Del III:
Fuglesimulering (180p)

Del 3 av eksamen er programmeringsoppgaver. Denne delen inneholder **11 oppgaver** som til sammen gir en maksimal poengsum på ca. **180 poeng** og teller ca. **60 %** av eksamen. Hver deloppgave kan gi en poengsum fra **5 - 25 poeng** avhengig av vanskelighetsgrad og arbeidsmengde.

De utdelte filene inneholder kompilerbare (og kjørbare) `.cpp`- og `.h`-filer med kode og en full beskrivelse av oppgavene i del 3 som en PDF. Etter å ha lastet ned koden står du fritt til å bruke et utviklingsmiljø (VS Code) for å jobbe med oppgavene. Vi anbefaler på det sterkeste å kontrollere at koden kompilerer og kjører før du setter i gang med oppgavene.

Sjekkliste ved tekniske problem

Hvis prosjektet du oppretter med den utdelte koden ikke kompilerer (før du har gjort egne endringer) bør du rekke opp hånden og be om hjelp. Mens du venter kan du prøve følgende:

1. Sjekk at prosjektmappen er lagret i mappen `C:\temp`.
2. Sjekk at navnet på mappen **IKKE** inneholder norske bokstaver (*Æ, Ø, Å*) eller mellomrom. Dette kan skape problemer når du prøver å kjøre koden i VS Code.
3. Sørge for at du har:
   - Last ned `.zip`-filen fra Inspera og pakk den ut (unzip). Lagre filen i `C:\temp` som **IO2TDT4102_kandidatnummer**. Du skal altså skrive *ditt eget kandidatnummer* etter understreken.
   - Åpne mappen i VS Code. Bruk deretter følgende TDT4102-kommandoar for å opprette et fungerende 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 riktig fil i VS Code.
5. Kjør følgende TDT4102-kommandoer:
   (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 kjør koden igjen (`Ctrl+F5` eller `Fn+F5` eller `F5`).
7. Hvis det fortsatt ikke fungerer, lukk VS Code vinduet og åpne det igjen. Gjenta deretter steg 5-6.

Introduksjon

Fuglesimuleringen er en enkel simulering av hvordan fugler flyr i flokker. Koden for fuglesimuleringen er inndelt i tre hoveddeler:

- Et `Application`-objekt som styrer funksjonsbaren på toppen av animasjonsvinduet, inn-/ut-datahåndtering og hovedløkken som tegner hver frame til skjermen og ber et `Simulator`-objekt om å oppdatere fuglenes tilstand.
- Et `Simulator`-objekt som har oversikt over alle fuglene i simuleringen.

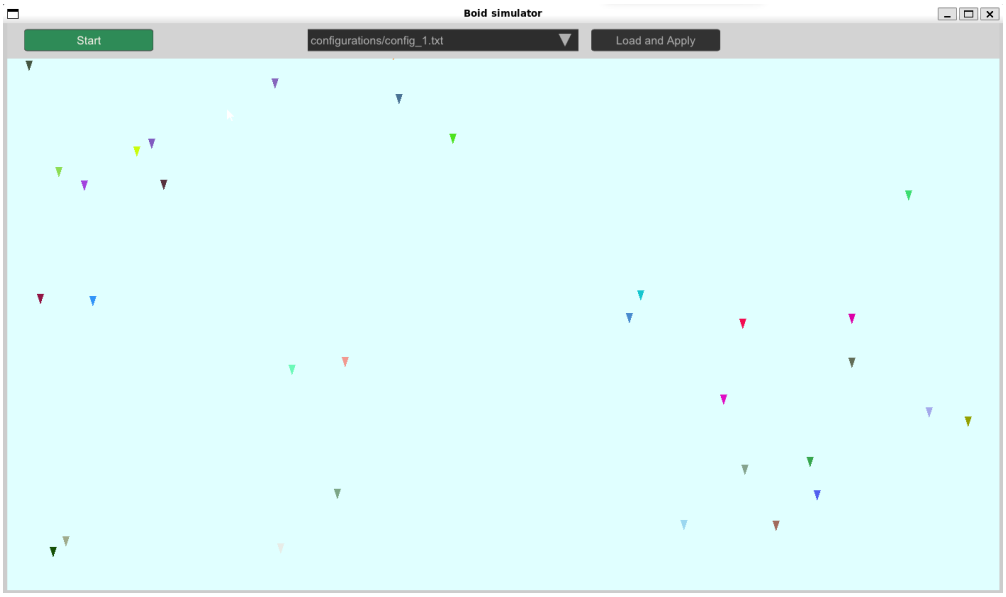
- En abstrakt baseklasse, `Bird`, som er ansvarlig for å oppdatere en fugls posisjon og hastighet, og å tegne fuglen til skjermen for hver frame.

