KARELIA-AMMATTIKORKEAKOULU
Tietojenkäsittelyn koulutusohjelma

Timo Hämäläinen

Pelaajakontrollit top-down-scrollereissa

Opinnäytetyö
Maaliskuu 2015

Karelia AMMATTIKORKEAKOULU

OPINNÄYTETYÖ
Maaliskuu 2015
Tietojenkäsittelyn koulutusohjelma

Karjalankatu 3
80200 JOENSUU
p. 013 260 600

Tekijä
Timo Hämäläinen

Nimeke
Pelaajakontrollit Top-down-scrollereissa

Toimeksiantaja
-

Tiivistelmä

Opinnäytetyön aihe on pelaajakontrollit top-down-scrollereissa ja työn käytännön osuutena toimii Space shooter-peliprojekti. Käytännön osuudessa käsitellään lisäksi peliprojektiin tehtyä koodia. Opinnäytetyössä käydään läpi kaksi käytännön työhön liittyvää peligenreä Shoot 'em up ja Vertical scroller sekä kaksi 2D-kuvakulmaa, top-down ja 2.5D.

Peli on toteutettu käyttäen työkaluina Unity3D:tä ja FingerGesturea. Unity3D on helppokäyttöinen ja hyvin dokumentoitu pelimoottori. FingerGesture on Unity 3D:hen lisättävä (Unity Asset) työkalu, joka tarjoaa helpot työkalut tehdä kosketusnäytölle ja hiirelle sopivia kontrolleja.

Työn lopputuloksena syntyi julkaistava peliprojekti sekä toimivat pelaajakontrollit top-down-scroller-peliin.

Kieli

suomi

Sivuja 43

Asiasanat

pelaajakontrollit, Unity3D, FingerGesture, 2D-kuvakulma, 3D-kuvakulma

Karelia
UNIVERSITY OF APPLIED SCIENCES

THESIS
March 2015
Degree Programme in Business Information Technology

Karjalankatu 3
80200 JOENSUU
p. 013 260 600

Author (s)
Thesis coordination group and KUAS

Title
Pelaajakontrollit Top-down-scrollereissa

Commissioned by
-

Abstract

The topic of the thesis is player controls in top-down-scrollers and as the practical part of the work, acts the Space shooter game project. In the practical part the code from the game project is examined. The thesis introduces two practical work-related game genres: Shoot ‘em up and Vertical scroller as well as two 2D-view angles: top-down and 2.5D.

The game is implemented by using Unity3D and FingerGesture tools. Unity3D is a user friendly and well-documented game engine. FingerGesture is a Unity3D plug-in (Unity Asset), which provides simple tools to design touch screen and mouse suitable controls.

The end result was a published game project and functional player controls to a top-down-scroller game.

Language

Finnish

Pages 43

Keywords

Player controls, Unity3D, FingerGesture, 2D-camera angle, 3D-camera angle

Sisältö

Käsitteet

Arcadepeli:	Arcadepeli (tai peliautomaatit) on kolikoilla toimiva julkinen pelikone, joita löytyy yleensä ravintoloista, baareista ja erityisesti pelihalleista. (Wikipedia 2015a).
Genre:	Tyylilaji, tyylisuunta eli genre tarkoittaa jonkin taiteenalan lajia, luokkaa, tyyppiä tai esitystapaa.
Parallax scrolling:	Parallax on tiettyjen taustakuvan osien itsenäinen liike taustalla. Parallax scrolling on näiden osien sivuttain tai pystyyn tapahtuva vierivä liike. (New Straits Times 1998).
Run-and-gun:	Shoot ‘em up -peli, jossa pelaaja taistelee liikkuen jalan ja mahdollisesti omaa kyvyn hypätä.
On-rails:	Eteenpäin automaattisesti vierivä peli, jossa pelaaja voi joskus valita mitä reittiä pelaaja kulkee.
Raycast:	Unityn metodi joka laskee viivan pisteestä A, pisteeseen B ja tarkastaa osuuko viivaan mikään pisteen A ja B välillä. Metodi palauttaa arvon osumasta.

1 Johdanto

Opinnäytetyön aiheena on pelaajakontrollit top-down-scrollereihin ja työn käytännön osuutena toimii Oasis-hautomolla toteutettu Space shooter -peli. Opinnäytetyön aihe tuli suoraan työharjoittelussa tehdystä peli projektista, jonka toteutimme 9 henkilön voimin. Tähän kyseiseen projektiin toteutin pelaajakontrollit, josta opinnäytetyön aihe lopulta muodostui. Projektissa työkaluina käytimme FingerGesture ja Unity 3D:tä.

Pelaajakontrollit ovat yksi pelin tärkein osa kokonaisuuden kannalta. Ilman hyvää pelaajan ohjausta hyväkin peli saattaa tuntua huonolta. (Llopis 2011, 220) Itseäni pelaajan ohjaus ja siihen liittyvät komponentit kiinnostavat, koska pelaajaa luodessa pääsee luomaan yhtä pelin ytimistä, joka määrittää pelin kokonaisuuden kannalta suurta osaa.

Aiheen motivaationa toimi mahdollisesti julkaistavan pelin toteuttaminen. Hyvä työtiimi auttoi motivaation suhteen hyvinkin paljon, koska hyvässä tiimissä on mukava työskennellä ja töitä tulee tehtyä yhteisen hyvän eteen paremmin. Kiinnostus itse pelaajakontrolleihin syntyi projektin myötä. Peliprojektiin alussa määrittelimme aihealueita koodareiden kesken ja minulle päätyi aikaisempien projektien pohjalta pelaajan toteuttaminen.

Projektin tavoitteena on tutkia muiden toteuttamia pelaajakontrolliratkaisuja ja vertailla niitä. Työn tavoitteena oli toteuttaa pelaajakontrollit omaan peliprojektiin ja tutustua samalla top-down-scroller-pelien ominaisuuksiin.

2 Yleistä

2.1 Genre

Opinnäytetyössä tehtävä peli sijoittuu shoot ’em up- ja vertical scroller –genreihin. Vertical scroller on shoot ’em up -alagenre ja shoot ’em up on ammuntapelien -alagenre, molemmat genret ovat kuitenkin useasti käytössä top-down-scroller peleissä. Esimerkiksi peleissä Half-Minute Hero (2009, Marvelous Entertainment), Twin Hawk (1989, Taito), Bomber Raid (1988, Sega), Dragon Saber (1990, Namco).

2.1.1 Shoot 'em up

Shoot ’em up, joka tunnetaan myös nimellä ”shmup” tai ”STG”, on peli, jossa päähenkilö taistelee useita vihollisia vastaan ampumalla heitä ja väistellen heidän ammuksiaan. Ohjaavan pelaajan on turvauduttava ensisijaisesti reaktioaikaansa menestyäkseen pelissä.

Kuva 1. Shogun (Kuvankaappaus).

On eri näkemyksiä siitä, millaisista suunnitteluelementeistä shoot ’em up muodostuu. Jotkut rajoittavat tämän genren peleihin, joissa on mukana

jonkinlainen alus ja jotka käyttävät kiinteää tai vierivää liikkumista. Toiset laajentavat käsitystään genrestä koskemaan myös pelejä, joissa ovat päähenkilöt robotteina tai ihmisinä kulkemassa jalan, sekä myös pelejä jotka sisältävät ”on-rails” tai ”run and gun” -liikkumista. (Wikipedia 2015b).

2.1.2 Vertical scroller

Vertical scroller tai vertically scrolling on videopeli, jossa pelaaja näkee pelikentän pääasiallisesti ylhäältä alaspäin kuvattuna, kun taas tausta rullaa näytön yläreunasta näytön alareunaan tai harvoin alhaalta ylös. Näin luodaan illuusiota, että pelaaja liikkuu pelimaailmassa. Kuten side-scroller-videopelit, vertical scroller-videopelit käyttävät usein hyödykseen parallax scrolling-tekniikkaa, luoden realistisemman liikkeen simuloinnin.

Taustan jatkuva vertikaalinen vieritys on suunniteltu simuloimaan jatkuvaa eteenpäin suuntautuvaa liikettä. Tällaisissa peleissä, peli asettaa tahdin pelata ja pelaajan täytyy reagoida nopeasti muuttuvaan ympäristöön. Vertical scroller on yleinen shoot ’em up -alalaji toimintapeleissä. Sitä joskus myös käytetään 2D-rallipeleissä.

Kuva 2. Starward.

Vertical scroller -genre konkretisoitui 1970-luvun puolivälissä. Kenties ensimmäinen vertical scroller -peli oli Atarin kehittämä arcade-rallipeli Hi-way(1975). Hi-way oli myös ensimmäinen Atarin arcadepeli, jossa oli ”ohjaamo-tyylinen” istuinkaukalo. Activision River Raid (1982) Atari 2600 -pelikonsoleille auttoi luomaan vertical shooter -alagenren. (Wikipedia 2013).

2.2 2D-kuvakulma

Työni toiminnallisen osuuden toteutuksessa käytettiin kahta 2D-kuvakulmaa: top-down- ja 2.5D-kuvakulmaa. Ne ovat laajalti käytössä top-down-scrollereissa kuten Raiden IV (2007, Taito), DoDonPachi Resurection (2008, AMI) ja Protöthea (2005, Ubisoft).

2.2.1 Top-down-kuvakulma

Top-down-kuvakulmasta käytetään myös nimitystä näköala lintuperspektiivistä, helikopterinäkymä ja pään yli -näkymä. Kuvakulmaa käytettäessä videopeleissä viitataan kamerakulmaan, joka näyttää pelaajan ja ympäristön ylhäältäpäin kuvattuna. Kuvakulmaa ei käytetä pelkästään videopeleissä, jotka käyttävät parallax scrolling -tekniikkaa. Kuvakulma oli aikoinaan yhteistä myös 2D-roolipeleissä ja sotapeleissä, kuten SimCity (1989, Electronic Arts), Pokémon (1996, Nintendo) ja Railroad Tycoon (1990, MicroProse), sekä toiminta- ja seikkailupeleissä kuten varhaisissa The Legend of Zelda- (1986, Nintendo EAD) ja Grand Theft Auto-peleissä (1997, Rockstar Games). (Wikipedia 2014).

Kuva 3. Starward.

2.2.2 2.5D

2.5D (”kaksi ja puoli -ulotteinen”) on termi jota lähinnä käytetään videopeliteollisuudessa kuvaamaan joko graafista 2D-projektiota ja vastaavia tekniikoita. 2.5D-tekniikkaa käytetään aiheuttamaan kuvasarjan simulointia niin, että se vaikuttaa kolmiulotteiselta (3D), vaikka todellisuudessa se ei ole.

Kuva 4. Starward.

Termiä 2.5D sovelletaan myös 3D-peleihin, jotka käyttävät monimuotoista grafiikkaa renderöidäkseen maailmaa ja/tai hahmoja, mutta jonka pelattavuus on rajoitettu 2D tasossa. (Wikipedia 2015f).

2.3 3D-kuvakulma

Esittelen seuraavaksi joitakin 3D-kuvakumia, fixed 3D- ja first-person-kuvakulmat.

2.3.1 Fixed 3D

Fixed 3D viittaa kolmiulotteiseen pelimaailmaan, jossa etualalla olevat objektit (kuten pelihahmot) on tyypillisesti sulatettu reaaliaikaisesti vasten staattista taustaa esimerkiksi Syder Arcade (2013, Meridian4) (Kuva 5). Pääasiallinen etu tässä tekniikassa on sen kyky näyttää korkeatasoisia yksityiskohtia vähäisillä tehoilla.

Kuva 5. Syder Arcade (Kuvankaappaus).

Suurin haitta kuvakulmassa on, että pelaajan kokonaiskuva pysyy staattisena koko ajan, mikä estää pelaajaa tutkimasta ympäristöä ja liikkumasta ympäristössä monissa näkökulmissa. Taustakuvat Fixed 3D-peleissä ovat usein valmiiksi renderoituja 2D-kuvia, mutta joskus niitä generoidaan myös reaaliajassa, esimerkiksi Blade Runner (1997, Virgin Interactive) -pelissä (Kuva 6). Pelissä pelaaja hahmo on kolmiulotteinen jonka liikkumatila on rajoitettu vain kiinteään kamerassa näkyvään alueeseen. Peliä pelatessa valmiiksi renderoituja taustoja on vaikea huomata, mutta kuvan pystyy tunnistamaan siitä, että kohtauksessa jossa kamera ei liiku kuva on staattinen ja valokuvamainen.

