H.C. Ørsted Gymnasiet

Opgaveformulering SOP 2021-22

Klasse:	3a2
Navn:	Frederik Cayré Hede-Andersen
Fødselsdato	

Fag:	Niveau:	Vejleder navn:	Vejleders mailadresse:
DDU - Digitalt design og udvikling	A	Anders Juul Refslund Petersen	ajrp@tec.dk
Fysik	A	Tina Hvid	thv@tec.dk

Hvordan simuleres væskedynamik og hvad påvirker tidskompleksiteten

Hovedspørgsmål:
Hvordan kan et program vise forskellene i simuleringer af væskedynamik, samt give indblik i relevante applikationer af simuleringer, med forskellige algoritmer.

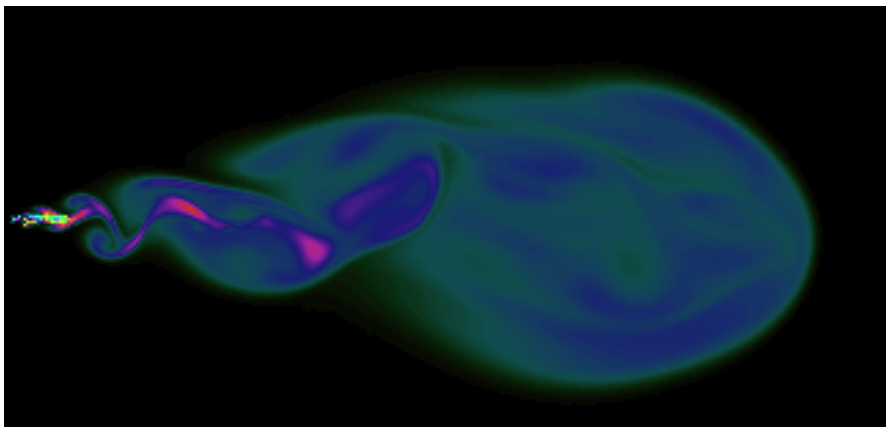
Opgaveformulering:

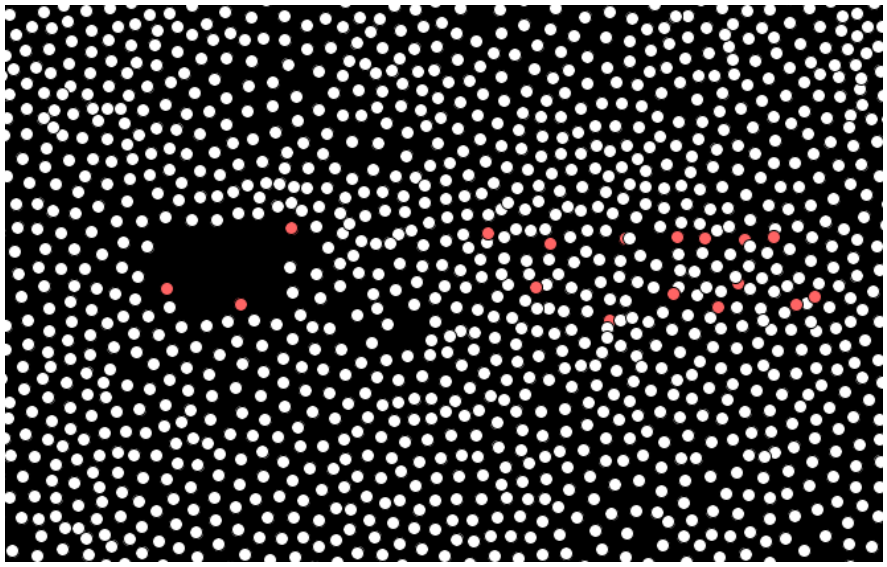
- Redegør for væskedynamik og fluidsims.
- Design og implementer et program der simulerer væskedynamik ved hjælp af flere forskellige metoder og algoritmer.
- Opstil et forsøg der tester programmets forskellige algoritmer til simulering af væskedynamik i forhold til tid og præcision.
- Analyser data fra forsøget og sammenlign resultater med en tidskompleksitets analyse af algoritmernes implementering.
- Diskuter forsøget og programmets validitet og hvad resultaterne kan fortælle om brugen af fluidsims.
- Perspektiver til anvendelsen af fluidsims inden for computerspilsindustrien.

Note: fluidsim = simulering af væskedynamik

Udleverede bilag	
Opgaven udleveres	**04. marts 2022 kl. 12:00**
Opgaven skal afleveres	**18. marts 2022 kl. 12:00**

18. MARTS 2022





SIMULERING AF VÆSKEDYNAMIK

EN SOP I FYSIK A OG DIGITAL DESIGN UNDERVISNING A

FREDERIK CAYRÉ HEDE-ANDERSEN
3.A2 HCØL

Indholdsfortegnelse

Abstract

Simuleringer af væskedynamik er vigtige og bruges i mange forskellige industrier. De er oftest baseret på Navier-Stokes ligningerne, men kan også være baseret på andre beskrivelser af væske flow. Præcision er ikke altid vigtigt når det gælder væskedynamik. To programmer skrives, et med en feltbaseret metode og et med en partikelbaseret metode. Programmerne måles, for at bestemme tiden det tager at beregne væskesimuleringen under forskellige forhold. Den feltbaserede metode skalerer bedst, og har altså den mindste stigning af tidsforbrug når størrelsen af simuleringen udvides, ud af de to programmer. Det partikelbaserede program simulerede ikke væske særlig godt, i hvert fald ikke på en måde der kan sammenlignes med den feltbaserede metode. At se på hvordan algoritmerne skalerer er relevant, da det fortæller om de forskellige områder en algoritme ville bruge i. Computerspilsindustrien ville gavne meget af brug af fluidsims, desværre er computere ikke kraftfulde nok til at kunne køre fluidsims sammen med alt det andet der også er i spil. Fluidsims er fantastiske.

Indledning:

Computere ses alle steder i verden og bruges til alle mulige ting. Fra styring af ovne, til design af havelåger, til beregninger af yderligere cifre af pi; computere kan en masse. Modellering af virkeligheden bliver også gjort på computere i mange industrier rundt omkring i verden. Det at kunne opleve verden og genskabe fænomener gennem en computer, uden faktisk at skulle gå ud med et kamera og måle på tingene, kan bruges til rigtig mange ting. Det kan gøre besværet med bestemte opgaver meget mindre og det kan gøre folk i stand til at opleve ting de normalt ikke ville. Naturen er vores at kommandere inde i computeren. Specifikt inden for feltet væskedynamik er computersimuleringer ekstra brugbare, da det kan være svært at måle på mange af en væskes egenskaber kontinuert. Væsker er dog svære at simulere, og det er tungt at køre på langt de fleste computere, især hvis simuleringen skal være naturlig. Der er dog mange måder at simulere væskedynamik på, ved hjælp af forskellige algoritmer. Det er det vi kigger på i denne opgave.

Problemformulering

”Hvordan kan et program vise forskellene i simuleringer af væskedynamik, samt give indblik i relevante applikationer af simuleringer, med forskellige algoritmer.”

Denne problemformulering vil jeg svare på ved hjælp af en række underspørgsmål, samt tests af et program jeg skriver, og en bedre generel forståelse af feltet. De følgende underspørgsmål skal give os en bedre forståelse af emnet, samt få os på det rette spor når det gælder analysen.

Underspørgsmål:

- Hvad er væskedynamik og hvordan beskrives væskedynamik matematisk?
- Hvordan kan computere bruges til fluidsims?
- Hvilke algoritmer og metoder kan bruges til fluidsims?
- Hvordan kan et program designes og testes for at vise forskellene mellem forskellige fluidsims?
- Hvad er tidskompleksiteten af de implementerede algoritmer?
- Hvad kan forsøgets resultater fortælle os om algoritmerne?
- Hvordan kan algoritmerne bruges i forskellige industrier?

- Hvad fortæller forsøget om anvendelsen og mulighederne af fluidsims inden for computerspilsindustrien?

Note: Fluidsims = simuleringer af væskedynamik.

Hoveddel

Navier-Stokes

Væskedynamik er det felt der håndterer bevægende stoffer på flydende og på gas form. Igennem lang tid har væsker været set på og undret over, for at kunne modellere deres bevægelse matematisk. Archimedes lov stammer fra denne undren om væsker, og der ville stille og roligt blive udviklet mere avancerede ideer om væskedynamik gennem århundrederne. Da oplysningstiden begyndte skete der virkelig noget, og i løbet af 1800-tallet fik to matematikere Claude-Louis Navier og George Gabriel Stokes udviklet en definitiv samling matematiske formler. Disse formler beskriver væsker med viskositet og er kendte for at være meget svære at løse for alle på nær de nemmeste væskesystemer. Vi kommer hovedsageligt til at arbejde med inkompressible væsker, da de er nemmere at have at gøre med. Luft er derfor ikke noget vi kommer til at se på, vand derimod er en inkompressibel væske, så det er mulig for os at modellere. Selvom alle væsker er en smule kompressible kommer det til at have en ubetydelig effekt i langt de fleste tilfælde.

For at holde formlerne nogenlunde overskuelige og for at gøre det nemmere at implementere kigger vi kun på inkompressible væsker. Et simpelt væskesystem kan derfor beskrives ved hjælp af et sæt af Navier-Stokes formlerne der ikke er så tunge. De følgende formler kan bruges til at beskrive et væskesystem med en inkompressibel væske.[1]

$$\nabla \cdot \vec{u} = 0 \qquad (1)$$

$$\frac{\partial \vec{u}}{\partial t} = -(\vec{u} \cdot \nabla)\vec{u} - \frac{1}{\rho_0}\nabla p + \nu\nabla^2\vec{u} + \vec{f} \qquad (2)$$

Den øverste af formlerne (1) beskriver hvordan masse konserveres gennem væskesystemet. Da væsken er inkompressibel må densiteten af væsken være den samme over alle punkter. Formlen (1) beskriver hvordan der ikke kan flyde mere eller mindre væske ind i et punkt end der flyder væske ud af punktet. ’∇’ og ’$\nabla \cdot$ ’ er to forskellige operatorer der arbejder på henholdsvis skalar- og vektorfelter. ∇ kan ses som en vektor af de delvist afledte af skalarfeltets dimensioner. ∇ kan beskrives således for et todimensionelt skalarfelt:[2]

$$\nabla = \left(\frac{\partial}{\partial x}, \frac{\partial}{\partial y}\right)$$

Så på et skalarfelt F bliver ∇F til et vektorfelt, kaldet en gradient, hvor hver vektor peger i retning af den største stigning af værdierne af skalarfeltet.

$$\nabla F = \left(\frac{\partial F}{\partial x}, \frac{\partial F}{\partial y}\right)$$