Den abstrakte baseklassen `Bird` har to subklasser; `Doves` og `Hawks`. Duene følger reglene beskrevet under. Haukene flyr litt tilfeldig over skjermen og ønsker å unngå andre hauker. Du trenger ikke ta stilling til haukene før i siste oppgave.

Reglene som duene skal følge er basert på fire flokkegenskaper:

- **Separasjon (separation):** Fuglen styrer for å unngå kollisjon med andre fugler i flokken.
- **Sammenstilling (alignment):** Fuglen styrer i samme retning som resten av flokken.
- **Samhold (cohesion):** Fuglen styrer mot sentrum av flokken.
- **Unngåelse (avoidance):** Fuglen styrer for å unngå jegerfugler.

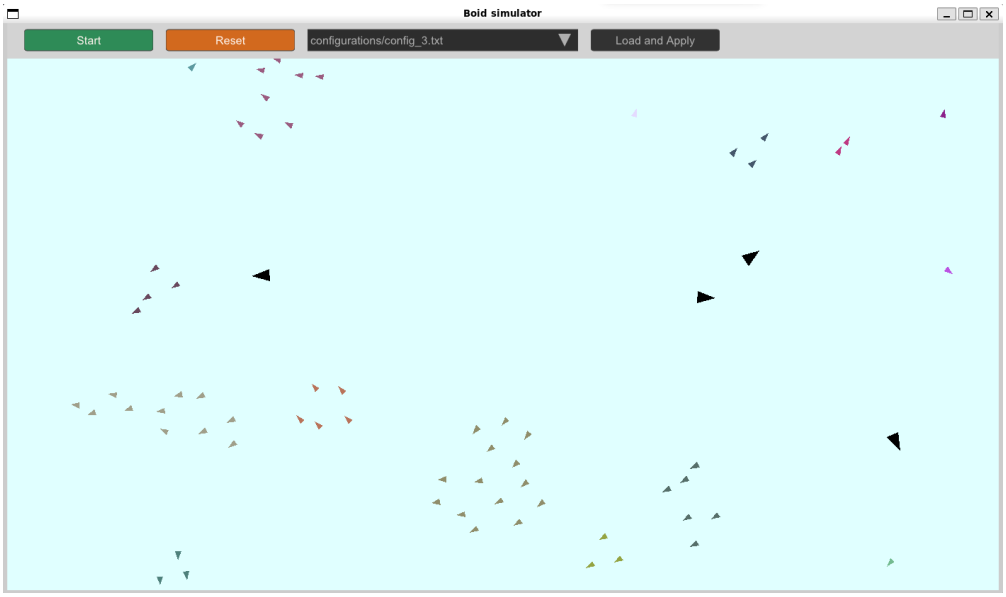
I `.zip`-filen finner du et kodeskjelett med klart markerte oppgaver.



Figur 1: Skjermbilde av kjøring av den utdelte koden. I animasjonsviduet ser man en funksjonsbar bestående av en Start/Stopp-knapp, en nedtrekksmeny for å laste inn ulike konfigurasjoner, og en knapp for å bruke den valgte konfigurasjonen (Load and Apply). I tillegg ser man 32 fugler i vinduet.

Før du begynner på oppgavene bør du sjekke at den utdelte koden kjører uten problemer. Du skal se omtrent det samme vinduet som i Figur 1. Fuglene plasseres tilfeldig i vinduet, så startposisjonen til de åtte fuglene vil variere og er ikke viktig.

Fuglesimuleringen i den utdelte koden inneholder minimalt med funksjonalitet (Figur 1). Når alle oppgavene er gjort vil man derimot kunne se at fuglene flyr i flokker mens de prøver å unngå jegerfuglene som flyr rundt i tilfeldige baner (Figur 2).



Figur 2: Skjermbilde av den kjørende simuleringen når den er ferdigstilt.

