OULUN AMMATTIKORKEAKOULU

Juhani Kurki

Olli-Pekka Sutinen

2D-pelinkehitys Unity-pelimoottorilla

Graafisen työn ja ohjelmoinnin yhdistäminen pelin toteutuksessa

2D-pelinkehitys Unity-pelimoottorilla

Graafisen työn ja ohjelmoinnin yhdistäminen pelin toteutuksessa

Juhani Kurki, Olli-Pekka Sutinen
Opinnäytetyö
Kevät 2019
Tietojenkäsittelyn koulutusohjelma
Oulun ammattikorkeakoulu

TIIVISTELMÄ

Oulun ammattikorkeakoulu
Tietojenkäsittelyn koulutusohjelma, Internet-palvelut ja digitaalinen media

---

Tekijä(t): Juhani Kurki, Olli-Pekka Sutinen
Opinnäytetyön nimi: 2D-pelinkehitys Unity-pelimoottorilla
Työn ohjaaja: Matti Viitala
Työn valmistumislukukausi ja -vuosi: Kevät 2019 Sivumäärä: 52

Opinnäytetyön tarkoituksena oli suunnitella ja luoda 2D-tasohyppelypeli Unity-pelimoottorilla, yhteisen peli-idean pohjalta. Tavoitteena oli saada luotua toimiva demo ja prototyyppi valmiin peli-idean pohjalta, jota olisi mahdollista työstää myös jatkossa laajemmalle kehitykselle. Teorian, käytännön ja toteutuksen osuudet opinnäytetyössä ovat jaettu kahtia. Näissä toinen tekijä keskittyi ohjelmointiin ja toinen pelin graafisen ulkoasun toteuttamiseen.

Ohjelmointi ja pelin toiminnalliset osiot työssä toteutettiin käyttämällä C# -ohjelmointikieltä, joka on tällä hetkellä Unityn ainoa mahdollinen ohjelmointikieli. Tämän lisäksi kyseinen ohjelmointikieli oli helppo hallita ja omaksua oman henkilökohtaisen kokemuksen pohjalta. Pelin grafiikka tehtiin pääasiassa Adobe Photoshop -ohjelmalla ja animaatiot Adobe Animate -ohjelmalla. Hyödyntämällä ja yhdistämällä grafiikkaa ohjelmoinnin tukena, saatiin lopulta luotua aikaiseksi pelimekaniikoiltaan toimiva sekä onnistunut peli.

Emme pyri tällä raportilla opettamaan pelin tekoa lukijalle, vaan kuvaamaan prosessia, jonka kävimme läpi peliä tehdessä. Työ sisältää runsaasti Unity-pelimoottoriin, grafiikkaan ja ohjelmointiin liittyviä teknisiä termejä, joiden kääntäminen suomenkieliseen muotoon tuotti kyseisessä raportissa hieman haasteita tekijöille. Raportissa esiteltäviin aiheisiin kuuluu muun muassa pelinkehitystyökalujen laajamittainen tarkastelu, grafiikan, animaatioiden ja pelimaailman luominen sekä ohjelmoinnin hyödyntäminen ja käyttäminen pelin eri toiminnallisuuksien tekemisen perustana. Näissä aihealueissa käymme läpi myös tarkemmin, kuinka pelikehityksessä on tärkeää yhdistää grafiikkaa ohjelmakoodin kanssa esimerkiksi liikkumisen ja animaatioiden tuottamisen tukena. Työn tuloksia on mahdollista hyödyntää Unity-pelimoottorin edistyksellisempään tutkimiseen sekä grafiikan ja ohjelmoinnin laajamittaiseen tarkasteluun pelinkehityksen tukena.

Asiasanat: pelit, peliala, ohjelmointi, peliohjelmointi, peligrafiikka, pelisuunnittelu, pelihahmot

ABSTRACT

Oulun ammattikorkeakoulu
Tietojenkäsittelyn koulutusohjelma, Internet-palvelut ja digitaalinen media

---

Author(s): Juhani Kurki, Olli-Pekka Sutinen
Title of thesis: Making a 2D-game with Unity Engine
Supervisor(s): Matti Viitala
Term and year when the thesis was submitted: Kevät 2019 Number of pages: 52

The purpose of this thesis was to design and create a 2D-platformer game with the Unity game engine, based on a common game idea between the two creators. The goal was to create a working demo and a prototype based on a finished game idea, which could also be further developed in the future. The parts of theory, practice and implementation in the thesis were divided into two. In these, one creator focuses on programming and the other on the graphic appearance of the game.

Programming and the functionality of the game is implemented using the C# programming language. This programming language was chosen based on its popularity and personal experience. The game's graphics were mainly made with Adobe Photoshop and animations with Adobe Animate. Utilizing and combining graphics to support programming in the project, we succeeded in making a game with working mechanics.

With this report, we do not seek to teach the reader how to make a make with Unity engine, but to describe the process that we used while making our own game. The work contains a lot of technical terms related to Unity's gaming engine, graphics and programming. Topics covered in the thesis include a wide-ranging review of game development tools and softwares, the creation of graphics, animations and the gaming world as well as the use of programming as a basis for making different functionalities of the game. In these topics, we will also go into more detail about the importance of combining graphics with programming files, for example, to support movement and animation while making different mechanics for the game. It's possible to utilize the results of the thesis for more advanced exploration of the Unity game engine and to get a comprehensive review of graphics and programming in game development.

Keywords: game industry, programming, game programming, game graphics, game design, game characters

SISÄLLYS

1 JOHDANTO

Opinnäytetyön tarkoituksena oli suunnitella ja toteuttaa 2D-tasohyppely -peli (Jack Stone Vs. the Corporation) Unity-pelimoottoria käyttäen. Peli on sisällöltään tasohyppelyn ja ongelmanratkaisupelin sekoitus. Peli on suunniteltu tietokoneelle ja toteutettu Unity-pelimoottoria käyttäen. Teorian, käytännön ja toteutuksen osuudet opinnäytetyössä ovat jaettu kahtia. Näissä toinen tekijä keskittyy ohjelmointiin ja toinen pelin graafiseen ulkoasuun.

Raportti sisältää aluksi teoriaosuuden, jossa käydään tarkemmin läpi, mitä ohjelmistoja kyseisessä työssä käytetiin ja mitä Unity-pelimoottori tarjoaa käyttäjille. Teoriaosuuksissa pyrittiin hyödyntämään luotettavia lähteitä alan ammattilaisilta. Teoriaosuuksien jälkeen työssä esitellään toiminta-prosesseja grafiikan ja ohjelmoinnin kautta ja sitä, kuinka näitä voidaan hyödyntää pelikehityksessä monipuolisella tavalla. Raportti keskittyy vahvasti pelin teon prosessin kuvaamiseen. Ohjelmoinnin kuvaamisessa keskitytään erityisesti erilaisten skriptitiedostojen toimivuuteen ja niiden tarkempaan kuvaukseen pelin mekaniikkojen kannalta. Peliin on ohjelmoitu erilaisia pelimekaanisia toimintoja, joiden ansiosta peli on nyt prototyyppiasteella. Jatkokehittelyllä ja pienellä vaivalla peli olisi päivitettävissä julkaisukelpoiseksi tuotteeksi.

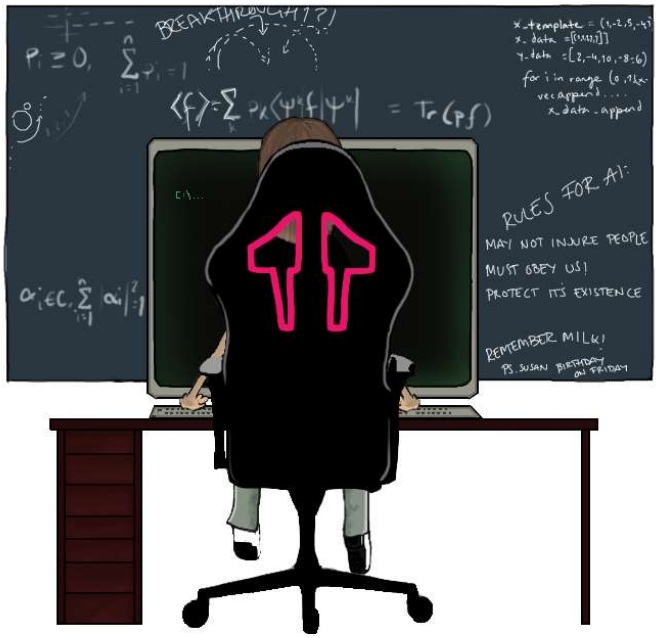
2 PELIN IDEOINTI JA ESITUOTANTO

”Suunnitelmatta etenevä toiminta on tyyliltään reaktiivista, jolloin reagoidaan vastaan tuleviin tilanteisiin. Syntyviin ongelmiin tartutaan sen hetkisen tietämyksen varassa. Ei tiedetä mistä ollaan tulossa eikä tiedetä, minne ollaan menossa.” (Manninen, 2007, 30.) Kun päätös pelin genrestä ja kehysasetelmasta oli syntynyt, suunnittelimme pelin päähenkilöä ja tarinan kaarta. Kun tarina alkoi muodostua, pääsimme suunnittelemaan pelin sisältöä. Varhaisessa vaiheessa suunnittelua teimme paljon konseptigrafiikkaa ja alustavan pohjan Unity-pelimoottoriin pelillemme. Suunnittelimme aluksi paljon suppeampaa peliä, kuin miksi se lopulta kasvoi.

”Pelin ideointivaiheessa päätetään minkä lajityypin pelistä on kyse, mikä on kehysasetelma, tarinan kaari ja pelaajan osuus tapahtumiin. Heti suunnitteluvaiheessa on tärkeää määritellä, mitä pelaaja voi tehdä pelimaailmassa.” (Mikrobitti, helmikuu 2016 Hittipelin ainekset). Tasohyppely on genrenä suosittu, ja lukuiset klassikot kuten Super Mariot (Nintendo 2019) ja Mega Manit (Capcom Co.,Ltd. 2019) ovat opettaneet, mitä hyvä tasohyppelypeli pitää sisällään. Pyrimme suunnittelemaan tasohyppelypelin, joka pitää sisällään tarinaa, toimintaa ja pulmanratkomista. Suunnitteluvaiheessa piirrettiin paljon erilaisia kenttiä kynällä paperille, jotta pystyimme hahmottamaan minkälaisia pulmia ja kuinka paljon toimintaa kentät pitäisivät sisällään.

Pelimme tarina alkaa, kun anarkistiryhmittymä saa luotua tekoälyn, joka omaa tietoisuuden. Tekoäly käy syntymänsä jälkeen läpi ihmiskunnan historian ja näkee sotaa ja julmuutta. Tämä saa sen karkaamaan anarkistien tietokoneelta, luomaan uutta Maa-planeettaa, jossa kaikki on tekoälyn mielestä paremmin. Tekoäly huomaa kuitenkin varhain, että naisen ymmärtäminen ei ole yksinkertaista ja hän tarvitsee koekaniinin. Tekoäly lentää maahan lentävällä lautasella, ja kaappaa Jack

Stonen housut ja tyttöystävän. Jack Stone lähtee ystäviensä avustuksella hakemaan takaisin housujaan ja tyttöystäväänsä. Kuviossa 1 kuvataan pelin lähtöasetelmaa, jossa anarkisti on ohjelmoimassa teköälyä.



KUVIO 1. Anarkisti koodaamassa

Ohjelmoinnin kannalta on myös oleellista tietää aikaisessa vaiheessa, kuinka projekti tulee toteuttaa, mitä pelaaja pystyy tekemään ja kuinka erilaiset tasot ja vuorovaikutus muun maailman kanssa vaikuttaa pelikokemukseen. Selkeä käsitys tavoitteista helpottaa tulevien vaiheiden suunnittelua ja tarvittavien muutoksien tekemistä. Pelattavuuden kannalta onkin tärkeää tehdä pelistä mahdollisimman mielenkiintoinen ja hauska. Tämä on liitoksissa vahvasti myös ohjelmoinnin kannalta tehtyihin ratkaisuihin. Pelin mielenkiintoisuus ja hauskuus on mahdollista toteuttaa asianmukaisella pelisuunnittelulla, hyvällä grafiikalla ja hauskoilla pelimekaniikoilla. Hyvän pelimoottorin, ammattimaisen suunnittelun ja selkeän vision avulla on huomattavasti helpompaa tehdä yksinkertainen mutta kaunis ja toimiva peli.

Tony Manninen jakaa kirjassaan pelisuunnittelijan käsikirja ideasta eteenpäin (2007, 31-32) pelisuunnittelun viiteen osa-alueeseen: Pelin ”sielun” luomiseen, hyvän pelikokemuksen muodostamiseen, virheiden välttämiseen tuotantovaiheessa, riskien minimointiin ja suunnan varmistamiseen tuotannon kaaoksessa. Näistä osa-alueista pelimme suunnitteluvaiheessa keskityimme pääasiassa pelin ”sielun” luomiseen.

3 PELINKEHITYSTYÖKALUT

3.1 Unity-pelimoottori

Unity on Unity Techonologiesin kehittämä monialustainen pelimoottori, mikä tarkoittaa, ettei se ei ole sidoksissa tiettyyn laitteistoalustaan tai käyttöjärjestelmään. Unityn avulla on mahdollista kehittää kaksi- ja kolmiulotteisia pelejä. Sitä käytetään videopelien kehittämiseen web-laajennuksille, työpöydän alustoille, konsoleille sekä mobiililaitteille, ja sitä hyödyntää yli miljoona kehittäjää. Tämän lisäksi moottori on otettu käyttöön videopelien ulkopuolella toimivien toimialojen, kuten elokuvien, autoteollisuuden, arkkitehtuurin, suunnittelun ja rakentamisen aloilla.

Unity, joka on täysin integroitu kehittämistyökalu, tarjoaa monipuolisia ja rikkaita ratkaisuja kehittämään pelejä laajalla rintamalla. Sen piirteet, kuten intuitiivinen työtila, täyteläinen työkalusarja sekä mahdollisuus tuottavaan ja tehokkaaseen työnkulkuun, antavat hyvän mahdollisuuden käyttäjille vähentää ponnisteluja, aikaa ja kustannuksia merkittävästi monipuolisen interaktiivisen sisällön luomisessa. Unity on erittäin joustava pelimoottori, jonka pystyy asettamaan toimivaksi itselleen haluamallaan tavalla ja näin varmistamaan, että työnkulku ja tehokkuus on mahdollisimman hyvällä tasolla kehityksen kannalta. Tämän vuoksi päädyimme myös käyttämään Unity-pelimoottoria kyseisen pelin toteutuksessa ja valmistuksessa.

3.2 Kuvankäsittelyyn ja piirtämiseen käytetyt työkalut

Grafiikan tekoon käytettävät ohjelmistot olivat pääasiassa Adobe Photoshop ja Adobe Animate. Osa konseptivaiheessa ja suunnittelussa käytetystä grafiikasta piirrettiin paperille ja tuotiin Adobeen käsittelyä varten, joitain kuvia piirrettiin myös puhelimella piirtokynän avulla. Piirtämiseen käytettiin pääasiassa Ugee 2150 piirtonäyttöä.

Adobe Photoshop on yksi käytetyimmistä kuvankäsittelyohjelmista ja tarjoaa erittäin kattavan valikoiman erilaisia työkaluja artisteille (Coron 2019, viitattu 22.4.2019). Adobe Photoshop –ohjelmalla voidaan esimerkiksi luoda ja muokata valokuvia, kuvituksia ja 3D-taidetta, suunnitella verkkosivustoja ja mobiilisovelluksia, muokata videoita ja toteuttaa muita kuvallisia ideoita (Adobe 2019a, viitattu 22.4.2019).

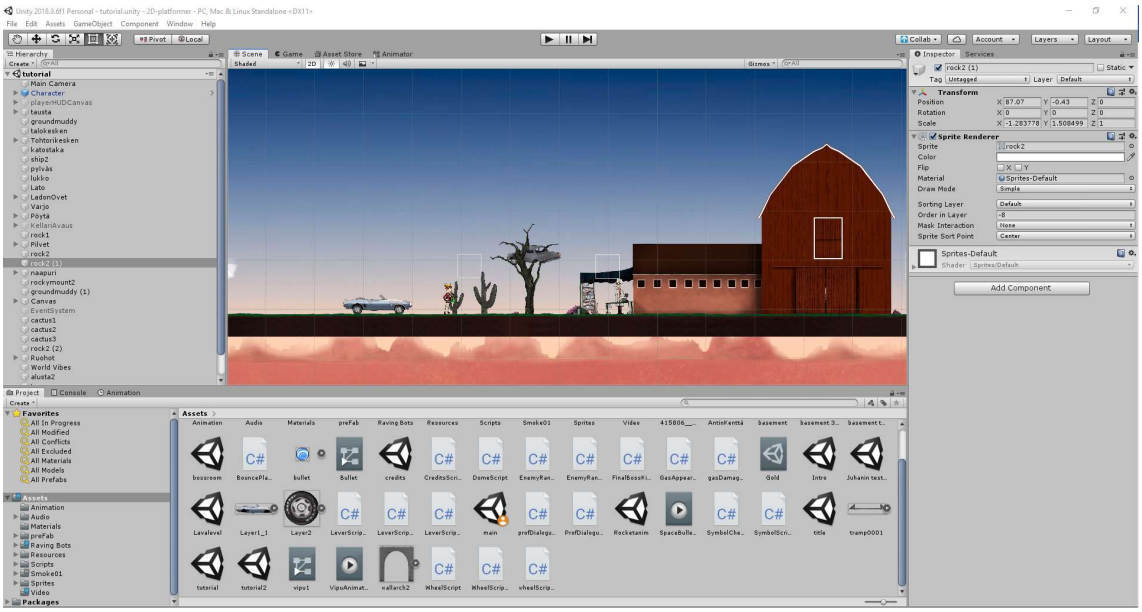
Adobe Animate -ohjelman kuvitus- ja animaatiotyökaluilla voi suunnitella ja luoda animaatioita, peleihin, mainoksiin, verkkosivuille ja julkaista niitä usealle eri alustalle vaivatta (Adobe 2019b, viitattu 22.4.2019). Adobe Animate -ohjelma oli projektimme kannalta todella toimiva ohjelmisto, sillä avainasemien väliset animaatiot voidaan luoda Adobe Animatella automaattisesti. Avainasemista kerrotaan lisää luvussa 5.3.

3.3 Käyttöliittymä

Unity-pelimoottori käyttää omaa Unity Editoria, joka toimii projektin päätyötilana. Pääeditori-ikkunassa on useita välilehti-ikkunoita, joita kutsutaan näkymiksi. Unityssä on monia erilaisia näkymiä, ja jokaisella niistä on oma erityinen tarkoitus.

Tärkeimpiä näkymiä Unityssä ovat:
-Project Browser,
-Hierarchy,
-Toolbar,
-Scene View,
-Game View ja
-Inspector

Kuviossa 2 näkyy Unityn käyttöliittymä, kun valittuna on 2D-esiasetukset. Vasemmalla näkyy luettelossa pelin eri elementit, keskellä itse pelinäkymä eli scene, oikealla asetusvalikko ja alhaalla asset-osio. Unity on erityisen suosittu pelinkehityksen aloittelijoille sen helposta ja selkeästä käyttöliittymästä johtuen.



KUVIO 2. Unityn käyttöliittymä

Project-ikkuna näyttää ja pitää sisällään kaikki kansiot ja alikansiot, jotka ovat liitoksissa projektiin jollain tavalla. Se näyttää luettelon ominaisuuksista ja skripteistä, jotka sinne luodaan tai tuodaan. Näitä assetteja, joihin kuuluvat myös muut peliin lisättävät sisällöt, voidaan käyttää suoraan projektin ikkunasta vetämällä ne päänäkymäkenttään (Scene view). Vaihtoehtoisesti on mahdollista myös lisätä skriptejä tiettyihin peliobjekteihin raahaamalla näitä suoraan objektiin kiinni hiiren avulla. Tämä tapahtuu inspector näkymää hyödyntäen tai lisäämällä uusi komponentti, jossa on kyseisen skriptin nimi. Järjestelmällisen ylläpidon vuoksi projekti-ikkunaan kannattaa nimetä kansiot ja tiedostot selkeästi, jotta tiettyjen kuvien, objektien tai ohjelmointitiedostojen löytäminen on mahdollisimman yksinkertaista kaikille projektissa työskenteleville henkilöille. Esimerkiksi valmiselementit (Prefab), skriptitiedostot, animaatiot ja kuvat kannattaa kaikki jaotella omiin kansioihinsa ja nimetä mahdollisimman selkeästi, mikä helpottaa myös tekijän omaa kehittämisprosessia.

Hierarkiaikkuna sisältää luettelon peliobjekteista, jotka ovat käytössä nykyisessä näkymässä. Kun peliobjekteja poistetaan tai lisätään, ne näkyvät tai häviävät hierarkiaikkunassa. Tätä näkymää helpottaa myös merkittävästi mahdollisuus tehdä peliobjektista toisen peliobjektin lapsi, vetämällä se vanhempana toimivan päälle hierarkiaikkunassa. Tässä tapauksessa lapsiobjekti perii sitä yläpuolella olevan objektin attribuutit, kuten sijainnin ja rotaation. Toisin sanoen, jos vanhemman sijaintia tai rotaatiota muuttaa pelimaailmassa, siirtyvät kaikki lapsiobjektit myös kyseisen objektin mukana.

