Juuso Tenhunen

Kelluvuuden mallintaminen videopeleissä

Tietotekniikan pro gradu -tutkielma

31. lokakuuta 2019

Jyväskylän yliopisto

Tietotekniikan laitos

Tekijä: Juuso Tenhunen

Yhteystiedot: juuso.j.o.tenhunen@student.jyu.fi

Ohjaajat: Sanna Mönkölä, Paavo Nieminen ja Tuomo Rossi

Työn nimi: Kelluvuuden mallintaminen videopeleissä

Title in English: Simulation of Buoyancy in Video Games

Työ: Pro gradu -tutkielma

Suuntautumisvaihtoehto: Pelit ja pelillisyys

Sivumäärä: 67+4

Tiivistelmä: Nesteiden fysiikan ja kelluvuuden mallintaminen on laskennallisesti hyvin vaativaa. Kelluvuutta sovellettaessa videopelien ympäristöön vaaditaan simulaatiolta realistisuuden lisäksi reaaliaikaista suorituskykyä. Tämä tutkielma käy vaihe vaiheelta läpi erilaisten kelluvuuden simuloinnin toteutuksia Unity-pelimoottorille. Aluksi toteutetaan mallinnus, joka mukailee realistista kellumista käyttämättä kuitenkaan oikeita fysikaalisia laskutoimituksia hyödykseen. Tämän jälkeen toteutetaan oikeiden fysikaalisten kaavojen mukainen kelluminen ja lopuksi vertaillaan näiden toteutusten suorituskykyä ja realistisuutta, sekä esitetään mahdollisia käyttökohteita.

Avainsanat: Kelluvuus, Simulointi, Videopelit, Unity3D

Abstract: Fluid dynamics and buoyancy simulation are difficult to calculate accurately. When applying buoyancy in video games, the simulation needs to look realistic as well as to be calculated in real-time. This thesis goes through different buoyancy simulations step-by-step in the Unity game engine. First, a simulation that imitates realistic buoyancy without actually using real life physical calculations is presented. Then, a simulation with real physical formulas is implemented and finally a comparison of the performance and lifelikeness with possible impelementation areas is given.

Keywords: Buoyancy, Simulation, Video Games, Unity3D

Termiluettelo

FPS	Frames-per-second, eli kuvataajuus. Kertoo kuinka monta kuvaa sovellus piirtää näytölle sekunnissa.
Skripti	C# -luokka, jolla ohjelmoidaan Unity3D-pelimoottorin peliobjekteja. Nämä ovat yksittäisiä, kokonaisia tiedostoja esimerkiksi ‘WaveController.cs’
Funktio	Ohjelmoitu aliohjelma skriptien sisällä, esimerkiksi ‘DisplayMesh(Mesh m)’

Kuviot

Kuvio 1. Varjostimen asetukset ... 4
Kuvio 2. Kuvakaappaus aallokosta ... 6
Kuvio 3. 2-ulotteinen kuvakulma hitaassa sini-aallossa ... 10
Kuvio 4. Nopeampi sini-aalto, kappale jää selkeästi pinnan alle ... 10
Kuvio 5. (a) Vektorien ristitulo. (b) Vektorien ristitulo sijoitettuna kuution keskelle ... 11
Kuvio 6. Kappale kääntyy allokon mukana ... 13
Kuvio 7. 3-ulotteinen kuvakulma ... 20
Kuvio 8. Tiheämmällä kolmioverkolla varustettu kappale ja sen vedenalaiset osat ... 21
Kuvio 9. Tavat, jolla kolmiota käsitellään, riippuen mitkä osat siitä ovat veden alla. Kuvan pohjana käytetty Kerner (2016) mallia ... 21
Kuvio 10. Kolmio, jonka kaksi kulmapistettä ovat veden alla ja kolmion pilkkomiseen tarvittavat arvot ja pisteet. Kuvan pohjana käytetty Kerner (2016) mallia ... 22
Kuvio 11. Kolmio, jonka yksi kulmapiste on veden alla ja kolmion pilkkomiseen tarvittavat arvot ja pisteet. Kuvan pohjana käytetty Kerner (2016) mallia ... 24
Kuvio 12. Kaksiulotteinen näkymä aallon liikkeestä kuution läpi ... 26
Kuvio 13. Kolmiulotteinen näkymä ... 26
Kuvio 14. Kolmioiden ja aallokon koon kontrasti tuottaa epätarkkuutta ... 27
Kuvio 15. (a) Kappale pysyy levossa. (b) Kappale uppoaa. ... 30
Kuvio 16. (a) Kappale pysyy levossa. (b) Kappaleeseen kohdistuu vääntö ... 30
Kuvio 17. (a) Kappale on levossa. (b) Kappaleeseen kohdistuu vääntö. (c) Kappale on levossa ... 31
Kuvio 18. Kappale kelluu, mutta pyörii holtittomasti ... 33
Kuvio 19. Vedenalainen ja vedenpäällinen osa visualisoituna ... 34
Kuvio 20. Levymäiset litteät kappaleet pudotetaan veteen ... 45
Kuvio 21. Levymäiset litteät kappaleet kohoavat vedessä ... 46
Kuvio 22. Kontrollitilanne ... 52
Kuvio 23. Yksinkertainen mallinnus useammilla kappaleilla ... 53
Kuvio 24. 10000 kuutiota pelikentällä ... 53
Kuvio 25. 10000 kuutiota pelikentällä ilman Collider-komponenttia ... 54
Kuvio 26. Painenosteen ja keskipistenosteen suorituskyvyn vertailu ... 55
Kuvio 27. Prosessorin laskentanopeus a) ilman vastuksia b) vastuksilla ... 55
Kuvio 28. Käännetty peli: 10000 laatikkoa ja 1024 laatikkoa/jänistä ... 56
Kuvio 29. Stanfod Bunny-mallit erilaisten vastussimulointien vaikutuksessa ... 57
Kuvio 30. Käännetty peli: 100 laatikkoa raskailla ja helpoilla vastuksilla, sekä 4 jänistä funktion A.5 vastuksilla ... 58
Kuvio 31. Realistiset voimat kallistavat kappaletta sen mukaan mihin osaan voimat vaikuttavat ... 58
Kuvio 32. Yksinkertainen ’Drag’ vastus hidastaa kappaletta kokonaisuutena ... 59

Sisältö

1 JOHDANTO 1

2 VEDENPINNAN AALTOILUN SIMULOINTI 3
2.1 Vedenpinnan aaltoilun simulointi 3
2.2 Suorituskyvyn mittaus 5

3 SIMULOINTI ILMAN FYSIIKKAMOOTTORIA 8
3.1 Kappale vedenpinnalle 8
3.2 Kappaleen suunta ja kaltevuus aaltoon nähden 10

4 VEDENALAISEN KAPPALEEN KOLMIOVERKON MUODOSTAMINEN 14
4.1 Kappaleen vedenalaisen osuuden tunnistaminen 14
4.2 Kaksi kulmapistettä veden alla 22
4.3 Yksi kulmapiste veden alla 24

5 FYSIIKAN MALLIEN MUKAINEN SIMULOINTI 28
5.1 Kelluvuuden teoria 28
5.2 Massakeskipiste ja nostevoiman keskipiste 29
5.3 Nostevoiman lisääminen kappaleelle 31
5.4 Liikettä vastustavien voimien lisääminen 33
5.5 Nosteen laskeminen tilavuuden keskipisteestä 46

6 SIMULAATIOTULOKSET 52
6.1 Johtopäätökset 55

7 YHTEENVETO 60

LÄHTEET 61

LIITTEET 63
A Apuluokat ja lisäfunktiot 63

1 Johdanto

Videopelit pyrkivät usein simuloimaan maailmaa mahdollisimman realistisesti. Kuitenkin vesi jätetään useasti toisarvoiseen osaan ja usein pelit tyytyvät jättämään veden vain koristeeksi, jonka kanssa ei voi olla vuorovaikutuksessa, tai mallinnukseen käytetään hyvin yksinkertaisia kelluntamekaniikkoja. Kenties yksi isoimmista syistä tähän on laskentateho, sillä täysin realistisen kellunnan simulointi reaaliajassa olisi laitteistolle hyvin haastavaa. Myös approksimointi kappaleen käytöksestä vedessä on hankalaa, sillä kappaleet käyttäytyvät vedessä hyvin eri tavalla kuin ilmassa ja oikean näköisten tulosten saaminen voi olla hankalaa.

Vaikka planeetta jolla elämme koostuu noin 71% vedestä (Arheimer 2016), veden osuus videopeleissä on yleensä hyvin pieni ja toissijainen. Varsinkin vanhemmissa peleissä vesi toimi esteinä, joiden yli pelaajien tuli hypätä (esimerkiksi *Teenage Mutant Ninja Turtles* (Konami, 1989)). Joissain peleissä vesiesteiden lisäksi oli yksi tai kaksi erillistä vesikenttää, jotka eivät muuttaneet tavallisia pelimekaniikkoja kovinkaan paljoa, vaan enintään tekivät hypyistä korkeampia kuten Mega Man 2-pelissä *Mega Man 2* (Capcom, 1988) tai sallimalla pelaajan 'hypätä' myös kesken vajoamisen, jolloin vedessä pystyi uimaan myö ylöspäin (esimerkiksi *Donkey Kong Country* (Rare, 1994)). Nykyään pelit käyttävät paremmin vettä hyödykseen, mutta useasti pienissä rooleissa ja edelleen tiettyihin kenttiin sidottuina. Esimerkiksi *Ship Graveyard* pelissä *Uncharted 3: Drake's Deception* (Naughty Dog, 2011), jossa vesi ja aallot liikkuvat esteettä kevyesti kelluvien laivojen läpi.

Osasyy veden ja kellunnan jättämiseen huomiotta peleissä on niiden vaatima suuri laskentateho (Rath 2014). Realistinen nestedynamiikka on hyvin raskasta simuloida veden fysikaalisten ominaisuuksien takia. Ilma on suhteessa veteen paljon helpompi käsitellä, sillä yleensä voidaan olettaa että kaikki kappaleet putoavat samalla tavalla alaspäin, jolloin pelin fysiikkamoottorin on yksinkertaista laskea nämä voimat. Vesi puolestaan käyttäytyy hyvin eri tavalla verrattuna ilmaan. Vaikka molemmat ovat fluideja, yleensä voidaan pelimaailmassa olettaa ilman pysyvän paikallaan, kun taas painovoima vaikuttaa veteen huomattavasti vahvemmin, joten veden virtaus ja liike ovat paljon selkeämmin havaittavissa, jolloin epärealistinen simulaatio erottuu helposti. Veden oma tiheys myös vaikuttaa muihin kappaleisiin huomattavasti vahvemmin kuin ilmassa, jolloin eri tiheyksiset kappaleet joko uppoavat tai kelluvat. Tästä

syystä pelit ottavat monesti oikoteitä ja simuloivat kelluvat esineet vain näyttämään käyttäytyvän oikeiden kelluntafysiikoiden mukaan. Algoritmeja tälläisen 'fysiikattoman' kellunnan aikaansaamiseksi käsitellään luvussa 3.

Peleissä, joissa vesi on kuitenkin iso osa pelimaailmaa ja pelimekaniikkoja, täytyy simuloida vettä ja kelluvuutta hieman tarkemmin, jotta pelikokemus olisi aito ja tyydyttävä. Pelissä *Assassin's Creed III* (Ubisoft, 2012) isot purjelaivat ovat suuressa roolissa pelissä, joten realistinen kelluminen on tärkeää pelikokemuksen kannalta. Pelinkehittäjät hyväksyivät, että täysin yksityiskohtainen nestedynamiikka ja jäykkien kappaleiden törmäyssimulaatiot nesteeseen olisivat mahdottomia reaaliajassa. Tästä syystä he yksinkertaistivat mallejaan hiukan, mutta jättivät taustalle oikeat fysikaaliset ilmiöt. Esimerkiksi laivoihin kiinnitettiin näkymättömiä 'kelluntapalloja' joita oli eri määrä ja eri paikoissa riippuen laivasta ja kellutettavasta kappaleesta. Nämä pallot välittivät realistiset voimat itse laivaan aallokon ja veden mukaan, jolloin kappale saatiin käyttäytymään vedessä tarpeeksi realistisesti (Seymour 2012). Luvussa 5 keskitytään tämän tyylisiin oikeilla fysikaalisilla voimilla toimiviin algoritmeihin.

Tässä tutkielmassa käydään läpi muutamia tapoja simuloida kappaleiden kelluntaa ja vertaillaan näiden tapojen vaikutusta suorituskykyyn ja niiden visuaalisia eroja. Luvussa 2 käydään lyhyesti läpi käytettävät ohjelmistot ja laitteet sekä suorituskyvyn mittausta ja vedenpinnan aaltoilun simulointiin luodut skriptit. Luku 3 aloittaa soveltavamman asian käsittelyn ja keskittyy kellunnan simuloimiseen pelimaailmassa. Tässä luvussa keskitytään sellaisiin approksimoituihin malleihin, joiden avulla kelluminen saadaan näyttämään realistiselta ilman realistisia fysiikan simulointeja. Luku 4 pohjustaa lukua 5 esittelemällä siihen vaaditun tavan pilkkoa kolmioverkkoa, jotta vedenalainen osio kappaleesta saadaan selville. Luvussa 5.1 käydään läpi kelluvuuden teoriaa. Luvussa esitetään kelluvuuteen liittyvät fysikaaliset kaavat ja periaatteet, joista kelluntа johtuu ja millaiset voimat kappaleeseen vaikuttavat sen kelluessa. Luku 5 siirtyy käyttämään luvussa 5.1 esiteltyjä fysiikan kaavoja ja pyrkii simuloimaan reaalimaailman kellumista mahdollisimman tarkasti, ottaen huomioon kellumiseen vaikuttavat voimat oikein. Luku 6 kertaa käsitellyt simulaatiot ja vertailee niiden tuottamia tuloksia, sekä esittää ehdotuksia eri käyttökohteille, mihin mitäkin simulaatiomallia voisi käyttää.

2 Vedenpinnan aaltoilun simulointi

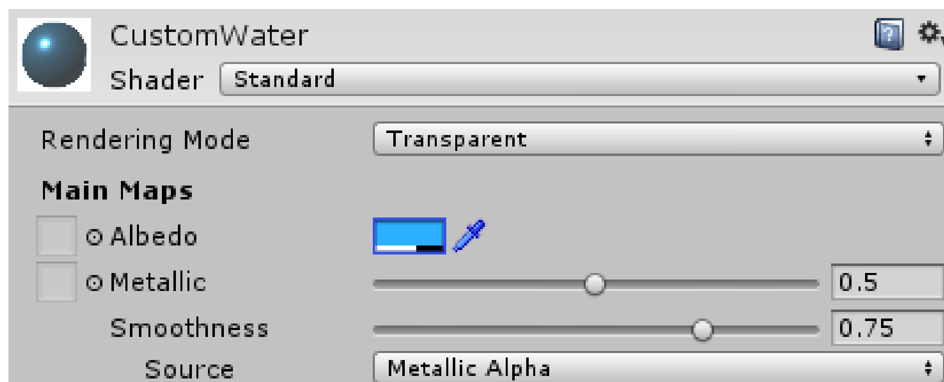
Eri kelluvuusalgoritmien kehittämiseen ja testaamiseen käytetään Unity3D-pelimoottoria ja Visual Studio 2017 -editoria. Ohjelmointikielenä toimii C#. Pelimoottorin sisäiset muuttujat yhtenäistetään metreiksi ja kilogrammoiksi, eli koordinaatiston arvot vastaavat suoraan metrejä ja kappaleisiin sijoitettava massa on kilogrammoina. Fyysisenä laitteistona käytetään pöytätietokonetta, jonka prosessori on Intel i5-4670K ja näytönohjaimena on Nvidian GTX 970 kortti.

2.1 Vedenpinnan aaltoilun simulointi

Vedenpinnan aaltoilun simulointi aloitetaan luomalla vedenpinnan materiaali Unityn oletusvarjostimella (Standard Shader [1]), joka on vakiovaihtoehtona kaikissa uusina luoduissa materiaaleissa. Kuvion 1 mukaisesti varjostimen valintaikkunassa väriksi asetetaan hiukan läpinäkyvä sinivihreä sävy, Albedo-rivin värivalinnasta aukeavaan ikkunaan syöttämällä RGBA (punainen, vihreä, sininen, läpikuultavuus) arvot [43,176,255,154]. Materiaalin Metallic-arvo säätää kuinka paljon valoa kappale heijastaa itsestään. Arvo 0 on täysin matta ja 1 on täydellinen valon heijastus. Smoothess-arvo määrää pinnan karkeuden, eli kuinka pinta sirottaa siitä kimpoavan valon. Arvo 0 sirottaa kimpoavan valon hyvin moneen eri suuntaan ja arvo 1 ei sirota valoa ollenkaan vaan valonsäde kimpoaa peilin tavoin yhteen suuntaan. Metallic-arvoksi valitaan 0.5 ja Smoothness-arvoksi 0.75, jonka seuraksena materiaalista saadaan keskiverto heijastus aikaiseksi. Tämä materiaali on tarpeeksi yksinkertainen, jotta se ei häiritsevästi taita eikä heijasta valoa oikean veden tavoin, mutta kuitenkin päästää hiukan valoa pinnan läpi, jotta vedenalaiset osat näkyvät selkeästi.

Veden pinnan aaltoiluun kirjoitetaan oma algoritmi, jotta aaltojen nopeuteen, korkeuteen ja tiheyteen voidaan helposti vaikuttaa. Tähän käytetään kahta eri skriptiä, jotka ovat GenerateWave.cs ja WaveController.cs. GenerateWave.cs käy läpi vedenpinnan monikulmioverkon kulmapisteet ja asettaa niille sopivat korkeusarvot käyttäen apuna WaveController.cs-skriptin funktiota 2.2.

1. https://docs.unity3d.com/Manual/shader-StandardShader.html



Kuvio 1: Varjostimen asetukset

GenerateWave.cs:n globaaleiksi muuttujiksi tarvitaan veden monikulmioverkko, Vector3-taulukko uusille ja alkuperäisille kulmapisteille ja muuttuja viittaus WaveController-skriptiin. Näitä muuttujia hyödyntäen funktio 2.1 'MoveSea()' saa oikean korkeuden kolmioverkkonsa kulmapisteille luoden aallokkomaisen pinnan.

Funktio 2.1: void MoveSea()

```
1 newVertices = new Vector3[originalVertices.Length];
2 for (int i = 0; i < originalVertices.Length; i++)
3 {
4     Vector3 vertice = originalVertices[i];
5     vertice = transform.TransformPoint(vertice);
6     vertice.y = waveScript.GetWaveYPos(vertice.x, vertice.z);
7     newVertices[i] = transform.InverseTransformPoint(vertice);
8 }
9 waterMesh.vertices = newVertices;
10 waterMesh.RecalculateNormals();
```

Funktio 2.2 'GetWaveYPos' laskee y-arvon annetuilla x ja y koordinaateilla käyttäen prosuduraalista Perlin kohinaa (Perlin 1985), sekä sen hetkistä kulunutta aikaa siirtäen kohinaa tasaisesti ajan kanssa luoden tasaisen aallon. Funktio käyttää muuttujia perlinScale, wavespeed ja waveHeight laskeakseen aallokon leveyden, nopeuden ja korkeuden.

Perlin kohinan yksi etu verrattuna muihin sattumanvaraisiin kohinoihin on laskettujen arvojen jatkuvuus: funktio palauttaa kahdelle vierekkäiselle pisteelle tasaisesti muuttuvat arvot,

jolloin tuloksena tulee tasaisesti laskeva ja nouseva käyrä. Toinen etu on toistettavuus, sillä funktio palauttaa aina saman arvon, jos sen lähtöarvot ovat samat. Tämä tarkoittaa että kellutettavat kappaleet voivat pyytää funktiolta 2.2 vedenpinnan korkeutta suoraan omille koordinaateilleen, jolloin se saa vastauksena tarkan ja ajankohtaisen arvon vedenpinnan korkeudeksi.

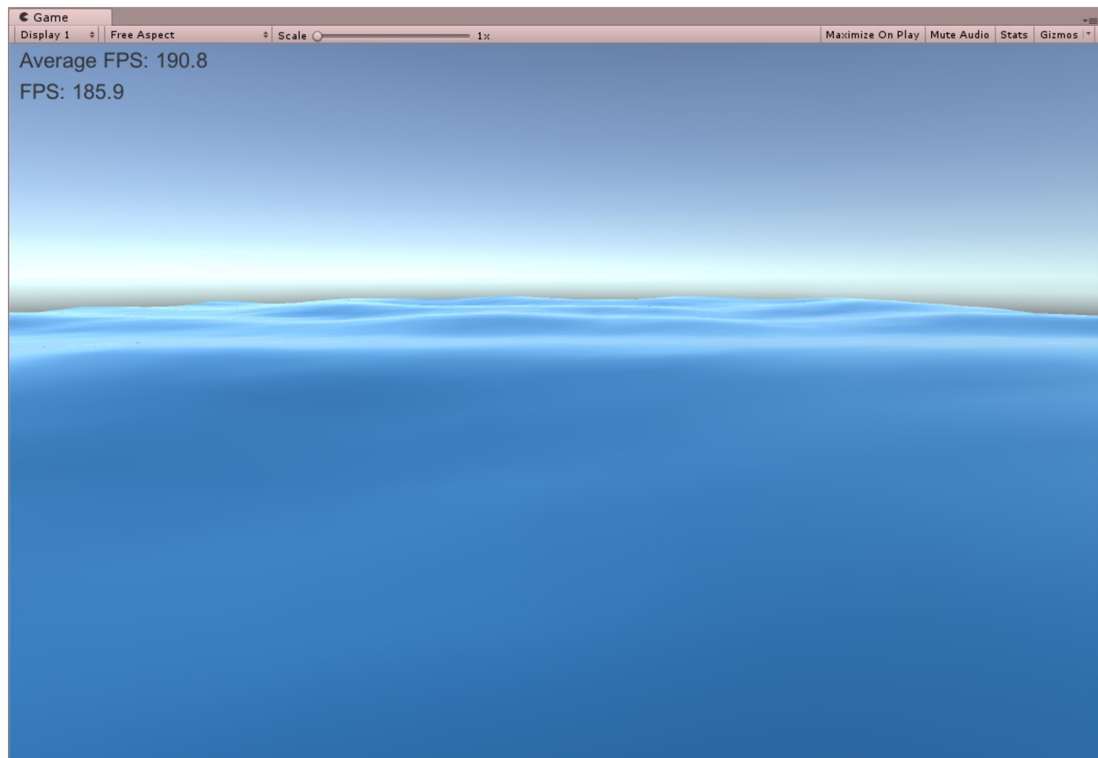
Funktio 2.2: float GetWaveYPos(float x_coord, float z_coord)

```
1 float y_coord = 0f;
2 float pX = (x_coord * perlinScale) + (timeSinceLevelLoad * wavespeed);
3 float pZ = (z_coord * perlinScale) + (timeSinceLevelLoad * wavespeed);
4 if (onlyX)
5 {
6    y_coord = (Mathf.PerlinNoise(pZ, pZ) - 0.5f) * waveHeight;
7    y_coord += (Mathf.PerlinNoise(pX, pZ) - 0.5f) * waveHeight / 2;
8 }
9 else
10 {
11     y_coord = (Mathf.PerlinNoise(pX, pZ) - 0.5f) * waveHeight;
12     pX = (x_coord * perlinScale * 6) + (timeSinceLevelLoad * wavespeed
          * 2);
13     pZ = (z_coord * perlinScale * 6) + (timeSinceLevelLoad * wavespeed
          * 2);
14     y_coord += (Mathf.PerlinNoise(pX, pZ) - 0.5f) * waveHeight/12;
15 }
16 return y_coord;
```

Tuloksena saadaan kuvassa 2 nähtävä merenpinnan aaltoilua simuloiva malli, jonka arvoja on helppo ja nopea säätää tarpeen vaatiessa.