Kuva 6. Blade Runner (Kuvankaappaus).

SimCity 4:n (2003, Electronic Arts) (Kuva 7) kehittäjät käyttivät hyväkseen Fixed 3D-kuvakulmaa ja jättivät teksturoimatta objektien taustapuolet, joita pelaaja ei voisi kuitenkaan nähdä ja näin ollen nopeuttivat objektien renderöintiä. (Wikipedia 2014). Näin ollen pelissä talon neljästä seinästä on kaksi taaimmaista jätetty teksturoimatta. Koska pelissä pelaaja ei voi vaikuttaa kameran kulmaan, ei pelaaja pysty näkemään teksturoimatta olevia talon seiniä.

Kuva 7. SimCity 4 (Kuvankaappaus).

2.3.2 First-person-kuvakulma

First-person viittaa graafiseen näkökulmaan, joka on renderöity pelaajahahmon näkökulmasta. Monissa tapauksissa tämä saattaa olla näkökulma ajoneuvon ohjaamosta esimerkiksi Wing Commander (1994, Origin System) (Kuva 8). Monet eri genret seikkailupeleistä lentosimulaattoreihin ovat käyttäneet hyväkseen first-person-kuvakulmaa. Ehkä kaikkein merkittävin genre, joka on käyttänyt hyväkseen first-person-kuvakulmaa, on räiskintäpelin genre, jossa graafisella näkökulmalla on erittäin voimakas vaikutus pelin pelaamiseen, koska hahmon on pystyttävä liikkumaan ahtaissa paikoissa ja tutkimaan asioita kulmien ja nurkkien takaa. First-person-kuvakulma tuo omanlaisensa pelikokemuksen, kuin pelaaja olisi itse juoksemassa pelikentällä ase kädessään.

Kuva 8. Wing Commander (Kuvankaappaus).

Pelit, joissa on first-person-kuvakulma, ovat useasti hahmopohjaisia, jolloin peli näyttää mitä pelaajan hahmo näkisi hahmon silmin. Pelaaja tyypillisesti ei näe hahmonsa ruumista, vaikka hän ehkä pystyy näkemään hahmonsa aseen tai kädet. (Wikipedia 2014).

2.4 Kosketuskontrollit

“Buttons are doomed; touchscreens are the new game controllers.” (Tim Rogers 2013).

Mitä pidemmälle menemme maailmaan jossa mobiililaitteet yleistyvät kosketusnäyttöineen, sitä enemmän pelinkehittäjien tulee miettiä ratkaisuja kosketuskontrollien toteuttamiseen. Kosketuskontrolleja miettiessä on hyvä ottaa huomioon pelin tarpeet, kuten mahdolliset aseiden vaihdot, pelin pysäytys, pelin hiljennys, inventaariot jne. Näytölle painikkeita ja kontrolleja muodostaessa on hyvä ottaa huomioon näytöllä oleva tila, joka on suhteellisen pieni ja rajallinen.

Kosketuskontrolleja tehdessä on hyvä ottaa huomioon pelaajan kädet, koska kosketuskontrollisia pelejä useasti pelataan tabletti- ja mobiililaitteilla, jolloin saattavat pelaajan kädet tulla pelin tielle. Tällöin on hyvä ottaa huomioon tarvittavien kontrollien sijainti. Kosketuskontrolleja tehdessä kannattaa ottaa huomioon muiden henkilöiden toteuttamia ratkaisuja. Näin voi löytää ratkaisuja, jotka voivat olla hyödyksi tehdessä omia kontrolleja. Voi esimerkiksi huomioida, miten toiset ovat ratkaisseet ongelman jossa käyttäjän kädet peittävät suurimman osan ruudusta tai kuinka monta painiketta on ruudulla liikaa tai liian vähän.

Seuraavaksi esittelen esimerkkejä kosketuskontrollien toteutuksista top-down-scroller peleissä. Esimerkiksi Sky Force-pelissä (2015, Infinite Dreams), pelaaja seuraa käyttäjän sormea liikuttaen hahmoa sormen mukaisesti. Pelissä pelaaja-alus ampuu automaattisesti ja pelin pysäytys menee päälle, kun sormen nostaa näytöltä.

Shogun-pelissä (2012, int13.net) pelaaja-alus seuraa sormea liikkuakseen. Pelaaja-alus ampuu automaattisesti aseillaan. Sormen nosto aktivoi valikon,

jossa pelaaja voi vaihtaa asetta ja peli hidastuu samalla. Pelin pysäytys aktivoidaan painamalla pysäytyspainiketta ruudulla.

Starward-pelissä pelaaja-alus seuraa käyttäjän sormea ruudulla. Pelaaja-alus ampuu automaattisesti eteenpäin. Pelin pysäytystila aktivoidaan nostamalla sormi ruudulta ylös.

Raiden Fighters 2-pelissä (1997, Fabtek) pelaaja-alus seuraa sormea ruudulla. Pelaaja-alus ampuu automaattisesti eteenpäin. Pelin pysäytystila aktivoidaan painamalla ruudulla olevaa pysäytyspainiketta. Pelissä olevat erikoiset aseet aktivoidaan painamalla asepainiketta joko sormella joka liikuttaa pelaaja tai toisella sormella.

Mielestäni pelaaja-aluksen liike ja automaattinen ampuminen on hyvä idea, koska tällöin pelaajalle jää käytettäväksi toinen käsi mahdollisille aktivoitaville erikoishyökkäyksille. Pelissä olevan pausen toteuttaminen pysäytyspainikkeella on hyvä ja toimiva idea, mutta siinä on omat heikkoutensa. Kuten se vie tilaa näytöltä ja sen aktivointi tiukassa tilanteessa voi olla hankalaa.

Käytännön projektissa toteutetut pelaaja kontrollit ovat toteutettu samalla pohjalla, kuin edellä mainituissa pelaajakontrolleissa koska, kun peli testasimme pelaajakontrolleita, huomasimme miten yksinkertaiset ja helppokäyttöiset kontrollit ovat. Joten päätimme toteuttaa mahdollisimman samankaltaiset pelaajakontrollit omaan projektiimme.

2.5 Käytettyjä kontrolliratkaisuja

Peli ilman pelaajan vuorovaikutusta olisi kuin elokuva. Siksi on hyvin tärkeää, että pelin kehittäjä osaa käsitellä tehokkaasti pelaajan syötteitä. Unity tarjoaa useita tapoja lukea pelaajan syötteitä ja on hyvin tärkeää ymmärtää niitä. (Managing Player Input in Unity. 2013.)

Top-down-scroller-peleissä pelaajan kontrollit toteutetaan useasti hiirellä tai sormella toimiviksi kokonaisuuksiksi. Esimerkiksi liikkuminen usein toteutetaan

niin, että kun pelaaja painaa hiiren painikkeen pohjaan tai sormen näytölle, seuraa hahmo liikettä. Ampuminen toteutetaan useasti automaattisena elementtinä hahmon liikkumisen kanssa. Kun pelaaja painaa hiiren pohjaan tai sormen näytölle, alkaa samalla hahmo ampua. Joissakin peleissä kuten Rage HD, Archetype ja N.O.V.A , ampuminen on toteutettu käyttäen toista näppäintä ampumisen suorittamiseksi.

Pelin aikana käytettävällä pysäytyksellä on useita erilaisia toteutustapoja, joista kaksi yleistä toteutustapaa esittelen seuraavaksi. Ensimmäinen näistä kahdesta on pelin yläreunaan laitettava pysäytyspainike, jota painamalla peli menee pysähdykselle, jonka lisäksi yleensä pelin menu aukeaa. Toinen toteutustapa on hiiren painikkeeseen tai sormeen sidottava pysäytys. Kun pelaaja nostaa hiiren painikkeen ylös tai sormen näytöltä, peli pysähtyy.

3 Kontrollien toteutus Unity 3D:ssä

3.1 Unity-kontrollit

Unity 3D:ssä on kaksi useasti käytettyä tapaa toteuttaa pelaajakontrolleja. Ensimmäinen tapa on käyttää Unityn sisään rakennettua Input Manageria, johon myös osa Unityyn lisättävistä työkaluista perustuu. Unity 3D:ssä sisäisen Input Managerin käyttäminen pelaajan ohjauksen toteuttamiseen on suhteellisen yksinkertaista. Menemällä Input Manageriin henkilö pystyy määrittämään pelissä käytettävät kontrollit suoraan Unity 3D:n inspektorin kautta. Kokemattomalle käyttäjälle helpoin tapa toteuttaa pelaajakontrollit on käyttää Input Manageria.

Toinen tapa toteuttaa pelaajakontrollit Unity3D:ssä on kirjoittaa kontrollit alusta loppuun UnityScriptillä. Tämä tapa vaatii vähän kokeneemman käyttäjän, koska käyttäjä itse joutuu määrittelemään kaiken ohjaukseen tapahtuvan toiminnan. Uusin Unity 3D -versio (4.5.0) tarjoaa käyttäjälleen tuen käyttää Unityn hiiren

tunnistusta sormen kontrolleihin, puhelin- tai tablettilaitteilla. Vanhemmissa versioissa käyttäjä joutuu itse määrittämään tuen hiiren ja sormen välillä.

3.2 Trivasion

Trivasion on peli jonka tuottajana ja julkaisijana toimii PolarBunny. Luku koostuu pelaajakontrollien toteuttajan haastattelusta ja hänen tekemästään koodista.

Trivasion-pelissä pelaajankontrollit on toteutettu tietokoneelle käyttäen joystick-ohjainta ja PS Vitalle käyttäen tämän omia ohjaimia. Pelissä pelaajan ylös-alas-liike on automaattinen ruudun ylhäältä alas ja päinvastoin. Pelissä on kaksi erilaista pelimuotoa joiden välillä pelaajan ohjaus vaihtuu. Molemmissa pelimuodoissa pelaajan sivuttainen liike (Kuva 9) ohjautuu joystick-ohjaussauvalla tai käyttäen ohjaimissa olevia nuolinäppäimiä.

```
    if (Input.GetAxis ("Horizontal") != 0) {
            if (Input.GetAxis ("Horizontal") < 0) {
                    //      mSpeed = 10;
                    xInput = Mathf.MoveTowards (xInput, -maxX * Mathf.Abs (Input.GetAxis ("Horizontal")), Time.deltaTime * 12);
            }

            if (Input.GetAxis ("Horizontal") > 0) {
                    //      mSpeed = 10;
                    xInput = Mathf.MoveTowards (xInput, maxX * Mathf.Abs (Input.GetAxis ("Horizontal")), Time.deltaTime * 12);
            }
    } else if (xInput != 0) {
            //mSpeed = 10;
            xInput = Mathf.MoveTowards (xInput, 0, Time.deltaTime * 6);
    }

    moveDir.x = xInput;

    if (!attackS.boosting)
            transform.Translate (moveDir * mSpeed * Time.deltaTime);
    else
            transform.Translate (moveDir * mSpeed * boostingSpeed * Time.deltaTime);


    if (transform.position.y >= bounds.y && moveDir.y == 1) {
            yMovement ();
    }

    if (transform.position.y <= -bounds.y && moveDir.y == -1) {
            yMovement ();
    }


    transform.position = new Vector3 (
Mathf.Clamp (transform.position.x, -bounds.x, bounds.x),
Mathf.Clamp (transform.position.y, -bounds.y, bounds.y), 0);
```

Kuva 9. Sivuttainen liike.

