L'interface de NoteApp peut être considérée comme complète en terme de fonctionnalité et d'interactions avec l'utilisateurs.
Cependant, il y a encore une grosse marge de progression pour rendre l'interface plus attrayante à l'utilisateur.
Le QML a été considéré comme un langage déclaratif, se rappelant des animations et des transitions fluides des éléments de l'interface.

Le chapitre va couvrir les thèmes principaux suivants :

Introduire des concepts à propos des animations et des transitions en QML

De nouveaux types de QML seront introduits, comme le *Behaviour*, *Transition* et plusieurs A*nimations* d'éléments

Le chapitre suit les étapes suivantes :

5.1 Animer la NoteToolbar

Voyons comment nous pouvons améliorer le composant *Note* et ajouter un *Behaviour* basé sur l'interaction avec l'utilisateur. Le composant *Note* a une toolbar avec un bouton *supprimer** pour supprimer la note. De plus, la toolbar est utilisé pour déplacer la note en gardant le clic de la souris appuyé dessus.

Une amélioration pourrait être de rendre le bouton *Supprimer* visible seulement si nécessaire. Par exemple, en rendant le bouton *Supprimer* visible quand la toolbar est survolée, et ce serait agréable d'utiliser des effets de fade-in et de fade-out.

Le QML fournit plusieurs approche pour implémenter ceci en utilisant les types *animation* et *transition.* Dans ce cas spécifique, nous utiliserons le type *Behaviour* du QML, et nous expliquerons plus tard pourquoi.

5.1.1 Type Behaviour et NumberAnimation

Dans le composant *Notetoolbar,* on utilise le type *Row* pour disposer le bouton Supprimer*, donc en changeant la propriété opacité du type *Row* va aussi affecter l'opacité du bouton Supprimer.

Note : La valeur de la propriété opacité est propagé des items parents aux items enfants

Le type *Behaviour* aide à définir le *Behaviour* de l'item basé sur les changements de propriété de cet item comme montré sur le code suivant :

```
// NoteToolbar.qml
...
MouseArea {
id: mousearea
anchors.fill: parent
// setting hoverEnabled property to true
```

```
// in order for the MouseArea to be able to get
// hover events
hoverEnabled:
true
}
// using a Row element for laying out tool
// items to be added when using the NoteToolbar
Row {
id: layout
layoutDirection: Qt.RightToLeft
anchors {
verticalCenter: parent.verticalCenter;
left: parent.left;
right: parent.right
leftMargin: 15;
rightMargin: 15
}
spacing: 20
// the opacity depends if the mousearea
// has the cursor of the mouse.
opacity: mousearea.containsMouse ? 1 :
0
// using the behavior element to specify the
// behavior of the layout element
// when on the opacity changes.
// using NumberAnimation to animate
// the opacity value in a duration of 350 ms
NumberAnimation { duration: 350 }
}
}
```

…

Comme vous pouvez le voir sur le code ci-dessus, on active la propriété hoverEnabled* du type MouseArea pour accepter les événements de survol de la souris. Ensuite, on bascule l'opacité du type Row à 0 si le type *Mousearea* n'est pas survolé et a 1 sinon. La propriété containsMouse de MouseArea est utilisé pour décider de la valeur de l'opacité pour le type *Row*.

Le type Behaviour est créé a l'intérieur du type *Row* pour définir son Behaviour basé sur sa propriété *opacité.* Quand la valeur de l'opacité change, *NumberAnimation* est appliquée.

Le type *NumberAnimation* applique une animation basée sur des changements de valeurs numériques, nous l'utilisons donc sur la propriété *opacité* du *Row* pour une durée de 350 milliseconds.

Note : Le type *NumberAnimation* est hérité de *PropertyAnimation*, qui a *Easting.Linear* comme animation de la courbe d'accélération par défaut.

Et ensuite ?

Dans la prochaine étape, nous verrons comment implémenter une animation en utilisant Transition* et d'autres types d'animation QML.

5.2 Utiliser états et des transitions

Dans l'étape précédente, nous avons vu une approche pratique pour définir de simples animations basées sur les changements de propriétés, en utilisant les types *Behaviour* et *NumberAnimation.*

Evidemment, il y a des cas dans lesquels l'animation dépends d'une palette de changements de propriétés qui pourraient être représentés par un *State*.

Voyons comment nous pouvons aller plus loin dans l'amélioration de l'interface du NoteApp*.

