Antti Riiali

2D-PELIN TOTEUTUS UNITYLLÄ

Opinnäytetyö

Liiketalouden ammattikorkeakoulututkinto

Tietojenkäsittelyn koulutus

2022

South-Eastern Finland University of Applied Sciences

Tutkintonimike	Tradenomi (AMK)
Tekijä	Antti Riiali
Työn nimi	2D-pelin toteutus Unityllä
Toimeksiantaja	XAMK Game Studios -hanke
Vuosi	2022
Sivut	50 sivua
Työn ohjaaja	Jukka Selin

TIIVISTELMÄ

Opinnäytetyö tutkii, kuinka Unity-pelimoottori soveltuu 2D-pelimaailmojen, ja erityisesti 2D-tasohyppelypelien, toteutukseen. Opinnäytetyössä suunnitellaan ja toteutetaan yksinkertainen 2D-tasohyppelypeli.

Työn toimeksiantajana on Kaakkois-Suomen ammattikorkeakoulun Game Studios -hanke. Hankkeen tarkoituksena on edistää peliohjelmoinnin opetusta ammattikorkeakoulussa.

Unity-pelimoottori on alun perin rakennettu 3D-kehitykseen, johon on myöhemmin lisätty tuki 2D-kehitykselle. Ero näiden kahden eri ulottuvuuden välillä on kuitenkin vähäinen, ja käyttöliittymät toimivat lähes samalla tavalla.

Opinnäytetyön pelidemo mukailee suunnitelmaltaan massiivissa monen pelaajan verkkoroolipeleissä (MMORPG, Massive Multiplayer Online Role-playing Game) nähtyä toimeksiantoa, jossa pelaajan täytyy suorittaa yksinkertainen tehtävä. Pelaajan tehtävä demossa on kerätä 10 omenaa yksinkertaisessa kentässä hyppimällä tasolta tasolle ennen kuin aika loppuu kesken.

Opinnäytetyössä käydään lävitse yksityiskohtaisesti, kuinka luodaan uusi projekti Unityssä, ja alustetaan se, kuinka luodaan objekteja ja kuinka niihin liitetään komponentteja, kuinka animoidaan pelihahmoa ja annetaan sille useita eri animaatioita ja kuinka koodataan eri toiminnallisuuksia objekteille ja pelille.

Lopputuloksena päädyttiin siihen, että Unity-pelimoottori soveltuu hyvin 2D-pelikehitykseen ja varsinkin 2D-tasohyppelyn luontiin. Opinnäytetyön tekijälle avautui ajatuksia siitä, kuinka pelidemoa voisi jatkokehittää, ja minkälaisia asioita kannattaisi tehdä toisin tai paremmin tulevaisuudessa tämän kaltaisessa projektissa.

Asiasanat: peliohjelmointi, unity, 2d, tasohyppely

XAMK
South-Eastern Finland
University of Applied Sciences

Degree	Bachelor of Business Administration
Author	Antti Riiali
Thesis title	Implementation of 2D game on Unity
Commissioned by	XAMK Game Studios project
Time	2022
Pages	52 pages
Supervisor	Jukka Selin

ABSTRACT

This thesis researched how well the Unity game engine was suited for building 2D game worlds, and in particular, 2D platform games. During the thesis a simple 2D platform game was designed and implemented. The commissioner of the work was Southern Finland University of Applied Science's Game Studio project. The aim of the project is to advance the teaching of game programming in the university.

The Unity game engine was originally meant for 3D development, but an official support for 2D development was added to it later. The difference between these two dimensions is minimal, however, and the user interfaces work quite similarly. The game demo in the thesis borrowed its design idea from massive multiplayer online role-playing games (MMORPG) where players have typically been given a simple task or quest. The player's task is to collect 10 apples in a simple level, jumping from a platform to platform before the time runs out.

This thesis showed in detail how to make a new project in Unity and how to prepare it, how to create objects and how to add components to them, how to animate player character and how to give it multiple working animations, and how to code different functionalities to objects and to the system. In conclusion it was found that the Unity game engine suited well for 2D game development and particularly for making a 2D platform game. Making the game demo of the thesis opened up new ideas for developing the demo further in the future and what could have been done differently or better when working on this kind of project.

Keywords: game programming, unity, 2d, platform game

SISÄLLYS

1 JOHDANTO 5

2 UNITYN 2D TUKI 5

3 2D-PELIN SUUNNITTELU 7

4 2D-PELIN TOTEUTUS 11

4.1 Projektin luonti 11

4.2 Visuaalinen valmistelu 14

4.3 Pelihahmon liikkuminen 20

4.4 Pelihahmon animointi 24

4.5 Pistehallinta 38

4.6 Siirtymä skenejen välillä 42

4.7 Pelin äänimaailma 44

4.8 Viimeistely ja pelin kääntäminen 46

5 PÄÄTÄNTÖ 48

LÄHTEET 49

1 JOHDANTO

Opinnäytetyön tavoitteena on tutkia, kuinka Unity-pelimoottori soveltuu alustaksi 2D-peleille ja erityisesti 2D-tasohyppelypeleille. Tarkoituksena on suunnitella ja rakentaa toimiva pelidemo Unityn sisällä.

Työ on tehty XAMK:in Game Studios -hankkeelle, ja sen tarkoituksena on selvittää Unity-pelimoottorin tukea 2D-peleille. Unity on alun perin kehitetty 3D-pelien rakentamiseen, joten on mielenkiintoista tutkia ja nähdä kuinka 2D-pelin tekeminen onnistuu sillä.

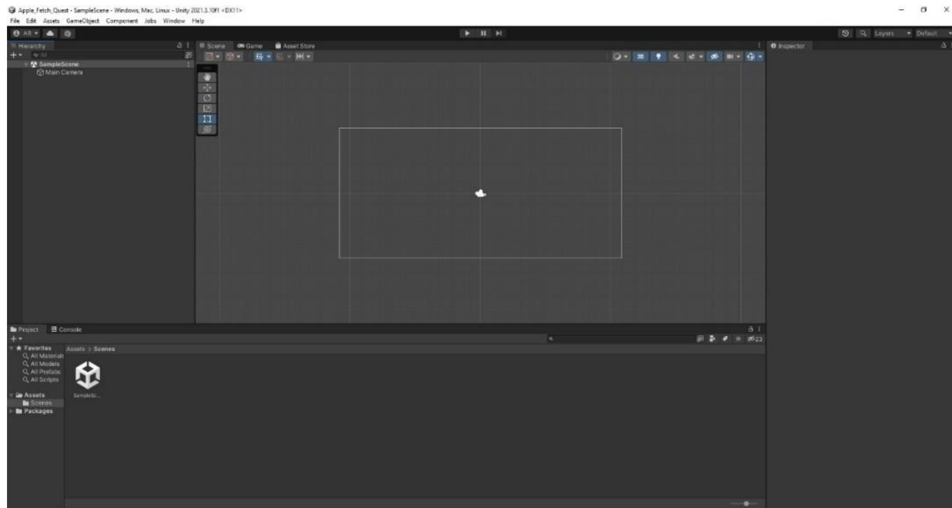
Opinnäytetyössä käydään läpi minkälaisen tuen Unity-pelimoottori tarjoaa 2D-peleille ja suunnitellaan yksinkertainen pelidemo, joka sitten rakennetaan ja dokumentoidaan. Lopuksi käydään lävitse päätelmiä ja ajatuksia, joita pelidemon rakentamisen aikana nousi.

Työn tuloksena saadaan käsitys siitä, kuinka Unity soveltuu 2D-kehitykseen. Opinnäytetyössä laaditaan ohjeistus 2D-pelin tekemiseen Unityllä, joka voi jatkossa auttaa muita aloittamaan oman 2D-projektin. Lisäksi työssä toteutetaan ohjeistuksen mukainen kaksiulotteinen tasohyppelypelin prototyyppi, joka toimii ja esittelee tämän kaltaisen pelin perustoiminnallisuudet.

2 UNITYN 2D TUKI

Unity on alun perin täysin kolmiulotteisten (3D) pelien kehitykseen tarkoitettu pelimoottori, johon on myöhemmin lisätty virallinen tuki kaksiulotteisille (2D) projekteille. Virallinen tuki toi mukanaan lukuisia lisäyksiä, jotka helpottavat kehitystä tässä ympäristössä (Unity 2D 2022).

Unityn käyttöliittymässä ei ole suuresti eroa projektin ollessa 2D tai 3D. 2D-tuen mukana Unityn käyttöliittymään projektin sisällä tuli mahdollisuus vaihtaa 2D- ja 3D-kameran välillä, joka vaikuttaa siihen, miltä projektin näkymä näyttää. Kun 2D-moodi on valittuna, kamera siirtyy ortographiseen, eli perspektivapaaseen moodiin kuten kuvassa 1 on nähtävissä. (Unity 2D 2022.)



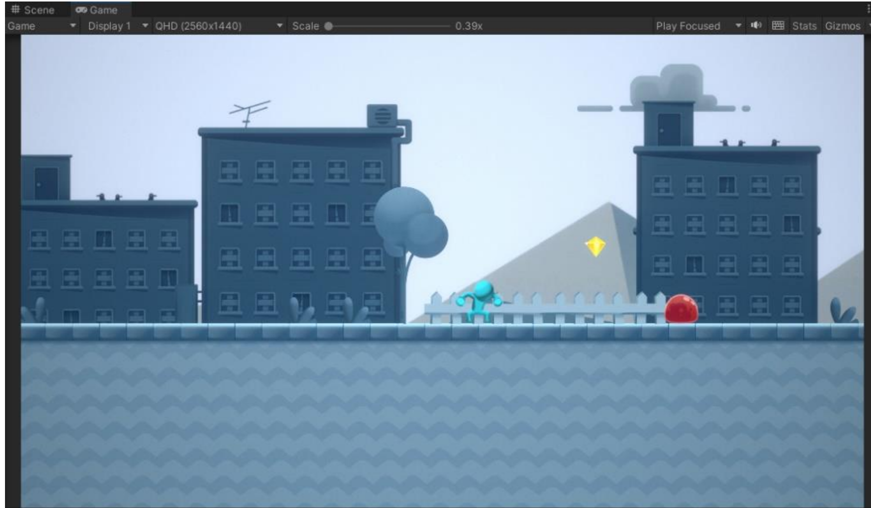
Kuva 1. Unityn käyttöliittymä 2D-näkymässä

Graafiset objektit ovat kaksiulotteisessa ympäristössä (*Sprites*), joita pystytään editoimaan Unityn sisällä *Sprite Editor* -työkalulla. 2D-projektissa spritet renderöidään *Sprite Renderer* -komponentilla, toisin kuin *Mesh Renderer* -komponentilla 3D-projekteissa. (Unity 2D 2022.)

Fysiikat toimivat kaksiulotteisessa ympäristössä samalla lailla kuin niiden vastikkeet 3D-projektissa, tosin nyt vain ”litteässä” muodossa: Box Collider -komponentti 3D-projektissa luo kuution, kun taas 2D-projektissa neliö tulee luoduksi. Jotta käyttäjä erottaisi eri ulottuvuuksiin tarkoitetut komponentit toisistaan, on Unityssä 2D-komponentteihin lisätty 2D-tunniste nimen perään (esimerkiksi edellä mainittu Box Collider -komponentti on nimeltään *Box Collider 2D* kaksiulotteisessa projektissa). (Unity 2D 2022.)

2D-projektissa Unityn näkymä on taso, jota kamera kuvaa keskeltä. Grafiikat ovat kaksiulotteisia, ne piirretään tasoon ja ne ovat käytännössä ”litteitä”. Tästä huolimatta projektissa voidaan myös halutessa yhdistellä eri ulottuvuuksia ja käyttää kolmiulotteisia objekteja, jotka sitten vain liikkuvat 2D-ulottuvuudessa (käytännössä X- ja Y-akselien suhteen). Koska objektit ovat käytännössä litteitä tasoa vasten, on tärkeää, että niiden piirtojärjestyksen on kunnossa. Tässä auttaa *Order in Layer*, jolla eri objekteille määritellään arvo, joka määrää missä järjestyksessä grafiikat näkyvät näkymässä.

Unitystä löytyy myös mahdollisuus uutta projektia luodessa valita erityinen 2D-tasohyppelypelin pohja. Tämä on täysin valmis projekti, joka sisältää suurimman osan kaksiulotteisessa tasohyppelypelissä kaivattavista ominaisuuksista.



Kuva 2. Unityn tarjoama tasohyppelypelin valmis pohja

Ainoastaan yksityiskohtaiset grafiikat on jätetty tästä pohjasta pois, kuten kuvasta 2 näkyy. Niiden sijaan käytetään paikanvaraus (*placeholder*) objekteja, jotka varaavat paikkaa varsinaisille graafeille, jotka käyttäjä voi täyttää ja räätälöidä itse myöhemmin.

3 2D-PELIN SUUNNITTELU

Tämän opinnäytetyön tarkoituksena on rakentaa toimiva 2D-tasohyppelypelin demonstraatio eli demo, jossa on pelillisesti tarvittavat perusominaisuudet ja toiminnallisuudet. Pelin aiheeksi on valittu useista roolipeleistä tuttu aihe, joka on noutotehtävä (fetch quest), mikä on tyypillistä näissä peleissä, etenkin niiden alkupuolella. Se on usein helppo ja yksinkertainen pelaajalle annettu aloitustason tehtävä, jossa tarkoitus on mennä ja kerätä tietty määrä vaadittuja esineitä saadakseen tehtävän päätökseen. Tyypillisesti tämän tapaiset tehtävät (quests) ovat roolipeleissä surullisen ylikäytettyjä pelin sisältönä, koska vaikka ne ovat kehittäjille varsin helppoja ja nopeita toteuttaa, pelaajalle ne muuttuvat nopeasti tylsiksi ja puuduttaviksi ”Mene paikkaan x ja tee asia y uudestaan ja uudestaan” -toiminnaksi, joka ei ole pelikokemuksen kannalta tyydyttävää (kuva 3). (Why I Hate: Fetch Quests 2022.)

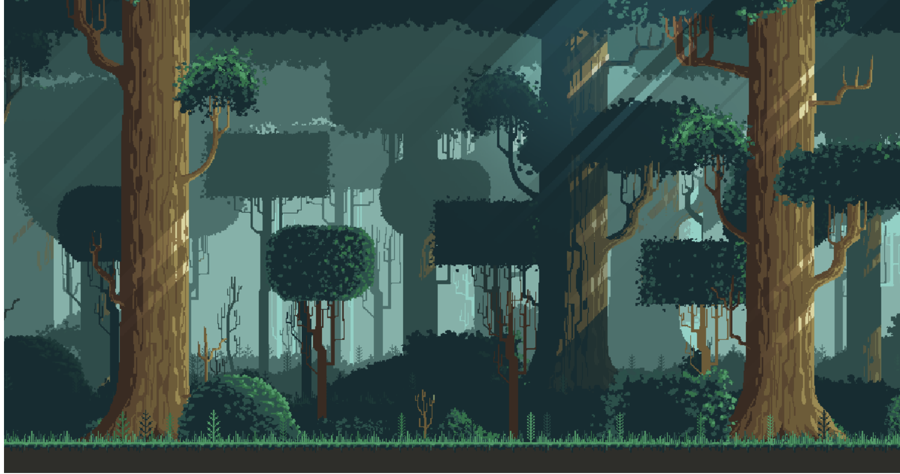
Kuva 3. Esimerkki fetch quest -tehtävästä peliyhtiö Blizzard'in World of Warcraft -massiiviroolipelissä (Why I Hate: Fetch Quests 2022)

Tässä opinnäytetyön pelidemossa fetch quest -aihetta käytetään niiden huonosta maineesta huolimatta juuri niiden yksinkertaisen idean takia (ja ihan vain puhtaasti ironisesta näkökulmasta). Pelidemossa pelaaja on tarinallisesti ritari, joka on saanut tehtäväkseen kerätä 10 omenaa ennen kuin aika loppuu kesken.

