MATEMATIIKAN HYÖDYNTÄMINEN UNITY-PELIOHJELMOINNISSA

HAMK
HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Ammattikorkeakoulututkinnon opinnäytetyö

Visamäki, tietojenkäsittelyn koulutusohjelma

Syksy, 2018

Pekka Kaski

HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Tietojenkäsittelyn koulutusohjelma
Visamäki

Tekijä Pekka Kaski **Vuosi** 2018

Työn nimi Matematiikan hyödyntäminen Unity-peliohjelmoinnissa

Työn ohjaaja Lasse Seppänen

TIIVISTELMÄ

Tämän opinnäytetyön tavoitteena oli tuoda matematiikan käyttöä esille peliohjelmoinnissa käyttäen Unity-pelimoottoria. Työssä käydään läpi erilaisia matematiikan työkaluja, joita peliohjelmoija voi tarvita Unityssa, kun tavoitteena on koodata pelin pelattavuutta.

Työ on jaettu teoria- ja käytännöntestausosuuteen. Teoriaosiossa käydään läpi peliohjelmointia, jonka jälkeen siirrytään erilaisin matematiikan aiheisiin, kuten vektoriin, trigonometriaan, matriisiin ja kvaternioon. Testausosiossa testataan erilaisia matematiikan työkaluja Unityssa, joita käytiin läpi teoriaosioissa.

Työn aikana testattiin erilaisia tapoja hyödyntää matematiikkaa Unity projektissa. Haasteena oli teorian varmistaminen, mutta internetistä löydetyistä materiaaleista oli paljon apua.

Avainsanat Peliohjelmointi, Unity, lineaarialgebra, trigonometria, kvaternio

Sivut 28 sivua

HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Business Information Technology
Visamäki

Author Pekka Kaski **Year** 2018

Subject Exploitation of math in Unity game programming

Supervisors Lasse Seppänen

ABSTRACT

The goal of this bachelor's thesis was to introduce how different math tools can be utilized in the gameplay programming within the Unity game engine. The work presents diverse math tools for coding the gameplay functionalities in the Unity environment.

The work is divided into theory and functionality testing sections. The theory section first reviews the gameplay programming in general, and then examines the different math tools, including vectors, matrices, trigonometry and quaternion. In the functionality testing section, these tools will be tested in the Unity environment.

During this thesis, various methods to utilize mathematics in the Unity project were tested. A theoretical problem was born from verifying the different math tools. This problem was solved by deriving material from the internet which enabled further proceeding.

Keywords Unity, game programming, linear algebra, quaternion, trigonometry

Pages 28 pages

SISÄLLYS

SANASTO

API	Koodikirjasto, joka sisältää valmiiksi luotuja funktioita ja metodeja
Aritmetiikka	Luvuilla laskeminen, kuten yhteen-, vähennys-, kerto- ja jakolasku
Asset	Ladattava paketti, joka voi sisältää esim. 3D-malleja, musiikkia tai koodia
Pelimoottori	Ohjelmistokehys, jonka päälle rakennetaan pelejä
Peliobjekti	Objekti, jolla on komponentteja kytkettynä kiinni
Prefab	Valmiiksi luotu peliobjekti, joka sisältää tallennettuja komponentteja
Scene	Unityssa näkymä, jossa peliobjektit tekevät toimintojaan
Skripti	Komponentti, jolla pystytään ohjelmoimalla manipuloimaan muita komponentteja koodin kautta
Unity	Pelimoottori ja pelinkehitysalusta

1 JOHDANTO

Peliohjelmointia yhdistetään yleensä tiettyihin matematiikan alueisiin, kuten 3D-matematiikkaan. Internetissä, kuten *Game Development Stackexchange*-sivustossa, moni sanoo erilaisia kriteerejä, joita peliohjelmoijan tulisi osata, kuten esimerkiksi mitä matematiikkaa pitäisi hallita, jos haluaa ryhtyä ohjelmoimaan pelattavuutta tai olla tekemisessä fysiikoiden kanssa. Lisäksi työpaikkasivuista voi työantajat etsiä työnhakijaa, jolla on osaamista esimerkiksi lineaarialgebrasta, joka on yksi matematiikan työkaluista, joita käytetään pelialalla. Tämän opinnäytetyön tarkoituksena on käydä läpi yleisempiä matematiikan alueita, joita Unitya käyttävä peliohjelmoija voi tarvita, kun hän koodaa pelin pelattavuutta. Toivon, että työni auttaa peliohjelmoijia soveltamaan matematiikkaa tilanteen mukaan.

Valitsin Unity-pelimoottorin, koska se on entuudestaan minulle tuttu ohjelma, ja tulevaisuudessa teen töitä Unityn parissa. Unity-pelimoottorilla voi tehdä 2D- tai 3D-pelejä, animaatioita ja muita 3D-näytöksiä.

Opinnäytetyön toisessa luvussa käydään läpi mitä peliohjelmointi on, mikä on Unity ja miten peliohjelmointi tapahtuu sillä. Kolmannessa luvussa käydään erilaisia matematiikan työkaluja peliohjelmoinnissa, kuten vektoria, matriisia, trigonometriaa ja kvaterniota. Teoriaosuuden jälkeen neljännessä luvussa otetaan Unity-ympäristö käyttöön ja katsotaan, miten se toimii. Viidennessä luvussa tehdään 3D-esimerkkipeli, jossa testataan matematiikan työkaluja Unityssa, joita opittiin teoria osuudessa.

Tutkimuskysymyksiäni ovat: mitä matematiikan työkaluja tarvitaan Unity-peliohjelmoinnissa? Miten Unity-peliohjelmoinnissa hyödynnetään lineaarialgebraa? Jotta opinnäytetyöni tarkoitusperä toteutuisi, oletan lukijan osaavan C#-ohjelmointikielen perusteita.

2 PELIOHJELMOINTI UNITYLLA

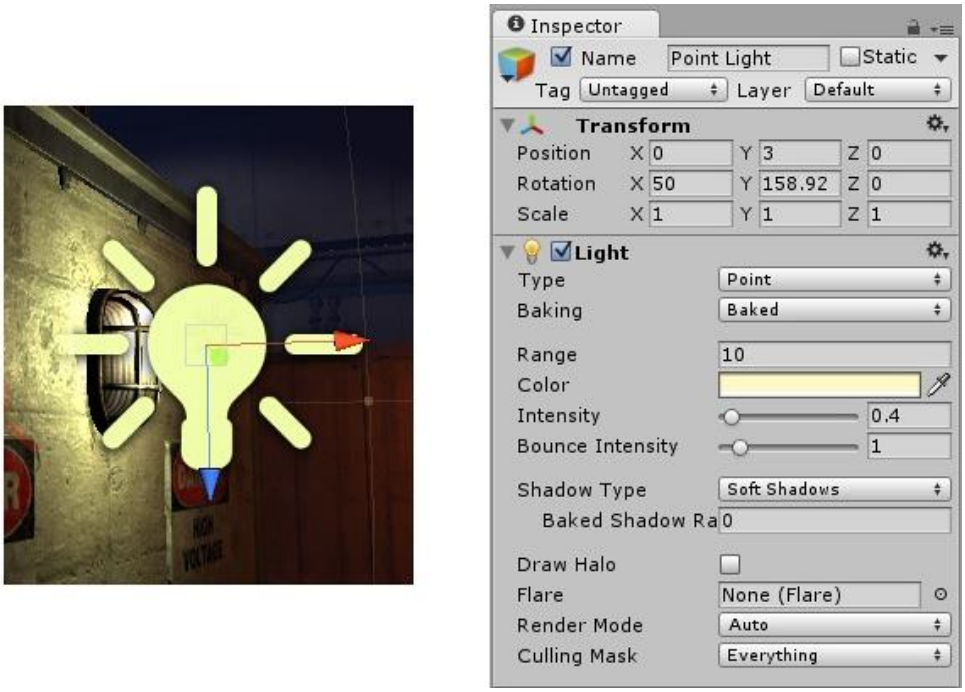
Peliohjelmointi on pelien kehittämisen ydinkohta. Siinä kehitetään pelin ohjaamista, kuten hahmojen liikkumista. Peliohjelmoinnissa on käytössä erilaisia koodikirjastoja eli API, joilla pystytään käyttämään valmiiksi luotuja funktioita. API voi sisältää esimerkiksi fysiikoista vektoreihin, joilla pystytään manipuloimaan objekteja. Peliohjelmoinnissa käytetään valittua ohjelmointikieltä ja alustaa, jossa työskennellään. Alusta voi olla esimerkiksi Unity, Unreal Engine, Game Maker tai omanlainen alusta, käyttäen OpenGL-ohjelmointi rajapintaa ja C++-ohjelmointikieltä (Unity Technologies, n.d.).

Peliohjelmointi työnä opettaa uusia asioita jatkuvasti, ja haasteena ovat muuttuvat ohjelmointikielet, ohjelmistot ja menetelmät. Ohjelmoijalta odotetaan hyvää ongelmanratkaisukykyä ja tietynlaista matematiikan osaamista, kuten 3D-ohjelmoijan olisi hyvä osata 3D-matematiikkaa, joita ovat mm. vektorit ja matriisit. Peliohjelmoijan roolia voidaan myös jakaa osiin, riippuen yrityksestä ja työntekijöiden määrästä, esimerkiksi yksi työntekijä voi keskittyä UI-ohjelmointiin eli user interface, kun toinen ohjelmoi pelattavuutta. (McShaffry & Graham, 2013, s. 47-48.)

Unity on pelikehitysalusta ja pelimoottori, joka on ilmainen, mutta sisältää myös maksullisia ominaisuuksia. Pelimoottori on ohjelma, josta löytyy valmiiksi työvälineitä kehittämään pelejä. Unitylla voi kehittää 2D- ja 3D-pelejä. Pelejä voi julkaista muun muassa PC:lle, Mac:lle, älypuhelimelle ja konsoleille. (Unity Technologies, n.d)

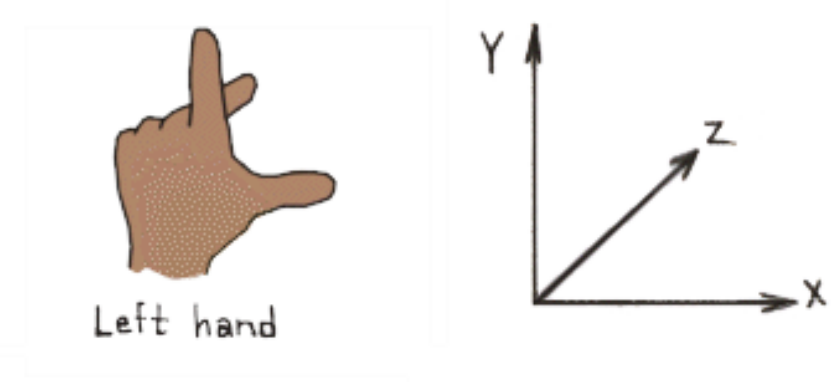
Unity käyttää C#-ohjelmointikielenä, joka on olioperustaista ohjelmointia. Vanhemmissa Unity versioissa vaihtoehtona on UnityScript, joka muistuttaa JavaScriptiä. Unitylla voi kehittää 2D- ja 3D-pelejä. Pelejä voi julkaista muun muassa PC:lle, Mac:lle, älypuhelimelle ja konsoleille (Unity Technologies, n.d.)

