Jere Tienhaara

Käyttöliittymien optimointi Unity-pelimoottorilla

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikka

Insinöörityö

11.5.2022

Tiivistelmä

Tekijä:	Jere Tienhaara
Otsikko:	Käyttöliittymien optimointi Unity-pelimoottorilla
Sivumäärä:	72 sivua
Aika:	11.5.2022
Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Tieto- ja viestintätekniikka
Ammatillinen pääaine:	Pelisovellukset
Ohjaaja:	Lehtori Antti Laiho

Insinöörityössä tutkittiin videopelien käyttöliittymiä ja hyviä tapoja optimoida Unitylla kehitettyjä käyttöliittymäratkaisuja. Erityisesti tutkinnan kohteena oli suorituskyvyn optimointi suoritusajan ja muistiallokaation suhteen. Raportti myös taustoittaa käyttöliittymien merkitystä videopeleissä ja eri pelilajien ja -laitteiden vaikutusta käyttöliittymiin ja yleistä optimointia Unity-ympäristössä.

Insinöörityötä varten kehitettiin erilaisia testitapauksia, joista saatiin testituloksia analysoitavaksi Unityn sisäänrakennetun profilointityökalun avulla. Testituloksia kerättiin ajamalla testejä sekä mobiililaitteella että tietokoneella. Testitulosten perusteella pystyttiin tekemään johtopäätöksiä optimaalisista tavoista ratkoa erinäisiä käyttöliittymäohjelmoijan yleisiä ongelmia. Suuri osa havainnoista pätee myös muihin peliohjelmoinnin osa-alueisiin.

Tärkeimmiksi tekijöiksi testeissä osoittautuivat raskaiden operaatioiden välttäminen silmukoissa ja Unity UI -työkalun ominaisuuksien maltillinen käyttäminen. Insinöörityössä havaittiin lisäksi muita tekijöitä optimointiin liittyen suorituskyvyn lisäksi, kuten koodin helppolukuisuus ja optimoinnin tarpeen tulkitseminen.

Testitapausten lisäksi insinöörityössä tutkittiin hyväksi todettuja käytäntöjä ja optimointivinkkejä alan kirjallisuudesta. Esimerkiksi käyttöliittymien animointi on varsin raskasta Unityn animointijärjestelmää käyttäen ja sen tilalle suositeltiin Tween-kirjastojen käyttöä.

Insinöörityön lopputuloksena syntyi tietopaketti aloitteleville ja kokeneemmillekin Unity-ohjelmoijille, jotka haluavat kehittää optimoituja videopelejä ja käyttöliittymiä.

Avainsanat: videopelit, pelikehitys, käyttöliittymät, optimointi, Unity

Abstract

Author:	Jere Tienhaara
Title:	Optimisation of user interfaces with Unity
Number of Pages:	72 pages
Date:	11 May 2022
Degree:	Bachelor of Engineering
Degree Programme:	Information Technology
Professional Major:	Game Applications
Supervisor:	Antti Laiho, Senior Lecturer

The thesis is about user interfaces of video games and how to optimise them with the Unity game engine. The thesis focuses mainly on the optimisation of performance and use execution time and memory allocation as a measurement for performance. The report also speaks about the importance of user interfaces in video games and how different genres and gaming devices affect user interfaces. In addition, thesis covers an optimised way of implementing simple animations for the user interface.

Various kinds of test cases were developed for this final year project and tests were profiled with Unity Profiler to gather research information and data to process. With processing the results there were conclusions made about the most optimal ways to solve common problems that an engineer faces when working with implementations of user interfaces. Most of the conclusions apply to other parts of game programming as well.

In conclusion the most valuable and efficient optimisation ways were avoiding heavy operations on the often-repeating loops and moderate usage of Unity UI tools. However, other factors besides pure performance gains need to be considered, for example, the readability of the code and the reasonable necessity of optimising.

Keywords: Video Games, Game Development, User Interface, Optimisation, Unity

Sisällys

1 Johdanto

Insinöörityössä tutkitaan ja selvitetään mahdollisimman tehokkaita ja optimoituja tapoja toteuttaa käyttöliittymiä videopeleihin käyttäen Unity-pelimoottoria. Havaintoja pyritään tekemään optimoinnista ohjelman suorituskyvyn optimoinnista, mutta myös koodin helppoluettavuudesta, ohjelmoijan työmukavuudesta sekä käyttöliittymien skaalautumisesta. Insinöörityössä ei käsitellä vaihe vaiheelta perusteista lähtien, kuinka käyttöliittymiä toteutetaan Unitylla, vaan oletetaan, että lukija on päässyt alkuun toteutuksensa kanssa, ja annetaan sen jatkeeksi työssä tutkittua tietoa suorituskyvyn optimoinnista ja hyvistä käytännöistä. Insinöörityö ei kuitenkaan myöskään pureudu kaikkiin edistyksellisiin osa-alueisiin, kuten grafiikkaohjelmointiin.

Videopelin suorituskyvyllä tarkoitetaan sitä, kuinka kauan peliohjelmaa pyörittävällä laitteistolla kestää piirtää yksittäinen ruutu. Ruudun piirtämiseksi laitteiston prosessorin, näytönohjaimen ja muistin on suoritettava niille annetut prosessit ja tehtävät. Videopeleissä pyritään usein suorituskyvyltä ennalta asetettuun minimi- tai tavoitearvoon siinä, kuinka monta ruutua näytölle piirretään sekunnissa. Hyvin yleinen tavoite on 60 kuvaa sekunnissa, joka tarkoittaa, että yhden ruudun piirtämiseen on aikaa noin 16,7 ms. Mitä laajempi videopeli on ja mitä monimutkaisempia ja laskentaa vaativia järjestelmiä se sisältää, sitä oleellisempaa on pelin kaikkien osa-alueiden tehokas optimointi.

Modernissa sovellusarkkitehtuurissa tekninen suorituskyky ei kuitenkaan voi olla ainoa merkittävä tekijä ohjelmointiratkaisuja tehdessä. Videopelien kehittäminen on monimutkaista kehitystyötä ja vaatii hyvin usein useiden työntekijöiden rinnakkaista kehittämistä, jolloin on otettava huomioon koodin luettavuus ja ohjelmoijien työmukavuus. Näitä on osattava punnita yhdessä teknisen suorituskyvyn optimoinnin kanssa, ja hyvän ohjelmoijan on pystyttävä tunnistamaan, milloin on syytä nipistää millisekunnin osia pois suorituskestosta ja milloin taas suorituskyvyn optimointia ei ole syytä lisätä luettavuuden ja työmukavuuden kustannuksella.

Skaalautuminen on toinen oleellinen asia käyttöliittymiä toteuttaessa, ja skaalautumisella voidaan tarkoittaa sitä, kuinka hyvin käyttöliittymä skaalautuu erikokoisille ja -muotoisille näyttöpäätteille, tai sitten skaalautumisella voidaan tarkoittaa sitä, että onnistutaan toteuttamaan dynaamisesti skaalautuva käyttöliittymäelementti, joka ei ole riippuvainen sisältöelementtien määrästä.

Näitä käyttöliittymien toteutuksien optimointeja ja skaalauksia tutkitaan tässä insinöörityössä ja pyitään koostamaan tietopaketti, jonka avulla erityisesti aloittelevat Unity-ohjelmoijat pääsevät hyvään vauhtiin käyttöliittymätoteutuksiensa kanssa, tekemään optimoitua koodia ja ketteriä ratkaisuja.

Kaikki insinöörityössä tehtävät testit ja havainnot tehdään Unityn kehitysympäristössä ja tulosten tarkasteluun käytetään Unityn omaa Profiler-työkalua. Insinöörityössä tehtävät havainnot soveltuvat suurelta osin kaikkiin erilaisiin videopeleihin, joita Unitylla pystyy tekemään eri alustoille. Suurin painoarvo pyritään tutkimustyössä antamaan suorituskyvyn optimoinnille, mutta pyritään myös tekemään havaintoja muista, aiemmin mainituista hyvän ja optimoidun koodin tekijöistä.

2 Käyttöliittymät peleissä

Käyttöliittymät (UI, engl. user interface) ovat sen verran keskeinen teema tässä insinöörityössä, että on hyvä selvittää heti alkuun, mitä käyttöliittymät tarkoittavat videopeleissä. Kielitoimiston sanakirjassa [1] sana “käyttöliittymä” määritellään seuraavasti: “välineet ja toiminnot, joilla käyttäjä viestii tietojärjestelmän kanssa”. Yleiskielessä käyttöliittymä tarkoittaa siis kokonaisuudessaan rajapintaa käyttäjän ja peliohjelman välillä, tapaa välittää syötettä ja tietoa näiden kahden välillä, kuten käyttäjän käyttämän peliohjaimen napin painallus tai näyttöpäätteelle piirrettävä informaatio.

Tässä insinöörityössä käyttöliittymällä tarkoitetaan tarkalleen ottaen graafista käyttöliittymää (GUI, graphical user interface) eli näytölle piirrettäviä graafisia elementtejä, jotka välittävät pelaajalle tietoa pelin tilasta. Micah Bowers [2]

tiivistää videopelien käyttöliittymien tarkoittavan komponentteja, jotka auttavat pelaajaa navigoimaan, löytämään informaatiota ja seuraamaan tavoitteitaan. Näitä komponentteja on lukuisia erilaisia, kuten esimerkiksi elämäpistepalkit, resurssilaskurit, kenttäkartat ynnä muut. Aivan kuten käyttöliittymäsuunnittelu yleisesti applikaatioille, videopelienkin käyttöliittymäsuunnittelu vaatii tarkkaa yksityiskohtiin panostamista ja toimivuuden hahmottamista. Käyttöliittymiin liittyy olennaisesti myös käytettävyys ja käyttäjäystävällisyys (UX, engl. user experience), mutta ne liittyvät erityisesti käyttöliittymien graafisen suunnittelun puolelle, eivätkä niinkään koodin optimointiin, joten ei juuri käsitellä tämän luvun jälkeen. Tästä lähtien käytettäessä termiä käyttöliittymä tarkoitetaan nimenomaan videopelin graafista käyttöliittymää. Unity-pelimoottori käyttää terminologiassaan lyhennettä UI (user interface, käyttöliittymä) tarkoittaen graafisia käyttöliittymiä.

2.1 Käyttöliittymien merkitys ja historia

Videopelien kehitys koostuu lukuisista eri elementeistä ja työvaiheista. Käyttöliittymien suunnittelu ja toteutus eivät aina nauti suurinta suosiota, kun listataan eri kehitysvaiheita. Aihe on kuitenkin erittäin tärkeä, koska käyttöliittymien toimivuus on suuressa roolissa koko pelin miellyttävyydessä.

Yksi yleinen ja tunnistettava käyttöliittymäelementti on pelin päävalikko. Se auttaa pelaajaa navigoimaan haluamaansa sisältöön pelin käynnistyessä. Päävalikko on usein ensimmäinen asia, jonka pelaaja näkee pelin käynnistettyään, joten voi ajatella päävalikon olevan pelin käyntikortti. On hyvin tärkeää, että tämän käyntikorttina toimivan valikon tyyli, selkeys ja luettavuus ovat erinomaista laatua. Pelaajan on hyvä saada heti kiinni pelin tyylistä ja tunnelmasta, joten tarkoituksenmukainen graafinen tyyli päävalikossa on varsin tärkeää.

Videopelit voidaan nähdä informaatioteknologian alalajina, ja informaatiota nykyajan videopelit todella sisältävätkin. Pelit voivat tätä nykyä olla hyvinkin laajoja kokonaisuuksia ja sisältää lukuisia datarakenteita, pelin sisäisiä muuttujia sekä tilastoja, ja näitä kaikkia olisi esitettävä pelin pelaajalle selkeästi ja

johdonmukaisesti. Informaation visuaalinen esittäminen on hienovaraista työskentelyä, ja siinä täytyy tasapainotella taitavasti, niin että ei uuvuta pelaajaa liian suurella informaation määrällä, mutta kuitenkin kaikki oleellinen tulee näytettyä ja siihen vielä pelaajan huomio kohdistettua. Tässä onnistuakseen tarvitsee käyttöliittymän suunnittelijan tuntea visuaalisen hierarkian käsite. Visuaalinen hierarkia [3] tarkoittaa graafisten elementtien keskinäistä tasapainoa koon, värin, fontin ja asettelun suhteen. Visuaalisen hierarkian avulla pyritään tuomaan lukijalle helposti saavutettava ymmärrys siitä, missä järjestyksessä tarjottu informaatio tulee lukea ja mihin kiinnittää milloinkin huomionsa.

Videopelit ovat olleet valtavirtaa jo yli 50 vuotta, ja pelien käyttöliittymät ovat muuttuneet historian saatossa. Käyttöliittymien tyylien ja toteutusten muutokset ovat johtuneet pitkälti pelien kasvaneesta laajuudesta, graafisen suorituskyvyn noususta, resoluution kasvusta ja vaihtuvista tyylitrendeistä. 1970-luvun ja 80-luvun tyylit ja toteutukset olivat vielä hyvin yksinkertaisia: ne tarjosivat pelaajalle kriittisimmän informaation ja usein tekstimuodossa, satunnaisia kuvakkeita käyttäen. 1990-luvulla pelien genret monipuolistuivat, kotikonsolit yleistyivät ja samalla pelien käyttöliittymät rikastuivat [4].

Viimeisen 20 vuoden ajan videopelien kasvaminen ja monipuolistuminen ovat jatkuneet nopeasti, joten peleissä nähtävät käyttöliittymät ovat monipuolistuneet todella merkittävästi. Erityisesti viimeisen 10 vuoden ajan omalaatuiset graafiset tyylit peleissä ja pelisarjoissa ovat yleistyneet, ja tätä nykyä saadaan nauttia hyvin näyttävistä, ilmeikkäistä ja persoonallisista käyttöliittymistä videopeleissä.

2.2 Hyvän käyttöliittymän palaset

Käyttöliittymäsuunnittelija Steph Chow kertoo Game Developers Conference -puheessaan [5], että pelin käyttöliittymän tarkoitus on tuoda pelaajalle mahdollisuus interaktioon pelin kanssa, mutta ei luoda häiriötä tai harhautusta. Toisella tapaa voisi sanoa, että hyvin toteutettua käyttöliittymää ei pelaaja ajattelekaan, mutta huono käyttöliittymä häiritsee pelaajan pelikokemusta. Ei ole yksiselitteistä, mistä palasista koostuu hyvä tai huono käyttöliittymä, mutta voidaan

puhua yleisesti hyväksytyistä suosituksista ja riskeistä, mitä tulee käyttöliittymien suunnitteluun ja toteutukseen. Olisi suositeltavaa pitää käyttöliittymän visuaalinen hierarkia mahdollisimman koherenttina ja loogisena, pitää yhtä aikaa näkyvien fonttien määrä mahdollisimman vähäisenä, pyrkiä käyttöliittymäelementtien tai -näkymän selkeyteen ja pitää graafinen tyyli yhtenäisenä läpi käyttöliittymän. Kuvassa 1 on esimerkki Overwatch-pelin varsin toimivasta käyttöliittymästä, jossa edellä mainitut asiat on otettu huomioon.

Kuva 1. Overwatch-pelin käyttöliittymä [6].

Vastaavasti riskit käyttöliittymien suunnittelussa löytyvät edellä mainittujen ominaisuuksien vastakohdista, eli huonosta visuaalisesta hierarkiasta, liian suuresta fonttien määrästä, liian monen tyylin sekoituksesta sekä epäselkeästä sommittelusta. Kuvassa 2 on esimerkki Mario Party 8 -pelin käyttöliittymästä, jossa on otettu edellä mainittuja riskejä, joiden seurauksena käyttöliittymä on sekavahko ja vaikealukuinen.

Kuva 2. Mario Party 8 -pelin käyttöliittymä [7].

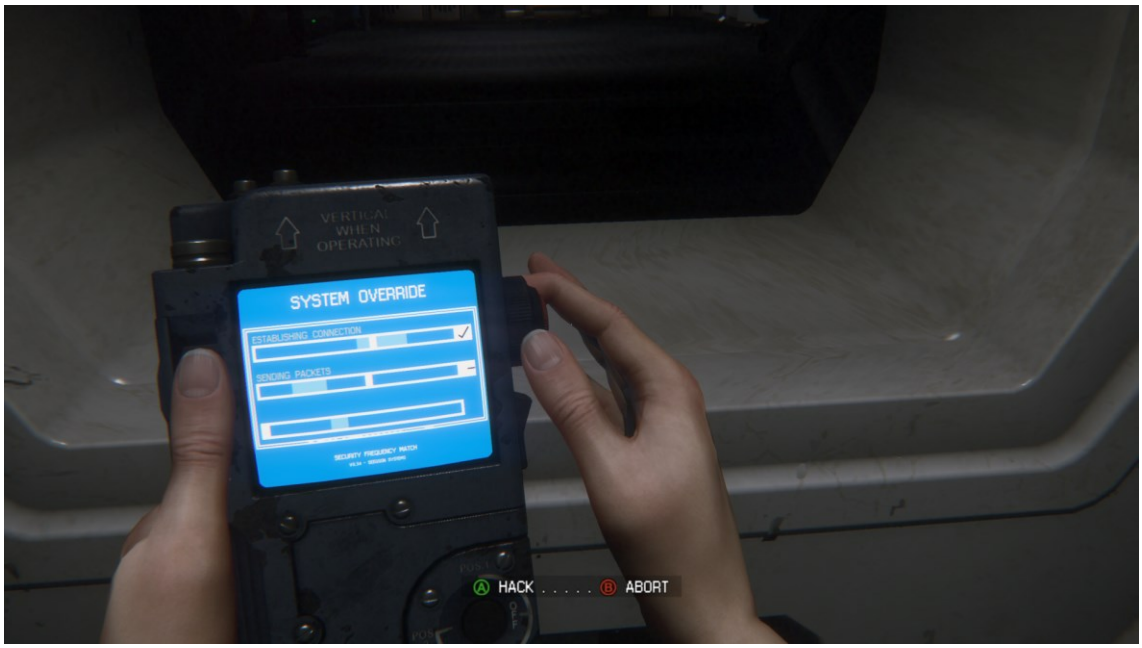
Onnistuneen käyttöliittymän ja pelikokemuksen kannalta on tärkeää esitellä oikea informaatio pelaajalle oikeaan aikaan, eli juuri sillä hetkellä, kun kyseistä informaatiota voidaan olettaa tarvittavan [8, s. 269].

2.3 Suunnittelutyylit

Peleissä esitettävän informaation laatu ja sisältö on hyvin vaihtelevaa. Osa informaatiosta näytetään pelinkulun aikana, ja osa informaatiosta näytetään, kun pelinkulku on tauolla. Yleisimpiä esimerkkejä pelien käyttöliittymissä näytettävän informaation laaduista ovat numeerinen data, sanallinen informaatio, erilaiset palkit, statussymbolit, taitopuut, kartat ja hahmovalinnat. Lisää esimerkkejä voisi keksiä lähes loputtomiin. Monipuolisen esiteltävän informaation ja vaihtelevien kontekstien vuoksi on erittäin kätevää, että käyttöliittymien suunnitteluun on muodostunut jo vakiintuneita suunnitteluratkaisuja ja -malleja.

”Neljäs seinä” on konsepti, joka tunnetaan elokuvien maailmassa, mutta sitä voi hyödyntää myös videopelien käyttöliittymien suunnittelussa [9]. Kuvitteellisen seinän tarkoitus on rajata, mitkä käyttöliittymien elementit sijaitsevat

pelimaailmassa ja mitkä irrallaan siitä, vain pelin pelaajalle näkyen. Usein videopelien käyttöliittymistä puhuessa termi, jolla tätä efektiä kuvataan, on diegeettisyys. Jos käyttöliittymän elementti on diegeettinen, se tarkoittaa, että elementti sijaitsee pelimaailmassa ja on kuvitteellisesti nähtävissä pelihahmojenkin silmin [9]. Kuvassa 3 näkyy esimerkki pelistä Alien Isolation, jossa diegeettisenä käyttöliittymäelementtinä toimii pelimaailman oman teknisen laitteen käyttöliittymä, joka kertoo informaatiota yhtäaikaisesti videopelin pelaajalle ja myös kuvitteelliselle pelihahmolle.