Kigger vi derimod på den anden operator, $\nabla \cdot$, agerer den på vektorfelter. Tager vi og kigger på $\vec{u}$, vektorfeltet over hastigheder i væsken, bliver $\nabla \cdot \vec{u}$ til et skalarfelt. ∇ prikkes essentielt med hastighedsvektoren, beskrevet således:

$$\nabla \cdot \vec{u} = \left(\frac{\partial}{\partial x}, \frac{\partial}{\partial y}\right) \cdot \left(\vec{u}_x, \vec{u}_y\right) = \frac{\partial \vec{u}_x}{\partial x} + \frac{\partial \vec{u}_y}{\partial y}$$

Dette beskriver den divergens der eksisterer i vektorfeltet, i dette tilfælde hastighedsvektorfeltet. Der betragtes de omkringliggende vektorer og ses på hvor meget 'hastighed' der løber ind og ud af punktet. $\nabla \cdot u$ er negativt når vektorerne løber mere ind mod punktet end ud. Formlen (1) sætter dette til at være 0. Da densiteten er konstant over hele væsken, fordi den er inkompressibel, kan der ikke løbe mere væske ind i et punkt end der løber ud. Dette ville nemlig give at masse skulle forsvinde eller skabes i punktet, hvilket ikke ville give nogen mening.

$$\frac{\partial \vec{u}}{\partial t} = -(\vec{u} \cdot \nabla)\vec{u} - \frac{1}{\rho_0}\nabla p + \nu \nabla^2 \vec{u} + \vec{f} \qquad (2)$$

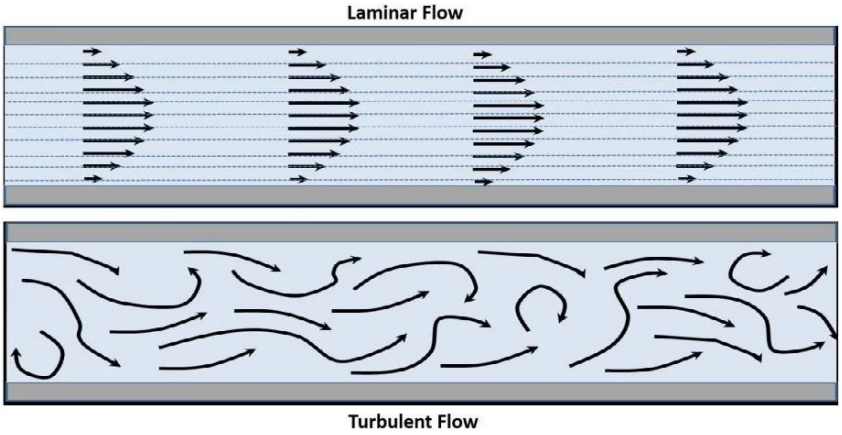
Betragter vi den anden formel (2), får vi beskrevet hvordan hastighedsvektorerne ændres over tid. Her konserveres hastighed. Formlens forskellige led har hvert at gøre med sin egen effekt på væsken, og kan betragtes alene. Det første led $(\vec{u} \cdot \nabla) \cdot \vec{u}$ har at gøre med konvektion og behandler komposanterne af hastighedsvektorerne for at få deres resulterende bevægelse. Leddet kan udvides til dette:

$$(\vec{u} \cdot \nabla)\vec{u} = \begin{pmatrix} u \cdot \frac{\partial u}{\partial x} + v \cdot \frac{\partial u}{\partial y} \\ u \cdot \frac{\partial u}{\partial x} + v \cdot \frac{\partial v}{\partial y} \end{pmatrix}$$

Vi ender altså med en vektor der er den originale hastighedsvektor $\vec{u}$ ganget på ændringen af komposanterne over deres retninger. Det andet led i formlen (2) beskriver hvordan trykgradienten påvirker bevægelsen af væsken, hvor meget er dog påvirket af densiteten. Det tredje led er diffusion, og beskriver ved hjælp af viskositeten af væsken hvordan væsken bliver påvirket af resten af væsken rundt om i forhold til viskositeten. Det sidste led er alle andre kræfter der kunne påvirke væsken. Det kunne f.eks. være tyngdekraft.[4]

Laminar og turbulent strømning[3]

Senere for at vurdere om simuleringerne afspejler den virkelige verden kan vi tage og kigge på nogle bestemte karakteristika for strømningen, eller 'flowet' som det også hedder. Til at hjælpe med dette kan vi betragte Reynolds tal, og ligningen bag det. Reynolds tal repræsenterer en serie af et flows karakteristika og kan bruges til at bestemme om flowet vil være laminart eller turbulent. Laminart flow, er når de forskellige vandlag bevæger sig parallelt uden at blande, og af samme vej. Turbulent flow er det modsatte. Det er kaotisk, svært at forudsige og blander vandlag af forskellige hastigheder.



Figur 1, forskelligt flow. Billedkredit: Joseasorrentino, Transicion laminar en turbulento

Størrelsen af Reynolds tal er altså en indikator på hvordan flowet vil være. Jo lavere tallet, desto mere laminart er det. Ligningen er givet ved[3]:

$$\mathrm{R\ e} = \frac{u\ D}{\nu}$$

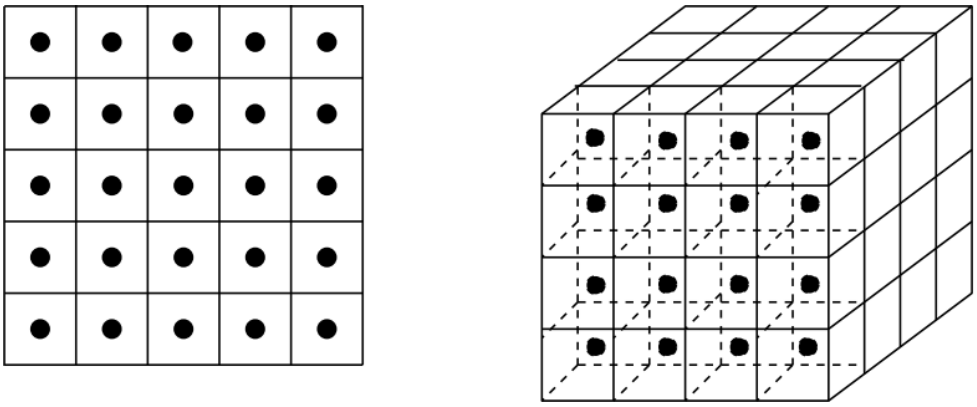
Hvor D_H er diameteren af flowet, u er hastigheden og ν er den kinematiske viskositet vi også så i Navier-Stokes ligningerne. Vi kan ud fra formlen konkludere at når viskositeten stiger må Reynolds tallet falde. Og falder Reynolds tallet tilstrækkeligt nok bliver flowet laminart.[3]

Computeren og væskedynamik som algoritmer

Vi har set på Navier-Stokes formlerne for inkompressible væsker, men hvordan oversætter vi det til et computerprogram. Den største forskel mellem hvordan formlen fungerer og hvordan vi kommer til at implementere den er domænet. Formlerne beskriver et kontinuert væskesystem, hvilket er besværligt at arbejde med. Da vi kommer til at simulere væsker numerisk og ikke analytisk må vi gøre væskesystemet diskret. I virkeligheden er en væske ikke kontinuer. Zoomer vi langt nok ind når vi et punkt, hvor væsken er individuelle partikler, der opfører sig meget anderledes end den overordnede væske formlerne beskriver. For at have et diskret væskesystem er der to metoder der er brugt oftest. Såkaldte feltbaserede metoder og partikelbaserede metoder. Vi kommer til at implementere to algoritmer i programmer vi skriver, en feltbaseret og en partikelbaseret metode. Vi starter med at se på den feltbaserede metode.

Feltbaseret[1,4]

Til den feltbaserede metode består væskesystemet af et stort gitter eller felt med celler, hvor hvert celle i feltet har en samling værdier. Hvert celle har værdierne defineret ud fra centrummet af cellen, og svarer til et gennemsnit af væskens egenskaber inden for cellen. Størrelsen og mængden af celler bestemmer hvor præcis simuleringen af væsken er, dvs. hvor fin en opløsning man får. Da tid også skal være diskret bliver der taget faste "timesteps" Δt. Et mindre timestep giver en bedre tidsopløsning, og gør simuleringen yderligere præcis, på bekostning af at flere steps skal tages før en vis mængde tid er gået i simuleringen, altså at den bliver langsommere.



Figur 2, De diskrete felters celler har deres værdier defineret ud fra deres centrum. Ophavsret: Jos Stam, fra "Stable Fluids" (1999)[4]

Ud fra gitteret er der flere felter, blandt andet vektorfeltet hastighedsfeltet. For at simulere væskesystemet itereres der en algoritme med en serie funktioner. Hver celle i domænet bliver betragtet for at opdatere dens værdier. Hvert iteration over et timestep giver væsken en ny tilstand, med alle felterne opdateret, som der så kan laves en ny iteration ovenpå. Tiden det tager at beregne en iteration er hvad vi er interesserede i når det gælder de tests vi vil lave på programmet.

Partikelbaseret[5]

Til den partikelbaserede metode bruges konceptet om Lagrange partikler, der beskriver en bestemt metode for at beregne positionen af partikler i en væske. Der er mange forskelle mellem den feltbaserede og den partikelbaserede metode. Måden væsken gøres diskret på i denne metode er ved at have væsken være en samling partikler. Væske i virkeligheden er også en samling partikler, men metoden her simulerer ikke enkelte molekyler. Partiklerne repræsenterer dele af væsken på samme måde som at et felt repræsenterer væsken i feltet. Der er dog nogle store fordele ved den partikelbaserede metode. Der skal ikke tages specielt hensyn til massebevarelse som med den feltbaserede metode, da hvert partikel svarer til en del af væskens samlede masse, og da partiklerne hverken forsvinder eller bliver skabt (medmindre det er en ønsket effekt). Der er også mulighed for områder ikke at have væske i. Det lyder måske lidt åbenlyst, men i den feltbaserede metode vi har talt om, er densiteten af væsken den samme over hele simuleringsområdet, og der er derfor altid den samme mængde væske over det hele. I stedet for at hvert felt itereres på, itereres der på hvert partikel i væsken. Der beregnes ændringer i hastighed og position baseret på partiklerne umiddelbart omkring det partikel man kigger på. Det har den fordel at man kan have et meget stort simulerings domæne, uden at bruge så meget simuleringstid, som man ville blive nødt til med den feltbaserede metode. Den partikelbaserede metode skalerer nemlig simuleringstid på antallet af partikler, i modsætning til antallet af felter for den feltbaserede metode.

