TAMPEREEN
AMMATTIKORKEAKOULU

SOVELLUSKEHYKSEN KEHITTÄMINEN UNITY-PELIMOOTTORILLE

Miikka Kivelä

Opinnäytetyö
Joulukuu 2015
Tietojenkäsittely
Ohjelmistotuotanto

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tietojenkäsittely
Ohjelmistotuotanto

KIVELÄ, MIIKKA:
Sovelluskehyksen kehittäminen Unity-pelimoottorille

Opinnäytetyö 58 sivua, joista liitteitä 11 sivua
Joulukuu 2015

Tämän opinnäytetyön tavoitteena oli kehittää Kyy Games Oy:n jo olemassa olevaa sovelluskehystä Unity-pelimoottorille. Sovelluskehys sisältää laajennoksia ja työkaluja Unity-pelimoottorille ja -editorille. Sovelluskehystä haluttiin kehittää, jotta voitaisiin edelleen nopeuttaa ja tehostaa toimeksiantajan pelinkehitysprosessia ja pelien prototyyppien tekemistä.

Opinnäytetyöllä oli kaksi erillistä tarkoitusta: kirjata sovelluskehyksen hallinnointiin liittyviä ongelmia ja ehdotuksia hallinnoinnin parantamiseen sekä tuottaa uusia laajennoksia sovelluskehykseen.

Työn tuotoksina syntyi kaksi uutta laajennosta sovelluskehykseen, joita ovat pelin tallennus- ja latausjärjestelmä sekä järjestelmä pelin dialogi-ikkunoiden luomiseen ja hallitsemiseen. Kummankin laajennoksen tueksi toteutettiin myös perusteellinen tekninen dokumentaatio ja esimerkki laajennoksen toiminnasta. Lisäksi laajennokset testattiin toimeksiantajan kanssa sovittujen vaatimusten mukaisesti. Laajennosten lisäksi kirjalliset ehdotukset sovelluskehyksen hallitsemisen parantamiseen kirjattiin tämän raportin kolmanteen lukuun.

Toteutetut laajennokset liitetään toimeksiantajan sovelluskehykseen sekä sovelluskehyksen hallinnan ongelmat ja ehdotukset sen parantamiseen käydään toimeksiantajan kanssa yhdessä läpi vuoden 2015 loppupuolella.

Asiasanat: Unity, sovelluskehys, pelinkehitys

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Degree Programme in Business Information Systems
Option of Software Development

KIVELÄ, MIIKKA:
Development of a Framework for Unity Game Engine

Bachelor's thesis 58 pages, appendices 11 pages
December 2015

The objective of this thesis was to develop Kyy Games Ltd's existing framework for the Unity game engine. The framework in question contains extensions and tools for the Unity game engine and Unity editor. The reason for developing the framework was to make the company's game development process and game prototyping faster and more efficient.

This thesis had two purposes. The first one was to write down problems concerning the framework's management and suggestions to improve it. The second one was to create new extensions to the framework.

Two new extensions for the framework were created: a system for saving and loading game data and another system for creating and managing dialog windows. Extensive technical documentation and an example of how each extension works were also created. Both extensions were tested according to the requirements that had been set together with the company. Suggestions for improving the framework's management were included in the third chapter of this report.

Extensions that were implemented will be integrated with the company's existing framework before the end of the year 2015. Means of improving the framework's management will also be discussed with the client before the end of the year 2015.

Key words: Unity, framework, game development

SISÄLLYS

LYHENTEET JA TERMIT

Ainokainen	Suunnittelumalli ohjelmistotuotannossa
Asset	Unity-projektissa oleva tiedosto
Asynkronisuus	Tietyn ohjelmaprosessin toimiminen itsenäisesti muusta ohjelmasta riippumatta
Binäärimuoto	Tiedostomuoto, jossa tieto esitetään tavuina
C#	Microsoftin kehittämä oliopohjainen ohjelmointikieli
Dialogi-ikkuna	Graafisen käyttöliittymän elementti, jonka tarkoituksena on viestiä tietoa käyttäjälle
JSON	JavaScript Object Notation, tekstimuotoinen tiedostomuoto jossa tieto esitetään avain-arvo-pareina
Kerrosarkkitehtuuri	Ohjelmistoarkkitehtuurityyli, jossa käytetään tasoja kuvaamaan järjestelmän arkkitehtuuria
Komponentti	Unityssä peliobjekteihin liitettävä toiminnallisuusosa
Nimiavaruus	Joukko formaalin kielen symboleja, joilla voidaan organisoida koodiprojekteja
Nimikonflikti	Ongelma ohjelmassa joka tapahtuu jos samassa nimiavaruudessa on kaksi samannimistä tunnistetta
Ominaisuus	Luokan jäsen C#-ohjelmointikielessä, jolla voidaan kirjoittaa ja lukea luokan muuttujaa
Parametri	Metodille välitettävä tieto
Pelimoottori	Ohjelmistokehys videopelien tekemiseen
Peliobjekti	Pelimaailmassa esiintyvä objekti Unityssä
Rajapinta	Tietotyyppi ohjelmointikielessä joka kuvailee ja määrittelee luokan toiminnallisuuden, mutta ei toteuta sitä
Scene	Pelimaailma Unityssä, jonne voidaan lisätä peliobjekteja
Serialisaatio	Prosessi, jossa olio muutetaan tallennettavaan tietoon
Skripti	Ohjelmointikieltä sisältävä tiedosto Unityssä
Sovelluskehys	Ohjelmistotuote, joka muodostaa rungon sen päälle rakennettavalle ohjelmalle
Säie	Ohjelmassa itsenäisesti työtä suorittava osa
Unity	Unity Technologies:n kehittämä videopelien kehitysympäristö

UnityScript	Unityä varten kehitetty JavaScriptin kaltainen ohjelmointikieli
Videopeli	Elektroninen peli, jota pelataan esimerkiksi tietokoneella tai mobiililaitteella
XML	Extensible Markup Language, tekstimuotoinen tiedostomuoto joka koostuu elementeistä, attribuuteista ja niiden arvoista

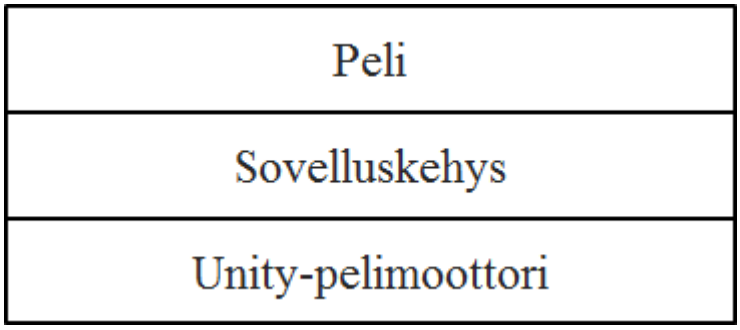
1 JOHDANTO

Opinnäytetyön toimeksiantajana on tamperelainen pelitalo Kyy Games Oy. Kyy Games kehittää pelejä sekä tietokoneille että mobiililaitteille. Niiden kehittäminen tapahtuu pääasiassa Unityllä, joka on suosittu videopelien kehitysympäristö.

Työn tavoitteena on kehittää toimeksiantajan jo olemassa olevaa sovelluskehystä Unity-pelimoottorille. Työn tarkoituksena on kirjata sovelluskehyksen hallinnointiin liittyviä ongelmia ja ehdotuksia hallinnoinnin parantamiseen sekä tuottaa kaksi uutta laajennosta sovelluskehykseen. Toteutetut laajennokset ovat pelin tallennus- ja latausjärjestelmä sekä järjestelmä pelin dialogi-ikkunoiden luomiseen ja hallitsemiseen. Näille laajennoksille toteutettiin myös perusteellinen tekninen dokumentaatio ja esimerkki laajennosten toiminnasta. Tämän lisäksi kumpikin laajennos testattiin toimeksiantajan kanssa sovittujen vaatimusten mukaisesti: laajennokset tulee testata Android- ja iOS-mobiililaitteilla sekä Windows-työpöytäympäristössä.

Sovelluskehyksellä tarkoitetaan luokka-, komponentti- ja/tai rajapintakokoelmaa, jota erikoistamalla voidaan saada aikaiseksi itsenäisiä sovelluksia. Erikoistamiseksi kutsutaan prosessia, jossa sovelluskehyksen päälle lisätään sovelluskohtainen koodi. (Koskimies & Mikkonen 2005, 187-188.) Tässä työssä kehitettävä kehys on toimeksiantajan yksityinen pelinkehityksessä käytettävä Unity-pelimoottorille tarkoitettu sovelluskehys. Sovelluskehystä on tarkoitus käyttää jokaisen uuden Unity-peliprojektin pohjana ja se on tarkoitettu olemaan toimeksiantajan ainoa kokonaisvaltainen kehys Unity-pelimoottorille.

Kuvio 1 on graafinen esitys kerrosarkkitehtuurista, joka havainnollistaa sovelluskehyksen käyttöä. Kerrosarkkitehtuurin määritelmä on, että se koostuu tasoista, jotka on järjestetty jonkin abstrahointiperiaatteen mukaan nousevaan järjestykseen (Koskimies & Mikkonen 2005, 126). Tässä tapauksessa abstrahointiperiaate on, että matalammalla olevat tasot ovat lähempänä laitetta kuin ihmistä. Unity-pelimoottori on kuviossa arkkitehtuurin alimmalla tasolla. Sovelluskehys rakennetaan tämän, eli alimman tason päälle ja sitä kuvaa keskimmäinen taso. Sovelluskehyksen päälle taas rakennetaan jokainen yksittäinen peli, jota edustaa korkein taso kuvassa ja joka on siis lähimpänä ihmistä käyttäjänä.



KUVIO 1. Kerrosarkkitehtuuri

Työn aihe valittiin, koska toimeksiantajalla oli tarve tehostaa ja nopeuttaa pelienkehitysprosessia sekä pelien prototyyppien tekemistä. Tämän lisäksi on huomattu, että toimeksiantajan pelien toteutuspuolen henkilöstö saattaa toteuttaa samoja ominaisuuksia tai toiminnallisuutta tarjoavia kokonaisuuksia useasti uudelleen erillissä peliprojekteissa. Sovelluskehyksen avulla tällaisten kokonaisuuksien uudelleen toteuttaminen vähentyy. Toimeksiantajan kanssa päätettiin yhdessä tämän opinnäytetyön aiheesta edellä mainittujen syiden perusteella.

2 UNITY-KEHITYSYMPÄRISTÖ

Unity on Unity Technologies:n kehittämä kokonaisvaltainen 3D- ja 2D-pelien sekä interaktiivisten kokemusten kehitysympäristö (Unity 2015). Itse Unity-termillä saatetaan tarkoittaa monia eri asioita eri yhteyksissä, kuten esimerkiksi Unity-editoria tai Unity-pelimoottoria. Tämän opinnäytetyön yhteydessä pelkkää Unity-termiä käytettäessä tarkoitetaan kuitenkin aina koko Unity-kehitysympäristöä. Unity-kehitysympäristön yksittäisistä osista mainitaan erikseen Unity-termiä käytettäessä.

Tärkein osa Unity-kehitysympäristöä on Unity-pelimoottori, jolla on mahdollista kehittää pelejä useille eri alustoille. Näitä alustoja ovat esimerkiksi mobiilikäyttöjärjestelmistä Applen iOS ja Googlen Android, henkilökohtaisten tietokoneiden käyttöjärjestelmistä Microsoftin Windows ja Applen Mac OS sekä pelikonsoleista Microsoftin Xbox One ja Sonyn Playstation 4. Yhteensä eri alustoja on tällä hetkellä 23. (Unity 2015.)

Unitysta on saatavilla kaksi eri versiota: Personal Edition ja Professional Edition. Personal Edition on harrastajille tarkoitettu täysin ilmainen versio, joka sisältää Unity-pelimoottorin täydet ominaisuudet. Professional Edition on ammattimaiseen käyttöön tarkoitettu versio, jonka hinta voidaan räätälöidä hankkijan omien tarpeiden mukaan. Professional Edition sisältää Personal Editioniin verrattuna lukuisia muita etuja ja Unity-kehitysympäristöön kuuluvia palveluja kuten analytiikkapalvelu Unity Analytics Pro sekä mahdollisuus käyttää Unity-pelimoottorin beta-versioita. (Get Unity 2015.)

Unity-pelimoottori on yksi suurimmista ja suosituimmista pelimoottoreista pelimoottorimarkkinoilla. Vuoden 2014 kolmannella neljänneksellä 29 % pelinkehittäjistä käytti ensisijaisena kehitystyökalunaan Unitya. Osuus on suurin kyseisestä vertailuryhmästä. Vertailun vuoksi 29 % pelinkehittäjistä käytti ensisijaisena kehitystyökalunaan natiiveja ratkaisuja ja Epic Games:n kehittämää Unreal Engine-pelimoottoria käytti ensisijaisena kehitystyökalunaan vain 3 % pelinkehittäjistä. (Wilcox & Voskoglou 2014, 25.)

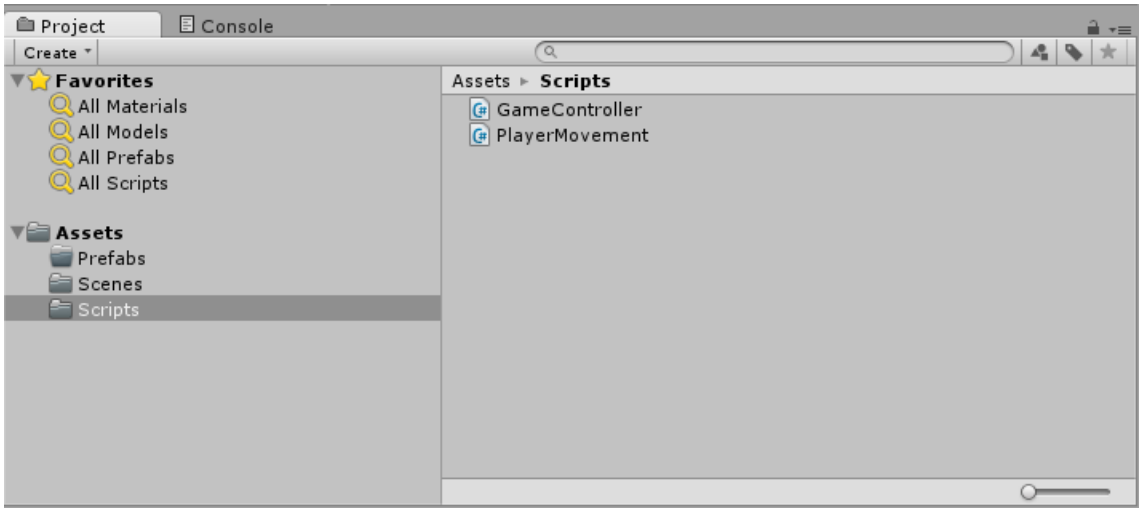
Unitylla tehtyjä kansainvälisesti menestyneitä pelejä ovat esimerkiksi tamperelaisen Colossal Orderin tekemä kaupunginrakennuspeli Cities: Skylines sekä Hipster Whalen kehittämä mobiilipeli Crossy Road (Made With Unity 2015).

2.1 Unity-editorin perusteet

2.1.1 Näkymät

Suurin osa Unitylla tehtävästä pelinkehitystyöstä tehdään Unity-editorin sisällä. Unity-editori koostuu erilaisista näkymistä. Näkymät ovat käyttöliittymäpaneeleita, joilla jokaisella on oma tarkoituksensa editorissa. Tärkeimpiä näkymiä ovat Scene-, Game-, Project-, Hierarchy- ja Inspector-näkymät.

Project-näkymässä voidaan selata kaikkia projektiin lisättyjä tiedostoja kansiorakenteessa. Jokaista Unity-projektiin kuuluvaa tiedostoa kutsutaan assetiksi. Unity-editorin ulkopuolelta Unity-projektiin tiedostojen lisäämistä kutsutaan tuonniksi. Jokaisella assetilla on oma tyyppinsä ja näitä ovat esimerkiksi 3D-mallit ja C#-skriptit. Myös kansiot ovat Unity-editorin projektihierarkiassa assetteja. Kuvassa 1 on Unity-editorin Project-näkymä, jossa näkyy esimerkkinä yksinkertainen kansiorakenne.

