Part III: Conway's Game of Life

Maksimal skår for 3 er 200 poeng.

Introduksjon til Conway's Game of Life

Conway's Game of Life er ein cellulær automaton. Det er ei simulering av eit 2D-univers med rektangulære cellar som deterministisk utvikler sin tilstand over tid. Hvis du køyrer ei simulering med dei same initialbetingelsane vil tilstanden i eit gjeven tidssteg vere det same mellom distinkte simuleringar. Det er eit antal forskjellige livsformer, mønster, som kan representeres i Game of life. Nokre mønster forflytt seg i universet med tida, dei kalles romskip. I den utdelte `zip`-fila vil du finne filar med filending `.cgol` som inneheld et utval romskip.

Di oppgåve er å implementere logikken som set liv til universet. Cellar er den primære byggeblokka i universet og du vil lage eit uendelig rutenettsunivers av celler. For kvart tidssteg som går ska tilstanden i universet reflektere eit sett reglar som er bestemt for Game of life. Kva som bestemmer utfallet av reglane er antal levande og døde nabocellar og den noverande tilstanden til cella regelen anvendes på.

Vi har inkludert ein kort demo-video av korleis programmet ser ut når det er ferdig implementert. Videoen ligg i `zip`-fila og heiter `game-of-life_demo.mp4`. I videoen demonstrerast handlingane som kan utføres og korleis dei utartar seg i det endelege programmet (klippet er utan lyd).

Korleis besvare del 3?

Alle oppgåva i del 3 er satt opp slik at dei skal besvarast i fila `Gameoflife.cpp`. Kvar oppgåve har ei tilhørande unik kode for å gjere det lettare å finne fram til kor i fila du ska skrive svaret. Koden er på formatet `<teikn><siffer> (TS)`, eksempelvis `C1`, `C2` og `G1`. I `Gameoflife.cpp` vil du for kvar oppgåve finne to kommentarar som definerer høvevis begynnelsa og slutten av koden du skal føre inn. Kommentarane er på formatet:

`// BEGIN: TS` og `//END: TS`.

Det er veldig viktig at alle svara dine er skrivne mellom slike kommentar-par, for å støtta sensurmekanikken vår. Viss det allereie er skrive nokon kode mellom BEGIN- og END-kommentarane i filene du har fått utdelt, så kan, og ofte bør, du erstatta den koden med din eigen implementasjon.

Til dømes, for oppgåve C1 ser du følgjande kode i utdelte `Gameoflife.cpp`

```
int Cell::get_value() const {
    // BEGIN: C1
    return 0;
    // END: C1
}
```

Etter at du har implementert løysinga di, bør du enda opp med følgjande i staden for