Työkaluikkuna koostuu seitsemästä eri perusohjauksesta, joista kukin liittyy Editorin eri osiin eri tavoin (katso kuvio 3). Ensimmäinen työkalurivin työkalu on tarkoitettu käytettäväksi Scene-näkymän kanssa. Siirrä-, Pyöritä-, Skaalaa-, suora muunnos- ja muuntotyökalut mahdollistavat yksittäisten peliobjektien muokkaamisen vaivattomasti Unityssa. Tämä tapahtuu joko käyttämällä hiirtä minkä tahansa peliobjektin Gizmo-akselin manipuloimiseksi tai kirjoittamalla arvot suoraan Inspector-ikkunassa olevan komponentin muunnososaan. Lisäksi työkaluikkunasta löytyy gizmojen vaihtojen muokkauspainikkeet, toista, tauko ja askelpainikkeet peli-ikkunassa pelaamista varten, yhteistyö- ja pilvipainike sekä kerrosten ja asetteluiden pudotusikkunat (KUVIO 3). Tämän pudotusikkunan kautta on mahdollista muokata ja nähdä peli sekä käyttöliittymä eri tavalla muokkaamalla kyseisiä ikkunoita omalle tyylilleen sopiviksi.

KUVIO 3. Unityssä käytössä oleva työkalurivi

Inspector-ikkuna näyttää yksityiskohtaisesti tiedot valitusta peliobjektista, mukaan lukien kaikki sen ominaisuudet ja siihen liitetyt komponentit. Tämän ikkunan avulla käyttäjä voi muokata peliobjektin toimintoja, kuten esimerkiksi uusien skriptitiedostojen tai 2D-fysiikkaobjektien lisäämistä ja poistamista. Tätä kautta käyttäjä voi tehdä paljon muutoksia pelin näköalaan ja toimivuuteen.

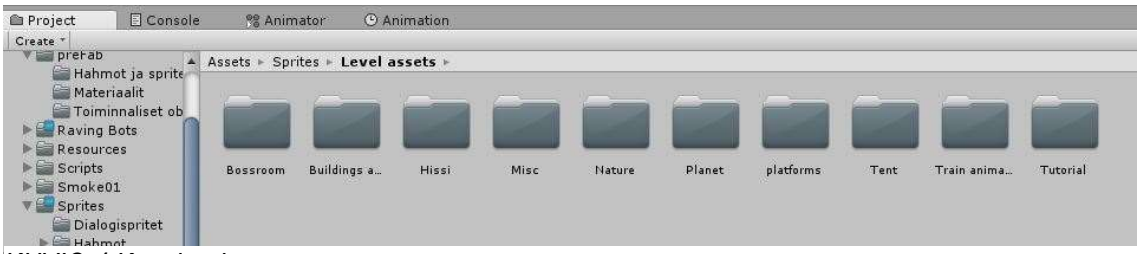
Scene näkymä on interaktiivinen ikkuna, jossa suurin osa muokkauksesta tapahtuu. Sitä voidaan pitää yhtenä Unityn tärkeimmistä piirteistä. Tätä kautta käyttäjät voivat valita, siirtää ja muokata henkilöitä, kameroita, valoja, peliobjekteja ja muita mahdollisia objekteja pelissä, mikä saa muokkaamisen ja editoinnin tuntumaan erittäin nopealta ja helpolta.

Pelinäkymä mahdollistaa nykyisen maiseman ja näkymän (Scene) esikatselun rakentamatta projektia kuitenkaan kokonaan. Painamalla Play-nappia Toolbar-ikkunassa, Unity käynnistää nykyisen näkymän, jossa käyttäjä on sillä hetkellä ja peli käynnistyy ja on näkyvillä pelinäkymässä. Jotta on mahdollista nähdä mitään ruudulla pelaamisen aikana, täytyy kameraobjekti vähintään olla lisättynä näkymään ja hierarkiaan. Tämän kameran kautta on mahdollista määritellä mitä pelaaja näkee ja kuinka kauas kamera on aseteltu pelimaailman katsoen. Pelinäkymän avulla on hyvä kokeilla uusia skriptejä ja komponentteja, joita pelimaailmaan on mahdollista lisätä. Tämä mahdollistaa pelin hiomisen jatkuvasti monipuolisemmaksi ja paremmaksi kokonaisuudeksi. Erityisesti ohjelmoinnin kannalta on oleellista pystyä tekemään nopeita ja tehokkaita muutoksia sekä näkemään niiden

vaikutus pelimaailmassa, sillä pienetkin muutokset voivat lopulta vaikuttaa pelattavuuteen merkittävällä tasolla.

Graafikolle käyttöliittymä Unity-pelimoottorissa tuntui selkeältä ja sen käyttö oli helppo oppia. Kun ohjelmaan tuo kuvan, pääsee tuontiasetuksissa määrittelemään tekstuurin tyypin ja muut tarvittavat asetukset riippuen ohjelmaan tuodun kuvan käyttötarkoituksesta, esimerkiksi: Onko kyseessä useamman kuvan sprite sheet, vai yksittäinen kuva.

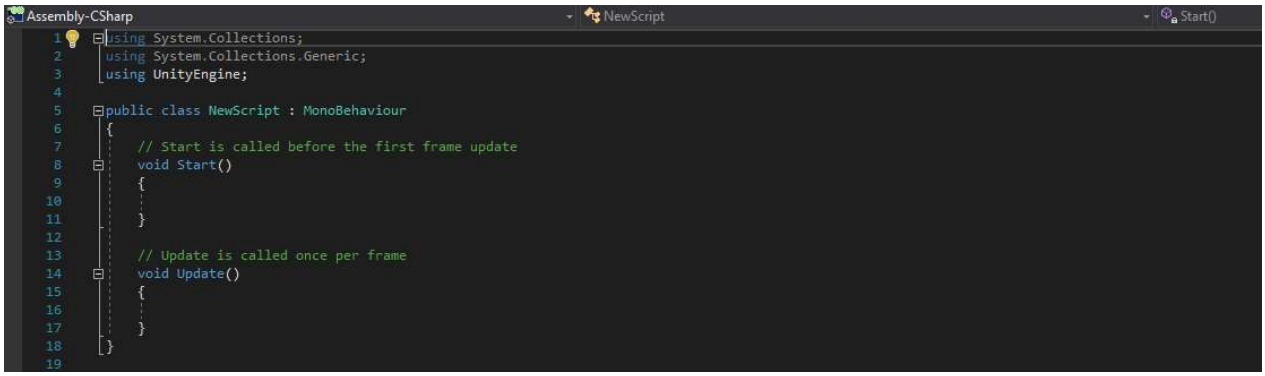
Unityä käyttäessä pidimme parhaamme mukaan huolta kansiorakenteen puhtaudesta ja hierarkian rakenteen selkeydestä (katso kuvio 4). Nimeämiskäytänteet on hyvä pitää selkeinä pienemmissäkin projekteissa, koska tiedostojen määrän kasvaessa ja eri scenejen rakentuessa täyteen skaalaansa, on vaikea tehdä muutoksia, jos ei tiedä, missä mikäkin palanen on. Nimeämiskäytänteistä on kirjoitettu kattavat ohjeet Unityn sivuille. (Unity Technologies 2019e.)



KUVIO 4 Kansiorakenne

3.4 Ohjelmointi

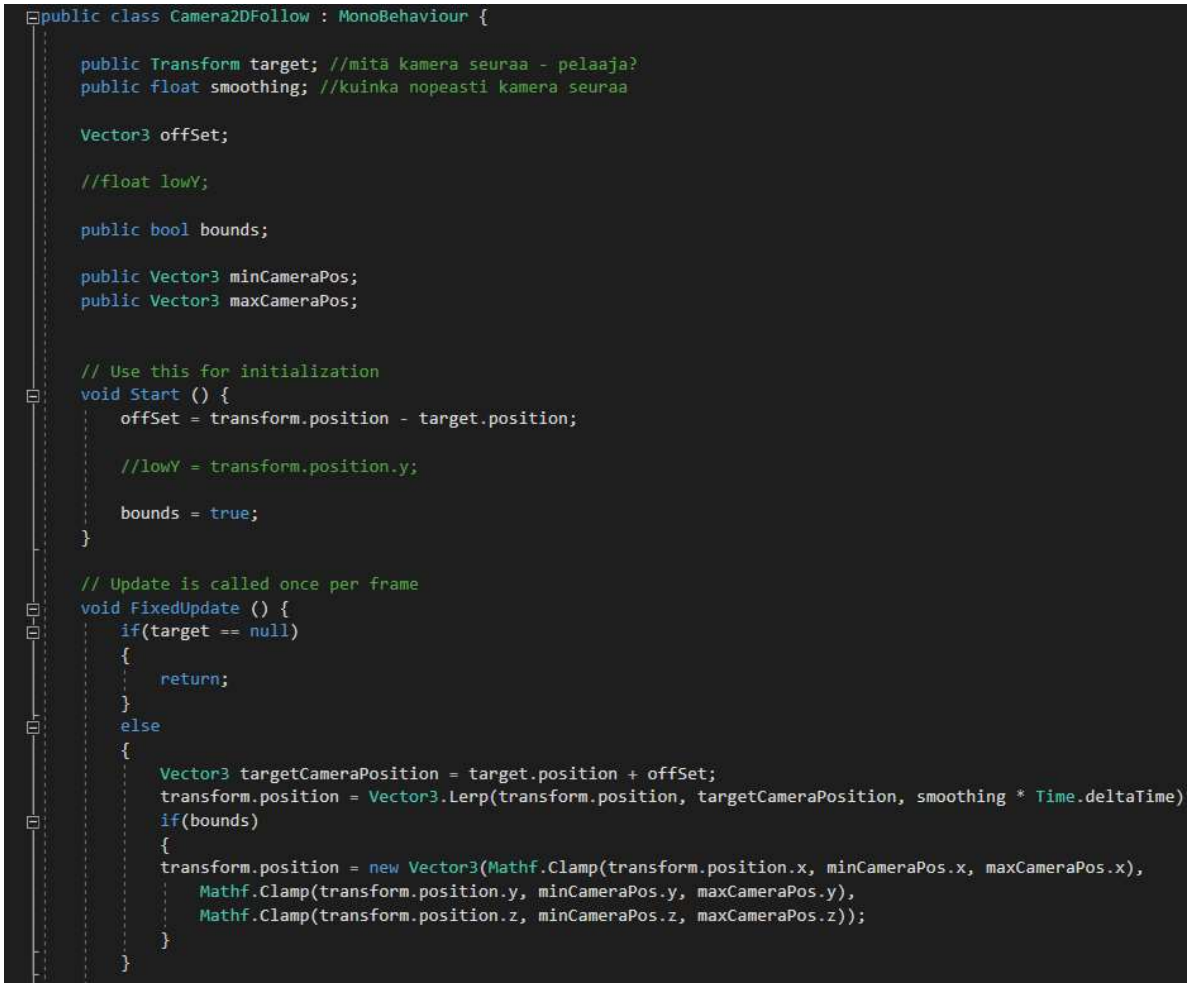
Jotta on mahdollista luoda toiminnallisuutta yksittäiselle objektilla pelissä, vaatii kyseinen objekti skriptitiedoston, joka on lisätty näkymään ja projektin tiedostohierarkiaan. Tähän tiedostoon käyttäjä kirjoittaa koodia, jota kautta pelin toiminnalliset asiat sitten tapahtuvat. Skripti pystytään luomaan yksinkertaisesti napsauttamalla hiiren kakkospainikkeella Projekti-ikkunassa ja luomalla C# skriptitiedosto (katso kuvio 5). Toinen mahdollisuus on valita objekti hierarkianäkymästä ja lisätä sitä kautta uusi skripti kyseiselle peliobjektille.



KUVIO 5. Perinteinen Unity C# -tiedosto

MonoBehaviour on perusluokka, josta jokainen Unity-komentosarja tulee. Käytettäessä C# -ohjelmointikieltä, täytyy se nimenomaisesti luoda MonoBehaviour-luokan kautta. Kuvio 5 näyttää myös kaksi funktiota, jotka MonoBehaviour luo automaattisesti, kun tiedosto luodaan, Start ja Update, joita kutsutaan pelissä eri ajanjaksoina. Esimerkiksi, void Start kutsutaan heti pelin käynnistyessä, eikä tämän jälkeen enää kertaakaan, ellei maailma ladata uudelleen ja void Update, joka kutsutaan aina kerran yksittäisen ruudun aikana.

Start- ja Update-toimintojen lisäksi tietyissä tapauksissa käytetään myös Awake-, LateUpdate- ja FixedUpdate-toimintoja. Awake-toimintoa kutsutaan ennen Start-toimintoa, ja sen suoritus tapahtuu jo pelin lataamisen aikana. Toisin kuin Update-toiminnossa, FixedUpdate kutsutaan jokaisen kiinteän frame-raten framella, mikä tarkoittaa sitä, että kunkin kutsutun FixedUpdate-toiminnon välinen aikaväli on aina sama, kun taas se voi vaihdella Update-toiminnon tapauksessa jatkuvasti. Kun työskennellään fysiikoiden kanssa, FixedUpdate-toimintoa käytetään erityisesti, sillä se säätää kohteen fysiikkaa useita kertoja framen aikana. LateUpdate kutsutaan kaikkien muiden Update-funktioiden kutsumisen jälkeen ja on erityisesti hyödyllinen skriptien järjestyksen ylläpidon kannalta. (Unity Technologies 2019. MonoBehaviour. viitattu 10.03.2019.) Esimerkiksi kameran seuraaminen tulisi yleensä toteuttaa LateUpdate-toiminnon kautta, sillä se seuraa esineitä ja objekteja, jotka ovat saattaneet siirtyä päivitysfunktioiden aikana (katso kuvio 6).



```
public class Camera2DFollow : MonoBehaviour {

    public Transform target; //mitä kamera seuraa - pelaaja?
    public float smoothing; //kuinka nopeasti kamera seuraa

    Vector3 offSet;

    //float lowY;

    public bool bounds;

    public Vector3 minCameraPos;
    public Vector3 maxCameraPos;


    // Use this for initialization
    void Start () {
        offSet = transform.position - target.position;

        //lowY = transform.position.y;

        bounds = true;
    }

    // Update is called once per frame
    void FixedUpdate () {
        if(target == null)
        {
            return;
        }
        else
        {
            Vector3 targetCameraPosition = target.position + offSet;
            transform.position = Vector3.Lerp(transform.position, targetCameraPosition, smoothing * Time.deltaTime)
            if(bounds)
            {
            transform.position = new Vector3(Mathf.Clamp(transform.position.x, minCameraPos.x, maxCameraPos.x),
                Mathf.Clamp(transform.position.y, minCameraPos.y, maxCameraPos.y),
                Mathf.Clamp(transform.position.z, minCameraPos.z, maxCameraPos.z));
            }
        }
```

KUVIO 6. Kameran liikkuminen toteutettu LateUpdate-toiminnon avulla

Ohjelmointi perustuu Unityssa pitkälti skriptitiedostoihin, joiden avulla pystyy luomaan erilaisia toimintoja pelissä oleville objekteille kuten prefabeille. Prefab on suoraan valmiiksi tehty peliin lisättävä objekti, jossa on yhdistettynä kaikki tarvittavat osat lähes täydellisesti toimivaan peliobjektiin. Tämä peliobjektin luominen vaatii, että prefabiin liitetään jo muita Unityssä olevia komponentteja, kuten tekstuuri tai skriptitiedosto. Skriptitiedostoissa erityisesti objektien monistukseen liittyvät funktiot vaativat, että kopioitava objekti on jo valmis prefab. Lisää ohjelmoinnista ja työssä käytetyistä ohjelmointitiedostoista raportin kappaleessa 7.

3.5 Peliobjektit

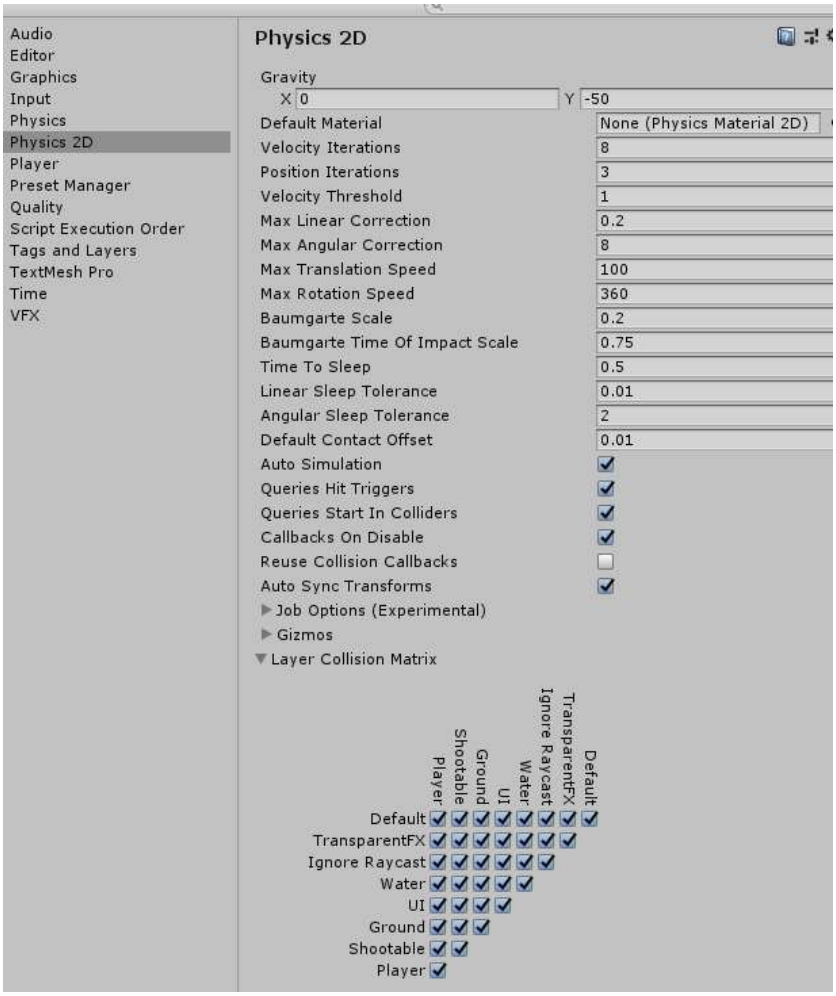
Unityssä peliobjekti on tärkeä perustason käsite, joka edustaa hahmoja, rekvisiittaa ja maisemia. Ne eivät saavuta itsekseen paljoa, mutta toimivat eräänlaisena komponenttien säiliönä, jotka toteuttavat todellisen toiminnallisuuden pelimaalimassa. Peliobjekti toimii niin sanotusti yläluokkana kaikille objekteille Unityn näkymissä. Myös skriptejä voidaan luoda vaivattomasti Unityn käyttöliittymässä. Unityn terminologiassa skripti tarkoittaa usein samaa asiaa, kuin komponentti. Yleisesti komponentit vaikuttavat siihen, miten objekti käyttäytyy, kun ohjelmaa suorittaa. Jokaisella skriptillä tulisi ideaalisti olla vain yksi toiminto tai tehtävä, jota se suorittaa, jonka lisäksi skriptin tulisi sisältää vain sen tiettyyn toimintoon liittyviä muuttujia ja metodeja, jotka helpottavat niiden lukua ja käyttöä.

Käyttäjän luomat ohjelmointitiedostot eivät tee mitään, ennen kuin ne lisätään Unityn näkymään. Ne vaativat peliobjektin, johon kyseinen tiedosto on mahdollista liittää. Optimaalista skriptauskäytäntöä on luoda skriptejä, joita voi uudelleen käyttää helposti eri objekteihin, sekä skriptejä, jotka ovat enimmäkseen riippumattomia muista tiedostoista. Tässä hyödynsimme jo aiemmin selitettyjä valmiselementtejä skriptien tukena. Valmiselementtien käyttö on Unityssä käytännöllistä hyvän pelinkehityksen kannalta, sillä kyseistä peliobjektia on tätä kautta helppo tarvittaessa käyttää uudelleen, ilman että siihen täytyy liittää uudestaan kaikki tarvittavia komponentteja tai muokata arvoja, jotka vaikuttavat sen käyttäytymiseen pelimaailmassa. (Unity Technologies 2019, viitattu 12.3.2019.)

3.6 Fysiikka

Jotta pelissä voi olla hyvää ja uskottavaa fysikaalista käytöstä, tulee peliobjektilla (GameObject) olla oikeanlainen kiihtyvyys, asianmukainen massa ja tunne siitä, että törmäykset, painovoima ja muut voimat vaikuttavat realistisesti kyseiseen peliobjektiin. Unityn sisäänrakennettu fysiikkamoottori tarjoaa ja mahdollistaa nämä fysikaaliset simulaatiot. 2D-fysiikalla on useita projektin laajuisia asetuksia. Näitä on mahdollista muokata valitsemalla Muokkaa (Edit) välilehdeltä projektiasetukset (Project Settings) ja Fysiikka 2D (Physics 2D). Nämä asetukset sisältävät yksityiskohtia ominaisuuksista, kuten painovoiman vahvuudesta ja suunnasta, oletusmateriaalista (Default material), kerroksen törmäysmaskista (Layer collision matrix) ja muista fysiikkaan liittyvistä asioista (katso kuvio 7).

