Construindo um jogo simples com o *QCanvas* da Qt

Uma tela diferente

A biblioteca gráfica Qt da Trolltech tem recursos atraentes para as necessidades de qualquer desenvolvedor, mas um dos componentes mais fascinantes e poderosos desse conjunto é a classe *QCanvas*.

POR GEORGE WRIGHT

O *QCanvas* é um elemento de interface gráfica (ou *widget*) muito versátil que permite acrescentar gráficos 2D de alto desempenho a um aplicativo Qt. Com recursos como detecção de colisões e suporte a *sprites* (elementos de imagem), o *QCanvas* é muito adequado a jogos 2D. Mas ele também é usado em diversos aplicativos, como o *KTurtle*, um interpretador da linguagem de programação *Logo*. Neste artigo, mostrarei como construir um jogo simples usando componentes do *QCanvas* [1].

O jogo de exemplo que discutirei neste artigo, a que chamarei *Tijolinhos* (um clone do clássico *Breakout*, da Atari), consiste numa área retangular (nosso campo de jogo) com diversos objetos retangulares. Esses objetos são os *tijolinhos*. Uma bola quica pela janela do programa; se a bola atingir um tijolo, ele se "quebra" e desaparece. Uma raquete, controlada pelo usuário, demonstra como é possível dar interatividade ao jogo. Para manter as coisas simples, a bola não pode sair do campo.

Você pode montar esse jogo rápida e facilmente usando os componentes do *QCanvas*. Os passos para construí-lo são os seguintes:

- Definir a função principal, que será o ponto de entrada do aplicativo
- Definir a janela principal do jogo, chamada *View* (ou Visão), que servirá como modelo para os outros elementos gráficos.
- Definir e implementar a bola
- Implementar a detecção de colisões (para fazer a bola quicar nos cantos da tela e acertar os tijolos)
- Definir e implementar o objeto tijolo
- Definir e implementar a raquete

O jogo dos Tijolinhos é obviamente um exemplo muito simples, mas será uma base suficiente para que você comece a fazer experiências com seus próprios programas QCanvas, se isso o interessar. Este tutorial mostra como é possível escrever

Quadro 1: Como conseguir o QCanvas

O *QCanvas* [1] é incluído como padrão na distribuição Qt, que está no site da Trolltech. Se quiser ganhar dinheiro com ele, é preciso uma licença comercial. Porém, se você planeja apenas escrever um programa que será lançado sob uma licença apropriada de Software Livre, precisará baixar apenas o *Qt Free Edition*, licenciado sob a GPL. Quando este artigo foi escrito, o mais recente lançamento do *Qt Free Edition* era o 3.3.3. O endereço para obtê-lo é ftp://ftp.trolltech.com/qt/source/qt-x11-free-3.3.3.tar.bz2.

A instalação do pacote segue a rotina padrão de pacotes fonte, embora o Qt seja diferente por instalar as bibliotecas Qt em seu próprio diretório. Portanto, será preciso extrair o tarball num diretório apropriado (como `/usr/lib/qt`) e compilar usando os seguintes comandos:

```
# cd /usr/lib/qt
# ./configure -system-zlib -qt-gif -system-libpng ➥
-system-libjpeg -plugin-imgfmt-mng -thread -no-stl ➥
-no-xinerama -no-g++-exceptions
# make
```

O Qt será então instalado em `/usr/lib/qt`. É preciso configurar a variável de ambiente `QTDIR` para apontar esse diretório. Não é preciso rodar `make install`, já que as bibliotecas Qt são instaladas automaticamente no diretório fonte principal.

Porém, é muito provável que sua distribuição já inclua o Qt como um pacote, de forma que freqüentemente é melhor instalar o pacote de desenvolvimento do Qt a partir de sua mídia de instalação original. Ele é normalmente chamado `qt-devel` ou algo assim. Sua distribuição deve ter uma ferramenta para instalar esse pacote, como o YaST no SuSE ou o APT no Debian.

um jogo 2D verdadeiro usando apenas as bibliotecas Qt e um pouquinho de código para “grudar” tudo isso.

