Assignment 2 rapport - INF-1400

Robin Kristiansen

19. april 2021

1 Introduksjon

Oppgaven ba oss replikere en flokkalgoritme kalt «Boids» i Python ved hjelp av pygame biblioteket. Vi skulle bruke objekt orienterte programmeringsprinsipper slik som klasser, metoder og arv i prosessen.

2 Teknisk bakgrunn

Boids originalt skapt i 1986 av Craig Reynolds er en flokksimulasjon som følger tre enkle regler: 1) unngåelse (avoidance/separation) en boid skal holde avstand til andre boids; 2) innretting (alignment) boids skal rette seg etter den generelle retningen til en nabo-flokk; og til slutt 3) samhold (cohesion), boids skal rette seg mot midten av flokken. Med disse tre reglene skal man kunne skape en nærmest livaktig simulasjon til hvordan f.eks. en gruppe fugler flyr i flokk.

Boids har en motpart, et rovdyr (eng. predator/hunter) kan man si, som kalles «hoiks». Dette er en mindre gruppe eller et enkel boid-liknende, flyvende rovdyr som har som mål å drepe/spise boids. Den flyr rundt med nesten de samme reglene som gjelder for boids. Den skal rette seg mot en flokk med boids og prøve å spise/drepe en av dem. De bryr seg ikke om å unngå boids, så separation regelen utgår, og det samme gjelder for alignment, de trenger ikke rette seg i samme retning som boids, bare de klarer å infiltrere en flokk har de bestått oppdraget.

Oppgaven ber oss bruke objekt orienterte programmeringsprinsipper slik som klasser, arv, metoder og så videre. Jeg bruker aktivt ovennevnte prinsipper, mm. i koden. Et eksempel kan være hvordan `Boid`-klassen er satt opp, hvor `Boid` arver fra `pygame.sprite.Sprite`, klassen har også definert et antall metoder som brukes aktivt på hver instans av klassen.

Arv innebærer at man tar alle metoder og klasse-variabler fra super-klassen (klassen man arver fra) og bruker dem selv, det vil si du arver den samme funksjonaliteten fra super-klassen og bruker dem selv. I tillegg til å arve kan man utvide klassen med egne variabler og metoder.

Metoder er funksjoner som kan utføres «på» et objekt av den spesifiserte klassen. Et eksempel kan være en hundeklasse som har en metode `bjeff()`, da kan brukeren skrive syntaksen `hund.bjeff()` for å få hundeobjektet til å «bjeffe». Metoder er altså funksjoner som er ment for den spesifikke klassen de er medlem av.

Jeg må jo også nevne *klasser*, som er noe av det mest essensielle i OOP. En klasse er en samling av metoder og variabler, for vanlige klasser (som ikke er abstrakte mv.) kan en instansiere et objekt av den klassen. Det vil si man lager et objekt som er av den klasse-typen. Med et objekt kan man utføre alle de metodene som er definert for den klassen, aksessere det objektets variabler mm. En *abstrakt klasse* er en type klasse – som ikke per definisjon egentlig *fins* i python – som ikke kan instansieres, bruksområdet til en slik klasse er å definere et grensesnitt (eng. interface) som brukeren må implementere selv, men som andre brukere eller brukeren selv, kan bruke seinere. Selve grensesnittet er det samme, men implementasjonen kan variere.

3 Design

Programmet starter med `Game`-klassen hvor alle ting rundt boids blir håndtert. Game-klassen setter opp vinduet hvor boids skal tegnes, den håndterer tastetrykk og avslutting av applikasjonen.

Det er fra `Boid`-klassen alt som direkte har noe med boids å gjøre skjer. Det er her vi bestemmer hvilken retningsvektor boid-en skal ha i neste oppdatering og hvordan boiden skal tegnes. I tillegg er det i `Boid`-klassen vi har definert reglene for boids.

Reglene alignment, cohesion og separation fungerer slik:

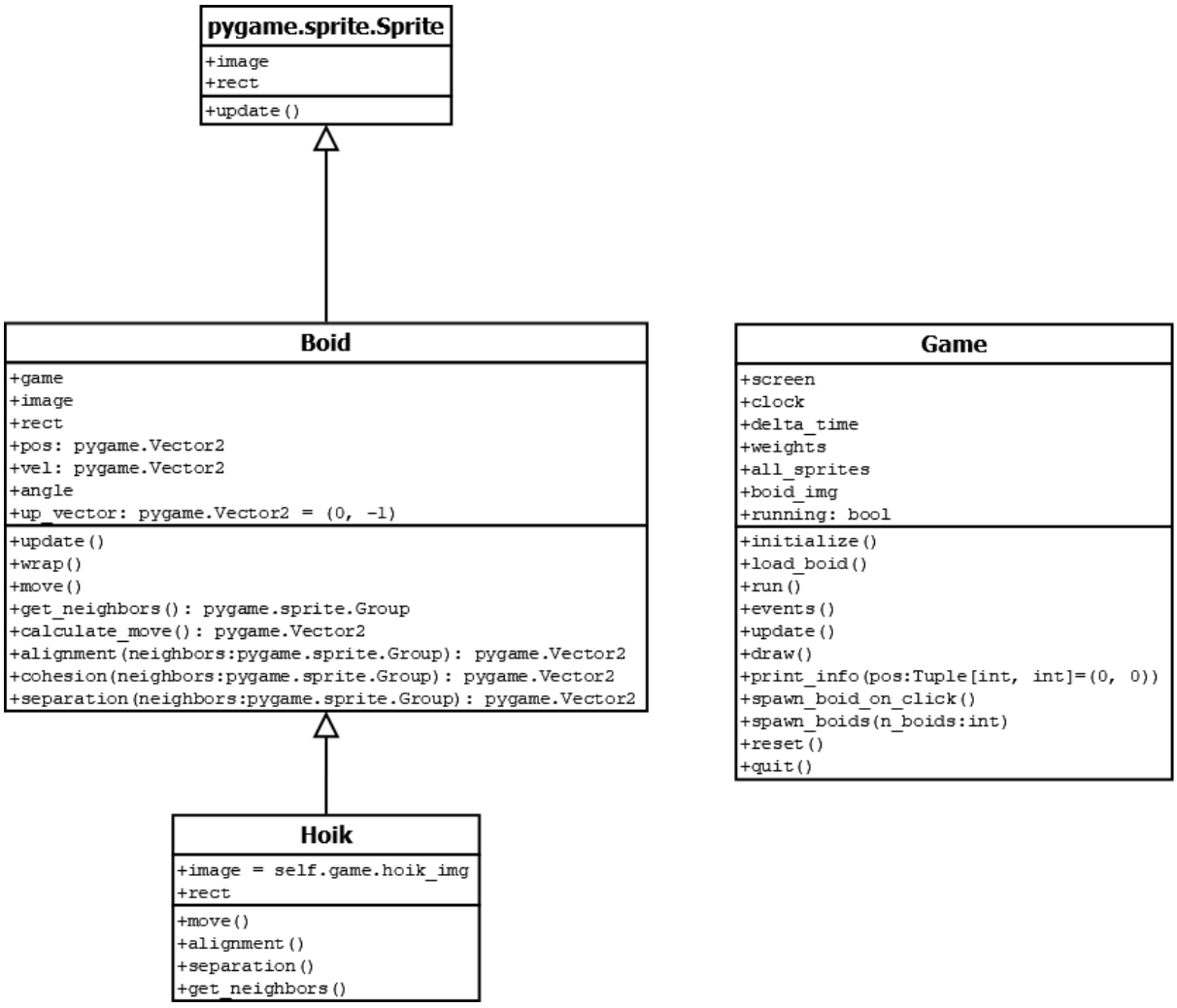
1) Alignment, finn gjennomsnittet av *retningen* (`neighbor.vel`) nabo-boids beveger seg i, og returner denne vektoren.
2) Cohesion, finn gjennomsnittet av *posisjonen* til nabo-boids og lagre den i en variabel. Trekk fra egen posisjon (`self.pos`) fra denne variabelen, til slutt returner variabelen.
3) Separation, begynn med å sjekke om nabo-boids er innenfor en mindre radius (`AVOIDANCE_RADIUS`), hvis sann øk en variabel med differansen av egen posisjon (`self.pos`) og nabo-boids posisjon (`neighbor.pos`). Etter det sjekkes om antall boids å unngå er større enn 0, hvis sann deles en variabel på antall boids å unngå og variabelen returneres.

Fellesnevneren til alle reglene er at de på en eller annen måte tar gjennomsnittet til et attributt ved nabo-boid-ene.

`Hoik` er konstruert slik at de arver fra `Boid`. Dette er en beslutning jeg tok som jeg tenkte ville være smart da de begge har likt utseende, men bare noen forskjellige «oppførsler». I ettertid tenker jeg at jeg kunne latt begge arve fra en felles klasse f.eks. `MovingObject` eller lignende. Hoiks følger nesten de samme reglene som boids, de følger ikke `alignment()` da jeg ikke ønsker at de nødvendigvis skal rette seg i samme retning som boids. De modifiserer `separataion()` ved at de bare returnerer en nullvektor (`Vector2(0,0)`) som betyr «ingen forandring».

