Kai Hyvönen

UNITY-SOVELLUKSEN OPTIMOINTI: WINMIND APP

Opinnäytetyö

Tekniikan ammattikorkeakoulututkinto

Peliohjelmoinnin koulutus

2025

Kaakkois-Suomen ammattikorkeakoulu

Kaakkois-Suomen ammattikorkeakoulu

Tutkintonimike	Insinööri (AMK)
Tekijä/Tekijät	Kai Hyvönen
Työn nimi	Unity-sovelluksen optimointi: WinMind App
Toimeksiantaja	WinMind Oy
Vuosi	2025
Sivut	61 sivua
Työn ohjaaja(t)	Pekka Vilpponen

TIIVISTELMÄ

Opinnäytetyön tarkoituksena oli optimoida Unity-sovelluksen toimintaa. Työssä tarkasteltiin profilointityökaluja ja perehdyttiin niiden käyttöön sovelluksen optimoinnin kohteiden etsinnässä. Työssä keskityttiin yhden sovelluksen optimointiin, joten löydetyt ratkaisut eivät suoraan paranna toisten sovelluksien suorituskykyä, mutta ratkaisujen takana olevaa logiikkaa voidaan hyödyntää muissakin sovelluksissa.

Opinnäytetyön toimeksiantajana toimi WinMind Oy. Työ on toiminnallinen tutkimus, jonka tavoitteena on optimoida WinMind App -sovelluksen toimintaa ja täten luoda siitä paranneltu versio toimeksiantajan käyttöön. WinMind App on lasten ja nuorten nykytilanteen kartoitussovellus.

Opinnäytetyön teoriaosuudessa on kerätty aineistoa virallisista dokumentaatioista sekä aiheeseen liittyvästä kirjallisuudesta. Tutkimuksen aikana aineistoa on lähtökohtaisesti syntynyt havainnoinnilla ja kokeilemalla. Molempia aineistoja hyödyntämällä pystyttiin tarkentamaan optimoinnin prosesseja sekä valitsemaan parhaat työkalut ongelmien ratkaisemiseen.

Opinnäytetyössä onnistuttiin ratkaisemaan tutkimusongelma, joka oli WinMind App -sovelluksen optimointi. Tutkimusongelma ratkaistiin vastaamalla siitä johdettuihin tutkimuskysymyksiin, joiden avulla voitiin määrittää tarpeelliset työkalut ja toimenpiteet halutun lopputuloksen saavuttamiseksi.

Opinnäytetyön lopputuloksena on syntynyt parempi versio sovelluksesta, jossa optimoinnin onnistuminen voidaan todennettavasti havainnoida profilointityökaluista kerätyn datan vertailuanalyysin kautta. Tutkimustuloksista tehtyjen johtopäätöksien perusteella Unity-sovelluksen optimoinnissa parhaat työkalut ovat jo valmiiksi sisällytettynä Unity Editor -ympäristöön. Unity-työkalujen käyttäminen helpottaa ongelmakohtien tunnistamista, auttaa kohdentamaan optimoinnin niitä tarvitseviin osa-alueisiin sekä selkeyttää sovelluksen kehitysprosessia.

Asiasanat: ohjelmistokehitys, optimointi, profilointi, suorituskyky

South-Eastern Finland University of Applied Sciences

Degree title	Bachelor of Engineering
Author (authors)	Kai Hyvönen
Thesis title	Optimization of a Unity application: WinMind App
Commissioned by	WinMind Oy
Time	2025
Pages	61 pages
Supervisor	Pekka Vilpponen

ABSTRACT

The objective of this thesis was to optimize a Unity application. The work scrutinized different profiling tools and researched how they can be used to locate optimization targets within an application. This work focused on a single application, which means that the found solutions regarding optimizations do not automatically correlate to possible solutions in other applications and cannot be applied as is, but the logic behind them can still be utilized in other applications to achieve a more optimized experience.

This work was commissioned by WinMind Oy. This work is conducted as practice-based thesis, and it aims to optimize the performance of an application called WinMind App, thus, creating a better version of the application. This version will be given back to WinMind to be used as a wellbeing survey tool to assess the current situation of children in the child welfare system.

Research material for the thesis was gathered from official documentations and from literature on the subject. More material was produced during research via trial and observations. Making use of all the available research material, be it newfound or already established, allowed for the improvement of the optimization processes. This also gave insight as to which profiling tools were best suited for a specific task.

The optimization of the WinMind App formed the research question for this thesis. To answer the research question, objectives were derived from it in the form of questions. Reaching these objectives helped defining the necessary tools and measures that needed to be taken to reach a desired outcome.

The result of this thesis is an improved version of the application where the success of the optimizations can be observed through benchmark comparison. Conclusions made from the research results indicate that the best tools for optimizing a Unity application are already included in the Unity Editor environment. Using Unity tools facilitates the identification of problematic sections, directs attention to areas in need of optimization and clarifies the software development process.

Keywords: optimization, performance, profiling, software development

SISÄLLYS

1 JOHDANTO

Teknologian edetessä huimaa vauhtia voi helposti vahingossa jättää huomiotta vanhemman sukupolven laitteiston. Kaikkien elämäntilanteet eivät salli uusimman puhelinmallin hankkimista, saati sitten tietokoneen ostamista. On siis tärkeää varmistaa sovelluskehityksen aikana, että mahdollisimman monella laitteella olisi mahdollista käyttää luotua sovellusta. Tätä varten sovelluskehityksen aikana on suoritettava optimointia. Tämän opinnäytetyön toimeksiantajana toimii WinMind Oy ja tarve työlle syntyi heidän tarjoamastaan palvelusta, sovelluksesta, johon työssä on tarkoituksena suorittaa optimointia.

WinMind on vuonna 2022 perustettu startup-yritys, jonka tarkoituksena on kehittää digitaalisia alustoja sosiaali- ja terveysalalle hyödyntäen peliteknologiaa (Meistä s.a.). Yritykseen suoritetun työharjoittelun aikana on kehitetty sovellus, WinMind App, ja sen optimointi toimii opinnäytetyön pääkohteena. WinMind App on Unity-pelimoottorilla toteutettu sovellus, jonka tarkoituksena on digitalisoida monet lastensuojelussa käytettävät paperilomakkeet interaktiiviseksi peliksi (Winmind Appi s.a.). Sovelluksen käyttöön ottaminen on lastensuojeluyksiköiden oma päätös, eikä WinMind Oy tarjoa laitteistoa sovelluksen käyttämiseen, sillä yritys tarjoaa ohjelmistoa palveluna (Liiketoimintamalli s.a.).

Lastensuojeluyksiköt hankkivat itse laitteiston tai joissain tapauksissa nuorten omia laitteita voidaan käyttää. Näiden laitteiden teknisistä tiedoista ei ole mitään varmuutta, joten olisi suotavaa saada sovelluksen vaatimukset alhaisemmiksi niin tehokkuuden kuin virrankulutuksenkin puolesta. Sovelluksen testauksen aikana on ollut havaittavissa, etenkin vanhemmilla laitteilla, suuri virrankulutus sekä kuvataajuuden suorituskyvyn puutteellisuus huonontuvan kuvataajuuden muodossa.

Opinnäytetyön tavoitteena on hyödyntää olemassa olevia työkaluja sovelluksen profilointiin, kartoittaa sovelluksen optimoinnin tarpeita sekä toteuttaa korjaavia toimenpiteitä tunnistetuille ongelmakohdille. Optimoinnin kohteet arvioidaan, ja niistä kriittisimmille suoritetaan optimointia, jotta sovelluksen toiminta paranee.

2 TUTKIMUSASETELMA

Työssä uppoudutaan Unity-pelimoottorin tarjoamiin profilointityökaluihin ja tarkastellaan ohjelmiston optimointia yleispätevällä ja kohdennetulla tasolla. Huomioitavaa työssä ovat myös sovelluksen loppukäyttäjät ja heidän olosuhteensa. Työssä tarkastellaan myös kolmannen osapuolen tarjoamia profilointityökaluja, mutta suurin osa tehtävästä profiloinnista tapahtuu Unityn editorissa Unityn tarjoamilla työkaluilla.

2.1 Tutkimusongelma- ja kysymykset

Sovelluksen kehityksen aikana ei ole käytetty resursseja optimointiin. Tämä saattaa ilmetä esimerkiksi korkeana virran kulutuksena tai epäsulavana käytettävyytenä. Optimoinnin avulla voidaan varmistaa, että sovellus toimii sujuvasti eri laitteilla ja alustoilla ja antaa käyttäjälle johdonmukaisen ja miellyttävän käyttökokemuksen (Essam 2024, 242).

Ennenaikainen optimointi voi kuitenkin aiheuttaa enemmän haittaa, sillä se voi viedä huomion ohjelmiston ei-kriittisiin osiin ja kuluttaa resursseja siinä vaiheessa, kun ohjelmiston tärkeimmät optimoinnin kohteet eivät ole vielä edes tunnistettavissa. Tämä ei kuitenkaan tarkoita sitä, etteikö optimointia voisi suorittaa sovelluksen kehityksen aikana. Optimointia tulisi suorittaa kriittisessä koodissa, mutta vasta sen tunnistamisen jälkeen. Usein intuitiivinen päättely koodin tärkeydestä kuitenkin menee väärin ja tämä huomataan mittaustyökaluja käytettäessä. (Knuth 1974, 268.)

WinMind App -sovellus on pilottikäytössä ja sen toiminnan kriittisimmät osa-alueet ovat jo olemassa. Optimoinnin kohdentamista ei kannata suorittaa oletamuksien pohjalta, vaan kriittinen koodi täytyy tunnistaa työkaluilla, sen toimivuutta tarkastella ja tarvittaessa parantaa koodia. Tutkimusongelmaksi kiteytyy WinMind App -sovelluksen optimointi ja siitä voidaan johtaa seuraavat tutkimuskysymykset:

1. Mitä työkaluja Unity-sovelluksen profilointiin kannattaisi käyttää?
2. Miten Unity-sovelluksen optimoinnin tarpeet voidaan tunnistaa?
3. Miten WinMind App -sovellusta voidaan optimoida?

Tutkimuskysymyksiin vastaamalla opinnäytetyössä pyritään ratkaisemaan asetettu tutkimusongelma. Mikäli ongelma saadaan ratkaistua, se parantaa sovelluksen käytettävyyttä heikommallakin laitteistolla ja varmistaa sen toimivuuden eri alustoilla, kuten Android ja WebGL.

2.2 Tutkimusote

Opinnäytetyössä keskitytään yksittäisen sovelluksen optimointiin, jossa tehtävät toimenpiteet eivät välttämättä ole yleistettävissä muihin sovelluksiin. Työn toteutus voitaisiin suorittaa joko tapaus-, kehittämis- tai toimintatutkimuksena.

Tapaustutkimuksessa voi olla piirteitä laadullisesta (kvalitatiivinen) sekä määrällisestä (kvantitatiivinen) tutkimuksesta. Tutkimus voi keskittyä puhtaasti vain toiseen näistä tutkimusotteista, mutta ne eivät sulje toisiaan ulos ilmiöitä tutkittaessa (Yin 2002, 15). Tässä tutkimuksessa otettiin huomioon loppukäyttäjien resurssit ja tarpeet, mutta samalla pohjustettiin mahdolliset optimoinnit olemassa olevaan teoriapohjaan. Vaikka pyrkimyksenä on saada syvällinen ymmärrys sovelluksen toiminnasta ja optimoinnin tarpeesta, ei tutkimusta voi luokitella enää tapaustutkimukseksi, koska mahdolliset optimoinnit on myös tarkoitus toteuttaa. Tapaustutkimuksen on tarkoitus selittää, kuvailla ja havainnollistaa tutkimuskohdettaan, eikä olla interventionaalinen tutkimus. (Yin 2002, 15.)

Opinnäytetyössä on molempien kehittämis- sekä toimintatutkimuksen vivahteita; työn kirjoittaja on ollut itse sovelluksen kehityksessä aktiivisesti läsnä ja sovelluksen käytettävyyttä tutkitaan työn kautta. Tutkimuskohteen ja siihen liittyvän ympäristön kehittäminen ovat yhdistäviä tekijöitä näille menetelmille. Toimintatutkimus keskittyy kehittämään paikallisesti toimivia ratkaisuja, kun taas kehittämistutkimuksen tarkoituksena on luoda uutta teoriaa ja pienen mittakaavan tuloksia pyritään yleistämään suurempaan mittakaavaan (Pernaa 2013).

Opinnäytetyössä hyödynnetään olemassa olevaa teoriapohjaa ratkaisujen luomisessa ja ratkaisujen pohjalta syntyy tapauskohtaista teoriaa sovelluksen op-

timoinnista. Toimintatutkimukselle yleisiä piirteitä ovat käytäntöihin suuntautuminen, muutokseen pyrkiminen ja tutkittavien osallistuminen tutkimusprosessiin. Toimintatutkimuksen tavoitteena on ratkaista organisaatiossa ilmenevä käytännön ongelma luoden samanaikaisesti uutta tietoja ja ymmärrystä ilmiöstä. (Kuula 1999, 10; Ojasalo ym. 2015, 58.) Työn kirjoittajan osallisuus kehitystyössä ja teorian soveltaminen toteutusvaiheessa rajoittavat työn tutkimusotteeksi toimintatutkimuksen.

2.3 Aineiston keruu ja analyysi

Työssä käytettiin olemassa olevaa teoriaa optimoinnin saavuttamiseksi ja tiedon alkuperään viitataan selkeästi. Alalla yleisesti tunnetuille tiedoille, kuten Unityn omista dokumentaatioista saaduille perustiedoille, ei välttämättä luoda lähdeviittauksia, mutta ne sisällytetään lähdeluetteloon (Hakala 2022, 117).

Toimintatutkimuksessa on haastavaa se, että tutkimuskohde on aina tilanteeseen sidottu. Tämän vuoksi aikaisempia tuloksia on vaikea hyödyntää. (Ojasalo ym. 2015, 59.) Opinnäytetyön kohteena ei kuitenkaan ollut esimerkiksi sosiaalinen ilmiö, vaan sovellus, jonka pysyvä tila oli hallittavissa.

Opinnäytetyössä pyrittiin välttämään keskustelupalstoja ja Q&A-sivustoja lähteinä, sillä niiden sisältö ei välttämättä ole oikeellista. Jos ainoa ratkaisu ongelmaan löytyi kyseenomaisista lähteistä, tarkasteltiin ratkaisun vaikutusta sovelluksen toimintaan ja etsittiin vaihtoehtoinen lähde selittämään ilmiötä ja ratkaisun vaikutusta. Ensisijaisesti lähteinä työssä käytettiin kuitenkin aiheeseen liittyvää kirjallisuutta ja virallisia dokumentaatioita. Materiaalia kerättiin kirjastoissa, tieteellisten julkaisujen julkaisukanavilla sekä virallisten tahojen verkkosivuilla, kuten Unityn dokumentaatiosivuilla.

Työssä ei pystytty hyödyntämään toimintatutkimuksen yleistä menetelmää, diskurssia, sillä työ suoritettiin yksin. Diskurssi tarkoittaa toimijoiden yhteisiä keskusteluita, joita voidaan hyödyntää kehittämisprosessissa (Ojasalo ym. 2015, 62). Yksin työskentely ei onneksi estä kyselyiden teettämistä tai haastattelua, jos halutaan kerätä tietoa loppukäyttäjiltä. Keskeisimmäksi keräysmetodiksi muodostui havainnointi, joka on yksi tehokkaimmista tavoista kerätä aineistoa (Ojasalo ym. 2015, 61).

2.4 Aiemmat tutkimukset

Opinnäytetyön aihe oli laajemmassa mittakaavassa Unity-sovelluksen optimointi. Theseuksesta löytyi lukuisia opinnäytetöitä, jotka keskittyivät optimointiin Unity-ympäristössä. Ohessa neljä esimerkkityötä:

Narkilahden (2021) opinnäytetyö, *Unity-optimointi*, käsittelee Unity-pelimoottorissa käytettäviä optimointitekniikoita sekä menetelmiä koodin, grafiikan ja suorituskykyongelmien etsimisen kannalta. Työ keskittyy hyviin käytäntöihin ja Unityn tarjoamiin työkaluihin.

Lehtolan (2018) opinnäytetyö, *Optimizing Unity Projects*, on englannin kielellä kirjoitettu työ, jossa keskitytään työn kirjoittajan työharjoittelussa suorittamaan projektin optimointiin. Työ pyrkii tehostamaan projektin toimivuutta Mac-laitteilla. Työssä käydään läpi teoriaa optimoinnin suorittamisesta ja lopuksi käydään läpi Virtual Frontiers -projektille suoritetut optimoinnit, niiden haasteet sekä ratkaisut.

Viljamaan (2017) opinnäytetyön, *Unity3D, palvelimet ja mobiilipelin optimointi*, tavoitteena oli opetella 3D-pelin optimointia niin, että peliä voitaisiin pelata sujuvasti keskivertokuluttajan mobiililaitteella. Nimensä mukaisesti työ ottaa mobiilialustat huomioon suunnittelussa ja kehitysvaiheessa. Lopuksi työssä käydään läpi aikaansaadun mobiilipelin ominaisuuksia.

Okkolan (2023) opinnäytetyö, *Pelitestaus ja profilointi Unity-pelimoottorissa: Case: VR-HYPO virtuaalinen oppimisympäristö*, keskittyy testaamaan Unityssa toteutettua virtuaalisovellusta. Työn yhteenvedossa havainnoidaan, että testaus tulisi aloittaa hyvissä ajoin, ja painotetaan testauksen tärkeyttä. Projektissa kokeiltiin monia optimointikeinoja ja joissakin tapauksissa päädyttiin käyttämään helpommin implementoitavaa, tehottomampaa ratkaisua, sillä ratkaisut eivät tuoneet huomattavia parannuksia.

2.5 Luotettavuuden arviointi ja eettiset näkökulmat

Tutkimuksissa luotettavuuden arvioinnissa käytetään usein validiteetin ja reliabiliteetin käsitteitä. Tarpeen mukaan, nämä voidaan jakaa vielä useampaan

alakäsitteeseen. (Yin 2002, 34–35.) Reliabiliteetti tarkoittaa sitä, että työn on oltava toistettavissa samoilla tuloksilla. Siihen voidaan vaikuttaa esimerkiksi muistiinpanojen ja kirjanpidon tarkkuudella. (Hakala 2022, 285.) Validiteetti koskee havaintojen tulkintojen oikeanmukaisuutta; käyttämällä oikeita termejä ja käsitteitä voidaan varmistaa, että tutkimuksessa tutkitaan oikeita asioita. (Hakala 2022, 289; Yin 2002, 34.)

Ongelmaksi luotettavuuden kannalta opinnäytetyössä nousee kirjoittajan olemassa oleva ymmärrys projektista. Yin (2002, 62) tuo esille taipumuksen puolueellisuuteen ja ennakkokäsityksiin, jos tutkimuksen laatijalla on olemassa oleva tuntemus projektista. Opinnäytetyössä tämä voi näkyä keskittymisellä tiettyihin projektin osa-alueisiin, joiden on oletettu olevan kriittisempiä optimoinnin kohteita.

Havainnoinnin ja muistiinpanojen ylös kirjaamisessa tulee ottaa myös huomioon se, että tutkija voi keskittyä tiettyyn aihealueeseen ja jättää huomiotta haluamansa osiot. Kirjallisissa lähteissä tätä ongelmaa ei kuitenkaan ole, sillä tutkija ei voi määrittää mihin olemassa olevan tekstin sisältö keskittyy. (Peräkylä, 283–284.) Ohjelmiston perusteellisella läpikäynnillä, kattavalla dokumentoinnilla ja päätöksien perustelulla nämä luotettavuuden ongelmat voidaan kuitenkin minimoida, ellei kokonaan välttää.

Ollakseen eettisesti hyväksyttävä ja luotettava täytyy tutkimuksen noudattaa hyvän tieteellisen käytännön edellyttämiä tapoja (Tutkimuseettinen neuvottelukunta 2024). Väärentäminen ja plagiointi ovat tyypillisimpiä tutkimuseettisistä väärinkäytöistä (Kuula 2011, 29). Työn tulokset tulevat vaikuttamaan suoraan kirjoittajan omaan työskentelyyn sovelluksen kanssa, joten olisi hyvin epäloogista väärentää tuloksia ja niistä johdettua tietoa.

