Qt Quick Painting using Canvas Item

Release 1.0

Digia, Qt Learning

February 28, 2013

Contents

1 About this Guide **1**
- 1.1 Why Would You Want to Read this Guide? 1
- 1.2 Get the Source Code and the Guide in Different Formats 1
- 1.3 License 2

2 Introduction **3**
- 2.1 A Basic Example 3
- 2.2 Essential Context2D Properties / Methods 5

3 A Pie Chart **7**
- 3.1 The Layout & the Conceptual Context of the Pie Chart 7
- 3.2 How to Draw a Sector 10
- 3.3 Drawing the Chart 12
- 3.4 Finalizing the Chart 13

4 Porting HTML5 Canvas Code to Qt Quick **18**
- 4.1 The HTML5 Canvas Code 18
- 4.2 The Qt Quick Canvas Code 20

5 Conclusion **23**

CHAPTER 1

About this Guide

1.1 Why Would You Want to Read this Guide?

The goal of this guide is to inform you about the best programming practices using the Canvas type in Qt Quick 2.0. A prerequisite to this guide is to have a solid understanding of the QML language, so I recommend reading the Qt Quick Application Development Primer first to understand how to use Qt Quick for application development. Throughout this guide, we'll walk you through various aspects and examples of Qt Quick 2.0 Painting API with Canvas. References to other information sources are provided to make it easy for you to deepen your understanding of the used API.

1.2 Get the Source Code and the Guide in Different Formats

A .zip file that contains the example source code referred in each chapter:

Source code[1]

The guide is available in the following offline formats:

- PDF[2]
- ePub[3] for ebook readers.
- Qt Help[4] for Qt Assistant and Qt Creator.

[1] http://releases.qt-project.org/learning/developerguides/canvastutorial/canvasexample_src.zip
[2] http://releases.qt-project.org/learning/developerguides/canvastutorial/QtQuickCanvasTutorial.pdf
[3] http://releases.qt-project.org/learning/developerguides/canvastutorial/QtQuickCanvasTutorial.epub
[4] http://releases.qt-project.org/learning/developerguides/canvastutorial/QtQuickCanvasTutorial.qch

1.3 License

Copyright (C) 2012 Digia Plc and/or its subsidiary(-ies). All rights reserved.

This work, unless otherwise expressly stated, is licensed under a Creative Commons Attribution-ShareAlike 2.5.

The full license document is available from http://creativecommons.org/licenses/by-sa/2.5/legalcode .

Qt and the Qt logo is a registered trade mark of Digia plc and/or its subsidiaries and is used pursuant to a license from Digia plc and/or its subsidiaries. All other trademarks are property of their respective owners.

What's Next?

Next we will go through a brief instruction of the Painting APIs in Qt Quick 2.0 and a basic example.

CHAPTER 2

Introduction

If you want to draw custom graphics within your Qt Quick application, the Qt Quick Canvas[1] item is your choice.

The Canvas type was introduced in Qt Quick 2.0 and provides an area in which you can draw using JavaScript. It uses a high-level command-set based on the HTML5 Canvas specification[2]. The Canvas type allows you to draw basic and complex shapes, add color, gradients, shadows, images, text, and access low-level pixel data. Using JavaScript, facilitates the presentation of dynamic content.

After a brief introduction to the Canvas type, we'll develop an interactive pie chart visualization. Later in this guide, we'll see how to port existing HTML5 Canvas code to a Qt Quick 2 application.

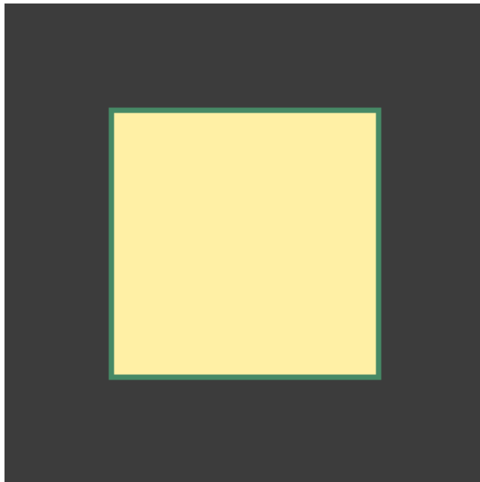
This tutorial introduces you to the Qt Quick Canvas using example programs and is not meant to show each and every aspect of this type. A detailed description of the Canvas type and its supported rendering commands can be found in the Qt documentation pages (Canvas[3], Context2D[4]). Also note that a large number of good HTML5 Canvas API tutorials are available on the internet. As the Qt Quick Canvas type is based on the HTML5 specification, these tutorials can serve as an excellent starting point to learn drawing. We have listed a few of those tutorials at *the end of this tutorial <conclusion.html>*. We also assume that you are already familiar with Qt Quick in general, as this tutorial does refer to some the non-Canvas features.

