QML : Qt Markup Language

François Delobel

DUT, Licence pro, IUT Aubière, Université d'Auvergne

Plan

Qt Markup Language

- Langage de QtQuick destiné à décrire des interface utilisateur.
- Ne pas confondre avec le XML des `.ui` pour les interfaces widget.
- Ça a franchement une tête de JSON (JavaScript Object Notation).
- Parfois utilisé en dehors de QtQuick
  - Les plasma de KDE
  - Unity2D, le gestionnaire de bureau de Canonical (pourtant à base de Gnome).
- Doit pouvoir être manipulé par un non informaticien...

Exemple super simple

```
// hello.qml
import QtQuick 2.0

Rectangle {
    Text {
      width: 50
      height: 50
      text: "Salut monde cruel"
    }
    color: "red"
}
```

Visible avec `qmlscene hello.qml`

Modules et éléments

Module

- `import` Importe un *module* : QtQuick 2 définit `Text`, `Rectangle`...
- Spécifie aussi (éventuellement) une version.

Élément

- Vient d'un module QML, de C++
- Peut être créé par l'utilisateur.
- Exemples pour QtQuick : `Rectangle`, `Text`, `Image`, `Item`

En bref

- Emboîtement d'éléments
- Chaque élément a des *propriétés*, de types différents...
- Tout élément parcouru est affiché.

Propriétés

- Exemples : `width`, `height`, `color`, `text`
- Initialisée par `nomProp: valeur`
- Documentées dans l'élément.
- Les propriétés sont héritées.

```
Rectangle {
  anchors.leftMargin: 5
  anchors.rightMargin: 10
}
// ou
Rectangle {
  anchors { leftMargin: 5; rightMargin:10 }
}
```

Positionnement des éléments

- L'ordre des éléments est important : éléments affichés dans l'ordre où ils sont rencontrés.
- Élément contenant : *parent*. Contenu : *enfant*
- un enfant a des coordonnées *relatives* à son parent.
- Utilisation des attributs `x` et `y` en pixels, coin haut/gauche.

Interactions entre composants

Nommer un élément

- Propriété `id`, nom commençant par minuscule (comme instance).
- `id: monIdentifiant`
- Permet de le désigner ailleurs, et donc d'en utiliser les propriétés.
- `parent` est un `id` implicite qui désigne l'objet parent.

Exemple d'utilisation de Propriété d'un autre élément

```
import QtQuick 2.0

Item {
    width: 400; height: 400
    Rectangle {
        id: monRectangle
        x: 100; y: 100
        color: "lightblue"
        height: parent.height - 150
        width: 200
    }
    Rectangle {
        x: 50; y: 50
        color: "lightgreen"
        height: monRectangle.height - 50
        width: parent.width - 100
    }
}
```

Types des propriétés

- int, size, string, bool, color, component, date, real, point, font, item...
- Voir QML Object Types...
- Liés à des types C++
- Plus un type *énumérations* utilisable comme valeur mais pas définissable dans QML
- Recalcul dynamique des propriétés !

Définir ses propres propriétés

```
import QtQml 2.0 // Pour QDate -> date

Item {
  id: myItem
  property string firstName: "Betty"
  property string lastName: "Boop"
  property int age: 99
  property color favColor: "red"
  property date born: "1970-01-01"
  property string name:  firstName + " " + lastName
  property string description: "%1, aged %2".arg(name).arg(age)
  property bool single: true

}
```

Le cas des multi dimensions

Size

- (`width` x `height`)
- `property size sz: "1024 x 768"`
- `property size sz: Qt.size(1024, 769)`

Point

- `x`, `y`
- `property point position: "51,64"`
- `property point position: Qt.point(51, 64)`

Rect

- `property rect r: "51,64, 200x200"`
- `property rect r: Qt.rect(51,64, 200,200)`

Définition d'un bouton

```
// Bouton.qml
import QtQuick 2.0

Rectangle {
    width: 100; height: 100
    color: "red"

    MouseArea {
        anchors.fill: parent
        onClicked: console.log("Button clicked!")
    }
}
```

- Une fois qu'un composant est défini, il peut être utilisé (attention à ce que le fichier ait le bon nom, ici Boutton).

