# Del III:
# Fuglesimulering (180p)

Del 3 av eksamen er programmeringsoppgåver. Denne delen inneheld **11 oppgåver** som til saman gir ein maksimal poengsum på ca. **180 poeng** og tel ca. **60 %** av eksamen. Kvar deloppgåve kan gi ein poengsum frå **5 - 25 poeng** avhengig av vanskegrad og arbeidsmengd.

Filene som er gitt ut inneheld kompilerbare (og køyrbare) `.cpp`- og `.h`-filer med kode og ei full beskriving av oppgåvene i del 3 som ei PDF. Etter å ha lasta ned koden står du fritt til å bruke eit utviklingsmiljø (VS Code) for å jobbe med oppgåvene. Vi anbefaler på det sterkaste at å kontrollere at koden kompilerer og køyrer før du sett i gang med oppgåvene.

## Sjekkliste ved tekniske problem

Viss prosjektet du oppretter med koden som er gitt ut ikkje kompilerer (før du har lagt til eigne endringar) bør du rekkje opp hånda og be om hjelp. Mens du venter kan du prøve følgjande:

1. Sjekk at prosjektmappa er lagra i mappa `C:\temp`.
2. Sjekk at namnet på mappa **IKKJE** inneheld norske bokstavar (*Æ, Ø, Å*) eller mellomrom. Dette kan skape problem når du prøver å køyre koden i VS Code.
3. Sørge for at du har:
   - Last ned `.zip`-fila frå Inspera og pakk den ut (unzip). Lagre fila i `C:\temp` som **IO2TDT4102_kandidatnummer**. Du skal altså skrive *ditt eige kandidatnummer* etter understreken.
   - Opne mappa i VS Code. Bruk deretter følgjande TDT4102-kommandoar for å oppretta eit fungerande kodeprosjekt:
     (a) `Ctrl+Shift+P` → `TDT4102: Force refresh of the course content`
     (b) `Ctrl+Shift+P` → `TDT4102: Create project from TDT4102 template` → `Configuration only`
4. Sørg for at du er inne i rett fil i VS Code.
5. Køyr følgjande TDT4102-kommandoar:
   (a) `Ctrl+Shift+P` → `TDT4102: Force refresh of the course content`
   (b) `Ctrl+Shift+P` → `TDT4102: Create project from TDT4102 template` → `Configuration only`
6. Prøv å køyr koden igjen (`Ctrl+F5` eller `Fn+F5` eller `F5`).
7. Viss det framleis ikkje fungerer, lukk VS Code vindauget og opne det igjen. Gjenta deretter steg 5-6.

## Introduksjon

Fuglesimuleringa er ein enkel simulering av korleis fuglar flyr i flokkar. Koden for fuglesimuleringa er inndelt i tre hovuddeler:

- Eit `Application`-objekt som styrer funksjonsbaren på toppen av animasjonsvindauget, inn-/ut-datahandtering og hovudløkka som teikner kvar frame til skjermen og ber eit `Simulator`-objekt om å oppdatere tilstanden til fuglane.

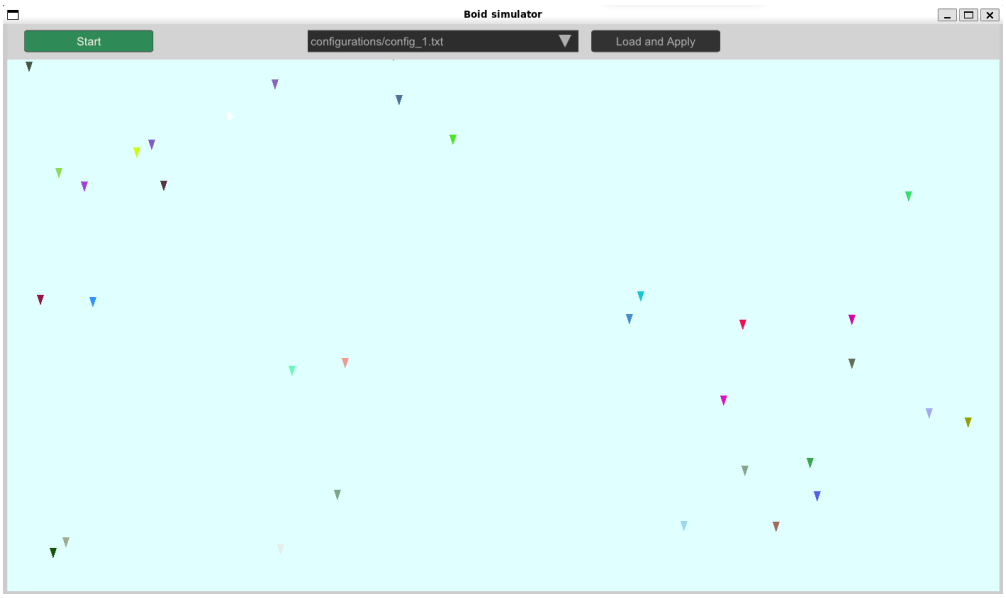
- Eit `Simulator`-objekt som har oversikt over alle fuglane i simuleringa.
- Ein abstrakt baseklasse, `Bird`, som er ansvarleg for å oppdatere posisjonen og hastigheita til ein fugl, og å teikne fuglen til skjermen for kvar frame.

