = red,
}

local bouncer = {
  position = vector(0, 520),
  width = 10, height = 10,
  direction = vector(1, 0),
  speed = 100,
  color = green,
}

local ball = {
  position = vector((WIDTH / 2) - 5, (HEIGHT / 2) - 5),
  width = 10, height = 10,
  direction = vector(1, 0)
    :rotateInplace((1 / 4) * math.pi)
    :rotateInplace((1 / 2) * math.pi * rng:random()),
  speed = 300,
  color = black
}

local paddle = {
```

```
110   position = vector(500, HEIGHT - 15 - 1),
111   width = 100, height = 15,
112   direction = vector(0, 0),
113   speed = 500,
114   color = black
115 }
116
117 local reflection_point = vector(paddle.width / 2, paddle.width / 2)
118
119 local bricks = {}
120
121 local ceiling = {
122   color = red,
123   x = -1, y = -1,
124   width = WIDTH + 2, height = 1
125 }
126
127 local right_wall = {
128   color = red,
129   x = WIDTH, y = 0,
130   width = 1, height = HEIGHT
131 }
132
133 local left_wall = {
134   color = red,
135   x = -1, y = 0,
136   width = 1, height = HEIGHT
137 }
138
139 local floor = {
140   color = red,
141   x = -1, y = HEIGHT - paddle.height,
142   width = WIDTH + 2, height = 1
143 }
144
145 local progress_line = {
146   color = red,
147   x = -1, y = HEIGHT / 2,
148   width = WIDTH + 2, height = 1
149 }
150
151 local paddle_left_limit = {
152   color = red,
```

```
153   x = -1 - (paddle.width * 0.90), y = 0,
154   width = 1, height = HEIGHT
155 }
156
157 local paddle_right_limit = {
158   color = red,
159   x = WIDTH + (paddle.width * 0.90), y = 0,
160   width = 1, height = HEIGHT
161 }
162
163 world:add(bouncer, bouncer.position.x, bouncer.position.y,
164         bouncer.width, bouncer.height)
165
166 world:add(ball, ball.position.x, ball.position.y,
167         ball.width, ball.height)
168
169 world:add(paddle, paddle.position.x, paddle.position.y,
170         paddle.width, paddle.height)
171
172 world:add(ceiling, ceiling.x, ceiling.y,
173         ceiling.width, ceiling.height)
174
175 world:add(right_wall, right_wall.x, right_wall.y,
176         right_wall.width, right_wall.height)
177
178 world:add(left_wall, left_wall.x, left_wall.y,
179         left_wall.width, left_wall.height)
180
181 world:add(floor, floor.x, floor.y,
182         floor.width, floor.height)
183
184 world:add(progress_line, progress_line.x, progress_line.y,
185         progress_line.width, progress_line.height)
186
187 world:add(paddle_left_limit, paddle_left_limit.x, paddle_left_limit.y,
188         paddle_left_limit.width, paddle_left_limit.height)
189
190 world:add(paddle_right_limit, paddle_right_limit.x, paddle_right_limit.y,
191         paddle_right_limit.width, paddle_right_limit.height)
192
193 function moveNaive(object, dt)
194   local WIDTH, HEIGHT = love.graphics.getDimensions()
195
```

```
196   object.position = object.position +
197           (object.direction * object.speed * dt)
198
199   if (object.position.x < 0) then
200     object.position.x = 0
201     object.direction.x = object.direction.x * -1
202   end
203
204   if (object.position.x > (WIDTH - object.width)) then
205     object.position.x = WIDTH - object.width
206     object.direction.x = object.direction.x * -1
207   end
208
209   if (object.position.y < 0) then
210     object.position.y = 0
211     object.direction.y = object.direction.y * -1
212   end
213
214   if (object.position.y > (HEIGHT - object.height)) then
215     object.position.y = HEIGHT - object.height
216     object.direction.y = object.direction.y * -1
217   end
218 end
219
220 function create_brick(x, y)
221   local brick = {
222     position = vector((x - 1) * 40 + x * 2, (y - 1) * 20 + y * 2),
223     width = 40,
224     height = 20,
225     brick = true,
226     color = black,
227     grid_x = x,
228     grid_y = y,
229     remove = true
230   }
231
232   table.insert(bricks, brick)
233
234   world:add(brick, brick.position.x, brick.position.y,
235           brick.width, brick.height)
236 end
237
238 function refresh_bricks(level_geometry)
```

