Capítulo 2

Cross-platform Multimedia Application Development for Mobile, Web, Embedded and IoT with Qt/QML

Manoel C. M. Neto[1], Sandro S. Andrade[1] e Renato L. Novais[1]

[1]Grupo de Sistemas Distribuídos, Otimização, Redes e Tempo-Real – GSORT
Instituto Federal da Bahia (IFBA)
R. Emídio dos Santos, s/n
Barbalho, Salvador – Ba – Brasil, 40301-015

{manoelnetom,renato,sandroandrade}@ifba.edu.br

Abstract. *Qt is cross-platform development toolkit that enables the quick and cost-effective design, development, deployment, and maintainance of software for multiple platforms while delivering a seamless user experience across a range of devices. Over the past years, Qt has been successfully enabling advanced multimedia applications through a quite simple declarative language named QML (Qt Modeling Language). QML provides a set of high-level components for handling audio and video, animations, maps, cameras, and sensors, just to mention a few. Qt is widely used by several companies around the world, but its adoption in Brazil has been hindered by different factors. This short-course aims at presenting the main features Qt/QML provides regarding the development of cross-platform multimedia applications.*

Abstract. *[Resumo] O Qt é um toolkit para desenvolvimento multiplataforma de aplicações que viabiliza o desenvolvimento, implantação e manutenção de soluções de software de forma rápida e com um baixo custo. Ao longo do últimos anos, o Qt tem sido utilizado com sucesso no desenvolvimento de aplicações multimídia avançadas através de uma linguagem declarativa simples, denominada QML (Qt Modeling Language). A QML disponibiliza um conjunto de componentes de alto nível para manipulação de áudio e vídeo, animações, mapas, câmeras e sensores, somente para citar alguns. O Qt é amplamente usado no mundo inteiro mas seu uso no Brasil tem sido dificultado por uma série de fatores. Este minicurso tem como objetivo apresentar as principais funcionalidades do Qt/QML voltadas ao desenvolvimento de aplicações multimídia multiplataforma.*

2.1. Introdução

Ambientes multimídia interativos são aqueles nos quais a computação é usada para melhorar de forma imperceptível as atividades comuns do dia a dia [Neto 2015]. Uma das forças motrizes do interesse emergente nestes ambientes é tornar os computadores não apenas verdadeiramente amigáveis ao usuário, mas também essencialmente invisíveis para ele. Um ponto importante para alcançar este objetivo é a computação Ubíqua (UBiComp) [Krumm 2009].

A UbiComp, em seus vários desdobramentos e aplicações, é considerada por muitos como o novo paradigma da computação para o século XXI. Ela é a área da computação que estuda o acoplamento do mundo físico ao mundo da informação e fornece uma abundância de serviços e aplicações, permitindo que usuários, máquinas, dados e objetos do espaço físico interajam uns com os outros de forma transparente. Uma das dificuldades do desenvolvimento de aplicações para ambientes ubíquos é lidar com a diversidade de plataformas necessárias para o correto funcionamento desses sistemas. Um efeito colateral dessa dificuldade é o retrabalho necessário para adaptar a mesma aplicação para cada uma das possíveis plataformas. Neste contexto, o *framework* Qt [Russo and Vytiniotis 2009, The Qt Company 2016] surge como uma alternativa importante.

O Qt é um *toolkit* para desenvolvimento de interfaces gráficas de usuário (GUI) multiplataforma [Summerfield 2007], aplicações de comunicação via rede, suporte ao processamento de documentos XML, bancos de dados, multimídia, aplicações embarcadas, etc. Ele tem sido usado por diferentes aplicações (comerciais e *open-source*) desde 1995 em companhias e organizações como: Adobe, Boeing, IBM, Motorola, NASA, Skype. Uma contribuição importante do Qt para o desenvolvimento de aplicações focadas nas interfaces de usuários é a ***Qt Modeling Language*** (QML). A QML é uma linguagem cujos documentos descrevem uma árvore de elementos. Esses elementos podem ser combinados para construir desde componentes simples como botões até aplicações mais complexas.

Este artigo representa o texto base do minicurso apresentado no XXIII Simpósio Brasileiro de Sistemas Multimídia e Web – WebMedia. O curso é motivado no interesse de pessoas e organizações por temas como *Internet of Things* (IoT), sistemas embarcados e dispositivos móveis. Esta é uma tendência cujas aplicações já ganham espaço em companhias, cidades e na vida cotidiana de muitas pessoas. O minicurso apresenta as primeiras noções de como desenvolver uma aplicação multimídia multiplataforma através de exemplos práticos que usam funcionalidades do Qt/QML. Ele apresenta a construção de elementos encontrados em muitas aplicações comuns em ambientes multimídia interativos. Estes conceitos introdutórios são fundamentos importantes para diversas aplicações em diversas áreas.

2.2. Visão Geral

Um amplo conjunto de soluções para o desenvolvimento de software com Android, iOS, embarcados e outras plataformas está atualmente disponível no mercado. Desenvolver aplicações que executem não só no Android/iOS, mas também em um smartWatch ou smartTV é um dos desafios atuais da Computação, que geralmente implicam no desenvolvimento de duas ou mais soluções totalmente diferentes.

O Qt é um *toolkit* moderno para criação de aplicações gráficas e uma das poucas soluções atualmente disponíveis para desenvolvimento multiplataforma e *multi-device*. Ele pode ser utilizado não só para o Android e iOS, mas também para o Linux, Windows, macOS e uma série de plataformas embarcadas, sem a necessidade de duplicação de trabalho. O desenvolvimento do *toolkit* foi iniciado em 1995. Desde então, ele evoluiu e hoje provê uma base sólida para muitas aplicações [Blanchette and Summerfield 2006]. Este curso foca no Qt 5.9, versão mais atual do *framework*.

O suporte ao desenvolvimento multiplataforma do Qt é realizado através da utilização de uma arquitetura em camadas, onde aspectos particulares de uma determinada plataforma são tratados em um componente denominado *Qt Platform Abstraction* (QPA)[1]. Cada plataforma suportada pelo Qt possui uma diferente implementação do QPA, disponibilizando as funções básicas de renderização gráfica e outros serviços. Com isso, uma mesma API é mantida para todas as camadas acima da QPA, viabilizando a compilação do código-fonte e execução da aplicação em múltiplas plataformas. Todos estes aspectos são transparentes para o desenvolvedor usuário do Qt.

Outro aspecto importante do Qt são as extensões que o *toolkit* oferece à linguagem de programação C++. Tais extensões têm como objetivo minimizar uma série de problemas e deficiências característicos do desenvolvimento com C++, tais como *memory leaks*, ausência de reflexão computacional, objetos *composite* e comunicação baseada em eventos (via *signals* e *slots*). Tais extensões são tratadas pelo *meta-object compiler* (`moc`) – um pré-processador que converte código-fonte escrito com as extensões do Qt para código C++ puro. Assim, aplicações/bibliotecas podem ser compiladas por qualquer compilador C++ padrão como o `clang` ou `g++`. O *toolkit* possui diversos módulos que cobrem muitos requisitos e funcionalidades. Neste artigo, serão apresentados algumas dessas funcionalidades, especialmente aquelas voltadas para o desenvolvimento de aplicações *mobile*, web, embarcadas e para IoT.