Unityssa käytetään peliobjekteja. Peliobjekti on olio, joka sisältää komponentteja, kuten fysiikkaa ja törmäyksen tunnistimia. Peliobjekteja pystytään kontrolloimaan skripteillä. Skriptit eli luodut koodit ovat myös komponentteja, joilla voi manipuloida muita komponentteja, kuten esim. jos vihollinen törmää pelaajaan, niin pelaaja -peliobjekti tuhoutuu. Peliobjektit näkyvät scenessä, jossa objektit tekevät ohjelmoituja asioita, kuten liikkumista. Peliobjekteja voi tallentaa myöhempää käyttöä varten, joista tulee prefabeja. Prefab on valmiiksi tehty peliobjekti, jolla pystytään esimerkiksi kesken pelin kutsumaan muita peliobjekteja paikalle, kuten pyssyn luoteja. Jos scenessä on saman prefabin useita peliobjekteja, muutoksia pystytään tallentamaan jokaiseen yhtä aikaa. (Unity Technologies, n.d.)



Kuva 1. Kuva peliobjektista, johon on transform- ja light-komponentit kytkettynä. (Unity Technologies, n.d.)

Unity käyttää vasemman käden koordinaatistojärjestelmää. Peukalo edustaa vaakasuuntaa (x), etusormi pystysuuntaa (y) ja keskisormi syvyyttä (z). Koordinaatistojärjestelmää käytetään Transform-komponenttia esittämään objektien sijaintia ja kääntymistä. Olioiden sijaintia voi muokata suoraan Unityssa tai skripteillä. (Unity Technologies, n.d.)



Kuva 2. Kuva vasemman käden koordinaatistosysteemistä (Scichart, n.d.)

3 MATEMATIIKKAA PELIOHJELMOINNISSA

Matematiikka on osa peliohjelmointia ja Ilman sitä, ei saada ohjelmoitua hahmoa tekemään yksinkertaisintakaan asiaa, kuten liikkumista. Matematiikan tarve ohjelmoinnissa riippuu täysin siitä, mikä on kyseisen henkilön rooli projektissa. Peliohjelmointiin voi liittyä mm. algebraa, trigonometriaa, laskentaa ja lineaarialgebraa. (Game Designing, n.d)

Lineaarialgebra on yleisesti laajempi aihealue, mutta peliohjelmoinnissa on otettu ne tärkeimmät osat huomioon, joista on hyötyä, kuten vektorit ja matriisit. Näillä pystytään tallentamaan objektien sijaintia, lisäämään liikkumisnopeutta, kääntämään haluttuun suuntaan ja yms. Madhavin (2013) mukaan ei pystytä liioittelemaan vektorien ja matriisien tarvetta peliohjelmoinnissa.

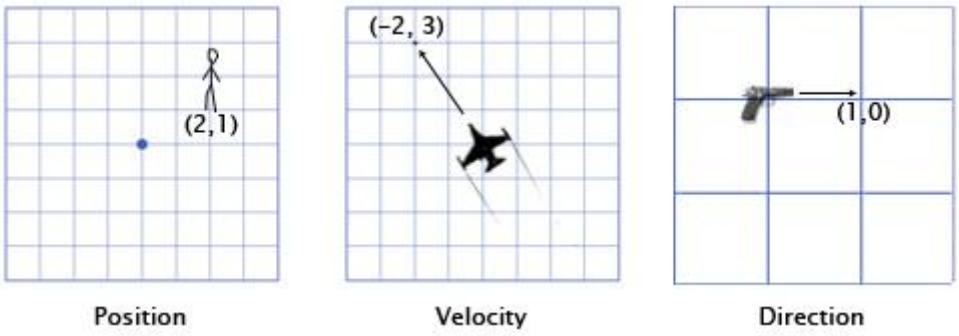
Matematiikkaa ei käytetä ainoastaan ohjelmoimaan objekteja, vaan koko pelin visuaalinen näkymä on myös toteutettu matematiikalla. Matematiikkaa käytetään myös mm. simuloimaan veden liikkumista meressä, animaatioihin, algoritmeihin, analytiikkaan, tekoälyihin ja varjostimiin. (Forbes, 2016)

Matematiikkaa voi hyödyntää käyttämällä API-kirjastoja, jotka tarjoavat matemaattisia funktioita laskemista varten. Unityn mukana tulee kirjastoja, joita voidaan käyttää laskemiseen apuna, kuten *Mathf*-luokka. *Mathf*-luokka sisältää yli 40 metodia, joilla pystytään mm. manipuloimaan muuttujien arvoja tilanteen mukaan (Unity Technologies, n.d.).

3.1 Vektorit

Vektori, $\vec{v}$, on suure, joka edustaa pituutta ja suuntaa n-ulottuvuudessa ja sillä on yksi reaaliluku komponentti per ulottuvuus. Peleissä vektori on yleensä 2D tai 3D, riippuen mitä ulottuvuutta peli tukee. Joissakin 3D-peleissä käytetään 4D-vektoreita. (Madhav, 2013)

Peliohjelmoinnissa käytetään vektoria, jolla on samat kirjaimet, kuin koordinaatiston: x, y ja z. Komponentti x kuvastaa vaakasuuntaa, y pystysuoraa ja z syvyyttä. Vektoreilla pystytään kuvailemaan olion sijaintia, nopeutta ja suuntaa (Rosen, 2009).



Kuva 3. Vektorien esimerkki käyttö 2D-peleissä. (Rosen, 2009)

Muutamia perus operaatioita voi tehdä vektoreilla, joissa tarvitaan aritmetiikan osaamista, mutta on paljon hyödyllisempää osata geometristä implikaatioita, joilla pystytään ratkaisemaan ongelmia pelimaailmassa (Madhav, 2013).

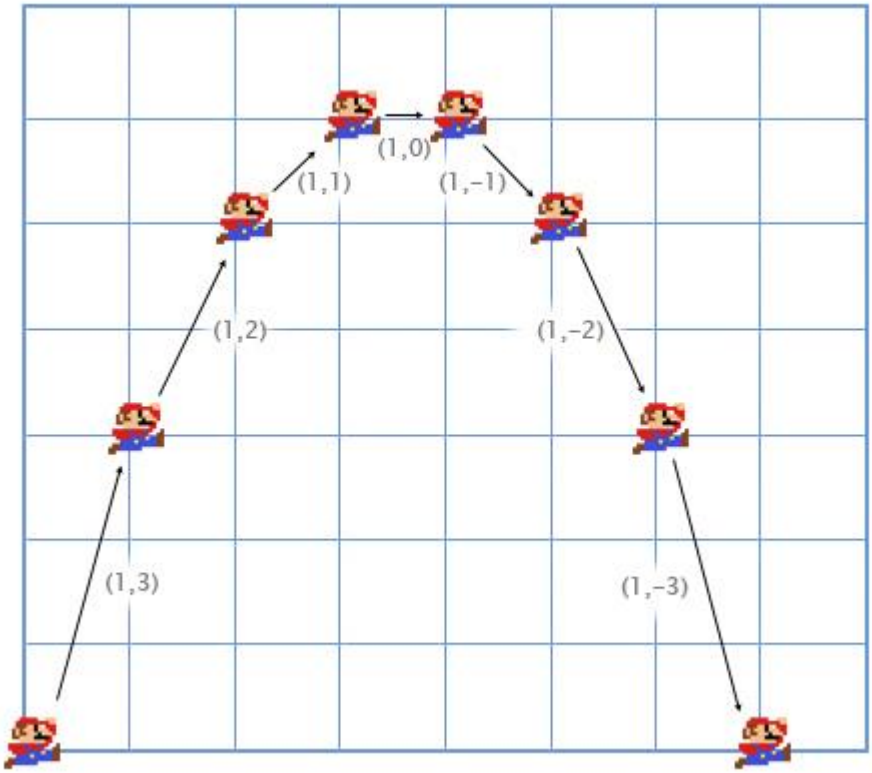
3.1.1 Yhteen- ja vähennyslasku

Yhteenlaskussa vektorien komponentit lasketaan yhteen. Samat komponentit laskevat keskenään, eli esimerkiksi kahden vektorin x-komponentit laskevat keskenään, ja sitten y ja z keskenään. Vähennyslaskussa komponentit vähentävät toisiaan keskenään. Madhavin (2013) mukaan vektorien yhteen- ja vähennyslaskuissa käy vaihdantalaki, kuin peruslaskennassa:

$$\vec{a}+\vec{b}=\vec{b}+\vec{a}$$

Yhteen- tai vähennyslaskua voi käyttää esimerkiksi sellaiseen tilanteeseen, jossa pelissä nuoli osoittaa pelaajalle mihin pitää seuraavaksi mennä. Nuolta voi päivittää suuntansa vähentämällä kohteen ja pelaajan sijaintia keskenään. (Madhav, 2013)

Toinen esimerkki käyttö on hahmon hyppääminen. 2D-hahmo hyppää eteenpäin (1, 3) – nopeasti ylöspäin, mutta myös oikealle. Painovoima vetää hahmon alaspäin (0, -1). Seuraavassa kuvassa näkyy hahmon sijainti, kun hän hyppää ja laskeutuu alas (Rosen, 2009).



