Eliel Martti

Fysiikan kaavojen soveltaminen VR-peliprojektiin

Metropolia Ammattikorkeakoulu

Muotoilija (AMK)

Muotoilun tutkinto-ohjelma

Opinnäytetyö

Joulukuu 2022

Tiivistelmä

Tekijä(t):	Eliel Martti
Otsikko:	Fysiikan kaavojen soveltaminen VR-peliprojektiin
Sivumäärä:	14 sivua
Aika:	Joulukuu 2022

Tutkinto:	Muotoilija (AMK)
Tutkinto-ohjelma:	Muotoilu
Suuntautumisvaihtoehto:	XR Design
Ohjaaja(t):	Lehtori Turkka Loimisto

Monesti aloittelevilla kehittäjillä tulee toteutusideoita peliprojekteihinsa, ja ne voivat sisältää toimintatapoja, joiden aihepiirit eivät ole niin tuttuja. Opinnäytetyössä tutkitaan, kuinka fysiikan lakien mukaisia kaavoja voi hyödyntää pelimekaniikan luomisessa.

Opinnäytetyö perustuu VR-peliin, jossa pelaaja pystyy lentämään linnun tavoin heiluttamalla ohjaimia ja käsiään kuten siipiä. Peli on tarvinnut toimiakseen aerodynamiikan kaavoja, joita on käännetty Unity-pelimoottorin fysiikkamallinnuksessa toimiviksi pelillisesti toimivan realistisuuden puitteissa.

Projektiin liittyvää koodia avataan melko yksityiskohtaisesti, ja opinnäytetyön tavoitteena on auttaa lukijaa ymmärtämään, kuinka voisi itse kehittää vastaavia ratkaisuja. Opinnäytetyössä käydään myös nopeasti läpi fysiikkaan ja koodaamiseen tarvittavia perusteita.

Lopullisessa projektissa on saatu hyväntuntuinen lentokokemus, jossa pystyy lentämään ympäri virtuaalista kaupunkiympäristöä. Lentäminen ei ole liian monimutkaista, mutta pelaajalla on kuitenkin paljon hallintaa lentäessä.

Avainsanat: VR, Unity, Rigidbody, Aerodynamiikka, Vektori, metodi, luokka

Abstract

Author(s):	Eliel Martti
Title:	Application of Physics Formulas Into a VR Game Project
Number of Pages:	14 pages
Date:	December 2022
Degree:	Bachelor of Culture and Arts
Degree Programme:	Design
Specialisation option:	XR Design
Instructor(s):	Turkka Loimisto, Senior Lecturer

Often beginner developers have ideas for their game projects, and they can include methods that are not very familiar. The thesis explores how it is possible to apply the formulas from aerodynamics in the creation of a game mechanic.

The thesis is based on a VR game, in which the player can fly like a bird by flapping their controllers and hands like a pair of wings. To function the game requires the use of aerodynamic formulas, which have been converted to be functional in the physics simulation of the Unity game engine, within the bound of gamified realism.

Code in the project is examined fairly in-depth, and the mission of the thesis is to help the reader to understand, how they could develop similar solutions. The thesis also quickly goes through some of the aspects of physics and coding that are needed in the project.

The end product gives a nice flying experience, where one can fly in a virtual city-scape. The flying is not too complex, but the player still has a lot of control over the movements while flying.

Keywords: VR, Unity, rigidbody, aerodynamics, vector, method, class

Sisällys

1 Johdanto

Joskus sovelluksissa ja projekteissa hyödynnetään fysiikkaa jollain tavalla, mihin ei löydy pelimoottoreissa suoraa komponenttia tai valmista osaa. Kehittäjä saattaa kokea omat ohjelmointitaitonsa tai fysiikan tietämyksen puutteelliseksi, jotta voisi itsevarmasti aloittaa luomaan tarkoituksiinsa soveltuvaa fysiikkamallinnusta itse.

Olen luonut Unity-pelimoottorilla pelisovelluksen, joka hyödyntää aerodynaamisia periaatteita, ja avaan prosessia, jolla olen ohjelmoinut pelihahmon käyttäytymään näiden periaatteiden pohjalta. Peli toimii VR:ssä eli virtuaalisessa todellisuudessa (Virtual Reality), ja pelissä on mahdollista lentää kuin lintu kaupunkiympäristössä.