2.2.1. Ambiente de Desenvolvimento

O Qt é disponibilizado como um *Software Development Kit* (SDKs) completo e sua instalação pode ser realizada em uma série de plataformas[2]. O SDK é de fácil instalação e já inclui uma IDE (*Integrated Development Environment*) denominada Qt Creator. O Qt Creator é um ambiente altamente produtivo para desenvolvimento Qt, recomendado para iniciantes e será o ambiente utilizado neste minicurso. A versão mais recente do SDK pode ser obtida do site `http://www.qt.io`. É importante ressaltar que o Qt também pode ser usado a partir de linhas de comando. Isso abre a possibilidade de usar outros editores de código no processo de desenvolvimento.

Para testar a instalação, abra o Qt Creator, crie um novo projeto do tipo *Qt Quick Controls 2* (*File → New File or Project → Qt Quick Controls 2 Application*) e nomeie-o `HelloWorld`. Depois disso, o Qt Creator irá criar vários arquivos automaticamente tais como o `HelloWorld.pro` e o `main.qml`. Arquivos `.pro` são usados para armazenar e gerenciar configurações relevantes do projeto. Os arquivos `.qml` armazenam o código-fonte de uma aplicação escrita em QML. Usando o QtCreator, copie e cole o código a

[1] `http://doc.qt.io/qt-5/qpa.html`
[2] `http://doc.qt.io/qt-5/supported-platforms.html`

seguir (Listagem 2.1) no arquivo main.qml e depois disso, selecione *Build* → *Run* no menu.

Listagem 2.1. Exemplo básico de QML

```
1  import QtQuick 2.6
2  Rectangle {
3      width: 360
4      height: 360
5      Text {
6          anchors.centerIn: parent
7          text: "Hello World"
8      }
9      MouseArea {
10         anchors.fill: parent
11         onClicked: Qt.quit();
12     }
13 }
```

2.3. Principais Módulos

Em função da grande quantidade de funcionalidades ofertadas, o Qt é dividido em módulos, possibilitando que o desenvolvedor indique quais partes do *toolkit* ele deseja utilizar. Um módulo é uma coleção de funcionalidades, disponibilizadas como APIs (*Application Programming Interfaces*) C++, QML ou ambos. Os módulos do Qt são divididos em dois grupos: *Qt Essentials* e *Qt Add-Ons*. Os módulos do grupo *Qt Essentials* contêm as funcionalidades mais comumente utilizadas, estão disponíveis em todas as plataformas suportadas e garantem compatibilidade binária entre todas as versões *major* do Qt (ex: toda a família de versões 5.x). Já os módulos do grupo *Qt Add-Ons* contêm funcionalidades relevantes apenas em casos mais específicos. As próximas seções apresentam alguns dos principais módulos do Qt usados em diferentes tipos de aplicações.

2.3.1. Qt Quick e Qt QML

O módulo `Qt QML` disponibiliza o interpretador (*engine*) QML e outros recursos de infraestrutura da linguagem. Já o módulo `Qt Quick` é responsável por disponibilizar todos os tipos básicos para criação de interfaces gráficas de usuário em QML. Estes módulos oferecem uma série de recursos: i) o interpretador QML; ii) um interpretador JavaScript; e iii) mecanismos para integração QML/C++. Eles permitem aos desenvolvedores de aplicativos tanto estender o QML com tipos personalizados quanto integrar o código QML com JavaScript e C++.

Assim como outras linguagens de marcação, um documento QML descreve uma árvore de objetos. Estes objetos incluem: um conjunto sofisticado de blocos de construção, gráficos (ex.: retângulo e imagem) e comportamentos (ex.: transição e animação) que podem ser combinados para criar componentes que variam em complexidade. Estes componentes podem ser botões simples e controles deslizantes, ou mesmo aplicativos móveis/embarcados muito mais complexos.

Nos documentos QML, cada objeto em uma árvore é especificado pelo tipo, seguido por um par de chaves. A Listagem 2.1 apresentou três objetos, um `Rectangle`

e seus dois filhos: `Text` e `MouseArea`. Entre as chaves, também podemos especificar atributos de um objeto. Cada objeto tem um atributo especial chamado `id`. Atribuir um `id` permite que um objeto seja referenciado por outros objetos.

O QML permite especificar o valor de um atributo de forma declarativa e isto é denominado *property binding*. Estes *bindings* são especificados como um par `atributo: valor`. Ao realizar um *binding*, o valor do atributo é atualizado automaticamente quando o valor for modificado. A Listagem 2.2 apresenta um exemplo de *binding*. O elemento `Rectangle` define a sua largura em função de `otherItem.width`, o que significa que ele irá possuir sempre o mesmo valor de largura quedo "otherItem".

Listagem 2.2. Exemplo de Binding

```
1 Rectangle {
2     function calculateMyHeight() {
3         return Math.max(otherItem.height, 300);
4     }
5     anchors.centerIn: parent
6     width: otherItem.width
7     height: calculateMyHeight()
8     color: (width > 10) ? "blue":"red"
9 }
```

A outra maneira de definir um *binding* é usar uma expressão JavaScript válida. Assim, os *bindings* podem acessar propriedades do objeto, fazer chamadas de função e até mesmo usar objetos JavaScript incorporados. No exemplo acima, um elemento `Rectangle` define sua própria altura ao se referir a uma função JavaScript chamada `calculateMyHeight`.

Outra característica importante do QML é o mecanismo para controlar mudanças de estado nas propriedades. Cada elemento possui um estado base "implícito". Cada mudança de estado é descrita listando as propriedades e valores dos elementos que diferem do estado base. No exemplo a seguir (Listagem 2.3), no estado base, myRect está posicionado no ponto (0, 0). No estado "moved", ele passa a estar posicionado no ponto (50, 50). Clicar dentro do `MouseArea` altera o estado para "moved", movendo assim o retângulo. O QML também suporta a animação de objetos. Para animar o exemplo acima, pode-se utilizar o elemento "transition".

Listagem 2.3. Exemplo de Binding e Animação

```
1  import QtQuick 2.6
2
3  Item {
4      id: myItem
5      width: 200; height: 200
6      Rectangle {
7          id: myRect
8          width: 100; height: 100
9          color: "red"
10     }
11     states: [
12         State {
```

```
13                name: "moved"
14                PropertyChanges {
15                    target: myRect
16                    x: 50; y: 50
17                }
18            }
19        ]
20        MouseArea {
21            anchors.fill: parent
22            onClicked: myItem.state = 'moved'
23        }
24        transitions: [
25            Transition {
26                from: "*"; to: "moved"
27                NumberAnimation {
28                    properties: "x,y"
29                    duration: 500
30                }
31            }
32        ]
33    }
```

2.3.2. Qt Multimedia

O módulo `Qt Multimedia` oferece uma série de funcionalidades relacionadas à manipulação de áudio, vídeos e outros artefatos multimídia. Ele fornece um rico conjunto de tipos QML e classes C++ para lidar com conteúdo multimídia além das APIs necessárias para suportar codificação e decodificação[3], acesso a recursos como câmera e microfone permitindo reproduzir/gravar som, vídeo ou tirar fotos.

