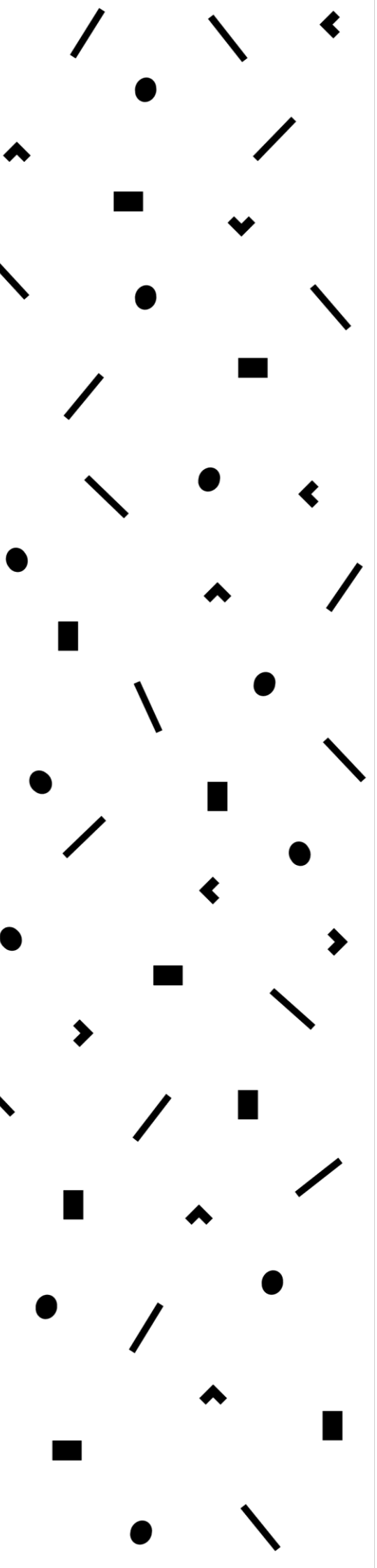


LAMK Lahden ammattikorkeakoulu
Lahti University of Applied Sciences

3D-MOBIILIPELIKEHITYS UNITY-PELIMOOTTORILLA

Kitten in Space-Time Universe

LAHDEN AMMATTIKORKEAKOULU
Tieto- ja viestintätekniikka
Ohjelmistotekniikka
Opinnäytetyö
Syksy 2018
Samu Saarikivi

Tiivistelmä

Tekijä(t)	Julkaisun laji	Valmistumisaika
Saarikivi, Samu	Opinnäytetyö, AMK	Syksy 2018
	Sivumäärä 39	

Työn nimi
3D-mobiilipelikehitys Unity-pelimoottorilla
Kitten in Space-Time Universe

Koulutusohjelma
Tieto- ja viestintätekniikan koulutus

Tiivistelmä

Opinnäytetyössä toteutetaan 3D-mobiilipeli Unity-pelimoottorilla Android-alustalle. Peli tehtiin vuonna 2015 valmistuneen prototyypin pohjalta, jota hyödynnettiin tavoitteiden hahmottamisessa. Opinnäytetyön toimeksiantajana on Tingleware Oy.

Opinnäytetyössä keskitytään erityisesti pelaajan sujuvan liikkumisen kehittämiseen 3D-maailmassa, proseduraalisesti kehittyvän reitin luomiseen ja pelaajan etenemisen seuraamiseen käytettyihin ratkaisuihin. Ratkaisuiden olennaisimmat C#-luokat kerrotaan yksityiskohtaisesti. Mobiilipelin laajuuden vuoksi pelin jokaiseen ominaisuuteen ja niiden toteutukseen ei tässä opinnäytetyössä tarkemmin voitu perehtyä.

Mobiilipeliä ei saatu täysin julkaisukelpoiseksi, mutta se saatiin Googlen Play Storeen suljettuun beta-testivaiheeseen. Pelin kehittäminen sujui suurimmilta osin hyvin, ja pahimmat ongelmat olivat pelin optimoinnissa mobiililaitteille. Lopulta pelin suorituskyky saatiin kiitettävälle tasolle useimpia nykypäivän puhelimia käyttäen. Tingleware Oy jatkaa mobiilipelin kehitystä edelleen tavoitteena saavuttaa mobiilipelin 1.0 julkaisuversio.

Avainsanat
3d, Android, pelinkehitys, ScriptableObject, Unity

Abstract

Author(s)	Type of publication	Published
Saarikivi, Samu	Bachelor's thesis	Autumn 2018
	Number of pages 39	

Title of publication

3D mobile game development using the Unity game engine

Kitten in Space-Time Universe

Degree programme

Bachelor's Degree Programme in Information and Communications Technology

Abstract

This Bachelor's thesis deals with a making 3D mobile game with the Unity game engine to the Android platform. The game was developed upon a prototype of the same game from year 2015, which gave a direction to making the final product. The thesis was commissioned by Tingleware Oy.

The main focus of the thesis was on developing fluid player movement in a 3D world, generating a procedural path for the player to follow and keeping a score of player progression. The most important C# classes are given a closer look with explanation about what is happening in code, with pictures of the end product. Given the size of the mobile game, every aspect of the development could not be given a closer look in this thesis.

The game did not reach its final commercial build, but it reached the closed beta test stage in Google Play Store. The development process went mostly without a problem. The biggest challenges were in the optimization to various mobile phones. The final product is performant enough to run with a solid framerate on most of the new mobile phones. The game will be developed further by Tingleware Oy.

Keywords

3d, Android, game development, ScriptableObject, Unity

SISÄLLYS

1 JOHDANTO

Opinnäytetyön toimeksiantajana on lahtelainen peliyritys Tingleware Oy. Yritys on erikoistunut tekemään hyötypelejä lapsille. Hyötypelit ovat pelejä, joiden tarkoituksena on saada pelaaja oppimaan tosielämän käsitteitä pelaamisen ohella. Tingleware Oy on esimerkiksi kehittänyt mobiilipelin Päiki Pörriäinen, joka on tehty yhteistyössä Päijä-Hämeen sosiaali- ja terveysyhtymän kanssa. Peli on suunnattu sairaalassa käyville lapsille. Sen tarkoituksena on poistaa lapsilla olevaa pelkoa ja ahdistusta sairaalakäyntejä kohtaan. (Tingleware.com 2018.)

Opinnäytetyössä kehitetään mobiilipeli, jossa pelaaja ja pelin sankari, Kisu, etenee poimittavien kalojen muodostamaa reittiä pitkin rakettireppu selässään. Kisu on myös mobiilipelin nimi, joka on akronyymi englanninkielen sanoista Kitten In Space-Time Universe. Pelaajan kuljettamaksi luotu reitti koostuu useista ennalta määritetyistä lyhyemmistä reiteistä, jotka yhdistyvät satunnaisessa järjestyksessä, luoden osittain proseduraalisesti kehittyvän reitin pelaajalle. Pelaajan etenemistä hankaloittaa jatkuvasti kehittyvä vaikeusaste. Pelaajan tavoitteena on päästä mahdollisimman pitkälle ja saavuttaa aurinkokunnasta löytyviä astronomisia kohteita.

Mobiilipeli kehitetään vuonna 2015 valmistuneen prototyypin pohjalta Unity-pelimoottorilla. Tingleware Oy antoi pelin kehittämiseen ja omien ratkaisujen soveltamiseen vapaat kädet.

Opinnäytetyössä on tavoitteena saada mobiilipeli julkaisuvalmiiksi Google Play Storeen. Työssä ei käydä koko pelin kehitystä läpi, vaan perehdytään muutamiin mielenkiintoisimpiin ratkaisuihin, kuten pelaajan liikkumiseen, pelikentän proseduraaliseen syntymiseen ja pelaajan etenemisen seuraamiseen.

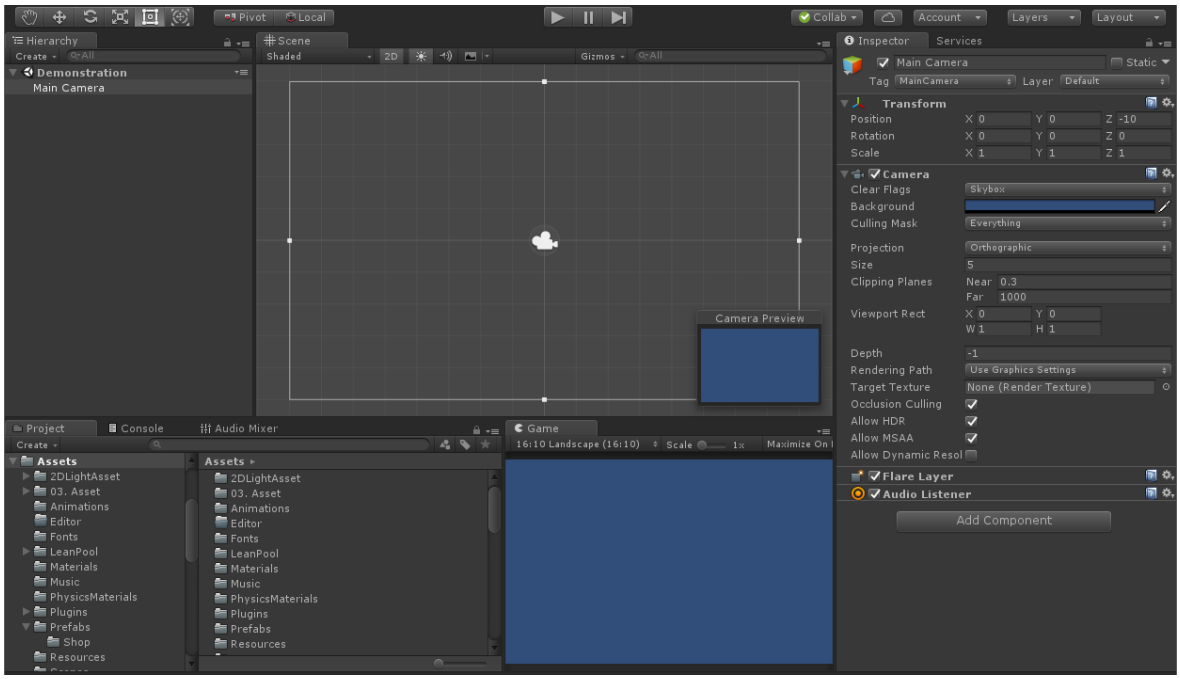
2 UNITY-PELIMOOTTORI

2.1 Unity

Unity on tanskalaisen Unity Technologiesin kehittämä pelimoottori, jolla on mahdollista tehdä 2D- ja 3D-pelejä lähes kaikille pelialustoille sekä VR-pelejä (Virtual Reality -pelejä) kyseistä teknologiaa tukeville alustoille. Unity Technologiesin toimistoja löytyy useita ympäri maailmaa ja yksi toimistoista sijaitsee Helsingissä, joka on vastuussa Unity Ads -mainospalvelusta. Palvelu helpottaa mainosten tuomisen pelaajan nähtäville, jolloin ilmaisesta pelistä saadaan tuottoa pelin julkaisijalle. Mainokset ovat suuri tuoton lähde ilmaisjakelussa olevalle pelille. Pelinkehitys Unity-pelimoottorilla on kehittäjille ilmaista, mikäli kehittäjän liikevaihto vuodessa on alle 100 000 dollaria (Unity Products 2018). (Unity3d 2018.)

2.2 Unityn käyttöliittymä

Unity Editorin käyttöliittymä koostuu vakiona kuudesta eri ikkunasta: Scene- ja pelinäkymästä, Inspectorista, Hierarkiasta, Projektikansiosta sekä konsolista. Näiden ikkunoiden järjestystä ja kokoa voi muuttaa vapaasti. Samoja ikkunoita voi editorissa olla auki monta yhtäaikaisesti ja perusikkunoiden lisäksi muilla toiminnoilla varustettuja ikkunoita on mahdollista käyttää. Ikkunoiden lisäksi Unityn käyttöliittymä koostuu sen yläreunassa sijaitsevasta työkalupalkista. (KUVA 1.)



KUVA 1. Unityn käyttöliittymä

2.2.1 Projekti-ikkuna

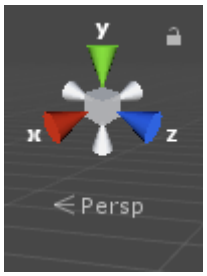
Projekti-ikkunassa sijaitsee avoinna olevan projektin Assets-kansio sisältöineen. Assets-kansion kansirakenne on näkyvillä ikkunan vasemmassa reunassa, ja valitsemalla kansion päivittyy ikkunan oikea puoli valitun kansion sisällöllä. Projekti-ikkunan oikeassa yläkulmassa on hakukenttä, joka etsii ja suodattaa Assets-kansion tiedostojen näkyvyyttä samalla, kun hakusanaa kirjoitetaan.

2.2.2 Scene-näkymä

Scene-näkymä näyttää senhetkisen Scenen maailman muokattavassa tilassa. Siellä on näkyvissä kaikki Sceneen lisätyt peliobjektit, kuten esimerkiksi hahmot, maailma, valaistus ja kamerat. Näitä peliobjekteja on helppo Scene-näkymässä lisätä, poistaa, liikuttaa tai kiertää. Ikkunassa näkyviä peliobjekteja klikkaamalla ne tulevat valituksi, jolloin niihin voi tehdä esimerkiksi edellä mainittuja muutoksia. Valitut peliobjektit muuttuvat valituksi myös hierarkiassa ja Inspectorissa.