Programmet

Til den feltbaserede metode kommer selve algoritmens design af den feltbaserede metode til at være stærkt inspireret af YouTube videoen "Coding Challenge #132: Fluid Simulation", 2019, af The Coding Train[6]. Den bygger på en anden artikel, "Real-Time Fluid Dynamics for Games", 2003, af Jos Stam[1], der er en efterfølger af "Stable Fluids", 1999.[4]

Til den partikelbaserede metode kommer programmet til at være baseret på pseudokoden der beskriver algoritmen, i "Interactive 2D Particle-based Fluid Simulation for Mobile Devices" (2013), af Daniel Månsson.[5]

Design

Designet af programmet skal facilitere simuleringerne og tests af simuleringerne. Programmet skrives i Processing (Java), da jeg har erfaring i det og da den feltbaserede algoritme er blevet implementeret i Processing allerede. For bedst at besvare problemformuleringen og få nogle gode testresultater, måles der på nogle få metrikker hvorefter visse forhold kan analyseres mellem dem. Tid er en stor faktor. Præcision er også en stor faktor. Der vil derfor ses på hvor lang tid det tager at beregne en enkel iteration, eller et enkelt timestep rettere sagt. Antallet af felter/partikler bliver også taget i betragtning, da det er direkte korreleret med præcisionen af simuleringen. Der vil også laves en visuel analyse af programmets output, for at dømme forskelle i præcision.

Et separat program skrives til den partikelbaserede metode, hvor algoritmerne implementeres baseret på den pseudokode beskrevet i den videnskabelige artikel.[5] Programmet kommer til at have struktur baseret på hvad der bedst vil give mening ud fra algoritmerne beskrevet i artklen. De samme metrikker som i det feltbaserede program vil også blive målt og testet på.

Dokumentation

Feltbaseret

Det feltbaserede program brugte programmet fra "Coding Challenge #132: Fluid Simulation"[6] som basis, men der er lavet store tilføjelser og ændringer. For at kunne teste programmet mest effektivt måtte programmet omskrives. Det hele skulle kunne instantieres for at kunne testes nemmest, og da programmet originalt var skrevet med en masse globale variabler skulle disse parameteriseres. En ny klasse blev oprettet for at holde på koden der egentlig lå i sketchens hoved klasse. De forskellige globale variabler blev derefter omskrevet til at være parametre i konstruktørerne for de tre klasser fra basis programmet. På den måde kunne hele væskesimuleringen blive lavet på ny når som helst, og med et nyt sæt parametre.

For at få et godt sæt data fra programmet blev der lavet en test metode. For at få mange datapunkter besluttede jeg mig for at lave 100 tests, hvert med en opløsning af simuleringen på test#*10. 100 datapunkter tænkte jeg ville være rigeligt til at kunne komme frem til et ordentligt resultat. At opløsningen skulle springe med 10 pixels på hver led hver test, var da jeg tænkte at det ville give nogle mere informative resultater. Forskellen på 37x37 og 38x38 er ikke særlig stor, 30x30 og 40x40 er en markant større forskel, og ville give mere indsigt over de 100 datapunkter. For at få nogle mere repræsentative data, ville tiden det tager at simulere en enkelt frame være et gennemsnit. De første 100 frames af simuleringen for hver opløsning ville få simuleringstiden skrevet ned og derefter taget gennemsnittet af. I slutningen ville programmet spytte en .csv fil ud, der derefter kunne bearbejdes i et program som Excel eller GeoGebra.

Originalt var ideen at gennemsnittet af simuleringstiden skulle være i et givent tidsrum, som 10 sekunder. Denne fremgangsmåde ville have fordelen at hele testen ville tage en fast mængde tid, nemlig 1000 sekunder i alt. Problemet var at de senere tests ville få meget færre værdier at tage gennemsnittet af end de tidligere tests, da hver frame ville tage markant forskellige tider at beregne. De tests med meget små opløsninger som 30x30 eller 80x80 ville nemt kunne få 1000 frames på de 10

sekunder, hvor en test som 900x900 ville få måske 20 frames i alt. Jeg endte derfor med at vælge et fast antal frames.

Partikelbaseret

Programmet med den partikelbaserede metode var svært at skrive. Algoritmen til den partikelbaserede metode var ikke skrevet på forhånd, men bestod kun af pseudokode.[5] Pseudokoden kunne følges, men var besværlig at skrive efter, da oversættelsen til datastrukturer i Processing var ret besværlig. Der bliver ofte refereret til partikel objekter der har positions- og hastigheds vektorer. Da forskellige metoder i den indbyggede PVector klasse i processing, opererer og ændrer selve vektoren, skulle man passe ekstra godt på.

$$\mathbf{v}_{p,n} \leftarrow n.\mathbf{pos} - p.\mathbf{pos}$$

Figur 3, pseudokode fra den videnskabelige artikel[5]

Her ses en del af pseudokoden, der beskriver en vektor $\boldsymbol{V}_{p,n}$ der bliver sat til en ny vektor der svarer til partikel p's positionsvektor trukket fra partikel n's vektor. I programmet er den indbyggede klasse PVector brugt til diverse vektorer der må være brug for. I oversættelsen af pseudokoden ville man nemt tænke på at bruge PVector.sub() til at trække den ene vektor fra den anden. Altså:

$$\mathbf{v}_{p,n} = n.\mathbf{pos}\ \mathrm{u}(p.\mathbf{pos})$$

Problemet med denne linje kode er dog at dette ville ændre n.**pos**. En sideeffekt vi ikke er interesserede i, da vi kommer til at skulle bruge n.**pos** senere i programmet. Der kunne man i stedet bruge .copy() metoden i PVector til at få en kopi, så man ikke ændrer den egentlige positionsvektor.

Et andet problem der skulle løses er hvordan objekter bliver behandlet i Processing, eller rettere sagt Java. Tager vi et bredere kig på pseudokoden fra før kan vi se et nyt problem.

```
1:  function ApplyViscosity(timeStep)
2:      for each Particle p
3:          for each Particle n ∈ neighbors_p
4:              v_{p,n} ← n.pos − p.pos
5:              vel_inward ← (p.vel − n.vel) · v_{p,n}
6:              if vel_inward > 0
7:                  length ← |v_{p,n}|
8:                  vel_inward ← vel_inward/length
9:                  v_{p,n} ← v_{p,n}/length
10:                 q ← length/radius
11:                 I ← 0.5 ∗ timeStep ∗ (1 − q) ∗ (σ ∗ vel_inward + β ∗ vel²_inward) ∗ v_{p,n}
12:             p.vel ← p.vel − I
```

Figur 4, pseudokode fra den videnskabelige artikel. Algoritme 3 fra underkapitlet ApplyViscosity.[5]

Pseudokoden henviser til at man itererer gennem en liste af partiklerne, og en liste af naboerne til partiklerne. Bruger man en klassisk for-each løkke, som 'for (Partikel p : partikler)', får man dog et problem længere nede. På linje 12, markeret med grønt, ændres partiklet p's hastighedsvektor. Problemet stammer fra forskellen mellem 'pass by value' og 'pass by reference'. for (Partikel p : partikler) giver p som en kopi af partiklet, altså det giver os værdien, og er dermed 'pass by value'. Det betyder at når vi prøver at ændre p.**vel** så ændrer vi ikke det rigtige partikels vektor, men kopiens vektor hvilket vi ikke ønsker. Dette problem løses ved at bruge en standard for-løkke. Der bruger vi

for-løkkens variabel, i f.eks., sammen med partikel listens .get() metode, til at få en reference til partiklen p. Ulempen ved at gøre det sådan er at der skal skrives mere, da vi skal kalde .get() af listen og eventuelt give referencen til et nyt variabel. Fordelen ved at gøre det på denne måde er at det faktisk virker, hvilket har en højere prioritet.

Analyse af data og algoritmer

Programmerne er skrevet, og der er blevet målt på simuleringstiden af algoritmerne. Der er blevet taget mange billeder, og lavet videoer af programmerne, men hvad kan det egentlig fortælle os om noget som helst? Ved at se på programmerne kan vi få en bedre forståelse for hvordan de kan bruges og eventuelle problemer med dem. Vi kommer til at se på tidskompleksiteten, de målte simuleringstider og programmerne visuelt.

Tidskompleksitet[7]

Tidskompleksitet er et begrebder bruges til at beskrive hvordan tiden det tager at køre en algoritme stiger, i forhold til et eller flere variabler. Det kan være meget relevant at kigge på tidskompleksitet, da det fortæller hvordan en algoritme skalerer. Et eksempel kan være hvis vi har en algoritme der sorterer en liste. Det kan godt være at det går stærkt når der er 10 elementer i listen, men hvordan udvikler tiden det tager at køre algoritmen, baseret på listens længde? Dette spørgsmål svarer en tidskompleksitetsanalyse på. En måde tidskompleksitet kan beskrives på er ved hjælp af 'store O notation'.

Store O notation[7,8]

Store O notation er et emne inden for matematik på højt niveau. Inden for datalogi og tidskompleksitet kan det simplificeres til at mene: Med hvilken funktion beskrives væksten af en funktion, når x bliver meget stor. Vi kigger altså efter essensen af væksten af en funktion når x bliver meget stor. Vi gør dette ved kun at beholde de led der har størst betydning for væksten af en funktion. Eksempel:

$$f(x) = 2x^3 + 10x^2 + 5 \Rightarrow$$

$$O(x^3)$$

Idet x bliver meget stort, kommer $10x^2$ og 5 leddene ikke til at være relevante, deres indflydelse bliver ubetydelig. Da faktoren 2 i det første led, ikke er afhængigt af x, kan det også fjernes. Dette giver os essensen af hvordan f stiger. Dette er dog kun en simplificering.

Algoritmer grupperes normalt efter hvilken type af funktion der ses i O(). De mest standard grupper er:

- O(1) Konstant
- O(log n) Logaritmisk
- O(n) Lineær
- O(n log n) Logartimisk Lineær
- O(n^c) Polynomiel
- O(c^n) Eksponentiel

For at bestemme tidskompleksiteten af algoritmerne i vores programmer, går vi gennem koden, og skriver ned hvilken effekt det valgte variabel har på hvor mange gange noget gøres i programmet.