Neste módulo, quatro elementos podem ser destacados: `MediaPlayer`, `Camera`, `VideoOutput` e `SoundEffect`. O elemento `MediaPlayer` permite controlar as funções relacionadas à reprodução de mídias (reproduzir, parar, pausar, etc). O elemento `VideoOutput` pode ser usado para reproduzir um vídeo. O elemento `Camera` é usado para capturar um fluxo de vídeo ao vivo. O elemento `SoundEffect` fornece uma maneira de reproduzir efeitos sonoros no QML.

Listagem 2.4. Exemplo de Funcionalidades de Multimídia

```
1  import QtQuick 2.5
2  import QtMultimedia 5.6
3
4  Item {
5      width: 1024
6      height: 600
7      MediaPlayer {
8          id: player
```

[3]É importante destacar que os recursos de decodificação e codificação são tratados através de `back-ends` específicos em cada plataforma (por exemplo, DirectShow usado no Windows ou QuickTime no OSX).

```
9           source: "trailer400p.ogg"
10      }
11      VideoOutput {
12          anchors.fill: parent
13          source: player
14      }
15      Component.onCompleted: {
16          player.play();
17      }
18  }
```

Na Listagem 2.19, um elemento `MediaPlayer`, denominado *player*, tem uma propriedade denominada *source* que aponta para o arquivo de vídeo trailer400p.ogg. A propriedade *source* de um elemento `VideoOutput` foi vinculada a *player*. Dessa forma, assim que o componente principal for completamente inicializado, a função `play()` do objeto *player* é chamada e a apresentação do vídeo é iniciada. O elemento `MediaPlayer` também possui outros atributos úteis. Por exemplo, os atributos de duração e posição podem ser usados para construir uma barra de progresso. Algumas outras operações básicas – como alterar o volume ao reproduzir a mídia – podem ser controladas através da propriedade de volume suportada por um elemento `MediaPlayer`.

Pode-se utilizar um elemento `Camera` para capturar imagens ou vídeos de uma câmera real e gerenciar as configurações de processamento aplicadas às imagens capturadas (Listagem 2.5). Por exemplo, para exibir um visualizador de imagens da câmera (*viewfinder*), pode-se combinar um `VideoOutput` com um `Camera` configurado como sua fonte. O elemento `Camera` contém um conjunto de sub-elementos que cobrem muitos recursos. Por exemplo, para capturar imagens, pode-se usar o elemento *Camera.imageCapture*. Quando o método de captura é chamado, uma foto é tirada. Isso significa que, primeiro o *Camera.imageCapture* emite um sinal *imagemCaptured* seguido de um sinal *imageSaved*. Se várias câmeras estiverem disponíveis, pode-se selecionar qual delas usar, vinculando o atributo *deviceId* a *QtMultimedia.availableCameras*. Em um dispositivo móvel, pode-se alternar convenientemente entre câmeras voltadas para a frente e para trás, definindo a propriedade da posição.

Listagem 2.5. Exemplo de uso da Câmera

```
1  VideoOutput {
2      anchors.fill: parent
3      source: camera
4  }
5
6  Camera { id: camera }
7
8  Button {
9      id: shotButton
10     text: "Take Photo"
11     onClicked: camera.imageCapture.capture();
12 }
```

Os elementos `SoundEffect` permitem reproduzir arquivos de áudio não com-

pactados (geralmente arquivos WAV) em um modo de latência geralmente menor e é adequado para sons de tipo *feedback* em resposta a ações do usuário (por exemplo, sons de teclado virtual, comentários para diálogos *popup* ou sons de jogos). Normalmente, o efeito sonoro deve ser reutilizado, o que permite que todas as análises e preparações sejam feitas antes do tempo e apenas sejam disparadas quando necessário. Isso é fácil de alcançar com o *property binding* do QML. O exemplo da Listagem 2.6 reproduz um arquivo WAV no clique do mouse.

Listagem 2.6. Exemplo de uso de Áudio

```
1  Text {
2      text: "Click Me!";
3      font.pointSize: 24;
4      width: 150; height: 50;
5
6      SoundEffect {
7          id: playSound
8          source: "soundeffect.wav"
9      }
10     MouseArea {
11         id: playArea
12         anchors.fill: parent
13         onPressed: playSound.play()
14     }
15 }
```

2.3.3. Qt Network

O Qt fornece classes para comunicação, integração com a web e para comunicação entre processos distribuídos através de uma rede de computadores. Essas classes estão agrupadas no módulo `Qt Network` que fornece uma camada de abstração para as operações de baixo nível e exibe apenas classes e funções de alto nível. O `Qt Network` também permite construir aplicações que usem protocolos de nível mais baixo, como TCP e UDP. Classes como `QTcpSocket` e `QUdpSocket` permitem que o desenvolvedor envie e receba mensagens usando estes protocolos. A resolução de nomes (DNS) – atividade comum em uma rede – é de responsabilidade da classe `QHostInfo`. Filtrar e redistribuir o tráfego de rede através de *proxies* pode ser tratado pela classe `QNetWorkProxy`.

O módulo `Qt Network` também fornece a API de *Bearer Management*. Esta API possui funções que podem iniciar ou interromper interfaces de rede e *roaming* entre pontos de acesso. No entanto, não ela não permite gerenciar as configurações de rede. Cada plataforma de cada aplicação deve assumir esta responsabilidade.

2.3.4. IoT e Qt

A quantidade de dispositivos e coisas que nos rodeiam estão aumentando rapidamente, tornando-se mais inteligentes e exigindo um software que funcione em uma maior variedade de hardwares que incluem dispositivos IoT com ou sem tela tai como: relógios inteligentes, televisores inteligentes, PCs, dentre outros. O Qt foi, desde sua concepção, projetado para criar interfaces de usuário e essa capacidade inata é necessária ao IoT.

Com o Qt, pode-se construir UIs simples ou complexas, clássicas ou modernas, padronizadas ou personalizadas. Porém, o mais importante no contexto da IoT é que tudo que é construído com o Qt é multiplataforma.

Outro benefício do Qt é o compartilhamento de software entre o *firmware* dos dispositivos embarcados e os aplicativos usados para controlá-los. A maioria das tecnologias usadas para RAD (*Rapid-Application Development*) obriga a usar, no ambiente de desenvolvimento, algo diferente do que está embarcado nos dispositivos. Isso significa, por exemplo, que podem haver duas instâncias do mesmo código, em duas linguagens, usando duas APIs, que precisam de dois conjuntos de testes e que podem gerar duas maneiras distintas de ocultar os erros. Com o Qt, o mesmo projeto, a mesma construção e os mesmos testes de um módulo serão compartilhados entre todos os locais em que o software será executado [Islam et al. 2015].

