Norges teknisk–naturvitenskapelige universitet
Institutt for datateknologi og informatikk

# TDT4102 Prosedyre- og objektorientert programmering Vår 2020

**Øving 12**

## Frist: 2020-04-03

## Mål for denne øvinga:

- Grafikk med FLTK 1.4
- Nytte det vi har lært i emnet til å lage eit enkelt spel/simulering
- Repetisjon av essensielle delar av pensum

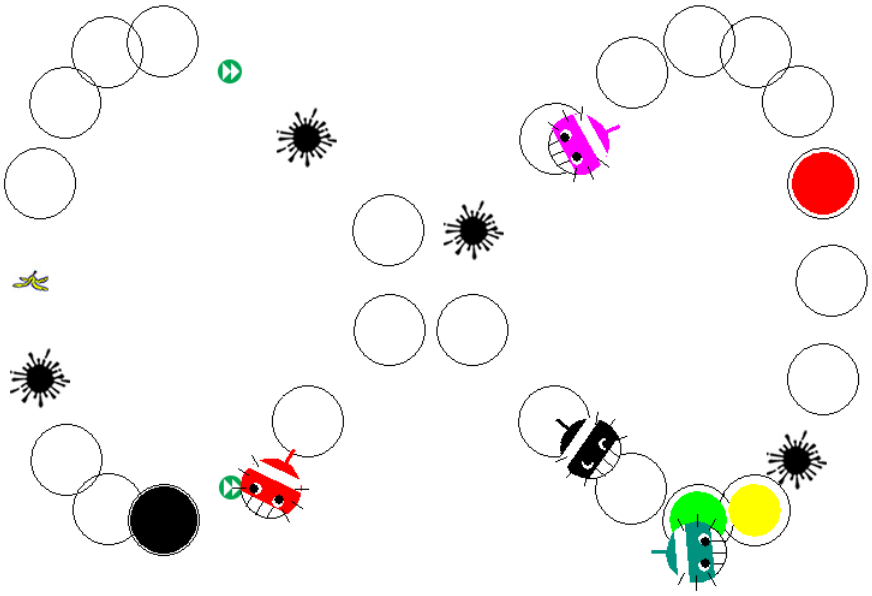
## Generelle krav:

- Bruk dei eksakte namn og spesifikasjonar gjeve i oppgåva.
- 70% av øvinga må godkjennast for at den skal vurderast som bestått.
- Øvinga skal godkjennast av stud.ass. på sal.
- Det anbefalast å nytte ein programmeringsomgjevnad(IDE) slik som Visual Studio Code.

## Tilrådd lesestoff:

- FLTK-dokumentasjonen

I denne øvinga er målet å lage eit fullstendig program for enkel simulering av 2D-kappløp. Ein skal kunne både styre eit køyretøy sjølv og lage ein algoritme for sjølvstyring. Der skal også vere ei rekkje hindringar som påverkar løpet.



I dei tidlegare øvingene har vi gjort oss godt kjende med `Graph_lib` biblioteket. Dette byggjer eigentleg på det underliggjande FLTK-biblioteket, men har enklare grensesnitt og organisering av filer. Det å kunne utforske og setje seg inn i nye bibliotek og kodebasar er ein veldig viktig eigenskap når ein kodar i praksis, så i samband med dette skal denne øvinga gå ut på å lage programmet i FLTK. FLTK-dokumentasjonen finn du her, og vi skal gå gjennom nokre spesifikke delar i løpet av øvinga.

**NB!** For denne øvinga må du lage eit nytt prosjekt og leggje inn den utdelte koda. Innhaldet skal forklarast etter kvart som det vert relevant.

## 1 Første penselstrøk(15%)

Komponentar i FLTK er i hovudsak subklasser av klassa `Fl_Widget`. Denne klassa har ein `void draw()`-funksjon der all teikning skal foregå, og som vert kalla ein gong kvar iterasjon.

**a) Lag ei ny klasse, `Vehicle`, som arvar frå `Fl_Widget`.**
`Fl_Widget` inkluderer du frå `<FL/Fl_Widget.H>`. I konstruktøren til `Vehicle` treng du berre kalle konstruktøren til `Fl_Widget` i initialiseringslista. Dei eksakte argumenta er ikkje viktige for oss, så du kan nytte verdien 10 på alle fire.

