Roni Rissanen

UUSINTATOIMINNALLISUUDEN TOTEUTTAMINEN UNITY-PELIMOOTTORISSA

Opinnäytetyö

Insinööri (AMK)

Peliohjelmoinnin koulutus

2024

Kaakkois-Suomen ammattikorkeakoulu

Kaakkois-Suomen ammattikorkeakoulu

Tutkintonimike	Insinööri (AMK)
Tekijä/Tekijät	Roni Rissanen
Työn nimi	Uusintatoiminnallisuuden toteuttaminen Unity-pelimoottorissa
Toimeksiantaja	Kaakkois-Suomen ammattikorkeakoulu, Xamk, Gamelab
Vuosi	2024
Sivut	36 sivua
Työn ohjaaja(t)	Pekka Vilpponen

TIIVISTELMÄ

Opinnäytetyön tarkoituksena oli kehittää uusintatoiminnallisuus (replay) Unity-pelimoottorissa. Ominaisuus tuli pulmapeliin, jossa pelaajan tekemät liikkeet toistetaan niiden tapahduttua. Pelaaja tallentaa itsestään kopioita, jotka auttavat pelaajaa ratkaisemaan pulmia. Työn löydökset pätevät myös perinteisempiin uusintaominaisuuksiin, joita on esimerkiksi e-urheilu peleissä otteluiden katsomista varten. Projekti tuottaa toimeksiantajalle ja alalle yleisesti tietoa peli- ja fysiikkamoottoreista sekä uusintatoiminnallisuuksista. Ominaisuus on toteutettu Unity-pelimoottorissa, mutta työn tietoja voi soveltaa myös muissa ympäristöissä.

Työssä analysoitiin aiemmin tehtyjä ratkaisuja. Tutkimuksen aikana löytyi kaksi varteenotettavaa vaihtoehtoa - snapshot-ratkaisu ja deterministinen ratkaisu. Ennen kehitystyön aloitusta tehtyjen laskelmien perusteella deterministinen ratkaisu oli muistinkäytön kannalta paljon tehokkaampi, jonka perusteella se valittiin toteutettavaksi. Deterministisen ratkaisun kannalta olennaisia pelimoottorin ominaisuuksia käydään työssä läpi. Suuri deterministisen ratkaisun ongelma on vaatimus deterministiselle fysiikalle, johon Unityn EC-framework järjestelmän mukana tuleva fysiikkamoottori ei pysty. Työssä käydään läpi syitä ja ratkaisuja ongelmaan.

Työn tuloksena valmistui uusintatoiminnallisuus. Ominaisuus kehitettiin deterministä ratkaisua mukaillen. Toiminnallisuuden koodia esitellään kuvakaappausten avulla. Deterministinen ratkaisu osoittautui myös kehitystyön kannalta hyväksi. Ratkaisua käytettäessä kehittäjän ei tarvitse itse huolehtia esimerkiksi animaatiotilojen toistamisesta.

Toista snapshot-ratkaisua ei ehditty kehittää pelikelpoiseen kuntoon. Ratkaisujen muistinkäyttöä pystyttiin silti mittaamaan tallentamalla snapshot-ratkaisun käyttämät tiedot ja vertaamalla muistinkäyttöä toimivaan deterministiseen ratkaisuun. Snapshot-ratkaisu osoittautui myös mittaustulosten perusteella käyttökelvottomaksi. Kehitetty ominaisuus on välttämätön pelin toiminnan kannalta.

Asiasanat: Unity, peli, ohjelmointi, peliohjelmointi, tallennus, uusinta

South-Eastern Finland University of Applied Sciences

Degree title	Bachelor of Engineering
Author (authors)	Roni Rissanen
Thesis title	Replay functionality in Unity
Commissioned by	Kaakkois-Suomen ammattikorkeakoulu, Xamk, Gamelab
Time	2024
Pages	36 pages
Supervisor	Pekka Vilpponen

ABSTRACT

The purpose of the thesis was to develop a replay feature in the Unity game engine. The feature was intended for a puzzle game where the player's movements are replayed after they occur. The player records copies of themselves that help solving puzzles. The findings of the thesis also apply to more traditional replay features, such as those used in e-sports games for viewing matches. The project provides the client and the industry in general with information about game engines, physics engines, and replay features. The feature was implemented in the Unity game engine, but the information from the thesis can also be applied in other environments.

Previously made solutions were analysed in the thesis. Two viable solutions were found during the research - a snapshot solution and a deterministic solution. Based on calculations made before the start of development, the deterministic solution was much more efficient in terms of memory usage, which is why it was chosen for as the implementation method. Essential features of the game engine for the deterministic solution are discussed in the thesis. A major problem with the deterministic solution is the requirement for deterministic physics, which the physics engine that comes with Unity's EC-framework cannot handle. The thesis discusses the reasons for and solutions to this problem.

As a result of the thesis, the replay feature was completed. The feature was developed following the deterministic solution. The code for the feature is presented with screenshots. The deterministic solution also proved to be good for development. When using the solution, the developer does not have to worry about replaying animation states, for example.

The other snapshot solution could not be developed to a playable state in time. However, the memory usage of the solutions could still be measured by recording the data used by the snapshot solution and comparing the memory usage to the functioning deterministic solution. Based on the measurement results, the snapshot solution also proved to be unusable. The developed feature is essential for the game to function.

Keywords: Unity, game, programming, physics, record, replay

SISÄLLYS

1 JOHDANTO

Tämän opinnäytetyön tarkoituksena on analysoida erilaisia menetelmiä uusintatoiminnallisuuden (replay) toteuttamiseen Unity-pelimoottoriympäristöön. Työssä käydään läpi yleisesti edellytyksiä uusinnalle, jotka soveltuvat myös muihin ympäristöihin.

Opinnäytetyön toimeksiantajana toimii Kaakkois-Suomen ammattikorkeakoulussa toimiva Xamk Gamelab. Gamelab on peliohjelmoinnin opiskelijoille tarkoitettu oppimisympäristö, joka sijaitsee Xamkin Kotkan kampuksella.

Uusintatoiminolla tarkoitetaan pelin tapahtumien tallentamista muistiin, josta ne voidaan toistaa uudelleen. Kyseessä ei ole video, vaan aikaisemmin tapahtuneet asiat tapahtuvat ns. oikeasti uudestaan moottorin sisällä.

Työn tavoitteena on antaa toimeksiantajalle tietoa pelifysiikasta ja pelimoottoreiden sisäisistä uusintatoiminnoista. Uusintatoiminto tulee tasoloikkapeliin nimeltä Chronological, jonka jatkokehitys on työn toissijainen tavoite.

Teoriaosiossa käydään läpi peli- ja fysiikkamoottoreiden ominaisuuksia ja historiaa sekä fysiikkaa, jota uusinnassa toistetaan. Kehitystyöhön valitusta Unity-pelimoottorista esitellään ja analysoidaan uusinnan kannalta oleellisia komponentteja ja ominaisuuksia. Opinnäytetyö on rajattu käsittelemään ainoastaan uusintatoiminnallisuutta. Muita Chronological-pelin kehityksen kannalta olennaisia Unity-ominaisuuksia ei esitellä.

2 TUTKIMUKSEN LÄHTÖKOHDAT

Opinnäytetyön tavoitteena on toteuttaa uusintamekaniikka Chronological-peliin. Chronological on kehitetty Unity-pelimoottorin työkaluilla. Opinnäytetyö tuottaa toimeksiantajalle ja alalle yleisesti tietoa peli- ja fysiikkamoottoreista, Unity-pelimoottorista sekä uusintatoiminnoista. Tutkimusongelma on uusintatoiminnon toteuttaminen Unity-pelimoottorissa. Kehitystyötä ohjaavat aiemmin asetetut tutkimuskysymykset, joihin työn on tarkoitus vastata.

Opinnäytetyön tutkimusongelma on jaettu kolmeen tutkimuskysymykseen:

1. Minkälaisia uusintatoiminnallisuusratkaisuja on kehitetty aiemmin?
2. Mikä aiemmista ratkaisuista on tarkoituksenmukainen?
3. Miten ratkaisu implementoidaan valittuun ympäristöön?

Tutkimus on rajattu koskemaan peliin tulevaa mekaniikkaa. Peli toimii mittausten testiympäristönä.

Työssä kehitetään peliprojektiin uusi ominaisuus. Tutkimusotteeksi on valittu tämän perusteella kehittämistutkimus. Kehittämistutkimus tähtää tuotteen, menetelmän tai organisaation muutokseen (Kananen 2017, 17). Kehittämistutkimuksella on kolme ominaispiirrettä: 1) kehittäminen syntyy muutoksen tarpeesta, 2) kehittäminen johtaa käytettävään tuotokseen ja 3) kehittäminen tuottaa tietoa. (Pernaa 2013, 11). Tutkimusmenetelmä perustuu kirjallisuuden analysointiin ja kehitystyön aikana tehtyihin havaintoihin. Työn alussa kerätään tietoa aikaisemmin tehdyistä ratkaisuista. Erilaisten ratkaisujen suorituskykyä voidaan laskea jo ennen kehitystyön aloittamista. Ratkaisujen suorituskykyä voi mitata muistinkäyttöä mittaamalla Unity-pelimoottorin työkalujen avulla. Aikaisemmat ratkaisut eivät välttämättä ole Unity-pelimoottorilla luotuja, mutta perusideoiden pitäisi välittyä moottorista toiseen.

Tarkoituksenmukainen ratkaisu on muistin kannalta tehokas, pelaajan liikettä on pystyttävä tallentamaan useita minuutteja kerralla. Myös pelaajan vuorovaikutus kentän kanssa sekä erilaiset animaatiotilat ja äänet on saatava toistettua, siten kuin ne pelatessa tapahtuivat.

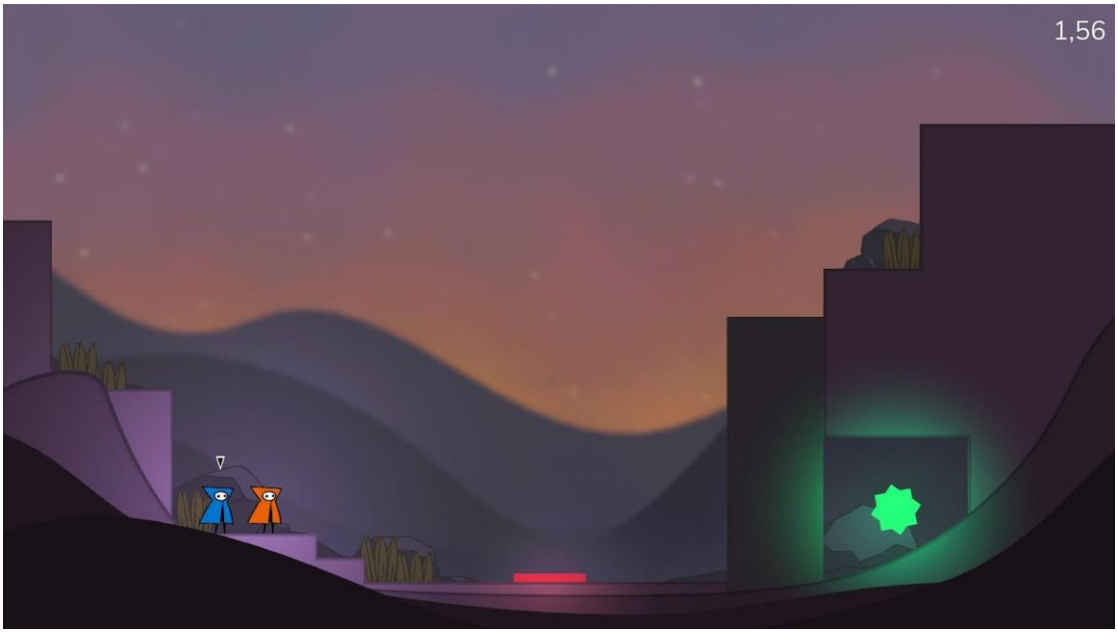
Teknisiä ratkaisuja perustellaan teoriaosuudessa esitettyjen tietojen perusteella. Koodia esitellään ruutukaappausten avulla. Ruutukaappausten koodi on kommentoitu englanniksi, mutta sen merkitys avataan vielä uudestaan suomeksi tekstissä.

