Part III: Conway's Game of Life

Maksimal score for del 3 er 200 poeng.

Introduksjon til Conway's Game of Life

Conway's Game of Life er en cellulær automaton. Det er en simulering av et 2D-univers med rektangulære celler som deterministisk utvikler sin tilstand over tide. Hvis du kjører en simulering med de samme initialbetingelsene vil tilstanden i et gitt tidssteg være det samme mellom distinkte simuleringer. Det er et antall forskjellige livsformer, mønster, som kan representeres i Game of life. Noen mønster forflytter seg i universet med tiden, de kalles romskip. I den utdelte `zip`-filen vil du finne filer med filendelse `.cgol` som inneholder et utvalg romskip.

Din oppgave er å implementere logikken som setter liv til universet. Celler er den primære byggeblokken i universet og du vil lage et uendelig rutenettsunivers av celler. For hvert tidssteg som går skal tilstanden i universet reflektere et sett regler som er bestemt for Game of life. Hva som bestemmer utfallet av reglene er antall levende og døde naboceller og den nåværende tilstanden til cellen regelen anvendes på.

Vi har inkludert en kort demo-video av hvordan programmet ser ut når det er ferdig implementert. Videoen ligger i `zip`-filen og heter `game-of-life_demo.mp4`. I videoen demonstreres handlingene som kan utføres og hvordan de utarter seg i det endelige programmet (klippet er uten lyd).

Hvordan besvare del 3?

Alle oppgavene i del 3 er satt opp slik at de skal besvares i filen `Gameoflife.cpp`. Hver oppgave har en tilhørende unik kode for å gjøre det lettere å finne frem til hvor i filen du skal skrive svaret. Koden er på formatet `<tegn><siffer> (TS)`, eksempelvis `C1`, `C2` og `G1`. I `Gameoflife.cpp` vil du for hver oppgave finne to kommentarer som definerer henholdsvis begynnelsen og slutten av koden du skal føre inn. Kommentarene er på formatet:

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

Det er veldig viktig at alle svarene dine er skrevet mellom slike kommentar-par, for å støtte sensurmekanikken vår. Hvis det allerede er skrevet noen kode mellom BEGIN- og END-kommentarene i filene du har fått utdelt, så kan, og ofte bør, du erstatte den koden med din egen implementasjon.

For eksempel, for oppgave C1 ser du følgende kode i utdelte `Gameoflife.cpp`

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

Etter at du har implementert din løsning, bør du ende opp med følgende istedenfor

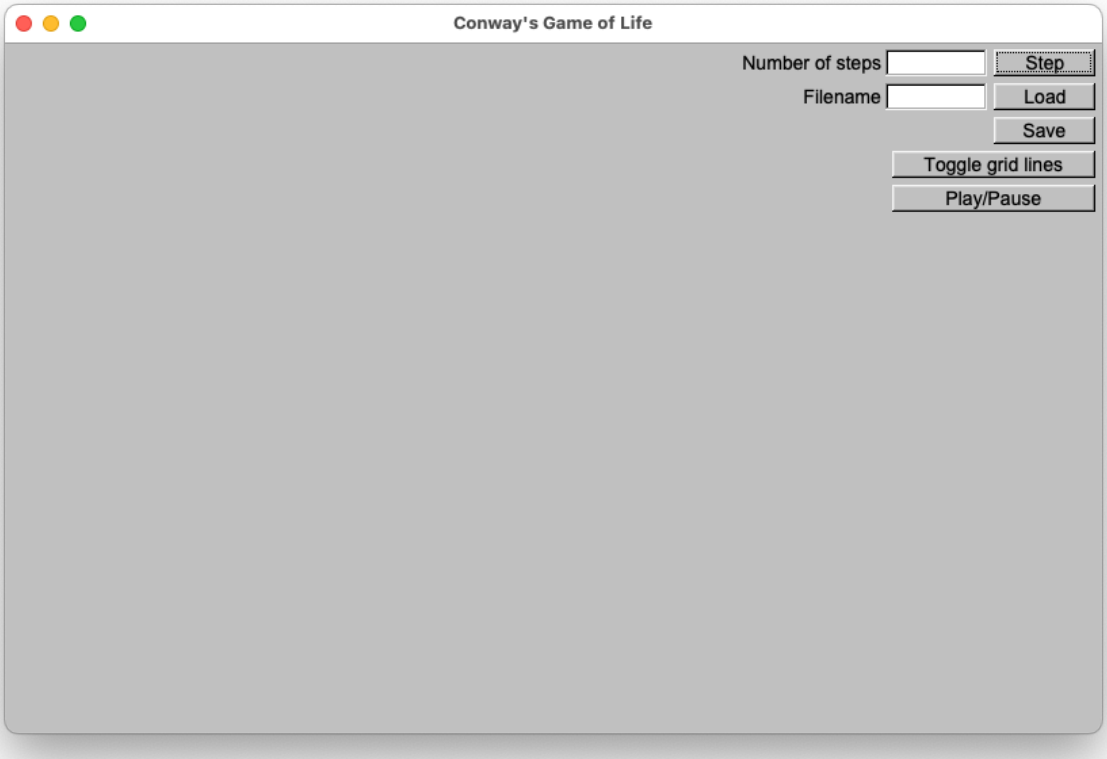
```
int Cell::get_value() const {
    // BEGIN: C1

    /* Din kode her */

    // END: C1
}
```