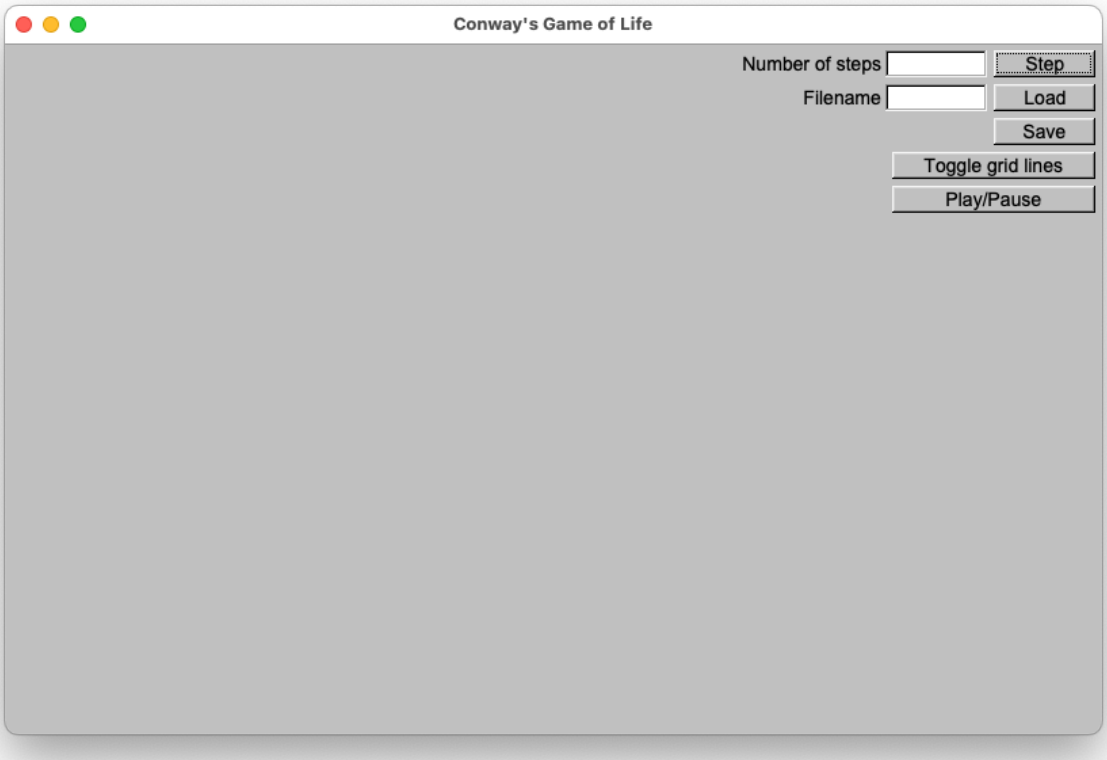
```
int Cell::get_value() const {
    // BEGIN: C1

    /* Din kode her */

    // END: C1
}
```

Merk at BEGIN- og END-kommentarane **IKKJE skal fjernast**.

Til slutt, viss du synest nokon av oppgåvene er uklare, si korleis du tolkar dei og skriv korleis du antek at oppgåva skal løysast som kommentarar i den koden du sender inn.

I `zip`-fila finner du m.a. `Gameoflife.h` og `Gameoflife.cpp`. Det er kun `Gameoflife.cpp` som ska redigerast for å komme i mål med oppgåvene. Headerfila inneheld klassedefinisjoner og konstantar du bør ta ei titt på før du startar å svare på spørsmåla i denne delen.

Før du startar må du sjekke at den (umodifiserte) utdelte koden køyrar uten problem. Du skal se det samme vindauget som i figur 1. Når du har sjekka at alt fungerer som det ska er du klar til å starte programmering av svara dine.



Figur 1: Utlevert kode utan endringar.

Klassa Cell (80 poeng)

Vi delar ut ein ikkje komplett versjon av klassa `Cell` som representerar ei celle i Game of life-universet. Di oppåve er å gjere cellen synleg på skjermen, endre dens tilstand (vekke den til live eller ta livet av den) og oppdatere den grafiske representasjonen.

Ei `Cell` har nokre attributtar du bør kjenne til. `enum class State` er ein *scoped enum* som enumererar ei celles tilstand, om den er i live eller død. Medlemsvariabelen `State state` holder orden på cellas noverande tilstand. `shared_ptr<Rectangle> rect` er den grafiske representasjonen av ei celle på skjermen.

Du kan bruke ein `shared_ptr` på samme måte som du bruker ein `unique_ptr` eller rå peikar, f.eks. derefereringsoperatoren, `*`. Vi brukar ein `shared_ptr` i denne eksamenen for å automatisk rydde opp `Rectangle`-instansen som er delt mellom kopiar av eit `Cell`-objekt.

På eit tidspunkt vil du lese og skrive frå ei fil, og kanskje terminalen for å debugge. Til det formålet har vi definert ein tilstand til tegn-tabell i tabell 1. Vi har delt ut eit sett tabellar og avbildningar(`map`) du bør vurdere å anvende for å omsetje mellom tilstand og tegn-representasjonen. Dei er definert i header-fila `Gameoflife.h` og heiter `Cell::chars` og `Cell::char_to_state`.