Koska ihmiset eivät osaa lentää, pitää ottaa huomioon monta lentoa auttavaa asiaa, joista suurin on lentämisen tasoittaminen ja liiallisien voimien vähentäminen, joita voi tulla lentämisestä huonolla tekniikalla.

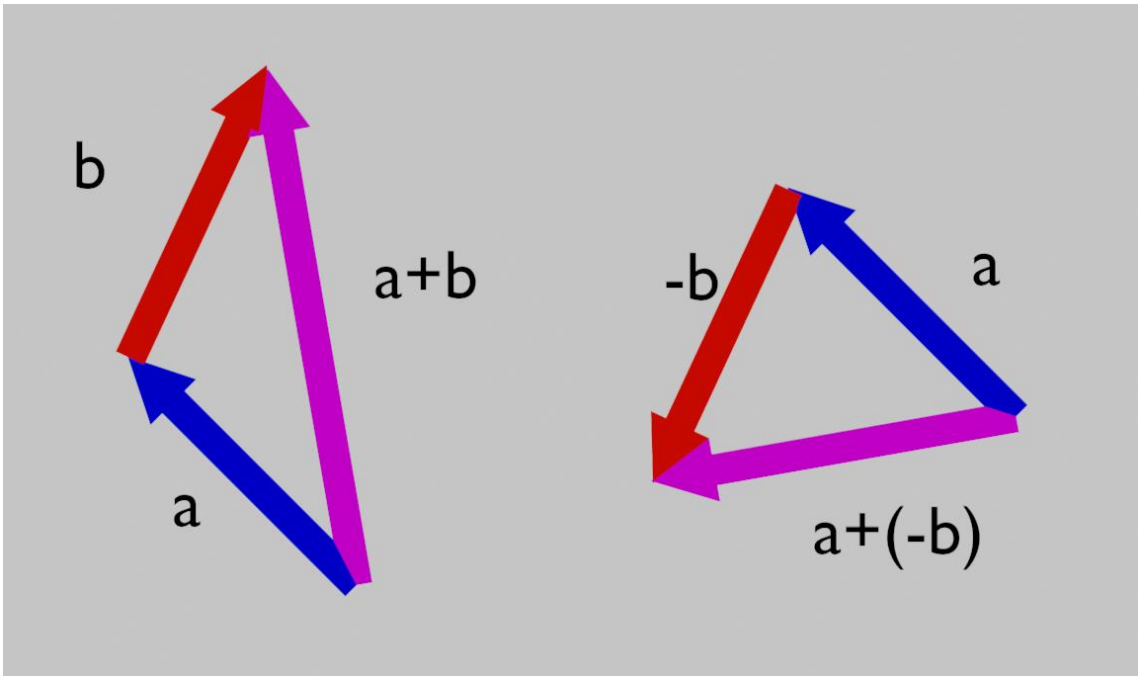
Normaalisti lentävillä kappaleilla on pyrstö, joka auttaa tasaamaan lentoa, sillä se hidastaa käännösten tekemistä ylös ja alas. Pyrstön puutteen huomasi heti pelissä, sillä ilman sitä pelaaja alkaa heittämään voltteja. Pyrstön korvaamiseksi ja käytettävyyden takia kallistumisen määrää on rajoitettu.

VR:ssä lentäessä on hankala sanoa, mihin suuntaan voima- ja nopeusvektorit osoittavat, mikä voi hankaloittaa ongelmatilanteiden ratkaisemisia. Vektorien visualisoinnin avuksi loin suuntailmaisimen, mistä voi nähdä vektorit lennon aikana. Olen myös tehnyt yksinkertaisen käyttöliittymän, jolla pystyy reaaliajassa muuttamaan lentämisen parametreja.

2 Vektorit

Fysiikassa ja varsinkin Unityn fysiikkamoottorissa vektorit ovat tärkeitä, ja vektorien ymmärtäminen auttaa koodin ja aerodynamiikan käsittämiseen. 3D-ulottuvuudessa Unityn vektori (Vector3) koostuu x-,y- ja z-komponenteista, mikä on verrattavissa sijaintiin Unityssä, jos sijaintia ajattelee vektorina Unityn ympäristön origosta. Projektissa vektoreita lisätään toisiinsa ja vähennetään toisistaan, kerrotaan ja muokataan Unityn toiminnoilla.