Começando a escrever o programa

Ao começar a escrever o programa, partimos do ponto de entrada do aplicativo, que normalmente está em `main.cpp`. O formato de `main.cpp` é exatamente o mesmo para qualquer outro aplicativo escrito em Qt, começando com a inclusão do arquivo de cabeçalho do `QApplication`, `qapplication.h`. Essa é a classe fundamental para qualquer aplicativo Qt, já que nos permite usar as classes Qt em nosso programa. A função principal é muito simples, englobando apenas o código para carregar um elemento gráfico Qt como janela principal. Para começar, é criado um objeto `QApplication`, que representará o aplicativo. Esse é seguido pelo elemento gráfico principal, chamado *View*, uma classe que será escrita posteriormente. Esse elemento gráfico servirá como janela principal do aplicativo, sobre a qual será colocada a tela. Como essa é a janela principal, a função QApplication `setMainWidget()` é chamada para usá-la como tal (ver **listagem 1**).

Os parâmetros atribuídos ao construtor do `QApplication` foram definidos como valores passados ao aplicativo após sua execução, de forma que o próprio Qt pode lidar com eles. Para exibir o elemento gráfico, devemos chamar a função `show()` para o objeto `View`; de outro modo o objeto é criado, mas nada aparece na tela. Finalmente, a função de retorno informa se a `QApplication` é executada sem erros – ou não.

A classe *View*

Agora que o `main.cpp` está completo, é hora de passarmos à classe mais importante do jogo, `View`. Essa é a janela principal do programa e precisa ser herdada de `QWidget` para ser configurada como o principal elemento gráfico do `QApplication`. Nesse caso, provavelmente é melhor fazer com que a classe herde `QMainWindow` e basear todo o aplicativo nessa classe. A razão para isso é que `QMainWindow` tem uma função `setCentralWidget()` muito útil, que nos permitirá acrescentar o elemento gráfico principal (ou seja, a janela do aplicativo) a ele utilizando o layout padrão, permitindo assim que ele mexa no layout e nos eventos de redimensionamento para o aplicativo.

Listagem 1: Chamando o *setMainWidget()*

```
01 #include <qapplication.h>
02 #include "view.h"
03 int main(int argc, char **argv)
04 {
05  QApplication a(argc, argv);
06  View *view = new View();
07
08  a.setMainWidget(view);
09  view->show();
10
11  return a.exec();
12 }
```

Em primeiro lugar, a classe precisa ser definida em `view.h`:

```
class View : public QMainWindow
{
 public:
  View(QWidget* parent = 0, ➘
  const char *name = 0);
  ~View();
};
```

Essa é uma definição para uma classe chamada `View` que herdará `QMainWindow`. Será usada como a janela principal do aplicativo, de forma que os objetos que criarmos a usarão como modelo. Os elementos gráficos do *QCanvas* serão adicionados a essa janela. Agora que temos uma definição de classe, podemos começar a escrever o construtor da classe em `view.cpp`:

```
View::View(QWidget* parent, const char ➘
*name)
 : QMainWindow(parent, name)
{
}
View::~View()
{
}
```

Esse é o construtor padrão da classe *View*, que passa seus argumentos para `QMainWindow`. Porém, um construtor vazio é bastante aborrecido, já que não faz realmente nada; assim, a próxima coisa a fazer é criar alguns elementos gráficos para exibir na janela.

O *QCanvas* precisará de um elemento gráfico `QCanvas` e um `QCanvasView` para que as classes do *QCanvas* possam ser usadas. Para adicioná-las à janela, elas terão de ser declaradas na definição de classe como `private` sob as declarações de classe públicas:

```
private:
 QCanvas *m_canvas;
 QCanvasView *m_canvasView;
```

Agora elas podem ser criadas no construtor como objetos CLASS-WIDE e o *QcanvasView* pode ser configurado como o elemento gráfico principal da janela. Teremos assim um espaço de trabalho *QCanvas* em que objetos 2D podem ser adicionados ao jogo. Para criar os objetos *QCanvas* e *QCanvasView* usamos o seguinte código no construtor:

```
m_canvas = new QCanvas(this);
m_canvas->resize(400, 300);
m_canvasView = new QCanvasView (m_canvas, ➘
this);
m_canvasView->show();
```