Feltbaseret tidskompleksitet[7]

For begge programmer kommer tidskompleksiteten til at blive bestemt ud fra kildekoden, og hvad algoritmen foretager sig for at lave et simulerings step. Der kigges på antallet af felter, som værende n, i koden skrevet som N. Vi starter med at tælle alle effekter af n sammen, hvorefter vi gør som i eksemplet og laver det om til store O notation. Et enkelt step bliver beregnet ved at kalde metoden .step() i fluid klassen, så det er der vi starter med at kigge. Vi kalder den løbende funktion for tid det tager at beregne, f(n).

```
void step() {
  int N          = this.size;
  float visc     = this.visc;
  float diff     = this.diff;
  float dt       = this.dt;
  float[] Vx      = this.Vx;
  float[] Vy      = this.Vy;
  float[] Vx0     = this.Vx0;
  float[] Vy0     = this.Vy0;
  float[] s       = this.s;
  float[] density = this.density;

  mF.diffuse(1, Vx0, Vx, visc, dt);
  mF.diffuse(2, Vy0, Vy, visc, dt);

  mF.project(Vx0, Vy0, Vx, Vy);

  mF.advect(1, Vx, Vx0, Vx0, Vy0, dt);
  mF.advect(2, Vy, Vy0, Vx0, Vy0, dt);

  mF.project(Vx, Vy, Vx0, Vy0);

  mF.diffuse(0, s, density, diff, dt);
  mF.advect(0, density, s, Vx, Vy, dt);
}
```

I step metoden bliver der instantieret nogle variabler, og der bliver kaldt en række metoder en efter en. Alt dette gøres én gang, det er derfor konstant. Vi kan tilføje dette til f som:

$$f(n) = 1$$

Vi fortsætter ned i metoderne der kaldes; diffuse.

```
void diffuse (int b, float[] x, float[] x0, float diff, float dt) {
  float a = dt * diff * (N - 2) * (N - 2);
  lin_solve(b, x, x0, a, 1 + 4 * a);
}
```

Diffuse kalder en metode yderligere; lin_solve():

```
void lin_solve(int b, float[] x, float[] x0, float a, float c) {
  float cRecip = 1.0 / c;
  for (int k = 0; k < iter; k++) {
    for (int j = 1; j < N - 1; j++) {
      for (int i = 1; i < N - 1; i++) {
        x[IX(i, j)] =
          (x0[IX(i, j)]
          + a*(    x[IX(i+1, j)]
          +x[IX(i-1, j)]
          +x[IX(i, j+1)]
          +x[IX(i, j-1)]
          )) * cRecip;
      }
    }

    set_bnd(b, x);
  }
}
```

Ud fra metoden kan vi se to for-løkker inden i hindanden, hvor hvert for-løkke skalerer direkte baseret på N. Inden for for-løkkerne bliver metoden IX kaldt, så vi tager og kigger på den.

```
int IX(int x, int y) {
  x = constrain(x, 0, N-1);
  y = constrain(y, 0, N-1);
  return x + (y * N);
}
```

IX skalerer ikke baseret på N, så vi kan gå tilbage til lin_solve(). Vi gør altså noget baseret på N gange N, på grund af de to for-løkker. Vi kan derfor tilføje n^2 til f.

$$f(n) = n^2$$

Da n^2 nu er den del der gør mest, kan vi ignorere alle led der gør mindre. Dvs. hvis en del af algoritmen, kun skalerer lineært med n, medtages den ikke. Uden for for-løkkerne kaldes set_bnd().

```
void set_bnd(int b, float[] x) {
  for (int i = 1; i < N - 1; i++) {
    x[IX(i, 0  )] = b == 2 ? -x[IX(i, 1  )] : x[IX(i, 1 )];
    x[IX(i, N-1)] = b == 2 ? -x[IX(i, N-2)] : x[IX(i, N-2)];
  }
  for (int j = 1; j < N - 1; j++) {
    x[IX(0, j)] = b == 1 ? -x[IX(1, j)] : x[IX(1, j)];
    x[IX(N-1, j)] = b == 1 ? -x[IX(N-2, j)] : x[IX(N-2, j)];
  }

  x[IX(0, 0)] = 0.5f * (x[IX(1, 0)] + x[IX(0, 1)]);
  x[IX(0, N-1)] = 0.5f * (x[IX(1, N-1)] + x[IX(0, N-2)]);
  x[IX(N-1, 0)] = 0.5f * (x[IX(N-2, 0)] + x[IX(N-1, 1)]);
  x[IX(N-1, N-1)] = 0.5f * (x[IX(N-2, N-1)] + x[IX(N-1, N-2)]);
}
```

Der er to for-løkker, de skalerer lineært med N. De er dog ikke inde i hindanden, så det svarer til lineær skalering i alt inden for denne metode. Tænker vi over hvor vi kommer fra, step(), diffuse(), lin_solve(), set_bnd(). Vi er ikke inde i en for-løkke baseret på N, og derfor har set_bnd() kun en lineær effekt på tidskompleksiteten. Vi ignorerer derfor dens effekt, og fortsætter op fra hvor funktionen blev kaldt. VI kommer tilbage til lin_solve(), vi er også færdige her, vi fortsætter tilbage til diffuse(), også færdige her, vi fortsætter tilbage til step(), og fortsætter videre til de næste metoder der kaldes.

Dette er processen, for den fulde proces, se Bilag 1.

Vi er færdige med processen og er kommet frem til at tidskompleksiteten er $O(n^2)$.
Tidskompleksiteten er godt nok $O(n^2)$, men det er ikke repræsentativt i forhold til hvor lang tid det egentligt tager. Der var mange dele af algoritmen der var n eller n², og som ikke bidrog specielt til tidskompleksiteten. Det tager vi et kig på nå vi ser på de målte data.

Partikelbaseret tidskompleksitet

Den partikelbaserede algoritme virker på en helt anderledes måde, så tidskompleksiteten kunne være markant anderledes end den feltbaserede metode. Vi kommer til at se på den partikelbaserede metodes tidskompleksitet hvor n er antallet af partikler. Se Bilag 2 for processen hvor tidskompleksiteten af algoritmen bestemmes. Tidskompleksiteten er $O(n^3)$.

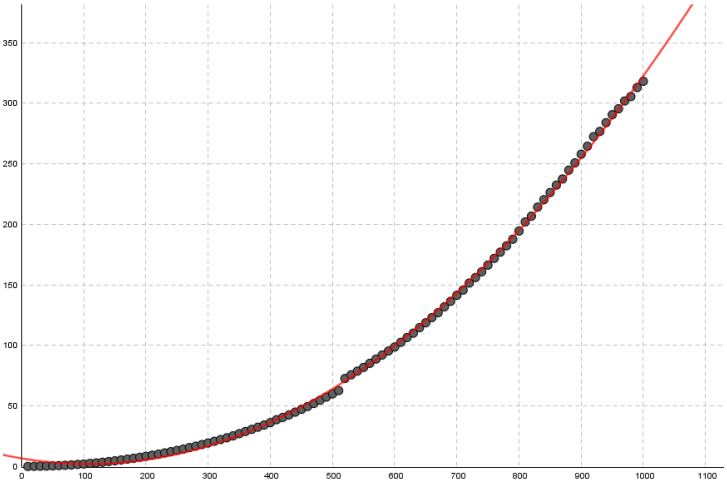
Dataene

Tidskompleksiteterne kan ikke alt. Vi kan ikke sige noget om hvor lang tid det tager at simulere et enkelt step med den feltbaserede metode. 4 millisekunder? 1 sekund? 23 minutter? Intet. Det giver os kun en indikation om hvordan tiden skalerer. Hvor effektiv algoritmen er. Har vi en ide om hvad tiden er for en bestemt n, f.eks. 100, så kan vi sammenligne. Så ville vi kunne lave forudsigelser for hvor lang tid det ville tage at køre et step for andre n, som vi ikke har målt på. Ved hjælp af tidskompleksiteten kan vi også sammenligne med andre algoritmer.

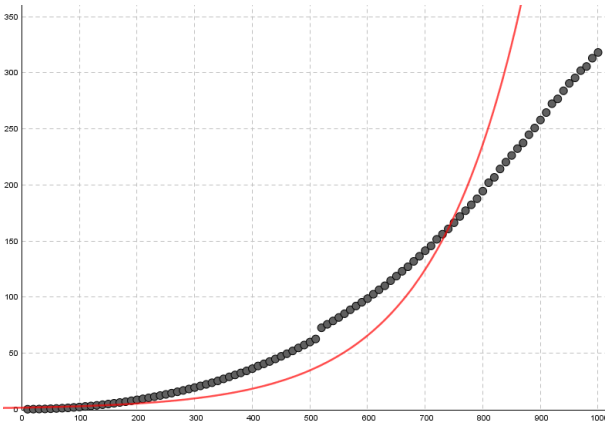
Vi tager datasættene og kører regression på dem. Vi regner med at de polynomielle regressioner kommer til at passe bedst, da algoritmerne begge to har tidskompleksitet der falder i den polynomielle kategori. Vi skal ved hjælp af regression kunne se at henholdsvis et andengradspolynomie og et tredjegradspolynomie bedst passer til den feltbaserede og den partikelbaserede algoritme. Vi tjekker hvor godt regressionen egentlig passer ved at kigge på hvad den siger om ukendte værdier, og ved så at måle de egentlige værdier, ved hjælp af programmet. Vi bruger GeoGebra til at lave regression og til visuelt at repræsentere graferne.

Den feltbaserede metode

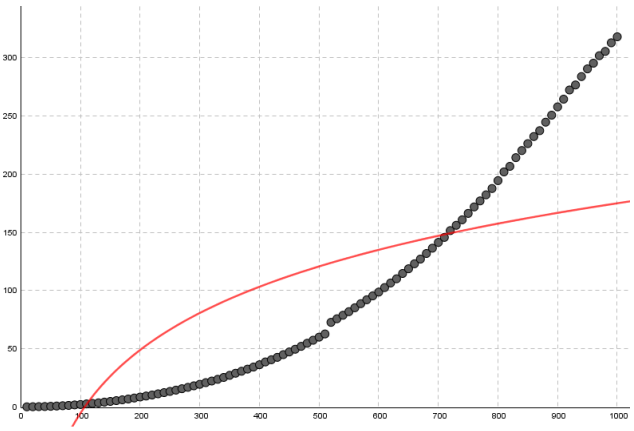
Figur 7 viser simuleringstiden i millisekunder ud fra opløsningen af simuleringen, altså antallet af felter på hver led. Den røde linje er polynomiel regression med et andengradspolynomie. I modsætning til en eksponentiel eller en logaritmisk regressionsmodel som set på figur 5 og 6, så passer den andengradspolynomiet. Nu til at bestemme hvor godt modellen passer. Datapunkterne går fra 10 til 1000, felter i hvert retning. Vi prøver derfor at måle og beregne for 1500, 2000 og 3000.