Vektorien vähentäminen toisistaan ja lisääminen toisiinsa on sama kuin niiden ketjuttaminen peräkkäin, joista lopputuloksena on uusi vektori. Jos vektori vähennetään toisesta, se on sama kuin vähennettävän vektorin eteen laitetaan miinusmerkki, joka tarkoittaa sen vektorin osoitussuunnan kääntämistä.



Kuva 1. Esimerkki vektorien lisäämisestä ja vähentämisestä.

Jos vektoria kerrotaan, sen pituus muuttuu, mikä on voimavektorissa sen voiman kasvamista. Vektorin pituuden saa sen *.magnitude*-määritteestä. Kun vektori normalisoidaan, sen pituus muunnetaan yhdeksi, mitä voidaan käyttää, kun tarvitaan vain vektorin suunta. Unityssä vektorin saa normalisoituna sen *.normalized*-määritteestä.

3 Unityn fysiikkamallinnus

Kaikki Unityn puolella simuloitu fysiikka tapahtuu FixedUpdate-methodissa, mikä tapahtuu aina tietyn, kehittäjän määrittelemän, aikavälin mukaan. Monissa peliprojekteissa hyödynnetään fysiikkaa, mutta monimutkaisemman fysiikkamallinnuksen pitää luoda itse, esimerkiksi Unityn mallinnuksessa ei ole aerodynamiikkaa.

3.1 Rigidbody

Unityssa rigidbody-komponentti peliobjektissa vastaa kappaleen fysikaalisia ominaisuuksia oikeassa elämässä, ja siihen sisältyy esimerkiksi massa, nopeus ja liikkeen vastus. Tärkeä asetus peliobjektin komponentissa on, minkä tyyppinen fysiikkalaskelmoinnin interpolointi on, joka voi tasoittaa fysiikkaa sen päivitystaajuuden välissä. Interpolointiasetuksia on *off*, *interpolate*, joka tasoittaa liikettä edellisen päivitetyn ruudun mukaan, ja *extrapolate*, joka arvioi seuraavan ruudunpäivityksen aikana olevan sijainnin ja yrittää tasoittaa liikettä sen mukaan.

Rigidbodyn *Drag* eli vastus toimii samankaltaisesti kuin ilmanvastus, mutta se ei ole pelissä käytännöllinen ilmanvastuksena. Vastuslaskelmat ei ota huomioon geometriaa, joten on helpompi säätää voimien määriä koodissa tilanteen mukaan. *Angular Drag* tarkoittaa kääntymisen vastusta, jota voi käyttää pyörimisen pehmentämiseen pelissä, esimerkiksi kun törmää seinään ja pelaaja menettää hallinnan lentäessään.

3.2 Aika pelimoottorissa

Tärkeä asetus fysiikkamallinnuksen kannalta on, miten aikaa käsitellään pelissä, joka tulee *Fixed Timestep*:in arvosta projektin fysiikka-asetuksissa. Fixed Timestep tai aika-askel tapahtuu perusasetuksilla joka 0,02 sekunnin välein, joka vastaa 50:tä kertaa sekunnissa. On hyvä käytäntö pitää kaikki fysiikkaan

liittyvät funktiot Unityn FixedUpdate-loopissa, minkä tiheys riippuu Fixed Timestep -asetuksesta.

VR-laitteissa näytön päivitysnopeus eli *fps* (Frames Per Second) on yleensä 72 fps:n ja 120 fps:n välillä, ja liian alhaisen päivitysnopeuden huomaa, kun liikkeiden latenssin takia pelissä ja oikeassa maailmassa on selkeä ero, joka voi aiheuttaa sekavuutta. Jos fysiikkapäivitys tapahtuu noin joka toinen ruudunpäivitys, pelissä saattaa näkyä selkeää tökkimistä, ja VR-laitteissa sen huomaa erityisen hyvin. Aika-askeleen lyhentäminen voi auttaa, mutta se myös lisää rasitusta laskelmointiin.

Sopiva arvo aika-askeleelle on 0,01, mikä tapahtuu 100 kertaa sekunissa. Asetukset ovat säädetty Oculus Quest 2 -laseihin parhaiten sopivimmiksi, joita olen itse käyttänyt. Laseissa ruudunpäivitysnopeus on 90 kertaa sekunnissa, joten fysiikan ei pitäisi vaikuttaa tökkivältä. On kuitenkin hyvä laittaa interpolointi päälle, jos käyttää laitteita, jotka päivittävät näytön useammin. Interpoloinnista ei ole kauhea haitta laskelmoinnille, koska se kuitenkin lasketaan vain pelaajan kohdalla.