Sphere-pelimuodossa pelaaja hyökkää käyttämällä R1-painiketta. Pelissä oleva hyökkäys (Kuva 10) aloittaa lataamaan hyökkäystä, kun R1-painike painetaan pohjaan. Pelaaja lataa hyökkäysvoimaa niin pitkään, kun R1-painike on pohjassa tai pelimuodossa oleva hyökkäysmittari tyhjenee. Hyökkäyksen voi aktivoida vapauttamalla R1-painikkeen tai käyttäen hyökkäysmittarin tyhjäksi. Ladatessa hyökkäystä pelaaja pystyy syöksymään vihollisia kohti tuhoten ne matkallaan ja näin ollen lisäämään aikaa mittariin.

```
        if (boosting) {
                //when boosting, timer is consumed faster
                boostingTimer = boostingTimer - Time.deltaTime * 2.65f;

                boostDuration = boostDuration + Time.deltaTime;

                if (ps.cooldownTimer > 0)
                        ps.cooldownTimer -= Time.deltaTime;

                if (ps.boostingSpeed > 1.5f)
                        ps.boostingSpeed = ps.boostingSpeed - Time.deltaTime;

            mms.boostSeconds += Time.deltaTime;

        }

        //boosting timer recharges
        if (!boosting && !ps.dead && boostingTimer < 8) {
                boostingTimer = boostingTimer + Time.deltaTime;
        }
        if (ps.invulnerable) {
                invulnerableTime -= Time.deltaTime;
        }

        if (ps.dead && boostingTimer < 8)
                boostingTimer = boostingTimer + Time.deltaTime * 8;


if(ps.attackMode)
        AttackSphere ();

    clock ();
}
```

Kuva 10. Hyökkäys.

Slowmo-pelimuodossa pelaaja aktivoi hidastuksen (Kuva 11) painamalla R1-painikkeen pohjaan. Hidastus toimii niin pitkään, kun R1-painike on pohjassa tai pelimuodossa oleva hidastusmittari tyhjenee. Hidastus poistuu käytöstä vapauttamalla R1-painikkeen tai käyttämällä hidastusta kunnes mittari tyhjenee. Käyttäen hidastusta pelaaja voi vaihtaa sivuttaisliikkeen suuntaa äkillisesti toiseen suuntaan.

```
if (!ps.attackMode && !ps.classicMode)
{
    if (Input.GetKeyDown ("joystick button 5") || Input.GetKeyDown ("space"))
    {
        slowMo = true;
        SlowMotionMode ();
    }
    else if (Input.GetKeyUp ("joystick button 5") || Input.GetKeyUp ("space" ) || slowTimer <= 0)
    {
        slowMo = false;
        SlowMotionMode ();
    }


    if (slowMo)
    {
        slowTimer -= Time.deltaTime * 4f;
    }

    if (!slowMo && !ps.dead && slowTimer <= 8)
    {
        slowTimer += Time.deltaTime;
    }

    if (ps.dead && slowTimer <= 8){
    slowTimer += Time.deltaTime * 8;
    }
}
 void SlowMotionMode ()
 {
     if(slowMo == true)
     {
         Time.timeScale = 0.5f;
         Time.fixedDeltaTime = 0.02F * Time.timeScale;
         mms.slowmoSeconds += Time.deltaTime * 2;
         float time = Time.timeScale;
     }
     else if (slowMo == false)
     {
         Time.timeScale = 1.0f;
         Time.fixedDeltaTime = 0.02F * Time.timeScale;

         float time = Time.timeScale;
     }
 }
```

Kuva 11. Hidastus.

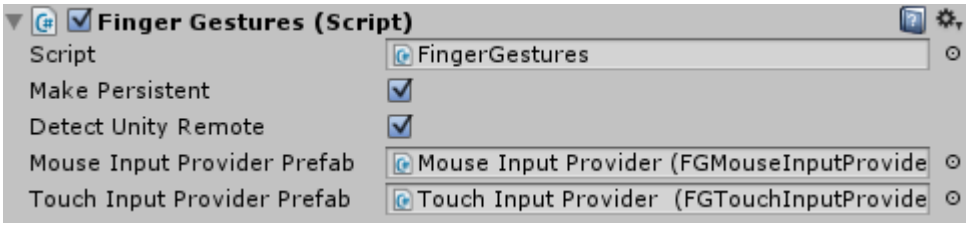
Pelissä olevat kontrollit on toteutettu vektoreilla, joilla pystytään luomaan sujuvaa ja tarkkaa liikettä. Fysiikkapohjaista liikettä ei ole pelissä käytetty, koska fysiikalla pelaajalle ei pystytä luomaan riittävän tarkkaa liikettä. Tämä johtuu siitä, että pelimoottori laskee fyysiset voimat, ja ohjelmoija pystyy vaikuttamaan niihin ainoastaan välillisesti.

Fyysisen voiman lisäys suuntaan antaa hahmolle liikkeen. Kun liike aloitetaan toiseen suuntaan, on alkuperäinen voima kumottava. Näin ollen hahmo ensin hidastuu, pysähtyy ja lähtee liikkumaan toiseen suuntaan. Kun kontrollit toteutetaan vektoreilla, pystytään liike vastakkaiseen suuntaan aloittamaan heti ilman hidastusta. (Timonen 2015).

3.3 FingerGesture

FingerGesture on Unity 3D:hen lisättävä (Unity Asset) työkalu, joka tarjoaa helpot työkalut tehdä kosketusnäytölle ja hiirelle sopivia kontrolleja. Paketti sisältää tarkat kontrollipohjat, niin perustason, kuin kehittyneemmän tason kontrolleille. FingerGesture vaatii kaksi peruskomponenttia Unity-kohtaukseen (Unity Scene) toimiakseen oikein.

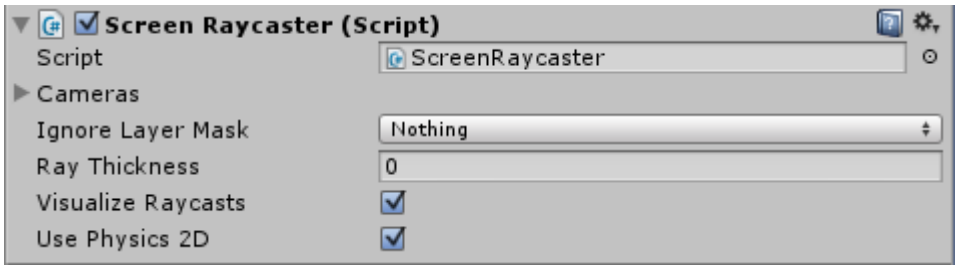
Ensimmäinen peruskomponentti on FingerGestures (Kuva 12), joka hoitaa kommunikoinnin laitteen ja kontrollien välillä. Komponentti tarkistaa automaattisesti käytössä olevan laitteen ja valmistelee yhteyden sekä varmistaa olevansa olemassa. Komponentti tulee laittaa jokaiseen Unity-kohtaukseen (Unity Scene), jotta muut komponentit toimisivat oikein.



Kuva 12. FingerGestures-komponentti (Kuvankaappaus).

Komponentista voi tehdä tuhoutumattoman aktivoimalla komponentin osan (”Make Persistent”), minkä jälkeen komponentti tulee automaattisesti jokaiseen Unity-kohtaukseen (Unity Scene). Komponentti tuhoaa kaikki ylimääräiset kopiot itsestään automaattisesti. Komponenttiin saa tuen Unity Remotelle, jos haluaa testata sormella toimivia näppäimiä puhelimella tai tabletilla. (FingerGestures Documentation 2013a.)

Toinen komponentti on ScreenRaycaster (Kuva 13), joka hoitaa kontrollien kohdistuksen hiiren ja sormen osalta käyttäen Unity Raycastiä. Komponentti luo Unity Raycastin kamerasta sormen tai hiiren osoittamaan kohtaan ja tarkastaa osuuko Raycast mahdollisiin objekteihin. Komponenttia voi muokata omien tarpeidensa mukaisesti ja säätää, että osuuko Unity Raycast tietyllä tasolla oleviin objekteihin vai ei, ja tarvittaessa Unity Raycastin voi myös säätää 2D:lle sopivaksi. (FingerGestures Documentation 2013b.)



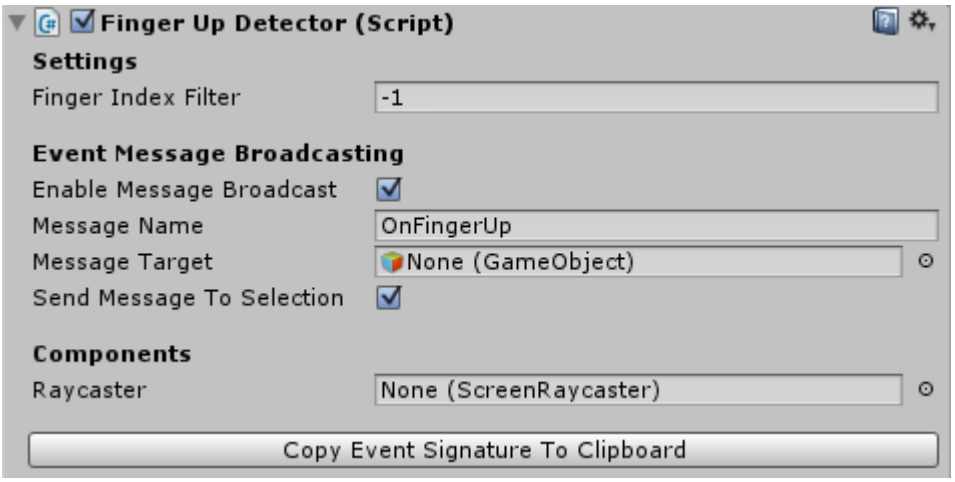
Kuva 13. ScreenRaycaster (Kuvankaappaus).

Komponentit hakevat ScreenRaycasterin itse ohjelman käynnistettyä, mutta käytettäessä kahta tai useampaa ScreenRaycasteriä on käyttäjän itse määriteltävä, mitä ScreenRaycasteriä komponentti käyttää. Kaikki FingerGesture-komponentit tarjoavat suoraan metodipohjan leikepöydälle napin painalluksella, mikä nopeuttaa koodin aloittamisessa tai uudelleen kirjoittamisessa. Esimerkiksi FingerUpDetectorin metodipohja saadaan void OnFingerUp(FingerUpEvent e) { /* Oma koodi tähän */ }.

3.3.1 FingerUpDetector

FingerUpDetector-komponentti (Kuva 14) hoitaa sormen tai hiiren painikkeen ”nosto”-toimenpiteen. Komponentti suorittaa koodin oman metodinsa sisältä

joka kerralla, kun käyttäjä nostaa sormen näytöltä tai nostaa hiiren painikkeen ylös.



Kuva 14. FingerUpDetector (Kuvankaappaus).

Käytettäessä hiirtä komponentin voi laittaa sivuuttamaan osa hiiren painikkeista vaihtamalla komponentin osan (”Finger Index Filter”) arvoa. Arvo -1 kertoo, että kaikki hiiren painikkeet ovat käytössä. Arvo 0 kertoo, että vain hiiren vasen painike toimii. Arvo 1 kertoo, että hiiren oikea painike toimii. Arvo 2 kertoo, että vain hiiren rullan painike toimii.

Käyttäessä sormea voidaan komponentti laittaa sivuuttamaan osan sormista. Arvo -1 kertoo, että kaikki sormet luetaan. Arvo 0 kertoo, että vain ensimmäinen näytölle laitettu sormi toimii. Arvo 1 kertoo, että näytöllä olevista sormista toisena näytölle laitettu sormi toimii. Arvo 2 kertoo, että kolmas näytölle laitetuista sormista toimii. (FingerGestures Documentation 2013c.)