2.1 A Basic Example

The Qt Quick Canvas type provides a place in your application to draw upon. The actual drawing as well as the resource handling is done by its associated **:qt5-snapshot:'Context2D <qtquick/qml-qtquick2-context2d.html>'_** type, which provides the drawing API and manages the transformation stack and style state. It also lets you customize some of its internals such as multithreading, tiling, and the usage of hardware acceleration.

[1] http://doc-snapshot.qt-project.org/qtquick/qml-qtquick2-canvas.html
[2] http://www.w3.org/TR/html5/the-canvas-element.html
[3] http://doc-snapshot.qt-project.org/qtquick/qml-qtquick2-canvas.html
[4] http://doc-snapshot.qt-project.org/qtquick/qml-qtquick2-context2d.html

Let's start with a simple example: a small piece of code that displays a colored rectangle:



This is the source code:

```
/****************************************************************************
**
** Copyright (C) 2012 Digia Plc and/or its subsidiary(-ies).
** Contact: http://www.qt-project.org/legal
**
** $QT_BEGIN_LICENSE:BSD$
** You may use this file under the terms of the BSD license as follows:
**
** "Redistribution and use in source and binary forms, with or without
** modification, are permitted provided that the following conditions are
** met:
**   * Redistributions of source code must retain the above copyright
**     notice, this list of conditions and the following disclaimer.
**   * Redistributions in binary form must reproduce the above copyright
**     notice, this list of conditions and the following disclaimer in
**     the documentation and/or other materials provided with the
**     distribution.
**   * Neither the name of Digia Plc and its Subsidiary(-ies) nor the names
**     of its contributors may be used to endorse or promote products derived
**     from this software without specific prior written permission.
**
**
** THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
** "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
** LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
** A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
** OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
** SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
** LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
** DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
** THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
** (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
** OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE."
**
** $QT_END_LICENSE$
**
****************************************************************************/
import QtQuick 2.0
```

```
Rectangle {
    id: root
    width: 360; height: 360
    color: "#3C3C3C"

    Canvas {
        id: canvas
        width: 300; height: 300
        anchors.centerIn: parent

        onPaint: {
            // get the drawing context
            var ctx = canvas.getContext('2d')

            // create a rectangle path
            ctx.rect(50, 50, 200, 200)

            // setup fill color
            ctx.fillStyle = "#FFF0A5"

            // fill path
            ctx.fill()

            // setup line width and stroke color
            ctx.lineWidth = 4
            ctx.strokeStyle = "#468966"

            // stroke path
            ctx.stroke()
        }
    }
}
```

The usual way is to declare a `Canvas` type and place the drawing commands inside its `onPaint` handler. After acquiring the drawing context, we prepare a rectangular path using `rect(real x, real y, real w, real h)`. Then we set up the fill color state to yellow using `fillStyle` and fill the rectangle by calling `fill()`. The green border of the rectangle is drawn by setting `strokeStyle` and calling `stroke()` respectively. The `lineWidth` property sets the width of the stroke.

Note: The order of `stroke()` and `fill()` matters: The stroke pattern is drawn centered along the path (in this example with a 2 px width pattern to the left side and a 2 px width pattern to the right side). If we call `stroke()` before `fill()`, `fill()` would over paint the inner part of the border resulting in a 2 px wide border.

