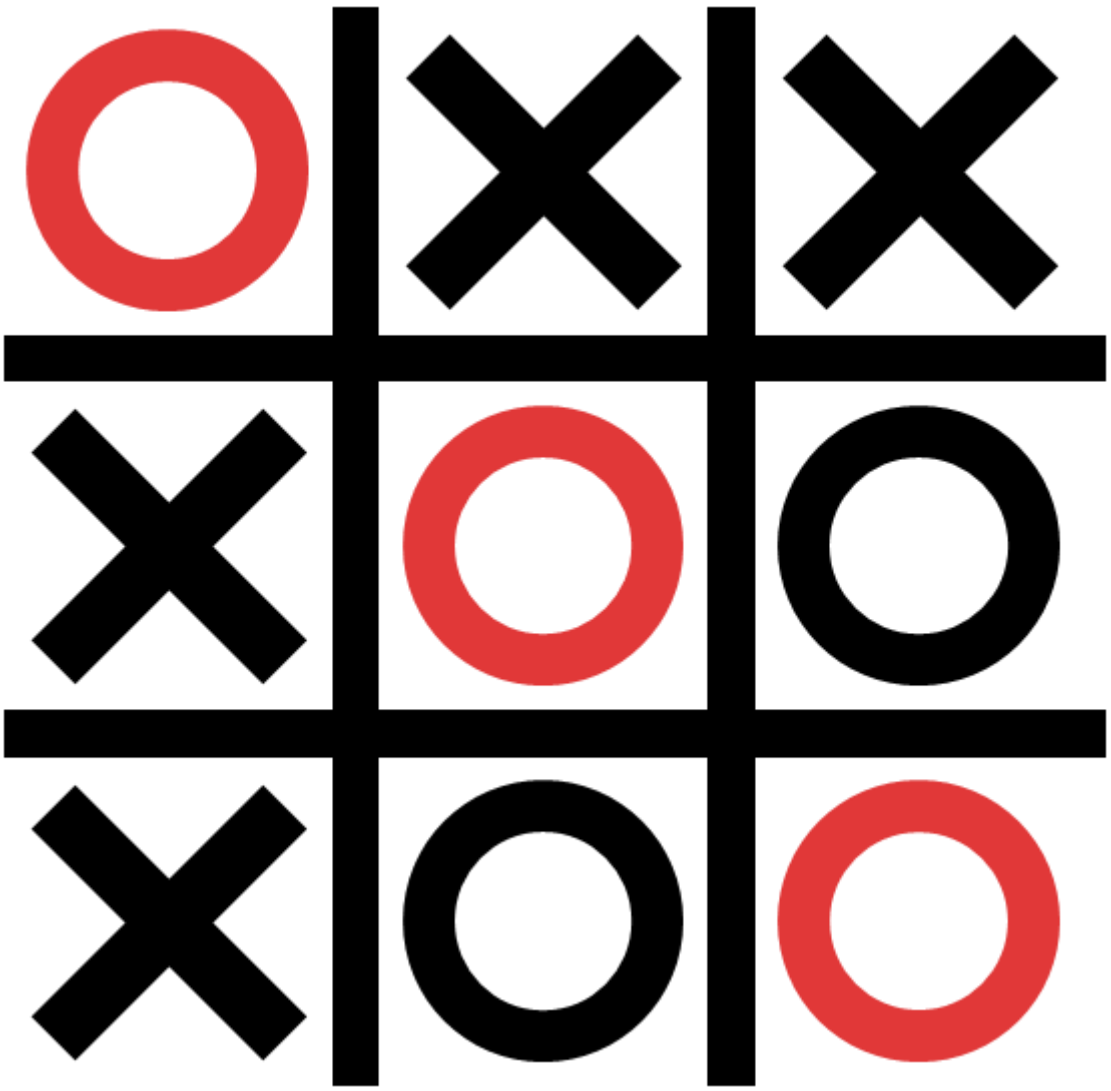
Kryds og bolle



Af: Thomas Therkelsen og Tobias Leth Skov

Titelblad

Uddannelsessted

Viborg Tekniske Gymnasium, Mercantec

Titel og undertitel

Kryds og Bolle

Fag

Programmering B

Sideantal

6½ normalsider

Projektperiode

18/02-2019 → 12/04-2019

Afleveringsdato

D. 12. april kl. 15:00

Vejleder

Jeppe Veirum Larsen

Klasse

3.z

Årgang

3.g 16-19

Forfattere

Synopsis: Thomas Therkelsen

Program: Thomas Therkelsen & Tobias Leth Skov

Indledning

I denne synopsis vil jeg beskrive Tobias' og min teori og proces for udviklingen og udarbejdningen af TicTacToe programmet som vi producerede i forbindelse med vores eksamensprojekt i Programmering B. Det består af en menu og et Kryds og Bolle spil som kan skaleres fra 3x3 op til 5x5 eller 7x7.

Indholdsfortegnelse

Figurliste

Projektbeskrivelse

Gruppen har fået til opgave at genskabe et af de tre givne spil i en digital version. Det spil som bliver genskabt digitalt, skal som minimum indeholde de basale funktioner der udgør spillet. Derudover er det selvfølgelig ikke tilladt at bruge kode eller dele af kode af andre som har løst samme opgave.

Problemanalyse

Regler

Kryds og Bolle er et paper-and-pencil game (der også brætspil baseret på konceptet), som består af et gitter på 3x3 felter. Det spilles af to spillere, som vælger enten kryds eller bolle, og så handler det om at få tre på stribe, hvad enten det er vertikalt, horisontalt eller diagonalt. I nogle kryds og bolle spil skal man lade brikkerne ligge som man har lagt, og blive ved med at lægge nye på, mens man i andre kun har tre brikker, og dermed skal genbruge og genplacere dem man allerede har lagt ned på pladen.

Problemafgrænsning

Der skal programmeres en digital version af Kryds og Bolle, uden anvendelse af kode skrevet af andre end gruppen selv. Med andre ord så skal der programmeres et spil fra bunden.