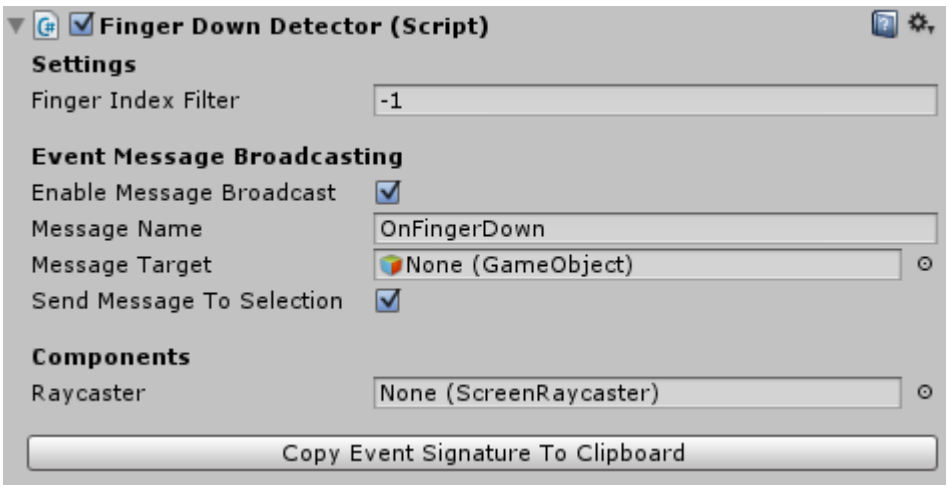
FingerUpDetectoria voidaan käyttää esimerkkinä pelin pysähdyksen toteuttamiseen, jolloin pelin pysäytys suoritetaan aina, kun pelaaja nostaa sormen ruudulta tai jopa, kun pelaaja erehdyksessään vie sormen ruudun reunalle ja ulos ruudulta.

FingerUpDetectoria voidaan käyttää monen muun FingerGesture-komponentin kanssa tunnistuksen lopetukseen. Esimerkiksi, pelaaja laittaa sormensa näytölle ja poimii sormellansa esineen. Kun sormi nostetaan näytöltä, voidaan

FingerUpDetector laittaa tunnistamaan sormen nosto ja pelaaja pudottamaan nostamansa esine.

3.3.2 FingerDownDetector

FingerDownDetector-komponentti (Kuva 15) hoitaa sormen tai hiiren painikkeen ”painallus”-toimenpiteen. Komponentti suorittaa koodin oman metodinsa sisältä joka kerralla, kun käyttäjä laittaa sormen näytölle tai painaa hiiren painikkeen pohjaan.



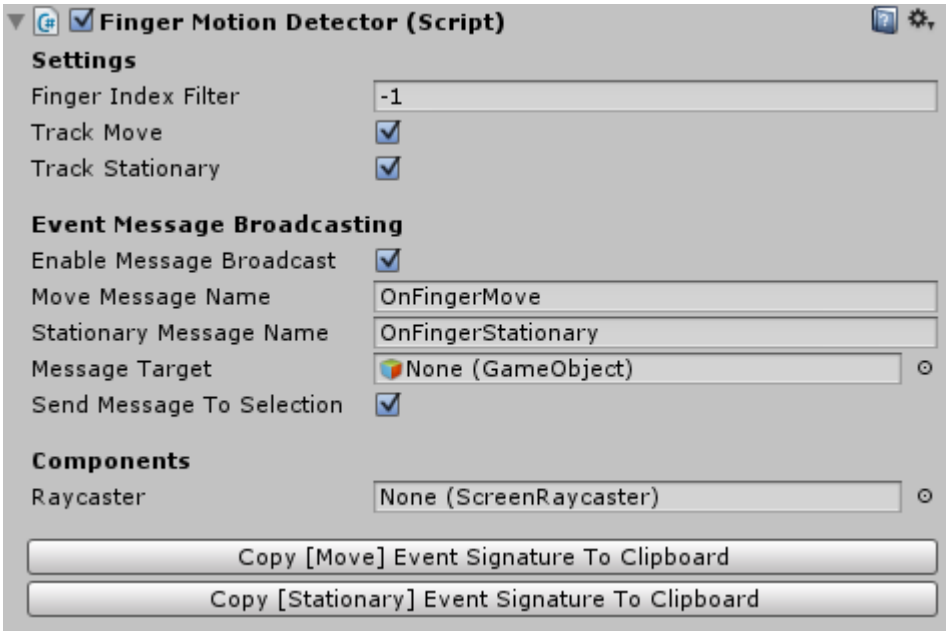
Kuva 15. FingerDownDetector (Kuvankaappaus).

FingerDownDetectoria voidaan käyttää esimerkiksi pelissä hahmon kykyjen käyttämiseen tai kaupassa ostettavien hahmon päivityksien tai tavaroiden ostamiseen. Esimerkiksi kun käyttäjä painaa sormen jonkin painikkeen päälle, komponentti ilmoittaa alla olevalle painikkeelle, että painiketta on painettu. (FingerGestures Documentation 2013c.)

Komponenttia voidaan käyttää myös hahmon hyökkäyksen toteuttamiseen, kun käyttäjä laittaa sormen näytölle, hahmo alkaa ampumaan tai huitomaan miekallaan. Hyökkäämisen tunnistaminen voidaan toteuttaa koko näytöllä toimivaksi tai näytöstä voidaan rajata tietty alue, jota painamalla pelaaja ampuu tai lyö.

3.3.3 FingerMotionDetector

FingerMotionDetector-komponentti (Kuva 16) hoitaa sormen tai hiiren liikkeen ja liikkumattomuuden tunnistamisen vain, kun sormi on näytöllä tai kun hiiren painike on pohjassa. Liikkeessä on kolme tunnistettavaa vaihetta: liikkeen aloitus (Started), liikkeen päivittyminen (Updated) ja liikkeen lopetus (Ended). Liikkumattomuudessa on myös kolme tunnistettavaa vaihetta: pysähdys (Started), liike paikallaan (Updated) ja liike jatkuu (Ended). (FingerGestures Documentation 2013c.)



Kuva 16. FingerMotionDetector (Kuvankaappaus).

Liikkeen kolme vaihetta alkavat siitä, kun käyttäjä laittaa sormen näytölle tai painaa hiiren painikkeen pohjaan. Jos liike jossain vaiheessa pysähtyy, alkaa liikkumattomuuden tunnistamisen kolme vaihetta, ja kun liike jatkuu, alkaa taas liikkeen tunnistaminen.

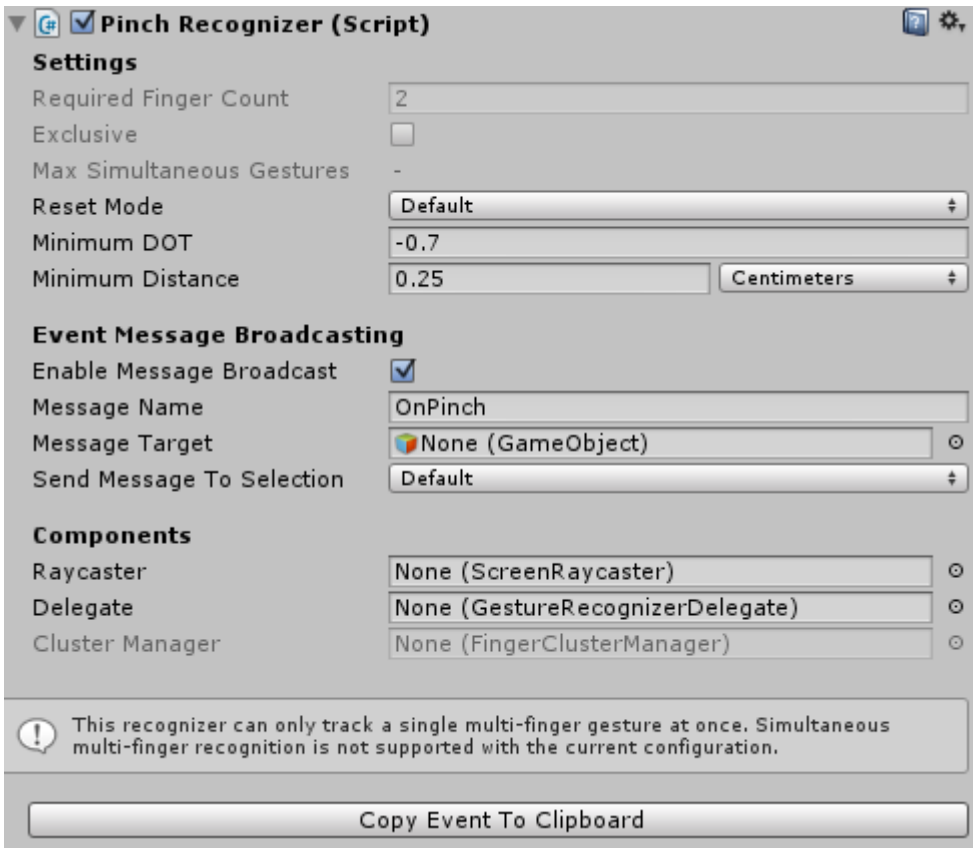
FingerMotionDetector on yksi komponenteista, jolla voi helposti toteuttaa pelaajan liikkumisen, koska komponentti antaa tarkan tunnistuksen sormen tai hiiren liikkeelle pelissä, josta käyttäjän on yksinkertainen toteuttaa hahmon liikkuminen.

Esimerkiksi, kun käyttäjä painaa sormen ruudulle, pelaajahahmo huomioi metodin (Started), jossa pelaaja voidaan laittaa huomioimaan sormen aktivointi. Kun käyttäjä siirtää sormea ruudulla, tulee tunnistuksen toinen vaihe (Updated) jolloin lähtee pelaajahahmo seuraamaan sormea.

Tällöin pelaajahahmo huomioi esimerkiksi sormen kohdan ruudulla ja lähtee seuraamaan sormea komponentin tunnistuksen myötä. Kun käyttäjä lopettaa liikkeen tai nostaa sormen ruudulta, tulee tunnistuksen kolmas vaihe (Ended) ja pelaajahahmo lopettaa liikkeen. Esimerkiksi kun pelaaja nostaa sormen näytöltä voidaan komponentin tunnistusta käyttää hyödyksi ja siirtää pelaajahahmo sormen viimeiseen kohtaan ruudulla.

3.3.4 PinchRecognizer

PinchRecognizer-komponentti (Kuva 17) hoitaa sormien nipistyksen ja loitonnuksen tunnistamisen. Komponentti suorittaa koodin oman metodinsa sisältä joka kerralla, kun käyttäjä suorittaa nipistys- tai loitonnusliikkeen ruudulla.



Kuva 17. PinchRecognizer (Kuvankaappaus).

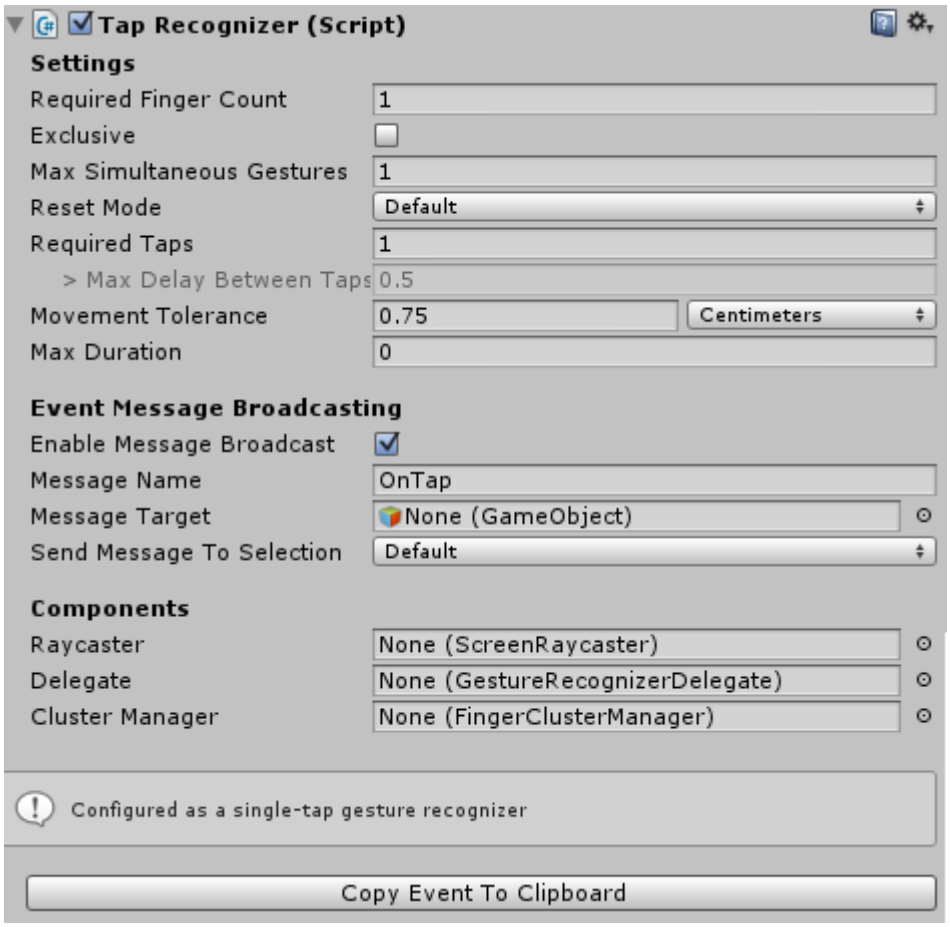
Komponentin osa (”Minimum Distance”) huomioi nipistysliikkeessä tarvittavan etäisyyden käyttäjän sormista ja osa myös huomioi tarvittavan sormien etäisyyden toisistaan käytettäessä loitontamisen tunnistusta. (FingerGestures Documentation 2013d.)

