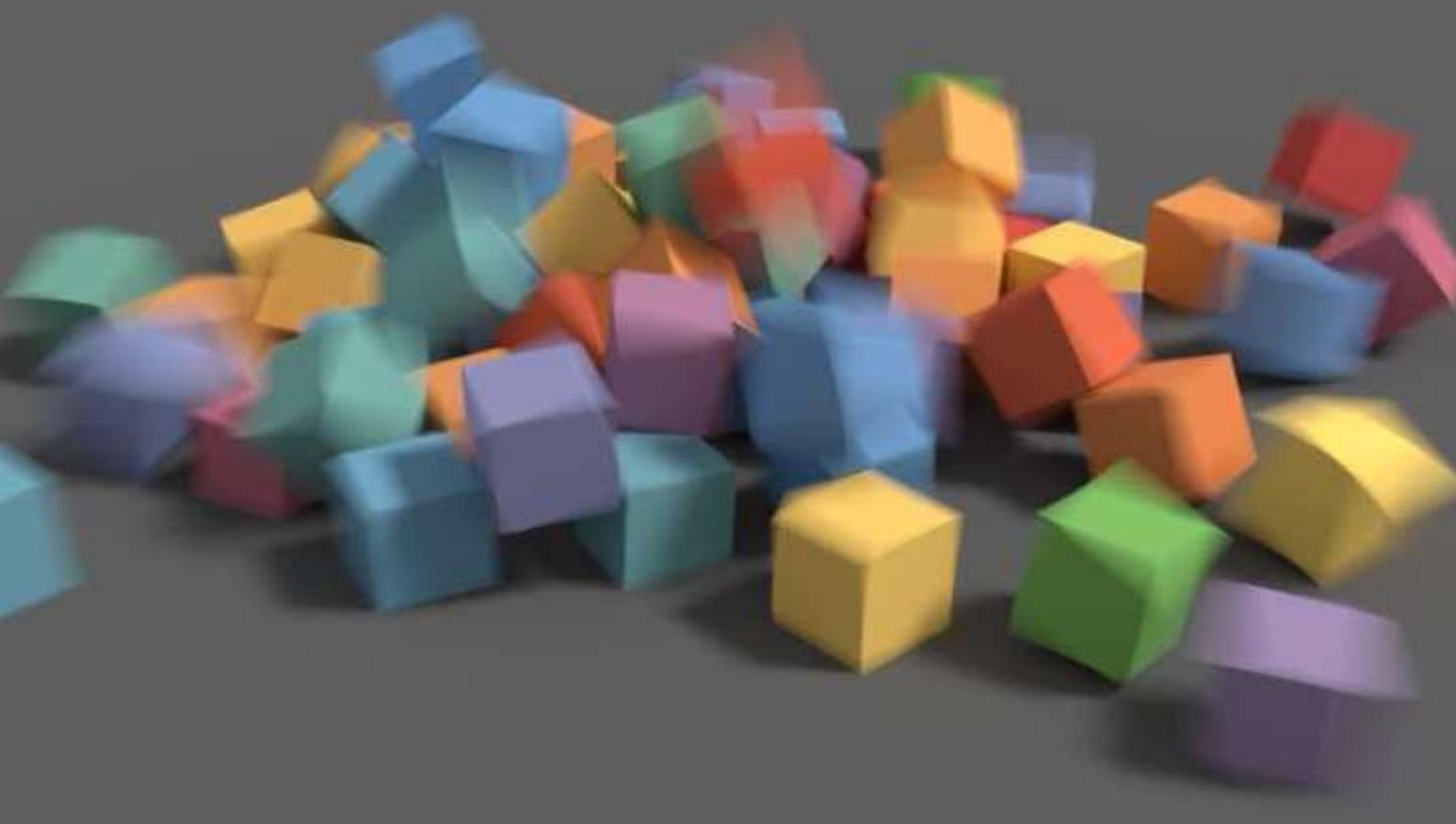


17-12-21

Nicklas Sirius Østerberg

SOP – Programmering B & Fysik A

## Simulation af Roterende Objekter



## Resume

Denne opgave dækker hvad inertimoment er og hvordan det er forskelligt for inertি. Den går ind på hvordan man regner på inertmoment og hvilke formler der giver mening at bruge. Så går inkluderer den noget af den programmerings-faglige metode, computational thinking, og går derefter i dybden med hvordan denne simulation er lavet ved at forklare nogle dele af koden som er relevant. Den indeholder eksperiment opstilling til at kunne bevise simulationens korrekthed og forklarer om det eksperiments resultater og validiteten af de resultater. Derefter vil der blive sammenlignet resultater fra simulatoren og den virkelige verden så man kan vurdere præcisionen af sådan en simulator. Der bliver også konkluderet på om sådan en simulator er praktisk at bruge i den virkelige verden til at forudse hvordan andre objekter kommer til at reagere på forskellige situationer.