Problemet afgrænses til at genskabe Kryds og Bolle i en digital version, da der er potentiale i at udvikle spillet til at være mere end bare et standard Kryds og Bolle spil. Det normale Kryds og Bolle spil består af et grid med 3x3 felter hvor det handler om at få tre på stribe for at vende, det kan f.eks. udvikles til at have et grid som er så stort som brugerne ønsker det, lave betingelsen for at vinde om til at være så mange som brugerne ønsker det skal være. Derudover behøver det f.eks. ikke at være krydser og boller der bruges som avatarer, der kunne sagtens laves en avatar selection inden man starter spillet.

Krav

Kravene til produktet med prioritering, 1 er højest og 6 er lavest.

1. Basic game rules skal digitaliseres
    a. 3x3 spilleplade
    b. Win condition = 3 på stribe
    c. De to spillere skiftes til at lægge en brik på et felt
2. Skalerbart spil
    a. Skalerbar spilleplade (grid størrelse)
    b. Skalerbar win condition (antal brikker på stribe)
3. HUD til spillet
    a. Viser hvis tur det er
4. Karaktervalg (Valg af brik)
5. Menu til spillet
    a. Basale funktioner (Start spil, luk spil)
6. Evt. ekstra features
    a. Antal brikker man maks. kan bruge
    b. Genbrug af brikker eller altid nye brikker
    c. Simulation af 'fire på stribe' (dvs. tyngdekraft på brikkerne.).

Problemformulering

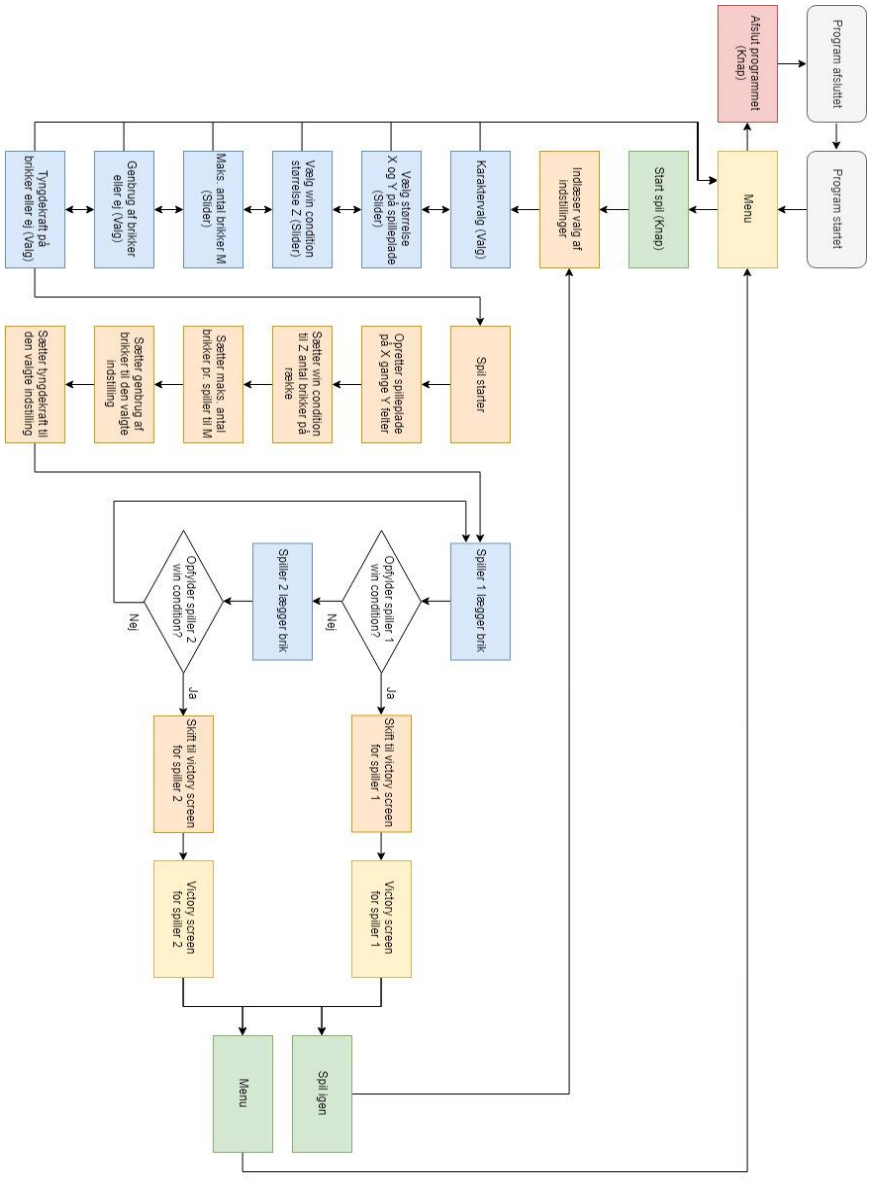
De problemer som skal løses, er dermed flg.:

- Hvilket coding environment er mest optimalt til genskabelse af Kryds og Bolle?
- Hvordan kan Kryds og Bolle genskabes i det valgte environment?
- Hvordan kan man bedst muligt skabe en menu?
- Hvordan kan man lave spillet skalerbar? (Specifikt ift. spillepladen og win condition)
- Hvordan kan der bedst muligt skabes et HUD til spillet?
- Hvordan kan man tilføje de ekstra features?
- Hvordan kan funktioner bedst muligt benyttes til at løse opgaven?
- Hvordan kan man få koden til at se overskuelig og let anvendelig ud?

Løsningsforslag

Flowchart over spillet

Herunder ses et flowchart-diagram over hvordan spillet ideelt vil fungere.
(Roteret 90° med uret for at det er læseligt, derudover kan det ses i fuld størrelse i bilag.)



Figur 1 - Flowchart over spillet

Coding Environment

Det allerførste problem i projektet som der skulle findes en løsning på, var at der skulle findes det mest optimale coding environment at bruge til at programmere spillet i. Her blev der overvejet to ret ensartede environments, nemlig det Java-baserede Processing og JavaScript-baserede p5.js.

