Nico Behnen

Yleiskäyttöinen hahmo-ohjain 2D-tasohyppelypeleihin

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikka

Insinöörityö

3.12.2021

Tiivistelmä

Tekijä:	Nico Behnen
Otsikko:	Yleiskäyttöinen hahmo-ohjain 2D-tasohyppelypeleihin
Sivumäärä:	38 sivua + 3 liitettä
Aika:	3.12.2021
Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Tieto- ja viestintätekniikka
Ammatillinen pääaine:	Pelisovellukset
Ohjaaja:	Lehtori Antti Laiho

Insinöörityön tarkotuksena oli yleiskäyttöisen hahmo-ohjaimen kehittäminen 2D-tasohyppelypeleille hyödyntäen Unity-pelimoottoria. Työssä perehdyttiin 2D-tasohyppelypelilajityypin historiaan ja hahmo-ohjaimen kehittymiseen, minkä pohjalta luotiin oma yleiskäyttöinen hahmo-ohjain Unity-pelimoottorin avulla. Yleiskäyttöistä hahmo-ohjainta testattiin jäljentämällä kolmen eri 2D-tasohyppelypelin pelituntumaa.

Yleiskäyttöisen hahmo-ohjaimen tavoitteena oli osata käsitellä yksinkertaisia törmäyksiä ja mahdollistaa erilaisten pelituntumien saavuttaminen säädettävillä liike- ja hyppyominaisuuksilla. Yleiskäyttöistä hahmo-ohjainta pitäisi pystyä käyttämään erilaisten pelituntumien testailussa ja mahdollisesti jopa pohjana pelin hahmo-ohjainta luodessa. Tarkoituksena oli myös työn avulla syventyä hahmo-ohjaimen ominaisuuksiin, jotka tekevät pelituntumasta miellyttävää.

Yleiskäyttöisen hahmo-ohjaimen testauksessa jäljennetyt pelit olivat Super Mario Bros, Celeste ja Hollow Knight, joista jokaisella oli erilainen pelituntuma. Jokaisesta pelistä replikoitiin osa ensimmäistä tasoa mukaan lukien hahmomalli, taustagrafiikka ja tasanteiden sekä esteiden asettelu. Tämän ja metronomin avulla oli mahdollista mitata pelihahmon liikkeitä replikoidulla pelikentällä.

Lopputuloksena projektissa toteutettu yleiskäyttöinen hahmo-ohjain todettiin soveltuvaksi erilaisten pelituntumien testailuun ja tasohyppelypelien prototyyppien tekemiseen, mutta ei kuitenkaan käytettäväksi suoraan pohjana pelinkehityksessä. Isoimmaksi puutteeksi jäi rajoitteinen törmäysjärjestelmä, jonka laajentaminen on hankalaa.

Avainsanat: hahmo-ohjain, 2D-tasohyppelypelit, Unity-pelimoottori, pelituntuma

Abstract

Author:	Nico Behnen
Title:	General-purpose character controller for 2D-platformer games
Number of Pages:	38 pages + 3 appendices
Date:	3 December 2021
Degree:	Bachelor of Engineering
Degree Programme:	Information and Communication Technology
Professional Major:	Game Applications
Supervisor:	Antti Laiho, Senior Lecturer

The subject of this thesis is to develop a general-purpose character controller for 2D-platformer games using Unity game engine. This report explores the history of 2D-platformers and the evolution of character controllers which are later used as a base for developing a general-purpose character controller with Unity game engine. The general-purpose character controller will be tested by replicating the game feel of three different 2D-platformers.

The goal of the general-purpose character controller is to be able to handle simple collisions and achieve different game feels with malleable moving and jumping mechanics. The general-purpose character controller should also be usable for testing out different game feels and even be used as a basis for game development. This thesis also served as a way to get a deeper understanding on what features and properties are required from a character controller for a pleasant game feel.

The games replicated for testing the general-purpose character controller were Super Mario Bros, Celeste, and Hollow Knight. All of them had a distinct game feel. Each game had a part of their first level replicated including the character model, background graphics and the positioning of platforms and obstacles. With the help of a metronome, this enabled the measuring of character movement on the replicated level.

As a result, the general-purpose character controller was found to be serviceable for testing-out different game feels and platformer prototypes. However, it was unsuitable to be used as a base for game development. The biggest drawback turned out to be the limited collision system that is difficult to expand upon.

Keywords: Character controller, 2D-platformer, Unity game-engine, game feel

Sisällys

1 Johdanto

2D-tasohyppelypelit ovat olleet osana videopelihistoriaa lähes alusta asti, ja tyylilaji on säilynyt merkittävänä osana videopelejä tähänkin päivään saakka. Nykyään pelinkehitys on helpommin lähestyttävissä kuin koskaan mm. valmiiden ilmaisten pelimoottorien ansiosta.

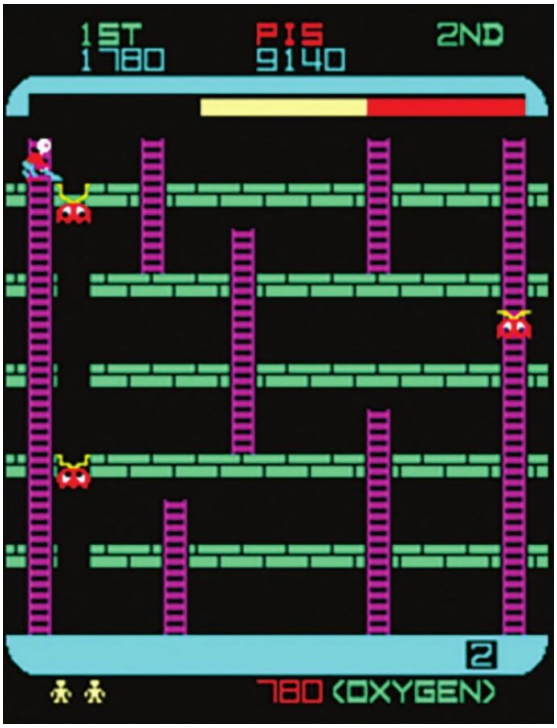
Tässä insinöörityössä tarkoituksena on yleiskäyttöisen hahmo-ohjaimen kehittäminen 2D-tasohyppelypeleille hyödyntäen Unity-pelimoottoria. Ensin raportissa tutustutaan 2D-tasohyppelypeli-lajityypin historiaan ja tarkastellaan sen vaikutusta hahmo-ohjaimen kehitykseen. Seuraavaksi syvennytään hahmo-ohjaimen perusominaisuuksiin, ja lopuksi sen pohjalta luodaan Unity-pelimoottorin avulla oma yleiskäyttöinen hahmo-ohjain, jota testataan kolmen pelin pelituntuman jäljittelemisellä.

Insinöörityön tavoitteena on tehdä hahmo-ohjain, joka osaa käsitellä yksinkertaisia törmäyksiä ja mahdollistaa erilaisten pelituntumien saavuttamisen säädettävillä liike- ja hyppyominaisuuksilla. Yleiskäyttöistä hahmo-ohjainta pitäisi pystyä käyttämään erilaisten pelituntumien testaamiseen ja mahdollisesti jopa pohjana peliprojektin hahmo-ohjainta luodessa. Tarkoituksena on myös työn avulla syventyä hahmo-ohjaimen ominaisuuksiin, jotka tekevät pelituntumasta miellyttävää.

2 2D-tasohyppelypelit

2.1 Tyylilajin muodostuminen

Tasohyppelypelit saivat alkunsa 1980-luvun alussa peliautomaattien kasvavan kehityksen myötä. Yksi ensimmäisistä tasohyppelypeleiksi luokitelluista peliautomaattipeleistä oli Universalin vuonna 1980 julkaistu Space Panic, jossa oli tavoitteena liikuttaa pelihahmoa sivultapäin kuvatulla pelikentällä väistellen ja tuhoten avaruushirviöitä. Kuvassa 1 nähtävä pelikenttä koostui kuudesta tasanteesta, joilla pelihahmo pystyi kävelemään vaakasuunnassa, sekä eripituisista tikkaista, joita pelihahmo pystyi käyttämään kiivetäkseen pystysuunnassa tasanteiden välillä. Pelikenttä oli staattinen, eikä pelihahmo pystynyt poistumaan kentältä. (1.) Pelihahmon liikuttamiseen käytettiin nelisuuntaista sauvaohjainta, jonka suuntina olivat pysty- ja vaakasuunta.



Kuva 1. Pelihahmo pudottamassa avaruushirviöitä ylimmältä tasanteelta pelissä Space Panic. Kenttä koostuu kuudesta tasanteesta ja eripituisista tikkaista. (1.)

Seuraava merkittävä tyylilajia määrittelevä tasohyppelypeli oli Nintendon peliautomaattipeli Donkey Kong vuodelta 1981. Donkey Kong perustui Space Panicin tavoin pystysuuntaiseen kiipeilyyn tasanteiden välillä hyödyntäen tikkaita. Nintendo kuitenkin toi peliin lisää syvyyttä lisäämällä hyppyominaisuuden, jolla pelihahmo pystyi väistämään vieriviä esteitä ja hyppimään tasanteelta toiselle. Donkey Kongin neljä kenttää sisälsivät myös uudentyyppisiä tasanteita, jotka kykenivät vuorovaikutukseen pelihahmon kanssa. Toisen tason liukuhihnatasanteet siirsivät pelihahmoa vaakasuunnassa liukuhihnan suunnan mukaisesti. Kolmannessa tasossa puolestaan esiintyi hissitasanteita, jotka nousivat tai laskivat toistuvasti siirtäen pelihahmoa pystysuunnassa hissitasanteiden päällä seistessä. (2.)

Donkey Kong toi muutoksia ohjausdynamiikkaan hyppyominaisuudella. Sen ansiosta ajoituksesta tuli entistä tärkeämpi osa pelihahmon ohjausta. Hyppykomento annettiin painamalla kuvassa 2 nähtävää Jump-painiketta. Donkey Kongin kenttäsuunnittelu oli myös lineaarisempaa. Kolmessa neljästä kentästä oli kentän yläreunassa selkeä päämäärä, joka pelaajan tuli saavuttaa ohjaamalla pelihahmoa. Ohjaaminen toimi samalla tavalla nelisuuntaisella sauvaohjaimella kuin Space Panicissa, ja hypylle oli erikseen sauvaohjaimen vieressä painike, jota painamalla pelihahmo hyppäsi joko suoraan ylöspäin tai pelihahmon juoksemaan suuntaan, muodostaen kuitenkin hyppykaaren matkalla.

Kuva 2. Donkey Kong -peliautomaatti. Hyppykomento annettiin keltaisella Jump-painikkeella (3).

Nintendo palasi jälleen vuonna 1985 mullistamaan tasohyppelypeligenreä julkaisemalla Super Mario Brosin Nintendon omalle kotikonsolille. Super Mario Bros sisälsi kymmeniä kenttiä, jotka jatkuivat sivulle seuraavan kameran avulla. Pelihahmoa ohjattiin peliautomaattien sauvaohjaimen sijaan kotikonsolin peliohjaimen nelisuuntaisella ristiohjaimella. Peliohjaimessa oli myös hyppynäppäimen lisäksi juoksunäppäin. Kuvassa 3 näkyvää A-näppäintä käytettiin hyppäämiseen ja B-näppäintä juoksemiseen.

Kuva 3. Nintendon NES-kotikonsolin peliohjain, jossa ristiohjain vasemmalla ja juoksu- ja hyppynäppäin oikealla (4).

Vaikka ohjaus ei fyysisesti paljoa poikennut Space Panicin ja Donkey Kongin ohjauksesta, oli Super Mario Brosin ohjaus paljon kehittyneempää. Esimerkiksi pelihahmon kiihtyminen täyteen kävelynopeuteen ei ollut enää välitöntä, vaan siihen kului aikaa. Samoin pysähtyessä piti ottaa huomioon, että pelihahmolla meni myös aikaa pysähtyä paikalleen palautuessaan kävelystä. Myös hyppyominaisuus koki isoja muutoksia Donkey Kongin jäykältä tuntuvaan hyppyyn verrattuna. Pelihahmoa oli nyt mahdollista liikuttaa myös ilmassa, ja hyppykorkeutta pystyi vaihtelemaan sillä, kuinka kauan hyppynäppäintä piti painettuna. Pelihahmo hyppäsi korkeammalle, kun hyppynäppäintä piti pidempään painettuna.

Super Mario Brosin pelihahmon ohjaus vaati pelaajalta entistä enemmän tarkkuutta. Vaihtelevan hyppykorkeuden ja liikkeen kiihtymisen ansiosta pelaajalla oli huomattavasti enemmän varaa hioa ohjaustaitoja. Liikkeen

kiihtyminen myös kannusti pelaajaa välttelemään törmäyksiä, jotka pysäyttäisivät saavutetun nopeuden edetä kentässä sulavasti.

2D-tasohyppelypelit alkoivat yksinkertaisista kiipeilypeleistä, joissa pelihahmoa pystyi liikuttamaan tasanteita pitkin ja tasanteiden välillä liikkuminen onnistui tikkaita pitkin. Hyppyominaisuus toi peleihin lisää dynaamisuutta, ja se nosti ajoituksen tärkeyttä ohjaamisessa. Lopulta ohjaus muuttui elävämmäksi antamalla pelihahmolle kiihtymistä liikkumiseen ja mahdollisuuden vaikuttaa pelihahmon hyppykorkeuteen.