A primeira linha cria uma tela chamada `m_canvas` tendo a janela como seu objeto pai, enquanto a segunda linha ajusta o tamanho da tela para adequá-lo ao jogo. A terceira linha cria um `QCanvasView` para que o usuário visualize a tela, tornando `m_canvas` seu objeto *QCanvas.* Agora que os principais objetos do jogo foram criados, é possível adicionar o `QCanvasView` à janela como o elemento gráfico central:

```
setCentralwidget(m_canvasView);
```

Antes de compilar, é preciso incluir os arquivos de cabeçalho relevantes. Todas as classes do *QCanvas* estão em `qcanvas.h` e a classe `QMainWindow` está definida em `qmainwindow.h`, de forma que só é necessário o seguinte acima de `view.h`:

```
#include <qcanvas.h>
#include <qmainwindow.h>
```

Também é preciso dizer ao `view.cpp` para incluir o `view.h`, já que ele contém sua declaração de classe:

```
#include "view.h"
```

Listagem 2: Declarando *Ball*

```
01 #include <qcanvas.h>
03 class Ball : public QCanvasEllipse
04 {
05  public:
06   Ball(QCanvas *canvas);
07   ~Ball();
08  private:
09   double vx;
10   double vy;
11 };
```

Pronto! isso completa a nossa classe `View` no momento.

A bola

Agora que temos uma classe que define a janela verdadeira, podemos começar a escrever a classe que controla o movimento da bola. Essa classe herdará o `QCanvasEllipse`, pois este é o `QCanvasItem` para objetos circulares e elípticos. Como esse objeto se moverá, precisaremos lançar mão do sistema utilizado pelo *QCanvas* para objetos em movimento. O *QCanvas* define a velocidade como dois componentes: a velocidade na direção horizontal (`xVelocity`) e a velocidade na direção vertical (`yVelocity`). Combinados, esses componentes podem criar uma velocidade em qualquer direção desejada.

Para declarar a classe da bola, usarei uma curta declaração de classe em `ball.h`, que manda que essa classe herde o `QCanvasEllipse` (ver **listagem 2**).

As duas variáveis declaradas na seção *private* são valores que representam as velocidades *x* e *y* da bola. As funções que modificam a velocidade da bola farão ajustes nessas variáveis e definirão a verdadeira velocidade em seguida. Também é mantido um registro interno da velocidade atual. O *QCanvas* usa duas *fases* no movimento de um objeto. A primeira é a fase 0, em que o objeto não deve se mover, mas realizar uma inspeção no objeto tela para ver se é necessário aplicar algum caso especial (como uma colisão) que afetará a velocidade. A segunda é a fase 1, que diz ao objeto para simplesmente se mover segundo seus valores de velocidade (quando a gravidade e a resistência podem ser aplicadas, caso necessário). Essas fases são implementadas na função `advance()` de forma que a classe `Ball` precisa reimplementá-las para poder realizar as operações apropriadas no caso de uma colisão. Para sobrecarregar a função, acrescentamos o seguinte à seção `public` da declaração de classe:

```
void advance(int phase);
```

Também é sensato declarar uma função à parte, na qual o código de detecção de colisão é chamado, assim como uma função para realizar as operações quando ocorre uma colisão:

```
void collisionDetect();
void collide(QCanvasItem *item);
```

E, finalmente, deve haver também outra função para atualizar a velocidade do objeto a partir das variáveis internas `vx` e `vy`:

```
void updateVelocities();
```

Agora que o `ball.h` foi totalmente escrito, é hora de passar ao arquivo `ball.cpp`. Primeiramente, precisamos dizer ao `ball.cpp` que inclua `ball.h` e em seguida declarar as diversas funções (ver **listagem 3**).

A **listagem 3** é um esqueleto simplificado de `ball.cpp` com todas as funções. A terceira linha diz ao `QCanvasEllipse` que há uma elipse com altura e largura de 10

Listagem 3: *ball.cpp*

```
01 #include “ball.h”
02 Ball::Ball(QCanvas *canvas)
03  : QCanvasEllipse(10, 10, canvas)
04 {
05 }
06 Ball::~Ball()
07 {
08 }
09 void Ball::advance(int phase)
10 {
11 }
12 void Ball::collisionDetect()
13 {
14 }
15 void Ball::collide(QCanvasItem *item)
16 {
17 }
18 void Ball::updateVelocities()
19 {
20 }
```