Processing vs. p5.js

Processing

Processing er et fleksibelt objektorienteret IDE som er baseret på og benytter sig af programmeringssproget Java, dog inkluderer det masser af tilføjelser i form af Processings eget library som er inkluderet i programmet som standard, samt utallige af andre libraries som kan hentes gennem programmet, både lavet af Processing selv og af private brugere. Processing bruges ofte til at lave grafisk baserede programmer og har målet at kunne benyttes af alle programmører, hvad enten de er nybegyndere, øvede eller meget dygtige og erfarne programmører.

Figur 2 - Processing logo

Den største fordel for ved at bruge Processing, er at gruppen har mere erfaring i det ift. p5.js, både fra skolen men også i private sammenhænge, samt at de er mere kendte med Java programmeringssproget og dets syntax end med JavaScript, som p5.js benytter sig af.

p5.js

p5.js er reelt bare et JavaScript-baseret library med groft set de samme funktioner som Processing. Den største forskel på de to programmer er sproget som de benytter sig af, altså p5.js benytter sig af JavaScript hvor Processing jo benytter sig af Java. En anden stor forskel er at i p5.js bruger man en .html fil som lærred til at vise sit program, hvor man i Processing bare har et vindue til at vise det, ellers eksporterer man sit program til en .exe fil, altså et helt normalt program. Derudover inkluderer p5.js masser af libraries som gør det nemt at interagere med HTML objekter og benytte dem i samspil med sit program som jo bliver vist vha. en .html fil.

Figur 3 - p5.js logo

Valget

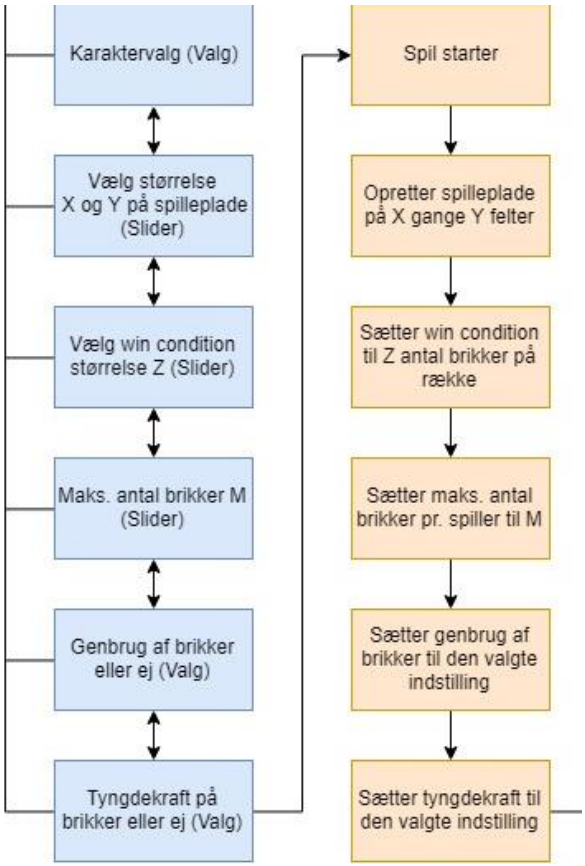
Ift. det endelige valg af environment blev der taget højde for gruppens tidligere erfaring med diverse environments og kodesprog og derudfra blev Processing valgt til fordel for p5.js, grundet gruppens erfaring i Processing og mangel på samme i p5.js. Valget af environment havde altså intet at gøre med programmets muligheder.

Dokumentation

Menu og Spil

I starten blev opgaven splittet op mellem os hvor Tobias skulle stå for at få lavet spillepladen til spillet og gøre så man kunne sætte X'er og O'er, mens jeg stod for at skulle lave en menu til spillet bestående af en titelskærm og en række options skærme hvor brugerne skulle vælge en række indstillinger til spillet.

Som det kan ses i et udsnit af spillets flowchart til højre, var det meningen at brugeren skulle kunne sætte en række indstillinger som karaktervalg, størrelse på spilleplade, størrelse på win condition, maks. antal brikker på banen ad gangen, genbrug af brikker og tyngdekraft på brikkerne. Dette skulle alt sammen vælges af brugerne inden de kunne gå i gang med at spille selve spillet.



Figur 4 - Udsnit af flowchart

Jeg startede ud med at lave en titelskærm samt en række options skærme hvor man skulle kunne sætte de forskellige indstillinger mens Tobias fik lavet et spil hvor man kunne sætte krydser og boller på pladen.

Yderligere afgrænsning af projektet

Herefter valgte vi at kombinere koden hvilket gik ret nemt da vi i forvejen havde snakket godt sammen om hvordan de problemstillinger skulle løses i programmet. Efter vi havde lagt koden sammen, besluttede vi kun at fokusere på størrelsen af spillepladen da det var det vi fandt mest spændende og skulle også planlægge ud fra den tid der var afsat til projektet.

Dvs. at jeg fik fjernet de overskydende options skærme fra programmet, sådan at der kun var titelskærmen, den første options skærm og det stadigt primitive spil tilbage.

MenuElements.pde

Ret hurtigt valgte jeg at lægge al menu-funktionalitet ind i en klasse for at spare plads i den primære kode samt at holde den overskuelig, dette valgte jeg bl.a. at gøre fordi jeg godt vidste at interaktion med knapperne ville komme til at fylde meget og at det hurtigt ville gøre main koden uoverskuelig at se på. I starten af projektets forløb var det MenuElements klassen som jeg brugte mest tid på at udvikle og færdiggøre så jeg kunne være med til at lavet spillet også.

Funktioner

Denne funktion indeholder en række funktioner som her vil blive beskrevet kort. Funktionerne title() og optionsTitle() bruges til at skrive overskriften på hhv. titelskærmen og options skærmen. Funktionen startBTN() bruges til at lave en knap hvor der står enten "Play" eller "Start" på, hvis man er på hhv. titelskærmen eller options skærmen. Når man klikker på knappen, går man et skridt frem, til options skærmen eller helt ind i spillet.

