Osaamista
ja oivallusta
tulevaisuuden
tekemiseen

Suvi Karjalainen

Puunäkymä tiedoston muokkauksen helpottamiseen

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikka

Insinöörityö

31.10.2019

Tekijä Otsikko Sivumäärä Aika	Suvi Karjalainen Puunäkymä tiedoston muokkauksen helpottamiseen 37 sivua 31.10.2019
Tutkinto	Insinööri (AMK)
Tutkinto-ohjelma	Tieto- ja viestintätekniikka
Ammatillinen pääaine	Pelisovellukset
Ohjaaja	Lehtori Antti Laiho

Insinöörityön tavoite oli tehdä selkeä ja helppokäyttöinen editori Unity-pelimoottorilla tehdyn pelin asetuksille, jotka ovat ScriptableObject-tiedostossa. Pelimoottorin oletusnäkymä ScriptableObject-tiedostoille on vaikealukuinen, kun kenttien määrä kasvaa. Asetuksia on yli 200, joten ihmissilmän on vaikea selata niitä ja uuden näkymän luominen oli tarpeen.

Työssä tehtiin puunäkymä pelin asetuksille käyttäen Unity-pelimoottorin TreeView-luokkaa, joka on IMGUI-kontrolli. Se ylläpitää puun rakennetta ja huolehtii, miten puun alkiot piirretään. Tämän lisäksi työssä tehtiin EditorWindow-luokasta periytyvä luokka, joka ylläpitää puun tilaa ja päivittää näkymää ikkunassa, joka piirretään Unity-näkymään.

TreeView- ja EditorWindow-luokista tehtiin omat luokat, jotka molemmat tarvitaan puunäkymään. Työn edetessä kävi ilmi, että TreeView-luokka on erinomainen työkalu suuren datamäärän näyttämiselle, kun data voidaan lajitella järkevällä tavalla. Työn data oli alun perinkin jaoteltu aihealueiden mukaan, joiden rajat merkitään lisämääreillä, ja datan rakenne oli pitkälti otsikkojen alla olevia kenttiä ja listoja, joten puurakenne oli luonnollinen lähestymistapa, vaikka puun syvyys suurelta osalta on vain yksi.

Insinöörityön tulokset ovat lupaavia. Jo pelkän hakukentän ansiosta työryhmä säästää aikaa tiedoston muokkauksessa, mikä mahdollistaa nopeampien muutosten tekemisen ja täten nopeamman työskentelyn. Tuloksista voidaan päätellä, että Unity-pelimoottorin kanssa työskennellessä kannattaa luoda omia editoreja, etenkin jos kyseessä on tiedosto, jota muokataan usein. Työhön liittyvän peliprojektin yksi muokatuimpia tiedostoja on sen asetukset, joten oli luonnollista tehdä sille oma editori.

Vaikka editori tehtiin tietyn peliprojektin tietylle tiedostolle, sitä voidaan pienillä muokkauksilla hyödyntää minkä tahansa Unity-projektin vastaavassa tiedostossa. Editori vaatii, että sille annetaan ScriptableObject-tiedosto näytettäväksi ja koodimuutokset ovat vähäisiä.

Avainsanat	Unity, TreeView, Editoriohjelmointi

Author Title Number of Pages Date	Suvi Karjalainen Tree view for simplifying the editing of a file 37 pages 31 October 2019
Degree	Bachelor of Engineering
Degree Programme	Information and communications technology
Professional Major	Game development
Instructor	Antti Laiho, Senior Lecturer

The goal of the final year project was to make an easy-to-use editor for a ScriptableObject file which contains settings for a game that was made with the Unity game engine. The Unity game engine's default view for ScriptableObject files is hard to read so it was necessary to create a new editor for it.

The final year project used an IMGUI control called TreeView to make an editor for the settings file which contains the structure for a tree data model. In addition, it was necessary to make an editor window for it which draws the tree into the Unity view. The TreeView class is a useful control to show large amounts of data if it can be sorted into a tree data model.

The results showed that a search bar alone makes working more efficient because it saves time while editing files. The conclusion that can be drawn is that when working with the Unity game engine, it is beneficial to make editors for the most edited files. Although the editor is made specifically for the settings file, it can be utilized for any ScriptableObject file with relatively small changes. It does require some code tweaks due to how generic classes are handled but generally speaking it is highly flexible.

Keywords	Unity, TreeView, Editor programming

Sisällys

Lyhenteet

GUI	Graphical user interface. Käyttäjäliittymä, eli näytöllä näkyvät kontrollit, kuten napit ja valikot.
IMGUI	Immediate mode graphical user interface. Käyttäjäliittymä, jonka GUI-elementit luodaan koodissa.
RMGUI	Retained mode graphical user interface. Käyttäjäliittymä, jonka GUI-elementit luodaan olioina.

1 Johdanto

Insinöörityössä oli tarkoitus tehdä editori eli muokkain erään peliprojektin asetuksille käyttäen Unity-pelimoottoria ja sen tarjoamia työkaluja. Peliprojekti on mobiilipeli, joka oli myös tehty Unity-pelimoottorilla, ja insinöörityön aihe valittiin sen perusteella, että pelin asetusten oletusnäkymä haluttiin muuttaa käyttäjäystävällisempään muotoon. Peliprojektin asetukset ovat staattisessa luokassa nimeltä Settings, ja luokasta on tehty ScriptableObject-tiedosto, johon pääsee käsiksi muista luokista. Settings-luokka sisältää erilaisia muuttujia, kuten kokonaislukuja, listoja ja animaatiokäyriä, joita hyödynnetään muualla pelissä.

Peliprojektin koon vuoksi Settings-luokka sisältää yli kaksisataa riviä muuttujia, joita on vaikea navigoida ScriptableObject-tiedostojen oletusnäkymän kautta. Oletusnäkymä ei jaottele kenttiä järkevästi, ja Unity-pelimoottorin lisämääreet, jotka on tarkoitettu näkymien jaotteluun, eivät enää riittäneet. Ihmissilmän on vaikea löytää tiettyjä sanoja listasta, jota ei ole järjestetty muutoin, kuin että samaan aihealueeseen liittyvät sanat ovat peräkkäin, vaikka niiden yllä olisikin otsikko. Oletusnäkymä todettiin peliprojektin tarpeisiin sopimattomaksi, koska sen käyttö vei liikaa aikaa, joten sille päätettiin tehdä editori, joka ratkaisisi tämän ongelman.

Unity-pelimoottori käyttää C#-ohjelmointikieltä, joten editori tuli kirjoittaa sillä. Editorissa tuli myös hyödyntää TreeView- ja EditorWindow-luokkia, joiden avulla Settings-tiedoston muuttujat voidaan jaotella puutietorakenteen mallin mukaan ja piirtää Unity-näkymään. Muita vaatimuksia olivat hakukenttä ja yleiset parannukset ulkoasussa. Insinöörityön tavoite oli helpottaa, selkeyttää ja nopeuttaa peliprojektin asetusten ScriptableObject-tiedoston muokkausta ja pohtia, onko TreeView-luokka sopiva työkalu peliprojektin asetusten kaltaisen datan jaotteluun. Oletettu lopputulos oli, että vaikka TreeView-luokka ei olisi paras työkalu tähän käyttötarkoitukseen, se kuitenkin poikkeaisi oletusnäkymästä riittävästi tarjotakseen näkyvän eron tiedoston muokkaamiseen käytettyyn aikaan.

2 Puunäkymän rakentamisen pohjatiedot ja käsitteet

Insinöörityö tehtiin Unity-pelimoottorin editoriominaisuuksia käyttäen. Unity-pelimoottori on Unity Technologies -yrityksen luoma ohjelma, joka on tarkoitettu sekä 2D- että 3D-pelien tekoon. Pelimoottori on järjestelmäriippumaton, ja sillä voi kääntää pelejä yli 25:lle eri alustalle [1]. Se on tarjolla Windows- ja MacOS-tietokoneille ja tukee C#-ohjelmointikieltä [2; 3]. Se tuki vuoteen 2018 asti sen omaa versiota JavaScript-ohjelmointikielestä nimeltä UnityScript, joka on sittemmin otettu pois käytöstä, ja vuoteen 2015 asti se tuki myös Boo-ohjelmointikieltä [4; 5]. Unity-pelimoottori julkaistiin vuonna 2005, ja vuonna 2016 Unity ilmoitti, että yli 5,5 miljoonaa käyttäjää on rekisteröitynyt alustalle [6]. Vuonna 2016 Unityn markkinaosuus oli 45 %, ja vuonna 2019 puolet mobiilipeleistä on tehty Unityllä [7; 8].

Peliprojekti, jota varten insinöörityö tehtiin, on mobiilipeli, ja se tehtiin Unity-pelimoottorilla, sillä se on suosituin alusta mobiilipelien tekemiseen ja pärjää hyvin tietokonepelimarkkinoillakin [8]. Työssä tehty editori tehtiin C#-ohjelmointikielellä Unity-pelimoottorin TreeView- ja EditorWindow-luokkia hyödyntäen, ja sen tarkoitus on parantaa pelimoottorin oletusnäkymää. Työ tehtiin Unity-pelimoottorin versiolla 2018.4.9f1, mutta se toimii tuoreemmillakin versioilla.

Peliprojektin Settings-luokka periytyy ScriptableObject-luokasta, joten asetuksista voidaan luoda ScriptableObject-tiedosto, joka on Unity-pelimoottorissa määritelty tietosäiliö. Se periytyy UnityEngine.Object-luokasta, ja sen tarkoitus on vähentää dynaamista muistinkäyttöä luomalla tiedosto, joka sisältää luokassa määriteltyä dataa. ScriptableObject-luokasta periytyvistä luokista voidaan luoda asset-tiedosto projektin kansioihin, ja se voidaan serialisoida. ScriptableObject-tiedosto on ilmentymä luokan sisältämästä datasta. Niitä voi olla useampia, joista jokaisella on omat arvonsa. Peliprojektissa tiedostoon on määritelty eri arvoja, jotka ovat olennaisia ohjelman eri vaiheissa. ScriptableObject-tiedosto on aina ladattu muistiin, joten niihin ei ole järkevää tallentaa esimerkiksi prefab-tiedostoja, jotka vievät huomattavasti enemmän muistia kuin primitiiviset kentät kuten kokonaisluvut. Tyypillisesti ScriptableObject-tiedostot soveltuvat erilaisten esiasetusten luomiseen. Vaikka peliprojektissa käytetään vain yhtä Settings-luokasta luotua ScriptableObject-tiedostoa, siitä voitaisiin tehdä toinen tiedosto eri arvoilla, joita voidaan käyttää

projektissa nykyisten sijaan, mikäli siinä haluttaisiin kokeilla täysin eri asetuksia menettämättä nykyisiä arvoja. [9.]

2.1 Serialisointi eli sarjallistaminen

Serialisoinnilla eli sarjallistamisella tarkoitetaan yksinkertaisimmillaan datan tallentamista muotoon, joka voidaan myöhemmin lukea ja palauttaa ohjelmaan. Peliprojektin asetukset on serialisoitu, sillä Unity-pelimoottori serialisoi ScriptableObject-tiedostot oletusarvona. Datan palauttamista käytettävään muotoon kutsutaan deserialisoinniksi. Serialisoidessa data käännetään muotoon, joka voidaan tallentaa tietokantaan, laitteen muistiin tai tiedostoon. Toimenpiteen tarkoitus on tallentaa olion tila, jotta se voidaan myöhemmin palauttaa. Peliohjelmoinnissa käytännön esimerkki on tallentaminen. Tallennettaessa pelaajahahmon tila voidaan serialisoida laitteen muistiin. Serialisoidessa voidaan tallentaa esimerkiksi pelaajan senhetkinen rahamäärä, sijainti ja elämät. Kun peli ladataan uudestaan, deserialisointiprosessi palauttaa tiedot pelille ja peli ottaa arvot käyttöön oletusarvojen sijaan. [10.]

Kentällä viitataan luokan tai struct-rakenteen jäsenmuuttujaan, esimerkiksi Settings-luokan sisällä olevaan kokonaislukukenttään. Funktioiden sisällä määritellyt muuttujat eivät ole kenttiä. Jotta kenttä voidaan serialisoida Unityssä, sen täytyy täyttää tietyt vaatimukset. Sen täytyy olla julkinen, tai sen on oltava merkitty ”SerializeField”-attribuutilla. Attribuuteilla voidaan lisätä metadataa käännökseen, jota kääntäjä voi lukea ja toimia niiden mukaan [11]. Jotta kentän voi serialisoida, sitä ei saa olla merkitty static-, const- tai readonly-määreillä. Static-määre merkitsee kentän osaksi luokkaa eikä sen ilmentymää. Const-avainsanalla merkitään kenttiä, jotka määritellään kerran, ja ne pysyvät samana koko ohjelman suorituksen ajan. Readonly-avainsanalla merkitään kenttiä, joita ei voi muokata mutta jotka voidaan määritellä esimerkiksi luokan muodostimessa (engl. constructor). [12; 13; 14.]

Kentän on oltava serialisoitavaa tyyppiä, jotta se voidaan serialisoida. Primitiiviset datatyypit, kuten int, float ja string, sekä listat ja taulukot voidaan serialisoida. Jotta luokka tai struct voidaan serialisoida, se täytyy merkitä ”Serializable”-attribuutilla eikä se saa olla abstrakti. Oliot, jotka periytyvät UnityEngine.Object-luokasta, voidaan serialisoida. Esimerkiksi ScriptableObject-tiedostot, mutta myös prefab-tiedostot on serialisoitu Unity-

pelimoottorissa. [15.] Luokan ja struct-rakenteen ero on, että luokasta luodaan ilmentymä ja struct on ikään kuin säiliö kentille, joita se sisältää. Kun luokan ilmentymän antaa parametrina funktiolle, siitä ei luoda kopiota, vaan funktiossa tehdyt muutokset säilyvät myös funktion ulkopuolella. Kun struct-rakenne annetaan funktiolle parametrina, siitä luodaan kopio ja funktion sisällä tehdyt muutokset eivät tallennu sen ulkopuolella olevaan struct-kenttään.

SerializedObject ja SerializedProperty ovat UnityEditor-luokassa määriteltyjä luokkia. Ne tarjoavat geneerisen tavan käsitellä olioita ja niiden kenttiä. SerializedObject on geneerinen viittaus olioon tai tiedostoon, joka on serialisoitu, esimerkiksi peliprojektissa Settings-luokalle luotuun ScriptableObject-tiedostoon. SerializedProperty puolestaan on viittaus olion tai tiedoston yhteen kenttään, joka on serialisoitu. Unity-editoreja tehdessä suositellaan SerializedObject-luokan käyttöä, kun muokataan olioita. [16; 17.]

