Olli Ketola

UNITYN 2D-FYSIIKKAMOOTTORI

Opinnäytetyö
Tietojenkäsittely

Marraskuu 2014

KUVAILULEHTI

MAMK University of Applied Sciences	**Opinnäytetyön päivämäärä** 06.11.2014
Tekijä(t) Olli Ketola	**Koulutusohjelma ja suuntautuminen** Tietojenkäsittely

Nimeke

Unityn 2D-fysiikkamoottori

Tiivistelmä

Opinnäytetyön aiheena on tutkia Unityn 2D-fysiikkaamoottorin ominaisuuksia. Teoriaosuuden alussa käydään läpi, mitä pelimoottoriohjelmistot ja fysiikkamoottorit ovat ja millaisia ominaisuuksia ne sisältävät. Teoriaosuuden lopuksi esitellään Unityn pelimoottoriohjelmistoa yleisesti, minkä jälkeen perehdytään tarkemmin Unityn 2D-fysiikkamoottoriin ja esitteellään sen sisältämät komponentit ja niiden toiminnot. Tarkoituksena ei ollut tehdä opasta Unityn peruskäyttöön, vaan antaa lukijalle ymmärrys millaisia ominaisuuksia Unity 2d-fysiikkamoottori sisältää ja mitä sillä voidaan käytännössä tehdä.

Käytännön osassa tutkitaan miten Unityn 2D-fysiikkamoottorin tarjoamia komponentteja voidaan soveltaa 2D-pelien teossa. Ominaisuuksia esitellään kahden pelidemon avulla, joiden tekemisessä pyrittiin hyödyntämään Unityn 2D-fysiikkakomponentteja. Molemmista aikaansaaduista demoista käännettiin PC:llä ja Windows Phone -laitteilla toimivat versiot.

Opinnäytetyön tuloksena syntyneissä pelidemoissa saatiin hyödynnettyjä miltei kaikkia Unityn tarjoamia fysiikkakomponentteja. Molemmat demot saatiin vaiheeseen, joka mahdollistaa niiden jatkokehityksen aina valmiiksi tuotteiksi asti. Valmiiden fysiikkamoottorien käyttö tarjoaa erittäin nopean tavan saada aikaan monimutkaisia toimintoja, joiden ohjelmoiminen alusta asti kestäisi huomattavasti kauemmin. Lisäksi valmiiden fysiikkamoottorien suosiminen takaa laadukkaan lopputuloksen.

Asiasanat (avainsanat)

peliohjelmointi, C#

Sivumäärä	**Kieli**	**URN**
37	Suomi	

Huomautus (huomautukset liitteistä)

Ohjaavan opettajan nimi	**Opinnäytetyön toimeksiantaja**
Jukka Selin	Mikkelin ammattikorkeakoulu

DESCRIPTION

MAMK University of Applied Sciences	**Date of the bachelor's thesis** 11 November 2014
Author(s) Olli Ketola	**Degree programme and option** Business Information Technology

Name of the bachelor's thesis

Unity's 2D physics engine

Abstract

The subject of this bachelor thesis was to examine the properties of Unity's 2D physics engine. The beginning of the theoretical part introduce game engines and physics engines in general and explained what kind of functionality this kind of programs include. The end of the theoretical part presented the Unity game engine's properties in general and after the that thesis focused on Unity's 2D physics engine and examined what kind of components and functions it had. The Purpose of this theoretical part was not to be a guide how to use the Unity's 2D physics but rather provide understanding of what kind of properties it had.

The practical part of this thesis examined how Unity's 2D physics engine could be used to make games. The properties of the physics engine were presented through two games which were made by using different 2D physics properties. Both of the game demos were made to work the with Windows phone mobile devices and a PC.

Almost all of the 2D physics components were used in the demos. Both of the demos also provided a very good frame for developing them to full games in the future. An integrated physics engine provided a fast and easy way to achieve complex functionality in games which would take a huge amount of time if made from beginning by on self. Using physics engines also gives one kind of guarantee of high-quality game physics.

Subject headings, (keywords)

game programming, C#

Pages	**Language**	**URN**
37	Finnish	

Remarks, notes on appendices

Tutor	**Bachelor's thesis assigned by**
Jukka Selin	Mikkeli University of Applied Sciences

SISÄLTÖ

1 JOHDANTO

Nykypäivänä suurin osa videopeleistä käyttää jonkinlaista fysiikkamoottoria, jonka tehtävänä on hoitaa laskutoimitukset, jotka mahdollistavat todellisuutta simuloivat tapahtumat peleissä. Tähän tarkoitukseen löytyy useita, sekä kaupallisia että ilmaisia ohjelmistoja. Fysiikkamoottorit eivät kuitenkaan ole itsenäisiä sovelluksia, vaan ne ovat usein liitetty suurempaan kokonaisuuden, jota kutsutaan pelimoottoriksi. Valmiiden fysiikkamoottorien tehtävänä on nopeuttaa ja helpottaa työtä. Mikäli projektin resurssit sen sallivat, voi fysiikkamoottorin toteuttaa myös alusta asti itse, joskin se vaatii hieman syvällisempää tietoutta fysiikasta ja matematiikasta.

Tässä opinnäytetyössä aihe on rajattu käsittelemään peliohjelmointia Unityn 2D-fysiikkamoottorin avulla, joka perustuu tunnettuun Box2D-fysiikkamoottoriin. Opinnäytetyössä päädyin käsittelemään juuri 2D-fysiikkaominaisuuksia, koska viime aikoina 2D-ohjelmointi on nostanut jälleen suosiotaan pääasiassa mobiililaitteiden pelikäytön yleistyessä. Lisäksi digitaalisten latauspalveluiden listauksia tarkasteltaessa voidaan todeta, että menestyvän pelin ei tarvitse olla graafisesti näyttävä, vaan 2D-graafikoilla toteutetuilla peleillä on yhtäläiset mahdollisuudet menestyä. Unityn valinta käytettäväksi pelimoottoriksi oli helppo, vaikka kyseessä on ilmainen ohjelmisto, se tarjoaa erittäin kilpailukykyiset ominaisuudet, etenkin kun kyseessä on 2D-ohjelmointi, jossa ei välttämättä tavoitella graafisesti realistisen näköistä tuotetta. Lisäksi Unity mahdollistaa valmiin projektin kääntämisen monelle eri laitteelle. Valintaan vaikutti myös oma aiempi kokemus Unityn käytöstä.

Opinnäytetyöni teoriaosuudessa käsittelen ensin pelimoottoreita ja fysiikkamoottoreita yleisesti. Teoriaosuuden lopuksi keskityn tarkemmin Unity-ohjelmistoon käymällä läpi sen perusominaisuuksia ja 2D-fysiikkaominaisuuksia hieman syvällisemmin.
Empiirisessä osassa tutkin, miten Unityn 2D-fysiikkamoottorin tarjoamia komponentteja voidaan hyödyntää peliohjelmoinnissa. Tätä tarkoitusta varten tein kaksi PC:llä ja Windows Phone -laitteilla toimivaa pelidemoa, joiden avulla havainnollistetaan fysiikkamoottorin ominaisuuksia. Tämän opinnäytetyön tarkoitus ei siis ole tehdä ohjeistusta Unityn peruskäyttöön, eikä paneutua pelintekoprosessin etenemiseen, vaan esitellä Unityn sisältämää 2D-fysiikkamoottoria.

2 PELIMOOTTORIT

Sanalle pelimoottori on vaikea antaa lyhyttä määritelmää, tosin kirjassaan Thorn vertaa (2011, 3–4) pelimoottoria auton moottorin. Ilman moottoria autoa ei voi liikkua, mutta toisaalta muiden osien puuttuessa pelkällä moottorillakaan ei tee mitään. Niin kuin auton moottori, myös pelimoottori koostuu pienemmistä komponenteista, jolla on kaikilla oma tehtävä. Pelimoottorin tapauksessa nämä komponentit ovat erilaisista ohjelmointikirjastoista, joiden avulla pelimoottori pystyy suorittamaan käyttäjän sille määrittämiä toimintoja. Pelimoottorien päätavoite on siis vähentää pelintekijöiden työmäärää hyödyntämällä valmiita komponentteja, joita yhdistelemällä varsinainen pelimoottoriohjelmisto syntyy. Esimerkkinä grafiikan piirtäminen tapahtuu aina samoja periaatteita hyödyntäen, joten sitä ei ole järkevää koodata jokaiseen pelimoottoriin uudestaan. Tähän tarkoitukseen on luotu ohjelmointirajapintoja, jotka sisältävät periaatteet grafiikan piirtämiseen. Sama pätee pelimoottorien fysiikoiden mallintamiseen, äänten toistamiseen ja tekoälyn toteuttamiseen. Perusperiaatteet pysyvät muuttumattomina.

Ennen peliyhtiöt päätyivät tekemään oman pelimoottorin, koska sen avulla ne julkaisivat omia tuotteitaan. Viime vuosina suurten pelitalojen rinnalle on tullut useita kilpailijoita, jotka eivät kehitä pelejä, vaan keskittyvät pelkästään pelimoottoreihin. Kilpailun koventuessa monet suuret pelitalot ovat joutuneet muuttamaan lisenssi käytäntöjään. Nykyään onkin tarjolla useita pelimoottoreita, joiden käytön voi aloittaa ilmaiseksi. Tarjolla olevien pelimoottorien skaala on niin laaja, että suurienkaan studioiden ei välttämättä ole edes järkevää ohjelmoida alusta alkaen omaa pelimoottoria, vaan hyödyntää tarjolla olevia. Joskus kuitenkin peli-idea saattaa olla niin ainutkertainen, että sitä on yksikertaisesti vaikea lähteä toteuttama valmiilla ratkaisulla. Tällöin valmista pelimoottoria jouduttaisiin muuttamaan niin paljon, että on helpompaa kehittää kokonaan oma. Usein pelimoottorit on suunniteltu jotain tiettyä pelingenreä varten, joka on tärkeää huomioida valittaessa pelimoottoria. Esimerkiksi pelimoottori, joka on alun perin suunniteltu ensimmäisestä persoonasta kuvattujen pelin tekoon, ei ole kovinkaan optimaalinen valinta avoimenmaailman verkkoroolipeliä tehdessä. (Gregory 2014, 12.)

2.1 Pelimoottorien historiaa

Ennen pelimoottoreita pelit olivat suljettua sovelluksia, joiden pelilogiikkaa oli tiukasti sidoksissa pelin muihin keskeisiin toimintoihin. Pelimoottorien historia juontaa juurensa vuonna 1993 ID Softwaren julkaisemasta Doom-pelistä, joka oli ensimmäinen pelimoottoria hyödyntävä tietokonepeli. Doom erosi aiemmista peleistä erityisesti arkkitehtuuriltaan, sillä pelin ydinkomponentit kuten grafiikan renderönti, törmäysten tunnistaminen ja ääni ominaisuudet, olivat erillään peligrafiikasta ja pelinlogiikkaan liittyvistä koodeista. Pelin ydintoimintojen eristäminen mahdollisti sen, että uutta peliä tehtäessä pysyttiin hyödyntämään vanhan pelin ydintoimintoja, eikä kaikkea tarvinnut tehdä alusta asti uudestaan. Doomin pelimoottori ei kuitenkaan ollut 3D-pelimoottori, vaan se hyödynsi Sprite-kuvia pelimaailman luomiseen. 1990-luvun lopulla aikana julkaistut Unreal, Half-life ja Quake 3, olivat ensimmäisiä pelejä, jotka sisälsivät laajasti kustomoitavat 3D-pelimoottorit. Samoihin aikoihin pelimoottorien lisenssien myynti pelitalojen käyttöön alkoi olla merkittävä tulonlähde pelimoottorienkehittäjille. (Gregory 2014, 11.)

2.2 Pelimoottorien keskeisiä ominaisuuksia

Ulospäin näkyvin pelimoottorien perusominaisuus on visuaalinen editori, joka on tärkeä osa kokonaisuutta. Editorin avulla päästään käsiksi pelimoottorien tarjoamiin ominaisuuksiin ja sen avulla liitettään pelin tuotu sisältö kokonaisuudeksi, jonka lopputuloksena on valmis peli. Editori sisältää yleensä tiedostorakenteen, jolla hallitaan peliin tuotavaa sisältöä. Editoria hyödyntävien pelimoottorien kanssa työskentely on nopeaa ja palkitsevaa, koska ruudulle saadaan suhteellisen pienellä määrällä luotua monimutkaisia toimintoja.

Fysiikka on hyvin keskeisessä osassa pelimoottoreissa, kun pelissä tavoitellaan todenmukaista lopputulosta. Usein hyvältäkään näyttävä peli ei tule menestymään, jos se ei ole todentuntuinen. Fysiikan perusperiaatteet ovat muuttumattomia, joten niiden uudelleen kirjoittaminen jokaista pelimoottoria varten ei ole järkevää. Tämän vuoksi on kehitetty erillisiä fysiikkamoottoreita, jotka ovat yleensä sisäänrakennettu lopullisiin pelimoottoreihin. Fysiikkamoottoreiden tehtävä on vastata laskutoimituksista, joita vaaditaan tunnistettaessa kappaleiden törmäyksiä ja simuloitaessa liikettä. Tun-

nettuja fysiikkamoottoreita ovat Box2D, Nvidian PhysX ja Havok. (Gregory 2014, 46.)

Hyvin keskeinen osa pelimoottoreissa on grafiikan renderöinti, joka käytännössä vastaan siitä, että käyttäjän ohjelmaan tuomat tai ohjelmalla tekemät tekstuurit, kuvat ja 3D-mallit piirretään näytölle. Pelimoottoreissa hyödynnetään valmiita grafiikan esittämiseen soveltuvia ohjelmointirajapitoja, joista tunnetuimmat ovat OpenGL ja DirectX. Kyseiset kirjastot sisältävät toiminnot eri muotojen piirtämiseen, joiden avulla grafiikka luodaan. DirectX on ohjelmointirajapintojen kokoelma, joka sisältää Direct3D ja Direct2D rajapinnat 3D- ja 2D- grafiikan piirtämiseen. DirectX tukee ainoastaan Windows-käyttöjärjestelmiä (DirectX Graphics and Gaming.) DirectX:än suurin kilpailija on OpenGL, ja yleensä pelimoottoreista löytyy tuki näille molemmille. OpenGL mahdollistaa grafiikan piirtämisen myös muilla käyttöjärjestelmillä, mukaan lukien matkapuhelimet ja pelikonsolit. OpenGL on alun perin kirjoitettu C++-kielellä, mutta käännetty esimerkiksi C#-, Java- ja Boo- kielille. (OpenGL Getting_Started 2014.)