Työn luotettavuus tulee olemaan hyvä. Lähteinä käytetään Unity-pelimoottorin dokumentaatiota ja kehittäjien blogikirjoituksia. Muiden lähteiden tietoja voi verrata näihin lähteisiin. Pelifysiikasta ja fysiikasta yleisesti on olemassa paljon hyvää kirjallisuutta. Teoriaosuuden tietojen pohjalta kehitetty ratkaisu testataan, jonka perusteella tietojen paikkansapitävyyttä voi arvioida edelleen.

3 PELIN ESITTELY

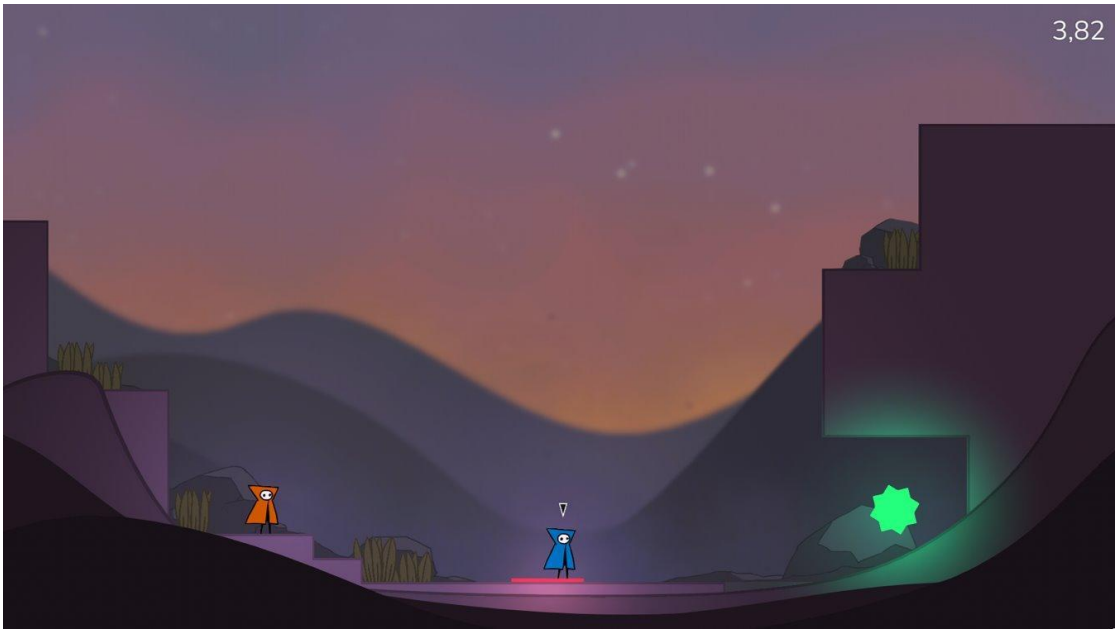
Chronological-pelissä uusinnalla on pelimekaaninen merkitys. Se on olennainen osa pelin toimintaa. Tyypillisesti uusinta on toissijainen ominaisuus, jolla voi tarkastella pelisuorituksia jälkikäteen.

Peli on kaksiulotteinen tasoloikkapeli, jossa pelaajan täytyy saavuttaa vihreä kuula päästäkseen seuraavaan tasoon. Kuvassa 1 harmaa pilari estää kuulan saavuttamisen.



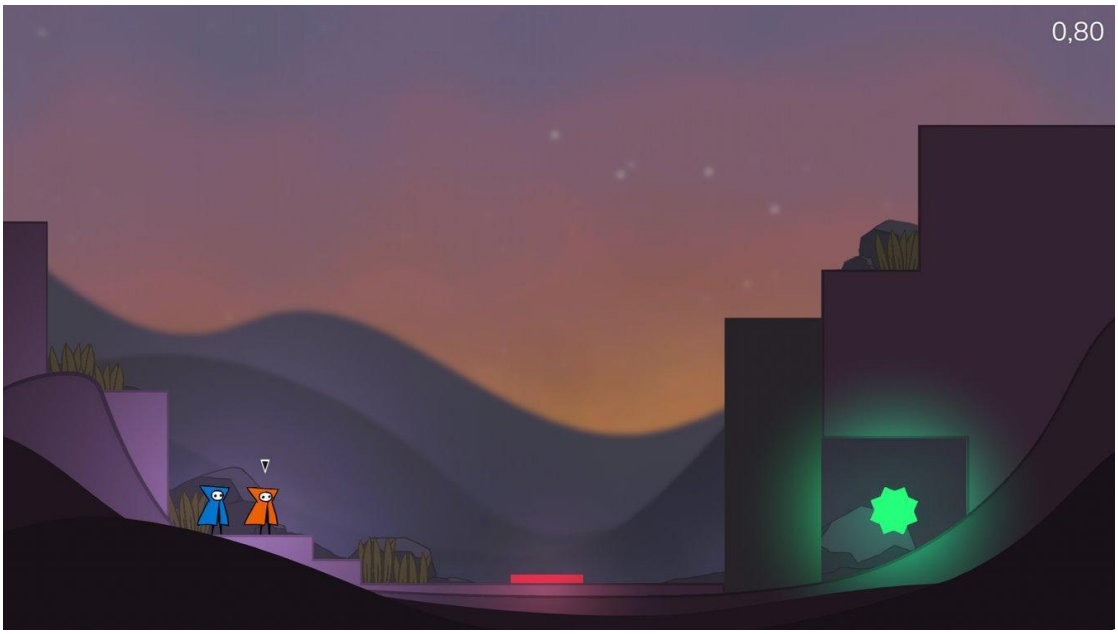
Kuva 1. Tason aloitustilanne: pilari estää pelaajan etenemisen

Esteet on asetettava oikeaan asentoon, kuten kuvassa 2 kentästä löytyvien punaisten nappien avulla, jotta tie kuulalle on avoin. Este palautuu takaisin aloitustilaan, kun napin päällä ei ole mitään.

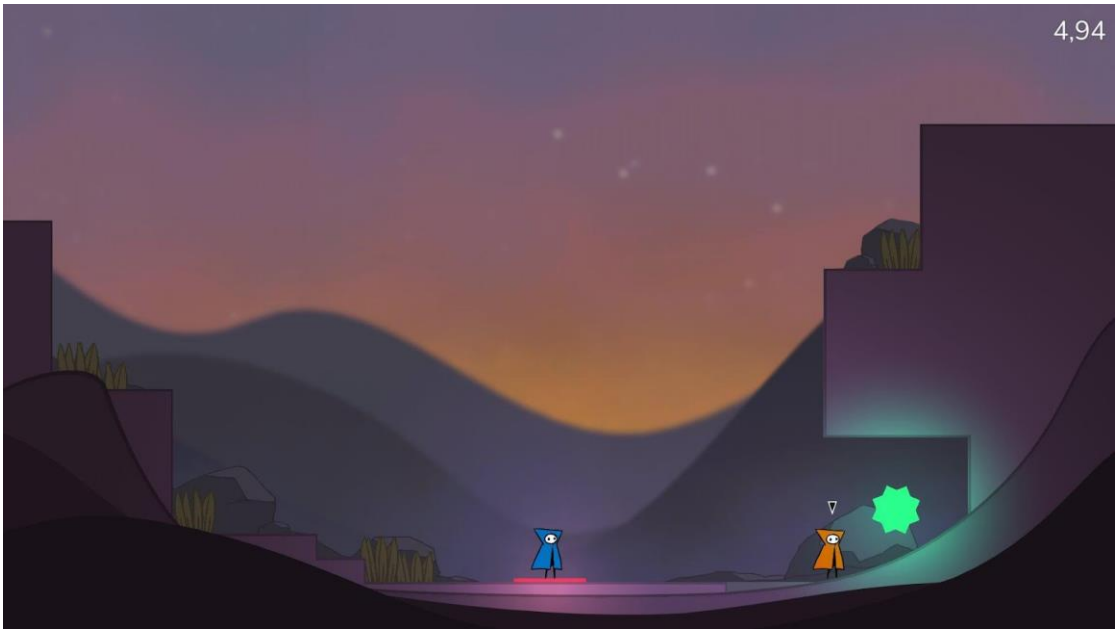


Kuva 2. Pelaaja laskee pilarin painamalla napin alas. Tie kuulan luokse avautuu.

Pelaajalla on kaksi hahmoa käytössä. Kun pelaaja vaihtaa hahmoa, kenttä alkaa alusta. Kuva 3 näyttää kentän aloitustilanteelta. Vaikka taso nollautuu, edellinen hahmo muistaa ja toistaa liikkeet, joita pelaaja edellisellä suoritusyrityksellä käytti. Pelaaja voi siis kentän läpäistäkseen ohjata ensimmäisen hahmon napin päälle ja, valita toisen hahmon, jolloin pelaajan ensimmäiseksi liikuttama hahmo muistaa liikkeensä ja kävelee napin päälle. Tällöin tie on pelaajalle avoin, kuten kuvassa 4.



Kuva 3. Kenttä palaa lähtötilanteeseen, kun pelaaja vaihtaa hahmoa.



Kuva 4. Ensimmäiseksi valittu sininen hahmo muistaa pelaajan aikaisemmat liikkeet ja avaa tien vihreän kuulan luokse. Pelaajan tällä hetkellä ohjaama oranssi hahmo pääsee maaliin.

Peli-idea perustuu samankaltaiseen mekaniikkaan Ratchet & Clank: A Crack In Time -pelissä. Pelissä Clank-robotin tasoissa on hyvin samankaltainen pulmamekaniikka. Pelin kymmenen tunnin kestosta kuitenkin alle tunti on näitä pulmia. (Sony Computer Entertainment 2009). Tavoitteena on ollut luoda peli, joka sisältää ainoastaan tämänkaltaisia pulmia. Samalla ulottuvuuksien määrää on vähennetty kolmesta kahteen, mikä helpottaa pelin kehittämistä ja pelaamista.

4 UUSINTATOIMINNALLISUUDEN TOTEUTUKSIA

Tutkiessani aikaisempia uusintaominaisuuksia löysin kaksi erilaista varteenotettavaa lähestymistapaa. Tapaa, jossa pelin tilasta (esimerkiksi kappaleiden sijainnit) tallennetaan tiedot, kutsutaan snapshot-toteutukseksi. Tapaa, jossa käyttäjän syöte tallennetaan, kutsutaan deterministiseksi toteutukseksi. Deterministinen toteutus on täysin riippuvainen fysiikkamoottorin tasaisesta toiminnasta. Molemmat tavat tallentavat tietoja tietyin väliajoin. Tämä tarve on helppo toteuttaa fysiikkamoottorin tasaisen virkistystaajuuden avulla. (Wagner 2004.)

4.1 Snapshot-ratkaisu

Snapshot-ratkaisussa ajatus on ottaa pelin tilasta ns. snapshot, tilannekuva, eli kaikki olennaiset muuttujat tallennetaan tietyin väliajoin. Tilannekuvien sarja voidaan myöhemmin toistaa kuva kerrallaan. (Wagner 2004.)

Toteutuksen ongelmana on sen monimutkaisuus. Täytyy muistaa tallentaa kaikki olennaiset pelitilanteen muuttujat. Esimerkiksi erilaisten animaatiotilojen tallentamisesta ja toistamisesta tulee monimutkaista. (Wagner 2004.)