2.2 Essential Context2D Properties / Methods

Here is an overview of the most frequently used drawing commands:

Group	Operation	Note
Path	beginPath()	Begin new path
Path	moveTo(x, y)	Move to position
Path	lineTo(x, y)	Add line path
Path	rect(x, y, width, height)	Add rect path
Path	ellipse(x, y, width, height)	Add ellipse path
Path	**arc(x, y, radius, startAngle,** endAngle, anticlockwise)	Add arc path
Path	arcTo(x1, y1, x2, y2, radius)	Add arc path
Path	text(text, x, y)	Add text path
Transformation	translate(x, y)	Move coordinate system
Transformation	rotate(angle)	Rotate coordinate system
Transformation	scale(x, y)	Scale coordinate system
Transformation	shear(sh, sv)	Shear coordinate system
Style	strokeStyle	Set up line style
Style	lineWidth	Set up line width
Style	fillStyle	Set up fill style
Drawing	stroke()	Draw path using style
Drawing	fill()	Fill path using style

What's Next?

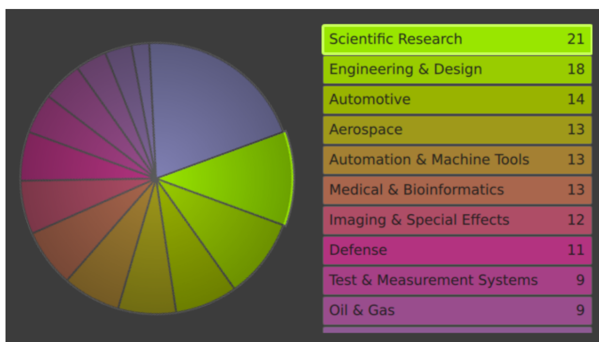
In the next chapter we will go through some more advanced usage of the API by drawing a Pie Chart.

CHAPTER 3

A Pie Chart

In this chapter, we'll present a more elaborate example: we'll create a pie chart item that visualizes the data of a Qt Quick ListModel similar to the Qt Widget example[1]. The ListModel's values are shown in a ListView next to the chart. If one of its entries is selected, the according sector of the pie chart gets highlighted.

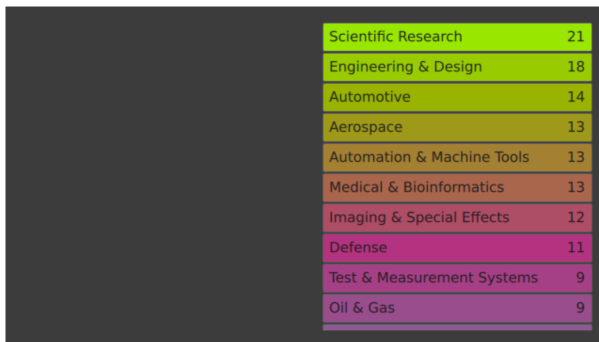
The final application looks like this:



3.1 The Layout & the Conceptual Context of the Pie Chart

Let's start with the layout of the application and the context in which the pie chart lives. This means setting up the ListModel, placing the Canvas and creating the ListView.

[1] http://qt-project.org/doc/qt-4.8/itemviews-chart.html



The data model holds roles for the item's labels, values and color values. We populate this model with an example dataset:

```
import QtQuick 2.0

Rectangle {
    id: root
    width: 640
    height: 360
    color: "#3C3C3C"

    ListModel {
        id: model
        ListElement { label:"Scientific Research";
                      value:21; color:"#99e600" }
        ListElement { label:"Engineering & Design";
                      value:18; color:"#99cc00" }
        ListElement { label:"Automotive";
                      value:14; color:"#99b300" }
        ListElement { label:"Aerospace";
                      value:13; color:"#9f991a" }
        ListElement { label:"Automation & Machine Tools";
                      value:13; color:"#a48033" }
        ListElement { label:"Medical & Bioinformatics";
                      value:13; color:"#a9664d" }
        ListElement { label:"Imaging & Special Effects";
                      value:12; color:"#ae4d66" }
        ListElement { label:"Defense";
                      value:11; color:"#b33380" }
        ListElement { label:"Test & Measurement Systems";
                      value:9;  color:"#a64086" }
        ListElement { label:"Oil & Gas";
                      value:9;  color:"#994d8d" }
        ListElement { label:"Entertainment & Broadcasting";
                      value:7;  color:"#8d5a93" }
        ListElement { label:"Financial";
                      value:6;  color:"#806699" }
        ListElement { label:"Consumer Electronics";
                      value:4;  color:"#8073a6" }
        ListElement { label:"Other";
                      value:38; color:"#8080b3" }
    }

...
```