Pelimoottorit sisältävät yleensä komponentit animaatioiden hallintaan ja luomiseen. Animaatioita tarvitaan esimerkiksi, kun halutaan saada peliin tuotu hahmo juoksemaan tai kävelemään. 2D-animaatioissa hyödynnetään Sprite-animaatio tekniikkaa, jossa yhdestä hahmosta on monta kuvaa eri asennoissa, näitä kuvia nopeasti vaihtaen saadaan luotua illuusio liikkuvasta hahmosta. 3D-animaatioiden toteuttamisessa suositu tapa on luurankomalli, jossa 3D-mallin eri osat ovat liitetty toisiinsa nivelten avulla. Luurankomalleilla voidaan jäljitellä esimerkiksi ihmisen liikkeitä hyvin realistisesti. (Gregory 2014, 543.) Itse mallien tekeminen ei tapahdu pelimoottorissa vaan 3D-mallintamiseen tarkoitetuilla ohjelmilla, joita ovat esimerkiksi Blender, Maya ja 3dsMax. Pelimoottorit tukevat yleensä myös kokonaan valmiiksi animoitujen mallien tuomista peliin. 2D-puolella Sprite-animaatiossa käytettävät kuvat voidaan tehdä yleisillä kuvankäsittelyyn tarkoitetuilla ohjelmilla, kuten Photoshop, Corel tai Gimp.

Useita pelimoottorien ominaisuuksia päästään hyödyntämään suoraan käyttöliittymän kautta. Tämän lisäksi pelimoottorit sisältävät erilaisia skriptikieliä, joiden avulla pystytään hyödyntämään kaikkia pelimoottorin ominaisuuksia, lisäksi skriptien avulla luodaan peliin toiminnallisuus. Skriptejä hyödyntämällä määritellään, mitä milloinkin pelissä tapahtuu, mikä tai kuka liikkuu. Skriptejä tarvitaan esimerkiksi silloin, kun

fysiikkamoottori on laskenut törmäyksen. Törmäyksen jälkeen voimme skriptillä tutkia, mitkä peliobjektit ovat törmänneet ja muuttaa pelitapahtumia sen mukaan.

Skriptit kirjoitetaan yleensä korkeantason ohjelmointikielillä, mikä käytännössä tarkoittaa, että skriptikielet ovat ohjelmointikieliä, joita tulkataan toisen ohjelman toimesta. Pelimoottorit usein sisältävät muutaman eri skriptikielen, joilla ohjelmointi pelimoottorin sisällä voidaan suorittaa. Skriptikielien suurimpana etuna on niiden korkealla tasolla toimiminen, joka mahdollistaa monimutkaisten tapahtumien ohjelmoinnin suhteellisen pienellä määrällä koodia. Skriptikielten suurin heikkous on niiden hitaus C++-kieleen verrattuna, jolla suurin osa pelimoottoreista on ohjelmoitu. (McShaffry,Graham 2012, 334–335.)

3 2D-FYSIIKKAMOOTTORIT

Fysiikkamoottorien tarkoitus on tuoda peliin oikeassa elämässä vaikuttavat fysiikan lait. Syy miksi fysiikkamoottoreita käytetään, on sama kuin pelimoottoreissa, niiden käyttö säästää aikaa, koska samoja asioita ei tarvitse tehdä uudestaan. Valmiiden ratkaisuiden käyttö takaa usein myös laadukkaamman lopputuloksen. Käytännössä fysiikkamoottorit ovat ohjelmointikirjastoja, jotka sisältävät matemaattiset kaavat erilaisten tapahtuminen laskemiseen. Fysiikkamoottori on siis laskukone, joka pitää liittää osaksi ohjelmistoa, jonka jälkeen sen sisältämiä laskukaavoja voidaan kutsua ohjelmassa. (Millington 2007, 2–4.) Pelimoottoreihin fysiikkamoottorit on yleensä integroitu niin että, fysiikoiden laskemiseen tarvittavia arvoja voidaan antaa peliobjekteille suoraan editorin kautta, tämän lisäksi niihin voidaan vaikuttaa skriptien avulla.

Todellisuutta simuloivien fysiikoiden käyttämisellä voidaan saavuttaa suuria etuja peleissä. Fysiikoiden käytöllä on helppoa luoda pelaajan kannalta tärkeää vaihtelevuutta peleihin. Otetaan esimerkiksi peli, jossa pallo voi heiton voimasta törmätä seinään. Fysiikkamoottorin avulla pallolle voidaan asettaa massa, jonka jälkeen heitolle voidaan lisätä voima ja suunta, joiden avulla pallo saadaan liikkeelle. Tämän jälkeen fysiikkamoottoria tarvitaan tutkimaan, milloin pallo törmää seinään. Törmäyksen jälkeen fysiikkamoottori laskee pallolle uuden suunnan ja nopeuden. Ilman fysiikkaa kyseinen operaatio pitäisi toteuttaa animaatioiden avulla, jolloin vaihtelevuuden aikaansaaminen olisi huomattavasti työläämpää. On tärkeää kuitenkin miettiä, milloin on tarpeellista käyttää valmista fysiikkamoottoria. Yksinkertaisissa 2D-peleissä ei

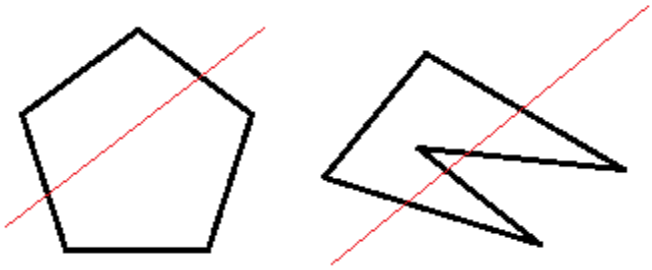
välttämättä tarvita muita ominaisuuksia kuin törmäyksen tunnistusta, jolloin valmiin fysiikkamoottorin käyttö saattaa tuntua ylimitoitetulta. Fysiikoiden laskeminen tapahtuu prosessorin laskentatehoa hyväksi käyttäen. Minkä vuoksi erityisesti mobiililaitteille ohjelmoitaessa, joissain tapauksissa saattaa olla viisaampaa tehdä oma pieni fysiikkamoottori, josta löytyy vain tarvittavat ominaisuudet. (Millington 2007, 4.)

Törmäysten tunnistaminen

Törmäysten tunnistaminen on fysiikkamoottorien perusominaisuus, sen tehtävä on käytännössä tutkia, milloin kaksi pelimaailmassa olevaan objektia on törmännyt toisiinsa. Jotta törmäyksen tutkiminen on mahdollista, tulee fysiikkamoottorien tietää objektin muoto, paikka ja rotaatio. Törmäyksen tutkimisessa fysiikkamoottorit hyödyntävät erilaisia geometrisia muotoja kuten ympyrä, neliö ja monikulmio. Muotojen avulla piirretään peliobjektin ympärille alue joka reagoi törmäyksiin. Myös monimutkaisempien muotojen käyttäminen on mahdollista. Esimerkiksi 2D-maailmassa autoa kuvaavan objektin tömäyksen tunnistamiseen voidaan käyttää erikokoisia suorakulmioita, jotka mukailevat auton muotoja. Kun halutaan erittäin tarkkaa törmäyksen tunnistusta, voidaan objektin ympärille piirtää erikokoisista monikulmioista muodostuva tömäysalue (Gregory 2014, 655–656.)

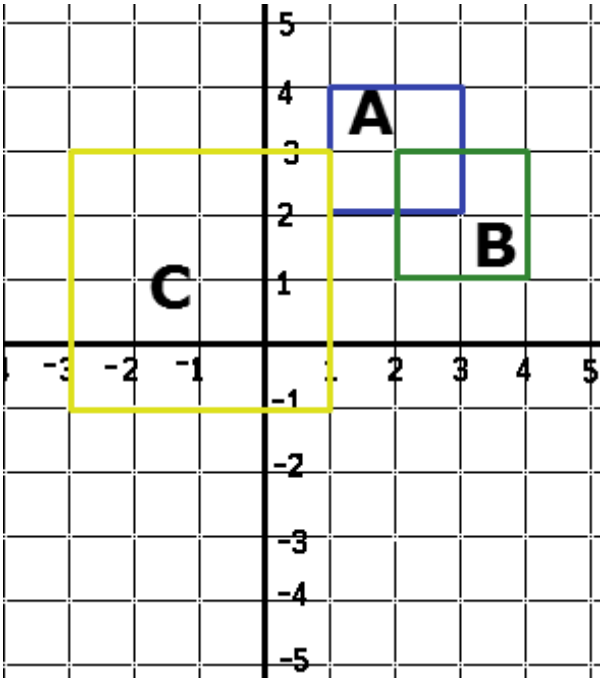
3.1 GJK-algoritmi

GJK on yksi monista törmäyksen tunnistamiseen kehitetyistä algoritmeista. Lyhenne tulee kehittäjien sukunimien mukaan E. G. Gilbert, D. W. Johnson ja S. S. Keerthi. Tarkoitus ei ole painetua algoritmin matematiikkaan, vaan kertoa pintapuolisesti mihin sen toiminta perustuu. GJK on hyvin yleinen tapa tutkia kuvassa 1 nähtävien kuperien muotojen törmäystä, ja sitä voidaan hyödyntää 2D- ja 3D-objektien kanssa. Kuperilla muodoilla tarkoitetaan sellaista muotoja, joiden poikki voidaan piirtää säde, joka leikkaa muodon pinnan vain kerran. (Gregory 2014, 660).



KUVA 1. Kupera ja ei kupera muoto

GJK-algoritmi hyödyntää Minkowskin-erotus nimistä laskukaavaa, jonka avulla voidaan laskea kahden kuperanmuodon pisteiden erotus. Erotuksen tuloksena saatujen pisteiden avulla voidaan piirtää uusi kuperamuoto. Jos muoto leikkaa koordinaatiston origon, silloin kappaleet törmäävät. Kuvassa 2 on laskettu A ja B muotojen pisteiden erotus, saatujen pisteiden avulla on piirretty uusi muoto C. C leikkaa koordinaatiston origon, tällöin voimme olla varmoja, että törmäys A:n ja B:n välillä on tapahtunut.

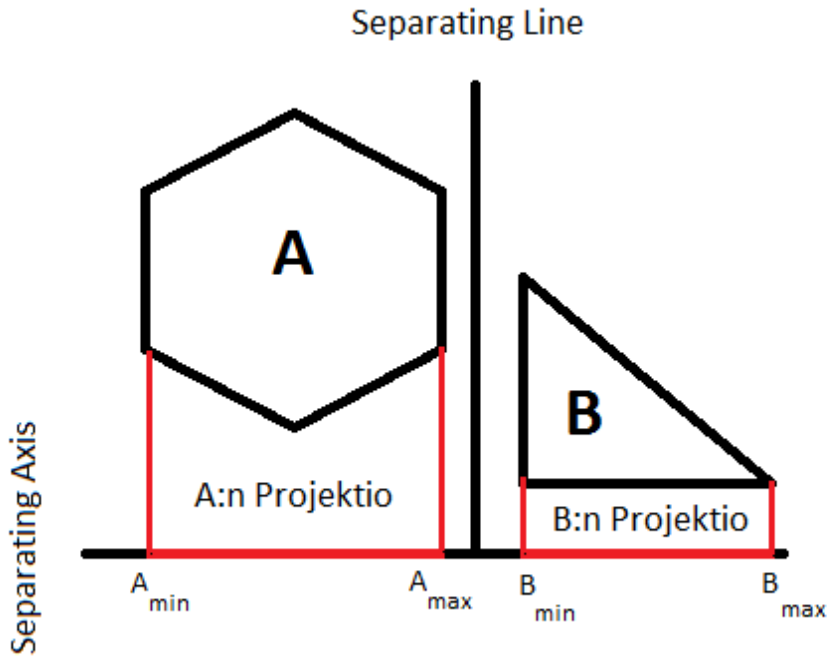


KUVA 2. Minkowskin erotus 2D koordinaatistossa

GJK-algoritmin tarkoitus tutkia on Minkowski-erotuksesta saatujen pisteiden avulla, voidaanko saadun kuperainmuodon sisään piirtää uutta monikulmiota, joka leikkaa origon. Jos voidaan niin, törmäys on tapahtunut. (Gregory 2014, 670–671.)

3.2 Separating Axis Theorem

Speartating axis theorem on toinen fysiikkamoottoreissa yleisesti käytettävä törmäyksen tunnustus menetelmä. Sen avulla tutkitaan onko kahden kuperan muodon projektion väliin mahdollista piirtää viiva, jota muodot eivät leikkaa. Mikäli tämä on mahdollista, voidaan olla varmoja että muodot eivät leikkaa myöskään toisiaan.



KUVA 3. Separating Axis Theorem kahden monikulmion välillä

Kuvasta 3 voidaan nähdä kuinka A_{max} ja B_{min} projektiopisteiden väliin jää tyhjää. Kuvitellaan että kappaleet liikkuisivat oikeakätisessä koordinaatistossa toisiaan kohti X-akselin suuntaisesti, tällöin törmäys voitaisiin tutkia seuraavalla tavalla. Jos $A_{max} > B_{min}$ ja $A_{min} < B_{max}$ niin kappaleet törmäävät. (Gregory 2014, 667 – 668.)

Jäykkien kappaleiden dynamiikka

Dynamiikka on mekaniikan ala, joka käsittelee miten voimat vaikuttavat objektien liikkeeseen. 2D-Fysiikkamoottorit keskittyvät yleensä jäykkien kappaleiden liikkeen tutkimiseen voimien vaikutuksesta. Jäykillä kappaleilla tarkoitetaan kiinteitä objekteja, jotka eivät voi muuttaa muotoaan. Fysiikkamoottorit tarjoavat yleensä mahdollisuuden luoda jäykkien kappaleiden välille erilaisia niveliä, joiden avulla saadaan todellisuutta vastaavia ominaisuuksia peleihin. Dynamiikka ominaisuudet ovat hyvin

paljon sidoksissa törmäyksentunnustamiseen, koska ne vaikuttava siihen miten kappaleet reagoivat tapahtuneen törmäyksen jälkeen. (Gregory 2014, 684–685.)

Box2D

2D-fysiikkamoottoreista tunnetuin ja käytetyin on Amerikkalaisen Eric Catton ohjelmoima Box2D, jonka pohjalta on luotu useita eri fysiikkamoottoreita useille eri ohjelmointikielille, myös monet pelimoottorit hyödyntävät Box2D:tä. Box2D-fysiikkamoottori on alun perin kirjoitettu C++-kielellä ja se on ilmaiseksi ladattavissa ja käytettävissä, myös kaupallisiin tarkoituksiin. Box2D sisältää toiminnot törmäysten havaitsemiseen, nivelten luomiseen ja jäykkien kappaleiden liikuttamiseen. (Perry 2009.)