4 Toiminnot

Suurimmat lentäessä siipiin kohdistuvat voimat ovat noste ja ilmanvastus, joiden suuruudet siivissä tulevat pääasiallisesti nopeudesta ja kulmasta suhteessa ilmanvirtaan (Wood 2022). Nopeuden ja suunnan saa Unityssä Rigidbody-komponentista.

Oikeassa elämässä näille voimille löytyy eri funktioita, jotka tulevat siiven fysikaalisista ominaisuuksista (muoto, paino, pinta-ala). Unityssä tätä ei saa suoraan automaattisesti, mutta arvoja laskukaavoissa pitää säätää, kunnes löytää sopivan tuntuisen funktion lentoa varten.

Oikeiden kertoimien löytäminen on hankalaa, ja vaatii paljon iteraatiota, sillä muuttujia on paljon, eikä niistä välttämättä heti tule vaikutusta, ellei ole jotain rajatapausta. Fysiikan periaatteiden hyödyntäminen ja koodin suhteellistaminen fysiikan lakeihin auttaa löytämään sopivia arvoja, mutta joskus arvoja pitää muuttaa käyttömukavuuden vuoksi, kuten nosteen voiman suhteeton suuruus, jotta voi liidellä kauemmin.

4.1 Vääntömomentti

Lennon tasaaminen tai stabilointi on pelaajan orientoiminen siten, että maa on aina alhaalla tai VR-lasien kallistuminen pidetään lokaalissa tilassa. Pelaaja voi eläytyä lentämiseen yleensä kääntämällä päätään kallistumisen mukana. Jos näkymää kallistetaan automaattisesti, pelaajalle voi tulla helposti huono olo simulaatiosairaudesta.

Käytän lennon tasaamiseen vääntömomenttia, joka suoristaa pelaajaa koko ajan melko voimakkaasti, mutta esimerkkitilanteessa jos törmää rakennukseen, niin pelaajan koko peliobjekti keikahtelee niin nopeasti, ettei stabilointi suorista ihan heti. Efekti on kuitenkin tilanteeseen sopiva koska ikkunaan törmännyt lintu olisi normaalistikin vähän sekaisin suunnasta.

Stabilointi toimii koodissa ottamalla pelaajan ylöspäin suuntautuva vektori ja vertaamalla sitä maailman ylöspäin suuntautuvaan vektoriin (Vector3.up) ja lisäämällä vääntömomenttia suhteessa siihen, jotta pelaajan lokaali ja maailman ylöspäin osoittava vektori olisivat samat. Koska voima tähän tulee *Rigid-*

body.AddTorque-metodilla kiihtyvyysmoodissa, stabilointi tuntuu melko luonnolliselta.

```
private void Torque()
    {
        torqueVector = transform.TransformVector(transform.up);

        Quaternion deltaQuat = Quaternion.FromToRotation(playerRB.transform.up, Vector3.up);

        Vector3 axis;
        float angle;
        deltaQuat.ToAngleAxis(out angle, out axis);

        float dampenFactor = 0.8f;
        playerRB.AddTorque(-playerRB.angularVelocity * dampenFactor, ForceMode.Acceleration);

        float adjustFactor = 0.5f;
        playerRB.AddTorque(axis.normalized * angle * adjustFactor, ForceMode.Acceleration);

        playerRB.AddTorque(Vector3.up * Vector3.SignedAngle(transform.forward, playerRB.ve-
        locity, Vector3.up) * stability * 0.01f);
    }
```

Kuva 2. Vääntömomentin metodi.

4.2 Ohjaimien nopeus

Ohjaimien nopeutta tarvitaan voimien laskemiseen, ja onneksi nopeuden saaminen on yksinkertaista. Fysiikan kaavan mukaisesti nopeus on sijainnin muutos jaettuna siihen kuluvalla ajalle eli $\mathbf{v} = \Delta\mathbf{s}/\Delta\mathbf{t}$.