```
239   local items, len = world:getItems()
240   local lookup = {}
241
242   for i = 1, len do
243     local item = items[i]
244     local key = tostring(item.grid_x)..","..tostring(item.grid_y)
245
246     lookup[key] = item
247   end
248
249   for y = 1, #level_geometry do
250     for x = 1, #level_geometry[y] do
251       local c = level_geometry[y]:sub(x,x)
252       local key = tostring(x)..","..tostring(y)
253       lookup[key].remove = c == "."
254     end
255   end
256 end
257
258 function grid_to_screen(x, y)
259   return (x - 1) * 40 + x * 2, (y - 1) * 20 + y * 2
260 end
261
262 function update_geometry()
263   local items, len = world:getItems()
264
265   for i = 1, len do
266     local item = items[i]
267
268     if item.remove ~= nil then
269       item.position.x, item.position.y = grid_to_screen(
270               item.grid_x,item.grid_y)
271
272       world:update(item, item.position.x, item.position.y)
273     end
274
275     if item.remove == true then
276       item.position.x, item.position.y = -1000, -1000
277
278       world:update(item, item.position.x, item.position.y)
279     end
280   end
281 end
```

```
282
283 function create_keyframe()
284   local items, len = world:getItems()
285   local alive = {}
286
287   for i = 1, len do
288     local item = items[i]
289
290     if item.brick == true then
291       local key = tostring(item.grid_x)..","..tostring(item.grid_y)
292
293       alive[key] = item.remove
294     end
295   end
296
297   keyframes[framecount] = {
298     rng_state = rng:getState(),
299
300     tunneler_position_x = tunneler.position.x,
301     tunneler_position_y = tunneler.position.y,
302     tunneler_direction_x = tunneler.direction.x,
303
304     bouncer_position_x = bouncer.position.x,
305     bouncer_position_y = bouncer.position.y,
306     bouncer_direction_x = bouncer.direction.x,
307
308     ball_position_x = ball.position.x,
309     ball_position_y = ball.position.y,
310     ball_direction_x = ball.direction.x,
311     ball_direction_y = ball.direction.y,
312
313     paddle_position_x = paddle.position.x,
314     paddle_position_y = paddle.position.y,
315     paddle_direction_x = paddle.direction.x,
316
317     alive = alive,
318
319     level = level,
320     score = score,
321     deaths = deaths
322   }
323 end
324
```

```
325 function load_keyframe(target_frame)
326   local keyframe = keyframes[target_frame]
327 
328   rng:setState(keyframe.rng_state)
329 
330   tunneler.position.x  = keyframe.tunneler_position_x
331   tunneler.position.y  = keyframe.tunneler_position_y
332   tunneler.direction.x = keyframe.tunneler_direction_x
333 
334   world:update(bouncer, keyframe.bouncer_position_x,
335           keyframe.bouncer_position_y)
336   bouncer.position.x  = keyframe.bouncer_position_x
337   bouncer.position.y  = keyframe.bouncer_position_y
338   bouncer.direction.x = keyframe.bouncer_direction_x
339 
340   world:update(ball, keyframe.ball_position_x,
341           keyframe.ball_position_y)
342   ball.position.x  = keyframe.ball_position_x
343   ball.position.y  = keyframe.ball_position_y
344   ball.direction.x = keyframe.ball_direction_x
345   ball.direction.y = keyframe.ball_direction_y
346 
347   world:update(paddle, keyframe.paddle_position_x,
348           keyframe.paddle_position_y)
349   paddle.position.x  = keyframe.paddle_position_x
350   paddle.position.y  = keyframe.paddle_position_y
351   paddle.direction.x = keyframe.paddle_direction_x
352 
353   local items, len = world:getItems()
354 
355   for i = 1,len do
356     local item = items[i]
357 
358     if item.brick == true then
359       local key = tostring(item.grid_x)..","..tostring(item.grid_y)
360 
361       item.remove = keyframe.alive[key]
362     end
363   end
364 
365   level  = keyframe.level
366   score  = keyframe.score
367   deaths = keyframe.deaths
```

