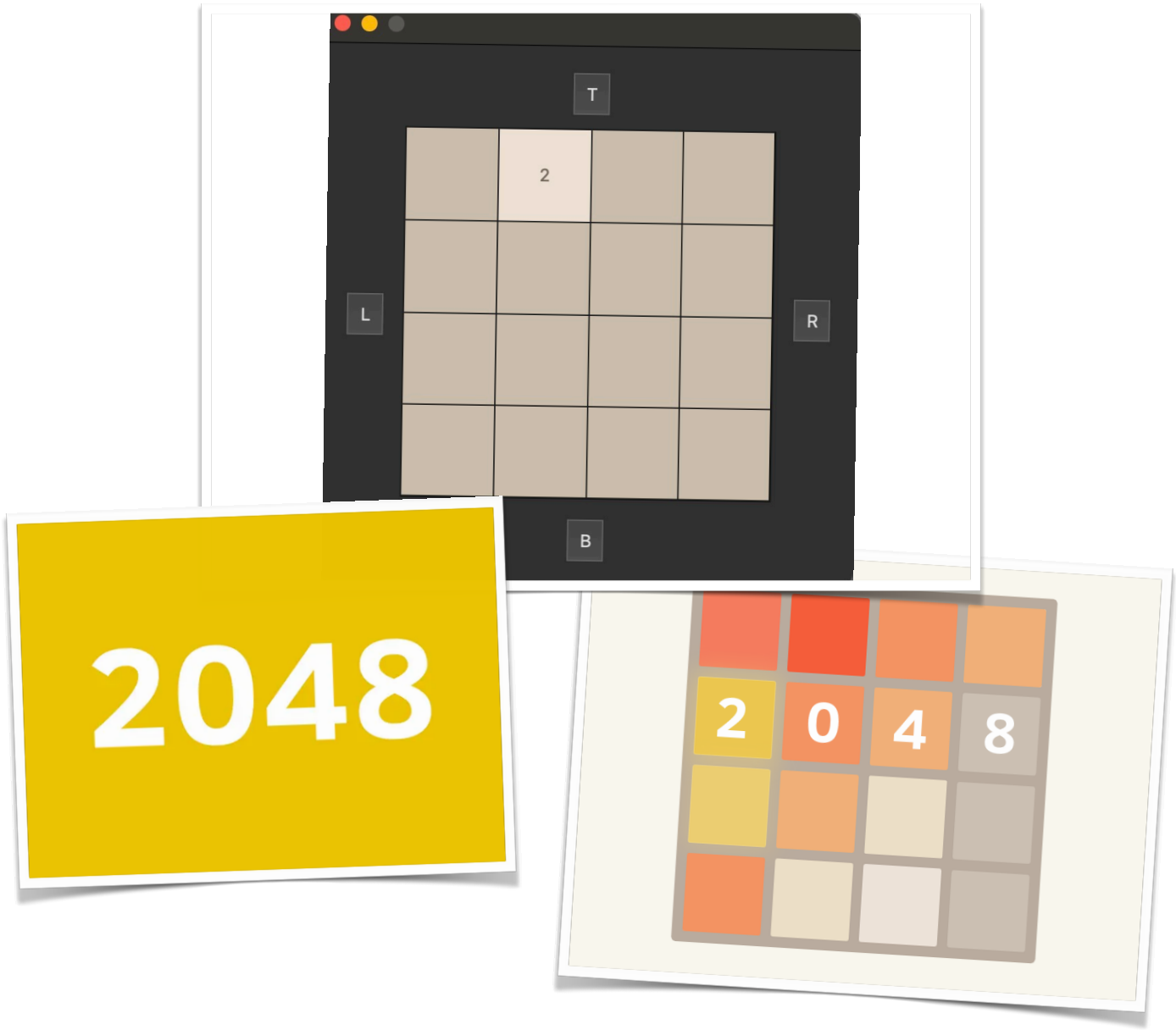
KAMEL Tarek
ABDALLAOUI Mohamed

# 2048 Game

## With Qt MVC




Supervised by Pr .BELCAID Anass

KAMEL Tarek
ABDALLAOUI Mohamed

# Table of content



Supervised by Pr .BELCAID Anass

KAMEL Tarek
ABDALLAOUI Mohamed

# 1 - Introduction :

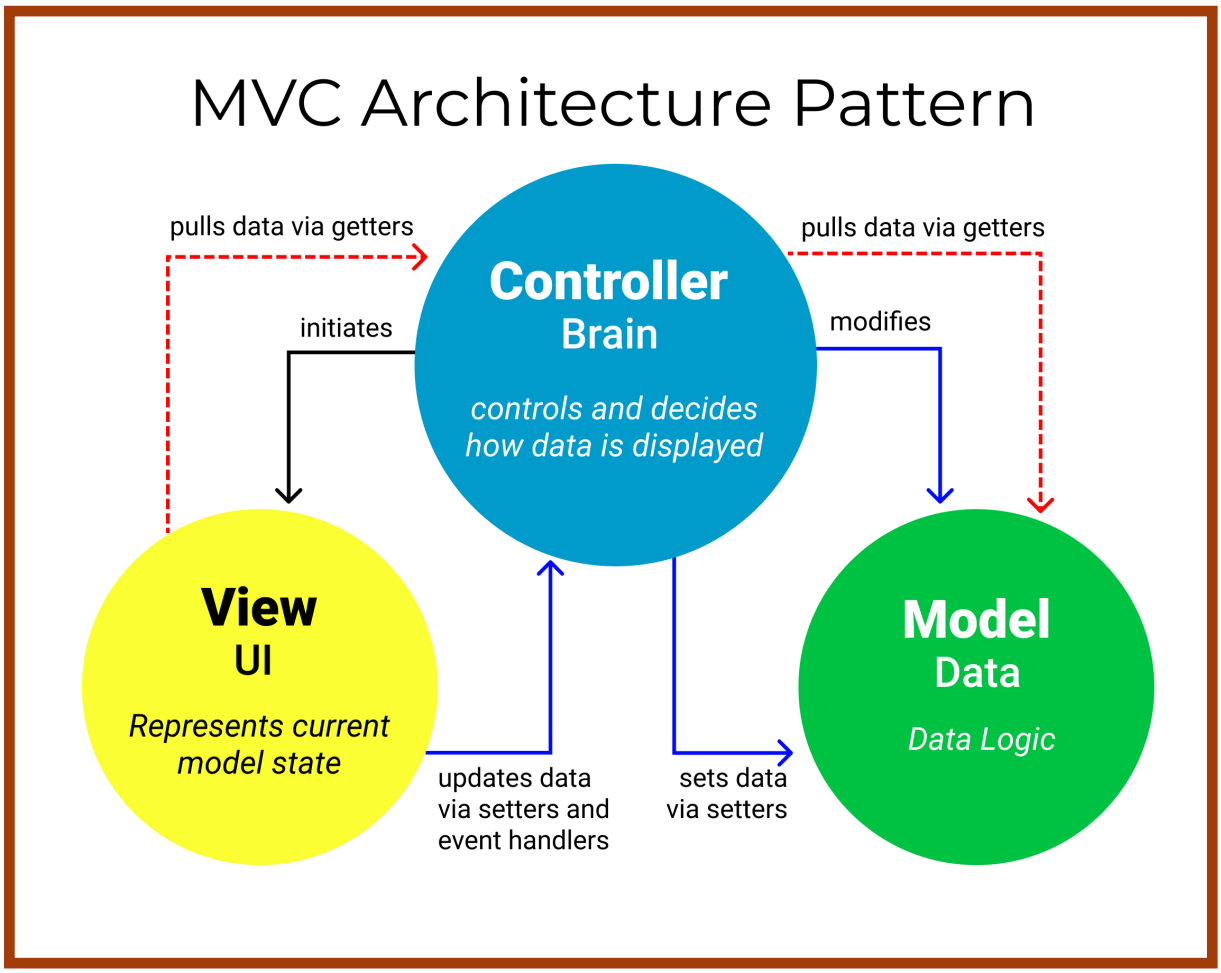
***2048*** is a single-player sliding tile puzzle video game. The objective of the game is to slide numbered tiles on a grid to combine them and finally create a tile with the number 2048; however, one can continue to play the game after reaching the goal, creating tiles with larger numbers. It was originally written in JavaScript and CSS, and released on 9 March 2014 as free and open-source software subject to the.