2.2 Suorituskyvyn mittaus

Suorituskyvyn mittaamiseen käytetään lyhyttä skriptiä, joka laskee pelin hetkellisen kuvataajuuden (FPS, engl. *frames per second*) käyttämällä UnityEnginen Time.unscaledDeltaTime-



Kuvio 2: Kuvakaappaus aallokosta

muuttujaa [2], joka kertoo ajan sekunteina kahden peräkkäisen kuvapiiron välillä. Tämän luvun käänteisluvulla saadaan tarkka FPS-lukema. Esimerkiksi jos aika edelliseen näytölle piirrettyyn kuvaan on 0.01s, kertoo lasku

$$1.0/0.01 = 100, \tag{2.1}$$

että tällä nopeudella saadaan piirrettyä 100 kuvaa sekunnissa = 100 FPS. Tämän lisäksi kerätään 100 viimeisintä FPS arvoa taulukkoon, joista lasken näiden keskiarvon. Tällöin saadaan nopeasti vaihtuvan luvun sijaan luettava ja pyöristetty luku, josta saa paremman kokonaiskäsityksen suorituskyvystä.

Jotta varmistetaan, että mikään kellutettava kappale tai muu muuttuja itsessään ei vaikuta suorituskykyyn, mitataan ennen jokaista testiä kontrolliarvo ilman kelluntafunktiota, jotta saadaan verrattava FPS arvo siihen, kun oikea kelluntafunktio on toiminnassa.

Kiinostavana asiana on myös menetelmien skaalautuvuus, jolla tarkoitetaan sitä, kuinka

2. `https://docs.unity3d.com/ScriptReference/Time-unscaledDeltaTime.html`

monta yhtä aikaa kelluvaa kappaletta funktio tukee. Sillä esimerkiksi jos jokin funktio puolittaa sen hetkisen FPS:n, yksi kappale ei vielä vaikuta peliin kovinkaan paljoa, mutta kaksi tai useampi kappale voi olla suuri hidaste pelille.

3 Simulointi ilman fysiikkamoottoria

Tässä luvussa esitetään ensiksi mahdollisimman yksinkertaisia funktioita, jotka eivät vielä käytä realistista fysiikkaa kappaleille eivätkä kaavoja simuloimaan kellumista. Nämä funktiot yrittävät vain näyttää mahdollisimman realistisilta, kuitenkin uhraamatta liikaa aikaa raskaille fysiikan laskuille. Näitä funktioita voi soveltaa peleissä, joissa kelluvuus on toissijainen seikka ja kellunta tarjoaa ennemmin yksityiskohtia pelille, kuin pelattavuutta ja mekaniikkaa. Näissä simulaatioissa kappaleet eivät tarvitse jäykän kappaleen dynamiikan ja painovoiman mallintamiseen käytettävää Rigidbody-komponenttia.

Simulointiin käytetään Unityn omaa kuutio-objektia, joka on vakiona kooltaan 1x1x1 metriä ja painaa kilon. Massalla ja koolla ei ole simuloinnin kannalta suurta merkitystä. Kuution sisälle luodaan vielä tyhjä Buoyancy.cs skripti kelluntaa varten. FPS on kuution kanssa noin 170, korkeimmillaan noin 175.

3.1 Kappale vedenpinnalle

Jotta saadaan kappaleen seuraamaan aallokkoa ja vedenpintaa, täytyy kappaleen paikan korkeus (y-koordinaatti) asettaa oikealla aallonkorkeudelle kappaleen koordinaateissa x ja z. Tätä varten voidaan käyttää WaveController.cs:n GetWaveYPos-funktiota (funktio 2.2), joka laskee aallokolle korkeuden x ja z koordinaattien perusteella. Samalla periaatteella voidaan käyttää kuution x ja z koordinaatteja, jolloin funktiolla palauttaa kuutiolle sen oikean korkeuden, oli se missä kohden maailmaa tahansa.

Kuution Buoyancy-skriptissä haetaan ensiksi WaveController peliobjekti Unityn FindWithTag-funktiolla[1], josta saatu WaveController-skriptin viite asetetaan *waves*-nimiseen muuttujaan, jotta funktiota ‘GetWaveYPos’ voidaan helposti kutsua.

Update-funktiossa 3.1, jota kutsutaan kerran jokaisella kuvapiirrolla voidaan asettaa kappaleen y-koordinaatin oikealle aallonkorkeudelle, jolloin kappaleen keskipiste pysyy aina oikeassa pisteessä aaltoihin nähden.

1. `https://docs.unity3d.com/ScriptReference/GameObject.FindWithTag.html`

Funktio 3.1: void Update()

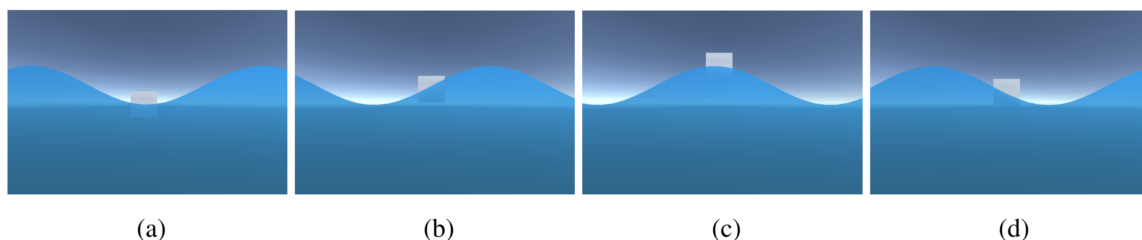
```
1 float x = transform.position.x;
2 float z = transform.position.z;
3 float y = waves.GetWaveYPos(x, z);
4 transform.position = new Vector3(x, y, z);
```

Tämä kuitenkin voi vaikuttaa epärealistiselta, jos aaltojen nopeus ja korkeus ovat tarpeeksi suuret: silloin kappaleen keskipiste pysyy tiukasti kiinni vedenpinnalla ja kappale voi liikkua epäluonnollisen nopeasti. Tähän voidaan ottaa avuksi Vector3-luokan SmoothDamp-funktio [2], joka pehmentää nopeaa liikettä. Funktio ottaa parametreina järjestyksessä nykyisen paikan, paikan johon halutaan liikkua, kyseisen nopeuden (asetetaan se nollaksi: Vector3.zero) ja arvon kuinka vahvasti halutaan liikettä pehmentää. Esimerkiksi smoothTime-arvolla 0.2 kappale liikkuu jo paljon luontevammin vahvemmassa aallokossa, eikä se pomppaa nopeasti pinnalle korkean aallon sattuessa kohdalle vaan kappale kohoaa hitaammin pinnalle. Haittavaikutuksena on kuitenkin, että myös liike alaspäin aaltojen mukana on pehmennetty ja ääritapauksissa kappale saattaa jäädä leijumaan ilmaan, jos aalto on tarpeeksi nopea ja aallonpituus tarpeeksi lyhyt, jolloin aallon mukana vedenpinta laskeutuu nopeammin, kuin SmoothDamp pystyy kappaletta liikuttamaan. Funktiossa voidaan ottaa kappaleen suunta huomioon, jolloin jos kappaleen liike on alaspäin, pehmennyksen nopeudelle voidaan asettaa uusi, pienempi arvo. Esimerkiksi arvolla 0.1 kappale liikkuu edelleen pehmeästi, mutta ei kuitenkaan kovemmassa aallokossa jää veden pinnan yläpuolelle. Tämä koodi 3.2 voidaan lisätä suoraan funktion 3.1 perään ja tuloksena saadaan kuvasarjojen 3 ja 4 mukainen liike.

Funktio 3.2: void Update()

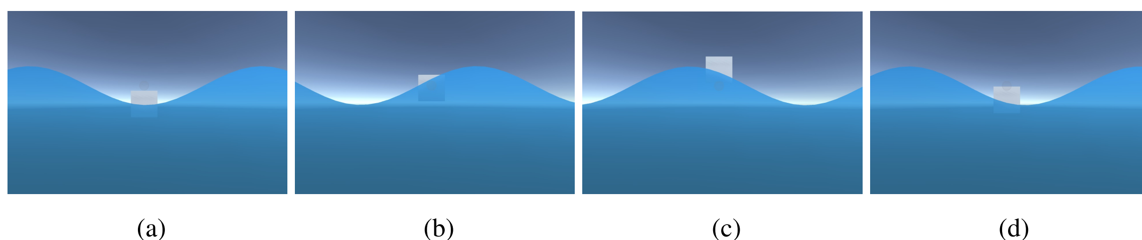
```
1 Vector3 targetPosition = new Vector3(x, y, z);
2 if (y < transform.position.y)
3     transform.position = Vector3.SmoothDamp(transform.position,
          targetPosition, ref velocity, smoothTimeDown);
4 else
5     transform.position = Vector3.SmoothDamp(transform.position,
          targetPosition, ref velocity, smoothTime);
```

2. https://docs.unity3d.com/ScriptReference/Vector3.SmoothDamp.html



(a) (b) (c) (d)

Kuvio 3: 2-ulotteinen kuvakulma hitaassa sini-aallossa



(a) (b) (c) (d)

Kuvio 4: Nopeampi sini-aalto, kappale jää selkeästi pinnan alle

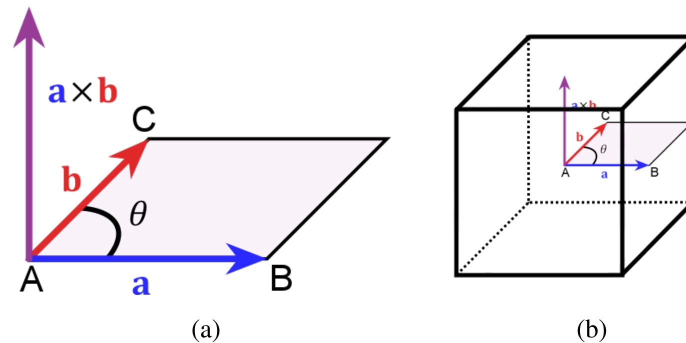
Tämä on yksinkertaisin funktio, jolla kappale saadaan pysymään veden pinnalla käyttämättä liikaa laskentatehoa toteutukseen. Kun näkymässä oli yksi kuutio, kuvataajuus oli keskimäärin 168 FPS. Sadan kuution tapauksessa kuvataajuus oli keskimäärin 165 FPS, joten kuutioiden määrän lisäämisen vaikutus suorituskykyyn oli vähäinen.

3.2 Kappaleen suunta ja kaltevuus aaltoon nähden

Edellisen osion funktio simuloi kappaleen kelluvuutta pelkästään aallokon korkeuden mukaan, mutta se ei ottanut huomioon itse kappaleen kallistumista ja kääntymistä aallokon mukaan, vaan kappale pysyi aina samassa asennossa ja liikkui vain paikallaan ylös ja alas. Realistisempi simulaatio saadaan aikaan, jos kappale mukailee vedenpinnan kulmaa kääntämällä itseään aaltojen kulman mukaan.

Vedenpinnan kulma aaltoihin nähden saadaan selville vektorien ristitulon avulla (kuva 5a). Tätä varten tarvitaan kolme pistettä kelluvan kappaleen sisältä, jotka ovat X ja Z koordinaateilla aina kappaleen mukana samassa kohdassa, mutta korkeuskoordinaatti Y on vedenpinnan korkeuden kohdalla. Kolme pistettä, jotka eivät ole samalla linjalla muodostavat aina tason (Martin 2012) ja tason normaali mukailee kappaleen kohdalla olevaa aaltoa. Tason nor-

maali voidaan laskea Unityn Vector3-luokkaa apuna käyttäen luoden kaksi suoraa vektoria joista toinen kulkee pisteestä A pisteeseen B ja toinen pisteestä A pisteeseen C. Olettaen että nämä vektorit eivät ole samansuuntaisia (vektorien ristitulo $\neq 0$), vektorien ristitulona saadaan aina vektori, joka on kohtisuorassa molempia vektoreita kohtaan, eli näiden pisteiden muodostaman tason normaalivektori (Elduque 2004).



Kuvio 5: (a) Vektorien ristitulo. (b) Vektorien ristitulo sijoitettuna kuution keskelle

Funktio 3.3: void AddTilt

```
A = transform.position;
B = transform.position + new Vector3(0.5f, 0f, 0f);
B = new Vector3(B.x, waves.GetWaveYPos(B.x, B.z), B.z);
C = transform.position + new Vector3(0f, 0f, 0.5f);
C = new Vector3(C.x, waves.GetWaveYPos(C.x, C.z), C.z);

right = B - A;
left = C - A;

direction = Vector3.Cross(left, right);
```

Funktiossa 3.3 pisteen A voidaan ajatella olevan kappaleen keskipiste, joka saadaan suoraan transform.position-arvolla. Pisteet B ja C saadaan siirtämällä koordinaatteja keskipisteestä x -ja z-akseleilla puolen yksikön verran, joka asettaa pisteet kuution reunoille kuva 5b mukaisesti. Eri muotoisille kappaleille pisteet täytyisi määrätä tarkemmin niiden kelluuntatason mukaan, mutta tavallisella kuutiolla tämä riittää. Molemmat pisteet ovat kuitenkin vielä samalla korkeudella kuin keskipiste, joten asetetaan näiden pisteiden y-koordinaatti vedenpin-

nan korkeudelle funktiossa 2.2. Näiden pisteiden väliset vektorit saadaan vähentämällä pisteen A koordinaatti pisteiden B ja C koordinaateista ("Computing a Normal/Perpendicular vector" 2018). Kuvan 5a vektori 'a' on funktion 3.3 muuttuja 'right' ja vektori 'b' on muuttuja 'left'. Lopuksi voidaan Unityn Vector3.Cross() -funktiolla laskea ristitulo ja tuloksena saadaan tason normaalivektori [3].

Funktio 3.4: void AddTilt

```
1 Vector3 lookat = transform.position + direction;
2 transform.LookAt(lookat, transform.up);
```

Kappaleen kääntäminen onnistuu Unityn LookAt()-funktiolla, joka kääntää kyseisen kappaleen forward-suunnan (joka on kappaleen posiitiivisen z-vektorin suuntaan) tiettyä pistettä kohti [4]. Funktiossa 3.4 voidaan luoda uusi koordinaattipiste 'lookat' lisäämällä kappaleen keskipisteesteen funktiossa 3.3 laskettu normaalivektori 'direction'. Tämä piste on koordinaattivektori kappaleen keskipisteestä normaalivektorin suuntaan, jota kohti kappale voi kääntää itsensä.

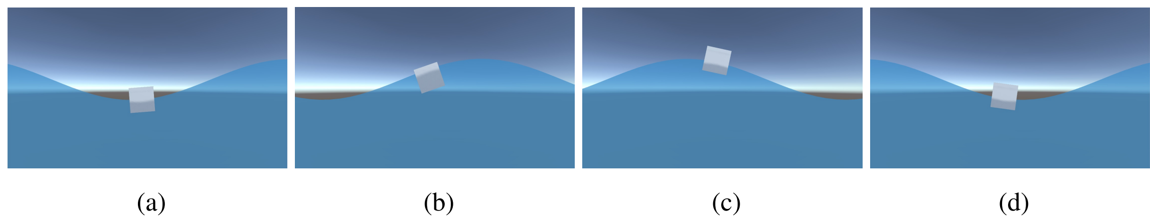
Funktio 3.5: void AddTilt

```
1 b2 = transform.position + new Vector3(-0.5f, 0f, 0f);
2 b2 = new Vector3(b2.x, waves.GetWaveYPos(b2.x, b2.z), b2.z);
3 c2 = transform.position + new Vector3(0f, 0f, -0.5f);
4 c2 = new Vector3(c2.x, waves.GetWaveYPos(c2.x, c2.z), c2.z);
5
6 right2 = b2 - a;
7 left2 = c2 - a;
8
9 direction = Vector3.Cross(left, right);
10 direction2 = Vector3.Cross(left2, right2);
11 direction = (direction + direction2) / 2;
```

Kappale ei kuitenkaan käänny vielä halutulla tavalla. Tämä johtuu siitä ettei alkuperäisen normaalivektorin muodostama kolmio kata ylhäältä katsottuna kuutiota aivan kokonaan kes-

3. https://docs.unity3d.com/ScriptReference/Vector3.Cross.html
4. https://docs.unity3d.com/ScriptReference/Transform.LookAt.html

keltä, vaan pelkästään toiselta sivulta. Tämän korjaamiseksi voidaan laskea funktion 3.5 tapaan vielä yksi kolmio vastakkaiselle puolelle ja käyttää tämän uuden normaalivektorin ja funktion 3.3 normaalivektorin keskiarvoa. Kappaletta käännettäessä osoittamaan tuloksena saadun keskiarvon suuntaan saadaan kuvasarjan 6 mukainen liike.



(a) (b) (c) (d)

Kuvio 6: Kappale kääntyy allokon mukana

4 Vedenalaisen kappaleen kolmioverkon muodostaminen

Luvun 3 malleilla simuloidessa kappaleet pystyvät olemaan suoranaisesti vuorovaikutuksessa vain veden ja aaltojen kanssa. Kappaleille ei lisätty Rigidbody-komponenttia, joka mahdollistaisi Unityn fysiikkamoottorin hyödyntämisen, jolloin niiden paikka ja liike ympäristössä on täysin kontrolloitu luvussa 3 laaditulla kelluvuus-skriptillä.

Seuraavaksi otetaan käyttöön Unityn tarjoamat fysiikkamoottorin ominaisuudet ja kehitetään kelluvuus-skripti, joka perustuu fysiikan lakeihin. Tällä tavoin yritetään saada kappaleen liikkeestä vedessä mahdollisimman realistinen, jolloin se myös simuloisi mahdollisimman monia siihen vaikuttavia voimia realististen kaavojen avulla. Simulointi tarvitsee käyttöönsä tiedon kappaleen kolmioverkon vedenalaisesta osasta, joten tämä luku käsittelee pelkästään sen selvittämistä. Luvussa 5 käytetään hyväksi tässä luvussa selvitettyä vedenalaista osiota ja sen perusteella siihen lasketaan tarvittavat nostevoimat.

Pelikentällä käytetään luvun 3 kuutiota, mutta luodaan objektille Rigidbody-komponentti ja uusi PhysicsBuoyancy.cs-skripti, johon voidaan toteuttaa kellunnan simuloinnissa tarvittavat funktiot. Vaihdetaan myös kuution paino aluksi vastaamaan hiukan realistisemmin kuution kokoisen kappaleen painoa. Kuutio koivua painaa keskimäärin 483 kg (Hakkila 1979) ja se sopii hyvin alkuun testikappaleeksi. Algoritmille käytetään hyvin pitkälti pohjana Nordeusin (2018) mallissa esitettyjä kaavoja ja funktioita, muokaten ja lisäten ominaisuuksia tarvittaessa.

4.1 Kappaleen vedenalaisen osuuden tunnistaminen

Luvussa 5.1 nosteen kokonaisvoiman kerrotaan olevan putoamiskiihtyvyyden, väliaineen tiheyden ja syrjäytetyn väliaineen tilavuuden summa, joten aloitetaan algoritmi selvittämällä tarkasti, mikä osa kappaleesta on veden alla. Aloitetaan luomalla uusi erillinen skripti, jonka tehtävänä on hoitaa kappaleen vedenalaisen osan laskeminen ja annetaan sille nimeksi 'UnderwaterMesh.cs' Skriptiltä voidaan ottaa pois Unityn MonoBehaviour-luokkaperintä sekä Start()- ja Update-funktiot. Skriptille voidaan lisätä valmiiksi vielä tyhjä konstruktori 'public UnderwaterMesh(GameObject parentObject)' ja vielä tyhjä funktio 'public void

GenerateUnderwaterMesh()’ vedenalaisen osan laskuun.

Lisätään seuraavaksi kelluntaskriptiin 4.1 (kappaleelle lisätty PhysicsBuoyancy.cs-skripti) tällä hetkellä tarpeelliset globaalit muuttujat, Start()-funktioon UnderwaterMesh-luokan alustus, sekä Update()-funktioon kutsu laskemaan vedenalaista osaa:

Funktio 4.1: public class PhysicsBuoyancy : MonoBehaviour

```
1  public GameObject underWaterObj;
2  private UnderwaterMesh underwaterMesh;
3  private Mesh debugMesh;
4
5  void Start()
6  {
7      underwaterMesh = new UnderwaterMesh(this.gameObject);
8      debugMesh = underWaterObj.GetComponent<MeshFilter>().mesh;
9  }
10 void Update()
11 {
12     underwaterMesh.GenerateUnderwaterMesh();
13     underwaterMesh.DisplayMesh(debugMesh, "Vedenalainen Osa",
           underwaterMesh.underWaterTriangleData);
14 }
```

Muuttuja ’underwatermesh’ on viittaus UnderwaterMesh-skriptiin, jotta jokaisella näyttöpiirrolla Update-funktiossa voidaan kutsua sen GenerateUnderwaterMesh-funktiota. Kappaleen vedenalaisen osan visualisointia varten kelluvaan kappaleeseen on luotava lapsi-objekti Unityn editorissa, johon liitetään komponenteiksi ’MeshFilter’, ’MeshRenderer’ ja uusi materiaali halutulla värillä. DisplayMesh-funktio (A.1) sisältää yksinkertaisen silmukan, joka käy UnderwaterMesh-luokan laskemat vedenalaiset kolmiot läpi ja muodostaa niistä yhtenäisen kolmioverkon tämän underwaterObj:tin MeshFilterin näytettäväksi. Funktio käy läpi luodun TriangleData-taulukon, joka sisältää ‘TriangleData’ (funktio A.3) kolmio-olioina arvot kulmapisteistä oikeassa järjestyksessä, jolloin funktiossa ei tarvitse kuin käydä kaikki yksitellen läpi ja lisätä niiden kolmiot järjestyksessä uuteen kulmapistelistaan ja kulmapistelistan vastaavat indeksit kolmiolistaan, jolloin Unityn Mesh-luokka osaa tulkita ne.

