A QML Memory Game Tutorial

This article shows how to create a simple memory game using Qt Quick. In the game users attempt to pair all the matching cards from a deck (which has been placed face-down) in the shortest possible time. The article explains the code in detail, from how to create one card, through to creating the entire deck and implementing the game logic.

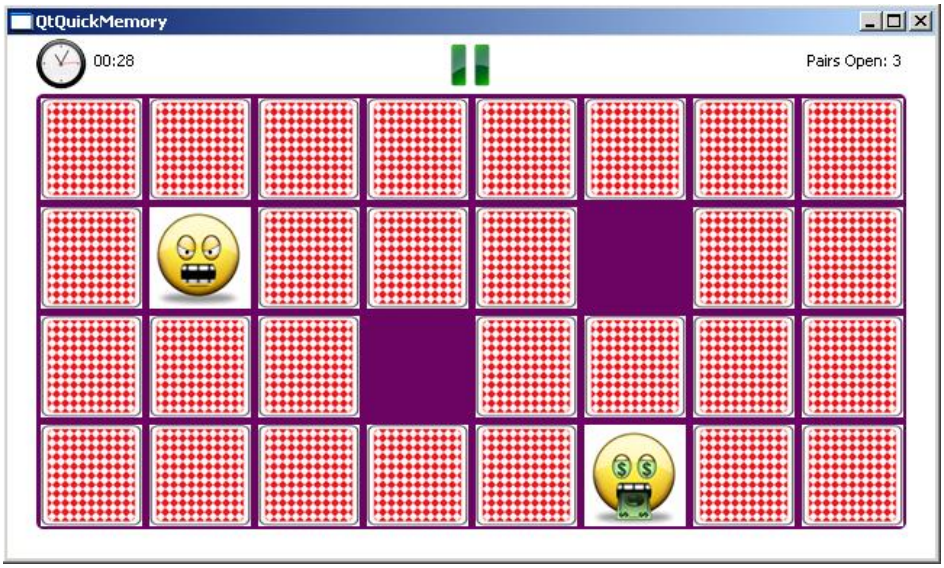
Game overview

30 Jan 2011

Below are the game rules:

- This game consists of a deck with 32 paired cards arranged randomly.
- You open a card by clicking or tapping on it, if so, the card will show its content
- You can open a second card in the same way
- If cards match, then they are removed from the deck
- If cards don't match, they cover themselves again
- Whether they match or not, the pair counter will go up 1 unit and this is displayed in the top/right
- On the top/left corner you will be able to see the elapsed time

Here is an image of the game scene:

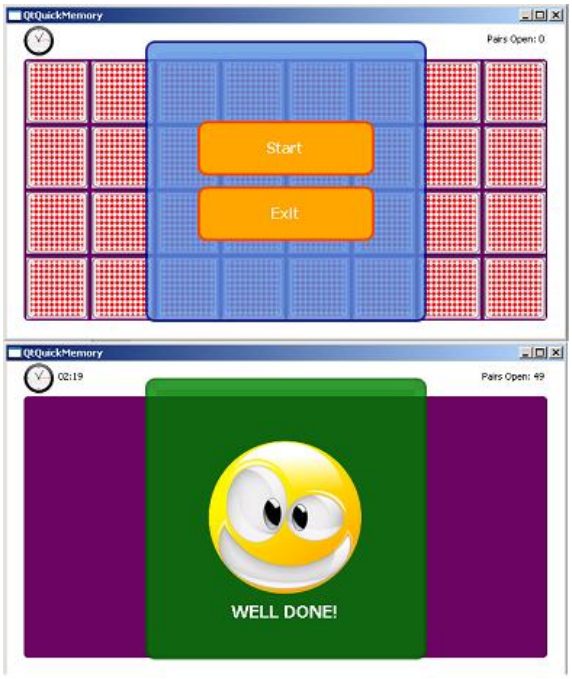


Main Menu and other Screens

The game contains a main menu presented in two different scenarios

- When the app starts or game is over: You will find two options (Start/Exit)
- When the game is paused: You will find three options (Continue/Restart/Exit)

It also contains an End Dialog box displayed when all the cards are matched and the game is over.



