Jouni Peltonen

Tietojen tallennusjärjestelmät pelinkehityksessä

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikan tutkinto-ohjelma

Insinöörityö

20.4.2018

Metropolia

Tekijä Otsikko Sivumäärä Aika	Jouni Peltonen Tietojen tallennusjärjestelmät pelinkehityksessä 34 sivua 20.4.2018
Tutkinto	Insinööri (AMK)
Tutkinto-ohjelma	Tieto- ja viestintätekniikka
Ammatillinen pääaine	Pelisovellukset
Ohjaaja	Lehtori Antti Laiho

Insinöörityönä tehtiin tietojen tallennusjärjestelmä muokattavia objekteja eli ScriptableObjecteja hyväksikäyttäen. Järjestelmä toteutettiin Unity-pelinkehitysympäristössä, ja sen käyttöä varten ohjelmoitiin myös oma Unityn editorilaajennus.

Tallennusjärjestelmä oli insinöörityön tekijän itse valitsema, määrittelemä, toteuttama ja testaama. Työ ei ollut mikään virallisen yrityksen tilaama, mutta tallennusjärjestelmälle on käyttöä tulevissa pelinkehitysprojekteissa. Tallennusjärjestelmään kannattaa perehtyä varsinkin kaikkien Unitya käyttävien pelinkehittäjien, mutta siitä voi olla hyötyä myös muille pelien tai ohjelmistojen kehittäjille.

ScriptableObjectien käytön ansiosta toteutettu tallennusjärjestelmä soveltuu suurienkin tiedon määrien tallentamiseen ja käsittelyyn. Tallennusjärjestelmä on rakenteeltaan yksinkertainen ja selkeä. Koko tallennusjärjestelmä sijaitsee yhden päätiedoston alla. Tallennusjärjestelmä koostuu määrittelemättömästä määrästä listoja, jotka kaikki voivat sisältää myös halutun määrän esimerkiksi asioita, esineitä tai olentoja. Järjestelmässä näitä kaikkia kutsutaan yhteisellä nimellä ”item”. Listojen käyttötarkoitus ja sisältö ovat käyttäjän itse määriteltävissä. Ehtona kuitenkin on, että kaikilla samalla listalla olevilla item-tyyppisillä asioilla ovat samat muuttujat, kuten esimerkiksi merkkijonot, numerosarjat tai värit. Muuttujilla ovat myös yhtenevät nimet, mutta niiden arvot voivat kuitenkin vaihdella.

Järjestelmä saatiin valmiiksi ja toimivaksi sille tehtyjen määritelmien ja suunnitelmien mukaisesti. Sitä toteuttaessa oli monia vastoinkäymisiä ja haasteita, mutta ne kaikki saatiin lopulta ratkaistua. Tallennusjärjestelmään on myös monia jatkokehitys- ja parannusmahdollisuuksia, kuten esimerkiksi järjestelmän varmuuskopiointi nappulaa painamalla.

Avainsanat	Unity, tallennusjärjestelmä, muistinhallinta, C#

Author Title Number of Pages Date	Jouni Peltonen Data saving systems in game development 34 pages 20 April 2018
Degree	Bachelor of Engineering
Degree Programme	Information and Communication Technology
Professional Major	Game Applications
Instructor	Antti Laiho, Senior Lecturer

Project part of the final year project was making a data saving system using ScriptableObjects. The saving system was done in Unity game development platform, and own Unity editor extension was created for the purpose of using it.

Saving system was designed, built and tested, by the author of this thesis. The project was not issued by any official firm, but the data saving system will come in handy in a future game development projects. Saving system will be most useful for those that make games with Unity, but there might be helpful information also for other game or software developers.

ScriptableObjects are especially useful for storing a significant amount of data, and that is why the saving system is also great for the purpose. The saving system is so simple and clear that the whole system is under one central file. Under that main file are multiple lists. There is no limit to the number of lists in the system. All the lists can also contain an indefinite number of items. Items can be for example objects, animals or anything else. Number or type of lists and the items can be decided entirely by the user of the saving system. There is a limitation that all the variables are the same in items that are on the same list. Names and types of those variables are also the same in every item on the same list. Values of those variables can be changed for every item individually. Possible variables are for example strings, objects or colors.

The project was completed successfully according to the specifications, and the data saving system is functional. There were many problems and challenges on the way, but all of those got solved by the end. There are many further development and improvement options still left that could be implemented to the data saving system in the future, for example, system back up with the press of a button.

Keywords	Unity, data saving system, memory management, C#

Sisällys

Lyhenteet

OOP	Objektikeskeinen ohjelmointimalli (engl. Object-oriented design).
DOD	Datakeskeinen suunnittelumalli (engl. Data-oriented design).
DMA	Oikosiirto tarkoittaa, että data voidaan käsitellä sitä siirtämättä (engl. Direct memory access).
HP	Elämäpisteet (engl. Hit Points).
DMG	Vahinko tai hyökkäysarvo (engl. Damage).

1 Johdanto

Insinöörityönä tehdään Unity-pelinkehitysympäristöön muokattavia objekteja eli ScriptableObjecteja käyttävä tallennusjärjestelmä. Tallennusjärjestelmä tehdään itse aina suunnittelusta ja toteutuksesta toimivuuden varmistukseen asti. Unityssä on valmiiksi rakennettuna mahdollisuudet tehdä juuri tämänkaltaisia omia lisäosia. Työ tehdään C#-ohjelmointikieltä käyttäen, joten kaikki koodiesimerkit ovat myös tällä kielellä.

Tietotekniikan alalla on monenlaisia ohjelmointi- ja suunnittelumalleja. Mallin valinnalla on yllättävän suuret vaikutukset muistin ja tiedon hallintaan. Unity-pelinkehitysympäristössä tietojen tallennukseen ja hallintaan ovat yleisesti käytössä MonoBehaviour-skriptit (engl. MonoBehaviour script), joilla myös ohjataan editorin käyttäytymistä. Unityssä on myös vaihtoehtoisia tapoja käsitellä tietoja, kuten ScriptableObjectit. Ne soveltuvat parhaiten tiedon tallentamiseen. ScriptableObjectin käytössä tiedon tallentamiseen on monia hyötyjä MonoBehaviour-skripteihin verrattuna, mutta siitä on paljon vähemmän ohjeita ja oppaita.

ScriptableObjectien käyttö on saanut hieman enemmän näkyvyyttä ja muutamia Unityn itse julkaisemia oppaita viime vuosina [1]. ScriptableObjectien käytön lisääntymisestä kertoo esimerkiksi se, että tunnettuja Unityä käyttäviä peliyhtiöitä on siirtynyt hyödyntämään niitä. Yhtenä esimerkkinä on peliyhtiö Schell Games, jonka pääinsinööri Ryan Hipple on pitänyt esityksen ScriptableObjecteista. Esitys julkaistiin Unityn Youtube-kanavalla 20.11.2017. [2.] Esityksessä Ryan Hipple kertoo ja näyttää esimerkkejä, kuinka peliyhtiö käyttää hyödyksi ScriptableObjecteja monilla eri osa-alueilla.

Insinöörityöraportti on jaoteltu niin, että toisessa luvussa keskitytään yleisesti käytössä oleviin ohjelmointi- ja suunnittelumalleihin sekä niiden tarjoamiin tallennusjärjestelmiin. Tallennusjärjestelmiä tarkastellaan pelinkehityksen näkökulmasta. Työn kolmannessa luvussa perehdytään nimenomaan Unity-pelinkehitysympäristössä oleviin tiedon tallennusjärjestelmiin. Tallennusjärjestelmiä vertaillaan keskenään, ja luvun lopussa käydään erot läpi vielä tarkemmin esimerkkitapauksen muodossa. Neljännessä luvussa selostetaan insinöörityön projektiosuus. Viidennessä luvussa käydään läpi tallennusjärjestelmien tulevaisuuden näkymiä ja työn tuloksia.

2 Tiedon tallennusjärjestelmät

Pelien ja muiden ohjelmistojen tekemisen lähestymiseen on monia suunnittelu- ja ohjelmointimalleja. Suunnittelu- ja ohjelmointimallin valintaan kannattaa kiinnittää erityistä huomiota, koska sillä voi olla suuria vaikutuksia pelin kehityksen eri vaiheissa aina valmiiseen peliin asti. Valittua mallia on monesti vaikea tai resurssien puitteissa jopa mahdoton vaihtaa myöhemmin. Mallin valinnasta johtuvat ongelmat saattavat ilmetä vasta, kun pelin kehittäminen on edennyt pidemmälle ja projektin eri osien, objektien tai asioiden pitäisi keskustella keskenään.

Pelialalla yleisimmin käytössä oleva malli on objektikeskeinen ohjelmointimalli (engl. Object-Oriented Programming), josta käytetään lyhennettä OOP. Pelialalla on myös yleistymässä datakeskeinen suunnittelumalli (engl. Data-Oriented Desing), josta käytetään lyhennettä DOD. Näiden kahden lisäksi yleisesti ohjelmiston kehityksessä on käytössä myös esimerkiksi funktionaalinen ohjelmointi (engl. Functional Programming) ja proseduraalinen ohjelmointi (engl. Procedural Programming), mutta nämä eivät ole niin yleisiä pelialalla. Funktionaalista ohjelmointia tosin käytetään koneoppimiseen (engl. Machine learning), josta saattaa olla hyötyä myös pelialalla. Ohjelmointi mallin valinnalla on suorat vaikutukset pelin tietojen tallentamiseen ja muistin käyttöön. Pari vuosikymmentä sitten pelien ohjelmointi muistutti enemmän DOD-mallia, mutta se on pitkälti saanut väistää OOP-mallin yleistyessä.

2.1 Tallennusjärjestelmien historiaa

Tietokoneiden kehityksen alkuaikoina useimpia tietokoneen osia ja komponentteja kehitettiin yhtä aikaa, ja tämän vuoksi ne olivat suurimmilta osin tasapainossa keskenään. Esimerkiksi muisti ja verkkokäyttöliittymä olivat lähes yhtä nopeita. Tietokoneiden rakenteen yhtenäistymisen myötä laitteistokehittäjät alkoivat optimoida ja panostaa enemmän koneen yksittäisiin osiin. Toisien komponenttien kehitys lähti nousuun ja toiset jäivät kehityksessä jälkeen. Kaikista suurimmaksi pullonkaulaksi muodostuivat massamuistit, joiden nopeus jäi kehityksessä huomattavasti muiden jälkeen. [3.]

Massamuistien hitautta on yritetty kompensoida ohjelmistopuolella pitämällä todennäköisimmin tarvittavat tiedot päämuistissa (engl. Main Memory), joka on monin kerroin

nopeampaa kuin kiintolevyjen massamuistit. Ohjelmistopuolen lisäksi muutoksia on jouduttu tekemään myös laitteiston puolella. Vaikka päämuisti on massamuistia huomattavasti nopeampaa, se ei silti ole tarpeeksi nopeaa muihin tietokoneen komponentteihin verrattuna. Päämuistin aiheuttaman pullonkaulan helpottamiseksi laitteistopuolella on käytössä nykypäivänä esimerkiksi RAM-muistit, CPU-välimuistit (engl. CPU-cache) ja DMA-laitteet. [3.]

Pelialalla yleisin käytössä oleva ohjelmointikieli on C++. Sen historia ylettyy aina 1970-luvulle asti, jolloin Bjarne Stroustrup alkoi kehittää ohjelmointikieltä, joka olisi kuin ”C-ohjelmointikieli luokilla” (engl. ”C with classes”). Tarkoituksena oli näin lisätä objektikeskeinen ohjelmointi C-kieleen. C-kieli oli silloin ja on edelleen hyvin arvostettu ohjelmointikieli, sen monille alustoille siirrettävyyden ansiosta ilman, että joudutaan luopumaan nopeudesta tai matalan tason toiminnollisuudesta. [4.]

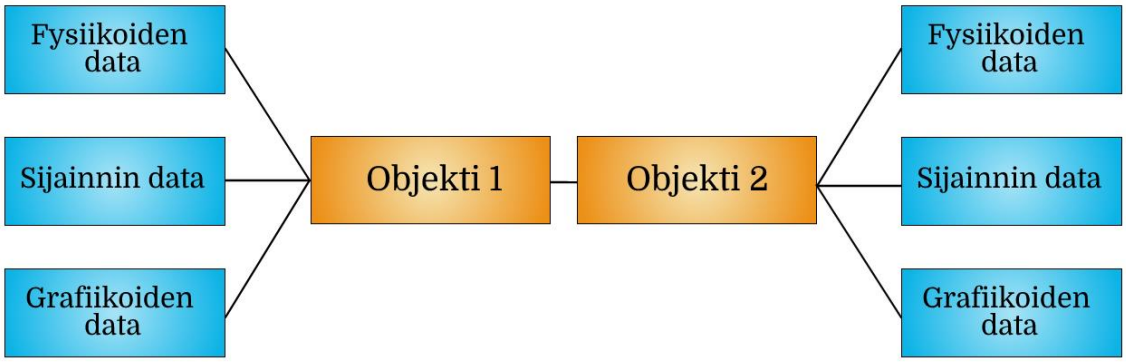
C++ on kielenä niin monipuolinen että sitä voi käyttää OOP-mallin lisäksi myös funktionaaliseen ja proseduraaliseen ohjelmointimalliin. Koska C++-kieli on objektikeskeistä, myös sillä ohjelmoidut ja sen pohjalta tehdyt pelimoottorit ovat olleet pääasiassa objektikeskeisiä. Datakeskeistä suunnittelumallia käyttäen ohjelmoidut pelit ja sovellukset muistuttavat koodiltaan C-ohjelmointikielen perus rakenteita. Koska C-kieli on perustana C++-kielelle, sitä on käytetty hyväksi pelien tekemisessä kautta aikojen, mutta on mielenkiintoista nähdä, yleistyykö se lisää DOD-mallin suosion kasvaessa.

2.2 Objektikeskeiset tallennusjärjestelmät pelinkehityksessä

Maailmassa on lukuisia pelimoottoreita (engl. Game Engine) ja niiden ympärille rakennettuja pelinkehitysympäristöjä. Yleisimmät julkisessa käytössä olevat pelinkehitysympäristöt ovat Unity, Unreal Engine ja CryEngine. Ne kaikki on pohjimmiltaan C++-ohjelmointikieltä käyttäviä pelimoottoreita. Unityä ja CryEngineä käytettäessä pelimoottoria ohjaavat skriptit kirjoitetaan muilla kielillä, kuten esimerkiksi C#-ohjelmointikielellä. Uudempana tulokkaana suosiota on kerännyt myös Amazon Lumberyard, joka julkaistiin helmikuussa 2016 [5]. Amazon Lumberyard on uudelleen rakennettu ja lisäosilla varustettu CryEnginen ympärille rakennettu pelinkehitysympäristö [6]. Edeltäjänsä tapaan Amazon Lumberyard on C++-kieleen pohjautuva pelinkehitysympäristö, johon skriptejä voi kirjoittaa myös Lua-ohjelmointikielellä.

Koska pelimoottorit pohjautuvat C++-kieleen, myös loogisesti skriptit ja niiden käyttö ovat ohjautuneet objektikeskeiseen ohjelmointimalliin eli OOP-malliin. OOP-malli tarkoittaa tiedon ja asioiden tallennusjärjestelmän kannalta, että keskitytään objekteihin ja asioihin. Näillä objekteilla on yleensä osinaan erilaista dataa ja komponentteja, jotka voi liittyä esimerkiksi kappaleen sijaintiin, käyttäytymisen, fysiikan mallinnukseen tai graafiseen esitykseen.

Suurin osa ohjelmoijista on tottunut käsittelemään ja ajattelemaan asioita OOP-mallin mukaan (kuva 1). Unityn ja muiden pelinkehitysalustojen toimiminen OOP-mallin mukaisesti on osasyy sille, että se on hyvin intuitiivinen ja helposti lähestyttävä. OOP-malli tekee Unityllä tehdyistä projekteista helpommin rakennettavia, huollettavia ja ylläpidettäviä, sekä joustavia.