Ohjaimien sijainnin voi saada Unityssä peliobjektista *.transform*-määritteestä, mutta siinä pitää käyttää *.transform.localPosition*-lisämääritettä, mikä antaa sijainnin suhteessa liitettyyn kappaleeseen eli kappaleen *parent:*iin. Nopeuteen ei saa laskea mukaan itse pelaajan nopeutta, joka voi pahimmassa tapauksessa aiheuttaa takaisinsyöttöä ja nopeus vain kasvaisi äärettömiin. Siiven omalle keskisijainnille pitää päättää sopiva etäisyys ohjaimista. Hyvä etäisyys on esimerkiksi siipien geometrian keskellä.

Sijainnin muutoksen saa tallentamalla sijainnin Unityn LateUpdate-luupissa muuttujaan, ja sitten kun Update-luuppi tapahtuu seuraavassa ruudunpäivityksessä, voidaan vaan erottaa nykyinen sijainti LateUpdatessa saadusta muuttu-

jasta. Koska tämä sijaintimuutos tapahtuu joka ruudunpäivityksen aikana, voidaan hyödyntää Unityn sisäistä Time.deltaTime-metodia, joka palauttaa ruudunpäivitykseen kuluneen ajan.

Lopuksi GetControllerVelocity-metodia voi kutsua muissa skripteissä, mikä palauttaa siiven nopeuden. Koska nopeuden laskeminen on kytkeytynyt ruudunpäivitystiheyteen, kun se tapahtuu useammin kuin fysiikan laskelmointitiheys, niin nopeuden arvo on mahdollisesti tarkempi.

```
public class ControllerVelocity : MonoBehaviour
{
    private Vector3 velocity = Vector3.zero;
    private Vector3 latePosition = Vector3.zero;

    public Transform liftPoint;

    private void Update()
    {
        velocity = (latePosition - transform.localPosition) / Time.deltaTime;
    }

    private void LateUpdate()
    {
        latePosition = transform.localPosition;
    }

    public Vector3 GetControllerVelocity()
    {
        return velocity;
    }

}
```

Kuva 5. Luokka ohjaimen nopeudelle.

On myös mahdollista hyödyntää VR-laiteen oma dataa ohjaimien nopeudesta, mutta koska pelissä pelaajaa pitää skaalata linnun kokoiseksi, nopeus poikkeaa virtuaalitilassa oikeasta maailmasta. On vain turvallisempaa laskea nopeus suoraan pelin maailmassa.

4.3 Siipien toiminta

Siipien fysiikkaa pystyy mallintamaan hyvin tarkasti, esimerkiksi Silantro Flight Simulatorilla (Oyedoyin 2018), mutta VR-peliin mahdollisimman realistinen mallinnus ei ole olennaista ja useinkin haitallista. VR-peleissä ja sovelluksissa pitää ottaa käyttömukavuus ja -kokemus erityisesti huomioon, jonka takia voi tehdä pieniä muutoksia mallinnukseen. Pelaajan ennakkoluulot myös pitää ottaa huomioon, jotta pelaamisesta ei tule turhauttavaa kokemusta.

4.3.1 Siipien nostovoima

Koska pelaajalla on käytössä kaksi siipeä, niiden voimat lasketaan erikseen, ja ne lisätään pelaajaan yhdessä samaan aikaan. Siivissä teknisesti ainoa ero on sijainti, ja siihen voidaan määritellä järkevä nosteen keskipiste. Voidaan tehdä yksi metodi, joka laskee nosteen, ja toinen metodi, joka laskee siipien liikuttamisesta annetun voiman, ottamalla muuttujaksi kyseinen keskipiste ja suorittamalla metodi kerran per siipi.

Nostovoima lasketaan seuraavasti (muunnettu ISO-järjestelmään):

$$\boldsymbol{L = \frac{1}{2} dV^2 aCL}$$

- **L** = Nostovoima (Lift).
- **d** = Ilmantiheys (Density).
- **v** = Nopeus (Velocity).
- **a** = Siiven pinta-ala (Area).
- **CL** = Nostovoiman kerroin (Coefficient of lift).
  (Hodanbosi 1997.)