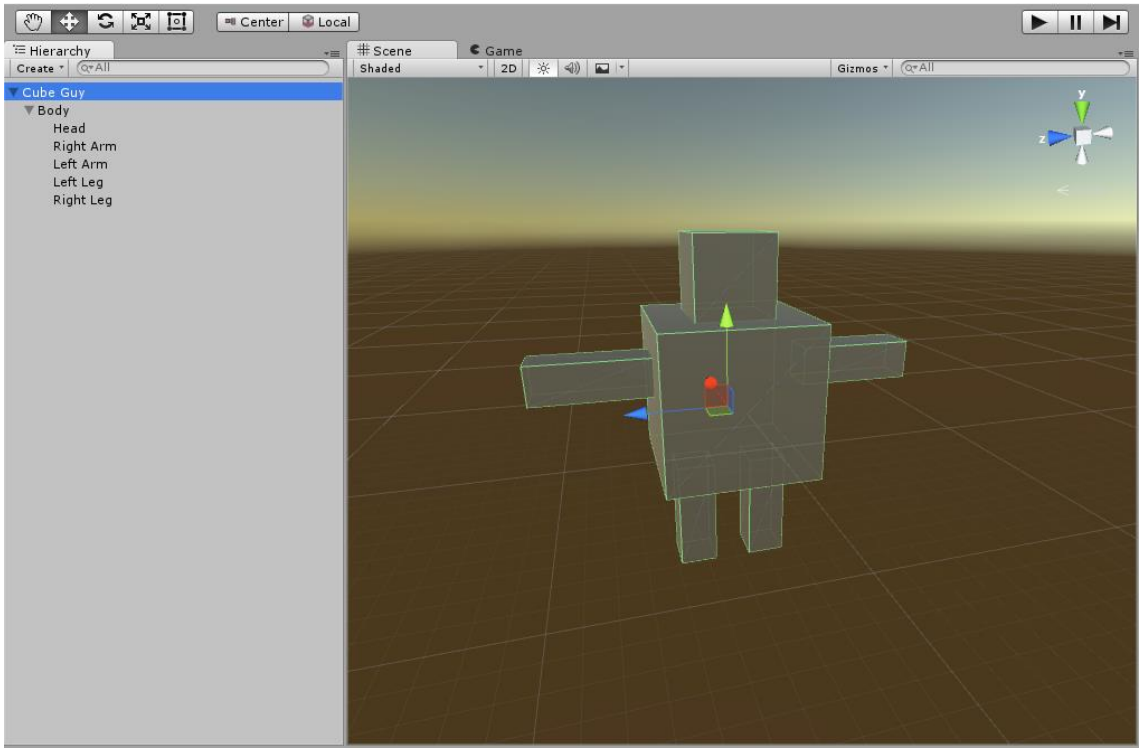


KUVA 1. Kuvakaappaus Unity-editorin Project-näkymästä (Unity 5.1.1f1 2015).

Scene-näkymässä näkyy sillä hetkellä aktiivisena oleva scene. Scene sisältää pelimaailman, jossa voidaan valita ja liikutella peliobjekteja, joita voivat olla esimerkiksi pelaajan hahmo ja ympäristö jossa hahmo liikkuu. Scenejä voi olla useita ja jokaista sceneä voidaan muokata erikseen. Peliobjekteja voi tallentaa projektiin asset-tiedostoina, jolloin niitä kutsutaan prefabeiksi. Prefabien tarkoitus on toimia valmiina malleina, joista voidaan luoda uusia peliobjekteja sceneen (Unity Manual: Prefabs 2015).

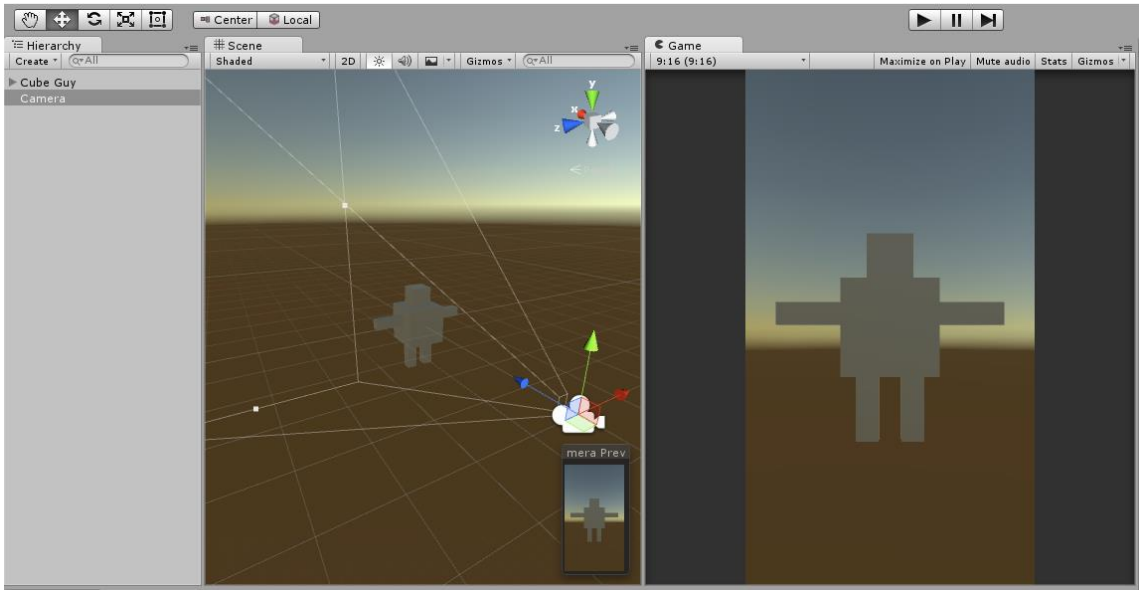
Hierarchy-näkymä sisältää kaikki scenessä olevat peliobjektit. Peliobjektit voivat olla näkymässä eriteltynä yksinään tai niitä voi olla useampi yhdessä hierarkiassa. Hierarkiassa peliobjektit voivat olla toistensa lapsia tai vanhempia. Peliobjekti joka on jonkin toisen peliobjektin lapsi, perii vanhemman liikkeen ja rotaation. Näin esimerkiksi scenessä hiearkian korkeimpaa peliobjektia liikuttamalla myös kaikki sen lapseksi luettavat peliobjektit liikkuvat samalla tavalla.

Kuvassa 2 on Unity-editorin Hierarchy- ja Scene-näkymät. Sceneen on luotu yksinkertainen kuutioista rakennettu neliraajaista olentoa muistuttava peliobjekti nimeltä Cube Guy. Cuby Guy rakentuu useista muista peliobjekteista, kuten vartaloa esittävästä Body-peliobjektista, joka on Cuby Guy-peliobjektin lapsi hierarkiassa. Tämän lisäksi Cuby Guy koostuu raajoista sekä päästä, jotka ovat taas Body-peliobjektin lapsia. Cube Guy-peliobjekti on tässä yhteydessä ylin hierarkiassa oleva peliobjekti, joten se on kaikkien muiden sen alla olevien peliobjektien vanhempi.



KUVA 2. Kuvakaappaus Unity-editorin Hierarchy- ja Scene-näkymistä (Unity 5.1.1f1 2015).

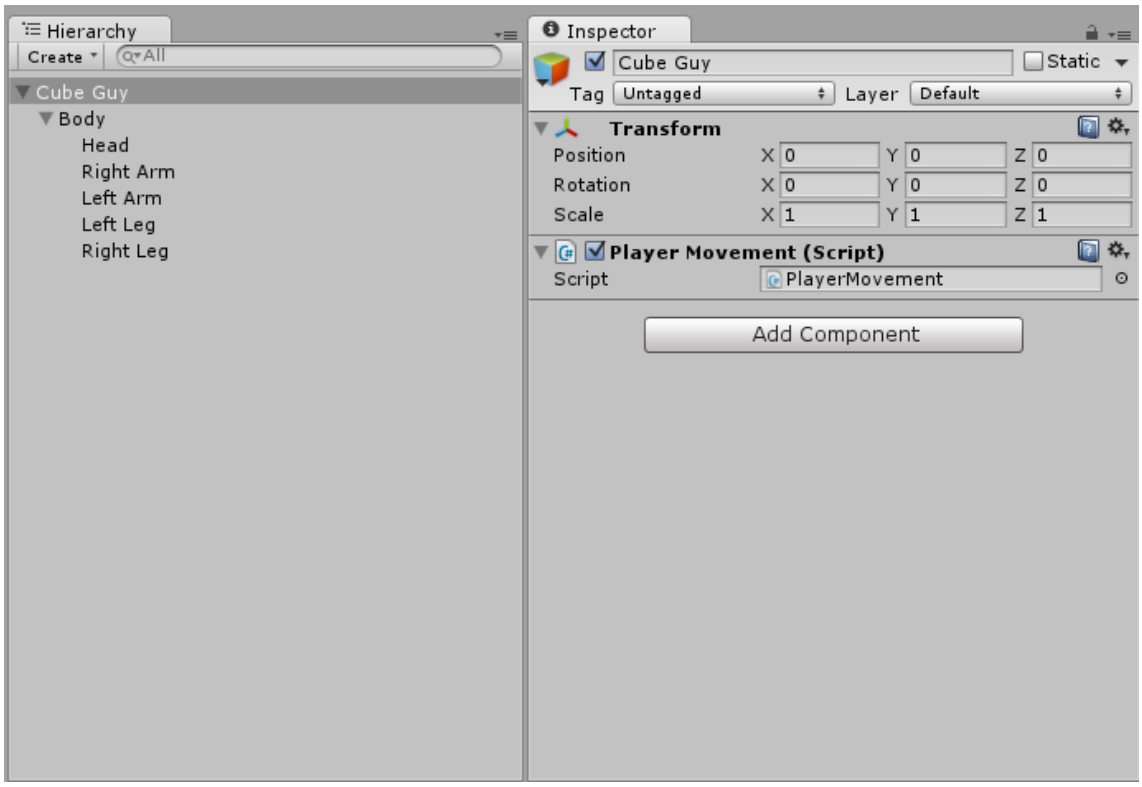
Game-näkymään renderoidaan pelissä olevien kameroiden näkymät. Kamerat ovat peliobjekteja jotka ovat pääasiassa tarkoitettu hallitsemaan sitä mitä pelaaja näkee kun hän pelaa peliä. Kuvassa 3 on Unity-editorin Inspector- ja Scene-näkymien lisäksi Game-näkymä. Sceneen on lisätty Camera-peliobjekti, jonka näkymä renderoidaan Game-näkymään.



KUVA 3. Kuvakaappaus Unity-editorin Hierachy-, Scene- ja Game-näkymistä (Unity 5.1.1f1 2015).

Inspector-näkymässä voidaan tarkastella sillä hetkellä valittuna olevaa peliobjektia tai assettia. Peliobjektit koostuvat komponenteista ja niitä voidaan lisätä, poistaa tai muokata Inspector-näkymän kautta. Komponenttien tarkoitus on tuoda peliobjekteille toiminnallisuutta.

Kuvassa 4 on Unity-editorin Hierarchy- ja Inspector-näkymät. Hierarchy-näkymässä on valittu aktiiviseksi Cube Guy-peliobjekti, joten sen ominaisuudet ja komponentit näkyvät Inspector-näkymässä. Jokaisella peliobjektilla on aina oletuksena joko RectTransform- tai Transform-komponentti, joka löytyy siis täten myös Cube Guy-peliobjektilta. Transform-komponentti hallitsee peliobjektin paikkaa, rotaatiota ja mittakaavaa pelimaailmassa. Peliobjektiin on myös itse lisätty Player Movement-komponentti, jonka tarkoituksena on hallita peliobjektin liikkumista.



KUVA 4. Kuvakaappaus Unity-editorin Hierarchy- ja Inspector-näkymistä (Unity 5.1.1f1 2015).

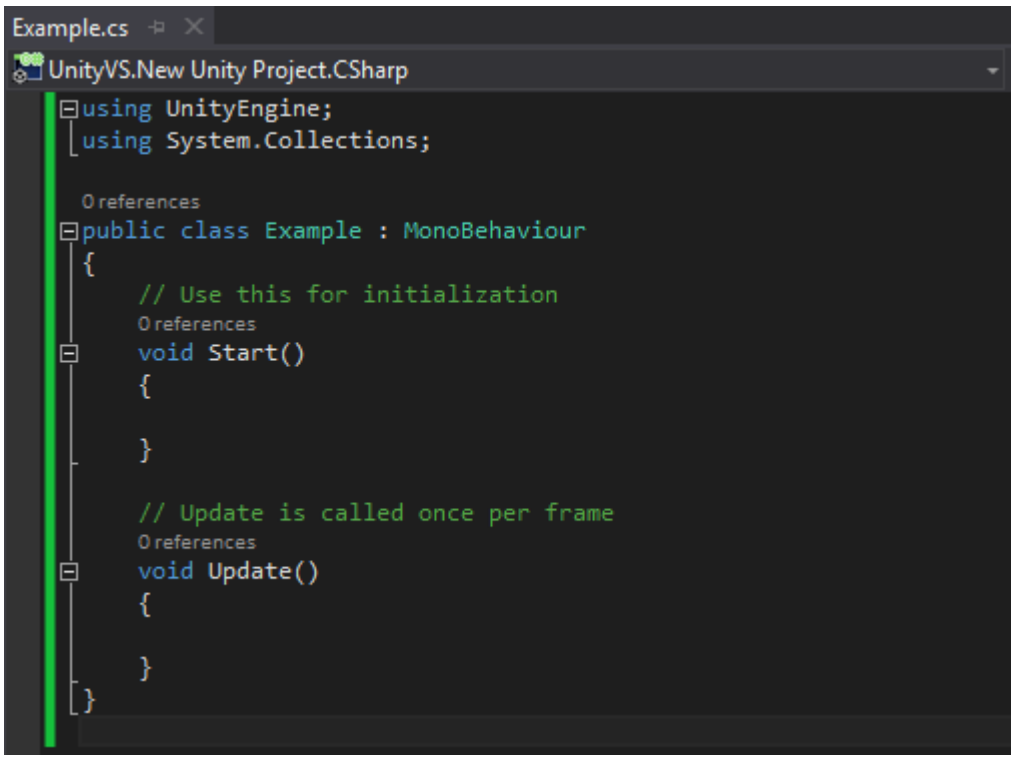
2.1.2 Ohjelmointi

Unity-pelimoottori tukee kahta ohjelmointikieltä, joilla pelejä voidaan tehdä: C# ja UnityScript (Unity Manual: Creating and Using Scripts 2015). C# on Microsoftin

kehittämä oliopohjainen ohjelmointikieli ja UnityScript on Unity-pelimoottoria varten suunniteltu ohjelmointikieli, joka on hyvin samankaltainen web-kehityksessä käytettävän JavaScript-ohjelmointikielen kanssa.

Tiedostoja jotka sisältävät C#- tai UnityScript-ohjelmointikieltä, kutsutaan Unityn sisällä skripteiksi. Skriptejä luodaan Unity-editorin sisällä ja niitä voidaan muokata millä tahansa tekstieditorilla. Oletuksena Unity käyttää MonoDevelop-tekstieditoria. Skriptejä voidaan liittää peliobjekteihin komponentteina, jolloin kyseessä oleva peliobjekti toteuttaa skriptissä olevan toiminnallisuuden.

Kuvassa 5 on tekstieditorissa näkyvä skripti, joka on luotu Unity-editorissa. Skriptin nimi on Example.cs, se on C#-skripti ja koska se on luotu Unity-editorissa, se sisältää oletuksena kuvassa jo olevat rivit. Kaikki skriptit periytyvät aina oletuksena MonoBehaviour-luokasta. C#-skriptien ei ole kuitenkaan pakko periä MonoBehaviour-luokkaa ja siitä periytyminen voidaan halutessa poistaa. Skriptin on kuitenkin periydyttävä MonoBehaviour-luokasta, jotta skriptin voi lisätä peliobjektiin komponenttina. MonoBehaviour-luokka tarjoaa metodeja ja tapahtumia, joiden avulla vuorovaikutus peliobjektien kanssa tapahtuu. Skriptistä löytyvät Start- ja Update-metodit ovat tapahtumametodeja, joita kutsutaan aina Unity-pelimoottorin toimesta tietyssä järjestyksessä. (Unity Scripting API: MonoBehaviour 2015.)



KUVA 5. Kuvakaappaus Unity-editorissa luodusta Example-skriptistä (Microsoft Visual Studio Ultimate 2013 2015).

3 SOVELLUSKEHYKSEN KÄYTTÖ JA HALLINTA

3.1 Sovelluskehyksen nykytilanne

Toimeksiantajalla oli jo ennen tämän opinnäytetyön tekemisen aloittamista olemassa oleva sovelluskehys Unity-pelimoottorille. Sovelluskehys on itsenäinen Unity-projekti ja sitä on tarkoitus käyttää jokaisen uuden peliprojektin pohjana.

Sovelluskehyksen lähdekoodin hallintaan käytetään avoimen lähdekoodin versionhallintajärjestelmää Git:iä ja kehyksen Git-tietovarasto sijaitsee ulkoisesti Bitbucket web-palvelussa. Jokaisella toimeksiantajan pelinkehittäjällä on oikeus lisätä sovelluskehykseen omia lisäyksiä tai muutoksia. Sovelluskehykseen ei ole olemassa yhteistä käytäntöä tai ohjeistusta sen hallinnointiin tai uusien laajennosten ja/tai työkalujen lisäämiseen.

