Norges teknisk–naturvitenskapelige universitet
Institutt for datateknologi og informatikk

TDT4102 Prosedyre- og objektorientert programmering Vår 2019

Øving 12

Frist: 2019-04-09

Mål for denne øvinga:

- Grafikk med FLTK 1.3
- Nytte det vi har lært i emnet til å lage eit enkelt spel/simulering

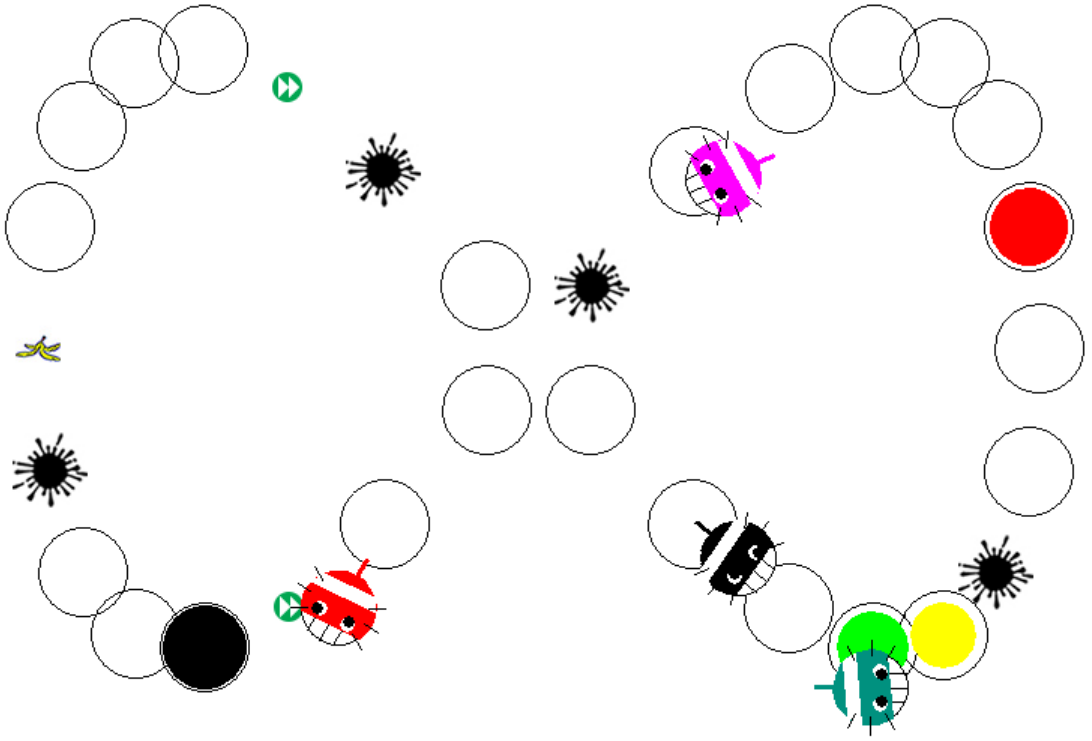
Generelle krav:

- Bruk dei eksakte namn og spesifikasjonar gjeve i oppgåva.
- Det anbefalast å nytte ein programmeringsomgjevnad(IDE) slik som Visual Studio eller XCode.
- 70% av øvinga må godkjennast for at den skal vurderast som bestått.
- **Hugs å lage eit TDT4102-grafikk prosjekt for denne øvinga. Start også med ein heilt tom main-fil.**
- Øvinga skal godkjennast av stud.ass. på sal.

Tilrådd lesestoff:

- FLTK-dokumentasjonen

I denne øvinga er målet å lage eit fullstendig program for enkel simulering av 2D-kappløp. Ein skal kunne styre eit køyretøy sjølv, men også kunne lage ein algoritme for styring. 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. I tillegg skal vi prøve ut eit par nye ting frå standardbiblioteket.

FLTK-dokumentasjonen finn du her, og denne skal vi nytte oss ein del av. I denne øvinga vil du likevel få konsis veiledning for mykje av FLTK, og før du startar oppgåvene bør du lese faktaboksa på neste side. Men det er anbefalt å lese i dokumentasjonen utover dette.