Para atender às demandas de IoT, o framework Qt oferece muitas APIs. O módulo `Qt Sensors`, por exemplo, possui dezenas de elementos e classes predefinidas que abstraem vários sensores existentes. Por exemplo, o elemento Accelerometer da QML é um *wrapper* da classe `QAccelerometer` e abstrai um acelerômetro real.

2.3.5. QtLite

Ao longo dos anos, o Qt foi usado em uma vasta gama de sistemas operacionais e dispositivos embarcados. Não demorou muito para que o Linux fosse tão importante para o Qt em embarcados quanto para *desktops*. Muitos outros sistemas operacionais embarcados seguiram essa tendência e o Qt suportou uma ampla gama de sistemas operacionais (Linux, Windows e vários sistemas operacionais de tempo-real). No entanto, para utilizar de forma eficiente o Qt nesses sistemas operacionais, e especialmente nos dispositivos embarcados, é necessário atender requisitos estritos de desempenho e consumo de memória. Muitas vezes é um desafio longo configurar o Qt para usar eficientemente os diferentes componentes de hardware, bibliotecas disponíveis ou retirar as partes do Qt que não são necessárias em um dispositivo embarcado.

A mais recente iniciativa Qt na área de IoT é o projeto QtLite. Ele tem como principal objetivo tornar o Qt uma estrutura muito mais direcionada, que facilite todo o ciclo de desenvolvimento e a vida útil dos produtos baseados em dispositivos embarcados. O QtLite disponibiliza um novo sistema de configuração que permite aos desenvolvedores definir as funcionalidades necessárias com mais detalhes, sem ter que incluir recursos desnecessários. O fluxo de trabalho de desenvolvimento também está sendo aprimorado para permitir que os desenvolvedores comecem com uma configuração mínima flexível que lhes permita adicionar recursos à medida que forem necessários.

2.4. Exemplo de Aplicativos Móveis QML

Nesta seção, serão apresentados exemplos de aplicativos móveis com um conjunto variado de componentes QML. Os aplicativos desenvolvidos contêm componentes básicos QML, conexão com banco de dados, utilização de serviços RESTful, uso de mídias e sensores. O código-fonte das aplicações está disponível em `http://github.com/GSORT/QML-Exemplos`.

2.4.1. Minhas Compras

Nesta seção, será apresentado um aplicativo que simula compras de produtos, focando em suas principais partes. O aplicativo faz uso de banco de dados (SQLite) e contém componentes básicos de um sistema em QML. A Figura 2.1 apresenta as telas do aplicativo.

Figura 2.1. Telas do aplicativo Minhas Compras

Para criar uma aplicação desse tipo, selecione *File* → *New File or Projects* → *Qt Quick Controls 2 Application*. Será criado uma aplicação com uma estrutura bastante comum em aplicativos atuais. Dois componentes comumente utilizados são o `SwipeView` e o `TabBar`, descritos na Listagem 2.7.

O `SwipeView` é composto de páginas (componente `Page`). O usuário pode navegar entre as páginas deslizando o dedo horizontalmente na tela do dispositivo. No código apresentado, existem duas páginas: `Compras` e `CadastrarCompras`. Estas duas páginas foram criadas como objetos definidos pelo usuário. Para fazer isso, basta salvar o arquivo com o nome do componente desejado e extensão .qml. A partir dai, é permitido referenciar o componente em outras partes do código. Tanto *Compras* quanto *CadastrarCompras* implementam o componente *Page*. O componente *TabBar* está associado à propriedade *footer* do *ApplicationWindow* (tela principal). Ele serve para criar opções de navegação para a aplicação. No nosso exemplo, tem dois *TabButton* (o código interno foi suprimido), os quais permitem visualizar um ícone e um texto.

Listagem 2.7. Exemplo de SwipeView e TabBar

```
1 SwipeView {
2     id: swipeView
```

```
3       anchors.fill: parent
4       width: applicationWindow.width
5       currentIndex: tabBar.currentIndex
6
7       Compras {
8           id: comprasPage
9       }
10
11      CadastrarCompra {
12          id: cadastrarCompraPage
13      }
14  }
15  footer: TabBar {
16      id: tabBar
17      currentIndex: swipeView.currentIndex
18
19      TabButton {...}
20      TabButton {...}
21  }
```

Na linha 5, há um *binding* entre o `SwipeView` e o `tabBar` (*currentIndex: tabBar.currentIndex*). Assim, quando o índice atual do `TabBar` for modificado, o índice do `SwipeView` também será. Da mesma forma, na linha 17, há o *binding* no sentido inverso (*currentIndex: swipeView.currentIndex*): quando o índice atual do `SwipeView` mudar o índice atual do `TabBar` também muda.

O Componente `Compras` (Listagem 2.8), que implementa a página "Minhas Compras", serve para listar os produtos comprados e o valor total gasto.

Listagem 2.8. Parte do Código Compras.qml

```
1  Page {
2      title: "Minhas Compras"
3
4      property double valorTotal
5      property alias listViewModel: listView.model
6
7      Label {
8          visible: !listView.model.count
9          text: "Nehuma compra registrada!"
10         anchors.centerIn: parent
11     }
12
13     ListView {
14         id: listView
15         model: ListModel { }
16         anchors.fill: parent
17
18         header: Rectangle {...}
19         delegate: Rectangle {...}
20     }
```

```
21  }
```

Neste página, há um `Label` para informar quando não há nenhum item comprado. Observe o *binding* que existe em sua propriedade *visible*. Ele só fica visível quando a lista de elementos está vazia. Esta lista de elementos, por sua vez, é implementado pelo componente `ListView`.

O `ListView` provê uma visão em lista para elementos de um modelo. O atributo *model* define o modelo como sendo do tipo `ListModel`, inicialmente vazio. Há ainda dois componentes: o *header* e o *delegate*. O *header* é o componente responsável por mostrar os totais das compras. Ele tem uma cor diferente e possui dois elementos de textos, um alinhado/ancorado à esquerda ("Valor Total") e o outro à direita (o total de fato). O *delegate* define um *template* para visualização de cada um dos itens do modelo.

A Listagem 2.9 apresenta um resumo do componente *delegate*, implementado como um `Rectangle`. Além de alguns atributos, ele possui um *MouseArea* que trata o evento sobre o retângulo. Foram especificados: i) um item `Column`, que serve para mostrar nome do produto (`Row` 1) e Valor Pago (`Row` 2). Aqui há um exemplo dos posicionadores do QML. O `Column` coloca os elementos um abaixo do outro. O `Row` por sua vez, um ao lado do outro. No caso dos dois `Rows` existentes, eles têm dois `Texts` para mostrar o *label* da informação e a informação propriamente dita (e.g., Produto: Laranja); ii) dois componentes `Text`. Eles são responsáveis por apresentar a data e a quantidade. Esses componentes estão ancorados à direita do *parent* (*delegate*); iii) por fim, há o `Divider` que é responsável por mostrar uma barra horizontal que divide os itens.