Listagem 4: Teste de Colisão

```
01 double nx = x() + xVelocity();
02 double ny = y() + yVelocity();
03 vx = xVelocity();
04 vy = yVelocity();
05 if ((nx - (width() / 2)) < 0 || (nx + (width() / 2)) > canvas()->width())
06  vx = -vx;
07 if ((ny - (height() / 2)) < 0 || (ny + (height() / 2)) > canvas()->height())
08  vy = -vy;
09 // Check for collisions
10 QCanvasItemList colList = collisions(true);
11 for (QCanvasItemList::Iterator it = colList.begin(); it != colList.end(); ++it) {
12  QCanvasItem *check = *it;
13  if ((check->collidesWith (this))) {
14   collide(check);
15  }
16 }
17 updateVelocities();
```

pixels – um círculo, portanto. Também diz ao `QCanvasEllipse` que a tela pai é a que foi passada para o construtor.

O `QCanvasItem`, do qual o `QCanvasEllipse` é herdado, pinta automaticamente esse item de branco. Porém, essa não é exatamente uma boa idéia, já que a própria tela é branca, o que torna a bola invisível. Por causa disso, no construtor de classes temos de dizer com todas as letras ao `QCanvasEllipse` que a pinte de preto. Isso também faz com que ela seja um círculo cheio em vez de um contorno:

```
setBrush(Qt::black);
```

Eis assim todo o necessário no construtor de classes. Tudo o mais é implementado nas funções `advance()`, `collisionDetect()`, `collide()` e `updateVelocities()`. O *QCanvas* cuidará de chamadas antecipadas quando necessário. Como mencionado anteriormente, há duas fases para o movimento nas classes `QCanvasItem`, que são representadas pelo argumento `int phase` passado à função `advance()`. Na fase 0 precisamos buscar colisões e na fase 1 devemos mover a bola de acordo com suas velocidades definidas:

```
#The advance() function
if (phase == 0) {
 collisionDetect();
} else {
 moveBy(xVelocity(), yVelocity());
}
```

No momento, o `collisionDetect()` não faz realmente nada. Porém, o *QCanvas* oferece poderosas funções embutidas de detecção de colisões. O `collisionDetect()` as usará para detectar uma possível colisão. No *QCanvas*, elas consistem em duas funções principais: `collisions()` e `collidesWith()`. A primeira função retorna um ponteiro para uma `QCanvasItemList` com todos os objetos atualmente na tela com os quais o objeto atual colidiu após ter se movido de acordo com suas velocidades atuais. A segunda função retorna um valor booleano informando se aquele objeto colidiu naquele momento com algum outro objeto determinado. Podemos portanto iterar através de todos os objetos no `QCanvasItemList` retornados por `collisions()` e testá-los para uma colisão atual usando `collidesWith()` (ver **listagem 4**).

As primeiras duas linhas declaram dois números em ponto flutuante de precisão dupla (double), que serão as novas posições `x` e `y` da bola após as velocidades terem sido aplicadas. O código passa então a ajustar os valores `vx` e `vy` internos às velocidades atuais `x` e `y` da bola. As duas declarações `if` fazem uma inspeção na futura posição da bola para ver se ela sairá totalmente da tela e para reverter a velocidade na direção apropriada, caso necessário. Por exemplo, se a bola for sair da tela pelo lado direito, elas reverterão a velocidade da bola para que ela comece a se mover da direita para a esquerda e não da esquerda para a direita. Em outras palavras, a bola vai "quicar".

A função `updateVelocities()` simplesmente atribui a verdadeira velocidade da bola às variáveis `vx` e `vy` usando a função `setVelocity()` do QCanvas. A função só precisa conter o seguinte código:

```
setVelocity(vx, vy);
```

Detecção de Colisões

A parte seguinte é a mais interessante, pois é a detecção de colisões interna do *QCanvas* em ação. Primeiro, é criada uma `QCanvasItemList` de todos os itens da tela com os quais a bola poderia colidir. Isso é obtido através da função `collisions()`, que toma um valor booleano como argumento. Se o valor booleano for ajustado como `false`, a detecção de colisões não será particularmente precisa. Por outro lado, se for ajustado para `true`, o *QCanvas* realizará uma detecção bem exata.

