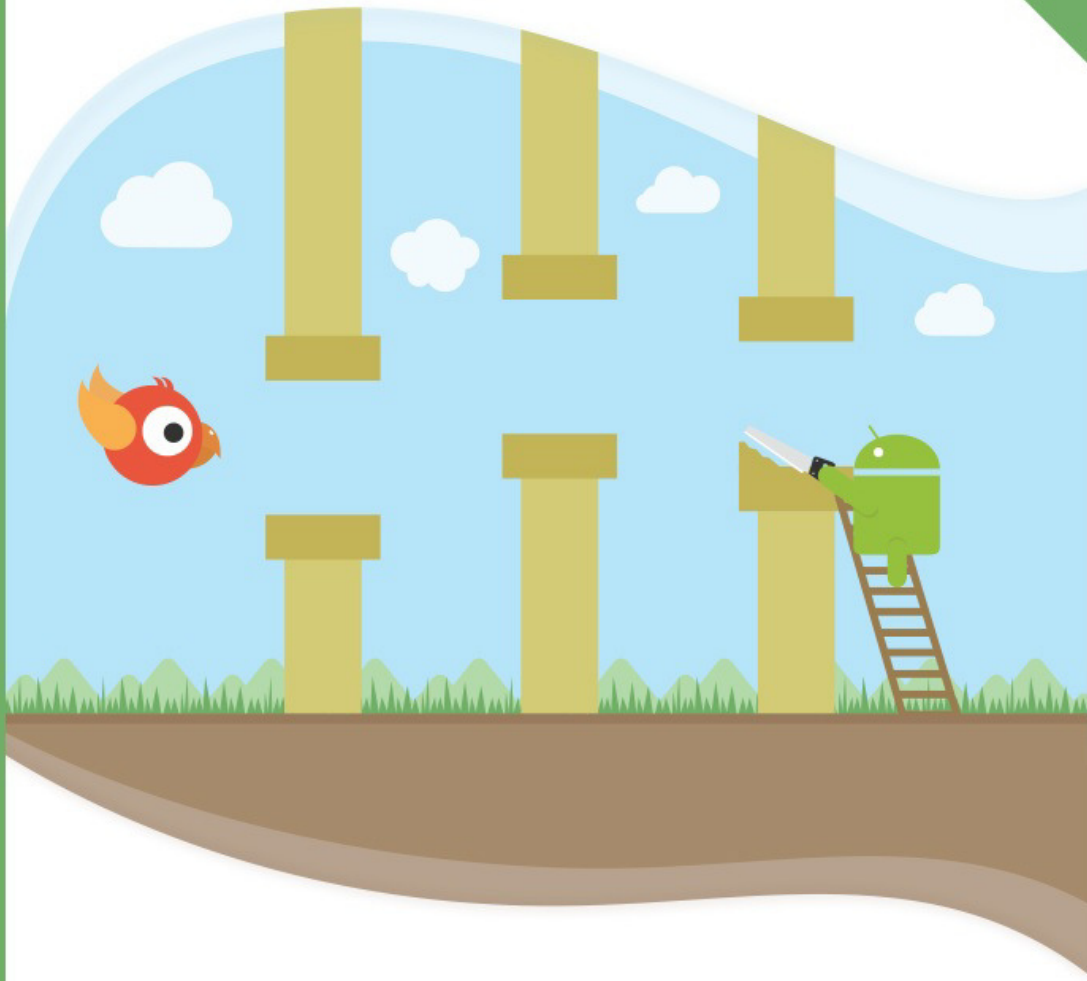
Jogos Android

Criando um game do zero usando classes nativas



Casa do Código

SÉRIE CAELUM

FELIPE TORRES

Edição

Adriano Almeida
Vivian Matsui

Revisão

Bianca Hubert
Vivian Matsui

[2017]

Casa do Código

Livros para o programador

Rua Vergueiro, 3185 - 8º andar

04101-300 – Vila Mariana – São Paulo – SP – Brasil

www.casadocodigo.com.br

SOBRE O GRUPO CAELUM

Este livro possui a curadoria da Casa do Código e foi estruturado e criado com todo o carinho para que você possa aprender algo novo e acrescentar conhecimentos ao seu portfólio e à sua carreira.

A Casa do Código faz parte do Grupo Caelum, um grupo focado na educação e ensino de tecnologia, design e negócios.

Se você gosta de aprender, convidamos você a conhecer a Alura (www.alura.com.br), que é o braço de cursos online do Grupo. Acesse o site deles e veja as centenas de cursos disponíveis para você fazer da sua casa também, no seu computador. Muitos instrutores da Alura são também autores aqui da Casa do Código.

O mesmo vale para os cursos da Caelum (www.caelum.com.br), que é o lado de cursos presenciais, onde você pode aprender junto dos instrutores em tempo real e usando toda a infraestrutura fornecida pela empresa. Veja também as opções disponíveis lá.

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. Assim, 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.

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 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 usaremos 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.

Como este livro está organizado?

Criar um jogo do zero é sempre algo bastante desafiador e, muitas vezes, nos vemos perdidos sem saber por onde começar, quais problemas atacar, qual a melhor prática etc. Este livro está organizado de uma forma que os desafios vão aparecendo naturalmente durante o desenvolvimento do nosso programa. Cada capítulo está focado na solução de um problema e, ao resolvê-lo, encontramos a motivação para o próximo desafio.

Apesar da teoria por trás de um jogo, a todo momento mostramos os trechos de código, dando uma atenção à programação e boas práticas!

Depois das apresentações sobre o autor e o projeto, no capítulo

Começando o Jumper criaremos o começo do jogo e discutiremos por onde devemos começá-lo. Com isso, teremos o *loop* principal, que será onde as principais funcionalidades do jogo serão gerenciadas. Porém, um jogo sem nada para mostrar é bastante chato, então criaremos nosso primeiro elemento: o pássaro, e já aproveitamos para definir seu comportamento. Ao fim desse capítulo, teremos um jogo com um pássaro caindo!

No capítulo *Colocando uma imagem de fundo*, colocaremos uma imagem para servir como plano de fundo e veremos como redimensioná-lo para caber em qualquer tela. Com isso, teremos um pássaro caindo, mas com um fundo bonito. Para melhorar isso, faremos o controle do jogador: ao tocar na tela, o pássaro pula.

Finalizado o pássaro, nosso jogo não está nada desafiador para o jogador. Resolveremos isso no capítulo *Cano inferior*, onde criaremos o outro elemento do jogo: o cano inferior! Ao exibi-lo na tela, veremos que ele estará imóvel. Ao término deste capítulo, faremos esse cano se mover na direção do pássaro.

Nosso jogo já está ficando mais interessante, mas ainda conta com apenas um cano inferior solitário. No capítulo *Criando vários canos*, nosso objetivo será criar vários canos inferiores!

Já no capítulo *Canos superiores*, criaremos os canos superiores! Além disso, tornaremos esses canos "infinitos" para o jogador. Mas, no nosso código, teremos apenas um número fixo deles.

Terminada a criação dos elementos do nosso jogo, vamos focar na jogabilidade. Até aqui, quando passamos por algum cano, nada acontece. Este será o momento de criarmos a pontuação! No capítulo *Pontuação*, veremos como será contabilizada a pontuação

do jogador e mais a fundo a classe `Paint` do Android, para exibirmos esse valor na tela!

No capítulo *Colisões*, percebemos que o jogo ficou muito fácil, pois em nenhum momento dissemos o que vai acontecer quando o pássaro bater em um cano. Focaremos, então, em definir o que será a colisão entre o pássaro e o cano. Por fim, discutiremos o que fazer caso haja uma colisão e criaremos o *game over*. Com isso, concluiremos a mecânica do jogo, com todos os detalhes para termos algo jogável em mãos.

Até aqui, nosso jogo está feio, pois sempre trabalhamos com formas geométricas simples, como círculos (para o pássaro) e retângulos (para os canos). Para melhorarmos sua estética, veremos no capítulo *Aprimorando o layout do jogo* como substituir essas formas por imagens sem que tenhamos de refazer toda a lógica da nossa aplicação.

No capítulo *Som*, nossa atenção será dedicada a um outro aspecto bastante importante em um jogo: os sons! Veremos como podemos usar a classe `SoundPool` para tocar um som para os principais momentos do jogo: pulo do pássaro, colisão e aumento na pontuação.

O capítulo *Física* possui um conteúdo um pouco mais matemático para tornar nosso jogo um pouco mais realista. Neste momento, você verá que, com um pouco de conhecimento em física, já é possível tornar a jogabilidade mais interessante.

Após isso, no capítulo *Menu principal* completamos o Jumper criando uma nova tela para servir como menu principal.

Todos os arquivos usados no Jumper estão disponíveis no GitHub, em https://github.com/felipetorres/jumper-arquivos.

Além disso, caso apareça alguma dúvida ou queira compartilhar alguma ideia, não deixe de participar do grupo de discussões deste livro, em http://forum.casadocodigo.com.br.

Caso você deseje submeter alguma errata ou sugestão, acesse http://erratas.casadocodigo.com.br

Sumário

Versão: 20.8.24

CAPÍTULO 1

COMEÇANDO O JUMPER

1.1 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* :

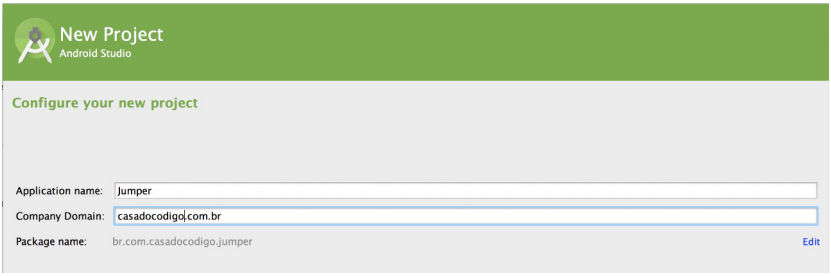


Figura 1.1: Tela de criação do projeto no Android Studio

Logo em seguida, diremos ao assistente como queremos que nosso projeto seja criado. Nesta tela, podemos escolher entre um projeto sem nenhuma tela pré-configurada, com uma tela configurada para exibir um mapa. Temos muitas possibilidades. Vamos escolher a opção *Empty Activity* para criarmos uma tela vazia!

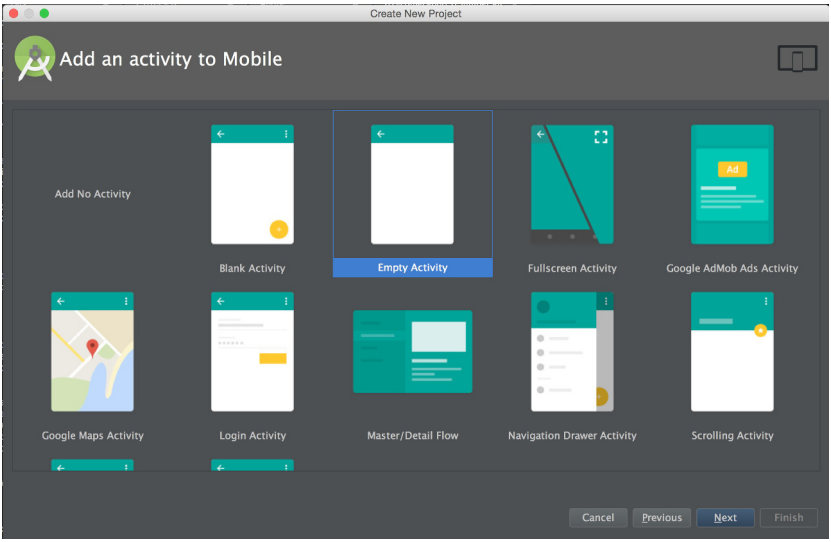


Figura 1.2: Definindo o tipo de Activity a ser criado

Agora, precisamos definir o nome dessa tela e seu layout. Vamos deixar como o sugerido pelo Android Studio:

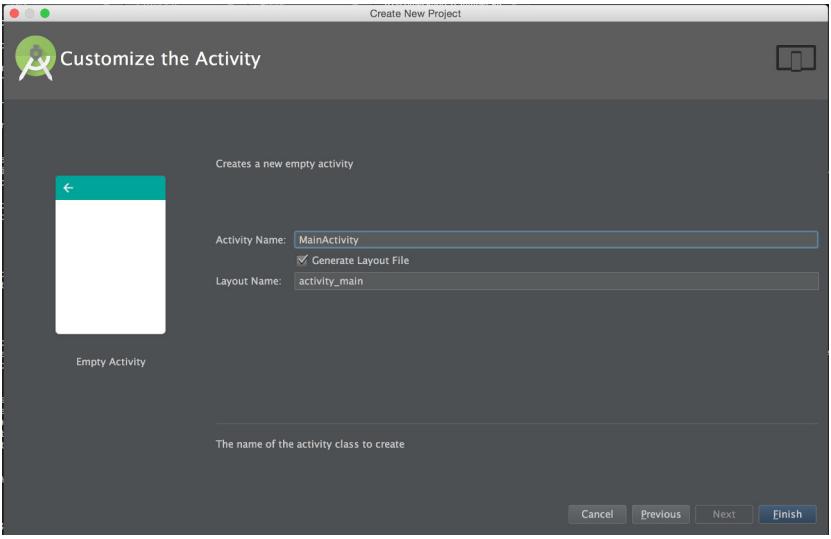


Figura 1.3: Nome da Activity e seu layout

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 de 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`. Só precisamos fazer uma pequena alteração na `MainActivity` criada: quando o assistente do Android Studio criou o projeto, nossa `MainActivity` é filha de `AppCompatActivity`. Vamos deixar nosso código mais simples, tornando nossa classe filha de `Activity`:

```
public class MainActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
}
```

> A classe `AppCompatActivity` tem como objetivo disponibilizar nas versões mais antigas do Android algumas funcionalidades presentes apenas nas versões mais atuais do sistema. Como nossa aplicação não usará funcionalidades de alguma versão específica do Android, podemos usar a classe `Activity` sem problemas de compatibilidade.

