# RAPPORT OPPGAVE 2

---

## INF-1400-Objektorientert programmering

**28. februar 2019**

Martin Soria Røvang
Universitetet i Tromsø

Inneholder 10 sider, inkludert forside.

# Innhold

# 1 Introduksjon

I denne oppgaven ble det laget en boid simulator, en simulator som simulerer en flokk. Det beskrives hvordan dette kan bli implementert og hvordan det gjøres med hensyn på objekt-orientert programmering.

## 1.1 Krav

# 2 Teknisk bakgrunn

*Arv(Inheritance)*: Ved å arve fra en annen klasse henter man ut alle egenskapene denne klassen har. Dette kan brukes til å utbedre eller legge til flere ting som er spesialisert, men også bruker andre generelle egenskaper. Dette gjør koden bedre å lese og enklere å bygge på, navigere og feilsøke.[Dusty [Oktober 2018] p.71
*super()*: Super metoden gir deg mulighet til å hente ut metoder eller attributter fra foreldreklassen(parentclass). Hvis du for eksempel skal legge til flere atributter ved instansering av et objekt, men vil også beholde de fra foreldreklassen må man kjøre *super().__init__()* for å hente ut disse [Dusty [Oktober 2018] p.74].

# 3 Design

## 3.1 Klassestruktur

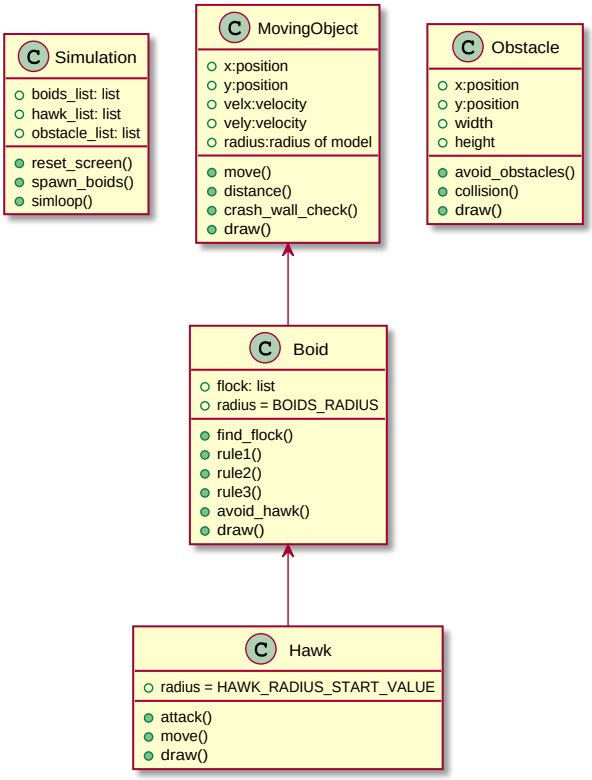
I dette prosjektet har vi klassestrukturen som vist i figur(1) under,

Her arver *Boid* klassen fra *MovingObject* og *Hawk*(Hoik) fra Boid. Hvis man arver fra en generell struktur kan man spare kodeplass og generelt får mer strukturert kode. Ved får man også alle metodene og attributtene fra klassen man arver fra[Dusty [Oktober 2018] p.71]. I noen av klassene ble noen av metodene og attributter erstattet. Dette ble gjort ved å lage en metode som heter det samme i barneklassen(den klassen som arver). Dette gjelder også *__init__()* metoden, derfor vil det oppstå problemer hvis man vil legge inn nye initialattributter, da dette vil erstatte arv-attributtene. et eksempel vil være å gjøre slik som vist i figur(2) ved bruk av super()[Dusty [Oktober 2018] p.74].

## 3.2 Simulation

Simulation er klassen som styrer hva som skal kjøres i programmet. Den har følgene metoder,

- *reset_screen*: Fyller skjermen med svart for å overskrive tidligere posisjoner.
- *spawn_boids*: Lager boids hvis man trykker på venstre museknapp.
- *simloop*: Dette er den metoden som kjører kontinuerlig i en while-loop så lenge simulasjonsprogrammet er på.