Agora que temos uma `QCanvasItemList` de todos os possíveis candidatos à colisão, é possível iterar entre eles usando um `QCanvasItemList::iterator` e checar cada item individualmente para uma colisão usando a função `collidesWith()`.

Listagem 5: Declarando *Brick*

```
01 #include <qcanvas.h>
02 class Brick : public QCanvasRectangle
03 {
04  public:
05   Brick(int x, int y, QCanvas *canvas);
06   ~Brick();
07 };
```

Listagem 6: *Brick* Constructor

```
01 #include "brick.h"
02 Brick::Brick(int x, int y, QCanvas *canvas)
03   : QCanvasRectangle (x, y, 30, 10, canvas)
04 {
05 }
06 Brick::~Brick()
07 {
08 }
```

Ela também usará de um argumento booleano, que opta entre detecção de colisões exata ou inexata. Isso oferecerá um ponteiro para o item com que a bola colidiu, mas sem indicação de qual era aquele objeto, de modo que a função `collide()` é chamada.

Tijolos

Embora já tenhamos uma bola, não é muito interessante tê-la sem nada com o que colidir. Agora, portanto, é hora de declarar uma nova classe em `brick.h` herdada de `QCanvasRectangle`, que serão os tijolos com que a bola colidirá e que "quebrarão" ao ser atingidos (ver **listagem 5**).

A **listagem 5** declara uma classe de nome `Brick`, que herda `QCanvasRectangle`. O construtor tem dois valores inteiros a mais, pois são os que definem as coordenadas de localização `x` e `y`. O construtor e as funções devem estar em `brick.cpp`. Essa é uma classe muito simples, porque tudo o que faz é criar um retângulo de tamanho fixo (ver **listagem 6**).

Identifique-se!

Cada um dos `QCanvasItem` tem um número de identificação único, chamado *valor rtti*. As classes derivadas de qualquer dos objetos do *QCanvas* devem retornar seus valores *rtti* através da reimplementação da função `rtti()`. Os valores *rtti* para os itens padrão do *QCanvas* são definidos no tipo enumerado `QCanvasItem::RttiValues` como segue:

```
QCanvasItem::Rtti_Item
QCanvasItem::Rtti_Ellipse
QCanvasItem::Rtti_Line
QCanvasItem::Rtti_Polygon
QCanvasItem::Rtti_PolygonalItem
QCanvasItem::Rtti_Rectangle
QCanvasItem::Rtti_Spline
QCanvasItem::Rtti_Sprite
QCanvasItem::Rtti_Text
```

Para obter o valor *rtti* de um objeto, é preciso chamar sua função `rtti()`, que retornará um inteiro. Para essa aplicação, é melhor definir o valor *rtti* para o objeto `Ball` (e quaisquer outros objetos) num arquivo de cabeçalho separado, `rtti.h`. Colocar os valores *rtti* num arquivo de cabeçalho facilita a adição de outros objetos, como a raquete. Você só precisa incluir este arquivo de cabeçalho em cada arquivo que use o valor *rtti*:

```
enum Rtti {
 Rtti_Ball = 1001,
 Rtti_Brick = 1002
};
```

Isso permitirá que a bola retorne `Rtti_Ball` como valor *rtti*. Agora podemos declarar o protótipo para a função reimplementada `rtti()` na parte *public* de `ball.h`:

```
virtual int rtti() const;
```

E em seguida implantar a função no arquivo fonte:

```
int Ball::rtti() const
{
 return Rtti_Ball;
}
```

Você deve adotar a mesma tática com a classe `Brick`, com o mesmo protótipo de `Ball` em `brick.h`. Já no arquivo `brick.cpp`, as coisas são ligeiramente diferentes:

```
int Brick::rtti() const
{
 return Rtti_Brick;
}
```

Porém, antes que isso possa funcionar, o arquivo de cabeçalho `rtti.h` precisa ser incluído nos arquivos fonte `brick.cpp` e `ball.cpp`:

```
#include "rtti.h"
```

Mais algumas colisões

Identificar objetos por seu número *rtti* funciona muito bem com detecção de colisões, de forma que podemos realizar diferentes operações sobre a bola dependendo do tipo de objeto com que ela colide. Isso é implementado na função `collide()`:

```
void Ball::collide(QCanvasItem *item)
{
 if (item->rtti() == Rtti_Brick) {
  moveBy(0, vy);
  vy = -yVelocity();
  updateVelocities();
  delete item;
 }
}
```

