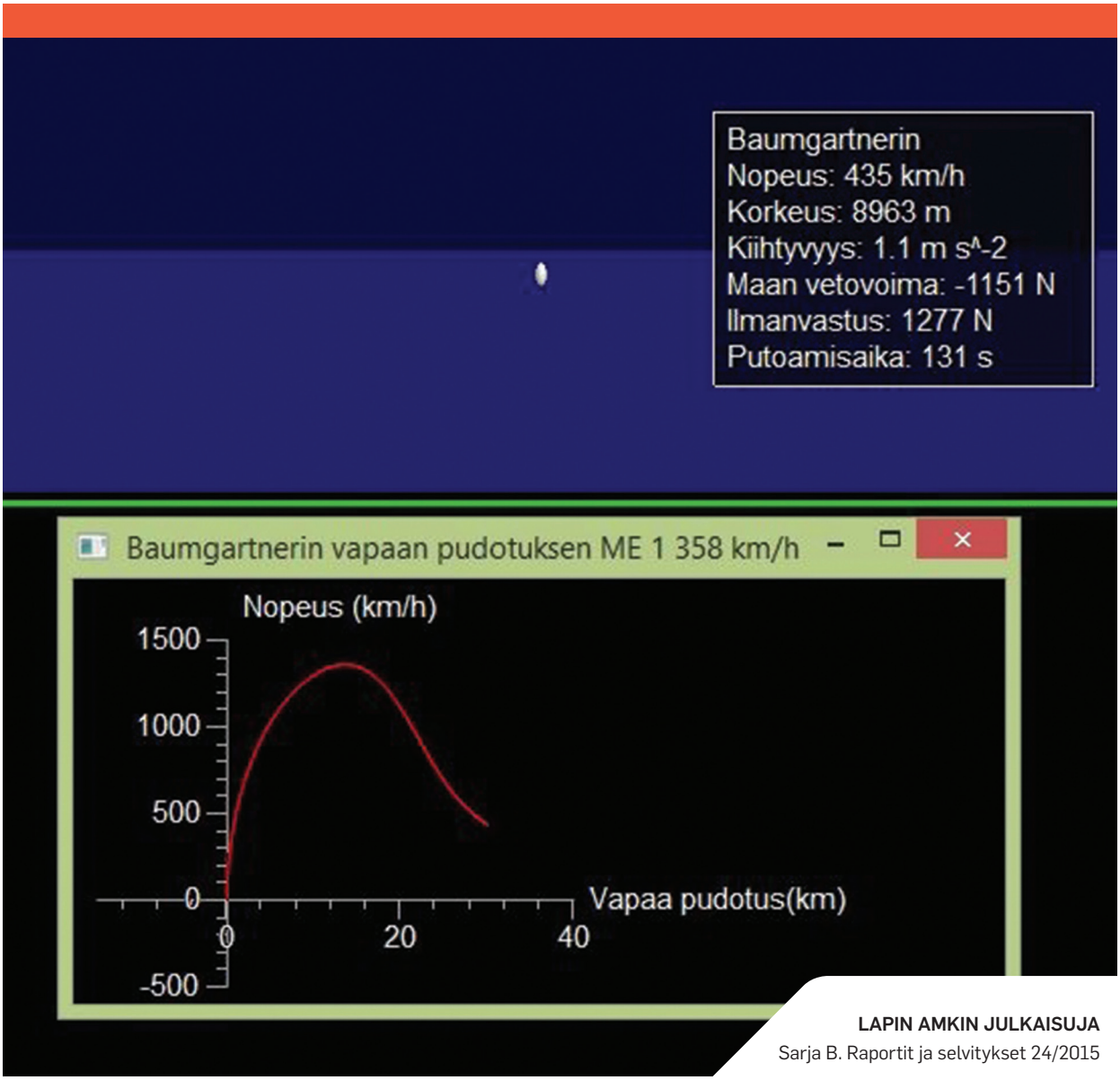
Kari Peisa

B

Pelifysiikka ja matematiikka CDIO-pohjaisessa oppimisessa



LAPIN AMKIN JULKAISUJA
Sarja B. Raportit ja selvitykset 24/2015

Pelifysiikka ja matematiikka CDIO-pohjaisessa oppimisessa

Kari Peisa, Lehtori Lapin AMK

Pelifysiikka ja matematiikka CDIO -pohjaisessa oppimisessa

Sarja B. Raportit ja selvitykset 24/2015

Lapin ammattikorkeakoulu
Rovaniemi 2015

ISBN 978-952-316-108-5 (pdf)
ISSN 2342-2491 (verkkojulkaisu)

Lapin ammattikorkeakoulun julkaisuja
Sarja B. Raportit ja selvitykset 24/2015

Kirjoittaja: Kari Peisa

Taitto: Lapin AMK, viestintäyksikkö

Lapin ammattikorkeakoulu
Jokiväylä 11 C
96300 Rovaniemi

Puh. 020 798 6000
www.lapinamk.fi/julkaisut

Sisällys

Esipuhe

Tämä selvitys on syntynyt Lapin ammattikorkeakoulun ohjelmistotekniikan laboratorion (pLab) vetämässä hankkeessa nimeltä *Peke*, jonka yhtenä tavoitteena on *peliteknologian osaamisen kasvattaminen Lapin AMK:n tieto- ja viestintätekniikan opettajien työkalupakissa*. Hanke on rahoitettu ELY:n (Elinkeino-, liikenne- ja ympäristö keskus) ESR (Euroopan unioni, Euroopan Aluekehitysrahasto, Euroopan Sosiaalirahasto) -kanavan kautta vuosien 2013 - 2015 välisenä aikana. Tämä selvitys tukee suoraan Peke-hankkeen tavoitteita ja on osa hankkeen tavoitteiden saavuttamiseen tarvittuja askelmia. Selvityksessä olevia peliteknologisia ratkaisuja hyödynnetään fysiikan taitojen oppimisessa ja soveltamisessa tieto- ja viestintätekniikan koulutusohjelmassa.

Tämä selvitys liittyy laajemmassa viitekehyksessä myös LUMA-aineiden integraatioon tietotekniikan insinöörikoulutuksen eri oppimisprojekteihin. Selvityksen taustalla on tarve tuoda esiin niitä mahdollisuuksia, joita LUMA-aineet edelleen tarjoavat laadukkaan insinöörikoulutuksen kehittämiselle.

1. Tiivistelmä

Tässä selvityksessä käsittelen pelifysiikkaa mahdollisuutena integroida matematiikan ja fysiikan opintoja ammattiaineiden CDIO-pohjaisiin oppimisprojekteihin, joiden ensimmäisiä kokemuksia Lapin ammattikorkeakoulussa on esitelty artikkelissa *Katsaus tieto- ja viestintätekniikan koulutuksen CDIO-projekteihin keväällä 2015* (Mielikäinen, Tepsa, 2015). Selvityksessä tuon esille pelifysiikan kursseilla käyttämiäni aihealueita, jotka mielestäni soveltuvat hyvin CDIO-pohjaisen oppimisnäkökulman toteuttamiseen. Aihealueet liittyvät niin kinematiikan ja dynamiikan perusteisiin kuin myös vaativampaan luonnon ilmiöiden mallintamiseen. Selvityksessä esittelen myös pelimoottoreiden käyttämän törmäyksien hallinnan fysiikan teoreettista taustaa, joka soveltuu edistyneemmän opiskelijan oppimiskohteeksi. Tätä oppia voi hyödyntää esimerkiksi sellaisten pelituotteiden valmistamisessa, joissa peliobjektien tarkka luonnonmukainen käyttäytyminen edellyttää fysiikan mallien ohjelmoimista.

Selvityksen jälkimmäisessä osassa käsittelen myös pelifysiikan ohjelmointiympäristöjä. Pohdintojen taustalla on integroituun fysiikan, matematiikan ja ohjelmoinnin oppimiseen liittyvä ongelma: pitääkö valita mahdollisimman valmis fysiikkamoottorilla varustettu pelinkehitysalusta vai mahdollisimman kevyt vaihtoehto, missä joutuu itse rakentamaan fysiikkamoottoria.

2. Johdanto

Ammattikorkeakoulujen tietotekniikan tai tieto- ja viestintätekniikan insinöörikoulutus on suurten muutosten kourissa. Työelämälähtöisyys ja jatkuvasti kehittyvät uudet substanssialueet sekä myös uudet oppimisnäkemykset ja –ympäristöt kuten CDIO-pohjaiset projektit ovat tuoneet sellaisen myllerryksen opetuksen suunnitteluun, että varsinkin perinteisten LUMA-aineiden opettajaa moinen on hirvittänyt. Samalla matematiikkaan ja fysiikkaan pohjautuva oppimisen perusta on menettänyt asemaansa - myös monilla muilla ammattikorkeakoulujen insinöörikoulutuksen aloilla.

Onko peliteollisuudessa ratkaisu LUMA-aineiden integraatioon tietotekniikan insinöörikoulutukseen?

Pelifysiikka on peliteollisuudessa käytettävien pelinkehitysalustojen pelimoottoreiden perusta. Kaikki ratkaisut, joilla simuloidaan tarkasti ja oikein reaalimaailman ilmiöitä, perustuvat fysiikan mallien matemaattisten ratkaisujen ohjelmoimiseen. Opetushallitus järjesti vuoden 2015 aikana ennakointihankkeen peliteollisuuden osaamistarpeen kartoittamiseksi. Julkaistusta raportista (Taipale-Lehto U., Vepsäläinen J, 2015) ilmenee, että Suomen yliopistoissa ja ammattikorkeakouluissa järjestetään pelialan koulutusta jo hyvin laajasti. Suurennuslasilla saa raportista kuitenkin etsiä mainintaa, josta ilmenisi tarvetta pelimoottoreiden fysiikan osaamiselle tai jopa kehittämiselle Suomen peliteollisuudessa. Esitetty kysymys jää siis ainakin vielä avoimeksi.

Aika monissa tietotekniikkaa opettavissa koulutusohjelmissa on jo siirrytty entisestä ”paperifysiikasta” tietokoneilla suoritettaviin simulointiharjoituksiin. Samalla fysiikan kurssin sijasta on alettu käyttää nimeä pelifysiikka, joka sopiikin paremmin uusiin projektiluonteisiin oppimismenetelmiin. Pelifysiikkaan siirtyminen tarkoittaa myös muutosta siinä, minkä tyyppisiä ongelmia opiskelijat joutuvat ratkomaan fysiikan opiskelussaan. Fysiikan käsitteiden paperiharjoittelun sijasta pelifysiikassa joudutaan ratkomaan myös oppimisympäristöön tai ohjelmointiin liittyviä ongelmia. Tämä muutos näkyy myös fysiikan aihepiirien muuttumisena toisaalta lähemmäs todellisia reaalimaailman ilmiöitä, mutta toisaalta teorian kattavuuden heikentymisenä.