2.2 Ohjaus ja hahmo-ohjain 2D-tasohyppelypeleissä

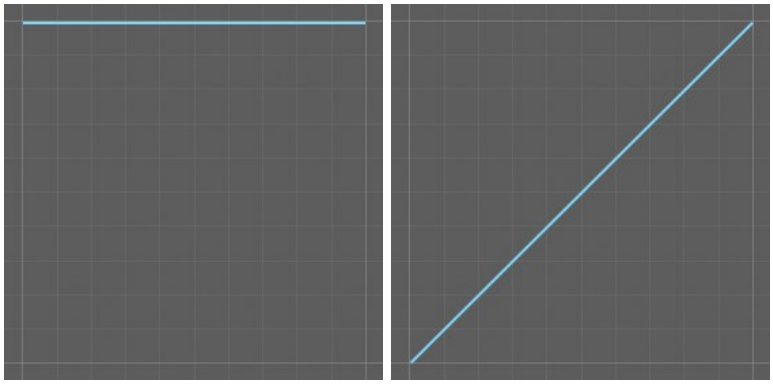
2D-tasohyppelypeleissä pelihahmon ohjaaminen on hyvin tärkeässä roolissa. Peleissä on lähes poikkeuksetta jokin päämäärä, jonka saavuttaminen vaatii pelaajalta pelihahmon ohjausta. Ohjaus tapahtuu useimmiten peliohjaimella, joka koostuu erilaisista painikkeista ja mahdollisesti sauvaohjaimista. Fyysisten painikkeiden lisäksi pelissä itsessään on oltava jonkinlainen ohjauslogiikka, joka tulkitsee pelaajan antamia syötteitä ja muuntaa ne liikkeeksi pelissä.

Insinöörityössä tätä logiikkaa kutsutaan hahmo-ohjaimeksi, joka tulee englanninkielisestä character controller -termistä. Hahmo-ohjaimen tehtävä on luoda yhteys pelaajan ja pelihahmon välille. Hahmo-ohjaimella on suuri vaikutus pelituntumaan, joka syntyy siitä, miten pelaaja ja pelihahmo kommunikoivat keskenään. Tätä kommunikointia voidaan kuvata eräänlaisena reagointikierteenä (5, s. 42), jossa pelaaja ensin tarkastelee pelin senhetkistä tilaa ja tarkastelun pohjalta antaa syötteen pelille. Tämän jälkeen kierre toistuu, kun pelaaja jälleen tarkastelee pelin uutta tilaa ja harkitsee seuraavaa syötettä. Jokaisen kierroksen jälkeen pelaaja huomaa, vastaako pelihahmon liike pelaajan odotuksia, ja tämän mukaan pelaaja oppii myös ennakoimaan pelihahmon käyttäytymistä.

Hahmo-ohjaimen tärkeimpiä ominaisuuksia on saada peli tuntumaan sujuvalta ja intuitiiviselta. Tähän voidaan päästä mm. siten, että pyritään vastaamaan

pelaajan odotuksiin mahdollisimman hyvin. Pelaajalle syntyy odotuksia esimerkiksi pelin ulkoasusta ja teemasta sekä peleistä, joita pelaaja on aikaisemmin pelannut. Toinen tärkeä ominaisuus on saada peli tuntumaan responsiiviselta. Pelaajan on helpompi saavuttaa hallinnan tuntu, jos peli antaa välitöntä palautetta syötteistä. Palautteen ei tarvitse kuitenkaan olla välttämättä liikettä, vaan se voi myös olla esimerkiksi muutos pelihahmon animaatioissa.

Super Mario Brosissa ja Donkey Kongissa ohjaus on binääristä. Toisin sanoen näppäimet ja sauvaohjain lähettävät ainoastaan kahdentyyppistä dataa pelille, koska ne voivat olla vain kahdessa eri tilassa: painettuna tai vapaana. Tämä näkyy selkeästi Donkey Kongin pelituntumassa, jossa pelihahmo liikkuu ainoastaan yhdellä nopeudella ja vain silloin, kun pelaaja parhaillaan antaa syötettä peliohjaimen kautta. Lopputuloksena on pelihahmon jäykkä ja konemainen liike. Super Mario Brosissa puolestaan binääristä syötettä ei tulkita suoraan liikkeeksi. Pelihahmolla on olemassa tavoitenopeus liikkua sivusuunnassa, ja pelaajan painaessa oikealle tai vasemmalle ristiohjaimesta, alkaa hahmo-ohjain interpoloida nopeutta paikallaanolon ja tavoitenopeuden välillä. Interpoloitu nopeus on riippuvainen siitä, kuinka kauan aikaa on kulunut siitä hetkestä, kun pelaaja on aloittanut syötteen antamisen. Välittömän ja kiihtyvän liikkeen huippunopeus voi olla täysin sama, mutta välitön ohjaus saavuttaa sen välittömästi, kuten esitettynä kuvassa 4.



Kuva 4. Y-akselilla kuvattuna nopeus ja x-akselilla aika. Vasemmalla välitön liike, joka ei sisällä kiihtymistä, ja oikealla tasaisesti kiihtyvä liike.

Super Mario Brosissa pelihahmon liike on kiihtyvää myös pysähtyessä, joten pelihahmo jatkaa liikettä vielä hetken sen jälkeen, kun pelaaja on lakannut antamasta syötettä. Tämä antaa pelihahmolle painon tuntua. Lisäksi pelaajan on opeteltava arvioimaan pelihahmon liikettä tarkemmin sen ollessa kiihtyvää. Kiihtyminen on kuitenkin aina johdonmukaista, eli pelihahmo käyttäytyy samoissa tilanteissa aina samalla tavalla, ja siksi pelaaja voi saavuttaa täyden hallinnan ohjatessaan pelihahmoa.

Hyvän pelituntuman kannalta on myös tärkeää, että hahmo-ohjain ottaa huomioon tilanteet, joissa pelaajan tekemä virhe saattaa tuntua pelaajasta siltä, että peli ei tulkinnut syötteitä oikein. Vaikka ohjaus olisi muuten tarkkaa ja miellyttävää, kaikki mahdolliset keskeytykset saattavat silti pilata pelituntuman ja saada pelin tuntumaan epätarkalta ja ennalta-arvaamattomalta.

Hyppyominaisuutta luodessa voi törmätä useaan tällaiseen ongelmaan. 2D-tasohyppelypeleissä on yleistä, että pelaaja voi hypätä ainoastaan ollessaan kosketuksessa maahan. Jos pelihahmo on ilmassa ja pelaaja haluaa pelihahmon hyppäävän heti sen osuttua maahan, on pelaajan annettava hyppykomento myös heti pelihahmon osuttua maahan. Jos pelaaja antaa komennon pelihahmon ollessa vielä ilmassa, se on hylättävä, koska muuten pelihahmo hyppäisi ilmassa. Tästä aiheutuu kuitenkin ongelma, jos pelaaja antaa hyppykomennon hieman etuajassa. Silloin hahmo-ohjain hylkää hyppykomennon, mutta pelaajasta saattaa silti tuntua, että hän antoi komennon oikeaan aikaan.

Hyppykomento voidaan kuitenkin pitää hahmo-ohjaimessa muistissa ja suorittaa heti pelihahmon osuttuaan maahan. Tällöin vältytään tarpeettomalta negatiiviselta keskeytykseltä ja pelaaja voi luottaa hallintaansa pelihahmon ohjauksessa. Pitää kuitenkin ottaa huomioon myös tilanne, jossa pelaaja antaa, syystä tai toisesta, hyppykomennon uudestaan heti pelihahmon hypättyä. Hahmo-ohjaimen ei kannata pitää komentoa muistissa muuta kuin tilanteissa, joissa se on annettu hetkeä ennen maahan osumista. Tällöin pelaaja ei voi

vahingossa jättää hyppykomentoa muistiin ja ihmetellä, miksi pelihahmo hyppää itsestään osuessaan maahan.

Toinen hyvin samanlainen ongelma saattaa tulla vastaan, kun pelaaja ohjaa pelihahmoa hyppäämään kielekkeen reunalta siten, että pelihahmo juoksee reunaa kohden. Jos pelaaja myöhästyy hetken komennon antamisesta, hylkää hahmo-ohjain taas komennon, koska hahmo-ohjaimen mukaan pelihahmo ei enää ole kosketuksissa maahan. Tässä tilanteessa voidaan antaa hieman lisäaikaa antaa hyppykomento vielä sen jälkeen, kun pelihahmo on astunut pois kielekkeeltä eikä enää ole kosketuksissa maahan. (6.)

3 Hahmo-ohjaimen kehitys Unity-pelimoottorissa

3.1 Yleiskäyttöinen hahmo-ohjain

Insinöörityön tavoitteena oli kehittää yleiskäyttöinen hahmo-ohjain, joka toimisi pohjana erilaisten 2D-tasohyppelypelien kehityksessä Unity-pelimoottorin avulla. Hahmo-ohjaimessa keskitytään monipuolisen liikkeen ja hypyn toteutukseen antaen kehittäjälle säätömahdollisuuksia vaikuttaa niiden tuntumaan. Lisäksi hahmo-ohjain osaa käsitellä yksinkertaisia törmäyksiä ja tukee pelaamista antamalla turhia pelaajan tekemiä virheitä anteeksi.

Pelihahmon vaakasuuntaista liikettä tulisi pystyä muuttamaan helposti ja monipuolisesti. Liikkeen huippunopeuden pitää olla määriteltävissä, jotta voidaan vaikuttaa siihen, mikä on suurin sallittu nopeus, jolla pelihahmo pystyy liikkumaan pelissä. Kiihtymisen huippunopeuteen pitää olla tarkkaan säädettävissä, ja se pitää jakaa useampaan komponenttiin. Positiivisen ja negatiivisen kiihtymisen pitää olla erikseen säädettävissä. Kummassakin on oltava oma kiihtymisaika, jonka aikana pelihahmo saavuttaa tavoitenopeuden, ja kiihtymistyyppi, joka voi olla lineaarista tai eksponentiaalista. Pitää myös olla vaihtoehto ottaa kiihtyminen pois käytöstä, jotta pelihahmon liikkeestä saadaan haluttaessa välitöntä.

Liikkeen pitää olla muokattavissa erikseen pelihahmon ollessa ilmassa (7). Toisin sanoen samat parametrit pitää toteuttaa kahteen kertaan. Hahmo-ohjain vaihtelee ohjaustilojen välillä pelihahmon ollessa kosketuksissa maan kanssa tai sen ollessa ilmassa. Joskus, jos ohjaus ilmassa on todella raskasta ja nopeuden saavuttaminen on hidasta, hypyt jyrkänteiden reunalta saattavat olla vaikeita toteuttaa, mikäli pelaajan ei ole mahdollista ottaa vauhtia pelihahmon hyppyyn. Tavoitteena on ratkaista ongelma viivästyttämällä siirtymistä ilmaohjaukseen muutamalla sekunnin kymmenyksellä, jolloin pelihahmo ehtii kiihtyä hypyn alussa maaohjauksella ennen siirtymistä ilma-ohjaukseen. Vaihtoehdoista pitää myös löytyä ominaisuus poistaa ilmaohjaus kokonaan käytöstä ja käyttää maa-ohjausta ilmassa.

Pelihahmon liike saattaa tuntua liian hitaalta sen vaihtaessa suuntaa, etenkin, jos kiihtyvyydet ovat hitaita. Tavoite on ratkaista tämä lisäämällä poikkeus kiihtymiseen silloin, kun pelihahmo vaihtaa suuntaa kesken liikkeen. Negatiivista kiihtymistä voidaan nopeuttaa ja aloittaa positiivinen kiihtyminen jo hieman ennen, kuin pelihahmon nopeus on kokonaan pysähtynyt. Tämä tuntuu usein luontevammalta vaihtoehdolta välittömän suunnanvaihtumisen ja täyden pysähtymisen sijaan. Lisäksi hahmo-ohjaimessa tulee olla mahdollisuus pysäyttää pelihahmon liike, vaikka nopeus ei olisikaan vielä nolla, jos pelihahmo on putoamassa jyrkänteeltä ja pelaaja on jo lakannut antamasta syötettä liikkeen suuntaan. Tämä auttaa pelaajaa ohjaamisessa estämällä hitaasti pysähtyviä pelihahmoja liukumasta tasanteiden reunoilta alas.

Hahmo-ohjaimen hyppyominaisuuden tulee tukea vaihtuvaa hyppykorkeutta (8), joka on erikseen määriteltävissä. Hypyssä pitää pystyä määrittämään maksimi korkeus hypylle ja hyppykorkeuden saavuttamiseen kuluva aika. Hyppytuntuman kannalta myös painovoima on tärkeässä roolissa, joten sen voimakkuuden pitää olla säädettävissä. Painovoima kuitenkin jatkaa pelihahmon kiihdyttämistä niin kauan, kuin pelihahmo on ilmassa, joten hahmo-ohjaimessa pitää myös olla määriteltävissä suurin sallittu nopeus pystysuuntaiselle liikkeelle, jotta nopeudet eivät kasva liian korkeiksi.

Kuten aiemmin jo mainittiin, pelaajan ohjatessa pelihahmoa hyppimään reunoilta saatetaan päätyä siihen, että pelaaja antaa hyppysyötteen myöhässä, jolloin pelihahmo ei enää ole kosketuksissa maahan ja siksi putoaa reunalta alas. Tavoitteena on ratkaista tämä samaan tapaan kuin ilmaohjauskin, eli annetaan pelaajalle lisäaikaa painaa hyppynäppäintä vielä senkin jälkeen, kun pelihahmo ei enää ole kosketuksissa maahan.

Toinen aiemmin mainittu ominaisuus on hyppykomennon pitäminen muistissa. Hahmo-ohjaimen pitää pystyä ottamaan vastaan hyppykomennot, jotka on painettu juuri ennen, kuin pelihahmo koskettaa maata. Tällöin pelaajan ei tarvitse olla liian tarkka siitä, kuinka aikaisin hyppykomennon uskaltaa antaa. Ilman ominaisuutta pelaaja saattaa usein törmätä tilanteeseen, jossa pelihahmo ei suostu hyppäämään ollessaan juuri osumassa maahan, kun hyppykomento annetaan.

Hahmo-ohjaimen tulee myös osata käsitellä yksinkertaiset törmäykset. Pelihahmo pystyy kävelemään ja hyppimään tasanteilla sekä törmäämään seiniin. Joskus pelihahmo saattaa osua tasanteiden reunoihin hypätessä ylös, ja se tuntuu väärältä varsinkin, jos pelihahmon osumalaatikko on leveämpi kuin itse hahmomalli. Tavoitteena on ratkaista tämä ongelma siirtämällä pelihahmoa oikeaan paikkaan, jotta vältetään törmäys tasanteen katon kanssa. Etäisyyden tasanteen reunasta, josta siirto toteutetaan, tulee olla käyttäjän määriteltävissä.