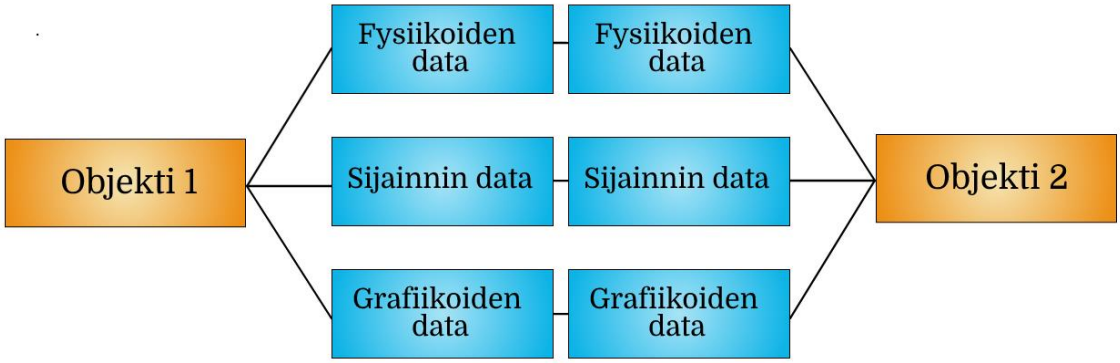


Kuva 1. Yksinkertaistettu esimerkki OOP-mallista [26].

OOP-mallin käytön luontevuutta voi selittää myös se, että se on yleinen lähestymistapa tosimaailman tilanteissa. Mikäli kotiin etsitään pöytää, on helppoa, että kaikki pöydät löytyvät niitä myyvästä kaupasta eikä myytäviä asioita ole jaoteltu esimerkiksi korkeuden mukaan eri kauppoihin. Pöydän löytäminen olisi vaikeaa, jos pitäisi käydä kaikki kaupat läpi tai valita pöytä pelkästään korkeuden mukaan. Tietokonemaailmassa ja erityisesti pelinkehityksessä kaikki ”pöydät” eivät ole samanlaisia. Osa pöydistä voi olla esimerkiksi rikkoutuvia, liikutettavia tai rikkoutumattomia. Kaikki näistä pöydistä ovat sisällöltään ja tiedoiltaan aivan erilaisia. OOP-mallissa muistin käsittelyn kannalta heikkoutena onkin se, että objektikeskeisessä ajattelussa eri objektien, esimerkiksi fysiikan mallinnukseen liittyvä, data on aivan eri paikoissa muistia. Muistin käytön ja käsittelyn ongelmaa on korjattu lähestymällä tallennusjärjestelmiä datakeskeisesti.

2.3 Datakeskeiset tallennusjärjestelmät pelinkehityksessä

Datakeskeinen suunnittelumalli tarkoittaa pelinkehityksen tallennusjärjestelmien kannalta, että objektien sijaan keskitytään lähtökohtaisesti ainoastaan dataan. Erityyppiselle ja eri tarkoitukseen olevalle datalle on omanlainen tallennusjärjestelmä. Tarkoituksena on tallentaa samanlainen tai samaan käyttötarkoitukseen oleva data samaan paikkaan (kuva 2). Dataa ei myöskään käytetä samoissa paikoissa kun sitä tallennetaan, vaan dataa käsittelevät skriptit tai funktiot sijaitsevat muualla. Esimerkiksi fysiikan mallinnusta koskeva data on kaikki muistissa tallennettuna samaan paikkaan ja datalla on vain viittaus, minkä objektin käyttäytymistä mikäkin data koskee.

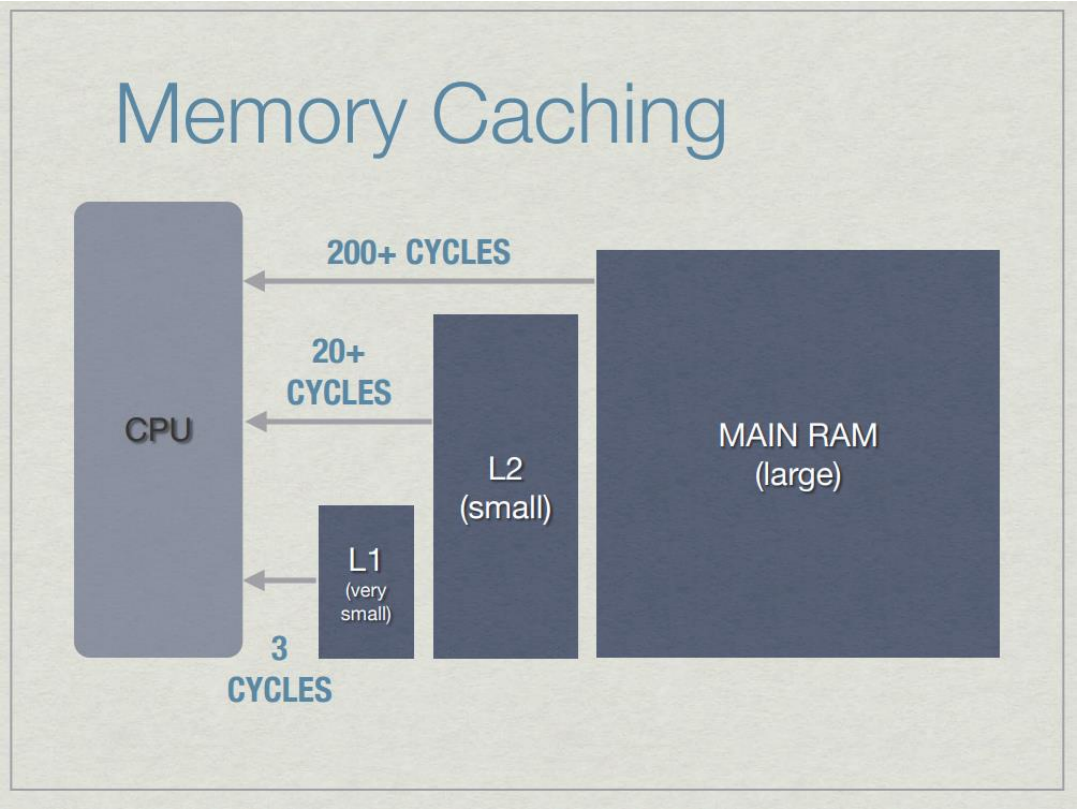


Kuva 2. Yksinkertaistettu esimerkki DOD-mallista [26].

Muistin järjestämisellä ja sen käytön parantamisella on suuria vaikutuksia pelin tekemiseen. Muistin käytön parannukset näkyvät selkeästi pelin sulavuudessa ja siinä, kuinka paljon peleissä voidaan käsitellä dataa. Mahdollisella datan määrällä on suorat vaikutukset aina pelin grafiikasta sen toimivuuteen. Peleissä vuosien aikana näkyvä kehitys on suoraan seurausta parantuneen muistin käsittelyn ja tietenkin peliä suorittavan järjestelmän tehokkuudesta.

Kuvan 3 luvut prosessorin (CPU) muistisykleille (engl. Memory Cycle) ovat vain suuntaa antavia nykyajan koneille. Ne kuitenkin kuvastavat hyvin muistin käsittelyn tärkeyttä. Kun prosessori joutuu hakemaan dataa yhä kauempaa ja isommasta muistista, muistisyklien määrä ei käytetty aika moninkertaistuu. Kaikkea dataa ei tietenkään voida säilyttää esimerkiksi L1- tai L2-välimuistissa, koska ne ovat sen verran pieniä. L1-välimuisti on kooltaan vain muutamien kymmenien kilotavujen kokoinen ja L2 puolestaan yhden megatavun luokkaa. RAM-muistin on huomattavasti näitä suurempi, keskimäärin 8-16 gigatavua

nykyajan tietokoneissa. RAM-muistiin mahtuu jo huomattavasti pelin tietoa, mutta se on monin kerroin L2-välimuistia hitaampi paikka hakea dataa. Kiintolevyltä tiedon hakeminen taas on RAM-muistia monin kerroin hitaampaa, mutta käytössä olevat kiintolevyt ovat vastaavasti nykyään jo monen teratavun kokoisia.[3.]



Kuva 3. Prosessorin (CPU) kuluttama aika sen hakiessa tietoa eri muisteista [7].

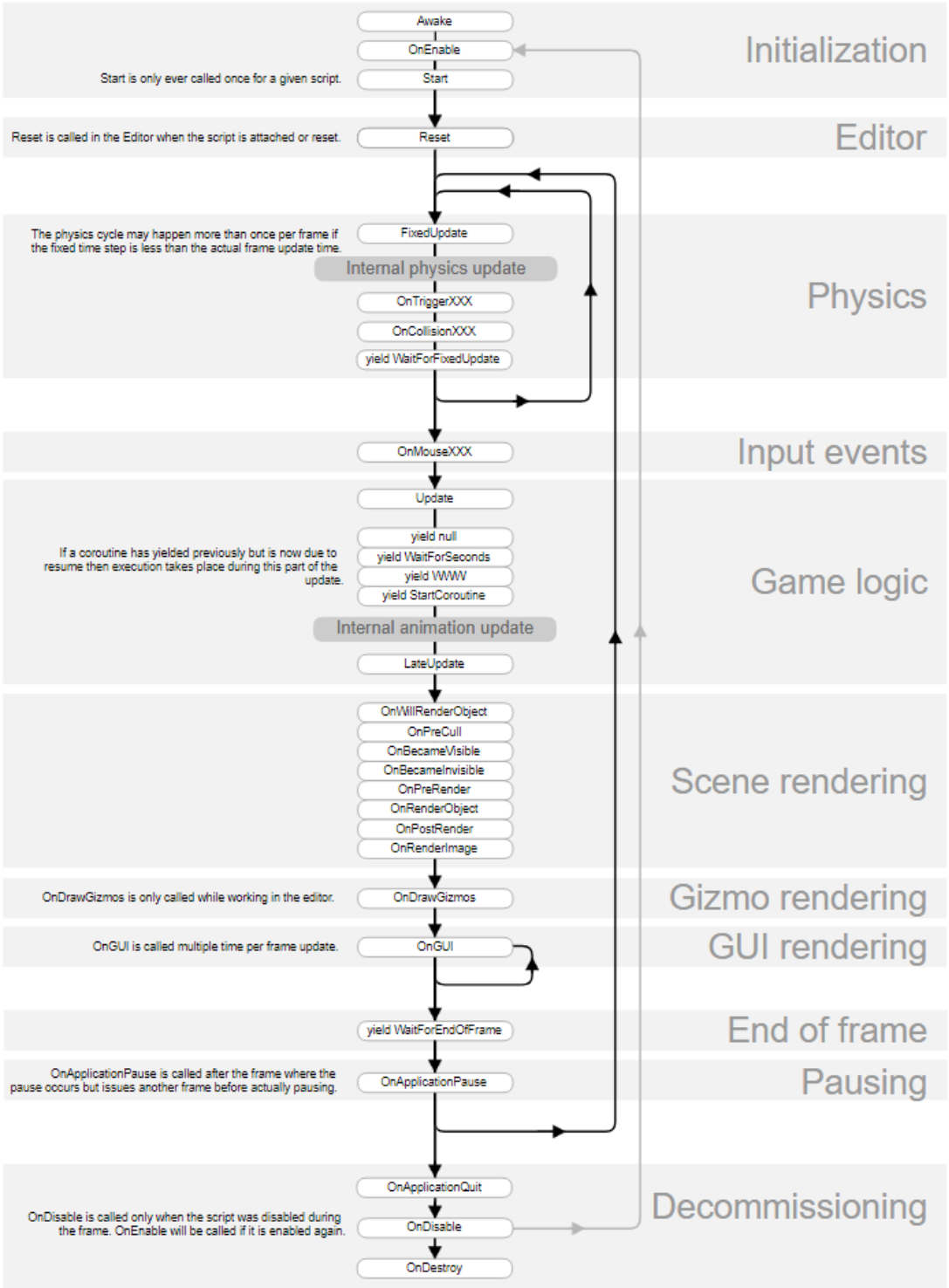
Toiminnassa olevista pelimoottoreista DOD-mallia käyttää ainakin esimerkiksi Frost-Byte-pelimoottori. Dice-peliyhtiö muokkasi FrostByte-pelimoottorinsa uudestaan nimenomaan DOD-mallia käyttäväksi tehdessään Battlefield 3 -nimistä peliä. FrostByte oli sitä ennen muiden pelimottoreiden tapaan enemmän OOP-mallinen. [8.] Daniel Collin toimii pelinkehittäjän Dice-peliyhtiössä, ja hän piti puheen DOD-mallin muuttamisen syistä ja eduista. Alkuperäiset syyt DOD-malliin muuttamiselle olivat pyrkimys parempaan skaalautuvuuteen ja tasojen parempi muistin hallinta. Skaalautuvuudella haluttiin suurentaa mahdollisten pelissä olevien objektien määrää, ilman että pelattavuus kärsii. Tasojen hallintaa haluttiin parantaa niin, että tarvittavat tasot ladataan muistiin, kun niitä tarvitaan. Aikaisemmassa mallissa kaikki pelin tasot olivat ladattuna muistiin, ja se rajoitti paljon tasojen suunnittelua. DOD-malliin toi mukanaan parannuksia juuri haluttuihin asioihin.

Dalien totesi, että Dicen peleissä objektien maksimimäärä on yleensä noin 15 000 kappaleen luokkaa. Näillä kappalemäärillä alustavien testien mukaan uusi DOD-malli toimi kolme kertaa paremmin kuin aiemmin käytössä ollut malli. [23.]

Unity on ryhtynyt omassa pelinkehitysympäristössään panostamaan enemmän DOD-malliin tai ainakin tekemään sen käytön helpommaksi. Unity palkkasi vuoden 2017 marraskuussa juuri tähän tarkoitukseen kaksi pitkään pelialalla ollutta henkilöä, jotka ovat urallaan keskittyneet suurissa peliprojekteissa käytettyihin pelimoottoreihin ja niiden tehokkuuteen. Palkatut henkilöt ovat Mike Acton ja Andreas Fredriksson, ja he tulisivat työskentelemään jo ennestään käynnissä olevassa projektissa tuoda DOD-malli Unityyn. [9.]

3 Tiedonkäsittely Unity-pelinkehitysympäristössä

Unity-editorissa on kaksi vaihetta, jotka ovat muokkausvaihe (engl. Edit Mode) ja pelivaihe (engl. Play Mode). Muokkausvaiheessa määritellään etukäteen, kuinka koko pelin pitäisi käyttäytyä sen eri vaiheissa. Kun Editori muutetaan pelivaiheeseen, Unity ajaa samaa luuppia yhä uudestaan (kuva 4). Kaikki muokkausvaiheessa tehdyt muutokset ja skriptit kertovat, mitä luupin aikana tulee tapahtua. Unityn luupit ovat kuin kuvakehykset (engl. Frames) elokuvissa. Kuvat tai editorin tapauksessa luupit kiertävät niin monta kertaa sekunnissa, että kaikki liike näyttää yhtenäiseltä.



Kuva 4. Unityssä yhden luupin aikana tapahtuvat toiminnot ja niiden käsittelyjärjestys [10].

Vaikka pelivaiheen aikana voidaan luoda peliin asioita, tieto luotavista asioista ja luontitavasta pitää kuitenkin olla etukäteen muokkausvaiheessa tallennettu tavalla tai toisella. Yleisin tapa tallentaa tieto on kirjoittaa se skriptiin, ja käytännössä kaikki Unityn skriptit

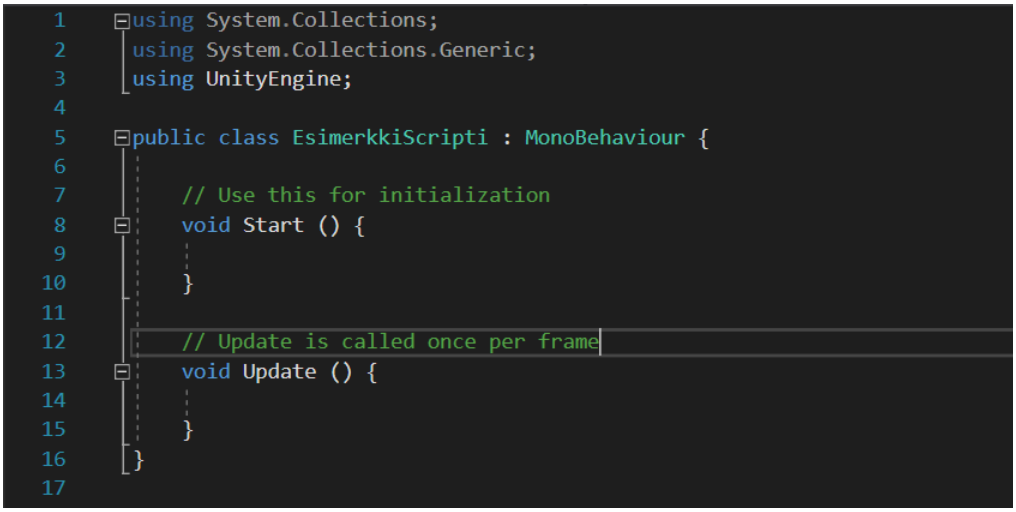
perivät ominaisuutensa MonoBehaviourista. Skriptit saavat näin MonoBehaviourin ominaisuudet, mutta myös sen rajoitteet. MonoBehaviour perii ominaisuutensa behaviour-luokasta eli Unityn käyttäytymistä käsittelevästä luokasta. MonoBehaviour ScriptableObject taas perii ominaisuudet objekti-luokasta. Tästä on monia hyötyjä MonoBehaviouriin verrattuna.