The canvas that shows the pie chart is placed in the left half of the application. We'll be

implementing the drawing later in the following sections:

```
...
    Canvas {
        id: canvas
        anchors.top: parent.top
        anchors.bottom: parent.bottom
        anchors.left: parent.left
        anchors.right: parent.horizontalCenter
    }
...
```

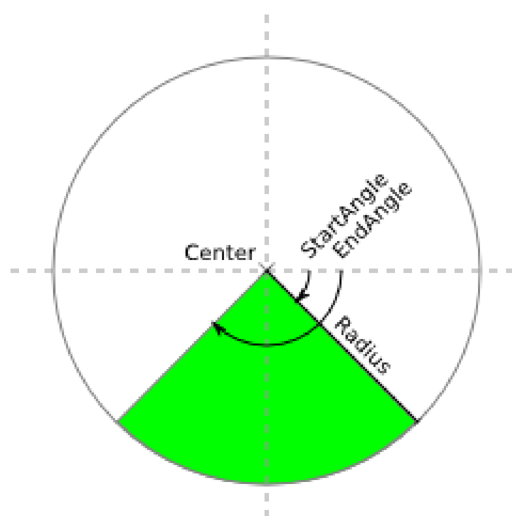
On the right side, we display the model's data in a list view, which shows the item's labels and values in colored rows. It represents the legend of the chart so to say:

```
...
    ListView {
        id: view
        anchors.top: parent.top
        anchors.bottom: parent.bottom
        anchors.left: parent.horizontalCenter
        anchors.right: parent.right
        anchors.margins: 16
        clip: true
        focus: true
        model: model
        delegate: Item {
            width: view.width
            height: 32
            Rectangle {
                anchors.fill: parent
                anchors.margins: 1
                radius: 2
                color: model.color
                border.color: Qt.lighter(root.color)
            }
            Text {
                anchors.verticalCenter: parent.verticalCenter
                anchors.left: parent.left
                anchors.margins: 8
                text: model.label
                color: "#1C1C1C"
            }
            Text {
                anchors.verticalCenter: parent.verticalCenter
                anchors.right: parent.right
                anchors.margins: 8
                text: model.value
                color: "#1C1C1C"
            }
        }
    }
}
```

3.2 How to Draw a Sector

A pie chart is a circular chart consisting of several sectors - one for each data set. The area of each sector is proportional to the value it represents. So let's first take a look at how to draw one of the pie chart sectors.

The size of a sector is defined by the length of its arc or the arc's subtending angle. If you think of the whole pie having an angle of `2 PI`, each sector should cover an angle of `value * (2 * PI) / SUM_OF_VALUES`.

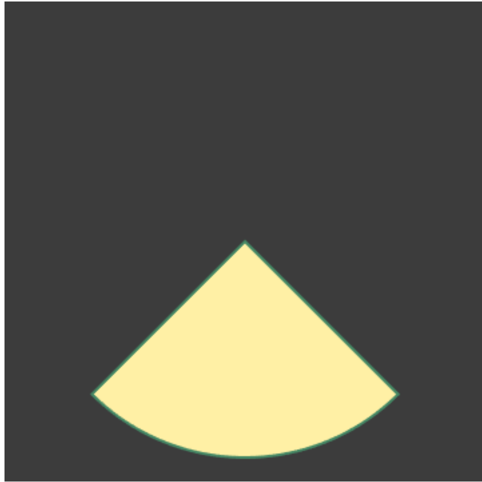


To draw the arc, we use the `arc(real x, real y, real radius, real startAngle, real endAngle, bool anticlockwise)` function. This function creates a circular path on the circumference of a circle, which is centered around `(x, y)` and has the given radius. To complete the pie segment, we also need to draw the lines that go from the circle's center to the arc's edges. Therefore we move the cursor's position to the center point, draw the arc, and then draw a line back to the center. The `arc()` function automatically generates a line from the cursor's initial position at the center to the starting point of the arc in addition to the arc itself.

```
/****************************************************************************
**
** Copyright (C) 2012 Digia Plc and/or its subsidiary(-ies).
** Contact: http://www.qt-project.org/legal
**
** $QT_BEGIN_LICENSE:BSD$
** You may use this file under the terms of the BSD license as follows:
**
** "Redistribution and use in source and binary forms, with or without
** modification, are permitted provided that the following conditions are
** met:
**   * Redistributions of source code must retain the above copyright
**     notice, this list of conditions and the following disclaimer.
**   * Redistributions in binary form must reproduce the above copyright
**     notice, this list of conditions and the following disclaimer in
```