Cohesion er den eneste reglen som ikke er modifisert, hoiks retter seg etter sentrum av massen (eng. center of mass), da jeg ønsker at de følger en gruppe boids. Ved en kollisjon mellom en hoik og en boid blir det kalt `boid.kill()` på den boiden som blir kollidert med. Dette fjerner den fra alle sprite grupper og slutter å tegne den på skjermen.

For at boids og hoiks skal holde seg innenfor vinduets rammer, og ikke bare fly avgårdet, har jeg implementert en metode i Boid kalt `wrap()`, denne utfører rett og slett en modulo-operarasjon på x- og y-posisjonsverdien med skjermens henholdsvis bredde og høyde til hver boid/hoik. Dette er en «billig» måte å «wrappe» på, men jeg fant ingen annen enkel måte å gjøre det på. Med denne implementasjonen av en wrap-metode kan bevegelsen se litt snodig ut, men det fungerer.



Figur 1 Klassediagram over Boid, Hoik og Game

4 Implementasjon

Måten boids er implementert på er ved at jeg har «tegnet» en polygon i `config.py` fila som bestemmer punkter hvor linjer skal tegnes fra og til. I Game-klassen lastes disse punktene inn og tegnes på en `pygame.Surface` som seinere blir brukt til å tegne boids.

I `main.py` har jeg en klasse som heter `Game` som har en metode som heter `initialize()`, denne kalles på av brukeren før en vil starte simulasjonen. Denne metoden kaller igjen på `load_boid()` og `load_hoik()`, det er disse to metodene som laster inn figuren jeg definerte i `config.py` og tegner den på hver sin respektive måte avhengig av om det er en boid eller hoik som skal tegnes. Produktet (ikke retur-verdi) av disse innlastingsfunksjonene lagres i globale variabler henholdsvis `self.boid_img` og `self.hoik_img`, som også er skalert ned med 40 % fordi jeg syns det passet bedre.

`Game` holder også på variablene `self.delta_time` og `self.weights[]` som er vital til funksjonen til simulasjonen. `delta_time` holder tiden i millisekunder siden siste bilde ble tegnet på skjermen, denne brukes i `Boid`-klassen for å gjøre bevegelse mer flytende. `weights[]` er et array med 3 flytpunkttall (eng. float), denne holder verdiene for henholdsvis alignment-, cohesion- og separation-vektene.

Jeg har valgt å dele opp funksjonaliteten i `Game`-klassen ved å lage en metode for hver ting som skal gjøres. Jeg har en metode for `run()`, en for `events()`, en for `draw()`, m.fl. Dette for å gjøre det mer tydelig hvilken metode som gjør hva, og at en metode *kun* gjør den ene tingen. run() kalles når brukeren vil starte simulasjonen, og `run()` har en uendelig løkke inni seg som tre metoder, `events()`, `update()` og `draw()`. Sjekk for hendelser, for eksempel: har brukeren trykket en knapp; oppdater sprites og følgende deres respektive posisjoner osv.; og `draw()`, ganske selvforklarende, tegne alle synlige objekter på skjermen.

Jeg har laget en Boid-klasse som arver fra `pygame.sprite.Sprite`. Dette gjør av vi arver noen variabler og metoder som kan brukes. Noen av variablene er `image` og `rect`, og noen metoder er `update()`, `draw()` og `kill()`. Siden `Boid` er en `Sprite` kan vi legge den til i såkalte sprite grupper (`pygame.sprite.Group`), dette er en samling av sprites. Slike grupper gjør det enkelt å utføre samme operasjon f.eks. `update()` på alle objektene i gruppen.

Implementasjon av regler er også splittet opp i flere metoder i `Boid`-klassen. De tre metodene har samme navn som reglene og returnerer alle en vektor med endring av retningsvektor. Jeg har valgt å lage reglene slik at de uavhengig returnerer en *endring* i retning, istedenfor at de påvirker retningsvektoren direkte. Dette ser du tydelig i `calculate_move()`.