**b) Implementer den abstrakte funksjonen `void draw()` med `final` og gi den noko innhald.** Ein nyttar `final` for å indikera at ein virtuell funksjon ikkje kan redefinerast i ein subklasse. Sidan `draw()` er ein virtuell funksjon i Widget er den også virtuell i `Vehicle`. I `draw` skal du kalle ulike teiknefunksjonar frå `<FL/fl_draw.H>`. Forklaring kan du finne her. Du kan gjere teikninga så enkel eller kompleks som du vil, men sørg for å teikne innanfor vindauget. Før du teiknar må du også først setje ein farge å teikne med vha. funksjonen `fl_color`, sjå dokumentasjon. Du teikne kva du vil, til dømes ein sirkel.

**c) Instansier eit `Vehicle`-objekt i `main.cpp` og sjå om noko dukkar opp.** Instansieringa av objektet skal skje i det markerte området i fila. Konstruktøren til `Fl_Widget` koplar automatisk objektet til vår `Fl_Double_Window`.
Sjekk no at noko vert teikna i vindauget når du køyrer programmet.

# 2 Implementasjon av køyring (30%)

**a) Gjer `Vehicle` til ei abstrakt klasse, og lag ei konkret klasse som arvar frå den, `PlayerVehicle`.**

Sidan vi har lyst å ha fleire ulike køyretøy med ulik utsjånad og styring, er det lurt å ha funksjonalitet for å bytte dette ut. Ein av dei mest idiomatiske måtene å gjere dette på er ved å lage subklasser og nytte polymorfisme, og ved å då gjere `Vehicle` om til ei abstrakt klasse så er dette tydeleg illustrert. Måten det skal fungere på er at `Vehicle` skal ha all kode som er felles for alle køyretøy, medan subklassene implementerer eigen styring og teikning.

- I `Vehicle`, lag to `protected` pure virtual medlemsfunksjonar: `std::pair<double,double> steer() const`, som skal styre køyretøyet, og `void drawBody() const`, som teiknar køyretøyet. Du må her inkludere `<tuple>`.
- Lag så ei konkret klasse, `PlayerVehicle`. Denne skal ha ein konstruktør som tek inn parameter og gir dei vidare til `Vehicle` sin konstruktør i initialiseringslista. Implementer så `PlayerVehicle::drawBody()` der du limar inn koda som teiknar køyretøyet frå `Vehicle::draw()`.
- I `void Vehicle::draw()` må du endre funksjonen slik den først kallar `steer()` og lagrar returverdien i ein lokal variabel, og til slutt kallar `drawbody()`.

**b) Gi Vehicle ein `PhysicsState` som `protected` medlemsvariabel.**

Køyretøya våre vil ha ein del fysiske størrelser som posisjon, retning og fart samt funksjonar for å oppdatere dei. Det er derfor gunstig å samle desse i ei klasse, `PhysicsState`. Denne kan du finne i den utdelte fila `utilities.h`. Variabelen skal vere `protected` slik subklassene kan nytte den i `steer` og `draw`.

Vi representerer farten med ein vinkel- og absoluttfart sidan dette er meir naturleg for eit køyretøy. Vinkelen skal vere i radianar. `grip` er for veggrepet til køyretøyet og skal normalt vere 1. Initialiser `x, y,` og `angle` med parameter i konstruktøren til `Vehicle`.

**c) Implementer `PlayerVehicle::steer()` ved å leggje inn tastaturstyring, og så sørg for at teikninga er avhengig av `Vehicle` sin `PhysicsState` slik du kan flytte på din `Vehicle`.**

Funksjonen `Fl::event_key`, som ligg i `<FL/Fl.H>`, sjekkar om ein tast er nedtrykt. Denne tek som argument anten ein stor bokstav (`char`) som tilsvarar tasten, eller ein enum for andre tastar; ei liste over desse er her under "`Fl::event_key` values". I `steer` returnerer du ulike tal avhengig av kva tastar som er nedtrykte.

I `Vehicle::draw()` kan du endre direkte på `x`- og `y`-posisjonen til objektet sin `PhysicsState` ved hjelp av returverdiane til `steer`. Oppdater teikninga di i `PlayerVehicle::drawBody` slik den er avhengig av `x`- og `y`-posisjonen til `PhysicsState`. Lag så ein instans av `PlayerVehicle` i `main`, og sjå at det er mogeleg å flytte teikninga med tastetrykk.