Pelidemo alkaa aloitusruudusta, josta siirrytään Enter-näppäintä painamalla varsinaiseen pelinäkymään, jossa suurin osa sen toiminnallisuudesta tapahtuu. Peli päättyy pelaajan tekemisen mukaan joko Game Over -ruutuun, joka tarkoittaa, että pelaaja hävisi pelin, tai sitten Game Win -ruutuun, joka tarkoittaa, että pelaaja voitti pelin. Kummastakin näistä lopetusruuduista päästään takaisin aloitusruutuun, josta peli alkaa uudestaan.

Pelidemon rakenne on yksinkertainen, ja se koostuu yhdestä kentästä, jossa pelihahmoa liikutetaan. Pelihahmoa pystyy liikuttamaan nuolinäppäimillä oikealle tai vasemmalle. Spacebar-näppäintä painamalla on mahdollista saada pelihahmo hyppäämään. Pelidemon kenttä muodostuu yhdestä taustakuvasta, jonka ”päällä” tai ”edessä” pelihahmo liikkuu. Kentästä löytyy tasoja, joille pelihahmo pystyy hyppäämään ja joille hahmon on hypättävä ja hypittävä niiden välillä saadakseen kaikki pelin voittoon tarvittavat esineet kerättyä.

Tyyliltään peli on piirrostyylinen ja siinä käytetyt grafiikat ovat Unityn Asset Store -kauppapaikasta tai otettu Googlen kuvahausta (opinnäytetyössä on pyritty käyttämään rojaltivapaita assetteja mahdollisuuksien mukaan). Koska käytetyt grafiikat on haalittu kokonaan tällä tavalla, ne eivät tyyleiltään täsmää aivan täydellisesti.



Kuva 4. Pelin taustakuva, joka toimii myös pelin kenttänä (Pixel Art Forest 2022)

Pelidemon hahmo toteutetaan sprite-tekniikalla, eli hahmo animoidaan käyttämällä samaa kuvaa monta kertaa pienillä eroilla (esim. jalkojen/käsien sijaintia muuttamalla), mikä antaa illuusion siitä, että hahmo elää ja liikkuu, kun se animoidaan eli pistetään toistamaan sprite-kuvia. Pelihahmon seisoessa paikallaan toistuu yksinkertainen animaatio, jossa pelihahmo liikkuu hieman.

Kuva 5. Pelihahmon paikallaanolo-spritekuva (Knight Sprite Sheet (Free) 2022)

Kun pelihahmoa liikutetaan, se näyttää juoksevan. Hahmo myös kääntyy siihen suuntaan demossa, johon sitä viedään.

Kuva 6. Pelihahmon juoksu spritekuva (Knight Sprite Sheet (Free) 2022)

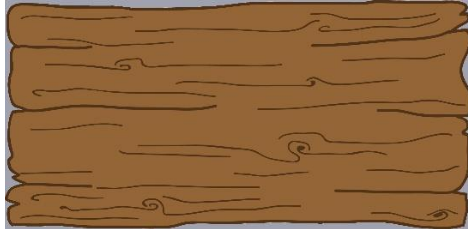
Kun pelihahmo hyppää, se käyttää oikealla olevaa spriteä, jossa se on haarat levällään, mutta hahmon laskeutuessa se käyttää vasemmalla olevaa spriteä, jossa se valmistautuu laskeutumaan (kuva 7).

Kuva 7. Pelihahmon hyppy-spritekuva (Knight Sprite Sheet (Free) 2022)

Pelidemossa hahmo pystyy sulavasti vaihtamaan eri animaatioiden välillä. Pelissä kerätään omenoita (kuva 8), joita on ripoteltu kentän eri osiin. Pelaajan on kerättävä nämä kaikki voittaakseen pelin.

Kuva 8. Pelidemon kerättävä esine, omena (Apple Cartoon Transparent PNG 2022)

Pelidemon kentästä löytyy tasoja, joille pelihahmo pystyy hyppäämään (kuva 9). Näiden sijoittelussa kenttään on hyvä pitää huoli siitä, että ne ovat riittävällä korkeudella kentän maanpinnan tasosta, jotta pelihahmo mahtuu juoksemaan niiden alta. Ne pitää myös sijoittaa niin, että pelihahmon hyppyvoima riittää hyppäämään tasolta toiselle.



Kuva 9. Puinen taso, jolle pelihahmo pystyy hyppäämään (Brown Wooden Board 2022)

Pelidemon taustalla kuuluu keskiaikainen tai fantasiatyylinen musiikki (ks. The Low Whistle - Traditional Celtic Music 2022). Pelihahmon kerätessä omenan kuuluu ääniefekti, joka kuulostaa siltä kuin puraistaisiin omenaa (ks. Apple_Crunch_17.wav – Eating Apple Crunches 2022). Pelin voittaessa kuuluu torvifanfaari (ks. Success Fanfare Trumpets 2022). Jos pelaaja häviää pelin, kuuluu pahaenteinen nauru (ks. Final Fantasy VI Kefka Laugh Sound Effect 2022).

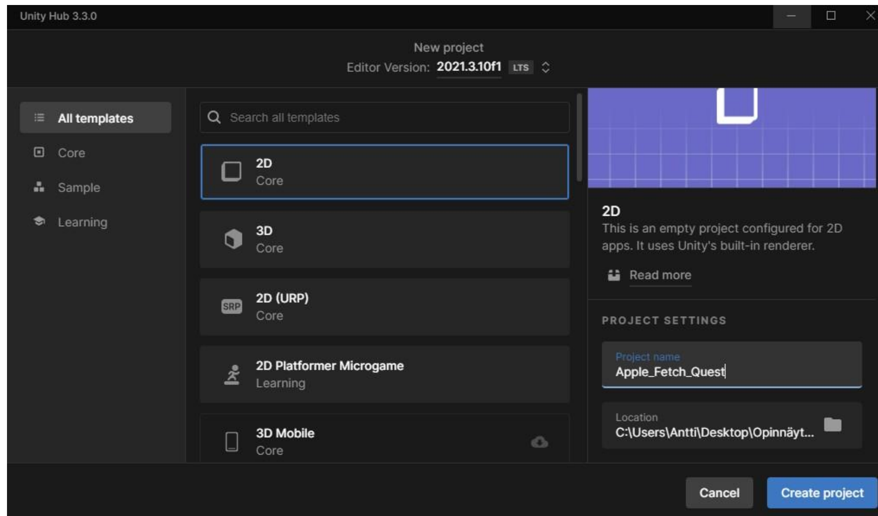
UI-elementtejä pelistä löytyy pelinäkymässä vasemmassa yläkulmassa oleva laatikko, joka kertoo, kuinka paljon on aikaa jäljellä voittaa peli, ja kuinka monta omenaa pelaaja on kerännyt.

4 2D-PELIN TOTEUTUS

Tämä luku käsittelee yksityiskohtaisesti opinnäytetyön pelidemon rakentamisen askel askeleelta.

4.1 Projektin luonti

Ennen kuin voidaan lähteä rakentamaan pelidemoa, se pitää ensiksi luoda. Tämä tapahtuu valitsemalla Unityn käyttöliittymästä uusi projekti (*New project*), joka johtaa seuraavaan näkymään (kuva 10), missä projektin alustamalli, projektin nimi sekä tallennuskohde määritellään. Alustamalliksi valitaan 2D ja projekti nimetään **Apple_Fetch_Quest**.



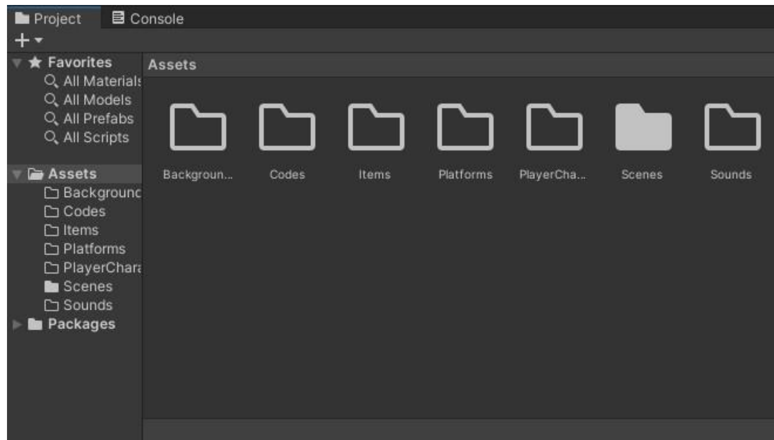
Kuva 10. Unity Hub -näkymä, jossa valitaan projektin haluttu muoto sekä määritellään nimi että tallennuskohde

Unityllä menee hetki aikaa luoda projekti, minkä jälkeen edessä on täysin tyhjä Unityn käyttöliittymän näkymä. Unity ei automaattisesti luo projektiin valmiiksi muuta kuin *Main Camera* -objektin sekä yhden kansion, **Scenes**, johon pelidemon eri skenet tallennetaan.

Jotta käyttöliittymän *Project*-ikkuna pysyisi siistinä ja työskentely helpottuisi, luodaan **Scenes**-kansion lisäksi *Assets*-valikon alle lisää kansioita, joihin varastoidaan organisoidusti erilaiset assets-välineet, joita tullaan tarvitsemaan pelidemon tekemisessä. Uusi kansio lisätään yksinkertaisesti viemällä hiiren osoitin *Project*-välilehdellä *Assets*-valikon päälle ja oikealla klikkauksella valitsemalla **Create** -> **Folder**.

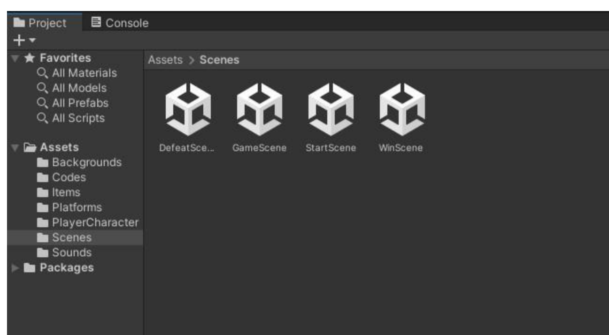
Tehdään 7 uutta kansiota (kuva 11), jotka ovat:

- **Backgrounds**, johon pelindemon taustagrafiikat tallennetaan
- **Codes**, johon tallennetaan pelidemon C#-skriptit
- **Items**, johon tallennetaan kerättävän esineen grafiikat
- **Platforms**, johon pelidemon tasografiikat tallennetaan
- **PlayerCharacter**, johon tallennetaan pelattavan hahmon grafiikat
- **Scenes**, johon pelidemon eri skenet tallennetaan
- **Sounds**, johon pelidemon musiikki ja äänet tallennetaan.



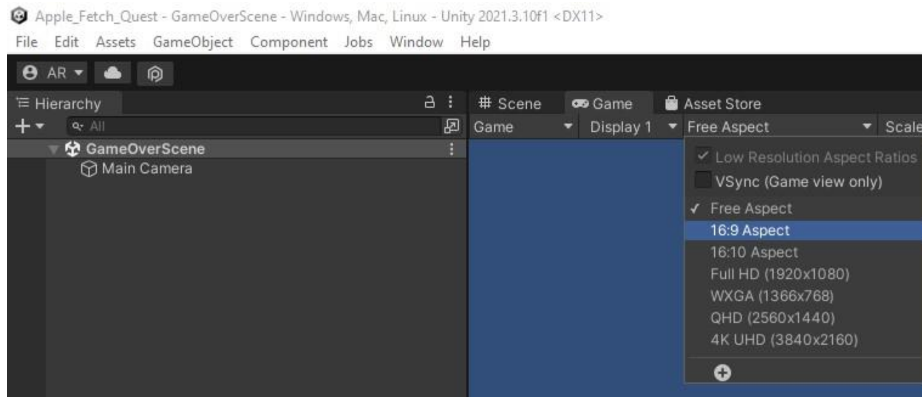
Kuva 11. Projektin kaikki tarvittavat kansiot

Unity on automaattisesti luonut **Scenes**-kansioon pelidemon ensimmäisen skenen, jonka nimi muutetaan muotoon **GameScene**. Tämä on pelidemon pääskene, jossa suurin osa työstä tehdään. Tähän kansioon lisätään skenet nimeltä **StartScene**, **DefeatScene** ja **WinScene**. Uuden skenen lisääminen tapahtuu klikkaamalla Unityn käyttöliittymän vasemmasta yläkulmasta kohdasta **File** -> **New Scene** ja valitsemalla 2D-vaihtoehto. Tämän jälkeen uusi skene tallennetaan klikkaamalla Unityn vasemmasta yläkulmasta **File** -> **Save**, minkä jälkeen voidaan valita minne kyseinen skene tallennetaan (valitaan kansio **Scenes**). Tämä toistetaan, kunnes **Scenes**-kansiossa (kuva 12) on neljä edellä nimettyä skeneä. Näitä skenejä tarvitaan myöhemmin, kun pelidemolle ohjelmoidaan siirtymät aloituksen ja lopetuksien välillä.



Kuva 12. Pelidemon kaikki skenet

Tässä kohtaa on hyvä vaihtaa *Game*-välilehdeltä pelidemon kuvasuhde. Automaattisesti se on *Free Aspect* -asetuksessa, josta se vaihdetaan *16:9 Aspec*t -asetukseen (kuva 13), mikä mahdollistaa pelin kuvasuhteen skaalautuvan oikein useimmilla näytöillä, kun pelidemo on rakennettu ja tuotu ulos Unitystä. Vaihdon jälkeen tarkistetaan, että kuvasuhde on oikein jokaisessa skenessä.



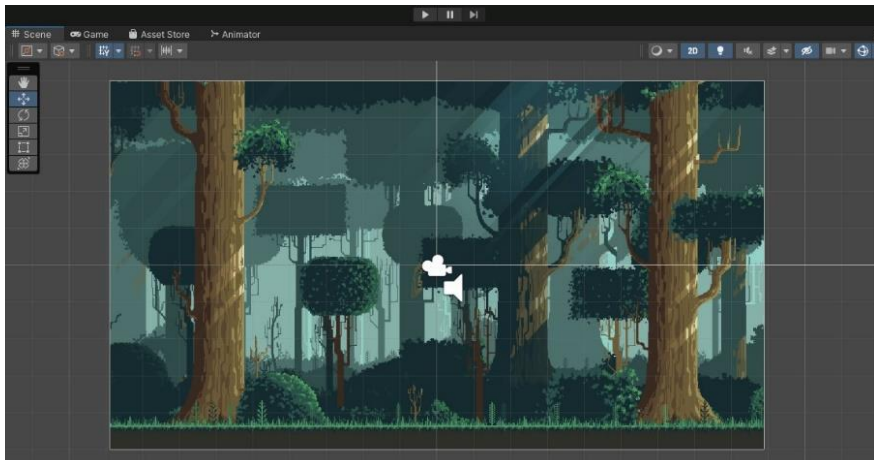
Kuva 13. Projektin kuvasuhteen vaihtaminen

Tämän jälkeen täytetään loput kansiot tarvituilla assets-objekteilla siten, että **Backgrounds**-kansioon siirretään aikaisemmin esitellyt taustagrafiikat, jotka nimetään **StartBackground**, **GameBackground**, **DefeatBackground** ja **WinBackground**. **Items**-kansioon siirretään aikaisemmin esitelty omenan graafi, joka nimetään **Apple**. **Platforms**-kansioon siirretään aikaisemmin esitelty tasograafi, joka nimetään **Platform**. **PlayerCharacter**-kansioon siirretään aikaisemmin esitellyt pelihahmon spritekalvot, jotka nimetään **Player_idle**, **Player_run** ja **Player_jump**. **Sounds**-kansioon siirretään pelidemon taustamusiikki sekä äänet, jotka nimetään **GameMusic**, **AppleCollect**, **GameOver** ja **GameWin**.

Kun kaikki assets-objektit ovat valmiina, voidaan aloittaa pelidemon visuaalinen rakentaminen.