Figur 7, punkt plot med de 100 tests af det feltbaserede programs simulerings tid, med andengrads polynomiel regression



Figur 5, graf med eksponentiel regressionsmodel



Figur 6, graf med logaritmisk regressionsmodel

Modellens forskrift er:

$$f(x) = 0{,}4\ 0\ 21\ 0^{3}\ x^{2} - 0{,}0\ 8\ 6x3 + 6{,}6\ 3$$

Med enheder:

$$f(x) = 0{,}4\ 0\ 21\ 0^{3}\ ms \cdot f\ et^{-2}\ x^{2} - 0{,}0\ 8\ 6ms \cdot f\ et^{-1}\ x + 6{,}6\ 3ms$$

Da x er antallet af felter i hver retning og har enheden felt, og da y-aksens enhed er millisekunder. Baseret på de tre x valgt til at teste modellen er resultatet følgende:

$$f(1\ 5\ 0) = 7\ 8\ 2ms$$

$$f(2\ 0\ 0) = 1\ 4\ 4m0s$$

$$f(3\ 0\ 0) = 3\ 3\ 7m0s$$

Nu for at teste tallene, tager vi gennemsnittet af de første 20 frames simuleret, ved hvert opløsning.

$$1\ 5\ 0 - 04\ 7\ ms$$

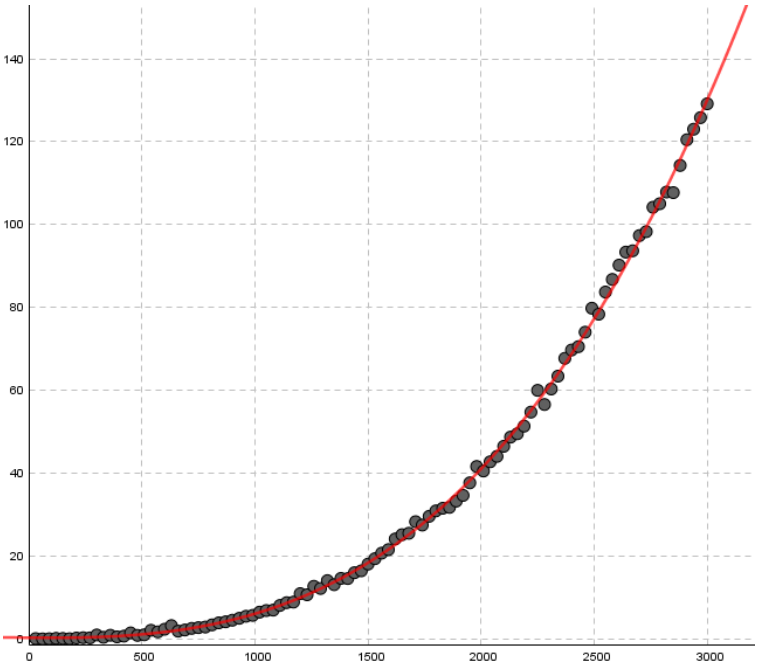
$$2\ 0\ 0 - 08\ 1\ 8ms$$

$$3\ 0\ 0 - 01\ 8\ 3m1s$$

Tallene matcher ikke. Modellen forudsete at tiderne ville være næsten dobbelt så store som de endte med at være. Skaleringen må derfor være anderledes end først tænkt. Modellen passer dårligt.

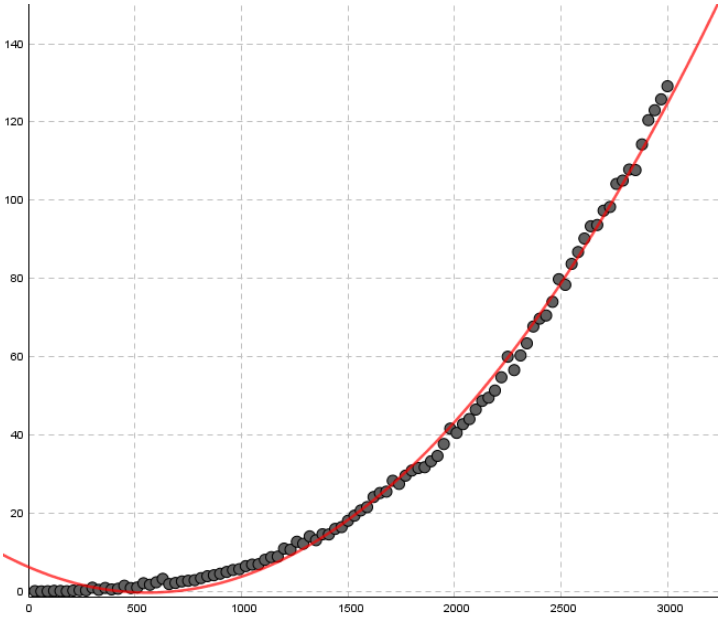
Den partikelbaserede metode

Vi bruger samme metoder som før. På figur 8 ses dataene fra måling af simuleringstid af programmet med den partikelbaserede metode. Den røde tendenslinje er en regressionsmodel med et tredjegradspolynomie. På x-aksen er antallet af partikler, og går fra 30 til 3000 i 30 partiklers skridt. Regressionsmodellen passer meget godt, men det gør en tilsvarende regressionsmodel med en andengradsligning også. Se figur 9.



Figur 8, graf over data fra det partikelbaserede program med en regressionsmodel.

Det passer meget godt. R^2 værdien for den første regression er 0,9991, mens den kun er 0,9959 for den anden regression. De er begge gode men den første regression, tredjegrands regressionen, passer bedst. Vi kan bekræfte at vi kom frem til den korrekte tidskompleksitet af programmet ved at sammenligne de to regressioner, og se hvilken der giver forudsigelser tættest på de egentlige målte værdier, for nogle testværdier for x. Vi gør ligesom før, dog med nye testværdier på 5000, 10000 og 15000 partikler. Værdierne for andengrads regressionen:



Figur 9, graf med en andengradspolynomiel regression

$$f_2(5\ 0\ 0) = 4\ 1\ ms$$

$$f_2(1\ 0\ 0) = 1\ 8\ 7ms$$

$$f_2(1\ 5\ 0) = 4\ 3\ 7ms$$

Værdierne for tredjegrads regressionen:

$$f_3(5\ 0\ 0) = 5\ 7\ ms$$

$$f_3(1\ 0\ 0) = 4\ 3\ 6ms$$

$$f_3(1\ 5\ 0) = 1\ 4\ 5\ ms$$

Vi simulerer de første 10 frames, hvorefter gennemsnittet tages. De tager markant længere tid at beregne per frame end den feltbaserede metode, så vi tager kun gennemsnittet af 10.

$$5\ 0\ 0 \rightarrow 6\ 3\ ms$$

$$1\ 0\ 0\ 0 \rightarrow 5\ 0\ 4ms$$

$$1\ 5\ 0\ 0 \rightarrow 1\ 7\ 8\ ms$$

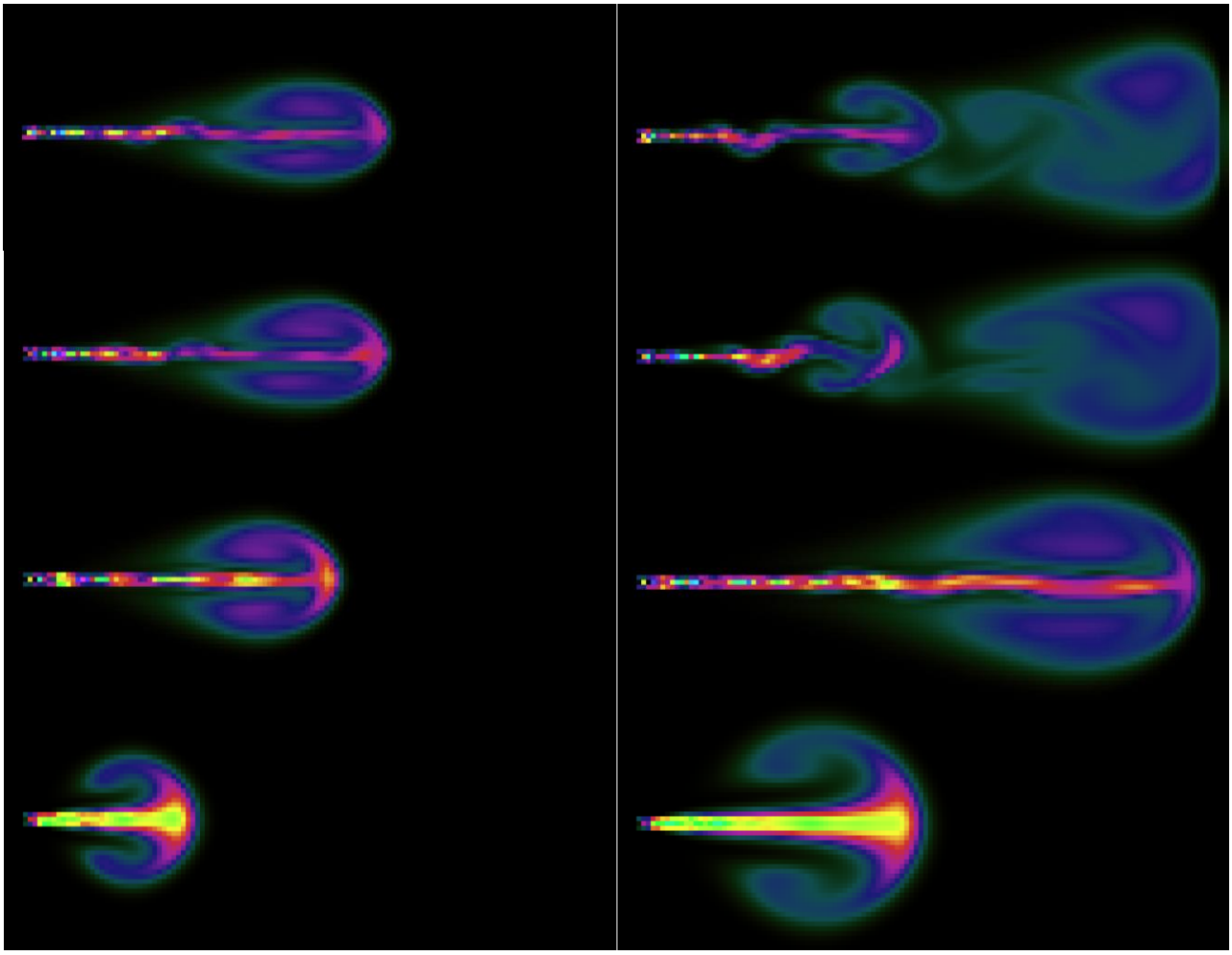
Vi kan se at tredjegrads regressionen er betydeligt tættere på de rigtige tal, end andengrads regressionen. De er begge ok tæt på den første værdi, men derefter er andengrads regressionen ikke engang tæt på. Tredjegrads regressionen er ikke perfekt, men den følger meget godt med, og er langt bedre end andengrads regressionen. Det bekræfter at tidskompleksitetsanalysen passer.

Visuel analyse[9]

Virker simuleringen faktisk, eller er det skrammel? Det er dette spørgsmål vi er ude på at besvare i dette afsnit. Et andet spørgsmål vi kan bruge til at hjælpe os er: Ser det ud som rigtig væske? Hvilket så giver et nyt spørgsmål: Hvordan ser væske egentlig ud? Svaret på dette spørgsmål er: turbulent. Væske der flyder rundt er ikke fint og ordentligt men turbulent. Vi kan også se på hvilken grad af præcision vores simulering har, ved hjælp af at ændre et eller flere parametre og se hvordan væsken opfører sig bagefter. Til hjælp med dette kan vi bruge konceptet om laminart og turbulent flow samt Reynolds tallet.

Ændringer i viskositet

Som nævnt tidligere bliver flowet mindre turbulent idet Reynolds tallet falder, og når det er faldet tilstrækkeligt er flowet laminart.[3] Dette kan gøres ved at hæve viskositeten. I simuleringerne er viskositeten et parameter som algoritmerne arbejder ud fra. For at vise viskositetens effekt på flowet tages en serie billeder af simuleringer med forskellig viskositet på faste tidspunkter efter start.



Figur 10, billeder af simuleringer med forskellig viskositet. Taget efter 100 frames på venstre side, og efter 200 frames på højre side.