Kuva 3. Diegeettinen käyttöliittymäelementti pelissä Alien Isolation [10].

Vastakohtana diegeettiselle käyttöliittymälle toimii ei-diegeettinen, jossa ajatellaan käyttöliittymän näkyvän vain pelaajalle eikä se ole sijoitettuna pelimaailmaan mukaan. Yleisimpiä esimerkkejä tällaisista käyttöliittymistä ovat valikot ja peliruudun reunoilla kerrottava info eli niin sanottu heijastusnäyttö [9].

Edellä mainittujen suunnitteluratkaisujen välimaastosta löytyy kaksi välimuotoa. Spatiaaliseksi elementiksi kutsutaan sellaista, joka on sijoitettu pelimaailmaan, mutta sen ei ajatella kuuluvan pelin tarinaan tai näkyvän pelin hahmoille. Spatiaalisia elementtejä videopeleissä ovat esimerkiksi autopeleistä tutut ohjaavat

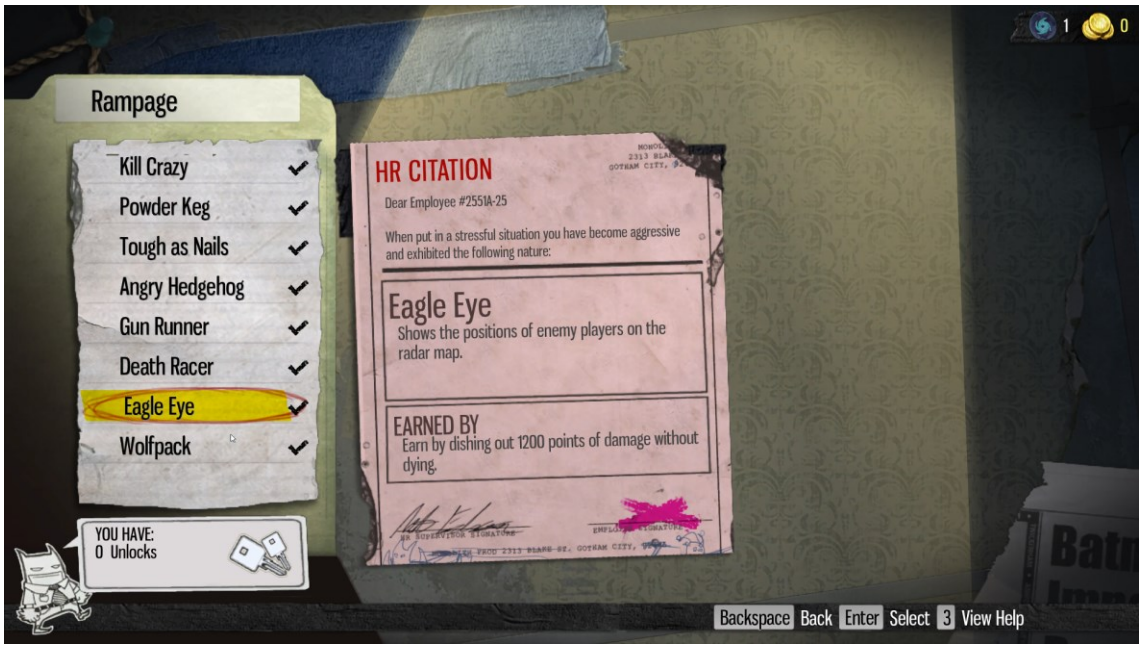
nuolet radan pinnassa. Toinen välimuoto kantaa nimeä meta, joka taas tarkoittaa elementtiä, jota ei ole sijoitettu pelimaailmaan, mutta sen kuitenkin ajatellaan näkyvän vastaavanlaisena pelihahmolle [9]. Tämmöisiä elementtejä ovat usein erilaiset pelinsisäiset puhelimet ja tietokoneet näyttöineen, varsinkin kolmannesta persoonasta kuvatuissa peleissä, kuten kuvan 4 pelissä Watch Dogs.

Kuva 4. Meta-tyylinen käyttöliittymäelementti pelissä Watch Dogs [11].

Käyttöliittymäelementin fyysisen ja tarinallisen sijainnin lisäksi toinen oleellinen suunnitteluvalinta on elementin graafinen tyyli. Kahtiajakoisuutta löytyy tästäkin aiheesta, sillä hyvin yleisesti graafiset tyylit jaetaan kahteen kategoriaan, jotka ovat toistensa vastakohdat – skeuomorfismi ja flat design [8].

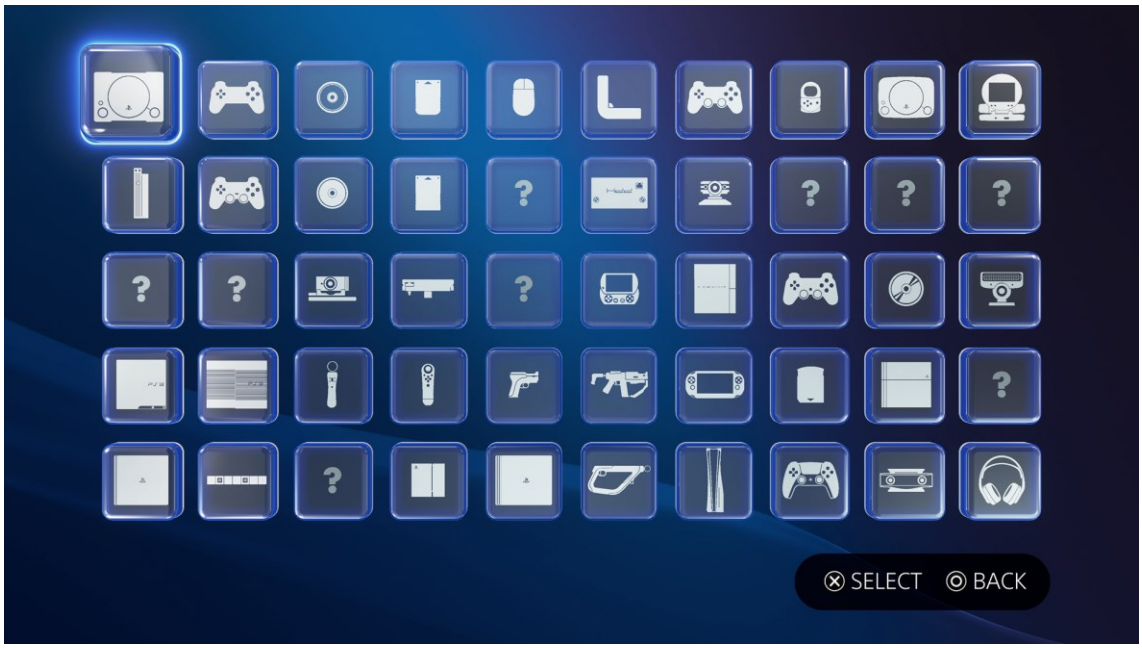
Skeuomorfismista voidaan käyttää toisena terminä ”siirtomuotoisuutta.” Tämä tyylisuuntaus tarkoittaa sitä, että elementeissä käytetään oikeasta maailmasta löytyviä tai niitä jäljitteleviä tekstuureita [12]. Pixel Perfect Precision -verkkojulkaisussa [13] esitellään mielenkiintoinen ajatus, että skeuomorfismi toimisi hyvin tyylisuuntana, jos sovelluksen kohderyhmä on iäkkäämpää ja vähemmän teknologiaorientoitunutta. Tunnettuja esimerkkejä skeuomorfismisesta designista ovat Applen iPhone-puhelinten alkuaikojen natiivisovellusten suunnitteluratkaisut,

joissa muistiosovellus näytti nahkaiselta muistikirjalta ja kalenterisovellus paperikalenterilta. Kuvassa 5 näkyy esimerkki videopelin käyttöliittymästä, joka on toteutettu skeuomorfismin tyyliin.



Kuva 5. Käyttöliittymä pelissä Gotham City Impostors [14].

Vastakohtana skeuomorfismille toimii flat design. Siinä elementit ovat minimalistisia, helppolukuisia ja puhtaan graafisia ilman reaalimaailmaa jäljittteleviä tekstuureita. Kuvassa 6 esiintyy Astro’s Playroom -pelin käyttöliittymä, joka on puhdasta flat designia.



Kuva 6. Käyttöliittymä pelistä Astro’s Playroom [15].

Flat design on suosittu tyylisuunta videopelien käyttöliittymissä, ja se valitaan usein, kun täytyy esitellä sekä jäsentää suuri määrä informaatiota ja halutaan maksimoida käyttöliittymän helppolukuisuus.

2.4 Pelilajin vaikutus käyttöliittymiin

Videopeliala on nykyään varsin laaja ja pelejä kehitetään laidasta laitaan. Erilaiset pelilajit vaativat erilaisia ratkaisuja niiden käyttöliittymissäkin. Suosittujen peligenrejen fanit ovat tottuneet jo tiettyihin konventioihin kyseisten pelien käyttöliittymissä, ja niitä työstävien suunnittelijoiden on tärkeää tuntea nämä käytänteet. Osa peleistä panostaa pelaajan kokemaan immersioon, jolloin käyttöliittymän on hyvä olla mahdollisimman huomaamaton, minimalistinen ja häiritsemätön. Toiset pelit taas ovat niin informaation täyteisiä, että saattavat koostua lähes pelkästään käyttöliittymistä.

Näitä genrejä ovat esimerkiksi simulaatio- ja strategiapelit. Kyseisille pelilajeille on ominaista, että peli ei välttämättä pyöri reaaliajassa, vaan pelaajalla on rauhassa aikaa tehdä valintansa ja siirtonsa käyttöliittymien avulla. Nämä pelit esittävät pelaajalle suuren määrän dataa, jonka avulla pelaaja tekee päätöksiään. Visuaalinen hierarkia ja käytettävyysheuristiikat ovat hyvin tärkeitä asioita tällaisissa peleissä [16].

Vastaavasti taisteluareenamoninpelit ja massiiviset monen pelaajan verkkoroolipelit ovat pelityyppejä, jotka suorastaan pursuavat informaatiota, mutta sillä erotuksella, että niihin on usein yhdistetty myös reaaliaikainen interaktiivisuus. Näissä peleissä käyttöliittymäsuunnittelijan on pidettävä huolta, että pelaajan huomio pystyy oikeissa kohdissa oikeissa paikoissa.

2.5 Pelilaitteen vaikutus käyttöliittymiin

Pelin lajityypin lisäksi käyttöliittymäsuunnitteluun vaikuttaa se, minkälaisella laitteella peliä pelataan. Yleisimmät pelilaitteet ovat tietokoneet, konsolit ja mobiililaitteet kuten puhelimet ja tabletit. Jokaisessa on mukana omat haasteensa ja hyötynsä pelikehityksessä ja myös käyttöliittymien tekemisessä. Kolme merkittävää tekijää laitteen vaikutuksessa ovat näytön fyysinen koko, katseluetäisyys ja käytettävissä olevat ohjauslaitteet [17].

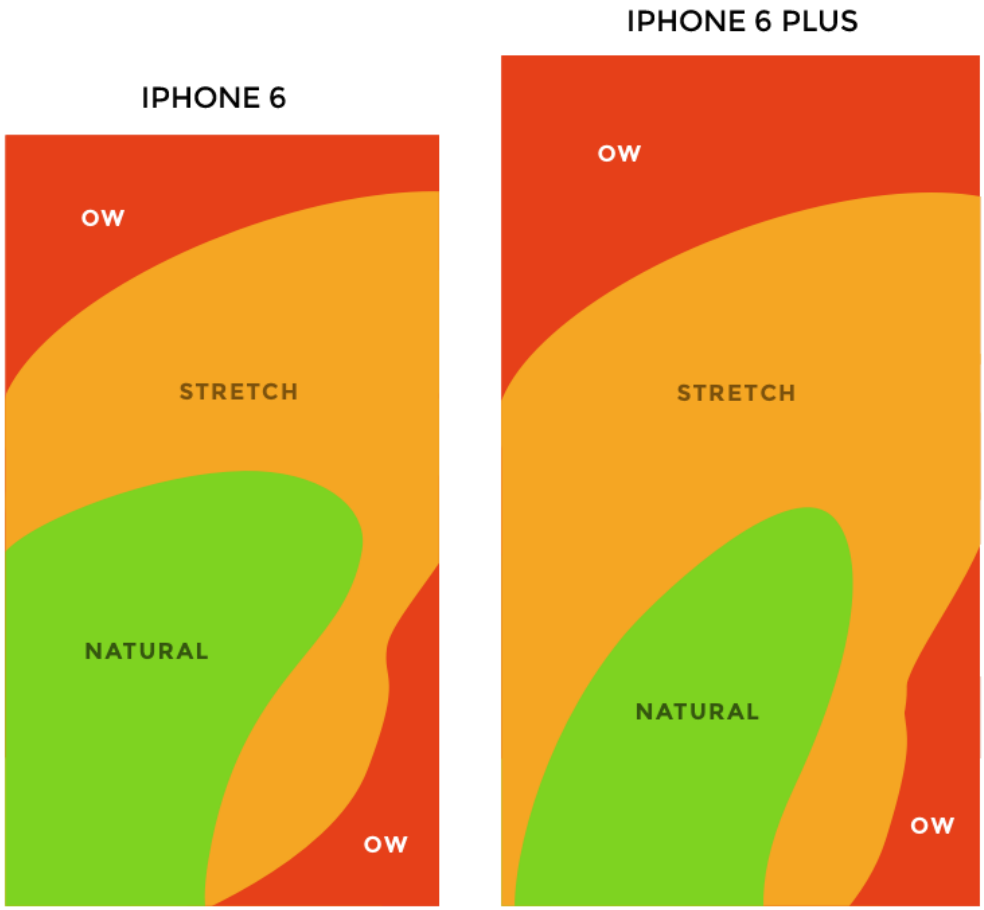
Tietokonepeleissä pelaajan ja näytön välinen etäisyys on yleensä melko lyhyt, jolloin käyttäjät todennäköisemmin pystyvät lukemaan pienempääkin tekstiä näyttöpäätteeltä. Sen sijaan konsoleita pelataan usein sohvalta käsin ja televisiosta katsoen, joten suuremman katseluetäisyyden myötä nousee ongelmaksi liian pieni fonttikoko. Tietokoneella ohjauslaitteita voi olla laidasta laitaan, yleisin kuitenkin on näppäimistön ja hiiren yhdistelmä, joka mahdollistaa tarkan ja monipuolisen navigoinnin valikoissa hiiren avulla. Konsolit taas käyttävät lähes aina erillisiä ohjaimia, jolloin niillä pelattavien pelien valikoiden täytyy olla helposti navigoitavissa ohjaimen tarjoamilla painikevaihtoehdoilla.

Mobiilipelit tarjoavat eniten omanlaisia haasteitaan ja erikoisuuksiaan käyttöliittymien toteutuksessa. Mobiilipelien pelaajien laitekanta on hyvin laaja, mobiililaitteita on hyvin eritasoisia suorituskyvyiltään ja näyttöjen koot sekä kuvasuhteet vaihtelevat huomattavasti enemmän kuin esimerkiksi tietokonepelien pelaajilla. Steamin maaliskuussa 2022 julkaiseman tutkimuksen [18] kyselyn perusteella 68,88 % tietokonepelaajista pelasi 1920 x 1080 pikselin resoluutiolla ja vastaavan kuvasuhteen 16:9 näytöillä pelasi yhteensä vähintään 88,21 % pelaajista. Nykykonsoleilla voidaan olettaa kuvasuhteen 16:9 kattavuuden olevan vähintään samaa luokkaa. Mobiililaitteissa tilanne on hyvin eri, sillä vaikka silmämääräisesti suurin osa puhelimista on suunnilleen samanmuotoisia, on todellisuudessa näyttöjen kuvasuhteissa ja resoluutioissa yllättävän paljon pieniä eroavaisuuksia. Puhelimissa yleisiä kuvasuhteita ovat 16:9:n lisäksi 18:9, 20:9, 17:10, 4:3 sekä uusimpien iPhonejen [19] käyttämä 19.5:9. Tableteissa yleisiä kuvasuhteita ovat esimerkiksi 3:2, 5:3 ja 4:3. Vaihtelevien kuvasuhteiden vuoksi on hyvin tärkeää rakentaa käyttöliittymistä laadukkaasti skaalautuvia, jotta kaikenlaisissa näytöissä kuvakkeet ovat järkevänkokoisia, teksti luettavaa eikä mitään katoa kuvaruudun reunojen yli piiloon.

Muita huomioon otettavia seikkoja mobiilipelien käyttöliittymissä ovat eri puhelinten omat erikoisominaisuudet laitteen suunnittelussa. Esimerkiksi hieman vanhemmista Android-puhelimissa on takaisin-nappula, jonka käyttäminen on hyvä implementoida esimerkiksi valikkorakenteisiin. Moderneissa älypuhelimissa on myös erimallisia ja -kokoisia lovia (engl. notch) näytön yläreunassa, joten käyttöliittymää suunnitellessa on hyvä luoda niin sanottuja turva-alueita yläreunaan, jotta tärkeää informaatiota ei vahingossa jää loven alle piiloon. Kuvassa 7 näkyy esimerkki iPhone X -puhelimen lovesta.

Kuva 7. Lovi iPhone X -puhelimen näytössä [20].

Mobiilipeliä suunnitellessa on hyvä myös ottaa huomioon, kuinka käyttäjä pitää laitetta kädessään. Jos peli on suunniteltu vertikaaliseen orientaatioon, pelaaja saattaa haluta pystyä pelaamaan peliä pitäen puhelinta vain yhdessä kädessä. Horisontaalisessa orientaatiossa taas pelaaja lähes varmasti pitää puhelimestaan kiinni kahdella kädellä. Käyttöliittymäelementtejä sijoittaessa on tärkeää miettiä sekä sitä, että painikkeet ovat saavutettavissa peukalolla, mutta myös sitä, että tärkeää informaatiota ei vahingossa sijoita peukalon tai kämmenen alle. Kuvassa 8 havainnollistuu, mitkä alueet ovat keskimääräisesti peukalolla helppo tai vaikea saavuttaa iPhone 6- ja 6+-puhelimissa.



Kuva 8. Kosketusnäytöllisten puhelinten käyttöalue yhdellä kädellä käytettäessä [21].

Jotkut pelit voivat olla suunniteltuja toimimaan sekä pysty- että vaakasuunnassa. Tällainen peli on esimerkiksi Bethesdan The Elder Scrolls: Blades, jonka käyttöliittymäsuunnittelija Marie Jasmin kertoo mielenkiintoisessa Game Developers Conference -puheessaan [22], kuinka suuria haasteita kahden orientaation yhtäaikainen suunnittelu toi mukanaan, mutta kuinka paljon lopulta pelaajat pitivät toteutuksesta, jossa he voivat joka pelikerralla itse valita, haluavatko pelata lyhyemmän session vertikaalisessa orientaatiossa vai uppoutua peliin immersiivisemmin pelaamalla horisontaalisessa orientaatiossa.

3 Unityn työkalut käyttöliittymiin ja profilointiin

3.1 Unity pelimoottorina

Kuten Unitylla työskentelystä kertovassa “Hands-on, Unity 2021 Game Development” -teoksessa [22] kerrotaan, pelikehityksen historian alussa suurina rajoitteina kehittäjille olivat sekä pelilaitteiden rajalliset suorituskyvyt että myös kehitykseen käytettävien laitteiden, työkalujen ja ohjelmistojen heikko tarjonta, saatavuus ja toiminnallisuus. Teknologia-alan ja siinä ohessa pelialan kehityksen jatkuessa ovat kaikki nämä rajoitteet muuttuneen vuosi vuodelta pienemmiksi ja peliteollisuuden toimijat ovat kehittäneet lukuisia toimivia ratkaisuja pelien kehittämiseen. Yksi mullistavimmista uudistuksista pelikehitykseen on ollut pelimoottorien synty. Pelimoottori tarkoittaa pienistä ohjelmistoista ja työkaluista koostuvaa kokonaisuutta, joka tarjoaa ratkaisuja pelinkehityksen keskeisiin ongelmiin sekä tarpeisiin ja on suunniteltu toimimaan yhtenäisesti näiden osa-alueiden kesken, ja jakamaan saman filosofian [23]. Pelimoottorin kehitys on erittäin vaativa tehtävä, ja yleensä vain suurimpien peliyritysten on mahdollista kehittää oma moottorinsa. Usein nämä yritysten itse kehittämät pelimoottorit ovat vain yrityksen sisäisessä käytössä, toisinaan niiden käyttöoikeuksia myydään yrityksen ulkopuolelle korkeaan hintaan.