4.2 Visuaalinen valmistelu

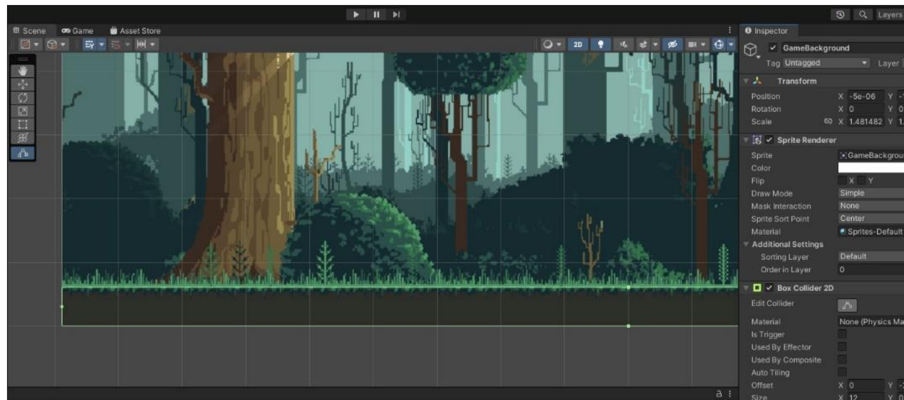
Aloitetaan pelidemon rakentaminen *GameScene*-skenessä, jossa raahataan **Backgrounds**-kansiosta **GameBackground**-kuva Unityn *Game*-ikkunaan (kuva 14). Venytetään kuva oikean kokoiseksi, jotta se peittää kameran koko alueen (tämä tehdään käsin yksinkertaisuuden nimissä, mutta kuvan pystyisi venyttämään pikselin tarkkuudella kamera-alueen kokoiseksi koodilla tai lisäämällä siihen erityisen komponentin). Tämä tehdään myös muiden skenejen taustoille.



Kuva 14. Pelin taustakuva lisätty näkymään

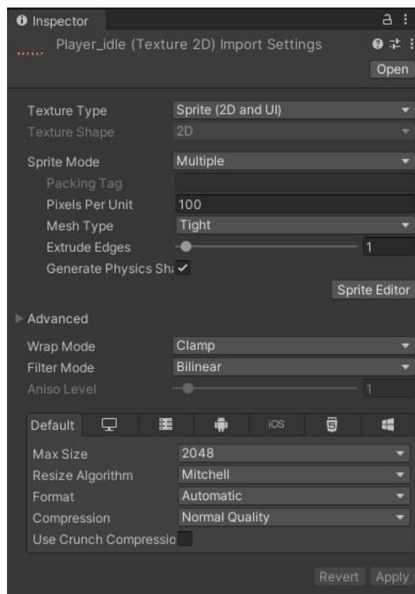
Tämän jälkeen lisätään fysiikkoja kenttään. Pelihahmon tulee seisoa taustakuvan ”maanpinnan” päällä, joten lisätään **GameBackground**-objektiin komponentti nimeltä *Box Collider 2D* (*Edge Collider 2D* toimisi myös, mutta *Box Collider 2D* -komponentillä on helpompi saada tasainen alusta aikaiseksi, joka ei myöhemmin aiheuta ongelmia pelihahmon kanssa). Komponentti lisätään klikkaamalla Unityn *Hierarchy*-välilehdeltä **GameBackground**-kuva aktiiviseksi, minkä jälkeen Unityn UI:n vasemmasta yläkulmasta klikataan **Component** -> **Physics 2D** (tärkeää, että valitaan aina 2D vaihtoehto, kun tehdään 2D-työtä!) -> **Box Collider 2D**.

Lisäämällä tämän komponentin se luo koko taustakuvan päälle vihreän laatikon, jota voidaan muokata käymällä *Inspector*-ikkunassa painamassa *Box Collider 2D* -komponentin alla olevaa nappia nimeltä *Edit Collider*. Pienennetään vihreä laatikko noin taustakuvan maanpinnan korkuiseksi, kuten kuvassa 15, ja annetaan sen olla leveydeltään koko taustakuvan levyinen. Box Collider mahdollistaa sen, että kun pelihahmolla on painovoima, hahmo ei putoa kuvasta pois vaan seisoo tukevasti collider-komponentin päällä. Tätä käytetään myös myöhemmin koodissa tunnistamaan tiettyjä asioita.



Kuva 15. **GameBackground**-objektiin lisätyn *Box Collider 2D* -komponentin säätäminen oikean kokoiseksi

Tämän jälkeen viimeistellään pelihahmo valmiiksi. Aloitetaan **Player_idle** -spritekuvasta. **PlayerCharacter**-kansiossa klikataan kuva aktiiviseksi, mikä aukaisee *Inspector*-ikkunan auki, josta nähdään kuvan kaikki asetukset (kuva 16). *Sprite Mode* -kohdasta käydään vaihtamassa asetus **Single** -> **Multiple**, jotta Unity ymmärtää kyseessä olevan useampi kuva. Vaihdon jälkeen klikataan alla olevaa *Apply*-nappia, joka tallentaa valinnan.



Kuva 16. Spritekalvojen editointi

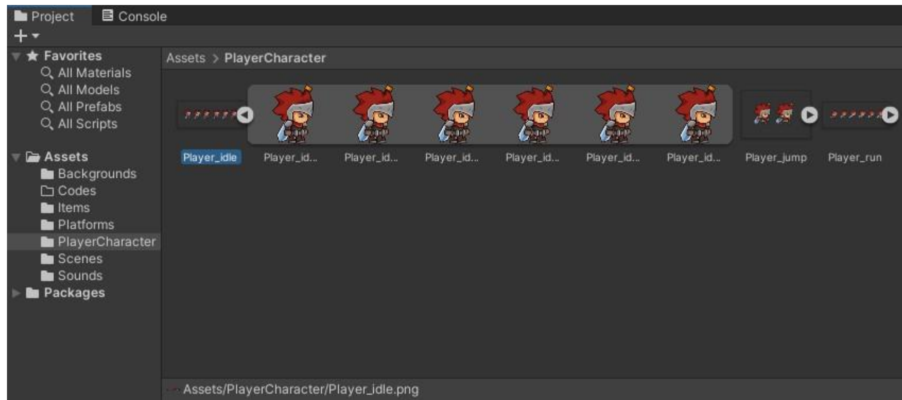
Tämän jälkeen klikataan *Sprite Editor* -nappia, joka avaa uuden ikkunan (kuva 17), jossa voidaan leikellä spritekuva yksittäisiksi kuviksi. Tämä tapahtuu yksinkertaisesti klikkaamalla *Sprite Editor* -ikkunan vasemmasta yläkulmasta *Slice*-nappia, josta avautuu toinen ikkuna, josta klikataan toisen kerran *Slice*-nappia.

Kuva 17. Unityn *Sprite Editor,* jossa spritekalvo voidaan leikata yksittäisiksi kuviksi

Spritekuvalle pitäisi nyt olla ilmestynyt jokaisen spriten ympärille valkoinen laatikko automaattisesti (kuva 18), joka merkitsee, että jokainen laatikon sisällä oleva sprite on nyt yksittäinen kuva (Unity usein tekee tämän varsin hyvin ja tarkasti taustakuvattoman spritekalvon kanssa, joten käyttäjän ei yleisesti tarvitse itse editoida spriten leikkaamista).

Kuva 18. *Sprite Editor* automaattisesti leikkaa spritet yksittäisiksi kuviksi

Tallennetaan työ klikkaamalla *Sprite Editor* -ikkunan oikeassa yläkulmassa olevaa *Apply*-nappia. Voidaan tarkistaa vielä **PlayerCharacter**-kansiossa, että leikkaaminen onnistui painamalla pientä nuolta **Player_idle**-kuvan kohdalla, minkä pitäisi avata spritekuva leveämmäksi ja näyttää kuvan spritet vierekkäin yksittäisinä kuvina (kuva 19). Tämän jälkeen käydään tekemässä sama **Player_run**- ja **Player_jump**-spritekuville.



Kuva 19. Miltä onnistunut spritekuvan leikkaaminen yksittäisiksi kuviksi pitäisi näyttää *Project*-ikkunassa

Raahataan seuraavaksi yksi pelihahmon **Player_idle**-kuvista ruudulle, jotta voidaan skaalata seuraavat esineet sopivan kokoisiksi. Pelihahmo itsessään on jo sopivan kokoinen, joten sen kokoasetuksiin ei tarvitse koskea.

Seuraavaksi valmistetaan kerättävä esine. **Items**-kansiosta valitaan **Apple**-kuva ja raahataan se pelidemon taustakuvan päälle. Oletuksena **Apple**-objekti jää **GameBackground**-objektin taakse, joten *Inspector*-ikkunassa sen asetuksista määritetään, että sen *Order in Layer* on 1. Tämä pistää **Apple**-objektin ilmestymään **GameBackground**-objektin päälle ja pysymään aina sen päällä. Seuraavaksi muutetaan **Apple**-objektin kokoa, että se näyttää sopivalle suhteessa pelihahmoon (kuva 20).

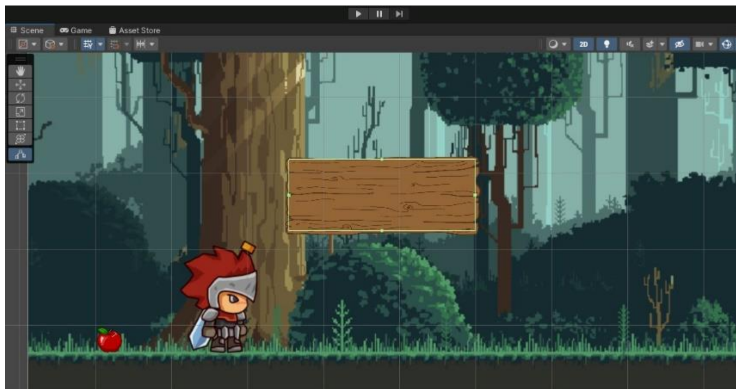
Kuva 20. Skaalataan omena suunnilleen sopivan kokoiseksi suhteessa pelihahmoon

Tämän jälkeen klikataan **Apple**-objekti valituksi *Hierarchy*-ikkunasta ja Unityn yläkulmasta klikataan **Component** -> **Physics 2D** -> **Circle Collider 2D**, mikä antaa *Apple*-objektille ympyrän muotoisen collider-componentin. **Apple**-objektin *Inspector*-ikkunassa käydään *Circle Collider 2D* -komponentin alta klikkaamassa *Edit Collider* -nappia, mikä mahdollistaa collider-komponentin koon

muuttamisen. **Apple**-objektin collider-komponentti muutetaan noin objektin kokoiseksi. Tämän jälkeen klikataan samoista asetuksista raksi *Is Trigger* -asetukseen, mikä mahdollistaa myöhemmin Unityn tajuamaan, kun objektiin osutaan toisella objektilla.

Seuraavaksi valmistetaan pelidemon tasot, joille pelihahmo pystyy hyppäämään myöhemmin. **Platforms**-kansiosta otetaan **Platform**-kuva ja raahataan se näkymään. Sen asetuksista käydään muuttamassa *Order in Layer* on 1, jotta se ei katoa **GameBackground**-objektin taakse. Tämän jälkeen lisätään samalla lailla **Platforms**-objektille *Box Collider 2D* -komponentti, kuten annettiin **GameBackground**-objektin ”maanpinnalle”. Unityn UI:n vasemmasta yläkulmasta valitaan **Component** -> **Physics 2D** -> **Box Collider 2D**.

Tehdään **Platform**-objektin collider-komponentista hieman pienempi, kuin objekti itse, jotta pelihahmon collider (lisätään myöhemmin) ei ota kiinni **Platforms**-objektin kanssa. Pidetään huoli siitä, että taso on tarpeeksi korkealla maanpinnasta, kuten kuvassa 21, että pelihahmo pystyy tarpeen tullen juoksemaan sen alta ilman, että tasojen collider-komponentit ottaisivat kiinni.



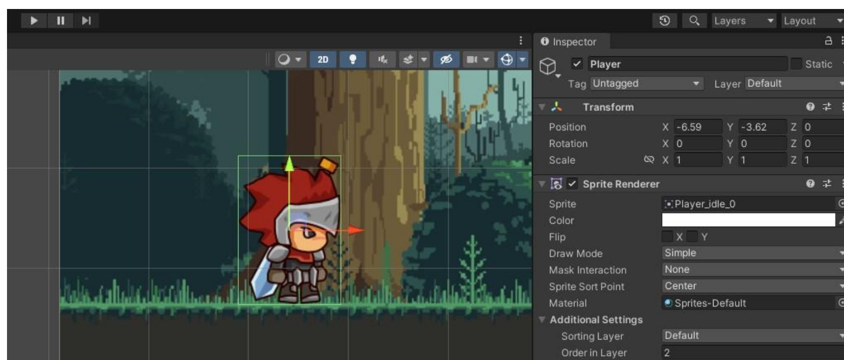
Kuva 21. **Platforms**-objekti skaalattu sopivan kokoiseksi pelihahmoon nähden ja sopivalla korkeudella maan pinnasta

Myöhemmin tasoja asetellessa on tärkeää, että pelihahmo yltää hyppäämään tasolta tasolle tai muuten hahmo ei saa kerättyä kaikkia omenoita. On säädettävä joko pelihahmon hyppyvoimaa ja/tai aseteltava tasoja toisin, jotta tämä onnistuu.

4.3 Pelihahmon liikkuminen

Tämä luku perustuu osittain videotutoriaaliin (Tilemap & Tile Palette 2022). Aloitetaan pelihahmon luonti tekemällä tyhjä neliön muotoinen objekti *Hierarchy*-ikkunaan. Tämä tapahtuu klikkaamalla hiiren oikealla painikkeella *Hierarchy*-ikkunassa ja valitsemalla **2D Object** -> **Sprites** -> **Square**. Nimetään tämä objekti **Player**.

Seuraavaksi avataan **PlayerCharacter**-kansio, josta valitaan **Player_idle**-kuvista ensimmäinen. Se raahataan pelihahmon *Inspector*-ikkunassa *Sprite Renderer* -komponentin alla olevan *Sprite*-kohtaan, jossa lukee tällä hetkellä Square. Tämä muuttaa valkoisen laatikon pelihahmon näköiseksi. *Sprite Renderer* -komponentin alta käydään myös muuttamassa pelihahmon *Order in Layer* 2, jotta se näkyy taustakuvan päällä. Kuvassa 22 nähdään, kuinka tämä on toteutettu oikein.

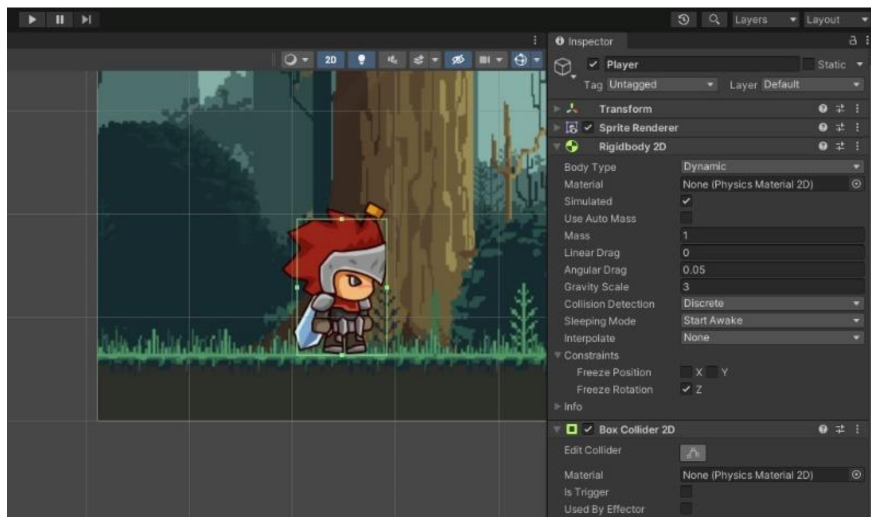


Kuva 22. Pelihahmo lisätty peliin

Seuraavaksi lisätään pelihahmolle fysiikoita. Tämä tapahtuu klikkaamalla pelihahmo aktiiviseksi *Hierarchy*-ikkunasta ja sitten klikkaamalla Unityn vasemmasta yläkulmasta **Component -> Physics 2D** -> **Rigidbody 2D**. Tämä antaa pelihahmolle massan ja painovoiman. Jos nyt siirretään pelihahmon ilmaan *Scene*-ikkunassa ja laitetaan Unity ajamaan skeneä, pelihahmo tippuisi maan lävitse pois skenestä. Tämä johtuu siitä, että pelihahmolla ei ole collider -komponenttia, joka pysäyttäisi sen osuessa taustakuvan maan tasolla olevaan *Box Collider 2D* -komponenttiin. Lisätään collider-komponentti pelihahmolle samalla tavalla kuin lisättiin *Rigidbody 2D*; klikataan pelihahmo aktiiviseksi *Hierarchy*-ikkunasta ja sen jälkeen Unityn yläkulmasta **Component** -> **Physics 2D** -> **Box Collider 2D**. Pelihahmolle voitaisiin myös valita *Capsule*

Collider 2D, joka laatikon sijaan tekisi kapselin muotoisen collider-komponentin pelihahmon ympärille. Tämä olisi parempi siinä mielessä, että pyöreämpi collider-komponentti pelihahmossa ei osuisi niin pahasti tai ottaisi kiinni tasojen neliömäiseen collider-komponenttiin. Syy, minkä takia tässä demossa käytetään neliötä kotelon sijaan, on se, että neliömäinen muoto helpottaa tunnistamaan paremmin, kun collider-komponentit törmäävät.