Kuva 4. Kuva hyppäävästä hahmosta. (Rosen, 2009)

3.1.2 Skalaari

Vektoria voi kerrata tai jakaa käyttäen skalaareja. Skalaari on desimaali- tai täysiluku, joka muuttaa vektorin pituutta. Vektorin suunta voi pysyä samana. Vektorin jokaista komponenttia jaetaan tai kerrataan skalaarilla. Skalaari voi olla esimerkiksi hahmon liikkumisnopeus, jota lisätään liikkumisnäppäimiin. Skalaaria käyttäessä vektorin suunta pysyy samana, mutta pituus muuttuu. (Madhav, 2013)

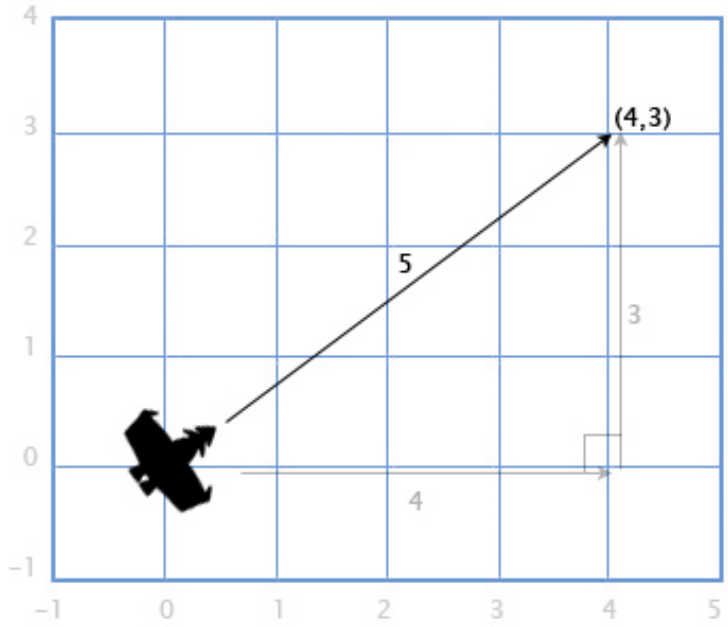
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Esimerkki : MonoBehaviour
{
    // Player controls for 2D game;
    Vector2 input;

    // Movement speed for Player
    public float movementSpeed = 3f;

    // Update is called once per frame
    void Update()
    {
        // Get Inputs from keyboard and joystick
        input.x = Input.GetAxis("Horizontal");

        // Apply movementSpeed to input
        var movement = input * movementSpeed;
    }
}
```

Kuva 5. Kuvakaappaus Visual Studiosta, jossa koodataan pelaajan liikkumista. Koodissa käytetään vektoria ja skalaaria.

3.1.3 Pituus

Vektorin pituus, |V|, kuvaa vektorin pituutta desimaalilukuna. Pituutta saadaan selville käyttäen vektorin metodia *magnitude* tai *sqrMagnitude*, joka muuttaa vektorin täysi- tai desimaaliluvuksi. Vektorin pituudella saadaan selville esimerkiksi peliobjektin nopeus (Rosen, 2009).

Vektorin pituutta voi käyttää esimerkiksi hahmon animaatioon tai auto pelissä, että saadaan polttoainetta kulutettua sen mukaan, kuinka nopeasti auto liikkuu (Rosen, 2009).

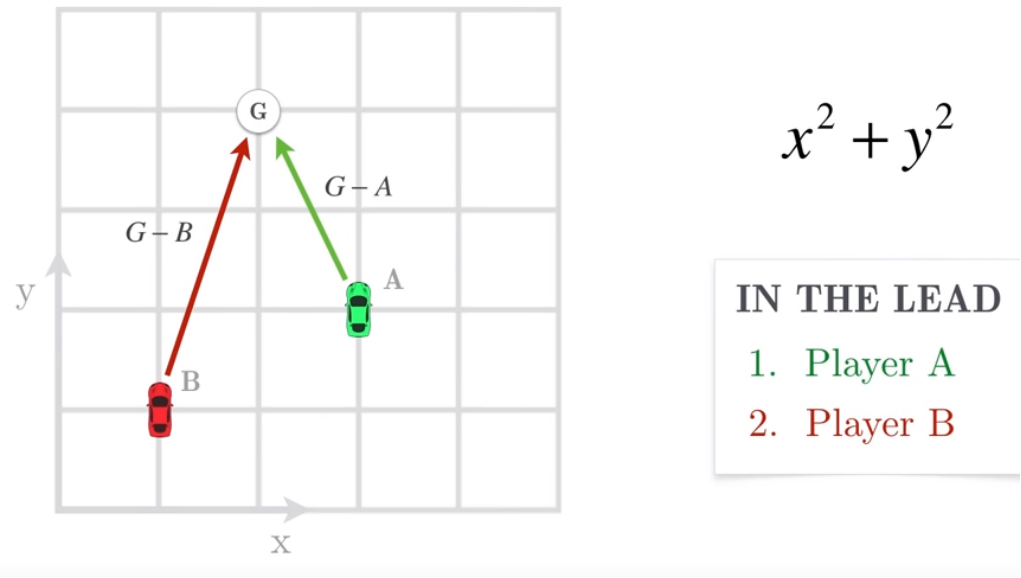


Kuva 6. Esimerkki kuva vektorin pituudesta. (Rosen, 2009)

3.1.4 Etäisyys

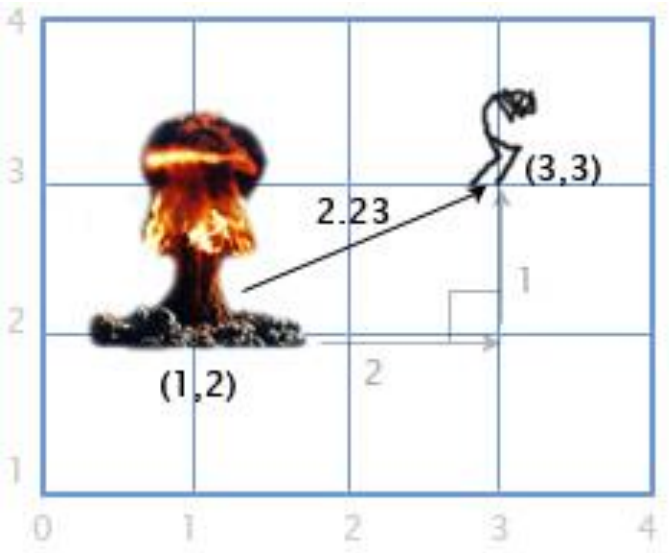
Kahden objektin etäisyys saadaan selville vähentämällä kahden objektien sijaintien komponentit (x, y, z), josta tulee uusi vektori ja siitä otetaan vektorin pituus (magnitude). Lisäksi löytyy myös *Vector3.Distance*-metodi, joka laskee kaiken valmiiksi. On kuitenkin tilanteita, joissa ei kannata käyttää turhaan neliöjuurta tai Vector3.Distance-metodia, sillä se voi syödä liikaa resursseja, kun pitää laskea jatkuvasti etäisyyksiä (Thirslund, 2016).

Esimerkiksi autopeleissä, jossa halutaan tietää vain, kuka on ensimmäisellä sijalla auto radalla. Silloin ei tarvita tarkkaa etäisyyden laskemista maalipisteestä. Maalin ja auton vektorit komponentit vähentävät toisiaan, ja tuloksen vektorista komponentit laskevat yhteen potenssi toiseen. (Thirslund, 2016)



Kuva 7. Kuva kahdesta autosta (A ja B), jotka ajavat kilpaa maaliin (G). (Thirslund, 2016)

Toinen esimerkkikäyttö on vihollisen hyökkääminen pelaajan kimppuun, kun tämä on tarpeeksi lähellä kohdetta. Kolmas esimerkki käyttö on räjähdys, joka tekee enemmän vahinkoa objekteihin, jotka ovat lähempänä räjähdystä (Rosen, 2009).



Kuva 8. Kuva räjähdyksestä ja hahmosta, joka juoksee pakoon. Räjähdyksen ja hahmon etäisyyttä lasketaan. (Rosen, 2009)

3.1.5 Normalisointi

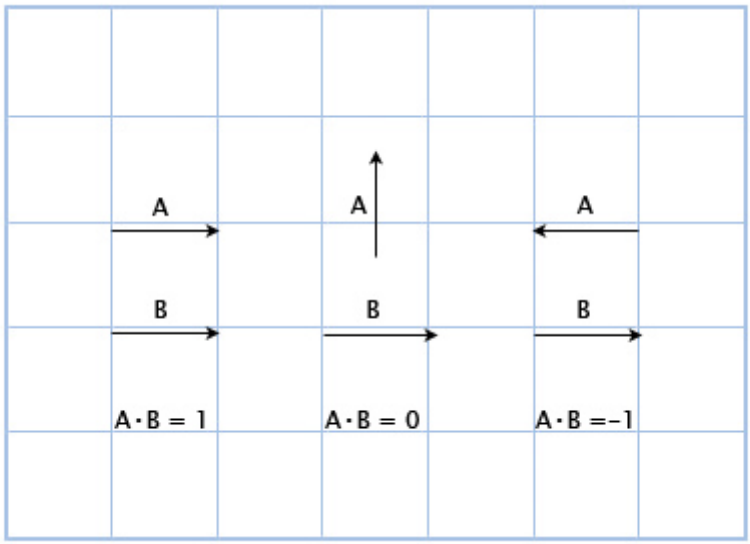
Vektoria voi normalisoida, eli vektori muutetaan yksikkövektoriksi. Vektorin suunta pysyy samana, mutta sen pituus on 1. Normalisointi tapahtuu käyttäen vektorin metodia *normalized* tai *Normalize*(). Normalisointia voi

käyttää sellaisiin tilanteisiin, jossa peliobjektin suunta pitää ottaa talteen. Nyrkkisääntö normalisointiin on se, että käytä silloin, kun haluat käyttää vektorin suuntaa, kun vektorin pituudella ole väliä (Madhav, 2013).

3.1.6 Pistetulo

Pistetulo, V • V, tuottaa skalaarin kahdesta vektorista. Metodi on Vector3.Dot(V^1, V^2) tai Quaternion.Dot(V^1, V^2), jossa molemmat palauttavat liukuluvun. Pistetuloa voi käyttää 2D- tai 3D-peliohjelmoinnissa (Madhav, 2013).

Käyttäessä *Vector3.Dot*-metodia voidaan laittaa parametreiksi kahden objektin *Vector3.forward*, joka on lyhenne objektin z suunnasta (0, 0, 1), johon objekti katsoo. Tuloksella pystytään päättelemään kahden objektin suuntien eroavaisuuksia. Jos tulos on 0, objektit katsovat kohtisuoraan toisiaan. Jos tulos on positiivinen, objektien katsovat samaan suuntaan. Jos tulos on negatiivinen, objektit katsovat vastakkaiseen suuntaan. (Rosen, 2009)



Kuva 9. Kuva kahden vektorien suunnasta. Pistetulo näistä kahdesta vektorista kertoo ovatko ne samassa vai eri suunnassa. (Rosen, 2009).

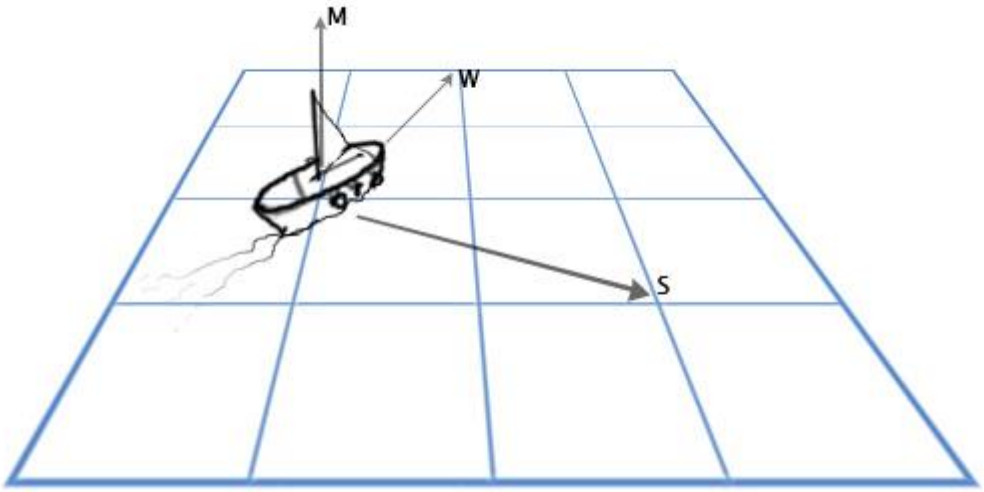
Toinen pistetulon esimerkki käyttö on etsiessä kahden vektorin kulmaa. Pistetulon tuloksesta käytetään trigonometrista formulaa, kuten *Mathf.Acos*. (Madhav, 2013)