Työn eettisyyden kannalta nousee esiin kohdeyleisö ja loppukäyttäjät. Sovellus on tarkoitettu nuorten käyttöön ja lapset kuuluvat suojeltavaan kohderyhmään, jolla ei ole täysivaltaista itsemääräämisoikeutta. He eivät voi päättää osallistumisestaan tutkimukseen. (Kuula 2011, 147.) Tästä syystä palaute, jota tutkimuksessa ja sovelluksen kehityksessä käytettiin, oli lähtökohtaisesti aikuisten ohjaajien palautetta. Lasten tai ohjaajien sovelluksen käytön aikana

luomaa dataa ei käytetty tutkimuksessa ja kun tarve syntyi demonstroida toimintoja, luotiin tekaistuja tietoja simuloimaan oikeaa toimintaa.

3 TEOREETTINEN VIITEKEHYS

Opinnäytetyössä keskitytään Windows- ja Android-käyttöliittymiin. Teoreettisen viitekehyksen ja koko opinnäytetyön rajaavia tekijöitä ovat resurssien puute budjetin muodossa sekä projektin alkuperä. Projektissa on käytetty olemassa olevaa infrastruktuuria Unity-pelimoottorissa aiemmin kehitetystä sovelluksesta ja toimeksiantaja on halunnut tarjota sovelluksen mobiililaitteille sekä tietokoneille. Näistä syistä sovelluksen kehityksessä on pyritty välttämään maksullisia työkaluja ja palveluita.

3.1 Profilointityökalut

Profilointityökaluilla voidaan tarkkailla jonkin sovelluksen toimintaa. Tyypillisesti profilointityökaluilla mitataan ohjelman suorittamista, esimerkiksi mittaamalla koodin suorittamisen kestoa tai muistin käyttöä. (Microsoft 2021.) Profilointi on paras keino mitata sovelluksen suorituskykyä, ja sillä voidaan lisätä ymmärrystä sovelluksen ongelmakohdista, olivat ne sitten muistin käytössä, grafiikkasuorittimen toiminnassa, prosessorin toiminnassa tai itse tehdyissä koodeissa (Unity Technologies 2025e).

Jos sovellus on vielä kehitysvaiheessa, eikä siitä ole koontiversiota, on suositeltavaa käyttää Unityn sisäisiä profilointityökaluja. Kolmannen osapuolen profilointityökaluja ei kuitenkaan kannata jättää hyödyntämättä. Joskus koontiversion toiminta eroaa editoriversiosta aiheuttaen hankalasti analysoitavia virhetilanteita sovelluksen käytössä.

3.1.1 Unity

Unityn kanssa käytettävät profilointityökalut voivat olla kolmannen osapuolen sovelluksia. Unity kuitenkin tarjoaa itse omia profilointityökaluja pelimoottorin sisäisesti. Näitä ovat esimerkiksi Unity Profiler, Memory Profiler, Profile Analyzer ja Frame Debugger.

Unity Profiler -työkalulla mitataan sovelluksen toimintaa editorissa tai sen voi yhdistää laitteeseen, jossa sovellusta käytetään kehittäjätilassa. Memory Profiler tarjoaa keinon syvälliseen muistin käytön analysointiin. Profile Analyzer antaa vertailla kahta tietoaineistoa keskenään ja tarkastella tehtyjen muutoksien vaikutusta sovelluksen toimintaan. Frame Debugger mahdollistaa sovelluksen graafisen suorituskyvyn mittaamisen. (Unity Technologies 2025e.)

Näitä kaikkia työkaluja yhdistää se, ettei niitä voida käyttää julkaistussa sovelluksessa, vaan sovelluksen täytyy olla kehitysympäristössä. Integroidut työkalut auttavat kehittäjää sovelluskehityksen vaiheissa, kun taas kolmannen osapuolen työkalut voivat auttaa loppukäyttäjien todellisen käyttökokemuksen selvittämisessä.

3.1.2 Intel

Intel ja AMD ovat kaksi yleisimmistä prosessoreiden valmistajista (Microsoft s.a.). Ne molemmat käyttävät Intelin kehittämää x86 suoritinarkkitehtuuria, johon AMD sai lisenssioikeudet 1980-luvulla tapahtuneen sopimuksen myötä, jossa IBM sai Intelin myöntämään lisenssioikeuksia kolmannen osapuolen valmistajille varmistaakseen tuotannon määrän (Sexton 2017). Pöytäkoneiden osalta, Intel ja AMD muodostavat todennäköisimmät valmistajat prosessoreille, joita loppukäyttäjien laitteista löytyy.

Intel tarjoaa profilointityökalun nimeltä Intel® VTune™ Profiler. Sitä voidaan käyttää Unity-sovelluksien profilointiin ja Intel on koostanut ohjeistuksen sen käyttöön Unity ympäristössä. Sovelluksesta tulee koota ensin versio, sillä työkalua ei voi käyttää kehitysympäristössä. 2019.2 ja uudemmissa Unity versioissa on editoriin sisäänrakennettu Instrumentation and Tracing Technology (ITT) rajapinta. Rajapinnan avulla koottua versiota voidaan analysoida Intelin profilointityökalulla. (Intel Corporation 2024c.)

3.1.3 AMD

Kuten luvussa 3.1.2 mainitaan, AMD on toinen merkittävä prosessoreiden valmistaja ja sekin on tehnyt ohjeistukset Unity-sovellusten profilointia varten. Ohjeistuksessa AMD varoittaa oman profilointityökalunsa AMD μProf käytöstä,

sillä se ei välttämättä sovellu käyttöön Unityn kanssa, joka käyttää kehityskielenään C# -ohjelmointikieltä eikä C++ -ohjelmointikieltä. (Advanced Micro Devices s.a.).

Ohjeistuksessa käytetään Microsoftin tarjoamaa Windows Performance Toolkit ohjelmaa. Ohjelma sisältää profilointityökaluja, jotka soveltuvat Windows-käyttöliittymässä suoritettavien sovelluksien analysointiin (Microsoft 2022). Oman profilointityökalunsa käytön AMD jättää Unityn suhteen selittämättä.

3.1.4 Android

Android tarjoaa Android Debug Bridge (ADB) -komentorivityökalun, jolla voidaan testata Android-sovelluksia (Android Developers 2025). Kun sovelluksesta on koottu versio Android-alustalle, kytketään mobiililaite tietokoneeseen USB-johdolla tai langattomalla yhteydellä. Sovelluksen käytön aikana laitteesta voidaan ottaa talteen viestejä, jotka kertovat sovelluksen tilasta. (Unity Technologies 2025b.)

Talteen otettujen viestien perusteella voidaan havaita mahdollisia häiriöitä suorituskyvyssä ja paikantaa sen alkuperä. Puhelimen komponentit ovat yleensä tietokoneisiin verrattuna heikkotehoisempia, joten sovelluksen tavallinen käyttäminen saattaa paljastaa huonosti toimivia ominaisuuksia. Ennen koontiversion luontia, voidaan käyttää editorin yhteydessä Unity Remote sovellusta. Sovellus yhdistää kohdelaitteen editoriin ja näyttää editorin syötteen kohdelaitteen näytöllä (Unity Technologies 2025f). Näin voidaan simuloida sovelluksen toimintaa mobiilialustalla jo kehitysvaiheessa.

3.2 Ohjelmistoalustat

Työssä kohteena oleva sovellus on kehitetty pääsääntöisesti mobiilialustoille, tarkennettuna Android-mobiilikäyttöjärjestelmälle. Sovellusta voidaan käyttää WebGL -ohjelmistorajapinnan avulla yleisissä nettiselaimissa, kuten Safari, Chrome, Edge ja Firefox (The Khronos Group Inc. s.a.). Tämä mahdollistaa sovelluksen käyttämisen Applen iOS- ja macOS-laitteilla ilman niitä varten luotua koontiversiota. Sovelluskehityksessä paikallisen koontiversion tuottaminen Applen laitteille vaatii Xcoden käyttöä, joka on saatavissa vain macOS-käyttöjärjestelmän omaavilla laitteilla (Unity Technologies 2025a).

Applen laitteiden hankkiminen tuo kustannuksia, joihin resursseja ei ole varattu. Sen lisäksi, Applen laitteille sovelluksen julkaiseminen toisi lisämaksuja. Applen kehittäjäohjelman jäsenyys, joka sallisi sovellusten julkaisemisen Applen kauppapaikkaan, maksaa 99 dollaria (noin 95 euroa) vuodessa (Apple Inc. s.a.). Vastaavasti Play Storeen, Googlen kauppapaikkaan, Android-sovelluksen julkaiseminen maksaa 25 dollarin kertamaksun (noin 24 euroa) per sovellus (Google s.a.).

Applen laitteille julkaisujen tekemättä jättäminen ei välttämättä karsi potentiaalisten käyttäjien määrää vaikuttavasti. StatCounterin (2025b) mukaan, helmikuussa vuonna 2025, 71.72 prosenttia puhelimista käytti Android-käyttöliittymää, 27.81 prosenttia iOS-käyttöliittymää ja alle prosentin osuus jäi lopuille käyttöliittymille. Vastaavasti pöytäkoneilla, 70.54 prosenttia laitteista käytti Windows-käyttöjärjestelmää, macOS-käyttöjärjestelmää käytti 15.77 prosenttia laitteista (StatCounter 2025a).

Windows- ja Android-käyttäjät muodostavat saavutettavamman kohdeyleisön loppukäyttäjistä. Näille alustoille kehittäminen sopii toimeksiantajan tavoitteisiin ja auttaa välttämään ylimääräisten kustannuksien kertymistä kehityksen aikana resurssien ollessa vähäisiä. Sovellukselle valitut alustat auttoivat myös rajaamaan potentiaalisten optimointityökalujen käyttöä ja vähensivät huomioon otettavien osa-alueiden monimutkaisuutta. Sovelluksen kehityksessä usealle alustalle optimoinnin haasteet tulevat näkyvämmin esille WebGL-version yhteydessä, kun sovellusta voidaan käyttää lukuisilla eri laitteilla ja käyttöliittymillä.

3.3 Optimoinnin periaatteet

Optimoinnin prosessi tarkoittaa parhaan lopputuloksen saavuttamista. Käytännössä tämä usein tarkoittaa suurimman mahdollisen, maksimin, tai pienimmän mahdollisen, minimin, saavuttamista. (Antoniou & Lu 2007, 1.) Antoniou ja Lu (2007) keskittyvät julkaisussaan matemaattiseen optimointiin ja parhaan mahdollisen ratkaisun löytämiseen todistettavalla tavalla. Opinnäytetyö pyrki parantamaan projektin toimivuutta parannuksilla ja teoreettiseen maksimiin tai minimiin ei päästy työn mittakaavassa, niihin voitiin vain pyrkiä. Jotta voitaisiin

saavuttaa parannuksia sovelluksen toiminnassa, on oltava vertailukohteita, joista johtaa tuloksia. Tätä varten voidaan käyttää benchmarkingia.

Benchmarking eli vertailuanalyysi tarkoittaa prosessia, jossa mitataan laatua ja kerätään informaatiota systeemin tilasta. Vertailuanalyysi pyrkii vastaamaan määriteltyyn kysymykseen ja tyypillisesti sen avulla verrataan keskenään eri versioita, asetuksia, vaihtoehtoisia ratkaisuja ja käyttöönoton alustoja. (Grambow ym. 2019, 242.) Yksinkertaistettuna nämä voisivat tarkoittaa uuden ja vanhan sovellusversion vertailua, versioihin eri resoluution asettamista, listojen lajittelemista eri tavoilla, esimerkiksi LINQ verrattuna foreach-läpikäyntiin ja pilvipalveluiden käyttäminen Azure-ympäristössä verrattuna Amazon AWS -ympäristöön.

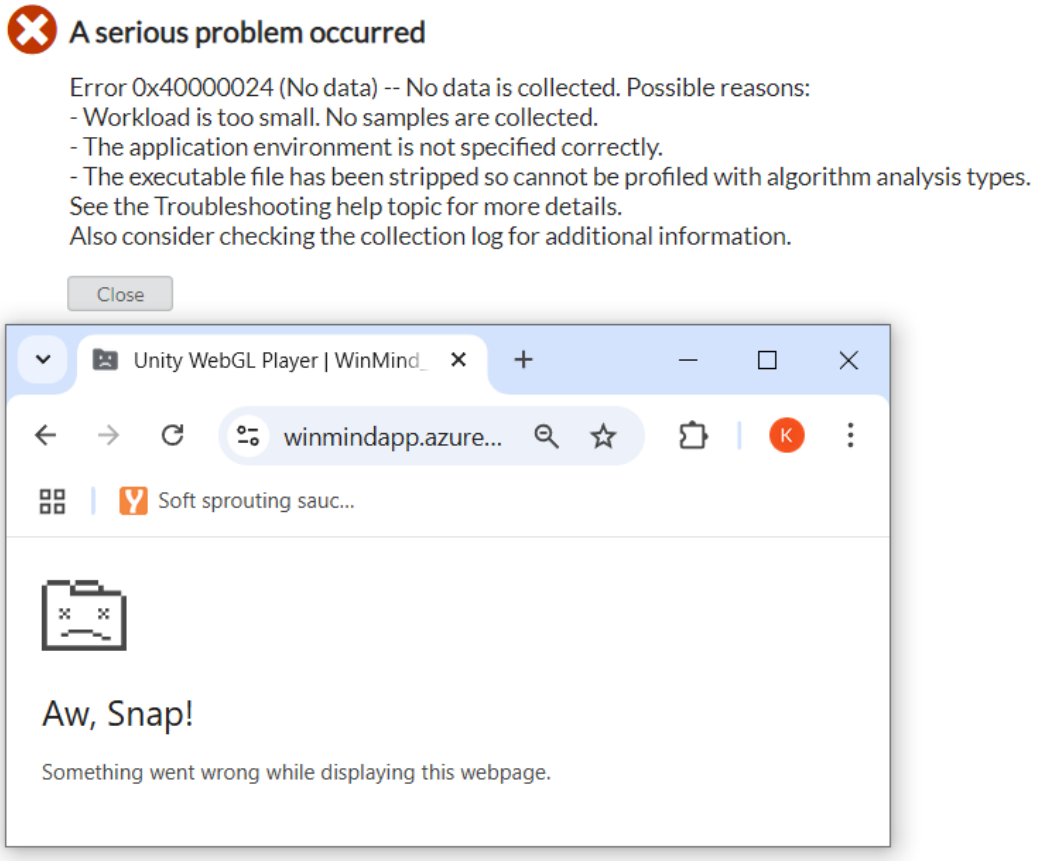
Smaalders (2006, 46) määrittelee artikkelissaan, että hyvä vertailuanalyysi on toistettavissa, siitä pitää pystyä tekemään havaintoja, sitä pitää pystyä käyttämään eri alustoilla, sen tulokset ovat helposti näytettävissä, se pysyy todellisten tilanteiden rajoissa ja sen tulee olla suoritettavissa. Työssä käytetyillä profilointityökaluilla voitiin suorittaa vertailuanalyysejä, joskin rajatusti Windows- ja Android-ympäristöissä. Smaalders (2006, 46) tekee myös huomion siitä, että kaikki laaditut vertailuanalyysit eivät saavuta hyvän vertailuanalyysin kriteerejä, joten olisi suotavaa käyttää useampaa projektin analysoinnissa.

Laatimalla ja suorittamalla vertailuanalyysejä, voidaan selvittää ovatko optimoinnit olleet onnistuneita ja mitata niiden onnistumista numeerisesti. Profiloimalla sovelluksen toimintaa eri tilanteissa ja optimoinnin vaiheissa, tuotetaan vertailuanalyysejä varten informaatiota. Samalla profilointi paljastaa sovelluksen puutteellisia osa-alueita ja ongelmakohtia, joihin voidaan alkaa suorittamaan optimointia. Näistä toimenpiteistä syntyy prosessi, jossa profiloidaan sovelluksen toimintaa, optimoidaan ongelmallinen osio, profiloidaan optimointien jälkeinen tila ja suoritetaan näistä vertailuanalyysi. Optimaalisessa tilanteessa lopputuloksena syntyy parempi, todistettavasti optimoitu versio sovelluksesta.

4 PROFILOINTITYÖKALUN VALINTA

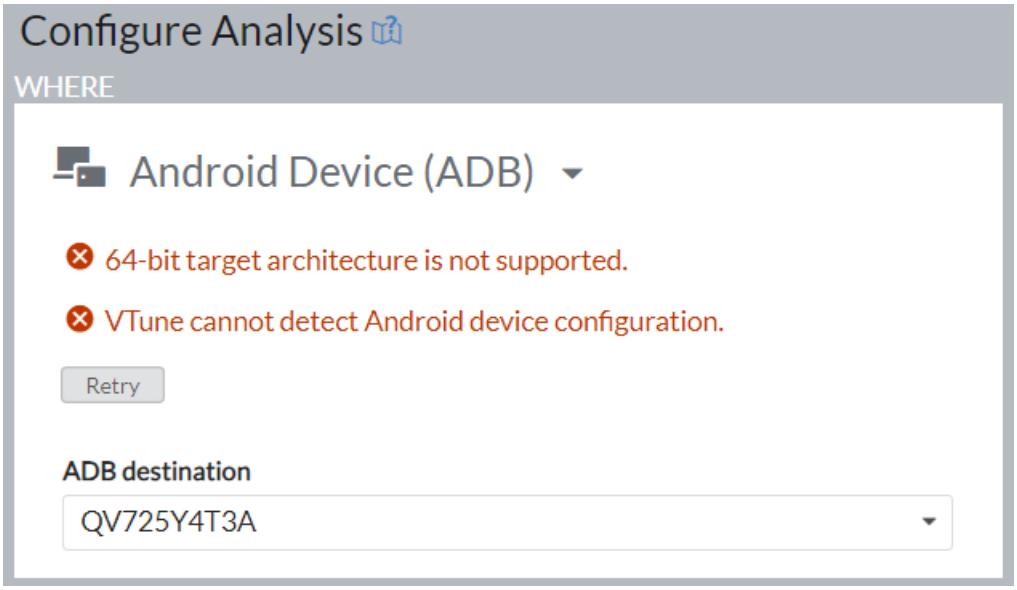
Johdonmukaisen profiloinnin varmistamiseksi on tarkasteltava eri työkaluja ja rajata niistä käyttöön vain sopivimmat. Tähän vaikuttavat esimerkiksi profilointityökalun helppokäyttöisyys ja tekniset rajoitukset. Ensimmäinen rajoite sovelluksen suhteen tulee sen kohdealustoista. Sovellusta kehitetään Android-laitteille sekä selainpohjaan hyödyntäen WebGL-ohjelmointirajapintaa, joten profilointityökalujen tulee pystyä kiinnittymään näihin prosesseihin.

Intelin VTune Profiler -työkalu on tarkoitettu käytettäväksi sovelluksille joita ajetaan suoraan prosessorin laskentateholla (Intel Corporation 2024b). Tämä tarkoittaa sitä, ettei Intelin tarjoamaa työkalua voida hyödyntää selaimessa WebAssemblyä käyttävän WebGL-sovelluksen profilointiin. VTune Profiler -työkalun voi kuitenkin yhdistää selaimeen sen prosessin tunnuksen avulla, mutta Intel Corporation (2024a) ei mainitse tukevansa WebGL-sovelluksen profilointia. Työkalun käytön aikana yhdistysvaiheessa tuen puute voidaan havaita virhekoodina ja WebGL-sovelluksen kaatumisena (kuva 1).



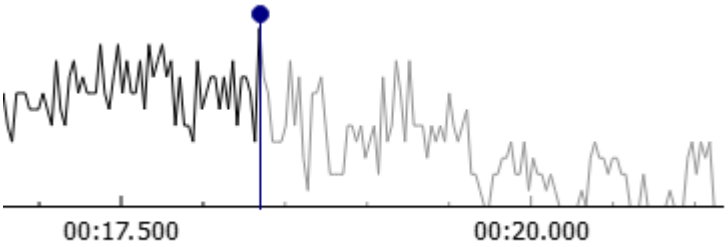
Kuva 1. Virhe prosessiin kiinnityksessä

VTune Profiler mahdollistaa Android-sovellusten profiloinnin Androidin ADB-työkalulla. Sen käyttö vaatii Wi-Fi-yhteyden tai USB-yhteyden tietokoneen ja puhelimen välille. Opinnäytetyössä käytettävä mobiililaite on vuonna 2020 julkaistu Sony Xperia 10 II -älypuhelin. Samana vuonna kirjoitetussa blogipostauksessaan Whaley (2020) kertoo, että lähes 90 prosenttia Android-laitteista käyttää 64-bittistä arkkitehtuuria tukevaa versiota käyttöliittymästä. Whaley (2020) myös toteaa, että laitemarkkinoille todennäköisesti ilmestyy 2023 vuonna tuotteita, jotka eivät tue 32-bittistä arkkitehtuuria enää ollenkaan. Testauslaite käyttää 64-bittistä arkkitehtuuria, joka tulee ilmi myös VTune Profiler -työkalun käytön yhteydessä. Työkalu ei tue 64-bittistä arkkitehtuuria, jonka myötä suurinta osaa nykyaikaisista laitteista ei voida hyödyntää työkalulla profilointiin (kuva 2).