Den abstrakte baseklassen `Bird` har to subklassar; `Doves` og `Hawks`. Duene følgjer reglane skrivne under. Haukane flyr litt tilfeldig over skjermen og ønskjer å unngå andre haukar. Du treng ikkje ta stilling til haukane før i siste oppgåve.

Reglane som duene skal følgje er basert på fire flokkeigenskapar:

- **Separasjon (separation):** Fuglen styrer for å unngå kollisjon med andre fuglar i flokken.
- **Samanstilling (alignment):** Fuglen styrer i same retning som resten av flokken.
- **Samhelde (cohesion):** Fuglen styrer mot sentrum av flokken.
- **Unngåelse (avoidance):** Fuglen styrer for å unngå jegerfuglar.

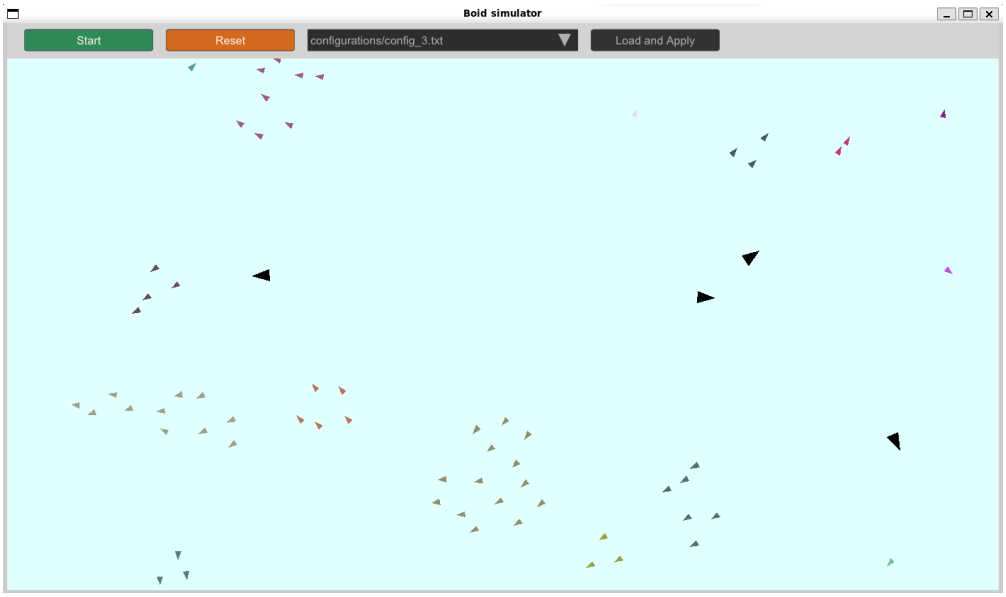
I `.zip`-fila finn du kodeskjelettet med klart markerte oppgåver.




**Figur 1:** Skjermbilete av køyring av koden som er gitt ut. I animasjonsvidauget kan ein sjå ein funksjonsbar som består av ein Start/Stopp-knapp, ein nedtrekksmeny for å laste inn ulike konfigurasjonar, og ein knapp for å bruke konfigurasjonen ein har valt (Load and Apply). I tillegg ser ein 32 fuglar i vindauget.