```
// Find angle between two vectors
float FindAngle(Vector3 A, Vector3 B)
{
    // Dot product of two vectors
    float dot = Vector3.Dot(A, B);

    // Divide the dot by the product of the magnitudes of vectors
    dot = dot / (A.magnitude * B.magnitude);

    // Get arc cosine angle ( turns into radians )
    var acos = Mathf.Acos(dot);

    // Turn radians into angles
    var angle = acos * Mathf.Rad2Deg - 90f;

    // Angle of two vectors
    return angle;
}
```

Kuva 10. Kuvakaappaus kahden vektorien kulman laskemisesta.

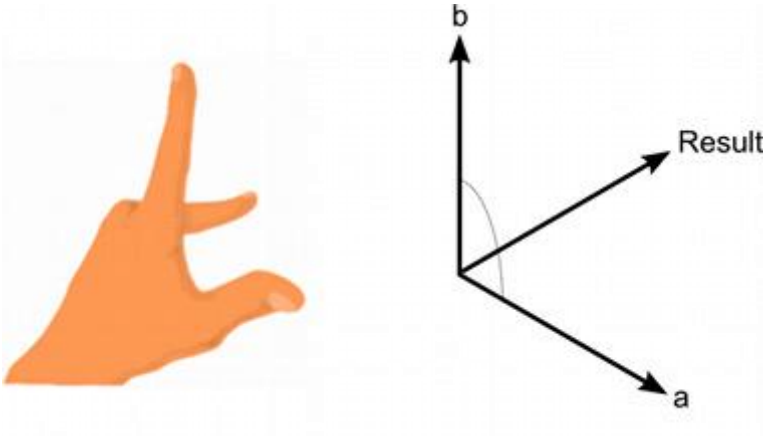
3.1.7 Ristitulo

Ristitulo, V x V, ottaa kaksi vektoria vastaan ja tuottaa niistä kolmannen vektorin, joka on kohtisuorassa kahteen vektoriin. Esimerkki-tilanne, on vene, joka seilailee meressä. Maston (M) suunta on (0, 1, 0) ja tuulen (W) nopeus on (1, 0, 2). Halutaan löytää purjeen (S) paras suunta, jotta vene saa lisää nopeutta tuulesta. Ristitulolla (S = M x W) saadaan tulos, johon purjeen pitää kääntyä. (Rosen, 2009)



Kuva 11. Kuva ristitulo -metodin käytöstä (Rosen, 2009)

Ristitulolla on nyrkkisääntö; kädellä voi tehdä vasemman käden koordinaatiston niin, että keskisormi osoittaa eteenpäin, etusormi ylös ja peukalo oikealle. Keskisormi on ristitulon tulos, ja etusormi ja peukalo ovat kaksi vektoria, joita käytetään parametrinä (Unity Technologies, n.d.).



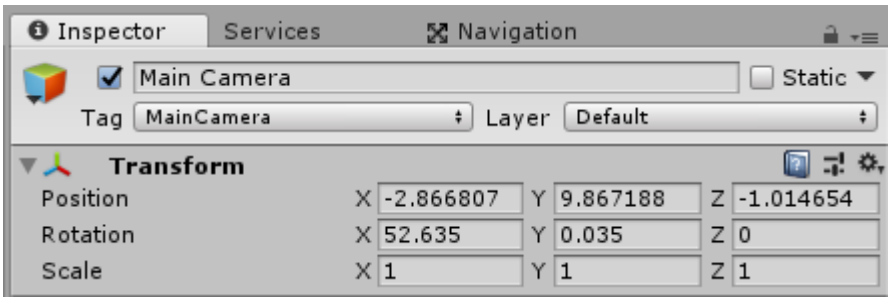
Kuva 12. Kuva ristitulosta, kun se tehdään vasemman käden koordinaatiston mukaisesti. (Unity Technologies, n.d.)

Ristituloa voi käyttää myös etsiessä normaalia vektoria. Normaali vektori pintaa kohti on vektori, joka on kohtisuorassa pintaa vasten annetussa pisteessä. (Weisstein, n.d.)

3.2 **Matriisit**

Matriisi on taulukko luvuista, jossa on m-määrä vaakariviä ja n-määrä pystyriviä. Peleissä käytetään yleensä 3x3 ja 4x4 matriiseja. (Madhav, 2013)

Lineaarialgebraa käyttää matriiseja laskemaan sijaintia ja liikkeitä, joita kutsutaan transformoinniksi. Transformointi koostuu kolmesta tyypistä: translaatio, rotaatio ja skaalaus. Jokaista transformointia voi esittää matriisina, jota kutsutaan *transformointi matriisi*ksi. (Quora, n.d.)



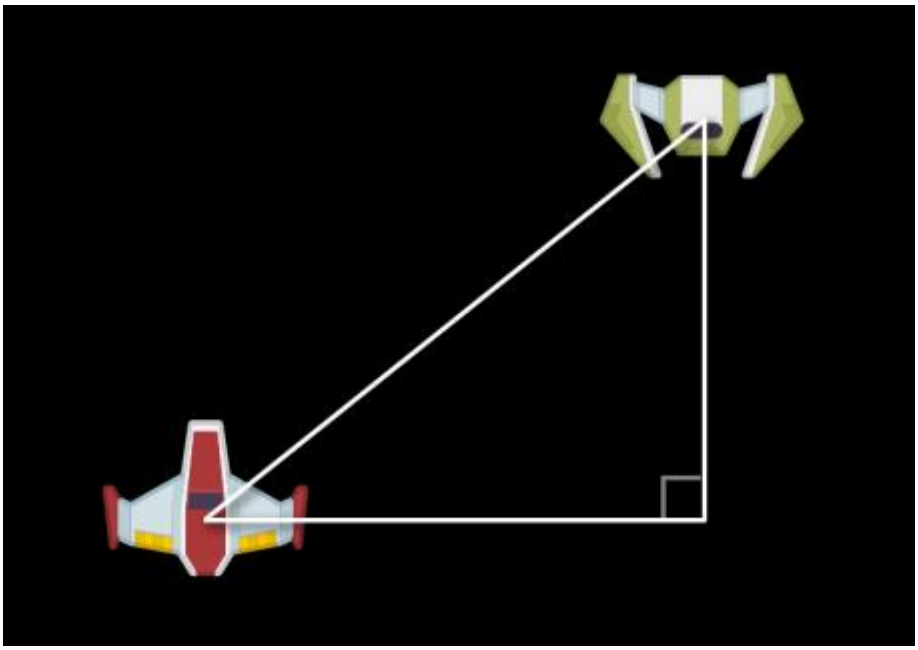
Kuva 13. Kuvakaappaus kameran transformoinnista Unityssa.

3.3 **Trigonometria**

Trigonometrialla saadaan selville muun muassa etäisyyksiä ja kulmia. Se on matematiikkaa ympyröistä ja suorakulmaisista kolmioista. Trigonometriaa käytetään pääsääntöisesti 2D-peleissä, mutta myös 3D-peleissä (Hollemans, 2013).

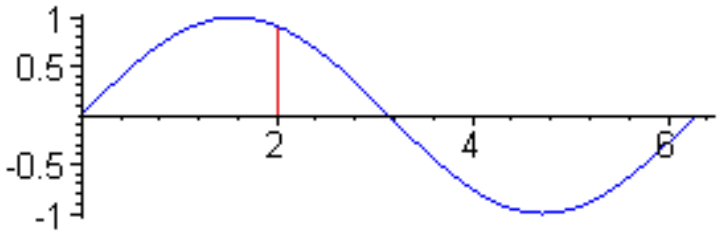
Olioiden kulmien hyödyntäminen ohjelmoinnissa ei ole kuitenkaan niin yksinkertaista, vaan se vaatii myös hieman säätöä. Normaalisti käytetään asteita arvosta 0 – 360, mutta Unityssa käytetään radiaaneja arvosta 0 - 2π. *Mathf*-trigonometristen funktiot palauttavat arvot radiaanina, mutta jotta olio kääntyisi oikeaan suuntaan, pitää muuttaa radiaanit asteiksi. Tätä varten käytetään kertolaskussa *Mathf.Rad2Deg* -muuttujaa ja vähennetään tulosta 90 asteella, sillä peliobjektin kierto pitäisi olla oletusarvoltaan 0. (Hollemans 2013)

Pythagoraan lauseella saadaan selville kahden olion etäisyyksiä. Tarvitsemme tätä varten kahden olion sijainnin x, y ja z arvot, jotka vähentävät toisten vektorin komponenttien arvoja. Lopuksi lisätään potenssi toiseen ja käytetään neliöjuurta. (Hollemans, 2013)



Kuva 14. Kahden avaruusaluksien etäisyyttä saadaan selville Pythagoraan lauseella (Hollemans 2013).

Trigonometriassa on myös aaltoja, joista on hyötyä pelimaailmassa. Aalloilla, kuten siniaallolla, pystytään liikuttamaan objektia ylös ja alas aallon mukaisesti. Siniaallon arvo aaltoilee annetun syötteen mukaisesti 1:sta -1:seen. Syöte voi olla esimerkiksi hahmon liikkumisnopeus tai aika, koska molempien arvot muuttuvat koko ajan (Fletcher, n.d.).



Kuva 15. Kuva siniaallosta, joka menee ylös ja alas loputtomiin, vaikka syötteen arvot nousevat. (Fletcher, n.d.)

2D-pelimaailmassa objektilla ei ole suuntaa, johon objekti katsoo. Siispä on ohjelmoijasta kiinni ohjelmoida muuttuja, joka tallentaa objektin suunnan. Tangentilla voidaan käyttää selvittämään 2D-objektin suuntaa käyttäen *Mathf.Tan*-metodia. Vastakkaisen- ja viereisen sivut ovat objektin x ja y nopeus. Nyt kun tiedetään objektin suunnan, voidaan päivittää esimerkiksi objektin 2D-kuvaa vastaamaan sitä suuntaa, tai jos pelaaja osaa ampua aseella, niin nyt luodit osaavat lentää oikeaan suuntaan. (Leeuwen, 2010)

3.4 **Kvaternio ja Eulerin kulmat**

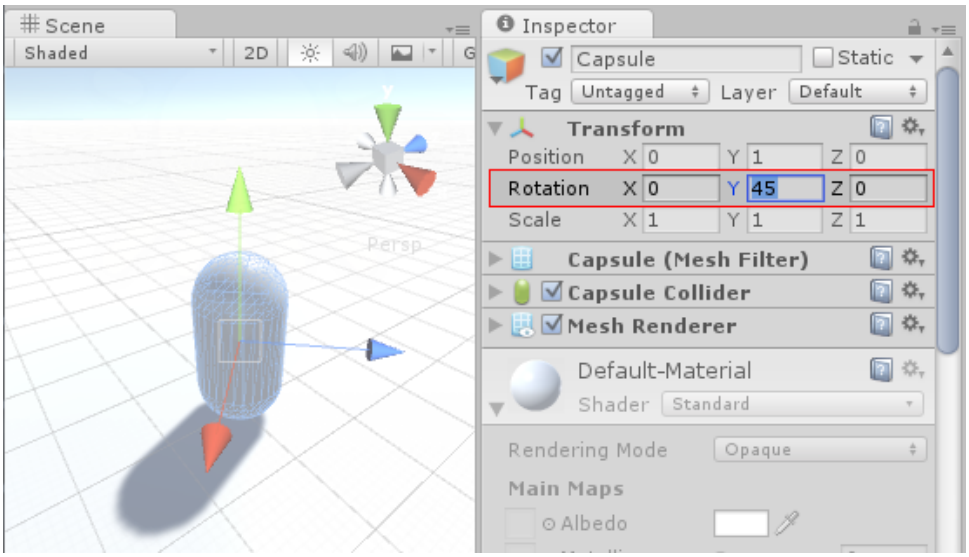
Kvaternio ja Euler kuvastavat objektin rotaatiota. Molempia käytetään kääntämään ja pyörittämään objektia, ja molemmat tekevät yhteistyötä keskenään Unityssa. Unity tarjoaa valmiiksi luotuja metodeja, joita voidaan käyttää skripteissä kääntämään peliobjektia haluttuun suuntaan (Unity Technologies, n.d.).

Taulukko 1. Valikoidut metodit Unityn-sivuilta

Metodin nimi	Kuvaus
LookRotation	Creates a rotation with the specified forward and upwards directions
AngleAxis	Creates a rotation which rotates angle degrees around axis
FromToRotation	Creates rotation which rotates from fromDirection to toDirection
Slerp	Spherically interpolates between a and b by t. The parameter t is clamped to the range [0, 1]
RotateTowards	Returns the inverse of rotation
Rotate	Rotates a rotation from towards to

Euleria käytetään yksinkertaisimmissa objektien rotaatioon, kuten 2D-peliohjelmoinnissa. Käytössä ovat vektori, jossa on x, y, ja z komponentit,

joilla kuvastetaan tiettyä rotaatiota tiettyyn suuntaan. Unityn Editor-näkymässä näkyy objektien rotaatio suoraan Eulerina, mutta niiden arvoja muutetaan salaa kvaternioksi. (Unity Technologies, n.d.)



Kuva 16. Kuva Unity Editor-ikkunasta, jossa muutetaan objektin rotaatiota suoraan. (Kuva Unity Technologies, n.d.)

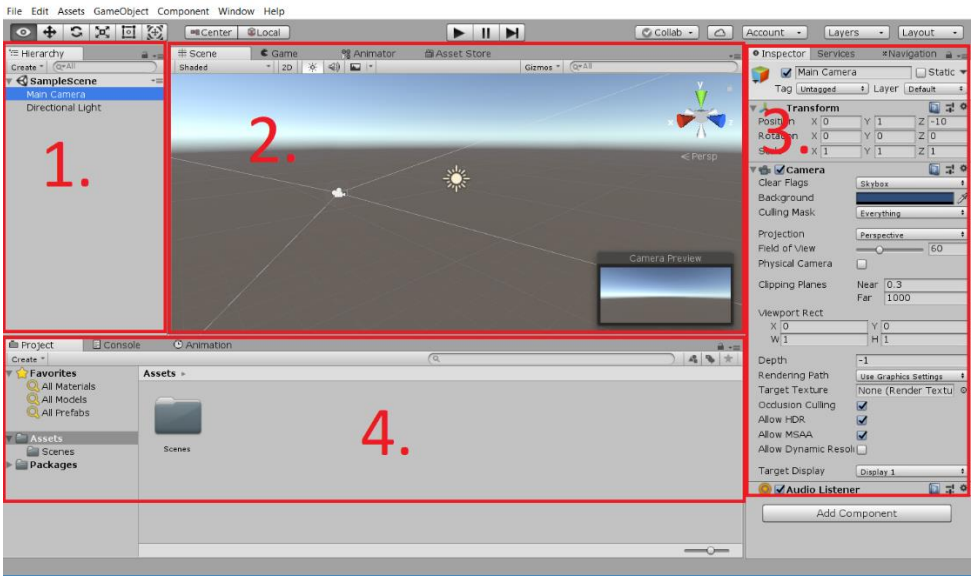
Kvaternio käyttää myös x, y, ja z komponentteja, mutta käyttää myös neljättä, w komponenttia, joka on skalaari. Tämä on vektori, jossa on neljä komponenttia. Kvaterniota käytetään 3D-peliohjelmoinnissa enemmän, kuin Euleria. Euleriä käyttäessä Unity muuttaa sen automaattisesti kvaternioksi, jotta vältytään ongelmilta, joita Euler aiheuttaa, kuten mm. Gimbal Lock. Gimbal Lock on rotaation lukitus, jossa kolmas komponentin akseli osoittaa samaan suuntaan, kuin ensimmäisen ja toisen komponentin akselit. Tämä estää akselin kääntymistä haluttuun suuntaan, ja voi aiheuttaa ongelmia mm. animaatioiden kanssa. (Unity Technologies, n.d.)

4 UNITY-PELIMOOTTORIN KÄYTTÖÖNOTTO

Käytössä oli Unity-versio 2018.2 ja Visual Studio 2017-ohjelmaa ohjelmointia varten. Unityn pystyy lataamaan Unityn omilta sivuilta ja Visual Studion voi ladata Microsoftin sivuilta.

Projektin luodatessa voi laittaa projektille nimen ja valita onko se 2D- vai 3D-projekti. Projektin asetuksia voi koska tahansa muuttaa. Unityssa luodaan tai avataan valmiiksi tehty projekti. Projektin avaamisen tai luomisen jälkeen siirrytään automaattisesti työskentely ikkunaan. Vasemmassa ylänurkassa on palkki Hierarchy (kohta 1), joka listaa kyseisen Scenen olemassa olevat peliobjektit. Scene on nimeltään SampleScene.

Hierarchyn vieressä on Scene-näkymä (kohta 2), joka näyttää peliobjektien, jos niitä on kameran näkökulmassa. Näkymää voi vaihtaa Game-näkymään, joka näyttää pelin Main Camera-peliobjektin näkökulmasta. Hierarchyn ja Scenen yläpuolella on GameObject-nappula, josta voi lisätä peliobjekteja peliin. Scene-näkymän vieressä on Inspector (kohta 3), joka näyttää valitun peliobjektin komponentit. Alimmaisena on projektin tiedostot (kohta 4). Projektin tiedostossa (kohta 4) voit hiiren oikealla klikkauksella luoda kansioita, skriptejä ja muita tiedostoja.



Kuva 17. Kuvakaappaus Unityn käyttöliittymästä. Kuvaan on merkitty kohdat 1-4.

Luodatessa uutta koodia, Unity laittaa valmiiksi pari metodia, jotka tulevat *Monobehaviour*-luokan mukana. Start-metodi käy koodeja läpi vain kerran, kun skripti käynnistyy. Update-metodi käy koodeja läpi jokaisen kuvaruudun aikana.