PhysicsBuoyancy-skripti voidaan jättää hetkeksi, sillä siihen ei tarvitse tehdä muutoksia ennen kuin kappaleen vedenalainen osa saadaan laskettua kokonaisuudessaan. Seuraavaksi luodaan funktiossa 4.2 UnderwaterMesh-skriptin globaalit muuttujat, joita tarvitaan algoritmin toimintaan.

Funktio 4.2: public class UnderwaterMesh

```
1 private Transform objectTransform;
2 private Vector3[] objectVertices;
3 private int[] objectTriangles;
4 public Vector3[] objectGlobalVertices;
5 private float[] distancesToSurface;
6 public List<TriangleData> underwaterTriangleData = new List<
      TriangleData>();
7 private WaveController waves = GameObject.FindWithTag("GameController"
      ).GetComponent<WaveController>();
```

’ObjectTransform’ on luokkaviittaus, joka sisältää alkuperäisen kappaleen Transform-luokan, jonka avulla voidaan vaihtaa kappaleen kulmapisteiden koordinaatisto paikallisesta globaaliin. ObjectVertices ja ObjectTriangles-taulukot sisältävät tiedot kappaleen kolmioverkosta. ObjectVertices-taulukko sisältää kaikki kolmioverkon käyttämät kulmapisteet Vector3-koordinaatti muodossa ja ObjectTriangles-taulukko sisältää järjestyksessä indeksit kolmioverkon muodostaviin kolmioihin. Esimerkiksi jos taulukko alkaisi: (5,12,7...), tarkoittaisi se sitä, että kolmioverkon ensimmäisen kolmion muodostaa ObjectVertices-taulukon kulmapisteet, jotka ovat indeksien 5, 12 ja 7 osoittamissa paikoissa.

’ObjectGlobalVertices’ on taulukko kolmioverkon kulmapisteistä, joiden koordinaatit on käännetty paikallisesta koordinaatistosta globaaliin, eikä kulmapisteiden järjestystä muuteta, jolloin näihin kulmapisteisiin voidaan käyttää samaa ObjectTriangles-kolmiotaulukkoa kuin alkuperäisiinkin kulmapisteisiin. ’DistancesToSurface’ on taulukko kulmapisteiden etäisyydestä veden pinnalle, jossa korkeuksien järjestys on sama kuin kulmapisteillä omassa taulukossaan, jolloin tietyn kulmapisteen korkeus saadaan samalla taulun indeksillä hakemalla tästäkin taulukosta.

’UnderwaterTriangleData’ on lista TrianlgeData structeista (apuluokka A.3). TriangleDa-

tan konstruktori ottaa parametreiksi kolmion kulmapisteiden koordinaatit ja sijoittaa ne globaaleiksi muuttujikseen. TriangleData sisältää kulmapisteiden koordinaattien lisäksi myös kolmion tason normaalivektorin, kolmion pinta-alan, kolmion keskipisteen ja keskipisteen etäisyyden vedenpintaan. Muuttujat lasketaan käyttämällä hyväksi annettuja kulmapisteiden koordinaatteja sekä WaveController-objektin GetWaveYPos-funktiota.

Waves-muuttuja sisältää viitteen luvussa 2 luotuun WaveController-objektiin, jonka funktiota 2.2 apuna käyttäen saadaan helposti kulmapisteiden etäisyys veden pintaan.

Lopuksi lisätään UnderwaterMesh-konstruktoriin alustukset kaikille muuttujille:

Funktio 4.3: public UnderwaterMesh(GameObject parentObject)

```
1 objectTransform = parentObject.transform;
2 objectVertices = parentObject.GetComponent<MeshFilter>().mesh.vertices
     ;
3 objectTriangles = parentObject.GetComponent<MeshFilter>().mesh.
     triangles;
4
5 objectGlobalVertices = new Vector3[objectVertices.Length];
6 allDistancesToWater = new float[objectVertices.Length];
```

Funktiossa 4.3 kappaleen Transform-objekti, kulmapistetaulukko ja kolmiotaulukko tulevat suoraan parametrina alkuperäisen kappaleen viitteestä. Globaalin koordinaatiston kulmapistetaulukko ja vedenpinnan etäisyystaulukko voidaan alustaa saman pituisiksi kuin alkuperäinen kulmapistetaulukko, sillä kulmapisteitä tulee olla saman verran.

Ennen kuin jatketaan tarkemmin kolmioverkon pilkkomiseen, voidaan testata toimivatko muuttujat ja funktiot normaalisti, sekä saadaanko visuaalisen vasteen vedenalaisesta kolmioverkon osasta. Tätä varten täytetään GenerateUnderwaterMesh-funktio silmukalla, joka käy kolmioita läpi ja tarkistaa onko niiden keskipiste veden alla. Jos näin on, lisätään kyseinen kolmio vedenalaiseen kolmioverkkoon, jolloin se näkyy pelimaailmassa.

Aloitetaan luomalla GenerateUnderwaterMesh-funktioon for-silmukka, joka vaihtaa kulmapisteiden koordinaatiston globaaliksi, jolloin saadaan pisteen todellisen paikan peliavaruudessa.

Funktio 4.4: void GenerateUnderwaterMesh()

```
1 underwaterTriangleData.Clear();
2 //Muunna koordinaatteja
3 for (int j = 0; j < objectVertices.Length; j++)
4 {
5     Vector3 globalPos = objectTransform.TransformPoint(objectVertices[
          j]);
6     objectGlobalVertices[j] = globalPos;
7
8     float surfaceLevel = waves.GetWaveYPos(globalPos.x, globalPos.z);
9     distancesToSurface[j] = globalPos.y - surfaceLevel;
10 }
11 AddTriangles();
```

Funktio 4.4 tyhjentää rivillä 1 taulukon vedenalaisista kolmioista, jotta se voidaan täyttää uusilla, ajankohtaisilla kolmioilla. Tämän jälkeen käydään läpi kaikki kulmapisteet ja transformoidaan niiden koordinaatisto kappaleen sisäisestä lokaalista koordinaatistosta globaaliksi rivillä 5. Tätä varten tarvitaan alkuperäisen kappaleen Transform-komponenttia. Rivillä 6 lisätään muutettu vektori tauluun, jotta se on helposti käytettävissä. Samalla lisätään rivillä 9 kulmapisteen etäisyys vedenpinnasta käyttämällä hyväksi aaltojen GetWaveYPos-funktiota ja juuri transformoitua kulmapisteen globaalia koordinaattia. Ennen seuraavaa vaihetta luodaan tyhjä AddTriangles-apufunktio ja kulmapisteiden käsittelyyn apuluokka A.2 ‘VertexData’. Luokka sisältää kyseisen kulmapisteen etäisyyden vedenpintaan, indeksin siitä kuinka mones kulmapiste se on koko kolmiosta (0, 1 tai 2), sekä kulmapisteen globaalin koordinaatin. Tämän apuluokan avulla voidaan päästä helposti käsiksi tietyn kulmapisteen korkeuteen ja indeksiin, ilman että niitä tarvitsisi erikseen laskea uudestaan.

Seuraavana täytetään AddTriangles-apufunktio, jonka tarkoituksena on selvittää mitkä kulmapisteet ovat kulloinkin veden alla ja muodostaa niistä kolmioverkko. Tässä vaiheessa kuitenkin funktio käy läpi valmista kolmioverkkoa ja selvittää kokonaisista kolmioista, onko niiden keskipiste veden alla, jolloin koko kolmio lisätään veden alla olevaksi.

Funktio 4.5: void AddTriangles()

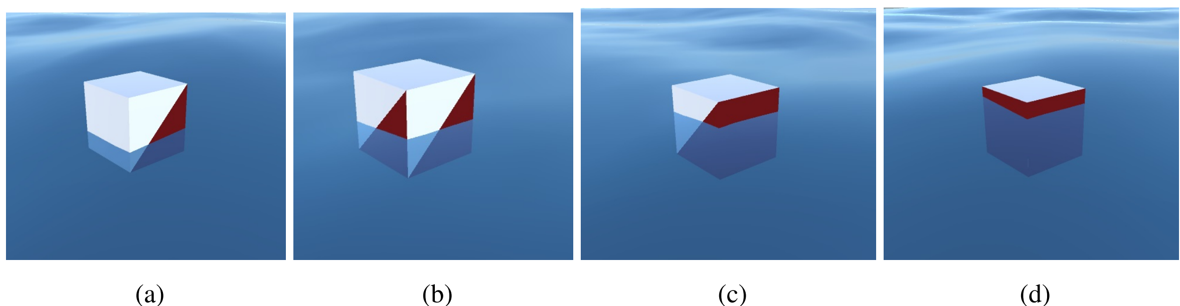
```
1 //Lista kolmion kulmapisteistä, joita käsitellään nyt
2 List<VertexData> vertexData = new List<VertexData>();
3 //Alustukset kolmelle kulmapisteelle
4 vertexData.Add(new VertexData());
5 vertexData.Add(new VertexData());
6 vertexData.Add(new VertexData());
7 //Käydään kulmapisteet kolmioittain läpi (kolme pistettä = yksi kolmio
    )
8 int i = 0;
9 while (i < objectTriangles.Length)
10 {
11     //Käy kulmapisteet kolme kerrallaan läpi
12     for (int x = 0; x < 3; x++)
13     {
14         vertexData[x].index = x;
15         vertexData[x].globalVertexPos = objectGlobalVertices[
               objectTriangles[i]];
16         vertexData[x].distance = distancesToSurface[objectTriangles[i
               ]];
17         i++;
18     }
19     Vector3 p1 = vertexData[0].globalVertexPos;
20     Vector3 p2 = vertexData[1].globalVertexPos;
21     Vector3 p3 = vertexData[2].globalVertexPos;
22
23     var tempTriange = new TriangleData(p1, p2, p3);
24     float surfaceLevel = waves.GetWaveYPos(tempTriange.center.x,
          tempTriange.center.z);
25     if (surfaceLevel - tempTriange.center.y > 0)
26         underwaterTriangleData.Add(new TriangleData(p1, p2, p3));
```

Funktio etenee alkuun luomalla listan, joka sisältää sillä hetkellä tarkasteltavan kolmion kolme kulmapistettä. Tähän listaan ei lisätä tai poisteta alkioita, vaan myöhemmän while-silmukan sisällä vaihdetaan listan sisällä olevien VertexData-alkioiden arvot uusiin. While-silmukka käy läpi kolmiotaulukkoa, joka sisältää kolmioverkon yksittäisten kolmioiden in-

deksit kolmen joukoissa. Tämän takia rivillä 12 olevaa for-silmukkaa käydään kolmesti läpi. Silmukan aikana kerätään talteen objectGlobalVertices-taulukosta kolme seuraavaa kulmapistettä ja lisätään jokaisen for-silmukan jälkeen while-silmukan indeksiä yhdellä. Tällöin, kun while-silmukka alkaa uudestaan, indeksissä ollaan hypätty kolmen kulmapisteen yli, jolloin vuorossa on seuraavan kolmion kolme kulmapistettä.

Kun käsittelyyn on saatu kolmion kolme kulmapistettä, saadaan kolmion keskipiste laskettua luomalla niistä TriangleData-apuluokka. Tämän keskipisteen korkeuskoordinaatti (y) vähennettynä keskipisteen kohdalla olevasta aallonkorkeudesta saadaan etäisyyden vedenpintaan. Tämän jälkeen lisätään TriangleData underwaterTriangleData-listaan, jos etäisyys on suurempi kuin nolla.

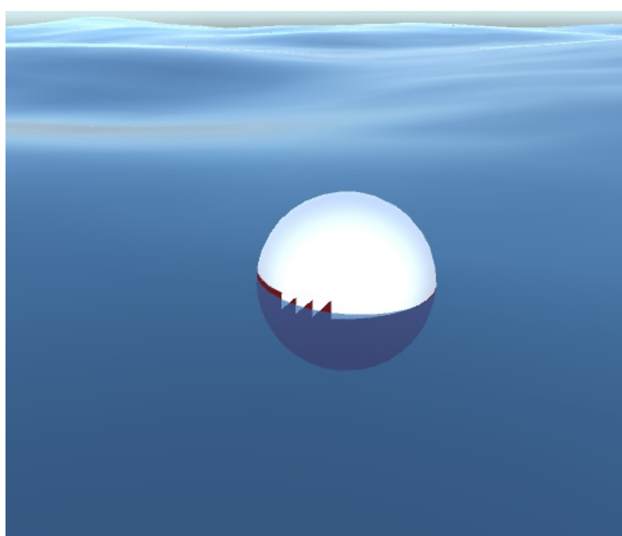
Tuloksena kappaleen päälle ilmestyy punaisia kolmioita riippuen siitä, onko kyseisen kolmion keskipiste veden alla vai ei. Kappaleen Rigidbody-komponentin painovoiman aktivoiva Use Gravity-vaihtoehto on syytä ottaa pois päältä väliaikaisesti, sillä algoritmi ei vielä lisää nostetta kappaleelle ja kappale muutoin alkaisi vapaapudotuksen. Paikallaan pysyvästä kappaleesta näkee selvemmin ohi liikkuvien aaltojen vaikutuksen.



(a) (b) (c) (d)

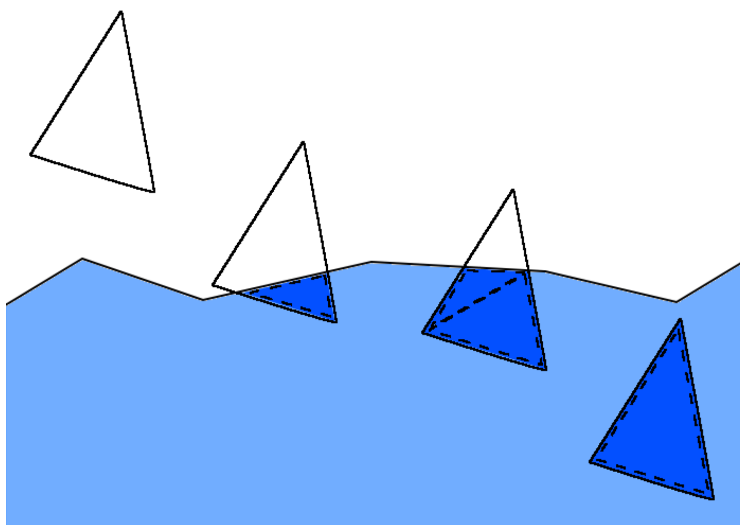
Kuvio 7: 3-ulotteinen kuvakulma

Kuvio 7 esittää kuinka GenerateUnderwaterMesh-funktio luo vedenalaisia osia kappaleelle aallon liikkuessa ohi ja punaisella värillä olevat kolmiot osoittavat algoritmin vedenalaiseksi osiksi tulkitsemia kolmioita. Algoritmin toiminnan näkee hiukan paremmin kappaleella, jolla on enemmän ja pienempiä kolmioita kolmioverkossaan, kuten kuviossa 8 esitetyllä pallolla. Pienemmät kolmiot tuottavat näin tarkemman tuloksen, sillä isoissa kolmioissa suuri osa pinta-alasta voi olla pinnan yläpuolella, vaikka keskipiste olisikin veden alla kuten kuviossa 7 näkyy. Tästä syystä AddTriangles-funktioon on lisättävä enemmän logiikkaa, jonka avulla



Kuvio 8: Tiheämmällä kolmioverkolla varustettu kappale ja sen vedenalaiset osat

algoritmi tarkistaa kolmion keskipisteen sijaan kulmapisteiden etäisyyden vedenpintaan ja muodostaa uusia kolmioita pilkkomalla niitä kolmioita, jotka ovat osittain pinnan alla.

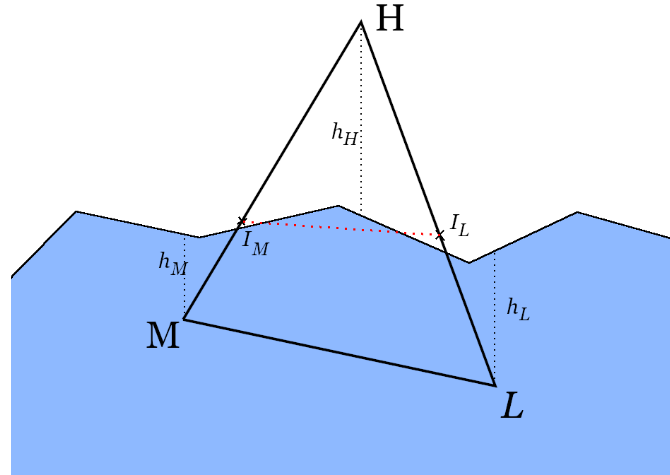


Kuvio 9: Tavat, jolla kolmiota käsitellään, riippuen mitkä osat siitä ovat veden alla. Kuvan pohjana käytetty Kerner (2016) mallia

Jos kolmion kaikkien kulmapisteiden etäisyys vedenpintaan on alle nolla, on kolmio kokonaisuudessaan veden alla ja kolmio voidaan lisätä suoraan underwaterTriangleData-listaan samalla tyylillä kuin funktiossa 4.5 (Kuvio 9, oikeanpuoleisin kolmio). Jos kaikkien kulmapisteiden etäisyys vedenpintaan on enemmän tai yhtä paljon kuin nolla, kolmio on kokonaan

pinnan yläpuolella ja kolmio voidaan jättää huomiotta ja siirtyä seuraavaan silmukassa (Kuvio 9, vasemmanpuoleisin kolmio). Jos kumpikaan ehto ei ole totta, tarkoittaa se sitä, että kolmiolla on vain joko yksi tai kaksi kulmapistettä veden alla, jolloin kolmio täytyy pilkkoa sopivan kokoisiksi palasiksi (Kuvio 9 kaksi keskimmäistä kolmiota). Tätä varten kolmion kulmien vertexData-lista voidaan järjestää vedenpinnan etäisyyden mukaan, korkeammasta matalimpaan. Tällöin on helpompi selvittää, onko kolmiolla yksi vai kaksi kulmapistettä veden alla. Jos vain ensimmäinen kulmapiste on suurempi tai yhtäsuuri kuin nolla ja loput alle nollan, on kolmion kaksi kulmapistettä veden alla. Muussa tapauksessa jos kaksi ensimmäistä pistettä on veden päällä, on kolmiolla vain yksi kulmapiste vedenpinnan alapuolella. Tapa, jolla approksimoidaan vedenalaisten kolmioiden pilkonta pohjautuu pääosin Gamasutran artikkeliin 'Water interaction model for boats in video games' (Kerner 2015), jonka kirjoittaja on ohjelmistokehittäjä Avalanche Studiosissa.

4.2 Kaksi kulmapistettä veden alla



Kuvio 10: Kolmio, jonka kaksi kulmapistettä ovat veden alla ja kolmion pilkkomiseen tarvittavat arvot ja pisteet. Kuvan pohjana käytetty Kerner (2016) mallia

Jos oletetaan esimerkkitapahtumaksi kuvio 10, nähdään että vedenpinta leikkaa sivun MH pisteessä I_M ja sivun LH pisteessä I_L, jolloin tämän kolmion vedenalainen osa saadaan kahdesta kolmiosta, jotka muodostuvat pisteistä (M, I_M, I_L) ja (M, I_L, L). Kuviosta nähdään, että leikkaus ei välttämättä ole täydellinen, mutta kuitenkin riittävä approksimaatio sillä kolmio-

verkon kolmiot ovat usein tarpeeksi pieniä, että virheen voi jättää huomiotta. Pisteet I_M ja I_L voidaan selvittää laskutoimituksilla

$$\overrightarrow{MI_M} = t_M \overrightarrow{MH} \tag{4.1}$$

ja

$$\overrightarrow{LI_L} = t_L \overrightarrow{LH}. \tag{4.2}$$

Huomioitavaa kaavoissa 4.1 ja 4.2 on tulokset, jotka eivät ole suoraan *koordinaattivektoreita* vaan *suuntavektoreja* pisteestä pisteeseen. Apumuuttujat t_M ja t_L voidaan selvittää käyttämällä hyväksi pisteiden etäisyyksiä vedenpinnasta:

$$t_M = \frac{-h_M}{(h_H - h_M)} \tag{4.3}$$

ja

$$t_L = \frac{-h_L}{(h_H - h_L)}. \tag{4.4}$$

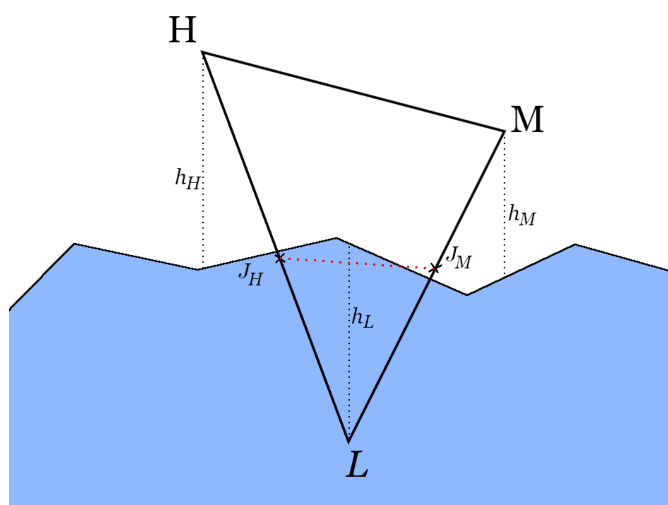
Funktio 4.6: void AddTrianglesOneAboveWater(List<VertexData> vertexData)

```
1 //Piste I_M
2 Vector3 MH = H - M;
3 float t_M = -h_M / (h_H - h_M);
4 Vector3 MI_M = t_M * MH;
5 Vector3 I_M = MI_M + M;
6 //Piste I_L
7 Vector3 LH = H - L;
8 float t_L = -h_L / (h_H - h_L);
9 Vector3 LI_L = t_L * LH;
10 Vector3 I_L = LI_L + L;
11 underwaterTriangleData.Add(new TriangleData(M, I_M, I_L));
12 underwaterTriangleData.Add(new TriangleData(M, I_L, L));
```

Unityn puolella käsiteltävän kolmion kulmapisteiden koordinaatit ja etäisyydet vedenpinnasta on tallennettu valmiiksi VertexData-olioihin, jolloin laskujen toteuttaminen on helppoa. Kuitenkin siirrettäessä näitä kaavoja Unityyn on huomioitava että Vector3-luokka toimii samanaikaisesti sekä *koordinaattivektorina* että *suuntavektorina*, jolloin näiden kahden välillä voi joskus tulla epäselvyyksiä. Esimerkiksi VertexData-luokan globalVertexPos-muuttuja on suoraan koordinaattivektorit M, H tai L ja kaavan 4.1 suuntavektori $\overrightarrow{MH}$ saadaan koordinattivektoreista laskemalla $H - M$. Vuorostaan juuri lasketusta suuntavektorista $\overrightarrow{MI_M}$ saadaan koordinaattivektori I_M laskemalla $\overrightarrow{MI_M} + M$.