Unityssä skriptien kirjoittaminen on mahdollista C#-, JavaScript- ja Boo-ohjelmointikielillä, mutta kaiken taustalla Unity suoritetaan C++-ohjelmointikielellä. Tämän seurauksena kaikilla Unityn skripteillä on myös C++-puoli, vaikka se kirjoitettaisiinkin esimerkiksi C#-ohjelmointikielellä. Unityn skriptien kielistä ehdottomasti yleisimmät ovat C# ja JavaScript. Kun Editorissa siirrytään muokkausvaiheen ja pelivaiheen välillä tai esimerkiksi projektin tallennusvaiheessa, kaikki mahdollinen pelinäkymässä oleva tieto tallennetaan C++-puolelle ja loput poistetaan. Tämän jälkeen tiedot tuodaan takaisin C++-puolelta. Asioiden muokkaaminen on pelivaiheen aikana mahdollista, mutta palattaessa muokkausvaiheeseen tiedot palautuvat ennen pelivaiheen aloittamista olevaan tilanteeseen. ScriptableObjectit eivät sijaitse pelinäkymässä vaan editorin puolella, joten pelivaiheen aikana tehdyt muokkaukset säilyvät myös muokkausvaiheeseen palattaessa. [11.]

3.1 MonoBehaviour

Unityssä skripteillä muokataan pelin käyttäytymistä. Pelivaiheessa Unity käy läpi kaikki MonoBehaviour-luokan perivät skriptit ja toteuttaa siellä olevat käskyt riippuen skriptissä olevista funktioista. Käskyt voivat olla ajan, pelaajan toimintojen tai jonkin toisen pelissä olevan toiminnon ohjaamia.

Jokainen skripti, joka Unityssä luodaan, tulee automaattisesti perimään ominaisuutensa MonoBehaviour-luokasta (kuva 5). Unityssä voi kyllä periytymisen poistaa ja luoda oman luokan, mutta samalla menettää MonoBehaviour-luokan ominaisuudet. Uuden skriptin luominen onnistuu monella tapaa. Sellaisen luominen on mahdollista esimerkiksi Unityn projekti-nimisestä ikkunasta hiiren oikeaa painamalla ja avautuvasta luo-valikosta (engl. Create Menu) valitsemalla skriptin.



```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class EsimerkkiScripti : MonoBehaviour {

    // Use this for initialization
    void Start () {

    }

    // Update is called once per frame
    void Update () {

    }
}
```

Kuva 5. Unityssä C#-ohjelmointikielellä luotu uusi skripti. Kirjastot, funktiot ja perimiset tulevat automaattisesti.

MonoBehaviour-luokan alaisten skriptien on oltava liitettynä johonkin peliobjektiin (engl. Game Object). Pelit monesti sisältävät useita samankaltaisia objekteja, kuten esineitä, asioita ja olentoja, jotka toimivat samojen sääntöjen mukaisesti. Näillä yhtenäisillä objekteilla on monesti myös toisiaan vastaavat tiedot. Tiedot ovat yleisesti kirjoitettuna MonoBehaviour-luokan alaisiin skripteihin arvoina. Koska monen objektit käyttäytyvät samalla lailla ja omaavat samat tiedot, tuntuisi turhalta luoda jokainen objekti erikseen. Vaikka luominen onkin helppoa Unityssä esimerkiksi kopioimalla, ongelmia tulee viimeistään siinä vaiheessa kun kaikkia vastaavanlaisia objekteja pitäisi muokata.

Samankaltaisten objektien muokkaamiseksi ja ylläpitämiseksi on Unityssä mahdollista luoda valmiiksi malli-objekti nimeltä prefab. Prefab-objekti toimii alkuperäisenä kappaleena kaikille siitä kopioiduille objekteille. Prefab-objektia muokkaamalla pystyy helposti muuttamaan kaikkia vastaavia objekteja. Alkuperäinen prefab-objekti on pelinäkymän ulkopuolella editorin puolella oleva objekti. Pelinäkymässä olevia prefab-objektin kopiota voi myös muokata ja yhdellä näppäimellä päivittää muutoksen alkuperäiseen prefab-objektiin sekä sitä kautta myös muille kopioiduille objekteille. Mikäli tekee muutoksen jota ei halua kaikille muille objekteille, se on mahdollista, mutta se irrottaa objektin alkuperäisestä prefab-objektista ja siitä kopioiduista objekteista irralliseksi.

MonoBehaviour-luokan mukana skripti perii monia valmiita funktiota ja callback-kutsuja. Callback- kutsuilla tarkoitetaan funktioita, jotka suoritetaan silloin, kun jokin ennalta määritelty asia tapahtuu tai tarvittavat ehdot täyttyvät. Funktioita on yhteensä yli 60 erilaisiin

tarkoituksiin, kuten skriptin aktivoitumiseen, objektien törmäykseen ja aikaan liittyen. Tärkeimpiä ja yleisimmin käytössä olevia funktiota ovat

- Start()
- Update()
- FixedUpdate()
- LateUpdate()
- Awake()
- OnEnable()
- OnDisable()
- OnDestroy().

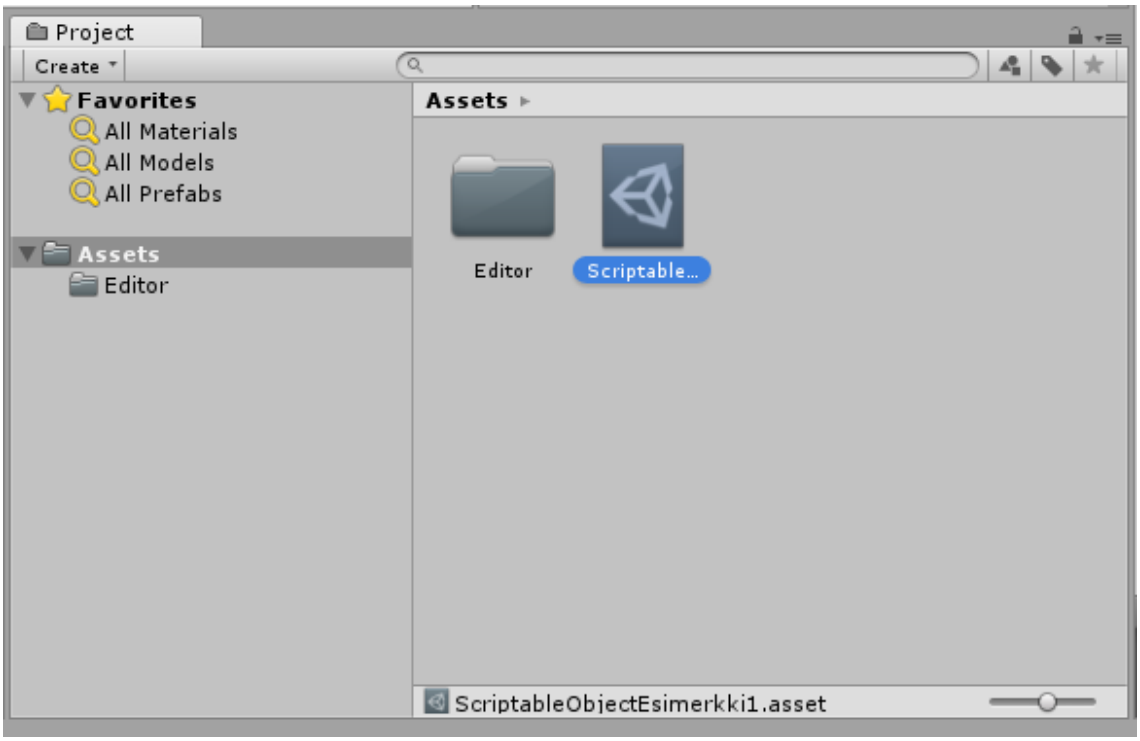
Start()- ja Update()-funktiot ovat niin yleisiä, että ne tulevat uuden skriptin luonnin yhteydessä jopa automaattisesti MonoBehaviour-luokalta periytymisen tapaan (kuva 5). Start()-funktiota kutsutaan kerran skriptin aktivoituessa. Update()-funktiota vastaavasti kutsutaan kerran joka luupissa, mikäli skripti on aktiivinen.

Useimmissa Unity-projekteissa skriptit sisältävät kaikki pelin käyttäytymiset, toiminnot ja tiedot. Pelin toiminnallisuuden kannalta on tärkeää, että skriptit voivat kutsua toisiaan MonoBehaviour-luokan alaisissa Callback-funktiossa tai muissa omissa funktioissaan. Kun skriptit kutsivat toinen toisiaan, kutsuista tulee kuitenkin niin monimutkaisia, että välillä on vaikea seurata, mitä kaikkea meneillään olevan luupin aikana tapahtuu. Ongelmatilanteissa ratkaisun selvittäminen monesti vaikeutuu juuri tästä syystä, vaikka Unity pystyykin aika hyvin ilmoittamaan virheen sijainnin. Toisinaan taas kyse ei ole suoranaisesti virheestä, vaan halutaan vain selvittää, miksi jotain tapahtuu kyseisellä ajanhetkellä.

3.2 ScriptableObject

Toisin kuin MonoBehaviour, ScriptableObject perii ominaisuutensa objekti-yläluokasta. ScriptableObject on tarkoitettu nimenomaan tietojen käsittelyyn ja tallentamiseen. ScriptableObject on käytännössä sama kuin MonoBehaviour, mutta sitä ei tarvitse liittää mihinkään objektiin, vaan se voi elää pelkässä muistissa. C++-puolella MonoBehaviour-luokan skripti ja ScriptableObject eivät käytännössä eroa toisistaan, sillä niiden esitysmuoto on täsmälleen sama [11]. ScriptableObjectia ei siis voi liittää peliobjektiin tai pre-

fab-tiedostoon, mutta sen voi lisätä asset-tiedostoon. Asset-tiedostot (kuva 6) ovat Unityn tiedostomuoto esimerkiksi tiedon ja kokoonpanoasetusten säilyttämiseksi. Useampia ScriptableObjecteja voi lisätä jopa samaan asset-tiedostoon.



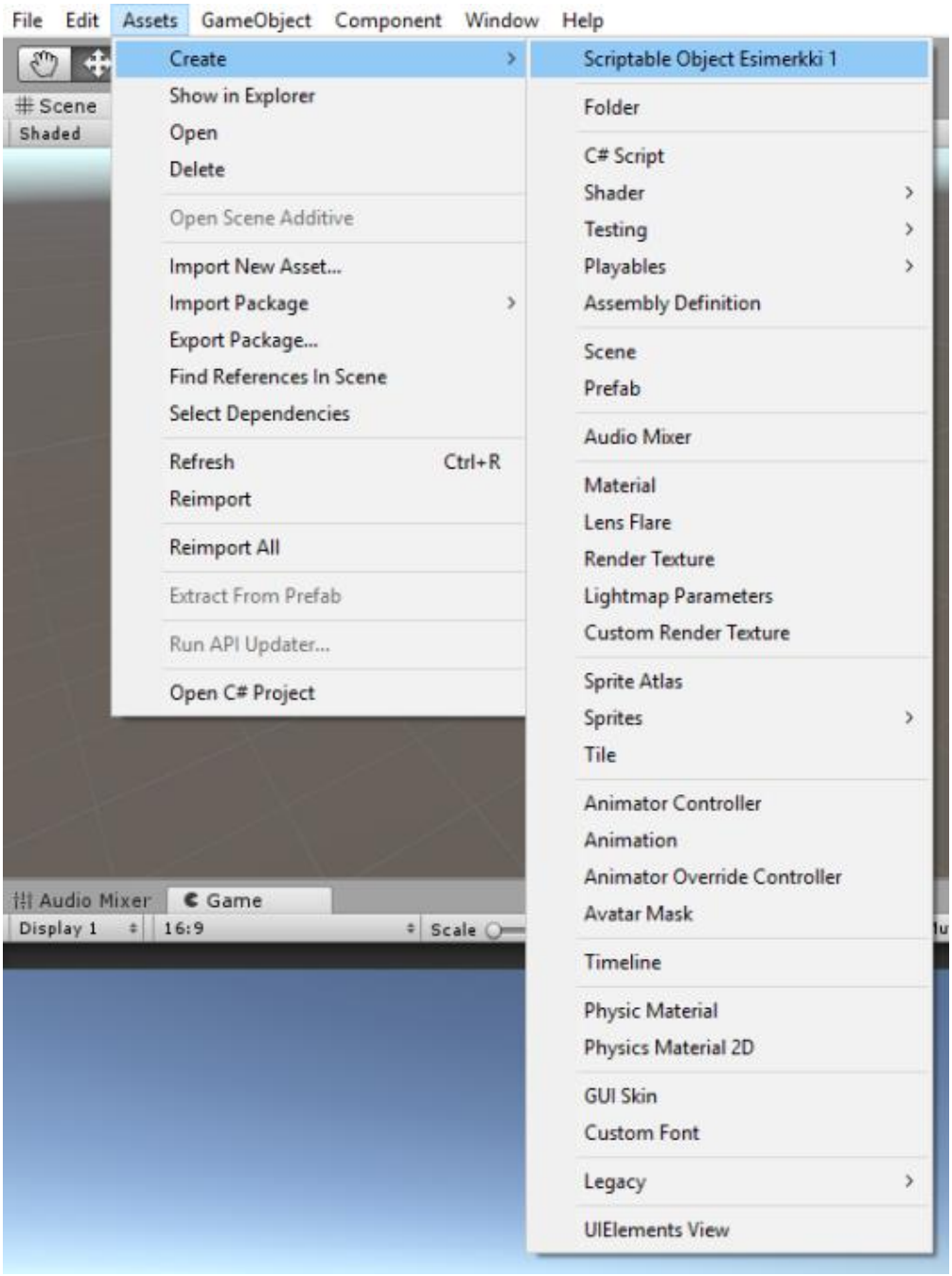
Kuva 6. Asset-tiedosto Unityn projektinäkymässä.

ScriptableObjectin luodaan eri tavalla kuin tavallinen skripti. ScriptableObjectin luomiseen on kaksi tapaa. Toinen niistä on ScriptableObject-luokan perivän skriptin kirjoittaminen ja ennen luokan määritystä koodiin tulee lisätä ”[CreateAssetMenu]”-ominaisuus (kuva 7).

```
1  using UnityEditor;
2  using UnityEngine;
3
4   [CreateAssetMenu]
5  public class ScriptableObjectEsimerkki1 : ScriptableObject
6   {
7   }
```

Kuva 7. C#-ohjelmointikielellä määritelty ScriptableObject.

”[CreateAssetMenu]”-ominaisuuden lisäyksen jälkeen uuden, juuri määritellyn ScriptableObjectin tekeminen on mahdollista Unityn omasta luo-valikosta (kuva 8). Tällä tavalla luotu ScriptableObject lisätään automaattisesti omaan asset-tiedostoon.



Kuva 8. Oma ScriptableObject on Unityn luo-valikossa.