Før du begynner på oppgåvene bør du sjekke at koden som er gitt ut køyrer uten problem. Du skal sjå omtrent det same vindauget som i Figur 1. Fuglane vert plassert tilfeldig i vindauget, så startposisjonen til dei åtte fuglane vil variere og er ikkje viktig.

Fuglesimuleringa i koden som er gitt ut inneheld minimalt med funksjonalitet (Figur 1). Når alle oppgåvene er gjort vil ein derimot kunne sjå at fuglane flyr i flokkar mens dei prøver å unngå jegerfuglane som flyr rundt i tilfeldige banar (Figur 2).




**Figur 2:** Skjermbilete av den køyrande simuleringa når den er ferdigstilt.

## Korleis svare på oppgåvene

Kvar oppgåve i del 3 har ein unik kode for å gjere det lettare for deg å vite kor du skal fylle inn svara dine. Koden er på formatet `<T><siffer>` (*TS*), der siffera er mellom 1 og 11 (*T1 - T11*). For kvar oppgåve vil ein finne to kommentarar som skal definere høvesvis begynninga og slutten av svaret ditt. Kommentarane er på formatet: `// BEGIN: TS` og `// END: TS`.

For eksempel ser oppgåve *T2* i koden som er gitt ut slik ut:

```
// Task T2: Update the position of the bird using
// on its current position and velocity.
void Bird::updatePosition()
{
// BEGIN: T2
    ;
// END: T2
}
```

**Det er veldig viktig at alle svara dine verte ført mellom slike par av kommentarar**. Dette er for å støtte sensurmekanikken vår. Viss det allereie er skriven kode *mellom* `BEGIN`- og `END`-kommentarane til ei oppgåve i kodeskjettet som er gitt ut, så kan du, og ofte bør du, erstatte denne koden med din eigen implementasjon. All kode som er skriven *utanfor* `BEGIN`- og `END`-kommentarane **SKAL** du la stå som den er. I oppgåve T6 og T9 er `BEGIN`- og `END`-kommentarane plassert utanfor deklarasjonen til funksjonen, noko som opnar for å bruke eigne hjelpefunksjonar og globale variablar. Du skal ikkje endre navn eller parameter til desse funksjonane.

**Merk:** Du skal **IKKJE** fjerne `BEGIN`- og `END`-kommentarane.

Dersom du synast nokon av oppgåvene er uklare kan du oppgje korleis du tolkar dei og det du antar for å løyse oppgåva som kommentarar i koden du leverer.

**Tips:** Trykkjer ein `CTRL+SHIFT+F` og søker på `BEGIN:` får ein snarvegar til starten av alle oppgåvene lista opp i utforskervindauget så ein enkelt kan hoppe mellom oppgåvene. For å kome tilbake til det vanlege utforskervindauget kan ein trykkje `CTRL+SHIFT+E`.

## Korleis levere del 3

Når du er ferdig med oppgåvene og er klar til å levere skal du laste opp alle `.h`- og `.cpp`-filene i hovudmappa som ei `.zip`-fil i Inspera. Du står fritt til å bruke den innebygde funksjonen i VS Code til å lage `.zip`-fila. Desse filene inkluderer:

- `Application.h`
- `Application.cpp`
- `Simulator.h`
- `Simulator.cpp`
- `main.cpp`

## Oppgåvene

Oppgåvene vil ha følgjande struktur:

Første del inneheld motivasjon, bakgrunnsinformasjon og korleis koden er satt saman for oppgåven.

> Neste del er ein tekstboks som inneheld oppgåven og spesifikke krav.

Siste del forklarar resultatet du kan forvente når du har fullført oppgåven.