Olhando o arquivo `activity_main.xml`, vamos alterá-lo

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`:

```
public class MainActivity extends Activity {

    private Game game;

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

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

        this.game = new Game(this);
        container.addView(this.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:

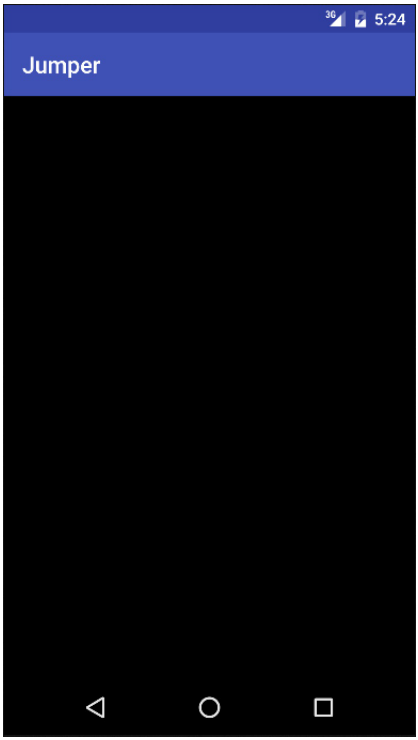


Figura 1.4: Nossa SurfaceView até o momento

Uma decisão importante que temos de 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 usar a propriedade `android:screenOrientation` e defini-la como `portrait`:

```
<activity
    android:name=".MainActivity"
    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 de alterar esse tema para:

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

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

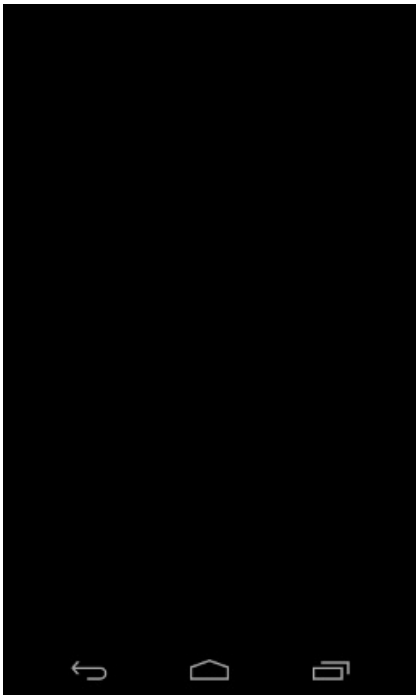


Figura 1.5: SurfaceView em tela cheia

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

1.2 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 seus elementos, 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 estaRodando = true;

    @Override
    public void run() {
        while(this.estaRodando) {
            //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, consumindo, desnecessariamente, recursos do aparelho. 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();
        this.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.estaRodando = 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();
        this.game.inicia();
        new Thread(this.game).start();
    }
}
```

O método `inicia` fica assim:

```
public class Game extends SurfaceView implements Runnable {

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

Com o loop principal funcionando, vamos criar os elementos do jogo?

1.3 COMO DESENHAR ELEMENTOS NO SURFACEVIEW?

Para desenhar algo em um `SurfaceView`, é necessário ter acesso ao objeto responsável por renderizar elementos na tela: o *canvas*. Com ele, 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 de 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 é desenhado em segundo plano. Quando o novo quadro está pronto, o Android só troca o velho pelo novo. A única coisa que temos de 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(this.estaRodando) {
            Canvas canvas = this.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(this.estaRodando) {
            Canvas canvas = this.holder.lockCanvas();

            //Aqui vamos desenhar os elementos do jogo!

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

Ao chamar o `getHolder` , precisamos garantir que temos acesso ao `SurfaceHolder` **antes** de começarmos a desenhar. Do contrário, tomaremos 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(this.estaRodando) {
            if(!this.holder.getSurface().isValid()) continue;

            Canvas canvas = this.holder.lockCanvas();

            //Aqui vamos desenhar os elementos do jogo!

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

1.4 NOSSO PRIMEIRO ELEMENTO: A CLASSE PASSARO

O principal personagem do nosso jogo será um pássaro! Então, no pacote `br.com.casadocodigo.jumper.elementos`, criaremos uma classe `Passaro`, que será responsável por todas as suas propriedades e ações. Para que possamos exibir nosso pássaro na tela, inicialmente o representaremos por um círculo vermelho.

Em um `Canvas`, temos uma variedade de métodos disponíveis para alguns desenhos simples. Para desenhar um círculo, usamos o método `drawCircle`, que recebe quatro argumentos:

- Coordenada X (horizontal) do centro do círculo;
- Coordenada Y (vertical) do centro do círculo;
- Raio do círculo;
- Cor de preenchimento.

```
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, this.altura, RAIO, ??);
    }
}
```

No pacote `br.com.casadocodigo.jumper.engine`, vamos criar uma classe chamada `Cores` para gerenciar e centralizar

todas as cores que usaremos no preenchimento dos elementos do nosso jogo. Nesta classe, criaremos um método `getCorDoPassaro` que retornará a cor vermelha usando a classe nativa do Android chamada `Paint`:

```
public class Cores {

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

ARGB

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 (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, this.altura, RAIO, VERMELHO);
    }
}
```

Terminado 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(this.estaRodando) {
            if(!this.holder.getSurface().isValid()) continue;

            Canvas canvas = this.holder.lockCanvas();

            this.passaro.desenhaNo(canvas);

            this.holder.unlockCanvasAndPost(canvas);
        }

    }
}
```

Nossa classe `Game` ainda está com um pequeno problema: neste momento, ao rodar o método `desenhaNo`, causaremos um `NullPointerException`, pois nosso `this.passaro` ainda não foi instanciado, ou seja, está nulo. Para resolver esse problema, onde poderemos instanciar um novo `Passaro`?

Temos de tomar bastante cuidado onde daremos `new` nos elementos do nosso jogo. Será que precisaremos de um novo

objeto `Passaro` a cada loop? **Não.**

Caso instanciemos o `Passaro` dentro do loop principal, criaremos muitos objetos na memória do aparelho, causando uma grande lentidão no nosso jogo! Então, vamos instanciar o `Passaro` uma única vez e manipularemos sempre a **mesma** instância de `Passaro` !

Para isso, 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();
    }
}
```

Com isso, ao rodar o Jumper, teremos a seguinte tela:

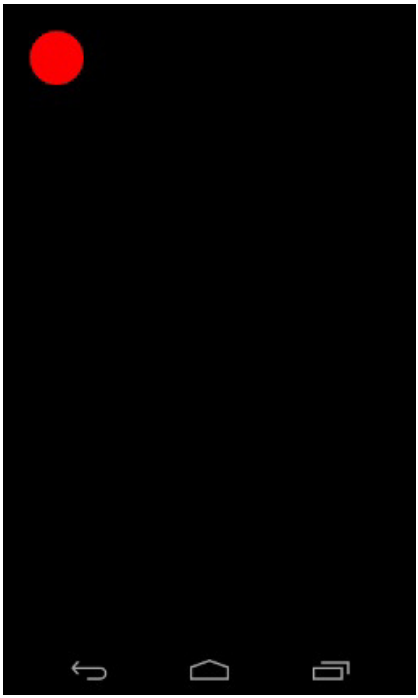


Figura 1.6: Nosso pássaro na tela do jogo

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

Vamos implementar isso?

1.5 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(this.estaRodando) {
            if(!this.holder.getSurface().isValid()) continue;

            Canvas canvas = this.holder.lockCanvas();

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

            this.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;
    }
}
```

Coordenadas em um canvas

Quando pensamos em queda, geralmente tentaríamos subtrair o valor da `altura` do nosso `Passaro`. No entanto, quando estamos em um `canvas`, o canto superior esquerdo é o ponto com `X = 0` e `Y = 0`. Logo, quanto mais descemos na nossa tela, maior o valor de `Y` (maior a `altura`). Sendo assim, se queremos descer `5` pixels, devemos somar `5` à altura atual.

Isso é o suficiente para derrubarmos o pássaro a cada iteração do nosso loop principal. Porém, ao implementarmos o método `cai`, criamos um rastro indesejado. Que tal resolvermos isso?

Capítulo 2

COLOCANDO UMA IMAGEM DE FUNDO

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 são 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 (o famoso *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 (e vice-versa), ele **não** limpa o conteúdo anteriormente desenhado! É exatamente esse conteúdo "desatualizado" que vemos. Perceba que o rastro do círculo é formado pelas suas posições anteriormente desenhadas:

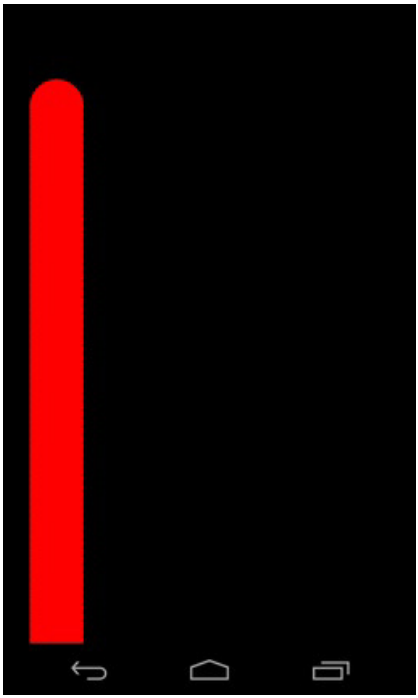


Figura 2.1: Rastro da bolinha ao cair

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

2.1 COMO SERÁ O BACKGROUND?

Arquivos usados no jogo

Lembre-se de que os arquivos que usaremos ao longo deste 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` . Caso essa pasta não exista no seu projeto, não se preocupe: basta criá-la no diretório `res` .

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

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

Porém, onde esse código ficaria? Como estamos dizendo ao `Game` qual será seu background, vamos deixar essa linha dentro do método `inicializaElementos` :

```
public class Game extends SurfaceView implements Runnable {

    private Bitmap background;

    private void inicializaElementos() {
        this.background =
            BitmapFactory.decodeResource(getResources(),
                                         R.drawable.background);
        //Inicialização anterior do pássaro...
    }
}
```

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 de ser a primeira coisa a ser desenhada na tela.

```
public class Game extends SurfaceView implements Runnable {

    //Códigos anteriores...

    @Override
    public void run() {
        while(this.estaRodando) {
            if(!this.holder.getSurface().isValid()) continue;
            Canvas canvas = this.holder.lockCanvas();

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

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

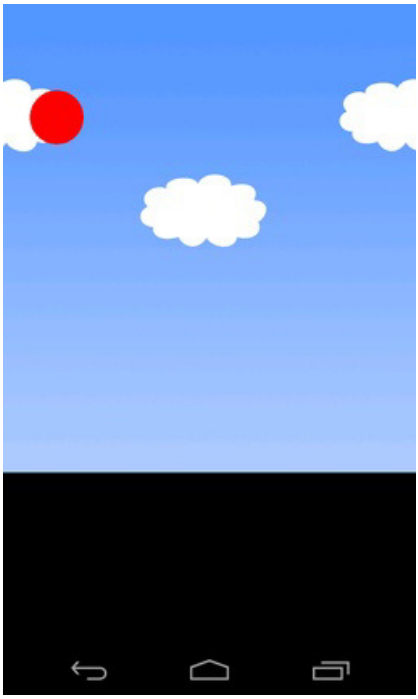


Figura 2.2: Background adicionado ao jogo

Esse fundo ficou meio estranho, não? Vamos garantir que toda a nossa tela fique preenchida por essa imagem!