Kuva 2. VTune Profiler Android-tuen puute

AMD uProf -profilointityökalun käyttämisessä törmättiin samoihin ongelmiin. Advanced Micro Devices (2024) ei listaa tukemissaan alustoissa Androidia tai WebGL-alustaa ja AMD uProf on suunniteltu analysoimaan sovelluksia, joita ajetaan Windows- ja Linux-käyttöjärjestelmillä. uProf sekä VTune Profiler voidaan halutessa kytkeä suoraan Unity Editor -prosessiin, mutta tuloksien analysointi on haastavaa. Prosessissa havaitut piikit olivat selkeästi kohdennettavissa ja tarkasteltavissa (kuva 3), eikä niiden havaitseminen ollut hankalaa.



Kuva 3. Piikki suorituksessa uProf-työkalussa

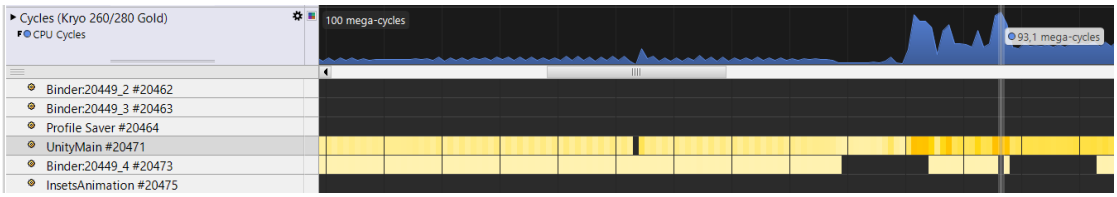
Tuloksia (kuva 4) tarkastelemalla piikin kohdalta ei kuitenkaan voinut suoraan paikantaa, mikä osa-alue sovelluksessa, tarkemmin Unity Editor -prosessissa, oli aiheuttanut piikin. Ei ollut myöskään selkeää, oliko tämä piikki peräisin Unity Editor -prosessin toiminnasta vai profilointisovelluksen toiminnasta, jonka kohteena unity.exe on. Tuloksista oli helppo tehdä vääriä johtopäätöksiä, ellei omannut kattavaa ymmärrystä tietokoneiden ja käytettävien ohjelmistojen toiminnasta. Ongelmallista on myös se, ettei sovelluksen profilointi editorissa koskaan vastaa todellista koontiversion toimintaa ja niiden välillä voi olla suuriakin eroja ero (Unity Technologies 2022).

Functions	Modules	CPU_TIME(s) ▼
ThreadedStreamBuffer::HandleOutOfBufferToReadFrom({1})	unity.exe	1.2440
PID-30924 : unity.exe		1.3300
Thread-10512		1.3270
Thread-18292		0.0030

Kuva 4. Eniten aikaa vienyt prosessi piikin aikana uProf-työkalussa

Android-laitteella sovelluksen profilointia voidaan suorittaa kolmannen osapuolen työkalujen kautta. Näitä ovat esimerkiksi Android GPU Inspector, Arm Mobile Studio (nykyisin Arm Performance Studio), PVRTune ja Snapdragon Profiler (Unity Technologies 2025e). Näistä työkaluista yhteensopivia käytettävän mobiililaitteen kanssa oli ainoastaan Arm Performance Studio. Tuettujen laitteiden listassa (Android Developers 2023) ei Android GPU Inspector -työkalulle listattu käytettävän mobiililaitteen mallia tai komponentteja. Tämä sama päti PVRTune ja Snapdragon Profiler -työkalujen kohdalla, sillä ne mahdollistivat profiloinnin omiin laitteistoihin, jotka eivät olleet käytettävän mobiililaitteen mukaisesti ARM-arkkitehtuuria hyödyntäviä laitteita.

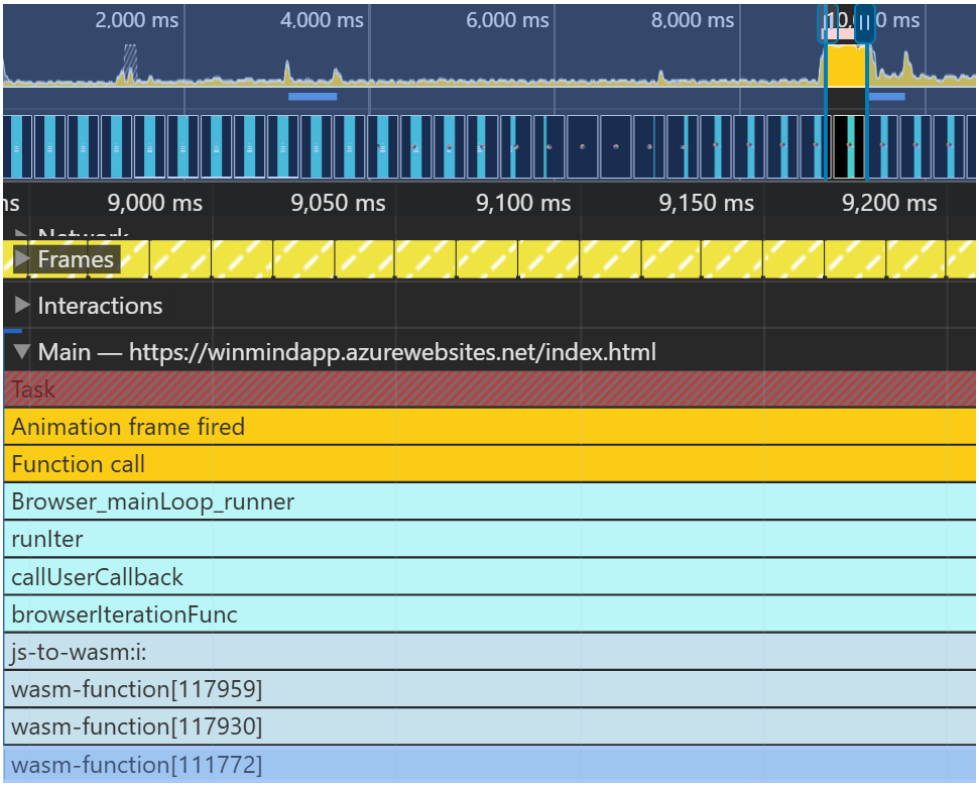
Arm Performance Studio tarjoaa työkalukokonaisuuden, jonka avulla voidaan profiloida mobiilisovelluksen toimintaa. Android-sovelluksen profilointiin voidaan käyttää Streamline-työkalua. Streamline hyödyntää ADB-työkalua yhdistääkseen mobiililaitteeseen ja ottamaan sovelluksen tarkastelukohteeksi. Streamline-työkalun avulla voitiin havaita piikki kuten toisillakin sovelluksilla, mutta ongelmaksi jäi taas ongelman kohdistaminen. Työkalu kertoi isoimman kuormituksen olevan UnityMain #20471 prosessilla (kuva 5), mutta tätä tarkempaa tietoa piikin alkuperästä ei annettu.



Kuva 5. Leike analyysinäkymästä Arm Performance Studion Streamline-työkalussa

Ongelmakohtien paikantamisen haasteellisuus sovelluksessa ei kuitenkaan ollut näiden profilointityökalujen syytä. Ne toimivat tarkoituksenmukaisesti ja mittasivat laitteen komponenttien toimintaa ja suorituskykyä. Samalla ne kertoivat ongelman alkuperän ja ajankohdan, mutta varsinainen paikantaminen jäi käyttäjän vastuulle. WebGL-alustaa profiloidessa törmätään samaan ongelmaan, jos käytetään esimerkiksi selainpohjaisia profilointityökaluja. Selaimet, kuten Google Chrome (Chrome) ja Mozilla Firefox (Firefox) tarjoavat kehittäjäntyökaluissaan suorituskyvyn mittaamista.

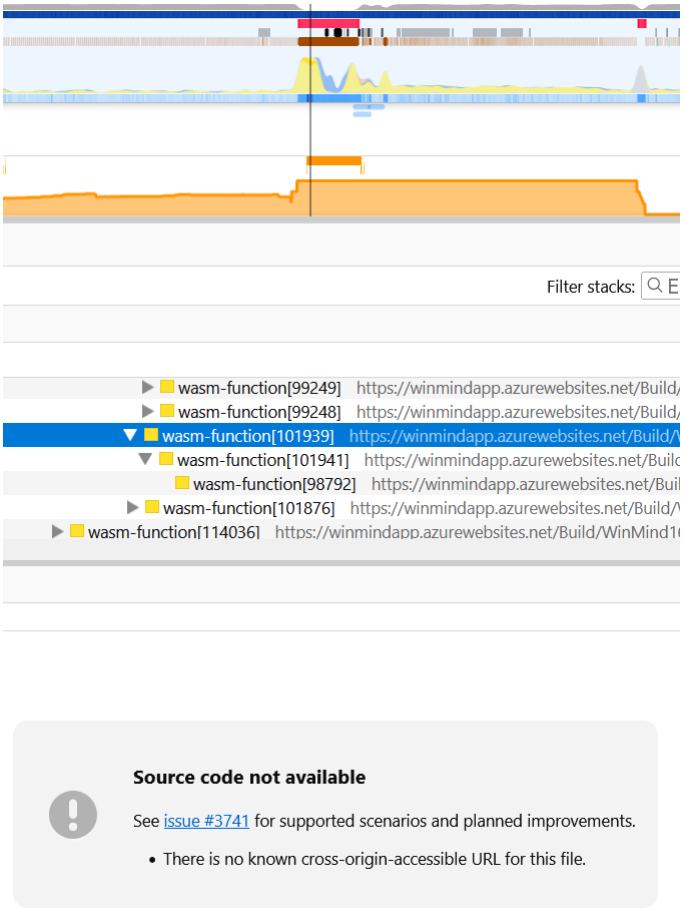
Chromen profilointinäkymässä voidaan tarkastella sovelluksen toimintaa funktio tasolla ja työkalu tarjoaa myös leikkeitä sovellusnäkymästä helpottamaan ongelmakohtien paikantamista (kuva 6). Piikin aiheuttajat ovat selkeämmin havaittavissa verrattuna aikaisempiin työkaluihin. Funktioita, joita WebGL-alustalla pyörivä sovellus kutsuu, voidaan nähdä kuvan 6 alareunassa.



Kuva 6. Leike Google Chromen Dev Tools -profilointinäkymästä

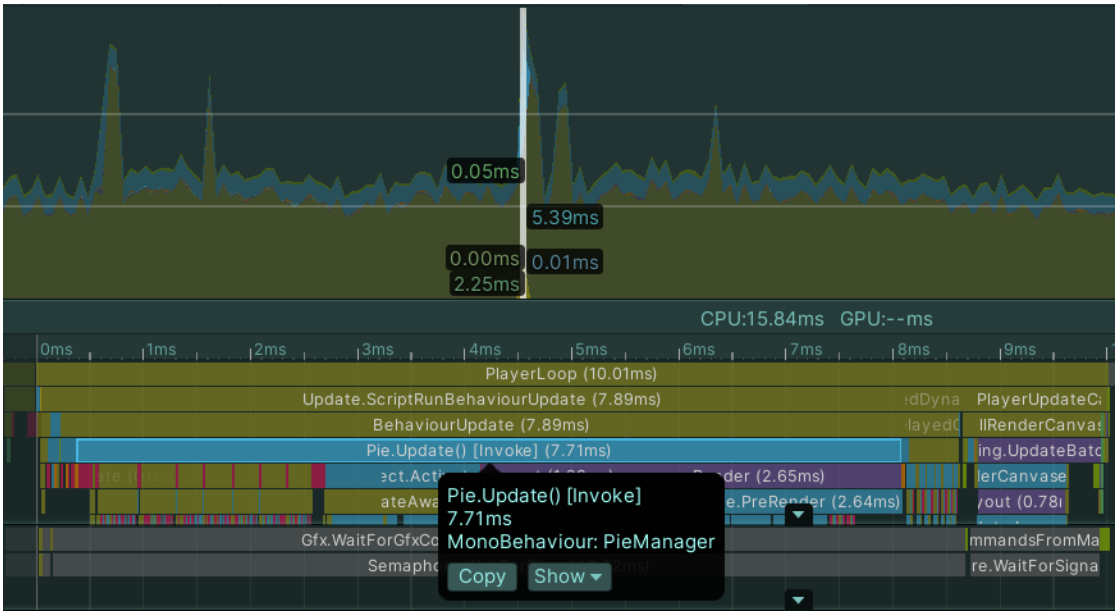
Funktiot on käännetty WebAssembly (wasm) -muotoon, mikä tekee niiden luettavuudesta selaimessa hankalaa tai joissain tapauksissa mahdotonta. WebAssembly on binäärinen ohjelmointikieli, joka mahdollistaa tehokkaan laskelmoinnin ja tilankäytön (WebAssembly Community Group s.a). Binäärinen ohjelmointikieli ei lähtökohtaisesti ole ihmisluettavaa ja se on tarkoitettu laitteiden tulkittavaksi tehokkaan laskelmoinnin saavuttamiseksi.

Firefoxin profilointityökalun näkymässä (kuva 7) voitiin havaita saman formaatin wasm-function[] -funktiokutsuja. Firefox Profiler ilmoitti vielä erikseen, ettei funktioiden lähdekoodit olleet saatavissa tarkasteltavaksi. Tässä tapauksessa piikin aiheuttaneen koodin tulkinnan yrittäminenkään ei ollut mahdollista selainpohjaisessa profiloinnissa.



Kuva 7. Leike Mozilla Firefoxin Firefox Profiler -profilointinäkymästä

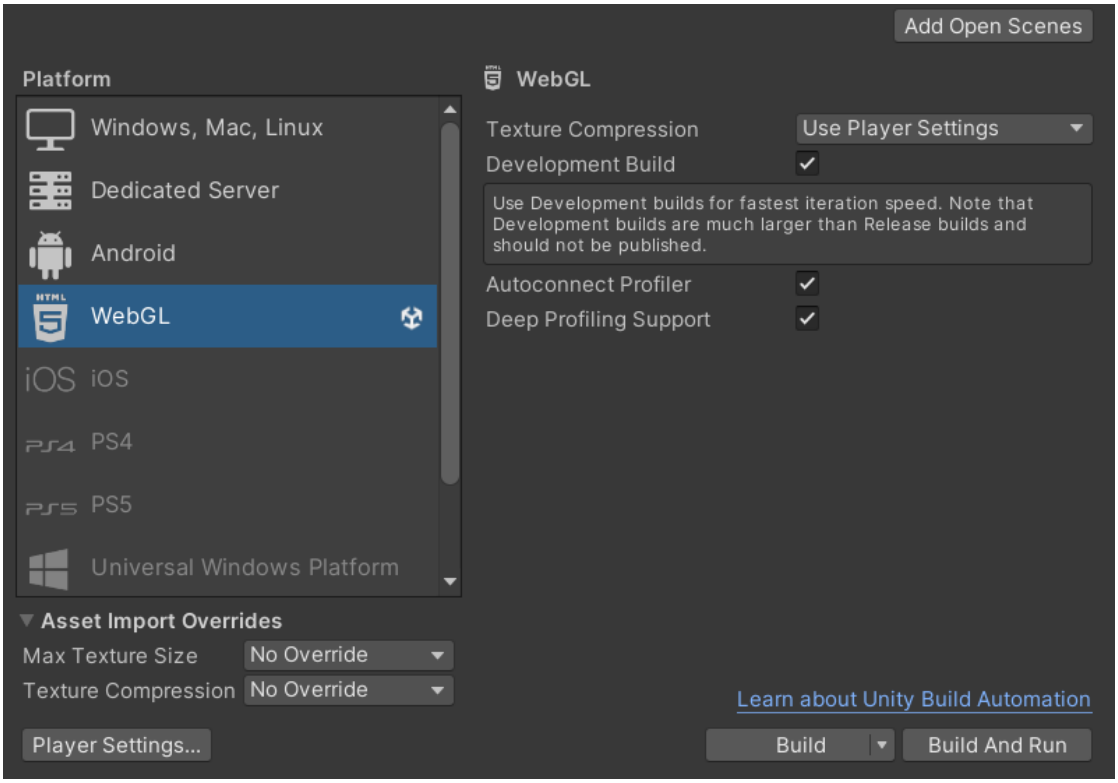
Kolmannen osapuolen profilointityökalut soveltuvat parhaiten laitteiston suorituskyvyn mittaamiseen ja keskittyvät etenkin työkalun tarjoajan kehittämiin laitteistoihin. Unityn omat työkalut ovat käytettävissä Unity Editor -kehitysympäristössä sisäänrakennetusti. Unityn Profiler-työkalun käyttöliittymä muistuttaa aiemmin läpikäytyjen työkalujen käyttöliittymiä. Muodostuneen käyrän avulla voidaan tarkastella piikkejä suorituskyvyssä ja kohdennetussa tarkastelussa voidaan havaita sen aiheuttavat tekijät (kuva 8).



Kuva 8. Leike Unityn Profiler -näkymästä

Toisin kuin kolmannen osapuolen työkalut, Unity Profiler on sisäänrakennettu työkalu, joka pystyy keräämään dataa sovelluksen toiminnasta syvemmällä tasolla. Kuvassa 8 kuvataan piikin alapuolella sen aiheuttaneet tekijät ja voidaan havaita Pie.Update() funktion olleen suurin suoritustehoa vaativa tapahtuma. Profilointityökalun tulkintaan perehdytään myöhemmin luvussa profilointi ja optimoinnin kohdennus, mutta tässä vaiheessa tärkeä havainto oli, että työkalu tunnisti ongelmallisen komentosarjan, eli skriptin, sekä sen sisältämän funktion nimeltä. Tämä helpotti profilointia merkittävästi ja auttoi kohdentamaan optimoinnit niitä tarvitseviin osa-alueisiin ilman ylimääräistä työtä.

Unityn Profiler ei myöskään vaatinut yhtä paljon toimenpiteitä käyttöä varten. Itse editorissa, profilointi alkaa käyttäjän laittaessa Play Mode -moodin päälle samalla kun Profiler-ikkuna on auki. Koontiversioita voidaan profiloida myös suoraan Unity Editor -kehitysympäristön kautta, jos määritellään koottava sovellus kehitysversioksi ja asetetaan Profiler kiinnittymään automaattisesti koottuun sovellukseen (kuva 9).



Kuva 9. Unity Editorin koontiasetukset WebGL-alustalle

Unity pystyi myös kiinnittämään profilointityökalunsa WebGL-sovellukseen, joka tuotti ongelmia muissa työkaluissa. Android-sovellukseen voidaan kiinnittyä ADB-työkalun avulla ja tulokset voidaan nähdä Profiler-näkymästä. Unity Profiler -työkalun käyttö muistuttaa enemmän Plug and Play tyylistä palvelua, jossa käyttäjän ei tarvitse asentaa ylimääräisiä ohjelmia tai määrittää asetuksia käyttöä varten.

Muissa sovelluksissa vaadittiin käyttäjää muun muassa määrittämään polku adb.exe tiedostoon Android-profilointia varten tai syöttämään tarkasteltavan prosessin tunnus, jonka voi löytää tehtävienhallinnan Lisätiedot-osiosta. Opinnäytetyön kohdesovelluksen profilointi oli haastavaa kolmannen osapuolen työkaluilla kohdealustojensa vuoksi. Työkalujen käytössä oli paljon vaatimuksia, jotka eivät välttämättä toteudu ja niidenkin työkalujen kohdalla, joita voitiin käyttää, saadut tulokset eivät ohjanneet tarkasti ongelmakohtiin.

Unityn omien profilointityökalujen saavutettavuus niiden helpomman käytön ja sovelluskohtaisen analyysin perusteella kannusti käyttämään niitä kolmannen osapuolen työkalujen sijaan. Jotta testaus ja profilointi pysyi johdonmukaisena ja helposti toistettavana, valittiin opinnäytetyössä käytettäväksi Unityn sisäänrakennetut profilointityökalut.

5 PROFILOINTI JA OPTIMOINNIN KOHDENNUS

Sovelluksen profilointi on tärkeä osa sovelluksen kehitysprosessia. Aikaisin aloitettu profilointi auttaa kehittäjiä ymmärtämään sovelluksen toimintaa ja muodostaa kuvan odotetusta suorituskyvystä. Tämän tiedon avulla voidaan kohdistaa profilointia tarkemmin ja välttää ennenaikaisen optimoinnin suorittamista.