Toinen tapa luoda ScriptableObject on tehdä se suoraan koodissa komennolla ”CreateInstance”. Tällä tavalla tehty ScriptableObject luodaan vain muistiin, ellei sitä erikseen lisätä haluttuun asset-tiedostoon. Kuvassa 10 on esimerkki ScriptableObjectin

luomisesta tällä tavalla. Rivillä 8 ensin luodaan ScriptableObject muistiin. Rivillä 9 puolestaan tehdään uusi asset-tiedosto ja lisätään rivillä 8 luotu ScriptableObjecti kyseiseen asset-tiedostoon.

```
1  using UnityEngine;
2  using UnityEditor;
3
4  public class ScriptableObjectEsimerkki2
5  {
6      public void CreateScriptableObject()
7      {
8          ScriptableObjectEsimerkki1 asset = ScriptableObject.CreateInstance<ScriptableObjectEsimerkki1>();
9          AssetDatabase.CreateAsset(asset, "Assets/ScriptableObjectEsimerkki1.asset");
10     }
11 }
```

Kuva 9. Vaihtoehtoinen esimerkki ScriptableObjectin luomisesta.

ScriptableObjectissa ainoat käytössä olevat Callback-funktiot ovat OnEnable(), OnDisable() ja OnDestroy(). OnEnable()-funktiota kutsutaan, kun objekti luodaan tai ladataan tai kun skriptien udelleen lataus on valmis. OnDisable()-funktiota kutsutaan, kun objektin poistamista aloitellaan tai skriptien uudelleen lataus on alkamassa. OnDestroy()-funktiota kutsutaan, kun objektia ollaan parhaillaan poistamassa. Funktioiden vähyys tuo mukanaan rajoitteita, mutta etuna on selkeys ongelmanselvitystilanteissa tai toiminnan tarkastelun vaiheessa.

ScriptableObjectit sijaitsevat objektien ja prefab-tiedostojen sijaan muistissa tai editorissa olevissa asset-tiedostoissa, joten pelivaiheesta muokkausvaiheeseen siirtymisessä tapahtuva tietojen palautus ei vaikuta niihin. Tämä mahdollistaa ScriptableObjectien tietojen muokkaamisen pelivaiheen ollessa käynnissä ja tuo selvän edun MonoBehaviour-skriptien käyttöön verrattuna. Peliä tehdessä on usein myös tilanteita, joissa arvojen palautuminen on hyödyllistä tai jopa välttämätöntä. Nämä tapaukset koskevat yleisesti pelin aikana muuttuvia arvoja, joita harvemmin säilytetään ScriptableObjecteissa.

Unityssä projektit sisältävät vähintään yhden tai useamman scene-tiedoston. Scene-tiedostot ovat niin sanottuja ”kohtauksia”. Yleisesti yksi scene-tiedosto on pelin yksittäinen taso, kohtaus tai kenttä. Peli on kyllä mahdollista toteuttaa myös niin, että kaikki pelin tasot ja kohtaukset ovat vaikka yhdessä scene-tiedostossa. Tämä ei tosin ole yleisesti suositeltavaa. Peleissä monesti samoja asioita ja objekteja sijaitsee monessa eri tasossa ja kohtauksessa. Tämän vuoksi objekteja joudutaan usein kopioimaan eri scene-tiedostojen välillä. ScriptableObjecteja käytettäessä kaikki niissä oleva tieto on käytettävissä mistä tahansa scenestä, koska ne sijaitsevat yksittäisen scene-tiedoston ulkopuolella.

Tämä mahdollistaa myös sen, että ScriptableObjecteja käyttävät asiat on helpompi siirtää, jopa eri projektien välillä.

ScriptableObjectit ja sen tiedot on mahdollista serialisoida (engl. Serialization). Serialisoiminen tarkoittaa esimerkiksi tietorakenteiden tai objektien muuttamista sellaiseen muotoon, että Unity voi tallentaa ne ja palauttaa ne myöhemmin taas käsiteltävään muotoon. Unityssä serialisointi tapahtuu muutamissa vaiheissa. Näistä vaiheista yksi on esimerkiksi järjestelmän uudelleenlatausvaihe (engl. assembly reload), joka tapahtuu, kun skriptit ladataan uudestaan tai editorin tila vaihdetaan pelivaiheen ja muokkausvaiheen välillä. Toinen vaihe, jossa serialisointi tapahtuu, on kun projekti tallennetaan tai ladataan. ScriptableObjectien serialisoimisesta on paljon hyötyä, mutta sen käyttö vaatii hieman tietotaitoa tai muuten kaikki tehty työ ja kerätty data saattaa kadota Unityn uudelleen käynnistyksen aikana. ScriptableObjectien käytössä on siis tarkkaan huomioitavia asioita, mutta niistä kyllä selviää ScriptableObjectiin ja serialisointiin perehtymällä. [18; 19; 20.]

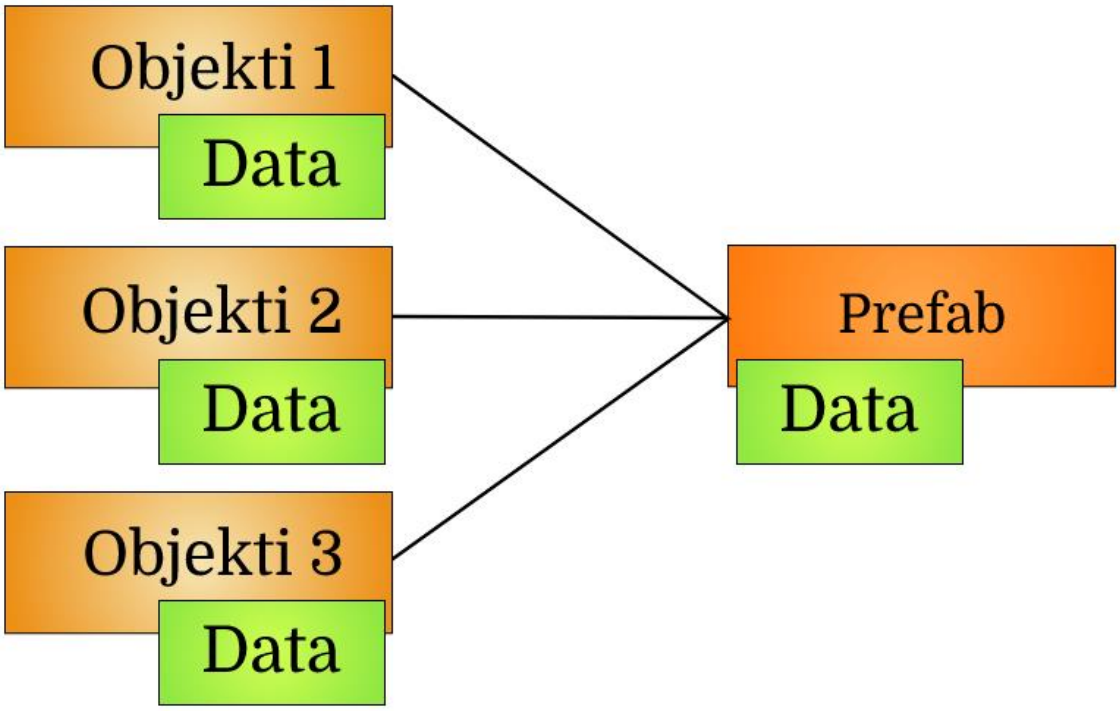
Unityssä on mahdollista luoda enum-skriptejä (engl. Enumerations). Enum-skripteillä on tarkoitus luoda joukko samantyyppisiä vakioita, joita voidaan vertailla keskenään. ScriptableObjecteilla voidaan tehdä sama editorin puolella, esimerkiksi vertailemalla erinimisiä tyhjiä ScriptableObjecteja keskenään. ScriptableObjectien vertailemisessa on myös se hyvä puoli, että vertailu voidaan suorittaa, vaikka vastaavan nimistä ScriptableObjectia ei listalta löytyisikään. Tämä ei toimi yhtä kätevästi enum-skriptien kohdalla, koska niitä käytettäessä joudutaan itse määrittämään, mitä tapahtuu jos vastaavaa vakiota ei löydykään.

ScriptableObjecteja hyödynnetään parhaiten, kun niissä tallennetaan tietoa, joka on kaikille sitä käyttäville sama. Sellaista tietoa on peleissä paljon, ja se on MonoBehaviour-skriptien tapauksessa monesti samassa paikassa, esimerkiksi objekteille yksilöllisten tietojen kanssa. Esimerkkinä tästä ovat pelissä olevat viholliset. Täysin samanlaisia vihollisia pelissä voi olla lukuisia ja niiden maksimielämäpisteet (engl. Hit Points) ovat yleisesti samat. Kun maksimielämäpisteet ovat kaikille samat, ne on turha määrittää jokaiselle erikseen, kun kaikki voi tarkista sen yhdestä ja samasta paikasta. Jokaisella näistä vihollisista on hyvä olla maksimielämäpisteiden lisäksi myös tämänhetkiset elämäpisteet. Nämä viholliselle yksilölliset tämänhetkiset elämäpisteet olisi toisaalta parempi säilyttää vihollisella itsellään tietona. Parhaat hyödyt ScriptableObjecteista saadaan, kun niitä

käytetään yhdessä MonoBehaviour-skriptien ja prefab-tiedostojen kanssa jakamalla tiedot käyttötarkoituksen mukaan.

3.3 Esimerkkitapaus muistin käytöstä

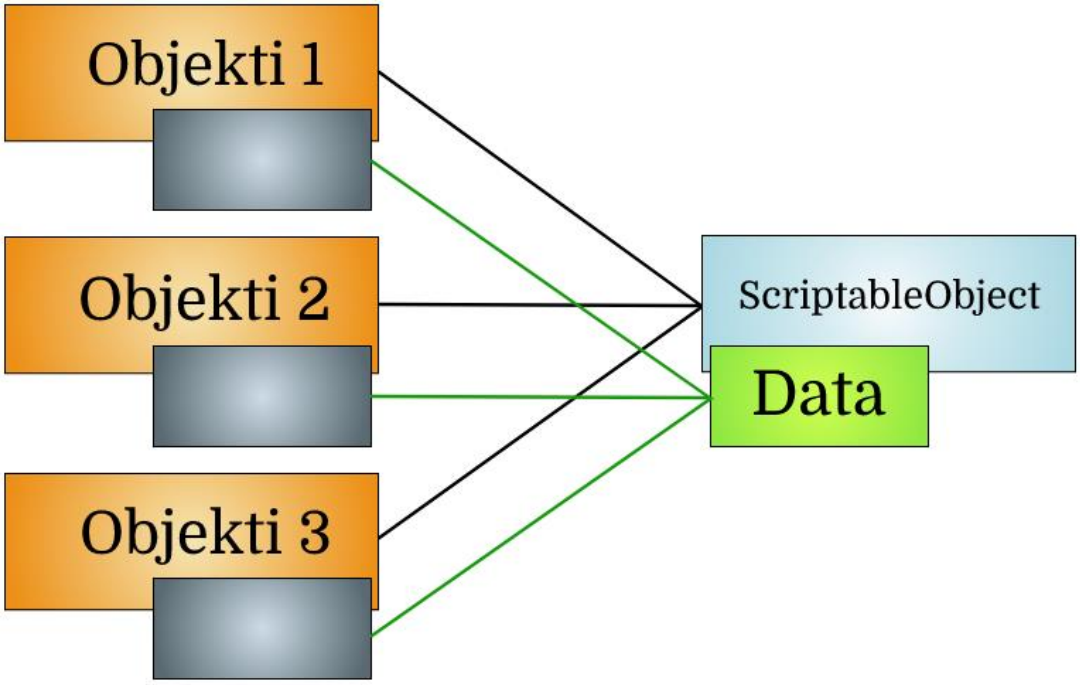
ScriptableObjectien tuoma hyöty tiedon eli datan talletuksessa käy helpoiten selville esimerkkitapauksen avulla. Kuvitellaan, että pelissä on luotu prefab-tiedosto, jolla on MonoBehaviour-skriptissä tieto miljoonan numerosarjan taulukosta. Taulukko vie muistista tilaa esimerkiksi neljän megatavun verran. Mikäli prefab-tiedostosta kopioidaan käsin tai koodissa kolme kopio-objektia peliin, samalla numerosarjataulukko kopioidaan myös jokaiselle objektille (kuva 10). Pelistä samoilla tiedoilla oleva taulukko vie näin neljä kertaa neljä megatavua tilaa, eli yhteensä 16 megatavua. [14.]



Kuva 10. Esimerkki prefab-tiedostosta kopioiduista objekteista.

Tarkastellaan vastaavaa tapausta ScriptableObjectia hyväksikäyttävällä mallilla. Nyt prefab-tiedoston ja MonoBehaviour-skriptin sijaan data on ScriptableObjectilla, josta luodaan kolme kopio-objektia (kuva 10). ScriptableObject omistaa oman datansa, ja objekteilla on kopioidun datan sijaan vain viite ScriptableObjectissa olevaan alkuperäiseen dataan. Viittausten käyttämä muistin määrä on niin pieni, että sitä ei tarvitse edes ottaa

laskuissa huomioon. Näin ollen ScriptableObjecteilla toteutettavassa esimerkissä muistia käytetään noin neljä megatavua. Tämä on huomattavasti vähemmän kuin prefab-tiedosto ja MonoBehaviour-skriptivaihtoehto. [14.]



Kuva 11. Esimerkki ScriptableObject-tiedostosta kopioiduista objekteista.

Esimerkki tapauksessa käytettiin vain kolme kopio-objektia, mutta monissa pelissä kopio-objekteja voi olla satoja, jopa tuhansia. ScriptableObjecteista on näin huomattavasti hyötyä suuria datamääriä käsitellessä. ScriptableObjectin esimerkkitapauksessa olisi pystytty käyttämään myös yhtä prefab-tiedostoa josta ne luodaan, mutta prefab-tiedostolla pitää tässä tapauksessa olla viittaus ScriptableObjectista löytyvään dataan (kuva 11). Näin kopioiduille objekteille tulee vain viittaus alkuperäiseen dataan.

4 Tiedon ja objektien tallennusjärjestelmä

Insinöörityössä ohjelmoitiin tiedon tallennusjärjestelmä Unity-pelinkehitysympäristöön ScriptableObjecteja käyttämällä. Tallennusjärjestelmä tuli käytettäväksi projektin aikana toteutettavan editorilisäyksen avulla. Projektin tekeminen edellytti paljon perehtymistä ScriptableObjecteihin ja editorin lisäosien toimintaan. Kaiken projektissa käytetyn koodin tuli olla itse kirjoitettua, mutta oppaita ja ohjeistuksia apuna käyttäen.

Tämäntyyppisen tallennusjärjestelmän tärkeys on käynyt ilmi aiempien, laajempien peliprojektien aikana. Eräässä peliprojektissa tallennusjärjestelmän tärkeys tuli erityisesti esille peliprojektiin itsestään generoituvan metsän suunnittelun aikana. Kaiken metsän generoitumiseen liittyvän tiedon ja siellä olevien asioiden ja objektien täytyy olla tallessa tavalla tai toisella. ScriptableObjectit soveltuvat tämäntyyppisen tiedon tallentamiseen erinomaisesti.

Insinöörityönä tehdyn tiedon tallennusjärjestelmän olisi hyvä olla myös yleishyödyllinen tallennusjärjestelmä, joka sopi monenlaisiin projekteihin ja moneen käyttötarkoitukseen. Työstä olisi näin eniten hyötyä myös tulevaisuuden projekteissa, ja monet pelinkehittäjät voisivat käyttää vastaavaa järjestelmää omissa projekteissaan.

4.1 Tavoitteet

Insinöörityön päätavoite oli saada valmiiksi toimiva tallennusjärjestelmä ScriptableObjecteja käyttäen. Tallennusjärjestelmän pitäisi olla valmis käytettäväksi tulevissa projekteissa ja sen ei, muutaman skriptin lisäksi, pitäisi tarvita mitään muuta toimiakseen. Tallennusjärjestelmään tallennettavat tiedot pitää tosin lisätä erikseen. Tallennusjärjestelmästä tiedon hakeminen koodissa ja sen käyttö ei sisältynyt projektin piiriin. Tiedon haku ja käyttö eivät tosin vaadi paljoa muutoksia, jos käytössä on esimerkiksi yleisempi MonoBehaviour- ja prefab-malli.

Järjestelmän suunnittelu alkoi jo syksyllä 2017. Suunnittelun aikana projektin laajuus ja tarkempi tietojen tallennustapa vaihteli useaan otteeseen. Tarkemmat tiedot eri vaihtoehdoista ja miksi päädyttiin juuri kyseiseen ratkaisuun, esitellään tarkemmin luvussa 4.4 kyseisen ominaisuuden selityksen yhteydessä.