4 UNITY

Unity on pelimoottoriohjelmisto, joka tarjoaa työkalut 3D- ja 2D-pelien tekemiseen. Se on tällä hetkellä ylivoimaisesti suosituin pelimoottori. Sivuillaan Unity ilmoittaa osuudekseen pelimoottori markkinoilla 45 %, mikä on lähes kolminkertainen lähimpään kilpailijaan. (Public relations 2014). Unityn vahvuuksia on sen nopea opittavuus ja tehokkuus, sekä mahdollisuus kääntää pelejä usealle eri alustalle. Tuettuja alustoja ovat

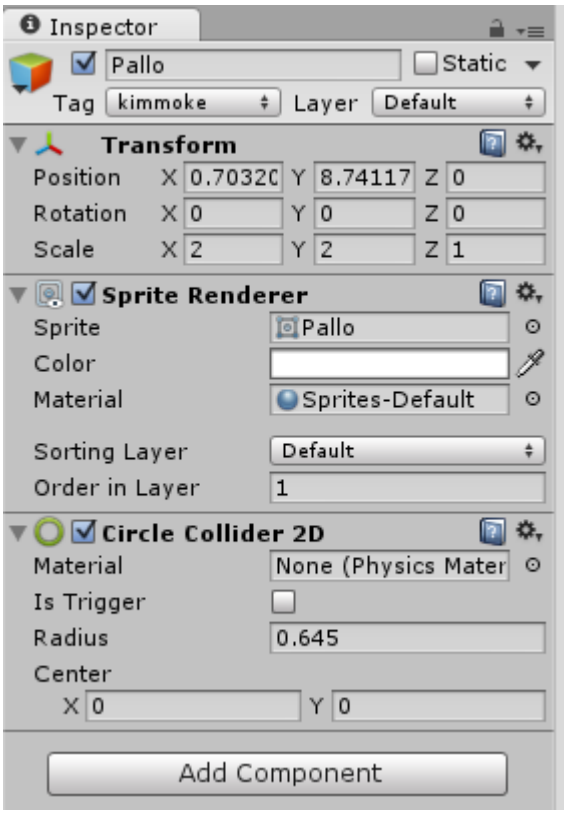
- Android,
- Windows phone
- iOS
- BlackBerry
- Windows
- Linux
- Mac
- PS3, PS4, PSvita
- Xbox One, Xbox360
- WiiU
- WEB-selaimet.

Suurimpana valttina kuitenkin on Unityn ilmaisuus, jonka takia se on erityisen suuressa suosiossa yksityisten kehittäjien keskuudessa. Ohjelmisto on ilmaiseksi ladattavissa

yhtiön nettisivujen kautta. Mikäli kaupallisessa tarkoituksessa toimiva taho, mielii käyttää Unityn ilmaisversiota, sen vuosittaiset tulot eivät saa ylittää 100 000 $. Tämän jälkeen lisenssi tulisi päivittää pro-versioon. Pro-versiosta voi ladata 30 päivän ilmaisen kokeiluversion, jonka jälkeen ohjelman käyttö maksaa 75$ kuukaudessa tai koko lisenssi 1500 $. Suurimmat erot pro- ja ilmaisversion välillä liittyvät grafiikan renderöintiin. Unitysta ei ole mahdollista ostaa versioita, joka sallisi lähdekoodi muokkaamisen. (License comparisons 2014.)

4.1 Unityn ominaisuuksia lyhyesti

Unityssä jokainen peli on oma projektinsa, yksi projekti voi kuitenkin sisältää useita scenejä. Scenejä käytetään jakamaan peli pienempiin osiin. Esimerkiksi yhteen sceneen voidaan sijoittaa pelin alkuvalikot, joista päästään pelin aloittavaan sceneen. Pelin kentät on myös järkevää jakaa scenehin luomalla jokaista kenttää varten oma scene. Peliin ulkopuolelta tuotu sisältö on käytettävissä kaikissa sceneissä, mutta jokaisen scenen rakentaminen alkaa tyhjästä. Scenet helpottavat huomattavasti pelikehitysprosessin hallintaa etenkin, jos kyseessä peli johon on tarkoitus tehdä useita kenttiä.



KUVA 4. Peliobjekti johon on liitetty komponentteja

Peliprojektiin tuotu digitaalinen sisältö liitetään sceneen peliobjektien avulla, joiden ympärille koko Unityn toiminta perustuu. Tyhjät peliobjektit eivät varsinaisesti tee mitään, vaan toimivat säilytystilana komponenteille. Esimerkiksi liittämällä Sprite renderer-komponentti peliobjektiin voidaan pelimoottoriin tuotu kuva piirtää pelimaailmaan. Yksi Unityn peruskomponenteista on Transform-komponentti, joka liitettään automaattisesti jokaiseen luotuun peliobjektiin. Transform-komponentti määrittää peliobjektin sijainnin, kierron ja mittakaavan pelimaailmassa. (Unity Manual Game-objects 2014.)

Peliobjekteista on mahdollista luoda prefabeja, jotka ovat kopiota alkuperäisistä peliobjektista ja näin ollen ne sisältävät kaikki samat komponentit ja asetukset mitä alkuperäinen peliobjekti. Kerran luotuja prefabeja voidaan käyttää kaikissa projektin sceneissä, mikä nopeuttaa scenejen rakentamista, koska peliobjekteja ei tarvitse jokaista sceneä varten koota uudestaan. Jos prefabeihin tulee tarve tehdä muutoksia, niin riittää että muutokset tehdään yhteen scenessä olevaan prefabiin, jolloin ne voidaan päivittää myös muihin samalla prefabilla luotuihin peliobjekteihin. (Unity Manual Prefabs 2014.)

Vaikka Unity tarjoaa nopean tavan saada näytölle tavaraa, vaatii sen käyttö perustietämyksen ohjelmoinnista, jonka avulla luodaan toiminnallisuus peliin. Tarjolla on kolme eri ohjelmointikieltä Boo, JavaScript ja C#, josta nykyään suosituin on C#. C#:n käyttö Unityssa perustuu Mono-ohjelmistokirjastoon, joka on avoin versio .NET-ohjelmistokirjastosta. Mono sisältää kääntäjän C#-kielelle, jonka ansiosta se toimii eri käyttöjärjestelmillä. Tämä mahdollistaa Unityn käytön Windowsin lisäksi myös Linux- ja Mac OS X -käyttöjärjestelmissä. (Sueman 2014, 197.) Unity tarjoaa asennuksen mukana MonoDevelop-editorin skriptien kirjoittamiseen, vaihtoehtoisesti voidaan käyttää myös muita ohjelmointieditoreja. Unityssä käytettävät skriptitiedostot toimivat samalla tapaa kuten muutkin komponentit, valmis skripti liitetään haluttuun peliobjektiin, jonka jälkeen peliobjekti tottelee skriptitiedostoon kirjoitettuja komentoja.

```
using UnityEngine;
using System.Collections;

public class NewBehaviourScript : MonoBehaviour {

    // Use this for initialization
    void Start () {

    }

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

    }
}
```

KUVA 5. C# skriptitiedosto Unityssa

Kuvassa 5 nähdään tyhjä skriptitiedosto, johon ei ole lisätty käyttäjän kirjoittamaa skriptiä. Unity luo uuteen skriptitiedostoon valmiiksi oletuksena käytettävät nimiavaruudet UnityEngine ja System.Collection. Mikäli käyttäjä ole määrittänyt skriptitiedostolle omaa nimeä Unity antaa sille oletuksena nimen NewBeahaviorScrip, annettu nimi toimii samalla skriptiedoston luokan nimenä, joten tiedostot on järkevää nimetä niiden sisältöä kuvaavalla nimellä. Jokainen luotu skriptitiedosto periytetään sisäänrakennetusta MonoBehavior-luokasta, joka sisältää ohjelmointiin tarvittavat oliot. Tiedostossa on valmiina kaksi funktiota Start ja Update. Start-funktio ajetaan joka kerta pelin käynnistyksen yhteydessä, jonka vuoksi sitä käytetään yleensä muuttujien alustamiseen ja peliobjekteihin liittämiseen. Update-funktio hoitaa ruudunpäivitystä, ja sen sisään sijoitetut toiminnot tapahtuvat pelissä 60 kertaa sekunnissa. (Unity Manual Scripting Overview 2014.) Update-funktion sisään voidaan esimerkiksi sijoittaa koodi, jolla halutaan liikuttaa pelihahmoa paikasta A paikkaan B. Komponentteihin liitettäviin skirpteihin on mahdollista luoda yleisentason muuttujia, joiden arvoja voidaan asettaa suoraan editorinäkymän välityksellä.

4.2 Unityn 2D-fysiikkaominaisuudet

Virallisen 2D tuen Unity sai 4.3 version mukana, tärkeimpänä lisänä oli Box2D-fysiikkamoottori. Muina uusina ominaisuuksina tuli tuki käyttää Sprite-grafiikkaa ja työkalut 2D-animaatioiden luomiseen. 2D-fysiikkakomponenttien käyttöönotto editorissa tapahtuu samaan tapaan kuin 3D-fysiikkakomponenteilla. Suuri osa uusista 2D-komponenteista, on samoja mitä löytyy 3D-puolelta

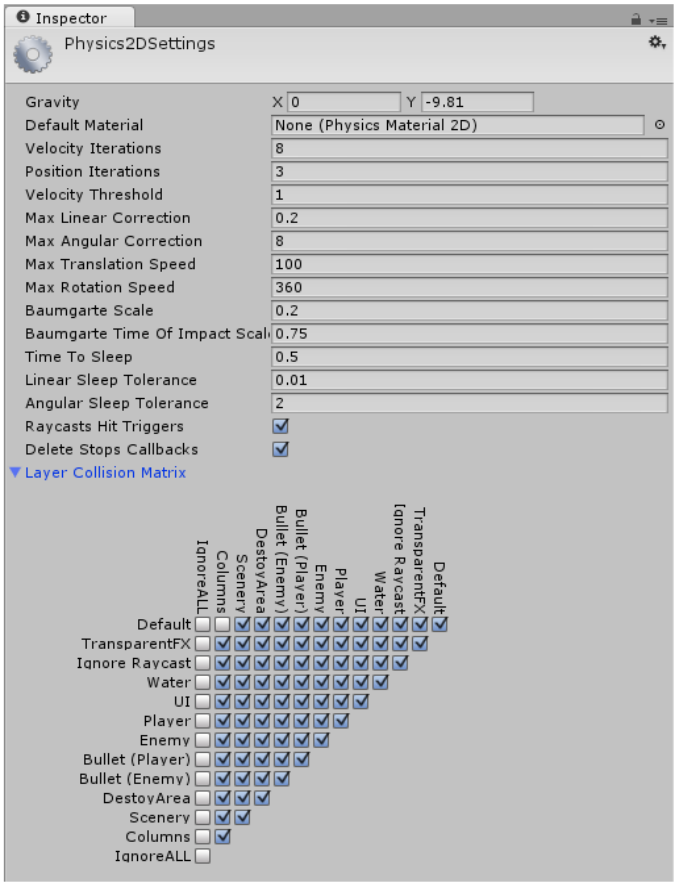
• Rigidbody 2D
• Circle Collider 2D
• Box Collider 2D
• Polygon Collider 2D
• Edge Collider 2D

• Distance Joint 2D
• Hinge Joint 2D
• Wheel Joint 2D
• Slider Joint 2D
• Spring Joint 2D.

2D- ja 3D-pelien keskeisin ero on Z-akselin puuttuminen pelimaailmasta. 2D-peleissä ei siis ole syvyyttä, vaan kaikki toiminta tapahtuu X- ja Y-akselien suuntaan. Unity suunniteltiin alun perin 3D-pelin tekemiseen, mutta sen alkuajoista asti sitä on sovellettu myös 2D-peleihin, vaikka pelimoottori ei niiden tekemistä varsinaisesti ole tukenut. Ennen Unityn 4.3 versiota, 2D-pelejä tehdessä käyttäjän piti manuaalisesti lukita Z-akselin suuntaan tapahtuva liike ja käyttää 3D-pelien tekemiseen tarkoitettuja fysiikkakomponentteja. Toinen hyvin työlästapa oli toteuttaa kokonaan oma yksikertainen fysiikkamoottori, mikä johti usein siihen, että peleissä oli huomattavasti enemmän virheitä. Lisäksi oli ylimitoitettu käyttää 3D-fysiikkamoottoria 2D-pelin tekemiseen. (Jackson 2014, 16 - 18.)

4.2.1 2D-fysiikka-asetukset

Unityn 2D-fysiikka-asetukset löytyvät editorissa polun Edit-Project-Settings-Physics2D takaa. Asetuksilla päästään vaikuttamaan fysiikkatoimintojen päivitysnopeuteen ja päästään asettamaan eri toiminnoille maksimi arvoja. Esimerkiksi Gravity-arvoa säätämällä voidaan muuttaa fysiikkamoottorin alaisuudessa toimiviin komponentteihin vaikuttavan painovoiman määrään. Arvoja muuttaessa on tärkeää ymmärtää, että suurempi päivitysnopeus, tekee pelistä raskaamman, mikä saattaa olla haitaksi etenkin mobiililaitteille tarkoitetuissa peleissä. (Unity Manual Physics 2D Manager 2014.)



KUVA 6. 2D fysiikka-asetukset Unityssa

Tärkeä ominaisuus fysiikka-asetuksissa on kuvan 6 alereunassa oleva Layer Collision Matrix, jonka avulla määritetään mitkä peliobjektit voivat törmätä toisiinsa. Kuvassa 6 nähdään ignoreALL-taso jossa ei ole yhtäkään rastia, tämä tarkoittaa että kyseiselle tasolle sijoitetut peliobjektit eivät voi törmätä minkään toisen peliobjektin kanssa.

4.2.2 Rigidbody 2D -komponentti

Rigidbody 2D on peliobjekteihin liitettävä komponentti, sen avulla saadaan haluttuun peliobjektiin fysiikkamoottorin tarjoamat ominaisuudet. Komponentin käyttöönoton jälkeen peliobjektiin vaikuttaa painovoima, ja sen liikkeitä voidaan hallita lisäämällä Rigidbody 2D -komponentille esimerkiksi nopeutta, voimaa ja massaa. Rigidbody 2D -komponenttia tarvitaan myös, kun halutaan kahden peliobjektin reagoivan törmäyksiin automaattisesti. Käyttäjän ei tarvitse huolehtia törmäyksessä muuttuvien voimien ja suuntien laskemisesta vaan fysiikkamoottori hoitaa ne.