1 FLTK introduksjon (20%)

Alle filene vi skal nytte i øvinga er tilgjengelege i TDT4012-grafikkmalen, men sidan vi skal klare oss utan `Graph_lib`, så må vi manuelt inkludere alle dei relevante filene. Etter at du har oppretta grafikkprosjektet skal du slette alle `#include`-linjene eller starte med ei heilt tom main-fil. Følg Nyttig å vite-boksen under for å inkludere dei nødvendige filene.

a) I `main`, lag ein `Fl_Double_Window` og deretter kall funksjonen `Fl::run()`.

`Fl::run()` ligg i `<FL/Fl.H>`. I FLTK skal vindauget vere dynamisk allokert, så `new` og `delete` (eller `unique_ptr`) må brukast. Rett etter vindauget er konstruert (og før `Fl::run()`) må medlemsfunksjonane `end()` og `show()` kallast for å få det opp. Dette skal åpne eit tomt vindauge.

Dersom ein vil ha ein annan bakgrunnsfarge kan den setjes vha. medlemsfunksjonen `color()` til `Fl_Double_Window`. FLTK har innebygde farger som kan verte funne her i dokumentasjonen under "Colors".

b) Lag ei ny klasse `Vehicle` som arvar frå `Fl_Widget`, og implementer ein konstruktør som inntil vidare tek ingen parameter.

Konstruktøren til `Fl_Widget` må kallast på slik at instansen av `Vehicle` er innanfor vindauget og slik breidda og høgda er større enn 0, men elles er det ikkje viktig kva posisjon og storleik den får. Til dømes `Fl_Widget{10,10,10,10}`.

c) Lag funksjonen `void draw() override` i `Vehicle`.

Dette er ein pure-virtual funksjonen i `Fl_Widget` som vert kalla når vindauget teiknar elementa som er kopla til det. Føreløpig kan du berre teikne noko vilkårleg for testing (f.eks. `fl_circle`), og for å utføre teikning må du nytte funksjonane i `<FL/Fl_draw.H>`. Sjå dokumentasjonen under "Drawing Functions" for ei liste og forklaringar for desse funksjonane. To ting som er viktig å vere merksam på er at all teikning skal skje `draw()`-funksjonar, og at du bør spesifisere farge vha. funksjonen `fl_color()` før du kallar nokre av teiknefunksjonane. Du kan gi fargen som RGB verdi eller som ein av enum-verdiane her under overskrifta "Colors".

Nå kan du leggje inn ein dynamisk instans av `Vehicle` rett etter du konstruerar vindauget, altså før `end()`. Sjå om du får opp det du teiknar.

d) Eit problem er at vindauget ikkje vil teikne elementa ofte og er følgjeleg dårleg tilpassa for animasjon. Så i staden for `Fl::run()`, sett inn ein `while`-loop. Den kan du køyre så lenge vindauget er oppe, noko som du kan sjekke med dens medlemsfunksjon `shown()`. Loopen skal kalle funksjonane `Fl::check()`, som oppdaterer ulike input i vindauget, og `Fl::redraw()`, som teiknar alle elementa.

e) Eit nytt problem no er at programmet er avhengig av hastigheita til datamaskinen, noko som kan gi store skilnadar i køyringa mellom ulike maskiner. Vi vil ha 60 bilete (teikninger av vindauget) per sekund i dette programmet (grensa for dei fleste skjermar). For å oppnå dette nyttar vi tidsbiblioteket til STL, `<chrono>`, og nederst her er eit døme på bruk. `<chrono>` er ikkje pensum i TDT4102, så sjølv om vi anbefalar at du prøvar oppgåva sjølv, så har vi laga eit løysingsforslag i appendiks B. Der er også ei anbefalt løysing som ikkje belastar datamaskina så mykje.

Nyttig å vite: FLTK 1.3 i eit nøtteskal (detaljar ikkje pensum)

`Graph_lib` byggjer på FLTK, men til tross for fleire fellestrekk så er der nokre viktige skilnadar. Typane og funksjonane i FLTK startar med prefikset "`Fl`", og desse finn du i tilsvarande headerfiler under <FL/...>. I tillegg er standarden at dei fleste objekt i FLTK bør vere dynamisk allokerte med `new`, men FLTK pleier ofte å ta ansvaret med deallokering.

