Fysik og Programmering:

Af Josva M.

Indholdsfortegnelse

[Projektbeskrivelse 3](#__RefHeading___Toc382_1637631377)

[Problemformulering 4](#__RefHeading___Toc384_1637631377)

[Indledning 5](#__RefHeading___Toc379_536262420)

[Krav Spec. 5](#__RefHeading___Toc386_1637631377)

[Rutediagram 5](#__RefHeading___Toc388_1637631377)

[Pseudo kode 6](#__RefHeading___Toc390_1637631377)

[Klassediagram 7](#__RefHeading___Toc392_1637631377)

[Prototype (skitse/mockup) 8](#__RefHeading___Toc394_1637631377)

[Litteraturliste 10](#__RefHeading___Toc396_1637631377)

# Projektbeskrivelse

Hoved idé:

Design og implementer et simuleringssystem i form af et Processing-program, der simulerer bevægelsen af en genstand under påvirkning af tyngdekraften og acceleration. Programmet skal tage højde for parametre som partiklens masse, initialhastighed og påvirkningskraftens egenskaber. Kriterier for bedømmelse: Korrekt fysisk modellering af bevægelsen, realistisk simulering af partiklens adfærd, og anvendelse af relevante programmeringskoncepter.

Programmet kører på telefonen og bruger telefonens sensorer for at bestemme tyngdekraften der påvirker genstanden. Telefonen vil bruge tyngdekraften til at påvirker partiklerne i simulationen. Man kan vælge mellem forskellige planeter som eksempel, månen og Mars hvor man bruger deres tyngdekraft i simulationen, der vil påvirke partiklerne i en anden måde som med Jordens tyngdekraft.

Appen kan have sliders der kan skifte på partiklernes egenskaber, som viskositet, vægt og størrelse osv.

Andre Ideer:

* Morse kode program
* Vejrudsigt app
* Budget omregningsprogram.
* Astronomi app...

# Problemformulering

Hvordan kan et interaktivt simuleringssystem designes og implementeres i Processing, der realistisk modellerer bevægelsen af en partikel under påvirkning af tyngdekraft og acceleration?

* Hvordan kan telefonens sensorer anvendes til at bestemme og simulere den aktuelle tyngdekraftpåvirkning?
* Hvordan kan simulationen tilpasses forskellige planeters tyngdekraft (f.eks. Månen, Mars og Jorden)?
* Hvilke parametre og brugerinteraktioner (f.eks. sliders til masse, viskositet, størrelse) kan implementeres for at skabe en fleksibel og realistisk simulation?
* Hvordan sikres det, at den fysiske modellering af bevægelsen er korrekt og realistisk?

# Indledning

Denne projekt startede med at alle i klassen brainstormede om hvad de ville lave som deres programmeringsprojekt. Efter lidt diskussion fandt alle deres egen ideer til et projekt. Målet var at finde en projekt der både var interessant og kunne laves realistisk med vores egne evner og i den givende tidsramme. Jeg valgte at arbejde med noget der fangede min opmærksomhed fra en tidligere forløb. Programmering i Processing i Android mode. Jeg fandt det interessant og teknisk udfordrende, og fordi man har andre muligheder for input, som telefonen sensorer, så kan man lave programmer man ikke direkte kan lave på Java mode. Jeg valgte derfor at lave en partikelsimuleringssystem, der kunne køre på en Android telefon. Allerede fra starten overvejede jeg om hvordan programmet skulle struktureres. Jeg lavede en tidlig udkast af et rutediagram, som gav mig en udblik af hvordan programmet opbygges, som eksempel hvordan partiklernes bevægelser skulle håndteres. Mit fokus var primært rettet mod at få partikelsystemet til at virke i processing. Den skulle virke på en mobiltelefon og skulle bruge den sensorer til at påvirke partiklernes bevægelser. Der fokuseres også på brugeroverfladen, da brugeren skal kunne ændre i nogle variabler på en intuitiv måde, og at den skal være naturligt at bruge. På baggrund af denne idé har jeg lavet en problemformulering som projektet tager udgangspunkt i:

**Hvordan kan et effektivt og brugervenligt partikelsimuleringssystem designes og implementeres til Android, med fokus på brugervenlighed og interaktion?**

For at kunne besvare denne problemformulering vil jeg undersøge forskellige metoder til at simulere partiklerne. Jeg vil også undersøge hvad gør en god brugeroverflade og hvordan brugeren kan interagere med appen via touchskærm.