Alkuvaiheissa suunnitteilla oli tehdä järjestelmä, joka kattaisi kaikki MonoBehaviour-skripteissä mahdolliset muuttujat ja arvot. Työn laajuutta jouduttiin kuitenkin rajaamaan ajan ja käyttökokemuksen parantamiseksi. Tallennusjärjestelmän toimivuuden kannalta on tärkeää, että tarvittavat muuttujat ovat helposti valittavana. Kaikkien mahdollisten MonoBehaviour-skripteissä olevien muuttujamahdollisuuksien määrän vuoksi niiden valinta listalta tai muilla tavoilla kasvoi liian monimutkaiseksi.

4.2 Editorilaajennus

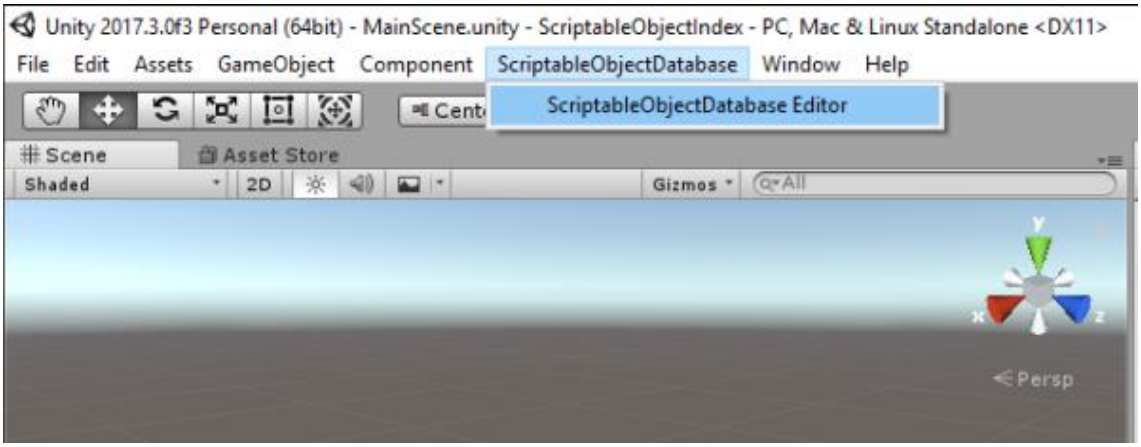
Unity antaa valmiiksi hyvät mahdollisuudet omien editorien tekemiselle. Itse tehdyt editorit voivat joko korvata Unityn omia editorin ikkunoita ja niiden tietoja tai vaihtoehtoisesti tuoda lisää ominaisuuksia Unityn omien editorin ikkunoiden lisäksi. Erityisesti ScriptableObjectien tapauksessa, mikäli kyseessä ei ole pelkästään muutama dataa sisältävä ScriptableObject, on suositeltavaa kirjoittaa editori niiden luontiin, käyttöön ja päivittämiseen. ScriptableObjecteissa olevat tiedot ovat nähtävissä Unityn oman editorin kautta, jos ScriptableObject on asset-tiedostoon lisätty. Tästä huolimatta yksinkertainenkin oman editorin lisäys tuo tiedon käsittelyyn huomattavasti helpotusta. Editorin muokkaukseen tarkoitetut skriptit pitää sijoittaa ”Editor”-nimisen kansion alle. Kansion sijainnilla ei ole väliä, ja kansioita voi olla useita, mutta nimenä pitää olla ”Editor”.

Koodissa luodaan Unityn omien editorin päävalikkojen lisäksi oma valikko, joka on nimeltään ”ScriptableObjectDatabase”, ja sen alle lisätään nappula nimeltä ”ScriptableObjectDatabase Editor” (kuva 12). OnEnable()-funktiossa ladataan itse tehty ScriptableObject polusta ”Assets/Database/DatabaseIndex.asset”, jos sellainen on jo olemassa.

```
34  [MenuItem("ScriptableObjectDatabase/ScriptableObjectDatabase Editor")]
35  static void Init()
36  {
37      GetWindow(typeof(ScriptableObjectDatabaseEditor));
38  }
39
40  void OnEnable()
41  {
42      _databaseIndex = AssetDatabase.LoadAssetAtPath("Assets/Database/DatabaseIndex.asset", typeof(DatabaseIndex)) as DatabaseIndex;
43  }
44
```

Kuva 12. Oman valikon ja editori-ikkunan luominen.

Kuvassa 13 olevaa ”ScriptableObjectDatabase Editor” -nappulaa painamalla aukaistaan itse määritelty ja ohjelmoitu Unity-editorin ikkuna. GetWindow()-kutsu avaa uuden ikkunan, jos ikkuna ei ole jo ennestään auki. Jos ikkuna on auki, kutsu vain valitsee sen aktiiviseksi. GetWindow() on EditorWindow-luokan alainen kutsu ja edellyttää sen aikaisempaa määritystä tai muuten kutsun pitää olla muodossa ”EditorWindow.GetWindow()”.



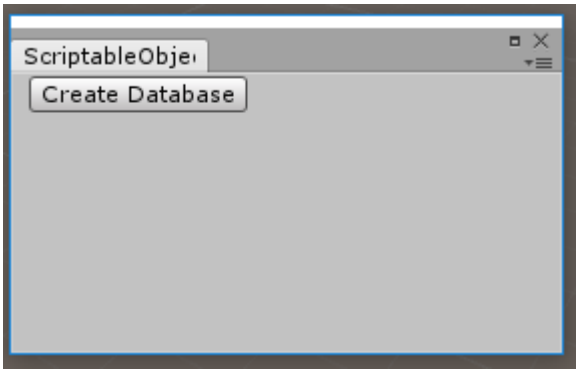
Kuva 13. Itse tehty valikko ja sen alainen nappula, jolla aukaistaan oma editori-ikkuna.

Kaikki editori-ikkunan sisällä tapahtuva kirjoitetaan OnGUI()-funktion alle. Aikoinaan Unityssä tämä funktio oli vielä yleisemmin käytössä. Sillä tehtiin muun muassa pelin sisäisiä valikoita, mukaan lukien päävalikot. Nykyään se on pelin sisäisien valikkojen tekemisessä jäänyt taka-alalle Unityn tämänhetkisen valikoiden luontiin tarkoitetun järjestelmän myötä.

4.3 SkriptableObjectien luonti

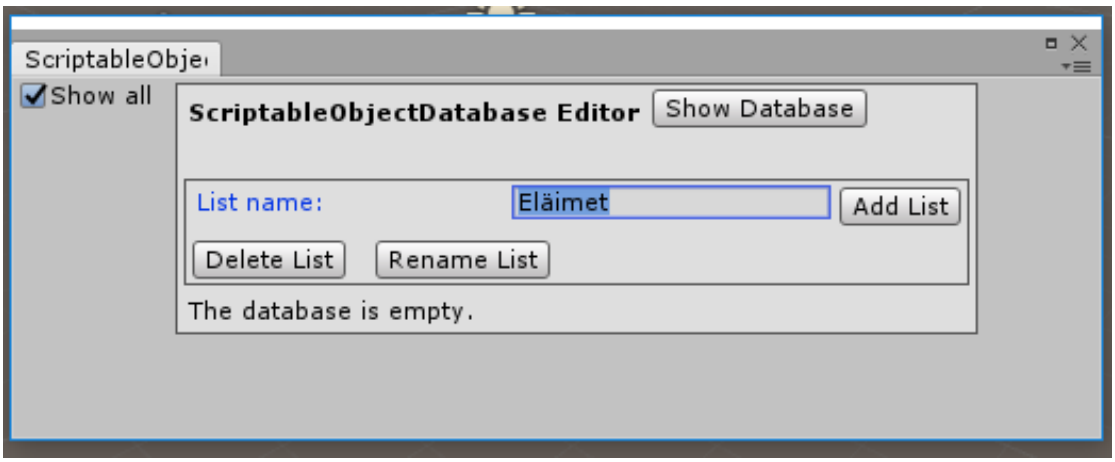
Insinöörityönä tehty tallennusjärjestelmä koostuu kahdentyyppisistä ScriptableObjecteista. Ylimpänä hierarkiassa on ”DatabaseIndex”-niminen ScriptableObject, joka sisältää tiedon kaikista tallennusjärjestelmän listoista. Listojen määrää ei ole rajoitettu. Tallennusjärjestelmässä tosin on vain yksi DatabaseIndex-tiedosto ja se on ennalta määritellyssä kansiossa ”Database”. DatabaseIndex-tyyppisen ScriptableObjectin sisältämät lista-tiedostot ovat kaikki omia ScriptableObjecteja omissa kansioissaan. Nämä kansiot sijaitsevat myös saman ”Database”-nimisen kansion alla. Kansioiden nimeksi tulee listalle annettu nimi. Lista-tyyppinen ScriptableObject sisältää itse tehdyn luokan ”item”-tyyppisiä olioita. Nämä voivat olla esimerkiksi esineitä, asioita, objekteja tai vaikka eläimiä. Listan item-olioilla on kaikilla samat arvot ja muuttujat, mutta niiden tiedot vaihtelevat. Jos siis listaan on määritelty kaikille merkkijono (engl. string) ja asetettu se tarkoittamaan nimeä, kaikilla item-olioilla on tämän jälkeen muuttuja ”nimi”, joka on merkkijonotyyppinen. Merkkijono voi sitten sisältää item-olion nimen, kuten esimerkiksi tuoli tai pöytä.

”ScriptableObjectDatabase Editor” -nappulaa (kuva 13) painamalla aukeavan ikkunaan tulevat näkyviin kaikki tallennusjärjestelmän tiedot, jos sellainen on jo luotu. Mikäli valmista DatabaseIndex-tyyppistä ScriptableObjectia ei löydy, aukeaa ikkuna, jossa on vain yksi nappula nimeltä ”Create Database” (kuva 14).



Kuva 14. Oman editorin ikkuna, kun valmista DatabaseIndex-tyyppistä ScriptableObjectia ei löydy.

Kuvan 14 ”Create Database” -nappulaa painamalla luodaan uusi DatabaseIndex-tyyppinen ScriptableObject, sekä ”DatabaseIndex”-niminen asset-tiedosto, joka lisätään ”Database”-nimiseen kansioon. Mikäli kansiota ei löydy, se luodaan samassa yhteydessä. Uusi DatabaseIndex-tyyppinen ScriptableObject lisätään ”DatabaseIndex”-nimiseen asset-tiedostoon. ScriptableObjecteja ja kansiota luotaessa tehdään tarpeellisia tarkistuksia kansion tai tiedostojen sijaintiin liittyen. Kaiken tämän jälkeen ikkunaan aukeavat tiedot uuden listan luomiseksi (kuva 15).



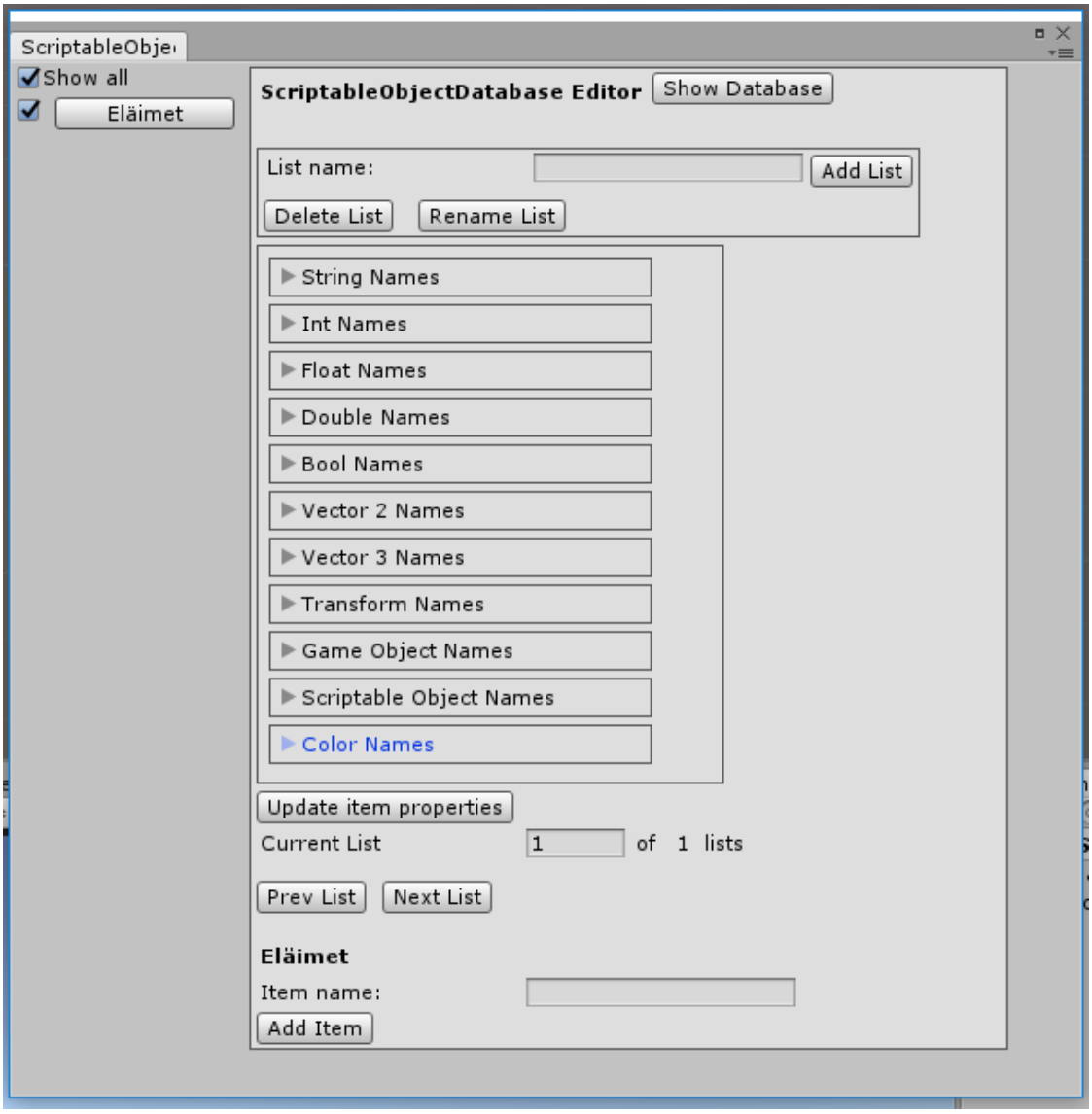
Kuva 15. Uuden lista luominen.

Kuvan 15 ”Add List” -nappulaa painamalla saadaan luotua tallennusjärjestelmään uusi lista, kunhan listan nimitieto-kenttä ei ole tyhjä tai samannimistä listaa ei ole jo olemassa. Listan luominen tekee sille uuden kansion ja asset-tiedoston. Samassa yhteydessä luotu lista-tyyppinen ScriptableObject lisätään sille tarkoitettuun asset-tiedostoon. Listan ScriptableObjectien luominen omiin assetti-tiedostoihin on selkeämpää, varsinkin listojen määrän kasvaessa. Listojen ScriptableObjectit olisi tosin pystytty myös lisäämään esimerkiksi samaan DatabaseIndex ScriptableObjectia varten luotuun asset-tiedostoon.

Listat luodaan yksitellen määrittelemällä niille nimi ja painamalla ”Add List” -nappulaa (kuva 15). Nimi on hyvä valita listan käyttötarkoituksen mukaan. Yleinen tapa jaotella pelin objektit listoihin olisi juuri luomalla omat listat esimerkiksi eläimille, esineille ja aseille. Kannattaa kuitenkin harkita tarkkaan, onko se paras mahdollinen ratkaisu, koska kuten luvussa 2 kävi ilmi, esimerkiksi pöytiä voi olla monenlaisia. Ei siis välttämättä ole järkevää laittaa kaikkia pöytiä samaan listaan. Hyödyllisempi listojen jaottelu voisi olla esimerkiksi erotella rikkoutuvat ja rikkoutumattomat objektit tai vaihtoehtoisesti pelaajan käsiteltävät ja ei käsiteltävät objektit. Tämäntyyppinen jako yhtenäistäisi paremmin samanlaisia tietoja käyttävät objektit saman listan alle.