SerializedObject ylläpitää kenttien muokkaukseen liittyvää tietojonoa (engl. data stream) ja sitä, että arvojen muokkaus tallentuu muutosten kumoamisjärjestelmään. Mikäli samaa tiedostoa muokataan usean SerializedObject-ilmentymän kautta, käyttäjän on manuaalisesti huolehdittava, että ne ovat ajantasalla. Insinöörityössä peliprojektin Settings-tiedostoon vittataan SerializedObject-muuttujan kautta. [16.]

SerializedProperty sisältää tietoa yhdestä serialisoidusta kentästä. Se sisältää muun muassa arvon jokaiselle primitiiviselle tai Unity-pelimoottorille ominaiselle tyypille, kuten kokonaisluvuille, animaatiokäyrille, merkkijonoille ja liukuluvuille. Se sisältää myös tiedot kentän tyypistä merkkijonona, onko kenttä taulukko, mille SerializedObject-ilmentymälle kenttä kuuluu ja kentän nimen. SerializedProperty-luokkaa suositellaan käytettävän yhdessä SerializedObject-luokan kanssa. [17.]

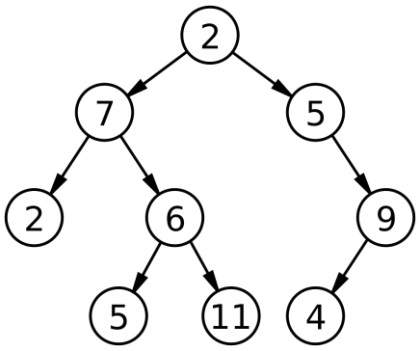
2.2 Puutietorakenne tietotekniikassa

Insinöörityön editori rakennettiin puutietorakenteen pohjalta. Puutietorakenne koostuu solmuista, jotka on hierarkkisesti linkitetty toisiinsa. Puulla on yksi kantasolmu (engl. root), josta puun muut solmut lähtevät. Solmulla voi olla vain yksi ”vanhempi” (engl. parent), jotta puurakenne säilyy. Lapsisolmujen määrää ei ole rajattu, ellei kyseessä ole

tietyntyyppinen puurakenne. Esimerkiksi binääripuussa lapsisolmujen määrä on rajattu kahteen ja se on yleisin puutietorakenne tietojenkäsittelyssä.

Nimeä “puu” käytti ensimmäisen kerran vuonna 1857 matemaatikko Arthur Cayley [18]. On kuitenkin mainitsemisen arvoista, että matemaatikko K. G. C von Staudt käytti puurakenteen ideaa kymmenen vuotta aiemmin vuonna 1847 todistaakseen Eulerin monitahokkaan matemaattisen lausekkeen, joten von Staudtia voidaan pitää yhtenä ensimmäisistä ihmisistä, jotka käyttivät puutietorakenteen ideaa [19]. Koska tietotekniikka nojaa vahvasti matemaattisiin periaatteisiin, nämä voidaan ajatella puurakenteen alkuperänä myös tietotekniikassa. Nykyisellään puut ovat yksi yleisimmistä tietorakenteista.

Kuvassa 1 näkyvä puurakenne on kolmen korkuinen. Puun korkeus katsotaan pisimmästä matkasta kantasolmusta lehteen, joka on solmu, jolla ei ole lapsisolmuja. Puun kantasolmu on puun huipulla, ja sen lapsisolmut muodostavat seuraavan tason. Kolmannesta tasosta eteenpäin tasot voivat koostua usean solmun lapsisolmuista. Kuvan puu on binääripuu, sillä sen jokaisella solmulla on korkeintaan kaksi lapsisolmua. Solun syvyys puussa lasketaan siitä, kuinka pitkä matka solmusta on kantasolmuun.



Kuva 1. Esimerkki binääripuurakenteesta [20].

Peliprojektin dataa voidaan käsitellä puurakenteena, sillä se on jaoteltu otsikoiden alle. Käytössä on myös listoja ja taulukoita, joilla on lapsia, joilla voi myös olla lapsia ja niin edelleen. Syvyyttä ei peliprojektissa ole rajoitettu, mutta käytännön syistä se harvoin ylittää syvyyttä kolme. Tietojenkäsittelyssä taulukot siis täyttävät puurakenteen ehdot.

2.3 Attribuutit eli lisämääreet

Insinöörityössä määriteltiin oma attribuutti eli lisämääre helpottamaan editorin toimintaa. Lisämääreet ovat tapa lisätä tietoa käännökseen, luokkaan, funktioon, delegaattiin eli tyyppiin, joka voi esittää funktioita, joilla on samat parametrityypit, enumiin eli nimettyyn kokonaislukulistaan, tapahtumaan, kenttään, käyttäjäliittymään, ominaisuuteen tai struct-rakenteeseen. Lisämääre on olio, joka liitetään johonkin edellä mainituista metadatana, ja reflektion avulla niitä voidaan käsitellä ohjelmassa.

Lisämääreet ovat joko niin sanotusti sisäsyntyisiä (engl. intrinsic) tai mukautettuja (engl. custom). Sisäsyntyinen lisämääre tarkoittaa lisämäärettä, joka on osa C#-ohjelmointikielen .NET-viitekehystä. Käyttäjä voi luoda omia lisämääreitä kirjoittamalla oman luokan, joka periytyy System.Attribute-luokasta. Insinöörityössä luotiin mukautettu lisämääre vastaamaan peliprojektin tarpeita. [14.]

Unity-pelimoottorin omat lisämääreet voidaan luokitella mukautetuiksi lisämääreiksi, sillä ne eivät ole osa .NET-viitekehystä, vaan ne ovat Unity-pelimoottorille ominaisia lisämääreitä. Mukautettuja lisämääreitä käytetään samalla tavalla kuin sisäsyntyisiäkin. .NET-viitekehyksen lisämääreet toimivat Unity-kehitysympäristössä, mutta ne eivät ole insinöörityölle hyödyllisiä.

Esimerkkikoodissa 1 määritellään lisämääre, jota käytettiin insinöörityössä. Se on HeaderAttribute-niminen luokka, joka periytyy System.Attribute-luokasta. System.Attribute on pohjaluokka, josta mukautettujen lisämääreiden tulee periytyä C#-ohjelmointikielessä. HeaderAttribute-luokalla on yksityinen merkkijonokenttä nimeltä ”header” eli otsikko ja muodostin, jolle annetaan parametrina merkkijono, jonka arvo asetetaan luokkailmentymän otsikon arvoksi. Header-kentälle on määritelty julkinen lambda-funktio nimeltä ”Header”. Lambda-funktio on yksi tapa määritellä funktio perinteisen tyylin sijaan [21]. Se tarkistaa ensin, onko yksityiselle otsikkokentälle annettu tyhjä tai null, eli määrittelemätön, arvo. Mikäli otsikon arvo on tyhjä tai määrittelemätön, se palauttaa arvon ”No header”, ja mikäli otsikkokentälle on määritelty jokin arvo, se palauttaa kyseisen arvon. Lambda-funktio tässä tapauksessa on vain korvike ominaisuudelle, joka sisältää ”get”- eli ”hae”-määreen. Lambda-funktio tai ominaisuus antaa mahdollisuuden palauttaa jotakin itse määriteltyä virheen sijaan, mikäli otsikkoa ei ole määritelty.

```
public class HeaderAttribute : System.Attribute
{
      private string header;
      public string Header => string.IsNullOrEmpty(this.header) ?
            “No header” :
            this.header;
      public HeaderAttribute(string header) {
            this.header = header;
      }
}
```

Esimerkkikoodi 1. Mukautetun lisämääreen määrittely.

Esimerkkikoodissa 2 näkyy kaksi kenttää: kokonaisluku ”intValue” ja merkkijono ”stringValue”. Lisämääreet annetaan hakasulkeissa ennen kentän määrittelyä. Kokonaisluvulle on annettu itse luotu lisämääre HeaderAttribute, joka saa parametrina merkkijonon ”Integer Field”. Lisämääre on annettu ainoastaan kokonaisluvulle, ja sen alla oleva merkkijono ei saa lisämäärettä. Tästä nähdään, että lisämääreet vaikuttavat vain sitä seuraavaan yhteen kokonaisuuteen, minkä vuoksi niitä voidaan käyttää aihealueiden rajaamiseen.

```
 [HeaderAttribute(“Integer Field”)] public int intValue;
public string stringValue;
```

Esimerkkikoodi 2. Esimerkki lisämääreen käytöstä.

Lisämääreitä voidaan lukea reflektiolla, ja insinöörityössä niiden avulla aloitettiin uusi aihealue. Unity-pelimoottorilla on useampia omia attribuutteja, joista lähimpänä SettingsHeader-attribuuttia on Header-attribuutti. Molemmat ottavat yhden string-tyypin parametrin otsikon arvoksi. Insinöörityössä tehtiin näennäinen kopio Header-attribuutista esimerkkikoodin 1 mukaan, koska Unity-pelimoottorin Editor-luokan PropertyField, jota insinöörityössä käytettiin, piirtää Header-attribuutin inspektoriin oletusarvona. Se ei ole toivottua toiminnallisuutta, ja SettingsHeader-attribuutilla kierretään tämä.

2.4 Reflektio eli heijastaminen

Reflektio on C#-ohjelmointikielen sisäinen tapa käsitellä dataa geneerisesti. Reflektio antaa ohjelmoijan päästä käsiksi luokkien sisältämiin kenttiin ja funktioihin ilman, että ohjelmoijalla on suoraan käsiteltävänä luokan ilmentymää. Insinöörityössä luokka, jota

käsiteltiin, on tyyppiä Settings, mutta luokan sisällä voi olla muita luokkia, joiden kenttiin halutaan päästä käsiksi. [22]

Esimerkkikoodi 3 palauttaa Settings-luokan kaikki kentät FieldInfo-tyypin taulukkoon. Kentät voivat olla primitiivisiä tai geneerisiä. FieldInfo sisältää tiedon muun muassa kentän tyypistä, nimestä, attribuuteista, näkyvyydestä ja arvosta. GetFields-funktio ei palauta kenttiä missään tietyssä järjestyksessä.

```
System.Reflection.FieldInfo[] fields = typeof(Settings).GetFields();
```

Esimerkkikoodi 3. Esimerkki reflektion käytöstä.

Käytännössä reflektio on tapa käsitellä luokkien dataa ilman varsinaista luokkailmentymää. Reflektion avulla on kuitenkin mahdollista käsitellä luokkailmentymän dataa esimerkiksi GetValue-funktion avulla, joka etsii kentän arvon tietystä ilmentymästä. Tämä on hyödyllistä, kun ei suoraan tiedetä tai välitetä, minkälaista dataa ollaan käsittelemässä.

Insinöörityössä reflektiota hyödynnetään puurakenteen luomisessa, jossa ei tiedetä, minkälaisia kenttiä on vastassa, vaan halutaan geneerisesti käydä ne läpi ja tallentaa ne myöhempään käyttöön. Reflektiolla voidaan välttää tarkastelemasta, minkälainen kenttä on kyseessä, koska ohjelman suorituksen kyseisessä vaiheessa tieto ei ole olennainen. Esimerkiksi lisämääreitä etsiessä ainoa kiinnostava tieto on, onko kentällä lisämääre, eikä se, minkätyyppinen tai -niminen kenttä on kyseessä.

3 Insinöörityön puunäkymän määrittely

Settings-luokka määriteltiin insinöörityössä Settings.cs-nimisessä tiedostossa, ja siitä on tehty ScriptableObject-tiedosto, joka sisältää peliprojektin asetukset. Settings-luokassa määriteltiin erilaisia globaaleja asetuksia mobiilipelille, kuten ajoituksiin ja etäisyyksiin liittyviä liukulukuja ja muita primitiivisiä tyyppejä, kuten kokonaislukuja ja merkkijonoja, sekä Unity-pelimoottorille ominaisia luokkia, kuten animaatiokäyriä ja animaatioita. Koodissa pyritään olemaan viittaamatta tiettyihin arvoihin (ns. ”hard coding”), eli sen sijaan, että koodissa sanotaan etäisyyden olevan kaksi yksikköä, koodissa viitataan Settings-

luokan kenttään ”etäisyys”, jonka arvoksi on määritelty 2. Tämä mahdollistaa ajonaikaiset muutokset sekä sen, että jos halutaan muokata arvoja, ei tarvitse etsiä koodista riviä, jossa arvo on määritelty, vaan arvoa voidaan muokata suoraan tiedostosta. Koska Settings-luokasta on tehty serialisoitava ScriptableObject-tiedosto, siihen tehdyt muutokset eivät aiheuta ohjelman kääntämistä, toisin kuin koodiin tehdyt muutokset. Hyviin ohjelmointitapoihin kuuluu ”kovakoodaamisen” välttäminen.

Insinöörityötä varten ehdotettiin käyttämään Unity-pelimoottorin tarjoamaa TreeView-luokkaa. Se on rajapinta, jolla voidaan luoda puunäkymiä laajentamaan Unity-näkymää. TreeView-luokka tarjoaa oletuksena monia työkaluja, jotka ovat hyödyllisiä insinöörityön kannalta ja jotka helpottavat työskentelyä, joten työ päätettiin tehdä sen avulla.

Kuvasta 2 huomataan, että Unity-pelimoottorin oletusnäkymää on vaikea lukea ilman minkäänlaista jaottelua. Oletusnäkymään on mahdollista lisätä otsikoita ja jakajia selkeyttämään näkymää, mutta se ei riitä peliprojektin datamäärää jaottelemaan. Lista on yksinkertaisesti liian pitkä ja vie liikaa tilaa. Kuvan oikeassa laidassa näkyy vieritin, josta näkee, että kuvassa näkyy murto-osa koko tiedostosta.

Kuva 2. Unity-pelimoottorin tarjoama oletusnäkymä ScriptableObject-tiedostoille.

Oletusnäkymän hyviä puolia on, että se on niin sanotusti ajateltu loppuun. Unity-kehittäjät ovat miettineet, mikä olisi paras yleiseen käyttöön sopiva näkymä, ja toteuttaneet sen hyvin. Se ei kuitenkaan riitä peliprojektin käyttötarkoitukseen, vaan insinöörityössä sitä lähdettiin parantamaan. Tämän vuoksi Unity-pelimoottori tarjoaa joukon IMGUI-kontrolleja, joilla voi lähteä tekemään juuri itselle sopivia näkymiä.