# Metode

For at kunne lave programmet, skal der bruges nogle metoder som vil hjælpe mig igennem projektet. Til denne projekt er det et krav at bruge Processing som kodesproget. Processing er Java, men med nogle få ændringer i måden man sætter den op på. Det er metoden der er brugt i denne projekt.

Rutediagrammer er brugt til at give en overblik over hvordan programmet skal opbygges og fungere. Dette vil give et overblik over hvilke funktioner der er i programmet og mere. Rutediagrammet bliver lavet før koden, da det er en del af planlægningen. Men hvis der kommer ændringer i koden senere hen, så må rutediagrammet ændres til at passe til koden.

Der anvendes også pseudokode både før og efter koden er skrevet. Den kan bruges som både planlægning og refleksion over koden. Det er en mere forklarende version af rutediagrammet. Pseudokoden vil også ændres alt efter ændringerne lavet i koden.

Hele processen laves iterativt. Som nævnt, det betyder at hvis der er ændringer i koden, så ændres rutediagrammet og pseudokoden til at passe til koden. Der vil laves kommentar i koden så den er nemmere at forstå når man ser på den.

Kommentare i koden vil blive lavet så hver del af kode har en lille forklaring af hvad den gør. På den måde, kan man nemmere se hvad koden handler om. Kommentarerne er lavet i engelsk, da Java kodesproget er baseret på engelsk og er derfor nemmere at referere til ting på engelsk. Pseudokoen bliver dog skrevet på dansk.

# Krav Spec.

Programmet skal have nogle hårde og bløde krav. Hårde krav er de krav som programmet skal have.

Hårde krav:

* Programmet skal bruge telefonens sensorer (accelerometer, gyroskop) til at påvirke partiklerne i simuleringen.
* Programmet skal have nogle sliders der tillader brugeren at skifte på partiklernes egenskaber.
* Simulationen skal ske i realtid på telefonen.

Bløde krav

* Der kan være justeringer til hvor mange partikler der er i simulationen, til de tilfælde hvor telefonen ikke har nok ydeevne til at simulere
* Programmets brugeroverflade skal være nem at bruge. Sliders skal være brugervenlige og intuitiv af bruge med en touchskærm.

# Hoveddel

Der skulle laves overvejelser af hvilke krav programmet skulle have og hvordan den skulle programmeres. Der var nogle opfordringer til at lave programmet med objektorienteret kodning. Dette betød at flere dele af programmet, som dens GUI og partikelsimuleringssystemet, skulle laves som hver deres objekt eller klasser. Så kan man lave klasser til GUI, sliders og partikelsimuleringssystemet, som kan kaldes ind i hoveddelen af programmet, enten i setup() eller draw().

Her er et eksempel på hoveddelen af programmet med setup(). Den kalder GUI klassen og partikelsystemet sammen med værdi af slider2.

Gui = new GUI(); opretter et objekt af GUI klassen. Det betyder at alt der har med brugergrænsefladen som sliders og baggrunde, er samlet i et sted. Det som position of størrelse af sliders og andre elementer ligger i GUI klassen.

ps = new ParticleSystem(gui.slider2.getValue());

her oprettet et objekt af ParticleSystem klassen. Antallet af partikler som den starter med er afhængig af værdien som slider2 har. Den viser hvordan noget af partikelsystemets data er taget fra slider2 når den starter.

void **setup**() {

fullScreen();

orientation(LANDSCAPE);

gui=new GUI(); //initiate GUI class

textSize(50);

ps=new ParticleSystem(gui.slider2.getValue()); // Initial number of particles

Her er koden for draw, set nedenfor, hvor den kalder på gui.drawGUI(); for at kalde GUI klassen. Den håndterer brugeroverfladen lidt ligesom i setup(). Men forksellen her er at denne klasse er opdateret 60 gange per sekund.