Nostovoima on lopullinen nostovoima, jonka suunta määräytyy siiven suunnasta. Ilmantiheys on vakio, ja sen vaikutus on lentäessä liidon helppous, ja siipienlyönnin antama tehokkuus. Nopeuden saa pelaajan Rigibody-komponentista, nimellisesti Rigidbody.velocity.magnitude. Magnitude ottaa kolmiulotteisen

velocity-vektorin suuruuden. Siiven pinta-alan voi päättää itse, ja se voidaan lisätä Unityn laskelmiin geneerisenä kertoimena. Unityyn laitettu koodi esitetään kuvassa 4.

```
private void WingLift(GameObject liftPoint)
{
    if (playerRB.velocity.magnitude > 0f)
    {
        liftVector = CalculateAirLift(liftPoint) * airDensity * playerRB.velocity.magnitude *
Mathf.Clamp(1 - playerRB.velocity.magnitude / velocityClamp, 0.1f, 10f);
        liftVector += CalculateWingDrag(liftPoint);
        playerRB.AddForce(liftVector);

        totalLiftVector += liftVector;
    }
}
```

Kuva 4. Siiven nostovoima.

CalculateAirLift-metodi palauttaa siiven noston suunnan vektorina, johon on laitettu optimaalisen siiven kulman ilmansuuntaan nähden olevat kertoimet. Oikeassa elämässä arvot ovat paljon tarkemmat kuin 90 astetta, mutta se tekee lentämisestä helpompaa pelaajalle. Tässä tilanteessa pelaaja saa eniten nostetta, kun hän osoittaa suoraan menosuuntaa kohti, ja noste vähenee, kunnes kulma on 90 astetta tai yli, milloin siivestä ei saa yhtään nostetta.

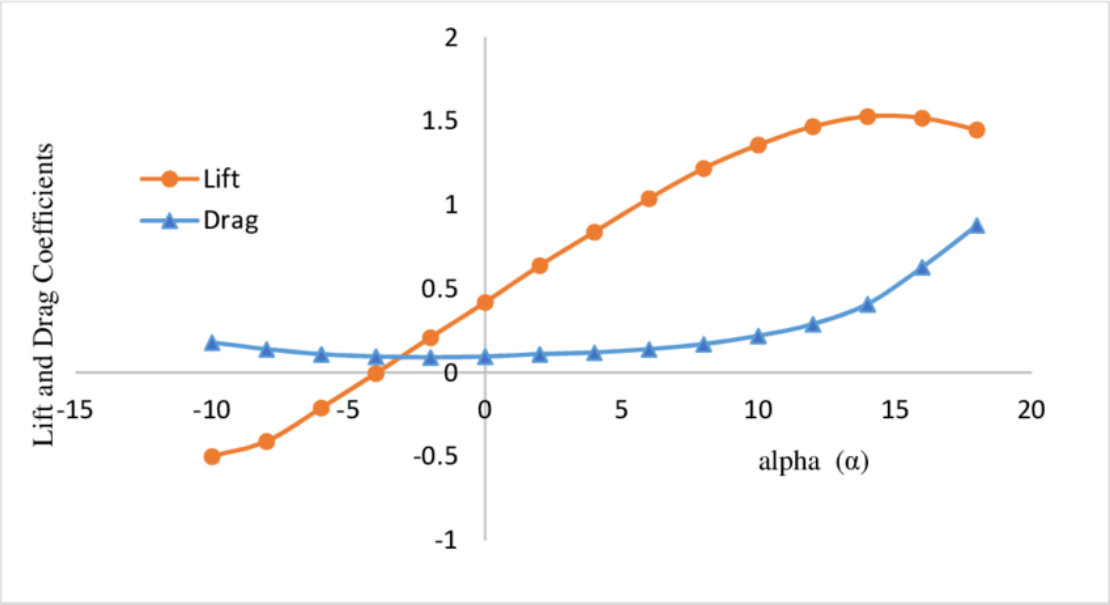
```
private Vector3 CalculateAirLift(GameObject liftPoint)
{
    float wingAngle = Vector3.Angle(playerRB.velocity, -liftPoint.transform.up);

    if (wingAngle > 90f) wingAngle = 90f;

    Vector3 liftVector = liftPoint.transform.forward * (1 - wingAngle / 90f);

    return liftVector
}
```

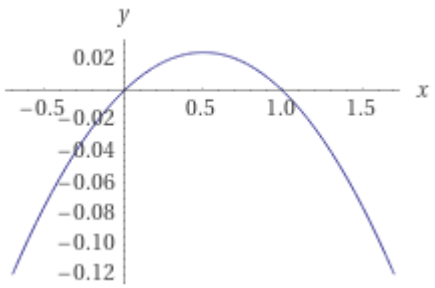
Kuva 5. Nostevektorin laskeminen.