For the final project we have chosen this game because of its complexity and the challenge it represents for us as beginners with Qt GUI programming, It was also an occasion for us to apply what we have learn with MVC programming.

*2048* is played on a plain 4×4 grid, with numbered tiles that slide when a player moves them using the four arrow keys. Every turn, a new tile randomly appears in an empty spot on the board with a value of either 2 or 4.
Tiles slide as far as possible in the chosen direction until they are stopped by either another tile or the edge of the grid. If two tiles of the same number collide while moving, they will merge into a tile with the total value of the two tiles that collided. The resulting tile cannot merge with another tile again in the same move.
If a move causes three consecutive tiles of the same value to slide together, only the two tiles farthest along the direction of motion will combine. If all four spaces in a row or column are filled with tiles of the same value, a move parallel to that row/column will combine the first two and last two. The user's score starts at zero, and is increased whenever two tiles combine, by the value of the new tile.
The game is won when a tile with a value of 2048 appears on the board. Players can continue beyond that to reach higher scores. When the player has no legal moves (there are no empty spaces and no adjacent tiles with the same value), the game ends

Supervised by Pr .BELCAID Anass

# 2 - Methods Overview (MVC architecture)




## Classes :

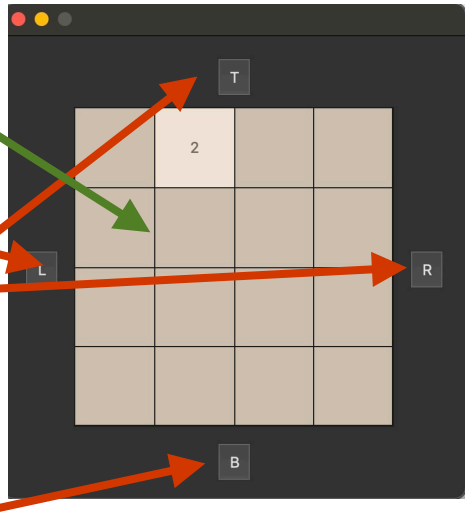
To initialise our project we needed first to create two classes representing our view (MainWindow class) and our model (dataModel class)

## MainWindow Class (View):

Let's dive deeper into Mainwindow class that basically creates our mainwindow interface :

This time we didn't use the Ui feature offered by Qt, and we creates the window from scratch

```
MainWindow::MainWindow(QWidget* parent)
    : QWidget(parent)
{
    setFixedSize(400, 400);

    QGridLayout* layout = new QGridLayout;

    tableview = new QTableView;
    // setting all rows and columns to the same size
    tableview->horizontalHeader()->setSectionResizeMode(QHeaderView::Stretch);
    tableview->verticalHeader()->setSectionResizeMode(QHeaderView::Stretch);
    // hiding headers
    tableview->horizontalHeader()->hide();
    tableview->verticalHeader()->hide();
    layout->addWidget(tableview, 1, 1);
    // 2048 data model
    model = new DataModel;
    tableview->setModel(model);
    // connecting failed signal to lose slot
    connect(model, &DataModel::lost, this, &MainWindow::lose);

    QPushButton* leftButton = new QPushButton("L");
    leftButton->setFixedSize(40, 40);
    leftButton->setShortcut(QKeySequence(Qt::Key_Left));
    layout->addWidget(leftButton, 1, 0, Qt::AlignCenter);

    QPushButton* rightButton = new QPushButton("R");
    rightButton->setFixedSize(40, 40);
    rightButton->setShortcut(QKeySequence(Qt::Key_Right));
    layout->addWidget(rightButton, 1, 2, Qt::AlignCenter);

    QPushButton* topButton = new QPushButton("T");
    topButton->setFixedSize(40, 40);
    topButton->setShortcut(QKeySequence(Qt::Key_Up));
    layout->addWidget(topButton, 0, 1, Qt::AlignCenter);

    QPushButton* bottomButton = new QPushButton("B");
    bottomButton->setFixedSize(40, 40);
    bottomButton->setShortcut(QKeySequence(Qt::Key_Down));
    layout->addWidget(bottomButton, 2, 1, Qt::AlignCenter);
```