Primeiro, a função precisa averiguar que tipo de objeto é aquele; assim, a declaração `if` checa se o valor retornado pela função `rtti()` do objeto combina com o valor rtti atribuído a um tijolo e executa o código seguinte condicionalmente. Como a detecção de colisão retornará uma colisão antes que a bola tenha realmente colidido (de fato, retornará uma colisão se o próximo movimento, de acordo com as velocidades atuais, for causar uma colisão), é necessário primeiro mover o objeto de forma que ele realmente colida, para que não pareça que a bola quicou antes de realmente atingir o tijolo. Em seguida a velocidade vertical é revertida fazendo com que `vy` seja o negativo da atual velocidade vertical; as velocidades são atualizadas com a função `updateVelocities()`. Finalmente, o tijolo é apagado, desaparecendo como se tivesse sido "quebrado" pela colisão com a bola.

Porém, nenhum tijolo ainda foi criado no construtor da classe `view`, o que significa que nada de mais vai acontecer. Precisaremos de uma matriz dinâmica para que o programa possa criar qualquer número de tijolos na tela. Para isso, a classe `QValueVector` é usada. Primeiro, é necessário incluir o arquivo de cabeçalho para `QValueVector`, assim como para as classes `Brick` e `Ball`, em `view.h`:

```
#include <qvaluevector.h>
#include "ball.h"
#include "brick.h"
```

Depois, algumas declarações na seção `private`, válidas para todas as classes, são necessárias para a bola, os tijolos e a matriz de tijolos:

```
typedef QValueVector<Brick*> BrickArray;
BrickArray *m_bricks;
Ball *m_ball;
```

Esse código define uma matriz de objetos `Brick` chamada `BrickArray` e cria então um ponteiro de tipo `BrickArray` chamado `m_bricks`. Um ponteiro `Ball` de nome `m_ball` é criado; ele é a bola que quica no programa.

Outra função, chamada `generateTable()`, é necessária para criar os tijolos na tela. A função `generateTable()` cria

uma linha de tijolos no alto da janela do jogo. Essa função deve ser definida na seção de declaração `public`:

```
void generateTable();
```

Então o código para criar os tijolos pode ser colocado em `generateTable()` (ver **listagem 7**), que será chamado a partir do construtor de classe.

A primeira linha instancia uma nova matriz de tijolos, através da qual os tijolos serão criados. Os dois inteiros definem de onde os tijolos devem começar a ser desenhados, já que a função simplesmente criará um número arbitrário de tijolos (especificado pela variável `numOfBricks`) da esquerda para a direita, iniciando no valor de `xPos`. O laço cria todos os tijolos desejados, mudando a posição horizontal para que sejam criados em uma linha, e não um em cima do outro. A função `push_back` coloca o objeto `Brick` recém-criado no final da matriz e a última linha do laço chama a função `show()` para que o novo tijolo seja mostrado na tela.

Essa função precisa ser chamada a partir do construtor de classes, `View()`, e a bola precisa estar criada e ter as velocidades atribuídas para poder se mover:

```
generateTable();
m_ball = new Ball(m_canvas);
m_ball->move(250, 50);
m_ball->setXVelocity(1);
m_ball->setYVelocity(-2);
m_ball->show();
```

Isso gera os tijolos e em seguida cria um objeto `Ball` na posição `x` 250 e `y` 50 na tela. Ajusta então a velocidade horizontal da bola para 1 pixel por quadro e a velocidade vertical para -2 pixels por quadro, chamando a função `show()`. Para fazer com que o objeto se mova, porém, a função `setAdvancePeriod()` do `QCanvas` terá de ser chamada para acertar quanto tempo dura um quadro. Um bom ajuste é 20ms:

```
m_canvas->setAdvancePeriod(20);
```

Agora que o programa está bastante maduro, pode ser compilado. O modo mais simples de compilar o programa é com o utilitário qmake da Trolltech, que gerará automaticamente os makefiles. O formato de arquivo do qmake é muito simples; sua extensão é `.pro`. Um arquivo qmake para este jogo seria como o seguinte (`bricks.pro`):

```
SOURCES = main.cpp ball.cpp view.cpp ➘
brick.cpp
HEADERS = ball.h view.h brick.h rtti.h
CONFIG += qt warn_on release
```