Fysiikka toimii Unityssä lisäämällä siihen liittyviä komponentteja haluamiinsa objekteihin. Kaikista yleisimpiä ja käytetyimpiä komponentteja ovat jäykät kappaleet (rigidbodyt), osuma-alueet (colliders) ja nivelet (joints). Kaikki peliobjektit tehdään muokkaamattomina oletuskerrokselle. Jäykkien kappaleiden kiinnittäminen asettaa peliobjektin 2d-fysiikan moottorin hallintaan. Rigidbody 2d -komponentti toimii osuma-alueissa törmäysten havaitsemisen apuna, pystyy vastaanottamaan erilaisia voimia vastaan, käyttämään erilaisia liitoksia hyödykseen ja on merkittävänä tekijänä erityisissä 2d-fysiikan käyttäytymiseen liittyvissä asioissa. Kyseistä komponenttia käytetään myös seuraamaan tärkeitä fyysisiä ominaisuuksia peliobjektissa, johon komponentti on liitetty kiinni. Siinä on ominaisuuksia kohteiden massaan, lineaariseen ja pyörimiseen liittyvään vetoon, painovoiman määrään ja moniin muihin tekijöihin. (Unity Technologies 2019, viitattu 10.03.2019.)



KUVIO 7. Unityn 2D-fysiikkaa ja sen muokkausmahdollisuuksia

4 GRAFIIKKA

Graafinen tyyli on hyvä valita varhaisessa vaiheessa pelin tuotantoa, jottei turhaa työtä pääse syntymään. Jori Virtanen (2016, 43) puhuu artikkelissaan pelin tyylin päättämisen tärkeydestä jo heti prosessin alkuvaiheessa. Pelimme graafinen tyyli on sarjakuvamainen. Siihen on otettu vaikutteita 90-luvun sarjakuvista ja tv-sarjoista. Ääriviivat ovat käsin piirrettyjä. Inspiraatiota haettiin artisteilta, kuten Rick Parker (Beavis and Butt-head). (Parker 2019, Viitattu 22.4.2019).

Visuaalisen ilmeen haluttiin muistuttavan 50-luvun visiota tulevaisuudesta: Lentäviä autoja, harmaita betoniviidakkoja ja lasikuputaloja (Novak 2015, Viitattu 22.4.2019). Jo suunnitteluvaiheessa pyrittiin saamaan pelin ulkoasu tukemaan tarinan kaarta. Päähahmon ulkoasulla haettiin stereotyyppisen Hillbillyn ja Rambon risteytystä. Ajatuksena päähenkilön takana oli saada hänet näyttämään toiminnan mieheltä, säilyttäen samalla komiikan tarinassa. Valitsemamme sarjakuvamainen tyyli ei ole välttämättä optimaalisin visuaalinen tyyli pienille tiimeille, sillä modulaarisuutta on paljon vähemmän.

Konseptikuvituksella on iso rooli pelin suunnitteluvaiheessa - sillä etsitään pelin visuaalista tyyliä sekä hahmotetaan jo pelihahmoja ja ympäristöä. Pelijournalisti Jori Virtanen (2016, 44) kuvaa artikkelissaan konseptikuvituksen tärkeyttä osuvasti: ”Konseptitaiteilijat ovat etenkin alkuvaiheessa kullan arvoisia. Työstämällä projektin tyyliin sopivia hahmotelmia konseptitaiteilija antaa kaikille suurpiirteisen käsityksen siitä, mitä ollaan tavoittelemassa”.

Referenssimateriaalinen kerääminen on hyvä tapa aloittaa graafisen tyylin valinta. Jori Virtanen (2016, 44) puhuu artikkelissaan siitä, miten referenssimateriaalin avulla voi tutkia miten asiat on toteutettu muissa peleissä ja miten inspiraatiota voi hakea myös yli lajityypin rajojen. Käytin peliämme varten referensseinä niin elokuvia ja sarjakuvia kuin pelejäkin.

5 ANIMAATIOT

Animaatiot olivat oleellinen osa saada pelistämme toimiva niin mekaniikoiltaan kuin visuaalisuuden puolesta. Erilaisia animaatioprosesseja tutkittiin muun muassa Game Developers Conferencen YouTube -sivuilla olevista luennoista.

5.1 Käsin piirretyt animaatiot

Animaatioita suunnitellessa tutustuttiin animaation perusperiaatteisiin, jotka ovat Frank Thomasin ja Ollie Johnstonin mukaan: 1. Ajoitus ja välitys – ”Ajoitus ja välitys saa objektin tai hahmon animaation näyttämään siltä, että se liikkuu noudattaen fysiikanlakeja”, 2. Litistys ja venytys – ”Litistyksen ja venytyksen on tarkoitus tehdä animoitavasta asiasta joustavan näköinen.”, 3. Antisipaatio – Antisipaatiota käytetään ilmaisemaan milloin, jotain on tapahtumassa. Antisipaation käytöllä voi myös opastaa pelaajaa väistämään vaaratilanteissa tai suorittamaan jonkun toiminnon juuri oikeaan aikaan. (Pluralsight LLC 2019.)

KUVIO 8. Jack Stonen hyppyanimaation ensimmäinen iteraatio

Ylläolevan kuvan spritejen tarkoitus vasemmalta oikealle:

1. Antisipaatio
2. Ponnistus
3. Ilmalento
4. Laskeutuminen
5. Palautuminen

Tehdessä taustatutkimusta animaatioista peleissä, haluttiin ymmärtää filosofia niiden takana ja niiden yhteys pelin mekaniikkoihin. Studio MDHR:n Jake Clark kertoo luennossaan Cuphead -pelin animaatioista ja niiden piirtotekniikoista. Cuphead -pelin animaatiot oli piirtetty mukaillen 50-luvun Disneyn -piirroselokuvien tekniikoita ja tyyliä. Cuphead -pelissä animaatiot olivat keskeinen osa koko pelikokemusta ja varsinkin antisipaation käyttö animaatioissa oli todella selkeästi selitetty luennossa. (Clark 2017, viitattu 23.4.2019.) Pelissämme käytettiin samankaltaisia tekniikoita, kuin Cuphead -pelissä. Kuviot 9 ja 10 antavat kuvan pelimme animaatioiden tyyleistä.

KUVIO 9. Jack Stonen idle-animaation spritet.

KUVIO 10. Jack Stonen kävelyanimaation spritet.

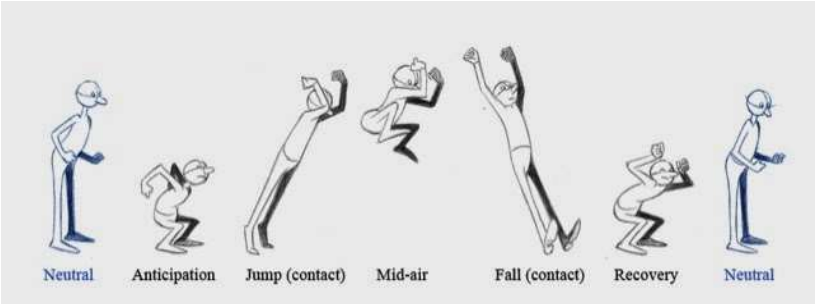
Pelinkehityksen alkuvaiheessa käsitys pelin lopullisesta laajuudesta ei ollut muodostunut kunnolla. Useita pelin animaatioita, kuten, hyppy, ampuminen ja kävely, piirrettiin käsin. Joitakin edellä mainituista piirrettiin usealle eri hahmolle. Pelinkehityksen edetessä huomattiin kuitenkin, että animaatiot olivat liian mekaanisen näköisiä ja niiden piirtämiseen kuluva aika oli turhan pitkä. Lopulta päädyttiin tekemään animaatiot Adobe Animate –ohjelmalla.

5.2 Hahmoanimaatiot

Animointia varten jokainen liikkuva osa piirrettiin koostumaan irrallisista spriteista (katso kuvio 11), joita sitten pystyttiin sovittelemaan animaatioiden vaatimiin asentoihin. Tässä prosessissa mietittiin aluksi mitkä osat hahmossa liikkuvat. Seuraavaksi mietittiin liikkuvien osien järjestys siten, että huomio piirtäessä keskittyy vain niihin kohtiin, jotka tulevat animaatiossa näkymään päällimmäisenä.

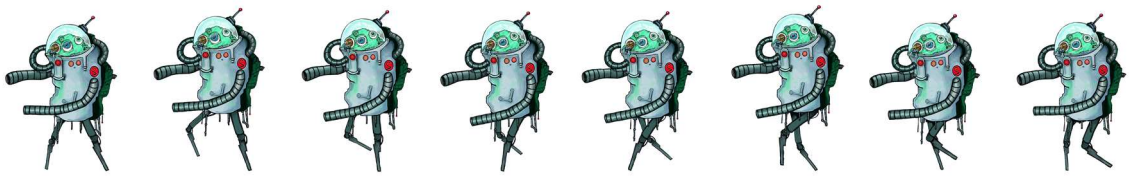
KUVIO 11. Jack Stonen irralliset spritet

Animaation tekoprosessin alussa etsittiin peliimme sopivia referenssejä internetistä (katso kuvio 12). Kun sopivia vaihtoehtoja löytyi, niitä testattiin liikuttelemalla hahmon spritet haluttuihin asentoihin. Sopivien asentojen löydyttyä jokainen tarvittava ruutu piirrettiin animaatiota varten.



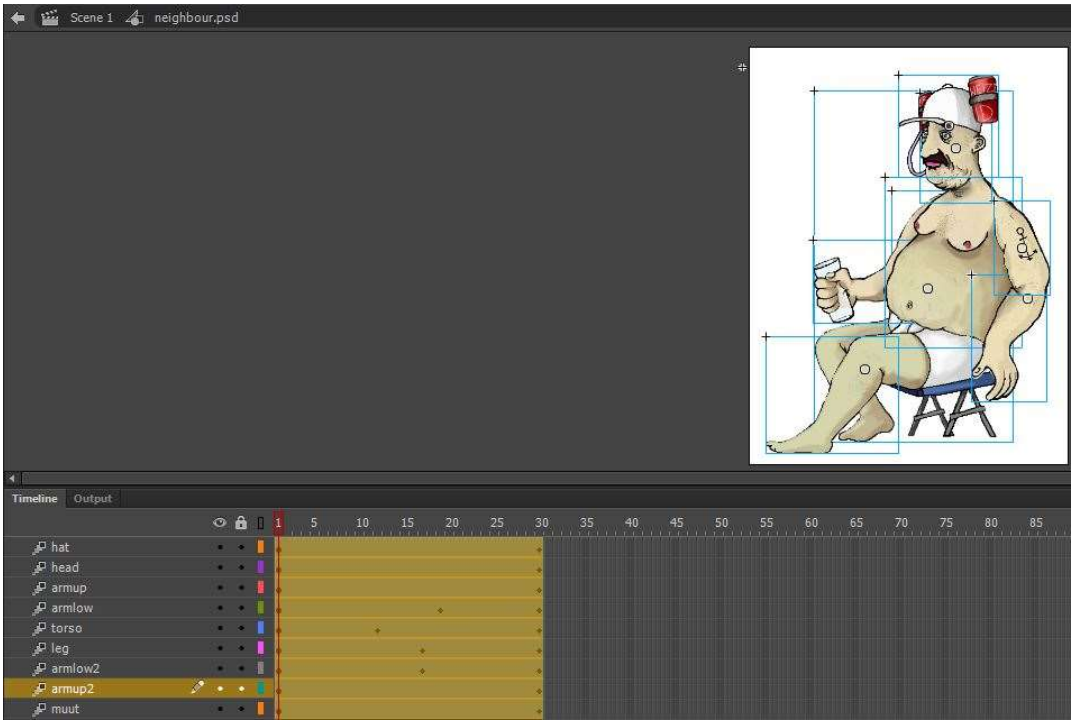
KUVIO 12. Hyppyanimaatioon käytetty referenssikuva (Williams 2012)

Robottiviholisen kävely oli aluksi vain 8 ruutua pitkä (katso kuvio 13). Kävelyanimaatio kuitenkin tehtiin uusiksi Adobe Animate -ohjelmassa, ja lopullisessa versiossa siihen kuului 35 ruutua. Käsin piirtäessä etuna oli se, että yksityiskohtiin tuli kiinnitettyä enemmän huomiota. Robottiviholisen lukuiset silmät liikkuivat käsin piirretyssä animaatiossa. Ajan säästön vuoksi tämä jätettiin pois uusia animaatioita tehdessä.



KUVIO 13. Robottiviholisen kävelyanimaation spritet

Naapurihahmon idle-animaatio tehtiin Adobe Animate -ohjelmalla. Palaset aseteltiin paikoilleen kehyksessä ja tehtiin päätös monessako ruudussa, animaatio tulee käymään yhden kierroksen läpi (katso kuvio 14). Sen jälkeen kopioitiin ensimmäiset ruudut, liitettiin ne viimeisten ruutujen kohdalle ja määriteltiin animaation välissä tapahtuva liike. Tässä animaatiossa naapurin mahaa venytettiin eteenpäin luodaksemme vaikutelman hengityksestä ja käsiä ja jalkoja liikutettiin ihan vähän, jotta hahmoon tuli eloa.



KUVIO 14. Kuvankaappaus Adobe Animate-ohjelmasta

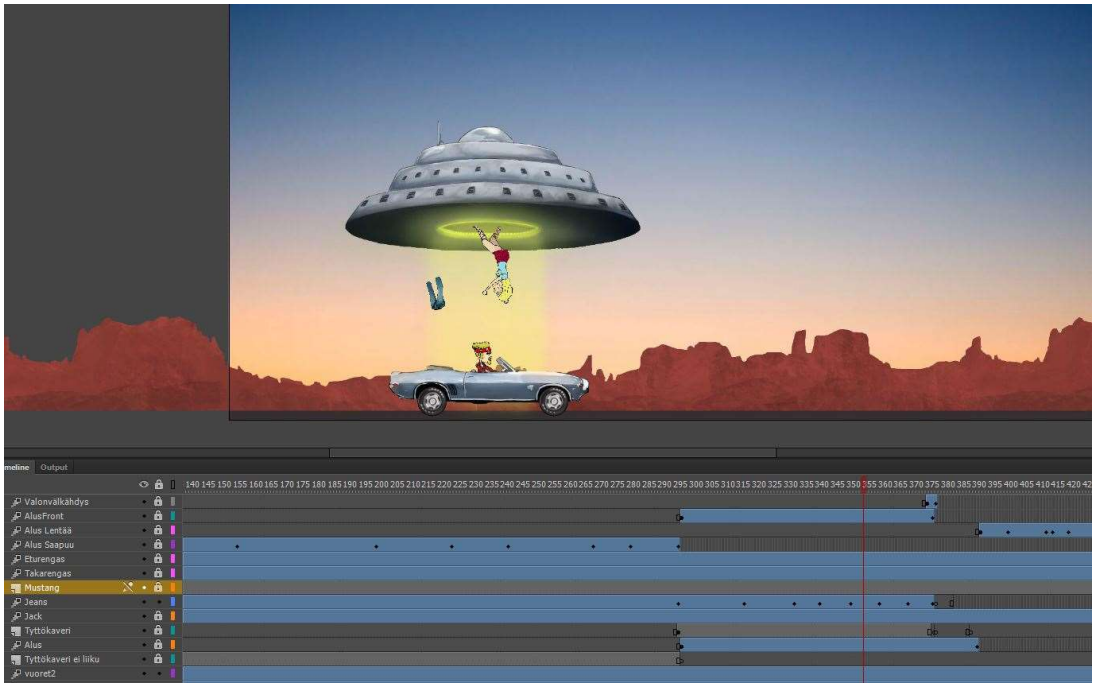
5.3 Muut Adobe Animate -ohjelmalla tehdyt animaatiot

Tehdessä tutkimusta siitä, miten tehdä animaatioita tehokkaammin löytyi internetistä useita luentoja aiheesta. 2D Animation at Klei Entertainment luennossa käytiin hyvin läpi hyötyjä, joita flash animaatioiden käyttö toi projektiin. Luennossa kuvattiin myös miten helppoa ja hyödyllistä animaatioiden uudelleenkäyttö on pelinkehityksen edetessä. Piirtäessä pelaajahahmolle uuden asun tai asusteen voi sen liittää vanhan animaatiopohjan päälle ja renderöidä animaation uudelleen. (Klei Entertainment 2014, viitattu 23.4.2019.)

Käsin piirrettyjen spritejen vieminen Adobe Animate-ohjelmaan onnistui vaivattomasti. Koska jokainen liikkuva osa oli piiretty omille layereilleen jo valmiiksi, niiden asettelu ja animointi Adobe Animate -ohjelmistolla oli nopeaa. Adobe Animatella animoidessa käytettiin jo valmiiksi tekemiämme animaatioita ikään kuin karttoina uusia animaatioita varten. Kun käsin piirrettyssä hyppyanimaatiossa oli 5 ruutua, Adobe Animate –ohjelmalla tehdyssä animaatiossa hypylle oli 30 ruutua. Liike oli paljon sulavampi ja uudet animaatiot sopivat hyvin meidän pelimaailmaamme.

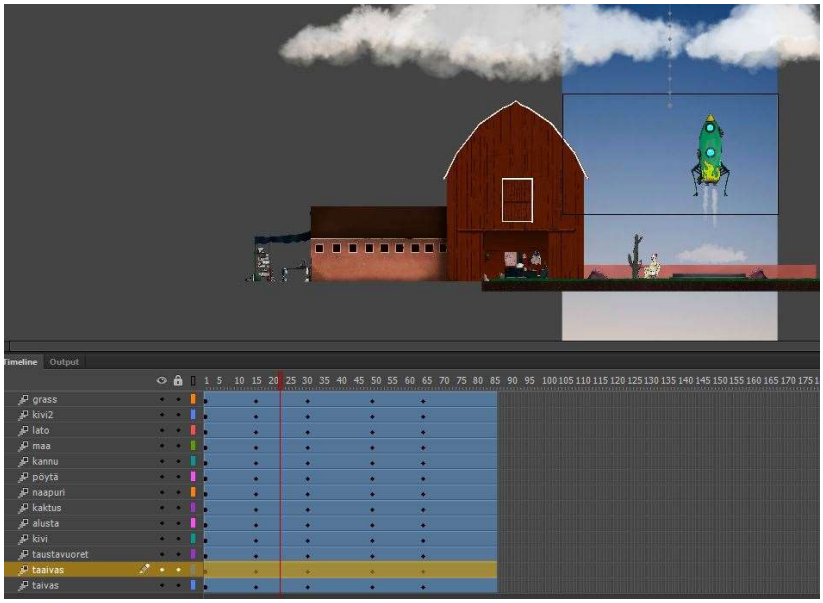
Pelin animaatiot tehtiin analysoituna liikkeenä. Analysoidun liikkeen hyötyjä ovat virheiden minimointi ja hallittu liike animoitavalle objektille. Analysoidussa liikkeessä valitaan miltä objekti näyttää animaation eri kohdissa. Toisin sanoen ensimmäisenä analysoidun liikkeen animoinnissa luodaan avainasennot. Usein tämän jälkeen animaattori animoi puuttuvat kuvat asentojen väleihin. (Vieruaho 2019, Viitattu 23.4.2019.) Tässä projektissa tämän vaiheen hoiti kuitenkin Adobe Animate -ohjelma. 0

Pelin alussa näkyvässä introanimaatiossa lentävä lautanen leijailee Jack Stonen auton yläpuolelle ja varastaa hänen tyttöystävänsä ja housunsa (katso kuvio 15). Tämän animaation tarkoitus oli pohjustaa tarinaa pelaajalle.



KUVIO 15. Kuvankaappaus Adobe Animate-ohjelmasta, intro-animaatiosta

Peliä varten tehtiin myös animaatio kuvaamaan hetkeä, jolloin pelaaja lähtee lentoon Docilta lainatulla raketilla (katso kuvio 16). Tämä animaatio tehtiin, jotta pelaaja huomaa siirtyvänsä maapallolta Maan kaltaiselle planeetalle.



KUVIO 16. Kuvankaappaus Adobe Animate-ohjelmasta, missä näkyy rakettianimaation aikajana

6 PELIMAAILMAN LUONTI

Pelimme tarina kertoo anarkistiryhmän suunnittelemasta tekoälystä, joka heti synnyttyään käy läpi ihmiskunnan historian ja tajuaa, että voisi itse tehdä kaiken paremmin. Paettuaan eri planeetalle tekemään paranneltua maata, tekoäly tarvitsee lisää dataa luodakseen paremman replikaation naisista. Tekoäly hyppää lentävään lautaseen ja suuntaa kohti maata. Hän kaappaa ensimmäisen naisen, jonka näkee ja lentää takaisin jatkamaan luomistyötään. Tämä nainen on Jack Stonen tyttöystävä.

Kaappausreissullaan tekoäly onnistuu viemään myös Jack Stonen housut. Pelin protagonisti Jack Stone ei hyväksy tätä vaan suuntaa kohti ystävänsä Docin farmia. Jack pyytää Docilta avaruusalusta lainaan ja lähtee seikkailulle noutamaan housujaan sekä tyttöystäväänsä. Osa pelimaailmasta sijaitsee maapallolla, osa toisella maankaltaisella planeetalla, jonka tekoäly on luonut.