```
1   using System.Collections;
2   using System.Collections.Generic;
3   using UnityEngine;
4
5   public class SkriptinNimi : MonoBehaviour
6   {
7       // Start is called before the first frame update
8       void Start()
9       {
10
11      }
12
13      // Update is called once per frame
14      void Update()
15      {
16
17      }
18  }
```

Kuva 18. Kuvakaappaus Visual Studiosta, kun avataan tyhjää koodia.

Koodin nimi on *SkriptinNimi*, joka on sama, kuin tiedoston nimi. Jos koodi-tiedoston nimeä muutetaan, tulee muuttaa myös luokan nimeä.

5 MATEMATIIKAN TYÖKALUJEN TESTAAMINEN UNITYSSA

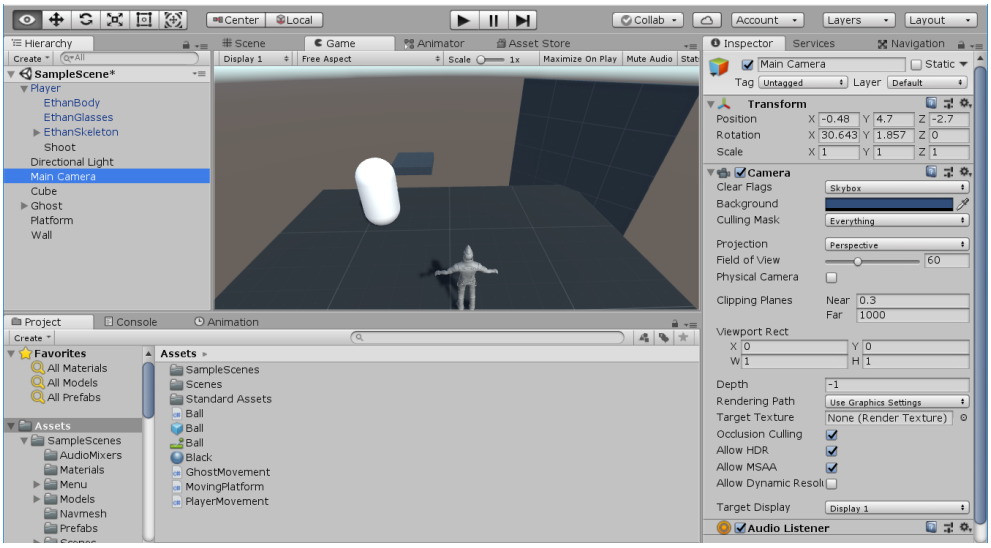
Tavoitteena oli käyttää seuraavia matematiikan työkaluja projektissa: vektoria, trigonometriaa ja kvaterniota. Projektista tuli esimerkki 3D-peli, jossa käytettiin erilaisia matematiikan työkaluja ratkaisemaan tavallisia ongelmia, kuten miten saadaan haamua seuraamaan pelaajaa tai palloa kimpoamaan seiniltä.

5.1 Esivalmistelu

Käytössä oli 3D-projekti, johon ladattiin **Standard Assets**-assetti Unity Asset Storesta. Unity Asset Store on kauppa, josta löytyy mm. 3D-malleja, koodeja ja työkaluja joko ilmaiseksi tai maksullisena. Assetti voi olla koodia, 3D-mallia, musiikkia, tekstuureja tai jotain muuta, jota voi käyttää projektissa. Projektissa käytettiin assetista 3D-mallia esimerkkinä visualisoimaan projektia. Paketista ei käytetty valmiiksi tehtyjä koodeja.

Scenessä oli 3D-hahmo nimeltään *Player*, johon kytkettiin *Character Controller*-komponentti, joka mahdollisti peliobjektia liikkumaan mutkia pitkin ilman, että jää jumiin. Kyseisellä komponentilla on törmäystunnistin mukana automaattisesti. *Player*-peliobjektiin kytkettiin myös tyhjä peliobjekti *Shoot*, joka on kyseisen peliobjektin edessä, vatsan kohdalla.

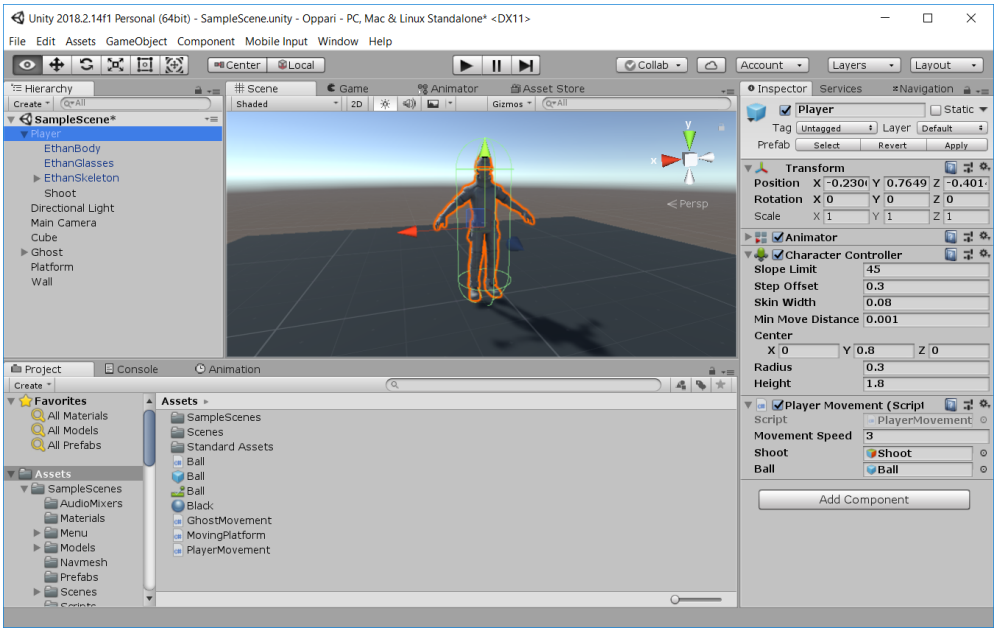
Scenestä löytyi myös useita Cube-peliobjektia, joilla oli *Box Collider*-komponentti kytkettynä. Useampaan Cube-peliobjekteihin oli laitettu tekstuurit, joita löytyi Standard Assets-kansiosta. Haamulla oli *Capsule Collider*-komponentti.



Kuva 19. Kuvakaappaus Scenestä, jossa on useita peliobjekteja.

5.2 Pelaaja hahmon liikuttaminen

Projektiin luotiin uusi koodi nimeltään *PlayerMovement*, joka kytkettiin *Player*-peliobjektiin. Pelaajaa liikutettiin nuolinäppäimillä, joiden perusteella hahmo kääntyy automaattisesti haluttuun suuntaan.



Kuva 20. Kuvakaappaus Player-peliobjektista ja sen komponenteista

Muuttuja *input* on vektori, ja pelaajan *movementSpeed* eli liikkumisnopeus on *skalaari*. *Shoot*-muuttujaan asetetaan Unityssa tyhjä peliobjekti, josta ammutaan *ball*-peliobjekteja.