I denne øvinga skal vi nytte `Fl_Double_Window` som lerret, og denne finn du i `<FL/Fl_Double_Window.H>`. I likheit med GUI-delen frå `Graph_lib`, så fungerer dette vindauget med å kople til klassar som arvar frå ei spesifikk superklasse, `Fl_Widget`. Når vindauget vert teikna og oppdatert, så vil den respektivt kalle `draw()` og `handle()`-funksjonane til dei tilkopla klassene. I denne øvinga skal vi halde oss til å bruke `draw()`-funksjonen.

`FL_Widget` finn du i `<FL/Fl_Widget.H>`. Alle `Fl_Widget` bør allokerast dynamisk etter FLTK-standarden, så sørg for å bruke `new` når du lagar instansar av `Vehicle`. Men du slepp å bruke `delete` sidan vindauget som objektet er kopla til slettar alle elementa sine i destruktøren sin. Merk at du derfor ikkje kan bruke `unique_ptr`, for det ville ført til at den slettast dobbelt.

Når vi brukte `Graph_lib` måtte vi kalle ein funksjon med elementet som argument for å kople det til vindauget, men i FLTK vil `Fl_Widget` konstruktøren automatisk kople seg til vindauget som vart oppretta sist. Dette er eigentleg styrt av vindauget sine medlemsfunksjonar `begin()` og `end()` som gjer at alle `Fl_Widget` som er oppretta mellom desse funksjonskalla automatisk koplar

seg til vindauget. Vindauget kallar `begin()` i konstruktøren sin, og derfor må vi kalle `end()` for å markere at vi er ferdig å leggje til element.

2 Implementasjon av køyring (30%)

a) Lag ein ny struct, `PhysicsState`, som du gir til `Vehicle` som medlemsvariabel.

Denne structen må du leggje i ei ny headerfil, `"utilities.h"`. Køyretøya våre vil ha ein del fysiske størrelser som posisjon, retning og fart. Det er derfor gunstig å samle desse i ein struct. Structen skal ha desse medlemane:

```
double x, y, angle, vel, grip = 1;
```

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`.

b) Legg inn tastaturstyring slik du kan flytte på din `Vehicle`, og sørg for at teikninga er avhengig av `Vehicle` sin `PhysicsState`.

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". F.eks. `"if( Fl::event_key(FL_UP))"` sjekker om piltast opp er nedtrykt. Styringa kan leggjast i `draw()`. Her i byrjinga kan du endre direkte på `x`- og `y`-posisjonen til objektet sin `PhysicsState`.

c) 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 variablar, `velAcc` og `angAcc`. Desse er to lokale variablar i `draw` som vi vil skal liggje i intervallet `[-1,1]`, og dei representerer kor kraftig fartsendringa og svinginga er og i kva retning. Det er viktig at du oppdaterer dei fysiske størrelsene til køyretøyet sin `PhysicsState` (her kalt `ps`) nøyaktig på måten under. cos og `sin` finn du i `<cmath>`.

```
ps.grip += 0.01*(1-ps.grip);
ps.vel += 0.006 * ps.grip * velAcc * (6-abs(ps.vel) + 5*((ps.vel > 0) != (velAcc > 0)));
ps.angle += 0.03 * ps.grip * angAcc;
ps.x += vel * cos(ps.angle); y += ps.vel * sin(ps.angle);
```

Som du ser, så er oppdatering av farten ein del meir komplisert enn dei andre pga. introduksjon av eit parentesuttrykk: ledda `6-abs(vel)` gir ein maksfart og gjer at akselerasjonen ved låge hastigheiter er større; det siste leddet gjer at bremsing er raskare. Oppdateringa av `grip` er for at køyretøyet gradvis får tilbake det opprinnelege veggrepet sitt.