Jos nyt ajetaan skeneä, pelihahmo putoaa ilmasta maahan ja jää seisomaan taustan maanpinnan päälle. Editoidaan pelihahmon collider-komponenttia sen verran, että se on vähän pelihahmoa pienempi (kuva 23). Käydään vielä tekemässä yksi pieni asia, että lukitaan pelihahmon sprite Z-akselin suhteen. Tämä tapahtuu klikkaamalla **Player**-objektia ja oikealla *Inspector*-ikkunassa *Rigidbody2D*-komponentin alta kohdasta *Constraits* klikataan ruksi *Freeze Rotation Z* -kohdassa. Tästä on hyötyä myöhemmin, kun ohjelmoidaan pelihahmoa, koska se estää hahmoa kallistumasta eri suuntiin.



Kuva 23. Pelihahmon *Box Collider 2D* -komponentin editoiminen

Pelihahmon ohjelmointi aloitetaan kirjoittamalla yksinkertainen koodi, joka liikuttaa hahmoa. Avataan **Codes**-kansio, jossa klikataan hiiren oikealla painikkeella ja valitaan **Create** -> **C# Script**, joka luo uuden C#-skriptiedoston. Nimetään tämä koodiskripti **PlayerMovement** ja raahataan se **Codes**-kansiosta *Hierarchy*-ikkunassa olevaan **Player**-objektiin. Kaksoisklikkaamalla **PlayerMovement**-skriptiä pitäisi aueta Microsoft Visual Studio, jossa kirjoitetaan koodia.

PlayerMovement-skriptissä kirjoitetaan lyhyt koodipätkä (kuva 24), joka hakee pelihahmon *Rigidbody 2D* -komponentin ja liikuttaa hahmoa tietyn määrän pystysuunnassa, kun painetaan näppäimistön Spacebar-painiketta.

```
13      // Update is called once per frame
        Unity Message | 0 references
14      void Update()
15      {
16          // Hahmo hyppää painamalla spacebar
17          if (Input.GetKeyDown("space"))
18          {
19              GetComponent<Rigidbody2D>().velocity = new Vector2(0, 14f);
20          } // if
21      } // Update
22  } // class
23
```

Kuva 24. Koodi, jolla pelihahmo hyppää

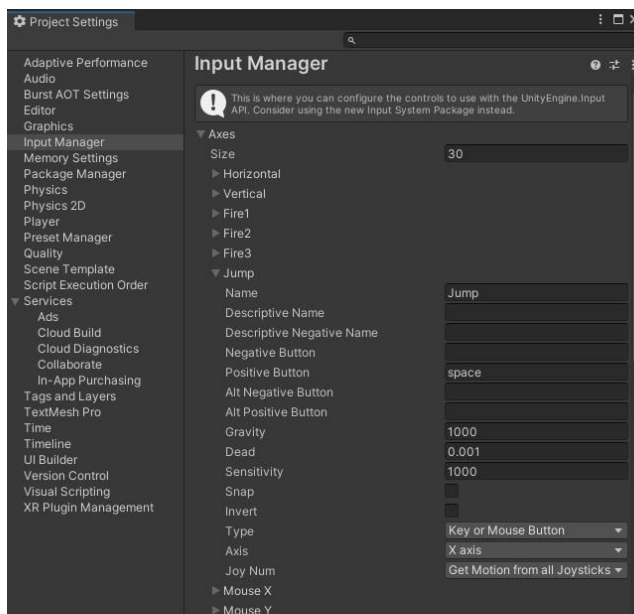
Tallennetaan koodi ja tarkastetaan Unityssä kuinka se toimii. Pelihahmon pitäisi nyt liikkua vaakasuoraan ylöspäin, kun painetaan Spacebar-näppäintä. Tässä kohtaa voidaan *Rigidbody 2D* -komponentissa vaikuttaa siihen, kuinka paljon massaa pelihahmolla on, mikä vaikuttaa pelihahmon hyppäämiseen; määritellään pelihahmon *Gravity Scale* arvoksi 3 (demossa pelihahmon massan arvo on 3 ja ”hyppyvoiman” arvo on 14). Toistaiseksi on mahdollista kaksoishypätä ilmassa, mutta tämä korjataan myöhemmin.

Muokataan koodia kevyempään muotoon (kuva 25), jossa *GetComponent<Rigidbody2D>()* ei kutsuta joka kerta, vaan vain kerran ohjelman alussa. Tämä tapahtuu määrittämällä *Rigidbody2D* yksityiseksi muuttujaksi nimeltään *rb*, jota voidaan sitten käyttää korvaamaan *GetComponent<Rigidbody2D>()* -komento. Muutetaan myös tämän skriptin *void Start()* ja *void Update()* yksityisiksi, jotta muut skriptit eivät voi käyttää niitä. Myöskään ei tarvitse koodata skriptiä käyttämään nimenomaan spacebar-painiketta hyppäämiseen, vaan voidaan käyttää universaalia hyppypainiketta, joka on määritetty Unityssä.

```
   Unity Script (1 asset reference) | 0 references
5  public class PlayerMovement : MonoBehaviour
6  {
7      private Rigidbody2D rb; // Luodaan variaabe
8
9      // Start is called before the first frame u
       Unity Message | 0 references
10     private void Start()
11     {
12         rb = GetComponent<Rigidbody2D>();
13
14     } // Start
15
16     // Update is called once per frame
       Unity Message | 0 references
17     private void Update()
18     {
19         // Hahmo hyppää painamalla spacebar
20         if (Input.GetButtonDown("Jump"))
21         {
22             rb.velocity = new Vector2(0, 14f);
23
24         } // if
25     } // Update
26 } // class
27
```

Kuva 25. Kevyempi versio hyppykoodista järjestelmälle

Unityn yläkulmasta klikkaamalla **Edit** -> **Project Settings** pääsee näkemään *Project Settings* -ikkunan, jonka alla on *Input Manager* -valikko, josta näkee kaikki erilaiset Input-komennot. Sieltä nähdään, että hahmon hyppäämiseen voi käyttää *Input.GetButtonDown("Jump")* -komentoa (tekemällä näin mahdollistetaan useampien erillisten näppäimistön näppäinten toimimisen tekemään samaa asiaa tai tarjotaan tuku pelikonsolien ohjaimien näppäimille). *Input Manager* -näkymässä voi myös uudelleen määrittää, mitä eri toiminnot tekevät (Unity 2D 2022.)



Kuva 26. Unityn *Input Manager* -valikko

Hahmon liikkumiseksi oikealle ja vasemmalle lisätään **PlayerMovement**-koodiskriptiin pätkä uutta koodia, joka näkyy kuvassa 27 (Player Movement 2022).

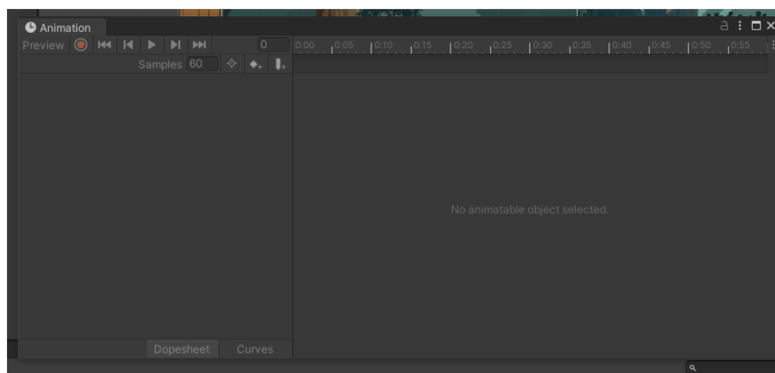
```
// Update is called once per frame
Unity Message | 0 references
private void Update()
{
    // Hahmo liikkuu X-akselin suhteen oikealle ja vasemmalle
    float dirX = Input.GetAxisRaw("Horizontal");
    rb.velocity = new Vector2(dirX * 7f, rb.velocity.y);

    // Hahmo hyppää painamalla spacebar
    if (Input.GetButtonDown("Jump"))
    {
        rb.velocity = new Vector2(rb.velocity.x, 14f); //Haeta
    } // if
} // Update
```

Kuva 27. Koodi, jolla hahmo liikkuu oikealle tai vasemmalle (Player Movement 2022)

Korvataan myös aikaisemmassa koodipätkässä, joka mahdollisti hyppäämisen, aikaisempi nolla arvolla *rb.velocity.x*, joka kertoo, miten koodin tulee toimia X-akselin suhteen, kun painetaan hyppynappia.

4.4 Pelihahmon animointi

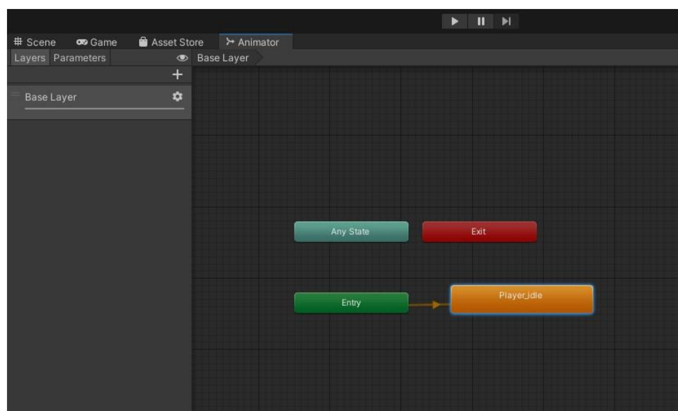
Seuraavaksi animoidaan pelihahmoa. Avataan **PlayerCharacter**-kansio, jossa säilytetään pelattavan hahmon spritekuvia. Klikataan hiiren oikealla näppäimellä kansion sisällä ja valitaan **Create** -> **Animation**. Unity luo kansioon uuden tyhjän animaation, joka nimetään **Player_idle**. Tämän jälkeen **Player_idle**-animaatio raahataan *Hierarchy*-ikkunassa **Player**-objektiin, mikä luo toisen tiedoston nimeltä **Player** (tämä tiedosto yhdistää animaation pelihahmo-objektiin). Tämän jälkeen Unityn käyttöliittymästä klikataan vasemmasta yläkulmasta **Window** -> **Animation** -> **Animation**, mikä avaa Unityn *Animation*-ikkunan, jossa animaatioita pystyy tarkastamaan ja säätämään. Klikkaamalla *Hierarchy*-ikkunasta **Player**-objektin aktiiviseksi, se avaa **Player_idle**-animaation *Animation*-ikkunaan (kuva 28). (Animation & Animator 2022.)



Kuva 28. Unityn *Animation*-ikkuna

Tällä hetkellä animaatio on tyhjä. Lisätään kaikki **Player_idle**-spritekuvat **Player_idle**-animaatioon ensiksi klikkaamalla ensimmäistä yksittäistä kuvaa ja sen jälkeen klikkaamalla muita kalvon kuvia samalla kun vasenta Ctrl-näppäintä painetaan. Tämän jälkeen kaikki spritekuvan yksittäiset kuvat pitäisi olla valittu, minkä jälkeen ne raahataan *Animation*-ikkunaan. Nyt voidaan testata, mille animaatio näyttää *Scene*-ikkunassa, kun painetaan Play-*näppäintä Animation*-ikkunassa. Oletuksena Unity ajaa animaatiota liian nopeasti, mutta siihen voidaan vaikuttaa säätämällä *Sample Rate* -arvoa (oletuksena on 60). Oletuksena Unity ei näytä tätä *Animation*-ikkunassa, mutta sen saa avattua klikkaamalla kolmea päällekkäistä pistettä *Animation*-ikkunan oikeassa yläkulmassa (juuri ruksin alla, joka sulkisi ikkunan) ja valitsemalla *Show Sample Rate*. Asetaan *Sample Rate* -arvoksi 12, joka tuntuu sopivalle tämän pelidemon tarpeisiin.

Seuraavaksi avataan *Animator*-ikkuna (kuva 29), mikä tapahtuu klikkaamalla Unityn käyttöliittymän vasemmasta yläkulmasta **Window** -> **Animation** -> **Animator**. Tässä ikkunassa pystytään lisää säätämään animaatioiden asetuksia sekä niiden siirtymisiä animaatiosta toiseen. Tällä hetkellä pelihahmossa on yksi animaatio, joka on **Player_idle** (pelihahmo tekee yhtä liikettä seisoessaan paikalla).



Kuva 29. Unityn *Animator*-ikkuna

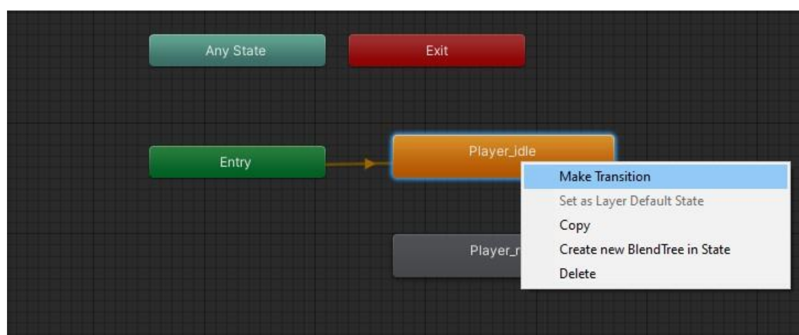
Jos nyt painetaan play-nappia, pelihahmo suorittaa animaation vain kerran ja sitten pysähtyy. Jotta animaatio pyörisi jatkuvasti, pitää Project-välilehdeltä **PlayerCharacter**-kansiossa käydä klikkaamassa **Player_idle**-animaatiota ja sen *Inspector*-ikkunassa käydä klikkaamassa ruksi kohtaan *Loop Time*. Nyt, kun pelihahmon idle-animaatio on valmis, tehdään muut animaatiot.

PlayerCharacter-kansiossa hiiren oikealla näppäimellä klikataan **Create** -> **Animation** ja nimetään uusi animaatio **Player_run**. Raahataan se **Player**-objektiin *Hierarchy*-ikkunassa (tämä saa sen näkyviin *Animator*-ikkunassa). Myös *Animation*-ikkunassa saadaan tehty animaatio tarkasteltavaksi klikkaamalla animaation nimen kohdalta pientä alaspäin näyttävää nuolta, minkä jälkeen pystyy siirtymään eri animaatioiden välillä.