3.2 Sovelluskehyksen hallinnan parantaminen

3.2.1 Yhtenäinen käytäntö ja ohjeistus

Koska sovelluskehyksen hallinnoimiseen ei ole olemassa yhtenäistä käytäntöä, ohjeistusta ja/tai rakennetta, siitä löytyviä laajennoksia ja työkaluja voi olla hankala löytää assettien seasta. Suurempi ongelma on myös se, ettei mistään saa nopeasti selville, mitä laajennoksia ja työkaluja sovelluskehyksestä edes löytyy. Nämä kyseiset ongelmat laajenevat suuresti sitä mukaa kun sovelluskehys kasvaa. Tästä syystä sovelluskehykseen olisi hyvä saada yhtenäinen käytäntö ja ohjeistus ennen kuin se kasvaa hallitsemattomaksi.

Ohjeistuksessa olisi hyvä olla mainittuna ja/tai kuvattuna ainakin seuraavat asiat: sovelluskehyksessä valmiina olevat laajennokset ja työkalut, sovelluskehyksen kansiorakenne, ohjeistus kansioiden ja tiedostojen nimeämiseen, ohjeistus nimiavaruuksien käytöstä, ohjeistus sovelluskehyksen käyttöönotosta sekä ohjeistus siitä kuinka sovelluskehykseen lisätään uusia tai muokataan vanhoja laajennoksia ja

työkaluja. Ohjeistuksen voisi lisätä esimerkiksi suoraan sovelluskehyksen lähdetiedostojen sekaan readme-tiedostona. Readme-tiedostoksi kutsutaan yleensä ohjelmistojen mukana tulevaa tekstitiedostoa, joka pitää sisällään dokumentaatiota ohjelmistosta. Bitbucketissa voidaan suoraan myös muokata ja tarkastella muotoiltuja readme-tiedostoja. Tämän raportin liitteinä olevat readme-dokumentaatiot toteutetuille laajennoksille on samalla tavalla lisätty kyseisten laajennosten yhteyteen sovelluskehyksen ulkoiseen Git-tietovarastoon.

3.2.2 Sovelluskehyksen kansiorakenne

Sovelluskehyksessä kansiorakenne on tällä hetkellä sekava ja epäyhtenäinen. Epäyhtenäinen kansiorakenne kuluttaa turhaa aikaa ja työtä, joka menee sovelluskehyksen rakenteen ymmärtämiseen ja haluttujen tiedostojen etsimiseen. Sovelluskehykseen olisi siis hyvä rakentaa selkeä ja yhtenäinen kansiorakenne. Sellaisen saa helposti rakennettua erottamalla sovelluskehyksessä olevia yhtenäisiä kokonaisuuksia ja sijoittamalla näitä omiin kansioihinsa.

Koska sovelluskehys on tarkoituksena tuoda jokaisen uuden peliprojektin pohjaksi, se olisi hyvä erottaa valmiiksi kansiorakenteessa sen päälle tehtävästä projektista. Tällä hetkellä sovelluskehyksen kansiot ja tiedostot ovat suoraan Unity-projektin pääkansion alla, johon yleisesti sijoitetaan suoraan myös kaikki peliprojektien tiedostot ja kansiot. Sovelluskehystä tuodessa tai käyttöön otettaessa peliprojektiin, siihen kuuluvat tiedostot menevät siis väistämättä sekaisin peliprojektin tiedostojen kanssa. Sovelluskehyksen erottaminen peliprojektista tapahtuu yksinkertaisimmin luomalla sovelluskehykselle oman kansionsa, johon sijoitetaan kaikki sovelluskehykseen kuuluvat tiedostot ja kansiot. Sovelluskehyksen erottelun myötä koko peliprojektin kansiorakenne selkeytyy sekä sovelluskehyksessä olevien laajennosten ja työkalujen päivitys käy tätä myötä pelinkehityksen aikana helpommaksi.

Tämän lisäksi sovelluskehyksen sisäisessä kansiorakenteessa olisi hyvä olla selkeästi eroteltuna jokainen erillinen kokonaisuus. Yksi kokonaisuus voi olla joko yksi laajennos, joka rakentuu monesta assetista tai se voi olla ryhmä skriptejä, joilla ei suoraan ole mitään yhteyttä toisiinsa, mutta liittyvät kuitenkin läheisesti johonkin tiettyyn toiminnallisuuteen tai pelin osaan, esimerkiksi käyttöliittymään. Tällaisen

yhden kokonaisuuden kaikki assetit olisi hyvä olla sijoitettuna omassa kansiossaan sovelluskehys-kansion alla. Syynä kokonaisuuksien erotteluun on päivitettävyys ja kansiorakenteen selkeys sovelluskehyksen sisällä. Tietyn kokonaisuuden assetit eivät ole sekaisin kaikkien muiden assettien kanssa, joten niiden löytäminen helpottuu.

Kansiorakenteen miettimisen yhteydessä on otettava huomioon Unity-projektissa olevat erikoiskansiot, joilla on jokaisella oma tarkoituksensa ja toiminnallisuutensa Unity-projektin sisällä. Erikoiskansioiden toiminnallisuus on sidottu suoraan niiden nimiin. Näitä erikoiskansioita ovat Assets-, Editor-, Editor Default Resources-, Gizmos-, Plugins-, Resources-, Standard Assets-, StreamingAssets-, ja WebPlayerTemplates-kansiot. (Unity Manual: Special Folder Names 2015.)

Assets-kansio on pääkansio, joka sisältää kaikki Unity-projektin assetit. Unity-editorissa ei ole suoraa tapaa tarkastella Unity-projektiin kuuluvia muita kansioita kuin Assets-kansiota. Nämä muut kansiot ja niissä olevat tiedostot eivät muutenkaan ole sellaisia, joiden sisältöä käyttäjän olisi tarvetta suoraan tiedostosta lukea tai muuttaa. Assets-kansio luodaan aina automaattisesti kun uusi Unity-projekti luodaan. Assets-kansio ja sen sisältö näkyvät Unity-editorissa Project-näkymässä.

Editor-kansio on tarkoitettu kaikille skripteille, jotka tuovat jotakin toiminnallisuutta itse Unity-editoriin pelin kehityksen aikana. Nämä skriptit eivät ole mukana ajonaikana lopullisessa pelissä. Editor-kansioita voi olla projektissa useita ja ne voivat sijaita missä tahansa Assets-kansion alla.

Plugins-kansio on tarkoitettu pitämään sisällään liitännäisiä, jotka ovat tuotettu Unity-editorin ulkopuolella. Näiden liitännäisten tarkoituksena on yleensä laajentaa Unity-editorin käytössä olevia toimintoja. Liitännäisten ei tarvitse olla tehty Unity-pelimoottorin tukemalla ohjelmointikielellä, joilla pelejä tehdään Unity-editorin sisällä, vaan ne voivat olla esimerkiksi tehty C++-ohjelmointikielellä. Liitännäinen voi tuoda esimerkiksi tuen Applen iOS-alustan pelin sisäisten ostoksien tekemiseen ja hallintaan. Plugins-kansioita voi olla Editor-kansion mukaisesti useista ja ne voivat sijaita missä tahansa Assets-kansion alla.

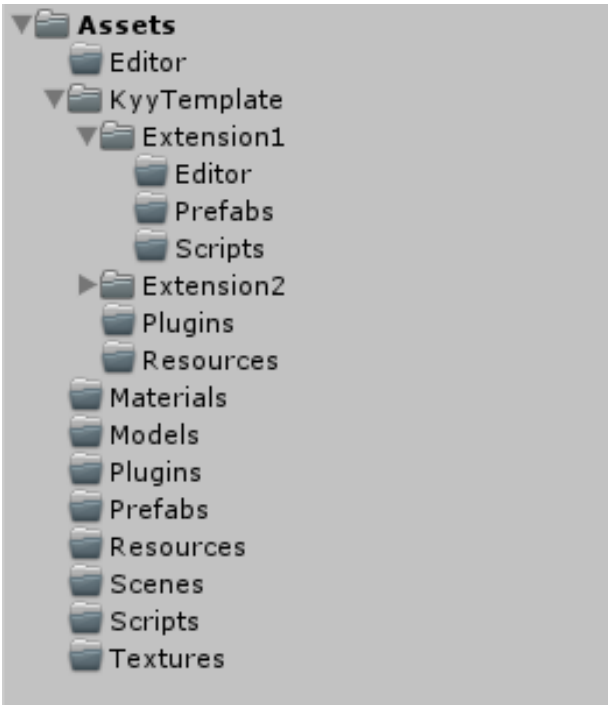
Resources-kansio on tarkoitettu sisältämään assetteja, joita voi ladata kansiosta ajonaikaisesti. Normaalisti kaikki assetit lisätään ilmentymiksi sceneä rakennettaessa,

mutta joitakin tiedostomuotoja on mahdotonta tai epäkäytännöllistä lisätä sceneihin ilmentymiksi. Tällaisia tiedostomuotoja ovat esimerkiksi tekstitiedostot. Tekstitiedostoja voidaan pelinkehityksessä käyttää esimerkiksi sisältämään kaikki pelissä käytetyt lokalisoidut merkkijonot. Resources-kansioita voi olla olemassa useita ja ne voivat sijaita missä tahansa Assets-kansion alla.

Editor Default Resources-, Gizmos-, Standard Assets-, StreamingAssets- ja WebPlayerTemplates-kansiot ovat toiminnallisuudeltaan vähemmän tärkeitä kuin edellä mainitut kansiot. EditorDefaultResources-kansio on toiminnaltaan samankaltainen kuin Resources-kansio, mutta sieltä voidaan ladata assetteja vain kehitysvaiheessa, eikä nämä assetit ole siis mukana lopullisessa pelissä. Gizmos-kansio on tarkoitettu sisältämään Unity-editorissa yksityiskohtien visualisointia helpottavaa grafiikkaa. Standard Assets-kansioon Unity-editori sijoittaa kaikki projektiin tuodut Unity-editorin mukana tulleet assetit. StreamingAssets-kansioon sijoitetaan tiedostoja, joiden halutaan olevan erillisessä tiedostossaan lopullisessa pelissä, sen sijaan että ne olisivat rakennettu sisään Unity-editorin rakentamaan ohjelmatiedostoon. WebPlayerTemplates-kansio liittyy vain web-pohjaisten pelien tekemiseen ja sinne voidaan sijoittaa kustomoituja web-sivuja, joihin on tarkoitus upottaa lopullinen peli.

Erikoiskansioihin liittyy myös skriptien kääntämisjärjestys. Unity-pelimoottorissa skriptit käännetään neljässä eri vaiheessa. Kääntämisvaihe riippuu siitä missä kansiossa skriptit Unity-projektissa sijaitsevat. Ensimmäisessä vaiheessa käännetään kaikki skriptit Standard Assets- ja Plugins-kansioissa. Toisessa vaiheessa käännetään kaikki skriptit Standard Assets/Editor- ja Plugins/Editor-kansioissa. Kolmannessa vaiheessa käännetään kaikki muut skriptit, jotka eivät sijaitse Editor-kansioissa ja lopuksi neljännessä vaiheessa käännetään kaikki loput skriptit, jotka sijaitsevat Editor-kansioissa. Tietyn kääntämisvaiheen skriptit voivat ainoastaan viitata kaikkien aiempien kääntämisvaiheiden skripteihin. Esimerkiksi toisen kääntämisvaiheen skriptit voivat siis viitata ensimmäisen kääntämisvaiheen skripteihin, mutta eivät voi viitata kolmannen eikä neljännen kääntämisvaiheen skripteihin. Kääntämisvaiheet ovat olemassa, jotta eri ohjelmointikielillä tehdyt skriptit voivat viitata toisiinsa ja jotta editori-skriptit saadaan erotettua muista skripteistä. (Unity Manual: Special Folders and Script Compilation Order 2015.) Skriptien kääntämisjärjestys on otettava huomioon sovelluskehyksen kansiorakennetta miettiessä, koska kaikki skriptit eivät välttämättä toimi kaikissa kansiossa.

Kuvassa 6 on esimerkki sovelluskehyksen mahdollisesta kansiorakenteesta. Kansiorakenteessa on eritelty erikseen sovelluskehys polkuun Assets/KyyTemplate/. KyyTemplate-nimellä tarkoitetaan tässä yhteydessä sovelluskehyksen nimeä. Tämän lisäksi sovelluskehyksessä olevat kokonaisuudet ovat eritelty erikseen, josta esimerkkinä Extension1-niminen laajennos joka löytyy polusta Assets/KyyTemplate/Extension1/. Koska ohjelmistokehyksen tarkoituksena on toimia jokaisen peliprojektin pohjana, kansiorakenteeseen on myös lisätty Unity-projekteissa yleisimmin käytettyjä kansioita. Näitä kansioita ovat kaikki Assets/KyyTemplate/-polun ulkopuolella olevat kansiot.



KUVA 6. Kuvakaappaus sovelluskehyksen kansiorakenteesta (Unity 5.1.1f1 2015).

3.2.3 Kansioiden nimeämiskäytännöt

Sovelluskehyksen kansiorakenteeseen Unity-projektissa yleisimmin käytettyjen kansioiden lisäämisen tarkoituksena on luoda kansioille yhtenäiset nimeämiskäytännöt. Nimeämiskäytännöt ovat käytännössä ohjeita siitä, kuinka kansiot tulisi nimetä ja missä kansiopolussa ne sijaitsevat. Kansioiden nimet tulisi olla sisältöään kuvaavia ja niiden sijainti kansiorakenteessa tulisi olla looginen.

Kansioiden nimeämiskäytäntöjen avulla säästetään aikaa ja vältetään turhaa työtä, kun jokaisessa peliprojektissa samat assetit löytyvät saman polun alta, eikä niitä tarvitse erikseen etsiä. Uutta projektia aloitettaessa yleisemmin käytetyt kansiot olisivat jo sovelluskehyksen lisäyksen jälkeen valmiina, joten ne olisivat pakosti joka projektissa samannimiset ja samassa kansiopolussa. Yleisimpien kansioiden lisäämisen yhteydessä sovelluskehykseen on otettava huomioon ettei Git-versionhallintajärjestelmä oletuksena ota huomioon tyhjiä kansioita. Kyseiset kansiot olisivat luonnollisesti tyhjiä, koska niihin tarkoitettu sisältö lisätään vasta peliprojektin aikana. Asia voidaan kuitenkin korjata lisäämällä jokaiseen tyhjään kansioon esimerkiksi tyhjä .keep-tiedosto. Kun tyhjään kansioon lisätään mikä tahansa tiedosto, Git ottaa automaattisesti myös kansion huomioon.

3.2.4 Nimiavaruudet

Nimiavaruus on ohjelmointikäsite ja C#-ohjelmointikielessä sille löytyy oma avainsanansa koodissa. Nimiavaruuksien tarkoituksena on organisoida suuria koodiprojekteja (C# Programming Guide: Namespaces 2015). Käytännössä tämä tarkoittaa kooditiedostoihin lisättävää merkkijonoa, jolla luokitellaan tiedostot eri kokonaisuuksiin. Nimiavaruuksien avulla vältetään nimikonflikteja, joilla tarkoitetaan C#-ohjelmointikielessä ongelmatilannetta, jossa samannimisiä tyyppejä löytyy samasta nimiavaruudesta (mukaan lukien tyypit joille ei ole määritelty nimiavaruutta). Tyypit voivat olla tässä tapauksessa esimerkiksi luokkia tai rajapintoja.