KUVA 7. Rigidbody 2D -komponentti Unity editorissa

Kuvassa 7 näkyy Rigidbody 2D -komponentin keskeiset ominaisuudet, joihin päästään vaikuttamaan suoraan editorin kautta. Rigibody 2D -komponentin Mass-ominaisuus kuvaa peliobjektin massaa, joka vaikuttaa peliobjektin liikkeeseen aivan kuten todellisuudessa, Newton 2 lain mukaan F = ma eli voima = massa * kiihtyvyys. Mitä suurempi massa peliobjektille annetaan, sitä suuremman voiman se vaatii liikkuakseen. Linear Drag- ja Angular Drag -arvojen avulla voidaan simuloida esimerkiksi ilmanvastuksen vaikutusta liikkeeseen. Mitä suurempi Drag-arvo on, sitä suurempi on peliobjektiin vaikuttava ilmanvastus. Linear Drag vaikuttaa peliobjektien X- ja Y-akselin suuntaan tapahtuvan liikkeen hidastumisnopeuteen, kun taas Angular Drag peliobjektin kiertoliikkeen hidastumiseen. Gravity Scale -asetuksella voidaan vaikutta panovoimaan, jos arvo asetetaan nollaan, peliobjektiin ei vaikuta painovoima, jolloin voimaa lisäämällä objekti jatkaisi liikettään ikuisesti. Fixed Angle -vaihtoehdon ollessa valittu peliobjektin kierto on lukittu, jolloin, sen ei ole mahdollista pyöriä akselinsa ympäri voiman vaikutuksesta. IsKinematic -valinnalla voidaan peliobjekti asettaa tilaan, jossa se ei reagoi annettuun voimaan. Interpolate -kohdan alta löytyy kolme valintaa, jotka ovat None, Interpolate ja Extrapolate. Näitä ominaisuuksia tarvitaan silloin, kun grafiikan päivittäminen tapahtuu eri aikaan kuin fysiikkamoottorin. Tämä voi johtaa peliobjektien nykivään liikkumiseen. Interpolate valinta sulavoittaa liikkeen peliobjektin edellisten paikkatietopäivitysten perusteella, kun taas Extrapolate pyrkii ennakoimaan tulevia paikkatietopäivityksiä. Interpolate- ja Extrapolate -ominaisuuksia suositellaan käytettäväksi kohteissa joita kamera pelissä seuraa. (Unity Manual Physics 2D Reference 2014.)

Sleeping Moden avulla voidaan peliobjekti asettaa lepotilaan, jonka aikana se ei kuluta prosessorin laskentatehoa. Valikko tarjoaa kolme vaihtoehtoa, jotka ovat NeverS-

leep, StartAwake ja StartAsleep. NeverSleep valittuna peliobjektin fysiikoiden laskeminen ei mene lepotilaan missään vaiheessa. StartAwake asetuksen ollessa valittu komponentti on toiminnassa pelin käynnistyksessä, mutta palaa automaattisesti lepotilaan, kun siihen ei enää vaikuta voima. StartAslpeep kohdan ollessa valittu kohde on pelin käynnistyessä lepotilassa ja herää törmäyksen voimasta. Sleep Moden käyttö on tarpeellista erityisesti, silloin kun käytössä on useita Rigidbody 2D -komponentteja, eikä kaikkien kohteiden jatkuva fysiikoiden päivittäminen ole välttämättä tarpeellista. Collision Detecteion -valinnalla voidaan vaikuttaa siihen miten fysiikkamoottori päivittää törmäys operaatiota. Valikko tarjoaa vaihtoehdot Continuous ja Discrete. Continuous ollessa valittu törmäyksien tutkiminen tapahtuu ruudunpäivitysten välissä. Discrete vaihtoehdon ollessa valittu törmäyksien tutkiminen suoritetaan fysiikkamoottorin päivityksen yhteydessä. Continuous valintaa tulisi käyttää, silloin kuin kyseessä peliobjekti jota liikutetaan nopeasti. (Unity documentation Scripting Api Rigidbody2D 2014.)

TAULUKKO 1. Rigidbody 2D -komponentin metodeja

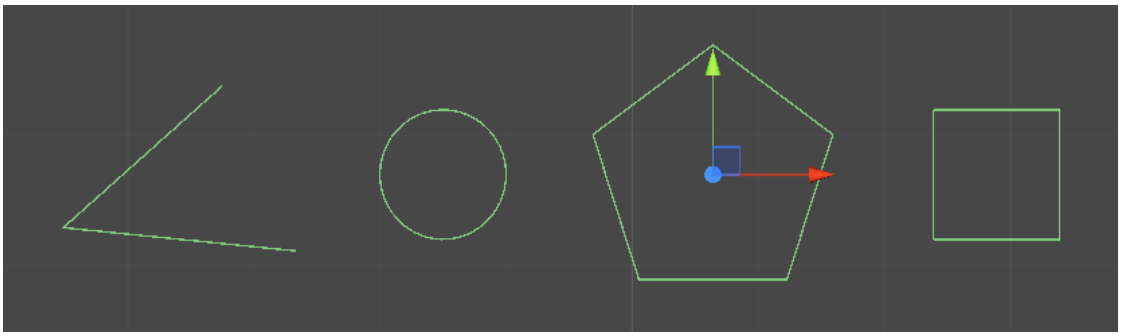
Metodi	**Tehtävä**
RigidBody2D.centerOfMass	Massakeskipiste, joka oletuksena sijaitsee keskellä objektia.
RigidBody2D.velocity	Nopeus Y -ja X-akselin suuntaan.
RigidBody2D.angularVelocity	Kulmanopeus.
rigidbody2D.Addforce	Voiman lisääminen.
rigidbody2D.AddTorque	Väännön lisääminen.
rigidbody2D.Sleep	Peliobjektin asettaminen lepotilaan.
rigidbody2D.WakeUp	Peliobjektin herättäminen lepotilasta.

Editorin kautta muunneltavien arvojen lisäksi, RigidBody 2D -komponentti sisältää useita muita ominaisuuksia, kuten voiman tai nopeuden lisääminen, joihin ei päästä vaikuttamaan suoraan editorista. Taulukkoon 2 on listattu keskeisiä ominaisuuksia, joita voidaan muuttaa tai tarkistella vain skriptien avulla.

4.2.3 Collider 2D -komponentit

Jotta Rigidbody 2D -ominaisuuksia päästään hyödyntämään tarvitaan Collider 2D -

komponentteja, joiden tehtävä on tutkia milloin kohde on törmännyt pelissä. Kun Collider 2D -komponentti on liitetty peliobjektiin, sen ympärille muodostuu kuvassa 8 nähtävä vihreällä viivalla piirretty alue, joka ilmaisee alueen jossa törmäyksentunnistus vaikuttaa. Jokainen Collider 2D -komponentti sisältää IsTrigger-toiminnon, jonka avulla voidaan tutkia milloin tömäys on tapahtunut, ilman että peliobjektit reagoivat törmäykseen. Esimerkiksi, jos kaksi palloa törmää pelissä toisiinsa, toinen palloista on asetettu IsTrigger-tilaan, toinen taas ei. Pallojen osuessa tosiinsa, fysiikkamoottori laskee törmäyksen ja tietoa voidaan hyödyntää skriptien avulla, mutta pallot eivät reagoi törmäykseen. Jokaiselle Collider 2D -komponentille on myös mahdollista liittää 2D-fysiikkamateriaali, joiden avulla pystytään simuloimaan erilaisten materiaalien toimintaa.



KUVA 8. Collider 2D -komponentit scene näkymässä

Box Collider 2D on komponentti, jonka avulla voidaan piirtää neliönmuotoinen törmäysalue. Törmäyslaatikon kokoa voidaan säätää X- ja Y-akselin suunnassa. Box Collider 2D on prosessoritehon kannalta kaikkein edullisin tapa tutkia tömäyksiä. Siksi sitä tulisikin suosia aina, kun törmäyksen tunnistamisen ei tarvitse olla erityisen tarkka. Circle Collider 2D -komponentti toimii samaan tapaan ainoana erona, että komponenttia käytetään pyöreiden peliobjektien törmäyksentunnistamiseen ja koon säätö tapahtuu muuttamalla tömäysalueen säteen kokoa. Polygon Collider 2D -komponenttia voidaan hyödyntää muodoltaan monimutkaisempien peliobjektin törmäyksen tunnistamiseen. Polygon Collider 2D -komponentti piirtää automaattisesti erikokoisista monikulmioista muodostuvan törmäysalueen vapaamuotoisen peliobjektin ympärille. Törmäys aluetta voidaan muokata jälkeenpäin yksinkertaisesti valitsemalla peliobjekti Scene-näkymästä ja painamalla näppäimistön vaihto-näppäin pohjaan, jonka jälkeen törmäysalueeseen voidaan lisätä uusia solmu kohtia tai muokata vahoja.

Viimeisin lisäys Collider 2D -komponentteihin on Edge Collider 2D -komponentti, joka toimii lähes samalla tapaa kuin Polygon Collider 2D -komponentti. Erona kuitenkin se että Edge Collider 2D tarjoaa valmiina vain viivan, johon solmukohtia lisäämällä käyttäjä voi luoda vapaamuotoisen törmäysalueen. Alueen ei kuitenkaan tarvitse sulkeutua, eli alku ja loppu solmut voivat olla eripaikoissa (Unity Manual Physics 2D Reference 2014).

TAULUKKO 2. Törmäysten tutkimiseen käytettäviä metodeja

Metodi	**Tapahtuma**
OnCollisionEnter2D	Kutsutaan hetkellä jolloin kontakti alkaa.
OnCollisionExit2D	Kutsutaan hetkellä jolloin kontakti päättyy.
OnCollisionStay2D	Kutsutaan koko sen ajan kun kontakti tapahtuu.

Törmäys tapahtumiin päästään käsiksi liittämällä Collider 2D -komponentin sisältävään peliobjektiin skriptitiedosto, jossa voidaan kutsua taulukon 3 ominaisuuksia. Vastaavat ominaisuudet löytyvät myös IsTrigger tilassa olevien Collider 2D -komponenttien törmäysten tunnistamiseen.

Raycast ja Linecast

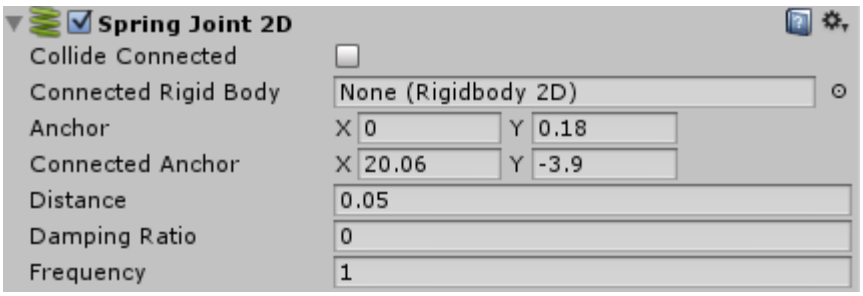
Raycast ja Linecast ovat Collider-komponenttien lisäksi toinen tapa havaita törmäyksiä. Raycast on käytännössä peliobjektista piirtyvä viiva, jolle määritetään lähtöpiste, suunta, matka ja taso jolla Raycastin halutaan toimivan. Eri tasolla olevat peliobjektit eivät voi törmätä Raycastin kanssa. Raycastia käytettäessä on tärkeää huomioida, että lähtöpiste on sijoitutettu niin, että se ei törmää peliobjektin omaan Collider-komponenttiin. Raycast-ominaisuuden käyttöön ei ole suoraa komponenttia editorissa, vaan sen luominen ja käyttö tapahtuu skriptin välityksellä. Linecast ominaisuus toimii muutoin samaan tapaa, paitsi se saa parametreina alku- ja lähtöpisteet. (Unity Documentation Physics2D.Raycast.)

Fysiikkamateriaalit

2D-materiaalinen avulla voidaan antaa peliobjekteille kitka ja kimmoisuus, joiden avulla simuloidaan erilaisia materiaaleja, kuten vaikkapa kumia tai jäätä. 2D-materiaalit liitetään Collider 2D -komponenteista löytyvään material kohtaan.

4.2.4 Joint 2D -komponentit

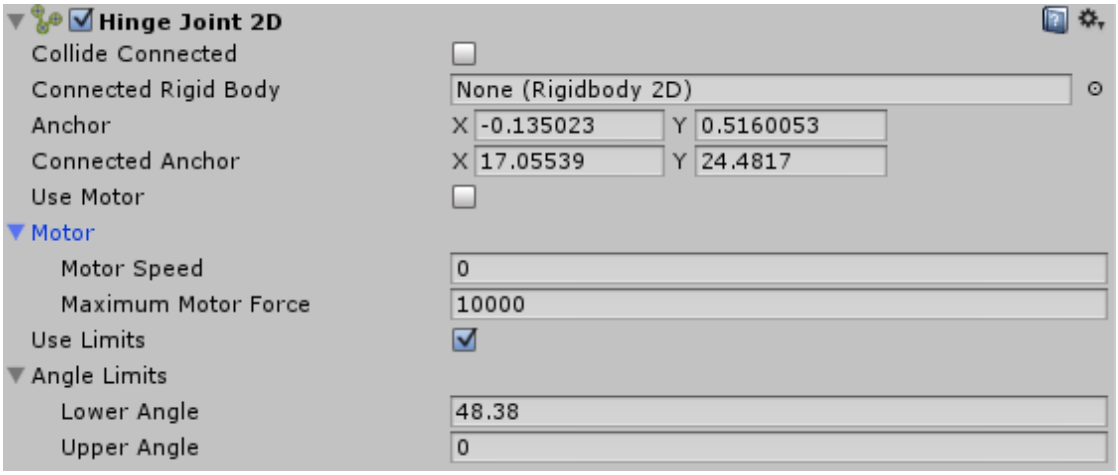
Joint 2D -komponenttien avulla peliobjektien välille voidaan luoda erilaisia niveliä, joiden avulla saadaan rajattua liikettä halutulle alueella tai luotua todellisuutta vastaavia efektejä, kuten taipuvia tai joustavia peliobjekteja. Kuvassa 9 nähtävän Spring Joint 2D -komponentin avulla voidaan kahden peliobjektin välille luoda jousi. Tällöin molemmissa peliobjekteissa tulee olla Rigidbody 2D -komponentit, joiden avulla määritetään peliobjektit joiden välillä jousi toimii. Rigidbody 2D -komponenteille annetut ominaisuudet vaikuttavat jousen toimintaan, sen mukaan miten niihin on lisätty massaa ja muita ominaisuuksia. Mikäli jousi on liitetty vain yhteen peliobjektiin, valinta jätetään tyhjäksi ja jousen toinen pää on tällöin kiinteä piste pelimaailmassa. Jousi komponentille voidaan asettaa välimatka, jota se ylläpitää lepotilassa. Lisäämällä jouseen liitettyihin peliobjekteihin voimaa, saadaan jousi venymään yli lepopituuden. Rigidbody 2D -komponenttien ja jousi komponentin asetukset vaikuttavat, siihen kuinka nopeasti jousi palautuu takaisin lepopituuteen. Collider Connected-kohdan ollessa valittuna kaksi peliobjektia, joiden välille jousi on luotu voivat törmätä toisiinsa.



KUVA 9. Spring Joint 2D -komponentti

Anchor valinnalla voidaan X- ja Y-koordinaattien perusteella määrittää missä jousen lähtöpiste on. Connected Anchor avulla määritellään jousen toinen päätepiste, joka voi olla toinen peliobjekti tai kiinteä piste pelimaailmassa. Distance-arvo kertoo välimatkan, jonka jousi pyrkii pitämään ankkuripisteiden välillä. Damping Ratio -asetuksella voidaan jouselle määrittää vaimennus-suhde, joka kertoo kuinka nopeasti venytetty jousi palaa takaisin lepotilaan. Frequency-kohdassa voidaan antaa jouselle värähtely taajuus, mitä suurempi arvo on, sitä nopeammin jousi värähtelee. (Unity Manual 2D physics Reference Spring Joint 2D 2014.) Distance Joint 2D -komponentin avulla voi-

daan kahden peliobjektin välillä säilyttää haluttu välimatka. Komponentin käyttö tapahtuu samaan tapaan kuin Spring Joint 2D -komponentin.



