De Simulatie van Botsingen

Profielwerkstuk Natuurkunde, 5e rev.

Michael D. Roeleveld
Begeleider: J.A. van de Konijnenberg

13 maart 2021

Inhoudsopgave

1 Inleiding **1**
1.1 Waarom simuleren? 1
1.2 Wat voor voorwerpen kun je simuleren? 1
1.3 Wat voor soort interacties kun je simuleren? 1

2 Termen, symbolen, conventies **2**
2.1 Vectoren 2
2.2 Matrices 2
2.3 Decimaalpunten 2
2.4 Termen en symbolen voor natuurkundige grootheden 2

3 Computationele Geometrie **3**
3.1 3d-modellen 3
3.2 Massa, Massamiddelpunt, Inertiatensor 3
3.3 Rotatie 3
3.3.1 Eulerhoeken 3
3.3.2 As-Hoek representatie 4
3.3.3 Quaternionen 5

4 Mechanica **6**
4.1 Verplaatsing 6
4.1.1 Krachtstoten 6
4.2 Rotatie 6
4.2.1 Tensors 7

5 Botsingen detecteren **7**
5.1 Cirkels en bollen 7
5.2 Polygonen 8
5.3 Polyeders (3d) 9
5.3.1 Optimalisatie: AABB 10
5.4 Manifold 10

6 Botsingen oplossen **11**
6.1 In 2d 11
6.2 In 3d 11
6.3 Luchtweerstand 12
6.4 Wrijving 12
6.5 Nieuwe snelheid en rotatie 12

7 Numeriek Integreren **13**
7.1 Eulermethode 13
7.2 Verletintegraal 14
7.3 Velocity-Verletintegraal 14
7.4 Runge-Kutta 14

8 Numerieke imperfecties **15**
8.1 Limieten in data 15
8.2 Onrepresenteerbare getallen 15
8.3 Tunneling 16

9 Eigen werk **16**

10 Conclusie **16**

11 Evaluatie **16**

12 Bronnen **17**

Voorwoord

Met dank aan Erik Cassel (R.I.V.) en Emil Ernerfeldt. Jullie software, Roblox en Phun/Algodoo resp., heeft mij geïnspireerd om dit project te ondernemen.

Ik wil ook meneer v.d. Konijnenberg bedanken voor zijn geduld en wijsheid.

1 Inleiding

Je hebt vast wel een game gespeeld of een CGI scène gezien waarin botsingen werden gemodelleerd. Maar hoe werkt dat? Hoe weet de computer wanneer iets botst, en hoe weet het ding de botsing weer op te lossen? Dat blijkt een heel vakgebied te zijn, en zodoende kan dit artikel in al zijn lengte licht schijnen op slechts het puntje van de ijsberg van natuurkundige simulaties, en een basis bieden aan de lezer om de ijsberg zelf verder te onderzoeken.

1.1 Waarom simuleren?

Behalve games en films hebben ook partijen als de wetenschap en defensie baat bij natuurkundige simulaties. Er bestaan programma's die interacties tussen kogels en pantser kunnen modelleren, of software die interacties tussen moleculen modelleren. Een simulatie is een makkelijke manier om iets uit te proberen, want het enige wat je ervoor nodig hebt is een computer en tijd. Voor grovere simulaties in bijvoorbeeld games, heb je tegenwoordig geen supercomputer meer voor nodig - het is *real time*.

1.2 Wat voor voorwerpen kun je simuleren?

Als je 2 voorwerpen laat botsen, kun je de voorwerpen grofweg op 2 manieren categoriseren:

- Starre voorwerpen (rigid bodies)
- Zachte voorwerpen (soft bodies)

Een star voorwerp kan niet vervormen. In het echt bestaan deze voorwerpen niet en kan alles wel buigen of anderzijds vervormen of breken. Maar om de simulatie makkelijker te maken wordt er in de industrie veel gewerkt met zogeheten *rigid bodies*. Van een afstandje lijken de interacties tussen rigid bodies meer dan "goed genoeg" voor de meeste zaken, en *rigid body dynamics* zijn redelijk makkelijk en dus snel te berekenen. Dit leent zich naar real-time simulaties, wat nodig is bij games. Het is ook de oudste methode: zie het "originele rigid body dynamics" artikel [1] uit 1989. Het typische voorbeeld van een *rigid body collision* zijn stotende biljartballen, maar in het echt deformeren ze een piepklein beetje.

Soft bodies daarentegen komen in meerdere smaken: een blok gelatine is een softbody, dat kan vervormen en daarna weer zijn oude vorm aanneemt, maar je hebt ook permanente deformaties, denk aan een plaat metaal die wordt verbogen. Soft bodies zijn nog een groot onderzoeksonderwerp. Zie [2] voor een van de eerste artikelen hierover door dezelfde auteur als [1]. Tegenwoordig kom je veel *particle based* implementaties [3] tegen. Daarin worden objecten gerepresenteerd als een verzameling punten, denk aan (heel) grote moleculen, dat doet denken aan de manier waarop een digitale foto wordt gerepresenteerd als een verzameling pixels. Particle-based physics leent zich voor effecten als doormidden hakken, vervormen, of juist terugveren naar een oude vorm. Helaas gaan we het verder niet behandelen in dit profielwerkstuk; dit artikel is gericht op rigid body dynamics.

1.3 Wat voor soort interacties kun je simuleren?

Interacties als botsingen kun je opdelen in weer 2 categorieën:

- elastische botsingen
- inelastische botsingen

Bij elastische botsingen gaat er geen kinetische energie verloren. Botsigen tussen biljartballen zijn bijvoorbeeld *nagenoeg* elastisch (ook al zijn de biljartballen zelf niet echt elastisch).

Bij inelastische botsingen gaat er kinetische energie verloren in de vorm van warmte. Een botsing tussen 2 auto's is inelastisch, want de kinetische energie gaat "verloren" bij het verbuigen van het metaal, en de energie wordt omgezet naar warmte. Een oude manier waarmee smeden vuur aanmaakten was om een staaf ijzer tot een klein puntje te hameren. Dit puntje werd dan zo heet dat het het tondel in brand kon zetten. De botsing tussen het puntje en de hamer is dus ook een voorbeeld van een inelastische botsing.

Dit artikel zal zich focussen op elastiche botsingen. Om te zorgen dat we geen simulatie krijgen van stuiterballen die nooit ophouden met stuiteren (immers, er gaat geen kinetische energie verloren bij elastische botsingen), zullen we een klein stukje van de kinetische energie bij een botsing *handmatig* opsnoepen. Later meer over het zogeheten *restitutiecoëfficiënt*.

Je kan ook interacties tussen twee vloeistoffen/gassen, of een vloeistof en een voorwerp modelleren [4], maar ook dat ligt buiten de strekking van dit artikel.

2 Termen, symbolen, conventies

2.1 Vectoren

Dit artikel neemt aan dat de lezer bekend is met vectoren. Dit artikel neemt ook aan dat een 3d vector in essentie hetzelfde is als een 3d coördinaat.
In Engelse literatuur wordt een vector vaak dikgedrukt angeduid: $\boldsymbol{v}$, maar in nederland doet we dat met een pijltje boven de letter: $\vec{v}$. Dit artikel gebruikt een pijltje om vectoren aan te geven. Een hoedje, $\hat{n}$ geeft aan dat het een eenheidsvector betreft.

Vectoren kun je zowel liggend als opstaand weergeven: $\vec{v} = (x, y, z) = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$

2.2 Matrices

In Engelse literatuur worden ook matrices dikgedrukt aangegeven ($\boldsymbol{M}$). In Nederland enkel met een hoofdletter (M). Maar niet altijd! Soms wordt er natuurlijk een kleine Griekse letter gebruikt. In feite is de notatie voor matrices dus heel los.

2.3 Decimaalpunten

Dit artikel gebruikt een punt als decimaalpunt, en een komma voor het scheiden van gegevens.

2.4 Termen en symbolen voor natuurkundige grootheden

Dit was de grootste bron van verwarring tijdens mijn onderzoek. Let vooral op termen als "impuls" en "moment". Voor nadere uitleg zie sectie 4 Mechanica.

NL	EN	Symbool	Eenheid
Plaats	Position	NL: $\vec{s}$ EN: **x**	m
Snelheid	Velocity	$\vec{v}$	m/s
Impuls	Momentum	$\vec{p}$	Ns (=kg·m/s)
Versnelling	Acceleration	$\vec{a}$	m/s/s
Kracht	Force	$\vec{F}$	N
(Kracht)stoot	Impulse	$\vec{J}$	Ns (F·dt)
Massa	Mass	m	kg

Voor de rotatietabel, houdt in gedachte dat dit artikel te maken heeft met 3d ruimte. Dus rotationele kwantiteiten zijn niet met scalaire waardes te representeren, daarom zijn het vectors.