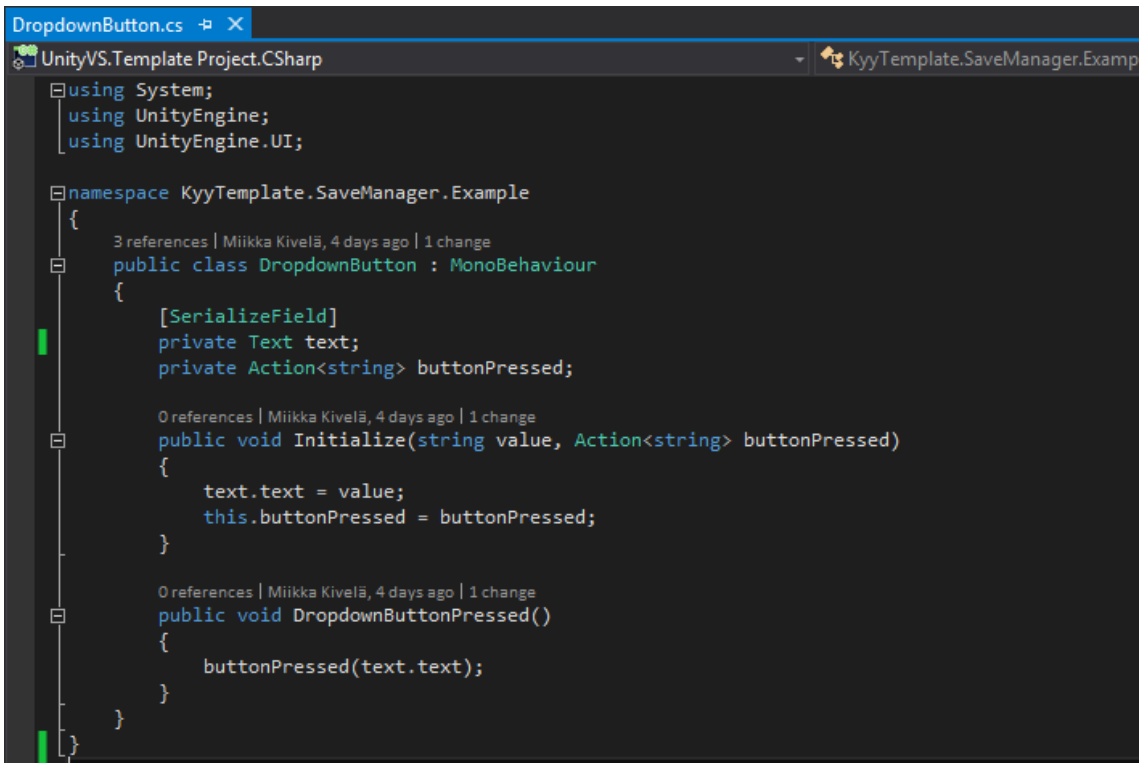
Unity-editorissa ei ole mahdollisuutta luoda samannimisiä tiedostoja, mutta skripteihin voidaan esimerkiksi tekstieditorissa manuaalisesti vaihtaa luokan nimi. Lähes jokainen ohjelmistokehittäjä tietää kuitenkin välttää luokkien samannimiseksi nimeämistä. Ongelma kuitenkin saattaa ilmetä tilanteessa, jossa usea kehittäjä työskentelee samassa projektissa. Kaksi eri kehittäjää saattavat tehdä toisistaan tietämättä kaksi samannimistä luokkaa samaan projektiin ja pistää nämä luokat yhteiseen tietovarastoon versionhallinnassa. Versionhallinta ei välitä, vaikka projektista löytyisikin kaksi samannimistä luokkaa eri tiedostopoluissa.

Jokaiseen sovelluskehyksen skripti-tiedostoon olisi siis hyvä lisätä kuvaava nimiavaruus, ei vain pelkästään nimikonfliktien välttämiseksi, vaan myös

skriptitiedostojen organisoinnin helpottamiseksi. Nimiavaruuksia käyttämällä parannetaan sovelluskehyksen laajennettavuutta ja selkeyttä.

Sovelluskehyksen yhteydessä nimiavaruuksien nimeämiskäytännöissä voisi käyttää osittain skriptin tiedostopolkua projektissa. Esimerkiksi jos skripti kuuluu Localization-laajennokseen ja on Localization-kansion alla kansiorakenteessa, sen nimiavaruudeksi tulisi KyyTemplate.Localization. Kuvassa 6 sovelluskehyksen kansiorakenteessa Extension1-nimisen laajennokseen sisältyvien skriptien nimiavaruudeksi tulisi täten KyyTemplate.Extension1.

Kuvassa 7 on DropdownButton-skripti. Kyseinen skripti on osa sovelluskehykseen toteutettua pelin tallennus- ja latausjärjestelmää ja sillä ei ole mitään tekemistä itse järjestelmän toiminnallisuuden kanssa, vaan se on osa järjestelmän testisceneä. Skriptin nimiavaruudeksi on määritelty KyyTemplate.SaveManager.Example ja se sijaitsee sovelluskehyksessä SaveManager/Example/Scripts/-polussa. KyyTemplate on tässä tapauksessa ohjelmistokehyksen nimi. Kaikki Unity-projektin skriptit on tapana laittaa aina Scripts-kansioiden alle, jotta ne olisivat erillä muiden tyyppisistä asseteista. Koska voidaan olettaa kaikkien skriptien löytyvän Scripts-nimisen kansion alta, on sitä turha lisätä nimiavaruuteen organisoinnin helpottamiseksi.



KUVA 7. Kuvakaappaus DropdownButton-skriptistä (Microsoft Visual Studio Ultimate 2013 2015).

3.2.5 Sovelluskehyksen ylläpitäjä

Hyvä tapa hallita sovelluskehystä on nimetä sille ylläpitäjä, joka pitää huolen että kaikki sovelluskehykseen tehdyt muokkaukset ja lisäykset noudattavat sovelluskehykselle luotua ohjeistusta. Käytännössä ylläpito onnistuisi helposti niin, että vain ylläpitäjällä on oikeus tehdä muutoksia sovelluskehyksen versionhallinnan ulkoiseen tietovarastoon. Muut kehittäjät tekisivät vetopyyntöjä, kun he ovat tehneet ohjelmistokehykseen muutoksia. Sovelluskehyksen ylläpitäjä joko hyväksyy muutokset ja yhdistää muutokset sovelluskehykseen tai pyytää kehittäjää korjaamaan tai muuttamaan koodia. Bitbucketissa voi helposti myös keskustella jokaisesta yksittäisestä vetopyynnöstä.

3.2.6 Unitypackage

Unitypackage on tiedostomuoto, johon voidaan pakata useita assetteja ja joka voidaan tuoda kokonaan tai osittain Unity-projektiin Unity-editorin kautta. Sovelluskehyksestä tehty unitypackage olisi hyödyksi, koska kehittäjä voisi valita mitä kaikkia osia

sovelluskehyksestä hän haluaa tuoda peliprojektiinsa sen sijaan, että sovelluskehys otetaan aina kokonaisuudessaan uuden Unity-peliprojektin pohjaksi. Sovelluskehyksen ottaminen suoraan uuden peliprojektin pohjaksi saattaa johtaa siihen, että projektiin tulee turhia laajennoksia ja tiedostoja joita ei itse projektissa käytetä lainkaan. Unitypackage-tiedostoja voi luoda helposti suoraan Unity-editorin sisältä. Jotta unitypackage-tiedosto olisi aina ajan tasalla, se täytyisi luoda aina uudestaan kun sovelluskehyksen ulkoiseen Git-tietovarastoon tehdään uusia valmiita muutoksia.

4 LAAJENNOSTEN TOTEUTUS SOVELLUSKEHYKSEEN

4.1 Käytetyt työkalut

4.1.1 Tekstieditori

Tekstieditorina laajennosten toteutuksessa käytettiin Microsoftin Visual Studio Ultimate 2013-kehitysympäristöä. Unity-editorin kanssa tehokkaamman vuorovaikutuksen takia Visual Studioon on myös liitetty Visual Studio 2013 Tools for Unity-laajennos, joka tuo Visual Studioon mukanaan esimerkiksi toimintoja ohjelmointivirheiden etsimiseen ja korjaamiseen skripteistä sekä Unity-editorin projektihierarkian sellaisenaan kun se editorissa on esitetty (Build Unity Games with Visual Studio 2015).

4.1.2 Versionhallinta

Versionhallintajärjestelmänä laajennosten hallinnassa käytettiin Git:iä. Git valittiin versionhallintajärjestelmäksi, koska jo olemassa oleva sovelluskehys käyttää sitä. Git:iä käytettiin Atlassianin kehittämän asiakasohjelman SourceTreen kautta.

4.2 Pelin tallennus- ja latausjärjestelmä

Lähes jokaisessa pelissä tallennetaan jollakin tavalla pelaajan pelissä tekemää edistystä johonkin tallennuspaikkaan. Pelaajan lopetettua pelin ja seuraavalla kerralla taas avatessaan sen, peli lataa pelaajan tekemän edistyksen ja näin pelaaja voi jatkaa siitä mihin viime kerralla pelissä jäi. Pelaajan tekemä edistys voi olla lähes mitä vain pelistä riippuen. Edistys voi olla esimerkiksi pelaajan hahmon taso tai se kuinka monta kolikkoa pelaaja on kerännyt pelin aikana.

Unity-pelimoottorissa ei ole sisäänrakennettuna pelin tallennus- ja latausjärjestelmää. Pelimoottorin rajapinnassa on olemassa PlayerPrefs-luokka, jonka avulla voidaan tallentaa yksinkertaisia avain-arvo-pareja. Avain-arvo-parit eivät kuitenkaan ole

käytännöllinen tapa tallentaa peliä, koska tallennettavan tiedon määrä on yleensä peleissä niin suuri, että sen hallinta avain-arvo-parein on hyvin epäkäytännöllistä. Lähes jokaiselle tallennettavalle asialle ja tilanteelle tulisi luoda oma avain-arvo-parinsa ja niitä jouduttaisiin tässä tapauksessa tallentamaan ja lataamaan aina yksi kerrallaan. PlayerPrefs-luokka tukee muutenkin vain numeraali- ja merkkijonotyyppejä, joten monimutkaisempia tietorakenteita, kuten listoja ei voi luokan avulla suoraan tallentaa. PlayerPrefs-luokka onkin tarkoitettu enimmäkseen pelaajan tekemien asetusten tallentamiseen pelissä. Asetuksia voivat olla esimerkiksi äänentaso tai se mitä kieltä pelissä käytetään.

Pelin tallennus- ja latausjärjestelmä joudutaan siis Unity-pelimoottorin yhteydessä luomaan itse tai käyttämään jotain jo olemassa olevaa laajennosta tähän tarkoitukseen. Koska tällaisia ilmaisia kattavia järjestelmiä ei ollut olemassa, pelin tallennus- ja latausjärjestelmä päätettiin tehdä itse.

4.2.1 Tallennettava tieto

Pelin tietoja tallennettaessa on tärkeää miettiä, mitä tietoa tallennetaan ja missä muodossa se tallennetaan. Koska jokainen peli on erilainen, myös tallennettava tieto on jokaisessa pelissä erilaista. Jokaisesta pelistä löytyvää yhtenäistä rakennetta tallennettavalle tiedolle ei voida siis määritellä.

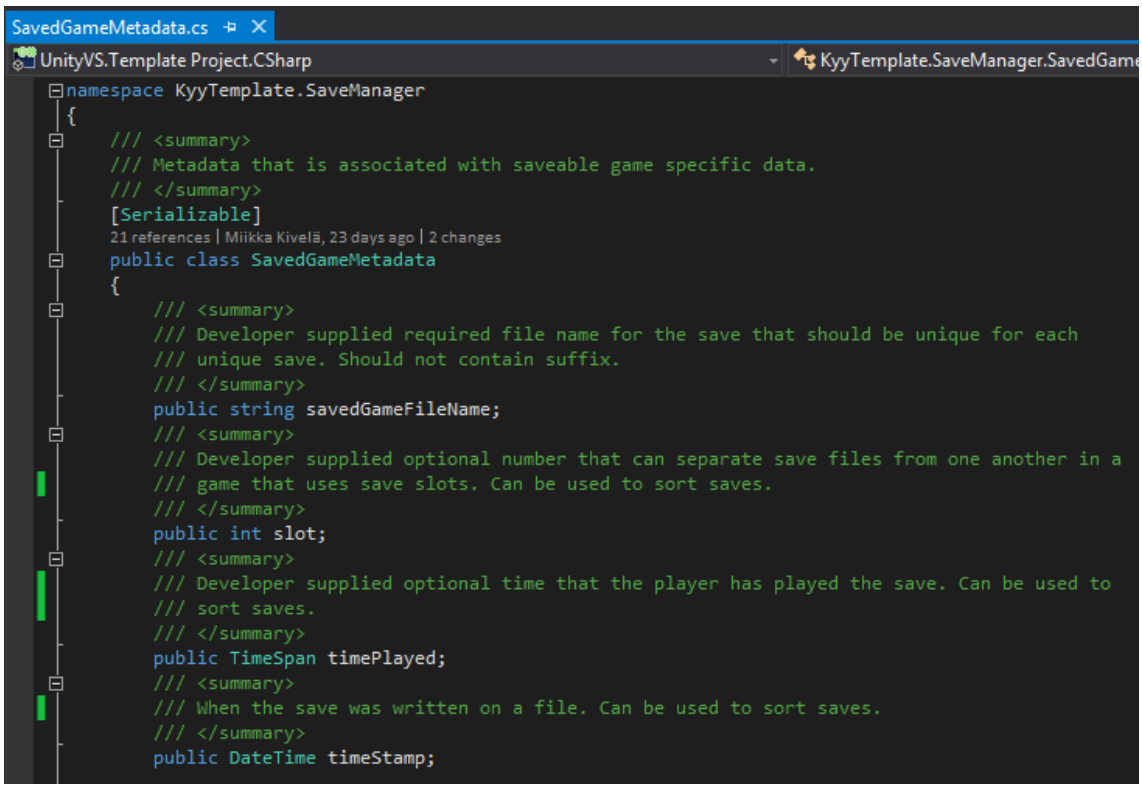
Kaikesta tallennettavasta tiedosta voidaan kuitenkin määritellä siihen liittyvää metatietoa. Tätä metatietoa on esimerkiksi se, kuinka paljon ajallisesti pelaaja on pelannut kyseistä tallennusta ja se milloin hän on tallennuksen tehnyt. Kaikki metatieto ei kuitenkaan aina koske kaikkia mahdollisia pelejä. Joissakin peleissä peli itse hoitaa kokonaan pelaajan edistyksen tallentamisen, eikä tässä tapauksessa pelaaja voi itse tallentaa peliä lainkaan. Tällöin tieto siitä milloin peli on tallennettu ei ole pelaajan kannalta välttämättä oleellista. Metatiedon onkin tarkoitus olla sellaista tietoa, mikä on yhteistä usealle eri peleissä olevalle tallennettavalle tiedolle, mutta ei välttämättä kaikille.

Sovelluskehykseen toteutettuun pelin tallennus- ja latausjärjestelmän perustaksi otettiin siis, että järjestelmä tallentaa pelikohtaisen tiedon lisäksi myös metatietoa

tallennuksesta. Järjestelmä erottelee nämä tiedot tallentamalla ne eri tiedostoihin. Tästä erottelusta on yksi suuri etu järjestelmän tehokkuuden kannalta. Pelikohtainen tieto saattaa sisältää todella suuren määrän erilaista tietoa kun taas siihen liittyvä metatieto sisältää vain muutaman yksinkertaisen muuttujan verran tietoa. Pelikohtaisen tiedon tallennustiedosto saattaa siis olla hyvinkin suuri. Ohjelmoinnissa tiedostoihin kirjoittaminen ja niistä lukeminen ovat yleensä raskaimpia yksittäisiä prosesseja ja mitä enemmän tietoa käsitellään, sitä enemmän kuluu aikaa.

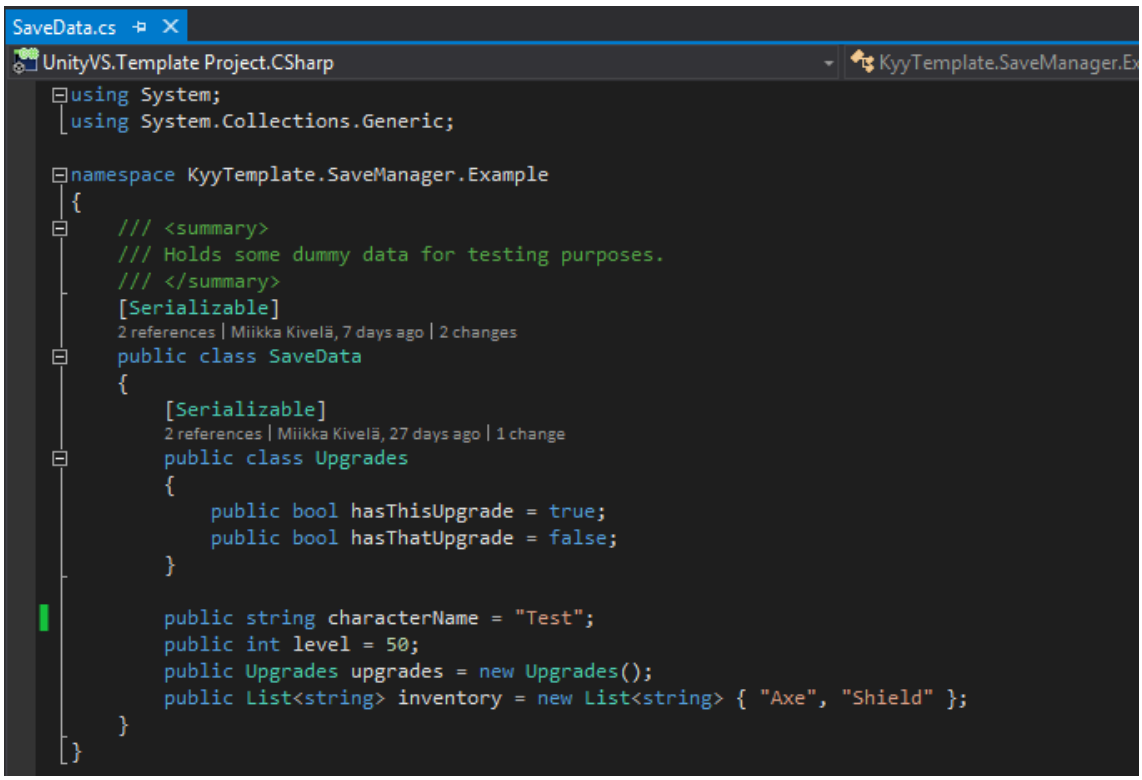
Kuvitellaan siis että on peli, jossa on kolme tallennuspaikkaa, joista pelaaja valitsee yhden tallennuksen jota hän haluaa pelata. Sen sijaan, että ladattaisiin kaikki tallennustiedostot, ladataan muistista vain niihin liittyvät pienet metadata-tiedostot. Pelaaja valitsee metadatan perusteella tallennustiedoston, jota hän haluaa pelata ja tämän jälkeen järjestelmä lataa kyseisen tallennustiedoston. Järjestelmä vältti tässä tapauksessa turhaa työtä: jos järjestelmä olisi ladannut alkuun kaikki tallennustiedostot, kaksi näistä tiedostoista olisi ladattu turhaan.

