Modul 4-9

Konstruktion af et simpelt akvarium af fisk og simulering af flokadfærd ved brug af vektorbiblioteket og objektorienteret programmering

Indhold

1 Agentbaseret modellering og simulering af et akvarium ved brug af vektorbiblioteket

I denne forløbsoversigt anvendes vektorbiblioteket fra forrige artikel til at simulere et simpelt akvarium med forskellige fisk.

Endelig indføres i detaljeret pseudokode en beskrivelse af Reynolds algoritme til at simulere flokadfærd.

Det svarer til en detaljeret beskrivelse af modul 4-9.

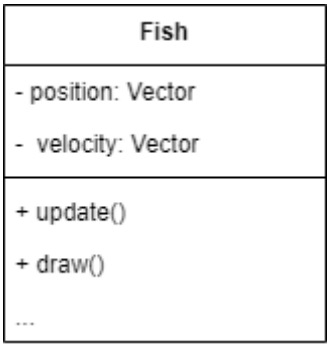
I denne del fokuseres på principperne for nedarvning og association. Eleverne skal have en forståelse af:

- Betydningen af nedarvning og hvordan den bruges i praksis
- Forskellen på børn og super-klassen herunder generalisering og specialisering
- Hvorledes implementeres nedarvning og associationer mellem klasser i praksis

1.1 Modul 4

Til at beskrive en fisk oprettes en ny klasse, der rummer data og metoder, der beskriver fiskens adfærd.

Igen vil det være oplagt at starte med at lave et UML-klassediagram. Herunder en grov skitse, der illustrerer, hvorledes fisken består af to vektorer, der beskriver dens placering og dens fart i hhv. x- og y-retningen.



På konceptuelt niveau kunne tegnes en ”association” mellem Vector og Fish. I den sammenhæng er det relevant at belyse de forskellige former for association, herunder ”aggregation” og ”komposition”. Øvelserne inddrager eksempler på disse.

Metoderne ”update()” og ”draw()” skal bruges til hhv. at opdatere fiskens position og tegne figuren. Førstnævnte opdaterer positionen ved at addere fartvektoren til positionsvektoren. I pseudokode:

```
position.x = position.x + velocity.x
position.y = position.y + velocity.y
```

Modul 4-9

Herunder en tentativ klasse i pseudokode, hvor der tjekkes for, at fisken bliver inden for akvariet angivet ved height og width. Dette kunne også være en øvelse for eleverne.

```
class Fish{
  constructor(location: Vector,velocity:Vector)
  {
    this.location: Vector = location
    this.velocity: Vector = velocity
    this.size: float = 50
  }

  update()
  {
    this.location.add(this.velocity);
  }

  draw()
  {
    // check that the fish is inside the canvas
    if ((this.location.x > width) || (this.location.x < 0)) {
     this.velocity.x = this.velocity.x * -1;
    }
    if ((this.location.y > height) || (this.location.y < 0)) {
     this.velocity.y = this.velocity.y * -1;
    }
    Draw fish from the location (this.location.x,this.location.y,this.size);
  }
}
```

1.2 Modul 5

Det bør betones over for eleverne, at vi skelner mellem to former for nedarvning:

- Specialisering sker, når der oprettes nye underklasser.
- Generalisering sker, når der ekstraheres fælles attributter og metoder fra underklasserne og disse placeres i superklassen.

Eleverne kan øve sig i brugen af nedarvning og disse principper ved:

1. I mindre grupper at oprette mindst to typer fisk (kunne være en jægerfisk, der forsøger at fange en byttefisk).
2. Benytte generalisering og specialisering til at konkretisere hvilken adfærd og egenskaber, de har til fælles og hvad der adskiller dem.
3. Modellere dette ved hjælp af klassediagrammer først, som man godt må iterere over og ændre på undervejs.

Modul 4-9

Det er vigtigt, at eleverne får lov til at sætte deres eget præg på de forskellige fisk. De har fået givet en superklasse med adfærd, som de kan bruge, modificere og dernæst skabe deres egen. Giv gerne eleverne mulighed for at præsentere i plenum. Ekspliciter desuden med eksempler hvorledes nedarvning implementeres i praksis.

1.3 Modul 6

Herefter bør børneklasserne implementeres i faktisk kode, og man kan bruge overskrivning og dynamisk binding eller overloading og statisk/sen binding, hvilket også er kendt som polymorfisme. Det er et svært begreb for mange elever at forstå, og man bør derfor tilbyde ekstra facilitering i form af "worked examples" og eksemplariske eksempler for det valgte sprog.