## Indholdsfortegnelse

Indledning .....	3
Inertimoment .....	4
Programmering.....	4
Computational thinking.....	5
Simulation.....	6
Fremgangsmåde til eksperiment.....	7
Sammenligning/test .....	8
Fejlkilder .....	8
Konklusion .....	9
Litteraturliste .....	10
Bilag .....	11

## Indledning

Rotationer er overalt omkring os. Men hvad er det der bestemmer hvor meget et objekt vil rotere? Hvad bruger industrier til at udregne rotations kræfter? Og hvad gør man hvis man har en kraft der hele tiden ændrer sig? Dette kommer denne opgave ind på og den laver et eksempel program til at simulere sådanne kræfter og bevægelser. Målet med denne opgave vil være at:

- Redegør for begrebet inertimoment og udled formlerne for udvalgte rotationssymmetriske objekter.
- Analysér hvorledes man kan simulere og visualisere rotation af fysiske objekter på en computer med inddragelse af inertimomentet. Implementér en simpel prototype i et selvvalgt programmeringssprog.
- Lav en eksperimentel undersøgelse af inertimomentet for forskellige rotationssymmetriske objekter på baggrund af en grundig beskrivelse af den indgående teori.
- Sammenlign din prototypes resultater med den eksperimentelle undersøgelses resultater med særligt fokus på hvor god prototypen er til at forudsige inertimomentet.
- Diskuter muligheder og begrænsninger i modellens simulation af det roterende objekt.

## Inertimoment

I fysik kender vi begrebet inertimoment som er et elements modstand mod at påbegynde en rotation. Altså hvis vi har et objekt som er stille på en akse og vi så begynder at rottere det objekt, så vil det kræve noget energi fra os. Men forskellige objekter kommer til at kræve forskellig mængder energi for at kunne komme op på samme rotationshastighed. Derfor bruger vi inertimoment til at beskrive hvor meget energi man skal bruge for at lave en vis rotationsændring.

Det er let nok at regne ud at inertimomentet afhænger af vægten af objektet siden at meget tunge objekter er sværere at rottere end lette objekter og uddover det er distancen mellem massen og rotationsaksen også en del af inertimomentet. Da inertimomentet er bestemt ud fra både masse og massens position er vi nødt til at dele emnet op i flere dele. Hver del har en masse og en distance til rotationsaksen og hver del skal summeres sammen så formlen vi skal bruge til at finde inertimomentet, ser sådan ud

$$I = m_1 \cdot r_1^2 + m_2 \cdot r_2^2 + m_3 \cdot r_3^2 + \dots$$

Dette er dog ikke en særlig god måde at udregne inertimomentet da det kræver en masse udregninger hvis man har et meget stort objekt og man gerne vil have et præcist resultat. Derfor er der lavet tabeller med lettere måder at udregne inertimomentet for typiske former.

Nogle af dem ser sådan ud:

$$I = 1/12 * m * r^2 \quad \text{stang med rotationsakse i enden}$$

$$I = 1/3 * m * r^2 \quad \text{stang med rotationsakse i midten}$$

Dem vil jeg teste senere i opgaven med min simulation.

## Programmering

I programmering laver vi et sæt instruktioner som kan læses og udføres af noget hardware, som fx en computer. Når man laver et program, bruger man programmeringssprog som er lettere for mennesker at forstå og skrive, men ikke alle programmeringssprog er lige hurtige til at udføre opgaver. Om programmeringssprog bruger man udtrykkene "high-level" og "low-level" til at beskrive hvor maskinnære de er og det hænger typisk også sammen med hvor hurtigt de kører. Grunden til at man ikke bare bruger low level sprog til alle sine programmer er at det er mere kompliceret at lave programmer med dem. I stedet bruger man high level programmeringssprog der er lettere at skrive som kompileres til maskin-kode som computeren kører.

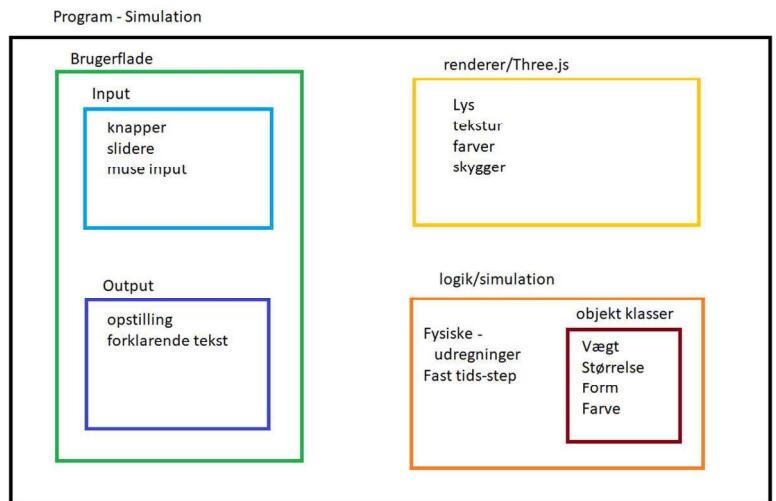
I dette projekt har jeg valgt at bruge JavaScript som er et meget high-level sprog, mest fordi det er det jeg har arbejdet i for tiden, men det har også de fordele at det er let at køre på andre

computere da de ikke skal installere noget for at kunne køre det, men blot gå ind på nicklasbns.me/sop for at køre koden.

## Computational thinking

Når man har et problem/opgave i programmering er det meget godt at dele det op i dele som man skal lave. Fx da jeg besluttede at jeg skulle lave en simulation begyndte jeg først at overveje hvad jeg skulle lave den i, altså hvilket programmeringssprog jeg skulle bruge. En af de ting jeg tænkte var gdScript som bruges til at kode i programmet godot (udtales go-dou), fordi det har indbygget funktioner til at udregne kræfter, kollisioner og vægte så det ville gøre det ganske let at "lave" en simulation ud fra det, men det valgte jeg at undgå fordi jeg hellere ville gå lidt dybere i opgaven og lave udregninger selv. Java og python var også programmeringssprog som jeg overvejede, men valgte at bruge javascript, mest fordi det er det jeg har arbejdet mest i for tiden og det ville virke fint til dette formål.