**d) Implementer ein meir realistisk fysisk modell i `Vehicle::draw()`.**

For å simulere eit køyretøy overbevisande trengjer vi ein meir realistisk modell for oppdatering av posisjon og fart. Ei betre tilnærming er å bruke akselerasjon i både fart og retning ved hjelp av to `double` som vi vil skal liggje i intervallet `[-1,1]`, og dei representerer kor kraftig fartsendringa og svinginga er og i kva retning. Det er dette funksjonen `steer` skal returnere. For å oppdatere køyretøyet med desse, så sender du dei inn i `PhysicsState` sin medlemsfunksjon `update`.

Vi vil også at køyretøyet skal halde seg innanfor skjermen til all tid. Det kan du gjere ved å setje `x` og `y` posisjonane til kantane av skjermen når dei går utanfor. Grensene kan du finne i `utilities.h`.

**e) Fullfør `drawBody`-funksjonen til `PlayerVehicle`.**

No skal det også vere mogeleg å fullstendiggjere funksjonen for teikning av køyretøyet ditt. Noko av teikninga skal vere avhengig av `PhysicsState` sin `angle` slik at retninga til

køyretøyet vert synleg. F.eks kan du teikne eit lite objekt, som ein sirkel, i den retninga du køyrer. Teikninga skal også vere på størrelse med ein sirkel med radius `vehicleRadius` (som du gjerne kan endre på) frå `utilities.h`, så gjerne nytt denne variabelen gjennom teikninga.

Du kan nytte veldig mange teiknefunksjonar, og dermed gjere teikninga så kompleks (eller enkel) som du vil! Eit tips er at dersom du vil lage objekt med fyll, så kallar du teiknefunksjonar mellom `fl_begin_polygon()` og `fl_end_polygon()`. Då må du også spesifisere når du vil teikne linjer med å kalle dei tilsvarande funksjonane med `line` i staden for `polygon`.

**f) Frivillig:** Lag ei ny køyretøy-klasse på same måte som `PlayerVehicle` og endr `steer` slik den brukar andre tastar til styring. Dermed vil du kunne kontrollere to køyretøy samtidig.

# 3 Implementasjon av bane (30%)

**a) Lag ei ny klasse, `Track`, som arvar frå `Fl_Widget`. Denne skal ha eit privat medlem `std::vector<std::pair<double,double> > goals`.**

Dette skal vere ei bane som fungerar med at køyretøyet må vere innanfor ein viss radius på kvart punkt for å gå vidare. Vektoren inneheld punkta som representerer bana bilen skal køyre. Hugs at du må inkludere `<tuple>` her for å få tilgong til `std::pair`.

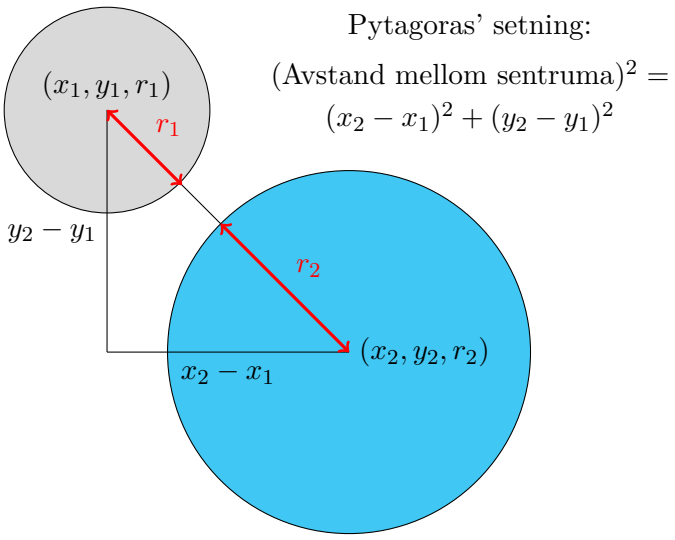
I konstruktøren til `Track` kan du også sette opp ein vilkårleg bane. Hugs at konstruktøren til `Fl_Widget` må kallast her også, der du kan nytte verdien 10 på dei fire argumenta.

**b) Gjer `Track` til ei konkret klasse ved å implementere funksjonen `void draw() override`.**

Denne skal sørgje for at bana teiknast opp. Teikn gjerne punkta som tomme sirklar slik som er gjort på biletet i starten av øvinga.