Insinöörityön vaatimukset olivat, että peliprojektin ScriptableObject-tiedostoa olisi helpompi lukea ja muokata. Siihen pyydettyjä ominaisuuksia olivat hakukenttä, jaottelu aihealueisiin ja aihealueiden avaaminen ja sulkeminen. Näiden lisäksi puunäkymään lisättiin aakkosten mukaan järjestely ja yleisiä parannuksia ulkoasuun. Ehdotettu lähestymistapa oli Unity-pelimoottorissa määritelty TreeView-luokka.

3.1 Unity-pelimoottorin TreeView-luokka

Insinöörityön puurakenne toteutettiin TreeView-luokkalla. TreeView-luokka on IMGUI-kontrolli, jota käytetään hierarkkisen datan näyttämiseen, jonka elementtejä voidaan avata (engl. expand) ja sulkea (engl. collapse). TreeView-luokan käyttöä suositellaan muokattavien listojen näkymien ja usean sarakkeen näkymien luomisessa, joita voidaan hyödyntää muiden IMGUI-kontrollien ja komponenttien, kuten nappien ja Unity-pelimoottorin PropertyField-kontrollin ohella.

TreeView-luokan näkymä koostuu TreeViewItem-luokan olioista, jotka sijoitellaan riveille. Puun jokaisella oliolla on vanhempi ja valinnainen lista lapsiolioita. Oliot, joilla on lapsia, voidaan avata tai sulkea. Puunäkymä hyväksyy sekä hiiren että näppäimistön kautta tulleen syötteen. Vaikka kyseessä on puunäkymä, sen ei välttämättä tarvitse esittää puhdasta puutietorakennetta, vaan esimerkiksi Unity-pelimoottorin omat puurakenteet kuten pelinäkymän hierarkia ovat sopivia. [23; 24.]

Peliprojektissa käytetyn datan rakenteen vuoksi puunäkymä sopii siihen erinomaisesti. Data on pitkälti yksittäisiä kenttiä isompien otsikoiden alla, mutta siinä on myös paljon listoja ja geneerisiä luokkia, joiden tapauksissa puu saa syvyyttä. Mikäli puun syvyys olisi aina enintään 1, puunäkymä ei välttämättä olisi paras vaihtoehto peliprojektiin, vaan paremmiksi vaihtoehdoiksi nousisivat vaikkapa omiin ikkunoihin tai välilehtiin jaotellut aihealueet.

3.2 Editoriohjelmointi Unity-pelimoottorissa

IMGUI, jota TreeView-luokka käyttää, tulee englannin kielen sanoista “immediate mode graphical user interface” eli suoraan käännettynä “välittömän moodin graafinen käyttöliittymä”. Graafisen käyttöliittymän yleisessä käytössä oleva lyhenne on GUI eli “graphical user interface”. GUI:lla tarkoitetaan esimerkiksi valikkoja, ikkunoita ja kuvakkeita, joita piirretään tietokoneen näytölle tekemään käyttäjäkokemuksesta selkeämpää. [25.]

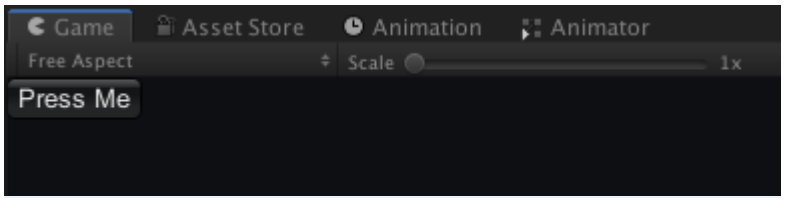
IMGUI viittaa käyttöliittymään, jonka tapahtumien käsittely on suoraan ohjelmoijan hallinnassa. Unity-pelimoottorissa editoriohjelmointia tehdään IMGUI:ta käyttäen. Siinä esimerkiksi nappula luodaan koodissa, ja se mitä nappulan painamisesta tapahtuu, määritellään myös koodissa. Unity-pelimoottorin GUILayout- ja EditorGUILayout-luokat toimivat IMGUI-periaatteella.

Esimerkkikoodista 4 nähdään, että Unity-pelimoottorin IMGUI-funktioita tulee kutsua OnGUI-funktion sisällä. Unity-pelimoottorissa MonoBehaviour-luokka implementoi OnGUI-funktion ja kaikki Unity-pelimoottorin omat luokat periytyvät siitä. Esimerkissä luodaan nappi GUILayout-luokan Button-funktiolla, joka ottaa parametrina tekstin, joka näkyy napin päällä. Se palauttaa boolean-arvon tosi, mikäli nappia painetaan, ja ohjelma tulostaa lokiin tekstin “Hello!”.

```
void OnGUI()
{
      if (GUILayout.Button("Press Me"))
            {
                  Debug.Log("Hello!");
            }
}
```

Esimerkkikoodi 4. Napin luominen IMGUI-periaatteella.

Kuva 3 näyttää, mitä edeltävä IMGUI-koodiesimerkki näyttää pelinäkymässä, kun se on lisätty MonoBehaviour-luokasta periytyvän luokan koodiin, joka on lisätty peliolioon. GUILayout-luokan Button-funktio asettelee napin oletuksena vasempaan ylälaitaan. Nappia painamalla lokiin ilmestyy koodissa määritelty teksti “Hello!”. MonoBehaviour on Unity-pelimoottorissa määritelty kantaluokka, josta kaikki muut Unity-pelimoottorin luokat periytyvät [26]. Käyttäjä voi kuitenkin tehdä omia luokkiaan, jotka eivät periydy siitä.



Kuva 3. Koodiesimerkin 4 luoma nappi pelinäkymässä.

Unity-manuaali suosittelee, että IMGUI:ta käytetään Unity-sovellusta laajennettaessa, esimerkiksi peliprojektin puunäkymässä. IMGUI vaatii, että käyttäjä ylläpitää mahdollisten olioiden tilaa. Vertailukohde tähän olisi RMGUI eli “retained mode GUI”. Sillä viitataan oliopohjaiseen käyttöliittymään, jossa oliot ovat olemassa RAM-muistissa. Unity-pelimoottorissa voi tehdä käyttäjäliittymän luomalla sen komponenteista peliolioita pelinäkymään. Näin vältytään paitsi koodaustyöltä, mutta myös komponenttien asettelu ja niiden tilan ylläpito helpottuu. Nappeihin voi määritellä Unity-pelimoottorin inspektorissa, mitä funktiota kutsutaan, kun nappia painetaan. [25.]

4 Puunäkymän toteutus Unity-pelimoottorissa

Insinöörityö toteutettiin SettingsEditorWindow- ja SettingsTreeView-luokissa. SettingsEditorWindow-luokka, joka periytyy EditorWindow-luokasta, huolehtii siitä, miltä editori-ikkuna näyttää käyttäjälle, ja SettingsTreeView-luokka, joka periytyy TreeView-luokasta, huolehtii siitä, miten editorin sisältöä käsitellään ja mitä sen sisältö on. SettingsTreeView-luokka määrittelee myös puunäkymälle ominaista visuaalista toiminnallisuutta. Nämä voidaan jossain määrin ajatella visuaalisena ja loogisena puolena, jotka täydentävät toisiaan. SettingsEditorWindow-luokka ei pysty näyttämään mitään ilman SettingsTreeView-luokan dataa, ja SettingsTreeView-luokan data on käytännössä hyödytöntä, ellei SettingsEditorWindow-luokka näytä sitä.

4.1 Editori-ikkunan määrittely

SettingsEditorWindow-luokka periytyy Unity-pelimoottorin EditorWindow-luokasta, jota käyttämällä voi luoda omia ikkunoita Unity-näkymään. EditorWindow-luokassa on funktio ”GetWindow”, joka piirtää ikkunan ohjelmaan. Luokalla on OnGUI-funktio, jonka sisällä

määritellään, mitä ikkunassa näkyy. OnGUI:ta kutsutaan joka kuvaruudulla (engl. frame), joten sen sisällä ei kannata suorittaa raskaita operaatioita, kuten uusien listojen luomista, mikä varaa muistia. SettingsEditowindow-luokan OnGUI:ssa kutsutaan InitIfNeeded-funktiota eli ”alusta jos tarpeen”. Se suoritetaan vain, jos puunäkymää ei ole alustettu ja tietoa ylläpidetään boolean-muuttujassa. Käytännössä alustus tehdään aina, kun ohjelma käännetään. Unity-pelimoottori kääntää ohjelman, kun siihen on tehty muutoksia ja käyttäjä palaa Unity-ohjelmaan.

Esimerkkikoodissa 5 näkyy, miten EditorWindow-luokasta periytyvien luokkien ikkuna saadaan näkyviin Unity-ohjelmassa. MenuItem-lisämääreellä voidaan merkitä funktioita, joille halutaan lisätä Unity-näkymän valikkoihin nappi, jota painamalla funktiota kutsutaan. Lisämääre ottaa parametrina polun, josta nappi löytyy, tiedon siitä, onko kyseessä validointi, ja tärkeysjärjestysnumeron. Kaksi viimeistä parametria ovat valinnaisia. Monen Editor-luokan johonkin funktioon on suositeltavaa lisätä MenuItem-attribuutti, jotta luokassa kirjoitetun ikkunan saa auki Unity-näkymästä. Insinöörityössä tämä funktio on EditorWindow-luokasta periytyvä GetWindow-funktio, joka avaa ikkunan Unity-näkymään. [27.]

```
[MenuItem(“Window/Settings Editor Window”)]
public static void ShowWindow()
{
      GetWindow(typeOf(SettingsEditorWindow)).titleContent =
            new GuiContent(“Settings Editor”);
}
```

Esimerkkikoodi 5. SettingsEditorWindow-luokassa määritelty funktio ikkunan avaamiselle.

SettingsEditorWindow-luokan OnGUI-funktiossa kutsutaan SettingsTreeView-luokan OnGUI-funktiota. SettingsTreeView-luokka määrittelee itse, miten se piirretään, ja SettingsEditorWindow-luokka kutsuu sitä tietämättä, mitä funktion sisällä tehdään. Unity-pelimoottorin GUILayout-luokka ja rect-tietorakenne huolehtivat, ettei elementtejä piirretä päällekkäin. Rect tulee englannin kielen sanasta ”rectangle” eli suorakulmio ja kuvaa tietorakennetta, joka määrittelee suorakulmion sijainnin ja koon. Editor-ohjelmoinnissa sillä määritellään alueita, jolle elementit kuten napit asetetaan. GUILayout-luokkaa käytettäessä ohjelmoija on itse vastuussa siitä, että rect-kentät on määritelty oikein, mutta TreeView-luokka tarjoaa automaattisesti määritellyt rect-arvot. EditorGUILayout-luokkaa voi käyttää GUILayout-luokan sijaan, ja se huolehtii automaattisesti elementtien asettelusta.

OnGUI-funktiossa määritellään myös painike puunäkymän jokaisen alkion avaamiselle, jotta jokainen alkio saadaan näkyviin, ja jokaisen alkion sulkemiselle, jotta vain otsikot näkyvät. Joskus on tarpeen nähdä kaikki kentät, mikäli käyttäjä haluaa vain selata, mitä on tarjolla, ja joskus on hyödyllistä lähteä otsikoista, jos käyttäjällä on jonkin tietyn aihealueen alkio mielessä. Yleensä nämä napit ovat turhia, sillä käyttäjä usein tietää suurin piirtein, mitä on etsimässä, ja voi hyödyntää hakukenttää sen löytämiseen. OnGUI-funktiossa on myös järjestyksen alkutilaan asettamiselle painike, joka lataa puunäkymän uudestaan ja palauttaa sen järjestykseen, jossa Settings-luokka on alkiot määritellyt.

SettingsEditorWindow-luokan alustusfunktiossa luodaan uusi TreeViewState-luokan ilmentymä, joka sisältää puun tilan, mikäli sellaista ei löydy. Tila säilyttää tiedon puurakenteen tilasta, ja sen sisältämä arvo on serialisoitu, jotta sen arvo ei katoa, kun ohjelma käännetään. Tila sisältää tiedon siitä, mitkä alkiot puurakenteessa on avattu, mitä hakukentässä on, mitkä alkiot ovat valittuna ja missä kohdassa vieritin on. Mikäli tilaa ei olisi serialisoitu, puunäkymä palautettaisiin alkutilaan aina, kun ohjelmassa tapahtuu muutos. Alustusfunktiossa luodaan myös uusi MultiColumnHeader-luokan ilmentymä, jota TreeView-luokan muodostin tarvitsee, mikäli sarakkeita on hyödynnetty. Se ottaa parametrina tilan, jossa määritellään, mitä sarakkeita sen tulee sisältää ja miten sarakkeet on järjestetty. TreeView-luokka ylläpitää MultiColumnHeaderState-tilaa, joka säilyttää puun järjestelyyn liittyvää dataa ja jonka voisi serialisoida, mutta sitä ei koettu tarpeelliseksi insinöörityön tässä vaiheessa. [23; 24.]

Alustusfunktiossa luodaan hakukenttä-luokasta ilmentymä, joka ottaa käyttäjän syötteen ikkunasta ja antaa ohjelmoijan käyttää syötettä koodissa. TreeView-luokka implementoi hakufunktion, joka seuraa hakukentän syötettä ja palauttaa alkiot, joiden nimi vastaa syötettä. Hakukenttä-luokalla on delegaattifunktio, joka seuraa, onko käyttäjä painanut jotakin nuolinäppäintä. Alustusfunktiossa delegaatille lisätään seurain, joka vahvistaa valinnan, aina kun nuolinäppäintä painetaan. Seurain poistetaan EditorWindow-luokan OnDestroy-funktiossa, jota kutsutaan, kun ikkuna suljetaan. Seuraimia ei tule olla enemmän kuin on tarpeen, ja usein yksi riittää.

Alustusfunktion viimeisessä kohdassa seurataan Undo-luokan ”undo redo performed”-delegaattia omalla funktiolla. Undo-luokka tallentaa olion tilan ja mahdollistaa muutosten kumoamisen tai uudelleen toistamisen. Undo-luokka itsessään hoitaa olion

arvomuutokset, ja funktio, jolla tapahtumaa seurataan, lataa puunäkymän uudestaan, jotta kumotut tai uudelleen tehdyt muutokset näkyvät käyttäjälle. Alustusfunktion lopussa boolean-muuttuja ”isInitialized” asetetaan todeksi, jotta alustus suoritetaan vain, jos se on tarpeen. Muuttujaa tarkkaillaan OnGUI-funktion alussa, ja se estää ohjelmaa suorittamasta alustusta enemmän kuin kerran ohjelman kääntämisen jälkeen, jotta alustusta ei tapahdu joka kuvaruudussa.