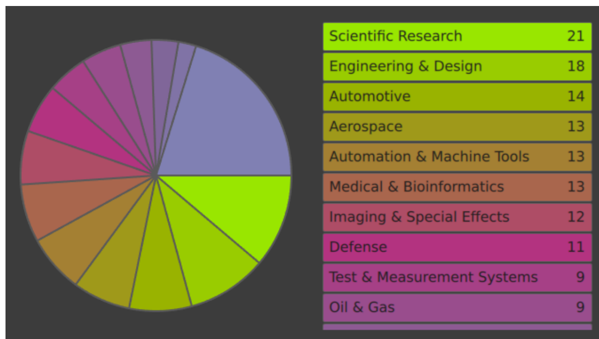
```
**     the documentation and/or other materials provided with the
**     distribution.
**   * Neither the name of Digia Plc and its Subsidiary(-ies) nor the names
**     of its contributors may be used to endorse or promote products derived
**     from this software without specific prior written permission.
**
**
** THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
** "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
** LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
** A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
** OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
** SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
** LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
** DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
** THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
** (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
** OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE."
**
** $QT_END_LICENSE$
**
****************************************************************************/
// begin a new path
ctx.beginPath()

// move the cursor to the center
ctx.moveTo(centerX, centerY)

// add the arc including the line to the beginning of the arc
ctx.arc(centerX, centerY, radius, startAngle, endAngle, anticlockwise)

// add the line back to the center
ctx.lineTo(centerX, centerY)

// fill the piece
ctx.fillStyle = fillStyle
ctx.fill()

// stroke the piece
ctx.lineWidth = lineWidth
ctx.strokeStyle = strokeStyle
ctx.stroke()
```

This is how the output of an application using the code described above looks:



3.3 Drawing the Chart

Next, we will draw the complete pie chart. After this step, the application looks like this:



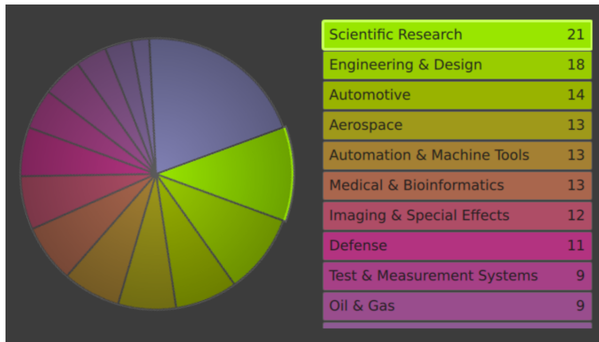
In the `onPaint` handler, we iterate over the model's elements and draw one pie sector for each entry (with its specific start and end angle, and filled with the previously assigned color). To make this set of pieces sum up to form a full circle, we also need to know the model's sum of values. We calculate this value in a JavaScript function.

```
...
    Canvas {
        id: canvas
        anchors.top: parent.top
        anchors.bottom: parent.bottom
        anchors.left: parent.left
        anchors.right: parent.horizontalCenter

        // enable anti-aliasing
        smooth: true

        onPaint: {
            var ctx = canvas.getContext('2d')

            ctx.clearRect(0, 0, width, height)

            // store the circles properties
            var centerX = width / 2
```

```
            var centerY = height / 2
            var radius = 0.9 * Math.min(width, height) / 2
            var startAngle = 0.0
            var endAngle = 0.0

            // calculate the factor that scales the angles
            // to make the sectors sum up to a full circle
            var angleFactor = 2 * Math.PI / modelSum()

            ctx.lineWidth = 2
            ctx.strokeStyle = Qt.lighter(root.color)

            // iterate over the model's elements
            for (var index = 0; index < model.count; index++) {

                // calculate the start and end angles
                startAngle = endAngle
                endAngle = startAngle + model.get(index).value * angleFactor

                ctx.fillStyle = model.get(index).color

                // draw the piece
                ctx.beginPath()
                ctx.moveTo(centerX, centerY)
                ctx.arc(centerX, centerY, radius, startAngle, endAngle, false)
                ctx.lineTo(centerX, centerY)
                ctx.fill()
                ctx.stroke()
            }
        }

        // calculate the model's sum of values
        function modelSum() {
            var modelSum = 0
            for (var index = 0; index < model.count; index++) {
                modelSum += model.get(index).value
            }
            return modelSum
        }
    }
...
```

