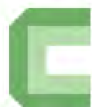
Jogos Android

Criando um game do zero usando classes nativas



FELIPE TORRES

© Casa do Código
Todos os direitos reservados e protegidos pela Lei nº9.610, de 10/02/1998.
Nenhuma parte deste livro poderá ser reproduzida, nem transmitida, sem autorização prévia por escrito da editora, sejam quais forem os meios: fotográficos, eletrônicos, mecânicos, gravação ou quaisquer outros.

Casa do Código
Livros para o programador
Rua Vergueiro, 3185 - 8º andar
04101-300 – Vila Mariana – São Paulo – SP – Brasil

Sumário

1 **Sobre o autor** 1

2 **Por que um jogo mobile?** 3

3 **Começando o Jumper** 5

3.1 Criando o projeto e a tela principal 5
3.2 Criando o loop principal do Jumper 11
3.3 Como desenhar elementos no SurfaceView? 13
3.4 Criando nosso primeiro elemento do Jumper: a classe Passaro 15
3.5 Criando o comportamento padrão do pássaro: o método cai 20

4 **Colocando uma imagem de fundo** 21

4.1 Implementando o controle do jogador: o pulo do pássaro . . 29

5 **Criando o cano inferior** 31

5.1 Movimentando o cano 37

6 **Criando vários canos** 39

6.1 Criando os limites para pulo: o chão e teto 44

7 **Criando os canos superiores** 47

7.1 Criando infinitos canos 51
7.2 Descartando canos anteriores 53

8 **Criando a pontuação do jogo** 57

9 Tratando colisões **69**

9.1 Criando a tela de game over 72

9.2 Centralizando um texto horizontalmente na tela 75

10 Aprimorando o *layout* do jogo **79**

10.1 Substituindo os retângulos por bitmaps 82

11 Adicionando som ao jogo **87**

Capítulo 1

Sobre o autor

Desde o momento em que escrevi minha primeira linha de código em um projetinho Android de estudos me senti completamente empolgado com as diversas possibilidades de aplicações que poderia desenvolver e resolvi mergulhar nesse mundo e aprender tudo o que pudesse sobre esse universo.

Naquele momento, cursava meu último ano da graduação no IME-USP e estava pesquisando o que faria no meu projeto de conclusão de curso. Não tive dúvidas: queria fazer algo com Android, pois teria uma chance para me dedicar a aprendê-lo a fundo. No fim das contas, acabamos fazendo em dupla um otimizador de rotas, onde tive meu primeiro contato com algumas tecnologias do Android, como o atual *Google Cloud Messaging*.

Ao término da graduação, senti a necessidade de participar de algum projeto *open source*, porém queria algo no qual eu realmente pudesse fazer diferença e que fosse divertido também, então comecei a pesquisar como deveria

fazer para participar do código-fonte do Android. Algumas noites mal dormidas (e gigabytes de download) depois e já tinha tudo configurado para o meu primeiro *commit* no *Android Open Source Project*.

Paralelamente a isso, lia alguns textos sobre desenvolvimento de jogos e resolvi entrar nessa área, porém muito material que via era focado no uso de algum *framework*, de modo que até mesmo aquele jogo mais simples precisava de um caminhão de API para ser desenvolvido. Neste momento, resolvi tentar fazer jogos simples graficamente, sem o uso de frameworks, e tive bastante êxito, rendendo um convite para palestrar no Conexão Java e um jogo publicado no Google Play com mais de 100 mil downloads.

Atualmente sou desenvolvedor e instrutor na Caelum apaixonado pelo mundo *mobile*, dedico um tempo ajudando a desenvolver o código-fonte do Android e muito do que aprendi durante esses anos de estudo está compartilhado neste livro.

Capítulo 2

Por que um jogo mobile?

Somente em um ano, dentro da indústria de jogos mobiles, temos faturamento em dólares como:

- O jogo *Clash of Clans* recebeu 800 milhões.
- A saga *Candy Crush* faturou 300 milhões.
- A série *Angry Birds* ganhou 195 milhões.

A venda de *smartphones* e *tablets* vem aumentando cada vez mais, tornando seus usuários um grande público não só para aplicativos mas também para jogos. Muitas das grandes desenvolvedoras de games como EA, Gameloft e Ubisoft já perceberam isso e contam com divisões inteiras destinadas somente ao desenvolvimento de games para plataformas móveis.

Não há como ignorar o tamanho desse mercado. Disponibilizar, ou não, uma versão *mobile* de um jogo é a diferença entre estar neste mercado bilionário ou ficar de fora.

Como este livro está organizado?

Este livro está organizado em capítulos focados na programação dos diversos elementos do nosso jogo e, principalmente, na teoria por trás do código. Dessa forma, em vez de simplesmente replicarmos o código apresentado, entenderemos o que se passa e teremos condições de criarmos nossos próprios jogos!

Como será o nosso jogo?

Um jogo que se destacou bastante e ganhou notoriedade na mídia foi o Flappy Bird, criado em apenas três dias pelo vietnamita Dong Nguyen, que chegou a faturar 120 mil reais por dia com anúncios. Como o Flappy Bird apresenta os principais elementos de um jogo (e é bem divertido), vamos criar a nossa versão desse game: o Jumper!

Agora que temos uma ideia do jogo que faremos, uma dúvida que aparece é: o que vamos usar para criar nosso game? Uma rápida busca na internet pelo tema "ferramentas para jogos Android" pode revelar inúmeras alternativas e nos deixar confusos: será que devemos usar libGDX ou Unity com Chipmunk? Será que o Cocos2D não seria melhor?

A pergunta que devemos fazer é: sempre teremos que usar algum framework para desenvolvimento de jogos? Muitas vezes, não.

Os frameworks podem nos ajudar em vários aspectos do desenvolvimento de um jogo, porém, para muitos jogos eles não são necessários. No nosso Jumper, não utilizaremos nenhuma ferramenta específica para jogos, apenas as funcionalidades que a API do Android nos oferece! Dessa forma, podemos aprender os conceitos por trás de um jogo e entender as vantagens e desvantagens de utilizar um framework.

Capítulo 3

Começando o Jumper

3.1 Criando o projeto e a tela principal

Como o Jumper é um jogo para Android, vamos criar um novo projeto no Android Studio. Para isso, vamos a `File -> New Project` e preencheremos os campos *Application Name* e *Company Domain*:

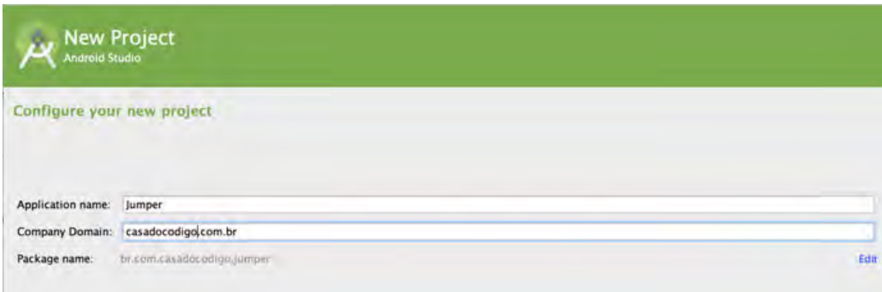


Fig. 3.1: Tela de criação do projeto no Android Studio.

No nosso jogo, teremos uma *View* com um pássaro, canos e um *background* com nuvens. Como o Android não a possui por padrão, vamos precisar criar nossa própria *View* customizada.

Para implementá-la, podemos criar uma classe filha de `View` ou de `SurfaceView`. A diferença entre elas é que, enquanto a `View` faz todos os desenhos na UIThread, a `SurfaceView` disponibiliza uma *thread* para que possamos fazer operações mais pesadas em segundo plano sem comprometer a usabilidade da aplicação. Como nosso jogo terá elementos dispostos na tela em posições calculadas, teremos que utilizar uma `SurfaceView`.

Para criar nosso próprio componente de *View*, vamos criar uma classe chamada `Game` no pacote `br.com.casadocodigo.jumper.engine`, herdar de `SurfaceView` e implementar seu construtor:

```
public class Game extends SurfaceView {

    public Game(Context context) {
        super(context);
    }
}
```

Agora que temos nossa *View*, precisaremos vinculá-la a uma *Activity*. Quando criamos nosso projeto, o próprio assistente já criou uma *Activity* chamada `MainActivity` e um *layout* chamado `activity::main.xml`. Vamos alterar esse layout para conter apenas um "espaço vazio" (um `FrameLayout` com o id `container`), no qual colocaremos nossa *View* customizada:

```
<FrameLayout xmlns:android=
    "http://schemas.android.com/apk/res/android"
    android:id="@+id/container"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

Para adicionar o `SurfaceView` ao `FrameLayout`, precisaremos instanciar nossa classe `Game` e chamar o método `addView`. Isso será feito no `onCreate` da nossa `MainActivity`:

```
protected void onCreate(Bundle savedInstance) {
    super.onCreate(savedInstance);
    setContentView(R.layout.activity_main);

    FrameLayout container =
        (FrameLayout) findViewById(R.id.container);

    Game game = new Game(this);
    container.addView(game);
}
```

Como nossa `SurfaceView` não faz nada, ao rodar o jogo será exibida uma tela preta e a barra superior, contendo o nome da nossa aplicação:

Fig. 3.2: Nossa SurfaceView até o momento.

Uma decisão importante que temos que tomar ao criar um jogo diz respeito à orientação da tela: devemos deixá-la na posição vertical (*portrait*) ou

horizontal (*landscape*)? Muitos jogos precisam de espaço horizontal para exibir elementos gráficos como placar, vidas, inimigos. Já outros não têm essa necessidade e optam pela orientação vertical. No caso do Jumper, vamos fixar a orientação na vertical (*portrait*).

No `AndroidManifest.xml` temos o registro de todas as activities da nossa aplicação. No nosso caso, como temos apenas a `MainActivity`, podemos vê-la registrada na tag `<activity>`. Como queremos fixar sua orientação, podemos utilizar a propriedade `android:screenOrientation` e defini-la como `portrait`:

```
<activity
    android:name="br.com.casadocodigo.jumper.MainActivity"
    android:label="@string/app_name"
    android:screenOrientation="portrait">
```

Seria interessante removermos a barra superior da nossa aplicação para que nosso jogo possa ser exibido em *fullscreen*. Por padrão, na tag `<application>`, no `AndroidManifest.xml`, o Android define um ícone, o nome da aplicação e um tema que contém a barra superior:

```
<application
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:theme="@style/AppTheme">
```

Para deixar nosso jogo em *fullscreen* teremos que alterar esse tema para:

```
android:theme="@android:style/Theme.NoTitleBar.Fullscreen"
```

Com isso, ele será exibido em *fullscreen* e em modo *portrait*!

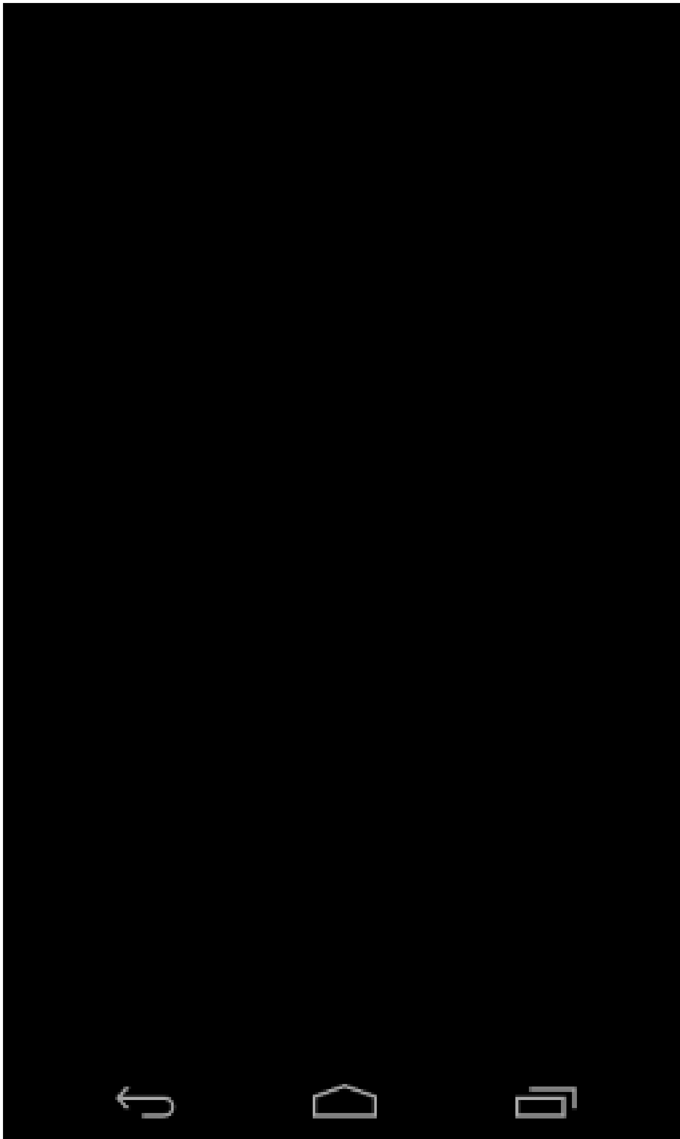


Fig. 3.3: SurfaceView em tela cheia.

Até o momento, não temos um jogo dos mais desafiadores. A seguir, vamos criar os elementos do Jumper para deixá-lo mais emocionante!

3.2 Criando o loop principal do Jumper

Um jogo é composto por vários quadros (ou *frames*), sendo que cada frame representa uma configuração dos seus elementos naquele momento específico. Porém, para dar a noção de movimento e animação, em um segundo muitos desses frames devem ser exibidos. A maioria dos jogos trabalha com uma quantidade de 60 FPS (frames por segundo).

Para criarmos esses frames, precisaremos de uma estrutura que fique rodando enquanto o jogo estiver sendo utilizado. Essa estrutura é chamada de *loop* principal do jogo. Nesse *loop* faremos os desenhos dos elementos do nosso jogo, atualizaremos a pontuação do jogador, verificaremos se houve colisão, enfim, gerenciaremos todos os aspectos dinâmicos do Jumper.

Como nossa classe `Game` é um `SurfaceView`, podemos utilizar uma *thread* separada para o nosso loop principal. Na classe `Game` vamos implementar a interface `Runnable`:

```
public class Game extends SurfaceView implements Runnable {

    @Override
    public void run(){
        // Aqui vamos implementar o loop principal do nosso jogo!
    }
}
```

Enquanto o usuário não sair do jogo, vamos deixar o loop rodando. Para isso, vamos criar uma variável *boolean* que indica se o jogo deve rodar ou não:

```
public class Game extends SurfaceView implements Runnable {

    private boolean isRunning = true;

    @Override
    public void run() {
        while(isRunning) {
            //Neste loop vamos gerenciar os elementos do Jumper.
        }
    }
}
```

O que acontecerá com o nosso loop quando o jogador deixar a aplicação?

Até o momento, quando o jogador sair do Jumper, o jogo continuará rodando pois a thread não foi parada! Isso pode consumir recursos do aparelho desnecessariamente.

Precisamos controlar o início e pausa do loop do jogo. Porém, como podemos saber que o jogador saiu da nossa aplicação? Utilizando o ciclo de vida da `MainActivity`!

No `onPause`, vamos pausar a thread do jogo:

```
public class MainActivity extends Activity {
    @Override
    protected void onPause() {
        super.onPause();
        game.cancela();
    }
}
```

Na nossa classe `Game` ainda não temos o método `cancela`. Vamos implementá-lo agora:

```
public class Game extends SurfaceView implements Runnable {

    public void cancela() {
        this.isRunning = false;
    }
}
```

Agora podemos pausar nosso loop ao deixar a aplicação, mas como faremos para rodá-lo novamente? Como podemos saber que o jogador voltou ao nosso jogo? No `onResume` da `MainActivity`!

Vamos iniciar o loop principal no Jumper no `onResume` da `MainActivity`:

```
public class MainActivity extends Activity {
    @Override
    protected void onResume() {
        super.onResume();
        game.inicia();
```

```
        new Thread(game).start();
    }
}
```

O método `inicia` fica assim:

```
public class Game extends SurfaceView implements Runnable {

    public void inicia() {
        this.isRunning = true;
    }
}
```

Com o loop principal funcionando, podemos criar os elementos do jogo!

3.3 Como desenhar elementos no SurfaceView?

Para desenhar algo em um `SurfaceView` precisamos ter acesso ao objeto responsável por renderizar elementos na tela: o *canvas*. Com esse objeto, podemos desenhar formas geométricas ou até mesmo *bitmaps*.

Desenhar elementos em um canvas consome um tempo de alguns milissegundos que já é o suficiente para o nosso olho perceber que um quadro está sendo substituído por outro. Enquanto o canvas não está pronto, a tela fica preta, por um período muito curto, criando o efeito de tela "piscante" chamado *flickering*.

Para não termos que nos preocupar com isso, o Android já tem uma forma de evitar esse *flickering*: enquanto um quadro está sendo exibido para o jogador, um novo está sendo desenhado em segundo plano. Quando o novo quadro está pronto, o Android só troca o velho pelo novo. A única coisa que temos que fazer é pegar o canvas para desenhar e, quando tudo estiver pronto, exibi-lo.

Para termos acesso ao canvas, precisamos de um objeto que permita a edição de cada pixel da nossa tela. Esse objeto é o `SurfaceHolder`. Como estamos em um `SurfaceView`, para obtermos acesso ao `SurfaceHolder`, basta chamar o método `getHolder`:

```
public class Game extends SurfaceView implements Runnable{
```

```
    private final SurfaceHolder holder = getHolder();
}
```

Por meio desse `SurfaceHolder` teremos acesso ao canvas! Para isso, basta chamar o método `lockCanvas` dentro do loop principal no nosso jogo:

```
public class Game extends SurfaceView implements Runnable{

    private final SurfaceHolder holder = getHolder();

    @Override
    public void run() {
        while(isRunning) {
            Canvas canvas = holder.lockCanvas();
        }
    }
}
```

Além disso, quando terminarmos de desenhar os elementos no nosso canvas, vamos liberá-lo para a tela do jogador por meio do método `unlockCanvasAndPost`:

```
public class Game extends SurfaceView implements Runnable{

    private final SurfaceHolder holder = getHolder();

    @Override
    public void run() {
        while(isRunning) {
            Canvas canvas = holder.lockCanvas();

            //Aqui vamos desenhar os elementos do jogo!

            holder.unlockCanvasAndPost(canvas);
        }
    }
}
```

Ao chamar o `getHolder`, precisamos garantir que temos acesso ao `SurfaceHolder` **antes** de começarmos a desenhar. Do contrário, tomare-

mos um *NullPointerException*. A forma mais simples para ter certeza de que só vamos desenhar quando nosso `holder` estiver pronto para isso é uma verificação em nosso loop:

```
public class Game extends SurfaceView implements Runnable{

    @Override
    public void run() {
        while(isRunning) {
            if(!holder.getSurface().isValid()) continue;

                //Restante do código...

        }
    }
}
```

3.4 Criando nosso primeiro elemento do Jumper: a classe Passaro

Vamos criar o principal componente do nosso jogo: o pássaro. Para isso, criaremos uma classe `Passaro`. Inicialmente, vamos utilizar alguma forma simples para representar nosso pássaro: um círculo. Além disso, ele terá a cor vermelha.

Para desenhar um círculo vermelho na tela, utilizamos um método da classe `Canvas` chamado `drawCircle`:

```
public class Passaro {

private static final int X = 100;
    private static final int RAIO = 50;

    private int altura;

    public Passaro() {
        this.altura = 100;
    }
```

```
    public void desenhaNo(Canvas canvas){
        canvas.drawCircle(X, altura, RAIO, ??);
    }
}
```

Para criarmos a cor vermelha, vamos utilizar a classe `Paint`. Como também serão usadas outras cores ao longo do jogo, vamos centralizá-las em uma classe chamada `Cores`. Nesta classe, teremos o método `getCorDoPassaro` que retornará um `Paint` representando a cor vermelha:

```
public class Cores {

    public static Paint getCorDoPassaro() {
        Paint vermelho = new Paint();
        vermelho.setColor(0xFFFF0000);
        return vermelho;
    }
}
```

Veja que a cor vermelha (0xFF0000) é representada como 0x**FF**FF0000. Esses dois primeiros valores representam a quantidade de **opacidade** da cor, sendo que FF significa o máximo de opacidade (ou o mínimo de transparência) possível. Esse formato que permite controlar a transparência da cor é chamado de **ARGB**.

Com o método `getCorDoPassaro` pronto, basta chamá-lo na classe `Passaro`:

```
public class Passaro {

    private static final Paint vermelho =
        Cores.getCorDoPassaro();

    private static final int X = 100;
    private static final int RAIO = 50;

    private int altura;

    public Passaro() {
        this.altura = 100;
```

```
    }

    public void desenhaNo(Canvas canvas) {
        canvas.drawCircle(X, altura, RAIO, vermelho);
    }
}
```

Tendo o método desenhaNo, vamos chamá-lo no loop principal da nossa classe Game:

```
public class Game extends SurfaceView implements Runnable{

private Passaro passaro;

    @Override
    public void run() {
        while(isRunning) {
            if(!holder.getSurface().isValid()) continue;

            canvas = holder.lockCanvas();

            passaro.desenhaNo(canvas);

            holder.unlockCanvasAndPost(canvas);
        }

    }
}
```

Perceba que em nenhum momento instanciamos a classe Passaro. Onde podemos fazer isso?

Temos que tomar bastante cuidado onde daremos new nos nossos elementos. Será que precisaremos de um novo Passaro a cada loop? Não. Dessa forma, sempre manipularemos a **mesma** instância de Passaro! Caso instanciemos o Passaro dentro do loop, criaremos muitos objetos na memória do aparelho, causando uma lentidão no nosso jogo!

Sendo assim, vamos criar um método inicializaElementos e chamá-lo no construtor da nossa classe Game:

```
public class Game extends SurfaceView implements Runnable{

    private Passaro passaro;

    public Game(Context context) {
        super(context);
        inicializaElementos();
    }

    private void inicializaElementos() {
        this.passaro = new Passaro();
    }
}
```

Ao rodar o Jumper, teremos a seguinte tela:

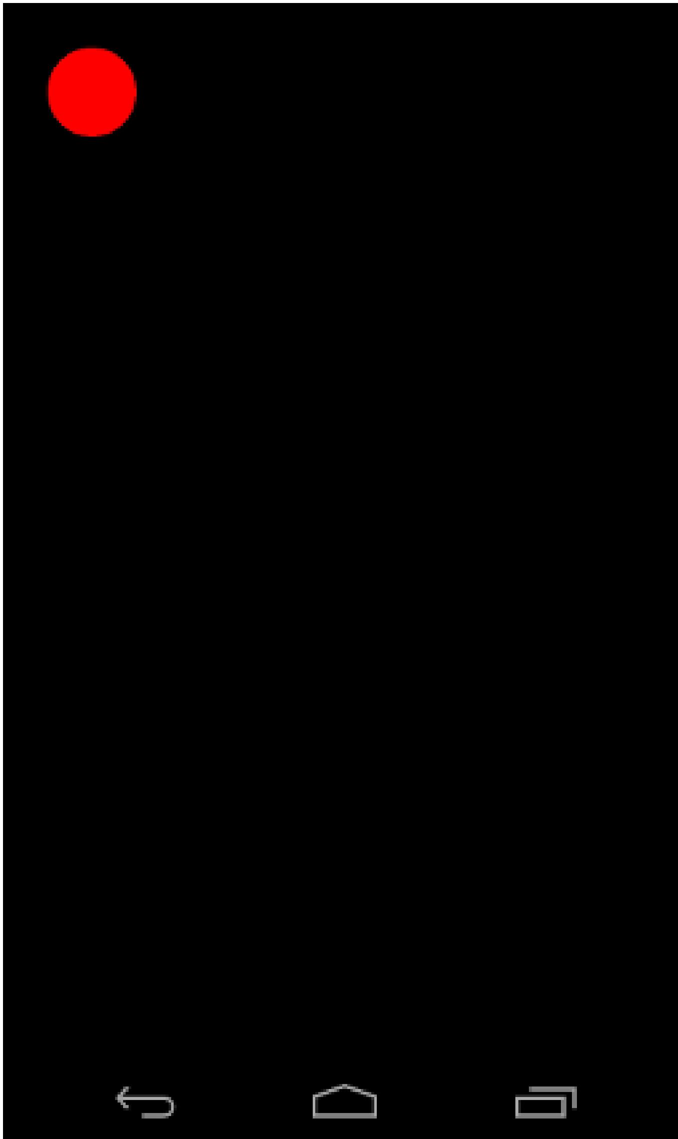


Fig. 3.4: Nosso pássaro na tela do jogo.

Temos nosso primeiro elemento sendo desenhado na tela, porém ele está estático. No decorrer do jogo, o que acontece com o pássaro? Ele cai. Vamos

implementar isso!

3.5 Criando o comportamento padrão do pássaro: o método cai

No loop principal, além de desenhar o `Passaro`, vamos fazê-lo cair por meio do método `cai`:

```
public class Game extends SurfaceView implements Runnable{

    @Override
    public void run() {
        while(isRunning) {
            if(!holder.getSurface().isValid()) continue;

            canvas = holder.lockCanvas();

            passaro.desenhaNo(canvas);
            passaro.cai();

            holder.unlockCanvasAndPost(canvas);
        }
    }
}
```

Faremos o pássaro cair 5 pixels a cada chamada ao método `cai`. Sendo assim, nossa implementação do método `cai` deve ser a seguinte:

```
public class Passaro {

    private int altura;