Kuvassa 8 on osa SavedGameMetadata-luokkaa, joka pitää sisällään toteutetussa pelin tallennus- ja lataamisjärjestelmässä tietorakenteen metatiedolle. Luokassa olevat muuttujat ovat metatietoa, jonka järjestelmä tallentaa aina pelikohtaisen tiedon lisäksi. Metatietoa ovat tiedoston nimi, johon pelikohtainen tieto tallennetaan, numeraalinen esitys tallennuspaikasta johon tieto tallennetaan, kuinka paljon pelaaja on ajallisesti tallennusta pelannut ja ajankohta jolloin tallennus on kirjoitettu viimeksi muistiin.



KUVA 8. Kuvakaappaus SavedGameMetadata-luokan muuttujista (Microsoft Visual Studio Ultimate 2013 2015).

Kuvassa 9 on pelin tallennus- ja latausjärjestelmän testaamista ja demoamista varten tehty SaveData-luokka, joka sisältää tietorakenteen pelikohtaiselle tiedolle. Kuten aiemmin todettiin, pelikohtainen tieto voi olla mitä tahansa. Tässä tapauksessa halutaan tallentaa esimerkiksi pelaajan hahmon nimi merkkijonona ja pelaajan hahmon taso kokonaislukuna.



KUVA 9. Kuvakaappaus SaveData-luokasta (Microsoft Visual Studio Ultimate 2013 2015).

4.2.2 Tiedon tallentaminen ja lataaminen

Tieto tallennetaan muistiin siis kahdessa eri osassa: pelikohtainen tieto, joka on jokaisella pelillä erilaista, sekä metatietoa pelikohtaisesta tiedosta. Nämä tiedot tallennetaan Unity-pelimoottorin rajapinnasta löytyvään tallennettavalle tiedolle tarkoitettuun polkuun. Tämä polku sijaitsee jokaisella eri alustalla eri paikassa ja kyseisen alustan polku saadaan haettua rajapinnan Application-luokan persistentDataPath-ominaisuudesta.

Ennen kuin metadataa tai pelikohtaista tietoa voidaan tallentaa, niistä täytyy ensin luoda oliot, joiden muuttujat alustetaan halutuilla tiedoilla. Näitä olioita ei ole kuitenkaan mahdollista suoraan tallentaa tiedostoon. Olio täytyy ensin muuttaa sellaiseen muotoon, että sen voi tallentaa tiedostoon. Olion muuntamista tallennettavaksi tiedoksi kutsutaan serialisoinniksi. Kun olio pystytään muuttamaan tallennettavaan muotoon, myös tämä tallennettavassa muodossa oleva tieto pystytään muuttamaan takaisin tietoa vastaavaksi olioksi. Oliot voidaan serialisoida eri muotoihin. Yleisimpiä serialisaatiomuotoja ovat binääri, XML ja JSON. Binäärimuodossa tieto esitetään tavuina ja vain binäärimuotoa

käyttävä peli tai ohjelma pystyy ymmärtämään sitä. XML ja JSON muodoissa tieto esitetään sen sijaan suoraan tekstinä, jota myös ihminenkin voi ymmärtää. Toteutettu järjestelmä tukee ainoastaan binäärimuotoa. Tarkoituksena oli myös sisällyttää tuki JSON muodolle, mutta järjestelmää kehitettäessä ei löytynyt yhtäkään ilmaista ja toimivaa vapaan lähdekoodin JSON liitännäistä Unity-pelimoottorille.

Pelin tallennus- ja latausjärjestelmässä serialisoidaan siis jokainen kerta peliä tallennettaessa sekä SavedGameMetadata-olio ja kehittäjän itse tekemä olio, joka sisältää tietorakenteen pelikohtaiselle tiedolle. Kun oliot on serialisoitu, ne kirjoitetaan tiedostoihin. Pelin lataaminen käy päinvastaisesti. Ensin luetaan tallennustiedostot jonka jälkeen ne muutetaan tietoa vastaaviksi olioiksi. Tiedostoon kirjoittaminen ja siitä lukeminen tapahtuu C#-ohjelmointikielen File-luokasta löytyvillä metodeilla.

Ladatut pelit voidaan järjestelmässä lajitella tiettyyn järjestykseen. Lajittelu tapahtuu metatiedoissa olevien tietojen perusteella. Järjestelmä tukee seuraavia tapoja lajitella ladattuja pelejä: ladatut pelit pienimmästä tallennuspaikasta suurimpaan, ladatut pelit viimeisimmästä tallennusajankohdasta myöhäisimpään sekä ladatut pelit suurimmasta tallennukseen käytetystä ajasta pienimpään.

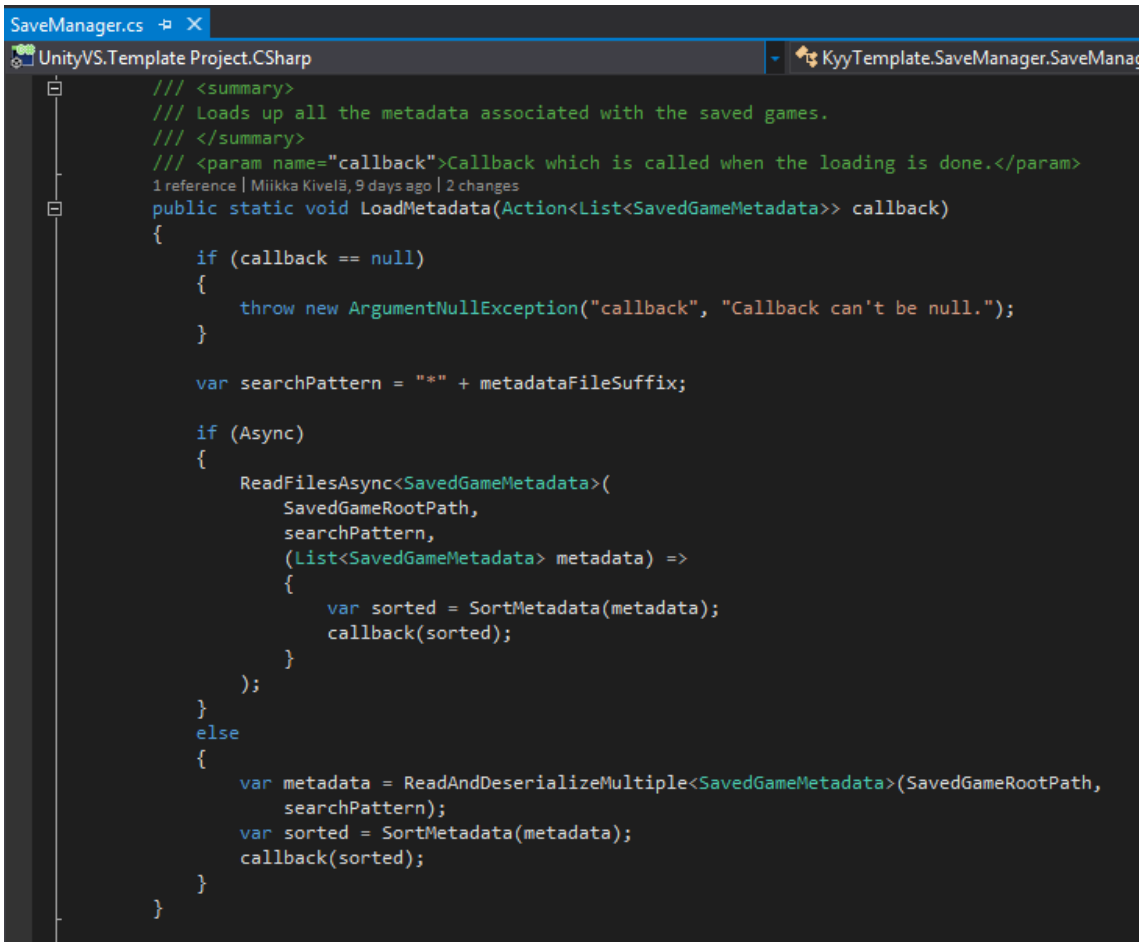
4.2.3 SaveManager-luokka

SaveManager-luokka tarjoaa laajennoksessa kaikki pelien tallentamiseen ja lataamiseen tarvittavat metodit. Jokaiselle tallennus- ja latausjärjestelmän toiminnallisuudelle eli pelien tallentamiselle, lataamiselle ja poistamiselle löytyy omat erilliset metodinsa. Luokan julkiset muuttujat vaikuttavat luokan toimintaan. Näiden muuttujien avulla voidaan hallita seuraavia asioita: toimiiko koko järjestelmä synkronisesti vai asynkronisesti, millä tavalla muistista ladatut tallennukset lajitellaan sekä millä muodolla tallennettava tieto serialisoidaan.

Kaikki SaveManager-luokan metodit toimivat niiden arvojen palautusten suhteen samalla tavalla, jotta luokan toiminnallisuus olisi kaikissa tilanteissa samankaltaista. Metodit ottavat parametrinä vastaan takaisinkutsumetodin, joiden tarkoituksena on että niitä kutsutaan silloin kun metodi on saanut tehtävänsä päätökseen. Esimerkiksi pelin tallennusten metadataa ladatessa sen toiminnallisuuden tarjoava metodi palauttaa listan

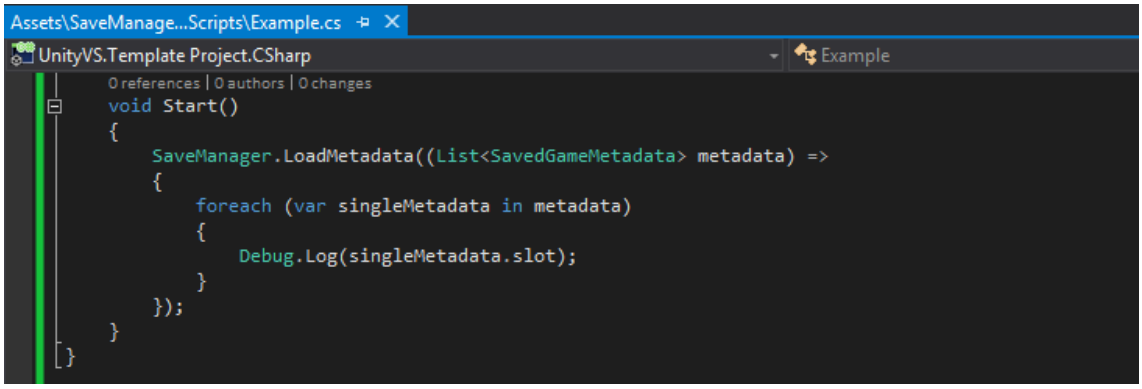
SavedGameMetadata-olioita takaisinkutsumetodin kautta. Peliä tallennettaessa ei kuitenkaan ole mitään, mitä sen toiminnallisuuden toteuttavan metodin olisi loogista palauttaa, mutta silti tallennusmetodikin ottaa vastaan takaisinkutsumetodin parametrinään. Vaikka metodi ei palauta takaisinkutsumetodin kautta mitään, pelin kannalta on kuitenkin hyvä tietää, milloin järjestelmä on saanut valmiiksi pelin tallentamisen prosessin. Esimerkiksi peli voi pelin tallennuksen aikana näyttää tallennusikonia pelaajan informoimiseksi prosessin käynnissäolosta.

Kuvassa 10 on SaveManager-luokan LoadMetadata-metodi. Metodi ottaa vastaan callback-nimisen parametrin, joka on siis takaisinkutsumetodi. Parametri on tyypiltään Action-luokka, joka kapsuloi sisäänsä metodin, joka taas ottaa vastaan listan SavedGameMetadata-olioita. LoadMetadata-metodissa takaisinkutsumetodia kutsutaan silloin, kun järjestelmä on saanut luettua kaikki metadata-tiedostot muistista ja muunnettua ne SavedGameMetadata-olioiksi.



KUVA 10. Kuvakaappaus SaveManager-luokan LoadMetadata-metodista (Microsoft Visual Studio Ultimate 2013 2015).

Kuvassa 11 on esimerkki SaveManager-luokan LoadMetadata-metodin käytöstä. Kyseistä metodia kutsuessa sille annetaan parametrinä anonyymi metodi. Kun LoadMetadata-metodi on saanut tehtävänsä valmiiksi, se kutsuu kyseistä anonyymiä metodia. Metodi käy tässä tapauksessa yksinkertaisesti läpi jokaisen ladatun SavedGameMetadata-olion ja tulostaa niiden slot-muuttujan konsoliin.



KUVA 11. Kuvakaappaus SaveManager-luokan LoadMetadata-metodin käytöstä (Microsoft Visual Studio Ultimate 2013 2015).

Se miksi takaisinkutsumetodeja käytetään metodien normaalien arvojen palautusten sijaan, johtuu järjestelmän mahdollisesta toiminnasta myös asynkronisesti. Asynkronisuus tarkoittaa, että erillisiä prosesseja suoritetaan samanaikaisesti eri säikeissä. Unity-pelimoottorissa pyörii oletuksena vain yksi säie ja tämä säie on säiesuojattu, joka tarkoittaa ettei säikeeseen pääse käsiksi muista säikeistä. Jos siis halutaan SaveManager-luokan toiminnallisuuden toimivan asynkronisesti ei voida määrittää luokan metodeja palauttamaan suoraan arvoja Unity-pelimoottorin säikeeseen, josta niitä todennäköisimmin kutsutaan. Myöskään takaisinkutsumetodit eivät suoraan toimi säikeiden välillä, mutta niitä voidaan kuitenkin käyttää C#-ohjelmointikielen ja Unity-pelimoottorin ominaisuuksia yhdistäen. SaveManager-luokkaan mahdollistettiin asynkroninen toiminta, koska pelien tallentaminen ja lataaminen saattaa viedä hyvinkin paljon aikaa ja jos kyseinen prosessi tehtäisiin Unity-pelimoottorin säikeessä, saattaisi esimerkiksi käyttöliittymä jäätyä prosessin ajaksi.

Kuvassa 12 näkyy osa AsyncActionHelper-luokasta. Luokka on tehty mahdollistamaan eri säikeiden kommunikointi Unity-pelimoottorin säikeen kanssa. Update-metodi on Unity-pelimoottorin tapahtumametodi, jota kutsutaan jokaisessa pelimoottorin ohjelmasilmukassa. Tämä metodi toimii aina Unity-pelimoottorin ainoassa säikeessä. Jostakin toisesta säikeestä voidaan vaikuttaa luokan muuttujiin, jota ovat boolean-

tyyppinen actionsAwaiting-muuttuja ja geneerinen lista gameThreadActions-muuttuja, jonka kuuluu pitää sisällään Action-luokan olioita. ActionsAwaiting-muuttujaa voidaan tarkastella ja muuttaa kaikista säikeistä ja sen tarkoituksena on hallita sitä, onko gameThreadActions-muuttujaan lisätty uusia takaisinkutsumetodeja. Update-metodissa katsotaan onko uusia takaisinkutsumetodeja lisätty ja jos niitä on, ne suoritetaan Unity-pelimoottorin säikeessä.