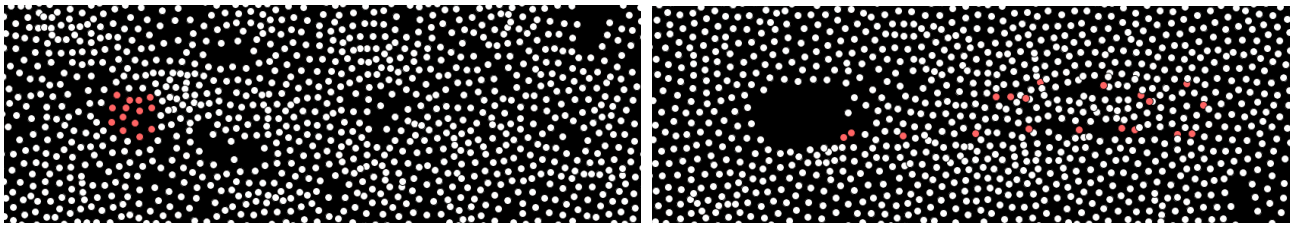
Farverne repræsenterer et 'farvestof' der bliver tilføjet i et felt. Farven bruges til at visualisere hvor det her farvestof ender henne, og hvordan væsken bevæger sig. Hvert flow fra det øverste billede har en viskositet på 10 gange det flow over det. Så det nederste flow har en viskositet på 1000 gange det øverste flow. Vi kan se hvordan 'formen' af væsken er anderledes alt efter hvilken viskositet væsken

har. Vi kan tydeligt se det turbulente og kaotiske flow i de øverste billeder. Det næstnederste flow er ikke særlig turbulent, men man kan stadig ane at det er en smule turbulent. Det nederste flow er dog laminart. Det kan ses på symmetrien af hvordan farven i væsken har bredt sig; en symmetri der ikke ville være mulig hvis flowet var turbulent. Programmet og den feltbaserede algoritme afspejler virkeligheden, og reagerer realistisk på ændringer i karakteristika af væsken der burde have indflydelse på flowet.

Se #9 i referencer for at se små videoer (GIFs) hvor flowet kan ses.[9]

Den partikelbaserede metode

Se video [1] af den partikelbaserede metode, for at få et bedre overblik over dette afsnit.[9] Vi ser på den partikelbaserede metode og viser billeder af flowet henholdsvis 100 og 200 frames efter start.



Figur 11, billeder af den partikel baserede metode efter 100 og 200 frames. De røde partikler begyndte at accelerere ved frame 100.

De røde partikler, er partikler der er blevet accelereret fra et bestemt område. Flowet af den partikelbaserede metodes simulering er meget anderledes end af den feltbaserede metodes. Vi kan derfor ikke lave samme test som på det feltbaserede program. Ved at se på videoen [1][9] af flowet, kan vi over længere tid se hvordan partiklerne reagerer på forstyrrelsen der sker når de røde partikler accelereres. Det er langsommere og mindre direkte. Følelsen af flowet af simuleringen, er som hvis man zoomede langt ind på den feltbaserede løsning, og havde betydeligt mindre timesteps. Jeg prøvede at ændre på parametrene, men det hjalp ikke. Da jeg prøvede med større time steps, vibrerede alle partiklerne underligt, og flowet virkede ikke rigtig. Ændringer i viskositeten har heller ikke de konsekvenser som jeg havde regnet med. Programmet kører også meget langsomt, med det antal partikler der ses (4000). Validiteten af flowet er derfor meget svært at bekræfte.

Diskussion

Programmerne viser ikke algoritmerne i det bedste lys. Det er begrænset hvor godt jeg kan skrive programmet og hvor effektive algoritmerne er. Dette skal man tænke over når man ser på validiteten af det hele. Algoritmerne kørte ikke specielt hurtigt. Gif [1][9] tog over 10 minutter at få samlet nok frames til, og der er kun 600 frames i den gif. Det tog ca. 1 sekund pr. frame. Det var med 4000 partikler. Den feltbaserede metode virkede og kørte meget bedre end den partikelbaserede metode. Der var mange problemer i processen af at lave det partikelbaserede program. Et af problemerne var at jeg skulle lave et program, der stod som basis for et helt bachelors projekt, uden at have særlig meget tid til det. Programmet jeg skrev kunne godt optimeres, der gøres meget brug PVector klassen, og metoder der ikke er specielt effektive. Et af de andre problemer var hvordan der manglede information, udover selve algoritmerne var der kun beskrivelser af resten af programmet. Der nævnes en grid klasse i pseudokoden, men den er aldrig vist.[5] Jeg måtte skabe grid klassen, med al dens funktionalitet ud fra beskrivelserne af hvad de andre metoder skulle bruge griddet til. De forskellige

tal for konstanter var heller ikke nævnt, og det var svært at finde tal til de forskellige konstanter for at få partiklerne til at opføre sig som en væske.

Hvor gode målingerne fra programmerne er, er også usikkert. Der er mange ting der kunne have haft en effekt på programmerne. Ting som overhead eller underlige quirks i hukommelseshåndtering eller et andet program på min computer der lige pludselig skulle til at køre i baggrunden. Der er mange forskellige fejlkilder, resultaterne kunne påvirkes af. En af måderne hvorpå de fleste af de her problemer kunne elimineres på, var ved at tage et gennemsnit af simuleringstiden. Ved at køre programmet i et længere stykke tid, og tage gennemsnittet, ville eventuelle kortvarige påvirkninger ikke rigtig kunne føles. Dog var der dog grænser for hvor lang tid der kunne tages målinger hen over. Det største antal frames jeg tog gennemsnittet over var 500 frames. Det tog en evighed at få kørt alle de tests igennem, over 45 minutter. Det gav dog også de bedste resultater, og var klart det værd.

Det bringer os til algoritmernes egentlige formål. Ikke 100% naturlig realisme, men flow der er 'godt nok' og ser ud som rigtig væske. Der laves antagelser for at simplificere algoritmerne, og for at gøre dem hurtigere. Begge algoritmer kommer fra videnskabelige artikler der taler om brug af fluidsims inden for computerspil eller simpel brugerinteraktion på mobiler eller tablets. Succesen af programmerne lå i at være 'gode nok', og det kan man nemt sige at det feltbaserede program opnåede.

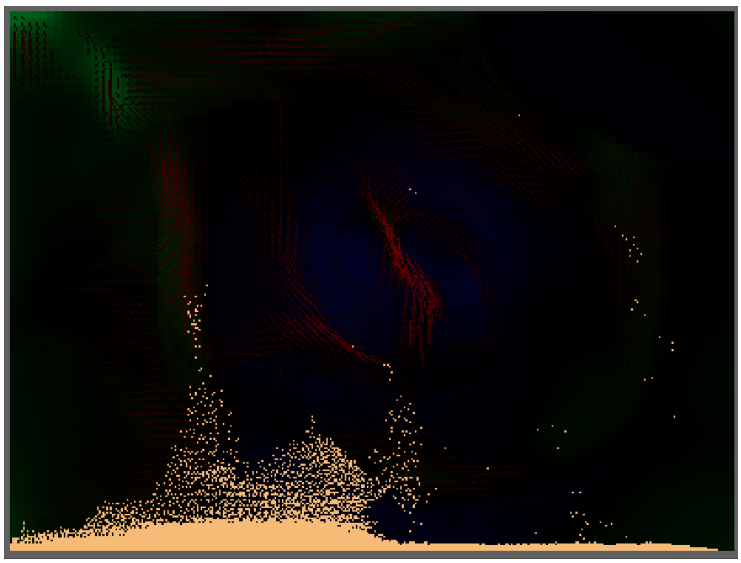
Gav tidskompleksitetsanalysen mening? Ja, det gjorde den. Hvordan algoritmerne skalerer kan fortælle os meget om algoritmernes eventuelle brug. At den feltbaserede metode kun skalerer kvadratisk fortæller os om hvordan den kunne være et alternativ til andre algoritmer. I et miljø hvor meget store væskesystemer skal simuleres kan algoritmen have en fordel. Hvor præcision ikke er det vigtigste kan algoritmerne bruges. I underholdningsindustrien er det oplagt, og ikke bare i computerspil. Til visuelle effekter i film eller andre medier kan væskesimuleringer bruges til stort effekt. Der er det jo ikke vigtigt at simuleringerne er præcise, men at de ser godt ud, og at de ikke tager for lang tid at køre. Jo nemmere simuleringerne er at køre, desto bredere bliver gruppen af eventuelle brugere. Hvis man kan køre simuleringer på sin bærbar kan hvem som helst få pæn væskedynamik med i sine små YouTube videoer. Inden for industri hvor præcision er vigtigt kan disse typer algoritmer også bruges. De komplicerede og tunge algoritmer der giver præcise resultater er muligvis ikke det værd at køre for visse opgaver og projekter. De simplere lettere algoritmer kan altså give mulighed for at få en ide om hvad der kan forventes når de store simuleringer bliver kørt. Dette kunne gøre det nemmere og hurtigere at lave prototyper, hvis man kunne køre en simpel simulering for at se om et koncept kunne fungere. Der kan bruges hurtige upræcise, hurtige fluidsims i mange industrier.

Perspektivering

Computerspilsindustrien er kæmpestor, og den fortsætter med at vokse. Der er mange typer af computerspil og tilsyneladende uendeligt mange spil af hver kategori man kunne tænke sig. I computerspil finder man derfor mange forskellige slags teknologier brugt på alle mulige måder. Der er selvfølgelig det klassiske at kunne skyde, gå eller se. Men mange andre teknologier findes også i computerspil, og endda teknologier som måske burde ses noget oftere. En af disse teknologier er simuleringer af væskedynamik; fluidsims.

Der er mange forskellige computerspil, og de laves til mange forskellige slags mennesker, men hvad er det egentlig der er et computerspils mål? For firmaet der laver dem; at sælge. Men for forbrugeren; at have det sjovt. Fluidsims hører hjemme i computerspil da teknologien kan hjælpe med at opnå begge mål. Fluidsims har potentiale i forhold til at være en central del af et spil. Simple arkadespil kan bygges omkring flowet af en væske, hvor man skal sejle rundt, eller opnå forskellige mål. Det kaotiske ved turbulent flow kan bruges til at gøre et spil interessant, ved at være forskelligt og uforudsigeligt hver gang man spiller spillet. Det ville også være muligt at bruge fluidsims til at få en bedre følelse af et våben f.eks. Man kunne have en vandkanon, eller en haveslange. Skylle ting rundt, og se vandet kollidere og interagere med miljøet på en realistisk måde. Fluidsims har dog også meget stort potentiale i baggrunden. Idet spil søger at øge realismen kan fluidsims hjælpe. I stedet for at spillet er bygget rundt om en simulering, kan en fluidsim sidde i baggrunden. Det kunne være en vandpyt, som en soldat ender med at træde i, eller flammen fra den fakkel helten holder. Der er mange steder hvor fluidsims kan hjælpe computerspil blive mere fordybende. Idet realismen stiger bliver det nemmere at leve sig ind i verdenen, og fluidsims har potentialet for at hjælpe med dette.