Listagem 2.9. Parte do Código do *delegate*

```
1  delegate: Rectangle {
2      id: delegate
3      width: parent.width; height: 65
4
5      property int _itemId: itemId
6      property int _qtde: qtde
7      property real _valor: valor
8
9      MouseArea {...}
10
11     Column {
12         anchors { fill: parent; margins: 16 }
13
14         Row {...}
15
16         Row {...}
17
18     }
19
20     Text {...}
21
22     Text {...}
23
```

```
24      Divider { }
25  }
```

Um ponto importante é como os elementos do *model* são preenchidos. Eles são preenchidos a partir dos dados que estão no banco de dados. O componente `DataBase` contém o código-fonte responsável por fazer as operações de acesso ao banco. Antes de apresentar o preenchimento do *model*, explicaremos o componente `DataBase`.

O componente `DataBase` é apresentado de forma resumida na Listagem 2.10. Ele foi definido como um *Item*, componente não-visual do QML. Inicialmente, foi definido um *id*: rootItem. Depois são definidos dois sinais. Os sinais são responsáveis por gerar eventos. O sinal *compraRegistrada(var compra)* é invocado quando uma compra é efetuada. Observe o parâmetro *var compra*. Ele é utilizado para armazenar a informação da compra realizada no momento. Desta forma, quem tiver interessado nesse evento, pode capturá-lo, realizando alguma operação. O sinal *comprasCarregado(var compras)* é utilizado após ter uma operação de seleção no banco, a qual retorna todas as compras realizadas até então. O parâmetro *var compras* é utilizado para armazenar a informação de todas as compras realizada no momento.

Listagem 2.10. Código para manipular o banco de dados

```
1  import QtQuick 2.0
2  import QtQuick.LocalStorage 2.0
3
4  Item {
5      id: rootItem
6
7      signal compraRegistrada(var compra)
8      signal comprasCarregado(var compras)
9
10     readonly property string mainTable: "create table if not
11     exists compras(id integer not null primary key autoincrement,
12     produto TEXT, valor float, qtde integer, data_compra integer)"
13
14     readonly property var database:
15     LocalStorage.openDatabaseSync("MinhasCompras",
16     "1.0", "Minhas Compras", 1000000)
17
18     Component.onCompleted: {
19         rootItem.database.transaction(function(tx) {
20         tx.executeSql(mainTable) })
21         carregarCompras()
22     }
23
24     function salvarCompra(produto, valor, qtde, dataCompra) {...}
25
26     function removerCompra(id) {...}
27
28     function carregarCompras() {...}
29 }
```

A seguir, tem-se duas definições de propriedades *readonly*. Elas são usadas para definir os comandos SQL que criam a tabela e o banco de dados "MinhasCompras". Essas duas variáveis são utilizadas dentro do *handler Component.onCompleted*. Como o próprio nome diz, esse evento é invocado logo após o componente ter sido construído. O código da linha 19 é responsável por criar o banco e a tabela da aplicação. Na linha 21, é realizada a chamada para a função *carregarCompras()*, responsável por realizar a operação de *select* no banco. A Listagem 2.10 apresenta ainda as funções *salvarCompra()* e *removerCompra()*, responsáveis por fazer o *insert* e o *delete* no banco de dados, respectivamente.

A Listagem 2.14 apresenta em detalhes o código da função *salvarCompra()*. Ela possui os parâmetros que são informados pelo usuário. Na função é possível observar o código responsável pelo *insert*, e logo após tem-se a chamada de *compraRegistrada*. Esse é um ponto importante. É ai que é invocado o sinal definido inicialmente. Observe que é definido um objeto chave:valor com os dados do elemento inserido. Desta forma, que tratar esse evento poderá ter acesso ao último registro de compra realizado. Na nossa aplicação, isso será utilizado para atualizar a lista de compras logo após a compra ter sido realizada.

Listagem 2.11. Função de Cadastrar uma Compra

```
1  function salvarCompra(produto, valor, qtde, dataCompra) {
2      var result = ({})
3      rootItem.database.transaction(function(tx) {
4          result = tx.executeSql('insert or replace into compras
5          VALUES(?, ?, ?, ?, ?)', [null, produto, valor, qtde,
6          dataCompra])
7          compraRegistrada({
8              "itemId": parseInt(result.insertId),
9              "produto": produto,
10             "valor": parseFloat(valor),
11             "qtde": parseInt(qtde),
12             "data_compra": dataCompra
13         })
14     })
15     return result.insertId
16 }
```

A Listagem 2.12 apresenta em detalhes o código da função *carregarCompras()*. Na função é possível observar o código responsável pelo *select* e, logo após, tem-se um *for* que constrói vários objetos definidos no mesmo formato chave:valor. Eles são inseridos em um vetor *result*. Por fim, é invocado o sinal *comprasCarregado*, passando a lista de compras. Desta forma, quem tratar este sinal poderá ter acesso a todas as compras realizadas. Na nossa aplicação, isso será utilizado para montar a lista de compras inicialmente.

Listagem 2.12. Função de Listar Compras

```
1  function carregarCompras() {
2      var result = []
3      rootItem.database.transaction(function(tx) {
```

```
4           var query = "SELECT * FROM compras"
5           var results = tx.executeSql(query, [])
6           for (var i = 0; i < results.rows.length; i++) {
7               var item = results.rows.item(i)
8               var objc = {
9                   "itemId": parseInt(item.id),
10                  "produto": item.produto,
11                  "valor": parseFloat(item.valor),
12                  "qtde": parseInt(item.qtde),
13                  "data_compra": item.data_compra
14              }
15              result.push(objc)
16              result = result
17          }
18      })
19      comprasCarregado(result)
20  }
```

Agora já é possível retornar para o preenchimento do *model* no componente *Compras*. No componente *main.qml* há um trecho de código responsável por instanciar o componente `Database` (Listagem 2.13). Além de instanciar o `Database`, permitindo que o banco de dados seja criado, é possível observar o tratamento dos dois sinais destacados anteriormente. Em QML, para capturar um sinal deve-se utilizar um *signal handler*, cujo nome é sempre *on* seguido do nome do sinal. Assim, *onCompraRegistrada* captura o sinal *compraRegistrada()* e *onComprasCarregado* captura o sinal *comprasCarregado()*. O primeiro efetua um *append* (adição) de um novo elemento no *model* definido em minhas compras. Por sua vez, o segundo preenche o *model* com todas as compras existentes.

Listagem 2.13. Código que preenche o *model* de compras

```
1  Database {
2      id: database
3      onCompraRegistrada: comprasPage.listViewModel.append(compra)
4      onComprasCarregado: {
5          for (var i = 0; i < compras.length; ++i)
6              comprasPage.listViewModel.append(compras[i])
7      }
8  }
```

A partir desse momento, explicaremos o componente *CadastrarCompra.qml*, responsável por fazer o cadastro de uma compra. A Listagem 2.14 apresenta um trecho de código desse componente. Além das propriedades *title* e *dataSelecionada*, ele tem os subcomponentes `DatePicker` e `Column`. Dentro de `Column` são definidos, um abaixo do outro, os componentes visuais `Label` e `TextField`, responsáveis pela representação visual da tela.

