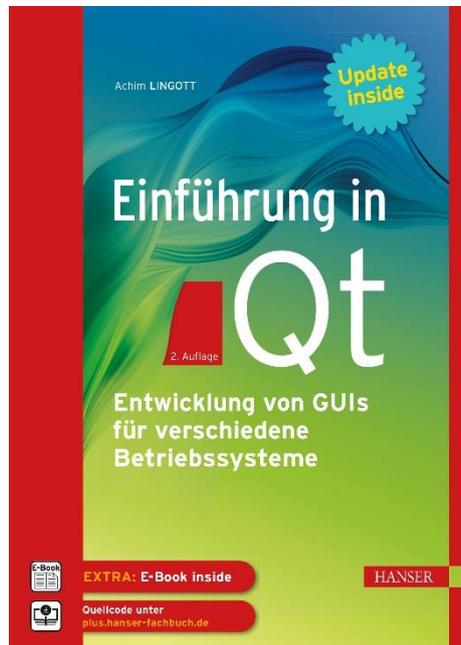


HANSER



Leseprobe

zu

Einführung in Qt

von Achim Lingott

Print-ISBN: 978-3-446-47610-3
E-Book-ISBN: 978-3-446-47784-1
E-Pub-ISBN: 978-3-446-47785-8

Weitere Informationen und Bestellungen unter
<https://www.hanser-kundencenter.de/fachbuch/artikel/9783446476103>

sowie im Buchhandel

© Carl Hanser Verlag, München

Inhalt

Vorwort	IX
Updates und Aktualität	X
Danksagung	X
1 Einführung und erste Schritte	1
1.1 Download und Installation	2
1.2 Die Qt-Bibliothek	4
1.2.1 Die Klassen der Bibliothek	5
1.2.2 Die Module der Bibliothek	5
1.3 Die installierten Programme	7
1.4 Im Qt Creator erstellte Dateien und ihre Bedeutung	10
1.4.1 Datei CMakeLists.txt	10
1.4.2 Dateien .pro und .pri	11
1.4.3 Datei .ui	12
1.4.4 Datei ui_mainwindow.h	12
1.4.5 Datei main.cpp	13
1.5 Der Qt Designer	14
1.6 Das Signal-Slot-Prinzip	15
1.7 Qt in Microsoft Visual Studio	16
2 Das Erstellen von Qt-Widgets-Applikationen	22
2.1 Unterschiedliche Oberklassen	22
2.2 Eine Auswahl von Widgets	23
2.3 Qt-Widgets-Anwendung mit dem Qt Designer	26
2.3.1 Signal- und Slot-Funktionen miteinander verbinden	29
2.4 Erstellen ohne Qt Designer	33
2.5 Die Benutzung von Layouts	35
2.6 Das Erstellen und die Funktion von Menüs	37

2.7	Ein Beispiel mit QTabWidget	39
2.8	Ein zweites Formular hinzufügen	42
2.9	Widgets selbst erstellen	45
2.10	Maus- und Tastatur-Events	49
2.11	Shortcuts für die Bedienung des Qt Creators	51
2.12	Von den Programmen auszugebende Meldungen	52
2.13	Animationen mit Qt	54
3	Daten, Variablen und ihre Benutzung in Qt	56
3.1	Qt-Datentypen	56
3.2	Das Model-View-Prinzip und seine Realisierung	57
3.3	Das Qt-Event-System	60
3.4	Qt-Containerklassen	61
3.5	Die Speicherverwaltung in Qt	62
3.6	Multithreading in Qt	64
3.7	Die Klasse QVariant	68
4	Zeichnen in Widget-Applikationen	71
4.1	Grundlagen des Zeichnens	71
4.2	Zeichnen auf ein Fenster	72
4.3	Zeichnen auf ein Widget	73
4.4	Freie Zeichnungen mit der Maus auf ein Fenster	79
4.5	Charts	81
4.6	SVG	82
4.7	Transformation	84
4.8	Farbverläufe bei Füllungen	86
4.9	Zeichnen mit QGraphicsView auf QGraphicsScene	88
5	Ressourcen in Qt	89
5.1	Das Erstellen einer Ressourcendatei	89
5.2	Die Benutzung von StyleSheets	92
5.3	Die Verwendung von RTF und HTML	98
5.4	Lokalisierung von Qt-Applikationen	101
5.5	Dynamischer Wechsel der Sprache	105

6	Datenbankanbindung an Qt-Applikationen	108
6.1	SQL und Datenbankdesign ganz kurz	108
6.2	Eine Datenbankanwendung	110
6.3	Verbindung zu einem MySQL-Datenbankserver	113
6.4	Verbindung zu einem Microsoft SQL Server	118
6.5	Verbindung zu einer MS Access-Datenbank	119
6.6	Lesen von Daten aus einer Tabelle	120
6.7	Einen neuen Datensatz eintragen	122
7	Drucken und Dateibearbeitung	125
7.1	Grundlagen des Druckvorgangs	125
7.2	Ausdrucken von Text und Bildern	126
7.3	Dateien lesen und schreiben	132
7.4	Texteditor	135
8	Qt Quick und QML	137
8.1	Das Erstellen einer Qt-Quick-Anwendung	138
8.2	Der Quick Designer	141
8.3	QML-Typen	146
8.4	JavaScript in QML	147
8.5	Canvas	151
8.6	QML 3D	158
8.7	QML Charts	161
8.8	Mediaplayer	165
8.9	Lokalisierung von Quick-Applikationen	167
8.10	Animationen mit QML	169
8.11	Zustände und ihre Übergänge	175
8.12	Das Zustandsdiagramm und der Zustandseditor	177
8.13	QML-Applikationen als GUI für C++-Klassen	180
8.14	QML-Datentypen	180
8.15	Signale von und zu QML-Oberflächen	181
8.16	Drucken über eine QML-Oberfläche	182
8.17	Datenaustausch QML-GUI und C++-Klasse	185
8.18	Datenbankanbindung über ein QML-GUI	187

9	Qt-Applikationen für andere Plattformen	191
9.1	Qt-Applikationen für Android	191
9.2	Qt-Applikationen für WebAssembly	196
9.3	Qt-Applikationen für Linux	200
9.3.1	Qt und Ubuntu	201
9.3.2	Qt und openSUSE	203
9.3.3	Qt und CentOS	205
9.3.4	Qt und iOS	205
10	Client-Server-Applikationen	206
10.1	TCP-Server und -Client	206
10.2	D-Bus	210
10.3	CAN-Bus	211
10.4	DTLS	211
10.5	Bluetooth	212
11	Verschiedenes	217
11.1	qmake-Projekte in CMake-Projekte wandeln	217
11.2	Ein Projekt mit mehreren Unterprojekten	218
11.3	Exceptions in Qt	220
11.4	Debugging von Qt- und QML-Anwendungen	220
11.4.1	Qt-Anwendungen	220
11.4.2	QML-Anwendungen	224
11.5	Dokumentation	224
11.5.1	Qt-eigene Dokumentationsmöglichkeit	225
11.5.2	Dokumentation mit Doxygen	225
11.6	Deployment von Qt unter Windows	227
11.7	Qt und OpenGL	229
11.8	Desktop Styling der Qt Quick Controls	232
	Literatur	235
	Index	237

Vorwort

Dieses Buch vermittelt Einsteigern mit C++-Vorkenntnissen die Grundlagen der Qt-Programmierung. Mit der hervorragenden Qt-Bibliothek lassen sich grafische User Interfaces für die unterschiedlichsten Anwendungsfälle programmieren.

Die 1. Auflage dieses Buches beruhte auf der damals gerade erschienenen Version 6.0 der Qt-Bibliothek. Dort waren noch nicht alle Module der vorhergehenden Version 5 enthalten, und es musste bei einigen Beispielen auf die Version Qt 5 zurückgegriffen werden. Das ist heute nicht mehr so.

Das Manuskript zur 2. Auflage wurde auf der Grundlage der Bibliotheksversion Qt 6.4 erstellt.

Ebenso werden alle Programme mit CMake erstellt, anstelle des in der 1. Auflage verwendeten qmake.

Einige Quelltexte sind in der vorliegenden 2. Auflage nicht mehr (oder nur in Ausschnitten) im gedruckten Buch zu finden, sondern auf dem Download-Portal. Sehen Sie sich also auf alle Fälle die vollständigen Quelltexte an, um die Beispiele auch zu verstehen.



Ihr Plus – digitale Zusatzinhalte!

Auf unserem Download-Portal finden Sie zu diesem Titel kostenloses Zusatzmaterial.

Geben Sie auf plus.hanser-fachbuch.de einfach diesen Code ein:

`plus-12abc-8xyz9`

Auf dieser Webseite sind alle Quelltexte für die Projekte zu finden, die im Buch beschrieben werden.

■ Updates und Aktualität

Damit Sie möglichst lange mit diesem Buch arbeiten können, haben Sie die Möglichkeit, sich für den kostenlosen Update-inside-Service zu registrieren: Geben Sie unter

www.hanser-fachbuch.de/qt-update

diesen Code ein:

plus-12abc-8xyz9

Dann erhalten Sie bis Februar 2025 Aktualisierungen in Form zusätzlicher Kapitel als PDF-Datei. Darin stelle ich Ihnen wichtige Neuerungen vor und gehe auf Änderungen ein, die die Inhalte dieses Buches betreffen.

■ Danksagung

An dieser Stelle möchte ich allen Beteiligten einen großen Dank aussprechen. Er gilt allen beteiligten Mitarbeitern des Carl Hanser Verlages, insbesondere Frau Sylvia Hasselbach und Frau Kristin Rothe, die sich stets allen meinen Fragen zu Inhalt und Gestaltung gestellt haben und hilfreich mit Ideen zur Seite standen.

Diesen Dank möchte ich aber auch an diejenigen richten, die, vermutlich ohne es zu ahnen, zur einen oder anderen Idee beigetragen haben: Das sind die Teilnehmer meiner Seminare, deren Fragen mich dazu anregen, das eine oder andere Problem etwas umfassender oder überhaupt anzugehen.

Und nun viel Spaß beim Lesen und bei der Arbeit mit Qt!

Achim Lingott

Der Verlag und die Autoren haben sich mit der Problematik einer gendergerechten Sprache intensiv beschäftigt. Um eine optimale Lesbarkeit und Verständlichkeit sicherzustellen, wird in diesem Werk auf Gendersternenchen und sonstige Varianten verzichtet; diese Entscheidung basiert auf der Empfehlung des Rates für deutsche Rechtschreibung. Grundsätzlich respektieren der Verlag und die Autoren alle Menschen unabhängig von ihrem Geschlecht, ihrer Sexualität, ihrer Hautfarbe, ihrer Herkunft und ihrer nationalen Zugehörigkeit.

8

Qt Quick und QML

Nicht für jeden gestandenen Programmierer ist Qt Quick eine tolle Sache. Manche mögen es nicht, weil die Syntax nichts mit C++ zu tun hat. Mit der Version 4.7 des Qt Frameworks wurde Qt Quick eingeführt und bietet eine vollkommen andere Möglichkeit der GUI-Erstellung. Wurde ein GUI bisher mit C++ erstellt, so wie Sie es bisher auch kennengelernt haben, kann das jetzt auch mit einer eigens dafür eingeführten deklarativen Skriptsprache, QML (Qt Modeling Language), geschehen. Diese Sprache hat eine JSON-ähnliche Syntax. Mit ihr hat auch die Anwendung von JavaScript Einzug gehalten. Voraussetzung ist die Benutzung des Moduls QtQuick. Es enthält die Standardbibliothek zum Schreiben von QML-Anwendungen. Im QtQML-Modul wird die QML Engine u. a. bereitgestellt.