2.2 REDIMENSIONANDO O PLANO DE FUNDO

Precisamos redimensionar nosso background para que ele ocupe todo o espaço vertical da tela. Porém, para definirmos a altura dessa imagem, devemos saber a altura da tela. Vamos usar 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`, no pacote `br.com.casadocodigo.jumper.engine`, 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();
        this.metrics = new DisplayMetrics();
        display.getMetrics(this.metrics);
    }

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

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

Como devemos exibir o background redimensionado, vamos fazer uma pequena alteração: vamos chamar de `back` a versão do `bitmap` antes de ser redimensionada, e a variável `background` será responsável agora por armazenar o `bitmap` já redimensionado, como no código a seguir:

```
public class Game extends SurfaceView implements Runnable {

  private Bitmap background;

  private void inicializaElementos() {

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

    //Inicialização do pássaro...
  }
}
```

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

No entanto, ainda temos um erro: não sabemos onde temos de inicializar essa variável `tela` para sabermos sua altura.

Vamos inicializá-la no próprio construtor da classe `Game`!

```
public class Game extends SurfaceView implements Runnable {

  private Bitmap background;
  private Tela tela;

  public Game(Context context) {
    super(context);
    this.tela = new Tela(context);
    inicializaElementos();
  }
```

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

    //Inicialização do pássaro...
  }
}
```

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

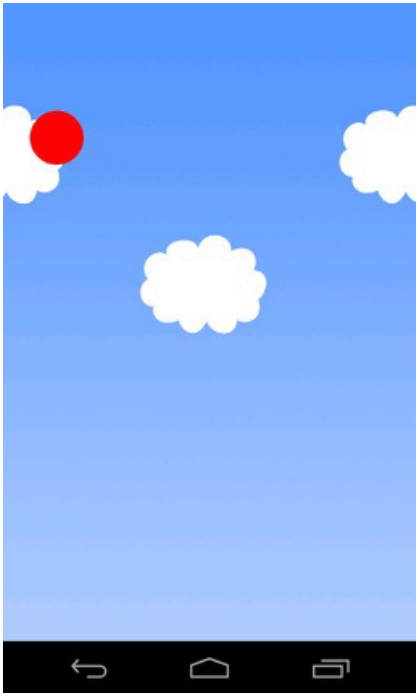


Figura 2.3: Background ocupando a tela inteira

2.3 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;
    }
}
```

RELEMBRANDO: COORDENADAS EM UM CANVAS

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 de 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,
                                        View.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,
                                        View.OnTouchListener {

    @Override
    public boolean onTouch(View v, MotionEvent event) {
        this.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*, já que 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,
                                        View.OnTouchListener {

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

```
        setOnTouchListener(this);
    }
}
```

Com isso, finalizamos o controle do jogador! Contudo, ainda precisamos criar alguns outros elementos neste jogo, para que ele fique mais desafiador.

Capítulo 3

CANO INFERIOR

Além do pássaro, o Jumper também possui outro tipo de elemento a ser desenhado na tela: os canos!

Como aproximamos inicialmente o pássaro para uma bolinha, vamos aproximar os canos para retângulos.

3.1 CRIANDO A CLASSE CANO

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

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

    @Override
    public void run() {
        while(this.estaRodando) {

            //lockCanvas e desenhos anteriores…

            this.cano.desenhaNo(canvas);

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

Como ainda não temos a classe `Cano` criada, nem o método

`desenhaNo` , o código acima não compilará. Vamos resolver isso agora?

No pacote `br.com.casadocodigo.jumper.elementos` , criaremos 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, podemos usar os métodos de desenho da própria classe `Canvas` . Neste caso, chamaremos o método `drawRect` .

```
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(??, ??, ??, this.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;
```

Juntando esses trechos de código, teremos uma classe `Cano` como mostrada a seguir. É verdade que ela ainda não compila, mas já está mais completa que antes:

```
public class Cano {

    private static final int TAMANHO_DO_CANO = 250;
    private static final int LARGURA_DO_CANO = 100;
    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(??,
                        this.alturaDoCanoInferior,
                        ??,
                        this.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 de desenhar esse cano de tal forma que sua posição horizontal possa variar, pois afinal de contas, ele deverá se mover horizontalmente para a direção do pássaro. Vamos receber essa posição no construtor do `Cano` e atribuí-la ao nosso atributo `posicao`:

```
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(this.posicao,
                        this.alturaDoCanoInferior,
                        ??,
                        this.tela.getAltura(),
                        ??);
    }
}
```

Agora, 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`!

Então, temos que o valor da lateral direita será `posicao +`

LARGURA_DO_CANO ! É isso que passaremos no terceiro argumento do método drawRect :

```
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(this.posicao,
                        this.alturaDoCanoInferior,
                        this.posicao + LARGURA_DO_CANO,
                        this.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 {

        //getCorDoPassaro...

        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 `FF00FF00`.

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

```
public class Cano {

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

    //Códigos anteriores...

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

Só precisamos instanciar esse `Cano` no método `inicializaElementos` da nossa classe `Game`, dizendo qual será sua posição inicial:

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

    private Cano cano;

    private void inicializaElementos() {
        //Outros elementos já inicializados aqui...

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

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

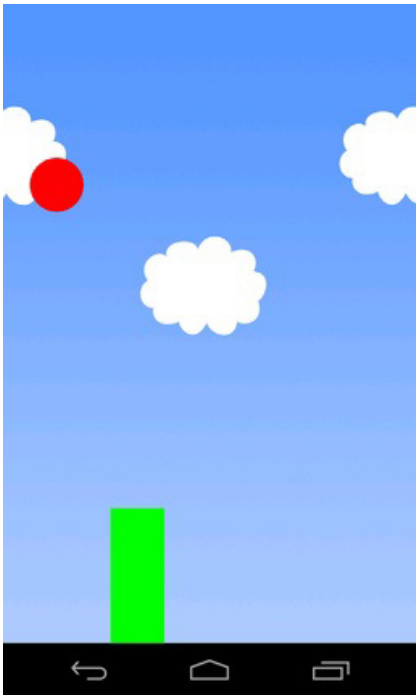


Figura 3.1: Jumper com um cano inferior

3.2 A MOVIMENTAÇÃO DO CANO

No Jumper, os canos devem se mover horizontalmente na direção do pássaro. Como 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,
                                        View.OnTouchListener {
```

```
    @Override
    public void run() {
        while(this.estaRodando) {
            //lockCanvas…
            //Desenho do pássaro…

            this.cano.desenhaNo(canvas);
            this.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 {
        //Outros métodos e variáveis...

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

Rode o jogo e veja nosso primeiro cano se movimentar para a esquerda! Até o momento, já temos um pássaro caindo, o cano se movimentando corretamente e o controle do jogador já implementado.

Um jogo com apenas um cano não é dos mais desafiadores, então, que tal colocarmos mais?

Capítulo 4

CRIANDO VÁRIOS CANOS

Até agora, nosso jogo conta com o inimigo, um plano de fundo e um cano. Parece pouco, mas já aprendemos um monte de conceitos do mundo de jogos, e manipulamos classes nativas do Android que podem ser usadas no nosso dia a dia em outras aplicações também!

Já que sabemos como criar e movimentar um único cano na nossa tela, o que será preciso para criarmos e movimentarmos vários canos?

4.1 MELHORIAS PARA GERENCIAR VÁRIOS CANOS

Em uma primeira tentativa, poderíamos fazer um código como o a seguir, distanciando os canos em 250 *pixels* :

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

O código parece feio, uma vez que o estamos copiando e colando por todo lado. Que tal colocar os canos em um laço? Já parece melhor! Vamos aproveitar e criar uma variável `posicao`

para contabilizar a distância de 250 *pixels* entre cada cano:

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

Já temos um código mais bonito, mas ainda não conseguimos desenhar esses canos na nossa tela. Como nosso cano possui o método `desenhaNo`, poderíamos chamar esse método agora, fazendo o seguinte código:

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

Veja que, com este código, sempre instanciamos um cano novo a cada vez que vamos desenhar no canvas. Nos capítulos anteriores, vimos que instanciar muitos elementos no nosso jogo e os manter na memória pode causar problemas sérios de lentidão. Vamos, então, usar a mesma ideia e trabalhar apenas com um conjunto fixo de canos na nossa memória.

Para isso, precisaremos primeiramente guardar os canos em algum lugar para depois desenhá-los na nossa tela. Uma `List` de `Cano`s parece uma boa ideia para gerenciar esses canos:

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

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

Pronto, temos cinco canos em nossas mãos. Só precisamos de uma forma para desenhá-los.

Poderíamos colocar a lista como variável membro da nossa classe `Game` e fazer um `for` lá no método `run` para desenhá-los, mas calma lá: nosso método da classe `Game` já tem muitas responsabilidades.

Nossa classe `Game` já está responsável por inicializar os elementos do jogo e pelo loop principal. Porém, apesar de toda essa responsabilidade, nossa classe `Game` apenas **chama os métodos** de outras classes do nosso jogo quando é necessário, o `Game` não sabe a lógica de desenhar um pássaro na tela ou de mover um cano. Vamos usar essa mesma ideia para gerenciarmos vários canos no jogo: isolaremos o seu comportamento em uma classe!

Agora, no pacote `br.com.casadocodigo.jumper.elementos`, criaremos uma classe chamada `Canos` para isolar o gerenciamento de vários canos:

```
public class Canos {

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

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

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

É bastante comum lidarmos com números para representar aspectos específicos de um jogo, como fase, valor da gravidade, volume etc. Durante o desenvolvimento do código, colocamos o valor, mas não dizemos o que significa aquele número, pois, afinal de contas, nós sabemos o que esse número representa na nossa cabeça.

No entanto, já no dia seguinte, podemos nos esquecer do significado daquele inocente número. A partir desse momento, esse valor se torna um *número mágico* no nosso código: um valor que não lembramos o seu significado, mas que deixa tudo funcionando como o esperado. Para evitar esquecermos os significados dos números `5` e `250` na nossa classe `Canos`, vamos refatorá-la e extrair algumas variáveis:

```
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 posicao = 200;

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

Agora sim, dentro do nosso `Game` a inicialização dos elementos fica mais simples: basta dizer que queremos criar os canos. Removeremos o único cano que tínhamos e criaremos todos eles de uma única vez, deixando nosso método `inicializaElementos` com essa cara:

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

  private Canos canos;

  //Outros métodos e atributos...

  private void inicializaElementos() {
    Bitmap back = BitmapFactory.decodeResource(getResources(),
                    R.drawable.background);

    this.background = Bitmap.createScaledBitmap(back,
                back.getWidth(), this.tela.getAltura(), false);

    this.passaro = new Passaro();

    //Em vez de criarmos um cano, agora criamos vários...
    this.canos = new Canos(this.tela);
  }
}
```

Temos como criar cinco canos, mas ainda não conseguimos desenhá-los, nem movê-los na tela do jogo. Anteriormente, no método `run` para um único cano fazíamos:

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

Seria bem legal se pudéssemos mover todos os cinco canos com a mesma facilidade com que movíamos um único. Será que é possível isso? Claro! Basta criarmos a lógica de desenhar e mover vários canos lá na classe `Canos`! Como isolamos as responsabilidades, fica fácil adicionar novos comportamentos.

Na classe `Canos`, criaremos o método `desenhaNo` e o método `move`, que farão um *loop* pelos canos da nossa `ArrayList`:

```
public class Canos {
```

```
    //...

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

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

Agora, voltando ao método `run` da classe `Game`, podemos **trocar** as linhas que desenhavam e moviam um único cano:

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

Pelas chamadas aos métodos que desenham e movem os cinco canos (veja que a variável está no plural para indicar os vários canos):

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

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

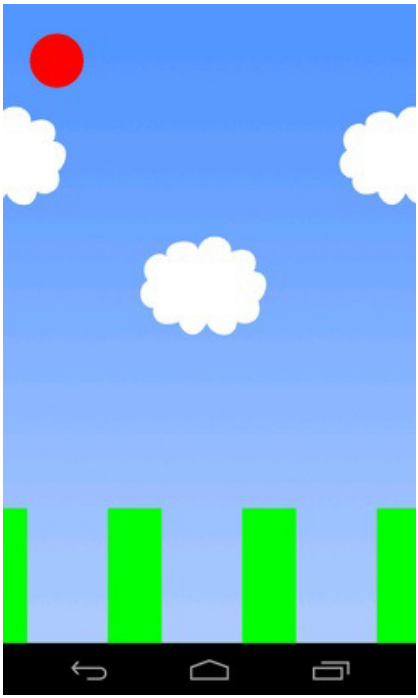


Figura 4.1: Nosso jogo com os canos inferiores

4.2 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 class Passaro {

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

    if(!chegouNoChao) {
```

```
      this.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 de verificar quando a borda do nosso círculo toca o seu topo. Como ele 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 de que quanto maior a altura, mais para baixo da tela o elemento está (o ponto `(0,0)` é o canto **superior** esquerdo!).

Nosso método `pula`, para verificar se chegamos ao topo da

tela, deve ficar assim:

```
public class Passaro {

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

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

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

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

Capítulo 5

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.