Så før jeg begyndte at skrive mit program, gik jeg i gang med at overveje de dele af programmet jeg skulle have lavet. Derfor lavede jeg et diagram over hvordan min simulation skulle se ud. Dette diagram er inspireret af c4 diagrammet som bruges til at visualisere hvordan programmerne burde virke og hvilke komponenter af programmet der kommunikerer med hvad. Grunden til det kun er inspireret af c4 modellen og ikke et rigtigt c4 diagram er at et c4 diagram er designet til større systemer med serverer der kommunikerer med hinanden, som jeg ikke har i min simulation her.



Overordnet set har vi hele min simulation. Den simulation består så af en brugerflade, en renderer, og en logik ting. Som I kan se, er selve simuleringen ikke en særlig stor del af at lave sådan et program. Brugerfladen har jeg valgt at dele op i to dele, input og output, input er alle knapper og tekstmærker og museklik. De er vigtige for programmet for at kunne fortælle hvad man gerne vil have programmet skal vise. Input mærker sender kun data til logik delen af programmet. Output er det der bliver vist til os på skærmen, som fx selve 3d objekterne, teksten der forklarer hvad de forskellige inputs gør, og ikoner der viser status af programmet. Rendereren er den der "tegner" alle de dele som simulatoren udregner. Til den har jeg valgt at bruge biblioteket three.js fordi det gør det dejlig let at lave visuelle 3d scener.

## Simulation

Jeg har lavet et simulations environment som fokuserer på rotation, kraftmoment, og inertimoment. Siden det er en simulation jeg laver, ville jeg undgå at bruge nogle biblioteker eller programmer som har fysik simulationer indbygget da det lidt ville ødelægge opgaven. Det jeg har brugt af biblioteker, er et 3d visualiserings bibliotek som hedder three.js og udnytter browseres WebGL engine.

Figur 2 er et udsnit af min HTML kode som viser selve hjemmesidens struktur. Min HTML kode er meget simpel fordi det meste af det der skal vises på hjemmesiden bliver lavet i JS. Vi kan se at der både på første linje og sidste linje står "body" (tekst med blå). Det er fordi vi kigger i body elementet på siden så først har vi et åbnende body element og til sidst et lukkende body element som indikerer at alt der sker mellem de to elementer skal være inde i det body element. Det første element vi har inde i body elementet er et canvas element. En canvas, eller på dansk "lærred", er det man bruger hvis man gerne vil kunne "tegne" på hjemmesiden pixel for pixel, og det er også derfor det hedder lærred, da man tegner på det ligesom man tegner på et lærred. Dette canvas element har en masse egenskaber (det der står med lyseblåt og orange). Den første er id, der står for identifier (identifikator) og bruges til at kunne finde vores element senere i vores kode. De næste to er width og height (bredde og højde) og de styrer hvor stort lærredet er i antal pixels. Nu har jeg sat dem til at være 1920 \* 1080 fordi det er standard opløsning på skærme, men det betyder ikke at selve elementet kommer til at fyldе hele skærmen fordi det bliver skaleret lidt i style egenskaben. Style egenskaben har en masse tekst og det er ikke særlig vigtigt for forståelse af koden. Det den overordnet set gør er at sætte elementet i midten af skærmen og laver en kant rundt om på 50 pixels. Det næste element er et script element. Script elementer er det vi bruger til at køre JavaScript kode og det betyder også at de ikke bliver vist frem på skærmen selvom de er inde i body elementet. Et script har en kilde (src/source) som er en sti til en fil på ens computer eller i cloud. Den første her er det bibliotek som jeg bruger, så den fil hedder three.js. På samme linje er der også noget tekst med grøn farve. Det er en kommentar i koden, så det er ikke noget computeren læser og er kun brugt til når folk skal læse koden kan det gøre det lettere med kommentarer. Denne kommentar er et link til der hvor jeg har hentet filen. Det sidste element er selve mit script som styrer selve simulationen, det har jeg kaldt script1.js.

```
<body>
    <canvas id="can" width="1000" height="1000" style=
        "position: fixed; top: 50px; left: 50px; width: 1000px; height: 1000px;
    "></canvas>
    <script src="three.js"></script><!--https://threejs.org/build/three.js-->
    <script src="script1.js"></script>
</body>
```

Siden min JS kode er over 200 linjer lang giver det ikke mening at forklare hele koden linje for linje, så jeg vil vise nogle udsnit fra koden

Dette er den første del af scriptfilen. Det er en del af setup koden, så altså den bliver den kun kørt en gang når man åbner hjemmesiden. Pointen med setup kode er at skabe alle de variabler og klasser man skal bruge globalt, så man kan bruge dem i hele ens kode. Først, lige efter den korte kommentar, definerer vi canvas variablen som bliver sat til at være det canvas element vi lavede i html koden. Fra den canvas variabel laver vi to nye variabler, width og height, som bliver "udpakket" fra canvas. Så laver vi en scene, et kamera, en renderer, og en raycaster, som jeg bruger senere i koden. Og så bliver der sat nogle indstillinger som bliver styret af three js, som skygger, baggrundsfarve, og kameraposition.

```
//setup
const canvas = document.getElementById("can");
const {width, height} = canvas;
const can = new three.Scene();
const cam = new three.PerspectiveCamera(75, width/height);
const renderer = new three.WebGLRenderer({canvas});
const raycaster = new three.Raycaster();
renderer.shadowMap.enabled = true
can.background = new three.Color(0xa0a0a0);
cam.position.set(0, 5, 10);
cam.lookAt(0, 0, 0);
```