Hvordan besvare oppgavene

Hver oppgave i del 3 har en unik kode for å gjøre det lettere for deg å vite hvor du skal fylle inn svarene dine. Koden er på formatet `<T><siffer>` (*TS*), der sifrene er mellom 1 og 11 (*T1* - *T11*). For hver oppgave vil man finne to kommentarer som skal definere henholdsvis begynnelsen og slutten av svaret ditt. Kommentarene er på formatet: `// BEGIN: TS` og `// END: TS`.

For eksempel ser oppgave *T2* i den utdelte koden 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 svarene dine føres mellom slike par av kommentarer. Dette er for å støtte sensurmekanikken vår. Hvis det allerede er skrevet kode *mellom* `BEGIN`- og `END`-kommentarene til en oppgave i det utdelte kodeskjettet, så kan du, og ofte bør du, erstatte denne koden med din egen implementasjon. All kode som står *utenfor* `BEGIN`- og `END`-kommentarene **SKAL** du la stå som den er. I oppgave T6 og T9 er `BEGIN`- og `END`-kommentarene plassert utenfor funksjonsdeklarasjonen, noe som åpner for å bruke egne hjelpefunksjoner og globale variabler. Du skal ikke endre navn eller parameter til disse funksjonene.

Merk: Du skal **IKKE** fjerne `BEGIN`- og `END`-kommentarene.

Hvis du synes noen av oppgavene er uklare kan du oppgi hvordan du tolker dem og de antagelsene du gjør for å løse oppgaven som kommentarer i koden du leverer.

Tips: Trykker man `CTRL+SHIFT+F` og søker på `BEGIN:` får man snarveier til starten av alle oppgavene listet opp i utforskervinduet slik at man enkelt kan hoppe mellom oppgavene. For å komme tilbake til det vanlige utforskervinduet kan man trykke `CTRL+SHIFT+E`.

Hvordan levere del 3

Når du er ferdig med oppgavene og er klar til å levere skal du laste opp alle `.h`- og `.cpp`-filene i hovedmappen som en `.zip`-fil i Inspera. Du står fritt til å bruke den innebygde funksjonen i VS Code til å lage `.zip`-filen. Disse filene inkluderer:

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

Oppgavene

Oppgavene vil ha følgende struktur:

Første del inneholder motivasjon, bakgrunnsinformasjon og hvordan koden er satt sammen for oppgaven.

> Neste del er en tekstboks som inneholder oppgaven og spesifikke krav.

Siste del forklarer resultatet du kan forvente når du har fullført oppgaven.

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

   I den utdelte koden brukes en hastighetsvektor for å bestemme fuglens fart og retning, og en posisjon for å bestemme en fugls plassering. Begge er lagret som en struktur av typen `FloatingPoint` som inneholder to flyttall:

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

   Vi ønsker å kunne legge sammen to `FloatingPoint`-strukturer uten å måtte aksessere feltene i strukturene direkte hver gang. Addisjon av to vektorer utføres på følgende 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 oppgave T1 er ferdig skal det være mulig å legge sammen to strukturer av typen `FloatingPoint`.

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

   Nå står alle fuglene helt i ro når man trykker på start. Det er fordi funksjonen `updatePosition` i `Bird` ikke er implementert og fuglene blir tegnet på samme posisjon i hver frame.

   > Implementer funksjonen `updatePosition` i `Simulator.cpp`.
   >
   > - Oppdater posisjonen (position) til fuglen ved å legge sammen fuglens hastighet (velocity) og posisjon (position).

   Når oppgave T2 er ferdig skal duene fly i rette linjer over skjermen hvis man klikker på `Start` og stå i ro hvis man deretter klikker på `Stop`.