Gui.slider1.getValue() og gui.slider2.getValue() er programmet bruger til at hente værdier fra GUI’en uden af vide præcist hvordan slidersne fungerer. Et OOP princip: Abstraktion

(<https://programmering.systime.dk/?id=204#c1066>)

ps.setGravity(gravity); kalder metoden på ps objektet, eller ParticleSystem okbjektet, der dynamisk ændrer systemet.

ps.update() og ps.display viser at ParticleSystem objektet har selv styr på at opdatere og tegne sig selv.

void **draw**() {

background(0);

gui.drawGUI();// draws the UI

// Update gravity based on slider1 value (0–10G)

float gravityMagnitude = gui.slider1.getValue() \* 0.2; // scale G (adjust if needed)

gravity.set(gravityDirection.copy().mult(gravityMagnitude)); // set gravity vector

ps.setGravity(gravity);// apply gravity to particle system

// Check if slider2 changed, recreate ParticleSystem

int desiredParticles = gui.slider2.getValue();

if (ps.numParticles != desiredParticles) {

ps=new ParticleSystem(desiredParticles); // recreate system if count changed

}

pushMatrix(); // isolate transformations

translate(simX, simY); // Shift origin to simulation area

ps.update();

ps.display();//draw particles

popMatrix();

}

Til designen af GUI’en, bruges gestalt loven om nærhed, med baggrunde som viser brugeren at indstillingsdelen og simuleringsdelen af grænsefladet er tilknyttet til hinanden.

Nogle dele af programmet har taget inspiration fra eksempler fra Processing referencen. Slidersne er nærmest er direkte port fra Processing hjemmesidens eksempel, og de fungerer på samme måde. Denne eksempel er ikke objektorienteret, så der skulle ændres lidt i koden for at gøre den til et objekt. Men kernefunktionaliteten er der stadig. (<https://processing.org/examples/scrollbar.html>)

Partikel systemet var lavet med hjælp af ChatGPT, hvor den også hjalp med at intigrere koden med min kode som sin egen klasse. GUI’en er selv lavet med hjælp af en vejleder.

# Konklusion

I dette projekt har jeg designet og implementeret et interaktivt partikelsimuleringssystem i Processing til Android, med fokus på brug af telefonens sensorer og en brugervenlig grænseflade. Formålet var at lave en realistisk simulering af partikelbevægelse under påvirkning af tyngdekraft og acceleration, som kunne tilpasses vi brugerinput. Gennem brugen af objektorienteret programmering blev koden struktureret i klasser, som fx. GUI og ParticleSystem, som gav en bedre adskillelse af funktionalitet og gjorde programmet lettere at udvikle og udvide. Programmet bruger telefonens sensorer til at simulere fysisk påvirkning, og brugeren har mulighed til at ændre tyngdekraften eller antal partikler i simuleringen, alt i realtid. Dette gør simuleringen fleksibel og tilpasselig, som var en central del a problemformuleringen.

Undervejs, var der brug af planlægningsværtøjer som rutediagrammer og pseudokode, som gav et overblik og gjorde det lettere at arbejde iterativt med koden. Resultatet er en Android applikation, der kombinerer sensordata og grafisk simulering på en engagerende måde.

# Litteraturliste

<https://processing.org/reference>

<https://android.processing.org/>

[https://programmering.systime.dk](https://programmering.systime.dk/)

<https://www.color-hex.com/color-palette/8919>

[https://www.chatgpt.com](https://www.chatgpt.com/)

# Bilag

## Prototype (skitse/mockup)



Her er en skitse af hvordan appen kunne se ud. Til højre i den sorte del, vil simuleringen foregå. Og til venstre er der nogle sliders og knapper, der ændrer på hvordan simulationen virker. Der er en ”tyngdekraft” slider, men der kunne også være en ”viskositet” eller ”vindmodstand” der også kunne skifte hvordan simulationen fungerer. Så er der knapperne. De her kunne virke som en slags ”preset” der sætter tyngdekraften og de andre sliders så de passer til den planet man trykker på.