Animation-ikkunassa tehdään sama kuin aikaisemman animaation kanssa, eli klikataan **Player_run**-spritekalvosta kaikki yksittäiset kuvat valituiksi ja raahataan ne *Animation*-ikkunaan **Player_run**-animaatioon. Tämän jälkeen säädetään *Sample Rate* -arvo 12. Painamalla play-nappia nähdään, että pelihahmo juoksee nyt *Scene*-ruudulla. Tämän jälkeen asetetaan animaatio jatkuvaksi; *Project*-välilehdeltä **PlayerCharacter**-kansiossa klikataan **Player_idle**-animaatiota ja sen *Inspector*-ikkunassa klikataan ruksi kohtaan *Loop Time*.

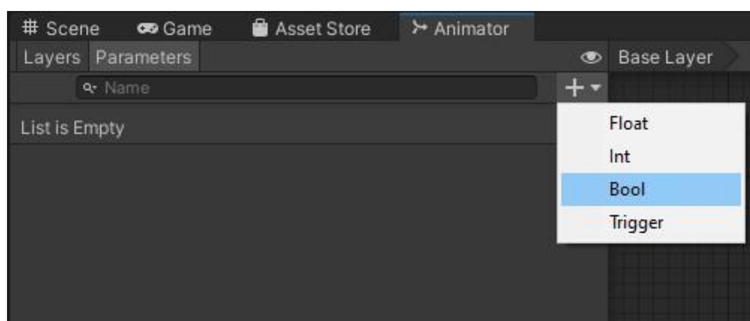
Nyt kun animaatioita on useampi, ne pitää saada toimimaan yhdessä, eli pitää saada siirtymä animaatiosta toiseen toimimaan. Järjestetään *Animator*-ikkunassa animaatiot siistiin järjestykseen (niiden paikoilla ei ole vaikutusta), ja hiiren oikealla klikataan **Player_idle**-animaation päällä (kuva 30) ja valitaan *Make Transition*, joka vedetään **Player_run**-animaatioon. Sitten **Player_run**-

animaation päällä tehdään sama ja vedetään se **Player_idle**-animaatioon. Nyt on siirtymät luotu, mutta seuraavaksi pitää vielä asettaa ehdot (*Conditions*), joilla siirtymä tapahtuu.



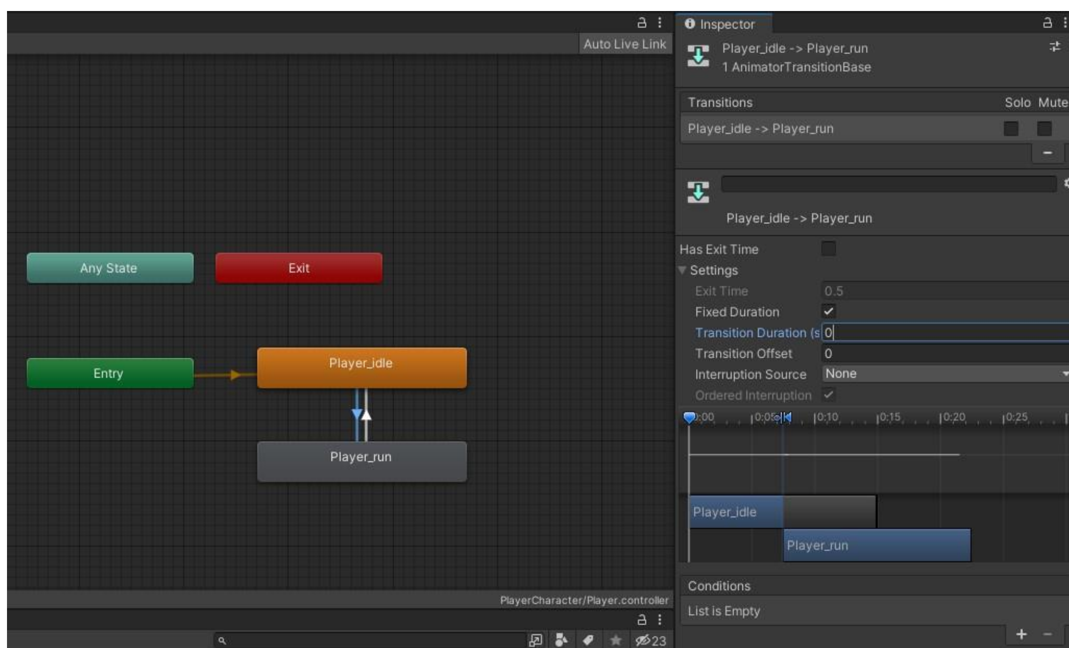
Kuva 30. **Player_idle**-animaatiosta tehdään siirtymä **Player_run**-animaatioon

Animator-ikkunassa avataan *Parameters*-välilehti, johon voidaan määrittää muuttujan arvo, joka vaikuttaa animaatioihin. Klikataan *Parameters*-välilehden hakukentän oikealla puolella olevan plusmerkin ja alaspäin näyttävän nuolen kohdalta ja valitaan auenneesta valikosta *Bool*. Tämä luo boolean muuttujan, joka nimetään **Running** (kuva 31).



Kuva 31. Luodaan boolean muuttuja nimeltä **Running**

Seuraavaksi klikataan **Player_idle**-animaatiosta **Player_run**-animaatioon menevää nuolta, joka avaa siirtymän tarkasteltavaksi *Inspector*-ikkunaan (kuva 32). Näistä asetuksista poistetaan ruksi kohdasta *Has Exit Time* ja määritellään *Settings*-valikosta *Transition Duration (s)* -asetuksen arvoksi 0. Tämä mahdollistaa sen, että siirtymä animaatiosta toiseen on välitön ja Unity ei odota, että animaatio suorittaa itsensä loppuun ennen siirtymää toiseen animaation (käytännössä tekee animaatioiden siirtymästä sulavampaa). (Animation & Animator 2022.)



Kuva 32. Animaation siirtymän sulavoittaminen

Tämän jälkeen *Inspector*-ikkunassa asetetaan siirtymälle ehto *Conditions*-valikosta painamalla plussaa ja valitsemalla aikaisemmin tehty **Running**-muuttuja ja annetaan sen olla *true*. Seuraavaksi valitaan siirtymä, joka menee **Player_run**-animaatiosta **Player_idle**-animaatioon. Sen asetuksista poistetaan myös ruksi kohdasta *Has Exit Time* ja asetaan *Transition Duration (s)* -asetus 0. Sille luodaan alhaalta samalla lailla ehto, kuten edellisen siirtymänkin kanssa, mutta tällä kertaa valitaan *false*.

Nyt on luotu valmiiksi ehdot, joilla pelihahmo paikallaan seisoessa suorittaa **Player_idle**-animaatiota, mutta juostessa siirtyy suorittamaan **Player_run**-animaatiota. Itsessään tämä ei vielä toimi täysin, vaan vaatii lisää koodaamista.

Avataan **Codes**-kansio ja avataan **PlayerMovement**-tiedosto Visual Studiossa. Kirjoitetaan aikaisemman koodin perään uusi pätkä, jossa asetetaan arvo, jonka avulla tiedetään, onko pelihahmo paikoillaan vai juoksussa (kuva 33).

```
30
31              // Asetaan arvo, jolla tiedetään juostaanko vai ei
32              if (dirX > 0f)
33              {
34                  anim.SetBool("Running", true);
35              }
36              else if (dirX < 0f)
37              {
38                  anim.SetBool("Running", true);
39              }
40              else
41              {
42                  anim.SetBool("Running", false);
43              }
44          } // Update
45      } // class
46
```

Kuva 33. Asetaan arvo, jolla animaatio tietää mitä se tekee

Lisäksi lisätään koodin alkuun uusi yksityinen muuttuja *private Animator anim* ja kutsutaan sitä ohjelman alussa komennolla *anim = GetComponent<Animator>();* (kuva 34).

```
10      // Start is called before the first frame update
        Unity Message | 0 references
11      private void Start()
12      {
13          rb = GetComponent<Rigidbody2D>(); // Kutsutaan variaabelilla
14          anim = GetComponent<Animator>(); // Kutsutaan variaabelilla
15
16      } // Start
```

Kuva 34. Yksityiset variaabelit, joita kutsutaan kerran ohjelman alussa

Nyt testattaessa Unityn puolella nähdään, että pelihahmo liikkuessa näyttää juoksevan ja paikallaan seisoessa toistavan eri animaatioita.

Parannetaan tehtyä koodia pitämällä se siistimpänä ja helpommin luettavana (kuva 35) luomalla uusi yksityinen metodi nimeltä *UpdateAnimationState*. Lisätään siihen äsken kirjoitettu koodi ja kirjoitetaan sen tilalle *UpdateAnimationState();*, joka kertoo edelliselle metodille, että sen pitää siirtyä uuteen metodiin, kun se on ajettu loppuun.

```
33              UpdateAnimationState();
34          } // Update
35
            1 reference
36          private void UpdateAnimationState()
37          {
38              // Asetaan arvo, jolla tiedetään juostaanko vai ei
39              if (dirX > 0f)
40              {
41                  anim.SetBool("Running", true);
42              }
43              else if (dirX < 0f)
44              {
45                  anim.SetBool("Running", true);
46              }
47              else
48              {
49                  anim.SetBool("Running", false);
50              }
51          }
52      } // class
```

Kuva 35. Siistitympi versio koodista, jolla animaatio tietää mitä se tekee

Poistetaan myös *private void Update()* -metodin alta float-komento *dirX = Input.GetAxisRaw(”Horizontal”);* -koodipätkästä, koska siirretään se yksityikseksi variaabeliksi *private float dirX = 0f;* (kuva 36).

```
    Unity Script (1 asset reference) | 0 references
5   public class PlayerMovement : MonoBehaviour
6   {
7       private Rigidbody2D rb; // Määritellään yksityinen variaabeli
8       private Animator anim; // Määritellään yksityinen variaabeli
9
10      private float dirX = 0f;
```

Kuva 36. Animaation perustila on 0, eli **Player_idle**-animaatio

Seuraavaksi korjataan pelihahmon liikkeestä se, että hahmo ei osaa kääntää itseään toiseen suuntaan mentäessä (oletuksena hahmon kasvot ovat koko ajan oikealle päin, vaikka liikuttaisiin vasemmalle). Halutaan, että hahmo kääntyy siihen suuntaan päin, mihin se on menossa. Tämä tapahtuu komennoilla, joilla haetaan Unityssä **Player**-objektin *Sprite Renderer* -komponentti ja pannaan koodi kääntämään pelihahmoa X-akselin mukaisesti haluttaessa.

Aluksi lisätään uusi yksityinen muuttuja *private SpriteRenderer sprite*, jota haetaan ohjelman alussa komennolla ”sprite = GetComponent<SpriteRenderer>();” (kuva 37).

```
    Unity Script (1 asset reference) | 0 references
5   public class PlayerMovement : MonoBehaviour
6   {
7       private Rigidbody2D rb; // Määritellään yksityinen variaabeli
8       private SpriteRenderer sprite;
9       private Animator anim; // Määritellään yksityinen variaabeli
10
11      private float dirX = 0f;
12
13      // Start is called before the first frame update
        Unity Message | 0 references
14      private void Start()
15      {
16          rb = GetComponent<Rigidbody2D>(); // Kutsutaan variaabelil
17          sprite = GetComponent<SpriteRenderer>();
18          anim = GetComponent<Animator>(); // Kutsutaan variaabelill
19
20      } // Start
```

Kuva 37. Tehdään yksityinen variaabeli, jota sitten haetaan ohjelman alussa

Tämän jälkeen lisätään äsken kirjoitettuun koodin seuraavat pätkät, jotka kääntävät hahmoa oikeaan suuntaan X-akselin suhteen (kuva 38).

```
39      1 reference
        private void UpdateAnimationState()
40      {
41          // Asetaan arvo, jolla tiedetään juostaanko vai ei
42          if (dirX > 0f)
43          {
44              anim.SetBool("Running", true);
45              sprite.flipX = false; // Hahmoa käännetään
46          }
47          else if (dirX < 0f)
48          {
49              anim.SetBool("Running", true);
50              sprite.flipX = true; // Hahmoa käännetään
51          }
52          else
53          {
54              anim.SetBool("Running", false);
55          }
56      }
57  } // class
58
```

Kuva 38. Hahmon kuva kääntyy siihen suuntaan mihin ollaan menossa

Tässä kohtaa voidaan taas parantaa koodin luettavuutta liittyen pelihahmon liikkumisnopeuteen ja hyppyvoimaan. Luodaan kaksi uutta yksityistä muuttujaa koodin alkuun, joiden arvoiksi annetaan aikaisemmin määritellyt halutut hyppy- ja juoksuvoiman arvot (kuva 39).

```
        Unity Script (1 asset reference) | 0 references
6   public class PlayerMovement : MonoBehaviour
7   {
8       private Rigidbody2D rb; // Määritellään yksityinen variaabeli
9       private SpriteRenderer sprite;
10      private Animator anim; // Määritellään yksityinen variaabeli
11
12      private float dirX = 0f;
13      [SerializeField] private float moveSpeed = 7f;
14      [SerializeField] private float jumpForce = 14f;
15
```

Kuva 39. Hahmon liikkeen ja hyppäämisen nopeusarvot, joita voidaan nyt myös editoida Unityn puolella

Tämän jälkeen poistetaan aikaisemmasta koodista tarkat määritellyt arvot ja korvataan ne äsken luoduilla muuttujilla. Lisäämällä komennon *[SerializedField]* muuttujan alkuun saadaan nämä arvot näkyviin Unityn puolella, missä niitä voi myös helposti muokata *Inspector*-ikkunassa. Tämä ei ole tarpeellista, mutta voi auttaa paljon, kun halutaan määritellä tarkemmin nopeutta pelihahmon juoksulle tai kuinka korkealle pelihahmo hyppää, koska muutokset näkyvät välittömästi Game-ruudulla projektia ajettaessa. (Multiple Animations 2022.)

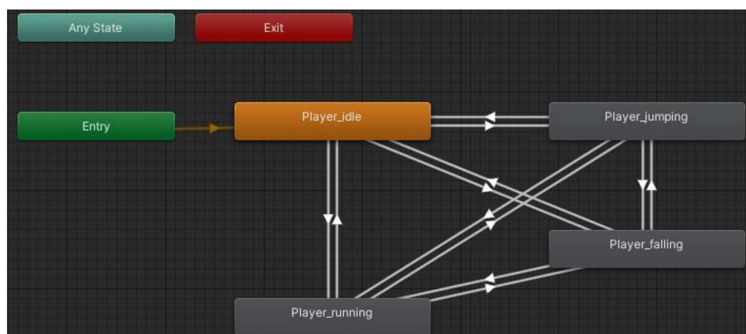
Seuraavaksi luodaan pelihahmolle hyppy- ja laskeutumisanimaatiot, koska tällä hetkellä pelihahmon hypätessä se jatkaa juoksemista. **PlayerCharacter**-kansiossa klikataan hiiren oikealla **Create** -> **Animation**, joka nimetään

Player_jumping. Tämä raahataan **Player**-objektiin, jotta saadaan se lisättyä *Animation*- ja *Animator*-ikkunoihin.

Avataan *Animation*-ikkuna, jossa valitaan **Player_jumping**-animaatio. Aikaisemmasta poiketen **Player_jumping**-animaatio tulee olemaan vain yksi kuva, joka ajetaan, kun hahmo hyppää ja joka vaihtuu **Player_falling**-animaatioon (tehdään seuraavaksi). Raahataan yksi **Player_jump**-spritekuvan yksittäisistä kuvista **Player_jumping**-animaatioon. Yhden kuvan takia ei tarvitse muuttaa *Sample Ratio* -arvoa. **Player_jumping**-animaation *Inspector*-ikkunassa klikataan ruksi *Loop Time* -asetukseen, jotta pelihahmon animaatio ei muutu ilmalennon aikana ennen kuin se alkaa laskeutumaan.

Tehdään sama uudestaan: luodaan uusi animaatio nimeltä **Player_falling**, lisätään se **Player**-objektiin raahaamalla ja sitten *Animation*-ikkunassa raahataan toinen **Player_jump**-spritekuvan kuvista (mitä ei ole vielä käytety) *Animation*-ikkunaan. *Sample Ratio* -arvoa ei tarvitse muuttaa. **Player_falling**-animaation *Inspector*-ikkunassa klikataan ruksi *Loop Time* -asetukseen, jotta pelihahmon animaatio ei muutu laskeutumisen aikana ennen kuin se on takaisin maan pinnalla.