4.2 Alkioiden erottelu Settings-tiedostosta

SettingsTreeView-luokka periytyy TreeView-luoksta. Sen muodostin ottaa parametrina TreeViewState-luokan ilmentymän ja MultiColumnHeader-luokan ilmentymän. Molemmat ovat osa Unity-pelimoottorin IMGUI-kontolleja ja periytyvät siitä. MultiColumnHeader on valinnainen parametri, joka määrittelee puulle sarakkeet ja sisältää tietoa esimerkiksi siitä, miten puu on järjestetty ja mitkä sarakkeet voidaan järjestää. Insinöörityössä käytetään kolmea saraketta, jotka ovat nimi, arvokenttä ja tyyppi. [23; 24.]

Omassa TreeView-luokan implementaatiossa täytyy olla määritelty BuildRoot-funktio, joka on absrakti funktio TreeView-luokassa. Abstrakteilla funktioilla ei ole implementaatiota luokassa, jossa ne on määritelty, vaan ne täytyy implementoida aliluokissa [28]. Funktion sisällä luodaan lista TreeviewItem-luokan ilmentymiä, jotka funktio palauttaa.

Kuvassa 4 luodaan oma implementaatio TreeViewItem-luokasta nimeltä MyTreeViewItem, joka sisältää oletusdatan lisäksi insinöörityölle ominaisia kenttiä ja apufunktioita. TreeViewItem-luokan ilmentymä sisältää kentät nimelle, listalle lapsiolioita, syvyydelle puussa, mahdolliselle kuvalle, tunnisteelle ja vanhemmalleen. Näiden lisäksi MyTreeViewItem-luokalle määritellään polku kenttään, johon se viittaa ja jonka avulla sen SerializedProperty-ilmentymä löydetään, tieto siitä, onko se listan tai taulukon aloittava kenttä tai onko kenttä viittaus geneeriseen luokkaan, joka sisältää muita tyyppejä. Sille on myös määritelty apufunktio sille, onko alkio listan, taulukon tai geneerisen luokan elementti, ja toinen apufunktio, joka palauttaa alkion indeksin listassa tai taulukossa, mikäli se on osa sellaista.

```
class MyTreeViewItem : TreeViewItem
{
    public string propertyPath;
    // Flag for custom classes that don't fit into one row and need a "holder" for their fields.
    public bool isMultiRowType;

    public bool IsArrayElement =>
        this.displayName.StartsWith( value: "Element ");
    public int IndexInArray =>
        this.IsArrayElement ?
            int.Parse(this.propertyPath.Split( params separator: '[').Last().Split( params separator: ']')[0]) :
            -1;
}
```

Kuva 4. TreeViewItem-luokasta periytyvän rakenteen määrittely.

Insinöörityön BuildRoot-funktio lukee Settings-luokasta tehdyn SerializedObject-ilmentymän kentät ja tallentaa ne listaan MyTreeViewItem-luokan ilmentyminä. SerializedObject-ilmentymän kentät voi lukea kutsumalla GetIterator-funktiota, joka palauttaa ilmentymän ensimmäisen kentän SerializedPropertynä. SerializedObject-ilmentymän ensimmäinen kenttä on viittaus itseensä ja sen yli tulee hypätä kutsumalla sen Next-funktiota. SerializedPropertyn Next- ja NextVisible-funktioille annetaan parametrina tieto siitä, halutaanko ominaisuuden lapset käydä läpi. Implementaatiossa lapsien yli hypätään ja ne käydään manuaalisesti läpi.

Esimerkkikoodi 6 näyttää apufunktion, jonka avulla voidaan käydä SerializedObject-ilmentymän kentät läpi. Sen tyyppi on IEnumerable<SerializesProperty>, eli funktiota voidaan käyttää foreach-kiersiössä, jossa se palauttaa SerializedProperty-olioita. Se ottaa parametrina SerializedObject-ilmentymän ja tiedon siitä, käydäänkö siitä löytyneiden kenttien lapset läpi. SerializedObject-tyypin parametrille on annettu “this”-määre, joka kertoo, että kyseessä on jatkefunktio (engl. extension function) SerializedObject-tyypin ilmentymille.

```
public static IEnumerable<SerializedProperty> EnumerateSerializedObject(this
SerializedObject source, bool enterChildren)
{
   SerializedProperty iterator = source.GetIterator();
   iterator.Next(true);
   while (iterator.NextVisible(enterChildren))
   {
       yield return iterator;
   }
}
```

Esimerkkikoodi 6. Apufunktio SerializedObject-ilmentymän läpikäymiseen.

Kuvassa 5 määritellään BuildRoot-funktion toteutus. Siinä ylläpidetään kokonaislukua ”id” eli tunniste, jota käytetään TreeViewItem-ilmentymien luomisessa. Jokaisella ilmentymällä tulee olla uniikki tunniste. Funktiossa luodaan kantasolmu, jonka syvyys on negatiivinen, eli se ei tule näkymään puunäkymässä, mutta puurakenteiden luonteen vuoksi se tarvitsee solmun, josta koko puu alkaa. Sen jälkeen funktiossa luodaan uusi lista, joka tulee sisältämään kaikki puun elementit. Foreach-kiersiössä käydään läpi kaikki kentät, jotka löytyvät SerializedObject-ilmentymästä. Siihen käytetään esimerkkikoodin 6 esittelemää jatkefunktiota. Löydetyistä SerializedProperty-ilmentymistä yritetään etsiä SettingsHeader-tyypin lisämääre reflektiolla. GetField-funktio palauttaa polun mukaisen FieldInfo-tyypin kentän, mikäli sellainen löytyi, ja sille voidaan kutsua GetCustomAttribute-funktiota, jolle annetaan lisämääreen tyyppi kulmasulkeissa ja joka palauttaa sentyyppisen lisämääreen. Mikäli lisämääre löytyi, sen sisältämän header-kentän tekstiä käytetään yläotsikon luomiseen. Sen jälkeen SerializedProperty-ilmentymä käydään rekursiivisesti läpi GoThroughProperty-funktiossa, jotta sieltä voidaan erotella muun muassa taulukoiden elementit omiksi alkioikseen.

```
int id = 0;
TreeViewItem root = new TreeViewItem {id = id, depth = -1, displayName = "Root"};
id++;

List<TreeViewItem> treeViewItems = new List<TreeViewItem>();

foreach (SerializedProperty serializedProperty in SerializedSettings.Instance.EnumerateSerializedObject( enterChildren: false))
{
    SettingsHeader header = typeof(Settings).GetField(serializedProperty.propertyPath)?.GetCustomAttribute<SettingsHeader>();

    if (header != null)
        AddItemToTree( depth: 0, header.header,  prop: null,  isMultiRowType: false);

    GoThroughProperty(serializedProperty);
}

SetupParentsAndChildrenFromDepths(root, treeViewItems);

return root;
```

Kuva 5. BuildRoot-funktion toteutus.

GoThroughProperty-funktion tarkoitus on käydä SerializedProperty-ilmentymää läpi, kunnes se löytää lehden eli kentän, jolla ei ole lapsia. Jokainen läpi käyty alkio lisätään TreeViewItem-listaan joko yläotsikkona tai varsinaisena datakenttänä, joista jokainen lisätään listaan MyTreeViewItem-ilmentymänä. Listat, taulukot ja geneeriset luokat voidaan myös lisätä sellaisenaan TreeViewItem-listaan yhtenä alkiona, joka sisältää useita kenttiä. Siinä tapauksessa alkio näkyisi yhtenä usean rivin alkiona puunäkymässä ja ainoastaan listan nimeä voisi etsiä. Haluttiin, että niiden kentät ovat omia alkioitaan ja näkyvät puussa omina kenttinään, jotta niiden jäseniä voi etsiä hakukentällä ja ne voisi

avata ja sulkea kutsumalla niiden TreeViewItem-ilmentymälle Expand- ja Collapse-funktioita.

GoThroughProperty-funktion sisällä käytetään reflektiota, kun luetaan geneeristen eli ohjelman omien luokkien kenttiä. Jotta reflektiota voidaan käyttää, tulee ensin selvittää geneerisen ilmentymän tyyppi, jota käännös osaa lukea. Geneerisen luokan SerializedProperty-ilmentymä tietää vain tyyppinsä nimen merkkijonona eikä itse tyyppiä. Jotta todellinen tyyppi saadaan selville, ohjelmassa käytetään apufunktiota, johon on kovakoodattu merkkijono, jonka System.Type.GetType-funktio hyväksyy parametrina.

Geneerisen luokan todellisen tyypin löytävä merkkijono saadaan selville esimerkkikoodin 7 mukaan. Siinä haetaan MyClass-luokan tyyppi merkkijonona, joka kelpaa käännökselle. Merkkijono sisältää tekstin "MyProject.Settings+MyClass, Assembly-CSharp, Version=0.0.0.0, Culture=neutral, PublicKeyToken=null". Sen sisällä on kaksi kohtaa, joita muuttamalla voidaan löytää peliprojektin kaikki luokat. Ensimmäinen niistä on tyypin yläluokka Settings, jonka sisällä MyClass-luokka määritellään. Toinen on SerializedPropertyn tuntema merkkijono luokkansa tyypistä, tässä tapauksessa ”MyClass”. ”Settings”- ja ”MyClass”-kenttiä muuttamalla GetType-funktio löytää minkä vain geneerisen luokan oikean tyypin, jota voidaan hyödyntää myöhemmin. ”MyProject” on peliprojektin nimiavaruus (engl. namespace).

```
string assemblyName = typeof(MyProject.Settings.MyClass).AssemblyQuali-
fiedName;
```

Esimerkkikoodi 7. Geneerisen luokan MyClass todellisen tyypin haku.

Esimerkkikoodi 8 näyttää, miten merkkijonosta saadaan selville luokka. Dollarisymboli merkitsee, että sitä seuraavissa lanausmerkeissä olevassa tekstissä voi olla aaltosulkujen sisällä viittauksia muuttujiin. Insinöörityössä koodin mukaista pätkää kutsutaan LINQ-operaation sisällä, joka käy läpi luokassa määritellyt mahdolliset yläluokat kullekin tarkastellulle luokalle, jotka on tallennettu listaan. Listassa on insinöörityössä arvot ”Settings” ja ”Utils”. Mikäli LINQ-operaatio ei löydä tyyppiä listasta, se palauttaa määrittelemättömän tyypin, koska funktio ei enää tiedä, mistä etsiä.

```
Type myType = Type.GetType($”MyProject.{classParent}+{typeAsString}, Assembly-
CSharp, Version=0.0.0.0, Culture=neutral, PublicKeyToken=null”);
```

Esimerkkikoodi 8. Todellisen tyypin selvittäminen merkkijonosta.

Kun SerializedProperty-ilmentymän todellinen tyyppi on saatu selville, sen kentät haetaan reflektiolla ja GetFields-funktiolla, joka palauttaa taulukon FieldInfo-ilmentymiä. FieldInfo-ilmentymä sisältää tiedon kentän nimestä, ja sen avulla voidaan hakea sen SerializedProperty-ilmentymän lapsikentät kutsumalla Settings-luokan SerializedObject-ilmentymälle FindProperty-funktiota liittäen FieldInfo-ilmentymän nimen polun loppuun. Löydetty kenttä voi olla uusi taulukko tai geneerinen luokka, joten sille kutsutaan uudestaan GoThroughProperty-funktiota.

Kun lista TreeViewItem-ilmentymiä on koottu GoThroughProperty-funktion kautta, TreeView-luokka muuntaa ne puunäkymälle sopivaksi listaksi syvyyden perusteella käyttäen TreeView-luokan valmista SetupParentsAndChildrenFromDepths-funktiota, eli ”aseta vanhemmat ja lapset syvyyden perusteella”. Se ottaa parametrina juurisolmun ja BuildRoot-funktiossa luodun listan TreeViewItem-ilmentymiä. Se lukee elementtien syvyydet ja päättelee niiden järjestyksestä, miten puu tulee rakentaa.

4.3 Puunäkymän solujen piirtäminen

Unity-pelimoottorissa ikkunoita tehdessä käytetään EditorWindow-luokan OnGUI-nimistä funktiota, joka yliajetaan kussakin implementaatiossa. Sitä kutsutaan joka kuvaruudulla, ja sen tehtävä on piirtää ohjelmoijan määrittelemät graafiset kentät ikkunaan. Ne voivat olla esimerkiksi nappeja, tekstikenttiä tai erilaisia valikkoja. Kaikki mainitut OnGUI-funktiot periytyvät MonoBehaviour-luokasta, joka on kantaluokka insinöörityössä käytetyille luokille.

SettingsEditorWindow-luokan OnGUI-funktiossa kutsutaan TreeView-luokan ilmentymän OnGUI-funktiota, jonka sisällä kutsutaan virtuaalifunktiota RowGUI, joka piirtää yhden solun. Virtuaalinen RowGUI-funktio on yliajettu SettingsTreeView-luokan implementaatiossa. TreeView-luokan OnGUI huolehtii sarakkeiden otsikoinnista, reunoista ja raidoitetusta taustasta, mikäli ne on määritelty. Ohjelmoijan vastuulle jää ainoastaan määritellä, miten TreeViewItem-luokan alkio näkyy ikkunassa.

Insinöörityön RowGUI-funktiossa alkio erotellaan pääosin sen perusteella, löytyykö sille SerializedProperty-tyypin vastinetta Settings-luokasta. Alkiot on määritelty niin, että yläotsikoille ei löydy vastinetta luokasta. Tämän tiedon pohjalta RowGUI-funktio on jaoteltu kahteen osaan. Mikäli SerializedProperty-alkio löytyi, sen sarakkeille kutsutaan CellGUI-funktiota. Poikkeus tähän on Settings-luokan määrittelevä “m_Script”-kenttä, joka on viittaus koodiin, jonka pohjalta Settings-luokan ScriptableObject-ilmentymä luotiin. Script-kenttä erotellaan muusta koodista siksi, että se haluttiin näyttää sen editorin huipulla ensimmäisenä alkiona eri tavalla kuin muut vastaavat kentät. Tämä tehtiin siksi, että näin koodiin on helpompi päästä käsiksi editorista.