**c) Lag ein fri funksjon `bool circleCollision(double delX, double delY, double sumR)` i `Vehicle.h` som avgjer om to sirklar overlappar.**

For enkelheitas skuld så velgjer vi å representere alle objekt fysisk som sirklar, så derfor er det nødvendig med ein funksjon som avgjer om to sirklar har kollidert. Dette er ein ganske enkel ting å sjekke for sirklar, og det går ut på å sjå om avstanden mellom sentruma til sirklane er mindre enn summen av radiusane. Parametrane til funksjonen er respektivt skilnad i x-koordinatar, skilnad i y-koordinatar, og summen av radiusane. Sjå komande figur for eit geometrisk oppsett.




PS! Det å finne kvadratrota er ofte ein relativt dyr kalkulasjon, så program som skal køyre intensivt, som her, samanliknar heller kvadrata av avstandane.

**d) Endr `Vehicle`-klassa slik at den lagrar ein `protected` konstant referanse til ein `Track`. Denne må `Vehicle` og alle subklasser ta inn i konstruktøren sin.**

Du må også lage `getGoals() const` i `Track` som gir ein konstant referanse til `goals`-vektoren. Vi vil gjere all behandling av bana innanfor `Vehicle`, noko som gjer at klassa må ha tilgang til vektoren som representerer bana. Ein måte å gjere dette hadde vore å lage eit globalt tilgangspunkt til den, men hyppig bruk av dette kan ofte gjere eit prosjekt uoversiktleg, spesielt i større samanhengar. Derfor skal vi bruke ein annan teknikk som heiter *dependency injection* der koplingar i programmet vert veldig tydelege. Det er dette vi gjer når vi let `Vehicle` eksplisitt be om ein `Track` i konstruktøren sin.

**e) Gi `Vehicle` eit `protected` heiltal som representerer indeksen til det gjeldande målet, og skriv kode i `draw` som sjekker om køyretøyet har treft målet sitt.**

Bruk `circleCollision` for å sjekke om målet er nådd, og i det tilfellet må du setje kva neste mål skal vere. Hugs å starte bana på nytt når det siste punktet er treft.

For å teste kollidering må du ha radiusen for køyretøyet og målet, og desse er gjevne i `utilities.h`. Det er også lurt å teikne det gjeldande målet i `PlayerVehicle::drawBody()` for å få betre oversikt. Sjekk nå at det gjeldande målet endrar seg når du treff det.

**f) Lag ein ny `enum class` i `Track.h`, `Obstacle`.**

Dette skal vere dei mogelege hindringene på bana og skal ha følgjande verdiar: `Spill`, `Boost`, `Peel` og `None`. `Spill` er eit oljesøl som reduserer veggrepet, `Boost` forbetrar køyreeigenskapane, og `Peel` er eit bananskal som får køyretøyet til å skli.

**g) Gi `Track` ein `std::vector<std::tuple<double,double,Obstacle>>`, og lag funksjonen `getObstacles() const` som returnerer ein konstant referanse til den.**

Kvar `tuple` i denne vektoren representerer ei hindring på bana. Sett opp desse hindringene i konstruktøren til `Track`, og teikn dei i `draw` funksjonen til `Track` som sirklar med kvar sin farge. Til dette skal du nytte dei respektive radiusene i `utilities.h`: `spillRadius`, `boostRadius`, og `peelRadius`. Alternativt kan du sjå i appendiks A dersom du vil nytte bilete i staden.

I `Vehicle` må du sjekke om køyretøyet kolliderer med ein av desse, og det er ryddig å også gjere dette i ein eigen medlemsfunksjon. Dersom den treff ei hindring, er det ofte nødvendig å gi den ein tilstand som markerar at den har kollidert med ei slik hindring. Til dette kan du lage ein `protected` `Obstacle`-medlemsvariabel `status` i `Vehicle`. Bruk `Obstacles::None` når den ikkje har treft noko.

**h)** No må vi definere oppførsel for køyretøyet når den har treft eit av hindringane. Du står fritt til å velje korleis denne koden skal vere implementert, men koda skal ligje i `Vehicle` og desse krava skal følgjast:

- `Spill`: Eit oljesøl vil så klart redusere veggrepet ditt. Sett derfor `grip` til 0.5. Effekten vil automatisk forsvinne gradvis i `PhysicsState::update`. Det bør ikkje vere nødvending å endre `status` her.
- `Boost`: Måten vi skal representere dette på, er å auke `grip` til 2. Her er det heller ikkje nødvendig å endre `status`.
- `Peel`: Viss du har treft eit bananskal og `ps.speed` er større enn 2 når du treff bananskalet, skal du nytte funksjonen `PhysicsState::slide` i staden for `PhysicsState::update`. Argumentet til funksjonen skal vere vinkelen frå då du traff bananen, så dette må du lagre. Dette skal fortsetje til `speed < 0.3`. Her skal du markere `status` som `Obstacle::Peel` til du er ferdig å skli.