Listagem 2.14. Componente *CadastrarCompra.qml*

```
1  Page {
2      title: "Cadastrar Compra"
3
```

```
4       property string dataSelecionada: ""
5
6       DatePicker {
7           id: datePicker
8           onDateSelected: {
9               dataSelecionada =
10              "%1/%2/%3".arg(date.day).arg(date.month).arg(date.year)
11              dataCompra.focus = false
12          }
13      }
14
15      Column {
16          spacing: 10
17          anchors { fill: parent; margins: 16 }
18
19          Label { text: "Digite o nome do produto" }
20          TextField {...}
21
22          Label { text: "Digite o valor do produto" }
23          TextField {...}
24
25          Label { text: "Digite quantidade de itens comprado" }
26          TextField {...}
27
28          Label { text: "Digite a data da compra" }
29          TextField {
30              id: dataCompra
31              width: parent.width
32              readOnly: true
33              text: dataSelecionada
34              onFocusChanged: if (focus) datePicker.open()
35          }
36
37          Button {...}
38      }
39  }
```

O `DatePicker` é um componente responsável por apresentar um calendário[4]. Ele tem o tratamento de um sinal *dateSelected* que preenche o atributo *dataSelecionada*. Observe que *dataSelecionada* é utilizado para ajustar a propriedade *text* do `TextField` com *id* igual a "dataCompra".

A Listagem 2.15 apresenta o código referente ao componente *Button* de *CadastrarCompra.qml*. Nele há o tratamento de um sinal *clicked*, que é responsável por invocar a função *salvarCompra* de *Database.qml*.

[4]Ele foi implementando separadamente no arquivo `DatePicker.qml` (o mesmo não será explicado em detalhes aqui)

Listagem 2.15. Componente *Button CadastrarCompra.qml*

```
1  Button {
2      text: "Salvar"
3      width: parent.width
4      onClicked: {
5          var result = database.salvarCompra(produto.text,
6          valor.text, qtde.text, dataSelecionada)
7          if (result) {
8              showdialog("OK", "Compra registrada com sucesso!")
9              produto.text = valor.text = qtde.text
10             = dataSelecionada = ""
11         } else {
12             showdialog("Erro!", "N o foi poss vel salvar o item!")
13         }
14     }
15 }
```

Desta forma, concluímos a explicação das principais partes da aplicação Minhas Compras.

2.4.2. Serviços RESTful e Mapas

Nessa seção apresentaremos o aplicativo que lista as estações de metrô do Rio de Janeiro[5]. Os dados das estações estão disponíveis em formato JSON[6]. Nessa aplicação, utilizaremos serviços RESTful e mapas.

A Figura 2.2 apresenta as telas do aplicativo. Neste exemplo, também são utilizados os componentes básicos do exemplo da Seção 2.4.1: `SwipeView`, `Page`, `TabBar`, `Column`, *ListView*, etc. A tela da esquerda lista todas as estações. Ao clicar em uma das estações, é apresentado o mapa com a localização dessa estação (tela da direita).

A Listagem 2.16 apresenta o código que foi utilizado para construir a tela que lista todas as estações. Foi suprimido o código do componente `Column` (Linha 28), responsável por mostrar os dados da estação. Ele segue o mesmo padrão do exemplo anterior.

Listagem 2.16. Código que implementa a listagem de estações

```
1  Page {
2      id: page1
3      title: "Esta\c{c}\~oes de Metr\ o  - RJ"
4
5      BusyIndicator {
6          visible: firstPageView.count === 0
7          anchors.centerIn: parent
8      }
9      ListView {
```

[5]Utilizamos os dados abertos da prefeitura do Rio de Janeiro http://data.rio/group/transporte-e-mobilidade

[6]JSON disponível em http://dadosabertos.rio.rj.gov.br/metro/api/v1/rest/Estacoes.cfm?token=40A22AF7-A9C6-86D6-E211ABA64BF73830pretty=true

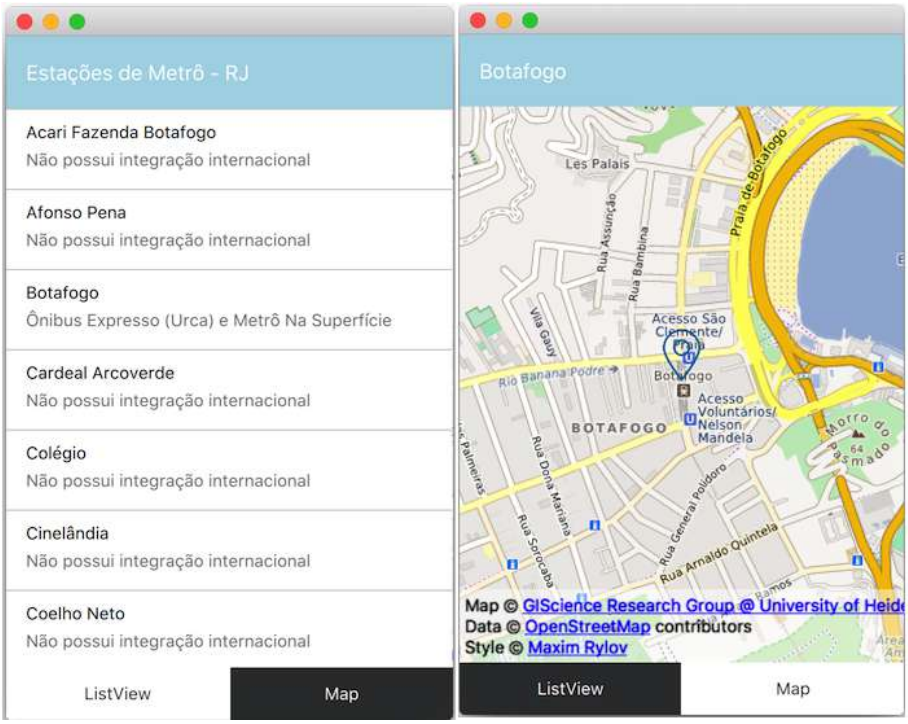

Figura 2.2. Telas da app Mapa Interativo

```
10          id: firstPageView
11          spacing: 1
12          anchors.fill: parent
13          delegate: Rectangle {
14              color: "#fff"
15              width: firstPageView.width; height: 60
16
17              MouseArea {
18                  anchors.fill: parent
19                  onClicked: {
20                      currentPlaceName = ESTACAO
21                      currentPlaceCoordinates =
22                      QtPositioning.coordinate(LATITUDE, LONGITUDE)
23                      swipeView.incrementCurrentIndex()
24                  }
25              }
26
27              Column {...}
28
29              Rectangle { width: parent.width; height: 1; color: "#ccc" }
30          }
31          model: ListModel {
32              id: firstPageModel
33          }
34
35          Component.onCompleted: {
```

```
36                 var baseUrl = "http://dadosabertos.rio.rj.gov.br/metro/api/v1
37                 /rest/Estacoes.cfm?token=40A22AF7-A9C6-86D6-E211ABA64BF73830&
38                 pretty=true&filter="
39                 requestHttp("GET", baseUrl, null, function(s,r) {
40                     for (var i = 0; i < r.length; ++i)
41                         firstPageModel.append(r[i])
42                 })
43             }
44         }
45 }
```

Nas linhas 5 a 8 define-se o componente *BusyIndicator*. Ele é utilizado para indicar que está havendo um processamento em *background*. No nosso caso, isso se refere à busca dos dados através do serviço RESTful. A partir da linha 10 é definido o componente `ListView`. Destacaremos as partes ainda não discutidas aqui nesse texto. O componente `MouseArea` (linhas 18-26) cria um evento de clique para cada elemento da lista. Quando o usuário clica em uma das estações é modificado o nome da estação atual e ajustadas as coordenadas atual para a estação selecionada. Os atributos ESTACAO, LATITUDE e LONGITUDE fazem parte do JSON recuperado. Por fim, é incrementado o índice do `SwipeView`, para abrir a página correspondente ao mapa.