5.1 Kuvabudjetin määrittäminen

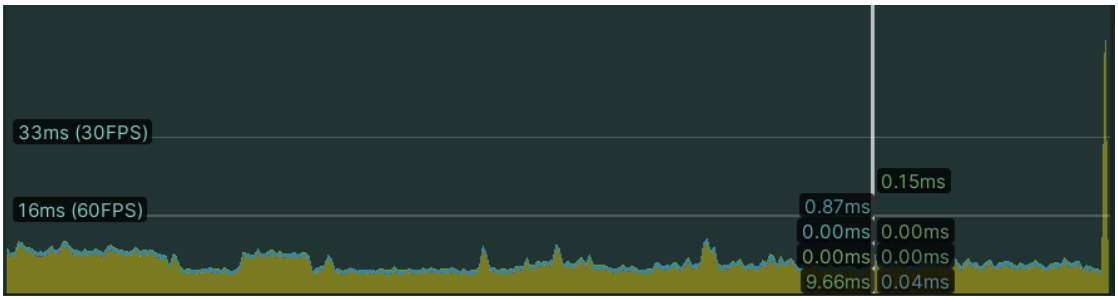
Sovellusta profiloitaessa on määritettävä ensin kuvabudjetti, jonka avulla voidaan havaita ongelmakohtia sovelluksen toiminnassa. Kuvabudjetti (frame budget) määritetään kuvan keston (frame time) perusteella. Kuvan kesto on ajan pituus, joka jokaisen kuvan (frame) luomisessa kestää, ja se voidaan laskea jakamalla yksi sekunti kuvataajuuden suuruudella. Laskutoimituksessa on suositeltavaa muuntaa sekunnit millisekunneiksi kuvan keston luettavuuden saavuttamiseksi. Taulukossa 1 esitetään kolmen yleisen kuvataajuuden laskettu kuvan kesto. Jos sovelluksen kuvataajuudeksi halutaan saavuttaa 60 kuvaa sekunnissa (FPS), voi yhden kuvan kesto olla maksimissaan 16.666 millisekuntia. Jos kuvan kesto ylittää arvon, on kuvabudjetti ylitetty ja voidaan alkaa selvittämään, mistä ylitys johtui.

Taulukko 1. Kuvan kesto eri kuvataajuuksilla

Kohde kuvataajuus	Laskutoimitus	Kuvan kesto
30 FPS	1000ms/s / 30 FPS	33.333ms
60 FPS	1000ms/s / 60 FPS	16.666ms
144 FPS	1000ms/s / 144 FPS	6.944ms

Kuvan kesto toimii parempana suorituskyvyn mittarina kuin kuvataajuus, sillä keskiarvollisen kuvataajuuden tasaisuus ei suoraan tarkoita kuvan keston tasaisuutta. Keskiarvollisesti 60 kuvaa sekunnissa kuulostaa hyvältä, mutta jos yhden kuvan kesto on ollut 0.25 sekuntia ja loppujen 59 kuvan kesto 0.75 sekuntia, käyttäjä voi havaita pätkimistä näytöllään, vaikka sovelluksen metriikka näyttäisi tasaisen 60 kuvaa sekunnissa. (Unity Technologies 2022.)

Unityn Profiler -työkalussa kuvan kestoa mitataan käyrällä ja kuvabudjetti on visualisoitu näkymään poikkiviivalla (kuva 10). Näkymästä voidaan havainnoida sovelluksen prosessien kestoa ja jokainen piikki tarkoittaa pitempää kuvan kestoa, jonka prosessien suorittaminen vaatii. Kuvasta 10 voidaan havaita oikeassa laidassa piikki, joka selkeästi ylitti 33ms kuvan keston rajan. Tämä tarkoitti sitä, ettei sovellus ylläpitänyt tasaisesti vähintään 30 kuvaa sekunnissa. Esimerkiksi mobiililaitteilla 30 kuvaa sekunnissa on heikomman laskentatehon vuoksi usein käytetty kuvataajuus.



Kuva 10. Prosessorin käytön näkymä Unityn Profiler-työkalussa

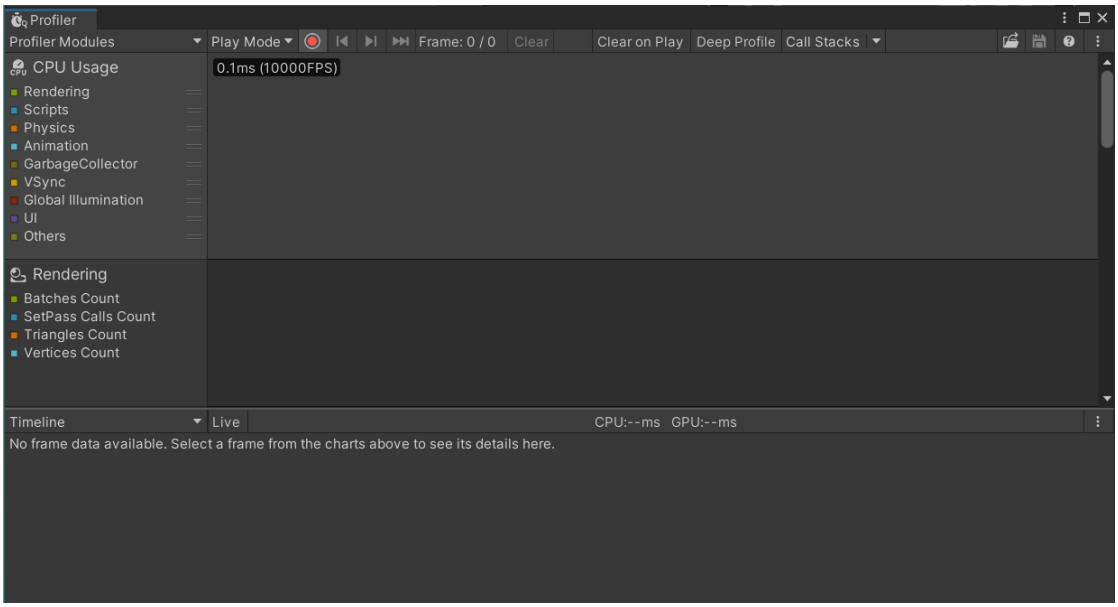
Mobiililaitteilla vaativiin sovelluksiin on myös suositeltavaa jättää noin 35 prosenttia kuvataajuudesta toimettomaksi laitteen lämmönsäätelyn vuoksi. Todellinen kuvan kesto voi mobiililaitteilla olla lähempänä 21.66ms (33.333 * 0.65 = 21.66ms), jos kuvataajuudeksi halutaan 30 kuvaa sekunnissa. Vastaavasti 60 kuvaa sekunnissa vaatii kuvan kestoksi enintään 10.83ms. Tämä on useille mobiililaitteille vaikeasti saavutettavissa ja samaan aikaan se kaksinkertaistaisi laitteen akun kulutuksen. Näiden syiden vuoksi 30 kuvaa sekunnissa on yleisemmin käytetty kuvataajuus mobiililaitteilla. (Unity Technologies 2022.)

Työn kohteena toimivan sovelluksen kuvabudjetiksi määritettiin Android-alustalle 33ms (30FPS) ja WebGL-alustalle 16ms (60FPS). Sovellusta voitiin myös profiloida Unity Editorissa kehitysvaiheessa, vaikka se ei vastannut todellista suorituskykyä. Havaittavat piikit kertoivat potentiaalisista optimoinnin kohteista ilman koontiversion luontia.

5.2 Profiler-työkalun käyttö

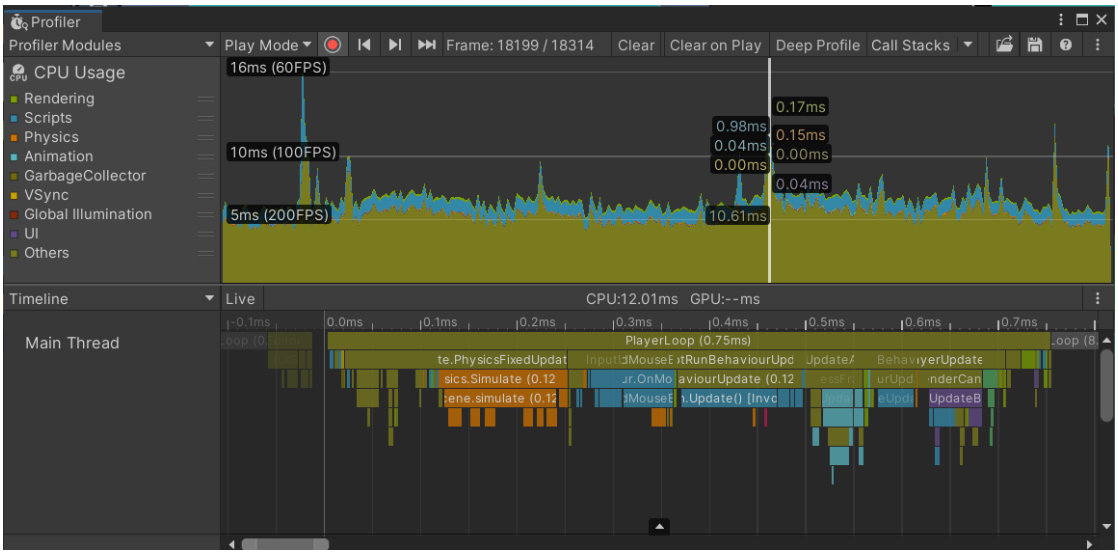
Yksinkertaisin tapa avata Unityn Profiler-työkalu on editorissa painaa Ctrl + 7 näppäinyhdistelmää. Profilerin voi avata myös editorin yläreunassa sijaitsevasta valikkopalkista kohdasta Window > Analysis > Profiler. Analysis-valikon

alta löytyy muutkin profilointityökalut kuten Frame Debugger ja Profile Analyzer. Käyttäjälle aukeaa kuvan 11 mukainen ikkuna.



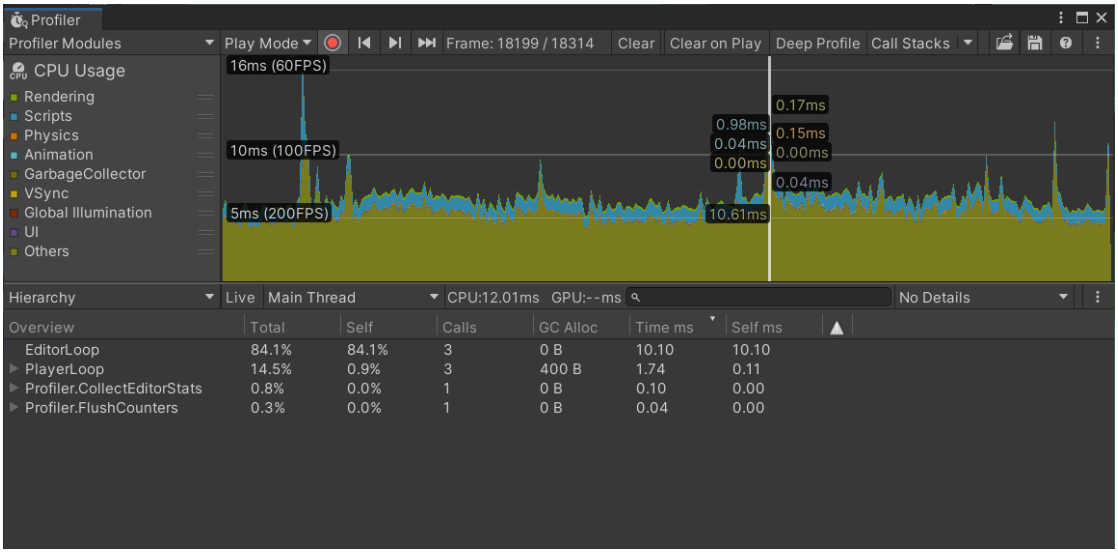
Kuva 11. Unity Profiler -näkymä

Ikkunan vasemmasta laidasta löytyy Profiler-moduulit, joita asettamalla aktiiviseksi tai inaktiivisiksi voidaan tarkastella sovelluksen toimintaa eri osa-alueilla kuten suorittimen toimintaa tai muistinhallintaa. Sovelluksen profiloinnin aikana suorittimen käytön (CPU Usage) moduulin kohdalle piirtyy suorituskykyä mittaava käyrä (kuva 12). Suorittimen käytön moduulista voidaan havaita enemmistö ongelmista ja muita moduuleja ja työkaluja voidaan hyödyntää tarvittaessa tarkempaan virheiden kohdentamiseen.



Kuva 12. Aikajanallinen Unity Profiler -näkymä profiloinnin aikana

Toinen tärkeä osa Profiler-työkalua on kuvan 12 alaosassa näkyvä aikajana (timeline). Aikajanalle on kuvattu visuaalisesti kuvan keston kooste. Palkit kuvastavat prosessien kestoa ja niiden alapuolella on eritelty sisempiä prosesseja, joista ylemmän prosessin kesto koostuu. Näkymä voidaan vaihtaa myös hierarkkiseen näkymään (kuva 13), jolloin tarkasteltava kuvan kesto on esitetty numeerisesti. Hierarkkinen näkymä soveltuu tarkempaan analysointiin, kun taas aikajana helpottaa ongelmakohtien ensimmäistä havainnointia.



Kuva 13. Hierarkkinen Unity Profiler -näkymä profiloinnin aikana

Profiler-työkalun toiminta perustuu instrumentointiin, jossa Unityn käyttämiin rajapintoihin on sijoitettu toimintojen kestoa mittaavia Profiler-markkereita (Unity Technologies 2022). Toimintojen kestot yhteen laskemalla voidaan muodostaa kuvan kesto, johon vertaamalla yksittäisten toimintojen kestoa voidaan määrittää suurimman rasitteen lähde. Profiler-työkalun Deep Profile -asetus asettaa Profiler-markkerin sovelluksen jokaisen funktion alkuun ja loppuun. Tämä kuitenkin kasvattaa yleisrasitteen (overhead) määrää ja profiloinnin tulokset eivät vastaa todellista suorituskykyä. Itse Unity Editor kasvattaa myös yleisrasitetta.

Kuvan 13 alaosassa on nähtävissä kooste kuvan kestosta, jossa EditorLoop on vienyt 84,1 prosenttia (10.10ms) kokonaisajasta, kun taas sovelluksen toiminnasta vastaava PlayerLoop on käyttänyt 14.5 prosenttia (1.74ms). EditorLoop on olemassa pelkästään Unity Editor -ympäristössä ja sen käyttämät resurssit ovat erillisiä itse sovelluksesta. Tästä syystä on suositeltavaa profiloida koontiversiota lopullisten tuloksien keräämiseksi. Suurimpien suorituskykyä

vaativien kohteiden paikantaminen on mahdollista ilman koontiversiota, jolloin on otettava huomioon editoriversiosta aiheutuva yleisrasite.

Profiler-työkalun näkymä visualisoi asetuksien mukaan viimeiset 300 kuvaa tai 2000 kuvaa. Kuvan määrän (frame count) asetusta säätäessä, työkalu varoittaa suuremman määrän potentiaalisesti aiheuttavan yleisrasitetta. Pienempi määrä visualisoituja kuvia voi aiheuttaa sen, että mahdollinen piikki jää huomaamatta. Tätä varten voidaan kirjoittaa avustava skripti, joka pysäyttää applikaation kuvabudjetin ylittyessä (kuva 14).

```
1  using UnityEngine;
2  using UnityEditor;
3  using UnityEngine.Profiling;
4
   Unity Script (1 asset reference) | 0 references
5  public class BudgetChecker : MonoBehaviour
6  {
7      public float kuvaBudjetti = 0.033f; // 33ms, eli 30fps
8
9  #if UNITY_EDITOR
       Unity Message | 0 references
10     void Update()
11     {
12         if (Time.deltaTime > kuvaBudjetti) /* kuvan kesto ylittää budjetin */
13         {
14             EditorApplication.isPaused = true;
15             Profiler.enabled = false;
16         }
17     }
18 #endif
19 }
```

Kuva 14. BudgetCheker-skripti

Skriptin avulla Profiler-työkalun ikkunaa ei tarvinnut jatkuvasti vahtia ja mahdolliset piikit eivät voineet jäädä huomaamatta. Muuttujaa kuvaBudjetti säätämällä voidaan määrittää kynnys pysäytykselle. Ensimmäisten profilointikertojen kanssa kynnys voidaan asettaa korkeammaksi, jotta voidaan havaita kaikista suurimmat suorituskykyä vaativat kohdat. Skriptin käytössä ongelmaksi voi muodostua profiloinnin yhteydessä turhautuminen jatkuviin pysäytyksiin, jos esimerkiksi EditorLoop-prosessin aiheuttama yleisrasite saa kuvabudjetin ylittymään jatkuvasti.

5.3 Profiloinnin aloitus

Sovellukselle suoritettiin profilointi Unity Profiler -työkalulla, jonka aikana hyödynnettiin BudgetChecker-skriptiä. Ensimmäinen profilointi suoritettiin kuvabudjetilla 40ms (25fps) ilman Deep Profile-asetusta editoriversiossa. Profiloinnin aikana pidettiin kirjaa budjetin ylityksistä ja eriteltiin taulukkoon 2, johtuiko ylitys EditorLoop-prosessista vai jostain muusta. Jos PlayerLoop ylitti budjetin yksin, ei EditorLoop-prosessia otettu siinä tapauksessa huomioon.

Taulukko 2. Kuvabudjetin ylitykset

	PlayerLoop	EditorLoop	Garbage-Collector	Profiler.Flush-Counter	Yhteensä
Budjetin ylitykset	10	11	10	2	33

Profiloinnin aikana generoitiin 103255 kuvaa ja keskiarvollinen kuvan kesto oli noin 6ms sovelluksen toimiessa kuvabudjetin rajoissa. Sovellus oli toiminnassa noin 8.7 minuuttia ottaen piikit huomioon. Vaikka profilointia varten määritettiin normaalia korkeampi kuvabudjetti, kuvan kesto ylitti sen 20 kertaa toiminnoissa, jotka tapahtuisivat koontiversiossakin.

Huomio tuloksissa kiinnittyi erityisesti GarbageCollector-prosessin yleisyyteen. Unity Engine -pelimoottorissa muistinhallinta tapahtuu automaattisesti sen C#-ohjelmointikielen myötä ja GarbageCollector on osa tätä prosessia. Lukuisien GarbageCollector-prosessien läsnäolo kielii huonoista käytännöistä objektien hallinnassa, sillä muistia on jouduttu vapauttamaan useasti. Unity käyttää pelimoottorissaan Boehm-Demers-Weiser-kerääjää inkrementaalisesti, tarkoittaen sitä, että muistinsiivous on jaettu usealle eri kuvalle (frame) ja sen suorittamisen aikana pelin logiikkaa ei suoriteta. (Unity Technologies 2025c).

Pelin logiikkaa ei suoriteta muistinsiivouksen aikana, joten useasti tapahtuva siivous aiheuttaa pätkimistä ja ylimääräisten siivousprosessien myötä myös virrankulutus kasvaa. Optimaalisesti sovelluksen suorituksen aikana ei pitäisi tapahtua muistinsiivousta, ellei sitä manuaalisesti itse haluta suorittaa esimerkiksi latausruutujen aikana.

WebGL- ja Android-alustalle kootuista versioista voitiin havaita piikkejä tismalleen samoista osa-alueista kuin Unity Editor -ympäristössä. Lähtökohtaisesti koontiversiot toimivat kuvabudjettiensa rajoissa, eli 16ms WebGL-alustalla ja 33ms Android-alustalla, mutta piikkejä oli havaittavissa. Sovelluksessa eniten resursseja käytti tekstien generointi sekä uusien peliobjektien käyttöön ottaminen.

Tekstien generoinnin vaativuus johtui sovelluksessa käytettävästä TextMesh Pro -komponentista, jonka Auto Size -asetus kasvatti kuvan kestoa merkittävästi. Asetus määrittää tekstille automaattisesti oikean fonttikoon, jotta se mahtuisi rajatun tekstikentän sisään. Peliobjektien käyttöön ottaminen ensimmäisen kerran aiheuttaa piikin, varsinkin jos niiden alla on lukuisia lapsiobjekteja. Ensimmäisen kerran jälkeen objektien Awake- ja Start-funktioita ei enää suoriteta, mutta on tärkeää ottaa huomioon, että kun hierarkiassa korkeampi objekti asetetaan aktiiviseksi, myös sen lapsiobjektien OnEnable-funktiot suoritetaan.