NL	EN	Symbool	Eenheid
Rotatie/hoek	Rotation/angle	$\vec{\theta}$	[radialen]
Hoeksnelheid	Angular velocity	$\vec{\omega}$	[radialen]/s
Impulsmoment	Angular momentum	$\vec{L}$	Nms
Hoekversnelling	Angular acceleration	$\vec{\alpha}$	[radialen]/s/s
(kracht)moment	Torque/moment of force	NL: $\vec{M}$ EN: $\boldsymbol{\tau}$	Nm
(kracht)momentstoot	Angular impulse	J	Nms
Traagheidsmoment	(Moment of/rotational) inertia	I	geen

3 Computationele Geometrie

3.1 3d-modellen

Voor een simulatie hebben we eerst een model nodig om de voorwerpen te representeren. Op een computer wordt een 3d voorwerp vrijwel altijd gerepresenteerd als een verzameling driehoeken, elk met drie punten: vertices, die weer hebben x, y, z coordinaten hebben. Dit heet een *mesh*.
Voor elk voorwerp zullen we ook een rotatie/draaisnelheid, positie/snelheid, massa/traagheidsmomenttensor bijhouden. Het massamiddelpunt is vaak impliciet: in het inertiaalstelsel/lokale assenstelsel van het voorwerp is $(0, 0, 0)$ het massamiddelpunt, mmp.

3.2 Massa, Massamiddelpunt, Inertiatensor

Als je meer wilt simuleren dan alleen balletjes en blokjes, is het handig om te werken met *meshes*: een verzameling driehoeken. Maar van geen unieke mesh is de traagheidsmomenttensor, massa, massamiddelpunt bekend.
Als we de aanname maken dat elk voorwerp een constante dichtheid heeft, dan kunnen we van elk voorwerp een massa (vanuit volume), massamiddelpunt, en traagheidsmomenttensor berekenen. Er zijn 3 voornaamste methodes hiervoor:

Tetgen: We kunnen het voorwerp afbreken in tetraëders, en de mmp'en + massa's van die tetraëders berekenen en daarvan het gewogen gemiddelde nemen. Deze methode geeft zowel volume als mmp als traagheidsmomenttensor. [5][6]

Monte Carlo: We kunnen een (groot) aantal willekeurige punten uitkiezen die niet buiten het voorwerp zitten, en dan het gemiddelde van alle punten nemen als mmp. Deze methode geeft het volume het mmp en de traagheidsmomenttensor (gebruik hiervoor de stelling van Steiner).

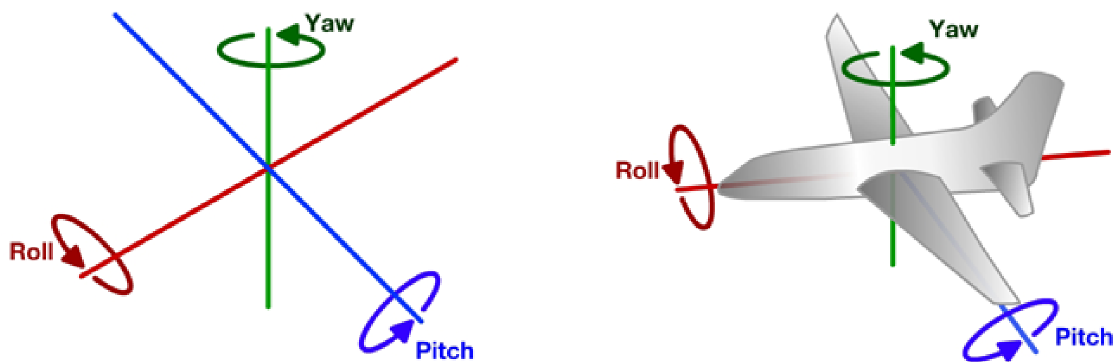
Centroid: Snel, maar niet helemaal correct: je neemt het middelpunt van elke driehoek gewogen met zijn oppervlakte en neemt het gemiddelde punt als mmp. Dit is niet helemaal correct omdat je dan aanneemt dat het voorwerp een holle huls is. Deze methode geeft geen volume, alleen het mmp.[7]

3.3 Rotatie

In 2d heb je 1 as van rotatie. In 3d ligt het wat ingewikkelder.

3.3.1 Eulerhoeken

Euler brak rotaties op in 3 assen: een rotatie om de x-as, de y-as, en tot slot de z-as. Dit werkt goed genoeg voor veel doeleinden, maar deze representatie heeft last van een naar probleem: gimbal lock. Als je een gyroscoop met 3 assen van rotatie hebt, kun je 2 ringen op elkaar draaien. Dan kun je het binnenste voorwerp (de gimbal) ineens niet meer vrij draaien, maar om slechts 2 assen. Je moet de ringen eerst uit elkaar halen om de *gimbal* weer te *unlocken*. Zie [8] voor een grafische uitleg.



Figuur 1: De 3 assen van rotatie. Afbeelding: crazepony.com

Alhoewel je elke 3d orientatie correct kan weergeven in Euler hoeken, kun je niet altijd een *verandering* van orientatie (oftewel draaisnelheid) toepassen met Euler hoeken, vanwege die gimbal lock.

In een typische toepassing worden Eulerhoeken gerepresenteerd als een vector $\vec{\Omega} = (\psi, \phi, \theta)$: in het Engels* schrijven ze "roll" toe aan ψ, "pitch" aan ϕ, en "yaw" aan θ. Een Eulerhoek wordt eerst omgezet naar een rotatiematrix (matrixvermenigvuldiging gaat van rechts naar links):

$$\mathbf{R} = (\mathbf{R}_\theta \times (\mathbf{R}_\phi \times \mathbf{R}_\psi)) = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} \cos(\phi) & 0 & \sin(\phi) \\ 0 & 1 & 0 \\ -\sin(\phi) & 0 & \cos(\phi) \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\psi) & -\sin(\psi) \\ 0 & \sin(\psi) & \cos(\psi) \end{bmatrix} \tag{1}$$

$$\mathbf{R} = \begin{bmatrix} \cos(\theta)\sin(\phi) & \cos(\theta)\sin(\phi)\sin(\psi) - \sin(\theta)\cos(\psi) & \cos(\theta)\sin(\phi)\cos(\psi) + \sin(\theta)\sin(\psi) \\ \sin(\theta)\cos(\phi) & \sin(\theta)\sin(\phi)\sin(\psi) + \cos(\theta)\cos(\psi) & \sin(\theta)\sin(\phi)\cos(\psi) - \cos(\theta)\sin(\psi) \\ -\sin(\phi) & \cos(\phi)\sin(\psi) & \cos(\phi)\cos(\psi) \end{bmatrix} \tag{2}$$

Die rotatiematrix kun je dan vermenigvuldigen met een punt (vector) om een puntrotatie uit te voeren. Als je elk punt/vertex van een 3d model roteert, roteer je het hele model. Je kan ook een rotatiematrix weer omzetten naar een vector, maar dit is buiten de strekking van dit artikel en ik refereer je hiervoor naar [9]

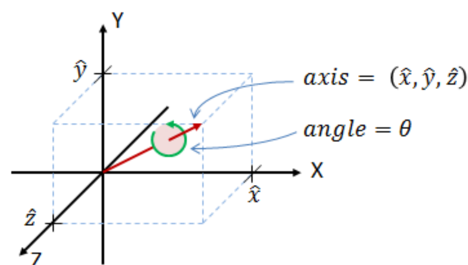
*Het Nederlands heeft hier helaas geen goede woorden voor. Ik zou zeggen roll is rol, pitch helling, yaw draaiing, maar als je naar beweegbare monitorbeugels gaat kijken en naar de fabrikanten luistert, dan is pitch kanteling, roll draaiing, yaw wenteling.

3.3.2 As-Hoek representatie

Een oplossing voor gimbal-lock is bijvoorbeeld om elk voorwerp rond een per-situatie-uitgekozen as te draaien. De as wordt gerepresenteerd via een eenheidsvector van de vorm (x, y, z), en de hoek θ, of w is de "rol" om die vector als as. Vanuit deze representatie kun je echter niet direct punten roteren, en je moet het eerst omzetten naar een quaternion of rotatiematrix.

$$\vec{a} = \begin{pmatrix} w \\ x \\ y \\ z \end{pmatrix} \tag{3}$$

Een rotatie van 180° rond de y-as is: $(180, 0, 1, 0)$



Figuur 2: Een as-hoek: groen is de hoek, de rode vector is de richting. Afbeelding: danceswithcode.net

N.B. Je zal waarschijnlijker werken met radialen en niet graden:
elke zogeheten *math library* die ik tot nu toe ben tegengekomen
werkt in radialen.

3.3.3 Quaternionen

Quaternionen zijn nog een manier om rotatie te representeren [10] [11]. Ze zijn echter **niet** synoniem aan as-hoeken. Quaternionen hebben te maken met complexe getallen en hebben een aantal eigenschappen en identiteiten die het gebruik ervan elegant kunnen maken; dat is echter buiten de strekking van dit artikel. Een quaternion heeft de vorm:

$$q = a + bi + cj + dk \tag{4}$$

a is het reële deel en b, c, d is het imaginaire deel. De *imaginaire eenheden* i, j, k produceren -1:

$$i \cdot j \cdot k = -1 \tag{5}$$

$$i^2 = j^2 = k^2 = -1 \tag{6}$$

Vanaf nu zal ik een quaternion representeren als een 4d vector (a, b, c, d). De imaginaire eenheden zijn dan impliciet.
N.B. in oude literatuur worden 3d vectoren soms geschreven als $ax + by + cz$, met a, b, c variabelen en x, y, z de assen, maar tegenwoordig worden ze genoteerd als (a, b, c). Je kan daarmee een parallel trekken hoe ik nu met quaternionen omga.