Funktiossa 4.6 lasketaan vedenpinnan leikkaavat uudet kulmapisteet sivuilla MH ja LH joiden avulla muodostetaan uudet vedenalaiset kolmiot, jotka lisätään underwaterTriangleData-listaan. Funktiossa oletetaan, että oikeat koordinaatit M, H, L ja korkeudet h_M, h_H, h_M on selvitetty etukäteen VertexData-listalta.

4.3 Yksi kulmapiste veden alla



Kuvio 11: Kolmio, jonka yksi kulmapiste on veden alla ja kolmion pilkkomiseen tarvittavat arvot ja pisteet. Kuvan pohjana käytetty Kerner (2016) mallia

Kuvan 11 mukaisessa tilanteessa, jossa vain yksi kulmapiste on vedenpinnan alapuolella, kolmiota ei tarvitse pilkkoa kuin kerran kolmioksi pisteistä (L, J_H, J_M). Vedenpinnan leikkaavat pisteet J_H ja J_M sivuilla LH ja LM saadaan laskettua samanlaisilla kaavoilla kuin

luvussa 4.2:

$$\overrightarrow{LJ_H} = t_H \overrightarrow{LH} \tag{4.5}$$

ja

$$\overrightarrow{LJ_M} = t_M \overrightarrow{LM}, \tag{4.6}$$

sekä apumuuttujat:

$$t_H = \frac{-h_L}{(h_H - h_L)} \tag{4.7}$$

ja

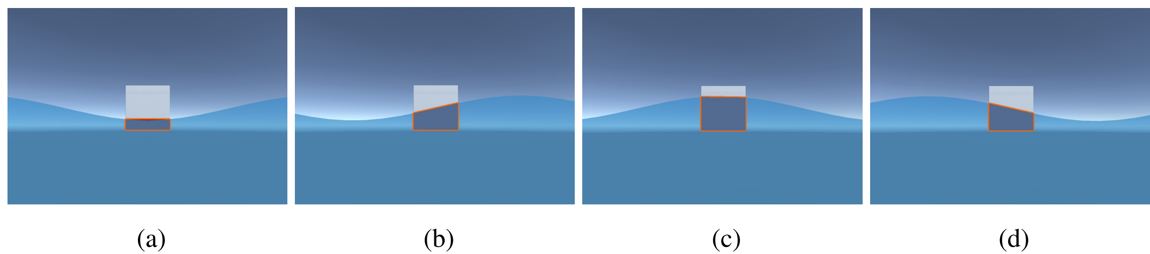
$$t_M = \frac{-h_L}{(h_M - h_L)}. \tag{4.8}$$

Kaavat muodostavat funktion samaan tapaan kuin refscript:AddTriangeOne, mutta tällä kertaa underwaterTriangleData-listaan lisätään vain yksi kolmio:

Funktio 4.7: void AddTrianglesTwoAboveWater(List<VertexData> vertexData)

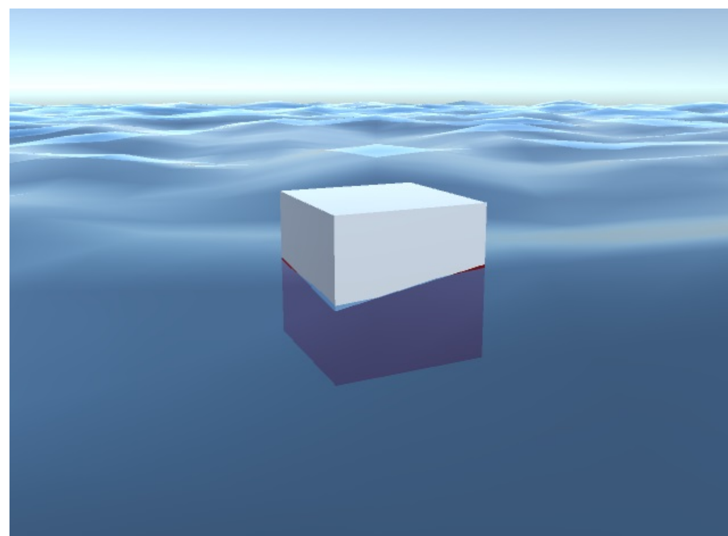
```
1 //Piste J_H
2 Vector3 LH = H - L;
3 float t_H = -h_L / (h_H - h_L);
4 Vector3 LJ_H = t_H * LH;
5 Vector3 J_H = LJ_H + L;
6 //Piste J_M
7 Vector3 LM = M - L;
8 float t_M = -h_L / (h_M - h_L);
9 Vector3 LJ_M = t_M * LM;
10 Vector3 J_M = LJ_M + L;
11 underwaterTriangleData.Add(new TriangleData(L, J_H, J_M));
```

Kun lisätään nämä kaksi funkiota mukaan, saadaan kuvan 13 mukainen, paljon tarkempi approksimaatio kappaleen vedenalaisesta osasta. Aallon mennessä yksittäisen kolmioverkon kolmion halki, kuvasarjasta 12 nähdään kuinka funktio pilkkoo kolmioverkon mukailemaan aallon pintaa.



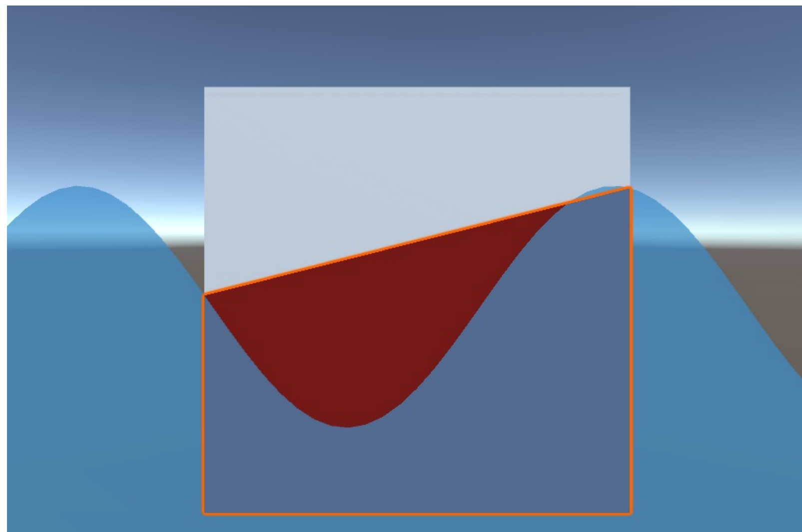
(a) (b) (c) (d)

Kuvio 12: Kaksiulotteinen näkymä aallon liikkeestä kuution läpi



Kuvio 13: Kolmiulotteinen näkymä

Laskettavien kolmoiden koon ja aallokon koon suhde aiheuttaa epätarkkuutta kappaleen vedenalaisen osan laskemiseen, kuten kuviossa 14 nähdään. Kuviossa kuutio on suurennettu 100-kertaiseksi, jolloin sivujen pinta-ala on hyvin suuri aaltoihin nähden, mutta sivut koostuvat edelleen vain kahdesta kolmiosta.



Kuvio 14: Kolmioiden ja aallokon koon kontrasti tuottaa epätarkkuutta

5 Fysiikan mallien mukainen simulointi

Mistä syystä kappaleet kelluvat tai uppoavat veteen joutuessaan? Tässä luvussa käsitellään, mistä tämä ilmiö johtuu ja mitkä asiat siihen vaikuttavat.

5.1 Kelluvuuden teoria

Kappaleen kelluvuuden aiheuttaa ilmiö nimeltä noste (eng. Buoyancy), joka on yksinkertaisimmillaan voima, joka nostaa kappaleita ylöspäin. Tämä voima tunnetaan paremmin Arkhimedeen lakina (Serway A. ja J. 2009), joka esitetään seuraavasti:

`"Jos kappale on upotettu nesteeseen, kappaleeseen kohdistuu ylöspäin työntävä voiman, joka on yhtä suuri kuin kappaleen syrjäyttämän nestemäärän paino."`

Nosteen matemaattinen kaava on

$$F_N = m_{neste}g = p_{neste}V_u g, \tag{5.1}$$

jossa F_N on nosteen voima, m_{neste} on syrjäytetyn nesteen tai kaasun massa, p_{neste} on nesteen tai kaasun tiheys, V_u on kappaleen upoksissa olevan osan tilavuus ja g on putoamiskiihtyvyys.

Kaavassa ei kuitenkaan ole mitään merkintöjä liittyen upotetun kappaleen massaan ja se tarkoittaa, että nosteen voimakkuuteen ei vaikuta itse kelluvan tai uppoavan kappaleen massa tai tiheys, vaan kappaleen tilavuus väliaineessa ja kyseisen väliaineen tiheys.

Esimerkiksi tilavuudeltaan yhden litran oleva esine upotettuna huoneenlämpöiseen veteen aiheuttaa

$$1kg \cdot 9,81m/s^2 = 9,81N \tag{5.2}$$

suuruisen nosteen. Kappaleen ei tarvitse olla kokonaan upotettuna väliaineeseen, vaan kappaleeseen vaikuttaa aina sen suuruinen noste, jonka kappale syrjäyttää tilavuudellaan väliainetta. Sivumainintana pitää huomioida, että noste ei tapahdu pelkästään vedessä tai edes nesteessä, vaan kaikki kaasut ja nesteet aiheuttavat nostetta. Esimerkiksi edellisen esimerkin

yhden litran kappale kokee myös kuivalla maalla ilmakehän nosteen seuraavasti:

$$0,001m^3 \cdot 1,2kg/m^3 \cdot 9,81m/s^2 = 0,01N. \quad (5.3)$$

Kelluntaan vaikuttaa myös painovoima

$$F_P = m_{kappale}g, \quad (5.4)$$

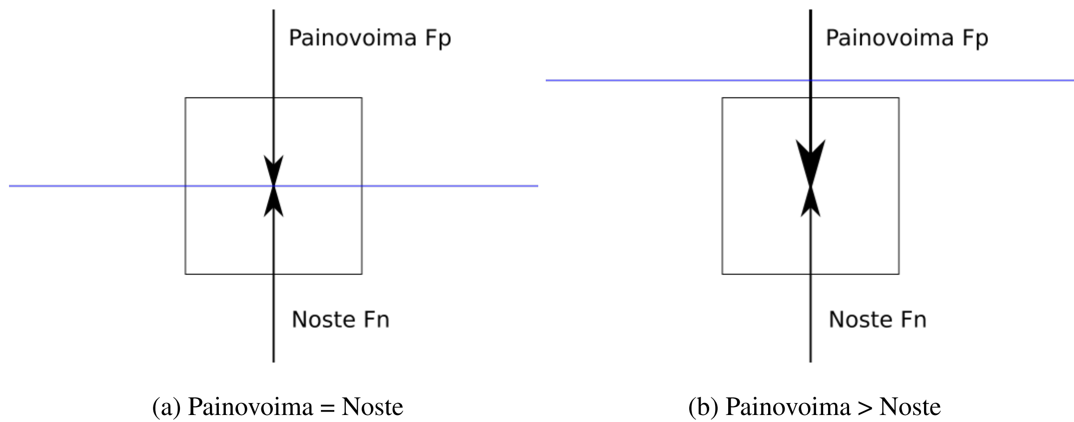
jossa m on kappaleen massa ja g on putoamiskiihtyvyys. Siinä missä noste on voima ylöspäin, painovoima on voima alaspäin. Kappaleeseen vaikuttava kokonaisvoima saadaan yhdistämällä noste ja painovoima ("Fluid Simulation for Video Games (Part 9)" 2012)

$$F_K = m_{kappale}g - p_{neste}V_ug, \quad (5.5)$$

jossa kokonaisvoiman F_K etumerkki kertoo kappaleen tilan. Positiivinen arvo tarkoittaa, että kappale uppoaa ja kokonaisvoima on alaspäin, kun taas negatiivinen arvo on päinvastaiseen suuntaan ja kappale kohoaa. Kelluminen syntyy kun F_K on nolla, eli noste ja painovoima ovat yhtä suuret, jolloin voimat kumoavat toisensa ja kappale pysyy lepotilassa (kuvio 15). Jos tässä tilassa kelluvan kappaleen päälle asetettaisiin toinen kappale, uuden painon johdosta noste ja painovoima eivät ole enää tasapainossa ja kappaleet alkavat upota. Samalla kuitenkin kappaleet syrjäyttävät enemmän vettä uppoamalla, jolloin myös noste kasvaa. Jos noste kasvaa tarpeeksi suureksi, jolloin se on taas yhtäsuuri kuin painovoima, kappale päätyy lepotilaan.

5.2 Massakeskipiste ja nostevoiman keskipiste

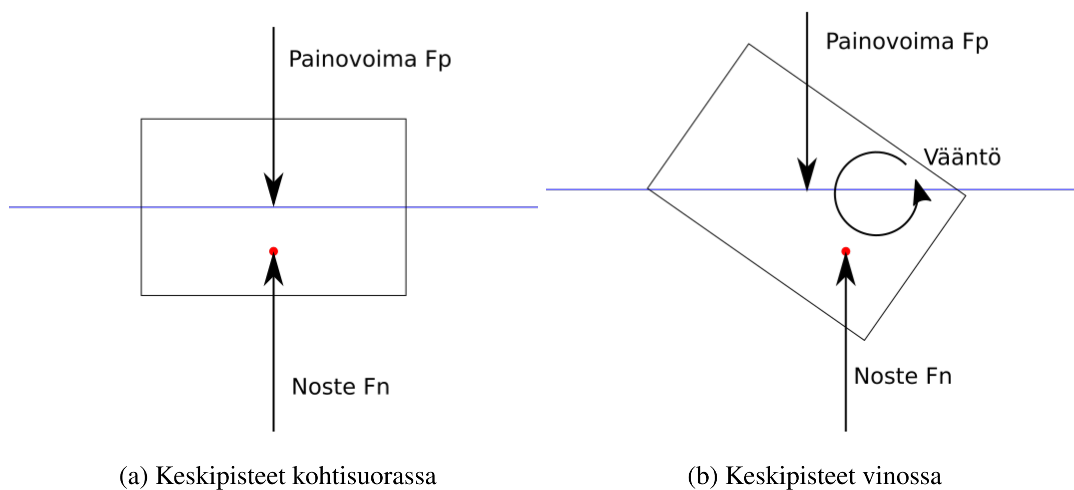
Painovoima (5.4) ja noste (5.1) vaikuttavat kappaleeseen tietyissä pisteissä ja nämä ovat nimeltään massakeskipiste ja nostevoiman keskipiste. Painovoima vaikuttaa kappaleen massakeskipisteestä suoraan alaspäin, kun taas noste vaikuttaa kappaleen nostevoiman keskipisteestä suoraan ylöspäin. Siinä missä massakeskipiste on aina samassa kohtaa kappaletta (olettaen, että kappale on täysin kiinteä, eikä sen sisällä ole muita liikkuvia kappaleita tai nesteitä), nostevoiman keskipiste on aina syrjäytetyn *väliaineen massakeskipisteessä*, jolloin se ei ole aina samassa paikassa (Matusiak 1995).



(a) Painovoima = Noste (b) Painovoima > Noste

Kuvio 15: (a) Kappale pysyy levossa. (b) Kappale uppoaa.

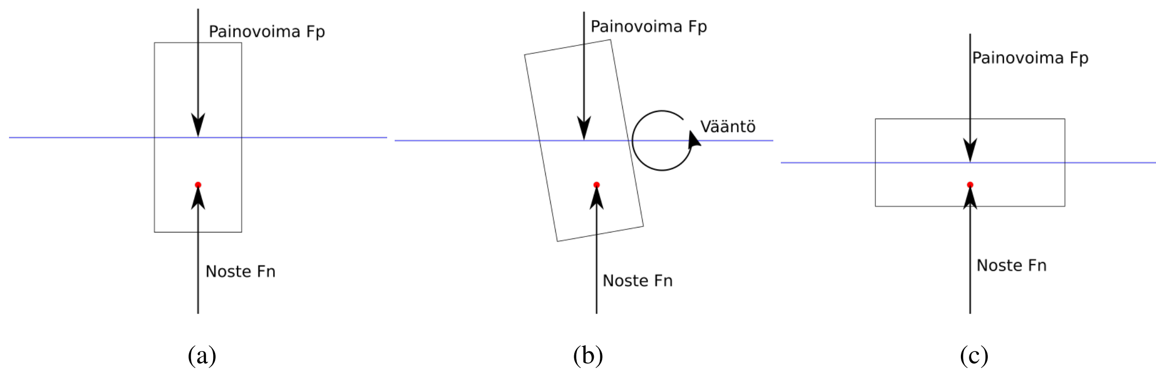
Kuvio 16 havannoillistaa, että kappale on levossa silloin, kun nämä kaksi pistettä ovat samalla pystysuoralla linjalla toisiinsa nähden: tällöin nosteen ja painovoiman vektorit kohtaavat suoraan ja kumoavat toisensa. Jos pisteet eivät ole pystysuoralla linjalla toisiinsa nähden, noste ja painovoima aiheuttavat kappaleeseen väännön, joka pyrkii kääntämään kappaleen taas asentoon, jossa massakeskipiste ja nostevoiman keskipiste ovat pystysuoralla linjalla.



(a) Keskipisteet kohtisuorassa (b) Keskipisteet vinossa

Kuvio 16: (a) Kappale pysyy levossa. (b) Kappaleeseen kohdistuu vääntö

Kuviosta 17 nähdään, että kappale voi myös olla levossa mutta ei vakaa, sillä jos kappaletta häiritsemällä se ei palaa enää alkutilaan, voidaan kappaleen alkutila todeta epävakaaksi.



Kuvio 17: (a) Kappale on levossa. (b) Kappaleeseen kohdistuu vääntö. (c) Kappale on levossa

5.3 Nostevoiman lisääminen kappaleelle

Tässä kohtaa Nordeus (2018) käyttää hiukan poikkeavaa tapaa laskea kelluvuuden voimaa kuin luvussa 5.1 esitetty nosteen kaava 5.1. Sen sijaan, että nosteen voima olisi yksi voimavektori vedenalaisen kappaleen keskipisteestä, Nordeus laskee voimavektorin jokaiselle vedenpinnan alaiselle kolmiolle erikseen. Kaavana hänellä on

$$F_N = p_{neste} V g, \tag{5.6}$$

jossa F_N on nosteen voima, p_{neste} on nesteen tiheys, V on veden tilavuus suoraan kolmion pinnan yläpuolella ja g on putoamiskiihtyvyys. Kolmion tason päällä olevan veden tilavuus lasketaan kaavalla

$$V = zSn, \tag{5.7}$$

jossa z on kolmion keskipisteen etäisyys pinnalle, S on kolmion pinta-ala ja n on kolmion pinnan normaalivektori. Tämä malli perustuu veden paineeseen joka on myös yhteydessä nosteeseen ja kuinka paine kasvaa syvemmälle mentäessä (Kerner 2015).

Kaava lisätään kellutettavan kappaleen alkuperäiseen kelluvuusskriptin 4.1 FixedUpdate()-funktioon. Tällöin nosteen voima lisätään kappaleeseen tasaisesti riippumatta sovelluksen kuvataajuudesta, sillä siinä missä Update()-funktio ajetaan kerran jokaisen kuvapiirron yhteydessä, FixedUpdate()-funktio ajetaan aina tasaisin väliajoin, noin 50 kertaa joka sekunti. Tästä syystä FixedUpdate()-funktiossa on hyvä käyttää fysikaalisten laskujen lisäämiseen [1].

1. `https://docs.unity3d.com/ScriptReference/MonoBehaviour.FixedUpdate.html`

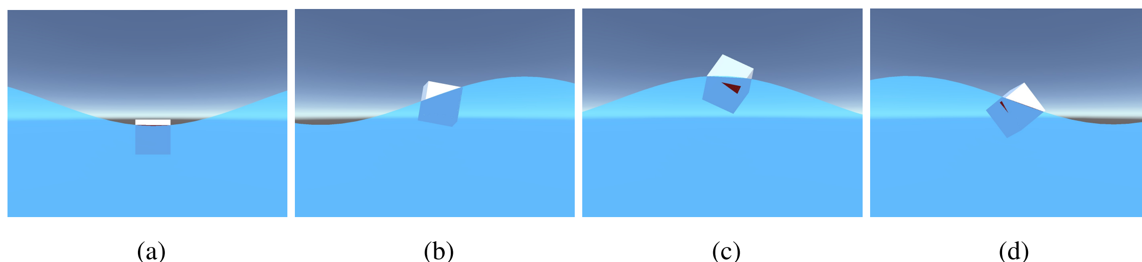
FixedUpdate()-funktiossa saadaan helposti skriptin 4.2 underwaterTriangleData-muuttujasta tarvittavat vedenalaiset kolmiot. Aluksi tarkistetaan vedenalaisten kolmioiden määrä, sillä jos vedenalaisia kolmioita ei ole, kappale on kokonaan vedenpinnan yläpuolella, eikä nostetta tarvitse laskea. Muussa tapauksessa käydään kolmiolistan alkiot läpi funktiossa 5.1.