Komponenttia voidaan käyttää esimerkiksi pelin kartan tarkentamiseen nipistysliikkeellä. Nipistysliike on helppo ratkaisu kartan tarkennuksen toteutuksessa, kun komponentti tarjoaa tarvittavan muuttujan, josta kartan tarkennuksen suuruus on helppo laskea.

Komponenttia voi myös käyttää hyväksi kartan pienentämiseen samoilla menetelmillä, mutta vähentäen kartan suuruutta. Kun pelaaja vie sormia yhteen nipistysmenetelmällä, komponentti laskee, kuinka iso liike tapahtuu sormien välissä ja tästä liikkeestä komponentti tarjoaa tarvittavan muuttujan, jota voidaan käyttää kartan pienentämiseen.

3.3.5 TapRecognizer

TapRecognizer-komponentti (Kuva 18) hoitaa napautuksen tunnistamisen. Komponentti suorittaa koodin oman metodinsa sisältä joka kerralla, kun käyttäjä suorittaa sormen napautuksen tai hiiren napautuksen sovelluksessa. Napautus luokitellaan yhdeksi lyhyeksi kosketukseksi tai lyhyeksi painallukseksi. Sormen ruudulle tai hiiren pohjaan painaminen luetaan komponentilla FingerDownDetector. Molempia komponentteja käyttäessä käyttäjän tulee olla erityisen tarkka, etteivät komponentit mene sekaisin.



Kuva 18. TapRecognizer (Kuvankaappaus).

Komponentin osa (”Required Finger Count”) säätää komponentissa tarvittavien sormien määrää ja milloin komponentti aloittaa napautuksen tunnistamisen, esimerkiksi yhden tai kahden sormen napautus. Komponentin osa (” Required Taps”) säätää komponentissa tarvittavien napautuksien määrä ja milloin komponentti aloittaa napautuksen tunnistamisen, esimerkiksi kahdella tai kolmella sormella suoritettava napautus.

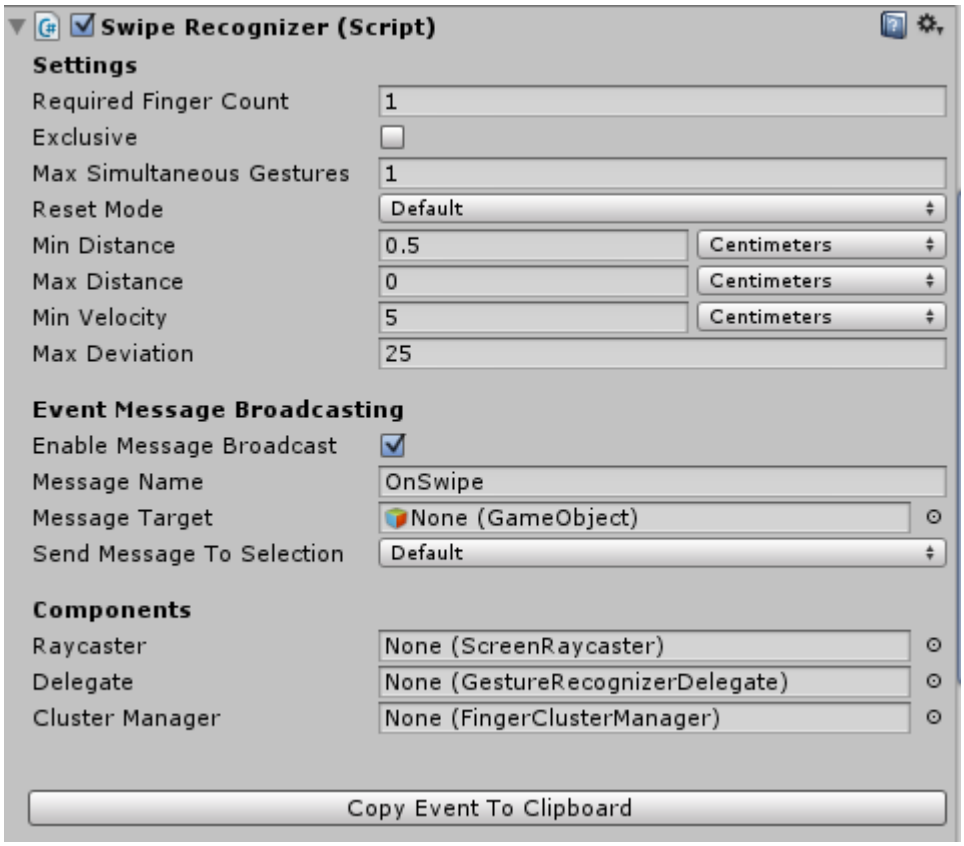
Komponentin osa (”Move Tolerance”) huomioi, miten paljon napautuksien välillä sormi saa liikkua, jotta komponentti tunnistaa kaksinkertaisen tai useamman kertaisen napautuksen. Komponentin osa (”Max Duration”) huomioi, kuinka pitkään sormi saa olla näytöllä tai hiiren painike pohjassa, jotta toiminto huomioidaan näpäytykseksi. (FingerGestures Documentation 2013e.)

Komponenttia voidaan käyttää esimerkiksi pelaajan käyttäessä inventaariota varusteidensa vaihtamiseksi. Kun pelaaja näpäyttää kerran varustetta

inventaariossa, näkee pelaaja varusteen tiedot, kun taas kaksinkertaisella näpäytyksellä pelaaja laittaa varusteen automaattisesti päälleen. Tällainen varusteiden vaihto-operaatio on yksinkertainen toteuttaa käyttäen hyväksi TapRecognizer-komponenttia.

3.3.6 SwipeRecognizer

SwipeRecognizer-komponentti (Kuva 19) hoitaa pyyhkäisyn tunnistamisen. Komponentti suorittaa koodin oman metodinsa sisältä joka kerralla, kun käyttäjä suorittaa sormen pyyhkäisyliikkeen tai hiiren pyyhkäisyliikkeen näytöllä.



Kuva 19. SwipeRecognizer (Kuvankaappaus).

Komponentin osa (”Required Finger Count”) määrittää, montako sormea tai hiiren painiketta täytyy olla käytössä, jotta komponentti huomioi pyyhkäisyn hyväksytysti. Komponentin osat (”Min Distance”) ja (”Max Distance”) määrittävät pyyhkäisyn rajat varmistamalla, että pyyhkäisy on riittävän pitkä ja ettei pyyhkäisy voi mennä tarkoitetun rajan yli.

Komponentin osa (”Min Velocity”) määrittää pyyhkäisyn vähimmäisnopeuden, jolla pyyhkäisy pitää suorittaa hyväksytysti. Komponentin osa (”Max Deviation”) määrittää maksimirajan pyyhkäisyn poikkeamiselle lähtöpisteestä. (FingerGestures Documentation 2013f.)

Komponenttia voidaan käyttää esimerkiksi jousipyssyn ampumiseen. Kun pelaaja vie sormen pelaajan päälle ja pyyhkäisee sormea tai hiirtä poispäin pelaajasta, ampuu pelaaja nuolen pyyhkäisyn voimakkuuden ja kulman mukaan. Komponenttia voidaan käyttää esimerkiksi hedelmien kuten vesimelonin tai banaanin puolittamiseen. Pelaaja yksinkertaisesti pyyhkäisee sormella tai hiirellä ruudulla näkyvän vesimelonin päältä, komponenttia voidaan käyttää tunnistamaan, osuiko pyyhkäisy meloniin vai ei.

3.3.7 Havaintoja

Käyttäessämme FingerGesture-komponenttia huomasimme kuinka Unityn playmode pysähtyi, kun muokkasimme komponenttien arvoa inspektorissa playmode päällä. Tätä ongelmaa yritimme selvittää itse etsimällä tietoa ja kyselemällä toisilta kehittäjiltä. Asiaa tutkittuamme huomasimme, että ongelma ilmestyi ainoastaan omassa projektissamme ja aikamme säädettyä emme saaneet ongelmaa korjatuksi.

Komponentin käytöstä löysimme myös tavan käyttää montaa FingerGesture-komponenttia yhtä aikaa. Kuten kahta FingerUpDetectoria tai FingerUpDetectoria ja TapRecognizer. Jälkikäteen tutkiessamme huomasimme, myös että monet muut kehittäjät olivat löytäneet tavan käyttää montaa komponenttia yhtä aikaa.

4 Pelaajakontrollien toteutus space shooter -peliin

4.1 Yleistä

Toiminnallisena osuutena opinnäytetyössä toimii Oasis-pelihautomossa toteutettu peliprojekti. Peli-idea syntyi tiimin sisäisten mietiskelyjen jälkeen, kun tavoitteena oli tehdä toteutuskelpoinen ja selkeäsisältöinen peli. Projektin alussa teimme muutamia prototyyppejä erilaisista peli-ideoista ja käytimme jokaisen proton tekemiseen kaksi viikkoa jonka jälkeen pidimme palaverin prototyypistä.

Peli-ideoiden ensimmäiset prototyypit toteutimme tietokoneelle ja samalla tutustuimme projektityöskentelyn kokonaisuuteen ja mahdollisiin käytettäviin työkaluihin. Kun kaikki prototyypit olivat valmiina, pidimme yhteisen kokouksen, jossa kävimme läpi prototyypit ja mietimme yhdessä, mikä olisi toteutuskelpoisin ja taloudellisesti kannattavin toteuttaa. Lopulta päädyimme space shooter -peli-ideaan.

Peli-idean jälkeen aloimme suunnitella pelin toteutusta ja tarkastella mahdollisia kanssakilpailijoita. Suunnittelun ja kilpailijatutkimuksen jälkeen huomasimme, että paras vaihtoehto alustalle olisi iPhone tai iPad, joilla mahdollinen kilpailu olisi alhaisempaa ja pelin kehitys laitepuolella olisi tarkempaa ja haastavampaa.

Peli-idean suunnittelun jälkeen mietimme tiimin sisällä mahdollista roolijakoa (6 graafikkoa ja 3 koodaajaa). Roolijaossa koodareiden kesken päädyimme siihen, että jaoimme koodin kolmeen pääosaan, joista kukin toteuttaa oman. Toteutusosina olivat viholliset, graafinen käyttöliittymä ja pelaaja. Minun vastuulleni jäi pelaajan toteuttaminen. Pelin toteutuksen alussa suunnittelimme tiimin kesken mahdolliset yhteiset toimintatavat, yhteiset kokoukset, määräajat, jne.

Projektin alussa pidimme yhteisiä kokouksia joissa määrittelimme kuukauden tavoitteen ja kahden viikon tavoitteet. Kuukauden tavoitteen määrittelimme

yhdessä ottaen huomioon kaikkien omat tavoitteet, projektin aikataulutuksen sekä huomioiden mahdollisia muuttujia.

Projektia toteuttaessa testasimme erilaisia toteutusmenetelmiä pelaajakontrolleille. Projektin edetessä huomasimme erilaisia ongelmia kontrollien toteuttamisessa, esimerkiksi että emme pystyneet tunnistamaan sormen tarjoamaa syötettä Android- ja iPhone-laitteilla tarvitsemallamme tavalla Unity 4.3.1ssä. Esimerkiksi sormen nipistyksen tunnistaminen, jonka takia emme olisi saaneet pelin karttavalikkoa tehtyä. Koska Unity 4.3.1ssä ei ole mitään sormen liikkeen tunnistusta. Unity tunnistaa ainoastaan sormen kohdan ruudulla ja montako sormea ruudulla on yhtä aikaa.