1. (10 points) **T1: Overlast + operatoren**

   I koden som er gitt ut blir det brukt ein hastigheitsvektor for å bestemme farta og retninga til fuglen, og ein posisjon for å bestemme plasseringa til fuglen. Begge er lagra som ein struktur av typen `FloatingPoint` som inneheld to flyttal:

   ```
   struct FloatingPoint {
       double x;
       double y;
   };
   ```

   Vi ønskjer å kunne leggje saman to `FloatingPoint`-strukturar utan å måtte aksessere felta i strukturane direkte kvar gong. Addisjon av to vektorar blir utført på følgjande måte:

   $$(x_1, y_1) + (x_2, y_2) = (x_1 + x_2, y_1 + y_2)$$

   > Overlast + operatoren for `FloatingPoint` i `Simulator.cpp`.

   Når oppgåve T1 er ferdig skal det vere mogleg å leggje saman to strukturar av typen `FloatingPoint`.

2. (5 points) **T2: Få fuglane til å bevege seg**

   Nå står alle fuglane heilt i ro når ein trykkjer på start. Det er fordi funksjonen `updatePosition` i `Bird` ikkje er implementert og fuglane blir teikna på same posisjon i kvar frame.

   > Implementer funksjonen `updatePosition` i `Simulator.cpp`.
   >
   > - Oppdater posisjonen (position) til fuglen ved å leggje saman hastigheita (velocity) og posisjonen (position) til fuglen.

   Når oppgåve T2 er ferdig skal duene fly i rette linjer over skjermen viss ein klikkar på `Start` og stå i ro viss ein deretter klikkar på `Stop`.

3. (10 points) **T3: Legg til tilbakestillingsknapp**

   Nå som fuglane flyttar på seg mellom start og stopp av simuleringa ønskjer vi å kunne tilbakestille simuleringa så ein kan køyre simuleringa på nytt uten å måtte restarte hele programmet. `Simulator` har ein klasse `ResetButton` som er definert i `Simulator.h`:

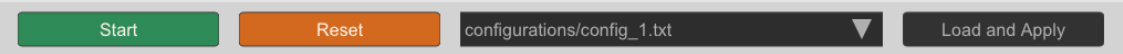
```
class ResetButton : public Button {
public:
    ResetButton ();
    ~ResetButton ();

    static void callback();
    static void setSimulator(Simulator* const _sim);

private:
    static Simulator *sim;
};
```

   > Legg til ein knapp av typen `ResetButton` i `run`-funksjonen til `Application` i `Application.cpp`.

   Når oppgåve T3 er ferdig skal funksjonsbaren til simuleringa sjå ut som i Figur 3.




**Figur 3:** Skjermbilete av funksjonsbaren etter at oppgåve T3 er fullført.

4. (10 points) **T4: Tilbakestill simuleringa**

   Tilbakestillingsknappen du la til i forrige oppgåve gjer foreløpig ingenting når ein klikker på den. Det er fordi callback-funksjonen ikkje er implementert enda.

   > Implementer callback-funksjonen til tilbakestillingsknappen i `Simulator.cpp`.
   >
   > - Tilbakestill simuleringa ved å utnytte simulatorobjektet sin `resetSimulation`-funksjon.

Når oppgåve T4 er ferdig skal tilbakestillingsknappen sette fuglane i startposisjon når den blir klikka på. Noter deg at knappen kun fungerer når simuleringa er stoppa og at fuglane blir plassert ulikt etter kvar tilbakestilling.

5. (20 points) Vi ønskjer å kunne teste fleire samansetninger av fuglar. For å gjere dette har vi eit sett med konfigurasjonar i form av `.txt`-filer. Disse filene finn du i mappa som hetar `configurations`. Kvar fil består av to tal som er separert med eit linjeskift. Det første talet representerer antall duer i simuleringa, mens det andre talet representerer antall haukar i simuleringa.

   Implementer funksjonen `loadAndApplyConfiguration` i `Application.cpp`.

   - Utløs eit passande unntak dersom fila ikkje kan opnast.
   - Les inn tala frå den gitte konfigurasjonsfila.
   - Sørg for at konfigurasjonen som blir lest inn blir brukt i simuleringa. Simulator-objekt har ein funksjon som heiter `applyConfiguration()`.

   Når oppgåve T5 er ferdig skal du kunne sjå ei endring av antal fuglar på skjermen når du trykkjer på knappen `Load and apply` etter at du har valt ein konfigurasjon frå nedtrekksmenyen.