Das QtQuick-Modul enthält sowohl QML-Typen zum Erstellen von Benutzeroberflächen mit QML als auch die Möglichkeit, QML-Anwendungen mit C++ zu erweitern.

Eine Übersicht der *Qt Quick Controls* finden Sie unter:

<https://doc.qt.io/qt-6/qtquick-controls2-qmlmodule.html>

Alle QML-Module werden unter

<https://doc.qt.io/qt-6/modules-qml.html>

vorge stellt.

Eine Gesamtübersicht der QML-Typen finden Sie hier:

<https://doc.qt.io/qt-6/qmltypes.html>

Der generelle Zugang zu allen Informationen ist immer

<https://doc.qt.io/qt-6/>

■ 8.1 Das Erstellen einer Qt-Quick-Anwendung

Erstellen Sie ein neues Projekt. Es wird im Qt Creator als *Qt-Quick-Application* bezeichnet. Es soll *QuickTest* heißen.

Das Build-System ist wieder *CMake*. Es sind nun einige Veränderungen gegenüber Qt-Widget-Anwendungen zu sehen.

- In der Datei *CMakeLists.txt* ist das Modul *QtQuick* eingetragen.

```
find_package(QT NAMES Qt6 Qt5 REQUIRED COMPONENTS Core Quick)
find_package(Qt${QT_VERSION_MAJOR} REQUIRED COMPONENTS Core Quick)
...
target_link_libraries(QuickTest PRIVATE Qt${QT_VERSION_MAJOR}::Core
Qt${QT_VERSION_MAJOR}::Quick)
```

- Der Projektordner enthält automatisch eine Ressourcendatei, und es ist eine Datei *main.qml* enthalten.

Ansonsten ist die einzige enthaltene C++-Datei die Datei *main.cpp*.

```
01 #include <QGuiApplication>
02 #include <QQmlApplicationEngine>
03
04
05 int main(int argc, char *argv[])
06 {
07     #if QT_VERSION < QT_VERSION_CHECK(6, 0, 0)
08         QCoreApplication::setAttribute(Qt::AA_EnableHighDpiScaling);
09     #endif
10     QGuiApplication app(argc, argv);
11
12     QQmlApplicationEngine engine;
13     const QUrl url(QStringLiteral("qrc:/main.qml"));
14     QObject::connect(&engine, &QQmlApplicationEngine::objectCreated,
15                     &app, [url](QObject *obj, const QUrl &objUrl) {
16         if (!obj && url == objUrl)
17             QCoreApplication::exit(-1);
18     }, Qt::QueuedConnection);
19     engine.load(url);
20
21     return app.exec();
22 }
```

Ihr Quelltext sieht aufregender aus, als er wirklich ist. Die entscheidenden Zeilen sind das Erstellen einer Instanz von *QGuiApplication* (Zeile 10), die wieder mit *exec()* in einem *event loop* gestartet wird (Zeile 21).

Die *connect()*-Funktion (Zeilen 14 - 18) aus der Klasse *QObject* ist eine statische Funktion, die insgesamt fünf Parameter hat. Der letzte besitzt einen Default-Wert. Dieser letzte Parameter kann eine Information zum Connection-Typ enthalten. Dazu gibt es einige Bemerkungen in Kapitel 1. Diese Funktion wird hier mit fünf Parametern aufgerufen:

1. Das Objekt, das das Signal aussendet (hier *engine*),
2. die Adresse der sendenden Funktion (hier *objectCreated()*),

3. das Objekt, das das Signal empfängt (hier *app*),
4. die Adresse der empfangenden Funktion (das ist hier eine Lambda-Funktion, also anonym),
5. der Connection-Typ (hier *QueuedConnection* aus dem Namespace *Qt*).

Das Erzeugen einer Instanz von *QqmlApplicationEngine* (Zeile 12) und das Laden der in den Ressourcen befindlichen Datei *main.qml* (Zeile 13) ermöglichen die Auswertung der in *main.qml* befindlichen QML-Typen.

Listing 8.1 Die Datei *main.qml*

```
01 import QtQuick
02 import QtQuick.Window
03
04 Window {
05     width: 640
06     height: 480
07     visible: true
08     title: qsTr("Hello World")
09 }
```

Diese Datei enthält den Quellcode zur Erzeugung eines Formulars.

- Es werden die Module *QtQuick* und *QtQuick.Window* importiert. Die Versionsangaben hinter den Modulnamen waren bis Qt 5.15 notwendig. Bei Qt 6 können Sie entfallen. Dann wird die jeweils höchste erreichbare Version benutzt.
- Es wird ein Hauptfenster mit dem QML-Typ *Window* erstellt. Die hier angegebenen Eigenschaften *width* und *height* bestimmen mit ihren Werten die Größe des Fensters. Der Wert des QML-Typs *title* enthält eine Zeichenkette, die Parameter einer Funktion *qsTr()* ist. Diese Funktion wird in QML zur Übersetzung benutzt (ähnlich *tr()* in Qt-Anwendungen). Wenn Sie unter QML-Typen den QML-Typ *Window* ansehen (Adresse Seite 1), werden weitere Eigenschaften angezeigt.
- Mit *x* und *y* könnten Sie die Koordinaten der linken oberen Ecke Ihres Fensters auf dem Bildschirm bestimmen.
- *color* würde die Hintergrundfarbe bestimmen.
- Alle weiteren QML-Typen, die Sie auf dem Fenster anzeigen möchten oder die ansonsten Bestandteil des Formulars sein sollen, werden innerhalb des Blockes von *Window* angeordnet (also zwischen den Zeilen 4 und 9).

Das Projekt *QuickTest* soll nun so ausgebaut werden, dass es einen QML-Typ *Button*, ein *TextEdit* und ein *Label* bekommt. Nach Klick auf den *Button* soll der Text, der eventuell im *TextEdit* steht, in das *Label* geschrieben werden. Dazu erweitern Sie die Datei *main.qml*. Um die QML-Typen später ansprechen zu können, erhalten alle eine Eigenschaft *id*, eine in der Datei eindeutige Kennung.

In welcher Reihenfolge die QML-Typen in die Datei *main.qml* geschrieben werden, ist im Prinzip egal. Sie werden ja jeweils durch die Angaben von *x* und *y* positioniert (das sind immer die Koordinaten der linken oberen Ecke). Es sei denn, dass sich die Elemente überdecken. Dann ist das Element oben zu sehen, das später in die Datei geschrieben wurde.

Der Qt-Bibliothek können Sie unter *All QML Types* die Informationen zu den einzelnen Elementen entnehmen. Gehen Sie z. B. zum QML-Typ `Button`, dann sehen Sie, dass er abgeleitet ist von `AbstractButton`. Auch hier gibt es wieder einen Link *List of all members, including inherited members*. Klicken Sie darauf, sehen Sie auch hier wieder alle Eigenschaften, die Sie benutzen können, sowie einige Funktionen wie zum Beispiel `clicked()` oder `pressed()`.

Listing 8.2 Die Datei `main.qml` nach der Erweiterung

```

01 import QtQuick
02 import QtQuick.Window
03 import QtQuick.Controls
04
05 Window {
06     id: root
07     x: 100
08     y: 100
09     width: 300
10     height: 400
11     visible: true
12     color: "#ffff00"
13     title: qsTr("Quick Test")
14
15     Label {
16         id: label1
17         x: 50
18         y: 70
19         font.pointSize: 14
20         font.bold: true
21         text: "Label"
22         color: "green"
23     }
24
25     TextInput {
26         id: text1
27         x: 50
28         y: 200
29         font.pointSize: 14
30         text: "TextInput"
31         color: "red"
32     }
33
34     Button {
35         id: button1
36         x: 50
37         y: 300
38         text: "OK"
39         onClicked: label1.text = text1.text
40     }
41 }

```

Das Hauptfenster erscheint auf dem Bildschirm unter den angegebenen Koordinaten mit der genannten Größe. Es wird als Hintergrundfarbe *Gelb* gewählt, und das Fenster wird mit *Quick Test* beschriftet (Zeilen 6 – 13). Wenn Sie den Mauszeiger übrigens im Qt Creator über den Farbnamen stellen (ohne zu klicken), wird die Farbe angezeigt. Dann kann man insbesondere bei Mischfarben sehen, ob man die richtige Wahl getroffen hat.

Label (Zeilen 15 – 23)

Die Zeile 19 gibt an, dass eine Schriftgröße von 14 pt gewählt wurde. Und die Zeile 20 setzt die Schrift auf *bold*. Die Zeile 22 setzt die Schriftfarbe auf *Grün*.

TextEdit (Zeilen 25 – 32)

Es wird wieder die Schriftgröße auf 14 pt gesetzt, und die Schriftfarbe ist *Rot* (Zeile 29 und 31).

Button (Zeilen 34 – 40)

Im Button ist insbesondere die Zeile 39 interessant. Der QML-Typ besitzt eine Funktion *clicked()*. Sie wird durch den Handler *onClicked* aufgerufen. Durch Klick auf den Button wird der Text des QML-Typs TextEdit (mit der id *text1*) der Eigenschaft *text* des QML-Typs Label (mit der id *label1*) zugewiesen.

Nun soll das Programm so erweitert werden, dass bei Klick auf den Button gleichzeitig die Hintergrundfarbe des Formulars geändert wird. Es sollen nach *onClicked* also mehrere Kommandos kommen. Deshalb ist es notwendig, nach dem Doppelpunkt einen Block aus geschweiften Klammern zu schreiben.

```
Button {
    id: button1
    x: 50
    y: 300
    text: "OK"
    onClicked: {
        label1.text = text1.text
        root.color = "blue"
    }
}
```

In QML muss nicht am Ende einer Zeile ein Semikolon geschrieben werden. Falls Sie aber in einer Zeile mehrere Eigenschaften und Werte hintereinanderschreiben wollen, ist ein Semikolon zur Trennung notwendig.

■ 8.2 Der Quick Designer

Auch QML-Dateien können Sie in einem Design-Modus bearbeiten.

Evtl. müssen Sie den Quick Designer aber erst aktivieren. Das geschieht im Hauptmenü über *Hilfe/Plugins*. Wählen Sie im dann erscheinenden Fenster *Qt Quick* aus und aktivieren Sie den *QmlDesigner*. Danach muss der Qt Creator neu gestartet werden.

Wählen Sie dann im Projektextplorer des Qt Creator die Datei *main.qml* aus. Klicken Sie zum Erreichen des Designmodus links oben im Qt Creator auf den Button *Design* (Bild 8.1).