KUVA 10. Hinge Joint 2D -komponentti

Hinge Joint 2D -komponentin avulla voidaan luoda saranan kaltaisia kiinnityskohtia peliobjektien välille. Kuvasta 10 voidaan nähdä kaikki Hinge Joint 2D -komponentin ominaisuudet, joihin päästään käsiksi editorin kautta. Komponentti sisältää samat perustoiminnot mitä jousi-komponentti, lisäksi siihen on rakennettu Motor- ja Angle Limits -ominaisuudet. Motor-ominaisuuden avulla voidaan antaa peliobjektin kiertoliikkeelle nopeus, jonka avulla se saadaan pyörimään ankkuripisteensä ympäri. Motor Speed-kohtaan syötetään kiertoliikkeen nopeus ja Maxium Motor Force -kohtaan suurin sallittu voima, jonka peliobjekti voi saada saavuttaessaan nopeutta. Angle Limits-ominaisuudella voidaan rajata maksimi- ja minimikulmat johon peliobjekti voi liikkua. (Unity Manual 2D physics Reference Hinge Joint 2D 2014.)

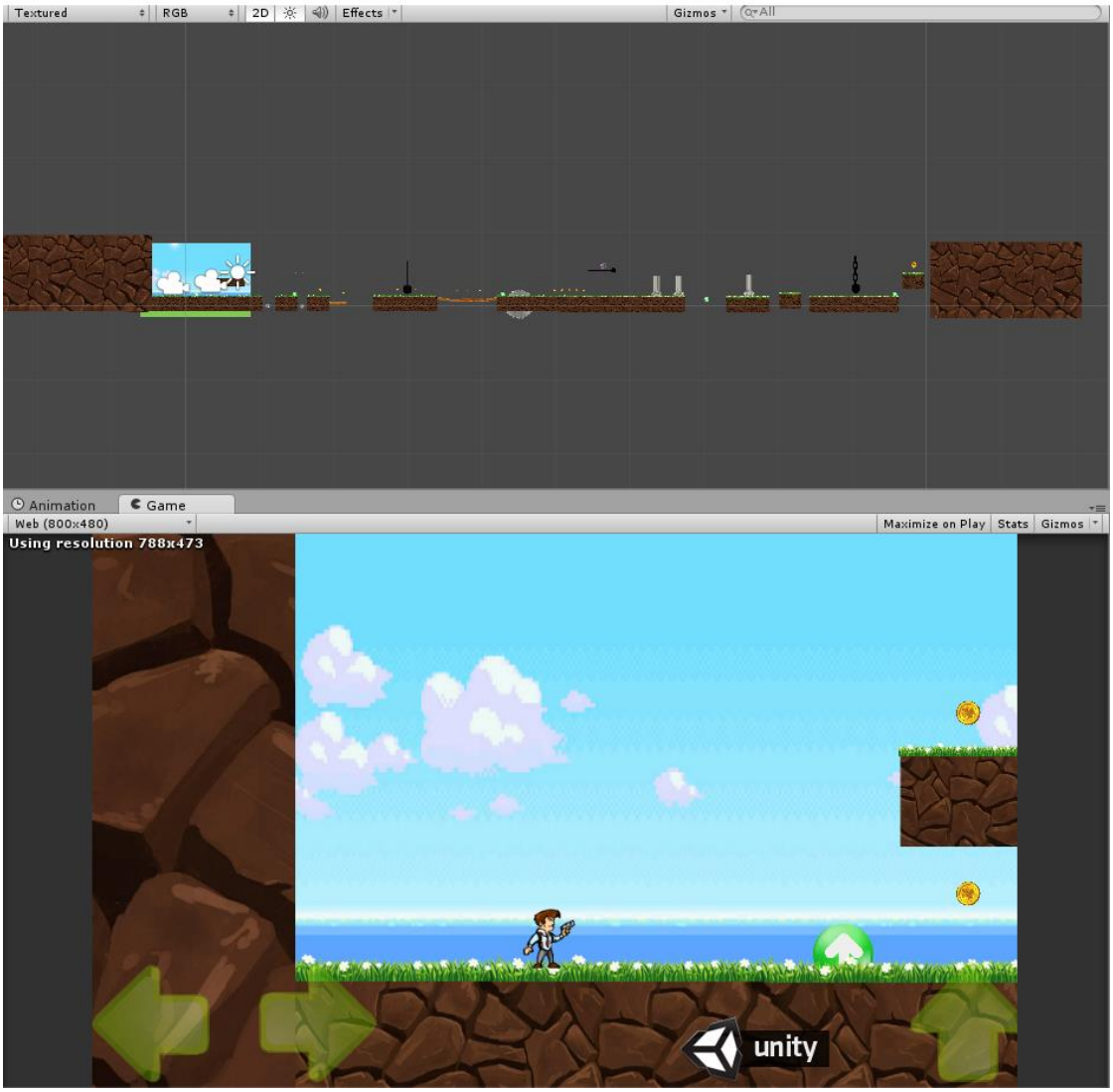
Slider Joint 2D -komponentin avulla voidaan kahden peliobjektin välille luoda linkki, jonka jälkeen ne liikkuvat samassa linjassa. Komponentti tarjoaa samat ominaisuudet kun Hinge Joint 2D, mutta kulman sijaan Slider Joint 2D -komponentilla voidaan vaikuttaa peliobjektin maksimi ja minimi matkaan ankkuripisteiden välillä. WheelJoint 2D -komponentilla voidaan simuloida pyörivän renkaan liikettä. Komponentti sisältää samat ominaisuudet mitä muutkin Joint 2D -komponentit, jonka lisäksi siitä löytyy Suspension-ominaisuus, johon kuuluu Damping Ratio ja Frequncy aivan kuten Spring Joint 2D -komponentilla (Unity Manual 2D physics Reference Wheel Joint 2D).

5 PELIDEMOT

Tein tätä opinnäytetyötä varten kaksi pelidemoa, joiden avulla voidaan havainnollistaa edellisissä luvuissa esiteltyjä Unityn 2D-fysiikkaominaisuuksia. Peleistä ei siis ollut tarkoitus tehdä visuaalisesti näyttäviä, loppuun asti hiottuja tuotoksia. Seuraavissa luvissa ole myöskään tarkoitus käsitellä kokonaisvaltaisesti pelien teko prosessia, vaan keskittyä siihen miten fysiikkamoottorin ominaisuuksilla pystytään tekemään asioita, joiden alusta asti ohjelmointi veisi huomamatavasti enemmän aikaa. Molemmat aikaan saadut demot tarjoavat kuitenkin hyvät jatkokehitys mahdollisuudet. Molempien demojen teossa on käytetty Unityn 4.5.4-versiota, joka oli kirjoitushetkellä uusin saatavilla oleva versio. Demot on suunniteltu toimivaksi PC:llä ja Windows phone -käyttöjärjestelmää tukevilla mobiililaitteilla. Molemmissa demoissa käytettiin skriptikielenä C#, koska sillä ohjelmoinnista itselläni oli ennestään eniten kokemusta. Seuraavissa kappaleissa kuvataan demojen PC-versioiden toimintaan. Ainoa ero mobiili- ja PC-versioiden välillä liittyy peliobjektin liikuttamiseen, joita varten mobiililaitteille on tehty kosketuksen tunnistavat napit, kun taas PC-versioissa liikkuminen tapahtuu näppäimistöllä.

5.1 Demo 1

Ensimmäinen demo on periteinen sivultapäin kuvattu tasohyppelypeli, jossa pelaan tehtävä kerätä matkan varrella olevia kolikoita ja varoa taivaalla lentävää lentokonetta, joka pudottelee pelimaailmaan pommeja. Pommit räjähtävät tietyn ajan kuluttua ja rähjähdyksen aikana pommin kanssa kosketuksissa ollut maa-alue tuhoutuu. Pelihahmo pystyy tuhoamaan maahan pudonneita pommeja ampumalla niitä, jolloin ne häviät ilman räjähdystä. Matkan varrelle on sijoitettu myös muita esteitä, joihin on pyritty soveltamaan Unityn tarjoamia 2D-fysiikkaominaisuuksia.



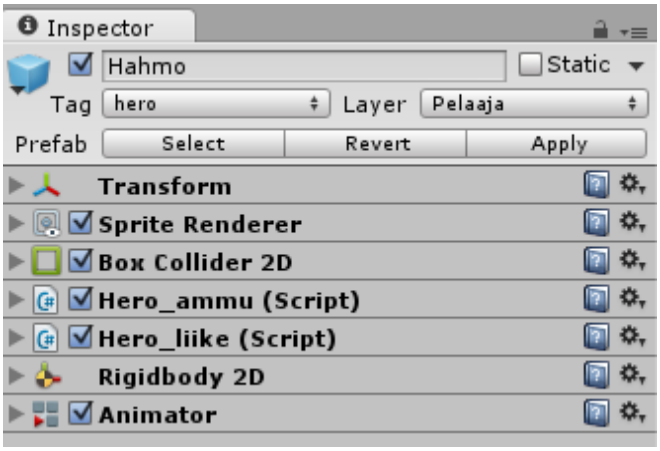
KUVA 11. Peli Unity-editorissa

Kuvan 11 yläosassa näkyy koko kenttä ja alaosassa pelin lähtötilanne. Taustalla näkyvä maisematekstuuri on sijoitettu tyhjän tasomuodon päälle, joka on käännetty 90 asteen kulmaan. Taso on skriptin avulla määritetty seuraamaan kameraa ja toistamaan tekstuuria. Näin saadaan luotua illuusio vaihtuvasta maisemasta, kun hahmoa liikutetaan. Pelimaailmaa kuvaava kamera on asetettu seuraamaan pelaajaa, siten että pelihahmo pysyy koko ajan keskellä kuvaruutua. Kamera voi liikkua ainoistaan X-akselin suuntaan, koska pelissä ei tule tilanteita, joissa pelihahmo pystyisi liikkuman Y-akselin suuntaan enempää, mitä kameran kuva-ala sallii. Kuvan 9 alareunasta voidaan havaita kosketuskontrollit, joita tarvitaan pelin pelaamiseen mobiililaitteilla. Vasemmassa reunassa on liikkuminen oikealle ja vasemmalle. Oikeassa reunassa sijaitseva nuoli ylös toimii hyppypainikkeena, ja Unity logon sisältävästä napista pelihahmo pystyy ampumaan. Kosketuskontrollit on toteutettu käyttämällä Unityn GUItexture-komponentteja.

Pelihahmolle on luotu animaatiotyökalujen avulla kävely- ja lepoanimaatiot, joiden tilaa vaihdellaan skriptin avulla, sen mukaan onko hahmo liikkeessä vai ei. Pelikuvan vasemmassa yläkulmassa on kaksi GUIlabel-elementtiä, joista toinen ilmaisee pisteet ja toinen hahmon elämän. Pistelaskuri lisää pisteitä sitä mukaa, kun pelaaja onnistuu keräämään pelimaailmaan sijoitettuja kolikoita. Elämää puolestaan vähennetään, sen mukaan miten pelaaja törmäilee matkan varrelle sijoitettuihin esteisiin. Pelinalueen loppuun on sijoitettu muita kolikoita suurempi kolikko, johon pelihahmon osuessa taso on pelattu läpi ja näyttöön piirrettään nappi, josta voitaisiin siirtyä seuraavaan tasoon.

5.1.1 Pelihahmo

Pelihahmon sisältävään peliobjektiin on liitetty yhteensä 7 komponenttia, joita tarvitaan pelihahmon eri toimintojen toteuttamiseen. Kuvasta 12 nähdään pelihahmon komponentit, joista ensimmäinen on Transform-komponentti, se sisältää tiedot peliobjektin sijainnista pelimaailmassa. Sprite Rendeder -komponentin avulla piirretään hahmon sisältävä kuva pelimaailmaan. Törmäyksen tunnistamiseen käytetään Box Collider 2D -komponenttia, joka tässä tapauksessa tarjoaa hahmon törmäyksen tunnistamiseen riittävän tarkkuuden. Hahmo-objektiin on liitetty kaksi skriptitiedostoa, joista toinen hoitaa ampumisen käsittelyn ja toinen vastaa hahmon liikuttamisesta ja törmäyksiin reagoimisesta. Rigidbody 2D -komponentin avulla saadaan pelihahmo fysiikkamoottorin toimintojen alaisuuteen. Animator-komponentti sisältää tiedot juoksu- ja lepoanimaatioiden toistamiseen.



KUVA 12. Pelihahmon komponentit

Liikkuminen pelissä tapahtuu vaihtamalla Update-funktion sisällä pelihahmon Transform-komponentin position-ominaisuuden X-arvoa. Hahmon liikuttaminen voitaisiin vaihtoehtoisesti käyttää myös Rigidbody 2D -komponenttia, vaihtamalla sen velocity-arvoa. Kuvassa 13 nähdään miten liikkuminen käytännössä toteutetaan. Ensin tutkitaan onko d-näppäintä painettu ja tarkistetaan että, elossa muuttujan arvo on true. Elossa on boolean-tyyppinen muuttuja, joka voi saada arvot true tai false, jos molemmat ehdot täyttyvät voidaan liikkua. Suunta muuttuja on toinen boolean-muuttuja, joka on asetutettu public static -tilaan, eli sitä voidaan kutsua kaikissa skriptitiedostoissa. Suunta arvoa muutetaan true -ja false -arvojen välillä, sen mukaan kumpaan suuntaan hahmo liikkuu.

```
if (Input.GetKey("d") && (elossa))

{
    suunta = true;
    nopeus = 4;
    transform.right = new Vector2(1f, 0f);
    anim.SetBool("liiku", true);
    transform.position += Vector3.right * nopeus * Time.deltaTime;
}
```

KUVA 13. Hahmon liikuttaminen oikealle

Tieto välitetään Hero_ammu skriptiin, jolloin ampumiseen suunta voidaan muuttaa, sen mukaan mihin suuntaan hahmo on menossa. Transform.right -komennon avulla märitetään mihin suuntaan hahmo katsoo. Arvo vaihtuu, sen mukaan kumpaan suuntaan liikutaan. Anim.SetBool -komennolla käynnistetään animaatio, jota toistetaan niin kauan kun d- näppäintä painetaan. Varsinainen hahmon liikkuminen tapahtuu alimmalla komennolla. Hahmon paikkaa siirretään oikealle nopeus muuttujan arvon verran yhden sekunnin aikana. Tämä saadaan aikaan kertomalla liikkumisnopeus Time.deltatime -arvolla. Muutoin hahmo liikkuisi nopeus muuttujan arvon verran jokaisella ruudinpäivityksellä, mikä tarkoittaisi että sekunnin aikana hahmo liikkuisi 60 kertaa nopeus muuttujan arvon verran.