Tilstand	Teikn
Live	`'#'` (hash)
Dead	`'.'` (punktum)

Tabell 1: Tilstand til teikn-mapping.

1. (10 points) **C1: Implementer medlemsfunksjonen** `Cell::get_value()`.

   Denne medlemsfunksjonen returnerer heiltallsrepresentasjonen av tilstandsvariabelen.
   (Hint: scope enum-en `State` representerer sine enumeratorar med heiltallsverdiar.)

   Verdiane som skal returneres frå denne funksjonen er 0 for døde cellar og 1 for levande cellar.

2. (10 points) **C2: Implementer** `Cell::update()`.

   Medlemsfunksjonens oppgåve er å oppdatere den grafiske tilstanden til cella, altså setje fyllfargen av medlemsvariabelen `rect`. Ei død celle er *svart* og ei levande celle er *kvit*. Vi har delt ut eit array som heiter `colors` i klassedefinisjonen til `Cell` i fila `Gameoflife.h` og oppfordrar til bruk av dette for å løyse oppgåva.

   Hugs å kall på denne funksjonen når du endrar tilstanden til ei celle.

3. (10 points) **C3: Implementer** `Cell::kill()`.

   Sett cellas tilstand til State::Dead.

   Hugs også å oppdatere cella grafiske tilstand.

4. (10 points) **C4: Implementer** `Cell::resurrect()`.

   Funksjonen komplementerar `Cell::kill()` og skal gjere det motsatte av den, nemleg bringe cella tilbake til live.

5. (10 points) **C5: Implementer** `Cell::set_state(char c)`.

   Programmen vil snart lese teikn frå ei fil og du må konvertere tegnrepresentasjonen lest frå fila til ein tilstand, `State`. Omsetjinga er lista i tabell 1 og vi har delt ut eit *constant* map namngjeve `Cell::char_to_state` som inneheld denne mappinga.

   Input-argumentet `c` er teiknet som representerer tilstanden cella skal ha blitt tilegna når funksjonen har returnert. Du kan anta at `c` alltid er gyldig og er anten `'.'` eller `'#'`.

6. (10 points) **C6: Implementer** `operator>>(istream& is, Cell& cell)`.

   Denne operatoroverlastinga les eit teikn frå input-strømmen og oppdaterer tilstanden til cella basert på input-verdien.

7. (10 points) **C7: Implementer** `Cell::is_alive()`.

   Medlemsfunksjonen returnerer `true` hvis cella er i live, `false` hvis ikkje.

8. (10 points) **C8: Implementer** `Cell::as_char()`.

   Denne medlemsfunksjonen returnerer teikn-representasjonen av cellas tilstand.

   Bruk den same tilstand til teikn-tabellen som før, frå tabell 1. Vi har delt ut eit array som heitar `Cell::chars` som passar fint til denne oppgåva.

Universet (90 poeng)

No ska vi ta fatt på universet og leggje til reglar så vi kan få eit innblikk i korleis Game of life fungerer.

Klassa `Gameoflife` er definert i `Gameoflife.h` og dei aller fleste definisjonane av medlemsfunksjonane befinn seg i `Gameoflife.cpp`. Du bør ta ein titt i headerfila for å få eit overblikk over medlemmene til klassa `Gameoflife`.

Legg merke til linja `using Grid = std::vector<std::vector<Cell>>`. Denne linja lagar eit alias med namn `Grid` vi kan bruke til erstatning for ein 2D-vector som inneheld `Cell`-objekt. Klassa lagrar to `Grid` i medlemsvariabelen `std::array<Grid, 2> grid`. Til einkvar tid vil eit av grid-ane (rutenetta) vere det noverande og det er det noverande grid-et som ska visast på skjermen, samstundes vil det andre grid-et vere midlertidig og du kan bruke det som mål for neste noverande tilstand. Det er nyttig når universets tilstand ska endre seg med tida, sidan vi treng kjennskap til den umodifiserte noverande tilstanden heilt til den neste tilstanden er heilt ferdig berekna. For å få tak i den `noverande` tilstanden kan du kalle på medlemsfunksjonen `Gameoflife::get_current_grid()` og motsett `Gameoflife::get_scratch_grid()` for å få tilgang til den `midlertidige`.