Suurin osa piikkejä aiheuttavista tapahtumista sijoittui sovelluksen käytössä sellaisiin ajankohtiin, joissa oltiin siirtymässä näkymästä toiseen. Tämä tekee piikeistä huomaamattomampia loppukäyttäjille, mutta suorituskyvyn vaatimusten noustessa vanhemmat laitteet eivät välttämättä pysty tuottamaan yhtä sulavaa käyttökokemusta.

Työn kohteena olevan sovelluksen tapauksessa voitaisiin keskittyä profilointiin Unity Editor -ympäristössä, sillä havaittavat piikit ilmenivät kaikissa versioissa. Sovelluksen yksinkertaisuus myös mahdollisti helpon toistettavuuden ilman automaatiotestausta. On kuitenkin otettava huomioon Unity Editor -ympäristön profiloinnissa muodostuvat kutsut, joita ei koontiversioissa ole olemassa.

Nämä kutsut voivat ylittää asetetun kuvabudjetin ilman, että ne olisivat jäljitettävissä EditorLoop-prosessiin. Tämän vuoksi lopullinen profilointi olisi suotavaa suorittaa koontiversiossa. Työssä suoritettava profilointi optimoinnin kohdentamista varten suoritettiin Android-alustalla. Kohdealustan kuvabudjetti oli 33ms, joten havaittavat ongelmat olisivat tulleet eteen myös WebGL-alustalla, jonka kuvabudjetti oli korkeampi.

5.4 Optimoinnin kohteet

Android-alustalla suoritetussa profiloinnissa havaittiin piikkejä tavallisen käytön yhteydessä sekä muutamia olosuhteista riippuvia piikkejä. Kuvan kestoa lisäsi kaikista eniten tekstien luonti TextMeshPro-tekstikomponenteilla, joissa käytettiin Auto Size -asetusta. Aina kun näkymä päivitetään, Auto Size säätää tekstin kokoa ja asettelua. Tämä fonttikoon pieneneminen ja suureneminen saa aikaan Canvas-komponentin uudelleenpiirtymisen ja luo siten yleisrasitetta. Peliobjekti, johon Canvas-komponentti on kiinnitetty, pitää allaan lapsiobjekteina käyttöliittymän elementit. Ne kaikki päivitetään, jos uudelleenpiirtyminen tapahtuu.

Sovelluksessa TextMeshPro-tekstikomponentteja käytetään kaikkiin teksteihin. Niiden aikaansaamia piikkejä havaittiin eniten näkymien vaihtumisen ja kysymyslaatikoiden luomisen yhteydessä. Koodin puolella piikkejä aiheutti myös joidenkin skriptien OnEnable- ja OnDisable-funktiot, joita kutsuttiin ylimääräisiä kertoja.

Graafisesti sovellus ei ole vaativa. Sovelluksen kehityksessä on käytetty URP-renderöintitekniikkaa, eli yleistä renderöintiliukuhihnaa (Universal Render Pipeline). URP-renderöintitekniikkaa on käytetty sovelluksessa tulevaisuuden turvaamiseksi; omien renderöintiominaisuuksien kehittäminen sovelluksen tarpeisiin on mahdollista URP-renderöintitekniikalla. Omia renderöintiominaisuuksia projektiin ei toistaiseksi ole vielä kehitetty eikä niitä ole tarvittu.

Käyttöliittymä on kevyt ja ruudulla on näkyvissä kerralla yleensä alle 15 käyttöliittymäkomponenttia. Yleisesti piirtokutsujen (draw call) määrä sovelluksessa oli alle 30 kappaletta, mutta minipeleissä määrä saattoi kasvaa merkittävästi. Graafisen suorituskyvyn piikkejä oli havaittavissa ainoastaan minigolf-minipelissä. Näkymän kuvasynteesissä, eli renderöinnissä, voitiin havaita lukuisia säännöllisiä piikkejä. Minigolf-minipeli on sovelluksen graafisesti vaativin osio ja sisältää suurimman määrän peliobjekteja.

Piirtokutsujen määrä nousi 200 kappaleeseen, kun kaikki väylät olivat näkyvissä minipelin alkunäkymässä. Muissa minipeleissä luku oli suurimmillaan 52. Osa minigolf-minipelin peliobjekteista on myös vuorovaikutuksessa vedessä

käytettävän varjostimen (shader) kanssa, mikä vaikuttaa sovelluksen graafiseen suorituskykyyn.

Sovelluksessa käytetään suhteellisen kevyitä varjostimia: TextMeshPro -tekstikomponenteissa on käytössä distance field -varjostin, jolla varmistetaan tekstin ulkonäön hyvä laatu eri näytöillä resoluutiosta riippumatta. Itse tehtyjä varjostimia on käytössä yksi ja sillä säädetään kuvan harmaasävyjä. Varjostimella asetetaan pieni kuva mustavalkoiseksi, kun minipeli poistetaan pelivalikoimasta. Minigolfissa käytettävä vesivarjostin on pintavarjostin (surface shader) ja peliobjekteissa on käytetty URP/Lit -varjostimia. Projektissa ei ole myöskään käytetty jälkikäsittelyä (post-processing), jolla voisi olla vaikutusta laskelmoinnin tehokkuuteen.

Sovelluksessa ei ole käytetty monimutkaisia valaistuksia, jotka lisäisivät varjostimien laskelmointien määrää. Sovelluksessa käytettävä valaistus on projektin luonnin yhteydessä automaattisesti luotu valaistus, joka on suunnattu valo (directional light). Valaistuksia ei ole myöskään laskelmoitu etukäteen valaistuskarttaan (lightmap), sillä useimmat ruudulla nähtävistä 3D-malleista ovat dynaamisia eikä valaistuskarttaa voi niissä hyödyntää.

Sovellus ei ole sisällöltään kovinkaan laaja, joten optimoinnin kohteiden löytäminen oli suoraviivaista. Potentiaalisia optimoinnin kohteita oli useita, mutta niiden toteutuksesta saatu hyöty olisi jäänyt vähäiseksi. Esimerkiksi kun minipeliasetuksia vaihdetaan, käyttäjäkohtaiset asetukset lähetetään tietokantaan talteen.

Tämä prosessi tapahtuu vain silloin, kun käyttäjä haluaa muuttaa minipelivalikoimaansa. Kuvan kesto tässä prosessissa saattoi ylittää budjetin seitsemällä millisekunnilla, jos asetuksia oli muutettu paljon. Budjetin ylitys ei kuitenkaan ollut aina havaittavissa.

Tämä mahdollinen piikki tapahtuu näkymänvaihdon yhteydessä ja sitä ei käytön aikana huomaa. Minipeliasetukset ovat käyttäjäkohtaisia, joten ensimmäisen kerran jälkeen niitä ei välttämättä muokata enää ollenkaan. Hyötyjen vähäisyys auttoi karsimaan tärkeimmät optimointien kohteet, joissa kuvabudjetin ylitykset olivat yleisiä ja paljon korkeampia. Nämä optimoinnin kohteet olivat:

- TextMeshPro-tekstikomponenttien käyttö
- Kysymyslaatikoiden luominen
- Peliobjektien enablointi sekä disablointi
- Minigolf-minipeli

Optimoinnin kohteiden rajaaminen mahdollisti keskittymisen niihin osa-alueisiin, joissa sovelluksen toiminnan kannalta välttämättömät toiminnot tapahtuivat. Se ei kuitenkaan tarkoita sitä, etteikö sovelluksessa olisi ollut muuta optimoitavaa näiden kohteiden lisäksi. Valittujen kohteiden vaikutukset sovelluksen toimintaan arvioitiin suurimmiksi ja niille suoritetut korjaukset olivat toteutettavissa opinnäytetyön aikataulussa.

6 OPTIMOINNIN TOTEUTUS

Optimointia voidaan toteuttaa monella eri tavalla. Korjaavat toimenpiteet voivat olla esimerkiksi puhtaasti paremman koodin kirjoittamista, asetusten vaihtamista, uusien systeemien kehitystä, turhien toiminnallisuuksien poistamista tai 3D-mallien parantelua. Lopputuloksena on kuitenkin tarkoitus saada suorituskyvyltään tehokkaampi ja paremmin toimiva sovellus. Optimoinnin onnistumista voidaan mitata Unity Profiler -työkalussa esiintyvien piikkien vähentymisellä.

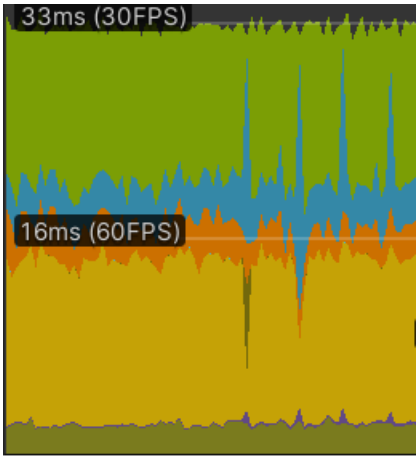
6.1 TextMeshPro

Optimointi aloitettiin TextMeshPro-tekstikomponenteista. Niiden yleisyys sovelluksessa vaikutti kaikkien osa-alueiden toimintaan ja potentiaalisten optimointien vaikutus oli suurin. Yksi vaihtoehto oli lopettaa TextMeshPro-tekstikomponenttien käyttö ja käyttää GUI Text -tekstikomponentteja sen sijaan. Tähän ratkaisuun ei kuitenkaan päädytty, sillä TextMeshPro-tekstikomponentti tarjoaa enemmän hyödyllisiä ominaisuuksia käytössään.

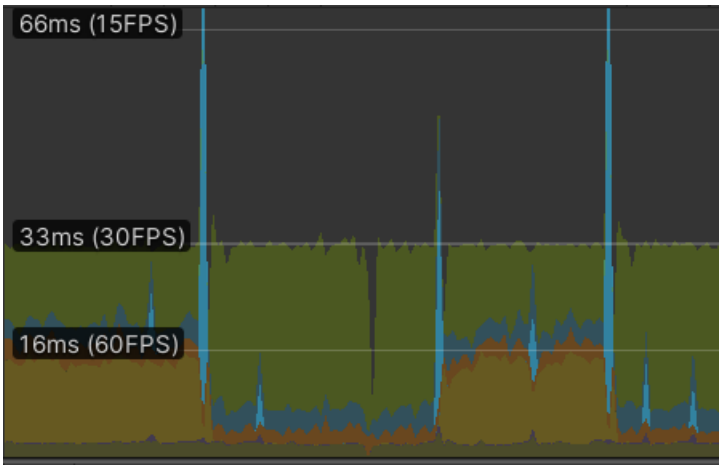
Tekstin asettelu ja ulkoasun muokkaus on kattavampaa ja komponentin käyttö mahdollistaa tulevaisuuden tarpeiden turvaamisen paremmalla tuella eri kirjaintyypeille, jos esimerkiksi halutaan käyttää sovelluksessa kieliä, jotka eivät

käytä latinalaista kirjaimistoa. Näkyvin ero tekstikomponenttien välillä oli TextMeshPro-tekstikomponenttien kirjainten parempi laatu samoilla fonttiasetuksilla. Laadun sekä luettavuuden varmistaminen oli tärkeää sovelluksen kohdealustojen näyttöjen kokoerojen vuoksi.

TextMeshPro tarjoaa työkaluja tekstin ulkoasun muokkaamiseen, mutta niiden käytössä on usein yleisrasitetta luovia toimintoja. Sovelluksen tekstikomponenteissa käytettiin oletusasetuksia ja ainoastaan Auto Size -asetus oli asetettu päälle. Ensin testattiin TestMeshPro -komponenttien asetusten muuttamisen vaikutusta. Auto Size -asetuksen pois päältä asettaminen madalsi kuvassa 16 havaittavat budjetin ylittävät piikit (kuva 15).



Kuva 15. Ohjeet-näkymän selaaminen ilman Auto Size -asetusta

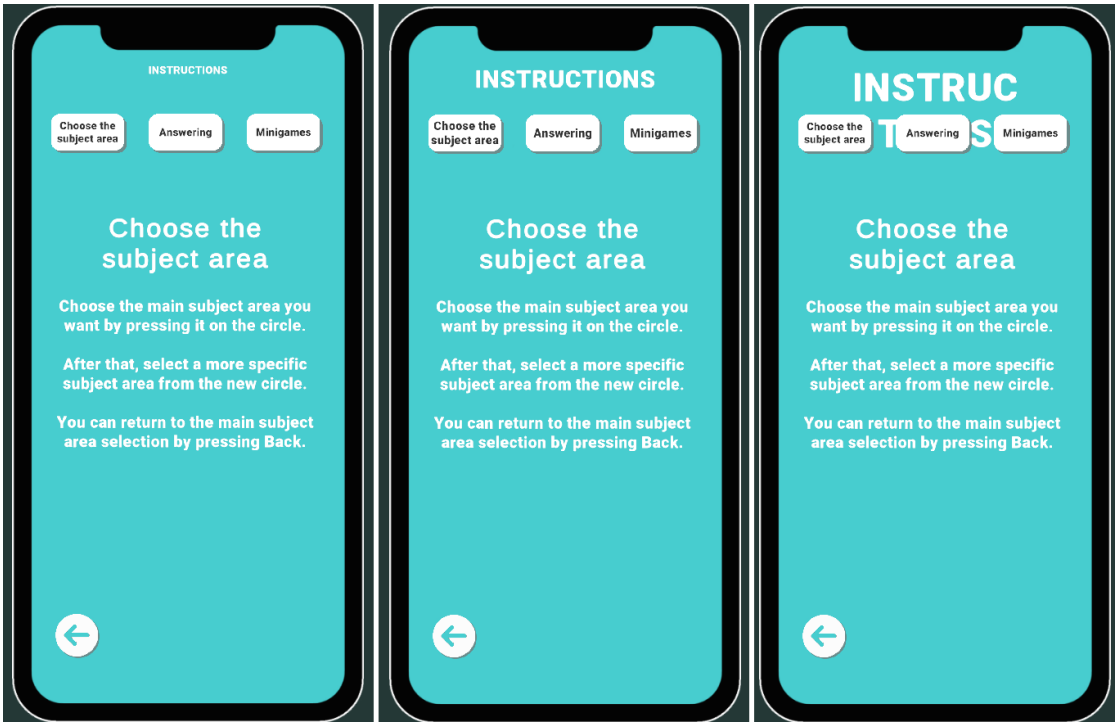


Kuva 16. Ohjeet-näkymän selaaminen Auto Size -asetuksella

Tekstin luomisesta aiheutuvat piikit olivat läsnä molemmissa tapauksissa, mutta kuvan 16 piikit ylittivät kuvabudjetin reilusti. Tekstin generointiin käyte-

tään huomattavasti vähemmän resursseja ilman automaattista tekstin sovitusta. Auto Size -asetuksen käyttämättä jättäminen siis vähentää yleisrasitetta, mutta sen käytöstä saatujen hyötyjen puuttuminen näkyy myös selkeästi.

Kuvassa 17 on kuvattu kolme eri näkymää. Vasemmassa laidassa on näkymä, jossa Auto Size -asetus on otettu pois käytöstä, keskellä on näkymä, jossa Auto Size -asetus on käytössä ja oikeassa laidassa on näkymä, jossa Auto Size -asetus on otettu pois käytöstä ja fonttikoko on säädetty niin, että suomenkielinen otsikko ”Ohjeet” on ollut oikean kokoinen.



Kuva 17. Fonttikoon muutoksia

Kuvasta 17 voidaan havaita fonttikoon skaalautuvuuden ongelmia, jotka johtuvat oletusfonttikoon määrityksestä sekä kielikäännöksistä. Sovelluksen teksteissä on luotettu kokonaan Auto Size -asetuksen asettamaan tekstikokoon, jolloin osa sovelluksen teksteistä on väärän kokoisia ilman automaattista fonttikoon muuttamista. Tämä voidaan huomata kuvan 17 vasemmassa laidassa otsikon pienenä tekstinä.

Vaihtoehtoisesti fonttikoon voi määrittää manuaalisesti, mutta oikeassa laidassa on nähtävissä, miten fonttikoko ei välttämättä tue kaikkia kieliä. Alkuperäinen otsikko ”Ohjeet” on englannin kielellä ”Instructions” ja sama fonttikoko

jakoi jälkimäisen kahdelle eri riville. Otsikko jäi osittain painikkeiden alle, mikä hankaloitti sen lukemista ja teki osiosta sekavan näköisen.

Pelkästään Auto Size -asetuksen pois päältä asettaminen ei siis ollut hyvä ratkaisu, joten tekstien skaalaus oli suoritettava itse. Skaalausta varten kirjoitettiin skripti, joka määritti tekstin fontille koon sen merkkien määrän perusteella. Ongelmaksi muodostui heti se, että tekstien merkkien leveys voi vaihdella suuresti. Esimerkiksi sanat ”hiiri” ja ”mouse” ovat toistensa käännökset ja niissä on yhtä paljon kirjaimia, mutta englanninkielinen sana on lähes kaksi kertaa leveämpi.

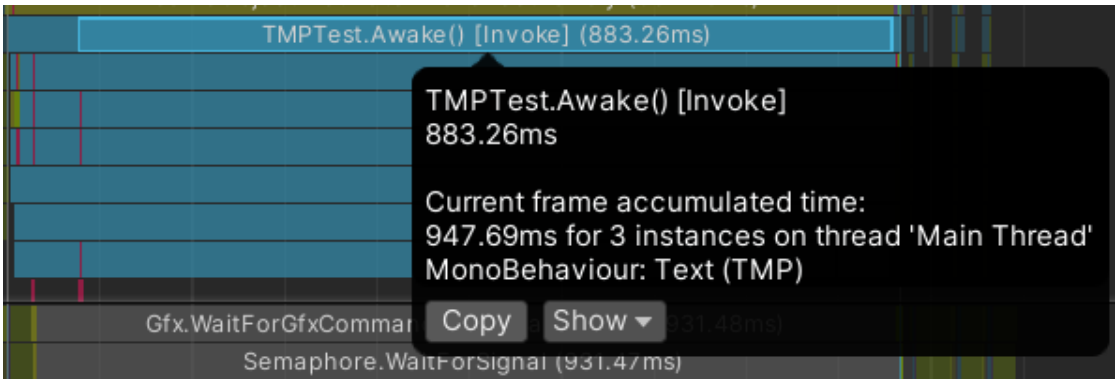
Tämä tarkoittaisi sitä, että aina kun sovellustekstejä muokataan tai uusi kieli lisätään sovellukseen, on olemassa mahdollisuus, etteivät tekstit tietyissä kohtaa enää sovi niille asetettuihin tekstikenttiin tai fontti näyttää eri kokoiselta samanpituisille teksteille. Kyseinen tapa muuttaa fonttikokoa ei myöskään ottanut huomioon eri laitteiden ruutujen resoluutioiden eroja, ja tekstit asettuivat useammin toistensa päälle.

Seuraavaksi pyrittiin kopioimaan Auto Size -asetuksen toimintaa ja asettamaan fontti sopimaan tekstikenttään. Tätä varten kirjoitettiin oma skripti (kuva 18), jossa fonttikokoa pienennettiin vähitellen, kunnes se sopii tekstikenttään pystysuunnassa. Funktion sisältämä skripti kiinnitettiin tekstiobjektiin ja FitText-funktio suoritettiin, kun tekstiobjekti otettiin käyttöön.

```
1 reference
public void FitText(string text)
{
    tmp.fontSize = maxFontSize;
    tmp.text = text;

    tmp.ForceMeshUpdate();

    float containerHeight = containerRect.rect.height;

    while (tmp.preferredHeight > containerHeight && tmp.fontSize > minFontSize)
    {
        tmp.fontSize -= 1f;
        tmp.ForceMeshUpdate();
    }
}
```

Kuva 18. FitText-funktio

Jokaiselle tekstille asetettiin suuri fonttikoko, jota voitiin alkaa alentamaan silmukassa (loop) while-komennolla. Silmukka toistetaan niin pitkään, kunnes sille asetettu ehto ei enää täyty. Ehtona toimi TextMeshPro-komponentista saatava muuttuja preferredHeight. Tämä muuttuja määrittää tekstikentän korkeuden, jossa sijoitettava teksti mahtuu olemaan kokonaisena. Jokaisella silmukan iteraatiolla, jos teksti ei vielä mahtunut kokonaan tekstikenttään, pienennettiin fontin kokoa yhdellä yksiköllä. Tämä skripti vähensi piikkejä ja madalsi niitä, jotka ylittivät kuvabudjetin. Sovelluksen testauksen yhteydessä kuitenkin voitiin huomata, että joissain kohdissa sovellusta skripti aiheutti valtavasti yleisrasitetta (kuva 19).