Der ses dog ikke særlig mange spil der gør brug af fluidsims. Et af de mere kendte spil er et gammel internet spil kaldet Powder Game.[10] Det gør brug af en lignende type feltbaseret algoritme, som den vi implementerede i programmet med den feltbaserede metode. I spillet er det en central del, at der er en væske, der kan blæse ting rundt. Spillet er af genren, sandbox, og målet med spillet er derfor at prøve spillets forskellige funktioner sammen med de andre funktioner for at se hvilke ting der mon sker. Spillet tillader meget kreativitet, og bruger den her fluidsim som en central del af det.



Figur 12, Powder Game (2007)[10]

En af de måder hvorpå spillet holder en acceptabel framerate er ved at have en lav opløsning. Jo færre felter på hver led, desto mindre simuleringssteppet tager at beregne, som vi også så i forsøget. Dette ses dog næsten ikke nogen andre steder. Der er ikke rigtig nogen spil der gør brug af fluidsims, i hvert fald ikke nogen der er fysisk baserede. Men hvorfor dog ikke? Der er jo så mange gode ting ved fluidsims! Den eneste logiske konklusion er at computer hardware ikke er kraftfuldt nok. Powder Game havde simuleringen som hovedpunkt, og med en meget lav opløsning. For at kunne have fluidsims med i sine spil som del af spillet uden at dominere det, skal der bare rigtig computer power til. Idet computere bliver bedre og bedre kan der bruges flere og flere kræfter på alle mulige aspekter af spil. Fluidsims er bare ikke det værd lige nu. Det tager for meget computerkraft at køre, i forhold til hvad man får retur, i form af fordybelse i en spilverden eller seje effekter. Dette kommer dog ikke til at være sandt for evigt. Idet computere fortsætter med at blive bedre vil der komme et tidspunkt hvor det godt kan betale sig at bruge computerkraft på fluidsims i sit spil. Idet andre former for forbedring og realisme stille og roligt bliver implementeret i spil, kommer fluidsims til at være det næste naturlige skridt. Ikke nu, men i fremtiden.

Konklusionen

Væskedynamik findes overalt omkring os og har en effekt på mange dele af vores liv. At simulere væskedynamik på computere er derfor en vigtig del af computervidenskab. Forskellige metoder og algoritmer bruges til at gøre simuleringerne simplere og nemmere at køre, mange baseret på Navier-Stokes ligningerne. Da tiden det tager at køre simuleringerne er et vigtigt aspekt, burde det kigges på. Programmer kan skrives der simulerer og sammenligner algoritmer for at bestemme egenskaber som tidskompleksitet for forskellige algoritmer og hvorledes de afspejler den virkelighed de ultimativt er baseret på. Den feltbaserede algoritme afspejler virkeligheden, og reagerer realistisk på ændringer i karakteristika af væsken der burde have indflydelse på flowet. De forskellige algoritmer har styrker og svagheder, og kan benyttes i forskellige situationer ud fra deres præcision og tidskompleksitet. Simuleringer af væskedynamik har stort potentiale inde for mange industrier, blandt anden computerspilsindustrien. Med bedre computere, vil fluidsims finde sted iblandt os meget mere end vi tror. Fluidsims er fantastiske.

Referencer

[1] Stam, J. (2003). "Real-Time Fluid Dynamics for Games". Hentet fra dgp.toronto.edu: https://www.dgp.toronto.edu/public_user/stam/reality/Research/pub.html

[2] Parth G. "*This Downward Pointing Triangle Means Grad Div and Curl in Vector Calculus (Nabla / Del) by Parth G*", youtube.com (23-03-2021). https://youtu.be/hI4yTE8WT88

[3] Jani, Deepak. "Reynolds-nummer". lambdageeks.com (Sidst besøgt: 18-03-2022). https://da.lambdageeks.com/reynolds-number/

[4] Stam, J. (1999). "Stable Fluids". Hentet fra dgp.toronto.edu: https://www.dgp.toronto.edu/public_user/stam/reality/Research/pub.htm

[5] Månsson, Daniel. (2013). "Interactive 2D Particle-based Fluid Simulation for Mobile Devices", Bachelor's Thesis at KTH. https://www.diva-portal.org/smash/get/diva2:676516/FULLTEXT01.pdf

[6] The Coding Train. "Coding Challenge #132: Fluid Simulation", youtube.com (12-02-2019). https://youtu.be/alhpH6ECFvQ

[7] Reducible, "What Is Big O Notation?", youtube.com. (29-03-2020). https://www.youtube.com/watch?v=Q_1M2JaijjQ&t=850s

[8] "Big O Notation", Wikipedia.com. (Sidst besøgt: 18-03-2022). https://en.wikipedia.org/wiki/Big_O_notation

[9] Google drev mappe med GIFs af simuleringerne. Klik på dem for at se dem i browseren. (Det kan tage et lille stykke tid for dem at loade) https://drive.google.com/drive/folders/1FOMilJMwKn2Sil_gUMKcww_ji2u0wyxt?usp=sharing

[10] ha55ii: "Powder Game" (2007), dan-ball.jp (Besøgt: 18-03-2022) https://dan-ball.jp/en/javagame/dust/

Kildekoden til programmerne kan findes i dette offentlige repository på GitHub. Under mapperne "Fluid_simulator" og "Fluid_sim_2".

https://github.com/Skysub/SOP-simulator

Bilag

Bilag 1:

Tidskompleksitet af den feltbaserede metode, fortsat.

```
mF.diffuse(1, Vx0, Vx, visc, dt);
mF.diffuse(2, Vy0, Vy, visc, dt);

mF.project(Vx0, Vy0, Vx, Vy);

mF.advect(1, Vx, Vx0, Vx0, Vy0, dt);
mF.advect(2, Vy, Vy0, Vx0, Vy0, dt);

mF.project(Vx, Vy, Vx0, Vy0);

mF.diffuse(0, s, density, diff, dt);
mF.advect(0, density, s, Vx, Vy, dt);
```

Vi blev færdige med at kigge på den første diffuse og kom frem til at funktionen for hvordan tiden stiger i forhold til *n* var $f(n) = n^2$. Der er dog to diffuse, så egentlig burde det være $f(n) = 2n^2$. Da faktoren ikke er vigtig og skal fjernes alligevel kan den dog ignoreres. Vi fortsætter ned i project().

```
void project(float[] velocX, float[] velocY, float[] p, float[] div) {
  for (int j = 1; j < N - 1; j++) {
    for (int i = 1; i < N - 1; i++) {
      div[IX(i, j)] = -0.5f*(
        velocX[IX(i+1, j)]
        -velocX[IX(i-1, j)]
        +velocY[IX(i, j+1)]
        -velocY[IX(i, j-1)]
        )/N;
      p[IX(i, j)] = 0;
    }
  }

  set_bnd(0, div);
  set_bnd(0, p);
  lin_solve(0, p, div, 1, 4);

  for (int j = 1; j < N - 1; j++) {
    for (int i = 1; i < N - 1; i++) {
      velocX[IX(i, j)] -= 0.5f * (  p[IX(i+1, j)]
        -p[IX(i-1, j)]) * N;
      velocY[IX(i, j)] -= 0.5f * (  p[IX(i, j+1)]
        -p[IX(i, j-1)]) * N;
    }
  }
  set_bnd(1, velocX);
  set_bnd(2, velocY);
}
```

Vi ser to for-løkker inden i hindanden baseret på N. Dette ville give n^2, og da der ikke sker yderligere ting indeni baseret på N, ændres f(n) ikke. Vi har allerede set på metoderne. Bagefter kaldes set_bnd(), og lin_solve(). Set_bnd() har en linær tidskompleksitet som vi så på tidligere, og lin_solve() har en n^2 kompleksitet, så der er ikke noget der ændres her. Resten af metoden er mere af det samme, enten lineært eller n^2. Tiden det tager at beregne hele algoritmen bliver markant større, men den vokser ikke hurtigere på grund af project() metoden. Til f(n) ville der tilføjes $3n^2 + 4n$ baseret på metoderne og løkkerne her. Vi bevæger os tilbage tilbage op af stacken til step() igen.

Nu dykker vi ned i advect().

```
void advect(int b, float[] d, float[] d0, float[] velocX, float[] velocY, float dt) {
  float i0, i1, j0, j1;

  float dtx = dt * (N - 2);
  float dty = dt * (N - 2);

  float s0, s1, t0, t1;
  float tmp1, tmp2, x, y;

  float Nfloat = N;
  float ifloat, jfloat;
  int i, j;

  for (j = 1, jfloat = 1; j < N - 1; j++, jfloat++) {
    for (i = 1, ifloat = 1; i < N - 1; i++, ifloat++) {
      tmp1 = dtx * velocX[IX(i, j)];
      tmp2 = dty * velocY[IX(i, j)];
      x    = ifloat - tmp1;
      y    = jfloat - tmp2;

      if (x < 0.5f) x = 0.5f;
      if (x > Nfloat + 0.5f) x = Nfloat + 0.5f;
      i0 = floor(x);
      i1 = i0 + 1.0f;
      if (y < 0.5f) y = 0.5f;
      if (y > Nfloat + 0.5f) y = Nfloat + 0.5f;
      j0 = floor(y);
      j1 = j0 + 1.0f;

      s1 = x - i0;
      s0 = 1.0f - s1;
      t1 = y - j0;
      t0 = 1.0f - t1;

      int i0i = int(i0);
      int i1i = int(i1);
      int j0i = int(j0);
      int j1i = int(j1);

      // DOUBLE CHECK THIS!!!
      d[IX(i, j)] =
        s0 * (t0 * d0[IX(i0i, j0i)] + t1 * d0[IX(i0i, j1i)]) +
        s1 * (t0 * d0[IX(i1i, j0i)] + t1 * d0[IX(i1i, j1i)]);
    }
  }

  set_bnd(b, d);
}
```

Igen, det værste vi ser er to for-løkker inden i hindanden baseret på N. Tidskompleksiteten af hele algoritmen ændres ikke af denne metode. Vi går tilbage til step().

```
void step() {
  int N          = this.size;
  float visc     = this.visc;
  float diff     = this.diff;
  float dt       = this.dt;
  float[] Vx      = this.Vx;
  float[] Vy      = this.Vy;
  float[] Vx0     = this.Vx0;
  float[] Vy0     = this.Vy0;
  float[] s       = this.s;
  float[] density = this.density;

  mF.diffuse(1, Vx0, Vx, visc, dt);
  mF.diffuse(2, Vy0, Vy, visc, dt);

  mF.project(Vx0, Vy0, Vx, Vy);

  mF.advect(1, Vx, Vx0, Vx0, Vy0, dt);
  mF.advect(2, Vy, Vy0, Vx0, Vy0, dt);

  mF.project(Vx, Vy, Vx0, Vy0);

  mF.diffuse(0, s, density, diff, dt);
  mF.advect(0, density, s, Vx, Vy, dt);
}
```

Det var det. Det var alle metoderne der kaldet på et enkelt simulerings step. Tidskompleksiteten ender med at blive $O(n^2)$.