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

   Nå som fuglene flytter på seg mellom start og stopp av simuleringen ønsker vi å kunne tilbakestille simuleringen så man kan kjøre simuleringen på nytt uten å måtte restarte hele programmet. `Simulator` har en klasse `ResetButton` som er definert i `Simulator.h`:

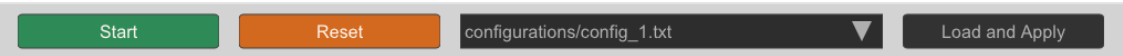
```
class ResetButton : public Button {
public:
    ResetButton ();
    ~ResetButton ();

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

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

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

   Når oppgave T3 er ferdig skal funksjonsbaren til simuleringen se ut som i Figur 3.



Figur 3: Skjermbilde av funksjonsbaren etter at oppgave T3 er fullført.

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

   Tilbakestillingsknappen du la til i forrige oppgave gjør foreløpig ingenting når man klikker på den. Det er fordi callback-funksjonen ikke er implementert enda.

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

Når oppgave T4 er ferdig skal tilbakestillingsknappen sette fuglene i startposisjon når den blir klikket på. Noter deg at knappen kun fungerer når simuleringen er stoppet og at fuglene vil plasseres ulikt etter hver tilbakestilling.

5. (20 points) **T5: Last inn konfigurasjoner T5: Last inn konfigurasjonar**

   Vi ønsker å kunne teste flere sammensetninger av fugler. For å gjøre dette har vi et sett med konfigurasjoner i form av `.txt`-filer. Disse filene finner du i mappen som heter `configurations`. Hver fil består av to tall som er separert med et linjeskift. Det første tallet representerer antall duer i simuleringen, mens det andre tallet representerer antall hauker i simuleringen.

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

   - Utløs et passende unntak dersom filen ikke kan åpnes.
   - Les inn tallene fra den gitte konfigurasjonsfilen.
   - Sørg for at konfigurasjonen som leses inn blir brukt i simuleringen. Simulator-objekter har en funksjon som heter `applyConfiguration()`.

   Når oppgave T5 er ferdig skal du kunne se en endring av antall fugler på skjermen når du trykker på knappen `Load and apply` etter at du har valgt en konfigurasjon fra nedtrekksmenyen.

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