Constructing The Game

This is the list of concepts we need to keep in mind while making this game. QML objects are a set of Items that contain properties (defined by the Item or by the user), states, transitions and many other things we will not cover here. Items can be defined by the language like Rectangle, Image, Text or can be just a container (in this case only use Item). An Item can contain as many other items as it is necessary and these enclosed Items can have other Items, etc.

Creating a single Card

Basically, the card is composed by two Images, one is the cover and the other one is the content. The cover is a single Image and we can address it directly, but the content could be any of the 16 Images of the deck and must be passed from outside. To do this, we can use **properties** and bind the content Image to this property.

We can rotate the images in its default state using a transform. In this case we will use a **Rotation**. With this, we can rotate the image in any of its axes.

The idea is to have one image perpendicular to the screen, meaning rotated 90 degrees on the Y axis and then not visible; and the other one parallel to the screen, rotated 0 degrees on the Y axis and then visible. These rotations need an **id**, which is a defined **property** we can use to address it later and change one of its properties.

These is for instance the code of the image that contains the content of the card

```
//Content of the card
    Image {
        id: contentImg

        source: "img/card"+parNumber+".PNG"

         // Centers the image on its own container
        anchors.horizontalCenter: parent.horizontalCenter
        anchors.verticalCenter: parent.verticalCenter

        //this rotation produces de effect of the card content turning over and showing up
        transform:  Rotation{
            id: contentRotation
            origin.x: 35;
            origin.y: 35;
            axis { x: 0; y: 1; z: 0 }
            angle: 90
        }
    }
```

The way an Item can communicate to the outside world is using **signals**. This is very simple, you only have to declare the signal and emit it in some time you need (usually when the user taps/clicks on some specific Item).

```
    signal selected

    MouseArea {
        id: interactiveArea
        anchors.fill: parent     //expand the MouseArea to be contained in its parent Item
        onClicked: card.selected()
    }
```

On the outside world you can then use the following property to address the signal directly

```
onSelected:  <some javascript code>
```

We also need to understand the concept of **States** and **Transitions**.

- **State**: Is a set of values of different properties of an Item in a very specific instant of time.

For example, think about a car (Item), it has an engine (property) and different states like off and on. In on state, the engine is turned on, in off state, the engine is off.

- **Transitions**: Is something that happens when you change from one state to another. You can animate the change of value of one property between states using a set of animations defined in QML.

Following these ideas, our card will basically have 3 different states

- **closed**: The cover of the card is shown. This is the default state and its represented by an empty quotes ""
- **Open**: The content of the card is shown
- **removed**: The card is not visible and has no interaction

We can create a very cool **transition** between Closed and Open states using a the rotation on the Y axis in each of the two images.

This is how we create the effect: On a closed state we rotate the content Image over the Y axis 90 degrees, it is there, but you cannot see it. The cover Image is then shown completely.

On an Open state, we first rotate the cover Image 90 degrees so it is not visible (perpendicular to the screen), after that is finished, we rotate the content Image 90 degrees (parallel to the screen) so it is completely visible. We have achieved a flip animation. The only parameter to change in this case is the angle of the rotation, and the sequence will be very important since it has to come one rotation after the other, we can do that in this way (Remember "" is our closed or default state):

```
        Transition {
            from: ""
            to: "open"
            reversible: true
            //This animation produces the effect of the card flipping over and showing the content
            SequentialAnimation{
                NumberAnimation { target: coverRotation; property: "angle"; duration: 150 }
                NumberAnimation { target: contentRotation; property: "angle"; duration: 150 }
            }
        }
```

You will see as well in the code bellow a transition between the "Open" and "removed" states that creates a spin effect rotating the card and decreasing its size.