5.1 CALCULANDO OS CANOS SUPERIORES

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 zero (ponto inicial da tela) mais o tamanho do cano:

```
public class Cano {

    private int alturaDoCanoSuperior;

    public Cano(Tela tela, int posicao) {
        //códigos anteriores...

        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. Na classe `Cano`, vamos aleatorizar um pouco essas dimensões pegando um número entre `0` e `150` para construir canos com alturas diferentes:

```
public class Cano {

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

Agora, podemos usar esse método para o cálculo das alturas dos canos inferiores e superiores:

```
public class Cano {

    private int alturaDoCanoSuperior;

    public Cano(Tela tela, int posicao) {
        //Códigos anteriores...

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

        this.alturaDoCanoSuperior =
            0 + TAMANHO_DO_CANO + valorAleatorio();
    }

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

Temos a altura do cano superior, mas ainda não somos capazes de desenhá-lo! Vamos usar a mesma ideia que tivemos para o desenho do cano inferior: na classe `Cano`, podemos criar um

método `desenhaCanoSuperiorNo` para ser responsável **apenas** pelo desenho do cano superior!

```
public class Cano {

    private void desenhaCanoSuperiorNo(Canvas canvas) {
        canvas.drawRect(??, ??, ??, ??, 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, ??, this.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 canos! Para nossa sorte, já sabemos como obter essa largura por meio da conta `this.posicao + LARGURA_DO_CANO`, e o ponto de

início na variável chamada `posicao` ! É só passar esses valores para o primeiro e terceiro argumentos do `drawRect` .

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

Juntando tudo, nosso método `desenhaCanoSuperiorNo` ficará assim:

```
public class Cano {

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

Já que temos esse método pronto, como faremos para chamá-lo? Podemos complementar o método `desenhaNo` , na própria classe `Cano` , para desenhar também os canos superiores!

```
public class Cano {

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

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

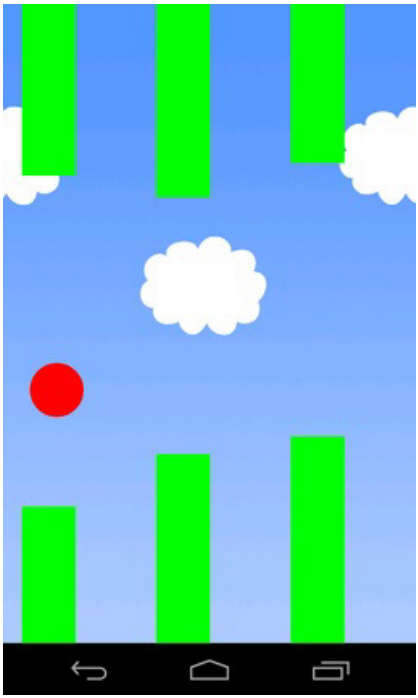


Figura 5.1: Canos com tamanhos diferentes

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?

5.2 GERENCIANDO INFINITOS CANOS

Uma tática em jogos infinitos é fazer com que, quando o cano saia da tela, um novo seja colocado no fim da lista! Dessa maneira, sempre exibiremos um cano para o jogador. Só precisamos saber onde faremos a verificação de que um cano saiu da tela!

Vamos olhar o nosso método `move` da classe `Canos`:

```
public class Canos {

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

Logo após mover um cano, podemos verificar se ele já saiu da tela:

```
public class Canos {

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

Caso ele tenha saído, podemos criar outro `Cano` e adicioná-lo ao nosso `ArrayList<Cano>`:

```
public class Canos {

    public void move() {
        for(Cano cano : this.canos) {
            cano.move();
            if(cano.saiuDaTela()) {
                Cano outroCano = new Cano(??, ??);
                this.canos.add(outroCano);
            }
        }
    }
}
```

Temos algumas coisas para completar esse código. Vamos lá?

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

```
public class Cano {

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

Neste código, dissemos que se a sua **lateral esquerda** sair da tela o cano inteiro saiu da tela. Na realidade, para o cano **sair totalmente** da tela, sua lateral **direita** tem de sair da tela, ou seja, toda largura do cano tem de passar por sua posição `0`. Então, podemos melhorar o método `saiuDaTela`:

```
public class Cano {

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

Perfeito! Resolvemos um problema. Agora, podemos olhar os argumentos que passaremos para essa nova instância da classe `Cano`:

```
    Cano outroCano = new Cano(??,??);
```

O primeiro argumento é uma instância da classe `Tela`. Como já recebemos no próprio construtor da classe `Canos` uma `Tela`, podemos passar para esse `Cano`:

```
public class Canos {

    private Tela tela;

    public Canos(Tela tela) {
```

```
        //...
        this.tela = tela;
    }

    //Outros métodos...

    public void move() {
        for (Cano cano : this.canos) {
            cano.move();
            if (cano.saiuDaTela()) {
                Cano outroCano = new Cano(this.tela, ??);
                this.canos.add(outroCano);
            }
        }
    }
}
```

Lembre-se de que o segundo argumento que devemos passar para o construtor da classe `Cano` é justamente sua posição inicial. Nossa ideia é posicionar esse cano sempre após os canos já existentes. Então, para determinar a `posicao` do novo cano, de alguma forma precisaremos saber qual é o cano que está mais à direita possível (ou seja, aquele que possui a maior `posicao`).

Para isso, basta percorrer nossa lista de canos e comparar suas posições:

```
public class Canos {

    private int maiorPosicao() {
        int maximo = 0;
        for(Cano cano : this.canos) {
            maximo = Math.max(cano.getPosicao(), maximo);
        }
        return maximo;
    }
}
```

Como fazemos `cano.getPosicao`, não se esqueça de fazer o `getPosicao` na classe `Cano`:

```
public class Cano {

    public int getPosicao() {
        return this.posicao;
    }
}
```

Com esses dois métodos prontos, podemos agora completar nosso método `move` da classe `Canos` ! Como já sabemos como obter a maior posição de um cano, usaremos esse valor e colocaremos o novo cano com a mesma distância dos outros:

```
public class Canos {

    private Tela tela;

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

    //Outros métodos...

    public void move() {
        for (Cano cano : this.canos) {
            cano.move();
            if (cano.saiuDaTela()) {

                Cano outroCano = new Cano(this.tela,
                    maiorPosicao() + DISTANCIA_ENTRE_CANOS);

                this.canos.add(outroCano);
            }
        }
    }
}
```

Vamos fazer um teste: rode o código que já temos e veja se aparece algum problema.

5.3 ITERATOR PARA MODIFICAR A LISTA DE CANOS

No método `move` da classe `Canos`, 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, basta usarmos um `ListIterator`, que toda `List` possui. Portanto, em vez de um `for`, vamos alterá-lo para um `while`:

```
public class Canos {

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

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

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

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?

5.4 DESCARTANDO CANOS ANTERIORES

Conforme o tempo passa (e os canos saem da tela), vamos criando outros `Cano`s e adicionando-os à lista. Veja que ela está ficando cada vez maior e 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:

```
public class Canos {

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

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

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

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

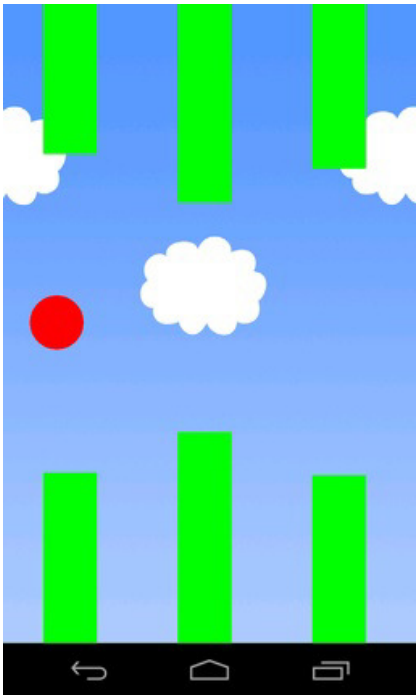


Figura 5.2: Agora podemos gerar infinitos canos

CAPÍTULO 6

PONTUAÇÃO

Agora que já conseguimos criar infinitos canos e somos capazes de movê-los pela nossa tela, já estamos com todos os elementos necessários para tornar nosso jogo divertido. Porém, o que motivará o jogador a querer jogar nosso jogo? Qual será o objetivo do jogo?

Lembre-se de que o objetivo é passar pelo maior número possível de canos! Entretanto, ainda não temos uma forma de contar e exibir na tela a quantidade de canos vencidos.

Vamos resolver isso?

6.1 CONTAGEM DOS CANOS VENCIDOS

Novamente, vamos tomar o cuidado de separar as responsabilidades, e criaremos uma classe para centralizar a contagem dos canos. Podemos fazer essa quantidade de canos vencidos ser a própria pontuação do jogador. Assim, chamaremos essa classe de `Pontuacao` e ela ficará no pacote `br.com.casadocodigo.jumper.elementos`:

```
public class Pontuacao {

}
```

Tínhamos combinado que essa pontuação seria exibida na tela para o jogador, mas onde manipulamos os elementos que são exibidos na tela? Lá na classe `Game`!

Então, vamos instanciar essa `Pontuacao` no nosso método `inicializaElementos`:

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

    private Pontuacao pontuacao;
    // outros atributos...

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

    //outros métodos...
}
```

Os pontos aumentam cada vez que um cano sai da tela, demonstrando a vitória do pássaro sobre os canos. Se conseguíssemos determinar quando um cano sai da tela, seria fácil aumentar a pontuação do jogador. Para nossa sorte, já temos um método na própria classe `Cano` que sabe se ele saiu da tela: o método `saiuDaTela`!

Porém, tão importante quanto termos esse método, é sabermos onde ele é chamado! Sempre que nossos canos eram movidos, precisávamos saber se algum cano saiu da tela para criarmos canos novos. Então, o `saiuDaTela` é chamado no método `move` da classe `Canos`, dê uma olhada no código que **já temos**:

```
public class Canos {

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

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

            if (cano.saiuDaTela()) {

                //Aqui podemos aumentar a pontuação do jogador!

                iterator.remove();
                Cano outroCano = new Cano(this.tela,
                        maiorPosicao() + DISTANCIA_ENTRE_CANOS);
                iterator.add(outroCano);
            }
        }
    }
}
```

Vamos então, se um cano sair da tela, chamar um método `aumenta`!