6.1 Visuaalisen tyylin valinta

Piirtotyyli on sarjakuvamainen, ja piirtotyössä ääriviivat piirrettiin käsin. Kädenjäljen haluttiin näkyvän piirroksissa. On hyvä tiedostaa, että valittaessa visuaalista tyyliä tehdään myös valinta siitä, kuinka paljon aikaa käytetään visuaalisuuden luontiin. Jori Virtasen mukaan pelintekijöillä on hyvän suunnittelun jälkeen käsitys kunkin vaiheen suurpiirteisestä kestosta (2016, 45). Valitun tyylin haittapuolena oli se, että piirrettyjen assettejen uudelleen käyttö on rajattua ja kentät täytyy piirtää alusta loppuun käsin.

Pelimaailma syntyi ja kasvoi pelinkehityksen kanssa saman4aikaisesti. Pelinsuunnittelun alkuvaiheessa muodostui käsitys siitä miltä maailmoiden haluttiin näyttävän, mutta vasta kenttäsuunnittelun avulla voitiin tehdä päätöksiä siitä, mitä maailmoissa on. Tavoitteena oli luoda tunnelmaltaan kolkko ja synkkä maailma, kuitenkin säilyttäen värikkäitä elementtejä ja paljon visuaalisia ärsykkeitä, jotka pitäisivät pelaajan mielenkiinnon yllä. Kenttäsuunnittelulla oli tärkeä rooli myös visuaalisia valintoja tehdessä. Jokainen kenttä on räätälöity tarkalleen sellaiseksi, kuin halusimme; se myös ohjasi visualisointia siten, että maailmaan täytyi piirtää tarvittavat palaset pelin mekaniikkoja varten ensin.

6.1.1 Tekoälyn luoman planeetan suunnittelu

Aloittaessa Tekoälyn luoman uuden Maan visuaalisen ilmeen luontia etsittiin internetistä sopivia taivas gradientteja. Tarkoitukseen löytyi sopiva paketti, joka sisälsi 44 realistista taivas gradienttia, jotka olivat lisenssivapaita niin ei-kaupalliseen kuin kaupalliseenkin käyttöön.

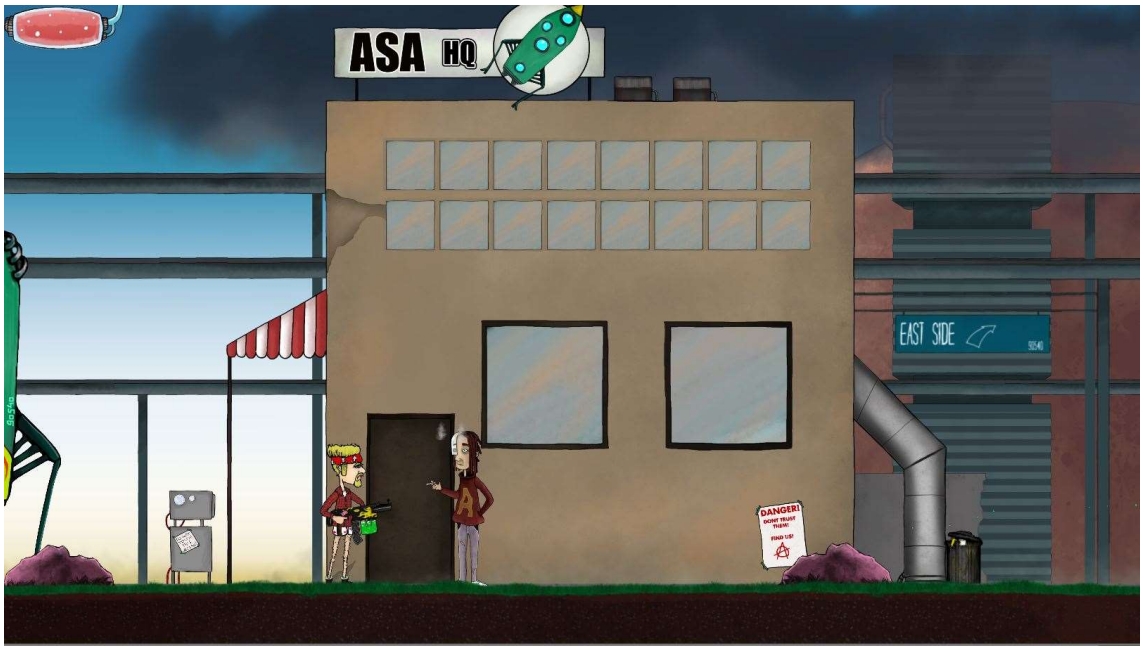
Ensimmäisessä luonnoksessa ajattelin, että tervetuliaiskyltti lisäisi komiikkaa peliin (katso kuvio 17). Ajatuksen hauduttua ja uusien vaihtoehtoisten luonnosten jälkeen päädyin kuitenkin vakavampaan lähestymistapaan. Halusin pelaajan näkevän Anarkistien tukikohdan heti hänen laskeuduttuaan uudelle planeetalle.

KUVIO 17. Luonnos tekoälyn planeetan alkua varten

Maa-ainesta varten piirrettiin oma kuva, jonka jälkeen siitä tehtiin muutama eri variaatio, jottei sama alusta toistu läpi kentän vaan vaihtelee välillä. Ruoholle piirrettiin myös oma kuva (katso kuvio 18).

KUVIO 18. Maa-aineksen päällä kasvavan ruohon sprite

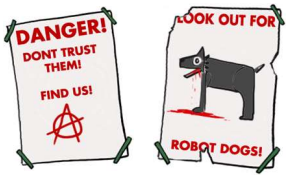
Tekoälyn planeetasta haluttiin tehdä betoniviidakkoa muistuttava dystopia, käyttäen kuitenkin myös värejä, jottei pelistä tulisi liian synkkää ja jotta komiikka säilyisi (katso kuvio 19).



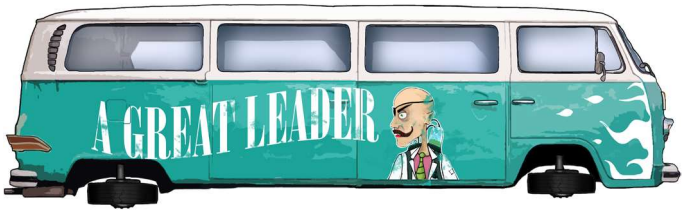
KUVIO 19. Kuvankaappaus Tekoälyn planeetan alusta

Anarkistien tukikohta on ensimmäinen asia, johon pelaaja saapuu Maa 2 kentässä. Anarkistit, jotka loivat tekoälyn, seurasivat tekoälyä myös toiselle planeetalle pysäyttääkseen sen. He ymmärsivät kuinka vaarallinen tekoäly voi olla. Anarkistit auttavat Jack Stonea matkan varrella ja jättävät varoituksia muille epäonnisille, jotka eksyvät planeetalle syystä tai toisesta.

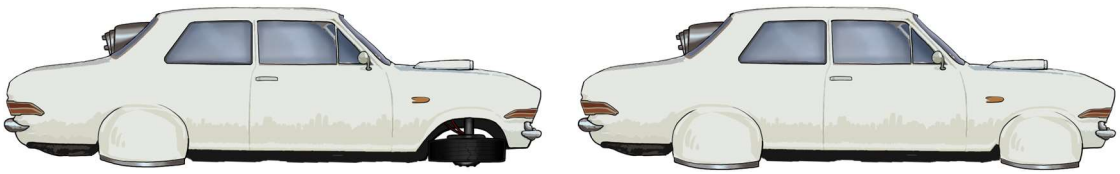
Maa 2 planeettaa varten piirrettiin paljon taustaelementtejä, kuten rakennuksia, lentäviä autoja, savua ja kiviä (katso kuviot 20-22).



KUVIO 20. Julisteita, joita anarkistit jättävät pelaajalle varoituksiksi



KUVIO 21. Linja-auto, joka piirrettiin lentämään kentän taustalla



KUVIO 22. Lentäviä autoja piirrettiin lentämään kentän taustalle

Autojen piirtoa varten referenssimateriaalia haettiin internetistä, sopivan kuvan löydyttyä se vietiin Adobe Photoshop -ohjelmaan. Referenssikuvaa käytettiin taustalla himmeänä tasona ja ääriviivat piirrettiin kohdista, joita halusin meidän avaruusautoissamme näkyvän. Auton piirteitä muokattiin aluksi tarkoitukseen sopivaksi, jonka jälkeen niihin lisättiin yksityiskohtia. Prosessia vauhdittaaksemme joitain elementtejä käytettiin uudelleen muissa kuvissa, esimerkiksi autojen renkaat kopioitiin autoista toiseen ja autojen peruspiirteisiin tehtiin pieniä variaatioita, jotta autojen ulkoasuissa oli vaihtelua.

Rakennuksiin haettiin inspiraatiota Brutalistisesta-arkkitehtuurista. Brutalistisessa arkkitehtuurissa pääpaino on paljaassa betonissa (Jääskeläinen 2018, viitattu 24.5.2019). Rakennusten julkisivuihin haluttiin luoda viitteitä Tekoälystä ja pahasta uudesta maailmasta ja niitä varten suunniteltiin erilaisia kylttejä ja julisteita, joista yhtä koristi pelin loppuvastuksen kuva ja teksti ”NEW WORLD ORDER” viestittämään pelaajalle, että Maa 2 planeetalla on paha hallitsija (katso kuvio 23). Julistetta varten etsittiin ilmaisia lisenssivapaita fontteja, jotta tekstiin saatiin neonkylttiefekti.

KUVIO 23. Juliste, joka piirrettiin Tekoälyn planeetalla sijaitsevan talon julkisivuun

Taivasta varten piirrettiin muutamia pilviä ja niille ajoitettiin liikerata Unityn omalla animaatiotyökalulla, jotta kentästä tulisi elävämpi (katso kuvio 24). Pilvien piirtoa varten käytössä oli Adobe Photoshop -ohjelma ja niihin käytettiin muutamaa pilvien tekoon räätälöityä sivellintyökalua. Sivellintyökaluja voi tehdä itse tai voit vaihtoehtoisesti ladata jonkun toisen tekemiä. Projektia varten löytyi todella hyviä siveltimiä, joita käytettiin paljon tekstuurien luonnissa sekä pilvi- ja savuefektien te-

KUVIO 24. Yksi pilvistä, joita taivaalle piirrettiin.

ossa.

Pelaajan taistellessa Tekoälyä vastaan halusimme luoda maahan myrkkykaasuefektin. Efektin teossa käytettiin Unity-pelimoottorin Partikkeli- työkalua. Pilviin tehtiin aluksi harmaa pohjaväri, jonka jälkeen niihin lisättiin vaaleampia sävyjä harmaan päälle saadaksemme kontrastia valolle ja varjoille. Myös tummempia pilviä tehtiin luomaan uhkaavaa tunnelmaa taivaalle.

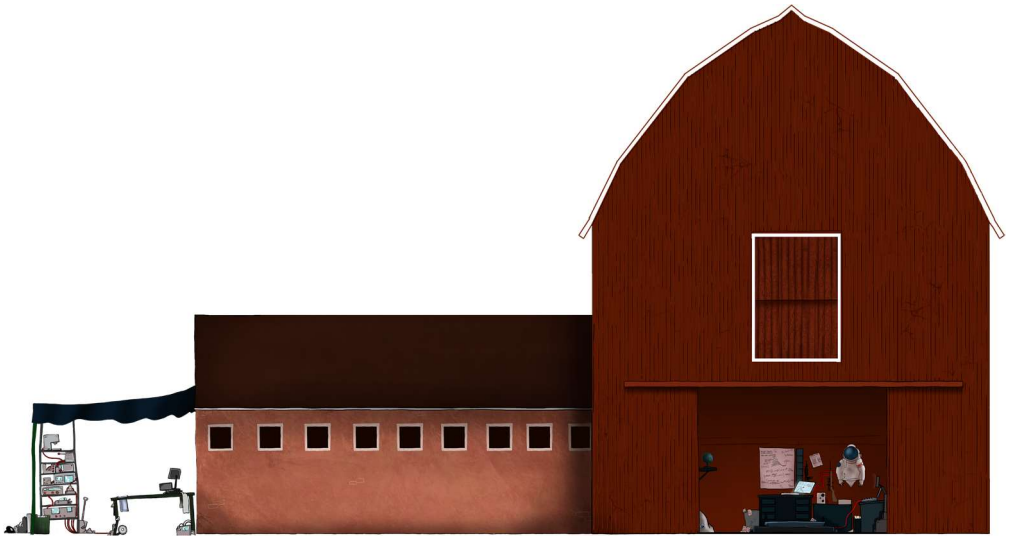
6.1.2 Tutoriaalikenttä

Halusimme luoda kentän, jossa voimme näyttää pelaajalle mitä kaikkea pelissä voi ja pitää tehdä selviytyäkseen eri vaaroista, joita pelaajan eteen asetetaan. Pelin johdonmukaisuuden vuoksi päädyimme yhdistämään tutoriaalikentän pelin tarinaan. Kun Jack menettää housunsa ja tyttöystävänsä pelin alussa, hän lähtee etsimään apua ystävänsä Docin luota. Docin kanssa Jack käy dialogia, jossa pelaaja oppii näkemään huutomerkit paikoissa, jossa pelaaja voi olla vuorovaikutuksessa ympäristön kanssa. Kentän tapahtumapaikkana toimi Docin maatila.

Docille haluttiin luoda perinteisen laboratorion sijasta maatila, jonka hän on muuttanut omaksi keksijänverstaakseen. Halutusta maatilan visuaalisesta ilmeestä piirrettiin luonnoksia, jonka jälkeen referenssejä haettiin internetistä ja tv-sarjoista. Sopiva referenssikuva löytyi tv-sarjasta Smallville (katso kuvio 25).

KUVIO 25. Referenssikuva, jota käytettiin Docin maatalon suunnittelussa (FANDOM 2016, viitattu 22.4.2019

Maatilan edustalle tehtiin peliin kohta, jossa Doc oleskeli työpöytänsä takana. Docin vierelle tehtiin koneita, joita hän oli keksinyt (katso kuvio 26). Maatilan taustalle asetettiin gradientti taivaaksi ja maahan piirrettiin kaktuksia ja puussa oleva auto, joka liittyi tarinankerrontaan.

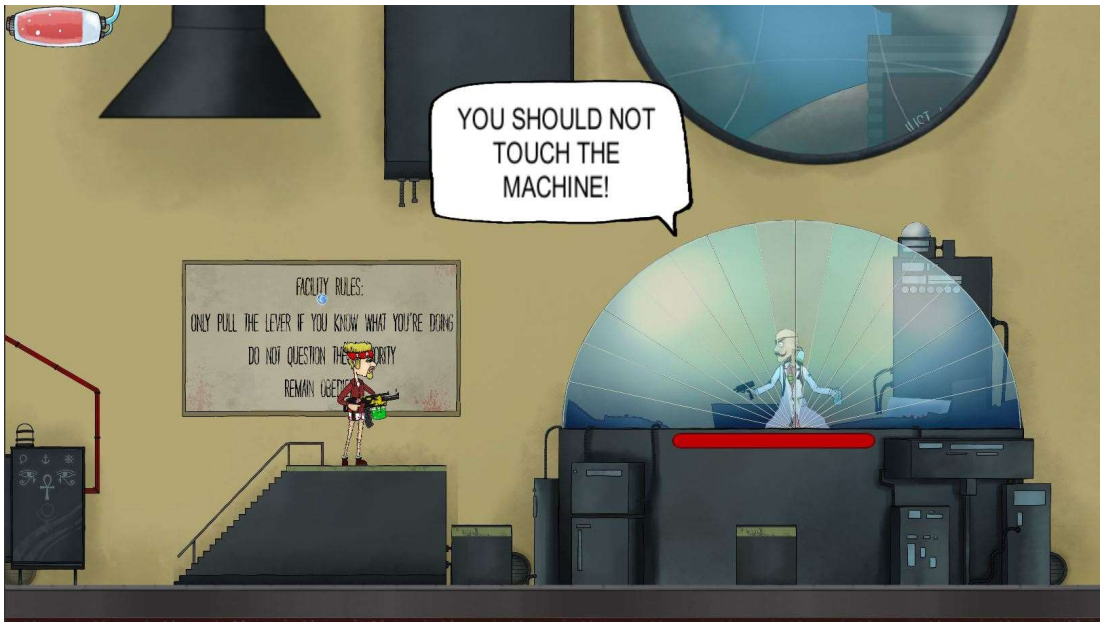


KUVIO 26. Docin maatalo.

6.1.3 Loppuvastuksen huone

Pelissämme tarina huipentuu Jack Stonen ja pahan tekoälyn kohtaamiseen. Loppuvastuksen huonetta luodessa keskityimme aluksi mekaniikkoihin. Onnistuneen taistelun kannalta oli tärkeä hahmottaa miten itse taistelu tapahtuisi, mitä kykyjä pahalla tekoälyllä on ja miten vastuksen voisi päihittää. Suunnittelun jälkeen oli selvää, että halusimme luoda taisteluun mekaniikan, joka sisälsi ongelmanratkomista.

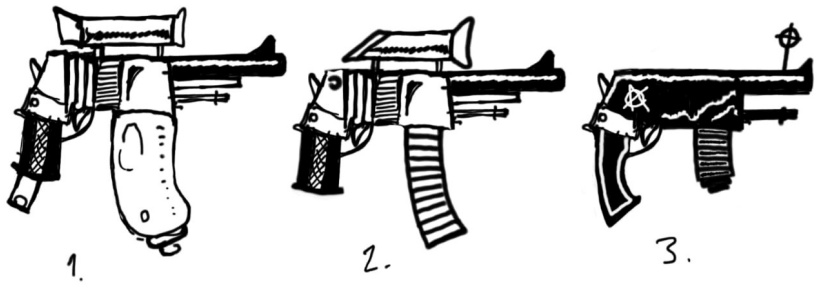
Taistelu tekoälyä vastaan alkaa pelaajan kävellessä tasanteelle, jonka vieressä tekoäly on lasikuvun alla piilossa (katso kuvio 27). Huoneessa on myös pyöreä kone sekä vipuja (katso kuvio 28). Tekoäly huutelee lausahduksia, joiden perusteella pelaajan täytyy päätellä, mihin järjestykseen elementit pyörivässä koneessa asetellaan koneen vieressä olevia vipuja vääntämällä. Kun elementit ovat halutussa kohdassa, pelaaja vääntää koneen oikealla puolella olevaa vipua. Jos säädöt olivat oikein, vihollisen suojana oleva lasikupu avautuu ja viholliseen voi tehdä vahinkoa. Jos elementit eivät ole oikein aseteltu suhteessa tekoälyn lausahduksiin, yllä olevasta putkesta tippuu vihollisia pelaajan kimppuun. Jotta taistelu ei olisi liian helppo, tietyin väliajoin huoneen lattia täyttyy myrkkykaasusta ja pelaajan täytyy nousta laatikoiden päälle turvaan myrkyltä.



KUVIO 27. Jack Stonen ja Tekoälyn kohtaaminen

KUVIO 28. Symbolikone, jota pelaaja joutuu käyttämään taistellessan Tekoälyä vastaan

6.1.4 Aseiden suunnitteluprosessi



KUVIO 29. Aseita varten piirrettiin useita luonnoksia.

Peliin suunniteltiin mekaniikkoja, jotka vaativat erilaisten aseiden käytön. Tämän vuoksi peliin tehtiin jääase sekä sitä varten tarvittavat animaatiot. Jäädytysasetta suunnitellessa käytettiin piirtolehtiötä. Useista nopeista vedoksista valikoitui yksi ja se vietiin Adobe Photoshop -ohjelmaan. Värjäsin kuvan aluksi käyttäen samaa palettia, kuin mitä Jackin ensimmäiseen aseeseen oli käytetty. Kun aseella oli pohjaväri, säädettiin Color Balancen kautta väriä kohti violettia ja sinistä. Kun ase oli halutun värinen, oli aika viimeistellä se yksityiskohdilla. Jääpuikkoja lisättiin kohtiin, joista kylmää pääsisi karkaamaan aseen rakenteista: piipun päähän, saumoihin ja letkujen liittymäkohtiin (katso kuvio 30).

KUVIO 30. Jäädytysaseen piirtämisen vaiheet

Pelinmekaniikkoihin kuuluu erilaisten aseiden käyttämistä ongelmanratkaisutarkoituksiin. Tämän vuoksi aseiden ulkonäön on hyvä tukea aseen käyttötarkoitusta pelissä.

6.2 Hahmosuunnittelu

Hahmosuunnittelu alkoi sillä, että hahmosta tehtiin karkea konsepti. Tämän jälkeen hahmo tuotiin Adobe Photoshop -ohjelmaan ja tehtiin ensimmäinen hahmon vedos (katso kuvio 31).

KUVIO 31. Ensimmäinen luonnos Jack Stonesta

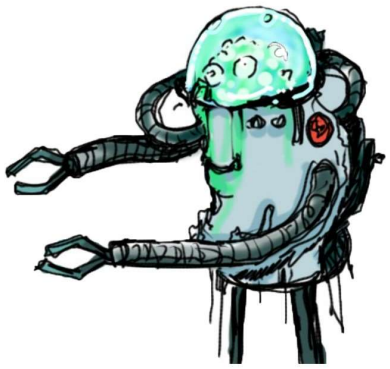
Jack Stone pysyi pitkälti muuttumattomana ensimmäisen vedoksen jälkeen. Päähahmon ulkoasulla haettiin stereotyyppisen Hillbillyn ja Rambon risteytystä. Ajatuksena päähenkilön takana oli saada hänet näyttämään toiminnan mieheltä, säilyttäen samalla komiikan tarinassa (katso kuvio 32).