Je kan vanuit een as-hoek (w, x, y, z) een quaternion (a, b, c, d) verkrijgen:

$$\vec{q} = \begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix} = \begin{pmatrix} \cos(w/2) \\ \sin(w/2) \cdot x \\ \sin(w/2) \cdot y \\ \sin(w/2) \cdot z \end{pmatrix} \tag{7}$$

Je combineert rotaties door het **hamiltonproduct p** tussen twee quaternionen **q** en **r** te nemen:

$$p = qr = \begin{pmatrix} a_q a_r - b_q b_r - c_q c_r - d_q d_r \\ a_q b_r + b_q a_r + c_q d_r - d_q c_r \\ a_q c_r - b_q d_r + c_q a_r + d_q b_r \\ a_q d_r + b_q c_r - c_q b_r + d_q a_r \end{pmatrix} \tag{8}$$

Je roteert een punt $\vec{v}$ met een quaternion $\vec{q}$ door het te sandwichen tussen de genormaliseerde (elk element van q gedeeld door de lengte van q) quaternion $\hat{q}$ en de geconjugeerde van $\vec{q}$, $\vec{q}'$. "geconjugeerd" wil zeggen dat b_q, c_q en d_q zijn vermenigvuldigd met -1. Je voegt ook een term $a = 0$ toe aan $\vec{v}$ zodat het een "frankenquaternion" $\vec{v}' = (0, x, y, z)$ wordt. Je neemt dan het Hamiltonproduct van $\hat{q}$ en $\vec{v}$, en dan nog eens het Hamiltonproduct met $\vec{q}'$. Na de rotatie halen we de term a gewoon weer van $\vec{v}$ af, zodat het weer een 3d punt of vector is:

$$\vec{v}_{gedraaid} = \hat{q}\vec{v}'\vec{q}' = \text{ham}(\text{ham}(\hat{q}, \vec{v}'), \vec{q}') \tag{9}$$

Een quaternion normaliseer je net zoals een vierdimensionale vector, je deelt elk element door de lengte van de quaternion.

$$\hat{q} = \frac{\vec{q}}{\sqrt{a_q^2 + b_q^2 + c_q^2 + d_q^2}} \tag{10}$$

Soms nemen we ook kruisproducten, bijvoorbeeld bij stukken over hoeksnelheid. In die gevallen bedoel ik puur het kruisproduct tussen de (b, c, d) componenten van de quaternionen. Je kan de quaternionen ook omrekenen naar Eulerhoeken of as-hoeken, deze formules zijn op het internet te vinden en vallen buiten de strekking van dit artikel.

4 Mechanica

Deze sectie dient als kleine refresher in mechanica om hoofdstuk 6 volledig te begrijpen. Hier gebruik ik Euler's methode van integreren. Dit heeft verschillende nadelen besproken in hoofdstuk 7, maar het is makkelijk te volgen en biedt een mooi overzicht.

4.1 Verplaatsing

Een voorwerp heeft een snelheid $\vec{v}$ en een positie $\vec{s}$.

$$\vec{s}_{nieuw} = \vec{s}_{oud} + \vec{v} \cdot dt \tag{11}$$

De versnelling $\vec{a}$ kun je berekenen met

$$\vec{a} = \vec{F}_{res}/m \tag{12}$$

Waarbij $\vec{F}_{res}$ de som is van alle krachten die werken op het voorwerp, en m de massa van het voorwerp.

$$\vec{v}_{nieuw} = \vec{v}_{oud} + \vec{a} \cdot dt \tag{13}$$

4.1.1 Krachtstoten

Een krachtstoot is een kracht die een korte tijd wordt uitgeoefend, denk aan een kogel die in korte tijd vooruit wordt gestuwd door de explosie van buskruit. De tijdsduur van een krachtstoot definiëren we bij deze als de grootte van de tijdstappen die we maken in onze simulatie. Stel, we simuleren met 100Hz, dan is de tijdsduur van een krachtstoot dus 1/100Hz = 0.01s. We werken in de context van krachtstoten en botsingen vaak met "voor" en "na" toestanden. Dit refereert naar *voor* of *na* de botsing.

4.2 Rotatie

Een kracht uitgeoefend op een voorwerp zal vaak niet alleen het voorwerp een versnelling geven; in de meeste gevallen ook een hoekversnelling. Wat een kracht is voor verplaatsing (versnelling, a), is een moment τ voor draaiing (hoekversnelling, α).

Een voorwerp heeft een bepaalde oriëntatie om zijn massamiddelpunt, en elke tijdstap komt daar een beetje bij op volgens de hoeksnelheid ω:

$$\theta_{nieuw} = \theta_{oud} + \omega \cdot dt \tag{14}$$

Ook kun je ω uitdrukken als integraal van de hoekversnelling α:

$$\omega_{nieuw} = \omega_{oud} + \alpha \cdot dt \tag{15}$$

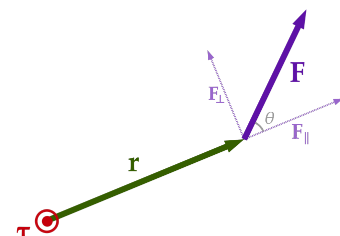
Let op dat je in de vergelijkingen 14 en 15 niet letterlijk de variabelen kunt *sommeren*. Met quaternionen moet je zoals eerder besproken in 3.3.3 het Hamiltonproduct nemen. Ik gebruik het plus-teken alleen voor de intuitie dat er een stukje draaiing bij komt. [12] (let bij deze bron op dat de auteur onconventionele notatie gebruikt, met name voor het kruisproduct.)
Hoekversnelling bereken je met behulp van het moment τ en de weerstand die het voorwerp biedt tegen hoekversnelling, traagheidsmoment I met de hoeksnelheid ω.

$$\alpha_{nieuw} = I^{-1} \cdot (\vec{\tau}_{res} - (\vec{\omega}_{oud} \times (I \times \vec{\omega}_{oud}))) \tag{16}$$

τ_{res} is de som van alle momenten, en I is het traagheidsmoment, grofweg een weerstand tegen verandering in hoeksnelheid, zie sectie 4.2.1.
We berekenen één moment als volgt, zie fig. 3 voor toelichting:

$$\vec{\tau} = \vec{r} \times \vec{F}_{\perp} \tag{17}$$

Omdat voor het moment belangrijk is *waar* op het voorwerp de kracht uitgeoefend wordt, moeten we voor elke kracht $\vec{F}$ een moment $\vec{\tau}$ berekenen. Als we eenmaal alle momenten hebben kunnen we ze simpel opsommen om $\vec{\tau}_{res}$ te krijgen. Sommige krachten, zoals zwaartekracht, worden vanaf het massamiddelpunt toegepast en hebben dus geen moment.



Figuur 3: De invloed van $\vec{F}$ op τ is $\vec{F}_{\perp}$. De arm is r. Afbeelding: wikimedia commons

4.2.1 Tensors

In 3d heb je te maken met traagheidsmoment*tensors*. In het algemeen betekent tensor "Een matrix van getallen die één eigenschap representeert". De tensor is in onze context (3d-rotatie-traagheidsmoment) een 3x3 matrix. Voor bijvoorbeeld een balk van breedte b, hoogte h, diepte d, en massa m is die matrix als volgt:

$$I_{balk} = \begin{bmatrix} \frac{1}{12}m(h^2+d^2) & 0 & 0 \\ 0 & \frac{1}{12}m(b^2+d^2) & 0 \\ 0 & 0 & \frac{1}{12}m(b^2+h^2) \end{bmatrix} \quad (18)$$

Voor elk uniek voorwerp is deze tensor weer anders: als je voorwerpen als een verzameling deeltjes ziet, dan dragen sommige deeltjes' massa meer bij aan I dan die van andere deeltjes, met name massa aan de rand van een voorwerp telt (kwadratisch) zwaar mee. Stel je wilt een pirouete maken: als je je armen uitstrekt hebt is het lastiger om te beginnen met draaien dan wanneer je je armen naast je hebt, omdat je meer energie nodig hebt voor de deeltjes in je armen omdat ze ver weg zijn. Als je je armen dan inklapt ga je sneller draaien, omdat je minder energie nodig hebt voor deeltjes die dichterbij de as zitten. In ieder geval heb je verschillende traagheidsmomenttensors in de twee situaties. Zie [13] voor de tensors van nog enkele andere vormen.

