Sudoku GUI 1

Final

3.1 Model a Sudoku Game

To implement my sudoku solving algorithm from P1 into a GUI I first created a new class Sudoku and imported all of my code from P1 into there. To easily get and set values from the sudoku grid in the Sudoku class and the GUI, I created functions in the Sudoku class which allowed this.

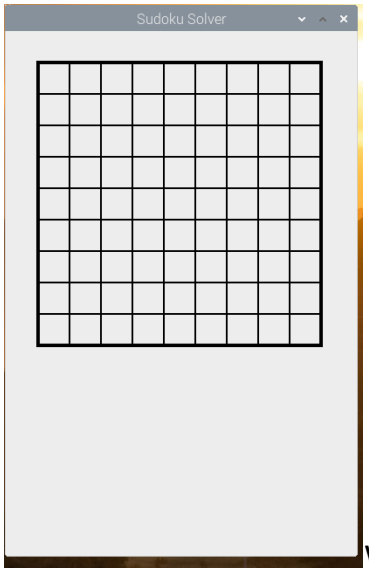
```
class Sudoku {
public:
    Sudoku() {

    }

    int solveSudoku();

    void fileToGrid(const char * fileName);
    void displayBoard(void);

    int getCell(int row, int column);

    bool isSolved(int row, int column);
    void handleSolution(int row, int column);

    bool solveGrid(void);

    void deleteFromRow(int row, int num);
    void deleteFromColumn(int column, int num);
    void deleteFromSubcell(int row, int column, int num);


private:
    cell sudokuGrid[GRID_SIZE][GRID_SIZE];
    bool zerError = false;
};

#endif
```

To test to see if all these functions are working, I added an instance of Sudoku in the window file and just read a sudoku from a file and printed it in the command line.

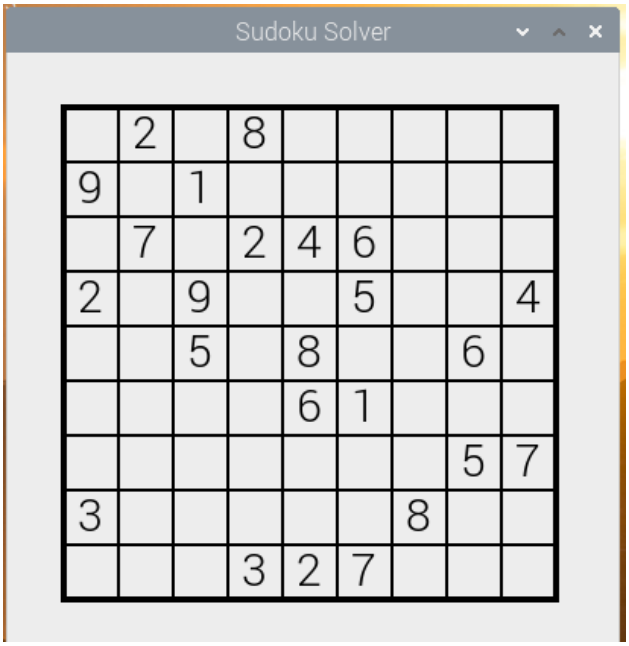
3.2 Display the Model

In order to make the size of my sudoku scalable, I used the size function in order to get the size of the windows I am drawing to. I then used the smallest side (either height or width) to then be the basis for the size of the square sudoku board. To add padding either side, I divided this size by 11 which allows for 1 blank square either side of the board.

Therefore, I wrote code which first draws the outline of the board with pen width 4, then in a for loop draws the dividing lines between each cell. The resulting graphics is shown below.



With the lines being drawn as expected, I then had to read the values of the grid from the Sudoku class and draw those numbers on the screen.