## Fremgangsmåde til eksperiment

Jeg skulle bruge en forsøgsopstilling som let kunne vise mig inertimoment for objekter. Jeg fandt så en opstilling som bestod af en holder til en stang som var monteret til en fjeder som ville trække stangen tilbage til sit origo. Man ville så på svingningstiden kunne bestemme inertimoment for objektet. Men for at kunne få nogle målinger fra forsøget var jeg nødt til at kende kraften i fjederen, så den målte jeg med et newton-meter ved et interval a 10 grader indtil jeg kom op på 2 newton som var den øvre grænse af det newtonmeter jeg brugte. Ved hjælp af lineær regression fandt jeg frem til en hældning på 0,022. Og det kunne jeg bruge med formlen for kraftmoment, hvor jeg havde en radius på 13,5 cm og en vinkel på 90 grader.

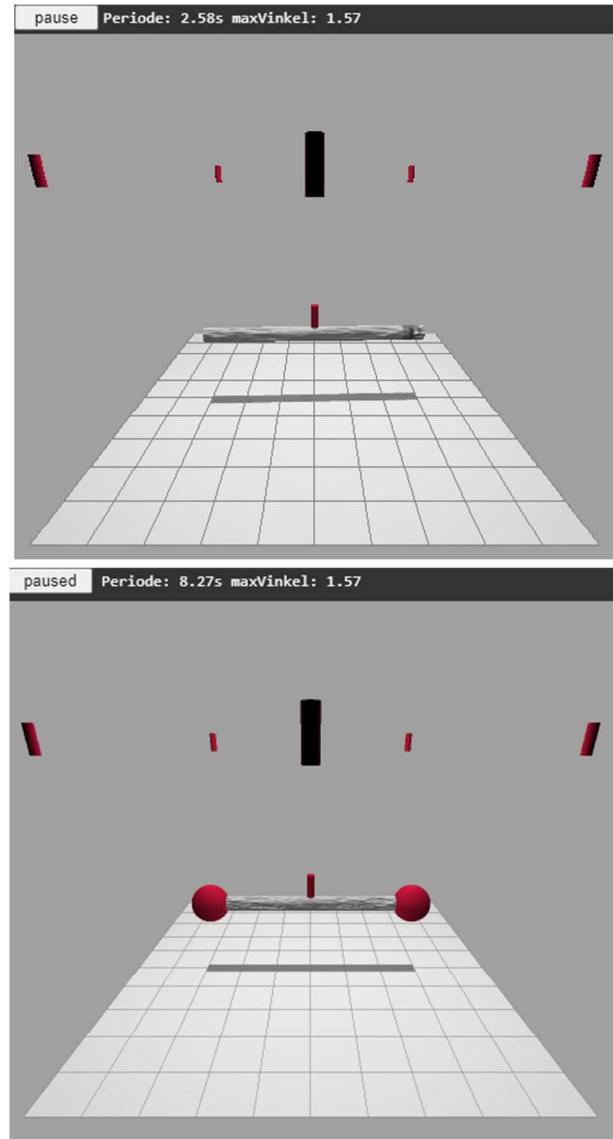
$$M = r \cdot F \sin(\theta)$$

Så jeg regnede med at jeg min fjeder havde et kraftmoment på 0,3 Nm per radian. Jeg lavede forsøget med fem forskellige opstillinger. Først et hvor jeg kun havde en stang i, derefter et hvor jeg også havde en vægt på hver ende af stangen 28 cm fra centrum, og derefter 3 opstillinger hvor jeg havde flyttet vægtene henholdsvis 6, 9 og 12 cm tættere ind mod midten.

## Sammenligning/test

I min simulation har jeg sat det op så det minder så meget som muligt om det rigtige eksperiment for at få de mest præcise resultater. Nu er det at det ville være meget optimalt hvis man kunne vise videoer i en pdf, men det svære kan man kun vise billeder og tekst i pdf'er så jeg må nøjes med at vise billeder fra simulationen. Programmet skriver selv perioden som den måler, oppe i toppen af billederne. Det første forsøg jeg satte op, var det mest simple som kun havde stangen på 60 cm uden nogle vægte. Den målte en svingningsperiode på 2,58 sekunder. Og når vi kigger på forsøgsdata fra det rigtige eksperiment, kan vi se at den gennemsnitlige svingningsperiode er på 2,64 sekunder. Det vil sige at der er en afvigelse på -2%.

Da jeg så at simulationen så ud til at virke ret godt med det eksempel, var det tid til at opsætte de andre forsøg i simulationen. Forsøg 2, 3, 4 og 5 simulerede en periode på henholdsvis 8,27s 6,69s 5,91s og 5,17s og det reelle forsøg havde en gennemsnitlig periode på 8,51s 6,87s 6,09s og 5,31s og de havde alle sammen en afvigelse på lidt under -2%.



## Fejlkilder

Som vi kan se, passer denne simulation ret godt med det vi ser i virkeligheden med det forsøg vi har prøvet. En af de ting jeg ikke har implementeret, er modstand fra friktion eller vindmodstand. Så det vil også sige at lige nu fortsætter pendulet med at svinge for evigt eller indtil man stopper simulationen, og det burde ikke være et stort problem for præcisionen af simulationen, men det vil have en lille effekt på den og det skal tilføjes hvis man vil have en helt perfekt løsning. Et andet potentielt problem er at jeg har regnet med at vægtene i forsøget var punkter med masse i stedet for vægte med en vis størrelse. Det vil gøre så massefordelingen ikke er helt korrekt og jeg tror det er derfor vi får en afvigelse på -2%.