Før du oppdaterer størrelsene, så må du sørgje for at `velAcc` og `angAcc` er i intervallet `[-1,1]`. Vi vil også at køyretøyet skal halde seg innanfor skjermen til all tid. Det gjerast ved å setje `x` og `y` posisjonane til max/min verdien når dei går utanfor.

d) Legg inn ny styring som berre nyttar `velAcc` og `angAcc`, og fullfør `draw()`-funksjonen til køyretøyet.

Korleis styringa skal fungere er opp til deg. No skal det også vere mogeleg å fullstendiggjere funksjonen for teikning av køyretøyet ditt, men til det treng du radiusen til køyretøyet: Den skal vere 15 pikslar, så legg dette som eit globalt flyttal med namn `vehRad` i headerfila "utilities.h". For teikninga er der nokre krav du må følgje:

- Teikninga skal ha `(x,y)`-posisjonen til køyretøyet i midten
- Teikninga skal vere omtrent på størrelse med ein sirkel med radius `vehRad` pikslar.
- Noko av teikninga skal vere avhengig av `PhysicsState` sin `angle` slik at retninga til køyretøyet vert synleg. Ei mogeleg løysing på dette er å nytte `fl_arc` eller `fl_pie`, som begge tek inn seks argument. Ein annan måte er å teikne eit lite objekt, som ein sirkel, i den retninga du køyrer.

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`.

Nyttig å vite: Alias for typar (ikkje pensum, men viktig for øvinga)

Eit alias introduserer eit nytt typenamn som er ekvivalent med eit anna. Dette brukar vi ofte for å gjere lange, kompliserte typer meir konsise og for å gjere det enklare å endre store delar av eit program:

```
using vecIt = std::vector<std::string>::iterator;
vecIt it; // it er av type std::vector<std::string>::iterator.
```

Funksjonstyper kan ofte vere uoversiktlege, så dette gjer dei til utmerkte kandidatar for eit alias. I deklarasjonen skriv du først returtypen, så parametrane i parentes:

```
// Deklarerer eit alias for funksjonstypen
using drivingAlgorithm = std::pair<double,double> ( PhysicsState ps,
        const std::vector<std::pair<double,double>>& goals,
        int currentGoal);
```

Dette gjer det mykje meir leseleg å deklarere og bruke funksjonar og funksjonspeikarar:

```
drivingAlgorithm* pDA; // pDA er ein peikar til ein slik funksjon
drivingAlgorithm myDrive; // Deklarasjon av slik funksjon med namn myDrive.
```

Implementasjonen av `myDrive` må ha heile funksjonstypen.

e) 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 måte er å lage ei subklasse, men her skal vi nytte funksjonspeikarar i staden. Styringa skal skje i ein funksjon med denne funksjonstypen:

```
std::pair<double,double> myDrive( PhysicsState ps,
                const std::vector<std::pair<double,double>>& goals,
                int currentGoal);
```

Funksjonen skal ikkje vere ein medlemsfunksjon. Denne returnerer eit par som høvesvis er `velAcc` og `angAcc`. `goals` og `currentGoal` skal vi bruke i neste oppgåve. Eit triks vi skal bruke for å redusere skriving og auke lesbarheit er å bruke eit alias. Sjå boksen over.

Endr konstruktøren til `Vehicle` slik at den tek inn ein `driveAlgorithm*` og lagrar det i ein privat variabel.

Legg definisjonen av `drivingAlgorithm`-aliaset i "utilities.h". Funksjonen skal du kalle i `draw`, og returverdien skal verte brukt som `velAcc` og `angAcc`. For å kalle funksjonen til ein funksjonspeikar, må du nytte peikarnamnet som eit funksjonsnamn. Inntil vidare kan du skrive inn `{},0` som dei to siste parametrane. Flytt tastaturstyringa frå `draw()` til ein eigen `drivingAlgorithm` funksjon, og test ein `Vehicle` som brukar denne styringa.

f) Gjer liknande for teikning slik at kvar `Vehicle` kan få ein unik utsjånad.

Aliaset for denne funksjonen skal også liggje i "utilities.h" og skal vere som følgjer:

```
using drawingAlgorithm = void(PhysicsState ps);
```

g) Frivillig: Lag eit nytt køyretøy og gi den ein ny styrefunksjon som brukar andre tastar til styring. Då kan du kan kontrollere to køyretøy samtidig. Hugs at `Fl::event_key()` tek inn store bokstavar.