This is the complete Card code:

```
//  Card.qml
import QtQuick 1.0


//This item represents a single card on the deck
Item {

    id: card
    width:75
    height:75

    signal selected
    property string parNumber: '0'

    //Cover of the card
    Image {
        id: coverImg
        source: "img/cover.PNG"

        // Centers the image on its own container
        anchors.horizontalCenter: parent.horizontalCenter
        anchors.verticalCenter: parent.verticalCenter

        //This rotation produces the effect of the card cover turning over
        transform:  Rotation{
            id: coverRotation
            origin.x: 35;
            origin.y: 35;
            axis { x: 0; y: 1; z: 0 }
            angle: 0
        }

    }

    //Content of the card
    Image {
        id: contentImg

        source: "img/card"+parNumber+".PNG"

         // Centers the image on its own container
        anchors.horizontalCenter: parent.horizontalCenter
        anchors.verticalCenter: parent.verticalCenter

        //this rotation produces de effect of the card content turning over and showing up
        transform:  Rotation{
            id: contentRotation
            origin.x: 35;
            origin.y: 35;
            axis { x: 0; y: 1; z: 0 }
            angle: 90
        }
    }

    MouseArea {
```

```
        id: interactiveArea
        anchors.fill: parent
        onClicked: card.selected()
    }


    //-----------STATES---------------------

    states: [
        //State presented when the card is flipped over
        State {
            name: "open"
            PropertyChanges {
                target: coverRotation
                angle: 90
            }
            PropertyChanges {
                target: contentRotation
                angle: 0

            }
        },
        //State presented when the card is removed from the deck
        State {
            name: "removed"
            PropertyChanges {
                target: contentRotation
                angle:0

            }
            PropertyChanges {
                target: coverImg
                visible: false
            }

            PropertyChanges {
                target: interactiveArea
                enabled: false

            PropertyChanges {
                target: contentImg
                width:0
                height:0
                rotation:360
            }
        }
    ]


    // ------------- TRANSITIONS-------------------------

    transitions: [
        Transition {
            from: ""
            to: "open"
            reversible: true
            //This annimation produces the effect of the card flipping over and showing the content
            SequentialAnimation{
                NumberAnimation { target: coverRotation; property: "angle"; duration: 150 }
                NumberAnimation { target: contentRotation; property: "angle"; duration: 150 }
            }
        },
        Transition {
            from: "open"
            to: "removed"
            NumberAnimation { target: contentImg; properties: "rotation,width,height"; duration: 200 }

        }
    ]
}
```

Creating the deck and game logic

Now we need to construct the place where the game is held and its logic. At the end of this tutorial is the complete source code if you happen to get lost somewhere.

Deck

For the deck we will use a **positioner**, this is a kind of Item that allows you to arrange elements in a particular way. The positioner we will use is a **Grid**, you need to specify the number of rows and columns that will make this Grid. Inside a positioner you can use a **Repeater**, this element is the most similar thing in QML to a bucle. The **model** determines the number of iterations and you have access to the **index** property with the number of the current iteration.

The deck will be then created like this:

```
    //Deck holding the 32 cards
    Grid{
        id: deck
        rows: 4
        columns: 8

        //Repeater will let us arrange the 32 cards easily
        Repeater{
            model: 32

            //When the selected signal is emited, the openCard method is called
            //and the index is passed as argument
            Card{onSelected: openCard(index)}
        }
    }
```

Game Logic

The logic of the game is completely done in JavaScript. We can always mix QML code with JavaScript code.

The first thing to do is to arrange the cards randomly on the deck. In the following code, remember that we use the property of the card **parNumber** to get the content image since we have an image folder with 16 different images called like this **card1.PNG, card2.PNG**, etc

```
    //Randomize the cards on the deck
    function randomizeCards(){

        var sortedArray = ['1','2','3','4','5','6','7','8','9','10','11','12','13','14','15','16','1','2','3','4','5','6','7','8','9

        var date = new Date()
        var mils = date.getMilliseconds()       //Use miliseconds avoids the same random secuece generation among calls

        for(var i=0;i<32;i++){
            var located = false
            while(!located){
                var randomnumber = Math.floor((Math.random()*mils)%32)    // we retrieve the integer part of the generated number no

                var content= sortedArray[randomnumber]
                if(content !=''){  //if field is already empty, try again
                    deck.children[i].parNumber = content                        // If a number has been found, asign it to the foll
                    sortedArray[randomnumber]=''
                    located=true;               //go for the next iteration
                }
            }
        }

    }
```

Now we have a set of 32 random cards (16 pairs) on the table. We will use two several properties to keep track of the game, apart from some index holders we will have a property to keep the number of the pairs open (matched or not) and another to keep the number of the remaining couples to be found so we know when the game is over.