Todos os arquivos `.cpp` do programa aparecem na lista `SOURCES`, todos os arquivos `.h` na lista `HEADERS` e a linha `CONFIG` é preparada como mostrado. Em seguida, tudo o que resta é rodar o `qmake` no diretório que contém todos esses arquivos. O `qmake` vai gerar o `Makefile`:

```
$ qmake
```

Agora o programa pode ser compilado usando o `GNU make`.

```
$ make
```

E executado através do nome executável, `bricks`:

```
$ ./bricks
```

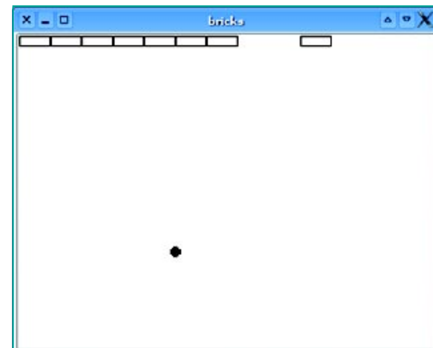

Figura 1: Temos uma bola quicando!

Se tudo houver sido feito apropriadamente, você deverá ver uma pequena bola preta quicando com 10 retângulos no alto da janela. Quando a bola colide com esses retângulos, muda de direção e o retângulo desaparece. A **figura 1** mostra uma imagem dessa cena.

Rebatendo

Uma bola que quica e esmaga tijolos não é uma coisa particularmente interessante de se ficar olhando – então é hora de escrever outra classe, que representará a raquete manejada pelo usuário para impedir a bola de cair pelo fundo da tela. A declaração de classe seguinte precisa ser escrita em `paddle.h`:

```
#include <qcanvas.h>
class Paddle : public QCanvasRectangle
{
 public:
  Paddle(QCanvas *canvas);
  ~Paddle();
  virtual int rtti() const;
};
```

O construtor de classes para isso é muito simples. Para manter a tendência, ele ficará em `paddle.cpp`. É necessário apenas configurar o pincel que pintará a raquete para fazer dela um retângulo

Listagem 7: *generateTable()*

```
01 void View::generateTable()
02 {
03  m_bricks = new BrickArray(1);
04  int xPos = 1, yPos = 1;
05  int numOfBricks = 10;
06  for (int i = 0; i < numOfBricks; ++i) {
07   xPos = (i * 30) + 1;
08   m_bricks->push_back(new Brick (xPos, yPos, m_canvas));
09   m_bricks->last()->show();
10  }
11 }
```

Listagem 8: Construtor da *raquete*

```
01 #include "paddle.h"
02 #include "rtti.h"
03 Paddle::Paddle (QCanvas *canvas)
04  : QCanvasRectangle (1, 1, 50, 5, canvas)
05 {
06  setBrush(Qt::black);
07 }
08 Paddle::~Paddle()
09 {
10 }
```

cheio em vez de um contorno e reimplementar a função `rtti()` para retornar `Rtti_Paddle` (ver **listagem 8**).

Porém, precisamos de um novo valor *rtti* para a raquete, `Rtti_Paddle`, que é retornada pela função `rtti()`:

```
int Paddle::rtti() const
{
 return Rtti_Paddle;
}
```

Assim, é necessário adicionar esse *rtti* ao tipo enumerado em `rtti.h`:

```
enum Rtti {
   Rtti_Ball = 1001,
   Rtti_Brick = 1002,
   Rtti_Paddle = 1003
};
```

Como uma nova fonte e um arquivo de cabeçalho foram adicionados ao projeto, eles precisam ser colocados nas linhas `SOURCES` e `HEADERS` no arquivo *qmake* para que o programa compile:

```
SOURCES = main.cpp ball.cpp view.cpp ➥
brick.cpp paddle.cpp
HEADERS = ball.h view.h brick.h rtti.h ➥
paddle.h
```