3.4 Finalizing the Chart

In this chapter we'll enhance the pie chart's appearance and also provide interactivity. This is how the application looks after the enhancement:



In order to make the application more alive, we change the chart's appearance according to the list view's currently selected item. We add a mouse area to the list view and mark the current item with a highlight. To make the canvas respond to changes in the selection, we request a repaint whenever the current item changes.

```
...
    ListView {
        id: view
        anchors.top: parent.top
        anchors.bottom: parent.bottom
        anchors.left: parent.horizontalCenter
        anchors.right: parent.right
        anchors.margins: 16
        clip: true
        focus: true
        model: model
        delegate: Item {
            width: view.width
            height: 32
            Rectangle {
                anchors.fill: parent
                anchors.margins: 1
                radius: 2
                color: model.color
                border.color: Qt.lighter(root.color)
            }
            Text {
                anchors.verticalCenter: parent.verticalCenter
                anchors.left: parent.left
                anchors.margins: 8
                text: model.label
                color: "#1C1C1C"
            }
            Text {
                anchors.verticalCenter: parent.verticalCenter
                anchors.right: parent.right
                anchors.margins: 8
                text: model.value
                color: "#1C1C1C"
            }

            // handle mouse clicks
            MouseArea {
                anchors.fill: parent
                onClicked: {
                    view.currentIndex = index
```

```
                }
            }
        }

        // highlight the currently selected item
        highlight: Item {
            z: 10
            width: view.currentItem.width
            height: view.currentItem.height
            Rectangle {
                anchors.fill: parent
                anchors.margins: 1
                radius: 2
                color: "transparent"
                border.width: 3
                border.color: Qt.lighter(model.get(view.currentIndex).color)
                Behavior on border.color {
                    PropertyAnimation {}
                }
            }
        }

        // request a repaint of the canvas whenever
        // the currently selected item changes
        onCurrentIndexChanged: {
            canvas.requestPaint()
        }
    }
...
```

The pie chart sector representing the selected entry is also highlighted by increasing the sector's radius by 2%. We also want the highlighted sector to be on the right side of the chart, so we rotate the canvas based on the list view's `currentItem` property. To smooth this change, we apply a behavior to the rotation using a spring animation. Finally, we overlay a radial gradient from transparent white to a darker gray to further brush up the pie chart's appearance.

```
...
    Canvas {
        id: canvas
        anchors.top: parent.top
        anchors.bottom: parent.bottom
        anchors.left: parent.left
        anchors.right: parent.horizontalCenter
        smooth: true

        // animate the rotation
        Behavior on rotation {
            SpringAnimation { spring: 1; damping: 0.2 }
        }

        onPaint: {
            var ctx = canvas.getContext('2d')

            ctx.clearRect(0, 0, width, height)

            var centerX = width / 2
            var centerY = height / 2
```

```
            var radius = 0.9 * Math.min(width, height) / 2
            var radiusFactor = 1.0
            var startAngle = 0.0
            var endAngle = 0.0

            var angleFactor = 2 * Math.PI / modelSum()

            ctx.lineWidth = 2
            ctx.strokeStyle = Qt.lighter(root.color)

            for (var index = 0; index < model.count; index++) {
                startAngle = endAngle
                endAngle = startAngle + model.get(index).value * angleFactor

                // scale the currently selected piece and
                // rotate the canvas element accordingly
                if (index == view.currentIndex) {
                    radiusFactor = 1.02
                    canvas.rotation = - 180 / Math.PI * (startAngle +
                                      (endAngle - startAngle) / 2)
                } else {
                    radiusFactor = 1.0
                }

                ctx.fillStyle = model.get(index).color

                ctx.beginPath()
                ctx.moveTo(centerX, centerY)
                ctx.arc(centerX, centerY, radius * radiusFactor,
                        startAngle, endAngle, false)
                ctx.lineTo(centerX, centerY)
                ctx.fill()
                ctx.stroke()
            }

            // overlay a radial gradient
            var gradient = ctx.createRadialGradient(centerX, centerY,
                                   0, centerX, centerY, radius)
            gradient.addColorStop(0.0, Qt.rgba(1.0, 1.0, 1.0, 0.0))
            gradient.addColorStop(1.0, Qt.rgba(0.0, 0.0, 0.0, 0.3))
            ctx.beginPath()
            ctx.moveTo(centerX, centerY)
            ctx.arc(centerX, centerY, radius, 0, 2 * Math.PI, false)
            ctx.fillStyle = gradient
            ctx.fill()
        }

        function modelSum() {
            var modelSum = 0
            for (var index = 0; index < model.count; index++) {
                modelSum += model.get(index).value
            }
            return modelSum
        }
    }
...
```