Helpommin saavutettavat ja kaikille kehittäjille sekä pienille pelialan yrityksille tarjolla olevat pelimoottorit ovat viimeisien vuosikymmenien aikana nostaneet suosiotaan suuresti. Yksi käytetyimmistä pelimoottoreista on tanskalaislähtöisen Unity Technologiesin Unity-pelimoottori. Ensimmäinen versio Unitysta julkaistiin vuonna 2005 [24] ja Unityn tietojen [25] mukaan nykyisin Unitylla tehdään yli puolet kaikista julkaistavista mobiili-, PC- ja konsolipeleistä. Vuonna 2021 Unitylla tehtyjä pelejä ladattiin 5 miljardia kertaa kuukaudessa ja 72 % suosituimmista mobiilipeleistä oli tehty Unitylla [25].

Hyviä syitä käyttää Unitya ovat esimerkiksi Unityssa käytettävä ohjelmointikieli C#, joka on varsin aloittelijaystävällinen olio-ohjelmointikieli, Unityn kehityksen alusta asti kulkenut panostus yksinkertaisuuteen, helppoon ja viimeisteltyyn

käyttöliittymään ja myös Unity-pelimoottorin ympärillä oleva suuri yhteisö, joka tuottaa paljon ohjeistuksia Unityn käyttämiseen kaikkien luettavaksi ja on aina valmiina antamaan vastauksia muiden kehittäjien kysymyksiin. Lisäksi yksi suurimpia etuja Unityn käytössä on monipuolinen eri käyttöympäristöjen tuki ja tehokkuus mobiilipelien kehityksessä [23]. Unity tukee nykyisin useaa käyttöympäristöä, johon julkaista pelejä, ja Unitylla pystyy kehittämään pelejä mobiililaitteille, PC:lle sekä monille pelikonsoleille. Unityn tietojen mukaan yli 20 alustaa tukee Unitylla kehitettyjä pelejä [25]. Unitylle on lisäksi tarjolla todella suuri määrä erilaisia lisätyökaluja ja kirjastoja, jotka auttavat kehittäjää sellaisissa yleisissä pelikehityksen ongelmissa, joita Unity pelimoottorina ei vielä natiivisti ratkaise.

3.2 Käyttöliittymätoteutukset Unitylla

Unity tarjoaa tällä hetkellä [26] kolmea erilaista ratkaisua käyttöliittymien toteutukseen: UI Toolkit, IMGUI (Immediate Mode Graphical User Interface, eli suomennettuna välittömän tilan graafinen käyttöliittymä) ja Unity UI. Unitylle on saatavilla myös kolmannen osapuolen työkaluja käyttöliittymien toteutukseen, kuten esimerkiksi DoozyUI, mutta tässä insinöörityössä jätetään käsittelemättä nämä ulkopuoliset työkalut.

Unityn UI Toolkit on vielä aktiivisessa kehityksessä oleva käyttöliittymäratkaisu, jonka on tarkoitus olla tulevaisuudessa Unityn suosittelema tapa toteuttaa käyttöliittymiä sekä Unity-peleihin, että Unityn editoriin itsessään. IMGUI on työkalu, jolla pystyy nopeasti pelin kehityksen aikana piirtämään tekstiä tai nappuloita pelin ruudulle, mutta sitä ei ole suositeltavaa käyttää lopullisen pelijulkaisun käyttöliittymän toteutukseen. Siihen Unity tällä hetkellä suosittelee Unity UI:ta, joka on myös se ratkaisu, jota tämä insinöörityö käsittelee.

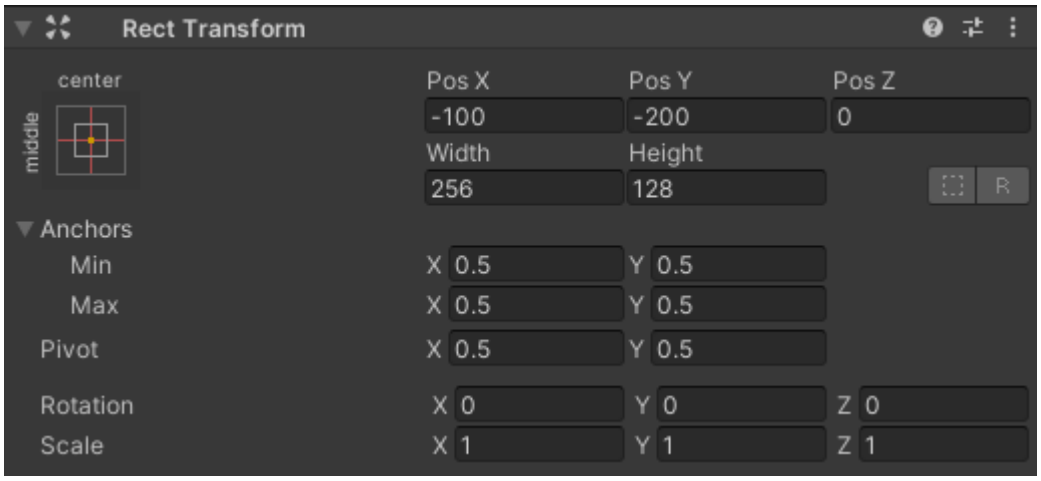
Unity UI:lla ei pysty kehittämään käyttöliittymiä Unityn editoriin, mutta se on monipuolinen ja kätevä työkalu Unitylla tehtävien pelien käyttöliittymien toteutuksiin. Unity UI:n keskeisin elementti on Canvas (suomeksi kangas, piirtoalue tai kanvaasi, työssä käytetään tästä lähtien termiä kanvaasi). Jokaisen

käyttöliittymäelementin täytyy sisältyä johonkin kanvaasiin [27, s. 67]. Kanvaasin luonnin yhteydessä Unity luo automaattisesti EventSystem-nimisen objektin, jos sellaista ei vielä löytynyt hierarkiasta. Tämä objekti sisältää oletuksena EventSystem- ja Standalone Input Module -komponentit. Niiden avulla hoituu käyttöliittymäelementtien interaktiivisuus, kuten painikkeiden valinta ja käyttäjän syöttölaitteen lukeminen [27, s. 168].

Käyttöliittymäelementtejä, joita Unity UI:n avulla voidaan luoda, ovat teksti, kuva, painike, kytkin, liukusäädin, vieritysnäkymä, vierityspalkki, alasvetovalikko, syötekenttä ja paneeli. Kaikista näistä voidaan tehdä dynaamisia editorin tai koodin avulla. Seuraavaksi käydään kevyesti läpi Unity UI -järjestelmän keskeisiä komponentteja ja asetuksia.

Rect Transform -komponentti

GameObject on yksi Unity-kehityksen keskeisimpiä käsitteitä, ja sillä tarkoitetaan kaikkia objekteja, jotka löytyvät hierarkianäkymästä. Jokaisella GameObjectilla on olemassa joko Transform- tai RectTransform-komponentti. Rect Transform löytyy kaikilta kanvaasin alla olevilta GameObjecteilta, toisin sanoen kaikilta Unity UI:lla tehdyiltä käyttöliittymäelementeiltä. Rect Transform -komponentin avulla säädellään käyttöliittymäelementtien kokoa ja sijaintia suhteessa kanvaasiin ja objektin vanhempaan. Rect Transform -komponentissa ovat tavallisesta Transform-komponentista tutut sijainti, rotaatio ja skaala, mutta niiden lisäksi ankkuroinnin esimääritettyjä tiloja, elementin leveys ja korkeus pikseleinä, ankkureiden minimi- ja maksimiarvot sekä pivot-pisteen sijainti [27, s. 85]. Kuvassa 9 näkyy esimerkki käyttöliittymäelementin Rect Transform -komponentista.



Kuva 9. Käyttöliittymäelementin Rect Transform -komponentti [28].

Canvas Render Mode -asetus

Käyttöliittymäelementin keskeinen asetus sijaitsee Canvas-komponentissa ja on nimeltään “render mode” eli renderöintitila [27, s. 101–108]. Asetus tarjoaa kolme vaihtoehtoa:

- World Space
- Screen Space – Overlay
- Screen Space – Camera.

World Space -renderöintitilan kanvaasit sijoittuvat pelimaailman sisälle ja piirretään ruudulle vain, jos kyseistä kohtaa pelimaailmasta kuvataan aktiivisella kameralla. Screen Space -vaihtoehdot taas piirretään nimenmukaisesti suoraan ruudulle, erotuksena vain, että Overlay-asetuksella kanvaasi piirretään aina koko ruudun päälle ja Camera-asetuksella kanvaasi piirretään kameran kautta, minkä myötä voi valita käytettävän kameran sekä muuttaa elementtien etäisyyttä kamerasta. Screen Space – Overlay on tässä asetuksessa oletustila ja se, jonka tyyppisiä käyttöliittymäelementtejä käsitellään tässä työssä. Kanvaasin renderöintitila myös määrittää luvussa 2 mainitun käyttöliittymäelementin diegeettisyyden. Ei-diegeettiset ja metatyyliset elementit toteutetaan Screen Space -renderöintitilassa ja diegeettiset sekä spatiaaliset elementit World Space -tilassa.

Renderöintitila vaikuttaa siihen, onko itse Canvas-komponentin sisältävän GameObjectin RectTransform kehittäjän hallittavissa vai ei. World Space -asetus avaa RectTransformin kehittäjälle säädettäväksi, jotta käyttöliittymäelementti saadaan sijoitettua haluttuun kohtaan, esimerkiksi pelihahmon yläpuolelle. Screen Space Camera -tila avaa Canvas-komponentin kautta z-akselin sijainnin muokattavaksi. Screen Space Overlay -tilassa Rect Transformin hallinta tapahtuu täysin automaattisesti Canvas-komponentin avulla ja käyttäjä itse ei voi muuttaa Rect Transformin arvoja.

Canvas Scaler -skaalauskomponentti

Canvas Scaler -komponentti luodaan automaattisesti, kun hierarkiaan lisää uuden Canvas-peliobjektin. Tämä skaalauskomponentti hoitaa kanvaasin sisällä olevien elementtien skaalauksen [27, s. 110–120]. Yksittäiselle resoluutiolle kehittäessä tätä komponenttia ei välttämättä tarvitse, mutta erityisesti mobiilipelejä kehittäessä se on käyttöliittymätoteutuksen yksi tärkeimmistä palasista, sillä näyttöjen resoluutiot vaihtelevat suuresti laitteiden välillä.

Skaalauskomponentin keskeinen asetus on UI Scale Mode eli skaalaustila, joka sisältää neljä erilaista vaihtoehtoa:

- World
- Constant Physical Size
- Constant Pixel Size
- Scale With Screen Size.

Vaihtoehto World on valittavana ainoastaan, jos kanvaasin renderöintitilana toimii World Space, ja tässä tapauksessa World on ainoa vaihtoehto skaalaustilaksi.

Constant Pixel Size on tarkoitettu yksittäisen resoluution käyttöliittymille, ja tällä asetuksella elementit säilyttävät aina alkuperäisen pikselikokonsa riippumatta näytön resoluutiosta. Tämänkin asetuksen kanssa skaalauskomponenttia voi

kuitenkin käyttää säätämään kanvaasin skaalauskerrointa tai referenssipikselien määrää pelin mittayksikölle.

Constant Physical Size -asetuksella kanvaasin elementit pitävät aina saman fyysisen koon, riippumatta näytön koosta tai resoluutiosta.

Scale With Screen Size -asetuksella skaalauskomponentti skaalaa referenssiresoluution mukaisesti käyttöliittymän elementit erikokoisille ruuduille. Tämä asetus valittuna tulee valittavaksi toinen asetus, nimeltään Screen Match Mode. Sillä säädellään vielä tarkemmin, kuinka skaalaus toteutetaan. Vaihtoehdot ovat Expand, Shrink ja Match Width Or Height. Expand-asetus venyttää kanvaasia, jos ruutu on pienempi kuin referenssiresoluutio. Shrink-asetus taas pienentää kanvaasia, jos ruutu on suurempi kuin referenssiresoluutio. Match Width Or Height -asetus säätää leveys-korkeusliukusäätimen mukaisesti käyttöliittymää leveyden tai korkeuden suhteen. Liukusäätimen arvot ovat väliltä 0–1. Ashley Godblod suosittelee teoksessaan ”Mastering UI Development With Unity” [27, s. 117] käyttämään arvoa 0 eli leveys, jos peliä pelataan laite pystyasennossa, ja arvoa 1 eli korkeus, jos peliä pelataan laite vaakatasossa.

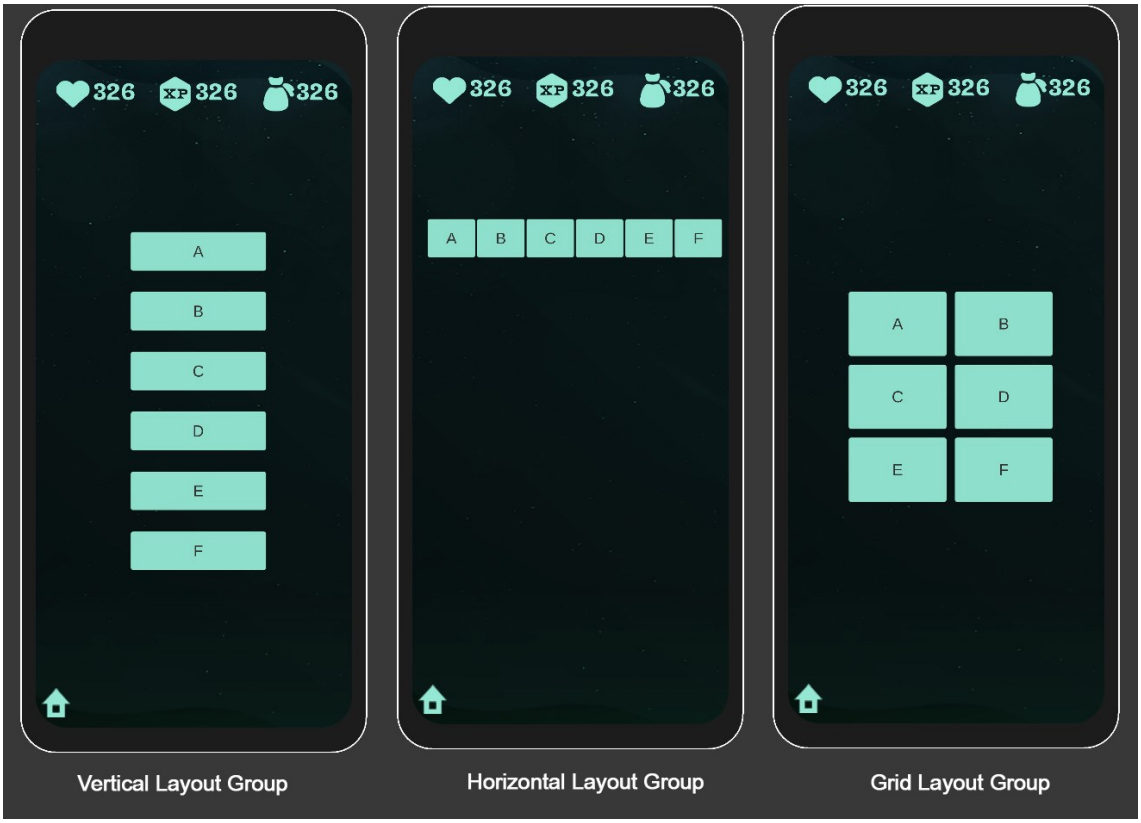
Erinomainen työkalu skaalausten toimivuuden tarkasteluun on Unityn Device Simulator -paketti, jonka asennettua pystyy ajamaan omaa peliään editorissa erilaisilla laitteilla ja näkemään realistisen kuvan, miten peli ja käyttöliittymä skaalautuvat kyseiselle resoluutiolle ja jääkö elementtejä piiloon reunojen tai loven taakse [29].

Automatic Layout Groups -komponentit

Unityn UI-järjestelmä tarjoaa työkaluja myös layoutin eli elementtien asettelun automatisointia varten [27, s. 190]. Näitä komponentteja on kolmenlaisia:

- Horizontal Layout Group
- Vertical Layout Group
- Grid Layout Group.

Kun hierarkiassa kanvaasin alla olevaan GameObjectiin lisää jonkin näistä komponenteista, kyseinen komponentti säätää kaikkien hierarkian seuraavan tason objektien RectTransform-asetukset automaattisesti. Kuvassa 9 näkyy esimerkkejä siitä, minkälaisia asetteluja automaattiset layout -komponentit tekevät lapsiobjekteille.



Kuva 9. Esimerkkejä automaattisen layout-ryhmien käytöstä [28].

Horizontal Layout Group asettelee elementit vaakatasoon vierekkäin, Vertical Layout Group pystysuuntaan päällekkäin ja Grid Layout Groupin avulla pystyy asettelemaan elementtejä ruudukkoon.

Layout Element -komponentti

Layout Element -komponentti liittyy vahvasti Automatic Layout Groups -komponentteihin, ja sen voi lisätä GameObjectiin, joka on jonkin automaattisen layout-ryhmän jäsen ja jonka avulla voi tehdä lisäasetuksia siihen, kuinka

elementti reagoi Layout Groupin määräämiin Rect Transform -muutoksiin [27, s. 215].

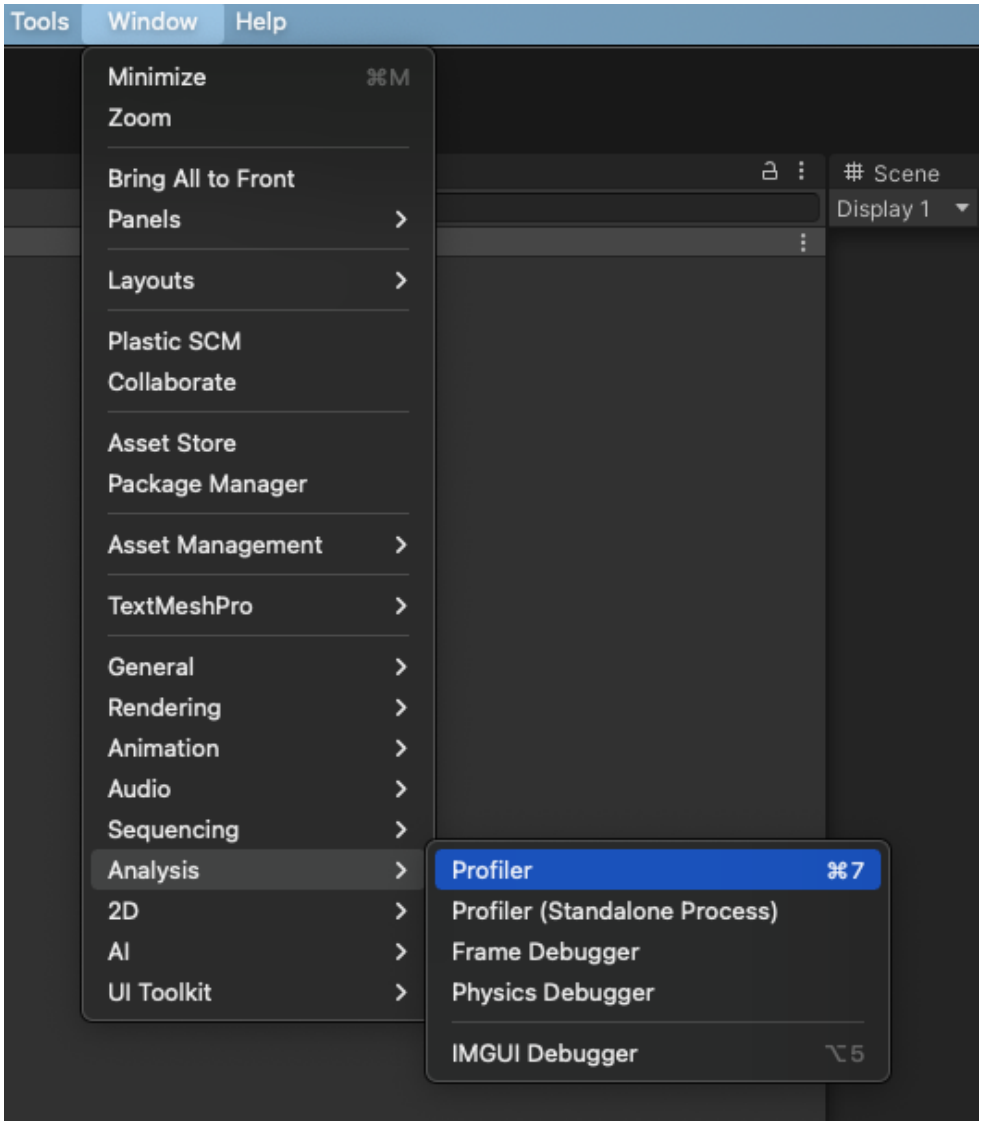
Ensimmäistä asetusta, Ignore Layout, voi käyttää ohittamaan Layout Groupilta tuleva asettelu ja vapauttamaan kyseisen GameObjectin Rect Transform -komponentti kehittäjän säädettäväksi.