Fysiikan opettajilta muutos on vaatinut täysin uudenlaisten opiskelumateriaalien suunnittelua ja valmistamista sekä tietenkin omien ohjelmointitaitojen petraamista.

Selvityksen ensimmäisen osa käsittelee pelifysiikan opiskeluun soveltuvia kohteita sekä tarvittavaa matematiikan, fysiikan ja ohjelmoinnin teoreettista pohjaa niin aloittaville kuin myös jo edistyneimmille opiskelijoille. Jälkimmäisessä osassa tarkastellaan pelifysiikan ohjelmointiin liittyviä periaatteita ja ratkaisumalleja. Tarkastelu perustuu kahden tyystin erilaisen ohjelmointiympäristön (VPython, Unity) käyttöönottoon liittyvien vaatimusten ja toisaalta mahdollisuuksien esittelyyn. Ensimmäinen ympäristö VPython edustaa hyvin kevyttä ohjelmistoympäristöä, jonka käyttöönotto soveltuu erityisesti aloitteleville pelisimulaatioiden opiskelijoille. Jälkimmäinen Unity puolestaan on erittäin kehittynyt valmiin fysiikkamoottorin omaava kaupallinen pelinkehitysalusta. En tarkastele Unitya tässä selvityksessä pelintuottamisvälineenä, vaan sen soveltuvuutta pelifysiikan oppimisympäristöksi. Molemmat ympäristöt ovat avoimeen lähdekoodiin perustuvia ohjelmistoja ja siten opiskelijoilla vapaasti käytettävissä. Oppilaitos joutuu kuitenkin lunastaman käyttöoikeuden Unityyn maksullisella lisenssillä.

3. Pelifysiikka = Matematiikan, fysiikan ja ohjelmoinnin CDIO-pohjaista oppimista

Parhaimmillaan pelifysiikan opiskelussa toteutuu hyvin CDIO –pohjainen oppimisnäkökulma **C**onceiving — **D**esigning — **I**mplementing — **O**perating real-world systems and products. Harjoittelun tehostamiseksi opettaja joutuu valmistelemaan materiaalia varsinkin alussa suoraan **I**mplementing vaiheeseen. LUMA-aineiden oppimisen kannalta ihanne on se, että opiskelijat kykenevät valmistamaan pelituotteen tai aihion, johon on itse suunniteltu ja rakennettu fysikaalinen käyttäytyminen. Tällöin tuotteet ovat enemmän todellisten reaalimaailman tapahtumien mahdollisimman tarkkoja simulaatioita – mitä ei kaupallisissa peleissä useinkaan kohtaa. Implementing –vaihe edellyttää tällöin opiskelijalta jo kohtalaiset perustaidot ja tiedot niin matematiikasta, fysiikasta kuin myös ohjelmoinnista. Ongelmaksi muodostuukin helposti se, ettei noita perustaitoja ehditä oppia laisinkaan, mikäli siirrytään liian aikaisin valmiiden tuotteiden rakentamiseen.

3.1 PELIFYSIIKKA ENSIMMÄINEN FYSIIKAN KURSSI

Kun pelifysiikan kurssi on opiskelijoiden ensimmäinen insinööriopintojen fysiikan kurssi, on useimmilla opiskelijoilla tilanne se, että aikaisempia fysiikan opintoja on hyvin niukasti. Toisaalta joukossa on myös niitä, jotka ovat suorittaneet lukiossa kaikki fysiikan mahdolliset kurssit. Tähän tilanteeseen pelifysiikan CDIO-pohjainen toteuttaminen sopii kuitenkin paremmin kuin perinteinen ”paperifysiikka”, sillä CDIO antaa kaikille mahdollisuuden toteuttaa osaamistaan enemmän omalla tasollaan ja tavallaan.

Fysiikan suureiden ohjelmoinnin perusta on vektorilaskennassa ja oikeiden yksiköiden hallinnassa. Vektoreiden ohjelmoimisessa tärkeää on kyetä erottamaan koodissa vektorimuuttujat skalaarimuuttujista. Kirjallisissa esityksissä kuten tässäkin vektorisymbolit yleensä lihavoidaan. Koodissa erottamisen voi halutessaan tehdä erilaisella nimeämiskäytänteellä. Pelifysiikka edellyttää käytännössä vektorilaskennan taitojen opiskelua ennen kurssia.

Ensimmäisen pelifysiikan kurssin yksi minimitavoite fysiikan ja matematiikan osalta on, että opiskelija kykenee mallintamaan pistemäisen kappaleen erilaisia liikeratoja kinematiikasta ja dynamiikasta johdettujen vektoriesityksien avulla. Kun puhutaan pistemäisesti käyttäytyvästä kappaleesta, tarkoitetaan silloin sitä, että kaikki kappaleeseen kohdistuvat voimavaikutukset tai voimien impulssit kohdistuvat aina kappaleen massakeskipisteeseen. Tällöin ei käsitellä lainkaan kappaleen pyörimiseen vaikuttavia momentteja.

Animaatiolla tarkoitan tässä esityksessä liikkeen kuvaamista ratakäyrän parametrimuotoisten vektorifunktioiden avulla erotukseksi *simulaatiosta*, jolla tarkoitan liikkeen implementoimista soveltamalla nk. **Eulerin menetelmää** (Euler's Method). Simulaatiolla opiskelija kykenee mallintamaan myös jatkuvasti muuttuvien voimien aiheuttamia kappaleiden liikkeitä kuten esimerkiksi ilmanvastuksen vaikutusta. Eulerin menetelmä dynamiikan ohjelmoinnissa on ns. ikuinen luuppi, jonka algoritmi perustilanteessa on yksinkertaisuudessaan seuraava:

1. Ratkaise kullakin ajanhetkellä kappaleeseen vaikuttava kokonaisvoima ja laske sitä vastaava dynamiikan peruslain mukainen kiihtyvyys ($\sum \boldsymbol{F} = m\,\boldsymbol{a} = m\,\ddot{\boldsymbol{x}}$)
2. Approksimoi kiihtyvyyden avulla riittävän pienen ajanhetken *dt* kuluttua kappaleen uusi nopeus (v+= **a** dt)
3. Approksimoi uuden nopeuden avulla ajanhetken *dt* kuluttua kappaleen uusi paikka olettaen nopeus vakioksi (x+ = v dt

Kolmannessa kohdassa vakionopeuden käyttäminen tuottaa käytännössä saman tuloksen, kuin jos käyttäisi tasaisesti kiihtyvää liikettä. Tämä johtuu siitä, että kiihtyvyyden aiheuttama lisäys eli termi $\frac{1}{2}\boldsymbol{a}\,dt^2 \ll \boldsymbol{v}\,dt,$ kun dt on pieni.

Eulerin menetelmän perusta on puhtaasti differentiaalilaskennassa. Sillä voidaan approksimoida tuntemattoman funktion arvoja siirtymällä pienin askelin eteenpäin aina, kun edellisessä kohdassa tiedetään jotain funktion arvosta ja sen derivaatasta. Eulerin menetelmällä voidaan simuloida monia sellaisia ratkaisuja, joille ei yksinkertaisesti ole olemassa mitään analyyttista ratkaisufunktiota. Jos simulaatiossa halutaan parantaa approksimoinnin tarkkuutta, on mahdollista käyttää parenneltua Eulerin menetelmää tai muita menetelmiä, joista tunnetuin on Runge-Kutta menetelmä. On huomioitavaa, että Eulerin menetelmän käyttö ei edellytä opiskelijalta aikaisempaa differentiaalilaskennan pohjaa.

Alla on lueteltu joitakin esimerkkejä matematiikan menetelmien hyödyntämisestä fysiikan mallien ohjelmoinnissa. Ensimmäisellä pelifysiikan kurssilla tärkeänä tavoitteena olisi ymmärtää vektoreiden pistetulon ja ristitulon soveltamisen mahdollisuuksia käsiteltäessä kappaleiden dynamiikkaa.

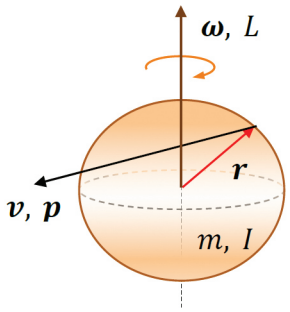
- Analyyttisen 3D geometrian käsitteiden kuten esimerkiksi suoran, ympyrän ja tason esittäminen ja käsitteleminen vektorisuureina mm. ratakäyrien vektorifunktioissa
- Kiertomatriisien käyttö haluttaessa esimerkiksi kallistaa ratakäyräesityksiä kuten vaikka kuun kiertoradan tarkemmissa animaatioissa (vaihtoehtoisesti kvaternionien käyttö)
- Vektoreiden pistetulon soveltaminen mm. pallomaisen kappaleen kitkattomissa törmäyksissä erisuuntaisille tasoille (myös impulssiperiaate kts. seur. kpl)
- Vektoreiden ristitulon käyttö simuloitaessa varatun hiukkasen liikettä magneettikentässä, tai simuloitaessa vaikka Magnus-efektin vaikutusta pyörivän pallon liikeradan kiertymiseen ilmanvastuksen alaisuudessa (http://en.wikipedia.org/wiki/Magnus_effect).