Herunder ses hvordan startBTN() vælger hvorvidt der skal stå "Play" eller "Start" (Uddrag fra en større funktion, se resten i bilag).

```
    //  Inserts text for button
    textAlign(CENTER, CENTER);
    fill(50);
    textSize(btnSz);
    if(gameState == 0) {
      text("Play", btn1PosX, btn1PosY - buffer2);
    } else if (gameState == 1) {
      text("Start", btn1PosX, btn1PosY - buffer2);
    }
```

Figur 5 - Uddrag fra startBTN() funktionen

De næste funktion man møder er exitBTN() og backBTN() som laver knapper placeret under hhv. "Play" knappen på titelskærmen og "Start" knappen på options skærmen. Den første funktion exitBTN() laver en "Exit" knap til titelskærmen som man kan klikke på for at lukke programmet ned, mens backBTN() laver en "Back" knap som fungerer som en omvendt "Play" / "Start" knap, man går altså et skridt tilbage når man klikker på den. Udover forskellen på hvad interaktion med den individuelle knap gør, så fungerer knapperne principielt på samme måde. Det kan ses i eksemplet nedenfor hvor exitBTN() er sat ind.

```
  void exitBTN() {
    //  Draw box for button
    strokeWeight(1);
    stroke(50);
    fill(118, 154, 220);
    rectMode(CENTER);
    rect(btn2PosX, btn2PosY, btnWidth, btnHeight);
    //  Display text for button
    textAlign(CENTER, CENTER);
    fill(50);
    textSize(btnSz);
    text("Exit", btn2PosX, btn2PosY - buffer2);

    //  Calculate interaction with button (Is mouse over button?)
    if (mouseX >= btn2PosX - btnWidth/2 && mouseX <= btn2PosX + btnWidth/2 &&
    mouseY >= btn2PosY - btnHeight/2 && mouseY <= btn2PosY + btnHeight/2) {
      mouseOverBTN = true;
    } else {
      mouseOverBTN = false;
    }

    //  Temp variable makes sure button is only activated once pr. click
    if (!mousePressed) {
      temp = true;
    }

    //  Functionality for when the button is clicked
    //  (Is mouse over button and is the user clicking?)
    if (mouseOverBTN && mousePressed && temp) {
      exit();
    }
  }
```

Figur 6 – exitBTN() funktionen

Den sidste funktion i MenuElements klassen hedder btnGrid() og bruges til at lave tre knapper på options skærmen, som brugerne skal benytte sig af for at vælge den størrelse spilleplade vil have til selve spillet. Her kan der vælges mellem en standard 3x3, en 5x5 og en 7x7 plade. Det skal dog nævnes at man skal have "tre på stribe" for at vinde uanset størrelsen på pladen. Disse knapper blev lavet lidt anderledes, i den forstand at den knap som brugeren har klikket på skifter farve, så de kan se hvilken størrelse spilleplade der er valgt. Som standard er det 3x3 der lyser op, da det er standard spillepladen. Det kan ses nedenfor hvordan dette er opnået. (Dette er igen et udsnit af en større funktion, resten kan ses i bilag.)

```
//  Draws boxes for buttons
rectMode(CENTER);
strokeWeight(1);
stroke(50);
//  If board is 3x3, make the color of the 3x3 button lighter
if(side == 3) {
  fill(140, 180, 255);
} else {  //  Otherwise color it normally
  fill(118, 154, 220);
}
rect(gridBTNPosX + buffer, 2*gridBTNPosY, gridBTNSize, gridBTNSize);
//  If board is 5x5, make the color of the 5x5 button lighter
if(side == 5) {
  fill(140, 180, 255);
} else {  //  Otherwise color it normally
  fill(118, 154, 220);
}
rect(2*gridBTNPosX, 2*gridBTNPosY, gridBTNSize, gridBTNSize);
//  If board is 7x7, make the color of the 7x7 button lighter
if(side == 7) {
  fill(140, 180, 255);
} else {  //  Otherwise color it normally
  fill(118, 154, 220);
}
rect(3*gridBTNPosX - buffer, 2*gridBTNPosY, gridBTNSize, gridBTNSize);

//  Inserts text for buttons
textAlign(CENTER, CENTER);
fill(50);
textSize(btnSz);
text("3x3", gridBTNPosX + buffer, 2*gridBTNPosY-2);
text("5x5", 2*gridBTNPosX, 2*gridBTNPosY-2);
text("7x7", 3*gridBTNPosX - buffer, 2*gridBTNPosY-2);
```

Figur 7- Udsnit af btnGrid() funktionen

Menuen

Herunder ses de to menu skærme som de ser ud i den færdige version af programmet. Jeg mener selv at de er gode fordi de er så simple.

Figur 8 – Titelskærmen og optionsskærmen

Cell.pde

For at spillet overhovedet kan spilles, skal der naturligvis være noget grafisk. Cell.pde klassen står for at tegne selve spillepladen og de brikker som bliver placeret af spillerne. Cell klassen indeholder to funktioner: click() og display(). Den første funktion click() bestemmer hvilken brik som skal tegnes der hvor der bliver klikket, og holder styr på hvem der har lagt brikken samt hvor mange ture der er gået. Funktionen kan ses nedenfor.

```
    //  Function for figuring out where to draw X or O
    void click(int tx, int ty) {
      //  Set mouse position
      int mx = tx;
      int my = ty;

      //  Decides where to draw X or O depending on the mouse position, remem-
bers whose turn it is and keeps track of the turns taken
      if (mx > x && mx < x+w && my > y && my < y+h) {
        if (player == 0 && state == 0) {
          state = 1;
          player = 1;
          turns++;
        } else if (player == 1 && state == 0) {
          state = 2;
          player = 0;
          turns++;
        }
      }
    }
```