Etuna on tarkkuus. Fysiikkamoottorin tehdessä virheitä, virheet jäävät talteen ja ne voidaan toistaa aivan kuin ne tapahtuivat ensimmäisellä kerralla. Deterministisessä ratkaisussa on vain pakko luottaa, että moottorin virheet ovat niin pieniä, että käyttäjä ei huomaa niitä. (Wagner 2004.)

4.2 Deterministinen ratkaisu

Deterministinen ratkaisu perustuu ainoastaan käyttäjän syötteen (input) tallentamiseen. Jos fysiikkamoottori on deterministinen, saadaan sama lopputulos pelaajan syötteellä ja kopiolla pelaajan syötteestä. Mikäli moottori ei toimi tasaisesti, uusinta helposti desynkronisoituu. (Wagner, 2004.) Esimerkiksi pelaajaan vaikuttava kitkavoima lasketaan 21 kertaa ennen hyppyä pelattaessa, mutta 22 kertaa toistettaessa, jolloin pelaajan hyppy jää uusinnassa liian lyhyeksi.

Deterministisen ratkaisun etuna on sen yksinkertaisuus. Tallennettavia muuttujia on vain sama määrä, kuin pelaajalla on käytössä painikkeita, joilla hän voi vaikuttaa pelin tilaan. (Wagner 2004.)

Esimerkiksi animaatiotilat toistuvat lähes itsestään. Tallennettaessa tallentava funktio kutsuu pelaajahahmoa liikuttavaa funktiota, jota kautta hahmoa myös animoidaan. Toistettaessa samaa funktiota voidaan kutsua tallennetuilla arvoilla, jolloin animaatiotilat toistuvat uudestaan liikkeen mukana niitä erikseen tallentamatta. (Wagner 2004.)

4.3 Muistin käytön laskentaa ja muita analyysejä

Voimme perustella ratkaisun valintaa myös suorituskyvyn perusteella. Snapshot-ratkaisussa jokaisen kappaleen jokainen pelimekaanisesti merkityksellinen muuttuja tallennetaan aina, kun moottori virkistyy. Käytännössä Transform-komponentin position-vektori ja rotation-kvaternion (englanniksi quaternion) tallennetaan muistiin. Komponentit koostuvat kolmesta (Vector3) ja neljästä (Quaternion) float-muuttujasta. Parhaassa tapauksessa muistia kuluu 28 bittiä per esine per ruutu ainoastaan kappaleen paikan ja rotaation seuraamiseen.

Deterministisessä toteutuksessa talteen otetaan vain pelaajan syöte, mikä on monimutkaisessakin pelissä tyypillisesti alle 20 painiketta, joita voidaan kuvata yksinkertaisilla boolean-muuttujilla. Tämän lisäksi tarvitaan float-muuttujia mahdollisia akseleita tai hiiren sijaintia kuvaamaan. 20 boolean-muuttujaa ja kaksi float-muuttujaa vievät myös 28 bittiä muistia jokaista ruutua kohti. Tämä pitää vielä kertoa jokaista tallennettavana olevaa pelaajahahmoa kohti.

Hyvin yksinkertaisessa ja pienessä kentässä snapshot-ratkaisu voi olla käytännöllisempi. Jos huomioidaan, että Chronological-pelin kentissä voi olla yli kymmenen liikkuvaa esinettä, deterministisestä ratkaisusta tulee huomattavasti nopeampi. Pelissä on myös vain murto-osa 20 esimerkissä tallennetusta boolean-muuttujasta. Deterministinen ratkaisu on valittu toteutettavaksi sen yksinkertaisuuden ja suorituskyvyn perusteella.

5 PELIMOOTTOREIDEN ESITTELY

Pelimoottorit ovat ohjelmistokehitystyökaluja, jotka vähentävät kehitystyön kustannuksia ja monimutkaisuutta sekä säästävät aikaa. Pelimoottoreihin on rakennettu valmiiksi ominaisuuksia, joita useimmat pelit tarvitsevat. Ajatuksena on rakentaa sisältöä muiden kehittämän teknologian päälle, jolloin kehittäjän ei tarvitse aloittaa kehitystyötä aivan alusta. (Simpson 2002.)

Aiemmin pelistudioilla oli hyvin tiettyihin tarkoituksiin kehitetyt talonsisäiset pelimoottoriohjelmistot. Tasohyppelypelin moottorilla olisi tuolloin ollut käytännössä mahdotonta kehittää autopeliä ilman suuria muutoksia. (Simpson 2002.)

5.1 Unity-pelimoottorin esittely

Unity-pelimoottori tuli markkinoille virallisesti vuonna 2005. Jo vuonna 2006 moottori sai merkittävää huomiota ensimmäisenä iPhone-alustalla toimivana pelimoottorina. Virallisesti Unity alkoi tukemaan iPhonea App Store -applikaatiokaupan julkaisun yhteydessä 2008. Unity sai alkunsa merkittävänä kilpailijana pelimoottorimarkkinoilla mobiilipelipuolella, mutta nykyään Unity-pelit ovat suosittuja kaikilla alustoilla. (Axon 2016.)

Unity on itse julkaisemansa raportin mukaan yksi markkinoiden johtavista pelimoottoreista. 70 prosenttia markkinoiden johtavista mobiilipeleistä käyttää moottoria. Moottorin kehittäjä väittää 50 prosentin kaikista peleistä, kaikilla alustoilla käyttävän Unity-pelimoottoria. Moottorilla voi luoda kaksi- ja kolmiulotteisia pelejä. (Dealessandri 2023.)

Unityn suosio perustuu osittain sen edulliseen hintaan. Moottori on kaikkien saatavilla Personal-lisenssillä 100 000 dollarin liikevaihtoon ja rahoitukseen asti. Unity on luvannut nostaa liikevaihto- ja rahoitusylärajaa 200 000 dollariin. Näin ei ole tapahtunut 13.2.2024 mennessä (Unity Personal s.a.). Moottoria käytetään työssä Personal-lisenssillä.

5.2 Fysiikkamoottori

Fysiikkasimulointi on tietotekniikan haara, joka pyrkii mallintamaan oikean maailman fysiikan toimintaa tietokoneella. Käytännössä tietokone laskee kaavoilla kappaleisiin vaikuttavia voimia.

Fysiikan simulointi peleissä on todella yleistä. Peli-ideat liittyvät usein konkreettiseen maailmaan, jolloin fysiikan jäljentäminen on välttämätöntä. Pelit kuten Pong (Atari 1972) ja Donkey Kong (Nintendo 1981) jäljittelevät alkeellisesti oikean maailman toimintaa. Pongissa pallon lentorata käyttäytyy jokseenkin uskottavasti. Donkey Kongissa on jo hahmoja ja kappaleita, joihin vaikuttaa jonkin näköinen painovoima.

Peleissä käytettäviä fysiikkamalleja usein yksinkertaistetaan. On tärkeää, että pelin fysiikkasimulaatio toimii sulavasti käyttämättä liikaa laskentakapasiteettia. Peleissä fysiikan tarkka simulointi on harvoin tarpeellista. Kaikkein realistisimmissakin peleissä uskottavuus riittää.

Fysiikkamoottori vastaa nimensä mukaisesti pelin fysiikan simuloinnista. Fysiikkaa päivitetään tasaisella virkistystaajuudella. Kehittäjä voi usein itse vaikuttaa taajuuteen. Tyypillinen päivitysnopeus on 50 kertaa sekunnissa, joka valittiin myös tähän työhön. Fysiikkamoottorilla on lista kappaleista, joihin simulaatio vaikuttaa. Jokainen fysiikkamoottorin virkistys siirtää simulaation ajanhetkeä eteenpäin, tässä tapauksessa 0,02 sekuntia. Tällöin kaikki kappaleet liikahtavat simulaatiossa askeleen eteenpäin. (Millington 2007, 40–54).

On muistettava, että fysiikkamoottori vastaa vain fysiikkalaskuista. Kuvan piirtäminen eli renderöinti tapahtuu tästä prosessista erillään, usein jopa eri virkistystaajuudella. Joissakin peleissä fysiikkamoottorin virkistystaajuus on sidottu renderöinnin virkistystaajuuteen. Jos renderöinnin virkistystaajuus on tässä tilanteessa eri, kuin kehittäjä on olettanut, fysiikkamoottori ei toimi enää suunnitellulla tavalla. (Walker 2018.)

Suuria pelejä, jotka toteuttavat fyysiikat näin ovat esimerkiksi Bethesdan Creation Engine -moottoria käyttävät pelit, kuten The Elder Scrolls V: Skyrim ja Fallout: New Vegas. Mainittuihin peleihin on saatavilla käyttäjien tekemiä muokkauksia (modeja), jotka irrottavat fysiikat renderöinnistä. (SSE Display Tweaks 2022.)

Yleisin syy virkistystaajuuksien synkronoimiseen on koodin yksinkertaisuus. Kehittäjien ei tarvitse käyttää eri muuttujia eri tarkoituksissa, vaan kaikki virkistyskierrokseen sidottu koodi toimii samoilla arvoilla. Koodia on helpompi kirjoittaa ja kierrättää, kun fysiikan voidaan olettaa toimivan tällä tavalla. Toinen yleinen syy on ajan deltan, eli virkistyskierroksien välisen ajan aiheuttamat ongelmat. Kun ajan delta kasvaa todella suureksi tai todella pieneksi, fysiikka ei välttämättä toimi odotetulla tavalla. Vaihtuvalla ajan deltalla laskeminen voi myös aiheuttaa muita epätarkkuuksia fysiikkaan tietyillä raja-arvoilla. Liukulukujen pyöristysten aiheuttamia ongelmia käydään yksityiskohtaisemmin determinismiä käsittelevässä alakappaleessa.

5.2.1 Fysiikka

Kappaleet simuloidaan usein jäykkinä kappaleina, eli kappaleeseen kohdistuvat voimat eivät muuta sen muotoa. Tämä yksinkertaistaa laskentaa huomattavasti. Myös pehmeän kappaleen tai nesteiden dynamiikkaa on mahdollista

simuloida, mutta videopeleissä ne ovat laskennan hitauden takia usein jääneet taka-alalle. (Millington 2007, 2.)

Yksinkertaisimmillaan jäykän kappaleen dynamiikkaa lasketaan Newtonin kolmella lailla:

1. Jatkuvuuden laki: Kappale jatkaa tasaista suoraviivaista liikettä vakionopeudella tai pysyy levossa, jos siihen ei vaikuta ulkoisia voimia.
2. Dynamiikan peruslaki: Kappaleeseen vaikuttava voima aiheuttaa kappaleen liikkeen muutoksen.
3. Voiman ja vastavoiman laki: Jos kappale A vaikuttaa kappaleeseen B voimalla, niin kappale B vaikuttaa kappaleeseen A yhtä suurella, mutta vastakkaissuuntaisella voimalla.

Newtonin toinen laki kertoo voiman ja kiihtyvyyden suhteen. Sama voima kohdistettuna kahteen eri massaiseen kappaleeseen saa aikaan eri kiihtyvyydet:

$$f = ma$$

Fysiikkamoottorilla on tyypillisesti kappaleen massa ja vaikuttava voima tiedossa, kun halutaan löytää, millä kiihtyvyydellä kappaletta liikutetaan:

$$a = \frac{1}{m} f$$

On myös yleistä, että tietyt kappaleet ovat vain osittain mukana simulaatiossa. Kappaleen maahan aiheuttamaa voimaa ei ole tarkoituksenmukaista laskea, jos ei haluta, että maa voi tämän voiman vaikutuksesta liikkua. (Millington 2007, 46.)