Min Width ja Min Height määrittävät minimileveyden ja -korkeuden elementin koolle, jota pienemmäksi elementtejä ei skaalata. Preferred Width ja Preferred Height ovat hieman monimutkaisia asetuksia, sillä niiden toimivuus riippuu Layout Groupin asetuksista. Preferred Width- ja Preferred Height -asetuksilla saa määritettyä elementeille optimaalisen koon, ja jos Layout Groupin asetuksista ottaa Child Force Expand -asetuksen pois päältä, näillä Layout Elementin asetuksilla pystyy määrittämään maksimaalisen leveyden ja korkeuden elementeille. Flexible Width ja Flexible Height -asetukset määrittävät elementeille prosenttimäärän, jonka elementit pyrkivät täyttämään Layout Groupista suhteessa muihin objekteihin, joilla on vastaava asetus.

3.3 Unity Profiler-työkalu

Voidakseen optimoida suorituskykyä tarvitsee ensin pystyä mittamaan suorituskykyä, ja sitä varten tarvitaan työkaluja. Yksi erinomainen työkalu tähän on pelimoottorin itsensä tarjoama Unity Profiler. Tämä työkalu on rakennettu Unityn editorin sisään ja tarjoaa monipuolisia tapoja seurata pelin suorituskykyä [30, s. 9]. Profileria voi käyttää kokonaisen pelin profilointiin, jolloin se antaa yleiskuvaa pelin suorituskyvystä ja voi tuoda kehittäjän tietoisuuteen niin sanottuja pullonkauloja, eli liian vaativia tehtäviä, jotka hidastavat pelin tasaista kuvanpäivitystä. Profileria voi myös käyttää irrallisten testien tai osa-alueiden profilointiin, ja tämä on juuri sitä, mihin työkalua käytetään tässä insinöörityössä. Kun halutaan vertailla esimerkiksi kahden eriävän toteutustavan tehokkuutta suorittaa sama toiminto, ajetaan molemmat versiot ja Profilerin avulla saadaan tietoa niiden eroavaisuudesta esimerkiksi muistihallinnan ja prosessorin suoritusajan suhteen.

Profiler saadaan editorissa avattua kuvan 10 mukaisesti yläpalkista Window-painikkeen alta löytyvästä alasvetovalikosta kohdasta Analysis.

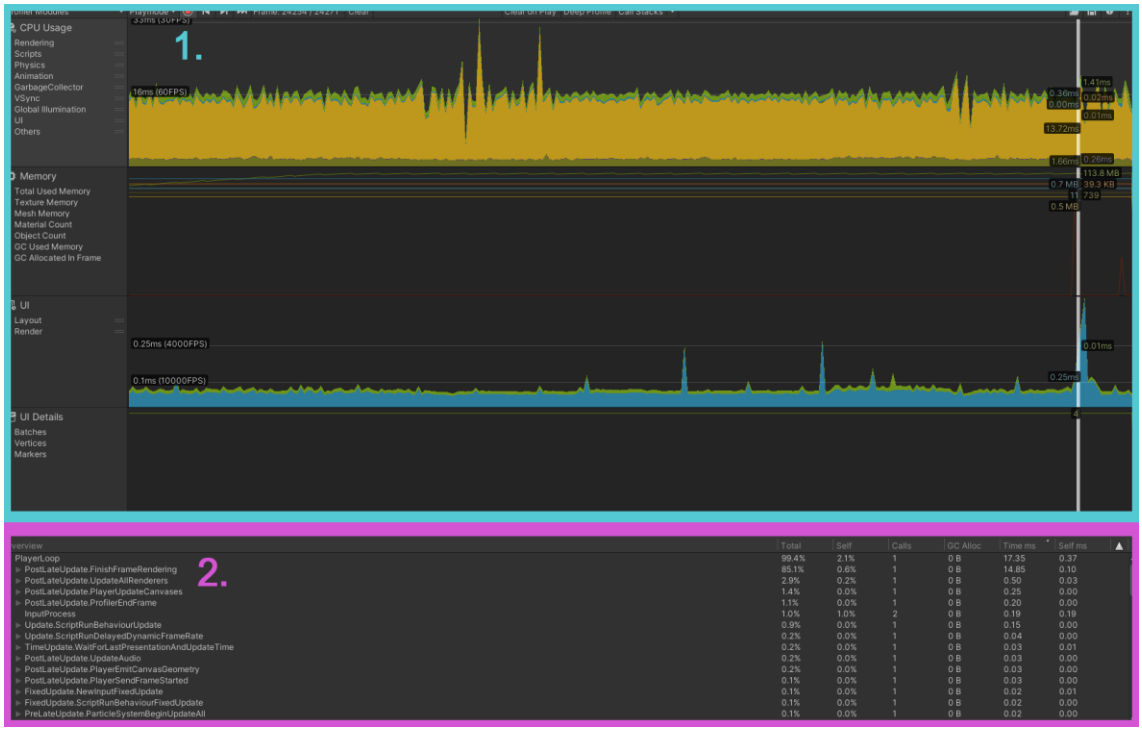


Kuva 10. Unity Profilerin avaaminen [28].

Ensimmäistä kertaa nähtynä Profilerin näkymät saattavat näyttää monimutkaisilta ja vaikealukuisilta, mutta niitä selvennetään seuraavaksi. Profiler tallentaa dataa editorissa pyörivän pelin suorituksesta joka ruudulta, kunhan Profilerin yläpalkin vasemmassa reunassa sijaitseva tallennuspainike on valittuna.

Oletuksena Profiler tallentaa muistiin 300 viimeisintä ruutua, mutta tätä asetusta pystyy muuttamaan.

Profilerin näkymä koostuu kahdesta päällekkäisestä osasta, jotka on merkitty kuvaan 11 numeroilla 1 ja 2.

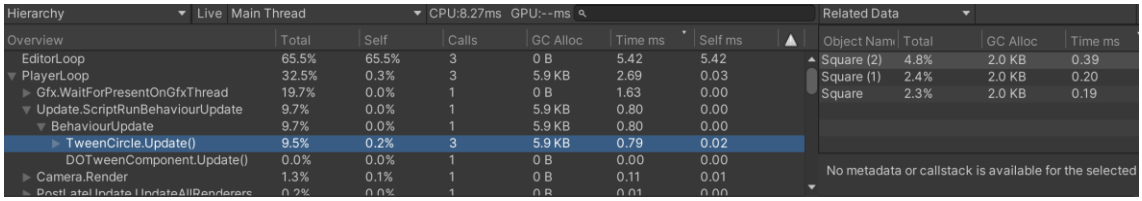


Kuva 11. Unity Profilerin näkymä [28].

Näkymä 1 sisältää vasemmalla puolella erilaisia profilointimoduuleita ja oikealla puolella aikajanan, joka esittää graafisesti informaatiota kyseisestä moduulista. Näkymä 2 eli alempi osa näyttää aina lisätietoa valitusta moduulista: esimerkiksi “CPU Usage” -moduulin ollessa valittuna Profilerin alempi näkymä näyttää kaikki kyseisen ruudun aikana kutsutut metodikutsut sekä niiden suoritukseen kuluneen ajan millisekunteina ja prosentteina ruudun kokonaiskestosta. Tätä näkymää voi katsella joko hierarkianäkymässä tai aikajananäkymässä, joka piirtää aikajanan yksittäisestä ruudusta, josta helposti voi havaita visuaalisesti, mikä osuus ruudun aikana tehtävästä prosessorin työstä kului mihinkin tehtävään.

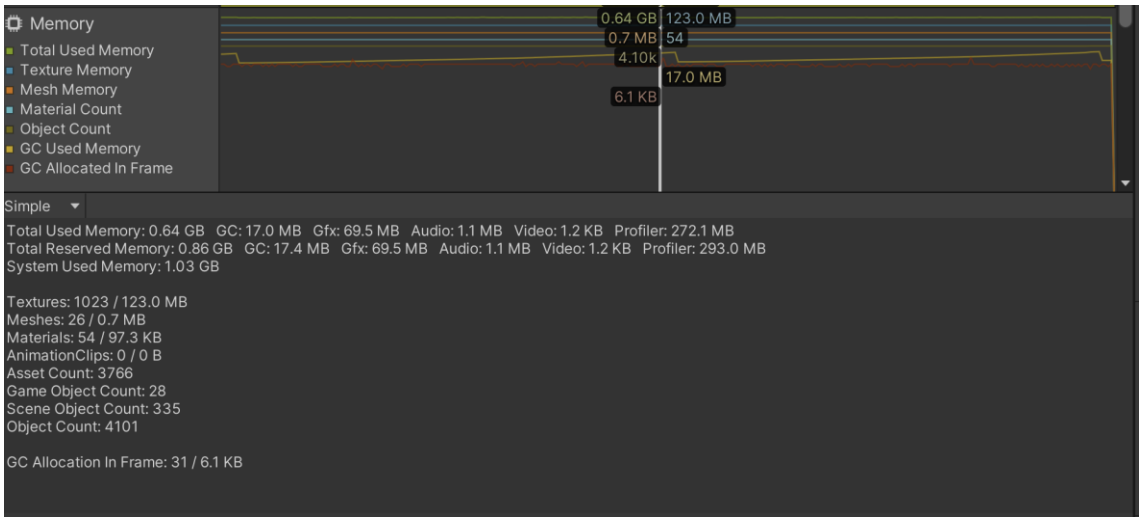
Profilointimoduuleita on Profilerissa reilun tusinan verran, niitä ovat muun muassa renderöinti, globaali valaisu, fysiikat ja äänet, mutta tässä insinöörityössä kiinnostavat erityisesti moduulit prosessorin käytölle (CPU Usage), muistinhallinnalle (Memory) ja Unity UI -komponenttien käytölle (UI).

Prosessorin käytön moduulista näkee yleiskuvaa prosessorin suoritusajoista sekä tarkempaa jaottelua, mitä tarkalleen ottaen prosessori on suorittanut ja kuinka kauan [31]. Tätä voi tarkastella ylänäkymän aikajanasta, johon eri osa-alueet, kuten skriptit, animaatiot ja käyttöliittymä, piirtävät omanväristä osuuttaan graafista tai sitten alanäkymän metodikutsuista suoraan. Näistä pystyy myös näkemään suoraan, mikä GameObject hierarkiasta kutsui kyseistä metodia, kun vaihtaa alanäkymän yläpalkin “No Details” -kohdan näyttämään “Related Data”, kuten kuvassa 12 on tehty.



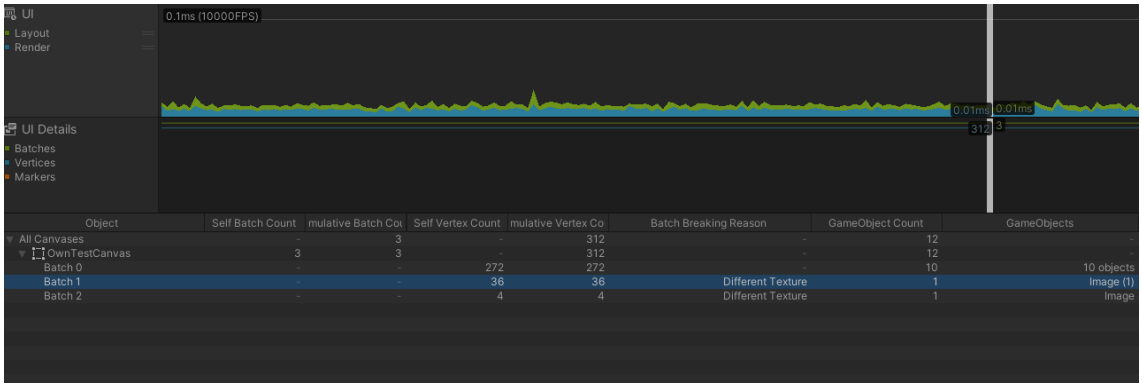
Kuva 12. Profilerin prosessorinhallintanäkymä [28].

Kuvassa 13 näkyvästä muistinhallinnan moduulista näkee hyödyllisiä tietoja siitä, miten peli käyttää ja varaa muistia peliä suorittavan laitteiston suoritusmuistista [31]. Graafista ja infonäkymästä näkee tarkempaa lisätietoa esimerkiksi materiaalien ja objektien määrästä, tekstuureihin menevästä muistista ja automaattisen roskienkeruun kuormasta.



Kuva 13. Profilerin muistinhallinnan moduuli [28].

Kuvassa 14 näkyvästä UI-moduulista näkee, kuinka paljon aikaa ja resursseja menee käyttöliittymien layoutien rakentamiseen ja renderöintiin [31]. Lisäksi voi tarkastella UI Details -moduulia, joka näyttää ruudun käyttöliittymän piirtämiseen tarvittavien piirtokutsujoukkioiden (engl. batches), verteksien ja merkintöjen (engl. markers) määrän. Merkinnät tarkoittavat Unityn tekemiä merkintöjä siitä, että käyttäjä on ollut interaktiossa käyttöliittymän kanssa eli esimerkiksi painanut painiketta tai liikuttanut liukusäädintä. UI- ja UI Details -moduulit jakavat yhteisen infonäkymän Profilerin alanäkymässä, ja näyttävät kaikki piirrettävät kanvaasit ja niiden piirtämiseen tarvittavat piirtokutsujoukkiot lisätietoineen, kuten objektit, joihin ne sisältyvät.



Kuva 14. Profilerin UI-näkymä [28].

Automaattisen suorituskykynäytteen nauhoituksen lisäksi Profileria voi kutsua suoraan koodissa nauhoittamaan tiettyjä koodilohkoja tai funktioita. Näihin toimintoihin pääsee C#-koodissa käsiksi tuomalla luokkaan UnityEngine.Profiling-nimiavaruuden ja kutsumalla Profiler-luokkaa [30, s. 36]. Tämä luokka tuo käytettäviksi metodit BeginSample ja EndSample. Oletuksena kaikki itse luodut metodit eivät näy Profilerin prosessorin käytön moduulin hierarkianäkymässä omina entiteetteinään, mutta kutsumalla metodia BeginSample näkyvät ennen EndSample-metodikutsua olevien rivien suorituskyvyn tilastot Profilerissa. BeginSample-metodille on olemassa metodin ylikuormitus, jolloin metodikutsun mukana voi lähettää argumenttina merkkijonona nimen omalle näytteelleen, kuten esimerkkikoodissa 1 näkyy.

```
public void Foo()
        {
            Profiler.BeginSample("Own_test_sample");
            for (int i = 0; i < 100; i++)
            {
                var x = 1 + 2;
            }
            Profiler.EndSample();
        }
```

Esimerkkikoodi 1. C#-skripti, joka tallentaa Profileriin testinäytteen.

Editorin pelaamistilassa (engl. play mode) käynnissä olevan pelin profiloinnin lisäksi Profilerin voi kytkeä käännettyyn koontiversioon (engl. build), jolloin saa vielä tarkempaa dataa suorituskyvystä, kun Unityn editorin prosessointi ei vaikuta pelitilan pyörittämiseen. Profileriin voi kytkeä koontiversion, joka pyörii tietokoneella natiivisti tai verkossa WebGL-instanssina. Mobiilipuolella Android-laitteella pyörivän koontiversion pystyy yhdistämään Unity Editorin joko langattoman verkon avulla tai käyttämällä Android ADB:tä ja iOS-laitteella yhdistämällä Profilerin langattoman verkon avulla. Koontiversiota luodessa on Unityn koontiasetuksista (engl. build settings) asetettava päälle Development Build ja Autoconnect Profiler. Tässä insinöörityössä käytetään Profileria Unity Editorin sisällä ja erillisen Android-koontiversion kanssa.

4 Suorituskyvyn optimointi Unityssa

Davide Aversan ja Chris Dickinsonin teoksessa Unity Game Optimization [30, s. 1] kerrotaan erinomaisesti, kuinka läheisesti suorituskyvyn optimointi kytkeytyy pelaajan kokemaan käyttökokemukseen. Huonosti optimoidut pelit voivat aiheuttaa pelaajalle turhautumista matalien ruudunpäivitysaikojen, pitkien latausaikojen, syöttöviiveen ja pelin kaatumisten vuoksi. Yksi suorituskyvyn optimoinnin tavoitteista on saada parhaalla mahdollisella tavalla hyöty tarjolla olevista resursseista ja laskentatehosta, jota pelilaitteen prosessori, suoritusmuisti ja grafiikkasuoritin tarjoaa. Tärkein tavoite on varmistaa, ettei yksikään näistä resursseista päädy niin sanotuksi pullonkaulaksi, mikä tarkoittaa sitä, että kyseisen laskentayksikön suoritusaika venyy niin pitkäksi, että muut laskentayksiköt joutuvat jäämään odottamaan tätä ja tämä odotus näkyy pelaajalle pelin tökkimisenä ja hitautena, joka voi ikävästi rikkoa pelaajan kokemaa immersiota.

Videopeliä pyörittävästä koodista löytyy lukuisia eri osa-alueita, joiden suorituskyvyssä voi olla optimoinnin varaa, mutta hyvin usein optimointi liittyy pohjimmiltaan näihin perusteisiin, eli että vältetään turhaa laskentatehon käyttämistä. Kompleksisten matemaattisten laskusuoritusten ajaminen on syytä tehdä niin harvoin kun mahdollista. Seuraavissa alaluvuissa käydään läpi erilaisia osa-alueita, joihin Unity-ohjelmoija usein törmää työssään niin käyttöliittymien, kuin muidenkin osa-alueiden toteutusten parissa. Lisäksi tehdään testivertailuja erilaisten toteutustapojen välillä, jotta pystytään analysoimaan optimaaliset tavat tarvittaviin tehtäviin. Testitapauksia varten kirjoitetut koodit ovat puhtaasti testejä varten kirjoitettua eivätkä välttämättä kuvasta puhtaasti todellisen pelin skenaarioita ja ovat iterointimäärissään liioiteltuja, jotta tuloksista saataisiin helposti tulkittavia johtopäätösten tekemistä varten. Testien ajamista varten luotiin Unitylla sovellus, jossa jokainen testi on oma Unity Scene (Unity-pelimoottorin tietorakenne yksittäisestä pelikentästä, -näkymästä tai -tilanteesta). Kuvassa 15 näkyy esimerkki testausta varten kehitetystä pelinäkymästä, joka koostuu painikkeista, joiden avulla pystyy valitsemaan, mitä esimerkkikoodia laite ajaa.

Kuva 15. Esimerkki testaamiseen käytetystä pelinäkymästä.

Sovellus käännettiin Androidille ja käännetyn version testit ajettiin Samsung S10+ -puhelimella. Lisäksi samat testit ajettiin Unityn editoritilassa vuoden 2021 Macbook Pro:lla, jossa on Apple M1 Pro -prosessori. Testitulokset koottiin kaikki Unity Profilerin avulla, niin että kirjattiin 5 tulosta, joista laskettiin keskiarvo, joka merkittiin skenaarion tulokseksi.