Peliobjektien lisäksi Scene-näkymässä on gizmoja, jotka auttavat hahmottamaan ympärillä olevaa maailmaa ja peliobjektien ominaisuuksia. Scene-näkymän oikeasta yläkulmasta löytyy Scene-gizmo, joka osoittaa aina World Space XYZ-suuntiin niitä vastaavilla

osoittimilla (KUVA 2). Scene-gizmon suunta riippuu Scene-näkymän kameran suunnasta. Scene-Gizmo on näkyvissä, kun Scene-näkymässä ei ole 2D-tila päällä. Scenen ja sen syvyyden hahmottamista helpottaa myös vaakatasossa oleva ruudukko Y-akselin nolla-kohdassa. Nämä ja kaikki muut gizmot on mahdollista ottaa käyttöön Scene-näkymän ylä-reunasta löytyvästä Gizmos-dropdown valikosta.



KUVA 2. Scene-Gizmo

2.2.3 Pelinäkymä

Pelinäkymässä näkyy sen nimen mukaisesti Scenessä määritetty pelimaailma renderöi-tynä Camera-peliobjektista. Scenessä on oltava vähintään yksi kamera tai pelinäkymässä ei näy mitään. Pelinäkymässä on myös pelaajan käyttöliittymä, joten pelinäkymä vastaa pelin lopullista olemusta.

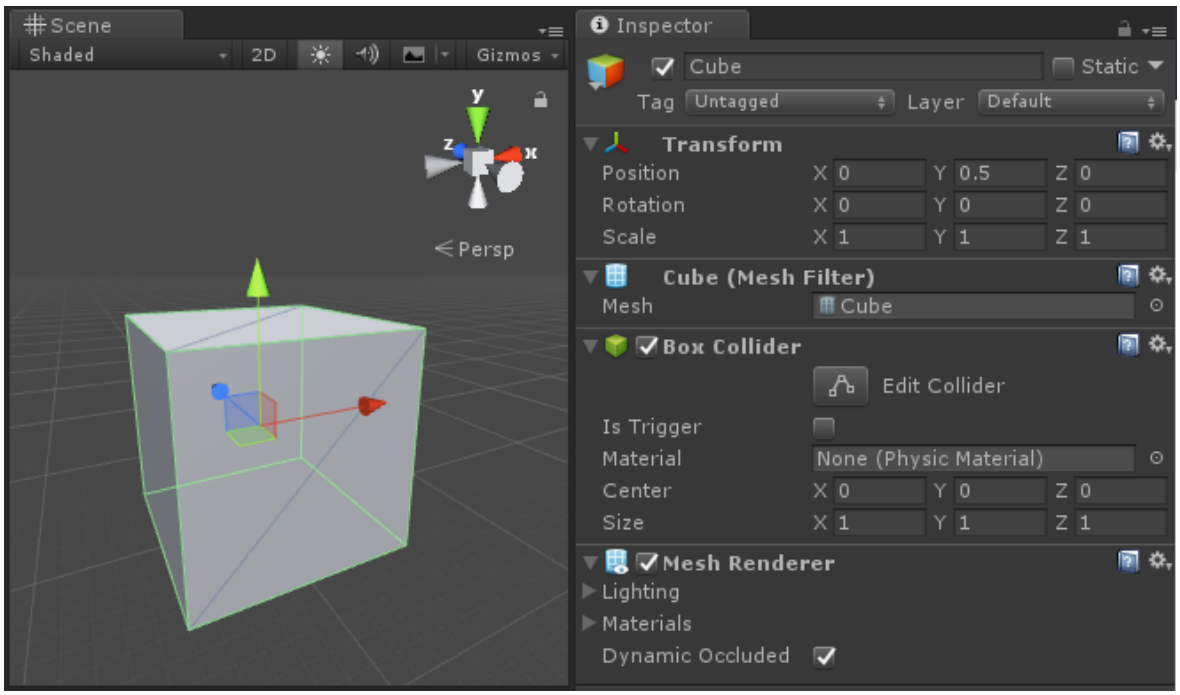
KUVA 3. Peli näkymän Play-, Pause-, ja Step-painikkeet

Pelinäkymän saa aktivoiduttua klikkaamalla työkalupalkin Play-painiketta, jolloin sen hetki-nen Scene on pelattavissa editorissa. Pause-painikkeella pelin ajon saa tauolle ja Step-painike siirtää ajettavaa peliä yhden kuvakehyksen verran eteenpäin. (KUVA 3.)

2.3 Peliobjekti

Kaikki Scenessä olevat hahmot, esineet ja asiat ovat peliobjekteja. Peliobjektit toimivat ikään kuin säiliöinä toimintaa omaaville komponenteille, eivätkä ne sellaisenaan pidä si-sällään minkäänlaista toiminnallisuutta. Jokaisessa peliobjektissa on vakiona Muunnos (Transform)-komponentti, jota ei voi poistaa. Muunnos-komponentti mahdollistaa

peliobjektin paikan, rotaation ja koon muokkaamisen. Komponentin lukemat kuvastavat peliobjektin olemusta Scene-näkymässä. Muunnos-komponentin muokkaukset tapahtuvat suhteessa lokaalisti ja muokkaukset vaikuttavat sen lapsi peliobjekteihin.



KUVA 4. Kuutio ja sen vakio komponentit

2.4 Komponentit

Komponentit ovat peliobjekteissa olevia skriptejä ja niitä voi olla peliobjektissa useita erilaisia samanaikaisesti (Unity3d Docs 2018). Kuvassa 4 näkyy kuution neljä eri komponenttia, joilla jokaisella on oma roolinsa; Mesh Renderer saa esineen muodon Mesh Filteriltä ja renderöi sen Muunnos-komponentin osoittamilla ominaisuuksilla. Box Collider määrittää kuutiolle fyysiset seinämät (Unity3d Docs 2018).

Omia komponentteja on toistaiseksi mahdollista luoda C#- ja JavaScript-ohjelmointikielillä. Elokuussa 2017 Unity Technologies ilmoitti blogikirjoituksessaan lopettavansa JavaScript-tuen jatkokehittämisen Unityssä. Blogikirjoituksessa ilmeni yksinkertaiseksi syyksi C#-kielen tuleva ylivertaisuus pelinkehityksessä JavaScriptiin nähden ja suosituimman C#-kielen tukeminen täysin resurssein on Unity Technologiesille kannattavampaa. Toistaiseksi (6.5.2018) Unityn tuoreimmassa 2018.1 versiossa JavaScript skriptien luominen ja käyttäminen on mahdollista. (Unity3d blogs 2018.)

Skriptien ja komponenttien luominen onnistuu projekti-ikkunan Assets-kansiossa. Jokainen komponentiksi suunnattu skripti periytyy MonoBehaviour-luokasta. Luokka mahdollistaa komponentin ja Unityn vuorovaikutuksen, ja se toimii komponenttien pohjana. MonoBehaviour-luokka tarjoaa useita valmiita metodeja pelilogiikan rakentamista varten, esimerkiksi Start()- ja Update()-metodit sekä fyysisiin törmäystarkasteluihin käytettävät metodit, kuten OnCollisionEnter() ja OnCollisionExit().

3 PELIN SUUNNITTELU JA PELIN IDEA

3.1 Pelin idea ja pelaajan tavoite

Pelin sankari Kisu etenee rakettirepun avulla avaruuden halki. Edetessään hän kerää kaloja saavuttaakseen tarpeeksi energiaa seuraavaan hyperavaruus-tasoon. Tasoja on neljä, ja jokainen taso saattaa Kisun lähemmäs valonnopeutta. Etenemistä hankaloittavat useat erilaiset esteet, jotka luovat jokaisella hyperavaruus-tasolla entistä hurjempia vaaroja. Kisun nopea vauhti nopeuttaa hänen pääsemistään seuraaviin tähtitieteellisiin kohteisiin, ensimmäisenä ohitettavana oleva kuu on vasta alkua ja lopullinen kohde on jopa 4,36 valovuoden päässä oleva Alpha Centauri. Kisun törmätessä vaaroihin kaksi kertaa peli epäonnistuu ja pelaaja menettää yhden Kisun yhdeksästä elämästä. Elämiä tulee tietyn väliajoin automaattisesti lisää yksitellen, mutta mainoksia katsomalla pelaaja voi välillä ansaita kolme elämää kerralla. Pelaajan eteneminen on kumulatiivista eli jokainen pelikerta vie Kisun lähemmäksi kohteita, epäonnistuminen ei nollaa kuljettuja kilometrejä.

Kisun kohteet, niiden järjestys ja etäisyydet ovat tähtitieteellisesti mahdollisimman tarkkoja. Pelaajalla on siis mahdollisuus oppia pelatessaan maapallon ympärillä olevaa aurinkokuntaa.

3.1.1 Kohderyhmä ja -alusta

Pelin kohderyhmänä on kaikenikäiset miehet ja naiset, mutta pelin piirrosmainen grafiikkatyyli ja ideat vetoavat parhaiten lapsiin ja lapsenmielisiin ihmisiin. Peli on suunniteltu julkaistavaksi alkuun vain Android-alustalle. Unity-pelimoottorin monialustaisen julkaisutuen ansiosta mobiilipelin kääntäminen iOS-yhteensopivaksi jälkikäteen ei vaadi suuria toimenpiteitä. Unity-pelimoottori mahdollistaa pelien vaivattoman julkaisemisen usealle eri alustalle.

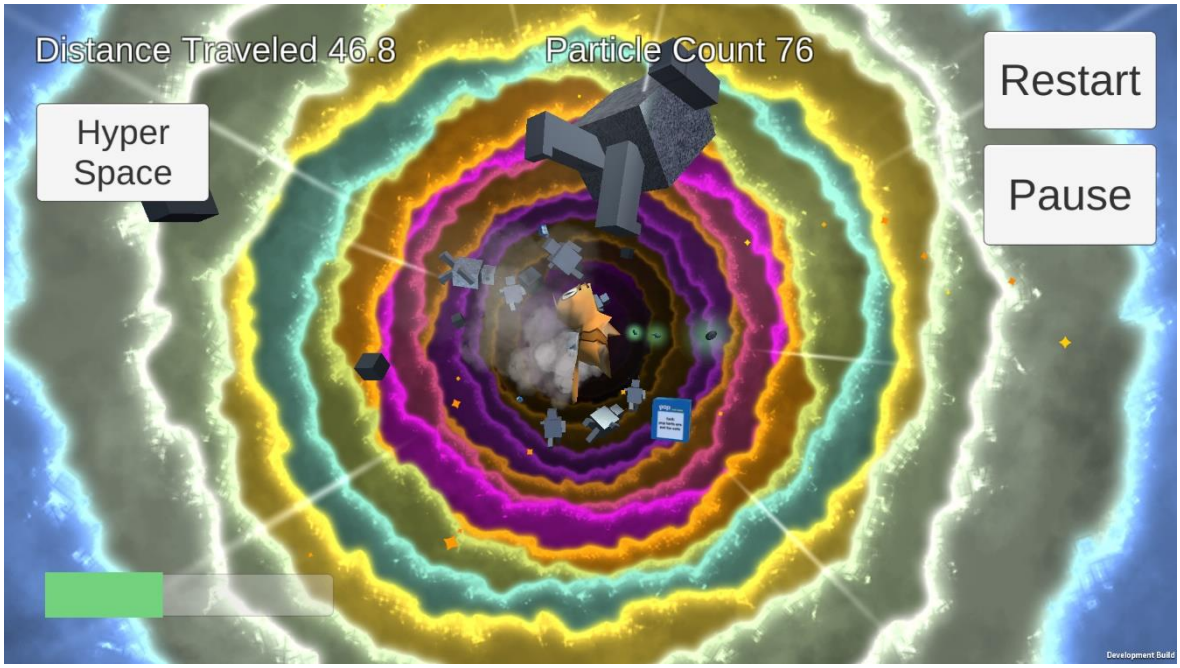
3.1.2 Pelin haasteellisuus

Pelin vaikeustaso nousee jokaisella hyperavaruus-tasolla. Aloittaessa pelaaja väistelee yksinkertaisia pieniä asteroideja ja keräilee kaloja, jotka muodostavat suoraviivaisia reittejä. Jokainen uusi hyperavaruus taso tuo uusia, entistä hankalampia esteitä pelaajan tielle edellisten esteiden lisäksi. Kalojen muodostamat reitit muuttuvat myös monimutkaisemmiksi. Myös pelaajan nopeus kasvaa pelin edetessä. Viimeisellä eli neljännellä hyperavaruus tasolla pelaajan nopeus on noin tuplasti nopeampi kuin aloittaessa. Kilometrejä pelaajalle kuitenkin kertyy pelin matkamittariin huomattavasti moninkertaisella nopeudella kuin miltä Kisun nopeus oikeasti näyttää, koska astronomiset nopeudet ovat

epärealistisen suuret. Suuri vauhdin tuntu luodaan esimerkiksi partikkeli systeemeillä, kameran näkökentän muokkauksilla ja musiikin vaihdoksilla.

3.2 Prototyyppi

Projektia aloittaessa pelistä oli jo valmiina prototyyppi, jossa oli muutamat pelin perusominaisuudet toteutettu, kuten pelaajan liikkuminen ja esteiden sekä kalojen satunnainen syntyminen. Prototyyppi antaa selkeän kuvan pelin teemasta ja helpottaa projektin tavoitteiden hahmottelemista. Prototyyppi on vuodelta 2015 ja se on toteutettu Unityn 5.1 versiolla. (KUVA 5.)