5.2.2 Determinismi

Mikäli simulaation lopputulos on tietyillä arvoilla aina sama, voidaan simulaatiota kutsua deterministiseksi. Fysiikan deterministisyydestä olisi merkittävää hyötyä nettipeleissä, sekä uusintatoiminnalisuudessa, jota työssä kehitetään. Sen implementointi fysiikkamoottoriin vaatii kuitenkin kehittäjiltä ylimääräistä työtä. Unity-pelimoottorin fysiikkamoottori ei ole deterministinen. Työssä käydään myöhemmin läpi asetuksia, joiden avulla fysiikkamoottorin saa toimimaan mahdollisimman ennalta-arvattavasti. (Dawson 2013.)

Fysiikkalaskuista vastaava prosessori ei ole täydellisen tarkka. Desimaalilu-vuilla laskettaessa tulee välttämättä tarve äärettömän tarkalle luvulle, jonka esittämiseen liukulukuna on varattu rajallinen määrä bittejä. Luku on teknisen rajoitteen vuoksi pakko pyöristää. (Goldberg 1991.)

IEEE-standardit ovat pyrkineet laitteiden väliseen liukulukujen deterministisyy-teen. Ne eivät kuitenkaan takaa, että sama ohjelma antaa laskulle täysin identtiset tulokset kahdella eri tietokoneella. C++ liukulukustandardit eivät vaadi IEEE-liukulukumatematiikkaa. (Dawson 2013.)

Newtonin ensimmäisen lain mukaan kappale jatkaa tasaista suoraviivaista lii-kettä vakionopeudella tai pysyy levossa, jos siihen ei vaikuta ulkoisia voimia. Liukulukujen aiheuttaman epätarkkuuden vuoksi kappale saattaa kuitenkin kiihtyä, vaikka siihen ei vaikuttaisi mitään voimia. Useita kertoja sekunnissa fy-siikkamoottorin kierroksia laskettaessa virhe saattaa kasvaa helposti merkittä-väksi. (Millington 2007, 44.)

Ongelma on tyypillisesti ratkaistu kohdistamalla karkea arvio ilmanvastuksesta simulaatiossa mukana oleviin kappaleisiin. Tätä voimaa ei tyypillisesti kutsuta kitkaksi, jotta se ei mene sekaisin, mikäli fysiikkamoottoriin implementoidaan myöhemmin yksityiskohtaisempia ilmanvastusta simuloivia voimia. Voimaa kutsutaan englanninkielisissä teksteissä nimellä damping. (Millington 2007, 45.)

6 UUSINTATOIMINNALLISUUS UNITY-PELIMOOTTORISSA

Luvun 4 laskutoimitusten perusteella toteutustavaksi on valittu deterministinen ratkaisu. Hitaampaa keskeneräistä snapshot-ratkaisua käytettiin vertailukoh-teena. Uusintatoiminnallisuuden osat esitellään siinä järjestyksessä, kun ne paljastuvat käyttäjälle.

Toiminnallisia kappaleita kootaan Unityn EC-frameworkissa komponenteista. Moottorin mukana tulee monia yleisesti hyödyllisiä komponentteja. Unity-peli-moottorin fysiikkamoottoria käytetään asettamalla kappaleelle RigidBody- ja Collider-komponentti, jolloin kappale on mukana fysiikkasimulaatiossa.

Uusintatoiminnallisuudelle ei ole valmiita komponentteja, joten kehittäjän on ohjelmoitava tarkoituksenmukaisia komponentteja itse. Teknisesti olisi mah-dollista luoda yksi valtava komponentti, joka toteuttaa toiminnallisuuden yksin.

Toiminnallisuus on hyviä ohjelmointikäytäntöjä noudattaen koottu useasta pienemmästä komponentista. (French 2024.)

6.1 Koodin esittely

Snapshot-ratkaisu koostuu neljästä suuresta komponentista sekä useista pienemmistä apukomponenteista. Apukomponentteja ei erikseen käsitellä tässä työssä. Neljä suurta komponenttia ovat PlayerInstanceController, ReplayActor, PlayerRecorder ja InputRecorder.

PlayerInstanceController-komponentti pitää kirjaa tasossa olevista pelaajahahmoista. Komponentti voi antaa hahmon pelaajan ohjattavaksi tai alkaa toistamaan hahmolle tallennettua liikettä. Tämä tapahtuu aina kutsumalla ReplayActor-komponentin funktioita. Pelaaja valitsee ohjattavan hahmon numeroilla 1-4. PlayerInstaceController-komponentteja on yhdessä tasossa vain yksi kappale. Kolme muuta tärkeää komponenttia ovat kiinni jokaisessa pelaajahahmossa.

ReplayActor on komponenteista suurin. Sen kautta ohjataan kaikkia muita pelaajahahmossa kiinni olevia uusintakomponentteja. Komponentin Update- ja FixedUpdate-funktioissa kutsutaan tallentamiseen ja toistamiseen olennaisia PlayerRecorder- ja InputRecorder-komponenttien funktioita. ReplayActor komponentilla on useita tiloja. Tiloja vastaavia funktioita kutsutaan Update- ja FixedUpdate-funktioissa.

PlayerRecorder-komponentissa päivitetään pelaajan input-arvoja tallennuksen ollessa käynnissä. Arvot palautetaan GetInputStruct-funktion kautta, kun joku muu komponentti tarvitsee niitä uusintaa toistettaessa.

InputRecorder pitää kirjaa jo tapahtuneista inputeista. Ne tallennetaan yksinkertaiseen tietorakenteeseen, josta ne saa noudettua, kun niitä tarvitaan liikkeen toistamiseen.

Yksinkertaisimmillaan snapshot-ratkaisun tietorakenne on lista, jonka kukin jäsen sisältää tiedon pelaajan inputeista. Rakenteesta saa paremman vaihtamalla listan dictionary-kokoelmaan, jolloin nauhoitusta pystyy kelaamaan dictionaryn key-arvolla. Chronological-pelin tapauksessa nauhoitus halutaan aloittaa aina alusta, kun pelaaja vaihtaa hahmoa, joten tämä ei ole tarpeellista. Toisto aloitetaan arvosta nolla.

Kaikki pelaajan käyttämät näppäimet tietyllä ajanhetkellä otetaan talteen structiin. Struct on esitelty kuvassa 5. W-, A-, S-, ja D-näppäimet olisi voinut ottaa talteen myös boolean-muuttujilla, mutta peli kääntyy helpommin pelionjaimelle, jos se on liukulukuna, joten niissä käytetään float-muuttujia. Osalle näppäimistä on luotu myös ylimääräinen muuttuja, jolla seurataan näppäimen ylös päästämistä. Pohjassa pitämistä vaativat näppäimet, kuten hyppy, toimivat näin paremmin. Nämä ylimääräiset boolean-muuttujat olisi voinut käsitellä myös character controller -skriptissä, jos muistia on tarpeellista optimoida edelleen.

```
1 reference
public struct InputStruct
{
    //WASD axis covered with these.
    1 reference
    public float horizontalInput;
    1 reference
    public float verticalInput;


    //All buttons the player is going to use.
    1 reference
    public bool space;
    1 reference
    public bool spaceUp;
    1 reference
    public bool e;
    1 reference
    public bool eUp;


    0 references
    public InputStruct(float _horizontalInput, float _vert
    {
        horizontalInput = _horizontalInput;
        verticalInput = _verticalInput;
        space = _space;
        spaceUp = _spaceUp;
        e = _e;
        eUp = _eUp;
    }
}
```

Kuva 5. InputStruct sisältää kaikki inputit, joita pelaja käyttää pelissä.

Pelaajan input-arvoja tallennettaessa jokaiselle fysiikkamoottorin päivitys kierroksella luodaan InputStruct, jossa on pelaajan input-tietoja vastaavat tiedot tallessa. Struct lisätään SortedDictionary-kokoelmaan käyttäen ajanhetkeä key-arvona, kuten kuvassa 6.

```
5 references
private SortedDictionary<float, InputStruct> InputRecords;

2 references
public void CreateRecord(float time, InputStruct input)
{
    InputRecords.Add(time, input);
}
```

Kuva 6. InputRecords-luokan SortedDictionary-kokoelma ja CreateRecord-funktio

Kuvan 7 PlayerRecorder-luokassa päivitetään koko ajan muuttujien arvoja, jotka kuvaavat pelaajan inputtien tilaa. Toisen komponentin tarvitessa arvoja ne palautetaan InputStruct-muodossa GiveInputs-funktiota vastaavalla GetInputs-funktiolla.

```
3 references | 3 references
private float horizontalValue, verticalValue;
3 references | 3 references | 3 references | 3 references
private bool space, spaceUp, e, eUp;

1 reference
public void GiveBooleanInputs()
{
    if (Input.GetKeyDown("space"))
    {
        space = true;
    }

    if (Input.GetKeyUp("space"))
    {
        spaceUp = true;
    }

    if (Input.GetKey("e"))
    {
        e = true;
    }

    if (Input.GetKeyUp("e"))
    {
        eUp = true;
    }
}

1 reference
public void GiveAxisInputs()
{
    horizontalValue = Input.GetAxis("Horizontal");
    verticalValue = Input.GetAxis("Vertical");
}
```

Kuva 7. PlayerRecorder-luokan inputteja päivittävät funktiot

Kuvan 8 Record-funktio suoritetaan tallennuksen aikana FixedUpdate-funktiossa. Funktiossa käsketään PlayerRecorder-komponenttia päivittämään akseleiden input arvot. Boolean-arvot päivitetään Update-metodissa FixedUpdate-metodin sijaan pelituntuman vuoksi. Pelitestauksen perusteella GetKeyDown- ja GetKeyUp-metodit toimivat huonosti FixedUpdatessa. FixedUpdaten virkistystaajuus (50 kertaa sekunnissa) on käyttötarkoitukseen liian hidas.

PlayerRecorder-komponentissa muodostetaan InputStruct senhetkisillä input-arvoilla. InputStruct-arvot tallennetaan PlayerRecorder-luokassa sijaitsevaan InputRecords-nimiseen SortedDictionary-kokoelmaan. Tavallisen Dictionary-kokoelman pitäisi toimia SortedDictionary-kokoelman sijasta. SortedDictionary käyttää enemmän muistia järjestyksen ylläpitoon. Tämä on mahdollisuus optimoida muistinkäyttöä. (SortedDictionary<TKey,TValue> Class s.a.)

```
1 reference
private void Record()
{
    // timer
    recordTimer += recordTimer + Time.deltaTime;
    // update timer UI element
    timerInstance.UpdateTimer(recordTimer);

    // axis inputs
    playerRecorder.GiveAxisInputs();

    // get input struct from recorder.
    // boolean inputs are given in update
    // so they're never missed in between physics update frames
    InputStruct input = playerRecorder.GetInputStruct();

    // save inputs with the timer as key
    inputRecorder.CreateRecord(recordTimer, input);

    // actually moves the character
    GiveInputs(input);

    // clear inputs for the next update cycle.
    // ensures that if the record ends there are no inputs left
    // to play on loop in the struct
    playerRecorder.ResetInput();
}
```

Kuva 8. Record-funktio

Tallennetuilla input-tiedoilla voidaan liikuttaa pelaajahahmoa Record-funktion lopussa kutsumalla CharacterController-komponenttia.