Figur 9 - click() funktionen

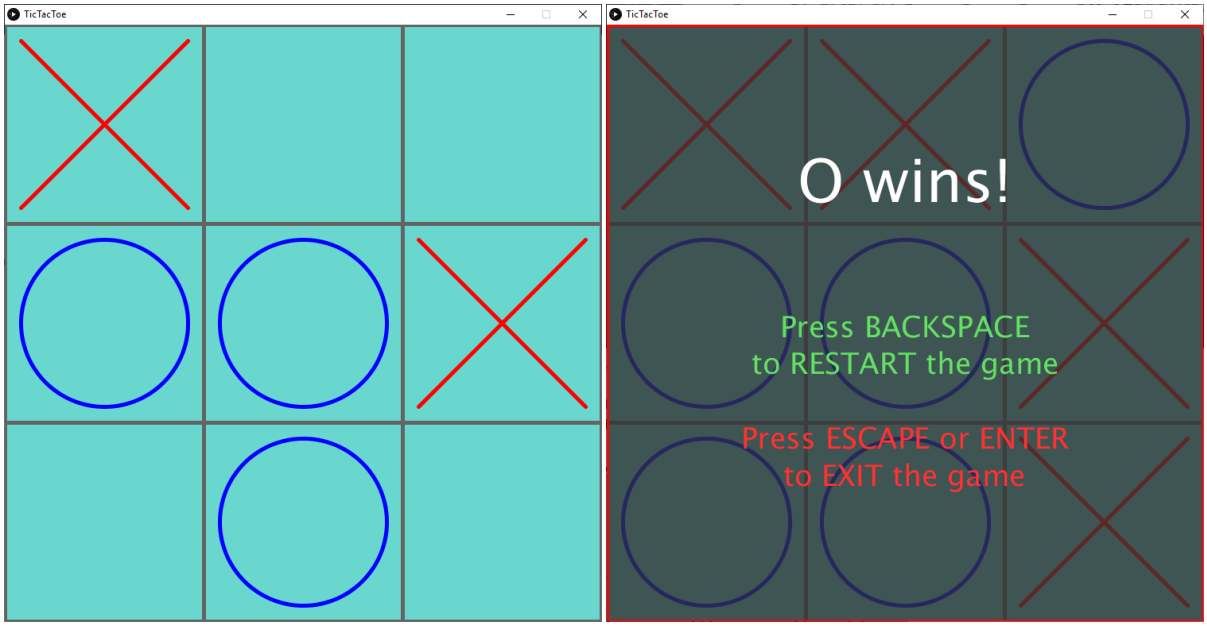
Den anden funktion, altså display() bliver brugt til at tegne selve spillepladen og de brikker der skal tegnes også. Det hele bliver blot tegnet med simpel geometri: firkanter ellipser og linjer.

```
//  Function for displaying the X and O in cells
void display() {
  rectMode(CORNER);
  fill(105, 215, 205);
  stroke(100);
  rect(x, y, w, h);
  strokeWeight(s);

  //  Draw an ellipse for O and a cross for X
  if (state == 1) {
    ellipseMode(CORNER);
    stroke(0, 0, 255);
    ellipse(x+4*s, y+4*s, w-8*s, h-8*s);
  } else if (state == 2) {
    stroke(255, 0, 0);
    line(x+4*s, y+4*s, (x+w)-4*s, (y+h)-4*s);
    line((x+w)-4*s, y+4*s, x+4*s, (y+h)-4*s);
  }
}
```

Figur 10 - display() funktionen

In game

Herunder ses det hvordan spillet ser ud in game og når en har vundet kampen.



Figur 11 - In game og win screen

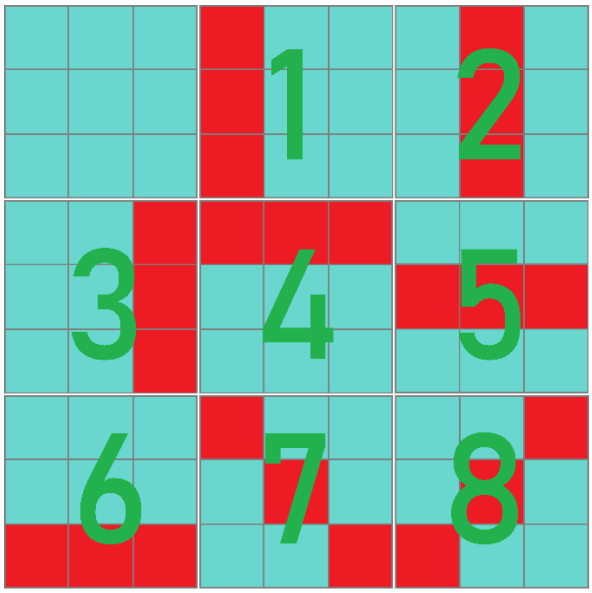
Win3x3.pde, Win5x5.pde & Win7x7.pde

Mens jeg var i gang med at udvikle menuerne i MenuElements, gik Tobias i gang med at gøre det muligt at vinde i spillet for indtil videre kunne man kun placere nogle brikker. – Det er der ikke meget spil over. Tobias fik hurtigt lavet tre statiske funktioner til at beregne på nogle win conditions som bliver sat med en lang række if-statements inde i hver funktion. Et eksempel på det kan ses på næste side. (Udsnit af win3x3 funktionen)

```
int row = 0;
 //  For every column on the board
 for (int col = 0; col< cols; col++) {
   //  Check win conditions on the vertical lines
   if (board[col][row].checkState() == 1 &&
   board[col][row+1].checkState() == 1 &&
   board[col][row+2].checkState() == 1) {
     win = 1;  //  Player 1 wins
   } else if (board[col][row].checkState() == 2 &&
   board[col][row+1].checkState() == 2 &&
   board[col][row+2].checkState() == 2) {
     win = 2;  //  Player 2 wins
   }
```

Figur 12 - Udsnit af Win3x3() funktionen

Denne ovenstående metode tjekker først alle kolonnerne, derefter rækkerne og til sidst diagonalerne. Se illustration nedenfor af hvordan metoden fungerer.



Figur 13 - Illustration af Win3x3()

På et 3x3 board er der 8 win conditions (3 vertikale, 3 horisontale og 2 diagonale), så her er det overskueligt nok at skrive manuelt. Men bare i et 5x5 board er der 39 forskellige win conditions og i 7x7 vil der være mange flere. Så selvom denne løsning er reel nok og den sagtens kan fungere på en 3x3 plade så er den ikke optimal til større plader, da man skal skrive ALLE de mulige win conditions i sine if-statements, hvilket ikke er optimalt. Da jeg var færdig med menu og Tobias havde udviklet de tre funktioner Win3x3(), Win5x5() og Win7x7(), gik jeg i gang med at finde en mere dynamisk og kompakt løsning til at finde win-conditions.