Kuva 6. Esimerkki nostekertoimen ja ilmanvastuksen suhteesta siiven kulmaan ilmavirrassa. (Petinrin 2017).

AirDensity eli ilmantiheys voidaan muuttaa esimerkiksi laittamalla se maantasolla olemaan yksi ja korkeammalla ilmassa nolla, jolloin liitovoimalle on maksimikorkeus missä se toimii.

Nopeuden saa playerRB.velocity.magnitude -metodilla. Nopeuden kasvaessa nostokerroin kasvaa, jolloin sivistä saa enemmän voimaa irti, mutta kun nopeus kasvaa liikaa, siivistä ei saa enää tarpeeksi voimaa päästäkseen ilmanvastuksen antamasta voimasta yli. Unityssä ei ole ilmanvastusta, joten nopeus pitää rajoittaa muilla tavoilla. Nopeus kerrottuna Mathf.Clamp-metodilla ratkaisee ongelman, jossa pelaaja saa liikaa nostovoimaa korkeissa nopeuksissa. Tästä tulee pehmeä katto pelaajan suurimmalle nopeudelle, jota voi pelin aikana säädellä ja kokeilla, mikä arvo toimii parhaiten. Nopeuden funktiona rajoitus näyttää käänteiseltä paraabelilta (arvot eivät vastaa pelin arvoja).



Kuva 7. Nopeusrajoittimen funktion muoto.

PlayerRB.AddForce lisää pelaajaan lasketun voiman, ja lopuksi voima lisätään *totalLiftVector*-muuttujaan, jota hyödynnetään molempien siipien yhteisen nosteen visualisoinnissa

4.3.2 Siipien tukivoima

Kun pelaaja on vielä paikallaan, nostovoimalla ei ole mitään väliä, joten lennon aloittamiseen tarvitaan muunlainen voima, jotta pääsee ilmaan. Tämän voi mallintaa helposti lisäämällä ohjaimien nopeuden voimavektorina pelaajaan, jos liike on oikeaan suuntaan, eli kun siipiä tai ohjaimia räpyttää alaspäin, niin pelaaja saa ylöspäin sen verran voimaa, kuinka nopeasti siipiä on liikuttanut.

Toinen tapa lähteä lentoon voisi mahdollisesti olla vain kääntämällä ranteita edestakaisin, jolloin siipien pitäisi tuottaa voimaa, mutta koska se ei pelillisesti ole kiinnostavaa, ohjaimia pitää liikuttaa alaspäin, jotta pääse ilmaan.

```
private void WingForce(ControllerVelocity velocity, GameObject liftPoint)
{
    Vector3 controllerVelocity = velocity.GetControllerVelocity();

    if (controllerVelocity.y >= 0f)
        playerRB.AddForce(liftPoint.transform.forward * controllerVelocity.magnitude *
forceMultiplier);

}
```

Kuva 8. Siivenlyönnin voima.

5 Ongelmatilanteet ja niiden ratkominen

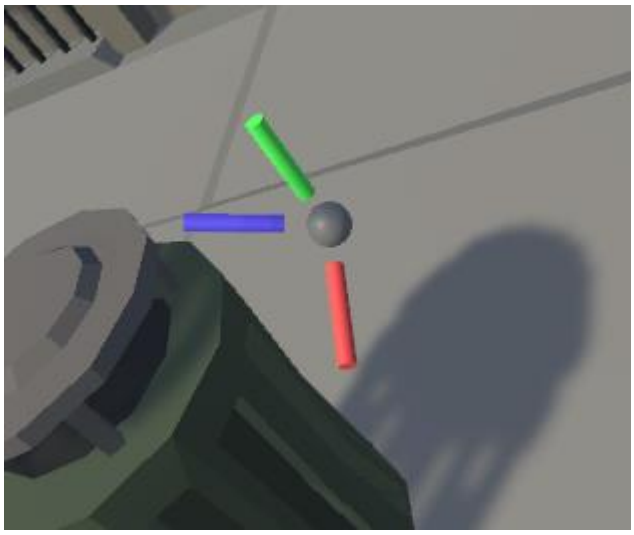
Muuttujia lentoon löytyy paljon, ja kaikilla on tärkeä ja huomattava vaikutus lentämisen tuntumaan. Oikeassa maailmassa on enemmän muuttujia, mutta niitä pystyy yksinkertaistamaan peliä varten, sillä vaikka moni kosmeettinen asia pelissä vastaisi jotain oikeassa elämässä. Loppujen lopuksi monet kaavat voi tiivistää vain yhteen muuttujaan, esimerkiksi siiven muoto vaikuttaa sen nosteen määrään, mutta pelissä voi laittaa vain nosteen määrän manuaalisesti.