# 4 Køyring/teikning algoritmar (25%)

Dette vil vere ei ganske åpen oppgåve der du skal sjølv lage ein algoritme for eit sjølvkøyrande køyretøy. Dette skal du gjere i `steer` i ein eigen subklasse av `Vehicle`. Lag også ein unik utsjånad i `drawBody()`.

Målet med algoritmen er at den skal vere så rask og smart som mogeleg. Her følgjer derfor nokre tips som kan hjelpe deg på veg:

- For å finne vinkelen mellom to (matematiske) vektorar, kan du nytte funksjonen `angleBetween` i `utilities.h`. Denne returnerer vinkelen frå den første til den andre vektoren i radianar, men det mest nyttige er kanskje forteiknet der positivt er definert mot klokka. (Men hugs at y-aksa i programmet peiker nedover.)
- Dette programmet har ingen skilnad på om du køyrer forlengs eller baklengs.
- Du kan sjå på så mange mål framover som du vil, så det kan vere lurt å tilpasse seg litt for målet etter det gjeldande.
- Dersom du gjer ei lita justering og lurar på om den er betre, eller du har laga fleire algoritmer og lurar på kva slags som er best, kan du more deg med å køyre dei mot kvarandre.
- Hugs at du kan leggje inn medlemsvariablar i subklassa di. Dersom du nyttar nøkkelordet `mutable` kan du også endre på dei i `steer` og `drawBody()`. Sjå om du kan finne eit bruksområde for dette.

## Appendiks A: Bilete i FLTK

FLTK har to måtar å teikne bilete på: direkte frå ein buffer, eller via ein `Fl_Image` klasse. Den sistnemnde er den enklaste måten og den vi skal gå gjennom her. `Fl_Image` har ein del subklasser som handterar ulike filtyper og er namngjeve slik: `Fl_PNG_Image`, `Fl_JPEG_Image`, `Fl_BMP_Image` etc. Alle desse ligg i ein headerfil med same namn.

Konstruktøren til kvar av desse klassene tek inn eit filnamn. Du må teikne biletet i `draw()` funksjonen til ein `Fl_Widget`, og dette gjer du med å kalle `Fl_Image::draw()` frå `Fl_Image` objektet med parametrane:

```
void draw(int x, int y, int w, int h, int ox = 0, int oy = 0);
```

`(x,y,w,h)` utgjer rektangelet i vindauget der biletet skal plasserast der `(0, 0)` er øverst til venstre. Den vil teikne delen av biletet gjeve av rektangelet `(ox, oy, w, h)`. Både `ox` og `oy` har defaultverdien 0. Der er også ein overload av funksjonen som tek inn kun `x` og `y` og plasserer heile biletet på denne posisjonen. Der er nokre tydelege begrensningar med dette, som at du ikkje kan rotere eit bilete og at skalering er uintuitivt. Bileta du nyttar bør derfor vere i korrekt størrelse i utgangspunktet.

Eit problem med `Graph_lib` prosjektmalen er at den ikkje har alle dei nødvendige filene for å bruke PNG-bilete, så vi får nøye oss med JPEG og GIF. Ein bakdel med dette er at vi ikkje kan bruke gjennomsiktigheit, noko som vil føre til at alle bilete må vere rektangulære. For å korrigere dette nokolunde, så kan vi sørgje for at bakgrunnsfargen på bileta matchar det i vindauget og teikne bileta først. Bruk medlemsfunksjonen `color` til vindauget for å endre bakgrunnsfargen, og dersom du nyttar dei utdelte bileta bør fargen vere `FL_WHITE`.

Bileta du kan bruke for hindringar finn du i utdelt filar. Vi tillet dei å vere litt større enn sirklane vi nyttar for kollisjonar: *peelSprite.jpeg* er 30x30 pikslar, *spillSprite.jpeg* er 50x50, og *boostSprite.jpeg* er 20x20. Hugs at `x,y`-posisjonen skal vere på midten!