Merk at BEGIN- og END-kommentarene **IKKE skal fjernes**.

Til slutt, hvis du synes noen av oppgavene er uklare, oppgi hvordan du tolker dem og de antagelsene du må gjøre som kommentarer i den koden du sender inn.

I `zip`-filen finner du bl.a. `Gameoflife.h` og `Gameoflife.cpp`. Det er kun `Gameoflife.cpp` som skal redigeres for å komme i mål med oppgavene. Headerfilen inneholder klassedefinisjoner og konstanter du bør ta en titt på før du starter å svare på spørsmålene i denne delen.

Før du starter må du sjekke at den (umodifiserte) utdelte koden kjører uten problemer. Du skal se det samme vinduet som i figur 1. Når du har sjekket at alt fungerer som det skal er du klar til å starte programmering av svarene dine.



Figur 1: Utlevert kode uten endringer.

Klassen Cell (80 poeng)

Vi deler ut en ikke komplett versjon av klassen `Cell` som representerer en celle i Game of life-universet. Din oppgave er å gjøre cellen synlig på skjermen, endre dens tilstand (vekke den til live eller ta livet av den) og oppdatere den grafiske representasjonen.

En `Cell` har noen attributter du bør kjenne til. `enum class State` er en *scoped enum* som enumererer en celles tilstand, om den er i live eller død. Medlemsvariabelen `State state` holder orden på cellens nåværende tilstand. `shared_ptr<Rectangle> rect` er den grafiske representasjonen av en celle på skjermen.

Du kan bruke en `shared_ptr` på samme måte som du bruker en `unique_ptr` eller rå peker, f.eks. dereferingsoperatoren, `*`. Vi bruker en `shared_ptr` i denne eksamenen for å automatisk rydde opp `Rectangle`-instansen som er delt mellom kopier av et `Cell`-objekt.

På et tidspunkt vil du lese og skrive fra en fil, og kanskje terminalen for å debugge. Til det formålet har vi definert en tilstand til tegn-tabell i tabell 1. Vi har delt ut et sett tabeller og avbildninger(`map`) du bør vurdere å bruke for å oversette mellom tilstand og tegn-representasjonen. De er definert i header-filen `Gameoflife.h` og heter `Cell::chars` og `Cell::char_to_state`.

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

Tabell 1: Tilstand til tegn-mapping.

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

   Denne medlemsfunksjonen returnerer heltallsrepresentasjonen av tilstandsvariabelen.
   (Hint: scope enum-en `State` representerer sine enumeratorer med heltallsverdier.)

   Verdiene som skal returneres fra denne funksjonen er 0 for døde celler og 1 for levende celler.