Nas linhas 36-44 é implementado o *handler* `Component.onCompleted`. Ele é invocado após o componente ao qual ele faz parte ser carregado. Nesse evento, define-se a URL que contém os dados da estação e realiza-se uma requisição GET do HTTP. Por fim, há um laço que preenche o *model* (linhas 32-34) com cada um dos objetos do JSON.

A função que faz a requisição HTTP é apresentada na listagem 2.17.

Listagem 2.17. Função que faz a requisição HTTP

```
1  function parseJson(str) {
2      return str.toString().replace("<pre>", "").replace("</pre>", "")
3  }
4
5  function requestHttp(type, url, args, callback) {
6      var xhr = new XMLHttpRequest
7      xhr.open(type === "POST" ? "POST" : "GET", url, true)
8      xhr.setRequestHeader("Content-type", "Application/Json")
9      xhr.onreadystatechange = function() {
10         if (xhr.readyState === XMLHttpRequest.DONE) {
11             try {
12                 callback(xhr.status, JSON.parse(parseJson(xhr.responseText)
13             } catch(e) {
14                 console.log("Request error:")
15                 console.error(e)
16             }
17         }
18     }
19     xhr.send(args || ({}))
20 }
```

A Listagem 2.18 apresenta o código da página que mostra o mapa. Primeiramente, tem-se o componente `Plugin` (linhas 5-18), responsável por descrever os serviços baseado em localização. Esse componente serve para especificar qual o *plugin* GeoServices será utilizado em várias tarefas da API de localização. Os *plugins* são usados nos componentes `Map`, `RouteModel`, `GeocodeModel`, entre outros.

Listagem 2.18. Código que implementa o mapa

```
1  Page {
2      id: page2
3      title: currentPlaceName
4
5      Plugin {
6          id: mapPlugin
7          name: "osm"
8          PluginParameter { name: "osm.useragent";
9              value: "My First QML App" }
10         PluginParameter { name: "osm.mapping.host";
11             value: "http://osm.tile.server.address/" }
12         PluginParameter { name: "osm.mapping.copyright";
13             value: "Open Street Map" }
14         PluginParameter { name: "osm.routing.host";
15             value: "http://osrm.server.address/viaroute" }
16         PluginParameter { name: "osm.geocoding.host";
17             value: "http://geocoding.server.address" }
18     }
19
20     Map {
21         id: map
22         anchors.fill: parent
23         plugin: mapPlugin
24         center: currentPlaceCoordinates
25         zoomLevel: 15
26
27         MapQuickItem {
28             id: marker
29             coordinate: currentPlaceCoordinates
30             anchorPoint.x: image.width * 0.5
31             anchorPoint.y: image.height
32             sourceItem: Column {
33                 Image { id: image; source: "qrc:/marker.png";
34                 width: 45; height: width }
35             }
36         }
37     }
38 }
```

Em seguida, tem-se o componente `Map` (linhas 20-37), responsável por apresentar o mapa. Observe os seguintes atributos: *center*, que define a centralização do mapa. Este atributo tem um *binding* com *currentPlaceCoordinates*, definido no `MouseArea` expli-

cado anteriormente; *plugin*, que relaciona com o *plugin mapPlugin* definido; *zoomLevel* que define o nível de *zoom* do mapa.

Dentro do `Map` é definido um objeto `MapQuickItem`, o qual apresenta um objeto no mapa. Nesse caso, é apresentado um ícone (imagem definida na linha 33). O componente `MapQuickItem` possui o atributo *coordinate*, que define a posição da imagem (ícone) no mapa. Este atributo também possui um *binding* com *currentPlaceCoordinates*. A imagem é definida pela propriedade *sourceItem*.

2.4.3. Multimedia

Conforme mencionado anteriormente, o módulo `Qt Multimedia` disponibiliza uma série de recursos para manipulação de áudio e video. A Listagem 2.19 apresenta um exemplo de uso das câmeras frontal e traseira de um smartphone, com a exibição instantânea da imagem capturada.

Listagem 2.19. Exemplo de uso de câmera frontal e traseira

```
1  import QtQuick 2.4
2  import QtQuick.Controls 1.3
3  import QtQuick.Layouts 1.1
4  import QtMultimedia 5.4
5
6  Item {
7      id: root
8      anchors.fill: parent
9      property var cameras: QtMultimedia.availableCameras
10     property int currentCamera: 0
11
12     ColumnLayout {
13         anchors { fill: parent; margins: spacing }
14         VideoOutput {
15             Layout { fillWidth: true; fillHeight: true }
16             source: Camera { id: camera }
17             autoOrientation: true
18         }
19         RowLayout {
20             Label {
21                 Layout.fillWidth: true
22                 text: camera.displayName
23             }
24             Button {
25              visible: cameras.length > 1
26              text: "Change camera"
27              onClicked: {
28               currentCamera++
29               if (currentCamera >= cameras.length)
30                 currentCamera = 0
31         camera.deviceId =cameras[currentCamera].deviceId
32               }
33             }
```

```
34              }
35          }
36  }
```

Na linha 9, é criada uma propriedade denominada *cameras*, inicializada com um *array* de objetos que representam todas as câmeras disponíveis no dispositivo utilizado. A propriedade *currentCamera* (linha 10) armazena a câmera sendo atualmente utilizada. Em seguida, utiliza-se um objeto do tipo `VideoOutput` (linhas 14 a 18) para exibir a imagem sendo capturada pela câmera atual. Isto é realizado ao ajustar o atributo *source* para um objeto do tipo `Camera`.

Logo abaixo do objeto `VideoOutput`, são apresentados o nome da câmera sendo atualmente utilizada (através de um objeto `Label` – linha 22) e um botão para alternar a câmera atual (objeto `Button` – linha 24). Ao clicar neste botão, a aplicação alterna a câmera atualmente utilizada para a próxima câmera do array de câmeras disponíveis, retornando para a primeira quando não há mais câmeras disponíveis (linhas 27 a 33).

2.4.4. Sensores

O módulo `Qt Sensors` disponibiliza um amplo conjunto de funcionalidades para acesso a sensores tais como acelerômetro, sensor de proximidade, altímetro, sensor de luz, sensor de temperatura, giroscópio, sensor de humidade, dentre outros. Grande parte destes recursos funcionam de maneira uniforme em uma ampla faixa de dispositivos.