The logic will be pretty simple:

1. card1 is open, index is stored
2. card 2 is open, index is stored
3. a Timer of 1 second is launched
4. when the Timer is triggered a function to check if cards are equal is called
5. If cards are equal, cards are removed
6. If cards are different, cards are closed again

Timer is actually a QML item that can be started or stopped at any time

```
    //When two cards are open, this timer is enabled to close them in 1 second
    Timer{
        id:closeTimer
        interval: 1000
        onTriggered: validatePar()
    }
```

And the code to implement this functionality is the following

```
     //custom properties to hold values
    property int card1          //holds the index of the first card selected
    property int card2          //holds the index of the second card selected
    property int parCount       //indicates how many pairs have been open in total (doesnt mean they necesary match)
    property int remaining      //Indicates how many couples are left to end the game

    //This function is triggered when a card is selected
    function openCard(index){

        //Changes the state of the card to open
        deck.children[index].state = "open"

        //Storages the index of the card in one of the two placeholders card1 or card2
        if(card1==-1){
            card1=index
        }
        else{

            if(index== card1) return  //return if its the same card selected

            card2=index
            deck.enabled = false    //disables the deck while the pair is closed
            closeTimer.start()      //closeTimer is enabled and will close the cards in one second
        }
    }


    //Validate if the two open cards match or not, called when the closeTimer is triggered
    function validatePar(){

        var parNumber1 = deck.children[card1].parNumber
        var parNumber2 = deck.children[card2].parNumber
        var state = ""

        //If cards are equal they are removed
        if(parNumber1==parNumber2){
            state = "removed"
            remaining--     //one less card to find
        }

        deck.children[card1].state = state
        deck.children[card2].state = state

        //restablish initial values
        card1 = -1
        card2 = -1
        deck.enabled = true
        parCount++


        //If no remaining pairs, end of the game
        if(remaining==0){
        game.state="finished"
        }
```

```
    }
```

Keeping track of the time

We need to keep track of the elapsed time of the game so we can later create some "Best times" functionality (not included in this tutorial), or just to be aware of the time it takes for one person to complete the game. This requires some Items to display the information and some logic to calculate the time. We will keep the track of milliseconds in a variable. We use milliseconds because it is easier to construct a Date object with it to give it a proper format. We will have a Timer item that triggers a callback every second and in this callback we implement some logic to format the elapsed time and update the Items that display the information on the screen.

```
//--- QML code ---

    //Text indicating the elapsed time on the right of the screen
    Text{
        id: elapsedTimeText
        anchors.top: parent.top
        anchors.topMargin: 10
        anchors.left: crono.right
        anchors.leftMargin: 5

    }

  //This timer is triggered every second to keep the timer on the top-left running.  It is started manually when the game begins usi
    Timer{
        id:elapsedTimer
        interval: 1000
        running: false
        onTriggered: calculateElapsedTime()
        repeat: true

//--- Javascript code ---


    //This function shows the timer on the upper-left side of the screen
    function calculateElapsedTime(){

        miliseconds+=1000   //Since we use a timer with 1000 mseconds we add this time to the counter

        var date = new Date(miliseconds)

        //Arrange the format

        var seg = date.getUTCSeconds()
        seg = seg >9 ? seg : '0'+seg

        var mins = date.getUTCMinutes()
        mins = mins>9 ? mins : '0'+mins

        //Display the elapsed time
        elapsedTimeText.text = mins+":"+seg

    }

    }
```

Main Menu Implementation

we will create a mainMenu in a different file and will refer to it in the game file. The main menu is displayed when the game is stopped or paused. If the game is stopped it will contain only two options: Start and Exit. We will call this state "normal" If the game is paused, it will contain more options: Continue, Restart and Exit. We will call this state "extended" If the game is running, the mainMenu will be hidden, we will call this a "hidden" state.