**Figur 1:** Klassestrukturen i programmet.

```
        class eksempel:
            def __init__(self):
                super().__init__()
                self.a = 10

```

**Figur 2:** Lage nye initialattributter og hente de andre fra foreldreklassen

## 3.3 MovingObject

MovingObject er en generell funksjon som lager instanser som skal kunne bevege seg, finne distanser til andre bevegelige objekter, unngå å kræsje i vegger og tegne dem. Dette er en klasse man vil arve fra når man skal ha mer spesifikke klasser som bruker alle/mange av metodene og attributtene til MovingObject. MovingObject består av funksjonene,

- *move*: Beveger objektet etter hvor stor hastigheten er.
- *distance*: Returnerer distansen til et annet objekt.
- *crash_wall_check*: Sjekker om objektet er på vei utafor skjermen. Hvis objektet er på vei ut vil hastigheten bli reflektert og demped. Dette skjer ved å gange hastigheten $\boldsymbol{v}$ med en refleksjonskonstant gitt i konfigurasjonefilen.
- *draw*: Tegner objektet ut på skjermen.

*Move* metoden summer på den tidligere posisjonen og på en enhetsvektor som er skalert av verdi gitt fra konfigurasjonsfilen.

$$\boldsymbol{V} = \frac{\boldsymbol{v}}{||\boldsymbol{v}||}\alpha \tag{1}$$

,der $\alpha$ er en skalar for hva farten skal være slik at,

$$||\boldsymbol{V}|| = \alpha, \forall\, t \tag{2}$$

Metoden *distance* finner distansen til et annet objekt ved ta lengden gitt fra pytagoras,

$$Distance = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

der subindeks {1} angir posisjonen til objektet man vil finne lengden fra og {2} er posisjonen til objektet man vil finne lengden til.

## 3.4 Boid

Denne klassen inneholder metodene som får objektene til å oppføre seg som en naturlig flokk. Dette blir gjort ved å lage 3 forskjellige regler som kjører kontinuerlig i while-loopen i simulasjons-løkka. Denne klassen arver også fra *MovingObject* derfor vil denne også inneholde alt den også har. Vi har metodene,

- *find_flock*: Putter alle boidsene som tilhører en boid sin flokk i en liste som attributt til en boid. Alle boids vil derfor få en liste med boids(denne tar ikke med boidsen selv).
- *rule1*: Finner gjennomsnitts-posisjonen og lager en vektor dit, denne blir summet på hastighets-vektoren til boidsen.
- *rule2*: Boidsene unngår å kræsje i hverandre ved å summe på en vektor som er i motsatt retning av boidsen som den er på vei å kræsje med.
- *rule3*: Boidsene prøver å matche hastigheten til boidsene som er i sin umiddelbare nærhet.[1]
- *avoid_hawk*: Summer på vektor som er i motsatt retning av haukene/hoikene slik at de prøver å unngå dem.
- *draw*: Tegner dem ut på skjermen.

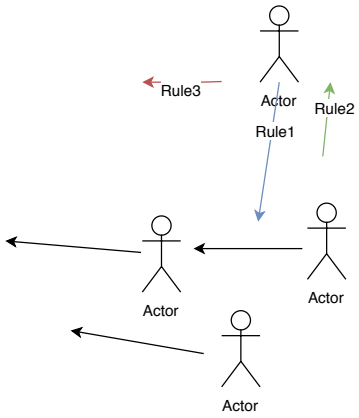
I figur(3) ser man hvordan de tre vektorene kan se ut for en boid.
Når disse blir summet opp vil man få en totalvektor som vist i figur(4) under,
Dette gir en fin og glatt bevegelse til flokkene som ser noe naturlig ut.

## 3.5 Hawk/Hoik

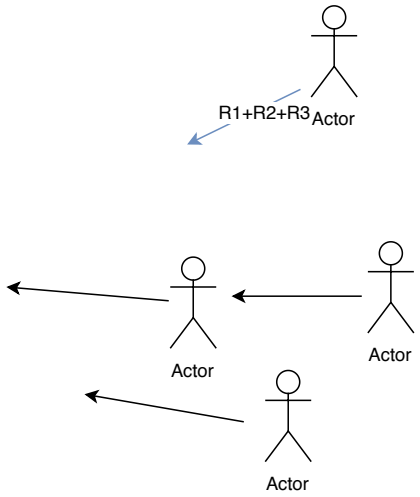
Denne klassen arver fra boids så disse vil oppføre seg likt som dem. Metodene er,

- *attack*: Summer opp hastighets-vektoren til hauken som peker i retningen av den nærmeste boiden.