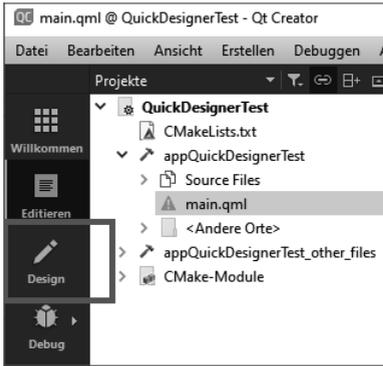


Bild 8.1
Einschalten des Designmodus für
Qt Quick-Anwendungen

Sie bekommen das Design der erstellten QML-Datei zu sehen, können bestimmte Elemente markieren und dann verschieben oder anderweitig verändern. Dazu werden die Eigenschaften des markierten Elements rechts angezeigt

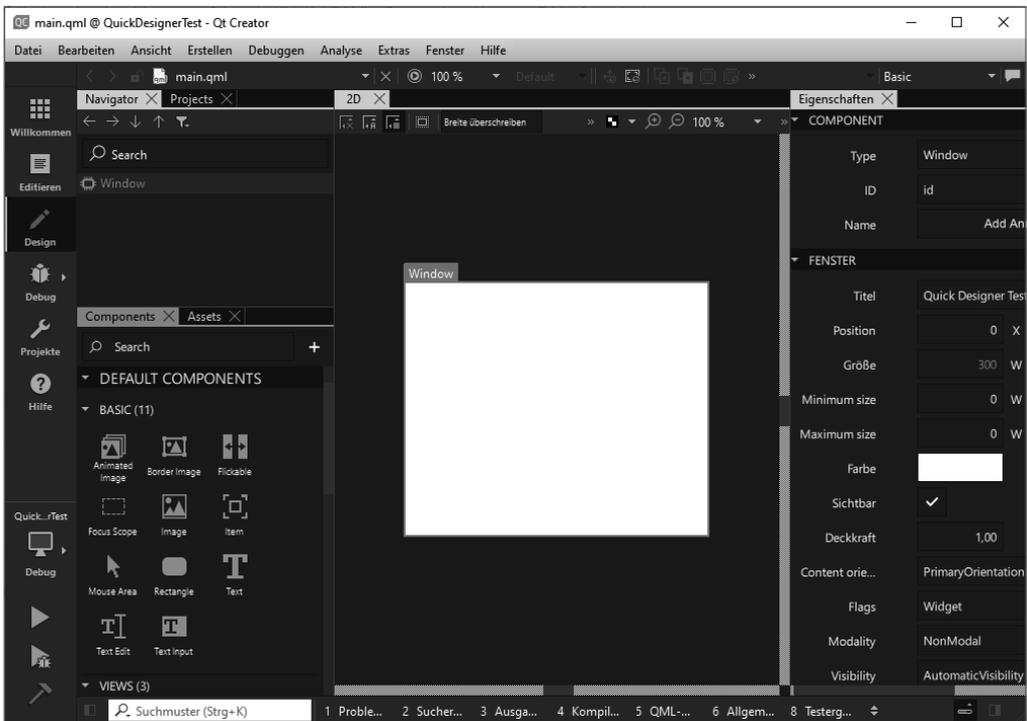


Bild 8.2 Der Quick Designer bei leerem Hauptfenster des Programms

```
import QtQuick

Window {
    width: 300
    height: 250
    visible: true
}
```

```

title: qsTr("Quick Designer Test")
}

```

Bei einer leeren QML-Datei ohne zusätzliche Importe, wie hier zu sehen, werden im Quick Designer auch nur die *DEFAULT COMPONENTS* zur Verfügung gestellt (Bild 8.2). Wenn der QML-Typ Window markiert ist, sehen Sie im rechten Teil des Quick Designers im *Eigenschaften*-Fenster alle Eigenschaften des Hauptfensters und können diese hier auch verändern. Die veränderten Eigenschaften werden sofort in die Datei *main.qml* eingetragen (und umgekehrt).

Möchten Sie in Ihrer Datei *main.qml* auch einen Button verwenden und diesen mit dem Quick Designer erstellen, tragen Sie die Zeile `import QtQuick.Controls` ein.

```

import QtQuick
import QtQuick.Controls

Window {
    width: 300
    height: 250
    ...
}

```

Dann ändert sich die Ansicht des Quick Designers wie folgt. Jetzt stehen auch sämtliche *QTQUICK CONTROLS* zur Verfügung. Markieren Sie ein Objekt im Designmodus, sehen Sie im rechten Fenster wieder seine Eigenschaften (Bild 8.3).

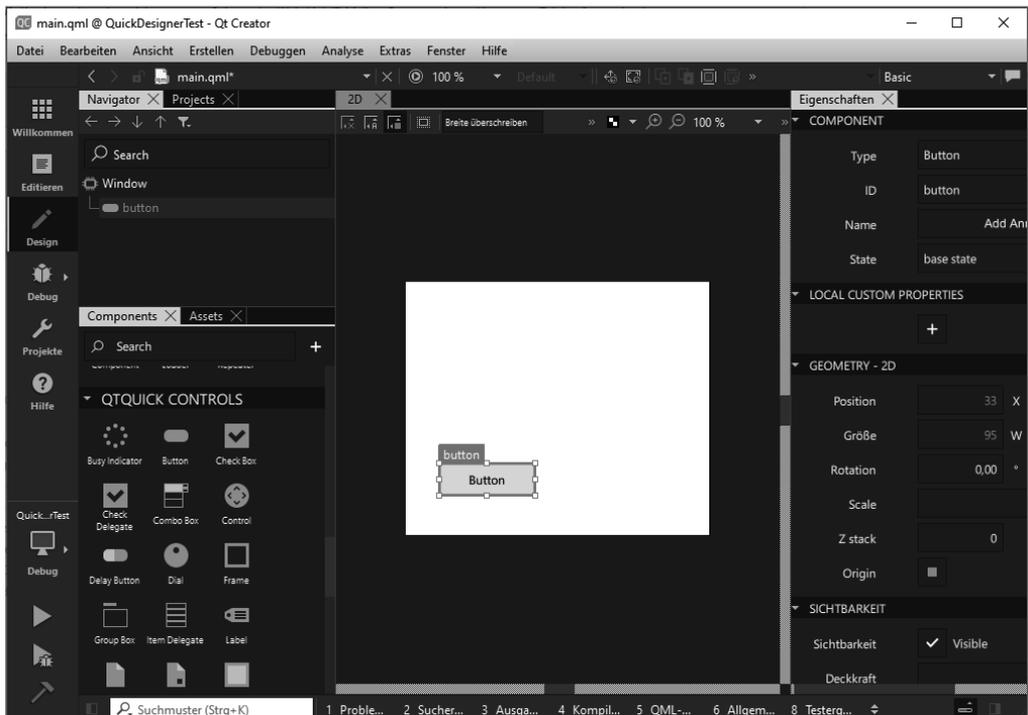


Bild 8.3 Der erweiterte Quick Designer bei importierten QtQuick Controls

Die erzeugte Datei *main.qml* sieht jetzt so aus:

```
import QtQuick
import QtQuick.Controls

Window {
    width: 300
    height: 250
    visible: true
    title: qsTr("Quick Designer Test")

    Button {
        id: button
        x: 33
        y: 179
        width: 95
        height: 32
        text: qsTr("Button")
    }
}
```

Ebenso können Sie weitere Module importieren, wie hier z. B. `QtMultimedia`.

```
import QtQuick
import QtQuick.Controls
import QtMultimedia
```

Im Designer werden wieder die entsprechenden Komponenten zur Verfügung gestellt, lassen sich verwenden und ihre Eigenschaften können bearbeitet werden (Bild 8.4).

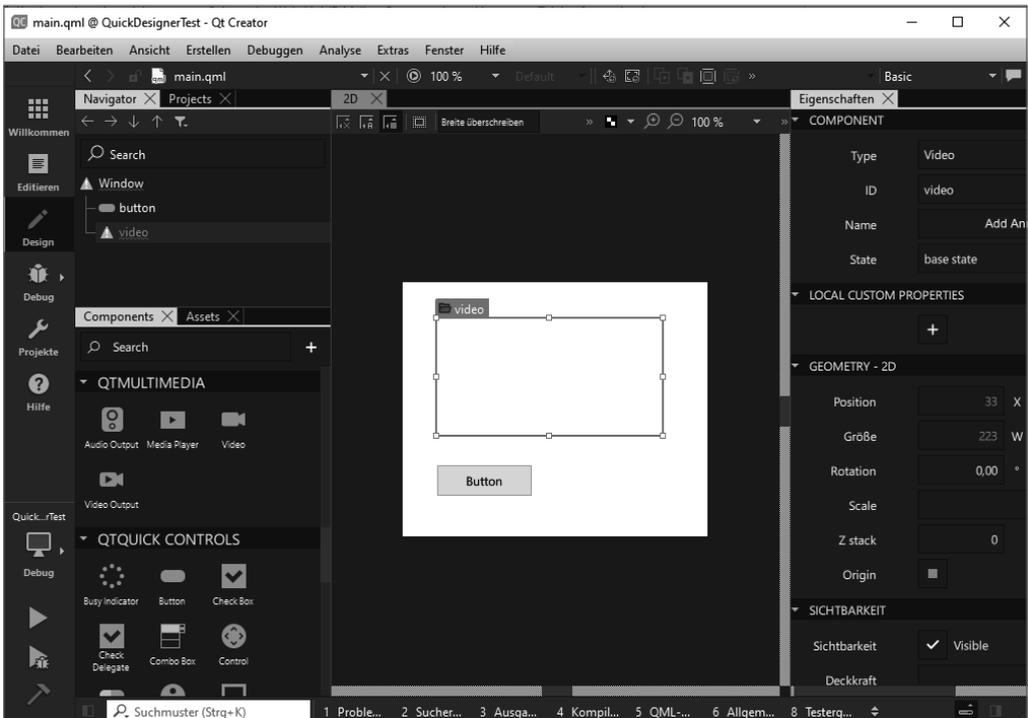


Bild 8.4 Der Quick Designer bei erweiterten Importen

Die entstehende Datei `main.qml` sieht jetzt so aus:

Listing 8.3 Die mit dem Quick Designer erzeugte Datei `main.qml`

```
01 import QtQuick
02 import QtQuick.Controls
03 import QtMultimedia
04
05 Window {
06     width: 300
07     height: 250
08     visible: true
09     title: qsTr("Quick Designer Test")
10
11     Button {
12         id: button
13         x: 33
14         y: 179
15         width: 95
16         height: 32
17         text: qsTr("Button")
18     }
19
20     Video {
21         id: video
22         x: 33
23         y: 35
24         width: 223
25         height: 116
26     }
27 }
```

In der linken oberen Ecke des Quick Designer können Sie den Reiter *Projects* auswählen, erhalten dann die Ansicht des Projekteexplorers des Qt Creator und können diese Dateien von hier aus auch aufrufen.

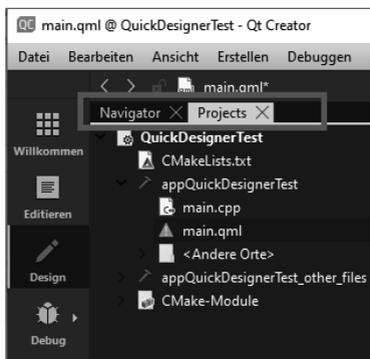


Bild 8.5

Der aktivierte Reiter *Projects* im Quick Designer

Über das Hauptmenü des Qt Creator und *Ansicht/Ansichten* lassen sich verschiedene Ansichten des Designers hinzufügen oder entfernen.

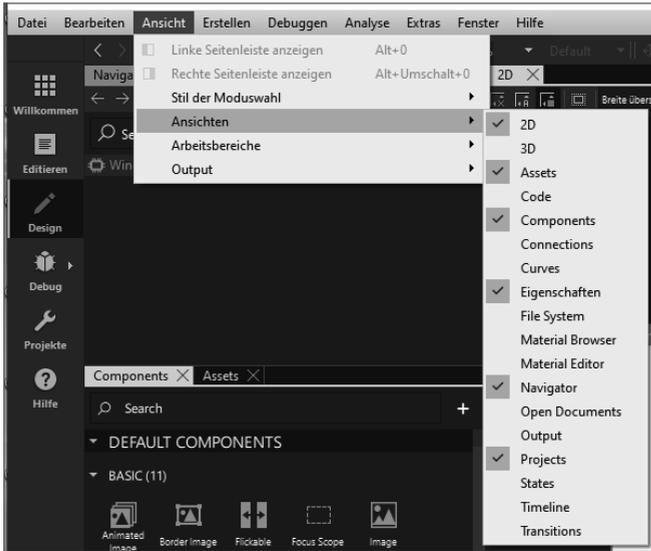


Bild 8.6
Die Auswahl der Ansichten
im Quick Designer

■ 8.3 QML-Typen

In QML existieren Werttypen (Value Types), also Datentypen, die als Wert und nicht als Referenz benutzt werden, z. B. *int* oder eine Zeichenfolge sowie Objekttypen, die per Referenz benutzt werden. Einige Werttypen erfordern keine Importanweisung, weil sie standardmäßig unterstützt werden, andere erfordern, dass das entsprechende Modul importiert wird.

