helloQt ist die Qt-Entwicklungseinheit
der Bergx2 GmbH

Fürstenstr. 15
80333 München
Deutschland

info@helloqt.de
www.helloqt.de

Displaymanagement mit Qt 6 meistern: eine Auswahl an Möglichkeiten

23. Juni 2024 von Lucas Moreira de Oliveira

Ein gutes Bildschirmmanagement ist keine Glücksache, vielmehr entscheidet auch die richtige Wahl passender Lösungen innerhalb der Qt 6-Anwendung. So lässt eine reibungslose und effiziente Benutzererfahrung erzielen. Egal, ob Sie eine einfache App mit wenigen Bildschirmen oder eine komplexe Anwendung mit dynamischen Schnittstellen entwickeln, das Verständnis der verschiedenen in Qt 6 verfügbaren Methoden hilft Ihnen dabei, fundierte Entscheidungen zu treffen. In diesem Artikel stelle ich verschiedene Techniken zur Bildschirmverwaltung vor und gehe auf ihre Implementierungen, Vor- und Nachteile sowie die Szenarien ein, in denen jede Methode glänzt.

1. Dynamische Bildschirmerstellung mit „component.createComponent“ und „component.createObject“

Eine der leistungsstarken Funktionen von Qt 6 ist die Möglichkeit, Komponenten dynamisch zur Laufzeit zu erstellen. Dies ist insbesondere dann nützlich, wenn die Anzahl oder Art der Bildschirme zur Kompilierzeit nicht bekannt ist.

1.1 Praktisches Beispiel

```qml
import QtQuick 2.15
import QtQuick.Controls 2.15

ApplicationWindow {
    visible: true
    width: 640
    height: 480

    Button {
        text: "Create Screen"
        onClicked: createScreen()
    }
    function createScreen() {
        var component = Qt.createComponent("Screen.qml");
        if (component.status === Component.Ready) {
            var dynamicScreen = component.createObject(applicationWindow);
            dynamicScreen.width = 640;
```

```
            dynamicScreen.height = 480;
        }
    }
    function destroyScreen(screen) {
        if (screen !== null) {
            screen.destroy();
        }
    }
}
```

In diesem Beispiel wird beim Klicken auf die Schaltfläche dynamisch ein neuer Bildschirm erstellt. Die Funktion `Qt.createComponent` lädt die QML-Komponente und `createObject` realisiert sie.

1.3 Vor- und Nachteile der dynamischen Bildschirmerstellung

Die dynamische Erstellung bietet Flexibilität und effiziente Speichernutzung, da Komponenten nur bei Bedarf dargestellt werden. Allerdings kann es während der Erstellung zu einer leichten Verzögerung kommen was eine sorgfältige Fehlerbehandlung erfordert,, um den Ladezustand der Komponente und mögliche Fehler zu verwalten. Diese Methode ist ideal für Szenarien, in denen Bildschirme nicht gleichzeitig benötigt werden, wie etwa in mehrstufigen Formularen oder funktionsreichen Anwendungen mit vielen optionalen Modulen.

1.4 Weiterführende Literatur

- Dynamische QML-Objekterstellung mit JavaScript: https://doc.qt.io/qt-6/qtqml-javascript-dynamicobjectcreation.html

2. Bildschirm Management mit `Loader`

Das „Loader“-Element in Qt 6 bietet eine praktische Möglichkeit, das Laden und Entladen von QML-Komponenten zu verwalten, ermöglicht Lazy Loading und reduziert den anfänglichen Speicherbedarf Ihrer Anwendung.

2.1 Praktisches Beispiel

```qml
import QtQuick 2.15
import QtQuick.Controls 2.15

ApplicationWindow {
    visible: true
```

```
    width: 640
    height: 480

    Loader {
        id: screenLoader
        anchors.fill: parent
    }

    Button {
        text: "Load Screen"
        onClicked: screenLoader.source = "Screen.qml"
    }
}
```

Hier wird ein `Loader` verwendet, um einen Bildschirm zu laden, wenn auf die Schaltfläche geklickt wird. Der `Loader` kann die Komponente auch entladen, indem er seine `source`-Eigenschaft auf ein leeres String setzt.

2.2 Vor- und Nachteile des Loaders in Qt 6

Die Verwendung von `Loader` vereinfacht die Speicherverwaltung und verbessert die Leistung, indem Komponenten nur bei Bedarf geladen werden. Häufiges Laden und Entladen kann jedoch zu Leistungsengpässen führen, insbesondere wenn die Komponenten komplex sind. `Loader` eignet sich am besten für Anwendungen, bei denen Bildschirme zeitweise verwendet werden und nicht im Speicher gehalten werden müssen, wenn sie nicht mehr sichtbar sind.