Les items Marker paraissent statiques quant on en vient à l'interaction de l'utilisateur. Et si on voulait ajouter quelques animations basées sur plusieurs scénarios d'interaction de l'utilisateur ?

De plus, nous voudrions rendre le marqueur actuelle actif et la page actuelle plus visible pour l'utilisateur.

5.2.1 Animer les items Marker

Si nous voulons résumer les scénarios possible pour améliorer les interactions de l'utilisateur avec des items *Marker*, les cas d'usage suivants sont décris :

Le *Marker* actif actuel devrait être plus visible. Un marker devient actif quand l'utilisateur clique dessus. Le marker actif est un peu plus gros, et il pourrait glisser de gauche à droite (tout comme un curseur).

Quand un utilisateur survole un marker avec une souris, le marqueur glisse de gauche à droite mais pas autant qu'un marqueur actif le ferait.

En considérant les scénarios mentionnés ci-dessus, nous devons travailler sur les composants *Marker* et *MarkerPanel.*

En lisant la descriptions ci-dessus à propos du comportement désiré (l'effet de glissement de gauche à droite), je pense en premier lieu à changer la propriété *x* de l'item *Marker* comme il représente la position de l'item sur l'axe X. De plus, comme l'item marqueur doit savoir si c'est le marqueur actif actuelle, une nouvelle propriété appelée *active* peut être introduite.

On peut introduire 2 états pour le composant *Marker* qui peuvent représenter le comportement décrit ci-dessus :

hovered- qui va mettre à jour la propriété *x* du marqueur quand l'utilisateur le survole en utilisant la souris.

S*elected*- qui va mettre à jour la propriété *x* du marqueur quand le marqueur devient actif, ce qui signifie, quand il est cliqué par l'utilisateur.

```
// Marker.qml

...
// this property indicates whether this marker item
// is the current active one. Initially it is set to false
property bool active:
false
// creating the two states representing the respective
// set of property changes
states: [
// the hovered state is set when the user has
// the mouse hovering the marker item.
State {
name: "hovered"
// this condition makes this state active
when: mouseArea.containsMouse && !root.active
PropertyChanges { target: root; x: 5 }
},
State {
name: "selected"
when: root.active
PropertyChanges { target: root; x: 20 }
}
]
// list of transitions that apply when the state changes
transitions: [
Transition {

to: "hovered"
NumberAnimation { target: root; property: "x"; duration: 300 }
},
Transition {
to: "selected"
NumberAnimation { target: root; property: "x"; duration: 300 }
},
Transition {
to: ""
NumberAnimation { target: root; property: "x"; duration: 300 }
}
]
...
```

On a donc états déclarés qui représentent les changements respectifs de propriétés basés sur le comportement de l'utilisateur. Chaque état est lié à une condition exprimée dans la propriété *when*

Note : Pour la propriété *containsMouse* du type MouseArea, la propriété *hoverEnabled* doit être mise à *true*.

Le type Transition est utilisé pour définir le comportement de l'item quand il change d'un état à un autre. Cela signifie que l'on peut définir plusieurs animations grâce aux propriétés qui changent quand un état devient actif.

Note:L'état par défaut d'un item est une chaîne de caractère vide, (“”)

Pendant que nous somme dans le composant *MarkerPanel*, nous devons régler la propriété *active* de l'item *Marker* sur *true* lorsque l'on clique dessus. Se référer à *MarkerPanel.qml* pour le code mis à jour.

5.2.2 Ajouter des transitions à PagePanel

Dans le composant *PagePanel*, nous utilisons des états pour gérer la navigation entre les pages. Ajouter des transitions nous vient naturellement à l'esprit. Comme nous changeons la propriété *opacité* dans chaque état, nous pouvons ajouter *Transition* pour tout les états qui contrôlent un *NumberAnimation* sur les valeurs de l'opacité pour créer les effets fade-in et fade-out.

// PagePanel.qml

```
...
// creating a list of transitions for
// the different states of the PagePanel
transitions: [
Transition {
// run the same transition for all states
from: " "; to: "
*
"
NumberAnimation { property: "opacity"; duration: 500 }
     }
]
…
```

Note: La valeur *opacity* d'un item est propagé à ces éléments enfants aussi

Et ensuite ?

Dans la prochaine étape, nous allons apprendre à améliorer plus en détails l'interface utilisateur et voir ce que l'on peut faire de plus.