Folgende Ausnahmen müssen beachtet werden:

- Eine Liste muss in Verbindung mit einem QML-Objekt stehen.
- Enums können nicht direkt verwendet werden, sondern müssen über einen QML-Objekttypen definiert werden.

Folgende Typen werden von QML direkt unterstützt:

```
bool, double, int, list, real, string, url, var
```

Einige QML-Module erweitern diese Liste mit:

```
color, date, font, matrix4x4, point, quaternion, rect, size, vector2d, vector3d, vector4d
```

Einige Werttypen besitzen Eigenschaften. So besitzt z. B. der Typ *font* die Eigenschaften *pixelSize*, *italic* oder *underline*. Diese Eigenschaften von Werttypen emittieren keine eigenen Signale bei ihrer Änderung. Sie müssen bei Bedarf einen Signal-Handler selbst schreiben. So existiert z. B. *onFontChanged*, aber nicht *onPointSizeChanged*.

Das globale Objekt QtQML enthält zahlreiche Funktionen zur Manipulation der Werte von Werttypen. Sie erreichen es in der Bibliothek über <https://doc.qt.io/qt-6/qml-qtqml-qt.html>.

■ 8.4 JavaScript in QML

Im QtQML-Modul werden die Grundlagen gelegt, um QML und JavaScript zu verbinden. Es können gültige JavaScript-Standardkonstrukte wie Operatoren, Arrays, Schleifen etc. ausgeführt werden. Zusätzlich wird noch eine Reihe von Hilfsmethoden für die Interaktion mit QML angeboten. Die von QML bereitgestellte JavaScript-Umgebung reagiert noch weiter eingeschränkt als die eines Webbrowsers.

- Variablen müssen vor ihrer Benutzung unbedingt deklariert werden. Eine Benutzung nichtdeklarerter Variablen ist grundsätzlich nicht zulässig.
- Eine JavaScript-Datei, die von einer QML-Datei eingelesen wird, hat keinen Zugriff auf die QML-Objekte und -Eigenschaften.
- Das Schlüsselwort *this* ist in der Regel in QML undefiniert, es sei denn, es bezieht sich auf den direkten Bereich, in dem sich die JavaScript-Funktion befindet.

Verschiedene Teile von QML-Dokumenten können JavaScript-Code enthalten.

- Mithilfe von JavaScript kann auf Eigenschaften eines QML-Typs zugegriffen werden. So kann z. B. ein QML-Typ `MouseArea` die Breite und Höhe des Hauptfensters erhalten:

```
MouseArea {
    id: ma
    width: parent.width
    height: parent.height
    ...
}
```

parent bezieht sich dabei auf das Elternelement. Anstelle von *parent* könnte auch die *id* eines anderen QML-Typs stehen.

- Es sind in QML JavaScript-Statements enthalten, die automatisch aufgerufen werden, sobald ein Signal gesendet wird. Der QML-Typ `MouseArea` enthält z. B. die Signal-Funktionen `clicked()` und `doubleClicked()`. Sie werden durch die Handler `onClicked` und `onDoubleClicked` aktiviert.
- JavaScript-Funktionen können im Body eines QML-Dokumentes enthalten sein.

Eine sehr häufige Anwendung ist die Benutzung von separaten JavaScript-Dateien, die mit der Endung `.js` in die Ressourcen eingebunden werden.

Wir erstellen ein Beispielprojekt für die Anwendung von JavaScript in QML. Erzeugen Sie mit dem Qt Creator ein Projekt *JavaScript*. Es ist eine Qt-Quick-Anwendung. Wir beginnen mit diesem Inhalt:

Listing 8.4 *main.qml* des Projekts *JavaScript*

```
01 import QtQuick
02 import QtQuick.Window
03 import QtQuick.Controls
04
05 Window {
06     width: 300
07     height: 300
08     visible: true
```

```

09     title: qsTr("JavaScript")
10
11     Label {
12         id: label1
13         x: 50
14         y: 50
15         font.pointSize: 18
16         text: "xxxxxxxxxxxxx"
17         color: "red"
18     }
19
20     TextEdit {
21         id: tel
22         x: 50
23         y: 150
24         font.pointSize: 18
25         text: "vvvvvvvvvvvvv"
26         focus: true
27     }
28
29     MouseArea {
30         id: ma
31         anchors.fill: parent
32         onClicked: label1.text = 10 + 5
33     }
34 }

```

Die `MouseArea` wird so groß gewählt, wie die Größe des Formulars eingestellt ist. Bei Klick irgendwo auf das Fenster wird dem `Label` der Text rechts vom Gleichheitszeichen in Zeile 32 zugewiesen. Dabei können Sie schon erkennen, dass bei zwei Zahlen, die durch ein `+` verbunden werden, die Summe dieser Zahlen übergeben wird. Ist einer dieser Summanden ein String, wird das Ergebnis auch ein String sein, z. B. in Zeile 32

```
onClicked: label1.text = „10“ + 5
```

ergibt im `Label` die Anzeige „105“.

Das gleiche Problem entsteht natürlich, wenn eine in das `TextEdit` eingetragene Zahl mit einer anderen addiert werden soll. Die Zeile 32 sieht jetzt so aus:

```
onClicked: label1.text = tel.text + 5
```

Natürlich wird auch wieder nicht einfach addiert, denn der Datentyp der Eigenschaft `text` des QML-Typs `TextEdit` ist `string`, ein *QML-Basis-Typ* (siehe Abschnitt 8.3).

Falls Sie diese Addition im Quellcode öfter durchführen müssen, bietet es sich an, eine Funktion zu schreiben, das wäre hier eine JavaScript-Funktion. (Natürlich ist sie hauptsächlich dann sinnvoll, wenn umfangreiche Aufgaben (wie Rechnungen) durchgeführt werden sollen.)

Listing 8.5 *main.qml* mit JavaScript-Funktion

```

01 import QtQuick 2.15
02 import QtQuick.Window 2.15
03 import QtQuick.Controls 2.5
04

```

```
05 Window {
06     width: 400
07     height: 300
08     visible: true
09     title: qsTr("JavaScript")
10
11     function f1(wert1, wert2)
12     {
13         return wert1 + " " + "wohnt in " + wert2;
14     }
15
16     function f2(wert1)
17     {
18         return "A" + wert1 + "X"
19     }
20
21     Rectangle {
22         id: rect
23         x: 50
24         y: 200
25         width: 50
26         height: 50
27         color: "red"
28     }
29
30     Label {
31         id: label1
32         x: 50
33         y: 50
34         font.pointSize: 18
35         text: "xxxxxxxxxxxxx"
36         color: "red"
37     }
38
39     TextEdit {
40         id: tel
41         x: 50
42         y: 130
43         font.pointSize: 18
44         text: "Name"
45         focus: true
46     }
47
48     MouseArea {
49         id: ma1
50         anchors.fill: parent
51         onClicked: label1.text =f1(tel.text, "Berlin")
52     }
53
54     MouseArea {
55         id: ma2
56         anchors.fill: rect
57         onClicked: tel.text = f2(label1.text)
58     }
59 }
```

Neu hinzugekommen sind zwei JavaScript-Funktionen *f1()* (Zeile 11 - 14) und *f2()* (Zeile 16 - 19), ein Quadrat (Zeile 21 - 28) und eine zweite *MouseArea* (Zeile 54 - 58). Diese zweite *MouseArea* hat als Elternelement das Quadrat bekommen (Zeile 56). Ein Klick auf das Formular führt also zum Aufruf von Funktion *f1()* und ein Klick auf das Quadrat zum Aufruf von Funktion *f2()*.

Sie können sich sicher vorstellen, dass mehr Inhalt bei diesen JavaScript-Funktionen oder eine größere Anzahl den Quelltext gewaltig erweitern würde. Damit wäre dann der Unterschied zwischen QML und JavaScript möglicherweise schwieriger zu erkennen. Deshalb bietet es sich an, externe JavaScript-Dateien zu verwenden, die den Ressourcen hinzugefügt werden und so jederzeit zur Verfügung stehen.

Rufen Sie im Projektexplorer *Add new...* auf. Wählen Sie dann *Qt* und *JS-Datei* aus.

Nennen Sie diese Datei *funktionen.js*. Fügen Sie sie den Ressourcen hinzu. Rufen Sie diese Datei im Editor auf und verschieben Sie die JavaScript-Funktionen aus der Datei *main.qml* dorthin.

Nun müssen Sie die JavaScript-Funktionen natürlich wieder der QML-Datei bekannt machen. Das geschieht durch einen Import der JavaScript-Datei, die die Funktionen enthält. Zum Zugriff aus der QML-Datei vergeben Sie noch einen Alias (den Sie frei wählen können - Zeile 4). Bei den Funktionsaufrufen setzen Sie jeweils den Alias davor (Zeile 17 und 23).

```

01 import QtQuick 2.15
02 import QtQuick.Window 2.15
03 import QtQuick.Controls 2.5
04 import "funktionen.js" as JS
05
06 Window {
07     width: 400
08     height: 300
09     visible: true
10     title: qsTr("JavaScript")
11
12     ...
13
14     MouseArea {
15         id: ma1
16         anchors.fill: parent
17         onClicked: label1.text = JS.f1(te1.text, "Berlin")
18     }
19
20     MouseArea {
21         id: ma2
22         anchors.fill: rect
23         onClicked: te1.text = JS.f2(label1.text)
24     }
25 }

```

Eigentlich können Sie auch fertige JavaScript-Dateien anstelle Ihrer selbst programmierten hier einbinden. Diese sind zwar ursprünglich für die Benutzung im Webumfeld entwickelt worden, dürften aber auch hier interessant sein, insbesondere bei Bedarf an umfangreichen Funktionalitäten. Solche Frameworks könnten z.B. sein: AngularJS, Backbone.js, Node.js und andere.

■ 8.5 Canvas

Als Quick noch neu war, wurde immer darüber diskutiert, dass ein QML-Typ `Ellipse` benötigt würde. Eine `Ellipse` hätte man nur durch ein Bild realisieren können, oder man hätte ein C++-Programm dazu geschrieben. Wäre man dem Wunsch nach einem eigenen QML-Typ nachgekommen, wären sicher weitere Wünsche nach anderen Formen aufgetaucht. So entschied man sich, in Qt 5 einen neuen QML-Typ – `Canvas` – einzubauen.

Das `Canvas`-Element stellt eine `Bitmap`-Zeichenfläche dar, die für Grafiken, Spiele oder zum Zeichnen mithilfe von JavaScript genutzt werden kann. Das `Canvas`-Element basiert auf dem `HTML5-Canvas`-Element. Die Idee dabei ist, Figuren anhand ihres Umrisspfades zu zeichnen. Die Figuren können dann auch mit Farbe ausgefüllt werden.