Usage de notre bouton

```
// application.qml
import QtQuick 2.0

Column {
    Bouton { width: 50; height: 50 }
    Bouton { x: 50; width: 100; height: 50; color: "blue" }
    Bouton { width: 50; height: 50; radius: 8 }
}
```

Component : principe

- Un élément QML,
- Réutilisable,
- Qui fournit une *encapsulation* et donc une interface.
- Alternative à l'écriture du composant dans un fichier à part.
- Définit un élément, ne provoque donc pas un affichage.
- Peut être chargé dynamiquement grâce à un `Loader`.

```
Item {
    Component {
        id: redSquare
        Rectangle {
            color: "red"
            width: 10; height: 10
        }
    }

    Loader {
        sourceComponent: redSquare
    }
    Loader {
        sourceComponent: redSquare
        x: 20
    }
}
```

Component et fichier

Utiliser un composant est équivalent à définir un document QML (i.e., dans un fichier séparé).

- Un document QML masque tout ce qui n'est pas *top level*.
  - Seules les propriétés définies dans le premier niveau sont accessibles !
- Donc pas possible d'accéder aux propriétés internes.
  - C'est l'encapsulation !
- La portée des *id* est donc limitée à un composant.

Utiliser des *alias* de propriétés

Fichier qui déclare un document QML

```
// Biniou.qml
Rectangle {
    property alias texte: titre.text
    property alias titre: titre
    color: "red"
    width: 20
    height: 100

    Text {
        id: titre
        text: qsTr("Rien à dire")
        anchors.centerIn: parent
    }
}
```

Fichier qui utilise le document QML

```
Item {
    Biniou {
        id: me
        color: "lightblue"
        width: 100
        texte: qsTr("Troll")
        titre.color: "ForestGreen"
    }
}
```

Moralité : On expose dans le composant les attributs que l'on veut exporter.

Principales propriétés des éléments graphiques

Afficher / Masquer

- `visible`, propriété booléenne.
- `opacity`, propriété réelle. 1 : opaque, 0 : transparente.

Géométrie

- `scale`, propriété réelle. 1 : échelle originale
- `rotation`, propriété réelle, en degrés, sens horaire.

Améliorations visuelles

- `antialiasing` : prop. bool. Évite les effets d'escalier. Cher à calculer.
- `smooth` : lisse les petits angles. Activé par défaut.
- `clip` : prop. bool. Limite un objet à la taille de son parent.

Optimisation des transformations

transform

permet non seulement de combiner des transformations efficacement (une seule matrice), mais aussi de les stocker et les réutiliser.

```
transform: Rotation { angle 51; }
transform: [ Scale { xScale: 10 }, Rotation {angle 51 } ]
```

transformOrigin

- Spécifie le centre de l'homothétie.
- `[Top||Bottom][Left||Right]` et `Center`
- Exemple : `TopLeft` ou `Bottom`

Problématiques de tailles et positionnement

- Quelle taille prennent les éléments ?
  - Quelle est la taille d'un rectangle ?
  - Souvent petits par défaut.
- Où se placent les éléments ?
  - S'empilent au début de leur élément père.
- Nombreuses solutions possibles.

Placement direct des éléments

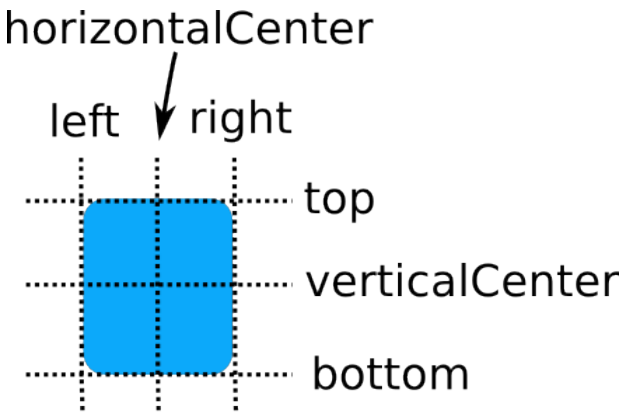
- Placement manuel (propriétés `x`, `y`, `width`, `height`)
  - Fastidieux et pénible pour redimensionnement.
- Placement relatif par rapport aux parents.