Seuraavaksi pitää asettaa animaatioiden siirtymät oikein *Animator*-ikkunassa. Luodaan seuraavat siirtymät, joista jokaisessa siirtymässä siirrytään toiseen siirtymään (kuva 40).



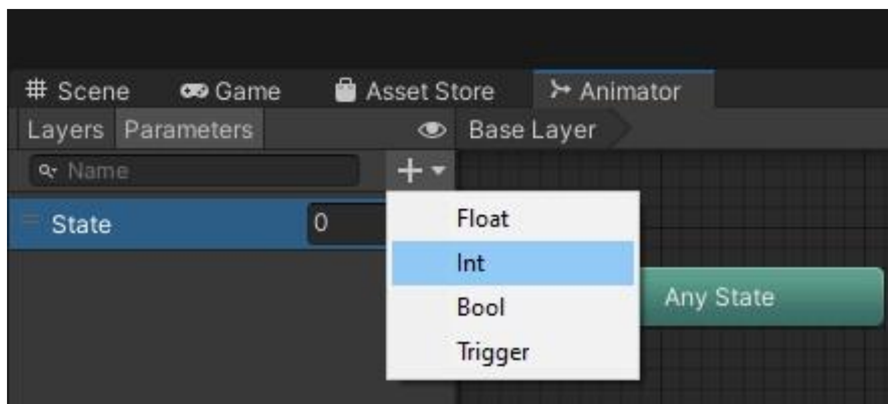
Kuva 40. Pelihahmon kaikki animaatiot, johon on tehty tarvittavat siirtymät (Multiple Animations 2022)

Jotta siirtymät toimisivat, pitää ne koodata oikein. Lisätään **PlayerMovement**-skriptin alkuun kaksi uutta muuttujaa, jotka määrittävät animaatioille parametrimuuttujan arvot (kuva 41).

```
2 references
16    private enum MovementState {idle, running, jumping, falling }
17    private MovementState state = MovementState.idle;
18
19    // Start is called before the first frame update
      Unity Message | 0 references
20    private void Start()
```

Kuva 41. Määritetään animaatioille parametri muuttujan arvot

Tämän jälkeen voidaan Unityn *Animator*-ikkunassa *Parameters*-välilehdeltä poistaa **Running**-parametri. Sen tilalle luodaan uusi Interger-parametri nimeltä **State** (kuva 42).

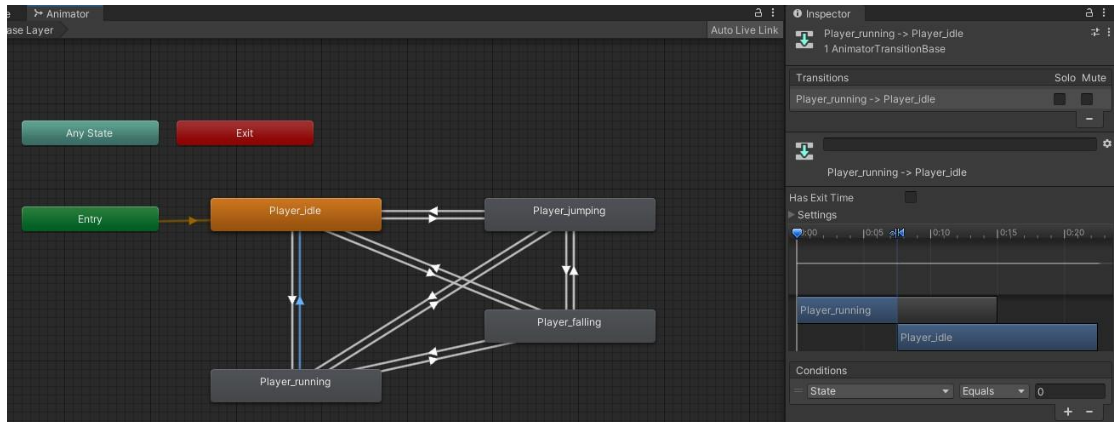


Kuva 42. Poistetaan aikaisempi boolean-parametri **Running** ja lisätään tilalle interger-parametri **State**

Tämän jälkeen joudutaan käymään lävitse kaikki siirtymät yksi kerrallaan ja määrittämään niiden asetukset ja ehdot oikein. Tässä kohtaa on hyvä olla tarkkana ja tarkistaa toistamiseen kaikki siirtymät, mutta testatessa kyllä huomaa helposti, jos virheitä on sattunut.

Aloitetaan kaikista ehdoista, jotka menevät **Player_idle**-animaatioon:

- **Player_running** -> **Player_idle** -siirtymässä haetaan *Conditions*-asetuksessa ehdoksi **State**-parametri, joka on *Equals* ja 0.
- **Player_jumping** -> **Player_idle** -siirtymässä klikataan ruksi pois kohdasta *Has Exit Time* ja asetetaan *Transition Duration (s) 0*. Tämän jälkeen *Conditions*-asetuksessa haetaan ehdoksi **State**, joka on *Equals* ja 0.
- **Player_falling** -> **Player_idle** -siirtymässä klikataan ruksi pois kohdasta *Has Exit Time* ja asetetaan *Transition Duration (s)* 0. Tämän jälkeen *Conditions*-asetuksessa haetaan ehdoksi **State**, joka on *Equals* ja 0.



Kuva 43. Siirtymien läpikäynti yksi kerrallaan (Multiple Animations 2022)

Seuraavaksi käydään lävitse kaikki siirtymät, jotka menevät **Player_running**-animaatioon:

- **Player_idle** -> **Player_running** -siirtymässä haetaan *Conditions*-asetuksessa ehdoksi *State*-parametri, joka on *Equals* ja 1.
- **Player_falling** -> **Player_running** -siirtymässä klikataan ruksi pois kohdasta *Has Exit Time* ja asetetaan *Transition Duration (s)* 0. Tämän jälkeen *Conditions*-asetuksessa haetaan ehdoksi **State**, joka on *Equals* ja 1.
- **Player_jumping** -> **Player_running** -siirtymässä klikataan ruksi pois kohdasta *Has Exit Time* ja asetetaan *Transition Duration (s)* 0. Tämän jälkeen *Conditions*-asetuksessa haetaan ehdoksi **State**, joka on *Equals* ja 1.

Seuraavaksi käydään lävitse kaikki siirtymät, jotka menevät **Player_jumping**-animaatioon:

- **Player_idle** -> **Player_jumping** -siirtymässä klikataan ruksi pois kohdasta *Has Exit Time* ja asetetaan *Transition Duration (s)* 0. Tämän jälkeen *Conditions*-asetuksessa haetaan ehdoksi **State**, joka on *Equals* ja 2.
- **Player_running** -> **Player_jumping** -siirtymässä klikataan ruksi pois kohdasta *Has Exit Time* ja asetetaan *Transition Duration (s)* 0. Tämän jälkeen *Conditions*-asetuksessa haetaan ehdoksi **State**, joka on *Equals* ja 2.
- **Player_falling** -> **Player_jumping** -siirtymässä klikataan ruksi pois kohdasta *Has Exit Time* ja asetetaan *Transition Duration (s)* 0. Tämän jälkeen *Conditions*-asetuksessa haetaan ehdoksi **State**, joka on *Equals* ja 2.

Seuraavaksi käydään lävitse kaikki siirtymät, jotka menevät **Player_falling**-animaatioon:

- **Player_jumping** -> **Player_falling** -siirtymässä klikataan ruksi pois kohdasta *Has Exit Time* ja asetetaan *Transition Duration (s)* 0. Tämän

jälkeen *Conditions*-asetuksessa haetaan ehdoksi **State**, joka on *Equals* ja 3.

- **Player_running** -> **Player_falling** -siirtymässä klikataan ruksi pois kohdasta *Has Exit Time* ja asetetaan *Transition Duration (s)* 0. Tämän jälkeen *Conditions*-asetuksessa haetaan ehdoksi **State**, joka on *Equals* ja 3.
- **Player_idle** -> **Player_falling** -siirtymässä klikataan ruksi pois kohdasta *Has Exit Time* ja asetetaan *Transition Duration (s)* 0. Tämän jälkeen *Conditions*-asetuksessa haetaan ehdoksi **State**, joka on *Equals* ja 3.

Kun siirtymien ehdot on asetettu oikein, palataan koodin. Poistetaan koodirivi *private MovementState state = MovementState.idle;* skriptin alusta. *UpdateAnimationState()*-metodin sisällä lisätään alkuun *MovementState State;* ja loppuun *anim.SetInterger("State, (int)State);*. Tämän jälkeen muokataan aikaisemmin kirjoitettua koodia ja lisätään koodipätkä (kuva 44), joka siirtää pelihahmon oikeaan animaatioon hypätessä tai laskeutuessa.

```
   1 reference
43 private void UpdateAnimationState()
44 {
45     MovementState State;
46
47     // Asetaan arvo, jolla tiedetään juostaanko vai ei
48     if (dirX > 0f)
49     {
50         State = MovementState.running;
51         sprite.flipX = false; // Hahmoa käännetään
52     }
53     else if (dirX < 0f)
54     {
55         State = MovementState.running;
56         sprite.flipX = true; // Hahmoa käännetään
57     }
58     else
59     {
60         State = MovementState.idle;
61     }
62
63     if (rb.velocity.y > .1f) // Hahmo siirtyy hyppy- tai laskeutumisanimaatioon
64     {
65         State = MovementState.jumping;
66     }
67     else if (rb.velocity.y < -.1f)
68     {
69         State = MovementState.falling;
70     }
71
72     anim.SetInteger("State", (int)State);
73
74 } // UpdateAnimationState
75 } // class
```

Kuva 44. Koodi, joka siirtää pelihahmon eri animaatioon, kun sen **State**-parametri muuttuu

Unityssä testatessa pelihahmo siirtyy nyt oikein eri animaatioiden välillä juostessa ja hypätessä. Tällä hetkellä pelihahmo pystyy hyppäämään loputtomasti **Player**-objektin ollessa ilmassa, joten korjataan se seuraavaksi. Halutaan, että pelihahmo pystyy hyppäämään vain kerran.

Lisätään **PlayerMovement**-skriptin alkuun uudet yksityiset muuttujat *private BoxCollider2D coll;*, jota sitten kutsutaan ohjelman alussa *coll = GetComponent<BoxCollider2D>();*, sekä [*SerializeField] private LayerMask jumpableGround;* (kuva 45).

```
8       private Rigidbody2D rb; // Määritellään yksityinen variaabeli
9       private BoxCollider2D coll;
10      private SpriteRenderer sprite;
11      private Animator anim; // Määritellään yksityinen variaabeli
12
13      [SerializeField] private LayerMask jumpableGround;
14
15      private float dirX = 0f;
16      [SerializeField] private float moveSpeed = 7f;
17      [SerializeField] private float jumpForce = 14f;
18
        6 references
19      private enum MovementState {idle, running, jumping, falling }
20
21      // Start is called before the first frame update
        Unity Message | 0 references
22      private void Start()
23      {
24          rb = GetComponent<Rigidbody2D>(); // Kutsutaan variaabelilla G
25          coll = GetComponent<BoxCollider2D>();
26          sprite = GetComponent<SpriteRenderer>();
27          anim = GetComponent<Animator>(); // Kutsutaan variaabelilla Ge
```

Kuva 45. Lisätään uusi yksityinen muuttuja, jota kutsutaan ohjelman alussa

Tämän jälkeen lisätään uusi koodipätkä, jolla halutaan estää kaksoishyppääminen ilmassa (kuva 46). Lähteenä käytetyssä videotutoriaalissa on tähän näppärä keino (ks. Grounding Check Using Boxcast 2022.)

```
        0 references
80      private bool IsGrounded() // Estetään kaksoishyppääminen ilmassa
81      {
82          return Physics2D.BoxCast(coll.bounds.center, coll.bounds.size, 0f, Vector2.down, .1f, jumpableGround);
83      }
84  } // class
85
```

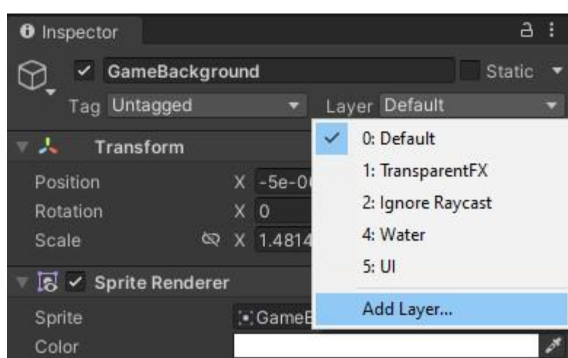
Kuva 46. Koodipätkä, jolla estetään kaksoishyppääminen

Uuden koodin jälkeen muutetaan aikaisempaa koodia, jossa määriteltiin hyppääminen lisäämällä sen perään *&& IsGrounded()* (kuva 47).

```
38          // Hahmo hyppää painamalla spacebar
39          if (Input.GetButtonDown("Jump") && IsGrounded())
40          {
41              rb.velocity = new Vector2(rb.velocity.x, jumpForce);
42          } // if
43
44          UpdateAnimationState();
45      } // Update
```

Kuva 47. Hahmo voi hypätä vain tasolta, joka on asetettu, että siitä voi hypätä

Koodin pitää lisäksi käydä Unityssä lisäämässä **GameBackground**-objektiin ja **Platform**-objektiin uusi kerros. Tämä tapahtuu klikkaamalla ensiksi **GameBackground**-objektia, minkä jälkeen *Inspector*-ikkunassa *Layer*-valikosta klikataan *Add Layer…* -vaihtoehtoa (kuva 48). Tämä avaa uuden ikkunan kaikista kerroksista, jossa voidaan valita uusi kerros. Valitaan *User Layer 6*, joka nimetään **Ground**. Tämän jälkeen *Hierarchy*-ikkunassa klikataan uudestaan **GameBackground**-objektia, minkä jälkeen *Inspector*-ikkunassa käydään vaihtamassa **GameBackground**-objektin **Layer** -> **6: Ground** (sama tehdään myös **Platform**-objektille!).



Kuva 48. Luodaan uusi kerros

Tämän lisäksi pitää **Player**-objektin *Inspector*-ikkunassa käydä valitsemassa **PlayerMovement**-skriptin alta *Jumpable Ground* aikaisemmin luomamme **Ground**-kerros (kuva 49).

Kuva 49. Asetetaan, että voidaan hypätä vain maasta, johon on asetettu **Ground** -layer

Tämän jälkeen pelihahmo ei pysty enää hyppäämään toista kertaa ollessaan ilmassa.

4.5 Pistehallinta

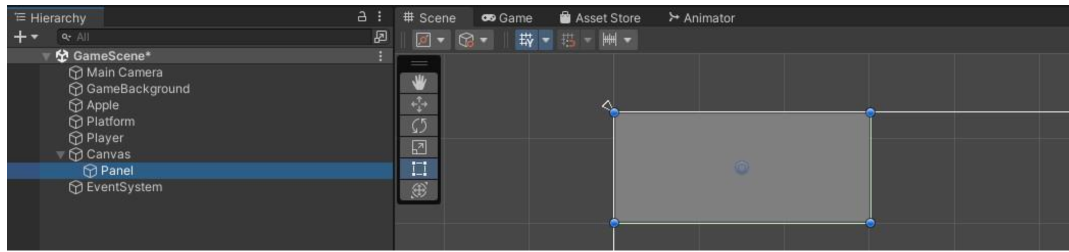
Luodaan **Codes**-kansioon uusi skripti nimeltä **AppleCode** ja avataan se Visual Studiossa. Kirjoitetaan kuvassa 50 oleva koodi, jolla pelihahmon osuessa **Apple**-objektiin se katoaa.

```
19        // Reagoidaan triggeriin
          Unity Message | 0 references
20        private void OnTriggerEnter2D(Collider2D collision)
21        {
22            // Oliko törmääjä Player
23            if (collision.name.Equals("Player"))
24            {
25                Debug.Log("OSUI!");
26                Destroy(this.gameObject);
27            } // if
28        } // OnTriggerEnter2D
29    } // class
30
```

Kuva 50. Omena katoaa, kun pelaaja osuu siihen

Tämän jälkeen halutaan luoda pistelaskuri, joka laskee, kuinka paljon aikaa pelaajalla on, ja kuinka monta omenaa pelaaja on kerännyt. Tätä varten luodaan **GameScene**-skenen päälle canvas-elementti, joka näyttää ajan ja kerättyjen omenien määrän. Tämä tapahtuu painamalla plusmerkkiä *Hierarchy*-ikkunan vasemmassa yläkulmassa, joka avaa valikon, josta valitaan **UI** -> **Canvas**. Tämä luo *Hierarchy*-ikkunaan uuden objektin nimeltä **Canvas** (annetaan sen olla tämän niminen). Nähdäkseen **Canvas**-objektin kunnolla täytyy käyttäjän hiiren keskipainikkeella rullata tarpeeksi taaksepäin, että näkymä pienenee niin paljon, että koko **Canvas**-objektin alue näkyy. Syy, minkä takia Unity tekee tästä alueesta niin ison, on se, että pystytään helpommin muokkaamaan UI-elementtiä ilman, että se sotkee tai menee päällekkäin sen hetkisen skenen kanssa.