Eksempelvis kunne foreslås at overskrive måden fiskene tegnes på, hvordan de svømmer, deres farver mm., så fiskene ser forskellige ud og opfører sig forskelligt. Der bør løbende testes, om fiskene opfører sig hensigtsmæssigt.

Det er væsentligt at betone begrebet dynamisk binding, som en særdeles nyttig egenskab ved objektorienteret programmering. Ved dynamisk binding er det objektets type, der afgør metodens kald. Dette er også kendt som dynamisk polymorfisme. Herunder et eksempel med en jægerfisk:

```
Fish f; //reference til forældre Fisk
f = new HunterFish() //HunterFish er barn af Fisk
f.draw()
```

Det leder til, at man meget elegant og kompakt kan opbevare og kalde deres respektive metoder. Herunder ses den pseudokode, der illustrerer hvordan fiskene, som er lagt ind i et array kaldet "fishes", kan tegnes og hvordan positionen opdateres med ganske få linjers kode:

```
foreach F in fishes:
  F.update()
  F.draw()
```

For at hjælpe eleverne på vej bør der faciliteres simple eksempler på dynamisk binding og overskrivning, som de kan bygge videre på. Tilsvarende gør sig gældende for statisk binding og overloading.

2 Modellering og implementering af en variant af Boids algoritme til at simulere flokadfærd

Craig Reynolds flokalgoritme er udvalgt af flere grunde. Fra et programmeringsteknisk perspektiv er den et glimrende eksempel på nyttigheden af både vektorer og objekter. Førstnævnte gør koden

kortere og meget mere elegant. Sidstnævnte er en god træning i at få objekter til at kommunikere med hinanden.

Eleverne trænes bl.a. i at:

- Anvende vektorer og objekter til at undersøge et problemområde og herigennem opnå ny erkendelse omkring flokadfærd.
- Benytte vektorer og interagerende objekter til at forstå og implementere simple regler i form af metoder hos fiske-objektet.

2.1 Modul 7

De sidste tre moduler er opbygget efter "Use-modify-create"-princippet. I det første modul skal eleverne have en forståelse for de grundlæggende regler, som de autonome fisk skal overholde:

- Seperation: En fisk vil forsøge at svømme væk fra andre fisk, der er tæt på den. Men ligesom i naturen vil ingen fisk have viden om alle fisk i flokken.
- Justering (alignment): En fisk vil forsøge at efterleve hastigheden af andre fisk i nærheden.
- Samhørighed (cohesion): En fisk vil forsøge at svømme mod centrum af flokken.

Det kan være en ide at udlevere et skelet eller færdigt program, hvor eleverne ved hjælp af sliders/skydere kan justere på forskellige parametre og prøve at forstå samt kommentere på koden.

Som udgangspunkt er fiskens klasse konstrueret ud fra to centrale attributter (positionen og farten) hvor begge er repræsenteret som vektorer:

```
class Fish{
  constructor(position,velocity)
  {
    //..
    this.position = position //(x,y) peger på fiskens placering
    this.velocity = velocity // (xvel,yvel) peger på fiskens hastighed
    //...
  }
}
```

2.1.1 Pseudokode for den abstrakte algoritme

Den centrale del handler om at opdatere deres position i forhold til de tre regler. Heldigvis kan det gøres både ret let og elegant ved brug af vektorer, som resulterer i tre fartvektorer, der kan summes til den samlede fart:

```
Vector v1,v2,v3
for hver fisk F:
 v1 = F.seperation()  //kaldt seperation på dansk
 v2 = F.alignment()   //kaldt justering på dansk
 v3 = F.cohesion()    //kaldt samhørighed på dansk
```

```
F.velocity += v1 + v2 + v3
F.position += F.velocity
```

Bemærk, at de tre metoder implementeres i fiskeklassen, der hver især repræsenterer en af de tre regler. Fartvektoren opdateres og ud fra det opdateres positionen (bemærk `+=` i de sidste to linjer).

Som det indirekte fremgår af pseudokoden, kommer man ikke uden om at skulle have en løkke i en løkke, da man er nødsaget til for hvert fiskeobjekt at kommunikere med alle andre fiskeobjekter om deres placering og fart. Desuden inddeles ovenstående pseudokode i nogle mindre metoder for at gøre det mere overskueligt (”separation of concerns”).

I det følgende gennemgås i detaljer, hvorledes de tre regler kan implementeres.