We will also add a nice transition between the states.

```
// MainMenu.qml

import QtQuick 1.0


//This item represents the main menu (where user can start or quit the gamde)
Item {

    id: mainMenu
    width: 320
    height: 320

    signal started      //emitted when the start button is pressed
    signal continued    //emitted when the continue button is pressed
    signal exited       //emitted when the exit button is pressed


    //This rotation transform will allow a nice rotation effect over the x axix
    transform:  Rotation{
        id: contentRotation
        origin.x: 160;
        origin.y: 0;
        axis { x: 1; y: 0; z: 0 }
        angle: 0
    }


    //Background of the menu with some alpha blending
    Rectangle{
        width: 320
        height: 320
        color: "cornflowerblue"
        border.width: 3
        border.color: "darkblue"
        opacity: 0.8
```

```
    radius: 10
}

// Positions whatever amount of buttons we have in order in the screen
Grid{
    id: grid
    rows: 3
    columns: 1
    anchors.horizontalCenter: parent.horizontalCenter
    anchors.verticalCenter: parent.verticalCenter
    spacing: 15


    // Continue button showed when the game is paused, hidden by default
    Rectangle{

        id: continueButton
        width: 200
        height:60
        radius: 10

        color: "orange"
        border.width: 3
        border.color: "orangered"

        visible: false


        Text {
            id: continueText
            text: "Continue"
            anchors.verticalCenter: parent.verticalCenter
            anchors.horizontalCenter: parent.horizontalCenter
            font.bold: true
            font.pointSize: 12
            color: "white"
        }

        MouseArea{
            anchors.fill: parent
            onClicked: mainMenu.continued()     //When pressed the continued signal is emitted
        }
    }

    // Start Button presented as "Start" on the beginning or "Restart" when game is paused
    Rectangle{

        id: startButton
        width: 200
        height:60

        radius: 10

        color: "orange"
        border.width: 3
        border.color: "orangered"

        visible: false


        Text {
            id: startText
            text: "Start"
            anchors.verticalCenter: parent.verticalCenter
            anchors.horizontalCenter: parent.horizontalCenter
            font.bold: true
            font.pointSize: 12
            color: "white"
        }
        MouseArea{
            anchors.fill: parent
            onClicked: mainMenu.started()       //emits the start signal
        }
    }



    // Exit button
    Rectangle{

        id: exitButton
        width: 200
        height:60
        radius: 10
        color: "orange"
        border.width: 3
        border.color: "orangered"


        Text {
            id: exitText
            text: "Exit"
            anchors.verticalCenter: parent.verticalCenter
            anchors.horizontalCenter: parent.horizontalCenter
            font.bold: true
            font.pointSize: 12
            color: "white"
        }

        MouseArea{
            anchors.fill: parent
            onClicked: mainMenu.exited()  //emits the exited signal
        }
    }
}


//-------------STATES---------------------

states: [

    //Presented when the game is only started or it has finished
    State {
        name: "normal"
        PropertyChanges{
            target: startButton
            visible: true
        }
        PropertyChanges {
```

```
                target: startText
                text: "Start"
            }
            PropertyChanges {
                target: continueButton
                visible: false
            }
        },

        //Presented when the game is paused
        State {
            name: "extended"
            PropertyChanges{
                target: startButton
                visible: true
            }

            PropertyChanges {
                target: startText
                text: "Restart"
            }
            PropertyChanges {
                target: continueButton
                visible: true
            }
        },

        //Presented when the game is running
        State {
            name: "hidden"
            PropertyChanges {
                target: contentRotation
                angle: 90
            }
        }
    ]


    //-----------TRANSITIONS-----------------

   //Cool effects of rotation when state is changed to hidden and vice
    transitions: [
        Transition {
            from: "normal"
            to: "hidden"
            reversible: true
            NumberAnimation { target: contentRotation; property: "angle"; duration: 300 }
        },
        Transition {
            from: "extended"
            to: "hidden"
            reversible: true
            NumberAnimation { target: contentRotation; property: "angle"; duration: 300 }
        }
    ]
}
```

End Dialog

We will also have a dialog indicating the game has come to an end. This dialog will contain a text and an Image and will be dismissed when the user touch it. We will implement it in a different QML file and will access it in the maingame file

```
// End..qml

import QtQuick 1.0


// This item is shown when all the pairs have been found.
// It means you win!
Item {

    id: endMessage
    width: 320
    height: 320

    signal selected

    Rectangle {
        id: background
        color: "green"
        opacity: 0.8
        width: 320
        height: 320
        radius: 10
        border.width: 3
        border.color: "darkgreen"

    }

    //Once touched it sends a selected signal
    MouseArea{
        anchors.fill: parent
        onClicked: endMessage.selected()
    }

    Image {
        id: img
        width: 180
        height: 180
        source: "img/welldone.PNG"
        anchors.horizontalCenter: parent.horizontalCenter
        anchors.verticalCenter: parent.verticalCenter

    }

    Text {
        id: name
        text: "WELL DONE!"
```

```
        anchors.horizontalCenter: parent.horizontalCenter
        anchors.top: img.bottom
        anchors.topMargin: 5
        color: "white"
        font.bold: true
        font.pointSize: 14

    }
}
```

