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 $O(n^3)$.