2.3 Weiterführende Literatur

- Qt Dokumentation zum Loader: https://doc.qt.io/qt-6/qml-qtquick-loader.html

- Lazy loading in Qt Quick: https://doc.qt.io/qt-6/qtquick-performance.html#lazy-loading

3. Bildschirmwechsel basierend auf Sichtbarkeit

Ein anderer einfacher Ansatz besteht darin, alle Bildschirme auf einmal zu erstellen und ihre Sichtbarkeit mit der Eigenschaft „sichtbar“ zu verwalten.

3.1 Praktisches Beispiel

```qml
import QtQuick 2.15
import QtQuick.Controls 2.15

ApplicationWindow {
    visible: true
    width: 640
    height: 480
```

```
    Rectangle {
        id: screen1
        widt


h: parent.width
        height: parent.height
        color: "red"
        visible: true
    }

    Rectangle {
        id: screen2
        width: parent.width
        height: parent.height
        color: "blue"
        visible: false
    }

    Button {
        text: "Switch Screen"
        onClicked: {
            screen1.visible = !screen1.visible;
            screen2.visible = !screen2.visible;
        }
    }
}
```

In diesem Beispiel werden zwei Bildschirme erstellt und ihre Sichtbarkeit durch Klicken auf eine Schaltfläche umgeschaltet.

3.2 Vor- und Nachteile der sichtbaren Eigenschaft

Die Verwaltung von Bildschirmen mit Sichtbarkeit ist einfach und schnell zu implementieren. Allerdings verbraucht sie mehr Speicher, da alle Bildschirme immer im Speicher gehalten werden, was bei Anwendungen mit vielen Bildschirmen oder ressourcenintensiven Komponenten ein Nachteil sein kann.

Dieser Ansatz eignet sich für kleine bis mittelgroße Anwendungen, bei denen der Aufwand, alle Bildschirme im Speicher zu halten, vernachlässigbar ist.

3.3 Weiterführende Literatur

- Qt-Dokumentation zur Eigenschaftsbindung: https://doc.qt.io/qt-6/qtqml-syntax-propertybinding.html

4. Stapelbasierte Navigation mit „StackView“

„StackView“ bietet ein stapelbasiertes Navigationsmodell, bei dem Bildschirme aus dem Stapel geschoben und herausgezogen werden. Das vermag ein typisches Navigationsmuster nachzuahmen und ist vor allem in mobilen Anwendungen zu finden.

4.1 Praktisches Beispiel

```qml
import QtQuick 2.15
import QtQuick.Controls 2.15

ApplicationWindow {
    visible: true
    width: 640
    height: 480

    StackView {
        id: stackView
        anchors.fill: parent
        initialItem: Screen1 {}
    }

    Button {
        text: "Push Screen"
        onClicked: stackView.push(Qt.resolvedUrl("Screen2.qml"))
    }
}
```

Hier verwaltet ein `StackView` die Navigation zwischen den Bildschirmen. Die `initialItem`-Eigenschaft legt den ersten Bildschirm fest und zusätzliche Bildschirme werden mit der `push`-Methode auf den Stapel geschoben.

4.2 Vor- und Nachteile von StackView

„StackView“ bietet eine intuitive Möglichkeit zum Navigationsmanagement und verwaltet einen klaren Bildschirmverlauf, was für die Rücknavigation nützlich ist. Die Verwaltung komplexer Navigationsstapel kann jedoch mühsam werden und die Leistung beeinträchtigen, wenn sie nicht richtig gehandhabt wird.

Diese Methode wird für Anwendungen mit hierarchischen Navigationsstrukturen empfohlen, wie z. B. Einstellungsmenüs oder mehrstufige Prozesse, bei denen eine Rücknavigation erforderlich ist.

4.3 Weiterführende Literatur

- Qt Dokumentation zu StackView: https://doc.qt.io/qt-6/qml-qtquick-controls-stackview.html

- Umsetzung der Navigation mit StackView: https://doc.qt.io/qt-6/qtquickcontrols2-navigation.html

5. Wischbasierte Navigation mit „SwipeView“

„SwipeView“ bietet ein berührungsfreundliches Navigationsmodell, ideal für mobile Anwendungen oder Touch-Schnittstellen, bei denen Benutzer zwischen verschiedenen Bildschirmen wischen können.