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

   Medlemsfunksjonens oppgave er å oppdatere den grafiske tilstanden til cellen, altså sette fyllfargen av medlemsvariabelen `rect`. En død celle er *svart* og en levende celle er *hvit*. Vi har delt ut et array som heter `colors` i klassedefinisjonen til `Cell` i filen `Gameoflife.h` og oppfordrer til bruk av dette for å løse oppgaven.

   Husk å kall på denne funksjonen når du endrer tilstanden til en celle.

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

   Sett cellens tilstand til State::Dead.

   Husk også å oppdatere cellens grafiske tilstand.

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

   Funksjonen komplementerer `Cell::kill()` og skal gjøre det motsatte av den, nemlig bringe cellen tilbake til live.

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

   Programmen vil snart lese tegn fra en fil og du må konvertere tegnrepresentasjonen lest fra filen til en tilstand, `State`. Oversettingen er listet i tabell 1 og vi har delt ut et *constant* map navngitt `Cell::char_to_state` som inneholder denne mappingen.

   Input-argumentet `c` er tegnet som representerer tilstanden cellen skal ha blitt tilegnet når funksjonen har returnert. Du kan anta at `c` alltid er gyldig og er enten `'.'` eller `'#'`.

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

   Denne operatoroverlastingen leser et tegn fra input-strømmen og oppdaterer tilstanden til cellen basert på input-verdien.

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

   Medlemsfunksjonen returnerer `true` hvis cellen er i live, `false` hvis ikke.

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

   Denne medlemsfunksjonen returnerer tegn-representasjonen av cellens tilstand.

   Bruk den samme tilstand til tegn-tabellen som før, fra tabell 1. Vi har delt ut et array som heter `Cell::chars` som passer fint til denne oppgaven.

Universet (90 poeng)

Nå skal vi ta fatt på universet og legge til regler så vi kan få et innblikk i hvordan Game of life fungerer.

Klassen `Gameoflife` er definert i `Gameoflife.h` og de aller fleste definisjonene av medlemsfunksjonene befinner seg i `Gameoflife.cpp`. Du bør ta en titt i headerfilen for å få et overblikk over medlemmene til klassen `Gameoflife`.

Legg merke til linjen `using Grid = std::vector<std::vector<Cell>>`. Denne linjen lager et alias med navn `Grid` vi kan bruke til erstatning for en 2D-vector som inneholder `Cell`-objekter. Klassen lagrer to `Grid` i medlemsvariabelen `std::array<Grid, 2> grid`. Til enhver tid vil et av grid-ene (rutenettene) være det nåværende og det er det nåværende grid-et som skal vises på skjermen, samtidig vil det andre grid-et være midlertidig og du kan bruke det som mål for neste nåværende tilstand. Det er nyttig når universets tilstand skal endre seg med tiden, siden vi trenger kjennskap til den umodifiserte nåværende tilstanden helt til den neste tilstanden er helt ferdig beregnet. For å få tak i den `nåværende` tilstanden kan du kalle på medlemsfunksjonen `Gameoflife::get_current_grid()` og motsatt `Gameoflife::get_scratch_grid()` for å få tilgang til den `midlertidige`.

For å holde styr på hvilket grid som for øyeblikket lagrer verdiene for nåværende og midlertidig tilstand har vi opprettet to heltallsvariabler: `current_grid` og `scratch_grid`. Til enhver tid skal en av dem inneholde heltallsverdien 0 og den andre inneholde 1. `Gameoflife::get_current_grid()` og `Gameoflife::get_scratch_grid()` returnerer nåværende og midlertidig grid basert på verdiene i disse variablene. *Du må endre verdiene til variablene `current_grid` og `scratch_grid` for å bytte på hvilket grid som er nåværende og midlertidig når det er en overgang som krever to separate tilstander.*

Vi har også definert noen konstanter i `Gameoflife.h` for å hjelpe deg å lage et rutenett av celler som passer vinduet. `int margin` holder en verdi som gjør at GUI-elementer i vinduet holder avstand til hverandre. `int cell_size` holder cellens grafiske størrelse i x- og y-aksen - alle celler er representert av kvadratiske rektangler på skjermen.