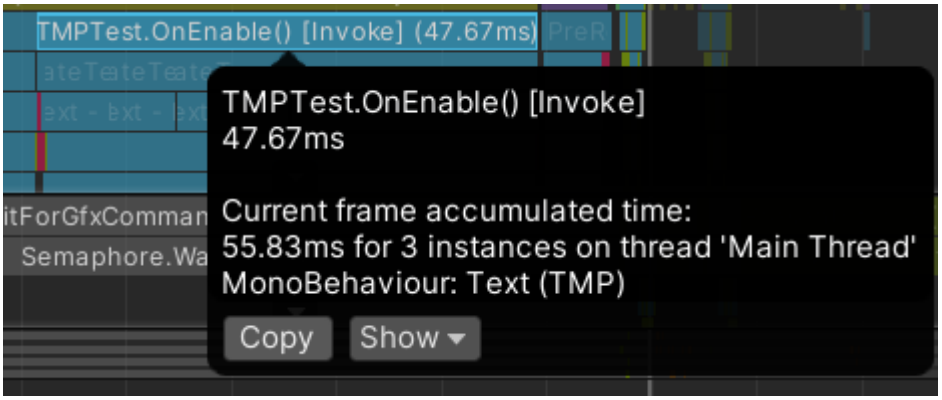
Kuva 19. Kuvan kesto Raportointi-näkymään siirtyessä FitText-funktion kanssa

Kuvasta 19 voidaan havaita, että skriptin suorittaminen oli kestänyt yhteensä 947.69 millisekuntia. Tämä ylitti kuvabudjetin 33 millisekuntia yli 28 kertaisesti ja kyseinen piikki aiheutti muiden prosessien kanssa kokonaisen sekunnin viiveen sovelluksen käytössä. Dynaaminen tekstin skaalaus manuaalisesti eri kokoisille tekstipätkille, resoluutioille ja usealle kielelle oli selkeästi ongelma, johon piti varata enemmän aikaa, mitä opinnäytetyölle oli varattu.

Viimeisenä vaihtoehtona kokeiltiin hyödyntää osittain TextMeshPro-komponentin toimintaa. Fonttikokoa säätelevää skriptiä muokattiin. Uudet toiminnot antoivat Auto Size -asetuksen määrittää ensin oikean koon sovelluksen teksteille, jonka jälkeen asetus poistettiin käytöstä (kuva 20). Skriptin suoritus luo yleisrasitetta ensimmäisen kerran tapahtuessaan, mutta sen luoma piikki on pienempi ja huomaamattomampi (kuva 21).

```
Unity Script (324 asset references) | 0 references
public class TMPTest : MonoBehaviour
{
    TextMeshProUGUI m_TextMeshProUGUI;

    Unity Message | 0 references
    void OnEnable()
    {
        m_TextMeshProUGUI = this.GetComponent<TextMeshProUGUI>();
        m_TextMeshProUGUI.ForceMeshUpdate();
        float newSize = m_TextMeshProUGUI.fontSize;
        m_TextMeshProUGUI.enableAutoSizing = false;
        m_TextMeshProUGUI.fontSize = newSize;
    }
}
```

Kuva 20. Auto Size -asetusta hyödyntävä skripti



Kuva 21. Kuvan kesto Raportointi-näkymään siirtyessä päivitetyn skriptin kanssa

Fonttikoot eivät enää muuttuneet sen jälkeen, kun ne ensimmäisen kerran asetettiin skriptin kautta. Piikki tapahtui, kun siirryttiin näkymästä toiseen. Viive oli tarpeeksi pieni, ettei sitä huomannut helposti käytön aikana, ja toisen kerran tekstin tullessa näkyviin, viivettä ei ollut ollenkaan. Kuvabudjetin ylityksissä jouduttiin tekemään myönnytyksiä, jotta voitiin ylläpitää johdonmukaista tekstin skaalausta koko projektissa riippumatta tekstin pituudesta, kielestä tai näytön resoluutiosta.

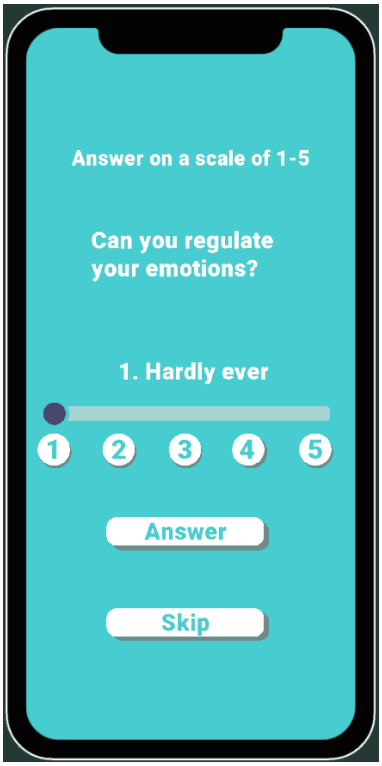
6.2 Kysymyslaatikot

Sovelluksen keskeisin osa on kysymyksiin vastaaminen. Vastauksista kerättyä dataa on tarkoitus käyttää vastaajien avuntarpeen kartoitukseen. Sovelluksessa pelattavat minipelit ovat toissijaisia ja niitä ei tarvitse koskaan pelata, mutta jokainen käyttäjä tulee vastaamaan kysymyksiin. Kokeneempi käyttäjä

voi selata kysymyksiä ja vaihtaa aihealueita nopeaan tahtiin, joten on tärkeää, että prosessissa on mahdollisimman vähän yleisrasitetta.

Kysymys-näkymään päästään valitsemalla alkunäkymästä ensin aihealue ja sitten kysymyspatteristo (kuva 22). Kysymyslaatikot, joita näkyy kuvan 22 oikeassa laidassa, ovat sovelluksessa luotuja painikkeita, joissa on kysymysteksti. Vastausnäkymään päästään kysymyslaatikkoa painamalla (kuva 23). Prosessia profiloidessa voitiin huomata piikkejä, jotka olivat lähtöisin Pie-skriptistä. Skripti vastaa kysymyksiin navigoinnista ja laittaa alulle kysymyslaatikoiden luomisen.

Kuva 22. Kysymys-näkymään siirtymisen prosessi alkunäkymästä



Kuva 23. Vastausnäkymä

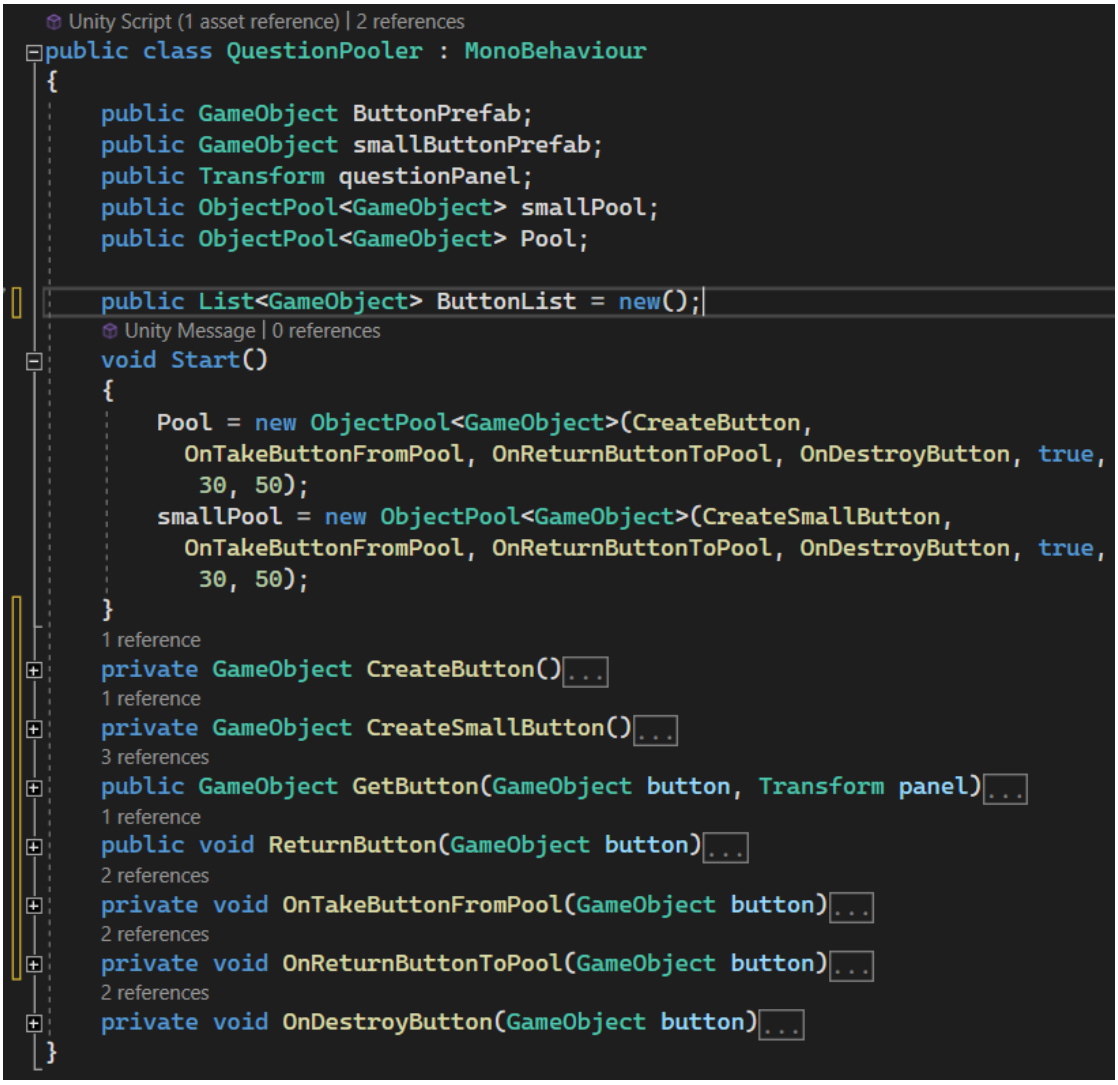
Profiloinnissa piikki tapahtui kuvan 22 keskimmäisessä vaiheessa eli juuri ennen kysymyslaatikoiden luontia. Luvussa 5.4 selitettiin fonttikoon muutoksien aiheuttavan Canvas-komponentin uudelleenpiirtymisen, sama pätee kaikkiin käyttöliittymässä oleviin elementteihin. Kuvan 23 vierityspalkin arvo palautettiin alkutilaan aina, kun kysymyslaatikoita luotiin. Tämä aiheutti uudelleenpiirtymisen ja loi yleisrasitetta käyttöliittymän päivityksen muodossa. Pie-skriptissä oli yksi koodirivi, ”questionsScrollbar.value = 1;”, joka sai aikaan kolmasosan kuvan kestosta. Ongelma korjattiin korvaamalla koodirivi toisella: ”questionsScrollbar.SetValueWithoutNotify(1);”. SetValueWithoutNotify on funktio, joka asettaa vierityspalkin arvon ilman, että sen OnValueChanged-vastakutsua (callback) suoritetaan. Tämä esti sen, ettei uudelleenpiirtymistä tapahtunut arvonmuutoksen takia.

Tarkemman tarkastelun tuloksena kuitenkin huomattiin, että ilman OnValueChanged-vastakutsua, kysymyslaatikoiden paneelia ei vieritetty ja näkymä jäi virheelliseksi. Sen sijaan, että vierityspalkin arvoa muutettiin, asetettiin kysymyslaatikoiden paneeli suoraan oikeaan kohtaan. Tämä tapahtui korvaamalla aiemmat koodirivit uudella: ”scrollRect.verticalNormalizedPosition = 1f;”. Vierityspalkki päivitti oman sijaintinsa ja arvonsa paneelin sijainnin mukaan, eikä ylimääräisiä vastakutsuja syntynyt.

Taulukossa 2 on esitettynä kuvabudjetin ylittäviä tekijöitä, joista GarbageCollector aiheutti noin puolet normaalin käytön budjetin ylittävistä piikeistä. Muistinsiivouksen tarve syntyi suurimmaksi osaksi kysymyslaatikoiden luomisesta ja niiden tuhoamisesta. Aina, kun käyttäjä navigoi vastausnäkymään, vanhat kysymyslaatikot tuhottiin ja uudet luotiin niiden tilalle. Prosessi varasi muistia ja sitä jouduttiin vapauttamaan muistin kapasiteetin vähentyttyä.

Peliobjektien luominen ja tuhoaminen lisäsi muistisiivouksen tarvetta ja samalla se loi yleisrasitetta. Sen sijaan, että peliobjekteja luotaisiin ja tuhottaisiin jatkuvasti, voidaan hyödyntää objektivarantoa (object pool). Varantoon nimensä mukaisesti varataan objekteja, joita voidaan uusiokäyttää sitä mukaan, kun tarve niiden käytölle syntyy. Esimerkiksi kysymyslaatikoiden tapauksessa uusia peliobjekteja ei tarvitse luoda, jos varannosta voidaan ottaa käyttöön jo olemassa olevia objekteja.

Olemassa olevaa koodia ei tarvinnut muokata paljon. Sen sijaan että uusia nappeja luotaisiin jatkuvasti, otettiin varannosta käyttöön kysymyslaatikon peliobjekti. Alkuperäiseen koodiin piti myös lisätä muuttuja, johon talletettiin peliobjektin järjestys hierarkiassa, jotta kysymykset saatiin pidettyä oikeassa järjestyksessä. Itse objektivarannon luonti tapahtui QuestionPooler-skriptissä (kuva 24).



```
Unity Script (1 asset reference) | 2 references
public class QuestionPooler : MonoBehaviour
{
    public GameObject ButtonPrefab;
    public GameObject smallButtonPrefab;
    public Transform questionPanel;
    public ObjectPool<GameObject> smallPool;
    public ObjectPool<GameObject> Pool;

    public List<GameObject> ButtonList = new();
    Unity Message | 0 references
    void Start()
    {
        Pool = new ObjectPool<GameObject>(CreateButton,
          OnTakeButtonFromPool, OnReturnButtonToPool, OnDestroyButton, true,
           30, 50);
        smallPool = new ObjectPool<GameObject>(CreateSmallButton,
          OnTakeButtonFromPool, OnReturnButtonToPool, OnDestroyButton, true,
           30, 50);
    }
    1 reference
    private GameObject CreateButton()...
    1 reference
    private GameObject CreateSmallButton()...
    3 references
    public GameObject GetButton(GameObject button, Transform panel)...
    1 reference
    public void ReturnButton(GameObject button)...
    2 references
    private void OnTakeButtonFromPool(GameObject button)...
    2 references
    private void OnReturnButtonToPool(GameObject button)...
    2 references
    private void OnDestroyButton(GameObject button)...
}
```

Kuva 24. QuestionPooler-skripti, joka vastaa objektivarannon hallinnasta

Skriptissä käytettiin Unityn omaa ObjectPool-luokkaa, jossa toteutettiin IObjectPool<T0>-rajapinnassa määritetyt funktiot. Funktiot määrittivät toiminnot varannon luonnille, varannosta objektin käyttöön ottamiselle sekä objektin varantoon takaisin palauttamiselle. Peliobjekteja, tässä tapauksessa kysymyslaatikoita, ei ole aluksi luotu ollenkaan. Vasta kun tyhjästä varannosta koetettiin ottaa käyttöön peliobjekti, luotiin uusi peliobjekti.

Tämä loi ensimmäisellä kerralla yleisrasitetta ja budjetin ylittävän piikin, mutta sen jälkeen kysymyslaatikoita käytettiin uudelleen uusilla määrityksillä sen sijaan, että luotaisiin kokonaan uusia peliobjekteja. Piikkejä ei tapahtunut enää kysymyslaatikoiden luonnin takia ja muistinsiivouksen tarve väheni, kun peliobjekteja ei enää luotu ja tuhottu jatkuvasti.

6.3 Peliobjektien aktivointi ja pois käytöstä ottaminen

Peliobjekteihin kiinnitetyissä skripteissä voidaan käyttää OnEnable- sekä OnDisable-funktioita. Funktioita kutsutaan, kun peliobjekti asetetaan pois käytöstä (disable) tai asetetaan aktiiviseksi (enable). Esimerkiksi CheckMark-skriptissä (kuva 25) laukaistaan tapahtuma (event), kun peliobjekti aktivoituu. Skripti on liitetty kysymyslaatikoiden lapsiobjektiin, jonka avulla tarkastetaan, onko käyttäjä vastannut kaikkiin kysymyksiin, jotka ovat kysymyspatteristossa vastattavissa.

```
Unity Script (3 asset references) | 22 references
public class CheckMark : MonoBehaviour
{
    public static event HandleAllQuestionsAnswered OnAllPatteristoQuestionsAnswered;
    public delegate void HandleAllQuestionsAnswered();

    Unity Message | 0 references
    private void OnEnable()
    {
        OnAllPatteristoQuestionsAnswered.Invoke();
    }
}
```

Kuva 25. CheckMark-skripti

Kuvassa 22 oikeassa reunassa on ilmoitettu kysymysten määräksi 12 ja niillä jokaisella on oma kysymyslaatikkonsa. Vastausnäkymän käyttöliittymän arkkitehtuurin takia jokainen CheckMark-skriptin omaava aktiivinen lapsiobjekti kutsuu OnEnable-funktiotaan aina, kun käyttäjä vastaa kysymykseen. Enimmillään kutsu tapahtuu 22 kertaa, kun suurimman kysymyspatteriston kaikkiin kysymyksiin on vastattu. Jokainen kutsu tekee tismalleen saman asian, eli tarkistaa onko kaikkiin kysymyksiin vastattu ja kysytäänkö käyttäjältä vastauksien lähettämisestä. Sen sijaan riittäisi, että uusin aktiiviseksi asetettu lapsiobjekti tarkistaisi tilan vain kerran. Tätä varten CheckMark-skriptiä muokattiin niin, että sen toiminnot suoritettiin vain kerran (kuva 26).

```
Unity Script (3 asset references) | 22 references
public class CheckMark : MonoBehaviour
{
    public static event HandleAllQuestionsAnswered OnAllPatteristoQuestionsAnswered;
    public delegate void HandleAllQuestionsAnswered();

    private bool isActive = false;

    Unity Message | 0 references
    private void OnEnable()
    {
        if(!isActive)
        {
            OnAllPatteristoQuestionsAnswered.Invoke();
            isActive = true;
        }
    }
}
```

Kuva 26. CheckMark-skripti, jonka OnEnable-toiminnot suoritetaan vain kerran

Totuusarvolla (boolean) voitiin määrittää, ettei ensimmäisen aktivoinnin jälkeen enää laukaista tapahtumaa. Turhien kutsujen ehkäiseminen ei rajoitu pelkästään koodin puolelle. Uusien toiminnallisuuksien kehityksessä voi unohtaa poistaa tarpeettomat skriptit peliobjekteista. Kuvan 22 vasemmassa laidassa on nähtävissä aihealueen ympyrä. Niitä on alkunäkymässä kolme ja niillä jokaisella on PieActiveStatus-skripti. Skriptissä laukaistaan tapahtumia OnEnable- ja OnDisable-funktioissa, joiden avulla säädellään sovelluksen toimintaa. Kutsut tapahtuvat kaksi ylimääräistä kertaa, sillä kaikki aihealueiden ympyrät asetetaan pois käytöstä ja aktivoidaan samaan aikaan joka tapauksessa. Ylimääräisten skriptikomponenttien poistaminen vähensi yleisrasitetta, vaikkakin vain hieman.