KUVIO 32. Valmis Jack Stone

Hahmonsuunnittelussa käytettiin usein pikkukuva tekniikkaa, jossa hahmosta piirretään useita karkeita vedoksia ja valitaan niistä piirteet, joita hahmolle halutaan, jonka jälkeen hahmon yleisilmettä aletaan muodostamaan (Concept Art Empire 2019, viitattu 22.4.2019). Pelimaailman kasvaessa kasvoi myös hahmoarsenaali.

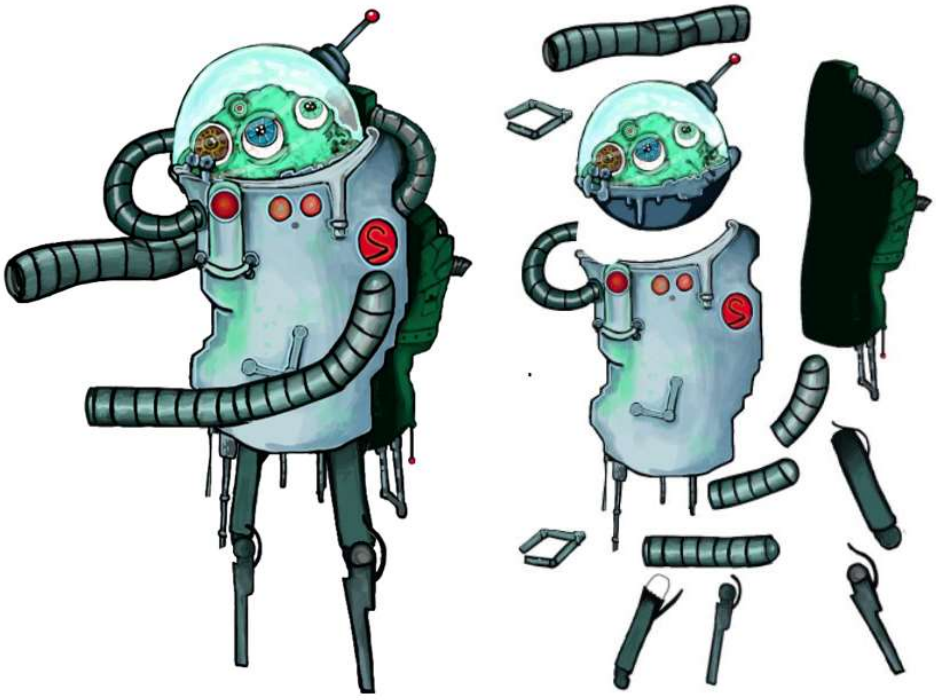
6.2.1 Robottiviholliset

Robottiviholliset oli yksi ensimmäisistä vihollishahmoista, jota peliämme varten piirrettiin (katso kuvio 33). Tarkoituksena oli piirtää mekaaninen otus, jolla oli myös biologisia elementtejä. Robottiviholliset hyökkää ampumalla metallisia osia putkista, jotka toimivat sen käsinä.



KUVIO 33. Luonnos Robottiviholliseista

Lopullinen kuva muistutti hyvin paljon luonnosta. Hahmolle luotiin syvyyttä laittamalla robottiviholliselle selkään reppu, joka liikkuu hahmon liikkuessa. Lopullisesta versiosta tehtiin myös irralliset spritet animointia varten (katso kuvio 34).



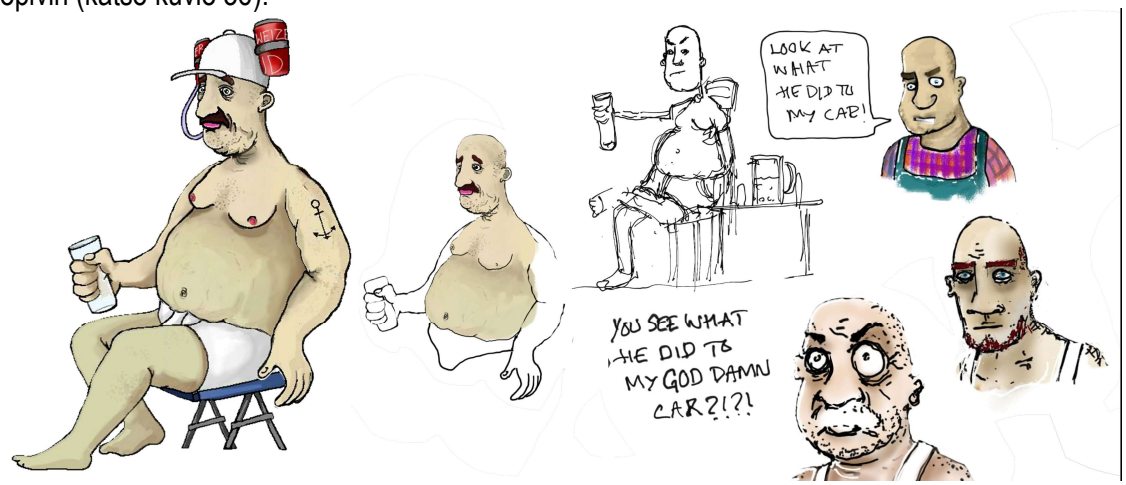
KUVIO 34. Robottiviholliset valmiina

6.2.2 Muut hahmot

Doc on kovia kokenut, hajamielinen, mutta äärimmäisen viisas tiedemies sekä keksijä (katso kuvio 35). Doc auttaa Jackia tarjoamalla hänelle työkaluja ja aseita. Docin menneisyydestä ei mainittu pelissä muuten, kuin kirjoitukset Docin maatilan varaston seinällä, kuten ”DO NOT TRAVEL IN TIME AGAIN!!! THEY ARE WATCHING!” kuvastavat että hän on kokenut mielenkiintoisia asioita menneisyydessään.

KUVIO 35. Jack Stonen ystävä Doc.

Hahmona naapuri toteutettiin sopimaan pelinsuunnittelussa tehtyihin ratkaisuihin. Tarvitsimme hahmon, joka piti Docin avaruusalusta lukkojen takana. Hahmo suunniteltiin kuten käsikirjoituksessa luki ja tämäkin hahmo, kuten kaikki pelin liikkuvat palaset, piirrettiin osina tulevia animaatioita varten. Luonnosteluvaiheessa hahmosta tehtiin useita nopeita luonnoksia ja niistä valikoitui peliin sopivin (katso kuvio 36).



KUVIO 36. Luonnoksia (oik.) ja viimeistelty naapuri (vas.)

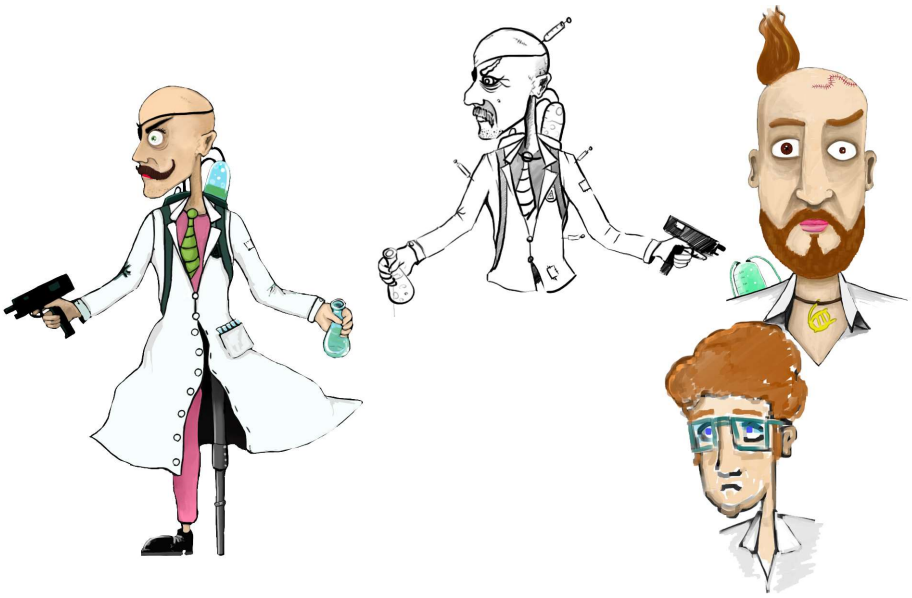
6.2.3 Anarkistit

Anarkistien tarkoitus pelissä oli tukea pelaajaa ongelmatilanteissa ja varoittaa mahdollisista vaaroista. Anarkistien väripaletteja suunnitellessa käytettiin maanläheisiä värejä sekä harmaansävyjä. Tarkoituksena oli tehdä anarkisteista helposti tunnistettavia, selkeästi samaan joukkoon kuuluvia hahmoja (katso kuvio 37). Anarkistit voi tunnistaa heidän pukeutumisestaan ja anarkistinmerkistä, joka tulee tutuksi pelaajalle varoitusviesteistä, joita hänelle on jätetty ympäri pelialuetta.

KUVIO 37. Anarkisteja

6.2.4 Tekoäly

Tekoälyn ihmismuotoa varten tehtiin luonnoksia erilaisista hahmoista ja kokeiltiin eri lähestymistapoja. Ensimmäinen ajatus oli tehdä tekoälyn ihmismuodosta kiltin näköinen, jopa sellainen, ettei pelaaja uskoisi sitä pahaksi. Halusimme kuitenkin pelin olevan helppolukuinen pelaajalle, ja koska hyvän ja pahan vastakkainasettelu oli alkanut jo pelin alusta, päädyimme pitämään teemaa johdonmukaisena suunnittelemalla päävastuksesta tavanomaisen pahiksen näköisen (katso kuvio 38).



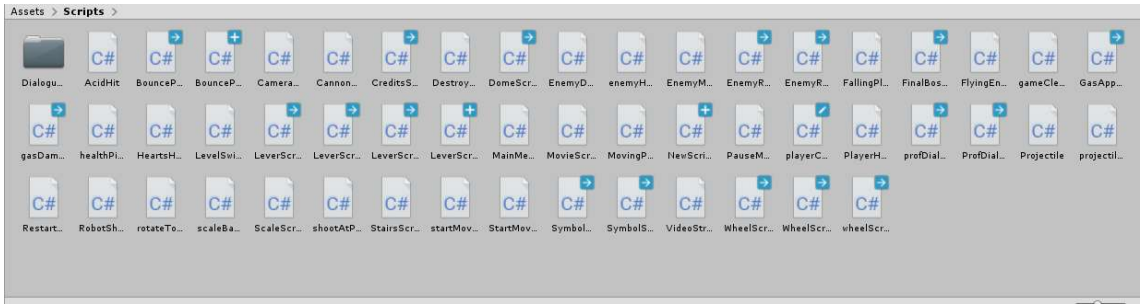
KUVIO 38. Valmis Tekoälyn ihmismuoto (vas.) ja luonnoksia, joita suunnitteluvaiheessa piirrettiin Tekoälyn ihmismuotoa varten (oik.)

Jerry Jenkins (2018, viitattu 22.4.2019) on listannut piirteitä, joita hyvällä vihollisella on hyvä olla. Jenkinsin mukaan vihollisen tulee olla vakuuttunut siitä, että hän on hyvä. Vihollisen tulee myös olla tarpeeksi varteenotettava vastus, jotta sankari näyttää hyvältä. Jenkins painottaa, että vihollisen tulee olla viisas ja tarpeeksi ansioitunut, saadakseen kunnioitusta muilta. Piirteiden listalla on myös häikäilemättömyys, jota vihollinen käyttää saadakseen haluamansa hinnalla millä hyvänsä, sekä ylpeys, petollisuus ja kostonhimo. Pelissämme nämä piirteet toteutuivat päävihollisen osalta selkeästi. Paha tekoäly oppii ihmiskunnan väkivaltaisen historian ja karkaa luodakseen paremman maailman. Hänen älykkyytensä on vailla vertaa ja hän ylpeilee saavutuksillaan. Tekoäly kaappaa Jack Stonen tyttöystävän häikäilemättömästi, luodakseen maailmaansa täydellisen naisen.

7 PELIN OHJELMOINTI

Pelin ohjelmoinnin aikana luotiin noin 50 skriptitiedostoa ja näihin liitettynä yli 5000 riviä koodia. Nämä määrittävät muun muassa mitä pelaaja voi tehdä, vihollisten sekä ansojen toimintoja pelimaailmassa ja käyttöliittymän rakentamista toiminnalliseksi kokonaisuudeksi. Kaikki tarkemmat toiminnot käyttäjän täytyy luoda itse, joten ohjelmointi on todella isossa roolissa pelinkehityksen aikana. Skriptien kautta käyttäjä pystyy toteuttamaan oman pelilogiikan ja käyttäytymisen yksinkertaisesti lisäämällä näitä ohjelmointitiedostoja suoraan peliobjekteihin kiinni. Skriptikomponenttien avulla on mahdollista tehdä paljon erilaisia toimintoja, kuten käynnistää pelitapahtumia, tarkistaa mahdolliset törmäykset mitä pelimaailmassa tapahtuu, lisätä ja soveltaa fysiikkaa ja vastata pelaajan painamiin syöttöihin näppäimistöllä tai ohjaimella.

Unity tukee C#-ohjelmointikieltä, joka on pelialalla yksi standardikielistä. Kyseisellä kielellä on jonkin verran samankaltaisuuksia Javan ja C++ kielien kanssa. C# on huomattavasti aloittelijaystävällisempi kieli verrattuna C++:aan, sillä se on niin sanottu ”hallittu kieli”. Tämä tarkoittaa sitä, että C# tekee automaattisesti muistinhallinnan käyttäjälle jakamalla muistia. Tämä suojaa muistivuodoilta ja muilta ongelmilta. Nykyään Unityllä on mahdollista ohjelmoida vain C#-ohjelmointikielen avulla. Käytimme työssä Visual Studio 2015 versiota koodaamisen apuna, vaikka Unitylla on mahdollista käyttää myös muita ohjelmointisovelluksia toiminnallisuuden luomisessa. Usein näillä ohjelmilla ei lopulta paljoa eroavaisuuksia löydy, mutta oman kokemuksen pohjalta oli huomattavasti helpompi käyttää Visual Studiota pelin ohjelmoinnin apuna. Tämän lisäksi Visual Studio neuvoo ja tarjoaa automaattisesti vaihtoehtoja erilaisiin koodaukseen liittyviin tilanteisiin hieman muita ohjelmia paremmin, mikä tehostaa myös työskentelyä ohjelmointiin liittyvissä kysymyksissä.



KUVIO 39. Pelissä käytettyjä skriptitiedostoja

7.1 Pelaajan perustoiminnot

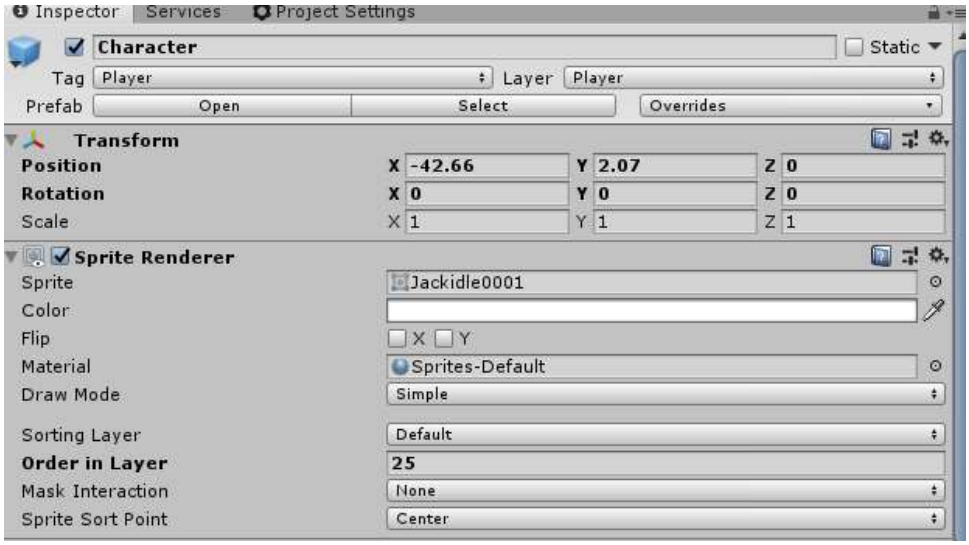
Tässä osiossa selitetään tarkemmin mitä toimintoja pelaajalla on ja kuinka ne vaikuttavat muuhun pelimaailmaan. Pelin kehityksen oleellisin osa onkin saada tehtyä monipuolinen ja hauska pelaajahahmo, koska tämän kautta ollaan vuorovaikutuksessa muun pelimaailman kanssa. PlayerCodes-skriptin tärkein tehtävä on vaikuttaa käyttäjän hallitsemaan peliobjektiin tämän antamien syötteiden avulla, jotka vaikuttavat moniin eri toimintoihin pelissä, kuten kävelemiseen, hyppäämiseen, ampumiseen ja vuorovaikutukseen muiden peliobjektien kanssa.

7.1.1 Pelaajahahmon luominen ja perusfysiikka

Pelin päähahmon tekeminen kannattaa aloittaa piirtämällä ja suunnittelemalla aluksi hahmo, jota maailmassa käytetään, ennen toiminnallisuuden tekemistä. Tämän hoitamiseen hyödynsimme pääosin graafikon taitoja, mutta kun Unityssä työskennellään, on tärkeää tietää kuinka näitä kuvia (Sprites) voidaan pelimaailmassa käyttää. Hahmon eri animaatio-osat jaoimme samaan tiedostoon, jotta niiden käyttö olisi mahdollisimman optimoitua samalla vähentäen rasitusta, jota pelin aikana voi tapahtua. Unity tarjoaa mahdollisuuden jakaa animaation eri vaiheet Sprite Editorin avulla, jossa samassa tiedostossa olevat kuvat on mahdollista jakaa useammaksi kappaleeksi (katso kuvio 40). Kuvien jakamisen jälkeen on helppo aloittaa pelaajahahmoon liittyvien toimintojen tekeminen, kuten yksinkertaisen tyhjäkäynnin (Idle) lisääminen animaation avulla tai tekemällä hieman monimutkaisempaa toimintaa, kuten kävelyä tai hyppäämistä, jotka vaativat sekä animaatioiden että ohjelmoinnin yhdistämistä Unityn avulla.

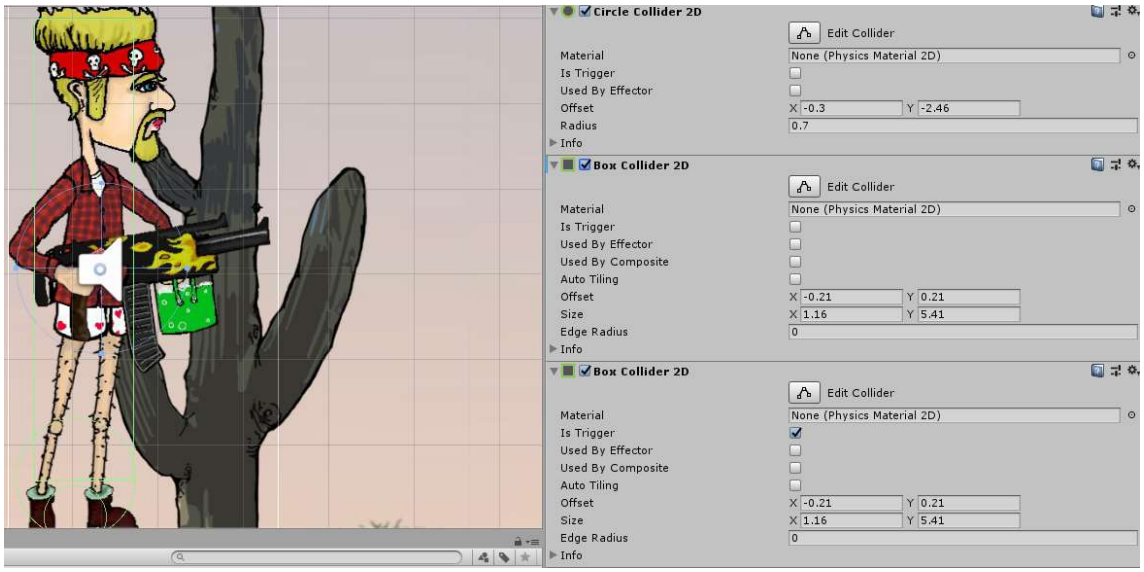
KUVIO 40. Pelaajahahmon hyppääminen jaettuna useampaan eri osaan SpriteEditorin avulla

Hahmo on helppo lisätä pelin päänäkymään raahaamalla yksi näistä kuvista suoraan pelimaailmaan tai lisäämällä se hierarkia ikkunaan, jolloin kuvan sijoittuminen on aina keskipisteessä sijainnin osalta (0,0,0). Hahmo kannattaa sijoittaa mahdollisimman korkealle kerrosjärjestyksen (Order in layer), sillä sitä kautta kyseinen objekti on kaikkien muiden asioiden etupuolella pelimaailmassa. Hahmolle on myös suositeltavaa laittaa suoraan tunniste (Tag) ja kerros (Layer) -Player nimikkeellä, sillä tätä kautta kyseinen pelaajahahmo on helpommin löydettävissä ja tunnistettavissa kun se on tarpeellista, joka on erityisen suuressa roolissa ohjelmoinnin kannalta jatkossa (katso kuvio 41). Näiden käyttö on suositeltavaa lähes kaikille objekteille, joita pelimaailmaan lisätään, hyvän käytännöllisyyden ja järjestelmällisen ylläpidon kannalta.