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`.

Er ikkje mykje til eit bilrace utan ei bane. Vektoren skal ha punkt som representerer bana bilen skal køyre, og det skal fungere med at køyretøyet må vere innanfor ein viss radius på kvart punkt for å gå vidare.

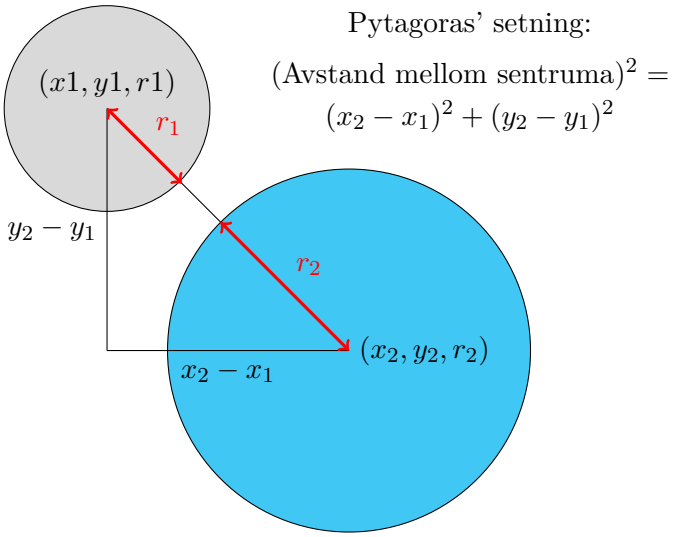
I konstruktøren til `Track` kan du også sette opp ein vilkårleg bane. Hugs at konstruktøren til `Fl_Widget` må kallast her også, men parametrane er enno likegyldige (så lenge verdiane er innanfor vindauget). Sørg også for at instansen som er kopla til vindauget er dynamisk allokert (men utan å nytte `unique_ptr`).

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 det første biletet.

c) Lag ein fri funksjon `bool circleCollision(double delX, double delY, double sumR)` 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 radiusene. Sjå komande figur for eit geometrisk oppsett. Kva slags sirkel som er 1 eller 2 har ikkje noko å seie:



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

d) Endr `Vehicle`-klassa slik at den lagrar ein konstant referanse til ein `Track`, og endr `Vehicle`-konstruktøren til å ta inn ein `Track` referanse som parameter.

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 heiltal som representerer indeksen til det gjeldande målet, og skriv kode som sjekker om køyretøyet har treft målet sitt.

Det er meir ryddig å leggje denne koden i ein eigen medlemsfunksjon som du kallar i `draw`. 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 kollidering må du ha definert radius for målet. Desse kan du ha som `constexpr double` i "utilities.h". Gi den namnet `goalRad` og ein radius på 30 pikslar. Under testing er det lurt å teikne det neste målet for køyretøyet i dets `draw` funksjon.

f) Lag ein ny enum i "utilities.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 ein funksjon som returnerer ein konstant referanse til den.

Kvar `tuple` i denne vektoren representerer ei hindring på bana. Hugs også å inkludere `<tuple>`. Sett opp desse hindringene i konstruktøren til `Track`, og teikn dei i `draw` funksjonen til `Track` som sirklar med kvar sin farge. Alternativt kan du sjå i appendiks A dersom du vil nytte bilete i staden. Til dette skal du definere nokre radiuser i "utilities.h", og dei skal heite `spillRad`, `boostRad`, og `peelRad` respektivt vere 20, 10, og 5 pikslar.