Je kan het traagheidsmoment I krijgen door de inverse van een I-tensor te nemen en hem te vermenigvuldigen met een as in de vorm van een 3d-vector. Je gebruikt de **stelling van Steiner** om de I te krijgen voor een bepaalde plaats op het voorwerp (dit haakt ook weer in op het pirouete-voorbeeld):

$$I' = I + md^2 \quad (19)$$

I' is het aangepaste traagheidsmoment voor het punt waarop we de kracht uitoefenen. d is de afstand van het punt van uitoefening tot het massamiddelpunt, m is de massa van het voorwerp. **Let op:** in 3d kun je deze formule alleen gebruiken voor assen die parallel zijn: anders moet je I opnieuw berekenen met behulp van de tensor.
In 2d draaien alle voorwerpen om dezelfde *lokale as*, d.w.z. de as die door hun massamiddelpunt loopt: deze as staat altijd haaks op het 2d vlak, oftewel een vector die recht omhoog uit dit papier springt.

5 Botsingen detecteren

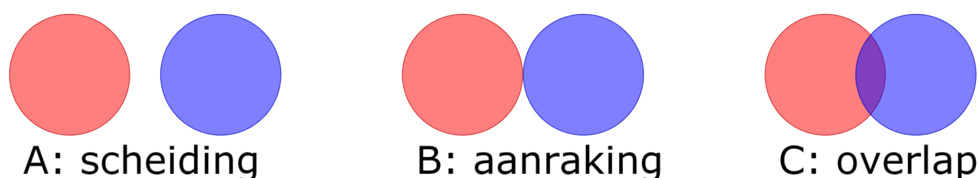
5.1 Cirkels en bollen

Dit is redelijk simpel. Om te kijken of twee cirkels **a** en **b** botsen, kijk je wat de afstand tussen de twee cirkels is met de stelling van Pythagoras. Je neemt de afstand tussen de middelpunten van **a** en **b**, en trekt de radii er van af.

$$\vec{d} = \vec{a_{midden}} - \vec{b_{midden}} \quad (20)$$

$$D = \sqrt{d_x^2 + d_y^2} - a_{radius} - b_{radius} \quad (21)$$

We kunnen onderscheid maken tussen 3 gevallen (zie fig. 4):
A: Als $D > 0$, zijn de cirkels gescheiden.
B: Als $D = 0$, raken de cirkels elkaar precies.
C: Als $D < 0$, overlappen de cirkels elkaar.



Figuur 4: 3 gevallen

Dit kun je makkelijk uitbreiden naar de derde dimensie, want ook daar is de stelling van Pythagoras gewoon waar:

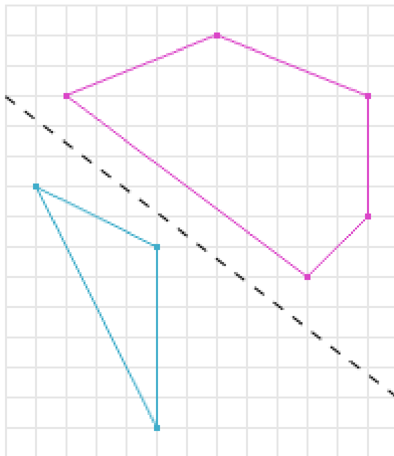
$$D = \sqrt{x_d^2 + y_d^2 + z_d^2} - r_a - r_b \tag{22}$$

Met r_x de radius van lichaam x, en $\vec{d}$ het verschil in positie.

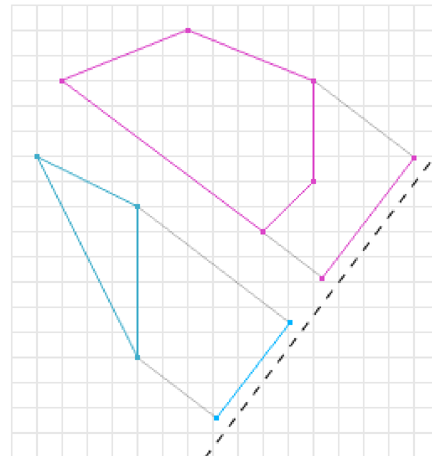
5.2 Polygonen

Een polygon is een verzameling lijnstukken die op elkaar aansluiten zodat er geen gaten in zitten. Zo'n verzameling van lijnen maakt een 2d vormpje, bijvoorbeeld een driehoek **ABC** die uit 3 lijnstukken **AB**, **BC**, **CD** bestaat.
Een populaire methode om botsingen te detecteren tussen polygonen is het **Gilbert–Johnson–Keerthi distance algorithm (GJK)**: zie [14] [15] [16].

Je hebt ook nog **Separating Axis Theorem (SAT)**. Het idee achter SAT is, als je een lijn kan trekken tussen twee voorwerpen zonder dat de lijn door een van de voorwerpen gaat, dan botsen de polygonen niet. [17] Zie fig. 5. Over het algemeen is SAT makkelijker om te implementeren en te beredeneren dan GJK, maar GJK is sneller. Dit artikel zal ingaan op SAT.



Figuur 5: Lijn tussen een driehoek en een 5-hoek.

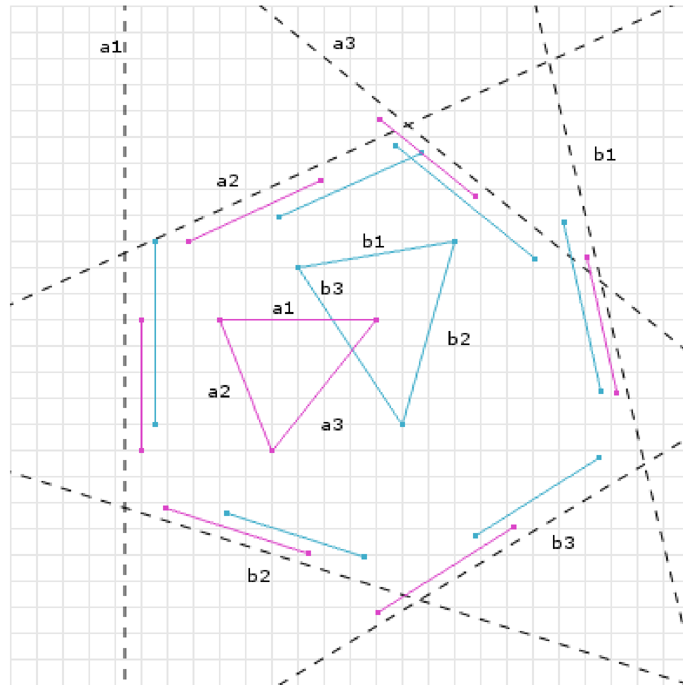


Figuur 6: Projectie van de twee vormen op één as van de 5-hoek.

Afbeeldingen: www.dyn4j.org

Om te kijken of polygon **A** met polygon **B** botst, ga je langs alle lijnstukken waaruit **A** bestaat, en bij elk lijnstuk maak je een lijn die er haaks op staat, de *as*. Je projecteert dan zowel **A** als **B** (alle lijnstukken ervan) op die as, om te kijken of ze overlappen, en als dat niet het geval is dan is die as *separating*. (zie fig. 6 voor de uitwerking van 1 as (van het roze vormpje)).

Van alle projecties op alle assen, als **A** en **B** in alle gevallen overlappen, dan botsen ze. Als er sprake is van alleen maar aanrakingen... Dan raken ze elkaar aan. Als er geen sprake is van zowel overlap als aanraking, dan zijn A en B ruimtelijk gescheiden.



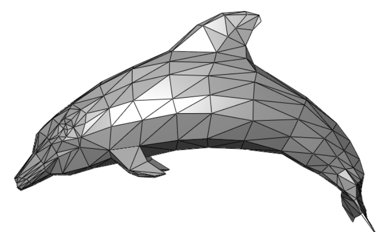
Figuur 7: De projecties van twee driehoeken A en B op hun assen. Afbeelding: www.dyn4j.org

Zie fig. 7 voor een volledige weergave van SAT op twee driehoeken A en B. De assen zijn gelabeld naar de lijnstukken waar ze haaks op staan. Merk op dat er op elke as overlap is tussen de projecties.

Een nadeel van SAT is dat het alleen werkt met convexe polygonen, en niet concave. Denk bij een concaaf polygon aan pacman, er zit een hapje in de vorm. Terwijl een convex polygon juist geen “inkepingen” heeft, zodat er geen enkele lijn bestaat door het convexe polygon die meer dan twee snijpunten heeft. Een lijn door een concaaf polygon kan meer dan twee snijpunten hebben. Een lijn die van boven naar onder door de mond van pacman gaat, heeft 4 snijpunten: 2x snijdt de lijn zijn mond, en 2x zijn hoofd. Een oplossing hiervoor is om pacman op te splitsen in twee convexe polygonen: de onderste helft van pacman en de bovenste helft. Nu kunnen we SAT individueel toepassen op de twee helften van pacman, en de resultaten samenvoegen. Zie ook [18] voor een andere aanpak.