KUVA 5. Kuvakaappaus Kisun prototyypistä

4 TYÖVÄLINEET JA PROJEKTIN ALOITUS

4.1 Projektin Unity-version päivittäminen

Kisun prototyyppi vuodelta 2015 päivitettiin Unity-versiosta 5.1 tuoreimpaan vakaaseen Unity versioon 2017.1.1. Virheitä ei tullut päivittäessä lainkaan, ainoastaan varoituksia vanhentuneiden metodien ja luokkien käytöstä. Varoitukset liittyivät vanhentuneeseen partikkeli systeemin ja Scenen lataamiseen Application.LoadLevel(int index)-metodilla. Projektin edetessä Unity-versio pidettiin ajan tasalla. Projektin aikana Unityyn tuli kaksi isompaa päivitystä aina versioon 2017.3 asti. Jokainen päivitys sujui ongelmitta.

4.2 Unity Collaborate

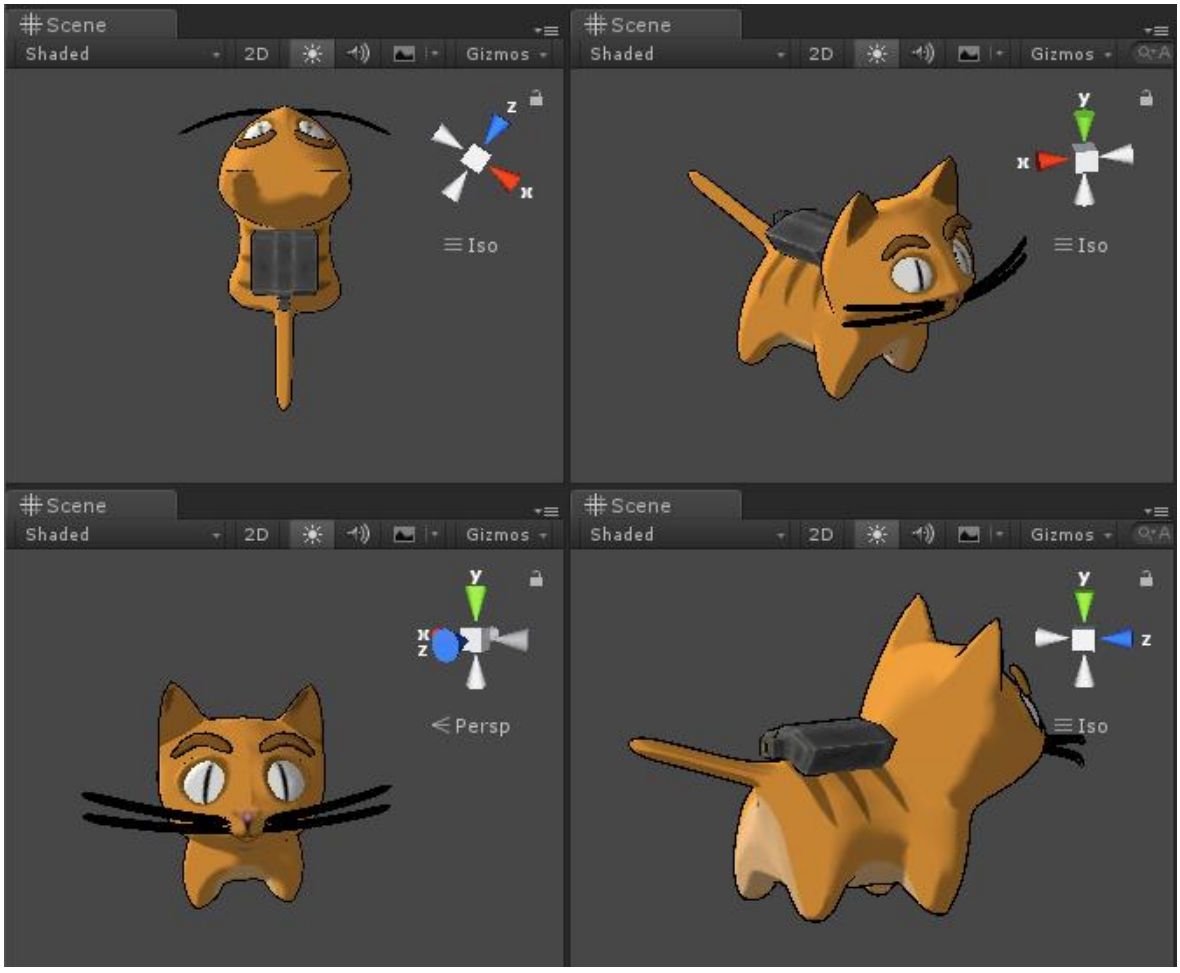
Collaborate on Unityyn sisäänrakennettu versionhallinta työkalu, joka on suunniteltu 1-10 hengen kokoisille pelinkehitystiimeille. Sen avulla kehitystiimi pysyy ajan tasalla projektiin tehdyistä muutoksista. Collaborate säilyttää pilvessä jokaisen sinne julkaistun version projektista, jotta vanhojen tiedostojen palautus tai koko projektin päivittäminen vanhempaan versioon olisi mahdollista. (Unity3d docs 2018.)

Collaboraten aktivoiminen tapahtuu Unityn Services -ikkunasta. Aktivoinnin jälkeen projektin voi lähettää Unityn isännöimälle pilvipalvelimelle, jolloin projekti on käytettävissä projektiin linkatuilla jäsenillä. KISU:ssa käytettiin Collaboraten ilmaisversiota, joka rajoittaa projektin koon yhteen gigabittiin ja jäseniä Collaborate projektissa saa olla enimmillään kolme (Unity3d Teams 2018). KISU:n tapauksessa pysyttiin ilmaisversion rajausten sisällä.

5 PELAAJA

5.1 Hahmo

Pelin pelihahmona on kuvassa 6 oleva rakettirepulla varustettu kissa nimeltä Kisu. Pelaaja peliobjektilla on kaksi olennaista lapsi peliobjektia muodon ja ulkonäön saavuttamiseksi, Torso ja Kitten texture. Torso on pelaajan selkäranka, jonka lapsina on kaikki sen raajat ja niiden osat. Torson hierarkia koostuu neljästä jalasta, kolmiosaisesta hännästä ja päästä, jonka lapsina on pelaajan korvat. Nämä kaikki muodostavat luurangon pelaajalle. Pelaajan hierarkkinen rakenne on nähtävissä kuvassa 7.



KUVA 6. Pelihahmo

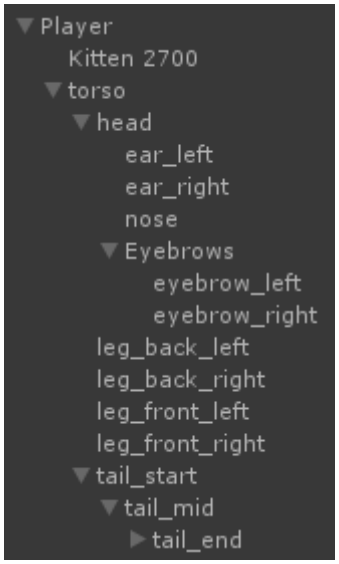
Kitten texture toimii pelaajan ihona. Peliobjektissa on Skinned Mesh Renderer -komponentti, jonka Unity lisää 3D-mallinnusohjelmista tuotuihin malleihin automaattisesti. Tätä

komponenttia käytetään, kun Meshin muoto liikkuu animaatioiden mukana tai jos halutaan saavuttaa realistinen räsynukke efekti peliobjektille. (Unity3d docs 2018.)

Tämän opinnäytetyön pelissä pelaajalla on ennalta määrättyjä animaatioita, kuten pelaajan kääntyillessä pään kääntyminen, sekä kaikille muille raajoille tapahtuva ilmanvastuksesta johtuva heiluminen pelaajan edetessä. Pelaajan törmätessä esteisiin sen Ragdoll-skripti aktivoituu, jolloin pelaaja pyörii hetken räsynukkena.

5.2 Räsynukke

Jokaiselle pelaajan ruumiinosalle on lisätty oma Rigidbody-komponentti simuloimaan realistista fysiikkaa ja Collider-komponentti, jotta peliobjektit huomioivat fyysisiä törmäyksiä. Primitiivisten Colliderien käyttö on suositeltavaa, sillä ne ovat huomattavasti kevyempiä prosessorille kuin monimutkaisemmat Colliderit. Primitiivisiä Collidereita ovat tehokkuus järjestyksessä kevyimmästä raskaimpaan Sphere Collider, Capsule Collider ja Box Collider (Unity3d docs 2018). Näiden komponenttien lisäksi ruumiinosilla on Character Joint-komponentti, joka on automaattisesti lisätty tuodessa pelaajan malli Unityyn. Character Joint -komponentti on pallonivel, jonka liikettä ja kiertämistä voi muokata ja rajoittaa jokaiseen suuntaan. Toimiakseen Character Joint -komponentti vaatii Rigidbodyn rinnakkaiskomponentiksi.



KUVA 7. Pelaaja-peliobjektin rakenne Scene-näkymän hierarkiassa

Räsynukke (Ragdoll) aktivoituu muutamaksi sekunniksi pelaajan törmätessä peliobjekteihin, joiden Tag-arvo on ”Hazard”. Peliobjektien Tag-arvot asetetaan Inpector-ikkunassa.

Aktivoiduttua pelaaja menettää ohjattavuuden Kisuun, joka räsynukkena pyörii törmäystä vastakkaiseen suuntaan. Räsynukke on toimeton, jos törmäyksen tapahtuessa ei simuloitaisi siitä syntyvää vastavoimaa, koska pelaajan liikkuessa transform.Translate-metodilla törmäyksien fysiikkalaskelmia ei tapahdu. (KUVIO 1.)

Pelaaja peliobjektin lapsiksi on lisätty kaksi tyhjää peliobjektia sensoreiksi, nämä sijaitsevat pelaajan selkärangan edessä ja takana. Sensorien nimet ovat SensorBack ja SensorFront. Ragdoll-komponentti vertaa törmätessään näiden kahden peliobjektin sijaintia ja tämän mukaan asettaa vastavoimat pelaajaan. Vastavoima tapahtuu kahdessa osassa. Pelaajan takaosaan laitetaan hetkeksi voimaa, jonka suunta on törmäyspintaa kohti. Tämän jälkeen etummaiseen sensoriin laitetaan voimaa vastakkaiseen, esteestä poispäin menevään suuntaan. Näiden kahden peräkkäisen voiman jälkeen pelaaja pyörii Inspectorissa asetetun float ragdollDuration -sekunnin ajan räsynukkena esteestä poispäin. Tämänkaltaisiin ajoituksiin käytetään Coroutineja. (KUVIO 1.)

```
IEnumerator Collision()
{
    // Aktivoi räsynukke Ragdoll- skriptissä.
    ragdoll.Ragdoll(true);
    rb.AddForceAtPosition(FirstBumpDirection() * bumpForcePower, sensorBack.transform.position);
    yield return new WaitForSeconds(0.25f);
    rb.AddForceAtPosition(ForceDirection(hazardTempPosition) * 600f, sensorFront.transform.position);
    yield return new WaitForSeconds(bumpDuration - 0.25f);
    // Jos pelaajalla on vielä elämiä, poista räsynukke käytöstä.
    if (PlayerStateController.Singleton.Playing)
        ragdoll.Ragdoll(false);
}

Vector3 FirstBumpDirection()
{
    float x = sensorFront.position.x > sensorBack.position.x ? -1 : 1;
    float y = sensorFront.position.y > sensorBack.position.y ? -1 : 1;
    return new Vector3(x, y, 1);
}
Vector3 ForceDirection(Vector3 hazardPosition)
{
    var heading = hazardTempPosition - transform.position;
    // Vektorin normalisointi.
    forceDirection = heading / heading.magnitude;
    return -forceDirection;
}
```

KUVIO 1. Luonnollisen törmäyksen luonti räsynukelle

5.3 WindResistance

WindResistance-komponentti hyödyntää pelaajan räsynukke ominaisuutta realistisen ilmanvastuksen simuloimisessa. Pelaajan edetessä avaruudessa kaikki sen raajat, korvat ja häntä liikehtivät kuin vastatuuleen lentäessä. Tämä tuo vauhdin tuntua peliin. WindResistance-komponentti lisää sen isäntä peliobjektin Rigidbody-komponenttiin AddForce()-metodilla valitun määrän voimaa säännöllisin aikavälein. (KUVIO 2.)

```
using UnityEngine;

public class WindResistance : MonoBehaviour {
    [SerializeField] float forceAmount = 20f;
    [SerializeField] float timing = 1f;
    Rigidbody rb;
    void Start () {
        rb = GetComponent<Rigidbody>();
        InvokeRepeating("AddForceToSelf", 0f, timing);
    }

    void AddForceToSelf()
    {
        rb.AddForce(-Vector3.forward * forceAmount);
    }
}
```

KUVIO 2. Ilmanvastus-skripti

Realistisuuden saavuttamiseksi voiman ja aikavälin suuruus asetetaan kohteen RigidBodyn massan mukaisesti. Suurempimassaiset, kuten jalkojen RigidBodyt, saavat enemmän voimaa, mutta harvemmin aikavälein kuin pienemmät ja kevyemmät ruumiinosat. Voima on suunnattu menosuunnan vastaiseksi.

WindResistance-komponentti on lisätty kaikkiin pelaajan ruumiinosiin, joihin halutaan ilmanvastusta. Jalkojen tuulenvastuksen aikavälit ovat jokaisella eri, satunnaisesti 0,45 ja 0,6 sekunnin välillä. Eriaikaisuus tekee jaloista luonnollisemman näköiset. Häntä on Kisulla kolmiosainen: se koostuu hännän juuresta, keskiosasta ja kärjestä. WindResistance-komponentti lisätään vain hännän kärkeen. Aikaväli ja voima asetetaan hyvin pieneksi. Muu häntä seuraa kärki osaa ja näin hännän liikehdintä muistuttaa käärmemäistä liikettä.