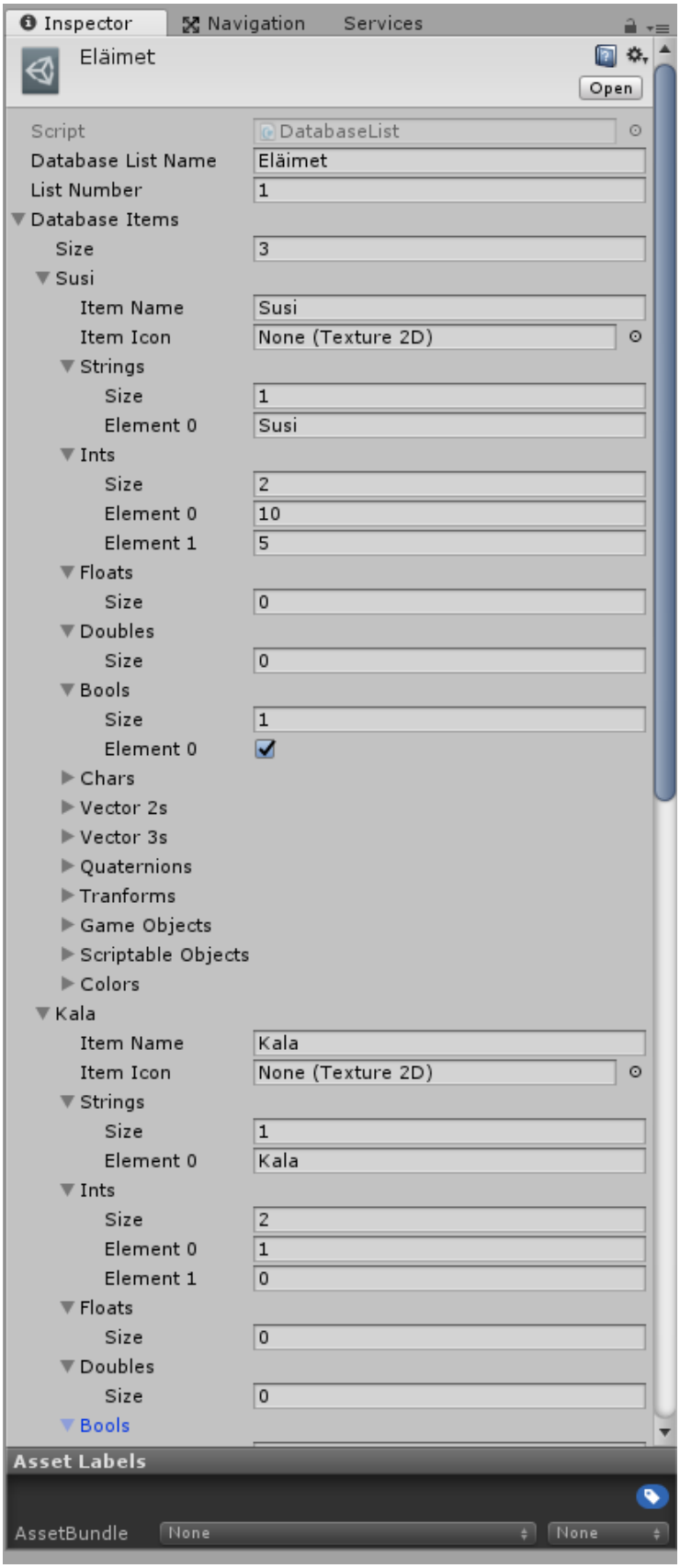
Kuten kuvassa 15 näkyy, listan luomisen jälkeen ikkunaan aukeavat mahdollisuudet muokata listaa, sekä sen määriteltyjä arvoja ja muuttujia. Samalla aukeavat myös mahdollisuudet luoda listalle uusia item-oliota. Listan item-olioille voidaan jo tässä vaiheessa valita halutut muuttujat tai ne voidaan päivittää vasta myöhemmin ”Update item properties” -nappulaa painamalla.

Kuvassa 16 listan muuttujat on kaikki pienennetty, mutta niiden edessä olevasta nuolesta aukeaa mahdollisuus lisätä käyttäjän haluama määrä muuttujia ja nimetä ne niiden tarkoituksen mukaan. Jokaisen lista-tyyppinen ScriptableObject sisältää kaikki listalle määritetyt muuttujat ja niiden nimet. Listojen välillä voi liikkua monella tavalla. Listan vaihtaminen onnistuu sen nimellä löytyvän nappulan painamisen lisäksi ”Current list” -kentästä listan numeron avulla. Listoja voi myös selata ”Prev List” -painikkeella taaksepäin ja ”Next List” -painikkeella eteenpäin.



Kuva 16. Listan sisältö ja muokkausvaihtoehdot. Mahdollisuus myös uuden item-olion luotiin.

Oman editorin tuoman selkeyden ja hyödyn huomaa helposti, kun listoja ja item-olioita aletaan lisätä. Unity editorin oma näkymä lista-tyyppisestä ScriptableObjectista on huomattavasti sotkuisempi eikä niin helposti käytettävissä kuin itse tehty editori-ikkuna. (kuva 17).

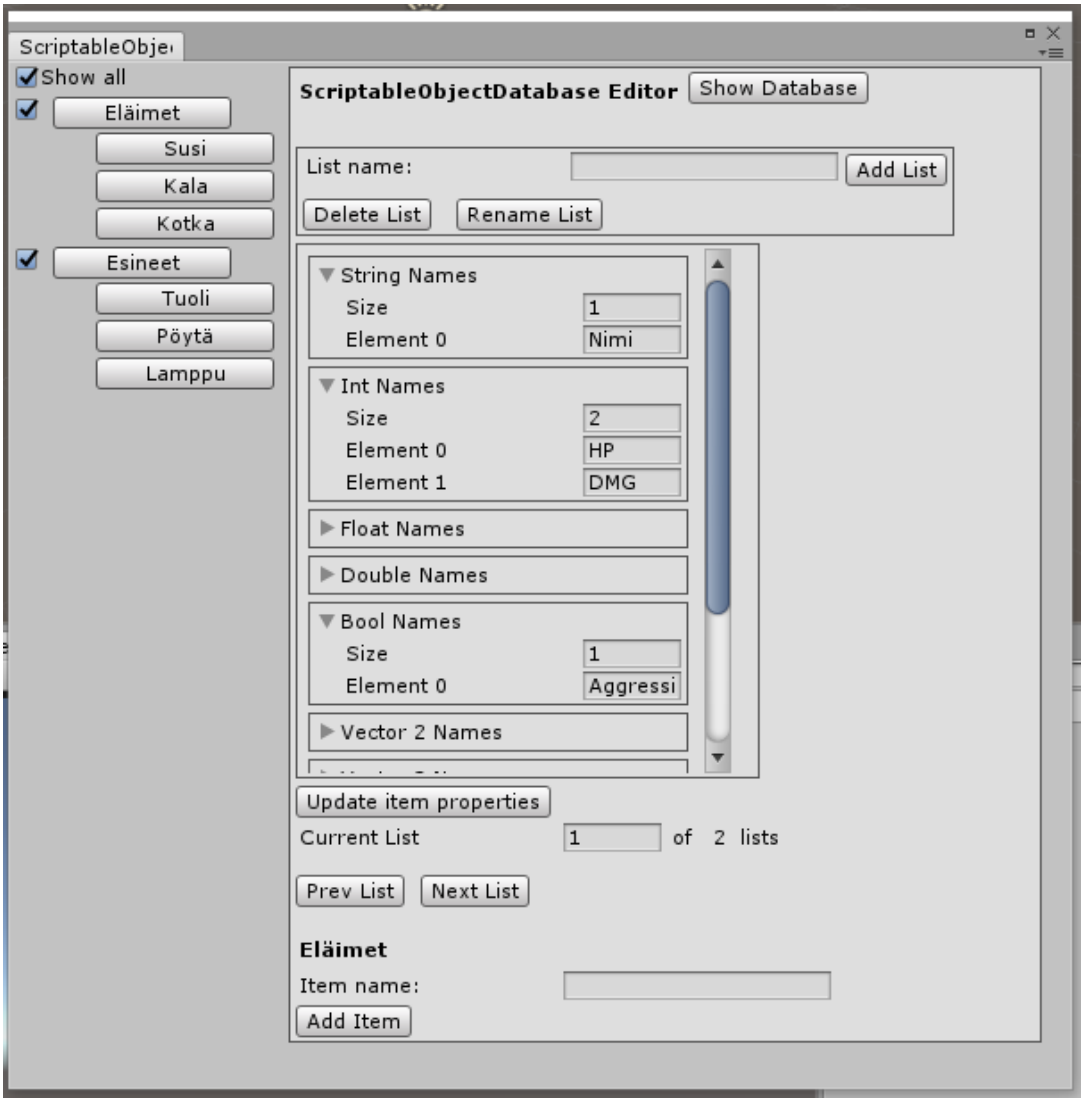


Kuva 17. Unityn oman editorin näkymä lista-tiedostolle.

4.4 Tallennusjärjestelmän toiminnallisuus

Tallennusjärjestelmä toimii niin, että listan tiedoista valitaan, kuinka monta erityyppistä muuttujaa jokaisella listassa olevalla item-oliolla on, ja annetaan muuttujille nimet. Listan item-olioiden muuttujien määrän, tyypin ja nimen voi päivittää koska tahansa ”Update item properties” -painikkeella, kun tiedot on muutettu halutuiksi. Listojen on tarkoitus toimia samanlaisten objektien ja asioiden listana, joten muuttujat ovat siitä syystä aina samat jokaiselle listan item-olioille.

Listan jotain item-olioita painamalla saa näkyviin juuri kyseisen item-olion tiedot ja pääsee muokkaamaan sen arvoja (kuva 18).

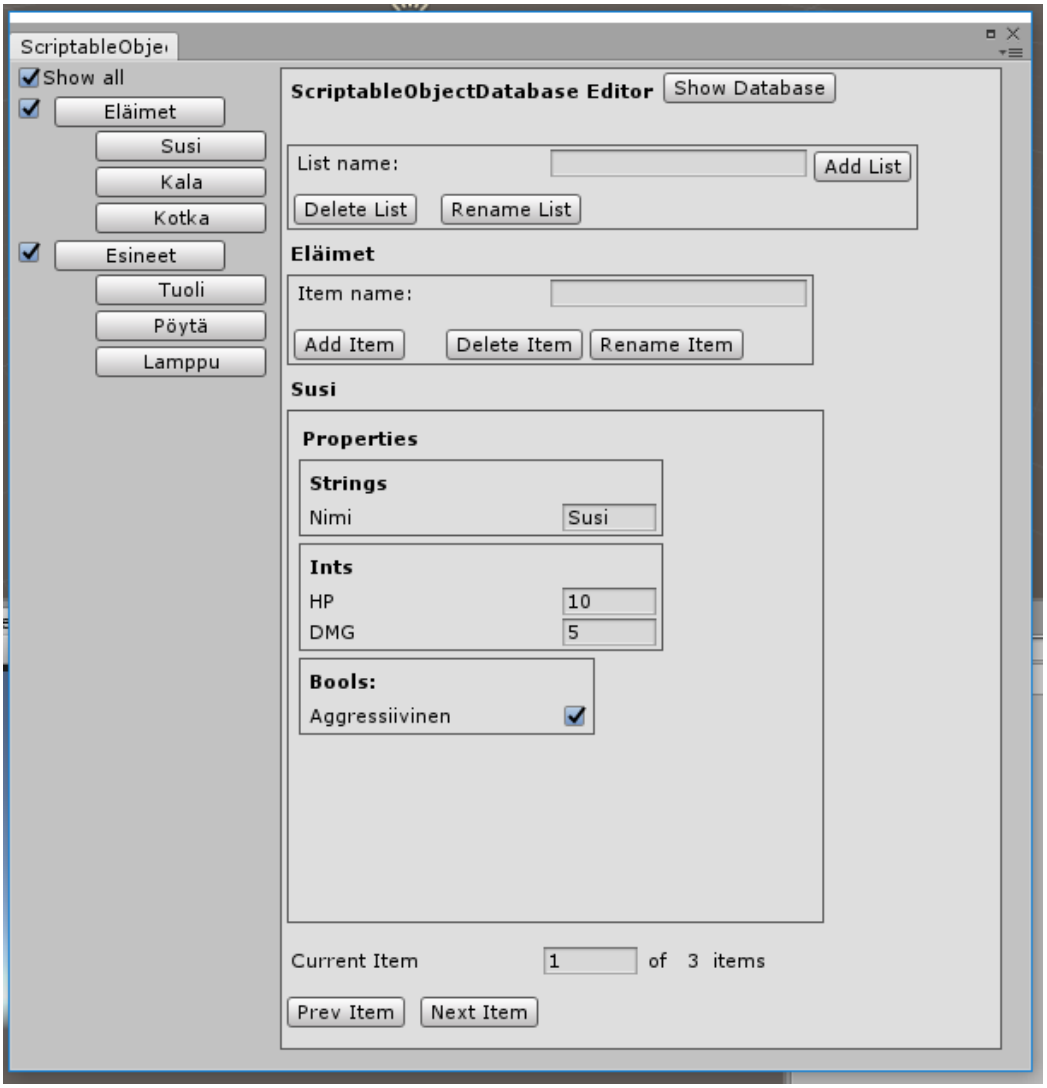


Kuva 18. Oman editori-ikkunan näkymä listan muuttujista.

Kuvassa 19 on valittuna ”Eläimet”-listalta ”Susi”-tyyppinen item-olio. ”Susi”-oliolle on annettu nimeksi ”Susi” ja määritelty HP- ja DMG-arvot. Näiden lisäksi ”Susi”-oliolta löytyy boolean-muuttuja, joka kuvaa tässä tapauksessa, onko eläin aggressiivinen vai ei. Kuvassa 18 on myös valmiiksi luotuna kaksi listaa, jotka ovat ”Eläimet” ja ”Esineet”. Listoja ja listoissa olevien item-olioiden määrää ei ole rajattu. Item-oliot voivat myös sisältää ScriptableObjecteja joten item-oliolla voi olla arvonaan toinen lista. Tämä mahdollistaa sen että tallennusjärjestelmällä voi luoda vaikka puumallisia (engl. Tree model) tiedon tallennusmuotoja. Tallennusjärjestelmässä käytössä olevat muuttujat ovat

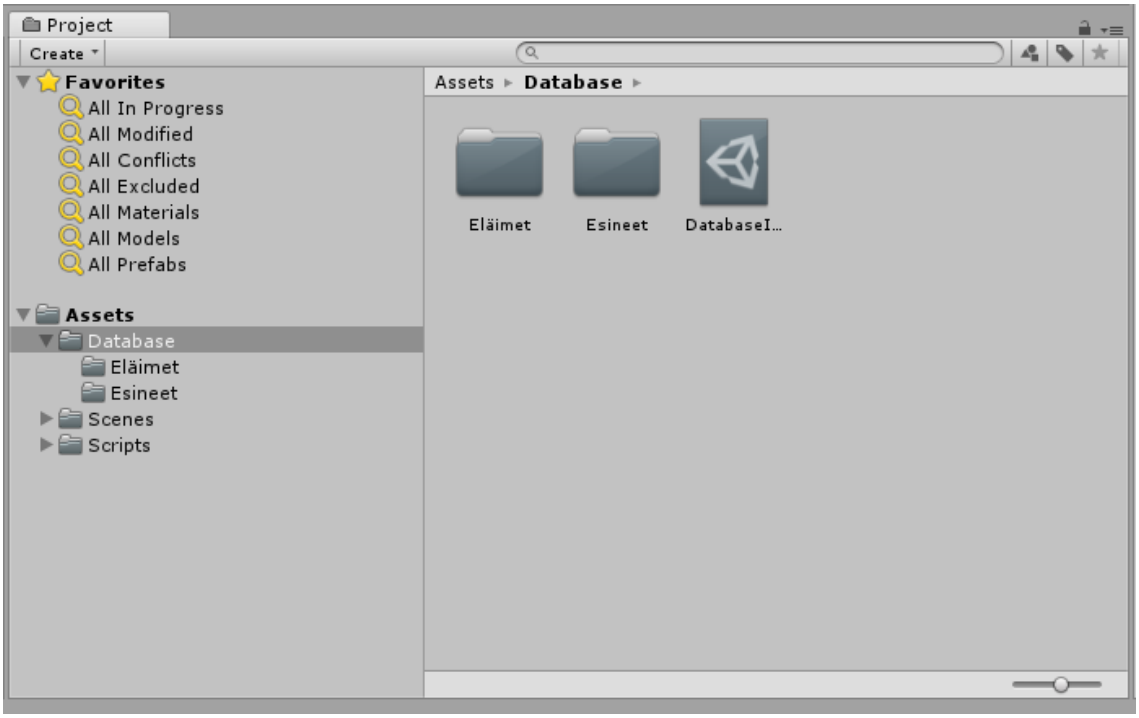
- string
- int
- float
- double
- bool
- Vector2
- Vector3
- Transform
- GameObject
- ScriptableObject
- Color.