Edellä mainittujen fysiikan ja matematiikan menetelmien ohjelmointi animaatioissa ja simulaatioissa edellyttää oppimisympäristöstä riippuen vähimmillään vain perusohjausrakenteiden haarautuma (if) ja toisto (while) hallintaa. Kun käytetään jotain dynaamista ohjelmointikieltä kuten Python, muuttujien tyyppejä ei tarvitse esitellä, mikä helpottaa myös alkuun pääsemistä. Simulaatiossa tarvittavat objektit ja grafiikka pitää saada ohjelmointiympäristössä helposti käyttöön. Artikkelin jälkimmäisessä osassa tarkastelen ohjelmoinnin näkökulmaa tarkemmin. Fysiikan opiskelussa ongelman ratkaiseminen tulee keskittää fysiikan rakenteiden ohjelmointiin, jolloin harjoitteet on pohjustettava osittain valmiilla koodirakenteilla. Aika pian joudutaan kuitenkin tekemisiin sellaisten simulaatioiden perusongelmien kanssa, kuin että kappaleet saattavat jumittua kiinni toisiinsa tai sitten syöksyä seinämien läpi. Nämä ovat tyypillisiä esimerkkejä algoritmiin liittyvistä ongelmista ja ne voidaan ratkaista ohjelmallisesti, kun analysoidaan mistä käyttäytyminen johtuu. Yleinen ratkaisu mm. törmäyksissä on siirtää kappaleetta esimerkiksi ”yksi aika-askel taaksepäin” ennen uuden nopeuden soveltamista. Animaatioiden ja simulaatioiden näyttävyyden kehittämisessä joudutaan perehtymään valitun kielen rakenteisiin syvällisemmin ja ottamaan käyttöön erilaisia oppimisympäristöön sisäänrakennettuja ominaisuuksia.

3.2 PELIFYSIIKKAA EDISTYNEEMMÄLLE OPISKELIJALLE

Jäykkien kappaleiden törmäyksien hallinta on pelimoottoreiden hyvin edistynyttä ja vaativaa teknologiaa, jossa opiskeluun voidaan määrittää useita osaamistasoja. Reaalimaailmassa törmäyksen aikana kappaleisiin vaikuttavat voimat ovat hyvin suuria ja ne vaikuttavat hyvin lyhyellä aikavälillä. Todellisuudessa kappaleissa tapahtuu tällöin aina myös pieniä palautuvia muodonmuutoksia. Näiden muutosten käsitteleminen pitäisi toteuttaa yhden tai korkeintaan muutaman aika-askeleen aikana, mikä käytännössä vaatii aivan liikaa laskenta-aikaa. Törmäyksien simuloinnissa ei dynamiikan peruslain soveltaminen onnistukaan, vaan nopeuden muutokset on määriteltävä tapahtumaan hyvin äkillisesti yhden ”aika-askeleen” aikana, mikä voidaan tehdä **impulssiperiaatteen** menetelmällä (seur. kpl).

Jäykkien kappaleiden törmäyksien hallinta alkaa siitä, että tietää ja ymmärtää kappaleen pyörimiseen liittyvät peruskäsitteet vektorisuureina. Kuvassa 1. on jäykän kappaleen törmäyksien hallinnan kannalta oleelliset suureet merkitty siten, että niiden vektoriesityksen vaikutussuunta ilmenee kuvaan piirretystä vektorista. Näillä suureilla kyetään hallitsemaan jäykkien kappaleiden törmäyksiä, kun otetaan käyttöön **impulssiperiaate**.



ω Kulmanopeus

L Liikemäärämomentti

m Kappaleen massa (skalaari)

I Kappaleen hitausmomentti (skalaari)

r törmäyskohdan ja massakeskipisteen välinen vektori

v törmäyskohdan nopeus maailman koordinaatistossa

p liikemäärä

Kuva 1. Jäykän kappaleen törmäyksien hallinnassa tarvittavat fysiikan suureet

Nopeuteen vaikuttava vektorisuure **impulssi J** määritellään kappaleen ulkoisen voiman **F** lyhyessä ajassa Δt aiheuttamana liikemäärän **p** = m **v** muutoksena.

$$\mathbf{J} = \mathbf{F}\,\Delta t = m\,\Delta \mathbf{v}$$

Kulmanopeuteen vaikuttava **impulssin momentti** lasketaan ristitulolla $\mathbf{r} \times \mathbf{J}$, missä **r** on törmäyskohdan ja massakeskipisteen välinen vektori. Impulssin momentti määritellään pyörimisliikkeen liikemäärämomentin **L**= I **ω** muutoksena.

$$\mathbf{r} \times \mathbf{J} = I\,\Delta \boldsymbol{\omega}$$

missä skalaarisuure on kappaleen hitausmomentti (kts. kpl 3.2.2) pyörimisakselin suhteen.

Impulssiperiaatteen idea on siinä, että impulssille pyritään löytämään ratkaisu ilman, että tarvitsee tarkastella voimaa **F**. Jäykille puhtaasti pyöriville kappaleille löytyy tähän ratkaisu, joka perustuu **liikemäärän ja liikemäärämomentin säilymislakeihin**. Puhtaasti pyörivällä kappaleella jokainen massapiste pyörii ympyräradalla ja jokaisen ympyrän keskipisteet sijaitsevat samalla kiinteällä suoralla eli pyörimisakselilla.

3.2.1 Pistemäisten kappaleiden törmäys

Pallomaisen kappaleen törmäys ilman kitkaa käsitellään periaatteessa samalla tavalla kuin pistemäisen kappaleen törmäys. Tällöin kappaleiden pyörimistä ei tapahdu tai se ei muutu törmäyksissä. Kun törmäyksessä ei vaikuta kitkavoimia, kappaleiden saamat impulssit vaikuttavat törmäyskohdassa vain pinnan normaalin $\boldsymbol{n}$ suuntaisesti. Voiman ja vastavoiman lain perusteella ne ovat yhtä suuria mutta vastakkaissuuntaisia. Tällöin impulssi voidaan merkitä vektorina $j\,\boldsymbol{n}$, missä j on skalaari. Jos $\boldsymbol{n}$ on yksikkövektori, antaa j myös impulssin fysikaalisen suuruuden.

Huom! Seuraavissa kaavoissa normaalin $\boldsymbol{n}$ oletetaan olevan yksikkövektori.

Liikemäärän säilymislain mukaan kummankin kappaleen törmäyksen jälkeinen liikemäärä on yhtä suuri kuin sen alkuperäisen liikemäärän ja törmäyksessä saadun impulssin j **n** summa.

$$\boldsymbol{p} \mathrel{+}= j\,\boldsymbol{n} \tag{1}$$

Edellä ja myös jäljessä tulevissa yhtälöissä merkintä += tulee ohjelmointikielistä ja tarkoittaa alkuperäisen arvon päivittämistä. Impulssin etumerkki määräytyy sen mukaan miten normaali on valittu. Koska $\boldsymbol{p} = m\,\boldsymbol{v}$, saadan edellisestä kappaleen nopeuden muutos jakamalla massalla m.

$$\boldsymbol{v} \mathrel{+}= \frac{j}{m}\,\boldsymbol{n} \tag{2}$$

Törmäyksien kimmoisuuden (elastisuuden) astetta hallitaan sysäyskertoimella e ϵ[0,1], joka kertoo törmäyksen jälkeisen kappaleiden nopeuseron $u_2 - u_2$ suhteen ennen törmäystä olleeseen nopeuseroon $v_2 - v_1$.

$$u_2 - u_2 = e\,(v_2 - v_1) \tag{3}$$

Täysin kimmoisalla törmäyksellä ja täysin kimmottomalla törmäyksellä e = 0. Sysäyskerroin ilmaisee samalla sen, että osa kineettisestä energiasta katoaa osittain kimmoisassa törmäyksessä. Sysäyskerroin on määritettävä kulloinkin kokeellisesti. Impulssin laskentakaava ratkaistaan soveltamalla molempiin kappaleisiin yhtälöitä (2) ja (3). Laskentakaavan johdon voi kuitenkin jättää käsittelemättä ilman, että mitään fysiikan keskeistä periaatteetta pitäisi jättää huomiotta.

$$j = \frac{-(e+1)(\boldsymbol{v_2}-\boldsymbol{v_1})\cdot\boldsymbol{n}}{\frac{1}{m_1}+\frac{1}{m_2}} \tag{4}$$

Impulssin laskentakaavassa (4) esiintyvät törmäävien kappaleiden nopeudet juuri ennen törmäystä $\boldsymbol{v}_1$ ja $\boldsymbol{v}_2$, massat m_1 ja m_2, yksikkönormaali $\boldsymbol{n}$ sekä sysäyskerroin e.

Nopeuden päivitykset tehdään nyt yhtälön (2) mukaisesti. Tapauksessa, jossa kappale törmää massiiviseen $(m \cong \infty)$ esteeseen, voidaan nimittäjässä jälkimmäinen termi jättää poiskin, sillä $\frac{1}{\infty} = 0$.

3.2.2 Jäykän kappaleen törmäys

Hitausmomentti on suure, joka ilmaisee kappaleen kyvyn vastustaa pyörimisnopeuden muutoksia. Sen suuruuteen vaikuttavat kappaleen massa ja sen sijoittuminen pyörimisakseliin nähden. Kun kappale pyörii puhtaasti, on sen hitausmomentti pyörimisakselin suhteen vakio. Esimerkiksi pallon hitausmomentti keskipisteen kautta kulkevan pyörimisakselin suhteen lasketaan kaavalla

$$I = c\, m\, r^2$$

missä kerroin $c = \frac{2}{5}$ umpinaiselle ja $c = \frac{2}{3}$ ohutseinäiselle pallolle.

Kun törmäävissä pinnoissa vaikuttaa kitka, saadaan kokonaisimpulssi normaalin suuntaisen impulssin ja kitkavoiman aiheuttaman törmäyspinnan tangentiaaliseen tasoon suuntautuvan impulssin summana. Yleisesti kumpikin impulssi voi vaikuttaa sekä lineaarisen liikemäärän että liikemäärämomentin muutokseen. Pallon muotoisilla kappaleilla kuitenkin normaalin suuntainen impulssi vaikuttaa vain kappaleiden lineaariseen liikemäärään (yhtälö (1)) eli kappaleen etenemiseen liittyvään liikemäärään. Tangentiaalinen kitkan aiheuttama impulssi muuttaa sekä liikemäärää että liikemäärämomenttia ja siten myös kulmanopeutta $d\boldsymbol{L} = I\, d\boldsymbol{\omega}$.