For å halde styr på kva grid som for augeblinket lagrar verdiane for noverande og midlertidig tilstand har vi oppretta to heiltallsvariablar: `current_grid` og `scratch_grid`. Til einkvar tid ska ein av dem innehalde heiltallsverdien 0 og den andre innehalde 1. `Gameoflife::get_current_grid()` og `Gameoflife::get_scratch_grid()` returnerer noverande og midlertidig grid basert på verdiane i desse variablane. *Du må endre verdiane til variablane `current_grid` og `scratch_grid` for å bytte på kva grid som er noverande og midlertidig når det er ein overgang som krev to separate tilstander.*

Vi har også definert nokre konstantar i `Gameoflife.h` for å hjelpe deg å laga eit rutenett av cellar som passar vindauget. `int margin` holder ein verdi som gjer at GUI-element i vindauget held avstand til kvarandre. `int cell_size` held cellas grafiske størrelse i x- og y-aksen - alle cellar er representert av kvadratiske rektangel på skjermen.

Du kan anta en konstant størrelse for Game of life-grid i alle oppgåvane. Alle utdelte univers-tilstander er formatert som rutenett med størrelse $50x50$. Det er den same størrelsen som er lagra i konstantane `x_cells` og `y_cells`.

9. (30 points) **G1: Implementer konstruktøren til** `Gameoflife`.

   Di oppgåve er å initialisere dei to 2D-rutenetta som inneheld `Cell`-objekt. Når du har gjort denne oppgåva ska vindauget sjå ut som det i figur 2.

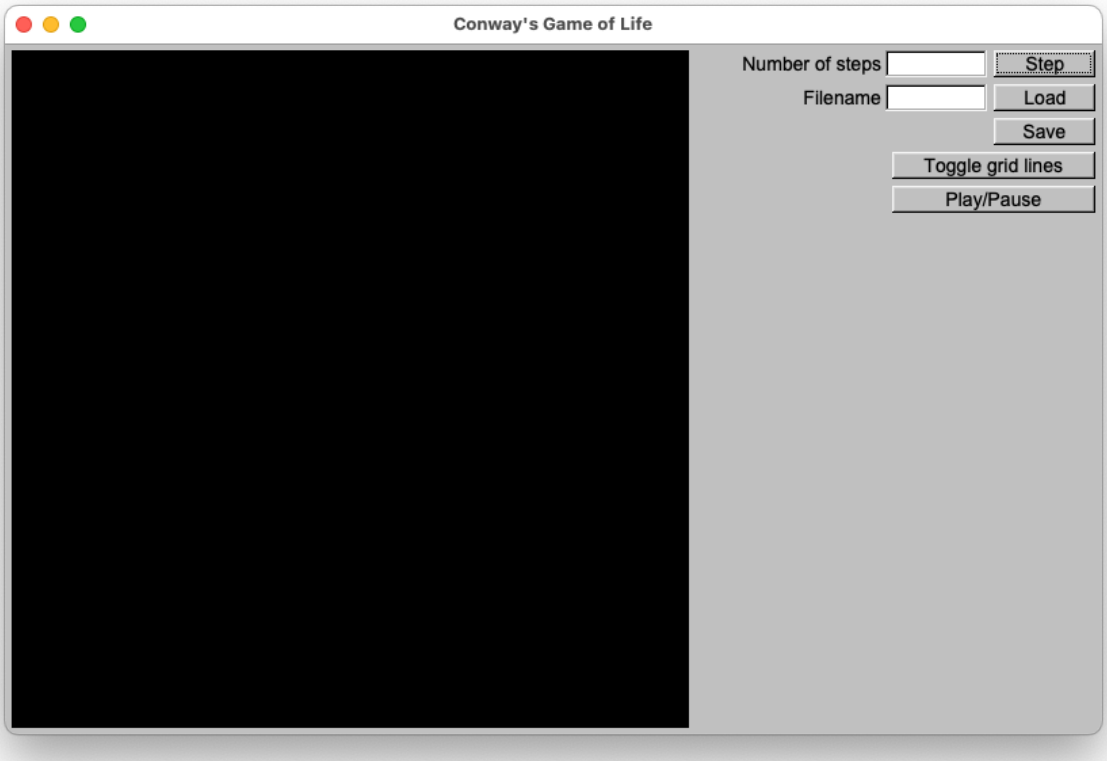
   I `Gameoflife.h` finn du konstantane `cell_size` og `margin` som du bør bruke for å plassere cellerektangla i eit rutenett inne i vindauget. Det grafiske rutenettet ska plasserast `margin` punkt frå topp- og venstrekanta til vindauget. Antall cellar på x- og y-aksen er lagra i hhv. variablane `x_cells` og `y_cells`.