    public void cai() {
        this.altura += 5;
    }
}
```

Isso é o suficiente para derrubarmos o pássaro a cada iteração do nosso loop principal.

Capítulo 4

Colocando uma imagem de fundo

Temos nosso pássaro desenhado na tela. Porém, ao implementarmos o método `cai`, criamos um rastro indesejado. Como poderemos resolver isso?

Vimos que a classe `SurfaceView` (mãe da nossa classe `Game`) é a responsável por desenhar os elementos do nosso jogo na tela e que, ao fazer o desenho, essa classe não manipula os *pixels* que estão sendo visualizados pelo jogador; o desenho é feito em segundo plano e então exibido ao jogador.

Essa estrutura, que foi tão útil para evitar bugs de atualização da tela (*flickering*), também é responsável por deixar o rastro dos elementos!

Quando o Android troca o *frame* que está em primeiro plano para aquele que está em segundo plano (e vice-versa), ele **não** limpa o conteúdo anteriormente desenhado! É exatamente esse conteúdo "desatualizado" que vemos.

Perceba que o rastro da bola é formado pelas suas posições anteriormente desenhadas.

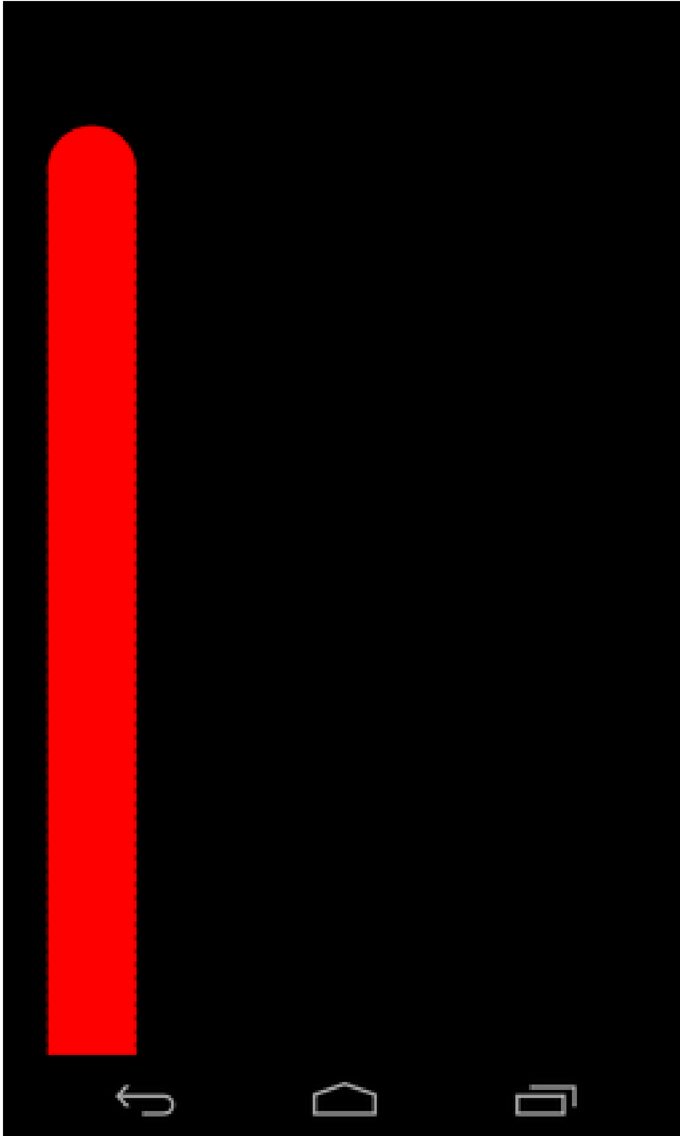


Fig. 4.1: Rastro da bolinha ao cair.

Vamos corrigir esse problema colocando uma imagem que ocupe a tela inteira. Dessa forma, vamos ocultar as imagens "desatualizadas" e deixar apenas a figura atual da bola. Será o *background* do nosso jogo!

> **Arquivos usados no jogo**
>
> Os arquivos que usaremos ao longo desse livro podem ser baixados em https://github.com/felipetorres/jumper-arquivos.

Como o background do Jumper será uma imagem, vamos precisar de um *drawable* para usar no nosso código. Onde podemos colocar essa imagem, já que existem várias pastas *drawable*?

Muitas vezes é importante ter controle sobre como as imagens serão redimensionadas pelo Android. Cada *qualifier* presente nas pastas *drawable* (mdpi, hdpi, xhdpi etc.) representa um tipo de densidade de tela (*medium*, *high*, *extra high*) que necessita de uma resolução e tamanho específicos. Podemos criar um tamanho do nosso background para cada tipo de tela ou redimensionarmos via código, diminuindo o tamanho da app. É o que faremos.

Para isso, precisaremos dizer ao Android **não** redimensionar automaticamente nosso background. Nesses casos, devemos colocar nossa imagem na pasta `drawable-**nodpi**`. Não se preocupe caso esta pasta não exista; basta criá-la no diretório `res`.

Na classe `Game` podemos pegar a imagem utilizando o método `decodeResource` da classe `BitmapFactory`:

```
this.background = BitmapFactory.decodeResource(getResources(),
R.drawable.background);
```

Agora, podemos desenhar esse `background` no *loop* principal do nosso jogo utilizando o método `drawBitmap` do `canvas`. Perceba que os desenhos são feitos em camadas: os desenhos que são feitos primeiro ficam abaixo dos seguintes, logo nosso background tem que ser a primeira coisa a ser desenhada na tela:

```
@Override
public void run() {
```

```
    while(isRunning) {
        if(!holder.getSurface().isValid()) continue;
        canvas = holder.lockCanvas();

        canvas.drawBitmap(background, 0, 0, null);
        //código de desenho do pássaro...

        holder.unlockCanvasAndPost(canvas);
    }
}
```

Fig. 4.2: Background adicionado ao jogo.

Precisamos redimensionar nosso background para que ele ocupe todo o espaço vertical da tela. Porém, para definirmos a altura dessa imagem, deve-

mos saber a altura da tela; vamos utilizar a classe `WindowManager` para nos fornecer essas informações.

```
WindowManager wm = (WindowManager) context.getSystemService(
                                        Context.WINDOW_SERVICE);
```

Com essa classe, podemos obter as dimensões da tela por meio do método `getDefaultDisplay` e do método `getMetrics`:

```
WindowManager wm = (WindowManager) context.getSystemService(
                                     Context.WINDOW_SERVICE);
Display display = wm.getDefaultDisplay();
metrics = new DisplayMetrics();
display.getMetrics(metrics);
```

Ao executar esse código, as dimensões da tela estarão no objeto `DisplayMetrics`. Para recuperar a altura da tela, basta chamar o método `heightPixels` na variável `metrics`:

```
int alturaDaTela = metrics.heightPixels;
```

Vamos centralizar essas operações em uma classe chamada `Tela` com um método `getAltura`:

```
public class Tela {

private DisplayMetrics metrics;

public Tela(Context context) {
    WindowManager wm = (WindowManager) context.getSystemService(
                                        Context.WINDOW_SERVICE);
    Display display = wm.getDefaultDisplay();
    metrics = new DisplayMetrics();
    display.getMetrics(metrics);
}

public int getAltura() {
    return metrics.heightPixels;
}
}
```

Com essa classe, podemos redimensionar nosso background da classe `Game` chamando o método `createScaledBitmap`:

```
Bitmap back = BitmapFactory.decodeResource(getResources(),
                R.drawable.background);
this.background = Bitmap.createScaledBitmap(back,
                   back.getWidth(), tela.getAltura(), false);
```

Como queremos redimensionar a imagem apenas na vertical, vamos manter a largura original do background. Para capturar essa largura, basta chamar o método `getWidth`.

Isso é o suficiente para nossa imagem de fundo ficar com a seguinte aparência:

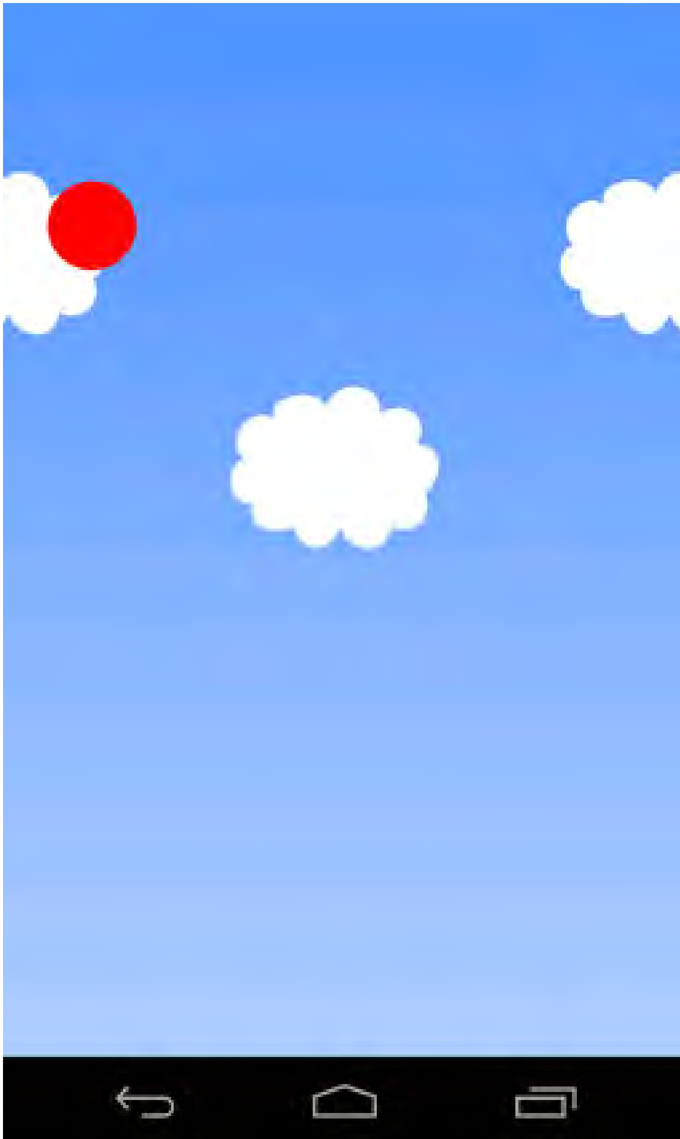


Fig. 4.3: Background ocupando a tela inteira.

4.1 Implementando o controle do jogador: o pulo do pássaro

Neste momento, o elemento `Passaro` não deixa na tela mais nenhum rastro visível ao jogador! Podemos nos dedicar ao seu movimento. O único controle disponível ao jogador do Jumper será o *toque na tela* (não teremos nenhum outro botão) que fará o pássaro pular.

Para implementar o pulo do pássaro, basta deslocá-lo uma quantidade de *pixels* para cima. Vamos fazer isso criando o método `pula` na classe `Passaro`:

```
public class Passaro {

    //...

    public void pula() {
        this.altura -= 150;
    }
}
```

Note que o método pula **diminui** a altura em 150 pixels. Como o ponto (0,0) representa o **canto superior esquerdo** da tela, ao decrementar a altura, estamos fazendo o pássaro ser deslocado para cima!

Em que momento vamos chamar esse método `pula`? Quando o jogador **tocar** a *View* do jogo!

Para isso, temos que implementar o método `onTouch` na classe que representa a *View* do nosso jogo: a classe `Game`. Para termos acesso ao método `onTouch`, vamos dizer que essa classe deve implementar a interface `onTouchListener`:

```
public class Game extends SurfaceView implements Runnable,
OnTouchListener {

    @Override
    public boolean onTouch(View v, MotionEvent event) {
        return false;
    }
}
```

Neste método, basta chamar o `pula` do objeto `passaro`:

```
public class Game extends SurfaceView implements Runnable,
OnTouchListener {

    @Override
    public boolean onTouch(View v, MotionEvent event) {
        passaro.pula();
        return false;
    }
}
```

Implementamos o *listener*, porém isso não é suficiente para que o método `onTouch` seja chamado sempre que o jogador tocar na nossa *View*, pois em nenhum momento dissemos qual *View* deverá chamar esse `onTouch`. Vamos fazer isso com o método `setOnTouchListener`.

Como nossa classe `Game` **é uma *View***, podemos chamar esse método diretamente no seu construtor:

```
public class Game extends SurfaceView implements Runnable,
OnTouchListener{

    public Game(Context context) {
        super(context);
        //...
        setOnTouchListener(this);
    }
}
```

Com isso, finalizamos o controle do jogador! Contudo, ainda temos alguns outros elementos neste jogo.

Capítulo 5

Criando o cano inferior

Além do pássaro, o Jumper também possui outro elemento a ser desenhado na tela: os canos! Como aproximamos inicialmente o pássaro para uma bolinha, vamos aproximar os canos para retângulos.

Assim como feito com o pássaro, nosso objetivo será desenhar os canos no *loop* principal da classe `Game`, algo como:

```
public class Game extends SurfaceView implements Runnable,
OnTouchListener{

    @Override
    public void run() {
        while(isRunning) {

            //lockCanvas e desenhos anteriores...
            cano.desenhaNo(canvas);
```

```
            //unlock...
        }
    }
}
```

Vamos criar uma classe `Cano` contendo o método `desenhaNo`. Além disso, vamos definir o cano com 250 *pixels* de altura e 100 *pixels* de largura:

```
public class Cano {

    private static final int TAMANHO_DO_CANO = 250;
    private static final int LARGURA_DO_CANO = 100;

    public void desenhaNo(Canvas canvas) {
        //Como será o desenho do cano?
    }
}
```

Primeiramente, vamos criar o cano inferior com o método `desenhaCanoInferiorNo`. Para desenhar um retângulo, chamaremos o método `drawRect` da classe `Canvas`:

```
public class Cano {

    public void desenhaNo(Canvas canvas) {
        desenhaCanoInferiorNo(canvas);
    }

    private void desenhaCanoInferiorNo(Canvas canvas) {
        canvas.drawRect(??, ??, ??, ??, ??);
    }

}
```

Para desenhar um retângulo com o método `drawRect`, precisamos passar as coordenadas (x,y) de dois pontos: o canto superior esquerdo e o canto inferior direito.

No entanto, quais serão esses pontos?

O canto inferior direito do cano deve estar na **base da tela**, ou seja, deve ter a mesma altura da tela (lembre-se de que o ponto (0,0) está no topo da tela). Como já conseguimos pegar a altura da tela, podemos passá-la para o `drawRect`:

```
canvas.drawRect(??, ??, ??, tela.getAltura(), ??);
```

O cano deve ter 250 *pixels* de altura, logo deve acabar 250 *pixels* **antes** da base da tela. Basta subtrair 250 *pixels* da altura da tela e teremos o segundo parâmetro do cano:

```
int alturaDoCanoInferior =
    tela.getAltura() - TAMANHO_DO_CANO;

public class Cano {

    private static final int TAMANHO_DO_CANO = 250;
    private int alturaDoCanoInferior;
    private Tela tela;

    public Cano(Tela tela) {
        this.tela = tela;
        this.alturaDoCanoInferior =
            tela.getAltura() - TAMANHO_DO_CANO;
    }

    private void desenhaCanoInferiorNo(Canvas canvas) {
        canvas.drawRect(??, alturaDoCanoInferior, ??,
        tela.getAltura(), ??);
    }
}
```

Até esse momento, limitamos a altura do cano. Porém, como limitaremos a sua largura? O primeiro e terceiro argumentos do `drawRect` se referem justamente a isso!

Temos que desenhar esse cano de tal forma que sua posição horizontal possa variar, pois afinal de contas, ele deverá se mover para a direção do pássaro. Vamos receber essa posição no construtor do `Cano`:

```
public class Cano {

    private static final int TAMANHO_DO_CANO = 250;
    private int alturaDoCanoInferior;
    private Tela tela;
    private int posicao;

    public Cano(Tela tela, int posicao) {
        this.tela = tela;
        this.posicao = posicao;
        this.alturaDoCanoInferior =
            tela.getAltura() - TAMANHO_DO_CANO;
    }

    private void desenhaCanoInferiorNo(Canvas canvas) {
        canvas.drawRect(posicao, alturaDoCanoInferior, ??,
        tela.getAltura(), ??);
    }
}
```

Agora temos que a lateral esquerda do `Cano` começará em `posicao`, só precisamos definir onde ficará a lateral direita desse `Cano`. Inicialmente, definimos que ele deveria ter 100 *pixels* de largura, o que significa que a lateral direita deverá ficar a 100 *pixels* da `posicao`!