Liikemäärämomentin säilymislain mukaisesti saadaan yhtälöä (1) vastaava yhtälö (5) liikemäärämomentin muutokselle. Kokonaisimpulssi muodostuu normaalin suuntaisen impulssin ja kitkan aiheuttaman tangentiaalisen tason suuntaisen impulssin summasta (Kuva 2.). **Liikemäärämomentin säilymislain** mukaan kummankin kappaleen törmäyksen jälkeinen liikemäärämomentti on yhtä suuri kuin sen alkuperäisen liikemäärämomentin ja törmäyksessä saadun impulssin momentin summa.

$$\boldsymbol{L} += \boldsymbol{r} \times (\boldsymbol{J}_n + \boldsymbol{J}_t) \qquad (5)$$

Koska $\boldsymbol{L} = I\, \boldsymbol{\omega}$, saadaan edellisestä kulmanopeuden muutos jakamalla hitausmomentilla I.

$$\boldsymbol{\omega} += \frac{1}{I}(\boldsymbol{r} \times (\boldsymbol{J_n} + \boldsymbol{J_t})) \qquad (6)$$

Normaalin suuntaisen ja tangentiaalisen impulssin ratkaiseminen nopeuden ja kulmanopeuden muutosyhtälöistä (2) ja (6) sekä sysäyskertoimen määrittelystä (3) on hyvin vaativaa symbolista laskentaa. Laskentakaavan johtaminen riippuu myös siitä kuinka realistisesti törmäystä halutaan mallintaa (Hecker C., 1997). On huomioitava, että pintojen törmäysnopeudet tulee määritellä valitun maailmankoordinaatiston

mukaisesti siten, että ne muodostuvat massakeskipisteen nopeuden ja törmäyskohdan kulmanopeutta vastaavan ratanopeuden summasta.

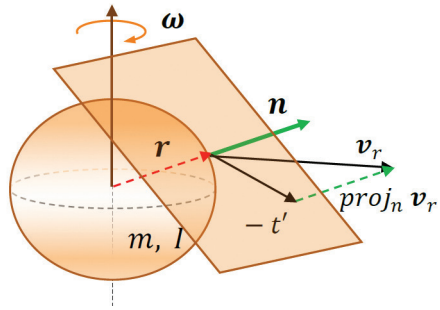
$$v^* = v + \omega \times r$$

Erilaisten esitysmuotojen ekvivalenttisuudet ja muut ominaisuudet on hyvä tutkimuskohde edistyneelle matematiikan taitajalle esimerkiksi opinnäytetyössä.

Impulssin suuruudelle esitetään yleensä yhtälön (7) mukainen esitys (Burg & Bywalec, 2013, ss. 115, Wikipedia https://en.wikipedia.org/wiki/Collision_response).

$$j = \frac{-(e+1)v_r \cdot n}{\frac{1}{m_1}+\frac{1}{m_2}+\left(\frac{1}{I_1}(r_1 \times n)\times r_1+\frac{1}{I_2}(r_2 \times n)\times r_2\right)\cdot n} \qquad (7)$$

Yhtälössä (7) esiintyvät törmäyspintojen suhteellinen nopeus ennen törmäystä, kappaleiden massat ja , (skalaariset) hitausmomentit ja , törmäyskohdan ja massakeskipisteen väliset etäisyysvektorit ja , yksikkönormaali , sekä sysäyskerroin . Pallon törmätessä toiseen palloon nimittäjässä olevat ristitulolausekkeet häviävät. Tämä johtuu siitä, että etäisyysvektorit ja ovat yhdensuntaisia normaalivektorin kanssa. Hecker esittää artikkelissaan *Physics, Part 4: The Third Dimension* (Hecker C., 1997) hieman erilaisen mutta käytännössä saman esitysmuodon, kun huomioi, että hänellä normaalivektori ei ole yksikkövektori.



Kuva 2. Kitkan aiheuttaman impulssin suunnan t' määräytyminen tangentiaalisella tasolla, kun v_r on törmäävien pintojen suhteellinen nopeus juuri ennen törmäystä.

Tangentiaalisen impulssin suunta vaikuttaa törmäyspinnan tangentiaalisessa tasossa ja se on päinvastainen pintojen välisen suhteellisen nopeuden projektiovektoriin nähden (Kuva 2.). Merkitään

$$v_r = v_2^* - v_1^* \qquad (8)$$

jolloin saadaan tangentiaalisen impulssin suunnan yksikkövektori vektoriprojektion $(v_r \cdot n)\, n$ avulla kuvan 2. mukaisesti

$$t' = -(v_r - (v_r \cdot n)\, n), \qquad t = t'/|t'| \qquad (9)$$

Burg ja Bywalec esittävät kirjassaan (Burg & Bywalec, 2013, ss. 118) tangentiaalisen suuntavektorin laskemiseksi yhtälöä

$$\boldsymbol{t' = (n \times v_r) \times n,}$$

mikä on kuitenkin sama esitys kuin yhtälö (9), kun huomioidaan vektoriristitulon kolmitulon yleinen ominaisuus:

$$\boldsymbol{(a \times b) \times c = (a \cdot c)\, b - (b \cdot c)\, a}$$

Tangentiaalisen impulssin suuruutta voidaan säätää pintojen välisen kitkakertoimen avulla. Koska kitkavoiman ja pintojen välisen normaalin suuntaisen puristusvoiman välillä on riippuvuus

$$F_\mu = \mu F_n,$$

saadaan vastaava verranto myös tangentiaalisen impulssin ja normaalin suuntaisen impulssin suuruuksien välille

$$J_t = \mu |J_n|$$

Todellisuudessa törmäystilanteessa mm. muodonmuutokset aiheuttavat erisuuntaisten voimien syntymistä tavalla, jossa tangentiaalisen impulssin suuruutta ei voi kuvata yhden kitkakertoimen avulla (Hecker C., 1997). Kuitenkin peleissä usein pallon kierteen ja kitkan hyödyntäminen on aivan avainasia, jolloin niille pitää löytää jokin ohjelmointiratkaisu.

Nyt voimmekin kirjoittaa lopulliset päivitysyhtälöt massakeskipisteen nopeudelle sekä kulmanopeudelle

$$\boldsymbol{v} \mathrel{+}= \frac{1}{m}(j\,\boldsymbol{n} + \mu\,|j|\,\boldsymbol{t}) \qquad \textbf{(10)}$$

$$\boldsymbol{\omega} \mathrel{+}= \frac{1}{I}(\boldsymbol{r} \times (j\,\boldsymbol{n} + \mu |j|\,\boldsymbol{t})\,), \qquad \textbf{(11)}$$

missä *j* lasketaan yhtälöllä (7). Impulssien etumerkkien kanssa tulee olla tarkka, vaikka vääränmerkkinen impulssi näkyykin heti simulaatiossa. Pallon tapauksessa impulssin j $\boldsymbol{n}$ voi poistaa yhtälöstä (11), koska normaalin suuntaisen impulssin ristitulo vektorin $\boldsymbol{r}$ kanssa on 0.

Edellä esitetyt törmäyksien hallintatyökalut riittävät simuloimaan tarkasti mm. erilaisia pallopelejä, joissa halutaan mallintaa myös pallojen kierteiden ja kitkan vaikutukset. Yleisempi tapaus, missä kappaleet eivät ole palloja, mutta jotka pyörivät kuitenkin puhtaasti, vaatii hitausmomentin laskemista kussakin törmäystilanteessa erikseen. Näissä tilanteissa hitausmomentti riippuu törmäyskohdan sijainnista kappaleessa. Laskennassa käytetään erityistä inertiamatriisia (Inertia matrix tai Inertia tensor). Menetelmän laskentaperusteet ja jopa valmiita funktioita löytyy

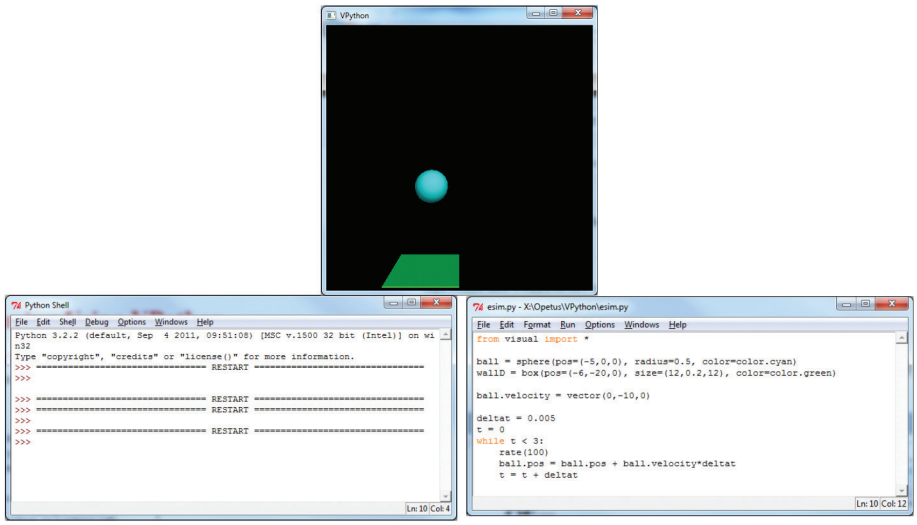
inertiamatriisia käsittelevistä alan teoksista (Lengyel, 2012, ss. 414-430, Burg & Bywalec, 2013 ss. 24-29). Inertiamatriisin tapauksessa hitausmomentin käänteisluku yhtälössä (7) tulee laskea käänteismatriisina. Kehittyneiden pelirakennusalustojen fysiikkamoottoreihin kuten Unity on implementoitu törmäysten hallintaa myös yleisessä tapauksessa.

4. Pelifysiikan ohjelmoitiympäristö

Seuraavissa kappaleissa esittelen esimerkein pelifysiikan ohjelmoinnin opiskeluun soveltuvia harjoituksia sekä esitän kahden tyystin erilaisen pelifysiikan ohjelmointiympäristön käyttöönottoon liittyviä huomioita. Käsittelemäni ohjelmointiympäristöt edustavat mielestäni kahta ääripäätä. Toinen on mahdollisimman helppo käyttöön otettavaksi ja ilman valmista fysiikkamoottoria oleva ympäristö, joka koostuu kahdesta osasta: Python-tulkki versio 2.7.9 ja David Schererin vuonna 2000 kehittämä Python kielinen grafiikkamoduuli VPython. Molemmat saadaan ladattua osoitteesta http://vpython.org/. Toinen on kaupallinen pelinkehitysalusta Unity, jossa on myös hyvin pitkälle kehitetty valmis fysiikkamoottori, ja joka saadaan ladattua osoitteesta http://unity3d.com/unity. Kumpikin voidaan ladata ja asentaa omalle koneelle ilmaiseksi, mutta oppilaitos joutuu kuitenkin ostamaan lisenssin Unitylle.

4.1 PYTHON-YMPÄRISTÖN EDUT JA HAITAT

Python kieli on dynaamisesti tyypittävä ohjelmointikieli, joka tukee proseduraalista ja oliopohjaista ohjelmointiparadigmaa. Grafiikkamoduuli VPython tuo ympäristöön paljon helposti käyttöön otettavaa toiminnallisuutta. Kineettisten 3D animaatioiden tai simulaatioiden esittäminen graafisesti voidaan toteuttaa helposti ilman ohjelmointikielen perusteellista hallintaa. Ympäristö koostuu vain kolmesta osasta: taustalla olevasta Python Shell komentotulkista, koodieditorista ja grafiikkaikkunoista, joissa ohjelman ajot esitetään (Kuva 3.)