Listing 8.6 Eine Datei `main.qml` könnte so aussehen

```
01 import QtQuick 2.15
02 import QtQuick.Window 2.15
03
04 Window {
05     visible: true
06     width: 400
07     height: 300
08     title: qsTr("Canvas")
09
10     Canvas {
11         id: cv
12         x: 0
13         y: 0
14         width: 400
15         height: 300
16         onPaint: {
17             var ctx = getContext("2d");
18             ctx.lineWidth = 4
19             ctx.strokeStyle = "blue"
20             ctx.fillStyle = "red"
21
22             ctx.moveTo(50,220)
23             ctx.lineTo(50,350)
24             ctx.lineTo(150,350)
25             ctx.lineTo(150,250)
26             ctx.closePath()
27
28             ctx.ellipse(50,50,150,100)
29             ctx.fill()
30             ctx.stroke()
31         }
32     }
33 }
```

Dabei haben die einzelnen Zeilen folgende Bedeutung:

- 10 Beginn des QML-Typs `Canvas`. Er reicht bis Zeile 32 und erhält die Größe des umliegenden Fensters.

- 16 Es wird ein Handler beschrieben, der die Funktion *paint()* aufruft und den Inhalt in das Canvas-Element zeichnet.
- 17 *getContext()* ist für die Zeichenumgebung verantwortlich. Als Parameter wird der erforderliche Kontext angegeben. Es wird nur 2d unterstützt. Um damit weiter arbeiten zu können, wird der Rückgabewert in eine JavaScript-Variablen geschrieben, die in Zeile 17 auch deklariert wird.
- 18 Einstellen der Strichstärke
- 19 Einstellen der Farbe des Striches
- 20 die Füllfarbe
- 22 Die Zeichnung beginnt bei einem Punkt mit diesen Koordinaten.
- 23, 24, 25 Es werden jeweils Linien der voreingestellten Farbe und mit der eingestellten Strichstärke zu den Koordinaten dieser Punkte gezogen.
- 26 Der Pfad wird geschlossen, d. h., es wird eine Linie zum Anfangspunkt gezogen. Das Ergebnis ist das Viereck, das Sie mit Zeile 29 in der gewünschten Farbe ausfüllen und mit Zeile 30 anzeigen.
- Sollten Sie die Zeile 30 mit Kommentarzeichen versehen, die Zeile 29 aber gelten lassen, wird ein Viereck gezeichnet, bei dem nur die Füllung zu sehen ist
- 28 Es wird eine Ellipse gezeichnet. Die beiden ersten Parameter nennen die Koordinaten des linken oberen Punktes eines umliegenden Rechtecks.

Und so könnte es aussehen:

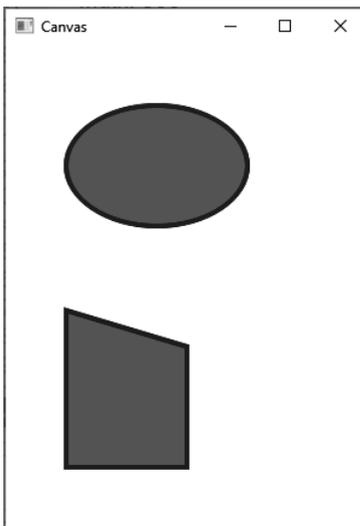


Bild 8.7
Mit Canvas gezeichnet

So wie eben gezeigt, können Sie prinzipiell jede beliebige zweidimensionale Figur zeichnen, Bilder einbinden etc. Weitere Informationen erhalten Sie wieder in der Bibliothek unter

<https://doc.qt.io/qt-6/qml-qtquick-canvas.html>

Einige Beispiele, die Ihnen die Verwendung von Canvas zeigen sollen. Erstellen Sie eine Qt-Quick-Anwendung *CanvasBeispiele*. In der Datei *main.qml* sollen einige Beispiele programmiert werden.

Listing 8.7 Datei *main.qml* des Projekts *CanvasBeispiele*

```
01 import QtQuick 2.15
02 import QtQuick.Window 2.15
03
04 Window {
05     visible: true
06     width: 640
07     height: 480
08     title: qsTr("Canvas")
09
10     Canvas {
11         id: root
12         x: 0
13         y: 0
14         width: 640
15         height: 480
16         onPaint: {
17             var ctx = getContext("2d")
18
19             var gradient = ctx.createLinearGradient(100,0,100,200)
20             gradient.addColorStop(0,"blue")
21             gradient.addColorStop(0.8,"red")
22             ctx.fillStyle = gradient
23             ctx.fillRect(50,50,100,100)
24
25             ctx.shadowColor = "green"
26             ctx.shadowOffsetX = 5
27             ctx.shadowOffsetY = 5
28             ctx.font = "80px Verdana"
29             ctx.fillStyle = "#33a9ff"
30             ctx.fillText("Hanser", 250,120)
31
32             ctx.strokeStyle = "blue"
33             ctx.lineWidth = 4
34             ctx.beginPath()
35             ctx.rect(50,200,100,100)
36             ctx.translate(120,70)
37             ctx.rect(50,200,100,100)
38             ctx.stroke()
39
40             ctx.strokeStyle = "red"
41             ctx.lineWidth = 4
42             ctx.beginPath()
43             ctx.rotate(Math.PI/8)
44             ctx.rect(50,200,100,100)
45             ctx.stroke()
46         }
47     }
48 }
```

Das gestartete Programm sieht so aus:

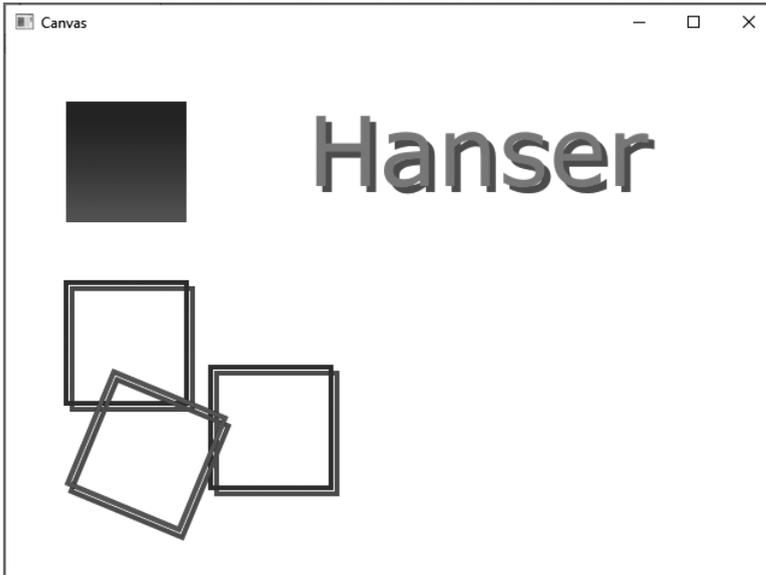


Bild 8.8 Die Ausgabe des Projekts *CanvasBeispiele*

Da alle 2D-Pixel-Operationen des HTML5-Tags Canvas unterstützt werden, bekommen Sie eventuell auch Informationen beim W3C unter:

https://www.w3schools.com/graphics/canvas_reference.asp

Ein weiteres Beispiel könnte so aussehen. Dabei wird ein Canvas-Element mit Button und Label kombiniert. Das gestartete Programm zeigt erst einmal das Formular mit drei Buttons und der roten Schrift. Ein Klick auf einen Button zeigt die Zeichnung in der gewählten Stärke (Bild 8.9).

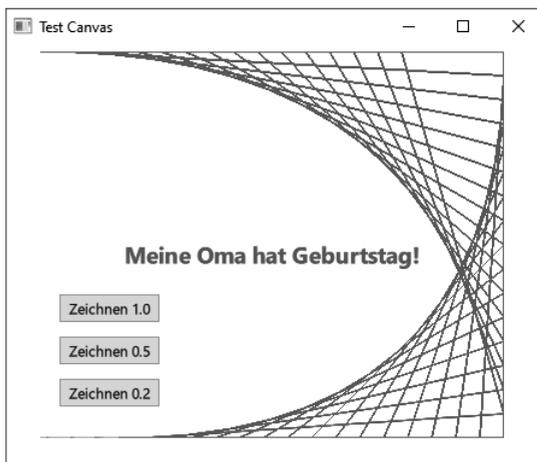
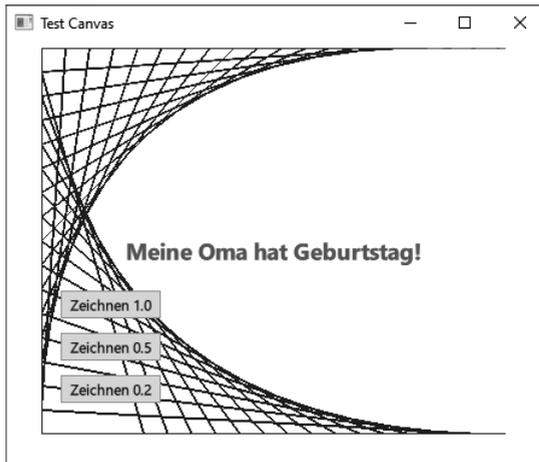


Bild 8.9

Die Ansicht nach Klick auf den Button *Zeichnen 0.5*

Bei Klick auf den oberen Button dreht sich die Zeichnung zusätzlich noch um 180° (Bild 8.10).

**Bild 8.10**

Die Ansicht nach Klick auf den oberen Button

Die Datei *main.qml* des Projekts (Listing 8.8) enthält außer ein wenig Programmierung und Mathematik keine Besonderheiten. Die Linien werden innerhalb einer *while*-Schleife erzeugt (Zeilen 30 – 58). Innerhalb dieser Schleife werden zwei Zeichnungen angefertigt (Zeilen 35 – 45 und 46 – 56). Danach wird die Schleife in Zeile 57 abgebrochen. Sie können jederzeit weitere solcher Zeichnungen programmieren.

Die JavaScript-Variablen *diff* in Zeile 23 bestimmt den Abstand zwischen den Linien. Auch dieser Wert kann natürlich jederzeit geändert werden. Mit Zeile 29 bestimmen Sie die Linienfarbe der Zeichnungen.

Listing 8.8 Die Datei *main.qml* des Projekts *TestCanvas*

```

01 import QtQuick
02 import QtQuick.Controls
03
04 Window {
05     width: 450
06     height: 350
07     visible: true
08     title: qsTr("Test Canvas")
09
10     Canvas {
11         id: can
12         x: 30
13         y: 5
14         width: 385
15         height: 320
16         opacity: 0.0
17
18         onPaint: {
19             id: x
20             var ctx = getContext("2d")
21             var w = can.width
22             var h = can.height
23             var diff = 20
24             var drawing = 1
25             var x1 = 0
26             var y1 = 0

```

```

27         var x2 = w
28         var y2 = 0
29         ctx.strokeStyle = "blue"
30         while (true)
31         {
32             ctx.moveTo(x1, y1);
33             ctx.lineTo(x2, y2);
34             ctx.stroke();
35             if (drawing==1)
36             {
37                 x1 += diff;
38                 y2 += diff;
39                 if (x1 >= w || y2 >= h)
40                 {
41                     x1 = w;
42                     y2 = h;
43                     drawing = 2;
44                 }
45             }
46             else if (drawing==2)
47             {
48                 y1 += diff;
49                 x2 -= diff;
50                 if (y1 >= w || x2 <= 0)
51                 {
52                     y1 = h;
53                     x2 = 0;
54                     drawing = 3;
55                 }
56             }
57             else if (drawing==3) break;
58         }
59     }
60 }
61
62 Button {
63     id: button1
64     x: 45
65     y: 275
66     width: 85
67     height: 25
68     text: "Zeichnen 0.2"
69     onClicked: can.opacity = 0.2
70 }
71
72 Button {
73     id: button2
74     x: 45
75     y: 240
76     width: 85
77     height: 25
78     text: "Zeichnen 0.5"
79     onClicked: can.opacity = 0.5
80 }
81
82 Button {
83     id: button3
84     x: 45
85     y: 205

```

```

86     width: 85
87     height: 25
88     text: "Zeichnen 1.0"
89     onClicked: {
90         can.rotation = 180
91         can.opacity = 1.0
92     }
93 }
94
95 Label {
96     id: label
97     x: 100
98     y: 160
99     width: 250
100    height: 31
101    color: "red"
102    text: "Meine Oma hat Geburtstag!"
103    font.weight: Font.Bold
104    font.pointSize: 14
105 }
106 }

```

Mit *opacity* des Canvas-Objektes (Zeilen 16, 69, 79, 91) bestimmen Sie die Deckkraft der Linienfarbe. Die Eigenschaft *rotation* (Zeile 90) erhält als Wert die Gradzahl der Drehung. Die grundlegende Anordnung des Canvas-Objekts und der Buttons können Sie wieder im Quick Designer vornehmen. Das Canvas-Objekt bekommen Sie angezeigt, wenn Sie links im Navigator-Fenster des Quick Designers *canvas* auswählen (siehe Bild 8.11).