5.3 Polyeders (3d)

Een polyhedron, of polyeder is een vorm in 3d die bestaat uit een verzameling polygonen. In games wordt dit aantal altijd beperkt tot driehoeken, omdat GPUs nou eenmaal met driehoeken werken. 3d-modellering gebeurt vaak met vierhoeken of zogeheten quads.



Figuur 8: Een dolfijn (polyeder) gemaakt uit driehoeken (polygonen). Afbeelding: wikimedia commons.

Je kunt SAT makkelijk naar 3d ombouwen: nu zoek je niet een lijn die je tussen twee voorwerpen kan maken zonder snijpunt, maar een vlak. Als je een vlak kan maken zodat het tussen de twee voorwerpen in ligt, en geen van beide voorwerpen snijdt, dat botsen de voorwerpen niet. Anders botsen of raken ze elkaar. Nu maak je voor elk vlak een normaalvector, alle normaalvectoren zijn een deel van alle assen die je itereert. De andere helft (eigenlijk 3/4) zijn alle randen van de driehoeken/polygonen. Om de vormen te projecteren gebruik je nog steeds het inproduct: je neemt de as en projecteert daarop alle edges d.m.v. het inproduct.

5.3.1 Optimalisatie: AABB

Omdat SAT traag is (omdat je zo veel vergelijkingen en berekeningen moet maken, en het algoritme verschrikkelijk schaalt, grofweg n^3 berekeningen in 3d, n is het aantal voorwerpen), kun je eerst een simpele doos om de voorwerpen heen maken, bestaande uit 2 punten. Je houdt een minimum en maximum vector bij, en dan itereer je elk punt in de vorm. Elke keer als een element uit het punt kleiner is dan het element uit de minimumvector of groter dan het element uit de maximumvector (bijvoorbeeld $punt_x < min_x$ of $punt_z > max_z$) dan werken we de desbetreffende vector bij.

Je kunt nu een balk maken tussen de minimum- en maximumvector. Die balk is dan een as Aligned Bounding Box (AABB). Een doos/balk heeft eigenlijk 8 punten, maar die hoeven we niet expliciet te definiëren doordat de lijnstukken van een AABB altijd parallel staan op de x, y of z as (vandaar de naam).

Als je voor elk voorwerp een AABB maakt, kun je voor botsingdetectie eerst vergelijken of er overlap/aanraking is tussen twee AABBs **A** en **B** door de maxima te vergelijken met de minima:
Geval 1: er is overlap op de x-as:

$$A.max_x > B.min_x \vee B.max_x > A.min_x \tag{23}$$

Geval 2: er is aanraking op de x-as:

$$A.max_x = B.min_x \vee B.max_x = A.min_x \tag{24}$$

Als je deze controles ook uitvoert op de y en z as, en alle resulterende logische vergelijkingen waar zijn, dan is er resp. een overlap of aanraking tussen A en B. Zo kun je veel werk voorkomen met een paar simpele checks, want pas als de AABBs aanraken of overlappen hoef je het intensievere SAT toe te passen. Bij grote aantallen voorwerpen kan dit aanzienlijke snelheidswinsten leveren. AABBs werken natuurlijk ook in 2d; je laat de z-as gewoon achterwege. [19]

Je kan ook versnellingsstructuren maken met AABBs: je kan de ruimte partitioneren in regio's (trefwoord: (binary) space partitioning) of de voorwerpen partitioneren in groepen (trefwoord: bounding volume hierarchy). Je krijgt dan een *tree*, ofwel hiërarchie, van voorwerpen, waarmee je eenvoudig veel botsingen uit kan sluiten.

5.4 Manifold

In de literatuur over rigid body dynamics wordt er vaak gesproken over een zogeheten *Collision Manifold*. Dit is niets anders dan het contactoppervlakte. SAT geeft ons standaard alleen een minimumverplaatsvector om de botsing op te lossen. De normaalvector van het het manifold is ook diezelfde minimumverplaatsvector. Het contactpunt is wat lastiger te krijgen: voor een zo exact mogelijk resultaat moet je gebruik maken van *polygon clipping*[20], waardoor je met de gegevens van SAT een punt, oppervlakte of volume krijgt. In het geval van een volume of oppervlakte moet je het massamiddelpunt ervan als contactpunt nemen.

Het contactpunt krijg je ook op deze wijze: tijdens het itereren van alle assen houdt je bij op welke as de overlap het kortst is. Het contactpunt ligt op deze as, en je neemt het punt dat midden in de overlap zit als contactpunt.

En de minimumverplaatsingvector krijg je op deze wijze: net als bij het contactpunt kijken we weer naar de kortste as. De lengte van overlap is de lengte van onze verplaatsingsvector, en de richting van de as is de richting van onze verplaatsingsvector. Afhankelijk van welk voorwerp je het bekijkt moet je deze vector wellicht 180° omdraaien. Aangezien we met convexe voorwerpen werken kunnen we controleren of de vector naar buiten wijst door het inproduct te nemen met de vector richting het massamiddelpunt vanaf het contactpunt.

En de normaalvector voor het aanrakingsvlak/manifold kun je uit de minimumverplaatsingvector halen: je hoeft hem alleen maar te normaliseren. Waarom? De kortste route om voorwerp a uit voorwerp b te halen zal wel over een normaalvector *moeten*, want anders is de vector niet loodrecht op het oppervlakte en loopt hij dus scheef; scheve vectoren zijn langer dan rechte als ze tot eenzelfde "hoogte" lopen, dus de kortste route moet wel een normaalvector zijn. In de gevallen dat je een overlap heb over alleen maar randen, dan kun je het zien als de gemiddelde normaalvector van de twee aangrenzende vlakken, gewogen door hun aandeel in het manifold.

6 Botsingen oplossen

Stel we hebben een botsing tussen twee voorwerpen **a** en **b**, en hun eigenschappen zijn bekend. We hebben de normaalvector $\hat{n}$ van het manifold, en het punt van contact $\vec{p}_c$. Nu kunnen we de grootte j van de krachtstoot berekenen tussen de twee voorwerpen.
We berekenen eerst nog het volgende als voorbereidend werk voor het werken met draaiingen:

$$\vec{r}_a = \vec{p}_c - \overrightarrow{mmp}_a \tag{25}$$

$$\vec{r}_b = \vec{p}_c - \overrightarrow{mmp}_b \tag{26}$$

6.1 In 2d

$$j = \frac{-(1+e)((\vec{v}_a - \vec{v}_b) \cdot \hat{n})}{\hat{n} \cdot \hat{n}(1/m_a + 1/m_b)} \tag{27}$$

Nu kan j ingevuld worden alsof het een kracht is door het te delen door de massa, en te vermenigvuldigen met de richting van de normaalvector van het manifold:

$$\vec{v}_{a,na} = \vec{v}_{a,voor} + \frac{j}{m_a} \cdot \hat{n} \tag{28}$$

$$\vec{v}_{b,na} = \vec{v}_{b,voor} + \frac{j}{m_b} \cdot \hat{n} \tag{29}$$

Als we ook draaiing willen hebben, moeten we bedenken dat

$$\omega_{x,na} = \omega_{x,voor} + \frac{\vec{r}_x \cdot j\hat{n}}{I_x} \tag{30}$$

Dus dan komen we uit op de volledige vergelijking:

$$j = \frac{-(1+e)((\vec{v}_a - \vec{v}_b) \cdot \hat{n})}{\hat{n} \cdot \hat{n}(1/m_a + 1/m_b) + (\vec{r}_a \cdot \hat{n})^2/I_a + (\vec{r}_b \cdot \hat{n})^2/I_b} \tag{31}$$

Waarbij e de som van de restitutiecoëfficienten is en $\hat{n}$ de normaalvector van het contactoppervlakte.
En stel we willen pacman laten draaien door een botsing, maar zoals al eerder besproken is hij opgedeeld in 2 convexe helften. Wat dan? We berekenen dan voor elke helft apart het traagheidsmoment rond de as van rotatie door hun eigen massamiddelpunten, dan gebruiken we de stelling van Steiner om alle assen naar het massamiddelpunt van pacman (het gemiddelde massamiddelpunt van alle delen gewogen met hun massa) te brengen, waarop we gewoon alle traagheidsmomenten bij elkaar optellen. We rekenen dan verder met dit opgesomde traagheidsmoment, en het massamiddelpunt van het geheel.

Notitie: dit werkt overigens ook voor complexe constructies, zoals tandwielen die op elkaar inhaken: je neemt het traagheidsmoment van alle tandwielen, transleert ze zodat ze door het midden gaan van de tandwiel die wordt gedraaid, en telt de momenten bij elkaar op.