Tallennusjärjestelmä on pohjimmiltaan hyvin yksinkertainen ja helposti skaalautuva suurinkiin projekteihin ja isojen tietomäärien tallentamiseen.



Kuva 19. Oman editori-ikkunan näkymä esimerkki item-oliolle.

Kuvassa 20 näkyy tallennusjärjestelmän rakenne ja näkymä ”Database”-kansiosta projekti-ikkunassa. Listojen ScriptableObjectit sijaitsevat omissa asset-tiedostoissaan listan nimeä vastaavien kansioiden alla. Listojen määrää ja nimiä voi näin ollen helposti tarkastella myös Unity-editorin projekti-ikkunasta.



Kuva 20. Database-kansio projekti-ikkunassa.

4.5 Haasteet ja kehitysmahdollisuudet

Vaikka tallennusjärjestelmä valmistui suunnitelmien ja alussa valittujen määritelmien mukaan onnistuneesti, silti ohjelmaa tehdessä vastaan tuli monia asioita, joissa olisi vielä kehittämisen ja lisäominaisuuksien mahdollisuuksia.

Nykyisellään ainoastaan listan nimissä tehdään tarkistus, että nimi ei ole tyhjä tai vastaavannimistä listaa ei ole ennestään olemassa. Listan kohdalla nimen tarkistus on tärkeää, koska listan nimi tulee myös kansion nimeksi ja kahta samannimistä kansiota ei voi sijaita saman kansion alla. Item-oliota luotaessa, mikäli nimi-kenttä on tyhjä, item-olion nimeksi tulee automaattisesti ”Item” ja tietty numero. Numero muodostuu siitä, monesko listan item-olio on kyseessä. Item-olion nimen tarkistuksessa olisi voinut tehdä erilaisia tarkistuksia siitä, löytyykö vastaava item-olio jo listalta. Mikäli sellainen löytyy, nimen perään tulisi numero kuvastamaan, monesko samanniminen listan item-olio on kyseessä. Item-olioiden kohdalla nimien erilaisesta tarkastelusta luovuttiin, koska joskus voi olla tarpeellista, että listalla on useampi samanniminen item-olio. Item-olion nimen muutos jälkeenpäin on myös helppoa ja mikäli nimeen tehtäisiin erilaisia tarkistuksia, ne

jouduttaisiin tekemään monessa vaiheessa. Nimen tarkistukset toisivat ylimääräistä hitautta tallennusjärjestelmään, varsinkin kun kyseessä ovat suuret listat, jotka sisältävät lukuisia item-olioita.

Tallennusjärjestelmään olisi voitu tehdä myös tapa hakea esimerkiksi kaikki tietyssä kansiossa olevat objektit tai prefab-tiedostot. Järjestelmä olisi voinut automaattisesti luoda listan ja tehdä jokaiselle haetulle asialle oman item-olion. Mikäli haetut asiat olisivat esimerkiksi prefab-objekteja, item-olioille voisi tulla automaattisesti prefab-muuttuja ja haettu prefab-objekti lisättäisiin siihen.

Uudet listat ja item-oliot luodaan järjestyksessä yksi kerrallaan. Tallennusjärjestelmään voisi tuoda lisäominaisuuden kopioida, liittää tai monistaa suuria määriä item-oliota kerralla, joko listojen välillä tai saman listan sisällä. Myös kokonaisien listojen monistaminen voisi olla hyödyllistä.

Yhtenä suurimmista tallennusjärjestelmää muokkaavista toiminnoista voisi olla siirtymisen helpottaminen item-olion alle lisättyjen listojen välillä. Mikäli tallennusjärjestelmällä nyt luodaan puumallinen rakenne item-olioille ja listoille, kaikki listat ovat edelleen tallennusjärjestelmässä niiden luontijärjestyksessä. Item-olion alle lisätyn listan kohdalle voisi tulla nappula, joka veisi suoraan oikeaan listaan tallennusjärjestelmässä. Nyt lista pitää etsiä kaikkien tallennusjärjestelmän listojen joukosta esimerkiksi nimen avulla. Mikäli listoja ei ole paljoa, se ei ole kovin työlästä, mutta listojen määrän kasvaessa se muodostuu huomattavasti työläämmäksi. Toinen vaihtoehto olisi listojen näkyminen puurakenteisesti, mutta se tarkoittaisi koko tallennusjärjestelmän uudelleen suunnittelua sellaista tukevaksi.

”Database”-nimisen kansion sijainnin valinta käyttäjän haluamaksi on mahdollista, mutta se tuo mukanaan paljon huomioitavia asioita. Mikäli usea henkilö käyttää järjestelmää, järjestelmä ei toimi, ellei sillä ole tiedossa, missä ”DatabaseIndex”-tiedosto sijaitsee. Tähänkin on tehtävissä varmistuksia, mutta tallennuspaikan valintamahdollisuutta ei lähdetty toteuttamaan. Sijainnin valinta toisi helposti mukanaan mahdollisuuden käyttää samassa Unity-projektissa useaa ”DatabaseIndex”-tiedostoa yhtä aikaa. Järjestelmä kuitenkin suunniteltiin juuri siihen, että kaikki järjestelmässä käytössä olevat listat löytyisivät samasta paikasta hyvin järjesteltynä. Usean DatabaseIndex-tiedoston käyttö ja niiden paikan valinta toisi sekavuutta järjestelmään.

Tallennusjärjestelmän pitkäaikaisen käytön tärkeimpiä lisäyksiä olisi varmuuskopiointimahdollisuuden lisääminen. Varmuuskopiointi olisi mahdollista tehdä pakkaamalla koko tallennusjärjestelmän yhteen asset-tiedostoon. Tämä tiedosto voitaisiin tallentaa käyttäjän määrittelemään kansioon. Varmuuskopion palauttaminen onnistuisi, kun käyttäjä valitsisi sellaisen asset-tiedoston, joka sisältää kelvollisen ”DatabaseIndex”-tyyppisen ScriptableObjectin. Varmuuskopiointi ja sen palautus ei vaatisi nykyiseen järjestelmään tehtäviä muutoksia, mutta sen toiminnon toteuttaminen toisi mukanaan tietenkin uusia haasteita ja huolellisesti tarkistettavia varmistuksia. Varmuuskopiointimahdollisuutta ei lähdetty tämän projektin puitteissa toteuttamaan. Tallennusjärjestelmän varmuuskopiointi onnistuu kyllä, mutta se on tehtävä manuaalisesti kaikki tiedot muualle kopioimalla ja sieltä tarvittaessa palauttamalla.

Projektin aikana ilmeni muutamia haasteita ja ongelmia. Haasteet ovat yleisiä ohjelman- ja pelinkehityksen alalla. Suurin haaste oli se, että ScriptableObjectin tiedot eivät säilyneet, kun Unityn sulki ja käynnisti uudestaan. Tiedot säilyivät kyllä pelivaiheeseen mentäessä ja sieltä poistuessa, mutta ei uudelleen käynnistyksessä. Ongelma johtui siitä, että tallennusjärjestelmän listat eivät serialisoituneet oikein ja listojen luonnin aikana tehdyt muutamat serialisointiin liittyvät korjaukset ratkaisivat ongelman.

Toisena haasteena projektin edetessä tuli vastaan editori-ikkunan jäsentely. Jonkin aikaa kului ongelman selvittämiseen, sillä ikkunan osat ja nappulat eivät järjestyneet odotetulla tavalla. Lopulta paljastui, että ongelman aiheutti yksi yksittäinen ryhmän aloitus, jota ei tietyissä tilanteissa lopetettu oikeassa kohdassa. Koodissa ryhmät aloitetaan esimerkiksi EditorGUILayout.BeginVertical()-funktiolla ja lopetetaan EditorGUILayout.EndVertical()-funktiolla. Ryhmän osion lopetuksen varmistaminen kaikissa tilanteissa korjasi ongelman. Omaa editori-ikkunaa tehtäessä kannattaa kiinnittää erityisesti huomiota ”OnGUI”-funktiossa ryhmien osien määriin ja niiden kaikkien sulkemiseen halutussa vaiheessa.

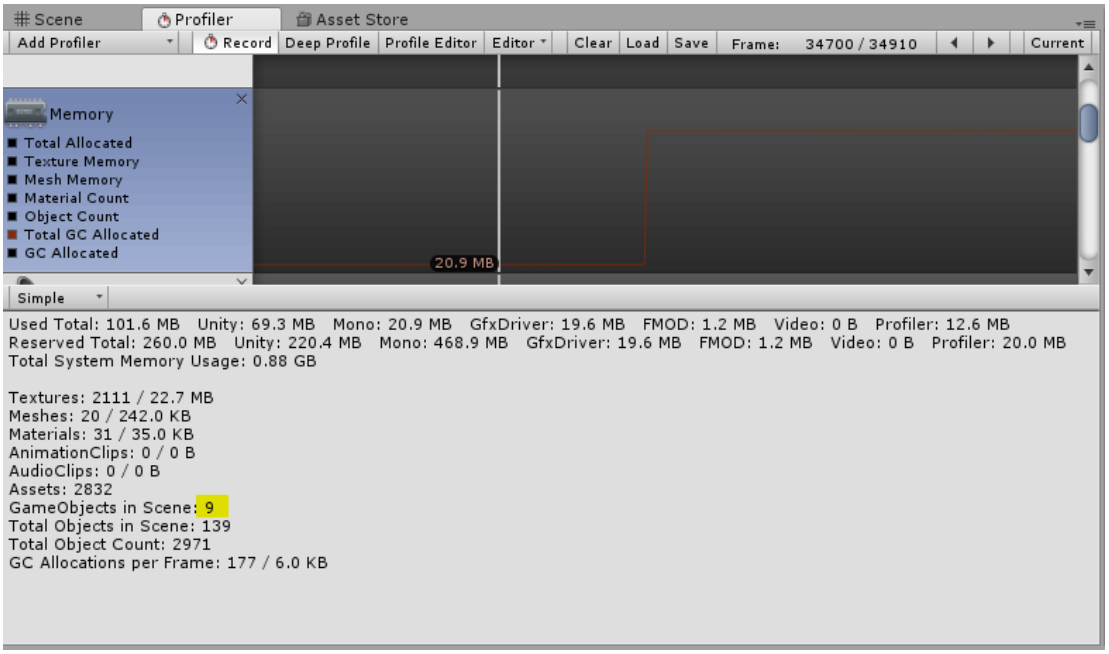
5 Tulokset ja tulevaisuuden näkymät

IT-alan kehityksen vauhti on huomattavan nopeaa. Muutoksia ja parannuksia tulee jatkuvasti, niin ohjelmistopuolella kuin laitteistopuolellakin. Ohjelmistopuolella keskittyminen muistin hallintaan on ollut hitaampaa, mutta DOD-malli on kerännyt erityisesti huo-

miota. DOD-malli tulee odotetusti edelleen kasvamaan ja leviämään mahdollisesti varteenotettavaksi vaihtoehdoksi OOP-mallille, myös pelialalla. Ihmiset ovat kuitenkin vastahakoisia muutoksille, ja pelinkehittäjien vuosia käyttämää OOP-mallia on vaikea lähteä muuttamaan. Kokonaan uuteen toimintamalliin vaihtaminen tuo tietenkin aina mukanaan omat haasteensa. DOD-mallin lisäksi OOP-malliin on myös hyviä muistin käytön parannusvaihtoehtoja, ja yksi niistä on ainakin Unityn pelinkehitysympäristöstä löytyvä ScriptableObject. ScriptableObjectien käyttö on edelleen OOP-mallin mukaista, mutta se tuo mukanaan etuja muun muassa muistinhallintaan, kuten myös käytettävyyteen.

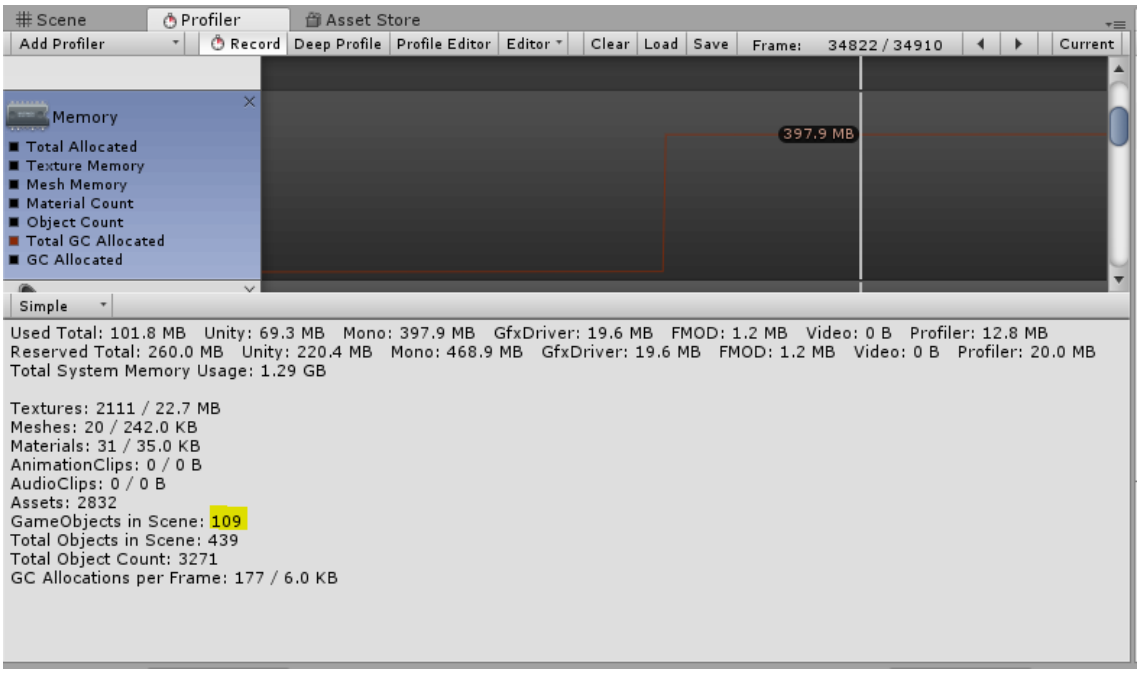
ScriptableObjectien tuoman hyödyn varmistamiseksi niitä testattiin luvussa 3.3 esiteltyä esimerkkitapausta laajentamalla. Tulokset otettiin suoraan Unityn oman profilointi-ikkunan (engl. Profiler Window) kautta. Profilointi-ikkuna antaa reaaliaikaista dataa muun muassa prosessorin, grafiikkasuorittimen ja muistin kulutuksesta pelin aikana. Testissä peliin luotiin miljoonan numerosarjan kokoisen taulukon sisältämästä prefab-tiedostosta sata kopio-objektia.

Kuvassa 21 näkyvä valkoinen pystyviiva kuvaa valittua ajanhetkeä. Kaikki kuvassa näkyvät numerot kertovat tilanteesta ja muistin käytöstä juuri kyseisellä hetkellä. Kuvassa keltaisella korostuksella on merkitty pelissä olevien objektien määrä, joka on yhdeksän. Muistin käyttö valitulla ajanhetkellä on 20,9 megatavua.



Kuva 21. Unityn profilointi-ikkunan kuva muistin kulutuksesta ennen kopioiden luontia tavallisesta prefab-tiedostosta.