4 Responding to User Input

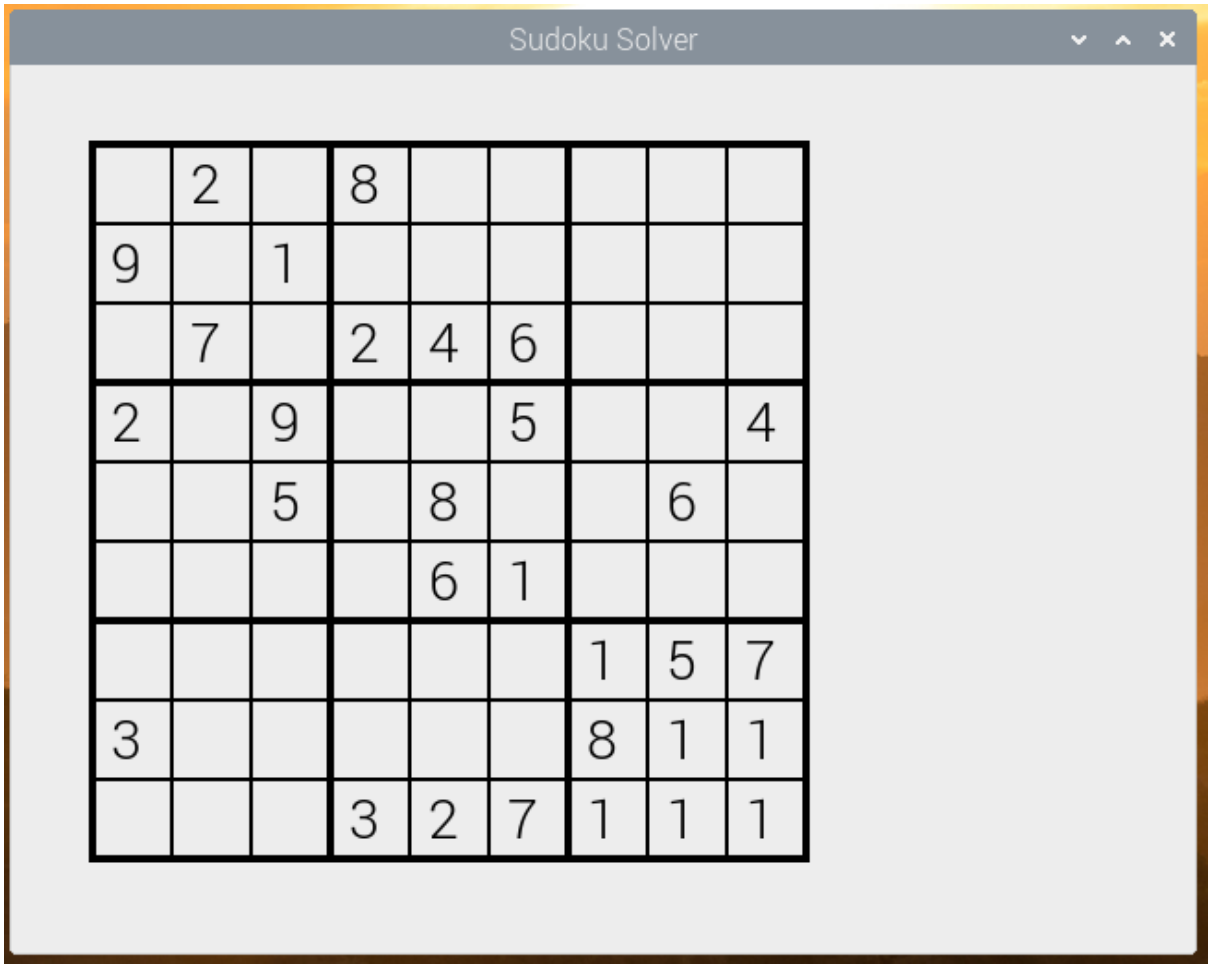
In order to implement user input, I had to override the mouseReleaseEvent function. In this function I took the mouse x, y coordinates, did a bound check to check that the mouse was in the grid size and then work out the x, y of the cell which the user is clicking in.

```
Screen Size: 480, 640
Screen Size: 480, 640
Screen Size: 480, 640
Screen Size: 480, 640
Mouse in cell  0 , 0
Mouse in cell  1 , 0
Mouse in cell  2 , 0
Mouse in cell  3 , 0
Mouse in cell  4 , 0
Mouse in cell  4 , 1
Mouse in cell  4 , 2
Mouse in cell  4 , 3
Mouse in cell  4 , 4
Mouse in cell  4 , 5
Mouse in cell  4 , 6
Mouse in cell  4 , 7
Mouse in cell  4 , 8
Mouse in cell  8 , 8
```

```
QSize screenSize = size();

if(screenSize.rheight() < screenSize.rwidth()) {
    cellPixelSize = (int) screenSize.rheight() / 11;
}
else {
    cellPixelSize = (int) screenSize.rwidth() / 11;
}

winMin = cellPixelSize;
winMax = (int) cellPixelSize * 10;


mouseX = (int) event->x();
mouseY = (int) event->y();


if((winMin < mouseX) && (mouseX < winMax) && (winMin < mouseY) && (mouseY < winMax)) {


    cellX = (mouseX / cellPixelSize) - 1;
    cellY = (mouseY / cellPixelSize) - 1;

    qDebug() << "Mouse in cell " << cellX << "," << cellY;
}
```

Finally, in order to respond to the users input, it reads the number from the Sudoku class then increments it and sets it as the new value. This works, allowing the user to cycle through the number on any cell in the grid.

Also, I include on cosmetic tweak which is to add thicker lines between the 3x3 sub cells allowing the user to read the cells easier.