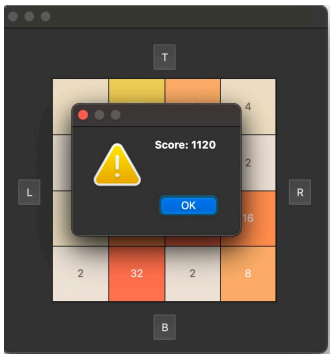

In addition to the widgets this class constructor links the view (QTableView) with the model we will going to be explaining further in the report

Now let's get familiar with the connections we made for our game to work properly

```
    // connecting buttons to their respective movements
    connect(leftButton,   &QPushButton::clicked, model, &DataModel::left);
    connect(rightButton,  &QPushButton::clicked, model, &DataModel::right);
    connect(topButton,    &QPushButton::clicked, model, &DataModel::top);
    connect(bottomButton, &QPushButton::clicked, model, &DataModel::bottom);

   this->setLayout(layout);
}
```

Notably left, right, top and button to slots we implemented in DataModel class that enable the tiles motion to the corresponding direction.

```
void MainWindow::lose()
{
    QMessageBox::warning(this, "End of game", "Score: " + QString::number(model->getScore()));
    qApp->quit();
}
```




Then we created an additional method to display a warning message to inform the user of his failure in the game and displays the final score Calling a method proper to dataModel getScore(), then closes the application when ok button pressed.

## dataModel Class: inheriting from QAbstractItemModel class

## Let's start with the constructor :

We first create a matrix in the .h file called Matrix with is basically a 2 dimension integers vector of 4x4 size, then we call for another method that adds either 2 or 4 values to a random empty cell as the game rules stipulate

```
DataModel::DataModel(QObject* parent)
{
    // creating matrix filled with 0 (by default int constructor)
    matrix.resize(4);
    for (int y = 0; y < matrix.size(); ++y)
        matrix[y].resize(4);

    // adding a first cell filled with a 2
    addInEmptyCell();
}
```

Going along, we declared some getters

Returning matrix width, Height and score values

```
int DataModel::rowCount(const QModelIndex& parent) const
{
    // we return the matrix height
    return matrix.size();
}

int DataModel::columnCount(const QModelIndex& parent) const
{
    // we return the matrix width
    return matrix.size() > 0 ? matrix[0].size() : 0;
}
```

```
int DataModel::getScore() const
{
    return this->score;
}
```

The following method represents the DELEGATE

Qt doesn't quite use an MVC architecture. It's no longer a C but now a D for...Delegate: Qt actually uses a model/view architecture framed by a delegate.

Unlike a classic MVC architecture, the model/view architecture introduced by Qt does not provide a real component to manage interactions with the user. Therefore, this is handled by the view or the model itself.

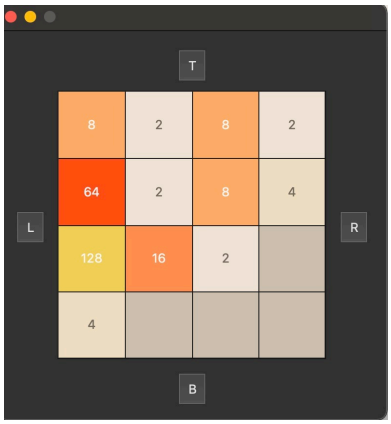
However, for reasons of flexibility, user interaction and input are not taken into account by a totally separate component, namely the controller, but by an internal component: the Delegate. This component is responsible for two things:

- Customize the editing of elements;
- Customize the rendering of elements inside a view.

```
QVariant DataModel::data(const QModelIndex& index, int role) const
{
    // if the index is not valid, we stop here
    if (!index.isValid())
        return QVariant();

    // if the view is asking for the text to display
    if (role == Qt::DisplayRole)
    {
        // if the cell is valid
        if (index.row() < rowCount() && index.column() < columnCount())
        {
            // if the cell number is higher than 0 (which means "not empty")
            // we return its value otherwise we return an empty string (instead of 0)
            if (matrix[index.row()][index.column()] > 0)
                return matrix[index.row()][index.column()];
            else
                QString();
        }
        return QVariant();
    }

    // if the view is asking for the text alignment
    // we return AlignCenter for every cell
    if (role == Qt::TextAlignmentRole)
        return Qt::AlignCenter;
```

KAMEL Tarek
ABDALLAOUI Mohamed

```
    // if the view is asking for the background color of the cell
    // we return colors based on the official 2048 game
    if (role == Qt::BackgroundRole)
    {
        switch (matrix[index.row()][index.column()])
        {
        case 0:
            return QColor("#ccc0b3");
        case 2:
            return QColor("#eee4da");
        case 4:
            return QColor("#ede0c8");
        case 8:
            return QColor("#f2b179");
        case 16:
            return QColor("#f59563");
        case 32:
            return QColor("#f67c5f");
        case 64:
            return QColor("#f65e3b");
        case 128:
            return QColor("#edcf72");
        case 256:
            return QColor("#edcc61");
        case 512:
            return QColor("#edc850");
        case 1024:
            return QColor("#edc53f");
        case 2048:
            return QColor("#edc22e");
        default:
            return QColor("#3e3933");
        }
    }

    // if the view is asking for the text color of the cell
    // we return colors based on the official 2048 game
    if (role == Qt::ForegroundRole)
    {
        switch (matrix[index.row()][index.column()])
        {
        case 2:
        case 4:
            return QColor("#776e65");
        default:
            return QColor("#f9f6f2");
        }
    }

    return QVariant();
}
```




This delegate sets the colors for each tile background and text depending on their values based on the 2048 original colors like so.

Supervised by Pr .BELCAID Anass

Then we defined 2 other methods in order to get the ModelIndex of the parent and another one creating an index at a given position

```
QModelIndex DataModel::index(int row, int column, const QModelIndex& parent) const
{
    return createIndex(row, column);
}

QModelIndex DataModel::parent(const QModelIndex& index) const
{
    return QModelIndex();
}
```

Then the left, right  methods that executes moveLeft and moveRight functions explained further, checks if nothing changed in our matrix or emits a signal to inform the view that data has changed,

Finally it adds either 2 or 4 values to a random empty cell as for the beginning of the game using addEmptyCell().

```
void DataModel::left()
{
    QVector<QVector<int>> temp = matrix;

    // move left
    moveLeft();

    // nothing changed so we stop as the movement is "invalid"
    if (matrix == temp)
    {
        checkIfDead();
        return;
    }

    // emiting signal for the view to know that data changed
    emit dataChanged(index(0, 0), index(rowCount() - 1, columnCount() - 1));

    // adding a cell filled with a 2
    addInEmptyCell();
}
```

```
void DataModel::right()
{
    QVector<QVector<int>> temp = matrix;

    // move right
    moveRight();

    // nothing changed so we stop as the movement is "invalid"
    if (matrix == temp)
    {
        checkIfDead();
        return;
    }

    // emiting signal for the view to know that data changed
    emit dataChanged(index(0, 0), index(rowCount() - 1, columnCount() - 1));

    // adding a cell filled with a 2
    addInEmptyCell();
}
```