E o *qmake* roda novamente:

```
$ qmake
```

Agora resta apenas fazer com que a raquete responda às teclas pressionadas e adicioná-la à função `collide()` para reverter a velocidade vertical da bola quando ocorre uma colisão com a raquete. Para fazê-lo, adicionamos o seguinte a `collide()` logo após dar a última volta no parafuso da primeira declaração `if`:

```
else if (item->rtti() == Rtti_Paddle) {
  moveBy(0, vy);
  vy = -yVelocity();
  updateVelocities();
}
```

Isso simplesmente ajusta a variável `vy` para o negativo da atual velocidade `y` e em seguida chama a função `updateVelocities()` para ajustar a velocidade da bola, revertendo assim sua direção no plano vertical. Para que a raquete exista, porém, precisa ser criada na classe `View`. Assim, precisamos primeiro incluir `paddle.h` em `view.h`:

```
#include "paddle.h"
```

Depois é necessário declarar a variável na seção `private` da declaração da classe `View`:

```
Paddle *m_paddle;
```

E podemos assim criar a raquete no construtor `View`, ajustando sua posição e chamando sua função `show()`:

```
m_paddle = new Paddle(m_canvas);
m_paddle->move(50, 250);
m_paddle->show();
```

Controle pelo teclado

Para oferecer ao usuário o controle pelo teclado, a biblioteca Qt tem uma função chamada `keyPressEvent()`, à qual a Qt chama automaticamente sempre que uma tecla é pressionada quando o elemento gráfico está selecionado. Um `QKeyEvent` é passado para a função que representa a tecla que o usuário pressionou. Portanto, para realizar uma ação, como mover a raquete para a esquerda com a seta, só é preciso reimplementar a função `keyPressEvent()` e realizar as ações necessárias. Primeiro, o protótipo da função precisa ser declarado na seção *public* da classe `View`:

```
void keyPressEvent(QKeyEvent *keyEvent);
```

Listagem 9: Controle pelo Teclado

```
01 void View::keyPressEvent(QKeyEvent *keyEvent)
02 {
03  switch (keyEvent->key()) {
04   case Key_Left:
05    m_paddle->moveBy(-5, 0);
06    break;
07   case Key_Right:
08    m_paddle->moveBy(5, 0);
09    break;
10   default:
11    break;
12  }
13 }
```

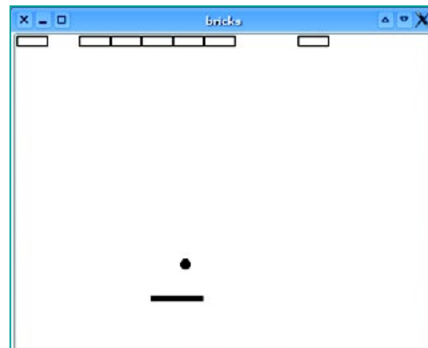

Figura 2: O jogo dos Tijolinhos com raquete, tijolinhos e bola.

E podemos usar uma declaração `switch` dentro da função para executar o código necessário, dependendo da tecla que houver sido pressionada pelo jogador (ver **listagem 9**).

Nesse caso, a declaração `switch` confere se a tecla pressionada é a seta para a direita ou a seta para a esquerda. Se for a da esquerda, ela moverá a raquete 5 pixels para a esquerda. Se for a da direita, a raquete será movida 5 pixels para a direita.

Missão Cumprida!

Assim, chegamos ao fim do jogo dos Tijolinhos. A **figura 2** mostra uma imagem do jogo terminado. Esperamos que este tutorial tenha lhe mostrado toda a flexibilidade e simplicidade do *QCanvas* para gráficos 2D. Boas idéias, e boa diversão! ■

Informações

[1] Documentação da API do QCanvas: http://doc.trolltech.com/3.3/canvas.html

[2] Trolltech: http://www.trolltech.com/

[3] O código fonte completo dos tijolinhos: http://www.gwright.org.uk/files/LinuxMag/bricks.tar.bz2

Sobre o autor

George Wright é aluno do primeiro ano da Harrow School, na Inglaterra; estuda Matemática, Matemática Avançada, Estatísticas, Física, Química, Computação e Eletrônica. Espera estudar Ciências da Computação na universidade e está envolvido no projeto KDE desde 2001. Mantém um agudo interesse no Linux, especialmente como estação de trabalho e aplicações embarcadas. Ele pode ser contactado no e-mail gwright@kde.org.