Amélioration en profondeur

A ce niveau, nous pouvons considérer que les fonctionnalités de NoteApp* sont complètes et que l'interface utilisateur corresponde aux spécifications de *NoteApp*. Néanmoins, il ya toujours de la place pour plus d'améliorations, celles-ci peuvent être mineures mais elle rendent l'application plus finie et prête à l'emploi.

Dans ce chapitre, nous verrons les petites améliorations qui sont implémentées pour NoteApp*, mais aussi suggérer de nouvelles idées et fonctionnalités qui pourraient être ajoutées. Bien entendu, nous souhaiterions encourager tout le monde à prendre le code source de NoteApp* et le développer plus en profondeur en redesignant éventuellement l'entièreté de l'interface utilisateur et en ajoutant de nouvelles fonctionnalités.

Voici une liste des thèmes principaux abordés dans ce chapitre :

Plus de javascript utilisé pour améliorer la fonctionnalités.

Travailler avec le classement en z des items QML

Utiliser des polices customisés pour l'application

6.1 Améliorer la fonctionnalité de l'item Note

Une chouette fonctionnalité pour les items *Note* serait d'avoir la note qui grandit au fur et à mesure que du texte est tapé.

Disons juste, pour des raisons de simplicité, que la note va grandir verticalement tant que du texte est entré et que la note entoure le texte pour tenir en largeur.

Le type *Text* a une propriété *paintedHeight* qui nous donne la taille actuelle du texte affiché sur l'écran. A partir de cette valeur, nous pouvons augmenter ou diminuer la hauteur de la note.

Premièrement, définissons une fonction Javascript helper qui calcule la valeur de la propriété hauteur du type *Item*, qui est le type de plus haut niveau pour le composant *Note.*

// Note.qml

```
...
// JavaScript helper function that calculates the height of
// the note as more text is entered or removed.
function
updateNoteHeight() {
// a note should have a minimum height
var
```

```
noteMinHeight = 200
var
currentHeight = editArea.paintedHeight + toolbar.height +40
root.height = noteMinHeight
if
(currentHeight >= noteMinHeight) {
root.height = currentHeight
}
}
...
```

Tant que la fonction *updateNoteHeight()* met à jour la propriété *height* de *root* basée sur la propriété *paintedHeight* de editArea, nous devons appeler cette fonction via un changement sur *paintedHeight.*

// Note.qml

...

// creating a TextEdit item

TextEdit {

id: editArea

...

// called when the painterHeight property changes

// then the note height has to be updated based

// on the text input

onPaintedHeightChanged: updateNoteHeight()

...

}

Note :Toutes les propriétés émettent un signal de notification à chaque fois qu'une propriété changeant

La fonction JavaScript *updateNoteHeight()* change la propriété *height,* on peut donc définir un comportement pour cela en utilisant le type *Behavior.*

```
// Note.qml

...

// defining a behavior when the height property changes

// for the root element

Behavior on height { NumberAnimation {} }
```

Et ensuite ?

La prochaine étape montre comment utiliser la propriété *z* du type *Item* pour arranger convenablement les notes.

6.2 Arranger les Notes

Les notes à l'intérieur d'une page ne savent pas sur quel note l'utilisateur est en train de travailler. Par défaut, tous les objets *Notes* créés ont la même valeur par défaut pour la propriété *z* et dans ce cas-ci, le QML crées un arrangement par défaut des objets basés sur celui qui a été créé en premier.

Le comportement désiré serait de changer l'ordre des notes selon une intercation de l'utilisateur.

Quand l'utilisateur clique sur la note toolbar ou commence à éditer une note, la dite note devrait ressortir pour ne pas être en dessous d'autres notes. Cela est possible en changeant la valeur *z* pour qu'elle soit plus haute que les autres notes.

```
// Note.qml

Item {

id: root

...

// setting the z order to 1 if the text area has the focus

z: editArea.activeFocus ? 1:0

...

}
```

Dans le gestionnaire de signal *onPressed* dans le type *MouseArea*, on émet le signal *pressed()* du *root* de la *NoteToolbar.*

Le signal *pressed()* du composant *NoteToolbar* est géré dans le composant *Note.*

```
// Note.qml

...

// creating a NoteToolbar item that will be anchored to its parent

NoteToolbar {

id: toolbar

...

// setting the focus on the text area when the toolbar is pressed

onPressed: editArea.focus = true

...

}
```