5.4 Ohjaus

Pelaajan liikkumista, rotaatiota ja kameran liikkeitä ohjataan puhelinta kallistamalla. Puhelimen kaltevuuden seuraamiseen käytetään lähes jokaisesta Android puhelimesta

löytyvää kiihtyvyysanturia. Pelaaja voi vaikuttaa pelin asetukset-valikosta liikkumisen ohjausherkkyyteen. Lisäksi ohjauksen suunnan voi vaihtaa käänteiseksi.

5.4.1 Kiihtyvyysanturi ja Kiihtyvyysanturin kalibrointi

Kiihtyvyysanturin lukemat saadaan Input-luokan acceleration-vektorista. Pelaajan syötettä ei käytetä sellaisenaan, vaan aluksi acceleration-vektori kalibroidaan AccelerometerCalibration-komponentissa. Seuraavaksi Accelerometer-komponentti muuttaa kalibroidun vektorin Update()-metodissa tiltInput-vektoriksi, jota pelaaja peliobjekti käyttää liikkeittensä määrittämiseen. TiltInput-vektoria hyödynnetään myös kääntäessä kameraa ja pelihahmon päätä pelin aikana.

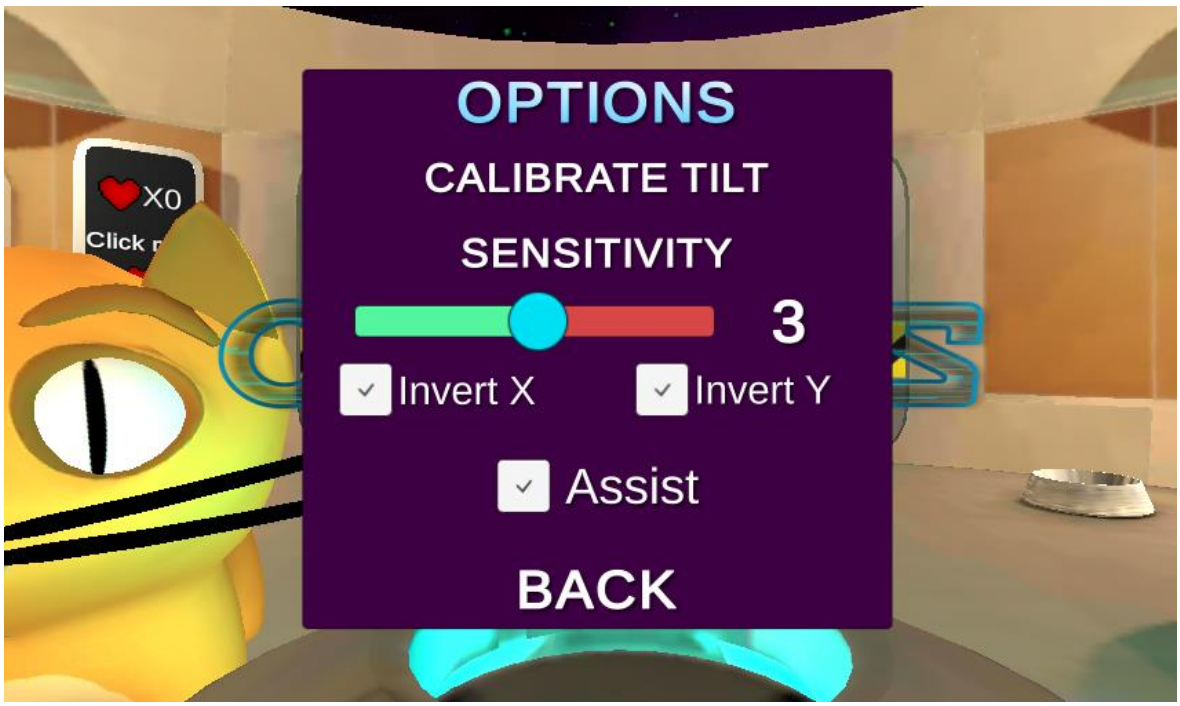
Kalibroitaessa kiihtyvyysanturin nollakohta asetetaan laitteen tämänhetkiseen asentoon. Kalibroinnin jälkeen AccelerometerCalibration.GetAccelerometer()-metodi palauttaa nollavektorin laitteen ollessa samassa asennossa, kuin kalibroitaessa. Kalibroinnin toteuttava skripti on nähtävissä kuviossa 3. Rivit 20 ja 21 liittyvät pelitestauksessa ilmenneiden ongelmakohtien ratkaisuihin, joista kerrotaan tämän pääkappaleen lopussa.

```
1   using UnityEngine;
2
3   public class AccelerometerCalibration : MonoBehaviour {
4       Matrix4x4 calibrationMatrix;
5       Vector3 calibratedAcceleration;
6
7       void Start () {
8           CalibrateAccelerometer();
9       }
10      public void CalibrateAccelerometer()
11      {
12          Vector3 wantedDeadZone = Input.acceleration;
13          Quaternion rotateQuaternion = Quaternion.FromToRotation(new Vector3(0f, 0f, -1f), wantedDeadZone);
14          Matrix4x4 matrix = Matrix4x4.TRS(Vector3.zero, rotateQuaternion, new Vector3(1f, 1f, 1f));
15          calibrationMatrix = matrix.inverse;
16      }
17      public Vector3 GetAccelerometer(Vector3 accelerator, float sensitivityX, float sensitivityY)
18      {
19          calibratedAcceleration = calibrationMatrix.MultiplyVector(accelerator);
20          calibratedAcceleration.x = Mathf.Clamp(calibratedAcceleration.x * sensitivityX * 1.45f, -1f ,1f);
21          calibratedAcceleration.y = Mathf.Clamp(calibratedAcceleration.y * sensitivityY, -1f, 1f);
22          return calibratedAcceleration;
23      }
24  }
```

KUVIO 3. AccelerometerCalibration-luokka

Pelaaja saa itse vaikuttaa pelin ohjausherkkyyteen muuttamalla pelin pause menun options-valikosta sensitivity-kentän arvoa. (KUVA 8.) Ohjausherkkyyden vakio arvo on 1,75 ja arvon ääripäät on pelitestauksen tuloksien perusteella laitettu yhdeksi (1) ja kolmeksi (3). AccelerometerCalibration-luokassa kalibroitu calibratedAcceleration-vektori kerrotaan pelaajan asettamalla ohjausherkkyydellä, ennen kuin metodi palauttaa sen Accelerometer-

luokan tiltInput-vektorille. Mitä isompi ohjausherkkyys on, sitä vähemmän laitetta tarvitsee kallistaa. Lopulta tiltInput-vektori normalisoidaan, eli sen pituus muutetaan yhdeksi (1), mutta suunta säilytetään samana. TiltInput.x ja tiltInput.y arvot ovat siis välillä -1 ja 1.



KUVA 8. Options-valikko

Options-valikossa pelaaja voi valita itselleen käänteisen X- ja Y-kääntyvyyden aktvioimalla invert X ja invert Y valintaruudut. Valintaruudun ollessa aktiivinen, kertoo se sitä vastaavan tiltInput.x tai -y arvon -1:llä. Näin pelaaja kääntyy vastakkaiseen suuntaan, kuin vakio asetuksilla puhelinta kääntäessä. (KUVA 8.)

Pelitestauksen aikana huomattiin, että pelaaja liikkuu herkemmin Y-akselin suuntaisesti pelattavien mobiililaitteiden mittasuhteista johtuen. Nykyään puhelinten kuvasuhde on noin 16:9, jonka takia Y-akselin suuntaiset kallistumiset tapahtuvat herkemmin, kuin sivuttaissuuntaiset kallistumiset. Pelaajaa oli siis työläämpi ohjata sivuttain. Tämä korjattiin yksinkertaisesti kertomalla tiltInput.x luvulla 1,45. Kerroin pohjautuu mobiililaitteiden kuvasuhteeseen 16:9. Lopulta kuvasuhteesta saatua arvoa laskettiin 1,45:een usean testikerran jälkeen.

Toinen ongelma oli kiihtyvyysanturin herkkyys, joka huomaa pelaajan pienimmätkin käden tärinät. Vector3.SmoothDamp()-metodi muuttaa vektorin kohti haluttua kohdetta tietyn

ajan kuluessa nimensä mukaan sulavasti ja palauttaa sen. Tällä funktiolla saatiin käden tärinät suodatettua pois pelaajan liikkeistä. (KUVIO 4.)

```
private void Update()
{
    // Hakee kiihtyvyysanturin lukemat AccelerometerCalibration- luokasta.
    // SensitivityX ja -Y on pelaajan valittavissa (0.9f - 2.2f),
    // invertX ja -Y on arvoltaan 1 tai -1.
    Vector3 targetTiltInput = calibration.GetAccelerometer(Input.acceleration, sensitivityX * invertX, sensitivityY * invertY);
    // Vaihtaa asteittain tiltInput vektoria targetTiltInput- vektoria kohden ajassa smoothTime.
    tiltInput = Vector3.SmoothDamp(tiltInput, targetTiltInput, ref velocity, smoothTime);
}
```

KUVIO 4. Käden tärinöiden suodatus Accelerometer.Update()-metodissa

5.4.2 Liikkuminen

Unityllä voi liikuttaa peliobjekteja viidellä eri tavalla. Näistä kolme, Rigidbody.MovePosition()-, Rigidbody.AddForce()-metodit, sekä Rigidbody.velocity-vektorin asettaminen liikuttavat pelaajaa fysiikkamoottoria hyödyntäen. Kaksi muuta vaihtoehtoa liikuttaa peliobjektia muokkaavat sen Muunnos-komponenttia. Transform.Translate()-metodi sekä Transform.position-vektorin asettaminen muuttaa pelaajan Muunnos-komponentin sijainnin haluttuun paikkaan. Peliobjekteja liikuttaessa Rigidbody-menetelmillä fyysiset törmäykset ovat tarkempia, kuin Muunnos-menetelmillä, mutta liikkuminen ei ole yhtä responsiivista nopeiden käännösten tapahtuessa. Rigidbody.velocity-vektoria muuttaessa Rigidbodyyn on mahdollista saada äkillisiä suunnanmuutoksia, mutta Unityn dokumentaatiossa sen käyttöä ei suositella, koska sen muuttaminen skriptillä luo epärealistisia fysiikka simulaatioita (Unity3d docs 2018).

Tämän opinnäytetyön pelissä tapahtuu pelaajan osuttua vaaroihin fyysisiä törmäyksiä, joiden tarkkuus ei ole olennaista. Tästä syystä pelaajaa liikutetaan Muunnos-komponentin sijainti vektoria muokkaamalla Transform.Translate(Vector3 translation, Space relativeTo = Space.Self)-metodilla. Vector3 translation on peliobjektiin kohdistuvan muutoksen suunta ja pituus. Space relativeTo-enumeraattori ilmaisee, minkä akselien suhteen peliobjekti liikkuu. Space.World-enumeraattoria käyttäessä peliobjekti liikkuu maailman akselien mukaisesti, kun Space.Self-enumeraattori liikuttaa peliobjektia sen rotaation mukaisesti. Kuviossa 5 on esimerkki, jossa peliobjekti liikkuu aina siihen suuntaan mihin se osoittaa, eli paikalliseen Vector3(0, 0, 1)-suuntaan.

```
void Update() {
    transform.Translate(Vector3.forward * Time.deltaTime, Space.Self);
}
```

KUVIO 5. Transform.Translate()-metodi

Pelaaja-peliobjekti liikkuu eteenpäin Z-akselin suuntaisesti tasaisella nopeudella. Kuviossa alla on pelaajan liikkumisen toteuttava PlayerMovement-luokka yksinkertaistettuna. X- ja Y-akselin suuntainen liike luetaan tiltInput-vektorista, eli pelattavan laitteen kiihtyvyysanturin lukemista. Ennen kuin pelaajaa liikutetaan rivin 12 Translate()-metodilla, asetetaan movement.z arvoksi vakionopeus (speed), koska kiihtyvyysanturin ei haluta vaikuttavan syvyys suuntaiseen nopeuteen.. Kertomalla Update()-metodissa tapahtuvat laskutoimitukset Time.deltaTime-floatilla varmistetaan, ettei pelin toiminnallisuus ole sidottu ruudunpäivitysnopeuteen. (KUVIO 6.)

```
1   using UnityEngine;
2
3   public class PlayerMovement : MonoBehaviour {
4
5       public float speedMultiplier = 1f;
6       [SerializeField] float speed = 20f;
7       [SerializeField] float jetpackMovementMultiplier = 0.5f;
8
9       void Update() {
10          Vector3 movement = Accelerometer.Singleton.tiltInput;
11          movement = new Vector3(movement.x * speed, movement.y * speed, speed) * speedMultiplier;
12          transform.Translate(movement * Time.deltaTime, Space.World);
13      }
14  }
```

KUVIO 6. PlayerMovement-luokka

5.4.3 Rotaatio

Pelaajan liikkumisesta tulee luonnollisempi, kun se katsoo menosuuntaa kohti. Pelaaja peliobjektin rotaatiota muokataan liikkumisessakin käytetyn tiltInput-vektorin avulla. Pelaajan rotaatio olisi mahdollista päivittää Update()-metodissa suoraa tiltInput vektorin osoittamaan arvoon, mutta peliobjekti kääntyilisi nopeissa liikkeessä liian epäluonnollisen näköisesti ja kääntymisen määrällä ei olisi rajoituksia. Pelaaja peliobjektin ei haluta kääntyvän kameraa kohti, koska menosuunta on koko pelin ajan positiivinen Z-akselin suuntainen ja kamera on päinvastaisessa suunnassa.