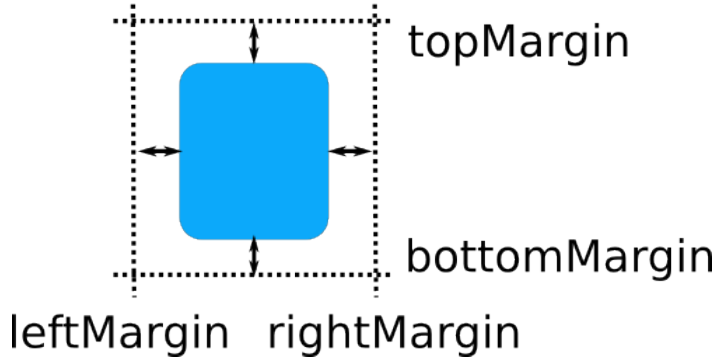
```
Rectangle {
  width: 220; height: 210
  Rectangle {
    id:rect1
    x:10; y:10
    width: (parent.width - 30) / 2
    height: parent.height - 20
  }
  Rectangle {
    x: rect1.width + 20; y:10
    width:  rect1.width
    height: rect1.height
  }
}
```

Ancres (anchors)

- Ancre : point de repère d'un élément qu'il est possible de positionner par rapport à une autre ancre d'un élément parent ou frère (*sibling*).
- Les sept lignes d'ancres prédéfinies :
  - Les bords : `anchors.top`, `anchors.bottom`, `anchors.left`, `anchors.right`.
  - Les centres : `anchors.horizontalCenter`, `anchors.verticalCenter`.
  - La base du texte : `anchors.baseline` (vaut `top`) si pas de texte.
- Deux ancres virtuelles :
  - Le milieu : `anchors.centerIn`
  - Remplir l'espace : `anchors.fill`

Positions des ancres et marges

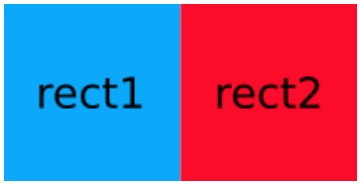




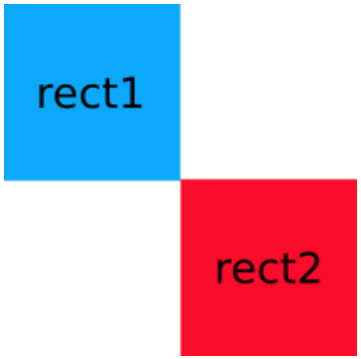
Les marges des ancres

- Pour les bords, des marges : `leftMargin`, `rightMargin`, `topMargin`, `bottomMargin`.
- Marges par défaut : `anchors.margins`
- Distances aux milieux ; `horizontalCenterOffset`, `verticalCenterOffset`, `baselineOffset`.

Petits exemples avec des ancres




```
Rectangle { id: rect1; ... }
Rectangle { id: rect2; anchors.left: rect1.right; ... }
```




```
Rectangle { id: rect1; ... }
Rectangle { id: rect2;
            anchors.left: rect1.right;
            anchors.top: rect1.bottom; ... }
```

Petits exemples avec des ancres (suite)


```
Rectangle { id: rect1; x: 0; ... }
Rectangle {
  id: rect2;
  anchors.left: rect1.right;
  anchors.right: rect3.left;
  ... }
Rectangle { id: rect3; x: 150; ... }
```

- Quand deux ancres opposées sont fixées, la taille de l'élément s'adapte !