CellGUI-funktio on apuväline siihen, miten RowGUI-funktio piirtää rivin. CellGUI-funktio ottaa parametrina Rect-tyypin arvon alueelle, jolle elementti piirretään, SerializedProperty-alkion, MyTreeViewItem-ilmentymän, sarakkeen ja Rect-tyypin “firstCellRect”-nimisen alueen. Viimeinen parametri annetaan, jotta nimisarakkeen alueella voidaan simuloida oletusinspektorin käyttäytymistä muun muassa liukulukujen kanssa. Oletusinspektori antaa käyttäjän muuttaa lukua painamalla kentän nimeä ja liu’uttamalla hiirtä sen päällä.

CellGUI-funktio voi tehdä kolme eri asiaa sen perusteella, mikä sarake sille on annettu parametrina. Nimi-sarakkeen tapauksessa CellGUI-funktio kutsuu EditorGUI-luokan LabelField-funktiota, joka piirtää tekstiä editoriin. CellGUI-funktio piirtää sen annetulle Rect-tyypin alueelle, joka on puunäkymän vasemmanpuoleisin sarake. Arvo-sarake on monimutkaisin tapaus, johon funktiota käytetään. Mikäli funktiolle syötetty SerializedProperty-alkio on lista tai taulukko ja sillä on näkyviä lapsiolioita, funktio piirtää sille kentän, joka näyttää sen lapsiolioiden määrän ja antaa muokata kyseistä numeroa. Tämä simuloi listojen ja taulukoiden oletusnäkymää, jossa niiden kokoa voidaan muokata suoraan inspektorista. Mikäli edellinen ehto ei täyttynyt ja MyTreeViewItem-ilmentymän “isMultiRowType”-kenttä on epätosi, sille kutsutaan DrawDefaultPropertyField-funktiota. DrawDefaultPropertyField-funktiossa määritellään hiirellä vedettävä alue arvojen muuttamiselle, mikäli kentällä ei ole lapsiolioita. Muussa tapauksessa sille piirretään EditorGUI-luokan PropertyField-funktiossa määritelty oletuskenttä.

Aina kun SerializedProperty-alkioon tehdään muutos, sen SerializedObject-ilmentymälle kutsutaan ApplyModifiedChanged-funktiota, joka asettaa arvot ilmentymälle ja ottaa ne

käyttöön [20]. Tämän jälkeen näkymä piirretään uudestaan, jotta muutokset näkyvät käyttäjälle. DrawDefaultPropertyField-funktiossa on tehty oma toiminnallisuus useamman kentän muokkaamiselle, joka käydään läpi arvojen muuttamisen yhteydessä. Siinä tarkastellaan, mitkä kentät ovat valittuina. Mikäli muokattu kenttä ei kuulu valittuihin kenttiin, muita kenttiä ei muokata, vaan ainoastaan muokatun kentän arvo muuttuu. Valinta tyhjennetään ja asetetaan muokattuun kenttään viestimään käyttäjälle, että useamman kentän muokkaus ei onnistunut. Mikäli muokattu kenttä kuuluu valintaan ja valittuja kenttiä on enemmän kuin yksi, valituista kentistä etsitään niitä vastaava SerializedProperty-alkio. Mikäli alkio on olemassa ja se on samaa tyyppiä kuin muokattu kenttä, muokatun kentän arvot kopioidaan siihen.

SerializedProperty-alkion arvojen kopioiminen toiseen SerializedProperty-alkioon ei ole kovin yksinkertaista, jos niiden tyyppi ei ole tiedossa. SerializedProperty-alkio sisältää kentät kaikille primitiivisille tyypeille, kuten kokonaisluvuille, sekä Unity-pelimoottorin kanssa työskentelylle olennaisille kentille, kuten olioviittauksille. Niiden arvoja ei ole määritelty, mikäli SerializedProperty-alkion sisältämän datan tyyppi ei sovi kentän tyyppiin, eli vain yksi arvokenttä on määritelty. SerializedProperty-alkion kokonaisuuden kopioiminen onnistuu käyttämällä sen Copy-funktiota, ja se soveltuu SerializedObject-ilmentymästä toiseen kopioimiselle. Insinöörityössä käyttötarkoitus on kuitenkin eri, sillä siinä on vain yksi SerializedObject-ilmentymä ja ainoa kenttä, joka SerializedProperty-alkiosta halutaan kopioida, on sen tietty arvokenttä. Valitettavasti insinöörityön puitteissa sille ei löytynyt helppoa tapaa, vaan jokaiselle 18:sta mahdollisesta tyypistä kirjoitettiin oma kopiointitoiminnallisuus. Poikkeus tähän on Utils-luokassa määritelty Range-rakenne. Sen kenttien kopioinnissa täytyy käyttää reflektiota, kutsumalla FindPropertyRelative-funktiota, joka ottaa parametrina kentän nimen, jota sillä etsitään. Range-tyypin SerializedProperty-alkion arvot kopioidaan vain, jos kyseessä ei ole taulukon aloittava kenttä.

4.4 TreeView-luokan virtuaalifunktiot

TreeView-luokka sisältää monia virtuaalifunktioita, joita ohjelmoija voi ottaa käyttöön tarpeen mukaan. Virtuaalifunktiolla tarkoitetaan funktiota, joka voidaan kirjoittaa uudestaan avainsanalla “override” luokassa, joka periytyy funktion omistavasta luokasta. Periytyvän luokan uudelleen määriteltyä virtuaalifunktiota kutsutaan virtuaalifunktion sijaan, kun

mahdollista. Tätä kutsutaan polymorfismiksi. Vaikka jossakin ohjelman osassa käsiteltäisiin TreeView-ilmentymää, käännös tietää sen todellisen tyypin olevan SettingsTreeView ja kutsuu SettingsTreeView-luokan määritelmää samasta funktiosta.

BuildRows-funktio, joka alustaa puun rakenteen, on pakko määritellä omassa TreeView-luokan implementaatiossa, ja loput ovat valinnanvaraisia. Insinöörityössä käyttöön otettuja funktioita ovat ContextClicked-, SearchChanged- ja DoesItemMatchSearch-funktiot. TreeView-luokka implementoi myös esimerkiksi puun jäsenten järjestyksen muuttamiseen liittyviä funktioita.

ContextClickedItem-funktiota kutsutaan aina, kun käyttäjä napauttaa jotakin kenttää hiiren oikealla painikkeella. Funktio ottaa parametrina TreeViewItem-ilmentymän id-kentän arvon, josta nähdään, mitä kenttää napautettiin. Insinöörityössä funktiota käytetään listojen toiminnallisuuden uudelleenkirjoittamiseen. Oletusnäkymässä listoja ja taulukoita pystyy laajentamaan ja supistamaan napauttamalla jotakin sen elementtiä hiiren oikealla painikkeella, jolloin ilmestyy vaihtoehto luoda kaksoiskopio elementistä tai poistaa se. SettingsTreeView-luokan GoThroughProperty-funktiossa kuitenkin rikottiin listojen ja taulukoiden kokonaisuudet ja niiden elementeistä tehtiin erillisiä puun alkioita.

ContextClickedItem-funktiossa ensin tarkastetaan, onko TreeViewItem-ilmentymä osa listaa tai taulukkoa. Funktio palaa tekemättä mitään, mikäli näin ei ole. Muussa tapauksessa kursorin kohdalle luodaan GenericMenu-luokan ilmentymä, jossa on vaihtoehdot elementin kaksoiskappaleen luomiselle ja elementin poistamiselle. GenericMenu-ilmentymän AddItem-funktio ottaa parametrina otsikon, merkinnän siitä, onko kenttä valittuna, ja toiminnon. Toiminnoksi voi antaa funktion, joka ei ota parametreja, tai sille voi määritellä lambda-funktion. AddItem-funktiolle annettu toiminnallisuus suoritetaan, kun valikon kyseistä kenttää napautetaan.

Kuvassa 6 näkyy koodipätkä, joka suoritetaan, kun jotakin listan elementtiä napautetaan hiiren oikealla painikkeella. Siinä luodaan uusi GenericMenu-luokan ilmentymä, jolle lisätään kaksi elementtiä otsikoilla “Duplicate Array Element” ja “Delete Array Element”. AddItem-funktion viimeinen parametri on funktio, ja kuvassa niiden nimet ovat OnDuplicate ja OnDelete, jotka vuorostaan kutsuvat HandleSelection-funktiota eri parametreilla.

GenericMenu-luokan ilmentymän ottamaa funktiota on rajattu siten, että se ei ota parametreja, joten se kierretään kuvan osoittamalla tavalla.

```
GenericMenu menu = new GenericMenu();
menu.AddItem(new GUIContent( text: "Duplicate Array Element"), on: false, OnDuplicate);
menu.AddItem(new GUIContent( text: "Delete Array Element"), on: false, OnDelete);
menu.ShowAsContext();

void OnDuplicate() { HandleSelection( duplicate: true); }
void OnDelete() { HandleSelection( duplicate: false); }
```

Kuva 6. Esimerkki GenericMenu-luokan ilmentymän luomisesta, jossa sille annetaan funktio parametrina.

Kuvan 7 koodipätkä tekee täysin saman asian kuin kuvan 6 koodi. Niiden ero on, että kuvassa 7 AddItem-funktiolle annetaan parametrina lambda-funktio. Labmda-funktio on ikään kuin käänteinen funktiokutsu. Sen alussa näkyvät sulkeet määrittelevät, mitä parametreja käsitellään. Tässä tapauksessa niissä ei ole mitään, sillä AddItem-funktion ottama toiminto ei ota parametreja. Sulkeita seuraava yhtäsuuruus- ja suurempi kuin -merkit osoittavat, että seuraava pätkä on lambda-funktion toiminnallisuus. Sillä ei ole varsinaisia rajoitteita, kuinka pitkän pätkän koodia lambda-funktioon voi laittaa, mutta luettavuuden kannalta pitkät koodipätkät kannattaa irrottaa omaan funktioonsa, kuten kuvassa 6 on tehty.

```
GenericMenu menu = new GenericMenu();
menu.AddItem(new GUIContent( text: "Duplicate Array Element"), on: false, func: () => HandleSelection( duplicate: true));
menu.AddItem(new GUIContent( text: "Delete Array Element"), on: false, func: () => HandleSelection( duplicate: false));
menu.ShowAsContext();
```

Kuva 7. Esimerkki GenericMenu-luokan ilmentymän luomisesta, jossa sille annetaan lambda-funktio parametrina.

HandleSelection-funktio implementoi listojen ja taulukoiden perustoiminnallisuuden puunäkymään, jossa niiden rakenne on rikottu, jotta niiden elementit olisivat yksittäisiä puunäkymän osia. Se lukee, mitkä puunäkymän alkiot ovat valittuina, ja suodattaa niistä ne, jotka ovat listan tai taulukon elementtejä käyttäen LINQ-operaatiota ja järjestää ne id-kentän arvon mukaan suuruusjärjestykseen. Operaatio palauttaa IOrderedNumerable<MyTreeViewItem> tyypin Enumerable-operaation nimeltä selectedItems eli valitut alkiot. Enumerable-tyyppisiä operaatioita voi käydä läpi foreach-kiersiössä, ja se palauttaa MyTreeViewItem-tyyppisiä alkioita.

Esimerkkikoodia 9 käytettiin löytämään listan tai taulukon elementin vanhempi. Koska koodissa käsitellään TreeViewItem-alkioita, jotka tietävät vanhempansa, niistä voidaan suoraan hakea sen vanhempi, ja koska insinöörityössä käytetään yksinomaan MyTreeViewItem-tyypin alkioita, sen tyyppi voidaan muuntaa MyTreeViewItem-tyypiksi ja sen propertyPath-muuttujan arvon voi lukea.

```
string parentPath = ((MyTreeViewItem)selectedItem.parent).propertyPath;
```

Esimerkkikoodi 9. Koodipätkä listan tai taulukon elementin vanhemman löytämiselle.

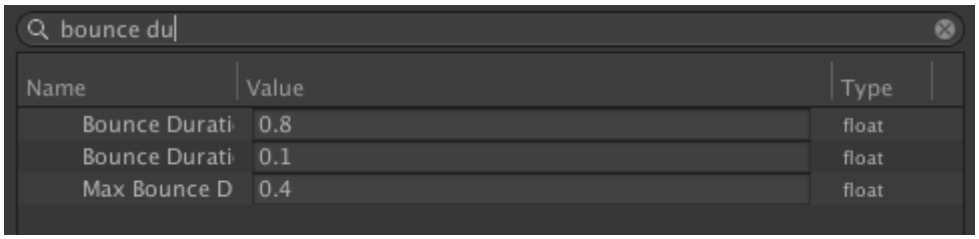
Vanhemman polusta ja Settings-luokan SerialziedObject-ilmentymästä voidaan hakea vanhemman SerializedProperty-ilmentymä. SerializedProperty-ilmentymille, jotka ovat listoja tai taulukoita, voi kutsua funktioita GetArrayElementAtIndex, DeleteArrayElementAtIndex ja InsertArrayElementAtIndex. Ne kaikki ottavat ainoastaan indeksin parametrina. Get-funktio palauttaa SerializedProperty-ilmentymän, joka löytyi sille annetusta indeksistä. Delete-funktio ei poista listan elementtiä, mikäli kyseessä on viittaus olioon, vaan se poistaa viittauksen kyseiseen olioon. Tämän vuoksi Get-funktiolla tarkastetaan, mikä elementti ollaan poistamassa, ja mikäli sieltä löytyy olio, se poistetaan ensin ja sen jälkeen kutsutaan Delete-funktiota uudestaan. Olioviittaukset vaativat siis kaksi kutsua Delete-funktioon, jotta listan elementti saadaan poistettua, mutta muille tyypeille riittää yksi kutsu. Insert-funktio kopioi automaattisesti edeltävän elementin, lisää kopion siitä annettuun indeksiin ja työntää listan loput elementit pykälän eteenpäin. Funktion lopuksi muutokset tallennetaan SerializedObject-ilmentymään kutsumalla sen ApplyModifiedChanges-funktiota ja puu ladataan uudestaan.

TreeView-luokassa hakukenttä toimii siten, että puu piilottaa kaikki ne elementit, jotka eivät täsmää hakukentän arvoon. Kun hakukenttä tyhjennetään, koko puu tulee taas näkyviin. SearchChanged-funktiota kutsutaan aina, kun hakukentän arvo muuttuu. Insinöörityössä funktion määrittelyssä tarkastetaan, että onko hakukentän arvo tyhjä. TreeView-luokalla on “hasSearch”-kenttä, joka palauttaa kyseisen tiedon käyttäen String-luokan “IsNullOrEmpty”-funktiota. Toisin sanoen mikäli hakukenttä on tyhjä ja puunäkymästä on valittu elementtejä, valinnat käydään läpi ja niiden vanhemmat asetetaan avatuiksi. Tämä tehdään jotta valinta säilyy, kun hakutilasta poistutaan. Puunäkymän hakutoiminto ei avaa elementtejä, joten valinta voi kadota, jos jokin elementin vanhemmista on suljettuna. Kun valittujen elementtien vanhemmat on avattu, puunäkymälle kutsutaan

SetFocusAndEnsureSelectedItem-funktiota. Se asettaa Unity-pelimoottorin syötteen osoittamaan puunäkymää ja puun vierittimen siten, että viimeisin valittu alkio jää näkyviin. Funktion tarkoitus on varmistaa, ettei käyttäjä menetä valintaansa haun jälkeen.