PlayerRotation skripti on nähtävissä kokonaisuudessaan kuviossa 7. TiltRotation-vektorin arvo muutetaan Vector3.Lerp()-metodilla kohti tiltInput-vektoria. Vector3.Lerp()-metodin ja käden tärinöiden suodatuksessa käytetyn Vector3.SmoothDamp()-metodin ero on niiden

tapa muuttaa vektorin arvo haluttuun kohteeseen. Lerp()-metodi interpoloi vektorin arvon lineaarisesti kohteesen, kun SmoothDamp()-metodi muuttaa vektorin arvon kohteeseen ikään kuin alussa kiihdyttäen ja lopussa hidastaen.

MaxRotation muuttuja määrittää pelaaja peliobjektille ääriarvot rotaatiolle. TiltRotation-vektori normalisoidaan, jos sen pituus on yli yksi ja kerrotaan maxRotationilla. Normalisoidessa vektori sen pituus muutetaan yhdeksi (1), mutta sen suunta säilytetään ennallaan. Peliobjektin rotaation muutoksia ei haluta tapahtuvan Z-akselin suuntaisesti, joten tiltRotation.z-arvo muutetaan nollaksi (0). Seuraavaksi peliobjektin rotaatio asetetaan halutulla nopeudella menemään kohti juuri luotua tiltRotation-vektoria. Rotaation muutos saadaan aikaiseksi Quaternion.RotateTowards()-metodilla.

```
using UnityEngine;

public class PlayerRotation : MonoBehaviour {
    [SerializeField] float RotationSpeed = 150f;
    [SerializeField, Range(0, 90)] float maxRotation = 30f;
    Vector3 tiltRotation;
    PlayerMovement playerMovement;

    private void Start()
    {
        tiltRotation = Accelerometer.Singleton.tiltInput;
        playerMovement = GetComponent<PlayerMovement>();
    }
    void Update () {
        tiltRotation = Vector3.Lerp(tiltRotation, Accelerometer.Singleton.tiltInput, 0.5f);

        if (tiltRotation.sqrMagnitude > 1) tiltRotation.Normalize();

        float xTiltRotation = tiltRotation.y * -maxRotation;
        float yTiltRotation = tiltRotation.x * -maxRotation;

        Quaternion target = Quaternion.Euler(xTiltRotation, -yTiltRotation, 0);
        transform.rotation = Quaternion.RotateTowards(transform.rotation, target, RotationSpeed *
          (playerMovement.movementMultiplier + (1f - playerMovement.movementMultiplier) / 2) * Time.deltaTime);
    }
}
```

KUVIO 7. PlayerRotation-luokka

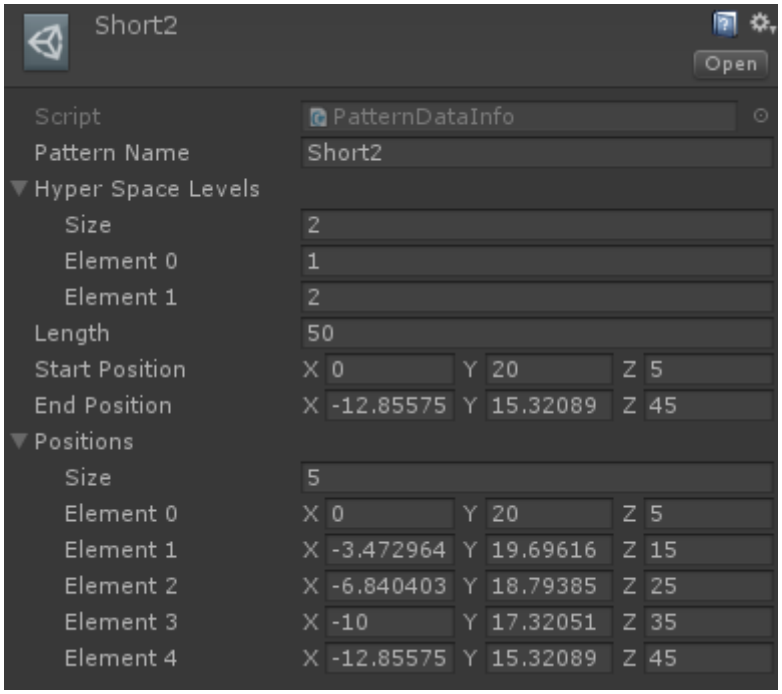
6 REITIN LUOMINEN PELAAJALLE

6.1 ScriptableObject

ScriptableObjectin päätarkoitus on toimia tallennustilaa säästävänä tiedostona, johon muut skriptit voivat viitata. ScriptableObjectiin viitatessa luettavia tietoja ei kopioida käytettävän skriptin peliobjektille, vaan ne luetaan suoraan ScriptableObjectista. ScriptableObjectista on mahdollista luoda asset-tiedosto *AssetDatabase.CreateAsset(UnityEngine.Object asset, string path)*-metodilla.

Unity ei sarjallista ScriptableObjectista luotuja assetteja samaan tapaan, kuin esimerkiksi luokkia, eli assetteihin voi tallentaa pelin ajonaikaisia tapahtumia ja ne säilyvät myös pelin loputtua. Esimerkiksi pelaajan sijainnin voi pelin loputtua tallentaa assettiin ja seuraavassa pelissä pelaaja voidaan asettaa tallennetulle sijainnille. Näin ScriptableObjectia hyödyntäen on luotu alkeellinen tallennusjärjestelmä peliin. (Unity3d docs 2018.)

Reitti editorissa tallennetaan assettiin reitin alku- ja loppupiste, pituus, nimi, reitillä olevien kalojen määrä ja sijainti vektorit sekä tasot, joilla kyseinen reitti ilmaantuu. (KUVIO 8.)



KUVIO 8. ScriptableObjectista luodun assetin näkymä Inspectorissa

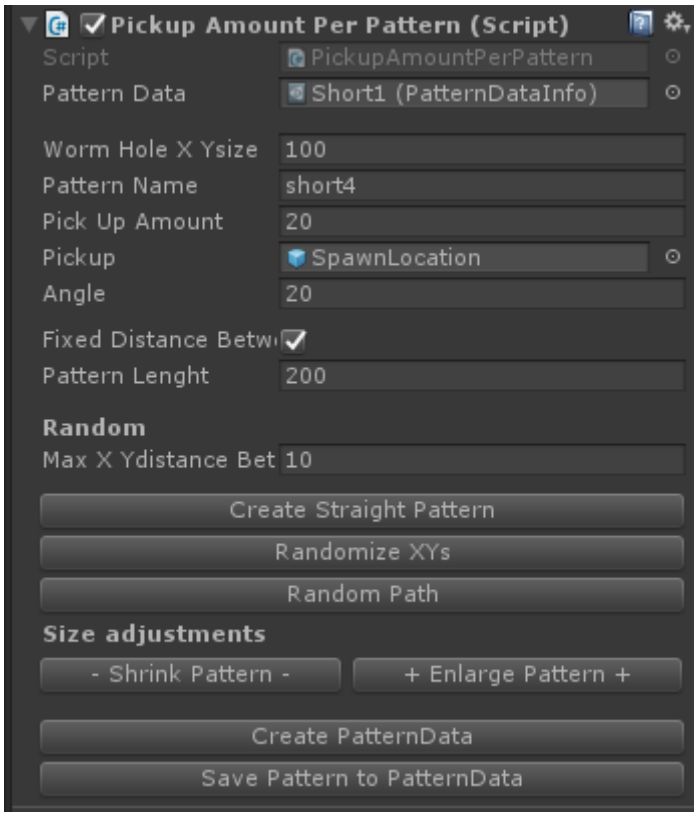
6.2 Muokattu editori

Unityssä editori skriptauksella on mahdollista tehdä omia työkaluja parantamaan työtehokkuutta. Editori skriptauksella siis muokataan tuttua Inspector näkymää. (KUVIO 10).

Editori skriptejä luotaessa tulee niiden periytyä Editor-luokasta, sekä käyttää UnityEditor-nimiavaruutta. Ennen skriptin luokan määritystä editorille on kerrottava, minkä tyyppiselle komponentille kyseinen editori on. GUILayout-luokasta löytyy kaikki muokatun editorin käyttöliittymään tarvittavat komponentit, kuten painikkeet ja editorin kokoon sekä jäsentelyyn käytettävät metodit. (KUVIO 9.)

```
using UnityEditor;
// Tämä kertoo editorille mille tyypille se on.
[CustomEditor(typeof(PickupAmountPerPattern))]
// Luokka periytyy Editor - luokasta.
public class ObjectBuilderEditor : Editor
{
    public override void OnInspectorGUI()
    {
        // Perus Inspector- näkymä.
        DrawDefaultInspector();
        GUILayout.Space(5);
        PickupAmountPerPattern patternScript = (PickupAmountPerPattern)target;

        if (GUILayout.Button("Create Straight Pattern"))
        {
            patternScript.CreatePattern();
        }
        if (GUILayout.Button("Create PatternData"))
        {
            // Luo PatternData asset.
            ScriptableObjectUtility.CreateAsset<PatternDataInfo>();
        }
        if (GUILayout.Button("Save Pattern to PatternData"))
        {
            patternScript.SavePattern();
        }
    }
}
```

KUVIO 9. Omien painikkeiden lisääminen Inspectoriin



KUVIO 10. Reitti-editorin käyttöliittymä

6.3 Reitti-editori

Reitti editori on visuaalinen työkalu, joka helpottaa reittien tekoa. Sen käyttöliittymänä on muokattu Inspector, joka luo ja tallentaa reitinosiot kuvio 8:n mukaisiin ScriptableObject asset-tiedostoihin. Asset-tiedosto on nimetty PatternData:ksi. Työkalulla tehdään lyhyitä reittiosioita, eli malleja, joista pelin ajonaikana muodostuu satunnaisessa järjestyksessä loputon reitti pelaajan kuljettavaksi. Mallit koostuvat lieriön sisällä olevista peräkkäisistä kuutioista, jotka kuvastavat pelin poimittavia kaloja. Kuutioiden sijainnit tallennetaan PatternData tiedostoihin, joiden pohjalta peli tekee loputtoman reitin pelaajalle.

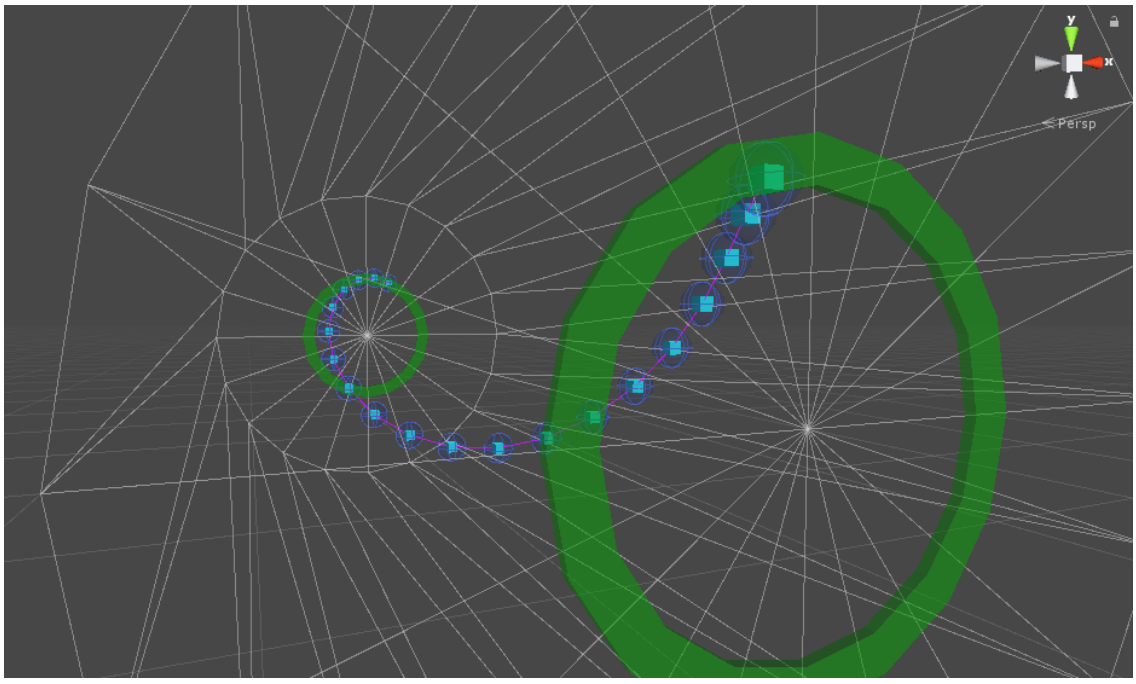
Työkalu löytyy omasta Pickup Pattern Testing -nimisestä Scenestä, jolloin Unityn scene-näkymässä ei ole mitään ylimääräistä reitti-editorin lisäksi. Reitille on päätetty muutamia ennalta määritettyjä attribuutteja, jotta pelin vaikeustaso pysyisi sopivana. Eräs näistä attribuuteista on kalojen välimatka toisiinsa. Tärkeimpänä valintana reitti-editorissa on yksittäisen reittiosion pituus, joka määrää myös poimittavien kalojen määrän. Pelissä kaloja on aina kymmenen yksikön välein, joten 120 yksikön pituisella reitillä on 12 poimittavaa kalaa. Editorissa on myös mahdollista itse päättää kalojen välinen etäisyys valitsemalla FixedDistanceBetweenPickups-boolean arvon epätodeksi. Angle-arvon ollessa joku muu

kuin nolla, luo editori vakio reitistä kaarevan, jättäen jokaisen kalan välille halutun kulman. Angle-arvon ollessa negatiivinen kääntyy reitti vastakkaiseen suuntaa. Kaarevuus lasketaan Sin- ja Cos-trigonometristen funktioiden avulla for-silmukassa. (KUVIO 11.)

```
for (int i = 0; i < pickUpAmount; i++)
{
    // Angle calculations.
    var x = radius * Mathf.Sin(angle * i * Mathf.Deg2Rad);
    var y = radius * Mathf.Cos(angle * i * Mathf.Deg2Rad);
    newPosition.x = x;
    newPosition.y = y;
}
```

KUVIO 11. Kaarevuuden laskeminen

Attribuutti valintojen jälkeen Create Pattern-painiketta painamalla editori luo halutun suoran tai kaarevan vakiomallin, tai täysin satunnaisen mallin, joka kuitenkin pysyy toivottujen rajojen sisäpuolella (KUVA 9). Nämä rajat tulevat WormHoleXYSize-muuttujasta, joka kertoo editorille lieriön halkaisijan, jonka sisälle malli rakentuu. Reittiosion vakiomallin luonnin jälkeen kaikkien sen osien sijaintia voi muokata. Mallin tallennus tapahtuu Save Pattern To PatternData -painiketta painamalla, jos reitti-editorin PatternData ei ole tyhjä.