Now let's explain the moveRight() and moveLeft() methods used previously

```
void DataModel::moveRight()
{
    // removing all empty cell in each row
    for (int y = 0; y < matrix.size(); ++y)
        matrix[y].removeAll({0});

    // merge
    for (int y = 0; y < matrix.size(); ++y)
    {
        for (int x = matrix[y].size() - 1; x > 0; --x)
        {
            // if the cell is the same as the previous one (without the empty cells removed previously)
            if (matrix[y][x] == matrix[y][x - 1])
            {
                // we merge both cells into the current one
                matrix[y][x] = matrix[y][x] + matrix[y][x - 1];
                // adding sum to score
                score += matrix[y][x];
                // we set the previous one to 0
                matrix[y][x - 1] = 0;
                // we skip the previous one
                --x;
            }
            else
            {
                matrix[y][x] = matrix[y][x];
            }
        }
    }

    for (int y = 0; y < matrix.size(); ++y)
    {
        // removing all empty cell in each row
        matrix[y].removeAll({0});
        // prepending empty cells to fill the matrix
        for (int i = matrix[y].size(); i < 4; ++i)
            matrix[y].prepend(0);
    }
}

void DataModel::moveLeft()
{
    // removing all empty cell in each row
    for (int y = 0; y < matrix.size(); ++y)
        matrix[y].removeAll({0});

    // merge
    for (int y = 0; y < matrix.size(); ++y)
    {
        for (int x = 0; x < matrix[y].size() - 1; ++x)
        {
            // if the cell is the same as the next one (without the empty cells removed previously)
            if (matrix[y][x] == matrix[y][x + 1])
            {
                // we merge both cells into the current one
                matrix[y][x] = matrix[y][x] + matrix[y][x + 1];
                // adding sum to score
                score += matrix[y][x];
                // we set the next one to 0
                matrix[y][x + 1] = 0;
                // we skip the next one
                ++x;
            }
            else
            {
                matrix[y][x] = matrix[y][x];
            }
        }
    }

    for (int y = 0; y < matrix.size(); ++y)
    {
        // removing all empty cell in each row
        matrix[y].removeAll({0});
        // appending empty cells to fill the matrix
        for (int i = matrix[y].size(); i < 4; ++i)
            matrix[y].append(0);
    }
}
```

our method first removes empty cells, and in order to merge our tiles if their values are similar using a loop nest we merge our cells to the right starting from matrix.size—-> 0 for right movement and from 0—->matrix.size, for left movement then our function increments the score with the value the the current cell and re initializes the emptied cells to 0

To decrease code redundancy we also created a transpose() method that flips the matrix to the left to use moveRight() and moveLeft() function respectively with bottom() and top() movements

```
void DataModel::transpose()
{
    // creating a temp matrix
    QVector<QVector<int>> temp;
    temp.resize(4);
    for (int y = 0; y < temp.size(); ++y)
        temp[y].resize(4);

    // transposing the matrix
    for (int y = 0; y < temp.size(); ++y)
        for (int x = 0; x < temp[y].size(); ++x)
            temp[x][y] = matrix[y][x];

    matrix = temp;
}
```

```cpp
void DataModel::top()
{
    QVector<QVector<int>> temp = matrix;

    // transpose
    transpose();
    // move left
    moveLeft();
    // transpose
    transpose();

    // nothing changed so we stop as the movement is "invalid"
    if (matrix == temp)
    {
        checkIfDead();
        return;
    }

    // emiting signal for the view to know that data changed
    emit dataChanged(index(0, 0), index(rowCount() - 1, columnCount() - 1));

    // adding a cell filled with a 2
    addInEmptyCell();
}
```

```cpp
void DataModel::bottom()
{
    QVector<QVector<int>> temp = matrix;

    // transpose
    transpose();
    // move right
    moveRight();
    // transpose
    transpose();

    // nothing changed so we stop as the movement is "invalid"
    if (matrix == temp)
    {
        checkIfDead();
        return;
    }

    // emiting signal for the view to know that data changed
    emit dataChanged(index(0, 0), index(rowCount() - 1, columnCount() - 1));

    // adding a cell filled with a 2
    addInEmptyCell();
}
```

As explained before top() and bottom() methods first start by flipping the matrix to the left with transpose() function then then operates moveLeft for top movement or moveRight for bottom and transposes back the matrix to its initial position.

# 3 - Conclusion :

The model-view design pattern can be easily incorporated into any Qt project, either through simple properties or Qt's MVD framework. Separating backend business logic from frontend user interface is an important aspect of GUI design, one that the MVD framework handles effortlessly.
We have put into practice all the way through the implementation process, all the notions learnt throughout this semester this project provided us with a strong knowledge for building application with Qt creator.