```
public class Canos {

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

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

            if (cano.saiuDaTela()) {

                //Esse método ainda não existe. Já vamos criá-lo!
                this.pontuacao.aumenta();

                iterator.remove();
                Cano outroCano = new Cano(this.tela,
                        maiorPosicao() + DISTANCIA_ENTRE_CANOS);
                iterator.add(outroCano);
            }
        }
    }
}
```

Aumentar a pontuação dentro da classe `Canos` é uma boa ideia, porém nosso código está quebrando por dois motivos:

1. A classe `Canos` não sabe quem é essa variável `pontuacao` ;
2. Não temos o método `aumenta` na classe `Pontuacao` .

Vamos resolver o primeiro problema!

Levando o objeto Pontuacao para a classe Canos

Como a classe `Canos` deve ter acesso à pontuação, podemos passá-la no construtor durante a inicialização dos nossos elementos:

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

  private Pontuacao pontuacao;

  private void inicializaElementos() {
    Bitmap back = BitmapFactory.decodeResource(getResources(),
                        R.drawable.background);

    this.background = Bitmap.createScaledBitmap(back,
                back.getWidth(), this.tela.getAltura(), false);

    this.passaro = new Passaro(this.tela);

    //Instanciamos a pontuação e passamos para os Canos:
    this.pontuacao = new Pontuacao();
    this.canos = new Canos(tela, pontuacao);
  }
}
```

E setar a variável no construtor da classe `Canos` :

```
public class Canos {

    private final Pontuacao pontuacao;
```

```
    public Canos(Tela tela, Pontuacao pontuacao) {
        this.tela = tela;
        this.pontuacao = pontuacao;

        int posicao = 200;

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

Perfeito! Agora que a classe `Canos` sabe quem é essa pontuação, podemos resolver o segundo problema que tínhamos: criar o método `aumenta`!

Implementando o método aumenta na classe Pontuacao

Na classe `Pontuacao`, o método aumenta simplesmente somará `1` aos pontos atuais do jogador:

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

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

Pronto. Quando o cano sai da tela, notificamos a pontuação de que ela deve aumentar em `1`! Agora, vamos nos dedicar a outro detalhe: como exibiremos essa pontuação?

6.2 EXIBIÇÃO NA TELA

Assim como fizemos com todos os outros elementos que

queríamos exibir na tela, vamos implementar o método `desenhaNo` com o objetivo de desenhar um texto chamando o método `drawText` do `canvas`:

```
public class Pontuacao {

    private int pontos = 0;

    public void aumenta() {
        this.pontos++;
    }

    public void desenhaNo(Canvas canvas) {
        canvas.drawText(??, ??, ??, ??);
    }
}
```

O método `drawText` recebe quatro argumentos:

- O texto a ser escrito;
- A coordenada X onde esse texto deverá ser posicionado;
- A coordenada Y da posição do texto;
- A cor do texto.

Para desenhar um texto na posição `(100,100)` da nossa tela na cor branca, teríamos a seguinte chamada:

```
public class Pontuacao {

    private static final Paint BRANCO =
                                Cores.getCorDaPontuacao();
    private int pontos = 0;

    public void aumenta() {
        this.pontos++;
    }

    public void desenhaNo(Canvas canvas) {
        canvas.drawText(String.valueOf(this.pontos),
```

```
                        100, 100, BRANCO);
    }
}
```

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

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

E pedir para desenhar no loop principal do jogo:

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

    private Pontuacao pontuacao;

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

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

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

            this.pontuacao.desenhaNo(canvas);

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

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

Ufa! Fizemos bastante coisa agora. Vamos rodar nossa aplicação e ver como está?

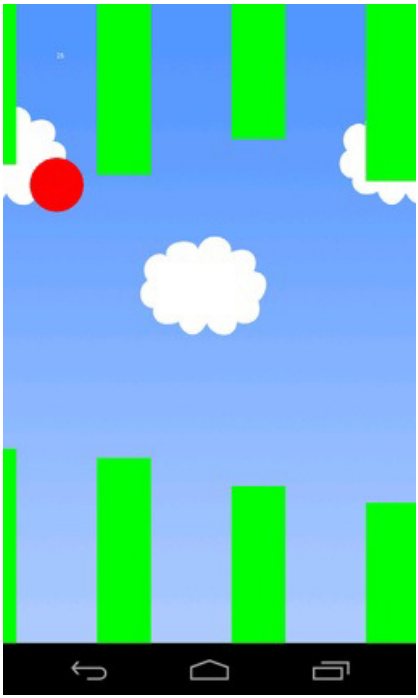


Figura 6.1: Será que ficou legal para o jogador?

A pontuação é exibida, mas ficou **muito** pequena. É bastante difícil (até mesmo na figura anterior) enxergar o número no canto superior esquerdo da tela.

Que tal colocarmos uma fonte maior e com *bold* (negrito)?

6.3 CONFIGURAÇÕES DA FONTE: A CLASSE PAINT

Até o momento, usamos a classe `Paint` para definir as cores dos nossos objetos. Porém, esse objeto `Paint` permite configurar diversas características relacionadas ao elemento que estamos

desenhando na tela. Agora, vamos definir seu tamanho e estilo usando os métodos `setTextSize` e `setTypeface`, respectivamente!

```
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.

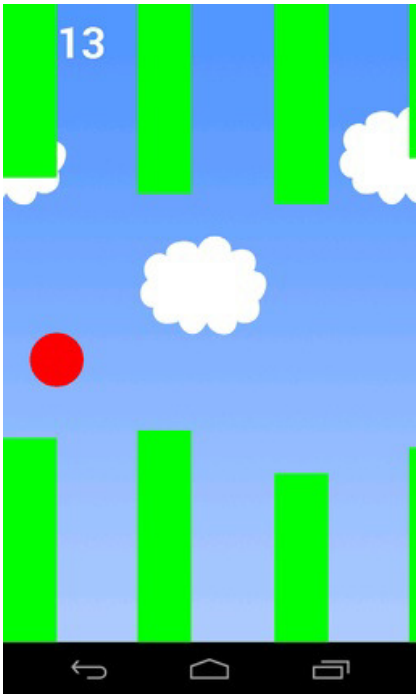


Figura 6.2: Aumentando um pouco a fonte, ficou bem melhor, não?

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` da classe `Paint`:

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

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

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

Temos de 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 de 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;
}
```

Agora, ao rodar o jogo, dá pra notar uma sombra na

pontuação:

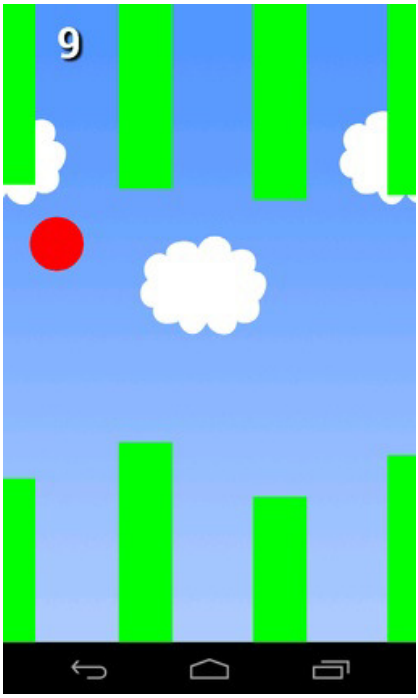


Figura 6.3: Pontuação com sombra para um maior destaque

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

6.4 ORGANIZANDO AS CAMADAS DE DESENHO

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

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

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

            canvas.drawBitmap(this.background, 0, 0, null);

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

            this.pontuacao.desenhaNo(canvas);

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

            //Unlock...
        }
    }
}
```

Neste código, nossa `pontuacao` é desenhada no canvas **antes** dos `canos`. Sempre que algo é desenhado em um canvas, uma nova camada de desenho é criada sobre as camadas anteriores. Então, no nosso código estamos dizendo ao canvas desenhar nossa tela na seguinte ordem de camadas:

1. Plano de fundo;
2. Pássaro;
3. Pontuação;
4. Canos.

Como o plano de fundo é a primeira camada a ser desenhada, ele fica por trás de todas as outras, como o esperado! Porém, estamos desenhando a pontuação **antes** dos canos, fazendo com que os canos passem **sobre** a pontuação!

Para a pontuação ficar **sempre** visível ao jogador, basta deixá-la

acima de todas as outras camadas. Vamos desenhá-la por último no nosso canvas:

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

        canvas.drawBitmap(this.background, 0, 0, null);

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

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

        this.pontuacao.desenhaNo(canvas);

        //Unlock...
    }
}
```

Agora, temos a seguinte ordem de camadas:

1. Plano de fundo;
2. Pássaro;
3. Canos;
4. Pontuação.

Por isso, a pontuação agora ficará sempre em cima dos canos. Perfeito!

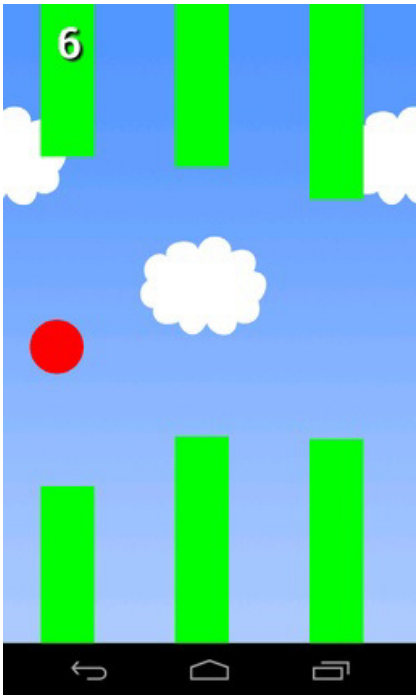


Figura 6.4: A pontuação é exibida por cima dos canos

Capítulo 7

COLISÕES

Agora que conseguimos contabilizar a pontuação do jogador, precisamos de uma forma de fazer o jogo terminar; caso contrário, podemos fazer uma pontuação infinita sem esforço algum. Sempre que o pássaro bater em qualquer cano, faremos nosso jogo terminar.

Como descobrir que um pássaro encostou no cano? Como nosso `Passaro` é um círculo e o `Cano` um retângulo, precisamos descobrir se o círculo tem alguma intersecção com o retângulo.

7.1 VERIFICANDO COLISÃO ENTRE O PÁSSARO E O CANO

Caso nosso `Passaro` colida com algum `Cano`, podemos simplesmente cancelar o *loop* principal. Como temos uma variável chamada `estaRodando` que controla o *loop*, para cancelar o jogo, basta setar `estaRodando` para `false`. Melhor ainda: podemos chamar o método `cancela`!

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

    private VerificadorDeColisao verificadorDeColisao;

    @Override
```

```
    public void run() {
        while(this.estaRodando) {
            //lockCanvas...
            //Desenho dos elementos...

            if (this.verificadorDeColisao.temColisao()) {
                cancela();
            }

            //unlockCanvasAndPost...
        }
    }
}
```

Podemos inicializar essa variável `verificadorDeColisao` no próprio método `inicializaElementos`!

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

    private VerificadorDeColisao verificadorDeColisao;

    private void inicializaElementos() {

        //...

        this.pontuacao = new Pontuacao();
        this.canos = new Canos(this.tela, this.pontuacao);

        this.verificadorDeColisao =
                        new VerificadorDeColisao(passaro, canos);
    }
}
```

Como ainda não temos a classe `VerificadorDeColisao`, vamos criá-la no pacote `br.com.casadocodigo.jumper.engine`. Além disso, já vamos passar o `Passaro` e os `Canos` no seu construtor:

```
public class VerificadorDeColisao {

    private final Passaro passaro;
```

```
    private final Canos canos;

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

Com a classe `VerificadorDeColisao` criada, vamos implementar o método `temColisao`, cuja função será percorrer todos os canos e perguntar a eles se bateram no pássaro:

```
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);
    }
}
```

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 class Canos {

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

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

```
public class Canos {

    public boolean temColisaoCom(Passaro passaro) {
        for (Cano cano : this.canos) {
            //Aqui vamos ver se tem colisão...
        }
        return false;
    }
}
```

Agora precisamos descobrir se o `Passaro` bateu com o cano atual. Para isso, verificaremos se ele cruzou verticalmente **e** horizontalmente com o cano:

```
public class Canos {

    public boolean temColisaoCom(Passaro passaro) {
        for (Cano cano : this.canos) {
            if (cano.cruzouHorizontalmenteComPassaro()
                    && cano.cruzouVerticalmenteCom(passaro)) {
                return true;
            }
        }
        return false;
    }
}
```

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

Como temos canos inferiores e superiores, para detectarmos uma colisão vertical entre o `Passaro` e o `Cano`, é necessário fazer duas verificações. Para 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`.

```
passaro.getAltura() + Passaro.RAIO > this.alturaDoCanoInferior;
```

Relembrando

Quanto maior o valor da altura de um elemento no canvas, mais baixo na tela estará o elemento!