Now we're done! We've successfully created a nice looking pie chart.

What's Next?

The next chapter demonstrates how to port HTML5 Canvas code into the Canvas Item of Qt Quick.

Porting HTML5 Canvas Code to Qt Quick

In this page, we want to show how easy it is to port existing HTML5 Canvas code to Qt Quick using the Canvas element.

Note: A general list of the necessary changes can be found in the Qt documentation for :qt5-snapshot: Qt Quick Canvas.

4.1 The HTML5 Canvas Code

We are going to port the spirograph[1] example used in this HTML5 Canvas Tutorial[2].

The HTML5 code looks like this:

```
/****************************************************************************
**
** Copyright (C) 2012 Digia Plc and/or its subsidiary(-ies).
** Contact: http://www.qt-project.org/legal
**
** $QT_BEGIN_LICENSE:BSD$
** You may use this file under the terms of the BSD license as follows:
**
** "Redistribution and use in source and binary forms, with or without
** modification, are permitted provided that the following conditions are
** met:
**   * Redistributions of source code must retain the above copyright
**     notice, this list of conditions and the following disclaimer.
**   * Redistributions in binary form must reproduce the above copyright
**     notice, this list of conditions and the following disclaimer in
**     the documentation and/or other materials provided with the
**     distribution.
```

[1] http://en.wikipedia.org/wiki/Spirograph

[2] https://developer.mozilla.org/en/Canvas_tutorial/Transformations

```
**   * Neither the name of Digia Plc and its Subsidiary(-ies) nor the names
**     of its contributors may be used to endorse or promote products derived
**     from this software without specific prior written permission.
**
**
** THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
** "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
** LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
** A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
** OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
** SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
** LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
** DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
** THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
** (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
** OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE."
**
** $QT_END_LICENSE$
**
****************************************************************************/
function draw() {
  var ctx = document.getElementById('canvas').getContext('2d');
  ctx.fillRect(0,0,300,300);
  for (var i=0;i<3;i++) {
    for (var j=0;j<3;j++) {
      ctx.save();
      ctx.strokeStyle = "#9CFF00";
      ctx.translate(50+j*100,50+i*100);
      drawSpirograph(ctx,20*(j+2)/(j+1),-8*(i+3)/(i+1),10);
      ctx.restore();
    }
  }
}
function drawSpirograph(ctx,R,r,O){
  var x1 = R-O;
  var y1 = 0;
  var i  = 1;
  ctx.beginPath();
  ctx.moveTo(x1,y1);
  do {
    if (i>20000) break;
    var x2 = (R+r)*Math.cos(i*Math.PI/72)
              - (r+O)*Math.cos(((R+r)/r)*(i*Math.PI/72))
    var y2 = (R+r)*Math.sin(i*Math.PI/72)
             - (r+O)*Math.sin(((R+r)/r)*(i*Math.PI/72))
    ctx.lineTo(x2,y2);
    x1 = x2;
    y1 = y2;
    i++;
  } while (x2 != R-O && y2 != 0 );
  ctx.stroke();
}
```