[1] Disse reglene ble laget ved bruk av informasjonen på `http://www.kfish.org/boids/pseudocode.html`




**Figur 3:** De forskjellige vektorene fra regel 1,2 og 3




**Figur 4:** Summen av vektorene

- *move*: Lagt til en egen move funksjon fordi hauken har en egen global variabel i konfigurasjonefilen som angir farten uavhengig av boid farten.
- *draw*: Tegner hauken/hoiken ut på skjermen.

Denne klassen var ganske grei å implementere da det meste var gjort ved arv fra boid klassen. Noe å ta med seg er at hauken/hoiken bør ha større hastighet enn boidsene, derfor måtte man implementere en ny metode som tok inn egen skalar for hauken.

## 3.6 Obstacles

Dette er en klasse som skal oppføre seg som en vegg for de bevegelige objektene. Metodene er;

- *collision*: Sjekker om et bevegelig objekt treffer veggene. returnerer *True* for å bruke det i *avoid_obstacles* metoden.

- *avoid_obstacles*: Summer opp en hastighetsvektor til boiden/hauken som peker i motsatt retning av veggen slik at de flyr unna. Det er også en test som setter hastigheten til den motsatte retningen om du skulle kræsje med veggen. Distansen fra veggen før boids/hoiks begynner å styre unna er satt i config filen under det globale variabelnavnet WALL_DISTANCE_AVOID_VALUE.

# 4 Implementasjon

Koden er skrevet i Python versjon 3.7[2]
OS: Windows 10
Systemtype: 64-bit OS, x64-basert prosessor
Skjermkort: NVIDIA Geforce 920MX
CPU: Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz
RAM: 4GB
Pygame [3] Version: 1.9.4
Nympy [4] Version: 1.14.5

# 5 Diskusjon

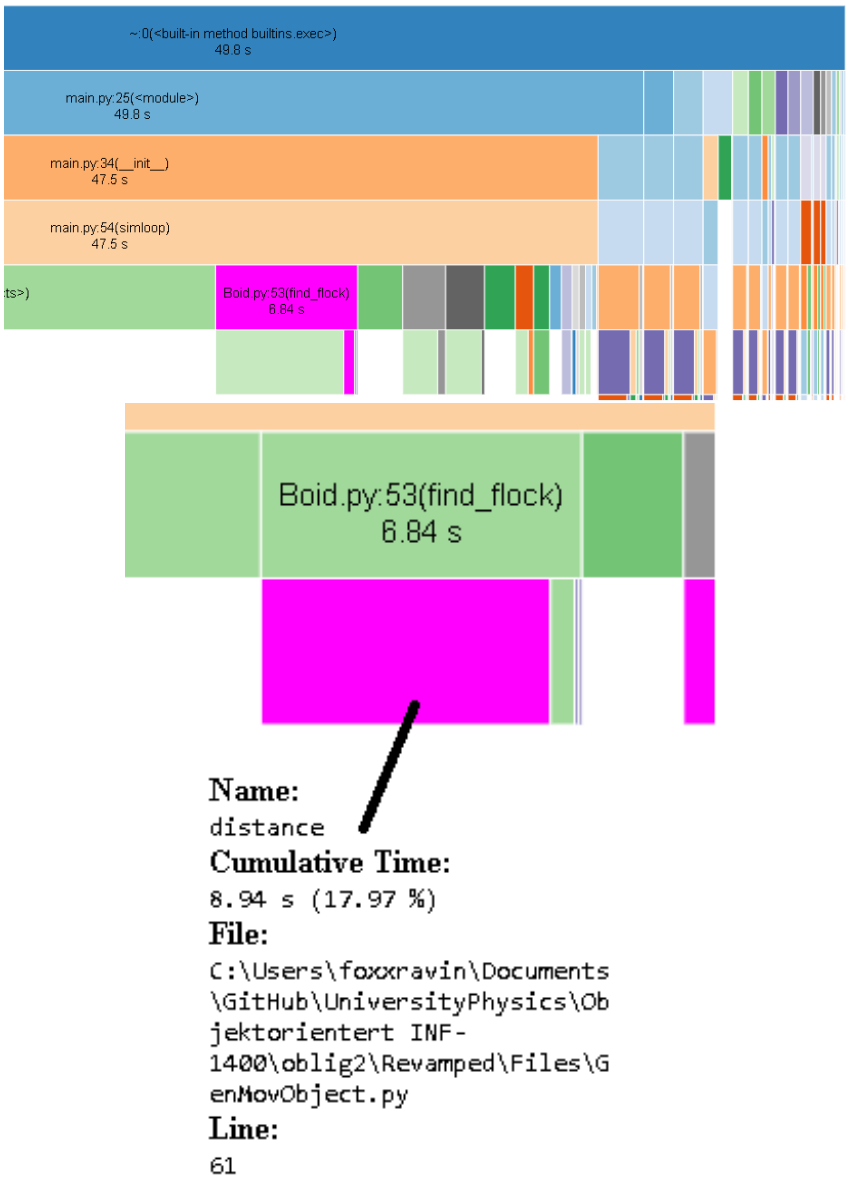
Via sumulasjons-klassen er det lagt inn mulighet for å lage boids ved å trykke på venstre *musetast*, for hoikene trykker man knappen *C* på tastaturet og for å lage en obstacle trykker man på *A*. Dette ble gjort ved bruk av events i pygame.[5] I denne oppgaven ble pseudokoden fra `http://www.kfish.org/boids/pseudocode.html`, dette gjorde det ganske enkelt å implementere reglene for simulatoren. Det vanskligste var å få hoikene/boidsene til å unngå obstacles så fint som mulig og rett og slett angi de riktige størrelsene på skalarene slik at bevegelsene så naturlige ut. Implementasjonen til kantene til simulasjonsvinduet gjorde at det så litt ping-pong ut da boids/hoiks traff dem. Her kunne man ha fjernet veggene ved å for eksempel flytte dem over til den andre siden da de var på vei ut av den ene siden. Dette ville ha skapet litt bedre flyt. Et problem med dette er at de første boidsene/hoiksene som havnet på den andre siden vil miste sine flokk egenskaper da ikke de andre flokk medlemmene har blitt flyttet enda. Dette kan også føre til at hauken/hoiken plutselig vil snu og bevege seg mot den andre siden istedet for å bli med ut av skjermen og også bli flyttet, noe som kan virke unaturlig.