6.2 In 3d

Onze volledige vergelijking in 3d wordt:

$$j = \frac{-(1+e) \cdot (\vec{v}_a + (\vec{\omega}_a \times \vec{r}_a) - \vec{v}_b - (\vec{\omega}_b \times \vec{r}_b)) \cdot \hat{n}}{1/m_a + 1/m_b + (I_a^{-1} \cdot (\vec{r}_a \times \hat{n}) \times \vec{r}_a + I_b^{-1} \cdot (\vec{r}_b \times \hat{n}) \times \vec{r}_b) \cdot \vec{n}} \tag{32}$$

In het geval dat **b** onbeweegelijk is, bijvoorbeeld een muur, dan kunnen we, omdat dan zo'n beetje de helft 0 wordt, de vergelijking versimpelen tot

$$j = \frac{-(1+e) \cdot (\vec{v}_a + (\vec{\omega}_a \times \vec{r}_a)) \cdot \hat{n}}{1/m_a + (I_a^{-1} \cdot (\vec{r}_a \times \vec{n}) \times \vec{r}_a) \cdot \hat{n}} \tag{33}$$

6.3 Luchtweerstand

Helaas beschouw ik deze factor niet in dit werkstuk, want aerodynamische simulaties zijn een veld op zich. In principe moet het mogelijk zijn om een windrichting uit te kiezen, en dan een voorwerp te projecteren op het vlak met die windrichting als normaalvector. Dan kun je dat oppervlakte vervolgens gebruiken in de luchtweerstandformule, en deze kracht elke tijdstap uitoefenen op het voorwerp:

$$F_w = \frac{1}{2}\rho C_D A c v^2 \tag{34}$$

Waarbij C_D het drag-coëfficient is; dit kun je eigenlijk alleen via metingen te weten komen, en bevat de effecten van onder andere de vorm van het voorwerp en hoe soepel het oppervlakte van het voorwerp langs de lucht kan bewegen. $\vec{v}$ is de snelheid van het voorwerp, A is het oppervlakte van het voorwerp in de richting van $\vec{v}$, en ρ de luchtdichtheid.

6.4 Wrijving

Dit onderdeel is complexer dan het op het eerste gezicht lijkt, zie ook het werk van David E. Stewart: [21]. Alsnog kunnen we een redelijke approximatie maken met zogeheten *Coulombwrijving*[22]: we nemen de snelheid van het voorwerp en halen er een beetje van af volgens een coëfficient c_f in de richtingsvector $\hat{f}$, een projectie van de snelheidsvector $\vec{v}$ op het vlak van aanraking. ($\hat{f}$ niet te verwarren met kracht $\vec{F}$) Dit pluggen we in in onze krachtstootvergelijking om j_w, de wrijvingskrachtstoot te krijgen. Die schalen we ook nog eens met hoe hard de voorwerpen tegen elkaar aandrukken, door de absolute waarde te nemen van het inproduct tussen de relatieve snelheid en de normaalvector van het aanrakingsvlak, $\vec{n}$.

$$\vec{a} = (\vec{v} \times \hat{n}) \times \hat{n} \tag{35}$$

$$\vec{f} = \vec{a} \cdot (\vec{v} \cdot \vec{a}) \tag{36}$$

Waarbij $\vec{n}$ de normaalvector van het manifold is, en $\vec{v}_{rel}$ de relatieve snelheid van de twee voorwerpen *op het punt van contact, $\vec{p}_c$* (dus met draaisnelheid erbij gerekend):

$$\vec{v}_{a,pc} = \vec{v}_a + \vec{\omega}_a \times \vec{r}_a \tag{37}$$

$$\vec{v}_{b,pc} = \vec{v}_b + \vec{\omega}_b \times \vec{r}_b \tag{38}$$

$$\vec{v}_{rel} = \vec{v}_{a,pc} - \vec{v}_{b,pc} \tag{39}$$

Waarbij $r_x = \overrightarrow{mmp}_x - \vec{p}_c$, waarbij $\overrightarrow{mmp}_x$ het massamiddelpunt van voorwerp x is. Met dit alles kunnen we de grootte van de krachtstoot berekenen (richting volgt later):

$$j_w = \frac{-(1 + c_{f,res}) \cdot (\vec{v_a} + (\vec{\omega}_a \times \vec{r}_a) - \vec{v}_b - (\vec{\omega}_b \times \vec{r}_b)) \cdot \vec{f}}{1/m_a + 1/m_b + (I_a^{-1} \cdot (\vec{r}_a \times \vec{f}) \times \vec{r}_a + I_b^{-1} \cdot (\vec{r}_b \times \vec{f}) \times \vec{r}_b) \cdot \vec{f}} \tag{40}$$

$(\vec{v}_a + (\vec{\omega}_a \times \vec{r}_a) - \vec{v}_b - (\vec{\omega}_b \times \vec{r}_b))$ zou ook nog vervangen kunnen worden door de net berekende $\vec{v}$ omdat het hetzelfde is.
$c_{f,res}$ is het product van de wrijvingscoëfficienten van de twee voorwerpen a en b en ligt in $[0,1]$; een waarde van 0 wil zeggen dat het perfect glad is, en een waarde van 1 wil zeggen dat er 100% wrijving is.

6.5 Nieuwe snelheid en rotatie

Nu moeten we alleen nog j vermenigvuldigen met de normaalvector van de botsing, $\hat{n}$ zodat het een richting krijgt, dan kunnen we de nieuwe (draai)snelheden berekenen:

$$\vec{v}_{a,na} = \vec{v}_{a,voor} + (j \cdot \hat{n})/m_a \tag{41}$$

$$\vec{v}_{b,na} = \vec{v}_{b,voor} - (j \cdot \hat{n})/m_b \tag{42}$$

$$\vec{\omega}_{a,na} = \vec{\omega}_{a,voor} + \vec{r}_a \times (j \cdot \hat{n}) \cdot I_a^{-1} \tag{43}$$

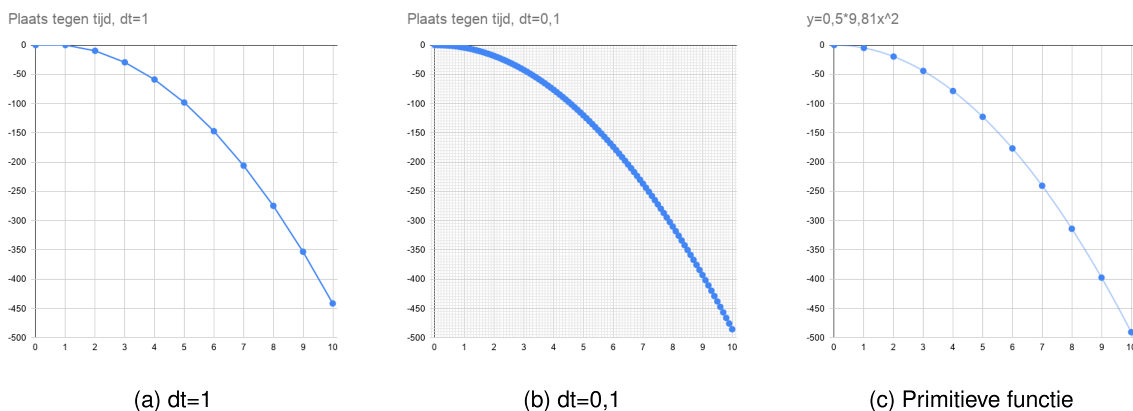
$$\vec{\omega}_{b,na} = \vec{\omega}_{b,voor} - \vec{r}_b \times (j \cdot \hat{n}) \cdot I_b^{-1} \tag{44}$$

Als je ook wrijving wilt hebben, dan moet je j_w volgens de bovenstaande 4 vergelijkingen apart toepassen. Let op dat je dan de richting $\vec{f}$ gebruikt in plaats van $\hat{n}$.

Let op dat Coulombwrijving niet oplost voor puntfrictie: dat wil zeggen, een tollend voorwerp zal voor altijd blijven tollen met dit wrijvingsmodel (opbouwende numerieke fouten daargelaten), omdat er geen *laterale* beweging is over het manifold met een tol die op zijn plaats blijft en alleen tolt. Het artikel waar ik naar verwees dient als oplossing hiervoor. Deze limitatie van Coulombwrijving is overigens mogelijk ook de reden dat Cobb in de film Inception een tol gebruikt om te bepalen of hij in de echte realiteit zit of een droom (simulatie).

7 Numeriek Integreren

Om bijvoorbeeld de plaats van een voorwerp over tijd te simuleren doe je $x_{nieuw} = x_{oud} + v \cdot dt$: zo integreer je de snelheid om zo de plaats te krijgen. Dit heet de Eulermethode, niet te verwarren met een "Euler integraal" wat naar een vrij specifieke en voor onze doeleinden ongerelateerde formule refereert. De Eulermethode werkt in theorie, met $dt = (heelklein)$. Maar hoe groter de dt, hoe meer de berekening af gaat wijken. Dit kan zeker een probleem zijn: De plaats-tijd grafiek van een voorwerp dat met $9.81 m/s^2$ naar beneden versnelt ziet er vrij anders uit, afhankelijk van de tijdstap (let op de scherpe hoeken bij dt=1, en ook de eindposities):



(a) dt=1 (b) dt=0,1 (c) Primitieve functie

Figuur 9: Een grote tijdstap, een kleine tijdstap en de primitieve functie ($y = \frac{1}{2} \cdot -9.81x^2$) met elkaar vergeleken.