Toisto toimii hyvin samalla tavalla kuin tallennus. Toistettaessa InputRecord-dictionarysta luetaan dataa kirjoittamisen sijaan. Data menee GiveInputs-funktioon liikuttamaan hahmoa kuten nauhoitettaessa.

```
1 reference
private void Playback()
{
    // timer
    playbackTimer += Time.deltaTime;
    timerInstance.UpdateTimer(playbackTimer);

    // make sure the key exists
    if (inputRecorder.FindKey(playbackTimer))
    {
        GiveInputs(inputRecorder.GetRecord(playbackTimer));
        return;
    }
    else if (!inputRecorder.KeysLeft(playbackTimer))
    {
        // if there are no recorded inputs left in the dictionary,
        // keep the input loop running so the character stops
        // instead of retaining it's horizontal velocity and sliding around
        // also makes sure that the character reacts to animation conditions
        GiveInputs(new InputStruct(0, 0, false, false, false, false));
    }
}
```

Kuva 9. Playback-funktio

Kun toistettavassa kokoelmassa ei ole enää input-tietoja jäljellä, toistoa jatketaan luomalla uusi InputStruct niillä arvoilla, jotka saadaan, kun pelaaja ei koske ohjaimiin. Tämä varmistaa, että hahmo jatkaa animaatiotiloihin reagointia oikein ja kohdistaa muihin kentän kappaleisiin samat voimat kuin pelaajan ohjaama hahmo.

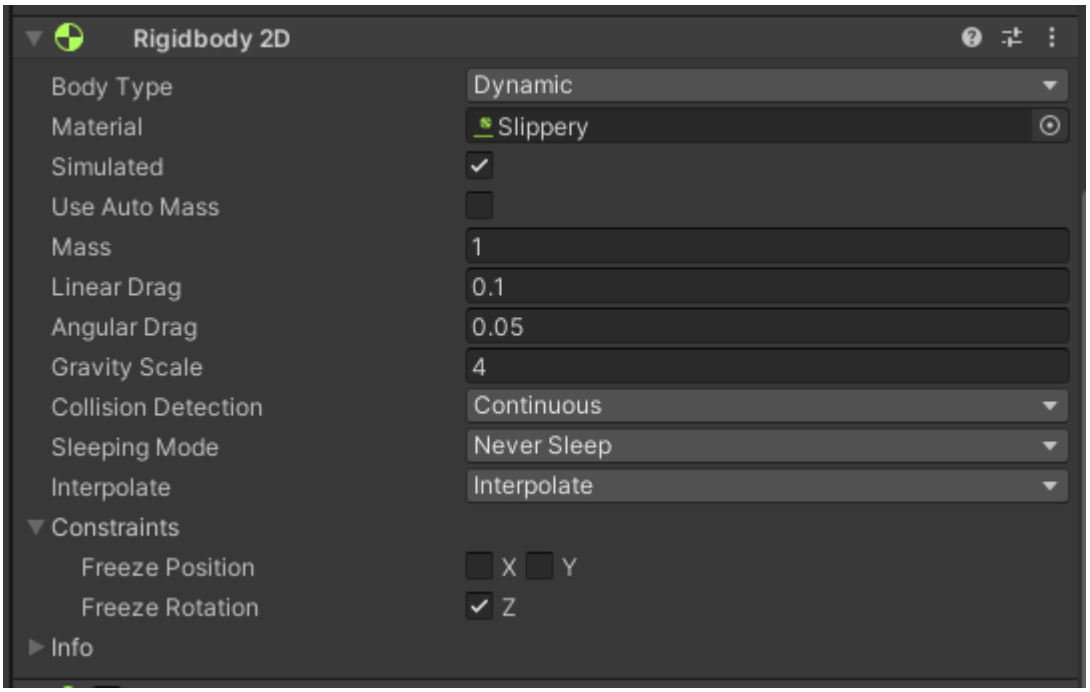
6.2 Fysiikkamoottorin deterministisyys

Pelitestauksen perusteella Unityn fysiikkamoottori ei ole deterministinen, varsinkaan sen vakioasetuksilla. Testien perusteella vakio Rigidbody-komponentin asetukset on asetettava tarkoituksenmukaiseen asentoon, jotta fysiikkakappaleet käyttäytyvät mahdollisimman ennalta-arvattavasti. Kuvassa 10 on käyttämäni Rigidbody2D-asetukset pelaajahahmolle. Samankaltaisia asetuksia kannattaa käyttää liikkuvilla kappaleilla, joiden liikkumista halutaan tallentaa ja toistaa.

Collision Detection -muuttujalla säädetään, milloin kappaleiden välisiä törmäyksiä lasketaan. Se on asetettu Continuous-asentoon, jolloin kappaleen törmäykset lasketaan aina kappaleen liikkuessa. Toinen vaihtoehto on Discrete-asento, jolloin törmäykset lasketaan ainoastaan kappaleen liikuttua sen uudessa sijainnissa. (CollisionDetectionMode2D s.a.)

”Sleeping” on fysiikan laskennassa optimointikeino, jossa kappaleen fysiikan laskenta keskeytetään, kun se on levossa eli kun sen nopeus- ja kiihtyvyysarvo on nolla. Sleeping mode -muuttajalla vaihdetaan, milloin kappaleen annetaan mennä sleeping-tilaan. Sen arvoksi on asetettu ”Never Sleep”, eli kappale on aina mukana fysiikkalaskuissa. Näin kappale on aina varmasti mukana fysiikkasimulaatiossa. (RigidBody2D.sleepMode s.a.)

Ongelmakohtana voisi olla esimerkiksi pelin alku. Hahmo aloittaa sleeping-tilassa, mutta liikkuessaan tila vaihtuu. Kun hahmon liikettä toistetaan uusinnassa, se ei palaa sleeping-tilaan, jolloin fysiikka-asetukset ovat nauhoituksen ja toiston lähtötilanteissa erilaiset.



Kuva 10. Pelaajahahmon Rigidbody2D-komponentin asetukset

Interpolaatio tarkoittaa matematiikkaa tunnettujen arvojen välissä olevien arvojen laskemista. Tässä tapauksessa käytetään useaa fysiikkamoottorin tilaa seuraavaa laskettaessa, jolloin fysiikan laskenta on tarkempaa. Haittana on hitaampi laskenta. Tämän lisäksi interpolaatiota käyttävä kappale on yhden fysiikkamoottorin tilan verran jäljessä, joten kaikki liikkuvat kappaleet on hyvä asettaa käyttämään interpolaatiota. (Apply interpolation to a Rigidbody s.a.)

7 MITTAUSTULOKSET

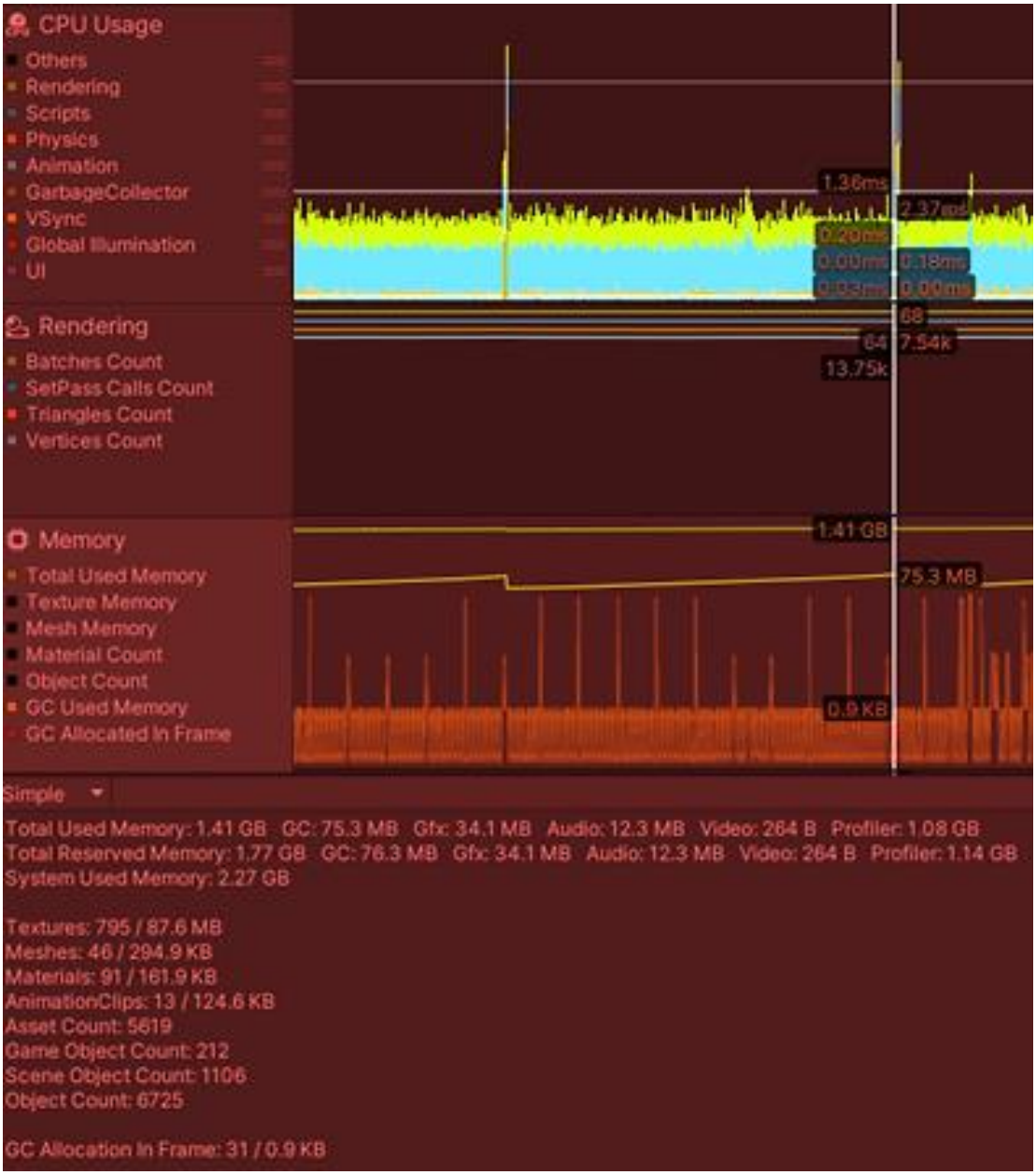
Työn toteutuksen aikataulu ei riittänyt kahden eri ratkaisun toteuttamiseen. Sen sijaan vertailen determinististä- ja snapshot-ratkaisua ainoastaan tietojen

tallennukseen kuluvaa muistia vertailemalla. Snapshot-ratkaisua varten on helppoa tallentaa liikkuvista kappaleista tietoja samalla periaatteella kuin deterministisessä ratkaisussa tallennetaan input-tietoja.