   For at duene skal kunne vite hvordan de skal forflytte seg i neste frame ønsker de en oversikt over fuglene rundt seg. Det er to typer fugler i simuleringen; duer (doves) og hauker (hawks). Duene ser på de andre duene som potensielle venner og hauker som potensielle fiender. I tillegg er duene sitt synsfelt begrenset av de globale variablene `FRIEND_RADIUS` for venner, og `AVOID_RADIUS` for fiender. Alle duer som er utenfor `FRIEND_RADIUS` og alle hauker som er utenfor `AVOID_RADIUS` skal derfor ignoreres. Dette er illustrert i Figur 4. Venner og fiender lagres i hver sin tabell (`vector`) av delte pekere (`shared_ptr`) til fugleobjekt (`Bird`). Funksjonen du skal implementere skal kalles på hvert `Bird`-objekt for hver frame.

   Avstanden mellom to punkt regnes ut på følgende måte:

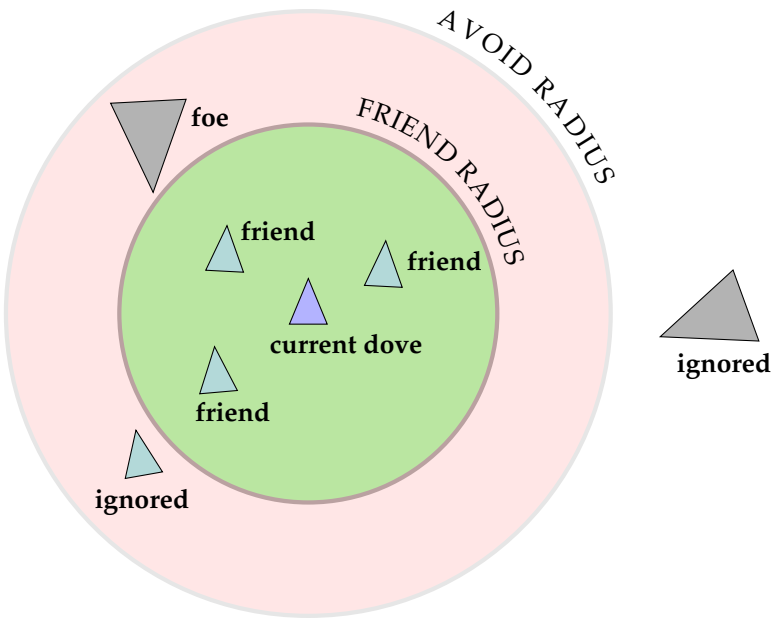
$$\text{distance}((x_1, y_1), (x_2, y_2)) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

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

   - Fjern elementene som allerede ligger i vektorene `friends` og `foes` fra forrige frame.
   - Filtrer de andre fuglene i simuleringen basert på type. Merk at listen funksjonen tar inn inneholder alle fugler. Du må altså hoppe over din egen Bird-instans.
   - Pass på at venner er innenfor `FRIEND_RADIUS` og at fiender er innenfor `AVOID_RADIUS`.

   Hint: Man kan finne informasjon om hvordan man bruker matematiske funksjoner som er implementert i standardbiblioteket ved å gå til C++ reference og siden som heter *Common math functions*.

   Merk: `BEGIN`- og `END`-kommentarene er utenfor funksjonsdefinisjonen, noe som åpner for muligheten til å lage egne hjelpefunksjoner eller globale variabler for å løse oppgaven.



Figur 4: Illustrasjon av hvilke fugler som er venner og fiender for en spesifikk due (lilla trekant). De andre duene er illustrert med blå trekanter, mens haukene er illustrert med grå trekanter.

Resultatet av oppgave T6 kan man ikke verifisere visuelt.

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

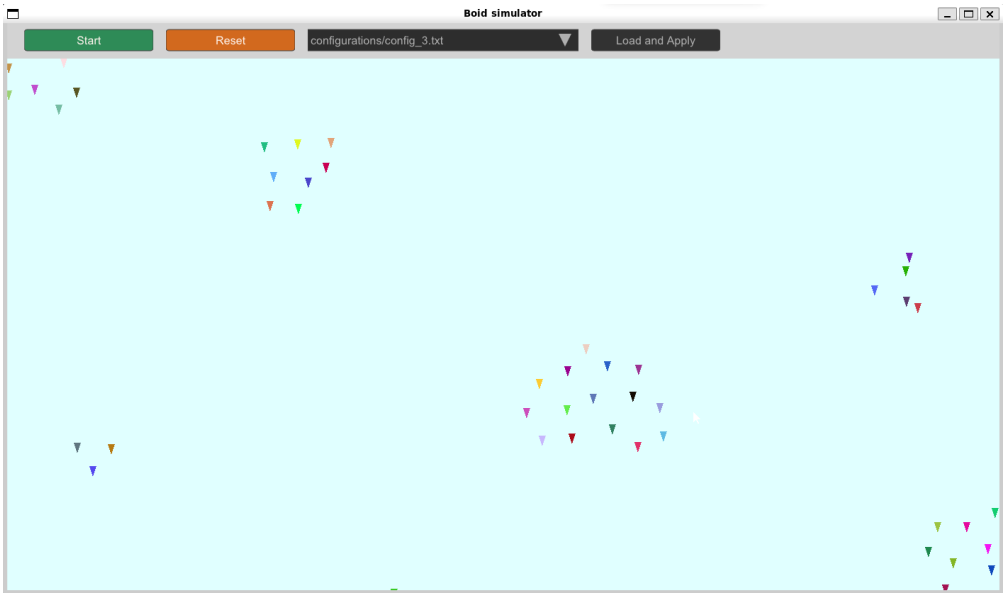
Vi ønsker nå å få duene til å følge reglene vi introduserte i introduksjonen til koden. Dette gjøres ved å oppdatere hastigheten til duene. Reglene er allerede implementert i funksjonene `calculateCoherence`, `calculateAlignment`, `calculateSeparation` og `calculateAvoidance`. Disse funksjonene regner ut hvordan hastigheten til en due må forandre seg for at den skal overholde reglene. Den nye hastigheten til en due kan man dermed finne ved å legge til bidraget fra hver regel til duens nåværende hastighet. Størrelsen til hastighetsvektoren skal ikke overstige `MAX_SPEED`.