AsyncActionHelper.cs

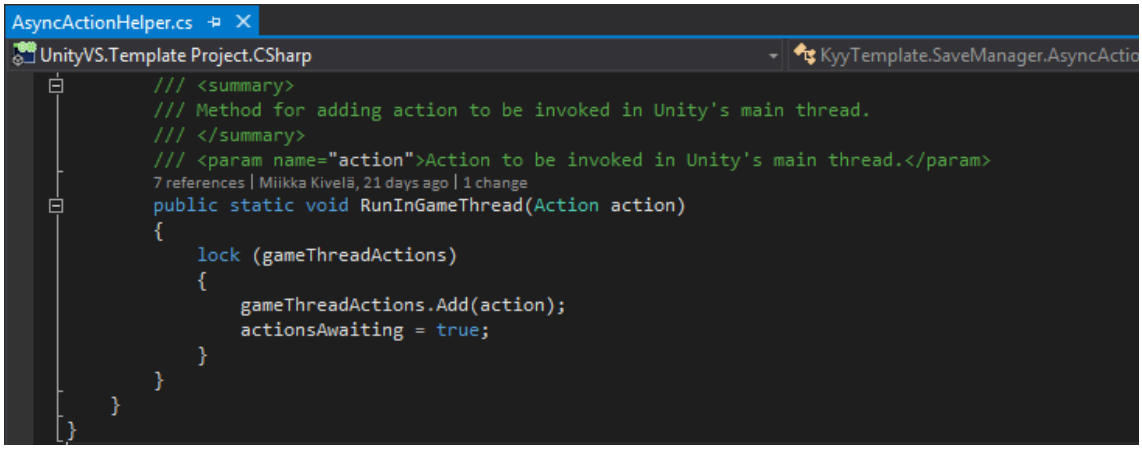
UnityVS.Template Project.CSharp — KyyTemplate.SaveManager.AsyncActi

```
/// <summary>
/// Helper class that can invoke actions in Unity's main thread from another thread. For
/// example, this can be used for callbacks when another thread has finished it's job.
///
/// This MonoBehavior must be included in the scene before using it.
/// </summary>
14 references | Miikka Kivelä, 21 days ago | 1 change
public class AsyncActionHelper : MonoBehaviour
{
    /// <summary>
    /// A list of action that are awaiting to get invoked in Unity's main thread.
    /// </summary>
    private static List<Action> gameThreadActions = new List<Action>();
    /// <summary>
    /// If there are any actions awaiting to be invoked in Unity's main thread. Multiple
    /// threads can access this at the same time.
    /// </summary>
    private static volatile bool actionsAwaiting;

    0 references | Miikka Kivelä, 21 days ago | 1 change
    void Update()
    {
        if (actionsAwaiting)
        {
            List<Action> tempActionList = new List<Action>();

            lock (gameThreadActions)
            {
                tempActionList.AddRange(gameThreadActions);
                gameThreadActions.Clear();
                actionsAwaiting = false;
            }

            foreach (var action in tempActionList)
            {
                action();
            }
        }
    }
```

KUVA 12. Kuvakaappaus osasta AsyncActionHelper-luokkaa (Microsoft Visual Studio Ultimate 2013 2015).

Kuvassa 13 on AsyncActionHelper-luokan RunInGameThread-metodi, jonka tarkoituksena on lisätä kapsuloituja metodeja suoritettavaksi Unity-pelimoottorin säikeeseen. GameThreadActions-muuttuja lukitaan, eli sitä ei voi muuttaa toisesta säikeestä ennen kuin lukko vapautetaan ohjelman toimesta. Lukitus tehdään, jottei kyseistä muuttujaa käsittele samaan aikaan useampi kuin yksi säie. Muuttujaan lisätään lukituksen aikana uusi Action-olio ja lopuksi muutetaan actionsAwaiting-muuttuja

kertomaan, että uusia kapsuloituja metodeja odottaa suoritusta Unityn-pelimoottorin säikeessä.



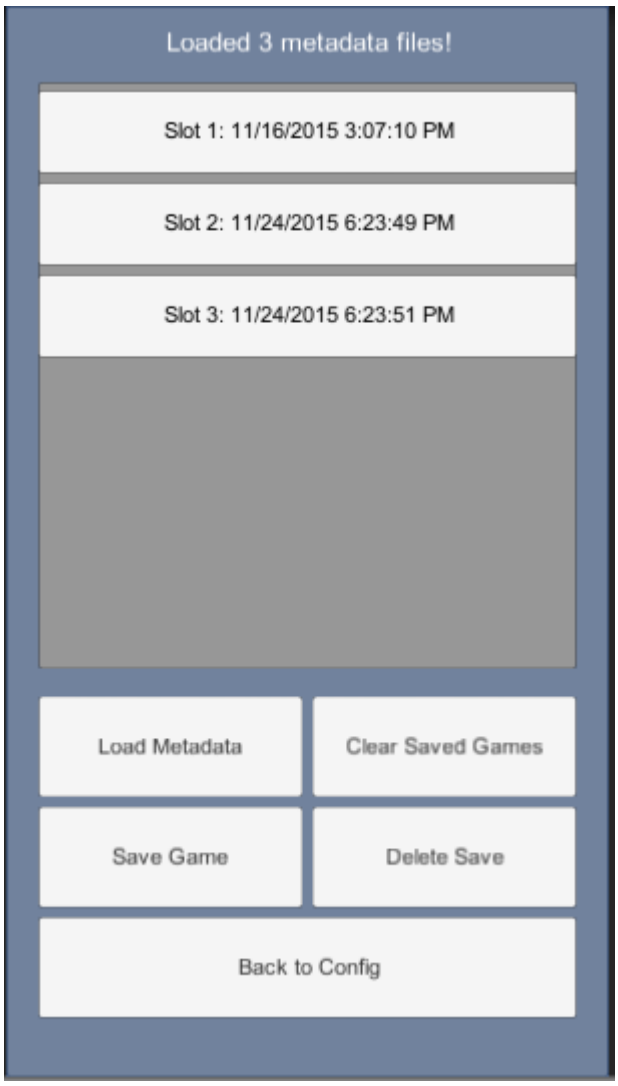
KUVA 13. Kuvakaappaus AsyncActionHelper-luokan RunInGameThread-metodista (Microsoft Visual Studio Ultimate 2013 2015).

Liitteessä 1 on kerrottu tarkemmin kuinka SaveManager-luokkaa käytetään.

4.2.4 Järjestelmän testaaminen

Järjestelmän testaamisen ja demonstroinnin tueksi luotiin myös testiscene, jossa voi sekä konfiguroida järjestelmän toimintaa sekä käyttää järjestelmää graafisessa käyttöliittymässä.

Kuvassa 14 on osa järjestelmän testisceneä, jossa voidaan ladata metadataa tallennuksista, tehdä tallennus, poistaa tallennus, poistaa kaikki tallennukset tai ladata tietty tallennus klikkaamalla sitä vastaavaa metadataa listassa.



KUVA 14. Kuvakaappaus SaveManager-järjestelmän testiscenestä (Unity 5.1.1f1 2015).

4.3 Järjestelmä dialogi-ikkunoiden luomiseen ja hallintaan

Dialogi-ikkunat ovat graafisen käyttöliittymän elementtejä, joiden tarkoituksena on viestiä käyttäjän kanssa ja jotka ottavat vastaan käyttäjältä jonkin syötteen. Dialogi-ikkunoita käytetään esimerkiksi ilmoitusten ja vahvistuskysymysten esittämiseen käyttäjälle. Dialogi-ikkunat voivat olla joko modaalisia tai ei-modaalisia. Ikkunan modaalisuus tarkoittaa sitä, ettei käyttäjä voi vuorovaikuttaa muun käyttöliittymän kanssa silloin kun dialogi-ikkuna on auki. Ikkunan modaalisuudella varmistetaan, että käyttäjä varmasti huomaa ja käsittelee dialogi-ikkunan ennen ohjelman käytön jatkamista. Ei-modaalisen ikkunan ollessa aktiivisena käyttäjä voi normaalisti vuorovaikuttaa muun ohjelman käyttöliittymän kanssa.

Dialogi-ikkunoita voidaan käyttää peleissä lukemattomiin eri tarkoituksiin, kuten esimerkiksi vahvistamaan pelistä poistuminen käyttäjältä tai kysymään pelaajan nimeä. Toisin kuin perinteisissä ohjelmissa, peleissä dialogi-ikkunat vaativat yleensä pelin ulkoasun kanssa yhtenäistä, erikseen tuotettua grafiikkaa ja eri dialogi-ikkunat voivat olla ulkoasultaan hyvinkin erilaisia.

Koska erilaisia dialogi-ikkunoita voi olla lähes lukemattomia ja koska kaikki dialogi-ikkunat eivät jaa samaa toiminnallisuutta, ei ole mahdollista tehdä yhtä assettikokonaisuutta joka kattaisi kaikkien dialogi-ikkunoiden toiminnallisuuden ja ulkoasun. Esimerkiksi jollakin dialogi-ikkunnalla on otsikko ja se on modaalinen kun taas toisella dialogi-ikkunalla ei ole otsikkoa ja se ei ole modaalinen. Tästä syystä sovelluskehykseen toteutettu dialogi-ikkunoiden luomis- ja hallintajärjestelmä tarjoaa lähinnä vain yksinkertaisen tavan hallita dialogi-ikkunoita sekä puitteet dialogi-ikkunoiden luomiseen.

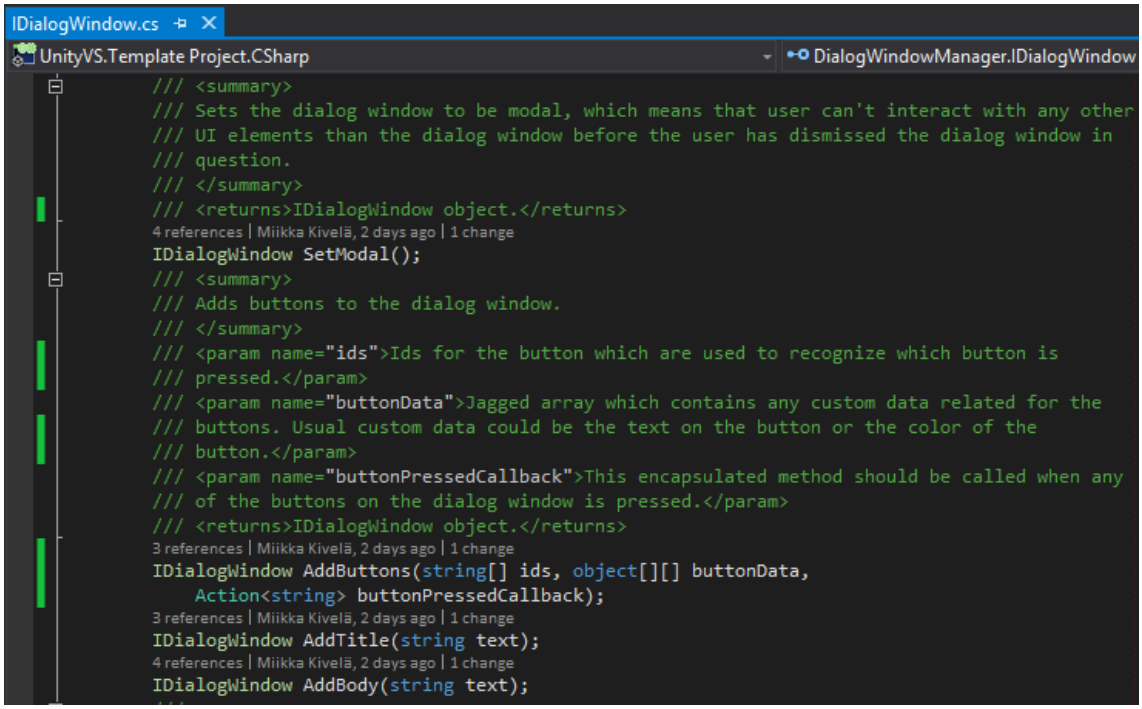
Kyseinen järjestelmä päätettiin toteuttaa sovelluskehykseen, koska dialogi-ikkunat ovat peleissä hyvin yleisiä eikä sovelluskehyksestä vielä löytynyt aiheeseen liittyviä laajennoksia. Järjestelmään tehtiin kaksi yksinkertaista dialogi-ikkunaa järjestelmän testausta varten sekä pelien prototyyppien tekemisen nopeuttamista varten.

4.3.1 Dialogi-ikkunan luominen ja näyttäminen

Omien dialogi-ikkunoiden luomisen pohjalla järjestelmässä on rajapinta IDialogWindow. IDialogWindow-rajapinta tarjoaa metodit yleisimpiin dialogi-ikkunoista löytyviin toiminnallisuuksiin. Dialogi-ikkunan tekijä päättää kuitenkin itse sen, miten toiminnallisuus ikkunaan toteutetaan. Käytännössä prosessi tapahtuu niin, että toteutetaan oma dialogi-ikkunan luokka IDialogWindow-rajapinnasta kirjoittamalla toiminta niille metodeille, jolle kyseisestä dialogi-ikkunasta kuuluisi löytyä toiminnallisuus. Vaikka rajapinta tarjoaakin monia eri toiminnallisuuksia dialogi-ikkunoille, ei kaikkia toiminnallisuuksia tarvitse eikä kuulukkaan toteuttaa.

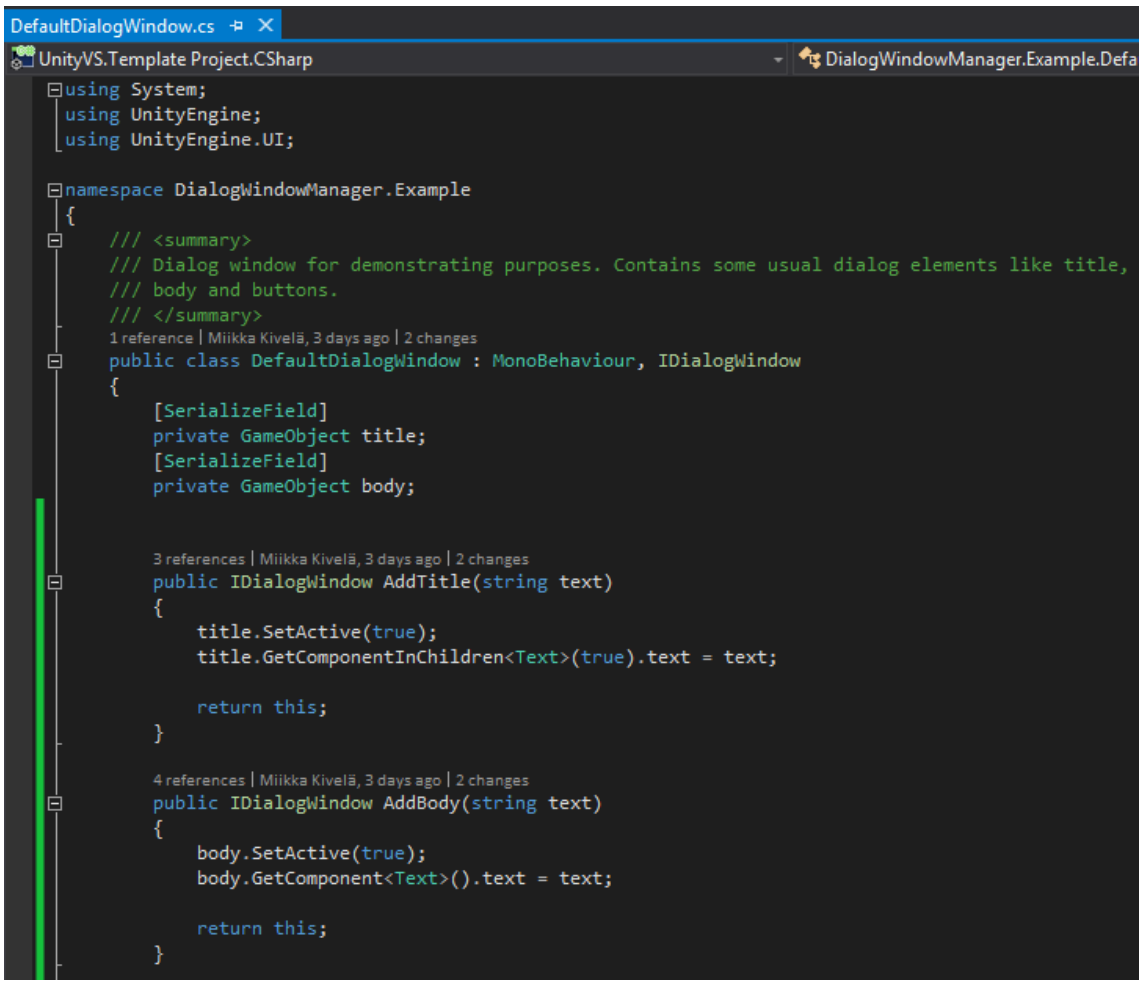
Kuvassa 15 on muutamia IDialogWindow-rajapintaan määriteltyjä metodeja. SetModal-metodi asettaa dialogi-ikkunan modaaliseksi, AddButtons-metodi lisää dialogi-ikkunaan

painettavia nappeja, AddTitle-metodi lisää dialogi-ikkunalle otsikon ja AddBody-metodi lisää dialogi-ikkunaan ns. leipätekstin. Jokaisen metodin palautusarvoksi on myös määritelty IDialogWindow-olio. Metodien on tarkoitus palauttaa aina olio, johon sillä hetkellä viitataan. Olio palautetaan, jotta voidaan yksinkertaistaa dialogi-ikkunoiden rakentamista ketjuttamalla olion metodikutsuja.