Met een zogeheten Taylorreeks valt er veel te zeggen over de numerieke afwijkingen en stabiliteiten van bepaalde integratietechnieken, maar dat is ver buiten de strekking van dit artikel.
In het algemeen itereer je over de staat van een simulatie. Dat wil zeggen, je begint met een beginstaat $i = 0$, en gaat stapje voor stapje simuleren, $i = i + 1$. In programmeren wordt vaak de variabele i gebruikt om het aantal iteraties bij te houden.

7.1 Eulermethode

De makkelijkste integratiemethode, maar ook de meest gevaarlijke. Uit eigen ervaring weet ik dat deze methode gevoelig is voor "ontploffingen", waar de eigenschappen van de voorwerpen ineens een steeds grotere fout krijgen totdat alles uit elkaar vliegt.

$$\vec{a}_{i+1} = \vec{g} + \vec{F}_{res,i}/m \tag{45}$$

$$\vec{v}_{i+1} = \vec{v}_i + \vec{a} \cdot dt \tag{46}$$

$$\vec{s}_{i+1} = \vec{s}_i + \vec{v} \cdot dt \tag{47}$$

Of in code (dit is "forward Euler"):

```
do {
    a = g + F_res/m;
    v = v + a*dt;
    s = s + v*dt;
    t = t + dt;
} while (t < 10);
```

Als je de volgorde van de bovenstaande code verandert kan het integraal ineens een stuk onstabieler worden.

7.2 Verletintegraal

Het oorspronkelijke Verletintegraal is bedacht voor het berekenen van projectieltrajecten, en blinkt hier dan ook in uit. Dit integraal is numerisch veel stabieler dan de Eulermethode, maar nog steeds vrij simpel. Het integreert echter achteruit: in plaats van met een staat i verder te werken naar staat $i+1$, werkt het terug van staat i naar staat $i-1$. Dit is mooi voor wiskundigen maar onbruikbaar voor computersimulaties dus is Velocity Verlet [23] bedacht. Als je echter de formule ombouwt zodat je $i+1$ berekent uit i, krijg je de volgende code:

```
do {
    a = g + F_res/m;
    s = s + v*dt + 0.5*a*dt*dt;
    v = v + a*dt;
    t = t + dt;
} while (t < 10);
```

Trivia: het Verletintegraal is heel vaak onafhankelijk uitgevonden en heeft dus niet echt één auteur.

7.3 Velocity-Verletintegraal

Zie [24].

$$\vec{s}_{i+1} = \vec{s}_i + \vec{v}_i \cdot dt + \frac{1}{2}\vec{a}_i \cdot (dt)^2 \tag{48}$$

$$\vec{v}_{i+1} = \vec{v}_i + \frac{1}{2}(\vec{a}_{i+1} + \vec{a}_i) \cdot dt \tag{49}$$

```
do {
    s_new = x + v*dt + a*(dt*dt*0.5);
    a_new = g + F_res/m;
    v_new = v + (a+a_new)*(dt*0.5);
    s = s_new;
    v = v_new;
    a = a_new;
    t = t + dt;
} while (t < 10);
```

7.4 Runge-Kutta

Runge-Kutta beschrijft een familie van numerieke integralen van een bepaalde“orde”. Of preciezer gezegd, RK formuleert een numerieke oplossing voor een gegeven differentiaalvergelijking (in de vorm van een discrete reeks getallen). Euler’s methode is Runge-Kutta van de eerste orde. Unreal Engine 4 en Unity gebruiken beide Runge-Kutta 4, vaak afgekort tot “RK4”. In Runge-Kutta wordt de discrete tijdstap niet met dt maar met h genoteerd.
Een voorbeeld van RK4: Hierin is x een onbekende functie van tijd t, dit zou bijvoorbeeld $\vec{v}(t)$ kunnen zijn. De afgeleide van x is een functie van t en x zelf; $f(t,x)$.

$$\frac{dx}{dt} = f(t,x) \tag{50}$$

$$t_{i+1} = t_i + h \tag{51}$$

$$x_{i+1} = x_i + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \tag{52}$$

Met de k’s:

$$k_1 = hf(t_i, x_i) \tag{53}$$

$$k_2 = hf(t_i + \frac{h}{2}, x_i + \frac{k_1}{2}) \tag{54}$$

$$k_3 = hf(t_i + \frac{h}{2}, x_i + \frac{k_2}{2}) \tag{55}$$

$$k_4 = hf(t_i + h, x_i + k_3) \tag{56}$$

Zie ook [25] en [26]. Omdat de volgorde van RK-4 niet heel strict is gespecificeerd zijn er meerdere geldige algoritmes te bedenken om met RK4 te integreren. Hier is één voorbeeld in C die de differentiaalvergelijking $\frac{dy}{dx} = 1 + y^2$ oplost, hierbij is t onze x-as en x is onze y-as (ja):

```
#define F(x,y)  (1 + (y)*(y)) /* dy/dx = 1 + y^2 */
void integrate(double x_min, double x_max, double h)
{
    double y=0,k1,k2,k3,k4;

    for(double x = x_min; x <= x_max; x = x+h)
    {
        k1 = h * F(x, y);
        k2 = h * F(x+h/2, y+k1/2);
        k3 = h * F(x+h/2, y+k2/2);
        k4 = h * F(x+h, y+k2);
        y = y + ( k1 + 2*k2 + 2*k3 + k4)/6;

        printf("%f: %f \n", x, y);
    }
}
```

Het nadeel van RK4 is dat alle formules moeten worden omschrijven om in de notatie van een $\frac{dx}{dt} = f(t,x)$ differentiaalvergelijking te passen. In dit opzicht is een Verlet-integraal handiger. Voor de versnelling, snelheid en plaats is RK4 bekend: zoekterm "RK4 for 2nd order ordinary differential equation".

8 Numerieke imperfecties

Alhoewel computers letterlijk voor *computaties* bedacht zijn, blijkt dat in de praktijk niet altijd helemaal goed te gaan. Er zijn meerdere gebieden waar onze simulatie de mist in kan gaan:

8.1 Limieten in data

In de natuurkundeboeken wordt alles weggestopt in een abstracte letter, een variabele. Dit doe je ook op een computer in code. Maar op een computer kun je maar een eindige hoeveelheid data kwijt; Een floating point number is verruit het meest-gebruikte datatype voor kommagetallen, en is een binaire versie van wetenschappelijke notatie: $getal \cdot 2^{macht}$. Je hebt 23 bits (impliciet 24 bits, omdat het getal altijd *genormaliseerd* is, wat overeen komt met een decimale significantie van ongeveer 7.22 sig. cijfers) om het getal op te slaan, en 8 bits om de macht op te slaan. De overige bit is om aan te geven of het getal positief of negatief is. [27]

Dit is een probleem bij grote simulaties (groot als in: het gebied van simulatie beslaat een groot volume): je hebt niet x aantal cijfers achter de komma, maar x aantal significante cijfers. Dat wil zeggen, hoe groter je getal, hoe minder cijfers achter de komma. Je verliest dus precisie, dit resulteert in een afwijking, en deze afwijkingen kunnen makkelijk opbouwen omdat je ermee verder rekent (je moet wel; waarmee anders?). De getallen die horen bij voorwerpen die ver weg zijn, met name wereldpositie, kunnen dus afwijken van wat ze eigenlijk horen te zijn.

Je kan 64-bit floating points gebruiken, maar dit verplaatst het probleem alleen maar: je krijgt alsnog steeds grotere afwijkingen bij grotere getallen.

8.2 Onrepresenteerbare getallen

Een ander probleem is het feit dat sommige getallen gewoon niet binair uit te drukken zijn: een beroemd voorbeeld is $0.2 + 0.1 = 0.30000000000000004$ (64-bit floating point). Met 32-bit floating points geldt zelfs $0.2 + 0.1 = 0.30000001$. Hoe kan dit? Dezelfde reden dat $1/3 = 0.3333333333333$...: je kan sommige getallen gewoon niet perfect noteren in sommige *radices*. Voor radix 10 is een voorbeeld dus $1/3$, en voor radix 2 is dat $3/10$. Nogmaals kun je het probleem verzachten door **doubles** (64-bit) in plaats van 32-bit **float**s te

gebruiken, maar verhelpen kun je het nooit. Sommige talen bieden ook ondersteuning voor een **decimal** of **bignum** type, waarin getallen worden gerepresenteerd als een (zeer lange) reeks decimale cijfers in plaats van binair. Dit is natuurlijk suboptimaal en zal tot trage simulaties leiden, maar het geeft wel meer precisie. Alsnog kan $1/3$ niet eindig gerepresenteerd worden hiermee.