I `Vehicle` må du sjekke om køyretøyet kolliderer med ein av desse, og det er ryddig å gjere dette i ein eigen medlemsfunksjon. Dersom den treff ei hindring, er det ofte nødvendig å markere at den har kollidert med ei slik hindring. Til dette kan du lage ein `Obstacle`-medlemsvariabel `status` som du setter til å vere den trufne hindringa. 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 desse krava må følgjast:

- `Spill`: Eit oljesøl vil så klart redusere veggrepet ditt. Sett derfor `grip` til 0.5. Effekten må forsvinne gradvis, og det er det oppdateringa av `grip` som vi sette inn oppgåve 2 gjer. Det bør ikkje vere nødvending å endre `status` til `Spill` her.
- `Boost`: Måten vi skal representere dette på, er å auke `grip` til 2. Her er det heller ikkje nødvendig å endre `status` til `Boost`.
- `Peel`: Viss du har treft eit bananskal og `vel` er større enn 2, skal du bremse fullt og setje `angAcc` til 8. Du må dermed ignorere returverdiane frå styrefunksjonen din. Dette skal fortsetje så lenge `vel > 0.05`. Du skal også skli langs den vinkelen du traff bananskalet med, så du må lagre den vinkelen i ein variabel som du brukar i staden for `angle` så lenge du sklir. Her er det lurt å markere `status` som `Obstacle::Peel` til du er ferdig å skli.

4 Køyring/teikning algoritmar (20%)

Dette vil vere ei ganske åpen oppgåve der du skal sjølv lage ein algoritme for eit sjølvkøyrande køyretøy. Dette skal du gjere i ein `drivingAlgorithm`-funksjon som vi kan plugge inn i køyretøyet. Dersom du ikkje har gjort det enno, så skal du også lage din eigen `drawingAlgorithm` for å gi køyretøyet ditt ein unik utsjånad. Ein viktig ting du må passe på er at desse to algoritmene ikkje er avhengig av andre lokale filer enn "utilities.h", så legg dei i eit eiget header/.cpp-par som berre inkluderar "utilities.h" som lokal fil. Du kan inkludere så mange filar du vil frå STL/grafikkmalen som då er av formen `#include <filename>`, men ingen andre enn "utilities.h" med hermeteikn.

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 reikne ut prikkproduktet og determinanten til vektorane og gi dette til `atan2`-funksjonen. Vi har laga ein funksjon som gjer dette for deg i appendiks B.
- 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 den nye og gamle mot kvarandre.
- Hugs at du kan lage statiske objekt og variablar i funksjonen. Det kan du kanskje finne eit bruksområde for.

Dersom du er fornøgd med din `drivingAlgorithm` og/eller `drawingAlgorithm`, så vil vi gjerne at du sender dei inn til faglærar! Send header- og cpp-filene til "Lasse@computer.org" med emnet "Øving 12 Algoritmer", og så får dei kanskje vere med i eit kappløp siste forelesing. Men då må du vere ekstra obs på at filene ikkje inkluderar andre lokale filar enn "utilities.h". Lukke til!

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. Merk at bileteklassene berre er behaldarar og trengjer ikkje å verte kopla til eit vindauge, så då er det heller ikkje nødvendig å gjere objektene dynamiske som med `Fl_Widget`. 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!

Appendiks B: Kode for utvalde oppgåver

Oppgåve 1 e)

```
    #include <chrono> // På toppen
    // Før hovud-while-løkka
    auto then = std::chrono::steady_clock::now();
        // I while-løkka
        double timePassed = 0;
        do {
                auto now = std::chrono::steady_clock::now();
                timePassed = std::chrono::duration<double>(now - then).count();
        } while (timePassed < 1.0 / 60);
        then = std::chrono::steady_clock::now();
```

Problemet med løysinga over er at den vil belaste programmet unødvendig når den kun treng å vente. Ei betre løysing som brukar `<thread>` (ikkje pensum) er derfor gjeve under:

```
    #include <chrono> // På toppen
    #include <thread>
    // Før hovud-while-løkka
    auto next = std::chrono::steady_clock::now();
        // I while-løkka
        std::this_thread::sleep_until(next);
        next += std::chrono::microseconds(1000000 / 60);
```

Oppgåve 4

```
double angleBetween( std::pair<double, double> vec1,
                     std::pair<double, double> vec2)
    // Returnerer vinkel mellom to vektorar i radianar i intervallet <-pi,pi].
{
    double dot = vec1.first * vec2.first + vec1.second * vec2.second;
    double det = vec1.first * vec2.second - vec1.second * vec2.first;
    return std::atan2(det,dot);
}
```