Du kan anta en konstant størrelse for Game of life-grid i alle oppgavene. Alle utdelte universtilstander er formatert som rutenett med størrelse $50x50$. Det er den samme størrelsen som er lagret i konstantene `x_cells` og `y_cells`.

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

   Din oppgave er å initialisere de to 2D-rutenettene som inneholder `Cell`-objekter. Når du har gjort denne oppgaven skal vinduet se ut som det i figur 2.

   I `Gameoflife.h` finner du konstantene `cell_size` og `margin` som du bør bruke for å plassere cellerektanglene i et rutenett inne i vinduet. Det grafiske rutenettet skal plasseres `margin` punkter fra topp- og venstrekantene til vinduet. Antall celler på x- og y-aksen er lagret i hhv. variablene `x_cells` og `y_cells`.

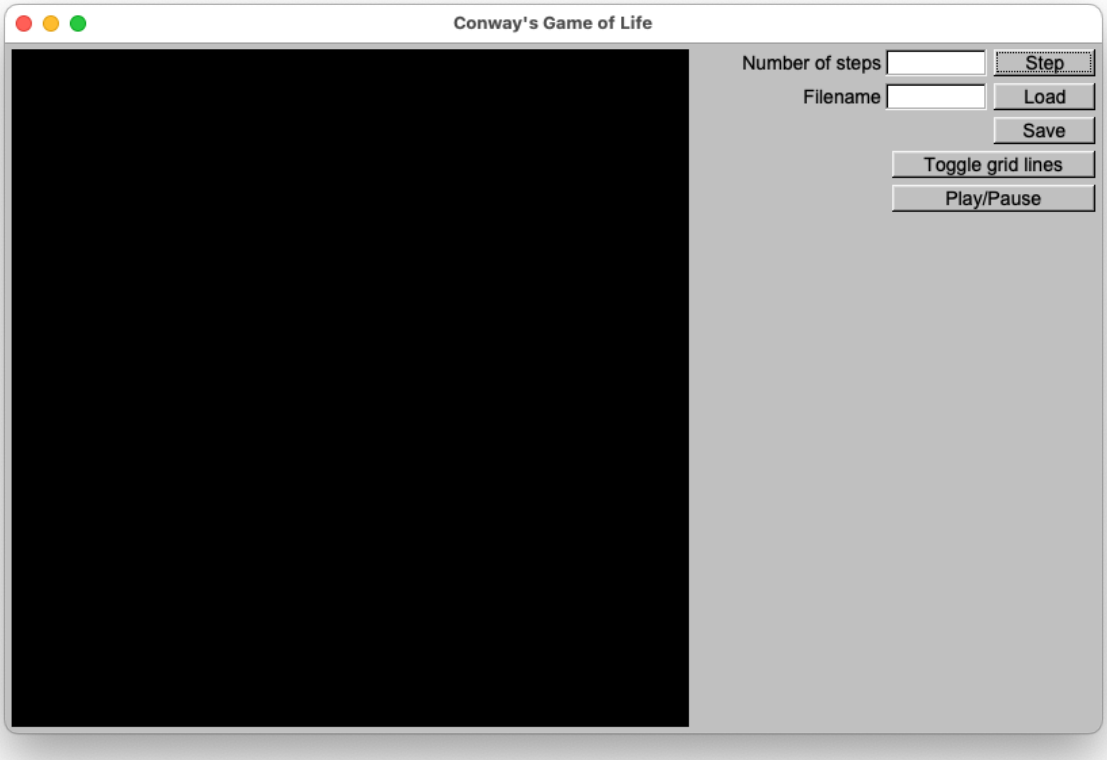
   Vi anbefaler at du først fyller ut ett av rutenettene og kaller på cellenes medlemsfunksjon `attach_to()` i det, før du deretter gjøre en kopi-tilordning (tilordningsoperatoren) av hele rutenettet

til det andre (tomme) rutenettet. *Dette vil fungere siden kopiering av en* `shared_ptr` *kun vil kopiere pekeren, ikke instansen det pekes til.*

Vi har delt ut medlemsfunksjonen `Cell::attach_to(Window& win)` som kobler den grafiske representasjonen til et vindu. Se til at denne funksjonen kalles for kun en kopi av et `Cell`-objekt.