Game States

Going back to the game and now that we have covered the mainMenu and the end screen, we will describe the different states the game itself has

These are the different states of the game in short:

* **Stopped:** This is the default state represented by "" . This state presten Items as they were declared. Here the mainMenu is visible, the deck is not enabled (no interaction), the end dialog is hidden and timers are stopped. * **running:** In this state the elapsedTimer is running, the deck is enabled so you can pick cards, mainMenu is hidden and end dialog is stopped * **paused:** In this state the elapsedTimer is stopped, the deck is disabled, the mainMenu has a "extended" state and the end dialog is hidden * **finished** In this state the elapsedTimer is stopped, the deck is disabled, the mainMenu is in "hidden" state and the end dialog is visible

Pausing the Game

To take the game to a paused state we will create a simple button that will be placed on the top/center of the screen

```
    //Pause Function.  Only appears when the game is running
    Image{
        id:pause
        source: "img/pause.PNG"
        anchors.horizontalCenter: parent.horizontalCenter
        visible: false
        MouseArea{
            anchors.fill: parent
            onClicked: game.state = "paused"
        }
    }
```

Game: complete code

Here is all the code of the game containing the logic described above and the states described above

```
// Game.qml

import Qt 4.7

// It represents the game escene
Rectangle {
    id: game
    width: 640
    height: 360


    property int card1          //holds the index of the first card selected
    property int card2          //holds the index of the second card selected
    property int parCount       //indicates how many pairs have been open in total (doesnt mean they necesary match)
    property int miliseconds    //miliseconds elapsed on the game
    property int remaining      //Indicates how many couples are left to end the game



    //Clock icon on the top left corner
    Image {
        id: crono
        source: "img/clock.PNG"
        width: 35
        height: 35
        anchors.top: parent.top
        anchors.topMargin: 2
        anchors.left: background.left

    }

    //Text indicating the elapsed time on the right of clock image
    Text{
        id: elapsedTimeText
        anchors.top: parent.top
        anchors.topMargin: 10
        anchors.left: crono.right
        anchors.leftMargin: 5

    }

    //Text indicating the pairs open on the upper left
    Text {
        id: parCountText
        anchors.top: parent.top
        anchors.topMargin: 10
        anchors.right: background.right
        anchors.rightMargin: 3
        text: "Pairs Open: "+parCount
    }


    //Background where all the cards are located
    Rectangle{
        id: background
```

```
        width: 600
        height: 300
        anchors.top: parent.top
        anchors.left: parent.left
        anchors.topMargin: 40
        anchors.leftMargin: 20
        color: "#6F0564"
        radius: 5

    }

    //Pause Function.  Only appears when the game is running
    Image{
        id:pause
        source: "img/pause.PNG"
        anchors.horizontalCenter: parent.horizontalCenter
        visible: false
        MouseArea{
            anchors.fill: parent
            onClicked: game.state = "paused"
        }
    }


    //Deck holding the 32 cards
    Grid{
        id: deck
        rows: 4
        columns: 8

        anchors.top: background.top
        anchors.left: background.left

        //Repeater will let us arrange the 32 cards easily
        Repeater{
            model: 32

            //When the selected signal is emited, the openCard method is called
            //and the index is passed as argument
            Card{onSelected: openCard(index)}
        }
    }


    //When two cards are open, this timer is enabled to close them in 1 second
    Timer{
        id:closeTimer
        interval: 1000
        onTriggered: validatePar()
    }

    //This timer is triggered every second to keep the timer on the top-left running
    Timer{
        id:elapsedTimer
        interval: 1000
        running: false
        onTriggered: calculateElapsedTime()
        repeat: true

    }

    //Screen presented when the game is complete, not visible by default
    End{
        id: endScreen
        anchors.horizontalCenter: parent.horizontalCenter
        anchors.verticalCenter: parent.verticalCenter
        visible: false
        onSelected: game.state=""       //Move to initial state with mainMenu
    }


    //Has the main options of the game, start/quit/about/scores
    MainMenu{
        id: mainMenu
        state: "normal"
        anchors.horizontalCenter: parent.horizontalCenter
        anchors.verticalCenter: parent.verticalCenter
        onStarted: startGame()
        onContinued: game.state = "running"
        onExited: Qt.quit()
    }


   //------------  Javascript Functions ----------------------

    //Init all Variables
    function init(){
        card1=-1
        card2=-1
        parCount=0
        miliseconds=0
        remaining=16
    }


    //Starts the Game from the beginning
    function startGame(){

        init()                              //init variables