## Konklusion

Jeg har fået lavet en simulator ud fra de fysiske formler om inertimoment og rotation, som passer ret godt på de virkelige eksperimenter og vil også i en vis grad kunne bruges til at forudse hvordan objekter vil reagere under forskellig kraft. Dog er modellen begrænset af at det er sjældent man ser et så kontrolleret system i den virkelige verden, så det vil i de fleste realistiske tilfælde kræve mange flere parametre for at kunne simulere præcist hvordan elementer vil rotere. Men dette er stadig en vigtig del til hvis man skal lave en simulation af noget som helst virkeligt, fordi der vil næsten altid være ting der roterer som har brug for at blive beregnet korrekt.

## Litteraturliste

Orbit A Htx - *Per Holck, Birgitte Merci Lund, Jens Kraaer.*

<https://orbithtxa.systime.dk/?id=260> – besøgt 12-12-21

Three.js - *Threejs organisation*

[Https://threejs.org](https://threejs.org) – besøgt 12-12-21

Mozilla Web Docs – *MDN contributors*

<https://developer.mozilla.org/en-US/docs/Web/JavaScript> – besøgt 13-12-21

The science of mechanics – *dr. Ernst Mach* (P. 165-187)

<https://archive.org/details/scienceofmechani005860mbp/page/n165/mode/2up> – besøgt 13-12-21

Programmering - *Jesper Buch*

<https://programmering.systime.dk/> – besøgt 13-12-21

Three js documentation – *Threejs organisation*

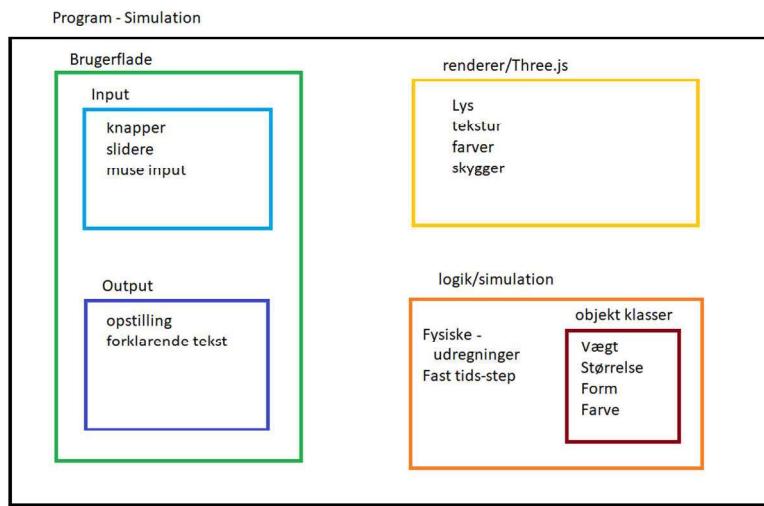
<https://threejs.org/docs/index.html> – besøgt 15-12-21

Isaac computer science – *US dept. of education*

[https://isaaccomputerscience.org/topics/programming\\_languages](https://isaaccomputerscience.org/topics/programming_languages) – besøgt 15-12-21