```
CharacterController cc;          // Character Controller component for move-ment

Vector3 input;                   // Input variable for getting input from key-board
Vector3 targetDirection;         // Target direction we will rotate towards
public float movementSpeed = 3f;
public GameObject shoot;         // Child gameobject which we will spawn balls from
public GameObject ball;          // Ball gameobject which has rigidbody compo-nent

// Use this for initialization
void Start()
{
    // GetComponent so we can access variables and methods of these compo-nents
    cc = GetComponent<CharacterController>();
}
```

Kuva 21. Kuvakaappaus koodin muuttujista ja *Start*-metodista.

Start-metodista otetaan komponentteja vastaan, jotta pystytään käyttämään niiden metodeja ja muuttujia.

```
// Update is called once per frame
void Update()
{
    // Get input
    input.x = Input.GetAxisRaw("Horizontal");
    input.z = Input.GetAxisRaw("Vertical");
    input.y = 0f;

    // Update direction only when we press input, so direction won't reset
    if (input.magnitude > 0)
    {
        targetDirection = input;
        transform.rotation = Quaternion.Slerp(transform.rotation,
            Quaternion.LookRotation(targetDirection), 0.3F);
    }

    // Normalize input so we won't move faster if we move by vertical and
    // horizontal at the same time
    input.Normalize();

    // Move by input, speed and by time
    cc.Move(input * movementSpeed * Time.deltaTime);
}
```

Kuva 22. Kuvakaappaus *Update*-metodista, jossa laitettiin pelaaja liikkumaan ja kääntymään näppäinten mukaan.

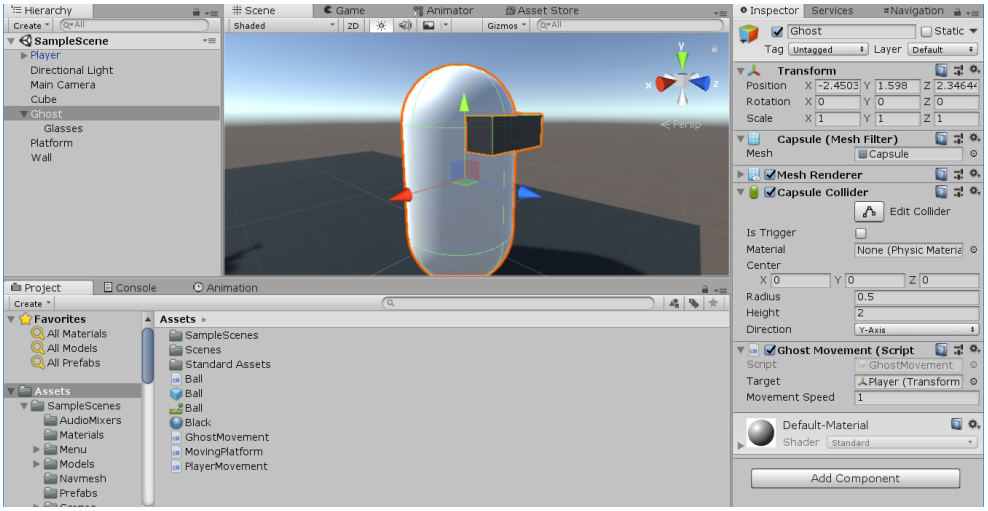
Update-metodissa otettiin näppäimiä vastaan, jotta saatiin pelaaja liikkumaan haluttuun suuntaan. Pelaajan laitettiin kääntymään vasta sitten, kun *input*-muuttujan pituus on suurempi, kuin 0, eli jotain liikkumisnappulaa painetaan edes vähän. *Input*-muuttujaa normalisoidaan mahdollisen rotaation jälkeen, jotta pelaaja ei pystyisi liikkumaan nopeammin, jos esim. pelaaja painaa ylös ja oikealle yhtä aikaa.

Hahmon rotaatioon käytetään *Quaternion.Slerp*-metodia, jotta pystytään sulavasti kääntymään haluttuun suuntaan.

Lopuksi käytettiin *CharacterController*in *Move*-metodia liikuttamaan hahmoa. Input-muuttujaa kerrataan skalaarilla, *movementSpeed-muuttujalla*. *Time.deltaTime* auttaa liikuttamaan hahmoa ajan mukaisesti, eikä pelin kuvaruudun mukaisesti. Jos *deltaTime*:ä ei käytetä, niin objekti liikkuisi todella nopeasti. Koodissa käytettiin vektoreita, skalaaria, normalisointia ja kvaterniota.

5.3 Haamun pieni tekoäly

Seuraavaksi luotiin koodi, joka liikutti haamua pelaajaa kohti, kun pelaajan hahmo ei katso sitä kohti tai etäisyyttä on vähemmän, kuin 2f. Koodin nimi oli *GhostMovement*. Sceneen lisättiin 3D-objektin Capsule, johon kytkettiin *GhostMovement*-koodi.



Kuva 23. Kuva kaappaus haamusta ja sen komponenteista.

GhostMovement-koodissa oli *target* ja *movementSpeed*-muuttujia. Muuttuja *target* on kohde, jota seurataan. *Update*-metodissa laitettiin peliobjekti katsomaan kohdetta koko ajan. Pistetuloa otettiin haamusta ja pelaajasta, jotta pystytään määrittelemään mihin suuntaan molemmat peliobjektit katsovat. Jos pistetulo on isompi, kuin 0, niin pelaaja ja haamu katsovat samaan suuntaan eli silloin haamu liikkuu pelaaja kohti. Haamu ei seuraa pelaajaa, jos etäisyyttä on alle 2f eli noin 2 metriä.

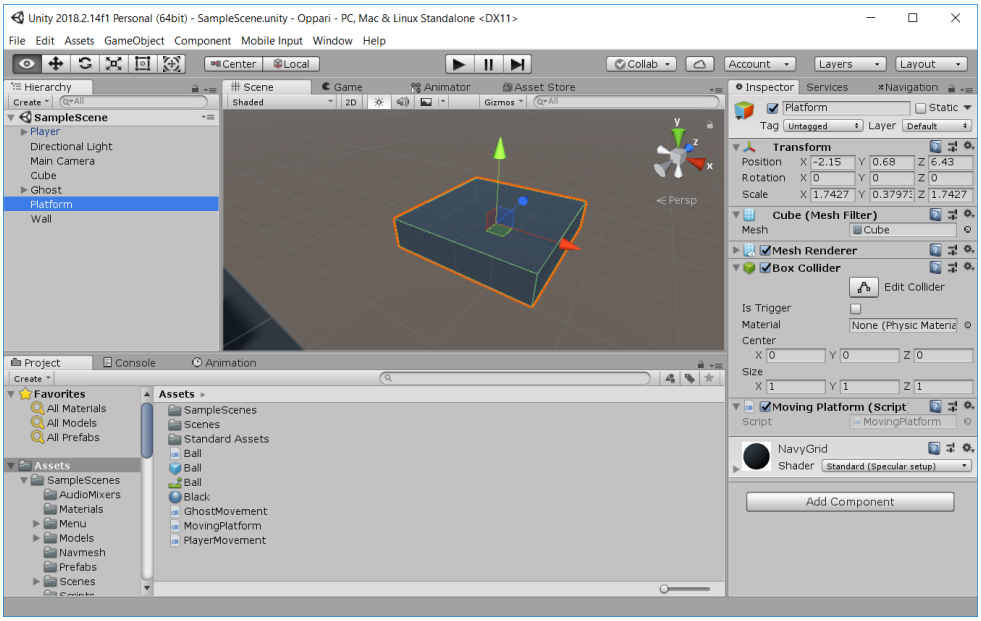
```
public Transform target;                // target object we follow
public float movementSpeed = 2f;        // movement speed