## Rutediagram

Ligger ind på Draw.io

Nyeste screenshot

A screenshot of a computer

AI-generated content may be incorrect.

Denne rutediagram viser hvordan programmet virker når den starter. Den beskriver hoveddelen af Processing-programmet, hvor setup() og draw() finder sted.

Efter programmet starter, så kører den setup() hvor den kalder og starter GUI og starter ParticleSystem. draw() sørger for at ParticleSystem er konstant opdateret. Data fra SensorManager og AccelerometerListener bliver sent til ParticleSystem som bliver brugt af simulationen.

## Pseudokode

Der vælges at lave pseudokoden efter koden til programmet er skrevet. Dette bruges som refleksion til programmet der forklarer i ord hvordan programmet fungerer og hvad der sker i koden. Pseudokoden er en forenklet version af koden forklarede i ord, i stedet for et kodesprog.

Programmet starter først med at initiere variabler for context, sensorer, GUI, partikel system, og tyngdekraft sensor (Accelerometer)

Når programmet starter (setup), dette køres en gang:

* sæt til fuldskærm og i landskabs orientering.
* Starte GUI instance
* Sætter text størrelse til 50
* Lav partikel system med de antal numre partikler bestemt af slider2
* Få Android context
* Initiere sensor manager og accelerometer sensorer
* Opret og registrere et accelerometer lytter

draw() kører i 60FPS

* lav baggrunden sort med background(0); | dette går skærmen blank for dem næste frame i stedet for at alle ting bliver tegner over hinanden.
* Tegne GUI componenter, som slidere og baggrunde for instilling og simulerings områder.
* Få tyngdekraft stykre fra slider1 og skalere det, så den virker som en G meter.
  + Tyngdekraftens retning ganges (fra accelerometeret) med denne styrke
  + Anvend denne tyngdekraften vektor til partikelsystemet.
* Hvis værdien fra slider2 ændres (nummer af partikler)
  + Så skal den gen-lave partikelsystemet med den nye værdi.
* Flyt startpunktet af simuleringen til midten af simuleringsstedet.
* Opdatere og tegner partiklerne

## Klassediagrammer

Her er nogle klassediagrammer til programmet.

|  |
| --- |
| Klasse: UI (brugeroverflade) |
| Attributter   * Sliders * Baggrunde |
| Metoder:   * justerEgenskaber() |

Klasse for UI/brugeroverfladen, hvor der er nogle sliders- og knapper til at skifte partiklernes egenskaber.

|  |
| --- |
| Klasse: Partikel/Partikler |
| Attributter:   * vægt * størrelse * viskositet |
| Metoder:   * simulerBevægelse() |

Klassen for en partikel eller partiklerne i simuleringen. Den har tre attributter med vægt, størrelse og viskositet af partiklerne, som afgør hvordan partiklerne bevæger sig på.

|  |
| --- |
| Klasse: Simuleringssystem |
| Attributter:   * form * kriterier * bedømmelse * anvendelse |
| Metoder:   * modellering() * simulering() |

Klassen for programmets simuleringssystem.

|  |
| --- |
| Klasse: UI (brugeroverfladen) |
| Attributter:   * Sliders * backgrounds |
| Metoder:   * justerEgenskaber() |

Klassen for brugeroverfladen.

# Source code

1.  import android.content.Context;

import android.hardware.Sensor;

import android.hardware.SensorManager;

import android.hardware.SensorEvent;

import android.hardware.SensorEventListener;

Context context;

SensorManager manager;

Sensor sensor;

AccelerometerListener listener;

GUI gui; //declare GUI

ParticleSystem ps;

PVector gravityDirection = new PVector(0, 0); // direction of gravity from accelerometer