Funktio 5.1: void AddPressureForce()

```
1 List<TriangleData> underWaterTriangleData = underwaterMesh.
    underwaterTriangleData;
2 for (int i = 0; i < underWaterTriangleData.Count; i++)
3 {
4     TriangleData triangleData = underWaterTriangleData[i];
5     Vector3 buoyancyForce = rhoWater * Physics.gravity.y *
        triangleData.distanceToSurface * triangleData.area *
        triangleData.normal;
6     buoyancyForce.x = 0f;
7     buoyancyForce.z = 0f;
8     thisRigidBody.AddForceAtPosition(buoyancyForce, triangleData.
        center);
9 }
```

Funktion 5.1 rivillä 5 on yhdistetty kaavat 5.6 ja 5.7, jossa lasketaan nesteen tiheys, painovoima, kolmion etäisyys pintaan, kolmion pinta-ala ja kolmion normaalivektorin tulo, josta saadaan vastaukseksi nosteen voimavektori kyseiselle kolmiolle. Myös Matusiak 1995 toteaa kirjassaan, että ’kelluvalla kappaleella ei esiinny vaakasuoria voimia’, jolloin vektorista voidaan poistaa vaakatason x ja z vektorit. Tällöin vektorille jäljelle jää vain pystysuuntainen voima y. Tuloksena saatu vektori lisätään kappaleesen Unityn AddForeAtPosition-funktiolla, jolloin saadaan voimavektori vaikuttamaan kappaleessa oikeaan kohtaan, eli jokaisen kolmioverkon kolmion keskipisteeseen.

Näillä lisäyksillä kappale kelluu veden pinnalla, mutta koska kappaleeseen ei erikseen lisätä vastuksia vedestä tai ilmasta, pyörii kappale holtittomasti itsensä ympäri kuten kuvasarja 18 osoittaa.



(a) (b) (c) (d)

Kuvio 18: Kappale kelluu, mutta pyörii holtittomasti

5.4 Liikettä vastustavien voimien lisääminen

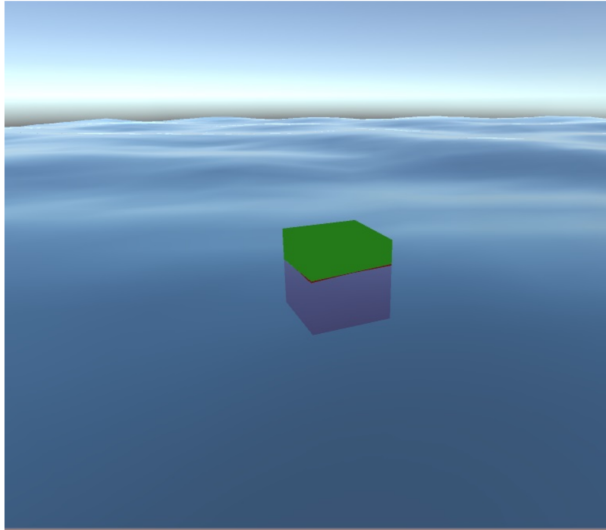
Kappale kelluu nyt realististen fysiikan voimien mukaisesti, mutta kappaleen liike vedessä ei vielä vastaa realistista liikettä, sillä kappaleeseen ei kohdistu mitään muita voimia, kuten veden- ja ilmanvastuksia.

Veden aiheuttamat voimat pystytään laskemaan hyödyntäen luvussa 4.1 selvitettyä kappaleen vedenalaisen osan kolmioverkkoa, mutta esimerkiksi ilmanvastuksiin tarvitaan vielä kappaleen vedenpäällisen osion kolmioverkko. Kuten vedenalaista kolmioverkkoakin varten, luodaan kappaleelle kolmioverkon visualisointia varten lapsi-objekti, joka sisältää MeshFilter- ja MeshRenderer-komponentit. Itse kolmioverkon muodostaminen onnistuu kuitenkin helposti käyttämällä hyödyksi funktioita 4.5, 4.6 ja 4.7, joihin lisätään muutamalla rivillä veden päällisten kolmioiden lisääminen.

Funktiossa 4.5 tarkistetaan, ovatko kaikki kulmapisteet veden alla, jolloin koko kolmio lisätään vedenalaiseen kolmioverkkoon. Vastaavasti voidaan lisätä ehto, jossa kaikkien kulmapisteiden ollessa veden päällä lisätään koko kolmio vedenpäälliseen kolmioverkkoon. Kolmion ollessa osittain veden alla voidaan kolmiot muodostaa käyttäen apuna jo laskettuja apupisteitä kolmion sivuilla. Kahden kulmapisteen ollessa veden alla voidaan lisätä pisteistä (I_M, H, I_L) muodostuva kolmio veden päälliseen kolmioverkkoon (kuvio 10). Kun yksi kulmapiste on veden alla, lisätään vedenpäälliseen kolmioverkkoon pisteistä (J_H, H, M) ja (J_H, M, J_M) muodustuvat kolmiot (ks. kuvio 11).

Kun nämä uudet ehdot on lisätty koodiin, saadaan selvitettyä kappaleen vedenpäällinen osa vaikuttamatta suuremmin laskentatehoon, sillä kaikki tarvittavat laskutoimitukset tehdään jo

muutenkin vedenalaista osaa selvitettäessä. Kun laskettu kolmioverkko asetetaan näkyviin värillisenä pintana, nähdään, että vedenpäällinen osuus sijoittuu oikein (kuvio 19).



Kuvio 19: Vedenalainen ja vedenpäällinen osa visualisoituna

Kun molemmat kolmioverkot on laskettu, voidaan niiden perusteella simuloida veden- ja ilmanvastuksia. Lisätään ensiksi ilmanvastuksen simulointi, sillä se on suhteellisen yksinkertainen verrattuna vedenvastuksiin. Tässä saadaan apua aiemmin luodusta TriangleData-structista (funktio A.3), joka pitää sisällään erilaisia hyödyllisiä muuttujia kolmioihin liittyen. TriangleData-luokka sisältää muun muassa globaaleina muuttujina kolmion liikkeen nopeuden, liikkeen suunnan ja suunnan kulman verrattuna kolmion normaalivektoriin. Tämä kulma on positiivinen, jos liikkeen suunta on normaalivektorin suuntainen ja negatiivinen, jos liike on normaalivektoria vastaan. Näiden muuttujien laskua varten täytyy konstruktoriin ottaa mukaan kelluvan kappaleen Rigidbody-komponentti.

Voimien lisääminen kappaleeseen voidaan sijoittaa samaan FixedUpdate-funktioon jossa luvussa 5.3 lisätään nostevoimakin. Funktion sisällä tarkistetaan onko vedenpäällisiä kolmioita ollenkaan, sillä jos kappale on kokonaan veden alla, ilmanvastusta ei tarvitse ottaa huomioon. Muussa tapauksessa käydään vedenpäälliset kolmiot silmukassa läpi ja lasketaan ilman aiheuttama vastus kappaleelle jokaisen kolmion kohdalla.

Funktiossa 5.2 ‘rho’ on ilman tiheys ja ‘C_air’ on ilman vastuskerroin, jotka kerrotaan kolmion pinta-alalla ja nopeudella. Saatu vektori puolitetaan, jolloin saadaan ilmanvastuksen

voimavektori yhden kolmioverkon kolmion kohdalla. Vektori on kuitenkin vielä kolmion liikkeen suuntaan, jolloin se täytyy rivillä 7 kääntää päinvastaiseksi. Nyt voimavektori voidaan lisätä kappaleeseen oikeaan pisteeseen Unityn AddForceAtPosition-funktiolla[2]. Huomioitavaa on, että funktion rivillä 1 varmistetaan, että kolmio on kappaleen liikesuuntaa kohti, jotta ilmanvastusta ei laskettaisi kappaleen takana oleviin kolmioihin.

Funktio 5.2: void AirResistanceForce()

```
1 if (triangleData.cosTheta < 0f)
2 {
3     return Vector3.zero;
4 }
5
6 Vector3 airResistanceForce = 0.5f * rho * triangleData.velocity *
      triangleData.area * C_air;
7 airResistanceForce *= -1f;
8 return airResistanceForce;
```

Ilmanvastus on kuitenkin vähäisin kappaleeseen vaikuttavista voimista, joten siirrytään seuraavaksi veden aiheuttamiin vastuksiin. Kerner 2016 jakaa veden aiheuttaman vastuksen kolmeen eri osaan, jotka ovat viskoosi vedenvastus (eng. Viscous Water Resistance), painevastus (eng. Pressure Drag Forces) ja iskuvoima (eng. Slamming Forces). Käsitellään nämä kolme voimaa ilmanvastuksen tavoin erillisinä funktioina, jotka laskevat vastuksen voimavektorin yhdelle vedenalaiselle kolmiolle, joka voidaan antaa AddForceAtPosition-funktiolle arvoksi.

Viskositeetti on kaasun tai nesteen kyky vastustaa virtaamista, joka ilmenee niin kaasun kuin nesteenkin liikkuessa itsekseen kuin myös silloin, kun jokin kappale kulkee sen läpi. Kerner kuvaa ilmiötä siten, että kun laiva liikkuu veden läpi, hyvin pieni kerros vettä tarttuu laivan köliin kiinni veden viskositeetin takia ja laiva joutuu ikään kuin vetämään pientä määrää vettä mukanaan. Viskositeetin ollessa suurempi, isompi osa nestettä tarttuu mukaan, jolloin liikettä vastustava voima kasvaa. Kerner esittää vedenalaisen kolmion viskoosiksi kitkavoimaksi kaavaa:

2. `https://docs.unity3d.com/ScriptReference/Rigidbody.AddForceAtPosition.html`

$$\overrightarrow{F_{vi}} = \frac{1}{2}\rho C_f(R_n) S_i v_{fi} \overrightarrow{v_{vi}}, \tag{5.8}$$

jossa ρ on nesteen tiheys, S_i on kolmion pinta-ala, v_{fi} on nopeuden suuruus, $\overrightarrow{v_{vi}}$ on kolmion nopeuden tangentiaalinen suunta ja $C_f(R_n)$ on vastuskerroin ja R_n Reynoldsin luku. Pinta-ala ja nopeuden suuruus saadaan helposti TriangleData-luokasta, mutta koska viskoosikitkavoima on nesteen virtaamisen suuntainen, eli liikkuvan kolmion tason tangentin suuntaan. Tämä vektori saadaan projisoimalla liikkeen suuntavektori kolmion tasoon nähden, joka saadaan kaavalla[3]

$$B \times (A \times B), \tag{5.9}$$

jossa B on tason normaalivektori ja A on nopeuden suuntavektori. Kaavassa 5.8 esiintyvä vastuskerroin $C_f(R_n)$ taas saadaan laskettua kaavalla

$$C_f(R_n) = \frac{0.075}{(log_{10}R_n - 2)^2}, \tag{5.10}$$

joka on kansainvälisen Towing Tank -konferenssin (eng. International Towing Tank Conference) sopima kaava, joka on johdettu useammista kokeista ja datasta liittyen vedenalaiseen viskositeetti kitkaan. Kaava ei ole tarkka analyyttinen funktio, vaan regressioanalyysiin pohjautuva funktio, joka on tarpeeksi lähellä vastaamaan käytännön tarkoituksia (Kerner 2016) (Resistance Committee of the 28th ITTC 2017). Kaavassa 5.10 $R_n = VL/\nu$ on Reynoldsin luku, jossa V on kappaleen nopeus, L on pituus jonka neste matkaa kappaletta pitkin ja ν on nesteen viskositeetti. Koska vastuskerroin on kappalekohtainen, eli se on jokaiselle kolmiolle sama, voidaan se laskea ennen kuin kolmioverkkoa aletaan käymään läpi, jolloin laskenta-aikaa säästyy. Myös muuttuja L täytyy määritellä, sillä se on kappaleen pituus kun se kulkee veden halki. Esimerkiksi jos kappale on vene, luku on kölin pituus. Asetetaan se tässä vaiheessa olemaan kappaleen z-akselin suuntainen pituus, sillä jos oletetaan kappaleen olevan

3. `http://www.euclideanspace.com/maths/geometry/elements/plane/lineOnPlane/`

esimerkiksi laiva jota liikutetaan eteenpäin Unityn transform.forward-muuttujan suuntaisesti, olisi se kappaleen z-akselin suuntainen[4].

Kaavojen 5.8, 5.9 ja 5.10 mukaan voidaan tehdä funktio 5.3 viskoosille vedenvastukselle, jonka palauttama vektorin voidaan lisätä kappaleeseen funktiossa 5.7.

Funktio 5.3: Vector3 ViscousResistanceForce()

```
1  Vector3 B = triangleData.normal;
2  Vector3 A = triangleData.velocity;
3
4  Vector3 velocityTangent = Vector3.Cross(B, (Vector3.Cross(A, B) / B.
       magnitude)) / B.magnitude;
5  Vector3 tangentialDirection = velocityTangent.normalized * -1f;
6
7  Vector3 v_f_vec = triangleData.velocity.magnitude *
       tangentialDirection;
8
9  Vector3 viscousWaterResistanceForce = 0.5f * rho * v_f_vec.magnitude *
        v_f_vec * triangleData.area * coefficientFriction;
10 return viscousWaterResistanceForce;
```

Seuraava lisättävä voima on painevastus (eng. Pressure Drag Force), joka simuloi voimia jotka tulevat esiin kappaleen noustessa pintaa päin ja ajelehtiessa sivuttain. Samalla liikettä halutaan vaimentaa kappaleen liikkuessa hitaasti. Kerner 2016 selittää, että käsitellessä veden hydrodynamiikkaa kappaleeseen kohdistuu huomattava määrä erinnäisiä voimia, mutta kaikkien mahdollisten voimien huomioonottaminen on liian monimutkaista peliä varten tehtävään mallinnukseen. Tästä syystä painevastus on voima, joka ei suoraan ole verrattavissa mihinkään yksittäiseen hydrodynamiikan vastusvoimaan, vaan approksimaatio useammasta voimasta jotka vaikuttaisivat kappaleeseen vedessä.

Siinä missä viskoosi vedenvastus 5.8 oli voimavektori kolmioverkon kolmioiden tangentin suuntaan, painevastuksen vektori on kolmion normaalin suuntainen. Vastuksen laskemisessa tulee ottaa huomioon mihin suuntaan kolmio on suhteessa liikkeeseen. Jos liike on kolmion

4. https://docs.unity3d.com/ScriptReference/Transform-forward.html

normaalin suuntaan, voima on painevastus (eng. Pressure Drag), mutta jos liikkeen suunta on kolmion normaalia vastaan, kolmio on liikkuvan kappaleen perässä ja voimaa kutsutaan imuvastukseksi (eng. Suction Drag).

TriangleDatalla on muuttuja kolmion liikkeen ja normaalin suunnan kulmaksi, ja sen ollessa positiivinen kolmio on liikkeen suunnan mukainen ja voiman kaava on

$$\overrightarrow{F_{Di}} = -\left(C_{PD1}\frac{v_i}{v_r} + C_{PD2}\left(\frac{v_i}{v_r}\right)^2 \right) S_i(\cos\theta_i)^{f_p}\overrightarrow{n_i}. \tag{5.11}$$

Kulman ollessa negatiivinen, voima on imuvastus ja kaava voidaan esittää muodossa

$$\overrightarrow{F_{Di}} = \left(C_{SD1}\frac{v_i}{v_r} + C_{SD2}\left(\frac{v_i}{v_r}\right)^2 \right) S_i(\cos\theta_i)^{f_s}\overrightarrow{n_i}. \tag{5.12}$$

Molemmissa kaavoissa v_i on kolmion nopeus, v_r on referenssinopeus, S_i on kolmion pinta-ala ja $\overrightarrow{n_i}$ on kolmion normaalivektori. f_p ja f_s ovat paineen ja imun voimien kerroin suhteessa liikkeen suunnan ja kolmion normaalin kulmaan, joka määrää vastuksen suuruuden. Jos kolmio on kohtisuorassa liikkeen suuntaan, vastus on suurimmillaan ja kolmion ollessa liikkeen mukainen vastus häviää. C_{PD1} ja C_{SD1} ovat paineen ja imun lineaariset vastuskertoimet, C_{PD2} ja C_{SD2} ovat taas paineen ja imun neliölliset vastuskertoimet.

Nämä vastuskertoimet, kuten myös paineen ja imun kertoimet f_p ja f_s, täytyy selvittää eri kokoisille ja muotoisille kappaleille erikseen, jotta saa parhaimmat tulokset. Esimerkiksi vastuskertoimien arvot ollessa 300 ja paineen ja imun kerrointen ollessa 0,3 Unityn pelikentälle sijoitetut testikappaleet käyttäytyivät hyvin todentuntuisesti. Vastuskerroin vaikuttaa siihen, kuinka vahva painevastus on, jolloin pienemmät arvot vastustavat liikettä vähemmän ja suuremmat arvot vastustavat voimakkaammin. Kaavan kääntäminen Unityn funktioksi 5.4 on yksinkertainen, sillä kaikki loput arvot tulevat kolmioiden TriangleData-luokasta suoraan, jolloin voimavektori saadaan helposti laskettua ja lisättyä kappaleeseen funktiossa 5.7, kuten viskoosi vedenvastuskin.

Funktio 5.4: Vector3 PressureDragForce()

```
1 float velocity = triangleData.velocity.magnitude / referenceVelocity;
2 Vector3 pressureDragForce = Vector3.Zero;
3 if (triangleData.cosTheta > 0f)
4 {
5     pressureDragForce = -(C_PD1 * velocity + C_PD2 * (velocity *
        velocity)) * triangleData.area * Mathf.Pow(triangleData.
        cosTheta, f_P) * triangleData.normal;
6 }
7 else
8 {
9     pressureDragForce = (C_SD1 * velocity + C_SD2 * (velocity *
        velocity)) * triangleData.area * Mathf.Pow(Mathf.Abs(
        triangleData.cosTheta), f_S) * triangleData.normal;
10 }
11 return pressureDragForce;
```

Viimeinen liikettä vastustava voima on iskuvoima (eng. Slamming Forces), joka pyrkii simuloimaan veden jäykkyyttä kappaleen osuessa ilmasta veden pintaan. Edelliset voimat (viskoosi vedenvastus ja painevastus) hidastivat kappaleen liikettä, kääntymistä ja pyörimistä kappaleen ollessa jo vedessä, mutta ne eivät vaikuta paljoa siihen, jos kappale pudotetaan ilmasta veteen jolloin veden kovuus vaikuttaa vahvasti kappaleen liikkeeseen. Kerner (2016) selittää, että tämän ongelman laskemiseen on paljon erilaisia tapoja, mutta ne ovat hyvin monimutkaisia suhteessa siihen mitä videopelien toteutus tarvitsisi. Yksinkertaistettuna voima kuitenkin tarkoittaa sitä, että mitä suuremmalla voimalla ja nopeudella kappale osuu veden pintaan, sitä voimakkaampi on iskuvoima tätä liikettä vastaan, kun taas hitaasti veteen laskeutuva kappale ei koe iskuvoimaa lainkaan. Kappaleen uppoamisnopeus vedenpinnassa saadaan selville vertailemalla upoksissa olevan kolmioverkon pinta-alan muutosta jokaisen kuvapiirron välillä. Pinta-ala vaikuttaa myös iskuvoiman suuruuteen, sillä suurempi pinta-ala aiheuttaa suuremman voiman. Iskuvoiman on kuitenkin oltava kohdistettu vain niihin kolmioihin joihin kohdistuu isku vedenpinnalla, joten suoraan kappaleen alkuperäistä pinta-alaa ja vedenalaisen osan pinta-alaa vertaamalla ei saada realistisia tuloksia ja näin ollen toteutus täytyy tehdä yksittäisille kolmioverkon kolmioille. Kuitenkaan jokaisen vedenalai-

sen kolmion läpikäyminen ei toimi suoraan, sillä vedenalaisten kolmioiden määrä vaihtelee sitä mukaa kun kappale liikkuu vedessä ja uusi vedenalainen kolmioverkko luodaan jokaisen uuden kuvapiirron aikana, joten ne eivät voi säilyttää itsessään tietoa pinta-alan muutoksesta. Ratkaisuksi Kerner (2016) käyttää alkuperäisen kolmioverkon kolmioita ja niiden pinta-alan muutosta. Tämän toteuttamiseksi jokaisessa vedenalaisen kolmion pilkkomisalgoritmissa (4.2 ja 4.3), joissa jokainen luotu kolmio tietää mistä alkuperäisestä kolmiosta on peräisin ja summaa pinta-alansa sitä varten luotuun listaan, joka pitää yllä alkuperäisten kolmioiden tämän hetkisiä ja edellisen kuvapiirron aikaisia pinta-aloja. Yksi ehdoton vaatimus kuitenkin iskuvoimassa on että voima ei saa olla suurempi kuin kappaleen sen hetkisen liikkeen voima, jolloin voiman täytyy olla voimakkaimmillaankin vain yhtä suuri kuin nopeuden voima, jolloin kappaleen liike vain pysähtyy. Jos iskuvoima olisi jossain tilanteessa suurempi kuin liikkeen voima, kappale vaihtaisi suuntaa ja pomppaisi takaisin vedenpinnan päälle, mikä ei vastaa todellisuutta.

Jotta pinta-alan muutos saadaan selville Kerner (2016) esittää pinta-alan vaihdon nopeus per sekunti kaavaksi edellisen kuvapiirron aikana

$$\mathrm{d}V_j^{vaihdos-pre} = A_j^{vedenala-pre} \overrightarrow{v}_i^{pre} \tag{5.13}$$

ja tämän hetkisen kuvapiirron aikana

$$\mathrm{d}V_j^{vaihdos} = A_j^{vedenala} \overrightarrow{v}_i. \tag{5.14}$$

$A_j^{vedenala-pre}$ ja $A_j^{vedenala}$ ovat käsiteltävänä olevan kolmion pinta-ala edellisellä ja nykyisellä kuvapiirrolla ja $\overrightarrow{v}_i^{pre}$ ja $\overrightarrow{v}_i$ ovat kolmion nopeus edellisen ja nykyisen kuvapiirron aikana.