        //puts all the cards on its initial state
        for(var i=0;i<32;i++){
            deck.children[i].state=""
        }

        randomizeCards()

        game.state="running"
    }


    //Randomize the cards on the deck
    function randomizeCards(){

        var sortedArray = ['1','2','3','4','5','6','7','8','9','10','11','12','13','14','15','16','1','2','3','4','5','6','7','8','9

        var date = new Date()
        var mils = date.getMilliseconds()       //Use miliseconds avoids the same random secuece generation among calls

        for(var i=0;i<32;i++){
```

```
            var located = false
            while(!located){
                var randomnumber = Math.floor((Math.random()*mils)%32)

                var content= sortedArray[randomnumber]
                if(content !=''){  //if field is already empty, try again
                    deck.children[i].parNumber = content
                    sortedArray[randomnumber]=''
                    located=true;               //go for the next iteration
                }
            }
        }

    }


    //This function is triggered when a card is selected
    function openCard(index){

        //Changes the state of the card to open
        deck.children[index].state = "open"

        //Storages the index of the card in one of the two placeholders card1 or card2
        if(card1==-1){
            card1=index
        }
        else{

            if(index== card1) return  //return if its the same card selected

            card2=index
            deck.enabled = false    //disables the deck while the pair is closed
            closeTimer.start()      //closeTimer is enabled and will close the cards in one second
        }
    }

    //Validate if the two open cards match or not
    function validatePar(){


        var parNumber1 = deck.children[card1].parNumber
        var parNumber2 = deck.children[card2].parNumber
        var state = ""

        //If cards are equal they are removed
        if(parNumber1==parNumber2){
            state = "removed"
            remaining--     //one less card to find
        }

        deck.children[card1].state = state
        deck.children[card2].state = state

        //restablish initial values
        card1 = -1
        card2 = -1
        deck.enabled = true
        parCount++


        //If no remaining pairs, end of the game
        if(remaining==0){
        game.state="finished"
        }
    }


    //This function shows the timer on the upper-left side of the screen
    function calculateElapsedTime(){

        miliseconds+=1000   //Since we use a timer with 1000 mseconds we add this time to the counter

        var date = new Date(miliseconds)

        //Arrange the format

        var seg = date.getUTCSeconds()
        seg = seg >9 ? seg : '0'+seg

        var mins = date.getUTCMinutes()
        mins = mins>9 ? mins : '0'+mins

        //Display the elapsed time
        elapsedTimeText.text = mins+":"+seg

    }


// --------Game States ---------------------------

//  Different game states during its life.

//  The default state  "" indicates a stopped state showing the mainMenu

   states: [

       //Indicates the game is running
       State {
           name: "running"
           PropertyChanges {
               target: elapsedTimer
               running: true

           }
           PropertyChanges {
               target: endScreen
               visible: false
           }

           PropertyChanges {
               target: deck         //All the cards are enabled
               enabled: true
           }
           PropertyChanges {
               target: mainMenu
               state: "hidden"

           }
           PropertyChanges {
```

```
                target: pause        //pause image is shown
                visible: true
            }
        },

        //  Paused state showing the mainMenu with extended options
        State {
            name: "paused"
            PropertyChanges {
                target: elapsedTimer
                running: false
            }
            PropertyChanges {
                target: deck
                enabled: false
            }
            PropertyChanges{
                target: pause
                visible:false
            }
            PropertyChanges {
                target: mainMenu
                state: "extended"
            }
        },

        // Shows the game has reached its end and "Well Done" message is displayed
        State {
            name: "finished"
            PropertyChanges {
                target: elapsedTimer
                running: false
            }
            PropertyChanges {
                target: deck
                enabled: false
            }
            PropertyChanges {
                target: endScreen
                visible: true
            }
            PropertyChanges {
                target: mainMenu
                state: "hidden"
            }
        }
    ]

}
```

Source Code and SIS file

If you want to check this code, test the game or improve it, here are all the files including a sis file you can install in your phone if you have the Qt 4.7.1 libraries on your phone already.

File:QMLMemoryFiles.zip