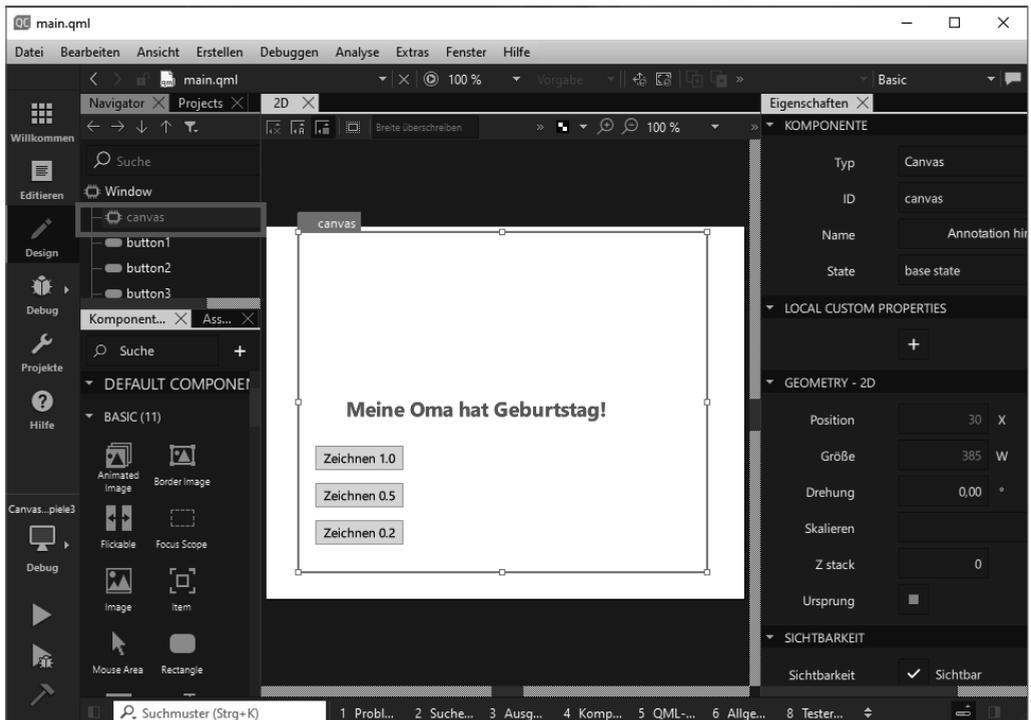


Bild 8.11 Der Quick Designer mit dem Projekt

■ 8.6 QML 3D

Bei den Add-on-Modulen gibt es ein Modul Qt 3D, das für alle Plattformen existiert. Es unterstützt 2D- und 3D-Funktionalitäten. Dazu existieren einige Klassen in diesem Modul. Die QML-Typen dieses Moduls erreichen Sie, wenn Sie in Ihrer *qml*-Datei das entsprechende Import-Statement eintragen (Zeile 3 des Listings 8.9).

Hier ein kleines Beispiel, das Projekt *QML3D*.

Listing 8.9 Die Datei *main.qml* des Projekts *QML3D*

```

01 import QtQuick
02 import QtQuick.Controls
03 import QtQuick3D
04
05 Window {
06     id: root
07     width: 1000
08     height: 800
09     visible: true
10
11     property int pos_z: 300
12     property int pos_x: 300
13
14     View3D {
15         id: view
16         width: 1000
17         height: 800
18         environment: SceneEnvironment {
19             clearColor: "#ffffcc"
20             backgroundMode: SceneEnvironment.Color
21             antialiasingMode: SceneEnvironment.SSAA
22         }
23         PerspectiveCamera {
24             id: cam
25             position: Qt.vector3d(pos_x,150,pos_z)
26             eulerRotation.x: -20
27         }
28         DirectionalLight {
29             eulerRotation.x: -40
30             eulerRotation.y: -70
31         }
32         Model {
33             position: Qt.vector3d(50,150,0)
34             source: "#Cube"
35             materials: [ DefaultMaterial {
36                 diffuseColor: "blue"
37             }
38         ]
39         SequentialAnimation on y {
40             loops: Animation.Infinite
41             NumberAnimation {
42                 duration: 3000
43                 to: -130
44                 from: 150
45             }

```

```

46         NumberAnimation {
47             duration: 3000
48             to: 150
49             from: -130
50         }
51     }
52 }
53 }
54 }
55 Label {
56     x: root.width -120
57     y: 20
58     font.pointSize: 12
59     text: "Kameraposition"
60 }
61 Label {
62     x: root.width -80
63     y: 45
64     font.pointSize: 12
65     text: "z"
66 }
67 Label {
68     x: root.width -80
69     y: 245
70     font.pointSize: 12
71     text: "x"
72 }
73 Slider {
74     id: s11
75     x: root.width -50
76     y: 50
77     from: 100
78     to: 450
79     height: 150
80     orientation: Qt.Vertical
81     onMoved: pos_z = valueAt(position)
82 }
83 Slider {
84     id: s12
85     x: root.width -50
86     y: 250
87     from: -50
88     to: 180
89     height: 150
90     orientation: Qt.Vertical
91     onMoved: pos_x = valueAt(position)
92 }
93 }

```

Der QML-Typ `View3D` erzeugt eine 2D-Zeichenfläche, auf der eine 3D-Szene angezeigt werden kann (Zeile 14 in Listing 8.9). Mit `SceneEnvironment` (Zeile 18 – 22) wird das Aussehen der Zeichenfläche bestimmt. Die Eigenschaft `source` des Typs `Model` (Zeile 34) nennt das darzustellende Objekt. Dieser Eigenschaft kann die URL einer Datei übergeben werden, oder es werden einige der eingebauten Objekte verwendet. Das sind:

#Rectangle, #Sphere, #Cube, #Cone und #Cylinder.

Sie können auch für dieses Beispiel wieder den Quick Designer benutzen und einzelne Eigenschaften der verwendeten QML-Typen damit einstellen. Markieren Sie links im Navigator *Model*, dann wird Ihnen bereits bei darüberstehendem Mauszeiger das verwendete Material angezeigt. Es gehört zum Modul `QtQuick3D`. Auch dafür können Sie im rechten Fenster des Quick Designers die Eigenschaften einstellen.

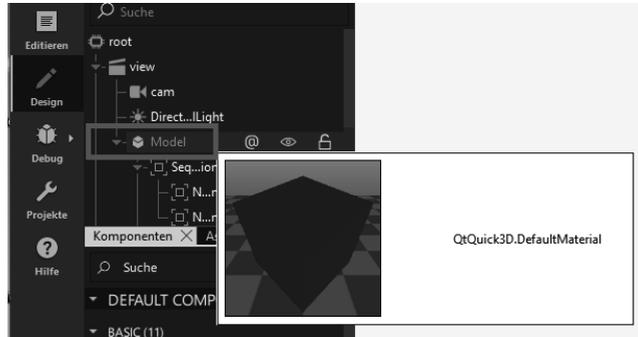


Bild 8.12 Die Materialangabe im Quick Designer

Letztlich muss bestimmt werden, wie ein 3D-Objekt auf einer 2D-Zeichenfläche dargestellt werden soll (projiziert werden soll). Das übernimmt der QML-Typ `Camera`, in unserem Fall die Ableitung davon, `PerspectiveCamera` (Zeile 23). Man könnte sich vorstellen, es wird bestimmt, wie Sie auf das gewählte Objekt sehen (von oben, von unten, von der Seite etc.). Mit `DirectionalLight` (Zeile 28) wird der Standort der Lichtquelle bestimmt, also die Aufhellungen und Schattierungen des dargestellten Objekts.

Die Zeilen 39 – 51 bestimmen die Animationen, die in unserem Falle gezeigt werden sollen. Vermutlich werden Sie 3D-Darstellungen hauptsächlich für Konstruktionen benutzen, um sich ein bestimmtes Objekt von verschiedenen Seiten ansehen zu können, dann spielen Animationen sicher keine Rolle.

Beim gestarteten Programm wird ein Würfel angezeigt, der sich ständig auf und ab bewegt. Die Kameraposition und die Seitenverschiebung lassen sich mit Schiebereglern ändern.



Bild 8.13
Das Programm *Quick3D*

■ 8.7 QML Charts

Wie bereits in Kapitel 4 beschrieben, gibt es unter den Qt Add-on-Modulen ein Modul Qt Charts. Es darf unter kommerzieller Lizenz oder im Rahmen der GNU General Public License v3 verwendet werden. Das kann als QWidget, QGraphicsWidget oder als QML Type sein. Zum QML Type erhalten Sie weitere Informationen auch unter:

<https://doc.qt.io/qt-6/qtcharts-qmlmodule.html>

Erstellen Sie zum Testen eine einfache Qt Quick-Anwendung mit *CMake*. Wir nennen sie *LineChart_QML*. Die QML-Datei erhält eine Zeile `import QtCharts`. Die Datei *CMakeLists.txt* wird ergänzt durch diese beiden Zeilen:

```
find_package(Qt6 REQUIRED COMPONENTS Charts)
target_link_libraries(LineChart_QML PRIVATE Qt6::Charts)
```

Der QML-Typ `ChartView` (Zeilen 11–36) enthält zwei QML-Typen `ValueAxis`, mit denen über `LineSeries` (Zeilen 31–35) die x- und die y-Achse bestimmt werden.

In Zeile 40 werden zur Anschauung in einer Schleife mithilfe der JavaScript-Funktion `Math.random()` die Koordinaten einzelner Punkte erzeugt und die Linie dargestellt.

Listing 8.10 Datei *main.qml* des Projekts *LineChart_QML*

```
01 import QtQuick
02 import QtQml
03 import QtCharts
04
05 Window {
06     width: 900
07     height: 700
08     visible: true
09     title: qsTr("LineChart")
10
11     ChartView {
12         anchors.fill: parent
13         title: qsTr("LineChart")
14         legend.visible: false
15         antialiasing: true
16
17         ValueAxis {
18             id: axisX
19             min: 0
20             max: 20
21             tickCount: 10
22         }
23
24         ValueAxis {
25             id: axisY
26             min: 0
27             max: 90
28             tickCount: 10
29         }
30
31         LineSeries {
```

```

32         id: series1
33         axisX: axisX
34         axisY: axisY
35     }
36 }
37
38 Component.onCompleted: {
39     for (var i = 0; i <= 20; i++) {
40         series1.append(i, Math.random()*80);
41     }
42 }
43 }

```

Nach dem Start des Programms könnten Sie folgendes Bild erhalten.