Kuva 3. Python ympäristön osat grafiikkaikkuna, komentotulkki ja koodieditori

Python ympäristön perusominaisuudet voidaan ymmärtää parhaiten tarkastelemalla pientä ohjelmaesimerkkiä, jonka koodi on alla. Tyypillisesti tällainen harjoite annetaan opiskelijoille osittain valmiiksi koodattuna. Esimerkiksi tässä opiskelijan on ohjelmoitava alkuarvojen laskenta sekä itse simulaation toteuttaminen, jolloin aika ei mene itse asetelman rakentamisen ohjelmointiin.

```
# Simuloi pallon vino heittoliike pisteestä A=(-50,0,0). Alkunopeus 25 m/s
# heitettyna 35 astetta vinosti ylos, ei ilmanvastusta from visual import *

# grafiikkaikkunan määritys (ei pakollinen)
scene.range=(60,60,12)
scene.width = 800
scene.height = 600

A=vector(-50,0,0)
# Tuotetaan ikkunaan pallo-objekti
ball=sphere(pos=A,radius=1,color=color.magenta)

# Lasketaan tarvittavat alkuarvot
g=vector(0,-9.8,0)
ang0=radians(35)
v0=25 #lahtonopeuden itseisarvo
vy0=v0*sin(ang0) #alkunopeuden y-komponentin suuruus
vx0=v0*cos(ang0) #alkunopeuden x-komponentin suuruus
# otetaan kayttoon pallon uusi vektoriattribuutti nopeus ja alustetaan
```

```
# se alkuarvoilla
ball.velocity=vector(vx0,vy0,0)

dt=0.01 # aika-askeleen määritys

while 1 :
    # vaadittava kontrollimuuttuja
    rate(100)   (hyva arvio 1/aika-askel)

    if ball.pos.y >=0:
        ball.velocity +=g*dt
        ball.pos+=ball.velocity*dt
```

Dynaaminen tyypitys tuo toisaalta ohjelmointiin sujuvuutta, kun muuttujien tyyppejä ei tarvitse esitellä saatikka pohtia. Tämä on kuitenkin myös suurin ajonaikaisen virheen syy, kun yritetään operoida vektoreilla jotain sellaista, missä vaadittaisiin skalaari tai päinvastoin.

Ohjelmointi ja kielen syntaksi ovat helppoja omaksua. Kielessä mm. ei ole lohkosulkeita, vaan syntaktinen lohko esitetään aina oikean suuruisella koodin sisennyksellä. Tästä johtuen kieli pakottaa ohjelmoijan tuottamaan hyvin luettavaa koodia. Tarvittavien ohjausrakenteiden if-lauseen ja while-lauseen rakenne on helppo omaksua, vaikka ohjelmointikokemusta ei olisikaan. Vertailuoperaattorit ja loogiset vertailulausekkeet muodostetaan hyvin samantapaisesti kuin perinteisissä proseduraalisissa kielissä. Rakenteisia tietotyyppejä (lista, monikko) ei tarvitse alkuvaiheessa käyttää.

Graafiset objektit luodaan ikkunaan suoraan kutsumalla koodissa valmiita konstruktoreita, joissa voi asettaa myös tarvittavat alkuarvot. Objekteille ohjelmoija voi asettaa uusia ominaisuuksia pelkästään kirjoittamalla muuttujaan pistenotaatiolla uuden attribuutin nimi ja asettamalla sille jokin arvo. Attribuutin tyyppi määräytyy dynaamisesti arvon perusteella. VPythonissa on mm. mahdollista asettaa pallolle pintamateriaaliksi maapallon kartta, mikä tuo näyttävyyttä avaruudellisiin animaatioihin.

Jokaisella objektilla on sen paikkaan koordinaatistossa liittyvä attribuutti `pos` ja objektin liikuttaminen tapahtuu yksinkertaisesti antamalla `pos` -attribuutille uusia arvoja. Tämä riittää jo siihen, että jokainen saa ensimmäisen harjoittelun alussa esineet liikkumaan näytöllään itse määrittelemillään käskyillä, mikä sinänsä on jo innostavaa. VPythonista löytyvät myös kaikki vektorilaskennassa tarvittavat funktiot sisäänrakennettuina. Matemaattiset funktiot saadaan käyttöön lataamalla **`math`**-kirjasto (`from math import *`). Omia funktioita on hyvä oppia muodostamaan. Niiden määrittely täytyy koodissa esiintyä ennen niiden kutsua. Perussyntaksi määrittelylle on esitetty alla

```
def funktionNimi(parametrienNimet):
    #runko
    return paluuarvo # mikäli funktio palauttaa arvon
```

Kun simulaation haluaa liittää tapahtuvaksi reaaliajassa, jonka kulun voi tuoda näkyviin grafiikka-ikkunaan, joutuu säätämään aika-askeleen ja `rate()`-funktion parametrin arvoja. Animaatioilla, joissa liike on ennalta määritetyn vektorifunktion määrittämä, joutuu lisäksi pohtimaan vektorifunktiossa esiintyvän aikaparametrin `t += dt` kertoimen määrittämistä, jotta esimerkiksi ympyräradan kiertoajaksi tulostuu juuri oikea arvo.

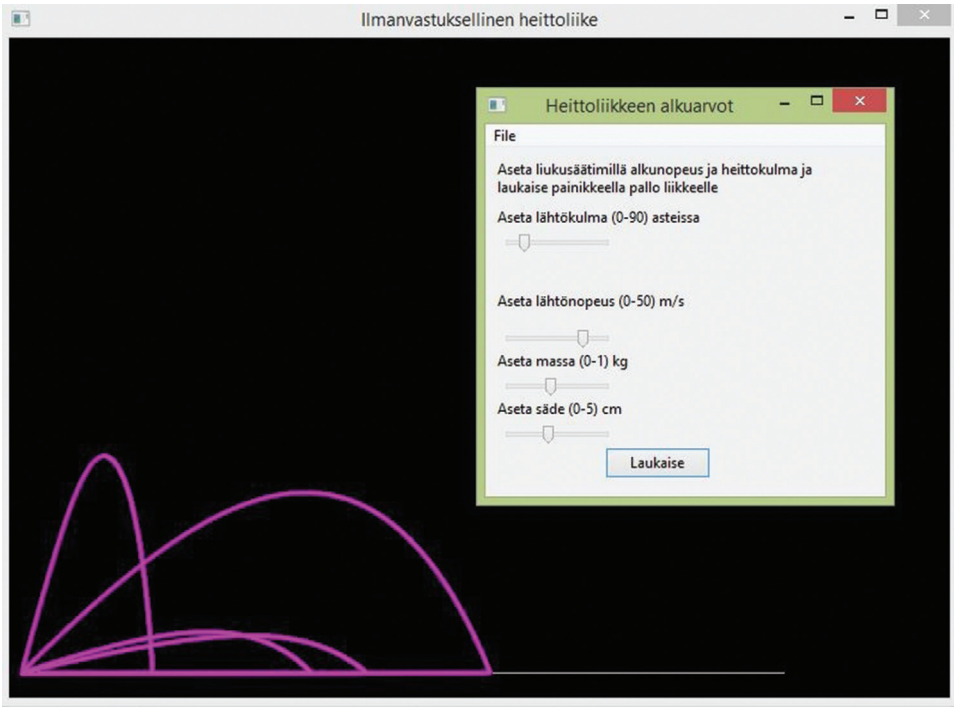
Edellä esitetty kuvailu ympäristöstä on riittävä sille, että pääsee toteuttamaan pelifysiikan animaatioita ja simulaatioita. Kun esityksiin halutaan vuorovaikutusta käyttäjän kanssa, joudutaan perehtymään erilaisiin **widget-käyttöliittymäelementteihin**, joiden käyttöönotto ilman syvällistä ohjelmoinnin osaamista onnistuu kyllä kopioimalla ja muuntamalla mukana tulevia ja netissä olevia esimerkkiohjelmia (Kuva 4.).

Negatiivisina piirteinä VPython ympäristöstä voidaan mainita mm:

- Tulkin ohjelmointivirheiden ilmoitukset ovat kömpelöitä eikä niitä aina edes generoida. Aika yleinen tilanne se, että ohjelma ns. jämähtää jumiin koodissa olevaan virheen takia, joka dynaamisesta tyypityksestä johtuen tulee esille vasta ajonaikana.
- Koska ympäristö ei ole kaupallinen tuote, voi löytyä yllättäviä epäyhteensopivuuksia käytetyn tietokoneympäristön kanssa, jolloin tukea täytyy yrittää etsiä pelkästään vertaiskäyttäjiltä netin kautta.
- Monimutkaisempien systeemien rakentaminen vaatii paljon aikaa ja vaivaa, koska kaikki tulee tehdä ”käsin” koodissa.
- Python sovelluksia ei saa ainakaan kovin helposti formaattiin, jossa niitä voisi ajaa esimerkiksi selainohjelmalla tai käyttöjärjestelmällä (exe).

Kuva 4. Widget –elementtejä simulaatiossa VPython ympäristössä

Omien kokemuksieni mukaan pelifysiikan toteutus Python-ympäristössä mahdollistaa edistyneimmille opiskelijoille etenemisen reaalimaailman ilmiöiden tutkimisessa



paljon pitemmälle kuin perinteisen fysiikan kurssilla tosin fysiikan teorian kattavuuden kärsiessä. VPython-ympäristössä voi simuloida näyttävästi ja hyvinkin tarkasti esimerkiksi maapallon ja kuun muodostaman systeemin liikkeitä yhteisen massakeskipisteen suhteen. Simuloimalla voi tutkia mm. niitä häiriöitä, joita mahdollisesti massiivinen taivaankappale aikaansaa taivaankappaleiden kiertoratoihin ohittaessaan niiden muodostaman systeemin riittävän läheltä. Edistyneimmät opiskelijat pystyvät luomaan koko aurinkokunnasta simulaatioita tai animaatioita, jossa kuut kiertävät planeettoja, jotka kiertävät aurinkoa. Kameran kytkeminen johonkin liikkuvaan planeettaan antaa mielenkiintoisia näkymiä muiden planeettojen ratakäyristä tähtitaivaalla – samoja, mitä entisaikojen tähtien tarkkailijat pystyivät havainnoimaan. Näytön zoomaus ja katselukulman säätö tapahtuvat grafiikkaikkunassa helposti hiiren avulla.