Hvordan legger du til rader til en 2D-vector, hvordan legger du til et element i en 2D-vector og hvordan kobler du et element til vinduet? Vi viser tre kodelinjer i figur 3 som viser hvordan du kan lage et rutenett med størrelsen $1x1$ med en `Cell` som har størrelse 40 plassert i vinduets posisjon $(0, 0)$.

(Du vil se at deler av konstruktøren allerede er definert. Disse delene tar seg av å koble GUI-elementer til vinduet og sjekker at input-argumentene er positive. Den gjør også at vinduet og dets elementer holder en konstant størrelse når programmet kjører.)



Figur 2: Game of life med kun døde celler (etter oppgave 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 setter opp et $1x1$-rutenett med en celle med størrelse 40 plassert i vinduets $(0, 0)$.

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

    Les fra input-strømmen inn i den samsvarende cellen i den nåværende tilstanden til ditt Game of life.

    Input-formatet er `y_cells` antall linjer med `x_cells` tegn per linje. Se for øvrig i en av de utleverte filene med filendelse `.cgol` hvordan en fullstendig initialtilstand ser ut. Vi har listet et eksempel-input med størrelse $5x5$ i figur 4.

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

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

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

    Denne medlemsfunksjonen laster inn en ny tilstand fra filen gitt i argumentet `filename`. Hvis filen ikke kan åpnes eller leses skal det kastes et unntak, `std::runtime_error`, med meldingen "Could not load a Game of life state from `<filename>`.", der du bytter ut `<filename>` med filnavnet programmet ikke kan lese.

    Kall på medlemsfunksjonen `Window::redraw()` etter at tilstanden er lest fra filen for å signalisere til vinduet at det skal oppdateres med endringene i den grafiske tilstanden til cellene.

    Når du har implementert denne funksjonen skal det være mulig å laste inn universene vi har delt ut som del av `.zip`-filen. Hvis du laster inn `glider.cgol` skal du se et vindu som ligner det i figur 5.

    Tips: for å spare tid og slippe å skrive inn et filnavn hver gang programmet starter kan du legge til linjen `load("glider.cgol")` på slutten av `Gameoflife`-konstruktøren for å laste inn glideren automatisk hver gang programmet starter på nytt.

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

    Denne oppgaven handler om `Gameoflife::step()` *uten* parametere i signaturen.

    Medlemsfunksjonen vil overføre universet fra tilstand til tilstand ettersom det utvikler seg over tid. En celles tilstand i tid $n + 1$ er et produkt av universets tilstand i tid n.

    Hvordan en celles liv utvikler seg fra et tidspunkt til det neste kan defineres av disse tre reglene: (`https://en.wikipedia.org/wiki/Conway's_Game_of_Life`)

    1. Enhver levende celle med to eller tre levende naboer overlever.
    2. Enhver død celle med tre levende naboer blir til en levende celle.
    3. Alle andre celler dør i den neste generasjonen. Likedan, alle andre døde celler forblir døde.

    Merk at du har to rutenett til disposisjon. Et som holder den nåværende tilstanden som også er den som vises grafisk i vinduet, og et midlertidig rutenett du kan bruke som kladd (scratchpad) og mål for neste tilstand.

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

    Universet i Game of life er uendelig, selv om vårt rutenett er endelig i 2D-rommet. Dette betyr at du må folde rundt (wrap around) rutenettets kanter for å beregne et steg i tid korrekt. Denne oppførselen er illustrert i figur 6.

    Vær obs på at modulo-operatoren (%) har en noe ikke-intuitiv oppførsel ved behandling av negative heltall, som kan bli et problem ved folding rundt kantene. Positive heltall fungerer som vi forventer,

Figur 5: Glider lastet inn.

f.eks. $4\%3 = 1$ 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 ikke er heltallsresultatet vi ønsker oss for å folde rundt kanten. Vi hadde ønsket å få resultatet 1.