## Bilag

### Bilag 1 – diagram

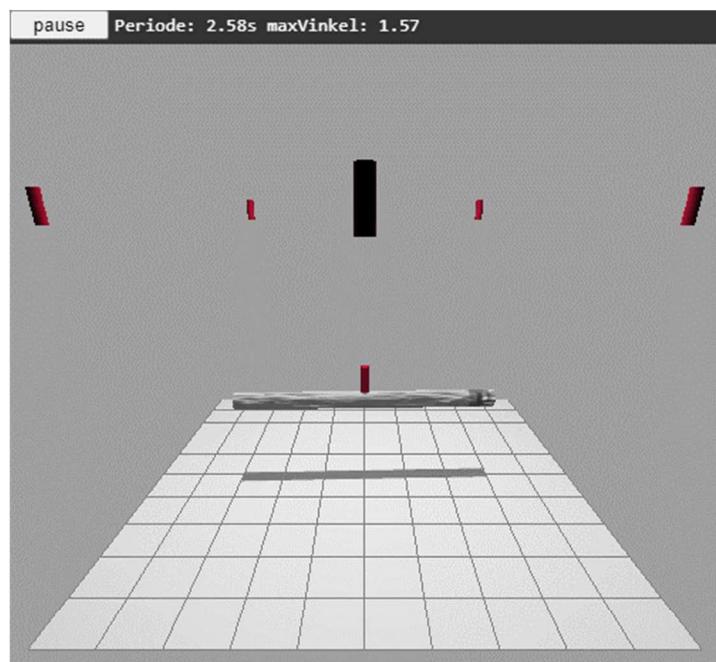


### Bilag 2 – testdata

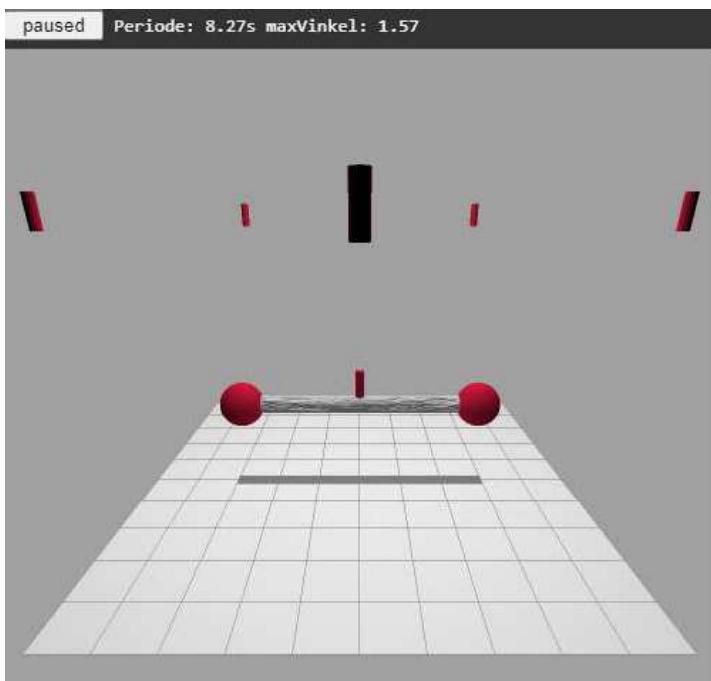
forsøg1	forsøg2	forsøg3	forsøg4	forsøg5
1.35	4.33	2.56	2.81	2.6
1.32	4.24	4.06	3.04	2.67
1.2	4.41	3.93	2.96	2.8
1.22	4.55	3.47	3.18	2.3
1.52	3.98	3.57	3.26	3.04
1.41	4.55	3.43	3.2	2.54
1.14	4.17	3.13	2.63	2.76
1.47	4.22	3.42	3.19	2.65
1.29	4.21	3.48	3.07	2.72
1.33	4.17	2.86	3.17	2.56
1.27	4.45	3.95	3.07	2.71
1.45	4.28	3.48	3.22	2.5
1.29	4.54	3.71	2.99	2.7
1.35	4.24	3.49	2.97	2.72
1.25	4.4	3.23	3.12	2.61
1.34	3.75	3.56	2.92	2.65
1.26	3.83	3.43	3.07	2.89
1.35		3.35	3.29	2.62
1.43		3.16	3.03	2.77
1.33			2.97	2.59
1.19			2.74	2.89
1.45				2.6
1.45				2.6
1.32				2.23