6. (20 points) **T6: Ven eller fiende**

   For at duene skal kunne vite korleis dei skal forflytte seg i neste frame ønskjer dei ei oversikt over fuglane rundt seg. Det er to typar fuglar i simuleringa; duer (doves) og haukar (hawks). Duene ser på dei andre duene som potensielle vener og hauker som potensielle fiendar. I tillegg er duene sitt synsfelt avgrensa av dei globale variablane `FRIEND_RADIUS` for vener, og `AVOID_RADIUS` for fiendar. Alle duer som er utanfor `FRIEND_RADIUS` og alle haukar som er utanfor `AVOID_RADIUS` skal derfor bli ignorert. Dette er illustrert i Figur 4. Vener og fiendar blir lagra i kvar sin tabell (`vector`) av delte peikare (`shared_ptr`) til fugleobjekt (`Bird`). Funksjonen du skal implementere skal kallast på kvart `Bird`-objekt for kvar frame.

   Avstanden mellom to punkt blir regna ut på følgjande måte:

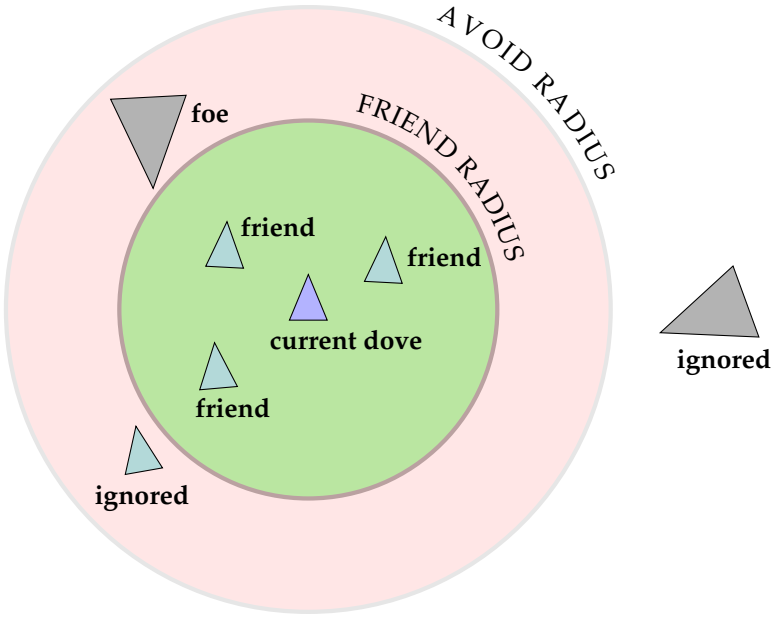
$$\text{distance}((x_1, y_1), (x_2, y_2)) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

   Implementer funksjonen `makeFriendsAndFoes` i `Simulator.cpp`.

   - Fjern elementa som allereie ligg i vektorane `friends` og `foes` frå forrige frame.
   - Filtrer dei andre fuglane i simuleringa basert på type. Merk at lista funksjonen tek inn inneheld alle fuglar. Du må altså hoppe over din eigen Bird-instans.
   - Pass på at vener er innanfor `FRIEND_RADIUS` og at fiendar er innanfor `AVOID_RADIUS`.

   **Hint:** Ein kan finne informasjon om korleis ein bruker matematiske funksjonar som er implementert i standardbiblioteket ved å gå til C++ reference og sida som heiter *Common math functions*.

   **Merk:** `BEGIN-` og `END`-kommentarane står utanfor funksjonsdefinisjonen, noko som opnar for moglegheiten til å lage eigne hjelpefunksjonar eller globale variablar for å løyse oppgåven.




**Figur 4:** Illustrasjon av kva for fuglar som er vener og fiendar for ein spesifikk due (lilla trekant). Dei andre duene er illustrert med blå trekantar, mens haukane er illustrert med grå trekantar.

Resultatet av oppgåve T6 kan ein ikkje verifisere visuelt.