En måte å få modulo-oppførselen vi ønsker er å legge sammen venstre- og høyresiden, deretter ta modulo av summen med høyresiden. Algebraisk får vi da at $a\%n$ blir $(a+n)\%n$. Fullføring av eksempelet med $-3\%4$ blir da $(-3+4)\%4 = 1$, som ønsket.

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

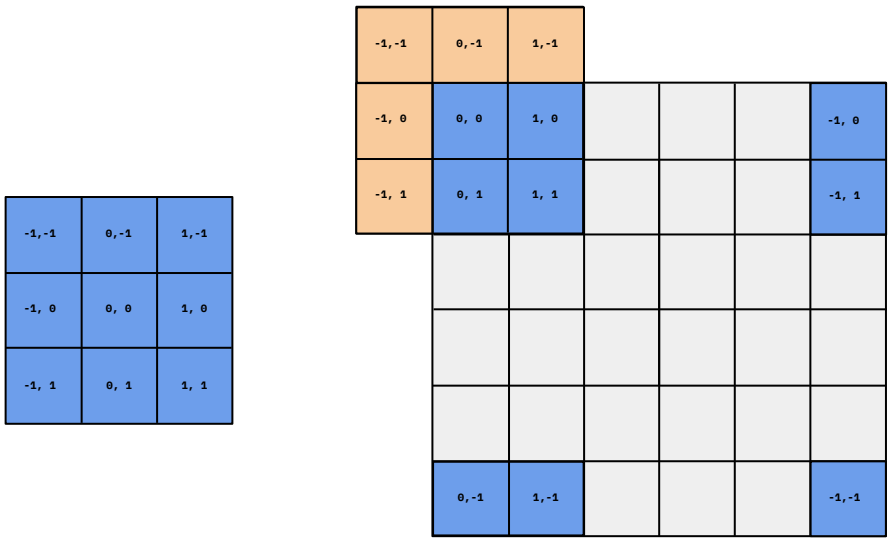
    Denne medlemsfunksjonens oppgave er å gjøre fremskritt i tilstanden med `steps` antall tidstrinn.

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

    For å verifisere at programlogikken er korrekt kan du laste inn tilstanden fra filen `glider.cgol` og utføre 50 steg. Vinduet ditt skal etter stegene ligne det i figur 7. Du kan bruke knappen "Toggle grid lines" for å skru av eller på linjer i rutenettet for å lettere se hvordan cellene i universet utvikler seg.

    Tips 1: når du skriver inn et tall i boksen med merkelappen "Number of steps" og trykker på "Step"-knappen vil tiden forflyttes med antall tidssteg i intervallet $[1, 100]$ - det betyr bl.a. at 0 som verdi blir 1 steg når du trykker "Step".

    Tips 2: når denne funksjonen er implementert kan du også trykke "Play/Pause"-knappen for å automatisk utvikle tiden slik at du får en animasjon i vinduet. Hvis du vil endre oppdateringsraten til



Figur 6: Eksempel på folding rundt kanter. Den blå kjernen til venstre anvendes på det lysegrå rutenettet til høyre. Anvendelsen av kjernen vises overlagt i rutenettets punkt $(0, 0)$. Orange firkanter viser delene av kjernen som må foldes rundt kantene til rutenettet. De blå firkantene er de konkrete avbildningene.

animasjonen kan du se på variabelen `animation_interval` i `Gameoflife.h`

Ekstrafunksjonalitet (30 poeng)

Veksle (toggle) en celle med pekerklikk

I den siste serien med oppgaver skal du gjøre rutenettet klikkbart og veksle tilstanden til cellen det klikkes på. Vi har allerede opprettet en behandler som fanger opp pekerklikk inni vinduet. Din oppgave er å prosessere klikkene, dvs. finne ut om klikket skjedde inni rutenettet og veksle cellen under pekeren.