// Update is called once per frame
void Update()
{
    // If we have a target..
    if (target != null)
    {
        // Keep looking at the target
        transform.LookAt(target);

        // Calculate distance
        var heading = target.transform.position - transform.position;
        var distance = heading.sqrMagnitude;

        if (distance > 2f)
        {
            // Take dot product of targets forward and this objects as well
            float dot = Vector3.Dot(target.forward, transform.forward);

            // If objects are looking in the same direction...
            if (dot > 0)
            {
                // Follow the target
                transform.position = Vector3.MoveTowards(transform.position,
                    target.position, movementSpeed * Time.deltaTime);
            }
        }
    }
}
```

Kuva 24. Kuvakaappaus haamun *GhostMovement*-skirptistä.

Kun *GhostMovemen*t-koodi on kiinni haamussa, pitää myös asettaa Unityssa *Player*-peliobjekti *target* muuttujaan, jotta seuraaminen onnistuisi. Koodissa käytettiin vektoreita, vektorin pituutta, etäisyyden laskemista ja pistetuloa.

5.4 **Liikkuva alusta**

Scenessä oli Platform-peliobjekti, johon kytkettiin *MovingPlatform*-koodi. Platformin tarkoitus oli liikkua ylös ja alas pysähtymättä, käyttäen siniaaltoa. Pelaaja voi mennä alustan päälle.



Kuva 25. Kuvakaappaus Platform-peliobjektista ja sen komponenteista.

MovingPlatform-koodissa liikutetaan alustaa ylös ja alas käyttäen siniaaltoa. Aluksi koodiin luotiin *startingPosition*-muuttuja, joka on vektori ja säilyttää peliobjektin sijainnin kerran, kun peli käynnistyy.

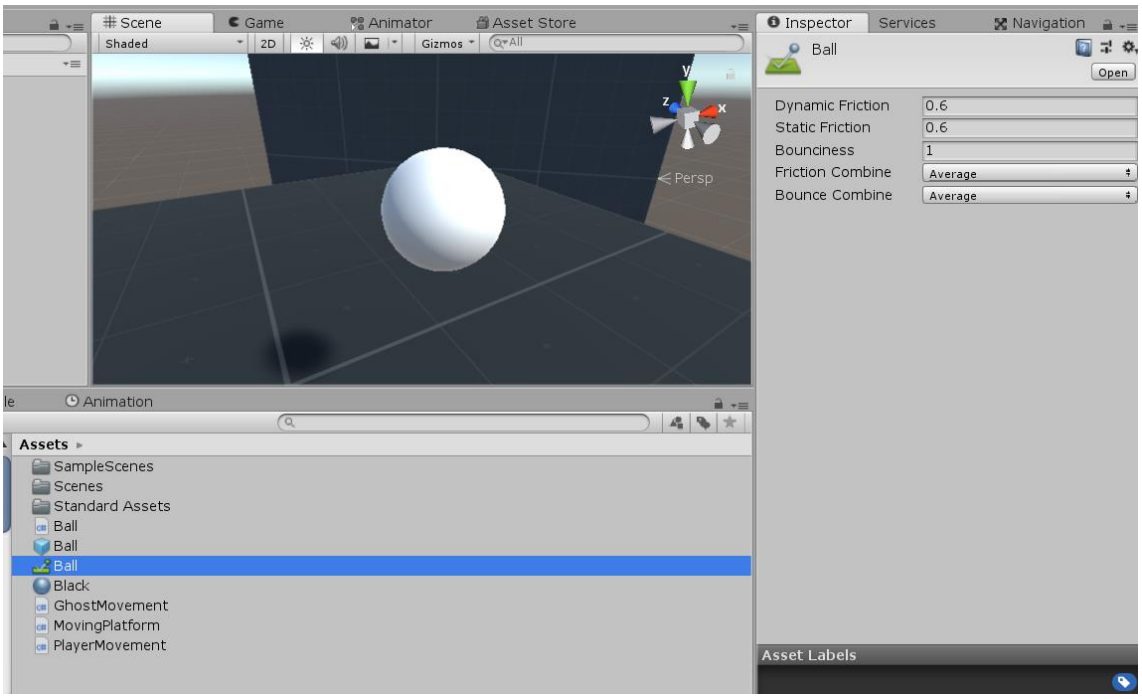
MovingPlatform-koodin *Update*-metodissa luotiin tilapäinen muuttuja y, joka otti *startingPosition*-muuttujasta y komponentin ja käytti *Mathf.Sin*-metodia, joka ottaa parametriksi ajan siitä lähtien, kun peli on käynnistynyt. Kyseisessä koodissa käytettiin vektoreita ja trigonometriaa, siniaaltoa.

```
Vector3 startingPosition;

// initialization
void Start()
{
    // Save starting position for later use
    startingPosition = transform.position;
}

// called once per frame
void Update()
{
    // Use sine wave to make platform move up and down
    var y = startingPosition.y + Mathf.Sin(Time.timeSinceLevelLoad);

    // Apply new y and use x and z starting position value
    transform.position = new Vector3(startingPosition.x, y, startingPosition.z);
}
```

Kuva 26. Kuvakaappaus *MovingPlatform*-koodista.

5.5 Pallon kimpoaminen

Sceneen luotiin uusi prefab-peliobjekti nimeltään Ball, johon on kytketty *Sphere Collider*-komponentti. Kyseisen peliobjektin tarkoituksena oli toimia luotina, kun pelaaja ampuu palloja. Transformista peliobjektia skaalaus on (0.25f, 0.25f, 0.25f). *Sphere Collideriin* on kytkettynä *Physics Material*-komponentti, joka lisää pallon ponnahdusta, kun se osuu seiniin. Peliobjektiin luotiin ja kytketiin *Ball*-skirpti.



Kuva 27. Kuvakaappaus pallosta, jolla on *Physics Material*-komponentti kytkettynä *Sphere Collideriin*.

Ball-koodissa luotiin kaksi muuttujaa. *Rigid*-muuttuja säilytti *Rigidbody*-komponentin ja *reflectPower*-muuttujaa käytettiin kimmottamaan palloa voimakkaammin, kun se osui peliobjektiin.

```
Rigidbody rigid;                    // Rigidbody component attached to this gameObject
public float reflectPower = 2f;     // Reflect power when we hit something

// Initialization
void Start()
{
    // Get rigidbody component
    rigid = GetComponent<Rigidbody>();

    // Destroy this gameobject in 2 seconds
    Destroy(this.gameObject, 2f);
}
```

Kuva 28. Kuvakaappaus koodin ensimmäisestä osasta.

Seuraavaksi koodiin kirjoitettiin *OnCollisionEnter*-metodi, joka tunnistaa törmäyksiä törmäystunnistimin kautta. Metodin sisälle kutsuttiin *GetReflect*-metodi, joka käytti ristituloa löytääkseen vektorin normaalin.

Ristitulolla pystytään kimmottamaan palloa sillä perusteella, että mihin kohtaan peliobjektia se osuu. Lopuksi heijastettiin vektoria, jotta se menisi vastakkaiseen suuntaan, kun se osuu peliobjektiin. *GetReflect*-metodi palauttaa vektorin, jota käytettiin lisäämään pallon nopeutta vastakkaiseen suuntaan *OnCollisionEnter*-metodissa. Unityssa asetettiin Player-peliobjektiin ball-peliobjekti muuttujaan. Koodissa käytettiin vektoreita ja ristituloja.

```
private void OnCollisionEnter(Collision collision)
{
    if (rigid != null)
    {
        // Get reflect from target collision
        var reflect = GetReflect(collision.transform);

        // Get current velocity from rigidbody
        var velocity = rigid.velocity;

        // Add reflect and current velocity plus some reflect power
        rigid.velocity = (reflect + velocity) * reflectPower;
    }
}

Vector3 GetReflect(Transform target)
{
    // Get targets and balls transform positions
    Vector3 ballVector = target.position - transform.position;

    // Get cross from ballVector and target's forward position, so we get a plane
    Vector3 plane = Vector3.Cross(ballVector, target.forward);

    // Get plane's normal
    Vector3 planeNormal = Vector3.Cross(plane, ballVector);

    // Reflect
    Vector3 reflected = Vector3.Reflect(target.transform.forward, planeNormal);

    // Return direction of reflect
    return reflected.normalized;
}
```

Kuva 29. Kuvakaappaus Ball-koodista

5.6 PlayerMovement-koodin päivittäminen

PlayerMovement-koodiin lisättiin *Update*-metodin sisälle näppäin tunnistin, jolla luodaan palloja aina, kun pelaaja klikkaa hiiren painikkeella. *Instantiate*-metodi luo halutun pallo-peliobjektin, jonka sijaintia asetetaan shoot-peliobjektin sijainnille, eli pelaajan eteen.

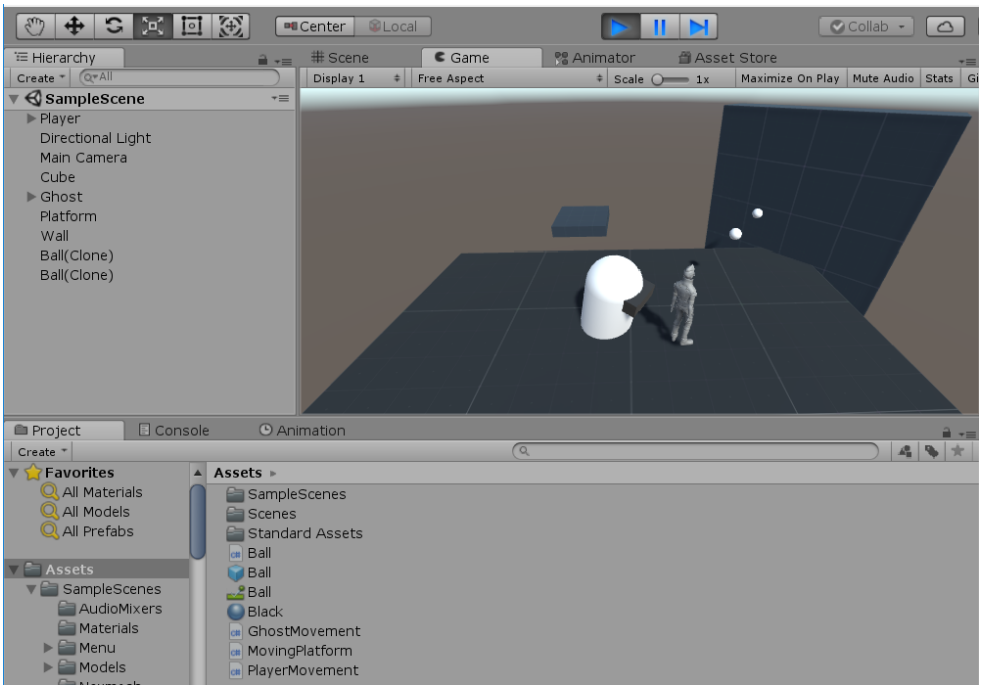
```
// Shoot balls from Mouse 1
if (Input.GetButtonDown("Fire1"))
{
    // Spawn ball-gameobject
    GameObject GO = Instantiate(ball);
    GO.transform.position = shoot.transform.position;
    GO.transform.forward = transform.forward;
    // Add force to ball