KUVA 9. Kaareva malli ja gizmot Scene-näkymässä

6.4 Gizmo

Reittiosion luomista ja havainnollistamista helpottaa gizmot. Gizmot mahdollistavat visuaalisemman debuggauksen ja niiden koon, sijainnin, muodon ja värin voi itse päättää. Gizmot ovat näkymissä ainoastaan Scene-näkymässä. Kuvassa 9 olevassa reitti-editorissa valkoinen lieriö-gizmo kuvastaa reitin äärirajoja. Lieriö-gizmojen luomiseen ei löydy suoraa metodia samaan tapaan, kuin primitiivisempiin pallo ja kuutio muotoihin. Lieriö-gizmon, tai minkä vain muotoisen gizmon saa aikaiseksi alla näkyvän kuvion DrawWireMesh()-metodilla (KUVIO 12). Metodin mesh-parametriksi voi määrittää minkä tahansa 3D-mallin. Reitti-editorin tapauksessa mesh-parametriksi on määritetty yksinkertainen lieriömalli. Muut metodin parametrit voi tarpeen mukaan jättää asettamatta, sillä niille vakioarvot on jo asetettu. Reitti-editorin siniset pallo-gizmot helpottavat reitin näkyvyyttä, ja vihreät rengas-gizmot reitti-editorin päädyissä kertovat reitti-editorin tämänhetkisestä rajallisuudesta. Jotta useat eri reitti osiot saadaan yhdistettyä yhdeksi kokonaisuudeksi, tulee reittien ensimmäisen ja viimeisen poimittavan kalan sijainti olla vihreällä renkaalla.

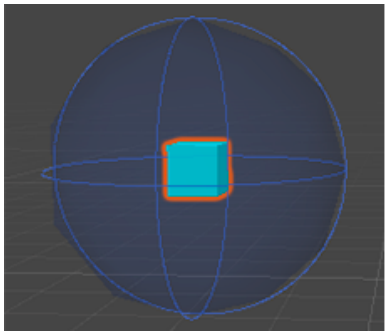
```
1   using UnityEngine;
2
3   public class CylinderGizmo : MonoBehaviour
4   {
5       [SerializeField] Mesh mesh;
6       [SerializeField] PatternEditor patternEditor;
7       private void OnDrawGizmos()
8       {
9           Vector3 zOffset = new Vector3(0, 0, patternEditor.patternLenght / 2);
10          Gizmos.DrawWireMesh(mesh, transform.position + zOffset, Quaternion.Euler(90f, 0, 0),
              new Vector3(patternEditor.wormHoleXYsize, patternEditor.patternLenght / 2,
              patternEditor.wormHoleXYsize));
11      }
12  }
```

KUVIO 12. Lieriö-gizmon luominen reitti-editoriin

Kuutiot ovat kaikki saman parent-peliobjektin alla, joka piirtää jokaisen lapsikuution välille DrawLine()-metodilla viivan. (KUVIO 13.) Gizmot toteutetaan MonoBehaviour-luokasta periytyvällä OnDrawGizmos()- tai OnDrawGizmosSelected()-metodeilla. OnDrawGizmos()-metodia kutsutaan jokaisella ruudunpäivityksellä, kun OnDrawGizmosSelected()-metodia kutsutaan vain kun peliobjekti, johon gizmo-skripti on liitetty on valittuna. Kuviossa 14 on esimerkkinä luotu sinisen värinen pallo-gizmo, jota ympäröi pallonmuotoinen rautalankamalli. (Unity3d docs2018.)

```
using UnityEngine;

public class LineBetweenChildren : MonoBehaviour {
    private void OnDrawGizmos()
    {
        for (int i = transform.childCount - 1; i >= 1; i--)
        {
            if (i > 0)
            {
                Vector3 from = transform.GetChild(i).localPosition;
                Vector3 to = transform.GetChild(i - 1).localPosition;
                Gizmos.color = new Color(0.8f, 0.1f, 1f, 0.8f);
                Gizmos.DrawLine(from, to);
            }
        }
    }
}
```

KUVIO 13. Viiva-gizmojen luominen peliobjektin lapsien välille

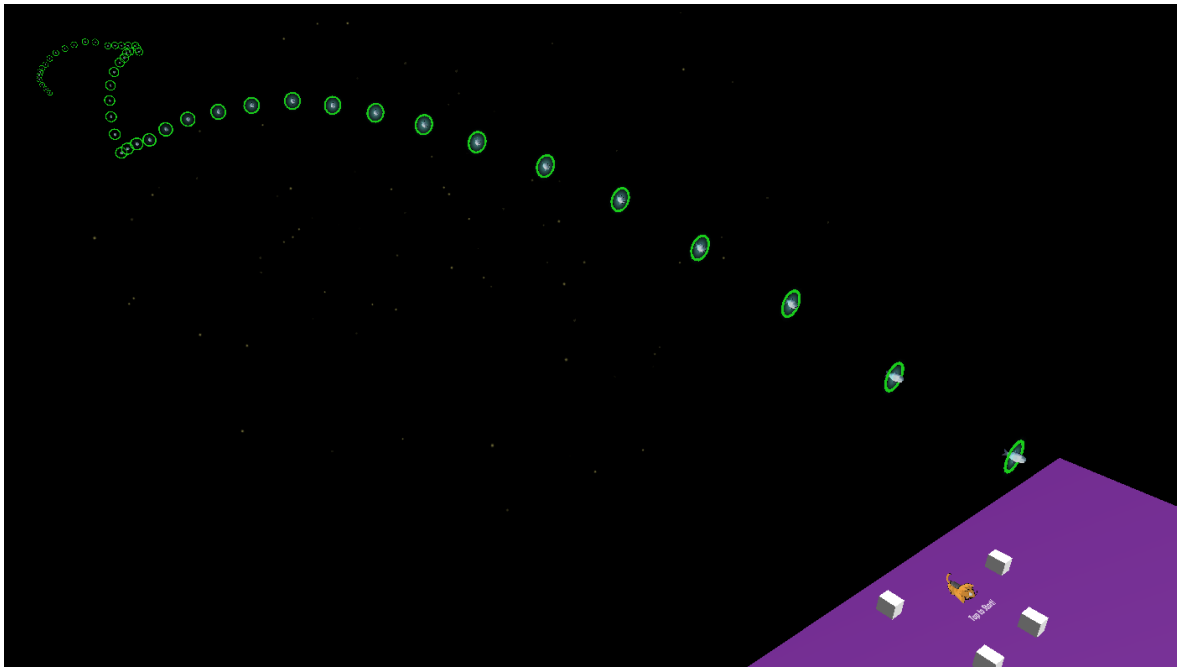


```
using UnityEngine;

public class SphereGizmo : MonoBehaviour {
    private void OnDrawGizmos()
    {
        Gizmos.color = new Color(0.3f, 0.4f, 0.8f, 0.8f);
        Gizmos.DrawWireSphere(transform.position, 3f);
        Gizmos.color = new Color(0.3f, 0.4f, 0.8f, 0.3f);
        Gizmos.DrawSphere(transform.position, 3f);
    }
}

```

KUVIO 14. Pallo-gizmon luominen rautalankamallin kanssa

7 KALA-GENERAATTORI

Kala-generaattori peliobjekti nimensä mukaisesti generoi, eli instansioi kaloja pelaajan kerättäväksi. Generoidut kalat muodostavat jatkuvan reitin. Kuvassa 10 on esimerkki generoidusta reitistä, joka koostuu neljästä reittiosiosta. Kala-generaattori peliobjektissa oleva FishSpawner komponentti huolehtii kalojen generoimisesta. Generoitavan kalan etäisyys pelaajasta on aina 250 yksikköä Z-akselin suuntaisesti. Generointi tapahtuu, kun pelaaja on liikkunut tulevan ja viimeisimmän kalan Z-akselin suuntaisen etäisyyden erotuksen verran. Nämä Z-akselin etäisyydet kuin myös X- ja Y-akselin suuntaiset sijainnit FishSpawner saa viitatuista PatternDatoista. PatternDatoja voi FishSpawnerissa olla rajattomasti ja mitä enemmän PatternDatoja on, sitä monipuolisempi ja ennalta-arvaamattomampi pelistä tulee.



KUVA 10. Neljästä eri mallista luotu reitti

Eri mallien yhdistäminen vaatii muutakin kuin seuraavan mallin siirtämisen edellisen eteen Z-akselin suuntaisesti. Tällöin reitistä ei tule jatkuva, mallien liitoskohdat olisivat selkeästi näkyvissä, jolloin reitti ei ole yhtenevä. Seuraavaa mallia ei voi myöskään laittaa alkamaan edellisen mallin viimeisen kalan sijainnista koska tällöin reitti ajautuisi ennemmin tai

myöhemmin ulos madonreiästä. Reitti olisi silloin jatkuva, mutta pelaajan edistyminen olisi mahdotonta.

Jotta reitistä tulisi täydellinen, tulee mallien vaihtuessa laskea edellisen viimeisen kalan sijainnin ja seuraavan ensimmäisen kalan Vector2 sijainnin välinen kulma. Tulevan mallin kaloja kierretään lasketun kulman verran Z-akselin ympärillä. (KUVIO 15.) (KUVIO 16.)

```
// Palauttaa kahden Vector2 vektorin välisen kulman.
float AngleBetweenVector2(Vector2 from, Vector2 to)
{
    // Kulman suunta.
    Vector2 vec1Rotated90 = new Vector2(-from.y, from.x);
    float sign = (Vector2.Dot(vec1Rotated90, to) < 0) ? -1.0f : 1.0f;
    // Kulman laskeminen.
    if (from != to)
        return Vector2.Angle(from, to) * sign;
    else return 0f;
}
```

KUVIO 15. Kahden Vector2-tyyppisen vektorin välisen kulman laskeminen.

```
transform.RotateAround(new Vector3(0, 0, 0), Vector3.forward,
    AngleBetweenVector2(currentStartPosition, previousEndPosition));
```

KUVIO 16. Peliobjektin kiertäminen origon ympäri, Z-akselin suuntaisesti lasketun kulman verran

Kalojen kierrettyä lasketun kulman verran Z-akselin ympäri mallien liitoskohdat ovat lähes huomaamattomat. Tämän kiertämisen takia mallien tekovaiheessa oli pidettävä huolta, että mallin ensimmäisen ja viimeisen kalan sijainnit ovat aina samalla etäisyydellä X- ja Y-akselin leikkauspisteestä.

Viimeisenä vaiheena on nykyisen mallin viimeisen kalan ja seuraavan mallin ensimmäisen kalan yhdistäminen yhdeksi. Näin vältetään mallien liitoskohdassa olevien kahden samalla Vector2-sijainnilla olevien kalojen esiintyminen. Tämä tapahtuu yksinkertaisuudessaan niin, että vaihdon tapahtuessa ohitetaan tulevan mallin ensimmäinen indeksi ja liikutetaan tulevaa mallia sen ensimmäisen ja toisen kalan Z-akselin etäisyyden verran taaksepäin.