3.2 Unity pelimoottorina 2D-tasohyppelypelissä

Insinöörityössä käytettiin Unity-pelimoottoria. Se soveltuu hyvin nopeiden prototyyppien luomiseen ilman suurempia valmisteluita. Unity-pelimoottorissa on myös paljon valmiita ominaisuuksia, jotka nopeuttavat huomattavasti pelinkehitystä verrattuna esimerkiksi siihen, että tekisi itse myös pelimoottorin peliä varten.

Tavoitteena oli käyttää hahmo-ohjaimen testauksessa muutamia valmiita komponentteja, jotka säästävät huomattavasti aikaa. Tärkeimpänä näistä oli

kamera. Kamerakomponentin avulla saa helposti luotua sopivan kuvakulman ja ruudun koon. Kamerasta saa myös renderöityä ortografisen kuvan, joka soveltuu hyvin 2D-tasohyppelypeleihin (9). Toinen, projektin kannalta vähemmän tärkeä komponentti oli Sprite Renderer -komponentti. Sen avulla on mahdollista tuoda grafiikkaa kuvamuodossa ruudulle ja liikuttaa niitä koodilla. Projektissa sitä käytettiin pelihahmon ja pelikentän tasanteiden grafiikan näyttämiseen.

Viimeisenä komponenttina projektissa käytettiin Box Collider 2D -komponenttia. Se on fysiikkajärjestelmän komponentti, jonka avulla voidaan seurata erilaisia törmäyksiä. Projektissa sitä käytettiin kahdella eri tavalla. Ensiksi kaikki pelin tasanteet ympäröidään Box Collider 2D -komponenteilla, jotta niiden sijaintia voidaan seurata Raycast-ohjelmointirajapinnalla. Box Collider 2D -komponentti koostuu pisteistä, joiden välille osumalaatikko piirretään. Toiseksi projektissa hyödynnettiin näitä pisteitä lähettämällä Raycast-säteitä niistä eri suuntiin pelihahmon osumalaatikosta. Tämä helpottaa vähän Raycast-säteiden alkupisteiden sijoittelua käyttöliittymässä, ja osumalaatikko auttaa visualisoimaan pelihahmon rajoja törmäyksissä.

Raycast-ohjelmointirajapinta on osa Unityn Physics-kirjastoa. Sen avulla voidaan lähettää säde halutusta alkupisteestä haluttuun suuntaan. Säteen pituus on myös määriteltävissä, ja säteen osuessa osumalaatikkoon saadaan tunnistettua objekti, johon osuttiin, ja etäisyys objektiin. (10.) Mikäli objekteja on paljon ja halutaan säteiden osuvan vain tiettyihin objekteihin, voidaan käyttää tasomaskeja. Projektissa käytettiin samaa tasomaskia kaikille objekteille, joihin pelihahmo voi törmätä.

Pelihahmon osumalaatikossa on neljä pistettä, jotka muodostavat nelikulmion pelihahmon ympärille siten, että se on suunnilleen samankokoinen kuin pelihahmon malli. Joskus voi olla tarvetta säätää tarkemmin osumalaatikon kokoa esimerkiksi, jos pelihahmolla on ulokkeita, kuten hiuksia, joiden ei haluta vaikuttavan törmäykseen. Usein pelaaminen on mielekkäämpää, kun osumalaatikko on hieman hahmomallia pienempi, jolloin hahmomalli selkeästi

osuu toisiin objekteihin. Näistä neljästä pisteestä lähetetään kahdeksan sädettä siten, että säteet lähtevät aina kohtisuoraan poispäin osumalaatikosta.

Säteen pituus määritellään sen mukaan, mihin pelihahmon on tarkoitus liikkua. Hahmo-ohjain antaa toivotun liikemäärän, jonka mukaan pelihahmoa tulisi liikuttaa, ja lähetettävien säteiden suunnan ja pituuden tulee vastata toivottua liikemäärä. Mikäli säteet eivät törmää osumalaatikkoihin, silloin hahmo-ohjain voi liikuttaa pelihahmoa toivotun liikemäärän verran. Jos säteet kuitenkin törmäävät osumalaatikkoon, voidaan tarkistaa osutun objektin etäisyys ja hahmo-ohjain voi liikuttaa pelihahmoa osumaetäisyyden verran toivotun liikemäärän sijaan.

Tämä on hyvin yksinkertainen tapa käsitellä törmäyksiä, ja se on otettava huomioon kenttäsuunnittelussa. Jos kentässä on hyvin ohuita tasanteita, ne saattavat osua säteiden väliin, jolloin hahmo-ohjain ei huomaa törmäystä ja jatkaa pelihahmon liikuttamista. Säteiden määrää voi erilaisten erityistarpeiden mukaan lisäillä, mutta projektissa kaikki tasanteet olivat tarpeeksi isoja, joten ne eivät mahtuneet säteiden väliin.

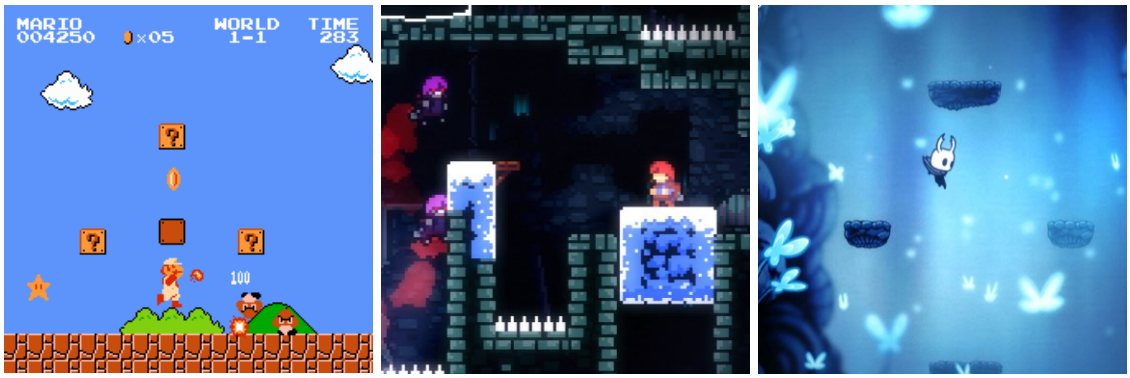
Tämä toteutustapa valittiin törmäysten käsittelylle, koska se on nopeasti toteutettavissa ja riittävä projektin tarpeisiin nähden. Unityllä on myös mahdollista käyttää valmista fysiikkamoottoria törmäysten seurantaan Rigidbody-komponentilla. Pelihahmolle annettaisiin Rigidbody-komponentti, joka simuloisi Box Collider 2D -komponenttien avulla törmäykset automaattisesti. Sama fysiikkasimulointi kuitenkin simuloisi myös pelaajan liikkeen kentällä ja siten hankaloittaisi pelituntuman hallitsemista.

Aiemmin 2D-tasohyppelypeleissä törmäysten käsittely perustui pienten laattojen välillä liikkumiseen. Kenttä oli jaettu pieniin laattoihin, joilla oli erilaisia ominaisuuksia. Osa laatoista oli esteitä, joihin pelihahmo törmäsi, ja toiset esimerkiksi tyhjiä laattoja, joiden läpi pelihahmo pystyi kulkemaan normaalisti. Pelihahmo pystyi kuitenkin liikkumaan ainoastaan laatalta toiselle eikä koskaan olemaan laattojen välissä, joten liike ei ollut kovin sulavaa. (11.)

Tämä laattoihin perustuva tekniikka kehittyi pidemmälle sallien sulavan liikkeen pelihahmolta, ja se on vielä nykyäänkin hyvä ja tarkka tekniikka käsitellä törmäyksiä. Pelihahmolla on osumalaatikko, jonka sijainnista aletaan käydä laattoja läpi etsien laattoja pelihahmon kulkusuuntaan, joihin pelihahmon voisi törmätä. Laattoja käydään läpi siltä matkalta, joka pelihahmon tulisi edetä. Jos törmäyslaattoja ei löydy, pelihahmoa liikutetaan normaalisti, ja jos taas löytyy törmäyslaatta, pelihahmoa liikutetaan ainoastaan törmäyslaatan reunaan asti. (11; 12.)

3.3 Yleiskäyttöisen hahmo-ohjaimen testaus

Tavoitteena oli testata yleiskäyttöistä hahmo-ohjainta yrittämällä kopioida kolmen erilaisen 2D-tasohyppelypelin pelituntumaa hypyn ja liikkeen osalta. Peleiksi valikoituivat Super Mario Bros, Celeste ja Hollow Knight. Super Mario Brossin liike on raskasta ja liukuvaa, hyppy tuntuu leijuvalta, mutta pelihahmo palaa maahan todella nopeasti, mikä tekee siitä samanaikaisesti tarkan. Celeste edustaa pelituntuman toista ääripäätä. Liike on lähes välitöntä, tarkkaa ja nopeaa myös ilmassa. Hollow Knight asettuu pelituntumaltaan näiden väliin, ja siinä on korkea ja leijaileva hyppy, joka eroaa kahdesta aikaisemmasta. Kaikki kolme peliä kuvassa 5 sisältävät tasohyppelypeleille ominaisia tasanteita ja niiden välillä hyppimistä, vaaroja vältellen.



Kuva 5. Projektissa käytetyt 2D-tasohyppelypelit. Vasemmalta oikealle Super Mario Bros, Celeste ja Hollow Knight (13; 14; 15).

Pelituntumaa on vaikea muuntaa numeerisiksi arvoiksi valmiista pelistä, joten tarkoitus oli replikoida jokaisesta pelistä osa kenttää käyttämällä ruutukaappauksia alkuperäisestä pelistä. Ruutukaappausten avulla voi asetella esteet ja tasanteet oikeille paikoille ja pitää ne oikean kokoisina suhteessa pelihahmoon. Taustat ja tasanteet sisältävät myös erilaisia yksityiskohtia, joiden ansiosta on mahdollista verrata esimerkiksi hyppykorkeutta ja hyppypituutta tasanteelta alkuperäisen pelin ja kopion välillä.

Kestojen ja ajoituksien mittaus ja kopiointi on puolestaan vaikeampaa. Tarkin tapa olisi varmasti tallentaa videolle liikkeet ja käydä niitä ruutu kerrallaan läpi laskien ruutujen määrän vaiheiden välillä. Esimerkiksi hypyn keston voisi selvittää laskemalla ruutujen määrän hypyn alusta lakipisteeseen ja lakipisteestä taas kosketukseen maan kanssa. Projektissa käytettiin kuitenkin videotallenteen sijaan metronomia. Sen avulla on mahdollista toistaa syötteet saman pituisilla intervalleilla. Hypyn keston voi kopioida asettamalla metronomin äänimerkit siten, että ensimmäinen kuuluu samalla hetkellä, kun hyppynäppäintä painetaan, ja toinen pelihahmon saavuttaessa hypyn lakipisteen.

Metronomia voi käyttää myös hahmon nopeuden ja kiihtymisen mittaamiseen. Ensimmäisen äänimerkin kohdalla aletaan liikuttaa pelihahmoa, ja toisen äänimerkin kohdalla lakataan liikuttamasta. Tämän jälkeen edettyä matkaa voidaan verrata käyttämällä erilaisia yksityiskohtia grafiikassa maamerkkeinä. Kun pelihahmon nopeus ja kiihtyvyys tuntuvat siltä, että ne ovat kohdallaan, metronomia voidaan vielä hyödyntää myös suunnanvaihtoon kuluvan ajan vertailussa. Voidaan valita metronomiin lyhyt intervalli äänimerkkien välille ja sitten toistaa painaen ja pitämällä painettuna suuntanäppäimiä vuorotellen äänimerkin kohdalla. Mitä nopeammin pelihahmo vaihtaa suuntaa, sitä pidemmän matkan se ehtii kulkea aina intervallin aikana.

Myös pelihahmon animaatioilla on suuri vaikutus pelituntumaan. Animaatiot lisäävät liikkeen sulavuuden tuntua ja pystyvät antamaan esimerkiksi hypyille lisää voiman tuntua (16). Animaatioiden kopiointi oli kuitenkin tämän projektin

laajuuden ulkopuolella. Animaatioruutujen löytäminen ja niiden ohjelmointi alkuperäisen pelin tapaan saattaa olla vaikeaa riippuen pelistä. Projektissa oli kuitenkin jokaisen pelin pelihahmosta staattinen kuva helpottamaan liikkeiden mittausta.

4 Yleiskäyttöisen hahmo-ohjaimen toteutus

4.1 Törmäysjärjestelmän toteutus

Insinöörityöprojektin ensimmäinen vaihe oli luoda uusi 2D-projekti Unityssa. Versioksi valikoitui aloituspäivänä uusin versio Unitysta, joka oli versio 2021.1.19f1. Projektin näkymässä oli valmiiksi kamera, jonka lisäksi piti luoda kaksi tyhjää peliobjektia pelihahmoa ja tasannetta varten. Kummastakin nollattiin transform-arvot ja lisättiin niille Box Collider 2D - ja Sprite Renderer -komponentit. Grafiikkana käytettiin aluksi kahta eriväristä 10 x 10 pikselin kuvaa.

Tasanteen kuvasta muutetaan mesh-tyyppi full rectangleksi, ja kummastakin kuvasta filter mode -asetukset muutetaan point-filteriksi. Full rectangle sopii kuvan laatoitukseen, ja point-filter poistaa kuvan pehmennysefektin. Yksi kuvista raahataan pelihahmon Sprite Renderer -komponentin Sprite-ominaisuuteen ja toinen vastaavasti tasanteen peliobjektiin. Tasanteen Box Collider 2D -komponentista asetetaan auto tiling -ominaisuus todeksi ja Sprite Renderer -komponentista draw mode -asetus tiled-tilaan. Nyt tasanteen kokoa pystyy muuttamaan suoraan Sprite Render -komponentista muokkaamalla korkeutta ja leveyttä. Lopuksi tasanne asetellaan näkymässä kamera-alueen alareunaan ja sille annetaan leveyttä täyttämään kamera-alueen leveys. Pelihahmon voi jättää keskelle kamera-aluetta. Pelihahmolle luodaan vielä uusi C# Movement -niminen skriptitiedosto hahmo-ohjainta varten.