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

- Regn ut den nye hastigheten til fuglen basert på separasjon, sammenstilling, samhold, og unngåelse.
- Dersom den utregnede hastigheten overstiger den maksimale hastigheten bestemt av `MAX_SPEED`, må den skaleres ned på en valgfri måte til å være under `MAX_SPEED`.

Hint: Koden inneholder en hjelpefunksjon `magnitude` som kan være nyttig til å regne ut størrelsen til en hastighetsvektor.

Når oppgave T7 er ferdig skal simuleringen se ut som i Figur 5 etter at den har kjørt litt.



Figur 5: Skjermbilde av simuleringen etter at oppgave T7 er implementert.

8. (20 points) **T8: Se hvor du flyr**

Frem til nå har fuglene vært orientert med spissen nedover i animasjonsvinduet uansett hvilken retning de har flydd. Vi ønsker nå å tegne trekantene slik at de reflekterer fuglenes retning. Likning (1) gir retningen en fugl flyr gitt x- og y-komponentene fra fuglens hastighetsvektor.

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

Oppdater funksjonen `draw` i `Simulator.cpp` slik at fuglene snur seg i den retningen de flyr.

- Finn retningen duene flyr (θ) ved å bruke Likning (1). Funksjonen `sign(y)` gir fortegnet til y. Du skal selv implementere funksjonaliteten til sign().
- Finn frontpunktet til trekanten ved å legge til

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

  til fuglens x-posisjon, og

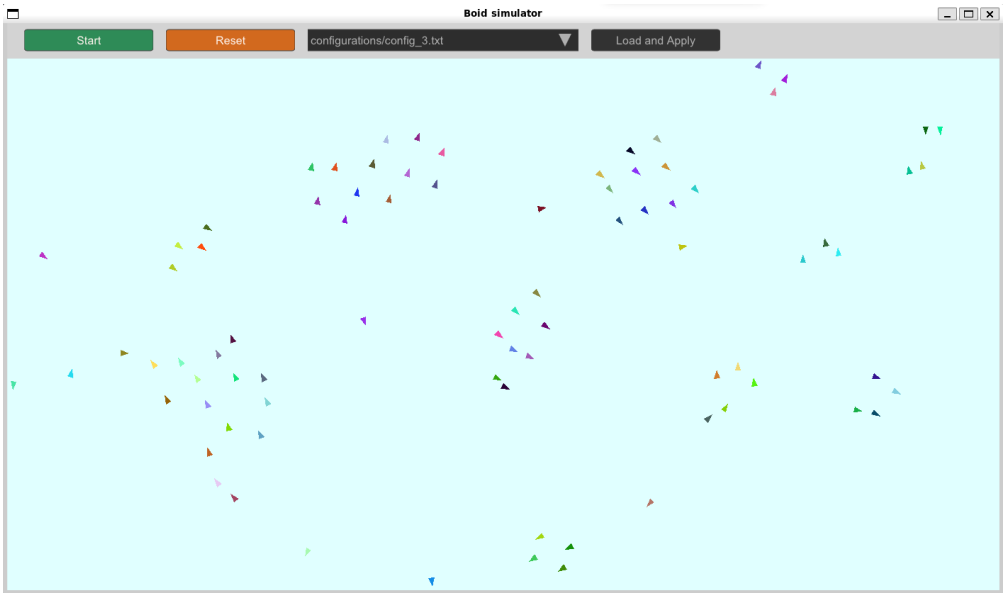
  $$\sin(\theta) \cdot \text{size}$$

  til fuglens y-posisjon. Husk at `size` er et attributt i `Bird`-klassen.
- Sidepunktene til trekanten finner man med samme fremgangsmåte som for frontpunktet, men i stedet for θ bruker man $\theta + \frac{2}{3}\pi$ for det ene sidepunktet og $\theta - \frac{2}{3}\pi$ for det andre sidepunktet. π er omtrent 3.1415926535.

Hint: Man kan finne informasjon om informasjon om bruken av matematiske funksjoner som er implementert i standardbiblioteket ved å gå til C++ reference og siden som heter *Common math functions*.

Noter deg at de trigonometriske funksjonene i standardbiblioteket opererer i radianer.

Når oppgave T8 er ferdig skal simuleringen se ut som i Figur 6 etter at den har kjørt litt.



Figur 6: Skjermbilde av simuleringen etter at oppgave T8 er implementert.

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

   For å synliggjøre hvilken flokk en due tilhører ønsker vi at fargen til alle duer i samme flokk skal være lik. En flokk består av alle duene som er lagret som venner i tillegg til alle duene dine venner har lagret som venner.