Hyppääminen pelissä on toteutettu fysiikkamoottorin avulla. Välilyöntinäppäintä painettaessa hahmon Rigidbody2D -komponenttia kutsutaan skriptitiedostossa komennolla Rigidbody2D.AddForce. AddForce-metodille annetaan parametrina vektori, joka määrittää voimalle suunnan ja suuruuden. Hyppäämisen yhteydessä tutkitaan koskettaako hahmo maahan, jos ehto täyttyy, niin voidaan hypätä. Ilman tätä tarkistusta hyp-

pyyn olisi mahdollista lisätä tehoa, vaikka hahmo olisi jo ilmassa. Tämä johtaisi lopulta siihen, että hahmo ajautuisi hypyn voimasta ulos pelinäkymästä. Skriptissä tutkitaan OnCollisonStay2D-funktion avulla onko pelihahmo törmäyksessä maaobjektin kanssa. Mikäli törmäyksen aikana hahmo on maassa, muuttujan arvoksi asetetaan true, eli hyppääminen mahdollista. OnCollisionExit2D-funktio tutkii milloin tömäyksestä on poistuttu, jolloin maassa muuttujalle asetetaan arvoksi false. Tämän jälkeen hyppääminen ei ole mahdollista, ennen kuin hahmon on palannut kosketukseen maan kanssa. Kuvassa 14 nähdään kuinka voidaan tutkia milloin hahmo on törmäyksessä ja vaihtaa maassa muuttujan arvoa sen mukaan.

```
void OnCollisionStay2D(Collision2D hit)
{
        maassa = true;
}
void OnCollisionExit2D(Collision2D hit)
{
        maassa = false;
}
```

KUVA 14. Tutkitaan koskettako hahmo maata

Pelikentälle on sijoitettu pallonmuotoisia peliobjekteita, joiden avulla pelihahmo pystyy hyppäämään korkeammalle mitä normaalisti. Hyppyobjekteille on annettu Circle Collider 2D -komponentti, joita tarvitaan kun tutkitaan milloin pelihahmo on hyppyobjektin päällä. Törmäyksen tunnustamista ei kuitenkaan voida toteuttaa normaalisti OnCollisionEnter2D-ominaisuuden avulla, koska hyppäämisen halutaan onnistuvan ainoastaan silloin kun pelihahmo osuu hyppyobjektiin ylhäältä päin. Helppo tapa ratkaista ongelma on piirtää Raycast peilihahmon alapuolelta, jonka jälkeen skriptin avulla voidaan tutkia milloin hahmosta piirretty Raycast törmää hyppyobjektiin.

```
hit = Physics2D.Raycast(hahmo.transform.position, Vector3.down, 0.2f);

if ((hit) && hit.collider.gameObject.tag == "pallo")

    {
        rigidbody2D.velocity = new Vector2(0, 13f);
    }
```

KUVA 15. Törmäyksen tutkiminen Raycastin avulla

Kuvassa 15 nähdään kuinka RaycastHit2D tyyppiselle hit-muuttajalle määritetään Raycast-säde. Säteelle annetaan parametreina lähtöpiste, suuntavektori ja säteen pi-

tuus. Hahmo.transform.position -muuttajalla ei viitata suoraan hahmon omaan Transform-komponenttiin, vaan tyhjään peliobjektiin joka on sijoitettu pelihahmon lapsiobjektiksi, siten että se ei törmää pelihahmon Collider-komponentin kanssa. If-lauseessa tutkitaan palauttaako hit-muuttuja arvo true, eli törmääkö se johonkin ja onko törmäyksessä olevan peliobjektin tag-nimi ”pallo”. Mikäli molemmat ehdot ovat tosia, niin pelihahmon Rigidbody 2D -komponentille lisätään nopeutta Y-akselin suuntaan, mikä saa hahmon hyppään korkeammalle mitä normaalisti.

Pelihahmon on mahdollista tuhota pelikentälle pudonneita pommeja ampumalla niitä. Ampumiseen vaadittu logiikka näkyy kuvassa 16, ampuminen on selkeyden vuoksi sijoitettu omaan skriptitiedostoonsa. Ammuksesta on tehty prefab-peliobjekti, joka sisältää Rigidbody 2D- ja Circle Collider 2D -komponentit. Skriptissä kutsutaan Instantiate-metodia, jonka avulla voidaan luoda kopioita peliobjekteista. Instantiate-metodi vaatii kolme parametria jotka ovat, peliobjekti josta kopio tehdään, paikka mihin kopio pelimaailmassa sijoitetaan ja asento missä peliobjektin halutaan olevan. Hero_ammu-skritpi on liitetty pelihahmoon, joten parametreilla transform.position ja transform.rotation viitataan suoraan pelihahmon omaan Transform-komponenttiin.

```
public void ammu()
 {
     if (Time.time > tulinopeus)
     {
         tulinopeus = hidaste + Time.time;
         uusiammus = (GameObject)Instantiate(kuula, transform.position, transform.rotation);

         if (Hero_liike.suunta)
             {
                 ammuksen_nopeus = 2;
             }
         else
             {
                 ammuksen_nopeus = -2;
             }
         uusiammus.rigidbody2D.AddForce(Vector2.right * ammuksen_nopeus, ForceMode2D.Impulse);
         Destroy(uusiammus, 3);
     }
 }
```

KUVA 16. Hero_ammu skripti

Pelihahmon suunta tarkistettaan kutsumalla Hero_liike-skriptin muuttujaa suunta, jos hahmo katsoo oikealle se saa arvon true, vasemmalle katsottaessa arvon false. Näiden tietojen perusteella ammus saa, joko positiivisen tai negatiivisen nopeuden. Seuraavaksi uudelle ammukselle lisätään voimaa Rigidbody2D.AddForce-metodilla. Viimeisenä AddForce-metodi saa parametrin ForceMode2D.Impulse, joka tarkoittaa että

peliobjekti saa lisätyn voiman heti. Lopuksi uusiammus tuhotaan 3 sekuntia sen luomisen jälkeen, ellei se ole tuhoutunut sitä ennen osumalla pommiin.

Pelimaailma

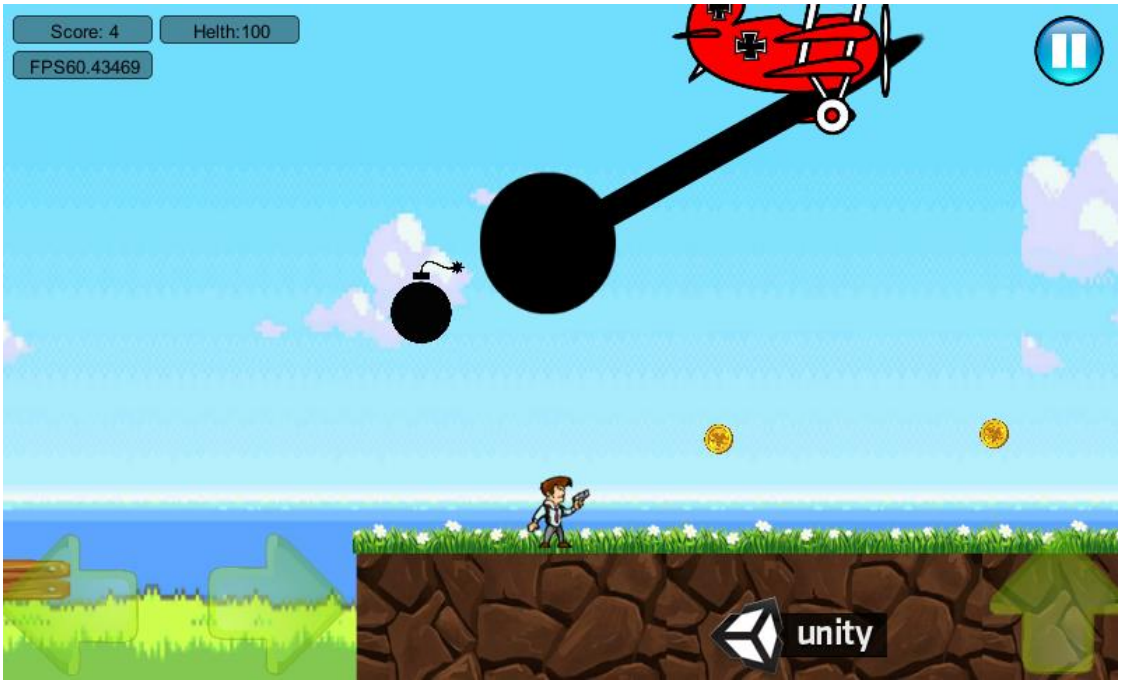
Pelimaailman maa on tehty yhteen liittämällä Sprite-kuvia. Jokainen maaobjekti sisältää Box Collider 2D -komponentin, jonka avulla pelihahmo voi liikkua maaobjektien päällä. Kentän loppuun ja alkuun on sijoitettu tyhjät peiliobjektit, jotka sisältävät Box Colldier 2D -komponentit, joilla rajataan alue, jolla pelihahmo pystyy liikkumaan. Tiedostorakenteessa kaikki maaobjektit on sijoitettu tyhjän peliobjektin lapsiobjekteiksi, josta on luotu prefabobjekti. Tämän avulla tason alustamisen yhteydessä saadaan luotua tuhoutuneiden alueiden tilalle uusi ehjä maa-alue Instantiate-metodin avulla. Pelikentän alapuolelle on sijoitettu koko kentän mittainen Box Collider 2D -komponentti, jonka avulla havaitaan onko pelihahmo pudonnut kentän varrella oleviin kuiluihin. Pelimaailmaan on sijoitettu kolikoita, joita pelaajan on kerättävä saadakseen pisteitä. Kolikko-objektit sisältävät Circle Collider 2D -komponentit, jotka on asetettu IsTrigger-tilaan, koska kolikoiden ei haluta reagoivan törmäyksiin muiden peliobjektien kanssa. Skriptissä tutkitaan milloin pelaaja osuu kolikkoon. Törmäyksen tapahduttua, kolikko-objekti piilotetaan pelinäkymästä.

5.1.2 Esteet

Peliin on pyritty tuomaan vaikeutta sijoittamalla matkan varrelle erilaisia esteitä, joita pelaajan tulee vältellä pelin edetessä. Pelin aikana hahmoa seuraa taivaalla lentävä lentokone, joka pudottelee pelimaailmaan pommeja. Mikäli pelaaja ei ehdi tuhota pommeja ennen räjähdystä, räjähdyksen kanssa kosketuksessa ollut maa-alue tuhoutuu. Lentokoneen paikkaa siirretään, sitä mukaan kun pelaaja liikkuu. Koneen lennettyä ulos kameranäkymästä, sen paikka X-akselilla päivitetään takaisin kameran taakse, jolloin saadaan aikaan efekti kameranäkymään saapuvasta uudesta lentokoneesta. Pommien pudottaminen on toteutettu omassa skriptissään hyödyntäen Instantiate-metodia. Pommien putoamistiheyttä säädellään Random.range-metodin avulla. Pommi peliobjektiin on liitetty Cirle Collider 2D -komponentti ja materiaali, johon on lisätty kimmoisuutta, jonka avulla saa pommit pomppimaan pitkin kenttää. Pommit räjähtävät viiden sekunnin kuluessa niiden luomisesta. Räjähdyksen aikana toistetaan Unityn

partikkeli systeemillä tehty räjähdys efekti, jos pelihahmo on efektin toiston aikana kosketuksissa pommin kanssa, johtaa se pelihahmon kuolemaan.

Pommien lisäksi pelihahmon reitille on sijoitettu erilaisia esteitä joihin osumista tulee välttää. Tarpeeksi monta osumaa saatuaan pelihahmo kuolee ja peli on aloitettava alusta. Ensimmäisenä esteenä on Y-akselin suuntaan liikkuvat sahanterät, joiden liike on toteutettu animaatioiden avulla, peliobjekteihin on liitetty Circle Collider 2D -komponentit, jolle on annettu materiaali, joka saa pelihahmon kimpoamaan sahasta. Jokainen törmäys sahan kanssa vähentää pelihahmon elämää 25 prosenttia. Seuraava este on heiluri, jonka toteuttamisessa on hyödynnetty Hinge Joint 2D -komponenttia.



KUVA 17. Heiluri pelinäkymässä

Kuvasta 17 nähdään miltä heiluri näyttää pelinäkymässä. Heiluri-peliobjekti koostuu kahdesta eri Sprite-kuvasta, jotka ovat heilurin varsi ja siihen liitetty moukari. Moukari ja varsi on liitetty Hinge Joint 2D -komponentin avulla yhteen, ja varsi on liitetty Hinge Joint 2D -komponentin avulla peilimaailmaan. Varren Hinge Joint 2D -komponentille on annettu maksimi- ja minimikulmat, joiden välillä sen sallitaan liikkua. Varsiosaan on liitetty skripti, jossa Hinge Joint 2D:n Motor ominaisuudelle annetaan maksimi vääntömomentti ja nopeus. Update-funktion sisällä tutkitaan milloin heilurin on saavuttanut ääriasennon, jonka jälkeen moottorin nopeus vaihdetaan negatiiviksesi, näin heilurissa saadaan ylläpidettyä tasaista liikettä. Törmäys heilurin kans-

sa johtaa pelihahmon kuolemaan. Kuvassa 18 nähdään, miten heilurin suunnanvahtiminen on koodissa toteutettu.