Käytettävissä olevat mallit vaihtuvat aina hyperavaruus-tason mukaan. FishSpawnerissa on PatternDataInfo[] currentPatterns-taulukko, jonka sisältö päivittyy hyperavaruustason vaihtuessa sille tasolle suunniteltuihin malleihin. Mitä korkeammalle tasolle pelaaja etenee, sitä haastavampia malleja pelaaja kohtaa. EnergyControllerin OnDifficultyChange(int level)-delegaattiin on lisätty kuuntelijaksi FishSpawner-luokan ChangePatterns(int level)-metodi. OnDifficultyChange(int level)-metodia kutsutaan, kun pelaaja on kerännyt tarvittavan määrän kaloja, tai menettänyt liikaa energiaa, jolloin hyperaaruustason vaihto tapahtuu. Vaihdon tapahtuessa ChangePatterns(int level)-metodi suoritetaan ja currentPatterns PatternData-taulukko korvataan ChangePatterns()-metodin ensimmäisen parametrin viittaamalla PatternData-taulukolla. Reitin jatkuvuuden varmistamiseksi nykyisen mallin generoimista ei jätetä kesken hyperavaruustason vaihdon tapahtuessa. Vaihdos tapahtuu, kun nykyinen malli on kokonaan generoitu. (KUVIO 17.)

```
110     void ChangePatterns(int level) {
111         switch (level)
112         {
113             case 1:
114                 currentPatterns = HS1Patterns;
115                 break;
116             case 2:
117                 currentPatterns = HS2Patterns;
118                 break;
119             case 3:
120                 currentPatterns = HS3Patterns;
121                 break;
122             case 4:
123                 currentPatterns = HS4Patterns;
124                 break;
125         }
126     }
```

KUVIO 17. FishSpawner-luokan ChangePattern()-metodi

8 ENERGYCONTROLLER

Kuljetun matkan lisäksi pelin tärkein mittari on pelaajan energia, joka kasvaa sen kerätessä kaloja. Kerääminen tapahtuu pelaaja peliobjektin ja kala peliobjektin törmätessä. Yhdestä kalasta saatava energiamäärä kasvaa kymmenellä prosentilla pelaajan kerätessä peräkkäisiä kaloja. Ohittaessa yhden kalan pelaajan energiakerroin nollaantuu.

Energiatasoa ja pelin etenemistä seuraa GameController-peliobjekti, johon on tehty oma EnergyController-komponentti. EnergyController pitää pelaajan energiatason ajan tasalla ja päivittää pelin käyttöliittymän sen mukaisesti. Pelaajan energiatasosta riippuen EnergyController muuttaa hyperavaruus tasoja ja vaikeustasoa.

8.1 ChangeEnergy

ChangeEnergy()-metodi muuttaa pelaajan energiamäärää (energy) sen parametrin osoittaman määrän verran. Ennen lisäystä parametri kerrotaan sen hetkisellä energiakertoimella (energyMultiplier), jonka suuruus on sitä isompi, mitä enemmän pelaaja on peräkkäisiä kaloja kerännyt. (KUVIO 18.)

```
176
177     public void ChangeEnergy(float value)
178     {
179         energy += value * energyMultiplier;
180         if (energy >= currentHyperspaceLevel * energyPerHyperspaceLevel)
181         {
182             NextHyperspaceLevel();
183         }
184     }
185
```

KUVIO 18. ChangeEnergy()-metodi

Jokaiselle kalalle asetetaan syntyessään oma id, joka on yhden isompi kuin edellisellä. Pelaajan kerätessä kaloja vertaillaan edellisen ja seuraavan kalojen id arvoja, jos id arvojen erotus on yksi (1), energiakerrointa kasvatetaan kymmenellä prosentilla.

ChangeEnergy()-metodin lopussa tarkistetaan if-lauseella, onko saavutettu tarvittava energiamäärä seuraavaa hyperavaruus tasoa varten. Vaadittava energiamäärä seuraavaan tasoon saadaan kertomalla nykyisen hyperavaruus tason numero energiamäärällä hyperavaruus tasoa kohden (KUVIO 18, rivi 180). Energiamäärä hyperavaruustasoa

kohden on ennalta määritetty sata (100). Energiamäärän ollessa tarpeeksi iso if-lause on tosi ja kutsutaan NextHyperSpace()-metodia. (KUVIO 18.)

8.2 NextHyperspaceLevel & PreviousHyperspaceLevel

NextHyperSpace()-metodin alussa tarkistetaan, onko seuraavaa hyperavaruus tasoa olemassa, eli ollaanko jo saavutettu neljäs taso (KUVIO 19). Mikäli neljättä tasoa ei ole saavutettu tapahtuu seuraavaa.

CurrentHyperspaceLevel arvoa nostetaan yhdellä, jotta pelilogiikka tietää, missä hyperavaruustasossa parhaillaan ollaan. Update()-metodissa pelaajalta vähenee energiaa tietyn verran jokaisen ruudunpäivityksen aikana, minkä seurauksena pelaajan energiataso saattaisi tippua heti edelliselle tasolle tason nousun jälkeen. Tämän epäreilun pelitilanteen välttämiseksi pelaajan energiatasoa nostetaan kolmanneksen energyPerHyperSpaceLevel määrästä. Samalla pelaaja saa aikaa hengähtää ja mahdollisuuden mukautua uuden hyperavaruus tason tuomiin vaaroihin ja muutoksiin.

Pelaaja peliobjektiin on kiinnitetty SpinToNextHyperSpace-komponentti, jonka StartSpin() -metodia kutsutaan. Metodi aloittaa animaation, jossa Player peliobjekti pyörähtää 360 astetta hidastetusti. Pyörähdyksen puolessa välissä kutsutaan MusicManager.Singleton.NextSong()-metodia, joka häivyttää nykyisen hyperavaruus tason kappaleen seuraavan tason kappaleeseen.

Lopulta kutsutaan EnergyControllerin SetDifficulty()-metodia, joka päivittää pelin vaikeusasteen nykyisen hyperavaruus tason mukaiseksi. SetDifficulty()-metodia kutsutaan aina hyperavaruus tason vaihtuessa.

PreviousHyperspaceLevel()-metodi poikkeaa hieman NextHyperspaceLevel()-metodista. Tason lasku tapahtuu välittömästi ilman pelaajan pyörähdys animaatioita ja energiataso pidetään muuttumattomana. Laskua ei tapahdu, jos pelaaja on jo tasolla yksi. (KUVIO 19.)

```
152
153     void NextHyperspaceLevel()
154     {
155         if (currentHyperspaceLevel < 4)
156         {
157             currentHyperspaceLevel++;
158             energy += (energyPerHyperspaceLevel * 0.33f);
159             spinToNextHyperspace.StartSpin();
160             SetDifficulty();
161         }
162     }
163     void PreviousHyperspaceLevel()
164     {
165         if (currentHyperspaceLevel > 1)
166         {
167             currentHyperspaceLevel--;
168             MusicManager.Singleton.PreviousSong();
169             SetDifficulty();
170         }
171     }
172
```

KUVIO 19. Metodit NextHyperspaceLevel() ja PreviousHyperSpaceLevel()

8.3 SetDifficulty

Vaikeustaso vaihtuu aina hyperavaruus tason muuttuessa. Seuraavat esimerkit kertovat metodin toiminnan, kun hyperavaruus taso nousee. Laskiessa toiminta olisi päinvastainen.

SetDifficulty()-metodi kutsuu aluksi SetCameraProperties()-metodia, joka asettaa kameran näkökentän suuremmaksi tuoden pelaajalle vauhdin kasvun tuntua. Metodi muuttaa myös kameran etäisyyttä pelaajasta. (KUVIO 20.)

Pelaaja peliobjektin nopeus saadaan muutettua siinä kiinni olevan PlayerMovement-komponentin nopeuskerrointa (hyperSpaceSpeedMultiplier) muuttamalla. Arvoksi asetetaan hyperSpaceSpeedMultiplier-taulukon currentHyperspaceLevel-1:n alkion osoittama määrä. Nopeuskertoimen muutos on nähtävissä kuviossa 20, rivillä 125. Tämä taulukko, kuin myös muut EnergyControllerista löytyvät hyperavaruus tasoihin liittyvät taulukot on merkitty [SerializeField]-property määreellä. Tämä pakottaa Unity Editorin serialisoimaan private määreellä varustetut muuttujat (Unity3d docs 2018). Normaalisti Unityn Inspectorissa näkyvät vain public määreellä varustetut muuttujat, mutta [SerializeField] tuo privaatit muuttujat Inspectoriin näkyviin. Kaikki vaikeustasoihin liittyvät taulukot näkyvät Unityn Inspectorissa ja niiden arvot on asetettu käsin skriptin ulkopuolella.

Hyperavaruustason noustessa pelaajan energiankulutus nousee. Energiankulutus nousee 0,3 verran jokaisella tason nousulla. Pelaajan nopeuden noustessa pystyy se nopeampaa

tahtia keräilemään kaloja, joten energiankulutusta nostamalla saadaan vaikeustaso säilymään vähintään yhtä vaikeana. RenderSettings.ambientLight arvoksi asetetaan väri, joka luo pelimaailmaan sen mukaisen tunnelmavalaistuksen. Hyperavaruus tasoille on ennalta päätetty omat väriteemat, vihreä, sininen, punainen ja liila.

```
122     void SetDifficulty()
123     {
124         SetCameraProperties();
125         playerMovement.hyperSpaceSpeedMultiplier = hyperspaceSpeedMultipliers[currentHyperspaceLevel - 1];
126         reductionMultiplier = (currentHyperspaceLevel > 1) ? 1 + ((currentHyperspaceLevel - 1) * 0.3f) : 1f;
127         RenderSettings.ambientLight = hyperspaceAmbientColors[currentHyperspaceLevel-1];
128         if (OnDifficultyChange != null)
129             OnDifficultyChange(currentHyperspaceLevel);
130     }
131
```

KUVIO 20. SetDifficulty()-metodi

Lopulta SetDifficulty()-metodi kutsuu OnDifficultyChange-delegaatin ja antaa sen parametriksi currentHyperspaceLevel. Delegaatin tilanneet skriptit saavat parametrin kera ilmoituksen hyperavaruus tason vaihdosta ja voivat toimia haluamallaan tavalla tämän tiedon kanssa. Esimerkiksi nopeusmittari päivittää tuntinopeutensa, käyttöliittymä muuttaa itsensä hyperavaruus tason teemaan sopivaksi, kala generaattori muuttaa käytössä olevia malleja ja uusia esteitä alkaa syntymään pelaajan tielle.

9 MATKAMITTARI

Pelissä kuljettua matkaa laskeva matkamittari kasvaa jatkuvasti Kisun ollessa liikkeessä. Pelikerta kohtainen matka näkyy pelaajalle ruudun alalaidassa. Tämä mittari nollaantuu jokaisen pelikerran alkaessa. Toinen matkamittari seuraa Kisun kaikilla pelikerroilla kulkemaa yhteenlaskettua matkaa. Tämä mittari toimii pelin taustalla ja tulee näkyviin ruudun vasempaan laitaan pelin alussa ja seuraavia kohteita saavuttaessa.

9.1 DistanceHelper

DistanceHelper-luokka eroaa aikaisemmin esittelemistä luokista siten, että se ei periydy mistään luokasta. Se tarjoaa staattisia metodeja, joita muut luokat voivat ajonaikana suorittaa, kuten GetDistance()-metodin, joka palauttaa sitä kutsuvalle luokalle sillä pelikerralla kuljetun matkan string-muodossa. Julkisen GetDistance()-metodin lisäksi DistanceHelper-luokka pitää sisällään julkisen SetMeter(float amount)- ja ResetDistance()-metodin, sekä privaatin Check()-metodin.

9.1.1 Muuttujat

Astronomiset mittayksiköt ovat niin isoja, ettei niitä ole järkevä kuvailla kilometreillä. Tästä syystä DistanceHelper-luokka omaa kuusi eri mittayksikköä omina muuttujinaan, jotka ovat nähtävissä kuviossa 21. Jokaisella mittayksiköllä on property määritetty public getterillä ja privaatilla setterillä. Public getterin avulla muuttujaa voi ulkopuoliset luokat lukea, mutta privaatti setter estää muuttujan asettamisen oman luokkansa ulkopuolella. (KUVIO 21.)

```
12
13      static float km;    // Kilometri.
14      static float r;     // Maapallon säde.
15      static float ld;    // Lunar Distance.
16      static float au;    // Astronominen yksikkö.
17      static float ly;    // Valovuosi.
18      static float pc;    // Parsek.
19
20      public static float Km { get; private set; }
21      public static float R  { get; private set; }
22      public static float Ld { get; private set; }
23      public static float Au { get; private set; }
24      public static float Ly { get; private set; }
25      public static float Pc { get; private set; }
26
```

KUVIO 21. Astronomiset pituusyksiköt

Muut luokat pystyvät lukemaan haluamansa muuttujan kutsulla DistanceHelper.Km, mutta muuttujan arvon asetusta ei sallita. DistanceHelper luokan ollessa saantimääreeltään public ja propertyjen ollessa public static ei tarvitse DistanceHelper luokkaa instansioida ennen sen käyttöä.