7. (15 points) **T7: Bevegelseslogikken**

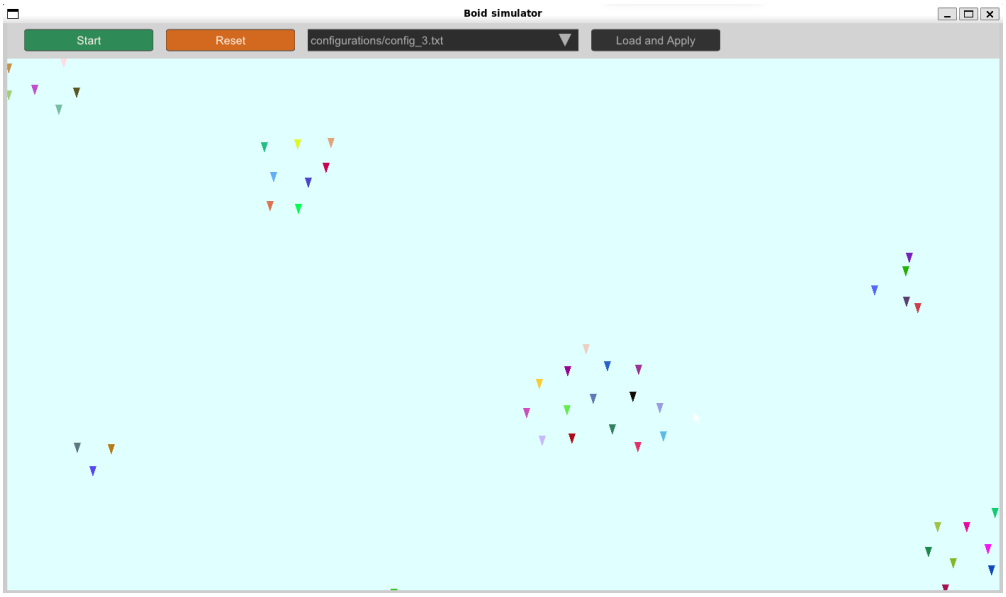
Vi ønskjer no å få duene til å følgje reglane vi introduserte i introduksjonen til koden. Dette blir gjort ved å oppdatere hastigheiten til duene. Reglane er allereie implementert i funksjonane `calculateCoherence`, `calculateAlignment`, `calculateSeparation` og `calculateAvoidance`. Disse funksjonane regnar ut korelis hastigheiten til ei due må forandre seg for at den skal overhalde reglane. Den nye hastigheita til ei due kan ein dermed finne ved å leggje til bidraget fra kvar regel til den nåverande hastigheita til dua. Storleiken til hastigheitsvektoren skal ikkje overstige `MAX_SPEED`.

Implementer funksjonen `updateVelocity` i `Simulator.cpp`.

- Rekn ut den nye hastigheita til fuglen basert på separasjon, samanstilling, samhelde, og unngåelse.
- Dersom hastigheita som blir rekna ut overstig den maksimale hastigheita bestemd av `MAX_SPEED`, må den bli skalert ned på ein valfri måte til å vere under `MAX_SPEED`.

**Hint:** Koden inneheld ein hjelpefunksjon `magnitude` som kan vere nyttig for å rekne ut storleiken til ein hastigheitsvektor.

Når oppgåve T7 er ferdig skal simuleringa sjå ut som i Figur 5 etter at den har køyrt litt.




**Figur 5:** Skjermbilete av simuleringa etter at oppgåve T7 er implementert.

8. (20 points) **T8: Sjå kor du flyr**

Fram til no har fuglane vore orientert med spissen nedover i animasjonsvindauget uansett korleis retning dei har floge. Vi ønskjer no å teikne trekantane slik at dei reflekterer retninga til fuglane. Likning (1) gir retninga ein fugl flyr gitt $x$- og $y$-komponentane frå hastigheitsvektoren til fuglen.

$$\theta(x,y) = \begin{cases} \arctan\left(\frac{y}{x}\right), & \text{if } x > 0 \\ \arctan\left(\frac{y}{x}\right) + \pi, & \text{if } x < 0 \\ \frac{\pi}{2} \cdot \text{sign}(y), & \text{if } x = 0 \end{cases} \tag{1}$$

Oppdater funksjonen `draw` i `Simulator.cpp` slik at fuglane snur seg i den retningen dei flyr.