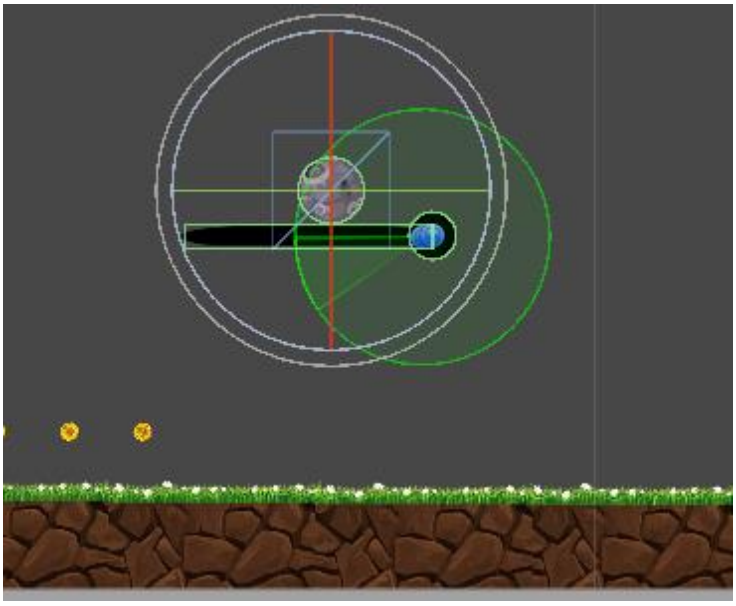
```
if (hj.jointAngle > 70)
{
    motor.motorSpeed = -100;
    hj.motor = motor;

}

else if (hj.jointAngle < -70)
{
    motor.motorSpeed = 100;
    hj.motor = motor;
}
```

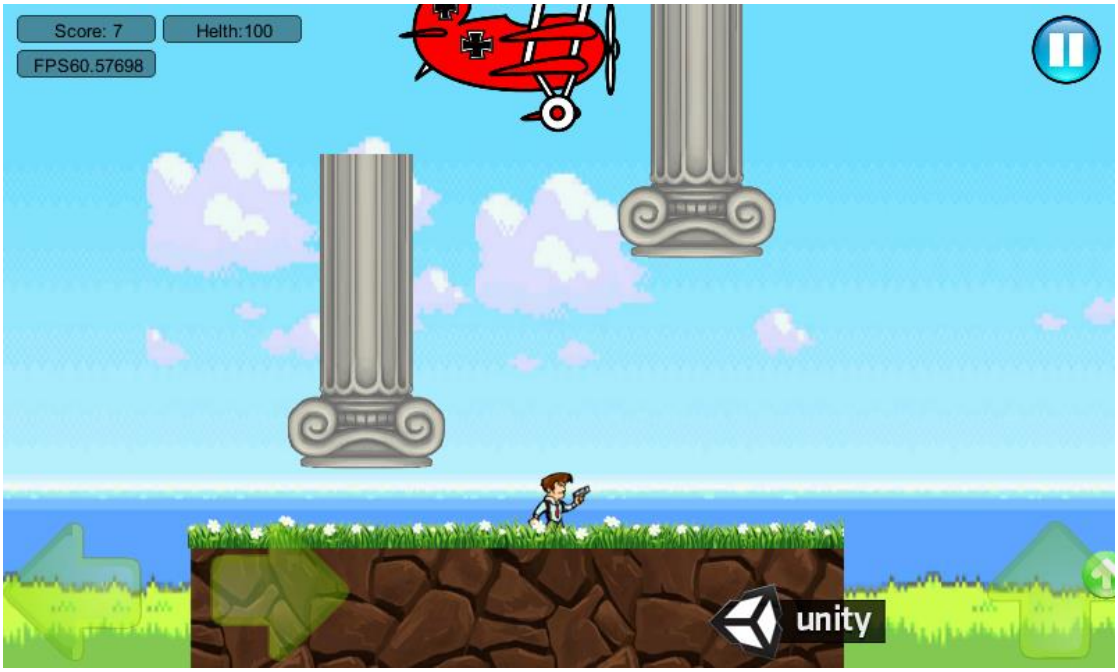
KUVA 18. Heilurin suunnan vaihtaminen

Hinge Joint 2D -komponenttia hyödynnetään myös seuraavassa esteessä, joka on taso jonka päällä oleva kivi putoaa pelimaailmaan. Peliobjekti koostuu kuvassa 19 nähtävistä kolmesta eri Sprite- kuvasta, jotka ovat kivi, taso ja pallo. Pallo- ja tasoobjekteille on annettu Hinge Joint 2D -komponentit ja Rigidbody 2D -komponentit. Pallon ankkuripiste on sijoitettu pelimaailmaan ja taso on liitetty Rigidbody2D -komponentin avulla palloon, jolloin komponenttien välille muodostuu sarana. Pallon Rigidbody2D -komponentti on asetutettu IsKinematic-tilaan, jolloin siihen ei vaikuta fysiikkamoottorin ominaisuudet.



KUVA 19. Tasopeliobjekti ja siihen liitetyt komponentit

Tasopeliobjektiin liitetyssä skriptissä tutkitaan Vector2 distace -metodin avulla milloin pelihahmo on tarpeeksi lähellä tasoa, jolloin tason kulmaa muutetaan 45 astetta, jonka jälkeen kivi putoaa pelikentälle. Kivelle on annettu Circle Collider 2D -komponentti kimmoisalla materiaalilla ja Rigidbody2D -komponentti, jotka saavat kiven pomppimaan. Törmäys kiven kanssa johtaa pelihahmon kuolemaan.

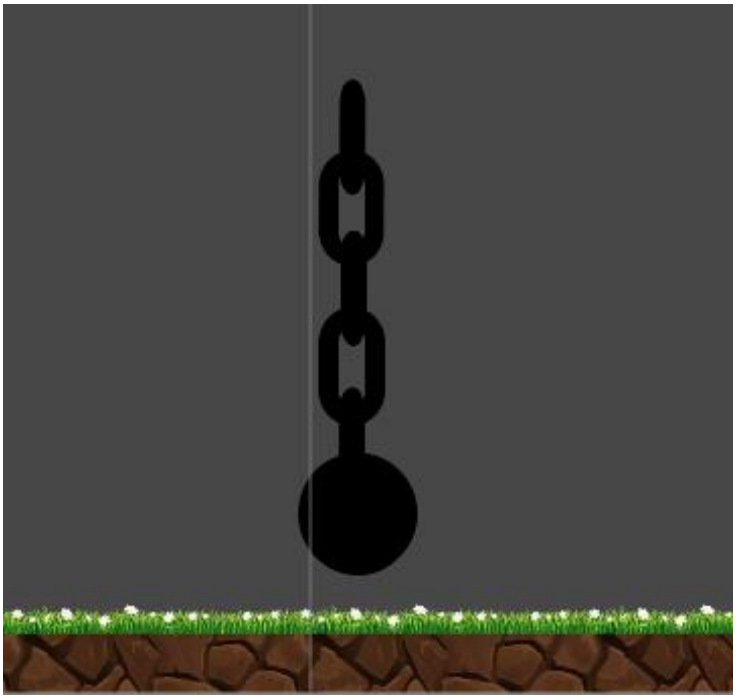


Kuva 20. Pylväät pelinäkymässä

Kuvan 20 näyhtävät liikkuvat pylväät ovat pelin toiseksi viimeisimmät esteet, niiden toteuttamiseen on käytetty Slider Joint 2D -komponenttia. Pylväät liikkuvat sammalla tappaa Y-akselin suuntaisesti, kun pelin ensimmäiset sahaesteet, joiden toteuttamiseen oli käytetty animaatiota. Sahat olivat kuitenkin toteutettu yhden animaation avulla, joten niiden nopeus pysyi kokoajan samana. Pylväitä on pelissä kolme peräkkäin ja halusin saada peliin hieman enemmän haastetta, jonka vuoksi pylväiden liikuttamiseen käytetään fysiikkamoottoria. Pylväisiin liitetyissä skriptitiedostoissa arvotaan Slider Joint 2D -komponentin Motor Speed -ominaisuudelle nopeus 150 ja 300 yksikön väliltä. Näin pylväät saavat lähes varmuudella, joka kerta eri nopeuden. Pylväiden liikkeen rajaamiseen on käytetty Slider Joint 2D -komponentin Translation Limits -ominaisuuksia, joiden avulla liike voidaan rajata Y-akselilla tiettyjen pisteiden välille.

Viimeinen este on ketjuun liitetty murskauspallo joka toimii samalla tapaa kuin heiluri, muuta kiinteän varren sijasta moukari on sijoitettu ketjun päähän. Ketjun luomiseen

on käytetty kahta erilaista Sprite-kuvaa, joita on liitetty toisiinsa Hinge Joint 2D -komponenttien avulla.



KUVA 21. Murskauspallo scene näkymässä

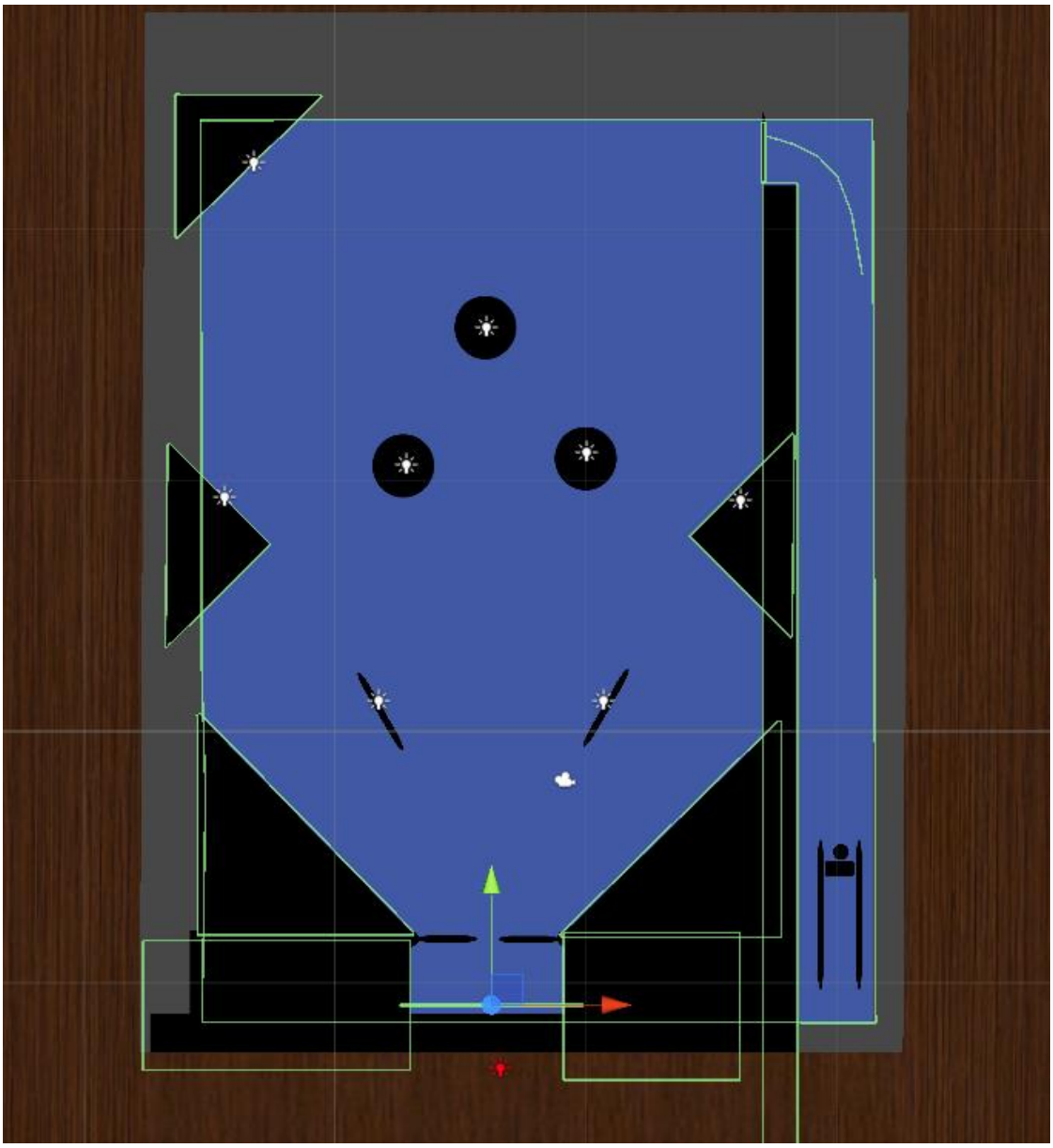
Moukariin on liitetty skriptitiedosto, jonka avulla peliobjektin X- ja Y-koordinaatteja muutetaan siten, että pelin alettua moukari virittyy. Virityksen jälkeen pallo-objekti asetetaan IsKinematic-tilaan, joka saa koko ketjun jäämään ilmaan, koska peliobjektien Rigidbody 2D -komponentit on liitetty toisiinsa. Kun pelihahmon ja Murskauspallon välimatka on tarpeeksi pieni, pallo-objektin IsKinematic-tilaa vaihdetaan ja moukari lähtee liikkeelle. Murskauspallon osuma johtaa pelihahmon kuolemaan.

5.2 Demo 2

Toisena pelidemona tein perinteisen flipperin, joka on idealtaan ja toteutukseltaan hyvin yksinkertainen. Peli koostuu kentästä, pallosta, mailoista ja kentälle sijoitetuista törmäysobjekteista. Pelaajan tehtävä on pitää palloa pelialueella, niin kauan kuin mahdollista. Osumat kentälle sijoitettuihin törmäysobjekteihin kerryttävät pisteitä. Pelialue on kehystetty tasomuodoilla joihin on liitetty puutekstuuri, jonka lisäksi kenttään on liitetty erivärisiä valokomponentteja tuomaan visuaalisuutta. Peli hyödyntää saamaa kamera skrtiptiä mitä tasohyppelydemo, erona kuitenkin se että kamera on asetettu seuraamaan pallo molempien akselien suuntaan.

5.2.1 Kenttä

Kentän pohjana toimii sininen Sprite-kuva, joka on ympäröity Edge Collider 2D -komponentin avulla tehdyllä törmäysalueella. Kentän reunoille on lisätty mustia Sprite-kuvia, jotka on ympäröity Polygon Collider 2D -komponenteilla. Reunoille lisättyjen peliobjektien tehtävä on luoda kenttään muotoja. Pallon laukaisuväylän yläreunaan on sijoitettu tyhjä peliobjekti, johon on lisätty Edge Collider 2D -komponentti, jonka avulla on tehty kaarre, mistä pallo kimpoaa peliin. Laukaisuväylän yläreunaan on tehty luukku, joka toimii Hinge Joint 2D -komponentilla. Luukun kulma on säädetty siten, että luukku aukeaa ainoastaan toiseen suuntaan, sen avulla pallo saadaan laukaistua peliin, mutta pelin aikana pallo ei voi kimmota takaisin laukaisuväylään.

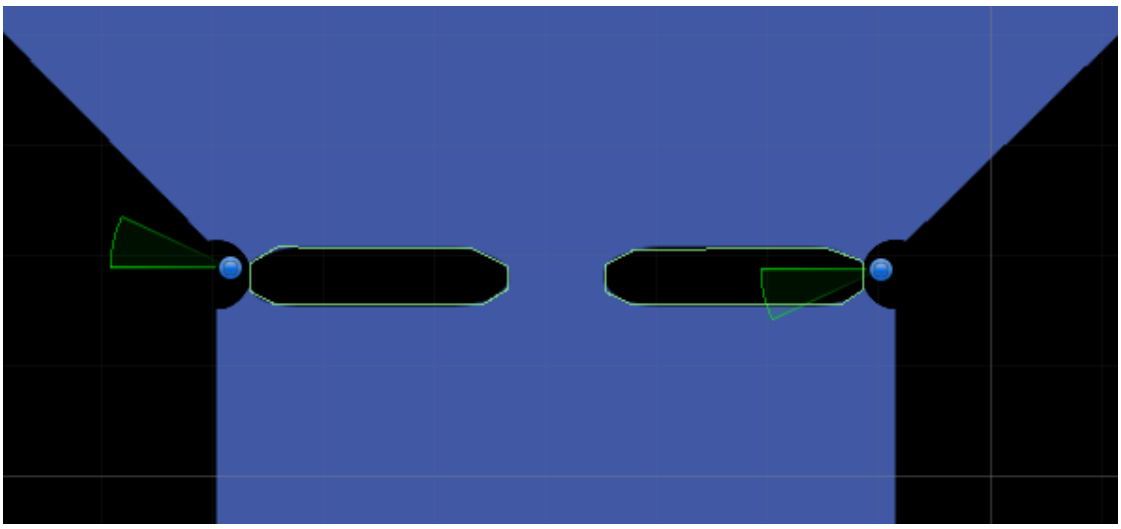


KUVA 22. Pelialueen rajaus Collider 2D -komponentien avulla

Kentälle on sijoitettu kolmenlaisia peiliobjekteja, joihin törmätessään pallolle arvotaan uusi voima, jonka avulla se saadaan kimpoamaan ennalta arvaamattomiin suuntiin. Pelikentän keskelle on sijoitettu kolme ympyrän muotoista törmäysobjektia, jotka sisältävät Circle Collider 2D -komponentit. Peliobjekteihin liitetyissä skriptitiedostoissa tutkitaan OnCollisionEnter2D-metodilla, milloin pallo koskettaa törmäysobjektia. Törmäyksen tapahduttua pallo saa uuden voimavektorin, jonka suunnat arvotaan Random.Range-metodin avulla. Pelikentän kulmista löytyy kolme kolmionmuutoista peliobjektia, jotka toimivat portaaleina. Pallon osuessa portaaliobjektiin, pallon Rigidbody2D -komponentti asetetaan IsKinematic-tilaan, jonka jälkeen palloin paikkaa siirrettään toisen portaalin eteen. Tämän jälkeen pallon IsKinematic poistetaan käytöstä ja pallolle annetaan uusi voimavektori, joka saa sen jatkamaan liikettä. Näin saadaan luotua efekti, joka pelatessa näyttää siltä että pallo teleporttaa törmäysobjektien välillä. Peli loppuu kun pallo pääsee putoamaan mailojen ohitse, loppuminen tutkitaan tyhjään peliobjektiin sijoitetun Box Collider 2D -komponentin OnCollisionEnter2D-ominaisuuden avulla.