```
368
369   framecount = target_frame
370
371   update_geometry()
372 end
373
374 function contact_response(item, other)
375   if item == ball and other.brick == true then
376     return "bounce"
377   end
378
379   if item == bouncer and other == ball then
380     return "cross"
381   end
382
383   if item == ball and other == bouncer then
384     return "cross"
385   end
386
387   if item == ball and other == progress_line then
388     return "cross"
389   end
390
391   if item == paddle and other == paddle_left_limit then
392     return "touch"
393   end
394
395   if item == paddle and other == paddle_right_limit then
396     return "touch"
397   end
398
399   if item == paddle and other == left_wall then
400     return "cross"
401   end
402
403   if item == paddle and other == right_wall then
404     return "cross"
405   end
406
407   if item == paddle and other == floor then
408     return "cross"
409   end
410
```

```
411   if item.brick and other.brick then return "cross" end
412
413   if item.brick and other == floor then return "cross" end
414   if item.brick and other == right_wall then return "cross" end
415   if item.brick and other == floor then return "cross" end
416   if item.brick and other == left_wall then return "cross" end
417
418   if other.brick and item == floor then return "cross" end
419   if other.brick and item == right_wall then return "cross" end
420   if other.brick and item == floor then return "cross" end
421   if other.brick and item == left_wall then return "cross" end
422
423   if item == paddle and other == ball then return "bounce" end
424   if item == ball and other == paddle then return "bounce" end
425   if item == ball and other == floor then return "bounce" end
426
427   return "bounce"
428 end
429
430 function collision_effects(cols)
431   for _,col in ipairs(cols) do
432     if col.item == bouncer then
433       if col.other == right_wall or col.other == left_wall then
434         bouncer.direction.x = bouncer.direction.x * -1
435       end
436     end
437
438     if col.item == ball and col.other == ceiling then
439       ball.direction.y = ball.direction.y * -1
440     end
441
442     if col.item == ball and col.other == floor then
443       ball.direction.y = ball.direction.y * -1
444
445       deaths = deaths + 1
446
447       ball.position.x, ball.position.y = WIDTH / 2, HEIGHT / 2
448       ball.direction = vector(1, 0)
449               :rotateInplace((1 / 4) * math.pi)
450               :rotateInplace((1 / 2) * math.pi * rng:random())
451     end
452
453     if col.item == ball and col.other == left_wall then
```

```
454       ball.direction.x = ball.direction.x * -1
455     end
456
457     if col.item == ball and col.other == right_wall then
458       ball.direction.x = ball.direction.x * -1
459     end
460
461     if col.item == ball and col.other == progress_line then
462       local alive_count = 0
463
464       for i = 1, #bricks do
465         if bricks[i].remove ~= true then
466           alive_count = alive_count + 1
467         end
468       end
469
470       if alive_count == 0 then
471         if level == #levels then level = 0 end
472
473         level = level + 1
474         refresh_bricks(levels[level])
475       end
476     end
477
478     if col.item == ball and col.other.brick == true then
479       if (col.normal.x ~= 0) then
480         ball.direction.x = ball.direction.x * -1
481       end
482
483       if (col.normal.y ~= 0) then
484         ball.direction.y = ball.direction.y * -1
485       end
486
487       score = score + 1
488
489       col.other.remove = true
490     end
491
492     if col.item == paddle and col.other == ball
493             or col.item == ball and col.other == paddle then
494
495       ball.direction = (
496               ball.position - (paddle.position + reflection_point)
```

```
497       ):normalizeInplace()
498     end
499   end
500 end
501
502 function move(object, dt)
503   local target = object.position +
504           (object.direction * object.speed * dt)
505
506   object.position.x, object.position.y, cols, len = world:move(
507           object, target.x, target.y, contact_response)
508
509   collision_effects(cols)
510 end
511
512 function abstract_input()
513   for i,v in pairs(input) do
514     input[i] = nil
515   end
516
517   input.direction = {x = 0, v = 0}
518
519   if love.keyboard.isDown("a") then
520     input.direction.x = input.direction.x - 1
521   end
522
523   if love.keyboard.isDown("d") then
524     input.direction.x = input.direction.x + 1
525   end
526 end
527
528 function save_input()
529   if mode == "save" then
530     local format = "d d d"
531
532     love.filesystem.append(filename, love.data.pack("string",
533             format, framecount, input.direction.x, dt))
534   end
535 end
536
537 function consume_input()
538   if mode == "load" then
539     input.direction.x = ordered_frames[framecount].input.direction.x
```