Sormen tunnistuksen koodin pystyisi toteuttamaan itse, jos meillä olisi ollut riittävästi taitoa ja aikaa sen toteuttamiseksi. Mutta aikamme ja taitomme olivat rajalliset niin, emme voineet toteuttaa tunnistusta itse.

Aloittelevina koodareina kysyimme neuvoa vanhemmalta koodarilta, kuinka pystyisimme toteuttamaan pelaajakontrollit tarvitsemallamme tavalla. Keskustelun jälkeen saimme käyttöömme FingerGesure-komponentin ja toimintaohjeita komponentin käyttämiseen. Kun olimme tutustuneet komponenttiin ja sen käyttöön, päätimme ottaa komponentin toteutukseen mukaan.

Suurimpana syynä FingerGesture-komponentin valintaan oli, että Unity 4.3.1 versioon emme löytäneet muita yhtä hyvin toimivia, helposti käytettäviä ja hyvin dokumentoituja komponentteja. Lisäksi saimme ohjausta FingeGesture-komponenttiin, joka tarjosi meille kaiken tarvittavan kontrollien toteuttamiseen. Muita mahdollisia komponentteja oli esimerkiksi Simple Touch, Touch Controller ja Input.Touches. Valintaan vaikutti myös komponenttien hinnat, kun FingerGesture-komponentin saimme ilmaiseksi käyttöömme.

Projektin aikana Oasis-pelihautomolle tuli Sony pelinkehitysoikeudet. Sonyltä saaduista lähdekoodeista ja esimerkeistä oli pelimme kannalta paljon hyötyä.

Saimme hyviä ideoita esimerkiksi siihen, kuinka voidaan toteuttaa pelaaja-aluksen sujuva kallistuminen.

Sonyn esimerkkejä tutkiessani löysin mielenkiintoisia ratkaisuja pelaajakontrollien suunnittelusta ja niiden toteutuksesta PS Vitalle. Osaa näistä ratkaisuista pystyimme soveltamaan omassa projektissamme. Näistä esimerkeistä on omasta mielestäni myös hyötyä tulevaisuuden projekteissa.

4.2 Pelaajakontrollien toteutus Unity 4.3.1:llä

Tässä luvussa kerrotaan pelaajakontrollien toteutuksesta Unityn versiolla 4.3.1 ja FingerGesture-komponentilla. Seuraavissa alaluvuissa käsittelen esimerkkikoodia esittelemällä kuvia ja selittämällä koodin tapahtumia. Koodiesimerkissä käyttämäni muuttujat on esitelty kuvassa 20. Kuvassa käytetyt muuttujat ovat julkinen Camera-muuttuja ”RaycastCamera”, suojattu tk2dCamera-muuttuja ”UIcam”, joka on avattu inspektoriin käyttämällä [SerializeField], suojattu boolean muuttuja ”movement” ja suojattu boolean muuttuja playerIsDead.

```
public class PlayerControls : MonoBehaviour {

    public Camera RaycastCamera;
    [SerializeField]
    protected tk2dCamera UIcam;
    protected bool movement;
    protected bool playerIsDead;
```

Kuva 20. Yleiset muuttujat.

4.2.1 Pelaajan liikkuminen

Seuraavaksi esittelen, miten pelaajan liikkuminen voidaan toteuttaa käyttäen OnFingerMove-komponenttia. Pelaajan liike on toteutettu käyttäen vektoreita, joilla saadaan pelaajalle selkeä ja tarkka liike. Liikkuminen on toteutettu vektoreilla joilla pystyttiin toteuttamaan pelaajan liike rajoitta, jolloin pelaaja liikkuu siinä missä sormikin.

Pelissä oleva taustan ja pelaajan nopeus on suhteutettu pelaajan sijaintiin näytöllä, tällöin pelaaja voi säätää pelin nopeutta. Tätä mekaniikkaa tarvitsimme pelissä olevan time attack pelimuodon toteutukseen.

Kuvan 21 koodi on selitetty luvussa. Komponentin koodin aluksi tarkastetaan, onko ruudulla yksi sormi ja onko pelin tämänhetkinen tila ”GamePlay”. Tämän jälkeen suoritetaan komponentin omaa tarkastusta hyväksi käyttäen metodi (Updated), jonka sisälle pelaajan liikkuminen on toteutettu.

Pelaajan liike hidastuu, kun pelaajan vie ruudun alareunaan ja liike nopeutuu, kun pelaajan vie ruudun yläreunaan. Pelaajan sijaintia ruudulla lasketaan 0 ja 1 välille. Kun pelaaja on kohdassa 0.775f tarkoittaa tämä, että pelaaja on melkein ruudun yläreunassa ja luku 0.10f tarkoittaa, että pelaaja on ruudun alareunassa.

Luodaan uusi Vector3-muuttuja ”fingerPos3d”. Tarkastetaan metodilla ”ProjectScreenPointOnDragPlane() sormen oikea korkeus ruudulla ja palautetaan ”fingerPos3d”-muuttujalle arvo Unityn 3D-maailmaan. Luodaan uusi ”Vector3”-muuttuja ”camBounds” ja annetaan muuttujalle arvo käyttämällä Unity Raycastiä ”RaycastCamerasta” arvolla ”fingerPos3d”.

```
void OnFingerMove( FingerMotionEvent e )
{
    if(e.Finger.Index == 0)
    {
        if(GameManager.Instance.CurrentGameState == GameManager.GameState.Gameplay)
        {

            if(e.Phase == FingerMotionPhase.Started)
            {
            }

            if( e.Phase == FingerMotionPhase.Updated )
            {
                Vector3 fingerPos3d;

                if( ProjectScreenPointOnDragPlane( transform.position, e.Position, out fingerPos3d ) )
                {
                    Vector3 camBounds = RaycastCamera.camera.WorldToViewportPoint(fingerPos3d);
                    if(GameManager.Instance.CurrentGameMode == GameManager.GameMode.TimeRun2)
                    {
                        if(GameManager.Instance.DifficultySpeed > 0.5f)
                        GameManager.Instance.DifficultySpeed -= Time.deltaTime * 0.1f;
                    }
                    if(camBounds.y < 0.775f)
                    {
                        transform.position = fingerPos3d + new Vector3(0f,7f,0f);
                    }
                    else
                    {
                        transform.position = new Vector3(fingerPos3d.x, transform.position.y, transform.position.z);
                    }

                    if(camBounds.y < 0.05f)
                    {
                        GameManager.Instance.BoostSpeed = (15 * GameManager.Instance.DifficultySpeed);
                    }
                    else if(camBounds.y < 0.10f)
                    {
                        GameManager.Instance.BoostSpeed = (16 * GameManager.Instance.DifficultySpeed);
                    }
                    else if(camBounds.y < 0.20f)
                    {
                        GameManager.Instance.BoostSpeed = (17 * GameManager.Instance.DifficultySpeed);
                    }
                    else if(camBounds.y > 0.70f)
                    {
                        GameManager.Instance.BoostSpeed = (35 * GameManager.Instance.DifficultySpeed);
                        #if UNITY_EDITOR
                        GameManager.Instance.BoostSpeed = (500 * GameManager.Instance.DifficultySpeed);
                        #endif
                    }
                    else if(camBounds.y > 0.60f)
                    {
                        GameManager.Instance.BoostSpeed = (30 * GameManager.Instance.DifficultySpeed);
                    }
                    else if(camBounds.y > 0.50f)
                    {
                        GameManager.Instance.BoostSpeed = (24 * GameManager.Instance.DifficultySpeed);
                    }
                    else
                    {
                        GameManager.Instance.BoostSpeed = (18 * GameManager.Instance.DifficultySpeed);
                    }
                }
            }

            if(e.Phase == FingerMotionPhase.Ended)
            {

            }
        }
    }
}
```

Kuva 21. OnFingerMove.

Tarkastetaan onko pelin tila ”TimeRun2” ja onko ”DifficultySpeed” suurempi kuin 0.5f. Jos ehdot ovat totta, vaihdetaan muuttujan ”DifficultySpeed” arvoa, laskemalla (DifficultySpeed - pelin aika * 0.1) lauseke. Tämän jälkeen tarkastetaan onko sormen sijainti kameran koordinaatistossa y pienempi kuin 0.775f. Jos ehto käy toteen, päivitetään pelaajan sijaintia sormen sijainnilla + Vector3.y:n arvolla 7.

Jos arvo on suurempi kuin 0.775f, päivitetään pelaajan sijaintia sormen sijainnilla x ja pelaajan omalla sijainnilla y ja z. Tarkastetaan pelaajan sijaintia kameran koordinaatistossa y ja määritetään pelaajan sijainnin mukaan muuttujaa ”BoostSpeed” esimerkiksi 17 * ”DifficultySpeed”.

4.2.2 Pelin pysäytys

Pelissä olevan pysäytyksen toiminta on toteutettu yhdelle kädelle sopivaksi kokonaisuudeksi. Pelin pysäytys ei pysäytä peliä kokonaan vaan hidastaa sitä, jolloin peliin saadaan ominaisuus, jolla voidaan suunnitella taktisia liikkeitä vihollisia vastaan.

Pelin pysäytys voidaan toteuttaa OnFingerUp- (Kuva 22) ja OnFingerDown-komponentilla (Kuva 23). Komponentin OnFingerUp koodin aluksi tarkastetaan, montako sormea näytöllä on, käyttäen e.Finger.Indexiä, joka palauttaa sormien lukumäärän indeksinä. Jos sormien lukumäärä on 1, haetaan pelaajalta boolean muuttuja onko pelaaja elossa vai ei. Seuraavaksi luodaan uusi muuttuja ”lives” ja haetaan pelaajan elämät muuttujaan.

```
void OnFingerUp(FingerUpEvent e)
{
    if(e.Finger.Index == 0)
    {
        playerIsDead = gameObject.GetComponent<Player>().PlayerDead();
        int lives = gameObject.GetComponent<Player>().PlayerLives;

        if(lives >= 0 && GameManager.Instance.CurrentGameState == GameManager.GameState.Gameplay)
        {
            if(!playerIsDead)
            {
                GameManager.Instance.CurrentGameState = GameManager.GameState.Pause;

            }
        }
    }
}
```

Kuva 22. OnFingerUp.

```
void OnFingerDown(FingerDownEvent e)
{
    if(e.Finger.Index == 0)
    {
        if(GameManager.Instance.CurrentGameState == GameManager.GameState.Pause)
        {
            Ray ray = UIcam.camera.ScreenPointToRay( e.Position );
            RaycastHit hit;

            if(Physics.Raycast(ray, out hit, 100f))
            {
                if(hit.collider.name == "ResumeBtn")
                {
                    GameManager.Instance.CurrentGameState = GameManager.GameState.Gameplay;
                }
            }
        }
    }

    if(e.Finger.Index == 1)
    {
        Ray ray = RaycastCamera.ScreenPointToRay( e.Position );
        RaycastHit hit;

        if(Physics.Raycast(ray, out hit, 100f, (1 << 12)))
        {
            if(hit.collider.tag == "Power")
            {
                Debug.Log("SCREEN CLEAR");
            }
        }
    }
}
```

Kuva 23. OnFingerDown.

Seuraavaksi koodi tarkastaa, onko pelaajan elämät suurempi kuin tai yhtä suuri kuin 0 ja onko pelin tila ”GamePlay”. Tämän jälkeen tarkistetaan, onko pelaaja vielä elossa vai ei. Jos pelaaja on vielä elossa, pelin tila vaihdetaan pysäytykselle.

OnFingerDown-komponentin koodin aluksi tarkastetaan, montako sormea on näytöllä. Seuraavaksi tarkastetaan, onko pelin tila ”Pause”. Pelin tilan tarkastuksen jälkeen luodaan uusi Ray-luokan mukainen olio ja nimetään muuttuja nimellä ”ray”. Tämän jälkeen annetaan muuttujalle arvo käyttäen UI-kameraa ja sormen sijaintia ruudulla. Luodaan muuttuja RaycastHit nimellä ”hit”.