Pelifysiikassa simuloinnin kohteet voivat syntyä helposti jostakin ajankohtaisesta tapahtumasta. Baumgartnerin 14. lokakuuta 2012 noin 40 km korkeudesta stratosfääristä suorittama vapaapudotus oli esimerkki tästä. Simulaatiossa pyrittiin toteuttamaan hyppy fysiikan lakien mukaisesti siten, että ennätysnopeus ja kokonaiskesto olivat julkaistujen tietojen mukaiset. Kaikki hyppääjän kinematiikkaan ja dynamiikkaan liittyvät suureet kuten hypyn pituus, nopeus, vaikuttavat voimat ja niiden aiheuttama kiihtyvyys voidaan tulostaa reaaliajassa näyttöruutuun jopa grafiikkana. Hyppääjän nopeuden kehittyminen vapaan pudotuksen aikana on aika yllättävääkin ilman vastuksen vähitellen lisääntyessä taulukoitujen ilmantiheystietojen mukaisesti (kts. kansikuva).

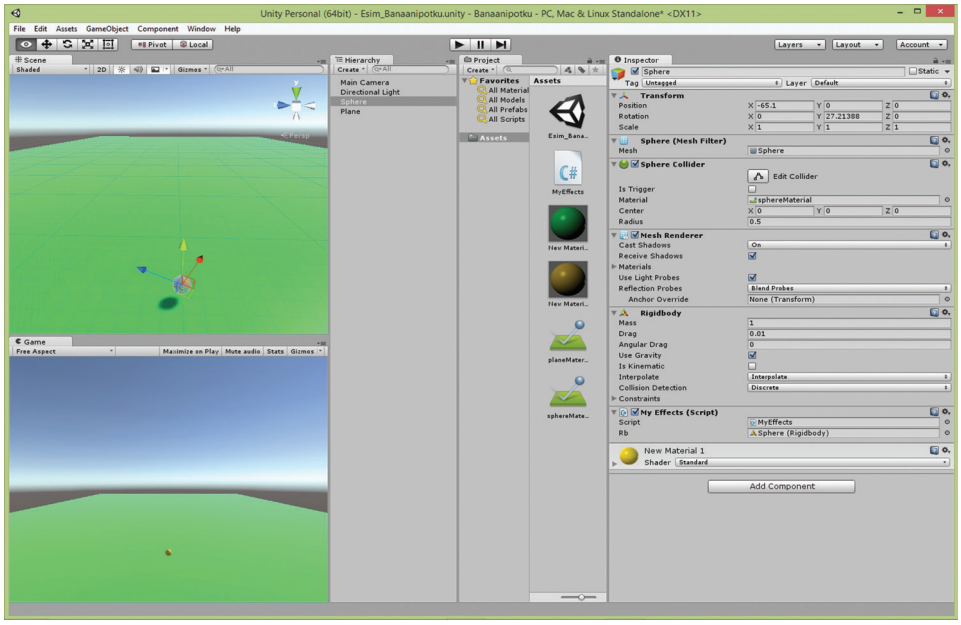
4.2 UNITY-YMPÄRISTÖN EDUT JA HAITAT

Kirjoittajan kokemukset Unityn käytöstä ovat suhteellisen rajoittuneet. Pitämilläni pelifysiikan kursseilla Unitya on käytetty lähinnä harjoitustöissä, joissa kaikissa on ollut valmis fysiikkamoottori käytössä. Ohjelmointityö on tällöin liittynyt

zmentoimiseen. Tässä esityksessä keskityn kuitenkin kuvailemaan Unityn soveltuvuutta pelifysiikan oppimisen ohjelmointiympäristöksi lähinnä tekemieni kokeilujen perustella.

Unityn pelinkehitinalusta koostuu yhdestä käyttöliittymäohjelmasta sekä erillisestä mukana tulevasta koodieditorista. Käyttöliittymässä on paljon toiminnallisuutta eri ikkunoissa ja sen oppiminen vie aikaa. Erona VPythonin ympäristöön nähden Unityssa on mm. kaksi grafiikkaikkunaa. Toista kutsutaan skeneksi (Scene) ja siinä voidaan luoda, asemoida ja muuttaa objektien sijaintia, asentoa ja kokoa sekä myös skenen näkymää ikkunan yläpuolella olevilla työkaluilla. Toisessa ikkunassa saadaan liikuteltavan kameran antama kuva pelin ajovaiheesta (Kuva 5.).

Kuva 5. Unityn käyttöliittymän osat: vasemmalla yllä skene ja alla pelinäkymä (kameran pelinäkymä), keskellä (Assets) projektiin liitetyt skriptit, materiaalit ym. sekä oikealla valittuna olevan objektin komponentti-ikkuna.



Unityssa on valmis fysiikkamoottori, jonka käyttöönotto tekee erilaisten simulaatioiden tekemisestä helppoa. Lisätään skeneen erilaisia kiinteitä rakennelmia. Sen jälkeen

lisätään objekteja, joihin lisätään **Rigidbody**-komponentti. Tässä komponentissa määritetään mm. objektin massa (suhteellinen arvo muihin objekteihin nähden), ilmanvastuksen vaikutusta etenevään liikkeeseen ja pyörimisliikkeeseen mallintavat arvot Drag (0 … 1) ja Angular Drag (0 … 1) sekä otetaan gravitaatiovoima käyttöön. Näiden toimintojen jälkeen simulaation voi jo käynnistää, koska fysiikkamoottori huolehtii jäykkien kappaleiden liikkumisesta ja törmäyksistä automaattisesti. Kaikki objektit, jotka ovat ilmassa, alkavat gravitaation voimasta pudota, kunnes kohtaavat jonkin kiinteän esteen (johon gravitaatio ei vaikuta) tai törmäävät muihin liikkuviin objekteihin. Törmäyksien kimmoisuutta ja kitkaominaisuuksia voidaan säätää lisäämällä kappaleisiin fysikaalinen materia (Physic Material). Samalla menetetään kuitenkin mahdollisuus puuttua valmiiksi sisään rakennettuihin fysiikkaratkaisuihin, joita ei saa muutoin esille kuin tutkimalla lähdekoodia. On selvää, että valmiin fysiikkamoottorin käyttö edellä kuvatulla tavalla ei ole kovin mielekäs lähestymistapa fysiikan opiskeluun.

Unityn ohjelmoiminen tapahtuu täysin erilaisen ohjelmointiparadigman puitteissa kuin edellä VPython ympäristössä. Unityssa käytetään **tapahtumapohjaista ohjelmointia**, missä ohjelmoija koodaa tapahtumien käsittelyjä. Tapahtumat voivat syntyä esimerkiksi törmäyksistä, joissa peliobjektit siirtyvät toistensa **Collider**-komponentin rajaamaan tilaan, tai esimerkiksi pelaajan näppäimistöjen painalluksista. Pelimoottori toimii tapahtumien kuuntelijana ja suorittaa niihin ohjelmoidun käsittelyn. Tapahtumien kuuntelu tapahtuu erityisten funktioiden välityksellä, joiden nimen alussa esiintyy prepositio On kuten esimerkiksi funktiossa `OnCollisionEnter()`.

Eulerin menetelmä on sisäänrakennettu ominaisuus eli Unity päivittää objektien paikat peräkkäisiin kuvaruutuihin automaattisesti. Kuvaruutujen välisen aika-askeleen suuruus määräytyy monista eri tekijöistä ja sen arvon sekunneissa saa luokkamuuttujalla `Time.deltaTime`.

Oma fysiikka ja pelaajan ohjaukset implementoidaan skriptien avulla. Kaikille objektiolioille voidaan liittää ohjelman koodi `Script` (C# tai Javascript, tässä esimerkit C#), jossa määritetään halutut objekteihin vaikuttavat efektit. Lisättäessä objektiin Script-komponentti Unity generoi oletusskriptin, joka ei tee vielä mitään. Siinä esiintyy kaksi funktiota, joista ensimmäinen `Start()` ajetaan yhden kerran, kun objekti luodaan peliin. Jälkimmäinen funktio `Update()`ajetaan aina peräkkäisten kuvaruutujen välissä, mikä vastaa Eulerin menetelmän toteutusta.