4.1 Referenssien hakeminen ja tallentaminen

Unityssa pelin toiminnallisuus ja erityisesti sen visuaalinen esitys tapahtuu pitkälti Unity-kirjaston MonoBehaviour-luokan instanssien avulla. MonoBehaviour-luokan perintä on vaatimuksena, että skriptin pystyy liittämään pelimoottorin hierarkiasta löytyvälle objektille komponentiksi. Kun skripti, eli tässä tapauksessa komponentti, lisätään objektiin, voi skriptissä kutsua GameObject-luokan

instanssia kirjoittamalla gameObject, ja se viittaa aina siihen objektiin, johon skripti on lisätty. GameObjecteille skriptejä kirjoittaessa ohjelmoijan tarvitsee usein tehdä referenssejä eli viittauksia toisiin GameObjecteihin. Tähän syynä voi olla esimerkiksi tarve laskea objektien välistä etäisyyttä tai välittää informaatiota niiden välillä. Näiden viittauksien tekeminen on yleistä myös käyttöliittymiä ohjelmoitaessa. Esimerkiksi kuvitteellisen pelin pistetilannetta kuvastavan tekstielementin on saatava itsensä ulkopuolelta tieto pelin pistetilanteesta, ja kun Unity-pelejä alkaa tehdä, tämä tieto usein tulee toisesta GameObjectista.

Unityn ohjelmointikirjasto tarjoaa GameObjectien löytämiselle omat vaihtoehtonsa, joita seuraavaksi käydään läpi ja pyritään löytämään optimoiduimmat toteutustavat. Ensimmäisessä testissä luokka sisältää kolme kokonaislukumuuttujaa, joiden arvot on haettava toisesta GameObjectista, joka sisältää MonoBehaviourin perivän luokan Test1ExampleData. Logiikka on asetettu silmukan sisään, jotta korkealla iterointimäärällä saadaan helposti tulkittavia tuloksia.

Esimerkkikoodi 2 sisältää ensimmäisen toteutustavan, joka hakee joka päivityskerralla FindObjectOfType-metodilla suoraan Test1ExampleData-luokan sisältävän objektin.

```
private void Update()
    {
        Profiler.BeginSample("test_1_case_A");
        for (int i = 0; i < iterations; i++)
        {
            var exampleData = FindObjectOfType<Test1ExampleData>();
            healthToDisplay = exampleData.Health;
            coinsToDisplay = exampleData.Coins;
            xpToDisplay = exampleData.XP;
        }

        Profiler.EndSample();
    }
```

Esimerkkikoodi 2. FindObjectOfType-metodin käyttäminen.

FindObjectOfType-metodi on hyvin raskas metodi kutsuttavaksi, minkä taulukossa 1 näkyvät testitulokset osoittavat.

Taulukko 1. FindObjectOfType-metodin käyttäminen.

500 iteraatiota tietokoneella	**10000 iteraatiota tietokoneella**	**500 iteraatiota puhelimella**	**10000 iteraatiota puhelimella**
1,60 ms	32,02 ms	7,64 ms	130,97 ms

Esimerkkikoodissa 3 kokeillaan korvata raskas FindObjectOfType-metodi toisella Unityn tarjoamalla hakumetodilla, joka on FindWithTag. Tätä varten haettavan GameObjectin tag-tunniste on oltava oikein asetettu. FindWithTag-metodi antaa referenssin GameObjectiin eikä Test1ExampleData-luokkaan, joten saadakseen tarvittavat tiedot exampleDatasta, täytyy vielä kutsua GetComponent-metodia.

```
private void Update()
    {
        Profiler.BeginSample("test_1_case_B");
        for (int i = 0; i < iterations; i++)
        {
            var exampleData = GameObject.FindWithTag("exampleData");
            healthToDisplay = exampleData
              .GetComponent<Test1ExampleData>().Health;
            coinsToDisplay = exampleData
              .GetComponent<Test1ExampleData>().Coins;
            xpToDisplay = exampleData
              .GetComponent<Test1ExampleData>().XP;
        }
        Profiler.EndSample();
    }
```

Esimerkkikoodi 3. FindWithTag.

Taulukosta 2 näkee, että prosessin suorituksen kesto laski merkittävästi, mutta tämä ei ole läheskään optimaalisin tapa suorittaa tehtävä.

Taulukko 2. Metodien FindWithTag ja GetComponent käyttäminen.

500 iteraatiota tietokoneella	**10000 iteraatiota tietokoneella**	**500 iteraatiota puhelimella**	**10000 iteraatiota puhelimella**
0,25 ms	4,62 ms	1,19 ms	14,92 ms

Yksi optimoinnin keskeisimpiä asioita on tehdä kukin operaatio vain niin usein, kuin on tarvetta, ja välttää yleistä virhettä, jossa viitteitä toisiin luokkiin haetaan

jatkuvasti [30, s. 51]. Esimerkkikoodeissa 2 ja 3 haettiin referenssin GameObjectiin joka iteraatiolla, vaikka se pysyi aina samana. Kun esimerkkikoodissa 4 siirretään GameObjectin hakeminen tapahtumaan vain kerran Start-metodissa ja tallennetaan se luokkamuuttujaan, nähdään taulukon 3 testituloksista, että saadaan yhä lisää millisekunteja pois suorituksen kestosta.

```
private GameObject exampleData;

    private void Start()
    {
        exampleData = GameObject.FindWithTag("exampleData");
    }

    private void Update()
    {
        Profiler.BeginSample("test_1_case_C");
        for (int i = 0; i < iterations; i++)
        {
            healthToDisplay = exampleData
              .GetComponent<Test1ExampleData>().Health;
            coinsToDisplay = exampleData
              .GetComponent<Test1ExampleData>().Coins;
            xpToDisplay = exampleData
              .GetComponent<Test1ExampleData>().XP;
        }

        Profiler.EndSample();
    }
```

Esimerkkikoodi 4. GameObjectin tallentaminen välimuistiin.

Taulukko 3. GameObjectin tallentaminen välimuistiin.

500 iteraatiota tietokoneella	**10000 iteraatiota tietokoneella**	**500 iteraatiota puhelimella**	**10000 iteraatiota puhelimella**
0,18 ms	3,40 ms	0,98 ms	11,57 ms

Kun esimerkkikoodissa 5 tehdään sama vielä GetComponent-kutsuille, saavutetaan huomattavasti optimoidumpi tapa hakea GameObject-referenssi luokalle.

```
private GameObject exampleData;
private Test1ExampleData;

    private void Start()
    {
        exampleData = GameObject.FindWithTag("exampleData");
        test1ExampleData = exampleData
          .GetComponent<Test1ExampleData>();
    }

    private void Update()
    {
        Profiler.BeginSample("test_1_case_D");
        for (int i = 0; i < iterations; i++)
        {
            healthToDisplay = test1ExampleData.Health;
            coinsToDisplay = test1ExampleData.Coins;
            xpToDisplay = test1ExampleData.XP;
        }

        Profiler.EndSample();
    }
```

Esimerkkikoodi 5. Kaikkien referenssien tallentaminen välimuistiin.

Taulukosta 4 havaitaan, että nyt tietokoneella suorituksen kesto 10 000 iteroinninkin määrällä on vain 0,03 ms, eli yli 100 kertaa vähemmän kuin variantissa C ja yli 1000 kertaa vähemmän kuin ensimmäisessä variantissa. FindObjectByTag- ja GetComponent-metodikutsujen kuorma tapahtuu vain kerran GameObjectin elinaikana Start-metodissa.

Taulukko 4. Kaikkien muuttujien tallentaminen välimuistiin.

500 iteraatiota tietokoneella	**10000 iteraatiota tietokoneella**	**500 iteraatiota puhelimella**	**10000 iteraatiota puhelimella**
0,00 ms	0,03 ms	0,00 ms	0,05 ms

Jos luokka tarvitsee suuren määrän muiden luokkien referenssikseen, niin Start-metodissa tapahtuvien kutsujen kuorma kasvaa ja siihenkin on muutamia apukeinoja. Yksi on määrittää luokkamuuttujista julkisia, jolloin Unity näyttää muuttujat editorin Inspector-näkymässä ja pelikehittäjä voi raahata oikean GameObjectin muuttujan arvoksi. Muuttujan voi myös pitää yksityisenä, mutta lisätä eteen attribuutin [SerializedField], jolloin muuttuja näkyy Inspectorissa, mutta pysyy koodissa yksityisenä muille luokille. Riippuvuuksien injektointi

(engl. Dependency Injection) on myös yksi kehittyneempi vaihtoehto, mutta siihen ei tässä insinöörityössä perehdytä.

4.2 Update-silmukan karsiminen

MonoBehaviour-luokan instansseilla on Unityn määrittämä elinkaari, johon kuuluu esimerkiksi metodit Start, Update ja OnDestroy [31]. Nämä metodit ovat hyvin käytännöllisiä siitä syystä, että pelimoottori kutsuu niitä tiettyinä hetkinä jokaiselta instanssilta, jolta ne löytyvät luokan sisältä. Update-metodi on hyvin keskeinen tekijä Unity-ohjelmoinnissa, sillä sitä kutsutaan joka kerta, kun uusi ruutu piirretään. Tämä tarjoaa kätevän tavan toteuttaa pelilogiikan osasia, jotka tapahtuvat jatkuvasti pelin pyöriessä, kuten esimerkiksi pelin hahmon liikkuminen, tekoälyyn liittyvät toiminnot tai käyttäjän syötteen lukeminen. Suorituskyvyn kannalta on kuitenkin erittäin tärkeää, että tässä silmukassa ei lasketa mitään turhaan, varsinkaan mitään raskasta operaatiota. Kuten jo edellisessä testissä todettiin, erittäin hyvä ja helppo optimointikeino on varmistaa, että jos skriptin vaatimia viittauksia muihin luokkiin ei tarvitse ratkaista dynaamisesti, niitä ei haeta toistuvasti. Myöskään uusia apumuuttujia ei kannata alustaa liikaa silmukoissa, vaan pyrkiä alustamaan ne jo aiemmin luokkamuuttujiksi.

Muistinhallinta on toinen osa-alue, jota on hyvä tarkastella optimointia tehdessä suoritusajan lisäksi. Automaattinen roskienkeruu (engl. Garbage Collector) on C#-ohjelmointikielen ominaisuus, joka huolehtii muistialueiden vapautuksesta, kun muistialueita varaaviin olioihin ei enää löydy viitteitä [32]. Profilerin avulla näkee, kuinka paljon esimerkiksi näytteenä toimiva koodilohko varaa muistia, jonka roskienkerääjä joutuu vapauttamaan. Jos määrät kasvavat liian suuriksi, voi niiden vapauttaminen näkyä pelaajalle pelin tökkimisenä. Muistia varataan aina, kun luodaan uusi muuttuja, on se sitten luokkamuuttuja tai metodin sisällä luotava dynaaminen muuttuja.

Työssä tehtiin vielä toinen testi, joka havainnollistaa, kuinka merkittävä ero on sen välillä, että luokkaviittaukset ja apumuuttujat ovat jo tallennettuna välimuistiin, kuin että niitä ratkaistaisiin ja luotaisiin silmukassa. Esimerkkikoodissa 6 on

tämän testin ensimmäinen skenaario, jonka tarkoitus on esittää kehnosti optimoitua tapaa hakea päivitettävä tekstielementti, hakea viittaus toiseen GameObjectiin ja sen Transform-komponenttiin, laskea objektien välinen etäisyys ja päivittää se tekstielementin arvoksi. Iteraatioiden määrä tässä testissä on 10.

```
private void Update()
    {
        Profiler.BeginSample("test_2_case_A");
        for (int i = 0; i < iterations; i++)
        {
            TMP_Text textToUpdate = FindObjectOfType<UIManager>()
              .distanceText;
            GameObject playerTwoGameObject = GameObject
              .FindGameObjectWithTag("PlayerTwo");
            Transform playerTwoTransform = playerTwoGameObject
              .transform;
            float distance = Vector3.Distance(
              playerTwoTransform.position, transform.position);
            textToUpdate.SetText(distance.ToString());
        }
        Profiler.EndSample();
    }
```

Esimerkkikoodi 6. Referenssien dynaaminen hakeminen silmukassa.

Silmukassa tehdään turhaan hakuja UIManager-luokkaan ja toiseen GameObjectiin ja luodaan joka ajokerralla dynaamisesti uusia muuttujia neljä kappaletta. Taulukosta 5 nähdään, kuinka hidasta tämän logiikan suorittaminen on ja kuinka monta tavua (B, engl. byte) operaatio allokoi muistia.

Taulukko 5. Luokkaviitteiden dynaaminen ratkaiseminen silmukan sisällä.

Suoritusaika tietokoneella	**Muistiallokaatio tietokoneella**	**Suoritusaika puhelimella**	**Muistiallokaatio puhelimella**
0,16 ms	800 B	0,80 ms	480 B

Esimerkkikoodissa 7 on optimaalisempi tapa toteuttaa testin vaatima logiikka.

```
[SerializeField] private UIManager uiManager;
[SerializeField] private Transform playerTwoTransform;
private TMP_Text textToUpdate;
private float distance;

private void Start()
{
    textToUpdate = uiManager.distanceText;
}

private void Update()
{
    for (int i = 0; i < iterations; i++)
    {
        Profiler.BeginSample("test_2_case_B");
        distance = Vector3.Distance(
          playerTwoTransform.position, transform.position);
        textToUpdate.SetText(distance.ToString());
        Profiler.EndSample();
    }
}
```

Esimerkkikoodi 7. Referenssien tallentaminen välimuistiin.

Kun siirretään tarvittavat referenssit luokkamuuttujiksi, asetetaan ne Inspector-näkymässä raahaamalla oikeat objektit paikoilleen Inspector-näkymässä, haetaan päivitettävä tekstiobjekti UIManager-luokan instanssilta Start-metodissa ja myös luodaan luokkamuuttuja etäisyydelle, saadaan taulukossa 6 nähtävien tulosten perusteella Update-metodi toimimaan noin puolet nopeammin ja allokoimaan noin puolet vähemmän roskaa muistinhallintaan. Muistiallokaatiota tulee liukuluvun merkkijonoksi muuttamiseksi. Koodista tulee lisäksi helppolukuisempaa, sillä nyt Update-silmukan sisällä tehdään vain etäisyyden laskemisen logiikka ja ajankohtaisen arvon asettaminen käyttöliittymään.

Taulukko 6. Luokkaviitteiden tallentaminen välimuistiin.

Suoritusaika tietokoneella	**Muistiallokaatio tietokoneella**	**Suoritusaika puhelimella**	**Muistiallokaatio puhelimella**
0,08 ms	420 B	0,45 ms	280 B

Pelikoodia kirjottaessa on hyvä usein pysähtyä miettimään, voisiko työstettävän logiikan ajaa metodikutsun avulla tai reaktiivisen tapahtuman perusteella. Jos voi, niin hyvin todennäköisesti se on paras vaihtoehto, ainakin parempi kun ajaa logiikkaa joka ruudulla ajettavassa silmukassa. Helppo esimerkki on vaikka

käyttöliittymäelementti, joka näyttää pelaajalle pelihahmon jäljellä olevat elämäpisteet. Sen sijaan että käyttöliittymän tekstielementtiä päivitettäisiin joka ruudulla vastaamaan tietorakenteesta löytyvää elämäpistemäärää, se kannattaa päivittää vain silloin, kun tiedetään, että kyseinen arvo on muuttunut. Esimerkkikoodissa 8 on epäoptimoitu tapa, jossa tekstinpäivitys suoritetaan Update-silmukassa.

```
public class Example : MonoBehaviour
    {
        [SerializeField] private Text HealthText;
        [SerializeField] private PlayerManager;

        private void Update()
        {
            HealthText.text = PlayerManager.Health.ToString();
        }
    }

    public class PlayerManager : MonoBehaviour
    {
        public int Health = 100;

        public void ChangeHealth(int delta)
        {
            Health += delta;
        }
    }
```

Esimerkkikoodi 8. Update-silmukassa päivittäminen.

Esimerkkikoodissa 9 päivityslogiikka on siirretty kutsuttavaksi metodiksi, jota kutsutaan PlayerManager-luokan puolelta, kun arvo muuttuu eli julkista metodia ChangeHealth kutsutaan jostain.

```
public class Example : MonoBehaviour
    {
        [SerializeField] private Text HealthText;

        public void UpdateText(string newHealth)
        {
            HealthText.text = newHealth;
        }
    }

    public class PlayerManager : MonoBehaviour
    {
        private int health = 100;
        [SerializeField] private Example example;

        public void ChangeHealth(int delta)
        {
            health += delta;
            example.UpdateText(health.ToString());
        }
    }
```

Esimerkkikoodi 9. Metodikutsu.

Jos taas koodilohkolle ei keksi loogista metodikutsua tai mitään tapahtumaa, johon kiinnittää sitä, voi miettiä, onko kyseistä logiikkaa pakko suorittaa joka ruutu. Jos tämä ei ole välttämätöntä, on helppoa ohjelmoida se laskettavaksi vain tietyn frekvenssin mukaan.

Esimerkkikoodissa 10 säätämällä muuttujaa Frequency pystyy määrittämään, kuinka usein raskas laskutoimitus ajetaan.

```
private int frames = 0;
    [SerializeField] private int Frequency = 10;

    private void Update()
    {
        frames++;
        if (frames % Frequency == 0)
        {
            HeavyCalculation();
        }
    }

    private void HeavyCalculation()
    {
        Debug.Log("This got called");
    }
```

Esimerkkikoodi 10. Joka n:s ruutu toteuttava metodikutsu.

Unity kutsuu automaattisesti kaikkien GameObjectien Update-metodeja, kuten kaikkia muitakin MonoBehaviourin elinkaareen kutsuvia metodeja, ja niitä kutsutaan vaikka niiden sisältö olisi tyhjä [30, s. 47]. Oletuksena Unity luo tyhjät Start- ja Update-metodit jokaisen uuden Monobehaviour-luokan kanssa. Jos näitä metodeja ei käytä, ne on syytä poistaa, ettei kääntäjä käy turhaan kutsumassa niitä ja aiheuta turhaa kuormitusta prosessorille.

Seuraavassa testissä tutkitaan, mitä tapahtuu, kun hierarkiassa on suuri määrä objekteja, joilta löytyy tyhjä Update-metodi. Taulukossa 7 näkyvät tulokset skenaariosta, jossa luodaan 5000 GameObjectia, joilta löytyy tyhjä Update-metodi ja 10 GameObjectia, joilta löytyy kohtalaisen monimutkainen laskutoimitus Update-metodista.

Taulukko 7. Tyhjien Update-metodien tuoma lisärasite.

Update-kutsujen lkm. tietokoneella	**Update-metodien suoritusaika tietokoneella**	**Update-kutsujen lkm. puhelimella**	**Update-metodien suoritusaika puhelimella**
5010	0,45 ms	5010	6,31 ms

Testitulokset näyttävät, että Update-kutsuja tehtiin ruudun aikana 5010 ja aikaa kului yhteensä 0,45 ms. Taulukossa 8 näkyvät tulokset toisesta skenaariosta, jossa 5000 luodulta objektilta ei löydy enää Update-metodia ollenkaan. Ne voisivat olla esimerkiksi objekteja, jotka reagoivat vain, jos niihin osuu. Update-metodit löytyvät vain 10 objektilta, joiden metodi-implementaatiossa on sisältöä.

Taulukko 8. Vain logiikkaa sisältävien Update-metodien läpikäynti.

Update-kutsujen lkm. tietokoneella	**Update-metodien suoritusaika tietokoneella**	**Update-kutsujen lkm. puhelimella**	**Update-metodien suoritusaika puhelimella**
10	0,03 ms	10	0,06 ms

Nyt kutsuja tehdään oletettu 10 kappaletta ja suoritusaika tietokoneella on vain 0,03 sekuntia. Ensimmäisen skenaarion suoritusajasta siis 93,75 % meni tyhjien metodien läpikäymiseen.