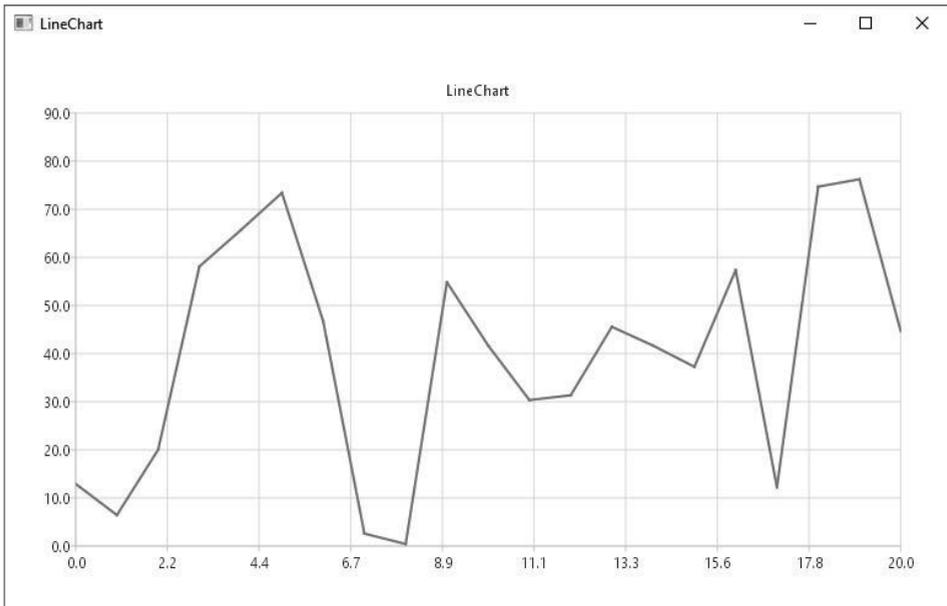


Bild 8.14 Beispiel für ein *LineChart*

Ersetzen Sie die Zeilen 31 – 35 durch folgenden Code.

```

StackedBarSeries {
    id: mySeries
    axisX: BarCategoryAxis { categories: ["2017", "2018", "2019", "2020", "2021", "2"] }
    BarSet {values: [2, 2, 3, 4, 5, 6]}
    BarSet {values: [5, 1, 2, 4, 1, 7]}
    BarSet {values: [3, 5, 8, 13, 5, 8]}
}

```

Damit ergibt sich Bild 8.15. Eine Kombination mit *LineSeries* könnte so aussehen (Bild 8.16).

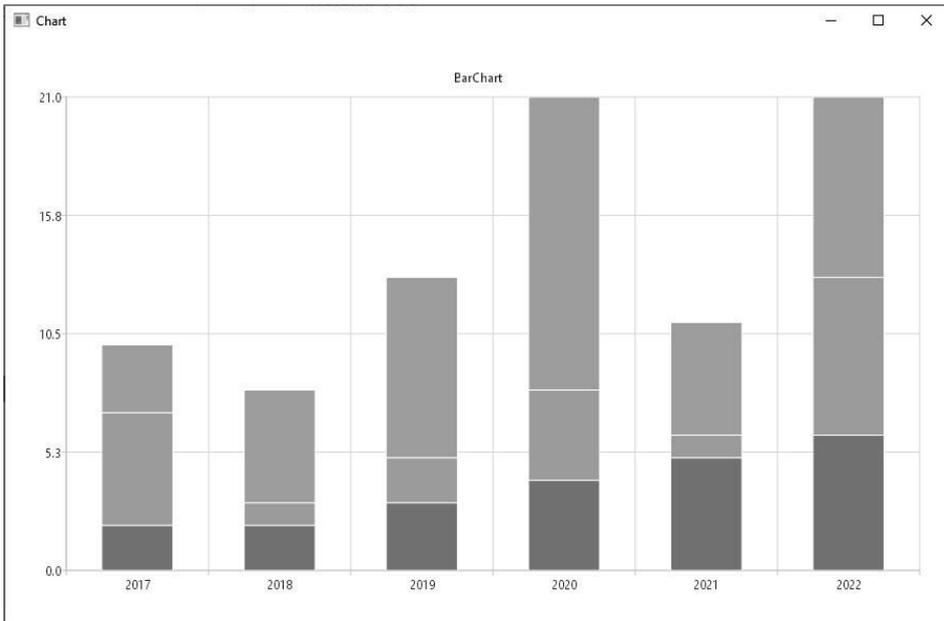


Bild 8.15 Ein *BarChart*

Eine Kombination beider Varianten sieht so aus (Bild 8.16).

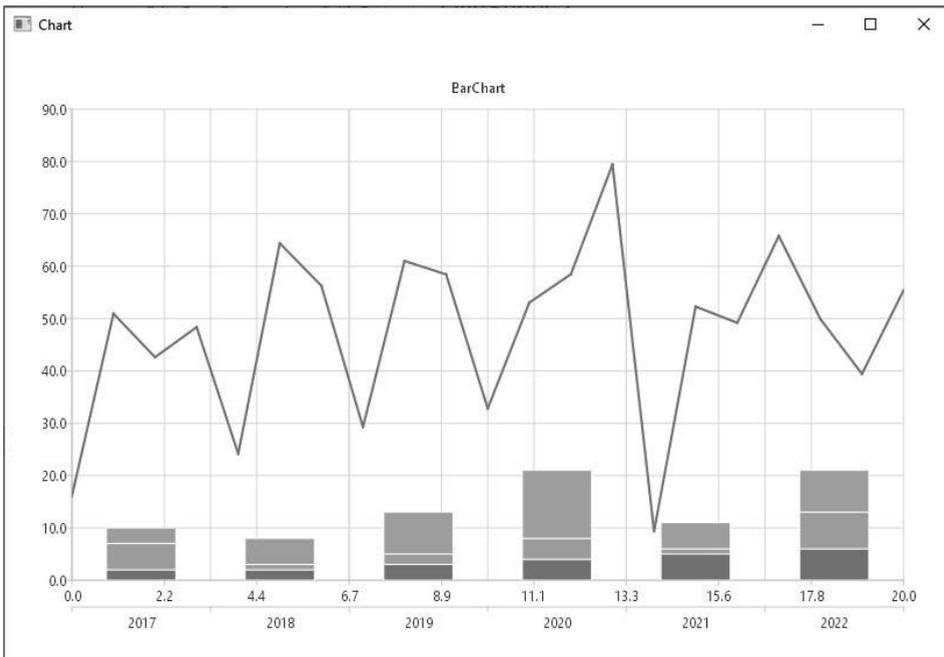


Bild 8.16 Kombination von *BarChart* und *LineChart*

Weitere Möglichkeiten wären das horizontale BarChart (Bild 8.17) oder ein PieChart nach Bild 8.18.

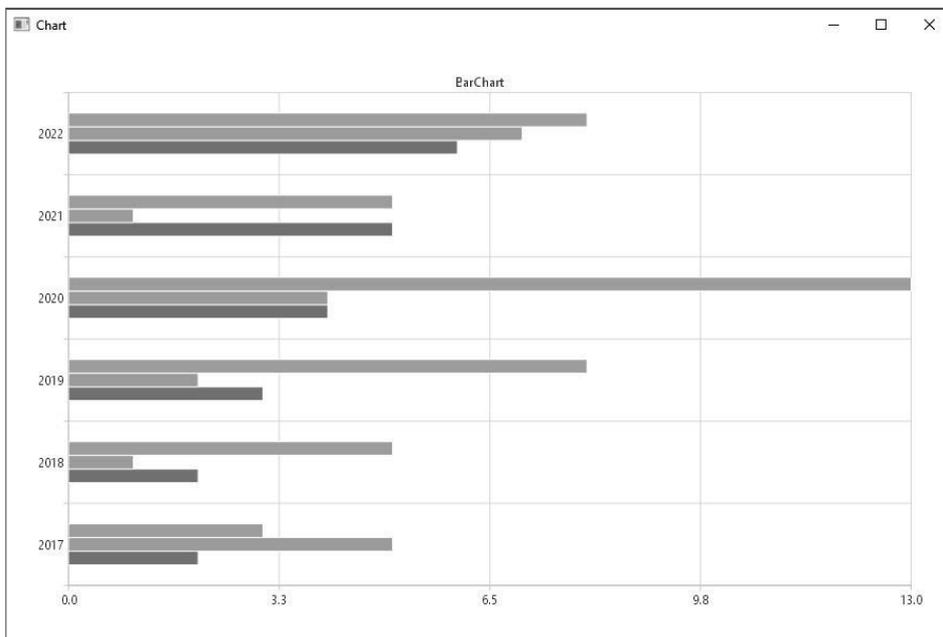


Bild 8.17 Horizontale BarCharts

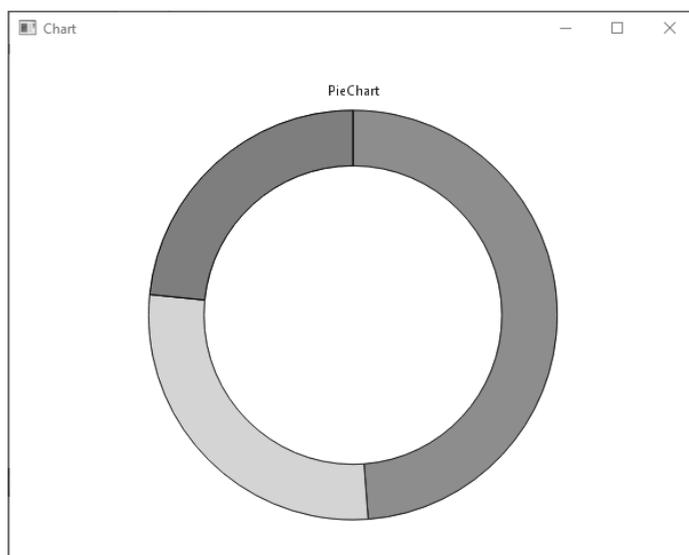


Bild 8.18 Ein PieChart

Die neuen Quellcodes dazu sehen folgendermaßen aus (zuerst für horizontale BarCharts)

```
HorizontalBarSeries {
  axisY: BarCategoryAxis { categories: ["2017", "2018", "2019", "2020", "2021"] }
  BarSet {values: [2, 2, 3, 4, 5, 6]}
  BarSet {values: [5, 1, 2, 4, 1, 7]}
  BarSet {values: [3, 5, 8, 13, 5, 8]}
}
```

oder für PieCharts.

```
PieSeries {
  id: pieOuter
  size: 0.96
  holeSize: 0.7
  PieSlice {value: 19511; color: "#8AB846"; borderColor: "#163430"}
  PieSlice {value: 11105; color: "#C0EEFF"; borderColor: "#3B391C"}
  PieSlice {value: 9352; color: "#DF8939"; borderColor: "#13060C"}
}
```

■ 8.8 Mediaplayer

Dieses Programm ist zum Abspielen von mp4-Dateien (und anderer Medienformate) geeignet (Bild und Ton). Dazu benötigen Sie auf Ihrem Rechner die entsprechenden Codecs.

Falls noch nicht vorhanden, installieren Sie am besten das *K-Lite Codec Pack 16.0.5 Basic*. Sie können es von verschiedenen Quellen downloaden. Die Quelltexte zum Programm sind wieder über die Webseite zum Buch zu erhalten. Wenn Sie dann den Mediaplayer starten, erhalten Sie erst einmal ein Fenster wie in Bild 8.19. Sie können nun den Pfad zu einer Bilddatei in das Textfeld eingeben oder Sie benutzen den Button rechts daneben mit dem Haken. Dann öffnet sich der Windows Explorer, und Sie können die Datei auswählen (Bild 8.20).