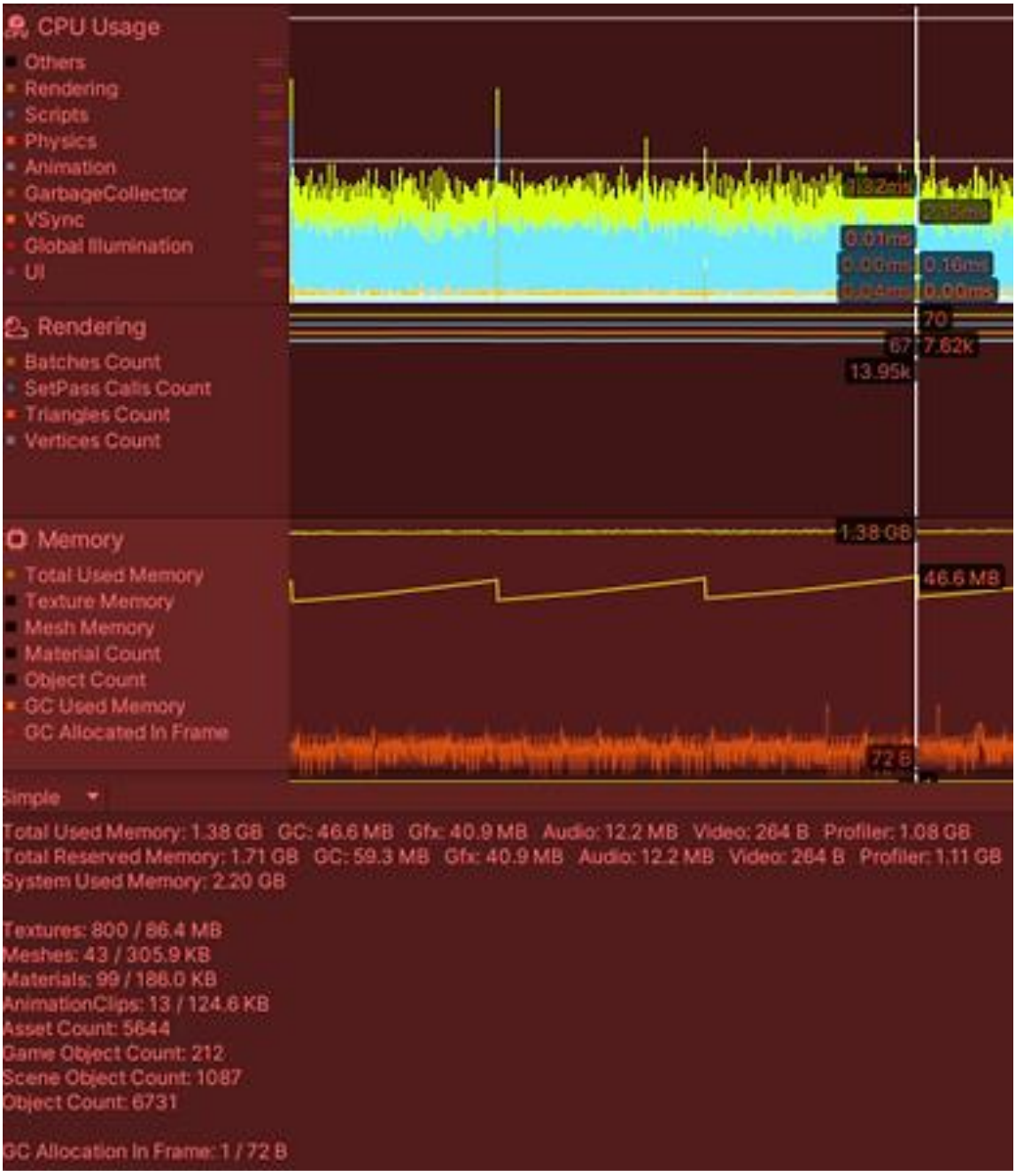
Mittasin eri toteutustapojen suorituskykyä ja muistinkäyttöä Unity-pelimoottoriin sisäänrakennetulla Profiler-työkalulla. Mitattavana oli noin viiden minuutin tallennus ja pidempi tunnin tallennus. Viisi minuuttia on aika, jota ei ylitetä Chronological-pelissä normaaleissa olosuhteissa. Tunti on tyypillinen kesto kilpailullisen (esports) pelin kartalle. Tunti on myös tarpeeksi pitkä aikaväli, jolla ratkaisujen suorituskykyero tulee varmasti esiin.

Deterministinen ratkaisu käytti odotetusti Snapshot-ratkaisua vähemmän muistia molemmilla aikaväleillä. Profiler-työkalu erottelee muistinkäyttöä huonosti, joten vertailun kohteena on kokonaismuistinkäyttö. Kuvassa 11 on snapshot-ratkaisun käyttämä muisti viiden minuutin tallennussession lopussa – 1,41 gigatavua. Kuvan 12 viiden minuutin pituisen tallennussession lopussa deterministinen ratkaisu käytti 1,38 gigatavua muistia. Deterministinen ratkaisu käytti viiden minuutin pituiseen tallennukseen 0,3 gigatavua vähemmän muistia.

Lyhyellä aikavälillä ratkaisujen välillä ei ole olennaista eroa, mutta snapshot-ratkaisu kuormittaa garbage collection -muistinhallintaa huomattavasti enemmän. Garbage collection tai roskien keruu tarkoittaa automaattista muistinhallintaa. Garbage collection vapauttaa automaattisesti tietoja, joihin sovellus ei enää viittaa. (Fundamentals of garbage collection 2023). Snapshot ratkaisu siis luo ja poistaa enemmän tietoja deterministiseen ratkaisuun nähden. Tämä voi Unity-pelimoottorissa aiheuttaa myöhemmin ongelmia, sillä moottorin muistinhallinta kasvattaa sen hallitseman kekomuistin (heap) kokoa roskienkeruun perusteella paljon aggressiivisemmin, kuin pienentää sitä (Understanding the managed heap 2020). Snapshot ratkaisu käytti 75,3 megatavua CG-muistia. Deterministinen ratkaisu käytti 46,6 megatavua CG-muistia, 38 prosenttia vähemmän.

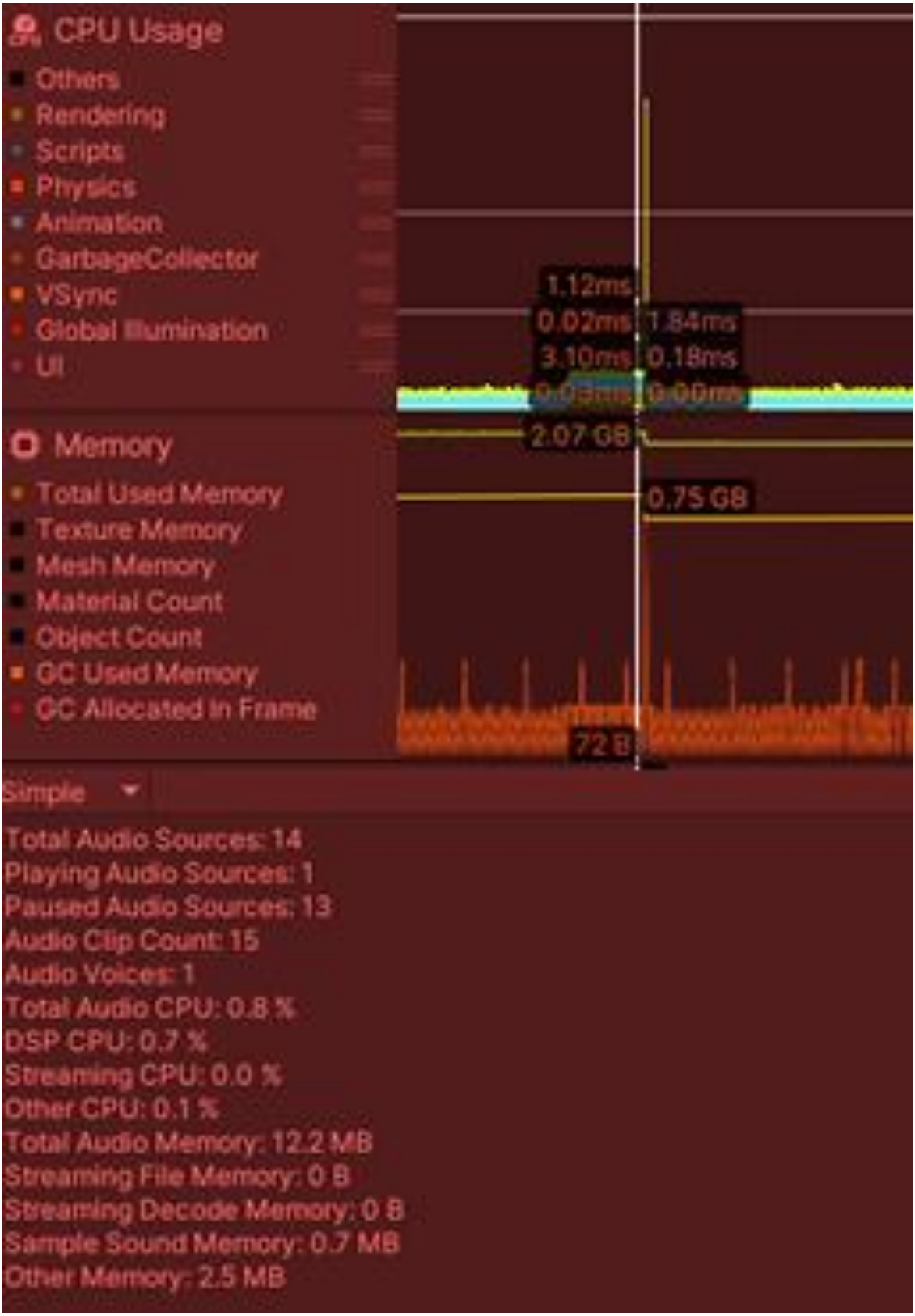


Kuva 11. Kuvassa on näkymä Profiler-työkalusta viiden minuutin snapshot-ratkaisun tallentamisen jälkeen. Kuvaa on tiivistetty kuvankäsittelyohjelmassa.

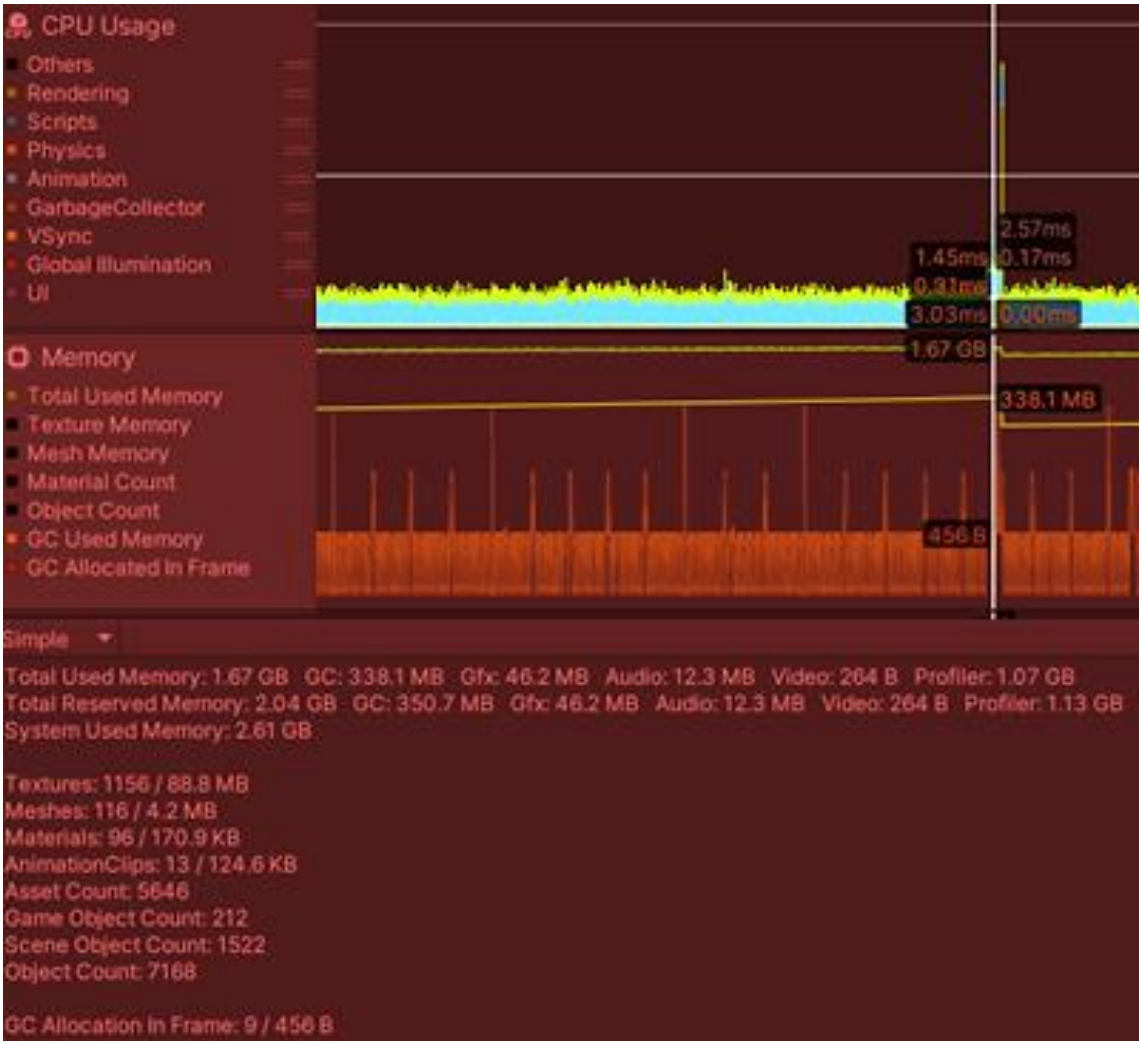


Kuva 12. Kuvassa on näkymä Profiler-työkalusta viiden minuutin deterministisen tallentamisen jälkeen. Kuvaa on tiivistetty kuvankäsittelyohjelmassa.