4.3 Monialkioiset tietorakenteet

Käyttöliittymiin liittyy hyvin usein tietorakenteena erilaiset listat ja taulukot, sillä usein pelaajalle esiteltävä data on haettavissa juuri näistä tietorakenteista, erityisesti laajemmissa strategia- ja manageerointipeleissä. Monialkioisten tietorakenteiden luomiseen ja läpikäymiseen on useita tapoja, joita seuraavassa testissä käydään läpi.

Testissä generoidaan monialkioiseen tietorakenteeseen 100 000 satunnaislukua väliltä 0–1000 ja eri tavoin haetaan tietorakenteesta lukumäärä luvuista, joiden arvo on yli 200.

Testin kolme erilaista tietorakennetta ovat lista, staattinen taulukko ja IEnumerable, joka on rajapinta, jonka listan ja taulukon luokat perivät. Testissä käytetään kolmea erilaista tapaa käsitellä tietorakenteita: ensimmäinen on tavallinen for-silmukka, jota iteroidaan tietorakenteen alkioiden lukumäärän verran, toinen on foreach-silmukka ja kolmantena käytetään LINQ-kirjastoa. LINQ, eli nimeltään Language Integrated Query, on C#-kielen sisään rakennettu kyselytyökalu, jolla pystyy kätevästi suodattamaan, järjestämään ja ryhmittelemään tietorakenteita vähäisellä koodimäärällä [33].

Yhteensä nämä tietorakenteet ja operaatiot tarjoavat yhdeksän skenaariota testattavaksi.

Koodiesimerkissä 11 on ensimmäisen variantin koodi kokonaisuudessaan ja taulukossa 9 kyseisen skenaarion testitulos.

```
public class Test4CaseA : MonoBehaviour
{
    private int[] arrayOfNumbers = new int[100000];
    private int frames = 0;
    private const int Frequency = 10;

    private void Start()
    {
        var random = new Random();
        for (int i = 0; i < arrayOfNumbers.Length; i++)
        {
            arrayOfNumbers[i] = random.Next(1000);
        }
    }

    private void Update()
    {
        frames++;
        if (frames % Frequency == 0)
        {
            int amountOfNumbersOver200 = GetAmountOfNumbersOver200(
              arrayOfNumbers);
        }
    }

    private int GetAmountOfNumbersOver200(int[] arrayOfNumbers)
    {
        int returnValue = 0;
        for (int i = 0; i < arrayOfNumbers.Length; i++)
        {
            if (arrayOfNumbers[i] > 200)
            {
                returnValue++;
            }
        }

        return returnValue;
    }
}
```

Esimerkkikoodi 11. For-silmukan käyttäminen taulukon läpikäyntiin.

Taulukko 9. Taulukon läpikäynti for-silmukalla.

Suoritusaika tietokoneella	**Muistiallokaatio tietokoneella**	**Suoritusaika puhelimella**	**Muistiallokaatio puhelimella**
0,67 ms	0 B	1,36 ms	0 B

Seuraavista koodiesimerkeistä näytetään vain GetSumOfNumbersOver200-metodin toteutus. Kyseistä metodia kutsutaan tässä testissä aina esimerkkikoodin

11 tapaan. Esimerkkikoodissa 12 esitetään foreach-silmukan käyttäminen taulukkorakenteen läpikäyntiin.

```
private int GetAmountOfNumbersOver200(int[] arrayOfNumbers)
    {
        int returnValue = 0;
        foreach (var number in arrayOfNumbers)
        {
            if (number > 200)
            {
                returnValue++;
            }
        }

        return returnValue;
    }
```

Esimerkkikoodi 12. Foreach-silmukan käyttäminen taulukon läpikäyntiin.

Taulukosta 10 voidaan todeta, että foreach-silmukka oli hieman nopeampi taulukon läpikäynnissä. Helppolukuinen foreach-silmukka on myös nopeampi kirjoittaa kuin for-silmukka, joten sitä voi hyvin käyttää taulukon läpikäyntiin.

Taulukko 10. Taulukon läpikäynti foreach-silmukalla.

Suoritusaika tietokoneella	**Muistiallokaatio tietokoneella**	**Suoritusaika puhelimella**	**Muistiallokaatio puhelimella**
0,60 ms	0 B	1,17 ms	0 B

Esimerkkikoodissa 13 näkyy LINQ-kyselyn käyttäminen taulukkorakenteen läpikäymiseen.

```
private int GetAmountOfNumbersOver200(int[] arrayOfNumbers)
    {
        return arrayOfNumbers.Count(number => number > 200);
    }
```

Esimerkkikoodi 13. LINQ-kirjaston käyttäminen taulukkorakenteen läpikäyntiin.

Taulukosta 11 nähdään, että LINQ-kysely oli selkeästi hitaampi kuin silmukoiden käyttäminen taulukkotietorakennetta käyttäessä. LINQ-kysely allokoi aina myös hieman muistia. Ohjelmoijan on kuitenkin arvioitava suoritettavan logiikan

laajuus projektin kontekstissa ja tehtävä sen perusteella päätös, onko syytä optimoinnin takia käyttää silmukkaa vai onko ruudun aikabudjetissa varaa käyttää LINQ-kirjastoa sen helppouden ja luettavuuden takia.

Taulukko 11. Taulukon läpikäynti LINQ-kyselyllä.

Suoritusaika tietokoneella	**Muistiallokaatio tietokoneella**	**Suoritusaika puhelimella**	**Muistiallokaatio puhelimella**
3,85 ms	32 B	5,42 ms	16 B

Esimerkkikoodissa 14 tietorakenteeksi on vaihtunut lista (List<int>) ja se käydään läpi for-silmukalla.

```
private int GetSumOfNumbersOver200(List<int> listOfNumbers)
    {
        int returnValue = 0;
        for (int i = 0; i < listOfNumbers.Count; i++)
        {
            if (listOfNumbers[i] > 200)
            {
                returnValue++;
            }
        }

        return returnValue;
    }
```

Esimerkkikoodi 14. For-silmukan käyttäminen listan läpikäyntiin.

Taulukosta 12 nähdään esimerkkikoodin 14 testitulokset.

Taulukko 12. Listan läpikäynti for-silmukalla.

Suoritusaika tietokoneella	**Muistiallokaatio tietokoneella**	**Suoritusaika puhelimella**	**Muistiallokaatio puhelimella**
1,60 ms	0 B	1,73 ms	0 B

Esimerkkikoodissa sama listarakenne käydään läpi foreach-silmukan avulla.

```
private int GetAmountOfNumbersOver200(List<int> listOfNumbers)
    {
        int returnValue = 0;
        foreach (var number in listOfNumbers)
        {
            if (number > 200)
            {
                returnValue++;
            }
        }

        return returnValue;
    }
```

Esimerkkikoodi 15. Foreach-silmukan käyttäminen listan läpikäyntiin.

Taulukossa 13 nähdään esimerkkikoodin 15 testitulokset.

Taulukko 13. Listan läpikäynti foreach-silmukalla.

Suoritusaika tietokoneella	**Muistiallokaatio tietokoneella**	**Suoritusaika puhelimella**	**Muistiallokaatio puhelimella**
1,71 ms	0 B	1,96 ms	0 B

Erot eivät ole suuria, mutta toisin kun taulukossa, listaa käyttäessä tavallinen for-silmukka oli tehokkaampi kuin foreach-silmukka. Tämä johtuu siitä, että foreach-silmukkaa käyttäessä staattisen taulukon kohdalla kääntäjällä on jo tiedossa, minkä kokoinen taulukko on kyseessä, kun taas dynaamisen listan koko ei ole kääntäjällä tiedossa. Listan läpikäynti on molemmilla silmukoilla hitaampaa kuin staattisen taulukon, mutta ohjelmoijan etuna on se, että alkioiden määrää ei tarvitse tietää tietorakennetta alustaessa.

Koodiesimerkissä 16 käytetään LINQ-kyselyä listan läpikäyntiin.

```
private int GetAmountOfNumbersOver200(List<int> listOfNumbers)
    {
        return listOfNumbers.Count(number => number > 200);
    }
```

Esimerkkikoodi 16. LINQ-kyselyn käyttäminen listan läpikäyntiin.

Taulukosta 14 nähdään, että LINQ-kysely on myös listaa käyttäessä hitain, mutta ero nopeimman silmukan ja LINQ:n välillä on prosentuaalisesti huomattavasti pienempi listassa kuin taulukkoa käyttäessä. Tietokoneella ajetuissa testeissä taulukkoa käyttäessä foreach-silmukan suoritusaika on 542 % nopeampi kuin LINQ-kyselyn suoritusaika ja listassa for-silmukan suoritusaika on vain 70 % nopeampi kuin LINQ-kyselyn suoritusaika. Jos optimoinnin tarve ei siis ole kriittisen suuri ja iteraatioiden tai kutsujen määrä mittava, LINQ-kyselyä voi hyvin käyttää listojen läpikäyntiin ja niistä tiettyjen tietojen noutamiseen ja järjestelyyn.

Taulukko 14. Listan läpikäynti LINQ-kyselyllä.

Suoritusaika tietokoneella	**Muistiallokaatio tietokoneella**	**Suoritusaika puhelimella**	**Muistiallokaatio puhelimella**
2,72 ms	40 B	4,45 ms	24 B

IEnumerable on rajapinta, jonka List- ja Array-luokat perivät. Sitä voi käyttää myös omana monialkioisena tietorakenteenaan.

IEnumerable ei tue suoraa indeksointia, vaan tietyn indeksin alkio on haettava ElementAt-metodikutsulla. Kun yritettiin hakea 100 000 alkion IEnumerable-tietorakenteesta indeksin mukaista alkiota ElementAt-metodikutsulla, operaatio sai ohjelman kaatumaan. Esimerkkikoodiin 17 vähennettiin alkioiden määrä tuhanteen, jotta testin pystyi ajamaan.

```
private int GetAmountOfNumbersOver200(IEnumerable<int> numbers)
    {
        int returnValue = 0;
        for (int i = 0; i < numbers.Count(); i++)
        {
            if (numbers.ElementAt(i) > 200)
            {
                returnValue++;
            }
        }

        return returnValue;
    }
```

Esimerkkikoodi 17. For-silmukan ja ElementAt-metodin käyttäminen IEnumerable-tietorakenteen läpikäyntiin.

Sata kertaa pienemmälläkin iteraatiomäärällä taulukon 15 tulosten mukaan operaatio allokoi muistia 500 kB roskienkeruulle kerättäväksi ja suoritusaika on yli 50 ms tietokoneella ja yli 120 ms puhelimella. Perinteistä for-silmukkaa ei selvästikään kannata käyttää IEnumerable-tietorakenteen läpikäymiseen.

Taulukko 15. IEnumerable-tietorakenteen läpikäynti for-silmukalla.

Suoritusaika tietokoneella	**Muistiallokaatio tietokoneella**	**Suoritusaika puhelimella**	**Muistiallokaatio puhelimella**
51,12 ms	600 000 B	128,71	600 000 B

Esimerkkikoodissa 18 käytetään foreach-silmukkaa IEnumerable-tietorakenteen läpikäymiseen, tällä kertaa alkioiden määrä on nostettu takaisin 100 000:een.

```
private int GetSumOfNumbersOver200(IEnumerable<int> numbers)
    {
        int returnValue = 0;
        foreach (var number in numbers)
        {
            if (number > 200)
            {
                returnValue++;
            }
        }


        return returnValue;
    }
```

Esimerkkikoodi 18. Foreach-silmukan käyttäminen IEnumerable-tietorakenteen läpikäyntiin.

Taulukosta 16 nähdään tulokset, jotka kertovat, että Foreach-silmukalla IEnumerable läpikäynti on jo huomattavasti tehokkaampaa kuin ElementAt-metodin käyttäminen, mutta kuitenkin keskiarvoisesti yli neljä kertaa hitaampaa kuin listan tai taulukon läpikäynti Foreach-silmukassa. Muistia allokoidaan tässä skenaariossa noin 300 tavua.

Taulukko 16. IEnumerable-tietorakenteen läpikäynti foreach-silmukalla.

Suoritusaika tietokoneella	**Muistiallokaatio tietokoneella**	**Suoritusaika puhelimella**	**Muistiallokaatio puhelimella**
4,16 ms	336 B	7,70 ms	292 B

Tämän testin viimeisessä esimerkkikoodissa 19 käytetään LINQ-kyselyä IEnumerable-tietorakenteen läpikäyntiin.

```
private int GetSumOfNumbersOver200(IEnumerable<int> numbers)
    {
        return numbers.Count(number => number > 200);
    }
```

Esimerkkikoodi 19. LINQ-kyselyn käyttäminen IEnumerable-tietorakenteen läpikäyntiin.

Taulukon 17 testituloksista nähdään, että LINQ:n avulla IEnumerablen läpikäynnissä saadaan suunnilleen vastaavia tuloksia kuin foreach-silmukkaa käyttäessä.

Taulukko 17. IEnumerable-tietorakenteen läpikäynti LINQ-kyselyllä.

Suoritusaika tietokoneella	**Muistiallokaatio tietokoneella**	**Suoritusaika puhelimella**	**Muistiallokaatio puhelimella**
4,84 ms	336 B	8,64 ms	292 B

Tutkimustulosten perusteella IEnumerable ei ole optimaalisin vaihtoehto, jos ohjelmoija luo tietorakennetta käyttöliittymässä esiteltävää dataa varten, mutta tilanteessa, jossa käyttöliittymäohjelmoijalla tulee vastaan tehtäväksi käsitellä jo luotua IEnumerable-tietorakennetta, on hyvä tietää, että ElementAt-metodia on vältettävä silmukassa ja suosittava foreach-silmukkaa tai esimerkkikoodin 19 mukaista LINQ-kyselyn käyttöä. Taulukossa 18 esitetään vielä kaikki tämän testin tulokset yhdessä.

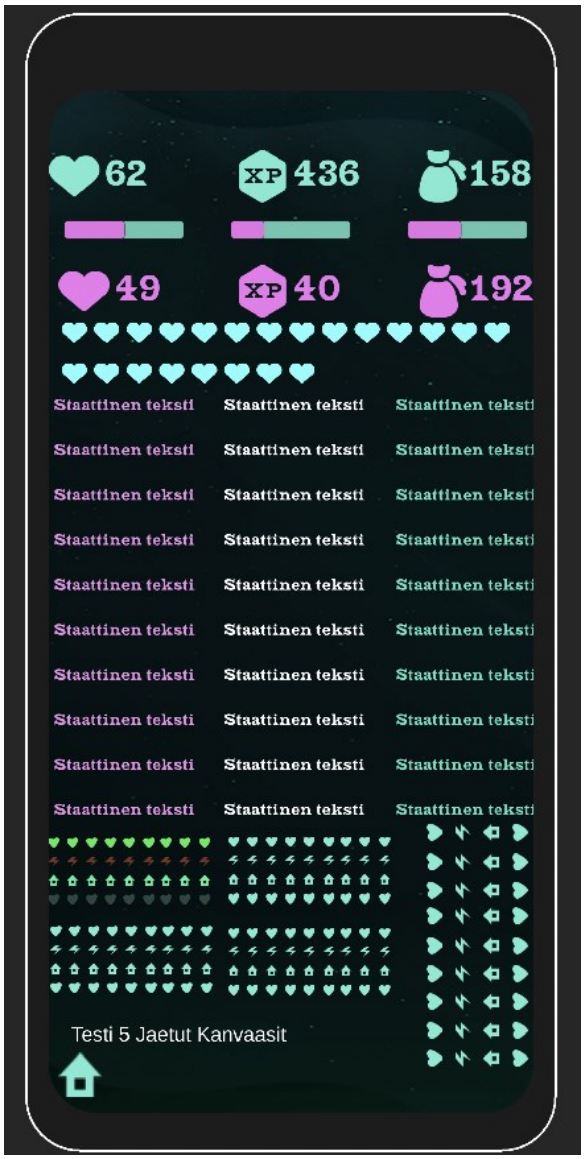
Taulukko 18. Testin 4 kootut testitulokset.

Tietorakenne	**For-silmukka, tietokone**	**For-silmukka, puhelin**	**Foreach-silmukka, tietokone**	**Foreach-silmukka, puhelin**	**LINQ-kysely, tietokone**	**LINQ-kysely, puhelin**
Taulukko	0,67 ms (0 B)	1,36 ms (0 B)	0,60 ms (0 B)	1,17 ms (0 B)	3,85 ms (32 B)	5,42 ms (16 B)
Lista	1,60 ms (0 B)	1,73 ms (0 B)	1,71 ms (0 B)	1,96 ms (0 B)	2,72 ms (40 B)	4,45 ms (24 B)
IEnumerable	51,12 ms (600 000 B)	128, 71 ms (600 000 B)	4,16 ms (336 B)	7,70 ms (292 B)	4,84 ms (336 B)	8,64 ms (292 B)

4.4 Unity UI -kanvaasien optimointi

Luvussa 3.2 todettiin, että kaikki käyttöliittymäelementit kuuluvat johonkin kanvaasiin. Kanvaasi piirretään näytölle joka kerta, kun ruutu päivitetään, ja jos yksikin kanvaasin alainen elementti on muuttunut edellisen ruudunpiirron jäljiltä, koko kanvaasi lasketaan aina uusiksi ja kaikki piirrettävä grafiikka generoidaan uusiksi ennen piirtokutsua [34]. Monimutkaisissa hierarkiarakenteissa, joissa kanvaasiobjektin alla on paljon objekteja monessa tasossa, tämä voi aiheuttaa tökkimistä ja hitautta. Tähän toimiva optimointiratkaisu on välttää liian suurta yksittäistä kanvaasia ja jakaa elementtejä useampaan kanvaasiin, mieluiten niin, että jakaa erikseen staattiset ja dynaamiset käyttöliittymäelementit omiin kanvaaseihin. Silloin staattisia elementtejä, kuten taustakuvia tai kuvakkeita, ei tarvitse kokonaan päivittää joka kerta, kun joku dynaaminen elementti, kuten laskuri tai elämäpistepalkki, päivittyy. Dynaamisetkin elementit voi vielä erotella omiin kanvaaseihin sen mukaan, päivittyvätkö ne tiheästi vai harvemmin. Esimerkiksi interaktiivisuutta vaativat elementit kuten painikkeet saattavat päivittyä huomattavasti harvemmin kuin jossakin silmukassa päivittyvät tekstielementit. Liian montaa kanvaasia ei kannata kuitenkaan luoda yksittäiseen Unity Sceneen, vaan sopivan tasapainon voi hakea jokaiseen projektiin Profilerin avulla.

Testissä 5 on luotu kohtuullisen raskas käyttöliittymä, joka koostuu staattisista elementeistä, kuten kuvakkeista ja teksteistä, sekä dynaamisista elementeistä, kuten päivittyvistä merkkijonoista ja liukusäätimistä sekä dynaamisesti elementtien määrää vaihtavasta horisontaalisesta layout-ryhmästä. Myös staattisissa elementeissä on käytetty automaattisia layout-ryhmiä. Kuvasta 16 näkyy, miltä testiä varten toteutettu käyttöliittymä näyttää.



Kuva 16. Testin 5 näkymä.

Testin ensimmäisessä skenaariossa ja sitä varten luodussa pelitilanteessa kaikki käyttöliittymäelementit ovat saman Canvas-objektin alla. Toisen skenaarion pelitilanteessa elementit on jaettu kahden erillisen Canvas-objektin alle, dynaamisiin ja staattisiin elementteihin. Skenaarioiden hierarkiavertailun näkee kuvassa 17.

Kuva 17. Hierarkioiden vertailu [28].

Taulukosta 19 nähdään näiden kahden skenaarion suoritusajat Profilerin UI-moduulista. Tulosten perusteella voidaan todeta, että kanvaasien jakaminen kahteen kannatti, sillä nyt kun dynaamiset elementit päivittyvät, pelimoottorin tarvitsee rakentaa kokonaan uusiksi vain dynaamiset elementit. Samalla staattisten elementtien piirtämistä jatketaan kuten edelliselläkin ruudulla, ilman ylimääräisiä piirtokutsuja tai tekstuurien hakemista.

Taulukko 19. Testi kanvaasien jakamisesta dynaamisiin ja staattisiin.