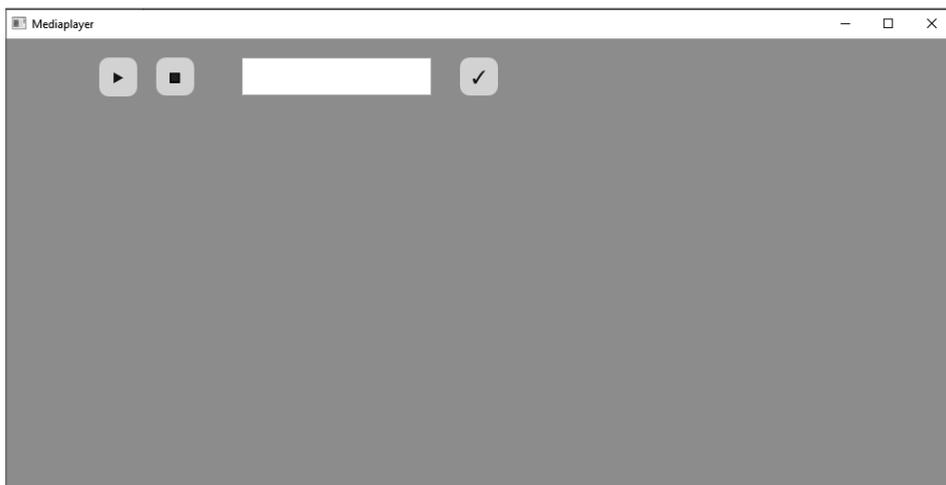


Bild 8.19 Das gestartete Programm *Mediaplayer*

Index

Symbole

2D-Zeichenfläche 159
3D-Darstellung 160
3D-Objekt 160
3D-Szene 159

A

AbstractButton 140
All QML Types 140
Android 191
Android-Compiler 195
Android-Emulator 194
Android NDK 192
Android SDK 192
Animationen 160, 171
– mit Qt 54
Animation Framework 55
Animationstypen von QML 169
Animator 169
ApplicationWindow 167
Assoziative Qt-Container 61
autogen/include 13

B

Bild zeichnen 73
Breakpoint 222
Bugs 220
Build-Ordner 12
Build-Tools 201
Button 143, 182

C

C++- oder Qt-Klasse 180
C++-Quelltext 13
Camera 160
Canvas 151, 153f.
CentOS 205
ChartView 161
Chrome 197
clicked() 140
CMake 2, 10
CMakeLists.txt 2, 10, 187, 207
CMakeLists.txt.user 10
Compiler 26
– für Qt 1
connect()-Funktion 16, 32, 34, 138, 181
Connections 185
Connection String 119
Containerklassen 57, 61
Control Area Network (CAN-Bus) 211

D

Datei 125
– .pri 12
– .ui 12
Datenbank 108
Datenbankklassen 111
Datenbankserver 108
Datenbanktreiber 117
Datentypen 56
Debugger 4
Debugging 217
DEFAULT COMPONENTS des Quick Designer
143
Delegates 57, 59

Deployment 217
 DirectionalLight 160
 Dokumentation 217, 224f.
 Dokumentationskommentar 225
 Domainnamen 118
 Doxygen 225
 Drucken per QML 182
 Druckvorgang 125
 DTLS (Datagram Transport Layer Security) 211
 Dynamisches Linken 227

E

easing.type 169
 Edge 197
 E-Mail-Client 210
 emit 16
 Emscripten 197
 Entwicklermodus 195
 Enums in QML 146
 Ereignistypen 60
 Event Handling 60
 Eventklassen in Qt 60
 event loop 138, 220
 Exceptions 220
 exec() 13, 138

F

Farbverläufe 86
 FIFO 61
 Firefox 197
 footer 167
 FORMS 12
 Formular 8

G

getContext() 152
 GUI-Erstellung 22

H

Handbuch, Qt Creator 4
 Handler 141
 HEADERS 12
 high-level API 64
 HTML 98, 209
 HTML5-Tags 154
 HTTP 207, 209

I

Installation 4
 Inter-Process Communication (IPC) 210
 isChecked() 130
 Item 167, 174f.
 Iterator 61

J

Java JDK 192
 JavaScript 137, 147
 – externe Dateien 150
 – Funktionen 147
 – Import einer Datei 150

K

keyPressEvent() 50
 Kommentarzeichen 14
 Konfigurationsdatei für QDoc 225
 Konsole 9

L

Label 148, 167, 182
 Layouts 24, 35
 LIFO 61
 LineSeries 161
 Linux 191
 Low-Level-Funktionen 64
 lupdate 167
 lupdate.exe 101

M

main.cpp 13
 main()-Funktion 13
 main.qml 139
 mainwindow.ui 12
 MariaDb 113
 Maus-/Tastatur-Events 49
 Medienformate 165
 Meldungen 52
 Menü 37
 menuBar 167
 MenuBar 167
 MenuItem 167
 Meta-Object Compiler 2, 11
 Microsoft Access 119

Microsoft SQL Server 118
 moc 2
 Model 159
 Model-View-Framework 25
 Module 5
 MouseArea 147
 Mouse Events 79
 mousePressEvent() 50
 mp4-Dateien 165
 MVC-Prinzip 57
 MySQL 113

N

Netzwerkprogrammierung 206
 NumberAnimation 171, 174

O

Objekt-Explorer 15
 Objekttypen 146
 onClicked 141
 onentry 178
 Online Installer 204
 open() 134
 OpenGL 217, 229
 Open Graphics Library 229
 OpenSSL 192
 openSUSE 200

P

paint() 152
 paintEvent() 71, 73
 ParallelAnimation 171, 173f.
 parent 147
 PDF-Datei erzeugen 125
 PerspectiveCamera 160
 phpMyAdmin 113
 Portnummer 207
 pressed(). 140
 Programmfehler 220
 Projektextplorer 141
 Projektmappen-Explorer 20
 PropertyAnimation 171
 PropertyChanges 176
 Property-System 68
 Python 197

Q

QApplication 13, 181
 QBluetoothDeviceDiscoveryAgent 213
 QBluetoothLocalDevice 213
 QBluetoothLocalDevice::Hostmode 213
 QBluetoothServer 214
 QBluetoothSocket 214
 QCanBusDevice 211
 QCanBusDeviceInfo 211
 QCoreApplication 230
 QDBus 211
 QDebug 52
 QDialog 22
 QDoc 225
 QDtls 211
 QException 220
 QFile 132, 134
 QFileDevice 132
 QFrame 78
 QGradient 86
 QGraphicsScene 80, 88
 QGraphicsView 88
 QGraphicsWidget 161
 QGuiApplication 138, 230
 QImage 73
 QIODevice::Append 133
 QIODevice::ReadOnly 133
 QIODevice::WriteOnly 133
 QLabel 129
 QMainWindow 13, 22
 qmake 2, 11
 QMenuBar 13
 QMessageBox 53
 QML 22, 180
 QML-Basistypen 148, 180
 QML-Debuggerkonsole 224
 QmlDesigner 141
 QML Engine 137
 QML (Qt Meta Language) 6
 QML (Qt Modeling Language) 137
 qmlRegisterType() 186
 QML-Typen 137
 – Button 140
 – Window 139
 Q_OBJECT 5
 QObject 5, 181
 QOpenGLFunctions 229 ff.
 QOpenGLWindow 229 ff.
 QPaintDevice 71
 QPaintEngine 71

QPainter 127, 183, 230
 QPrintDialog 125, 183
 QPrinter 125, 127, 183
 Q_PROPERTY 6
 Q_PROPERTY() 185f.
 Q_PROPERTY()-Makros 188
 QPropertyAnimation 54
 QPushButton 29
 QQmlApplicationEngine 139, 184
 QQmlEngine 184, 186
 QSqlDatabase 5, 112, 117, 187
 QSqlQuery 112, 187
 QStatusBar 13, 53, 129
 qsTr() 139, 167
 QSurfaceFormat 230
 QT 11
 Qt 3D 158
 QtTabWidget 39
 Qt Add-on-Module 161
 Qt Add-Ons 4
 Qt Assistant 7
 Qt-Bibliothek 2
 Qt Bluetooth 212, 215
 Qt CAN Bus 211
 Qt Charts 161
 Qt::ConnectionType 16
 Qt-Containerklassen 61
 Qt Core 54
 QtCore 217
 QTcpServer 208
 QTcpSocket 206
 Qt Creator 7
 – Beispiele 4
 – Handbuch 4
 Qt-Datentypen 57
 Qt D-Bus 211
 Qt Designer 8, 14f.
 Qt Essentials 4
 QTextBrowser 73
 QTextEdit 209
 QTextStream 134
 QtGui 217
 Qt Linguist 9, 101f., 167
 Qt Maintenance Tool 5, 191
 Qt Marktplatz 4
 Qt Meta Language 22
 Qt Modeling Language 22
 QtMultimedia 144
 Qt Network 207f., 211
 Qt Online-Community 4

Qt Online Installer 2
 Qt OpenGL 229
 Qt-Plug-ins 2
 QtPrintSupport 7, 125, 183
 Qt Property-System 185
 QtQML 137, 147
 Qt Quick 137
 QtQuick 137
 Qt-Quick-Application 138
 Qt Quick Controls 137, 232
 QtQuick.Controls 143
 qtquickcontrols2.conf 234
 QtQuick.Window 139
 Qt Ressourcen-Datei 90
 Qt SerialBus 211
 QtSQL 7, 187
 Qt StyleSheets 93
 QtWidgets 217
 QueuedConnection 139
 Quick Designer 141
 Quick-Oberflächen 180
 QUnhandledException 220
 QVariant 68
 QWidget 22
 QXmlReader 5

R

Rectangle 177
 Regulärer Ausdruck 24
 Remote Procedure Calling (RPC) 210
 RESOURCES 12
 Ressource Compiler 11
 – (rcc) 98
 Ressourcen 73, 91
 Ressourcensystem 89
 Rich Text 98
 – Dokumente 99
 rootObjects() 184
 rotate() 85
 RotationAnimation 175

S

Safari 197
 scale() 85
 SceneEnvironment 159
 Scribe 99
 send 178
 Sequenzielle Qt-Container 61

setContextProperty() 184
 set(PROJECT_SOURCES) 187
 setupUi() 28
 shear() 85
 Signale und Slots 15
 Signal-Funktion 15, 183
 Signal-Handler 181
 signals 5, 15
 Signal-Slot-Prinzip 15
 Signal- und Slot-Funktionen
 – verbinden 30
 Slot-Funktion 15, 183
 slots 5, 15
 SOURCES 12
 Sprachumstellung 105
 SQL (Structured Query Language) 108
 State 175
 states 175
 Statusmeldung 54
 Statuszeile 53
 string 181
 Style Imagine 233
 Style Material 233
 StyleSheets 92, 94
 – externe Datei 97
 Stylesheet-Editor 95
 Styling von QtQuick Controls 232
 Subdirs-Projekt 218

T

Tastenkombinationen für Qt Creator 51
 TCP/IP-Protokoll 206
 TextEdit 148, 182, 209, 214
 TextField 190
 Threads 64, 66
 tr() 103
 Transformation 84
 translate() 85
 triggered() 38, 167

U

Übersetzen von Texten im Programm 101
 Übersetzungsdatei 101
 Ubuntu 200
 UDP 211
 ui_[Klassenname].h 12
 ui_mainwindow.h 13
 UML 177, 220
 – Zustandsdiagramm 55, 177
 Uninstall Qt 5
 User Interface Compiler 11, 13

V

ValueAxis 161
 View3D 159
 Visual Studio 1 f., 7, 16 ff.

W

Wasm 196
 WebAssembly 191, 196
 Werttypen 146
 Widgets 23
 windeployqt 228
 Window 139, 143, 181, 194
 Windows Deployment Tool 228
 Wrapperklassen 229

X

XML-Datei 8

Z

Zeichnen
 – auf Widgets 71
 – freihändig 80
 Zustandseditor 55, 177
 – von Qt 176
 Zweites Fenster 44