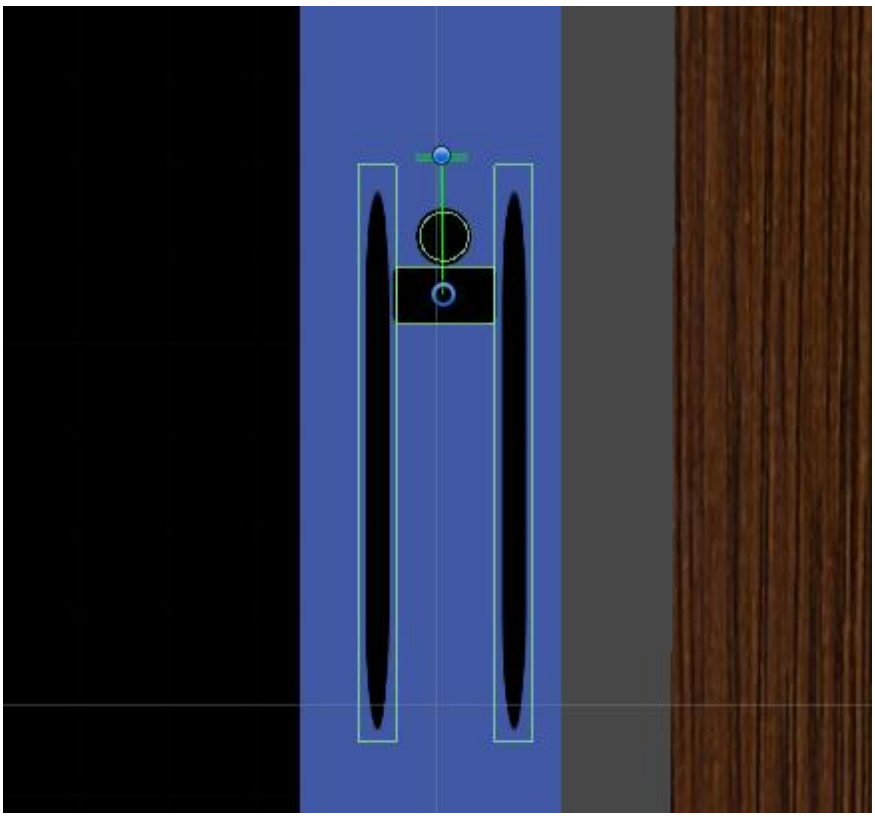
5.2.2 Pallo, mailat ja jousi

Pallon sisältävä peliobjekti on oma Sprite-kuvansa, jolle on annettu Rigidbody 2D- ja Circle Collider 2D -komponentit. Pallon massa on säädetty sopivaksi suhteessa mailojen massaan, sekä törmäyksien yhteydessä lisättäviin voimiin. Pallon Collideriin on liitetty kimmoisa materiaali, joka aktivoidaan pallon peliin laukaisun jälkeen. Pallon Rigidbody 2D -komponentin päivittäminen on asetettu Continuous-tilaan, koska se saa tömäyksien voimasta fysiikkamoottorin päivitysnopeuteen nähden niin suuria nopeuksia, että Discrete-tilan käyttö saattaa jättää törmäyksiä havaitsematta. Kakki palloon liittyvä pelilogiikka, kuten pisteiden lisääminen ja törmäysten tunnistaminen löytyy pallopeliobjektiin liitetystä skriptitiedostosta.



KUVA 23. Mailojen Collider ja Hinge Joint 2D -komponentit

Kuvan 23 mailat ovat pelin toiminnan kannalta tärkeät peliobjektit, niiden avulla pelaajan on tarkoitus pitää palloa pelissä. Molemmat mailaobjektit koostuvat kahdesta Sprite-kuvasta, jotka ovat mailanvarsi ja pallo. Pallo-Spriten ainoa tehtävä on toimia visuaalisena ankkuripisteenä mailalle joten se on asetettu IsKinematic-tilaan, eikä sen tarvitse sisältää minkäänlaista törmäyksen tunnistusta. Maila- ja pallo-osien välille on luotu nivelet Hinge Joint 2D -komponenteilla, ja niille on määriteetetty maksimi- ja minimikulmat joiden välillä ne voivat liikkua. Mailoja liikutetaan Rigidbody 2D -komponenttien avulla, joille skriptitiedostossa lisätään impulssi tyyppinen voima. Tömäyksen tunnistamiseen mailoissa käytetään Polygon Collider 2D -komponenttia.



KUVA 24. Latausjousi

Kuvan 24 latausjousi on koottu kolmesta Sprite-kuvasta, jotka ovat lautasväylän reunat ja neliönmuotoinen palikka, joka toimii jousen iskurina. Iskuriin on lisätty Spring Joint 2D -komponentti, jonka toinen pää on liitetty pelimaailmaan. Iskurin Rigidbody 2D -komponentin massaksi on asetettu 1, lisäksi Fixed Angle on valittu, koska iskurin ei haluta pyörivän akselinsa ympäri. Skriptin avulla tutkitaan koska pallo on kosketuksessa iskurin kanssa, jolloin jousen virittäminen on mahdollista, samaan aikaan pallo siirretään jousen lapsiobjektiksi, jonka jälkeen se seuraa jousen liikettä. Jousen viritys tapahtuu pitämällä välilyöntinäppäintä pohjassa, jolloin sen Transform-komponetin Y-akselin arvoa pienennetään. Samaan aikaan vaihdetaan jousen IsKinematic-tilaa, mikä mahdollistaa jousen venymisen. Jousen saavutettua skriptissä asetettu maksimi pituuden, venyminen pysähtyy. Kun välilyöntinäppäimen painaminen lopetetaan, siirretään pallo takaisin omaksi objektikseen, jonka ansioista se voi liikkua vapaasti. Tämän jälkeen jousi pyrkii automaattisesti palaamaan Spring Joint 2D -komponentille asetettuun lepopituuteen, jolloin se antaa samalla edessään olevalle pienempi Massiselle pallolle voiman, jonka avulla pallo kimpoaa peliin.

6 PÄÄTÄNTÖ

Opinnäytetyön tarkoitus oli perehtyä Unityn tarjoamiin 2D-fysiikkaominaisuuksiin. Työtä tehdessä tutustuin hyvin perusteellisesti Unityn tarjoamaan dokumentaatioon, joista oli paljon apua tätä työtä tehdessä. Hyvän dokumentaation lisäksi Unity suuresta käyttäjäyhteisöstä oli erittäin paljon apua etsiessä ratkaisuja matkan varrella ilmenneisiin ongelmiin. Pelidemot olivat mielestäni onnistuneita ja niiden avulla sain luotua käyttöesimerkit suurimmasta osasta Unityn 2D-fysiikkakomponenteista. Demojen ohjelmointi sisälsi fysiikkamoottoriin liittyvien komponenttien, lisäksi paljon muitakin itselleni aiemmin tuntemattomia ominaisuuksia, joita opin tämän opinnäytetytön teon aikana hyödyntämään. Molemmat aikaansaadut pelidemot tarjoavat myös erinomaisen runkoon jatkokehitystä ajatellen.

Mielestäni Unity pelimoottori tarjoaa erittäin nopean ja helpon tavan saada aikaan näyttäviä toimintoja joiden toteuttaminen vaatisi muutoin huomattavasti suuremman määrän työtä. Jos olisin toteuttanut tämän opinnäytetyön ilman valmista pelimoottorisovellusta, en usko että lopputulokset olisivat olleet lähelläkään tasoa mitä tässä työssä olivat. Unityn fysiikkamoottori tarjosi helpon ja nopean tavan saada aikaan toimintoja, joiden toteuttaminen ilman 2D-fysiikkamoottoria olisi ollut huomattavasti haastavampaa. Aiemmasta Unityn käytöstä oli varmasti apua tätä työtä tehdessä, vaikka varsinaista kokemusta läheskään kaikkien Fysiikkakomponenttien käytöstä ei ollut. Unityn 2D-fysiikkamoottorin keskeisimpänä puutteena mainittakoon pehmeiden kappaleiden fysiikkamallinnuksen puuttuminen, joka Unityn 3D-fysiikkamoottorista löytyy.

Mielestäni pelinkehittäminen pelimoottorien avulla on hyvä tapa aloittaa peliohjelmointi. Tämä tietysti riippuu henkilön taustoista. Jos kyseessä on henkilö, jolla on useiden vuosien kokemus ohjelmoinnista, niin jopa oman yksikertaisen pelimoottorin rakentaminen saattaa olla varteen otettava vaihtoehto. Itse kuitenkin varsi vaatimattomalla ohjelmointi kokemuksella, pidin oman pelimoottorin tekemistä aivan liian haastavan tehtävänä. Perehtymällä ensin valmiisiin tarjolla oleviin ratkaisuihin perusteellisesti ja ymmärtämällä miten komponentit pelimoottorin sisällä toimivat, tarjoaa hyvän pohjan syventyä pelimoottoriohjelmistojen eri osa-alueisiin omina kokokoneisuuksinaan. Esimerkiksi itselleni heräsi työtä tehdessä kiinnostus fysiikkamoottoreiden toimintaperiaatteista, miten jokin fysiikkamoottorin toiminto on toteutettu koditasolla, sekä millaisia laskutoimituksia eri toiminnot oikeasti vaativat. Kuten jo todettu

oman fysiikkamoottorin tai pelimoottorien tekeminen on yhdelle ihmiselle valtava urakka, vaikka tavoitteena ei missään mittakaavassa olisikaan kilpailla suurten ohjelmistotalojen kanssa, jotka saattavat käyttää useita vuosia tuotteidensa kehittämiseen. Lisäksi laadukkaan fysiikkamoottorin toteuttamiseen tarvitaan syvällistä tietoutta fysiikasta ja matematiikasta. Nykyään pelinkehittäjille on tarjolla kymmeniä laadukkaita pelimoottoriohjelmistoja, jotka sisältävät valmiit fysiikkamoottorit. Lisäksi pelimoottorien avulla voidaan kehittää pelejä kaikille suosituille alustoille, oli sitten kyseessä mobiililaite, PC, tai pelikonsoli. Uskon että tulevaisuudessa valmiiden pelimoottorien käyttö kaupallisten pelin teossa tulee kasvamaan, etenkin mobiilipelaamisen ansiosta.

LÄHTEET

Blackman, Sue 2014. Unity for absolute beginners. New York: Apress.

DirectX Graphics and Gaming. Microsoft. WWW-dokumentti. http://msdn.microsoft.com/en-us/library/windows/desktop/ee663274(v=vs.85).aspx. Luettu 13.9.2014

Gregory, Jason 2014. Game Engine Architecture second edition. Massachusetts: A K Peters Ltd.

McShaffry, Mike. Graham, David. Game Coding Complete – forth edition. Boston: Course Technology.

License comparisons. 2014. Unity Technologies. WWW-dokumentti .http://unity3d.com/unity/licenses. Luettu 13.9.2014.

Millington, Ian 2007. Game Physics - Engine Development. San Francisco: Morgan Kaufman Publishers.

Jackson, Simon 2014. Mastering Unity 2D Game Development. Birmingham: Packt Publishing.

Thorn, Alan 2011. Game Engine Implementation. Burlington: Jones & Barlett Learning.

OpenGL Getting started 2014. Getting Started . WWW-dokumentti. http://www.opengl.org/wiki/Getting_Started. Päivitetty 3.7.2014. Luettu 27.0 2014.

Perry, Michael 2009. Torque 2D Development Blog - Box2D Overview. Blogi. http://www.garagegames.com/community/blogs/view/18641. Päivitetty 30.10.2009. Luettu 10.9.2014.

Public relations. 2014. Unity Technologies. WWW-dokumentti. http://unity3d.com/public-relations. Luettu 13.9.2014.

Unity Scripting Api Rigidbody2D. 2014. Unity Technologies. WWW-dokumentti. http://docs.unity3d.com/ScriptReference/Rigidbody.html. Luettu 15.9.2014.

Unity Manual Gameobjects. Unity Technologies. 2014. WWW-dokumentti. http://docs.unity3d.com/Manual/GameObjects.html. Luettu 17.9.2014.

Unity Manual 2D physics Reference Spring Joint 2D. 2014. Unity Technologies. WWW-dokumentti. http://docs.unity3d.com/Manual/class-SpringJoint2D.html. html. Luettu 19.8.2014

Unity Manual 2D physics Reference Hinge Joint 2D. 2014. Unity Technologies. WWW-dokumentti. http://docs.unity3d.com/Manual/class-SpringJoint2D.html. html. Luettu 19.8.2014

Unity Manual 2D physics Reference Wheel Joint 2D. 2014. Unity Technologies. WWW-dokumentti. http://docs.unity3d.com/Manual/class-WheelJoint2D.html. Luettu 19.8.2014

Unity Manual Physics 2D Manager. 2014 . Unity Technologies. WWW-dokumentti. http://docs.unity3d.com/Manual/class-Physics2DManager.html. Luettu 15.9.2014.

Unity Manual Prefabs. 2014. Unity Technologies. WWW-dokumentti. http://docs.unity3d.com/Manual/Prefabs.html. Luettu 15.9.2014.

Unity Manual Scripting Overview. 2014. Unity Technologies. WWW-dokumentti. http://docs.unity3d.com/Manual/CreatingAndUsingScripts.html. Luettu 20.9.2014.

Unity Scripting API. Physics2D.Raycast. 2014. Unity Technologies. WWW-dokumentti. http://docs.unity3d.com/ScriptReference/Physics2D.Raycast.html. Luettu 20.9.201