Jakamalla näiden kahden nopeuden erotus alkuperäisen kolmion pinta-alalla S_j ja edellisestä kuvapiirrosta kuluneella ajalla dt, saadaan tulokseksi vaihdoksen kiihtyvyyttä vastaava arvo

$$\overrightarrow{\Gamma}_j = \frac{\mathrm{d}V_j^{vaihdos} - \mathrm{d}V_j^{vaihdos-pre}}{S_j \mathrm{d}t}. \tag{5.15}$$

Tässä pisteessä Kerner (2016) huomauttaa kaksi eri skenaariota. Ensimäisessä kappale on se-

kä edellisessä että tämän hetkisessä kuvapiirrossa veden alla, jolloin dV kaavojen pinta-alat voidaan korvata kolmion alkuperäisellä pinta-alalla S_j. Tässä tilanteessa kaava 5.15 muuttuisi muotoon

$$\overrightarrow{\Gamma}_j = \frac{S_j \overrightarrow{v}_i - S_j \overrightarrow{v}_i^{pre}}{S_j \mathrm{d}t} = \frac{\overrightarrow{v}_i - \overrightarrow{v}_i^{pre}}{\mathrm{d}t}, \tag{5.16}$$

joka kuvaisi vain kolmion keskipisteen muutoksen kiihtyvyyttä veden alla. Toisessa skenaariossa oletetaan, että kolmio olisi edellisellä kuvapiirrolla kokonaisuudessaan veden päällä ja tämän hetkisessä kuvapiirrossa kokonaan veden alla. Kaava 5.15 olisi silloin

$$\overrightarrow{\Gamma}_j = \frac{S_j \overrightarrow{v}_i - \overrightarrow{0}}{S_j \mathrm{d}t} = \frac{\overrightarrow{v}_i}{\mathrm{d}t}, \tag{5.17}$$

mikä vastaa erittäin suurta kiihtyvyyttä, jossa kolmio kiihtyisi nykyiseen nopeuteensa vain yhden kuvapiirron välissä. Tämän voidaan olettaa olevan arvo, jolloin iskuvoiman tulisi olla suurimmillaan kolmiota kohden. Tätä arvoa voidaan siten käyttää apuna kun iskuvoiman suuruutta kolmiolle lasketaan.

Iskuvoiman toteuttamiseksi Kerner (2016) on kehittänyt kaavan

$$\overrightarrow{F}_j^{isku} = \mathrm{clamp}\left(\frac{\Gamma_j}{\Gamma_{max}}, 0, 1\right)^p \cos(\theta_j) \overrightarrow{F}_j^{stop}, \tag{5.18}$$

jonka avulla määritellään voiman suuruus riippuen uppoaman kiihtyvyydestä Γ_j, kulmasta jolla kolmio liikkuu suhteessa veden pintaan $\cos(\theta_j)$ ja osuvan kolmion pinta-alasta, joka on osa arvoa $\overrightarrow{F}_j^{stop}$. Γ_{max} on arvo kaavasta 5.17, jonka saavutettua kolmion pysäyttämiseksi vaadittu voima $\overrightarrow{F}_j^{stop}$ käytetään kokonaisuudessaan. Tämän takia Γ_j jaetaan arvolla Γ_{max}, sillä siitä saadulla suhdeluvulla voidaan iskuvoiman suuruus määrätä sopivaksi. Esimerkiksi maksimiarvosta puolet hitaammin veteen osuvan kolmion suhdeluku olisi $0,5$, jolloin iskuvoiman suuruus olisi puolet suurimmasta mahdollisesta. Jos kolmio uppoaa veteen yhtä suurella tai suuremmalla kiihtyvyydellä kuin Γ_{max}, tulee iskuvoiman olla suurimmillaan, joten suhdeluvun arvon tulisi olla 1. Tästä syystä laskettu suhdeluku on syytä sitoa arvojen 0 ja 1 välille. p auttaa tekemään nopeuden suhteen voiman suuruuteen eksponentiaaliseksi lineaa-

risen sijaan. Kun saatu kiihtyvyys kahden kuvapiirron välille on laskettu, voidaan arvo jakaa kaavan 5.17 tuloksella. Niiden välinen suhde kuvaa kuinka suuri iskuvoima kolmioon tulisi kohdistaa.

$\overrightarrow{F}_j^{stop}$ on siis tarvittava voima pysäyttämään liike kokonaisuudessaan ja Kerner (2016) käyttää sen laskuun kaavaa

$$\overrightarrow{F}_j^{stop} = mv_j \frac{2A_j^{vedenala}}{S^{kappale}}, \tag{5.19}$$

jossa tulee esille kolmion pinta-ala, jolloin voima suhteutetaan siihen. m on koko kappaleen massa ja v_j kolmion nopeus. Jos ajatellaan, että kappale tippuu suoraan veteen ilman pyörimistä, on kolmion nopeus v_j yhtä suuri kuin koko kappaleen nopeus v. Laskemalla tulo mv saadaan siis kappaleen kokonaisliikevoima, joka pysäyttää kappaleen liikkeen jos sen kohdistaa kappaleen liikkeen suuntaa vastaan. Tästä syystä tämä kokonaisvoima täytyy suhteuttaa vedenalaisen kolmion pinta-alan ja kappaleen kokonaispinta-alan suhteeseen, jolloin edes aivan äärimmäisissä tapauksissa voimien summasta ei tule suurempaa kuin mv tulosta, jolloin kappaleeseen kohdistuisi suurempi voima kuin alkuperäinen liikevoima ja kappaleen liike vaihtaisi suuntaa. Vedenalaisen kolmion pinta-ala kerrotaan kahdella, sillä veden pintaan osuu korkeintaan puolet kappaleen kokonaispinta-alasta, josta aiheutuu iskuvoimaa. Kuvitellaan esimerkiksi kolmion muotoinen litteä levy, jolla ei ole korkeutta ollenkaan. Olkoon kappaleen kokonaispinta-ala 1. Kappaleen osuessa kohtisuoraan veteen vain kappaleen vettä kohti osuvaan alaosaan kohdistuu iskuvoima, joka on puolet kappaleen kokonaispinta-alasta. Kun pinta-ala kerrotaan kahdella, saadaan kokonaisvoimaksi siten kappaleen kokonaisliikevoima kaavan 5.19 mukaan $\overrightarrow{F}_j^{stop} = mv \cdot (2 \cdot 0,5/1) = mv$.

Koodiin iskuvoiman toteutus vaatii muutamaa ylimääräistä listaa ja apuluokkaa. Ensin apuluokka 'SlammingForceData' (liite A.4), joka sisältää tiedon kolmion alkuperäisestä pinta-alasta, vedenalaisesta pinta-alasta, kolmion nopeudesta nykyisellä ja edellisellä kuvapiirrolla, sekä keskipisteestä nopeuden laskemiseksi. Ensimmäinen lista pitää sisällään alkuperäisen kolmioverkon kolmioden kanssa vastaavia ja yksi yhteen meneviä SlammingForceData-apuluokkia. Toinen lista 'indexOfOriginalTriangle' pitää sisällään kokonaislukuja, jotka ovat listan 'SlammingForceData' indeksejä. Lista 'IndexOfOriginalTriangle' vastaa vedenalaisen

kolmioverkon kolmioita koollaan ja järjestyksellään, ja indeksit näin ollen ovat suora viite tietyn vedenalaisen kolmion ja alkuperäisen kolmioverkon kolmion välille. Lista 'IndexOfOriginalTriangle' on aina tyhjä jokaisen uuden vedenalaisen kolmioverkon luomisen alussa, jolloin se voidaan aina tyhjentää ja alustaa uudelleen 4.4 funktion alussa. Lista 'SlammingForceData' taas täytyy luoda vain kerran luokan 'UnderwaterMesh' konstruktorissa 4.3. Konstruktorissa voidaan käydä läpi alkuperäisen kolmioverkon kolmiot ja täydentää lista järjestyksessä SlammingForceData-olioilla.

Kun alkuperäistä kolmioverkkoa käydään läpi silmukassa, voidaan jokaisen kolmion aikana päivittää listan 'SlammingForceData' alkioita silmukan indeksin avulla, sillä molempien listojen indeksit vastaavat toisiaan. Ennen silmukkaa päivitetään listan 'SlammingForceData' kaikkien alkioiden edellisen kuvapiirron pinta-ala lyhyellä silmukalla funktion 5.5

Funktio 5.5: slammingForceData silmukka

```
1 for (int j = 0; j < slammingForceData.Count; j++)
2 {
3   slammingForceData[j].previousSubmergedArea = slammingForceData[j].
        submergedArea;
4 }
```

Tämän jälkeen voidaan jatkaa funktion 4.5 silmukkaan, jossa lasketaan ja pilkotaan kaikki vedenalaiset kolmiot ja siten voidaan päivittää tämänhetkisen kuvapiirron pinta-alat. Kaikkien kulmapisteiden ollessa vedenpinnan yläpuolella listan 'SlammingForceData' alkion vedenalaiseksi pinta-alaksi asetetaan 0, eikä listaan 'indexOfOriginalTriangle' tarvitse lisätä kolmion indeksiä, sillä sitä ei tulla käsittelemään. Kaikkien kulmapisteiden ollessa vedenalla, vedenalaiseksi pinta-alaksi asetetaan suoraan kolmion alkuperäinen pinta-ala, mutta tällä kertaa listaan 'indexOfOriginalTriangle' lisätään silmukan indeksi, joka vastaa siis myös listan 'SlammingForceData' kolmiota. Samalla tyylillä päivitetään listan 'SlammingForceData' vedenalainen pinta-ala ja silmukan indeksi lisätään listaan 'indexOfOriginalTriangle', kun vain yksi tai kaksi kulmapistettä on vedenpinnan alapuolella. Luvussa 4.3 kolmio pilkotaan vain yhdeksi vedenalaiseksi kolmioksi ja listojen päivitys on suoraviivainen mutta luvun 4.2 kolmioita pilkotaan kahdeksi erilliseksi vedenpinnan alaiseksi kolmioksi. Tässä tapauksessa vedenalainen pinta-ala on näiden kahden kolmion summa ja listaan 'indexOfO-

riginalTriangle’ on lisättävä sama indeksi kahteen kertaan, jotta listan pituus ja vedenalaisten kolmioiden määrä pysyy samana.

Kun kaikki tarvittavat muuttujat on saatu laskettua ja listat täytettyä, voidaan voimavektori laskea funktion 5.6 mukaan. Saatu voimavektori voidaan lisätä kappaleeseen funktiossa 5.7, samalla tavalla kuin viskoosi vedenvastus ja painevastus.

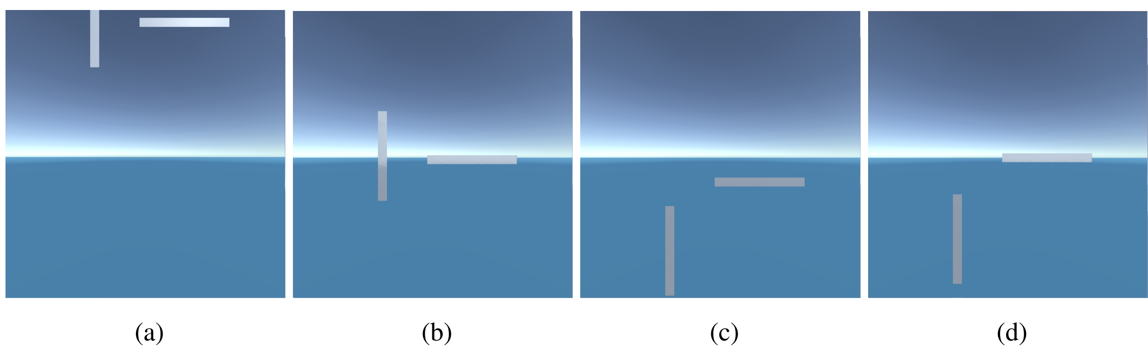
Funktio 5.6: Vector3 SlammingForce()

```
1 if (triangleData.cosTheta < 0f || slammingData.originalArea <= 0f)
2 {
3   return Vector3.zero;
4 }
5 Vector3 dV = slammingData.submergedArea * slammingData.velocity;
6 Vector3 dV_previous = slammingData.previousSubmergedArea *
     slammingData.previousVelocity;
7 Vector3 accVec = (dV - dV_previous) / (slammingData.originalArea *
     Time.fixedDeltaTime);
8
9 float acc = accVec.magnitude;
10 float acc_max = (slammingData.velocity / Time.fixedDeltaTime).
     magnitude / 2f;
11
12 Vector3 F_stop = boatMass * triangleData.velocity * ((2f *
     triangleData.area) / boatArea);
13
14 Vector3 slammingForce = Mathf.Pow(Mathf.Clamp01(acc / acc_max), slam_p
     ) * triangleData.cosTheta * F_stop * slammingMulti;
15
16 slammingForce *= -1f;
17 return slammingForce;
```

Kaikki edeltävät voimat, viskoosi vedenvastus, painevastus ja iskuvoima voidaan laskea perätysten joten niille voidaan laskea yhteinen voimavektori, joka voidaan laskutoimitusten jälkeen lisätä summana kappaleelle funktiossa 5.7 käyttämällä Unityn AddForceAtPosition-funktiota.

Funktio 5.7: void AddResistanceForces()

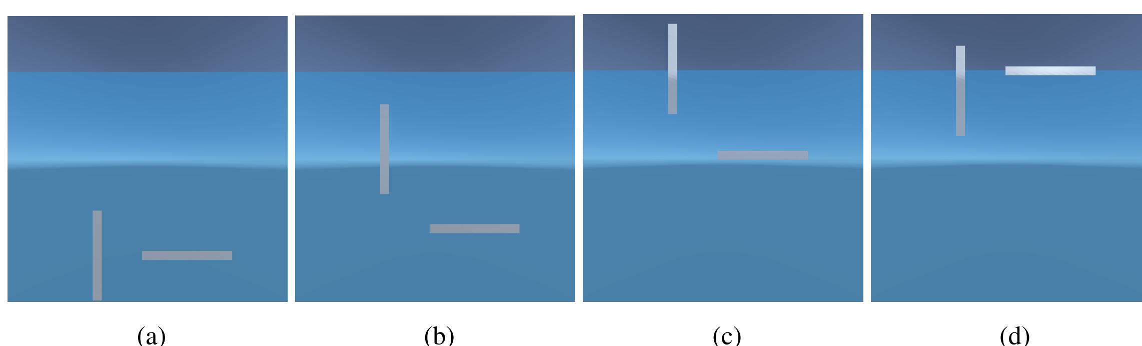
```
for (int i = 0; i < underWaterTriangleData.Count; i++)
{
    TriangleData triangleData = underWaterTriangleData[i];
    Vector3 forceToAdd = Vector3.zero;
    //Voima 2 - Viskoosi vedenvastus
    if (useViscousResistance)
        forceToAdd += ViscousWaterResistanceForce(rhoWater,
            triangleData, Cf_water);
    //Voima 3 - Painevastus
    if (usePressureDragResistance)
        forceToAdd += PressureDragForce(triangleData);
    //Voima 4 - Iskuvoima
    if (useSlammingResistance)
    {
        int originalTriangleIndex = indexOfOriginalTriangle[i];
        SlammingForceData slammingData = slammingForceData[
            originalTriangleIndex];
        forceToAdd += SlammingForce(slammingData, triangleData,
            boatArea, boatMass);
    }
    if (forceToAdd.magnitude > 0)
        thisRigidBody.AddForceAtPosition(forceToAdd, triangleData.
            center);
}
```



(a) (b) (c) (d)

Kuvio 20: Levymäiset litteät kappaleet pudotetaan veteen

Tuloksena saadaan vastus, joka ottaa huomioon kappaleen muodon ja asennon suhteessa siihen, miten kappale liikkuu vedessä ja veteen. Kuvasarjassa 20 pudotetaan veteen kolmen metrin korkeudesta kaksi identtistä leveän levyn muotoista kappaletta, joista oikeanpuoleinen on lappeellaan veden pintaa kohden, jolloin sillä on liikesuuntaa vasten suurempi pinta-ala kuin vasemmanpuoleisessa kappaleessa. Oikeanpuoleinen kappale odotetusti vastustaa liikettä paljon enemmän kuin vasemmanpuoleinen, joka putoaa veteen virtaviivaisemmin. Kuvista 21 havaitaan samanlainen ilmiö jos kappaleet ovat lähtötilanteessa jo veden alla ja ne päästetään kohoamaan pintaan, jossa virtaviivaisempi kappale kohoaa nopeammin, mutta lappeellaan oleva saa vastaansa paljon enemmän vastusta vedestä.



(a) (b) (c) (d)

Kuvio 21: Levymäiset litteät kappaleet kohoavat vedessä

5.5 Nosteen laskeminen tilavuuden keskipisteestä

Tässä alaluvussa vaihdetaan nosteen laskukaavaksi luvussa 5.1 esitetty nosteen kaava 5.1, eli lasketaan noste suoraan vedenalaisen osan keskipisteeseen. Tällöin voimavektorien laskeminen ja asettaminen kappaleeseen yksinkertaistuu, sillä voimavektoreita tulee olemaan aina yksi, huolimatta siitä kuinka monta kolmiota kappale sisältää.

Kolmiulotteisen kappaleen painopisteen löytämiseksi on olemassa useampi eri tapa, jotka tuottavat hiukan eri tuloksia tarkkuuteen nähden. Yksinkertaisin ratkaisu on Bourken (1988) kaava, joka ei tuota täydellisen tarkkaa tulosta vedenpinnan alaiselle vaillinaiselle monitahokkaalle, mutta virhe on niin pieni, että sen voi jättää huomiotta. Nürnberg (2013) esittää kaavan, joka tuottaa paljon tarkemman tuloksen ja keskipisteen laskeminen edellyttää kappaleen tilavuuden laskemista, jonka laskukaavan Nürnberg on myös esittänyt. Koska nos-

teen toteuttaminen keskipisteen kautta tarvitsee myös upoksissa olevan kappaleen tilavuuden, molemmat Nürnbergin kaavat ovat hyödyllisiä, mutta laskut ovat myös vaativampia.

Tarkastellaan ensin Bourken (1988) kaavaa. Kappaleen, jonka kolmioverkossa on N kolmiota jotka koostuvat (a_i, b_i, c_i) kulmapisteistä, painopiste C saadaan laskettua kaavalla

$$C = \frac{\sum_{i=0}^{N-1} A_i R_i}{\sum_{i=0}^{N-1} A_i}. \tag{5.20}$$

Bourken (1988) mukaan arvo $A_i = ||(b_i - a_i) \times (c_i - a_i)||$ on kolmion pinta-ala kaksinkertaisena ja $R_i = (a_i + b_i + c_i)/3$ on kolmion keskipiste. Nämä arvot ovat kuitenkin saatavilla TriangleData-luokasta suoraan.

Funktio 5.8: void AddUnderWaterForces()

```
1  Vector3 centroid = Vector3.zero;
2  Vector3 SumAiRi = Vector3.zero;
3  float SumAi = 0f;
4  float undewaterArea = 0f;
5  foreach (var triangle in underWaterTriangleData)
6  {
7      Vector3 Ri = triangle.center;
8      float Ai = triangle.area * 2f;
9      SumAi += Ai;
10     SumAiRi += Ri * Ai;
11
12     undewaterArea += triangle.area;
13 }
14 centroid = SumAiRi / SumAi;
15 float volume = (undewaterArea / origMeshArea) * origMeshVolume;
```

Funktiossa 5.8 muuttuja 'centroid' on kaavan 5.20 keskipiste. Tähän keskipisteeseen kohdistuu kaavan 5.1 noste, mutta nosteen laskemiseen tarvitaan vielä upoksissa olevan kappaleen osan tilavuus. Kuten aiemmin on mainittu, Nürnberg 2013 esittää kaavan tilavuuden laskuun, jota voi hyödyntää. Toinen hyvin yksinkertainen tapa tilavuuden laskuun on käyttää karkeaa arviota, joka perustuu kolmioverkon pinta-alan suhteeseen, jolloin nosteen lasku

pysyy edelleen yksinkertaisena ja nopeana. Funktiossa 5.8 olevan foreach-silmukan sisällä voidaan laskea yhteen kaikkien kolmioiden pinta-ala, jota voidaan verrata kappaleen alkuperäiseen kokonaispinta-alaan. Tätä suhdetta käyttäen kappaleen alkuperäiseen kokonaistilavuuteen saadaan arvio upoksissa olevan kappaleen pinta-alasta.