Tämän jälkeen hiiren oikealla klikataan **Canvas**-objektin päällä ja valitaan **UI** -> **Panel**, joka luo uuden objektin nimeltä **Panel Canvas**-objektin alle. Tämä luo harmaan laatikon, joka peittää koko **Canvas**-objektin alueen. Pienennetään tätä aluetta sopivamman kokoiseksi (kuva 51). *Game*-näkymässä kokeillessa näkyy nyt vasemmassa yläkulmassa harmaan läpikuultava laatikko, joka pysyy näkymän päällä.



Kuva 51. **Canvas**-objektiin liitetyn **Panel**-objektin alueen säätäminen

Klikataan **Panel**-objektia *Hierarchy*-ikkunassa, joka avaa sen *Inspector*-ikkunan. *Image*-komponentin asetuksista käydään muuttamassa harmaa alue paremmin nähtäväksi, mikä tapahtuu painamalla *Color*-asetuksessa värialuetta, joka avaa *Color*-ikkunan. Kokeilemalla pystyy löytämään, mitkä värit sopivat parhaiten, mutta tässä tapauksessa käytetään Hexadecimal-värikoodia 1E438C, joka tekee **Panel**-objektin alueesta sinertävän ja joka näkyy selkeästi ruudulla. Säädetään vielä *A*-asetuksesta laatikon läpikuultavuuden arvoksi 200.

Seuraavaksi ankkuroidaan **Panel**-objekti UI-elementissä haluttuun kohtaan, joka on vasen yläkulma. Tämä tapahtuu klikkaamalla *Inspector*-ikkunassa *Rect Transform* -komponentissa laatikkoa, joka avaa kaikki mahdolliset ankkurointipisteet valittavaksi. Valitaan *top left* -vaihtoehto, joka ankkuroi **Panel**-objektin UI-elementin vasempaan yläkulmaan.

Tällä hetkellä **Panel**-objekti pysyy tietyn kokoisena huolimatta siitä, kuinka paljon sitä yrittäisi muuttaa transform-työkalulla. Tämä saadaan muutettua klikkaamalla *Hierarchy*-ikkunasta **Canvas**-objektia ja sen *Inspector*-ikkunassa on *Canvas Scaler* -komponentti, jonka *UI Scale Mode* -asetuksesta vaihdetaan lukemaan *Scale With Screen Size*. Tämä luo sen, että **Panel**-objektin koko korreloi aina oikeassa suhteessa *Game*-näkymässä. Määritetään *Inspector*-ikkunassa **Panel**-objektin leveydeksi 210 ja korkeudeksi 52.

Seuraavaksi luodaan tekstikentät, jotka näyttävät pelaajalle jäljellä olevan peliajan ja kerättyjen omenien määrän. Tämä tapahtuu klikkaamalla *Hierarchy*-ikkunassa **Panel**-objektin päällä hiiren oikealla ja valitsemalla **UI** -> **Legacy** -> **Text**. Tämä luo uuden tekstiobjektin, joka nimetään **TextTime**. **TextTime**-objektin *Inspector*-ikkunassa käydään kirjoittamassa *Text*-komponentin tekstikenttään **TIME:** ja fontiksi annetaan olla *Arial*, *Font Style* on *Bold* ja *Font Size*

on 20. Väriksi valitaan keltainen, johon käytetään Hexadecimal-arvoa DDD73E.

Tämän jälkeen **Panel**-objektin alle luodaan toinen tekstiobjekti samalla lailla, joka nimetään **TextApples**. Sen tekstikenttään kirjoitetaan **APPLES:** ja käytetään samoja visuaalisia elementtejä kuin edellisen tekstiobjektin kanssa. Tämän jälkeen asetellaan **TextTime**- ja **TextApples**-objektit siististi allekkain (kuva 52) **Panel**-objektin alueen sisällä. Nämä voidaan *Inspector*-ikkunassa kumpikin ankkuroida vasempaan yläkulmaan, kuten tehtiin **Panel**-objektin kanssa.

Kuva 52. Tekstit aseteltu hyvin **Panel**-objektissa

Kun UI-elementti on valmis, voidaan luoda toiminallisuus, jolla **Panel**-objektissa näkyy alaspäin juokseva aika ja kerättyjen omenien määrä.

Luodaan **Codes**-kansiossa uusi skripti nimeltä **ScoreCounter** ja avataan se Visual Studiossa. Aloitetaan kirjoittamaan koodia (kuva 53), joka luo aikalaskurin, joka laskee, kuinka paljon aikaa pelaajalla on, ennen kuin peli päättyy.

```
public class ScoreCounter : MonoBehaviour
{
    // Tässä ylläpidetään ajan määrää
    public float aikalaskuri = 150f; // Luodaan julkinen variaabeli, aika alkaa arvosta

    // Tähän tallennetaan kerätyt omenat
    public float omenat = 0f; // Luodaan julkinen variaabeli, kerättyjä omenia on aluss

    // Tähän haetaa TextTime
    private GameObject t1 = null;

    // Start is called before the first frame update
    void Start()
    {
        this.t1 = GameObject.Find("TextTime"); // Haetaan TextTime-objekti
    } // Start

    // Update is called once per frame
    void Update()
    {
        // Vähennetään aikalaskuria
        this.aikalaskuri -= Time.deltaTime * 10; // Vähentää aikaa
        this.t1.GetComponent<Text>().text = "TIME: " + this.aikalaskuri.ToString("0");

    } // Update
} // class
```

Kuva 53. Laskuri, joka laskee, kuinka paljon pelaajalla on aikaa

Koodin alkuun pitää lisätä rivi *using UnityEngine.UI;*, jotta UI-elementit näkyvät. Tämän jälkeen Unityn puolella käydään *Hierarchy*-ikkunassa luomassa uusi tyhjä objekti (klikataan hiiren oikealla ja valitaan *Create Empty*), joka nimetään **Storage**. Tähän objektiin voidaan säilöä **GameScene**-skenessä tarvittavia koodeja (vaihtoehtoisesti koodin voi säilöä **Main Camera** -objektiin). Raahataan siihen **ScoreCounter**-skripti. Kokeillaan *Game*-näkymässä, kuinka aikalaskuri toimii, ja nyt nähdään, kuinka **Panel**-objektin sisälle on tulostettu ajan arvo, joka alkaa vähenemään, kun peli on käynnissä. Tässä kohtaa on hyvä tarkistaa, että **Panel**-objekti on tarpeeksi leveä sekä korkea, että tulostettu lukema näkyy siinä, koska muuten arvo saattaa kadota näkymättömiin!

Seuraavaksi tehdään koodi, joka laskee omenien määrän ja tulostaa sen **Panel**-objektiin. Tämä tapahtuu seuraavalla koodilla kuvassa 54.

```
19      // Tähän haetaa TextApples
20      private GameObject t2 = null;
21
22      // Start is called before the first frame update
        Unity Message | 0 references
23      void Start()
24      {
25          this.t1 = GameObject.Find("TextTime"); // Haetaan TextTime-objekti
26          this.t2 = GameObject.Find("TextApples"); // Haetaan TextApples-objekti
27      } // Start
28
29      // Update is called once per frame
        Unity Message | 0 references
30      void Update()
31      {
32          // Vähennetään aikalaskuria
33          this.aikalaskuri -= Time.deltaTime * 10; // Vähentää aikaa
34          this.t1.GetComponent<Text>().text = "TIME: " + this.aikalaskuri.ToString("0");
35
36          this.t2.GetComponent<Text>().text = "APPLES: " + this.omenat.ToString("0"); //
37
38      } // Update
39  } // class
40
```

Kuva 54. Koodi, joka laskee kerättyjen omien määrän ja tulostaa sen pelinäkymään

Tämän jälkeen täytyy käydä **AppleCode**-skriptissä tekemässä pieni muutos, että koodit tunnistavat toisensa. Lisätään **AppleCode**-skriptissä rivi, joka hakee Unityssä **Storage**-objektin sisältä **ScoreCounter**-skriptin ja sen sisältä omenat-muuttujan arvon, jolla kasvatetaan **Panel**-objektissa olevaa lukemaa yhdellä, kun **Player**-objekti osuu **Apples**-objektiin.

```
Unity Script (1 asset reference) | 0 references
5   public class AppleCode : MonoBehaviour
6   {
7       // Tähän haetaan koodivarasto
8       private GameObject koodit = null;
9
10      // Start is called before the first frame update
        Unity Message | 0 references
11      void Start()
12      {
13          // Haetaan valmiiksi
14          this.koodit = GameObject.Find("Storage");
15
16      } // Start
17
18      // Update is called once per frame
        Unity Message | 0 references
19      void Update()
20      {
21
22      } // Update
23
24      // Reagoidaan triggeriin
        Unity Message | 0 references
25      private void OnTriggerEnter2D(Collider2D collision)
26      {
27          // Oliko törmääjä Player
28          if (collision.name.Equals("Player"))
29          {
30              Debug.Log("OSUI!");
31
32              // Päivitetään omenien määrää Panel-objektissa
33              this.koodit.GetComponent<ScoreCounter>().omenat += 1f;
34
35              Destroy(this.gameObject);
36          } // if
37      } // OnTriggerEnter2D
38  } // class
39
```

Kuva 55. Pannaan koodit tunnistamaan toisensa

Unityn puolella pelidemoa testatessa nähdään UI-elementissä, että aika tulostuu siihen ja se vähenee sekä kerätyt omenat tulostuvat siihen ja niiden määrä kasvaa.

4.6 Siirtymä skenejen välillä

Aikaisemmin on jo luotu valmiiksi pelidemon eri skenet ja nyt niille tehdään siirtymät. Halutaan, että:

- **StartScene**-skenestä siirrytään **GameScene**-skeneen
- **GameScene**-skenestä siirrytään **DefeatScene**-skeneen tai **WinScene**-skeneen
- **DefeatScene**- tai **WinScene**-skeneistä siirrytään takaisin **StartScene**-skeneen.

Tämä tapahtuu kirjoittamalla koodin, jossa määritetään, että **StartScene**-, **DefeatScene**- ja **WinScene**-skeneissä siirrytään haluttuun skeneen, kun painetaan siirtymän mahdollistavaa nappia. **GameScene**-skenessä siirrytään **DefeatScene**-skeneen, kun **Panel**-objektiin tulostettu ajan arvo saavuttaa luvun 0. **WinScene**-skeneen siirrytään, kun **Panel**-objektiin tulostettu omenien määrä saavuttaa luvun 10.

Scenes-kansiosta valitaan **StartScene** ja siirrytään siihen, minkä jälkeen luodaan **Codes**-kansioon uusi skripti nimeltä **StartCode** ja avataan se Visual

Studiossa. Ensimmäiseksi koodissa pitää linkittää mukaan *Scene Manager*-olio, joka tapahtuu kirjoittamalla skriptin alkuun *using UnityEngine.SceneManagement;*. Tämä hallitsee eri skenejä Unityssä. Tämän jälkeen kirjoitetaan yksinkertainen koodi (kuva 56), joka hoitaa siirtymän **StartScene**-skenestä **GameScene**-skeneen.

```
5   // Scenemanager
6   using UnityEngine.SceneManagement;
7
    Unity Script | 0 references
8   public class StartCode : MonoBehaviour
9   {
10      // Start is called before the first frame update
        Unity Message | 0 references
11      void Start()
12      {
13
14      } // Start
15
16      // Update is called once per frame
        Unity Message | 0 references
17      void Update()
18      {
19          // Kun pelaaja painaa Enter-nappia niin siirry
20          if (Input.GetKeyDown(KeyCode.Return))
21          {
22              SceneManager.LoadScene("GameScene");
23          } // if
24
25      } // Update
26  } // class
27
```

Kuva 56. Siirrytään seuraavaan skeneen, kun tietty ehto täyttyy

Unityssä raahataan **StartCode**-skripti **Main Camera** -objektiin. Tämän jälkeen nykyinen skene käydään lisäämässä *Build Settings* -asetuksiin. Tämä tapahtuu klikkaamalla Unityn vasemmasta yläkulmasta **File** -> **Build Settings**, mikä avaa *Build Settings* -ikkunan, jossa skenejä voidaan lisätä ja niiden järjestystä voidaan vaihtaa. Seuraavissa skeneissä tehdään sama lisäys lopuksi.

Scenes-kansiosta valitaan **DefeatScene** ja siirrytään siihen, minkä jälkeen luodaan **Codes**-kansioon uusi skripti nimeltä **DefeatCode** ja avataan se Visual Studiossa. Ensimmäiseksi koodissa pitää linkittää mukaan *Scene Manager* -olio, joka tapahtuu kirjoittamalla skriptin alkuun using *UnityEngine.SceneManagement;*. Tämä hallitsee eri skenejä Unityssä. Tämän jälkeen kirjoitetaan yksinkertainen koodi, joka hoitaa siirtymän **DefeatScene**-skenestä **StartScene**-skeneen (koodi on sama kuin kuvassa 56, mutta tällä kertaa siirrytään **StartScene**-skeneen).

Scenes-kansiosta valitaan **WinScene** ja siirrytään siihen, minkä jälkeen luodaan **Codes**-kansioon uusi skripti nimeltä **WinCode** ja avataan se Visual Studiossa. Ensimmäiseksi koodissa pitää linkittää mukaan *Scene Manager* -olio,

joka tapahtuu kirjoittamalla skriptin alkuun *using UnityEngine.SceneManagement;*. Tämä hallitsee eri skenejä Unityssä. Tämän jälkeen kirjoitetaan yksinkertainen koodi, joka hoitaa siirtymän **WinScene**-skenestä **StartScene**-skeneen (koodi on sama kuin kuvassa 56, mutta tällä kertaa siirrytään **StartScene**-skeneen).

Scenes-kansiosta valitaan **GameScene** ja siirrytään siihen, minkä jälkeen avataan **Codes**-kansiosta **ScoreCounter**-skripti Visual Studiossa. Linkitetään tämän skriptin alkuun *Scene Manager* -olio, joka tapahtuu kirjoittamalla skriptin alkuun *using UnityEngine.SceneManagement;*.