DoesItemMatchSearch-funktio suoritetaan aina, kun hakukentän arvo muuttuu. Se ottaa parametrina puunäkymän alkion, jota tarkastellaan ja hakukentän syötteen. Funktion käyttötarkoitus on palauttaa tosi, mikäli annettu hakukentän syöte täsmää TreeViewItem-ilmentymän nimen kanssa. Esimerkiksi kun käyttäjä etsii kenttää nimeltä “Bounce Duration”, hän tyypillisesti kirjoittaa hakukenttään “bounce duration” tai “bounceduration”.

Kuvassa 8 näkyy, että käyttäjä on syöttänyt hakukenttään ”bounce du” ja puu palauttaa kolme alkiota, joiden nimen DoesItemMatchSearch-funktio tulkitsi sopivan hakuun. Tässä tapauksessa kentän löytäminen on suhteellisen suoraviivaista, mutta huomattavaa on, että hakusana ”bounce du” ei löydä kenttiä, joiden nimi on esimerkiksi ”bounce up duration”. Hakufunktio ei siis riko syötettä osiin ja vertaa sen osia.

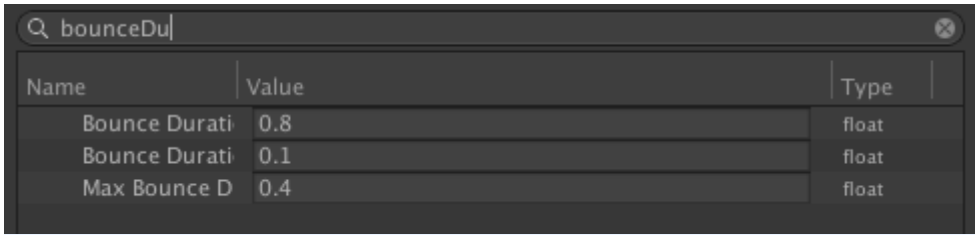


Kuva 8. Hakukenttä, jossa hakuarvo löysi alkioita.

Kuvassa 9 nähdään, että välilyönnin unohtaminen ei palauta toivottuja tuloksia. Ohjelmoinnissa ei kirjoiteta välilyöntejä esimerkiksi funktioiden tai muuttujien nimiin, vaan niissä käytetään isoja kirjaimia merkitsemään sanojen raja: esimerkiksi ”bounceDuration” voisi olla kelvollinen muuttujan nimi. Puun alkioiden näyttönimet ovat muotoa ”Bounce Duration”. Niissä on välilyönti, jota kuvan 9 haku ei pysty löytämään. Tämän voisi korjata niin, että MyTreeViewItem-ilmentymän näyttönimen lisäksi haussa verrataan sen SerializedObject-ilmentymän nimeä ”bounceDuration”. Tämä korjaisi kuvan 9 havainnollistaman ongelman, jossa sanojen rajaa on mahdoton arvata.

Kuva 9. Hakukenttä, jossa hakuarvo ei löydä yhtäkään alkiota.

Kuvassa 10 hakuarvo ”bounceDu” tuottaa saman tuloksen kuin kuvan 8 hakuarvo ”bounce du”. Tämä johtuu siitä, että isosta D-kirjaimesta on helppo nähdä sanan raja. DoesItemMatchSearch-funktiossa on määritelty tapaus, jossa isojen alkukirjainten eteen lisätään välilyönti, mikäli sana ei sellaisenaan tuota tulosta.



Kuva 10. Hakukenttä, jossa hakuarvoa on muokattu.

Muita mielenkiintoisia virtuaalisia funktioita, joita insinöörityössä ei hyödynnetty, ovat esimerkiksi GetCustomRowHeight- ja RefreshCustomRowHeights-funktiot. Ne toimivat yhdessä siten, että jälkimmäistä ei voi kutsua, ellei ensimmäistä ole implementoitu. GetCustomRowHeight-funktio ottaa parametrina rivin indeksin ja TreeViewItem-alkion, joka rivillä on. GetCustomRowHeight-funktio on hyödyllinen esimerkiksi, jos puunäkymässä ei ole eroteltu listojen jäseniä vaan koko lista näytetään yhtenä useamman rivin elementtinä. GetCustomRowHeight-funktiossa voidaan määritellä kyseisen puun elementin korkeus sen perusteella, kuinka iso lista on kyseessä ja montako elementtiä siinä on auki. RefreshCustomRowHeights-funktiota tulee kutsua esimerkiksi, kun alkioita avataan tai suljetaan. Se päivittää rivien korkeudet vastaamaan edeltävässä funktiossa määritellyn korkeuden verran.

TreeView-luokka implementoi myös funktion alkioiden uudelleen nimeämiselle, mutta se ei ole hyödyllinen insinöörityön puitteissa, sillä siinä käytetään peliprojektin koodissa määriteltyjä nimiä. Mikäli kyseessä olisi puurakenne, jossa luodaan dynaamisesti uusia

alkioita tai jos rakenne ei viittaisi koodissa määriteltyihin alkioihin, se olisi hyödyllisempi. Funktiota voisi kuitenkin hyödyntää, mikäli insinöörityöhön implementoidaan Settings.cs-tiedoston tekstin muokkaaminen editorista, mikä on tällä hetkellä epätodennäköistä.

4.5 Alkioiden järjestäminen aakkosjärjestykseen

Insinöörityön puunäkymälle toteutettiin sen järjestäminen nimijärjestykseen käyttäen nimisaraketta. Siihen voisi lisätä arvon mukaan järjestämisen, mutta se jäi pois, sillä Settings-luokassa on kenttiä, joiden vertaaminen on vaikeaa, kuten animaatiokäyrät. Tyypin mukaan järjestäminen on myös vaihtoehto, mutta sitä ei koettu tarpeelliseksi insinöörityön puitteissa.

MultiColumnHeader-luokka implementoi event-kutsun, kun järjestystä muutetaan, mikäli se sallitaan sarakkeessa. Event-kutsuun voi lisätä funktioita, jotka suoritetaan, kun event tapahtuu. Insinöörityössä siihen on lisätty funktiokutsu nimeltä SortIfNeeded eli “järjestä, jos tarpeen”. Mikäli rivejä on vain yksi tai vähemmän, siinä ei ole mitään järjestettävää, joten funktio palaa tekemättä mitään. Mikäli rivejä on enemmän kuin yksi, ne järjestetään nimen mukaan.

Nimen mukaan järjestäminen tapahtuu SortByName-funktiossa, ja siinä hyödynnetään apufunktiota SortRecursively, joka järjestää sille parametrina annetun solmun lapset aakkosjärjestykseen tai käänteiseen aakkosjärjestykseen riippuen siitä, kumpi on viimeisin järjestyssuunta, ja funktion sisällä kutsutaan samaa funktiota, kunnes solmulla ei enää ole lapsisolmuja. Funktio ensin katsoo, onko sille parametrina annetulla TreeViewItem-ilmentymällä lapsisolmuja. Jos solmulla on lapsisolmuja, ne järjestetään nimen mukaan, minkä jälkeen lapsisolmuille kutsutaan samaa funktiota uudestaan. Näin puu järjestetään muuttamatta alkioiden syvyyttä tai vanhempia. Luokan MultiColumnHeader-ilmentymä tietää, missä järjestyksessä se on viimeksi ollut, ja sitä tarkastelemalla lapsisolmut voidaan järjestää käänteiseen järjestykseen, joka vastaa toivottua lopputulosta.

Kun puun kaikki alkiot on järjestetty, sieltä etsitään alkio ”m_Script”, joka viittaa Settings-luokan koodiin, joka siirretään listan huipulle. Tämä tehdään siksi, että sen halutaan aina olevan puun huipulla, jotta se löytyy helposti ja jotta puu näyttää paremmalta.

Koodikenttä piirretään hieman eri tavalla kuin muut kentät, joten sen asettaminen huipulle varmistaa, ettei puu näytä omituiselta.

Järjestämisen lopuksi näkymä pitää päivittää ilman uudelleenrakentamista, sillä se poistaisi järjestelyn lopputuloksen. Päivitys tehdään hieman kikkaillen siten, että SettingsTreeView-ilmentymän kantasolmu ”root” suljetaan ja avataan uudestaan SetExpanded-funktiolla. Se ei vaikuta puun muiden solmujen tilaan. Päivitys tehdään näin, sillä tällä tavoin vältetään ohjelman kääntäminen. Kääntäminen poistaisi juuri tehdyn järjestelyn lopputuloksen. SetExpanded-funktion sisällä päivitetään näkymä ilman BuildRoot-funktion kutsumista, joka palauttaisi näkymän oletustilaansa.

Kuvassa 11 näkyy, miten puun järjestely toteutetaan. Kantasolmulle ”rootItem” kutsutaan SortRecursively-funktiota, joka järjestää sille annetun TreeViewItem-ilmentymän lapsisolmut nimijärjestykseen. Jokaiselle lapsisolmulle kutsutaan samaa funktiota, kunnes lapsisolmuja ei enää ole ja puu on järjestyksessä. MultiColumnHeader-ilmentymän IsSortedAscending-funktio palauttaa toden tai epätoden sen perusteella, onko sille parametrina annettu sarake järjestetty nousevassa järjestyksessä vai ei. Sarakkeen indeksi nolla viittaa nimisarakkeeseen. Lapsisolmut järjestetään aakkosjärjestykseen tai käänteiseen aakkosjärjestykseen riippuen siitä, mitä funktio palauttaa. MultiColumnHeader-ilmentymän tieto sen järjestyksestä päivitetään järjestämisen jälkeen, jolloin seuraava järjestys tuottaa käänteisen tuloksen.

```
SortRecursively(this.rootItem);
// NOTE: Keep m_Script on top
TreeViewItem scriptItem = this.rootItem.children.First(v => v.displayName == "Script");
this.rootItem.children.Remove(scriptItem);
this.rootItem.children.Insert( index: 0, scriptItem);

void SortRecursively(TreeViewItem parent)
{
    if (parent.children != null)
    {
        parent.children = this.multiColumnHeader.IsSortedAscending( columnIndex: 0)
            ? parent.children.OrderByDescending(l => l.displayName).ToList()
            : parent.children.OrderBy(l => l.displayName).ToList();
        foreach (TreeViewItem child in parent.children)
        {
            SortRecursively(child);
        }
    }
}
```

Kuva 11. SortByName-funktion implementaatio.

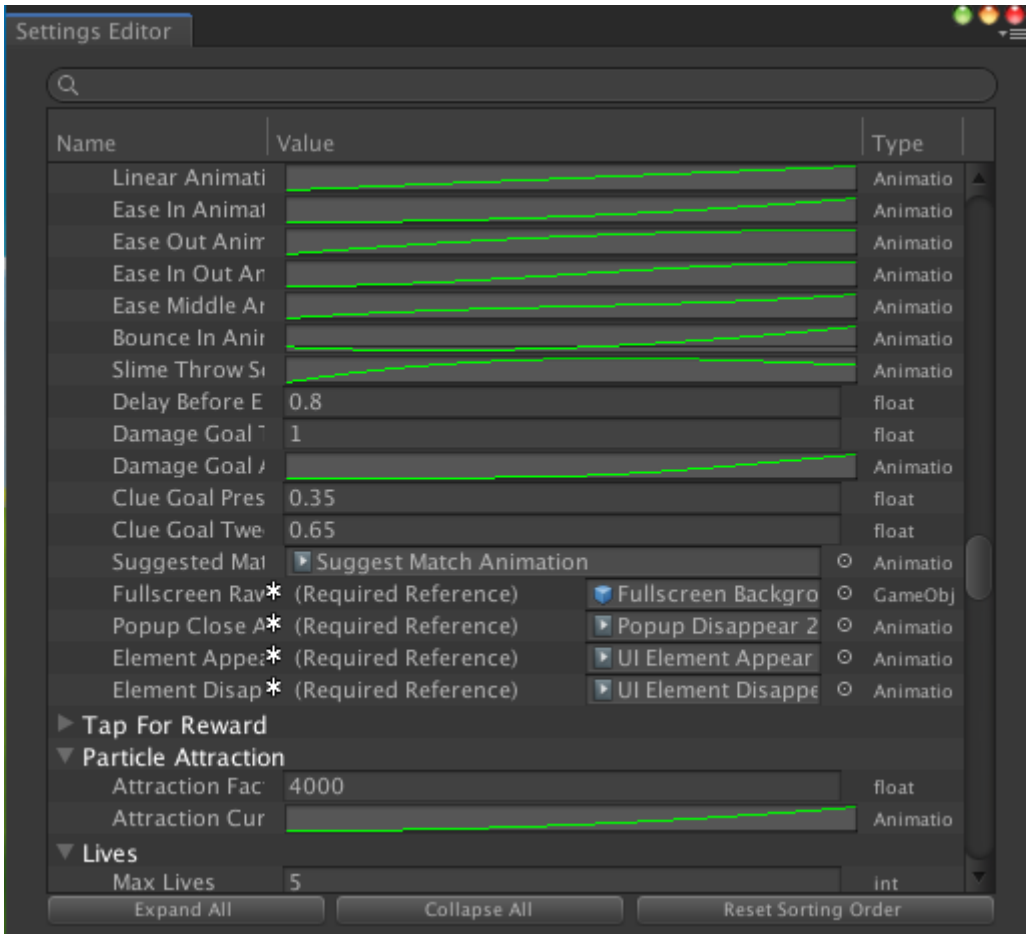
5 Tulokset ja jatkokehitys

Insinöörityön lopputulos oli puunäkymä työryhmän sisäistä käyttöä varten. Työryhmä pystyy käyttämään aikansa paremmin, kun sen jäsenten ei tarvitse enää tarkalleen tietää mistä päin Settings-tiedostoa jokin tietty kohta löytyy. Kahdensadan kentän selaamisen välttämisellä säästetyn ajan voi käyttää paremmin muuhun työskentelyyn. Puunäkymä tarjoaa asetusten muokkaamiselle työkaluja, joita ei ennen ollut käytettävissä, mutta joita ilman ei enää voisi työskennellä. Luotu puunäkymä on pääosin valmis kokonaisuus, mutta siinä on mahdollisuuksia jatkokehitykselle ja korjauksille.