Win.pde

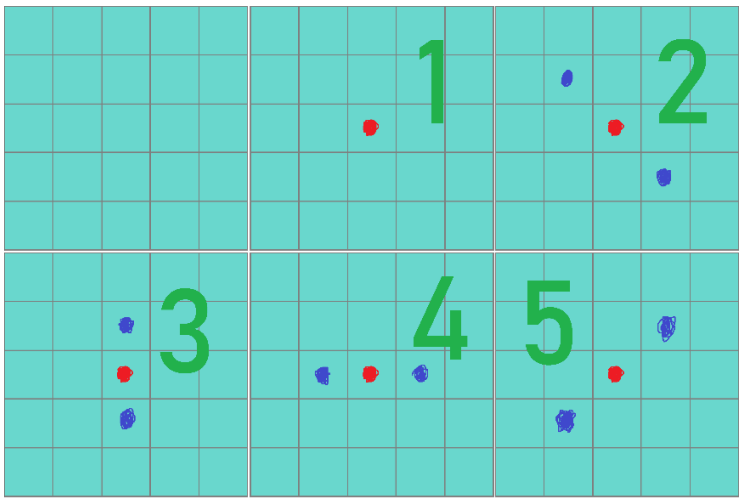
Det jeg havde i tankerne da jeg udviklede win() funktionen var at jeg ville kunne kalde den én gang uden at give den nogen parametre og så ville den kunne tjekke alle felter uanset størrelsen på boardet. Det første jeg gjorde, var at stille to for-loops op, et som kører gennem alle kolonner og inden i det er der et som kører gennem alle rækkerne i hver kolonne. Idéen var så at for alle placeringer i boardet skal win() funktionen tjekke om nogen har vundet, og vise en victory screen hvis nogen vinder. Derudover tjekker den også om spillet ender uafgjort ved at se om pladen er fuld og om der ikke er nogen der har vundet endnu, for så er spillet uafgjort.

```
//  Function for setting win variable
  void win() {

    //  For every row and column, if no one has won, run the win checker func-
tion, and attempt to run the winScreen function.
    for (int row = 0; row < rows; row++) {
      for (int col = 0; col < cols; col++) {
        if (win == 0) {
          win = checkWin(col, row, board[col][row].state);
        }
      }
    }

    if (win == 0 && turns >= cols*rows) {
      tied = true;
    }

    winScreen();
  }
```

Figur 14 - win() funktionen

checkWin()

Den vigtigste ting er dog checkWin() funktionen, da det er den som gør al arbejdet når der tjekkes om nogen har vundet. Idéen med checkWin() er, at for hvert felt i boardet skal den tjekke alle de mulige win conditions der befinder sig rundt om det enkelte felt, og så tjekker den for alle felter på hele spillepladen. En illustration kan ses nedenfor, i eksemplet tjekker den for det midterste felt på en 5x5 plade. (Den røde prik er cellen som checkWin() er blevet bedt om at tjekke rundt om, og de blå prikker er de celler som der tjekkes.)



Figur 15 - Illustration af checkWin()

Funktionen bliver kaldt med tre parametre: Hvilken kolonne den skal tjekke, hvilken række den skal tjekke, og hvilken spiller den skal tjekke for. Så tjekker den ved hjælp af min contain() funktion om de celler den skal tjekke ligger uden for spillepladens array, for så skal den ikke tjekke da der naturligvis ikke kan opfyldes en win condition.

Nedenfor ses et udsnit af checkWin() funktionen, resten kan ses i bilag.

```
  // Function for checking win conditions on any board size,
  // Checks if the given player has put their tile in the set cell (the for
loops above cycles through all cells)
  // it also checks in the adjacent cells (both vertically, horizontally and
diagonally)
  // To counteract an out of bounds error, a contain function was made which
makes sure that we don't check a cell which doesn't exist in the board array
  // Then if a win condition is met, returns the player who won, or return 0 if
no win condition has been met
  int checkWin(int col, int row, int player) {
    if (contain(col-1, side) == true && contain(row-1, side) == true) {
      if (board[col-1][row-1].state == player) {
        if (contain(col+1, side) == true && contain(row+1, side) == true) {
          if (board[col+1][row+1].state == player) {
            return player;
          }
        }
      }
    }
```

Figur 16 - checkWin()

winScreen()

Denne funktion tegner blot den skærm som popper frem når nogen har vundet eller hvis spillet ender uafgjort. Spillet bliver bare sløret og så kommer der tekst op som forklarer hvem der har vundet og hvilke knapper man kan trykke på for at genstarte eller lukke programmet.

Input.pde

Input.pde håndterer mus- og keyboard-inputs, den bruges til at køre click() funktionen fra Cell klassen og lukke eller genstarte spillet når nogen har vundet eller spillet ender uafgjort. Dette kunne i realiteten også sagtens bare stå i den primære kode, men det står i sin egen fane for overskuelighed.

mousePressed()

Funktionen mousePressed() køres hver gang der bliver klikket med musen. Hvis der ikke er nogen der har vundet og der ikke står uafgjort, køres click() med musens position på alle celler.

keyPressed()

Denne funktion ved navn keyPressed() bliver kørt hver gang der bliver trykket på en knap på tastaturet. Hvis spillet er aktivt og der er nogen der har vundet eller det er endt uafgjort, så kan man afslutte programmet med ENTER eller ESC, mens man kan genstarte programmet med BACKSPACE knappen. Dette gøres med exit() og reset().

reset()