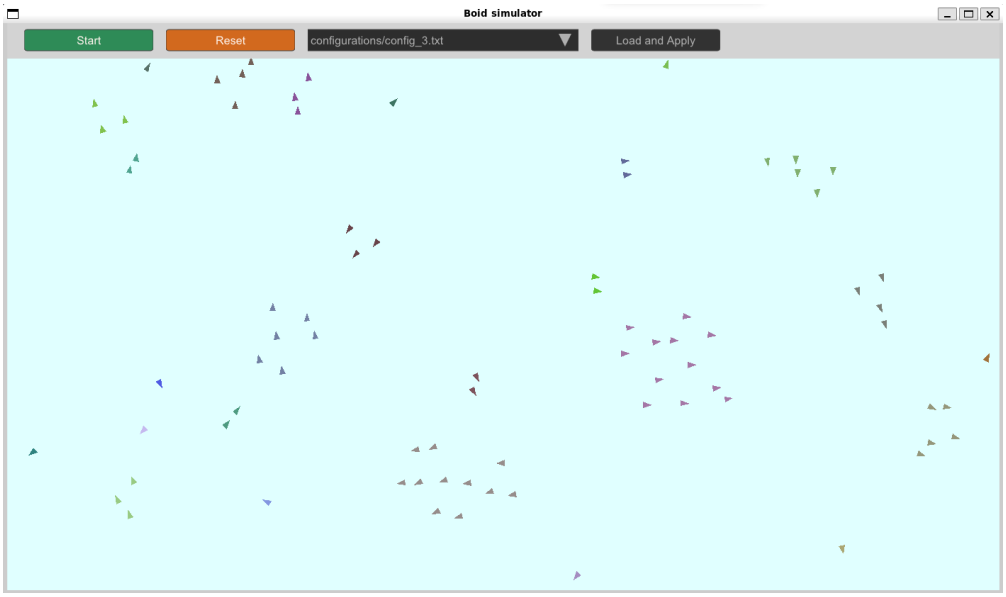
   I koden har hver fugl to farger; `originalColor` og `displayColor`. `originalColor` er fargen fuglen ble tildelt da den ble opprettet. Denne fargen kan man ikke endre. `displayColor` er fargen som fuglen viser på skjermen. Denn fargen kan man endre med metoden `setColor` i `Dove`.

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

   - Alle duer som er i samme flokk skal ha samme farge.
   - Ulike flokker skal helst ha forskjellig farge.

   Merk: `BEGIN`- og `END`-kommentarene er utenfor funksjonsdefinisjonen, noe som åpner for muligheten til å lage egne hjelpefunksjoner eller globale variabler for å løse oppgaven.

   Når oppgave T9 er implementert skal simuleringen se ut som i Figur 7 etter at den har kjørt litt.



Figur 7: Skjermbilde av simuleringen etter at oppgave T9 er implementert.

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

    Vi ønsker å kunne legge til flere duer mens simuleringen kjører.

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

    - Legg til en ny due hvis knappen D på tastaturet holdes inne samtidig som man klikker på venstre museknapp i simuleringsvinduet.
    - Startposisjonen til den nye duen skal være gitt av koordinatene til museklikket.
    - Starthastigheten og startfargen til den nye duen er valgfrie.

    Når oppgave T10 er implementert skal man kunne lage nye duer med museklikk dersom korrekt tast også holdes inne.

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

    For å gjøre simuleringen enda mer spennende ønsker vi å introdusere hauker som duene skal prøve å unngå. I denne oppgaven skal du oppdatere `Hawk`-klassen i `Simulator.h`. Alle metodene og attributtene som `Hawk`-klassen trenger er allerede implementert. Det eneste som gjenstår er å legge dem til i klassedeklarasjonen.

Oppdater klassen `Hawk` slik at den inneholder relevante deklarasjoner for metoder og attributter.

- Fjern eller kommenter ut kodelinjen `#define HAWK_IS_IMPLEMENTED`.
- Bruk tilbakemeldingene du nå får fra kompilatoren til å oppdatere klassedeklarasjonen til `Hawk`.
- Husk å tenk gjennom hvilket synlighetsnivå de ulike attributtene og metodene bør ha.

Tips: Du kan legge til kodelinjen `#define HAWK_IS_IMPLEMENTED` igjen dersom du ikke har løst oppgaven enda og ønsker å fjerne feilene du får fra kompilatoren.

Når oppgave T11 er gjort er simuleringen ferdigstilt, og du skal kunne se simuleringen som i Figur 2.