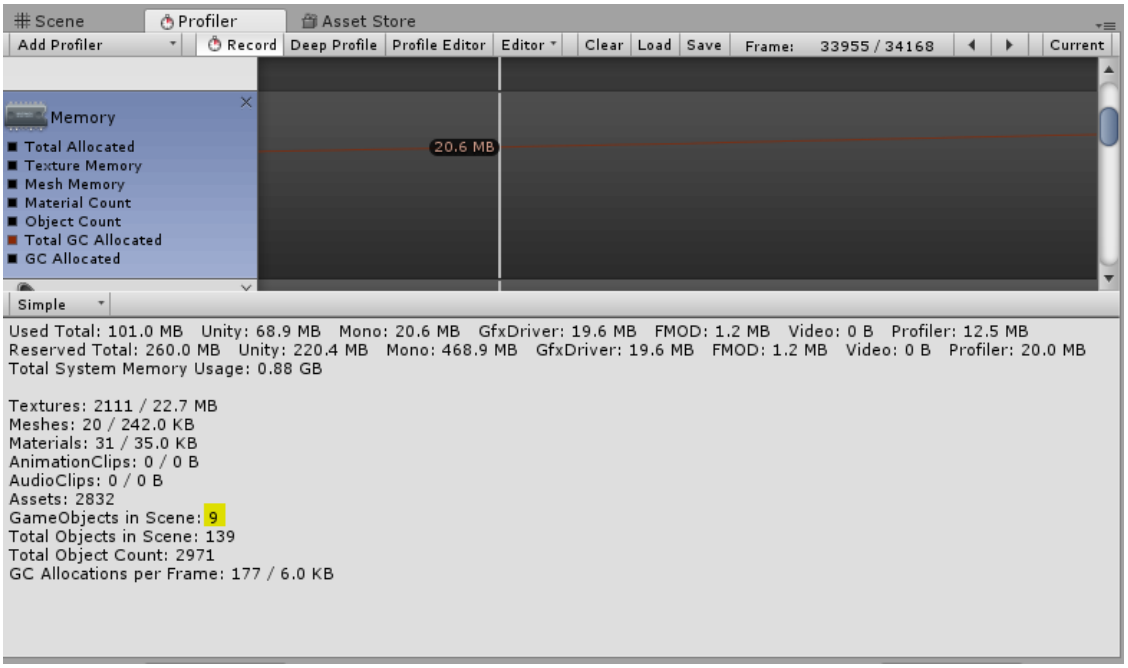
Kopioitujen objektien luomisen ajankohdan erottaa selvästi kuvasta 22 muistinkäytön suurena hyppäyksenä. Kuvassa 22 valittu ajanhetki on kopioitujen objektien lisäyksen jälkeen ja muistinkäyttö on noussut 20,9 megatavusta aina 397,9 megatavuun asti. Sata lisättyä objektia näkyvät kuvasta keltaisella korostuksella merkittynä, kun yhteensä objektien määrä on 109.



Kuva 22. Unityn profilointi-ikkunan kuva muistin kulutuksesta kopioiden luomisen jälkeen tavallisesta prefab-tiedostosta.

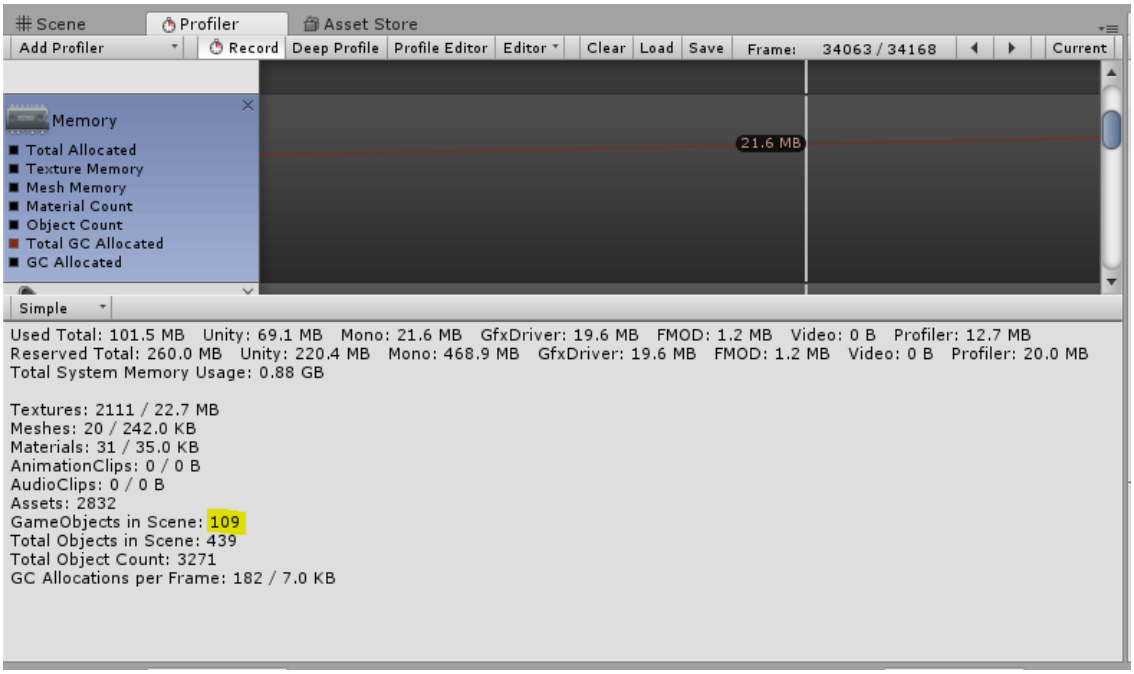
ScriptableObjectien hyödyn varmistamiseksi sama testi toistettiin ScriptableObjecteja käyttämällä. Taulukko oli myös tässä tapauksessa miljoonan numerosarjan kokoinen, mutta se sijaitsee ScriptableObjectissa.

Kuvassa 23 näkyy tavallisella prefab-tiedostolla suoritettua testiä (kuva 21) vastaava alkutilanne, kun käytössä ovat ScriptableObjectit. Objekteja on vastaavasti valitulla ajanhetkellä yhdeksän. Muistia on käytössä 20,6 megatavua. Aikaisempaan testiin (kuva 21) verrattuna 0,3 megatavun ero johtuu yleisestä muistin käytön vaihtelusta pelivaiheessa.



Kuva 23. Unityn profilointi-ikkunan kuva muistin kulutuksesta ennen ScriptableObjectia hyödyntävien kopioiden luontia.

Kuvassa 24 ei näy muistinnäytössä huomattavaa piikkiä kopioitujen objektien luontivaiheessa. Lisäyksen jälkeen käytössä oleva muisti on vain 1 megatavu enemmän kuin ennen sitä.



Kuva 24. Unityn profilointi-ikkunan kuva muistin kulutuksesta ScriptableObjectia hyödyntävien kopioiden luomisen jälkeen.

ScriptableObjectien tapauksessa 1 megatavun muistinkäytön lisäyksestä suurin osa on tavallista pelivaiheen aikana tapahtuvaa muistinkäytön vaihtelua. Objekteilla olevien alkuperäiseen taulukkoon osoittavien viitteiden käyttämä muisti on niin pieni, että sitä ei tarkasti voi edes erottaa Unityn profilointityökalulla. ScriptableObjeteilla toteutetun tallennusjärjestelmän hyödyt, nimenomaan muistin käyttöön liittyen, ovat nähtävissä kuvia 22 ja 24 vertaamalla.

Insinöörityönä tehty tallennusjärjestelmä on toimiva, ja koska se käyttää hyödykseen ScriptableObjecteja, se on muistin hallinnan kannata myös hyvä vaihtoehto pelkälle MonoBehaviour-skriptin ja prefab-tiedoston yhdistelmälle. Pelkästään ScriptableObjectien käyttö ei myöskään ole hyödyllisintä, vaan parhaat tulokset saadaan, kun sitä käytetään yhdessä MonoBehaviour-skriptien ja prefab-tiedostojen kanssa. ScriptableObjectien käyttö ohjaa suoraan tiedon jaotteluun, vaihtuviin kaikille objekteille yhteisiin arvoihin ja kaikille objekteille yksilöllisiin arvoihin. Yhteiset arvot on hyödyllistä säilyttää ScriptableObjecteissa, ja objekteille yksilölliset arvot on hyvä säilyttää MonoBehaviour-skripteissä. Skriptit sitten taas voivat olla osana prefab-tiedostoa, joka vuorostaan voi esimerkiksi olla osana ScriptableObjectia.

6 Yhteenveto

Insinöörityössä perehdyttiin erilaisiin tiedon tallennusjärjestelmän malleihin. Erityisesti keskityttiin malleihin, jotka ovat käytössä pelinkehityksessä. Nämä mallit ovat OOP-malli ja DOD-malli, joita vertailtiin keskenään. Yksityiskohtaisemmin käytiin läpi Unity-pelinkehitysympäristössä vallitseva OOP-malli. Unityn OOP-mallia läpikäydessä perehdyttiin MonoBehaviour-skripteihin, prefab-tiedostoihin ja ScriptableObjecteihin. MonoBehaviour-skripteistä ja ScriptableObjecteista käytiin läpi niiden hyviä ja huonoja puolia. Tästä oli myös esimerkkitapaus, josta näkee vielä selkeämmin näiden kahden eron. Esimerkkitapaus toteutettiin myös Unityssä ja tarkasteltiin saatuja tuloksia. ScriptableObjectien tuoma hyöty muistin hallinnan kannalta tuli tuloksista selvästi esille. Tallennusjärjestelmää varten Unityyn tehdyn editori-ikkunan lisäyksen ansiosta sillä voidaan myös käsitellä isoja taulukkoja. Unityn omassa editori-ikkunassa ScriptableObjectia tarkastellessa on rivimääräinen raja, jota ikkuna ei saa ylittää. Tämä rajoittaisi tallennusjärjestelmän käyttöä huomattavasti.

Insinöörityönä tehdyn tallennusjärjestelmän määrittelystä, toteutuksesta ja tuloksista on raportissa tarkat kuvaukset. Tallennusjärjestelmä käyttää hyväkseen ScriptableObjectin parhaita puolia, jotka liittyvät muistin hallintaan ja käytön yksinkertaisuuteen. Käytön yksinkertaisuus tulee siitä, että järjestelmä ei sisällä kutsuja sen ulkopuolelle. Järjestelmä on siis irrallinen kaikista muista pelin mahdollista osista. Pelin muut osat voivat tosin käyttää vain sitä tai sen tietoja hyväkseen. Tästä syystä tallennusjärjestelmä on helppoa siirtää eri Unity-projektien välillä ja se sopii monentyyppisiin tarkoituksiin. Projektin ulkopuolelle jäi vielä parannus- ja kehitysmahdollisuuksia. Kaikki parannus- ja kehitysmahdollisuudet ovat toteutettavissa, ja niihin palataan kun tallennusjärjestelmä otetaan käyttöön tulevien peliprojektien yhteydessä.

Insinöörityöraportti on kokonaisuutena kattava katsaus eri suunnittelu-, ohjelmointi- ja tallennusmalleihin. Ne kaikki ovat keskeisiä asioita ohjelmien ja pelien kehityksessä. Käytössä olevat tavat ja mallit vaihtuvat aika ajoin, mutta niiden yleinen pyrkimys asioiden tehostamiseen ja optimointiin säilyy. Tietokoneiden ja niiden osien kehittyessä myös niiden suorituskyky kasvaa. Siitä huolimatta ohjelmistopuolella tehdyt parannukset tuovat niin suuria hyötyjä, ettei niiden hylkääminen ole järkevää. Insinöörityöhön perehtymällä tutustuu nykyisin käytössä oleviin mahdollisuuksiin ohjelman tehostamiseen ja optimoimiseen liittyen. Tästä syystä se on hyödyllinen niin pelinkehittäjille kuin kenelle tahansa ohjelmistojen kehittäjälle tai suunnittelijalle.

Lähteet

1 Introduction to Scriptable Objects. Verkkoaineisto. Unity. <https://unity3d.com/learn/tutorials/modules/beginner/live-training-archive/scriptable-objects>. Luettu 14.12.2017.

2 Game Architecture with Scriptable Objects. 2017 Verkkoaineisto. Unity. <https://youtu.be/raQ3iHhE_Kk>. Luettu 14.12.2017.

3 Drepper, Ulrich. 2007. What Every Programmer Should Know About Memory. Red Hat, Inc.

4 History of C++. Verkkoaineisto. Cplusplus. <http://www.cplusplus.com/info/history/>. Luettu 30.3.2018.

5 Welcome to the Amazon GameDev Blog! 2016. Verkkoaineisto. Amazon Web Services, Inc. <https://aws.amazon.com/blogs/gamedev/welcome-to-the-amazon-gamedev-blog/>. Luettu 30.3.2018.

6 Frequently Asked Questions. Verkkoaineisto. Amazon Web Services, Inc. <https://aws.amazon.com/lumberyard/faq/>. Luettu 30.3.2018.

7 Dogged Determination. 2014. Verkkoaineisto. Naughty Dog, Inc. <http://www.gameenginebook.com/resources/SINFO.pdf>. Luettu 30.3.2018.

8 Culling the Battlefield Data Oriented Design in Practice. Verkkoaineisto. Frostbite. <https://www.ea.com/frostbite/news/culling-the-battlefield-data-oriented-design-in-practice>. Luettu 30.3.2018.

9 We’re joining Unity to help democratize data-oriented programming. 2017. Verkkoaineisto. Unity. <https://blogs.unity3d.com/2017/11/08/were-joining-unity-to-help-democratize-data-oriented-programming/>. Luettu 30.3.2018.

10 Monobehaviour flowchart. Verkkoaineisto. Unity. <https://docs.unity3d.com/uploads/Main/monobehaviour_flowchart.svg>. Luettu 30.3.2018.

11 Overthrowing the MonoBehaviour Tyranny in a Glorious Scriptable Object Revolution. 2016. Verkkoaineisto. Unity. <https://youtu.be/6vmRwLYWNRo>. Luettu 24.3.2018.

12 Thorn, Alan. 2016 Unity 5.x By Example. Birmingham: Packt Publishing Ltd.

13 Williams, Andrew. 2017. History of Digital Games: Developments in Art, Design and Interaction. Focal Press.

14 ScriptableObject. Verkkoaineisto. Unity. <https://docs.unity3d.com/Manual/class-ScriptableObject.html>. Luettu 24.3.2018.

15 Sherrod, Allen. 2007. Data Structures and Algorithms for Game Developers. Charles River Media.

16 NVIDIA RTX Technology Delivers Biggest Advance in Computer Graphics in 15 Years. 2018. Verkkoaineisto. Nvidia. <https://nvidianews.nvidia.com/news/nvidia-reinvents-the-workstation-with-real-time-ray-tracing>. Luettu 31.3.2018

17 What is Data-Oriented Game Engine Design? 2014. Verkkoaineisto. Envato. <https://gamedevelopment.tutsplus.com/articles/what-is-data-oriented-game-engine-design--cms-21052>. Luettu 29.3.2018.

18 How Unity's Serialization system works. 2017. Verkkoaineisto. Unity. <https://youtu.be/N-HJvfVuKRw>. Katsottu 1.4.2018

19 Script Serialization. 2017. Verkkoaineisto. Unity. <https://docs.unity3d.com/Manual/script-Serialization.html>. Luettu 30.3.2018

20 Serialization in-depth with Tim Cooper: a Unite Nordic 2013 presentation. 2013. Verkkoaineisto. Unity. <https://youtu.be/MmUT0ljrHNc>. Katsottu 24.3.2018.

21 Enumerations. Verkkoaineisto. Unity. <https://unity3d.com/learn/tutorials/topics/scripting/enumerations>. Luettu 1.4.2018

22 Collaborate. 2017 Verkkoaineisto. Unity. <https://docs.unity3d.com/Manual/UnityCollaborate.html>. Luettu 1.4.2018.

23 Collin, Daniel. Culling the Battlefield Data Oriented. Verkkoaineisto. <http://www.gdcvault.com/play/1014491/Culling-the-Battlefield-Data-Oriented>. Luettu 3.4.2018

24 Doran, John. 2016 Unity 5.x Game Development Blueprints. Packt Publishing.

25 Smith, Matt & Queiroz, Chico. 2015 Unity 5.x Cookbook. Packt Publishing.

26 Data-Oriented Design. 2013. Verkkoaineisto. Game Engine Architecture Club. <https://youtu.be/16ZF9XqkfRY>. Luettu 18.4.2018