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

    Denne metoden tar inn en posisjon som argument og returnerer en peker til `Cell`-objektet som befinner seg på den posisjonen i vinduet. Hvis det ikke er en celle på posisjonen returneres en `nullptr`.

    Merk at en posisjon som treffer en celle i den grafiske representasjonen er en celle i den *nåværende* tilstanden, så du skal returnere en peker til en celle fra det rutenettet.

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

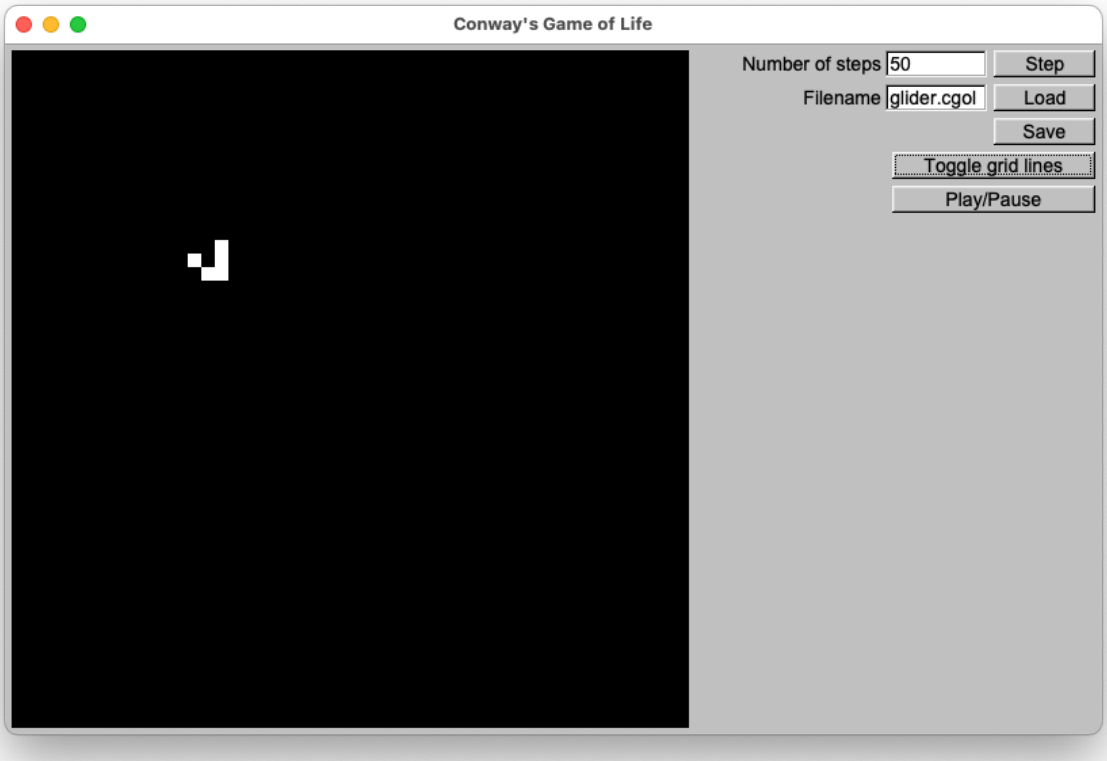
    Denne medlemsfunksjonen skal veksle en celles tilstand. Er cellen død skal den bli levende og omvendt skal den bli død hvis den er levende.

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

    Medlemsfunksjonen tar inn en posisjon og veksler tilstanden til cellen på den posisjonen. Hvis den ikke lykkes i å veksle en celle, f.eks. fordi at det ikke finnes en celle på posisjonen, skal den returnere `false`, ellers skal den returnere `true`.

    Siden du endrer på den nåværende (og grafiske) tilstanden må du signalisere til vinduet at grafikken er oppdatert.

    Når du har gjennomført denne oppgaven skal du kunne klikke på en vilkårlig celle i rutenettet og visuelt se at cellens tilstand veksler mellom levende og død.



Figur 7: En glider lastet fra `glider.cgol` etter 50 steg .