Tämän jälkeen lisätään skriptin loppuun seuraavat koodipätkät (kuva 57), joilla peli päättyy, jos aikalaskurin arvo on 0, tai pelin voittaa, jos omenien määrä on 10.

```
40
41          // Jos aika loppuu niin peli on ohi
42          if (this.aikalaskuri < 0f)
43          {
44              SceneManager.LoadScene("DefeatScene");
45          } // if
46
47          //Jos omenia on 10, voittaa pelin
48          if (this.omenat >= 10f)
49          {
50              SceneManager.LoadScene("WinScene");
51          }
52      } // Update
53  } // class
54
```

Kuva 57. Koodi, joilla peli päättyy joko voittoon tai tappioon

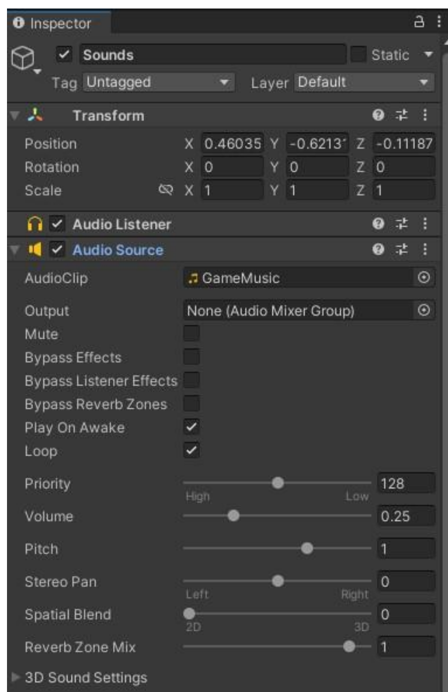
Unityssä kokeillessa tarkistetaan, että siirtymät toimivat halutulla tavalla ja siirtyvät seuraavaan skeneen, kun Enter-näppäintä painetaan.

4.7 Pelin äänimaailma

Pelidemon äänimaailma aloitetaan luomalla pelille taustamusiikki, joka kuuluu **StartScene**- ja **GameScene**-skeneissä. Tätä varten luodaan *Hierarchy*-ikkunaan uusi tyhjä objekti, joka nimetään **Sounds**. Seuraavaksi klikataan **Main Camera** -objektia, josta käydään klikkaamassa *Inspector*-ikkunassa ruksi pois *Audio Listener* -komponentista (Unity lisää tämän komponentin automaattisesti **Main Camera** -objektiin, kun luodaan uusi skene). Tämän jälkeen

Sounds-objektiin lisätään Unityn yläkulmasta klikkaamalla **Component** -> **Audio** -> **Audio Listener** ja **Audio Source**.

Tämän jälkeen klikataan **Sounds**-objekti valituksi *Hierarchy*-ikkunasta, jossa **Sounds**-kansiosta valitaan **GameMusic**-äänitiedosto, joka raahataan **Sounds**-objektin *Inspector*-ikkunassa *Audio Source* -komponentin kohtaan *AudioClip*. Sitten voidaan *Audio Source* -komponentissa säätää taustamusiikin erilaisia asetuksia (kuva 58). Valitaan, että musiikki pyörii jatkuvasti klikkaamalla raksi kohdassa *Loop*. Tämän jälkeen voidaan säätää taustamusiikin voimakkuutta, asetetaan taustamusiikin arvo 0.25 (tämän enempää musiikin säätelyyn ei kosketa).



Kuva 58. *Audio Listener*- ja *Audio Source* -komponentit lisätty

Taustamusiikin jälkeen lisätään ääni, joka kuuluu, kun pelaaja kerää omenan. **Sounds**-objektiin lisätään toinen *Audio Source* -komponentti, jonka *Inspector*-ikkunassa *AudioClip*-kohtaan lisätään **AppleCollect**-äänitiedosto. Klikataan ruksi pois *Play On Awake* -asetuksesta, jotta tätä äänitiedostoa ei lähdetä soittamaan, kun näkymä käynnistyy.

Tämän jälkeen käydään lisäämässä **AppleCode**-skriptiin koodipätkä (kuva 59), joka pistää toistamaan **AppleCollect**-äänitiedostoa joka kerta, kun pelihahmo kerää omenan.

```
27          // Oliko törmääjä Player
28          if (collision.name.Equals("Player"))
29          {
30              Debug.Log("OSUI!");
31
32              // Soitetaan ääniefekti
33              GameObject.Find("Sounds").GetComponents<AudioSource>()[1].Play();
34
35              // Päivitetään omenien määrää Panel-objektissa
36              this.koodit.GetComponent<ScoreCounter>().omenat += 1f; //Haetaan St
37
38              Destroy(this.gameObject);
39          } // if
40      } // OnTriggerEnter2D
41  } // class
```

Kuva 59. Lisätään äänitiedosto, joka soi, kun pelaaja kerää omenan

Tämän jälkeen luodaan **DefeatScene**- ja **WinScene**-skeneissä tyhjä objekti nimeltä **Sounds**, joihin lisätään *Audio Listener*- ja *Audio Source* -komponentit (klikataan *Audio Listener* -komponentti pois päältä kummankin skenen **Main Camera** -objektissa). **DefeatScene**-skenessä raahataan **Sounds**-objektin *Audio Listener* -komponenttiin **Sounds**-kansiosta **GameOver**-äänitiedosto ja sama tehdään **WinScene**-skenessä **GameWin**-äänitiedostolle.

4.8 Viimeistely ja pelin kääntäminen

Viimeisenä asiana pelidemoon tehdään koodi, joka lopettaa pelin, kun painetaan Esc-näppäintä. **Codes**-kansioon tehdään uusi skripti nimeltä **EndCode**, johon kuvassa 60 kirjoitetaan lyhyt koodipätkä.

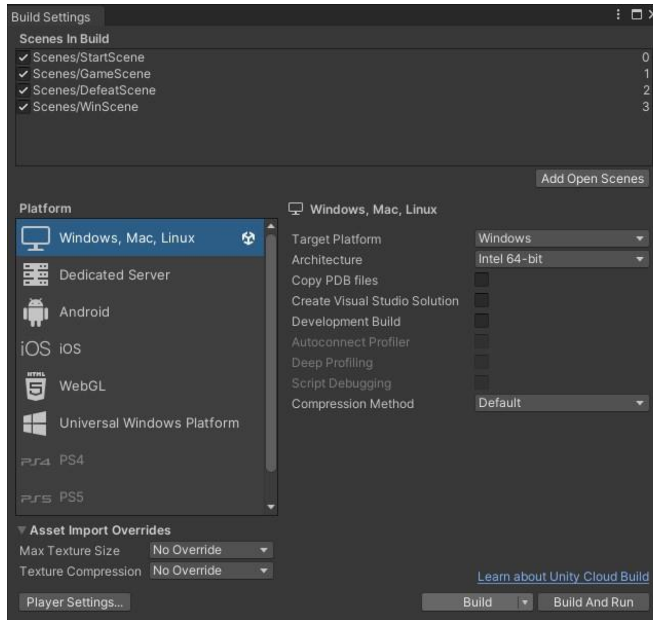
```
13      // Update is called once per frame
        Unity Message | 0 references
14      void Update()
15      {
16
17          // Esc-näppäimellä voidaan lopettaa sovellus
18          if (Input.GetKeyDown(KeyCode.Escape))
19          {
20              Application.Quit();
21          } // if
22
23      } // Update
24  } // class
25
```

Kuva 60. Ohjelma sulkeutuu, kun painetaan Esc-näppäintä

Tämä lisätään **StartScene**-, **DefeatScene**- ja **WinScene** -skeneissä **Main Camera** -objektiin ja **GameScene**-skenessä **Storage**-objektiin.

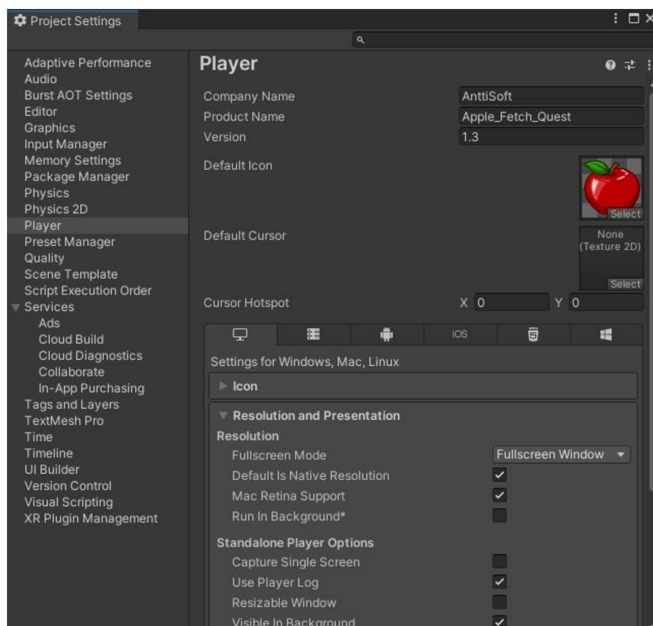
Kun peli on saatu valmiiksi, se pitää ”rakentaa”, että sitä pystyy pelaamaan Unityn ulkopuolella. Tämä tapahtuu valitsemalla Unityn vasemmasta yläkul-

masta **File** -> **Build Settings**, josta aukeaa ikkuna (kuva 61), missä rakentamisen asetuksia voidaan säätää ja peli voidaan rakentaa ulos Unitystä. Tarkistetaan, että kaikki halutut skenet on näkyvissä *Scenes In Build* -kentässä (**StartScene**, **GameScene**, **DefeatScene**, **WinScene**).



Kuva 61. *Build Settings* -ikkuna, jossa valitaan halutut skenet, säädetään viimeisiä asetuksia ja voidaan rakentaa peli

Tämän jälkeen voidaan painaa *Build Settings* -ikkunan vasemmasta alanurkasta *Player Settings* -nappia, joka avaa uuden ikkunan (kuva 62), missä erilaisia tietoja voi täyttää tai lisäasetuksia säätää.



Kuva 62. Projektin *Project Settings* -asetukset

Tämän jälkeen painetaan Build-nappia ja valitaan mihin kansioon peli rakennetaan. Tässä menee hetki, kun odotetaan, että Unity rakentaa ja tulostaa pelin ulos systeemistään. Kun tämä on valmis, voi pelin avata valitsemalla kansion, johon peli rakennettiin ja klikkaamalla pelin käynnistyskuvaketta, mikä aloittaa pelin.

5 PÄÄTÄNTÖ

Opinnäytetyön tekemisen aikana saatiin vastaus, että Unity soveltuu mainiosti 2D-pelin toteuttamiseen. Käyttöliittymä itsessään sisältää kaikki tarvittavat yhteensopivuudet ja avut tämän kaltaisen projektin tekemiseen.

Opinnäytetyön aikana toteutettu pelidemo on vain ripaus niistä toiminallisuuksista ja mekaniikoista, joita varsinaiseen kaupalliseen peliin voisi laittaa. Opinnäytetyön pelidemoa voisi jatkokehittää ja viimeistellä lisää parantamalla musiikin toimivuutta (musiikki ei alkaisi joka kerta uudestaan skenejen välillä siirtyessä vaan jatkuisi), lisäämällä eri kenttiä ja maisemia, erilaisia vastuksia tai vihollisia, mikä taas mahdollistaisi elämäpistesysteemin rakentamisen (peli loppuisi, kun pelaajan elämäpisteet putoavat nollaan. Nyt peli loppuu, kun pelin aika loppuu). Varsinaisen pelin voisi myös säätää asteittain vaikeutuvaksi sitä mukaa, mitä pidemmälle pelissä pääsisi.

Omia kokemuksia opinnäytetyön tekemisestä nousi tunteiden kirjon laidasta laitaan. Positiivisia kokemuksia oli onnistuminen ja sen näkeminen, kuinka pelidemo etenee ja alkaa pikkuhiljaa muodostua toimivaksi kokonaisuudeksi, mutta koin myös turhautumista siitä miten pikkutarkkoja jotkin asiat ovat. Pelidemon rakentamisen yhteydessä huomasin, kuinka tärkeää selkeä dokumentaatio ja nimeäminen sekä koodiskripteissä tapahtuva asioiden muistiinpaneminen on. Jotkin mekaniikat itsessään ovat yksinkertaisia ja selkeitä tehdä, mutta kun niistä halutaan yhteensopivia toiminnallisuuksia, jotka kommunikoivat keskenään, koodista voi tulla varsinainen viidakko, johon eksyy ja hukkuu, ellei ole tehnyt itselleen selkeitä viittauksia siihen, mikä vaikuttaa mihinkin.

Oikeassa projektissa olisi kuitenkin useita henkilöitä tekemässä kukin oman osaamisalueensa hommia, mikä helpottaisi huomattavasti tämänkaltaisen projektin suorittamista. Yhden ihmisen ei tarvitsi osata aivan kaikkea.

Unity ei ole ainut pelimoottori markkinoilla, jolla itsenäisesti oman peliprojektin pystyisi luomaan, mutta tämän opinnäytetyön perusteella se on sekä pätevä, yksinkertainen että ilmainen työkalu, jolla asiasta kiinnostuneet voivat kokeilla taitojaan ja ehkä aloittaa oman uransa pelikehityksessä. Tällä hetkellä Unity on eniten käytetty pelimoottori etenkin mobiiliympäristössä, johon 2D-pelejä tehdään (opinnäytetyön pelidemo soveltuisi mahdollisesti parhaiten jatkokehitettäväksi juuri mobiilipelinä).

LÄHTEET

Animation & Animator. 2022. Build a 2D Platformer Game in Unity. Videoleikesarja. Päivitetty 3.6.2021. Saatavissa: https://www.youtube.com/watch?v=GChUpPnOSkg [viitattu 30.9.2022].

Apple Cartoon Transparent PNG. 2022. Kuva. Saatavissa: https://www.subpng.com/png-kc94ab/ [viitattu 5.11.2022].

Apple_Crunch_17.wav – Eating Apple Crunches. 2022. Musiikkitiedosto. Saatavissa: https://freesound.org/people/Koops/sounds/20280/ [viitattu 1.11.2022].

Brown Wooden Board. 2022. Kuva. Saatavissa: https://www.pngwing.com/en/free-png-bxrcq [viitattu 5.11.2022].

Final Fantasy VI Kefka Laugh Sound Effect. 2022. Musiikkitiedosto. Saatavissa: http://soundfxcenter.com/download-sound/final-fantasy-vi-kefka-laugh-sound-effect/ [viitattu 1.11.2022].

Grounding Check Using Boxcast. 2022. Build a 2D Platformer Game in Unity. Videoleikesarja. Päivitetty 7.6.2021. Saatavissa: https://www.youtube.com/watch?v=LEUhxe9vUOM [viitattu 10.10.2022].

Knight Sprite Sheet (Free). 2022. Kuva. Saatavissa: https://assetstore.unity.com/packages/2d/characters/knight-sprite-sheet-free-93897 [viitattu 5.11.2022].

Multiple Animations. 2022. Build a 2D Platformer Game in Unity. Videoleikesarja. Päivitetty 5.6.2021. Saatavissa: https://www.youtube.com/watch?v=65E-q0JxYwU [viitattu 5.10.2022].

Pixel Art Forest. 2022. Kuva. Saatavissa: https://assetstore.unity.com/packages/2d/textures-materials/nature/pixel-art-forest-80909 [viitattu 5.11.2022].

Player Movement. 2022. Build a 2D Platformer Game in Unity. Videoleikesarja. Päivitetty 1.6.2021. Saatavissa: https://www.youtube.com/watch?v=Uv5tfMSKlnU [viitattu 6.10.2022].

Success Fanfare Trumpets. 2022. Musiikkitiedosto. Saatavissa: https://pixabay.com/sound-effects/success-fanfare-trumpets-6185/ [viitattu 1.11.2022]

The Low Whistle – Traditional Celtic Music. 2022. Musiikki. Saatavissa: https://pixabay.com/music/folk-the-low-whistle-traditional-celtic-music-1275/ [viitattu 1.11.2022].

Tilemap & Tile Palette. 2022. Build a 2D Platformer Game in Unity. Videoleikesarja. Päivitetty 30.5.2021. Saatavissa: https://www.youtube.com/watch?v=QkbGr1rAya8 [viitattu 28.9.2022].

Unity 2D. 2022. Unity Technologies. WWW-dokumentti. Saatavissa: https://docs.unity3d.com/Manual/Unity2D.html [viitattu 13.10.2022].

Why I Hate: Fetch Quests. 2022. BagoGames. WWW-dokumentti. Saatavissa: https://bagogames.com/why-i-hate-fetch-quests/ [viitattu 9.10.2022].