Tarkemman simuloinnin saa hyödyntämällä Nürnbergin (2013) toteutusta, joka hyödyntää keskipisteen laskemiseen kappaleen tilavuutta. Kaava tilavuuden laskuun on hyvin tarkka, vaikka kappale olisikin vain osaksi veden alla, eikä kolmioverkko olisi umpinainen. Nürnberg myös painottaa, että kappaleen ei tarvitse olla konveksi. Tilavuuden kaavaksi annetaan:

$$V = \frac{1}{3}\sum_{i=0}^{N-1}\int_{A_i} a_i \cdot n_i = \frac{1}{6}\sum_{i=0}^{N-1} a_i \cdot \dot{n}_i, \tag{5.21}$$

jossa $\dot{n}_i = (b_i - a_i) \times (c_i - a_i)$ on tason normaali ja (a_i, b_i, c_i) ovat kolmion kulmapisteet. Vaikka Nürnberg ei erikseen mainitse, on kaavassa huomioitava, että kulmapisteiden koordinaatit täytyvät olla maailman koordinaatiston origokeskeisiä, muutoin kaava antaa virheellisiä tuloksia. Koordinaatistomuunnos onnistuu vähentämällä jokaisen kulmapisteen koordinaatista koko isäntäkappaleen paikkakoordinaatti (olettaen, että kappaleen paikkakoordinaatti lasketaan sen keskipisteestä). Tällöin kolmion kulmapisteet ovat kuin kappale sijaitsisi pelimaailman origossa. Koska kaavan täytyy käydä läpi kaikki vedenalaiset kolmiot, voidaan kaava sijoittaa samaan tapaan kuin 5.8:

Funktio 5.9: void AddUnderWaterForces()

```
1  float volume = 0f;
2  foreach (var triangle in underWaterTriangleData)
3  {
4      Vector3 origoP1 = triangle.p1 - transform.position;
5      Vector3 origoP2 = triangle.p2 - transform.position;
6      Vector3 origoP3 = triangle.p3 - transform.position;
7      Vector3 ni = Vector3.Cross((origoP2 - origoP1), (origoP3 - origoP1
           ));
8      volume += Vector3.Dot(origoP1, ni);
9  }
10 volume *= (1f / 6f);
```

Kun tilavuus on saatu selville, keskipisteen laskemiseksi Nürnberg (2013) esittää kaavan

$$c \cdot e_d = \frac{1}{2V} \sum_{i=0}^{N-1} \int_{A_i} (x \cdot e_d)^2 (n_i \cdot e_d), d = 1,2,3. \tag{5.22}$$

Kaavan tulos on yhdelle koordinaatille kerrallaan, jolloin $e_1 = (1,0,0)$ (x-koordinaatisto), $e_1 = (0,1,0)$ (y-koordinaatisto) ja $e_1 = (0,0,1)$ (z-koordinaatisto). Jäljelle jää vain integraalin lasku:

$$\int_{A_i} (x \cdot e_d)^2 (n_i \cdot e_d) = \frac{1}{24} \dot{n}_i \cdot e_d \left([(a_i + b_i) \cdot e_d]^2 + [b_i + c_i) \cdot e_d]^2 + [(c_i + a_i) \cdot e_d]^2 \right), \tag{5.23}$$

joka voidaan sijoittaa 5.9 silmukan perään. Kaavaa kuitenkin on syytä optimoida. Esimerkiksi $\dot{n}_i \cdot e_d$ on kulloisenkin laskettavan koordinaatiston x, y tai z komponentti ja koska $\dot{n}_i$ on Unityn Vector3-olio, saa komponentit siitä suoraan [5]. Myös $(a_i + b_i) \cdot e_d$ on myös kahden kulmapisteen vektorien summavektorin x, y tai z komponentti joka saadaan suoraan vektorista. Lopuksi kertolasku $\frac{1}{24}$ voidaan ottaa silmukkasummaamisen sisältä pois ja laskea viimeisenä koko summaan, jolloin funktio 5.9 on kokonaisuutena:

Funktio 5.10: void AddUnderWaterForces()

```
1 float volume = 0f;
2 float centroidX = 0;
3 float centroidY = 0;
4 float centroidZ = 0;
5 foreach (var triangle in underWaterTriangleData)
6 {
7     Vector3 origoP1 = triangle.p1 - transform.position;
8     Vector3 origoP2 = triangle.p2 - transform.position;
9     Vector3 origoP3 = triangle.p3 - transform.position;
10    Vector3 ni = Vector3.Cross((origoP2 - origoP1), (origoP3 - origoP1
          ));
11    volume += Vector3.Dot(origoP1, ni);
12
```

5. https://docs.unity3d.com/ScriptReference/Vector3-x.html

```
    float p1p2e1 = (origoP1 + origoP2).x;
    float p2p3e1 = (origoP2 + origoP3).x;
    float p3p1e1 = (origoP3 + origoP1).x;

    float p1p2e2 = (origoP1 + origoP2).y;
    float p2p3e2 = (origoP2 + origoP3).y;
    float p3p1e2 = (origoP3 + origoP1).y;

    float p1p2e3 = (origoP1 + origoP2).z;
    float p2p3e3 = (origoP2 + origoP3).z;
    float p3p1e3 = (origoP3 + origoP1).z;

    centroidX += ni.x * (p1p2e1 * p1p2e1 + p2p3e1 * p2p3e1 + p3p1e1 *
        p3p1e1);
    centroidY += ni.y * (p1p2e2 * p1p2e2 + p2p3e2 * p2p3e2 + p3p1e2 *
        p3p1e2);
    centroidZ += ni.z * (p1p2e3 * p1p2e3 + p2p3e3 * p2p3e3 + p3p1e3 *
        p3p1e3);
}
volume *= (1f / 6f);
centroidX *= (1f / 24f) * (1.0f / (2f * volume));
centroidY *= (1f / 24f) * (1.0f / (2f * volume));
centroidZ *= (1f / 24f) * (1.0f / (2f * volume));

centroidX += transform.position.x;
centroidY += transform.position.y;
centroidZ += transform.position.z;
centroid = new Vector3(centroidX, centroidY, centroidZ);
```

Kaavan hyödyntämisessä on tärkeää huomata, että vaikka keskipisteen laskemiseen voi käyttää samaa origokeskeistä $\dot{n}_i$ normaalivektoria, täytyy lasketut koordinaatit lopulta siirtää takaisin oikealle paikalle pelimaailman koordinaatistossa. Muussa tapauksessa tulokseksi saatu keskipiste sijaitsee pelimaailman origon läheisyydessä, eikä siellä missä kappale todellisuudessa sijaitsee.

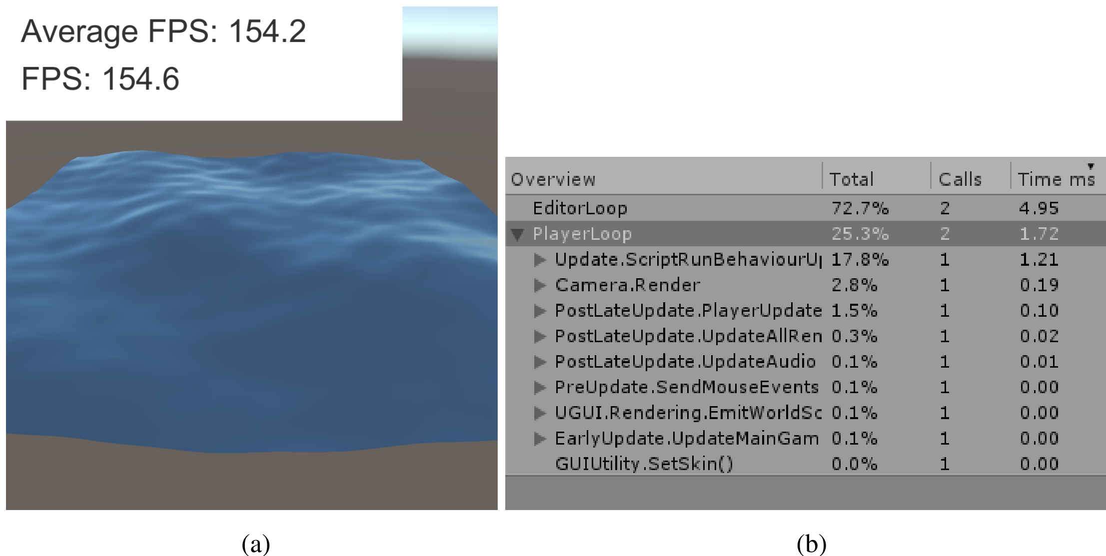
Näiden laskujen jälkeen kaava 5.1 voidaan suoraan sijoittaa skriptiin ja lisätä voimavekto-

ri kappaleelle funktion 5.11 tavoin. Valinnan mukaan muuttujat ‘volume’ ja ‘centroid’ voivat tulla joko Bourken (1988) sovelletusta funktiosta 5.8 tai Nürnbergin (2013) sovelletusta funktiosta 5.10.

Funktio 5.11: void AddUnderWaterForces()

```
1 Vector3 force = rhoWater * volume * -Physics.gravity.y * Vector3.up;
2 thisRigidBody.AddForceAtPosition(force, centroid);
```

6 Simulaatiotulokset



Overview	Total	Calls	Time ms
EditorLoop	72.7%	2	4.95
▼ PlayerLoop	25.3%	2	1.72
▶ Update.ScriptRunBehaviourU	17.8%	1	1.21
▶ Camera.Render	2.8%	1	0.19
▶ PostLateUpdate.PlayerUpdate	1.5%	1	0.10
▶ PostLateUpdate.UpdateAllRen	0.3%	1	0.02
▶ PostLateUpdate.UpdateAudio	0.1%	1	0.01
▶ PreUpdate.SendMouseEvents	0.1%	1	0.00
▶ UGUI.Rendering.EmitWorldSc	0.1%	1	0.00
▶ EarlyUpdate.UpdateMainGam	0.1%	1	0.00
GUIUtility.SetSkin()	0.0%	1	0.00

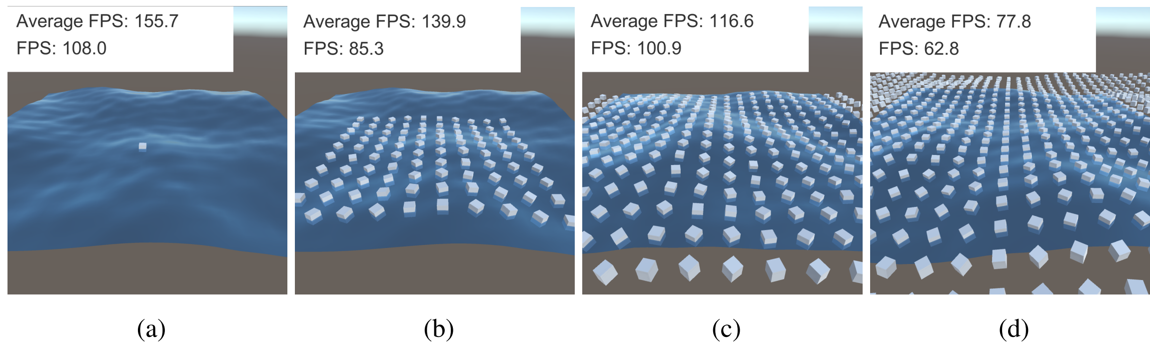
(a) (b)

Kuvio 22: Kontrollitilanne

Käsitellään nyt yhtenäisesti luvuissa 3 ja 5 esitettyjen mallinnuksien, sekä varsinkin luvun 5 erilaisten voimien vaikutusta laskentatehoon. Kuvioissa 22 nähdään kontrollitilanne, jossa on muutoin tyhjä pelikenttä, jonka taustalla toimivina skripteinä on ainoastaan aallokon laskeminen ja FPS-kuvataajuuden laskeminen ja näyttäminen ruudulla. Kontrollitilanteen kuvataajuus oli noin 170 FPS ja prosessorin laskentanopeus skripteille ja fysiikoille oli hieman alle kaksi millisekuntia. Huomioitavaa kuitenkin on, että laskentatehoa ja kuvataajuuden suuruutta alentaa huomattavasti jos peliä ajetaan Unityn editorin kautta, eikä käännettynä ja itsestään ajettavana '.exe' tiedostona, joten tulokset ovat suuntaa antavia.

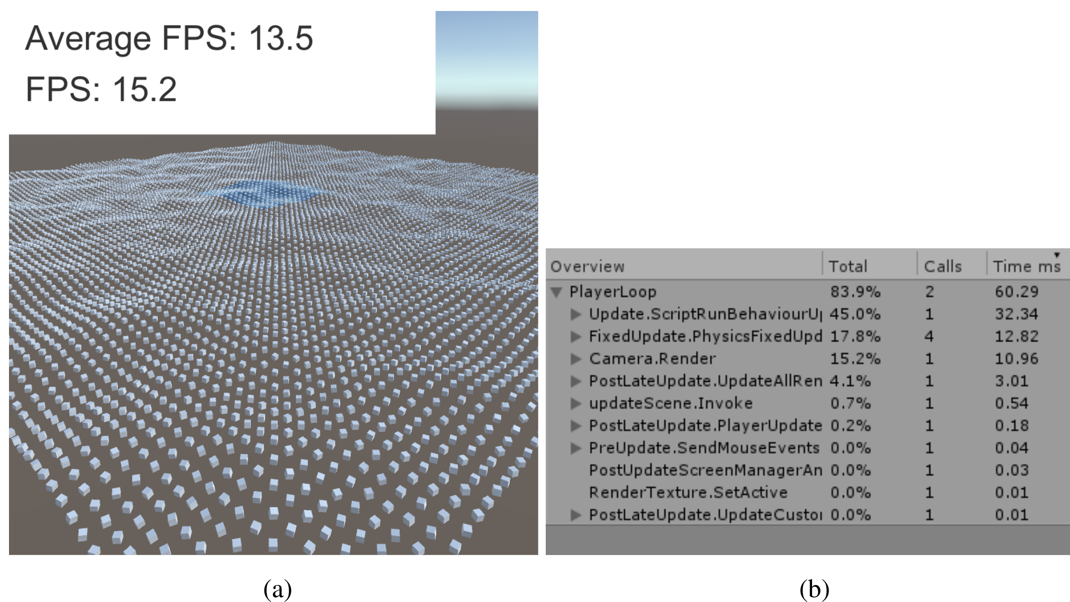
Luvun 3 mallinnus antoi kuvion 23 mukaiset tulokset, joissa kappaleiden määrät ovat 1 FPS, 100 FPS, 400 FPS, 1024 FPS ja keskiarvot kuvataajuudesta 155 FPS, 140 FPS, 116 FPS ja 77 FPS. Laskentateho pysyy hyvänä, vaikka kappaleita olisikin suuri määrä. Äärimmäisessä tapauksessa kuviossa 24 kappaleita on jo 10000 pelikentällä kuvataajuus oli 13 FPS.

Kuvasta 24b näkyy, että profiler näyttää fysiikkamoottorin FixedUpdate-kutsujen vievän lähes 20% koko suoritusajasta, vaikka kappaleilla ei ole Rigidbody-komponenttia. Kutsut syntyvät kappaleiden Collider-komponentista, jonka avulla fysiikkamoottori päättelee, osuvatko



(a) (b) (c) (d)

Kuvio 23: Yksinkertainen mallinnus useammilla kappaleilla



Overview	Total	Calls	Time ms
▼ PlayerLoop	83.9%	2	60.29
▶ Update.ScriptRunBehaviourU	45.0%	1	32.34
▶ FixedUpdate.PhysicsFixedUpd	17.8%	4	12.82
▶ Camera.Render	15.2%	1	10.96
▶ PostLateUpdate.UpdateAllRen	4.1%	1	3.01
▶ updateScene.Invoke	0.7%	1	0.54
▶ PostLateUpdate.PlayerUpdate	0.2%	1	0.18
▶ PreUpdate.SendMouseEvents	0.0%	1	0.04
PostUpdateScreenManagerAn	0.0%	1	0.03
RenderTexture.SetActive	0.0%	1	0.01
▶ PostLateUpdate.UpdateCusto	0.0%	1	0.01

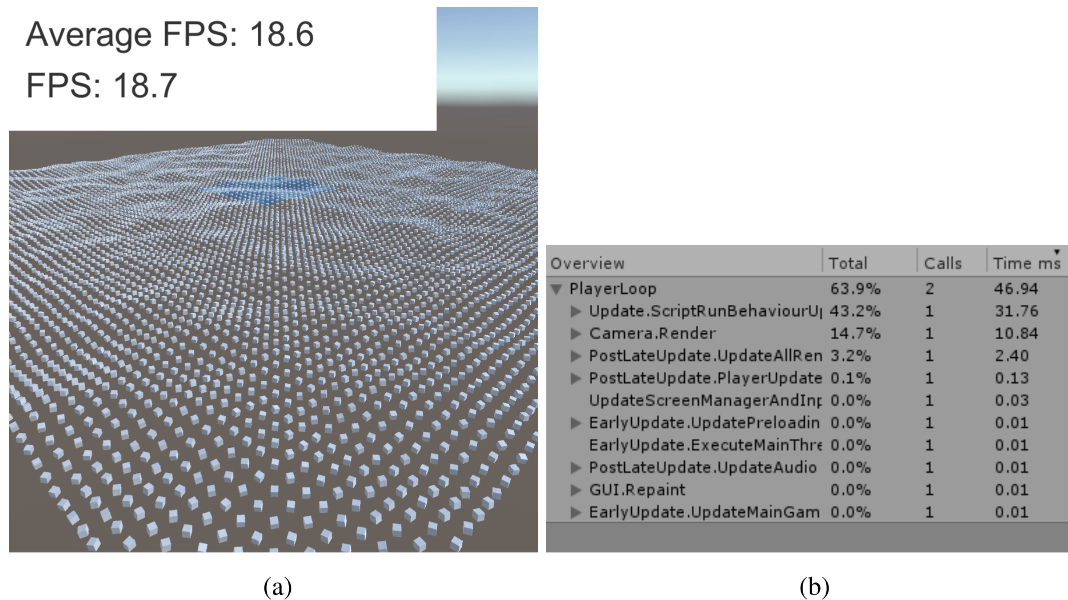
(a) (b)

Kuvio 24: 10000 kuutiota pelikentällä

kappaleet toisiinsa ja 10000 kappaleen simuloinnissa tämä alkaa jo vaikuttaa suuresti laskentatehoon, vaikka kappaleet eivät törmäilisikään toisiinsa.

Jos kappaleista poistetaan Collider-komponentti kokonaan, suorituskyky nousee entisestään ja kuviosta 25b nähdään, että laskentatehoon vaikuttaa eniten kelluvuuden skripti ja pelikentän piirtäminen näytölle.

Koska luvun 5 mallinnuksessa on useampia erillisiä laskettavia voimia ja kaksi eri tapaa nosteen laskemiseen, olisi jokaisen eri tilanteen esittäminen usean eri kuutiomäärän tapauksessa hyvin epäkäytännöllistä. Tästä syystä vertaillaan näiden laskujen vaikutusta prosessorin las-



(a) (b)

Kuvio 25: 10000 kuutiota pelikentällä ilman Collider-komponenttia

kentanopeuteen ja kuvataajuuteen kun pelikentällä on 100 kuutiota, jolloin vaikutus laskentatehoon on myös helpommin nähtävissä. Laskentatehon vertailussa jätetään huomiotta veden aaltoilu, sillä laskentatehoon vaikuttaa kappaleen vedenalaisten kolmioiden määrä, jolloin aallokossa keinuvan kuution vedenalaisten kolmioiden määrä voi vaihdella hyvinkin paljon. Tästä syystä tasaisella vedenpinnalla saadaan tarkemmat vertailuarvot eri voimien laskentateholle. Yhdellä kuutiolla kuvataajuus oli noin 150 FPS ja prosessorin laskentanopeus eri skriptien ja eri mallien tapauksessa oli hieman yli kahden millisekunnin.

Nostevoiman laskentatyylillä ei ollut suurta vaikutusta. Paineen avulla laskeminen oli hieman tehokkaampaa, se oli noin 1-2 kuvapiirtoa sekunnissa nopeampaa ja prosessorin laskunopeus vajaan millisekunnin nopeampi kuin tilavuuden keskipisteestä laskemalla. Suurimmat laskentatehon erot kuitenkin syntyvät liikettä vastustavien voimien laskemisessa.

Taulukosta nähdään kuinka kaikki vastukset laskevat kuvataajuutta lähes kymmenellä kuvapiirrolla sekunnissa ja jokainen yksittäinenkin voima laskee kuvataajuutta melkein viidellä kuvapiirrolla sekunnissa. Kuvista 27 nähdään myös, että prosessorin käyttämä aika skripteille ja fysiikoille nousi noin viidellä millisekunnilla.

Voimat	Painenoste	Keskipistenoste
Kaikki vastukset	35 FPS	34 FPS
Ainoastaan viskoosivastus 5.3	37 FPS	36 FPS
Ainostaan painevastus 5.4	38 FPS	36 FPS
Ainoastaan iskuvastus 5.6	39 FPS	37 FPS
Ei vastuksia ollenkaan	43 FPS	42 FPS

Kuvio 26: Painenosteen ja keskipistenosteen suorituskyvyn vertailu

Overview	Total	Calls	Time ms
PlayerLoop	73.6%	2	16.58
Update.ScriptRunBehaviourU	61.0%	1	13.74
FixedUpdate.ScriptRunBehavi	6.3%	1	1.44
Camera.Render	2.5%	1	0.56
FixedUpdate.PhysicsFixedUpd	1.2%	1	0.27
PostLateUpdate.UpdateAllRen	0.5%	1	0.13
PostLateUpdate.PlayerUpdate	0.4%	1	0.10
PostUpdateScreenManagerAn	0.1%	1	0.03
RenderTexture.SetActive	0.0%	1	0.01
PostLateUpdate.UpdateAudio	0.0%	1	0.01
PreUpdate.SendMouseEvents	0.0%	1	0.01

(a)

Overview	Total	Calls	Time ms
PlayerLoop	83.3%	2	25.96
Update.ScriptRunBehaviourU	45.0%	1	14.03
FixedUpdate.ScriptRunBehavi	32.6%	2	10.18
Camera.Render	2.1%	1	0.68
FixedUpdate.PhysicsFixedUpd	1.7%	2	0.54
PostLateUpdate.PlayerUpdate	0.3%	1	0.12
PostLateUpdate.UpdateAllRen	0.1%	1	0.05
updateScene.Invoke	0.1%	1	0.05
PostLateUpdate.UpdateAudio	0.0%	1	0.01
PreUpdate.SendMouseEvents	0.0%	1	0.01
UGUI.Rendering.EmitWorldSc	0.0%	1	0.01

(b)

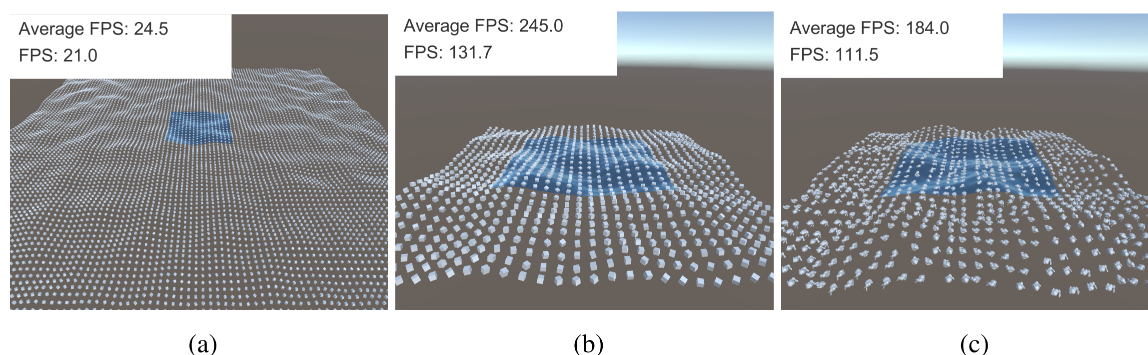
Kuvio 27: Prosessorin laskentanopeus a) ilman vastuksia b) vastuksilla

6.1 Johtopäätökset

Luvun 3 mallinnuksessa on ehdottomasti paras hyötysuhde laskentatehon ja kelluvuuden näyttävyyden välillä. Kappaleet näyttävät kelluvan hyvin realistisen oloisesti ilman kovinkaan suurta vaikutusta suorituskykyyn. Vaikka kappaleita ei saakaan vedestä nostettua pois, voi niihin listätä Unityn Rigidbody-komponentin, josta voi valita, että painovoima jätetään huomioimatta. Tällöin kappaleet voivat törmäillä toisiinsa ja liikkua vedessä törmäysten voimasta. Mallinnus ei myöskään tarvitse sen suurempia valmisteluja, ainoastaan skriptin liittämisen kappaleeseen. Jos kappaleen halutaan kelluvan eri syvyydellä tai eri kulmassa, joudutaan luvussa 3.2 esitetyt kaltevuustason kolmion kulmapisteet määrittelemään erikseen.

Kuten tämän luvun alussa mainittiin, Unityn editorista ajamalla pelin suorituskyky on huomattavasti alhaisempi kuin itsenäisenä ajettavan pelin. Kuvista 28 nähdään kuinka luvun 3 mallinnukset lasketaan huomattavasti nopeammin, esimerkiksi 1024 kuution pelikentän suo-

rituskyky on noussut kuvataajuudesta 74 FPS kuvataajuuteen 245 FPS. Huomioitavaa on myös se, että kappaleiden monimutkaisuus ei vaikuta kelluvuuden laskentaan. Kuva 28 c koostuu 1024 Stanford Bunny-mallista, jonka kolmioverkko sisältää 4968 kolmiota. Kuvataajuuden ero johtuu näytönohjaimelle aiheutuvasta työstä monimutkaisempien kolmioverkkojen takia, mutta prosessorin laskuaika skripteihin pysyi samana.