PVector gravity = new PVector(0, 0); // actual gravity force applied to particles

int simX = 855; // X position of simulation area

int simY = 45; // Y position of simulation area

int simW = 990; // Width of simulation area

int simH = 990; // Height of simulation area

int value = 0;

boolean firstMousePress = false;

void setup() {

fullScreen();

orientation(LANDSCAPE);

gui=new GUI(); //initiate GUI class

textSize(50);

ps = new ParticleSystem(gui.slider2.getValue()); // Initial number of particles

context = getActivity(); // get Android context

manager = (SensorManager)context.getSystemService(Context.SENSOR\_SERVICE); // get sensor manager

sensor = manager.getDefaultSensor(Sensor.TYPE\_ACCELEROMETER); // use accelerometer sensor

listener = new AccelerometerListener(); // create new sensor event listener

manager.registerListener(listener, sensor, SensorManager.SENSOR\_DELAY\_GAME); // register listener

}

void draw() {

background(0);

gui.drawGUI(); // draws the UI

// Update gravity based on slider1 value (0–10G)

float gravityMagnitude = gui.slider1.getValue() \* 0.2; // scale G (adjust if needed)

gravity.set(gravityDirection.copy().mult(gravityMagnitude)); // set gravity vector

ps.setGravity(gravity); // apply gravity to particle system

// Check if slider2 changed, recreate ParticleSystem

int desiredParticles = gui.slider2.getValue();

if (ps.numParticles != desiredParticles) {

ps = new ParticleSystem(desiredParticles); // recreate system if count changed

}

pushMatrix(); // isolate transformations

translate(simX, simY); // Shift origin to simulation area

ps.update();

ps.display(); //draw particles

popMatrix();

}

class AccelerometerListener implements SensorEventListener {

public void onSensorChanged(SensorEvent event) {

// Android landscape: event.values[0] = left-right, [1] = up-down, [2] = Z

gravityDirection.set(event.values[1], event.values[0]); // Invert X because phone is in landscape

gravityDirection.normalize();

}

public void onAccuracyChanged(Sensor sensor, int accuracy) {

}

}

void mousePressed() {

if (!firstMousePress) {

firstMousePress = true;

}

}

1.  class GUI { //static GUI elements

//Attributes

int settingsWindowHeight = 800;

int settingsWindowWidth = 990;

int particleWindowHeight = 990;

int particleWindowWidth = 990;

Slider slider1 = new Slider(settingsWindowWidth/2-350,settingsWindowHeight,"Tyngdekraft");

Slider slider2 = new Slider(settingsWindowWidth/2-350,settingsWindowHeight-175,"Partikler");

//Constructor

//Methods

void drawGUI(){

fill(45);

rect(45,45,settingsWindowHeight,settingsWindowWidth);

rect(855,45,particleWindowHeight,particleWindowWidth);

slider1.update();

slider2.update();

slider1.drawSlider();

slider2.drawSlider();

}

}

1.  class Particle { //Ref. ChatGPT

// Position of the particle in 2D space

PVector position;

// Velocity of the particle, representing its speed and direction

PVector velocity;

// Acceleration of the particle, representing the change in velocity

PVector acceleration;

// Radius of the particle, used for rendering and collision detection

float radius = 10;

// Constructor to initialize the particle with a specific position

Particle(float x, float y) {

position = new PVector(x, y); // Set the initial position of the particle

// Initialize velocity with random values for a more dynamic effect

velocity = new PVector(random(-1, 1), random(-2, 0));

acceleration = new PVector(0, 0); // Start with no acceleration

}

// Apply a force to the particle, affecting its acceleration

void applyForce(PVector force) {

acceleration.add(force); // Add the force to the current acceleration

}

// Update the particle's position and velocity based on its acceleration

void update() {

velocity.add(acceleration); // Update velocity based on acceleration

position.add(velocity); // Update position based on velocity

acceleration.mult(0); // Reset acceleration for the next frame

}

// Check if the particle has hit the edges of the window and apply damping

void checkEdges(float damping) {

// Horizontal edges

if (position.x < radius) {

position.x = radius;

velocity.x \*= -damping;

} else if (position.x > simW - radius) {

position.x = simW - radius;

velocity.x \*= -damping;

}

// Vertical edges

if (position.y < radius) {

position.y = radius;

velocity.y \*= -damping;