Reset er en funktion lavet til at genstarte hele programmet og gå tilbage til titelmenuen, hvorefter brugerne på ny kan vælge størrelse på spillepladen og så spille igen. Funktionen starter med at tegne baggrunden igen, genskabe menu objektet, den tømmer spillepladen ved at fylde

dens array med nuller og så redefinerer den alle de samme værdier som blev defineret lige før setup() funktionen. Dette gør at programmet kan starte helt forfra uden at brugeren skal til at genstarte det, som man godt kan komme til mange gange hvis man sidder og spiller kryds og bolle med en ven, hvilket ville ødelægge ens flow fuldstændigt.

```
//  Reset the entire game, can be run when the game ends.
  void reset() {
    background(105, 215, 205);
    menu = new MenuElements();
    for (int i = 0; i < cols; i++) {
      for (int j = 0; j < rows; j++) {
        board[i][j] = new Cell(0, 0, 0, 0, 0);
      }
    }
    cols = 3;
    rows = 3;
    side = 3;
    gameState = 0;
    player = 0;
    win = 0;
    turns = 0;
    executed = false;
  }
```

Figur 17 - reset()

Test

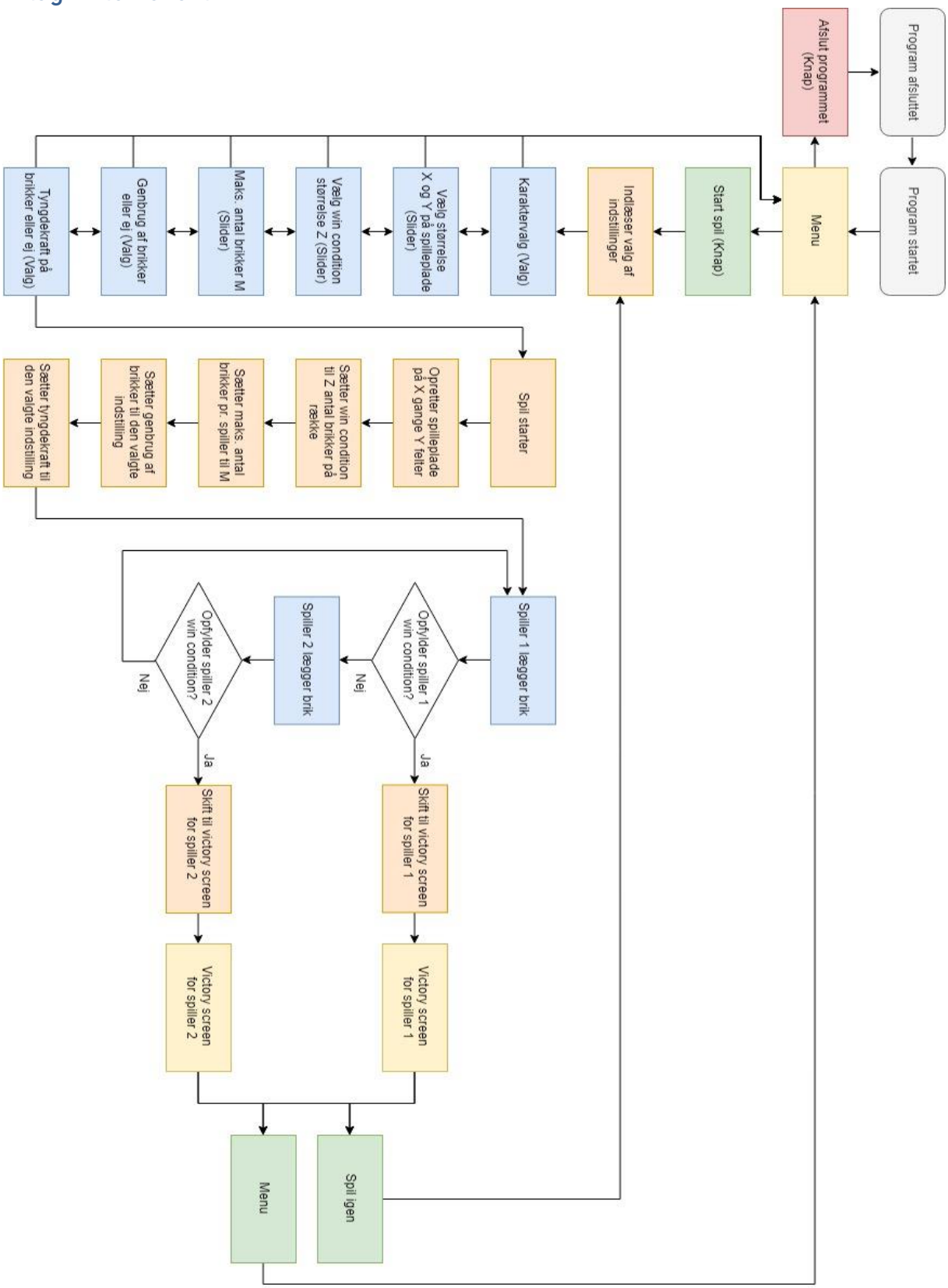
Der blev løbende kørt test under udviklingen af programmet for at forsikre os om at det kode vi skrev faktisk virkede. Kun meget få gange stødte vi på problemer, og det var primært under udviklingen af det funktioner som skal tjekke om nogen har vundet og om spillet står uafgjort. Jeg testede løbende menuen så jeg vidste at den virkede når vi skulle samle vores kode, Tobias testede løbende selve spillet hvor man kunne sætte X'er og O'er. Da vi havde samlet vores kode, testede vi også hele tiden om de ting vi tilføjede, faktisk virkede, hvilket de som regel gjorde.

Konklusion

Jeg vil mene at i forhold til opgaven Tobias og jeg blev stillet og de mål vi stillede for os selv, har vi løst opgaven til meget stor tilfredsstillelse. Under udviklingen tænkte vi hele tiden på at programmet skulle virke så intuitivt som muligt, så vi har med vilje skaleret alting større end vi umiddelbart havde tænkt at gøre. Udover det har vi under tests ikke fundet nogen fejl i programmet, hvilket blot forhøjer tilfredsstillelsen af vores løsning på opgaven.