2.1.2 Samhørighed: Fisk søger mod centrum af naboerne

Givet ***N*** fisk, $F_1, \ldots, F_N$,med hver deres positionsvektor, $pos_1, \ldots, pos_N$ „ da kan konstrueres en vektor, kaldet $\mathrm{pCenter}$, som betegner et opfattet (”percieved”) centrum, hvor hver fisk vægtes ligeligt, ved at bruge addition og skalering med (1/**N**):

$$pCenter = \frac{(F_1.pos_1 + \cdots + F_n.pos_N)}{N}.$$

Man kan diskutere eller overveje om det er hensigtsmæssigt, at fiskens egen position inkluderes i beregningen, hvilket der tages højde for herunder i pseudokoden:

```
F.cohesion():
  Vektor pCenter
  for hver fisk G:
    Hvis F !=G:
      pCenter += G.position
  pCenter = pCenter / N-1
  returner (pCenter - F.position) / 100
```

Divisionen med 100 er for at sikre, at den ikke bevæger sig for hurtigt mod centrum. Ved at dividere med 100 svarer det til, at den bevæger sig 1 procent tættere på det opfattede centrum. Ydermere kunne det overvejes, hvor stor ***N*** bør være. I øvelserne opfordres til at der indføres skyder, som justerer på denne parameter.

2.1.3 Seperation: Fisk søger væk fra nærmeste naboer

Givet en fisk F kigges her på enhver anden fisk G. Målet er at undersøge om G er inden for en prædefineret afstand, *D*. I så fald flyttes F væk. I praksis ved at trække fra en vektor pCenter forskydningen af alle fisk G, som er i nærheden. Til at starte med sættes pCenter til nul, da den resulterende vektor skal flytte fisken fra den nuværende position og væk fra de omkringliggende.

Herunder pseudokode:

Modul 4-9

```
F.seperation():
 Vektor pCenter= 0 //nulvektoren
 for hver fisk G:
  Hvis F !=G:
   Hvis |G.position - F.position| < D :
     pCenter = pCenter - (G.position - F.position)
 returner pCenter
```

2.1.3 Justering: Fiskene søger samme gennemsnitshastighed som nærmeste naboer

Denne regel minder en del om reglen om seperation. I stedet for at regne vægtene af positionen, udregnes den gennemsnitlige fart.

Der indføres endnu en parameter, chgVelocity, til at justere, hvor meget fiskens fart i praksis ændres ud fra den beregnet gennemsnitsfart. Igen kan det være godt at indføre en skyder og variere chgVelocity:

```
F.alignment():
 Vektor pVelocity;
 for hver fisk G:
  Hvis F !=G:
   pVelocity +=  G.velocity
 pVelocity = pVelocity/N-1
 returner (pVelocity-G.velocity)/chgVelocity
```

Et godt bud på en værdi for chgVelocity kunne være mellem 8 og 12.

2.2 Modul 8

På baggrund af programmet og pseudokoden, der illustrerer algoritmen, kan eleverne efterfølgende implementere den i deres fiskeklasse eller et af børnene. Vigtigt at betone er, at der arbejdes med små skridt. Dvs. en regel ad gangen og løbende test.

Vær opmærksom på, at ud over reglerne beskrevet i algoritmen, bør man indføre nogle globale parametre, såsom maksimum- og minimumhastighed for fiskene, hvor mange fisk der måles på i den beskyttede zone (de fisk de søger væk fra) og den visuelle zone (de fisk som er inden for deres synsfelt).

For nogle vil det tage længere tid end et modul at implementere alle tre regler. Det er vigtigt at give grupperne den tid, som det nu tager. Alternativt kan det for nogle elever være nok at implementere bare en af reglerne og få eksempler fra de andre og måske i stedet bruge tiden på at forstå, hvad der sker eller dokumentere given kode.

2.3 Modul 9

Til de elever, som er blevet færdige med at implementere reglerne foreslås, at man udvider eksempelvis med følgende:

- Fiskene bevæger sig mod et bestemt punkt eller – omvendt – flygter fra et bestemt punkt
- En vektor beskriver strømninger i vandet, der påvirker fiskenes hastighed
- Når tilfældige fisk nærmer sig bunden, holder de lejlighedsvis en kortere pause før de svømmer mod flokken igen
- Grupper af fisk opfører sig biased i forhold til hele gruppen. Det kan eksempelvis være, at en bestemt andel af fiskene kender til et område, hvor der ligger mad eller lignende og derfor foretrækker sig at opholde sig dér

Såfremt eleverne ikke har tid til at udvikle den nye regel, kan man stille krav om, at man forsøger at formulere en pseudokode for, hvorledes det skulle gøres.