Skenaario	**Layout-suoritus-aika tietokoneella**	**Renderöinnin suoritus-aika tietokoneella**	**Layout-suoritus-aika puhelimella**	**Renderöinnin suoritusaika puhelimella**
Kaikki elementit samassa kanvaasissa	0,02 ms	0,27 ms	0,05 ms	2,67 ms
Kanvaasit jaettu dynaamiseen ja staattiseen	0,015 ms	0,07 ms	0,035 ms	0,58 ms

Käyttöliittymien piilottamiseen ajon aikana on olemassa optimaalisemmat ja vähemmän optimaaliset tavat. Yksittäisen elementin piilottamiseen voisi tulla mieleen säätää elementin värin alfa-arvo eli läpinäkyvyys nollaan, mutta tämä edelleen aiheuttaa piirtokutsun. Parempi vaihtoehto yksittäisen elementin piilotukseen on käyttää komponentin tai GameObjectin IsActive-ominaisuutta [30, s. 224]. Koko kanvaasin piilottamiseen taas ei kannata käyttää GameObjectin IsActive-ominaisuutta, vaan nimenomaan Canvas-komponentin vastaavaa [35]. Jos Canvas-komponentin asettaa pois päältä, se lakkaa tekemästä piirtokutsuja grafiikkasuorittimelle, mutta ei hylkää verteksipuskuriaan (engl. Vertex Buffer), kun taas GameObjectin pois päältä asettaminen hylkää verteksipuskurin, joten kun kanvaasi halutaan asettaa takaisin aktiiviseksi, joudutaan ruudunpiirtoa varten piirtämään uusiksi kaikki verteksit.

Käyttöliittymäelementeillä on Raycast Target -asetus, jonka avulla pystyy määrittämään, onko elementin tarkoitus reagoida käyttäjän interaktioihin, kuten painalluksiin. Joka kerta, kun käyttäjän syötteitä käsittelevä EventSystem reagoi käyttäjän toimintaan, se käy for-silmukassa läpi kaikki kyseisen pikselin alueelta löytyvät elementit, joilla on Raycast Target -asetus päällä [30, s. 222]. Vähentääkseen operaation vaatimaa suorituskykyä, kannattaa asettaa Raycast Target -asetus pois päältä niiltä elementeiltä, joiden ei ole tarkoitus olla interaktiivisia.

Kanvaasin sisällä on syytä välttää päällekkäisiä elementtejä, erityisesti mobiilialustoille kehittäessä [36]. Päällekkäiset elementit aiheuttavat sitä, että Unityn

täytyy piirtää yksittäinen sama pikseli monta kertaa, ja tämä vie paljon grafiikkasuorittimen muistia. Sama pätee, vaikka käyttöliittymässä käytettävän kuvan kyseiset pikselit olisivat läpinäkyviä. Tästä syystä on esimerkiksi hyvä idea kytkeä muut kanvaasit pois päältä, jos niiden päälle piirtää esimerkiksi koko ruudun täyttävän taukovalikon.

4.5 Automaattisten layout-työkalujen optimoitu käyttäminen

Luvussa 3.2 esitellyt Automatic Layout Groups -komponentit ovat erinomaisia apureita layoutien tekemiseen käyttöliittymissä, mutta tuovat aina oman lisäkuormituksensa suorituskyvylle. Rubén Torres Bonet käy blogikirjoituksessaan ”Unity UI: My Top 3 Optimization Strategies” [36] hyvin läpi, kuinka kaksijakoinen aihe näiden työkalujen käyttäminen on, sillä mitä enemmän Unity UI:n “ekstroja” käyttää, sitä hitaammin peli päivittyy, mutta edut ja helppous, joita kehittäjä saavuttaa näitä käyttämällä tuovat merkittävän parannuksen kehitystyön mukavuuteen. Oikea tasapaino on taas kerran löydettävä puhtaan suorituskyky-optimoinnin ja kehittämisen sujuvuuden väliltä.

Suurin osa automaattisten layout-ryhmien käyttämisen tarpeesta tulee, kun elementtien määrä on dynaamisesti vaihteleva. Toki myös staattisia käyttöliittymiä voi toteuttaa näiden apuvälineiden avulla. Yksi tekniikka, jota Bonet suosittelee, on käyttää apuvälineitä avukseen käyttöliittymää rakentaessa, mutta kytkeä ne pois päältä ennen tallentamista. Tämä keino tosin toimii vain yksittäiselle resoluutiolle kehitettäessä ja kun elementtien määrän tiedetään pysyvän vakiona. Ian Dundore on käsitellyt Unityn optimointia puheessaan vuonna 2017 [35] ja korostanut, että layout-ryhmien käyttöä olisi hyvä välttää aina kun mahdollista. Kun yksikin ryhmään kuuluvista elementeistä muuttuu tai päivittyy, koko layout-järjestelmä lasketaan aina uusiksi. Tämä tulee erityisen raskaaksi sisäkkäisten layout-ryhmien kanssa. Joka kerta kun layout-ryhmään kuuluva elementti päivittyy, se kutsuu GetComponent-metodikutsua etsien layout-ryhmän määrittävää GameObjectia, ja näitä kutsuja kutsutaan hierarkiassa ylöspäin niin kauan, kuin objekteista löytyy jokin Automatic Layout Group -komponentti. GetComponent-

kutsut ovat raskaita, joten niiden jatkuvaa, moninkertaista kutsumista olisi hyvä välttää ja täten välttää sisäkkäisiä layout-ryhmien rakentamisia.

4.6 Merkkijonojen muodostaminen

Käyttöliittymissä näytetään hyvin usein pelaajalle jotakin informaatiota merkkijonoina, on informaatio sitten numeroita tai kirjaimia sisältävää. Erityisesti päivittyvissä elementeissä optimointi on tärkeää, ettei esimerkiksi mikään tekstimuodossa näytettävä mittari aiheuta suurta piikkiä prosessorin suorituskykyyn päivittäessään näytettävää arvoa.

Merkkijonoja luodaan C#-kielessä string-tyyppisillä muuttujilla, joille pyritään seuraavassa testissä löytämään optimaalisimmat käsittelytavat. Testiskenaarioissa iteraatioiden määrä oli tuhat, ja testissä tarkoituksena on päivittää TextMeshPro-tekstielementti näyttämään tiettyä tekstiä. Kuvassa 18 näkyvässä testissä skenaarioita oli neljä, joista ensimmäinen käytti perinteistä merkkijonojen ketjuttamista “+=”-operaattorilla, toinen “+”-operaattorilla, kolmas käytti C#-kielestä löytyvää StringBuilderia ja neljäs samaista StringBuilderia niin, että instanssi StringBuilderista tuli aina metodikutsun mukana. Lopulta näytettävään merkkijonoon lisättiin neljä merkkijonoa luokkamuuttujista ja toiset neljä dynaamisesti kovakoodaamalla.

Kuva 18. Testin 6 näkymä.

Esimerkkikoodi 20 sisältää ensimmäisen tekstin ensimmäisen skenaarion koodin kokonaisuudessaan.

```
[SerializeField] private TMP_Text TextToDisplay;
private string city = "Helsinkiin";
private string school = "Metropoliaan";
private string subject = "tietotekniikkaa";
private string major = "pelikehitystä";

private void Update()
{
    Profiler.BeginSample("test_6_case_A");
    for (int i = 0; i < iterations; i++)
    {
        UpdateText();
    }

    Profiler.EndSample();
}

private void UpdateText()
{
    string newText = "Tervetuloa opiskelemaan ";
    newText += school;
    newText += ", ";
    newText += city;
    newText += "! Täällä voi opiskella esimerkiksi ";
    newText += subject;
    newText += " ja pääaineena ";
    newText += major;
    newText += ".";
    TextToDisplay.SetText(newText);
}
```

Esimerkkikoodi 20. Merkkijonon ketjuttaminen.

Taulukosta 20 näkee ensimmäisen skenaarion suoritusajat ja roskienkeruuseen allokoidun muistin määrän. Tämä skenaario on melko hidas suorittaa ja allokoi paljon muistia. Koodi allokoi muistia joka kerta, kun merkkijonoa päivitetään.

Taulukko 20. Merkkijonon ketjuttaminen +=-operaattorin avulla.

Suoritusaika tietokoneella	**Muistiallokaatio tietokoneella**	**Suoritusaika puhelimella**	**Muistiallokaatio puhelimella**
9,61 ms	1500 kB	56,98 ms	1400 kB

Seuraavien skenaarioiden koodiesimerkit sisältävät vain UpdateText-metodin implementaation, muu osuus skriptistä on identtinen esimerkkikoodin 20 kanssa. Koodiesimerkissä 21 vähennetään merkkijonon muutokset vain yhteen operaatioon.

```
private void UpdateText()
    {
        string newText = "Tervetuloa opiskelemaan " + school + ", " +
city + "! Täällä voi opiskella esimerkiksi " +
                         subject + " ja pääaineena " + major + ".";
        TextToDisplay.SetText(newText);
    }
```

Esimerkkikoodi 21. Merkkijonon ketjuttaminen yhdellä operaatiolla.

Taulukosta 21 näkee, että nyt muistia allokoitiin yli kolme kertaa vähemmän ja suoritusaikakin lyheni merkittävästi.

Taulukko 21. Merkkijonon ketjuttaminen +-operaattorin avulla.

Suoritusaika tietokoneella	**Muistiallokaatio tietokoneella**	**Suoritusaika puhelimella**	**Muistiallokaatio puhelimella**
6,89 ms	476 kB	28,05 ms	363 kB

Koodiesimerkissä 22 siirrytään käyttämään C#-kielestä löytyvää StringBuilderia ja luodaan joka iteraatiolla uusi StringBuilder-luokan instanssi.

```
private void UpdateText()
    {
        StringBuilder stringBuilder = new StringBuilder();
        stringBuilder.Append("Tervetuloa opiskelemaan ");
        stringBuilder.Append(school);
        stringBuilder.Append(", ");
        stringBuilder.Append(city);
        stringBuilder.Append("! Täällä voi opiskella esimerkiksi ");
        stringBuilder.Append(subject);
        stringBuilder.Append(" ja pääaineena ");
        stringBuilder.Append(major);
        stringBuilder.Append(".");

        TextToDisplay.SetText(stringBuilder.ToString());
    }
```

Esimerkkikoodi 22. StringBuilderin alustus ja käyttäminen merkkijonon ketjuttamiseen.

Taulukosta 22 nähdään, että muistin allokoinnin perusteella testitulos sijoittuu kahden ensimmäisen skenaarion väliin. Suoritusaika on melko lähellä ensimmäistä skenaariota.

Taulukko 22. Merkkijonon ketjuttaminen StringBuilder-luokan avulla.

Suoritusaika tietokoneella	**Muistiallokaatio tietokoneella**	**Suoritusaika puhelimella**	**Muistiallokaatio puhelimella**
9,23 ms	800 kB	48,56 ms	700 kB

Esimerkkikoodissa 23 metodin implementaatio saa parametrina valmiiksi alustetun instanssin StringBuilderista ja käyttää sitä esitettävän merkkijonon rakentamiseen. StringBuilder-instanssi on alustettu for-silmukan ulkopuolella.

```
private void UpdateText(StringBuilder stringBuilder)
    {
        stringBuilder.Clear();
        stringBuilder.Append("Tervetuloa opiskelemaan ");
        stringBuilder.Append(school);
        stringBuilder.Append(", ");
        stringBuilder.Append(city);
        stringBuilder.Append("! Täällä voi opiskella esimerkiksi ");
        stringBuilder.Append(subject);
        stringBuilder.Append(" ja pääaineena ");
        stringBuilder.Append(major);
        stringBuilder.Append(".");

        TextToDisplay.SetText(stringBuilder.ToString());
    }
```

Esimerkkikoodi 23. Parametrina tulevan StringBuilderin käyttäminen merkkijonon ketjuttamiseen.

Taulukosta 23 nähdään, että viimeisin skenaario on testin selkeästi optimaalisin toteutustapa sekä muistiallokaation että suoritusajan suhteen.

Taulukko 23. Merkkijonon ketjuttaminen metodin parametrina saapuvan StringBuilder-luokan instanssin avulla.

Suoritusaika tietokoneella	**Muistiallokaatio tietokoneella**	**Suoritusaika puhelimella**	**Muistiallokaatio puhelimella**
5,86 ms	274,3 kB	20,87 ms	262,4 kB

Jos yhteen liitettävien merkkijonojen lukumäärä on hyvin vähäinen, iteraatioiden määrä on vähäinen ja metodikutsuja tehdään harvoin, voi hyvin käyttää myös perinteistä merkkijonojen ketjutusta, kunhan pyrkii tekemään sen

mahdollisimman vähäisellä muistiallokaatiolla käyttäen “+”-operaattoria. Operaation kasvaessa on hyvä harkita StringBuilderin käyttämistä.

4.7 Käyttöliittymien animointi

Luvussa 4.4 lajiteltiin käyttöliittymäelementtejä staattisiin ja dynaamisiin. Dynaamiset elementit ovat aina raskaampia suorituskyvylle, mutta ne ovat väistämättömiä, kun pelissä on päivittyvää dataa, jota on esiteltävä pelaajalle reaaliajassa. Dynaamiset käyttöliittymäelementit tuovat peliin eloa ja pelaajalle visuaalista palautetta. Dynaamisten elementtien harvoin halutaan vain päivittyvän lineaarisesti ja yksinkertaisesti, vaan usein erilaisilla efekteillä, tehosteilla tai animaatioilla halutaan herättää pelaajan huomio. Esimerkkinä päivittyvät resurssilaskurit voivat väliaikaisesti kasvaa, kun luku päivittyy tai elämäpistemittari voi vilkkua punaisena, kun pisteitä menettää.

Unityssa käyttöliittymäelementtejä voi animoida monin tavoin. Käytännössä Unityn animaatiotyökaluja voi käyttää myös käyttöliittymissä, mutta tämä ei missään nimessä ole optimaalisin tapa. Jos kanvaasin alaiseen objektiin on lisätty animaatioiden ajamista hoitava Animator-komponentti, kanvaasi “likaantuu” uudelleen laskettavaksi ja piirrettäväksi joka ruudunpäivityksellä, vaikka mitään animaatiota ei ajettaisi kyseisellä hetkellä [35]. Näitä Unityn perinteisiä animaatiotyökaluja on siis perusteltua käyttää elementeissä, joiden on tarkoitus päivittyä joka ruudulla, mutta vain toisinaan päivittyvien elementtien animointia varten on syytä harkita muita vaihtoehtoja. Yksi vaihtoehto on kirjoittaa itse animaatioita ajavia koodeja, esimerkiksi käyttämällä Unityn vuorottaisrutiineja ja lineaarista interpolointia. Toinen erinomainen vaihtoehto on Tween-kirjastojen käyttäminen. Yksi esimerkki ilmaisista Tween-kirjastoista on DOTween. Sen avulla pystyy helposti koodissa tekemään yksinkertaisia animaatioita, myös Unity UI -elementeille [37].

Esimerkkikoodissa 24 näkyy, miten projektiin tuodun DOTween-kirjaston avulla tehdään hyvin yksinkertainen animaatio, jossa elementti suurennetaan kaksinkertaiseksi.

```
transform.DOScale(new Vector3(2, 2, 1), 1f);
```

Esimerkkikoodi 24. DOTween-metodin käyttäminen skaalaukseen.

DOTween-kirjaston metodeja, kuten yllä käytettyä DOScale-metodia, voi kutsua suoraan GameObjectin transform-komponenttia kutsumalla. Argumenteiksi metodille syötetään ensin halutun skaalauksen lopputila vektorimuodossa ja toisena argumenttina animaation kesto liukulukuna.

Usein animaatioita halutaan ketjuttaa yhteen tai luoda niistä silmukka. Esimerkkikoodissa X luodaan silmukka, jossa elementtiä siirretään x-akselin suuntaisesti muuttujilla määritetyn matkan verran, niin että matka alusta päätepisteeseen ja takaisin kestää yhteensä 4 sekuntia ja animaatiosilmukka jatkaa loputtomiin tätä liikettä.

```
private void StartTweening()
{
   Sequence tweenSequence = DOTween.Sequence();
   tweenSequence.Append(transform.DOLocalMoveX(endPositionX, 2f))
     .SetEase(Ease.InQuad);
   tweenSequence.Append(transform.DOLocalMoveX(startPositionX, 2f)
     .SetEase(Ease.InQuad));
   tweenSequence.Play().SetLoops(-1);
}
```

Esimerkkikoodi 25. DOTween-kirjaston avulla tehty animointisilmukka.

DOLocalMoveX-metodikutsun jälkeen kutsuttavan SetEase-metodin avulla pystyy muuttamaan liikkeen animaatiokäyrää.

DOTween-metodeja käyttämällä niihin kytketyt elementit ovat dynaamisia vain animaatioiden hetkellä ja niiden ulkopuolella elementit käyttäytyvät kuin staattiset elementit. Tween-kirjastojen käyttäminen on siis hyvä ja optimoitu tapa luoda animaatioita käyttöliittymäelementeille. Niiden varjopuolena on kuitenkin se, että ne vaativat koodin kirjoittamista, eivätkä täten ole yhtä artistiystävällisiä

kuin visuaalisten animaatiotyökalujen käyttäminen. Isoissa projekteissa ohjelmoijat voivat kuitenkin kirjoittaa artistien ja suunnittelijoiden käytettäväksi valmiita komponentteja, jotka toteuttavat Tween-kirjaston avulla animaatioita, joita voi asettaa elementeille ja säädettävien muuttujien avulla säätää animaatioiden parametreja Inspector-näkymästä.

Esimerkkikoodissa 26 näyte valmiiksi kirjoitetusta animaatioskriptistä. Lisäämällä tämän MonoBehaviour-skriptin minkä tahansa GameObjectin komponentiksi, kyseistä objektia suurennetaan ja pienennetään jatkuvassa silmukassa. Koodi paljastaa Inspector-näkymään neljä muuttujaa, joiden avulla voi säätää, kuinka paljon elementtiä skaalataan, millä nopeudella ja minkälaisella animaatiokäyrällä.

```
public class ZoomLoopTween : MonoBehaviour
{
    [SerializeField] private float zoomAmount;
    [SerializeField] private float animatingInDuration;
    [SerializeField] private float animatingOutDuration;
    [SerializeField] private Ease;
    private Vector3 startScale;

    private void Start()
    {
        startScale = transform.localScale;
        StartTweening();
    }

    private void StartTweening()
    {
        transform.DOScale(new Vector3(zoomAmount, zoomAmount, 1), ani-
matingInDuration);

        Sequence mySequence = DOTween.Sequence();
        mySequence.Append(transform.DOScale(new Vector3(zoomAmount,
zoomAmount, 1), animatingInDuration).SetEase(ease));
        mySequence.Append(transform.DOScale(startScale, animat-
ingOutDuration).SetEase(ease));
        mySequence.Play().SetLoops(-1);
    }
}
```

Esimerkkikoodi 26. Elementtiä suurentavan työkalun toteutus DOTween-kirjaston avulla.

Animoituja käyttöliittymäelementtejä kehittäessä on hyvä pitää mielessä luvussa 4.4 esitetyt vinkit kanvaasien jakamisesta dynaamisiin ja staattisiin. Kaikkia

mahdollisia animointityökaluja oikein käyttäen voi saada pelinsä käyttöliittymän heräämään henkiin, tekemään vahvan vaikutuksen pelaajaan ja toimimaan samalla tehokkaasti ja optimoidusti.

5 Tulokset ja analyysi

5.1 Numeeriset tulokset

Taulukossa 24 ovat tämän insinöörityön ensimmäisen testin tulokset. Testissä käytiin läpi erilaisia tapoja hakea viittaus toiseen GameObject-instanssiin.

Taulukko 24. Testin 1 tulokset.