Usage des ancres : exemple

```
Rectangle {
    width: 300; height: 200
    Rectangle {
        anchors.left: parent.left
        anchors.top: parent.top
        anchors.bottom: parent.bottom
        anchors.right: parent.horizontalCenter
        anchors.margins: 10
        anchors.rightMargin: 5
    }
    Rectangle {
        anchors.left: parent.horizontalCenter
        anchors.top: parent.top
        anchors.bottom: parent.bottom
        anchors.right: parent.right
        anchors.margins: 10
        anchors.leftMargin: 5
    }
}
```

Les Layouts

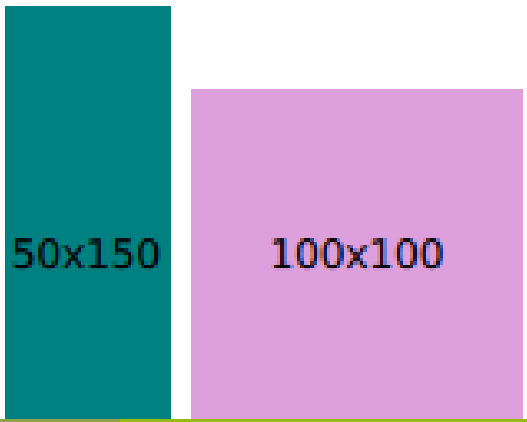
- Positionnement automatique comme avec les layout des QWidget
- À partir de QtQuick 2.1 (Qt 5.1)
- Très pratique : contient des éléments et les range !
- Répartit l'espace disponible entre les éléments.
- Gère le redimensionnement !
- Importer `QtQuick.Layouts 1.0`
- `RowLayout`, `ColumnLayout`, `GridLayout`, `GridLayout`
- Pré-QtQuick 2.1 : les `RowLayout`, `Column`, `Grid`, `Flow`. Moins bonne gestion du redimensionnement (mais fort utilisés encore).

La colonne par l'exemple

```
Item {
    width: 310; height: 170
    Column {
        anchors.horizontalCenter: parent.horizontalCenter
        anchors.verticalCenter: parent.verticalCenter
        spacing: 5
        Rectangle { color: "lightblue"; radius: 10.0
                    width: 300; height: 50
                    Text { anchors.centerIn: parent
                           font.pointSize: 24; text: "Books" } }
        Rectangle { color: "gold"; radius: 10.0
                    width: 300; height: 50
                    Text { anchors.centerIn: parent
                           font.pointSize: 24; text: "Music" } }
        Rectangle { color: "lightgreen"; radius: 10.0
                    width: 300; height: 50
                    Text { anchors.centerIn: parent
                           font.pointSize: 24; text: "Movies" } }
    }
```

Rendu sans surprise

```
RowLayout {
    id: layout
    anchors.fill: parent
    spacing: 6
    Rectangle {
        color: 'teal'
        Layout.fillWidth: true
        Layout.minimumWidth: 50
        Layout.preferredWidth: 100
        Layout.maximumWidth: 300
        Layout.minimumHeight: 150
        Text {
            anchors.centerIn: parent
            text: parent.width \
                  + 'x' + parent.height
        }
    }
    Rectangle {
        color: 'plum'
        Layout.fillWidth: true
        Layout.minimumWidth: 100
        Layout.preferredWidth: 200
        Layout.preferredHeight: 100
        Text {
            anchors.centerIn: parent
            text: parent.width \
                + 'x' + parent.height
        }
    }
}
```



La grille

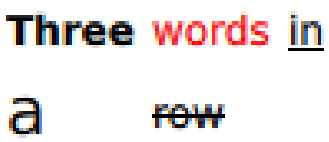
- La base des autres layouts : Le `row` est une `grid` d'une seule ligne !
- On peut ne donner qu'une seule des deux dimensions (ligne ou colonne).

```
GridLayout {
    rows: 4; columns: 4
    width: 200; height: 200
}
```

Exemple de grille

```
GridLayout {
    id: grid
    columns: 3

    Text { text: "Three"; font.bold: true; }
    Text { text: "words"; color: "red" }
    Text { text: "in"; font.underline: true }
    Text { text: "a"; font.pixelSize: 20 }
    Text { text: "row"; font.strikeout: true }
}
```



Flow

- Place les élément les uns à la suite des autres.
- Choix de l'axe principal de de l'axe secondaire (Major and minor axis) : Propriété `Flow`.
- `Flow` : `TopToBottom`, de haut en bas.
- `Flow` : `LeftToRight` de droite à gauche (suivant la `layoutdirection` qui permet d'inverser le sens d'écriture.