   Vi anbefaler at du først fyllar ut eitt av rutenetta og kallar på cellanes medlemsfunksjon `attach_to()` **i det, før du deretter gjer ei kopi-tilordning (tilordningsoperatoren) av heile rutenettet til det andre (tomme) rutenettet.** *Dette vil fungere sidan kopiering av ein `shared_ptr` kun vil kopiere peikaren, ikkje instansen det peikes til.*

   Vi har delt ut medlemsfunksjonen `Cell::attach_to(Window& win)` som koblar den grafiske representasjonen til eit vindauge. Sjå til at denne funksjonen kallas for kun ein kopi av eit `Cell`-objekt.

Korleis legg du til radar til ein 2D-vector, korleis legg du til eit element i ein 2D-vector og korleis koblar du eit element til vindauget? Vi viser tre kodelinjer i figur 3 som viser korleis du kan lage eit rutenett med størrelsen $1x1$ med ein `Cell` som har størrelse 40 plassert i vindaugets posisjon $(0, 0)$.

(Du vil sjå at delar av konstruktøren allereie er definert. Desse delane tek seg av å koble GUI-element til vindauget og sjekkar at input-argumenta er positive. Den gjer også at vindaugets og dets element held ei konstant størrelse når programmet køyrer.)



Figur 2: Game of life med kun døde celler (etter oppgåve 9. G1).

```
// Add a new row to the grid
get_current_grid().push_back({});
// Add a Cell to that row
get_current_grid().back().push_back(Cell{Point{0,0}, 40});
// Attach that Cell to the window
get_current_grid().back().back().attach_to(*this);
```

Figur 3: Eksempelkode som set opp eit $1x1$-rutenett med ei celle med størrelse 40 plassert i vindaugets $(0, 0)$.

10. (10 points) **G2: Implementer** `operator>>(istream& is, Gameoflife& gameoflife)`.

    Les frå input-strømmen inn i den samsvarande cella i den noverande tilstanden til ditt Game of life.

    Input-formatet er `y_cells` antall linjar med `x_cells` teikn per linje. Sjå for øvrig i ei av dei utleverte filane med filending `.cgol` korleis ei fullstendig initialtilstand ser ut. Vi har lista eit eksempel-input

med størrelse $5x5$ i figur 4.

```
.....
..#..
...#.
.###.
.....
```

Figur 4: Eksempel-input med størrelse $5x5$ som inneheld romskipet kalla "glider".

11. (10 points) **G3: Implementer** `Gameoflife::load(const std::string& filename)`.