Der evalueres ved en ”Expo”, hvor 2/3 af gruppemedlemmerne bevæger sig rundt og ser og hører om, hvad de andre grupper har lavet. Den sidste i hver gruppe står tilbage og præsenterer for andre gruppers medlemmer. Man sørger hele tiden for at skifte ud, så forskellige står for at præsentere.

Endelig evalueres formativt i plenum.

3 Øvelser

3.1 Om animationer af fisk

Herunder følger skitser af en række øvelser, som udvider økosystemet eller akvariet med ny funktionalitet. Igen bør man løbende opdatere klassediagrammet med nye metoder og attributter samt teste og dokumentere sin kode.

1. Konstruer klassediagrammer med metoder og attributter for de to slags fisk.

2. Implementer klassen fisk, så fisken tegnes som en trekant og at de kan bevæge sig rundt på skærmen.

3. Sørg for at fiskene "bouncer" af på kanterne af skærmen i en tilpas afstand f.eks. med en margin på 100 pixels. Det giver en mere naturtro effekt.

4. Overskriv måden fiskene tegnes på i de to børn, så jægerfiskene og byttefiskene ser forskellige ud i forhold til form og farve.

5. Udvid byttefiskenes måde at svømme på, så de svømmer væk fra jægerfisk.

6. Udvid jægerfiskene så de forsøger at fange byttefisk.

7. Hvis jægerfisk kolliderer med byttefisk, så spises byttefisk. Dvs. den forsvinder og jægerfisk bliver en lille smule større i næste frame.

8. Udvid så jægerfiskene undervejs og helt tilfældigt holder nogle små pauser, hvor de "står stille" i vandet.

9. Udvid med en ny klasse "Food", som dumper ned for oven og lander i bunden. Der må max ligge 5 stykker mad på bunden til et givent tidspunkt.

10. Hvis byttefiskene kolliderer med maden, svarer det til at de har spist maden. Spiser de noget mad, skal de tegnes en lille smule større i næste frame.

11. Udvid byttefiskenes svømmemønster, så ud over at undgå jægerfisk forsøger de også at fange maden. Lad afstanden bestemme, hvilket mål de prioriterer.

3.2 Øvelser om implementering af flokadfærd hos fisk

Herunder følger skitser af en række øvelser, der skal udvide akvariet så særligt byttefiskene har tendens til at svømme i flok. Igen bør man løbende opdatere klassediagrammet med nye metoder og attributter samt teste og dokumentere sin kode.

1. Udvid fiskeklassen med en metode, så den kan håndtere seperation. Inddrag gerne relevante parametre så det er muligt at justere på graden af seperation.

2. Udvid fiskeklassen med en metode, så den kan håndtere samhørighed. Dvs. den svømmer mod centrum af flokken. Inddrag gerne relevante parametre, så det er muligt at justere på graden af samhørighed.

3. Udvid fiskeklassen med en metode, så den kan håndtere justering. Dvs. hver fisk forsøger at svømme med samme hastighed som resten af flokken. Inddrag gerne relevante parametre, så det er muligt at justere på graden af samhørighed.

4. Indfør skydere der kan justere på de parametre og prøv at finde en fornuftig balance mellem de valgte parametre, så det ser mere realistisk ud.

5. Udvid fiskenes adfærd, så når de nærmer sig bunden, holder de en kortere pause før de svømmer hen til flokken igen. Igen kan man indføre skydere, der justerer på hvor mange, hvor ofte de gør det og hvor længe ad gangen.

6. Udvid adfærden så grupper af fisk opfører sig biased i forhold til hele gruppen. Det kan eksempelvis være, at en bestemt andel af fiskene kender til et område, hvor der ligger mad eller lignende og derfor foretrækker de at opholde sig dér.

7. Udvid, så fiskene som flok bevæger sig mod et bestemt punkt eller – omvendt - flygter fra et bestemt punkt. Eksempelvis en fjendtlig fisk.

8. Udvid med vektorer, der beskriver strømninger i vandet og som påvirker fiskenes hastighed

9. Modeller en ny klasse, der kaldes for "Flock". Klassen skal håndtere flokke af fisk, der opfører sig ens og samle alle parametre som kan bruges til at justere flokken. Overvej, hvilken relation/association, der er mellem "Flock" og "Fish".

10. Implementer klassen "Flock", så man kan oprette en flok bestående af N fisk, der opbevarer og justere på alle relevante parametre for flokken. På denne måde kan akvariet have flere (forskellige) flokke eller arter med hver deres egenskaber.