8.3 Tunneling

Elke keer als je in de simulatie een tijdstap laat verlopen, teleporteren de voorwerpen in zekere mate naar hun volgende positie. Stel: een auto van 3,6m lang gaat 72m/s (260km/h) op de Autobahn, en elke 0.1s wordt de simulatie bijgewerkt: de auto teleporteert dan eigenlijk over het wegdek in stappen van 6 meter. En stel, er is een andere auto (ook 3.6 meter lang) met pech, die dom stil staat midden op de weg. Botsen de auto's? Nee! Want in één tijdstap kan de bewegende auto "door" de stilstaande auto heen teleporteren (wel krapjes, met 0m speelruimte), ook al hadden ze eigenlijk moeten botsen. Dit probleem wordt ook wel tunneling genoemd. Hoe groter het snelheidsverschil en hoe dunner de voorwerpen en hoe groter de tijstap hoe sneller je tunneling krijgt. Een minder gekunsteld voorbeeld is dat van een kogel die door een dunne gipsmuur tunnelt.

In games draaien simulaties vaak op 60Hz (tijdstap van 16.666...ms), dus is dit probleem een stuk minder uitgesproken; als nog bestaat het fenomeen.

In theorie is dit perfect op te lossen door voor elk vlak van beide voorwerpen A en B de positie van elk vlak van voorwerp A als functie van de tijd gelijk te stellen met elk vlak van voorwerp B, en dan de simulatie door te laten lopen tot de tijd(en) die daar uit komt/komen; maar in de praktijk wordt dit niet vaak gedaan omdat het veel rekenwerk is voor de computer en dus resulteert in een trage simulatie. Maar het *kan* wel.

Een andere oplossing is een kleine regel als "neem plaatselijk wat kleinere tijdstappen als er twee voorwerpen A en B zijn, die met hun huidige afmetingen en snelheden door elkaar heen zouden kunnen teleporteren". Ook dit is niet perfect, en het komt ook met extra kosten qua rekentijd.

9 Eigen werk

Ik heb een 2d en 3d deeltjessimulatie gemaakt. De 3d versie gebruikt Velocity Verlet, de 2d versie gebruikt Euler (die dan ook altijd vroeger of later ontploft). De code is als volgt te vinden:

- 2d: https://www.michaelroeleveld.nl/simulaties/deeltjes.html
- 3d: https://gitlab.com/MichaelRoeleveld/illumina

Voor de 2d versie, druk op "element inspecteren" om de code te zien. Voor de 3d versie, zie *src/physics.h*.

10 Conclusie

"Botsingen simuleren" is een heel breed onderwerp. Behalve Rigid Body Dynamics (waarvan ik slechts een deel heb kunnen behandelen in dit werkstuk) heb je ook nog eens softbodies, vloeistof- en gassimulaties, en raytracing (de botsing van licht). Alleen het veld van Rigid Body Dynamics al is zeer breed en diep, maar zeker interessant, en persoonlijk ben ik er nog niet klaar mee. Het maken van dit profielwerkstuk heeft mij een dieper begrip gegeven voor de natuurkunde, gepaard met een diepere appreciatie. Ik snap nu ook waarom natuurkunde in games vaak zo *glitchy* is; het is allemaal nog behoorlijk lastig.

11 Evaluatie

Uiteindelijk had ik meer tijd kunnen spenderen aan het schrijven van simulaties in plaats van het onderzoeken ervan, maar ik heb het op theoretisch vlak nu in ieder geval redelijk uitgewerkt. Ik vond het PWS als geheel leuk; maar soms wat omslachtig, bijvoorbeeld dat je een logboek bij moet houden, etc. Maar ach, dat hoort er bij, denk ik dan maar.

12 Bronnen

[1] D. Baraff. “Analytical Methods for Dynamic Simulation of Non-Penetrating Rigid Bodies”. In: *SIGGRAPH Comput. Graph.* 23.3 (jul 1989), p. 223–232. ISSN: 0097-8930. DOI: 10.1145/74334.74356. URL: https://doi.org/10.1145/74334.74356.

[2] David Baraff en Andrew Witkin. “Dynamic Simulation of Non-Penetrating Flexible Bodies”. In: *SIGGRAPH Comput. Graph.* 26.2 (jul 1992), p. 303–308. ISSN: 0097-8930. DOI: 10.1145/142920.134084. URL: https://doi.org/10.1145/142920.134084.

[3] Yuanming Hu e.a. “A Moving Least Squares Material Point Method with Displacement Discontinuity and Two-Way Rigid Body Coupling”. In: *ACM Transactions on Graphics (TOG)* 37.4 (2018), p. 150.

[4] Lucas V. Schuermann. *Particle-Based Fluid Simulation with SPH*. 2016. URL: https://lucasschuermann.com/writing/particle-based-fluid-simulation.

[5] Hang Si. *TETGEN A Quality Tetrahedral Mesh Generator and 3d Delaynay Triangulator*. 2012. URL: https://people.sc.fsu.edu/~jburkardt/examples/tetgen/tetgen.html.

[6] Yixin Hu e.a. “Fast Tetrahedral Meshing in the Wild”. In: *ACM Trans. Graph.* 39.4 (jul 2020). ISSN: 0730-0301. DOI: 10.1145/3386569.3392385. URL: https://doi.org/10.1145/3386569.3392385.

[7] Dr. Robert Nürnberg. *Calculating the volume and centroid of a polyhedron*. 2013. URL: http://wwwf.imperial.ac.uk/~rn/centroid.pdf.

[8] GuerrillaCG. *Euler (gimbal lock) Explained*. 2019. URL: https://www.youtube.com/watch?v=zc8b2Jo7mno.

[9] Brian Moore. *3D Rotation*. 2017. URL: https://robosub.eecs.wsu.edu/wiki/cs/localization/rotation/start.

[10] Grant Sanderson. *Visualizing quaternions: An explorable video series*. 2019. URL: https://eater.net/quaternions/.

[11] D. Rose. *Rotation Quaternions and How to Use Them*. 2015. URL: http://danceswithcode.net/engineeringnotes/quaternions/quaternions.html.

[12] Chris Hecker. *Physics, Part 4:The Third Dimension*. 1997. URL: http://www.chrishecker.com/images/b/bb/Gdmphys4.pdf.

[13] Amir Vaxman. “Lecture 3: Rigid Body Physics”. University Lecture. 2017. URL: http://www.cs.uu.nl/docs/vakken/mgp/2017-2018/Lecture%203%20-%20Rigid-Body%20Physics.pdf.

[14] E. G. Gilbert, D. W. Johnson en S. S. Keerthi. “A fast procedure for computing the distance between complex objects in three-dimensional space”. In: *IEEE Journal on Robotics and Automation* 4.2 (1988), p. 193–203. DOI: 10.1109/56.2083.

[15] Gino Van den Bergen. “A Fast and Robust GJK Implementation for Collision Detection of Convex Objects”. In: *Journal of Graphics Tools* 4.2 (1999), p. 7–25. DOI: 10.1080/10867651.1999.10487502. eprint: https://doi.org/10.1080/10867651.1999.10487502. URL: https://doi.org/10.1080/10867651.1999.10487502.

[16] Winterdev. *GJK Algorithm Explanation and Implementation*. 2020. URL: https://www.youtube.com/watch?v=MDusDn8oTSE.

[17] William Bittle. *SAT (Separating Axis Theorem)*. 2010. URL: http://www.dyn4j.org/2010/01/sat/.

[18] Eran Guendelman, Robert Bridson en Ronald Fedkiw. “Nonconvex Rigid Bodies with Stacking”. In: *ACM Trans. Graph.* 22.3 (jul 2003), p. 871–878. ISSN: 0730-0301. DOI: 10.1145/882262.882358. URL: https://doi.org/10.1145/882262.882358.

[19] Mozilla Developer Network. *3D collision detection: Axis-aligned bounding boxes*. 2019. URL: https://developer.mozilla.org/en-US/docs/Games/Techniques/3D_collision_detection.

[20] William Bittle. *Contact Points Using Clipping*. 2011. URL: http://www.dyn4j.org/2011/11/contact-points-using-clipping/.

[21] David E. Stewart. “Rigid-Body Dynamics with Friction and Impact”. In: *SIAM Review* 42.1 (2000), p. 3–39. DOI: 10.1137/S0036144599360110. eprint: https://doi.org/10.1137/S0036144599360110. URL: https://doi.org/10.1137/S0036144599360110.

[22] Glenn Fiedler. *Collision Response and Coulomb Friction*. 2013. URL: https://gafferongames.com/post/collision_response_and_coulomb_friction/.

[23] William C.; H. C. Andersen; P. H. Berens; K. R. Wilson Swope. “A computer simulation method for the calculation of equilibrium constants for the formation of physical clusters of molecules: Application to small water clusters”. In: *The Journal of Chemical Physics* (1982), p. 637–649.

[24] Prof. Branislav K. Nikolic. *Verlet Method*. URL: http://www.physics.udel.edu/~bnikolic/teaching/phys660/numerical_ode/node5.html.

[25] Erik Cheever. *Fourth Order Runge-Kutta*. 2005. URL: https://lpsa.swarthmore.edu/NumInt/NumIntFourth.html.

[26] Autar Kaw. *Runge-Kutta 4th Order Method for Ordinary Differential Equations*. 2010. URL: https://mathforcollege.com/nm/mws/gen/08ode/mws_gen_ode_txt_runge4th.pdf.

[27] IEEE Computer Society. *IEEE 754*. 2008.