Bilag

Bilag 1: Flowchart



Bilag 2: startBTN()

```
    //  Drawing and functionality for the Play button
    void startBTN() {
      //  Draws box for button
      strokeWeight(1);
      stroke(50);
      fill(118, 154, 220);
      rectMode(CENTER);
      rect(btn1PosX, btn1PosY, btnWidth, btnHeight);

      //  Inserts text for button
      textAlign(CENTER, CENTER);
      fill(50);
      textSize(btnSz);
      if(gameState == 0) {
        text("Play", btn1PosX, btn1PosY - buffer2);
      } else if (gameState == 1) {
        text("Start", btn1PosX, btn1PosY - buffer2);
      }

      //  Calculates button area to get pressed
      if (mouseX >= btn1PosX - btnWidth/2 &&
      mouseX <= btn1PosX + btnWidth/2 &&
      mouseY >= btn1PosY - btnHeight/2 &&
      mouseY <= btn1PosY + btnHeight/2) {
        mouseOverBTN = true;
      } else {
        mouseOverBTN = false;
      }

      //  If the mouse isn't pressed, set temp to true so the button can be
pressed
      if (!mousePressed) {
        temp = true;
      }

      //  Adds 1 to gameState if the button is pressed and sets temp to make
the button only work once every click
      if (mouseOverBTN && mousePressed && temp) {
        gameState++;
        temp = false;
      }
    }
```

Bilag 3: btnGrid()

```
void btnGrid() {  //draws grid for selection of board size
     //  Draws boxes for buttons
      rectMode(CENTER);
      strokeWeight(1);
      stroke(50);

      if(side == 3) {
        fill(140, 180, 255);
      } else {
        fill(118, 154, 220);
      }
      rect(gridBTNPosX + buffer, 2*gridBTNPosY, gridBTNSize, gridBTNSize);

      if(side == 5) {
        fill(140, 180, 255);
      } else {
        fill(118, 154, 220);
      }
      rect(2*gridBTNPosX, 2*gridBTNPosY, gridBTNSize, gridBTNSize);

      if(side == 7) {
        fill(140, 180, 255);
      } else {
        fill(118, 154, 220);
      }
      rect(3*gridBTNPosX - buffer, 2*gridBTNPosY, gridBTNSize, gridBTNSize);

      //  Inserts text for buttons
      textAlign(CENTER, CENTER);
      fill(50);
      textSize(btnSz);
      text("3x3", gridBTNPosX + buffer, 2*gridBTNPosY-2);
      text("5x5", 2*gridBTNPosX, 2*gridBTNPosY-2);
      text("7x7", 3*gridBTNPosX - buffer, 2*gridBTNPosY-2);

      //  Calculates button area to get pressed for button 1
      if (mouseX >= gridBTNPosX + buffer - gridBTNSize/2 &&
      mouseX <= gridBTNPosX + buffer + gridBTNSize/2 &&
      mouseY >= 2*gridBTNPosY - gridBTNSize/2 &&
      mouseY <= 2*gridBTNPosY + gridBTNSize/2) {
        mouseOverBTN = true;
      } else {
        mouseOverBTN = false;
      }

      //  If the mouse isn't pressed, set temp to true so the button can be
pressed
      if (!mousePressed) {
        temp = true;
      }

      //  Sets side to 3 if the button is pressed and sets temp to make the
button only work once every click
      if (mouseOverBTN && mousePressed && temp) {
        side = 3;
        temp = false;
      }
```

```
      //  If the mouse isn't pressed, set temp to true so the button can be
pressed
      if (!mousePressed) {
        temp = true;
      }

      //  Sets side to 3 if the button is pressed and sets temp to make the
button only work once every click
      if (mouseOverBTN && mousePressed && temp) {
        side = 3;
        temp = false;
      }

      //  Calculates button area to get pressed for button 2
      if (mouseX >= 2*gridBTNPosX - gridBTNSize/2 &&
      mouseX <= 2*gridBTNPosX + gridBTNSize/2 &&
      mouseY >= 2*gridBTNPosY - gridBTNSize/2 &&
      mouseY <= 2*gridBTNPosY + gridBTNSize/2) {
        mouseOverBTN = true;
      } else {
        mouseOverBTN = false;
      }

      //  Sets side to 5 if the button is pressed and sets temp to make the
button only work once every click
      if (mouseOverBTN && mousePressed && temp) {
        side = 5;
        temp = false;
      }

      //  Calculates button area to get pressed for button 3
      if (mouseX >= 3*gridBTNPosX - buffer - gridBTNSize/2 &&
      mouseX <= 3*gridBTNPosX - buffer + gridBTNSize/2 &&
      mouseY >= 2*gridBTNPosY - gridBTNSize/2 &&
      mouseY <= 2*gridBTNPosY + gridBTNSize/2) {
        mouseOverBTN = true;
      } else {
        mouseOverBTN = false;
      }

      //  Sets side to 7 if the button is pressed and sets temp to make the
button only work once every click
      if (mouseOverBTN && mousePressed && temp) {
        side = 7;
        temp = false;
      }
    }
```

Bilag 5: checkWin()

```
  // Function for checking win conditions on any board size,
  // Checks if the given player has put their tile in the set cell (the for
loops above cycles through all cells)
  // it also checks in the adjacent cells (both vertically, horizontally and
diagonally)
  // To counteract an out of bounds error, a contain function was made which
makes sure that we don't check a cell which doesn't exist in the board array
  // Then if a win condition is met, returns the player who won, or return 0 if
no win condition has been met
  int checkWin(int col, int row, int player) {
    if (contain(col-1, side) == true && contain(row-1, side) == true) {
      if (board[col-1][row-1].state == player) {
        if (contain(col+1, side) == true && contain(row+1, side) == true) {
          if (board[col+1][row+1].state == player) {
            return player;
          }
        }
      }
    }

    if (contain(row-1, side) == true) {
      if (board[col][row-1].state == player) {
        if (contain(row+1, side) == true) {
          if (board[col][row+1].state == player) {
            return player;
          }
        }
      }
    }

    if (contain(col-1, side) == true) {
      if (board[col-1][row].state == player) {
        if (contain(col+1, side) == true) {
          if (board[col+1][row].state == player) {
            return player;
          }
        }
      }
    }

    if (contain(col-1, side) == true && contain(row+1, side) == true) {
      if (board[col-1][row+1].state == player) {
        if (contain(col+1, side) == true && contain(row-1, side) == true) {
          if (board[col+1][row-1].state == player) {
            return player;
          }
        }
      }
    }
    return 0;
  }
```