- Finn retningen duene flyr ($\theta$) ved å bruke Likning (1). Funksjonen `sign(y)` gir fortegnet til $y$. Du skal sjølv implementere funksjonaliteten til sign().
- Finn frontpunktet til trekanten ved å leggje til

  $$\cos(\theta) \cdot \text{size}$$

  til $x$-posisjonen til fuglen, og

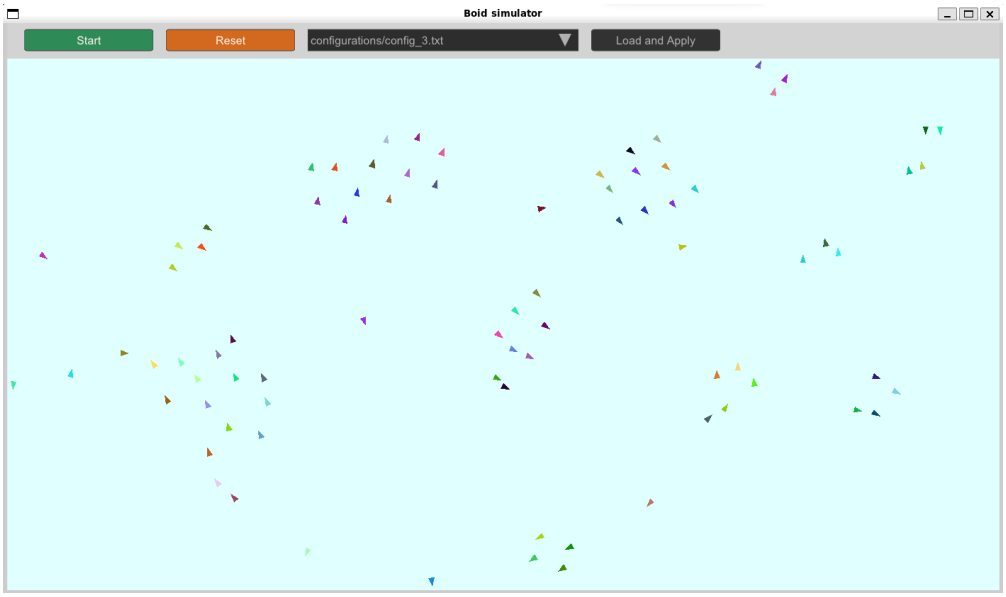
  $$\sin(\theta) \cdot \text{size}$$

  til $y$-posisjonen til fuglen. Hugs at `size` er eit attributt i `Bird`-klassen.
- Sidepunkta til trekanten finn ein med same framgangsmåte som for frontpunktet, men i staden for $\theta$ brukar ein $\theta + \frac{2}{3}\pi$ for det eine sidepunktet og $\theta - \frac{2}{3}\pi$ for det andre sidepunktet. $\pi$ er omtrent 3.1415926535.

**Hint:** Ein kan finne informasjon om korleis ein bruker matematiske funksjonar som er implementert i standardbiblioteket ved å gå til C++ reference og sida som heiter *Common math functions*. Noter deg at

dei trigonometriske funksjonane i standardbiblioteket opererer i radianar.

Når oppgåve T8 er ferdig skal simuleringa sjå ut som i Figur 6 etter at den har køyrt litt.




**Figur 6:** Skjermbilete av simuleringa etter at oppgåve T8 er implementert.

9. (25 points) **T9: FLOKKEN STILLER LIKT!**

   For å synleggjera kva for ein flokk ei due høyrer til ønskjer vi at fargen til alle duer i same flokk skal vere lik. Ein flokk består av alle duene som er lagra som vener i tillegg til alle duene dine vener har lagra som vener.