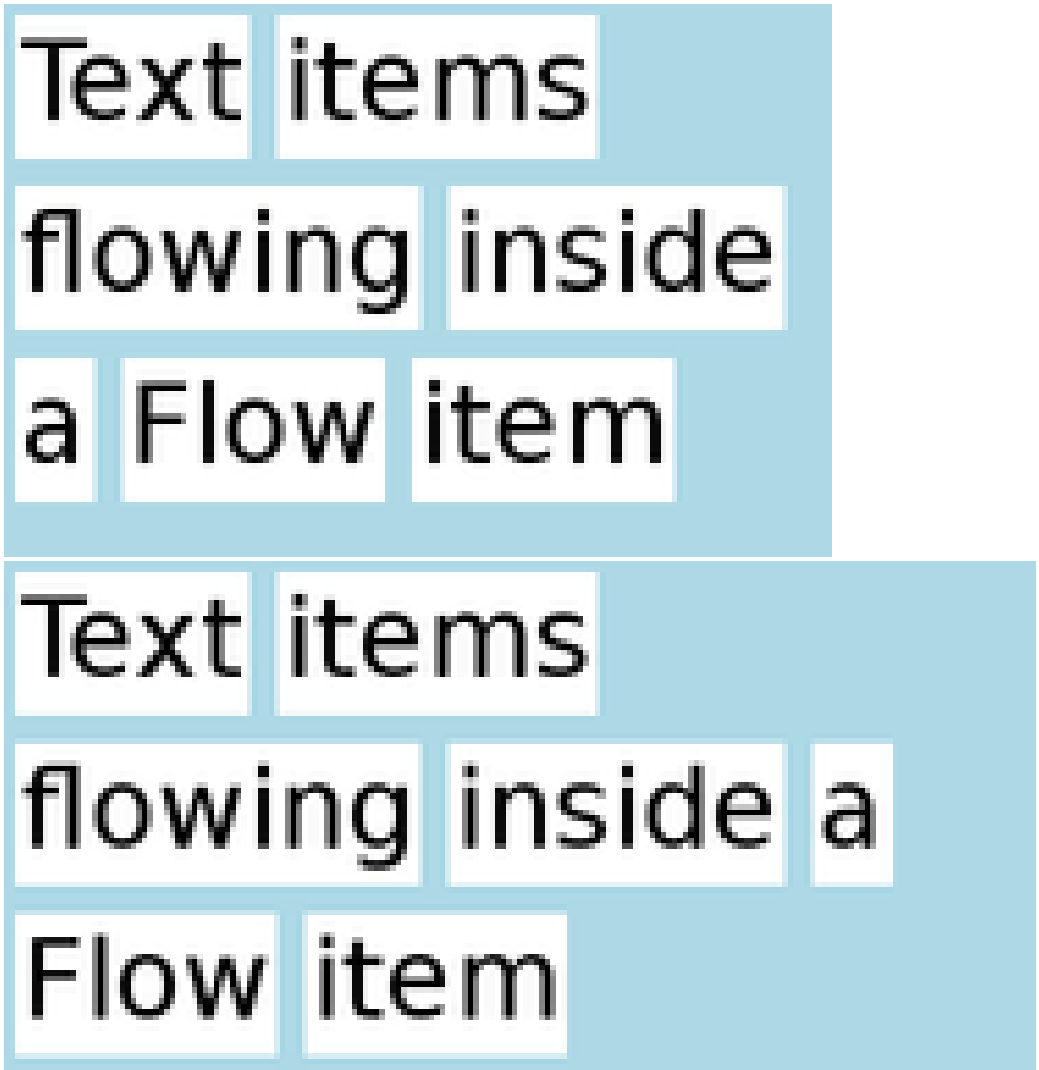
Un flow de la doc

```
Rectangle {
    color: "lightblue"
    width: 300; height: 200

    Flow {
        anchors.fill: parent
        anchors.margins: 4
        spacing: 10

        Text { text: "Text"; font.pixelSize: 40 }
        Text { text: "items"; font.pixelSize: 40 }
        Text { text: "flowing"; font.pixelSize: 40 }
        Text { text: "inside"; font.pixelSize: 40 }
        Text { text: "a"; font.pixelSize: 40 }
        Text { text: "Flow"; font.pixelSize: 40 }
        Text { text: "item"; font.pixelSize: 40 }
    }
}
```