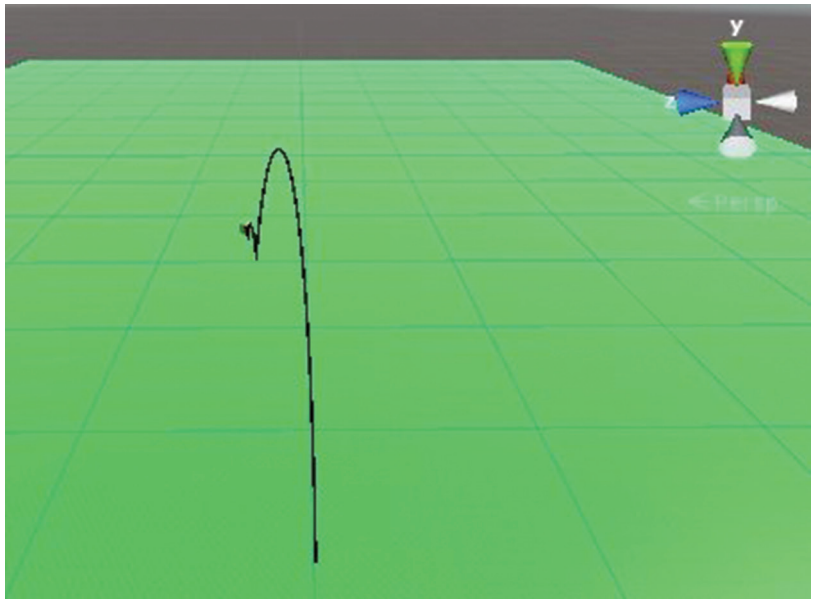
```
using UnityEngine;
using System.Collections;
public class myEfects : MonoBehaviour {
        // Use this for initialization
        void Start () {
        }

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

Fysiikkamoottorin (Rigidbody komponentti) ollessa käytössä on muistettava seuraavat perussäännöt:

- Jokaisella objektilla on **Transform**-komponentti, joka huolehtii objektin paikkaan, nopeuteen, asentoon (Eulerin kulmat) ja kokoon (scale) liittyvien asetusten tai ohjelmallisten muutosten vaikutuksien päivittämisestä. Transform-komponentin ikkuna näkyy aina ensimmäisenä käyttöliittymän oikeassa reunassa.
- Mikäli rigidbodyn fysiikkamoottori otetaan käyttöön objektille, ei skriptin Update lohkossa pidä päivittää mitään Transform-komponentin ominaisuutta. Lohkossa Start voidaan kuitenkin antaa alkupaikkaan ja nopeuteen liittyviä määrityksiä.
- Skriptin update-lohkossa ohjelmoidaan vain objektiin vaikuttavia voimia (mm. `AddForce()`) tai pyörimisnopeuteen vaikuttavia voimien momentteja (mm. `AddTorque()`), jotka vaikuttavat objektiin fysiikkamoottorin laskemina. Voimat voivat olla myös pelaajan näppäimistöltä antamia ohjauksia.
- Rigidbodyn komponentissa massa ja ilmanvastusarvot ovat vain suhteellisia. Esimerkiksi eri objektien massojen suhteen suositellaan olevan enimmillään 100.

Objektiolion Transform-komponenttia voi skripteissä käyttää suoraan jäsenmuuttujalla `transform`. Objektin muut komponentit täytyy ottaa käyttöön funktiolla `GetComponent<Component>()`. Esimerkiksi Rigidbody-komponentin kutsu on `GetComponent<Rigidbody>()`. Oheisessa esimerkkiskriptin start-lohkossa asetetaan objektille alkunopeus ja alkukulmanopeus vektorisuureina. Esimerkkikoodin update-lohkossa puolestaan implementoidaan omaa fysiikkaa lisäämällä objektille Magnus-efektiä, joka vaikuttaa pallon ilmanvastukselliseen liikeradan kiertymiseen. Magnus-efektin vaikutussuunta määräytyy kulmanopeus- ja nopeusvektoreiden ristitulolla. Fysiikkamoottori ei siis luo todellista ilmanvastusta Rigidbody-komponentin muuttujien Drag ja Angular Drag avulla, vaan ne itse asiassa säätävät vain nopeutta ja kulmanopeutta. Magnus-efektin suuruus ei riipu suoraviivaisesti vain nopeudesta

ja kulmanopeudesta kuten esimerkkikoodissa on ohjelmoitu. Erimuotoisille ja -materiaalisille kappaleille voidaan Magnus-efekti määritellä käytännössä vain kokeellisesti kuten myös ilmanvastuskertoimetkin (Kuva 6.).



Kuva 6. Magnus-efektin vaikutus pallon liikerataan. Lähtösuunta on viistosti ylöspäin tasoon merkityn viivan suunnassa. Kulmanopeusvektorin suunta on suoraan tasosta ylös eli pallo kiertyy ylhäältä katsottuna vastapäivään.

```
private Rigidbody rb; // yksityinen jäsenmuuttuja

void Start () {
                rb=GetComponent<Rigidbody>();
                rb.velocity=new Vector3(20f,16f, 0f);
                rb.angularVelocity = new Vector3(0,15f,0f);
        }

void Update () {
                Vector3 w = rb.angularVelocity;
                Vector3 v = rb.velocity;
                rb.AddForce (Vector3.Cross (w,v)/30f);
        }
```

Kuvaan merkitty liikeradan jälki saadaan näkymään skenessä lisäämällä update-lohkoon ao. funktiokutsu.

```
Debug.DrawRay(transform.position, -v.normalized*0.5f, Color.black,10f);,
```

Jäljen alkupisteen määrittävä 1. parametrin `transform` viittaa sen objektiin olioon (this), johon skripti on liitetty. Jäljen suunnan ja pituuden määrittävässä 2. parametrissa paikallinen muuttuja `v` viittaa objektin nopeuteen. Viimeinen parametri `10f` on aika sekunneissa, jonka aikaa jäljen pisteet ovat näkyvissä.

Edellä kuvattu esimerkki on tehty fysiikkamoottorin toimiessa taustalla. Tällöin ei mekaniikan perusasioille jää kyllä mitään ohjelmoitavaa eikä valmiin fysiikkamoottorin käyttö siten sovellu ensimmäisen fysiikan kurssin opiskeluympäristöksi. Mutta Unitylla on tietenkin mahdollista toimia niinkin, että fysiikkamoottori ja Rigidbody-komponentti on kokonaan pois käytöstä. Sitten voi myös yrittää löytää kompromissia, jossa fysiikka voidaan ohjelmoida itse, mutta hyödynnetään osittain Unityn tapahtumapohjaiseen ohjelmointiin perustuvaa törmäysten hallintaa. Tällainen kompromissi löytyi kirjoittajan yrityksissä seuraavasta yhdistelmästä (ei siis välttämättä paras yhdistelmä):

- Lisätään kappaleille Rigidbody-komponentti ja siinä asetetaan **IsKinematic** päälle. Tämä määre asettaa kyseisen objektin fysiikkamoottorin laskennan ulkopuolelle, jolloin kaikki objektin muutokset tulee ohjelmoida **Transform**-komponentin avulla.
- Valitaan funktion `Update()` tilalle `FixedUpdate()`. Tällöin ikkunaruutujen välinen aika on koko ajan vakio.
- Törmäysten tapahtumien käsittelyyn käytetään funktioita `OnTriggerEnter()`, `OnTriggerStay()` ja `OnTriggerExit()` tai sitten funktioita `OnCollisionEnter()`, `OnCollisionStay()` ja `OnCollisionExit()`. Edelliset välittävät ohjelmoijan käyttöön törmäävän colliderin `OnTriggerEnter(Collider someCollider)`, kun taas jälkimmäiset itse törmäysolion `OnCollisionEnter(Collision thisCollision)`, joka sisältää tietoa törmäävästä colliderista ja sen komponenteista sekä muutamasta törmäyksen hallintaan liittyvistä fysikaalisista parametreista.

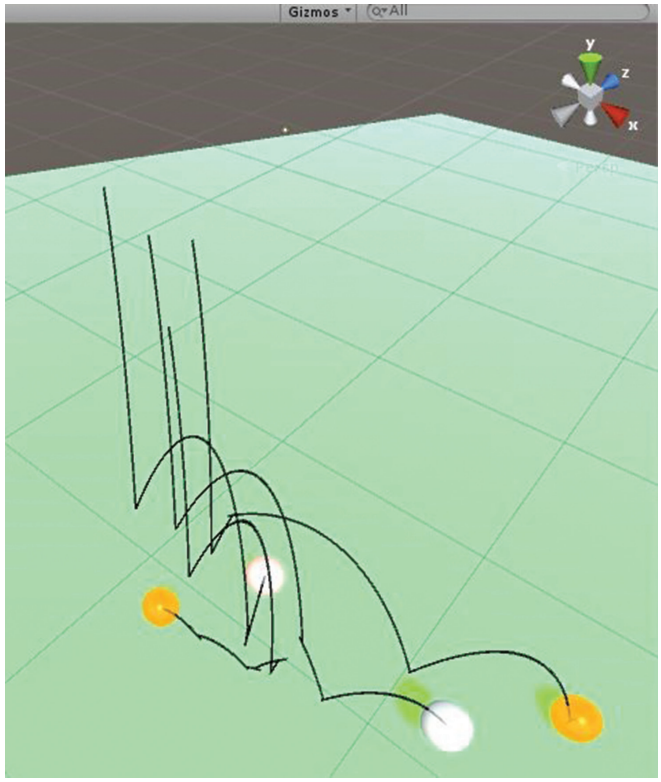
Tällöin fysiikan ohjelmoinnissa noudatetaan periaatetta, jossa kaikki vapaa liikkuminen (kappale ei ole kosketuksissa muihin objekteihin) ohjelmoidaan Eulerin menetelmällä funktiossa `FixedUpdate()` ja törmäyksien hallinta puolestaan tapahtuman käsittelijäfunktioissa. Törmäyksien aikana yleensä vapaan liikkumisen voimat kuten gravitaatio eliminoidaan pois esimerkiksi jollakin boolean-tagilla siltä varalta, että törmäyksen aikana ajetaan myös update-lohko.

Se mikä hyöty tapahtumapohjaisessa ohjelmointiympäristössä on VPythoniin nähden, ei liity fysiikan ohjelmoimiseen eikä itse törmäysten käsittelyynkään, vaan mahdollisuuteen hallita kokonaisuutta ilman tarkkaa eri tapahtumien loogisen järjestyksen analyysia. Pelissä voi olla useita samanlaisia peliobjekteja, joihin kaikkiin liitetään sama skripti. Tällöin jokaisella peliobjektilla on oma ilmentymä kyseisestä skriptistä ja pelimoottori huolehtii aina oikean skriptin ilmentymän ajosta. Voimme ohjelmoida peliobjektille törmäysten käsittelyn samalla skriptillä suhteessa kaikkiin muihin peliobjekteihin tutkimalla ensin törmäävän colliderin tyyppi ja ominaisuudet, jonka jälkeen ohjelmoidaan sitä vastaava käsittely. Tässä auttaa myös C#:n uudet kehittyneet tietorakenteet ja niihin ohjelmoidut algoritmit.

Esitän tässä viimeisen esimerkin koodin, siten, että kokonaisuuden hallinta ilmenee siitä. Koodin jälkeen käyn läpi oleelliset periaatteet.

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;//List collector

public class LinearImpulseScript : MonoBehaviour {

        private List<string> spheres = new List<string> { "Sphere (1)",
                                "Sphere (2)","Sphere (3)", "Sphere (4)"};
        private Rigidbody rb1, rb2;


void Start () {
                /* Alkuarvojen asetus */
        }

void OnTriggerEnter(Collider someCollider){
        . . .

        rb2=someCollider.attachedRigidbody;
        . . .
        if (somCollider.name == "Plane") {
                /* Tasoon törmäämisen käsittely */
        }

        else if (spheres.Exists(name=>name==someCollider.name)) {
                /* Toiseen palloon törmäämisen käsittely */
        }
```

```
void OnTriggerStay(Collider some){
                /* käsittely, kun törmääminen vaihtuu tasolla vierimiseksi */
}

void OnTriggerExit(Collider some){
                flying = true; //kun pallo irtautuu tasosta
        }


void FixedUpdate () {
        /* Vapaan liikkumisen käsittely Eulerin menetelmän periaatteella*/
        float dt= Time.deltaTime;

        if (flying == true) {
                . . .
        }
```



Kuva 7. Pallojen törmäyksien hallinta. Valkoiset pallot ovat 2.5 kertaa raskaampia, mikä näkyy etualalla olevien pallojen liikeratojen jäljissä.