5.1 Lopputulokset

Insinöörityö auttoi työryhmän työskentelyä huomattavasti. Hakukenttä on työn todellinen saavutus, sillä pelkästään sen ansiosta tietyn kentän etsiminen ja sen arvon muuttaminen on äärimmäisen helppoa. Selaaminen, usean kentän samanaikainen muokkaaminen ja yleiset parannukset ulkoasussa tekevät kokonaisuudesta tehokkaamman ja selkeämmän käyttää, joten käyttäjälle jää enemmän aikaa keskittyä muihin asioihin.

Kuvassa 12 näkyy insinöörityössä luotu valmis puunäkymä. Siitä nähdään, että kentät on jaoteltu aihealueisiin, joita voidaan avata ja sulkea niiden vasemmalla puolella olevasta nuolesta. Tausta on raidoitettu selkeyttämään lukemista, ja sarakkeiden kokoa voi muuttaa tarpeen mukaan esimerkiksi näyttämään enemmän tekstiä. Väliotsikot tekevät koko tiedostosta pidemmän, ja kuvan oikeassa laidassa olevasta vierittimestä näkee, että uudessa näkymässä on enemmän rivejä kuin vanhassa. Ylälaidassa on hakukenttä, ja alalaidassa on napit kaikkien alkioiden avaamiseen ja sulkemiseen ja nappi koko puun uudelleenrakentamiseen. Ylälaidassa näkyvät sarakkeet, joiden nimet ovat nimi, arvo ja tyyppi. Nimisaraketta napauttamalla puun voi järjestää aakkosjärjestykseen. Luotu puunäkymä helpotti tiedoston navigointia merkittävästi.



Kuva 12. Insinöörityössä luotu valmis näkymä.

TreeView-luokka soveltuu ison datamäärän käsittelyyn, mutta paikoittain sen muokkaaminen juuri omiin tarpeisiin sopivaksi on vaikeaa, ellei tiedä, mitä on tekemässä. Unity-pelimoottorissa editoriohjelmointia usein pidetään rasittavana sen vuoksi, että se nojaa IMGUI:hin eli että koodissa määritellään, missä napit ovat ja mitä niiden painaminen tekee. Moni suosii lähestymistapaa, jossa elementit luodaan muualla kuin koodissa, esimerkiksi pelinäkymässä, jossa ne on helpompi asetella haluamiinsa kohtiin, mutta editoriohjelmointi ei tue sitä. IMGUI:lla ohjelmointi on mielenkiintoinen haaste, ja tämäntyyppinen projekti on erinomainen oppiväline.

5.2 Tulevaisuus ja ongelmakohtien kehitys

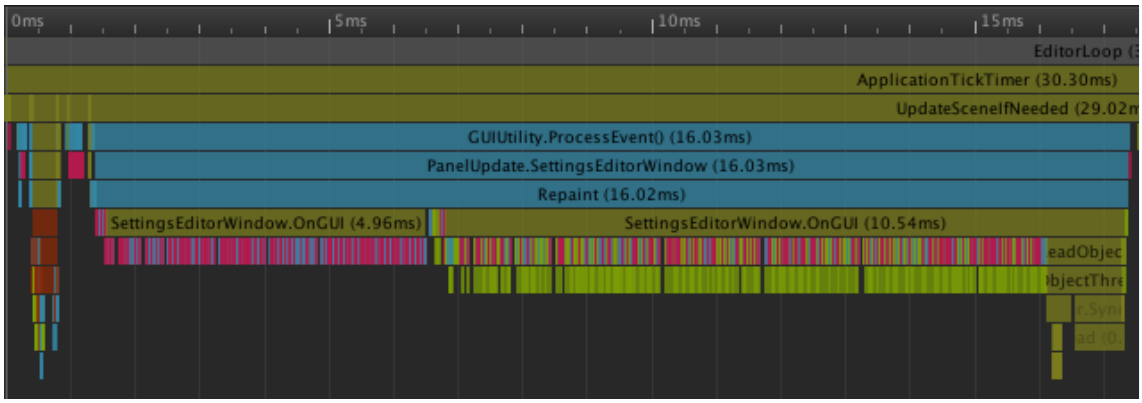
Editorin tulevaisuuden kannalta harkinnassa on välilehtien käyttö, jossa data olisi jaoteltu otsikoiden mukaan. Tämä saattaa yhä toteutua laajennuksena nykyiseen implementaatioon, mikäli kenttien määrä kasvaa niin isoksi, ettei yksi puunäkymä enää riitä järkevällä tavalla ja alkaa näyttää sotkuiselta. Nykyisellään näkymää on helppo selata, mutta kenttien määrän kasvaessa se saattaa taas hankaloitua. Välilehdet voisi myös jaotella esimerkiksi tyyppien mukaan, tai ne voisi dynaamisesti jaotella haluamallaan tavalla. Peliprojektin asetuksissa on määritelty huomattava määrä animaatiokäyriä, jotka voisivat olla omassa välilehdessä. Välilehtien kanssa haasteeksi saattaa nousta se, miten hakukenttä toimii niiden kanssa, sillä ehdoton vaatimus on, että haku kattaa kaikki välilehdet.

Mahdollisesti tärkein jatkokehityksen kohde on hakukentän syötteen käsittelyn päivittäminen. Se täytyy korjata hyväksymään yhteen kirjoitettua syötettä, ja sen on tunnistettava, että haettujen sanojen välissä voi olla muitakin sanoja. Esimerkiksi haulla ”bounceduration” tulee löytää kentät, joiden näyttönimessä on ”bounce duration”, ja haulla ”bounce duration” tulee löytää kentät, joiden nimessä missään kohdassa on sanat ”bounce” ja ”duration”, kuten ”bounce up duration” ja ”bounce duration”.

Muita haastavia kehitysideoita voisi olla kokonaan uusien kenttien luominen suoraan editorista. Tällä hetkellä uusi kenttä on pakko kirjoittaa Settings.cs-tiedostoon manuaalisesti, jotta se näkyy puussa. Olisi mielenkiintoista implementoida vaihtoehto, joka ei vaadi, että Settings.cs-tiedostoa avataan, vaan se kirjoittaisi kentän sinne automaattisesti. Esimerkiksi nappia painamalla aukeaisi tekstikenttä, joka ottaa syötteen, joka lisätään tiedostoon ja tallennetaan. Tässä on monia haasteita, kuten juuri oikean kohdan löytäminen ja mahdolliset virhesyötteet, mutta teoriassa sen voisi toteuttaa tavallisilla tiedostonmuokkausvälineillä avaamalla tietojonon, joka lisää syötteen tiedostoon. Virhesyötteiden välttämiseen voisi käyttää valmista valikkoa kentän tyypille, ja ainoa käyttäjän syöte olisi kentän nimi. Idealle ei kuitenkaan olisi paljon käyttöä, sillä uudet kentät syntyvät ohjelmoidessa, jolloin on helpompi käydä lisäämässä uusi kenttä suoraan Settings.cs-tiedostoon eikä käydä editorin kautta. Editorin käyttötarkoitus on kenttien arvojen muokkaaminen, ei uusien kenttien lisääminen.

Tulevaisuudessa optimointi saattaa nousta oleelliseksi osaksi editorin kehitystä. Tämä kuitenkin riippuu täysin siitä, miten käyttäjät kokevat asian. Puunäkymään profiloimalla voidaan nähdä, onko näkymä raskas vai ei. Unity-pelimoottorin profiloija on voimakas työkalu sen selvittämiseen, mihin resursseja menee eniten. Sitä voi käyttää sekä editoriin että yhdistettyihin laitteisiin.

Kuvassa 13 näkyy, mihin puunäkymää ladatessa aikaa menee, ja siinä huomataan, että puun rakentamiseen meni kokonaisuudessaan 16,03 millisekuntia, joka on jaoteltu pienempiin osiin. Repaint-funktion sisällä siihen meni ensin 4,96 millisekuntia ja pienen tauon jälkeen 10,54 millisekuntia eli yhteensä 15,5 millisekuntia. Profiloija näyttää, että nämä ajanjaksot koostuvat lähinnä muistiallokaatioista ja renderöinnistä eli ikkunan eri komponenttien piirtämisestä. Nämä tulevat siitä, kun puunäkymän osat rakennetaan ja tallennetaan muistiin ja lopulta piirretään ikkunaan. 15,5 millisekuntia alkaa olla ihmissilmälle näkyvä aika. Peleissä selittämätön 15 millisekunnin viive ei ole hyväksyttävää, mutta editorin tapauksessa tämän voi hyväksyä, sillä ei juurikaan vaikuta käyttäjäkokemukseen.



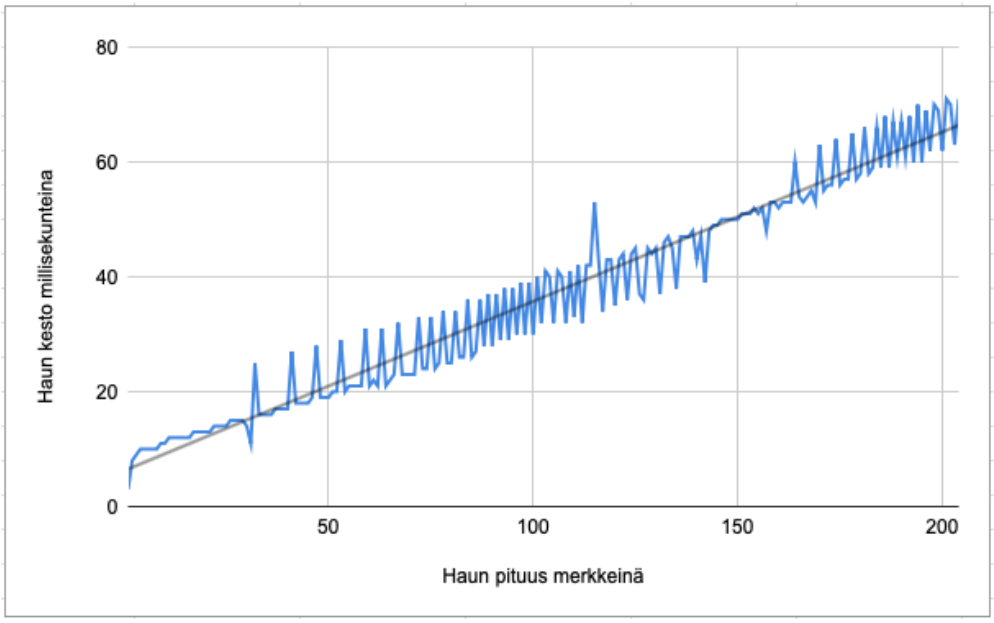
Kuva 13. Unity-pelimoottorin profiloijan näyttämä data, kun puunäkymä rakennetaan uudestaan.

Vaikka 15,5 millisekuntia on pieni aika, se saattaa jatkossa kasvaa isommaksi. Mikäli näin käy, optimointi saattaa olla yksi tulevaisuudenhaaste. Profiloijan datasta huomataan, että muistiallokaatiot ovat yksi osa-alue, jota kannattaa lähteä parantamaan. Itse muistinkäyttö ei ole ongelma vaan se, kun varataan uutta muistia. Insinöörityössä tähän voi vaikuttaa esimerkiksi siirtämällä listan luonnin pois BuildRoot-funktiosta ja varaamalla sopivankokoisen muistialueen puun elementeille, ennen kuin sitä tarvitaan ohjelman suorituksessa. Muita optimointikohtia voisi olla foreach-kiersiöiden ja LINQ-

operaatioiden muuttaminen for-kiersiöiksi, mikä on nyrkkisääntönä nopeampi kuin edeltävät ratkaisut, mutta kenties vaikeammin luettava.

Toinen optimoinnin kohde voisi olla hakukentän syötteen käsittely. Siihen voi vaikuttaa muun muassa laittamalla ne hakuehdot, jotka todennäköisimmin tuottavat positiivisen tuloksen, ensin ja vaativammat haut, joissa syötettä muokataan, kutsutaan viimeisenä.

Kuva 14 osoittaa, että mitä pitempi haku, sitä kauemmin sen käsittelyyn menee. Aika kasvaa lineaarisesti, ja kuvasta voidaan laskea, että hakuaika merkkiä kohden on keskimäärin 0,5 millisekuntia. Yllättävää on, että haun kesto merkkiä kohden laskee noin 0,3 millisekuntiin, mitä enemmän merkkejä käydään läpi. Noin 20 merkkiin asti, haun kesto merkkiä kohden on noin yksi millisekunti ja 1–5 merkin kohdalla se on jopa 4 millisekuntia merkkiä kohti. Tämä todennäköisesti johtuu siitä, että kun merkkejä on vain yksi, siinä on enemmän tilaa sattumanvaraisille tekijöille, mutta kun funktio käy 200 merkkiä läpi, data tasoittuu. Haun keston kasvu johtuu siitä, että hakufunktion on käytävä jokainen hakukenttään syötetty merkki erikseen läpi. Tämän lisäksi pidemmät haut tuottavat epätodennäköisemmin positiivisen lopputuloksen ensimmäisen ehdon kohdalla, joten niille joudutaan myös todennäköisemmin käymään useampi ehto läpi.



Kuva 14. Hakukentän syötteen pituuden vaikutus haun kestoon.

Koska TreeView-luokka oli alkuun tuntematon ja insinöörityö lähti liikkeelle vain tiedosta, että se on olemassa, insinöörityön aikana ilmeni monia haasteita. Ratkaistavaksi tuli TreeView-luokan omia ongelmia ja toimintaperiaatteita, mutta myös siihen liittyvien asioiden ratkaisemista, kuten hakujen parsiminen ja SerializedProperty-ilmentymien käyttämiseen liittyvät haasteet.