Näillä valmisteluilla on mahdollista alkaa rakentaa hahmo-ohjainta Movement-tiedostoon ja testata ominaisuuksia näkymässä. Kuitenkin ennen liikkeen toteutusta rakennetaan ensin törmäysjärjestelmä, jotta pelihahmo pystyy

seisomaan tasanteen päällä. Tätä varten luodaan uusi C#-skripti nimeltä RaycastMoveDirection, joka ei peri MonoBehaviour-luokkaa. Tällä luokalla on DoRaycast-metodi, joka ottaa vastaan parametreina pelihahmon sijainnin ja toivotun liikemäärän ja palauttaa suurimman liikemäärän ennen törmäystä, joka on kuitenkin pienempi tai yhtäsuuri kuin toivottu liikemäärä.

Luokkaa alustettaessa sille annetaan kahden pisteen sijainnit, joista säteet tulee lähettää. Tämän lisäksi on määritettävä säteiden suunta, tasomaski ja offset, eli siirros pisteiden sijainnille, kuten on esitetty esimerkkikoodissa 1. Pisteiden sijainti voidaan välittää suoraan Box Collider 2D -komponentista, mutta säteiden etäisyyksissä saattaa esiintyä pieniä virheitä. Pienen testailun tuloksena virheellisistä etäisyyksistä pääsee eroon lähettämällä säteet hieman osumalaatikon sisäpuolelta. Tämä siirros on määriteltävissä parallel- ja perpendicular-muuttujilla, jotka vastaavat yhdensuuntaista ja kohtisuoraa suuntaa osumalaatikon sivuun nähden. addLength-muuttuja kompensoi siirroksen pituutta lähetettävän säteen pituuteen, sillä muuten säde olisi siirroksen verran lyhyempi.

```
public RaycastMoveDirection(Vector2[] points, Vector2 dir, LayerMask
mask, Vector2 parallel, Vector2 perpendicular)
{
     this.raycastDirection = dir;
     this.offsetPoints = new Vector2[] { points[0] + parallel + perpen-
dicular , points[1] - parallel + perpendicular };
     this.addLength = perpendicular.magnitude;
     this.layerMask = mask;
}
```

Esimerkkikoodi 1. RaycastMoveDirection-luokan alustusmetodi, jossa esillä jäsenmuuttujat.

DoRaycast-metodi käyttää sisällään Raycast-apumetodia, jolla saadaan visualisoitua säteet näkymässä vianetsintää varten. Raycast-apumetodi ottaa esimerkkikoodin 2 tapaan vastaan samat parametrit kuin Unityn Physics2D-luokan Raycast-metodi, mutta käyttää ensin Debug-luokan DrawLine-metodia viivojen piirtämiseen samoilla arvoilla. Lopuksi apumetodi palauttaa oikean RaycastHit2D-datatyypin.

```
private RaycastHit2D Raycast(Vector2 start, Vector2 dir, float len ,
LayerMask mask)
{
     Debug.DrawLine(start, start + dir * len, Color.blue);
     return Physics2D.Raycast(start, dir, len, mask);
}
```

Esimerkkikoodi 2. Apumetodi säteiden piirtämiseen.

DoRaycast-metodi hoitaa säteiden lähettämisen kummastakin pisteestä ja niiden törmätessä mittaa törmäyksen etäisyyden. Sädettä lähetettäessä sen pituuteen lisätään siirros, ja säteen törmätessä törmäysetäisyydestä vähennetään sama siirros, jotta jäljelle jää oikea etäisyys osumalaatikon reunasta törmättävään objektiin. Tässä on kuitenkin uusi ongelmatilanne, jossa säteen etäisyys ei välttämättä pidä paikkansa törmäysetäisyyden ollessa todella lyhyt. Siksi esimerkkikoodin 3 cleanDistance-muuttujaa käytetään varmistamaan, että etäisyys ei ole negatiivinen. Lopuksi tallennetaan aina minDistance-muuttujaan lyhin törmäysetäisyys kummaltakin säteeltä ja sen jälkeen metodi palauttaa minDistance-muuttujan arvon. MinDistance-muuttujan arvo on aluksi aina tavoiteliikemäärä, joka on myös palautettava arvo, mikäli törmäyksiä ei tapahdu.

```
public float DoRaycast(Vector2 origin, float distance)
{
     float minDistance = distance;
     foreach (var offset in offsetPoints)
     {
          RaycastHit2D hit = Raycast(origin + offset, raycastDirection,
distance + addLength, layerMask);
          if (hit.collider != null)
          {
               var cleanDistance = hit.distance - addLength;

               if (cleanDistance < 0f)
                    cleanDistance = 0f;

               minDistance = Mathf.Min(minDistance, cleanDistance);
          }
     }
     return minDistance;
}
```

Esimerkkikoodi 3. DoRaycast-metodi, joka seuraa säteiden törmäystä ja palauttaa lyhimmän säteen pituuden.

Seuraavaksi lisätään RaycastMoveDirection Movement-skriptiin. Movement-skripti sisältää valmiiksi MonoBehaviour-luokan perinnästä Start- ja Update-metodit. RaycastMoveDirectionista tehdään neljä instanssia, yksi jokaiseen kulkusuuntaan. Jokainen näistä alustetaan Start-metodissa. Aluksi selvitetään pisteiden sijainnit suhteessa pelihahmon sijaintiin. Ne saadaan tässä tapauksessa, kun pisteitä on neljä, suoraan Box Collider 2D -komponentilta käyttämällä sen size-metodia. Osumalaatikon mitat jaetaan siis kahdella, jolloin saadaan oikean yläreunan pisteen koordinaatit. Loput koordinaatit ovat itseisarvoltaan samansuuruiset, mutta erisuuntaiset, joten voidaan tässä vaiheessa tallentaa tämän pisteen koordinaatit 2-ulotteiseen vektorimuuttujaan ja käyttää sitä invertoimalla sen koordinaatteja tarpeen mukaan.

Kaksi pistettä annetaan taulumuodossa. Tämän lisäksi alustuksessa annetaan luokalle liikkeen suunta, tasomaski, samansuuntainen siirros kerrottuna sen suunnalla ja kohtisuora siirros kerrottuna sen suunnalla. Esimerkiksi ylöspäin kohdistuvan liikkeen luokassa ensimmäisen pisteen x- ja y-koordinaatit ovat molemmat positiivisia, jolloin kyseessä on oikean yläreunan piste. Kuten esimerkkikoodissa 4, toisen pisteen koordinaateista x-koordinaatti on käänteinen, jolloin kyseessä on vasemman yläreunan piste. Tässä säteiden suunta on ylös ja siirrokset ovat vasemmalle ja alas. Vasen siirros invertoidaan toiselle pisteelle automaattisesti luokan sisällä.

```
Vector2 col = GetComponent<BoxCollider2D>().size / 2;

     moveUp = new RaycastMoveDirection(
          new Vector2[] { new Vector2(col.x, col.y), new Vector2(-
col.x, col.y) },
          Vector2.up, platformMask, Vector2.left * parallelInsetLength,
Vector2.down * perpendicularInsetLength);
```

Esimerkkikoodi 4. RaycastMoveDirection-luokan alustus Movement-skriptissä.

Movement-skriptiin lisätään FixedUpdate-metodi, jossa käsitellään pelihahmon liikuttaminen. FixedUpdate-metodia kutsutaan automaattisesti, mutta toisin kuin Update-metodia, FixedUpdate-metodia kutsutaan säännöllisesti riippumatta pelin ruudunpäivityksestä. Tästä syystä on yleistä, että sen sisällä käsitellään kaikki fysiikkasimulointiin liittyvät asiat. (17.)

Painovoimalle ja suurimmalle sallitulle nopeudelle luodaan muuttujat ja lisäksi luodaan nopeusvektori pitämään nopeus muistissa. Painovoima voidaan toteuttaa helposti vähentämällä nopeusvektorin y-akselin suuntaisesta nopeudesta painovoiman arvo kerrottuna ajanmuutoksella. FixedDeltaTime antaa ajanmuutoksen FixedUpdate-metodin kutsujen väliltä. Tässä vaiheessa ajanmuutosta käytetään vain siksi, että käyttäjän ei tarvitsisi syöttää todella pieniä arvoja painovoimalle. Esimerkkikoodissa 5 y-akselin suuntainen nopeus on rajattu vielä suurimpaan sallittuun nopeuteen, jotta pelihahmo ei kiihdy rajattomasti.

```
velocity.y += -gravity * Time.fixedDeltaTime;

if (velocity.y < -maxFallSpeed)
{
     velocity.y = -maxFallSpeed;
}
```

Esimerkkikoodi 5. Painovoiman ja suurimman sallitun putoamisnopeuden toteutus.

Pelihahmoa liikutetaan käyttäen Transform-luokan Translate-metodia, joka liikuttaa sellaisenaan Transform-luokan omaa peliobjektia suhteessa omaan aikaisempaan sijaintiin (18). Ennen nopeusvektorin syöttämistä metodille pitää kuitenkin varmistaa, että se ei ole törmäämässä mihinkään. Tämä onnistuu käyttämällä RaycastMoveDirection-luokkaa.

Ensin luodaan paikalliset muuttujat siirtymälle ja toivotulle siirtymälle. Sitten lasketaan toivottu siirtymä y-akselin suuntaisesti, jossa nopeus kerrotaan kymmenellä ja ajanmuutoksella. Esimerkkikoodissa 6 liukuluku 10 taas on välissä vain siksi, että se tekee hypyn ja painovoiman arvoista käyttäjäystävällisempiä. Toivottu siirtymä annetaan sitten RaycastMoveDirection-luokan DoRaycast-metodille, joka palauttaa lopullisen sallitun siirtymän. Mikäli lopullisen siirtymän ja toivotun siirtymän välillä on iso ero, pitää nopeus nollata, jotta hahmo ei jää törmätessä kiihtyvään tilaan.

```
Vector2 displacement = Vector2.zero;
Vector2 wantedDisplacement;

wantedDisplacement.y = velocity.y * 10f * Time.fixedDeltaTime;

if (velocity.y > 0f)
     displacement.y = moveUp.DoRaycast(transform.position, wanted-
Displacement.y);
else if (velocity.y < 0f)
     displacement.y = -moveDown.DoRaycast(transform.position, -wanted-
Displacement.y);

if (!Mathf.Approximately(displacement.y, wantedDisplacement.y))
     velocity.y = 0f;

transform.Translate(displacement);
```

Esimerkkikoodi 6. Pelihahmon siirtäminen pelimaailmassa nopeusvektorin y-akselin mukaan.

Tähän mennessä törmäysjärjestelmä on toteutettu vain y-akselin suuntaisen liikkeen osalta. X-akselin toteutus on kuitenkin hyvin samanlainen, ja se voi hyödyntää samoja nopeus-, siirtymä- ja toivottu siirtymä -vektoreita. X-akselin suuntainen nopeus on kuitenkin vielä kerrottava pelihahmon liikenopeudella, jota käyttäjä voi mielivaltaisesti säädellä, sekä ajanmuutoksella.

4.2 Liikkeen toteutus

Seuraavaksi tarkastellaan pelihahmon liikuttamista. Siihen käytetään Unityn Input-luokan GetAxisRaw-metodia, joka palauttaa 1 tai -1, riippuen käyttäjän antamasta syötteen suunnasta, tai 0, jos käyttäjä ei anna syötettä liikkumiseen ollenkaan. On myös olemassa GetAxis-metodi, joka sisältää itsessään interpoloitua kiihtymistä, mutta se ei ole käyttäjän määriteltävissä ja on siksi huonompi vaihtoehto.

Kiihtyminen voidaan toteuttaa itse käyttämällä Mathf-luokan Lerp-metodia, jolla voidaan interpoloida arvo kahden arvon väliltä käyttäen kolmatta arvoa. Liikkeen kiihtyvyyden interpoloinnissa voidaan käyttää ajastinta, joka alkaa laskea kulunutta aikaa siitä hetkestä, kun pelaaja on antanut komennon liikkua, ja kun ajastin saavuttaa käyttäjän määrittelemän maksimiarvon, pelihahmo saavuttaa silloin huippunopeuden. Esimerkiksi positiivisesti kiihtyvässä

liikkeessä ensimmäinen arvo on nopeusvektorin x-akselin suuntainen nopeus, toinen arvo on tavoitenopeus ja interpolointiarvo on kulunut aika jaettuna maksimiarvolla. Tavoitteena oli kuitenkin pystyä vaikuttamaan myös kiihtyvyyden tyyppiin, ja se voidaan toteuttaa käyttämällä Unityn AnimationCurve-luokkaa.

AnimationCurvella käyttäjä pystyy itse määrittelemään käyrän avulla kiihtyvyyden tyypin. Esimerkkikoodissa 7 AnimationCurven Evaluate-metodi palauttaa käyrän y-akselin arvon, kun metodille syötetään x-akselin arvo. Evaluate-metodille voidaan siis syöttää tulkinta-arvoksi kulunut aika jaettuna maksimiajalla ja tämä kokonaisuus puolestaan suoraan interpolaatioarvoksi.

```
elapsedDecelTime = 0f;
elapsedAccelTime += Time.fixedDeltaTime;
velocity.x = input * Mathf.Lerp(velocity.x * input, 1f, acceleration-
CurveAir.Evaluate(elapsedAccelTime / accelTime));
```

Esimerkkikoodi 7. Interpoloidun positiivisen kiihtymisen toteutus.