`alignment(self, neighbors)` tar som argument en sprite-gruppe med nabo-boids. Om denne gruppen er tom returneres nåværende retningsvektor (`self.vel`), dette vil få boids til å holde samme retning som den allerede har, dermed ikke gjør noen endring på styringen. Om den ikke er tom utføres der litt beregning: lager en lokal variabel `alignment_move` for å holde returverdien (`Vector2`); itererer gjennom alle boids fra `neighbors` argumentet, med en for-løkke; inkrementerer `alignment_move` med verdien av `neighbor.vel`; når løkken er ferdig deler jeg `alignment_move` med antall naboer, dette tar effektivt gjennomsnittet av naboers `vel`. Returnerer `alignment_move`.

`cohesion(self, neighbors)` sjekker først om vi har noen naboer, returnerer også `self.vel` om naboer er lik null. Hvis det derimot er naboer starter vi beregningen. Lager en variabel `cohesion_move`, itererer over alle naboene, men i denne regelen summerer vi naboers `pos` istedenfor `vel`. Vi vil finne en gruppes midtpunkt (eng. center of mass). Tar igjen og deler på antallet naboer for å finne gjennomsnittet. Før vi returnerer `cohesion_move` trekker jeg fra `self.pos`, dette gjør at vi får «offsett»-et eller endringen fra der hvor vi er nå.

`separation(self, neighbors)` sjekker også om vi har 0 naboer, returnerer en 0-vektor altså `Vector2(0,0)` om det ikke er naboer. Har vi naboer så starter beregningen. Itererer gjennom alle naboene, sjekker en etter en om distansen til nabo-boiden er mindre enn en konstant jeg har satt i `config.py` kalt `AVOIDANCE_RADIUS`. Hvis nabo-boid-en er innenfor denne radiusen inkrementeres `n_avoid` med 1 og `separation_move` inkrementeres med differansen av `self.pos` og `neighbor.pos`. Grunnen for dette er å samle opp distansen fra egen posisjon (`self.pos`) og naboens posisjon, den oppsamlede verdien vil peke mer og mer unna retningen til de naboene vi vil unngå desto flere naboer som er med i beregningen. Før vi returnerer noe sjekkes det om vi har noen naboer å unngå i det heletatt, hvis ja, del `separation_move` på `n_avoid` og lagre resultatet i `separation_move`, likt som de andre reglene tar vi effektivt gjennomsnittet her. Helt til slutt returnerer vi `separation_move`.

En problemstilling jeg satt med en stund var «hvordan få boids til å snu seg etter den retningen de beveger seg i?» For det er jo nesten det samme som «hvordan bevege seg i den retningen man 'ser' mot», men på en måte motsatt. Det var den sistnevnte løsningen jeg fant når jeg lette på internett, men heldigvis fant jeg og hjelpelæreren ut av dette. Jeg starter med å definere en vektor kalt `up_vector` som har verdien `Vector2(0, -1)`. Denne eksakte verdien er valgt på grunn av måten jeg tegner boiden på. Som du ser på fig. 2 er den tegnet slik at den peker oppover. Skal vi representere denne retningen med en normalisert vektor vil den ha en vinkel $\theta = 90$ (fra horisonten) og lengde 1, konvertere vi dette til kartesiske koordinater blir det $(\cos(90), \sin(90)) = (0,1)$, men siden vi tegner y-aksen fra topp til bunn, altså $y = 0$ er toppen og den øker nedover, må vi snu y-verdien. Resultatet blir da en vektor med verdi $(0, -1)$.

Figur 2 Sketch av boid

Grunnen til at jeg trenger en «opp-vektor» er at jeg vil regne distansen (i grader) *fra* opp, som er lik rotasjon 0 grader, *til* den retningen jeg ønske å peke. Å finne rotasjonen er i seg selv ganske enkel og utføres med én linje kode: `self.angle = self.vel.angle_to(self.up_vector) % 360`. Jeg bruker

`Vector2`-klassens innebygde metode for å finne vinkelen til en annen vektor, når den er funnet tar jeg svaret modulo 360, siden vi aldri trenger høyere rotasjon enn 360 grader.

For å unngå grafiske artefakter (eng. graphical artifacts) som jeg støtte på da jeg prøvde dette første gang, utfører jeg et rotasjonskall på *originalbildet* istedenfor objektets bilde: `self.image = pygame.transform.rotate(self.game.boid_img, self.angle)`. Bildet jeg lagret i Game-klassen blir brukt her som utgangspunkt, og objektets bilde blir satt til resultatet av rotasjonen av det originale bildet. Grunnen til at jeg presiserer dette er at jeg først prøvde å kalle `pygame.transform.rotate()` med `self.image`, istedenfor `self.game.boid_img`, dette førte til mye hodebry før jeg skjønte at jeg måtte ha en «kopi» av originalen slik at jeg ikke maltrakterte objektets bilde mer og mer for hver bildeoppdatering.