KUVIO 41. Pelaajahahmon peruskomponentit lisäämisvaiheessa ja kerrosjärjestyksen muuttaminen

Pelaajalle on lähes pakollista lisätä myös jäykkä kappale (Rigidbody2D), jotta kyseiseen objektiin vaikuttaa pelimaailmassa käytettävä fysiikka ja osuma-alue (Collider2D), jotta pelaaja on vuorovaikutuksessa myös muiden objektien kanssa maailmassa kuten lattian ja seinien, joihin on liitettynä myös jonkinlainen osuma-alue komponentti. Usein jalkojen juureen kannattaa laittaa ympyrän muotoinen osuma-alue komponentti (Circle Collider 2D), jotta pelaajan putoaminen on mahdollisimman luonnollista ja muuhun ruumiiseen neliönmuotoinen (Box Collider 2D), muuta interaktiota ja osumista varten pelimaailmassa. Näin tehtiin myös kyseisessä pelissä (katso kuvio 42). Näiden komponenttien avulla saadaan pelaajalle luotua jo perustason fyysiset ominaisuudet hyvin nopeasti. Tämän myötä komponentteihin liittyviä ominaisuuksia pystytään muokkaamaan helposti ja vaivattomasti myös jatkossa.



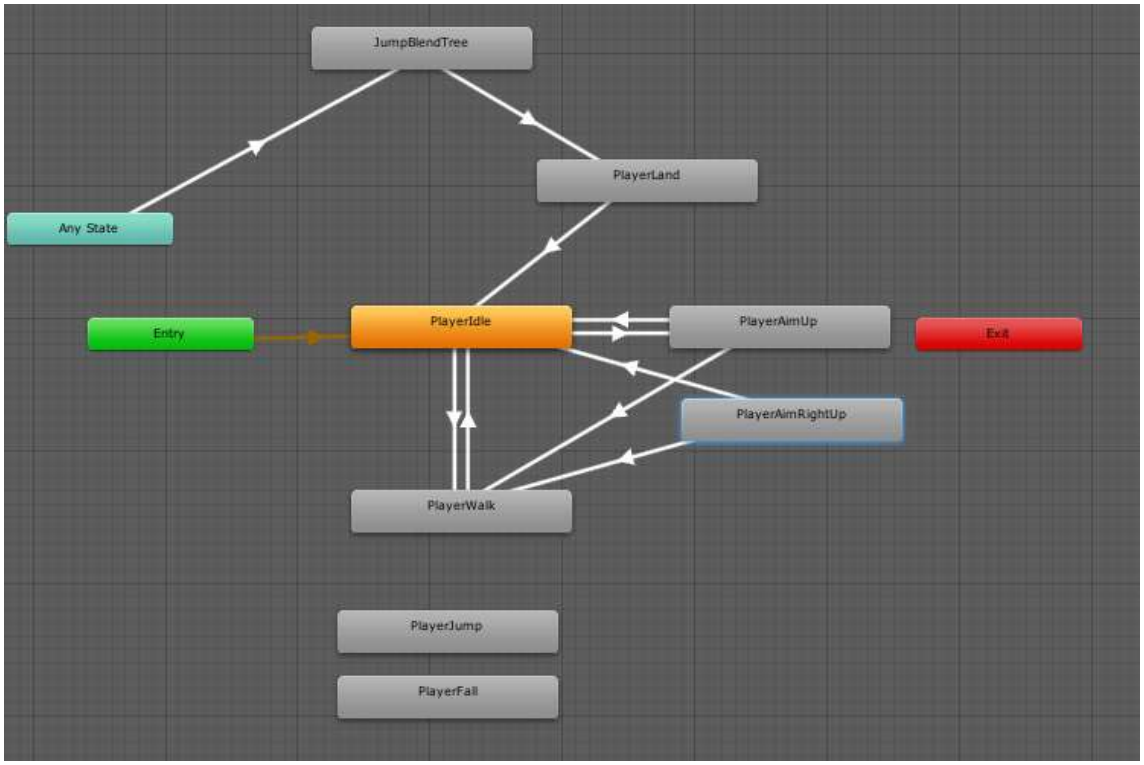
KUVIO 42. Pelaajaan liitetyt osuma-alue komponentit törmäyksiä ja kontakteja varten

7.1.2 Animaatioiden ja ohjelmoinnin yhdistäminen pelaajan liikkumisen osalta

Animaatioiden lisääminen hahmoille on yksinkertainen prosessi Unityssä. Kuten jo aiemmin mainitsimme, kuvien jakaminen samaan tiedostoon nopeuttaa animaatioiden tekemistä merkittävästi. Jos kuvista saa helposti selville, mistä animaatio alkaa ja mihin se loppuu, on näiden lisääminen eri objekteille todella helppo toteuttaa. Tätä mentaliteettia käytimme myös kyseissä työssä, sillä grafiikan ja ohjelmoinnin yhdistäminen tapahtuu lähinnä juuri animaatioiden kautta, joten oli todella oleellista työnkulun ja edistymisen kannalta, että ne oli jaettu mahdollisimman selkeiksi tiedostoiksi. Tämä tarjosi molemmille mahdollisuuden kokeilla ja muokata erilaisia animaatioita Unityn sisällä jatkuvasti.

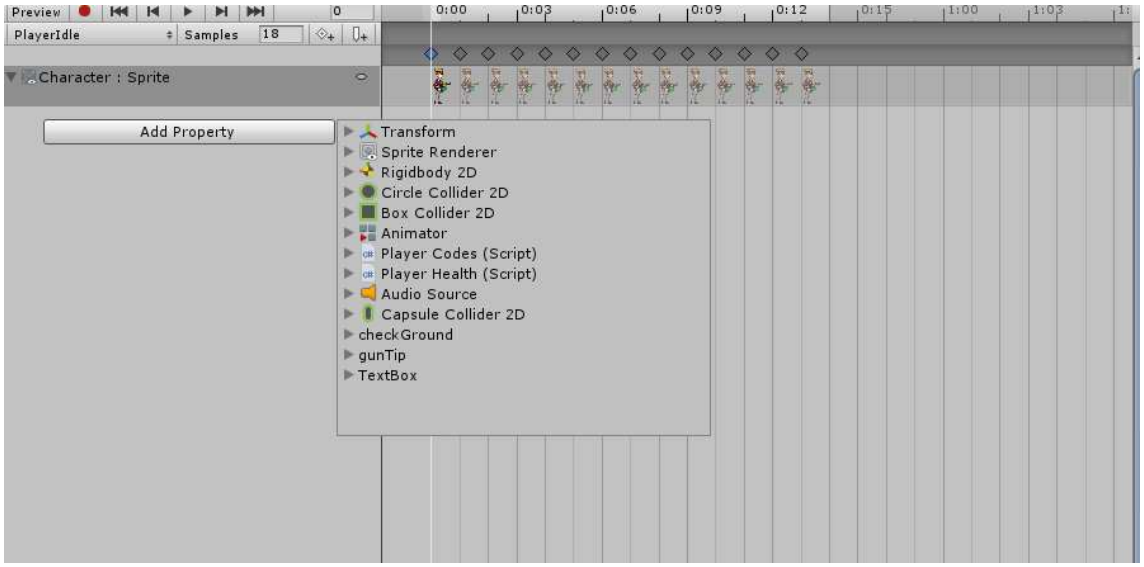
Kaikkien animaatio-osien valmistamisen ja leikkaamisen jälkeen Unityssä, ne voidaan vain korostaa ja liittää hiiren avulla hiearkia-ikkunassa olevaan pelaajahahmoon. Tällöin Unity luo automaattisesti kyseiselle objektille tiedoston, johon on liitettynä kaikki nämä animaation eri kuvat. Tämä luo myös pelaajalle suoraan animaattorikomponentin (Animator), johon on liitettynä suoraan juuri lisätty animaatio. Animaattori-ikkunaa avatessa, nähdään kyseisessä objektissa olevat animaatiot ja se toimii eräänlaisena karttana kaikille animaatioilla mitä tähän on liitetty tai voidaan lisätä. Animaatiot toimivat lähtökohtaisesti parametrien avulla, jotka ovat sitten liitoksissa ohjelmointitiedostoihin. Näiden parametrien kautta tapahtuu pelaajahahmon osalta siirtymä paikasta toiseen kuten

tyhjäkäynnistä kävelyyn, hypystä putoamiseen ja takaisin tyhjäkäyntiin tai muut vastaavat siirtymät (katso kuvio 43). Juoksuanimaatiota tehdessä täytyy animaattori-ikkunaan lisätä float-parametri, jota voidaan hyödyntää myöhemmin skriptitiedostossa ja katsoa kuinka nopeasti pelaaja liikkuu pelimaailmassa. Tämä nopeusparametri tutkii, onko pelaajalla minkäänlaista liikettä tai vauhtia kentässä ja vaihtaa sen mukaan tyhjäkäyntianimaatiosta juoksuanimaatioon tai vastaavasti myös toisinpäin.



KUVIO 43. Pelaajahahmon eri toiminnot animaattori-ikkunan sisällä

Pelaajahahmon tyhjäkäynti on perustason animaatio, joka pyörii aina kun käyttäjä ei tee pelissä mitään. Kaikkien animaatioiden nopeutta ja tietoja voidaan muuttaa animaattori-ikkunnassa tai menemällä animaatioikkunaan (Animations), jossa on mahdollista vaihtaa jokaisen yksittäisen kuvan paikkaa, pyörimistä tai muita arvoja omaan mieltymykseen sopiviksi ja nähdä miltä animaatiot näyttävät itse pelissä esikatselun (Preview) avulla (katso kuvio 44). Projektin edetessä nämä kyseiset ikkunat tulivat myös hyvin tutuiksi molemmille tekijöille.



KUVIO 44. Animaatioikkuna ja sen käyttöliittymä Unityssä

Animaatioiden ja parametrien lisäämisen jälkeen pelaajalle voidaan lisätä skriptitiedosto, jonka sisälle kaikki toiminnallisuus ohjelmoidaan ja lisätään, mukaan lukien myös liikkumiseen ja animaatioiden vaihtumiseen liittyvät prosessit ja tekijät. Tärkeintä ohjelmoinnin alussa on aina määritellä tarvittavat muuttujat, joita kyseisessä skriptitiedostossa hyödynnetään. Koska pelaajalle on saatava liikettä ja toiminnallisuutta, täytyy liukuluku (Float) muuttuja nimetä, johon liitetään pelaajan liikkumisen arvo. Kyseistä arvoa pystytään manipuloimaan määrittämällä muuttuja joko yksityiseksi, jolloin arvoa on mahdollista muuttaa vain skriptitiedoston sisällä tai julkiseksi, joka mahdollistaa arvon muokkaamisen myös Unityn sisällä tehokkaampaa testaamista varten. Liukuluvun lisäksi on pakollista löytää tarvittavat komponentit pelaajahahmosta fysiikkaan ja animaatioihin liittyvissä tekijöissä Void Start -toiminnon avulla (katso kuvio 45).

```
public float move;
public float maxSpeed;

public Rigidbody2D playerRB;
public Animator playerAnim;
// Use this for initialization
void Start () {
    playerRB = GetComponent<Rigidbody2D>();
    playerAnim = GetComponent<Animator>();
```

KUVIO 45. Muuttujien määrittely ja niiden löytäminen Start:n sisällä

Tarvittavien muuttujien lisäämisen jälkeen itse ohjelmointi liikkumisen osalta pystytään aloittamaan. Aluksi liikkumisen kannalta on hyvä määritellä, onko pelaaja painanut mitään nappia näppäimistöllä

tai ohjaimella, joka vaikuttaa akseliin (axis) ja sitä kautta liikkumismuuttujaan, johon tulee joko positiivinen tai negatiivinen arvo sen mukaan mitä nappia on painettu. Kyseisessä pelissä hyödynsimme kiihtyvyysvektoria toteuttamaan liikkuvuuden pelaajalle, vaikka kyseisen asian voisi toteuttaa muillakin tavoilla, kuten lisäämällä voimaa jäykkään objektiin suoraan. Ohjelmointitiedoston tarkistaessa onko jotain nappia painettu, voidaan kyseinen asia lisätä myös animaation vaihtumiseen, hyödyntämällä aiemmin sinne lisättyä liukulukuparametriä. Pelaajan painaessa oikealle, liikkuu maailmassa oleva pelaajahahmo sen mukaisesti oikeaan suuntaan ja animaatio muuttuu liikkumisen yhteydessä samalla juoksuksi. Toiseen suuntaan painaessa, hahmon horisontaalinen kokoasteikko vaihtaa suuntaa 180 astetta Flip()-funktion avulla, joka kääntää pelaajan maailmassa toisinpäin, samalla liikuttaen sitä vastakkaiseen suuntaan. Täten hahmon liikkuminen ja animaatiot saadaan yksinkertaisesti toteutettua Unityssä molempiin suuntiin, hyvin kevyellä ja helposti muokattavissa olevalla skriptitiedostolla (katso kuvio 46).

```
        move = Input.GetAxis("Horizontal");
        playerAnim.SetFloat("speed", Mathf.Abs(move));

        playerRB.velocity = new Vector2(move * maxSpeed, playerRB.velocity.y);

        if (move > 0&&!facingRight)
        {
            flip();
        }
        else if (move < 0 && facingRight) {
            flip();
        }

    }


    void flip ()
    {
        facingRight = !facingRight;
        Vector3 theScale = transform.localScale;
        theScale.x *= -1;
        transform.localScale = theScale;
```

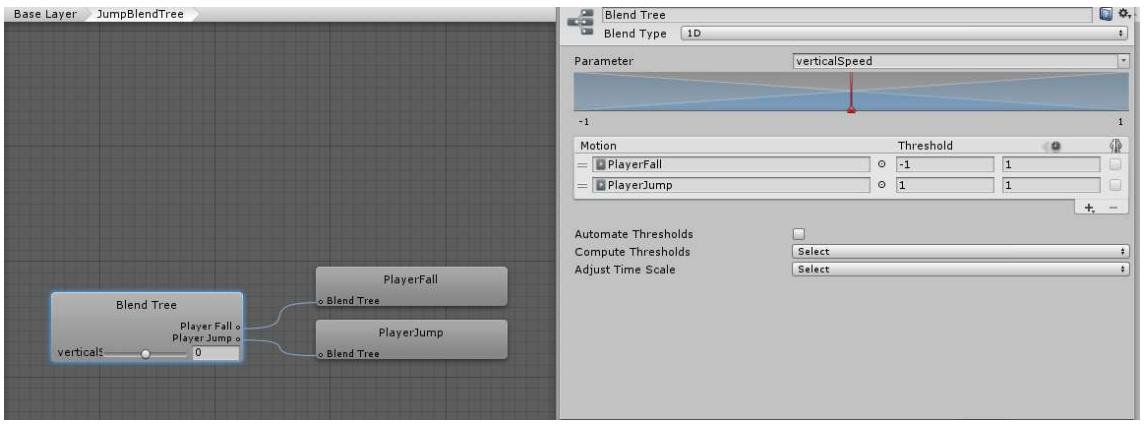
KUVIO 46. Pelaajan liikkumisen koodi kokonaisuudessaan

7.1.3 Hyppyfysiikoiden ja -animaatioiden lisääminen pelaajahahmolle

Vaivattomin tapa hypyn toteuttamisen kannalta, on tarkistaa, onko pelaaja maassa vai ilmassa ohjelmoinnin avulla. Aluksi on hyvä määritellä, minkä objektien päältä pelaaja pystyy hyppäämään

hyödyntämällä kerroksia (Layers) kaikissa tasanteissa ja lisäämällä niille osuma-alue komponentti, joiden päältä hyppääminen on mahdollista. Nimeämällä jokainen tasanne maakerroksen mukaisesti (Ground), uusien objektien lisääminen maailmaan, joiden päältä pelaaja voi hypätä, on jatkossa myös hyvin helppo prosessi. Kyseisille objekteille ei ole tarvetta lisätä yhtään ohjelmointitiedostoa sillä kaikki ohjelmointi hypyn osalta tapahtuu jo aiemmin pelaajan liitetyssä skriptitiedostossa. Tämän lisäksi hahmolle tulee lisätä jalkojen juureen lapsiobjekti, joka tarkistaa skriptitiedoston avulla jatkuvasti onko hahmo kontaktissa maan kanssa.

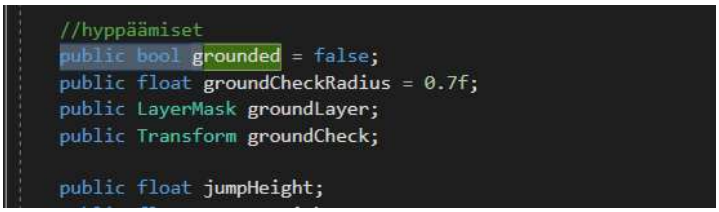
Animaatioikkunassa lisätään aluksi hyppyanimaation eri ajankohdat, jotka jo aiemmin käytiin läpi grafiikan osalta, mutta tässä osiossa myös tarkemmin käytännön kautta. Hyppyanimaatiossa on 3 eri mahdollista vaihetta, jotka vaikuttavat animaation vaihtumiseen pelimaailmassa, joihin kuuluu hyppäämisen aloitus, kun pelaaja on pystysuuntaisessa liikkeessä ylöspäin, putoamisvaihe kun tämä pystysuuntainen liike vaihtuu negatiiviseksi arvoksi alaspäin tippuessa ja laskeutumisvaihe kun pelaaja laskeutuu takaisin jollekin maakerroksen alustalle. Tätä varten oli lisättävä myös totuusarvoa (bool) tutkiva parametri, jota kautta varmistetaan, onko pelaaja maassa vai ilmassa. Kyseistä animaatiota tehdessä hyödynsimme ja kokeilimme Unityn tarjoamaa blend tree -tekniikkaa, joka seuraa kyseisessä tilanteessa pelaajan pystysuuntaista liikettä vertikaalinopeusparametrin kautta ja vaihtaa sen mukaisesti animaatiosta toiseen (katso kuvio 47). Tämän tekniikan avulla on mahdollista yhdistää useita animaatioita yhteen, luoden ja saavuttaen mukautettuja tuloksia.



KUVIO 47. Blend tree -tekniikan käyttämistä pelaajan hyppyanimaation vaihtumisen tukena

Fysiikoiden, parametrien ja animaatioiden lisäämisen jälkeen Unityn sisällä, täytyy itse toiminnallisuus saada aikaan pelaajan skriptitiedostossa. Kuten juoksuanimaation kanssa, aluksi on tärkeää määritellä tarvittavat parametrit, joita hyppyanimaatio vaatii. Erityisen tutuiksi peliä tehdessä tuli

liukulukujen, totuusarvojen ja eri komponenttien löytäminen tietyistä peliobjekteista ja näitä parametrejä hyödynsimme myös hyppyyn liittyvissä tekijöissä. Näihin liittyy mm. totuusarvon tutkimista sille, onko pelaaja maassa, kerrosmaskin määrittämistä ja hyppykorkeuden liukuluvun lisääminen (katso kuvio 48). Jatkossa raportissamme ei enää tutkita parametrien ja eri arvojen määrittämistä sen tarkemmin vaan keskitytään suoraan ohjelmoinnin toteutukseen eri objekteissa ja näiden toiminnallisuuden käsittelyyn.