Liikkeen syötteen logiikka muuttuu hieman hankalammaksi, kun halutaan, että pelihahmo kiihtyy nopeammin vaihtaessa suuntaa ja että ilmassa ja maassa on eri pelituntuma. Suunnanvaihdossa on tarkkailtava sekä nopeuden suuntaa että syötteen suuntaa, koska silloin, kun pelaaja haluaa pelihahmon vaihtavan suuntaa, on syöte erisuuntaista pelihahmon nopeuden kanssa. Tässä projektissa suunnan vaihto toteutettiin siten, että pelihahmo kokee vahvempaa negatiivista kiihtymistä vaihtaessa suuntaa, jotta suunnanvaihto tapahtuisi nopeammin. Jos pelaajan syöte on erisuuntaista pelihahmon nopeuden kanssa ja nopeuden itseisarvo on suurempi kuin 10 prosenttia, silloin pelihahmo vaihtaa suuntaa. Muussa tapauksessa pelihahmo kokee positiivista kiihtymistä, mikäli syötettä annetaan.

Ohjauslogiikka on esitetty kokonaisena esimerkkikoodissa 8. Siinä verrataan käyttäjän antaman syötteen ja pelihahmon liikkeen suuntaa, jotta tiedetään, minne pelihahmoa on seuraavaksi liikutettava. Esimerkiksi jos pelaaja vaihtaa syötteen suuntaa, pitää pelihahmo saada ensin hidastumaan, ennen kuin se voi alkaa taas kiihtyä uuteen suuntaan. Pelihahmo on aina hidastuvassa tilassa, jos

pelaaja ei anna syötettä liikuttaakseen pelihahmoa. Samoin, jos pelaajan antama syöte on samansuuntaista kuin pelihahmon liike, voidaan pelihahmoa kiihdyttää vapaasti huippunopeuteen saakka.

```
float input = 0f;

input = Input.GetAxisRaw("Horizontal");
velocityDir = GetDir(velocity.x);

if (!separateAirControls && enableAirControls)
     movementType = MovementTypes.Ground;

switch (movementType)
{
     case MovementTypes.Ground:

          if (input != 0f)
          {
               if (input != velocityDir && Mathf.Abs(velocity.x) >
0.1f)
               {
                    // Vaihda suuntaa
               }
               else
               {
                    // Positiivinen kiihtyminen
               }
          }
          else
          {
               // Negatiivinen kiihtyminen
          }
          break;

     case MovementTypes.Air:

          if (!enableAirControls)
               break;

          if (input != 0f)
          {
               if (input != velocityDir && Mathf.Abs(velocity.x) >
0.1f)
               {
                    // Vaihda suuntaa
               }
               else
               {
                    // Positiivinen kiihtyminen
               }
```

```
            }
            else
            {
                 // Negatiivinen kiihtyminen
            }
            break;
}
```

Esimerkkikoodi 8. Ohjauslogiikan toteutus.

Ilma- ja maaohjauksen vaihtamiseen käytetään switch-case-ratkaisua. Ohjaus toteutetaan siis kahteen kertaan eri muuttujilla, jotka käyttäjä pystyy määrittelemään erikseen ilma- ja maaohjaukselle. Ilmaohjaus voidaan ottaa pois käytöstä kokonaan, pitää erillisenä tai käyttää maaohjausta ilmaohjauksen sijaan.

4.3 Hypyn ja maatarkistuksen toteutus

Maatarkistua varten tehdään erillinen RaycastCheckTouch-luokka, joka on hyvin samanlainen RaycastMoveDirection-luokan kanssa. Luokat eroavat kuitenkin toisistaan parametrien ja palautusarvojen osalta. RaycastCheckTouch-luokkaan määritellään lähetettävän säteen pituus jo alustusvaiheessa. Luokan DoRaycast-metodi vaatii parametreina pelihahmon sijainnin ja palauttaa säteiden törmätessä etäisyyksien arvot tauluna. Mikäli säde ei törmää niin metodi palauttaa etäisyytenä -1. Luokka alustetaan Movement-skriptin Start-metodissa. Tarkoitus on lähettää säde pelihahmon osumalaatikon alareunoista alas päin. Säteen pituus määritellään muuttujalla, jota käyttäjä pystyy muuttamaan myöhemmin.

Movement-skriptiin luodaan uusi esimerkkikoodin 9 mukainen GroundCheck-metodi, jonka tehtävänä on tulkita Raycast-CheckTouch-luokan DoRaycast-metodin säteen etäisyyksiä. Tavoitteena on saada selville, milloin pelihahmo on kosketuksissa maan kanssa. Samalla voidaan selvittää, milloin pelihahmo on tarpeeksi lähellä maata, jotta hyppykomento voidaan pitää muistissa. DoRaycast-metodin palauttamien säteiden pituutta verrataan groundCheckLength-muuttujaan, jonka arvoksi on määritelty suurin sallittu etäisyys, joka tulkitaan vielä kosketukseksi maan kanssa. Jos säteen pituus on

suurempi, silloin voidaan tulkita, että pelihahmo on lähellä maata. Jos säteen pituus taas on pienempi, mutta kuitenkin suurempi kuin 0, voidaan tulkita, että ollaan kosketuksissa maan kanssa.

```
private void GroundCheck()
{
     float[] groundDis = checkDown.DoRaycast(transform.position);
     leftGroundDis = groundDis[0];
     rightGroundDis = groundDis[1];

     if (leftGroundDis >= groundCheckLength || rightGroundDis >=
groundCheckLength)
     {
          nearGround = true;
     }
     else if (leftGroundDis >= 0f || rightGroundDis >= 0f)
     {
          nearGround = false;
          onGround = true;
     }
     else
     {
          nearGround = false;
          onGround = false;
     }
}
```

Esimerkkikoodi 9. Maatarkistuksen toteutus käyttäen RaycastMoveDirection-luokkaa.

Nyt, kun hahmo-ohjaimella on tieto siitä, milloin pelihahmo on ilmassa, maassa tai lähellä maata, voidaan toteuttaa hyppyominaisuus. Hyppyominaisuus tulee asettaa Movement-skriptin FixedUpdate-metodiin ennen pelihahmon liikuttamista, jotta hyppy voidaan lisätä nopeusvektoriin. Hypyn toteutuksen tavoitteena on hyppy, jonka korkeutta pelaaja voi hallita pitämällä hyppynäppäintä pohjassa pidempään. Lisäksi hyppylogiikan pitää ottaa huomioon ohjaustapojen asettaminen ilma- ja maaohjauksen välillä, pitää muistissa hyppykomento pelihahmon ollessa lähellä maata ja sallia hyppykomento hetki sen jälkeen, kun pelihahmo on jo poistunut tasanteen reunalta.

Esimerkkikoodin 10 hyppylogiikka selvittää ensin, onko pelihahmo ilmassa. Ilmassa ollessa hoidetaan ajastimien kasvattaminen, joiden avulla annetaan mahdollisuus hypätä vielä ilmassa sekä käyttää maaohjausta ilmassa. Maassa

ollessa alustetaan jälleen kaikki hyppyyn liittyvät muuttujat valmiiksi seuraavaa hyppyä varten.

```
GroundCheck();

if (!nearGround && !onGround)
{
     if (!jumped)
          lateJumpTimer += Time.fixedDeltaTime;

     lateGroundControlsTimer += Time.fixedDeltaTime;

     if (lateGroundControlsTimer > lateGroundControlsMaxTime)
     {
          movementType = MovementTypes.Air;
     }
     if (pressedJumpBtn && !jumped && lateJumpTimer < lateJumpMaxTime
&& velocity.y < 0f) // Late Jump
     {
          jumped = true;
          jumping = true;
          jumpHoldTimer = 0f;
          pressedJumpBtn = false;
     }
     else
          pressedJumpBtn = false;
}
else if (onGround)
{
     if (pressedJumpBtn) // Normal Jump
     {
          jumped = true;
          jumping = true;
          jumpHoldTimer = 0f;
          pressedJumpBtn = false;
     }
     else
     {
          jumped = false;
          lateJumpTimer = 0f;
          lateGroundControlsTimer = 0f;
          movementType = MovementTypes.Ground;
     }
}

if (jumping)
     jumpHoldTimer += Time.fixedDeltaTime;

if (releasedJumpBtn && jumpHoldTimer > jumpHoldMinTime || jumpHold-
Timer > jumpHoldMaxTime)
{
     jumping = false;
}
```

Esimerkkikoodi 10. Hyppylogiikan toteutus kokonaisuudessaan.

Hypyn toteutuksessa käytetään ajastimia myöhäisen hypyn ja myöhäisen maaohjauksen sallimiseen. Vaihtuvaa hyppykorkeutta hallitaan myös ajastimen kanssa. Pelihahmo nousee ylöspäin, kunnes pelaaja lakkaa antamasta syötettä hypylle tai jos ajastin on saavuttanut suurimman sallitun ajan antaa hyppysyötettä. Ajastimella on myös pienin sallittu arvo, jolla voidaan hallita matalinta sallittua hyppyä. Toisin sanoen, vaikka pelaaja lakkaisi heti antamasta hyppysyötettä, jatkaa pelihahmo nousua, kunnes ajastin on saavuttanut pienimmän sallitun arvon.

Hyppyfysiikka esimerkkikoodissa 11 on toteutettu jälleen interpoloimalla hyppyvoiman arvoa. Hyppyvoima alkaa tasaisesti vähetä siitä hetkestä, kun hyppykomento on annettu. Laskentakaavaan on myös lisätty siirros, jolla voi viivästyttää hyppyvoiman vähentymistä. Tässä voisi myös käyttää AnimationCurve-luokkaa, jolla saisi muokattua hyppytuntumaa vielä tarkemmin käyttämällä käyrää.

```
if (jumping)
{
     velocity.y = jumpForce * Mathf.Clamp01(1 + jumpDecLerpOffset -
jumpHoldTimer / jumpHoldMaxTime);
}
```

Esimerkkikoodi 11. Hyppyfysiikan toteutus interpoloimalla.

4.4 Testausympäristön rakentaminen

Kaikista kolmesta pelistä luotiin kopio osasta ensimmäistä kenttää hahmo-ohjaimen testausta varten. Kopiolla tässä tapauksessa tarkoitetaan samaa taustaa, hahmomallia sekä esteitä ja tasanteita, jotka sijaitsevat samassa kohdassa kuin alkuperäisessä pelissä. Tämä auttaa visualisoimaan eroja pelihahmon ohjauksessa, ja fyysiset esteet voivat toimia esimerkiksi mittausapuna.

Taustat kerättiin ruutukauppauksina alkuperäistä peliä pelatessa. Poikkeuksena oli Super Mario Bros, joka ei ollut saatavilla PC-alustalle. Sen kohdalla taustat kerättiin pelivideoista. Kullekin pelille luotiin oma Unity-näkymä ja taustat

aseteltiin näkymiinsä saumattomasti siten, että pelihahmon leveys oli aina 1 leveysyksikkö. Pelihahmon hahmomallit löytyivät lähes kaikki internetistä lyhyen etsinnän jälkeen. Ainoastaan Celesten hahmomalli oli väärän kokoinen, joten se oli tehtävä itse. Celesten hahmomalli oli onneksi tehty todella pienellä resoluutiolla, koska se oli vain 16 pikseliä korkea, joten sen jäljentäminen ei ollut vaikeaa eikä vaatinut erityisemmin taiteellisia taitoja.

Kun taustat ja hahmomalli on aseteltu paikalleen, voidaan lisätä niille Box Collider 2D -komponentit. Pelihahmolle riittää, että osumalaatikko on suunnilleen samankokoinen tai vähän pienempi kuin hahmomalli. Taustojen osalta jokainen este ja tasanne on ympäröitävä osumalaatikoilla. Tämän jälkeen pelihahmolla on mahdollista liikkua ja hyppiä tasanteilla sekä törmäillä seiniin.

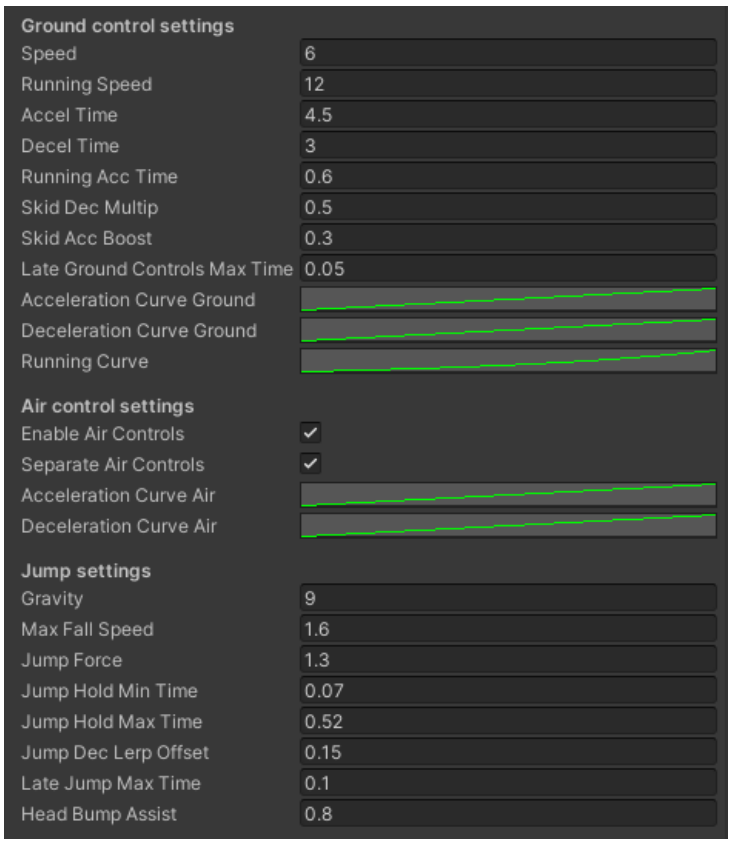
Viimeisenä osana pitää saada kamera liikkumaan, mikäli kenttä jatkuu ruudun ulkopuolelle. Tämä on yksinkertaisimmillaan toteutettavissa skriptillä, joka päivittää kameran x-akselin sijaintia samaksi kuin pelihahmon sijainti. Tämän projektin vertailupeleissä on melko monimutkaiset kamerajärjestelmät, ja ne olivat siksi projektin laajuuden ulkopuolella.