```
public class Cano {

    private static final int TAMANHO_DO_CANO = 250;
    private static final int LARGURA_DO_CANO = 100;

    public Cano(Tela tela, int posicao) {
        this.posicao = posicao;
        this.alturaDoCanoInferior =
            tela.getAltura() - TAMANHO_DO_CANO;
    }

    private void desenhaCanoInferiorNo(Canvas canvas) {
        canvas.drawRect(posicao, alturaDoCanoInferior,
        posicao + LARGURA_DO_CANO, tela.getAltura(), ??);
```

```
    }
}
```

Com isso, só precisamos definir a cor que esse elemento terá.

Todo elemento desenhado no `canvas` pode receber uma cor por meio de um `Paint`. Da mesma forma que fizemos com o `Passaro`, vamos criar um método `getCorDoCano` na classe `Cores`:

```
public class Cores {

    public static Paint getCorDoCano() {
        Paint verde = new Paint();
        verde.setColor(0xFF00FF00);
        return verde;
    }
}
```

Definimos a cor verde para o `Cano` (00FF00), porém lembre-se de que a cor deve estar no formato **ARGB**, que permite o uso de transparência. Neste formato, a cor verde é **FF**00FF00.

Com a cor do `Cano` criada, podemos passá-la para o último argumento do seu `drawRect`:

```
public class Cano {

    private final Paint verde = Cores.getCorDoCano();

    //Códigos anteriores...

    private void desenhaCanoInferiorNo(Canvas canvas) {
        canvas.drawRect(..., ..., ..., ..., verde);
    }
}
```

Só precisamos instanciar esse `Cano` na nossa classe `Game` dizendo qual será sua posição inicial:

```
this.cano = new Cano(tela, 200);
```

Com isso, finalizamos a criação do cano inferior do Jumper!

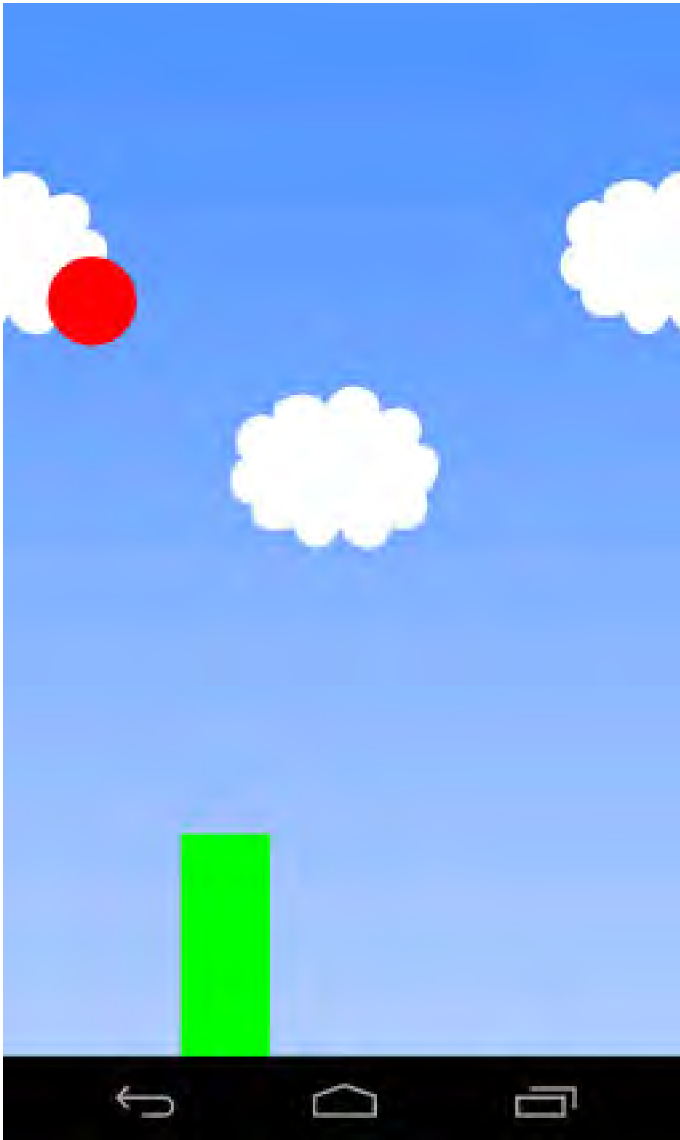


Fig. 5.1: Jumper com um cano inferior.

5.1 Movimentando o cano

No Jumper, os canos devem se mover na direção do pássaro. Até o momento, nosso inimigo está estático na tela. Vamos implementar seu movimento para a esquerda.

Utilizando a mesma ideia da classe `Passaro`, vamos deslocar esse `Cano` alguns *pixels* para a esquerda a cada iteração do nosso *loop* principal. Para isso, precisaremos de um método `move`:

```
public class Game extends SurfaceView implements Runnable,
OnTouchListener{

@Override
    public void run() {
        while(isRunning) {
            //lockCanvas...
            //Desenho do pássaro...

            cano.desenhaNo(canvas);
            cano.move();

            //unlock...
        }
    }
}
```

Como deixamos a `posicao` do `Cano` variável e a consideramos para fazer a conta da largura, podemos simplesmente alterar essa `posicao` e o `Cano` se moverá corretamente mantendo sua largura. O método `move` apenas alterará a variável `posicao`:

```
public class Cano {
    //...

    public void move() {
        posicao -= 5;
    }
}
```

Isso é o suficiente para movermos o `Cano` para a esquerda!

Capítulo 6

Criando vários canos

Ainda temos somente um cano no nosso jogo, além disso ele sai muito rápido da tela. Está na hora de colocar um pouco mais de emoção com mais canos. Podemos alterar nossa criação de um cano para criar diversos canos, cinco por exemplo:

```
this.cano1 = new Cano(tela, 200);
this.cano2 = new Cano(tela, 450);
this.cano3 = new Cano(tela, 700);
this.cano4 = new Cano(tela, 950);
this.cano5 = new Cano(tela, 1200);
```

O código parece feio, uma vez que o estamos copiando e colando por todo lado. Vamos criar os canos em um laço, colocando uma distância de 250 *pixels* entre cada um:

```
int posicao = 200;
for(int i=0; i<5; i++) {
    posicao += 250;
    Cano cano = new Cano(tela, posicao);
}
```

Mas queremos agora acumular todos esses canos para poder desenhá-los. Para isso devemos criar uma `List` de `Canos`:

```
List<Cano> canos = new ArrayList<Cano>();
int posicao = 200;
for(int i=0; i<5; i++) {
    posicao += 250;
    Cano cano = new Cano(tela, posicao);
    canos.add(cano);
}
```

Pronto, cinco canos em nossas mãos. Só falta desenhá-los. Devemos colocar a lista como variável membro e fazer um `for` na hora do desenho, mas calma lá: nosso código do `Game` está misturando muita coisa. Tem cano, tem pássaro, tem *background*, tem mais canos, tem *touch*, start, stop. Vamos isolar um pouco melhor as coisas. Assim como `Cano` é um elemento de nosso jogo, `Canos` também parece ser: temos um conjunto de canos no jogo e vamos criar uma classe chamada `Canos` para representar todos eles, no pacote **elementos**:

```
public class Canos {

    private List<Cano> canos = new ArrayList<Cano>();

    public Canos(Tela tela) {
        int posicaoInicial = 200;

        for(int i=0; i < 5; i++) {
            posicaoInicial += 250;
            canos.add(new Cano(tela, posicaoInicial));
        }
    }
}
```

Para tirar a impressão de estarmos usando "números mágicos", vamos refatorar e extrair algumas variáveis para dar significado aos números importantes do nosso jogo:

```
public class Canos {

    private static final int QUANTIDADE_DE_CANOS = 5;
    private static final int DISTANCIA_ENTRE_CANOS = 250;
    private final List<Cano> canos = new ArrayList<Cano>();

    public Canos(Tela tela) {
        int posicaoInicial = 200;

        for(int i=0; i<QUANTIDADE_DE_CANOS; i++) {
            posicaoInicial += DISTANCIA_ENTRE_CANOS;
            canos.add(new Cano(tela, posicaoInicial));
        }
    }
}
```

Agora sim, dentro de nosso `Game` a inicialização dos elementos fica mais simples: basta dizer que queremos criar os canos. Removemos o único cano que tínhamos e criamos todos eles de uma única vez:

```
private Canos canos;

private void inicializaElementos() {
    this.passaro = new Passaro();
    this.canos = new Canos(tela);
    Bitmap back = BitmapFactory.decodeResource(getResources(),
                    R.drawable.background);
    this.background = Bitmap.createScaledBitmap(back,
                        back.getWidth(), tela.getAltura(), false);
}
```

Falta agora desenhar esses cinco canos! Onde antes desenhávamos um único cano e o movíamos, chamaremos o método equivalente para todos de uma única vez:

```
canos.desenhaNo(canvas);
canos.move();
```

Claro, os dois métodos ainda não existem em nosso `Canos` e devemos implementá-los, fazendo um *loop* pelos nossos `canos`:

```
public void desenhaNo(Canvas canvas) {
    for(Cano cano : canos)
        cano.desenhaNo(canvas);
}

public void move() {
    for(Cano cano : canos)
        cano.move();
}
```

Rodamos nosso jogo e agora temos cinco canos! Todos de tamanho igual, é verdade, mas já são cinco canos!

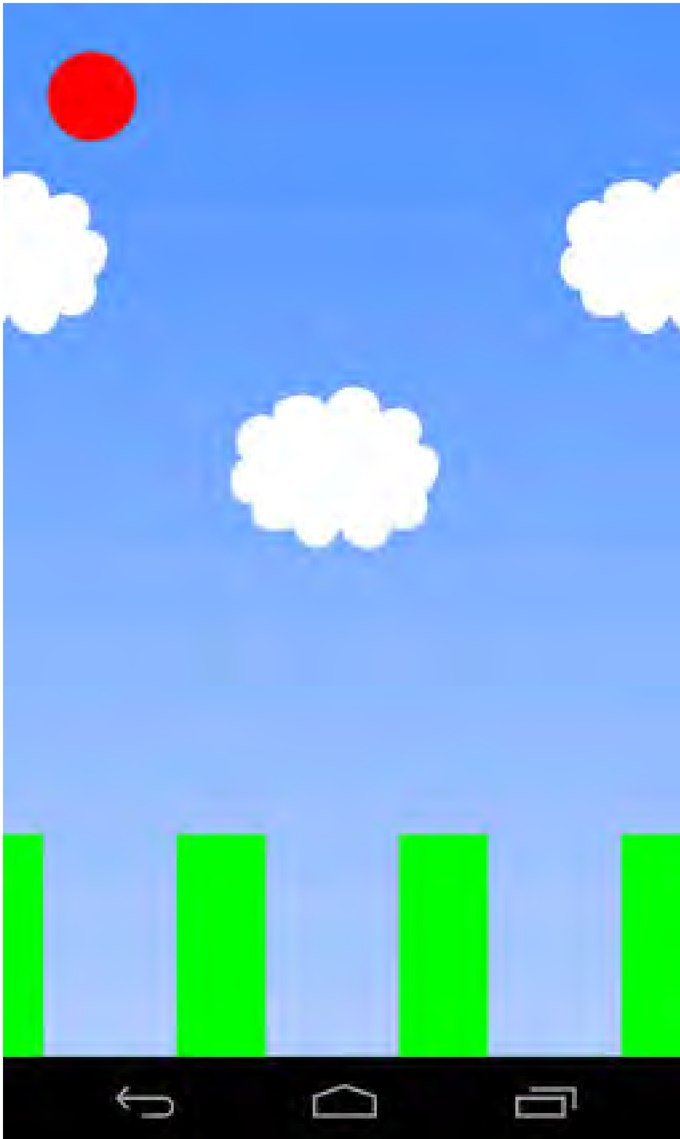


Fig. 6.1: Nosso jogo com os canos inferiores.

6.1 Criando os limites para pulo: o chão e teto

Vamos agora limitar nossos saltos e quedas. Primeiramente, quando o pássaro tocar na base da tela, devemos parar de cair.

Como estamos lidando com um círculo, para saber se sua borda toca a base da tela basta verificar se a sua `altura` mais seu `RAIO` é **maior** que o tamanho da `tela`:

```
public void cai() {
    boolean chegouNoChao = altura + RAIO > tela.getAltura();

    if(!chegouNoChao) {
        altura += 5;
    }
}
```

Veja que precisamos de um método que está na classe `Tela`, porém ainda não temos acesso a esse objeto dentro da classe `Passaro`. Para resolver isso, vamos receber a `Tela` no construtor do `Passaro`:

```
public class Passaro {

    private Tela tela;

    public Passaro(Tela tela) {
        this.tela = tela;
        //...
    }

    //Outros métodos...
}
```

Também não podemos pular além do topo da tela, isso é, temos que verificar quando a borda do nosso círculo toca o topo da tela. Como o topo da tela tem `altura` igual a zero, nosso círculo tocará o topo da tela **sempre** que sua `altura` for igual ao seu próprio `RAIO`. Logo, se essa `altura` for **maior**, sabemos que ele ainda não está no topo. Lembre-se que quanto maior a altura, mais para baixo da tela o elemento está ((0,0) é o canto **superior** esquerdo!).

Nosso método `pula`, para verificar se chegamos ao topo da tela, deve ficar assim:

```
public void pula() {
    if(altura > RAIO) {
        altura -= 150;
    }
}
```

Como alteramos o construtor do `Passaro`, apareceu um erro na classe `Game`. Basta passar a variável `tela` para o `Passaro`:

```
private void incializaElementos() {
    this.passaro = new Passaro(tela);
    //...
}
```

Capítulo 7

Criando os canos superiores

Ganhar um jogo que só possui canos embaixo é muito fácil, basta ficar voando no alto. Está na hora de deixarmos nosso jogo mais difícil! Para isso, praticaremos os cálculos de dimensão de tela e os métodos de desenho que já vimos anteriormente. Nosso próximo passo é colocar canos não só embaixo, mas também em cima da tela.

Não se esqueça de que o tamanho do cano inferior era igual ao tamanho da tela menos o tamanho do cano:

```
this.alturaDoCanoInferior = tela.getAltura() - TAMANHO_DO_CANO;
```

Portanto, é natural colocarmos o tamanho do cano superior como sendo o (ponto inicial da tela) mais o tamanho do cano:

```
this.alturaDoCanoInferior = tela.getAltura() - TAMANHO_DO_CANO;
this.alturaDoCanoSuperior = 0 + TAMANHO_DO_CANO;
```

Mas ainda tem algo de estranho: nossos canos têm todos o mesmo tamanho. Vamos aleatorizar um pouco essas dimensões pegando um número entre 0 e 150 para construir canos diferentes:

```
this.alturaDoCanoInferior =
    tela.getAltura() - TAMANHO_DO_CANO - valorAleatorio();
this.alturaDoCanoSuperior =
    0 + TAMANHO_DO_CANO + valorAleatorio();
```

Definimos o `valorAlteatorio`:

```
private int valorAleatorio() {
    return (int) (Math.random() * 150);
}
```

Agora, precisamos desenhar o cano de baixo **e** o cano de cima. Para desenhar o cano superior, vamos criar um método `desenhaCanoSuperiorNo`:

```
private final Paint VERDE = Cores.getCorDoCano();

public void desenhaNo(Canvas canvas) {
    desenhaCanoSuperiorNo(canvas);
    desenhaCanoInferiorNo(canvas);
}

private void desenhaCanoSuperiorNo(Canvas canvas) {
    canvas.drawRect(??, ??, ??, ??, VERDE);
}

private void desenhaCanoInferiorNo(Canvas canvas) {
    canvas.drawRect(posicao, alturaDoCanoInferior,
        posicao + LARGURA_DO_CANO, tela.getAltura(), VERDE);
}
```

Vamos olhar com calma o conteúdo do método `drawRect` do `desenhaCanoSuperiorNo`:

```
canvas.drawRect(??, ??, ??, ??, VERDE);
```

Lembre-se de que os dois primeiros argumentos representam o **canto superior esquerdo** do retângulo e os outros dois são o seu **canto inferior direito**. Sendo assim, onde estará o canto superior esquerdo do nosso cano superior? No topo da tela!

Logo, o segundo argumento deverá ser zero:

```
canvas.drawRect(??, 0, ??, ??, VERDE);
```

Qual deverá ser a altura do cano superior? Como já criamos uma variável `alturaDoCanoSuperior`, podemos passá-la diretamente ao quarto argumento:

```
canvas.drawRect(??, 0, ??, alturaDoCanoSuperior, VERDE);
```

A altura do cano superior está definida. Vamos ver como ficará sua largura. Como nosso cano superior deverá ter a **mesma largura do cano inferior**, devemos passar os mesmos valores para os dois. Como o cano inferior começa em `posicao` e tem `posicao + LARGURA_DO_CANO` de largura, vamos passar esses mesmos valores para o cano superior!

```
canvas.drawRect(posicao, 0, posicao + LARGURA_DO_CANO,
    alturaDoCanoSuperior, VERDE);
```

Nosso método `desenhaCanoSuperiorNo` ficará assim:

```
private void desenhaCanoSuperiorNo(Canvas canvas) {
    canvas.drawRect(posicao, 0, posicao + LARGURA_DO_CANO,
                    alturaDoCanoSuperior, VERDE);
}
```

Agora podemos rodar nosso jogo novamente e vemos que temos dez canos com tamanhos diferentes para desviarmos!

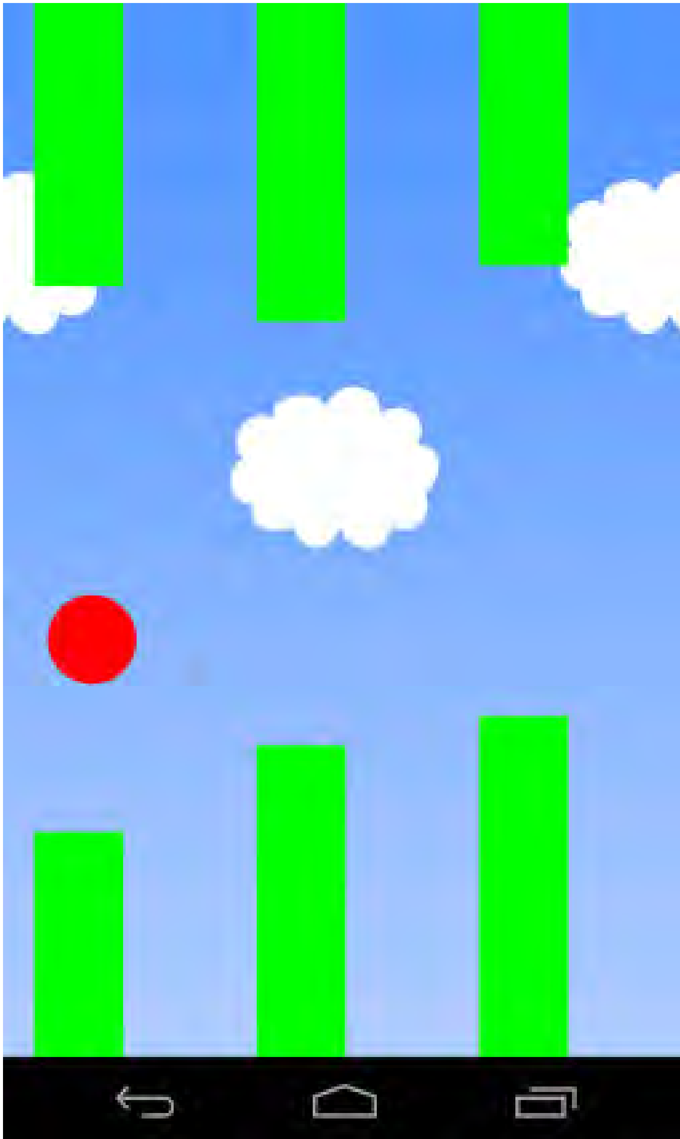


Fig. 7.1: Canos com tamanhos diferentes.

Mas nosso jogo ainda é muito curto. Cinco pares de canos passam muito rápido por nossa tela e ficamos sem ter mais o que fazer no jogo. Vamos fazer

com que os canos nunca parem de aparecer. Para isso precisamos de canos infinitos, mas como?

7.1 Criando infinitos canos

Uma tática em jogos infinitos é de fazer com que quando o cano saia da tela, um novo cano seja colocado no fim de todos os canos: dessa maneira sempre temos a mesma quantidade de canos. Lembre-se de nosso método de mover canos:

```
public void move() {
    for(Cano cano : canos) {
        cano.move();
    }
}
```

Após mover um cano, verificaremos se ele saiu da tela:

```
for(Cano cano : canos) {
    cano.move();
    if(cano.saiuDaTela()) {
        //O que faremos?
    }
}
```

Caso ele tenha saído da tela, podemos criar outro `Cano` e adicioná-lo à nossa `List<Cano>`:

```
for(Cano cano : canos) {
    cano.move();
    if(cano.saiuDaTela()) {
        Cano outroCano =
            new Cano(tela, getMaximo() + DISTANCIA_ENTRE_CANOS);
        canos.add(outroCano);
    }
}
```

Perceba que tentamos adicionar um `cano` à mesma lista que é percorrida no `for`. Isso causará uma `ConcurrentModificationException`! Para

adicionar algo à lista que é percorrida, precisaremos de um `ListIterator`; para obtermos esse objeto basta chamar o método `listIterator` que toda `List` possui. Portanto, em vez de um `for` vamos alterá-lo para um `while`:

```
ListIterator<Cano> iterator = canos.listIterator();
while(iterator.hasNext()) {
    Cano cano = (Cano) iterator.next();
    cano.move();

    if(cano.saiuDaTela()) {
        Cano outroCano =
            new Cano(tela, getMaximo() + DISTANCIA_ENTRE_CANOS);
        iterator.add(outroCano);
    }
}
```

Para verificar se um cano saiu da tela, na classe `Cano` basta conferir se sua posição é menor que zero:

```
public boolean saiuDaTela() {
    return posicao < 0;
}
```

Quase isso... O cano não sai da tela quando seu ponto esquerdo sai da tela, ele sai quando ele passou por completo (toda sua largura):

```
public boolean saiuDaTela() {
    return posicao + LARGURA_DO_CANO < 0;
}
```

Agora só precisamos saber como faremos o método `getMaximo`. Temos que adicionar um novo `Cano` àqueles já existentes, em qual posição ele deverá ser colocado?

Precisamos saber qual é o `Cano` que possui a maior posição (ou seja, o `Cano` mais à direita possível) da nossa lista para podermos determinar a posição do novo `Cano`. Então, basta percorrer nossa lista e comparar a posição:

```
private int getMaximo() {
    int maximo = 0;
```

```
    for(Cano cano : canos) {
        maximo = Math.max(cano.getPosicao(), maximo);
    }
    return maximo;
}
```

Não se esqueça de fazer o `getPosicao` na classe `Cano`.

Com isso, sempre teremos um novo `Cano` criado assim que algum sair da tela. Agora temos uma sequência infinita de canos dos quais o jogador precisa desviar!

Neste momento, ao jogar o Jumper, veremos algo estranho conforme o tempo passa. O que será isso?

7.2 Descartando canos anteriores

Conforme o tempo passa (e os `Canos` saem da tela), vamos criando outros `Canos` e adicionando-os à lista. Veja que ela está ficando cada vez maior, com mais elementos! A partir de um momento é de se esperar que teremos tantos `Canos` na nossa lista, que nosso aparelho não vai dar conta de gerenciá-los e o Jumper ficará bastante lento.

Para resolver esse problema, podemos nos livrar de alguns `Canos`. Quais? Justamente aqueles que saíram da tela!

Logo, se o `Cano` sair da tela, vamos removê-lo da lista!

Vamos usar o método `remove` do `Iterator` assim que o `cano` sair da tela:

```
ListIterator<Cano> iterator = canos.listIterator();
while(iterator.hasNext()) {
    Cano cano = (Cano) iterator.next();
    cano.move();

    if(cano.saiuDaTela()) {
        iterator.remove();
        Cano outroCano =
            new Cano(tela, getMaximo() + DISTANCIA_ENTRE_CANOS);
        iterator.add(outroCano);
```

```
    }
}
```

Rodando o jogo agora temos uma serie infinita de canos de tamanhos diferentes!

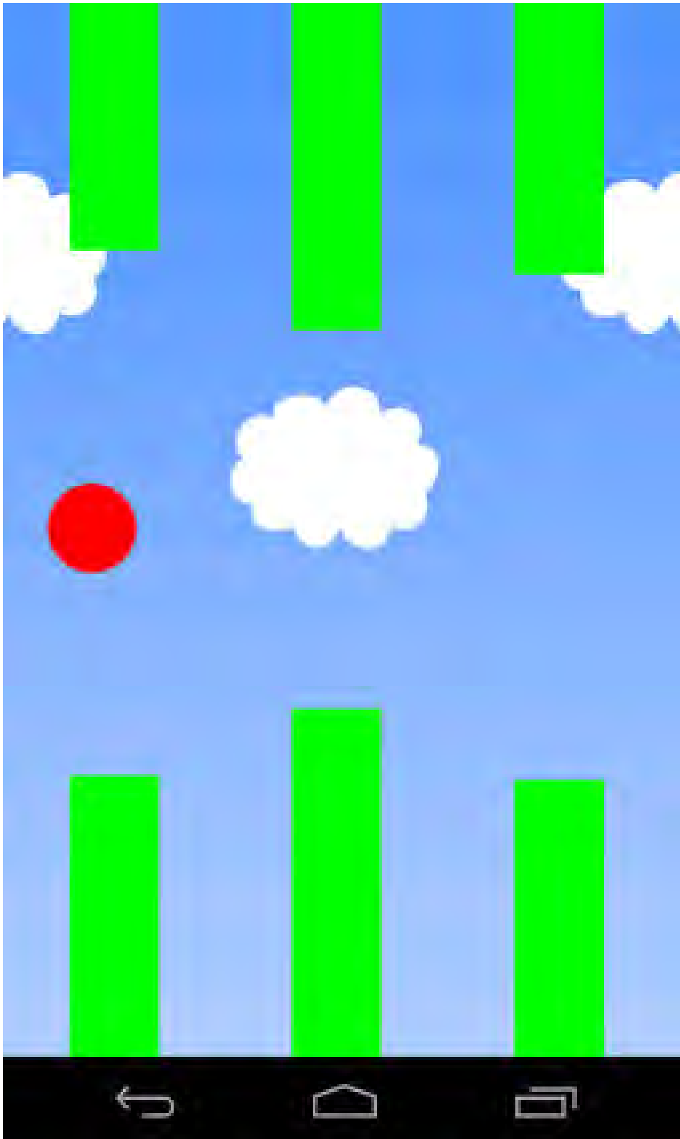


Fig. 7.2: Agora podemos gerar infinitos canos.

Capítulo 8

Criando a pontuação do jogo

Chegou a hora de colocar uma maneira de nos desafiarmos no jogo: quantos pontos somos capazes de fazer? Para contá-los, ao inicializar o jogo devemos criar um novo objeto, a `Pontuacao`:

```
private void inicializaElementos() {
    this.pontuacao = new Pontuacao();
    // outras inicializações
}
```

Os pontos sobem cada vez que um `Cano` sai da tela, demonstrando a vitória do pássaro sobre os canos. No nosso método de movimento dos canos, devemos aumentar a pontuação toda vez que um cano sai da tela:

```
public void move() {
    ListIterator<Cano> iterator = canos.listIterator();
```

```
    while(iterator.hasNext()) {
        Cano cano = (Cano) iterator.next();
        cano.move();

        if(cano.saiuDaTela()) {
            pontuacao.aumenta();
            iterator.remove();
            Cano outroCano =
             new Cano(tela, getMaximo() + DISTANCIA_ENTRE_CANOS);
            iterator.add(outroCano);
        }
    }
}
```

Para isso a classe `Canos` deve ter acesso à pontuação. Devemos passá-la no construtor durante a inicialização de nossos elementos:

```
private void inicializaElementos() {
    this.pontuacao = new Pontuacao();
    this.passaro = new Passaro();
    this.canos = new Canos(tela, pontuacao);
    Bitmap back = BitmapFactory.decodeResource(getResources(),
                    R.drawable.background);
    this.background = Bitmap.createScaledBitmap(back,
                       back.getWidth(), tela.getAltura(), false);
}
```

E setar a variável no construtor:

```
private final Pontuacao pontuacao;

public Canos(Tela tela, Pontuacao pontuacao) {
    this.pontuacao = pontuacao;
    int posicaoInicial = 200;

    for (int i = 0; i < QUANTIDADE_DE_CANOS; i++) {
        posicaoInicial += DISTANCIA_ENTRE_CANOS;
        canos.add(new Cano(tela, posicaoInicial));
    }
}
```

Faltou criar o método de aumento de pontos dentro de nossa classe `Pontuacao`:

```
public class Pontuacao {
    private int pontos = 0;

    public void aumenta() {
        pontos++;
    }
}
```

Pronto. Quando o cano sai da tela, notificamos a pontuação de que ela deve aumentar em um. Mas onde ela vai aparecer na tela? Assim como qualquer outro elemento, vamos implementar o método `desenhaNo`, pedindo para desenhar um texto por meio do método `drawText` do `canvas`:

```
public class Pontuacao {
    private static final Paint BRANCO =
        Cores.getCorDaPontuacao();
    private int pontos = 0;

    public void aumenta() {
        pontos++;
    }

    public void desenhaNo(Canvas canvas) {
        canvas.drawText(String.valueOf(pontos), 100, 100,
                        BRANCO);
    }
}
```

O método `drawText` recebe o texto a ser escrito, o ponto (x,y) onde esse texto deverá ser posicionado e sua cor. Vamos desenhar um texto na posição (100,100) da nossa tela na cor branca.

Agora, basta definir a cor branca na classe `Cores`:

```
public static Paint getCorDaPontuacao() {
    Paint branco = new Paint();
    branco.setColor(0xFFFFFFFF);
```

```
    return branco;
}
```

E pedir para desenhar no nosso jogo:

```
@Override
public void run() {
    while (isRunning) {
        if (!holder.getSurface().isValid()) continue;

        canvas = holder.lockCanvas();
        canvas.drawBitmap(background, 0, 0, null);

        passaro.desenhaNo(canvas);
        passaro.cai();

        pontuacao.desenhaNo(canvas);

        canos.desenhaNo(canvas);
        canos.move();

        holder.unlockCanvasAndPost(canvas);
    }
}
```

Desse jeito, a pontuação ficou **muito** pequena. É bastante difícil enxergar o número no canto superior esquerdo da tela:

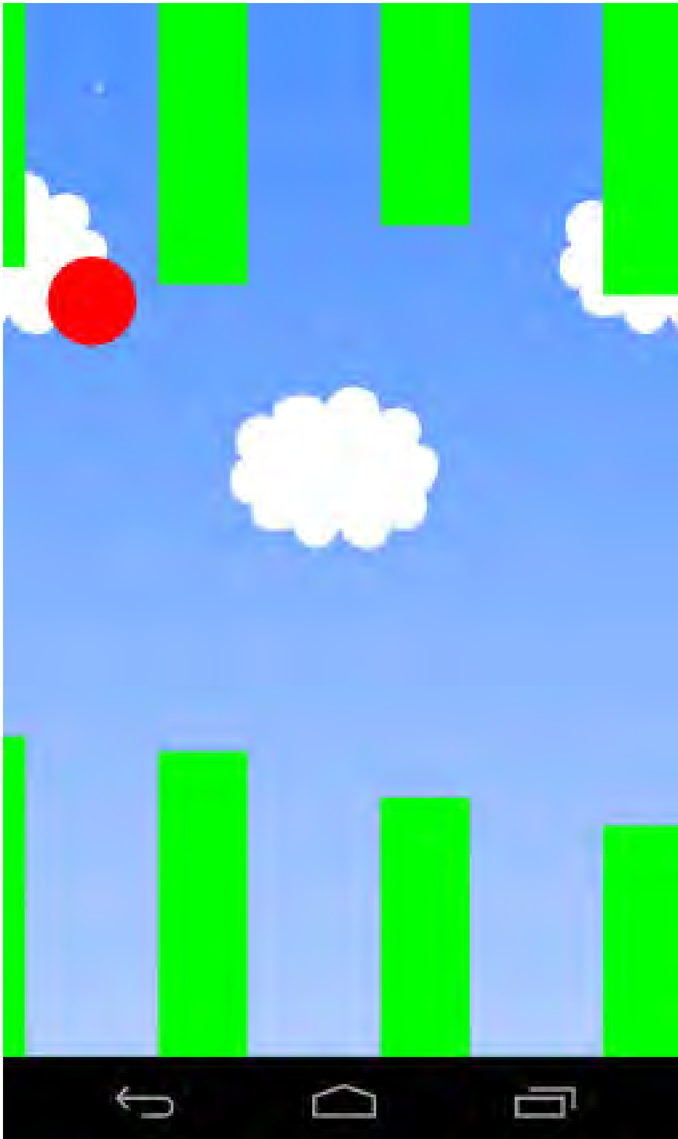


Fig. 8.1: Será que ficou legal para o jogador?

Que tal colocarmos uma fonte maior e com *bold* (negrito)? Repare que o objeto `Paint` permite configurar diversas características relacionadas ao

elemento que estamos desenhando na tela.

```
public static Paint getCorDaPontuacao() {
    Paint branco = new Paint();
    branco.setColor(0xFFFFFFFF);
    branco.setTextSize(80);
    branco.setTypeface(Typeface.DEFAULT_BOLD);

    return branco;
}
```

Bem melhor! Agora é possível ver a pontuação sem problemas.

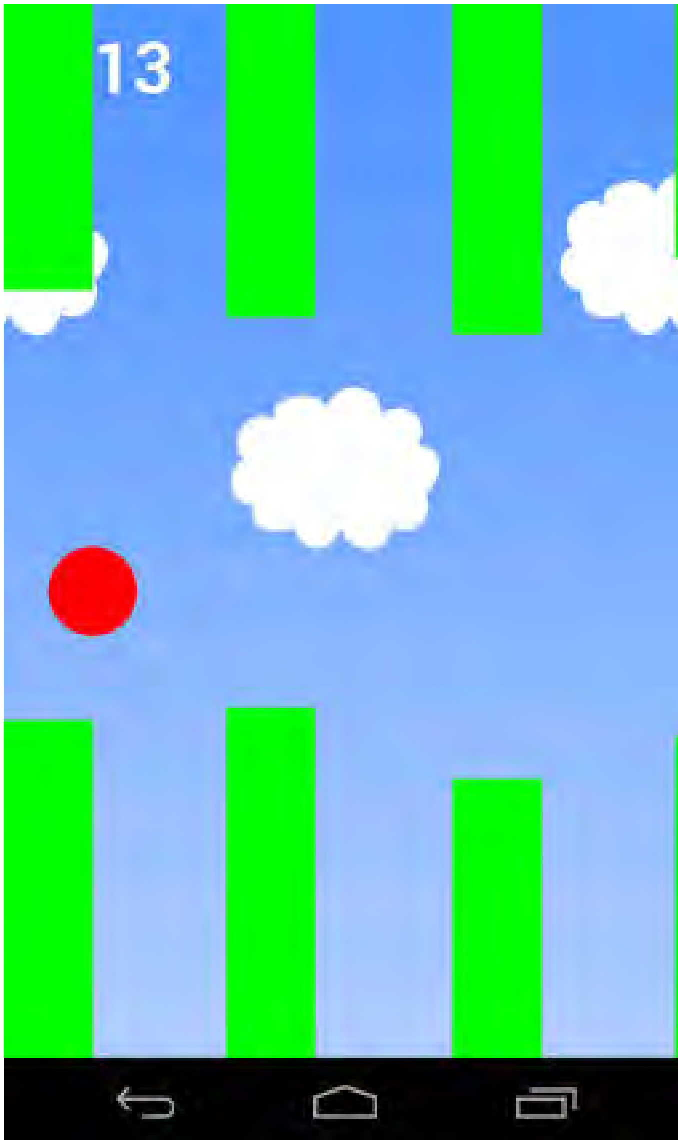


Fig. 8.2: Aumentando um pouco a fonte, ficou bem melhor, não?

Podemos ir além e definir uma sombra para a nossa pontuação. A sombra é um recurso comum utilizado pelo pessoal de design para dar destaque

em um texto que está por cima de uma imagem. Basta usarmos o método `setShadowLayer` do `Paint`:

```
branco.setShadowLayer(??, ??, ??, ??);
```

No último argumento dizemos qual a cor dessa sombra. No nosso caso, vamos deixá-la preta:

```
branco.setShadowLayer(??, ??, ??, 0xFF000000);
```

Temos que dizer qual será o deslocamento dessa sombra em relação ao texto original; quanto mais deslocada, mais visível ela estará, porém menos natural será sua aparência. Vamos deslocar 5 *pixels* para baixo e 5 *pixels* para a direita:

```
branco.setShadowLayer(??, 5, 5, 0xFF000000);
```

Agora, temos que dizer o quão definida será a borda dessa sombra. Quanto maior o valor, menos definidos serão os seus limites (mais "esfumaçada" será). Vamos colocar um valor 3 para esse argumento:

```
branco.setShadowLayer(3, 5, 5, 0xFF000000);
```

Pronto! Basta chamar esse método no `getCorDaPontuacao`:

```
public static Paint getCorDaPontuacao() {
    Paint branco = new Paint();
    branco.setColor(0xFFFFFFFF);
    branco.setTextSize(80);
    branco.setTypeface(Typeface.DEFAULT_BOLD);
    branco.setShadowLayer(3, 5, 5, 0xFF000000);

    return branco;
}
```

Veja que a pontuação possui uma sombra:

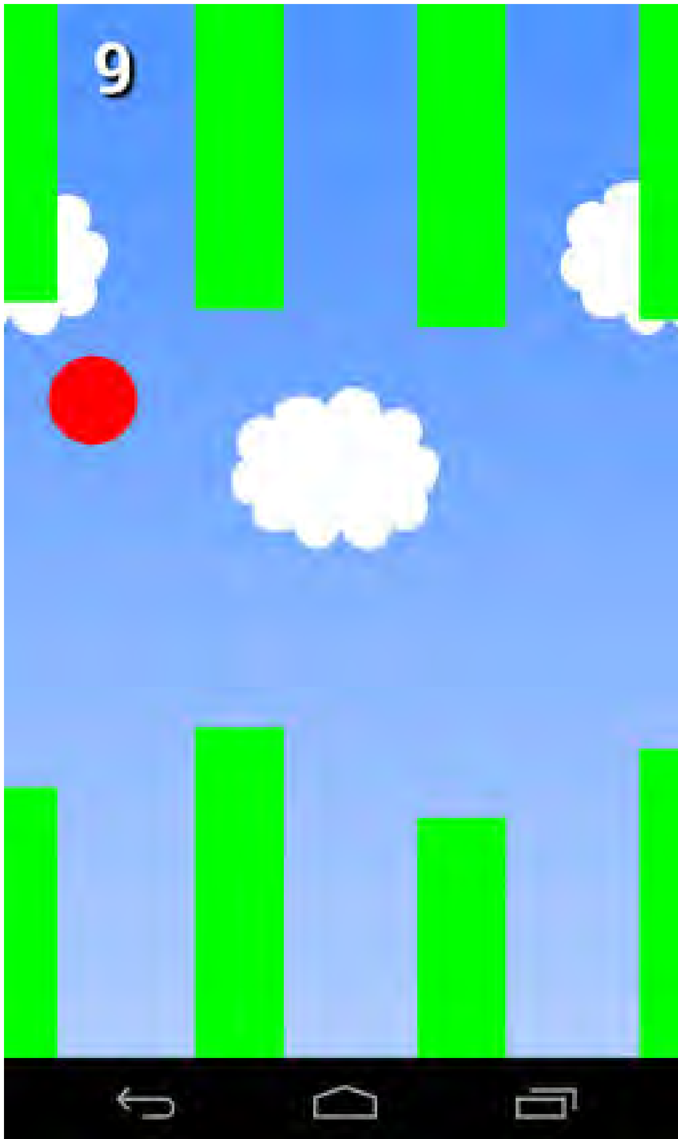


Fig. 8.3: Pontuação com sombra para um maior destaque.

Nosso jogo já parece bem legal! Porém, temos um pequeno problema: o que acontece com a pontuação quando os canos passam pela posição onde

ela está desenhada?

Veja o *loop* principal da classe `Game`:

```
@Override
public void run() {
    while(isRunning) {
        //Lock canvas...

        canvas.drawBitmap(background, 0, 0, null);
        passaro.desenhaNo(canvas);
        passaro.cai();

        pontuacao.desenhaNo(canvas);

        canos.desenhaNo(canvas);
        canos.move();

        //Unlock...
    }
}
```

Perceba que a `pontuacao` é desenhada **antes** dos `canos`. Como o *canvas* trabalha com desenho por camadas, as camadas desenhadas primeiro ficarão por **baixo** das desenhadas posteriormente.

Isso significa que os `canos` "passarão por cima" da nossa pontuação! Queremos que a pontuação **sempre** esteja visível ao jogador, então vamos apenas trocar a ordem das camadas:

```
@Override
public void run() {
    while(isRunning) {
        //Lock canvas...

        canvas.drawBitmap(background, 0, 0, null);
        passaro.desenhaNo(canvas);
        passaro.cai();

        canos.desenhaNo(canvas);
        canos.move();
```

```
        pontuacao.desenhaNo(canvas);

        //Unlock...
    }
}
```

Veja que a pontuação agora fica em cima dos canos. Perfeito!

Fig. 8.4: A pontuação é exibida por cima dos canos.

Capítulo 9

Tratando colisões

Quando nosso jogo acaba? Quando o pássaro bater em qualquer cano. Mas como descobrir que um pássaro encostou no cano? Como nosso `Passaro` é um círculo e o `Cano` um retângulo, temos que descobrir se o círculo tem alguma intersecção com o retângulo.

Caso nosso `Passaro` colida com algum `Cano`, podemos simplesmente cancelar o *loop* principal. Como temos uma variável `isRunning` que controla o *loop*, para cancelá-lo basta setar `isRunning` para `false`:

```
if(new VerificadorDeColisao(passaro, canos).temColisao()) {
    isRunning = false;
}
```

Ainda não temos a classe `VerificadorDeColisao`. Vamos criá-la no pacote **engine**.

Como precisamos verificar a colisão entre o `Passaro` e os `Canos`, vamos receber esses objetos no construtor da classe `VerificadorDeColisao`:

```
public class VerificadorDeColisao {

    private final Passaro passaro;
    private final Canos canos;

    public VerificadorDeColisao(Passaro passaro, Canos canos) {
        this.passaro = passaro;
        this.canos = canos;
    }

    public boolean temColisao() {
        return canos.temColisaoCom(passaro);
    }
}
```

Mas como implementar o algoritmo de colisão? Precisamos conferir que o `Passaro` não bateu em nenhum cano, portanto devemos passar por cada um dos canos da classe `Canos`:

```
public boolean temColisaoCom(Passaro passaro) {
    for (Cano cano : canos) {
    }
}
```

Se ele não bater em nenhum cano, não tem colisão:

```
public boolean temColisaoCom(Passaro passaro) {
    for (Cano cano : canos) {
    }
    return false;
}
```

Mas temos que descobrir se o `Passaro` bateu com o cano atual. Para isso verificaremos se ele colidiu verticalmente **e** horizontalmente:

```
public boolean temColisaoCom(Passaro passaro) {
    for (Cano cano : canos) {
```

```
        if (cano.temColisaoHorizontalCom(passaro)
                    && cano.temColisaoVerticalCom(passaro)) {
            return true;
        }
    }
    return false;
}
```

Agora falta implementar os dois métodos de colisão na classe `Cano`.

Para detectarmos uma colisão vertical entre o `Passaro` e o `Cano` inferior, basta sabermos quando a **borda do pássaro** toca o topo do cano. Ou seja, quando a altura do pássaro mais seu raio for maior que a `alturaDoCanoInferior`. Lembre-se que, quanto maior o valor da altura, mais baixo está o elemento!

Para o cano superior, a ideia é semelhante: precisamos saber quando a borda superior do pássaro toca a base do cano superior. Para pegarmos a borda superior do pássaro basta subtrair o `RAIO` da sua `altura`.

Com isso, temos o método `temColisaoVerticalCom`:

```
public boolean temColisaoVerticalCom(Passaro passaro) {
    return passaro.getAltura() -
           passaro.RAIO < this.alturaDoCanoSuperior
           || passaro.getAltura() + passaro.RAIO >
           this.alturaDoCanoInferior;
}
```

Se a distância entre a `posicao` horizontal do cano e a posição `X` (horizontal) do pássaro for **menor** que o seu `RAIO`, sabemos que houve uma colisão. O método `temColisaoHorizontalCom` tem esse conteúdo:

```
public boolean temColisaoHorizontalCom(Passaro passaro) {
    return this.posicao - passaro.X < passaro.RAIO;
}
```

Temos nosso `VerificadorDeColisao` pronto! Ao bater em um cano, o que acontece com o jogo?

9.1 Criando a tela de game over

Testamos o jogo e ele para repentinamente quando perdemos. Faltou algo mais bonito aqui, uma tela de fim de jogo, de *game over*. Vamos mudar nossa verificação de colisão para desenhar a tela de *game over*:

```
if(new VerificadorDeColisao(passaro, canos).temColisao()) {
    new GameOver(tela).desenhaNo(canvas);
    isRunning = false;
}
```

Nossa tela de *game over* deve ser construída no pacote **elementos** e ter o método de desenho tradicional:

```
public class GameOver {

    private final Tela tela;

    public GameOver(Tela tela) {
        this.tela = tela;
    }

    public void desenhaNo(Canvas canvas) {
    }
}
```

Vamos criar a fonte do texto de *game over*, uma fonte vermelha, com negrito e sombra, definida na classe `Cores`:

```
public static Paint getCorDoGameOver() {
    Paint vermelho = new Paint();
    vermelho.setColor(0xFFFF0000);
    vermelho.setTextSize(50);
    vermelho.setTypeface(Typeface.DEFAULT_BOLD);
    vermelho.setShadowLayer(2, 3, 3, 0xFF000000);
    return vermelho;
}
```

Partimos para o desenho em si, primeiramente colocando o texto no meio da tela verticalmente (`tela.getAltura() / 2`), à esquerda:

```java
private static final Paint VERMELHO = Cores.getCorDoGameOver();

public void desenhaNo(Canvas canvas) {
    String gameOver = "Game Over";
    canvas.drawText(gameOver, 0, tela.getAltura() / 2, VERMELHO);
}
```

Fig. 9.1: Game over exibido no jogo.

9.2 Centralizando um texto horizontalmente na tela

O nosso `GameOver` já funciona mas ainda precisamos centralizá-lo na horizontal. Este já passa a ser um problema um pouco maior. Para centralizar na horizontal temos que saber quanto o texto `"Game Over"` ocupa da tela e isso depende do tamanho da fonte, da tela etc. Felizmente, o objeto `Paint` pode nos ajudar e dizer quantos *pixels* serão necessários para desenhar essa `String` na tela. Com o método `getTextBounds` ficamos sabendo qual o tamanho do retângulo que engloba toda a nossa `String`:

```
Rect limiteDoTexto = new Rect();
vermelho.getTextBounds(texto, 0, texto.length(), limiteDoTexto);
```

No método `getTextBounds` dizemos que queremos o retângulo (`limiteDoTexto`) que engloba a `String` (`texto`), que começa na posição `0` e termina em `texto.length()`.

Com isso, nosso `limiteDoTexto` passa a ser o retângulo que contém as dimensões exatas do `texto` (no caso, `"Game Over"`). Agora, basta descobrir onde está o centro horizontal desse retângulo; se soubermos isso, podemos centralizá-lo horizontalmente na tela.

Para encontrarmos o centro horizontal do nosso retângulo `limiteDoTexto`, basta dividir seu tamanho horizontal por 2. Mas qual é o seu tamanho horizontal?

Seu tamanho horizontal depende de onde o retângulo começa (`left`) e onde ele acaba (`right`). Basta fazer:

```
(limiteDoTexto.right - limiteDoTexto.left)/2
```

Agora que temos o centro horizontal do retângulo `limiteDoTexto`, queremos centralizá-lo na `tela` e para isso vamos precisar da sua largura. Caso coloquemos nosso texto **começando** na metade da `tela`, ele não estará centralizado (somente a primeira letra estará realmente no centro); então, vamos encontrar o centro horizontal da tela e **"recuar" metade do nosso retângulo**. Isso alinhará o centro do retângulo com o centro horizontal da tela:

```
int centroHorizontal = tela.getLargura()/2 -
    (limiteDoTexto.right - limiteDoTexto.left)/2;
```

Com isso, nosso `centralizaTexto` ficará assim:

```
private int centralizaTexto(String texto) {
    Rect limiteDoTexto = new Rect();
    vermelho.getTextBounds(texto, 0, texto.length(),
                            limiteDoTexto);
    int centroHorizontal = tela.getLargura()/2 -
        (limiteDoTexto.right - limiteDoTexto.left)/2;
    return centroHorizontal;
}
```

Precisamos também criar o `getLargura` na classe `Tela`, claro:

```
public int getLargura() {
    return metrics.widthPixels;
}
```

E mandar desenhar nossa `String` no centro horizontal:

```
public void desenhaNo(Canvas canvas) {

    String gameOver = "Game Over";
    int centroHorizontal = centralizaTexto(gameOver);

    canvas.drawText(gameOver, centroHorizontal,
                    tela.getAltura()/2, VERMELHO);
}
```

Agora sim podemos ver o resultado final.

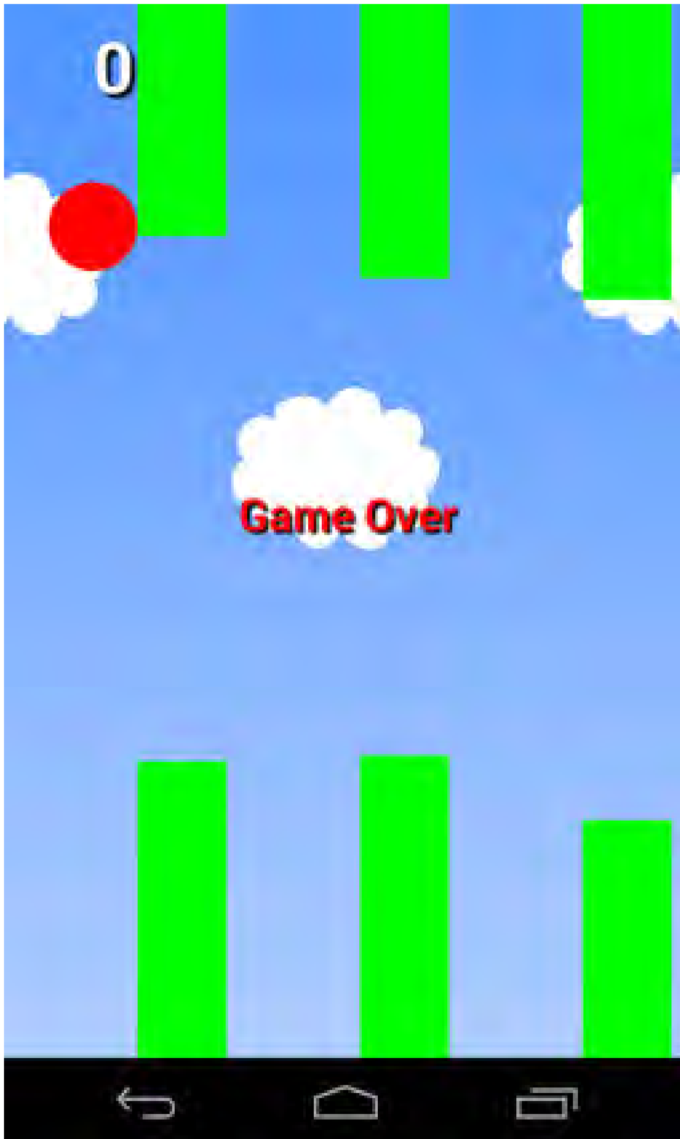


Fig. 9.2: Game over é exibido centralizado na tela.

Capítulo 10

Aprimorando o *layout* do jogo

Agora que temos toda a mecânica do jogo funcionando corretamente, vamos melhorar o aspecto visual dos elementos do Jumper substituindo o círculo do `Passaro` e os retângulos dos `Canos` por imagens bonitas!

Vamos precisar de uma imagem para representar o `Passaro`, mas não se preocupe: todas as imagens que usaremos já estão no `zip` em https://github.com/felipetorres/jumper-arquivos.

Tudo o que precisamos fazer é utilizar essa imagem no nosso jogo para substituir o círculo vermelho. Perceba que nossa imagem é basicamente um círculo, dessa forma podemos manter todas as contas de tamanho e colisão inalteradas!

Para utilizar a imagem chamada `passaro.png` da pasta `drawable-nodpi`, vamos chamar o método `decodeResource` da classe `BitmapFactory`, da mesma forma como fizemos para o background:

```
Bitmap bp = BitmapFactory.decodeResource(context.getResources(),
                                         R.drawable.passaro);
```

Qual tamanho essa imagem terá no nosso jogo? Será que é **exatamente** o mesmo tamanho do círculo vermelho? Temos que tomar cuidado com isso, pois se a imagem for maior (ou menor) teremos problemas no tratamento das colisões.

Para garantir que as dimensões sejam as mesmas do círculo, no construtor da classe `Passaro` vamos redimensionar nossa imagem com o método `createScaledBitmap` e passar as dimensões que nosso círculo tinha!

```
Bitmap bp = BitmapFactory.decodeResource(context.getResources(),
                                         R.drawable.passaro);
this.passaro = Bitmap.createScaledBitmap(bp, RAIO*2, RAIO*2 ,
                                         false);
```

Como precisamos de um `Context` para o `getResources`, basta chamá-lo no construtor do `Passaro`.

Veja que passamos `RAIO*2` para a altura e largura do novo *bitmap*. Como nosso círculo tinha o raio do tamanho `RAIO`, sua largura e altura eram `RAIO*2`; agora nossa imagem tem as mesmas dimensões que o círculo.

Garantimos que nossa imagem está com o tamanho correto. Podemos garantir que essa imagem será desenhada com o mesmo posicionamento do círculo?

Ao lidar com círculos, usamos o seu centro como referência de posicionamento. Dissemos que seu centro estaria na posição (*X*, *altura*):

```
canvas.drawCircle(X, altura, RAIO, vermelho);
```

Porém, ao lidar com imagens (*bitmaps*), a referência para desenho é o **canto superior esquerdo**. Logo, se mantivermos (*X*, *altura*) para seu desenho,

```
canvas.drawBitmap(passaro, X, altura, null);
```

teremos uma imagem desalinhada com a posição original. Para arrumar isso, vamos **centralizar** a imagem em (*X*, *altura*). Como ela tem `RAIO*2` de altura e largura, basta subtrair `RAIO` de ambas as coordenadas e teremos o *bitmap* centralizado:

```
canvas.drawBitmap(passaro, X-RAIO, altura-RAIO, null);
```

Veja como ficará nossa aplicação:

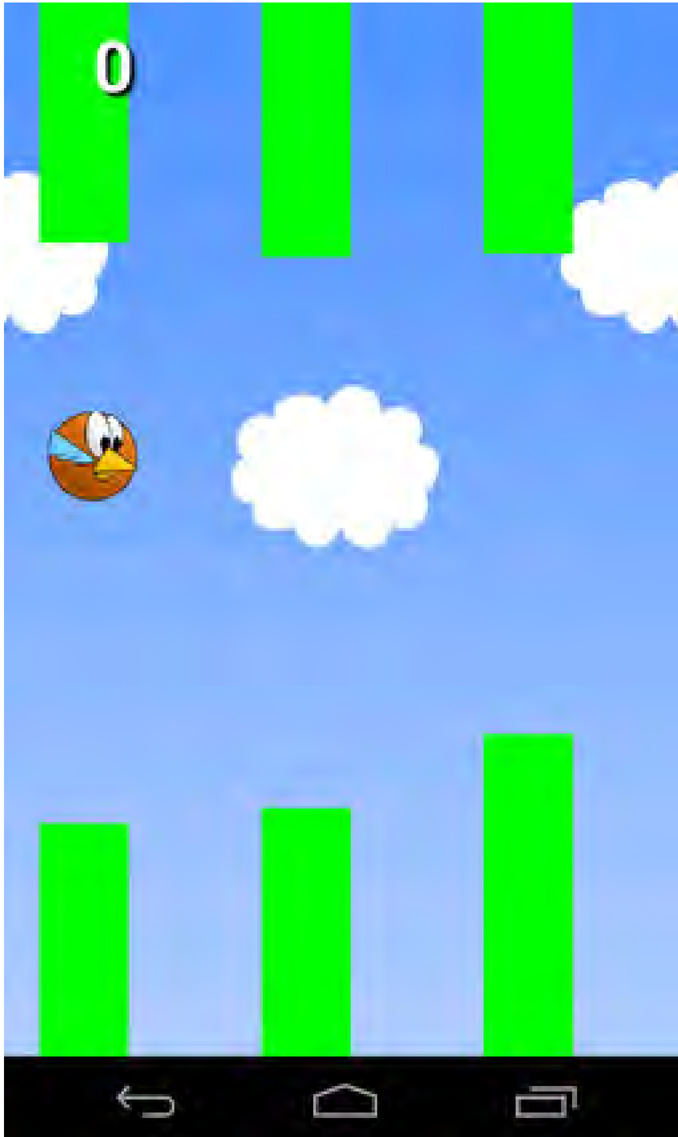


Fig. 10.1: Nosso pássaro ficou bem melhor, não?

10.1 Substituindo os retângulos por bitmaps

Vamos usar a mesma ideia anterior para substituir os retângulos dos Canos por *bitmaps* bonitos! Porém, em vez de possuir uma imagem para **todo** o cano, teremos apenas um "filete", que será esticado para o mesmo tamanho dos retângulos iniciais.

Assim como feito no Passaro, na classe Canos vamos utilizar o BitmapFactory para pegar a imagem. Precisaremos de um Context em seu construtor:

```
Bitmap bp = BitmapFactory.decodeResource(context.getResources(),
                                          R.drawable.cano);
```

Para o cano inferior, vamos redimensionar esse filete que está em bp para o tamanho correto. Veja que esse processo é bem simples, pois já temos as dimensões guardadas em LARGURA_DO_CANO e alturaDoCanoInferior:

```
this.canoInferior = Bitmap.createScaledBitmap(bp,
    LARGURA_DO_CANO, this.alturaDoCanoInferior, false);
```

Para desenhar esse canoInferior na tela, devemos alterar o desenhaCanoInferiorNo. Atualmente, temos o seguinte código:

```
private void desenhaCanoInferiorNo(Canvas canvas) {
    canvas.drawRect(posicao, alturaDoCanoInferior,
        posicao + LARGURA_DO_CANO, tela.getAltura(), VERDE);
}
```

Lembre-se que os dois primeiros valores do drawRect representam o **canto superior esquerdo** do retângulo. Da mesma forma que o drawBitmap! Ou seja, ao contrário do Passaro, no qual tivemos que fazer uma pequena correção, para os Canos já temos essas coordenadas prontas!

```
canvas.drawBitmap(canoInferior, posicao, alturaDoCanoInferior,
                  null);
```

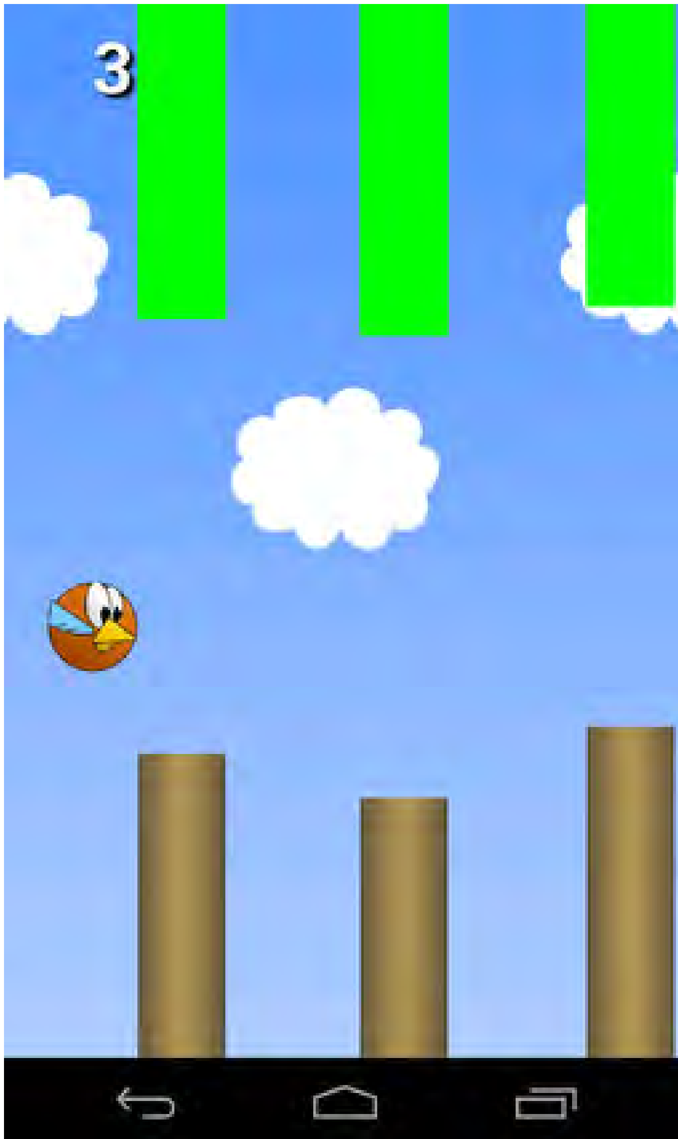


Fig. 10.2: Canos inferiores agora são bitmaps também.

Para os canos superiores, vamos fazer da mesma forma, porém não teremos que utilizar o `BitmapFactory` novamente, basta redimensioná-lo de

acordo com os retângulos superiores:

```
this.canoSuperior = Bitmap.createScaledBitmap(bp,
    LARGURA_DO_CANO, this.alturaDoCanoSuperior, false);
```

Vamos substituir o `desenhaCanoSuperiorNo`, onde chamávamos o `drawRect`, pelo `drawBitmap`, mas antes precisaremos da coordenada do seu canto superior esquerdo:

```
private void desenhaCanoSuperiorNo(Canvas canvas) {
    canvas.drawRect(posicao, 0, posicao + LARGURA_DO_CANO,
         alturaDoCanoSuperior, VERDE);
}
```

Como esse retângulo está com seu canto superior esquerdo em (*posicao*, *0*) teremos que usar essa mesma coordenada para nosso *bitmap*:

```
private void desenhaCanoSuperiorNo(Canvas canvas) {
    canvas.drawBitmap(canoSuperior, posicao, 0, null);
}
```

Ao término, temos nosso Jumper bem mais bonito e sem nenhuma mudança drástica no código!

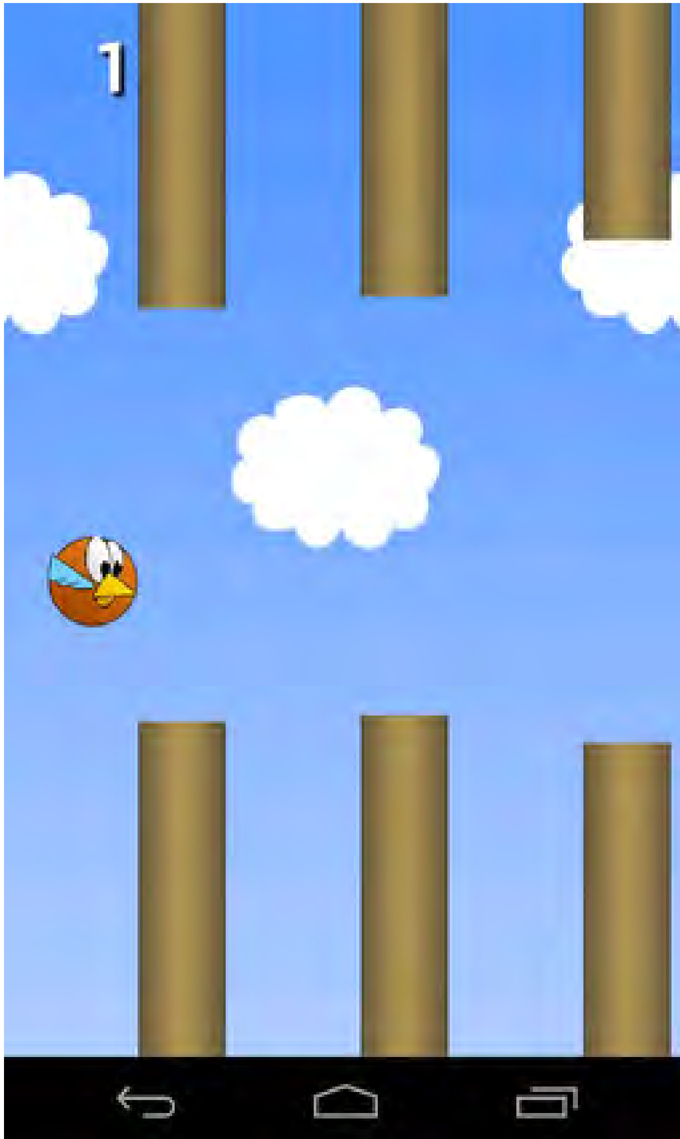


Fig. 10.3: Aparência do Jumper com os bitmaps.

Capítulo 11

Adicionando som ao jogo

Nosso jogo já está bastante divertido, com todo o necessário para desafiar os jogadores. Porém falta uma parte crucial de um jogo: sua música. Pense em todos os principais jogos do mercado, a maioria deles tem uma trilha sonora ou até mesmo um som marcante que cativa o jogador.

No Jumper não vamos fazer todo o trabalho altamente complexo de um engenheiro de som, mas vamos ter sons para os principais momentos do jogo, como o pulo do pássaro, a colisão e aumento da pontuação!

Para colocarmos um som em uma aplicação Android, a primeira solução é utilizar a classe `MediaPlayer` do próprio Android. Essa classe é interessante quando precisamos criar formas mais elaboradas de interação com sons. Pense em um aplicativo de música: nele temos os controles para *play*, *stop*, *pause*, *avançar* etc. Entretanto, em um jogo não precisamos de todos esses controles; na realidade só queremos tocar um som de curta duração e, no máximo, uma trilha sonora constante ao fundo.

Tendo em mente essa estrutura mais simples, temos a classe `SoundPool`, cujo objetivo é justamente fornecer uma forma mais leve para tocar sons de curta duração ideal para jogos!

Ao instanciar um `SoundPool` podemos passar, no seu construtor, o tipo de som que será tocado, além de quantos arquivos simultâneos poderão ser tocados. Como temos três sons possíveis, setaremos esse valor para `3`:

```
SoundPool pool = new SoundPool(3, AudioManager.STREAM_MUSIC, 0);
```

Vamos criar uma classe responsável por gerenciar esse `SoundPool`. No pacote **engine**, criaremos a classe `Som`:

```
public class Som {

    private SoundPool soundPool;

    public Som() {
      soundPool = new SoundPool(3, AudioManager.STREAM_MUSIC, 0);
    }
}
```

Para tocarmos algum som, antes precisaremos carregá-lo com o método `load`. Nele, passaremos o arquivo de som, que deverá estar na pasta **res/raw**. Se o carregamento for feito com sucesso, o `load` retornará um *id* que deverá ser usado sempre que quisermos tocá-lo.

```
public class Som {

    private SoundPool soundPool;
    public static int PULO;

    public Som(Context context) {
      soundPool = new SoundPool(3, AudioManager.STREAM_MUSIC, 0);
      PULO = soundPool.load(context, R.raw.pulo, 1);
    }
}
```

Agora que temos o som `PULO` carregado, vamos tocá-lo com o método `play`, que recebe o *id* do som, o volume esquerdo, volume direito e, por último, a velocidade do som (Lembre-se: 1 é a velocidade normal).

```
soundPool.play(idDoSom, 1, 1, 1, 0, 1);
```

Na nossa classe `Som`, teremos o seguinte código:

```
public class Som {

    private SoundPool soundPool;
    public static int PULO;

    public Som(Context context) {
      soundPool = new SoundPool(3, AudioManager.STREAM_MUSIC, 0);
      PULO = soundPool.load(context, R.raw.pulo, 1);
    }

    public void play(int som) {
        soundPool.play(som, 1, 1, 1, 0, 1);
    }

}
```

Em qual momento tocaremos o som do pulo do pássaro? Quando o `Passaro` pular. Porém, em qual lugar do nosso código temos a lógica do pulo? No método `pula` da classe `Passaro`!

```
public class Passaro {

    public void pula() {
        if(altura > RAIO) {
            som.play(Som.PULO);
            altura -= 150;
        }
    }
}
```

Agora, basta receber o `Som` no construtor do `Passaro`:

```
private Som som;

public Passaro(Tela tela, Context context, Som som) {
    this.som = som;
}
```

Como mudamos o construtor do `Passaro`, quebramos a classe `Game`, que instancia o `Passaro`. Na classe `Game`, vamos instanciar o `Som` e passar para o `Passaro`:

```
public class Game extends SurfaceView implements Runnable,
OnTouchListener{

    //Outras variáveis...
    private Som som;

    public Game(Context context) {
        //...
        this.som = new Som(context);
    }


    private void inicializaElementos() {
        this.passaro = new Passaro(tela, context, som);
        //...
    }

    //Outros métodos...
}
```

Com isso, nosso `Passaro` emitirá um som a cada pulo!