```
540       dt = ordered_frames[framecount].dt
541     end
542   end
543
544   function draw_object(object)
545     local r, g, b, a = love.graphics.getColor()
546
547     if (object.color ~= nil) then
548       love.graphics.setColor(object.color)
549     end
550
551     if object.remove then love.graphics.setColor(red) end
552
553     love.graphics.rectangle("fill", object.position.x, object.position.y,
554             object.width, object.height)
555     love.graphics.setColor(r, g, b, a)
556   end
557
558   function time_travel()
559     local closest = 1
560
561     for i,_ in pairs(keyframes) do
562       if i > closest and i <= pause_frame then
563         closest = i
564       end
565     end
566
567     load_keyframe(closest)
568   end
569
570   function love.load()
571     for y = 1,9 do
572       for x = 1,19 do
573         create_brick(x, y)
574       end
575     end
576   end
577
578   function love.update(delta_time)
579     if love.keyboard.isDown("escape") then
580       if mode == "load" then
581         love.filesystem.remove("breakout_trace_file.txt")
582
```

```
583       love.filesystem.append("breakout_trace_file.txt",
584               bitser.dumps(keyframes))
585     end
586
587
588     love.event.quit()
589   end
590
591
592
593   if mode == "load" and ordered_frames[framecount + 1] == nil then return
594   if stop_to_pause_frame and framecount >= pause_frame then return end
595
596   framecount = framecount + 1
597   dt = delta_time
598
599   abstract_input()
600   save_input()
601   consume_input()
602
603   paddle.direction.x = input.direction.x
604
605   moveNaive(tunneler, dt)
606   move(bouncer, dt)
607
608   move(ball, dt)
609   move(paddle, dt)
610
611   if framecount == 1 or framecount % keyframe_interval == 0 then
612     create_keyframe()
613   end
614
615   update_geometry()
616 end
617
618 function love.draw()
619   love.graphics.print("Frame: "..tostring(framecount))
620   love.graphics.print("Score: "..tostring(score), 0, 16)
621   love.graphics.print("Deaths: "..tostring(deaths), 0, 32)
622
623   love.graphics.setColor(paddle_track)
624   love.graphics.rectangle("fill", 0, paddle.position.y,
625           WIDTH, paddle.height + 1)
```

```
626   love.graphics.setColor(foreground)
627
628   if stop_to_pause_frame then
629     love.graphics.print("Pause: "..tostring(pause_frame), 0, 48)
630   else
631     love.graphics.print("Jump: "..tostring(pause_frame), 0, 48)
632   end
633
634   if love.keyboard.isDown("backspace") then return end
635
636   draw_object(tunneler)
637   draw_object(bouncer)
638
639   for i = 1, #bricks do draw_object(bricks[i]) end
640
641   draw_object(paddle)
642   draw_object(ball)
643 end
644
645 function love.keypressed(key)
646   if key == "left"  then pause_frame = pause_frame - 1 end
647   if key == "right" then pause_frame = pause_frame + 1 end
648   if key == "f6"    then pause_frame = pause_frame - 10 end
649   if key == "f7"    then pause_frame = pause_frame + 10 end
650   if key == "f5"    then pause_frame = pause_frame - 100 end
651   if key == "f8"    then pause_frame = pause_frame + 100 end
652   if key == "f4"    then pause_frame = pause_frame - 1000 end
653   if key == "f9"    then pause_frame = pause_frame + 1000 end
654
655   if key == "b" then debug.debug() end
656   if key == "r" then pause_frame = 1 end
657   if key == "p" then stop_to_pause_frame = not stop_to_pause_frame end
658
659   if key == "space" then
660     time_travel()
661   end
662 end
663
664 function love.mousemoved(x, y)
665   if love.mouse.isDown(1) then
666     pause_frame = x + y * 50
667   end
668
```

```
669   if love.keyboard.isDown("space") then
670     time_travel()
671   end
672 end
```

Listaus A.1: Breakout-peli käänteisellä debuggauksella (Lua)