Insinöörityön alussa listat, taulukot ja geneeriset luokat näkyivät puussa yhtenä monen rivin elementtinä. Esimerkiksi lista, jolla on 5 elementtiä, näkyi noin 11 rivin kokoisena alkiona, joka oli vain yksi TreeViewItem-ilmentymä. Työn suurin haaste oli käydä ne läpi niin, että niiden yksittäiset elementit voidaan syöttää puuhun omina alkioinaan. GoThroughProperty-funktio lopulta ratkaisi tämän ongelman. Oletuksena Unityn PropertyField piirtää ne yhtenä useamman rivin kenttänä, mutta editorin haluttiin piirtävän ne erikseen. Tämän ratkaiseminen vaati sen, että TreeViewItem-luokasta tehtiin siitä periytyvä aliluokka MyTreeViewItem, joka sisältää polun SerializedPropertyyn, johon alkio viittaa. Jotta alkioihin päästään käsiksi, ne joudutaan käymään rekursiivisesti läpi jokaisen SerializedPropertyn joita SerializedObjectissa on, kunnes niistä löytyy primitiivinen tyyppi, joka voidaan tallentaa sellaisenaan puuhun. Kentät, jotka itsessään eivät olleet primitiivisiä tyyppejä, tallennettiin yläotsikoksi niiden alta löytyville tyypeille.

Edellä kuvatun haasteen ratkaiseminen toi esille uuden ongelman eli listojen käsittely. Oletuksena Unity-näkymä lisää listoille automaattisesti vaihtoehdot muokata listan kokoa, duplikaattien luomisen ja listan välistä alkion poistamisen. Nämä kaikki katoavat, kun elementit piirretään erikseen, koska ne eivät editorin mielestä enää ole osa listaa vaan yksittäisiä elementtejä. Tämä kuitenkin ratkaistiin TreeViewItemistä perityllä luokalla, jota kaikki puusta löytyvät elementit ovat. Luokkaan lisättiin tieto siitä, onko elementti listan jäsen. Mikäli elementti on merkitty listan jäseneksi puun luomisvaiheessa, voidaan myöhemmin OnGUI-funktiossa lisätä näin merkityille jäsenille erillinen valikko, johon määrittelen duplikoinnin ja poistamisen. SerializedPropertyllä on olemassa jäsenfunktiot, jotka mahdollistavat suoraan taulukon elementtien poistamisen tai lisäämisen. Koodissa voidaan etsiä alkion yläluokka, ja yläluokalle voidaan sanoa, että se poistaa tai duplikoi elementin.

Hakukenttään syötetyn tekstin muotoilu osoittautui omanlaiseksi haasteekseen. Esimerkiksi alkion nimi koodissa voi olla ”cascadeDuration” ja puunäkymässä ”Cascade

Duration”, mutta sen halutaan löytyvän sekä hakusanalla ”cascadeduration” että ”cascade duration”. Tämä tarkoittaa, että syötetty teksti joudutaan parsimaan muotoon, joka löytyy puusta. Tämä on yhä ratkaisematon ongelma, sillä on mahdoton päätellä, missä sanan raja menee, jos käyttäjä kirjoittaa hakukenttään ”cascadeduration”. Ongelman voisi ratkaista SerializedPropertyä hyödyntäen, sillä se sisältää tiedon nimestä, joka löytyy koodista, sekä nimestä, joka esitetään käyttäjälle. Teoriassa puuhun tallennettu nimi voisi olla ”cascadeDuration” ja se voidaan koodissa muokata muotoon ”Cascade Duration”, joka näytetään käyttäjälle. Hakua voitaisiin verrata molempiin.

Editor-luokalle voidaan lisätä attribuutti ”CanEditMultipleObjects”, joka mahdollistaa sen ilman ohjelmoijan sekaantumista, mutta TreeView- ja EditorWindow-luokat eivät periydy Editor-luokasta, vaan UnityEditor-luokasta, joka ei tue tätä ominaisuutta. Tämän vuoksi insinöörityössä piti tehdä oma implementaatio useamman elementin kopioinnille. TreeView-luokka näyttää, mitkä kentät ovat valittuna. Jokainen valittu kenttä voidaan käydä läpi, ja niistä nähdä, mitä kenttää parhaillaan muokataan. Oma usean kentän muokkaaminen toteutettiin siten, että sen kaikki valitut käydään kentät läpi ja katsotaan, mitkä ovat samaa tyyppiä kuin muokattu kenttä ja lopulta kaikkien kenttien arvot asetetaan samaksi kuin muokatun kentän arvot.

SerializedProperty on tyyppi, jota ei ole tarkoitettu osittaiseen kopiointiin, koska se on viittaus koko kenttään ja sen kaikkiin tietoihin. Se sisältää tiedon kentän tarkasta polusta, sen tyypistä ja sen arvosta. Kun SerializedProperty-tyyppinen alkio kopioidaan, oletuksena on, että se kopioidaan esimerkiksi ScriptableObject-tiedostosta toiseen saman tyypin tiedostoon, jolloin kaikki sen sisältämä data on olennaista. Peliprojektissa on kuitenkin vain yksi ScriptableObject-tiedosto, ja sen tapauksessa kopioidessa halutaan ainoastaan kopioida kentän arvo eikä sen polkua tai tyyppiä. Tämä tarkoittaa, että kentän sisältämän arvon tyyppi on selvitettävä ja sen perusteella on kopioitava tietyn arvokentän arvo. Siitä tulee virheilmoitus, mikäli SerializedProperty-ilmentymän tyyppi on kokonaisluku ja siitä yritetään kopioida merkkijonon arvoa.

6 Yhteenveto

Insinöörityön tavoite oli luoda selkeä ja helppokäyttöinen muokkain ScriptableObject-tiedostolle Unity-pelimoottorin oletusnäkymän tilalle. Pelimoottorin oletusnäkymä listaa kenttiä allekkain, ja peliprojektin asetusten kahdensadan kentän kohdalla siitä tulee vaikealukuista. Pelimoottorin tarjoamat lisämääreet muokkaimen selkeyttämiselle eivät riitä jaottelemaan senkokoista datamäärää, kuin mistä peliprojektissa oli kyse. Insinöörityössä hyödynnettiin SerializedObject- ja SerializedProperty-luokkia siten, että niiden avulla ScriptableObject-tiedoston kenttiin päästiin käsiksi ja niitä pystyi muokkaamaan. Reflektio on avainasemassa, kun käsitellään dataa, jonka tyypistä ei olla varmoja tai kun siitä ei olla kiinnostuneita. Sen avulla löydetään muun muassa metadatana olevat lisämääreet.

TreeView-luokka soveltuu ison datamäärän jaottelemiselle. Se jaottelee kentät syvyyden mukaan otsikoiden alle, joita voi avata ja sulkea, jolloin vain olennainen tieto on näkyvissä. Luotu TreeView-näkymä piirretään EditorWindow-luokan avulla, jossa käytetään IMGUI-kontrolleja piirtämään puun elementtejä ja kontrolleja. Insinöörityön puunäkymässä hyödynnettiin sarakkeita, jotka merkitsivät kentän nimen, arvon ja tyypin. Toiminnallisuuteen lisättiin myös alkioiden järjestäminen nimen mukaan ja hakukenttä.

Puunäkymä rakennettiin käymällä SerializedObject-ilmentymän kentät läpi rekursiivisesti, jotta jokainen kenttä myös listojen ja taulukoiden alta saadaan omaksi alkioksi puuhun. Tämän vuoksi siitä puuttui listoille ja taulukoille ominaiset toiminnallisuudet, kuten elementtien lisääminen ja poistaminen. Ne tehtiin manuaalisesti TreeView-luokan tarjoaman ContextClickedItem-funktiolla. Elementin vanhemman voi löytää sen SerializedProperty-ilmentymän polusta, ja elementin vanhemmalle voi kutsua SerializedProperty-luokan funktioita DeleteElementAtIndex ja InsertElementAtIndex. Niille annetaan elementin indeksi, joka myös löytyy polusta, ja SerializedProperty-luokka hoitaa loput.

Puunäkymä on todettu erittäin hyödylliseksi, kun pitää jaotella isoja määriä dataa. Mikäli elementtejä ei voi jaotella aihealueisiin tai datasta ei saa rakennettua puuta, puunäkymä on turha. Tähän käyttötarkoitukseen se kuitenkin sopii erinomaisesti.

Tulevaisuudessa editori todennäköisesti kehittyy käyttämään välilehtiä. Yksi mahdollisuus niille on dynaaminen jaottelu esimerkiksi tyypin mukaan. Toinen kehitysidea on kenttien lisäämisen helpottaminen, mutta se ei liene ajankohtaista pitkään aikaan.

Editoria ei ole optimoitu, mutta jatkossa siitä saattaa tulla ajankohtaista, mikäli datamäärä kasvaa huomattavasti. Sitä tehdessä ilmeni paljon haasteita, jotka ratkaistiin tai jotka kierrettiin. Esimerkiksi listojen ja taulukoiden elementtien saaminen omiksi alkioikseen oli ongelma, joka ratkaistiin, mutta sitä, miten UnityEngine-luokan PropertyField-funktion saa piirtämään ainoastaan kentän, mutta ei sen lisämääreitä, ei varsinaisesti ratkaistu vaan se kierrettiin luomalla oma lisämääre, jota PropertyField-funktio ei tunnista.

Insinöörityö pakotti tekijän tutustumaan syvemmin Unity-pelimoottorin toimintaan ja C#-ohjelmointikielen ominaisuuksiin. TreeView-näkymä on helppo ottaa käyttöön, mutta vaikea muokata juuri omaan käyttötarkoitukseen sopivaksi. Se vaatii käyttäjältä aiempaa kokemusta editoriohjelmoinnista tai kärsivällisyyttä oppia.

Lähteet

1 Build once, deploy anywhere - Industry-leading multiplatform support. 2019. Verkkoaineisto. Unity Technologies. <https://unity3d.com/unity/features/multiplatform>. Luettu 14.9.2019.

2 System Requirements for Unity 2019.2. 2019. Verkkoaineisto. Unity Technologies. <https://unity3d.com/unity/system-requirements>. Luettu 14.9.2019.

3 Experienced programmer, but new to Unity? - You're already ahead of the game. 2018. Verkkoaineisto. Unity Technologies. <https://unity3d.com/programming-in-unity>. Luettu 14.9.2019.

4 Fine, Richard. 2017. UnityScript's long ride off into the sunset. Verkkoaineisto. <https://blogs.unity3d.com/2017/08/11/unityscripts-long-ride-off-into-the-sunset/>. 11.8.2017. Luettu 14.9.2019.

5 Unity 5.0. 2014. Verkkoaineisto. Unity Technologies. <https://unity3d.com/unity/whats-new/unity-5.0>. Luettu 14.9.2019.

6 Matney, Lucas. 2019. Unity raises $181M monster round at a reported $1.5B valuation. Verkkoaineisto. <https://techcrunch.com/2016/07/13/unity-announces-181-million-monster-round-led-by-dfj-growth/>. 13.7.2019. Luettu 14.9.2019.

7 This engine is dominating the gaming industry right now. 2016. Verkkoaineisto. TNW Deals. <https://thenextweb.com/gaming/2016/03/24/engine-dominating-gaming-industry-right-now/>. Luettu 14.9.2019.

8 Unity growth facts. 2019. Verkkoaineisto. Unity Technologies. <https://unity3d.com/public-relations>. Luettu 14.9.2019.

9 ScriptableObject. 2018. Verkkoaineisto. Unity Technologies. <https://docs.unity3d.com/Manual/class-ScriptableObject.html>. 15.10.2018. Luettu 15.10.2019.

10 Serialization (C#). 2018. Verkkoaineisto. Microsoft Corporation. <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/serialization/>. 26.3.2018. Luettu 14.9.2019.

11 Kanjilal, Joydip. 2015. How to work with attributes in C#. Verkkoaineisto. <https://www.infoworld.com/article/3006630/how-to-work-with-attributes-in-c.html>. 20.11.2015. Luettu 14.9.2019.

12 Static (C# reference). 2015. Verkkoaineisto. Microsoft Corporation. <https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/static>. 20.7.2015. Luettu 14.9.2019.

13 Const (C# reference). 2015. Verkkoaineisto. Microsoft Corporation. <https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/const>. 20.7.2015. Luettu 14.9.2019.

14 Readonly (C# reference). 2018. Verkkoaineisto. Microsoft Corporation. <https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/readonly>. 21.6.2018. Luettu 14.9.2019.

15 Meijer, Lucas. 2014. Serialization in Unity. Verkkoaineisto. <https://blogs.unity3d.com/2014/06/24/serialization-in-unity/>. 24.6.2014. Luettu 14.9.2019.

16 SerializedObject. 2019. Verkkoaineisto. Unity Technologies. <https://docs.unity3d.com/ScriptReference/SerializedObject.html>. 8.10.2019. Luettu 15.10.2019.

17 SerializedProperty. 2019. Verkkoaineisto. Unity Technologies. <https://docs.unity3d.com/ScriptReference/SerializedProperty.html>. 8.10.2019. Luettu 15.10.2019.

18 Cailey, Arthur. 1857. On the theory of the analytical forms called trees. The London, Edinburgh and Dublin Philosophical Magazine and Journal of Science, 1857, s. 172–176.

19 Von Staudt, K. G. C. 1847. Geometre der Lage. Nürnberg, Deutschland 1847, s. 20–21.

20 Coetzee, Derric. 2005. Verkkoaineisto. File:Binary tree.svg. <https://commons.wikimedia.org/wiki/File:Binary_tree.svg>. 31.12.2005. Luettu 15.10.2019.

21 Lambda expressions (C# Programming Guide). 2019. Verkkoaineisto. Microsoft Corporation. <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/statements-expressions-operators/lambda-expressions>. 29.7.2019. Luettu 15.10.2019.

22 Reflection (C#). 2015. Verkkoaineisto. Microsoft Corporation. <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/reflection>. 20.7.2015. Luettu 15.10.2019.

23 TreeView. 2019. Verkkoaineisto. Unity Technologies. <https://docs.unity3d.com/ScriptReference/IMGUI.Controls.TreeView.html>. 8.10.2019. Luettu 15.10.2019.

24 TreeView. 2019. Verkkoaineisto. Unity Technologies. <https://docs.unity3d.com/Manual/TreeViewAPI.html>. 8.10.2019. Luettu 15.10.2019.

25 Liu, Edward. 2015. Immediate Mode GUI. Verkkoaineisto. <http://behindthe-pixels.io/IMGUI/>. 19.2.2015. Luettu 15.10.2019.

26 MonoBehaviour. 2019. Verkkoaineisto. Unity Technologies. <https://docs.unity3d.com/ScriptReference/MonoBehaviour.html>. 8.10.2019. Luettu 16.10.2019.

27 MenuItem. 2019. Verkkoaineisto. Unity Technologies. <https://docs.unity3d.com/ScriptReference/MenuItem.html>. 8.10.2019. Luettu 15.10.2019.

28 Object Oriented Programming - Abstract Methods and Classes in C#. 2019. Verkkoaineisto. SyntaxDB. <https://syntaxdb.com/ref/csharp/abstract>. Luettu 17.10.2019.