5.1 Praktisches Beispiel

```qml
import QtQuick 2.15
import QtQuick.Controls 2.15

ApplicationWindow {
    visible: true
    width: 640
    height: 480

    SwipeView {
        anchors.fill: parent
        Rectangle { color: "red"; width: parent.width; height: parent.height }
        Rectangle { color: "blue"; width: parent.width; height: parent.height }
        Rectangle { color: "green"; width: parent.width; height: parent.height }
    }
}
```

In diesem Beispiel wird „SwipeView“ verwendet, um Benutzern das Wischen zwischen drei verschiedenen Bildschirmen zu ermöglichen, die als „Rechteck“-Komponenten mit unterschiedlichen Farben dargestellt werden.

5.2 Vor- und Nachteile von SwipeView

`SwipeView` is highly intuitive and easy to use, especially for mobile applications. However, it is less suited for desktop applications where swiping is not a common interaction method. `SwipeView` is ideal for mobile applications, image galleries, or any interface where quick and natural navigation between screens is desired.

5.3 Weiterführende Literatur

- Qt Dokumentation zu SwipeView: https://doc.qt.io/qt-6/qml-qtquick-controls-swipeview.html

6. Registerkartenbasierte Navigation mit „TabBar“

„TabBar“ bietet ein Navigationsmodell mit Registerkarten, bei dem Benutzer zwischen verschiedenen Registerkarten wechseln können, die jeweils einen anderen Bildschirm oder Abschnitt der Anwendung darstellen.

6.1 Praktisches Beispiel

```
import QtQuick 2.15
import QtQuick.Controls 2.15

ApplicationWindow {
    visible: true
    width: 640
    height: 480

    TabBar {
        id: tabBar
        width: parent.width
        TabButton {
            text: "Red"
            onClicked: stackView.replace(Qt.resolvedUrl("RedScreen.qml"))
        }
        TabButton {
            text: "Blue"
            onClicked: stackView.replace(Qt.resolvedUrl("BlueScreen.qml"))
        }
        TabButton {
            text: "Green"
            onClicked: stackView.replace(Qt.resolvedUrl("GreenScreen.qml"))
        }
    }

    StackView {
        id: stackView
        anchors.top: tabBar.bottom
        anchors.left: parent.left
        anchors.right: parent.right
        anchors.bottom: parent.bottom
        initialItem: RedScreen {}
    }
}
```

In diesem Beispiel wird `TabBar` in Verbindung mit `StackView` verwendet, um eine Oberfläche mit Registerkarten zu erstellen. Jede Registerkartenschaltfläche ersetzt den aktuellen Bildschirm in `StackView` durch einen neuen.

6.2 Vor- und Nachteile von TabBar

`TabBar` bietet ein vertrautes und einfach zu verwendendes Navigationsmodell und ist daher ideal für Desktop-Anwendungen oder Anwendungen mit klar definierten Abschnitten. Es ist jedoch möglicherweise nicht so geeignet für Anwendungen mit einer großen Anzahl von Bildschirmen oder für solche, die einen dynamischeren Navigationsfluss erfordern. `TabBar` ist perfekt für Einstellungsseiten, Dashboards oder Anwendungen mit einigen primären Abschnitten.

6.3 Weiterführende Literatur

- Qt Dokumentation zu TabBar: https://doc.qt.io/qt-6/qml-qtquick-controls-tabbar.html

- Und zu TabButton: https://doc.qt.io/qt-6/qml-qtquick-controls-tabbutton.html

7. Fazit

Die Wahl der jeweils passenden Lösung für das Management von Bildschirmen in Qt 6 hängt von den spezifischen Anforderungen Ihrer Anwendung ab. Die dynamische Erstellung mit `component.createComponent` und `component.createObject` bietet Flexibilität und Effizienz für dynamische Schnittstellen. `Loader` bietet eine einfache Möglichkeit, die Speichernutzung zu verwalten, indem Bildschirme bei Bedarf geladen werden. Sichtbarkeitsbasiertes Umschalten ist schnell zu implementieren, kann aber mehr Speicher verbrauchen. `StackView` ist ideal für Anwendungen mit hierarchischer Navigation. `SwipeView` und `TabBar` bieten intuitive Navigationsmodelle für mobile bzw. Desktopanwendungen.

Lassen Sie sich auf diese Methoden ein. Das Qt Team von helloQt hilft Ihnen sehr gerne bei der Umsetzung. Dann können Sie effizientere und reaktionsschnellere Qt 6-Anwendungen erstellen, die auf die individuellen Anforderungen Ihrer Projekte zugeschnitten sind.