A Listagem 2.20 apresenta um exemplo de uso de acelerômetro e sensor de proximidade. O exemplo funciona em qualquer smartphone Android ou iOS que possua estes dois recursos. O aplicativo permite que um pequeno círculo verde tenha a sua posição (x, y) controlada através do acelerômetro do smartphone. Adicionamente, a ativação do sensor de proximidade modifica a cor do círculo de verde para preto.

Listagem 2.20. Exemplo de uso de acelerômetro e sensor de proximidade

```
1  import QtQuick 2.8
2  import QtQuick.Controls 2.1
3  import QtSensors 5.8
4  import QtQuick.Window 2.0
5
6  ApplicationWindow {
7      id: window
8      width: 640; height: 480
9
10     property var pixelDensity
11
12     Rectangle {
13         id: ball
14         x: window.width/2 - width/2; y: window.height/2 - height/2
15         width: 24*pixelDensity; height: 24*pixelDensity
16         color: "green"; radius: width/2
17         Behavior on x { NumberAnimation { duration: 100 } }
18         Behavior on y { NumberAnimation { duration: 100 } }
```

```
19      }
20      Accelerometer {
21          active: true; dataRate: 20
22          onReadingChanged: {
23              var newX = ball.x - reading.x * pixelDensity * 30
24              var newY = ball.y + reading.y * pixelDensity * 30
25              if (newX + ball.width > window.width)
26                  newX = window.width - ball.width
27              if (newY + ball.height > window.height)
28                  newY = window.height - ball.height
29              if (newX < 0) newX = 0
30              if (newY < 0) newY =  0
31              ball.x = newX
32              ball.y = newY
33          }
34      }
35      ProximitySensor {
36          active: true; dataRate: 20
37          onReadingChanged: ball.color = (reading.near) ? "black":"green"
38      }
39
40      Component.onCompleted: pixelDensity = Screen.logicalPixelDensity
41  }
```

A linha 3 realiza o import do módulo `Qt Sensors`, fazendo com que os objetos QML relacionados a sensores estejam disponíveis para uso na aplicação. O círculo verde a ser controlado é criado nas linhas 12 a 19 (no QML, um círculo é representado como um retângulo com o atributo *radius* ajustado para metade da sua largura). O acesso ao acelerômetro é realizado nas linhas 20 a 34, onde um *handler* para o signal readingChanged realiza o posicionamento do círculo de acordo com os dados coletados no acelerômetro. As linhas 35 a 38 realizam a leitura do sensor de proximidade.

2.5. Conclusão

Em um ambiente multimídia interativo, os dispositivos do usuário são muito heterogêneos. Embora alguns dispositivos possam ter gráficos, endereços IP, telas sensíveis ao toque (painéis de controle de casa, por exemplo), a maioria dos casos pode ter muito menos recursos. Os usuários precisam ter uma maneira de se comunicar com seus dispositivos, configurar os seus comportamentos, obter dados ou visualizá-los e alterar o seu status. Nestes ambientes, os requisitos de experiência do usuário (UX) transcendem um simples LED intermitente ou uma interface de um botão e é parte importante para o sucesso de um ambiente multimídia interativo.

Mais do que nunca, o mundo da IoT depende de software para criar dispositivos mais inteligentes e conectados. Os critérios para a sua escolha das tecnologias de software em um projeto ubíquo pode ou não coincidir com os que definimos neste artigo. No entanto, se seu projeto precisa ser conectado, desenvolvido rapidamente, multiplataforma com interfaces de usuário e recursos de compartilhamento, então você deve considerar o uso do Qt.

Referências

Blanchette, J. and Summerfield, M. (2006). *C++ GUI Programming with Qt 4*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

Islam, S. M. R., Kwak, D., Kabir, M. H., Hossain, M., and Kwak, K. S. (2015). The internet of things for health care: A comprehensive survey. *IEEE Access*, 3:678–708.

Krumm, J. (2009). *Ubiquitous Computing Fundamentals*. Chapman & Hall/CRC, New York, NY, USA, 1st edition.

Neto, M. C. M. (2015). Desenvolvimento de aplicações ubíquas com arduino e raspberry pi. In *Proceedings of the 20st Brazilian Symposium on Multimedia and the Web*, pages 1–40. SBC.

Russo, C. V. and Vytiniotis, D. (2009). Qml: Explicit first-class polymorphism for ml. In *Proceedings of the 2009 ACM SIGPLAN Workshop on ML*, ML '09, pages 3–14, New York, NY, USA. ACM.

Summerfield, M. (2007). *Rapid GUI Programming with Python and Qt : the Definitive Guide to PyQt Programming*.

The Qt Company (2016). Qt : cross-platform application and ui framework.

Biografia Resumida dos Autores

Manoel C M Neto

Manoel Carvalho Marques Neto é professor do Instituto Federal de Educação, Ciência e Tecnologia da Bahia (IFBa), Doutor em Ciência da Computação pela Universidade Federal da Bahia (UFBa - 2011), Mestre em Redes de Computadores (2004) e graduação em Análise de Sistemas (2002), ambos pela Universidade Salvador (UNIFACS) . É pesquisador-membro do Grupo de Pesquisa em Sistemas Distribuídos, Otimização, Redes e Tempo-Real (GSORT - IFBa). Suas atuais áreas de interesse incluem: IoT, Computação Ubíqua, software livre, multimídia, web, mobile, entre outras. CV: http://lattes.cnpq.br/7300048297400666

Renato L Novais

Renato Lima Novais é doutor em Ciência da Computação pela Universidade Federal da Bahia (2013), Mestre em Informática pela Pontifícia Universidade Católica do Rio de Janeiro (2007), e graduado em Ciência da Computação pela Universidade Federal da Bahia (2004). Durante o doutorado realizou estágio sanduíche no Fraunhofer Center for Experimental Software Engineering, Maryland, USA. É professor do Instituto Federal de Educação, Ciência e Tecnologia da Bahia (IFBA). Tem experiência nas áreas de Engenharia de Software, Big Data e Smart Cities, atuando principalmente nos seguintes temas: visualização de informação, compreensão e evolução de software, e engenharia de software experimental. CV: http://lattes.cnpq.br/5036544358013553

Sandro S Andrade

Sandro Santos Andrade é professor do Instituto Federal de Educação, Ciência e Tecnologia da Bahia (IFBa), Doutor em Ciência da Computação e Mestre em Mecatrônica, ambos pela Universidade Federal da Bahia (UFBa). É pesquisador-colaborador do Laboratório de Sistemas Distribuídos (LaSiD - UFBa) e pesquisador-membro do Grupo de Pesquisa em Sistemas Distribuídos, Otimização, Redes e Tempo-Real (GSORT - IFBa). Suas atuais áreas de interesse incluem: arquitetura de software, software livre, engenharia de software experimental, engenharia de software baseada em busca, sistemas distribuídos, sistemas self-adaptive/self-organizing e design science. Trabalhou por muitos anos em projetos envolvendo computação gráfica aplicada à medicina, desenvolvimento de jogos, visão computacional e computação musical. CV: http://lattes.cnpq.br/0177714301545658