Skenaario	**500 iteraatiota tietokoneella**	**10000 iteraatiota tietokoneella**	**500 iteraatiota puhelimella**	**10000 iteraatiota puhelimella**
FindObjectOfType Update-silmukassa	1,60 ms	32,02 ms	7,64 ms	130,97 ms
FindObjectByTag Update-silmukassa	0,25 ms	4,62 ms	1,19 ms	14,92 ms
Objektin tallentaminen välimuistiin	0,18 ms	3,40 ms	0,98 ms	11,57 ms
Kaikkien luokkaviitteiden tallentaminen välimuistiin	0,00 ms	0,03 ms	0,00 ms	0,05 ms

Taulukossa 25 ovat koottuna toisen testin tulokset. Testin avulla havainnollistettiin, kuinka tärkeää on, että tallentaa luokkaviittaukset muuttujiksi välimuistiin, sen sijaan, että niitä hakee silmukassa raskaiden metodikutsujen avulla.

Taulukko 25. Testin 2 tulokset.

Skenaario	**Suoritusaika tietokoneella**	**Muistiallokaatio tietokoneella**	**Suoritusaika puhelimella**	**Muistiallokaatio puhelimella**
Luokkaviitteiden dynaaminen ratkaiseminen	0,16 ms	800 B	0,80 ms	480 B
Luokkaviitteiden tallentaminen välimuistiin	0,08 ms	420 B	0,45 ms	280 B

Taulukossa 26 ovat koottuna kolmannen testin tulokset. Testissä havainnollistettiin tyhjäksi jätettyjen Update-metodien tuoma lisäkuormitus, jos objekteja on paljon.

Taulukko 26. Testin 3 tulokset.

Skenaario	**Update-kutsujen lkm. tietokoneella**	**Update-metodien suoritusaika tietokoneella**	**Update-kutsujen lkm. puhelimella**	**Update-metodien suoritusaika puhelimella**
Tyhjät Update-metodit	5010	0,45 ms	5010	6,31 ms
Ei tyhjiä Update-metodeja	10	0,03 ms	10	0,06 ms

Taulukossa 27 ovat koottuna neljännen testin tulokset. Testissä käytiin erilaisia monialkioisia tietorakenteita läpi erilaisin keinoin, etsien suorituskyvyltään tehokkainta yhdistelmää.

Taulukko 27. Testin 4 tulokset.

Tietorakenne	**For-silmukka, tietokone**	**For-silmukka, puhelin**	**Foreach-silmukka, tietokone**	**Foreach-silmukka, puhelin**	**LINQ-kysely, tietokone**	**LINQ-kysely, puhelin**
Taulukko	0,67 ms (0 B)	1,36 ms (0 B)	0,60 ms (0 B)	1,17 ms (0 B)	3,85 ms (32 B)	5,42 ms (16 B)
Lista	1,60 ms (0 B)	1,73 ms (0 B)	1,71 ms (0 B)	1,96 ms (0 B)	2,72 ms (40 B)	4,45 ms (24 B)
IEnumerable	51,12 ms (600 000 B)	128, 71 ms (600 000 B)	4,16 ms (336 B)	7,70 ms (292 B)	4,84 ms (336 B)	8,64 ms (292 B)

Taulukossa 28 ovat koottuna viidennen testin tulokset. Testissä testattiin kanvaasien jakamista erikseen dynaamisia ja staattisia käyttöliittymäelementtejä varten.

Taulukko 28. Testin 5 tulokset.

Skenaario	**Layout-suoritusaika tietokoneella**	**Renderöinnin suoritusaika tietokoneella**	**Layout-suoritusaika puhelimella**	**Renderöinnin suoritusaika puhelimella**
Kaikki elementit samassa kanvaasissa	0,02 ms	0,27 ms	0,05 ms	2,67 ms
Kanvaasit jaettu dynaamiseen ja staattiseen	0,015 ms	0,07 ms	0,035 ms	0,58 ms

Taulukossa 29 ovat koottuna kuudennen testin tulokset. Testissä käytiin läpi erilaisia tapoja muodostaa merkkijonoja käyttöliittymän näytettäväksi.

Taulukko 29. Testin 6 tulokset.

Skenaario	**Suoritus-aika tieto-koneella**	**Muistiallo-kaatio tieto-koneella**	**Suoritusaika puhelimella**	**Muistiallo-kaatio puhe-limella**
String-luokan =+-operaattori	9,61 ms	1500 kB	56,98 ms	1400 kB
String-luokan +-operaattori	6,89 ms	476 kB	28,05 ms	363 kB
StringBuilder	9,23 ms	800 kB	48,56 ms	700 kB
Metodin para-metrina lähe-tettävä StringBuilder-instanssi	5,86 ms	274,3 kB	20,87 ms	262,4 kB

5.2 Analyysi

Testitulosten perusteella voidaan tehdä johtopäätöksiä siitä, mitkä metodit, tietorakenteet, kirjastot, työkalut ja ohjelmointimallit tarjoavat tehokkainta suorituskykyä mitattavissa olevilla mittayksiköillä, kuten suorituksen kesto tai allokoitavan muistin määrä, mutta optimoidussa ohjelmistokehityksessä nämä seikat eivät ole yksipuolisesti ainoita tekijöitä. Kokonaisuus jokaisessa omassa kontekstissaan on aina monimutkaisempi kuin yksittäiset testitulokset. Suorituskyvyltään raskaanpuoleinen kirjasto saattaa tarjota kehittäjän työtä todella suuresti helpottavia työkaluja ja metodeita. Joissain tapauksissa millisekunnin osan tehokkaampi koodi voi olla huomattavasti vaikeammin luettavaa kuin toinen vaihtoehto ja täten vaikeuttaa useasta kehittäjästä koostuvan työryhmän työskentelyä. Hyvä idea on valita pelille tavoite ruudunpäivitystiheyteen ja ruudun piirtämiseen menevään aikaan ja profilointia tehdessä reagoida vain piikkeihin, jotka ylittävät tämän rajan. Muuten on riskinä, että vaikka kuinka paljon optimointiin käyttäisi aikaa, se ei vaikuta loppukäyttäjään millään tavalla. Koodin muuttaminen tuo aina myös mahdollisuuden uusien ohjelmointivirheiden syntymiselle.

Tasapainoa olisi löydettävä senkin suhteen, missä vaiheissa kehitystä panostaa optimointiin minkäkin verran. Pelikehityksen alkuvaiheessa ei kannata jäädä liian pitkäksi aikaan jumiin hiomaan optimointia viimeisen päälle, kun ei vielä tiedä, kuinka tarpeellista se on kokonaiskuvassa. Ei ole kuitenkaan myöskään hyvä jättää aivan projektin loppuvaiheille suurta optimointityötä, vaan keskeisimmät optimoidun koodin periaatteet olisi hyvä pitää mielessä alusta asti peliä kehittäessä.

Oleellisimpia asioita Unity-ohjelmoinnin ja käyttöliittymien optimoinnissa on suorittaa operaatiot, varsinkin raskaat, vain silloin kun ne tarvitsevat suorittaa, eikä koskaan turhaan. Raskaat komennot, kuten Unityn GetComponent-metodikutsut, on syytä siirtää pois silmukoista ja korvata muuttujien tallentamisella välimuistiin Start- tai Awake-metodeissa. Unity UI:n keskeisimmän elementin eli Canvas-komponentin optimoitu käyttö auttaa pitkälle käyttöliittymien suorituskyvyssä. Unity UI -järjestelmän lisätyökaluja, kuten automaattisia layout-ryhmiä, on syytä käyttää maltillisesti. Animaatiotyökalujen käyttämistä on hyvä välttää ja suosia Tween-kirjastoja käyttöliittymien animointiin. Tässä insinöörityössä monesti lähteenä käytetty teos Unity Game Optimization Book [30] kiteyttää optimoinnin tarpeen hyvin: “Miettiessä suorituskykyongelman korjaamisen tarpeellisuutta, on syytä kysyä kysymys, tuleeko käyttäjä huomaamaan korjausta. Jos vastaus on ei, optimointi saattaa olla tarpeetonta.” Optimoinnissa on siis syytä, kuten kaikessa pelikehitystyössä, pitää jatkuvasti mielessä tuotteen loppukäyttäjä, eli videopelin pelaaja ja hänen käyttökokemuksensa.

5.3 Käyttöliittymien tulevaisuus

Käyttöliittymien tulevaisuus näyttää yhtä valoisalta ja kiinnostavalta kuin videopelienkin tulevaisuus. Pelejä tehdään koko ajan enemmän ja monipuolisemmin, eikä ole mitään syytä, miksi käyttöliittymien merkitys katoaisi videopeleistä. Virtuaalitodellisuus on jatkuvasti kasvava osa-alue peliteollisuudessa, ja se on mahdollisesti monen pelinkehittäjän julkaisualusta tulevaisuudessa. Virtuaalitodellisuus tuo omat lisähaasteensa käyttöliittymien suunnittelijoille ja kehittäjille ratkaistavaksi. Kuvakulman erilaisuus on kenties suurin eroavaisuus verrattuna

perinteisiin näytöltä pelattaviin videopeleihin. Kun pelaaja katselee pelimaailmaa kolmiulotteisessa ympäristössä koko pelikokemuksen läpi, ei käyttöliittymiä, kuten valikkoja tai heijastusnäyttöjä, voi suunnitella aivan samoin kuin kaksiulotteiselta näytöltä pelattaviin peleihin. Tämän lisäksi virtuaalitodellisuuspelit vaativat tarkkaa optimointia, sillä pelien on toimittava korkealla päivitystaajuudella. Tutkimukset osoittavat, että alle 90 kuvaa sekunnissa päivittyvät virtuaalitodellisuuspelit aiheuttavat pelaajille huonovointisuutta ja matkapahoinvointimaista oloa [38]. Käyttöliittymien suunnittelussa on alettu viime vuosina [39] panostamaan yhä enemmän käytettävyyteen ja saavutettavuuteen, kuten säädettäviin fontin kokoihin, puheohjauksiin ja eri väriasetustiloihin, jotta mahdollisimman monet ihmiset pystyisivät nauttimaan videopeleistä.

Perinteiset pelilaitteet kehittyvät jatkuvasti tehokkaammiksi, mikä sinänsä vähentää tarkan optimoinnin tarvetta, mutta tulevaisuuden teholaitteillekin kehittäessä on kahdesta syystä hyvä pitää opitut optimointikeinot mielessä. Ensimmäinen syy on se, että vaikka markkinoiden tehokkaimmat pelitietokoneet tai puhelimet olisivat kuinka tehokkaita, käyttäjillä on kaikentehoisia pelilaitteita vuosien varrelta. Mitä menestyksellisempää laitteiden tehon kehitys on, sitä helpommin kehittäjiltä unohtuu, kuinka heikkotasoisia laitteita suurella määrällä potentiaalisista pelaajista on. Steamin tutkimuksen [18] mukaan pelaajien suosituin grafiikkasuoritin vuonna 2022 oli Nvidian GTX 1060, joka on vuonna 2016 julkaistu näytönohjain [40]. Kehittäjien on siis hyvin tärkeää miettiä, mille kohdeyleisölle haluavat pelejään kehittää, selvittää kohdeyleisön laitekanta ja kehittää sille, eikä vain seurata uusimpien ja tehokkaimpien laitteiden suorituskykyä. Toinen syy jatkaa optimointia laitteiden suorituskyvynkin kasvaessa on se, että ylimääräinen suorituskyky on todella paljon mielekkäämpää käyttää kaikkeen muuhun pelikokemusta parantavaan, kuten näyttävään grafiikkaan, laajoihin pelin sisäisiin järjestelmiin, uskottaviin animaatioihin ja luovaan tarinankerrontaan, eikä yksinkertaisen käyttöliittymän kehnon optimoinnin synnyttämään kuormitukseen.

6 Yhteenveto

Käyttöliittymien merkitys videopeleissä on suuri, ja niiden riittävä optimointi auttaa tarjoamaan pelaajalle mieluisan käyttökokemuksen. Insinöörityössä tehdyt testit ja niiden tulokset tarjoavat tietoa optimaalisista keinoista moniin käyttöliittymäohjelmoijan usein kohtaamiin ongelmiin. Osa tuloksista oli etukäteen ennakoitavissa loogisesti ajattelun avulla, mutta oli mielenkiintoista nähdä tarkkana datana, kuinka suuria tai pieniä eroavaisuudet eri skenaarioissa olivat. Tulosten ja lähdemateriaalien avulla havaittiin myös, että aina ei ole selvästi parasta yleistä vaihtoehtoa tarjolla, vaan jokainen peli, tiimi ja ongelma luo oman kokonaisuutensa, jota täytyy tarkastella monesta näkökulmasta, ottaen huomioon muitakin tekijöitä kuin ohjelman suorituskyky.

Insinöörityöstä syntyi lopulta kohtuullisen kattava tietopaketti käyttöliittymistä, Unitysta, optimoinnista ja pelikehityksestä, ja sen voisi uskoa olevan hyödyllinen erityisesti pelialan opiskelijoille ja vastavalmistuneille pelikehittäjille, jotka työskentelevät Unity-kehitysympäristössä ja toteuttavat videopelien käyttöliittymiä.

Insinöörityössä onnistuttiin toteuttamaan kaikki tärkeimmät osa-alueet, joita alussa oli suunniteltu, sekä yhdistelemään mielenkiintoisista lähteistä monipuolista lisätietoa aiheesta. Insinöörityön tekeminen opetti paljon uusia tekniikoita ja optimointitapoja, joista varmasti on hyötyä pelikehittäjän ammatissa.

Lähteet

1 Käyttöliittymä. Verkkoaineisto. Kielitoimiston sanakirja. <https://www.kielitoimistonsanakirja.fi/>. Luettu 25.4.2022.

2 Bowers, Micah. Level Up: A Guide to Game UI (with Infographic). Verkkoaineisto. Toptal. <www.toptal.com/designers/gui/game-ui>. Luettu 25.4.2022.

3 Marshall, Sarah. The Ultimate Guide To Visual Hierarchy. Verkkoaineisto. Canva. <www.canva.com/learn/visual-hierarchy/>. Luettu 25.4.2022.

4 Shcherbinina, Anna. 2020. A Look Into Game UI: From 1960s to the Present. Verkkoaineisto. 80 Level. <www.80.lv/articles/a-look-into-games-ui-from-1960s-to-the-present///>. Luettu 25.4.2022.

5 Chow, Steph. 2018. Immersing a Creative World into a Usable UI. Verkkoaineisto. GDC. <https://youtu.be/ONYmBBZiBj8>. Luettu 25.4.2022.

6 Overwatch. Verkkoaineisto. Interface In Games. <https://interfaceingame.com/games/overwatch/>. Luettu 25.4.2022.

7 Good Design, Bad Design - The Best & Worst of Graphic Design in Games ~ Design Doc. Verkkoaineisto. Design Doc. <https://youtu.be/bE_ZuNp1CTI>. Luettu 25.4.2022.

8 Schell, Jesse. 2015. The Art of Game Design. 2nd ed. Florida: CRC Press.

9 Bowers, Micah. Level Up: A Guide to Game UI (with Infographic). Verkkoaineisto. Toptal. <www.toptal.com/designers/gui/game-ui>. Luettu 25.4.2022.

10 Alien: Isolation. 2014. California: SEGA Of America.

11 Watch Dogs. 2014. Montreuil: Ubisoft.

12 Joutjärvi, Jutta. 2016. Skeuomorfismi ja flat design – visuaaliset tyylit ja käytettävyys älypuhelimen käyttöliittymässä. Taiteen maisterin opinnäyte. Aalto-yliopisto. Aaltodoc-tietokanta.

13 Pixel Perfect Precision. 2014. Verkkoaineisto. Ustwo. <https://www.ustwo.com/blog/pixel-perfect-precision-handbook/>. Luettu 25.4.2022.

14 Gotham City Impostors. California: Warner Bros. Interactive Entertainment.

15 Astro's Playroom. California: Sony Interactive Entertainment.

16 Raj, Ajai. 2020. UI and UX in Tactical Games: Three Considerations. Verkkoaineisto. Medium. <https://www.medium.com/games-r-ux/ui-and-ux-in-tactical-games-three-considerations-82c546e9e48/>. Luettu 25.4.2022.

17 Impey, Sarah. 2018. Mobile vs Desktop UI: Key Differences In Design. Verkkoaineisto. Game Analytics. <www.gameanalytics.com/blog/mobile-desktop-ui-design/>. Luettu 25.4.2022.

18 Steamin laitteisto- ja ohjelmistokysely: March 2022. Verkkoaineisto. Steam. <https://store.steampowered.com/hwsurvey/Steam-Hardware-Software-Survey-Welcome-to-Steam>. Luettu 25.4.2022.

19 Cryer, Paul. Measuring up – reference table for iOS device dimensions. Verkkoaineisto. TwentyEightB. <https://28b.co.uk/ios-device-dimensions-reference-table/>. Luettu 25.4.2022.

20 I finally understand what the iPhone X 'notch' is for. Verkkoaineisto. Mashable. <https://mashable.com/article/iphone-x-notch-different>. Luettu 25.4.2022.

21 Hurff, Scott. How to design for thumbs in the Era of Huge Screens. Verkkoaineisto. <https://www.scotthurff.com/posts/how-to-design-for-thumbs-in-the-era-of-huge-screens/>. Luettu 25.4.2022.

22 Jasmin, Marie. Building the Interface of The Elder Scrolls: Blades in Landscape and Portrait. Verkkoaineisto. GDC. <https://www.youtube.com/watch?v=Tmt5v0bL1-Y&t=550s>. Luettu 25.4.2022.

23 Borromeo, Nicolas Alejandro. 2021. Hands-on Unity 2021 Game Development. 2nd ed. Birmingham: Packt.

24 Haas, John. 2014. A History of the Unity Game Engine. An Interactive Qualifying Project. Worcester Polytechnic Institute.

25 Welcome to Unity - Key facts. Verkkoaineisto. Unity. <www.unity.com/our-company>. Luettu 25.4.2022.

26 Comparison of UI systems in Unity. Verkkoaineisto. Unity. <https://docs.unity3d.com/2020.2/Documentation/Manual/UI-system-compare.html>. Luettu 25.4.2022.

27 Godbold, Ashley. 2018. Mastering UI Development. Birmingham: Packt.

28 Unity 2020.3.26f1. 2020. California: Unity Technologies.

29 Schardon, Lindsay. A Guide to the Unity Device Simulator. Verkkoaineisto. GameDev Academy. <https://gamedevacademy.org/unity-device-simulator-webclass/>. Luettu 25.4.2022.

30 Aversa, Davide & Dickinson, Chris. 2019. Unity Game Optimization. Birmingham: Packt.

31 Unity Documentation. Verkkoaineisto. Unity. Luettu 25.4.2022.

32 Fundamentals of garbage collection. Verkkoaineisto. Microsoft. <https://docs.microsoft.com/en-us/dotnet/standard/garbage-collection/fundamentals>. Luettu 25.4.2022.

33 Language integrated Query. Verkkoaineisto. Microsoft. <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/>. Luettu 25.4.2022.

34 Optimize Your Games In Unity – The Ultimate Guide. Verkkoaineisto. Awesome Tuts. <https://awesometuts.com/blog/optimize-unity-game/>. Luettu 25.4.2022.

35 Dundore, Ian. 2017. Squeezing Unity: Tips for raising performance. Verkkoaineisto. Unity. <https://www.youtube.com/watch?v=_wxitgdx-UI>. Luettu 25.4.2022.

36 Bonet, Rubén Torres. Unity UI: My Top 3 Optimization Strategies. Verkkoaineisto. The Gamedev Guru. <https://thegamedev.guru/unity-ui/optimization-strategies/>. Luettu 25.4.2022.

37 DOTween Documentation. Verkkoaineisto. Demigiant. <http://dotween.demigiant.com/documentation.php>. Luettu 25.4.2022.

38 The Importance of Frame Rates. Verkkoaineisto. IrisVR. < https://help.irisvr.com/hc/en-us/articles/215884547-The-Importance-of-Frame-Rates>. Luettu 25.4.2022.

39 Shin, Mira. A Growth of Accessibility in Video Games. Verkkoaineisto. DO-IT. < https://www.washington.edu/doit/growth-accessibility-video-games>. Luettu 25.4.2022.

40 NVIDIA GeForce GTX 1060. Verkkoaineisto. TechPowerUp. <https://www.techpowerup.com/gpu-specs/geforce-gtx-1060-6-gb.c2862>. Luettu 25.4.2022