Então, temos esse código:

```
public class Cano {

    public boolean cruzouVerticalmenteCom(Passaro passaro) {
        return passaro.getAltura() + Passaro.RAIO >
                            this.alturaDoCanoInferior;
    }
}
```

Ainda falta verificarmos se houve cruzamento com o cano superior, mas não se preocupe que 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`.

```
passaro.getAltura() - Passaro.RAIO < this.alturaDoCanoSuperior
```

Com isso, podemos completar o método `cruzouVerticalmenteCom`:

```
public class Cano {

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

Apesar de termos finalizado o método `cruzouVerticalmenteCom`, ainda há alguns erros nele: não temos o método `getAltura`, nem a constante `RAIO` disponíveis. Vamos consertar isso!

Na classe `Passaro`, vamos criar o método `getAltura` e alterar para `public` a constante `RAIO`:

```
public class Passaro {

    //Outros métodos e atributos...

    public static final int RAIO = 50;

    public int getAltura() {
        return this.altura;
    }
}
```

Ótimo! Arrumamos o método `cruzouVerticalmenteCom`. Vamos criar agora o método `cruzouHorizontalmenteComPassaro` na classe `Cano`!

Para o pássaro cruzar horizontalmente um cano, basta a distância entre a `posicao` horizontal do cano e a posição `X` (horizontal) do pássaro ser **menor** que o seu `RAIO`! O método `cruzouHorizontalmenteComPassaro` tem esse conteúdo:

```
public class Cano {

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

Na classe `Passaro`, vamos deixar `public` a constante `X`:

```
public class Passaro {
```

```
    public static final int X = 100;

}
```

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

7.2 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* :

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

    @Override
    public void run() {
        while(this.estaRodando) {
            //lockCanvas...
            //Desenho dos elementos...

            if (this.verificadorDeColisao.temColisao()) {
                new GameOver(this.tela).desenhaNo(canvas);
                cancela();
            }

            //unlockCanvasAndPost...
        }
    }
}
```

Nossa tela de *game over* deve ser construída no pacote `br.com.casadocodigo.jumper.elementos` , e ter o método `desenhaNo` :

```
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 class 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:

```
public class GameOver {
    private static final Paint VERMELHO =
                        Cores.getCorDoGameOver();

    private final Tela tela;

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

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

```
    }
}
```

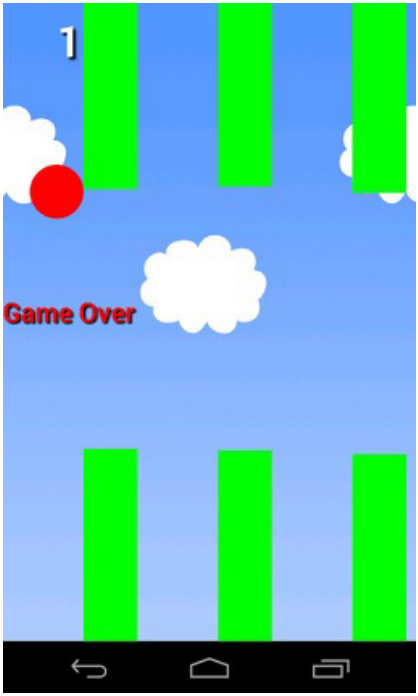


Figura 7.1: Game over exibido no jogo

O nosso `GameOver` já funciona, mas ainda precisamos centralizá-lo na horizontal. Este já passa a ser um desafio um pouco mais legal. Vamos lá?

7.3 CENTRALIZANDO UM TEXTO HORIZONTALMENTE NA TELA

Para centralizar na horizontal, é preciso 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`), começando na posição `0` dessa `String`, e terminando 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, seremos capazes de 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 acaba (`right`). Basta fazer:

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

Agora que temos o centro horizontal do retângulo `limiteDoTexto`, queremos centralizá-lo na `tela`. 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, temos todo o necessário para criar um método `centralizaTexto` na classe `GameOver`:

```
public class GameOver {

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

    //Outros métodos...

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

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

```
public class Tela {

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

Por fim, é só mandar desenhar nossa `String` no centro horizontal:

```
public class GameOver {

    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.

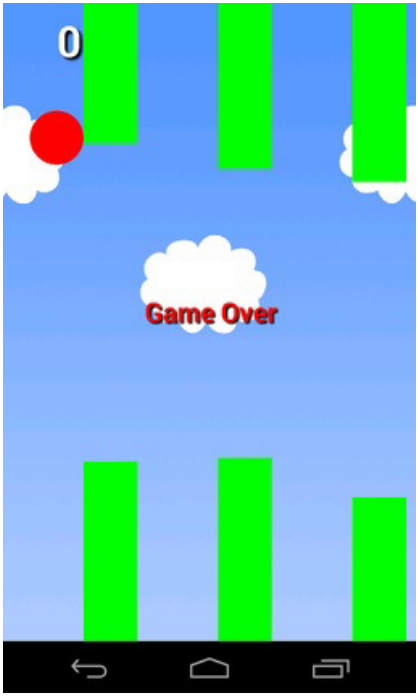


Figura 7.2: Game over é exibido centralizado na tela

Capítulo 8

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!

Arquivos das imagens

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!

8.1 SUBSTITUIÇÃO DO CÍRCULO VERMELHO

DO PÁSSARO

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, lembra?

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

Quando essa imagem for carregada, quais serão as dimensões dela? Será que é **exatamente** o mesmo tamanho do círculo vermelho?

Temos de tomar cuidado com isso, pois se a imagem for maior (ou menor), teremos problemas no tratamento das colisões.

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

```
public class Passasro {

    private final Bitmap passaro;

    public Passaro(Tela tela) {
        //atribuições...

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

        this.passaro = Bitmap
                        .createScaledBitmap(bp, RAIO*2,
                                            RAIO*2, false);
    }
```

```
}
```

> 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.

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

```
public class Passasro {

    private final Bitmap passaro;

    public Passaro(Context context, Tela tela) {
        //atribuições...

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

        this.passaro = Bitmap
                        .createScaledBitmap(bp, RAIO*2,
                                            RAIO*2, false);
    }
}
```

Agora alteramos o construtor da classe `Passaro`, então precisamos arrumar a classe que instancia o nosso `Passaro`! Qual classe será? A classe `Game`!

No método `inicializaElementos` da classe `Game`, vamos passar um `Context` para nosso `Passaro`:

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

```
    private void inicializaElementos() {
        //outras inicializações...

        this.passaro = new Passaro(this.context, this.tela);
    }
}
```

Mas isso não é o suficiente para corrigirmos o problema. Temos de dizer de onde vem esse `context`. Vamos pegá-lo do próprio construtor da classe `Game`:

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

    private Context context;

    public Game(Context context) {
        super(context);
        this.context = context;

        //outros códigos...
    }

    private void inicializaElementos() {
        //outras inicializações...

        this.passaro = new Passaro(this.context, this.tela);
    }
}
```

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

8.2 REPOSICIONANDO O BITMAP

Ao lidar com círculos na classe `Passaro`, no método `desenhaNo`, usamos o seu centro como referência de

posicionamento. Dissemos que seu centro estaria na posição `(X, altura)`:

```
public class Passaro {

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

Porém, ao lidar com imagens (*bitmaps*), a referência para desenho é o **canto superior esquerdo**. Logo, se mantivermos o ponto `(X, altura)` para seu desenho, teremos uma imagem desalinhada com a posição original.

```
public class Passaro {

    public void desenhaNo(Canvas canvas) {
        canvas.drawBitmap(this.passaro, X, this.altura, null);
    }
}
```

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:

```
public class Passaro {

    public void desenhaNo(Canvas canvas) {
        canvas.drawBitmap(this.passaro, X - RAIO,
                                this.altura - RAIO, null);
    }
}
```

Veja como ficará nossa aplicação:

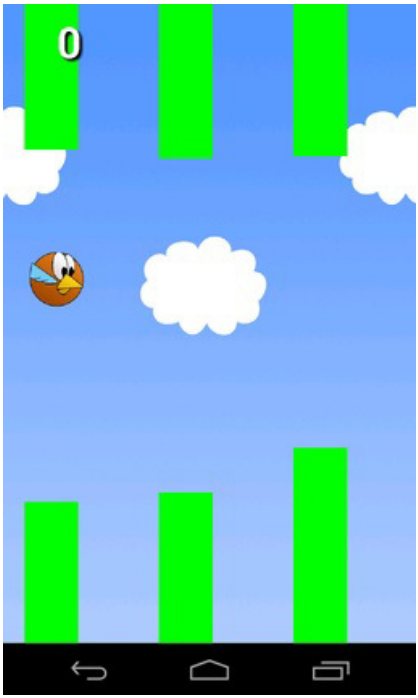


Figura 8.1: Nosso pássaro ficou bem melhor, não?

8.3 SUBSTITUINDO OS RETÂNGULOS INFERIORES POR BITMAPS

Como faremos para trocar nossos retângulos verdes por imagens se nossos canos possuem alturas variadas, calculadas aleatoriamente? Se tivéssemos uma imagem para cada altura de cano, nosso jogo ficaria muito pesado e seria bem complicado gerenciar todas essas imagens no nosso código!

Em vez de possuir uma imagem para **todo** o cano, uma ideia para resolver esse problema é termos apenas um único "filete" (uma imagem com apenas `1` pixel de altura), que será esticado

para o mesmo tamanho dos retângulos iniciais!

Assim como feito no `Passaro`, na classe `Cano`, usaremos o `BitmapFactory` para pegar a imagem, então, precisaremos de um `Context` em seu construtor:

```
public class Cano {

    public Cano(Context context, Tela tela, int posicao) {
        //Outros atributos

        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`:

```
public class Cano {

   private Bitmap canoInferior;

   public Cano(Context context, Tela tela, int posicao) {
      //Outros atributos

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

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

Agora, para desenhar esse `canoInferior` na tela, devemos alterar o `desenhaCanoInferiorNo`. Atualmente, temos o seguinte código na classe `Cano`:

```
public class Cano {

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

Lembre-se de que os dois primeiros valores do `drawRect` representam o **canto superior esquerdo** do retângulo. Por uma incrível coincidência, quando usamos o `drawBitmap`, precisamos dos valores do canto superior esquerdo também! Ou seja, ao contrário do `Passaro`, no qual tivemos de fazer uma pequena correção, para os `Cano`s já temos essas coordenadas prontas!

Com isso, nosso método `desenhaCanoInferiorNo` fica assim:

```
public class Cano {

    private void desenhaCanoInferiorNo(Canvas canvas) {
        canvas.drawBitmap(this.canoInferior,
                          this.posicao,
                          this.alturaDoCanoInferior,
                          null);
    }
}
```

Ao rodar o projeto, temos um erro de compilação! Como alteramos o construtor da classe `Cano` para receber um `Context`, precisamos passá-lo ao instanciar um `Cano`.

Recebendo um Context

Vamos à classe `Canos` receber um `Context` no seu construtor:

```
public class Canos {

  private Context context;

  public Canos(Context context, Tela tela, Pontuacao pontuacao){
    this.context = context;

    //Outros códigos...
  }
}
```

Agora, basta passá-lo à instância de `Cano`:

```
public class Canos {

  private Context context;

  public Canos(Context context, Tela tela, Pontuacao pontuacao){
    this.context = context;

    //Outras inicializações...

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

E na instância que fazemos no método `move`:

```
public class Canos {

    private Context context;

    public void move() {
        //Códigos anteriores...
```

```
        Cano outroCano = new Cano(this.context, this.tela,
                maiorPosicao() + DISTANCIA_ENTRE_CANOS);
        iterator.add(outroCano);
    }
}
```

Por fim, basta olharmos a classe `Game` e passarmos um objeto `Context` para a classe `Canos`:

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

    public void inicializaElementos() {

        //Outras inicializações...

        this.canos = new Canos(this.context,
                                this.tela, this.pontuacao);
    }
}
```

Ufa! Com os problemas corrigidos, ao rodar o Jumper teremos uma cara parecida com essa:

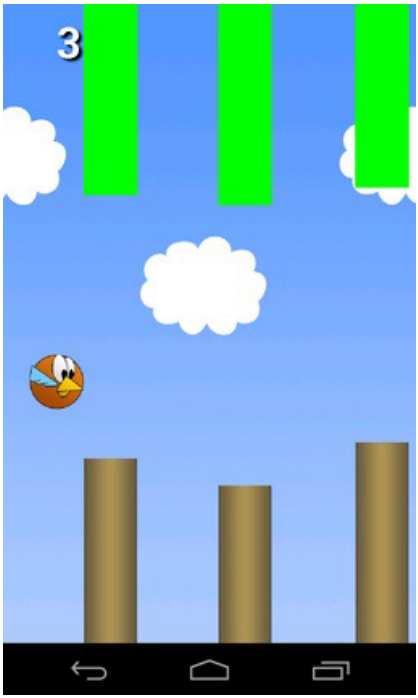


Figura 8.2: Canos inferiores agora são bitmaps também

Que tal cuidarmos dos canos superiores?

8.4 SUBSTITUINDO OS RETÂNGULOS SUPERIORES POR BITMAPS

Para os canos superiores, vamos fazer da mesma forma, porém não precisaremos usar o `BitmapFactory` novamente, basta redimensioná-lo de acordo com as medidas dos retângulos superiores:

```
public class Cano {

    private Bitmap canoSuperior;
```

```
    public Cano(Context context, Tela tela, int posicao) {

        //códigos anteriores...