Kutsutaan Raycast-metodia ja annetaan sille arvot ray, hit ja 100f. Tarkastetaan osuuko Raycast objekteihin joiden nimi on ”ResumeBtn”. Jos Raycast osuu oikeaan objektiin, vaihdetaan pelin tilaksi ”GamePlay”. Tarkastetaan, onko ruudulla myös toinen sormi ja kutsutaan sormelle Raycast-metodia arvoilla ray, hit, 100f ja varmistetaan, että Raycast osuu vain Unityn tasolle 12. Sitten tarkastetaan osuuko Raycast objekteihin joiden merkki on ”Power”. Jos osuu, tulostetaan ”SCREEN CLEAR”.

4.3 Pelaajakontrollien toteutus Unity 4.5.1:llä

Vaihtaessamme Unityn versioon 4.5.1 huomasimme yhteensopivuus ongelmia FingerGesture komponentin kanssa, jolloin päädyimme tulokseen toteuttaa pelaaja kontrollit käyttäen Unityn omia kontrolleja. Luvussa siis esitellään toinen toteutustapa toteuttaa pelaajakontrollit Unityn versiolla 4.5.1 ja Unityn omia kontrolleja käyttäen.

Seuraavissa alaluvuissa käsittelen esimerkkikoodia esittelemällä kuvia ja selittämällä koodin tapahtumia. Koodiesimerkissä käyttämäni muuttujat on esitelty kuvassa 24. Kuvassa käytetyt muuttujat ovat julkinen Camera-muuttuja ”RaycastCamera”, suojattu tk2dCamera-muuttuja ”UIcam”, suojattu boolean-muuttuja ”movement”, suojattu boolean-muuttuja playerIsDead ja suojattu boolean-muuttuja ”FingerDown”.

```
public class PlayerControls : MonoBehaviour {

    public Camera RaycastCamera;
    protected tk2dCamera UIcam;
    protected bool movement;
    protected bool playerIsDead;
    bool fingerDown = false;
```

Kuva 24. Yleiset muuttujat 2.

4.3.1 Pelaajan kontrollit

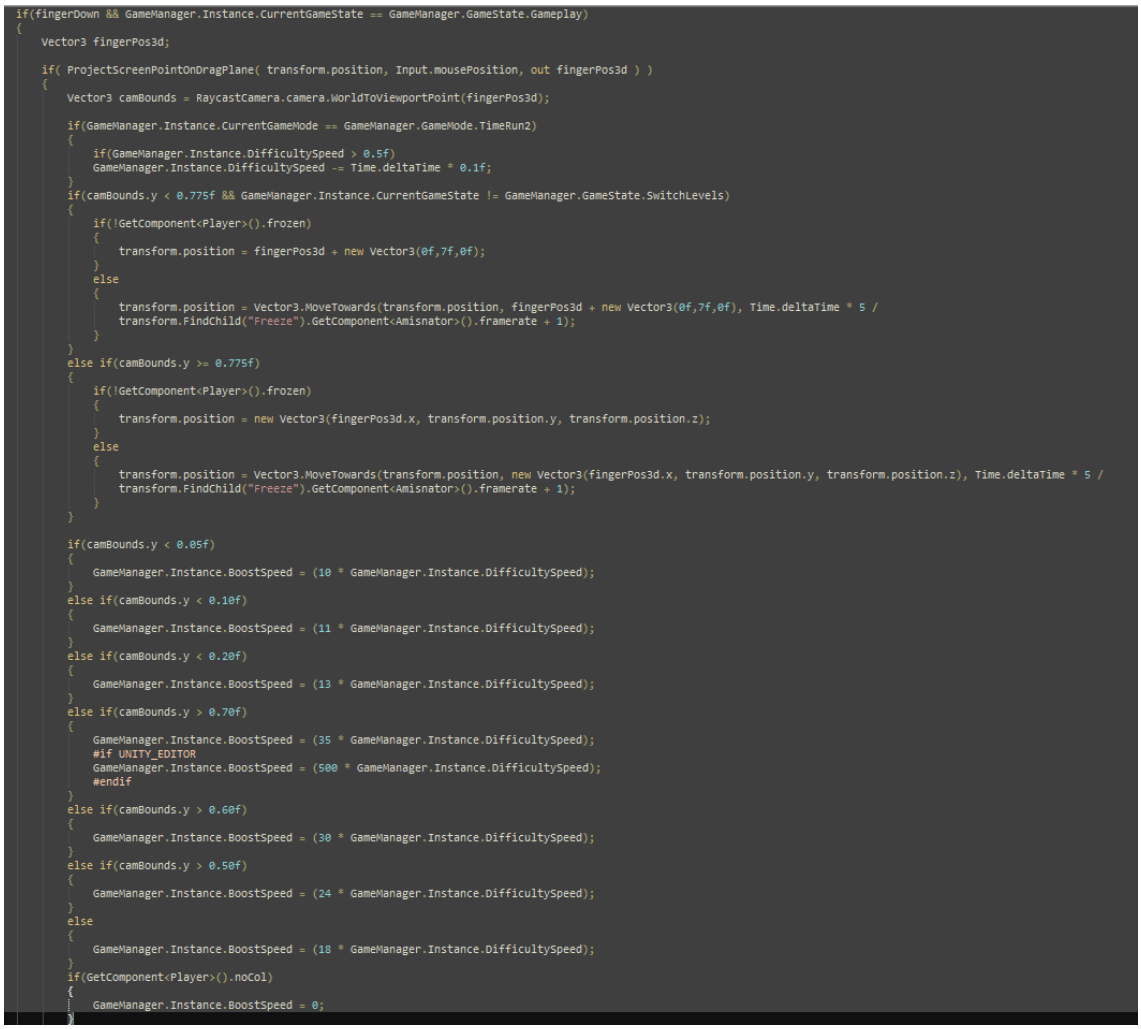
Pelaajan kontrollit on toteutettu Unityn Update()-metodin sisällä. Ensimmäiseksi (Kuva 25) tarkastetaan, onko hiiren vasen painike pohjassa ja onko ”fingerDown”-muuttuja epätosi. Jos ehto käy toteen, muutetaan ”fingerDown”-muuttuja todeksi. Sitten koodi tarkastaa onko tämän hetkinen pelin tila ”Pause”. Jos pelin tila on pause, luodaan uusi Ray-luokan mukainen olio ja annetaan sille arvo UI-kamerasta lähtevän rayn mukaan sormen tai hiiren kohta ruudulla. Luodaan muuttuja ”RaycastHit” ”hit”. Kutsutaan Raycast-oliota arvoilla ray, hit, ja 100f. Tarkastetaan osuuko Raycast objektiin jonka nimi on ”ResumeBtn”, jos Raycast osuu, vaihdetaan pelin tilaksi ”GamePlay”.

```
if(Input.GetMouseButtonDown(0) && !fingerDown)
{
    fingerDown = true;
    if(GameManager.Instance.CurrentGameState == GameManager.GameState.Pause)
    {
        Ray ray = UIcam.camera.ScreenPointToRay( Input.mousePosition );
        RaycastHit hit;

        if(Physics.Raycast(ray, out hit, 100f))
        {
            if(hit.collider.name == "ResumeBtn")
            {
                GameManager.Instance.CurrentGameState = GameManager.GameState.Gameplay;
            }
        }
    }
}
if(Input.GetMouseButtonUp(0) && fingerDown)
{
    fingerDown = false;
    Time.timeScale = 1.0f;
    GameManager.Instance.ResetFps();
    playerIsDead = gameObject.GetComponent<Player>().PlayerDead();

    int lives = GameManager.Instance.PlayerTotalLives;
    if(lives >= 0 && GameManager.Instance.CurrentGameState == GameManager.GameState.Gameplay)
    {
        if(!playerIsDead)
        {
            GameManager.Instance.CurrentGameState = GameManager.GameState.Pause;

        }
    }
}
```

Kuva 25. Update osio 1.

Seuraavaksi (Kuva 25) koodi tarkastaa onko hiiren vasen painike nostettu ylös ja onko ”fingerDown”-muuttuja totta, jos ehto käy toteen. Annetaan ”fingerDown” muuttujalle arvo epätosi, annetaan arvo unityn Time.timescale 1.0f, kutsutaan metodia ResetFps(), haetaan muuttujalle playerIsDead arvo metodilla PlayerDead() ja luodaan muuttuja ”lives” ja haetaan muuttujalle arvo PlayerTotalLives. Koodi tarkastaa onko muuttuja ”lives” suurempi tai yhtäsuuri kuin 0 ja onko pelin nykyinen tila ”GamePlay”. Jos ehto käy toteen tarkastetaan onko muuttuja ”playerIsDead” epätosi. Jos muuttuja on epätosi, vaihdetaan pelin nykyiseksi tilaksi ”Pause”.

Koodi (Kuva 26) tarkastaa onko muuttuja ”fingerDown” tosi ja onko pelin nykyinen tila ”GamePlay”. Jos ehto käy toteen luodaan uusi ”Vector3”-muuttuja fingerPos3d. Seuraavaksi koodi tarkastaa metodilla ”ProjectScreenPointOnDragPlane() sormelle oikean korkeuden ruudulla ja palautetaan ”fingerPos3d”-muuttujalle arvo Unityn 3D-maailmaan.



```
if(fingerDown && GameManager.Instance.CurrentGameState == GameManager.GameState.Gameplay)
{
    Vector3 fingerPos3d;

    if( ProjectScreenPointOnDragPlane( transform.position, Input.mousePosition, out fingerPos3d ) )
    {
        Vector3 camBounds = RaycastCamera.camera.WorldToViewportPoint(fingerPos3d);

        if(GameManager.Instance.CurrentGameMode == GameManager.GameMode.TimeRun2)
        {
            if(GameManager.Instance.DifficultySpeed > 0.5f)
            GameManager.Instance.DifficultySpeed -= Time.deltaTime * 0.1f;
        }
        if(camBounds.y < 0.775f && GameManager.Instance.CurrentGameState != GameManager.GameState.SwitchLevels)
        {
            if(!GetComponent<Player>().frozen)
            {
                transform.position = fingerPos3d + new Vector3(0f,7f,0f);
            }
            else
            {
                transform.position = Vector3.MoveTowards(transform.position, fingerPos3d + new Vector3(0f,7f,0f), Time.deltaTime * 5 /
                transform.FindChild("Freeze").GetComponent<Amisnator>().framerate + 1);
            }
        }
        else if(camBounds.y >= 0.775f)
        {
            if(!GetComponent<Player>().frozen)
            {
                transform.position = new Vector3(fingerPos3d.x, transform.position.y, transform.position.z);
            }
            else
            {
                transform.position = Vector3.MoveTowards(transform.position, new Vector3(fingerPos3d.x, transform.position.y, transform.position.z), Time.deltaTime * 5 /
                transform.FindChild("Freeze").GetComponent<Amisnator>().framerate + 1);
            }
        }

        if(camBounds.y < 0.05f)
        {
            GameManager.Instance.BoostSpeed = (10 * GameManager.Instance.DifficultySpeed);
        }
        else if(camBounds.y < 0.10f)
        {
            GameManager.Instance.BoostSpeed = (11 * GameManager.Instance.DifficultySpeed);
        }
        else if(camBounds.y < 0.20f)
        {
            GameManager.Instance.BoostSpeed = (13 * GameManager.Instance.DifficultySpeed);
        }
        else if(camBounds.y > 0.70f)
        {
            GameManager.Instance.BoostSpeed = (35 * GameManager.Instance.DifficultySpeed);
            #if UNITY_EDITOR
            GameManager.Instance.BoostSpeed = (500 * GameManager.Instance.DifficultySpeed);
            #endif
        }
        else if(camBounds.y > 0.60f)
        {
            GameManager.Instance.BoostSpeed = (30 * GameManager.Instance.DifficultySpeed);
        }
        else if(camBounds.y > 0.50f)
        {
            GameManager.Instance.BoostSpeed = (24 * GameManager.Instance.DifficultySpeed);
        }
        else
        {
            GameManager.Instance.BoostSpeed = (18 * GameManager.Instance.DifficultySpeed);
        }
        if(GetComponent<Player>().noCol)
        {
            GameManager.Instance.BoostSpeed = 0;
        }
```

Kuva 26. Update osio 2.