## 5.1 Evaluasjon

Ved bruk av snakeviz[6] kan man visualisere tidsbruk av de forkjellige klassene og metodene. I figur(5) ser man en visualisering av dette programmet.

Distanse metoden (untatt simloop osv.) er den som bruker mest tid i programmet. Dette er ikke så rart da denne metoden blir kjørt av mange metoder. (rule3, rule2, avoid_hawk, attack etc). Tar man

[2] https://www.python.org/
[3] https://www.pygame.org/wiki/GettingStarted
[4] http://www.numpy.org/
[5] https://www.pygame.org/docs/ref/event.html
[6] https://jiffyclub.github.io/snakeviz




**Figur 5:** Visualisering av programmet, her ser man at find_flock metoden er den som bruker mest tid av de selvskapte metodene, der mesteparten av tidsbruket kommer fra distance metoden.

en titt på find_flock metoden kan man se en grunn til at dette er den metoden som bruker mest tid. I denne metoden går man igjennom alle boidsene(som kan bli veldig mange) også sjekker man distansen fra en boid til en annen (dette skjer for hver eneste boid!!). deretter blir de lagt inn en liste-attributt som hver boid har.

# 6 Konklusjon

I denne oppgaven så vi på hvordan man kan implementere en boid simulator i python med pygame modulen. Her så man hvordan tre enkle regler gjorde at bevegelige objekter fikk en naturlig flokkbevegelse

som man kan observere med blant annet fugler og fisker. Det ble også introdusert rovdyr(hoik) som prøver å spise boidsene for å simulere flokk-rovdyr effekt. Reglene ble implementert ut ifra psuedokoden gitt fra `http://www.kfish.org/boids/pseudocode.html`. Det krevde at man finjusterte parametere på hastigheter osv for å få det til å fungere på best mulig måte.

# 7 Appendix

# 8 Referanser

Phillips Dusty. *Python 3 Object-Oriented Programming*. Packt Publishing Ltd., third edition, Oktober 2018.