        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. Veja como está nosso código atualmente:

```
public class Cano {

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

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

```
public class Cano {

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

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

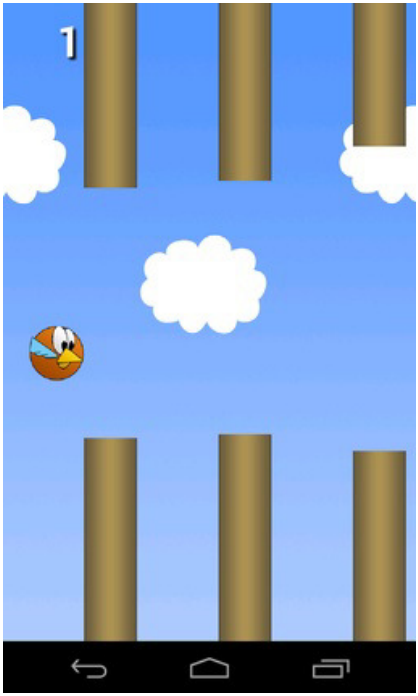


Figura 8.3: Aparência do Jumper com os bitmaps

Apesar da beleza, ainda falta um outro ingrediente bastante importante: o som! Até o momento, quando pulamos, colidimos ou há um aumento na pontuação, nenhum *feedback* sonoro é emitido, tornando a experiência incompleta para o jogador. Vamos melhorar isso!

CAPÍTULO 9

SOM

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!

9.1 A CLASSE SOUNDPOOL

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 (pulo do pássaro, colisão e aumento da pontuação), setaremos esse valor para `3`:

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

Mas onde ficará esse código? Vamos criar uma classe responsável por gerenciar esse `SoundPool`. No pacote `br.com.casadocodigo.jumper.engine`, criaremos a classe `Som`:

```
public class Som {

    private SoundPool soundPool;

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

Usando o SoundPool.Builder

A partir do Android 5.0 (API 21), há uma forma diferente de instanciar um `SoundPool`, usando um *builder*. Em vez de passarmos os argumentos diretamente no construtor do `SoundPool`, temos métodos que tornam tudo mais semântico. Veja como ficaria nosso código usando esse *builder*:

```
SoundPool.Builder builder = new SoundPool.Builder();
   builder.setMaxStreams(3);
   soundPool = builder.build();
```

Caso rodemos o `SoundPool` em versões superiores ao Android 5.0, basta fazermos essa simples alteração.

Para tocarmos algum som, antes precisaremos carregá-lo com o método `load`. Nele, passaremos um `context` e 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. Vamos guardar esse `id`:

```
public class Som {

    private SoundPool soundPool;
    public static int PULO;

    public Som(Context context) {
        this.soundPool = new SoundPool(3,
                                       AudioManager.STREAM_MUSIC,
                                       0);

        PULO = this.soundPool.load(context, R.raw.pulo, 1);
    }
```

```
}
```

Agora que temos o som `PULO` carregado, vamos tocá-lo com o método `play`, que recebe seis argumentos:

- O `id` do som;
- O volume esquerdo;
- O volume direito;
- A prioridade do som;
- O loop (`1` para loop infinito, `0` para tocar uma única vez);
- A velocidade do som (lembre-se: 1 é a velocidade normal).

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

Vamos criar um método `toca` na nossa classe `Som`, onde chamaremos o `play` do `soundPool`:

```
public class Som {

    private SoundPool soundPool;
    public static int PULO;

    public Som(Context context) {
        this.soundPool = new SoundPool(3,
                                       AudioManager.STREAM_MUSIC,
                                       0);

        PULO = this.soundPool.load(context, R.raw.pulo, 1);
    }

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

}
```

Temos nossa classe pronta, mas ainda não sabemos quando

usá-la. Em qual momento tocaremos o som do pulo do pássaro? Quando o `Passaro` pular!

9.2 TOCANDO UM SOM NO PULO DO PÁSSARO

Em qual lugar do nosso código temos a lógica do pulo? No método `pula` da classe `Passaro` ! Lá chamaremos o método `toca` :

```
public class Passaro {

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

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

```
public class Passaro {

    private Som som;

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

        //Outras atribuições...
    }
}
```

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,
                                        View.OnTouchListener {
```

```
  //Outras variáveis...
  private Som som;

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

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

  //Outros métodos...
}
```

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

9.3 SOM DA PONTUAÇÃO

Vamos continuar a colocar o som no nosso jogo. Dessa vez, faremos o som da pontuação! De uma forma bastante semelhante à feita com o `Passaro`, vamos carregar um som na classe `Som`:

```
public class Som {

    public static int PONTUACAO;

    public Som(Context context) {
        //Carregamento do som do pulo...

        PONTUACAO =
            this.soundPool.load(context, R.raw.pontos, 1);
    }
}
```

Só precisamos chamar o método `toca` no local em que o

jogador ganha pontos! Vamos olhar nossa classe `Pontuacao`:

```
public class Pontuacao {

    //outros métodos e atributos...

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

Esse método `aumenta` é chamado sempre que o jogador ganha pontos. Vamos alterá-lo para tocar o som também!

```
public class Pontuacao {

    //outros métodos e atributos...

    public void aumenta() {
        this.som.toca(Som.PONTUACAO);
        this.pontos++;
    }
}
```

Basta agora recebermos o `Som` no seu construtor:

```
public class Pontuacao {

    //outros métodos e atributos...
    private Som som;

    public Pontuacao(Som som) {
        this.som = som
    }

    public void aumenta() {
        this.som.toca(Som.PONTUACAO);
        this.pontos++;
    }
}
```

Como alteramos o construtor da classe `Pontuacao`, estamos com um erro de compilação no nosso código na classe `Game`.

Vamos arrumar isso.

No método `inicializaElementos`, basta passarmos a instância de `Som` que já temos:

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

    private void inicializaElementos() {

        //Outras inicializações...
        this.pontuacao = new Pontuacao(this.som);
    }
}
```

Ao rodar o Jumper, temos um som legal quando ganhamos pontos! Faça o teste!

9.4 SOM DA COLISÃO

Por fim, vamos dar som à colisão do pássaro! Como sempre nos preocupamos em separar as responsabilidades no nosso código, torna-se bastante fácil implementar uma nova funcionalidade, sem precisarmos alterar diversas classes. Podemos seguir a mesma ideia que usamos para o som do pássaro e da pontuação: na classe `Som`, vamos carregar o som da colisão.

```
public class Som {

    public static int COLISAO;

    public Som(Context context) {
        //Carregamento do som do pulo e da pontuacao...

        COLISAO = this.soundPool.load(context, R.raw.colisao, 1);
    }
}
```

Basta agora chamarmos o método `toca` onde quisermos tocar o som da colisão. Vamos fazer isso na classe `Game`!

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

    //...
    @Override
    public void run() {
        while(this.estaRodando) {
            //códigos anteriores...

            if (this.verificadorDeColisao.temColisao()) {
                this.som.toca(Som.COLISAO);
                new GameOver(this.tela).desenhaNo(canvas);
                cancela();
            }

            //unlockCanvasAndPost...
        }
    }
}
```

Com esse código, se houver uma colisão, tocaremos o som, mostraremos o *game over* e finalizaremos o jogo! Com poucas linhas de código, fomos capazes de adicionar som ao nosso jogo. Faça o teste para ver como ficou!

Capítulo 10

FÍSICA

Com nosso jogo terminado, vamos torná-lo ainda mais interessante refinando o comportamento do pássaro. Até o momento, quando pulamos, apenas deslocamos o pássaro para cima em uma quantidade fixa de *pixels*, dando um efeito estranho de teletransporte para os jogadores mais atentos. Da mesma forma, quando o pássaro cai, apenas alteramos sua posição em um valor fixo, tornando essa queda bem fictícia.

Será que podemos deixar essa movimentação mais realista usando um pouco de física?

Para a maioria das pessoas, o primeiro contato com física se dá no colégio, e é geralmente lá que começa uma batalha para entender as diversas fórmulas e as sopas de letrinhas, de forma que siglas como MU, MUV, MHS se tornam verdadeiros terrores. No fim, usamos esse conhecimento apenas para passar em provas. No entanto, nas *engines* ou *frameworks de games*, esses conhecimentos são aplicados em diversos jogos para garantir um efeito realista!

Fique tranquilo: não há a necessidade de ser um gênio em física para conseguirmos alguns efeitos bem interessantes na jogabilidade do Jumper. Vamos ver como deixar a queda do pássaro mais fiel ao mundo real!

10.1 MODELANDO A FÍSICA DA QUEDA

De forma simplificada, sempre que um objeto real cai, além da sua velocidade inicial, temos também a ação da gravidade sobre ele. É só imaginar uma bola caindo: a cada intervalo de tempo a gravidade faz com que esse corpo vá cada vez mais rápido para baixo. Esse comportamento é descrito na física como *Movimento Uniformemente Variado* (ou MUV)!

Saber que a queda do pássaro deve ser descrita como um movimento uniformemente variado já nos ajuda muito na programação, pois as fórmulas não necessitam de uma matemática elaborada para serem calculadas.

Vamos olhar como está o método `cai` da classe `Passaro` :

```
public class Passaro {

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

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

Sempre que esse método é chamado, alteramos a altura do pássaro. Então, agora nosso objetivo será determinar essa altura seguindo um movimento uniformemente variado. Para nossa sorte, uma das fórmulas do MUV é a que determina a altura em função do tempo, gravidade e velocidade inicial de queda:

```
S = S0 + V0 * t + (g * (t * t)) / 2
```

Onde:

- `S` = altura final;
- `S0` = altura inicial;
- `V0` = velocidade inicial;
- `g` = gravidade;
- `t` = tempo.

Após um pulo, nosso pássaro começa a cair quando sua velocidade se torna zero. Aí ele fica sujeito à ação da gravidade, configurando uma queda-livre. Então, da nossa fórmula inicial, podemos simplificá-la ao deixar a velocidade inicial igual a zero:

```
S = S0 + V0 * t + (g * (t * t)) / 2

S = S0 + 0 * t + (g * (t * t)) / 2   //V0 = 0;

S = S0 + 0 + (g * (t * t)) / 2

S = S0 + (g * (t * t)) / 2
```

Pronto! Essa será a fórmula da queda do pássaro! Só precisamos inseri-la no contexto do nosso jogo!

Como `S` representa a altura e `S0` a altura inicial, no nosso código podemos fazer assim:

```
S = S0 + (g * (t * t)) / 2

S += (g * (t * t)) / 2
```

Lembre-se de que `g` representa o valor da gravidade, que costuma ser 9.8. Vamos simplificar e deixar `g = 10` :

```
S += (g * (t * t)) / 2

S += (10 * (t * t)) /2
```

Por fim, vamos trocar `S` pela variável `altura` , e melhorar a legibilidade da fórmula:

```
    S += (10 * (t * t)) /2

    this.altura += (10 * (tempo * tempo)) / 2;
```

Pronto! Temos a queda do pássaro seguindo um movimento uniformemente variado (MUV)! Vamos aplicá-la no método `cai` da classe `Passaro`:

```
public class Passaro {

  public void cai() {
    double novaAltura = ((10 * (tempo * tempo)) / 2.0);

    boolean chegouNoChao = this.altura + RAIO > tela.getAltura();

    if(!chegouNoChao) {
      this.altura += novaAltura;
    }
  }
}
```

Só precisamos saber como passaremos esse `tempo` para nossa classe `Passaro`.

10.2 A CLASSE TEMPO

Esse `tempo` representa o tempo atual do nosso jogo, então podemos fazer algo como:

```
public class Passaro {

  private Tempo tempo;

  public void cai() {
    double tempo = this.tempo.atual();
    double novaAltura = ((10 * (this.tempo * this.tempo)) / 2.0);

    boolean chegouNoChao = this.altura + RAIO > tela.getAltura();

    if(!chegouNoChao) {
```

```
      this.altura += novaAltura;
    }
  }
}
```

E popular esse `tempo` no construtor da classe `Passaro`:

```
public class Passaro {

  private Tempo tempo;

  public Passaro(Context context, Tela tela,
                                    Som som, Tempo tempo) {

    //Outras atribuições...
    this.tempo = tempo;
  }

  public void cai() {
    double tempo = this.tempo.atual();
    double novaAltura = ((10 * (this.tempo * this.tempo)) / 2.0);

    boolean chegouNoChao = this.altura + RAIO > tela.getAltura();

    if(!chegouNoChao) {
      this.altura += novaAltura;
    }
  }
}
```

Corrigimos o problema da variável `tempo` na nossa fórmula. Agora só precisamos saber como será essa classe `Tempo` e esse método `atual`. Vamos criar a classe `Tempo` no pacote `br.com.casadocodigo.jumper.engine` e seu método chamado `atual`:

```
public class Tempo {

    private double atual;

    public double atual() {
        return atual;
```

```
    }
}
```

Pronto! Só precisamos saber onde instanciaremos essa classe `Tempo` para passarmos para o construtor do `Passaro`.

Instanciando a classe Tempo

Como instanciamos a classe `Passaro` na classe `Game`, no método `inicializaElementos` vamos instanciar também a classe `Tempo`:

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

  private Tempo tempo;

  private void inicializaElementos() {
    this.tempo = new Tempo();

    this.passaro = new Passaro(this.context, this.tela,
                                        this.som, this.tempo);

    //Outras inicializações...
  }
}
```

Com isso, arrumamos nosso código. Porém, ao rodar o jogo, veremos que nosso pássaro não será mais capaz de cair. Faça o teste. Por que será que isso aconteceu?