    Denne medlemsfunksjonen lastar inn ein ny tilstand frå fila gitt i argumentet `filename`. Hvis fila ikkje kan opnas eller lesast ska det kastast eit unntak, `std::runtime_error`, med meldinga "Could not load a Game of life state from `<filename>`.", der du byter ut `<filename>` med filnavnet programmet ikkje kan lese.

    Kall på medlemsfunksjonen `Window::redraw()` etter at tilstanden er lest frå fila for å signalisere til vindauget at det ska oppdateres med endringane i den grafiske tilstanden til cellane.

    Når du har implementert denne funksjonen ska det vere muleg å laste inn universa vi har delt ut som del av `.zip`-fila. Om du lastar inn `glider.cgol` ska du sjå eit vindauge som liknar det i figur 5.

    Tips: for å spare tid og slippe å skrive inn eit filnamn kvar gong programmet startar kan du leggje til linja `load("glider.cgol")` på slutten av `Gameoflife`-konstruktøren for å laste inn glideren automatisk kvar gang programmet startar på ny.

12. (30 points) **G4: Implementer** `Gameoflife::step()`.

    Denne oppgåva omhandlar `Gameoflife::step()` *utan* parameter i signaturen.

    Medlemsfunksjonen vil overføre universet frå tilstand til tilstand ettersom det utviklar seg over tid. Ei celles tilstand i tid $n + 1$ er eit produkt av universets tilstand i tid n.

    Korleis ei celles liv utviklar seg frå eit tidspunkt til det neste kan defineres av desse tre reglane: (`https://en.wikipedia.org/wiki/Conway's_Game_of_Life`)

    1. Einkvar levande celle med to eller tre levande naboar overlever.
    2. Einkvar død celle med tre levande naboar blir til ei levande celle.
    3. Alle andre celler døyr i den neste generasjonen. Likeins, all andre døde cellar forblir døde.

    Merk at du har to rutenett til disposisjon. Eit som holder den noverande tilstanden som også er den som visest grafisk i vinduet, og eit midlertidig rutenett du kan bruke som kladd (scratchpad) og mål for neste tilstand.

    Din oppgåve er å implementere overførselen frå den noverande tilstanden til den neste tilstanden. Når `Gameoflife::step()`-funksjonen har returnert forventes det at både den nye noverande tilstanden og den grafiske representasjonen (`Cell`-ens `rect`-medlemsvariabel) i Game of life-vinduet reflekterer den nylig berekna tilstanden.

    Universet i Game of life er uendeleg, sjølv om vårt rutenett er endeleg i 2D-rommet. Dette betyr at du må folde rundt (wrap around) rutenettets kantar for å berekne eit steg i tid korrekt. Denne oppførselen er illustrert i figur 6.

    Ver obs på at modulo-operatoren (%) har ein noko ikkje-intuitiv oppførsel ved behandling av negative heiltal, som kan bli eit problem ved folding rundt kantane. Positive heiltal fungerer som vi forventar, t.d. $4\%3 = 1$ og $3\%4 = 3$. Negative tall, derimot, vil for samme eksempel gi $-4\%3 = -1$ og $-3\%4 = -3$.

    I det siste eksempelet ser vi at -3 ikkje er heiltalsresultatet vi ønsker oss for å folde rundt kanten. Vi hadde ønskt å få resultatet 1.

Figur 5: Glider innlasta.

Ein måte å få modulo-oppførselen vi ønsker er å leggje saman venstre- og høgresida, deretter ta modulo av summen med høgresida. Algebraisk får vi da at $a\%n$ blir $(a+n)\%n$. Fullføring av eksempelet med $-3\%4$ blir da $(-3+4)\%4 = 1$, som ønskt.

13. (10 points) **G5: Implementer** `Gameoflife::step(int steps)`.