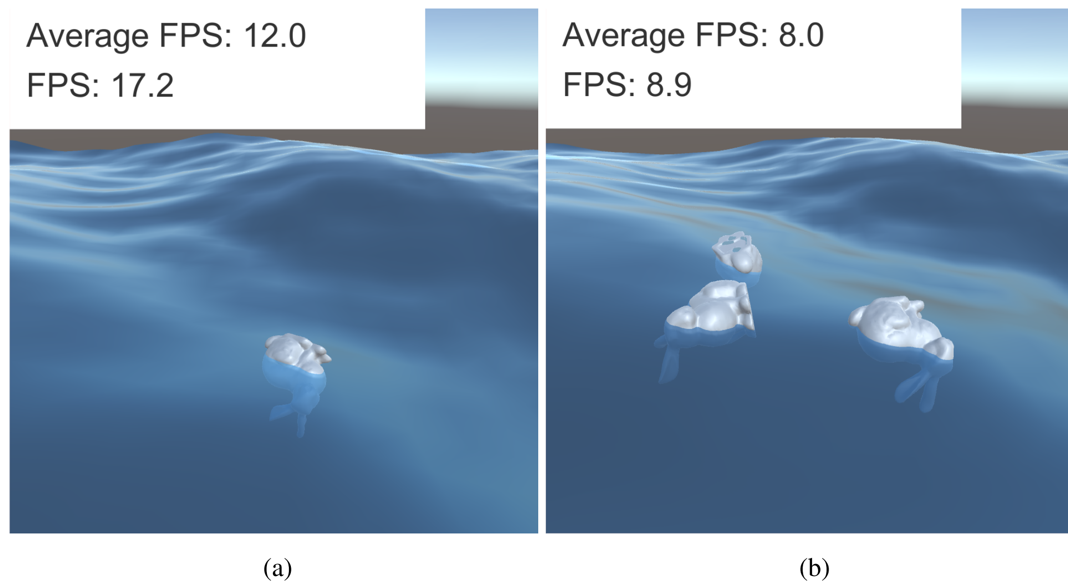


(a) (b) (c)

Kuvio 28: Käännetty peli: 10000 laatikkoa ja 1024 laatikkoa/jänistä

Odotetusti luvun 5 mallinnukset olivat vaativampia laskennallisesti ja kelluvuuden skripti vaatii enemmän alkuvalmisteluja, kuten lapsi-peliobjektin massakeskipisteelle, Rigidbody- ja Collider-komponentit sekä kunnollisen 3D-kolmioverkon. Mallinnuksen tuottama kelluvuus realistisemman ja dynaamisemman oloinen kuin luvun 3 mallinnus. Vaikka nosteen laskutyylien erot eivät ole suuret suorituskykyyn nähden, näkyvämmät erot tulevat kappaleiden liikkeissä. Painetta mallintava noste lisää useita voimavektoreita eri puolille kappaletta, mikä voi aiheuttaa kappaleelle ei-toivottua liikettä. Esimerkiksi täysin tyynellä pinnalla kelluva kappale saattaa alkaa pyörimään holtittomasti, vaikka siihen ei kohdistuisi muita voimia kuin veden aiheuttama noste. Luvun 5.4 vastukset kuitenkin estävät myös nämä ei-toivotut liikkeet. Liikettä vastustavat voimat ovat hyvin kattavat ja tarkat, varsinkin jos kappaleelle löydetään sopivat vastuskertoimet kaavoihin 5.11 ja 5.12. Tästä huolimatta laskutoimitukset ovat hyvin raskaita, varsinkin jos kappaleen kolmioverkko on monimutkaisempi. Kuviossa 29a on Stanford Bunny-malli kellumassa ja ainoastaan yksi tälläinen kappale laskee kuvataajuuden lähelle kymmentä jos sille lasketaan kaikki liikettä vastustavat voimat.

Pelien näkökulmasta tälläiset mallinnukset ovat mahdollisesti siihen tarkoitukseen liian tarkkoja ja vaativia, varsinkin kun jokaiselle kappaleelle täytyisi määritellä vastuskertoimet erik-



(a) (b)

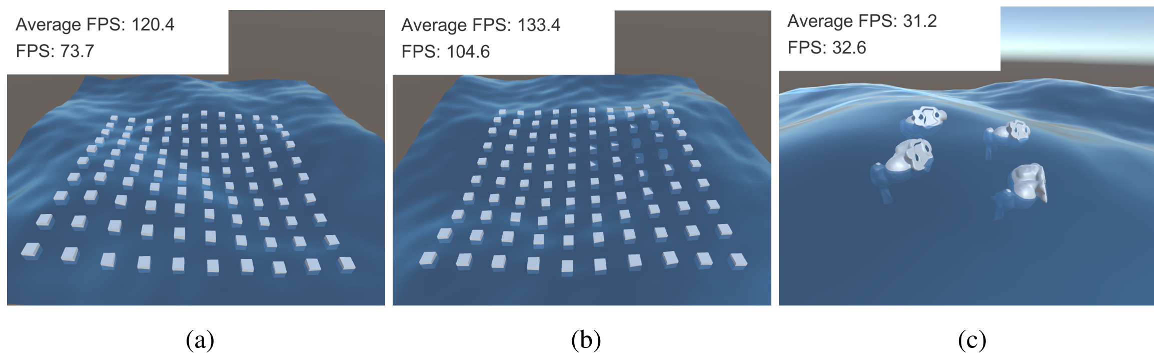
Kuvio 29: Stanfod Bunny-mallit erilaisten vastussimulointien vaikutuksessa

seen. Tästä syystä yksi mahdollinen vaihtoehto olisi käyttää Unityn Rigidbody-komponentissa jo löytyviä Drag[1]- ja Angular Drag[2]-ominaisuuksia, jotka ovat vastuskertoimet kappaleen etenevään liikkeeseen ja pyörimisliikkeeseen. Sen sijaan, että jokaista kolmioverkon kolmiota käsiteltäisiin erikseen, voidaan 'Drag' ja 'Angular Drag' asettaa koko kappaleelle. Tämä onnistuisi esimerkiksi antamalla suurimman ja pienimmän mahdollisen arvon molemmille vastuksille. Tällöin voidaan laskea sopiva vastus sen mukaan, kuinka iso osa kappaleen pinta-alasta on veden alla.

Esimerkiksi funktiossa A.5 saadaan suhteellisen lineaarisesti muuttuvat vastukset vedenalaisen ja alkuperäisen pinta-alan suhteen avulla. Tämä toteutus ei vaikuta laskentatehoon merkittävästi, sillä vedenalainen pinta-ala lasketaan jo kolmioita pilkottaessa ja laskutoimitus vastukseen on hyvin nopeasti laskettavissa. Tämän avulla saadaan kuvassa 29b näkyvät kolme Stanford Bunny-mallia kellumaan lähes samalla rasituksella kuin kuvan 29a yksi Stanford Bunny.

Kuten aiemmin kuvion 28 perusteella havaittiin, että ajettavaksi ohjelmaksi käännetyn pelin suorituskyky on parempi kuin Unity sisällä ajettava, pätee sama myös luvun 5 raskaampiin

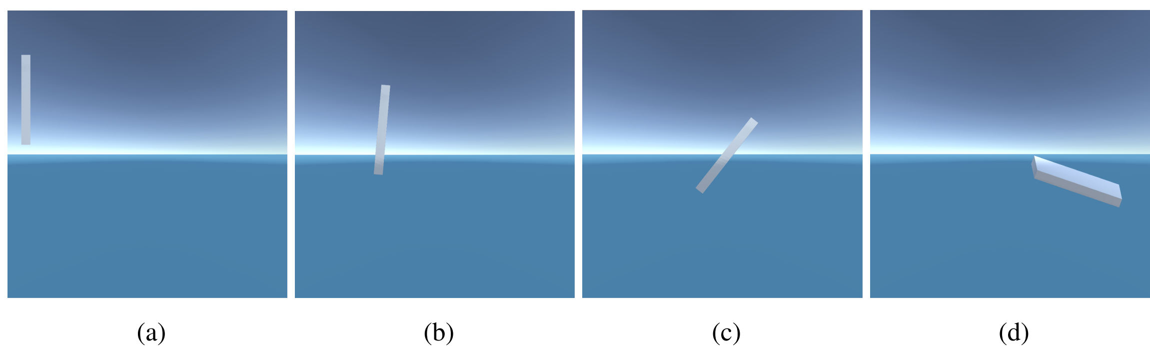
1. `https://docs.unity3d.com/ScriptReference/Rigidbody-drag.html`
2. `https://docs.unity3d.com/ScriptReference/Rigidbody-angularDrag.htmll`



(a) (b) (c)

Kuvio 30: Käännetty peli: 100 laatikkoa raskailla ja helpoilla vastuksilla, sekä 4 jänistä funktion A.5 vastuksilla

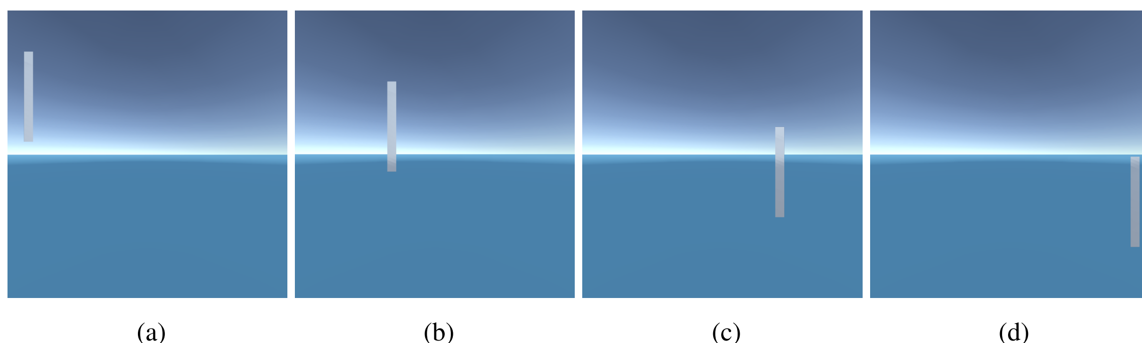
laskuihin. Kuvissa 30a ja 30b nähdään kuinka 100 kelluvaa laatikkoa pääsee hyvin yli 100 FPS:n kuvataajuuden ja neljä Stanford Bunny-malliakin suoriutuu huomattavasti paremmin verrattuna kuvan 29b tilanteeseen. Drag- ja Angular Drag-ominaisuuksilla toteutettu veden vastus ei kuitenkaan ole aivan realistinen, sillä vastus vaikuttaa koko kappaleeseen, siinä missä luvun 5.4 voimat vaikuttavat vain kappaleen vedenalaiseen osaan.



(a) (b) (c) (d)

Kuvio 31: Realistiset voimat kallistavat kappaletta sen mukaan mihin osaan voimat vaikuttavat

Kuvioissa 31 ja 32 levymäiseen litteään kappaleeseen kohdistuu liikevoimaa vasemmalta oikealle, jonka jälkeen se pudotetaan veteen. Kuviossa 31 kappale kallistuu realistisesti osuessaan veteen. Tämä johtuu siitä, että heti kappaleen osuessa veteen vedenalaiseen osaan kohdistetaan liikettä vastustavia voimia, jotka aiheuttavat kappaleelle vääntöä. Kuvasarjassa 32 puolestaan Unityn Drag-ominaisuus hidastaa kappaletta kokonaisuutena, jolloin vääntöä ei synny.



Kuvio 32: Yksinkertainen 'Drag' vastus hidastaa kappaletta kokonaisuutena

Mallinnukset ovat tälläisenään jo riittävän realistisia ja suhteellisen helppoja laskea suorituskyvyn kannalta, mutta monia osia toteutuksissa pystyisi optimoimaan. Laskutoimitusten säikeistäminen parantaisi suorituskykyä, jolloin vaikeat kolmioverkon laskutoimitukset eivät hidastaisi pelin pääsäiettä, vaan ne laskettaisiin taustalla erillisessä säikeessä. Eri toteutukset soveltuisivat hyvin eri tilanteisiin, esimerkiksi luvun 5 mallinnukset voivat olla liian yksityiskohtaisia, jos mallinnettavat kohteet olisivat paikallaan pysyviä, ankkuroituja laivoja satamassa, kun taas luvun 3 toteutus olisi liian yksinkertainen dynaamisille peliobjekteille, joita pitäisi pystyä poimimaan vedestä ja heittämään uudestaan takaisin. Luvun 5.4 voimat ovat hyvin tarkkoja ja voivat tuottaa erittäin realistisen oloisen veden, mutta laskujen vaativuus voi olla liian suuri hyötyihin pelien näkökulmasta. Yksinkertainen, nopeasti laskettava vastuksen simulointi, kuten A.5 on monesti parempi peleihin. Taulukon 26 mukaan iskuvastuksen simulointi on suhteellisen nopea verrattuna painevastukseen ja viskoosiin vedenvastukseen, joten yhdistämällä iskuvastuksen ja A.5 voisi saada kuviota 31 muistuttavan mallinnuksen vähäisemmällä vaikutuksella suorituskykyyn.

7 Yhteenveto

Vaikka kelluvuuden realistinen mallintaminen on haastavaa ja laskentatehoa vaativaa, videopeleille on mahdollista saada aikaan toimivia toteutuksia, jotka näyttävät realistisilta ja toimivat reaaliajassa. Esitetyt menetelmät sopivat myös moniin erilaisiin käyttökohteisiin, riippuen millaista mallinnusta pelissä haetaan. Esimerkiksi vaikka yksinkertaisempi kelluvuuden toteutus ei käyttäisikään oikeaa fysikaalista toteutusta, se voi suorituskykynsä ja realistisen käyttäytymisensä takia soveltua hyvin myös peleihin jotka esimerkiksi keskittyvät kokonaan veden pinnan päällä liikkumiseen.

Näiden tulosten valossa olisi seuraavaksi syytä tutkia kelluvuuden mallintamisen optimointia, esimerkiksi kaavojen laskemisen säikeistämistä tai jakamista erilliselle näytönohjaimelle, joka suoriutuisi yksinkertaisimmista laskuista huomattavasti prosessoria nopeammin.

Lähteet

Arheimer, Berit. 2016. “The active liquid Earth–importance of temporal and spatial variability”.

Bourke, Paul. 1988. “Calculating the area and centroid of a polygon”. *Swinburne Univ. of Technology* 7.

Capcom. 1988. *Mega Man 2,* 24. joulukuuta.

“Computing a Normal/Perpendicular vector”. 2018. Viitattu 22. elokuuta. `https://docs.unity3d.com/Manual/ComputingNormalPerpendicularVector.html`.

Elduque, Alberto. 2004. “Vector cross products”. *Talk presented at the Seminario Rubio de Francia of the Universidad de Zaragoza on April* 1:2004.

“Fluid Simulation for Video Games (Part 9)”. 2012. Viitattu 20. maaliskuuta 2018. `software.intel.com/en-us/articles/fluid-simulation-for-video-games-part-9`.

Hakkila, Pentti. 1979. “Wood Density Survey and Dry Weight Tables for Pine, Spruce and Birch Stems In Finland”. Luku 96.3 teoksessa *Metsäntutkimuslaitoksen julkaisuja 96.* Metsäntutkimuslaitos.

Kerner, Jacques. 2015. “Water interaction model for boats in video games”. Viitattu 20. maaliskuuta 2018. `https://www.gamasutra.com/view/news/237528/Water_interaction_model_for_boats_in_video_games.php`.

———. 2016. “Water interaction model for boats in video games: Part 2”. Viitattu 20. maaliskuuta 2018. `https://www.gamasutra.com/view/news/263237/Water_interaction_model_for_boats_in_video_games_Part_2.ph`.

Konami. 1989. *Teenage Mutant Ninja Turtles,* 12. toukokuuta.

Martin, George Edward. 2012. *The foundations of geometry and the non-Euclidean plane.* Springer Science & Business Media.

Matusiak, Jerzy. 1995. *Laivan Kelluvuus ja Vakavuus.* 2. painos. Otatieto OY.

Naughty Dog. 2011. *Uncharted 3: Drake's Deception,* 1. marraskuuta.

Nordeus, Erik. 2018. "Unity Boat Tutorial". Viitattu 20. maaliskuuta. http://www.habrador.com/tutorials/unity-boat-tutorial/2-basic-scene/.

Nürnberg, Robert. 2013. "Calculating the volume and centroid of a polyhedron in 3d". *url: http://www2.imperial.ac.uk/%7Ern/centroid.pdf.* Viitattu 1. elokuuta 2019.

Perlin, Ken. 1985. "An image synthesizer". *ACM Siggraph Computer Graphics* 19 (3): 287–296.

Rare. 1994. *Donkey Kong Country,* 21. marraskuuta.

Rath, Robert. 2014. "Why Games are Terrible at Water". Viitattu 25. huhtikuuta 2018. http://www.escapistmagazine.com/articles/view/video-games/columns/criticalintel/11840-Why-Games-are-Terrible-at-Water.

Resistance Committee of the 28th ITTC. 2017. "High Speed Marine Vehicles Resistance Test". Luku 7.5-02-05-01 teoksessa *ITTC Recommended Procedures and Guidelines.*

Serway A., Vuille C., ja Faughn J. 2009. *College Physics Eighth Edition: Buoyancy.* Cengage Learning.

Seymour, Mike. 2012. "Assassin's Creed III: The tech behind (or beneath) the action". Viitattu 20. maaliskuuta 2018. https://www.fxguide.com/featured/assassins-creed-iii-the-tech-behind-or-beneath-the-action.

Ubisoft. 2012. *Assassin's Creed III,* 31. lokakuuta.

Liitteet

A Apuluokat ja lisäfunktiot

Funktio A.1: public void DisplayMesh(Mesh debugMesh, string name, List<TriangleData> underwaterTriangleData)

```
1  List<Vector3> vertices = new List<Vector3>();
2  List<int> triangles = new List<int>();
3  //Rakenna kolmioverkko
4  for (int i = 0; i < underwaterTriangleData.Count; i++)
5  {
6      //From global coordinates to local coordinates
7      Vector3 p1 = objectTransform.InverseTransformPoint(
           underwaterTriangleData[i].p1);
8      Vector3 p2 = objectTransform.InverseTransformPoint(
           underwaterTriangleData[i].p2);
9      Vector3 p3 = objectTransform.InverseTransformPoint(
           underwaterTriangleData[i].p3);
10     vertices.Add(p1);
11     triangles.Add(vertices.Count - 1);
12     vertices.Add(p2);
13     triangles.Add(vertices.Count - 1);
14     vertices.Add(p3);
15     triangles.Add(vertices.Count - 1);
16 }
17 //Tyhjennä vanha verkko
18 debugMesh.Clear();
19 debugMesh.name = name;
20 //Aseta kolmiot ja kulmapisteet
21 debugMesh.SetVertices(vertices);
22 debugMesh.SetTriangles(triangles, 0);
23 debugMesh.RecalculateBounds();
```

Funktio A.2: public class VertexData

```
//Kulmapisteen etäisyys vedenpintaan
public float distance;
//Kulmapisteen indeksi
public int index;
//Globaali koordinaattivektori
public Vector3 globalVertexPos;
```

Funktio A.3: public struct TriangleData

```
public struct TriangleData
{
    //Globaalit koordinaattivektorit kulmapisteille
    public Vector3 p1;
    public Vector3 p2;
    public Vector3 p3;
    public Vector3 center;
    public Vector3 normal;
    public float area;
    public float distanceToSurface;
    public Vector3 velocity;
    public Vector3 velocityDir;

    //Kulma nopeuden suunnan ja normaalin välillä
    //=Negatiivinen jos suunta on normaalia vastaan
    //=Positiivinen jos suunta on normaalin suuntaan
    public float cosTheta;

    public TriangleData(Vector3 p1, Vector3 p2, Vector3 p3, Rigidbody
        objRB/*, int surfacePoints*/) // 0 all points underwater, 1
        second is surface, 2 two last surface
    {
        this.p1 = p1;
        this.p2 = p2;
        this.p3 = p3;
        this.center = (p1 + p2 + p3) / 3f;

```

```
26          float surfaceLevel = GameObject.FindWithTag("GameController").
                GetComponent<WaveController>().GetWaveYPos(center.x, center
                .z);
27          distanceToSurface = surfaceLevel - center.y;
28
29          this.normal = Vector3.Cross(p2 - p1, p3 - p1).normalized;
30          float a = Vector3.Distance(p1, p2);
31          float c = Vector3.Distance(p3, p1);
32          this.area = (a * c * Mathf.Sin(Vector3.Angle(p2 - p1, p3 - p1)
                 * Mathf.Deg2Rad)) / 2f;
33          this.velocity = GetTriangleVelocity(objRB, this.center);
34          this.velocityDir = this.velocity.normalized;
35          this.cosTheta = Vector3.Dot(this.velocityDir, this.normal);
36      }
37      private static Vector3 GetTriangleVelocity(Rigidbody rb, Vector3
            triangleCenter)
38      {
39          // v_A = v_B + omega_B cross r_BA
40          // v_A - nopeus pisteessä A
41          // v_B - nopeus pisteessä B
42          // omega_B - pyörimisnopeus pisteesä B
43          // r_BA - Suuntavektori A:n ja B:n välillä
44          Vector3 v_B = rb.velocity;
45          Vector3 omega_B = rb.angularVelocity;
46          Vector3 r_BA = triangleCenter - rb.worldCenterOfMass;
47          Vector3 v_A = v_B + Vector3.Cross(omega_B, r_BA);
48          return v_A;
49      }
50  }
```

Funktio A.4: public class SlammingForceData

```
1 //Alkuperäinen kolmioiden pinta-ala
2 public float originalArea;
3 //Kuinka iso pinta-ala on veden alla
4 public float submergedArea;
5 //Pinta-ala veden alla edellisellä kuvapiirrolla
6 public float previousSubmergedArea;
```

```
7 //Kolmion keskipiste, jonka avulla nopeus lasketaan
8 public Vector3 triangleCenter;
9 //Nopeus
10 public Vector3 velocity;
11 //Nopeus edellisessä kuvapiirrossa
12 public Vector3 previousVelocity;
```

Funktio A.5: void AddSimpleDrag()

```
1 var ratio = (underArea / origArea);
2
3 var dragValue = ((maxDrag - minDrag) * ratio) + minDrag;
4 var angDragValue = ((maxAngDrag - minAngDrag) * ratio) + minAngDrag;
5
6 thisRigidBody.drag = dragValue;
7 thisRigidBody.angularDrag = angDragValue;
```