Kolmannella koodirivillä liitetään nimiavaruuteen geneerinen kokoelma, jonka tietorakenne lista esitellään luokan yksityisenä (`private`) jäsenenä ja alustetaan kaikkien pallo-objektien nimillä (oliot erotteleva jäsenmuuttuja). Funktiossa `OnTriggerEnter()` käytetään geneerisen listan funktiota `Exist()`, joka käy läpi listan jäseniä ja palauttaa arvon tosi, mikäli argumenttina annettu *anonyymi (puhdas) funktio* eli *lambda-funktio* tulee todeksi jollakin listan jäsenellä. Tämä koodin osa on esimerkki funktionaalisten piirteiden lisääntymisestä C-kielen johdannaisissa. Puhdas funktio eli lambda-funktio on tuttu ainakin kaikille symbolisten matematiikkaohjelmien käyttäjille. Tällä runkorakenteella voidaan hallita kaikkien pallojen keskinäiset törmäykset sekä myös niiden törmäämiset alustaan (Kuva 7.).

Tottumattomalle Unityn käyttäjälle tuo alussa kovasti ongelmia epätietoisuus miten viitataan skripteissä objektin itsensä ja muiden peliobjektien eri komponenttien ominaisuuksiin. Objektin omaan Transform-komponenttiin viitataan olion (`this`) jäsenmuuttujalla `transform` ja Rigidbody-komponenttiin viitataan `GetComponent()`-funktiolla kuten jo aikaisemmin esitettiin. Sen sijaan esimerkin funktiossa `OnTriggerEnter(Collider someCollider)`törmäävän objektin Rigidbody-komponenttiin viitataan suoraan Collider-objektin jäsenmuuttujalla `attachedRigidbody`. Funktiolla `GetComponent()` viitataan yleisesti objektien komponentteihin mutta myös objektin skriptin julkisiin (`public`) jäsenmuuttujiin ja funktioihin kuten esimerkiksi seuraavassa, jossa paikalliselle muuttujalle `m2` asetetaan rigidbodyn skriptissä määritellyn julkisen jäsenmuuttujan arvo.

```
m2=rb2.GetComponent<LinearImpulseScript>().mass1
```

Tällä tavalla voidaan fysiikan ohjelmointi toteuttaa kokonaan skriptien jäsenmuuttujilla. Selkeä strategia, kun fysiikkamoottori ei ole käytössä, onkin se, että pidetään kaikki laskentaan tarvittavat fysikaaliset suureet skriptien jäsenmuuttujina ja päivitetään niiden arvot Transform-komponentin muuttujille toteutuvaksi pelinäkymässä. Fysiikka ohjelmoidaan skripteissä ja näkymä on visualisointia.

Skriptien julkiseksi määritellyt jäsenmuuttujat tulevat esille myös skriptin ikkunaan käyttöliittymässä, jolloin niiden muuttumista voi seurailla ajon eri vaiheessa pause-painikkeen avulla. Ikkunassa voi lisäksi antaa eri objektille erilaisia alkuarvoja. Tämä piirre osoittautui erinomaiseksi debuggaus-keinoksi virheitä jäljitettäessä.

Unityssa objektien luonnit, asemoinnit ja parametrien arvojen muutokset voi tehdä käyttöliittymässä, mikä helpottaa monimutkaisempienkin pelitilanteiden rakentelun. Unityn sovellus on lisäksi helposti muutettavissa formaattiin, jossa sitä voi ajaa eri käyttöjärjestelmissä.

Unityn ympäristö on kuitenkin aivan liian monipuolinen ja monimutkainen ensimmäisten fysiikan animaatioiden tekemiseen. Ohjelman itsenäinen käyttö vaatii suuren joukon eri parametrien ja komponenttien tunnistamista ja aika hyvää oliopohjaisen ohjelmointiparadigman hallintaa ainakin C#-skripteilla. Toinen kielivaihtoehto javascript on dynaaminen kieli ja lienee helpompi ottaa käyttöön. Fysiikkamoottorin ollessa käytössä tuntuu Unity-ympäristö aika sekavalta aloittelevalle fysiikan ohjelmoijalle, koska fysiikan suureet on hajautettu eri komponenteille. Kappaleen paikkaa päivitetään Transform-komponentissa, nopeus ja massa löytyvät puolestaan Rigidbody-komponentista. Lisäksi kappaleen koko skenellä määritellään erikoisesti kolmella Transform-komponentin koordinaattiakseleiden suuntaisella skaalaustekijällä.

Kun sisään rakennetun fysiikkamoottorin toimintaperiaate ei ole esillä, jää toiminnan oikeellisuus pelkästään visuaalisen näkemyksen varaan. Joissakin yksinkertaisissa tilanteissa fysiikkamoottori toimi jopa näkyvästi väärin. Tällainen käyttäytyminen osoittautui esimerkiksi materiaalin kimmoisuuden osalta. Täysi kimmoisuus (1) pomppivan pallon ja tason materiaalien ominaisuutena aiheuttaa hiljalleen pallon mekaanisen energian kasvun, mikä ilmeni nousukorkeuden jatkuvana kasvuna.

5. Johtopäätökset

Pelifysiikka on peliteollisuudessa käytettävien pelinkehitysalustojen pelimoottoreiden perusta. Kaikki ratkaisut, joilla simuloidaan tarkasti ja oikein reaalimaailman ilmiöitä, perustuvat fysiikan mallien matemaattisten ratkaisujen ohjelmoimiseen.

Pelifysiikan opiskelusta voin todeta seuraavaa:

- Pelifysiikan opiskelussa toteutuu ihanteellisella tavalla fysiikan, matematiikan ja ohjelmoinnin CDIO-pohjainen integroitu oppimisnäkökulma.
- Pelifysiikassa jokainen voi opiskella omalla tasollaan ja tavallaan paremmin kuin perinteisessä ”paperifysiikassa”.
- Pelifysiikka soveltuu erinomaisesti luonnon ilmiöiden tutkimiseen. Tämä edellyttää kuitenkin vahvoja niin fysiikan ja matematiikan kuin myös ohjelmoinnin taitoja. Luonnonilmiöiden mallintaminen erilaisilla vuorovaikutteisilla ohjelmilla on eräs peliteknologian soveltamisalue.

Pelifysiikan ohjelmointiympäristöistä voin todeta seuraavaa:

- Ilman fysiikkamoottoria olevat kevyet vaihtoehdot kuten VPython-ympäristö soveltuvat paremmin animaatioiden ja simulaatioiden perusteiden oppimiseen erityisesti vähäisellä ohjelmointikokemuksella
- Tapahtumapohjaiseen ohjelmointiin perustuvat kaupalliset pelinkehitysalustat kuten Unity ovat parempia vaihtoehtoja jo edistyneemmille pelifysiikan osaajille sekä erityisesti valmiin kaupallisen pelituotteen tekemiseen. Valmis fysiikkamoottori mahdollistaa pelituotteiden valmistamisen myös ilman fysiikan vaativampien menetelmien hallintaa.
- Pelifysiikan oppimisympäristöjen tehokas käyttö edellyttää ehdottomasti yhteistyötä matematiikan ja ohjelmoinnin opettajien kanssa. Sekä matematiikan vektorilaskennan että olio-ohjelmoinnin taitoja on harjoiteltava ennen kuin mitään fysiikan simulointiin perustuvaa tuotetta ryhdytään valmistamaan.

6. Lähdeluettelo

Bourg D.M. & Bywalec B., Physics for Game Developers, 2013, ISBN: 978-1-449-39251-2

Hecker Chris, Physics, Part 3: Collision Response, 1997 sekä Physics, Part 4: The Third Dimension, 1997 http://chrishecker.com/Rigid_Body_Dynamics, viitattu 26.10.2015

Lapin ammattikorkeakoulu, CDIO. http://www.lapinamk.fi/fi/Opiskelijalle/Lomakkeet-ja-ohjeet/Alakohtaiset-lomakkeet-ja-ohjeet/Tekniikka/CDIO, viitattu 26.10.2015

Lengyel E., Mathematics for 3D Game Programming and Computer Graphics, 3. Ed., 2012, ISBN-13:978-1-4354-5886-4

Mielikäinen M. & Tepsa T., Katsaus tieto- ja viestintätekniikan koulutuksen CDIO-projekteihin keväällä 2015, 2015, ISBN: 978-952-316-093-4

Taipale-Lehto U., Vepsäläinen J, Peliteollisuuden osaamisraportti Opetushallitus: Raportit ja selvitykset 2015:6, ISBN:978-952-13-6192-0

Tässä selvityksessä käsitellään pelifysiikkaan siirtymistä ammattikorkeakoulujen tietotekniikan fysiikan koulutuksessa. Pelifysiikassa toteutuu mahdollisuus integroida matematiikan ja fysiikan opintoja ammattiaineiden uusiin projektiluonteisiin oppimisympäristöihin. Selvityksessä esitellään pelisimulaatioiden vaatimaa teoreettista pohjaa niin matematiikasta, fysiikasta kuin myös ohjelmoinnista. Pelifysiikan opiskeluun soveltuvia sovellusesimerkkejä esitellään niin aloittaville kuin myös jo edistyneimmille opiskelijoille. Lisäksi perehdytään kehittyneempien pelinkehitysalustojen pelimoottoreiden käyttämään törmäyksien hallintaan ja sen edellyttämiin fysiikan malleihin. Pelifysiikan ohjelmointiympäristöjen käyttöön ottoon liittyviä vaatimuksia ja mahdollisuuksia tarkastellaan kahden periaatteeltaan erilaisen ohjelmointiympäristön VPython ja Unity esittelyllä.

Julkaisu on tehty Lapin ammattikorkeakoulun ohjelmistotekniikan laboratorion (pLab) vetämässä hankkeessa nimeltä *Peke*, jonka yhtenä tavoitteena on *peliteknologia osaamisen kasvattaminen Lapin AMK:n tieto- ja viestintätekniikan opettajien työkalupakissa*. Hanke on rahoitettu ELY:n (Elinkeino-, liikenne- ja ympäristö keskus) ESR (Euroopan unioni, Euroopan Aluekehitysrahasto, Euroopan Sosiaalirahasto) -kanavan kautta vuosien 2013 - 2015 välisenä aikana.

LAPIN AMK
Lapland University of Applied Sciences

www.lapinamk.fi

ISBN 978-952-316-108-5