On hankalaa testailla muuttujien arvoa, jos pitää aina hyppiä Unityn *editorin* ja pelin välillä. Siksi hyvä käytäntö sovellusta kehittäessä on jonkinlaisen *debugaus*-ikkunan tai välineen kehittäminen.

Kuva 9. Muuttujien säätämisen käyttöliittymä.

Pelissä on piilotettava ikkuna, jossa on liukusäätimiä muuttujille, joita ovat ilman tiheys, siipien iskuvoiman kerroin, nostovoiman kerroin, stabilointi, nopeusrajoitin ja ilmanvastuksen kerroin. Toinen hyvä apuri ongelmien huomaamiseen on pieni apuri, joka on pelaajan edessä oleva pallo, josta tulee kolme eriväristä nuolta, jotka osoittavat menosuuntaan, yhteislasketun voiman suuntaan, ja siipien tuottaman nostovoiman suuntaan. Vaikka numerot ja arvot voisivat auttaa

myös, apuri tuntuu luonnolliselta ja melkein sen avulla pystyisi jo lentämään, eri lennon tilanteet on mahdollista ymmärtää vain vilkaisemalla.



Kuva 10. Apuri voimien ja liikkeen suuntien visualisointi.

Kun kesken lennon pystyy kokeilemaan eri arvoja ja rajoja, oikean tuntuman löytäminen onnistuu huomattavasti nopeammin, ja muunlaisten ongelmatilanteiden huomaaminen on myös paljon helpompaa.

6 Päätäntö

Projektiin kuului paljon muutakin kuin fysiikkamallinnusta, mutta se oli selkeästi hankalin osuus. Projektin kehittäminen jatkuu vielä, ja toivon tekeväni siitä lopuksi viimeistellyn tuotteen, jonka voisi julkaista. Fysiikka oli projektin selkäranka, ja pelatessa lentäminen tuntuu mieluisalta ympäristöstä riippumatta.

Aerodynamiikan kaavan osia pystyy soveltamaan, esimerkiksi sukeltamissimulaattorin tekemiseen. Myös koodin pienempiä osia on mahdollista muuttaa hieman, jotta ne olisivat hyvä omiin käyttötarkoituksiinsa, jotka saattavat poiketa alkuperäisestä.

Vaikka tutkielman aihepiiri on hankala, jos ei ole ennen yrittänyt samankaltaisia ongelmien ratkaisua, toivon että lukemisen jälkeen lukijalla on enemmän itse-varmuutta yrittää kehittää jotain oman osaamistason yläpuolella olevaa projektia ja inspiroitua luomaan jotain fysiikkamallinnukseen perustuvaa.

Lähteet

Petinrin, Onoja 2017. Computational Study of Aerodynamic Flow over NACA 4412 Airfoil, British Journal of Applied Science & Technology. https://www.researchgate.net/figure/Graph-of-lift-and-drag-coefficient-versus-angle-of-attack-at-Re-6-x-10-6_fig3_317430315 (luettu 25.8.2022)

Time Manager. Unity Technologies a. https://docs.unity3d.com/Manual/class-TimeManager.html (luettu 19.9.2022)

Hodanbosi, Fairman 1997. Lift Formula. NASA. (luettu 19.9.2022) https://www.grc.nasa.gov/WWW/K-12/WindTunnel/Activities/lift_formula.html

Oyedoyin 2020, Unity Forums. https://forum.unity.com/threads/released-silantro-flight-simulator.522642/ (luettu 31.10)

Rigidbody. Unity Technologies b. https://docs.unity3d.com/Manual/class-Rigidbody.html (luettu 31.10)

Wood 2022, Aerodynamic Lift, Drag and Moment Coefficients, Aerotoolbox. https://aerotoolbox.com/lift-drag-moment-coefficient/ (luettu 28.11.2022)