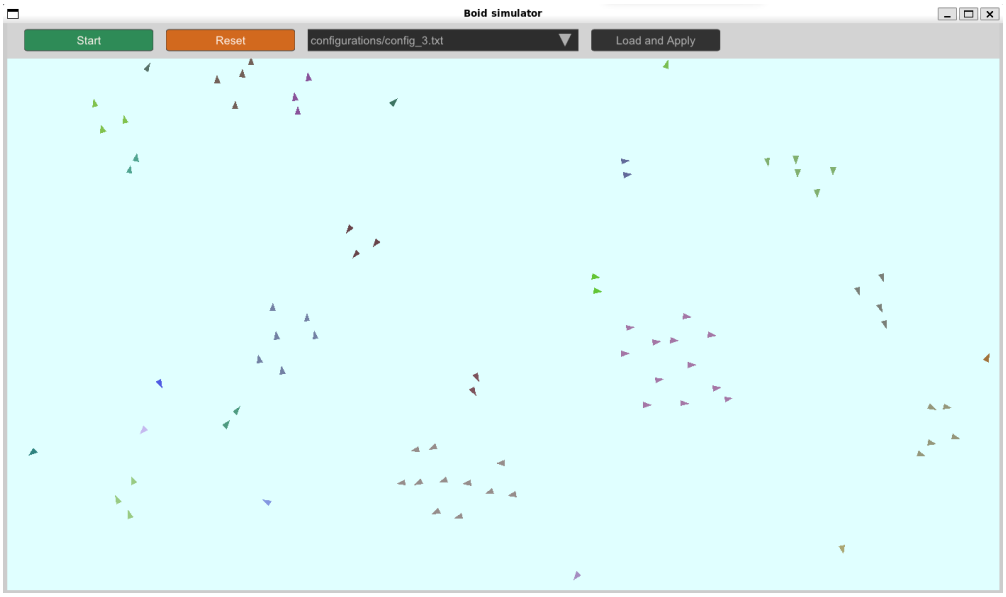
   I koden har kvar fugl to fargar; `originalColor` og `displayColor`. `originalColor` er fargen fuglen vart tildelt da den vart oppretta. Denne fargen kan ein ikkje endre. `displayColor` er fargen som fuglen viser på skjermen. Denne fargen kan ein endre med metoden `setColor` i `Dove`.

   Implementer funksjonen `colorBirds` i `Simulator.cpp`.

   - Alle duer som er i same flokk skal ha same farge.
   - Ulike flokkar skal helst ha forskjellig farge.

   **Merk:** `BEGIN`- og `END`-kommentarane står utanfor funksjonsdefinisjonen, noko som opnar for mogleg-heiten til å lage eigne hjelpefunksjonar eller globale variablar for å løyse oppgåven.

   Når oppgave T9 er implementert skal simuleringa sjå ut som i Figur 7 etter at den har køyrt litt.




**Figur 7:** Skjermbilete av simuleringa etter at oppgåve T9 er implementert.

10. (20 points) **T10: Throw more doves!**

    Vi ønskjer å kunne leggje til fleire duer mens simuleringa køyrer.

    Implementer funksjonen `addBird` i `Simulator.cpp`.

    - Legg til ei ny due viss knappen D på tastaturet blir heldt inne samtidig som ein klikkar på venstre museknapp i vindauget til simuleringa.
    - Startposisjonen til den nye dua skal vere gitt av koordinatane til museklikket.
    - Starthastigheita og startfargen til den nye dua er valfrie.

    Når oppgåve T10 er implementert skal ein kunne lage nye duer med museklikk dersom korrekt tast også blir heldt inne.

11. (25 points) **T11: Hauken kjem!**

    For å gjere simuleringa enda meir spennande ønskjer vi å introdusere haukar som duene skal prøve å unngå. I denne oppgåven skal du oppdatere `Hawk`-klassen i `Simulator.h`. Alle metodane og attributta som `Hawk`-klassen treng er allereie implementert. Det einaste som står igjen er å leggje dei til i klassedeklarasjonen.

Oppdater klassen `Hawk` slik at den inneheld relevante deklarasjonar for metodar og attributt.

- Fjern eller kommenter ut kodelinja `#define HAWK_IS_IMPLEMENTED`.
- Bruk tilbakemeldingene du nå får frå kompilatoren til å oppdatere klassedeklarasjonen til `Hawk`.
- Husk å tenk gjennom kva for synlegsnivå dei ulike attributta og metodane bør ha.

**Tips:** Du kan leggje til kodelinja `#define HAWK_IS_IMPLEMENTED` igjen dersom du ikkje har løyst oppgåven enda og ønskjer å fjerne feilane du får frå kompilatoren.

Når oppgåve T11 er gjort er simuleringa ferdigstilt, og du skal kunne sjå simuleringa som i Figur 2.