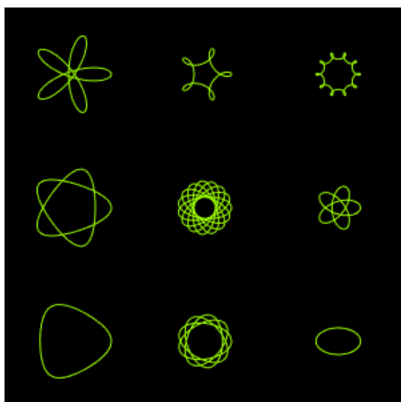
4.2 The Qt Quick Canvas Code

To port this code to a Qt Quick application, we can copy the code for drawing function into the `onPaint` handler of Qt Quick Canvas. We only need to change the line in which we acquire the drawing context: instead of using a DOM API call (`document.getElementById('canvas')`), we access the canvas directly. The JavaScript function can be inserted as a member function of the canvas.

```
/****************************************************************************
**
** Copyright (C) 2012 Digia Plc and/or its subsidiary(-ies).
** Contact: http://www.qt-project.org/legal
**
** $QT_BEGIN_LICENSE:BSD$
** You may use this file under the terms of the BSD license as follows:
**
** "Redistribution and use in source and binary forms, with or without
** modification, are permitted provided that the following conditions are
** met:
**   * Redistributions of source code must retain the above copyright
**     notice, this list of conditions and the following disclaimer.
**   * Redistributions in binary form must reproduce the above copyright
**     notice, this list of conditions and the following disclaimer in
**     the documentation and/or other materials provided with the
**     distribution.
**   * Neither the name of Digia Plc and its Subsidiary(-ies) nor the names
**     of its contributors may be used to endorse or promote products derived
**     from this software without specific prior written permission.
**
**
** THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
** "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
** LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
** A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
** OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
** SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
** LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
** DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
** THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
** (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
** OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE."
**
** $QT_END_LICENSE$
**
****************************************************************************/
import QtQuick 2.0

Canvas {
    id: canvas
    width: 300
    height: 300

    // the function "draw()" can be put into the "onPaint" handler
    onPaint: {

        // the acquisition of the rendering context needs to be adapted
```

```
        var ctx = canvas.getContext('2d');

        ctx.fillRect(0,0,300,300);
        for (var i=0;i<3;i++) {
            for (var j=0;j<3;j++) {
                ctx.save();
                ctx.strokeStyle = "#9CFF00";
                ctx.translate(50+j*100,50+i*100);
                drawSpirograph(ctx,20*(j+2)/(j+1),-8*(i+3)/(i+1),10);
                ctx.restore();
            }
        }
    }

    // the utility function "drawSpirograph()" can remain unchanged
    function drawSpirograph(ctx,R,r,O) {
        var x1 = R-O;
        var y1 = 0;
        var i  = 1;
        ctx.beginPath();
        ctx.moveTo(x1,y1);
        do {
            if (i>20000) break;
            var x2 = (R+r)*Math.cos(i*Math.PI/72)
                     - (r+O)*Math.cos(((R+r)/r)*(i*Math.PI/72))

            var y2 = (R+r)*Math.sin(i*Math.PI/72)
                     - (r+O)*Math.sin(((R+r)/r)*(i*Math.PI/72))

            ctx.lineTo(x2,y2);
            x1 = x2;
            y1 = y2;
            i++;
        } while (x2 != R-O && y2 != 0 );
        ctx.stroke();
    }
}
```

This is how the ported Qt Quick application looks like:



As you can see, it is surprisingly easy to use existing HTML5 Canvas code in your Qt Quick application.

What’s Next?

The next chapter concludes this tutorial.

CHAPTER 5

Conclusion

In this tutorial, we explored some of the capabilities of the Qt Quick Canvas type. We developed a nice looking pie chart visualization and ported HTML5 Canvas code to Qt Quick. If you want to learn more about the canvas API, you can refer to some of the HTML5 tutorials available on the internet.

Here is a small list of such tutorials:

- http://www.html5canvastutorials.com - A page dedicated to HTML5 Canvas tutorials
- https://developer.mozilla.org/en/Canvas_tutorial - An HTML5 Canvas tutorial on the Mozilla Developer Network
- http://dev.opera.com/articles/view/html-5-canvas-the-basics - HTML5 Canvas basics on the Opera developer portal
- http://www.canvasdemos.com - A page presenting HTML5 Canvas demos

The Qt Webkit Guide about canvas graphics[1] also gives a detailed introduction to the HTML5 Canvas API. Also note this very handy cheat sheet[2].

[1] http://qt-project.org/doc/qt-4.8/qtwebkit-guide-canvas.html
[2] http://blog.nihilogic.dk/2009/02/html5-canvas-cheat-sheet.html