9.1.2 SetMeter ja Check

Kuviossa 22 näkyvä SetMeter()-metodi ottaa parametrikseen amount muuttujan, joka lisätään Km muuttujaan eli pelikertaiseen kilomerimäärään. Lisäyksen jälkeen SetMeter()-metodi kutsuu privaattia Check()-metodia, joka tarkistaa, onko seuraava pituusyksikkö jo saavutettu. (KUVIO 22.)

```
27
28      public static void SetMeter(float amount)
29      {
30          Km += amount;
31          Check();
32      }
33
```

KUVIO 22. SetMeter()-metodi

Jokaiselle mittayksikölle laskettiin oma suhdelukunsa. Omalla suhdeluvulla kerrottuna mittayksikkö on niin suuri että on saavutettu yksi (1) kappale seuraavaa mittayksikköä. Esimerkkinä metrin suhdeluku olisi tuhat (1000), sillä 1000 * 1 metri = 1 kilometri. Laskettuja astronomisia suhdelukuja hyödynnetään pyöristäessä suuria kilometrimääriä siistimmäksi. alla kmToR siis tarkoittaa kilometri määrää, joka on yhtä suuri kuin maapallon halkaisija (R). (KUVIO 23.)

```
3
4       const float kmToR = 6371f;
5       const float RToLd = 60.33621095589389f;
6       const float LdToAu = 389.5776885259f;
7       const float AuToLy = 63241.07708807f;
8       const float LyToPc = 3.261563776944f;
9
```

KUVIO 23. Astronomisille mittayksiköille lasketut suhdeluvut

Check()-metodi tarkistaa, onko seuraavan mittayksikön verran kuljettu, jos on, korotetaan seuraavan mittayksikön arvoa yhdellä ja nollataan nykyinen mittayksikkö. (KUVIO 24.)

```
61
62      static void Check()
63      {
64          if (Km > kmToR)
65          {
66              Km = Km - kmToR;
67              R++;
68          }
69          if (R > RToLd)
70          {
71              R = R - RToLd;
72              Ld++;
73          }
74          if (Ld > LdToAu)
75          {
76              Ld = Ld - LdToAu;
77              Au++;
78          }
79          if (Au > AuToLy)
80          {
81              Au = Au - AuToLy;
82              Ly++;
83          }
84      }
85
```

KUVIO 24. Check()-metodi

9.1.3 GetDistance ja ResetDistance

GetDistance()-metodi palauttaa sitä kutsuvalle luokalle pelikertaisen kuljetun matkan muotoiltuna tekstinä. Metodi tarkistaa, mitkä luokan mittayksiköistä on jo kerran saavutettu ja tämän mukaan asettaa palautettavalle tekstille oikeat yksiköt. Esimerkiksi, jos on jo kuljettu maapallon säteen verran, palautettava teksti muodostuu R ja KM mittayksiköistä. (KUVIO 25.)

```
33
34      public static string GetDistance()
35      {
36          string distance = "0";
37
38          if (Ly > 0)
39          {
40              distance = string.Format("{0:#}LY {1:#} AU", Ly, Au);
41          }
42          else if (Au > 0)
43          {
44              distance = string.Format("{0:#}AU {1:#} LD", Au, Ld);
45          }
46          else if (Ld > 0)
47          {
48              distance = string.Format("{0:#}LD{1:0#}R", Ld, R);
49          }
50          else if (R > 0)
51          {
52              distance = string.Format("{0:#}R{1:000#}KM", R, Km);
53          }
54          else
55          {
56              distance = string.Format("{0:0000000#}KM", Km);
57          }
58
59          return distance;
60      }
61
```

KUVIO 25. GetDistance()-metodi

ResetDistance()-metodia kutsutaan GameControllerin puolesta aina, kun uusi pelikerta aloitetaan. ResetDistance()-metodi yksinkertaisesti asettaa jokaisen staattisen mittayksikkö muuttujan arvoksi nolla (0). Näin varmistetaan, että vanhat tulokset eivät tule pelikerta kohtaiseen pistelaskuun mukaan. Mittayksiköiden ollessa staattisia, ne eivät nollaannu pelin ladatessa Scene uudelleen. (KUVIO 26.)

```
85
86      public static void ResetDistance()
87      {
88          Pc = 0;
89          Ly = 0;
90          Au = 0;
91          Ld = 0;
92          R = 0;
93          Km = 0;
94      }
95  }
96
```

KUVIO 26. ResetDistance()-metodi

9.2 Matkamittarin toteutus

Matkamittari (DistanceMeter) peliobjekti sisältää samannimisen komponentin, joka käyttää DistanceHelper-luokan tarjoamia metodeja kuljetun matkan päivittämiseen ja näyttämiseen pelaajalle. DistanceMeter-komponentissa on Moving-muuttuja, joka on tyypiltään boolean. Sen arvo on tosi aina pelaajan ollessa liikkeessä. Törmätessä pelaaja pysähtyy ja samalla muutetaan Moving arvon epätodeksi. Moving boolean arvon ollessa tosi kutsuu DistanceMeter Update()-metodissaan äsken läpikäytyä DistanceHelper.SetMeter()-metodia, parametrinä pelaajan sen hetkinen nopeus kmPerSec. DistanceHelper.GetDistance()-metodi palauttaa pelikertaisen kuljetun matkan ja se asetetaan pelaajan näkyviin remainderText teksti komponentin text-muuttujan arvoksi. Text-komponentti remainderText on käyttöliittymä elementti, joka on merkitty DistanceMeter luokassa [SerializeField]-property määreellä ja asetettu Inspectorista osoittamaan haluttuun käyttöliittymän Text-komponentin omaavaan peliobjektiin. (KUVIO 27.)

```
50
51      void Update()
52      {
53          if (Moving)
54          {
55              DistanceHelper.SetMeter(Time.deltaTime * kmPerSec);
56              remainderText.text = DistanceHelper.GetDistance();
57          }
58      }
59
```

KUVIO 27. DistanceMeter.Update()-metodi

Nopeus kmPerSec ei tule suoraan pelaajan näennäisestä nopeudesta, vaan on se todellisuudessa huomattavasti pelaajan nopeutta suurempi. Mittarin nopeus on erikseen asetettu DistanceMeter-luokassa distanceSpeed-taulukossa. Taulukossa on neljä (4) alkiota, jokaiselle hyperavaruus tasolle on oma mittarinopeutensa. DistanceMeter-luokan OnEnable()- ja OnDisable()-metodeissa on huolehdittu EnergyController.OnDifficultyChange-delegaatin tilauksesta DistanceMeter.SpeedChange()-metodille. OnDifficultyChange tapahtui aina vaikeustason vaihtuessa hyperavaruus tason muutoksen myötä ja sen tilanneet metodit aktivoituvat samalla. Kuvio 29:n SpeedChange()-metodi muuttaa SetMeter()-metodissa käytetyn parametrin kmPerSec arvoa delegaatin osoittaman distanceSpeed-taulukon alkion arvoksi. Näin mittarinopeus on päivitetty sille hyperavaruus tasolle tarkoitetulle nopeudelle. (KUVIO 28.)

```
41
42      void OnEnable() { EnergyController.Singleton.OnDifficultyChange += SpeedChange; }
43      void OnDisable() { EnergyController.Singleton.OnDifficultyChange -= SpeedChange; }
44
```

KUVIO 28. Delegaatin tilaus ja peruminen

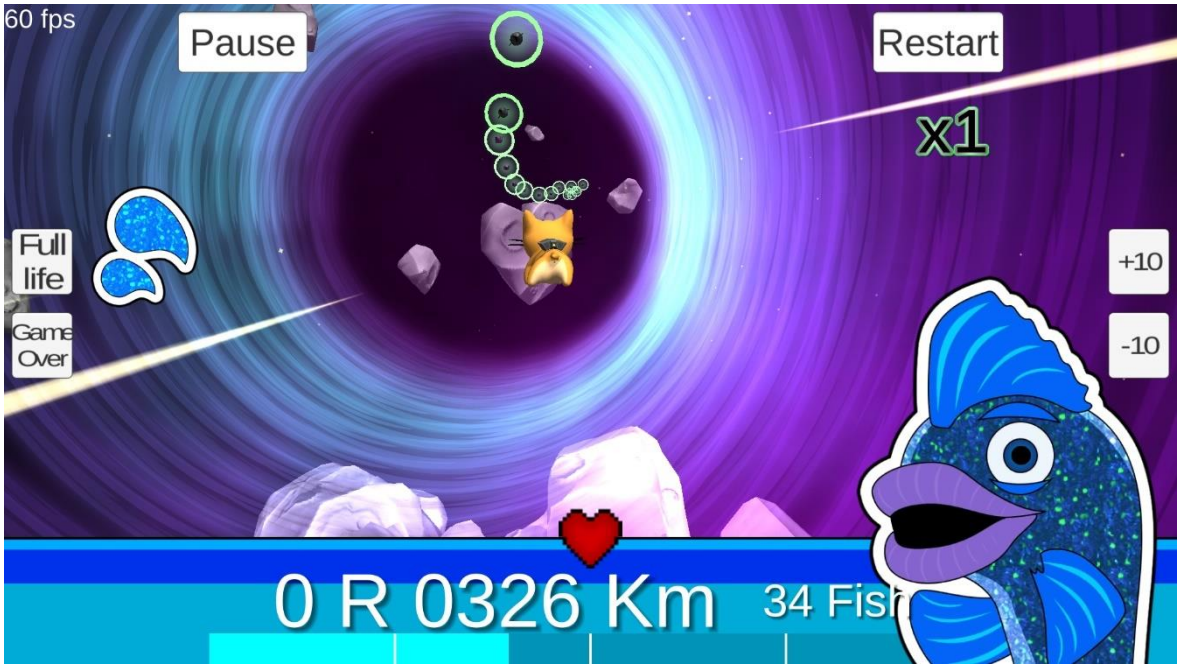
```
59
60      public void SpeedChange(int level)
61      {
62          kmPerSec = distanceSpeed[level - 1];
63      }
64
```

KUVIO 29. Delegaatin tilannut SpeedChange()-metodi parametrillä

10 YHTEENVETO

Mobiilipeli Kisu saatiin viidessä kuukaudessa suljettuun beta-testivaiheeseen Googlen Play Storeen. Alkuperäisestä prototyypistä ei tullut lopulliseen tuotokseen muuta, kuin sen tarjoama pelin idea. Mobiilipeli on ollut testaajien mielestä hieno ja mielenkiintoinen. (KUVA 11.)

Tässä opinnäytetyössä läpikäydyt pelinkehityksen osa-alueet olivat tavoitteisiin nähden riittävät, vaikka muutamia asioita olisi voinut tehdä paremmin. Reitti editori toimii hyvin ja se tarjoaa kenttäsuunnittelijalle työkalun rakentaa jokaiselle hyperavaruus tasolle omat reittinsä. Jatkokehitettävää reitti-editorille jäi vielä suuremman vapauden tarjoaminen reittejä luodessa. Tähän peliin reitti-editori oli riittävä ja pelin tiivis kehitystiimi hyväksyi rajoitteet täysin.



KUVA 11. Kuvakaappaus pelin ulkoasusta ennen beta-testivaihetta

Pelaajan liikkuminen 3d-maailmassa saatiin hyvän tuntuiseksi ja tarkaksi. Vaikka peliin laitettiin asetukset, joita muokkaamalla pelaaja voi vaikuttaa liikkumisen nopeuteen ja peilata kääntyvyyden suunnan, oli osalla pelitestaajilla vaikeuksia ymmärtää, miten ohjaus toimii. Aikaa oli rajatusti viisi kuukautta, enemmällä ajalla peliin olisi ollut hyvä kehittää lyhyehkö opastus ennen ensimmäistä pelikertaa, miten peliä pelataan ja mitkä ovat pelin tavoitteet.

Opinnäytetyöstä jäi pois useita peliin kehitettyjä ominaisuuksia. Näitä ominaisuuksia on esimerkiksi elämälogiikka, madonreiän optimoitu ulkoasu, hyperavaruustaso-kohtainen esteiden luonti ja dynaaminen musiikki eri hyperavaruustasojen välillä. Lisäksi pelissä oli Unity Ads -mainokset ja Unity Analytics -pelaajatilastojärjestelmä. Mainoksia katsomalla pelaaja ansaitsee lisää elämiä, ja pelaajatilastojärjestelmä kerää pelaajien suorituksista hyödyllistä tietoa jatkokehitystä varten.

Opinnäytetyö antoi selkeän viitteen siitä mihin nykypäivän mobiililaitteet pystyvät. Graafisesti näyttävät ja raskaat partikkeli systeemit, sekä paljon prosessointitehoa vaativat useat samanaikaiset fysiikkalaskelmat eivät tuota uusille mobiililaitteille ongelmia.

Mobiilipeli Kisu on toistaiseksi () beta-testivaiheessa, mutta sen kehitys jatkuu Tingleware Oy:n puolesta.

LÄHTEET

Tingleware.com 2018. Tingleware Oy [viitattu 26.5.2018]. Saatavissa: http://www.tingleware.com/

Unity3d 2018. Company Facts [viitattu 28.1.2018]. Saatavissa: https://unity3d.com/public-relations

Unity3d blogs 2018. UnityScript's long ride off into the sunset [viitattu 6.5.2018]. Saatavissa: https://blogs.unity3d.com/2017/08/11/unityscripts-long-ride-off-into-the-sunset/

Unity3d docs 2018. Components [viitattu 9.5.2018]. Saatavissa: https://docs.unity3d.com/Manual/Components.html

Unity3d docs 2018. Creating Components [viitattu 9.5.2018]. Saatavissa: https://docs.unity3d.com/Manual/CreatingComponents.html

Unity3d docs 2018. Gizmos [viitattu 5.5.2018]. Saatavissa: https://docs.unity3d.com/ScriptReference/Gizmos.html

Unity3d docs 2018. MeshRenderer [viitattu 9.5.2018]. Saatavissa: https://docs.unity3d.com/Manual/class-MeshRenderer.html

Unity3d docs 2018. Rigidbody.velocity [viitattu 3.5.2018]. Saatavissa: https://docs.unity3d.com/ScriptReference/Rigidbody-velocity.html

Unity3d docs 2018. ScriptableObject [viitattu 29.1.2018]. Saatavissa: https://docs.unity3d.com/530/Documentation/Manual/class-ScriptableObject.html

Unity3d docs 2018. Skinned Mesh Renderer [viitattu 31.1.2018]. Saatavissa: https://docs.unity3d.com/530/Documentation/Manual/class-SkinnedMeshRenderer.html

Unity3d docs 2018. Unity Collaborate [viitattu 19.3.2018]. Saatavissa: https://docs.unity3d.com/Manual/UnityCollaborate.html

Unity Products 2018. Unity Personal [viitattu 28.1.2018]. Saatavissa: https://store.unity.com/products/unity-personal

Unity3d Teams 2018. Unity Teams [viitattu 19.3.2018]. Saatavissa: https://unity3d.com/teams