1.16				
1.41				
1.38				
1.26				
1.12				
2.64	8.51	6.87	6.09	5.31

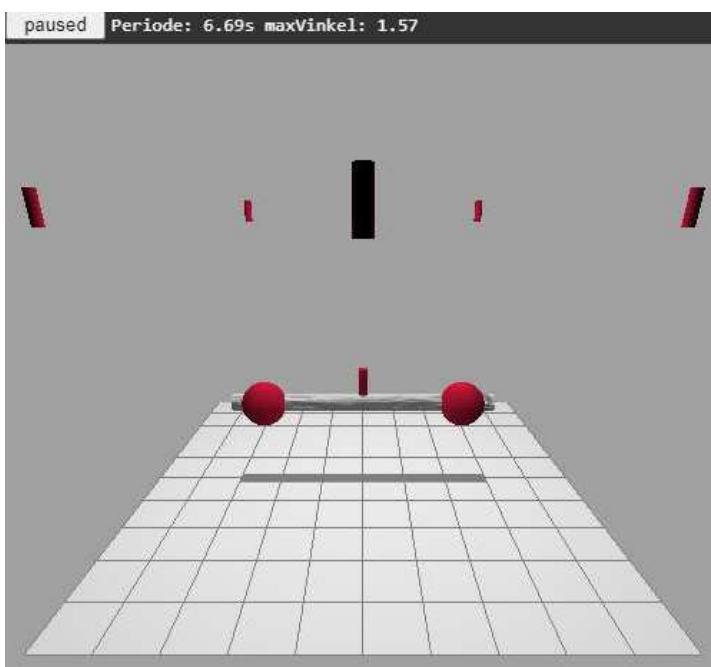
Bilag 3 – Simulation forsøg 1



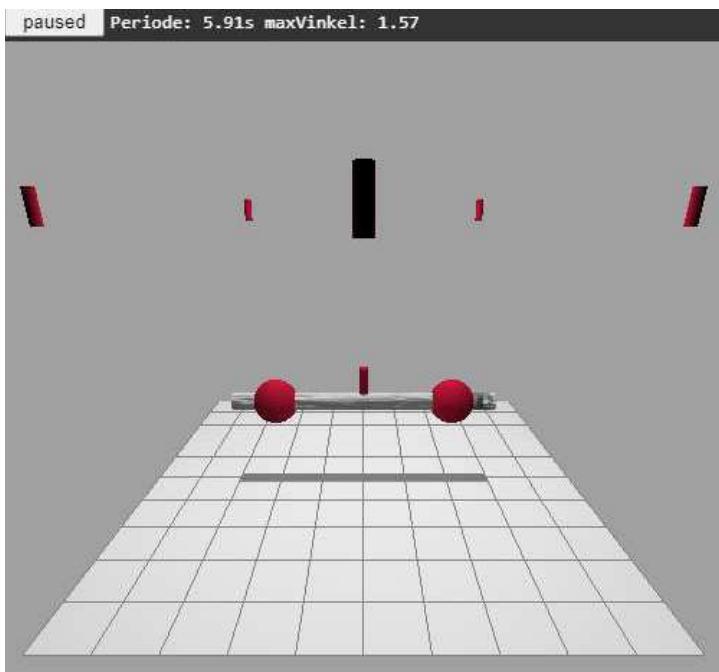
Bilag 4 – Simulation forsøg 2



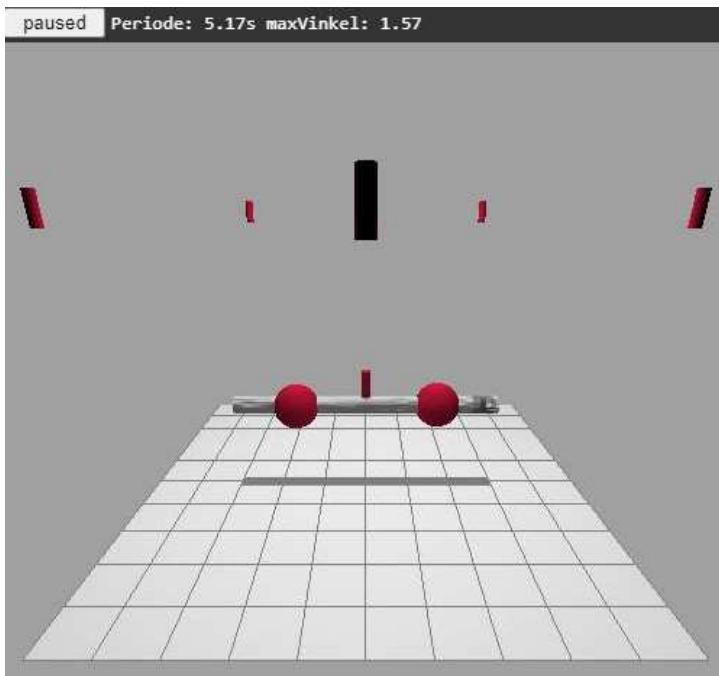
Bilag 5 – Simulation forsøg 3



Bilag 6 – Simulation forsøg 4



Bilag 7 – Simulation forsøg 5



Bilag 8 – index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>SOP</title>
    <meta charset="UTF-8">
```

```

<meta http-equiv="X-UA-Compatible" content="IE=edge">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
</head>
<body>
    a
    <p id="box" style="width: 1000px; position: fixed; left: 50px; top: 10px;">
        <button id="pause" style="width: 70px">pause</button>
        date:
    </p>
    <!-- <canvas id="can" width="300" height="300" style="position: fixed; right: 400px; top: 70px;"></canvas> -->
    <canvas id="can" width="1000" height="1000" style="position: fixed; top: 50px; left: 50px; width: 1000px; height: 1000px;"></canvas>
    <script src="three.js"></script><!--https://threejs.org/build/three.js-->
    <script src="script1.js"></script>
</body>
</html>

```

### Bilag 9 – Script1.js

```

try {
    const THREE = require("three");
    const log4j = require("log4j");
} catch(e){}
const three = THREE;
const canvas = document.getElementById("can");
canvas.onclick = click;
canvas.oncontextmenu = click;
canvas.onmousemove = hover;
document.getElementById("pause").onclick = pause;

//setup
document.body.style.backgroundColor = "#333";
const {width, height} = canvas;
const can = new three.Scene();
const cam = new three.PerspectiveCamera(75, width/height);
const renderer = new three.WebGLRenderer({canvas});
const raycaster = new three.Raycaster();
renderer.shadowMap.enabled = true
can.background = new three.Color(0xa0a0a0);
cam.position.set(0, 5, 10);
cam.lookAt(0, 0, 0);

//grid
var grid = new three.GridHelper(10, 10, "gray", "gray");

```

```
grid.receiveShadow = true;
grid.position.y = -2;
can.add(grid);

//materials
var plaster = new THREE.TextureLoader().load("https://thumbs.dreamstime.com/z/k-
rough-plaster-roughness-texture-height-map-specular-imperfection-d-materials-black-
white-hi-res-200368950.jpg");
const textured = new three.MeshPhongMaterial({color: "lightgray", flatShading:
true, vertexColors: false, shininess: 1, bumpMap: plaster, bumpScale: 1});
const flat = new three.MeshPhongMaterial({color: "lightgray", flatShading: true,
vertexColors: false, shininess: 1});
const transparent = new three.MeshPhongMaterial({color: "white", flatShading:
true, vertexColors: false, shininess: 1, transparent: true, opacity: 0});
const crimson = new three.MeshPhongMaterial({color: "crimson", flatShading: true,
vertexColors: false, shininess: 1});
const yellow = new three.MeshPhongMaterial({color: new three.Color(0.862, 0.90,
0.235), flatShading: true, vertexColors: false, shininess: 1});

const wire = new three.MeshBasicMaterial({color: "black", wireframe: true});

//vars
var time = Date.now();
var Dtime = 0;
var rotation = 0;
var inert = 0;
var momentumn = 0;
var vinkelacceleration = 0;
var speed, limit;
// var objects = [];
var highlight = 0;
var running = true;
// logging/data collection
var hitEnd = 0;
var period = Date.now();
var timer = 0;

var geo = new three.CylinderGeometry(0.2, 0.2, 6, 16);
geo.rotateZ(Math.PI/2)
var bar = new three.Mesh(geo, textured);
bar.receiveShadow = bar.castShadow = true;
can.add(bar);
// createCylinder({height: 2, width:1, x:2, z:0});
// createCylinder({height: 2, width:1, x:-2, z:0});
```