Løsningen på problemstillingen «hvordan holde boids-ene innenfor vinduet, uten at de flyr avgårde (boid in the void)» er litt «billig», men jeg mener det er adekvat. Løsningen finnes i Boid-klassens metode `wrap()`, denne tar x- og y-koordinatet til `self.pos` og utfører en modulo operasjon på de med vinduets henholdsvis bredde og høyde. Dette fører til litt unaturlig bevegelse når en boid treffer vinduskanten, men det holder dem i det minste innenfor vinduet.

Hoiks er implementert slik at de arver fra `Boid`-klassen, dette var en rask løsning i øyeblikket, men jeg vurderer i ettertid at dette kunne vært gjort på en annen måte. Grunnen til det er at hoiks ikke følger *alle* de samme reglene til boids. Jeg overskriver alignment-regelen med å bare returnere `self.vel`, for å signalisere «ingen endring». Det er ønskelig at hoiks jager en flokk, ikke at en prøver å lede eller direkte følge dem. Separation-regelen er også endret på, her returnerer jeg kun en null-vektor (`Vector2(0,0)`), for å si null endring. Den eneste regelen som ikke er endret på er cohesion, dette fordi den hjelper hoik å styre mot midten av en flokk, som er ønskelig oppførsel. Det sjekkes hver bildeoppdatering om det fins boids som er nære nok til at en hoik kan drepe dem. Jeg har modifisert get_neighbors() metoden slik at hoiks kan «se» 10 ganger lengre enn boids.

Alle konstanter for programmet er lagret i `config.py`.

Instruksjoner for brukergrensesnittet fins i en egen `README.md` fil, sees best med markdown-kompatibelt tekstverktøy.

5 Evaluering

Examine if your submission fulfils the requirements and what shortcomings exist.

In this solution, all requirements are fulfilled, but collision detection between the ball and paddle is inaccurate, due to differences between the visual representation and the implementation...

Jeg har evaluert oppgaven slik at jeg mener jeg oppfyller kravene. Jeg bruker objektorientert programmeringsdesign og virkemåte. Jeg har klasser som arver. Reglene for Boids er korrekt implementert. Hoiks er med, men hindre klarte jeg ikke få til. Jeg har beskrevet litt i Teknisk

bakrgrunn om hvordan arv fungerer og hvordan jeg har valgt å bruke det. UML diagram fins i Fig. 1.

Implementasjonen *ser* kanskje ikke så pen ut, men det fungerer; boids flyr, de samler seg i flokk, hoik kommer og spiser dem. Det er ingen hindre i oppgaven annet enn hoik-en selv, som boids dog ikke tar mye hensyn til.

6 Diskusjon

Discuss what could be done better, problems you had, experiences etc. (we also appreciate feedback on the assignment or group sessions).

The implementation of the paddle-ball collision could be done some other way, but due to some reason, the current implemetation is better. After spending two days trying to write the report in LATEX, I gave up, and wrote it in Word instead.

Jeg hadde noe problem med å få rotasjonen til å fungere. Altså dette med å få boids til å rette seg etter retningen de bevegde seg i. Med god hjelp fra hjelpelærer fiska vi det. Var bare en ekstra vektor og litt triksing som skulle til.

Hadde jeg hatt mer kjennskap til pygame biblioteket skulle jeg ha lagt inn mer interessant grafikk, men jeg vil påstå at innleveringen er fullført i henhold til de krav satt i oppgaven.

7 Konklusjon

Sum up the previous sections.

I have implemented a solution that fulfills the requirements, the implementation is moderately buggy, but does not crash too much..

Jeg har implementert en løsning som nesten fyller alle kravene satt i oppgaven. Implementasjonen er rimelig fungerende, bare ikke skap mer enn et par hundre boid-instanser, da blir programmet ganske tregt, som følge av implementasjonen min. Jeg har en lite effektiv måte å sjekke for naboer på, så ved mer enn ca. 200 boids faller det fra hverandre på min maskin, og ved 600 eller mer sitter jeg med 5 FPS.

8 Referanser

[1] The Dia Developers. Dia website, 2014.

URL http://dia-installer.de/shapes/UML/index.html.en.

[2] Python Software Foundation. Python language reference, version 3.4, 2014.

URL http://www.python.org.

[3] Pygame wiki - RotateCenter

http://www.pygame.org/wiki/RotateCenter?parent=CookBook%22here%22

[4] Pygame docs

https://www.pygame.org/docs