5 Projektin lopputulos

5.1 Pelituntuman jäljittely

Hahmo-ohjaimen yleiskäyttöisyyttä testattiin jäljentämällä kolmen eri 2D-tasohyppelypelin hahmo-ohjainta, joista kaikilla oli erilainen pelituntuma. Hahmo-ohjain on onnistunut toteutus, jos sillä voidaan jäljentää nopeasti ja tarkasti vertailtavien pelien pelituntumaa, ja vastaavasti epäonnistunut, jos jäljennys on todella hankalaa tai ei ollenkaan mahdollista. Vertailu koostuu kuitenkin kahdesta eri osasta, joista ensimmäinen on pelituntuman tulkitseminen alkuperäisestä pelistä ja toinen tulkitun pelituntuman jäljentäminen. Tulosten analysoinnissa on otettu huomioon tulkinnan vaikeuden vaikutus pelituntuman jäljentämiseen.

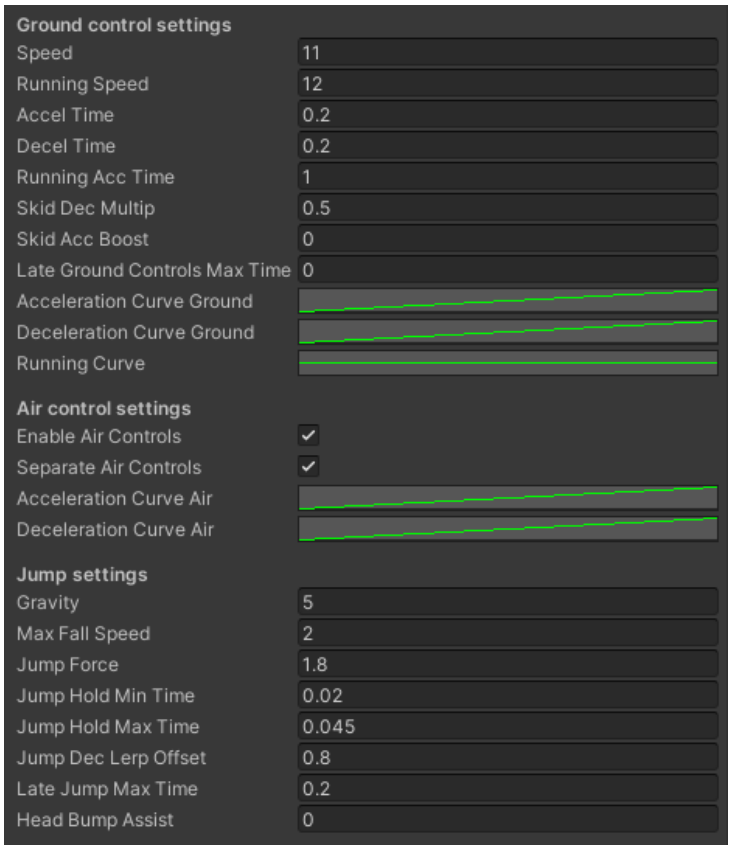
Ensimmäinen jäljitelty peli oli Super Mario Bros. Hahmo-ohjaimella oli mahdollista jäljitellä kaikki liikkumiseen ja hyppäämiseen liittyvät ominaisuudet. Näihin kuuluivat erillinen liikkeen pelituntuma ilmassa ja maassa, vaihtuva hyppykorkeus ja kiihtyvä liike. Liikkuminen maassa ja ilmassa oli todella raskasta, ja samanlaisen painon tunnun sai luotua lineaarisella kiihtyvyydellä. Ilma-ohjauksessa negatiivinen kiihtyminen puolitettiin siitä, mitä se oli maassa. Pelihahmon hyppy oli leijaileva, mutta tarkka, ja senkin jäljentäminen osui lähelle pitämällä painovoima korkealla ja samanaikaisesti hyppyvoima matalana. Hyppykorkeutta pystyi säätämään antamalla lisää hyppyaikaa. Hyppyvoima tuntui pysyvän samana pitkään, ja vasta hieman ennen lakipistettä se alkoi hiipua. Tämän sai toteutettua myöhästyttämällä hyppyvoiman hiipumisen alkamista. Lopulliset arvot näkyvät kuvassa 6.



Kuva 6. Super Mario Bros -pelin jäljentämisessä käytetyt arvot hahmo-ohjaimessa. Käyrät ovat lineaarisesti nousevia välillä 0 ja 1. Deceleration Curve Air on poikkeuksena välillä 0 ja 0,5.

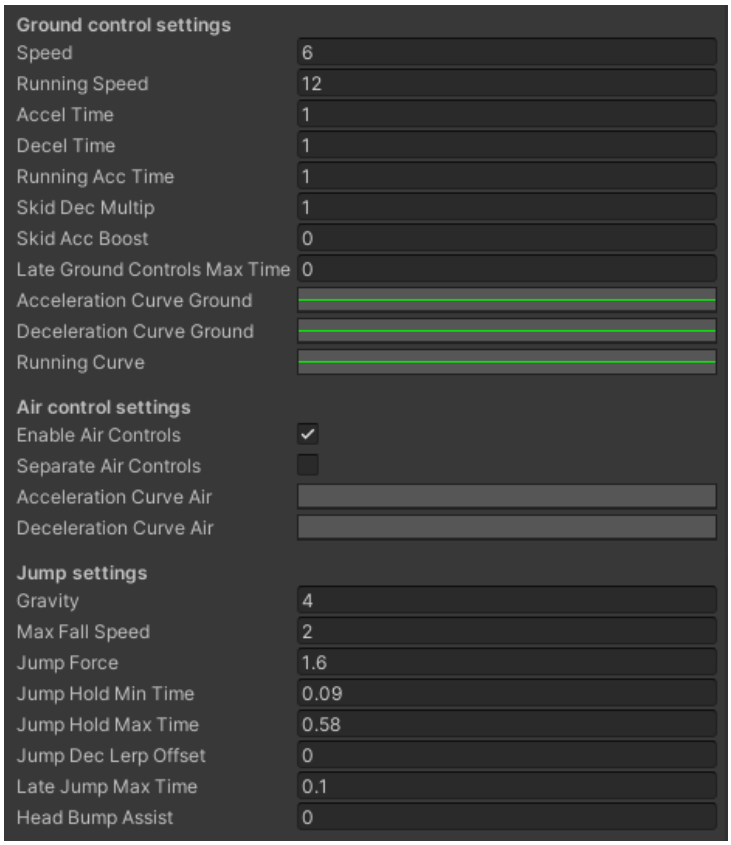
Seuraava peli oli Celeste, jonka pelituntuma oli puolestaan todella responsiivinen, nopea ja tarkka. Tässä testissä hahmo-ohjain ei suoriutunut ihan yhtä hyvin kuin aikaisemmassa. Celesten pelihahmo tuntuu saavan lisää nopeutta hypätessään sivusuuntaan paikaltaan. Ikään kuin hyppyvoima ei kohdistuisi suoraan ylöspäin. Tämä erottui melko selkeästi testatessa hyppyä sivusuunnassa juosten siten, että pelihahmon suuntaa vaihdetaan heti sen hypättyä. Celestessä pelihahmo ehtii lentää pidemmälle kuin jäljennöksessä, ennen kuin se kääntyy ympäri. Toinen huomattavasti isompi ongelma oli hypyn kanssa. Celestessä matala hyppy pysähtyy todella nopeasti, kun syötettä lakataan antamasta. Jäljennöksessä nopeusvektoriin jää nopeutta, jonka takia hahmo jatkaa matkaa vielä ylöspäin hypyn jälkeen.

Tämä on kuitenkin melko helposti korjattavissa lisäämällä hahmo-ohjaimeen ominaisuus korvata nopeusvektorin y-akselin arvo silloin, kun hyppysyötettä lakataan antamasta eikä ajastin ole vielä saavuttanut suurinta sallittua hyppyaikaa. Muilta osin pelituntuma tuntui hyvin samanlaiselta. Celestessä pelihahmolla on animoitu tukka, joka on simuloitu seuraamaan pelihahmon liikettä. Se tuo paljon sulavuutta peliin, minkä merkitystä hahmo-ohjaimen kanssa ei ollut mahdollista testata. Kuvasta 7 nähdään arvot, johon testissä päädyttiin.



Kuva 7. Celeste-pelin jäljentämisessä käytetyt arvot hahmo-ohjaimessa. Maaohjauksen käyrät ovat lineaarisesti nousevat välillä 0 ja 1. Ilmaohjauksen käyrät ovat lineaarisesti nousevat välillä 0 ja 0,9.

Viimeinen peli oli Hollow Knight, joka osoittautui vaikeimmaksi peliksi jäljitellä. Liike tuntuu välittömältä ja todella nopealta. Vaikka pelihahmon huippunopeus liikkeessä on melko pieni, se pystyy vaihtamaan suuntaa todella nopeasti. Projektin hahmo-ohjaimella ei suunnan vaihto onnistunut yhtä nopeasti ilman, että hahmon liikenopeutta nostettiin, vaikka kiihtyminen oli poistettu kokonaan käytöstä. Saattaa olla, että alkuperäisessä pelissä pelihahmon liikenopeus on hetken aikaa suurempi suunnanvaihdon jälkeen, jolloin pelihahmo ehtii kulkea pidemmän matkan toistuvien suunnanvaihtojen välissä. Hypyssä oli sama ongelma kuin Celestessä: hypyssä kertyvä nopeus nopeusvektorille y-akselin suunnassa pitäisi saada pysäytettyä, jotta pelihahmo lakkaisi nousemasta, kun hyppysyötettä ei enää anneta. Hahmo-ohjaimen arvot näkyvissä kuvassa 8.



Kuva 8. Hollow Knight -pelin jäljentämisessä käytetyt arvot. Animaatiokäyrät ovat vaakasuoria arvolla 1.

Kaiken kaikkiaan testausprosessi onnistui yllättävän hyvin käyttämällä metronomia pelituntuman mittaamisessa ja sen jäljittelemisessä. Celeste ja Hollow Knight olivat lopulta odotettua lähempänä liikkeen kiihtyvyyden osalta, ja parempi vaihtoehto olisi ollut korvata toinen peleistä pelillä, jossa liike olisi ollut vähän raskaampaa. Myös liike, jossa positiivinen ja negatiivinen kiihtyminen ovat hyvin erilaisia, olisi tehnyt testauksesta monipuolisempaa.

5.2 Yleiskäyttöinen hahmo-ohjain

Yleiskäyttöisen hahmo-ohjaimen tarkoitus oli toimia pohjana 2D-tasohyppelypelien kehityksessä ja pelituntuman testailussa sekä suunnittelussa. Yleiskäyttöinen hahmo-ohjain on nopea ottaa käyttöön, ja sillä on helppoa alkaa säätää liikkeen ja hypyn ominaisuuksia. Yleiskäyttöisessä hahmo-ohjaimessa on kuitenkin myös rajoitteita.

Liike on nopeasti ja monipuolisesti muokattavissa säätämällä kiihtymisaikoja ja kiihtymiskäyriä. Liike on myös erikseen säädettävissä ilmaohjaukselle. Hahmo-ohjain ei kuitenkaan tue suoraan ilma- ja maaohjauksen lisäksi muita ohjaustiloja, kuten omia ohjaustiloja erityyppisille maastoille. Eri ohjaustilojen lisääminen onnistuisi kohtalaisen helposti lisäämällä switch-case-valintaan controlTypes-luetteloon uusi ohjaustila. Ohjaustilan valitsemiseen pitäisi kuitenkin rakentaa oma logiikka, joka tunnustelisi erityyppisiä maastoja pelihahmon alla. Yleiskäyttöisen hahmo-ohjaimen tarjoama liike ei myöskään ole dynaamista siten, että sitä voisi muokata kesken pelin. Erilaiset juoksu- ja syöksyominaisuudet vaatisivat liikkeen muuttumista näppäintä painaessa. Myös hetkelliset suuremmat nopeudet, kuten Hollow Knightissa ympäri kääntyessä, on vaikea toteuttaa.

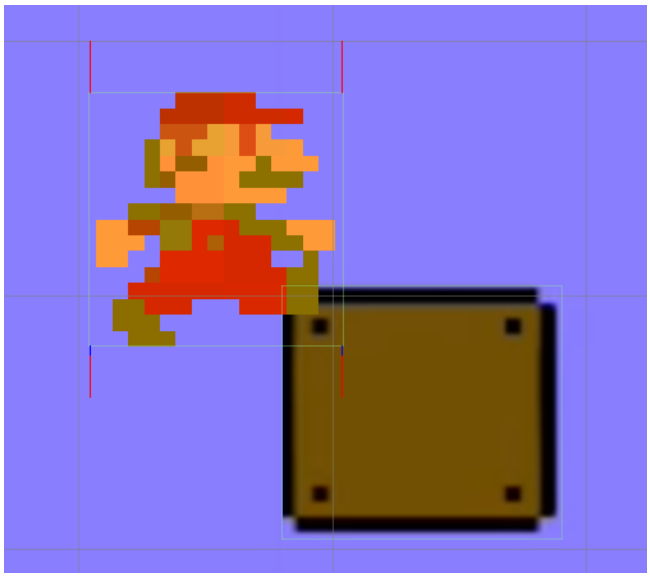
Hypyn toteutuksessa isona rajoitteena oli nopeusvektorin y-akselin suuntaisen nopeuden hallinnan puute. Matalissa hypyissä pelihahmoa ei ole mahdollista saada pysähtymään tarpeeksi nopeasti, jotta matalat hypyt voisivat olla tarkkoja. Tämän voi kuitenkin korjata helposti esimerkkikoodin 12 mukaan. JumpBrake-muuttujalla voidaan säätää hypyn pysähtymisen terävyyttä. Yleiskäyttöisessä hahmo-ohjaimessa hypyn suunta on myös aina ylöspäin. Esimerkiksi Celestessä tuntuu, että osa hyppyvoimasta kohdistuu myös hyppysuuntaan, ja tämän toteuttaminen vaatisi myös muutoksia hahmo-ohjaimeen.

```
if (releasedJumpBtn && jumpHoldTimer > jumpHoldMinTime || jumpHold-
Timer > jumpHoldMaxTime)
{
     if (releasedJumpBtn)
     {
          if (velocity.y > jumpBrake)
          {
               velocity.y = jumpBrake;
          }
     }
     jumping = false;
}
```

Esimerkkikoodi 12. Nopeusvektorin y-akselin nopeuden katkaisu jumpBrake-muuttujalla. Tämä mahdollistaa terävän pysähdyksen matalissa hypyissä.