```
var light2 = new three.DirectionalLight(0xffffffff, 0.5);
light2.castShadow = true;
light2.position.set(0, 5, 0);
can.add(light2)
var shadowGeo = new three.PlaneGeometry(10, 10);
var shadow = new three.Mesh(shadowGeo, flat);
shadow.rotateX(-Math.PI/2);
shadow.position.y = -2.01
shadow.receiveShadow = true;
can.add(shadow);
shadow = new three.Mesh(shadowGeo, transparent);
shadow.rotateX(-Math.PI/2);
can.add(shadow);

for (var i = 0, max = 5; i < max*2; i++) {
    let light = new three.PointLight(0xffffffff, 1/max, 40);
    let phi = Math.PI/2-(Math.PI/10*2)*(i>=max?-1:1);
    light.position.setFromSphericalCoords(7, phi, Math.PI*2/max*i);
    can.add(light);
    let marker = createPoint(0, 0, 0);
    light.add(marker);
}

var point = createPoint(0, 0);
point.position.set(0, 0.5, 0);
point.castShadow = true;
point.receiveShadow = true;
can.add(point);

function pause(e) {
    if (running) {
        e.srcElement.innerText = "paused";
        running = false;
        rotation = 0;
        momentumn = 0;
    } else {
        e.srcElement.innerText = "pause";
        running = true;
        rotation = 0;
        momentumn = 1;
    }
}

function click(event) {
    if (running) return;
```

```
var button = event.button == 2;
raycaster.setFromCamera({x:event.layerX/width*2-1, y:-event.layerY/height*2+1}, cam);
var intersects = raycaster.intersectObjects(button ? bar.children : [shadow]);
if (!intersects.length) return;
if (button) {
    // // objects = objects.filter(e => e.uuid != intersects[0].object.uuid);
    bar.remove(intersects[0].object);
    event.preventDefault();
} else {
    let geo = new three.SphereGeometry(0.5);
    let mesh = new three.Mesh(geo, crimson);
    mesh.position.setX(intersects[0].point.x);
    mesh.position.setY(intersects[0].point.y);
    mesh.position.setZ(intersects[0].point.z);
    // objects.push(mesh);
    bar.add(mesh);
}
}

function hover(event) {
    if (running) return;
    raycaster.setFromCamera({x:event.layerX/width*2-1, y:-event.layerY/height*2+1}, cam);
    var intersects = raycaster.intersectObjects(bar.children);
    if (!intersects.length) return highlight.material = crimson;
    if (highlight != intersects[0].object || !highlight) {
        highlight.material = crimson;
        highlight = intersects[0].object
    }
    highlight.material = yellow;
}

function loop() {
    Dtime = Date.now();
    inert = 1/12*0.128*0.6**2;

    // bar.children.forEach(weight => {
    //     weight.position.distanceTo(Vector3())
    // });

    for ( ; Dtime > time; time++) {
        momentumn -= rotation*0.022*0.0135/inert; // angle * spring force
        timer++
        // speed = 2.38, limit = 18800;
        // momentumn = Math.PI;
        rotation += momentumn/1000;
    }
}
```

```

if (timer == limit) {
    console.log(rotation, 1/2*speed*Math.pow(limit/1000, 2),
rotation/(1/2*speed*Math.pow(limit/1000, 2)), Date.now()-period);
    // debugger;
}
if (hitEnd ? momentumn > 0 : momentumn < 0) {
    hitEnd = !hitEnd;
    console.log((timer)*2, Math.abs(rotation).toFixed(3));
    timer = 0;
}
bar.rotation.y = rotation;
}

// setInterval(() => {
//     rotation = 0; console.log(timer, timer = 0, Date.now()-period, period =
Date.now());
// }, 1000);

function point3d(x, y) {
    var point = new three.Vector3();
    point.x = Math.sin(y)*Math.cos(x);
    point.z = Math.sin(y)*Math.sin(x);
    point.y = 2+Math.cos(y);
    return point
}

function createCylinder(options = {weight: 10, height: 2, width: 1, x:2, z:0}) {
    var geo = new three.CylinderGeometry(options.width, options.width,
options.height, 64);
    var mesh = new three.Mesh(geo, textured);
    mesh.receiveShadow = true;
    mesh.castShadow = true;
    mesh.position.set(options.x, 0, options.z);
    bar.add(mesh);
    // var wireframe = new three.Mesh(geo, wire);
    // mesh.add(wireframe);
    var pos = mesh.position;

    mesh.draw = function(rotation) {
        pos.x = Math.sin(rotation)*1.4;
        pos.z = Math.cos(rotation)*1.4;
        mesh.setRotationFromAxisAngle(new three.Vector3(0, 1, 0).add(new
three.Vector3(0, 0, 0)).setLength(1), rotation)
    }
}

```

```
// mesh.setRotationFromAxisAngle(new thre.Vector3(1, 0, 0), rotation/2)
}
// objects.push(mesh)
return mesh
}

function createPoint(x, y, r=1) {
    var geo = new three.CylinderGeometry(0.1, 0.1, 0.6, 16);
    var mesh = new three.Mesh(geo, crimson);
    mesh.receiveShadow = true
    mesh.castShadow = true
    can.add(mesh);
    mesh.position.setFromSphericalCoords(r, x, y);
    return mesh
}

(renderLoop = () => {
    loop();
    renderer.render(can, cam);
    time++;
    requestAnimationFrame(renderLoop)
})();
```