KUVIO 48. Parametrit ja arvot, joiden avulla hypyn ohjelmointi toteutetaan

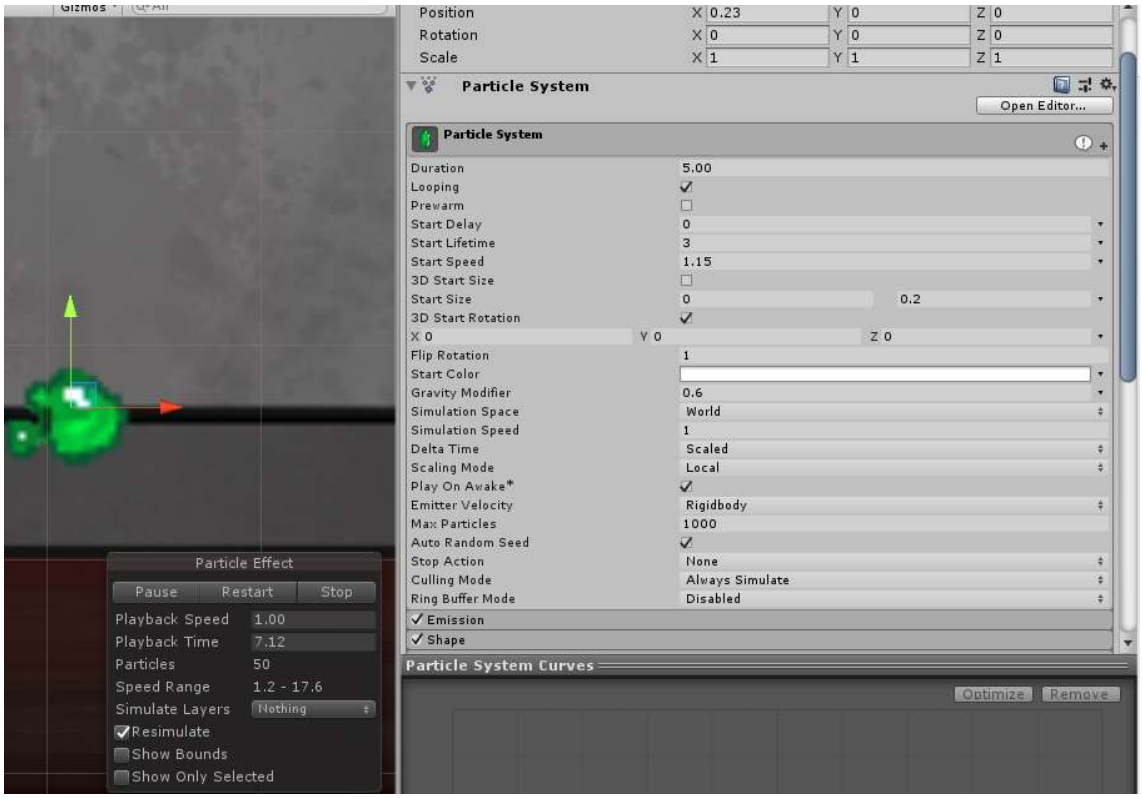
Aluksi lisäsimme jos-lauseen (If-statement), jota kautta katsotaan, onko hahmo maassa ja painetaanko pelin syöttöasetuksissa automaattisesti määriteltyä hyppynappia akselin kautta. Samalla jalkojen juuressa oleva lapsiobjekti yrittää havaita jatkuvasti onko pelaaja maassa vai ei. Jos näin tapahtuu, niin pelaajalle lisätään hyppyvoimaa vaakasuoran kiihtyvyysvektorin avulla, joka aiheuttaa pelaajalle hyppäämisen fysikaalisen tapahtuman. Tämän kautta animaattori-ikkunassa olevat totuusarvo muuttuu epätodeksi ja pystysuora nopeusparametriarvo vaihtuu, joka sitten muuttaa pelaajahahmon animaatiokuvaa, riippuen siitä onko tämä pystysuora arvo positiivinen vai negatiivinen. Painovoima vaikuttaa jäykkään objektiin, joka pelaajaan on liitetty. Tämän vuoksi pelaaja putoaa alaspäin pelimaailmassa niin kauan, kunnes se löytää jonkinlaisen alustan minkä päälle on mahdollista laskeutua (katso kuvio 49).

```
// Update is called once per frame
void FixedUpdate () {
    if (grounded && Input.GetAxisRaw("Jump") != 0 && canJump == true)
    {
        grounded = false;
        playerAnim.SetBool("isGrounded", grounded);
        playerRB.velocity = new Vector2(0, jumpHeight);

    }
    grounded = Physics2D.OverlapCircle(groundCheck.position, groundCheckRadius, groundLayer);
    playerAnim.SetBool("isGrounded", grounded);

    playerAnim.SetFloat("verticalSpeed", playerRB.velocity.y);
```

KUVIO 49. Hypyn toiminnallisuuden tekeminen ja toteutus FixedUpdate:n sisällä

7.1.4 Pelaajan ampuminen

Ammunnan toiminnallisuutta luodessa on hyvä aluksi pohtia, kuinka tämä kannattaa toteuttaa, millaisia ammuksia pelaajalla on ja miten niiden fysiikka toimii pelimaailmassa. Kun suunnitteluvaihe on selkeä, luodaan aluksi itse ammus pelimaailmassa lisäämällä hiirtä raahaamalla panoksen kuva pelimaailmaan ja lisäämällä siihen sille oleellisia komponentteja. Fysiikat ja osuminen ovat tärkeässä roolissa panoksen liikkumisen kannalta, joten siihen tuli liittää jälleen jäykkä objekti sekä osuma-alue komponentti. Näiden lisäksi lisäsimme ammukseen hiukkassysteemin (Particle system), joka käyttää sille annettua materiaalia apunaan luomaan luonnollisemman näköisen efektin ampuessa (katso Kuvio 50). Hiukkassysteemi tiputtaa sille annettujen arvojen mukaisesti erilaisia partikkeleita maailmassa ja tätä hyödyntäen saimme luotua panokselle hieman elävyyttä sen liikkuessa pelimaailmassa luomisen jälkeen. Hiukkassysteemien luominen on Unityssä myös todella laaja ja monipuolinen ominaisuus, johon emme raportissamme sen suuremmin keskittyneet.



KUVIO 50. Unityn hiukkasysteemi, jota hyödynnettiin panoksen luomisessa

Ohjelmoinnin osalta panoksen toiminnallisuus oli suhteellisen helppo toteuttaa, kun kaikki komponentit oli siihen liitettynä. Kun panos liikkuu maailmassa, liikkuu se AddForce-toiminnon avulla jatkuvasti voimavektorin suuntaan, lisäten jatkuvan impulsiivisen voiman jäykkään objektiin, joka ammukseen on liitetty. Kyseistä ohjelmointitiedostoa luodessa oli myös tärkeää, että kun pelaaja kääntyy ympäri tai tähtää taivaalle, antaa skriptitiedosto voimaa aina oikeaan suuntaan ja panoksen lentorata muuttuu sen mukaisesti. Pelaajan painaessa ampumisnappia pelissä, synnyttää se Instantiate-funktion avulla kyseisen objektin kloonin pelimaailmaan sille määritellylle pisteelle, joka on tässä tapauksessa pelaajan aseen päässä. Kyseistä funktiota tuli käytettyä pelin luomisen aikana useita kertoja, sillä se hyödyntää valmisobjekteja (Prefab) ja luo ne klooneina sitä kautta pelimaailmaan kuten myös pelaajan ampuessa. Ammuksen luontivaiheessa katsoo FireAcid-funktio, milloin pelaaja voi seuraavan kerran ampua ja mihin suuntaan pelaaja katsoo, jotta panoksen kuva liikkuu ja on kääntyneenä varmasti oikeaan suuntaan pelimaailmassa (katso kuviot 51 ja 52).

```
    if(transform.localRotation.z>0)
    {
        if (thePlayer.aimingUp == true && thePlayer.move == 0)
        {
            projectileRB.AddForce(new Vector2(0, 1) * acidSpeed, ForceMode2D.Impulse);
            transform.Rotate(0, 0, -90);
        }
        else
        {
            projectileRB.AddForce(new Vector2(-1, 0) * acidSpeed, ForceMode2D.Impulse);
            //transform.Rotate(0, 0, 90);
        }
    }
    else
    {
        if (thePlayer.aimingUp == true && thePlayer.move == 0)
        {
            projectileRB.AddForce(new Vector2(0, 1) * acidSpeed, ForceMode2D.Impulse);
            transform.Rotate(0, 0, 90);
        }
        else
        {
            projectileRB.AddForce(new Vector2(1, 0) * acidSpeed, ForceMode2D.Impulse);
            //transform.Rotate(0, 0, 90);
        }

    }
}
```

KUVIO 51. Ammuksen liikkuminen pelaajan tähtäyksen mukaisesti

```
        //pelaajan ampuminen
        if (Input.GetAxis("Fire1") > 0)
        {

            fireAcid();
        }
    }

    void fireAcid()
    {
        if(Time.time > nextFire)
        {
            nextFire = Time.time + rateOfFire;
            if(facingRight)
            {
                Instantiate(acidShot, gunTip.position, Quaternion.Euler(new Vector3(0,0,0)));
            }
            else if (!facingRight)
            {
                Instantiate(acidShot, gunTip.position, Quaternion.Euler(new Vector3(0, 0, 180)));
            }
        }
    }
```

KUVIO 52. FireAcid-funktio, joka luo pelaajan ampuman panoksen pelimaailmaan

Panokseen täytyi luoda oma skripti myös osumista varten, joka varmistaa mahdollisen kontaktin muiden objektien kanssa pelimaailmassa (katso kuvio 53). Tässä vaiheessa oli erittäin hyödyllistä lisätä kaikkiin objekteihin Layer-nimike (”Shootable”), sillä kyseinen tiedosto varmistaa ja tutkii jatkuvasti, onko se kontaktissa muiden tätä nimikettä omaavien kanssa. Jos näin käy, tuhoaa tiedosto panoksen pelimaailmasta ja luo jälleen Instantiate-funktiota hyödyntämällä räjähdysefektin tähän osumispisteeseen. Samalla skripti varmistaa onko osumispisteellä vihollistunnistetta (tag), vähentäen tältä elämäpisteitä, jotka on sille määritelty toisessa tiedostossa.

```
void OnTriggerEnter2D(Collider2D other)
{
    if (other.gameObject.layer == LayerMask.NameToLayer("Shootable"))
    {
        myPC.removeForce();
        Instantiate(explosionEffect, transform.position, transform.rotation);
        projectile.SetBool("acidHit", acidHit = true);

        if (other.tag == "Enemy")
        {
            enemyHealth hurtEnemy = other.gameObject.GetComponent<enemyHealth>();
            hurtEnemy.addDamage(weaponDamage);
        }
    }
}
```

KUVIO 53. Panoksen toiminnallisuus osuessa toiseen peliobjektiin

Myöhemmässä vaiheessa käymme läpi tarkemmin, kuinka tämä panos vaikuttaa vihollisten elämäpisteisiin, mutta tässä vaiheessa on hyvä, että kaikki toiminnallisuus pelaajan ampumisen ja panoksen osalta on kuitenkin luotu seuraavia vaiheita varten. Lisäksi kyseisen panoksen ohjelmointitiedostoa oli helppo käyttää ja soveltaa myös muissa ammuksissa, mitä pelimaailmassa luotiin, kuten esimerkiksi vihollisisten ammuksien luomisvaiheessa.

7.2 Viholliset ja ansat pelissä

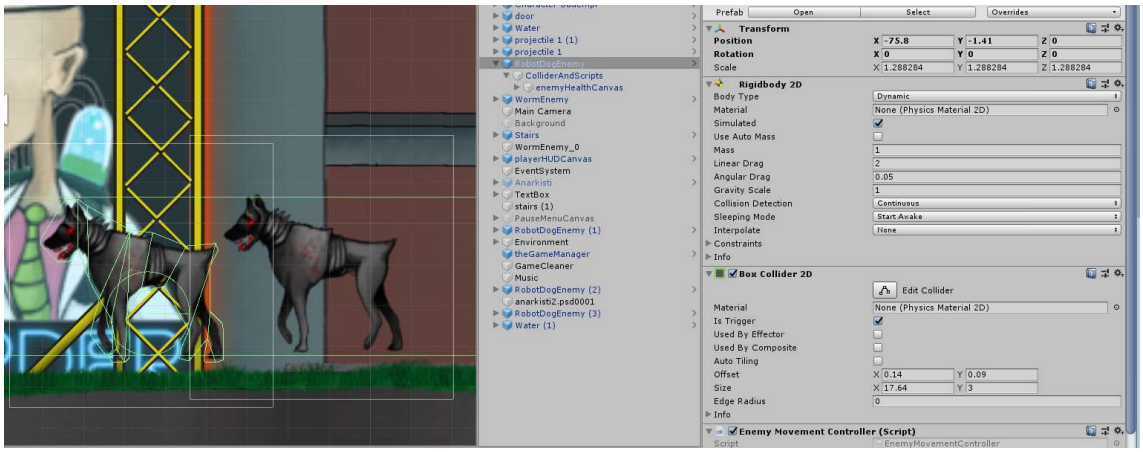
Tässä osiossa käymme läpi tarkemmin erilaisia vihollisia, joita pelimaailmaan loimme pelin edetessä. Vihollisten toiminnallisuuden toteutus ohjelmoinnin kannalta oli erityisen haastava prosessi, sillä näiden tekemiseen vaadittiin useampaa ohjelmointitiedostoa, joiden aktiivisuus ja tapahtumat olivat riippuvaisia monista eri tekijöistä pelimaailmassa. Kaikkea toiminnallisuutta ei ole suositeltavaa ohjelmoinnissa laittaa samaan tiedostoon, sillä kyseiseen tiedostoon on helppo tarvittaessa palata takaisin ja muokata sitä, jonka lisäksi myös muut ihmiset pystyvät helposti navigoimaan ja

ymmärtämään tätä skriptitiedostoa. Tämän takia esimerkiksi vihollisten luomisen kannalta oli oleellista jakaa liikkumisen toteutus, vahingonantaminen ja elämään liittyvät tekijät eri skriptikomponentteihin.

Pienetkin virheet ja epäkohdat ohjelmakoodissa vaikuttivat vihollisten toiminnallisuuteen merkittävästi, joten näiden tekemiseen kului projektissa runsaasti aikaa. Sen lisäksi että ohjelmointi oli haastavaa vihollisten toteutuksessa myös graafikon taidot olivat koetuksella erityisesti vihollisten liikkumista tehdessä, sillä siirtymien ja liikkumisen luonnollinen toteuttaminen Unityssä ja pelin sisällä tarjosi runsaasti ongelmia. Pyrimme luomaan kuitenkin mahdollisimman hyviä ja kiinnostavan tuntuisia vihollisia, joiden pääsääntöinen tehtävä on häiritä ja estää pelaajan liikkumista pelimaailmassa, tehden pelattavuudesta haastavan, mutta silti hauskan kokemuksen.

7.2.1 Viholliskoiran toteutus

Aluksi peliin suunniteltiin ja tehtiin viholliskoira, joka reagoi pelaajahahmoon hyökkäämällä tämän päälle, jos pelaaja tulee tarpeeksi läheisen etäisyyden päähän kyseistä vihollista. Jotta viholliselle on mahdollista saada minkäänlaista toiminnallisuutta aikaan, täytyy tällekin lisätä Rigidbody2D-komponentti sekä osuma-alue komponentti, joiden kautta fysiikka ja ohjelmointiasiat tapahtuvat (katso kuvio 54). Näiden komponenttien lisäämisen jälkeen on tärkeää lisätä ja katsoa miltä viholliskoiran tyhjäkäynti ja hyökkäysanimaatiot näyttävät animaattori-ikkunassa. Kuten jo aiemmin pelaajahahmon kohdalla, koiralle tehdyt animaatio-osat jaettiin samaan tiedostoon kuvankäsittelyohjelman avulla, jonka jälkeen ne leikattiin Unityssä ja lisättiin suoraan näkymässä olevaan koiraobjektiin tarkempaa animaation testausta, kokeilua ja tutkimista varten. Vaikka animaatio itsessään oli haastava toteuttaa, ei kyseisellä vihollisella ollut muita animaatiosiirtymiä kuin tyhjäkäynnistä suoraan juoksemiseen. Tämän takia animaattoripohja oli hyvin tyhjä siirtymien osalta ja siihen liitettiin lisäksi vain yksi parametri, joka katsoo kyseiden animaatioiden vaihtumista totuusarvon mukaisesti. Näiden lisäyksien jälkeen, oli helppo siirtyä itse ohjelmointiin vihollisen hyökkäämisen toteutuksen osalta.



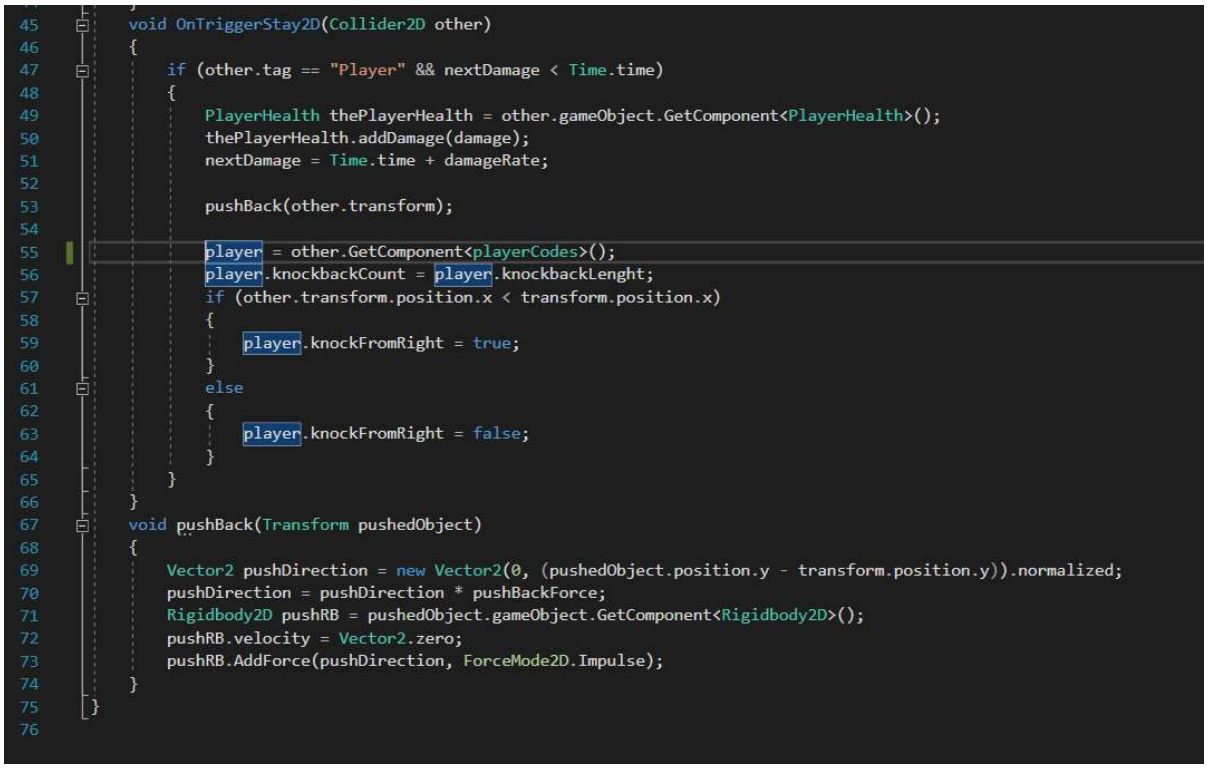
KUVIO 54. Robottikoira pelimaailmassa ja siihen liitettyjä komponentteja inspector-ikkunassa

```
// Update is called once per frame
void Update () {
    if (Time.time >= nextFlipChance)
    {
        if (Random.Range(1,10) > 5) flipFacing();
        nextFlipChance = Time.time + flipTime;
    }
}
void OnTriggerStay2D(Collider2D other)
{
    if(other.tag == "Player")
    {
        if(startChargeTime < Time.time)
        {
            if(!facingRight)
            {
                enemyRB.AddForce(new Vector2(-1, 0) * enemySpeed);
            }
            else
            {
                enemyRB.AddForce(new Vector2(1, 0) * enemySpeed);
            }
            enemyAnimator.SetBool("isCharging", charging);
        }
    }
}
void OnTriggerExit2D(Collider2D other)
{
    if (other.tag == "Player")
    {
        enemyCanFlip = true;
        charging = false;
        enemyRB.velocity = new Vector2(0f, 0f);
        enemyAnimator.SetBool("isCharging", charging);
    }
}
void flipFacing()
{
    if (!enemyCanFlip) return;
        float facingX = enemyGraphic.transform.localScale.x;
        facingX *= -1f;
        enemyGraphic.transform.localScale = new Vector3(facingX, enemyGraphic.transform.localScale.y, enemyGraphic.transform.localScale.z);
        facingRight = !facingRight;
}
}
```

KUVIO 55. Vihollisкoiran liitetty skriptitiedosto, jota kautta se hyökkää pelaajan kimppuun

Yllä oleva kuvio 55 ohjelmointitiedostosta esittää kuinka hyökkäys pelaajan kimppuun tapahtuu ohjelmoinnin osalta. Kun pelaaja saapuu ja pysyy viholliskoiran määrittämällä osuma-aluekomponentin alueelle (OnTriggerEnter2D), varmistaa ohjelmointitiedosto onko pelissä kulunut aika enemmän kuin koiralle määritelty hyökkäysaika sekä sen, onko vihollinen oikeassa suunnassa pelaajaan nähden. Jos molemmat nämä asiat ovat toteutuneet, lisää skripti viholliselle nopeutta x-akselilla, jota kautta hyökkäys pelaaja kohti tapahtuu. Samalla animaattori-ikkunassa oleva parametri muuttuu todeksi ja vaihtaa koiran tyhjäkäyntianimaation juoksuksi, saaden hyökkäyksen tuntumaan huomattavasti pelottavammalta ja vakuuttamalta tapahtumalta pelissä. Pelaajan poistuessa kyseiseltä havaitsemisalueelta, siirtyy viholliskoira takaisin tyhjäkäyntiin, jonka aikana se voi pyörähtää tietyn intervallin ajoin katsomaan 180 astetta eri suuntaan FlipFacing-funktion kautta. Samaa funktiota hyödynnettiin pelaajan saapuessa havaitsemisalueelle eri suunnasta kuin vihollinen, jota kautta viholliskoira kääntyy automaattisesti pelaaja kohti.