Yleiskäyttöisen hahmo-ohjaimen törmäysjärjestelmä on hyvin yksinkertainen, mutta toimiva erilaisten pelituntumien testailussa. Se ei kuitenkaan sovellu sellaisenaan hahmo-ohjaimen laajentamiseen. Jotta se toimisi osumalaatikossa, pitää olla neljä pistettä ja lähetettäviä säteitä on oltava kahdeksan, eikä niiden suuntaa voi muuttaa. Törmäysjärjestelmä ei myöskään tunnista törmäyksiä, joissa törmättävä objekti mahtuu säteiden väliin, eikä osaa käsitellä portaita tai mäkiä.

Nykyisessä törmäysjärjestelmässä esiintyi kaksi virhettä. Kumpikin ovat melko harvinaisia, mutta ne selkeästi häiritsevät pelituntumaa. Ensimmäinen virhe tapahtuu, kun pelihahmo putoaa diagonaalisesti alaspäin ja törmää suoraan tasanteen kulmaan, kuten kuvattuna kuvassa 9. Silloin säteet eivät voi osua tasanteeseen ennen törmäystä eikä hahmo-ohjain osaa pysäyttää liikettä ajoissa. Lopputuloksena pelihahmo uppoaa tasanteen kulmaan sisään eikä pysty liikkumaan kohti tasanteen keskustaa. Pelihahmon voi kuitenkin vapauttaa tilanteesta hyppäämällä tai liikkumalla poispäin tasanteen keskustasta.



Kuva 9. Pelihahmo on joutunut tasanteen sisään törmättyään tasanteen kulmaan diagonaalisesti.

Toinen virhe liittyy nopeuden käsittelyyn törmätessä. Nykyisellä ratkaisulla käytetään Mathf-luokan Approximately-metodia vertaamaan lopullisen siirtymän ja toivotun siirtymän eroa. Jos metodin mukaan eroa ei ole, nopeudelle ei tehdä mitään, mutta jos ero löytyy, silloin nopeus nollataan. Tämä nollaa kertyneen nopeuden törmätessä, jotta pelihahmo aloittaa kiihtymisen alusta lähtiessään uudelleen liikkeelle. Kuitenkin harvinaisessa tapauksessa nopeus jää nollaamatta, vaikka seinään törmätään. Toisin sanoen, jos pelihahmo törmää matalaan esteeseen ja pelaaja antaa hyppykomennon pitäen edelleen liikenäppäintä painettuna, säilyy pelihahmon nopeus törmäyksen jälkeen.

Yleiskäyttöinen hahmo-ohjain soveltuu hyvin erilaisten pelituntumien ja kenttien testailuun. Hahmo-ohjaimen lisääminen uuteen projektiin on helppoa ja kenttien kokoaminen osumalaatikoista nopeaa. Isoin rajoite hahmo-ohjaimessa on kuitenkin törmäysjärjestelmä. Jotta hahmo-ohjainta voitaisiin käyttää pohjana pelinkehityksessä, pitäisi käyttää toista törmäysjärjestelmää.

5.3 Hahmo-ohjainten tulevaisuus

Modernit pelimoottorit ovat tuoneet pelikehitykseen niin paljon erilaisia työkaluja, että enää ongelma ei tunnu olevan se, onko jokin ominaisuus mahdollista toteuttaa, vaan se, miten se on helpoiten ja tehokkaimmin toteutettavissa. Sama koskee myös hahmo-ohjaimen kehittämistä 2D-tasohyppelypeleihin.

Unity tarjoaa valmiin fysiikkamoottorin, jolla on mahdollista luoda todella helposti ja yksinkertaisesti realistinen hahmo-ohjain 2D-tasohyppelypeleihin. Toisena ääripäänä kehittäjät voivat vapaasti luoda oman laattapohjaisen törmäysjärjestelmän Unity-peleihin, mikä on huomattavasti työläämpää ja aikaavievää. Jos näiden kahden ääripään väliltä löytyisi pohja hahmo-ohjaimelle, se nopeuttaisi paljon 2D-tasohyppelypelien kehittämistä.

Unityssa on jo tälläkin hetkellä oma tilemap-ratkaisu, jolla voidaan jakaa näkymässä ruutuihin ja ruudut puolestaan täyttää erilaisilla laatoilla. Laatoille on

saatavilla Tilemap Collider 2D -komponentti, joka antaa laatoille valmiiksi osumalaatikon ja mahdollistaa myös erilaisten fysiikkamateriaalien käytön. Fysiikkamateriaaleilla voidaan mm. muuttaa kitkaa pelihahmon ja laatan välillä. (19.) Fysiikkamateriaalien käyttö kuitenkin vaatii jälleen Unityn oman fysiikkamoottorin käyttöä, joka simuloi myös hahmon liikkeen rajoittaen kehittäjän hallintaa pelituntumaan.

Unitya kuitenkin kehitetään jatkuvasti käyttäjien ja alan tarpeiden mukaan. Eri osa-alueita pelinkehityksessä pyritään automatisoimaan ja tekemään käyttäjäystävällisemmäksi. Tällä hetkellä Unityn etenemissuunnitelmassa (20) ei kuitenkaan ole suunnitteilla kehitystä 2D-tasohyppelyiden hahmo-ohjaimeen tai uuteen törmäysjärjestelmään.

6 Yhteenveto

Insinöörityön tavoitteena oli syventyä hahmo-ohjaimen perusominaisuuksiin analysoimalla hahmo-ohjaimen kehitystä vuosien varrella ja hyödyntää opittua yleiskäyttöisen hahmo-ohjaimen kehityksessä. Työn alussa tutustuttiin 2D-tasohyppelypelien historiaan ja hahmo-ohjaimen perusominaisuuksiin, joiden avulla määriteltiin tavoitteet yleiskäyttöiselle hahmo-ohjaimelle. Tämän lisäksi määriteltiin testausmenetelmä testaamaan hahmo-ohjaimen mukautuvuutta erilaisiin pelituntumiin.

Tavoitteiden määrittelyn jälkeen toteutettiin yleiskäyttöinen hahmo-ohjain käyttämällä Unity-pelimoottoria ja sen muutamaa valmiskomponenttia. Kehitysprosessista dokumentoitiin ja tarkasteltiin tärkeimpiä ominaisuuksia ja niiden toteutustapoja käyttäen koodiesimerkkejä. Lopullinen ja kokonainen lähdekoodi hahmo-ohjaimen toteutukselle on työn liitteinä 1, 2 ja 3.

Yleiskäyttöisen hahmo-ohjaimen testauksen jälkeen tarkasteltiin onnistumisia ja ongelmia. Suurimpina ongelmista olivat rajoitteinen törmäysjärjestelmä ja suunnitteluvirhe hypyn toteutuksessa, joista kummallekin esitettiin kehitysehdotuksia. Yleiskäyttöinen hahmo-ohjain todettiin riittäväksi erilaisten pelituntumien testailuun ja suunnitteluun, mutta rajoitteisen törmäysjärjestelmänsä takia epäsoveltuvaksi käyttöön hahmo-ohjaimen pohjana pelikehityksessä. Jatkuvasti kehittyvät pelimoottoriominaisuudet saattavat kuitenkin tuoda tähän helpotusta tulevaisuudessa.

Lähteet

1 Thomasson, Michael. 2019. The foundation of all platformers. Old School Gamer Magazine, 07/2019, s. 11–12.

2 The player's guide to climbing games. Electronic Games, 01/1983, s. 50–51.

3 Arcade Hall of Fame: Donkey Kong (Nintendo). 2017. Verkkoaineisto. Retrorefurbs. <https://www.retrorefurbs.com/arcade-hall-of-fame-donkey-kong-nintendo/>. Luettu 17.11.2021.

4 NES-peliohjain. Wikimedia Commons. <https://upload.wikimedia.org/wikipedia/commons/b/b5/Nintendo-Entertainment-System-NES-Controller-FL.jpg>

5 Dahl, Gustav & Kraus, Martin. 2015. Measuring How Game Feel Is Influenced by the Player Avatar's Acceleration and Deceleration. Academic-MindTrek '15: Proceedings of the 19th International Academic Mindtrek, s. 41–46.

6 The Making of Toto Temple Deluxe: Platforming Part 2. 2014. Verkkoaineisto. Juicybeast. <http://juicybeast.com/2014/02/24/the-making-of-toto-temple-deluxe-platforming-part-2/> Luettu 20.11.2021.

7 The Making of Toto Temple Deluxe: Platforming Part 1. 2014. Verkkoaineisto. Juicybeast. <http://juicybeast.com/2014/02/12/the-making-of-toto-temple-deluxe-platforming-part-1/> Luettu 20.11.2021.

8 Pignole, Yoann. 2014. Platformer controls: how to avoid limpness and rigidity feelings. Verkkoaineisto. Game Developer. <https://www.gamedeveloper.com/design/platformer-controls-how-to-avoid-limpness-and-rigidity-feelings>. 3.1.2014. Luettu 7.9.2021.

9 Camera. 2021. Unity Manual. Unity Documentation.

10 Physics2D. Raycast. 2021. Scripting API. Unity Documentation.

11 Monteiro, Rodrigo. 2012. The guide to implementing 2D platformers. Verkkoaineisto. Higher-Order Fun. <http://higherorder-fun.com/blog/2012/05/20/the-guide-to-implementing-2d-platformers/>. 20.5.2012. Luettu 7.9.2021.

12 Iz-Tavarez, Gregg & Chang, Dan. 2003. Programming M.C. Kids. Verkkoaineisto. Games. Greggman. <https://games.greggman.com/game/programming_m_c__kids/>. 12.6.2003. Luettu 8.10.2021.

13 Super Mario Bros. Verkkoaineisto. Wikipedia. https://en.wikipedia.org/wiki/Super_Mario_Bros.#/media/File:NES_Super_Mario_Bros.png. Luettu 17.11.2021.

14 Celeste. Verkkoaineisto. Matt Makes Games. <http://www.celestegame.com> Luettu 17.11.2021.

15 Hollow Knight. Verkkoaineisto. Team Cherry. <https://www.hollow-knight.com/> Luettu 17.11.2021.

16 Jonasson, Martin & Purho, Petri. 2012. Juice it or lose it. Verkkoaineisto. Youtube. <https://www.youtube.com/watch?v=Fy0aCDmgnxg>. Luettu 15.10.2021.

17 MonoBehaviour. FixedUpdate(). 2021. Scripting API. Unity Documentation.

18 Transform. Translate. 2021. Scripting API. Unity Documentation.

19 Tilemap Collider 2D. 2021. Unity Manual. Unity Documentation.

20 Unity Platform Roadmap. Verkkoaineisto. Unity Software, Inc. <https://unity.com/roadmap/unity-platform>. Luettu 7.11.2021.