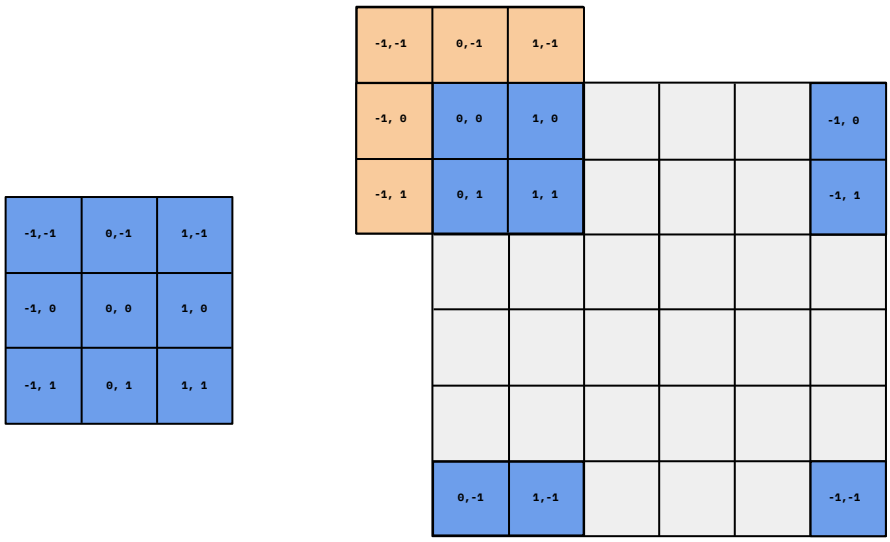
    Denne medlemsfunksjonens oppgåve er å gjere framskritt i tilstanden med `steps` antall tidstrinn.

    Når funksjonen har utført forflytningen i tid må du kalle på den arva medlemsfunksjonen `Window::redraw()` for å signalisere til vindauget at du har oppdatert grafikk inni det.

    For å verifisere at programlogikken er korrekt kan du laste inn tilstanden fra fila `glider.cgol` og utføre 50 steg. Vindauget ditt skal etter stegene likne det i figur 7. Du kan bruke knappen "Toggle grid lines" for å skru av eller på linjar i rutenettet for å lettere sjå korleis cellane i universet utviklar seg.

    Tips 1: når du skriv inn eit tal i boksen med merkelappen "Number of steps" og trykk på "Step"-knappen vil tida forflyttes med antall tidssteg i intervallet $[1, 100]$ - det betyr m.a. at 0 som verdi blir 1 steg når du trykk "Step".

    Tips 2: når denne funksjonen er implementert kan du også trykke "Play/Pause"-knappen for å automatisk utvikle tida slik at du får ein animasjon i vindauget. Hvis du vil endre oppdateringsraten til animasjonen kan du sjå på variabelen `animation_interval` i `Gameoflife.h`.



Figur 6: Eksempel på folding rundt kantar. Den blå kjernen til venstre anvendast på det lysegrå rutenettet til høgre. Bruk av kjernen vises overlagt i rutenettets punkt $(0, 0)$. Orange firkantar viser delane av kjernen som må foldast rundt kantane til rutenettet. Dei blå firkantane er dei konkrete avbildningane.

Ekstrafunksjonalitet (30 poeng)

Veksle (toggle) ei celle med peikarklikk

I den siste serien med oppgåve ska du gjere rutenettet klikkbart og veksle tilstanden til cella det klikkes på. Vi har allerede oppretta ein behandler som fangar opp peikerklikk inni vindauget. Din oppgåve er å prosessere klikka, dvs. finne ut om klikket skjedde inni rutenettet og veksle cella under peikaren.

14. (10 points) **E1: Implementer** `Gameoflife::cell_at_pos(Point pos)`.

    Denne metoden tek inn ein posisjon som argument og returnerer ein peikar til `Cell`-objektet som befinn seg på den posisjonen i vindauget. Hvis det ikkje er ei celle på posisjonen returneres ein `nullptr`.

    Merk at ein posisjon som treff ei celle i den grafiske representasjonen er ei celle i den *noverande* tilstanden, så du skal returnere ein peikar til ei celle frå det rutenettet.