KUVA 15. Kuvakaappaus IDialogWindow-rajapinnan metodeista (Microsoft Visual Studio Ultimate 2013 2015).

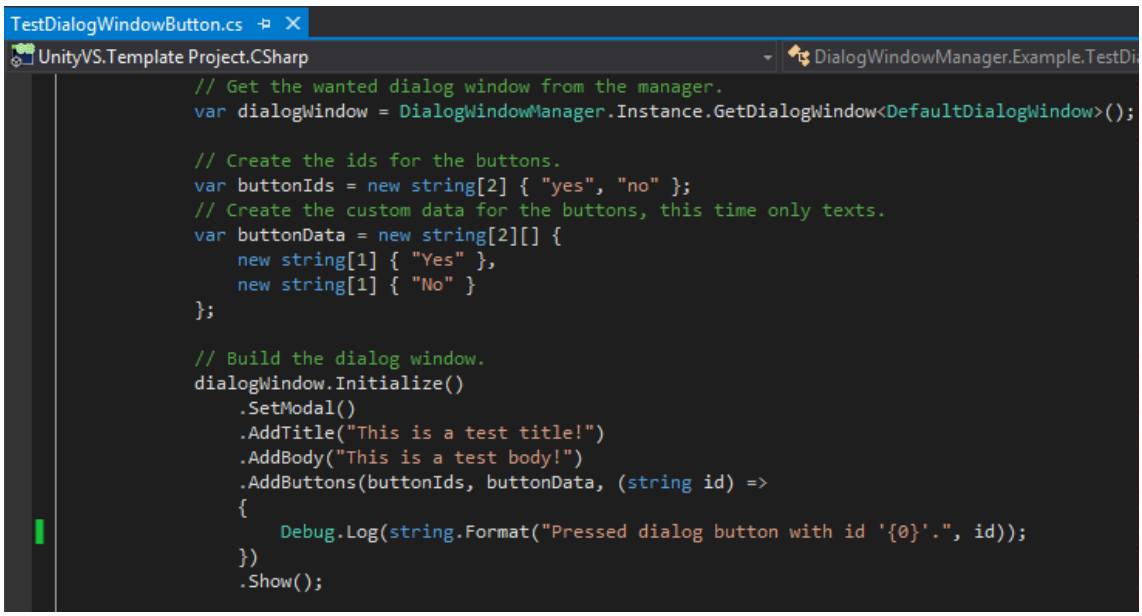
Kuvassa 16 on osa DefaultDialogWindow-luokan toteutusta. DefaultDialogWindow-luokka on järjestelmää varten tehty esimerkki yksinkertaisesta dialogi-ikkunasta. Se toteuttaa IDialogWindow-rajapinnan ja kuvassa näkyy toteutettu toiminnallisuus dialogi-ikkunan otsikon ja leipätekstin lisäämiselle. Luokka on myös luonnollisesti periytetty MonoBehaviour-luokasta, koska skriptin on tarkoitus tuoda toiminnallisuus sitä vastaavalle peliobjektille.



KUVA 16. Kuvakaappaus osasta DefaultDialogWindow-luokan toteutusta (Microsoft Visual Studio Ultimate 2013 2015).

Järjestelmässä on tarkoitus rakentaa dialogi-ikkuna pala kerrallaan. Palalla tarkoitetaan jotakin yhtä toiminnallisuutta, esimerkiksi otsikkoa. Jos dialogi-ikkuna ei toteuta jotakin toiminnallisuutta, sitä ei myöskään rakenneta mukaan dialogi-ikkunaan. Tällä tavalla yhtä dialogi-ikkunaa voidaan käyttää useaan eri tarkoitukseen.

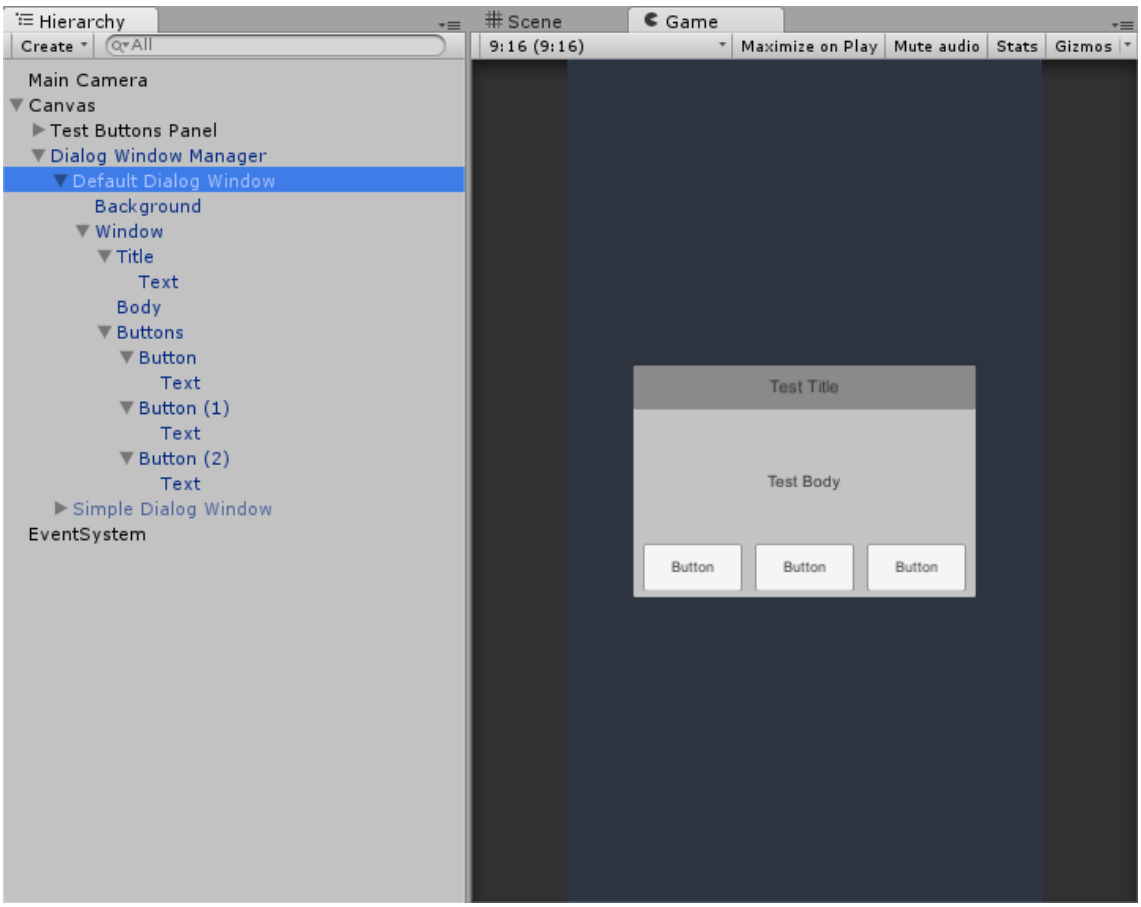
Kuvassa 17 on esimerkki DefaultDialogWindow-ikkunan rakentamisesta. Dialogi-ikkuna ensin alustetaan, jonka jälkeen siihen lisätään modaalisuus, otsikko, tekstiä ja kaksi painiketta. AddButtons-metodin yhteydessä annetaan parametrinä anonyymi takaisinkutsumetodi, jota kutsutaan kun jotakin dialogi-ikkunassa olevaa painiketta painetaan. Lopuksi dialogi-ikkuna pistetään näkyviin.



KUVA 17. Kuvakaappaus dialogi-ikkunan rakentamisesta (Microsoft Visual Studio Ultimate 2013 2015).

Sen lisäksi, että dialogi-ikkunalle tehdään IDialogWindow-rajapinnan toteuttava luokka, joudutaan dialogi-ikkunalle rakentamaan oma käyttöliittymäpeliobjekti sceneen. Järjestelmää varten esimerkeiksi tehdyt dialogi-ikkunat tehtiin Unityn omilla käyttöliittymäelementeillä.

Kuvassa 18 on DefaultDialogWindow-luokkaa varten tehty Default Dialog Window-peliobjekti. Title-peliobjekti sisältää dialogi-ikkunan otsikon, Body-peliobjekti sisältää dialogi-ikkunan leipätekstin ja Buttons-peliobjekti sisältää dialogi-ikkunan painikkeet. Näitä peliobjekteja päälle ja pois päältä pistämällä voidaan vaikuttaa siihen, mitä toiminnallisuutta dialogi-ikkunassa on. Esimerkiksi Title-peliobjektin pois päältä laittamalla dialogi-ikkunassa ei ole enää otsikkoa. Peliobjektien päälle ja pois laittamista hallinnoidaan DefaultDialogWindow-skriptistä.



KUVA 18. Kuvakaappaus Default Dialog Window-peliobjektista (Unity 5.1.1f1 2015).

4.3.2 Dialogi-ikkunoiden hallinta ja niiden skripteihin viittaaminen

Ennen kuin tiettyä dialogi-ikkunaa voidaan rakentaa ja näyttää pelin sisällä, dialogi-ikkunan sisältävään skriptiin täytyy ensin hankkia viittaus. Unity-editorissa peliobjektiin toiminnallisuutta tuovat skriptit liitetään peliobjekteihin komponentteina ja Unity-pelimoottori huolehtii skriptissä olevien luokkien ilmentymien luomisesta. Olioita ei siis itse luoda lainkaan peliobjektien skripteille, vaan viittaaminen toisesta skripti-komponentista toiseen tapahtuu joko asettamalla suora viittaus skriptistä toiseen tai hakemalla viittaus scenestä Unityn rajapinnan metodeja käyttäen.

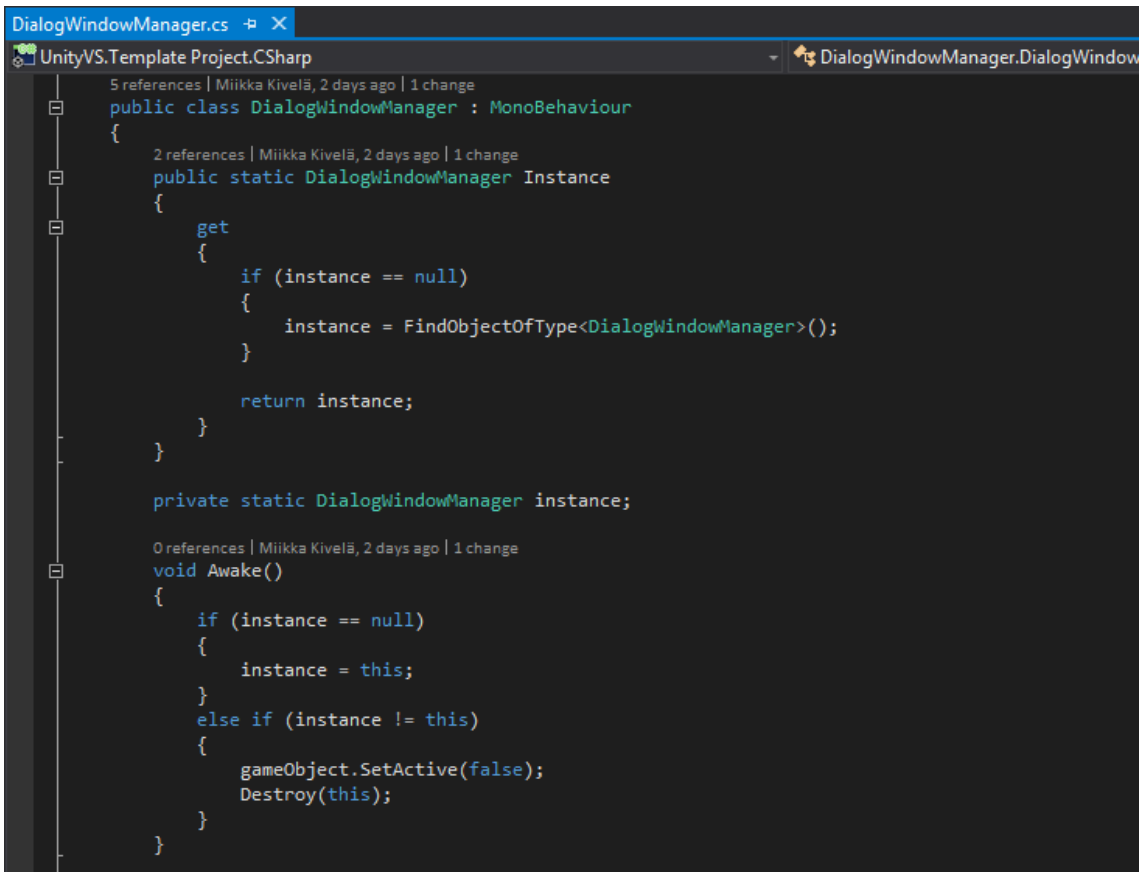
Unity-editorin sisällä suoria viittauksia komponentteihin, joita voivat siis olla myös skriptit, voidaan helposti asettaa Inspector-näkymän sisällä. Kaikki skriptissä olevat julkiseksi määritellyt muuttujat näkyvät Inspector-näkymässä ja näihin paikkoihin voidaan vetää viittauksia muihin komponentteihin drag and drop-menetelmällä. Koska dialogi-ikkunat ovat yleisesti käytettyjä käyttöliittymäelementtejä, niitä voidaan käyttää

hyvin useasta eri skriptistä. Mitä useammasta skriptistä dialogi-ikkunoita käytetään, sitä useampi suora viittaus dialogi-ikkunan skriptiin vaaditaan. Aina dialogi-ikkunaa käytettäessä jouduttaisiin tekemään uusi viittaus scenessä, joka käy pidemmän päälle toistuvaksi ja hyvin epäkäytännölliseksi. Tästä syystä suora viittaaminen dialogi-ikkunoiden skripteihin ei ole kannattavaa.

Suoran viittaamisen lisäksi komponentteja voidaan hakea scenestä Unity-pelimoottorin rajapinnasta löytyvillä metodeilla, kuten esimerkiksi FindObjectOfType-metodilla. Mutta kuten kyseisen metodin dokumentaatiossa todetaan, metodi on hyvin hidas (Unity Scripting API: Object.FindObjectOfType 2015). Myös muut samantyyppiset hakuun perustuvat metodit ovat hitaita, eivätkä ne siitä syystä sovi dialogi-ikkunan skriptin viittauksen hankkimiseen.

Ratkaisu dialogi-ikkunoiden skripteihin viittaamiseen on luoda niiden hallitsemiseen tarkoitettu skripti ja käyttää tähän skriptiin viittamiseen ainokainen-suunnittelumallia. Ainokainen-suunnittelumallilla tarkoitetaan, että luokasta on olemassa vain yksi ilmentymä johon tarjotaan globaali pääsy (Gamma, Helm, Johnson & Vlissides 1995, 127).

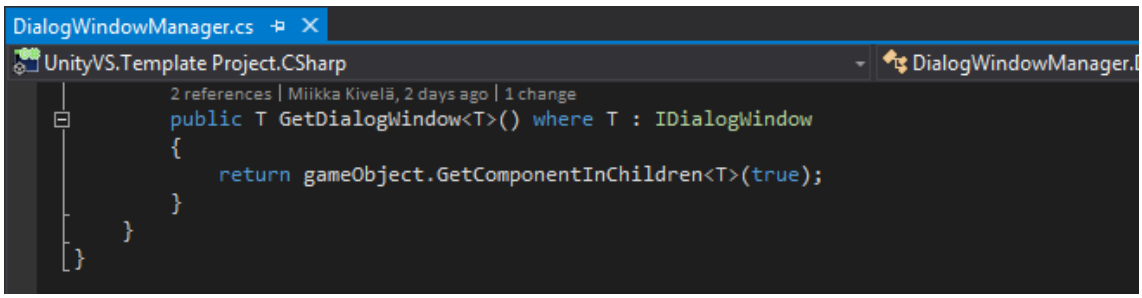
Kuvassa 19 on osa dialogi-ikkunoiden hallitsemiseen tarkoitettua DialogWindowManager-luokkaa, jonka toteuttamisessa on käytetty hyväksi ainokainen-suunnittelumallia. Instance-ominaisuuden kautta haetaan luokan ainokais-ilmentymä, joka edustaa instance-muuttuja. Koska Instance-ominaisuus on määritelty julkiseksi ja staattiseksi, siihen voidaan viitata mistä tahansa skriptistä suoraan pelkkää luokan nimeä käyttämällä. Jos instance-muuttuja on alustettu, Instance-ominaisuus palauttaa sen, mutta jos sitä ei ole vielä alustettu, instance-muuttuja alustetaan etsimällä se scenestä. Kyseinen muuttuja alustetaan myös Awake-metodissa, joka on Unity-editorin tapahtumametodi, jota Unity-pelimoottori kutsuu aina kun scriptin ilmentymä ladataan. Awake-metodissa varmistetaan myös ettei skriptistä ole olemassa scenessä muita kuin yksi ilmentymä. Jos muita ilmentymiä on lisätty sceneen, niiden peliobjekti poistetaan.