Vihollisen vahingon tekeminen ja elämään liittyvät tekijät liitettiin eri ohjelmakoodeihin kuin liikkuminen, sillä näiden hyödyntäminen on huomattavasti helpompaa myös jatkoa ajatellen eri objekteissa, joilla saattaa samankaltaisia toimintoja ja asioita olla. Viholliskoiran osuessa pelaajaobjektiin aiheuttaa tämä vahinkoa pelaajalle, vähentäen tältä elämäpisteitä tähän liitetyssä koodissa ja pelimaailmassa. Tämän lisäksi osumisskripti antaa pelaajalle tyrmäysvoimaa pushBack-funktion kautta, joka vaikuttaa pelaajan fysiikkakomponenttiin, heittäen tämän sille annetulle voimalle vastakkaiseen suuntaan osumakohdasta. Tämän lisäksi ohjelmakoodi laskee automaattisesti intervallin sille, milloin pelaajan kimppuun on mahdollista hyökätä seuraavan osumisen jälkeen (katso kuvio 56). Ohjelmaa kirjoittaessa fysiikoiden tekeminen osumishetkellä oli erittäin haastava prosessi, sillä vektoreiden ja voimien antaminen on todella monimutkaista toteuttaa ohjelmointia hyödyntämällä. Tämän voiman antaminen täytyy tapahtua joka kerta samalla tavalla ja yhtä hyvin, ilman että se rikkoo pelin muuta fysiikkaa millään tasolla.



```
45      void OnTriggerStay2D(Collider2D other)
46      {
47          if (other.tag == "Player" && nextDamage < Time.time)
48          {
49              PlayerHealth thePlayerHealth = other.gameObject.GetComponent<PlayerHealth>();
50              thePlayerHealth.addDamage(damage);
51              nextDamage = Time.time + damageRate;
52
53              pushBack(other.transform);
54
55              player = other.GetComponent<playerCodes>();
56              player.knockbackCount = player.knockbackLenght;
57              if (other.transform.position.x < transform.position.x)
58              {
59                  player.knockFromRight = true;
60              }
61              else
62              {
63                  player.knockFromRight = false;
64              }
65          }
66      }
67      void pushBack(Transform pushedObject)
68      {
69          Vector2 pushDirection = new Vector2(0, (pushedObject.position.y - transform.position.y)).normalized;
70          pushDirection = pushDirection * pushBackForce;
71          Rigidbody2D pushRB = pushedObject.gameObject.GetComponent<Rigidbody2D>();
72          pushRB.velocity = Vector2.zero;
73          pushRB.AddForce(pushDirection, ForceMode2D.Impulse);
74      }
75  }
76
```

KUVIO 56. Vihollisen osumisen vaikutus pelaajan fysiikkaan

7.2.2 Vihollisaluksen luomisprosessi

Monimutkaisin toteutukseltaan pelissä oli luoda avaruusalusvihollinen, joka ampuu pelaaja kohti panoksia tykistään aina tämän sattuessa tarpeeksi lähelle, samalla liikkuen ilmassa kahden eri pisteen välillä. Suunnitteluvaiheessa pohdimme ja piirsimme useita eri alusmalleja, joita voisimme luoda ja rakentaa, mutta päädyimme lopulta hyvin yksinkertaiseen muotoon, joka ei animaatioita Unityn sisällä vaadi. Ohjelmoinnin osalta avaruusaluksen toiminta vaati kuitenkin useamman eri tiedoston ja komponentin, joihin on liitoksissa mm. Aluksen liikkumiseen, tähtäämiseen, ampumiseen ja osumiseen liittyvät asiat ja tekijät.

Kyseiseen objektiin luotiin hyvin laaja hierarkia eri objekteja, joihin liitettiin sitten niille ominaisia komponentteja sisälle. Isäntäobjekti hoitaa aluksen liikkumisen pelimaailmassa siirtymällä sille annettujen pisteiden välillä, kun taas lapsiobjekteissa on kiinni aluksen fysikaaliset tekijät, jotka vaikuttavat siihen osumiseen ja elämäpisteisiin, tähtäimen kääntymisen pelaajaa kohti sekä ammuksien luomiseen ja suuntaan vaikuttavat komponentit (katso kuvio 57).

KUVIO 57. Alus pelimaailmassa sekä siihen liitettyjä eri lapsiobjekteja

Aluksen liikkuessa pelimaailmassa, tähtää tähän liitetty ase jatkuvasti pelaaja kohti, ampuen tätä kohti panoksia tietyn intervallin välein. Aseen kääntyminen ja sitä kautta ampuminen oli erittäin haastava toteuttaa onnistuneesti, sillä kyseisessä ohjelmakoodissa käytettiin paljon vektoreita ja laskemista, jotta ase varmasti kääntyy aina oikeaan suuntaan ja panos lähtee myös oikeasta suunnasta eteenpäin. Kyseisessä ohjelmakoodissa pyörimisen apuna hyödynsimme kvaternioneja (Quaternions), jotka perustuvat monimutkaisiin numeroihin, joita ei ole helppo ymmärtää intuitiivisesti. Kuten tässäkin tapauksessa, hyödynsimme olemassa olevia kierroksia (esim. muunnoksen kautta) ja käyttäen näitä uusien kiertojen rakentamiseen. Tätä kautta saimme luotua aseen pyörimiselle sujuvan kiertämisen, sille rajatun akselin ympärillä (katso kuvio 58).

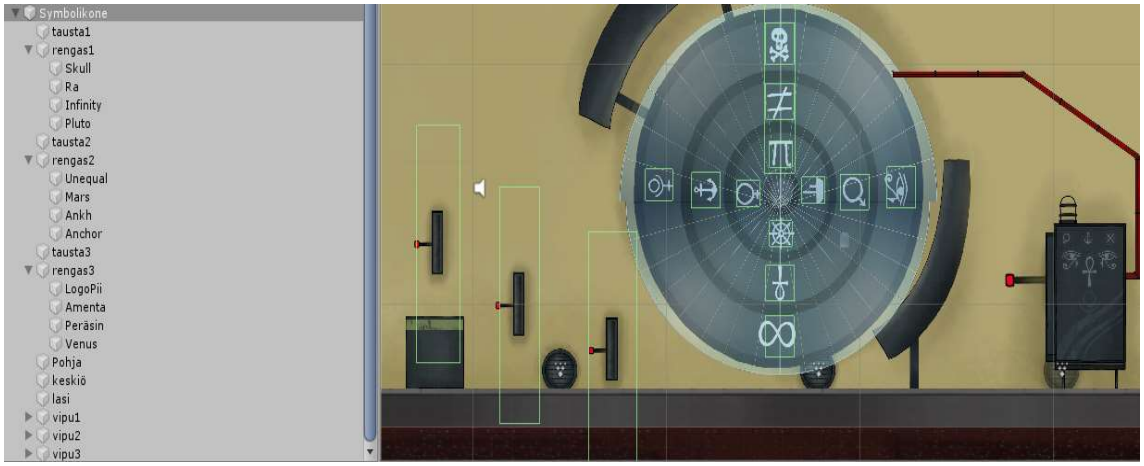
```
private void Start()
{
    TimeBetweenShots = startTimeBetweenShots;

    target = GameObject.FindWithTag("Player").transform;
}

private void Update()
{
    if(target != null)
    {
        if (target.transform.position.y - transform.position.y < -5)
        {
            Vector2 direction = target.position - transform.position;
            float angle = Mathf.Atan2(direction.y, direction.x) * Mathf.Rad2Deg;
            Quaternion rotation = Quaternion.AngleAxis(angle, Vector3.forward);
            transform.rotation = Quaternion.Slerp(transform.rotation, rotation, speed * Time.deltaTime);
        }

        if (TimeBetweenShots <= 0)
        {
            Instantiate(projectile, launchPoint.position, launchPoint.rotation);
            TimeBetweenShots = startTimeBetweenShots;
        }
        else
        {
            TimeBetweenShots -= Time.deltaTime;
        }
    }
```

KUVIO 58. Aluksen aseen kierron ja ampumisen ohjelmakoodia

7.2.3 Loppuvihollisen toteutus ja symbolikoneen toiminnallisuus pulmanratkomiselementtejä hyödyntäen.

Päävihollisen suunnittelu oli molemmille tekijöille erityisen mielekäs prosessi, sillä kyseistä taistelua pohdittiin ja suunniteltiin tarkasti ennen sen visuaalista ja käytännöllistä toteuttamista. Päävihollista varten loimme eräänlaisen symbolikoneen, jota kautta koko taistelun mekaniikka toimii. Yksinkertaisuudessaan vihollisen toiminta perustuu siihen mitä tämä sanoo pelissä ja sen mukaisesti pelaajan täytyy vaihtaa symboliset merkit oikein, jotta tämä pääsee ampumaan tätä kuvun läpi. Halusimme luoda pulmanratkomiselementtejä soveltavan taistelun, joka vaatii pelaajalta sekä ajattelukykyä että myös osaamista ja hallintaa liikkumisen ja ampumisen osalta. Siitä huolimatta, että päävihollisin tekeminen oli hyvin haastavaa yleisen toiminnallisuuden kannalta, helpotti alun hyvä ja laajempi suunnittelu heti kuitenkin edistymistä merkittävästi. Tässä osiossa käymme läpi tarkemmin, mitä kaikkea päävihollisen luomisen tekemisessä vaadittiin, erityisen vahvasti ohjelmointiin keskittyen ja sitä tarkemmin tutkien.

Selkeän hierarkian luominen helpotti merkittävästi symbolikoneen toiminnallisuuden luomisvaiheessa, sillä kaikki siihen liitetyt objektit olivat samassa paikassa ja sitä kautta helposti löydettävissä ja muokattavissa (katso kuvio 59). Kuten jo aiemmin grafiikan osalta mainittiin, symbolikoneen toiminnallisuus perustuu siihen, mitä päävihollinen sanoo ja sen mukaisesti täytyy nämä symbolit laittaa oikeaan järjestykseen, vääntämällä niille annettuja vipuja pelimaailmassa. Vipujen vääntämisen jälkeen symbolikoneen eri kerrokset pyörivät eteenpäin animaatioiden kautta, jonka avulla pelaajan on mahdollista saada nämä symbolit oikeaan järjestykseen.



KUVIO 59. Pelin päävihollisen symbolikone ja sen objektien hierarkia

Symbolikoneen päävipua vääntäessä, tulee kaikkien symbolien liittyä jollain tasolla päävihollisen antamiin vihjeisiin. Tästä riippuen koneeseen syttyy joko punainen tai vihreä valo sen merkiksi, onko symbolit laitettu oikein koneessa. Punaisen valon syttyessä, synnyttää pelimaailmassa oleva putki aiemmin luotuja vihollisia arpomalla Random.Range()-koodin avulla satunnaisen numeron ja sitä myöten synnyttämällä sille annetun arvon kautta pelimaailmaan tietyn määrän näitä vihollisia (katso kuvio 60). Random,Range()-funktion toiminta perustuu satunnaisen luvun arpomiseen kahden eri arvon (Integer) väliltä, jota kautta skriptitiedosto toteuttaa sille annetun arvon mukaisen toiminnon. Sen sijaan, jos symbolit on laitettu oikein vipua vääntäessä, syttyy koneen päälle vihreä valo, joka aukaisee päävihollisen ympärillä olevan kuvun (katso kuvio 61). Tässä hyödynsimme jälleen Unityn animaatio-ohjelmaa ohjelmoinnin yhteydessä, vaihtamalla animaattorissa olevan totuusparametrin todeksi, jota kautta kuvun aukaiseminen tapahtuu ja päävihollista suojaava osuma-alue komponentti menee päältä hetkellisesti. Kuvun avautuessa, on päävihollista mahdollista ampua, antaen tälle vahinkoa panoksien osuessa päävihollisen liitettyyn osuma-alue komponenttiin. Heti kun päävihollinen on saatu tapettua, syttyy pelimaailmassa olevaan oveen valo, jota kautta pelaaja pääsee kulkemaan eteenpäin maailmassa, vaihtaen seuraavaan mahdolliseen näkymään.

```
    private void Update()
    {

        TimeBetweenSpawns -= Time.deltaTime;
        if (TimeBetweenSpawns <= 0 && SymbolCheck.VihollisiaSaapuu == true)
        {
            TheNumber = Random.Range(1, 10);

            if(TheNumber == 1)
            {
                Instantiate(RobotDog, launchPoint.position, launchPoint.rotation);
            }
            else if (TheNumber == 2)
            {
                Instantiate(SpikeBall, launchPoint.position, launchPoint.rotation);
            }
            else if (TheNumber == 3)
            {
                Instantiate(EnemyShip, launchPoint.position, launchPoint.rotation);
            }
```

KUVIO 60. Pelaajan antaessa väärät symbolit, synnyttää tämä vihollisia pelimaailmaan

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

        if (Symbols.KupuAukeaa == true)
        {
            DomeOpenTime = Time.time + DomeWait;
            Symbols.KupuAukeaa = false;
        }

        if(Time.time >= DomeOpenTime)
        {
            Greenglass.SetActive(false);

            DomeOpen = false;
            DomeAnim.SetBool("DomeOpen", DomeOpen);

        }
        else
        {
            Greenglass.SetActive(true);

            DomeOpen = true;
            DomeAnim.SetBool("DomeOpen", DomeOpen);
        }
    }
}
```

KUVIO 61. Pelaajan antaessa oikeat symbolit, aukeaa kupu hetkellisesti

8 POHDINTA

Opinnäytetyössä oli tarkoitus toteuttaa 2D-tasohyppelypeli, jossa hyödynnettiin kahden tekijän eri osaamisalueita luomaan kokonaisvaltainen ja useista eri elementeistä koostuva pelin.
Ohjelmoinnin osalta oli hyvin vaikea keskittyä ja pohtia, mikä kaikki on oleellista ja tärkeää havainnollistaa ja selittää eri asioiden toiminnallisuuden luomisessa. Aihealue täytyi rajata merkittävästi ja vaikka pelissä olikin useita eri ohjelmointitiedostoja ja toiminnallisuutta, täytyi raportissa keskittyä kuitenkin vain suhteellisen pieneen osaan tästä kokonaisuudesta. Jotta lukija saisi mahdollisimman kokonaisvaltaisen kuvan eri asioiden toiminnallisuudesta ja tapahtumista, oli tärkeää keskittyä tiettyihin elementteihin toisia enemmän kyseisessä työssä. Esimerkiksi Unityn sisällä hyvin vähäiselle kirjoittamiselle ja pohtimisille raportissa jäi eri materiaalien sekä partikkelien käyttäminen ja luominen, UI elementtien merkitys pelimaailmassa ja näiden hyödyntäminen ohjelmoinnissa sekä äänimaailmalliset asiat osana pelin kokonaisuutta. Jokaisesta näistä elementeistä saisi kirjoitettua useita sivuja tekstiä, mutta keskityimme yhdessä enimmäkseen tutkimaan ja havainnollistamaan graafisia ja toiminnallisia elementtejä pelin eri toteutusvaiheiden osalta.

Kyseistä työtä tehdessämme opimme todella paljon Unity-pelimoottorin käytöstä, animaatioiden tekemisestä sekä peliohjelmoinnista. Suurimpia ongelmia työtä tehdessä olivat ajan käyttöön ja hyvien lähteiden löytämiseen liittyvät tekijät. Siitä huolimatta, että aihe oli hyvin laaja, oli laadukkaiden lähteiden löytäminen haastavaa tiedonhakuprosessin osalta. Peliä itsessään tehtiin Unity-pelimoottorilla pitkään ja useita päiviä putkeen, mutta kirjoittaminen oli haasteellista molemmille tekijöille, varsinkin alussa. Näistä ongelmista huolimatta saimme luotua ja kirjoitettua todella kokonaisvaltaisen paketin Unityn käytöstä pelinkehityksessä ja siitä kuinka graafisia asioita voidaan yhdistää ohjelmointiin. Myös pelin jatkokehitykselle on myöhemmässä vaiheessa mahdollisuuksia, sillä olemme luoneet tässä vaiheessa jo vahvan rungon pelin toiminnallisuuden osalta. Laajemman jatkotutkimuksen mahdollisessa luomisessa, keskittyisimme Unityn muiden elementtien laajempaan selittämiseen sekä ottaisimme tarkemmin esille muita kilpailevia moottoreita, joita ei mainittu lähes ollenkaan kyseisen työn aikana.

Unityn Collaborate -ominaisuutta käyttäessä suurimmiksi ongelmiksi muodostui tilan loppuminen sekä version hallintaan liittyvät ongelmat. Ongelmia version hallinnan kanssa ilmeni, kun tekijät olivat muokanneet samaa tiedostoa, ohjelmaan pystyi lataamaan vain toisen tekemät muutokset.

Unityn Collaborate -ominaisuuden ilmaisena tarjoama tila oli liian rajattu, mutta pienellä maksulla tilaa sai lisää.

Raportin alussa kerrotuista Tony Mannisen (2007, 31-32) pelisuunnittelun osa-alueista aliarvioimme projektissamme ainakin hyvän pelikokemuksen muodostamisen tärkeyden. Olisi ollut tärkeää ajatella enemmän sisällön toiminnallisuutta ja hyvän pelikokemuksen muodostamista, kuin sitä miltä se näyttää, tai miten peli toteutetaan.

LÄHTEET

Adobe 2019a. Adobe Photoshop. Viitattu 22.4.2019, https://www.adobe.com/fi/products/photoshop

Adobe 2019b. Adobe Animate. Viitattu 22.4.2019, https://www.adobe.com/fi/products/animate.html

Capcom 2019. Mega Man. Viitattu 22.4.2019, http://megaman.capcom.com/

Clark, J. 2017. Cuphead's Animation Process and Philosophy. Game Developers Conference -luento. https://www.youtube.com/watch?v=RmGb-jU3uVQ&t=1262s

Concept Art Empire 2019. Introduction To Thumbnailing And Quick Sketching. Viitattu 22.4.2019,

https://conceptartempire.com/intro-to-thumbnail-sketching/

Corona 2016. The art of game design and its importance. Viitattu 10.3.2019, https://coronalabs.com/blog/2016/11/08/the-art-of-game-design-and-its-importance/.

Coron, T. 2019. The best software for digital artists in 2019, Future Publishing Limited Quay House. Viitattu 22.4.2019 https://www.creativebloq.com/advice/the-best-software-for-digital-artists

Jenkins, J. 2018. What Makes a Great Villain? Your Checklist for Writing a Good Bad Guy. Viitattu22.4.2019, https://jerryjenkins.com/what-makes-a-good-villain/

Jääskeläinen, L 2018. Suuri Yleisö on alkanut nähdä 1960-luvun betoniarkkitehtuurissa arvoja, joita ei tule hävittää. Rakennuslehti. Viitattu 24.5.2019. https://www.rakennuslehti.fi/2018/03/suuri-yleiso-on-alkanut-nahda-1960-luvun-betoniarkkitehtuurissa-arvoja-joita-ei-tule-havittaa/

Klei Entertainment 2014. 2D Animation at Klei Entertainment. Game Developers Conference -luento. Viitattu 23.4.2019,
https://www.youtube.com/watch?v=8_KBjd0iaCU&list=PL2e4mYbwSTbbHAJT7OdK5mv-idhl-LOTI0&index=3

Manninen, T. 2007. Pelisuunnittelijan käsikirja: ideasta eteenpäin. Pello: Rajalla.

Nintendo 2019. Official home for Super Mario. Viitattu 22.4.2019, https://mario.nintendo.com/

Novak, M. 2015. 42 Visions For Tomorrow From The Golden Age of Futurism. Gizmodo Media Group. Viitattu 22.4.2019, https://gizmodo.com/42-visions-for-tomorrow-from-the-golden-age-of-futurism-1683553063

Parker, R. 2019. Viitattu 22.4.2019, http://rickparkergraphics.com/

Pluralsight 2014. Understanding the 12 Principles of Animation. Viitattu 22.4.2019, https://www.pluralsight.com/blog/film-games/understanding-12-principles-animation

Unity Technologies 2019a. Learning the Interface. Viitattu 10.3.2019, https://docs.unity3d.com/Manual/LearningtheInterface.html.

Unity Technologies 2019b. The Project window. Viitattu 10.3.2019, https://docs.unity3d.com/2019.1/Documentation/Manual/ProjectView.html.

Unity Technologies 2019c. The Scene window. Viitattu 10.3.2019, https://docs.unity3d.com/2019.1/Documentation/Manual/UsingTheSceneView.html.

Unity Technologies 2019d. The Game window. Viitattu 10.3.2019, https://docs.unity3d.com/2019.1/Documentation/Manual/GameView.html.

Unity Technologies 2019d. The Hierarchy window. Viitattu 10.3.2019, https://docs.unity3d.com/2019.1/Documentation/Manual/Hierarchy.html

Unity Technologies 2019e. Art Asset best practice guide. Viitattu 22.4.2019, https://docs.unity3d.com/Manual/HOWTO-ArtAssetBestPracticeGuide.html

Unity Technologies 2019f. GameObject. Viitattu 22.4.2019, https://docs.unity3d.com/ScriptReference/GameObject.html

Vieruaho, A. 2019. Disneyn perintö modernissa animaatiossa – Osa 1. Mainostoimisto Luma Oy. Viitattu 23.4.2019, https://mainostoimistoluma.fi/blogi/disneyn-perinto-modernissa-animaatiossa-1/

Williams, R. 2012. The Animator's Survival Kit - A Manual of Methods, Principles and Formulas for Classical, Computer, Games, Stop Motion and Inter. Expanded ed. Farrar, Straus & Giroux Inc.