Perceba que o pássaro dever cair conforme o tempo passa. Entretanto, em nenhum momento dissemos para o tempo passar. Vamos fazer isso?

10.3 PASSANDO O TEMPO

Na classe `Tempo`, vamos criar o método `passa`, que deve alterar o valor do atributo `atual`:

```
public class Tempo {

    private double atual;

    public void passa() {
        atual += 0.1;
    }

    public double atual() {
        return atual;
    }
}
```

Mas em qual momento chamaremos o método `passa`? Qual lugar do nosso código deve dizer para o tempo passar?

No *loop* principal da classe `Game`!

Então, logo após o `lockCanvas`, vamos fazer o tempo passar:

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

    public void run() {
        while(this.estaRodando) {
            //LockCanvas...

            this.tempo.passa();

            //Desenho dos elementos...
            //UnlockCanvas...
        }
    }
}
```

Agora, ao rodar o código, podemos ver o pássaro caindo de uma forma mais realista. Perceba que ele começa a cair lentamente e sua velocidade aumenta ao longo da queda! Vamos fazer algo

parecido quando o pássaro pular!

> **Usando outros valores**
>
> Teste valores diferentes para o método `passa` e veja como a queda do pássaro é influenciada!

10.4 FÍSICA NO PULO DO PÁSSARO

Surpreendentemente, ao modelar a física da queda, ficamos muito próximos de ter um pulo mais realista também. Primeiramente, no método `pula`, não precisaremos mais deslocar o pássaro 150 pixels para cima, removendo a seguinte linha:

```
this.altura -= 150;
```

Deixando o método com a seguinte cara:

```
public class Passaro {

    public void pula() {
        if(this.altura > RAIO) {
            this.som.toca(Som.PULO);
        }
    }
}
```

Anteriormente, sempre que o método `pula` era chamado, fazíamos um "teletransporte" do pássaro para cima. Agora, vamos reiniciar nosso tempo sempre que esse método for chamado:

```
public class Passaro {

    public void pula() {
        if(this.altura > RAIO) {
```

```
            this.som.toca(Som.PULO);
            this.tempo.reinicia();
        }
    }
}
```

Ele deve ter a seguinte cara:

```
public class Tempo {

    private double atual;

    public void reinicia() {
        atual = 0;
    }
}
```

Com isso, ao rodar o código e tocarmos na tela, faremos nosso pássaro parar no ar. Faça o teste! É verdade que ele não pula, mas também não cai enquanto tocarmos na tela. Vamos fazer o pássaro ser "lançado" para cima ao tocarmos na tela.

10.5 REFATORAÇÃO DO MÉTODO CAI

O método `cai` agora é o responsável pela queda do pássaro de acordo com um movimento uniformemente variado. Porém, o interessante é que podemos complementar aquela fórmula para fazer o pássaro pular com um deslocamento constante para cima!

Quando o tempo for zero (ou seja, quando tocarmos na tela), vamos adicionar um valor constante para a `novaAltura` :

```
public class Passaro {

  public static final int DESLOCAMENTO_DO_PULO = 20;

  public void cai() {
    double tempo = this.tempo.atual();
    double novaAltura =
```

```
          -DESLOCAMENTO_DO_PULO + ((10 * (tempo * tempo)) / 2.0);

    boolean chegouNoChao = this.altura + RAIO > tela.getAltura();

    if(!chegouNoChao) {
      this.altura += novaAltura;
    }
  }
}
```

Com isso, ao tocar na tela, nosso pássaro será deslocado para cima ao longo do tempo até que sua velocidade se torne zero e ele comece a cair. Rode o jogo e veja como ficou!

> **Testando valores diferentes**
>
> Após rodar o Jumper, pode ser que o pássaro suba muito rápido e torne o jogo bastante difícil, causando até um Game Over. Se esse for o seu caso, basta ajustar o `DESLOCAMENTO_DO_PULO` para um valor menor. Faça testes para ver como valores diferentes do `DESLOCAMENTO_DO_PULO` altera o comportamento do pulo!

Após essa alteração no código, mudamos a semântica do método `cai`. Veja que agora ele controla o voo em geral do pássaro. Então, vamos **alterar seu nome** para `voa`:

```
public class Passaro {

  public static final int DESLOCAMENTO_DO_PULO = 20;

  public void voa() {
    double tempo = this.tempo.atual();
    double novaAltura =
          -DESLOCAMENTO_DO_PULO + ((10 * (tempo * tempo)) / 2.0);
```

```
    boolean chegouNoChao = this.altura + RAIO > tela.getAltura();

    if(!chegouNoChao) {
      this.altura += novaAltura;
    }
  }
}
```

E, consequentemente, **alterar** sua chamada na classe `Game`:

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

    public void run() {
        while(this.estaRodando) {
            //LockCanvas...

            //Desenho dos elementos...
            this.passaro.desenhaNo(canvas);
            this.passaro.voa();

            //UnlockCanvas...
        }
    }
}
```

Rode o jogo e veja que tudo continuará funcionando! Agora temos o voo do pássaro seguindo uma física mais realista!

CAPÍTULO 11

MENU PRINCIPAL

Agora que já temos um jogo bem mais divertido e com um maior realismo no movimento do pássaro, podemos nos dedicar a criar os elementos extras do jogo, ou seja, aquelas partes que não têm a ver com o loop principal do game, mas o deixa com um melhor acabamento.

Além da tela onde o jogo é executado de fato, há diversas outras, sendo a tela de menu principal uma das mais importantes. Vamos criá-la?

11.1 UMA NOVA TELA AO JOGO

Para criarmos nossa tela de menu principal, precisaremos de uma nova `Activity`, em que diremos qual será o layout e o comportamento do menu. Para isso, no Android Studio, criaremos uma nova classe chamada `MenuPrincipalActivity` no pacote `br.com.casadocodigo.jumper`, e já implementaremos seu método `onCreate`:

```
public class MenuPrincipalActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
    }
```

```
}
```

Como este é o menu principal do Jumper, precisamos dizer que essa *activity* deverá ser a primeira tela a aparecer para o jogador. Faremos isso alterando o `AndroidManifest.xml` :

```
<activity
    android:name=".MenuPrincipalActivity"
    android:screenOrientation="portrait">
    <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

        <category
            android:name="android.intent.category.LAUNCHER"/>
    </intent-filter>
</activity>
```

Veja que, anteriormente, o atributo `android:name` da tag `<activity>` possuía o valor `.MainActivity` . Esta foi a única alteração que precisamos fazer!

Como todas as activities devem estar declaradas no `AndroidManifest` , vamos registrar nossa `MainActivity` apenas como uma activity comum, deixando nosso arquivo com essa cara:

```
<manifest ... >

    <application ... >

        <activity
            android:name=".MenuPrincipalActivity"
            android:screenOrientation="portrait">
            <intent-filter>
                ...
            </intent-filter>
        </activity>

        <activity
            android:name=".MainActivity"
            android:screenOrientation="portrait"/>
```

```
    </application>

</manifest>
```

Pronto! Rode a aplicação e veja que, ao abri-la, será exibida nossa `MenuPrincipalActivity` ! Porém, qual é o layout dessa tela? Vamos melhorar isso!

11.2 LAYOUT DO MENU PRINCIPAL

Ao rodar nosso jogo, uma tela preta é exibida, pois ainda não definimos o layout dessa activity. Vamos criar um novo arquivo de layout na pasta `res/layout` , chamado `activity_menu_principal.xml` , que possuirá um componente centralizado que, ao ser clicado, iniciará nosso jogo!

```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="@drawable/background"
    android:gravity="center"
    android:orientation="vertical">

    <TextView
        android:id="@+id/menu_principal_jogar"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:shadowColor="#000000"
        android:shadowDx="3"
        android:shadowDy="3"
        android:shadowRadius="2"
        android:text="Jogar"
        android:textColor="#FF0000"
        android:textSize="35sp"
        android:textStyle="bold" />
</LinearLayout>
```

Agora, basta chamarmos esse XML na Activity usando o

`setContentView`:

```
public class MenuPrincipalActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_menu_principal);
    }
}
```

Ao rodar nosso jogo, teremos uma tela parecida com essa:

Figura 11.1: Menu principal do Jumper

Tente clicar no item "*Jogar*". O que acontece?

Comportamento do item Jogar

Ao clicar nele, nada acontece, pois ainda não dissemos qual será o comportamento desse componente de tela. Para resolver esse problema, precisaremos manipular a classe que gerencia esse layout: a `MenuPrincipalActivity`.

Para podermos configurar sua ação, primeiramente precisaremos capturá-lo no nosso código. Faremos isso com o método `findViewById`:

```
public class MenuPrincipalActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_menu_principal);

        TextView jogar =
            (TextView) findViewById(R.id.menu_principal_jogar);
    }
}
```

Agora, ao clicar nesse `TextView`, queremos iniciar nosso jogo. Vamos permitir que esse `TextView` reaja a um evento de clique adicionando um *listener* nele, mais especificamente, um *listener* de clique, um `OnClickListener`:

```
public class MenuPrincipalActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_menu_principal);

        TextView jogar =
            (TextView) findViewById(R.id.menu_principal_jogar);

        jogar.setOnClickListener(new View.OnClickListener() {
```

```
            @Override
            public void onClick(View view) {

                // O que faremos ao clicar em Jogar?

            }
        });
    }
}
```

Implementamos um `OnClickListener`, mas ainda não dissemos o que será feito ao clicarmos em jogar. No nosso caso, quando o jogador clicar nesse `TextView`, vamos abrir nossa activity do jogo, nossa `MainActivity`.

Para abrirmos uma activity, o Android possui uma classe chamada `Intent` pronta para uso, onde só precisamos dizer em qual tela estamos e para qual tela queremos ir, no seguinte esquema:

```
Intent proximaActivity =
            new Intent(activityAtual, ProximaActivity.class);

startActivity(proximaActivity);
```

Com isso, como estamos na `MenuPrincipalActivity` e queremos ir para a `MainActivity`, teremos o seguinte código:

```
public class MenuPrincipalActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_menu_principal);

        TextView jogar =
            (TextView) findViewById(R.id.menu_principal_jogar);

        jogar.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {
```

```
                Intent main =
                    new Intent(MenuPrincipalActivity.this,
                                MainActivity.class);
                startActivity(main);
            }
        });
    }
}
```

Pronto! Rode o Jumper e veja que é possível ir para a tela do jogo ao clicar em "*Jogar*"! Temos um menu principal!

Capítulo 12

CONSIDERAÇÕES FINAIS

Ao longo deste livro, criamos um jogo do zero usando somente as classes nativas do Android e mostramos que é possível fazer uma aplicação divertida sem o uso de bibliotecas específicas de games.

Não se engane imaginando que este conhecimento será usado apenas em jogos, pois as aplicações que vimos neste livro vai muito além! Podemos fazer telas customizadas em outras aplicações fora do universo dos jogos.

Como aprendemos a desenhar bitmaps na tela, podemos criar os nossos próprios componentes customizados também. Vimos como manipular sons, e isso pode ser usado em qualquer outra aplicação! As possibilidades são muito grandes.

Como continuar os estudos?

Neste instante, um mundo de ideias de projetos aparece na nossa cabeça. Aproveite esse momento de animação e comece a fazer o seu próprio jogo! Não se preocupe em fazer o novo jogo que revolucionará o mundo, nem mais bonito. Faça um que seja o mais divertido para você! Divirta-se programando tanto quanto jogando.

Durante essa jornada, haverá muitos problemas que não foram cobertos neste livro, mas haverá também muito aprendizado a cada *stacktrace* entendida e a cada *exception* resolvida. Supere cada um desses obstáculos como fazemos em um jogo e, no fim, você terá o seu próprio game!

Leia a documentação do Android em http://developer.android.com, converse com amigos, participe de grupos de discussão e fóruns como o GUJ (http://guj.com.br) e, até mesmo, o fórum deste livro: http://forum.casadocodigo.com.br. Pergunte, troque informações, e exponha suas ideias de jogos!

Estou ansioso para ver o seu jogo! Vamos lá?