KUVA 19. Kuvakaappaus osasta DialogWindowManager-luokan toteutusta (Microsoft Visual Studio Ultimate 2013 2015).

DialogWindowManager-skripti täytyy myös lisätä sceneen ja siksi se löytyy valmiina liitettynä Dialog Window Manager-prefabiin, jonka kehittäjän kuuluu lisätä itse sceneen Canvas-peliobjektin alle. Canvas-peliobjekti on Unityn käyttöliittymäelementti, jonka alle muut käyttöliittymäelementit on tarkoitus pistää.

Kuvassa 20 on DialogWindowManager-luokan GetDialogWindow-metodi. Metodi on geneerinen, joka ottaa vastaan vain IDialogWindow-rajapinnan toteuttavia luokkia. Metodi hakee Dialog Window Manager-peliobjektin lapsista haluttua IDialogWindow-rajapinnan toteuttavaa skriptiä. Periaattena on siis, että kun Dialog Window Manager-prefabi on lisätty sceneen, sen lapsiksi lisätään kaikki halutut dialogi-ikkuna peliobjektit tai prefabit. Esimerkiksi kuvassa 18 Dialog Window Manager-peliobjektin alle on lisätty kaksi dialogi-ikkuna prefabia, joita ovat Default Dialog Window ja Simple Dialog Window.



KUVA 20. Kuvakaappaus DialogWindowManager-luokan GetDialogWindow-metodista (Microsoft Visual Studio Ultimate 2013 2015).

Liitteessä 2 on kerrottu tarkemmin dialogi-ikkunoiden luomisesta ja niiden hallinnasta.

5 POHDINTA

Tässä opinnäytetyössä kehitettiin toimeksiantajan sovelluskehystä kahdella erillisellä tavalla. Ensinnäkin olemassa olevaa sovelluskehystä pyrittiin kehittämään siten, että se olisi tulevaisuudessa laajennettava ja selkeä kokonaisuus, jota voidaan käyttää jokaisen tulevan peliprojektin pohjana. Tämän tueksi kirjoitettiin kehitysehdotuksia sovelluskehyksen parantamiseen, jotka käydään toimeksiantajan kanssa yhdessä läpi ja joiden toteutuksesta toimeksiantaja itse päättää. Toisekseen sovelluskehystä oli tarkoitus laajentaa tekemällä siihen uusia laajennoksia. Sovelluskehykseen tehtiin lopulta kaksi uutta laajennosta, joita olivat pelin tallennus- ja latausjärjestelmä sekä järjestelmä dialogi-ikkunoiden luomiseen ja hallitsemiseen.

Opinnäytetyön aihetta rajatessa olisi ollut hyvä miettiä tarkemmin keskittymistä vain toiseen näistä kehittämisaiheista. Työtä tehdessä tuli selväksi, että kummastakin kehittämisaiheesta olisi hyvin voinut saada tehtyä oman yksittäisen opinnäytetyönsä. Tästä huolimatta tässä opinnäytetyössä saatiin kuitenkin kattavasti käsiteltyä sovelluskehyksen kehittämistarpeet sekä tehtyä sovelluskehykseen uusia laajennoksia. Toimeksiantajan mukaan opinnäytetyölle asetetut tavoitteet täyttyivät hyvin ja opinnäytetyön tuloksia tullaan käyttämään jatkossa toimeksiantajan pelinkehitysprosessissa ja sen kehittämisessä.

Sovelluskehyksen kehittämiseen liittyen yhteisten käytäntöjen käyttöönotto ja ohjeistuksen lisääminen sovelluskehykseen olisi hyvä saada toteutettua mahdollisimman nopeasti. Kuten yleisesti kaikkien ohjelmistosovellusten kanssa, sovelluskehyksen kasvaessa ongelmat sen laajennettavuuden ja selkeyden kanssa kasvavat hyvin nopeasti ilman selkeää ja tarkkaan mietittyä hallinnointitapaa. Vapaasti kasvavasta sovelluskehyksestä saattaa jossakin vaiheessa tulla jopa taakka sen sijaan, että se nopeuttaisi pelinkehitysprosessia.

Tulevaisuudessa sovelluskehyksen laajentamisessa eli uusien laajennosten tekemisen yhteydessä on myös hyvä miettiä jo valmiina olevien ratkaisuiden käyttämistä. Unitylle on olemassa paljon valmiita laajennoksia, liitännäisiä ja työkaluja pelinkehittämisen nopeuttamiseen ja helpottamiseen. Unity Asset Storesta, joka on Unityn oma kauppapaikka Unity-assettien myymiseen ja jakamiseen, löytyy lukuisia valmiita assetteja kokonaisista viitekehyksistä yksittäisiin 3D-malleihin. Asset Storessa on sekä

maksullisia että maksuttomia assetteja. Maksullisten assettien laatua ja toimintaa ei voi kuitenkaan millään varmistaa ennen assetin ostamista. Jokaisella assetilla on arviointijärjestelmä, joista voi saada kuvan assetin laadusta, mutta koska suosituimmillakin asseteilla on parhaimmillaan vain muutamia kymmeniä arvioita, ei niiden perusteella voida tehdä johtopäätöksiä assettien laadusta. Asset Storen lisäksi on hyvä muistaa etsiä myös avoimen lähdekoodin assetteja.

LÄHTEET

Build Unity Games with Visual Studio. 2015. Microsoft. Luettu 10.11.2015. https://www.visualstudio.com/features/unitytools-vs

C# Programming Guide: Namespaces. 2015. Microsoft. Luettu 10.11.2015 https://msdn.microsoft.com/en-us/library/0d941h9d.aspx

Gamma E. & Helm R. & Johnson R. & Vlissides J. 1995. Design Patterns. Elements of Reusable Object-Oriented Software. 42. painos. Addison-Wesley.

Get Unity. 2015. Unity Technologies. Luettu 10.11.2015. https://unity3d.com/get-unity

Koskimies K. & Mikkonen T. 2005. Ohjelmistoarkkitehtuurit. Helsinki: Talentum Media Oy.

Made With Unity. 2015. Unity Technologies. Luettu 10.11.2015. https://unity3d.com/showcase/gallery/games

Microsoft Visual Studio Ultimate 2013. 2015. Microsoft Corporation.

Unity. 2015. Unity Technologies. Luettu 10.11.2015. https://unity3d.com/unity

Unity 5.1.1f1. 2015. Unity Technologies.

Unity Manual: Creating and Using Scripts. 2015. Unity Technologies. Luettu 10.11.2015. http://docs.unity3d.com/Manual/CreatingAndUsingScripts.html

Unity Manual: Prefabs. 2015. Unity Technologies. Luettu 10.11.2015. http://docs.unity3d.com/Manual/Prefabs.html

Unity Manual: Special Folders and Script Compilation Order. 2015. Unity Technologies. Luettu 10.11.2015. http://docs.unity3d.com/Manual/ScriptCompileOrderFolders.html

Unity Manual: Special Folder Names. 2015. Unity Technologies. Luettu 10.11.2015. http://docs.unity3d.com/Manual/SpecialFolders.html

Unity Scripting API: MonoBehaviour. 2015. Unity Technologies. Luettu 10.11.2015. http://docs.unity3d.com/ScriptReference/MonoBehaviour.html

Unity Scripting API: Object.FindObjectOfType. 2015. Unity Technologies. Luettu 10.11.2015. http://docs.unity3d.com/ScriptReference/Object.FindObjectOfType.html

Wilcox M. & Voskoglou C. 2014. Developer Economics: State of the Developer Nation Q3 2014. VisionMobile. Luettu 10.11.2015. http://www.visionmobile.com/product/developer-economics-q3-2014/

LIITTEET

Liite 1. Readme-dokumentaatio pelin tallennus- ja latausjärjestelmälle

1 (8)

Save Manager for Unity

Allows saving, loading and deleting persistent game data both asynchronously and synchronously.

Currently supported formats are:

* Binary

Overview

Saved game is technically constructed from two different pieces: game specific data and metadata. Game specific data is unstructured data which is provided by the developer (like the name of the character that player creates and level of that character). This data is best provided as a class that holds all the data that needs to be saved. Metadata on the other hand is structured data that is partly provided from the developer as well. Metadata contains following information:

* File name for the file that holds the saved game and which are written on the persistent storage. This is supplied by developer and should not include suffix. Players of the game should not any have reason to see the name of the files.
* Slot which is an optional number that can separate save files from one another in a game that uses save slots.
* Optional total time that the player has played the save.
* Time when the save was written on a file last time.

Game specific data and metadata are separated because instead of reading and deserializing all the game specific data (this could take pretty long if you have many

saves), only small metadata files are read and after that one can choose to load game specific data files based on the metadata.

There is also possibility to save, load and delete game data without metadata. This is useful in games that only need one file for all persistent data.

Configure

Add the source into your project and simply use SaveManager's static methods to access its functionality.

How the test scene works

When running the scene, one can configure the SaveManager class from the UI. Pressing the Run-button will present all the saved games in a scroll view and all the functionality related to them. Buttons do what they are labeled to do. Click any save game element on the scroll view to select it. Selecting one element loads up the game specific data associated with the metadata. UI in the test scene is built for portrait resolutions only.

Save manager’s settings

Save manager have a few settings that can be customized according to one's needs:

* Async: Should the SaveManager class' functions work asynchronously or synchronously.
* Saved Game Sort Mode: How the metadata loaded should be sorted. Metadata can be sorted by time when the save was last written, by slot and by total game time played that save.
* Saved Game Serialization: What format of serialization save manager should use. Look up the supported formats above.

Defining data structure for game specific data

Before saving games, you must define data that you want to save. The data can virtually be anything. Save manager supports all the types that are supported by the serializers. Here we have a class that holds few game specific variables. Note that the data object must be marked with `[Serializable]` attribute.

```
#!c#

[Serializable]
public class SaveData
{
    public string characterName = "Test";
    public bool hasSomeUpgrades = true;
}
```

Saving a new game

```
#!c#

// Create a new object specific data.
SaveData saveData = new SaveData();
// Create a new metadata for the saved game. Pass a file name without suffix to the constructor.
SavedGameMetadata metadata = new SavedGameMetadata("SavedGame");

SaveManager.SaveGame(metadata, saveData, () =>
{
    Debug.Log("Game saved successfully!");
});
```

Loading metadata for all saved games

```
#!c#

SaveManager.LoadMetadata((List<SavedGameMetadata> metadata) =>
{
    // Iterate through all the metadata loaded.
    foreach (var singleMetadata in metadata)
    {
        Debug.Log("Saved game file name: " + singleMetadata.savedGameFileName);
        Debug.Log("Save slot: " + singleMetadata.slot);
    }
});
```

Loading single game specific data for a saved game

After you have loaded metadata for saved games, you can use that metadata to load game specific data associated with the metadata. After loading game specific data hold up to the metadata passed in SavedGame object to save that existing game in the future.

```
#!c#

SaveManager.LoadSavedGame(metadata, (SavedGame savedGame) =>
{
    // Cast the loaded game data into data structure that holds your game specific data.
    var saveData = (SaveData)savedGame.gameSpecificData;

    Debug.Log("Character name: " + saveData.characterName);
});
```

Saving existing game

Also when saving existing game, you have to have metadata loaded for saved games first. Load up the metadata and choose a single metadata you want to save and then pass it along with game specific data to save the existing game.

```
#!c#

SaveManager.SaveGame(metadata, saveData, () =>
{
   Debug.Log("Game saved successfully!");
});
```

Deleting a game

Once again, you have to have metadata loaded before you can delete any games. Load up the metadata and use that metadata to delete a game you want.

```
#!c#

SaveManager.DeleteSave(metadata, () =>
{
   Debug.Log("Save deleted");
});
```

Deleting all the saved games

This method clears ALL the saved games.

```
#!c#

SaveManager.DeleteAllSavedGames(() =>
{
   Debug.Log("All saves deleted!");
});
```

Saving game data

Saves data to the path without messing with the metadata.

```
#!c#

// Create a new object specific data.
var saveData = new SaveData();
// Create a path for the file that holds the data. Notice that now you need to supply the
// full path in Application.persistentDataPath.
var path = Path.DirectorySeparatorChar + "Example.data";

SaveManager.SaveData(saveData, path, () =>
{
   Debug.Log("Data saved successfully!");
});
```

Loading game data

Loads up the game data straight from a file without the need to mess with the metadata.

```
```

```
#!c#

SaveManager.LoadData<SaveData>(path, (SaveData data) =>
{
   Debug.Log("Character name: " + saveData.characterName);
});
```

Loading multiple files of game data

There is also a possibility to load multiple files of game data simultaneously using a search pattern.

```
#!c#

SaveManager.LoadDataMultiple<SaveData>(path, "*.data", (List<SaveData> data) =>
{
   foreach (var singleData in data)
   {
      Debug.Log("Character name: " + singleData.characterName);
   }
});
```

Deleting game data

Delete any file straight with a file name.

```
#!c#

var paths = new string[1] { Path.DirectorySeparatorChar + "Example.data" };
```

```
SaveManager.DeleteData(paths, () =>
{
    Debug.Log("Paths deleted!");
});
```

Liite 2. Readme-dokumentaatio dialogi-ikkunoiden luomis- ja hallintajärjestelmälle

Dialog Window Manager for Unity

Since there can be infinite amount of different dialog windows and because all dialog windows don't share the same functionality, this extension mainly provides help to manage the dialog windows and also provides few simple dialog windows that can be used for quick prototyping.

Configure

Add the source into your project and drag the Dialog Manager prefab from DialogWindowManager/Prefabs/ into your main UI canvas. Use dialog manager's singleton instance to access its methods.

About the Test scene

Test scene demonstrates few simple dialog windows. These dialog windows can be used for quick prototyping, but keep in mind that their UI layout is built only for portrait resolutions.

Creating custom dialog windows

To create custom dialog windows, create a class that extends MonoBehaviour and implements IDialogWindow. IDialogWindow interface consists of functionality that is common to some or all dialog windows. Your custom dialog window doesn't need to use all that functionality, because dialog windows are built piece by piece and if you don't want some functionality on your dialog window, just left it unimplemented.

Below we have a TestDialogWindow class. Two of the interface's methods are implemented for demoing purposes. This dialog window does not have functionality for showing title, so the method AddTitle(string text) is not implemented. Good practice is to throw NotImplementedException() if the functionality is not implemented by the

dialog window. But like most of the dialog windows, this dialog window has body text and thus AddBody(string text) method is implemented. All interface's methods, which adds some functionality on the dialog window, are declared to return instance of that window. This is just to make the building of the dialog windows easier.

When you are done with the code side of things, create graphical UI for it in the scene and make the created game object child of Dialog Manager game object. Highest parent of that custom dialog window game object should contain the IDialogWindow script you created.

```
#!c#

public class TestDialogWindow : MonoBehaviour, IDialogWindow
{
  public Text body;

  public IDialogWindow AddTitle(string text)
  {
    throw new NotImplementedException();
  }

  public IDialogWindow AddBody(string text)
  {
    body.text = text;

    return this;
  }

  // All the other interface methods below...
}
```

IDialogWindow interface only have the basic functionality that are usually found on dialog windows. Extend the interface freely to add new functionality.

Getting instance of the custom dialog window

To make custom made dialog window visible, first get the instance of that window by calling DialogWindowManager's GetDialogWindow<T>() method.

```
#!c#

var dialogWindow = DialogWindowManager.Instance.GetDialogWindow<TestDialogWindow>();
```

Building and showing the custom dialog window

When you have an instance of you custom dialog window, use its methods to build and show it.

```
#!c#

// Build the dialog window.
dialogWindow.Initialize()
  .SetModal()
  .AddBody("This is a test body! Click anywhere to dismiss.")
  .DismissByClick(() =>
  {
    Debug.Log("Dialog window dismissed!");
  })
  .Show();
```

4 (4)

Because dialog windows are built piece by piece, one can create multiple different dialog windows from one dialog window game object.