} else if (position.y > simH - radius) {

position.y = simH - radius;

velocity.y \*= -damping;

}

}

// Render the particle on the screen

void display() {

fill(100, 150, 255); // Set the fill color for the particle

stroke(50); // Set the stroke color for the outline

// Draw the particle as a circle at its position

ellipse(position.x, position.y, radius \* 2, radius \* 2);

}

}

1.  class ParticleSystem { //Ref. ChatGPT

// Array to hold all the particles in the system

Particle[] particles;

// Total number of particles in the system

int numParticles;

// Vector representing the gravitational force applied to the particles

PVector gravity;

// Radius within which particles will interact with each other

float interactionRadius = 30;

// Strength of the repulsion force between particles

float repulsionStrength = 0.5;

// Damping factor to reduce the velocity of particles over time

float damping = 0.7;

// Constructor to initialize the particle system with a specified number of particles

ParticleSystem(int count) {

numParticles = count; // Set the number of particles

gravity = new PVector(0, 0.2); // Initialize gravity vector

particles = new Particle[numParticles]; // Create an array for particles

reset(); // Initialize particle positions

}

// Update the state of the particle system

void update() {

for (int i = 0; i < numParticles; i++) {

particles[i].applyForce(gravity);

for (int j = 0; j < numParticles; j++) {

if (i != j) {

PVector dir = PVector.sub(particles[i].position, particles[j].position);

float d = dir.mag();

if (d < interactionRadius && d > 0) {

dir.normalize();

float force = repulsionStrength / d;

particles[i].applyForce(dir.mult(force));

}

}

}

particles[i].update();

particles[i].checkEdges(damping); // Check inside sim area

}

}

void reset() {

for (int i = 0; i < numParticles; i++) {

particles[i] = new Particle(simW/2, simH/2); // local to simulation box

}

}

// Display all particles in the system

void display() {

for (Particle p : particles) {

p.display(); // Call the display method for each particle

}

}

// Set a new gravity vector for the particle system

void setGravity(PVector g) {

gravity.set(g); // Update the gravity vector

}

}

1.  class Slider {

String sliderName; // Label for the slider

int hojde = 50; // Height of the slider

int laengde = 600; // Length of the slider

PVector pos; // position of the slider

PVector sliderPos; // Position of the slider knob

int loose = 10; // Controls how smoothly the slider knob moves (higher = smoother)

boolean over = false; // True if mouse is over the slider

boolean locked = false; // True if slider is currently being dragged

float newspos; // Target position for the slider knob

// Constructor

Slider(int x, int y, String sliderN) {

pos = new PVector(x, y); // Set slider position

sliderName = sliderN; // Store the name

sliderPos = new PVector(pos.x + random(laengde), y); // Random starting position of slider

newspos = sliderPos.x; // Set starting position

}

// Draws the slider UI

void drawSlider() {

fill(255);

text(sliderName + ": " + getValue(), pos.x, pos.y - 10); // Show name + current value

fill(125);

rect(pos.x, pos.y, laengde, hojde); // Draw the background bar

fill(over || locked ? color(0) : color(255)); // Change knob color when hovered or locked

rect(sliderPos.x, sliderPos.y, 30, hojde); // Draw the slider knob

}

// Smoothly moves the slider knob and handles interaction

void update() {

over = overEvent(); // Check if mouse is over

if (firstMousePress && over) {

locked = true;

}

if (!mousePressed) {

locked = false;

}

if (locked) {

newspos = constrain(mouseX - 15, pos.x, pos.x + laengde - 30); // Set knob pos

}

if (abs(newspos - sliderPos.x) > 1) {

sliderPos.x += (newspos - sliderPos.x) / loose; // Smooth interpolation to target.

}

}

// Maps the knob position to an integer value between 0–11

int getValue() {

return int(map(sliderPos.x, pos.x, pos.x + laengde - 30, 0, 11));

}

// Detects if mouse is within slider bounds

boolean overEvent() {

return mouseX > pos.x && mouseX < pos.x + laengde &&

mouseY > pos.y && mouseY < pos.y + hojde;

}

}