Sudoku Solver
2 8
9 1
7 2 4 6
2 9 5 4
5 8 6
6 1
1 5 7
3 8 1 1
3 2 7 1 1 1

Final Code

```
void Window::paintEvent(QPaintEvent * event) {
    int boardPixelSize = 0;
    int cellPixelSize = 0;

    QSize screenSize = size();

    QPainter painter(this);
    QPen pen;
    QFont font;

    font.setPixelSize(30);
    pen.setWidth(4);
    painter.setPen(pen);
    painter.setFont(font);

    cout << "Screen Size: " << screenSize.rheight() << ", " << screenSize.rwidth()
 << endl;

    //takes the smallest side (height, width) and makes it the size of the square
    if(screenSize.rheight() < screenSize.rwidth()) {
        boardPixelSize = screenSize.rheight();
    }
    else {
        boardPixelSize = screenSize.rwidth();
    }

    cellPixelSize = (int) boardPixelSize / 11;



    //draw board outline
    painter.drawLine(QPoint(cellPixelSize, cellPixelSize), QPoint(cellPixelSize *1
0, cellPixelSize));                    //horizontal top bar
    painter.drawLine(QPoint(cellPixelSize, cellPixelSize * 10), QPoint(cellPixelSi
ze *10, cellPixelSize * 10));          //horizontal bottom bar


    painter.drawLine(QPoint(cellPixelSize, cellPixelSize), QPoint(cellPixelSize, c
ellPixelSize * 10));                   //vertical left bar
    painter.drawLine(QPoint(cellPixelSize * 10, cellPixelSize), QPoint(cellPixelSi
ze *10, cellPixelSize * 10));          //vertical right bar



    //draw cell divider lines
    painter.drawLine(QPoint(cellPixelSize, cellPixelSize * 4), QPoint(cellPixelSiz
e * 10, cellPixelSize * 4));           //draw horizontal lines
```

```
    painter.drawLine(QPoint(cellPixelSize, cellPixelSize * 7), QPoint(cellPixelSiz
e * 10, cellPixelSize * 7));

    painter.drawLine(QPoint(cellPixelSize * 4, cellPixelSize), QPoint(cellPixelSiz
e * 4, cellPixelSize * 10));          //draw vertical lines
    painter.drawLine(QPoint(cellPixelSize * 7, cellPixelSize), QPoint(cellPixelSiz
e * 7, cellPixelSize * 10));



    //draw square boardSize / 11 - 1 square blank each side
    pen.setWidth(2);
    painter.setPen(pen);

    for(int i=2; i<10; i++) {
        //draw horizontal line

        painter.drawLine(cellPixelSize, cellPixelSize * i, cellPixelSize * 10, cel
lPixelSize * i);                      //draw horizontal grid lines

        painter.drawLine(cellPixelSize * i, cellPixelSize, cellPixelSize * i, cell
PixelSize * 10);                      //draw vertical grid lines


    }


    //draw the numbers in the squares just created

    //increment row (y)
    for(int i=0; i<9; i++) {

        //increment column (x)
        for(int j=0; j<9; j++) {
            int num = sud.getCell(i, j);

            if(num != 0) {
                //draw number to grid
                painter.drawText((1.25 * cellPixelSize) + (j * cellPixelSize), (1.
75 * cellPixelSize) + (i * cellPixelSize), QString().setNum(num));
            }
        }
    }
}

void Window::mouseReleaseEvent(QMouseEvent * event)
{
    // get click position
    //qDebug() << "Mouse x " << event->x() << " Mouse y " << event->y();
```

```
    int winMin, winMax, cellPixelSize;
    int mouseX, mouseY;
    int cellX, cellY;


    //get cell and board size to work out mouse clicks
    QSize screenSize = size();

    if(screenSize.rheight() < screenSize.rwidth()) {
        cellPixelSize = (int) screenSize.rheight() / 11;
    }
    else {
        cellPixelSize = (int) screenSize.rwidth() / 11;
    }

    winMin = cellPixelSize;
    winMax = (int) cellPixelSize * 10;


    mouseX = (int) event->x();
    mouseY = (int) event->y();


    if((winMin < mouseX) && (mouseX < winMax) && (winMin < mouseY) && (mouseY < wi
nMax)) {


        cellX = (mouseX / cellPixelSize) - 1;
        cellY = (mouseY / cellPixelSize) - 1;

        qDebug() << "Mouse in cell " << cellX << "," << cellY;


        //row (y), collumn(x)
        int num = sud.getCell(cellY, cellX);

        if(num > 8) {
            num = 0;
        }
        else {
            num++;
        }

        qDebug() << "New Num: " << num;

        sud.setCell(cellY, cellX, (int)num);

        update();
```

```
    }

}
```