Movement-skripti

```
using UnityEngine;

public class Movement : MonoBehaviour
{
     #region Inspector

     [Header("Ground control settings")]

     [SerializeField]
     private float
          speed = 6f;

     [SerializeField]
     private float
          runningSpeed = 12,
          accelTime = 0.5f,
          decelTime = 0.5f,
          runningAccTime = 0.5f,
          skidDecMultip = 0.25f,
          skidAccBoost = 0.3f,
          lateGroundControlsMaxTime = 0f;

     [SerializeField]
     private AnimationCurve
          accelerationCurveGround = null,
          decelerationCurveGround = null,
          runningCurve;


     [Header("Air control settings")]

     [SerializeField]
     private bool
          enableAirControls;

     [SerializeField]
     private bool
          separateAirControls;

     [SerializeField]
     private AnimationCurve
          accelerationCurveAir = null,
          decelerationCurveAir = null;


     [Header("Jump settings")]

     [SerializeField]
     private float
          gravity = 9f;
```

```
[SerializeField]
private float
     maxFallSpeed = 2f,
     jumpForce = 1.3f,
     jumpHoldMinTime = 0.08f,
     jumpHoldMaxTime = 0.53f,

     jumpDecLerpOffset = 0.15f,
     lateJumpMaxTime = 0.1f,
     headBumpAssist = 0.8f;


[Header("Collision settings")]

[SerializeField]
private LayerMask
     platformMask = 0;

[SerializeField]
private float
     nearGroundLength = 0.2f,
     groundCheckLength = 0.05f,
     parallelInsetLength = 0.005f,
     perpendicularInsetLength = 0.005f;

#endregion

private enum MovementTypes
{
     Ground = 0,
     Air = 1
}

private MovementTypes
     movementType = MovementTypes.Ground;

private float
     elapsedAccelTime = 0f,
     elapsedDecelTime = 0f,
     runningTime = 0f,
     leftGroundDis = 0f,
     rightGroundDis = 0f,
     leftRoofDis = 0,
     rightRoofDis = 0f,
     headDodge,
     jumpHoldTimer = 0f,
     lateJumpTimer = 0f,
     lateGroundControlsTimer = 0f;

private int
     velocityDir = 0;

private bool
     jumping,
     jumped,
     pressedJumpBtn,
     releasedJumpBtn,
     skid,
     running,
     nearGround,
```

```
          onGround;

     private Vector2
          velocity;

     private RaycastMoveDirection
          moveUp,
          moveDown,
          moveLeft,
          moveRight;

     private RaycastCheckTouch
          checkDown,
          checkUp,
          checkRoofLeft,
          checkRoofRight;

     private void Start()
     {
          #region Raycast Setup

          Vector2 col = GetComponent<BoxCollider2D>().size / 2;

          moveUp = new RaycastMoveDirection(
               new Vector2[] { new Vector2(col.x, col.y), new Vector2(-
col.x, col.y) },
               Vector2.up, platformMask, Vector2.left * parallelIn-
setLength, Vector2.down * perpendicularInsetLength);

          moveDown = new RaycastMoveDirection(
               new Vector2[] { new Vector2(-col.x, -col.y), new Vec-
tor2(col.x, -col.y) },
               Vector2.down, platformMask, Vector2.right * parallelIn-
setLength, Vector2.up * perpendicularInsetLength);

          moveLeft = new RaycastMoveDirection(
               new Vector2[] { new Vector2(-col.x, col.y), new Vec-
tor2(-col.x, -col.y) },
               Vector2.left, platformMask, Vector2.down * parallelIn-
setLength, Vector2.right * perpendicularInsetLength);

          moveRight = new RaycastMoveDirection(
               new Vector2[] { new Vector2(col.x, -col.y), new Vec-
tor2(col.x, col.y) },
               Vector2.right, platformMask, Vector2.up * parallelIn-
setLength, Vector2.left * perpendicularInsetLength);

          checkDown = new RaycastCheckTouch(
               new Vector2[] { new Vector2(-col.x, -col.y), new Vec-
tor2(col.x, -col.y) },
               Vector2.down, platformMask, nearGroundLength, Vec-
tor2.right * parallelInsetLength, Vector2.up * perpendicu-
larInsetLength);

          checkUp = new RaycastCheckTouch(
               new Vector2[] { new Vector2(col.x, col.y), new Vector2(-
col.x, col.y) },
               Vector2.up, platformMask, nearGroundLength, Vector2.left
* parallelInsetLength, Vector2.down * perpendicularInsetLength);
```

```
          checkRoofLeft = new RaycastCheckTouch(
               new Vector2[] { new Vector2(col.x, col.y + 0.1f), new
Vector2(col.x, col.y + 0.1f) },
               Vector2.left, platformMask, col.x * 2, Vector2.zero,
Vector2.zero);

          checkRoofRight = new RaycastCheckTouch(
               new Vector2[] { new Vector2(-col.x, col.y + 0.1f), new
Vector2(-col.x, col.y + 0.1f) },
               Vector2.right, platformMask, col.x * 2, Vector2.zero,
Vector2.zero);

          #endregion
     }

     private void Update()
     {
          if (Input.GetKeyDown(KeyCode.Space))
          {
               pressedJumpBtn = true;
               releasedJumpBtn = false;
          }

          if (Input.GetKeyUp(KeyCode.Space))
          {
               releasedJumpBtn = true;
               pressedJumpBtn = false;
          }

          if (Input.GetKeyDown(KeyCode.UpArrow))
          {
               running = true;
          }

          if (Input.GetKeyUp(KeyCode.UpArrow))
          {
               running = false;
          }

          if (GetDir(velocity.x) == -1)
               gameObject.GetComponent<SpriteRenderer>().flipX = true;
          else if (GetDir(velocity.x) == 1)
               gameObject.GetComponent<SpriteRenderer>().flipX = false;
     }

     private void FixedUpdate()
     {
          #region GroundCheck and Jump

          GroundCheck();

          if (!nearGround && !onGround)
          {
               if (!jumped)
                    lateJumpTimer += Time.fixedDeltaTime;

               lateGroundControlsTimer += Time.fixedDeltaTime;

               if (lateGroundControlsTimer > lateGroundControlsMaxTime)
               {
```

```
                    movementType = MovementTypes.Air;
               }
               if (pressedJumpBtn && !jumped && lateJumpTimer <
lateJumpMaxTime && velocity.y < 0f) // Late Jump
               {
                    jumped = true;
                    jumping = true;
                    jumpHoldTimer = 0f;
                    pressedJumpBtn = false;
               }
               else
                    pressedJumpBtn = false;
          }
          else if (onGround)
          {
               if (pressedJumpBtn) // Normal Jump
               {
                    jumped = true;
                    jumping = true;
                    jumpHoldTimer = 0f;
                    pressedJumpBtn = false;
               }
               else
               {
                    jumped = false;
                    lateJumpTimer = 0f;
                    lateGroundControlsTimer = 0f;
                    movementType = MovementTypes.Ground;
               }
          }

          if (jumping)
               jumpHoldTimer += Time.fixedDeltaTime;

          if (releasedJumpBtn && jumpHoldTimer > jumpHoldMinTime ||
jumpHoldTimer > jumpHoldMaxTime)
          {
               jumping = false;
          }

          RoofCheck();
          HeadBump();

          #endregion

          #region Inputs

          float input;

          input = Input.GetAxisRaw("Horizontal");
          velocityDir = GetDir(velocity.x);

          if (!separateAirControls && enableAirControls)
               movementType = MovementTypes.Ground;

          switch (movementType)
          {
               case MovementTypes.Ground:

                    if (input != 0f)
```

```
                    {
                         if (input != velocityDir && Mathf.Abs(veloc-
ity.x) > 0.1f)
                         {
                              Debug.Log("Skid");
                              elapsedAccelTime = 0f;
                              elapsedDecelTime += Time.fixedDeltaTime;
                              velocity.x = Mathf.Lerp(velocity.x, 0, de-
celerationCurveGround.Evaluate(elapsedDecelTime / (decelTime * skid-
DecMultip)));
                              skid = true;
                              runningTime = 0f;
                         }
                         else
                         {
                              Debug.Log("Accelerate");
                              elapsedDecelTime = 0f;
                              elapsedAccelTime += Time.fixedDeltaTime;
                              if (skid)
                              {
                                   elapsedAccelTime = skidAccBoost;
                                   skid = false;
                              }
                              velocity.x = input * Mathf.Lerp(velocity.x
* input, 1f, accelerationCurveGround.Evaluate(elapsedAccelTime / ac-
celTime));
                         }
                    }
                    else
                    {
                         Debug.Log("Decelerate");
                         elapsedAccelTime = 0f;
                         elapsedDecelTime += Time.fixedDeltaTime;
                         velocity.x = Mathf.Lerp(velocity.x, 0f, decel-
erationCurveGround.Evaluate(elapsedDecelTime / decelTime));
                    }
                    break;

               case MovementTypes.Air:

                    if (!enableAirControls)
                         break;

                    if (input != 0f)
                    {
                         if (input != velocityDir && Mathf.Abs(veloc-
ity.x) > 0.1f)
                         {
                              Debug.Log("Skid");
                              elapsedAccelTime = 0f;
                              elapsedDecelTime += Time.fixedDeltaTime;
                              velocity.x = Mathf.Lerp(velocity.x, 0, de-
celerationCurveAir.Evaluate(elapsedDecelTime / (decelTime * 0.5f)));
                         }
                         else
                         {
                              Debug.Log("Accelerate");
                              elapsedDecelTime = 0f;
                              elapsedAccelTime += Time.fixedDeltaTime;
```

```
                              velocity.x = input * Mathf.Lerp(velocity.x
* input, 1f, accelerationCurveAir.Evaluate(elapsedAccelTime / accel-
Time));
                         }
                    }
                    else
                    {
                         Debug.Log("Decelerate");
                         elapsedAccelTime = 0f;
                         elapsedDecelTime += Time.fixedDeltaTime;
                         velocity.x = Mathf.Lerp(velocity.x, 0f, decel-
erationCurveAir.Evaluate(elapsedDecelTime / decelTime));
                    }
                    break;
          }

          if (running && onGround)
               runningTime += Time.fixedDeltaTime;
          else if (!running && onGround)
               runningTime -= Time.fixedDeltaTime * 1.5f;

          if (runningTime > runningAccTime)
               runningTime = runningAccTime;
          if (runningTime < 0f)
               runningTime = 0f;

          #endregion

          #region Displacement

          if (jumping)
          {
               velocity.y = jumpForce * Mathf.Clamp01(1 + jumpDecLer-
pOffset - jumpHoldTimer / jumpHoldMaxTime);
          }

          if (!jumping)
          {
               velocity.y += -gravity * Time.fixedDeltaTime;
          }

          if (velocity.y < -maxFallSpeed)
          {
               velocity.y = -maxFallSpeed;
          }

          Vector2 displacement = Vector2.zero;
          Vector2 wantedDisplacement;

          wantedDisplacement.x = velocity.x * Mathf.Lerp(speed, run-
ningSpeed, runningCurve.Evaluate(runningTime / runningAccTime)) *
Time.fixedDeltaTime;
          wantedDisplacement.y = velocity.y * 10f * Time.fixedDel-
taTime;

          if (velocity.y > 0f)
               displacement.y = moveUp.DoRaycast(transform.position,
wantedDisplacement.y);
          else if (velocity.y < 0f)
```

```
                displacement.y = -moveDown.DoRaycast(transform.position,
-wantedDisplacement.y);
          if (velocity.x < 0f)
                displacement.x = -moveLeft.DoRaycast(transform.position,
-wantedDisplacement.x);
          else if (velocity.x > 0f)
                displacement.x = moveRight.DoRaycast(transform.position,
wantedDisplacement.x);

          if (!Mathf.Approximately(displacement.x, wantedDisplace-
ment.x))
                velocity.x = 0f;

          if (!Mathf.Approximately(displacement.y, wantedDisplace-
ment.y))
                velocity.y = 0f;

          displacement.x += headDodge;

          if (headDodge !=0)
          {
                displacement.y = wantedDisplacement.y;
          }

          if (displacement.x == 0f)
          {
                elapsedAccelTime = 0f;
                runningTime = 0f;
          }

          transform.Translate(displacement);

          #endregion
     }

     private int GetDir(float value)
     {
          if (Mathf.Approximately(value, 0f))
                return 0;
          else if (value > 0f)
                return 1;
          else
                return -1;
     }

     private void GroundCheck()
     {
          float[] groundDis = checkDown.DoRaycast(transform.position);
          leftGroundDis = groundDis[0];
          rightGroundDis = groundDis[1];

          if (leftGroundDis >= groundCheckLength || rightGroundDis >=
groundCheckLength)
          {
                nearGround = true;
          }
          else if (leftGroundDis >= 0f || rightGroundDis >= 0f)
          {
                nearGround = false;
                onGround = true;
```

```
            }
            else
            {
                nearGround = false;
                onGround = false;
            }
        }

        private void RoofCheck()
        {
            float[] roofDis = checkUp.DoRaycast(transform.position);
            leftRoofDis = roofDis[1];
            rightRoofDis = roofDis[0];
            if (leftRoofDis > 0f && rightRoofDis > 0f)
                if (leftRoofDis < 0.5f && rightRoofDis < 0.5f)
                    jumping = false;
        }

        private void HeadBump()
        {
            float[] headSpace;
            float col = gameObject.GetComponent<BoxCollider2D>().size.x;
            if (leftRoofDis >= 0f && leftRoofDis < 0.5f)
            {
                headSpace = checkRoofLeft.DoRaycast(transform.position);
                if (headSpace[0] > headBumpAssist)
                    headDodge = (col - headSpace[0]) * 1.1f;
                else if (headSpace[0] > 0f)
                    jumping = false;

            }
            else if (rightRoofDis >= 0f && rightRoofDis < 0.5f)
            {
                headSpace = checkRoofRight.DoRaycast(transform.posi-
tion);
                if (headSpace[0] > headBumpAssist)
                    headDodge = -(col - headSpace[0]) * 1.1f;
                else if (headSpace[0] > 0f)
                    jumping = false;
            }
            else
                headDodge = 0f;
        }
}
```

RaycastMoveDirection-skripti

```
using UnityEngine;

public class RaycastMoveDirection
{
     private Vector2 raycastDirection;
     private Vector2[] offsetPoints;
     private LayerMask layerMask;
     private float addLength;

     public RaycastMoveDirection(Vector2[] points, Vector2 dir, Layer-
Mask mask, Vector2 parallel, Vector2 perpendicular)
     {
          this.raycastDirection = dir;
          this.offsetPoints = new Vector2[] { points[0] + parallel +
perpendicular , points[1] - parallel + perpendicular };
          this.addLength = perpendicular.magnitude;
          this.layerMask = mask;
     }

     public float DoRaycast(Vector2 origin, float distance)
     {
          float minDistance = distance;
          foreach (var offset in offsetPoints)
          {
               RaycastHit2D hit = Raycast(origin + offset, raycastDi-
rection, distance + addLength, layerMask);
               if (hit.collider != null)
               {
                    var cleanDistance = hit.distance - addLength;

                    if (cleanDistance < 0f)
                         cleanDistance = 0f;

                    minDistance = Mathf.Min(minDistance, cleanDis-
tance);
               }
          }
          return minDistance;
     }

     private RaycastHit2D Raycast(Vector2 start, Vector2 dir, float len
, LayerMask mask)
     {
          Debug.DrawLine(start, start + dir * len, Color.blue);
          return Physics2D.Raycast(start, dir, len, mask);
     }
}
```

RaycastCheckTouch-skripti

```
using System.Collections.Generic;
using UnityEngine;

public class RaycastCheckTouch
{
     private Vector2 raycastDirection;
     private Vector2[] offsetPoints;
     private LayerMask layerMask;
     private float raycastLength;
     private float perpendicular;

     public RaycastCheckTouch(Vector2[] points, Vector2 dir, LayerMask
mask, float checkLength, Vector2 parallel, Vector2 perpendicular)
     {
          this.raycastDirection = dir;
          this.offsetPoints = new Vector2[] { points[0] + parallel +
perpendicular, points[1] - parallel + perpendicular };
          this.raycastLength = perpendicular.magnitude + checkLength;
          this.perpendicular = perpendicular.magnitude;
          this.layerMask = mask;
     }

     public float[] DoRaycast(Vector2 origin)
     {
          List<float> distance = new List<float>(); ;
          foreach (var offset in offsetPoints)
          {
               RaycastHit2D hit = Raycast(origin + offset, raycastDi-
rection, raycastLength, layerMask);
               if (hit.collider != null)
               {
                    // Sometimes raycast returns slightly wrong dis-
tances. This prevents negative distances if the raycast is shorten
than perpendicular.
                    var cleanDistance = hit.distance - perpendicular;

                    if (cleanDistance < 0f)
                         cleanDistance = 0f;

                    distance.Add(cleanDistance);
               }
               else
                    distance.Add(-1f);
          }
          return distance.ToArray();
     }

     private RaycastHit2D Raycast(Vector2 start, Vector2 dir, float len
, LayerMask mask)
     {
          Debug.DrawLine(start, start + dir * len, Color.red);
          return Physics2D.Raycast(start, dir, len, mask);
     }
}
```