6.4 Minigolf-minipeli

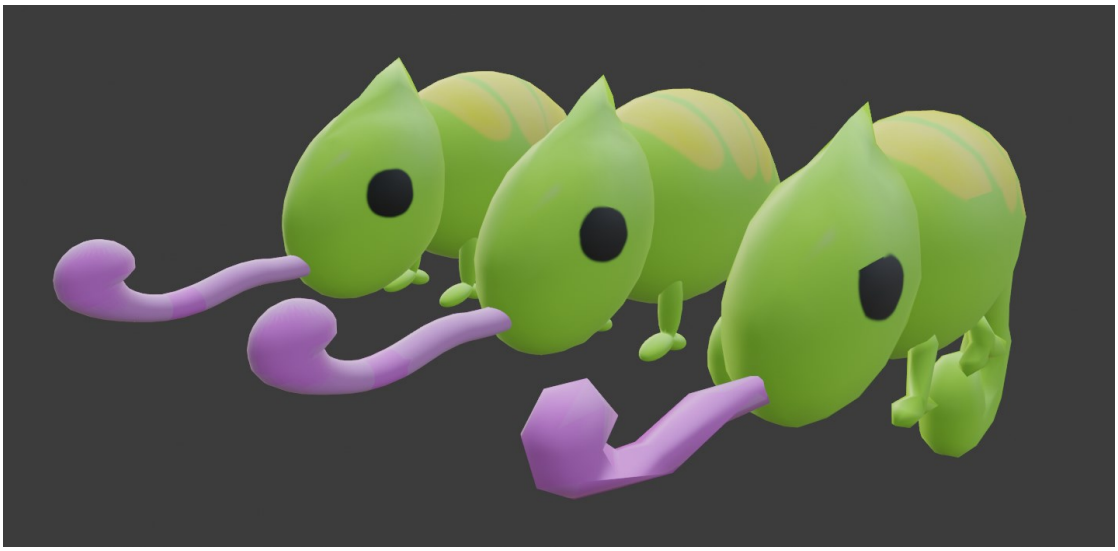
Sovelluksen minipelit ovat lähtökohtaisesti yksinkertaisia ja niiden ei ole tarkoitus olla pitkäkestoisia. Poikkeuksena on minigolf-minipeli, jossa on useita väyliä ja niille tehtyjä 3D-malleja. Osa 3D-malleista koostuu tuhansista ja jopa kymmenistä tuhansista kärkipisteistä (vertex), jotka muodostavat monikulmioista (polygon) kolmiulotteisen mallin (kuva 28). Mitä enemmän 3D-mallilla on monikulmioita, sen suurempi on sen resoluutio.

Suuren resoluution omaavat 3D-mallit käyttävät enemmän muistia ja voivat pitkittää grafiikkasuorittimen laskelmointeja (Unity Technologies 2023, 41). Tämä oli myös nähtävissä profiloinnissa, sillä minigolf-minipelissä oli havaitta-

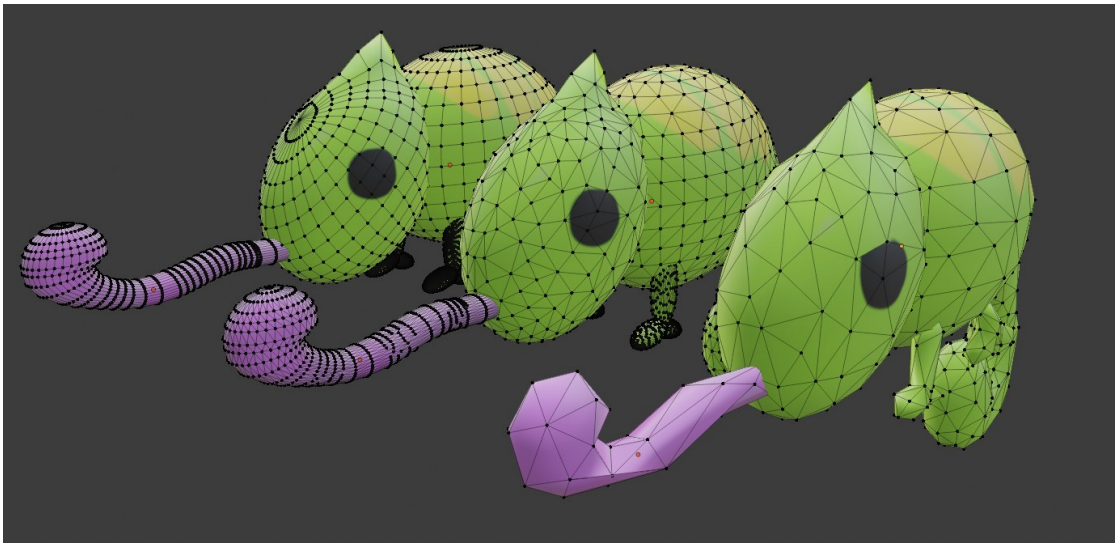
vissa piikkejä useassa eri kohdassa sen mukaan, miten paljon 3D-malleja ruudulla oli yhtä aikaa näkyvissä. Grafiikkasuorittimen taakkaa olisi suotavaa alentaa ja koska itse 3D-mallien tekstuurit eivät olleet suuria ja monimutkaisia, oli keskityttävä itse malleihin.

Yksi vaihtoehto oli mallintaa 3D-mallit uudestaan niin, että niissä käytettäisiin vähemmän kärkipisteitä. Toinen vaihtoehto oli hyödyntää 3D-mallien yksityiskohtaisuustasojen (level of detail) vaihtamista. Vaihtoehdoista jälkimmäinen ei vaatinut mallintajien työpanosta ja sen toteuttaminen opinnäytetyön aikataulun rajoissa oli helpompaa, joten se valittiin ratkaisuksi. Samalla alkuperäisten mallien resoluutiot ja yksityiskohdat voitiin pitää juuri sellaisina, kuin niiden mallintaja oli tarkoittanut.

Yksinkertaisuudessaan yksityiskohtaisuustasojen vaihto tarkoittaa sitä, että mitä kauempana pelimaailman kamera on, sitä pienemmän resoluution 3D-malleja se renderöi. Kuvassa 27 on kuvattuna kolme eri mallia, joista käytetään nimitystä LOD-taso. Vasemmalta oikealle, mallit ovat LOD 0, LOD 1 ja LOD 2. Mitä pienempi luku nimessä on, sitä suurempi on 3D-mallissa käytettävä resoluutio. Esimerkin LOD 0 -malli koostuu 16 570 monikulmiosta ja LOD 2 -mallissa on vain 891 monikulmiota, eli suurimman resoluution omaavassa 3D-mallissa on yli 18 kertaa enemmän monikulmioita, kuin pienimmän resoluution 3D-mallissa.



Kuva 27. 3D-mallien yksityiskohtaisuuksien eroja



Kuva 28. 3D-mallien koostumuksia

Kuvissa 27 ja 28 kuvatut 3D-mallit ovat muokattu Blender-mallinnusohjelmalla. Malleihin käytettiin Decimate-muokkainta (harvennus), jolla voitiin vähentää mallin kärkipisteiden määrää ja samalla säilytettiin alkuperäinen muoto mahdollisimman hyvin. Unity Technologiesin (2025d) dokumentaatiossa on kerrottuna kaksi tapaa määrittää yksityiskohtaisuuksien tasoja. Mallinnusohjelmassa voidaan joko nimetä mallit LOD-nimeämismallilla, jolloin Unity osaa luoda niille automaattisesti LOD-ryhmän 3D-mallin tuonnin yhteydessä tai LOD-ryhmän voi luoda itse jälkikäteen.

LOD-ryhmässä määritetään kameran etäisyydet, joissa 3D-mallien eri LOD-tasoja aletaan käyttämään (kuva 29). Asetuksista voidaan myös määrittää myös etäisyys, jossa 3D-mallia ei renderöidä enää ollenkaan. Aina kun kamera on tietyllä LOD-alueella, kuvan 29 tapauksessa LOD 2, asetetaan vastaava malli aktiiviseksi, kun loput otetaan pois käytöstä. Oikeiden etäisyyksien löytäminen oli lähtökohtaisesti kameran siirtämistä ja 3D-mallien tarkkailua, kunnes etäisyydeltä nähtävä 3D-malli oli visuaalisesti miellyttävä, eikä siitä voitu havaita pienempää resoluutiota.

Kuva 29. LOD-ryhmä

LOD-ryhmiä ja niille tehtyjä LOD-tasoja luotiin viidelle peliobjektille, joista kolme oli kameleontteja. Kaksi muuta peliobjektia olivat silta ja narusta tehty kaide. Ilman LOD-ryhmiä, minigolf-minipelissä oli yhteensä 284 000 monikulmiota. Monikulmioiden kappalemäärä alentui 160 700 monikulmioon, kun LOD 2 -mallit olivat aktiivisina. Piikkien määrä laski, mutta niitä silti tapahtui yhä. Piikit johtuivat vedessä käytettävästä varjostimesta ja peliobjekteja läpinäkyviksi tekevästä ominaisuudesta.

Pallon ja kameran väliin tuleva peliobjekti muutettiin läpinäkyväksi, ettei pallo jäisi käyttäjältä piiloon toisten peliobjektien taakse. Varjostimessa sekä läpinäkyvyyden aiheuttavassa ominaisuudessa yhteistä on se, että molemmissa renderöidään peliobjekteja läpinäkyvien peliobjektien takaa. Veden alle jäävät peliobjektit luovat grafiikkasuorittimelle yleisrasitetta, sillä ne ovat vielä näkyvissä läpinäkyvän pinnan alapuolella. Ongelmaa voitiin lievittää kameroiden kulmien uudelleen säätämisellä, jolloin päällekkäisiä peliobjekteja ei tarvinnut renderöidä samaan aikaan. Enimmillään minigolf-minipelissä piirtokutsuja oli

kuvakulmien säädön jälkeen 158 kappaletta ja nekin laskivat yleisesti alle 100 kappaleeseen, kun monimutkaisemmat väylät eivät olleet näkyvissä.

Ongelman varsinainen ratkaiseminen vaatisi syvempää perehtymistä varjostimien ja Unity-pelimoottorin grafiikkaliukuhihnan (graphics pipeline) toimintaan, jotta voitaisiin kehittää paremmin toimivat varjostimet. Esimerkiksi sen sijaan, että peliobjektien näkyvyyttä säädettäisiin pallon jäädessä niiden taakse, voitaisiin kehittää varjostin, joka näyttää pallon sijainnin peliobjektien takaa.

6.5 Optimoinnin jälkeinen tarkastelu

Profiloinnin ja optimoinnin aikana harjoitettiin suorituskyvyn vertailua. Potentiaalinen korjaava toimenpide toteutettiin ja tuloksia verrattiin aiempiin profiloinnin tuloksiin. Näin voitiin todeta, että korjaava toimenpide oli joko onnistunut tai epäonnistunut ja sen perusteella voitiin siirtyä seuraavaan optimoinnin kohteeseen tai laatia uusia korjauksia.

Luvussa 5.3 suoritettiin ensimmäinen sovelluksen profilointi 40ms kuvabudjetilla. Sama profilointi samoilla määrityksillä suoritettiin uudestaan. Profiloinnin aikana generoitiin 103291 kuvaa. Taulukossa 3 on kuvattu kuvabudjetin ylitykset ja tuloksista voidaan havaita kuvabudjetin ylittyvän kahdeksan kertaa niin, että ylitys tapahtuisi myös koontiversiossa.

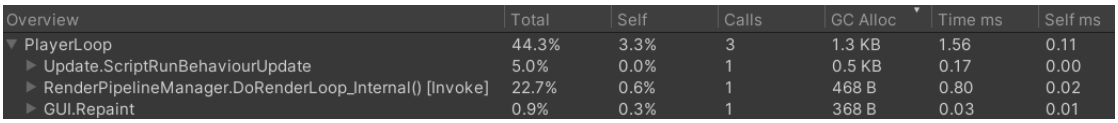
Taulukko 3. Kuvabudjetin ylitykset optimoinnin jälkeen sekä luvun 5.3 profiloinnin tulokset

	Player-Loop	EditorLoop	Garbage-Collector	Profiler.Flush-Counter	Yhteensä
Budjetin ylitykset	5	7	3	0	15
Luvun 5.3 profiloinnin tulokset	10	11	10	2	33

Kuvabudjetin ylittäviä piikkejä havaittiin jopa kaksi kertaa vähemmän kuin ensimmäisen profiloinnin aikana. Kuvabudjetiksi määritettiin 30ms itse korjaustoimenpiteiden toteutuksen aikana, joten budjetinylitysten vähentyminen anteliaammalla kuvabudjetilla oli odotettavissa. 40ms kuvabudjetin testausta ei

kuitenkaan suoritettu koontiversiossa ja Unity Editor -ympäristössä toiminta voi poiketa huomattavasti todellisesta käytöstä.

Muistinsiivouksen tarve vähentyi huomattavasti, mutta prosessi tapahtui yhä muutaman kerran. Profilointi-ikkunan hierarkkisessa näkymässä voidaan havaita kolme lähdettä, jotka varaavat muistia jatkuvasti joka kuvan luonnin yhteydessä (kuva 30). Sovelluksen koontiversiossa ei kahta alempaa lähdettä ollut havaittavissa; ne ilmenivät pelkästään Unity Editor -ympäristössä.



Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
PlayerLoop	44.3%	3.3%	3	1.3 KB	1.56	0.11
Update.ScriptRunBehaviourUpdate	5.0%	0.0%	1	0.5 KB	0.17	0.00
RenderPipelineManager.DoRenderLoop_Internal() [Invoke]	22.7%	0.6%	1	468 B	0.80	0.02
GUI.Repaint	0.9%	0.3%	1	368 B	0.03	0.01

Kuva 30. Muistinvarauksen lähteitä

Jäljellejäänyt muistia varaava kohde jäljitettiin skriptiin, jossa luotiin joka kuvan aikana uusi taulukko. Taulukkoon kerättiin peliobjektin kulmat ja sen avulla voitiin tarkkailla peliobjektin sijainta tarkemmin. Taulukon luonti siirrettiin erilleen funktiosta niin, ettei sitä luotu aina uudestaan. Tämän jälkeen muistinvaraus pysyi nollalukemassa, ellei esimerkiksi objektivarantoihin lisätty uusia objekteja tai listoja käyty läpi koodissa. Lisäksi koontiversion muistinsiivouksen tarvetta vähentää se, ettei siihen sisällytetä Debug.Log-funktioita, joita Unity Editor -ympäristössä käytetään virheenjäljitykseen.

7 TULOKSET JA JOHTOPÄÄTÖKSET

Opinnäytetyön tavoitteena oli optimoida WinMind App -sovelluksen toimintaa ja hyödyntää olemassa olevia profilointityökaluja optimoinnin tarpeiden tunnistamiseen. Tavoite saavutettiin ja opinnäytetyön aikana paikannettiin useita ongelmakohtia, joille suoritettiin korjaavia toimenpiteitä onnistuneesti. Löytämällä vastaukset tutkimusongelmasta johdettuihin kysymyksiin, saatiin kattavampi ymmärrys Unity-sovelluksen profiloinnista ja optimoinnista.

7.1 Tutkimustulokset

Tutkimustulokset ovat vastauksia opinnäytetyön luvussa 2.1 laadittuihin tutkimuskysymyksiin:

Mitä työkaluja Unity-sovelluksen profilointiin kannattaisi käyttää?

Opinnäytetyössä tarkasteltiin useita eri profilointityökaluja, joista suurin osa oli kolmannen osapuolen työkaluja. Unity-sovellusten profilointia voidaan suorittaa muilla kuin Unity-työkaluilla, mutta niistä saadut tulokset eivät tarjoa kattavaa analyysia sovelluksen taustalla tapahtuvista prosesseista. Useat kolmannen osapuolen profilointityökalut on suunniteltu työkalun kehittäjien omien kehitysalustojen tai komponenttien toiminnan tarkasteluun.

Luvussa 4 tarkasteltiin eri vaihtoehtoja profilointia varten ja huomattiin useasti profiloinnin olevan tuen puutteen vuoksi vajavaista tai kokonaan mahdotonta sovelluksen ohjelmistoalustan perusteella. Kehitysvaiheessa on kannattavampaa käyttää Unity-työkaluja, sillä ne mahdollistavat tehokkaan ongelmakohtien havaitsemisen ja kohdentamisen. Vasta koontiversiota kannattaa profiloida kohdealustalle tarkoitetuilla profilointityökaluilla mahdollisten laitekohtaisten ongelmien löytämiseksi ja komponenttien suorituskyvyn mittaamiseksi.

Miten Unity-sovelluksen optimoinnin tarpeet voidaan tunnistaa?

Unity-sovellusta profiloidessa tulee määrittää kuvabudjetti, jota seuraamalla voidaan tunnistaa optimointia tarvitsevat kohteet. Toimenpiteitä vaaditaan, jos kuvan kesto ylittää kuvabudjetin. Profiloidessa on tärkeää ottaa huomioon koontiversion ja Unity Editor -ympäristön erot, kun tarkastellaan suorituskyvyn piikkejä. Kuvabudjetin ylittävä piikki voi johtua pelkästään kehitysympäristön prosesseista, eikä sitä ole havaittavissa koontiversiossa.

On tärkeää ottaa huomioon myös kohdealustan resurssit, etenkin jos on tarkoitus kehittää sovellus mobiilialustalle. Mobiilialustojen suorituskyky ei vastaa tehokkaampia alustoja, kuten pöytäkoneita, joten profilointi on suoritettava mobiililaitteella. Unity Profiler -työkalu voidaan yhdistää sovelluksen kehitysversioon tai sitä voidaan käyttää suoraan Unity Editor -ympäristössä.

Optimoinnin tarve voidaan havaita sovelluksen käytön yhteydessä ilman profilointia, jos esimerkiksi sovelluksen kuvataajuus tippuu merkittävästi. Unity Profiler -työkalun avulla voidaan selvittää funktiotasolla kuvan keston aiheuttajat

ja muilla Unity-työkaluilla voidaan tarkastella esimerkiksi muistin käyttöä ja kuvan renderöintiä vaihe vaiheelta.

Miten WinMind App -sovellusta voidaan optimoida?

WinMind App -sovelluksessa havaittiin useita eri optimoinnin kohteita. Näistä kohteista sovellustekstien luonnilla oli suurin vaikutus suorituskykyyn, sillä sovelluksessa näkyy tekstiä lähes jokaisessa näkymässä. TextMeshPro-komponentin Auto Size -asetus aiheutti suuren määrän Canvas-komponentin uudelleenpiirtymisiä ja sen käytön lopettaminen vähensi yleisrasitetta.

Tekstien optimointi vaikutti myös kysymyslaatikoiden luontiin, mutta suurempi vaikutus kysymyslaatikoiden toiminnassa oli objektivarannon käyttöönottamisella. Peliobjekteja kierrätettiin uusien luomisen sijaan ja ne otettiin pois käytöstä ilman peliobjektin tuhoamista. Objektivarannon käytössä havaittiin suorituskyvyssä piikki varannon alustamisen aikana, kun ensimmäiset peliobjektit luotiin, mutta sen jälkeen yleisrasitetta ja muistinsiivouksen tarvetta ei enää syntynyt.

Monen peliobjektin skripteissä on niiden aktivoinnin ja pois käytöstä ottamisen yhteydessä kutsuttavia funktioita. Sovelluksessa näitä funktioita kutsuttiin useasti ylimääräisiä kertoja tai niiden sisältämä koodi oli toteutukseltaan toistuvaa. Koodien osalta ongelmakohdat olivat syntyneet ymmärryksen ja tiedon puutteesta, koska ennen profilointia niiden toiminnan virheellisyydestä ei ollut havaintoa. Koodia parantelemalla pystyttiin vähentämään GarbageCollector-prosessien tarvetta sekä vähentämään yleisrasitetta luovia turhia kutsuja.

Sovellus ei ollut graafisesti vaativa, mutta minigolf-minipeli aiheutti selkeästi ongelmia sen 3D-objektien takia. Osa 3D-malleista koostui kymmenistä tuhansista monikulmioista ja mobiilialustalla voitiin havaita grafiikkasuorittimen suorituskyvyssä ongelmia. Opinnäytetyötä varten ei koettu tarpeelliseksi työllistää sovelluksen kehitystiimin muita jäseniä uudelleenmallinnusta varten, joten päädyttiin käyttämään LOD-malleja.

Renderöitävä 3D-malli vaihdettiin pienemmän tai suuremman resoluution malliin kameran etäisyyden mukaan. Näin kaukaisemmat peliobjektit pystyivät säilyttämään näköisyytensä, vaikka niiden monikulmioiden määrä laski. 3D-mallien renderöinnissä ongelmia aiheutti myös minigolf-minipelissä käytettävä vesivarjostin, jonka käytön takia peliobjekteja renderöitiin päällekkäin läpinäkyvien peliobjektien takaa. Minipelin kuvakulmia säätämällä ongelmaa saatiin lievitettyä niin, ettei siitä aiheutunut kuvabudjetin ylityksiä. Lopullinen ratkaisu vaatisi uuden varjostimen luomista ja sen tekemiseen vaadittaisiin enemmän ymmärrystä ja tietotaitoa varjostimien toiminnasta. Ottaen huomioon potentiaalisten hyötyjen pienuuden nykytilanteeseen verrattuna, uuden varjostimen kehitystä ei aloitettu.

7.2 Johtopäätökset

Opinnäytetyön tutkimusongelmaksi määritettiin luvussa 2.1 WinMind App -sovelluksen optimointi. Jotta optimointia voitiin suorittaa, oli ensin selvitettävä, miten optimointia tarvitsevat kohteet voitiin löytää. Tämä oli mahdollista profilointityökalujen avulla. Eri työkaluja vertailemalla voitiin selvittää paras mahdollinen profilointityökalu Unity-sovelluksen tarkasteluun. Opinnäytetyössä käytiin läpi profiloinnin pääpiirteitä ja selvitettiin, miten profilointityökalua lukemalla voitiin tunnistaa suorituskyvyn piikkejä aiheuttavia prosesseja.