Seuraavaksi koodi tarkastaa onko pelin nykyinen tila ”TimeRun2”. Jos ehto käy toteen, tarkastetaan onko muuttuja ”DifficultySpeed” suurempi kuin 0.5f. Jos ehdot ovat totta, vaihdetaan muuttujan ”DifficultySpeed” arvoa, DifficultySpeed - pelin aika * 0.1. Tämän jälkeen tarkastetaan onko muuttujan ”camBounds.y:n” arvo pienempi kuin 0.775f ja onko pelin nykyinen tila erisuuri kuin ”SwitchLevels”. Jos ehto käy toteen, tarkastetaan onko ”frozen”-muuttuja epätosi. Jos ehto käy toteen, päivitetään pelaajan sijaintia, sormensijainnilla + Vector3.y:n arvolla 7.

Jos ehto ei käy toteen, päivitetään pelaajan sijaintia käyttämällä metodia Vector3. MoveTowards() arvoilla ”pelaajan sijainti x, y, z” ja uudella ”Vector3”-muuttujalla ”sormen sijainti x”, ”pelaajan sijainti.y”, ”pelaajan sijainti.z” ja annetaan metodille nopeus ”Time.deltatime * 5 / ”framerate” + 1. Seuraavaksi koodi tarkastaa onko muuttujan ”camBounds.y:n” arvo suurempi tai yhtäsuuri

kuin 0.775f. Jos ehto käy toteen, tarkastetaan onko muuttuja ”frozen” epätosi. Jos muuttuja on epätosi, päivitetään pelaajan sijaintia uudella ”Vector3” arvolla ”sormen sijainti.x”, ”pelaajan sijainti.y” ja ”pelaajan sijainti.z”. Jos ehto ei käy toteen, päivitetään pelaajan sijaintia, käyttäen metodia Vector3.MoveTowards(), arvoilla ”pelaajan sijainti x, y, z”, uusi ”Vector3” arvoilla ”sormen sijainti.x, ”pelaajan sijainti.y”, ”pelaajan sijainti.z” ja annetaan metodille nopeus ”Time.deltatime * 5 / ”framerate” + 1.

Seuraavaksi koodi tarkastaa muuttujan ”camBounds.y:n” arvoa ja muuttaa muuttujaa ”BoostSpeed” arvoilla X * ”DifficultySpeed”, riippuen siitä mikä on muuttujan ”camBounds.y:n” arvo. Koodin lopuksi tarkastetaan vielä onko muuttuja ”noCol” tosi. Jos ehto käy toteen, vaihdetaan muuttujan ”BoostSpeed” arvoksi 0.

5 Yhteenveto

Työni päätavoitteena oli tehdä toimiva pelaajakontrolliratkaisu julkaistavaan peliprojektiin ja lopuksi tehdä siitä oma opinnäytetyö. Aikaa kokonaisuudessaan opinnäytetyön tekemiseen kului yhdeksän kuukautta, joista 6 kuukautta meni käytännön työn tekemiseen ja loput kolme kuukautta käytin opinnäytetyön kirjalliseen osuuteen.

Koodin toteutustapa jolla teimme koodin, ei ehkä ollut mikään paras vaihtoehto varsinkin, kun koodin kommentointi jäi vähäiseksi. Näin jälkikäteen viisaampana on hyvä huomioida miten tärkeää on alkuperäinen koodin suunnittelu ja sen kommentointi. Koodi on kuitenkin omasta mielestäni toteutettu uudelleen käyttökelpoisena sekä koodissa on laajennuksen varaa esimerkiksi aktivoitaville erikois aseille. Alkuperäiseen koodiin on sisällytetty jo tämän koodin testi pohja, jolla voidaan aktivoida painikkeita pelin aikana käyttäen toista sormea.

Työn tavoitteena oli myös, tutkia muiden tekijöiden pelaajakontrolliratkaisuja sekä tutustua top-down-scroller-pelien ominaisuuksiin. Opinnäytetyötäni tehdessä yritin etsiä internetistä mahdollisia esimerkkejä muiden toteuttamista pelaaja kontrolleista. Suurin osa löydetyistä esimerkeistä selittää vain miten objektia A voidaan liikuttaa painamalla painiketta B sijaintiin C. Lopputuloksena päädyin opettajan kanssa siihen, että pelien lähdekoodit ovat yleensä suojattua materiaalia, johon ulkopuolisena henkilönä on hyvin vaikea päästä käsiksi. Onneksi opinnäytetyöni aikana pääsin tutustumaan muutamaan muiden toteuttamiin pelaajakontrolliratkaisuihin kuten Trivasion. Top-down-scroller-pelien ominaisuuksiin tutustuin työni aikana tutkimalla niitten toimintaa sekä, toteuttamalla omia ominaisuuksia käytännön työhön.

Joten löytääkseni käytännön esimerkkejä muista pelaajakontrolli toteutuksista minun täytyi turvautua muihin peleihin joita olin pelannut esimerkiksi Shogun ja Squadron. Pelejä tutkiessani huomasin kuinka pelaaja liikkuu sulavasti sormen mukana ja pelaajan liikettä ei ole rajoitettu maksimi nopeuksilla. Pelaajakontrolli

ratkaisuja testattuani, huomasin kuinka samankaltaisia ne olivat. Suurin osa pelaajanliikkeestä oli toteutettu yhdellä sormella toimivaksi kokonaisuudeksi. Omassa projektissamme toteuttamani pelaajanliike on yhdellä sormella toimiva kokonaisuus, koska testatessamme eri kontrolli vaihtoehtoja huomasimme tämän olevan yksinkertainen ja helppokäyttöinen kokonaisuus. Testi peleissä liike tuntui sulavalta ilman viivettä, joten pyrin toteuttamaan samankaltaista liikettä käyttäen vektoreita liikkeen toteutukseen.

Käytännön projekti antoi hyvää kokemusta ja taitoa tulevaisuuden projekteihin. Omasta mielestäni pienen projektin toteuttaminen hyvällä ryhmällä antaa paljon, koska hyvässä ryhmässä on joustamisen varaa suuntaan ja toiseen. Jouston avulla pystyimme olemaan stressaamatta vaikeissakin tilanteissa. Aloittelevalle ohjelmoijalle työ toi paljon uutta tietoa ja taitoa ohjelmien käyttämisestä sekä koodaamisesta.

Opin työn aikana, kuinka Unity 3D toimii ja minkälaisia työkaluja Unityyn voi lisätä (esimerkiksi 2D Toolkit ja Amplify Color). Opin myös uusia näkökulmia koodin toteuttamiseen toisten koodareiden kanssa, kuten koodin jakaminen. Esimerkiksi meidän tapauksessamme jaoimme koodiin viholliset, GUI ja pelaaja. Opin myös, miten SVN-palvelimen käyttö toimii ja miten se helpottaa työskentelyä. Esimerkiksi SVN vertailee vanhaa ja uutta versiota, minkä jälkeen se yhdistää koodin jos vanhan ja uuden version välillä on tapahtunut muutoksia, eikä vain yli kirjoita koodia.

Kirjallista osuutta kirjoittaessani löysin mielenkiintoisia pelaajakontrollitoteutuksia eri peli-alustoilta kuten PS Vitalta. Niistä selvisi esimerkiksi, miten pelaajan liikkumisen toteutuksessa pystytään luomaan sulava liike, ilman ruudun tärinää tai kuinka PS Vitan kosketusnäytöt toimivat ja miten niitä voidaan käyttää sormella toimivien kontrollien kanssa. Näiden kokemusten jälkeen pystyin parantamaan omaa osaamistani toteuttaessani pelaajakontrolleja. Näköpiirissäni on jo tuleva projekti johon pääsen toteuttamaan pelaajakontrollit ja aion käyttää opinnäytetyössäni saatua kokemusta ja näkökulmia.

Lähteet

Virgin Interactive Entertainment. 10.31.1997. http://en.wikipedia.org/wiki/Blade_Runner_%281997_video_game%29 26.3.2015.

Electronic Arts. 01.14.2003. SimCity 4 https://www.origin.com/fi-fi/store/buy/simcity-2013-/mac-pc-download/base-game/standard-edition?utm_campaign=origin-search-fi-pbm-g-sim13-b&utm_medium=cpc&utm_source=google&utm_term=+sim%20+city&sourceid=origin-search-fi-pbm-g-sim13-b 26.3.2015

FingerGestures Documentation. 2013a. Detecting Gestures. http://fingergestures.fatalfrog.com/docs/manual:gestures:start 27.01.2015

FingerGestures Documentation. 2013b. Interacting with scene objects http://fingergestures.fatalfrog.com/docs/manual:object_interaction 27.01.2015

FingerGestures Documentation. 2013c. Detecting Finger Events. http://fingergestures.fatalfrog.com/docs/manual:fingers:start 27.01.2015

FingerGestures Documentation. 2013d. Using the Pinch Recognizer. http://fingergestures.fatalfrog.com/docs/manual:gestures:pinch 27.01.2015

FingerGestures Documentation. 2013e. Using the Tap Recognizer. http://fingergestures.fatalfrog.com/docs/manual:gestures:tap 27.01.2015

FingerGestures Documentation. 2013f. Using the Swipe Recognizer. http://fingergestures.fatalfrog.com/docs/manual:gestures:swipe 27.01.2015

Geig M. 2013. Managing Player Input in Unity. https://blog.safaribooksonline.com/2013/11/26/managing-player-input-in-unity/ 28.01.2015

Llopis, N. 2011. Programming Fundamentals. Teoksessa S. Rabin Introduction to Game Development: Second Edition. 209 – 320.

https://books.google.fi/books?id=79ud9_8mbgYC&pg=PA250&lpg=PA250&dq=player+inputs+most+important+part&source=bl&ots=HgoyzTRiVR&sig=ZN3xeKO2NnMjjNgtboXCw_LEqVk&hl=fi&sa=X&ei=PZTkVN2-EK-PkyAPxvoKQBg&ved=0CC8Q6AEwAg#v=onepage&q=player%20inputs%20most%20important%20part&f=false 18.02.2015

int13.net – Smartphone Games. 9.4.2012. Shogun. http://www.shogun-mobile.com/ 24.2.2015

New Straits Times. 1998. Scrolling. http://news.google.com/newspapers?id=drgTAAAAIBAJ&sjid=S5ADAAAAIBAJ&pg=3478,303305&dq=parallax+scrolling 19.02.2015

Origin Systems. 26.9.1990. Wing Commander. http://en.wikipedia.org/wiki/Origin_Systems 24.2.2015

Studio Evil. 24.10.2013. Syder Arcade. http://www.syderarcade.com/ 24.2.2015

Timonen, R. 4.2.2015. Tutkimushaastattelu. Pelaaja kontrollien toteutus Trivasion pelissä. Joensuu.

Tim Rogers. 22.2.2013. http://www.gamasutra.com/view/feature/187126/lets_talk_about_touching_making_.php 24.2.2015

Wikipedia. 22.2.2015a. Arcade game. http://en.wikipedia.org/wiki/Arcade_game 24.2.2015

Wikipedia. 30.1.2015b. Shoot ’em up. http://en.wikipedia.org/wiki/Shoot_%27em_up 24.2.2015

Wikipedia. 22.2.2015c. Video game genres. http://en.wikipedia.org/wiki/Video_game_genres 24.2.2015

Wikipedia. 5.12.2015d. Side-scrolling video game. http://en.wikipedia.org/wiki/Side-scrolling_video_game 24.2.2015

Wikipedia. 26.6.2013. Vertically scrolling video game. http://en.wikipedia.org/wiki/Vertically_scrolling_video_game 24.2.2015e

Wikipedia. 3.6.2015f. 2.5D. http://en.wikipedia.org/wiki/2.5D 3.6.2015

Wikipedia. 15.12.2014. Video game graphics.
http://en.wikipedia.org/wiki/Video_game_graphics 24.2.2015