Bilag 2:

Tidskompleksitet af den partiel baserede metode. Simuleringen af et enkelt simulation step foregår i metoden SimStep(), så det er der vi starter med at kigge. Da der ikke er felter på samme måde som i den feltabaserede metode, men partikler i stedet for, bliver n, antallet af partikler. Da partiklerne holdes i en liste, bliver størrelsen af denne liste brugt som begrænsning for hvor langt løkker baseret på partikkel antallet løber. Dvs. ses particles.size() som betingelse i en for-løkke, så er den baseret på antallet af partikler.

```
void SimStep() {
  ApplyExternalForces();
  ApplyViscosity();
  AdvanceParticles();
  UpdateNeighbors();
  DoubleDensityRelaxation();
  UpdateVelocity();
}
```

Vi kalder 6 forskellige metoder, for at bestemme tidskompleksiteten af hele algoritmen kan vi bestemme kompleksiteten af de 6 metoder hvert for sig, og hele algoritmens tidskompleksitet vil matche den største tidskompleksitet blandt metoderne. Vi starter med ApplyExternalForces().

```
//Step 1
void ApplyExternalForces() {
  for (int i = 0; i < particles.size(); i++) {
    if (grid.PosToIndex(particles.get(i).getPos())[0]>9 && grid.PosToIndex(particles.get(i).getPos())[1]>9
    && grid.PosToIndex(particles.get(i).getPos())[0]<12 && grid.PosToIndex(particles.get(i).getPos())[1]<12
    ) {
      particles.get(i).setVel(new PVector(10, 0).add(particles.get(i).getVel()));
      //println(particles.get(i).getVel());
      reds.append(i);
    }
  }
}
```

Der er en løkke der løber gennem alle partiklerne, det svarer til en kompleksitet på n. Inde i løkken kaldes der dog mange metoder. Vi er dog kun interesseret i metoder hvor del af dem er baseret på n.

```
int[] PosToIndex(PVector pos) {
  int[] out = new int[2];
  out[0] = floor(pos.x/(1024/gridSize));
  out[1] = floor(pos.y/(1024/gridSize));
  return out;
}
```

PosToIndex() skalerer ikke baseret på n. Resten af metoderne der ses i ApplyExternalForces() er enten indbygget i ArrayList(), der er standard i Java, eller er en simpel get/set metode jeg har kodet ind i partikel klassen. Get/set metoderne kører i konstant tid. Alle metoderne der bruges fra ArrayList er også konstante, i hvert fald dem der har med listen af partikler at gøre.

Vi går videre til næste metode i SimStep(); ApplyViscosity().

```
//Step 2
void ApplyViscosity() {
  for (int i = 0; i < particles.size(); i++) {
    for (int j = 0; j < neighbors.get(i).size(); j++) {
      Particle p = particles.get(i), n = neighbors.get(i).get(j);
      //PVector Vpn = n.getPos().sub(p.getPos());
      PVector Vpn = new PVector(n.getPos().x-p.getPos().x, n.getPos().y-p.getPos().y);
      float velInward = new PVector(p.getVel().x-n.getVel().x, p.getVel().y-n.getVel().y).dot(Vpn);
      if (velInward > 0) {
        float l = Vpn.mag();
        velInward = velInward/l;
        Vpn.div(l);
        float q = l/radius;
        PVector I = Vpn.mult(0.5*timeStep*(1-q)*(viscLD*velInward+viscQD*(velInward*velInward)));
        particles.get(i).setVel(p.getVel().sub(I));
      }
    }
  }
}
```

Her ses to for-løkker. Det er dog kun den ene der er direkte baseret på antallet af partikler. Alle metoderne der kaldes inderst i løkkerne, er enten vektorberegning, eller ArrayListens metoder, ingen af dem skalerer lineært med antallet af partikler. Den anden for-løkke er baseret på antallet af naboer til partiklet der ses på. Det er muligt for alle af partiklerne at være alle de andre partiklers naboer. I det tilfælde bliver kompleksiteten n^2. Det kan sagtens ske når programmet kører, og da tidskompleksitet tager det værste tilfælde betragtning, bliver ApplyViscosity()'s tidskompleksitet n^2. Vi tager et kig på den næste metode i SimStep().

```
//Step 3
void AdvanceParticles() {
  for (int i = 0; i < particles.size(); i++) {
    particles.get(i).setPosPrev(particles.get(i).getPos().copy());
    particles.get(i).setPos(new PVector(timeStep*particles.get(i).getVel().x,
     timeStep*particles.get(i).getVel().y).add(particles.get(i).getPos()));
    grid.MoveParticle(i, particles.get(i).getPos(), particles.get(i).getPosPrev());
  }
}
```

Der en enkelt for-løkke baseret på n. Den eneste metode her vi ikke har set på eller diskutere endnu er grid.MoveParticle(), så den tager vi et kig på.

```
void MoveParticle(int index, PVector pos, PVector posPrev) {
  int indexO = PosToIndex(posPrev)[0];
  int indexOO = PosToIndex(posPrev)[1];
  if (PosToIndex(pos)[0]>31 || PosToIndex(pos)[0]<0 || PosToIndex(pos)[1]> 31 || PosToIndex(pos)[1]<0) return;
  for (int i = 0; i < grid[indexO][indexOO].size(); i++) {
    if (grid[indexO][indexOO].get(i) == index) {
      grid[indexO][indexOO].remove(i);
    }
  }
  grid[PosToIndex(pos)[0]][PosToIndex(pos)[1]].append(index);
}
```

I denne metode er der en for-løkke. Den bruges til at tjekke en ArrayList igennem for at fjerne at fjerne et element. Det er muligt at der kan være et element for hvert partikel i den ArrayList. Hvilket giver at MoveParticle() har en lineær kompleksitet i værste tilfælde. Og da MoveParticle() metoden kaldes for hvert partikel, da det er inde i for-løkken i AdvanceParticles(), så må AdvanceParticles have en tidskompleksitet på n^2.

Vi tager og kigger på den næste metode i SimStep(); UpdateNeighbors().

```
//Step 4
void UpdateNeighbors() {
  for (int i = 0; i < particles.size(); i++) {
    neighbors.get(i).clear();
    for (int j = 0; j < grid.getPossibleNeighbors(particles.get(i).getPos(), i).size(); j++) {
      if (new PVector(particles.get(i).getPos().x, particles.get(i).getPos().y).sub(
       grid.getPossibleNeighbors(particles.get(i).getPos(), i).get(j).getPos()).mag() < radius
       ) {
        neighbors.get(i).add(grid.getPossibleNeighbors(particles.get(i).getPos(), i).get(j));
      }
    }
  }
}
```

Vi har en for-løkke der går gennem alle partikler. Under det har vi en for-løkke der går gennem alle mulige naboer. Da det er muligt for alle partikler at være en mulig nabo, så må kompleksiteten indtil videre være n^2. Der kaldes dog en ny metode inderst inde i løkkerne. grid.getPossibleNeighbors(). Vi undersøger dens tidskompleksitet.

```
ArrayList<Particle> getPossibleNeighbors(PVector pos, int index) {
  ArrayList<Particle> possibleNeighbors = new ArrayList<Particle>();
  int[] indexes = PosToIndex(pos);
  for (int i = -1; i < 2; i++) {
    for (int j = -1; j < 2; j++) {
      if (indexes[0]+i > -1 && indexes[0]+i < 32 && indexes[1]+j > -1 && indexes[1]+j < 32) {
        for (int s = 0; s < grid[indexes[0]+i][indexes[1]+j].size(); s++) {
          if (grid[indexes[0]+i][indexes[1]+j].get(s) != index) {
            possibleNeighbors.add(usePart.Copy(master.particles.get(grid[indexes[0]+i][indexes[1]+j].get(s))));
          }
        }
      }
    }
  }
  return possibleNeighbors;
}
```

Vi har 3 løkker inden i hinanden. De to yderste løbes kun igennem 3 gange hvert. Den inderste er baseret på en liste over partikler i en bestemt gitter celle. Det er muligt for alle partiklerne at være inde i den bestemte gittercelle. Det giver at tidskompleksiteten af denne algoritme bliver n. Og da getPossibleNeighbors() er inde i to for-løkker der hver er baseret på n, så bliver tidskompleksiteten af UpdateNeighbors() n^3, da tidskompleksitet ser på det værste mulige tilfælde.

Vi fortsætter og ser på den næste metode i SimStep().

```
//Step 5
void DoubleDensityRelaxation() {
  for (int i = 0; i < particles.size(); i++) {
    Particle p = particles.get(i);
    float density = 0;
    float densityNear = 0;
    for (int j = 0; j < neighbors.get(i).size(); j++) {
      Particle n = neighbors.get(i).get(j);
      float tempN = new PVector(p.getPos().x, p.getPos().y).sub(new PVector(n.getPos().x, n.getPos().y)).mag();
      float q = 1f-(tempN/radius);
      density = density + (q*q);
      densityNear = densityNear + (q*q*q);
    }
    float P = k * (density - densityBase);
    float Pnear = kN * densityNear;
    PVector delta = new PVector(0, 0);
    for (int j = 0; j < neighbors.get(i).size(); j++) {
      Particle n = neighbors.get(i).get(j);
      float tempN = new PVector(p.getPos().x, p.getPos().y).sub(new PVector(n.getPos().x, n.getPos().y)).mag();
      if (tempN == 0) tempN = 0.001f;
      float q = 1f-(tempN/radius);
      PVector Vpn = new PVector(p.getPos().x, p.getPos().y).sub(new PVector(n.getPos().x, n.getPos().y)).div(tempN);
      PVector D = Vpn.mult(0.5*(timeStep*timeStep)*(P*q+Pnear*(q*q)));
      neighbors.get(i).get(j).setPos(neighbors.get(i).get(j).getPos().add(D));
      delta.sub(D);
    }
    particles.get(i).setPos(p.getPos().add(delta));
  }
}
```

Der er tre løkker, den ene baseret på antallet af partikler, og de andre baseret på antallet af naboer. Der er ellers ikke nogen nye metoder vi burde dykke ind i. Det giver at denne metode har en kompleksitet på n^2. Nu ser vi på den sidste metode:

```
//Step 7
void UpdateVelocity() {
  for (int i = 0; i < particles.size(); i++) {
    particles.get(i).setVel(particles.get(i).getPos().copy().sub(particles.get(i).getPosPrev()).div(timeStep));
  }
}
```

Denne metode er meget simpel. Der er en enkelt løkke der løber gennem alle partiklerne, og opdaterer deres hastighed. Denne metode er lineær.

Da en af metoderne i SimStep() havde en tidskompleksitet på $O(n^3)$ bliver tidskompleksiteten for hele algoritmen $O(n^3)$.