Sovellukselle suoritettiin testaus, jonka aikana otettiin ylös kuvabudjetin ylitykset ja niiden syyt. Syitä tutkittiin tarkemmin ja niistä pystyttiin eristämään ongelmia aiheuttavat tekijät. Ongelmia pyrittiin ratkaisemaan ja jos korjaus ei tuottanut positiivisia tuloksia uudelleen testatessa, koetettiin uusia ratkaisuja. Korjaustoimenpiteiden ja testauksen sykli jatkui niin pitkään, kunnes profiloinnissa voitiin havaita budjettien ylityksien vähentyminen.

Optimoinnissa jouduttiin tekemään myönnytyksiä tiettyjen kuvabudjetin ylitysten suhteen, sillä ne mahdollistivat sovelluksen ongelmattoman toiminnan myöhemmässä käytössä. Taulukossa 3 on todennettu suoritettujen optimointien hyödyt. Kuvabudjetin ylityksiä tapahtuu huomattavasti vähemmän korjauksien jälkeen, joten voidaan todeta optimoinnin olleen onnistunutta.

7.3 Jatkokehitysideat

Luvussa 3.3 määritettiin optimoinnin olevan teoreettisten maksimien ja minimien tavoittelemista. Opinnäytetyössä suoritettiin optimointia vain siihen pisteeseen saakka, että kuvabudjetin ylityksiä ei enää havaittu optimoitavissa kohteissa. Piikkejä silti tapahtui, mutta ne eivät ylittäneet budjettia. Optimointia voitaisiin suorittaa paljon enemmän ja samalla pystyttäisiin tarkastelemaan yli-optimoinnin vaikutusta sovelluksen kehitysprosessiin.

Opinnäytetyössä hyvät koodauksen käytännöt jäivät toissijaisiksi ja tutkimuksessa keskityttiin enemmän tiettyjen ongelmien paikantamiseen ja niiden korjaamiseen. Lisätöitä optimoinnin osalta voitaisiin tulevaisuudessa välttää, jos kirjoitettu koodi olisi jo valmiiksi optimoitua. Hyvien käytäntöjen määrittäminen ja niiden omaksuminen rutiineihin tutkimuksen puitteissa mahdollistaisi syvemmän perehtymisen jo hyviksi todettujen käytäntöjen taustoihin. Näin ratkaisut eivät jäisi vain toteutuksen tasolle ja päätöksien takana olisi syvempi ymmärrys pintaraapaisun sijaan.

Sovelluksen profilointia varten voitaisiin kehittää omia työkaluja Unity Editor -ympäristössä. Profilointia avustettiin esimerkiksi luvussa 5.2 itse tehdyllä skriptillä, mutta varsinaisten työkalujen tasolle ei edetty. Uudeksi tutkimukseksi työkalun kehitys voisi olla hankala tutkimuksen aihe, sillä sitä varten täytyy olla jokin tarve, mutta jatkokehityksen kannalta tarpeita on voinut syntyä eri osa-alueissa optimoinnin aikana. Esimerkiksi automaatiotestaus olisi voinut nopeuttaa ja yhdenmukaistaa profiloinnin prosesseja.

Optimoinnin osa-alueita on lukuisia ja syvemmän ymmärryksen saavuttaminen suorituskykyä parantavista toimenpiteistä voisi toimia omana tutkimuskohteenaan. Opinnäytetyössä mainittiin useasti omien varjostimien kehittämisen olevan mahdollisuus. Aiheen laajuuden vuoksi varjostimien kehitys toimisi mainiosti omana tutkimuksenaan, mutta voisi silti pitää optimoinnin läheisenä aihealueena. Yksinkertaisuudessaan olemassa olevan varjostimen toimintaa voitaisiin parantaa.

Olemassa olevien ratkaisujen parantaminen yltäisi myös sovelluksessa käytettävien rajapintakutsujen (api call) toteutukseen. Opinnäytetyön kohteena oli

itse Unity-sovellus, joten rajapinnassa tapahtuvien toimintojen katsottiin jäävän tutkimuksen laajuuden ulkopuolelle. Tähän voisi sisältyä myös itse tietokannan rakenteen parantaminen, jotta rajapintakutsut voitaisiin suorittaa mahdollisimman tehokkaasti.

Sovelluksessa käytettävien ratkaisujen parantaminen ei välttämättä olisi aina paras mahdollinen vaihtoehto ja uusien ratkaisujen kehitys voisi antaa parempia tuloksia. TextMeshPro-komponenttien käytön voisi välttää kokonaan, jos sovelluksen elementtien skaalautuvuudelle luotaisiin oma ratkaisu. Kaiken kaikkiaan optimointi on ehtymätön tutkimusten lähde, jossa pitäisi vain määritellä mihin optimointi kohdistetaan.

8 POHDINTA

Opinnäytetyön tuloksena saatiin aikaan paranneltu versio sovelluksesta, jossa optimoinnit ovat vähentäneet yleisrasitetta. Tutkimuskysymyksiin löydettiin vastauksia ja tutkimusongelma onnistuttiin ratkaisemaan. On kuitenkin kyseenalaista, oliko opinnäytetyön mittakaava vajaa, sopiva vai liian suuri, sillä pienikin, yksi onnistunut optimointi olisi määrittänyt opinnäytetyön olleen onnistunut. Optimointien laajuuteen tai niiden vaikutukseen ei otettu kantaa sen enempää kuin että optimoinnin tuloksena oli tarkoitus saada kuvabudjetin ylitykset katoamaan.

Optimoinnin laajuuden määrittäminen olikin opinnäytetyössä haastavaa. Liian laaja-alainen tutkimus olisi ollut ikuisuusprojekti, sillä pieniä optimoinnin kohteita löytyi profiloidessa paljon odotettua enemmän. Kehittämistyön aikana huomattiin useaan otteeseen, että sovelluksen kehityksessä ei ollut pyritty parhaimpiin mahdollisiin ratkaisuihin, vaan niihin, jotka toimivat. Joskus yksinkertaisetkin asiat oli tehty turhan monimutkaisesti tai ymmärryksen puutteen vuoksi huonosti. Aina kun vastaavia ongelmakohtia löytyi, oli selkeää, etteivät koodaustaidot olleet halutulla tasolla. Tämä herätti epäilyksiä siitä, olivatko toteutetut optimoinnit tarpeeksi hyviä, mutta ainakin oppimista oli tapahtunut, sillä aiemmat ratkaisut eivät enää vaikuttaneet toimivilta ja niihin keksittiin parempia vaihtoehtoja.

Ennen opinnäytetyötä profilointityökalujen käyttö oli vieras käsite ja toteutuksen aikana huomattiin, ettei niiden käyttäminen ollutkaan ennakko-oletusten mukaisesti liian monimutkaista. Profilointi itsessään tukee projektin kehitystä ja auttaa ymmärtämään sovelluksen toimintaa entistä paremmin. Profilointi ja optimointi auttoivat myös havaitsemaan puutteita omassa ymmärryksessä ja jokainen kuvabudjetin ylitys oli mahdollisuus oppia jotakin uutta joko koodin tai Unity-pelimoottorin toiminnasta.

Opinnäytetyön teoriaosuudessa olisi voinut olla hyvä perehtyä lisäksi koodauksen hyviin käytäntöihin, mutta muuten koottu teoria tuki tutkimusta hyvin: kohdealustojen valinta, profilointityökalujen mahdollisuudet ja itse optimoinnin tason ja onnistuneen optimoinnin määrittäminen saivat pohjan teoreettisessa viitekehyksessä.

Keskeisin osa opinnäytetyössä oli optimoinnin toteutus. Toteutetut optimoinnit ja niiden tulokset, positiiviset kuin negatiivisetkin, pyrittiin esittämään selkeästi ja tuloksiin viittaamalla. Monessa osiossa päädyttiin kuitenkin jättämään kuvia käyttämättä havainnollistamiseen ja turvauduttiin selostamiseen, ettei kuvien määrä kasvaisi mittavasti ja teksti pysyisi helppolukuisena. Havainnollistavien, olemassa olevien näkymien käyttämättä jättäminen, voi herättää epäilyksiä informaation todenmukaisuudesta, mutta luotettavuuden vuoksi päätökset ja tulokset on aina pyritty selittämään.

Opinnäytetyön luotettavuus kärsii hieman siitä, että testauslaitteita oli käytössä vain yksi. Optimoituja versioita ei ehditty testaamaan läpikotaisin, jotta niiden toimintaa olisi voitu testauttaa loppukäyttäjillä ja heidän omilla laitteillaan. Sovelluksen toiminta on todistetusti parantunut, mutta ei ole mitään näyttöä siitä, että sovellus olisi käytettävissä jokaisella sitä tukevalla laitteella. Lisäksi potentiaaliset ongelmat lasten ja nuorten täysivaltaisen itsemääräämisoikeuden puutoksen vuoksi tutkimukseen osallistuminen olisi voinut hankaloittaa tutkimuksen suorittamista.

Tämä kuitenkin tarkoittaa sitä, että tutkimus on pyritty suorittamaan eettisesti ja sen toteutuksessa on otettu huomioon sovelluksen loppukäyttäjät. Haastattelut sovelluksen käytön tuntumasta olisivat voineet auttaa hahmottamaan todellisen käytön kestoa ja sitä, mitä käyttäjät yleisimmin sovelluksessa tekevät.

Optimoinnin tarve sovelluksessa kuitenkin syntyi samoista ongelmista eri alustoilla, joten optimointia voitiin suorittaa kohdelaitteen tehokkuudesta huolimatta.

Opinnäytetyö oli hyvin opettavainen kokemus, joka laajensi näkemyksiä sovelluskehityksestä. Tutkimusaiheena optimointi on yleinen ja sitä on tutkittu laajasti muiden tahojen osalta, joten uuden teorian kehittämistä ei välttämättä siihen vaadita. Tutkimus auttoi hyväksymään omien taitojen puutteellisuuden ja antoi mahdollisuuden kehittää taitoja, joiden avulla voidaan tuottaa laadukkaampia ja kestävämpiä ratkaisuja.

LÄHTEET

Advanced Micro Devices s.a. Unity CPU profiling guide. WWW-dokumentti. Saatavissa: https://gpuopen.com/learn/unity-cpu-profiling-guide/ [viitattu 1.3.2025].

Advanced Micro Devices. 2024. Specification. WWW-dokumentti. Saatavissa: https://docs.amd.com/r/en-US/57368-uProf-user-guide/Specification [viitattu 20.3.2025]

Android Developers. 2023. AGI supported devices. WWW-dokumentti. Saatavissa: https://developer.android.com/agi/supported-devices [viitattu 21.3.2025].

Android Developers. 2025. Android Debug Bridge (adb). WWW-dokumentti. Saatavissa: https://developer.android.com/tools/adb [viitattu 1.3.2025].

Antoniou, A. & Lu, W.-S. 2007. Practical Optimization: Algorithms and Engineering Applications. New York: Springer Science+Business Media, LLC.

Apple Inc. s.a. Membership Details. WWW-dokumentti. Saatavissa: https://developer.apple.com/programs/whats-included/ [viitattu 2.3.2025].

Essam, M. 2024. Mastering Unity Game Development with C#: Harness the full potential of Unity 2022 game development using C#. Birmingham: Packt Publishing Ltd.

Google s.a. Get started with Play Console. WWW-dokumentti. Saatavissa: https://support.google.com/googleplay/android-developer/answer/6112435?hl=en#zippy=%2Cstep-choose-a-developer-account-type%2Cstep-pay-registration-fee [viitattu 2.3.2025].

Grambow, M., Lehmann, F. & Bermach, D. 2019. Continuous Benchmarking: Using System Benchmarking in Build Pipelines. *IEEE International Conference on Cloud Engineering,* 241–246. PDF-dokumentti. Saatavissa: https://doi.org/10.1109/IC2E.2019.00039 [viitattu 2.3.2025].

Hakala, J. 2022. Hyvä, parempi, valmis: Opinnäyteopas ammattikorkeakouluille. Helsinki: Gaudeamus.

Intel Corporation. 2024a. Intel® VTune™ Profiler System Requirements. WWW-dokumentti. Saatavissa: https://www.intel.com/content/www/us/en/developer/articles/system-requirements/vtune-profiler-system-requirements.html [viitattu 20.3.2025]

Intel Corporation. 2024b. Introduction. WWW-dokumentti. Saatavissa: https://www.intel.com/content/www/us/en/docs/vtune-profiler/user-guide/2025-0/introduction.html [viitattu 20.3.2025].

Intel Corporation. 2024c. Profiling Games built with Unity* (NEW). WWW-dokumentti. Saatavissa: https://www.intel.com/content/www/us/en/docs/vtune-profiler/cookbook/2025-0/profiling-games-built-with-unity.html [viitattu 1.3.2025].

Knuth, D. 1974. Structured Programming with *go to* Statements. AMC Computing Surveys (CSUR) 4, 261–301. Verkkolehti. Saatavissa: https://doi.org/10.1145/356635.356640 [viitattu 28.2.2025].

Kuula, A. 1999. Toimintatutkimus: Kenttätyötä ja muutospyrkimystä. 2. painos. Tampere: Vastapaino.

Kuula, A. 2011. Tutkimusetiikka: Aineistojen hankinta, käyttö ja säilytys. 2. painos. Tampere: Vastapaino.

Lehtola, A. 2018. Optimizing Unity Projects. Kajaanin ammattikorkeakoulu. Tietojenkäsittely. Opinnäyte. PDF-dokumentti. Saatavissa: https://urn.fi/URN:NBN:fi:amk-2018052710496 [viitattu 2.3.2025].

Liiketoimintamalli s.a. WinMind Oy. WWW-dokumentti. Saatavissa: https://www.winmind.fi/education-for-teachers [viitattu 28.2.2025].

Meistä s.a. WinMind Oy. WWW-dokumentti. Saatavissa: https://www.winmind.fi/our-vision [viitattu 28.2.2025].

Microsoft. 2021. Profiling Overview. WWW-dokumentti. Saatavissa: https://learn.microsoft.com/en-us/dotnet/framework/unmanaged-api/profiling/profiling-overview#supported-features [viitattu 1.3.2025].

Microsoft. 2022. Windows Performance Toolkit. WWW-dokumentti. Saatavissa: https://learn.microsoft.com/en-us/windows-hardware/test/wpt/ [viitattu 1.3.2025].

Microsoft s.a. All about processors (CPUs). WWW-dokumentti. Saatavissa: https://support.microsoft.com/en-us/windows/all-about-processors-cpus-06dc72ec-3de2-4eb8-8cc2-7e5f2417b90b [viitattu 1.3.2025].

Narkilahti, A. 2021. Unity-optimointi. Lapin ammattikorkeakoulu. Tieto- ja viestintätekniikka. Opinnäyte. PDF-dokumentti. Saatavissa: https://urn.fi/URN:NBN:fi:amk-202103253792 [viitattu 2.3.2025].

Ojasalo, K., Moilanen, T. & Ritalahti, J. 2015. Kehittämistyön menetelmät: uudenlaista osaamista liiketoimintaan. Helsinki: Sanoma Pro Oy.

Okkola, A. 2023. Pelitestaus ja profilointi Unity-pelimoottorissa: Case: VR-HYPO virtuaalinen oppimisympäristö. LAB-ammattikorkeakoulu. Tieto- ja viestintätekniikka. Opinnäyte. PDF-dokumentti. Saatavissa: https://urn.fi/URN:NBN:fi:amk-202305088206 [viitattu 2.3.2025]

Peräkylä, A. 2004. Reliability and validity in research based on naturally occuring social interaction. Teoksessa Silverman, D. (toim.) Qualitative Research: Theory, Method and Practice. 2. painos. Lontoo: Sage Publications Ltd, 283–304.

Sexton, M. 2017. The History of AMD CPUs. Future US Inc. WWW-dokumentti. Saatavissa: https://www.tomshardware.com/picturestory/713-amd-cpu-history.html [viitattu 1.3.2025].

Smaalders, B. 2006. Performance Anti-Patterns: Want your apps to run faster? Here's what not to do. *Queue* 4, 44–50. Verkkolehti. Saatavissa: https://doi.org/10.1145/1117389.1117403 [viitattu 2.3.2025].

StatCounter. 2025a. Desktop Operating System Market Share Worldwide. WWW-dokumentti. Saatavissa: https://gs.statcounter.com/os-market-share/desktop/worldwide [viitattu 2.3.2025].

StatCounter. 2025b. Mobile Operating System Market Share Worldwide. WWW-dokumentti. Saatavissa: https://gs.statcounter.com/os-market-share/mobile/worldwide [viitattu 2.3.2025].

The Khronos Group Inc. s.a. LOW-LEVEL 3D GRAPHICS API BASED ON OPENGL ES. WWW-dokumentti. Saatavissa: https://www.khronos.org/api/index_2017/webgl [viitattu 2.3.2025].

Tutkimuseettinen neuvottelukunta. 2024. Hyvä tieteellinen käytäntö (HTK). WWW-dokumentti. Saatavissa: https://tenk.fi/fi/hyva-tieteellinen-kaytanto-htk [viitattu 27.2.2025].

Unity Technologies. 2022. Ultimate guide to profiling Unity games. PDF-dokumentti. Saatavissa: https://unity.com/resources/ultimate-guide-to-profiling-unity-games [viitattu 21.3.2025]

Unity Technologies. 2023. Optimize your mobile game performance (Unity 2022 LTS). PDF-dokumentti. Saatavissa: https://unity.com/resources/optimize-mobile-game-performance-unity-2022lts [viitattu 17.4.2025]

Unity Technologies. 2025a. Platform development. iOS. Building and delivering for iOS. Build an iOS application. Unity User Manual 2022.3 (LTS). WWW-dokumentti. Saatavissa: https://docs.unity3d.com/2022.3/Documentation/Manual/iphone-BuildProcess.html [viitattu 2.3.2025].

Unity Technologies. 2025b. Platform development. Android. Developing for Android. Testing and debugging. Debug on Android devices. Unity User Manual 2022.3 (LTS). WWW-dokumentti. Saatavissa: https://docs.unity3d.com/2022.3/Documentation/Manual/android-debugging-on-an-android-device.html [viitattu 2.3.2025].

Unity Technologies. 2025c. Working in Unity. Analysis. Memory in Unity. Managed memory. Garbage collector overview. Incremental garbage collection. Unity User Manual 2022.3 (LTS). WWW-dokumentti. Saatavissa: https://docs.unity3d.com/2022.3/Documentation/Manual/performance-incremental-garbage-collection.html [viitattu 4.4.2025].

Unity Technologies. 2025d. Graphics. Meshes. Level of Detail (LOD) for meshes. Unity User Manual 2022.3 (LTS). WWW-dokumentti. Saatavissa: https://docs.unity3d.com/2022.3/Documentation/Manual/LevelOfDetail.html [viitattu 17.4.2025]

Unity Technologies. 2025e. Working in Unity. Analysis. Profiling tools. Unity User Manual 2022.3 (LTS). WWW-dokumentti. Saatavissa:

https://docs.unity3d.com/2022.3/Documentation/Manual/performance-profiling-tools.html [viitattu 1.3.2025].

Unity Technologies. 2025f. Platform development. Cross-platform features and considerations. Unity Remote. Unity User Manual 2022.3 (LTS). WWW-dokumentti. Saatavissa: https://docs.unity3d.com/2022.3/Documentation/Manual/UnityRemote5.html [viitattu 2.3.2025].

Viljamaa, H. 2017. Unity3D, palvelimet ja mobiilipelin optimointi. Oulun ammattikorkeakoulu. Tietotekniikan koulutusohjelma. Opinnäyte. PDF-dokumentti. Saatavissa: https://urn.fi/URN:NBN:fi:amk-201705097296 [viitattu 2.3.2025].

Whaley, D. 2020. What 64-Bit Android Apps Mean for the Future of Mobile. Blogi. Julkaistu 7.10.2020. Saatavissa: https://newsroom.arm.com/blog/android-64bit-future-mobile [viitattu 20.3.2025].

Winmind Appi s.a. WinMind Oy. WWW-dokumentti. Saatavissa: https://www.winmind.fi/materials-for-learning [viitattu 28.2.2025].

WebAssembly Community Group s.a. WebAssembly. WWW-dokumentti. Saatavissa: https://webassembly.org/ [viitattu 22.3.2025].

Yin, R. K. 2002. Case Study Research: Design and Methods. 3. painos. Thousand Oaks, California: Sage Publications Inc.