Kuvan 13 tunnin pituisen tallennuksen lopussa snapshot-ratkaisu käytti 2,07 gigatavua muistia. Kuvassa 14 on deterministisen ratkaisun käyttämä muisti tunnin tallentamisen jälkeen – 1,67 gigatavua. Deterministinen ratkaisu käytti tunnin mittauksessa 0,4 gigatavua vähemmän muistia, 19 % vähemmän kuin snapshot-ratkaisu.

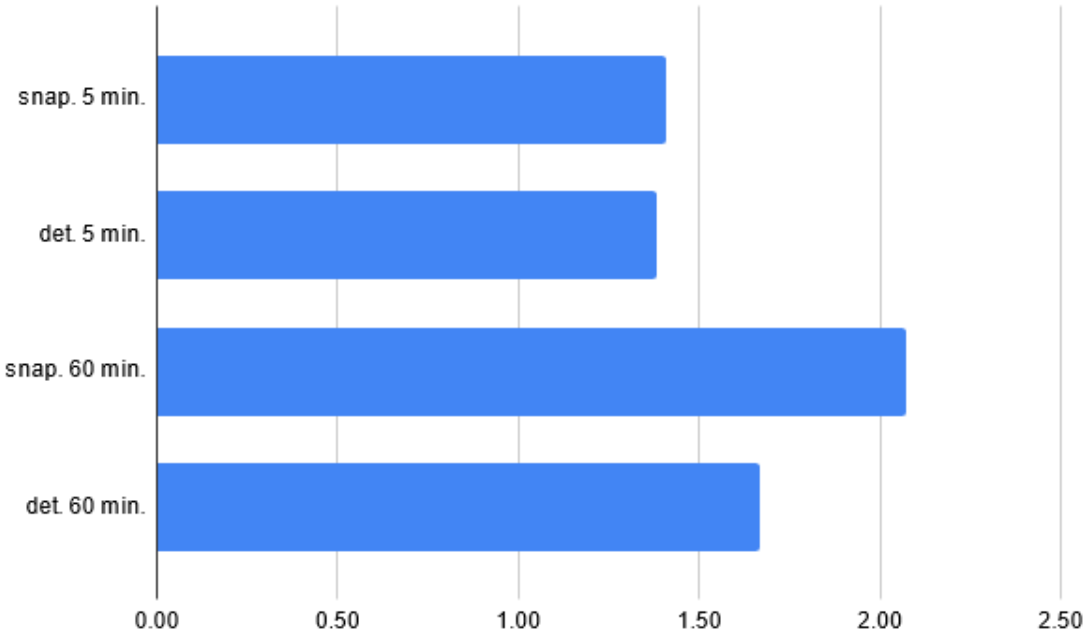


Kuva 13. Kuvassa on näkymä Profiler työkalusta tunnin snapshot-ratkaisun tallentamisen jälkeen. Kuvaa on tiivistetty kuvankäsittelyohjelmassa.



Kuva 14. Kuvassa on näkymä Profiler-työkalusta tunnin determistisen ratkaisun tallentamisen jälkeen. Kuvaa on tiivistetty kuvankäsittelyohjelmassa.

Kuvan 15 taulukossa on kuvien 11, 12, 13 ja 14 “Total Used Memory” -kategorian muistinkäyttö kerätty yhteen taulukkoon. Taulukosta näkee miten paljon muistin käyttö kasvaa pidemmillä aikaväleillä snapshot-ratkaisua käytettäessä.

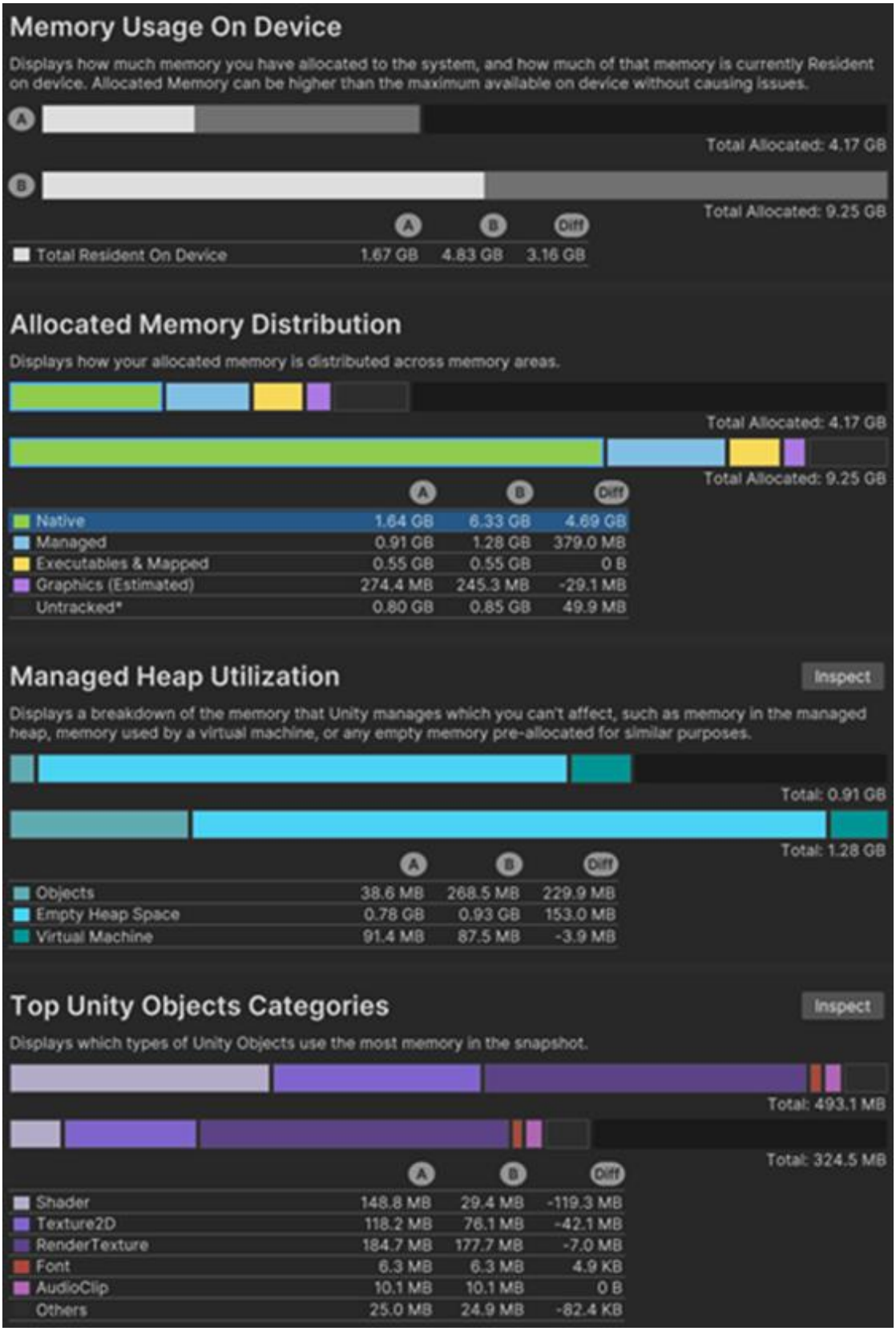


Kuva 15. Snapshot-ratkaisun muistin käyttö kasvaa huomattavasti enemmän kuin deterministisen ratkaisun.

Unity-pelimoottoriin on julkaistu 2022 uusi muistianalyysityökalu – Memory Profiler. Uutta työkalua käytetään varmistamaan yksinkertaisemmalla Profiler-työkalulla saadut tulokset. Uusi työkalu ei sisälly pelimoottorin asennukseen, vaan se asennetaan pelimoottorin Package Manager -ikkunasta erikseen. (Memory Profiler s.a.)

Memory Profiler osoittautui hyväksi työkaluksi. Sen avulla pystyy luomaan tilannekuvan muistin tilasta ja vertaamaan kahta kuvaa keskenään. Kuvassa 16 deterministinen ratkaisu on merkitty kirjaimella A. Snapshot-ratkaisu on merkitty kirjaimella B. Peliä nauhoitettiin tunnin verran, kuten pidemmässä Profiler-työkalulla tehdyssä testissä.

Struct- ja dictionary-tietorakenteet, joilla eri ratkaisutyypit toimivat kuuluvat Memory Profiler -työkalun native memory -muistikategoriaan. Snapshot-ratkaisu käytti 6,33 gigatavua native-kategorian muistia. Deterministinen ratkaisu käytti 1,64 gigatavua muistia. Deterministinen ratkaisu säästi Memory Profiler -työkalun mukaan tunnin mittaisessa tallennuksessa 4,69 gigatavua muistia verrattuna snapshot-ratkaisuun. Snapshot-ratkaisu käyttää paljon enemmän kekomuistia, kuten garbage collection -kategorian perusteella odotettiin.



Kuva 16. Kuvassa on näkymä Unity-pelimoottorin Memory Profiler -työkalusta. Deterministinen ratkaisu on merkitty kirjaimella A ja snapshot-ratkaisu kirjaimella B.

Eri työkalujen antamat arvot muistin käytöstä ovat erilaiset. Eri työkalut ja näkymät kategorisoivat muistinkäyttöä eri tavoin, joten tulokset eivät ole menetelmien välillä verrannolliset. Molempien tulosten mukaan deterministinen ratkaisu on parempi.

8 JOHTOPÄÄTÖKSET

Tutkimusongelma selvitettiin kehittämällä tarvetta vastaava ominaisuus. Ominaisuus toimii luvussa 3 kuvatulla tavalla, mikä on pelin toiminnan kannalta välttämätöntä. Tutkimuksen aikana löytyi kaksi lähestymistapaa uusintatoiminnallisuuden toteuttamiselle, snapshot-ratkaisu ja deterministinen ratkaisu. Deterministisen ratkaisun ennakoitiin olevan parempi, joten se valittiin kehitettäväksi. Kehittämistyön ja pelitestauksen perusteella se on tähän käyttötarkoitukseen hyvä tapa lähestyä uusintaongelmaa. Ratkaisu rakennettiin Unity-pelimoottorin EC-framework komponenteista. Kuten suorituskykyä ennakoivassa luvussa arveltiin, snapshot-ratkaisu on sellaisenaan käyttökelvoton. Mitattu heikko suorituskyky sulkee sen käytön pois tässä tapauksessa. Ratkaisua animaatiotilojen ja partikkeliefektien toistamiseen snapshot-ratkaisulla ei löytynyt. Ne pitäisi kaikki toteuttaa yhtä monimutkaisesti kuin itse liike, mikä kuormittaa muistia edelleen. Tämä monimutkaisuus on myös kehittäjälle ikävä piirre ratkaisussa. Kaikki pelin osat on kehitettävä uudestaan toimimaan snapshot-ratkaisun kanssa.