15. (10 points) **E2: Implementer** `Cell::toggle()`.

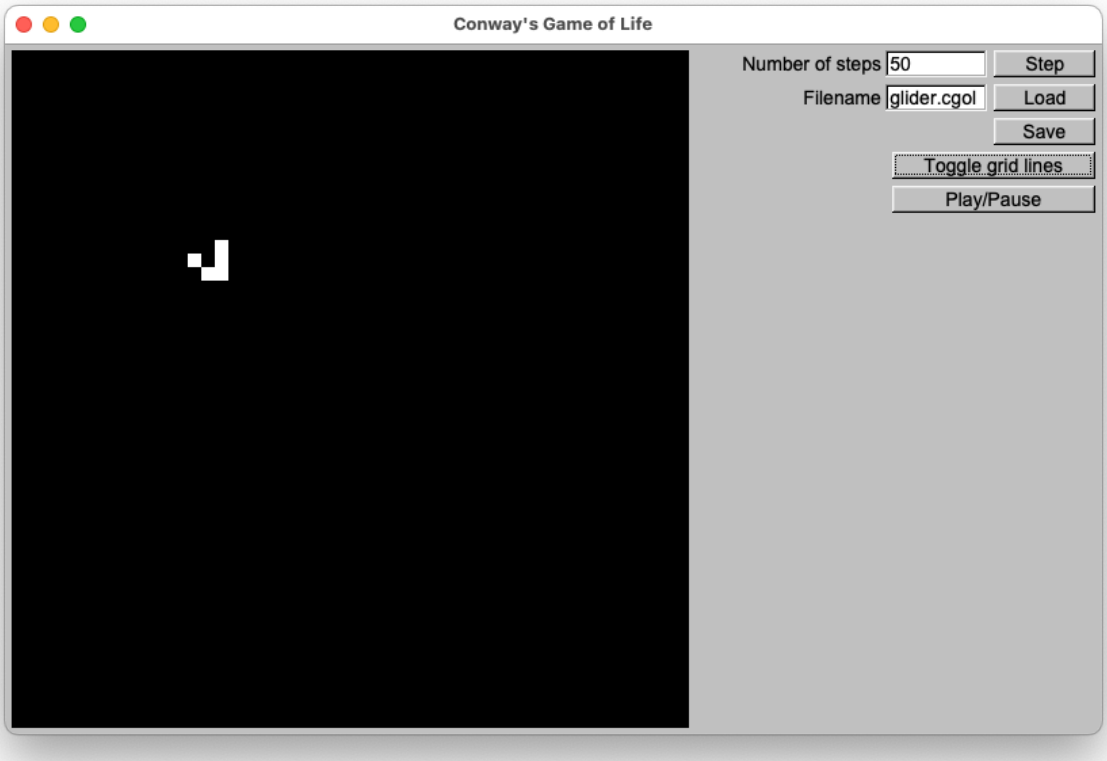
    Denne medlemsfunksjonen ska veksle ei celles tilstand. Er cella død ska den bli levande og omvendt skal den bli død hvis den er levande.

16. (10 points) **E3: Implementer** `Gameoflife::toggle_cell(Point pos)`.

    Medlemsfunksjonen tar inn ein posisjon og vekslar tilstanden til cella på den posisjonen. Hvis den ikkje lykkast i å veksle ei celle, t.d. fordi at det ikkje finst ei celle på posisjonen, ska den returnere `false`, ellers ska den returnere `true`.

    Siden du endrar på den noverande (og grafiske) tilstanden må du signalisere til vindauget at grafikken er oppdatert.

    Når du har gjennomført denne oppgåva ska du kunne klikke på ei vilkårlig celle i rutenettet og visuelt sjå at cellas tilstand vekslar mellom levande og død.



Figur 7: Ein glider lasta frå textttglider.cgol etter 50 steg.