Une sortie de flow avec deux largeurs différentes



Contrôles

- *Contrôles* : éléments plus complexes fournis par le langage pour réutilisation.
- Attention, moins disponibles dans les anciennes versions de QtQuick.
- Ajout QtQuick Controls à partir Qt5.1 :
  `import QtQuick.Controls 1.0`
  Dispo à l'IUT : `import QtQuick.Controls 1.2`
- Adaptent leur style à la plateforme.
- Un peu les `QWidget` de QtQuick.
- À manipuler avec le designer pour une première présentation.
- Styles possibles avec QtQuick.Controls.Styles : un style par contrôle visuel (e.g., `ButtonStyle`)
- Silica propose ses propres contrôles (qui ressemblent aux contrôles QtQuick 1.0).

Liste des contrôles Qt 5.4 (Controls 1.2)

- Action
- ApplicationWindow
- BusyIndicator
- Button
- Calendar
- CheckBox
- ComboBox
- ExclusiveGroup
- GroupBox
- Label
- Menu
- MenuBar
- MenuItem
- MenuSeparator
- ProgressBar
- RadioButton
- ScrollView
- Slider
- SpinBox
- SplitView
- Stack
- StackView
- StackViewDelegate
- StatusBar
- Switch
- Tab
- TabView
- TableView
- TableViewColumn
- TextArea
- TextField
- ToolBar
- ToolButton

Le *Hello World* du contrôle : `Button`

Hérite de

- FocusScope (donc de Item)

Propriétés

- action : Action
- activeFocusOnPress : bool
- checkable : bool
- checked : bool
- exclusiveGroup : ExclusiveGroup
- hovered : bool
- iconName : string
- iconSource : url
- isDefault : bool
- menu : Menu
- pressed : bool
- style : Component
- text : string
- tooltip : string

Signaux

- clicked()

▸ Regarder la doc officielle

Décorer un bouton : les styles

- Un bouton ne contient rien lié à l'affichage du bouton.
- `ButtonStyle` contient les propriétés visuelles des boutons.
  - Évite de complexifier inutilement le bouton.
  - Permet de factoriser les styles.
- La propriété `style` du bouton contient un `ButtonStyle`
- Tout Style contient une propriété `control` qui désigne l'objet incorporant le style.
  - Le `control` du `ButtonStyle` est donc un `Button`.
- `ButtonStyle` contient aussi
  - background : Component
  - label : Component

```
//import QtQuick  2.0 + QtQuick.Controls 1.2 + QtQuick.Controls.Styles 1.2

Row { Button {
    text: "A button"
    id: firstButton
    style: ButtonStyle {
      background: Rectangle {
        implicitWidth: 100   // Largeur par defaut
        implicitHeight: 25  //  Hauteur par defaut
        border.width: control.activeFocus ? 2 : 1
        border.color: "#888"
        radius: 4
        gradient: Gradient {    GradientStop { position: 0 ;
          color: control.pressed ? "#222" : "#eee" }
          GradientStop { position: 1 ;
            color: control.pressed ? "#fff" : "#ccc" }
        }
      }
    }
  }
  Button {
    text: "Another button"
    style: firstButton.style
  }
}
```

Quelques définitions pour QML

- *Type* : *Basic Type* ou *Object Type*
- *Basic Type* : `int`, `string`, `bool`...
- *Object Type* : instanciable par QML. Défini soit par un fichier QML éponyme, soit par un QObject C++.
- *Object* : Une instance d'*Object Type*, créé par l'évaluation d'une une page QML (ou par `Component.CreateObject()` ou `\wQt.createQmlObject()`.
- *Component* : un template qui sert à créer un objet QML (éventuellement composite). Essentiellement créé par le parse d'un *Document* mais peut être aussi déclaré *en ligne* dans un document.
- *Document* : Essentiellement un fichier `.qml`, avec un nom capitalisé (mais peut être aussi stocké dans une chaîne.