Vaikka pelin kehityksen kannalta olennaiset ominaisuudet saatiin toteutettua, mielestäni jatkokehityksessä olisi vielä paljon tehtävää. Opinnäytetyö olisi löydöksiltään vakuuttavampi, jos huonoksi todettu snapshot-ratkaisu olisi ehditty toteuttaa niin hyvin kuin mahdollista. Peliä ei voi pelata snapshot-ratkaisulla, joten esimerkiksi prosessorin kuormitusta ei ole mahdollista luotettavasti mitata.

Kehityksen aikana tehtyjen havaintojen perusteella snapshot-ratkaisua ei kannata sellaisenaan implementoida. Sen kehittäminen olisi kuitenkin parempien mittaustulosten kannalta olennaista. Pelin toimimisen kannalta sen jatkokehitys ei ole tarkoituksenmukaista.

Snapshot-ratkaisusta voisi olla hyötyä osana determinististä ratkaisua. Deterministisen ratkaisun suuri heikkous on toiston epätarkkuus. Snapshot-ratkai-

sulla voitaisiin tallentaa liikkuvien kappaleiden paikka esimerkiksi kerran sekunnissa, jonka avulla voidaan varmistaa, että deterministinen ratkaisu toimii tarkoituksenmukaisesti. Deterministisellä ratkaisulla saavutetaan animaatiotilojen toisto ja suorituskyky, snapshot-ratkaisu paikkaisi sen ongelmakohtia. Tällä hetkellä noin yksi sadasta toistosta epäonnistuu tarkkuutta vaativissa kentissä. Haluttaessa tätä lukua olisi mahdollista pienentää.

Suurin haaste oli fysiikan determinismi. Harkitsin oman fysiikkamoottorin kirjoittamista ongelman ratkaisemiseksi. Eri fysiikka-asetuksia testaamalla toisto saatiin kuitenkin toimimaan tyydyttävällä tarkkuudella. Fysiikkamoottorin kirjoittaminen olisi myös kasvattanut opinnäytetyön liian laajaksi.

Jatkokehityksen kannalta olisi mielenkiintoista tallentaa pelisuorituksia massamuistiin. Tämän pitäisi onnistua helposti tallentamalla kopiot hahmokohtaisia input-tietoja kirjaavista dictionaryistä JSON-tiedostoon. Tulosten jakaminen tietokoneiden välillä ei olisi mahdollista liukulukujen determinismiongelman vuoksi.

9 POHDINTA

Työn mittaustulokset ovat vain suuntaa antavia. Ne on tehty Unity-editorin sisällä, oikean peliversion suorituskyky voi poiketa tästä. Samojen muistinkäyttöä kuormittavien prosessien voidaan olettaa tapahtuneen jokaisella mittauskerralla. Mittaukset eivät ole myöskään kaikkein tarkimmat mahdolliset. Otin tilannekuvan, kun pelikello näytti kuluneeksi 3600 sekuntia (eli yksi tunti). Tämä olisi kannattanut automatisoida. Yhdellä mittauskerralla en ollut koneen ääressä oikeaan aikaan, joten tunnin kestävä mittaus piti tehdä uudestaan. Ihminen aiheuttaa mittauksiin aina virheitä. Mahdolliset virheet ovat tässä tapauksessa varsinkin tunnin kestävässä mittauksessa tulosten kannalta merkityksettömiä.

Mittaustulokset olivat teorian pohjalta ennakoitavissa. Deterministinen ratkaisu oli suorituskyvyn sekä kehittämisen helppouden kannalta parempi. Ero kahden ratkaisun välillä todennäköisesti kasvaisi, mikäli tekisin deterministiseen ratkaisuun muistin optimointia tai jatkaisin snapshot-ratkaisun kehittämistä pelikelpoiseksi. Tällä hetkellä snapshot-ratkaisuun ei tallenneta esimerkiksi mitään animaatiotilojen tietoja, jotka ovat uskottavan uusinnan kannalta välttämättömiä.

Unity valittiin pelimoottoriksi aikaisempien kokemusten perusteella. Minulla oli aikaisempia positiivisia kokemuksia yksinkertaisten 2D-pelien toteutuksesta moottorilla. Unreal Engine -pelimoottorista puolestaan negatiivisia. Kolmas varteenotettava vaihtoehto olisi ollut Godot-pelimoottori. Vähäisen Godot kokemuksen vuoksi työ olisi keskittynyt tuolloin todennäköisesti enemmän Unityn ja Godotin ominaisuuksien vertailuun, kuin tietyn ominaisuuden kehitystyöhön. Unity-pelimoottorin tavoin Unreal Engine ja Godot ovat molemmat ilmaisia ottaa käyttöön, joten hinta ei vaikuttanut valintaan.

Toinen mielenkiintoinen valinta olisi ollut käyttää Unityn Data-Oriented Technology Stack (DOTS) -järjestelmää. DOTS:in fysiikkamoottori mahdollistaa deterministisen fysiikan. DOTS kuitenkin siirtyy EC-framework-arkkitehtuurista ECS-arkkitehtuuriin, jonka käytöstä minulla on vain vähän kokemusta. Opinnäytetyön aihe olisi myös tällöin irtautunut kehitetyistä ominaisuuksista.

Työstä olisi tullut mielestäni parempi, jos olisin ehtinyt kehittämään oman fysiikkamoottorini. Opinnäytetyön eri osuudet olisivat voimakkaammin yhteydessä toisiinsa, mikäli fysiikkamoottori toimisi täysin samalla tavalla, kuin sen teoreettista toimintaa luvussa 5.2.1 esittelen. Tällöin olisi myös mahdollista analysoida kaupallisten moottoreiden heikkouksia determinismin suhteen. Jos sen saavuttaminen olisi ollut minun taidoillani helppoa, en näe syytä olla sisällyttämättä sitä moottoriin. Kaavojen kääntäminen koodiksi on aina mielenkiintoista.

Huomasin tutkiessani uusintatoiminnallisuuksia, että tiettyjen pelien nettikoodi toimii hieman samalla tavalla. Käyttäjältä kerätään vain input-tiedot, jotka matkaavat internetin yli pelipalvelimelle. Palvelimella tiedoilla päivitetään pelin tila. Eli jos pelaaja on esimerkiksi painanut ampumisnäppäintä, ainoastaan tieto näppäinpainalluksesta päivitetään palvelimelle, jossa itse laukaus tapahtuu. Pelin tila, eli esimerkissä tapahtunut laukaus, palautetaan käyttäjälle. Uskon, että tällaisella periaatteella toimiviin nettipeleihin olisi todella helppoa toteuttaa uusinnat keräämällä kaikki inputit ja toistamalla ne kuten deterministisessä ratkaisussa.

Kokonaisuudessaan työ onnistui hyvin. Tavoitteisiin päästiin, eikä ylitsepääsemättömiä ongelmia tullut vastaan. Opin paljon projektien aikatauluttamisesta.

Ohjelmoinnin osalta olisin toivonut enemmän haasteita, tekniset ongelmat deterministisen ratkaisun tarkkuuden osalta ratkaistiin moottorin asetuksia vaihtamalla. Lopputuloksena syntyneen deterministisen ratkaisun tallennettuja uusintoja on silti ilo seurata.

LÄHTEET

Apply interpolation to a Rigidbody s.a. Unity. WWW-dokumentti. Saatavissa: https://docs.unity3d.com/Manual/rigidbody-interpolation.html [viitattu 23.2.2024].

Atari. 1972. Pong. Videopeli. Sunnyvale: Atari.

Axon, S. 2016. Unity at 10: For better-or worse-game development has never been easier. Ars-Technica. WWW-dokumentti. Päivitetty 27.9.2016. Saatavissa: https://arstechnica.com/gaming/2016/09/unity-at-10-for-better-or-worse-game-development-has-never-been-easier/ [viitattu 13.2.2024].

CollisionDetectionMode2D s.a. Unity. WWW-dokumentti. Saatavissa: https://docs.unity3d.com/ScriptReference/CollisionDetectionMode2D.html [viitattu 23.2.2024].

Dawson, B. 2013 Floating-Point Determinism. Ramdom ASCII. WWW-dokumentti. Saatavissa: https://randomascii.wordpress.com/2013/07/16/floating-point-determinism/ [viitattu 26.4.2024].

Dealessandri, M. 2023. What is the best game engine: is Unity right for you? GamesIndustry.biz. WWW-dokumentti. Saatavissa: https://www.gamesindustry.biz/what-is-the-best-game-engine-is-unity-the-right-game-engine-for-you [viitattu 17.5.2024].

French, J. 2024. How to use script composition in Unity. Game Dev Beginner. WWW-dokumentti. Saatavissa: https://gamedevbeginner.com/how-to-use-script-composition-in-unity/ [viitattu 26.4.2024].

Fundamentals of garbage collection. 2023. Microsoft. WWW-dokumentti. Saatavissa: https://learn.microsoft.com/en-us/dotnet/standard/garbage-collection/fundamentals [viitattu 19.5.2024].

Goldberg, D. 1991. What Every Computer Scientist Should Know About Floating-Point Arithmetic. Oracle. WWW-dokumentti. Saatavissa: https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html [viitattu 30.4.2024].

Kananen, J. 2017. Kehittämistutkimus interventiotutkimuksen muotona. Jyväskylä: Jyväskylän ammattikorkeakoulu.

Memory Profiler s.a. Unity. WWW-dokumentti. Saatavissa: https://docs.unity3d.com/Packages/com.unity.memoryprofiler@1.0/manual/index.html [viitattu 18.4.2024].

Millington, I. 2007. Game Physics Engine Development. San Francisco: Elsevier – Morgan Kaufman Publishers.

Nintendo. 1981. Donkey Kong. Videopeli. Kioto: Nintendo R&D1.

Pernaa, J. 2013. Kehittämistutkimus opetusalalla. Jyväskylä: PS-kustannus.

Rigidbody2D.sleepMode s.a. Unity. WWW-dokumentti. Saatavissa: https://docs.unity3d.com/ScriptReference/Rigidbody2D-sleepMode.html [viitattu 23.2.2024].

Simpson, J. 2002. Game Engine Anatomy 101. *ExtremeTech* 12.4.2002. Verkkolehti. Saatavissa: http://www.blueopal.com/pdf/gameengines_parts_1-4.pdf [viitattu 17.5.2024].

Sony Computer Entertainment. 2009. Ratchet & Clank: A Crack in Time. Videopeli. Burbank: Insomniac Games.

SortedDictionary<TKey,TValue> Class s.a. Microsoft. WWW-dokumentti. Saatavissa: https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic.sorteddictionary-2?view=net-8.0 [viitattu 18.5.2024].

SSE Display Tweaks. 2020. Nexus Mods. WWW-dokumentti. Saatavissa: https://www.nexusmods.com/skyrimspecialedition/mods/34705 [viitattu 19.3.2024].

Understanding the managed heap. 2020. Unity. WWW-dokumentti. Saatavissa: https://docs.unity3d.com/2020.1/Documentation/Manual/BestPracticeUnderstandingPerformanceInUnity4-1.html [viitattu 22.5.2024].

Unity Personal s.a. Unity. WWW-dokumentti. Saatavissa: https://unity.com/products/unity-personal [viitattu 13.2.2024].

Wagner, C. 2004. Developing Your Own Replay System. Game Developer. WWW-dokumentti. Saaatavissa: https://www.gamedeveloper.com/programming/developing-your-own-replay-system [viitattu 19.2.2024].

Walker, A. 2018. Fallout 76 Quietly Unlocks Frame Rate Without Breaking The Game. Kotaku. WWW-dokumentti. Saatavissa: https://www.kotaku.com.au/2018/11/fallout-76-quietly-unlocks-frame-rate-without-breaking-the-game/ [viitattu 7.4.2024].