    GO.GetComponent<Rigidbody>().AddForce(transform.forward * 1000f);
}
```

Kuva 30. Kuvakaappaus PlayerMovement-koodista.

Unityssa asetettiin Player-peliobjektille ball-muuttujaan luotu pallo -prefab, jotta pelaaja pystyisi ampumaan palloja.

5.7 **Lopputulos**

Projektiin tehtiin useita peliobjekteja, jotka käyttävät tiettyjä matemaattisia metodeja suorittamaan toimintojaan. Pelaajaa pystyttiin liikuttamaan nuolinäppäimillä, ja hiiren klikkauksella ammuttiin palloja hahmon katseen mukaisesti käyttäen *shoot*-peliobjektia. Haamu seuraa pelaajaa, kun molemmat katsovat samaan suuntaan, kunnes pituutta on vähemmän, kuin kaksi metriä. Scenessä Platform-peliobjekti liikkuu ylös ja alas käyttäen siniaaltoa.



Kuva 31. Kuvakaappaus lopputuloksesta.

6 YHTEENVETO

Saatiin tehtyä projekti, joissa käytettiin lähes jokaista matematiikan työkaluja, joita on tullut vastaan tässä opinnäytetyössä.

Erilaisten matematiikan työkaluista on hyötyä peliohjelmoinnissa, mutta niiden käyttö on ohjelmoijasta kiinni. Ohjelmoinnin osaaminen on yhtä tärkeää, kuin matematiikan soveltaminen pelin kehityksessä, sillä jopa yksinkertaisimpiin asioihin tarvitaan matematiikkaa. Toivoisin, että ammattikoreakouluissa otettaisiin matematiikan tarve huomioon peliohjelmoinnissa, sillä se on mielestäni yksi tärkeimmistä taidoista, jota kyseinen ohjelmoija voi tarvita työelämässään.

Opin työn aikana lineaarialgebran tärkeyden peliohjelmoinnissa. Samalla tuli tutuksi trigonometria, matriisit ja kvaterniot. Koen ymmärtäväni näitä enemmän, kuin silloin, kun aloitin työn kirjoittamista. Lisäksi koen tarvitsevani näitä taitoja tulevaisuudessa, kun teen töitä pelifirmassa peliohjelmoijan roolina.

Työtä olisi voitu jatkaa vielä pidemmälle niin, että esille olisi tuotu mm. pelifysikkaa, kinemaattisia yhtälöitä ja tilastoja, mutta aika tuli vastaan, eikä taitoni riittäneet käsittelemään näitä aiheita.

LÄHTEET

Fletcher, J. (n.d.). *THE INCREDIBLY USEFUL SINE WAVES PART 1 – USING THE SIN FUNCTION – (TRIGONOMETRY) (GAME DEV PRIMER)*. Haettu 10.11.2018 osoitteesta https://weeklycoder.com/2015/07/22/the-incredibly-useful-sine-waves-part-1-trigonometry-game-dev-primer/

Forbes (2016). *This Is The Math Behind Super Mario*. 21.10.2016. Haettu 1.11.2018 osoitteesta https://www.forbes.com/sites/quora/2016/10/21/this-is-the-math-behind-super-mario/#603847132154

Game Designing (2018). *Using Math in Programming & Game Design*. 12.5.2018. Haettu 27.10.2018 osoitteesta 15.10.2018 osoitteesta https://www.gamedesigning.org/learn/game-development-math/

Leeuwen, J. (2010). *Mathematics in game development: Trigonometry*. Haettu 4.11.2018 osoitteesta http://www.softlion.nl/download/article/Trigonometry.pdf

McShaffry M. & Graham D. (2013). *Game Coding Complete, Fourth Edition*. USA, Course Technology.

Quora (n.d.). *How are matrices useful in game development*. Haettu 1.12.2018 osoitteesta https://www.quora.com/How-are-matrices-useful-in-game-development

Rosen, D. (2009). Agenda 2009 - Linear algebra for game developers. Blogijulkaisu 3.7.2009. Haettu 1.11.2018 osoitteesta http://blog.wolfire.com/2009/07/linear-algebra-for-game-developers-part-2/

Madhav, S. (2013). *Game Programming Algorithms*. 12.2013. Haettu 10.10.2018 osoitteesta https://www.oreilly.com/library/view/game-programming-algorithms/9780133463200/ch03.html

Scichart (n.d.). The Left Handed Coordinate System (LHS). Haettu 10.11.2018 osoitteesta https://www.scichart.com/documentation/v5.x/Orientation%20(3D%20Space)%20in%20the%20SciChart3DSurface.html

Thirslund, A. (2016). Game Math Theory – VECTORS. 18.9.2016. Haettu 5.11.2018 osoitteesta
https://www.youtube.com/watch?v=wXI9_olSrqo

Unity Technologies (n.d.). *Experienced programmer, but new to Unity? You're already ahead of the game*. Haettu 27.10.2018 osoitteesta
https://unity3d.com/programming-in-unity

Unity Technologies (n.d.). *Game Engines – How do they work?* Haettu 5.11.2018 osoitteesta
https://unity3d.com/what-is-a-game-engine

Unity Technologies (n.d). Mathf. Haettu 27.10.2018 osoitteesta
https://docs.unity3d.com/ScriptReference/Mathf.html

Unity Technologies (n.d.). Rotation and Orientation in Unity. Haettu 10.11.2018 osoitteesta
https://docs.unity3d.com/Manual/QuaternionAndEulerRotation-sInUnity.html

Unity Technologies (n.d.). *Understanding Vector Arithmetic*. Haettu 1.11.2018 osoitteesta
https://docs.unity3d.com/Manual/UnderstandingVectorArithmetic.html

Weisstein, E. (n.d.). Normal Vector. Haettu 8.12.2018 osoitteesta
http://mathworld.wolfram.com/NormalVector.html

Projektin koodit

PlayerMovement.cs

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

public class PlayerMovement : MonoBehaviour
{
    CharacterController cc;    // Character Controller component for movement

    Vector3 input;             // Input variable for getting input from keyboard
    Vector3 targetDirection;   // Target direction we will rotate towards
    public float movementSpeed = 3f;
    public GameObject shoot;  // Child gameobject which we will spawn balls from
    public GameObject ball;   // Ball gameobject which has rigidbody component

    // Use this for initialization
    void Start()
    {
        // GetComponent so we can access variables and methods of these compo-
nents
        cc = GetComponent<CharacterController>();
    }
    // Update is called once per frame
    void Update()
    {
        // Get input
        input.x = Input.GetAxisRaw("Horizontal");
        input.z = Input.GetAxisRaw("Vertical");
        input.y = 0f;
        // Update direction only when we press input, so direction won't reset
        if (input.magnitude > 0)
        {
            targetDirection = input;
            transform.rotation = Quaternion.Slerp(transform.rotation,
                Quaternion.LookRotation(targetDirection), 0.3F);
        }
        // Normalize input so we won't move faster if we move by vertical and
        // horizontal at the same time
        input.Normalize();

        // Move by input, speed and by time
        cc.Move(input * movementSpeed * Time.deltaTime);

        // Shoot balls from Mouse 1
        if (Input.GetButtonDown("Fire1"))
        {
            // Spawn ball-gameobject
            GameObject GO = Instantiate(ball);
            GO.transform.position = shoot.transform.position;
            GO.transform.forward = transform.forward;
            // Add force to ball

            GO.GetComponent<Rigidbody>().AddForce(transform.forward * 1000f);
        }
    }
}
```

Ball.cs

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

public class Ball : MonoBehaviour
{
    Rigidbody rigid;                    // Rigidbody component attached to this
gameObject
    public float reflectPower = 2f;     // Reflect power when we hit something

            // Initialization
            void Start ()
    {
        // Get rigidbody component
        rigid = GetComponent<Rigidbody>();

        // Destroy this gameobject in 2 seconds
        Destroy(this.gameObject, 2f);
             }

    private void OnCollisionEnter(Collision collision)
    {
        if (rigid != null)
        {
            // Get reflect from target collision
            var reflect = GetReflect(collision.transform);

            // Get current velocity from rigidbody
            var velocity = rigid.velocity;

            // Add reflect and current velocity plus some reflect power
            rigid.velocity = (reflect + velocity) * reflectPower;
        }
    }

    Vector3 GetReflect(Transform target)
    {
        // Get targets and balls transform positions
        Vector3 ballVector = target.position - transform.position;

        // Get cross from ballVector and target's forward position, so we get a
plane
        Vector3 plane = Vector3.Cross(ballVector, target.forward);

        // Get plane's normal
        Vector3 planeNormal = Vector3.Cross(plane, ballVector);

        // Reflect
        Vector3 reflected = Vector3.Reflect(target.transform.forward, planeNor-
mal);

        // Return direction of reflect
        return reflected.normalized;
    }
}
```

MovingPlatform.cs

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

public class MovingPlatform : MonoBehaviour
{
    Vector3 startingPosition;

            // initialization
            void Start ()
    {
        // Save starting position for later use
        startingPosition = transform.position;
            }

            // called once per frame
            void Update ()
    {
        // Use sine wave to make platform move up and down
        var y = startingPosition.y + Mathf.Sin(Time.timeSinceLevelLoad);

        // Apply new y and use x and z starting position value
        transform.position = new Vector3(startingPosition.x, y,
startingPosition.z);
            }
}
```

GhostMovement.cs

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

public class GhostMovement : MonoBehaviour
{
    public Transform target;                // target object we follow
    public float movementSpeed = 2f;        // movement speed

    // Update is called once per frame
    void Update()
    {
        // If we have a target..
        if (target != null)
        {
            // Keep looking at the target
            transform.LookAt(target);

            // Calculate distance
            var heading = target.transform.position - transform.position;
            var distance = heading.sqrMagnitude;

            if (distance > 2f)
            {
                // Take dot product of targets forward and this objects as well
                float dot = Vector3.Dot(target.forward, transform.forward);

                // If objects are looking in the same direction...
                if (dot > 0)
                {
                    // Follow the target
                    transform.position = Vector3.MoveTowards(transform.position,
                        target.position, movementSpeed * Time.deltaTime);
                }
            }
        }
    }
}
```